CS 305 Data Structures

Spring 2020

Course Handouts

Author: Dr. Tammy VanDeGrift

Name: _____

If found, call/email:_____

CS 305 Calendar Spring 2020

Calendar subject to change – see Moodle for latest updates. Complete assigned reading prior to lecture. GNU = GNU C Programming Tutorial; Kal = Data Structures in C; CP = coursepack

Week	Monday	Wednesday	Friday
Jan 13/15/17	C: variables, functions, printing, scanning, control flow **Reading**: GNU pp. 1 – 43, CP pp. 1 - 8	C: pointers **Reading**: GNU pp. 45 – 69, Kal 2.1 – 2.8, CP pp. 14 – 28; Watch video about pointers	Lab 1: intro to C and tools **DUE: Prelab 1** **DUE: HW 0, Java review** Watch video about memory
Jan 20/22/24	**MLK Jr. DAY – No Class**	C: arrays, strings, malloc, free **Reading**: GNU pp. 89 – 107, Kal 3.3, CP pp. 29 - 41	Lab 2: pointers, arrays, and memory **DUE: Lab 1, Prelab 2**
Jan 27/29/31	C: structs **Reading**: GNU pp. 189 – 207, Kal 1.1 – 1.10, CP pp. 42 - 49	C: file I/O C: preprocessor/libraries (video) **Reading**: GNU pp. 71 – 88, 109 – 150, CP pp. 51 – 63; Watch video about passing pointers	Lab 3: structs, file I/O and C preprocessor **DUE: Lab 2, Prelab 3**
Feb 3/5/7	C: Style, Review **Reading**: GNU pp. 219 – 230, CP pp. 64 – 69, **DUE: HW 1**	**EXAM 1**	Lab 4: make and debugging **DUE: Lab 3, Prelab 4** Watch video on makefiles
Feb 10/12/14	Binary search, Recursion, Big O analysis **Reading**: CP pp. 71 - 80	Big O analysis **Reading:** CP pp. 81 - 87	Lab 5: recursion **DUE: Lab 4, Prelab 5**
Feb 17/19/21	Linked lists **Reading**: Kal 3.1 – 3.13, CP pp. 88 - 99	Linked lists **Reading**: Kal 3.14 **DUE: HW 2**	Lab 6: linked lists **DUE: Lab 5, Prelab 6**
Feb 24/26/28	Stacks **Reading**: Kal 4.1 – 4.4, CP pp. 100 - 107	Queues **Reading**: Kal 4.6, CP pp. 108 - 115	Lab 7: stacks and queues **DUE: Lab 6, Prelab 7**
Mar 2/4/6	**SPRING BREAK – NO CLASS**		
Mar 9/11/13	Lab 7: stacks and queues (continued)	Review **Reading:** CP pp. 116 - 121 **DUE: HW 3 and Lab 7**	**EXAM 2**
Mar 16/18/20	Trees **Reading**: Kal 5.1 – 5.5, CP pp. 122 - 130	Binary search trees **Reading**: Kal 5.6 – 5.7, CP pp. 131 - 140	Binary search trees **Reading**: Kal 5.9 – 5.13, CP pp. 141 - 150
Mar 23/25/27	Lab 8: trees **DUE: Prelab 8**	Sorting: selection/insertion **Reading**: Kal 6.1 – 6.3, CP pp. 151 – 156, **DUE: HW 4**	Sorting: quicksort/merge sort **Reading**: Kal 6.8 – 6.9, CP pp. 157 - 164
Mar 30/Apr 1/3	Lab 9: sorting **DUE: Lab 8, Prelab 9**	Graphs **Reading**: Kal 7.1 – 7.5, 7.7, CP pp. 165 – 181	Graphs **Reading**: Kal 7.6, 7.8, 7.10, CP pp. 182 - 197
Apr 6/8/10	Graphs Lab 10: graphs **DUE: Lab 9, Prelab 10**	Lab 10: graphs (continued) **DUE: HW 5**	**GOOD FRIDAY – NO CLASS**
Apr 13/15/17	**EASTER MONDAY – NO CLASS**	Review **Reading**: CP pp. 198 – 204 **DUE: Lab 10**	**EXAM 3**
Apr 20/22/24	Hash functions, Hash tables **Reading**: Kal 8.1 – 8.3, CP pp. 206 – 213; Watch video on hashing	Hash tables, Review of course **Reading**: CP pp. 214 – 230 **DUE: HW 6**	Lab 11: course evaluations and hashing **DUE: Prelab 11**
FINALS WEEK Tuesday, Apr 28, 8 – 10am	**FINAL EXAM (COMPREHENSIVE), Tuesday 8 – 10am** **DUE: Lab 11 (8am)** time set by registrar and cannot be moved; make your travel arrangements accordingly		

Page left intentionally blank

CS 305: Data Structures
Spring 2020

Course Information

Instructor: Dr. Tammy VanDeGrift

Email: vandegri@up.edu
Office Phone: 503-943-7256
Office: Shiley 223
Website: Course information on Moodle, learning.up.edu

Meetings: MWF 1:35 – 2:30 pm
Classroom: Shiley 249
Office Hours: Tentatively: M 11:25-12:20pm; 2:45 – 4pm, T 9:45-noon, W 8:15 – 9am, F 2:40-3:40pm

Bulletin Description: Continues the study of computer science and software engineering methodologies with the C programming language. Analysis of common data structures, time and space efficiency, stacks, queues, linked lists, trees, graphs, hash tables, recursion, searching, and sorting algorithms. (Prerequisites: CS 203 with a grade of C- or better.)

Student Outcomes

At the end of the course, students should be able to:

- Utilize the C programming language using good programming practices.
- Compile, debug, and execute C programs from the command line.
- Implement fundamental data structures in C, such as linked lists, stacks, queues, trees, and graphs.
- Implement iterative and recursive algorithms in C, such as sorting algorithms, tree traversals, and graph search algorithms.
- Identify and evaluate the Big-O complexity analysis of algorithms.

These outcomes will be accomplished by:

- Completing lab and homework assignments
- Participating in class discussion and group activities through regular class attendance
- Seeking help of the professor, tutors, and classmates when necessary
- Communicating ideas orally and in writing
- Providing help rather than giving answers/code to classmates seeking help

Course Philosophy

General: This course is designed to introduce concepts related to the C programming language, data structures, and algorithms utilizing those data structures. ***Because this course covers topics that build on one another, it is critical that you keep up with the material by completing assignments on time and preparing for each class session by reading and viewing assigned material.*** It's okay to struggle with the concepts. I expect students to be challenged, but it is your responsibility to seek help when you are confused. Many students find the material challenging and it stretches their minds to think creatively and differently. You will **not** succeed in this course if you start homework assignments the day before they are due.

Seeking Help: I expect you to have questions as you learn the course material. You may receive help from classmates (see below about Collaborative Learning), help from the fellows, help from the course tutors, and seek help from the instructor. I encourage you to ask questions during lecture meetings and attend office hours.

University of Portland's Code of Academic Integrity: Academic integrity is openness and honesty in all scholarly endeavors. The University of Portland is a scholarly community dedicated to the discovery, investigation, and dissemination of truth, and to the development of the whole person. Membership in this community is a privilege, requiring each person to practice academic integrity at its highest level, while expecting and promoting the same in others. Breaches of academic integrity will not be tolerated and will be addressed by the community with all due gravity. See University Bulletin for policy.

Collaborative Learning: Your classmates are a huge resource available to you. Because we understand material in different ways, I encourage you to discuss concepts from the course with your classmates and peers who have already taken the course, but any **prelabs, labs, and homework that you turn in must be your own code and your own writing**. Unacknowledged copying or using parts of someone else's work (peers or online), even if it has been modified by you, is plagiarism and is not acceptable. When you get help, you must acknowledge places where you received help in your code comments (tutors, books, websites, classmates). **Never show classmates your homework code**. **Do not ask classmates to see their homework code. It is ok to show your code to tutors, fellows, and the professor.** An acceptable way to collaborate is to discuss problems and potential solutions and then writing the solutions and code on your own. When giving help to classmates, do not give them the answer. Instead, ask questions to learn of their understanding and give conceptual explanations - this practice will help you master the material yourself. Remember: you must turn in work that is your own and conceived from your own brain, you must acknowledge the people who helped you, and you are encouraged to seek help when you are confused. Many class sessions will be used to work on programming problems using active and collaborative learning.

Instructor's and Students' Responsibilities: In this course, the instructor's job is to guide you in learning about C and data structures. In additional to some traditional lecturing, I will have regular discussions, labs, and group activities during lectures. I expect your full participation, preparation, and readiness to learn at all class meetings by reading assigned material in advance of lecture. In return, I will do my best to offer suggestions, activities, and explanations to help you learn the material.

Classroom Conduct: The Shiley School of Engineering is committed to developing and actively protecting a classroom environment in which respect must be shown to everyone in order to facilitate and encourage the expression, testing, understanding and creation of a variety of ideas and opinions. Failure to meet these standards will result in removal from the class session. Making mistakes is a necessary part of learning – be patient with yourself and be patient with others throughout the learning process.

In order to maintain a positive learning environment, students should avoid disruptive behaviors such as: not muting cell phones and other electronics, receiving cell phone calls during class, leaving class early or coming to class habitually late, talking out of turn, doing assignments for other classes during lecture, reading the newspaper, sleeping, and engaging in other activities that detract from the classroom learning experience. The classroom is a professional meeting space.

This class takes place in a classroom designed for active learning and includes workstations in the room. Students should use computers for activities related to the course when instructed by the professor.

Computers are not to be used for email, web surfing, social media, watching videos, playing games, and other personal entertainment. Do not unplug computer cables. Respect the computer equipment (no food and drink spills).

The Learning Commons

Trained peer tutors and writing assistants in the Learning Commons, located in Buckley Center 163, work with you to facilitate your active learning and mastery of skills and knowledge. For questions about the Learning Commons, please send all correspondence to Jeffrey White, Administrator, at white@up.edu. The Learning Commons is a program of the Shepard Academic Resource Center (SARC.)

Math Resource Center: Appointment-based tutoring is available through our online scheduler at www.bit.ly/up_mrc. Walk-in tutoring Sundays through Thursdays evenings. For MTH 141, request appointments at math141@up.edu. The course-specific schedule can be found at www.up.edu/learningcommons, or the reception desk in BC 163.

Writing Assistance: Brainstorming ideas for your paper, create an outline, work on citations, or review a draft with a Writing Assistant. Visit www.up.edu/learningcommons to access our Writing Center schedule.

The Language Studio: Contact the language assistance hotlines to schedule a time to meet throughout the semester at chinesetutor@up.edu, frenchtutor@up.edu, germantutor@up.edu, or spanishtutor@up.edu.

Natural Sciences Center: Send your tutoring requests to biotutor@up.edu, chemtutor@up.edu, or physicstutor@up.edu.

Speech & Presentation Lab: Improve your presentations by requesting an appointment at speech@up.edu.

Group Work Lab: Make an appointment for your group project at groupwork@up.edu.

Nursing Tutoring: Tutoring is available for pathophysiology, BIO205, anatomy and physiology, and other nursing courses on a walk-in or appointment basis. Up-to-date schedule information is at www.up.edu/learningcommons/nursing.

Economics and Business Tutoring: For support in economics, OTM, finance, accounting, and business law courses, send requests for appointments to your discipline's tutor email hotline: econtutor@up.edu, otmtutor@up.edu, financetutor@up.edu, accountingtutor@up.edu, or bizlaw@up.edu.

Shiley Sophomore Fellows: Provides tutoring in several sophomore engineering classes. To make an appointment, send a request to stepUP@up.edu.

Learning Assistance Counselor: Learning assistance counseling is also available in BC 163. The counselor teaches learning strategies and skills that enable students to become more successful in their studies and future professions. The counselor provides strategies to assist students with reading and comprehension, note-taking and study, time management, test-taking, and learning and remembering. Appointments can be made in the on-line scheduler available to all students in Moodle or during posted drop-in hours.

Assessment of Learning

Methods, Activities, and Evaluation Tools

I will assess your learning based on your submitted work, including homework assignments, labs, exams, and in-class activities. Generally, the assignments, labs, and in-class activities are your chance to learn, while the exams are the main way I know you have learned the material. I expect you to submit your homework by the due date and time. If circumstances arise (e.g. you are ill for an extended period of

time, you are out of town for a university-related activity) that prevent you from submitting your work on time, please discuss the reason with me before the due date.

Method for Determining Final Grade

Percent Weight	Category	Description
30	Midterm Exams	Three exams @10% each, completed **individually**. Exams are intended to assess students' mastery and understanding of the course material.
15	Final Exam	One comprehensive final exam, completed **individually.**
25	Homework assignments	Programming assignments, completed **individually.** There are 7 (one in Java for review, six in C) programming assignments to give your practice with coding, implementing, and using data structures. Each assignment includes a short written reflection.
20	Prelabs & Labs	Pre-labs should be completed **individually**. Labs are completed in **pairs**. You will work with a different partner for each lab to get to know several other students. Labs are intended to give you practice programming and using data structures.
10	Professionalism, Quizzes, and In-class Activities	Attendance and active engagement in class sessions; professional behavior demonstrated toward peers and classroom environment; if a student misses class, it is the student's responsibility to get notes from a classmate; some sessions may include quizzes and some may include activities that are submitted for credit; this grade is **individual**

Grading Standards

Course grades will be assigned based on the total points you earn during the semester, weighted accordingly to the categories shown below. The minimum cutoffs for grades will not change. I do reserve the right to raise your grade, but the following minimum percentages are guaranteed. (For example, if you earn 90% of the points, you will get an A-. If you earn 89% of the points, you earn a B+ but I reserve the right to raise your grade to an A-.)

- >= 93% A
- >= 90% A-
- >= 87% B+
- >= 83% B
- >= 80% B-
- >= 77% C+
- >= 73% C
- >= 70% C-
- >= 67% D+
- >= 63% D
- >= 60% D-
- < 60% F

Performance Criteria
Grades of A- and A show excellent mastery of the material and skill in creating proofs and communication. Grades of B-, B, and B+ show good understanding and skills. Grades of C-, C and C+ show adequate understanding and skills. Grades D+ and below demonstrate inadequate understanding and inadequate skills development in C and data structures.

Late Assignments: You are granted **two full days** to use at your discretion for submitting late pre-labs, labs or homework without penalty. For example, you may choose to turn in two different assignments 24 hours after their due dates and times. You could choose to use the two days (48 hours) on a single assignment. Weekends count as regular days. If you use a late day, indicate the use of late day(s) with a note on the assignment when you submit your work. For labs, both partners will be charged late days. If you need to submit an assignment late (due to illness, death in the family, out of town for a university event) and you have already used your two late days, please contact the instructor before the due date and time to discuss your options. Late days may **not** be used for exams, in-class activities, or quizzes.

Logistics

Textbook and Readings: The required textbook for the course can be found in the bookstore: *Data Structures in C* by Noel Kalicharan [ISBN: 9781438253275]

A second textbook, freely available, is also required: *The GNU C Programming* Tutorial by Mark Burgess and Ron Hale-Evans. http://www.crasseux.com/books/ctut.pdf. It is up to you if you want to print this book and put it in a binder for your convenience.

The course lecture notes are provided as a pdf on moodle and provided in hard copy form.

You should read certain chapters or sections *before* attending the accompanying class session (see the online course calendar for the latest updates to the readings). Lectures are intended to supplement the textbook and GNU Tutorial and the instructor expects that you have read the assigned material.

Recommended books:

- *C Pocket Reference, edition 2* by Peter Prinz and Ulla Kirch-Prinz [ISBN: 9780596004361]
- *The C Programming Language,* 2nd edition by Brian W. Kernighan and Dennis M. Ritchie [ISBN: 978-0131103627]

Course Calendar and Website: The latest version of the calendar is on Moodle. The course calendar lists the lecture topics, assigned readings, exams, and due dates for assignments. The calendar is subject to change as the semester progresses. Course material will be posted to Moodle. All programming homework assignment should be submitted to Moodle.

Accessibility Statement: The University of Portland endeavors to make its courses and services fully accessible to all students. Students are encouraged to discuss with their instructors what might be most helpful in enabling them to meet the learning goals of the course. Students who experience a disability are also encouraged to use the services of the Office for Accessible Education Services (AES), located in the Shepard Academic Resource Center (503-943-8985). If you have an AES Accommodation Plan, you should make an appointment to meet with your faculty member to discuss how to implement your plan in this class. Requests for alternate location for exams and/or extended exam time should, where possible, be made two weeks in advance of an exam, and must be made at least one week in advance of

an exam. Also, you should meet with your faculty member to discuss emergency medical information or how best to ensure your safe evacuation from the building in case of fire or other emergency.

Non-Violence Statement: The University of Portland is committed to fostering a community free from all forms of violence in which all members feel safe and respected. Violence of any kind, and in particular acts of power-based personal violence, are inconsistent with our mission. Together, we take a stand against violence. Join us in learning more about campus and community resources, UP's prevention strategy, and reporting options on the Green Dot website, www.up.edu/greendot or the Title IX website, www.up.edu/titleix.

Assessment Disclosure Statement: Student work products for this course may be used by the University for educational quality assurance purposes.

Academic Regulation Statement: Policies governing your coursework at the University of Portland can be found in the University Bulletin at www.up.edu/registrar**.**

Mental Health Statement: As a college student, you may sometimes experience problems with your mental health that interfere with academic experiences and negatively impact daily life. If you or someone you know experiences mental health challenges at UP, please contact the University of Portland Health and Counseling Center in Orrico Hall (down the hill from Franz Hall and Mehling Hall) at www.up.edu/healthcenter or at 503-943-7134. Their services are free and confidential, and if necessary they can provide same day appointments. In addition, after-hours phone counseling is available if you call 503-943-7134 and press 3 outside of business hours. Also know that the University of Portland Public Safety Department (503-943-4444) has personnel trained to respond sensitively to mental health emergencies at all hours. Remember that getting help is a smart and courageous thing to do – for yourself, for those you care about, and for those who care about you.

Ethics of Information: The University of Portland is a community dedicated to the investigation and discovery of processes for thinking ethically and encouraging the development of ethical reasoning in the formation of the whole person. Using information ethically, as an element in open and honest scholarly endeavors, involves moral reasoning to determine the right way to access, create, distribute, and employ information including: considerations of intellectual property rights, fair use, information bias, censorship, and privacy. More information can be found in the Clark Library's guide to the Ethical Use of Information at libguides.up.edu/ethicaluse.

Improving the Course: I welcome your feedback about the course at any time. I may ask for your feedback periodically and you will have the opportunity to evaluate the course at the end of the semester.

Course Handouts

Downloading Required Software

How to get the software you need to write, debug, and test C programs on your personal computer.

WINDOWS:

- You can download a free version of Mobaxterm here: https://mobaxterm.mobatek.net
  Get the home edition. If you want to pay, you can get the professional edition
- Once Mobaxterm is installed, open it. Click on the Packages icon.
  - This will bring up a window of possible packages. Search for gcc-core. Highlight it and install it.
  - After this installs, click on the Packages icon and search for emacs. Highlight it and install it.
  - After this installs, click on the Packages icon and search for gdb. Highlight it and install it.
  - After this installs, click on the Packages icon and search for ddd. Highlight it and intall it.

MAC:

- You probably already have gcc installed as part of xcode. If not, you can get it by downloading and installing xcode: https://developer.apple.com/support/xcode/
- You can get emacs for Mac here: http://emacsformacosx.com/.

LINUX:

- You probably already have gcc, gdb, and emacs installed. If not, you can get gcc here: https://gcc.gnu.org/
- Emacs for Linux is here: https://www.gnu.org/software/emacs/

Visual IDEs:

The above instructions will get you the compiler and emacs (a text editor). It is not fancy – just command-line execution. If you prefer to use a visual IDE, here are some suggestions, but these are not part of the course:

- Codebocks: http://www.codeblocks.org/
- Eclipse: https://www.eclipse.org/downloads/packages/release/2019-12/r/eclipse-ide-cc-developers
- Codelite: https://codelite.org/

You can also use the Shiley build to do your homework. To access a Shiley Windows computer from your own computer:

- Download and install the vmware client: https://desktop.up.edu/
- Or you can use the html version if you do not want to install the software.
- The name of the computer you want to connect to is: desktop.up.edu
- Log in with your UP username and password. Choose **Engineering Kiosk**. You are now logged into a Windows machine.

Unix/Linux Cribsheet

CS 376

Linux Commands

cd	change directory
chgrp	change the group of a file/directory
chmod	change permissions on file/directory
chown	change owner of file/directory (must be super user)
colrm	remove columns from a file
cp	copy a file/directory
find	find files meeting certain criteria
ln	create link (hard or symbolic)
ls	list contents of directory
mkdir	make directory
mount	list mounted devices, mount a device
mv	move a file/directory
pwd	print working directory
rm	remove a file (be careful! – file is gone)
rmdir	remove directory
touch	change file access and modification times
unmount	unmount a device

search/manipulation of files:

cat	concatenate and display files
cmp	compare two files
diff	display line-by-line differences between files
echo	echo arguments
egrep	search a file for a pattern using a full regular expression
grep	search a file for a pattern
head	display first few lines of files
less	browse or page through a text file
more	browse or page through a text file
sort	sort, merge, or sequence check text files (ASCII order)
hunspell	find spelling errors
strings	find printable strings in an object or binary file
tail	deliver the last part of a file
uniq	display adjacent repeated lines as single line
wc	display a count of lines, words, and characters in a file

formatting:

awk/gawk	pattern scanning a processing language
expand	expand TABs to spaces
fold	filter for folding lines
sed	stream editor
tee	copies stdin to specified files and stdout
tr	translate/replace characters
unexpand	unexpand spaces to TABs

system status:

chsh	change shell type for logins
du	summarize disk usage
df	display status of disk space on file systems
groups	list groups
jobs	lists active jobs
ps	process status information
time	report execution time of a command
ulimit	limits the resources (time, file space) that a process uses
uptime	show how long the system has been up
users	display a compact list of users logged in
w	who is logged in and what they are doing
who	who is on the system

miscellaneous:

at	schedule a job to run at a certain time
bg	run job in background
cal	display calendar
clear	clear screen
compress	used for compressing (note: gzip/gunzip is newer)
crontab	create scheduling table for executing jobs
date	write date and time
fg	run job in foreground
gzip/gunzip	compress/uncompress files
java	run a Java program
javac	run the Java compiler
kill	terminate process
lp	send job to printer
man	manual page for a command
nice	run command at a different priority
passwd	change password
script	make a record of a terminal session
tar	create tape/file archives (tar –cvf dat.tar . ; tar –xvf dat.tar)
uncompress	used for uncompressing by old 'compress' utility
CTRL-Z	suspend current job
CTRL-C	kill current job

*Information about other Unix commands can be found using the `man` command.

Bash Operators

>	output redirection to file (from standard output stream)
>>	output redirection, appending to file (from standard output stream)
2>	error redirection to file (from stderr)
2>>	error redirection, appending to file (from stderr)
&>	output redirection to file (from both standard output and stderr streams)
&>>	output redirection, appending to file (from both standard output and stderr streams)
<	input redirection from file
|	pipe standard output of one command to input of another example: `hunspell <myfile.txt | sort`
|&	pipe standard output and stderr of one command to input of another example: `hunspell <myfile.txt |& sort`
*	file substitution wildcard (any string) example: `ls *.c` (will list any file with .c at the end of the name)
?	file substitution wildcard (single character) example: `ls test?.c` (will list any file with testx.c where x is any character)
[...]	file substitution wildcard (any character in group) example: `ls test[1-5].c` (will list any file test1.c, test2.c, ...test5.c)
$(command)	substitute output of command into command line example: `echo "There are $(who | wc -l) users"`
;	separate commands on a single line
(... ; ...)	group commands example: `(echo "USER NAMES" ; cat userNames.txt) >users.txt`
||	conditional execution, execute second command only if first fails example: `ls *.c || ls ../*.c`
&&	conditional execution, execute second command only if first succeeds
&	run command in background example: `emacs file.txt &`
#	begins comment
$	expand value of a variable example: `echo $DISPLAY`
\	prevents special interpretation of next character (escape character) Example: `echo "\$DISPLAY=$DISPLAY"`
"...."	quote string, treated as single token (but expand variables in the string) example: `grep "This is $DISPLAY" name.txt`
'...'	quote string, treated as single token (do not expand variables) example: `grep 'This is $DISPLAY' name.txt`

Part 1: The C Programming Language

We will use the C programming language in this course. It looks a lot like Java syntax, so much of the syntax will be familiar to you. However, C does not support object-oriented programming. Instead, we will write programs that are organized into FUNCTIONS and DATA.

You should know the foundation of programming in Java from CS 203. The chart below contains items you learned about Java on the left as compared to their equivalent in C.

Java	**C**
Method	Function
Class	<no such thing in C, but can organize data and functions that are related into a single file>
Object	Struct (data only – no methods)
Constructor	<no such thing, but can define a function to return a struct with members initialized>
Variables (int, double, char, etc.)	Variables (int, double, char, int *)
For loop	For loop
While loop	While loop
Conditionals (if, then, else)	Conditionals (if, then, else)
System.out.print	printf
Scanner	scanf, getc
Arrays	Arrays
String	char *, array of chars with '\0' at the end, functions in string.h
Files	Files (fscanf, fgetc, fputc)
Numerical operations (+, -, *, /, %)	Numerical operations (+, -, *, /, %)
Comparison (==, !=, >, <, >=, <=)	Comparison (==, !=, >, <, >=, <=)
Object storage in memory	Memory addresses and pointers
import	include
public static void main(String[] args)	int main(int argc, char * argv[])
Compile with javac	Compile with gcc

We will move through C fairly quickly during the first four weeks of the course, highlighting the parts that are new. Be sure to keep up with the assigned readings and videos.

CS 305: In-class Activity 1 (Intro to C)

Complete this in your team. The recorder will write down the team's consensus answers to the questions on one sheet. We will be doing group work in this course. Engineers often work in teams, so one objective of this course is to get you more practice working with a team during lecture time. We'll be using these four roles throughout the semester and the roles will rotate for every activity.

Manager: Ensures team is on task, watches the clock, keeps team moving along, ensures all members get to speak, ensures people are fulfilling their roles

Recorder: Records discussion and team answers, ensures the team sheet is submitted at the end of activity to get credit

Presenter: Presents oral reports to the class for the group, answers for group when called upon

Strategy Analyst: Observes group dynamics, offers suggestions to improve group dynamics, may be called upon to describe how group is operating

Names: (M)_____, (R)_____, (P)_____, (S)_____

Manager: Make sure the team introduces themselves to each other. Find out what their favorite part of winter break was.

Let's jump into learning C. Since you are familiar with Java, the syntax of C should not be too surprising to you. Read through the code below to try to determine what it does.

```c
#include <stdio.h>
#include <stdlib.h>

/* This is an example of a C program
 * CS 305
 * Lecture 1, version 1
 * Author: Tammy VanDeGrift
 */

/* lucky_num
 * calculates and returns a lucky number given two integers
 */
int lucky_num(int a, int b) {
  return 50 - a + (3 * b);
}

/* str_len
 * determines the length of a string within a character array
 * by finding the first position of the null character
 */
int str_len(char *str) {
  int i = 0;
  for(i = 0; str[i] != '\0'; i++);
  return i;
}

/* main
 * greets user, prompts user for information, and prints lucky number
 */
int main(int argc, char* argv) {
  char name[50];
  int age = -1;
  int lucky_number = 0;

  printf("Welcome to CS 305!\n");
  printf("Does C look like Java?\n");
  printf("What is different?\n\n\n");

  /* acquire user's name */
  printf("What is your name?\n");
  scanf("%s", name);
  /* note: scanf can be dangerous with limited size char arrays */

  /* acquire users's age */
  while(age < 0) {
    printf("What is your age?\n");
    scanf("%d", &age);
  }

  /* determine and print lucky number */
  lucky_number = lucky_num(age, str_len(name));
  printf("Your lucky number, %s, is: %d\n", name, lucky_number);

  /* EXIT_SUCCESS is a specially defined return value to say
     everything went well */
  return EXIT_SUCCESS;
}
```

Questions:

1. What is returned by the function call `lucky_num(4, 5)`? _____

2. What is returned by the function call `lucky_num(5, 4)`? _____

3. Examine the `str_len` function. What do you think the character `'\0'` means? (This is single quote, backslash, zero, single quote.)

_____

4. What is returned when calling `str_len("Tammy")`? _____

5. In the function `str_len`, the parameter is specified as `char *str`. Make a guess as to what you think char * means.

6. How are comments delimited in C?

7. What built-in C function is used to get data from the keyboard? _____

8. What built-in C function is used to print to the screen? _____

Strategy Analyst: How is the team working together? Are there ways the team could be more effective?

9. Suppose the code compiles and is executed. Which function do you think is executed first?
    a. main
    b. lucky_num
    c. str_len

10. The code executes. Suppose the user types "Tammy" for the name and "25" for the age. What does the program print as the lucky number? _____

11. Suppose the user types -12 for their age at the prompt for age. What happens?

12. What is the type of the variable `name`? _____

13. What does your group see in the C code that is similar to Java?

(Mark up the code with * for all items that are similar to Java)

(Mark up the code with # for all items that are different than Java)

14. What questions does your group still have about the C code above?

<u>Be ready to present for whole-class discussion.</u>

15. (if time) Write a C function called `sum_digits` that takes an `int` as a parameter, returns an `int` as a parameter, and calculates the total 1 + 2 + 3 + … + N and returns the total. If the parameter N is < 0, the function should return 0.

For example, `sum_digits(5)` should return 15, since 1+2+3+4+5 = 15.

```
int sum_digits(int n) {

}
```

Reading data from the keyboard

Options: scanf, fgets, getc

Below, the code snippets each read in a string from the keyboard differently.

```
/* option 1 with scanf */
  char name[50];
  scanf("%s", name);
  /* note: scanf can be dangerous with limited size char arrays */


/* option 2 with scanf - only reads up to 49 chars */
  scanf("%49s", name);
  /* note: scanf can be dangerous with limited size char arrays */


/* option 3 with fgets - reads data into name up to size of name */
  fgets(name, sizeof(name), stdin);


/* option 4 with getc - read each char from stdin and only put up to
   NAME_SIZE number of chars into name */
  char name[NAME_SIZE];
  int i;
  int ch;
  for(i=0; i<NAME_SIZE; i++) {
    ch = getc(stdin);
    if(ch == '\n') {
      break;
    }
    /* put ch in name */
    name[i] = ch;
  }
  /* put string terminating character at position i */
  name[i] = '\0';

  /* get any remaining characters until newline and ignore them */
  if(i == NAME_SIZE) {
    /* read chars until newline, ignoring them */
    int ch = getc(stdin);
    while(ch != '\n') {
      ch = getc(stdin);
    }
  }
```

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

CS 305: printf, variables, arrays, pointers

Hello world:

```
-----------------------------------------------------
#include <stdio.h>


# hello world program
int main(void){
  printf("Hello world\n");
  return 0;
}
```

What is printed when executing this program?

```c
#include <stdio.h>
#include <stdlib.h>

/* CS 305 lecture 2, code example a
 * intro to pointers and printf conversion codes
 * author: Tammy VanDeGrift
 */

/* main
 * if you are not using command-line arguments, you can put void as the
 * parameter list for main */
int main(void) {
  int a = 16;
  char c = 'b';
  int *ap = &a;
  int *ap2 = &a;

  printf("Value of a: %d\n", a);

  *ap = *ap + 1;
  printf("Value of a: %d\n", a);

  a = a + 1;
  printf("Value of a: %d\n", a);

  *ap2 = *ap2 + 3;
  printf("Value of a: %d\n", a);

  /* data in C is treated how _you_ specify it to be treated */
  //printf("Value of a: %s\n", a); /* core dumps */
  printf("Value of a: %c\n", a);
  printf("Value of a: %f\n", a);
  printf("Value of a: %1.2f\n", a);
  printf("Value of a: %u\n", a);
  printf("Value of a: %5d\n", a);
  printf("Value of c: %c\n", c);
  printf("Value of c: %d\n", c);

  /* more fun with pointers */
  char *cp = &c;
  printf("Value of cp: %p\n", cp);
  *cp++;
  printf("Value of c: %c\n", c);
  printf("Value of *cp: %c\n", *cp);
  printf("Value of cp: %p\n", cp);

  /* What happened here with *cp++? */
  // pointer value incremented, so cp points to the memory address
  // one byte later than the memory address for cp
  // if the intention is the increment what cp points to,
  // must use (*cp)++

  /* Each time you run the program, does the pointer value remain the same? */

  /* What does picture of this look like? */

  return EXIT_SUCCESS;
}
```

Draw the picture of variables and their content here:

a [ ] c [ ]

```
#include <stdio.h>
#include <stdlib.h>

/* CS 305 lecture 2, code example b
 * author: Tammy VanDeGrift
 * more pointer and array fun -- strings and arrays
 */

/* main
 * if you are not using command-line arguments, you can put void as the
 * parameter list */
int main(void) {
  char * city = "Portland"; // constant string pointer
  char state[] = "Oregon";  // string pointer

  printf("Value of city: %s\n", city);

  //city[2] = 'a'; /*seg fault, core dumped */

  state[3] = 'Z';  //ok to assign character
  printf("Value of state: %s\n", state);

  /* print each character on own line*/
  int i=0;
  while(city[i] != '\0') {
    putc(city[i], stdout);
    putc('\n', stdout);
    i++;
  }
  putc('\n', stdout);

  /* C knows how much space was allocated for the string */
  int len = sizeof(city)/sizeof(char);

  printf("Size of city: %d\n", sizeof(city));
  printf("Size of char: %d\n", sizeof(char));
  printf("Length of array: %d\n", len);

  for(i=0;i<len;i++) {
    printf("%c", city[i]);
  }
  printf("\n");

  /* experiment with array of ints */
  /* *values points to the first element in the array */
  int values[] = {2, 24, 80, 3, 100, -5, -3};
  printf("Address of values: %p\n", values);
  printf("Contents of *values: %d\n", *values);
  printf("Address of values+1: %p\n", values+1);
  printf("Contents of *values+1: %d\n", *(values+1));

  /* using sizeof function to get size of the array */
  for(i=0; i< sizeof(values)/sizeof(*values); i++) {
    printf("%d\n", *(values+i));
  }

  return EXIT_SUCCESS;
}
```

My notes about printf, arrays, variables, and memory addresses:

CS 305: In-class Activity 2 (Pointers)

Assign roles to the team members. Write down the team's consensus answers to the questions on one sheet.

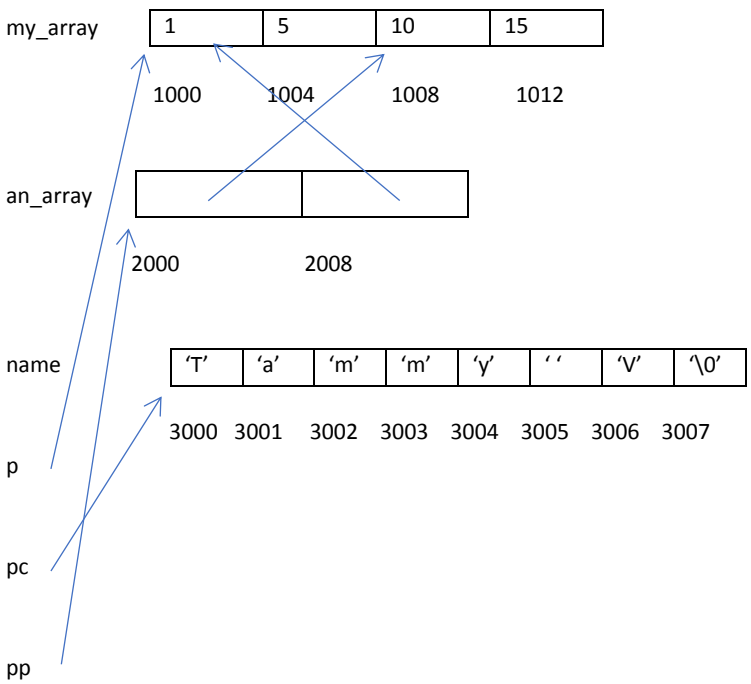
Names: (M)_____, (R)_____, (P)_____, (S)_____

Examine the C code below:

```
#include <stdio.h>
#include <stdlib.h>

/* CS 305 lecture 2
 * author: Tammy VanDeGrift
 * in-class exercise, pointer fun 3
 */

/* main
   determine what is printed and draw pictures to show pointers and data
   assume memory address of myArray is 1000
   assume memory address of anArray is 2000
   assume memory address of name is 3000
*/
int main(void) {
  int my_array[] = {1, 5, 10, 15};
  int *an_array[] = { &my_array[2], &my_array[0] };
  char name[] = "Tammy V";

  int *p = my_array;
  char *pc = name;
  int **pp = &an_array[0];

  (*p)++;
  printf("Value of *p: %d\n", *p);

  p++;
  printf("Value of p: %p\n", p);

  p++;
  *p = 30;
  printf("Value of my_array[2]: %d\n", my_array[2]);

  // draw picture for activity
  printf("DRAW FIRST DATA PICTURE\n\n");

  pp++;
  printf("Value of *pp: %p\n", *pp);
  printf("Value of **pp: %d\n", **pp);

  pp--;
  (*p)++;
  printf("Values in an_array: %p, %p\n", an_array[0], an_array[1]);
  printf("Deferencing values in an_array: %d, %d\n", *an_array[0], *an_array[1]);
```

```
  // draw picture for activity
  printf("DRAW SECOND DATA PICTURE\n\n");

  printf("Value of *pc: %c\n", *pc);

  (*pc)++;
  printf("Value of *pc: %c\n", *pc);

  pc++;
  printf("Value of *pc: %c\n", *pc);

  printf("DRAW THIRD DATA PICTURE\n\n");
  return EXIT_SUCCESS;
}
```

Here is a data picture showcasing the data and pointers just after they are declared. Assume my_array's data is stored starting at memory address 1000. Assume an_array's data is stored starting at memory address 2000. Assume name is stored at memory address 3000.



1. Draw the data picture when the "DRAW FIRST PICTURE HERE" is printed.

2. Draw the data picture when the "DRAW SECOND DATA PICTURE" is printed.

3. Draw the data picture when the “DRAW THIRD DATA PICTURE” is printed.

4. What is printed to the screen when this program runs?

```
Value of *p: _______
Value of p: _____________
Value of my_array[2]: ____
DRAW FIRST DATA PICTURE

Value of *pp: ___________
Value of **pp: ____
Values in an_array: _______, _______
Deferencing values in an_array: _____, _______
DRAW SECOND DATA PICTURE

Value of *pc: ________
Value of *pc: _______
Value of *pc: ________
DRAW THIRD DATA PICTURE
```

The data stored in memory after the initialization of the variables looks like this (assuming my_array starts at memory address 1000, an_array starts at 2000, name starts at 3000, p is at 4000, pc is at 4008, and pp is at 4016).

	Memory Address	Memory Contents
my_array	1000	1
	1004	5
	1008	10
	1012	15
	….	…
an_array	2000	1008
	2008	1000
	…	…
name	3000	‘T’
	3001	‘a’
	3002	‘m’
	3003	‘m’
	3004	‘y’
	3005	‘ ’
	3006	‘V’
	3007	‘\0’
	…	…
p	4000	1000
pc	4008	3000
pp	4016	2000

5. What questions does you team have about pointers and arrays?

CS 305: Run-time errors, dynamic memory allocation, arrays, and strings

Run-time errors

Now that you have had a chance to compile, (perhaps even debug), and execute a C program, let's discuss run-time errors. In Java, typically an exception is thrown when a runtime error occurs. For example, you may recall getting an array out of bounds exception, an I/O exception, a null pointer exception, or a class cast exception. That means Java is doing a lot of work at run-time to do these types of checks for the programmer.

C was built to be fast, at the cost of safety. C does not do array bounds checking with arrays. Instead of doing this check (increasing time at execution), it is up to the programmer to do this check if the programmer so wishes to check for array bounds checking. With lots of power comes lots of responsibility. That means the programmer is responsible to verify data if she so chooses. That also means that run-time errors are not as "programmer-friendly" as they are in Java.

Here are typical run-time errors that you get in C and what they mean:

Run-time error	What it most likely means
Segmentation fault (sometimes with a core dump)	The program attempts to access memory at an address that does not exist or a memory address for which the program does not have permission to access.
Bus error	The program attempts to access a word of memory (multiple bytes) on a non-exact word boundary.
Illegal instruction	The program tries to execute a bit pattern as an instruction that is not a legal instruction.
Infinite loop (not reported – program could hang, print out lots of repeated strings) Hit ctrl-C to stop a C program execution	The program is stuck in an infinite loop. Check loops and data scanned in prior to a loop (remember scanf reads from stdin as a buffer and data from a previous read could be populating the next scanf).

Memory

In both Java and C, the program variables (data) are stored in memory. You can think of this memory as one huge chunk of space, where every byte is addressable starting at address 0.

Memory address	Example Data
0	10011000
1	11111110
2	00101111
3	00000000
4	00000000
….	….

The compiler divides the memory into sections to store the program elements:

- Machine instructions (such as compare, add, function call); specific to the processor in C
- Static data (constants)
- Stack for local variables and bookkeeping information for functions
- Heap for dynamically allocated data
    - In Java, objects (when the `new` keyword is used) are stored on the heap
    - In C, malloc-ed or calloc-ed memory is on the heap
        - Note that this dynamically allocated memory stays around until the program returns it by using `free`

<u>Arrays</u>

Like Java, C supports a collection of the same type as an array. For example, the programmer can create an array of 20 ints called data with the following syntax:

```
int data[20];    // array of 20 ints
```

Likewise, can create an array of 100 chars and a 2D array (20 x 30) of doubles:

```
char name[100];             //array of 100 chars
double matrix[20][30];      //2d array, 20x30, of doubles
```

The above code will assign data to 20 consecutive 4-byte words in memory to store ints. But, the initial values stored in data are likely "junk" – whatever is leftover from that memory location. Sometimes, the initial values will be 0. Do not count on this, however!! It is ALWAYS a good idea to set the initial values of your data to something. For example, you can assign all values in data to 0 with:

```
int i;     //note that in C, the variable for indexing a loop
           //must be declared prior to the loop header
for(i=0; i<20; i++) {
     data[i] = 0;
}
```

If you know the data values of the array at the point of declaration, you may assign them using the { } notation, similar to Java:

```
int my_array[] = {1, 2, 3, 4, 5};
```

You may also specify the size in the declaration:

```
int my_array[5] = {1, 2, 3, 4, 5};
char abc[6] = {'H', 'o', 'w', 'd', 'y', '\0'};
```

Arrays can store any type, so you could create arrays that contain pointers to ints, pointers to chars, pointers to pointers to ints, etc. Later, we will see how to create a custom type in C, called a struct. If a

struct object is defined, then its type can be used for storing data items (similar to Java – creating a class called Student and creating an array of Student objects, `Student names[]`).

Arrays and Pointers

In C, arrays do not have a built-in length, like the do in Java (`names.length`). Arrays are simply a contiguous set of bytes in memory. So, in C, arrays and pointers are closely related. When the array variable *name* is used, C uses the address (pointer) of the 0th element of the array *name* to find its location in memory.

In order to get the length of an array called `data`, you can use:

```
int len = sizeof(data) / sizeof(*data);
```

`*data` follows the pointer and is the value of `data[0]`.

When passing arrays as parameters to functions and the function needs to know the length of the array, you should pass two parameters: the array itself and its length as an int. For example, if you want to define a function that finds the minimum value of an int array, the function should be defined as:

```
int min_value(int * arr, int size) {
      …
}
```

< Or >

```
int min_value(int arr[], int size) {

}
```

Dynamically Allocating Arrays

To allocate an array using malloc, use the following syntax:

```
int *x = malloc(40 * sizeof(int));
```

Malloc finds 40*4 bytes (ints are 4 bytes long) of memory on the heap to store the array x. Malloc returns the memory address (pointer to int) of the first byte of this chunk of memory and stores it in the variable x. From here, you can store and access data in the array as you would any other array.

```
x[0] = 3;
x[1] = 6;
```

etc.

Why would we need/want to dynamically allocate arrays?

- If the size of the array is not known at run-time, we would need to dynamically allocate enough space during the program's execution.
- If we want the array contents to outlive the function in which it was created. Remember, an array that is declared as int x[6] goes away at the } that encloses its scope.

Memory location of arrays (also applies to all variables in C)

Global:

x[4] is defined outside any function definition

The value inside [ ] must be a compile-time constant

The array is created before program starts execution at main and exists until program exits

Initial values are 0 or NULL

Declared as a local variable in a function:

x[4] is defined inside a function f

The value inside [ ] must be a run-time integer value

The array is created at the point of declaration and is destroyed at its enclosing right }

Be careful!! Could create a pointer to an array and leave the { } block, in which case the array is destroyed. This creates a dangling pointer.

Initial values of the array are garbage, unless explicitly initialized.

Declared on the heap, using malloc:

```
int * x = malloc(40*sizeof(int));
```

The value 40 may be a run-time integer value

The array is created when malloc function call completes. It is destroyed when the call to free this pointer is made.

Malloc-ed arrays can outlive the function call in which they were created

Initial values of the array are garbage.

Note: if calloc is used, the initial values are 0's.

Strings

In C, strings are arrays of characters with the null-terminating character '\0' at the end. One can store a string shorter than the length of the array into a character array, such as:

```
char name[50] = "Tammy";
```

One can also write:

```
char * place = "Portland";    // note: cannot update the individual values of place
place[0] = 'A';    //seg faults, cannot update the individual characters of place
(read-only), but name[0] is fine (since that is declared as a mutable array)
```

If you write:

```
printf("Length: %d\n", strlen(name));
```

this will print 5 (length of "Tammy", even though the character array has space for 50 characters).

You need to include `<string.h>` to use the `strlen` function, in addition to other useful string functions:

- strcat
- strcpy
- strlen
- strcmp

If you want to see all the functions that are available, type at the Linux shell window (or google it):

```
man string.h
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* Lecture 3
 * Tammy VanDeGrift
 * Code to demonstrate how to pass arrays to functions
 * arrays are passed as the pointer to the first element
 */

/* demonstrates the function header with int * as parameter */
int min_value(int *arr, int size) {
  int i;
  int min = INT_MAX;  //from limits.h
  for(i=0; i<size; i++) {
    if(min > arr[i]) {
      min = arr[i];
    }
  }
  return min;
}

/* demonstrates function header with int arr[] as parameter */
int min_value2(int arr[], int size) {
  int i;
  int min = INT_MAX;  //from limits.h
  for(i=0; i<size; i++) {
    if(min > arr[i]) {
      min = arr[i];
    }
  }
  return min;
}

int main(void) {
  int my_array[] = {-543, 10002, -33333333, 4, 2, 7, 500000000};
  int len = sizeof(my_array)/sizeof(*my_array);
  printf("%d\n", min_value(my_array, len));
  printf("%d\n", min_value2(my_array, len));
  return EXIT_SUCCESS;
}
```

Same type:
int * arr
int arr[]

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
  char name[50] = "Tammy";
  char *place = "Portland";

  name[0] = 'S';
  //place[0] = 'T';    //this one seg faults

  printf("Length: %d\n", strlen(name));
  return 0;
}
```

C will seg fault if you try to use [loc] notation for variables not declared as arrays.

```c
#include <stdio.h>
#include <stdlib.h>

/* Lecture 3
 * demonstrates how to pass by reference -- passing pointer to
 * variable declared in calling function
 * author: Tammy VanDeGrift
 */

/* f takes parameter x, adds 1 to x, and returns 0 */
int f(int x) {
  x = x + 1;
  return 0;
}

/* g takes parameter pointer to x, adds 1 to x, and returns 0 */
int g(int *x) {
  *x = *x + 1;
  return 0;
}

/* main tests the 2 functions to illustrate passing a pointer
 * versus passing the variable */
int main(void) {
  int a = 5;
  int b = 5;

  f(a);
  g(&b);

  printf("Value of a: %d\n", a);
  printf("Value of b: %d\n", b);
  return EXIT_SUCCESS;
}
```

What is the value of a?

What is the value of b?

CS 305: In-class Activity 3 (Arrays and Strings)

Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____, (R)_____, (P)_____, (S)_____

Examine the C code below. First, look at the part1 function.

```
#include <stdio.h>
#include <stdlib.h>

/* Lecture 3
 * Examples of using arrays
 * Tammy VanDeGrift
 */

double *create_array(int n) {
  double *arr = NULL;

  if(n < 0) {
    return NULL;
  }

  // allocate memory for array of length n doubles
  arr = malloc(sizeof(double)*n);
  // initialize array values
  int i;

  for(i=0; i<n; i++) {
    arr[i] = i;
  }
  return arr;
}

/* determine how the contents of my_array changes */
int part1(int len) {
  int my_array[len];
  int *p = my_array;

  int i;
  for(i=0; i<len; i++) {
    my_array[i] = 20*i;
  }
  // what are the contents of my_array at this point?

  *p = *p + 2;
  // what are the contents of my_array at this point?

  *(p+3) = *(p+3) + 8;
  // what are the contents of my_array at this point?

  for(i=0; i<len; i++) {
    printf("%d\t", my_array[i]);
```

```
  }
  printf("\n");
  return 0;;
}

/* determine contents of my_array */
int part2(void) {
  double * my_array = create_array(4); //note that the scope of my_array is local
                                       //to this function
  printf("my_array:\n");
  int i;
  for(i=0; i<4; i++) {
    printf("%f\t", my_array[i]);
  }
  printf("\n");
  free(my_array);
  return 0;
}

/* call both parts */
int main(void) {
  part1(6);
  part2();
  return EXIT_SUCCESS;
}
```

<u>Part 1: examine the part1 function and answer the questions below.</u>

1. Does the `part1` function take parameters? If so, how many and what is/are their type(s)?

2. Assume `part1(6)` is called.
Draw the contents of my_array at each comment in the function.

my_array (comment 1)

my_array (comment 2)

my_array (comment 3)

Part 2: Examine the code for the part2 function and the create_array function

1. What is the return type of the `create_array` function?

2. Assume `part2()` is called.
What is printed?

3. Is `my_array` in part2 stored on the stack or on the heap? Stack Heap

4. Is `my_array` in part1 stored on the stack or on the heap? Stack Heap

5. In which function is the memory for `my_array` defined in part2 allocated?

6. In which function is the memory for `my_array` defined in part2 free'ed?

Now, consider the following code that uses strings in C. The first part of the code concatenates three strings, as the line `first_name = first_name + " " + last_name` does in Java.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_NAME 50

/* Lecture
 * Tammy VanDeGrift
 * Illustrates how string functions from string.h may be
 * used
 * note: many of these functions have an n version, such
 * as strncpy (copy first n chars), strncmp (compare first n
 * characters)
 * use man <insert function name here> to learn more about
 * the string functions */

int main(void) {
  // part 1
  char first_name[MAX_NAME+1];
  char last_name[MAX_NAME+1];

  strcpy(first_name, "Tammy");
  strcpy(last_name, "VanDeGrift");

  if((strlen(first_name) + strlen(last_name) + 1) > MAX_NAME) {
    printf("Could not append strings. Exiting\n");
    return EXIT_FAILURE;
  }

  strcat(first_name, " ");
  strcat(first_name, last_name);  /* appends last name
                                 to first_name
                                 note: first_name must have
                                 enough memory space to
                                 append last_name */

  printf("Full name: %s\n", first_name);

  //part 2
  if(strcmp(first_name, "Sally") > 0) {
    printf("A\n");
  }
  if(strcmp(first_name, "Will") > 0) {
    printf("B\n");
  }
  if(strcmp(first_name, "Tammy VanDeGrift") == 0) {
    printf("C\n");
  }

  return EXIT_SUCCESS;
}
```

1. In order to use the string functions, what must be included at the top of this C program?

2. What do you think strcpy does?

3. What do you think strlen does?

4. What do you think strcat does?

5. What is printed just before the //part 2 comment?

Now consider part 2.
6. What do you think strcmp does?

7. What is printed under the //part 2 comment when the code executes?

8. What questions does your group have about arrays and/or strings?

9. (if time) Write the C code to create a 2D array of ints called table. It should be 5 x 7 in size. In the table, at row r and col c, store the value r * c. Essentially, this is creating a multiplication table.

CS 305: Data and structs

In Java, a class defines the attributes (instance variables), constructor(s), and methods of objects of that type. Essentially, a class defines a new data type in Java.

In C, new data types can be created with structs. A struct is like a class, except that it only contains the object attributes (members), the members are always public, and there are no constructors and no methods.

Defining a struct

Here is an example declaration of a struct to represent a pixel:

```
struct pixel {
	int red;		//could be chars in C, since a char stores values 0 to 255
	int green;
	int blue;
};
```

This creates a data structure that stores three ints as a single type. Note that a struct could contain heterogeneous data types, such as:

```
struct student {
	int id;
	char * name;
	double gpa;
};
```

Let's see how to declare a pixel struct:

```
struct pixel p1;
struct pixel p2;
```

Note that when a struct variable is declared, its type is struct <name of structure>.

Accessing struct member variables

Then, to access the member variables, we write:

```
p1.red = 150;
p1.green = 200;
p1.blue = 0;
```

You may also initialize structs statically:

```
struct pixel p3 = {25, 50, 200};  //note: red = 25, green = 50, blue = 200
                                  // the order is the order of the member variables
```

Typedef to save typing

You may also define a type using `typedef` in C, so you do not need to keep writing the word `struct` when declaring struct variables. If you already have the struct defined, you may do this with:

```
typedef struct pixel pixel;		// defines "struct pixel" as just "pixel"
```

OR you can do this at the time you declare the struct:

```
typedef struct pixel {
       int red;            //could be chars in C, since a char stores values 0 to 255
       int green;
       int blue;
} pixel;
```

Dynamically allocating structs and accessing member variables

You can allocate a struct dynamically by creating a *pointer to a struct* (assume from now on that the typedef has been used, so we can drop the struct keyword):

```
pixel * p4p = malloc(sizeof(pixel));
```

Now, `p4p` is a pointer to a pixel. In order to access the member variables, we use the -> notation instead of the dot notation:

```
p4p->red = 30; //this is syntactically equivalent to (*p4p).red = 30
```

Once you are finished using a struct that was dynamically allocated, you need to free it:

```
free(p4p);
```

Before freeing memory, you MUST be sure that the memory contents to which `p4p` is pointing will no longer be used and there are no other "live" pointers pointing to that memory. If you free the memory and there are other pointers that are still using it, this creates a dangling pointer and could create some amazing buggy program behavior.

Arrays of structs

Now that we can define new data types, we can declare arrays to store that data type, just as we can declare arrays of ints, arrays of doubles, arrays of chars, etc.

```
pixel picture[60][80];     //creates 2D array of pixels
```

Then, to store data into the array:

```
picture[0][0] = p1;        //assuming p1 is a pixel
picture[0][1].red = 3;     //can store individual members directly
picture[0][1].green = 45;
picture[0][1].blue = 205;
```

You may also store pointers to structs in an array:

```
pixel *row[10];            //creates array called row to store pointers to pixels
```

Boolean type

There is no built-in boolean type in C, as there is in Java. In general, 0 means false in C and any other value means true.

```
int value = 5;
int value2 = 0;
if(value) {
```

```
        //this will execute
}

if(value2) {
        //this will not execute
}
```

If you want to have access to a type bool, you may `#include <stdbool.h>`. It contains the constants true and false.

CS 305: In-class Activity 4 (Structs)

Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____, (R)_____, (P)_____, (S)_____

Examine the code below and answer the questions that follow.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_STUDENTS 10

/* enum example -- for class years */
typedef enum {FR, SO, JU, SR} class_type;

/* structures -- similar to the collection of instance variables
 * for a Java class */
typedef struct student {
  unsigned int id;
  char * name;
  double gpa;
  class_type class;
} student;

/* function to update student struct */
/* example of taking a struct and returning a struct */
student update_gpa(student s, double new_gpa) {
  s.gpa = new_gpa;
  return s;
}

/* function to update student's class year */
/* example of passing a pointer to a struct */
/* when class_inc returns, the value for s->class remains updated */
/* note: use s->class notation instead of s.class notation
 * when referring to a member of a pointer to a struct */
int class_inc(student * s) {
  if(s->class < FR || s->class > SR) {
    printf("No such class exists.\n");
    return -1;
  }
  if(s->class < SR) {
    s->class++;
    return 0;
  }
  if(s->class == SR) {
    // do nothing
    return 0;
  }
}

/* print student information */
```

```
void print_student(student s) {
  printf("ID:\t%d\n", s.id);
  printf("NAME:\t%s\n", s.name);
  printf("GPA:\t%.2f\n", s.gpa);
  printf("CLASS:\t%d\n", s.class);
}

/* prints roster */
void print_roster(student roster[], int len) {
  /* print students and calculate total GPA */
  printf("ROSTER:\n");
  int i;
  double total_gpa = 0.0;
  for(i=0; i<len; i++) {
    print_student(roster[i]);
    total_gpa += roster[i].gpa;
  }

  /* print average gpa */
  printf("Average GPA: %.2f\n", total_gpa/(double)len);
  printf("\n\n");
}

/* main function
 * creates an array of student structs
 * demonstrates how to use structs
 */
int main(void) {
  student mary = {1234, "Mary Smith", 3.05, SO};
  mary = update_gpa(mary, 3.20);
  update_gpa(mary, 4.0);
  print_student(mary);
  //Question 1: What is mary's GPA?

  student *maryp = &mary;
  int ok = class_inc(maryp);
  maryp->name = "Mary Knotting";
  print_student(*maryp);
  printf("\n");
  //Question 2: What is mary's name?

  /* create array of students */
  student roster[MAX_STUDENTS];
  int num = 4;

  /* assign values to 4 positions of roster */
  roster[0] = mary;  // copies contents of mary to roster[0]
  student george = {2222, "George Brown", 3.02, FR};
  roster[1] = george;
  student tim = {3278, "Tim Allen", 2.65, SO};
  roster[2] = tim;
  roster[2].id = 3333;
  roster[3].name = "Sheila Ko";
  roster[3].id = 5555;
  roster[3].class = SO;
```

```
  roster[3].gpa = 3.5;

  /* print roster */
  print_roster(roster, num);
  // Question 3: How many students are printed?

  /* print tim */
  print_student(tim);
  printf("\n");
  // Question 4: What is tim's ID?

  /* create array of pointers to students */
  student * rosterp[MAX_STUDENTS];
  rosterp[0] = &tim;  // creates pointer to time for rosterp[0]
  rosterp[1] = &george;
  rosterp[2] = &mary;

  print_student(*rosterp[1]);
  // Question 5: What is george's ID?

  george.id = 4000;
  printf("After updating id for George\n");
  print_student(*rosterp[1]);
  // Question 6: What is george's ID?

  // Question 7: Draw the contents of roster and rosterp

  int val = 5;
  /* no boolean type in C, any non-zero value is considered true */
  if(val) {
    printf("val is true\n");
  }
  int val2 = val - 5;
  if(val2) {
    printf("val2 is true\n");
  }
  // Question 8: What is printed from the prior two if statements?

  return EXIT_SUCCESS;
}
```

Answer these questions:

1. What does the function update_gpa return? _____

2. Suppose the function class_inc is called on a pointer to a student whose class is SR. What does class_inc return? _____

3. Consider the main function. Answer the questions embedded in the comments. Consider the code up to the point of execution of that comment.

```
//Question 1: What is mary's GPA? _____
```

```
//Question 2: What is mary's name? _______________________

//Question 3: How many students are printed? ______________

//Question 4: What is tim's ID? ___________________

//Question 5: What is george's ID? _________________

//Question 6: What is george's ID? ________________

//Question 7: Draw the contents of roster and rosterp (one has been done for you)
```

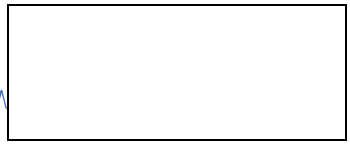
roster

			5555 “Sheila Ko” 3.5 SO	Garbage	Garbage	Garbage	Garbage	Garbage	Garbage

mary

george

tim



rosterp

			Pointer to non-init student	Pointer to non-init student	Pointer to non-init student	Pointer to non-init student	Pointer to non-init student	Pointer to non-init student	Pointer to non-init student

```
// Question 8: What is printed from the prior two if statements?
```

4. What questions does your team have about structs?

CS 305: C Preprocessor

C preprocessor: macro processor that is used automatically by the C compiler to transform your program prior to actual compilation. Macros are brief abbreviations for longer constructs.

<u>There are four main phases of the preprocessor</u>:

1. trigraph replacement (some characters are really written by 3 characters in a row – think limited keyboard keys)

for example:

`??=` translates to #
`??/` translates to \
`??'` translates to ^

Why? if using a reduced character set, can still use fewer characters which get replaced by a single character.

probably not going to affect your programs, but watch out
with gcc, only happens if –trigraph switch is used

2. Line splicing**:** Lines ending with \ are folded by deleting the \n character. So, it splices multi-line statements into one.

helpful for macro expansion, since macros must be defined "on one line", but it is easier for the programmer to read them if defined on multiple lines

3. Tokenization**:** Program is split into tokens separated by white-space characters. Comments are removed by replacement of a single space.

4. Macro expansion**:** Macros expanded, including conditional compilation (obeys the directives) and other files are included.

<u>How the preprocessor is useful</u>

1. Conditional compilation: useful for compiling with debug print statements on or off (part of lab 3)

2. Include code/definitions from other sources:

```
#include < >          // looks for file in system directory
#include " "          // looks for file in your directory
```

3. Create macros to do code replacement (see below)

4. To make sure a file does not get compiled (included) possibly infinitely many times:

```
#ifndef INCLUDE_H
      #define INCLUDE_H 1
      #include "include.h"
```

```
#endif
```

If this code above gets executed more than once, the second time the `INCLUDE_H` token should be set and will not be defined again

Macros:

Macros give the programmer the ability to define symbols on command line, conditionally (so the user-specified value will be used or a default value will be used) – will see this in lab 3.

It is simply token replacement with parameters corresponding to the tokens.

example:

```
#define min(a, b) ((a) < (b)? (a) : (b))
int x = min(4, z);
```

What happens?

The macro will be inserted, so the code will be:

```
int x = ((4) < (z)? (4) : (z));
```

!!!!!! It's important to put ( ) around things, because the substitution could make items behave with different precedence rules. !!!!!!

Can also write macros to be on multiple lines with \ at the end of each line:

```
#define do_10_times(stmt_list) \
{ int xxxyyy; \
for (xxxyyy = 0; xxxyyy < 10; xxxyyy++) { \
   stmt_list; \
} \
}
```

!!!!!!! Be careful!! It's a straight token replacement, so the variable name could conflict with the name of a variable in your program. !!!!!!!!!

So, in your program, you could write:

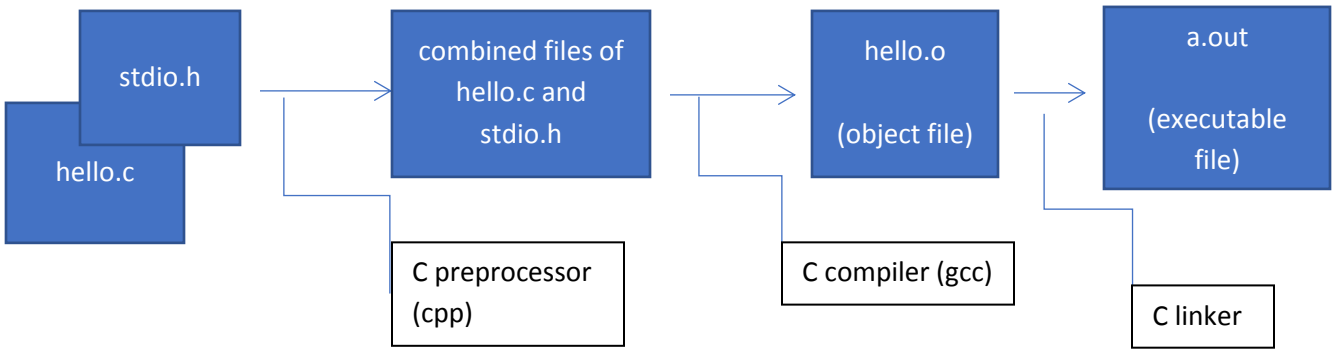
```
do_10_times(printf("Hello\n");)
```

Compiling multiple files together:

If you have a program that has code across main.c, processor.c and mutator.c, then you can compile:

```
gcc -o my_prog main.c processor.c mutator.c
```

In general, here is the compilation process:



```
#include <stdio.h>
#include <stdlib.h>

/* example of using preprocessor macros
 * CS 305
 * Lecture 5
 * Tammy VanDeGrift */

#ifndef FAV_NUM
   #define FAV_NUM 20
#endif

#define min(a,b) ((a)<(b)? (a) : (b))

#define do_10_times(stmt_list) \
  { int xxxyyy; \
  for (xxxyyy = 0; xxxyyy < 10; xxxyyy++) { \
  stmt_list; \
  } \
  }

int main(void) {
  int x = min(4, 15);
  printf("Value of x: %d\n", x);
  do_10_times(printf("Howdy!\n"));
  printf("Value of favorite number: %d\n", FAV_NUM);

  return EXIT_SUCCESS;
}
```

My notes about the C preprocessor:

CS 305: Libraries and File I/O

Common C libraries that you might use in CS 305:

Library Name	Example functions
stdio.h	Core input/output functions, such as fput, fgetc, fopen, printf, scanf, etc.
stdlib.h	Standard library definitions, such as EXIT_FAILURE, EXIT_SUCCESS, NULL, atoi, atof, atoll, free, malloc, rand
ctype.h	Character types, such as isalpha, isascii, isdigit, tolower, toupper
stdbool.h	Boolean type and values, such as bool, true, false
math.h	Mathematical declarations, such as isfinite, isinf, isgreater, isless, M_E (value of e), M_PI (value of pi), MAXFLOAT, INFINITY, acos, asin, atan, ceil, cos, exp, fabs, floor, log, log10, log2, round
string.h	String operations, such as memcpy, strcat, strcmp, strcat, strlen, strncmp
limits.h	Implementation-defined constants, such as INT_MAX, CHAR_MIN, INIT_MIN,

You may learn more about a library's definitions by typing:

```
man stdio.h
```

You include libraries in your code by using:

```
#include <stdio.h>
```

File I/O: Just as in Java, your C programs can read from and write to files.

File Writing

<u>Method 1</u>: redirect `stdout` to a file (this is handy if you do not want all your code printing to go to the screen!!!! ☺)

Suppose you have a program called `my_prog` as an executable. Suppose you run it and it usually prints to the screen:

```
This is a bunch of output.
This is more output.
```

You may instead send the characters going to `stdout` to a file by typing:

```
my_prog > output.txt
```

Note: this method will send all printf commands to the file instead of stdout.

<u>Method 2:</u> explicitly open a file in the C code and write to that file using `fprintf` or `fputc`. Note that this method is necessary if the program writes to a file and writes (separate info) to the console.

```
FILE *out = NULL;             //declare FILE pointer
out = fopen("out.txt", "w");  //open the file out.txt for writing
```

```
                                //note: if out.txt exists, this will overwrite what is in the file
                                //note: if out.txt does not exist, this will create a file with
                                //this name

if(out == NULL) {
        //something went wrong
        //return from main or return from the function
}

fprintf(out, "The special word of the day is \"creative\"!\n");

//now use fputc
char c;
for(c=50; c<75; c++) {
        fputc(c, out);
}

fclose(out);
```

File Reading

Method 1: redirect file to `stdin` (this is handy if you have interactive prompts for the user, but you want to run your code without having to retype all that data!!! ☺).

Suppose you have a program that scans in data from the keyboard. Note: `scanf` is dangerous, but for the purpose of this example, we will use `scanf`:

```
int main(void) {
  int a, b, c;
  printf("Please enter a number:\n");
  scanf("%d", &a);

  printf("Please enter a second number:\n");
  scanf("%d", &b);

  printf("Please enter a third number:\n");
  scanf("%d", &c);

  printf("Total: %d\n", a+b+c);
  return EXIT_SUCCESS;
}
```

Now, if you have a file that contains 3 ints called input.txt, you can run this program (called read3) like this:

```
read3 < input.txt
```

Method 2: Read from files using C code; note that this method is necessary if the program must read from the keyboard and a file.

```
#include <stdio.h>
#include <stdlib.h>

/* example program for reading from a file
```

```
 * and printing its contents to the screen
 * Tammy VanDeGrift
 */

int main(void) {
  char filename[101]; //char array to store the filename of at most 100 chars
  printf("Enter the name of the input file. Only works for file names less \
than or equal to 100 characters.\n");
  scanf("%100[^\n]s", &filename);
  //note: not error checking scanf, but probably should

  FILE *in = fopen(filename, "r");
  printf("Trying to open file: %s\n", filename);

  if(in == NULL) {
    fprintf(stderr, "Error opening file. Exiting.\n");
    return EXIT_FAILURE;
  }

  // read each character until end of file
  int c;
  while ((c = fgetc(in)) != EOF) {
    putc(c, stdout);
  }
  putc('\n', stdout);

  fclose(in);
  return EXIT_SUCCESS;
}
```

stores up to 100 characters typed at keyboard (not including newline) to char array filename

EOF means end of file (can check it as you read a char)

Some useful functions on files:

- `FILE * fopen(const char * filename, const char * mode)`
- `int fclose(FILE *fp)`
- `int fscanf(FILE *fp, const char * control_string, … addresses of variables for storage)`
- `int fprintf(FILE *fp, const char * control_string, …addresses of variables for storage)`
- `int fputc(int c, FILE *fp)`
- `int fgetc(FILE *fp)`
- `ssize_t getline(char **linept, size_t *n, FILE *fp) //reads one line into linept and reallocates linept if it is not big enough - getline is a safe function`

You can do:

```
man getline
```

to get more information about a function called `getline`

```
#include <stdio.h>
#include <stdlib.h>

/* Lecture 6
 * example of opening and writing to a file
 * Tammy VanDeGrift
 */

int main(void) {
  FILE *out = NULL;
  out = fopen("out.txt", "w"); //if "a" is used, file is opened
                               //in append mode
                               //file modes: "r", "r+", "w", "w+"
                               //"a", "a+" -- look these up
  if(out == NULL) {
    printf("File out.txt did not open properly. Exiting.\n");
    return 0;
  }
  fprintf(out, "The special word of the day is \"creative\"!\n");

  // use fputc
  char c;
  for(c=50; c<75; c++) {
    fputc(c, out);
  }

  fclose(out);
  return EXIT_SUCCESS;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

/* Lecture 6
 * example code that reads 3 ints and prints the total
 * Tammy VanDeGrift
 * used to show that when compiled, can do a.out < input_file.txt
 * to use input_file.txt instead of what is typed at the keyboard
 */

int main(void) {
  int a, b, c;
  printf("Please enter a number:\n");
  scanf("%d", &a);

  printf("Please enter a second number:\n");
  scanf("%d", &b);

  printf("Please enter a third number:\n");
  scanf("%d", &c);

  printf("Total: %d\n", a+b+c);
  return EXIT_SUCCESS;
}
```

```
#include <stdio.h>
#include <stdlib.h>

/* Lecture 6
 * example program for reading from a file
 * and printing its contents to the screen
 * Tammy VanDeGrift
 */

int main(void) {
  char filename[101]; //char array to store the filename of at most 100 chars
  printf("Enter the name of the input file. Only works for file names less \
than or equal to 100 characters.\n");
  scanf("%100[^\n]s", &filename);
  //note: not error checking scanf, but probably should

  FILE *in = fopen(filename, "r");
  printf("Trying to open file: %s\n", filename);

  if(in == NULL) {
    fprintf(stderr, "Error opening file. Exiting.\n");
    return EXIT_FAILURE;
  }

  // read each character until eof
  int c;
  while ((c = fgetc(in)) != EOF) {
    putc(c, stdout);
  }
  putc('\n', stdout);

  fclose(in);
  return EXIT_SUCCESS;
}
```

CS 305: In-class Activity 5 (Files)

Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____, (R)_____, (P)_____, (S)_____

Examine the code below. Part of the code needs to be completed (by your team). The program opens three files (one input file and two output files). The every_other function is called to write every other character to out1 and the remaining characters to out2.

If in has the text:
`Hello world.`

Then, out1 should have the text:
`Hlowrd`

and out2 should have:
`el ol.`

```
#include <stdio.h>
#include <stdlib.h>

#define NUM_PARAMS 4

/* prototypes */
void usage(char * program_name);
void every_other(FILE * in, FILE * to1, FILE * to2);

/* main
 * opens files, calls every_other, and closes files */
int main(int argc, char * argv[]) {
  FILE *in, *out1, *out2;

  if(argc != NUM_PARAMS) {
    usage(argv[0]);
    exit(EXIT_FAILURE);  // same as return EXIT_FAILURE
  }

  in = fopen(argv[1], "r");
  out1 = fopen(argv[2], "w");
  out2 = fopen(argv[3], "w");

  if(in == NULL || out1 == NULL || out2 == NULL) {
    fprintf(stderr, "A file could not be opened. Exiting.\n");
    exit(EXIT_FAILURE);
  }

  every_other(in, out1, out2);
```

```
  //close files
  fclose(in);
  fclose(out1);
  fclose(out2);

  exit(EXIT_SUCCESS);
}

/* usage
 * prints message about how to use the program
 */
void usage(char * program_name) {
  printf("Usage: %s <input file> <output file 1> <output file 2> \n", program_name);
  printf("Every other character in <input file> will be written to");
  printf(" <output file 1>; others will be written to <output file 2> \n");
  return;
}

/* every_other
 * outputs every other character from in to to1; outputs the other
 * characters to to2
 */
void every_other(FILE * in, FILE * to1, FILE * to2) {
  int c;
  int toggle = 1;
  while ((c = fgetc(in)) != EOF) {
       // put your code here









  }
}
```

Answer these questions:

1. How would the user enter the command for this program (assuming the executable is called every_other) if the input file is called input.txt and the output files are called out1.txt and out2.txt?

every_other _____ _____ _____

2. Complete the every_other function so that the first char read from in is written to to1, the second character is written to to2, third to to1, fourth to to2, etc. Put the code inside the while loop above. Remember to set the toggle at the end of the loop: you can do this with toggle = -toggle.

3. What questions does your group have about files?

CS 305: Good programming practices

Brainstorm answers to the following questions:

1. What contributes to good programming style?

2. What contributes to good program design? (how to use separate files, how to organize structs and functions within a file)

3. What are helpful debugging strategies?

4. What are good code development practices? (example: how many lines of code should you write before compiling? How should you assess compiler errors?)

Exam #1 Review Guide and Practice Questions

The first CS 305 exam is: Wednesday, Feb 5 at 1:35 pm. The exam length is 55 minutes.

The exam focuses on topics covered in lectures from January 13 through February 3 and will focus on the C language. You should review your coursepack, in-class activities, the GNU C tutorial book, labs, and HW1.

Here are topics from these lectures, labs, and homework:

- C control flow (if, if/else, switch, for, while)
- Functions (prototypes, definition, parameters, return type)
- Variables and types (examples: int, char, double, int *, char *, int **, etc.) and their sizes
- Arrays
- Structs
- Typedef
- Malloc, free, sizeof (allocating memory on the heap)
- Pointers (to variables, to structs, to arrays)
- NULL, dereferencing pointers, dangling pointers, pointer arithmetic
- Preprocessor directives
- Printing to console (stdout with fprintf, or use printf)
- Reading data from keyboard (stdin with fscanf, or use scanf)
- File I/O (reading from text files, writing to text files)
- Identifying when a segmentation fault would occur
- Identifying syntax and semantic errors
- Good programming style

You can be expected to write code on the exam, read code, and answer questions about code. You may be asked to find syntax errors and run-time errors in code without the aid of a computer.

You will be allowed one 8”x11.5” crib sheet (both sides) to use while taking the exam. Your crib sheet can be hand-written or typed. No other aids are permitted (computers, calculators, headphones, music, phones).

Much of the classtime on February 3 will be set aside for review for the exam. Come to class with questions that you have. The remaining portion of this review guide has practice questions to prepare for the exam.

!!! Remember: as you study, you can write small programs to see how the code compiles and executes. !!!

Question 1 (Multiple choice, choose the best answer):
Fill in the blank. If _____ is dereferenced, a segmentation fault can occur.
a. NULL.
b. a dangling pointer (points to memory not allocated to the program)
c. Both a and b.
d. None of the above.

Question 2 (Multiple choice, choose the best answer):
Could the following program result in a segmentation fault or bus error?
a. Yes, because the integer i cannot be used to index a character array.
b. Yes, because the for-loop executes too many times.
c. Yes, because the size of the array is too large for a C program.
d. No, the program is fine; it would not result in any memory faults.

```
#include <stdio.h>
int main(void) {
      int i;
      char a[100];
      for(i = 0; i <= 100; i++)
      a[i] = 'a' + i;
      return 0;
}
```

Question 3 (Multiple choice, choose the best answer):
What is the output of the program below?
a. The program does not execute successfully; a segmentation fault occurs.
b. abc
c. 012
d. aaa

```
#include <stdio.h>
int main(void){
      char c = 'a';
      char *p;
      p = &c;
      printf("%c%c%c", *p, *p + 1, *p + 2);
      return 0;
}
```

Question 4 (Multiple choice, choose the best answer):
The C compiler has a preprocessor built into it. How would one define a program-specific constant called "PI" using a preprocessor directive?
a. define 3.14159 PI;
b. #define 3.14159 PI;
c. #define PI 3.14159
d. #PI 3.14159

Question 5 (short answer):
Consider the short program below. Note that this program compiles and executes successfully. Suppose that this program is invoked from the command line using:

```
$ ./a.out Go Pilots
```

If it is invoked using the above, what are the values of argc and argv at the start of the program? If you want to use a picture to illustrate your answer, please do.

```
int main(int argc, char * argv[]) {
      if(argc != 3) {
            printf("%s error: incorrect number of parameters \n", argv[0]);
            return 1;
```

```
        }
        printf("Correct! \n");
        return 0;
}
```

Question 6 (short answer):
Consider the short program below. Note that the program compiles and executes successfully. Answer the following questions.
a. The program below has flaws. Name one style flaw and one actual error. Explain each answer.
b. Directly in the program below, add the line or lines of code that will fix the errors you identified in part a.
c. Draw the picture of memory (with pointers) that represents the program after you fixed the error.

```
int main(void) {
        int *A = malloc(sizeof(int));
        int *B = malloc(sizeof(int));
        *A = 5;
        *B = 17;
        A = B;
        B = 6;
        printf("A points to %d \n", *A);
        return 0;
}
```

Question 7 (short answer):
Consider the short program below. Note that this program compiles but does not execute correctly. Explain why this program's execution could result in a segmentation fault.

```
#include <stdio.h>
#include <stdlib.h>
int initializePointer(int * z);

int main (void) {
        int *x;
        initializePointer(x);
        printf("%d \n", *x);
        return 0;
}
int initializePointer(int * z) {
        z = (int *)malloc(sizeof(int));
        *z = 5;
        return 0;
}
```

Question 8 (Multiple choice, choose the best answer):
C is a typed language. This means that when variables are declared, their type is also declared: int, char, ListNode*, and so forth. The C compiler may not compile a program if a variable assignment has mismatched types or if a struct's fields are accessed incorrectly. Consider the following type definitions and declarations.

```
// type definitions
typedef struct tag {
        int count;
        int *first;
        int *last;
} Quack;

// variable declarations
Quack q1;
struct tag * q2;
```

```
int myCount = 5;
```

Assume that each variable's memory has been properly allocated. For each of the assignments below, identify if the assignment would be successfully compiled.

`q2->count = myCount;`	would compile	would not compile
`q2->first = &myCount;`	would compile	would not compile
`q1->count = myCount;`	would compile	would not compile

Question 9 (Multiple choice, choose the best answer):
Suppose that you have the following interface, or function prototype:

```
      int getValue(struct mydata** list);
```

The function returns an int representing the value of the elements in a "struct mydata" object. Complete the following program by calling the function *getValue* on the variable data1. Store the result in the variable *value*. You may assume that the functions *createDataObject* and *freeDataObject* have been defined elsewhere and work properly.

```
int main(void) {
      int value = 0;
      struct myData* data1 = createDataObject();
      // Call the getValue function here using one of
      // (a) thru (d) below.
      printf("The value of the object is %d \n", value);
      freeDataObject(data1);
      data1 = NULL;
      return EXIT_SUCCESS;
}
```

a. value = getValue(data1);
b. value = getValue(&data1);
c. &value = getValue(&data1);
d. value = getValue(*data1);

Question 10 (write code):
Write a *main* method that:
• Attempts opens a file, "xyz.txt", for reading.
• If the file cannot be opened, the function should return EXIT_FAILURE.
• If the file can be opened, it should attempt to read a single character from the file.
• If no character exists (e.g., because the file is empty), the function should return EXIT_FAILURE.
• If a character is read, the function should print the character (as a character) to the console, followed by a newline character, and then return EXIT_SUCCESS.

The function should always close any files that it has successfully opened.

You may assume that any necessary #includes are given at the beginning of the file. Write your function definition here:

Question 11 (write code):
Given the type definition

```
typedef struct abc {
      int arr[10];
} abc;
```

write a function *createAbc* that
• takes one parameter, an *int*
• malloc's memory for one *abc* object
• initializes all 10 array elements in the newly created object using the value of the parameter
• returns a pointer to the *abc* object

Write your function definition here:

Question 12 (short answer):
What are the contents of the array *myArray* after the following snippet is executed?

```
int myArray[] = {0,2,4,6,8,10,12,14,16,18,20};
int *p = myArray;
p += 3;
*p += 2;
p += 4;
*p += 5;
p--;
(*p)--;
```

Draw array *myArray* here:

Question 13 (short answer):
What does the following program print?

```
#include <stdio.h>
#define ZZ 3
#define f(x) x*x
int main(void) {
      printf("%d\t%d\t%d\n", ZZ, f(ZZ), f(ZZ+2));
      return EXIT_SUCCESS;
}
```

Part 2: Recursion & Foundational Data Structures

Binary Search
Recursion
Big O Analysis
Linked Lists
Stacks
Queues

CS 305: In-class Activity 6 (binary search and complexity)

Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____ (R)_____ (P)_____(S)_____

Background

One aspect of computer science is related to efficiency. How long does it take a program to process the data and return results? Up to now, we have focused on writing "correct" programs – programs that process data and return results according to a specification. Moving forward in this course, we will also focus on *efficiency*, how fast the program is in completing its task. In general, faster programs are preferred.

One common task that we often encounter is searching for data. You probably conduct searches at least once per day. It can be as simple as, "Where did I put my coat?" to "What is Shelley's phone number?".

1. What searches do you conduct? (in the physical world or with software)

2. One specific example of searching for data is looking for an item in a sorted list. Tammy does this regularly during lab – looking for your name on the roster so she can mark checkpoints. Can you think of a collection of data that you use that is sorted in some way? What is it?

3. If we have data that is sorted, searching for that data can be made more efficient by using a technique called binary search. Assume the data is stored in sorted order in an array. Assume you are looking for a particular item, called key. Then, you can maintain two indices in the array that get closer and closer together to "zero in" on the key. The indices start out at positions 0 and the length of the array minus 1. You calculate the midpoint of the indices and look at the array element at that midpoint. That value tells you to search to the right of that midpoint or to the left of that midpoint. Eventually, the indices will be equal or cross over each other. At that point, the key is found or not found. Examine the code below. Note that it also includes a version of linear search.

```c
/*****************************************************************
 * Program: bin_search.c
 * CS 305
 * Lecture: recursion
 * Purpose: implements recursive binary search
 *****************************************************************/
#include <stdio.h>
#include <stdlib.h>

/*****************************************************************
 * binary_search - finds position of an element in (portion of a)
 * sorted array
 *
 * usage:
 *    pos = binarySearch(key, array, first_index, last_index)
 *
 * parameters -
 *    key - the integer that we're searching for
 *    array - the array we're searching
 *    first_index - index of the first element of the sorted array
 *    last_index - index of the last element of the sorted array
 *
 * result -
 *    If the element is in the array, the position in the array where
 *    the element was found.  Otherwise, -1 returned.
 *
 * precondition -
 *    The array must contain at least one element, and must be sorted
 *    in increasing order for this function to work.
 *    first_index and last_index must be in bounds on the array
 *****************************************************************/
int binary_search(int key, int arr[], int first_index, int last_index) {
  // base case: first_index has reached or crossed last_index
  if (first_index >= last_index) {
    if (arr[first_index] == key) { // check for key
      return first_index; // found key at first_index
    }
    else {
      return -1; // key was not found in arr
    }
  }
  // recursive case: need to look to the left or right of arr
  else {
    int mid = (first_index + last_index) / 2;
    if(key > arr[mid]) {
      // look to the right of mid
      return binary_search(key, arr, mid + 1, last_index);
    } else {
      // look to the left of mid
      return binary_search(key, arr, first_index, mid);
    }
  }
}
```

```
/*****************************************************************
 * linear_search - finds position of an element in an array
 *
 * usage -
 * linear_search(key, array, len)
 *
 * parameters -
 * key - the integer we're searching for
 * array - the array we're searching
 * len - the length of the array we're searching
 *
 * result -
 * if the key is found, returns the position in the array of
 * that key. Otherwise, -1 is returned
 *
 * note: works on sorted and unsorted arrays
 *****************************************************************/
int linear_search(int key, int arr[], int len) {
  int i;
  if(len <= 0) { // check len, if invalid return -1
    return -1;
  }
  // search for key in arr
  for(i = 0; i < len; i++) {
    if(arr[i] == key) {
      return i;
    }
  }
  return -1;
}

/*****************************************************************
 * main - main program to exercise 'binary_search'
 *
 * This program prompts the user for a number to search for, and then
 * invokes 'binary_search' on a predefined (sorted) array.
 *
 *****************************************************************/
int main(void) {
  // the array we're searching
  int test_array[] = {3, 5, 8, 12, 25, 34, 57, 58, 59, 67, 87};

  // prompt user and read number
  int to_search = 0;
  printf("Please type number to search for: ");
  if(1 != scanf("%d", &to_search)) {
    fprintf(stderr, "Error reading number. Exiting program.\n");
    return EXIT_FAILURE;
  }

  int len = sizeof(test_array)/sizeof(*test_array);

  // search for the number using binary search
  int result = binary_search(to_search, test_array, 0, len - 1);
```

```
  // report result to console
  if (result == -1) {
    printf("Element not found using binary search\n");
  }
  else {
    printf("Element is in position %d using binary search\n", result);
  }

  // search for the number using linear search
  int result2 = linear_search(to_search, test_array, len);
  if (result2 == -1) {
    printf("Element not found using linear search\n");
  }
  else {
    printf("Element is in position %d using linear search\n", result2);
  }
  return EXIT_SUCCESS;
}
```

Answer these questions:

1. Suppose the user types 67 when this program is executed.
What are the recursive calls for binary_search?

```
binary_search(67, test_array, 0, 10)
binary_search(67, test_array, _____, ______)
binary_search(67, test_array, _____, ______)
// keep going until recursion stops
```

2. In the above execution, what are the successive values of `mid` that are assigned as the function makes recursive calls?

```
mid = 5
mid = _____
// keep going
```

3. Suppose the user types 18. What are the recursive calls for binary_search?

```
binary_search(18, test_array, 0, 10)
binary_search(18, test_array, _____, ______)
binary_search(18, test_array, _____, ______)
// keep going until recursion stops
```

4. Assume linear_search is called on (67, test_array, 11). How many numbers in test_array are examined before the function returns? _____

5. How many recursive calls were made when using binary search to find 67? (see problem #1)

_____

6. Which function is faster, in general, for searching? binary search or linear search?

7. Is linear search a recursive function? YES NO

How do you know?

Big-O running time analysis is a mathematical function that is based on the input size N of the data for the program. In these searching algorithms, N is the size of the array. Think about how much "work" is done in linear search and binary search given an array of N items. In this case, "work" is how many elements are examined in linear search for an array of length N. "work" for binary search is how many function calls binary search makes for an array of length N, since each function execution is a constant amount of code.

8. What is the big-O running time of linear search if there are N elements in the array? O( )

9. What is the big-O running time of binary search if there are N elements in the array? O( )

(break for class discussion… if you have reached this point, your group may do the bonus and move on to the next set of activities.)

(bonus) How could the implementation of binary search above be made faster?

(bonus) How could the implementation of linear search above be made faster assuming the data in the array is in sorted order? (converting it to binary search is not acceptable… do linear search but stop earlier)

Mathematically, **big-O** is defined as follows. Let f and g be two functions defined on the real numbers.

f(x) = O(g(x)) as x -> infinity
if and only if there is a positive real number C and real number x0 such that:
|f(x)| <= C*|g(x)| for x >= x0

What does this really mean? It means that g(x) serves as an upper bound, aside from a constant factor, for the function f(x) – the actual running time of a program. In computer science, we care about trends in terms of how fast a program executes, so we generally do not care about the exact value of constants when analyzing the running time of a program. This is why we use big-O analysis – figuring out the g(x) is good enough for us. We refer to the running time of a function as the **complexity** of the function.

Here is an example.

```
void func1(int n) {
      int i;
      for (i = 0; i < 2*n; i++) {
            printf("Hello\n");
      }
}
```

How much "work" is done by funct1 given the size of n?

The complexity (running time) of func1 is O(N). The loop executes 2*N times, so the actual running time f(x) = 2N. Big-O, then, is O(N).

10. Here is another function. What is its complexity?

```
void func2(int n) {
      int i, j;
      for(i = 0; i < n; i++) {
            for(j = 0; j < n; j++) {
                  printf("I love complexity.\n");
            }
      }
}
```

The complexity of func2 is O( ).

11. What is func3's complexity?

```
void func3(int n) {
      int i;
      for(i = 0; i*i < n; i++) {
            printf("Howdy\n");
      }
}
```

The complexity of func3 is O( ).

12. What is func4's complexity?

```
void func4(int n) {
      int i, j;
      for(i = 0; i < n; i++) {
            for(j=0; j< i; j++) {
                  printf("****\n");
            }
      }
}
```

The complexity of func4 is O( ).

Computing the total running time of functions is as follows.

Suppose we have a program that first calls f(n) and then calls g(n).

```
void p(int n) {
      f(n);
      g(n);
}
```

In earlier analysis, we found the running time of f(n) to be O(N) and the running time of g(n) to be $O(N^2)$.

Our program p(n) first calls f(n) and then g(n). What is the total running time of p(n)? So, the total complexity of p(n) is $O(N) + O(N^2)$. The dominant term here is N^2. Since $N < N^2$, the overall running time would be less than $2*N^2$, so the complexity of p(n) is $O(N^2)$.

13. Now, suppose our program looks like:

```
void p(int n) {
      f(n);
      g(n);
      h(n);
}
```

We found the running time of h(n) to be O(N * lg(N)). What is the complexity of p(n)? O(_____)

14. Suppose our program looks like:

```
void p(int n) {
      int i;
      for(i = 0; i < n; i++) {
            g(n);
      }
}
```

What is the complexity of p(n) given g(n) is $O(N^2)$? O( )

15. The following gives you practice adding and multiplying functions with big-O. Remember, big-O is an upper bound, so think about the maximum of the polynomials.

a. $O(N^2) + O(N \lg N) = O($ _____$)$

b. $O(N) + O(N^3) + O(N^2) = O($ _____$)$

c. $O(N^2) * O(N^3) = O($ _____$)$

d. $O(2^N) + O(N^8) = O($ _____$)$

e. $O(N^3) + O(N^{2.5})*O(N) = O($ _____$)$

16. What questions does your group have about recursion and/or complexity?

My notes about recursion and complexity:

CS 305: In-class Activity 7 (complexity practice)

Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____ (R)_____ (P)_____(S)_____

Some notes about big-O analysis:

- Note that we "lose" information going from left to right.
  - $10N^3 = O(N^3)$
  - We <u>do not</u> write $O(N^3) = 10N^3$
- Note that big-O analysis factors out machine dependencies (how fast an actual machine executes instructions – this speeds up over time, but the heart of the algorithm is still dependent on its O(___) running time analysis)
- The highest order term will eventually dominate when N gets really large.
  - for example, $.0000001N^5 + 10000000000N^2$

Below is a chart of some classic algorithms and their running times. They are in order from slowest growing in terms of big-O analysis.

Algorithm	**Running Time**
Adding two values	O(1)
Binary search	O(lg N)
Linear search	O(N)
Merge sort	O(N lg N)
Selection sort	$O(N^2)$
Matrix multiplication	$O(N^3)$ // can speed this up a bit
Scheduling	$O(2^N)$

1. Consider the following code:

```
int array[N][N] = ...;            //assigned something
// start here
int i;
int j;
int sum = 0;
for (i = 0; i < N; i++) {
       sum++;
       for (j = 0; j < N; j++) {
              sum = sum + array[i][j];
       }
}
```

a. How many times does array, i, j, and sum get initialized? _____

b. How many times does the outer loop execute? _____

c. How many times does the inner loop execute? _____

d. The running time of this program is O(_____)

2. Answer these problems:

a. $5N^2 + 1000N + 3 = O($ _____$)$?

b. $.0000001N^5 + 10000000000N^2 = O($ _____$)$?

c. Which is a faster algorithm? Algorithm A runs in $O(N^2)$ time and algorithm B runs in O(NlgN) time. Circle your answer.

a. Algorithm A is faster
b. Algorithm B is faster

Finding prime numbers

A prime number is a positive integer that has no factors other than 1 and itself. For example, 2, 3, 5, 7, and 11 are examples of prime numbers. The number 15 is not prime, since it has factors 1, 3, 5, and 15.

On a separate handout are two implementations of programs to print all prime numbers up to a given number. The `repeated_div.c` implementation checks for factors from 2 to sqrt(n) of the number to determine if it is prime. Note that we do not need to check for factors above sqrt(n), since those have corresponding factors that are less than or equal to sqrt(n).

The second implementation is the Sieve of Eratosthenes. In pseudocode, this algorithm does the following:

1. Create a list of consecutive integers from 2 to N (the code creates an array of size n+1)
2. Let p = 2 (smallest prime number)
3. Enumerate the multiples of p by counting from 2p in increments of p; mark these on the list
4. Find next number prime non-marked prime number p in the list. Repeat step 3. If no such prime number is found, stop.
5. The unmarked numbers in the list are prime.

To give you a feel of how this second implementation works, suppose N (the max number) is 50:

The original list is:

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Starting with p=2, cross out 2p (number 4), then cross out 3p (number 6), etc. Do that above.
The next prime number is p=3. Cross out 2p (number 6), 3p (number 9), 4p (number 12), etc.

The next prime number (not crossed out already) is p=5. Cross out 2p, 3p, 4p, 5p, ….
Keep doing this until you get to the number p > sqrt(50).

The uncrossed numbers are the prime numbers.

3. What are they? _____

Now, look at the implementations of `repeated_div.c` and `sieve.c`. You may focus on the "heart of the algorithm" for this exercise.

repeated_div.c:

Suppose N = max.
a. How many times does the outer loop execute? _____
b. How many times does the inner loop execute? _____
c. What is the total running time? O(_____)

sieve.c:

This one is a bit more tricky to count the "work", but let's try:
Suppose N = len.
d. How many times does the loop to clear the hits array execute? _____
e. How many times does the loop to print the primes execute? _____

So, these two parts (set-up and printing) are O(N) + O(N) = O(N). Now, the trickier part.

f. How many times does a value in the hits array get assigned to true (middle part of algorithm)? _____

It looks like the outer array executes sqrt(N) times. The inner loop only executes for primes. So, really, the inner loop only executes if the number is prime. Here's a mathematical way to see how many numbers get crossed off through the execution of the algorithm:

(N/2) //first pass when i = 2
(N/3) // second pass when i=3
(N/5) // third pass when i = 5
(N/7) …

(N/P) // until the denominator is p where p > sqrt(N).

So, this is the summation: (N/2) + (N/3) + (N/5) + (N/7) + ….(N/P)
If we factor out N, we get: N(1/2 + 1/3 + 1/5 + 1/7 + … + 1/P)

The series of 1 divided by prime numbers (maybe you remember this from calculus…) converges to loglogN. So, the total running time for the sieve version is O(NloglogN).

g. How much storage space (memory) does `repeated_div.c` use? _____

h. How much storage space (memory) does `sieve.c` use? _____

There is often the tradeoff in computer science between time and space. If an algorithm is faster, it be because it uses more memory.

Other tradeoffs in CS include time efficiency at the cost of readability (is the algorithm straightforward? can this code be maintained?). Another tradeoff may be time efficiency versus implementation difficulty (is there a library already out there that I can use? It might be slightly slower, but it saves programmer time)

So, efficiency is important in computer science. We do want algorithms that are fast. People become famous for finding efficient solutions to problems that are thought to take a long time (exhaustive search problems). But, efficiency may not always WIN in terms of design due to other considerations: memory usage, program maintenance, programmer time.

(if time) Breaking passwords

Consider a search algorithm that tries to guess a password. A password can be comprised of 100 characters in length. You want to write a decoder (tries to guess the password). Assume there are 72 valid characters (uppercase, lowercase, digits, special characters like !, $, #, etc.) that can be used for passwords.

How many combinations must be tried for passwords of length 100? _____

Assume you could compute 100 combos per second. How many could be done in a year?
100 comp/sec * 60 sec/min * 60 min/hr * 24 hrs/day * 365 days/year = ~3.15 * 10^9

Even if we had computers equal to the number of atoms in the universe (10^82) to do this in parallel, we're only at 3.15 * 10^91 possible combos per year. Not even close to enough time and enough computing power to do this exhaustive search.

4. What questions does your group have about big-O and complexity?

```
/**********************************************
 * CS 305 lecture
 * algorithm analysis - printing prime numbers
 *
 * repeated_div.c
 * based on code written by Dr. Vegdahl
 * modified by Tammy VanDeGrift
 **********************************************/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define NUM_PARAMS 2

/* prints primes up to the number specified by the user
 * at command line
 */
int main(int argc, char * argv[]) {

  // check that # of command-line arguments is correct
  if(argc != NUM_PARAMS) {
    printf("usage:\nargv[0] max_num\nto print primes up to max_num\n");
    return EXIT_FAILURE;
  }
  // check that # entered is non-negative
  if(atoi(argv[1]) < 0) {
    printf("max_num must be non-negative. exiting\n");
    return EXIT_FAILURE;
  }

  /* heart of the algorithm */
  int i, j;
  int max = atoi(argv[1]);
  for(i = 2; i <= max; i++) {  // go through each number
    bool found_prime = true;  // assume we have a prime # to start
    // look for a smaller number that divides i
    for(j = 2; j*j <= i; j++) {
      if (i % j == 0) {
       // found divisor
       found_prime = false;
       break;
      }
    }
    // print it to the screen if prime
    if (found_prime) {
      printf("%d\n", i);
    }
  }
  return EXIT_SUCCESS;
}
```

```
/**********************************************
 * CS 305 lecture
 * algorithm analysis - printing prime numbers
 *
 * sieve.c
 * based on code written by Dr. Vegdahl
 * modified by Tammy VanDeGrift
 **********************************************/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define NUM_PARAMS 2

/* prints primes up to the number specified by the user
 * at command line
 */
int main(int argc, char * argv[]) {

  // check that # of command-line arguments is correct
  if(argc != NUM_PARAMS) {
    printf("usage:\nargv[0] max_num\nto print primes up to max_num\n");
    return EXIT_FAILURE;
  }
  // check that # entered is non-negative
  if(atoi(argv[1]) < 0) {
    printf("max_num must be non-negative. exiting\n");
    return EXIT_FAILURE;
  }

  /* heart of algorithm */
  int i, j;
  int max = atoi(argv[1]);
  int len = max + 1;     // length of array
  int prime_count = 0;   // number of primes found
  bool * hits = malloc(len*sizeof(bool));  // array to store T/F for primes

  // clear hits array
  for (i = 0; i < len; i++) {
    hits[i] = false;
  }

  // go through array until at sqrt(len)
  // mark off multiples of primes
  for (i = 2; i*i <= len; i++) {
    if (!hits[i]) { // mark off its multiples in hits
      for (j = i*2; j < len; j = j+i) {
       hits[j] = true;
      }
    }
  }

  // print primes
  for (i = 2; i < len; i++) {
```

```
    if (!hits[i]) {
      printf("%d\n", i);
    }
  }

  free(hits);
  return EXIT_SUCCESS;
}
```

CS 305: Linked Lists

Suppose you want to model a collection of retail items.

```
typedef struct retail_item {
       int id;
       char * name;
       int price;
} retail_item;
```

We could create an inventory of retail_items and store them into an array, as follows:

```
retail_item inventory[10000];
```

Or, we could create an array of pointers to retail_items like this:

```
retail_item * inventory_p[10000];
```

What do we need to do to complete the following tasks?

- Add a new retail item (think HW 1)
- Remove a retail item from the array (think HW 1)
- Update the price of a retail item (let's say it goes on sale)
- Print all retail items (think HW 1)
- Find an item in the array (think HW 1)
- Get total number of retail items in the array (think HW 1)

1. For these tasks, which are fast to do when retail items are stored in an array? Which take longer?

2. What if the store has more than 10000 items?

3. What if the store has much fewer than 10000 items?

Linked Lists

A linked list is an alternate model for storing a list of items; it can grow and shrink dynamically.

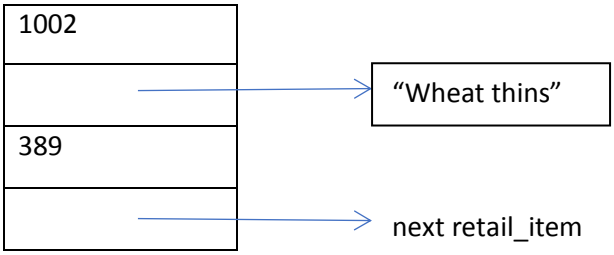
A linked list is a chain of nodes, where each node has:

- storage for a data item (or pointer to a data item)
- a pointer to the next node (the last item's pointer is NULL)
- there is a pointer to the first node in the list

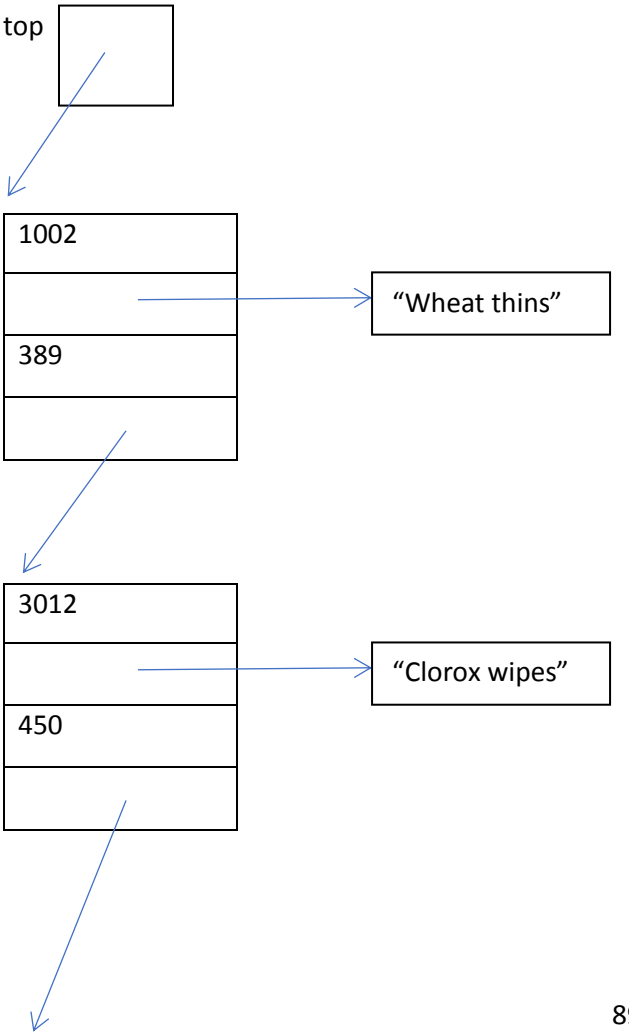
To create a linked list of retail_items:

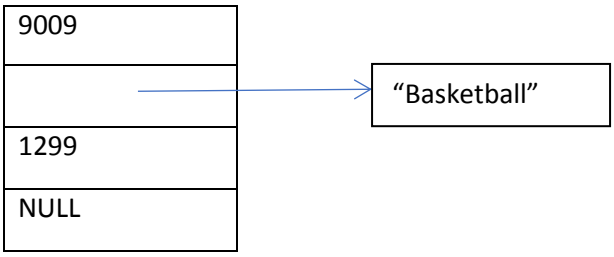
```
typedef struct retail_item_list_tag retail_item_list;

struct retail_item_list_tag {
       retail_item item;
       retail_item_list * next;
};
```

So, a single node would look like, where the first three boxes are retail_item struct members and the fourth box is the pointer to the next retail item:



A linked list would look like:





In memory, the consecutive nodes of the linked list do not need to occupy sequential places in memory (remember: an array is laid out in contiguous memory). Each individual node would be stored in contiguous bytes as a struct object; however, the location of the wheat thins node could be after the location of basketball which is after the Clorox wipes in memory.

In the above picture, the memory could look like:

Memory address	Data	Corresponds to:
2000	3012	Id of second item in linked list
2004	9000	Pointer to "Clorox Wipes"
2012	450	Price of second item in linked list
2016	2050	Pointer to third item in linked list
…		
2050	9009	Id of third item in linked list
2054	4000	Pointer to "Basketball"
2062	1299	Price of third item in linked list
2066	0 / NULL	NULL
…		
3000	3008	top
3008	1002	Id of first item in linked list
3012	10000	Pointer to "Wheat thins"
3020	389	Price of first item in linked list
3024	2000	Pointer to second item in linked list
…		
…		

Creating a linked list node

You can create a local variable as a node, like this:

```
retail_item_list  r1 = {{9009, "Basketball", 1299}, NULL};
retail_item_list  r2 = {{3012, "Clorox wipes", 450}, &r1};
retail_item_list  r3 = {{1002, "Wheat thins", 389}, &r2};
retail_item_list * top = &r3;
```

However, this is pretty unusual. It requires that you know all the members of the list; in this case, it may make more sense to use an array. Instead, we can write a function to create a linked list node and return a pointer to it.

```
retail_item_list * create_retail_item_list(int product_id, char * product_name, int
product_price, retail_item_list * next_item) {
	retail_item_list *ret = malloc(sizeof(retail_item_list));
	ret->item.id = product_id;
	ret->item.name = product_name;
	ret->item.price = product_price;
	ret->next = next_item;
	return ret;
}
```

So, to use this function in the code:

```
retail_item_list * top = NULL;
// insert basketball
top = create_retail_item_list(9009, "Basketball", 1299, top);
// insert Clorox wipes
top = create_retail_item_list(3012, "Clorox wipes", 450, top);
// insert wheat thins
top = create_retail_item_list(1002, "Wheat thins", 389, top);
```

What if instead we have a pointer to `retail_item` in the retail_item_list struct instead?

```
typedef struct retail_item_list_tag retail_item_list;
struct retail_item_list_tag {
	retail_item * item;   // pointer to retail_item object
	retail_item_list * next;
}
```

4. What would the picture look like now? (assume the same 3 items are put in the list)

5. What are some advantages of linked lists?

6. What are some disadvantages of linked lists?

7. When are arrays a better choice?

8. When are linked lists a better choice?

Summary

Linked lists are a FUNDAMENTAL data structure in CS. They are often used in job interviews. *Example: Write the code to define a linked list of integers. Write the code to traverse the linked list and find its length.*

What might we want to do with a linked list?

- Create an empty linked list
- Determine the number of nodes in a linked list (length)
- Print the elements of the linked list
- Find an element in the linked list
- Insert item at beginning (head of list)
- Insert item at end (tail of list)
- Insert item in between certain items (for example, keep data in sorted order)
- Delete first item
- Delete last item
- Delete a particular item
- Delete the entire list
- Reverse the list
- Sort the list
- Split into two lists
- Merge two linked lists

9. Can you think of other operations you might want to do?

We will now use a linked list of integers (similar to the textbook code) to simplify the discussion and some of these operations. Note that your textbook shows the implementation of several of the functions from the above list.

CS 305: In-class Activity 8 (linked lists)

Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____ (R)_____ (P)_____(S)_____

Review the code for linked lists below. Note that the function definitions are not shown here, but are part of the code file.

```
#include <stdlib.h>
#include <stdio.h>

typedef struct nodeTag Node;

/* similar to the textbook -- a Node represents one node in the linked list */
struct nodeTag {
  int num; // value stored in node
  Node * next; // pointer to next node in list
};

// book also defines Node * as NodePtr
// Tammy prefers to use type Node * instead of NodePtr for the reminder
// of the pointer type

/* function prototypes on linked lists */
Node * makeNode(int n, Node * nextItem);
int    length(Node * list);
void   print(Node * list);
void   insertTail(int n, Node ** list);
Node * find(int n, Node * list);
int    delete(Node * toDelete, Node ** listPtr);

/* main function */
int main(void) {
  // create linked list
  Node * top = NULL;
  top = makeNode(6, top);
  top = makeNode(10, top);
  top = makeNode(-3, top);
  top = makeNode(2, top);
  print(top);

  // q1: What does the picture of top look like now?

  int len = length(top);
  printf("Length of list: %d\n", len);

  // q2: What is the value of len?

  insertTail(25, top);
  print(top);
```

```
  // q3: What does the picture of top look like now?

  Node * ten = find(10, top);
  if(ten == NULL) {
    printf("Not found: 10\n");
  } else {
    printf("Found: 10, memory location: %p\n", ten);
  }

  printf("deleting 10:\n");
  int ret = delete(ten, &top);
  print(top);

  // q4: What does the picture of top look like now?

  printf("deleting 2:\n");
  Node * two = find(2, top);
  ret = delete(two, &top);
  print(top);

  // q5: What does the picture of top look like now?

  printf("deleting 15:\n");
  Node * fifteen = find(15, top);
  ret = delete(fifteen, &top);
  print(top);

  return EXIT_SUCCESS;
}
```

Answer these questions:

1. Without seeing the code implementation, answer the questions in the comments in the main function above.

q1:

q2:

q3:

q4:

q5:

2. Write the code snippet to insert at the bottom of the main function (before the return) to insert a node with a value of 11 at the front of `top`:

3. Write the code snippet to insert at the bottom of the main function (before the return) to insert a node with value -8 at the back of `top`:

< see function definitions below >

Review the implementations of makeNode, length, print, insertTail, find, and delete.

4. Now that you can see the function definitions, do you need to edit any answers in question 1? If so, write your new answers to the right of your original answers.

5. Why must the function `delete` take `Node ** listPtr` as a parameter instead of `Node *list`?

6. Complete the function definition for `insertHead`. This function should insert a new node with value `n` at the head of the list pointed to by `listPtr`.

```
void insertHead(int n, Node ** listPtr){




}
```

Circular linked lists

7. If the linked list's last node does not have next assigned as NULL, but rather another node in the list, we have a circular linked list. Write a code snippet to create 3 nodes in main that creates a circular linked list. Use numbers 1, 2, and 3 for the values of the linked list.

Doubly linked lists

We can create a linked list struct like this:

```
typedef struct nodeTag Node;

struct nodeTag {
  int num; // value stored in node
  Node * next; // pointer to next node in list
  Node * prev; // pointer to previous node in the list
};
```

This is called a *doubly linked list*, since it has a link to the next item and a link to the previous item. What's nice about doubly linked lists is that they allow for forward and backward traversal. It does require more overhead – each node has an extra field. Plus, for backward traversal to start at the end of the list, one must know the address of the last Node in the list.

8. Suppose we had definitions for functions to create doubly linked lists and inserted the values 2, 4, and 6 (inserted at the back each time). What does the picture of a doubly linked list look like?

9. (if time) Write the function definition for `numPos`. This function should return the number of items in the linked list (passed as a parameter) whose `num` value is greater than 0.

10. (if time) If you wrote the function for `numPos` iteratively, try to write a recursive version. If you wrote it recursively, try to write it iteratively.

11. What questions does your group have about linked lists?

```
/******************** function definitions ****************/

/* makeNode
 * parameters -- n (the number to store in the node)
 *            -- nextItem (the next link of the node)
 * slightly different than textbook version */
Node * makeNode(int n, Node * nextItem) {
  Node * ret = (Node * ) malloc(sizeof(Node));
  ret->num = n;
  ret->next = nextItem;
  return ret;
}

/* length
 * parameter -- list (the linked list)
 * returns the length (# nodes) in the linked list
 * implemented iteratively */
int length(Node * list) {
  int len = 0;
  while(list != NULL) {
    len++;
    list = list->next;
  }
  return len;
}

/* print
 * parameter -- list (the linked list)
 * prints the values of the nodes (in order) of the list
 */
void print(Node * list) {
  printf("Linked list contents: ");
  while(list != NULL) {
    printf("%d ", list->num);
    list = list->next;
  }
  printf("\n");
}

/* insertTail
 * parameters -- n (the value of the node to insert)
 *            -- list (the linked list)
 * inserts new node at the end with value n
 * note: this is done by passing the pointer to list, so
 * when the function returns, the list object that was
 * passed to this function has been altered
 */
void insertTail(int n, Node ** listPtr) {
  Node * list = *listPtr;
  if(list == NULL) {
    // create a 1-node list
    *listPtr = makeNode(n, NULL);
    return;
  }
  while(list != NULL) {
```

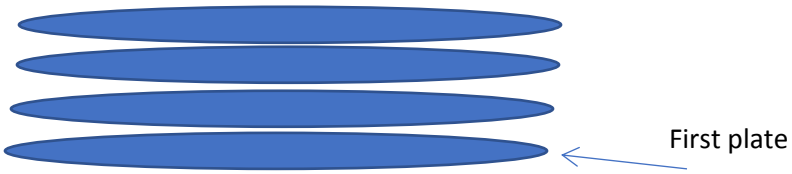
```
    if(list->next == NULL) {
      // insert new node here since we found the last node
      list->next = makeNode(n, NULL);
      return;
    }
    list = list->next;
  }
}

/* find
 * parameters -- n (the value to search for)
 *            -- list (the linked list)
 * returns a pointer to the first node found with value n
 * if no such value is found, returns NULL
 */
Node * find(int n, Node * list) {
  while(list != NULL) {
    if(list->num == n) {
      return list;
    }
    list = list->next;
  }
  // no node with value n found
  return NULL; // or could return list, since list has value NULL
}

/* delete
 * parameters -- toDelete (the node to find and delete)
 *            -- listPtr (pointer to the list)
 * note: must pass listPtr in case the first element of the list
 * is deleted -- passing the list by reference, so the address
 * to the first item in the list can get updated if necessary
 *
 * returns 0 if no item found and deleted
 * returns 1 if a node is deleted
 */
int delete(Node * toDelete, Node ** listPtr) {
  Node * list = *listPtr;  // list is the linked list
  // case: either toDelete or list is null -- will not be deleting
  if(toDelete == NULL || list == NULL) {
    return 0;  // indicates no change to the list
  }

  // special case: toDelete is first node in list
  if(toDelete == list) {
     *listPtr = list->next;   // now list->next becomes first node in list
                              // returning new first address via pointer
     free(toDelete);
    return 1;  // indicates that a node was deleted
  }

  // case: need to find toDelete somewhere other than first node in list
  Node * before = list;
  list = list->next;
  while(list != NULL) {
```

```
    if(toDelete == list) {
      // redo pointers and then free memory
      before->next = list->next;
      free(list);
      return 1;
    }
    before = list;  // update for next iteration
    list = list->next;
  }
  return 0;  // toDelete not found
}
```

CS 305: Stacks

Think about a cafeteria or buffet line at a party. The plates are usually stacked together. What other items in the world are stored in "stacks"?

Think about the first plate put into a stack. 3 more plates are placed on top of it.



Which plate comes off the stack first?

Stacks provide LIFO behavior. LIFO == Last In First Out

Suppose you are waiting in line for the concert of your favorite band. It is general admission for the concert, so only 500 people can attend. Would you want the event staff to use LIFO behavior? Why or why not?

ADT == Abstract Data Type

- is a data type that supports certain operations and has documented behavior, but the underlying implementation is left unspecified

In theory, this means that the underlying implementation could change (for example, a linked list representation is changed to an array implementation) and the program that uses the ADT will still work.

Can think of this as an API (Application Programmer Interface). Remember all those classes in Java? Each one specifies the methods and constructor(s) that a programmer can use; someone working on the Java language might change the underlying representation of ArrayList, but any program that uses ArrayList should still work.

Back to Stacks

The Stack ADT supports:

- initialize() – initializes a new stack to be empty
- isEmpty(Stack s) – tells whether stack s is empty
- push(Stack s, Item it) – pushes it to stack s, it is put at the top
    - could fail if stack is "full"
- pop(Stack s) – pops top item from stack and returns it; fails if stack is empty
- May also provide (depending on ADT):
    - isFull(Stack s) – tells if a stack is full
    - peek(Stack s) – returns the top of the stack but leaves it on the stack (equivalent to a pop following by pushing what was popped); fails if the stack is empty

This is just the stack's interface. Only the programmer of the stack data structure knows the underlying implementation. This uses *information hiding* in software development. APIs make use of information hiding.

Using Stacks: An Example of Parsing For Nested ( ) and [ ]

One task of a compiler is to determine if an expression has properly nested parenthesis and square brackets in an arithmetic expression. Note that the compiler must also determine property nested { } in C programs.

Here is an expression that one might put in a program:

3 + [(8 – 6) * [(80 – 6)*(8 + 16)]] * (cos (x) + 15.2)

Does this expression properly nested parentheses and [ ]? YES NO

As you traversed this expression, what did you do to determine if the ( ) and [ ] were properly nested?

We can think of this as:
OK + [OK * [OK*OK]] * (cos OK + OK)
OK + [OK * OK] * OK
OK + OK * OK
OK

Psuedocode for an algorithm to detect properly nested ( ) and [ ]:

```
S = initStack()
While still have input:
        Examine next symbol
        If symbol is not a delimiter, discard it
        If symbol is a left delimiter, push it to stack
        If symbol is a right delimiter, pop stack (call it top)
                If top does not exist, return false
                If top is not the left delimiter of symbol, return false
                If top is the left delimiter of symbol, discard symbol and keep going
If stack is empty, return true
```

Try this on a few inputs (in-class activity):

([x + 4) * 6]
4+3)
(8 + [[x + y] * 4)
3 + 2
7 +] 3 [
((6 + x) * (y + z)) * [(3 + z)]
empty string

Implementation of Stacks

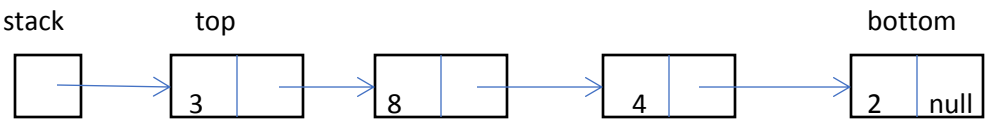
1. Using an array

0	1	2	3	4	5	6	7	8
5	-3	7	9	2	'junk'	'junk'	'junk'	'junk'
bottom				top				

1A. What are the pros of using an array to store the stack?

1B. What are the cons of using an array to store the stack?

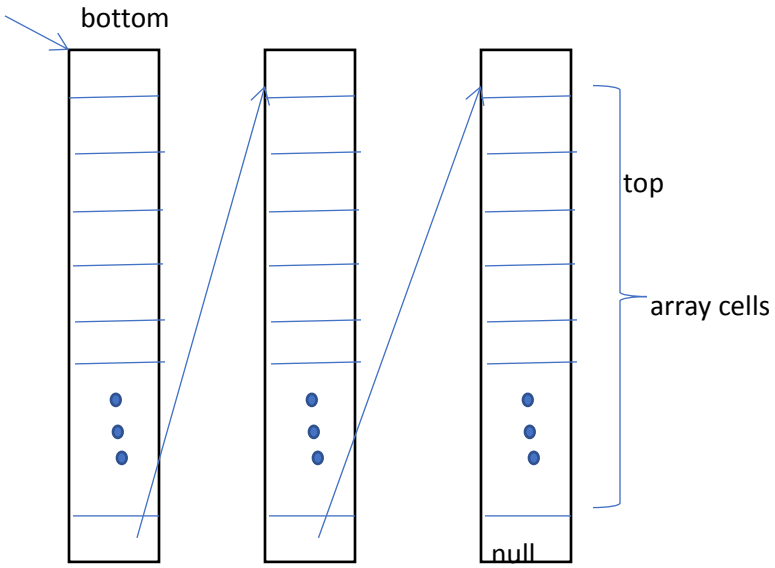
2. Using a linked list



2A. What are the pros of using a linked list to store the stack?

2B. What are the cons of using a linked list to store the stack?

3. A hybrid approach (linked list of arrays of size ~100)



It's like dynamically allocating a new 100-sized array when you need to grow the stack. If you go off the end of the array on a push, you create a new node in the linked list and put the item in the 0th location of the array of that node. If you pop off the front of an array, you move back to the previous node in the linked list.

3A. What are the pros of the hybrid approach?

3B. What are the cons of the hybrid approach?

Note: with any implementation, the programmer using the stack data structure should not know the difference.

4. What are other ways in which stacks are used in computing?

A. Function calls (local variables are put on the stack; when function exits, the information associated with the function is popped from the stack). Suppose main calls f, f calls g, g calls h and then calls k.

```
main -> f -> g -> h
             g -> k
```

So, the information for main is on the stack, then the info for f is pushed to the stack, then info for g is pushed to the stack, then info for h is pushed to the stack; when h returns, the info for h is popped, then g calls k so info for k is pushed. When k returns, info for k is popped. When g returns, info for g is popped. When f returns, info for f is popped. When main returns, info for main is popped.

Recursion uses the stack to solve problems.

B. What software do you use in which you think a stack is the underlying data structure?

In lab, you will have practice coding a stack data structure. The textbook shows the implementations of stacks using linked lists and arrays.

CS 305: In-class Activity 9 (stacks)

Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____ (R)_____ (P)_____(S)_____

Stacks can by handy for solving problems. For example, detecting properly nested ( ) and [ ] in expressions (something a compiler must do) can be solved using a stack.

Psuedocode for an algorithm to detect properly nested ( ) and [ ]:

```
S = initStack()
While still have input:
        Examine next symbol
        If symbol is not a delimiter, discard it
        If symbol is a left delimiter, push it to stack
        If symbol is a right delimiter, pop stack (call it top)
                If top does not exist, return false
                If top is not the left delimiter of symbol, return false
                If top is the left delimiter of symbol, discard symbol and keep going
If stack is empty, return true
```

Try this on a few inputs (in-class activity):

a. ([x + 4) * 6]

Show the contents of the stack S after each push and pop:

____
Stack S

What does this return? true false

b. 4+3)

Show stack:

What does this return? true false

c. (8 + [[x + y] * 4)

Show stack:

What does this return? true false

d. 3 + 2

Show stack:

What does this return? true false

e. 7 +] 3 [

Show stack:

What does this return? true false

f. ((6 + x) * (y + z)) * [(3 + z)]

Show stack:

What does this return? true false

g. empty string

Show stack:

What does this return? true false

What questions does your group have about stacks?

```c
/* Tammy VanDeGrift
 * CS 305
 * Stacks Lab
 * implements functions on stacks: initStack, empty, push, and pop
 */
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

/* initializes a new stack */
Stack initStack() {
  Stack s = (Stack) malloc(sizeof(StackType));
  s->top = NULL;
  return s;
}

/* empty returns 0 if S is empty and non-zero if S is not empty */
int empty(Stack S) {
  return (S->top == NULL);
}

/* pushes d to S */
void push(Stack S, StackData d) {
  Node * n = (Node *)malloc(sizeof(Node));
  n->data = d;
  n->next = S->top;
  S->top = n;
}

/* pops top item from S */
StackData pop(Stack S) {
  if(empty(S)) {
    printf("Stack is empty. Attempting to pop an empty stack. Exiting program.\n");
    exit(1);  // exiting program
  }
  // there is data to pop
  StackData toReturn = S->top->data;
  Node * tmp = S->top;  // in order to free this later
  S->top = S->top->next;  // move pointer to next item in stack
  free(tmp);
  return toReturn;
}
```

CS 305: Queues

Suppose you are standing in line for your favorite amusement park ride. Wheeee!!!!! This is an example of a queue. Patrons wait in a line. As they join the line, they are "enqueued". As they get into the rollercoaster car, they are "dequeued".

1. What other scenarios can you think of that use queues?

Queues provide FIFO behavior. FIFO == First In First Out

With the queue ADT, we have the following operations:

- initQueue() – initializes an empty queue
- isEmpty(Q) – returns true if queue is empty; false, otherwise
- enqueue(Q, data) – puts data in the back of the queue
- dequeue(Q) – removes first item in queue
- May support the following:
    - isFull(Q) – if the queue has reached maximum capacity, return true; else, return false
        - You will implement this in lab

This is the queue's interface. Only the programmer knows how the queue is implemented.

2. How might queues be used in computing?

Network routers – forwarding packets
Operating systems – scheduling processes

Can you think of others?

Implementation of Queues

1. Using an array
Think of this as a circular queue, with head == index to the left of first element in the queue. Another variable called tail == index to the element that is the last in the queue. The array is a fixed size.

The QueueType is the data storage of the elements in the queue.

```
#define MAX_Q 6  // 1 more than what can be stored in the queue
```

```
/* data to store into queue */
typedef int QueueData;  // can change type by updating 'int'
                        // or defining a struct

/* queue data structure */
typedef struct QueueTag QueueType;
typedef struct QueueTag* Queue; // pointer to queue struct
                                // so when it is passed, the values
                                // can be updated in functions

struct QueueTag {
  int head;
  int tail;
  QueueData data[MAX_Q]; // space for items in queue
};
```

A new queue has head set to 0 and tail set to 0.

```
/* initializes empty queue */
Queue initQueue() {
  Queue q = malloc(sizeof(QueueType));
  q->head = 0;
  q->tail = 0;
  return q;
}
```

An empty queue has head == tail.

```
/* returns 1 if queue is empty and 0 otherwise */
int empty(Queue Q) {
  return (Q->head == Q->tail);
}
```

Inserting an element into the queue: update tail by adding 1; if it goes off end of array, put it at 0, insert data item at tail.

```
/* puts data item d into queue */
void enqueue(Queue Q, QueueData d) {
  Q->tail++;
  Q->tail = Q->tail % MAX_Q; // in case it goes off array
  Q->data[Q->tail] = d;
}
```

Removing an element from the queue: if empty, exit. update head by adding 1; if it goes off end of array, put it at 0. Return data item at head.

```
/* removes data item from queue */
QueueData dequeue(Queue Q) {
  if(empty(Q)) {
    printf("Attempting to remove from empty queue\n");
    exit(1);
  }
  Q->head++;
  Q->head = Q->head % MAX_Q; // in case it goes off array
  return Q->data[Q->head];
}
```

Let's see what happens in the array as items are enqueued and dequeued.
Suppose we have a queue of max size 6 (so at most 5 items can be the queue at any given time). In this implementation, we need to have one extra cell in the array, so we can tell if the head is equal to the tail (for empty) rather than this signifying full.

```
Queue q = initQueue();
```

Initial queue:

'junk'	'junk'	'junk'	'junk'	'junk'	'junk'

head = 0

tail = 0

```
 enqueue(q, 5);
```

'junk'	5	'junk'	'junk'	'junk'	'junk'

head = 0 tail = 1

```
enqueue(q, 8);
```

'junk'	5	8	'junk'	'junk'	'junk'

head = 0 tail = 2

```
enqueue(q, 4);
```

'junk'	5	8	4	'junk'	'junk'

head = 0 tail = 3

```
QueueData a = dequeue(q);
```

'junk'	5	8	4	'junk'	'junk'

head = 1 tail = 3

```
QueueData b = dequeue(q);
```

'junk'	5	8	4	'junk'	'junk'

head = 2 tail = 3

```
enqueue(q, 20);
```

'junk'	5	8	4	20	'junk'

head = 2 tail = 4

```
enqueue(q, 40);
```

'junk'	5	8	4	20	40

head = 2 tail = 5

```
enqueue(q, 50);
```

50	5	8	4	20	40

tail = 0 head = 2

```
enqueue(q, 6);
```

50	6	8	4	20	40

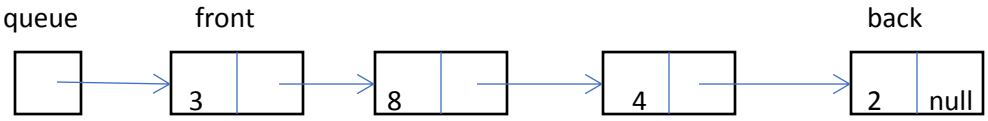
tail = 1 head = 2

Now, what would happen if we tried to add another item to the queue? (You can either test that the queue is full and not add it or copy this array to a larger array and keep processing.)

1A. What are the pros of using an array to store the queue?

1B. What are the cons of using an array to store the queue?

2. Using linked lists



If a new element (for example, 20) is enqueued, it would be put at the back of the linked list. When an element is dequeued, the front node is removed.

2A. What are the pros of using a linked list to store the queue?

2B. What are the cons of using a linked list to store the queue?

Doing a hybrid approach (linked list nodes where each node is an array) is probably more complicated than with stacks. The circular queue of the array will take some management (to know what element is the front of the queue of each linked list node). You would need to keep track of a head and tail for each linked list node array.

Priority Queues

In some computing applications, you may want to set the priorities of certain items. For example, in networking, a router may prioritize forwarding TCP packets over UDP packets. If so, the router could maintain a priority queue (TCP packets get placed in front of UDP packets in a single queue or the two priority levels are stored in separate queues.)

Think about the airport check-in line. Some airlines have a regular line and a first class line.

first-class line 

regular line 

One algorithm for serving customers might be:
If the first-class line is not empty, serve the first customer in first-class.
If the first-class line is empty, serve the first customer in the regular line.

This is an example of a priority queue (data has an attached priority). What potential issue does serving customers in a priority queue have?

What other examples can you think of that use priority queues?

Now, practice using a queue with underlying representation as an array (according to code above).

CS 305: In-class Activity 10 (queues)

Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____ (R)_____ (P)_____(S)_____

Queues are a FIFO data structure and allow items that are first in the queue to exit first from the queue. Think of a line of waiting people. Refer to the lecture code for queues with an array as the underlying representation. Update the values the queue, head, and tail as items are enqueued and dequeued.

You may also want to keep track of the items in the queue as a simple list, crossing out dequeued items.

```
#define MAX_Q 8  // 1 more than what can be stored in the queue
typedef char QueueData;  // storing chars into the queue

Queue q = initQueue();   // done for you
```

Initial queue:

'junk'	'junk'	'junk'	'junk'	'junk'	'junk'	'junk'	'junk'

head = 0

tail = 0

```
enqueue(q, 'A');
```


head = _____

tail = ____

```
enqueue(q, 'A');
enqueue(q, 'C');
enqueue(q, 'G');
enqueue(q, 'T');
```


head = _____

tail = ____

```
enqueue(q, 'R');
QueueData remove = dequeue(q);
enqueue(q, 'W');
remove = dequeue(q);
enqueue(q, 'X');
enqueue(q, 'B');
enqueue(q, 'H');
remove = dequeue(q);
// What does q have in it now as a list of items? ____________________________________
```


head = _____

tail = ____

```
remove = dequeue(q);
enqueue(q, 'J');
remove = dequeue(q);
remove = dequeue(q);
enqueue(q, 'L');
```


head = _____

tail = ____

```
int emp empty(q);
if(emp) {
        printf("q is empty\n");
} else {
        printf("q is not empty\n");
}

//What is printed? ______________________________

enqueue(q, 'P');
remove = dequeue(q);
remove = dequeue(q);
printf("val of remove: %c\n", remove);

//What is printed? ________________________________

enqueue(q, 'K');

//What is stored in the queue data structure now?
```


head = _____

tail = ____

How many items are currently "in the queue"? _____

What questions does your group have about queues?

My notes about stacks and queues:

Exam #2 Review Guide and Practice Questions

The second CS 305 exam will be: **Friday, Mar 13** at 1:35 pm. The exam length is 55 minutes.

The exam focuses on topics covered in lectures from February 7 through March 11. Although the exam's focus is not on the C language, it is expected that you can read and write C code that you learned prior to exam 1. You should review your coursepack, in-class activities, data structures textbook, prelabs, labs, HW 2, and HW 3.

Here are the topics:

- make and makefiles
- gdb, ddd
- Complexity and O-notation
- Linear search
- Binary search
- Prime numbers (looking for divisors, sieve of Eratosthenes)
- Recursive functions (examples: palindrome, gcd, multiplication with Russian Peasant Algorithm, printing numbers in different bases, linked list length, linked list copy, linked list merge lists)
- Arrays
- Abstract Data Types
- Linked Lists, Circular Linked Lists, Doubly Linked Lists
- Stacks and Operations (pop, push, peek, isEmpty, isFull, initialize)
- Queues and Operations (enqueue, dequeue, isEmpty, isFull, initialize)
- C programming (prior to exam 1)

You can be expected to write code on the exam, read code, and answer questions about code. You may be asked to find syntax errors and run-time errors in code.

You will be allowed one 8"x11.5" crib sheet (both sides) to use while taking the exam. Your crib sheet can be hand-written or typed. No other aids are permitted (computers, calculators, headphones, music, phones).

Much of the class time on March 11 will be set aside for review for the exam. Come to class with questions you have about the material. The remaining portion of this review guide has practice questions to prepare for the exam.

Remember: as you study, you can write small programs to see how the code compiles and executes. You may use the lab files to experiment.

SAMPLE QUESTIONS

Question 1: You want to find the number 41 in the following sorted array. Using binary search from lecture, what are the recursive calls that are made if the array contains the following integers:

3	6	14	15	18	20	22	35	37	39	40	41	45	57	60	62

```
binarySearch(0, 15);
```

List the recursive calls until the recursion terminates:

Question 2: Now assume linear search is conducted on the array in question 1. How many items in the array are examined before linear search returns?

_____

Question 3: Using the linked list code from lecture, define the function called `printAndFree`. It should print each item in the list (starting with the first node) and then free the node before moving on to the next node in the list. This is a combination of the print function and the free function.

```
void printAndFree(Node * list) {

}
```

Question 4: Simplify the following O-notation expressions:

$O(n^2) + O(n^3) = O($ _____ $)$

$O(n^2 lgn) + O(n^{2.5}) = O($_____$)$

$O(n^2)*O(n^3) = O($_____$)$

$O(2^n) + O(n^7) = O($_____$)$

$n^5 + 100000n^3 + 50n^2 + 2 = O($_____$)$

Question 5: Consider the stack data structure. It has the following operations: initStack(), empty(S), push(S, item), pop(S) as defined in lab. The stack data type is int.

Assume the main function is written as follows:

```
int main(void) {
      Stack s = initStack();
      push(s, 3);
      push(s, 15);
      push(s, 10);
      push(s, 8);
      pop(s);

      // A: What is s's contents?

      push(s, 12);
      push(s, 2);
      pop(s);
      push(s, 7);

      // B: What is s's contents?

      pop(s);
      pop(s);
      push(s, 4);

      // C: What is s's contents?

      freeStack(s);
}
```

A: bottom of stack _____ top of stack

B: bottom of stack _____ top of stack

C: bottom of stack _____ top of stack

Question 6: Determine the complexity of the following functions. Express you answer in big-O notation, as a function of n. Giving your reasoning may improve your chances for partial credit.

```
void function1(int n) {
      int i, j, k;
      for(i = 0; i < n; i++) {
            for(j = 0; j < n; j++) {
                  for(k = 0; k < j; k++) {
                        printf("Hello\n");
                  }
            }
      }
}
```

complexity of function1: O(_____)

```
void function2(int n) {
      int i;
      for(i = 0; i * i < n; i++) {
            printf("I love C.\n");
      }
}
```

complexity of function2: O(_____)

```
void function3(int n) {
      int i, j;
      for(i = 0; i < n; i++) {
            for(j = 1; j < n; j = j*2) {
                  printf("Howdy!\n");
            }
      }
}
```

complexity of function3: O(_____)

Question 7: Write a **recursive** version of the function `sumDigits`. If the value 10 is passed as a parameter, it should return the sum of 1 + 2 + 3 + 4 + … + 10.

```
int sumDigits(int n) {









}
```

In the function above, label the base case(s) and the recursive case(s).

Question 8: Consider the queue data structure. It has the following operations: initQueue(), empty(S), enqueue(S, item), dequeue(S) as defined in lab. The queue data type is int. Assume the queue data array has size 10 (so 9 elements can be in the queue at any given time).

Assume the main function is written as follows:

```
int main(void) {
      Queue q = initQueue();
      enqueue(q, 3);
      enqueue(q, 15);
      enqueue(q, 10);
      enqueue(q, 8);
      dequeue(q);

      // A: What is q's contents?

      enqueue(q, 12);
      enqueue(q, 2);
      dequeue(q);
      enqueue(q, 7);

      // B: What is q's contents?

      dequeue(q);
      dequeue(q);
      enqueue(q, 4);

      // C: What is q's contents?

      freeQueue(q);
}
```

A: front of queue _____ back of queue

B: front of queue _____ back of queue

C: front of queue _____ back of queue

In the code above, what is q->data, q->head, and q->tail at each point (A, B, C)?

A
data:

head: _____
tail: _____

B
data:

head: _____
tail: _____

C
data:

head: _____
tail: _____

Question 9: What are the advantages of using a linked list as the underlying representation of a stack?

Question 10: What are the advantages of using an array as the underlying representation of a queue?

Question 11: What is the big-O complexity of binary search (given a list with N items)? O(_____)

Question 12: Suppose the following makefile is written. The source code compiles. The command `make prog1` is given. Now assume `util.h` is modified and saved. `make prog1` is executed again.

```
prog1: prog1.o util.o
        gcc -o prog1 prog1.o util.o
prog1.o: prog1.c util.h
        gcc -c prog1.c
util.o: util.c util.h
        gcc -c util.c
```

Which command(s) is/are executed?

Question 13: What is the gdb command to list the current breakpoints?

Question 14: What is the gdb command to set a breakpoint at line 20?

Part 3: Trees, Sorting, and Graphs

Trees
Binary Search Trees
Selection Sort
Insertion Sort
Quicksort
Merge Sort
Graphs

CS 305: Dictionary ADT and Trees

1. Think about how you use a dictionary. What do you "search for" in a dictionary?

2. What information to you get back from a dictionary?

A **dictionary ADT** stores a collection of items and supports the following operations:

- create() // create new Dictionary
- insert(Dictionary d, key k, value v) // insert a new value with given key
- find(Dictionary d, key k) // search for a given key and return the value
- delete(Dictionary d, key k) // delete key and item that corresponds to the key
- print(Dictionary d) // print all items in dictionary

Some optional operations that may be implemented:

- size(Dictionary d) // returns number of items in dictionary
- sort(Dictionary d) // put items in order by key values
- join(Dictionary d1, Dictionary d2) // union two dictionaries; return new one
- kth_smallest(Dictionary d) // return kth smallest item accd to key values

In the above example of a real dictionary, the keys correspond to words and the items correspond to the definitions.
In some dictionary ADTs, all we need to store are the keys (the values are the keys themselves). In general, dictionaries hold *(key, value)* pairs.
Dictionaries come up all the time in computing. Here are some examples:

Username -> Account Information
Gene name -> Location in genome
IP address -> Router's forwarding port

3. What other computing contexts use dictionaries?

Let's think about how much time each of the dictionary operations would take if using linked lists and arrays as underlying data structures.

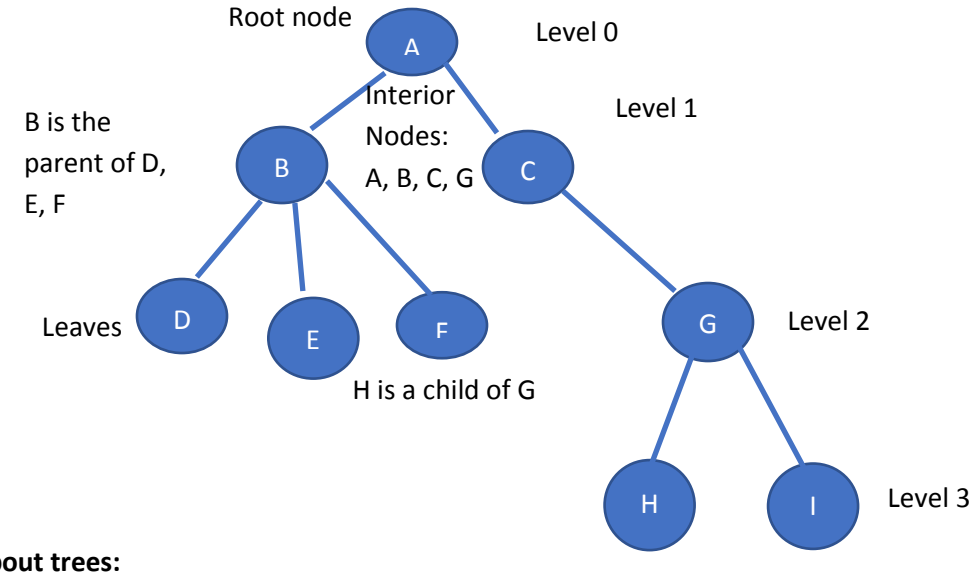
	Insert	Find	Delete
Unsorted linked list	O(1)*	O(n)	O(n)
Unsorted array	O(1)*		#
Sorted linked list			
Sorted array			#

O(N) time if you first need to see if (key, value) pair is already in the dictionary; not allowing duplicates *
lazy deletion (do not shift elements; have extra field to mark data as valid or not) #

We generally want to make operations as efficient as possible in computing. So, let's look at a new data structure that supports the Dictionary ADT that can perform better than arrays and linked lists.

TREES

A tree is a fundamental data structure in computing. We generally draw trees with the root node at the top of the tree (upside-down from regular trees you see in nature).



Some more vocabulary about trees:

Children are ordered left to right; a parent could have 0 or more children.

A tree with 0 nodes is *an empty* tree.

Ancestors of a node N are the nodes in the path from N to the root of the tree.

Descendants of a node N are the set of nodes that can be reached from any downward path from N.

The *height* of the tree is the number of nodes along the longest (deepest) path of the tree. The height of an empty tree is 0.

The *subtree* at node N is node N with all its descendants.

4. How could trees be useful for modeling data in computing applications?

Example: From Java, class hierarchies.
Example: From Coding, recursive function calls (hw 2, mazes).
Example: From Data Compression, Huffman coding.
Example: From Phone Menus, sequence of instructions (press 1 if you want to make a reservation, press 2 if you want to check on a reservation, press 3 if you want to speak to an operator; pressing 1 then asks, press 1 if you are making a reservation within the US, press 2 if you are making a reservation within Canada, etc.)
Others?

A **BINARY** tree is a tree in which each node has at most two children. Children are named left child or right child.
Each node contains:

Data (key,value) pairs if a dictionary
Left child
Right child

```
typedef int TreeData;  // can change type with
                       // primitive or struct type
typedef struct TreeNodeTag TreeNode;

struct TreeNodeTag {
  TreeData value;    // value stored in node
  TreeNode * left;   // left child
  TreeNode * right;  // right child
};
```

The left child is a pointer to another tree node. The right child is a pointer to another tree node. For simplicity, we will store just one data item per node (but this data item could be a struct that contains key, value pairs).

CS 305: In-class Activity 11 (binary trees)

Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____ (R)_____ (P)_____(S)_____

Suppose T is a binary tree, where each node has at most two children.

1. Draw a binary tree with 6 nodes (labeled A, B, C, D, E, F) where there are exactly 3 leaves.

2. Draw a binary tree with 6 nodes where there is exactly 1 leaf.

Recall that the height of a tree is the number of nodes of the longest (deepest) path from the root to a leaf node.

For Q1, what is the height of your tree? _____

For Q2, what is the height of your tree? _____

Assume a binary tree has height H.

3. What is the maximum # of leaf nodes in a tree of height H? _____

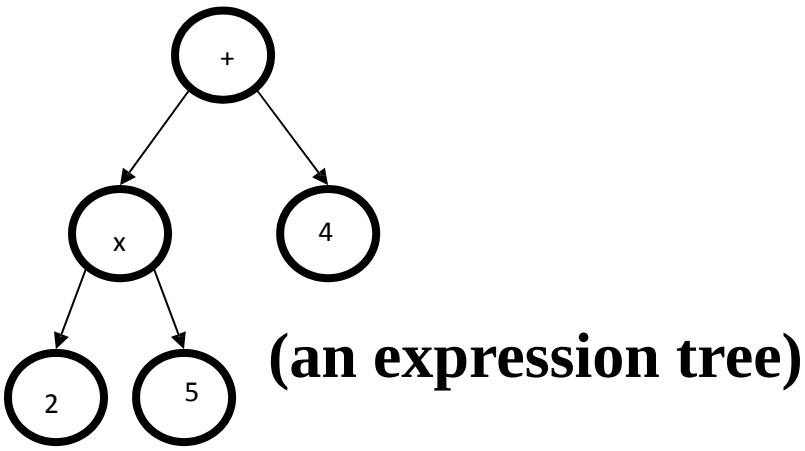
4. What is the maximum # of nodes in a tree of height H? (pack them in) _____

5. What is the minimum # of leaf nodes in a tree of height H? _____

6. What is the minimum # of nodes in a tree of height H? _____

In general, trees only speed things up if the tree is "full", meaning that we have close to the maximum number of nodes in a tree for a given height. A long, skinny tree does not outperform a linked list.

Here is a binary tree of arithmetic expressions.



We can traverse the tree in one of three ways:

- Preorder: examine node, left subtree, right subtree
- Inorder: examine left subtree, node, right subtree
- Postorder: examine left subtree, right subtree, node

If you get confused about the names, think about ***when*** the node is examined. First (pre), Second (in), Third (post).

In the example above, here are the orders of these traversals:

- Preorder: + x 2 5 4
- Inorder: 2 x 5 + 4
- Postorder: 2 5 x 4 +

Sometimes, the order of traversal does not matter for certain operations. For example, if you want to know how many nodes are in a tree, the way you traverse the tree does not impact your result. Any

traversal would be fine. Sometimes, though, order does matter. If you want to print a tree such that each level of a tree is indented further to the right, you would want to examine the tree in preorder fashion. If you are evaluating an expression tree, such as the one above, you would want to do this in postorder (get value of children before processing new operator).

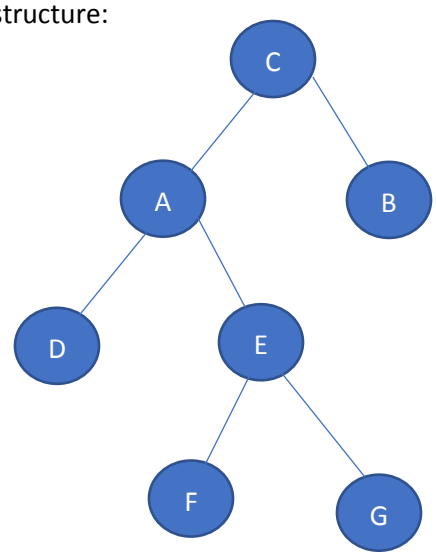
Here is the code for inorder traversal. Note that visit is also defined for visiting the node.

```
/* inorder
 * visits the nodes inorder (left, current, right) traversal
 */
void inorder(TreeNode * t) {
  if(t != NULL) {
    inorder(t->left);
    visit(t);
    inorder(t->right);
  }
}
```

7. Write the code to do preorder traversal:

```
void preorder(TreeNode * t) {





}
```

Suppose a tree has this structure:



8. What is the inorder traversal of this tree? _____

9. What is the preorder traversal of this tree? _____

10. Write the recursive function to return the number of leaves in a tree. This should be written recursively, since the tree data structure is recursive. Recall that if t is null, there are no leaves. If it's right child and left child are both null, t is a leaf, so return 1. Else, add together the recursive calls to process the left subtree and right subtree.

```
int numLeaves(TreeNode * t) {





}
```

11. (if time) Write a function to count the number of interior nodes with two children.

12. (if time) Assume a binary tree is traversed in-order and preorder. Here is the output. What does the tree look like?

Inorder: E F D B A C G H
Preorder: D E F G A B C H

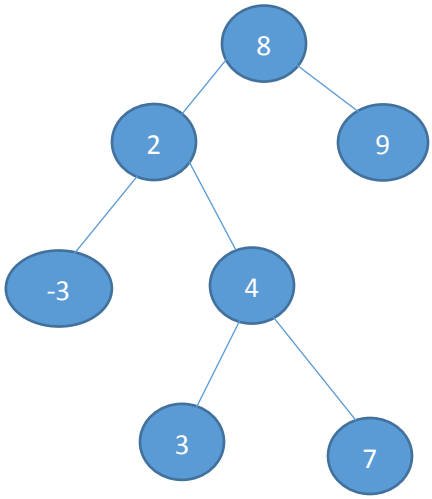
13. Does your group have any questions about binary trees?

CS 305: Binary Search Trees

A **BINARY SEARCH** tree is a binary tree in which the data (keys) are stored in order such that all nodes to the right of node N have keys bigger than N and all nodes to the left of node N have keys smaller than N.

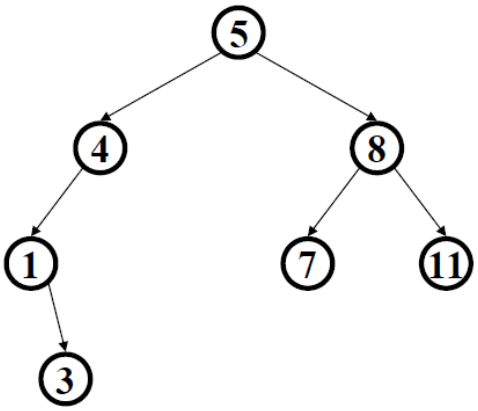
BSTs give us a good data structure to implement the dictionary ADT (insert, find, delete, create, print).

Here is a simple BST where the keys are ints:

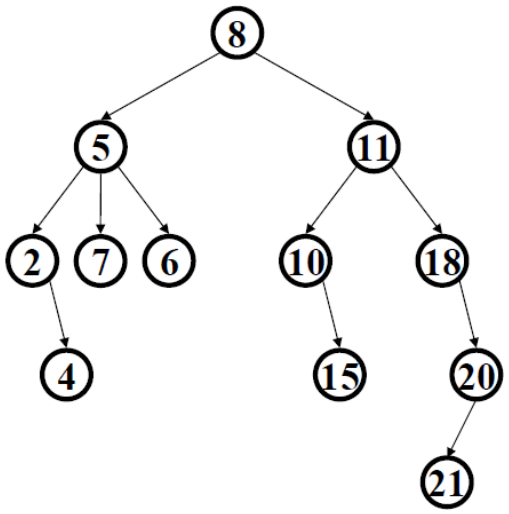


Choose any node in the tree. Its left subtree descendants are less than the node's value. Its right subtree descendants are greater than the node's value.

Is this a valid binary search tree?

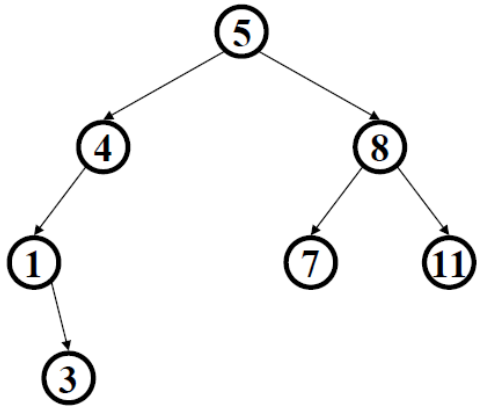


Is this a valid binary search tree?



Inserting new nodes

How do we insert new nodes into a BST?



Suppose we want to insert the value 6. Where does it go?
Now, we want to insert 0. Where does it go?
Now, we want to insert 9. Where does it go?

Inserting into a BST is quite simple. Insertions happen at the leaves. Here is an iterative version:

```
/* insert
 * inserts data item d into tree; note that this is a BST so it is ordered
 */
void insert(TreeData d, TreeNode ** tptr) {
```

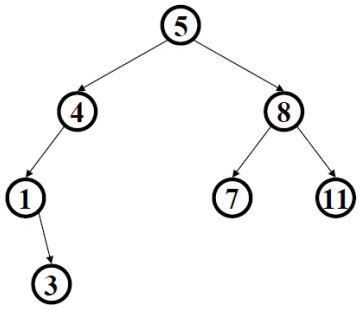
```
  // create new node for data
  TreeNode * toInsert = newTreeNode(d);
  TreeNode * curr = *tptr;
  if(curr == NULL) {
    *tptr = toInsert;  // make this the tree
    return;
  }
  // check value of t to see if new node should be to the right or left of curr
  while(curr != NULL) {
    if(d < curr->value) { // goes to left
      if(curr->left == NULL) {
       curr->left = toInsert;
       return;
      }
      // keep going left
      curr = curr->left;
    } else { // goes to right
      if(curr->right == NULL) {
       curr->right = toInsert;
       return;
      }
      // keep going right
      curr = curr->right;
    }
  }
}

/* newTreeNode
 * helper function, creates a new tree node with value d
 * returns the address of the new node
 */
TreeNode * newTreeNode(TreeData d) {
  TreeNode * toReturn = (TreeNode *) malloc(sizeof(TreeNode));
  toReturn->value = d;
  toReturn->left = NULL;
  toReturn->right = NULL;
  return toReturn;
}
```

Here is a recursive version to insert an item:

```
/* insertR
 * inserts data item d into tree; note that this is a BST so it is ordered
 * note: this function is written recursively
 */
void insertR(TreeData d, TreeNode **tptr) {
  if(*tptr == NULL) {
    *tptr = newTreeNode(d);
  } else if(d < (*tptr)->value) {
    insertR(d, &(*tptr)->left);
  } else {
    insertR(d, &(*tptr)->right);
  }
}
```

Finding keys

Now, how would we **find** an element in the tree?



Let's find 7. Start with the root. If the item is equal to 7, return true (or a pointer to this item). If the item you are looking for is > than the root, treat right subtree as root. Otherwise, treat left subtree as root. Keep applying this procedure until you hit a leaf.

What nodes are examined when looking for 7? _____

Now, look for 10. What nodes are examined when looking for 10? _____

You will implement the find function in lab.

Creating a new tree

Creating a new tree is pretty straightforward. A tree with no items is NULL.

```
TreeNode * tree = NULL;
```

To instantiate a tree with a list of items, we could do this:

```
/* createTree
 * creates a binary search tree with data stored in array a
 */
TreeNode * createTree(TreeData a[], int size) {
  if(size <= 0) {
    return NULL;
  }
  TreeNode * toReturn = newTreeNode(a[0]); // insert first item from list
  int i;
  for(i = 1; i < size; i++) {
    insert(a[i], &toReturn);
  }
  return toReturn;
}
```

An optional dictionary operation is size. Here is an implementation of size:

```
/* size
 * returns the number of nodes in the tree
 */
int size(TreeNode * t) {
  if(t == NULL) {
    return 0;
  }
  return 1 + size(t->left) + size(t->right);
}
```

CS 305: In-class Activity 12 (binary search trees)

Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____ (R)_____ (P)_____(S)_____

Suppose you create an empty tree and items are inserted as follows:

```
TreeNode * tree = NULL;
insert(5, &tree);
insert(8, &tree);
insert(2, &tree);
insert(1, &tree);
insert(10, &tree);
insert(7, &tree);
insert(9, &tree);
insert(12, &tree);
```

1. What does the BST look like?

2. What nodes are examined when finding 7?

3. What nodes are examined when finding 3?

4. Now, suppose this is a new tree and insertions are done in this order:

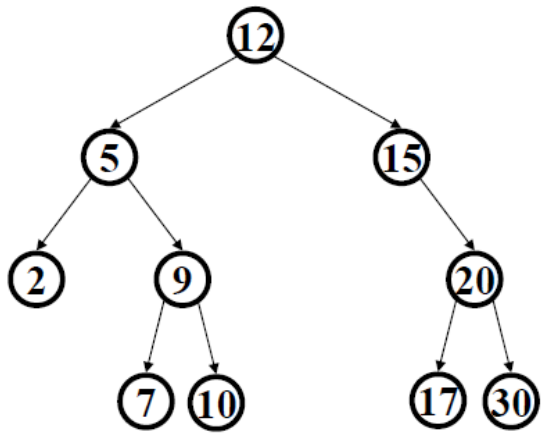
```
TreeNode * tree = NULL;
insert(1, &tree);
insert(3, &tree);
insert(4, &tree);
insert(6, &tree);
insert(7, &tree);
insert(8, &tree);
insert(9, &tree);
```

What does this tree look like?

There are ways to balance trees, so we get the win of searches happening closer to O(lgN) rather than O(N). You can read about specific kinds of trees, such as red-black trees and AVL trees that support tree rotations.

5. Give an insertion order of the same nodes in problem 4 that results in a full (complete) BST where most interior nodes have two children. Show the tree that results from this insertion order.

The final dictionary operation that we need to examine is delete. Given the tree below, how would you delete each of the nodes (assume the deletions are independent, so you are starting with the same tree prior to each deletion).



6. How would you delete 7?

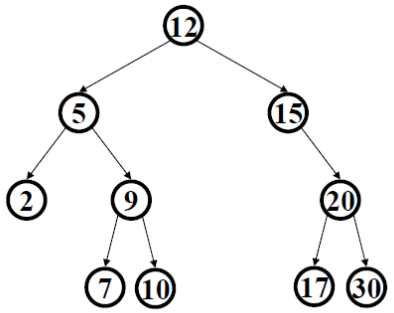
7. How would you delete 15?

8. How would you delete 5?

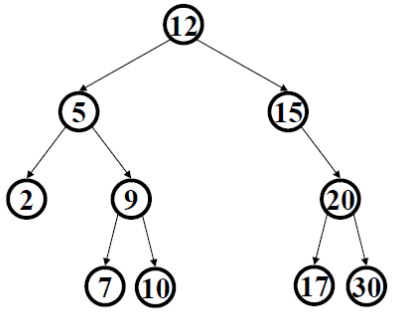
In general, here is the strategy for deletion:

Delete(D, T):
If Find(D, T) is false, do nothing.
If T is a leaf node, delete it and update its parent to point to null instead of T.
If T is an interior node and T has just a right child, delete T and update its parent to point to T's right child.
If T is an interior node and T has just a left child, delete T and update its parent to point to T's left child.
Else (T is interior with 2 children):
Find the next successor of T by traversing to T's right child and then going all the way to the leftmost leaf. This leftmost leaf is the next largest item in the tree. Copy the value of this leftmost leaf to T. If leftmost leaf does not have a right subtree, delete leftmost leaf with same procedure as leaf node above. If leftmost leaf has a right subtree, then delete with the same procedure as interior node with just a right child.

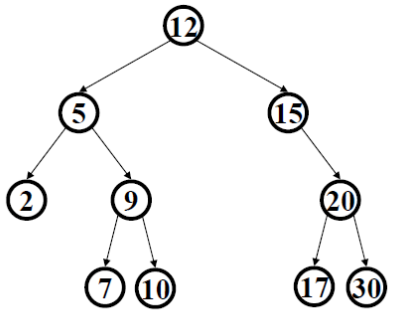
9. Delete node 5 with procedure above. Cross out nodes that are deleted and values that are updated.



10. Delete node 10 with procedure above. Cross out nodes that are deleted and values that are updated.



11. Delete node 15 with procedure above. Cross out nodes that are deleted and values that are updated.



Even though we are modeling BSTs with nodes having just one value, a (key, value) pair could be stored at each node, with the keys used as the comparison values when inserting, finding, and deleting.

12. What questions does your group have about binary search trees?

CS 305: Binary Search Trees Deletion and Analysis

As presented in the in-class activity, here is the overall idea for deleting a node in a BST:

Delete(D, T):
If Find(D, T) is false, do nothing.
If T is a leaf node, delete it and update its parent to point to null instead of T.
If T is an interior node and T has just a right child, delete T and update its parent to point to T's right child.
If T is an interior node and T has just a left child, delete T and update its parent to point to T's left child.
Else (T is interior with 2 children):
Find the next successor* of T by traversing to T's right child and then going all the way to the leftmost leaf. This leftmost leaf is the next largest item in the tree. Copy the value of this leftmost leaf to T. If leftmost leaf does not have a right subtree, delete leftmost leaf with same procedure as leaf node above. If leftmost leaf has a right subtree, then delete with the same procedure as interior node with just a right child.

*note that finding the previous predecessor could also work for the swap

Here is the C code:

```
/* delete
 * deletes node with data value d from the tree
 * note: passing in a pointer to the root of the tree in case the
 * root is updated
 */
void delete(TreeData d, TreeNode ** tptr) {
  TreeNode * curr = *tptr;
  TreeNode * found = NULL;
  TreeNode * parent = NULL;
  if(curr == NULL) { // no data in tree
    return;
  }
  parent = NULL;
  while(curr != NULL) {
    if(d == curr->value) {
      found = curr;
      break;
    } else if(d < curr->value) {
      parent = curr;
      curr = curr->left;
    } else {
      parent = curr;
      curr = curr->right;
    }
  }
  if(found == NULL) {
    return;   // not found in tree
```

d not found

```
}

// case 1: found is a leaf (just delete the node)
if(found->left == NULL && found->right == NULL) {
  printf("case 1\n");
  // update parent's correct child
  if(parent == NULL) {
    // found was the only node in the tree
    free(found);
    *tptr = NULL;
    return;
  }
  // parent is not null, so need to update its child to be null
  if(parent->left == found) {
    parent->left = NULL;
  } else if(parent->right == found) {
    parent->right = NULL;
  } else {
    printf("something went wrong: parent has invalid children\n");
    return;
  }
  free(found);
  return;
}
```

d found as a leaf node

```
// case 2: found is an interior node with just one child on right side
if(found->left == NULL) {
  printf("case 2:\n");
  // determine if found is left or right child of parent
  if(parent->left == found) {
    parent->left = found->right;
  } else if(parent->right == found) {
    parent->right = found->right;
  } else {
    printf("something went wrong: parent has invalid children\n");
    return;
  }
  free(found);
  return;
}
```

d is interior node with right subtree

```
// case 3: found is an interior node with just one child on the left side
if(found->right == NULL) {
  printf("case 3:\n");
  // determine if found is left or right child of parent
  if(parent->left == found) {
    parent->left = found->left;
  } else if(parent->right == found) {
    parent->right = found->left;
  } else {
    printf("something went wrong: parent has invalid children\n");
    return;
  }
  free(found);
  return;
```

d is interior node with left subtree

```
  }

  // case 4: found is an interior node with two children
  // find next larger element in tree (go right, then go left as far as
  // possible)
  printf("case 4:\n");
  TreeNode * traverse = found->right;
  TreeNode * traverseParent = found;
  // now go left until reach a node with no left
child
  while(traverse->left != NULL) {
    traverseParent = traverse;
    traverse = traverse->left;
  }
  // at this point traverse should be the next largest node in the tree
  found->value = traverse->value;  // put data in found
  // check if traverse is a leaf node
  if(traverse->left == NULL && traverse->right == NULL) {
    // leaf node -- just delete it
    if(traverseParent->left == traverse) {
      traverseParent->left = NULL;
      free(traverse);
    } else if(traverseParent->right == traverse) {
      traverseParent->right = NULL;
      free(traverse);
    } else {
      printf("something went wrong: parent of traversed node has invalid children");
      return;
    }
    return;
  }
  // traverse has a right subtree
  if(traverse->left == NULL && traverse->right != NULL) {
    if(traverseParent->left == traverse) {
      traverseParent->left = traverse->right;
      free(traverse);
    } else if(traverseParent->right == traverse) {
      traverseParent->right = traverse->right;
      free(traverse);
    }
  }
  return;
  // that is all the cases
}
```

d has two children; find successor and do swap and delete successor node

Complexity

Insert new data node: how long does this take given a tree with N nodes?

What is the “worst-case” tree? (long, skinny)

May need to compare new data with every other node in tree

O(N)

Find data in tree: how long does this take given a tree with N nodes?

What is the “worst-case” tree?

O(N)

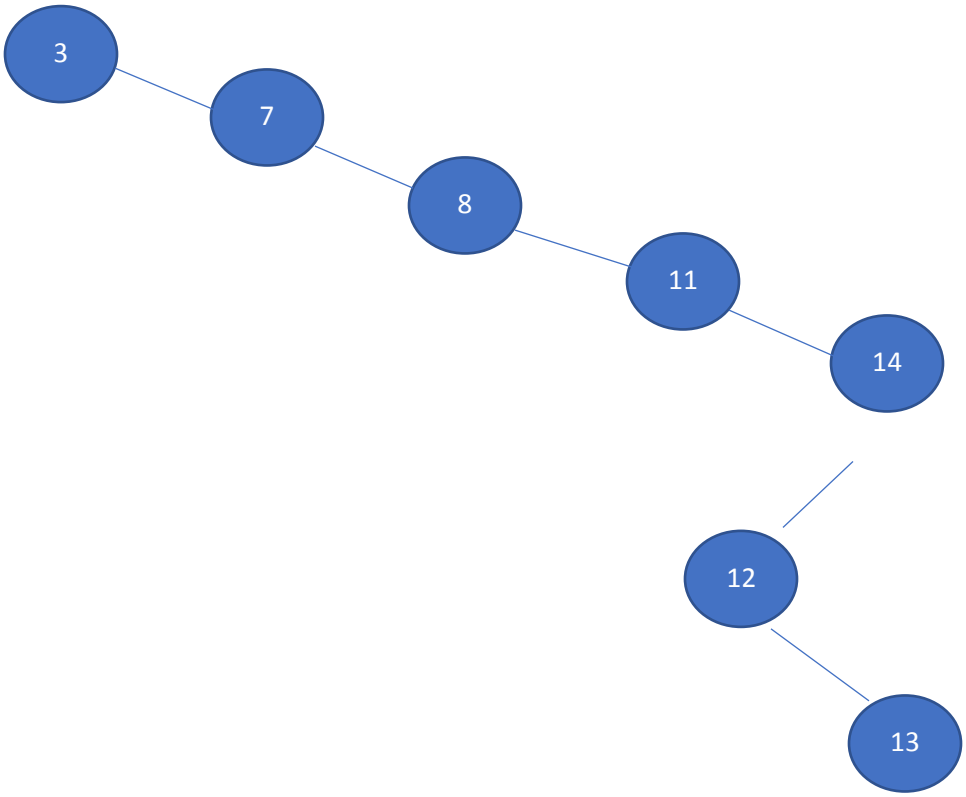
Delete node: how long does this taken given a tree with N nodes?

Find takes O(N) time; if not found, this is the complexity
Leaf case: O(N) to get to leaf, deleting leaf takes O(1); total: O(N)
Non-full interior node: O(N) to get partway down; updating pointer takes O(1); total: O(N)
Full interior node: O(N) to get partway down, O(N) to get to next successor, O(1) to copy data value, O(1) to delete leaf or do update pointer for non-full interior node

O(N)

How is this better than a linked list?

In the worst-case, we have a tree that looks like a list:



Traversing this tree takes O(N) time. Ugh. We really want “complete” trees, where most interior nodes have two children.

How could we re-order the insertions to make a complete tree with the data above? (hint: put 11 at the root)

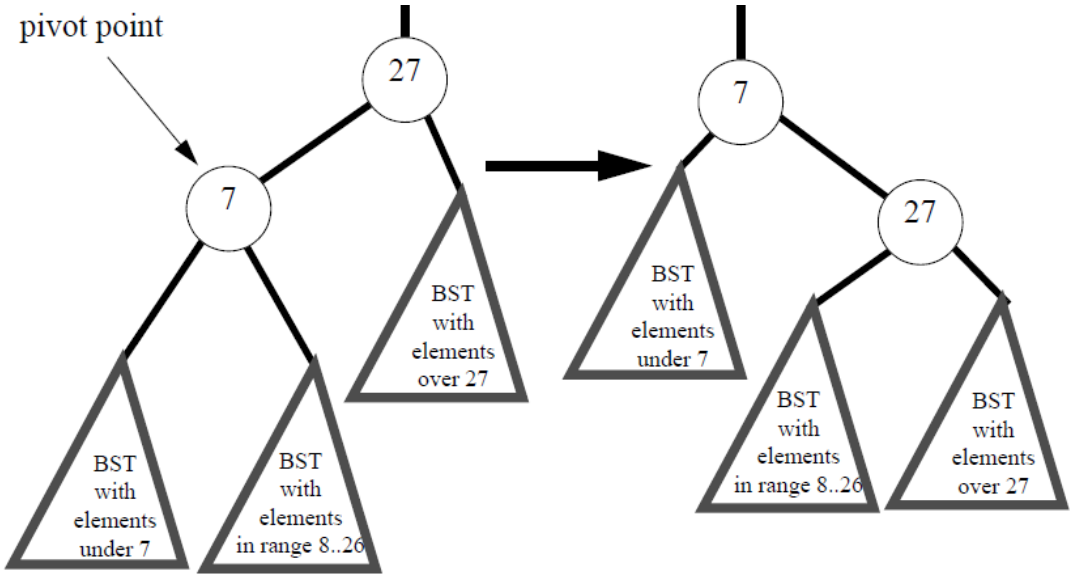
By creating complete trees, the height of the tree is closer to lgN instead of N for a tree with N nodes. That means find, insert, and delete run in time closer to O(lgN) instead of O(N).

How do we get trees that are complete?

Option 1: If we know all the data in advance, we can randomize the data before insertion. There is a high probability that this will give us a height lgN tree. (Try it…)

That does not work if we have an algorithm that uses trees where data is being inserted, searched for, and deleted over time. We need another approach.

Option 2: Tree rotations. These can be done in constant time and keep the tree in “sorted” order.



Think about a tree as a physical mobile, hung from the ceiling. The left tree is hung by the node with 27. When the tree becomes "out of balance", we pick a new root. In this case, 7 becomes the new root. If we just made 7 the root, we would have 3 children of 7. So, we need to fix the tree to keep it a binary tree. So, the right subtree of 7 becomes the left subtree of 27. 27 becomes the right subtree of 7.

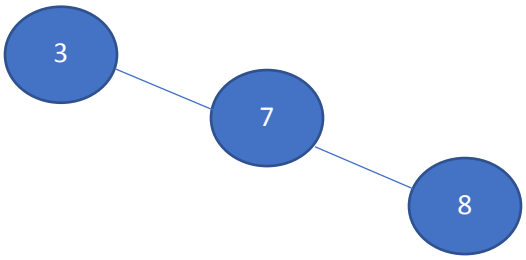
Note that these rotation operations are VERY FAST – just updating of pointers.

The tree above is heavy on the left side, so we rotated "to the right". A similar rotation can be done if the tree becomes heavy on the right side. In that case, we rotate "to the left".

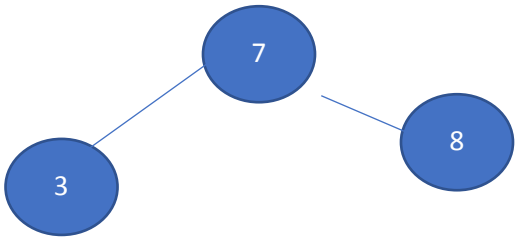
Now, we need to determine when to do the tree rotations. There are algorithms for doing this, which will be covered in CS 324 (Algorithms). For now, we will just do this by inspection – can see if the height of the left subtree is vastly different than the height of the right subtree.

Let's try it:
Tree insertions: 3, 7, 8, 11, 14, 12, 13

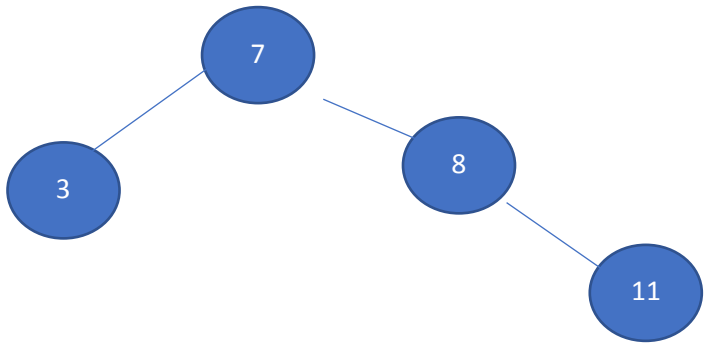
After 3 insertions:



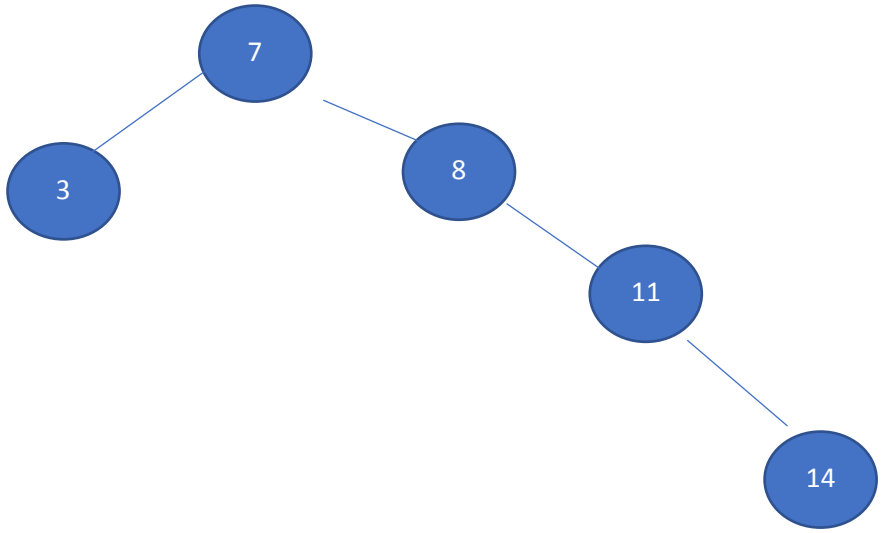
Tree is out of balance: left subtree of 3 has height 0, right subtree of 3 has height 2.
Rotate left at 7 (pivot point).



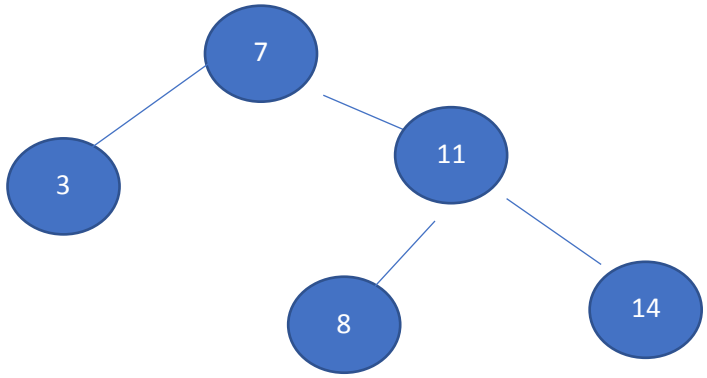
Now, insert 11.



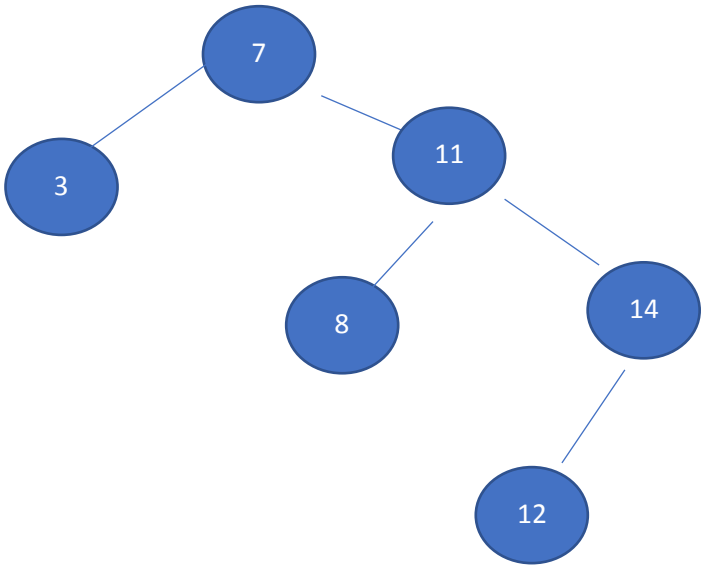
Now, insert 14.



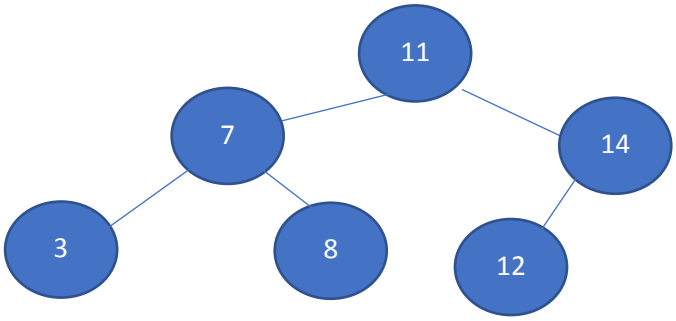
Tree is out of balance at 8 (left subtree has height 0; right subtree has height 2). Rotate left at 11.



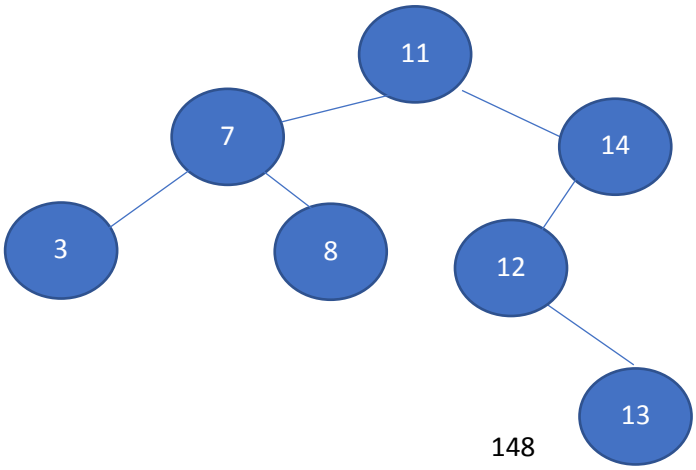
Now, insert 12.



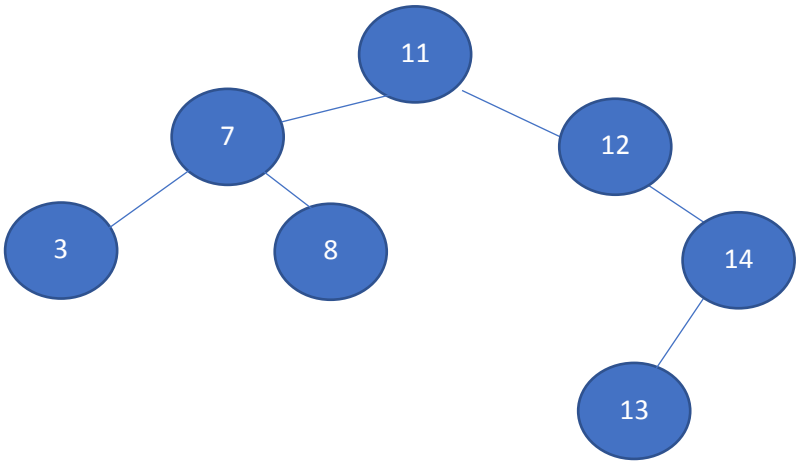
Tree is out of balance. Left subtree at 7 has height 1. Right subtree has height 3. Rotate left, making 11 new root.



Now, insert 13.



Tree is out of balance at 14 (left subtree has height 2; right subtree has height 0). Rotate right to make 12 root instead of 14.



Note that this last rotation did not help make it balance. Sometimes, this will happen with this rotation procedure when the tree has a zigzag in it.

My notes about binary search trees:

CS 305: Sorting

1. Think about your daily tasks. When do you put items “in order”?

2. What software keeps data in sorted order?
    For example, file lists in a directory/folder
    Others?

We have already seen one computational advantage of keeping data in sorted order. If data is stored in order, we can use binary search to find items.

A human advantage is that we can find items much more quickly if they are sorted.
    Imagine if your files were listed in a random order in a folder.
    Imagine if your phone listed your contacts in random order.
    Imagine if your email inbox stored your messages in random order.
    Imagine if the library stored the books on random shelves!

You have already seen sorting algorithms in CS 203, so this should be review for you.

3. How would you sort N items?

SELECTION SORT

Idea: Find smallest item from unsorted part of list and swap it into that place

Pseudocode:

```
SelectionSort(List L, int len):
        for i = 0 to i < (len – 2):
                p = position of smallest item from L[i] to L[len-1]
                swap L[p] with L[i]
```

Example:

Suppose L has the following:

5	8	9	3	4	10	2	6	

After 1st pass:

2	8	9	3	4	10	5	6	// 5 and 2 are swapped

After 2nd pass:

2	3	9	8	4	10	5	6	// 8 and 3 are swapped

After 3rd pass:

2	3	4	8	9	10	5	6	// 9 and 4 are swapped

After 4th pass:

2	3	4	5	9	10	8	6	// 8 and 5 are swapped

After 5th pass:

2	3	4	5	6	10	8	9	// 6 and 9 are swapped

After 6th pass:

2	3	4	5	6	8	10	9	// 8 and 10 are swapped

After 7th pass:

2	3	4	5	6	8	9	10	// 9 and 10 are swapped

DONE

Code:

```
/* selectionSort
 Parameters:
 - arr: the array
 - l: the left index of the range we're sorting
 - r: the right index of the range we're sorting
*/
void selectionSort(Item a[], int l, int r) {
  int i, j;

  // start at left, finding correct element
  for (i = l; i < r; i++) {
    // location of minimum element found so far on this iteration
    int minIdx = i;
    // loop through to find a
    for (j = i+1; j <= r; j++) {
        if (less(a[j], a[minIdx])) minIdx = j;
    }
    swap(&a[i], &a[minIdx]); // swap the "small" element into right spot
  }
}

/* swap -- exchanges the values of two pointers
   Parameters:
   - p1: pointer to first value
   - p2: pointer to second value
*/
```

```
void swap(Item *p1, Item *p2) {
  Item temp = *p1;
  *p1 = *p2;
  *p2 = temp;
}
```

Can also implement selection sort with a linked list to store the data.

INSERTION SORT

Idea: Keep the result list in sorted order; insert each element from the non-sorted list into the result list

Pseudocode:

InsertionSort(list L, int len):
    for i = 1 to (len – 1):
        insert L[i] among L[0]…L[i-1] such than L[0]…L[i] are sorted

Example:

Suppose L has the following:

5	8	9	3	4	10	2	6	
After 1st insertion:								
5	8	9	3	4	10	2	6	//note: 8 was inserted after 5
After 2nd insertion:								
5	8	9	3	4	10	2	6	//note: 9 was inserted after 8
After 3rd insertion:								
3	5	8	9	4	10	2	6	//note: 3 was inserted before 5
After 4th insertion:								
3	4	5	8	9	10	2	6	//note: 4 was inserted after 3
After 5th insertion:								
3	4	5	8	9	10	2	6	//note: 10 was inserted after 9
After 6th insertion:								
2	3	4	5	8	9	10	6	//note: 2 was inserted before 3
After 7th insertion:								
2	3	4	5	6	8	9	10	//note: 6 was inserted after 5

Code:

```
/* array-based insertion sort

   Parameters:
   - arr: the array
   - l: the left index of the range we're sorting
   - r: the right index of the range we're sorting
*/
void insertionSort(Item a[], int l, int r) {
  // starting with leftmost element, insert its right neighbor
  // into the correct place in the array
  int i;
  for (i = l; i < r; i++) {
    // the neighbor that we're inserting
    Item val = a[i+1];

    // find the correct spot for the element, shifting elements
    // over that are larger
    int j;
```

```
    for (j = i; j >= l && !less(a[j], val); j--) {
      a[j+1] = a[j];
    }

    // insert our element into the correct spot
    a[j+1] = val;
  }
}
```

Can also implement insertion sort using a linked list to store the data.

CS 305: In-class Activity 13 (selection and insertion sorts)

Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____ (R)_____ (P)_____(S)_____

Review the notes about selection sort and insertion sort. When everyone in the group is certain of the steps in the sorting routines, continue below.

1. Using *selection sort*, show the list contents after each swap is performed.

10	-3	0	8	4	-5	2	1

after first swap:

after second swap:

keep going…

2. Using insertion sort, show the list contents after each insertion is performed.

10	-3	0	8	4	-5	2	1

after first insertion:

after second insertion:

keep going…

3. Assume a list has N items. How many comparisons are performed to sort this list using selection sort? (note: the comparisons happen when finding the smallest element in the unsorted portion of the list)

_____

4. What is the complexity of selection sort? O(_____)

5. Assume a list has N items. How many comparisons are performed to sort this list using insertion sort? (note: the comparisons happen when finding where to insert the first element of the unsorted part of the list)

_____

6. What is the complexity of insertion sort? O(_____)

7. Does your group have questions about selection sort or insertion sort?

CS 305: Quicksort and Merge sort

Recall that selection sort and insertion sort run in $O(N^2)$ time. Is there a better (faster) way?

Quicksort

Idea: Quicksort relies on partitioning a list using a *pivot*. After partitioning, the items in the list to the left of the pivot are less than the pivot and items to the right of the pivot are greater than the pivot. Then, the two sublists are recursively quicksorted.

Here is the pseudocode:

```
Quicksort(List L, int low, int high):
    if(low < high):
        part = Partition(L, low, high)
        Quicksort(L, low, part-1)        // note: recursive call
        Quicksort(L, part+1, high)       // note: recursive call

Partition(List L, int low, int high):
    pivot = L[low]
    lastSmall = low
    for i = low+1 to high:
        if(L[i] < pivot)
            lastSmall++
            Swap(L, lastSmall, i)
    Swap(L, low, lastSmall)
    return lastSmall        // division point

Swap(List L, int a, int b):
    tmp = L[a]
    L[a] = L[b]
    L[b] = tmp
```

Example execution:

Assume a list has the following data:

36	10	54	14	83	25	60	72	44	31

Low is set to 0 and high is set to 9, so we need to partition:

pivot = 36

lastSmall = 0

10 is less than 36, so lastSmall = 1. We swap position 1 with 1.

54 is greater than 36, so we leave it.

14 is less than 36, so lastSmall = 2. We swap position 2 with 3.

Now, the array looks like:

36	10	14	54	83	25	60	72	44	31

83 is greater than 36, so we leave it.
25 is less than 36, so lastSmall = 3. We swap position 3 with 5.

Now, the array looks like:

36	10	14	25	83	54	60	72	44	31

60 is greater than 36, so we leave it.
72 is greater than 36, so we leave it.
44 is greater than 36, so we leave it.
31 is less than 36, so lastSmall = 4. We swap position 4 with 9.
Now, the array looks like:

36	10	14	25	31	54	60	72	44	83

Now, we just need to swap position low (0) with lastSmall (4):

31	10	14	25	36	54	60	72	44	83

Partition returns 4. So, all the elements to the left of L[4] are less than 36. All the elements to the right of L[4] are greater than 36. Note that the two sublists are not yet in sorted order, but the recursion will get them sorted.

Quicksort is then recursively called on the following two sublists:

31	10	14	25	36	54	60	72	44	83

After each of these is partitioned, we get:

25	10	14	31	36	44	54	72	60	83

//Note: 31, 36, and 54 are in place.
Then, the sublists to the left of 31, right of 31 (nothing), left of 54 (just 44), right of 54 are each partitioned and quicksorted.

This continues until we have empty lists to partition.

Complexity
What is the running time of quicksort given a list of N items?

What is the worst case in terms of the original data? list is already sorted
If the list is sorted, what is each pivot? smallest value in not yet sorted part of list

In practice, quicksort is “quick”. With randomized data, pivots chosen as the first element in the list are usually good choices. A good pivot splits its two sublists such that the sublists are about the same size. This means that quicksort is recursively called on the order of O(lgN) times and the partition step takes O(N) time. The total running time is then O(NlgN) in the *average case.* The *worst case* is $O(N^2)$.

Choosing good pivots

One can get “good” pivots by choosing a random element from the list rather than the first item in the list. Another option is to look at L[0], L[mid], and L[end] of the list L. Choose the middle value of these three values as the pivot.

Notes on quicksort

- Average case is O(N lgN); worst case is $O(N^2)$
- Can sort “in place” with a single array; no need to make copies of sublists. Memory usage is O(N) for quicksort.
- For small lists (N <= 10), the recursion function call overhead dominates, so some implementations may go into selection sort at this point.
- Quicksort can be implemented with linked lists, too.
- The implementation of partition can vary, as long as after the partition, all values less than the pivot are to the left of the pivot and all values greater than the pivot are to the right of the pivot.

Merge sort

(may be review from CS 203)

Idea: Merge sort is also recursive. Like quicksort, it divides the list and recursively calls merge sort on the smaller lists. Unlike quicksort, merge sort always divides the lists into two approximately equal sublists.

Here is the pseudocode:

```
MergeSort(List L, int low, int high):
        If (low < high):
                mid = (low + high) / 2
                MergeSort(L, low, mid)
                MergeSort(L, mid+1, high)
                Merge(L, low, mid, high)

Merge(List L, int low, int mid, int high):
        //note: L[low…mid] is sorted and L[mid+1…high] is sorted
        T = new list with size(L)
        i = low, j = mid+1, k = low
        while (i <= mid or j <= high)
                if(i > mid) T[k] = L[k], k++  // L[low…mid] is done being processed
                else if(j > high) T[k] = L[k], k++  // L[mid+1…high] is done being processed
                else if(L[i] < L[j]) T[k] = L[i], k++, i++ // choose item from left side
                else T[k] = L[j], k++, j++ // choose item from right side
        for i from low to high:
                L[i] = T[i]   // copy data back into list L
```

Example execution:

Suppose L has the following data:

32 11 24 6 2 8 17 10

The list is processed into sublists until we have lists of size 1. Once the lists are divided, then merge is called.

First division:

32 11 24 6 | 2 8 17 10

Second set of divisions:

32 11 | 24 6 | 2 8 | 17 10

Third set of divisions:

32 | 11 | 24 | 6 | 2 | 8 | 17 | 10

Now, the merges can happen. These happen with lists of length 0 or 1 first:

First set of merges:
11 32 | 6 24 | 2 8 | 10 17

Second set of merges:
6 11 24 32 | 2 8 10 17

Third set of merges:
2 6 8 10 11 17 24 32

The list is now sorted.

Complexity
Let's assume the original list has N items. How many levels of divisions are there? O(lgN)

How much "work" is done at each level of merge? Each item needs to be examined, so that is O(N)

So, the amount of work at each level is O(N) and the number of levels is O(lgN).
Total: O(NlgN)

Notes on merge sort:

- For unrestricted data (range of data and distribution of data in the list is not known ahead of time), merge sort is the fastest in terms of worst case complexity at O(NlgN).
- Merge sort has the overhead of recursion and function calls.
- Could be implemented with arrays or linked lists.
- Works with lists of any length. If list is an odd length, it gets split into lists where one list is one larger than the other list.

How could you make sorting **really slow?**

CS 305: In-class Activity 14 (quicksort and merge sort)

Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____ (R)_____(P)_____(S)_____

Review the process for quicksort and merge sort.

1. Suppose you are merge sorting the following list of numbers:
5 8 2 4 10 6 3 7 1

low = 0
high = 8

Let's look at the recursive calls as a tree:

MS(L, 0, 8)
MS(L, 0, 4) MS(L, 5, 8)
MS(L, 0, 2) MS(L, 3, 4) MS(L, 5, 6) MS(L, 7, 8)
MS(L, 0, 1) MS(L, 2, 2) MS(L, 3, 3) MS(L, 4, 4) MS(L, 5, 5) MS(L, 6, 6) MS(L, 7, 7) MS(L, 8, 8)
MS(L, 0, 0) MS(L, 1, 1)

Now, the merges:
Merge(L, 0, 0, 1): [5 8]
Merge(L, 0, 1, 2): [2 5 8]

// you complete the rest
Merge(L, 3, 3, 4):
Merge(L, 5, 5, 6):
Merge(L, 7, 7, 8):

Merge(L, 0, 2, 4):
Merge(L, 5, 6, 8):

Merge(L, 0, 4, 8):

2. Suppose you are quicksorting the same set of numbers:

5 8 2 4 10 6 3 7 1

5 is the pivot

What are the contents of the list after the partition step?

What value is returned from the partition step?

3. (if time) Now, run partition on the left and right sublists and keep going until the recursion stops.

4. Does your group have any questions about merge sort or quicksort?

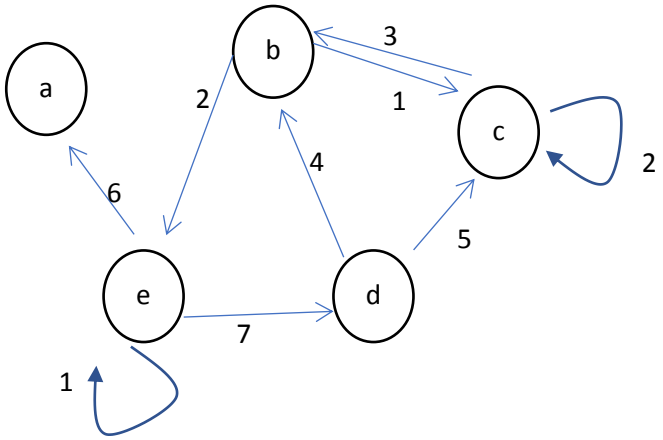
My notes about sorting:

CS 305: In-class Activity 15 (graphs)

Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____ (R)_____(P)_____(S)_____

A graph in computer science consists of vertices (also called nodes) and edges (also called links). Nodes are generally depicted as circles and labeled with a unique identifier. Edges are generally depicted as lines for undirected graphs and arrows for directed graphs. See how much you can glean from the graph below.



1. What are the vertices? _____

2. Is this graph directed or undirected? directed undirected

3. Is there an edge from b to d? yes no

4. Is there an edge from b to e? yes no

5. What is the cost of the edge from a to e? _____

Graphs with edges that have costs are called ***weighted*** graphs.

6. What is the lowest cost path from d to a? _____

The ***degree*** of a vertex is the number of edges that are incident (touch) the vertex. For directed graphs, each node has an "**in degree**" and an "**out degree**". The in degree is the number of edges that point to the vertex. The out degree is the number of edges that point away from the vertex. The in degree and out degree values do not need to match for a vertex. However, the total degree for a vertex of a directed graph is the in degree plus the out degree.

7. What is the out degree for e? _____

8. What is the in degree for e? _____

9. What is the out degree for b? _____

10. What is the in degree for b? _____

A ***path*** through a graph is a sequence of vertices where each successive pair of vertices has an edge in the graph.

11. Name two different paths, as a sequence of vertices, to get from b to c:
path 1: _____
path 2: _____

A ***connected*** graph has the property that every vertex is connected via a path to every other vertex in the graph.

12. Is the graph above connected? yes no

A ***cyclic*** graph has the property that it contains one or more cycles. A ***cycle*** is a path that starts and ends with the same vertex. An ***acyclic*** graph has no cycles. A directed graph that is acyclic is often called a **DAG** (directed acyclic graph).

13. Is the graph above cyclic? yes no

We can think of a graph as having |V| vertices and |E| edges where V is the set of vertices and E is the set of edges. The vertical bars mean "size of" in mathematical notation. So, |V| is the size of the vertex set.

14. What is |V| for the graph above? _____

15. What is |E| for the graph above? _____

Two vertices are ***adjacent*** if they are connected by an edge. In an undirected graph, you can think of adjacent vertices as neighbors. In an undirected graph, if x is adjacent to y, then y is adjacent to x. In a directed graph, a vertex x is adjacent to vertex y if there is an edge from y to x.

16. In the graph above, is e adjacent to b? yes no

17. In the graph above, is d adjacent to b? yes no

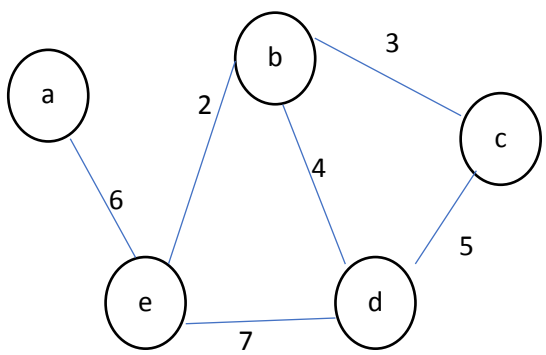
Just like trees, we have names for vertices in relationship to other vertices in a directed graph. A ***successor*** of a vertex v is any node n where there is a path from v to n. A ***predecessor*** of vertex v is any node n where there is a path from n to v.

18. Is a a predecessor of d? yes no

19. Is a a successor of d? yes no

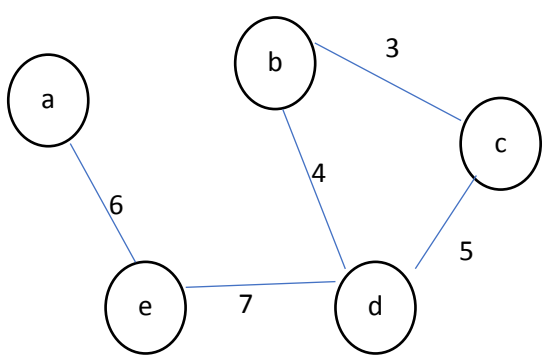
Undirected Graphs

Now, let's look at an undirected graph. An undirected graph may or may not be weighted (costs for edges). Recall that an undirected graph has links for edges instead of arrows.



20. What is different about this undirected graph versus a directed graph?

Note that the edges are links, so the cost between node b and c is the same as the cost between c and b. Here, the cost between b and c (and c and b) is 3. A path can be traversed in either direction of the link. So, in an undirected graph, if there is a path from vertex x to vertex y, there must also be a path from vertex y to vertex x (following the same set of vertices in reverse order). Because there are no more directed arrows, it no longer makes sense to think about cycles, successors, and predecessors in an undirected graph. An undirected graph could be connected, as follows:



Modeling with Graphs

Now that we have looked at graphs and the terminology, let's now focus on how graphs are useful for computation. Graphs are the basis for modeling relationships among entities. Here are some examples:

- The global internetwork of computers
  - Vertices are routers and computers; Edges are links between routers/computers
- Social networks
  - Vertices are people; Edges represent the "friend" relationship
- State diagrams
  - Vertices represent current state of computer; Edges represent transition to new states
  - Often the model used by event-driven programming
- Rubik's cube
  - Vertices represent current state of cube; Edges represent one twist to new configuration of the Rubik's cube
- Transportation networks
  - Vertices are intersections; Edges are roads/highways
- Airline flights
  - Vertices represent cities/airports; Edges are flights between the cities

21. What real life scenario (not from above list) could you model with a directed graph?

22. What real life scenario (not from above list) could you model with an undirected graph?

At this point, hopefully, your group realizes the importance of graphs in computing. Note that a graph can be thought of as a tree that can have cycles. A tree is also always a graph. A graph is not always a tree.

Representing Graphs

Now, to the implementation part of graphs. There are two standard ways to represent a graph in code. One representation uses an adjacency matrix. The other representation uses adjacency lists.

Adjacency Matrix

An adjacency matrix M is a |V| x |V| (2D array) of integers. You can number the vertices 0 to |V|-1 (or have a mapping of vertex names to numbers 0 to |V|-1). If there is no edge from vertex x and vertex y, then M[x][y] =0 (note that book uses infinity; either value works as long as the implementation knows

how to treat the value representing “no edge”). If there is an edge from vertex x to y with cost C, then M[x][y] = C.

If the graph is directed, M[x][y] does not need to equal M[y][x]. If the graph is undirected, M[x][y] = M[y][x]. If the graph edges have no costs, then M contains just 0’s and 1’s (and can store bits instead of ints).

Adjacency List

An adjacency list L is a list (linked list or array) of vertices. Each vertex stores a list of the vertices that are adjacent to it. If the edges have costs, then the adjacent vertices are stored as (V, C) pairs where V is the vertex and C is the cost.

Here is the adjacency matrix for the first graph in this handout:

	a	b	c	d	e
a	0	0	0	0	0
b	0	0	1	0	2
c	0	3	2	0	0
d	0	4	5	0	0
e	6	0	0	7	1

Here is the adjacency list for the first graph in this handout:

a -> {}
b -> {(c, 1), (e, 2)}
c -> {(b, 3), (c, 2)}
d -> {(b, 4), (c, 5)}
e -> {(a, 6), (d, 7), (e, 1)}

23. Here is the adjacency matrix for an undirected graph. Draw the circles and edges (links) below.

	a	b	c	d
a	0	2	1	0
b	2	0	3	1
c	1	3	0	4
d	0	1	4	0

24. Here is the adjacency list for an undirected unweighted graph. Draw the circles and edges (links) below.

a -> {c, e}
b -> {c, d}
c -> {a, b, d, e}

d -> {b, c, e}
e -> {a, c, d}

25. What is the maximum number of edges an undirected graph with 5 vertices can have?_____

26. What is the maximum number of edges a directed graph with 5 vertices can have? _____

A *sparse* graph is a graph that has few edges compared to the number of vertices in the graph. Draw a sparse graph here:

A dense graph is a graph that has lots of edges compared to the number of vertices in the graph. Draw a dense graph here:

27. Suppose you know your graph is sparse. Would you use the adjacency matrix or adjacency list representation?

matrix list

28. Suppose you know your graph is dense. Would you use the adjacency matrix or adjacency list representation?

matrix list

29. Suppose you want to determine if node a is connected to node c in a graph. Node a is mapped to position 0 in the matrix and list representations. Node c is mapped to position 2 in the matrix and list representations.

How long (worst-case) does it take to determine if a is connected to c using the matrix representation? O(_____)

How long (worst-case) does it take to determine if a is connected to c using the adjacency list representation? Assume the list of adjacent nodes is not in order. O(_____)

30. Does your group have any questions about graphs?

CS 305: Graphs (DFS and BFS)

Recall that a graph consists of vertices (V) and edges (E). As with a tree traversal, we may want to traverse a graph to see which vertices are reachable from a given start vertex. For example, suppose the graph consists of airports as the vertices and flights between airports as the edges. One might want to know if there exists a flight plan (could consist of multiple flights) from Portland to Moscow. The actual flight plan may be Portland -> Chicago -> Paris -> Moscow.

In HW 4, you programmed depth-first search and breadth-first search for the maze. While the maze was not implemented as a graph, you could turn the maze into a graph by having maze cells be vertices and form edges between adjacent (N, S, E, W) maze cells. We can do the same type of searching with graphs.

Information about graphs:
Assume a graph has a set V of vertices. An edge is denoted as (x, y) if x is connected to y. For now, we will work with unweighted graphs.

In the searches that we will consider, vertices have one of three colors:

white = not yet processed
gray = in process
black = finished being processed

A vertex v has the following properties:

v.color (white, gray, or black)
v.label (the name of the vertex)
v.discover (the time in which the vertex is discovered for DFS)
v.finish (the time in which the vertex is finished being processed for DFS)
v.parent (predecessor in search)
v.distance (the distance away from source, used for BFS)

Depth First Search (DFS)
Pseudocode for DFS starting from node v in a graph; also keeps track of reverse path back to v

```
time = 0            // used to set discovery and finish times

DFS_INIT(G):
       for all v in G
              v.color = white
              v.parent = NULL

DFS(v):
       visit v       // could print v.label
       v.color = gray
       time++
       v.discover = time
       for each edge (v, x):
```

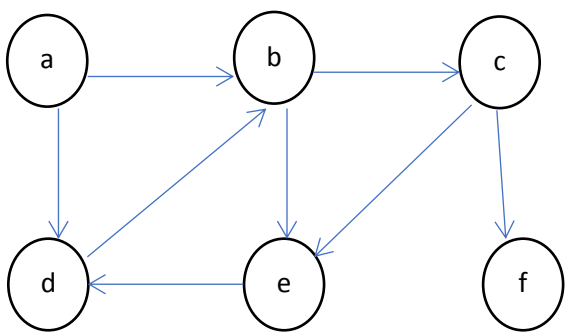
```
            if(x.color == white)
                x.parent = v
                DFS(x)
    v.color = black
    time++
    v.finish = time
```

In order to run DFS from all vertices in a graph G, just do the following:

```
DFS(G):
    for all x in V:
        x.color = white
    for all x in V:
        if(x.color == white)
            DFS(x)
```

Just like your HW, you could also implement DFS using a stack instead of recursion.

Example:



Let's do DFS from node a:

When choosing which edge to process first (successors of a node), let's do it in alphabetical order. For example, b < d in the alphabet, so node b will be processed before node d.

Node a's parent is set to NULL.
Node a gets colored gray and discover time is set to 1.
Node b's parent is set to a.
DFS(b) is called.
    b gets colored gray and discover time is set to 2.
    Node c's parent is set to b.
    DFS(c) is called.
        c gets colored gray and discover time is set to 3.
        Node e's parent is set to c.
        DFS(e) is called.
            e gets colored gray and discover time is set to 4.
            Node d's parent is set to e.

DFS(d) is called.

d gets colored gray and discover time is set to 5.

d has no edges to white nodes, so d gets colored black.

d's finish time is set to 6.

e has no edges to white nodes, so e gets colored black.

e's finish time is set to 7.

Node f's parent is set to c.

DFS(f) is called.

f gets colored gray and discover time is set to 8.

f has no edges to white nodes, so f gets colored black.

f's finish time is set to 9.

c has no edges to white nodes, so c gets colored black.

c's finish time is 10.

b has no edges to white nodes, so b gets colored black.

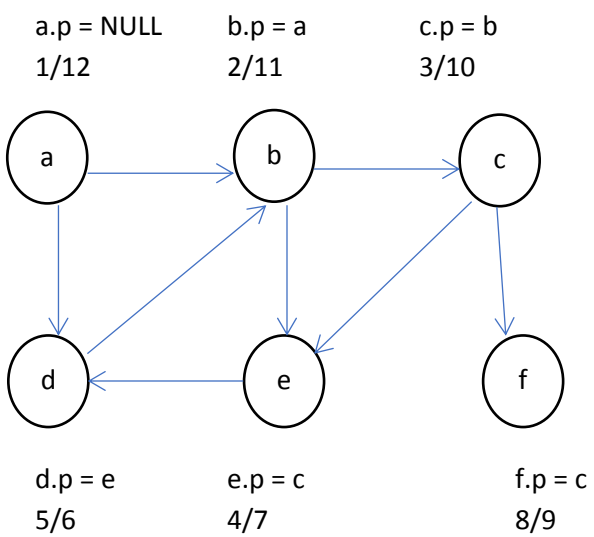
b's finish time is set to 11.

a has no edges to white nodes, so a gets colored black.

a's finish time is set to 12.

In the end, here is the graph showing the parents (p) and discover/finish times.

Final data:

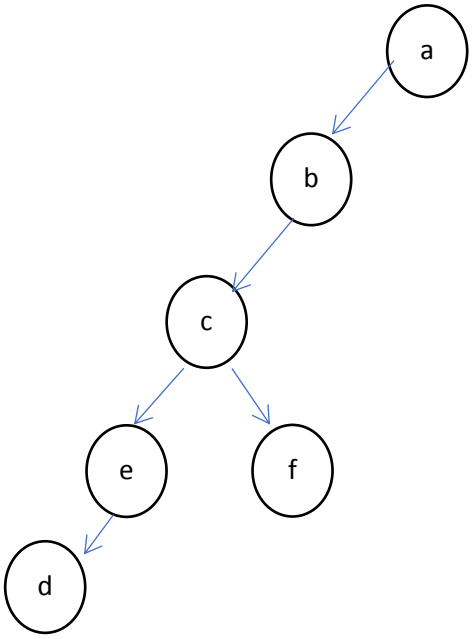


Suppose instead you did DFS from node f in the graph above. How many nodes would you find?

Suppose you did DFS from node c in the graph above. How many nodes would you find?

So, to traverse the entire graph, one must start a new search from any node colored white.

You can represent the traversal for DFS from node a as a tree:



Finding paths
Now, since we kept track of the parents during the DFS search, we can construct a path from node x to node y by starting with node y and following its parent, following its parent, etc. until we reach node x.

For example, to find the path from a to e:
Start with node e:
e's parent is c
c's parent is b
b's parent is a
STOP

Path is: a -> b -> c -> e [reverse order of parents]

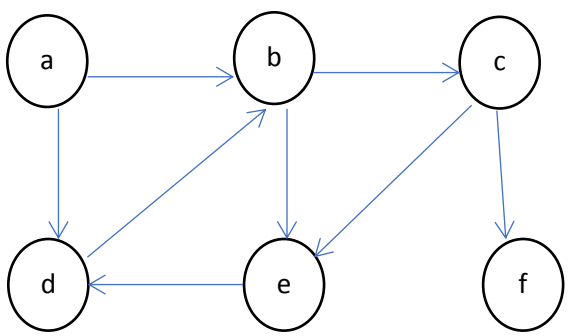
If the path-finding does not reach the desired node or reaches NULL, there is no path between the nodes.

Breadth First Search (BFS)

BFS uses a queue to maintain the order in which to process vertices. This search will first find all nodes 1 neighbor away from the source. Then, it finds all neighbors 2 away from the source. Then, 3 away, etc. until all nodes connected to the source are examined.

```
BFS(v):
        for all x in V:
                x.color = white
                x.distance = infinity
                x.parent = NULL
        v.color = gray
        v.distance = 0
        v.parent = NULL
        Q = new queue
        enqueue(v, Q)
        while(Q not empty):
                p = dequeue(Q)
                visit p              //could print p.label
                for each edge (p,x):
                        if(x.color == white)
                                x.color = gray
                                x.parent = p
                                x.distance = p.distance + 1
                                enqueue(x, Q)
                p.color = black
```

Example:



Let's do BFS from node a:

a is colored gray.
a's parent is NULL.
a's distance is 0.
Q = [a]
p = a
Put b and d into Q, so Q = [b d]
b's parent is a; d's parent is a
b and d are colored gray
distance of b and d is 1
a is colored black.
p = b
Put c and e into Q, so Q = [d c e]

c’s parent is b; e’s parent is b
c and e are colored gray
distance of c and e is 2
b is colored black.
p = d
(no white nodes from d)
d is colored black.
p = c
Put f into Q, so Q = [e f]
f’s parent is c
f is colored gray
distance of f is 3
c is colored black.
p = e
(no white nodes from e)
e is colored black.
p = f
(no white nodes from f)
f is colored black.

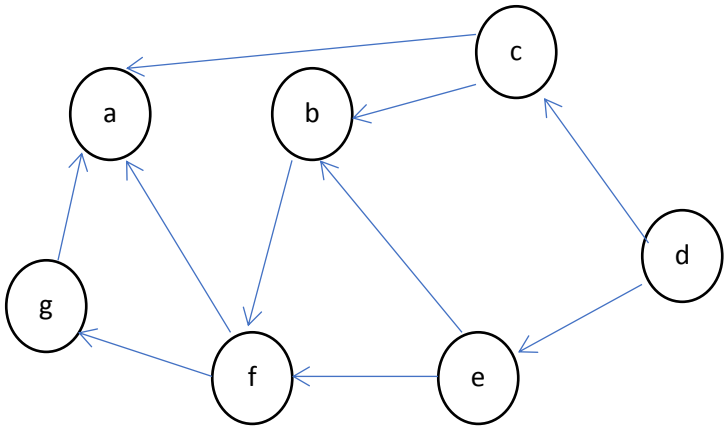
```
/* CS 305 Lab 10 code
 * Tammy VanDeGrift
 * Graph - matrix representation for a directed, weighted graph
 */

typedef enum {white, gray, black} COLOR;

typedef struct Graph {
  int V;     //number of vertices in G
  int ** M;  //2D array of ints, adjacency matrix
} Graph;

typedef struct DFS {
  COLOR color;  // white, gray, or black
  int parent;
  int discover;
  int finish;
} DFS;

int time = 0;  // note that this is a global variable (not great programming
               // practice) but need to have a global time count for DFS
               // it would be better to pass in a pointer to a variable that
               // stores the time that dfs could access and update, but
               // for the purpose of keeping this lab simple, we have a
               // global variable

// other functions for depth-first search in lab

/* dfsInit initializes the array of DFS structs, so that the parent
   is -1 for all nodes, color is white for all nodes, and discover and finish
   times are -1 */
DFS * dfsInit(Graph * g) {
  if(g == NULL || g->V <= 0) {
    time = 0;
    return NULL;
  }
  DFS * arr = malloc(sizeof(DFS) * g->V);
  int i;
  for(i = 0; i < g->V; i++) {
    arr[i].parent = -1;
    arr[i].color = white;
    arr[i].discover = -1;
    arr[i].finish = -1;
  }
  time = 0;
  return arr;
}
```

CS 305: In-class Activity 16 (DFS and BFS)

Write down the team's consensus answers to the questions on one sheet.

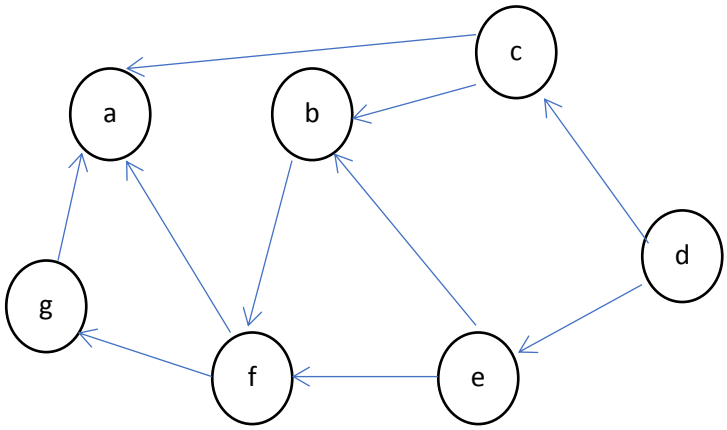
Names: (M)_____ (R)_____(P)_____(S)_____

Below is an unweighted graph of vertices and edges.



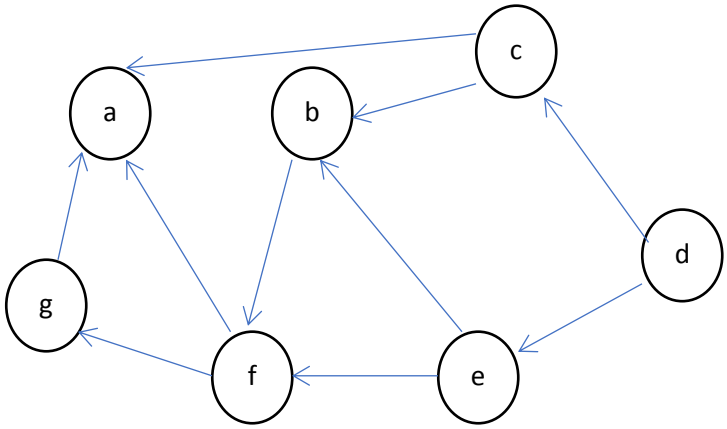
1. Do DFS from node d. When choosing which edges to process first, do them in alphabetical order. So, node c would be processed before node e.

For each node, indicate its parent (predecessor) and the discover/finish times. For example, d's parent is NULL and its discover time is 1.



2. Now, suppose you do DFS from node f. Which nodes will not be discovered? _____

3. Do BFS from d on the same graph:



Show the order in which vertices are explored: d _____

You may want to keep track of the queue here:

Q: d

4. For each node, indicate its distance from d:

a:

b:

c:

d: 0

e:

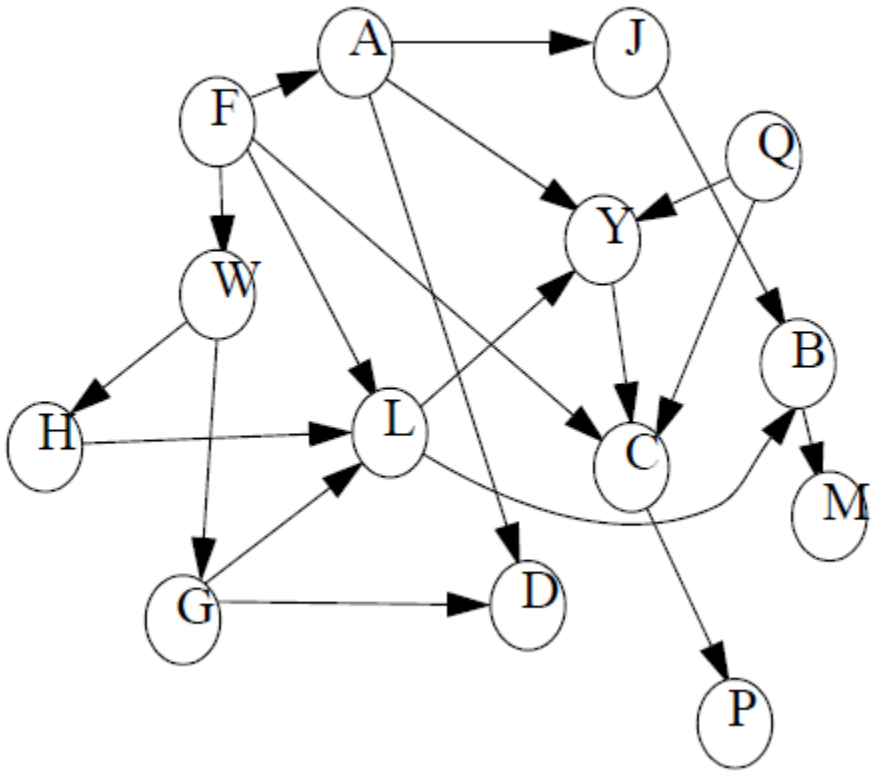
f:

g:

3. Does your group have questions about DFS or BFS?

CS 305: Topological Sort (if time)

Suppose we have a graph, such as the one below:



Assume each node is a course at a university and the arrows indicate prerequisites. Determining an order in which to take courses such that all prerequisites are satisfied is an example of a topological sort.

Note: we can only do a topological sort on a graph that is directed and acyclic (DAG). Another representation for a DAG is a partial order.

For example, in the graph above, a partial order could contain the sequence of vertices:
Q Y C P

Partial orders come up in many applications, such as scheduling and project management. You can think of a graph as modeling a set of constraints.

1. Task A must be done before Task B.
2. Task C must be done before Task D.
3. Task A must be done before Task D.

A. What is a partial order of these tasks?

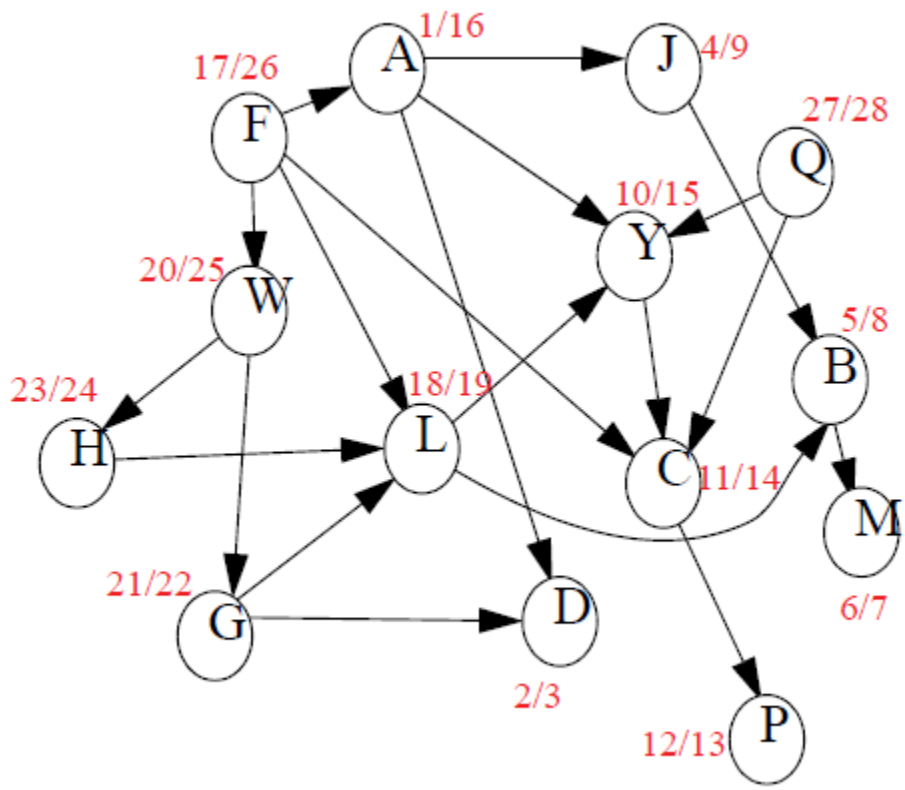
B. What is another partial order of these tasks?

C. What is a sequence that is NOT a partial order?

The partial order can help set the timeline for a project. In general, given a DAG, how can we find a partial order?

```
TOPOLOGICAL_SORT(G):
       L = empty list
       Do DFS(G)
              Just before finishing a visit to a vertex V, push V to the front of L

       Return L
```

Great – DFS just needs a small alteration to produce a topological sort. Here are the discover/finish times for DFS on the graph above:



We start with node A. Then, we discover node D. Just before D's finish, we push D to the list L.
L = [D]

Then, node J is visited, then node B, then node M. Now, we push M to the front of the list L.
L = [M D]

Now, just before finishing B, we push B and then push J.
L = [J B M D]

Now, we visit Y, then C, then P. So, we push them to L:
L = [Y C P J B M D]

Now, we visit A:
L = [A Y C P J B M D]

We continue with DFS with node F (next alphabetical node not yet explored):
We visit F and then L:
L = [L A Y C P J B M D]

Then visit W and G:
L = [G L A Y C P J B M D]

Then visit H, back to W, back to F:
L = [F W H G L A Y C P J B M D]

We continue with DFS with node Q (next alphabetically smallest node):
We visit Q and its neighbors are already colored black, so we push Q to L:
L = [Q F W H G L A Y C P J B M D]

The list L has the topological sort of the graph's vertices. If the graph represents tasks and an arrow between X and Y means task X must happen before task Y, then L gives an ordering of all the tasks that satisfies the graph.

This algorithm processes each vertex just once to insert it into the topological sort.

CS 305: Dijkstra's Algorithm

Suppose you have a weighted (no negative weights) graph G. You want to know the shortest paths from some node S in G to all other nodes in G. Dijkstra's algorithm to the rescue!

When might this be useful?

- transportation network; shipping goods from Amazon's warehouse in Las Vegas to several different cities; edges are point-to-point shipping costs; trying to minimize overall shipping costs
- routing networks for computers; there may be several ways to get from a gateway in Portland to a gateway in Amsterdam. With the current network traffic, what is the best computer network route from Portland to Amsterdam?
    - A major network routing protocol uses Dijkstra's algorithm

How does Dijkstra work?

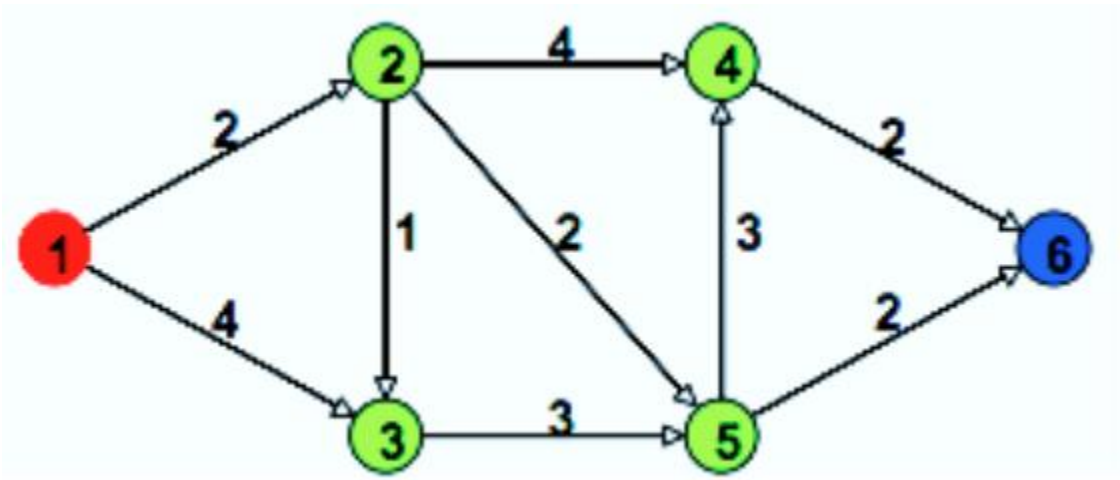
Each node will have the following information:

- color (white or black; white if not yet processed)
- dValue (distance from source to the node; gets updated as algorithm executes)
- pred (predecessor node along best path from source to node)

```
// G is a graph, W is the weights for the edges, S is the source vertex
Dijkstra(G, W, S):
       Initialize(G, S)
       Put all vertices of G into Q
       While Q not empty:
              U = getMin(Q)       //Q is a priority queue
                                  //returns white node with smallest dValue
              for each edge (U, X):
                     if(X.dValue > [U.dValue + W(U, X)]
                            X.dValue = U.dValue + W(U, X)
                            X.pred = U
              U.color = black

       For each V in G:
              Print V.dValue as the total cost to get from S to V

Initialize(G, S):
       For each V in G:
              V.dValue = infinity
              V.color = white
              V.pred = NULL
       S.dValue = 0
```

Example:

Suppose you have the following graph:



Suppose we run Dijkstra's algorithm from vertex 1:

Here is a table showing the values of each vertex's initial data:

	1	2	3	4	5	6
color	White	White	White	White	White	White
dValue	0	Infinity	Infinity	Infinity	Infinity	Infinity
Pred	NULL	NULL	NULL	NULL	NULL	NULL

All vertices are put into the priority queue Q. The node with the smallest dValue is node 1. So, we remove node 1 from Q.

Node 1's neighbors: 2 and 3

	1	2	3	4	5	6
color	Black	White	White	White	White	White
dValue	0	2	4	Infinity	Infinity	Infinity
Pred	NULL	1	1	NULL	NULL	NULL

Now, 2 is removed from Q:

Node 2's neighbors: 3, 4 and 5

	1	2	3	4	5	6
color	Black	Black	White	White	White	White
dValue	0	2	3	6	4	Infinity
Pred	NULL	1	2	2	2	NULL

Now, 3 is removed from Q:

Node 3's neighbors: 5

	1	2	3	4	5	6
color	Black	Black	Black	White	White	White
dValue	0	2	3	6	4	Infinity
Pred	NULL	1	2	2	2	NULL

*Note that node 5 is not updated, since dValue is 4. Using node 3, the dValue would be 6.

Now, 5 is removed from Q:

Node 5's neighbors: 6

	1	2	3	4	5	6
color	Black	Black	Black	White	Black	White
dValue	0	2	3	6	4	6
Pred	NULL	1	2	2	2	5

Now, either 4 or 6 could be removed, since they have equal dValues. Let's remove 4:

Node 4's neighbors: 6

	1	2	3	4	5	6
color	Black	Black	Black	Black	Black	White
dValue	0	2	3	6	4	6
Pred	NULL	1	2	2	2	5

*Note that node 6 is not updated, since dValue is 6. Using node 4, the dValue would be 8.

Now, 6 is removed from Q:

Node 6's neighbors: none

	1	2	3	4	5	6
color	Black	Black	Black	Black	Black	Black

dValue	0	2	3	6	4	6
Pred	NULL	1	2	2	2	5

How do we get the actual paths?

Suppose we want to know the shortest path from node 1 to node 6.

We start with node 6. Its predecessor is 5. We look at node 6. Its predecessor is 2. We look at node 2. Its predecessor is 1. So, the path from node 1 to node 6 is:

1 -> 2 -> 5 -> 6

Some notes about Dijkstra:

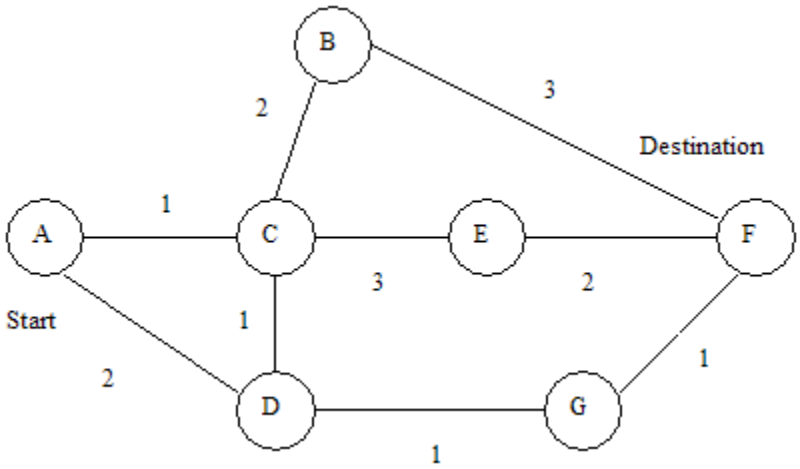
- Algorithm runs in O(NlgN) time for where |V| + |E| = N.
- Algorithms for single source and single destination are computationally as expensive to doing minimal cost of paths from single source to all other nodes. [Why? The optimal path from the source to the destination may go through all other nodes.]
- Graphs can be directed or undirected.
- Graphs cannot have negative weights. There is another algorithm: Bellman-Ford that works with negative weights and detects if a negative cycle exists. Think about that: if there is a negative cost cycle, we would want to execute that cycle infinity number of times to continue to minimize the path.
- If a graph has negative weights, can add a constant value to all edge weights to get the costs >= 0.

CS 305: In-class Activity 17 (Dijkstra)

Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____ (R)_____(P)_____(S)_____

Perform Dijkstra's algorithm on the graph below. Start with node A and determine the shortest path to node F.



	A	B	C	D	E	F	G
color	White	White	White	White	White	White	White
dValue	0	Infinity	Infinity	Infinity	Infinity	Infinity	Infinity
Pred	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Choose node: _____
Its neighbors: _____

	A	B	C	D	E	F	G
color							
dValue	0						
Pred							

Keep going…

You can edit the above table as the algorithm proceeds if you wish.

What is the shortest path from A to F?

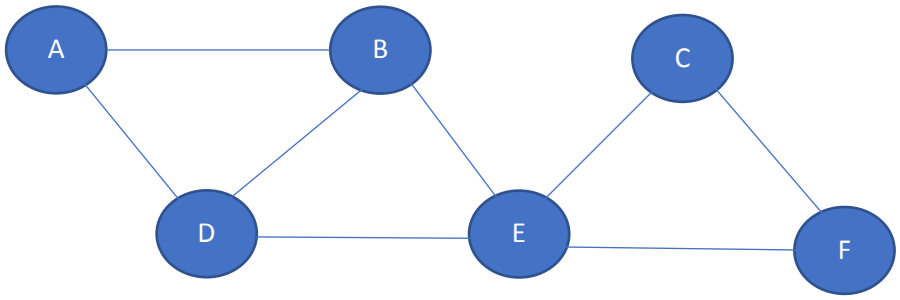
A ->

2. Does your group have questions about Dijkstra's Algorithm?

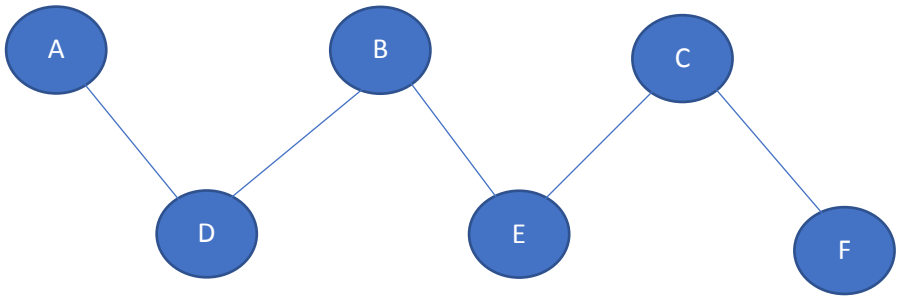
CS 305: Spanning Trees

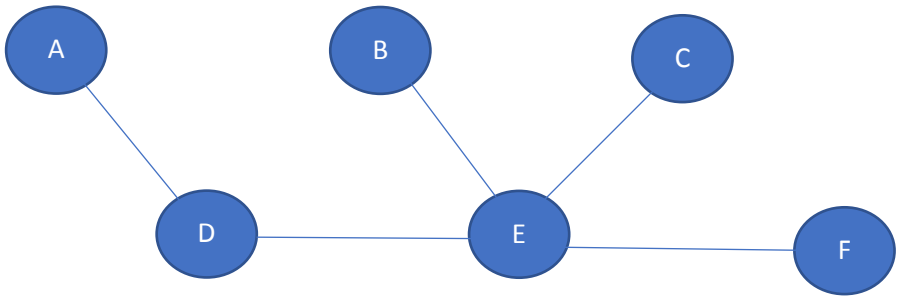
Given an undirected graph G, we can create a *spanning tree*. A spanning tree is a subgraph of G that contains all the vertices of G, but only edges that form a tree.

For example, here is an undirected graph:



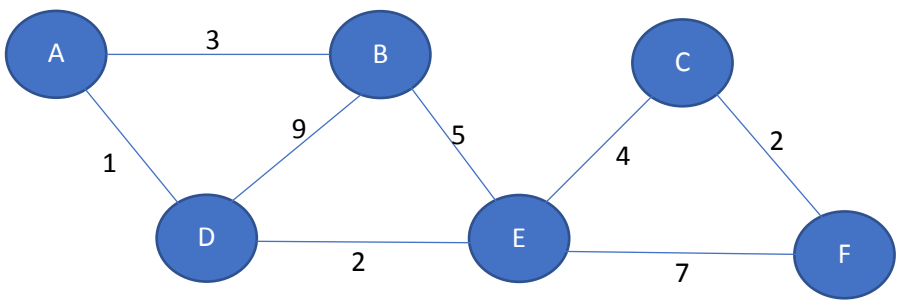
Here are possible spanning trees:





Given a graph of N nodes, the spanning tree with contain all N nodes and N-1 edges. Can you create a different spanning tree from the graph above?

Now, let's consider a weighted, undirected graph:



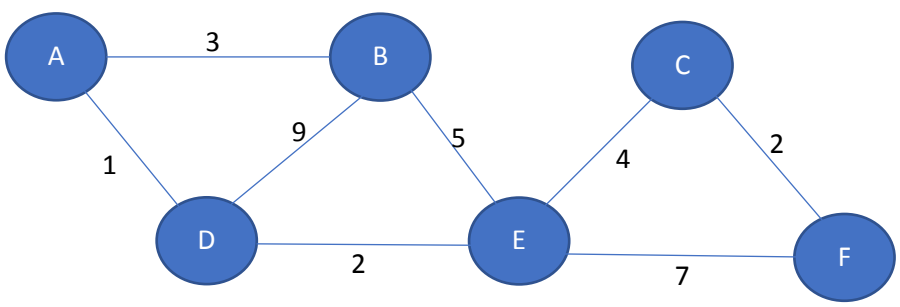
What is the spanning tree that contains edges with the minimum total cost?

This is a minimum spanning tree. One of the bridge-learning algorithms in networking determines the minimum spanning tree for forwarding packets. In a network that has redundancy (like a graph with cycles), one does not want messages traveling across cycles. Using only connections for a minimum spanning tree guarantees that packets won't loop endlessly in a network.

There are two main algorithms for finding the minimum spanning tree from a weighted, undirected graph.

Prim's Algorithm

```
Prim(G, W,  s):
       InitPrim(G, s)
       Add all vertices of G to Q (Q is a priority queue)
       While Q is not empty:
              p = get_Min(Q) // returns white node with minimum cost
              if(p.cost is infinity), break;   // no reachable vertices from p
              p.color = black
              // p is added to the minimum spanning tree
              for each edge (p, x):
                     if(W(p, x) < x.cost):
                            x.cost = W(p, x)
                            x.parent = p
       for each v in G:
              if (v != s):
                     print edge (v.parent, v) and v.cost

InitPrim(G, s):
       for each v in G:
              v.cost = infinity
              v.parent = null
              v.color = white
       s.cost = 0
```



Let's do Prim using A as the source node. Note that at each iteration, we choose the least cost edge to add to the tree that continues growing the tree. Before looking at the details, let's just work through it at a high-level.

Start with A. Can reach B and D and maintain the tree. Choose edge to D to add, since that edge has cost 1 and the edge to B has cost 3.

Now, {A, D} are part of the tree. The edges that can be added are: (A, B), (D, B), and (D, E). Since (D, E) has least cost, we add (D, E) to the tree.

Now, {A, D, E} are part of the tree. Edges that can be added: (A, B), (D, B), (B, E), (C, E), (E, F). Edge (A, B) has the least cost, so we add (A, B) to the tree.

Now, {A, D, E, B} are part of the tree. Edges that can be added: (B, E), (D, B), (B, E), (E, C). Edge (C, E) has the least cost, so we add (C, E) to the tree.

Now, {A, D, E, B, C} are part of the tree. Edges that can be added: (B, E), (D, B), (B, E), (C, F), (E, F). (C, F) has the least cost, so we add (C, F) to the tree.

Now, all vertices are part of the tree and we stop. Final edges: (A, D), (D, E), (A, B), (C, E), (C, F).

With the structs. Minimum node selected is highlighted.

	A	B	C	D	E	F
Parent	Null	Null	Null	Null	Null	Null
Cost	0	Inf	Inf	Inf	Inf	Inf
Color	W	W	W	W	W	W

A is min white node:

	A	B	C	D	E	F
Parent	Null	Null	Null	Null	Null	Null
Cost	0	Inf	Inf	Inf	Inf	Inf
Color	W	W	W	W	W	W

Neighbors of A: B, D

	A	B	C	D	E	F
Parent	Null	A	Null	A	Null	Null
Cost	0	3	Inf	1	Inf	Inf
Color	B	W	W	W	W	W

Neighbors of D: A, B, E

	A	B	C	D	E	F
Parent	Null	A	Null	A	D	Null
Cost	0	3	Inf	1	2	Inf
Color	B	W	W	B	W	W

Neighbors of E: C, F

	A	B	C	D	E	F
Parent	Null	A	E	A	D	E
Cost	0	3	4	1	2	7
Color	B	W	W	B	B	W

Neighbors of B: A, D, E

	A	B	C	D	E	F
Parent	Null	A	E	A	D	E

Cost	0	3	4	1	2	7
Color	B	B	W	B	B	W

Neighbors of C: E, F

	A	B	C	D	E	F
Parent	Null	A	E	A	D	E
Cost	0	3	4	1	2	7
Color	B	B	B	B	B	W

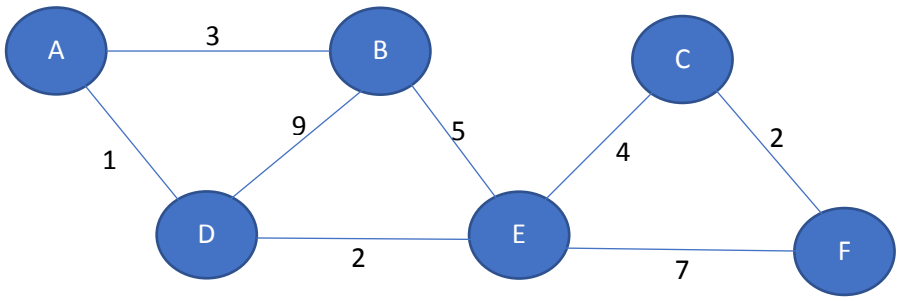
Neighbors of F: C, E

	A	B	C	D	E	F
Parent	Null	A	E	A	D	C
Cost	0	3	4	1	2	2
Color	B	B	B	B	B	B

Edges of MST: (A, B), (C, E), (D, A), (E, D), (F, C)
[just look at the parent and the node value in the final struct]

Kruskal's Algorithm

```
Kruskal(G):
        For each v in V, create a tree of v only
        Sort edges by non-decreasing weight, call sorted set E
        For each edge (u, v) in E:
                If u and v belong to different trees, add (u, v) to MST
```



Each vertex is own tree:

A B C D E F

Edges in sorted order:

(A, D), (D, E), (C, F), (A, B), (E, C), (B, E), (E, F), (D, B)

MST:

(A, D)
(D, E)
(C, F)
(A, B)
(E, C)
// (B, E) not added since this would create a loop
// (E, F) not added since this would create a loop
// (D, B) not added since this would create a loop

My notes about graphs:

Exam #3 Review Guide and Practice Questions

The third CS 305 exam will be: **Friday, April 17** at 1:35pm. The exam length is 55 minutes.

The exam focuses on topics covered in lectures from March 16 through April 15. Although the exam's focus is not on the C language, it is expected that you can read and write C code that you learned prior to exam 1. You should review your coursepack, in-class activities, data structures textbook, prelabs, labs, HW 4, HW 5, and HW 6.

Here are topics:

- Dictionary ADT
- Trees
    - Traversals (preorder, postorder, inorder)
    - Terminology, such as parent, child, leaf, root, descendant, ancestor, height
    - Applications of trees
    - Recursive functions on trees
- Binary Search Trees
    - insertion
    - deletion
    - finding items (find a key, find min in tree, find max in tree)
    - rotations (balancing)
    - Recursive functions on trees, such as height and num-nodes
- Sorting
    - Selection sort
    - Insertion sort
    - Quicksort
    - Merge sort
    - Complexity of sorting routines
- Graphs
    - Terminology, such as vertex, edge, directed, undirected, degree, predecessor, successor, connected, acyclic, weighted
    - Searches
        - Depth-first search
        - Breadth-first search
    - Applications of graphs
    - Representations
        - Adjacency Matrix
        - Adjacency List
    - Dijkstra's algorithm: single source, shortest paths
    - Topological Sort
    - Minimum Spanning Trees

- Kruskal's Algorithm
- Prim's Algorithm

You can be expected to write code on the exam, read code, and answer questions about code. You may be asked to find syntax errors and run-time errors in code. Code that you need to read and process will be provided on the exam.

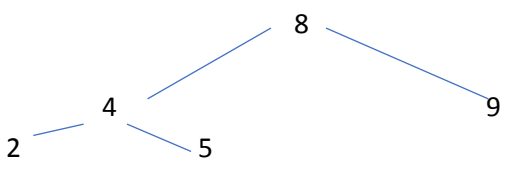
You will be allowed one 8"x11.5" crib sheet (both sides) to use while taking the exam. Your crib sheet can be hand-written or typed. No other aids are permitted (computers, calculators, headphones, music, phones).

Some of the class time on Apr 12 will be set aside for review for the exam. Come to class with questions you have about the material. The remaining portion of this review guide has practice questions to prepare for the exam. Be sure to study all topics above to prepare for the exam (not every topic has a practice exam question below).

!!! Remember: as you study, you can write small programs to see how the code compiles and executes. You may use the lab files to experiment !!!

SAMPLE QUESTIONS

1. Suppose the following binary search tree is created:



a. Show the tree after 7 is inserted.
b. Show the tree after 12 is inserted.
c. Show the tree after 3 is inserted.

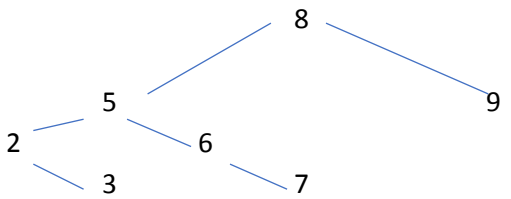
2. Suppose the following items are inserted into a binary search tree in this order:

2 5 3 8 10 1 4 7

Draw the BST.

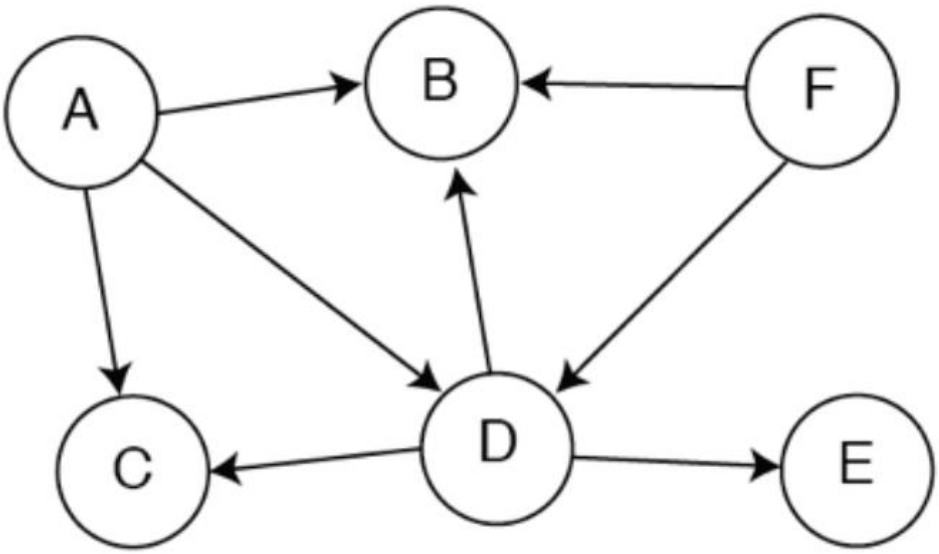
3. Suppose the value 3 is deleted from the tree in #2. What does the BST look like after 3 is deleted?

4. Suppose a BST looks like:



Rotate the tree to the right, so that 5 is the new root of the tree. What does the tree look like?

5. Consider the graph below:



A. Is this graph directed or undirected?

B. Perform depth-first search from node A in the graph. Show the discovery/finish times for each node on the node itself. If there is an option for which node to discover first, break ties by which node is first in alphabetical order. For example, if either B or C could be discovered at the same time, choose B to discover first.

C. Perform breadth-first search on the graph starting at node A. Show the order of the nodes as they are discovered by breadth-first search.

6. I have a binary tree (not necessarily a binary search tree). When I print the elements using preorder traversal, this is the order:

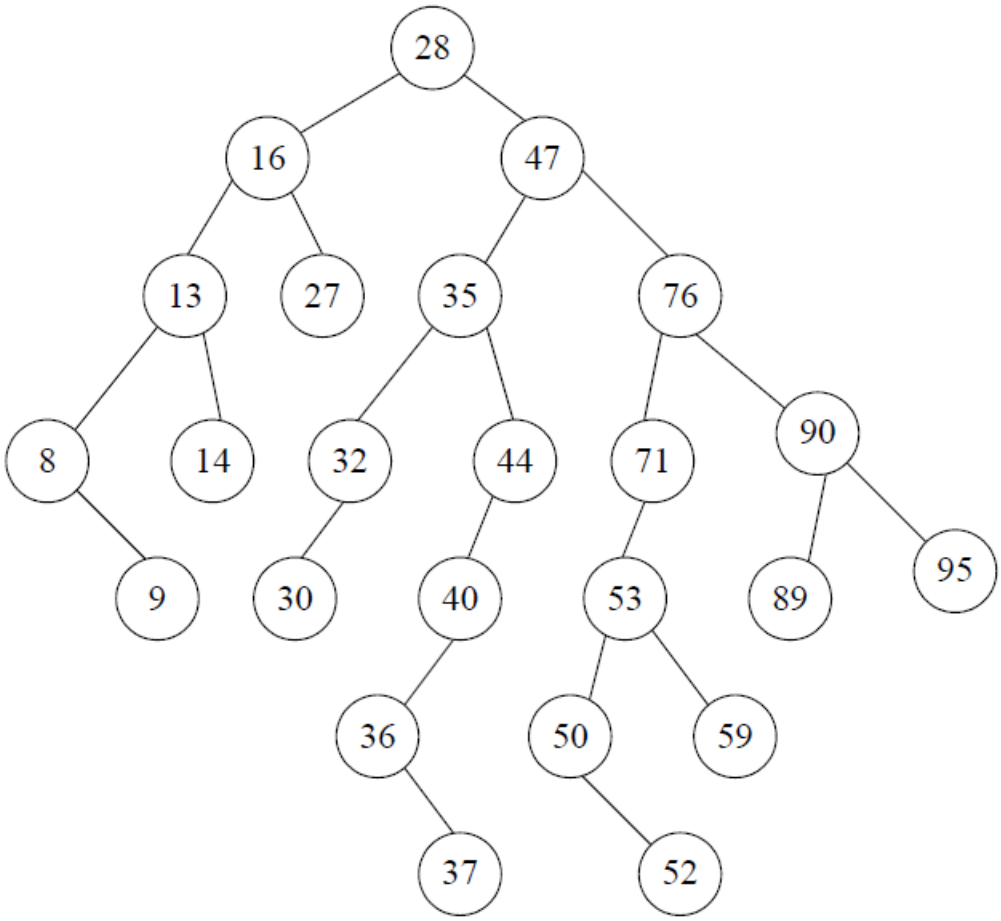
- 6 4 12 8 7 2 36 94

When I print the elements using inorder traversal, this is the order:

- 12 4 8 6 7 36 94 2

Draw the binary tree.

7. Assume we have the following BST. Delete the node with value 47. Explain all the steps in the deletion process:



8. Assume a binary tree has the following struct definition:

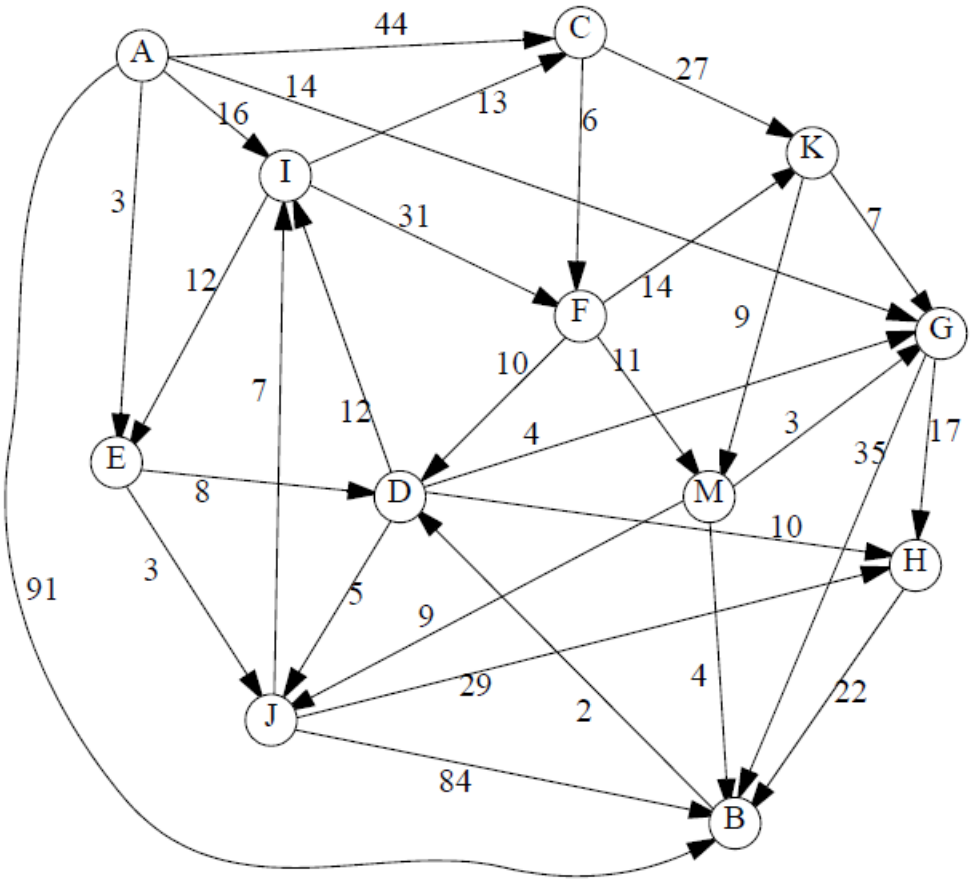
```
typedef struct TreeTag {
      int data;
      struct TreeTag * left;
      struct TreeTag * right;
} Tree;
```

Write the recursive definition for `countLeaves` that returns the number of leaves in the tree.

```
int countLeaves(Tree * root) {
```

}

9. Perform Dijkstra's algorithm on the following graph. Find the shortest path from C to B. Show the properties (white/black, dValue, pred) for each node after each step.



10. Using the graph in #9, find a topological sort of the vertices. First, show the discovery/finish times of DFS. Then, show a topological sort of the vertices.

11. What is the in degree of node B? _____

12. What is the out degree of node B? _____

13. What is the in degree of node F? _____

14. What is the out degree of node F? _____

15. Show the adjacency matrix representation of the graph in #5 below.

16. Show the values of the list after the partition step in quicksort (textbook's version) using pivot value 30:

30 15 75 64 20 2 8 35

17. Show the division and merge steps to sort the following list of integers using merge sort:

30 15 75 64 20 2 8 35

18. Assume a list has N items. What is the worst-case complexity of insertion sort? O(_____)

19. Assume a list has N items. What is the worst-case complexity of quicksort? O(_____)

20. What is the worst-case complexity of the insert operation for a binary search tree? O (_____)

Part 4: Hashing

Dictionary ADT
Hash Functions
Hash Tables

CS 305: In-class Activity 18 (Hashing)

Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____ (R)_____(P)_____(S)_____

Recall the Dictionary ADT. Its operations include insert, find, and delete. Earlier, we looked at how binary search trees can be used to implement the Dictionary ADT. One advantage of a BST is that the amount of memory used is proportional to the number of items in the dictionary. If the BSTs are kept balanced, the costs of the insert, delete, and find operations can run in O(lgN) time. Recall that if BSTs are not balanced, the costs of the operations are O(N).

One common tradeoff in computer science is the time/space tradeoff. In some (maybe many) cases, one can increase runtime speed at the expense of storage space. On the flip side, one can often gain efficiency in storage at the expense of running time.

We'll look at another way to implement the Dictionary ADT using hash tables. As you might guess, we will speed up the dictionary operations at the expense of using more storage space.

Consider this scenario. Your dictionary should store (student ID, student record) information. Student IDs are non-negative numbers. Your dictionary stores only currently enrolled UP students. Suppose the range of possible student IDs goes from 0 to 999,999,999.

1. How might you build a Dictionary ADT for this scenario so that insert, find, and delete are O(1) operations?

2. How much memory does your solution in #1 take? _____

Consider that there are about 4000 currently enrolled UP students. If you implement an array of size 1,000,000,000 to store (student ID, student record) pairs, the array is sparse. Certainly, the insert, delete, and find operations are O(1) with this implementation, but the wasted space is quite large given the number of items in the dictionary.

Here is one way to reduce the size of the array. If we know there are 4000 items in the dictionary, let's instead create an array of size 100,000 to store these items. Note that there are still 96,000 empty cells with this approach, but that is quite a bit smaller than 1 billion.

So, in order to insert a (student ID, student record) pair into the dictionary of size 100,000, we need to map student IDs to the range [0…99,999]. One common way to do this is to use the mod operator:

```
location = studentID % 100000;
```

The mod (remainder) operator gives us values in the appropriate range.

3. What is your student ID? _____

4. In what array location would your ID be inserted, assuming the array has 100,000 items? _____

5. Consider student ID 1152436. In what array location would this ID be inserted? _____

Because we have limited the array size, it is now possible to have *collisions*. A collision happens when two dictionary items are mapped to the same location in the array.

6. What is another student ID number that would be mapped to the same array location as 1152436? _____

Hash Functions

Above, we have considered the situation where the items are numbers to insert into the dictionary. Suppose now we want to insert strings into the dictionary (like an actual dictionary!!). We first need to map the string to a number. In this case, the string is our key and the number is our hash value. Once we have the value, we need to map the value to a location in the array.

The mapping from a key to an integer is called the *hash function*. There are many implementations of hash functions. A good hash function spreads keys across the range of integers. A good hash function is fast to compute, given the length of the key. A really bad hash function is one that maps all keys to the value 6. You will learn more about creating good hash functions in CS 324, Algorithms, if you take that course. Another property of a good hash function is that it maps the strings “abc” and “cba” to different integers, so permutations of the same set of letters have different hash values.

Another property of a good hash function is that if two keys are considered equal, they will map to the same hash value. For example, if uppercase “SAM” and lowercase “sam” are considered equivalent in your application, then they should map to the same hash value.

Here is an example of a hash function:

```
unsigned int hash(char *key) {
       unsigned int rtnVal = 3253;
       char *p;
       for (p = key; *p != '\0'; p++) {
              rtnVal *= 28277;
              rtnVal += *p * 2749;
```

```
        }
        return rtnVal;
}
```

So, hash("a") is:

rtnVal = 3253
rtnVal = 3253 * 28277 // 91985081
rtnVal = 91985081 + 97*2749 //note: 'a' in ASCII is 97
rtnVal = 92251734

7. What is hash("ab")? _____

8. What is hash("ba")? _____

(Note: 'b' in ASCII is value 98)

Hash Tables

The array that was mentioned earlier is called a hash table. A hash table stores dictionary items. The storage location is based on the hash value and using % of the hash table size.

Insertion

Assume we have a hash table of size 10. We want to insert the following items:

- "coconut"
- "milk"
- "apple"

STEP 1: We'll use the hash function listed above to calculate the hash values for these keys:

- "coconut" (hash value = 2104178476)
- "milk" (hash value = 461110994)
- "apple" (hash value = 3515030035)

STEP 2: We'll map the hash values to the size for this hash table (% 10):

- "coconut" (location = 6)
- "milk" (location = 4)
- "apple" (location = 5)

				"milk"	"apple"	"coconut"			
0	1	2	3	4	5	6	7	8	9

9. Now, suppose we want to insert “orange” into this hash table. Its hash value is 3410197053. Insert this key into the table above.

10. Now, let’s insert “corn”. Its hash value is 1347376851. Insert this key into the table above.

11. Now, let’s insert “eggs”. Its hash value is 881505635. Insert this key into the table above.

What happens now? “apple” is already stored at position 5. There are several techniques (see textbook) to address collisions.

We’ll first resolve collisions using *open address linear probing*.

Open Address Linear Probing

When a collision occurs, the linear probing technique finds the first unoccupied cell in the hash table to insert the item. In this case, “eggs” cannot go into position 5, cannot go into position 6, but position 7 is open. We put “eggs” into position 7 in the table above. Do that now.

One drawback of linear probing is that the runtime for insertion can now go to O(N) instead of O(1) for a hash table of size N. A second drawback of linear probing is that it produces primary clustering. When keys hash to the same value, the sequence of looking for open spots is the same. Suppose two new keys hash to location 5; both inserts would follow the same process of looking at position 5, 6, 7, 8 (and position 9 for the second key). Suppose a third key hashes to location 5; now this would try to insert the key into position 5, 6, 7, 8, 9. It would then wrap-around to looking at position 0 and insert it there since position 0 is empty.

But, the upside is that linear probing is simple to implement. Note that most hash table implementations use more sophisticated collision resolution techniques, which we will see later.

Delete

We have just examined the insert operation using linear probing. Now, let’s consider the delete operation. Suppose the hash table contains the following keys:

	“corn”		“orange”	“milk”	“apple”	“coconut”	“eggs”	“potato”	“salad”
0	1	2	3	4	5	6	7	8	9

Now, we want to delete the key “potato”. Suppose we just remove it by setting the position to NULL (empty box):

	“corn”		“orange”	“milk”	“apple”	“coconut”	“eggs”		“salad”
0	1	2	3	4	5	6	7	8	9

Now, let's suppose I want to delete "salad". Its hash value is 125082698. So, its location is 8. We look at position 8 in the table. It's null, so we conclude that "salad" is not in the table. Uh – that's not quite right. "salad" is in the table. We just didn't find it.

So, we need to do something a little more sophisticated to delete items. Table entries can store keys, NULL (empty), or a special symbol representing that the item was deleted. So, we'll use "D" as a special symbol to denote that an item was deleted. Suppose we delete "potato", as in the above example. This time, though, we put "D" in that position.

	"corn"		"orange"	"milk"	"apple"	"coconut"	"eggs"	"D"	"salad"
0	1	2	3	4	5	6	7	8	9

Now, when we go to delete "salad", we first look in position 8. We see that it has "D". So, we use linear probing to continue looking for the item "salad". We look at position 9. There it is, so we can set that position to "D".

	"corn"		"orange"	"milk"	"apple"	"coconut"	"eggs"	"D"	"D"
0	1	2	3	4	5	6	7	8	9

Now, suppose we want to delete "beans". Its hash location is 4. We look up T[4]. It has "milk", so we go to T[5]. It has "apple". We go to T[6]. It has "coconut". We go to T[7]. It has "eggs". We go to T[8]. It has "D". We go to T[9]. It has "D". We go to T[0] (wrap-around). It has NULL, so we conclude that "beans" is not in the dictionary.

12. Delete the key "eggs". Recall that this key maps to location 5. Update the table above. What strings are compared during the delete operation? _____

13. Now, insert the key "mango". This key maps to location 5. Update the table above. What location does "mango" go into? _____ What strings are compared during the insert? _____

Note that a dictionary holds just one copy of each key. If we try to insert "milk" into the hash table, it would look at location 4 and see that it already has "milk" and not update the hash table.

Finding

Searching for items in a dictionary is similar to the insertion and deletion processes. We need to hash the key, gets its hash value, and map it to the hash table location. We look at that location. If it is empty, we return -1. If it has what we are looking for, we return the location (or the actual key/value contents in the dictionary). Else, we go to the next position and continue this process.

```
D = special delete symbol
```

```
Find(H, key):
       value = hash(key)
       location = value % H.size
       origLocation = location
       while(H[location] != empty):
              if(key == H[location]), return location
              //note: H[location] does not equal key or H[location] is D
              location++
              location = location % H.size
              if(location == origLocation), return -1 //gone through entire hash table
       return -1
```

Suppose the hash table has:

	"corn"		"orange"	"milk"	"apple"	"coconut"	"mango"	"D"	"D"
0	1	2	3	4	5	6	7	8	9

14. Find "corn". This hashes to the location 1. How many string comparisons are done? _____

15. Find "mango". This hashes to the location 5. How many string comparisons are done? _____

16. Find "eggs". This hashes to the location 5. How many string comparisons are done? _____

Watch Dr. Vegdahl's video on hashing and chaining before the next lecture. Chaining is another major technique to resolve collisions in a hash table. Instead of each hash table cell containing a single dictionary item, we instead have a pointer to a linked list of dictionary items.

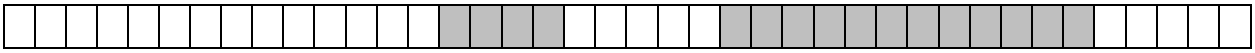
17. Does your group have questions about hash functions and/or hash tables?

```
/* CS 305
 * Hashing lecture
 * Code written by Dr. Steven Vegdahl, modified by Tammy VanDeGrift
 */

#include <stdio.h>
#include <stdlib.h>

/* hash function maps strings to unsigned ints */
unsigned int hash(char *key) {
  unsigned int rtnVal = 3253;
  char *p;
  for (p = key; *p != '\0'; p++) {
    rtnVal *= 28277;
    rtnVal += *p * 2749;
  }
  return rtnVal;
}

/* main gets data to hash at command line
 usage: h string (table size) */
int main(int argc, char *argv[]) {
  unsigned int modNum = 100;  //default hash table size
  if (argc == 2) {
  }
  else if (argc == 3) {
    modNum = atoi(argv[2]);
  }
  else {
    printf("Need one or two arguments \n");
    return EXIT_FAILURE;
  }

  unsigned int hashVal = hash(argv[1]);
  unsigned int moddedHashVal = hashVal % modNum;
  printf("Hash slot for '%s' is %u (hash value=%u)\n",
        argv[1], moddedHashVal, hashVal);

  return EXIT_SUCCESS;
}
```

CS305: Hash Table Collisions

As seen before, when two keys hash to the same location in a table, we have a collision. In the previous activity, we saw the **open address linear probing** technique. If a collision occurs, the new key is placed in the first empty or deleted cell to the right of the hash location (including wraparound).

An issue with the linear probing technique is that we get clusters of keys that hash to the same location.



Primary clustering occurs when keys that hash to different locations trace the same sequence when looking for an empty spot. In the table above, any keys hashed to the gray zones (table entries are not null) will look to the right for an empty location. So if positions [loc…loc+K] are full, any key that maps to this range will look to the right until finding loc+K+1. Linear probing suffers from primary clustering.

Secondary clustering occurs when keys that hash to the same location trace the same sequence when looking for an empty location to place the key. Linear probing suffers from secondary clustering as well.

Quadratic Probing

This technique uses open addressing (one key per hash table location). But, instead of just looking at loc + 1 when trying to find an empty slot, it does something a bit different:

When a collision happens, we instead go forward in the table $a*i + b*i^2$ slots. i is set to 1 for the first collision, 2 for the second collision, etc. a and b are constants, set by the hash table implementation.

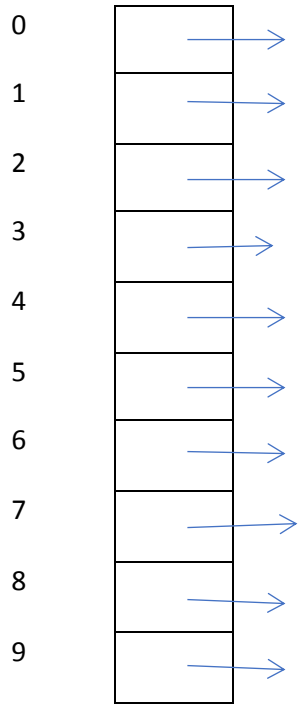
Suppose now we have a table of size 100. A key is already inserted in position 5. A new key hashes to position 5.

Assume the probing function is: $i + i^2$ //a and b are set to 1
Then, on the first collision, we get: 1 + 1 = 2.
We would check T[7] //5 + 2
If that collides, we would calculate: 2 + 4 = 6.
We would check T[13] //7 + 6, goes forward from previous location
If that collides, we would calculate: 3 + 9 = 12.
We would check T[25] //13 + 12

The insert, find, and delete all use the same quadratic probing technique when collisions happen. This quadratic probing technique reduces the clustering effect (lots of data in adjacent slots).

Chaining

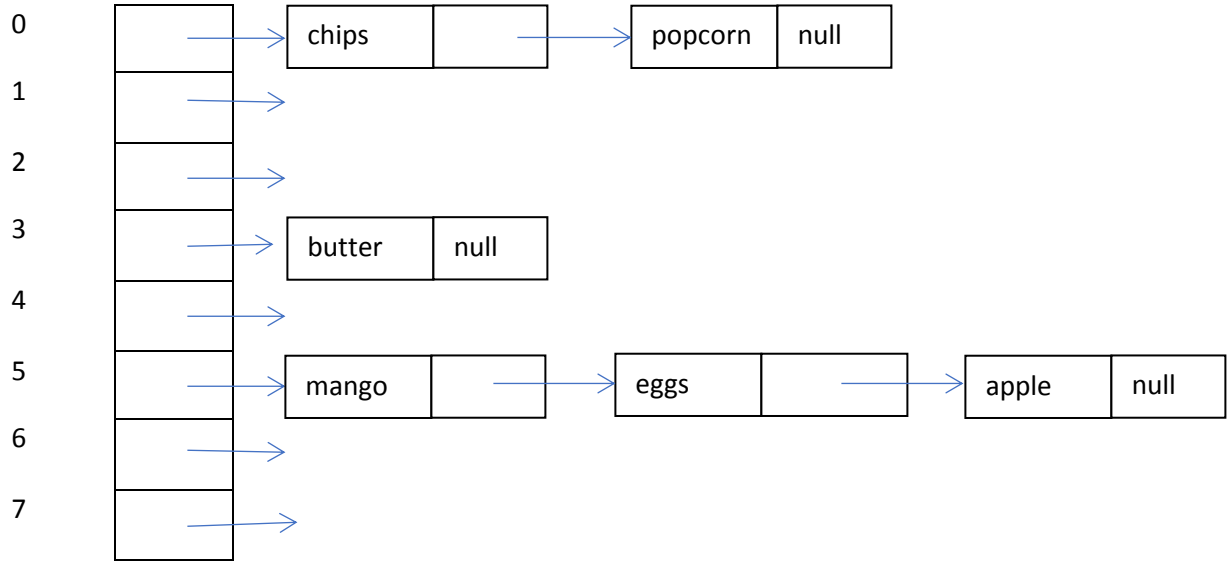
This technique is presented in Dr. Vegdahl's video. Instead of our hash table storing just keys (1 key or maybe 1 key/value pair) in the cells, we instead store pointers to linked lists in each cell.

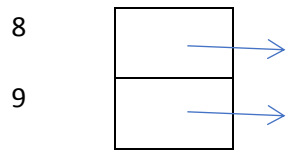


Suppose we want to insert the following items with hash values mapped to these locations:

- apple 5
- popcorn 0
- butter 3
- eggs 5
- mango 5
- chips 0

Then, when collisions happen, we insert the new key at the front of the linked list for that location.





What is the worst-case scenario in terms of running time for insert, find, and delete? _____

Computing hash function:	O(1)	//if it is a fast function
Computing location:	O(1)	
Finding location in array:	O(1)	
Navigating through linked list:	O(N)	//if all keys map to same location
Accessing record (key/value):	O(1)	//once it is found

How could we make this faster or do we really get "long" linked lists?

Assumptions:

- We have a good hash function that spreads keys across entire range of values.
- Our table size is proportional to the size of the dictionary; we can grow the table size when the table gets some % full; note that growing a table means calculating new hash locations for every item in the table
- With these two assumptions, long linked lists "should not" happen. This gives us O(1) complexity!!

Double Hashing (open address)

When a collision occurs, use a second hash function. The first hash function generates the hash value, which determines the location. If there is a collision, a second hash function on the key will determine the increment k to use for probing. So, in linear probing, we used loc + 1. Now, we use loc + k (with wrap-around) to find an empty or deleted cell. This reduces clustering, since each key has (hopefully) a fairly unique offset value.

In general, it is a good idea to choose the hash table size to be a prime number. We have used tables of size 10 for ease of human calculation. But, in general, you want to choose table sizes that are prime numbers.

Do you have other ideas for handling collisions?

CS 305: In-class Activity 19 (Handling collisions)

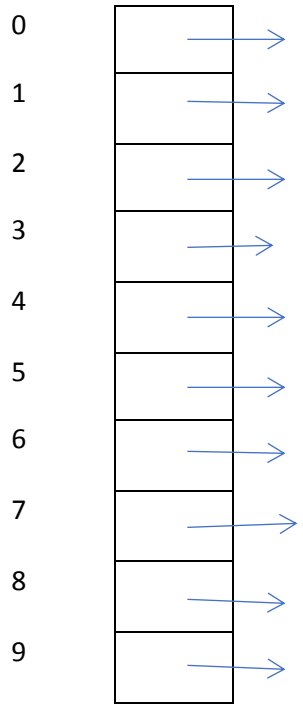
Write down the team's consensus answers to the questions on one sheet.

Names: (M)_____(R)_____(P)_____(S)_____

1. Suppose we are using chaining to handle hash table collisions. Show the contents of the hash table after the following items are inserted (in this order). Each item's hash value is shown. The hash table size is 10.

Items:

- "table" (hash value = 944611085)
- "car" (hash value = 401203259)
- "truck" (hash value =2173423306)
- "spoon" (hash value = 4169904816)
- "cup" (hash value = 1955867221)
- "chair" (hash value = 1637910816)
- "couch" (hash value = 3993966603)
- "picture" (hash value = 3936368289)



2. Suppose we are using open address linear probing with same data as in problem 1. Show the contents of the hash table after all items are inserted.

0	1	2	3	4	5	6	7	8	9

3. Suppose we are using a second hash function for collisions. The second hash function is the length of the key. Note that this is a really BAD hash function, but it is something you can compute easily. Show the contents of the hash table after cup has been inserted.

0	1	2	3	4	5	6	7	8	9

4. What happens when “chair” is being inserted? _____

This is why it is a good idea to create hash tables that have prime number sizes. You will try different sets of locations each pass through the table.

5. Does your group have questions about handling collisions?

My notes about the entire course:

Final exam review guide and practice questions

The final exam is Tuesday, April 28, 8 – 10am. The exam length is *two* hours. The final exam is comprehensive and will cover material from the entire course. You should review your coursepack, in-class activities, data structures textbook, GNU tutorial, prelabs, labs, homework assignments, and previous exams.

Here are topics from the entire course:

EXAM 1:

- C control flow (if, if/else, switch, for, while)
- Functions (prototypes, definition, parameters, return type)
- Variables and types (examples: int, char, double, int *, char *, int **, etc.)
- Arrays
- Structs
- Typedef
- Malloc, free, sizeof (allocating memory on the heap)
- Pointers (to variables, to structs, to arrays, etc.)
- NULL, dereferencing pointers, dangling pointers, pointer arithmetic
- Preprocessor directives
- Printing to console (stdout)
- Reading data from keyboard (stdin)
- File I/O (reading from text files, writing to text files)
- Identifying when a segmentation would occur
- Identifying syntax errors
- Good programming style

EXAM 2:

- make and makefiles
- gdb
- Complexity and O-notation
- Linear search
- Binary search
- Prime numbers (looking for divisors, sieve of Eratosthenes)
- Recursive functions (examples: palindrome, gdb, multiply, printing numbers in different bases)
- Arrays for data storage
- Linked Lists, Circular Linked Lists, Doubly Linked Lists
- Stacks
- Queues

EXAM 3:

- Dictionary ADT
- Trees
    - Traversals (preorder, postorder, inorder)
    - Terminology, such as parent, child, leaf, root, descendant, ancestor, height
    - Applications of trees
- Binary Search Trees
    - insertion
    - deletion
    - finding items (find value, find min, find max)
- Tree Rotations and Balancing
- Sorting
    - Selection sort
    - Insertion sort
    - Quicksort
    - Merge sort
    - Complexity of sorting routines
- Graphs

- Terminology, such as vertex, edge, directed, undirected, degree, predecessor, successor, connected, acyclic, weighted
- Searches
  - Depth-first search
  - Breadth-first search
- Applications of graphs
- Representations
  - Adjacency Matrix
  - Adjacency List
- Dijkstra's algorithm: single source, shortest paths
- Topological Sort
- Minimum Spanning Trees
  - Kruskal
  - Prim (similar to Dijkstra)

SINCE EXAM 3:

- Dictionary ADT (reprise)
- Hash Functions
- Hash Tables
  - insert, delete, find
- Collisions with hash tables
  - Open Address Linear Probing
  - Open Address Quadratic Probing
  - Chaining
  - Double Hashing
  - Pros/Cons of open address hashing vs. chaining

You can be expected to write code on the exam, read code, and answer questions about code. You may be asked to find syntax errors and run-time errors in code.

You will be allowed **two** 8"x11.5" crib sheets (both sides) to use while taking the exam. Your crib sheets can be hand-written or typed. No other aids are permitted (computers, calculators, headphones, music, phones).

The remaining portion of this review guide has practice questions to prepare for the exam. Be sure to study all topics above to prepare for the exam (not every topic has a practice exam question below).

!!! Remember: as you study, you can write small programs to see how the code compiles and executes. You may use the lab files to experiment !!!

Example problems (see example problems from earlier exams)

1. Write C declarations for the following variables:

A. An array of 10 integers initialized to 0 called `arr`.

B. A pointer to an integer called `p1`. It should point to slot 6 of `arr`.

C. An array than contains 3 pointers to integers, each initialized to null. This should be called `arr2`.

D. A pointer-to-pointer to integer which points to slot 1 of arr2. This should be called `p2`.

2. Consider the linked list code from lab 6. Write a function called `deleteAlternates` that deletes every other node in the linked list, beginning with the first. It should free each deleted node.

If the linked list has items 3, 6, 8, 2, 4, 7, then the function should delete the items 3, 8, and 4. When the function finishes, `listPtr` should contain a pointer to the Node containing 6.

```
void deleteAlternates(Node ** listPtr) {

}
```

3. Simplify the following O-notation expressions:

$O(n^8) + O(n^{18}) = O($_____$)$

$O(n^3 (\log n)^3) + O(n^5) = O($_____$)$

$O(n^3) * O(n^8) = O($_____$)$

$O(23) + O(3^{17}) = O($_____$)$

$n^5 + 3n^5 + n^7 + 8152 = O($_____$)$

4. Consider the definition for a binary tree (not necessarily a binary search tree).

```
typedef struct TreeTag Tree;
struct TreeTag {
      int data;
      Tree *left;
      Tree *right;
};
```

Write a function called `countGreater` that counts the number of elements in the tree that are greater than or equal to a given value n. Remember – this is just a binary tree and not necessarily a binary search tree.

```
int countGreater(Tree *root, value n) {









}
```

5. Draw a binary search tree of height 3 that contains the following nodes: 10, 20, 30, 40, 50, 60. This is not necessarily the order in which they are inserted.

6. Consider the code below.

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdbool.h>

void alpha(int *p) {
      (*p)--;
      printf("%d\n", *p);
}

void check(int *p) {
      printf("%d\n", *p);
      alpha(p);
}

int count(int a) {
      int sq = a*a;
      return sq;
}

int main(void) {
      int n = count(3);
      check(&n);
      printf("%d\n", n);
      return EXIT_SUCCESS;
}
```

a. What does the code print?

b. What functions are on the program stack when alpha is executing? (leftmost is bottom of stack in answers below)

a. main
b. main, count
c. main, count, check

d. main, count, check, alpha
e. main, check, alpha

7. Consider the `queue.c` and `queue.h` code from lab 7. Recall that this implementation is a circular queue. Assume the `MAX_Q` is set to 5.

Assume the following code is executed:

```
Queue q = initQueue();
enqueue(q, 4);   // statement 1
enqueue(q, 7);   // statement 2
QueueData a = dequeue(q);   // statement 3
enqueue(q, 9);    // statement 4
enqueue(q, 3);    // statement 5
QueueData b = dequeue(q);   // statement 6
QueueData c = dequeue(q);   // statement 7
enqueue(q, 8);    // statement 8
QueueData d = dequeue(q);   // statement 9
enqueue(q, 1);    // statement 10
```

A. Show the contents of the queue (both the array, its indices, and contents) after statement 2 has finished.

B. What is a? _____

C. Show the contents of the queue after statement 5 has finished.

D. Show the contents of the queue after statement 10 has finished.

8. I want to sort an array using quicksort. The array has the following values:

45 7 28 93 77 -3 18 101

This version of quicksort always selects the first element in the array as the pivot element. Consider the contents of the array after the partition step is called, but before recursive calls to quicksort are made. Give three different orderings of this array that could be valid after partition is called:

Ordering #1:

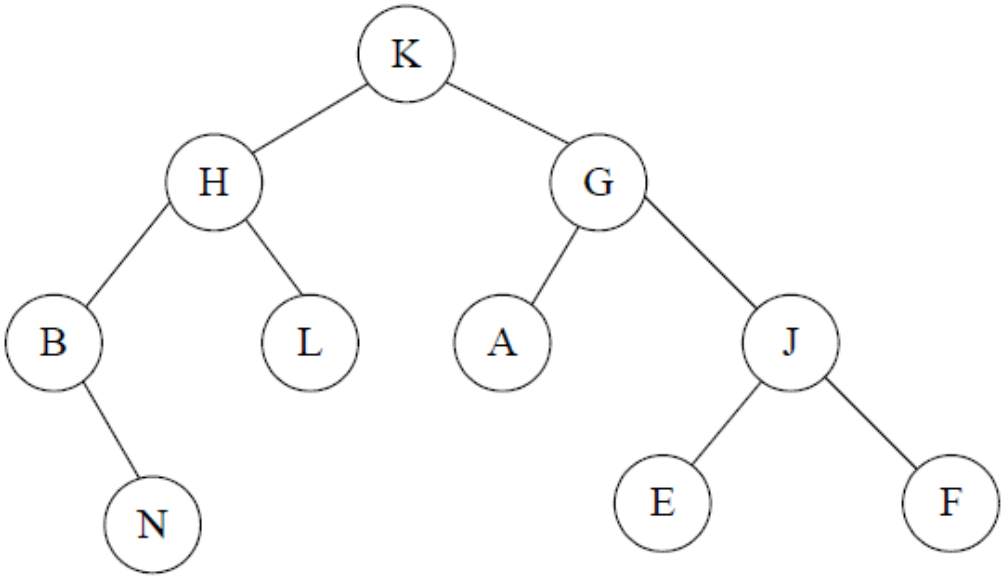
Ordering #2:

Ordering #3:

9. Give the worst-case O-notation for each of the following algorithms:

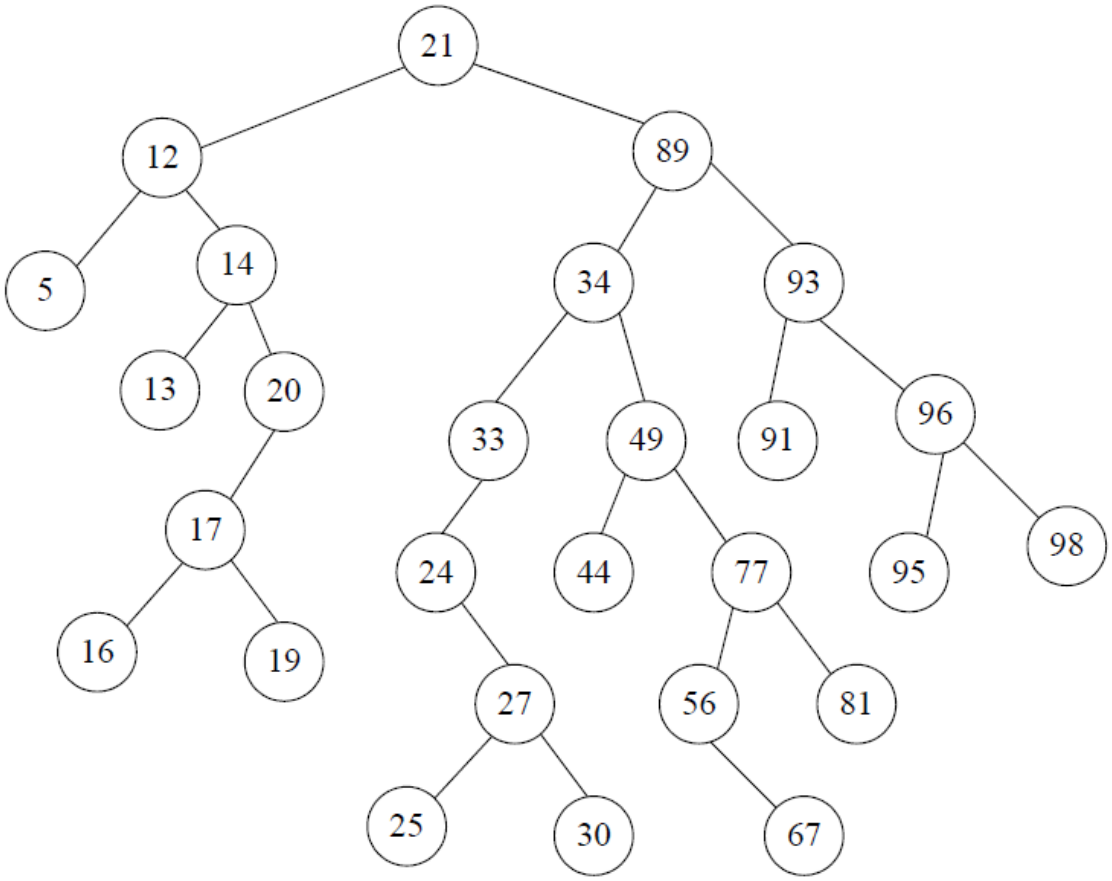
A. mergesort, data is already known to be sorted
B. mergesort, data is not already sorted
C. quicksort, data is already known to be sorted
D. quicksort, data is not already sorted but is randomized
E. binary search tree lookup, tree is balanced
F. binary search tree lookup, tree is not known to be balanced

10. Here is a binary tree:



A. What is the postorder traversal?
B. What is the inorder traversal?

11. Here is a BST:



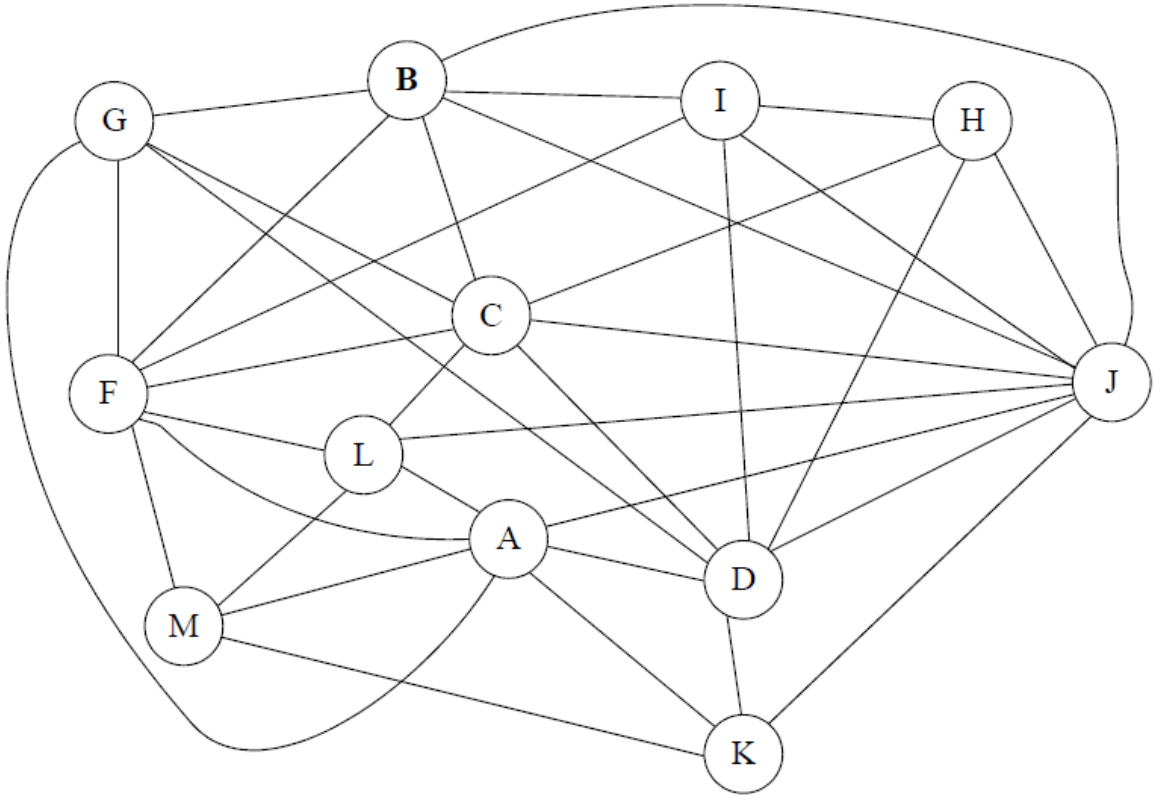
A. Insert data 32.

B. Insert data 60.

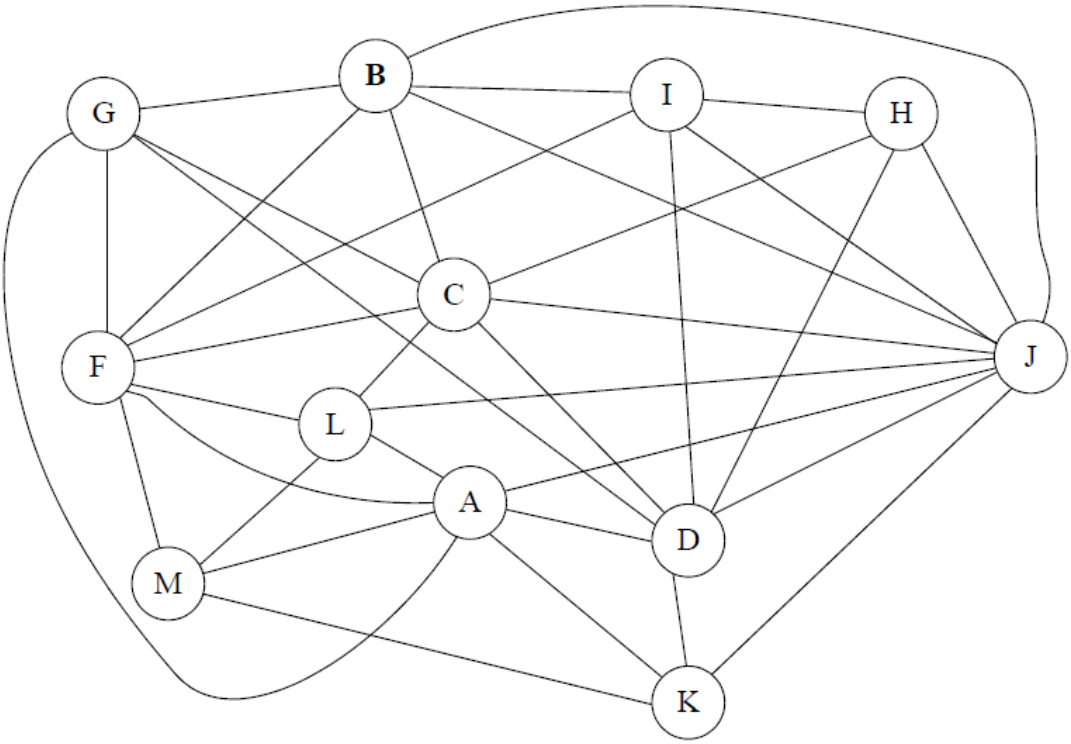
C. Put *s near each node that is accessed when searching for data item 29.

D. Delete value 21 using the next successor approach. Cross out any values that are updated and cross out edges and nodes that are removed.

12. Consider the following graph. Do depth-first traversal of the following graph starting with node A. If there is a choice, select the earliest vertex in alphabetical order. Show the parent, discover/finish times.



13. Do breadth-first search on the graph in #12. Start with node A. For each node, label the parent and its distance from A:

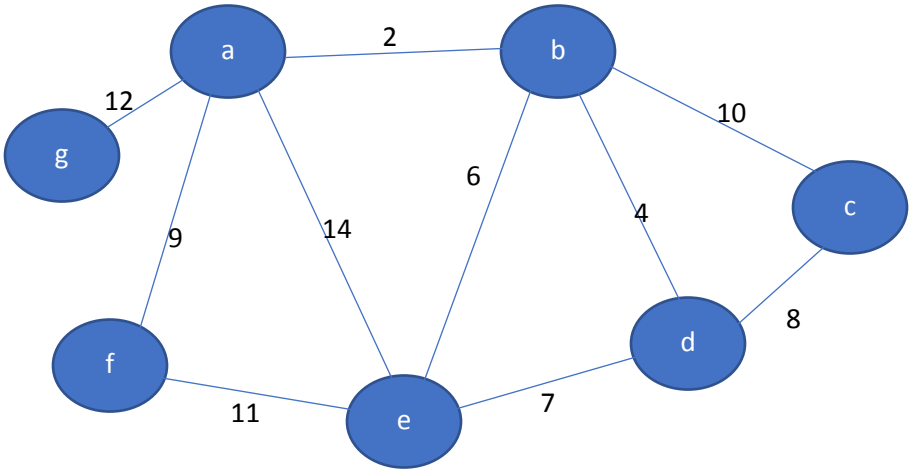


14. We are about the insert the following elements into an empty 11-slot open-address hash table with linear probing. What does the table look like after all items have been inserted?

Ham (hash value 34)
Butter (hash value 28)
Jello (hash value 27)
Jam (hash value 22)
Burger (hash value 11)
Salad (hash value 82)

0	1	2	3	4	5	6	7	8	9	10

15. Find the minimum spanning tree for the graph below using Kruskal's Algorithm. Show the ordering of the edges and determine which edges are added to the MST and which edges are not.



16. Provide a definition of a data structure. An example is not a definition, but may help clarify the definition.

17. It is important to program defensively in C. Consider the following code snippets. Is it safe to execute *as-is* or should more defenses be added?

```
int i = 0;
while(i < 5) {
      printf("%s", label);
      i++;
}
```

Safe? YES NO

If not safe, what defenses should be added?

```
printf("%s", argv[2]);
```

Safe? YES NO

If not safe, what defenses should be added?