# CONTENTS

1 Tutorials
   1.1 Linux Intro .......................................................... 3
   1.2 Makefiles ............................................................. 5
   1.3 Python ................................................................. 8
   1.4 Java ................................................................. 10

2 Algorithm Analysis .......................................................... 13
   2.1 Preliminaries .......................................................... 13
   2.2 Asymptotic Notation (Big-O and the others) .................... 16
   2.3 Finding the functions for an algorithm ....................... 19
   2.4 Recurrence Relations For Recursion .......................... 21

3 Lists Stacks and Queues .................................................. 23
   3.1 Abstract Data Types (ADTs) ....................................... 23
   3.2 Lists ................................................................. 24
   3.3 Stacks ............................................................... 25
   3.4 Queues ............................................................. 27

4 Trees .............................................................................. 29
   4.1 Unbalanced Binary Trees ............................................ 29
   4.2 Balanced Trees ....................................................... 33
   4.3 Tries ................................................................. 59
   4.4 Roundup .............................................................. 63

5 Review of Multithreaded Programming ................................. 65
   5.1 Sum3 Problem Overview ............................................ 65
   5.2 Sum3 Simple Implementation (Serial) ....................... 68
   5.3 Sum3 Parallel Implementation with Pthreads and OpenMP ... 73

6 Hashing and Hash Tables .................................................. 89
   6.1 Hash Tables .......................................................... 89
   6.2 Bloom Filters ........................................................ 94

7 Heaps and Priority Queues ................................................ 99
   7.1 Priority Queues ....................................................... 99
   7.2 Advanced Heaps .................................................... 106

8 Graphs ............................................................................ 113
   8.1 Graph Definitions .................................................... 113
   8.2 Graph Representation ............................................... 114
   8.3 Topology .............................................................. 116
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.4</td>
<td>Topological Sort</td>
<td>117</td>
</tr>
<tr>
<td>8.5</td>
<td>Unweighted Shortest Path</td>
<td>119</td>
</tr>
<tr>
<td>8.6</td>
<td>Dijkstra’s Algorithm</td>
<td>120</td>
</tr>
<tr>
<td>8.7</td>
<td>Prim’s Algorithm (MST)</td>
<td>120</td>
</tr>
<tr>
<td>8.8</td>
<td>Kruskal’s Algorithm (MST)</td>
<td>120</td>
</tr>
<tr>
<td>8.9</td>
<td>Euler Circuits</td>
<td>121</td>
</tr>
</tbody>
</table>

9 Indices and tables  

Bibliography
1.1 Linux Intro

There are many Linux command line tutorials online, so this will be a brief one. Search Linux command line tutorial for a bunch of options.

The basic idea behind Linux, from the user’s perspective, is that there are a lot of small programs that can form powerful combinations and do complex things. The first step is to get to a Linux command line. We will assume you are connecting to a remote Linux host. You can do this through a variety of programs:

Programs to connect to a remote Linux host.

Mac: use the terminal program. This is actually a Linux command line

Windows: the easiest thing to use the putty application. It is a free download. Enter the web address of the host you want to connect to, and then start a session. Note that with putty, you will not be able to use applications that open up new windows. You might want to create a Linux live CD, essentially a bootable CD (or USB stick) that will boot your computer into Linux, but will not alter anything on your hard drive (leaving your precious Windows untouched). I recommend the Ubuntu live CD.

Linux: use the shell or terminal. There are many options depending on which Linux flavor you have. You might have gterm, kterm, etc. Basically, open a command line

Once you have the appropriate program, connect to a host. home.cs.siue.edu should almost always be available, I will give you some others in class.

To connect to the remote Linux server, you need to tell a secure shell program to connect the appropriate host. If you are using a terminal, you invoke the ssh program from the command line. If you are using putty, simply put the server address in the dialogue box and click the connect button. Putty will ask for your username and password. To invoke the ssh program by hand in a terminal, type the following. The -X -Y options will allow you to use programs that create new windows:

```bash
ssh home.cs.siue.edu -l [your username] -X -Y
```

Once connected, you should be at a simple command line in a shell. The shell is actually an interpreter for a programming language, the default language is usually bash. In its simplest form, bash takes a line that you enter, and executes it. Typically, the lines you enter will call other programs, but you can enter if statements, loops, etc in the bash syntax.

Once connected, you can use some commands. The following is a brief list of commands. Note that every command should have a man page that describes what it does and how to use it. If you want to learn more about a command, you simply type `man [the command].`

Some basic Linux commands

ls - list the contents of a directory
mkdir - make a directory

cd - change to another directory

touch - update the timestamp of a file, or create the file if it does not exist

nano - a simple text editing program

cat - show the contents of a file (dump it to the screen)

less - browse the contents of a file (hit q to quit)

gcc - the c compiler

g++ - the c++ compiler

The following is a script showing some of the commands being used. Note that the prompt tells you what directory you are in. ~ is a shortcut for your home directory. Everything after the $ sign is a command that I have entered. If the command produces output, it is shown on the next line:

marmcke@beet/~$ ssh home.cs.siue.edu -l marmcke -X -Y
Last login: Tue May 13 12:04:33 2014 from beet.cs.siue.edu

If you are using PuTTY to connect to this system, some characters will not show up correctly because PuTTY defaults to the wrong character encoding. To get all the characters to display correctly, you must set the encoding/translation to UTF-8.

If you are getting disconnected from your ssh session every five minutes or so, it is probably because the SIUE firewall is resetting its connection-state tables when no traffic is seen for a few minutes. PuTTY or openssh can be configured to send a NULL character every minute or so to keep the session alive (see http://www.kehlet.cx/articles/129.html for openssh instructions).

If you are denied access to this server due to your having entered too many invalid passwords(3), reset your modem to receive a new ip address or email cs-support@siue.edu to have your ip address unblocked.

marmcke@vm-02/~$ mkdir test340
marmcke@vm-02/~$ cd test340
marmcke@vm-02/~/test340$ ls
marmcke@vm-02/~/test340$ ls -a
. ..
.

marmcke@vm-02/~/test340$ mkdir example1
marmcke@vm-02/~/test340$ ls
example1
.
.

marmcke@vm-02/~/test340$ cd example1/
.

marmcke@vm-02/~/test340$ touch ex.cpp
.
.

marmcke@vm-02/~/test340$ nano ex.cpp
.
.

marmcke@vm-02/~/test340$ g++ ex.cpp -o ex
.
.

marmcke@vm-02/~/test340$ ./ex
hiiiiiiiiii
.
.

marmcke@vm-02/~/test340$ cd ..
.
.

marmcke@vm-02/~/test340$ echo "the .. directory is a shortcut for the parent directory of the current directory"
.
.

marmcke@vm-02/~/test340$ ls
.
.

marmcke@vm-02/~/test340$ ls -a
.
.

marmcke@vm-02/~/test340$ exit
.
.

4 Chapter 1. Tutorials
1.2 Makefiles

Ususally, the make program is installed on linux servers. To invoke the program, type make. The default behavior is to find a file in the current directory called makefile (or Makefile), and execute the default target in that file. The default target is named all: Make can do some very complicated stuff, but the most basic behavior is that it will simply execute every line after the all: target. So, you just put commands to be executed. Make sure the commands are indented.

We will use makefiles to compile the programs you submit in this class. That way, you can have any crazy compilation technique you want, but I simply type make, and it all compiles. This allows you to use your language of choice. The following code lists the creation and usage of a simple c++ program and a makefile to compile it.

```bash
marmcke@beet/~/$ ssh home.cs.siue.edu -l marmcke -X -Y
Last login: Tue May 13 14:04:02 2014 from client49-37.wpa.siue.edu

If you are using PuTTY to connect to this system, some characters will not show up correctly because PuTTY defaults to the wrong character encoding. To get all the characters to display correctly, you must set the encoding/translation to UTF-8.

If you are getting disconnected from your ssh session every five minutes or so, it is probably because the SIUE firewall thinks your session is too long. Disconnect and reconnect every minute or so to keep the session alive (see http://www.kehlet.cx/articles/129.html for openssh instructions).

If you are denied access to this server due to your having entered too many invalid passwords(3), reset your modem to receive a new ip address or email cs-support@siue.edu to have your ip address unblocked.

marmcke@vm-02/~/test_340Ex2$ mkdir test_340Ex2
marmcke@vm-02/~/test_340Ex2$ cd test_340Ex2/
marmcke@vm-02/~/test_340Ex2$ touch ex.cpp
marmcke@vm-02/~/test_340Ex2$ nano ex.cpp
marmcke@vm-02/~/test_340Ex2$ cat ex.cpp
#include <iostream>

int main(){
    std::cout<<"hihihi"<<std::endl;
}

marmcke@vm-02/~/test_340Ex2$ touch Makefile
marmcke@vm-02/~/test_340Ex2$ nano Makefile
marmcke@vm-02/~/test_340Ex2$ cat Makefile
all:
g++ ex.cpp
echo "hi"

marmcke@vm-02/~/test_340Ex2$ make
make: Warning: File 'Makefile' has modification time 572 s in the future
make: Warning: File 'Makefile' has modification time 572 s in the future
g++ ex.cpp
echo "hi"
hi
make: warning: Clock skew detected. Your build may be incomplete.
marmcke@vm-02/~/test_340Ex2$ ls
a.out ex.cpp Makefile
marmcke@vm-02/~/test_340Ex2$ ./a.out
hihihi
marmcke@vm-02/~/test_340Ex2$
1.2.1 Makefiles using Java

Using a makefile for Java is no different that for C++, or any other language, for that matter. The main problem people have is that often I will require your program to have a specific name for its executable. This can be a little troublesome if you use a package or class name that is not the name of the executable I specified. This part shows how to create an executable jar file with whatever name you choose, regardless of what you named your packages or class files.

Lets say you have an assignment and the executable name must be exampleProgram. However, your solution looks like the following:

```java
// this is in OtherClass.java
package example;

class OtherClass{
    public OtherClass(){
        // nothing to see here
    }
    public void theFunction( String x ){
        System.out.println( x );
    }
}

// this is in Main.java
package example;

class Main{
    public static void main(String[] args){
        OtherClass oc = new OtherClass();
        if( args.length > 0 )
            oc.theFunction( args[0] );
        if( args.length > 1 )
            oc.theFunction( args[1] );
    }
}
```

Those files are in an example directory, consistent with the package name. For instance:

```
~$ pwd
~
~$ ls
example
~$ cd example/
~/example$ ls
Main.java OtherClass.java
~/example$
```

**Warning:** We assume you are in the TOP LEVEL DIRECTORY, not in the examples directory.

To compile you program, you would type:

```
~$ javac example/*.java
```

So, we would just put that in a Makefile

To create an executable jar, we use the jar command. The jar command requires the name of the desired jarfile, the directory containing the package, and a manifest file. The manifest file simply indicates which class contains the public static void main(String[] args) function. The following sequence shows the contents of a manifest file, and the correct jar command. Note the that jar options are for the following:
c: create a jar file

m: the next argument will be the name of the manifest file

f: the argument after the manifest file will be the directory holding the package to jar (the compiled files are there)

Here is it how it looks:

```bash
$ ls
example Makefile manifest.mf
$ cat manifest.mf
manifest-Version: 1.0
Main-Class: example.Main

$ jar cmf manifest.mf exampleProgram.jar example
$```

Finally, putting all that in a Makefile looks like this:

```bash
$ cat Makefile
all:
  javac example/*.java
  jar cmf manifest.mf exampleProgram.jar example
  @echo "+ to run type: java -jar exampleProgram.jar"
  @echo "+ to run with command line arguments, simply append them to the line above"
```

So, the final directory structure with the Makefile looks like the following:

```bash
$ ls example/
Main.java OtherClass.java
$ ls .
example Makefile manifest.mf
$ make
javac example/*.java
jar cmf manifest.mf exampleProgram.jar example
+ to run type: java -jar exampleProgram.jar
+ to run with command line arguments, simply append them to the line above
$ ls example
Main.class Main.java OtherClass.class OtherClass.java
$ ls .
example exampleProgram.jar Makefile manifest.mf
```

To run the program, you simply use the java command, but give it the -jar option:

```bash
$ java -jar exampleProgram.jar
$ java -jar exampleProgram.jar 1
1
$ java -jar exampleProgram.jar 1 42
1
42
```

So, we were able to name the jar file whatever we wanted.

**Exercise**

This example also utilized command line arguments. The final code block shows the program being run with 0, 1 and 2 command line arguments. Command line args are super useful, you should walk through the program to see how they work. The basic idea is that the static public void main(String[] args) function gets an
array of strings from the command line when the program runs. The operating system takes the arguments from the
command line, and gives them to the main function when it sets up the program’s memory.
Modify the program so that it will print out up to 3 arguments.
Modify the program so that it will print out ALL arguments given.
Modify the program so that the 1st argument is an integer indicating how many of the following command line argu-
ments should be printed.

1.3 Python

There are two trunks of python that are actively developed: the 2.x and 3.x trunks. We will use the 2.x trunk in this
class. So download the appropriate one (python 2.7 is current).

Python is a scripting language, and happens to be an interpreted language. As a scripting language, it is meant to
easily handle text processing, be oriented to writing code quickly, and is easily integrated with other executables and
libraries. Furthermore, python has rules to enforce the readability of code, and lots of tools/libraries to make coding
easier. One of the guiding philosophies to coding in python is to concentrate on the algorithm, not the code. In fact,
the speed of your code should generally not concern you when coding in python, except for egregious mistakes like
implementing an \(O(n^2)\) algorithm when a constant time option is available and easy to implement.

In python, white space matters. There are no curly braces for code blocks; code blocks are identified by indentation.
This forces nice readable code:

```python
a = 3
if a < 2:
    print 'a is less than 2'
    a = 4
    print 'not any more'
else:
    print 'a is >= 2'
    a = 1
    print 'not any more'
```

In python, types are not declared (unless you want to). Basically, type information is figured out as the program
progresses. for example:

```
a = 4
a = 4 + 7  # a is an integer
a = a + 6.2  # a is now a double
b = 'hi there'  # b is a string
b = 4  # b is now an integer
```

Python is an interpreted language, meaning the source code is run through an interpreter (think virtual machine, like
Java). Therefore, errors can occur on the fly and exceptions can/will be thrown.

The following is a simple script illustrating some basic python stuff. Get an interpreter going and try it out. You can
type directly into an interpreter, or use the interpreter to execute the entire program:

```
a = 4
print a
a = a + 7.2
print a
a = 'hi'
print a
```
# tuples are like arrays, but are IMMUTABLE, you cannot change them once they are made

```python
a = (1, 2, 3, 4, 'hi')
print(a)
# to update the tuple, make a new one and copy what you need from the old one
a = (2, 3, 4, a[3], a[4])
print(a)
```

# lists. Lists are like linked lists, but can be accessed with brackets like arrays. From a usage standpoint, they are most like dynamic arrays.

```python
a = []  # empty list
a = [2, 5, 6, 'hi']
print(a)
print(a[2])
print(a[3])
a.append('more!')
print(a)
```

# loops. Python typically does not do all the manual loop incrementing that C++ does. Instead, it provides an IN keyword that will get 1 item at a time out of a container (a tuple, list, etc)

```python
for item in a:
    print(item)

# to loop a specific number of times, or to generate a counting variable to be used in the loop, the range function will create a list of integers in a sequence.

# range( startInt (optional), stopInt (non-inclusive), incrementStep (optional)

for i in range(10):
    print(i)
for i in range(2, 10, 3):
    print(i)
```

The output of the above script is shown below

```
4
11.2
hi
(1, 2, 3, 4, 'hi')
(2, 3, 4, 4, 'hi')
[2, 5, 6, 'hi']
hi
[2, 5, 6, 'hi', 'more!']
2
5
6
hi
more!
o
0
1
2
```
The final program uses command line arguments. To include a library, python uses `import` statements. To use command line args, we need to import `sys`. There is a try catch block since command line args come in as strings, and we need to convert it to a number (there is no guarantee the user entered a number and not text).

```python
import sys

print sys.argv

if len(sys.argv) < 2:
    print 'usage: python', sys.argv[0], '[a number]
    print 'this program computes the sum of 1 to the number you enter'
    exit()

# a try catch block for the number conversion exception
try:
    stopInt = int(sys.argv[1])  # args will come in as strings, convert to int
except ValueError:
    print 'You must enter an integer'
    exit()

sumNum = 0
for i in range(stopInt+1):
    sumNum += i

print sumNum
```

At this point you should have a feel for the basics. I would suggest finding a tutorial online and checking it out. There are lots of online resources for python. In general, just search for whatever you need and the answer is usually in the top 2 or 3 search results.

### 1.4 Java

When submitting projects using Java, you must ensure that they are able to be compiled and run on one of the department linux servers (specifically, the home server). If you use a program with multiple source files, you must create an executable `jar` that can be run from the command line. The following example shows two source files, a `manifest` file, and a `Makefile`. If you put all of these files in the same directory, you should be able to compile them and execute program from the command line. The manifest file simply tells the `jar` where the `static main` function resides in the java files.

**Source file: HelloWorld.java**

```java
class HelloWorld
{
    public static void main(String[] args)
    {
```
OtherClass OC = new OtherClass();
OC.printSomething( "hi there, human!" );
}
}

Source file: OtherClass.java

class OtherClass
{
    public void printSomething( String thing )
    {
        System.out.println( thing );
    }
}

Manifest file: MANIFEST.MA

Main-Class: HelloWorld

Makefile:

all:
javac -d . HelloWorld.java OtherClass.java
jar cvmf manifest.mf myJar.jar HelloWorld.class OtherClass.class
@echo "To run the file, type: 'java -jar myJar.jar'"

To compile and execute, put all three files in the same directory and type the following:

$ make
javac -d . HelloWorld.java OtherClass.java
jar cvmf manifest.mf myJar.jar HelloWorld.class OtherClass.class
added manifest
adding: HelloWorld.class(in = 381) (out= 277)(deflated 27%)
adding: OtherClass.class(in = 392) (out= 272)(deflated 30%)
To run the file, type: 'java -jar myJar.jar'
$ java -jar myJar.jar
hi there, human!
$

The contents if the directory after all of this will be:

$ ls
HelloWorld.class HelloWorld.java MANIFEST.MF Makefile OtherClass.class OtherClass.java
$
2.1 Preliminaries

An Algorithm is simply a set of instructions. Generally, the instructions must be specific enough that one can estimate the amount of work required to complete each step. Multiple algorithms can exist to solve the same problem or complete the same task. How do we choose which algorithm to use?

The appropriate algorithm can be determined based on a number of factors:

1. How long the algorithm takes to run
2. What resources are required to execute the algorithm
3. How much space or memory is required
4. How exact is the solution provided by the algorithm (is estimation involved, what about floating point rounding?)

Let's look at an example, the 3 sum problem. The 3 sum problem is simple, given an array of numbers, how many triples of those numbers sum to the value 0:

```
int count(long[] a) {
    int N = a.length;
    int cnt = 0;
    for (int i = 0; i < N; i++)
        for (int j = i+1; j < N; j++)
            for (int k = j+1; k < N; k++)
                if (a[i] + a[j] + a[k] == 0)
                    cnt++;
    return cnt;
}
```

3-Sum analysis

- How do we expect the program to run?
- How do we quantify this expectation?
- What the properties of this algorithm do the quantization of expected running time apply to?

Running time is one way to quantify our expectation:

- Best case: the easiest inputs. The best we can do
- Worst case: The hardest inputs. The worst we can do
- Average case: What we expect on average, for random input
Note that best case doesn’t mean just a small input. Rather we can consider differing inputs of the same size with varying hardness.

What is the expected **best case, worst case, and average case** running time?

Here is 3-sum running on home.cs. Why is 300 doing the wrong thing?

Just to get an idea of running time and order:
Those numbers are nice, but they are hiding something. What are they hiding?

2 Different 3-Sum implementations

In the following are 2 different 3-sum implementations.

```
int count(long[] a)
{
    int N = a.length;
    int cnt = 0;
    for (int i = 0; i < N; i++)
        for (int j = i+1; j < N; j++)
            for (int k = j+1; k < N; k++)
                if (a[i] + a[j] + a[k] == 0)
                    cnt++;
    return cnt;
}
```

```
int count(long[] a)
{
    int N = a.length;
    int cnt = 0;
    for (int i = 0; i < N; i++)
        for (int j = i+1; j < N; j++)
            for (int k = j+1; k < N; k++)
                if (a[i] + a[j] + a[k] == 0)
                    cnt++;
    return cnt;
}
```

rhs = lower code
lhs = upper code
These were run on the same computer. Why is there such a large difference? what is the order of each algorithm?

### 2.2 Asymptotic Notation (Big-O and the others)

Recall, we were interested in describing the best, average, and worst case performance of an algorithm. For now, lets concentrate on just the worst case performance. To describe the worst case performance of an algorithm, we need to represent its work in terms of a function that takes a parameter defining the input size. For example, the 3-sum problem described by a cubic function whose parameter is the length of the input array. $3sum = 3sum(n) = n^3$.

To describe the worst case behavior of an algorithm, we need to identify a function that grows at the same rate as the expected running time of the algorithm. In the worst case scenario, we use **O notation**.

**Definition: Big-O**

A function $g(n)$ is said to be $O(f(n))$ if there are positive constants $c$ and $n_0$ such that $g(n) \leq cf(n)$ for any $n \geq n_0$.

In other words, $g(n)$ is bounded above by $f(n)$.
g(n) = O(f(n))

Some questions...

Is g(n) = O(f(n))?
Is f(n) = O(g(n))?

Is g(n) = O(f(n))?
Is f(n) = O(g(n))?

Is g(n) = O(f(n))?
Is f(n) = O(g(n))?

Big O is the worst case performance of an algorithm. We can also model best case, and average case. Mathematically, the various cases can be defined as follows:
### Computational Notation | Mathematical Equivalent
---|---
$T(n)$ is $O(f(n))$ | $\lim_{n \to \infty} \left| \frac{T(n)}{f(n)} \right| < \infty$
$T(n)$ is $\Omega(f(n))$ | $\lim_{n \to \infty} \left| \frac{T(n)}{f(n)} \right| > 0$
$T(n)$ is $\Theta(f(n))$ | $0 < \lim_{n \to \infty} \left| \frac{T(n)}{f(n)} \right| < \infty$
$T(n)$ is $o(f(n))$ | $\lim_{n \to \infty} \frac{T(n)}{f(n)} = 0$

Big O is a tight upper bound, while little O is a loose upper bound. In other words, if $g(n) = O(f(n))$, then there is no function $t(n)$ that bounds $g(n)$ from above but has a smaller order than $f(n)$.

For example, the function $g(n) = n + 3$ is $O(f(n) = n)$ and $o(t(n) = n^2 + 4)$. Here are some examples to illustrate:

**Examples of asymptotic analysis**

- $a(x) = x^2 + x$
- $b(x) = 5 \times x^2$
- $c(x) = x$

![Graph showing running time differential for input size (n)](image1)

![Graph showing running time differential for input size (n)](image2)
\[a(x) = x^2 + x\]
\[b(x) = x^2 / 2\]
\[a(x)/b(x) = \frac{x^2 + x}{x^2 / 2} = \frac{2x^2 + 2x}{x^2} = 2 + \frac{2}{x}\]
\[= 2(1 + \frac{1}{x})\]

Plug in some big numbers for x, what happens?

**NOTE:** \(a(x)/b(x) = 2\) as the limit approaches infinity. So, it fits the definition of a tight upper bound, tight lower bound, and average case bound. This is the case with the 3Sum analysis as well.

Another example:
\[a(x) = x^2 + x\]
\[c(x) = x\]
\[a(x)/c(x) = \frac{x^2 + x}{x} = x + \frac{x}{x} = x + 1\]

We do not have a limit = \(\infty\), but we do have a limit = 0. Limit = \(\infty\) is essentially the opposite of little o. We could call this little omega, \(a(x)\) is loosely bounded below by \(c(x)\)

What do the loose bounds actually tell us?

---

**Exercise**
Identify an algorithm that has different best case and worst case bounds for the same size input.

---

**The Usual Suspects**
The following functions often come up in asymptotic analysis
\[f(n) = k\text{ (constant)}\]
\[f(n) = k \lg n\text{ (logarithmic)}\]
\[f(n) = kn\text{ (linear)}\]
\[f(n) = kn \lg n\text{ (linearithmic)}\]
\[f(n) = kn^2\text{ (quadratic)}\]
\[f(n) = k2^n\text{ (exponential)}\]

---

**2.3 Finding the functions for an algorithm**

This discussion is quite nice, but we must come up with a model of converting algorithms to functions. To do this, we must make an assumption:

- Every basic operation takes 1 time unit to complete on a computer.

Let's define a basic operation as an arithmetic operation, a comparison operation, an assignment operation, a return from a function, etc. In general, if it can be represented by a low level assembly language instruction, we assume its implemented in 1 time unit.

**1 time unit per operation?**
Is the above assumption actually true? Provide some examples of when it is and when it isn’t. (hint, memory operations? floating point operations?, pipelining?)

given the above assumption, we can simply add up the number of operations a code segment takes to execute. Typically, it will vary based on input size.

Loop

```java
for (i = 0; i < n; i++)  // i+(n+1)+ n
    Z = 8 * i + 5;      // 3*n
```

Running time is $2n + 2 + 3n = O(n)$

Nested Loop

```java
for (i = 0; i < n; i++)  // i+(n+1)+n
    for (j = 0; j < n; j++) // n * (i+(n+1)+n)
        x += 4 * (i + j)   // n * n * (4)
```

Running Time is:

$$2 + 2n + 2n^2 + 2n + 4n^2 = 6n^2 + 4n + 2 = O(n^2)$$

Consecutive Statements

Just add all the consecutive statements together. (Or just keep the biggest part)

```java
for (i = 0; i < n; i++)  // i+(n+1)+ n
    Z = 8 * i + 5;      // 3*n
for (i = 0; i < n; i++)  // i+(n+1)+n
    for (j = 0; j < n; j++) // n * (i+(n+1)+n)
        x += 4 * (i + j)   // n * n * (4)
```

$$6n^2 + 4n + 2 + 2n + 2 + 3n = 6n^2 + 9n + 4 = O(n^2)$$

Conditionals: (if statements)

Use the most expensive branch:

```java
if ( x < 2 )  // 1
    x = 2;   // 1
else {
    for (i = 0; i < n; i++)  // i+(n+1)+n
        for (j = 0; j < n; j++) // n * (i+(n+1)+n)
            x += 4 * (i + j)   // n * n * (4)
}
```

Running Time is: $2 OR 6n^2 + 4n + 2 = O(n^2)$

Recursion

Recursion is trickier. Basically, you have to find the worst possible cost of a recursive case, and the worst possible cost of a non-recursive case (base case). Then you have to find out how many times a recursive case is called in a given function call. An example follow in the next section.
2.4 Recurrence Relations For Recursion

A recurrence relation is simply a function that describes the cost of a recursive function call in an algorithm. Because the function calls itself, we can plug the function into itself, and generate a series. The time complexity of the function is equivalent to the convergence of the series. An example follows.

### Simple Example

```cpp
int powerOf2(const int &n) {
    if (n == 0) // 1
        return 1; // 1
    return powerOf2(n-1) + powerOf2(n-1); // 4 (+1 to fail at the IF)
}
```

The base case through function, when \( n = 0 \), takes 2 time units.

The recursive case takes 5 time units, 1 to fail at the IF statement, and 4 to complete the function calls and add their results.

So, we can define a function \( t(n) \) that behaves as follows:

\[
t(0) = 2 \\
t(k) = 5 + 2T(k - 1) \text{ for } k > 0 \text{ (there are 2 function calls)}
\]

So, we can start plugging in values for the function:

\[
t(n) = 5 + 2T(n - 1) = 5 + 2(5 + 2t(n - 2)) = 5 + 2(5 + 2(5 + 2t(n - 3))) = 5(1 + 2 + 2^2 + 2^3 + \ldots + 2^{n-1}) + 2^n t(0) = O(2^n)
\]

### Another Example

```cpp
int betterPowerOf2(const int &n) {
    if (n == 0)
        return 1;
    return 2*betterPowerOf2(n-1);
}
```

\[
t(0) = 2 \\
t(k) = 4 + t(k - 1) \text{ for } k > 0 \\
t(n) = 4 + t(n - 1) = 4 + (4 + t(n - 2)) = 4 + (4 + (4 + (t(n - 3)))) = \ldots \\
= 4n + t(0) = 4n + 2 = O(n)
\]
3.1 Abstract Data Types (ADTs)

Lists stacks and queues are examples of collection abstract data types; basically, they hold a collection of objects. From a user’s perspective, the main difference is the API or Application Programming Interface that each collection provides. The API is the interface to collections. APIS provide a:

1. standard interface that is separate from the implementation of the actual algorithms.
2. reference for programmers to use the collection.

Important
The concept of an API is extremely important from a software engineering perspective! The API should be designed first! (why?)

The API should not change! (why?)

The concept of an ADT is another powerful software engineering concept. An ADT is purposefully ABSTRACT! In other words, it doesn’t matter what we are modeling, we simply model it in an abstract form keeping only the parts we need.

Example ADT
Lets create an ADT for a student in a student information system. We need to model a student, but only the parts we need for our application. So, a student might be a:

- student ID number
- name
- address
- emergency contact
- classes taken
- grades for classes
- credit hours for classes

We might also need to perform some operations on a student:

- register for a class
- calculate a GPA
- update an emergency contact
The data and operations together form an ADT. The operations form an API.

Note that the ADT and API have no mention of how any of the operations are implemented. The point is that users of the class don’t necessarily need to know, beyond some basic details. This allows us flexibility to update implementations without breaking existing code using the API (as long as the API doesn’t change).

So, lists, stacks and queues are 3 container ADTs with varying APIs. The question, and one of the goals of this class, is to determine which one should be used for a given situation.

### 3.2 Lists

**Definition of List**

A list is:

- an item
- an item followed by a list

What is in item? anything. Even an empty item.

Operations on a list:

- empty the list
- determine if the list is empty
- determine size of the list
- find an element
- determine value of an element at a particular location
- insert an element at a particular location
- remove an item
- print

There are 2 straightforward implementation of the list ADT with identical APIs. An array structure and a node structure (linked list).

**Exercise**

Implement both list structures.

Determine the time complexity of each list operation for each structure.

How can we improve/alter the time complexities.

Which version requires more/less memory?

C++ STL has 2 alternatives for list-like structures:

- vectors (actually dynamic arrays)
- lists (node structured linked lists)

Each have different performance characteristics. How does a list grow? How does a vector grow?
3.3 Stacks

A stack is like a stack of paper on a desk

1. What operations are available on stacks?
2. What are their time complexities? Can they be bad?

Stacks can be easily implemented as a list.

Exercise

Convert one of the list implementations to a stack.

Which implementation is better for a stack? why?

Stacks in Practice: Postfix notation

rules:
- When you see a number, push it
- When you see an operation,
  - Pop two times
  - Perform operation
  - Push the result

7 6 + 3 5 8 2 / + *

7 7 6 13 3 13 8 5 5 8 4 9 27 -14

Stacks in Practice: C++ run-time stack

Stores data and call information for nested scopes (often functions).

Grows downward from an origin

Stack pointer points to the top of stack

Every time you enter a new scope, all local variables and some scope info are pushed to the stack

If you know how the stack works, you can be root!
So, with this information, we can create a buffer overflow! This is exactly why most languages protect against array over-runs, but C++ is designed for SPEED, not safety!

```cpp
#include <iostream>
using namespace std;
```
```c
void f1( int theArray[] ) {
    for( int i = 0; i < 10; i++ )
        theArray[i] = i+20;
}

int main() {
    int a=3, b=4, c=5;
    int array[] = {1,2};
    cout << "a: " << a << endl
         << "b: " << b << endl
         << "c: " << c << endl << endl;
    f1( array );
    cout << "a: " << a << endl
         << "b: " << b << endl
         << "c: " << c << endl;
}
```

The output of the above code...

```
marmcke@beet$ ./a.out
a: 3
b: 4
c: 5
a: 24
b: 23
c: 22
marmcke@beet$
```

What's going on?

### 3.4 Queues

Not just an awesome scrabble word!

An ADT organized as a list in which insertions occur at one end and deletions occur at the other.

Think of standing in line.

Applications:
- printer queue
- (there is a queue application that most people use every single day, without realizing it. What is it?)

Implementation:
- Array (usually dynamic). This can be trickier, why? Let's give it a try.
- Node structure. How do we make constant time operations?

Note:
Queues and stacks are super useful as constructs to prove an algorithm runs in a certain amount of time (big-O). If you can prove that all items are put into a queue or stack exactly once, the running time of the algorithm is the running time of the push and pop operations! usually constant.
4.1 Unbalanced Binary Trees

4.1.1 Binary Trees

**Definition**

For a binary tree:

A tree is a:

- empty tree.
- an item.
- an item followed by \( 1 \leq x \leq 2 \) trees.

For an n-ary tree:

A tree is a:

- empty tree.
- an item.
- an item followed by \( 1 \leq x \leq n \) trees.

Note the recursive nature of the definitions. Recursion is a natural fit for trees.

A **forest** is a group of 2 or more trees

A binary tree API typically has the following operations:

- create a tree
- isEmpty
- contains( value )
- traverse // often through iterators
- remove item
- add item

**Tree Traversals:**

There are 3 classic tree traversals: **pre-order, in-order, and post-order**. They are summarized in the following code snippets:
preorder(node) {
  visit(node);
  preorder(node.leftChild);
  preorder(node.rightChild);
}

inorder(node) {
  inorder(node.leftChild);
  visit(node);
  inorder(node.rightChild);
}

If items in the tree are sorted (in a binary search tree), in-order visits items in order

postorder(node) {
  postorder(node.leftChild);
  postorder(node.rightChild);
  visit(node);
}

Post-order is useful for printing a tree such that the structure is visible.

For the following tree:
pre-order: HDBACFEGPKINURY
in-order: ABCDEFGHIJNPRUY
post-order: ????

Note about recursion
Recursion if often frowned upon in production code. Why?
What are the advantages of recursion?
What are the disadvantages of recursion?
If you need a recursive function, but you don’t want some of the disadvantages of recursion (call stack limitations),
you can essentially implement the stack yourself. In other words, use an iterative function, but push and pop data onto a stack to simulate recursion. This allows much more efficient stack management (less overhead) and larger stack sizes.

A last note. Binary trees can be implemented as arrays rather than node structures. For example:

To compute the children of node with array index $i$:

leftChild($i$) = ($i * 2$) + 1
	rightChild($i$) = ($i * 2$) + 2

Array construction of binary trees is super useful for complete or nearly complete trees. One issue to consider is how to tell if a node is empty (essentially, detect a null pointer)?

Array construction of binary trees will be super useful later.

4.1.2 Binary Search Trees

Definition

A binary search tree is a binary tree where each node contains an item such that a total ordering exists over the items. Furthermore, for any node $n$ containing item $x$, all items in the left subtree from $n$ are less than or equal to $x$, and all items in the right subtree from $n$ are greater than $x$.

Thus, the binary search tree must preserve both tree structure, and the order of the items.

Typically, binary search trees are used for searching for items

Search operation: Look at a node to see if it has the value we want, if what we are looking for is less than the value, visit the left subtree, otherwise visit the right subtree.

Insert operation. Perform a search, but continue until we hit a leaf. Insert the item at the leaf. **Inserting at leaves is EASY**, inserting in the middle of the tree is hard. we see this a lot with trees.

4.1. Unbalanced Binary Trees
Exercise
Insert the following into a binary search tree. Show the tree at each step:
9, 6, 8, 7, 3, 0, 10, 5

Remove operation: The easiest method is the remove from a leaf (just like with insert). So the general idea is copy whatever we want to delete to a leaf, then delete the leaf. Sometimes, we cannot find a leaf, so we must simply find a node with a single child pointer (which is an easy case to handle).

One problem, we must exchange the value of a node with a value in a leaf. And, we must do this while preserving the tree ordering (after the delete of course). So, we need to identify either the previous or successive value in the tree with respect to item ordering. Luckily, there is an easy formula to find these.

Finding the next/previous value in sequence in a BST for the delete operation
Consider the following tree:

The value in the tree immediately less than 9 can be found by visiting the left child of the node containing 9, then following right child pointers until a NULL pointer is found.

The value in the tree immediately greater than 9 can be found by visiting the right child of the node containing 9, then following left child pointers until a NULL pointer is found.

Note that in this case, if we want to delete 9, we can swap it with 8, resulting in:
To delete the node that now contains 9, we can simply connect the right child of the 6 node to the 7 node (because the 9 node only has 1 child):

```
9
  8
 6
 3   10
 0   5
 7   8
```

The idea is simply to never delete a node with 2 children.
Note that the structure and order is maintained in the tree.

Finally, the time complexities of BST operations are related to the height of the tree. Why?
What are the worst case and best case time complexities of search, insert, and remove?

**Balance and BSTs**

BSTs are NOT guaranteed to be balanced, but, if the input is inserted in a random order, they tend to be. For BSTs, randomness is your friend. Although there is no guarantee of balance, if the input is expected to be random, you can expect logarithmic operation time.

However, deleting from a BST tends to degrade randomness an favor more unbalanced trees. BSTs can be a very good and simple choice. There tends to be costs to achieve balance.

### 4.2 Balanced Trees

For operations on trees, the worst case cost of the operation is equivalent to the height of the tree. For a BST with items inserted in random order, the costs is roughly $O(lg \ n)$. Items inserted roughly in order lead to costs of $O(n)$. Insertions and deletions tend to make things worse (less random).

Instead of relying on the randomness of data ordering, lets look at some trees that are guaranteed to be balanced. The guarantee balance comes with a price, these trees are significantly more difficult to implement than traditional BSTs.

**Definition: Balanced Tree**

A balanced tree is a tree that is close to its minimal possible height (its longest branch is close to the minimal possible height for the number of items in the tree). For $n$ items, a balanced binary tree will have height close to $lg \ n$, a trinary tree will have height close to $log_3 \ n$, an k-ary tree will have height close to $log_k \ n$.

#### 4.2.1 2-3 Trees

A 2-3 tree node has space to hold 2 items and 3 pointers. A node will always have 1 item and 2 pointers (a 2 node), or 2 items and 3 pointers (a 3 node):
Items in nodes are ordered. So, the first item is always less than or equal to the second item (if there are 2 items in the node). Searching for a value is similar to a BST, except there are 3 possible pointers to follow. The algorithm is as follows:

```c
find( node*, item):
    if the item in the first slot?
        if Yes: done
        if No: Is the item less than item in first slot?
            if Yes: find( node.leftChild )
            if No: Is the item in second slot?
                if Yes: done
                if No: Is the item less than item in second slot?
                    if Yes: find( middleChild, item )
                    if No: find( rightChild, item )
```

**Inserting into a 2-3 tree**

The 2-3 tree is guaranteed to ALWAYS be balanced. Therefore, we cannot simply insert items by creating new leaves like we do with a BST. The process is as follows:

- Always insert a new item at a leaf, but never increase the depth of the tree at the leaves.
- Use the search algorithm to find the leaf that a new item should go in.
- If it is a 2 node, then just put the item there.
- If it is a 3 node, split the node into 2 nodes, and promote the middle of the 3 values up to the parent node.
- If the parent does not have room (it is already a 3 node), split that node and promote the middle value to its parent. Recursively repeat until no splitting occurs.

If this procedure is followed, the height of the tree will be increased at the TOP of the tree (the root). Basically, tree height will increase once a root is split. An example follows:

**Example 2-3 Tree Insertion**

Insert the following: 10 40 20 5 15 17 19
On paper, the insertion algorithm is not too bad. By drawing pictures, you can usually figure out where things should go. Actually implementing the code is another story. The following is a working 2-3 tree implementation. Try to summarize the portions of the code in pseudo-code. Make sure you know what each code block is doing. The downloadable source file is [here](#).

```python
class node:
    def __init__(self):
        self.data = [None, None]
        self.next = [None, None, None]
        self.size = 0

    def __str__(self):
        return str(self.data) + '(node has '+ str(self.size) + ' item(s))'

class tree23:
    def __init__(self):
        self.head = None

    def insert(self, item):
        # special case: empty
        if self.head == None:
            self.head = node()
            self.head.data[0] = item
            self.head.size = 1
            return

        # find leaf
        # leaf will ALWAYS have next[0] == None
        ptr = self.head
        myStack = [ptr]  # keep a stack of nodes visited back up to the root
        while (ptr.next[0] != None):
            if item < ptr.data[0]:
                nextIndex = 0
            elif ptr.size == 2 and item > ptr.data[1]:
                nextIndex = 2
            else:
                myStack.append(ptr)
                ptr = ptr.next[nextIndex]
        myStack.pop()  # remove myStack since we're not coming back
        ptr.next[nextIndex] = node()
        ptr.next[nextIndex].data[0] = item
        ptr.next[nextIndex].size = 1
```
nextIndex = 1
ptr = ptr.next[ nextIndex ]
myStack = myStack + [ptr]

# pop the current node off the stack
myStack.pop()

print 'INSERT -- we have found a leaf:', ptr
# now we are at a leaf. If we have room, just put it in
if ptr.size == 1:
    insertIndex = 1
    if item > ptr.data[0]:
        insertIndex = 0
    ptr.data = ptr.data[:(-1*(insertIndex+1))] + ptr.data[:(-1*(insertIndex+1))]
    ptr.data[insertIndex] = item
    ptr.size +=1
    return

print 'INSERT -- leaf is full, must split a node'
# final case, node is full.
# make a new siblind and parent. Set up the nodes
# we are at a leaf, so we don’t need to worry about pointers
leftNode = node()
leftNode.size = 1
rightNode = node()
rightNode.size = 1
parNode = node()
parNode.size = 1

# assign data and pointers
if item < ptr.data[0]:
    leftNode.data[0] = item
    parNode.data[0] = ptr.data[0]
    rightNode.data[0] = ptr.data[1]
elif item < ptr.data[1]:
    leftNode.data[0] = ptr.data[0]
    parNode.data[0] = item
    rightNode.data[0] = ptr.data[1]
else:
    leftNode.data[0] = ptr.data[0]
    parNode.data[0] = ptr.data[1]
    rightNode.data[0] = item
parNode.next[0] = leftNode
parNode.next[1] = rightNode
if len(myStack) == 0 :
    # we are splitting the root
    self.head = parNode
    return

# At this point, we have made a new sub-tree by splitting a node.
# parNode is the parent of the sub-tree

# now merge the parnode with the parent. possibly recursively splitting
done = False
while len( myStack) > 0 and not done:
    theParent = myStack.pop()
    print 'INSERT -- splitting back up the tree at parent node:', theParent
# if we have room in the parent node, merge it with parNode

```python
if (theParent.size == 1):
    insertIndex = 0
    if parNode.data[0] > theParent.data[0]:
        insertIndex = 1
    # shift data, and put the new value in
    theParent.data = theParent.data[(-1*(insertIndex+1)):] + theParent.data[:(-1*(insertIndex+1))]
    theParent.data[insertIndex] = parNode.data[0]
    theParent.size += 1
    theParent.next[insertIndex] = parNode.next[0]
    theParent.next[insertIndex+1] = parNode.next[1]
    done = True
    break
else:
    # need to merge the nodes
    # Each time through here we already have the sub tree built
    # we are just merging its root node with the parent node that was
    # pointing at the subtree we are working on.
    # parnode is the node to be merged. Its children are already set
    # theParent is the node parNode must be merged with
    # promote node is the node of the data that will be promoted up since
    # we only get here if theParent is full
    promoteNode = node()
    if parNode.data[0] < theParent.data[0]:
        print 'small'
        promoteNode.data[0] = theParent.data[0]
        promoteNode.next[0] = parNode
        promoteNode.next[1] = theParent
        promoteNode.size = 1
        theParent.data[0] = theParent.data[1]
        theParent.size = 1
        theParent.next[0] = parNode.next[0]
        parNode = promoteNode
        continue
    elif parNode.data[0] < theParent.data[1]:
        print 'med'
        promoteNode.data[0] = parNode.data[0]
        promoteNode.next[0] = theParent
        promoteNode.next[1] = parNode
        promoteNode.size = 1
        theParent.size = 1
        theParent.next[1] = parNode.next[0]
        parNode.data[0] = theParent.data[1]
        parNode.next[0] = parNode.next[1]
        parNode.size = 1
        parNode = promoteNode
        continue
    else:
        print 'big'
        promoteNode.data[0] = theParent.data[1]
        promoteNode.next[0] = theParent
        promoteNode.next[1] = parNode
```

4.2. Balanced Trees
promoteNode.size = 1
theParent.size = 1
parNode = promoteNode
continue
# if we never set done, then we broke all the way up to the root node
# so reset the root pointer
if not done:
    self.head = parNode

def printTree( self ):
    self.printRecursive( self.head, 0 )

def printRecursive( self, curr, depth ):
    # post order traversal
    # print right child, middle child, left child, then the current node
    # base case for recursion
    if curr == None:
        return

    # the following string will allow us to indent the nodes properly
    aString = ['	']*depth # make a list of a bunch of tab characters
    aString = ''.join( aString ) # convert the list into a string for printing

    printedNodeDelimiter = False
    if curr.size == 2:
        # if it is a 2 node, handle the rightmost pointer and right most data
        self.printRecursive( curr.next[2], depth+1 ) # recursively print the tree
        print aString, '-' # a character to mark the node
        if not printedNodeDelimiter:
            print aString, curr.data[1] # print the left most data
            print aString, '^' # a character to mark the node
            self.printRecursive( curr.next[0], depth+1 ) # recursively print the tree for the left pointer
    if not printedNodeDelimiter:
        print aString, '-' # a character to mark the node
        print aString, curr.data[0] # print the left most data
        print aString, '^' # a character to mark the node
        self.printRecursive( curr.next[0], depth+1 ) # recursively print the tree for the left pointer

    t = tree23()
t.insert( 1 )
t.printTree()
print '---------------------'
t.insert( 2 )
t.printTree()
print '---------------------'
t.insert( 3 )
t.printTree()
print '---------------------'
t.insert( 4 )
t.printTree()
print '---------------------'
t.insert( 5 )
t.printTree()
Delete: Deleting from a 2-3 tree is super tricky. Consider the following example. We can’t delete in that fashion because it breaks the search algorithm!
The basic idea to delete from a 2-3 tree is to do something similar to BST delete, but to enforce balancing. Basically, move the item to be deleted to a leaf node. Then remove the item. Then, if a node is empty, recursively merge nodes up the tree (possible shortening the tree). Merging nodes is a pain, and there a lot of special cases to code. Typically, you try to avoid merging by stealing data from other nodes, but this usually leads to more special cases to code. We will not focus on 23 tree delete.

**Roundup:**
- **Pro:** 2-3 trees are balanced! insert, search, delete are all $O(\lg n)$.
- **Con:** Super hard to implement, especially delete.
- **Con:** Nodes are not all full (wasted space?), but we get a balanced tree.
- **Con:** Hard to debug.

So, 2-3 trees are a nice introduction to balanced trees because they succinctly summarize the tradeoffs faced by all balanced trees: balanced trees are HARD to implement (especially fully with delete, etc). But, we get good guaranteed worst case performance!

### 4.2.2 2-3-4 Trees

One reason that 2-3 trees are so difficult is that when splitting/combining nodes, we end up having 3 items to deal with, but only space for 2 in a single node. This just makes things more difficult. 2-3-4 trees are similar to 2-3 trees, but allow 1, 2, or 3 items per node, alleviating that particular problem. Furthermore, we will implement the splitting strategy with a different philosophy to avoid recursive node splitting, greatly simplifying the code.

#### 2-3-4 Tree Definition:
A 2-3-4 tree is a height balanced search tree such that the leaves of the tree are all at the same depth. A node can contain 1, 2, or 3 items and 2, 3, or 4 pointers. Items in a node are sorted, and an item is always surrounded by valid pointers (except for leaf nodes).
The node is structured as follows:

```python
class node:
    def __init__(self):
        self.data = [None, None, None]
        self.ptr = [None, None, None, None]
```

The height of the tree will be somewhere between $O(lg n)$ and $O(log_4 n)$:

- Balanced binary trees $\rightarrow$ height $lg n$
- Balanced tri-nary trees $\rightarrow$ height $log_3 n$
- Balanced quad-nary trees $\rightarrow$ height $log_4 n$

**Insert Algorithm:**

Like a 2-3 tree, always insert at a leaf.

Recall that inserting into a 2-3 tree is complicated mainly because of the recursive code. We will take a different philosophy for 2-3-4 trees. Instead of recursively splitting, we will preemptively split full nodes as we descend the tree, so we will **NEVER** have to recursively split a node.

Thus, the algorithm is as follows:

Get the root node.
If the root is full:
    split the root, assign items and pointers
Get the pointer $P$ to the appropriate child node.
while $P$ is not null:
    if $P$ is full:
        split the node pointed to by $P$, assign items and pointers
    if $P$ is a leaf:
        add the item to the appropriate spot in $P$ (it will always have room)
        $P$ is assigned NULL (break out of the loop)
    else:
        Get the pointer $P$ to the appropriate child node

Let's see an example:

- **Insert 10**

- **Insert 17**

- **Insert 15**

- **Insert 12**
Exercise: Implement 2-3-4 Tree Insert

Here is a skeleton to get you started. I have essentially implemented the easy parts of the pseudocode above, and left the hard parts to you. One thing to note is that because we preemptively split down the tree, there are only 2 cases that need to be handled for insert.

```python
class node:
    def __init__(self):
        self.data = [None, None, None]
        self.ptr = [None, None, None, None]

    def isLeaf(self):
        return self.ptr[0] == None

    def getDataIndex( self, data ):
        if self.data[0] == None:
            return 0
        elif data <= self.data[0]:
            return 0
        elif self.data[1] == None:
            return 1
        elif data <= self.data[1]:
            return 1
        else:
            return 2

    def getPtrIndex( self, data ):
        if data <= self.data[0]:
            return 0
        elif self.data[1] == None:
            return 1
        elif self.data[1] == None:
            return 1
        else:
            return 2
```
```python
return 1
elif data <= self.data[1]:
    return 1
elif self.data[2] == None:
    return 2
elif data <= self.data[2]:
    return 2
else:
    return 3
class t234:
    def __init__( self ):
        self.root = node()

    def splitRoot(self, nodePtr):
        if nodePtr.data[2] == None:
            return nodePtr

        # Your code here
        return newRoot

    def split( self, nodePtr, index ):
        if nodePtr.ptr[index].data[2] == None:
            return

        # Your code here
        return

    def insert( self, data ):
        #check the split for root
        self.root = self.splitRoot( self.root )
        curr = self.root
        while curr != None:
            #check if we are at a leaf
            if curr.isLeaf():
                index = curr.getDataIndex( data )
                #for i in range( len( curr.data)-1, index, -1 ):
                #    curr.data[i] = curr.data[i-1]
                curr.data[(index+1):] = curr.data[index:2]
                curr.data[index] = data
                return
            else:
                #find pointer to follow
                index = curr.getPtrIndex( data )
                self.split( curr, index )
                index = curr.getPtrIndex( data )
                curr = curr.ptr[index]

    def printTree( self ):
        self.printRecursive( self.root, 0 )

    def printRecursive( self, curr, depth ):
        #print right to left, depth
        if curr == None:
            return

        # Finish the print function
```

4.2. Balanced Trees
**DELETE**

Deleting from a 234 tree is, again, hard. The idea, again, is to move the item to delete to a leaf, then remove it from the leaf. So, you swap the item to delete with its in-order successor. Then delete the item. If the node is then empty, you must merge nodes, again maintaining the tree properties of being perfectly balanced. Because you have 3 items in a node, it is possible to steal items from a neighbor.

**Roundup:**

The 234 tree is easier to implement than a 2-3 tree, but still tough, especially with delete. Again, there are lots of special cases with the delete algorithm that must be handled.

- Pro: height balanced, so good performance
- Con: increased memory usage (possibly 2 empty items and 3 empty pointers in every node)
- Con: still hard to debug

### 4.2.3 B+ trees

**Disk Architecture Review**

The hard disk is considered part of the memory hierarchy in computers (it is used to implement virtual memory). A typical memory hierarchy is:
Hard disks are typically the rotating magnetic platter type (SSDs are too expensive to completely replace them). Thus, HDs are the only mechanical device in your computer (other than fans and the DVD drive), and thus are the SLOWEST! HD access times are so long that computer users notice it takes a long time (think booting up the computer).

Because HD access times are so long, any data structure that needs to work well in a disk environment must be designed specifically for disk interaction.

Purpose:
• Long term, nonvolatile storage
• Lowest level in the memory hierarchy
• Slow, large, inexpensive

General structure
• A rotating platter coated with a magnetic surface
• A moveable read/write head to access the information on the disk

Typical numbers:
• 1 to 4 platters (each with 2 recordable surfaces) per disk of 1” to 3.5” in diameter
• Rotational speeds of 5,400 to 15,000 RPM
• 10,000 to 50,000 tracks per surface
• cylinder - all the tracks under the head at a given point on all surfaces
• 100 to 500 sectors per track. The smallest unit that can be read/written.

### B+ trees:

B trees and (more commonly used) B+ trees are a generalization of 2-3-4 trees to nodes with larger amounts of items and pointers. They are typically used for external memory search trees (in DATABASES!). Thus, they are typically referred to as an index structure. An index structure is simply database terminology for a data structure that provides faster data access than simply iterating through all items in a collection. B+ trees provide:

1. An index structure that is separate from the data.
2. Hard lower and upper bounds on size and time to complete operations
3. Design specific to disk based structures.

B trees and their variants are used in virtually every DBMS out there.

We need a concept of a key that uniquely defines a record (basically an item). We will store complete records in one file (the relation). We will create a new file to hold the index. The index will contain keys that identify records, and
the disk address of that record in the relation’s file. This means we don’t need to re-order the relation’s data file ever, we can simply append to it. Also, we can create as many index files as we want for whatever we want to search on.

We will talk about B+ trees, they are easier to use and manage, and work more naturally.

B+Tree

A B+tree contains 2 types of nodes: internal nodes and leaf nodes.

- internal nodes contain keys and pointers to other internal nodes or leaf nodes.
- leaf nodes contain keys and disk addresses of records in the relation’s data file.

A node is fixed size. Usually, a node is the size of a disk page. Sometimes it will be a multiple of the size of a disk page. This means the number of items stored in a node is dependent on the size of the item, and the size of the disk page (usually 4k).

let $b$ be the branching factor of nodes in a B+tree. $b$ indicates the number of child nodes for a node in the tree. Let $m$ be the actual number of keys in a node. For internal nodes, $\lceil b/2 \rceil \leq m \leq b$. For leaf nodes: $\lfloor b/2 \rfloor \leq m \leq b - 1$. Exceptions are made for the root node if there are simply aren’t enough items in the tree to reach the minimum threshold.

Node Structure: Each node contains the following structure: [pointer, key, pointer, key, pointer, ..., key]. Pointers in internal nodes point to other nodes. Pointers in leaf nodes contain disk addresses, expect for the last pointer which points to the next successive node to the “right” in the tree.

Note in B+ trees, keys are repeated in internal nodes and in leaves. This means every key will show up in a leaf. Some keys will show up in internal nodes. This is a major difference with standard B Trees.

B+Tree Example

```plaintext
b=3

Root

Keys in the tree
(stored in its own file)

3 5 11 30 35 100 101 110 120 130 150 156 157 179 180 200

Data in a separate file
```

Searching the tree. Search the tree like a binary search tree.

Insertion Into the Tree

1. Find the node at which to insert the item
2. If there is room in the node, insert it.
3. If there is no room, find the median value $x$.

4. Create two new nodes, place all keys $k \leq x$ in the left node, and all keys $k > x$ in the right node.

5. Move the median value $x$ up to the parent node of the split node, and connect the tree. If there is no room for the median value in the parent node, repeat steps 4-5 on that node.

B and B+trees are complete trees. They do not grow at the leaves, instead, they grow from the root!

Example

Put the following keys into a B Tree and a B+Tree. Let the branching factor $b = 4$:

20 5 8 40 23 31 1 10 33 41 22 7 16 44 27 35 11 9 2 17

How many nodes are visited to find the value 20? 33? 44? How many nodes are visited to traverse the tree starting with value 31? What is the time complexity of search and insert?

The algorithm for searching and inserting into a B+tree is straightforward, but the implementation of insertion can be tricky. In the worst case, a split can propagate up the entire tree to the root. It is useful to keep track of a stack of node pointers that lead back from the node of insertion to the root.

As with most trees, the time complexities of the tree are dependent upon the tree height, for B+trees, the tree is always balanced, but trees with more full nodes might be shorter than trees holding the same amount of data with less full nodes.

For example, a B+tree with branching factor $b$ and height $h$ can have:

- at most $b^h$ pointers to data records. why?
- at min $\lceil b/2 \rceil^h$ pointers to data records. why?
- a height bound for $n$ elements of $O(\log_b n)$, meaning logarithmic search, insert and delete!

Example

Assume a 4096 (4Kb) disk page and records with integer keys (4 bytes). Assume a 64 bit system (8 byte memory pointers). If B+tree blocks contain no header information, the we have $(4096 - 8)/(4 + 8) = 340$ keys per block (1 pointer and 1 key takes 12 bytes. the -8 is for the last pointer).

Lets assume a typical block will have 255 keys (somewhere between 340 and 340/2=170).

- With a root, we can store 255 pointers to records
- With a root and 255 leaves, we have pointers to $255^2 = 65,025$ records
- with a root and 255 internal nodes and 255 leaves for each internal node, we have pointers to $255^3 = 16.6mil$ records

So, such a B+tree with height 3 requires 3 disk accesses to search 16.6 mil records! Furthermore, the top few levels of a B+tree are typically buffered in memory, eliminating the need to disk access in many cases.

B+ Tree Delete

Again, like the 23 and 234 trees, delete is hard. In fact, because B+trees are used in databases, delete is often not worth it to implement. Instead, we use a lazy delete scheme. Instead of deleting and re-balancing, just mark an item as being deleted, and hopefully reuse the space later. Databases are in the business of remembering information, so delete should not happen too often. Think of an example.

A problem with this is wasted space, but hard drives are CHEAP!, so it is not such a problem.
I/O Complexity

We now have a new type of complexity, I/O complexity. Because disk access takes so long, the point is to have few I/O operations, and not worry too much about the in-memory stuff.

So, for a B+ tree with $a$ min nodes and $b$ max nodes and block size (disk page size) of $B$:

- Number of leaf blocks is $O(n/B)$
- IO complexity for all operations is $O(\log_B n)$
- Height of tree is $\Omega(\log_a n)$ and $O(\log_b n)$
- Time complexity to find is between $\Omega(f(a) \log_a n)$ and $O(f(b) \log_b n)$
- Where $f(b)$ is the time to find an element in a node

A final note when dealing with data that is big: Always remember your bandwidth

http://hothardware.com/News/Homing-Pigeon-Faster-Than-Internet-in-Data-Transfer/

Time to transfer 4GB at 2.04MB per second is ......
   4 hours, 39 minutes, and 37 sec

Time to transfer 2.57 PB == 2570000GB at 2.04Mbits per second is
   130821 Days 12 Hours 32 Minutes 13.54 Seconds == 358 years!

Size of a hard drive: .01 cubic foot

Cargo capacity of a Toyota Yaris: 25.7 cubic feet

Number of hard drives I can transport: 2570

If these are 1 TB hard drives, that’s 2.57 PB == roughly 20.56 peta bits

Time to drive to Chicago: 5hrs == 18000 seconds

Which gives a bandwidth of 1.14 Tbits/second == 142 GB/second

And so the saying is: “Never underestimate the bandwidth of a station wagon loaded with hard drives hurtling down the highway at 70mph”

4.2.4 Red Black Trees (RBTs)

2-3-4 trees are nice, but have some significant drawbacks:

1. they are hard to implement fully.
2. they have potentially a lot of wasted space in the nodes
3. delete is still a nightmare

A nice idea is to have the 2-3-4 tree advantages in a binary tree format. This is the idea behind red-black trees. Basically, for every 234 tree, there is an equivalent RBT. Basically, we are going to split up the 3 and 4 nodes into a group of binary nodes. The connected binary nodes are said to be connected with RED links, and the other links are black links. For example, a mapping between 234 and RB nodes follows:
Note: because red links are, in some sense, internal node links in the 2-3-4 tree, there can never be 2 consecutive red links on a root to leaf path through the tree!

Note: black links are pointers in the 2-3-4 tree, so if we compress the red links to re-create the 2-3-4 tree, all the tree height properties still remain:

Properties
\( n \) := number of nodes

\( H \) := tree height

\( B \) := height of tree with red links compressed (height of the equivalent 2-3-4 tree)

- Property: \( 2^B \leq n + 1 \leq 4^B \)
- Property: \( \frac{1}{2} \log(n + 1) \leq B \leq \log(n + 1) \)
- Property: \( \log(n + 1) \leq H \leq 2 \log(n + 1) \)

So, tree height is \( O(\log n) \)

With this style of implementation, the drawbacks to 2-3-4 trees (mostly) disappear. At least the wasted space disappears. Implementation is non-trivial, but easier than 2-3-4 trees.

So, a RBT node must contain an extra bit of information that records the color of the incoming link to each node.

**Insertion Algorithm**

Perform an insertion identical to binary search tree insertion (insert at the leaf! just like a 2-3-4 tree)

Color the incoming link to the newly inserted node RED

If the parent node of the newly inserted node also has a RED link coming into it, then we have 2 consecutive red edges

Fix the problem using rotations and promotion!

**Fixing the Tree!**

If the tree is unbalanced after an insertion, we perform rotations and promotions to fix it. Here are the cases:

Key:
- \( G \) is the grandparent of the inserted node
- \( P \) is the parent of the inserted node
- \( N \) is the newly inserted node

Case 1: No violation
Case 2: The incoming edge of $P$ is red and its sibling is black
Case 2.5: Same as Case 2, but the value of $N$ falls in between the value of $P$ and the value of $G$ in sequence.
If the new node is between its parent and grandparent in sequence, then we need another type of rotation:

![Diagram of violation and left-right double rotation]

Basically, do a left rotation with N, P, and P's left subtree, then a right rotation with P, N, and G. Again, the right-left double rotation is symmetric.

Case 3: The incoming edge of P is red and its sibling is also red. This is called a promotion. This will only occur after a rotation has already taken place. It may have to be recursively repeated up the tree.

![Diagram of violation and promotion]

Summary:
If 2 consecutive red edges are present immediately after an insertion, perform a rotation to restructure the tree. Only ONE rotation will be necessary. After that, traverse the path to the root and perform promotions if you come across any consecutive red edges.

- only 1 rotation per insertion
- promotions may continue up the tree \(O(\lg n)\) times
- time complexity is \(O(\lg n)\)

**Example of Insertion**

Insert the following numbers in this order: 10, 17, 15, 12, 20, 19, 18

As an exercise, identify the nodes in the equivalent 2-3-4 tree!

**Delete from a RBT**

Use the delete algorithm for a simple BST

If the node deleted was a red node, then we are done (we removed an item from a 3 or 4 node in the equivalent 234 tree)

If the node deleted was a black node (the incoming link is black), then we deleted a node from the equivalent 234 tree, and the entire thing is now unbalanced. We must rebalance the tree.

Re-balancing after a delete is similar to re-balancing after an insert, we do at most 1 restructuring, then we do recolorings (demotions instead of promotions) up the tree.
4.2.5 AVL Trees

AVL trees are very similar to red black trees, except that each node contains a number indicating its balancing factor rather than a boolean indicating a color value. If a node becomes unbalanced, then rotations, identical to RBT rotations, are used to recursively re-balance the tree. Like the other balanced trees, operations are $O(\log n)$

A very good C implementation of an AVL tree exists in the wild as the gnuAVL library. If you need a height balance binary search tree, and can’t find a built in language solution, I recommend it.

4.2.6 Splay Trees

Splay trees are an interesting tree. They are not guaranteed to be balanced, but they tend to be. Recall that that BST created with insertions of numbers in a random order tends to be balanced. Splay trees try to introduce randomness into a BST by performing rotations (similar to RBT rotations) when items are accessed. They also have a nice philosophical approach to finding items even faster than a typical BST.

Thought...
What does computer cache do?
Why?
Why does it work?
Does this hold for data structures?

The basic premise of the splay tree is that recently accessed items should percolate to the top of the tree (for the same reasons as cache). In order to do this, every time an item is accessed (inserted, searched for, etc.), rotations occur on the tree to raise the item to the root. The rotations cause the tree to shuffle its structure, leading to a tree that is usually pretty close to being balanced, but not guaranteed to be. The rotations used are as follows:

Splay Tree Rotations
A single rotation (similar to red black tree rotation), known as a ZIG:

A ZIG-ZIG double rotation:

A ZIG-ZAG double rotation:
Note that there are symmetric left/right versions of each rotation.

A single ZIG only occurs if no grandparent node exists (the parent is the root of the tree).

Here are two examples:
Because splay trees tend to be balanced, their operations tend to be \( O(\lg n) \). However, proving that a splay tree always performs within a constant factor of other height balanced binary search trees is an open problem.

- **Pro**: Easy to implement, no bookkeeping, just always bubble items to the top
- **Pro**: Tends to behave well (fast access times)
- **Con**: cannot guarantee \( O(\lg n) \), but we usually get it.
- **Con**: cannot guarantee balance.

### 4.2.7 Last word on Delete: Lazy Delete!

As we have seen, delete is often hard to implement and frequently it is just not worth it. Delete causes the following problems:

1. It’s hard to implement and debug
2. Tree restructuring can be expensive (especially for B+ trees on disk, but also in terms of memory operations on in memory trees). Also, changing the tree structure means other threads cannot access the tree while the restructuring is occurring.

Instead, we can relax the rules a little, and just mark an item as being deleted instead of actually deleting it. This allows us to reuse the space on future inserts.

Be careful, we cannot simply destroy the value, because it might be needed to properly search the tree. This is why we simply mark it as deleted. Try an example on one of the trees we have examined so far (234 trees are good for this).

- **Pro**: easier to implement, but must maintain deleted values until they are overwritten
- **Con**: possibly wasted space.
4.3 Tries

So far we have been focusing very intently on balanced trees. However, general purpose trees are not always the answer. We can always do better at an application if we can make assumptions about what is and isn’t allowed. The only requirement for search trees is that the items must have a total ordering and less than and an equal operation defined for them. But, what if what we want to store is text, or any data in a list form.

Tries are designed to store items constructed as a list of symbols from an alphabet. The most intuitive example is that tries are superuseful for storing strings.

A couple of asides:
1. The name trie comes from its use for re-trie-val, and so is pronounced like tree by some and like try by others. I will pronounce them like try to avoid confusion with other trees.
2. Implementations vary
3. Tries store any data items such that items are constructed as a list of symbols from some alphabet.

Tries are very fast for lookup and can exploit redundancy in items to save space.

We will look at the following:
1. Standard tries
2. Compressed tries (eliminating chains of nodes)
3. Compact tries (storing array indices instead of symbols)
4. Suffix tries (storing all suffixes of a string)

Standard Tries

The basic idea is that a node only contains a boolean value, and the pointer represents the character being stored. So, a string is constructed by examining its sequence of pointers throughout the data structure. For example, the following image shows a trie containing the words a, an, and, any, at. A node will have a boolean value of TRUE if a word ends at that node, and FALSE otherwise:

To find out if the word “an” is stored in the trie, we go the root and see if the “a” pointer is not NULL. We then follow the “a” pointer to another node and see if the “n” pointer is not NULL. We then follow the “n” pointer and see if the node has a value of TRUE, indicating that a word ends at that node.

Thus, for a trie containing lowercase character strings, a node is structured as follows:
Let:

- $n =$ total size of the strings to be indexed by the trie
- $d =$ length of the string being inserted/ searched for/ deleted in an operation
- $m =$ the size of the alphabet (remember case, punctuation, whitespace, numbers, etc)

Standard tries use:

- $O(n)$ space
- $O(d \times m)$ time for search/insert/delete

So they are super efficient!

An example, a word matching trie:

Watch out for memory!

For a trie with only lowercase letters, it has 26 pointers per node. A complete trie of depth 6 will have:

- $26^1$ pointers at level 1
- $26^2$ pointers at level 2
- $26^3$ pointers at level 3
- $26^4$ pointers at level 4
- $26^5$ pointers at level 5
- $26^6$ pointers at level 6
So, at level 6, we have $26^6 = 308,915,776$ bytes by pointer, which is $2,471,326,208 = 2GB$ of pointers at level 6 alone! This can be problematic.

**Compressed Tries**

Observation, if we have a chain of nodes with only 1 pointer, we can simply compress them into a single node:

**Compact Tries**

A compact representation of a compressed trie. Instead of storing all the symbols in a node, just store array indices:

**Suffix Trie**

4.3. Tries
A compressed trie of all suffixes of a piece of text. Its useful for finding patterns in any part of text

Example: IPDPS.

Suffixes:

- IPDPS
- DPS
- PDPS
- PS
- S

Let:

- n = length of a string X
- m = Alphabet size
- d = length of Pattern P

Properties:

- Uses $O(n)$ space
- Can be constructed in $O(n)$ time
- Find pattern P in X in $O(d \times m)$ time (Proportional to length of pattern, not text)

A suffix trie example:
Some uses of tries

- Search engine index:

  A collection of all searchable words stored in compressed trie.

  Each leaf of trie is associated with a word and a list of pages (URLs) containing that word (called occurrence list).

  Trie is kept in memory (fast).

  Occurrence lists kept in external memory and ranked by relevance.

- Computational Biology:

  DNA: sequences of 4 different nucleotides (ATCG). Portions of the sequence produce proteins, etc. It’s useful to find patterns.

  Genome: Master DNA sequence for an organism. For humans: 46 chromosomes and 3 billion nucleotides.

4.4 Roundup

Trees are great and easy to implement, but can degenerate into linear structures. Height balanced trees are great and guarantee logarithmic time operations (for operations based on the height of the tree), but are HARD to implement. If we make some assumptions about our data/application area, we can get away with not exactly height balanced trees (like splay trees that are easy to implement), or something like tries that are easy to implement and incredibly fast in terms of operation speed.

For each of the following data structures, you should be able to:

1. list the best case and worst case time complexities of the insert, search, and delete operation.
2. list the best case and worst case time complexity of iterating over all elements in the structure.
3. list pros and cons for each structure.
4. Given a scenario, you should be able to justify which structure you would use (or the inverse, provide scenarios that a particular structure is well suited to, including any assumptions you must make about the scenario).

5. Perform insertions and searches on the data structure.

6. Understand the approach to performing deletes on the structures, and be able to do it on paper (not be able to reproduce the algorithm)
   - Unbalanced Binary Trees
     1. Binary Trees
     2. Binary Search Trees
       - Balanced Trees
         1. 2-3 Trees
         2. 2-3-4 Trees
         3. B+ trees
     4. Red Black Trees (RBTs)
     5. AVL Trees
     6. Splay Trees
       - Tries
CHAPTER
FIVE

REVIEW OF MULTITHREADED PROGRAMMING

5.1 Sum3 Problem Overview

5.1.1 Basic Description

The Sum3 problem is described by a rather simple question: Given a set of \( n \) integers, how many triples of distinct elements in that list add up to 0.

For example, given the following list:

\[-1, -2, 0, 2, 3\]

the answer is 2:

\[-1 + -2 + 3 = 0\]
\[-2 + 0 + 2 = 0\]

The simplest way to code a solution to the Sum3 problem is to use a double nested looping structure to generate the indices of all possible triples in the input array. Clearly, this is simple to express in code, but has an unfortunate time complexity of \( O(n^3) \). For example, the following C++ code uses 3 for loops to achieve a correct solution.

```cpp
#include <iostream>
using namespace std;

int main()
{
    int dataSize = 5;
    int* data = new int[ dataSize ];
data[0] = -1;
data[1] = -2;
data[2] = 0;
data[3] = 2;
data[4] = 3;
    // do the naive Sum3 computation.  \( O(n^3) \)
    int count = 0;
    for (int i = 0; i < dataSize-2; i++)
        for (int j = i+1; j < dataSize-1; j++)
            for (int k = j+1; k < dataSize; k++)
                if (data[i] + data[j] + data[k] == 0)
                    count++;
    cout<< count <<endl;
}
```

65
5.1.2 Complete Naive Solution

The next code block uses the same solution as above, but includes a command line parameter that will generate an array of random numbers of a user-specified size. The `getRandInt()` function loads the vector passed into it with random integers. The `data` vector allocates space for the number of integers required, and is then passed to the `getRandInt()` function. This version of the program allows the user to get a feel for how and $O(n^3)$ program behaves in practice. Try compiling the program and running it.

```cpp
#include <iostream>
#include <sstream>

using namespace std;

int getRandInt(int* dataVec, int dataSize)
{
    // load up a vector with random integers
    int num;
    for(int i = 0; i < dataSize; i++)
    {
        // integers will be 1-100
        num = rand() % 100 + 1;
        if(rand() % 2 == 0)
        {
            // make some integers negative
            num *= -1;
        }
        dataVec[i] = num;
    }
}

int main(int argc, char * argv[])
{
    int dataSize = 0;

    if(argc < 2)
    {
        std::cerr << "usage: exe [num of nums] [optional seed value]" << std::endl;
        exit(-1);
    }

    std::stringstream ss1;
    ss1 << argv[1];
    ss1 >> dataSize;

    if(argc >= 3)
    {
        std::stringstream ss1;
        ss1 << argv[2];
        ss1 >> seed;
        srand(seed);
    }
    else
    {
        srand(0);
    }

    // create a data vector
    int* data = new int[dataSize];

    // load it up with random data
    getRandInt(data, dataSize);
```
Here are some running times for the previous program:

<table>
<thead>
<tr>
<th>Array Size</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>.106</td>
</tr>
<tr>
<td>1000</td>
<td>.804</td>
</tr>
<tr>
<td>1500</td>
<td>2.698</td>
</tr>
<tr>
<td>2000</td>
<td>6.382</td>
</tr>
<tr>
<td>2500</td>
<td>12.459</td>
</tr>
<tr>
<td>3000</td>
<td>21.519</td>
</tr>
<tr>
<td>3500</td>
<td>34.162</td>
</tr>
</tbody>
</table>

The running times are plotted. The cubic nature of the curve is clearly visible.

Exercise

Create a Makefile and use the Makefile to compile the program listed above. Run the program using the time command on linux to see some running times. Collect running times on two different machines. Write a brief report indicating your results and listing the machine specifications. List reasons why one machine is faster than the other.

To find out machine specs on linux, check out the lscpu and lscpu commands, and check out the /proc filesystem. Check the man pages and/or google for information on how to use/interpret these resources.
5.2 Sum3 Simple Implementation (Serial)

For an input array of arbitrary integers (that is, we make no assumptions as to the range of the integers), Sum3 can be solved in $O(n^2)$ time using relatively simple algorithms.

The purpose of this chapter is to emphasize the point that parallelism is not always the best solution to a problem. Sometimes, making algorithmic improvements (resulting in better theoretical complexity) can provide better speedups than parallelism; especially if you don’t have the hardware resources to take advantage of a parallel algorithm.

5.2.1 Using a Hash Table

The easiest approach to code is to take advantage of hash tables. Hash tables are implemented in many languages, so they are cheap from a coding perspective, and they offer $O(1)$ time for inserting an element and determining if an element is in the hash table ... under ideal circumstances. The approach is to:

1. Load all numbers in a hash table
2. Generate all possible pairs of numbers
3. Check if the negation of the sum of each pair of numbers is in the hash table.

However, this approach suffers from some drawbacks. First, hash table operations can degrade to linear time under certain conditions (although this can be managed rather easily). Second, care must be taken to ensure that unique triples are identified. For example, if the input array was $-2, 0, 4$, one must ensure that $-2+(-2)+4 = 0$ does not get counted as a valid triple (since -2 only appears once in the input).

5.2.2 Using a Sorted Input Array

A slightly more complicated $O(n^2)$ algorithm requires that the input array be sorted. The algorithm contains a loop that will traverse the input array from first to last.

Assume a sorted input array array containing $n$ elements. We will first find all triples that add up to 0 and involve the first element in array. We set $a=\text{array}[0]$. A triple is then formed with $b=\text{array}[0+1]$ and $c=\text{array}[n-1]$. If $a+b+c > 0$, then we need to reduce the sum to get it closer to 0, so we set $c=\text{array}[n-2]$ (decrease the largest number in the triple to get a smaller sum). If $a+b+c < 0$, then we need to increase the sum to get it closer to 0, so we set $b=\text{array}[0+2]$ (increase the smaller number to get a larger sum). This process continues until $b$ and $c$ refer to the same array element. At the end of this process, all possible triples involving the first array element have been computed.

To compute all triples involving the first array element, we took a pair of numbers from the array. After examining the pair, one number was excluded from further consideration in future triples involving the first array element. Therefore, for $n$ array elements, we will construct $n-2$ triples due to the following: the first array element is used in every triple, leaving us to create pairs of the remaining $n-1$ elements; the first pair uses two elements and eliminates one from further consideration; all following pairs will reuse 1 element form the previous pair, and eliminate one element from consideration. Thus, computing all triples involving a single element requires $O(n)$ time. Repeating this for $n$ elements in the input array requires $O(n^2)$ time. A C++ implementation of this algorithm follows:

```cpp
#include <iostream>
#include <sstream>
using namespace std;

int getRandInt( int *dataVec, int dataSize )
{
    // load up a vector with random integers
    int num;
    for( int i = 0; i < dataSize; i++ )
```
int main ( int argc, char * argv[] )
{
    int dataSize = 0;

    if ( argc < 2 ) {
        std::cerr << "usage: exe [num of nums] [optional seed value]" << std::endl;
        exit ( -1 );
    }

    std::stringstream ss1;
    ss1 << argv[1];
    ss1 >> dataSize;

    if ( argc >= 3 ) {
        std::stringstream ss1;
        int seed;
        ss1 << argv[2];
        ss1 >> seed;
        srand( seed );
    }
    else {
        srand ( 0 );
    }

    // create a data vector
    int *data = new int[ dataSize ];

    // load it up with random data
    getRandInt( data, dataSize );

    // sort the data
    sort( data, data + dataSize );

    // do the Sum3 computation. O(n^2)
    int count = 0;
    int a,b,c, sum; // array elements
    int j,k; // array indices
    for ( int i = 0; i < dataSize-2; i++ ){
        a = data[i];
        j = i+1;
        k = dataSize-1;
        while( j < k ) {
            b = data[j];
            c = data[k];
            sum = a+b+c;
            if( sum == 0 ){
                cerr << a << " " << b << " " << c << endl;
            }
        }
    }

    return 0;
}
One pitfall in this algorithm is that duplicate values cause problems.

Example
Consider the input string array = [-4, 1, 1, 3, 3]. There are 4 triples that sum to 0:


However, the above code will only find 2 triples. The algorithm progresses as follows, with items being examined in bold:

-4, 1, 1, 3, 3 sum = 0, so index k is decremented
-4, 1, 1, 3, 3 sum = 0, so index k is decremented
-4, 1, 1, 3 sum = -2, so index i is incremented, resulting in j == k and the while loop exits

Effectively, the algorithm only finds triples involving the first 1, and none involving the second 1.

In fact, duplicates cause 3 problems that must be addressed:

1. 3 or more 0s will independently sum to 0. In fact, for \( n > 2 \) 0s, there will be \( \binom{n}{3} \) triples that sum to 0.

2. A sequence of repeating numbers exists such that a pair of those numbers along with a third forms a triple that sums to 0. For example: “-4, 1, 2, 2, 2” contains 3 triples that sum to 0. When such a sequence is detected with length \( n > 2 \), there will be \( n \choose 2 \) triples that sum to 0. Note that a correct handling of this case can also handle the case where 3 or more 0s exist.

3. The situation in the above example when two sequences of repeating numbers exist such that the repeated element of each sequence along with a third number not equal to those elements sum to 0. For sequences of non-equal numbers x and y with respective lengths \( m \geq 1 \) and \( n \geq 1 \), such that there exists a number \( z \mid x+y+z = 0 \), there will be \( x*y \) triples that sum to 0 for each individual copy of \( z \).

The following program handles duplicates correctly. Problem 1 and Problem 2 are handled in a single conditional, the first if statement in the while loop. Note that when such a situation occurs, we have effectively found the point at which j and k converge, and so we break out of the while loop. Problem 3 is handled in the second if statement in the while loop; the modification of j and k are again handled specially in this situation (since they may be increased...
or decreased by more than 1). The third if statement in the while loop handles the situation in which duplicates do not occur. The final else block in the while loop handles the normal increment of $j$ or decrement of $k$ when a triple with a non-zero sum is visited. The complete solution for handling duplicates is:

```cpp
#include <iostream>
#include <sstream>

using namespace std;

int getRandInt( int *dataVec, int dataSize )
{
    // load up a vector with random integers
    int num;
    for( int i = 0; i < dataSize; i++ ) {
        // integers will be 1-100
        num = rand() % 100 +1;
        if( rand( ) % 2 == 0 ) {
            // make some integers negative
            num *= -1;
        }
        dataVec[i] = num;
    }
}

int main(int argc, char * argv[] )
{
    int dataSize = 0;

    if( argc < 2 ) {
        std::cerr << "usage: exe [num of nums] [optional seed value]" << std::endl;
        exit( -1 );
    }
    std::stringstream ss1;
    ss1 << argv[1];
    ss1 >> dataSize;

    if( argc >= 3 ) {
        std::stringstream ss1;
        int seed;
        ss1 << argv[2];
        ss1 >> seed;
        srand( seed );
    }
    else {
        srand( 0 );
    }

    // create a data vector
    int *data = new int[ dataSize ];

    // load it up with random data
    getRandInt( data, dataSize );

    // sort the data
    sort( data, data + dataSize );

    // do the Sum3 computation. $O(n^2)$
    int count = 0;
    int a,b,c, sum; // array elements
```
```c
int j,k;  // array indices
for (int i = 0; i < dataSize-2; i++) {
    a = data[i];
    j = i+1;
    k = dataSize-1;
    while( j < k ) {
        b = data[j];
        c = data[k];
        sum = a+b+c;
        if( sum == 0 && b == c ) {
            // case where b == c. ie, -10 + 5 + 5
            // or where a == b == c == 0
            int num = k-j+1;
            count += (num*(num-1))/2;
            break;
        }
        else if( sum == 0 && (data[j+1] == b || data[k-1] == c ) ){
            // case where there are multiple copies of b or c
            // find out how many b’s and c’s there are
            int startj = j;
            while( data[j+1] == b ) j++;
            int startk = k;
            while( data[k-1] == c ) k--;
            count += (j-startj+1) * (startk-k+1);
            j++;
            k--;
        }
        else if( sum == 0 ){
            // normal case
            count++;
            j++;
        }
        else {
            // if sum is not 0, increment j or k
            if( sum < 0 ) j++;
            else k--;
        }
    }
    cout<< count <<endl;
}
```

Here are some running times for the previous program:

<table>
<thead>
<tr>
<th>Array Size</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.005</td>
</tr>
<tr>
<td>1000</td>
<td>0.008</td>
</tr>
<tr>
<td>1500</td>
<td>0.013</td>
</tr>
<tr>
<td>2000</td>
<td>0.019</td>
</tr>
<tr>
<td>2500</td>
<td>0.026</td>
</tr>
<tr>
<td>3000</td>
<td>0.033</td>
</tr>
<tr>
<td>3500</td>
<td>0.043</td>
</tr>
<tr>
<td>4000</td>
<td>0.054</td>
</tr>
<tr>
<td>10000</td>
<td>0.299</td>
</tr>
<tr>
<td>20000</td>
<td>1.142</td>
</tr>
</tbody>
</table>

Here we show the graph of running times for the cubic and quadratic versions of the algorithm. Notice the vast difference in running times for the two programs. Again, the cubic and quadratic nature of the algorithms are visible.
Exercise

Create a Makefile and use the Makefile to compile the program listed above. Run the program using the `time` command on Linux to see some running times. Collect running times on two different machines. Write a brief report indicating your results and listing the machine specifications. List reasons why one machine is faster than the other.

To find out machine specs on Linux, check out the `lspci` and `lscpu` commands, and check out the `/proc` filesystem. Check the man pages and/or google for information on how to use/interpret these resources.

5.3 Sum3 Parallel Implementation with Pthreads and OpenMP

At this point we have seen two possible algorithms and implementations for the Sum3 problem.

1. A $O(n^3)$ algorithm that is extremely easy to implement, but that suffers from poor execution time.
2. A $O(n^2)$ algorithm that is much faster in terms of execution time, but that requires care to handle sequences of repeating values correctly.

The question now whether or not we can do better. It turns out that algorithmically, we cannot...at least not yet; there are currently no known algorithms to solve the Sum3 problem faster than $O(n^2)$ time. In fact, a class of problems exists called Sum3 hard problems [GOM1995CG]. Any problem that is constant time reducible to a Sum3 problem is Sum3-hard; the implication being that a sub-quadratic time solution to any problem in the Sum3-hard class provides a sub-quadratic time solution to Sum3.

5.3.1 Introducing Parallelism with Multiple Threads

In terms of running time, we can do better with the Sum3 problem. One way to improve running time is to utilize a multi-core CPU with a multi-threaded program. The general approach is to divide the work that must be completed among multiple cores. Ideally, if we evenly divide the work among two processors, the program should run twice as fast. Another way to express this is to say that the program should have a 2x **speedup** over the serial version. The concept of speedup is a useful measure to gauge the effectiveness of a particular optimization, and is defined as follows:

**Definition**
Let $t_{\text{old}}$ be the running time of a program and $t_{\text{new}}$ be the running time of the same program that has been optimized. The speedup of the new program is equal to the running time of the old program divided by the running time of the new program:

$$\text{speedup} = \frac{t_{\text{old}}}{t_{\text{new}}}$$

### Design of a Multi-Threaded Algorithm

Designing a multi-threaded program, or a parallel algorithm for that matter, requires a different thought process than defining a serial program. When defining a serial algorithm, the focus is usually to determine the steps that must be completed in succession to achieve an answer. When developing a multi-threaded program, one must think in terms of how the work can be divided among threads. It is sometimes useful to begin with a serial program and try to modify it to divide work, and other times it is easier to simply start from scratch.

#### Concept

**How should the work be divided among threads?**

In order to effectively parallelize work, each thread must be able to do a portion of the work. Ideally, each thread will take roughly the same amount of time to execute; it is generally undesirable for a bunch of threads to complete their work quickly and sit around waiting on a single, slow thread to finish.

Lets take the cubic version of the Sum3 algorithm as an example (the core of the program is listed below). The main work of the program consists of the double nested loops that generate all possible triples of numbers from the input array. A reasonable method of dividing the work is to make 2 threads such that each thread generates roughly half of the triples that must be tested. We can achieve this by modifying the outer loop such that one thread will only compute triples in which the first number in the triple comes from the first half the array, and the second thread will compute triples in which the first number comes from the second half of the array. Such a modification requires minor changes to the core of the program.

```c++
#include <iostream>
using namespace std;

int main( )
{
    int dataSize = 5;
    int* data = new int[ dataSize ];
    data[0] = -1;
    data[1] = -2;
    data[2] = 0;
    data[3] = 2;
    data[4] = 3;
    // do the naive Sum3 computation.  O(n^3)
    int count = 0;
    for (int i = 0; i < dataSize-2; i++)
        for (int j = i+1; j < dataSize-1; j++)
            for (int k = j+1; k < dataSize; k++)
                if (data[i] + data[j] + data[k] == 0)
                    count++;
    cout<< count <<endl;
}
```

Chapter 5. Review of Multithreaded Programming
Cubic Sum3 Using Pthreads

**Pthreads** are a low level threading mechanism that have been around for a long time. Pthreads are provided by a C library, and such, they do things in a very old-school, C way of thinking. The basic idea that the programmer defines a function, and then tells the pthread library to create a thread to run that function. Therefore, we have a concept of a master thread, which is the thread of the program that is started when the program is initially executed, and the concept of worker threads, which are spawned by the master thread. Typically, the master thread will spawn some worker threads, and then wait for them to complete before moving on, as illustrated in the following:

![Diagram of thread creation and execution]

In order to achieve a pthread version of the program, we must first put the code that each thread will execute into a function. We will then call a pthread library call and tell it to use that function as the code the thread will run. The pthread library will then launch the thread. Our master thread can then launch additional threads, do other work, or wait on the launched threads to finish executing. Here are the function calls we will use... you should check your system’s man pages for the most up to date information.

**Pthread Library Calls**

```c
int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void *), void *restrict arg);
```

Create a thread. The first argument will be assigned a pointer to a thread handle that is created. The second argument is a pointer to a thread attributes struct that can define attributes of the thread, we will simply pass a NULL to use defaults. The third argument is the function pointer to the function that this thread will execute. The final pointer is a pointer to a single argument that will be passed to the function. Note, to use the argument, the function must type cast it to the appropriate type. Also, if more than 1 arguments are required, you must pack them into a struct. Returns a 0 on success, a non-zero error code on failure

```c
int pthread_join(pthread_t thread, void **value_ptr);
```

Join a thread. The join command will cause the calling thread to wait until the thread identified by the thread handle in the first argument terminates. We will pass NULL as the second argument

Returns a 0 in success, a non–zero error code on failure
Now that we have the ability to create and join threads, we can create a simple pthread program. The following example shows how to create threads to do some work, and pass arguments to them. Remember that pthreads require the function that will be executed as a thread to take a single argument; so we must wrap all the arguments we want to pass to the function into a single struct. We define the struct on line 23, and instantiate two instances of the struct on line 87. Because a pointer to the struct is passed to our `partial3SumThread` function as a `void *`, we must cast the pointer back to the correct type on line 33 before we can use it in the function.

We want to pass the input array to each thread, so we can simply pass it to the function.

**Note**

The function launched as a thread by the pthread library must take a single argument of type `void` pointer. This argument must be type-cast to a `void` to pass it into the function, and type-cast back to its original type within the function. Of course, this breaks type checking for the compiler, so be careful and double check the types yourself!

Here is the code:

```c++
#include <iostream>
#include <sstream>

using namespace std;

int getRandInt(int *dataVec, int dataSize)
{
    int num;
    for (int i = 0; i < dataSize; i++) {
        num = rand() % 100 + 1;
        if (rand() % 2 == 0) {
            num *= -1;
        }
        dataVec[i] = num;
    }
}

// define a struct to pass information to the threads
struct threadInfo{
    int myID;
    int *dataPtr;
    int dataSize;
    int count;
};

void* partial3SumThread(void* arg) {
    // type cast the argument back to a struct
    threadInfo *myInfo = static_cast<threadInfo*>(arg);
    // each thread only works on half the array in the outer loop
    // compute the bounds based on the ID we assigned each thread.
    // remember, we only have 2 threads in this case, so we will hard code a 2
    int start = (myInfo->dataSize / 2) * myInfo->myID;
    int stop = (myInfo->dataSize / 2) * (myInfo->myID + 1);
    if (myInfo->myID == 1)
        stop = myInfo->dataSize - 2;
```
// do the naive Sum3 computation. 0(n^3)
for (int i = start; i < stop; i++)
    for (int j = i+1; j < myInfo->dataSize-1; j++)
        for (int k = j+1; k < myInfo->dataSize; k++)
            if (myInfo->dataPtr[i] + myInfo->dataPtr[j] + myInfo->dataPtr[k] == 0)
                myInfo->count++;

int main(int argc, char * argv[])
{
    int dataSize = 0;
    if (argc < 2 ) {
        std::cerr << "usage: exe [num of nums] [optional seed value]" << std::endl;
        exit( -1 );
    }
    std::stringstream ss1;
    ss1 << argv[1];
    ss1 >> dataSize;
    if( argc >= 3 ) {
        std::stringstream ssl;
        int seed;
        ssl << argv[2];
        ssl >> seed;
        srand( seed );
    }
    else {
        srand( 0 );
    }
    // create a data vector
    int *data = new int [ dataSize ];
    // load it up with random data
    getRandom( data, dataSize );
    // allocate thread handles
    pthread_t worker1tid, worker2tid;
    // allocate and set up structs for 2 threads
    threadInfo info1, info2;
    info1.myID = 0;
    info1.dataPtr = data;
    info1.dataSize = dataSize;
    info1.count = 0;
    info2.myID = 1;
    info2.dataPtr = data;
    info2.dataSize = dataSize;
    info2.count = 0;
    // allocate space for a return value
    int returnVal;
    // call the worker threads
    if ( returnVal = pthread_create( &worker1tid, NULL, partial3SumThread, &info1 ) ) {

5.3. Sum3 Parallel Implementation with Pthreads and OpenMP
cerr<< "pthread_create 1: " << returnVal <<endl;
exit( 1 );
}
if ( returnVal = pthread_create( &worker2tid, NULL, partial3SumThread, &info2 ) ) {
cerr<< "pthread_create 2: " << returnVal <<endl;
exit( 1 );
}

// now wait on the threads to finish
if ( returnVal = pthread_join( worker1tid, NULL ) ) {
cerr<< "pthread_join 1: " << returnVal <<endl;
exit( 1 );
}
if ( returnVal = pthread_join( worker2tid, NULL ) ) {
cerr<< "pthread_join 2: " << returnVal <<endl;
exit( 1 );
}

cout<< info1.count + info2.count <<endl;
}

The program has a relatively simple structure, the core of the Sum3 computation is done in the partial3SumThread function. 2 instances of the function are launched, and we assigned the first instance an ID of 0, and the second instance and ID of 1. These thread IDs are manually assigned and stored in a threadInfo struct that we create. The structs are created and initialized on lines 87-95. Based on the thread’s ID, we determine which half of the array a thread should use in the outer loop of the Sum3 computation (lines 39-42). A thread is launched on line 100 and line 104. Once all the threads are launched, the main thread must wait for them to complete before printing the result, so the main thread calls one join function for each thread (lines 109 and 114). Once all worker threads have joined, we print the result.

Here are some running times on a machine with a 2 core cpu, and the results graphed along with the serial and serial quadratic versions of the algorithm:

<table>
<thead>
<tr>
<th>Array Size</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>.01</td>
</tr>
<tr>
<td>200</td>
<td>.019</td>
</tr>
<tr>
<td>300</td>
<td>.029</td>
</tr>
<tr>
<td>400</td>
<td>.063</td>
</tr>
<tr>
<td>500</td>
<td>.125</td>
</tr>
<tr>
<td>600</td>
<td>.207</td>
</tr>
<tr>
<td>700</td>
<td>.324</td>
</tr>
<tr>
<td>800</td>
<td>.475</td>
</tr>
<tr>
<td>900</td>
<td>.668</td>
</tr>
<tr>
<td>1000</td>
<td>.912</td>
</tr>
</tbody>
</table>
The pthreads version is clearly faster. As expected, the version with two threads runs roughly twice as fast as the version with 1 thread. For an array of 500, the speedup is 1.9, for an array of 1000, the speedup is 2.08. A speedup of more than 2 is likely due to timing granularity. The Linux `time` command was used, which is not the most accurate time keeping method. Other considerations should have an impact. For example, the two threads could have an effect on cache, causing a greater amount of cache hits or misses than the single threaded version. Cache misses cause the processor to stall, wasting time. More cache hits means the processor stalls less, and does the work in a shorter amount of time.

Exercise

The Pthreads version of the Sum3 problem is hard coded to handle 2 threads. Convert the program such that it will take the number of threads to use as a command line argument, and then generate that many threads. Hint: you will have an easier time of it if you use vectors to hold all thread information. For example, you will have a vector of threadInfo structs, one for each thread.

Thread Communication

In the Pthreads version of Sum3, very little thread communication took place. In fact, the only thread communication that happened was the master thread provided some data to the worker threads in the form of a function argument. Usually, threads will communicate with each other through shared memory. The key to understanding shared memory is understanding scoping rules. Basically, a thread will have access to all memory locations that are in scope for that thread, and all memory locations accessible through pointers that are in scope for that thread. This means that if a variable has global scope, then ALL threads will be able to read and write to it. This means, two threads can communicate by reading and writing to such a value. In our program above, we declared two instances of the threadInfo struct in the main() function (line 87); those instances are info1, info2. Those structs now exist in memory, but the names info1, info2 exist in scope for the main() function. We store information that the threads need in those structs. To provide access to that information, we pass the pointer to one of those structs to each thread during the pthread_create call. This means that a thread can access its struct through that pointer. Essentially, the master thread has communicated with the worker threads, letting them know their thread ID, the input array, etc.

Note that in the program, only 1 copy of the data array exists. It is created in the main() function. Pointers to that array are stored in the threadInfo structs, so that each thread can access the same array. This does not cause any problems because both threads only read the array, they are not making changes to it. Also note that each struct has its own memory location to store a count. Thus, each thread is keeping track of its own count. This is why all the count values from the threadInfo structs must be summed in line 119.
Exercise
Change the pthreads Sum3 program such that both threads access the same memory location for keeping track of their count variable. Don’t use any other pthread library features. Run the program multiple times. What happens to the result value? why?

5.3.2 Doing Things a Little Differently with OpenMP

Pthreads is just one method to create multi-threaded programs. Most languages have some built-in mechanism or library to launch threads. Another mechanism in C/C++ is OpenMP. One drawback to pthreads is that although the core Sum3 computation changed very little in the pthread program, the program did require some somewhat significant structural changes. Recall that we had to put our computation into a function, set up a struct to pass data to the function, introduce create and join function calls, etc. Furthermore, we only implemented 2 threads, implementing more threads requires even more work. So, there is some programming overhead to converting a serial program to a parallel program using pthreads.

OpenMP takes a compiler-based approach to threading. Instead of inserting function calls and re-organizing code to create threads, OpenMP requires you to place compiler directives near portions of code that will be run in parallel so the compiler can generate a team of threads. These directives are known as pragmas. One of the goals of OpenMP is to provide a mechanism whereby serial code can be parallelized using multiple threads with only minor code modifications. The result is that you can write a serial program, debug it while it is serial (a much easier task than debugging parallel code), and then simply add a pragma to achieve parallelism using multiple threads.

As an example, we will write a OpenMP version of the Sum3 problem with a similar structure as the Pthread version: we will create 2 threads, and each thread will only iterate over a portion of the outer loop. We will need the following OpenMP pragmas and functions:

### OpenMP Interface

```c
#pragma omp parallel
{
    // some code
}
```

Will create a team of \( n \) threads where \( n \) is the number of computational cores available on the computer on which this code is executed. Each thread will execute the code in the code block. No code will be executed beyond the code block until all threads have joined.

```c
int omp_get_thread_num();
```

Returns the unique thread identifier of the thread that executes this function. If only a single thread exists, the function always returns 0. If multiple threads exist, the function returns a number in the range from 0 to \( \text{omp_get_num_threads}()-1 \).

```c
int omp_get_num_threads();
```

Returns the number of threads in the current team. If only 1 thread exists (the serial portion of the code), it returns 1.

```c
void omp_set_num_threads( int num_threads );
```

Sets the default number of threads to create in subsequent parallel sections (for example, sections defined by \#pragma omp parallel).

```bash
g++ -fopenmp source.cpp
```

remember to use the OpenMP compiler flag to compile programs using OpenMP.
With these pragmas and functions, we can duplicate the Pthreads version of the program by forcing 2 threads to operate on the data. We will call `omp_set_num_threads(2)` (line 57) to force 2 threads, then break up the work similarly to what we did before. Much like Pthreads, OpenMP threads will communicate through memory locations; again, scoping rules apply: (1) any variable whose scope is external to a parallel section will be shared among all threads, and (2) all threads will have their own private copy of any variable declared within the scope of a parallel section. So, we will define an integer for each thread to keep track of the number of triples summing to 0 that it sees external to the parallel section, so we can add those counts together at the end (lines 54-55). Recall in the Pthread code, we had to explicitly set thread identifiers to 0 and 1; OpenMP will do this for us and each thread can find its ID using the `omp_get_thread_num()` function (line 62):

```cpp
#include <iostream>
#include <sstream>
#include <omp.h>

using namespace std;

int getRandInt( int *dataVec, int dataSize )
{
    // load up a vector with random integers
    int num;
    for( int i = 0; i < dataSize; i++ ) {
        // integers will be 1-100
        num = rand() % 100 +1;
        if( rand( ) % 2 == 0 ) {
            // make some integers negative
            num *= -1;
        }
        dataVec[i] = num;
    }

    int main(int argc, char * argv[] )
    {
        int dataSize = 0;

        if( argc < 2 ) {
            std::cerr << "usage: exe [num of nums] [optional seed value]" << std::endl;
            exit( -1 );
        }

        std::stringstream ss1;
        ss1 << argv[1];
        ss1 >> dataSize;

        if( argc >= 3 ) {
            std::stringstream ss1;
            ss1 << argv[1];
            ss1 >> dataSize;
            srand( seed );
        }
        else {
            srand( 0 );
        }

        // create a data vector
```
```c
int *data = new int[ dataSize ];

// load it up with random data
getRandInt( data, dataSize );

// do the naive Sum3 computation. O(n^3)
int count1 = 0;
int count2 = 0;

omp_set_num_threads( 2 );
#pragma omp parallel
{
    int count = 0;
    int myID = omp_get_thread_num();
    // each thread only works on half the array in the outer loop
    // compute the bounds based on the ID we assigned each thread.
    // remember, we only have 2 threads in this case, so we will hard code a 2
    int start = (( dataSize / 2) * myID );
    int stop = ((dataSize / 2)* (myID+1));
    if( myID == 1 )
        stop =dataSize-2;
    for (int i = start; i < stop; i++)
        for (int j = i+1; j < dataSize-1; j++)
            for (int k = j+1; k < dataSize; k++)
                if (data[i] + data[j] + data[k] == 0){
                    count++;
                }
    if( myID == 0 )
        count1 = count;
    else
        count2 = count;
} cout<< count1 + count2 <<endl;
```

Now, some running times. Again, speedup is roughly 2 for the OpenMP version, but the OpenMP version required much less code reorganization to implement. The running times are a few thousandths of a second higher than the Pthread versions. This is due to a different implementation of threading. In OpenMP, the compiler implements the threading, rather than OS system calls. Because of this, the threads get compiled a little differently. Also, different OpenMP compilers will have slightly different performance results.

<table>
<thead>
<tr>
<th>Array Size</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>.005</td>
</tr>
<tr>
<td>200</td>
<td>.014</td>
</tr>
<tr>
<td>300</td>
<td>.032</td>
</tr>
<tr>
<td>400</td>
<td>.066</td>
</tr>
<tr>
<td>500</td>
<td>.124</td>
</tr>
<tr>
<td>600</td>
<td>.212</td>
</tr>
<tr>
<td>700</td>
<td>.333</td>
</tr>
<tr>
<td>800</td>
<td>.492</td>
</tr>
<tr>
<td>900</td>
<td>.696</td>
</tr>
<tr>
<td>1000</td>
<td>.952</td>
</tr>
</tbody>
</table>

Exercise

Compile the serial cubic version of the program and the openmp version with compiler optimizations turned on. How do the running times compare then?
Creating a Lot of Threads

Because OpenMP is integrated into the compiler, the overhead of creating threads is small; thus, it is possible to create many more threads than available processor cores. An important concept of OpenMP is the idea of worker threads: on a computer with 2 cores, the number of worker threads defaults to 2. Even if many threads are created, only 2 will be able to run at any given time. Therefore, it is acceptable, although not always optimal, to create many more threads than there are processors available. One easy way to create a lot of threads is through the `omp parallel for` pragma. This pragma must be placed directly before a `for` loop. OpenMP will then generate a single thread for every iteration of that for loop. To be safe, make sure that the `for` loop contains the initialization, increment, and stopping condition in the loop declaration!

One problem with generating lots of threads is that we also need to then generate lots of `count` variables (lines 54-55 above). One alternative is to create a single global count variable, but then make sure that only 1 thread accesses it at a time. To ensure that only 1 thread accesses a variable at a time, we put that variable access in a critical section using an OpenMP `omp_critical` pragma. Anything in a code block directly following an `omp_critical` pragma is guaranteed to be accessed by exactly 1 thread at a time. Using these directives, our multi-threaded code looks very similar to our original code:
```cpp
int seed;
ss1 << argv[2];
ss1 >> seed;
srand( seed );
}
else {
    srand( 0 );
}

// create a data vector
int * data = new int[ dataSize ];

// load it up with random data
getRandInt( data, dataSize );

int count = 0;
// do the naive Sum3 computation. O(n^3)
#pragma omp parallel for
for (int i = 0; i < dataSize-2; i++)
    for (int j = i+1; j < dataSize-1; j++)
        for (int k = j+1; k < dataSize; k++)
            if (data[i] + data[j] + data[k] == 0) {
                //pragma omp critical
                count++;
            }

cout<< count <<endl;
```

The one drawback of the critical section is that execution may serialize on it. Therefore, if every thread enters the critical section in every execution of the loop, then the program will behave much like a serial program in terms of running time. Therefore, placement of the critical section is important! Note that we placed it inside the if statement. Every triple must be tested to see if it sums to 0, but only a small portion of the triples actually sum to 0. If we placed the if statement in the critical section, we would get serial behavior. If every triple summed to 0 (an array of lots of 0s), we would also get serial behavior (regardless of the placement of the critical section with respect to the if statement). Thus, the previous method will have more reliable speedups in these edge cases, but this method will get speedups for input that we are likely to see, and requires only a few lines of modification to the code, and no logic changes!

Here are the running times. Note that the critical section does have an impact, but it is not too bad. A speedup of 1.9 vs 1.9 for the Pthreads version at an array size of 500 (the same!), and a speedup of 1.9 vs 2.1 for the Pthreads version at an array size of 1000:

<table>
<thead>
<tr>
<th>Array Size</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>.005</td>
</tr>
<tr>
<td>200</td>
<td>.014</td>
</tr>
<tr>
<td>300</td>
<td>.032</td>
</tr>
<tr>
<td>400</td>
<td>.068</td>
</tr>
<tr>
<td>500</td>
<td>.130</td>
</tr>
<tr>
<td>600</td>
<td>.218</td>
</tr>
<tr>
<td>700</td>
<td>.343</td>
</tr>
<tr>
<td>800</td>
<td>.512</td>
</tr>
<tr>
<td>900</td>
<td>.722</td>
</tr>
<tr>
<td>1000</td>
<td>.989</td>
</tr>
</tbody>
</table>

## Chapter 5. Review of Multithreaded Programming
Exercise

Change the OpenMP program using a critical section to serialize in 2 ways. First, use an input array of all 0’s. Then move the critical section to contain the if statement. Compare running times to the serial version of the algorithm (the cubic one), and give a list of reasons as to why one is faster/slower than the other.

Reductions

OpenMP has a lot of other options, constructs, and functionality. Check the official documentation for more details. One of the advantages of OpenMP is that many of these constructs make it very easy to achieve tasks that one has to do manually with pthreads. A good example of this is the reduction.

A reduction simply means to combine many values into a single value using a specified operator. For example, we can reduce an array of numbers to a single sum of all the numbers in the array (by adding them up). Alternatively, we could reduce an array to its max or min value, by searching for the largest or smallest value, respectively, in the array. We can use a reduction to completely get rid of the critical section in the previous version of the code.

When declaring an omp parallel for pragma, we can identify values that will be reduced after all threads have executed. Each thread will then make its own local copy of any variable specified in a reduction, and OpenMP will automatically perform the specified reduction of all of those local variables into a single value. The reduction operation is also specified.

For example, to get rid of the critical section above, we should tell OpenMP to perform a reduction on count. Thus, every thread will get its own copy of count. We will specify a sum reduction, so that at the end of thread execution, all the local versions of the count variable get summed into a single variable. The result is that we only have to add a single line of code to achieve parallel implementation of Sum3. Line 55 specifies that a sum reduction on the variable count will be computed:

```c++
#include <iostream>
#include <sstream>
#include <omp.h>

using namespace std;

int getRandInt( int *dataVec, int dataSize )
{
    // load up a vector with random integers
    int num;
    for( int i = 0; i < dataSize; i++ ) {
        // integers will be 1-100
        num = rand() % 100 +1;
        if( rand() % 2 == 0 ) {
            // make some integers negative
            num *= -1;
        }
        dataVec[i] = num;
    }

    int main(int argc, char * argv[] )
    {
        int dataSize = 0;
        if( argc < 2 ) {
            std::cerr << "usage: exe [num of nums] [optional seed value]" << std::endl;
            exit( -1 );
        }
```

5.3. Sum3 Parallel Implementation with Pthreads and OpenMP
```cpp
std::stringstream ss1;
s1 << argv[1];
s1 >> dataSz;
}

if ( argc >= 3 ) {
    std::stringstream ss1;
    int seed;
    ss1 << argv[2];
    ss1 >> seed;
    srand( seed );
} else {
    srand( 0 );
}

// create a data vector
int * data = new int[ dataSz ];

// load it up with random data
getRandInt( data, dataSz );

int count = 0;
// do the naive Sum3 computation. \( O(n^3) \)
#pragma omp parallel for reduction(+:count)
for (int i = 0; i < dataSz-2; i++)
    for (int j = i+1; j < dataSz-1; j++)
        for (int k = j+1; k < dataSz; k++)
            if (data[i] + data[j] + data[k] == 0){
                count++;
            }
    cout<< count <<endl;
}
```

And finally, here are some running times:

<table>
<thead>
<tr>
<th>Array Size</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>.005</td>
</tr>
<tr>
<td>200</td>
<td>.013</td>
</tr>
<tr>
<td>300</td>
<td>.031</td>
</tr>
<tr>
<td>400</td>
<td>.068</td>
</tr>
<tr>
<td>500</td>
<td>.125</td>
</tr>
<tr>
<td>600</td>
<td>.213</td>
</tr>
<tr>
<td>700</td>
<td>.332</td>
</tr>
<tr>
<td>800</td>
<td>.505</td>
</tr>
<tr>
<td>900</td>
<td>.697</td>
</tr>
<tr>
<td>1000</td>
<td>.963</td>
</tr>
</tbody>
</table>

The running times are a bit slower than the Pthreads version, and a bit faster than the OpenMP version with a critical section. However, remember that the Pthreads version is hard coded to 2 threads, and the programmer essentially must implement their own reduction if you want to use more threads. This version required adding 1 pragma to an otherwise unchanged serial implementation and achieved threading that will automatically scale to the number of available processors with NO extra work! This code is easy to debug (you simply comment out the pragma and debug in serial), uses advanced features, and is easily adaptable to multiple hardware configurations. This should convince you of why OpenMP is a popular choice for threading.
Exercise

Look at the scheduling clauses for OpenMP (the wiki has a concise description). Try dynamic scheduling with various chunk sizes on the above program. Report the effects on execution time, and describe why those effects are occurring.

Convert the $O(n^2)$ version of the algorithm to a parallel algorithm using OpenMP. Try to get the fastest time for 100,000 numbers. Make sure to use a computer with multiple cores (and hyperthreading turned off!)
Our discussions on trees centered around a data structure that stored items efficiently, but to get the balanced height trees, things got tough to implement. Instead of focusing so much on the structure, hashing takes the approach that the structure should be rather simple. The downside is that iterating over elements does not come for free, as in trees, but is possible with a few tricks.

So, for hashing we are looking at simple structures, usually arrays. We will manage the size of the array to be not too much bigger than the amount of data stored (like C++ vectors), to preserve iteration in linear time.

### 6.1 Hash Tables

The basic question is “Why not just use an array as a table?”. It’s a good question...

Let’s think about a table containing products that a store wants to keep track of. Here is an example. There are serious problems with this basic approach, what are they?

<table>
<thead>
<tr>
<th>prodID</th>
<th>desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>camera</td>
</tr>
<tr>
<td>2</td>
<td>tabletPC</td>
</tr>
<tr>
<td>3</td>
<td>DVDburner</td>
</tr>
<tr>
<td>4</td>
<td>scanner</td>
</tr>
</tbody>
</table>

#### Real World Data

Here are some examples of real world data we may want to store in a table. If we are simply using the numbers as array indexes, then:

1. How big of an array do we need?
2. How much of the array will actually get used?
   - Students: Student ID (9 digits)
   - People: SSN (9 digits)
   - ZIP code: 5 digits, 9 digits
6.1.1 Basic Hash Tables

A Hash Table will consist of 2 parts:

1. a table (an array), and
2. a hash function that will convert key values to array indices. (used for insert/delete/search)

A hash function can really be anything, but there are some recipes for reliably good ones. Here are a couple examples of some that might work out in specific cases:

- Use certain digits form a long number. ex: last 4 digits of student ID. Will this work at our university?
- Folding. Use some function to get a smaller range of values. ex: add the digits of student ID. Will this work at our university?

A Basic Hash Table Example

size: 5

Hash function: Add first and last digits, then mod the result by the table size.

Here is the table:

<table>
<thead>
<tr>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Insert the following:

1. 349587 → 10%5 = 0
2. 98745 → 14%5 = 4
3. 84743 → 11%5 = 1

Now find the same numbers in the hash table. (just apply the formula and look for them)

Now insert:

- 24544 → 7%5 = 1. Collision with 84743!
Collisions are a problem, but there are various ways to handle them:

Open addressing collision handling methods:
1. Linear probing – look for next open spot
2. Quadratic probing
3. Double Hashing
4. Increase the table size

Multiple-Item Storage collision handling methods
1. Buckets
2. Chaining

### 6.1.2 Open Addressing: Linear Probing

If there is a collision, just look for the next open slot and insert the item there.

| Insert 84743 | 84743 → 11 % 5 = 1 | 43912 → 24545 | 24544 → 6 % 5 = 1 | 43912 → 43912 |
| Insert 24544 | 43912 | 24545 | 43912 | 43912 |

**Deleting.** Deleting is problematic, since removing an item might break the linear probe. Instead of actually deleting items, mark them as being deleted (lazy delete!)

What happens when the hash table fills up?

One problem with linear probing is that it can lead to a degenerate situation where items that map to the same portion of a hash table overflow into other parts of the hash table, causing a cascading series of probes for lots of items.

### 6.1.3 Open Addressing: Quadratic Probing

Instead of just looking at the next slot for an opening, follow a quadratic sequence of indices (1,2,4,8,16,...)
Insertion fails – with openings!

Resolves some of the clustering problems of linear probing. Can fail with a non-full hash table (but we can make an odd sized table).

What do we do when insertion fails?

6.1.4 Open Addressing: Double Hashing (or triple, or quadruple for that matter)

If a collision occurs on one hash function, simply use another one.

What do we do if a collision occurs on all hash functions?

The hash functions can be tried in parallel! but hash functions are typically pretty computationally cheap.

6.1.5 Open Addressing: Increase the Table Size

Make a new array with more room. How much more room?

Insert each item into the new array. Do we reuse the same hash functions?

Delete the old array.

When should this occur? When insertion fails, when the table is full? when the table is almost full? when a certain percent of probes have occurred?

6.1.6 Multi-item Storage: Buckets

The idea is simple, just keep room for more than 1 item at each table location. Pick a fixed number, and possibly use an additional open addressing strategy if the bucket fills.
6.1.7 Multi-item Storage: Chaining

Use a linked list (or other ADT) at each table location.

Might need to consider increasing the table size if a list (or lists) get too long.

6.1.8 String Hash Functions

String hash functions are little tougher. Here are some examples:

- Add the numeric values of first five characters:
  1. burner = 2 + 21 + 18 + 14 + 5 = 60
  2. scanner = 19 + 1 + 14 + 14 + 5 = 53
  3. camera = 3 + 1 + 13 + 5 + 18 = 40
  4. tablet = 20 + 1 + 2 + 12 + 5 = 40

Values range from 1 to 130

- Concatenate positional values of first five characters
  1. burner = 2 21 18 14 5 = 22,118,145
  2. scanner = 19 3 1 14 14 5 = 19,314,145
  3. camera = 3 1 13 5 18 = 3,113,518
  4. tablet = 20 1 2 12 5 = 2,012,125

Values range from 1,048,576 to 28,142,426

Instead, look into production hash functions: MD5, SHA, etc.

6.1.9 Good Hash Functions

Good hash functions are easy to compute, and distribute values evenly throughout the table.

Here is a recipe:

\[ h(k) = (f(a, k) + b) \% p \% S \]

Where:

- \( S \) is the size of the table.
- \( p \) is a prime number larger than any number that will be hashed (4,294,967,291 is the largest unsigned 32 bit prime integer. 2,147,483,647 is the largest signed 32 bit prime number. 18,446,743,684,679,551,577 is the largest unsigned 64 bit number. 9,223,372,036,854,775,808 is the largest signed prime integer. NOTE you MUST BE CAREFUL with integer overflows, when performing these calculations!!!)
- \( a, b \) are positive constants both less than \( p \). Make sure \( a \neq 0 \).
- \( f(a, k) \) is some interaction of \( a \) and \( k \). A good basic choice is \( f(a, k) = a \times k \)

Multiple hash functions can be easily generated by choosing random values of \( a \) and \( b \).

One implementation note, intermediate values need to be 64 bit to prevent overflow!

This type of hash function falls in the family of 2-universal family hash functions, with a probability of items colliding \( \leq \frac{1}{2} \). (just make sure \( k < p \) for all \( k \))

Let's try this out in class.
6.1.10 Load Factor of a Hash Table

The load factor for a hash table is: \( \frac{num\text{InsertedItems}}{num\text{Locations}} \). This is between 1 and 0. A high load factor indicates the hash table is almost full, and you might want to think about resizing it.

6.1.11 A final note

Make sure the hash table size is ODD. Prime numbers help too.

No matter what, a hash table doesn’t store data in sorted order!

6.2 Bloom Filters

Bloom filters are one of the coolest data structures around, but they are not used too often.

The idea behind a bloom filter is that we don’t always need to store every item, sometimes we just need to record the fact that the item exists. This is known as membership testing.

**Question**

What are some example applications that might use membership testing?

Bloom filters don’t store the actual data, they keep track of a bit string. Also, they do not use a single hash function, but use a group of \( k \) hash functions. Each hash function will address a single bit in the array.

**Example**

Hash\_function1( int ): mod the sum of the first two digits

Hash\_function2( int ): mod sum of 2nd pair of digits

Hash\_function3( int ): mod sum of 3rd pair of digits

Now insert: 937789, 932243, 106616

Does this number exist in our set? 134898
Not so fast! The thing about bloom filters is that they are not always right, but they are never wrong. Bloom filters have the property that they may have **False Positives**, a number is recorded in the filter but was never explicitly put into the filter! This occurs by chance, when other numbers simply cause those bits to be 1. It can also happen when the bloom filter begins to fill up!

So what do we do about false positives? **CONTROL THEM!!**

We just make the probability of false positives sufficiently small.


Assume that a hash function selects each array position with equal probability. If $m$ is the number of bits in the array, the probability that a certain bit is not set to one by a certain hash function during the insertion of an element is then

$$1 - \frac{1}{m}.$$ 

The probability that it is not set by any of the hash functions is

$$\left(1 - \frac{1}{m}\right)^k.$$ 

If we have inserted $n$ elements, the probability that a certain bit is still 0 is

$$\left(1 - \frac{1}{m}\right)^{kn};$$ 

the probability that it is 1 is therefore

$$1 - \left(1 - \frac{1}{m}\right)^{kn}.$$ 

This means that the probability of a false positive is roughly the probability of $k$ bits being 1

After inserting $n$ elements that each set $k$ bits:

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

Assumes independence of hash functions
From: 
\[
\left( 1 - \left[ 1 - \frac{1}{m} \right]^{kn} \right)^k \approx \left( 1 - e^{-kn/m} \right)^k,
\]

We can derive the optimal value of \( k \) for a given \( m \) and \( n \):
\[
\frac{m}{n} \ln 2 \approx 0.7 \frac{m}{n},
\]

Which gives a false positive probability of:
\[
2^{-k} \approx 0.6185^{\frac{m}{n}}.
\]

- We can determine \( n \), the number of items we need to store
- Then choose an appropriate \( m \), compute the optimal \( k \), and see if the false positive probability is within acceptable bounds
- If not, try new values for \( m \) or \( n \)

Or just look at the table:

<table>
<thead>
<tr>
<th>( m/n )</th>
<th>( k=1 )</th>
<th>( k=2 )</th>
<th>( k=3 )</th>
<th>( k=4 )</th>
<th>( k=5 )</th>
<th>( k=6 )</th>
<th>( k=7 )</th>
<th>( k=8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.39</td>
<td>0.393</td>
<td>0.400</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.08</td>
<td>0.283</td>
<td>0.237</td>
<td>0.253</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2.77</td>
<td>0.221</td>
<td>0.155</td>
<td>0.147</td>
<td>0.160</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3.46</td>
<td>0.181</td>
<td>0.109</td>
<td>0.092</td>
<td>0.092</td>
<td>0.101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4.16</td>
<td>0.154</td>
<td>0.0804</td>
<td>0.0609</td>
<td>0.0561</td>
<td>0.0578</td>
<td>0.0638</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4.85</td>
<td>0.133</td>
<td>0.0618</td>
<td>0.0423</td>
<td>0.0359</td>
<td>0.0347</td>
<td>0.0364</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5.55</td>
<td>0.118</td>
<td>0.0489</td>
<td>0.0306</td>
<td>0.024</td>
<td>0.0217</td>
<td>0.0216</td>
<td>0.0229</td>
</tr>
<tr>
<td>9</td>
<td>6.24</td>
<td>0.105</td>
<td>0.0397</td>
<td>0.0228</td>
<td>0.0166</td>
<td>0.0141</td>
<td>0.0133</td>
<td>0.0135</td>
</tr>
<tr>
<td>10</td>
<td>6.93</td>
<td>0.0952</td>
<td>0.0329</td>
<td>0.0174</td>
<td>0.0118</td>
<td>0.00943</td>
<td>0.00844</td>
<td>0.00819</td>
</tr>
<tr>
<td>11</td>
<td>7.62</td>
<td>0.0869</td>
<td>0.0276</td>
<td>0.0136</td>
<td>0.00864</td>
<td>0.0065</td>
<td>0.00552</td>
<td>0.00513</td>
</tr>
<tr>
<td>12</td>
<td>8.32</td>
<td>0.08</td>
<td>0.0236</td>
<td>0.0108</td>
<td>0.00646</td>
<td>0.00459</td>
<td>0.00371</td>
<td>0.00329</td>
</tr>
<tr>
<td>13</td>
<td>9.01</td>
<td>0.074</td>
<td>0.0203</td>
<td>0.00875</td>
<td>0.00492</td>
<td>0.00332</td>
<td>0.00255</td>
<td>0.00217</td>
</tr>
<tr>
<td>14</td>
<td>9.71</td>
<td>0.0689</td>
<td>0.0177</td>
<td>0.00718</td>
<td>0.00381</td>
<td>0.00244</td>
<td>0.00179</td>
<td>0.00146</td>
</tr>
<tr>
<td>15</td>
<td>10.41</td>
<td>0.0645</td>
<td>0.0156</td>
<td>0.00596</td>
<td>0.003</td>
<td>0.00183</td>
<td>0.00128</td>
<td>0.001</td>
</tr>
</tbody>
</table>

**Question**

What are some example applications that might use membership testing now that we have to deal with false positives?
Lets say I want to do existence testing for a set of 1,000,000 items.

- If an item is an integer, that means we would need to store 4,000,000 bytes $\approx 3.8$ MB of data
- We would need roughly twice that to prevent saturation in a hash table $\approx 7.6$MB
- What we are storing may be much larger than an integer!

In the case of the Bloom Filter...

- Lets say we want a false positive rate of 0.001
- According to the table, we need $m/n = 15$ and $k = 7$
- So, we need 15,000,000 bits $\approx 1.8$MB. That's a quarter of the storage of the hash table, and, no matter how large the items are, it will always take 1.8MB
- We just need to make sure that we can come up with 7 independent hash functions! But that's easy with our hash function recipe above!

Finally, some properties of Bloom Filters:

- A BF can represent an entire universe of elements, whereas hash table runs out of space. Also, a hash table must explicitly store all elements
- Union and Intersection of bloom filters is possible using bitwise operations on the bit strings (super useful for combining tables in databases without having to rebuild indexes)
- If the filter becomes saturated, we have to change the $m/n$ ratio. There are types of BFs that grow like vectors!
- there are tons of variants for all types of applications: Bloomier filters, counting filters, stable bloom filters, scalable filters
- Deletion is a problem! What happens if we try to delete something?

Lets practice a little...
CHAPTER
SEVEN

HEAPS AND PRIORITY QUEUES

Our major container structures that we have seen so far are:

- Trees
- Hash Tables

Height balanced trees provide logarithmic operations and ordering of elements, but are limited to logarithmic time for operations. Hash tables provide constant insertion and lookup time, but do not provide ordering of elements. So, each structure introduces tradeoffs between features, requirements of the data being stored in the structure, and running time. Heaps offer new alternatives to these tradeoffs.

7.1 Priority Queues

So, we have a bunch of items that we want to store in and remove from a data structure, BUT we want to get the highest priority (or lowest priority) items out first, regardless of the order in which the items were inserted. We need:

1. A key indicating an item’s priority
2. A data structure that arranges items by this key

Some obvious approaches:

1. A linked list. Insertion in $O(1)$ and removal in $O(n)$ (or reversed)
2. A binary tree. Should we consider balanced trees, splay trees?

NOTE

Trees provide a LOT of operations, but a priority queue, much like a queue, does not need that many operations. We just need:

- enqueue
- dequeue

The only difference is that dequeue should return the highest priority item in the structure

A priority queue has only 2 required operations/goals!

1. Remove the minimum (or highest priority element) quickly
2. support quick insertions

This is why trees are overkill. We don’t need a total ordering of elements and a bunch of operations. Instead, we can get away with a PARTIAL ORDERING elements since we don’t need to traverse the items in order. Its OK if elements themselves have a total ordering, but the priority queue will only represent a partial ordering of the items.
Applications of priority queues:
Anything with Quality Of Service (QOS) levels.

- What about multi-QOS internet protocols at routers?

Heap sort.

Also just a useful algorithmic tool (to verify time complexities).

A Partial Order is: (from wiki)

A partial order is a binary relation \( \leq \) over a set \( P \) which is reflexive, antisymmetric, and transitive, i.e., for all \( a, b, \) and \( c \) \in \( P \), we have that: * \( a \leq a \) (reflexivity); * if \( a \leq b \) and \( b \leq a \) then \( a = b \) (antisymmetry); * if \( a \leq b \) and \( b \leq c \) then \( a \leq c \) (transitivity). * For \( a, b \), elements of a partially ordered set \( P \), if \( a \leq b \) or \( b \leq a \), then \( a \) and \( b \) are comparable. Otherwise they are incomparable. A partial order under which every pair of elements is comparable is called a Total order.

If \( X \) is totally ordered under \( \leq \), then the following statements hold for all \( a, b, \) and \( c \) \in \( X \): * If \( a \leq b \) and \( b \leq a \) then \( a = b \) (antisymmetry); * If \( a \leq b \) and \( b \leq c \) then \( a \leq c \) (transitivity); * \( a \leq b \) or \( b \leq a \) (totality).

Graphically, partial orders look like a lattice:

- Partial ordering of the set of all subsets
  - \( \{x\} \leq \{x\} \) (reflexive)
  - \( \{x\} \leq \{x, y\}, \{x, y\} \leq \{x, y, z\}, \{x\} \leq \{x, y, z\} \) (transitive)
  - \( \{x\} \leq \{x\} \) and \( \{x\} \leq \{x\} \) so \( \{x\} \leq \{x\} \) (antisymmetric)
  - \( \{x\} \leq \{y\} \) == false and \( \{y\} \leq \{x\} \) == false
    - \( \{x\} \) and \( \{y\} \) are incomparable, which is why the set of subsets is partially ordered
    - In a total ordering, every pair of elements in a set is comparable (totality)

Partial orders come up rather frequently, it is good to remember how to deal with them.

Usually pop up when non-intuitive notions of Ordering arise

Is one image < another?
Is one song < another?

7.1.1 Binary Heaps

A binary heap is a natural fit for a priority queue. It enforces a partial order over elements, leading to fast enqueue, dequeue, and build times.

Definition

A binary heap is a binary tree with a few properties:
1. The heap is **complete binary tree**. So, we can use the array implementation of a tree.

2. The heap order property is maintained. This property is stated such that for every node $X$, the key of $X$ is less than or equal to the key of $X$’s parent (except the root which has no parent).

Note the for property 2, there is NO ordering among siblings of a parent. This is the partial ordering part.

When inserting deleting, we must maintain the structural property of having a complete tree, but we don’t need rotations! rotations are a consequence of total ordering.

Instead, structural operations only involve swapping the items between a parent and child!

- Moving an item up or down along a path to a root is called a **percolation**

**INSERT Operation:** Add the new node to the end of the tree (maintaining the complete tree property), then percolate up:

```
13 21 16 24 31 19 68 65 26 32
```

**DELETE highest priority item Operation (deleteMin):** 1. Find the min element. It is always at the root! 2. Swap the item in the root with the item in the last node of the tree. 3. Delete the last node of the tree. (Its always...
Excercise
What is the time complexity of heap insert and heap deleteMin?

Other operations on heaps
Heaps are great at removing the highest priority item and inserting new items, but not other operations. For example:

1. findMax – must traverse the entire tree (because of partial ordering, we don’t know if it will be the leftmost leaf, the rightmost leaf, or something on between)
2. decreaseKey – we must traverse the heap to find the item, before decreasing the key
3. increaseKey – same as decreaseKey
4. Remove arbitrary element – use decreaseKey to make the item the smallest in the heap, then just use deleteMin

If we can locate an element quickly, we can speed up some of the operations. Perhaps a hash table that:

HashTable: item -> heap index
OR
HashTable: item -> element
OR
HashTable: item -> node pointer

So far, the heap hasn’t given us any time complexity advantages. This is where building a heap shines!

We can build a heap by continually inserting items, leading to $O(n \log n)$ time.

Or, we can just throw all the items into an array, then just percolate down EVERY ITEM beginning from the END of the array. This leads to $O(n)$ time:
We can guarantee $O(n)$ if we visit each node and percolate down instead.

- Compute the sum of all subtree heights that we must percolate down:
  \[
  s = \sum_{i=0}^{h} 2^i \cdot (h-i)
  \]
- (eq1) $s = h + 2(h-1) + 4(h-2) + 8(h-3) + 16(h-4) + \ldots + 2^{h-1}$
- Multiply by 2
- (eq2) $2s = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \ldots + 2^h$
- Subtract eq2 – eq1
  \[
  2s - s = -h + 2h - 2(h-1) + 4(h-1) - 4(h-2) + 8(h-2) - 8(h-3) + \ldots + 2^{h-2}(h-1)
  \]
- Notice that most terms cancel:
  \[
  4(h-1) - 4(h-2) = 4(h-1-(h-2)) = 4(h-1-h+2) = 4(1) = 4
  \]
- Thus $s = -h + 2 + 4 + 8 + \ldots + 2^{h-1} + 2^h = (2^{h+1} - 1) - (h+1) = O(n-1 - \lg n) = O(n)$

- Since a complete tree has between $2^h$ and $2^{h+1}$ nodes
- The sum is $O(n)$

---

**Example: Building a Heap**

Start with a broken Heap, then fix it

By repeatedly percolating Down. Begin at the bottom
7.1.2 Double Ended Priority Queues (DEPQs)

Sometimes, it is not enough to be able to get just the highest priority element. Occasionally, we want to be able to be
able to get rid of some the lowest priority elements if, for example, the heap is getting too large. Normal heaps don’t
provide a nice way to do this. Double ended priority queues offer a solution.

One Approach:

- use 2 normal priority queues.
- One is a min heap, the other is a max heap
- each element goes into both
- keep pointers between identical elements

More than double storage
More than double operation cost

 Clearly, we are duplicating space.

A better approach: a correspondence structure:

- use a min and max heap, but each only stores \( n/2 \) nodes
- if \( n \) is odd, keep 1 node in a buffer
- keep a total correspondence between nodes in each heap (every node in a heap is associated with exactly 1
node in the other heap)
- We must be able to support a remove operation in the heaps

Definition: Total Correspondence: Each element in the min priority queue is pairs with (has a pointer to) an element
in the max priority queue that is >= to it

Example:
Min Heap

Max Heap

Buffer = 12

Insert:
1. If buffer is empty, place in buffer
2. Otherwise, insert smaller of new element and buffer into min queue, and larger into max queue
3. Establish the correspondence between the newly inserted elements

Insert: 12 15 1 3 18 4 7 8 19

RemoveMin:
1. If buffer is the min, remove the buffer
Otherwise
2. Remove min from min heap
3. Remove corresponding element from max heap
4. Reinsert element from max heap (the element and the buffer form a pair)
7.2 Advanced Heaps

These heaps use some fancy tricks to improve the time complexity of some operations.

7.2.1 Leftist Heaps

Up until now, we have spent a lot of time trying to make sure trees are balanced. **Leftist heaps** take the opposite approach:

- Make the tree predictably unbalanced
  - If the tree is unbalanced towards the left, then...
  - The tree is deep on the left parts, **but** the tree is shallow on the right parts

So we can focus our operations on the right parts!

Furthermore, findMin in a (min) heap does not need to search down to a leaf! So having a deep part of the tree does not affect performance much

**STRUCTURE:** Leftist heaps are binary trees, but not necessarily complete trees, so we can’t just use the array implementation of a binary tree (stuck with nodes)
**OPERATIONS:** Leftist heaps can do everything a normal heap can do. But, we can meld two leftist heaps in $O(\log n)$ time. For normal heaps, we can append the arrays and re-build the heap in $O(n)$ time, so leftist heaps have a big advantage there.

**Preliminaries:**

Let’s treat a binary tree as an extended binary tree. Basically, just draw the NULL pointers, and call them **external nodes**.

![Binary Tree Diagram]

**Definition: the $s()$ function:**

The $s()$ function defines the **NULL PATH LENGTH**

For any node $x$ in an extended binary tree, let $s(x)$ be the length of a shortest path from $x$ to an external node in the subtree rooted at $x$.

1. If $x$ is an external node, then $s(x) = 0$

   Otherwise

2. $s(x) = \min ( s(\text{leftChild}(x)), s(\text{rightChild}(x)) + 1 )$

![Second Binary Tree Diagram]

**Definition:** A Height Biased Leftist Tree:

A binary tree is a (height biased) leftist tree iff for every internal node $x$:

$s(\text{leftChild}(x)) \geq s(\text{rightChild}(x))$
Length of rightmost path is 2.

Properties of Leftist Trees:

1. In a leftist tree, the rightmost path is a shortest root to external node path and the length of this path is $s(\text{root})$.

2. The number of internal nodes is at least $2^{s(\text{root})} - 1$. (Because levels 1 through $s(\text{root})$ have no external nodes).

3. Length of rightmost path is $O(\log n)$, where $n$ is the number of (internal) nodes in a leftist tree.

So… the rightmost path has a guaranteed bound, and is the shortest path through the tree…

Make sure all operations are performed on the rightmost path!!

Melding leftist trees

The major operation is to meld. In fact, insert, remove mi, and build heap will ALL be implemented with a meld operation!

Always meld with the right (shortest) subtree!

Merge Algorithm:

1. Take right subheap from root of heap with smallest root
2. Merge that with other heap
3. repeat

Once a merge with an empty tree takes place, Walk back up the tree

4. Swap child nodes if leftist property does not hold
Meld right subtree of tree with smaller root and all of other tree.

Meld right subtree of tree with smaller root and all of other tree.
Now meld the result of that with the smaller of that is left from the rightmost branch of the original trees.

Finally, we only have 1 choice, so attach it, then walk back up the rightmost path, switching children pointers if the leftist property is violated.
Walk back up the tree
Swapping child pointers
Where the leftist property is violated

Meld takes $O(\lg n)$, since the rightmost branch is logarithmically bounded in height. However, because the rightmost branch can be arbitrarily short, it often is much faster!

**Insert Operation**
the new item is a singleton leftist tree. Meld it with the existing tree

**Remove Min Operation**
remove the root node. The roots children are two leftist trees. Meld them together.

**Build a leftist Heap:**

1. Create $n$ single-node min leftist trees and place them in a FIFO queue.
2. Repeatedly remove two min leftist trees from the FIFO queue, meld them, and put the resulting min leftist tree into the FIFO queue.
3. The process terminates when only 1 min leftist tree remains in the FIFO queue.
   - Analysis is the same as for heap initialization
   - Remember a binary heap is a complete tree and satisfies the leftist property

**7.2.2 Skew Heaps**

Very similar to leftist tree, except that no $s()$ values are stored on the nodes.

When melding, instead of swapping left and right subtrees just when $s(l(x)) < s(r(x))$, ALWAYS swap them!

- Amortized complexity of each operation is $O(\log n)$
- Don’t need to store $s()$ values, so no extra information is needed in a node

7.2. Advanced Heaps 111
• Easy to implement, just need a merge (meld) operation
• (relationship to leftist heaps is similar of the relationship of splay trees to AVL trees)

Exercise:
Perform a meld on the trees used in the leftist tree meld example, but treat them as skew heaps.

7.2.3 Binomial Queues
8.1 Graph Definitions

A graph $G$ is a tuple containing two sets: a set of vertices $V$ and a set of edges $E$. Each edge is simply a tuple containing two vertices:

$$G = (V, E)$$

where:

- $V$ is a set of vertices
- $E$ is a set of edges

$$e = (v, w) \in E \mid v, w \in V$$

Because the edges are only defined as pairs of vertices, no concept of embedding space is implied by the definition of a graph. In fact, no concept of the location of a vertex is implied. For example, a vertex can be a point in 2D space, a number, a string, or anything else.

If a graph is a directed graph, then direction of an edge is implied by the ordering of the edge tuple (the source vertex is first, the destination is second). Otherwise, the graph is undirected and an edge implies a path in both directions between the vertices.

$v$ and $w$ are adjacent if $(v, w) \in E$.

An edge may have a weight or a cost. The weight does not necessarily imply a distance between two vertices, but simply provides a value for an edge.

A path is a sequence of vertices $(v_1, v_2, \ldots, v_n)$ such that $\forall 1 \leq i < n : (v_i, v_{i+1}) \in E$.

An edge $(v, v)$ forms a loop.

A simple path contains distinct vertices, although the first and last vertices can be identical.

A cycle has at least 3 vertices and starts/ends on the same vertex. Edges in a cycle must be distinct.

A graph is connected if there is a path to/from all pairs of nodes in the graph.

A directed graph that is connected is strongly connected. It is weakly connected if it is not strongly connected, but its underlying undirected graph is connected.

Examples

Airport systems: usually strongly connected.

- Vertices $=$ airports
- Edges $=$ direct flights
- Edges may have costs: distances, fuel costs, airport fees, likelihood of weather delays, etc.

Traffic flow:

- Edges $=$ roads
• Vertices = intersections
• Costs: road congestion, road capacity, construction, etc.

8.2 Graph Representation

8.2.1 Adjacency Matrix:

**Adjacency Matrix:** A matrix $M$ such that if edge $(u, v) \in E$ then $M[u][v] = True$:

<table>
<thead>
<tr>
<th></th>
<th>V=1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>U=1</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
</tr>
</tbody>
</table>

**Exercise**

Is the above graph strongly connected?

Is the graph weakly connected?

For weighted graphs, a non-connected edge is typically represented by a max or a min cost. The assumption is that the chosen cost will not show up in an actual graph.

Storage of an adjacency matrix is $\Theta(|V|^2)$. It is good for **dense** graphs in which the matrix is nearly or completely full. It is not so good for **sparse** graphs in which the matrix is mostly empty. A sparse matrix means we are allocating space to indicate the absence of an edge.

**Real world graphs are often sparse:**

Manhattan (not counting Broadway): intersections usually have 4 edges emanating.

4 edges pr intersection: $|E| \approx 4|V|$

if there are 3,000 intersections = 12,000 edges stored in 9,000,000 matrix cells. A lot of wasted space.

8.2.2 Adjacency List:

**Adjacency List:** For each vertex, keep a list of all adjacent vertices. Space complexity: $O(|E| + |V|)$. For the example graph given above, the space would be $12 + 7 = 19$ instead of $7^2 = 49$
The Adjacency list for the graph above is:

1: 2, 3, 4
2: 4, 5
3: 6
4: 6, 7, 3
5: 4, 7
6:
7: 6

Concept

For a graph, **linear space complexity** means that the vertices and edges are stored a constant number of times, leading to \( O(|E| + |V|) \). Similarly for **linear computational complexity**, the edges and vertices are visited a constant number of times.

Adjacency lists vs adjacency matrices?

Adjacency lists are better for sparse graphs. For dense graphs, adjacency matrices tend to be better because the overhead of the node structure of linked lists is not present. Recall that a node must contain a pointer to the next node in the list, leading to more memory usage. For complete graphs (every edge is connected to every other edge), an adjacency matrix is preferable, in general.

Adjacency list:

1 : 2, 3
2 : 1, 3
3 : 1, 2

\(|E| + |V| = 3 + 6 = 9 = |V|^2\)
Adjacency list:
1 : 2, 3, 4
2 : 1, 3, 4
3 : 1, 2, 4
4 : 1, 2, 3

\[ |E| + |V| = 4 + 12 = 16 = |V|^2 \]

### 8.3 Topology

In general, graphs can be considered to be embedded in **topological space**, as opposed to **metric space**. In metric space, there is a concept of distance (among other things). For example, we are used to dealing with the metric space \( \mathbb{R}^2 \) (i.e., Euclidean space), in which the concept of Euclidean Distance exists. For example, we can compute the distance between two points to determine how close or far they are apart.

**Topological Space**

In topological space, there is no concept of distance between points. So, points are either **disjoint**, or are **equal**: however, we cannot say if they are close or far because these concepts rely in distance measures. We can tread Euclidean space as a topological space if we choose.

**Topological Relationships** are qualitative relationships among objects in a topological space. Assume the topological space \( \mathbb{R}^2 \). If two lines are embedded in that space, the following are some possible topological relationships.

Intersects:

Disjoint:
Meet:

Topology joke: A doughnut is topologically equivalent to a coffee cup: they are both solid objects that have a single handle. So... a topologist cannot tell the difference between his/her doughnut and coffee cup! This is why there are so few topologists around.

Although graphs are often considered to be embedded in a topological space, many useful graphs are actually embedded in geographic space. For example, road networks definitely have a concept of distance.

Exercise
Think about a road network, for example the blocks in Manhattan. Assume your location is known and you want to find the nearest gas station to your location. The algorithm will be very different if you consider the graph to be embedded in topological space or metric space. Sketch out the algorithm. Which one is faster? Which one returns more reliable results?

8.4 Topological Sort

Definition: A directed acyclic graph (DAG) is a directed graph that contains no cycles.

A topological sort of a DAG results in a list of vertices ordered such that if there is a path from $v_i$ to $v_j$ in the graph, then $v_i$ appears before $v_j$ in the ordering. Note that due to this definition, there is no way to topologically sort a graph with a cycle.
A topological sort is not necessarily a total ordering or a unique order. In fact, it is much more like a lattice ordering. Vertices may exist such that one cannot say that one vertex must come before the other.

**Example: Course sequences.**

One can represent course sequences as a topological sort of a course prerequisite graph. For example, if cs111 and cs140 are prerequisites for cs314, but cs140 and cs111 have no prerequisites, then there are two possible topological sorts of the graph:

- cs111, cs140, cs314
- cs140, cs111, cs314

Both sorts are valid, and correct according to the graph. Specifically, cs140 and cs111 have no ordering implied by the graph that forces you to take one before the other; you can take them in any order. However, both cs111 and cs140 are less than cs314 in terms of the topological sort of the graph.

**Definition:** The in-degree of a vertex is the number of incoming edges to a vertex: \( \text{in-degree}(v) = |\{(u, v)\}| \)

\( G = (V, E) \) \( \land (u, v) \in E \)

The simple algorithm for computing a topological sort is as follows:

1. Compute the in-degree of all vertices.
2. Find a vertex with in-degree of 0. Print it.
3. Decrease the in-degree of all vertices with an incoming edge that comes from the just-printed vertex.
4. Repeat steps 2 and 3 until no more vertices with in-degree = 0 remain.

The simple algorithm has time complexity \( O(|v|^2) \) since we must repeatedly scan all vertices to find a vertex with in-degree of 0. We can improve upon this by storing all vertices with an in-degree of 0 in a queue.

**Topological Sort Algorithm (better)**

```
1: 2, 3, 4
2: 4, 5
3: 6
4: 6, 7, 3
5: 4, 7
6:
7: 6
```

Algorithm:

1. Compute the in-degree of all vertices.
2. All vertices with in-degree=0 go into a queue.
3. While the queue is not empty:
   4. Dequeue a vertex \( v \).
5. ~Decrement edge counts for all neighbors of $$v$$
6. ~If the in-degree of a vertex falls to 0, enqueue it.

For the above the graph, the iterations of the algorithm look like the following:

<table>
<thead>
<tr>
<th>Vertex (v)</th>
<th>In-degree(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>0</td>
</tr>
<tr>
<td>v2</td>
<td>1 0</td>
</tr>
<tr>
<td>v3</td>
<td>2 1 1 1 0</td>
</tr>
<tr>
<td>v4</td>
<td>3 2 1 0</td>
</tr>
<tr>
<td>v5</td>
<td>1 1 0</td>
</tr>
<tr>
<td>v6</td>
<td>3 3 3 3 2 1 0</td>
</tr>
<tr>
<td>v7</td>
<td>2 2 2 1 0</td>
</tr>
</tbody>
</table>

enqueue | v1 v2 v5 v4 v3,v7 | v6 |
queue   | v1 v2 v5 v4 v3,v7 v7 v6 |

dequeue (topological ordering) | v1 v2 v5 v4 v3 v7 v6 |

Time complexity: $$O(|E| + |V|)$$. Scan all edges to create the initial in-degree list (scan the adjacency list). When a vertex is dequeued, use the adjacency list to find all vertices that must have their in-degree decremented (use a hash table, array index, or other constant time lookup structure to store the in-degree values). Each vertex is enqueued and dequeued exactly once.

Remember, topological sort is a graph induced ordering. Metric ordering is an embedded-space-induced ordering on the vertices, but does not necessarily reflect the graph.

**Note:** What does it mean if you get to a point where no vertex has an in-degree of 0?

---

### 8.5 Unweighted Shortest Path

**Definition:** The **unweighted shortest path** problem is stated as: given a source vertex in a graph, find the shortest paths to all vertices.

**Strategy:** Use a **Breadth First Search (BFS)** approach. The basic idea is to find all vertices that 1 edge away from the source, then 2 edges away, then 3 edges away, etc. This is in contrast to depth first search in which we would travel as far as possible along a path until it completes before we begin a new path. For example, a BFS of a binary tree would visit the root, then all nodes at level 1, then all nodes at level2, etc. The depth first search would travel all the way from the root to a leaf, then to another leaf, then to another leaf, etc.

**Algorithm:** Find the neighbors of a source vertex, add 1 to their path length. Get the neighbors of those, and add one to their path length, etc.

**Improvement:** Just like the topological sort, uses a queue and adjacency list to achieve linear time complexity. Simply enqueue the source vertex. Each iteration, dequeue a vertex, add 1 to its path length, and add its neighbors the queue. Effectively, all vertices 1 hop away from the source will show up first in the queue, followed by all vertices 2 hops away from the source, followed by all vertices three hops away, etc.

**Unweighted Shortest Path Example:**
Example, find all shortest paths from v3:

K = known

PV = previous vertex

DV = distance value

<table>
<thead>
<tr>
<th>Vertex</th>
<th>K</th>
<th>PV</th>
<th>DV</th>
<th>K</th>
<th>PV</th>
<th>DV</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>T</td>
<td>v3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v3</td>
<td>T</td>
<td>v3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v4</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v5</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v6</td>
<td>T</td>
<td>v3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v7</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>enqueue</td>
<td>6,1</td>
<td>4,2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue</td>
<td>6,1</td>
<td>4,2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dequeue</td>
<td>6,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that 3,6,7,5 are all known when they are dequeued in the final iterations.

8.6 Dijkstra’s Algorithm

See Notes and Book

8.7 Prim’s Algorithm (MST)

See Notes and Book

8.8 Kruskal’s Algorithm (MST)

See Notes and Book
8.9 Euler Circuits

See Notes and Book
CHAPTER

NINE

INDICES AND TABLES

• genindex
• modindex
• search