Algorithms for addition and multiplication

Let's try to remember your first experience with numbers, way back when you were a child in grade school. In grade 1, you learned how to count up to ten and to do basic arithmetic using your fingers. When doing so, you memorized sums of *single digit* numbers ($4 + 7 = 11$, $3 + 6 = 9$, etc). Later on in about grade 4, you learned a method for adding *multiple digit* numbers, which was based on the single digit additions that you had memorized. For example, you were asked to compute things like:

$$\begin{array}{rr} & 2343 \\ + & \underline{4519} \\ & ? \end{array}$$

The method that you learned was a sequence of computational steps, commonly called an *algorithm*. What was the algorithm ? Let's call the two numbers a and b and let's say they have N digits each. Then the two numbers can be represented as an array of single digit numbers $a[\,]$ and $b[\,]$. We can define a variable *carry* and compute the result in an array $r[\,]$. You know how this works. You go column by column, adding the pair of single digit numbers in that column and adding the carry (0 or 1) from the previous column. We can write the algorithm in *pseudocode*[1] as follows:

Algorithm 1 Addition (base 10): Add two N digit numbers a and b which are represented as arrays of digits

```
carry = 0
for i = 0 to N − 1 do
    r[i] ← (a[i] + b[i] + carry) % 10
    carry ← (a[i] + b[i] + carry)/10
end for
r[N] ← carry
```

The operator % is the "mod" operator. It computes the remainder of the division. The operator / ignores the remainder, i.e. it rounds down (often called the "floor").

Also note that the above algorithm requires that you can compute (or look up in a table that you have "memorized") the sum of two single digit numbers with '+' operator, and also (possibly) add 1 to that result.

Later on in grade school, you learned how to multiply two numbers. Again, you first memorized a multiplication table for single digit numbers (e.g. $6 \times 7 = 42$). You then learned a sequence of steps for multiplying a pair of N digit numbers. The algorithm is shown on the next page.

There are two stages to the algorithm. The first is to compute a 2D array whose rows contain the first number a multiplied by the single digits of the second number b (times the corresponding power of 10). The second stage is to add up the rows of this 2D array.

Note: in the example below, I have not shown various "carries" that were used to compute the 2D array and the final result.

[1] not code in a real programming language, but good enough for communicating between humans i.e. me and you

```
   352
 x 964
-------
  1408
 21120
316800
-------
339328
```

Algorithm 2 Multiplication (base 10) of two numbers a and b

for $j = 0$ to $N - 1$ **do**
  $carry \leftarrow 0$
  for $i = 0$ to $N - 1$ **do**
    $prod \leftarrow (a[i] * b[j] + carry)$
    $tmp[j][i + j] \leftarrow prod\%10$
    $carry \leftarrow prod/10$
  end for
  $tmp[j][N + j] \leftarrow carry$
end for
for $i = 0$ to $2 * N - 1$ **do**
  $sum \leftarrow carry$
  for $j = 0$ to $N - 1$ **do**
    $sum \leftarrow sum + tmp[j][i]$
  end for
  $r[i] \leftarrow sum\%10$
  $carry \leftarrow sum/10$
end for
$r[2 * N] \leftarrow carry$

Analysis of Algorithm

Let's compare the addition and multiplication algorithms in terms of the number of operations required. The addition algorithm involves a single **for** loop which is run N times. For each pass through the loop, there is a fixed number of simple operations. There are also a few operations that are performed outside the loop. We would say that the addition algorithm requires $c_1 + c_2 N$ operations, i.e. a constant c_1 plus a term that is proportional (with factor c_2) to the number N of digits. We are ignoring the details that define c_1 and c_2 which have to do with the actual machine implementation of the various instructions. (You will learn about this in COMP 273.)

We saw that the multiplication algorithm involves two components, each having a pair of **for** loops, one inside the other. This "nesting" of loops leads to N^2 passes through the operations within the inner loop. For each pass, there are various basic operations performed.

Suppose we consider the first component, in which we produce the two dimensional matrix *tmp*. Suppose some number (say c_3) of operations are inside both **for** loops, some number (say c_4) of

operations are inside just one of the **for** loops, and some number (say c_5) of operations are outside both **for** loops. Then the number of operations is $c_5 + c_4N + c_3N^2$. The same argument would apply for the second step of the multiplication algorithm, since again we have two nested **for** loops.

To summarize, the number of operations taken by the addition and multiplication algorithms depends both on the c values as well as on the number of digits N. *It is very important to realize that, for large N, multiplication requires far more operations than additions since N^2 will dominate over N, regardless of the particular values of the c's.*

Let's next consider the space that is required to perform the above algorithms. The addition algorithm used arrays $a[], b[], r[]$ which are each of size N so the space required grows with N. The multiplication algorithm used a 2D array $tmp[][]$ which grows with N^2. When N is very large, the space requirements would be significant. Can this be avoided? Yes, it can.

Rather than making a 2D table and then summing up the rows after the table is constructed, you could build the rows one at a time and use the $r[]$ vector to acculumate the sum *as you build the rows*. Once a row has been added to the sum, there is no reason to keep it around. Notice that this does not speed up the algorithms. The total number of operations still grows with N^2, since you ultimately are adding up N rows which each have N digits. But you are saving significantly on space.

Subtraction

In grade school, you also learned how to perform subtraction. Subtraction was more difficult to learn than addition since you needed to learn the trick of borrowing, which is the opposite of carrying. In the example below, you needed to write down the result of 2-5, but this is a negative number and so instead you change the 9 to an 8 in the first number and you change the 2 to a 12, then compute 12-5=7 and write down the 7.

```
   924
 -  352
 -------
   572
```

This "borrowing" trick is straightforward (easy to say now!). In Assignment 1, you will be asked to code up an algorithm for doing this.

Long Division

The last basic arithmetic operation you learned in grade school was long division. Suppose you want to compute `41672542996 / 723`. Again you learned a method for doing this.

```
        5 ...
    --------------
723 |  41672542996
       3615
       ----
        552  ...
```

In the example above, you asked yourself, does 723 divide into 416? No. So, then you figure out how many times 723 divides into 4167. The answer is 5. You multiply 723 by 5 and then subtract this from 4167, etc, etc.

Why does this work? How would you write it down as an algorithm? (In Assignment 1, you will be required to code it up.)

Arithmetic in other bases?

You are used to representing positive integers in base 10, that is, as a sum of powers of 10. But there is nothing special about the number 10. In principle, you can represent positive integers as the sum of numbers in any base.

Let's consider a few examples of representing numbers in base 8, that is, as a sum of powers of 8. So,

$$(\ a[N-1]\cdots a[2]\ a[1]\ a[0]\ )_8\ =\ a[N-1]*8^{N-1}+\cdots+a[2]*8^2+a[1]*8+a[0]$$

where the $a[i]$ are all in $\{\ 0,1,2,\cdots,7\}$. For example,

$$(736)_8 = 7*8^2+3*8+6 = (478)_{10}$$

and

$$(1205)_8 = 1*8^3+2*8^2+0*8+5 = (645)_{10}.$$

The addition and multiplication algorithms we saw earlier work similarly as before. See if you can carry out the following sum and product of two numbers which are written in base 8. (Note: For technical reasons having to do with a typesetting hassle, I have not put the subscript 8 below which indicates base 8. But you should consider it there!)

```
  (1205)
+ (736)
-------
  (2143)      which is 1123 in base 10.
```

```
   (1205)
 * (736)
 -------
   (7436)
  (3617 )
 (10643  )
-----------
 (1132126)    which is 308310 in base 10.
```

Binary numbers

The reason humans represent numbers using decimal (the ten digits from 0,1, ... 9) is that we have ten fingers. There is no other reason than that. There is nothing special otherwise about the number ten. Computers don't represent numbers using decimal. Instead, they represent numbers using binary, or "base 2". Let's make sure we understand what binary representations of numbers are. We'll start with positive integers.

In decimal, we write numbers using *digits* $\{0, 1, \ldots, 9\}$, in particular, as sums of powers of ten. For example,

$$(238)_{10} \;=\; 2 * 10^2 + 3 * 10^1 + 8 * 10^0$$

In binary, we represent numbers using *bits* $\{0, 1\}$, in particular, as a sum of powers of two:

$$(11010)_2 \;=\; 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$$

I have put little subscripts (10 and 2) to indicate that we are using a particular representation (decimal or binary). We don't need to always put this subscript in, but sometimes it helps.

Let's consider how to count in binary. You should verify that the binary representation is a sum of powers of 2 that indeed corresponds to the decimal representation on the left.

```
decimal        binary
-------        ------
   0              0
   1              1
   2             10
   3             11
   4            100
   5            101
   6            110
   7            111
   8           1000
   9           1001
  10           1010
  11           1011
      etc
```

Just as we did last lecture with other bases, let's add two numbers which are written in binary:

```
  11010          26
+ 1111           15
  -----          --
 101001          41
```

Make sure you see how this is done, namely how the "carries" work. For example, in column 0, we add $0 + 1$ to get 1 and there is no carry. In column 1, we add $1 + 1$ (in fact, $1 * 2^1 + 1 * 2^1$) and we get $2 * 2^1 = 2^2$ and so we carry a 1 over column 2. Do not proceed further until you understand this.

Converting from decimal to binary

It is trivial to convert a number from a binary representation to a decimal representation. You just need to know the decimal representation of the various powers of 2.

$2^0 = 1,\ 2^1 = 2,\ 2^2 = 4,\ 2^3 = 8,\ 2^4 = 16,\ 2^5 = 32,\ 2^6 = 64,\ 2^7 = 128,\ 2^8 = 256,\ 2^9 = 512,\ 2^{10} = 1024,\ \ldots$

Then, for any binary number, you write each of its bits as a power of 2 (in decimal) and then you add up these decimal numbers, e.g.

$$11010_2 \ = \ 16 + 8 + 2 = 26.$$

The other direction is more challenging. How do you convert from a decimal number to a binary number?

Here is an algorithm for doing so which is so simple you could have learned it in grade school. The algorithm repeatedly divides by 2 and the "remainder" bits give us the binary representation. Recall that "/" is the integer division operation which ignores the remainder i.e. fractions. If you want the remainder of the division, use "%" which is sometimes called the "mod" (or modulus) operator.

Algorithm 3 Convert decimal to binary

INPUT: a number m expressed in base 10 (decimal)
OUTPUT: the number m expressed in base 2 using a bit array $b[\,]$

```
i ← 0
while m > 0 do
    b[i] ← m%2
    m ← m/2
    i ← i + 1
end while
```

Example: Convert 241 to binary

i	m	b[ ]
0	241	
1	120	1
2	60	0
3	30	0
4	15	0
5	7	1
6	3	1
7	1	1
8	0	1
9	0	0
10	:	:

and so $(241)_{10} = (11110001)_2$. Note that there are an infinite number of 0's on the left which are higher powers of 2 which we ignore.

What does this algorithm work? To answer this question, it helps to recall a few properties of multiplication and division. Let's go back to base 10 where we have a better intuition.

Suppose we have a positive integer m which is written in decimal (as a sum of powers of 10) and we then multiply m by 10. There is a simple way to get the result, 10 m, namely shift the digits left by one place and put a 0 in the rightmost position. So, $238 * 10 = 2380$ and the reason is

$$238 * 10 \;=\; (2 * 10^2 + 2 * 10^1 + 8 * 10^0) * 10 = 2 * 10^3 + 3 * 10^2 + 8 * 10^1 + 0 * 10^0.$$

Similarly, to divide a number m by 10, we shift the digits to the right

$$238/10 \;=\; (2 * 10^2 + 3 * 10^1 + 8 * 10^0)/10 = 2 * 10^1 + 3 * 10^0$$

We have dropped the rightmost digit 8 (which becomes the remainder) since we are doing integer division, and thus ignoring terms with negative powers of 10 i.e $8 * 10^{-1}$.

In binary, the same idea holds. If we represent a number m in binary and multiply by 2, we shift the bits to the left by one position and put a 0 in the rightmost position. So, e.g. if

$$m = (11010)_2 \;=\; 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$$

then multiplying by 2 gives

$$(110100)_2 \;=\; 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0.$$

If we divide by 2, then we shift right by one position and drop the rightmost bit (which becomes the remainder).

Let's put the left shift and right shifts together. For any positive integer m, we can write

$$m = 10 * (m/10) + (m\%10),$$

for example,

$$549 = 540 + 9 = 10 * (549/10) + (549 \;\%\; 10).$$

More generally, for any positive integer – call it *base* – we can write

$$m = base * (m/base) + (m \;\%\; base),$$

and in particular, in binary we use $base = 2$, so

$$m = 2 * (m/2) + (m \;\%\; 2).$$

Now let's apply these ideas to the algorithm. Representing a positive integer m in binary means that we write it

$$m = \sum_{i=0}^{n-1} b_i \; 2^i$$

where b_i is a bit, *i.e* either 0 or 1. So we write m in binary as a bit sequence $(b_{n-1}\ b_{n-2}\ \ldots\ b_2\ b_1\ b_0)$. In particular,

$$\begin{aligned} m\ \%\ 2 &= b_0 \\ m\ /\ 2 &= (b_{n-1} \ldots b_2 b_1) \end{aligned}$$

Thus, the algorithm essentially just uses repeated division and mod to read off the bits of the binary representation of the number.

If you are still not convinced, let's run another example where we "know" the answer from the start and we'll see that the algorithm does the correct thing. Suppose our number is $m = 241$, which is $(11110001)_2$ in binary.

i	m	b[i]
0	11110001	
1	1111000	1
2	111100	0
3	11110	0
4	1111	0
5	111	1
6	11	1
7	1	1
8	0	1

and so the remainders are just the bits used in the binary representation of the number.

Interestingly, the same algorithm works for any base. For example, let's convert 238 from decimal to base 5, that is, let's write 238 as a sum of powers of 5. We apply the same algorithm as earlier but now we divide by 5 at each step and take the remainder.

```
238
 47          3
  9          2
  1          4
  0          1
```

and so $(238)_{10} = (1423)_5$. Verify this by converting the latter back into decimal by summing powers of 5.

Binary fractions

Up to now we have only talked about integers. We next talk about binary representations of fractional numbers, that is, numbers that lie between the integers. Take a decimal number such as 26.375. We write this as:

$$(26.375)_{10} = 2 * 10^1 + 6 * 10^0 + 3 * 10^{-1} + 7 * 10^{-2} + 5 * 10^{-2}.$$

The "." is called the *decimal point*.

One uses an analogous representation using binary numbers, *e.g.*

$$(11010.011)_2 \;=\; 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2} + 1*2^{-3}$$

where "." is called the *binary point*. Check for yourself that this is the same number as above, namely

$$16 + 8 + 2 + 0.25 + 0.125 \;=\; 26.375.$$

How do we convert from decimal to binary for such fractional numbers in general ? Suppose we have a number x that is written in decimal and has a finite number of digits to the right of the binary point. Let's look at a particular example. *This is different from the one given in the slides.*

Let $x = 4.63$ and let's convert it to binary. Since $x = 4 + .63$, we know the answer will have the form 100.___ since 100 is the binary representation of 4 and .63 is a sum of powers of negative powers of 2. So we just need to find the bits to the right of the binary point. To get the first say five bits, we multiply and divide by 2 five times (or alternatively, as was suggested in class, we just directly multiply 0.67 by $2^5 = 32$ and also divide by 2^5).

$$\begin{aligned} 0.67 &= 1.34 * 2^{-1} \\ &= 2.68 * 2^{-2} \\ &= 5.36 * 2^{-3} \\ &= 10.72 * 2^{-4} \\ &= 21.44 * 2^{-5} \end{aligned}$$

We convert $(21)_{10}$ from decimal to binary which gives $(10101.\_\_)_2$ and then we shift the binary point left by five places. Thus .67 = .10101___, and so 4.63 = 100.10101___. If we drop off the unspecified part to have a finite number of bits only, then we have an approximation.

As was observed in class, we can sometimes obtain a better approximation than truncating (chopping) the unknown bits. In this example, the approximation error is $0.44 * 2^{-5}$ and since 0.44 is closer to 0 than to 1, we are better of truncating. For other examples, however, it might be better to "round up". This is the case of the one example in the slides, where we had

$$\begin{aligned} (.247)_{10} &= 3.952 * 2^{-4} \\ &= (.0011\_\_)_2 \end{aligned}$$

Since .952 is closer to 1 than to 0, we would obtain a better approximation we replaced $.954 * 2^{-4}$ by $1 * 2^{-4}$, and so the approximation would be $(.0011)_2 + 2^{-4} = (.0100)_2$.

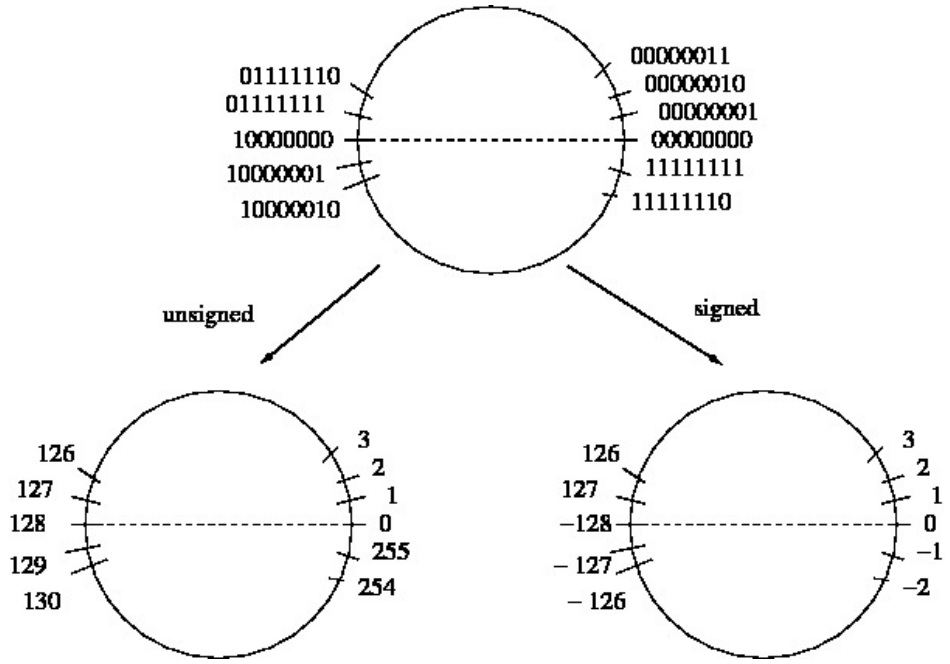
Today I will will the discussion of binary numbers. Then I will move on and discuss arrays and an algorithm for sorting which uses arrays. The discussion of arrays is a natural lead into the next big topic, lists, which I will begin next lectures.

Unsigned vs. signed numbers

Consider 8 bit numbers (one byte). From what I said last lecture, these numbers would go from 0 to 255. In general, n-bit numbers would go from 0 to 2^{n-1}. Such a representation is called *unsigned* since all the numbers are non-negative. It is illustrated by the figure below on the left.

Note that rather than showing a line of numbers, I have drawn the numbers on a circle. The idea here is that if take $(11111111)_8 = 255$ and you add 1, then you get $(100000000)_8 = 256$ which is has 9 bits rather than 8. If we only keep the "lower" 8 bits, then we have $(00000000)_8 = 0$. Hence, the circle.

To allow for negative numbers, one uses a different interpretation of the binary numbers, which is illustrated in the figure below on the right. This is called the *signed* number representation. Here we count from 0 up to 127 and then, rather than going to 128, we jump to -128. We continue counting up from there to 0. Note that the leftmost bit (the "most significant bit" or MSB) indicates the sign of the number. If the MSB is 0, then the number is non-negative. If the MSB is 1, then the number is negative. Also, note that we are not simply using 7 bits for the number and then using the eight bit for the sign. i.e. (10000001) represents the number -127, not -1.



More generally, the set of *unsigned* n-bit numbers are $\{\ 0,\ 1,\ 2, \ldots,\ 2^n - 1\ \}$. It is common to use $n = 16,\ 32,\ 64$ or 128, though any value of n in possible. The *signed* n bit numbers are $\{-\ 2^{n-1}, \ldots,\ 0,\ 1,\ 2, \ldots,\ 2^{n-1} - 1\ \}$. Here is a table for $n = 8$ and $n = 16$.

binary	signed	unsigned

00000000	0	0
00000001	1	1
:	:	:
01111111	127	127
10000000	-128	128
10000001	-127	129
:	:	:
11111111	-1	255

If n=16, the corresponding table is:

binary	signed	unsigned
0000000000000000	0	0
0000000000000001	1	1
:	:	:
0000000001111111	127	127
0000000010000000	128	128
0000000010000001	129	129
:	:	
0111111111111111	$2^{15} - 1$	$2^{15} - 1$
1000000000000000	-2^{15}	2^{15}
1000000000000001	$-2^{15} + 1$	$2^{15} + 1$
:	:	
1111111101111111	-129	$2^{16} - 129$
1111111110000000	-128	$2^{16} - 128$
1111111110000001	-127	$2^{16} - 127$
:	:	
1111111111111111	-1	$2^{16} - 1$

Funny example from Java

Consider the following lines of Java code:

```
for (short s = 32767; s < 32768; s++)
   System.out.println(s);
```

The number 32767 is $2^{15} - 1$ and so it is the largest `short`. The code is a loop which starts with this number, prints it, and then increments the number. The loop terminates when the value is not less than 32768 which common sense tells you will occur after just one pass through the loop. However, that is not what happens! When the program you write contains the number 32768, Java treats this number as an `int` and this number can indeed be represented correctly as a (32 bit) `int`. When Java adds 1 to the `short` value `s`=32767, it gets -32768. This value is less than 32768 obviously and so the program will print it and indeed it will print all the values from -32768 up to 32767, and then go through that loop infinitely many times. If you don't believe me, try it yourself. *See Exercises 1 Question 10 for a few other examples.*

Unsigned numbers as memory addresses

Binary number representations are used in two ways. The first is to represent *data*. The second is to represent the *addresses* in memory. Addresses are unsigned numbers. When you hear that a computer is a "32 bit machine" this means that there are 2^{32} addressable bytes. A "64 bit machine" has 2^{64} bytes. Don't be concerned for now about what this actually means in terms of hardware. You'll learn about that in COMP 273. For now, just think of possible addresses of bytes in memory.

As as aside, just to remind you (or in case you didn't know)...

- 2^{10} bytes = 1 kilobyte (1 KB) $\approx 10^3$ bytes (one thousand)
- 2^{20} bytes = 1 megabyte (1 MB) $\approx 10^6$ bytes (one million)
- 2^{30} bytes = 1 gigabyte (1 GB) $\approx 10^9$ bytes (one billion)
- 2^{40} bytes = 1 terabyte (1 TB) $\approx 10^{12}$ bytes (one trillion)
- 2^{50} bytes = 1 petabyte (1 PB) $\approx 10^{15}$ bytes
- 2^{60} bytes = 1 exabyte (1 EB) $\approx 10^{18}$ bytes

The latter two seem like very large numbers. Indeed there are data sets (sometimes called "big data") with that many bytes.

You learned in COMP 202 that Java has primitive types and reference types. Each primitive type in Java uses a certain number of bits (or bytes), namely `boolean, byte, char, short, int, long, float, double` use 1,1,2,2,4,8,4,8, bytes respectively. A variable that is defined as a *primitive type* just stands for a particular set of consecutive bytes of memory where some data is stored.

Variables that have a *reference type* are different. These variables holds an *address* of an object, in particular, the starting address of the object.

You should think of both primitive and reference variables as standing for an address in memory where something is stored. The thing that is stored can either be data itself (in the case of a primitive type), or it could be the starting address of some object (in the case of a reference type).

Note that sometimes I write *starting address* rather than just address. Why? In Java only `boolean` and *byte* types use one byte. Everything else uses multiple bytes. When I say "starting address", I am just reminding you that the item being stored takes more than one byte, and the address only refers to the first of these bytes.

Example

In class I discussed an example. There was alot of discussion and here I give an abridged version. See the figure slides for further details.

Suppose we define:

```
int  i  = 4;
double  x = 47.35;
```

This defines 4 consecutive bytes somewhere in memory where the integer `i` is stored and 8 consecutive bytes where the double `x` is stored. Whenever your program subsequently has `i` or `x`, it is referring to these two sequences of bytes. We would say that the address of the integer `i` is the first byte the 4 byte sequence and the address of the double `x` is the first byte of the 8 byte sequence.

Now suppose we have defined a class `Student`, and we have an instruction

```
Student[ ]  studentArray =  new Student[13];
```

This is in fact two declarations

```
Student[ ]      studentArray;
studentArray  = new Student[13];
```

The first defines a reference variable which holds the address of an object, namely a `Student` array. Until we construct the object, this address is `null` which is just the number 0. The second line constructs/instantiates a `Student` array which reference 13 students. These references are initialized to `null` which is the address 0.

Next we instantiate objects of the class `Student` and and use the array to reference them,

```
studentArray[0] = new  Student("Fred");}
studentArray[2] = new  Student("Mustafa");
```

In this case, we would have a `Student` object which has some string field in it that is assigned "Fred". This `Student` object would have starting address which is stored the array slot 0 in the `Student` array object.

The key concept to understand here is that any data – whether it is an `int` or a `double` or a an array, or an object or an array of objects – is stored as a set of consecutive bytes in memory. When we talk about the "address" of any particular data item, we are talking about the first of these bytes. We will come back to these ideas throughout the course...

Insertion Sort: an algorithm for sorting an array

Let's use arrays to solve a problem that comes up often in programming, namely *sorting*. Suppose we have an array of objects that is in no particular order and the objects are such that it is meaningful to talk about an ordering e.g. the objects might be numbers, or they might be strings which can be sorted alphabetically (e.g. to make a phonebook). Given the under-ordered items in an array, we would like to rearrage the items in the array such that they are sorted.

There are possible algorithms for doing so. "Insertion sort" is one of the simplest to describe. The basic idea of the algorithm is to assume that the k elements of the array (indices $0, \ldots, k-1$) are already in the correct order, and then insert element at index k into its correct position with respect to the first k elements. We start with $k = 0$. The first element is clearly in its correct position if we only have one element, so there is nothing to do.

Now suppose that the first k elements are correctly ordered relative to each other. How do we put the element at index k into its correct position? The idea is to look backwards from index k until we find the right place for it. Element $a[k-1]$ is the largest of all $a[0], \ldots, a[k-1]$ by assumption, since the first k elements are in their correct order. When stepping backwards, if the element in the next position is greater than $a[k]$, then we move that element forward in the array

to make room for the $a[k]$. If, on the other hand, if we find an element that less than (or equal to) $a[k]$ then we go no further.

The algorithm is listed on the next page. You should step through it and make sure you follow it. I suggest you also have a look at an applet that allows you to visualize how the algorithm works, such as `http://tech-algorithm.com/articles/insertion-sort`.

ALGORITHM: INSERTION SORT
INPUT: an array $a[\,]$ with N elements that can be compared $(<, =, >)$
OUTPUT: the array $a[\,]$ containing the same elements, in increasing order

```
for k = 1 to N − 1 do
    tmp ← a[k]
    i ← k
    while (i > 0) & (tmp < a[i − 1]) do
        a[i] ← a[i − 1]
        i ← i − 1
    end while
    a[i] = tmp
end for
```

[ASIDE: I did not get to cover the following in class. You should read it on your own (for homework).]

Analysis of insertion sort

Suppose you are given an array $a[\,]$ which is of size N, and you step through this algorithm line by line. How many steps will you take? Interestingly, the answer depends on the data.

Suppose the operations inside the **for** loop but outside the **while** loop take c_1 time steps, and the operations inside the **while** loop take c_2 times steps. Then, in the worst case, you need to take about $c_1N + c_2(1 + 2 + \cdots + N - 1)$, where the latter expression occurs in the case that, for each k, the **while** loop decrements i all the way from k back to 0. But you should recall from high school math that

$$1 + 2 + \ldots N - 1 = \frac{N(N-1)}{2}$$

so you can see that in the worst case the algorithm takes time that depends on N^2. This worst case scenario occurs in the case that the array is already sorted, but it is sorted in the wrong direction, namely *from largest to smallest.*

In the "best" case, the array is already correctly sorted from smallest to largest. Then the condition tested in the **while** loop will be false every time (since $a[i-1] < tmp$), and so each time we hit the **while** statement, it will take a *constant* amount of time. This is the *best case* scenario, in the sense that the algorithm executes the fewest operations in this case. Since there are N passes through the **for** loop, the time taken is proportional to N.

[I strongly recommend that you consult the slides as well as the notes for the next few lectures. The slides have many important figures which I am not reproducing here.]

array as a "list"

In the next part of the course, we consider data structures that hold an ordered set of elements. By "ordered', I mean that we can talk about the first element, the second element, etc. Such a set of ordered objects is often called a *list*. One also refers to list data structures as "linear data structures".

The *array* is a natural data structure for representing a list. If we have *size* elements in the list, then we store these elements in positions $0, 1, \ldots, size - 1$. An array allows us to access any of the elements in the list by providing the position (index) of the element. Arrays can sometimes be an awkward way to represent a list, however. If you want to insert a new element into the array then you have to make room for that element in the array. For example, if you want to insert a new element at the front (into array slot 0), then you need to move all the elements ahead one position ($i \rightarrow i + 1$). We saw this type of problem with the insertion sort algorithm last lecture.

```
//  add new element to front of the list
//  assuming that there is room left in the array
//
    for (i = size; i > 0; i--)
       a[i] = a[i-1]
    a[0] = new element
    size = size + 1
```

Note that you need the for loop to go backwards. Think what happens if you go forwards!

A similar issue arise if you want to remove an element, namely you need to shift all elements back one position ($i \rightarrow i - 1$).

```
//  remove the element at front of the list
//
    for (i = 1; i < size-1; i++)
       a[i-1] = a[i]
    a[size-1] = null
    size = size - 1
```

If you have *size* elements in the array, then adding or removing the element with index 0 takes *size* operations. This is obviously inefficient, especially if we are doing alot of adding (also called "insertions") and removals (also called "deletions") at the front of the list.

Compare the above to the problem of adding or removing at the last element in the list:

```
//  add new last element to the list
//  assuming that there is room left in the array
//
    a[size] = new element
    size = size + 1
```

```
//  remove the last element from the list
//
    a[size-1] = null
    size = size - 1
```

These algorithms take constant time, i.e. independent of the number of elements in the array.

One note about the above "algorithms" is that, when we remove an element, we have been careful to set the slot to have a `null` value. This is not strictly speaking necessary, since we have a `size` variable which keeps track of how big the list is and hence it keeps track of where the elements are, namely in slots 0 to `size-1`.

Singly Linked lists

Let's look at an alternative data structure for representing a list. Define a list *node* which contains:

- an element of a list (this could be the element itself or it could be a reference to an element)
- a reference to the `next` node in the list.

In Java, we could have a node class defined as follows:

```
class  SNode{
   Type         element
   SNode        next
}
```

where `Type` is the type of the object in the list. To define the list itself, we would define another class:

```
class  SLinkedList{
   SNode        head;
   SNode        tail;
   int          size;
}
```

The field `head` points to the first node in the list and the field `tail` points to the last node in the list. If there is only one node in the list, then `head` and `tail` would point to the same node.

Let's look at a few methods for manipulating a linked list:

```
addFirst( newNode ){
   newNode.next = head
   head = newNode
}
```

Here we assume the input parameter is a node rather than an element. If we were to write this "algorithm" in Java, we would need to be more careful about such things. Don't concern yourself with this detail here. Instead, notice the order of the two instructions. The order matters!

```
removeFirst(){
   tmp = head
   head = head.next
   tmp.next = null
}
```

Notice that we have used `tmp` here. We could have just had one instruction (`head = head.next`) but then the old first node in the list would still be pointing to the new first node in the list, even though it isn't part of the list. (You might argue that since the old first node is not part of the list anymore, then you don't care if it points to the new first node. In that case, the first and third instructions above which involve `tmp` would be unnecessary.)

Also note that the `removeFirst()` method ignores certain cases. For example, if there is only one element in the list, then removing the first means that we are also removing the last. In that case, we should set the `tail` reference to `null`.

Next lecture, we will look at what happens when you add or remove from the end of a linked list.

Singly linked lists (continued....)

I began the lecture by discussing two more methods that are commonly defined for linked lists, namely adding or removing an element at the back of the list. This requires manipulating the `tail` reference.

Adding a node at the tail can be done in a small number of steps.

```
addLast( newNode ){
   tail.next = newNode
   tail = tail.next
}
```

Removing the last node from a list, however, requires many steps. The reason is that you need to modify the `next` reference of the the node that comes before the *tail* node which you want to remove. But you have no way to directly access the node that comes before `tail`, and so you have to find this node by searching from the front of the list.

The algorithm begins by checking if the list has just one element. If it does, then the last node is the first node and this element is removed. Otherwise, it scans the list for the next-to-last element.

```
removeLast(){
   if (head == tail)
      head = null
      tail = null
   }
   else{
     tmp = head
     while (tmp.next != tail){
        tmp = tmp.next
     }
     tmp.next = null
     tail = tmp
   }
}
```

This method requires about `size` steps. This is significantly more expensive than what we had with an array implementation, where we had a constant cost in removing the last element from a list. We will come back to this comparison at the end of the lecture, when we compare arrays with singly and doubly linked lists.

Java generics

In our discussion of linked lists, we concentrated on how to add or remove a node from the front or back of a list. The fact that each node held a primitive type versus a reference type, or whether the reference type was a `Shape` object or `Dog` object (or whatever) was not important to that discussion. Rather, we were mainly concerned with how to manipulate the nodes in the list.

When you implement a linked list in some programming language such as Java, you do have to consider the types of elements at each node. However, you don't want to have to reimplement your linked list class for each new type (`Shape, Dog, etc`. Java allows you to write classes with a *generic type*, which addresses this issue.

Rather than declaring classes for a particular type `E` – for example, `E` being an `int` or `Shape` as we saw last lecture – we would like to define our linked list in a more general way.

```
class  SNode{
   E            element
   SNode        next
      :
}

class  SLinkedList{
   SNode        head;
   SNode        tail;
   int          size;

      :
}
```

Java allows us to do so. We can declare these classes to have a *generic* type `E`.

```
class  SNode<E>{
   E            element
   SNode<E>        next
     :
}

class  SLinkedList<E>{
   SNode<E>        head;
   SNode<E>        tail;
   int             size;
     :
}
```

This way, we can write all the methods of these classes such that they can be used for any type of reference type. For example, we could have a linked list of `Shape` objects, or a linked list of `Student` objects, etc.

```
SLinkedList<Shape>    shapelist = new SLinkedList<Shape>();
SLinkedList<Student>  studentlist = new SLinkedList<Student>();
```

You should think of the class name as a parameter `E` that is passed to the constructor `SLinkedList<E>()`.

See the examples that are given in the online code.

Doubly linked lists

The "S" in the `SLinkedList` class did not stand for `Shape`, but rather it stood for "singly", namely there was only one link from a node to another node. We next look at "doubly linked" lists. Each node of a doubly linked list has two links rather than one, namely each node of a doubly linked list has a reference to the previous node in the list and to the next node in the list. These reference variables are typically called `prev` and `next`.

```
class  DNode<E>{
   E            element;
   DNode<E>        next;
   DNode<E>        prev;
     :
}

class  DLinkedList<E>{
   DNode<E>        head;
   DNode<E>        tail;
   int             size;
     :
}
```

One advantage of doubly linked lists is that they allow us to access elements near the back of the list without having to step all the way from the front of the list. Recall the `removeLast()` method from earlier this lecture. To remove the last node of a singly linked list, we needed to follow the next references from the head all the way to the node whose next node was the last/tail node. This required about `size` steps which is inefficient. If you have a doubly linked list, then you can remove the last element in the list in a constant number of steps, e.g.

```
tail      =  tail.prev
tail.next = null
size      = size-1
```

More generally, with a doubly linked list, we can remove an arbitrary element in the list in constant time.

```
remove( node ){
   node.prev.next = node.next
   node.next.prev = node.prev
   size--
}
```

(Here I am not concerning myself with the `next` and `prev` references in the removed node, i.e. if we don't set them to `null` then they will continue to point to nodes in the list.)

Another advantage of doubly linked lists is that, if we want to access an element, then we don't necessarily need to start at the beginning of the list. If the node containing the element is near the

back of the list, then we can access the node by following the links from the back of the list. For example, suppose we wished to remove the i'th node.

```
getNode(i){
   if (i < size/2){
      tmp = head
      index = 0
      while (index < i){
         tmp = tmp.next
         index++
      }
   else{
      tmp = tail
      index = size - 1
      while (index > i){
         tmp = tmp.prev
         index--
      }
   return tmp
}
```

This algorithm takes at most `size/2` steps, instead of `size` steps.

Dummy nodes

The above method for removing a node assumes that `node.prev` and `node.next` are well defined. However, this sometimes will not be the case, for example, if the node is the first or the last in the list, respectively. Thus, to write correct code for removing an element, you need to take these special end cases into account. (Recall the `removeLast()` method on page 1.) It is easy to forget to do so.

To avoid this problem, it is common to define linked lists by by using a "dummy" head node and a "dummy" tail node, instead of head and tail reference variables. The dummy nodes[2] are nodes just like the other nodes in the list, except that they do not contain an element (e.g. a shape object). Rather the `element` field points to `null`. These nodes do not contribute to the `size` count.

```
class  DLinkedList<E>{
   DNode<E>        dummyHead;
   DNode<E>        dummyTail;
   int             size;
     :
```

[2]Dummy nodes are somewhat controversial. Some programmers regard them as an ugly "hack". Others find them elegant and useful. I personally don't have a strong opinion either way. I am teaching you about dummy nodes so that you are aware of them.

```
    DLinkedList<E>(){
       dummyHead = new DNode<E>();
       dummyTail = new DNode<E>();
       size      = 0;
    }

    //  ... lots of list methods
}
```

Comparison of arrays and linked lists

Let's compare the time required to execute several methods, using arrays versus singly versus doubly linked lists. Let `N = size`, namely the number of elements in a list.

	array	singly linked list	doubly linked list
addFirst	N	1	1
removeFirst	N	1	1
addLast	1	1	1
removeLast	1	N	1
getNode(i)	1	i	min(i, N/2 - i)

TODO

To understand how linked lists work, you should go beyond reading these notes. You should do the Exercises 2a, and you should also have a look and run the code that I have made available to you (next to these lecture notes).

List as an "abstract data type" (ADT) or "interface"

We have considered three data structures for representing a list of elements: an array, a singly linked list, and a doubly linked list. Regardless of which of these data structure we use, operations such as adding or removing from the front or back of the list, or removing the i-th element of the list, or adding an element e before the i-th element of the list, etc, are all well defined. We can say what these operations do, without saying how the list is implemented. In this sense, a list is an *abstract data type* (sometimes called an ADT), namely a particular set of things and a particular set of methods that can be applied on/by/with these things. We have seen several examples:

```
add(i,element)  //  Inserts element into the i-th position
                //  (and increments the indices of elements that were
                //  previously at index i or up)
set(i,element)  //  Replaces the element in the i-th position
remove(i)       //  Removes the i-th element from list
get(i)          //  Returns the i-th element (but doesn't alter list)
clear()         //  Empties list.
isEmpty()       //  Returns true if empty, false if not empty.
size()          //  Returns number of elements in the list
  :
```

We can think of these methods as an *interface*[3] to a list object, in the sense that the methods allow us to perform operations on the list. Sometimes we don't care how the list is implemented. We just want to make sure that it performs the operations as are specified in the interface.

Other times, we do care how the list is implemented, since this may affect the speed and the memory space that is used. We saw, for example, that singly linked lists use less memory than doubly linked lists but singly linked lists are slower for some operations. Our choice of using a singly linked versus doubly linked list might depend on whether this tradeoff is a consideration for our application.

Java `LinkedList`

Java has a `LinkedList` class which is implemented as a doubly linked list.

```
LinkedList<Student>  studentList  =  new  LinkedList<Student>();
```

You should check out the Java API for the methods that this class provides, beyond those listed above.

Suppose we add n students to the front (or back) of the list. For example, the `add` method appends the element to back of the list. Adding n students to an empty list takes about cn steps, where c is the number of steps required to set the `prev` and `next` references in the nodes of the underlying data structure.

Suppose we have a list of n elements which is implemented using a doubly linked list, and we want to print out the elements. What if we were to define a `display` method that prints each of the elements. At first glance, the following pseudocode would seem to work fine.

[3]The word "interface" has a specific technical meaning in Java which we will discuss later in the course.

```
for (j = 1; j < n;  j++)
   print( mylist.get(j) )        //  or the equivalent Java statement
```

For simplicity, suppose that `get` is implemented by starting at the head and then stepping through the list, following the next reference. Then, with a linked list as the underlying implementation, the above code would require

$$1 + 2 + 3 + ... + n = \frac{n(n+1)}{2}$$

which grows like n^2. Eeeks! What went wrong? The j^{th} `get` starts again at the beginning of the (linked) list and walks to the j^{th} element, which is very inefficient. Note:

- Even if we were to take advantage of the doubly linked list and search for the i node by starting from the front or back, whichever is closer, the time still grows with n^2. See Exercises 2a.
- What alternatives to we have? In Java, we can use something called an `Iterator`. We will discuss these later in the course.

Java `ArrayList`

You have seen in COMP 202 that Java allows you to declare arrays, and these arrays can have either primitive or reference type, e.g.

```
int[]      intArray  = new int[desiredSize];
:
Student[]  studentArray = new Student[numStudents];
```

If you are using an array to maintain a list which can vary over time, then you will want to have methods for adding and removing elements such as we have been discussing. You don't want to write these list manipulation methods yourself. For this, Java has an `ArrayList` class. This class has a generic type, so you can define:

```
ArrayList<Student>  students = new ArrayList<Student>(numStudents);
```

The `ArrayList` class implements a list using an underlying array, but you do not index the elements of the array using the `[ ]` syntax that you are used to. Instead you access an array element using a `get` or `set` method (see below). Some of the `ArrayList` methods were listed above. Here I list some of them again, and specify what happens in the underlying array implementation.

```
   :
add(element);    //  Expands the underlying array, if it is full.
add(i,element);  //  Inserts a new element into the i-th position,
                 //  shifting up the positions of all elements with
                 //   index >= i.
remove(i)        //  Removes the element at the i-th position,
                 //  shifting down the positions of all elements with
                 //  index > i.
```

```
a.clear();       //  Makes a new (small) array with no elements or
                 //  alternatively sets all big array elements to null
a.size();    //  returns number of elements in the list (NOT the
             //  length of the underlying array)
```

It is important to realize that when you use the `ArrayList` class to represent a list, and you `add` or `remove` an elements to/from the *front* of your list, this operation will take time proportional to `size` (the number of elements in the list) since the index of every other element in the list changes. i.e. if you are removing then elements need to be shifted down and if you are adding then elements need to be shifted up.

Another important property of an `ArrayList` is that the underlying array has a certain length i.e. capacity. What happens if the underlying array is full and you try to add another element. In this case, the `add` method in Java's `ArrayList` class will create a bigger underlying array[4] and it will *copy* all references from the existing array to this new underlying array. It will then add the reference to the new element. The old array will become garbage, and the new bigger array will be used instead.

Copying references from a small array to a big array takes time. Let's think about how much. Suppose you start with an empty list and you add n items. To keep the math simple, suppose that whenever you try to add to a full array, you first double the size of the array (i.e. expanding by 100%). If you double the size k times (starting with size 1), then you have an underlying array with 2^k elements. So let's say $n = 2^k$.

How does the amount of work that we need to do depend on n? When we double the size of the underlying array and then fill it, we need to copy the elements from the smaller array to the bigger array and then fill the remaining elements of the bigger array. When $k = 1$, we copy 1 element from an array of size 1 to an array of size 2 and then add the second element in the array of size 2. When $k = 2$ and we expand from the array of size 2 to the array of size 4, we copy 2 elements and then add the remaining 2 elements. In general, the number of copies and adds together is:

$$1 + 2 + 4 + 8 + \cdots + 2^k.$$

But the sum is of the form

$$1 + x + x^2 + x^3 + \cdots + x^k = \frac{x^{k+1} - 1}{x - 1}$$

where $x = 2$ and so

$$1 + 2 + 4 + 8 + \cdots + 2^k = 2^{k+1} - 1.$$

Since we have expanded k times, we have an array of size $n = 2^k$ and. If we keep adding to fill that array (but no more) then we will have $n = 2^k$ elements in the array.

Thus, in adding n elements to an empty array list, we need to set n references (to set a reference to each element for the first time) and we also need to set $n - 1$ references (for the copying that needs to be done when we expand the underlying arrays k times). This accounts for the $2n - 1$ reference settings in total. Thus, doubling the size of a filled array and copying the references does require some work, and this extra amount of work grows with $2n$. This order of growth is not a problem; we need to set n references anyhow to add n elements.

[4] bigger by 50% in Java, but our arguments here assume we increase by 100% i.e. double the size of the array

Comparison of `ArrayList` and `LinkedList`: worst case

As a summary, let's look collect the *worst case* performance of the two implementations of lists, namely the number of steps we need to take. Here I am ignoring constant factors, including the division by 2 for the doubly linked list.

	`LinkedList`	`ArrayList`
`add(element)`	1	1
`add(i,element)`	n	n
`set(i,element)`	n	1
`remove(i)`	n	n
`get(i)`	n	1
`clear()`	1	1
`isEmpty()`	1	1
`size()`	1	1

ADT's in Java: the `interface`

At the beginning of the lecture, we discussed what a list was in an abstract sense: a set of things and a certain set of methods (operations) that are applied to these things. Stated in this general way, a *list* is an abstract data type (ADT). We will see two more ADT's in the next few lectures, namely the stack and the queue.

In Java, one does not use the term ADT. Rather, there an entity called an `interface`. An interface is set of methods, defined formally by a return type, a method name, and method argument types in a particular order, together called the *signature* of the method. An interface does *not* and *cannot* include an implementation of the methods. Rather, one needs to define a class that implements the interface. In Java, for example, there is a `List` interface (look it up in the Java API), and this interface is implemented by the `ArrayList` and `LinkedList` classes. We will see more examples of interfaces throughout the course.

Stack ADT

You are familiar with stacks in your everyday life. You can have a stack of books on a table. You can have a stack of plates on a shelf. In computer science, a *stack* is an abstract data type (ADT) with two operations: `push` and `pop`. You either push something onto the top of the stack or you pop the element that is on the top of the stack. A more elaborate ADT for the stack might allow you to check if the stack has any items in it (`isEmpty`) or to examine the top element without popping it (`top`, also known as `peek`). But these operations not necessary for us to call something a stack.

[ASIDE: Note that a stack is a kind of list, in the sense that it is a finite set of ordered elements. However, with stacks, you typically only are allowed to use `push, pop, isEmpty,` and `top` operations. In Java, you can declare a variable to be of type `Stack` which this allows you to use the methods of a `List` too. Traditional computer scientists frown this, since it blurs the notion of what is a stack. A traditional stack does not allow you to say `get(i)`.]

Example 1

Here we make a stack of numbers. We assume the stack is empty initially, and then we have a sequence of pushes and pops.

```
push(3)
push(6)
push(4)
push(1)
pop()
push(5)
pop()
pop()
```

The elements that are popped will be `1`, `5`, `4` in that order, and afterwards the stack will have two elements in it, with `6` at the top and `3` below it.

```
                 1            5
           4     4      4     4      4
     6     6     6      6     6      6     6
3    3     3     3      3     3      3     3
--   --    --    --     --    --     --    --
```

Example 2: Balancing parentheses

It often occurs that you have a string of symbols which include left and right parentheses that must be properly nested or balanced. (In this discussion, I will use the term "nested" and "balanced" interchangeably.) One checks for proper nesting using a stack.

Example 2a

Consider the opening and closing parentheses "`(`" and "`)`" and an expression:

```
3 + (4 - x)* 7 + (y - 2 * (2 + x)).
```

in which the parentheses *are* balanced. But how do we determine that it is balanced?

Let's scan through this expression and ignore the variables (x,y), numbers, and operators *, +, -. Whenever we see a left parenthesis, we push it on the stack, and when we see a right parenthesis, we pop the (matching) left parenthesis from the stack. The sequence of stack states is:

```
                    (
      (         (   (   (
---  ---  ---  ---  ---  ---  ---
```

If were to try to pop an empty stack (i.e. we reach a right parenthesis and the stack is empty), or we were to finish scanning the string and the stack were non-empty, then we would have an error – the parentheses would not be balanced.

For our problem here, you don't really need a stack. You just need a counter. You start the counter at 0 and increment the counter when you see a left parenthesis and decrement the counter when you see a right parenthesis. The parentheses are balanced if the counter never becomes negative, and it is 0 when you are finished scanning.

Example 2b

A more interesting example of balancing parentheses in which you *do* need a stack is if there are multiple types of left and right parentheses, for example, (,), {, }, [,], and you require that the parentheses are properly *nested*. Consider the string:

```
( ( [ ] ) ) [ ] { [ ] }
```

You can check for balanced parentheses using a stack. You scan the string left to right. When you read a left parenthesis you push it onto the stack. When you read a right parenthesis, you pop the stack (which contains only left parentheses) and check if the popped left parenthesis matches the right parenthesis that you just read. For the above example, the sequence of stack states would be as follows.

```
          [
      (   (   (                       [
   (  (   (   (   (       [       {   {   {
-  -  -   -   -   -   -   -   -   -   -   -   -
```

and the algorithm terminates with an empty stack. So the parentheses are properly balanced.

Example 2c

Here is an example where each type of parenthesis on its own is balanced, but overall the parentheses are not balanced.

```
( ( [ ) ] ) [ [ ] ] { [ } ]
```

```
             [
         (   (
     (   (   (
--- --- --- ---  X  since next symbol is ")" which doesn't match top
```

Algorithm for balancing parentheses

The basic algorithm for matching parentheses is shown below. We assume the input has been already partitioned ("parsed") into disjoint *tokens*. A token can be one of the following:

- a left parenthesis (there may be various kinds)
- a right parenthesis (there may be various kinds)
- a string not containing a left or right parenthesis (operators, variables, numbers, etc)

Any token other than a left or right parenthesis is ignored by the algorithm.

```
ALGORITHM:  CHECK FOR BALANCED LEFT AND RIGHT PARENTHESES
INPUT:  SEQUENCE OF TOKENS
OUTPUT: TRUE OR FALSE (I.E. BALANCED OR NOT)

WHILE (not at end of token sequence){
   token <-  get next token
   if token is a left parenthesis
      push(token)
   else if token is a right parenthesis {
           if (stack is empty)
               return false
           else{
               left  <-  pop()
               if  !( left.matches(token) )
                   return false
           }
         }
}
return (stack.isEmpty()
```

Example 3: HTML tags

The above problem of balancing different types of parentheses might seem a bit contributed. But in fact, this arises in many real situations. An example is HTML *tags*.[5] They are of the form `<tag>` and `</tag>` and these correspond to left and right parentheses, respectively. For example, `<b>` and `</b>` are "begin boldface" and "end boldface".

[5] If you have never looked at HTML source code before, then open a web browser right NOW and look at "view → page source" and check out the tags. They are the things with the angular brackets.

HTML tags are *supposed to be* properly nested[6] For example, consider

```
<b> I am boldface, <i> I am boldface and italic, </i>
</b><i> I am just italic </i>.
```

The tag sequence is `<b><i></i></b><i></i>` and the "parenthesis" are indeed balanced, i.e. properly nested. Compare that too

```
<b> I am boldface, <i> I am boldface and italic </b>
  I am just italic </i>
```

whose tags sequence is `<b><i></b></i>` which is not properly balanced. The latter is the kind of thing that novice HTML programmers write – and its logical. They are treating the tags and turning an imaginary attribute state (bold, italic) on or off. But the HTML language is not supposed to allows this. And writing HTML code this way can get you into trouble since errors such as a forgotten or extra parenthesis can be very hard to find.

See `http://www.w3schools.com` for basic HTML tutorials (and other useful simple tutorials).

Example 4: if-then-else statements (see Assignment 2)

Consider a language that allows two types of if-then-else statements:

```
if  boolean  then statement  else statement
if  boolean  then statement
```

Notice that statements can be nested, i.e. you can have a statement within a statement (within a statement within a statement etc). For such a language, you often need brackets to disambiguate a given statement. For example, there are two interpretations possible for the statement:

```
if  (i > 0) then  if (a > 0) then b=4 else b = 5
```

These two interpretations are

```
if  (i > 0) then { if (a > 0) then b=4   else b = 5 }    //  FIRST
if  (i > 0) then { if (a > 0) then b=4 } else b = 5      //  SECOND
```

Let `F,T` be false and true, respectively. The table below shows the actions taken in the four combinations of the conditions. Note how easy it is to make mistakes with this, even with brackets !

i > 0	a > 0	FIRST	SECOND
F	F	do nothing	b = 5
F	T	do nothing	b = 5
T	F	b = 5	do nothing
T	T	b = 4	b = 4

[6]Many HTML authors write improply nested HTML code. Because of this, web browsers typically will allow improper nesting. Why? Because web browser programmers (e.g. google employers who work on Chrome) want people to use their browser and if the browser displayed junk when trying to interpret improper HTML code, then users of the browser would give up and find another browser.

Interestingly if the language only allows one type of if-then-else statement:

```
if  boolean  then statement  else statement
```

then you don't need brackets, no matter how deep the nesting. For example,

```
if (i > 0) then if (a > 0) then b=4 else if (i < 5) then b = 6 else b = 7
```

is uniquely parsed as

```
if (i>0) then {if (a>0) then b=4 else {if (i < 5) then b = 6 else b = 7}}
```

In Assignment 2, you will write a stack-based algorithm for doing this.

Example 5: stacks in graphics

The next example is a simple version of how stacks are used in computer graphics. Consider a drawing program which can draw unit line segments (say 1 cm). Suppose the pen tip has a *state* (x, y, θ) that specifies its (x, y) position on the page and an angular direction θ. This is the direction in which it will draw the next line segment (see below). The pen state is initialized to be (0,0,0), where $\theta = 0$ is in the direction of the x axis.

Let's say there are five commands:

- `D` - draws a unit line segment from the current position and in the direction of θ, that it, it draws it from (x_0, y_0) to $(x_0 + \cos\theta, y_0 + \sin\theta)$.
- `L` - turns left (counter-clockwise) by 45 degrees
- `R` - turns right (clockwise) by 45 degrees
- `[` - pushes the current state onto the stack
- `]` - pops the stack, and current state ← popped state

See the slides for a few examples.

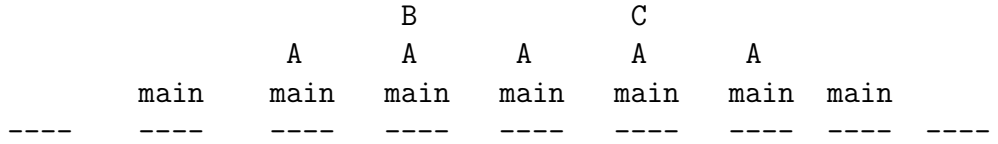
Example 6: the "call stack"

We have been discussing stacks of things. One can also have a stacks of tasks.[7] Imagine you are sitting at your desk getting some work done (main task). Someone knocks on your door and you let them in and chat (task A). While chatting, the phone rings and you answer it (task B). You finish the phone conversation and go back to the person in your office (A). Then maybe there is another interruption (C) which you take care of, return to A and return to main.

A similar stack of tasks occurs when a computer program runs. Say we have a `main` method where the program starts. The main method typically has instructions that cause other methods to be called. The program "jumps" to these methods, executes them and returns to the main method. Sometimes these methods themselves call other methods, and so the program jumps to these other methods, executes them, returns to the calling method, which finishes, and then returns to main.

A natural way to think of jumping from method to method is in terms of a stack. Suppose `main` calls method `mA` which calls method `mB`, and then when `mB` returns, `mA` calls `mC`, which eventually returns to `mA`, which eventually returns to `main` which then finishes. (See slides for this lecture.)

[7] though typically we don't call them "stacks", that is what they are.



Why do we need to use a stack? Can't you just jump from method to method as the program runs? No, that won't work. The problem is that when a method finishes and returns, it needs to remember where to return to. This information (the "return address") is part of the bundle of information (called a stack frame) that is thrown on the stack.[8]

Data structure for a stack

Finally, ... what is a good data structure for a stack? A stack is a list, so its natural to use one of the list data structures we have considered.

If you use a singly linked list to implement a stack, then you should push and pop to/from the front of the list, not to/from the back. The reason is that removing (popping) from the back of a singly linked list is inefficient i.e. you need to walk through the entire list to find the node that points to the last element (which you are popping). For a doubly linked list, it doesn't matter whether you push/pop at the front (head) or at the back (tail) as long as you do one or the other.

If you use an array, then you should push and pop to/from the end of the list (indicated by index top or size-1). The reason is that if you add/remove from the front of an array that you need to shift all the other elements if you want to maintain the property that the top is at index 0.

[8] How *exactly* the call stack and the "return address" work is a much more advanced topic which you will learn about properly in COMP 273 or in ECSE 221.

Queue

Let's now turn from stacks to queues. A queue is an ordered set of objects (a list) where the ordering is determined by *when* each object was inserted. With a stack, one accesses/removes the newest element (most recently added) whereas with a queue, one accesses/removes the oldest element (least recently inserted).

You are familiar with queues in daily life. You know that when you have a single resource such as a cashier in the cafeteria, you need to "join the end of the line" and the person at the front of the line is the one being served. The key property of a queue is that, among those objects/persons/etc currently in the queue, the one being served/removed is the one who first entered the queue.

The queue abstract data type (ADT) has two basic operations associated with it: `enqueue(e)` which adds an element to the queue, and `dequeue()` which removes an element from the queue. We could also have operations `isEmpty()` which checks if there is an element in the queue, and `peek()` which returns the first element in the queue (but does not remove it), and `size()` which returns the number of items in the queue. But these are not necessary for a core queue. [ASIDE: Java has a `Queue` interface, but uses the keyword `poll` instead of `dequeue` and `offer` instead of *enqueue*.]

Often one writes `add` and `remove` instead of `enqueue` and `dequeue`. Of course, these operations are very different for a queue than they are for a stack. Removing an element from a queue removes the least recently added element, whereas removing an element from a stack removes the most recently added. We say that queues implement "first come, first served" policy (also called FIFO, first in first out), whereas stacks implement a LIFO policy, namely last in, first out.

Example

Suppose we add (and remove) items `a,b,c,d,e,f,g` in the following order, shown on left. On the right is show the corresponding state of the queue *after* the operation.

OPERATION	STATE
	-
add(a)	a
add(b)	ab
remove()	b
add(c)	bc
add(d)	bcd
add(e)	bcde
remove()	cde
add(f)	cdef
remove()	def
add(g)	defg

Data structures for implementing a queue

Singly linked list

One way to implement a queue is with a singly linked list. Just as you join a line at the back, when you add an element to a singly linked list queue, you manipulate the `tail` reference. Similarly,

just as you serve the person at the front the queue, when you remove an item from a singly linked list queue, you manipulate the `head` reference. The `enqueue(E)` and `dequeue()` operations are equivalent to `addLast(E)` and `removeFirst()` operations from a singly linked list.

Array

Can we implement a queue using an array? Yes. But we need to be clever about it. Let's suppose there are `size` elements in the queue and the array has `length` slots. One *inefficient* way to use an array would be to enforce that the elements are in positions 0 to position `size-1`: When we remove an element, we would remove it from position 0 and when we add an element, we would add it at position `size` (and then increment `size`). Adding an element can be done in only a few operations (assuming `size < length`). The inefficiency comes when we remove an element. We remove from position 0, so when we remove we have to shift the remaining elements from positions 1 to `size-1` by one position, so they would go from positions 0 to `size-2`. This is obviously inefficient.

A second attempt is to relax the requirement that the front of the queue is at position 0. Instead of shifting when we dequeue, we just keep track of an index `head` (and `size`) where `head` is the index of the next item to be removed.[9] Note that, with this approach, both `add` and `remove` require only a few operations (independent of the length of the array). Below is the state of the array queue for the same example as above.

```
-------------------------------------------
0123456789.........       head    size

                            0       0
a                           0       1
ab                          0       2
 b                          1       1
 bc                         1       2
 bcd                        1       3
 bcde                       1       4
  cde                       2       3
  cdef                      2       4
   def                      3       3
   defg                     3       4
-------------------------------------------
```

The problem, of course, is that when `head + size >= length` then we will exceed the length of the array. Moreover, once we remove an element from the array, we never use that array position again. This is clearly inefficient.

Circular array

To take advantage of the empty positions, we treat the array as *circular*, so that the last array position (`length-1`) is followed by position 0. The next available position is thus `(head + size)`

[9] In the context of linked lists, `head` was a reference variable. In the context of arrays, we can treat `head` as an integer index.

`% length`. Note that this only holds when `size < length`. If this doesn't hold, i.e. if `size == length`, then the length of the array needs to be increased (see below) if we are to add another element. So the algorithm for adding an element would go like this:

```
enqueue( element ){    //  array implementation
   if ( size == length)
      increase length of array  //  *** SEE BELOW  **

   a[ (head + size) % length ] = element
   size = size + 1
}
```

The algorithm for dequeueing is simpler:

```
dequeue(){

   //  return the element at position "head % length"
   //  (but I leave out that code here)

   head = (head + 1) % length
   size = size - 1
}
```

Take the above example and suppose that the array has `length = 4`:

```
--------------------------
0123...     head   size

             0      0
a            0      1
ab           0      2
 b           1      1
 bc          1      2
 bcd         1      3
ebcd         1      4
e cd         2      3
efcd         2      4
ef d         3      3
efgd         3      4
--------------------------
```

At any time that `size` is 4, if we were to add another element then we would need to increase the length of the array.

Increasing the length of the (circular) array

To increase the length of the circular array, we create another array (say twice as big) and then copy the `length` elements to the new array. In the case of a queue, you need to be careful how you copy the elements, namely you need to copy the `head` element of the small array to position 0 in the new array, etc.

```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to length-1
   big[i] = small[ (head + i) %  small.length ]
```

Miscellaneous

At the end of this lecture, I briefly discussed two fun problems:

- a stack and queue problem in Exercises 2a, where you either use stacks to implement a queue or vice-versa (use queues to implement a stack)
- a fun problem of finding a moving point on an infinite 2D grid; this problem has nothing to do with queues, nor it related to induction and recursion (at least, not directly) which is the next general to topic of the course. Rather, the reason I am introducing this problem is that it is great practice for thinking about the finite and the infinite. Its also a terrific example of a complex problem that can be solved by coming up with a simple problem that is easier to think about. (If you missed this lecture, then no worries ... I won't be examining you on this fun problem.)

The next topic in the course is *recursion*. Since recursion is closely related to a proof technique in mathematics called *mathematical induction*, which many of you are familiar with, this is a good place to start.

Mathematical induction

Suppose we would like to prove some statement about the natural numbers (non-negative integers). For example we would like to prove: for any $n \geq 1$,

$$1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2}.$$

Here we are stating some *proposition* about the number n — call it $P(n)$ – and we want to prove the statement that $P(n)$ is true for all $n \geq 1, P(n)$".

The proof of the statement can be done using the technique of *mathematical induction*, which requires us to prove two things:

1. *base case*: the statement $P(n)$ is true for some $n = n_0$. In our example here, $n_0 = 1$.
2. *induction step*: for any $k \geq n_0$, *if* $P(k)$ is true, *then* $P(k+1)$ must also be true.

   The statement $P(k)$ is called the "induction hypothesis".

The logic is that if you can show $P(n)$ is true for $n = n_0$ (base case), then the induction step would imply that the statement is true for $n = n_0 + 1$ (since the induction hypothesis holds for $n = n_0$ which is the base case), which in turn implies that it must be true for $n = n_0 + 2$, and so on.

Example 1

Prove the formula for the well-known *arithmetic series*:

$$1 + 2 + 3 + \cdots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

The base case is $n_0 = 1$. It is easy to prove:

$$\sum_{i=1}^{1} = \frac{1 \cdot (1+1)}{2} = 1.$$

We next prove the induction step. For any $k \geq 1$, we assume $P(k)$ is true. We want to show that $P(k+1)$ must also be true.

$$\begin{aligned}
\sum_{i=1}^{k+1} i &= (\sum_{i=1}^{k} i) + (k+1) \\
&= \frac{k(k+1)}{2} + (k+1), \text{ by induction hypothesis} \\
&= (k+1)(\frac{k}{2} + 1) \\
&= \frac{1}{2}(k+1)(k+2) \text{ whis proves the induction step!}
\end{aligned}$$

Example 2

Claim: for all $n \geq 1$,

$$1 + 3 + 5 + \cdots + (2n - 1) = n^2$$

Proof: The base case of $n_0 = 1$ is obvious, since there is only a single term on the left hand side, i.e. $1 = 1^2$. The induction hypothesis is the statement $P(k)$:

$$P(k) \equiv \text{" } 1 + 3 + 5 + \cdots + (2k - 1) = k^2 \text{ "}$$

To prove the induction step, we hypothesize that $P(k)$ is true, and we show that it follows that $P(k + 1)$ must also be true.

$$\begin{aligned}
\sum_{i=1}^{k+1}(2i - 1) &= 2(k + 1) - 1 + \sum_{i=1}^{k}(2i - 1) \\
&= 2(k + 1) - 1 + k^2, \quad \text{by the induction hypothesis} \\
&= 2k + 1 + k^2 \\
&= (k + 1)^2.
\end{aligned}$$

Thus, the induction step is proved.

Example 3

Claim: for all $n \geq 3,\ 2n + 1 < 2^n$.

The induction hypothesis $P(k)$ is the statement "$2k+1 < 2^k$". Note that this statement is not true for $k = 1, 2$.

Proof:
The base case $n_0 = 3$ is easy to prove, i.e. $7 < 8$.
Next we prove the induction step. Let k be some integer such that $k \geq n_0 = 3$. Suppose $P(k)$ is true. We want to show that it follows that $P(k + 1)$ is true. We write $P(k + 1)$ on the left side:

$$\begin{aligned}
2(k + 1) + 1 &= 2k + 3 \\
&= (2k + 1) + 2 \\
&< 2^k + 2, \quad \text{by induction hypothesis} \\
&< 2^k + 2^k \\
&< 2^{k+1} .
\end{aligned}$$

The induction step is proved.

Example 4

Claim: For all $n \geq 5$, $n^2 < 2^n$.

Proof:
The base case $n_0 = 5$ is easy to prove, i.e. $25 < 32$.
Next we prove the induction step. The induction hypothesis $P(k)$ is the statement "$k^2 < 2^k$." We show that if $P(k)$ is true, then $P(k+1)$ must also be true.

$$\begin{aligned}(k+1)^2 &= k^2 + 2k + 1 \\ &< 2^k + 2k + 1, \quad \text{by induction hypothesis} \\ &< 2^k + 2^k, \quad \text{from Example 3} \\ &= 2^{k+1}\end{aligned}$$

which proves the induction step.

Example 5: upper bound on Fibonacci numbers

Consider the Fibonnacci[10] sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ... where

$$F(0) = 0, \qquad F(1) = 1,$$

and, for all $n > 2$, we define $F(n)$ by

$$F(n) \equiv F(n-1) + F(n-2).$$

Claim:

$$\text{for all } n > 0, \quad F(n) < 2^n.$$

Proof:

We take the base case to be $n_0 = 1$. Note $F(0) = 0, F(1) = 1$ and so (*) holds for the base case since $F(0) = 0 < 2^0$ and $F(1) = 1 < 2^1$.

The induction hypothesis $P(k)$ is that $F(k) < 2^k$. For the induction step, we assume that $P(k)$ is true and we show $P(k+1)$ must also be true.

$$\begin{aligned}F(k+1) &\equiv F(k) + F(k-1) \\ &< 2^k + 2^{k-1} \quad \text{by induction hypothesis} \\ &< 2^k + 2^k \\ &= 2^{k+1}\end{aligned}$$

and so the induction step is proven.

[10] `http://en.wikipedia.org/wiki/Fibonacci_number`

Induction and Recursion

Induction is used to prove many such statements in mathematics. It is also used to prove the correctness of certain computer programs. For example, "n factorial" is defined

$$n! = 1 \cdot 2 \cdot 3 \ldots (n-1) \cdot n$$

Here is an algorithm (written in Java) for computing it. The algorithm just applies the definition so there is nothing to prove about correctness.

```
int factorial(int n){  //  assume  n >= 1
   int result = 1;
   for (int i = 1;  i <= n;  i++)
       result *= i;
   return result;
}
```

But here is another way to define $n!$ which is more subtle, namely if $n > 1$, then

$$n! = n \cdot (n-1)!$$

The corresponding algorithm (written in Java) is:

```
int factorial(int n){   //   algorithm assumes argument:  n >= 1
   if  (n == 1)
     return 1;
   else
      return  n * factorial( n - 1);
}
```

Notice that the definition of the algorithm `factorial` involves a call to itself. Such an algorithm is said to be *recursive*.

Claim: The recursive algorithm `factorial(n)` computes $n!$ for any input value $n \geq 1$.

Proof:

First, the base case: If the parameter `n` is 1, then the algorithm returns 1, so the base case is true.

Second, the induction step: Suppose that the algorithm returns $k!$. (the induction hypothesis). We show that it follows that the algorithm returns $(k+1)$! when input argument is $n = k+1$. But this is easy, since when the argument is $k+1$, the algorithm returns $(k+1) * k!$ which is just $(k+1)!$.

Recursion

factorial

Last lecture, we saw an algorithm for computing the factorial function. There were two ways to compute it. One used a `for` loop[11] and the other used recursion.

Let's write the recursive version a bit differently, by inserting a tmp variable to store the returned result of the recursive call. Its not necessary to do this, but it makes it a bit easier to see what's going on.

```
int factorial(int n){   //   algorithm assumes argument:  n >= 1
   int factn = 0
   if  (n == 1)
     return 1;
   else
      factn = n * factorial( n - 1);
      return  factn
}
```

Each time a method calls another method (or a method calls itself, in the case of recursion), the computer needs to do some administration to keep track of the "state" of method at the time of the call. This information is called a "stack frame". Suppose we call `factorial(6)`, that is, we want to compute "6!". Then this leads to a sequence of calls and subsequent returns from these calls. For example, right before returning from the `factorial(3)` call, we have made the following sequence of calls and returns:

```
factorial(6)
  factorial(5)
    factorial(4)
      factorial(3)
        factorial(2)
          factorial(1)
          return from factorial(1)
        return from factorial(2)
```

the stack looks like this,

```
frame for factorial(3):  [factn = 6, n=3] <---- top of stack
frame for factorial(4):  [factn = 0, n=4]
frame for factorial(5):  [factn = 0, n=5]
frame for factorial(6):  [factn = 0, n=6] <---- bottom of stack
```

You can inspect the stack frame with the Eclipse debugger. I strongly recommend that you do this for a simple example like factorial so that you can see for yourself how this works.

[11] not discussed in class, but it was mentioned in notes

There are many other problems for which there is a non-recursive and a recursive solution. Sometimes the recursive way is more natural for expressing what you want to do, and other times the non-recursive way is more natural. Another issue is the computational cost. For factorial, the costs of the two algorithms are similar in that both run in time proportional to n. Let's now look at another problem in which the non-recursive vs. recursive solutions have quite different time costs.

Fibonacci numbers

Consider the Fibonnacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F(n) = F(n-1) + F(n-2),$$

where $F(0) = 0, F(1) = 1$. The standard way to calculate the Fibonacci numbers is just to iterate, starting at $n = 0$. Suppose you wanted the n^{th} Fibonacci number, where $n > 0$.

```
ALGORITHM:  fib(n)

   if ((n == 0) | (n == 1))
      return n
   else{
      f0 = 0
      f1 = 1
      for i = 2 to n{
         f2 = f1 + f0
         f1 = f2             //  set  F(n+1) for next round
         f0 = f1             //  set  F(n) for next round
      }
      return f2
   }
```

The method requires n passes through the "for loop". Each pass takes a small (fixed) number of operations. So we would expect the number of steps to be about cn for some constant c.

A *recursive algorithm* for computing the nth Fibonacci number goes like this:

```
Algorithm:  fib(n)    //   assume n > 0
//  Input:  the index of the Fibonacci number to be computed
//  Output:  the n-th fibonacci number
//
   if ((n == 0) || (n == 1))
      return n
   else
      return fib(n-1) + fib(n-2)
```

The trouble with this algorithm is that you end up calling `fib` on the same parameter *many* times. For example, suppose you are asked to compute the 247-th Fibonacci number. `fib(247)` calls `fib(246)` and `fib(245)`, and `fib(246)` calls `fib(245)` and `fib(244)`. But now notice that

`fib(245)` is called twice. Similar redundancies occur each step of the way until you reach `fib(1)` and `fib(2)`. As we will see a few lectures from now, the number of steps required in the computational grows like 2^n, which takes alot of longer than the iterative (for loop) algorithm !

Tower of Hanoi

Let's now turn to an example in which recursion allows us to express a solution in a very simple manner. Unlike in the previous example, the issue here is expressibility, rather than computational efficiency. (The algorithm here still takes a long time for large n, as we'll see a few lectures from now.)

The problem is called *Tower of Hanoi*. There are three stacks (towers) and a number n of disks of different radii. (See `http://en.wikipedia.org/wiki/Tower_of_Hanoi`). We start with the disks all on one stack, say stack 1, such that the size of disks on each stack decreases from bottom to top. The objective is to move the disks from the starting stack (1) to one of the other two stacks, say 2, while obeying the following rules:

1. A larger disk cannot be on top of a smaller disk (at any time).
2. Each move consists of popping a disk from one stack and pushing it onto another stack, or more intuitively, taking the disk at the top of one stack and putting it on another stack.

The (remarkably simple) recursive algorithm for solving the problem goes as follows. The three stacks are labelled $s1, s2, s3$. One of the stacks is where the disks "start". Another stack is where the disks should all be at the "finish". The third stack is the only remaining one.

```
tower(n, start, finish, other)
  if n>0 then
    tower(n-1, start, other, finish)
    move from start to finish          // i.e.  finish.push(  start.pop() )
    tower(n-1, other, finish, start)
  end if
```

For example, `tower(1,s1,s2,s3)` would produce to the following sequence of instructions:

```
tower(0,s1,s3,s2)       //  don't do anything
move from s1 to s3
tower(0,3,2,1)          //  don't do anything
```

The two calls `tower(0,*,*,*)` would do nothing since the condition $n > 0$ is not met.

```
move from 1 to 2
```

What about `tower(2,1,2,3)` ? This would produce the following sequence of instructions:

```
tower(1,1,3,2)
move from 1 to 2
tower(1,3,2,1)
```

and the two calls `tower(1,*,*,*)` would each move one disk, similar to the previous example (but with different parameters). So, in total there would be 3 moves:

```
move from 1 to 3
move from 1 to 2
move from 3 to 2
```

Here are the states of the tower for `tower(3,1,2,3)` and the corresponding print instructions. Notice that we need to do the following:

```
tower(2,1,3,2)
move from 1 to 2
tower(2,3,2,1)
```

So first we do `tower(2,1,3,2)`:

```
    *
    **
    ***
    ---       ---      ---    (initial)


    **
    ***       *
    ---       ---      ---     (after moving disk from 1 to 2)



    ***       *        **      (after moving disk from 1 to 3)
    ---       ---      ---


                       *
    ***                **      (after moving from 2 to 3)
    ---       ---      ---
```

```
which completes the tower(2, 1, 3, 2) call.
Next we do "move from 1 to 2":
```

```
                       *
              ***      **
    ---       ---      ---     (after moving from 1 to 2)
```

```
Then we call  tower(2, 3, 2, 1)

   *        ***       **
   ---      ---       ---     (after moving from 3 to 1)


            **
   *        ***
   ---      ---       ---     (after moving from 3 to 2)


             *
             **
             ***
   ---       ---      ---     (after moving from 1 to 2)

and we are done!
```

For any $n \geq 0$, towers of Hanoi algorithm is correct for n disks

For the algorithm to be "correct", we need to ensure that a larger disk is never place on top of a smaller disk, and that the n disks are moved from the start tour to the finish tour. The proof is by induction.
Base case: The rule is obviously obeyed if $n = 1$ and the algorithm simply moves the one disk from `start` to `finish`.
Induction step: Suppose the algorithm is correct if there are $n = k$ disks *in the initial tower*. This is the induction hypothesis. We need to show that the algorithm is therefore correct if there are $n = k + 1$ disks *in the initial tower*. For $n = k + 1$, the algorithm has three steps, namely,

- `tower(k,start,other,finish))`
- `move from start to finish`
- `tower(k,other,finish,start)`

The first recursive call to `tower` moves k disks from *start* to *other*, while obeying the rule for these k disks (by the induction hypothesis). The second step moves the biggest disk $(k + 1)$ from `start` to `finish`. This also obeys the rule, since `finish` does not contain any of the k smaller disks (because these smaller disks were all moved to the `other` tower). Finally, the second recursive call to `tower` move k disks from `other` to `finish`, while obeying the smaller-on-bigger rule (by the induction hypothesis). This completes the proof.

In the slides, I also illustrate the time required by the algorithm. I won't present it here. We'll see it again a few lectures from now when we look at recurrences.

Today we will look at three algorithms that have a similar behavior, namely they all require time proportional to $\log n$ to compute. I am presenting these algorithms within the broad topic of recursion, though some of the algorithms are expressed just as easily without recursion.

Twenty questions (with numbers), and decimal-to-binary conversion

Let's play a game. I am thinking of a number n between 0 and $2^{20} - 1$ (inclusive). Your task is find this number, by asking a sequence of 20 questions that have yes/no answers. How would you do it? Easy. Your i^{th} question is, "does bit b_i of the binary representation of n have value 1 ?" Once I give you all the bits, you can convert from binary to decimal and get the number n.

We have seen the algorithm for converting a decimal number $n \geq 1$ to binary. We can rewrite this algorithm so that it is recursive.

Algorithm: DecimalToBinary(n)
Input: a decimal number n
Output: bit sequence $b_0, b_1, \ldots$ representing n in binary
  if $n \geq 1$ **then**
    print $n\%2$
    DecimalToBinary(n / 2)
  end if

Note that the above algorithm prints $b_0, b_1, \cdots, b_{m-1}$ in that order. If we were to write the swap the order of the two instructions, then the bits would be output in the opposite order.

Algorithm: DecimalToBinary(n)
Input: a decimal number n
Output: bit sequence $b_{m-1}, b_{m-2}, \ldots, b_1, b_0$ representing n in binary
  if $n \geq 1$ **then**
    DecimalToBinary(n / 2)
    print $n\%2$
  end if

How does the number of bits depend on the n ? Let m be the number of bits needed to represent n, that is,

$$n = b_{m-1}2^{m-1} + b_{m-2}2^{m-2} + \ldots b_1 2 + b_0$$

where $b_{m-1} = 1$, that is, no leading 0's. Since each of the other b_i's is either 0 or 1, we have

$$\begin{aligned} n &\leq 2^{m-1} + 2^{m-2} + \cdots + 2 + 1 \\ &= 2^m - 1 \\ &< 2^m \end{aligned}$$

Taking the log (base 2) of both sides gives:

$$\log n < m$$

Moreover, since $b_{m-1} = 1$, we also have a lower bound

$$n \geq 2^{m-1}$$

and so

$$\log n >= m - 1.$$

Noting that m is an integer, it follows immediately that $m - 1 = floor(\log_2 n)$. Thus, the number of bits in the binary representation of n is always:

$$m = floor(\log_2 n) + 1.$$

Power (x^n)

Our next example looks very different at first glance, but we will see that it is in fact very similar. Consider an algorithm for computing `x` raised to some power `n`. An iterative (non-recursive) method for doing it is:

Algorithm: Power(x,n)
Input: a real number x and a positive integer n
Output: x^n

```
result ← 1
for i = 1 to n do
    result ← result * x
end for
return result
```

A faster way to compute x^n is to use recursion. For example, suppose we wish to compute x^{18}. We can write

$$x^{18} = x^9 * x^9$$

So, to evaluate x^{18}, we could evaluate x^9 and then perform *only one more* multiplication, i.e. $x^9 * x^9$. But how do we evaluate x^9 ? Because 9 is odd, we cannot do the same trick exactly. Instead we compute

$$x^9 = (x^4)^2 * x.$$

Thus, *once we have* x^4, two further multiplications are required.

Breaking it down again, we write $x^4 = (x^2)^2$ and so we have

$$x^{18} = (((x^2)^2)^2 * x)^2.$$

Thus we see that we need a total of 5 multiplications (4 squares and one multiplication by x).

Notice that the number of recursive calls will be about $\log n$, for the same reason as in the decimal-to-binary algorithm, namely each call divides n by 2 and so the number of calls is the number of times you can divide the original n by 2 until you get to 0.

The number of multiplications that is executed for each recursive call will be either 1 or 2, depending on whether the n parameter passed in that call is even or odd. For example, if the

Algorithm: power(x,n) – recursive
Input: a real number x and an integer $n \geq 0$
Output: x^n

```
if n = 0 then
  return 1
else
  tmp ← power(x, n/2)
  if n is even then
    return tmp * tmp
  else
    return tmp * tmp * x
  end if
end if
```

original n's binary representation has all 1's, e.g. $63 = (111111)_{two}$, then two multiplications will be executed at each recursive call since the parameter n will always be odd.

In class, I gave a demo using the Eclipse debugger where I examined the stack frame on the top of the call stack, just before the return from each recursive call. You can find the code here.

`http://www.cim.mcgill.ca/~langer/250/code/TestPower.java`

You should definitely run this code using Eclipse (or NetBeans or some other powerful IDE). Set two breakpoints: the first at the line where you enter the `power` method and the second at the line where the `power` method returns the value. You can examine the stack frames as they are pushed and popped. Look how the values in the various stack frames change. (I use F11 to launch the debugger and F8 to resume execution after a break.)

If you can draw the stack with pencil and paper as it evolves over time, then you understand it. Remember: a stack frame is created whenever a method is called. (This doesn't just happen in recursion. It happens whenever you have methods, including `main` which indeed defines the frame on the bottom of the stack.)

Binary search in a sorted array

Let's now look at a third problem in which this $\log n$ behavior appears. Suppose we have an *array* of n elements which are sorted from smallest to largest. These could be numbers or strings sorted alphabetically as in last names in a phone book. Now we would like to search for a particular element (number or string) and return the index in $0, .dots, n-1$ of that value in the array. If that value is not present in the array, then we return the index -1.

One way to do this would be to scan the values in the array, using say a `while` loop. In the worst case that the value that we are searching for is the last one in the array, we would need to scan the entire array to find it. This would take n steps. Such a method is called *linear search.*

A much faster way to search takes advantage of the ordering of the elements. You are familiar with this idea. Think of when you look up a phone number in a telephone book. You don't start from the first page and scan. Instead, you pick a page somewhere in the middle. If the name you are looking for comes before those on the page, then you continue your search in only pages that come before the chosen one, otherwise you continue your search in the pages that come after the

chosen one. The *binary search* algorithm is similar to this.

Algorithm: **binarySearch**$(a, v, low, high)$
Input: array a, value v, lower and upper bound indices $low, high$ ($low = 0, high = n - 1$ initially)
Output: the index i of element v (if it is present), -1 (if v is not present)

```
if low == high then
  if a[low] == v then
    return low
  else
    return −1
  end if
else
  mid ← (low + high)/2
  if v ≤ a[mid] then
    return binarySearch(a, v, low, mid )
  else
    return binarySearch(a, v, mid + 1, high)
  end if
end if
```

Each time the recursion is called, the number of elements in the array that still need to be examined is cut approximately in half. I say "approximately" because if $[low, high]$ has an odd number of elements, then we cannot cut this odd number exactly in half. (Note: $(low + high)/2$ is an integer division, and so the remainder is ignored.) For an input array with n elements, there are approximately $\log_2 n$ recursive calls.

Several observations can be made. First, if $a[mid] == v$ before $low == high$, then the algorithm keeps going recursively, even though we have found the element.[12] Second, if the element appears more than once in the array, then algorithm will return the index of one of elements but not all.

Here is a small example to illustrate the low (L) and high (H) settings for each recursive call. The columns indicate successive calls and the index of L and H.

```
  0    L
  1
  2
  3        L   L
* 4            H  L=H
  5    H   H
```

If the original number of elements is a power of 2, then the search can be interpreted in terms of the binary representation of the indices, namely each decision of which recursive call to make

[12] This seems inefficient, at first glance. However, it is not so bad. In order to make the algorithm stop when $a[mid] == v$, you would need to test for this explicitly. And you would need to do this explicit test every time the recursion is called. This would be extra work and would cancel out (to some extent) the savings you sometimes would get by stopping the recursion early.

amounts to choosing the next bit of the binary representation of the index (from most significant to least significant).

In the example below, suppose $n = 16$ and the item we are searching for happens to be at index 5. The succession of $low, high$ values will be (0,15), (0,7), (4,7), (4,5), (5,5). See figure below, with the low and high values marked `L` and `H`. In the first call, you determine that $b_3 = 0$. In the second call you determine that $b_2 = 1$. In the third call, you determine that $b_1 = 0$. And the final call determines that bit $b_0 = 1$.

```
   0  0000   L   L
   1  0001
   2  0010
   3  0011
   4  0100           L   L
*  5  0101               H    L=H
   6  0110
   7  0111       H   H
   8  1000
   9  1001
  10  1010
  11  1011
  12  1100
  13  1101
  14  1110
  15  1111   H
```

I emphasize that this interpretation of the algorithm only makes sense if the original n is a power of 2.

Mergesort

In lecture 3, we saw the "insertion sort" algorithm for sorting n items. We saw that, in the worst case, this algorithm requires $\frac{n(n-1)}{2}$ or $\frac{n^2}{2} - \frac{n}{2}$ or operations. As discussed in the lecture slides, n^2 can be very prohibitively large if the number of items to be sorted is too large. e.g. if $n = 2^{20} \approx 10^6$, then $n^2 \approx 10^{12}$. Today's machines run at about 10^9 operations per second (i.e. GHz), and so this means thousands of seconds to sort such a list (worst case).

We now consider an alternative sorting algorithm that runs much faster in the worst case. This algorithm is called *mergesort*. Here is the idea. If there is just one number to sort ($n = 1$), then do nothing. Otherwise, partition the list of n elements into two lists of size about $n/2$ elements each, sort the two individual lists (recurively, using mergesort), and then merge the two sorted lists.

For example, suppose we have a list

$$< 8, 10, 3, 11, 6, 1, 9, 7, 13, 2, 5, 4, 12 > .$$

We partition it into two lists

$$< 8, 10, 3, 11, 6, 1 > \quad < 9, 7, 13, 2, 5, 4, 12 > .$$

and sort these (by applying mergesort recursively):

$$< 1, 3, 6, 8, 10, 11 > \quad < 2, 4, 5, 7, 9, 12, 13 > .$$

Then, we merge these two lists

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13.$$

Algorithm: **mergesort**(list)
Input: list of elements that can be indexed by position
Output: Sorted list

```
if (list.length = 1) then
   return list
else
   mid ← (list.size -1) / 2
   l1 ← list.getElements(0,mid)
   l2 ← list.getElements(mid+1,list.size-1)
   l1 ← mergesort(l1)
   l2 ← mergesort(l2)
   return merge( l1, l2 )
end if
```

Algorithm: **merge**$(l1, l2)$
Input: Sorted sequences l1 and l2
Output: Sorted sequence l containing the elements from l1 and l2

```
initialize empty list l
while l1 is not empty & l2 is not empty do
  if l1.first < l2.first then
    l.addlast( l1.remove(l1.first))
  else
    l.addlast( l2.remove(l2.first))
  end if
end while
while l1 is not empty do
  l.addlast( l1.remove(l1.first))
end while
while l2 is not empty do
  l.addlast( l2.remove(l2.first))
end while
return l
```

Note that I have written the algorithm using *abstract List operations only.* This has the advantage of getting you quickly to the main ideas of the algorithm: what is being computed and in which sequence. However, be aware that it has the disadvantage of hiding the "implementation" details, by which I mean the data structures [13] you use. As we have seen, sometimes the choice of data structure can be important. For example, compare an array versus a linked list implementation. The call `getElements()` is very different for these data structures. For the array, the partitioning into two lists could be done just by computing indices which could be passed as extra parameters to the *mergesort* calls. To do the merge, the most obvious way would be to copy the elements to a second array. There is a clever way to organize these partitions and copies which allows you to just use one extra array (of the same size n as the original one). But the details on how to do are not what I want to emphasize now, since they would obscure the more abstract ideas of the algorithm. So I have left them out. Similarly, for a linked list implementation, there are details that one needs to address. One example is that the partitioning of a list into two requires you to split the list at the middle. But with a linked list, you don't have immediate access to the middle element. To find it, you have to scan for it by traversing the list from the beginning.

mergesort and the call stack

In the lecture slides, I went over al example with an set of elaborate figures showing how the partitions and merges are done. I also discussed the sequence of calls that are made to mergesort and merge and how the call stack evolves. I am not going to attempt to describe that example here. You should instead see the slides. Try to understand how the ordering of the calls (in red) is

[13] rather than code in some programming language code

determined, and why the call stack evolves as I have drawn it. If you can do so, then you understand mergesort very well.

"mergesort is $n \log n$"

Another point to note is that there are $\log n$ levels of the recursion, namely the number of levels is the number of times that you can divide n by 2 until you reach 1 element per list. As we will discuss a few lectures from now, the mergesort algorithm requires about $n \log n$ steps, namely at each of the $\log n$ levels of the recursion, the total number of operations you need to do is proportional to n.

To appreciate the difference between the worst case number of operations for insertion sort (say n^2) versus the worst case[14] of mergeshort ($n \log n$), consider the following table.

n	$\log n$	$n \log n$	n^2
$10^3 \approx 2^{10}$	10	10^4	10^6
$10^6 \approx 2^{20}$	20	20×10^6	10^{12}
$10^9 \approx 2^{30}$	30	30×10^9	10^{18}
...	...	...	...

Thus, the time it takes to run mergesort becomes significantly less than the time it takes to run insertion sort, as n becomes large. Very roughly speaking, on a computer that runs 10^9 operations per second (which is typical these days), running mergesort on a problem of size $n = 10^9$ would take in the order of minutes, whereas running insertion sort would take centuries.

[14] mergesort always takes $cn \log n$ operations, i.e. best case = worst case

big O (informal)

In the next several lectures, we will study how the number of operations performed by various algorithms grows with n which is the size of the input. We have seen several examples of this growth.

- addition grows linearly with the number of digits in the two numbers (assuming they have the same number of digits). We say that the addition algorithm is $O(n)$.
- Multiplication grows with the square of the number of digits. We say this algorithm is $O(n^2)$. The reason it is n^2 is that the algorithm involves the single digit products of each possible *pair* of digits in the first and second numbers and there are n^2 pairs.
- Insertion sort (worst case) grew as n^2 so we say it is $O(n^2)$. The reason it is n^2 is that the algorithm involves two loops, one nested inside the other, which leads to 1 + 2 + ... + n operations.
- Insertion sort (best case) grew as n so we say it is $O(n)$. The best case occurs if the list is already sorted. In this case the inner loop only executes a small number of operations (not dependent on n).
- Mergesort is $O(n \log n)$. The merge steps at each "level" involve a total of n operations and there are $\log n$ levels, since the number of levels is the number of times we divide n by 2 before we reach 0. (It is more challenging to visualize this.)
- Binary search is $O(\log n)$. It involves $\log n$ recursive calls (for the same reason mergesort has $\log n$ levels).

A few lectures from now, we will be more formal about big O. We will talk about concrete functions $t(n)$ and $g(n)$ and we will say what it means for $t(n)$ to be $O(\ g(n)\ )$. Intuitively, it means that $t(n)$ grows at a rate whose dependence on n is no faster than the rate of growth of $g(n)$.

Before we can understand the more formal definition of "big O", we need more familiarity with various functions and their rates of growth. In particular, we will try to characterize the time it takes for various recursive algorithms to run.

Recurrences

We have seen many algorithms thus far, and for each one we have tried to express how many operations $t(n)$ are required as a function of some parameter n which is typically the "size" of the problem. For algorithms that involve `for` loops, it is straightforward to write the number of operations of the loop component as a polynomial, whose degree is the number of nested loops. For example, if we have two nested `for` loops, each of which run n times, then these loops take time proportional to n^2. Insertion sort was an example.

For recursive algorithms, it is less obvious how to express $t(n)$. Today, we will look at a number of examples. In each example, we express $t(n)$ in terms of $t(*)$ where the argument depends on n but it is a value smaller than n. Such a recursive definition of $t(n)$ is called a *recurrence relation.*

Example 1: Factorial

Let $t(n)$ be the time it takes to compute $n!$. You should have an intuition that $t(n)$ is $O(n)$. This is easy to see if you use an iterative algorithm to compute $n!$ since we have a `for` loop which we iterate n times. What about if we use a recursive algorithm?

Algorithm: factorial(n)
if $n > 0$ **then**
return $n*$ **factorial**$(n-1)$
end if

The recursive algorithm for computing $n!$ involves a method call, and a multiplication. These operations can each be done in constant time. Moreover, each method call reduces the problem from size n to size $n-1$. This suggests a relationship:

$$t(n) = c + t(n-1)$$

namely the time $t(n)$ that it takes to compute $n!$ is some constant plus the time it takes to compute $(n-1)!$.

To obtain an expression for $t(n)$ that is not recursive, we repeatedly substitute on the right side, as follows:

$$\begin{aligned} t(n) &= c + t(n-1) \\ &= c + c + t(n-2) \\ &= c + c + c + t(n-3) \\ &= \dots \\ &= nc + t(0). \end{aligned}$$

This method is called *backwards substitution* because we substitute starting at $t(n)$ and work our way back to $t(0)$. Note $t(0)$ is the base case of the recursion and done in constant time c_0. We would conclude that $t(n)$ is $O(n)$ since it grows no quicker than n.

One often writes such a recurrence in a slightly simpler way ($c = 1$):

$$t(n) = 1 + t(n-1) \ .$$

The idea is that since the constant c has no "units" anyhow, its meaning is unspecified except for the fact that it is constant, so we just treat it as a unit (1) number of instructions.

Example 2: Tower of Hanoi

A more interesting example is the Tower of Hanoi problem. Let $t(n)$ be the number of disk "moves". The recurrence relation is:

$$t(n) = 1 + 2\ t(n-1)$$

and $t(0) = 0$ since there is nothing to do when there are no disks to move. The "1" on the right side refers to the unit of work that is done with each call to `tower`. There is the single disk move

that is done in each call, and there is also some administrative work that is done namely creating two stack frames. All this constant time work is bundled together as a single unit. This work is added to a term 2 $t(n-1)$ which is the time needed for the *two* recursive calls on the problem of size $n-1$.

Proceeding by back substitution, we get

$$\begin{aligned} t(n) &= 1 + 2t(n-1) \\ &= 1 + 2(1 + 2t(n-2)) \\ &= 1 + 2 + 4t(n-2) \\ &= 1 + 2 + 4(1 + 2t(n-3)) \\ &= 1 + 2 + 4 + 8t(n-3) \\ &= 1 + 2 + 4 + 8 + \cdots + 2^{n-1} + 2^n t(0) \\ &= 2^n - 1 + 2^n t(0), \qquad (*) \\ &= 2^n - 1, \qquad \text{since } t(0) = 0 \end{aligned}$$

where (*) used the familiar geometric series

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

for the case that $x = 2$.

Since $t(n) = 2^n - 1$, we say that $t(n)$ is $O(2^n)$. Notice that this analysis ignores many of the details of what happens within the Tower of Hanoi solution. It ignores the specific cost of "moving a disk", or printing out a statement that we are moving a disk, or what is involved in making stack frames. These details don't matter in this analysis since they are just constant factors and what we are caring about here is the dependence on n. [Note: the constant factors do matter for some other types of analysis. For sure! In particular, if you want to express the "run time" of some particular program in seconds on a particular computer, then yes you certainly do need to consider constant factors. But that is not our goal in this course.]

Example 3

You may be wondering whether or not you need the factor "2" in the recurrence equation. For example, in the factorial recurrence, I used $c = 1$ instead of the more general c and suggested it didn't matter. What happens if we drop the factor 2 in the recurrence of Towers of Hanoi, and instead consider

$$t(n) = 1 + t(n-1).$$

However, notice that this just the recurrence we saw with $n!$, which was $O(n)$ rather than $O(2^n)$. Quite a big difference! So clearly the factor 2 matters.

To see what's going on here, compare

$$t(n) = c + t(n-1)$$

versus

$$t(n) = c\, t(n-1).$$

The first recurrence says that it takes time c to reduce the size of the problem by 1. But since we haven't specified the units of time (seconds? milliseconds? multiplications?, additions? etc) we might as well just replace c by 1 and call it "one mystery unit" of time. The total number of steps behaves as cn or just n "mystery units" of time. The second recurrence is quite different. It says that we need to carry out c versions of the smaller problem when we reduce the problem size by 1. Here, the units of c are quite well specified, namely we need to do c recursive calls. The number of steps now behaves as c^n which is $O(c^n)$ rather than $O(n)$.

Example 4

Let's consider another recurrence:

$$t(n) = n + t(n-1)\ .$$

For example, consider a sorting algorithm that scans a list of size n, finds the minimum element, removes that element, and then (recursively) sorts the remaining list of size $n-1$.

How do you solve this recurrence? By backwards substitution, we get

$$\begin{aligned} t(n) &= n + t(n-1) \\ &= n + n - 1 + t(n-2) \\ &= \dots \\ &= n + n - 1 + n - 2\ + \cdots + 2 + 1, \text{where we assume } t(1) = 1 \\ &= \frac{n(n+1)}{2} \end{aligned}$$

which is $O(n^2)$.

Example 5

Here is a similar one:

$$t(n) = cn + t(n-1)\ .$$

$$\begin{aligned} t(n) &= cn + t(n-1) \\ &= cn + c(n-1) + t(n-2) \\ &= \dots \\ &= c(n + n - 1 + n - 2\ + \cdots + 2 + 1) \\ &= \frac{cn(n+1)}{2} \end{aligned}$$

which again is $O(n^2)$.

Example 6: binary search

Recall the binary search algorithm. We assume we have an ordered list of elements, and we would like to find a particular element `e` in the list. The algorithm computes the mid index and compares the element `e` to the element at that mid index. The algorithm then recursively calls itself, searching for `e` either in the lower or upper half of the list. So,

$$t(n) = c + t(n/2) \ .$$

To keep the analysis simple, let's suppose that n is a power of 2, namely $n = 2^k$, and proceed by back substitution.

$$\begin{aligned} t(n) &= c + t(n/2) \\ &= c + c + t(n/4) \\ &= c + c + \cdots + t(n/2^k) \\ &= c + c + \cdots + t(n/n) \\ &= c \log_2 n + t(1) \end{aligned}$$

In this lecture we will see several more examples of recurrences.

Example 7: power

Recall the recursive algorithm for computing x^n. At each level of the recursion, one either does two multiplications or one multiplication depending on whether the exponent argument at that level is odd or even respectively. This means that you cannot write the recurence as an equality. Instead you need to use an inequality, e.g.

$$t(n) \leq c + t(\frac{n}{2}).$$

You cannot solve this recurrence exactly, but instead you need to use an upper bound:

$$\begin{aligned} t(n) &\leq c + t(n/2) \\ &\leq c + c + t(n/4) \\ &\leq c \log_2 n + t(1) \end{aligned}$$

Again we would say $t(n)$ is $O(\log n)$ since big O is a statement about upper bounds. Next lecture we will be more formal about upper bounds.

Example 8 : matrix power

Let's extend the above example to another definition of "power". Suppose we have two 2×2 matrices A and B,

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

From basic linear algebra, the matrix product $A \cdot B$ is defined:

$$A \cdot B \equiv \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

It can be computed with 8 (scalar) multiplications and 4 (scalar) additions.

Define the power A^n of a matrix to be the product $A \cdot A \cdot \cdots \cdot A$.

$$A^1 = A, \qquad A^2 = A \cdot A, \quad etc.$$

An algorithm for computing power is shown on the next page.

We can define a recurrence relation for the time taken by the algorithm as follows:

$$t(n) = c + t(\frac{n}{2})$$

which is similar to the previous example except that the constant c would be bigger since it needs to account for more work to carry out a matrix multiplication. This extra work is still constant though (namely 8 scalar multiplications and 4 additions). Thus, the solution of the recurrence is the same as at the top of the page and so the algorithm is still $O(\log n)$.

Algorithm: **power**(A, n), where $n > 0$
Input: *Square matrix A and exponent $n > 0$*
Output: $A^n = A \cdot A \cdot A \cdot \cdots \cdot A$

```
if n = 1 then
  return A
else
  B ← power(A, n/2)
  if (n%2) = 1 then
    return B · B · A
  else
    return B · B
  end if
end if
```

Example 9: a $O(\log n)$ algorithm for computing $Fibonacci(n)$

We can obviously compute the n^{th} Fibonacci number in $O(n)$ time just by computing all the $F(k)$ iteratively for $k = 0$ up to $k = n$. Surprisingly, we can compute the n^{th} Fibonacci number much faster than that, namely there is a $O(\log n)$ algorithm for computing $F(n)$. Here is how we do it. Observe that, for all $n \geq 0$,

$$\begin{bmatrix} F(n+1) \\ F(n) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix}$$

We prove below (by induction) that

$$\begin{bmatrix} F(n+1) \\ F(n) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} F(1) \\ F(0) \end{bmatrix}. \qquad (*)$$

It follows that we can compute $F(n)$ and $F(n+1)$ in the time it takes us to compute A^n where A is any 2×2 matrix. Thus, we can compute $F(n)$ is time $t(n)$ that is $O(\log n)$.

Proof of (*): the base case $n = 1$ is trivial. For the induction step, assume that (*) holds for any $n = k \geq 1$ (the induction hypothesis), and show it must therefore be true for $n = k + 1$.

$$\begin{aligned} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{k+1} \begin{bmatrix} F(1) \\ F(0) \end{bmatrix} &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^k \begin{bmatrix} F(1) \\ F(0) \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F(k+1) \\ F(k) \end{bmatrix} \text{ by induction hypothesis} \\ &= \begin{bmatrix} F(k+2) \\ F(k+1) \end{bmatrix} \end{aligned}$$

which is what we wanted to prove, and so we are done.

Example 10: mergesort

Recall the mergesort algorithm for the case that n is a power of 2. In this algorithm, we divide the list of things to be sorted into two approximately equal size sublists, sort each of them, and then merge the result. Merging two sorted lists of size $\frac{n}{2}$ takes time proportional to n since (if you review the algorithm in detail) we have several loops, each of which removes one of the elements from the two lists.

Let's assume that n is a power of 2. This keeps the math simpler since we don't have to deal with the case that the two sublists are not exactly the same length. In this case, the recurrence relation for mergesort is therefore:

$$t(n) = 2t(\frac{n}{2}) + cn.$$

By backwards substitution,

$$\begin{aligned} t(n) &= 2t(\frac{n}{2}) + cn \\ &= 2(\ 2t(\frac{n}{4}) + c\frac{n}{2}) + cn \\ &= 4\ t(\frac{n}{4}) + cn + cn \\ &= 4(2t(\ n/8\ ) + c\frac{n}{4}) + cn + cn \\ &= 8t(\frac{n}{8}) + cn + cn + cn \\ &= \dots \\ &= n\ t(\frac{n}{n}) + cn \log n \\ &= n\ t(1) + cn \log n \end{aligned}$$

which is $O(n\ \log n)$. Notice that the bottleneck of the algorithm is the merging part, which takes a total of cn operations at each "level" of the recursion. There are $\log n$ levels.

Quicksort

Another well-known recursive algorithm for sorting a list is *quicksort*. (See next page for details.) At each call of quicksort, an element `e` is removed from the argument list. The remaining elements in the argument list are then partitioned into two sets, those smaller than `e` and those that are greater than or equal to `e`. These two lists are recursively sorted. The final sorted list by concatenating the two sorted lists with the element `e`, namely `e` is sandwiched between the list of elements that are less than `e` and the list of elements that are not less than `e`.

In the best case, each recursive call to quicksort partitions the remaining elements into two equal sized sets. This is the best case because then one only needs $\log n$ levels of the recursion and approximately the same recurrence as mergesort can be written and solved. (The only minor difference to consider is that one is partitioning a list of size $n-1$ rather than n, but this makes no difference to the $O(n \log n)$ behavior.)

In the worst case, each recursive call to quicksort partitions the remaining elements into two lists, *one of which is empty*. This can happen if the list happens to already be sorted already and if we remove the first element (as stated in the algorithm). In this case, we have $n-1$ recursive

calls, and the recurrence will be

$$t(n) = t(n-1) + cn$$

whose solution grows with n^2.

One practical way around this problem is to remove a randomly chosen element, rather than the first one. In this case, one can show (wait for COMP 251) that in the *average case*, quicksort is an $O(n \log n)$ algorithm. Proving average case behavior requires a greater set of tools than we have at present, so we don't do that here.

Algorithm: quicksort(list)
Input: list of elements that can be indexed by position
Output: Sorted list

```
if (list.length ≤ 1) then
  return list
else
  e ← list.removeFirst()
  l1 ← list.getElementsLessThan(e)
  l2 ← list.getElementsNotLessThan(e)
  l1 ← quicksort(l1)
  l2 ← quicksort(l2)
  return concatenate( l1, e, l2 )
end if
```

ASIDE: Why is quicksort called "quick" when its worst case depends on n^2? This would make it seem slower than mergesort which always runs in time $t(n)$ that is $O(n \log n)$. The quickness refers to the average case, not the worst case. One reason that quicksort is quick is that it can[15] be encoded so that it does not require extra space. By contrast, the obvious implementation of mergesort requires that you create a new array to merge the elements into. In practice, this tends to slow mergesort down a bit.[16]

Forward substitution

Although we will only use back substitution to solve recurrences in this course, you should be aware that there are other ways to solve recurrences. For example, the method of *forward substitution* starts with the base case and then works forward and tries to find the pattern. Take the tower of hanoi recurrence:

$$t(n) = 1 + 2t(n-1)$$

with $t(0) = 0$. We can start with the base case and substitute in, and work our way forward: $t(0) = 0,\ t(1) = 1,\ t(2) = 1 + 2 \cdot 1 = 3,\ \ t(3) = 1 + 2 \cdot t(2) = 7,\ t(4) = 1 + 2 \cdot t(3) = 15$. Then you can notice the pattern that $t(i)$ is $2^i - 1$. This leads you to guess the solution $t(n) = 2^n - 1$.

[15] without too much effort – see Prof. Precup's COMP 250 notes, in you are interested

[16] There are more clever ways to implement mergesort which make it run faster, but these methods are beyond the scope of the course. You can look them up if you are interested.

Once you guess what the solution of the recurrence is, you can try to prove it, for example by induction. The base case is true since $t(0) = 0$. Now assume $t(n) = 2^n - 1$ is true for $n = k$ and prove it for $n = k + 1$.

$$\begin{aligned} t(k+1) &= 1 + 2\ t(k) \\ &= 1 + 2 \cdot (2^k - 1) \ \text{ by the induction hypothesis,} \\ &= 2^{k+1} - 1 \end{aligned}$$

which proves the induction step, so we are done.

The trouble with forward substitution in practice is that it may be hard to guess at the answer, just by seeing the successive $t(n)$ values starting from 0.

We have seen several algorithms in the course, and loosely characterized the time it takes to run them in terms of the "size" n of the input. Let's now tighten up our analysis. We will study the behavior of algorithms by comparing the number of operations required – which is typically a complicated function of n of the input size n – against some simpler function of n. This simpler function describes either an upper bound, in a certain technical sense to be specified below.

The technical definition is reminiscent of the technical definition of a limit that some of you may have seen in Calculus (and that all of you have seen in Real Analysis if you are in Math). However, there are important differences. A "limits" describes an *asymptotic convergence* of a function (or asymptotic divergence, in the case that the limit is ∞). Big O describes an *asymptotic upper bound*.

Big O

Let $t(n)$ be a well-defined sequence of integers. This sequence $t(n)$ represents the "time" or number of steps it takes an algorithm to run as a function of some variable n which itself represents the "size" of the input. We will consider $n \geq 0$ and $t(n)$ to both be positive integers.

Let $g(n)$ be another well defined sequence of integers. We commonly consider $g(n)$ to be one of the following:

$$1, \log n,\ n,\ n \log n,\ n^2,\ n^3,\ 2^n, \ldots$$

We would like to say that $t(n)$ is bounded about by a simple $g(n)$ function if, for n sufficiently large, we have $t(n) \leq g(n)$. We would say that $t(n)$ is "asymptotically bounded above" by $g(n)$. Formally, let's say that $t(n)$ *is asymptotically bounded above by* $g(n)$ if there exists a positive number n_0 such that, for all $n \geq n_0$,

$$t(n) \ \leq g(n).$$

For example, consider the function $t(n) = 5 + 7n$. For n sufficiently large, $t(n) \leq 8n$. Thus, $t(n)$ is "asymptotically bounded above" by $g(n) = 8n$ in the sense that I just define. Notice that the constant 8 is arbitrary here. Any constant greater than 7 would do. For example, $t(n)$ is also "asymptotically bounded above" by $g(n) = 7.00001n$.

It is much more common and useful to talk about a asymptotic upper bounds on $t(n)$ in terms of a simpler function $g(n)$, namely where we don't have constants in the $g(n)$. To do this, one needs a slightly more complicated definition. This is the standard definition of an asymptotic upper bound:

Definition (big O): The sequence $t(n)$ is $O(g(n))$ if there exists two positive numbers n_0 and c such that, for all $n \geq n_0$,

$$t(n) \ \leq \ c\ g(n).$$

We say $t(n)$ is "big O of $g(n)$".

I emphasize that the condition $n \geq n_0$ allows us to ignore how $t(n)$ compares with $g(n)$ when n is small. In this sense, it describes an *asymptotic* upper bound.

Example 1

The function $t(n) = 5 + 7n$ is $O(n)$. To prove this, we write:

$$\begin{aligned} t(n) &= 5 + 7n \\ &\leq 5n + 7n, \quad n \geq 1 \\ &= 12n \end{aligned}$$

and so $n_0 = 1$ and $c = 12$ satisfies the definition. An alternative proof is:

$$\begin{aligned} t(n) &= 5 + 7n \\ &\leq n + 7n, \text{ for } n \geq 5 \\ &= 8n \end{aligned}$$

and so $n_0 = 5$ and $c = 8$ also satisfies the definition.

A few points to note:

- If you can show $t(n)$ is $O(g(n))$ using constants c, n_0, then you can always increase c and/or n_0 and be sure that these constants with satisfy the definition also. So, don't think of the c and n_0 as being uniquely defined.
- There are inequalities in the definition, e.g. $n \geq n_0$ and $t(n) \leq cg(n)$. Does it matter if the inequalities are strict or not? Not really. You can easily verify that, for any $t(n)$ and $g(n)$, the statement "$t(n)$ is $O(g(n))$" is true (or false) whether we a strict inequality or just "less than or equal to".
- We are looking for tight upper bounds for our $g(n)$. But the definition of big O doesn't require this. For example, in the above example, $t(n)$ is also $O(n^2)$ since $t(n) \leq 12n \leq 12n^2$ for $n \geq 1$. By a similar argument, $t(n)$ is also $O(n \log n)$ or $O(n^3)$, etc.

Many students write proofs that they think are correct, but in fact the "proofs" are incomplete (or wrong). For example, consider the following "proof" for the above example:

```
 5 + 7n   <  c n
5n + 7n   <  c n, n >= 1
   12n    <  c n

Thus,  c > 12,  n_0 = 1
```

There are several problems with this, as a formal proof.

- the first statement seems to assume what he is trying to prove; some indication should be given whether this first line is an assumption, or this is what he is going to show. And is the statement true for some c, or for all c, or for some n, or all n. Its just not clear.
- There is no clear connection between the statements. What implies what? Are the statements equivalent, etc ? *Such proofs may get grades of 0. This is not the "big O" you want.*

What would it mean to say $t(n)$ is $O(1)$, i.e. $g(n) = 1$? Applying the definition, it would mean that there exists a c and n_0 such that $t(n) \leq c$ for all $n \geq n_0$. That is, $t(n)$ is bounded by some constant.[17]

Finally, some students try to do big O proofs by using ideas from Calculus and taking "limits". There are situations in which this can be done. But generally you should avoid limits in proving statements about big O, unless you really know what you are doing (and have taken courses like MATH 242 Real Analysis, and so you know what a limit is, in a formal sense).

[17] Why would this be used? Sometimes when we analyze an algorithm's performance, we consider different parts of the algorithm separately. Some of those parts (such as assigning values to variables outside of any loops or any recursive calls) might be done once. Such parts take $O(1)$ time.

Example 2 (see a different example in slides)

The function $t(n) = 17 - 46n + 8n^2$ is $O(n^2)$. To prove this, we need to show there exists positive c and n_0 such that, for all $n \geq n_0$,

$$17 - 46n + 8n^2 \leq cn^2 \ .$$

$$\begin{aligned} t(n) &= 17 - 46n + 8n^2 \\ &\leq 17 + 8n^2, \quad n > 0 \\ &\leq 17n^2 + 8n^2, \text{ if } n \geq 1 \\ &= 25n^2 \end{aligned}$$

and so $n_0 = 1$ and $c = 25$ does the job.

We can perform an alternatively manipulation:

$$\begin{aligned} t(n) &= 17 - 46n + 8n^2 \\ &\leq 17 + 8n^2, \quad n > 0 \\ &\leq n^2 + 8n^2, \ \textit{if } n \geq 5 \\ &= 9n^2 \end{aligned}$$

and so $c = 9$ and $n_0 = 5$ does the job.

Example 3

Show $t(n) = \frac{500+20\log n}{n}$ is $O(1)$.

First note

$$t(n) = \frac{500 + 20\log n}{n} \leq \frac{500 + 20n}{n}$$

since $\log n < n$ for all $n \geq 1$ (see below). We now want to show there exist positive c and n_0 such that for all $n \geq n_0$,

$$\frac{500 + 20n}{n} < c$$

Take $c = 520$. Then we want to show there exists an n_0 such that

$$500 + 20n < 520n$$

for all $n \geq n_0$. But $n_0 = 2$ clearly does the job.

[BEGIN ASIDE: Can you formally prove the "obvious" claim that $\log n < n$ for $n \geq 1$. One easy way to do this is to note the "even more obvious" claim $n < 2^n$ for all $n \geq 1$ and take the log of both sides. To formally *prove* that $n < 2^n$ for $n \geq 1$, use induction. The base case is trivial. For the induction step, we have

$$\begin{aligned} k+1 &< 2^k + 1 \ \text{ by the induction hypothesis} \\ &< 2^k + 2^k \\ &= 2^{k+1} \text{ and we are done.} \end{aligned}$$

END ASIDE]

Some background logic

I began the lecture by reviewing some of the logic tools that we will need for this lecture. The main tool is to show how to negate statements that have logical quantifiers such as "for all" and "there exists".

The first thing to appreciate is that when you have multiple quantifiers in a single statement, the order matters. For example, suppose I say that "for all integers x, there exists an integer y such that $x > y$". This statement is true; it just means that there is no largest integer. However if I reverse the quantifiers, then I get the statement "there exists an integer y such that for all integers x, $x > y$", which is now false; it just means that there exists a largest integer.

We next looked the problem of how to negate statements that involve logical quantifiers. Recall that my leadup to the big O definition last class when I introduced a simpler definition[18]: $t(n)$ *is asymptotically bounded above by* $g(n)$ if there exists a positive number n_0 such that, for all $n \geq n_0$,

$$t(n) \leq g(n).$$

How do we negate such a statement? We say $t(n)$ is *not* asymptotically bounded above by $g(n)$ if there *does not* exist a positive number n_0 such that, for all $n \geq n_0$,

$$t(n) \leq g(n).$$

This means that whatever $n_0 \geq 0$ we choose, we'll be able to find an n such that $n \geq n_0$ and $t(n) > g(n)$.

not big O

What does it mean for a function $t(n)$ *not* to be $O(g(n))$, or equivalently, that that statement "$t(n)$ is $O(g(n))$" is false? Saying $t(n)$ is *not* $O(g(n))$ means that there do *not* exist two positive constants c and n_0 such that, for all $n \geq n_0$, $t(n) \leq cg(n)$. Logically, this is equivalent to saying that, *for any* two positive constants c_0 and n_0, there exists at least one n such that $n > n_0$ and $t(n) > cg(n)$. Let's look a few examples of $t(n)$ and $g(n)$ for which $t(n)$ is not $O(g(n))$.

Example: $t(n) = 3^n$ **is** *not* $O(2^n)$

Take any $c > 0$ and $n_0 \geq 0$. We want to show there exists an n such that $n \geq n_0$ and $3^n > c2^n$, or equivalently, $(\frac{3}{2})^n > c$. Clearly such an n exists since the left side of the last inequality increases without bound, whereas the right side is a constant. To find a value of n, take logs of both sides and manipulate to get $n > \log(\frac{3}{2})/\log c$. Thus, if we choose an n greater than this this (and greater than n_0) then we have shown that $t(n) = 3^n$ is *not* $O(2^n)$.

Example: $t(n) = 3n^2 + 5n + 2$ **is** *not* $O(n)$

To prove formally that $t(n)$ is not $O(\ n\ )$, we use a different argument from what we saw last lecture, namely we use a *proof by contradiction*. We hypothesize that $t(n)$ is indeed $O(n)$, and then we show that this hypothesis cannot be correct, i.e. it leads to a contradiction.

[18] which I admit is non-standard, but hopefully you find useful

Assume $t(n)$ is $O(n)$. Let $c > 0$ and $n_0 \geq 0$ be two constants such that, for all $n \geq n_0$,

$$3n^2 + 5n + 2 \leq cn.$$

Dividing both sides by n gives an equivalent inequality,

$$3n + 5 + \frac{2}{n} \leq c \; . \qquad (*)$$

But we see that this cannot be true for all $n \geq n_0$ since the left side grows without bound.

To get a concrete value of the n that produces the failure, note that

$$\begin{aligned} 3n + 5 + \frac{2}{n} &> 3n + 5, \quad \text{if } n > 0 \\ &\geq c, \quad \text{if } n \geq max\{0, \frac{c-5}{3}\} \end{aligned}$$

So, any $n > max(\frac{c-5}{3}, n_0)$ will contradict the big O definition for the chosen constants n_0, c. Hence it is *not* possible to find such constants that satisfy the big O definition. Hence $t(n)$ is not $O(n)$.

In the proof above, do you need to provide the "concrete value" of n? Or could you stop at the line (*) ? In this case, I would say the latter is fine since there is nothing very subtle about the claim that the left side increases without bound as $n \to \infty$.

Big Omega (asymptotic lower bound)

With big O, we defined an asymptotic upper bound. There is a similar definition for asymptotic lower bounds. This lower bound is used to claim that something can only be done so fast. You will make great use of Ω() arguments in future Algorithms courses.

As an example, consider $t(n) = \frac{n(n-1)}{2}$. For sufficiently large n, this function is greater than $g(n) = \frac{n}{4}$. More formally, we would say that $t(n)$ is asymptotically bounded below by $g(n)$ if there exists an n_0 such that, for all $n \geq n_0$, we have $t(n) \geq g(n)$. The definition of Ω( ) is slightly more complicated than this, though, since we want to use functions $g(n)$ that don't have constants in them – just like in the definition of $O(\ )$.

Definition (big Ω): We say that $t(n)$ is $\Omega(g(n))$ – "big Omega of $g(n)$" – if there exists positive constants n_0 and c such that, for all $n \geq n_0$,

$$t(n) \;\geq\; c\, g(n).$$

The idea is that $t(n)$ grows at least as fast as $g(n)$ times some constant, for sufficiently large n. Note that the only difference between the definition of $O()$ and $\Omega()$ is the $\leq$ vs. $\geq$ inequality.

As an example, you will prove in COMP 251 that the average case behavior, $t(n)$, of *any* sorting algorithm is $\Omega(n \log n)$. That is, it is impossible that algorithm that sorts n arbitrary numbers using fewer than $cn \log n$ operations, on average.[19]

See the slides for a few examples and see Exercises 5 for many more examples.

[19] If you are wondering how "average" is defined here, note that there are $n!$ arrangments of n distinct numbers and only one of the arrangements is correctly sorted. "Average case" analysis treats the $n!$ arrangements as equally likely.

Big Theta

It often happens that $t(n)$ is both $O(g(n))$ and $\Omega(g(n))$. For example, $t(n) = \frac{n(n+1)}{2}$ is both $O(n^2)$ and $\Omega(n^2)$. In this case, we say that $t(n)$ is $\Theta(g(n))$, i.e. "big theta".

Subset notation

Sometimes you find statements like "t(n) $\in$ O(g(n))" which means that the function $t(n)$ belongs to the set of functions that are $O(g(n))$. If we think of $O(g(n))$ as a set of functions, then we can define strict containment relations on these sets:

$$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset O(n^2) \ \ldots$$

Similarly, we can write

$$\cdots \subset \Omega(n^2) \subset \Omega(n \log n) \subset \Omega(n) \subset \Omega(\sqrt{n}) \subset \Omega(\log n) \subset \Omega(1)$$

That is, $\Omega(1)$ is the largest of the sets shown. It is the set of functions that are asymptotically bounded below by a positive constant – note this includes any function that does not go to 0 as $n \to \infty$.

(Rooted) Trees

Thus far we have been working with "linear" collections, namely *lists*. For each element, it made sense to talk about the previous element (if it exists) and the next element (if it exists). We saw two data structures for representing lists, namely linked lists and arrays. Both have limitations, though, in that certain common operations are $O(n)$. For linked lists, finding the i^{th} element is $O(n)$. For arrays, add an element at the i^{th} position is $O(n)$, as is removing the element at the i^{th} position if we require that the array doesn't have any holes.

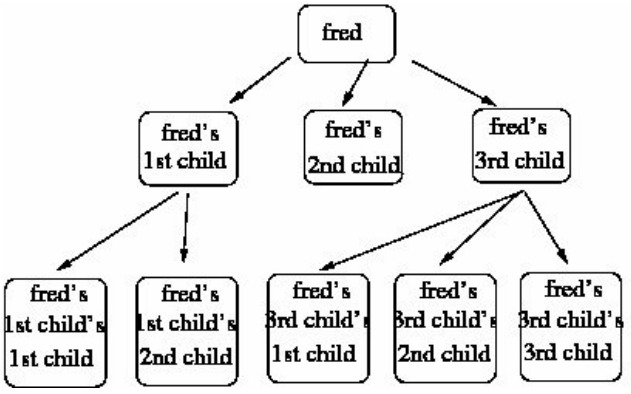
To get around these limitations, one often organizes a collection of items in a "non-linear" way. In the next several lectures, we will look at some examples. Today we begin with *(rooted) trees*.

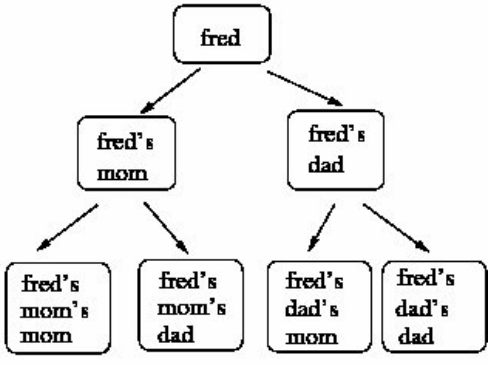
Like a list, a rooted tree is composed of nodes that reference one another. With a tree, each node can have one "parent" and multiple "children". You can think of the parent as the "prev" node and the children as "next" nodes. What is new here with rooted trees is that a node can have multiple children, whereas in a linked list a node has (at most) one "next" node.

You are familiar with the concept of rooted trees already. Here are a few examples.

- Many organizations have a hierarchical structures that are trees. For example, as a McGill professor in the School of Computer Science, I report to my department Chair, who reports to the Dean of Science, who reports to the McGill Provost, who reports to the Principal. A professor in the Department of Electrical and Computer Engineering reports to the Chair of ECE, who reports to the Dean of Engineering, who like the Dean of Science reports to the Provost, who reports to the Principal. (The "B reports to A" relationship in an organization hierarchy defines a "B is a child of A" relation in a tree – see below).

- Family trees. There are two kinds of family trees you might consider, both having a person "fred" at the root. The first tree is the more conventional "family tree. It defines the parent/child relation literally, so that the tree children of a node correspond to the actual children ("kids") of the person represented by the node. The second tree is less conventional, but it is interesting too so I'll mention it. It defines each person's mother and father as its "tree children". A person's real mother and father are obviously not the person's "children", but here we are using "children" only in a formal sense of a tree definition. Note that, in this sense, each person has two children (the person's real parents). Thus, each person has four grandparents, eight greatgrandparents, etc.





- A file directory on a MS Windows or UNIX/LINUX operating system. For example, this lecture notes file is stored on a UNIX system and has a path[20]

  `/home/perception/langer/public_html/250/lecture17.pdf`

  where the backslashes indicate a parent/child relationship.

Terminology for rooted trees

We begin by treating a *rooted tree* as an abstract data type (ADT). Here is some terminology to get us started. It is not as formal as you would see in a math course, but it should be unambiguous enough to give you the basic ideas. *See slides for pictures that go along with these "definitions". These pictures will help alot.*

- *node* (or *vertex*) - nodes play a similar role in trees as they play in lists. This will be clear when we say more about what kinds of nodes there are and how they are used.
- (directed) *edge* - an *ordered* pair of nodes. For simplicity today, unless I say otherwise, we let all edges be of the form $(parentnode, childnode)$, that is, the edge goes from one node called the "parent" to another node called the "child"
- a node v has a *parent* if there is another node p and an edge (p, v). All nodes (except for the *root node*) have one and only one parent. The root node does not have a parent.

  It follows immediately that a tree with n nodes has $n - 1$ edges, i.e. each node except the root has exactly one parent.
- *sibling relation*: two nodes are siblings if they have the same parent.
- *leaf*: a node with no children, that is, a node v such that there does *not* exist an edge (v, w) where w is any other node in the tree. Leaves are also called *external* nodes.
- *internal* node: a node that has a child (i.e. a node that is not a leaf node).

  For example, in the UNIX or windows file system, files and empty directories are leaf nodes, and non-empty directories are internal nodes. A directory is a file that contains a list of references to its children nodes, which maybe files (leaves) or subdirectories (internal nodes).
- *path*: a sequence of nodes $v_1, v_2, \ldots, v_k$ where v_i is the parent of v_{i+1} for all i.
- *length of a path*: the number of edges in the path. If a path has k nodes, then it has length $k - 1$.

  You can define a path of length 0, namely a path that consists of just one node v. (This is useful sometimes, if you want to make certain mathematical statements bulletproof.)
- *depth* of a node (also called *level* of the node): the length of the (unique) path from the root to the node. Note that the root node is at depth 0.

[20] The "/home" at the beginning indicates that "home" is a child of the root directory "/". The root directory has many other children. Type "`ls /`" on a unix/linux command line to see the other sub-directories of the root directory.

- *height* of a node v: the maximum length of a path from v to a leaf. Note: a leaf has height 0.
- *height* of a tree: the height of the root node

  See the slides for some examples of depth and height.
- *ancestor*: v is an ancestor of w if there is a path from v to w
- *descendent*: w is a descendent of v if there is a path from v to w. Note that "v is an ancestor of w" is equivalent to "w is a descendent of v".

Trees and recursion

It is quite common to define operations on trees recursively. Here are a few common examples.

```
depth(v){
  if (v is a root node)    //  that is,  v.parent == null
    return 0
  else
    return 1 + depth(v.parent)    //   parent is well defined for all
                                  //   non-root nodes (though some
}                                 //   implementations don't have a
                                  //   "parent" field)

height(v){
  if (v is a leaf)    //  that is,  v.child == null for any child
    return 0
  else{
    h = 0
    for each child w of v
       h = max(h, height(w))
    return 1 + h
  }
}
```

Tree implementations

There is much to learn about trees as ADTs, which is independent of how they are implemented. Despite this, it might help you to think concretely at this time about how to implement the tree ADT.

The main tricky issue is how to represent the set of children of a node. If node can have only two children (as is the case of a binary tree, which we will describe soon) then each node can be given exactly two reference variables for the children. If, however, a node can have many children, then it is less clear what to do. You could define an N-dimensional array of children references, which would allow for up to N children, though this might waste alot of memory if the number of children was typically much less than N. If you using Java, then you might want to use an `ArrayList` or `LinkedList`.

An alternative, which is a much older and much more common representation, is to use the *"first child, next sibling"* implementation of a tree.

```
  class  TreeNode<T>{
     T             element;
//   TreeNode<T>   parent;         //  this field is optional
     TreeNode<T>   firstChild;
     TreeNode<T>   nextSibling;
             :
             :                     //  methods
  }
```

This implementation defines a singly linked list for the siblings, and either a singly or doubly linked list for the parent/first-child relationship. The parent/first-child is singly linked if only the `firstChild` field is defined, and it is doubly linked if also the `parent` field is defined. (Note that to define the `depth()` method as done earlier, we would need to have a `parent` field.)

To define the rooted tree, we could use:

```
  class  Tree<T>{
     TreeNode<T>   root;
             :
             :                     //  methods
  }
```

The `root` here serves the same role as the `head` field in our implementation of `SLinkedList`. It gives you access to the nodes of the tree.

Tree traversal

Often we wish to examine (or visit) all the nodes of the tree. This is called *tree traversal*, or *traversing* a tree. There are two aspects to traversing a tree. One is that we need to follow references from parent to child, or child to its sibling. The second is that we need to "visit" the node. By "visit", I mean doing some computation, e.g. getting or setting a field of an element referenced by that node.

There are several different ways in which we can traverse a tree. They differ in the order in which the nodes are visited.

Depth first traversal

The first two traversals we look at are called "depth first". In these traversals, a node and all its descendents are visited before the next sibling is visited. There are two ways to do depth-first-traversal of a tree, depending on whether you visit a node before its descendents or after its descendents. In a *pre-order* traversal, you visit a node, and then visit all its children. In a *post-order* traversal, you visit the children of the node (and their children, recursively) and then visit the node.

```
preorderTraversal(root){
  if (root is not empty){
    visit root
    for each child of root
      preorderTraversal(child)
  }
}

postOrderTraversal(root){
  if (root is not empty){
    for each child of root
      postorderTraversal(child){
    visit root
  }
}
```

Two examples are illustrated in the lectures slides. Suppose we have a file system. The directories and files define a tree whose internal nodes are directories and whose leaves are files. We first wish to print out the root directories, namely list the subdirectories and files in the root directory. For each subdirectory, we also print its subdirectory and files, and so on. This is all done using a pre-order traversal. The visit would be the print statement. Here is a example of what the output of the print might look like. (This is similar to what you get on Windows XP when browsing files in the Folders panel.)

```
  My Documents                  (directory)
     My Music                   (directory)
        Raffi                   (directory)
           Shake My Sillies Out (file)
           Baby Beluga          (file)
```

```
      Eminem                    (directory)
         Lose Yourself          (file)
   My Videos                    (directory)
      Kids skiing at Tremblant  (file)
   Work                         (directory)
      COMP250                   (directory)
         etc
```

Next let's look at an example of post-order traversal. Suppose we want to calculate how many bytes are stored in all the files within some directory (including sub-directories, etc). The reason this is post-order is that, to know the total bytes in some directory, we first need to know the total number of bytes in all the subdirectories Hence, I need to visit the subdirectories first. Here is an algorithm for computing the number of bytes. Note that it traverses the tree in postorder in the sense that it computes the sum of bytes in each subdirectory by summing the bytes at each child node of that directory.

```
numBytes(root){
  if root is a leaf
     return number of bytes at root
  else{
     sum = 0    // local variable
     for each child of root{
        sum += numBytes(child)}
     return sum
  }
}
```

Depth first traversal without recursion

As we have discussed alrad in this course, recursion is implemented using a call stack which keep track of information needed in each call. You can often avoid recursion without much trouble by using an explicit stack instead. Here is an algorithm for doing a pre-order depth first traversal which uses a stack but does not use recursion. As you can see by running an example (see lecture slides), this algorithm visits the children of a node in the opposite order to that defined by the "for each" statement. The algorithm is still pre-order though, since it visits each node before visiting its children.

```
preOrderWithoutRecursion(root){
  s.push(root)
  while !s.isEmpty()
    cur = s.pop()
    visit cur
    for each child of cur
       s.push(child)
  }
}
```

Breadth first traversal

The opposite of a depth first traversal is a breadth first traversal. (Note that the following algorithm is not recursive.)

```
for each level i in the tree  //  level is the same as depth
   visit all nodes at level i
```

Here is a more detailed sketch of how to implement this algorithm. It is the same as the previous algorithm (for non-recursive depth first traversal) except that it uses a queue instead of a stack.

```
q = empty queue
q.enqueue(root)
while !q.isEmpty() {
  cur = q.dequeue()
  visit cur
  for each child of cur
    s.enqueue(child)
}
```

Note that you can replace the last two lines of the above algorithm as follows. Recall first-child/next-sibling data structure for representing a tree, which we saw last lecture. Then

```
//    for each child of cur           //   REPLACE THESE TWO LINES
//       s.enqueue(cur.child)         //

     child = child.firstChild
     while (child != null){
       q.enqueue(child)
       child = child.nextSibling }
```

Why would you want to do a breadth first traversal? Let's consider an example of a two player game, such as chess. Let the root node be the current state of the game, and suppose it is your turn to move. You have several possible moves you could make. Each of these moves defines a child node of the root, which would be the next state of the game if you were to make that move. Since it would then be your opponent's move, each of these child nodes would define another set of states (the child node's children) which the game would be in after your opponent has made its next move, etc. One way to think of a chess strategy is to consider each of your next moves, and then consider your opponent's moves, and then your moves that would follow. You can quickly see the number of states to be searched would rise very fast with the number of levels of your breadth first search. If you want to learn more about these types of search problems, then take COMP 424 Intro to AI.)

Example

See the slides (and exercises) for many examples !

Binary Trees

The *order* of a (rooted) tree is the maximum number of children of any node. A tree of order n is called an n-ary tree. It is very common to use trees of order 2. These are called *binary trees*.

Each node of a binary tree can have two children, called the *left child* and *right child*. The terms "left" and "right" refer to their relative position when you draw the tree.

Number of nodes of a binary tree

How many nodes can a binary tree have at each level? The root has one node. Level 1 has two nodes (the two children of the root). Level 2 has four nodes, namely each child at level 1 can have two children. You can easily see that the maximum number of nodes doubles at each level, and so level l can have 2^l nodes. For a binary tree of height h, the maximum number of nodes is thus:

$$n_{max} = \sum_{l=0}^{h} 2^l = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1.$$

You have seen this geometric series several times, and you will see it again...

The minimum number of nodes in a binary tree of height h is of course $h + 1$, namely each node has at most one child (the sole leaf has no children). It follows that

$$h + 1 \leq n \leq 2^{h+1} - 1$$

We can rearrange this equation to give

$$\log(n + 1) - 1 \leq h \leq n - 1.$$

These inequalities be very useful in the coming lectures, when we use binary trees to solve many problems.

Binary tree nodes: Java

Last lecture we looked at the first-child/next-sibling data structure for general trees. For binary trees, one can use a (conceptually) simpler data structure:

```
class  BTNode<T>{
   T             e;
   BTNode<T>   left;
   BTNode<T>   right;
}
```

One can have a `parent` reference too, if necessary, but we don't use it now. A `parent` reference is analogous to a `prev` reference in a doubly linked list.

Binary tree traversal

A binary tree is a special case of a tree, so the algorithms we have discussed for computing the depth or height of a tree node and for traversing a tree apply to binary trees as well. We saw two simple depth-first search algorithms for general trees, namely pre- and post-order, and for binary trees they can be written as follows:

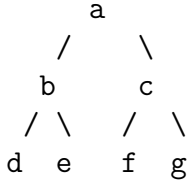
```
preorder(root){
    if (root is not null){              // base case
       visit root
       preorder(root.left)
       preorder(root.right)
    }
}

postorder(root){
    if (root is not null){              // base case
       postorder(root.left)
       postorder(root.right)
       visit root
    }
}
```

For binary trees, there is one further traversal algorithm to be considered, which is called *in-order traversal*.

```
inorder(root){
    if (root is not null){              // base case
       inorder(root.left)
       visit root
       inorder(root.right)
    }
}
```

You *could* define an inorder traversal for general trees. For example, you could visit the first child, then visit the root, then visit any remaining children. But such inorder traversals are typically not done for general trees.

Example



```
level order:  a b c d e f g   (breadth first)
pre-order:    a b d e c f g   (depth first)
post-order:   d e b f g c a       "
in-order:     d b e a f c g       "
```

Expressions

You are familiar with forming expressions using *binary operators* such as `+,-,*, /, ^`. (The operator `^` is the power operator i.e. `x ^ n` is `power(x,n)`.) Each of the operators takes two arguments, called the left and right *operands*. Let's define a set of simple expressions recursively as follows, where the symbol `|` means *or*.

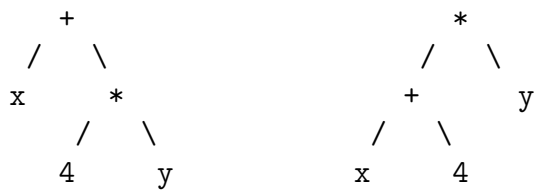
```
baseExpression  =  digit | letter
operator        =  + | - | * | / |  ^
expression      =  baseExpression |  expression operator expression
```

Examples[21] of a `baseExpression` are "3", "x", "a", namely a base expression can be either a number (one digit) or a variable (one letter). An `operator` is a binary operator. An `expression` can consist of either a base expression, or it can consist of one expression followed by an operator followed by an expression. *Notice that expressions are defined recursively, and that we have a base case.*

[You have seen such notation in Assignment 2 Question 1. See slides for an example.]

Expression trees

You can represent these expressions using trees, called *expression trees*. For example, the expression `x + 4 * y` could be defined either of these two trees:



Typically though, when we have an expression with multiple operators, there is a particular order in which the operators are supposed to be applied. You learned these precedence orderings in grade school. For example "`x + 4 * y`" is to be interpreted as "`x + (4 * y)`" shown on the left, rather than "`(x + 4) * y`" shown on the right. (There is also a convention that `6 ^ z ^ 8` means `6 ^ (z ^ 8)` rather than `(6 ^ z) ^ 8`, although some may use just the opposite convention (it is arbitrary).

The above precedence order implies that an expression such as

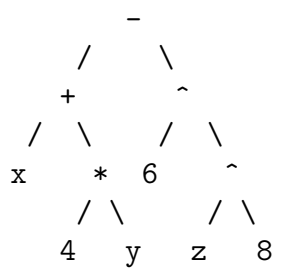
`x + 4 * y - 6 ^ z ^ 8`

[21] In an earlier version of the notes, I also wrote "-16" is an example. That was a typo.

can be uniquely interpreted as if there were a nesting of brackets:

$$(\texttt{x} + (\texttt{4} * \texttt{y})) - (\texttt{6} \,\hat{}\, (\texttt{z} \,\hat{}\, \texttt{8}))$$

and the expression can be represented as a tree:



Expression trees can be evaluated recursively as follows.

```
evaluateET(root){
   if (root is a leaf)   // can be determined by checking if it has
                         // any children
      return value
   else{ //  the root is an operator
      firstOperand = evaluateET(left child of root)
      secondOperand = evaluateET(right child of root)
      return evaluate(firstOperand, root, secondOperand)
   }
```

We may think of this algorithm as performing a *postorder* traversal of the tree in the sense that, to evaluate the expression defined by a tree, you *first* need to evaluate the left and right child of the root, and *then* you can apply the operator at the root.

In-fix, pre-fix, post-fix expressions

You are used to writing expressions as two operands separated by an operator. This representation is called *infix*, because the operator is "in" between the two operands. For infix expressions, the order of evaluation is determined by precedence rules.

An alternative way to write an expression is to use *prefix* notation. Here the operator comes *before* the two operands. For example,

```
- + x * 4 y ^ 6 ^ z 8
```

which is interpreted as

```
(- (+ x (* 4 y))(^ 6 (^ z 8))) .
```

Notice that a prefix expression gives the ordering of elements visited in a preorder traversal of the expression tree.

An second alternative is a *postfix* expression, where the operators comes *after* the two operands, so

```
x 4 y * + 6 z 8 ^ ^ -
```

is interpreted as

```
((x (4 y *) + ) (6 (z 8 ^) ^ ) - ) .
```

Notice that the ordering of elements is the visit order in a post-order traversal of the expression tree.

One can formally define a set of *in, pre, and post*fix expressions recursively as follows:

```
baseExpression    =  digit | letter
operator          =  + | - | * | / |  ^
infixExpression   =  baseExpression |  infixExpression operator infixExpression
```

which is just what we saw earlier in the lecture (when all expressions were infix).

One modifies the definition slightly for prefix expressions:

```
baseExpression    =  digit | letter
operator          =  + | - | * | / |  ^
prefixExpression   =  baseExpression |  operator prefixExpression prefixExpression
```

and for postfix expressions

```
baseExpression    =  digit | letter
operator          =  + | - | * | / |  ^
postfixExpression =  baseExpression |  postfixExpression  postfixExpression operator
```

ASIDE: Prefix notation is sometimes called "Polish" notation – it was invented by a Polish logician, Jan Lukasiewicz (about 100 years ago). Postfix notation is sometimes called "reverse Polish notation".

The advantage of postfix and prefix expressions over infix expressions is that you do not need a precedence rule to define the order of operations. It is easiest to see the usefulness of this with postfix expressions. Here is a stack-based algorithm for evaluating a postfix expression. The algorithm assumes the expression is a list of variables, numbers, and operators, specifically, binary operators i.e with two operands. It does not need to know about precedence orderings, since these are "built in".

See the slides for an example of how the stack evolves over time.

```
s = empty stack
cur = head;
while (cur != null){
   if (cur.element is a variable or number)
      s.push(cur.element)
   else{ //  cur is an operator
     operand1 = s.pop()
     operand2  = s.pop()
     operator  = cur.element
     s.push( evaluate( operand1 operator operand2 ) )
   }
   cur = cur.next
}
```

ASIDE: In the 1970's Hewlett-Packard introduced the first desktop calculators. These required that users enter expressions using postfix! Ask your parents (or grandparents?).

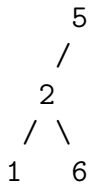
Binary Search Trees

Today we consider a specific type of binary tree in which there happens to be an ordering defined on the set of elements the nodes. If the elements are numbers then there is obviously an ordering. If the elements are strings, then there is also a natural ordering, namely the dictionary ordering, also known as "lexicographic ordering".

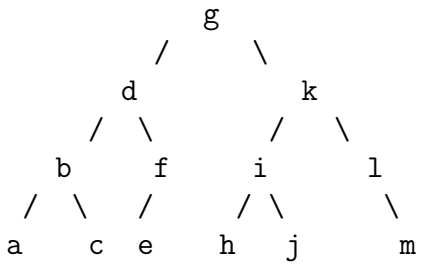
A *binary search tree* is a binary tree such that

- the elements (also called *keys*) are comparable, namely there is a strict ordering relation `<`
- any two nodes have different keys (i.e. no repeats)
- for any node,
  - all keys in the left subtree are less than the node's key
  - all keys in the right subtree are greater than the node's key.

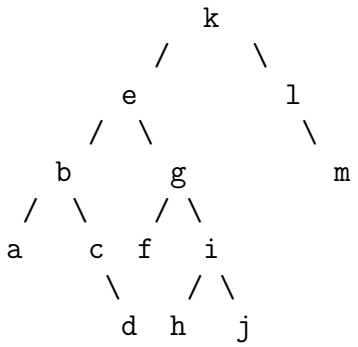
  (Note that this is stronger than just saying that the left child's key is less than the node's key which is less than the right child's key. For example, the following is not a binary search tree.)



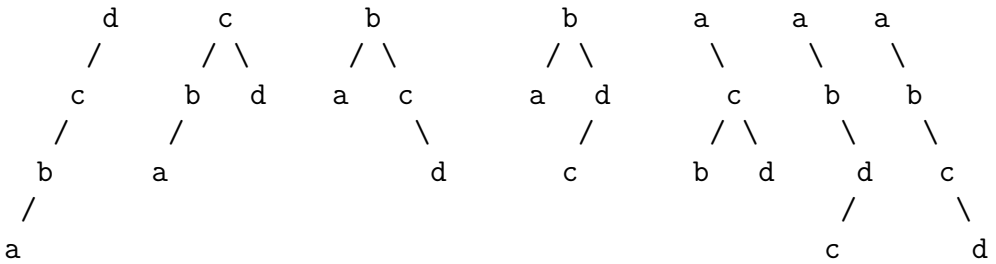
One important property of binary search trees is that *an inorder traversal of a binary search tree gives the elements in their correct order.* Here is an example with nodes containing keys `abcdefghijklm`. Verify that an inorder traversal gives the elements in their correct order.

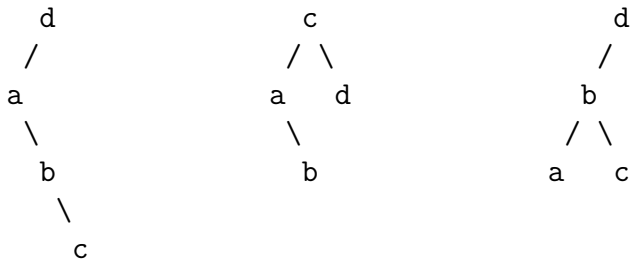


and here is another example with the same set of keys:



Here are several examples of binary search trees whose nodes contain the characters `a,b,c,d`. There are other examples as well (see lecture slides).





Enumerating BSTs with n nodes

Here is a systematic way to enumerate all the binary search trees (BSTs) containing a given set of keys. The idea is to consider all possible binary search trees with each key at the root. If there are n keys, then for each of the n choices of the key at the root node, there are $n-1$ non-root nodes and these non-root nodes must be partitioned into those whose keys are that are less than the key of the chosen root and those whose keys are greater than the key of the chosen root.

Let the key's be numbered from 1 to n. If key i is the root, then there are $i-1$ keys smaller than i and $n-i$ keys bigger than i. For each of these two sets of keys, there is a certain number of possible subtrees. Let $t(n)$ be the total number of BSTs with n nodes. The total number of BSTs with the i-th at the root is $t(i-1) \ * \ t(n-i)$. The two terms are multiplied together because the arrangements in the left and right subtrees are independent. That is, for each arrangement in the left tree and for each arrangement in the right tree, you get one BST with key i at the root. Summing over i then gives the total number of binary search trees with n nodes,

$$t(n) = \sum_{i=1}^{n} t(i-1)\ t(n-i).$$

The base case is $t(0) = 1$ and $t(1) = 1$, i.e. there is one empty BST and there is one BST with one node.

$$t(2) = t(0)t(1) + t(1)t(0) = 2$$

$$t(3) = t(0)t(2) + t(1)t(1) + t(2)t(0) = 2 + 1 + 2 = 5$$

$$t(4) = t(0)t(3) + t(1)t(2) + t(2)t(1) + t(3)t(0) = 5 + 2 + 2 + 5 = 14$$

etc.. Note that we are computing $t(n)$ here using forward substitution, not back substitution.

BST as an abstract data type (ADT)

One performs several common operations on binary search trees:

- `find(key)`: given a key, return a reference to the node containing that key (or null if key is not in tree)
- `findMinimum()` or `findMaximum()`: find the node containing the smallest or largest key in the tree and return a reference to the node containing that key
- `add(key)`: insert a new node into the tree such that the node contains the key and the node is in its correct position (if the key is already in the tree, then do nothing)
- `remove(key)`: remove from the tree the node containing the key (if it is present)

There are various ways of specifying these operations. Here are a few examples.

```
find(root,key){
  if (root == null)
     return null
  else if (root.key == key))
    return root
  else if (key < root.key)
     return  find(root.left, key)
  else
     return  find(root.right, key)
}

findMinimum(root){
   if (root == null)    // only necessary for the first call
     return null
   else if (root.left == null)
      return root
   else
      return findMinimum(root.left)
}
```

Notice that the minimum key is not necessarily a leaf i.e. it can occur if the key has a right child but no left child.

```
findMaximum(root){
   if (root == null)
      return null
   else if (root.right == null))
      return root
   else
      return findMaximum(root.right)
}
```

Let's next consider adding (inserting) a key to a binary search tree. If the key is already there, then do nothing. Otherwise, make a new node containing that key, and insert that node into its *unique* correct position in the tree.

```
insert(root,key){
   if (root == null)
      root =  new BSTnode(key)    //  makes a new node and returns it
   else if (key < root.key){
      root.left = insert(root.left,key)
   else if (key > root.key){
      root.right = insert(root.right,key)
   return root
}
```

Advanced Discussion (for your interest only)

In COMP 251, you will learn about data structures and algorithms for adding and removing keys to/from a BST such that the BST height is $O(\log n)$, rather than $O(n)$. This ensures a fast search. Depending on the prof, you might also study *random trees* and carry out a probabilistic analysis of how binary search trees grow when random elements are added. For example, here is a "back of the envelope" sketch of why the expected height[22] of a binary search tree is $O(\log n)$.

Let $t(n)$ be the expected height of a binary search tree with n nodes. Suppose we have a BST with n nodes, where n is big. If we add a $n+1^{st}$ node, the probability that this node will extend the height of the tree is about $\frac{2}{n}$. The reasoning is as follows. First, if h is the height of this tree, then it is very unlikely that there will be more than one path of length h. Thus, if adding a $n+1^{st}$ node extends the height by 1, then we only need to consider one leaf, namely the leaf with the greatest depth. In order for the height to be extended, the element that we are adding has to come either just before or just after this leaf. Since there are n nodes, the probability of it coming just before this leaf is about $\frac{1}{n}$ and the probability of it coming just after is about $\frac{1}{n}$. Another way to see this is to note that there are $n+1$ positions where the new key could go in the ordering of the n existing keys, and two of these positions would increase the height of the tree. To keep the math simple, let's say that the probability of extending the height of the tree is about $\frac{2}{n}$.

The expected height $t(\ )$ of the tree with $n+1$ nodes now can be written in terms of the expected height with n nodes as follows:

$$\begin{aligned} t(n+1) &\approx \frac{2}{n}(t(n)+1) + (1-\frac{2}{n})t(n) \\ &\approx \frac{2}{n}t(n) + \frac{2}{n} + t(n) - \frac{2}{n}t(n) \\ &= \frac{2}{n} + t(n) \end{aligned}$$

The argument about the probability being $\frac{2}{n}$ only holds for large n, say $n \geq n_0$. So we back

[22]The "expected value" of a random variable is well-defined defined in probability theory. If you haven't seen it before, then you will see it in MATH 323 or equivalent.

substitute to n_0:

$$\begin{aligned} t(n) &\approx 2(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots \frac{1}{n_0}) + t(n_0) \\ &\approx 2\log_e n - 2\log_e n_0 + t(n_0) \end{aligned}$$

where we used the fact that

$$\sum_i^j \frac{1}{n} \approx \int_i^j \frac{1}{x} dx = \log_e j - \log_e i,$$

from Calculus. From here, you can easily see that $t(n)$ is $O(\log n)$. This is a slightly more sophisticated sketch of a solution than the one given in class but the idea is the same.

Binary search trees (continued)

Here is an algorithm for removing a node from a binary search tree.

```
remove(root, key){
    if( root  == null )
        return null
    else  if ( key < root.key )                                    (*)
        root.left = remove( root.left, key )
    else  if ( key > root.key )                                    (*)
        root.right = remove( root.right, key)
    else  if root.left == null                                     (**)
        root  =  root.right         // or just "return root.right"
    else  if root.right == null                                    (**)
        root  = root.left           // or just "return root.left"
    else{                                                          (***)
        root.key = findMin( root.right ).key;
        root.right = remove( root.right, root.key );
    }
    return root;
}
```

The (*) conditions handle the case that the key is not at the root, and in this case, we just recursively remove the key from the left or right subtree. Note that we replace the left or right subtree with a subtree that doesn't contain the key. To do this, we use recursion, namely we remove the key from the left or right subtree.

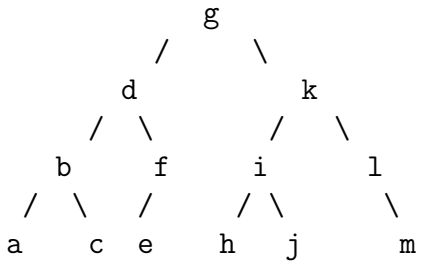
When the key that we want to remove is at the root (perhaps after a sequence of recursive calls), then we need to consider four possibilities:

- the root has no left child
- the root has no right child
- the root has no children at all
- the root has both a left child and a right child

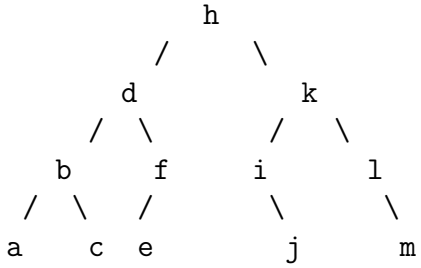
In the first two cases, we just replace the root node with one of its children. Note that the third case is accounted for here as well; since both children are null, the root will become null. In the fourth case, we need to remove the root node and reorganize the tree so that it has a new root node. This can be done by replacing the root with either the minimum node in the right subtree or the maximum node in the left subtree. The algorithm does the first of these. It removes the smallest node in the right subtree, in two steps. It copies the key from the smallest node in the right subtree into the root node, and then it removes the smallest node node from the the right subtree.

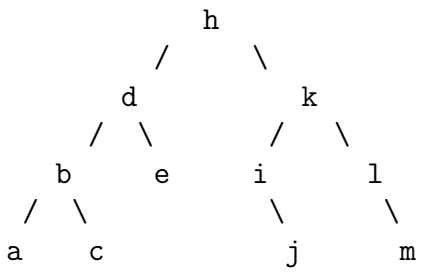
Example (remove)

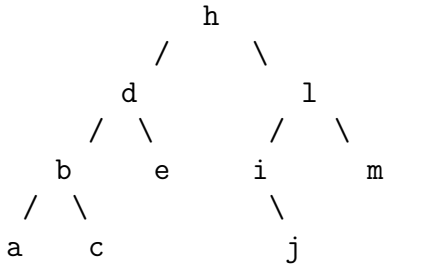
Take the following example with nodes `abcdefghijklm`:

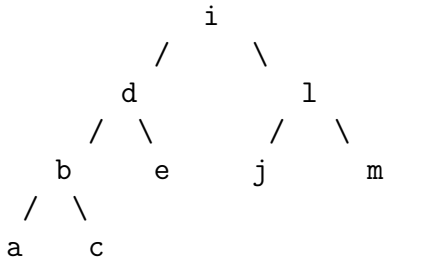


and then remove elements `g,f,k,h` in that order.









Binary search: using arrays vs. BST

The BST `find` algorithm from last lecture is reminiscent of the binary search algorithm we saw lecture 11, where we searched for an item in sorted array. Recall that the idea was to use recursion and to divide the search interval in half at each step. We kept track of indices `low` and `high`, computed their mean at each step, and then checked if the value (key) we were searching for was in the lower half or the upper half of the interval `[low,high]`. This was a very efficient way to search, namely $O(\log n)$.

There is a problem with using arrays, however, namely if you want to add or remove an element then you need to perform $O(n)$ shifts. Indeed, $O(n)$ is the average case behavior as well which we can see intuitively by noting that if we add or remove an element to/from the lower half then we need to do at least $n/2$ shifts.

The binary search tree is a better data structure for doing searches, if we need to add and remove elements as well. Although it is still $O(n)$ for an add or remove, this worst case arises very rarely; as I sketched out last class, the average height of a binary search tree is $O(\log n)$, so finding, adding, and removing elements all typically take $O(\log n)$ time.

BST Java code

Next to the link for these lecture notes is a Java implementation of a binary search tree. The tester class runs a fun experiment where it generates 2^i random integers and adds them to a binary search tree. The method then lists out the height of the resulting binary search tree as a function of i. You can see for yourself that it rises roughly with $\log_2 n$ (times a constant which happens to be just over 2).

Review of logs

At the end of this lecture, I reviewed some properties of logs that you should understand. Please see the slides.

ADTs versus APIs

We have seen many abstract data types (ADTs): lists, stacks, queues, trees, binary search trees. Each ADT consists of a set of data and operations that one performs on the data. These operations are defined independently of any implementation in some programming language. It is useful to keep the implementation details "hidden", so as not to confuse *what* is computed from *how* it is computed, namely we sometimes want to write and analyze algorithms in pseudocode without having to commit outselves to any particular programming language.

Although ADT's are meant to be independent of any particular programming language, in fact they are similar to concrete quantities in programming, namely *interfaces* that are given to a programmer. In Java, for example, there is the *Java API*, which you are familiar with. The API (*application program interface*) gives a user many predefined classes with implemented methods. What makes this an "interface" is that the implementation is hidden. You are only given the names of (public) classes and methods, and possibly fields.

The word "interface" within "Java API" should not be confused with related but different usage of the word, namely the Java reserved word `interface`, which is what this lecture is about.

Java `interface`

A typical first step in designing an object oriented program is to define the classes and the method signatures[23] within each class and to specify (as comments) what operations the methods should perform. Eventually, you implement the methods or you hire someone to implement them.

A user (client) of the class should not need to see the implementation of a method to be able to use it. If the design is good, then all the client needs is a description of what the method does and what is the method signature. This hiding of the implementation is called *encapsulation*.

In Java, if we write *only* the signatures of a set of methods in some class, then technically we don't have a class. What we have instead is an `interface`. So, an `interface` is a Java program component that declares the method signatures but does not provide the method bodies.

We say that a class `implements` an interface if the class has each method that is defined in the interface, in particular, the signatures of the method must be the same. If a class `C` implements an interface `I`, then `C` must implement all the methods from interface `I`, meaning that `C` must specify the body of these methods. (In addition, `C` can have other methods.)

`List` interface

Consider the two classes `ArrayList<T>` and `LinkedList<T>` which are used to implement lists. These two classes share many list-like methods which have the same signatures. Of course, the underlying implementations of the methods are very different since arrays and linked lists are very different, but the result of the methods are the same, in the sense of maintaining a list. For example, if you have a list and then you remove the 3rd item, you get a well-defined new list which doesn't depend on whether the original list was implemented with an array or with a linked list.

[23] By "signature", we mean the return type, method name, and parameters with their types.

The `List<T>` interface includes familiar method signatures such as:

- `void add(T o)`
- `void add(int index, T element)`
- `boolean isEmpty()`
- `T get(int index)`
- `T remove(int index)`
- `int size()`

Both `ArrayList<T>` and `LinkedList<T>` implement this interface, namely they implement all the methods in this interface.

Why is the `List` interface useful? Sometimes you may wish to write a program that uses either an `ArrayList<T>` or a `LinkedList<T>` but you want you code to be general enough to allow either to be used. In this case, you can use the generic Java interface `List<T>`. For example,

```
void myMethod( List<String> list ){
       :
    list.add("hello");
       :
}
```

The Java compiler will see the declared `List<String>` type in the parameter, and it will infer that the argument passed to this method will be an object of some class that implements the `List` interface, with generic type `String`. As long as `list` only uses methods from the `List` interface, the compiler will not complain. *We will say more about this in the next few lectures.*

See the lecture slides for a similar example. I defined a `Shape` interface which has two methods: `getArea()` and `getPerimeter`, where the latter is the length of the boundary of the shape. I then defined two classes `Rectangle` and `Circle` that implement the `Shape` interface, namely they provide method bodies.

Let's now turn to a few other commonly used Java interfaces.

`Comparable` interface

Recall that to define a binary search tree, the elements that we are collecting in the tree must be comparable to each other, meaning that there must be a well-defined ordering. For strings and numbers, there is a natural ordering and you can use the "<" operator. However, for more general classes, you need to define the ordering yourself. How?

Java has an interface called `Comparable<T>` which has one method `compareTo(T)` that allows an object of type `T` to compare itself to another object of type `T`. Any class that implements the interface `Comparable<T>` must, by definition, have a method `compareTo( T )`.

The `compareTo()` method returns an integer and it is defined as follows. Suppose we have variables

```
T  t1, t2;
```

Then the Java API recommends that `t1.compareTo(t2)` return:

- a negative integer, if the object referenced by `t1` is "less than" the object referenced by `t2` in the ordering,
- 0, if `t1.equals(t2)` is true (more on this below)
- a positive integer, if the object referenced by `t1` is "greater than" the object referenced by `t2`.

Notice that this definition depends on the `equals()` method. The default meaning of the `equals()` method is that `t1.equals(t2)` is true if and only if `t1` and `t2` reference the same object. But a class can and often does override this default method. We will revisit this issue (and specify what "default" means) in the next few lectures.

Example: `Rectangle`

Let's define a Java class `Rectangle` that implements the `Comparable` interface. `Rectangle` has two fields `height` and `width`, two methods `getArea()` and `getPerimeter()`, and a `compareTo()` method. How should the `compareTo()` be defined for `Rectangle` ? We could compare two rectangles by their areas, perimeters, widths, or heights, etc. In the code below, we compare by area.

```
public class Rectangle implements Comparable<Rectangle>{
   private double width;
   private double height;

   public Rectangle(double width, double height){
      this.width = width;
      this.height = height;
   }

   public double getArea(){
       return width * height;
   }

   public int  compareTo(Rectangle r) {
      if (getArea() > r.getArea() )
         return 1;
      else if (getArea() < r.getArea() )
         return -1;
      else return 0;      //  Note this will return 0 for any two
                          //  two Rectangles with the same area.
   }

   public double getPerimeter(){
       return  2*(width + height);
```

```
    }
  }
```

Does this definition of the `compareTo()` method satisfy the Java recommendations? Not really. The Java API recommends[24] that `r1.compareTo(r2)` is 0 if and only if `r1.equals(r2)` retrns `true`. This recommendation is ignored in the above code. In particular, there is no `equals` method in the `Rectangle` class and so the default `equals` method will be used here – namely two Rectangles are considered equal if and only if they are the same object (as opposed to them having the same area, or the same width and height, etc). This could potentially produce weird behavior i.e. the `compareTo` and `equals` method could disagree on whether two objects are "equal". Here is a simple example:

```
public class Test{
   public static void main(String[] args){
      Rectangle  r1 = new Rectangle(4.0,5.0);
      Rectangle  r2 = new Rectangle(2.0,10.0);
      System.out.println("result: " + r1.compareTo(r2))

// Would print "result: 0"  since 20.0 == 20.0

      System.out.println("equals    result: " + r1.equals(r2))

// Would print "result: false"  since r1 references a different object.
   }
}
```

[**Added Dec. 12, 2012**] Following the Java recommendation, one should re-define the `equals()` method so that it is consistent with the `compareTo()` method.

```
  public boolean equals(Object r){
      if (this.compareTo( (Rectangle) r) == 0 )
         return true;
      else
         return false
  }
```

Here I am overriding the Object class'es `equals(Object)` method. Now when I do this, the above test code prints ``result: true''.

`Iterator` interface

We have seen many data structures for representing collections of objects (including lists, trees) and we will see more (graphs, hashtables, etc). Often it happens that we would like to visit all the objects in a collection. We have seen how to do this for lists and trees (traversal).

Because stepping through a collection is so common, though, Java defines an interface `Iterator<E>` that makes this a bit easier to do.

[24] `http://docs.oracle.com/javase/6/docs/api/java/lang/Comparable.html`

```
interface Iterator<E>{
  boolean hasNext()
  E       next();    //  Returns the next element and advances.
  void    remove():  //  I did not mention this one in the slides.
}
```

Note that an Iterator object is distinct from the collection that it is iterating over. Moreover, you might have several different Iterators defined for a given collection.[25]

Consider how an Iterator might be implemented for a singly linked list class. It would have a private field `cur` which would be initialized to reference the first element in the list, namely the element referenced by `head`. The constructor of the Iterator would set this field. The `hasNext()` method would then check if `cur == null`. The `next()` method would advance to the next element in the linked list if `iterator.hasNext()` returns `true`. I have implemented an iterator for the `SLinkedList` class – see online code for lecture 4. Note that `Iterator` is an interface, not a class, so I had to implement a class `SLLIterator` which implements the `Iterator` interface.

I also implemented an iterator for binary search trees. See code online for lecture 21.

`Iterable` interface

Collections such `ArrayList` and `LinkedList` do not themselves implement the `Iterator` interface, since they are not Iterators. Rather, they construct Iterator objects that implement the `Iterator` interface. How is this done? Iterator objects are constructed by a method `iterator()` which is the one method in a interface called `Iterable`. The idea here is that if you have a collection such that it makes sense to step through all the objects in the collection, then you can define an Iterator object to do this stepping (if you want). Because the collection has this property, you would say that the collection is "iterable". So, Java has an interface:

```
interface Iterable<T>{
  Iterator<T>  iterator();
}
```

Any class (such as LinkedList or BST) for which it makes sense to have Iterator objects should also have a method called `iterator` that returns an object (an Iterator) whose class implements the `Iterator` interface.

[ASIDE: For many people, the above interfaces are very confusing. But let me assure you that eventually they do make sense and will seem natural.]

Example: list of rectangles

Let's look at an example of how iterator works. We make a linked list of rectangles.

```
LinkedList<Rectangle>  list = new LinkedList<Rectangle>();
Iterator<Rectangle>    iter = list.iterator();
```

[25] As an analogy, consider a collection of quizzes that need to be marked. Each quiz is an object. Suppose each quiz consisted of four questions and suppose there were four T.A.'s marking the quizzes, namely each T.A. marks one question. Think of each T.A. as an iterator that steps through the quizzes.

Suppose we build up a list of rectangles. Then we want to visit all the rectangles in the list and do something with them. We don't have access to the underlying data structure of the linked list (nor do we want to have access). Instead the usual way to do it is:

```
for (int i=0;  i < list.size();  i++){
   list.get(i)   //  do something with i-th rectangle
    :
}
```

Using an iterator instead, we could write:

```
   while (iter.hasNext() ){
      r = iter.next();
      //   do something with rectangle pointed to by r
   }
```

Admittedly, for this LinkedList example, the latter seems no better than the former. To see the advantage of iterators, you need to go beyond this example. In the slides, I give a simple example of a linked list with multiple iterators. (Recall the exam grader analogy from earlier in the lecture.)

For other collections, such as trees, using an iterator would allow you to avoid having to write traversal code over and over again. You would just write the traversal code once (in the class that implements Iterator). Then your program (client) which uses this class would use the `next()` method to step through the collection.

Java "enhanced for loop"

Java allows you to replace the `while` loop above by an *enhanced for loop*, which does the same thing.

```
   for (Rectangle r: myRectangleList){
       //  do something with rectangle r
   }
```

Similarly, suppose you have a binary search tree holding objects of type `Integer`.

```
   BST<Integer>    tree;

   for ( Integer  i : tree ){
       //  do something with element i in the tree
   }
```

For this to work, you need to have an `iterator()` method in the `BST` class. (See code provided with lecture 21.) But once you define this method, the enhanced for loop makes it is very easy to traverse the tree and visit the elements in order.

Inheritance

In our daily lives, we classify the many things around us. We know that the world has objects like "dogs" and "cars" and "food" and we are familiar with talking about these objects as classes: "Dogs are animals that have four legs and people have them as pets and they bark, etc". We also talk about specific objects (instances): "When I was growing up I had a beagle named Buddy. Like most beagles, he loved to hunt rabbits."

We also talk about classes of objects at different levels. For example, take animals, dogs, and beagles. Beagles are dogs, and dogs are animals, and these "is-a" relationships between classes are very important in how we talk about them. Buddy the beagle was a dog, and so he was also an animal. But certain things I might say about Buddy make more sense in thinking of him as an animal than in thinking about him as a dog or as a beagle. For example, when I say that Buddy *was born* in 1966, this statement is tied to him being animal rather than him being a dog or a beagle. (Being born is something animals do in general, not something specific to dogs or beagles.) So being born is something that is part of the "definition" of a being an animal. Dogs *automatically* "inherit" the being-born property since dogs are animals. Similarly, beagles automatically inherit it since they are dogs and dogs are animals.

A similar classification of objects is used in object oriented programming languages. In Java, for example, we can define new (sub)classes from existing classes. When we define a class in Java, we specify certain fields and methods. When we define a subclass, the subclass inherits the fields and methods from the (higher) class. We also may introduce entirely new fields and methods into the subclass. Or, some of the fields or methods of the subclass may be given the same names as those of an existing class, but maybe we change the body of a method. We will examine these choices over the next few lectures.

Terminology

If we have a class `Dog` and we define a new class `Beagle` which `extends` the class `Dog`, we would say that `Dog` is the *base class* or *super class* or *parent class* and `Beagle` is the *subclass* or *derived class* or *extended class*. We say that a subclass *inherits* the fields and methods of the superclass.

```
class Dog  {
   String        dogName
   String        ownerName
   int           serialNumber
   Date          birth
   Date          death
   void    Dog(){ ..  }
     :
   void    bark(){
        System.out.println("woof");
   }
}
```

```
class Beagle extends Dog{
   void  bark(){
        System.out.println("aaaaawwwwooooooo");
   }
}
class Doberman extends Dog{
   void  bark(){
        System.out.println("GRRRR!  WO WO WO!");
   }
}
class Terrier extends Dog{
   void  bark(){
        System.out.println(" yap! yap! yap! ");
   }
}
```

When we declare the `Beagle`, `Doberman`, `Terrier` classes, we don't need to re-declare all the fields of the `Dog` class. These fields are automatically inherited, because of the keyword `extends`. We also don't have to re-declare all the methods. We *can* redefine them though. For example, we have redefined the method `bark` for the sub-classes above. The method `bark` in the subclasses is said to *override* the method `bark` in the `Dog` superclass. More on that below.

Overriding $\neq$ overloading

Overriding a method (namely a subclass redefines a method that is defined in its superclass) is different from overloading it. When a subclass method *overrides* the superclass method, the signatures of the two methods are the same, by definition – namely the method name, return type, and the types of formal parameters are the same. When you *overload* a method, the method name is the same but signature changes, namely the type and/or number and order of formal parameters changes (and possibly the type of the return value changes). Overriding can only occur from a child class (subclass) to parent class (superclass), whereas overloading can occur either within classes, or between a child and parent class.

Overloading within a class

Let's first consider overloading of a method *within* a class. You should have seen examples of this in COMP 202, but I will briefly review it here. We take the example of a constructor method. When a class has multiple fields, these fields are often initialized by parameters specified in the constructor. One can make different constructors by having a different subset of fields. For example, if I want to construct a new `Dog` object, I may sometimes know the dog's name and sometimes I may know the dog's name and the owner's name, and sometimes neither, so I use different constructors in each case.

```
public  Dog(String dogName, String ownerName){
       :
}
public  Dog(String dogName){
       :
}
public  Dog( ){
}
```

The last of these constructors is the default constructor which has no parameters. In this case, all numerical variables (type int, float, etc) are given the value zero, and all reference variables are initialized to `null`.

Finally, notice that the following constructors are actually the same and including the both will generate a compiler error. i.e. The parameter identifiers (`ownerName` vs. `dogName`) are not part of the signature. Only the parameter types matter for deciding if two methods have the same signature.

```
public  Dog(String ownerName){
       :
}
public  Dog(String dogName){
       :
}
```

Overloading between classes

We can use the term *overloading* if we have a method that is defined in a subclass and in a superclass, namely if the signatures are different. Such a situation can easily arise. The subclass will often have more fields than the superclass and so you may wish to include one or more of the these new fields as a parameter in the method's signature in the subclass. Or one of these new fields might replace one of the fields in the signature from the superclass. We will see examples of overloading in the lectures.

Constructor chaining

When an object of a subclass is instantiated using one of the subclass's constructor, the fields of the object are created and these fields include the fields of the superclass and the fields of the superclass'es superclass, etc. This is called *constructor chaining*. How is it achieved ?

The first line of any constructor is

```
super(...);   //  possibly with parameters
```

If you leave this line out as you have done in the past, then the Java compiler puts in the following (with no parameters):

```
super();
```

The `super()` command causes the superclasse'es constructor to be executed, which typically sets the fields of the superclass to some value and these fields and values are inherited by the subclass. Note that the superclass has its own `super(..)` statement, and so on, which causes the fields of *all* the ancestor classes to be initialized.

If the superclass has more than one constructor then the subclass can choose among them by including parameters of the `super()` call to match the signature (number and types of arguments) of the superclass constructor. In the following example, it is assumed that the class `Dog` has `String` fields that specify the name and owner and that the class `Beagle` merely inherits these fields (but doesn't redefine them). So, the `Beagle` constructors might be:

```
public  Beagle(){
       :                              //  calls Dog()
}

public  Beagle(String dogName, String ownerName){
      super(dogName, ownerName);   // calls  Dog(String dogName,
                                                String ownerName)
       :
}

public  Beagle(String dogname){
      super(dogname);              //  calls Dog(String dogname)
       :
}
```

A few notes:

- Java does not allow you to write `super.super`. So if a sub-class wants to explicitly invoke a method from the superclasses' superclass, this is not possible (in Java).
- The `super` keyword is fundamentally different from the `this` keyword. `this` is a reference variable, namely it references the object that is invoking the method. `super` does not refer to an object, but rather it refers to a class, namely the superclass.
- It doesn't make sense to talk about a subclass constructor overriding a constructor from a superclass, since a constructor is a method whose name is the same as the class in which it belongs and the name of the subclass will obviously be different from the name of the superclass.

This is just an introduction to how inheritance works in Java. To learn more, consult a Java textbook such as Lewis and Loftus, or Liang. There are hundreds of these Java textbooks sitting on the shelf in the Schulich library, crying because no one signs them out. These textbooks have lots of good examples. See also online tutorials such as `docs.oracle.com/javase/tutorial/java/IandI/subclasses.html`

The `Object` class

Java allows any class to directly extend at most one other class.[26] The definition of a class is of one of the two forms:

```
class MyClass


class MySubclass extends MySuperclass
```

where `extends` is a Java keyword, as mentioned above.

If you don't use the keyword word `extends` in the class definition then Java automatically makes `MyClass` extend the `Object` class. So, the first definition above is equivalent to

```
class MyClass extends Object
```

The `Object` class contains a set of methods that are useful no matter what class you are working with. An instantiation of any class is always some object, and so we can safely say that the object belongs to class `Object`. As stated under the `Object` entry in the Java API: the class "Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class." Notice that this statement uses the word "hierarchy" and, more specifically, it could have used the term *tree*. Class relationships in Java define a tree. The subclass-superclass relationship is child-parent edge. When we say that "every class has `Object` as a superclass", we mean here in that `Object` is an ancestor, namely it is the root of the class hierarchy.

The `equals( Object )` method

In natural languages such as English, when we talk about particular instances of classes e.g. particular rooms or particular dogs, it always makes sense to ask "is this object the same as that object?" We can ask whether two rooms or dogs or hockey sticks or computers or lightbulbs are the same. Of course, the definition of "same" needs to be given. When we say that two hockey sticks are the same, do we just mean that they are the same brand and model, or do we mean that the lengths and blade curve are equal, or do mean that the instances are identical as in, "is that the same stick you were using yesterday, because I thought that one had a crack in it?"

In Java, the Object class has an `equals( Object )` method, which checks if one object is the same instance as the other, namely if `o1` and `o2` are declared to be of type `Object`, then `o1.equals(o2)` returns true if and only if `o1` and `o2` reference the same object. For the class `Object`, the `equals()` method does the same thing as the "==" operator: it checks if two referenced *objects* are in fact the same instance.

For many other classes, you will want to overload the `equals()` method, namely use a less restrictive version of the `equals` method. You may want `x.equals(y)` to return true if and only if the objects are of the same class (i.e. type) and certain but perhaps not all fields of the objects are the same. (We met this issue last lecture with `Rectangle` and we will meet it again.) Note that I say overloading rather than overriding here. In the `HockeyStick` class, we might define an `equals(HockeyStick)` method, which might specify that two hockey sticks are equal if they have the same length:

[26] C++ allows for *multiple inheritance*, that is, a class can extend more than one superclass. This leads to complications, for example, if two superclasses have a method with a common name, which one gets inherited?

```
public equals(HockeyStick stick){
    return (this.length == stick.length)
}
```

Another example of the subtleties with the `equals()` method which you are familiar with is in the `String` class. You were probably told in COMP 202 that, when comparing `String` objects, you should avoid using the `==` operator and instead you should use `equals()`. The reason is that the `==` operator for `String` objects can behave in a surprising way. Here are a few examples:

```
String s1 = "sur";
String s2 = "surprise";
System.out.println(("sur" + "prise") == "surprise");    // true
System.out.println("surprise" == new String("surprise")); // false
System.out.println("sur" == s1);                         // true
System.out.println((s1 + "prise") == "surprise");       // false
System.out.println((s1 + "prise").equals("surprise"));  // true
System.out.println((s1 + "prise") == s2);               // false
System.out.println((s1 + "prise").equals(s2));          // true
System.out.println( s2.equals(s1 + "prise"));           // true
System.out.println(("surprise" == "surprise"));         // true
```

This behavior is a result of certain arbitrary choices made by the designers of the Java language and *it is not something you need to understand or remember.* As long as you use the `equals(String` method of the `String` class instead of `==` to compare Strings, you will be fine.

More generally, when you define an `equals( YourNewClass )` method in a class `YourNewClass`, you should ensure the following:

- `x.equals(x)` returns true
- `x.equals(y)` returns true if and only if `y.equals(x)` returns true
- if `x.equals(y)` and `y.equals(z)` both return true then `x.equals(z)` returns true
- if `x` is not `null`, then `x.equals(null)` returns false

`hashCode()` method

The class `Object` has a method called `hashCode()` which returns an integer. How this integer is defined is not part of the Java specification, but often it is related to the memory address of the object. For the Java Virtual Machine running on my laptop, the `hashCode()` method of an object returns a 24 bit number. Test the `Object.hashCode()` method on Eclipse (or whatever you are using). Create a `System.out.print(new Object())` and you will find that it it returns a 24 bit number, written as 6 hexadecimal symbols. See the Java API for the `Object` class for details. We will discuss the `hashCode()` method later in the course when we discuss what hashing is.

`clone()`

Another commonly used method in class `Object` is `clone()`. This method creates a different object, which is of the same class as the invoking object and which has fields that have identical values to those of the invoking object at the time of the invocation. Cloned objects are supposed to obey the following:

```
The expression ( x == x.clone() )   must return false.

The expression ( x.equals( x.clone() )  should returns true
                                        (suggested, but not required).
```

Note that the latter doesn't hold for `Object` objects. But that's ok, as one rarely uses `Object` objects and even more rarely clones them.

`toString()`

This method is commonly used to write out a description of an object, namely the values of its fields. As an author of the class, you are free to define `toString()` however you wish (as long at it returns a String).

In the `Object` class, the `toString()` method prints out the class name of the object, the @ symbol, and the hashCode of the object represented in hexadecimal. Test it out. Notice that if you define your own class e.g. `HockeyStick` and you don't override the `toString()` method from the `Object` class, then your class will inherit the `Object` class'es `toString()` method. So if the variable `myDog` references an object from the class `Doberman`, then `myDog.toString()` might return the string say "`Doberman@34a212`."

Advanced topic: class descriptors and objects

At the end of the lecture, I spent a few minutes discussing the `Class` class and how it relates to the `Object` class. Here I fill in some more of the details. *This material is mostly for your interest. It is not on the final exam, and it is included here only to give you a initial sense of how it works.*

First, note that a class definition includes various things: the name of the class, a list of fields and types, a list of methods and their signatures, and the instructions of each method, a reference to the superclass, and other info. When a program uses a particular class, the information about that class is stored in memory in a data structure called a *class descriptor*. This class descriptor is different from a `.class` file generated by a compiler when you compile your `.java file`. The class descriptor is a runtime object, which is created from the `.class` file. The class descriptor is not a file, but rather it is a data structure inside a running Java program. The information in a class descriptor does correspond to the information in a class file though.

Specifically, class descriptors are objects of a class called `Class`.[27] Also, note that the class descriptor of the `Object` class, and the class descriptor of the `Class` class are also class descriptors. They are instances of the `Class` class! So, we can objects that are instances of the `Object` class. We can have objects (class descriptors) that are instances of the `Class` class.

[27] In my opinion, it might have been better if the folks who wrote Java called this class "`ClassDescriptor`", since saying "an object of class `Class`" is more confusing than saying "an object of class `ClassDescriptor`".

How is this useful? In lecture 23, I mentioned several methods in the `Object` class. One that I did not mention is `getClass()` which returns a reference to to the class descriptor of the class that this object belongs to. So, the return type of `getClass()` is `Class`, since this method returns a reference to a class descriptor. For exmaple, `myDog.getClass()` would return a reference to the `Dog` class descriptor, which is an object of class `Class`.

The `Class` class itself has several methods, in particular, `getSuperClass()` whose return type is `Class`. This method `getSuperClass()` returns a reference to the class descriptor of the (direct) superclass. So, if `myDog` were a `Beagle`, then `myDog.getClass().getSuperClass()` would return a reference to the `Dog` class descriptor which is an object of type `Class`.

Notice that when we consider class descriptors as objects, we have to think of objects referencing each other in two different ways:

- "instance of" - an object is an instance of a class. This defines a reference from an instance object to a class descriptor object i.e. `getClass()`.
- "subclass of" - a class descriptor (a particular kind of object, namely one that is an instance of the class `Class`) will have a reference to its superclass descriptor, i.e. `getSuperclass()`. (The `Object` class is the sole exception.)

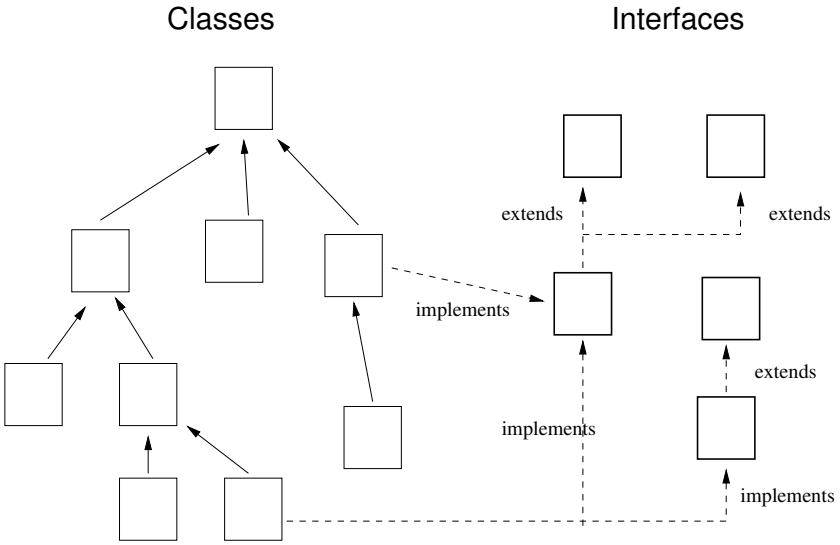
One more thing: a class is not the same thing as the descriptor of that class. A class is an abstract thing, like the set of integers. It exists in our heads, not in the computer. The descriptor of a class, though, is a concrete object in a running Java program. It is an instance of class `Class`. It corresponds to particular set of bytes/code in the memory of the computer. Whenever you see a statement like

```
Dog  myDog = new Dog():
```

you should think of two objects in memory: one is the descriptor of the `Dog` class, and the second is the newly created instance of this class. The `Dog` class descriptor must be created (loaded from the .class file) before the class instance can be created, since the `Dog` class descriptor contains information needed to create the `Dog` object instance.

Interfaces (revisited)

Earlier when we discussed Java classes and their inheritance relationships, we considered a hierarchy where each class (except `Object`) extends some other unique class. See below left. Thus Java classes define a tree, with each note having a reference to its parent[28] i.e. superclass. How, if at all, do Java interfaces fit into the class hierarchy?



As shown above right, an interface is another "node" in the inheritance diagram. We used dashed arrows to indicate inheritance relationships that involve interfaces.

- If a class `C` implements an interface `I` then we put a dashed arrow from `C` to `I`. Recall that a "class implements an interface" means that the class provides the method body for each method signature defined in the interface.
- One interface (say `I2`) can extend another interface (say `I1`). This means that `I2` inherits all the method signatures from `I1`. We don't need to write the method signatures out again in the definition of `I2`. In the class diagram, we would put a dashed line from `I2` to `I1`.
- Although each class (other than `Object`) directly extends exactly one other class,[29] a class `C` can implement multiple interfaces. The parent interfaces can even contain the same method signature. This is no problem since the interfaces only contain the signatures (not the bodies), so there can be no conflict. We would say:
  `class C2 extends C1 implements I1, I2, I3`

We have seen examples of interfaces in lecture 22. The following example is used to illustrate one of the limitations of interfaces, and motivates the use of abstract classes discussed next.

[28] But nodes do not have references to their children, i.e. subclasses.

[29] If this 'unique parent' constraint were not in place, and a class `C` were allowed to extend multiple classes (say **A** and `B`), then it could happen that there is method conflict – superclasses **A** and `B` could contain a method with the same signature (but with different bodies). Which of these methods would an object of class `C` inherit?

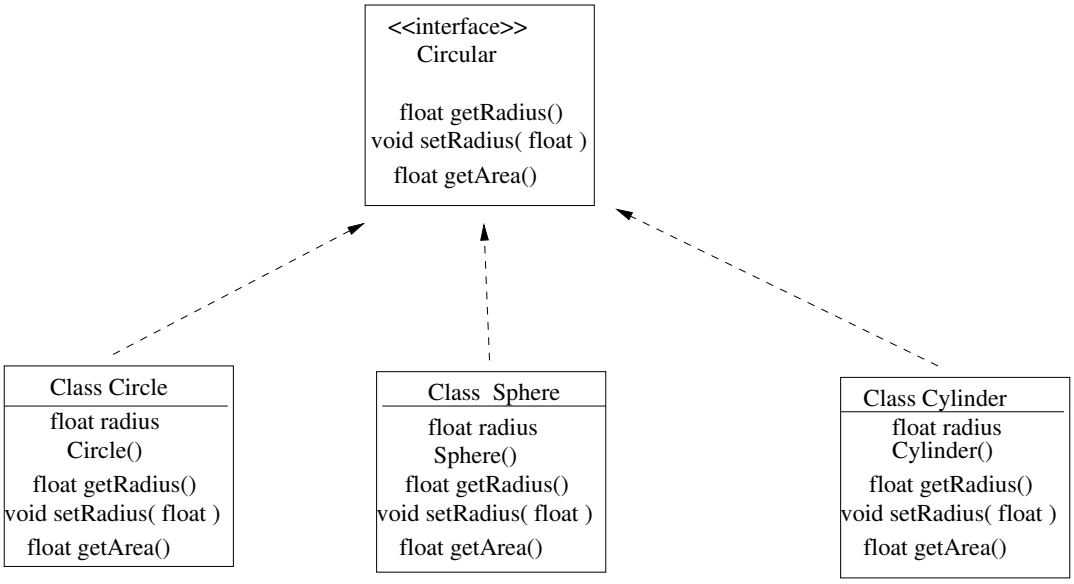
Example: Circular

Many geometrical shapes have a `radius`, for example, of a circle, sphere, and cylinder. Suppose we wanted to define classes `Circle`, `Sphere`, `Cylinder` of shapes that have a radius. In each case, we might have a private field `radius` and public methods `getRadius()` and `setRadius()`. We might also want a `getArea()` method.

We could define an interface `Circular`

```
public interface Circular{
    public double getRadius();
    public void   setRadius(double radius);
    public double getArea();
}
```

and define each of these classes to implement this interface. The problem with such a design is that we would need to define each class to have a local variable `radius` and (identical) methods `getRadius()` and `setRadius()`. Only the `getArea()` methods would differ between classes. We could do this, but there is a better way to deal with these class relationships.



Abstract classes

The better way is to use a hybrid of a class and an interface in which some methods are implemented but other methods are specified only by their signature. This hybrid is called an `abstract class`. One adds the modifier `abstract` to the definition of the class and to each method that is missing its body. For example:

```
public abstract class Circular{

    private double radius;

    Circular(){};

    Circular(double radius){
       this.radius = radius;
    };

    public double getRadius(){
        return radius;
    }

    public void   setRadius(double radius){
        this.radius = radius;
    }

    public abstract double getArea();
}
```

This abstract class has just one abstract method that would need to be implemented by the subclass `Circle`, `Cylinder`, or `Sphere`.

Note that the subclass `Circle` might also have a method `getPerimeter()`. Such a method would make no sense for a `Sphere` or `Cylinder`. Similarly, `getVolume()` would make sense for a `Sphere` and `Cylinder`, but not for a `Circle`.

An abstract class *cannot* be instantiated. However, abstract classes do have constructors. This seems like a contradiction, but it is not. Abstract classes are extended by concrete subclasses which provide the missing method bodies. When these subclasses are instantiated, they must inherit the fields and methods of the superclass. In particular, the values of the inherited subclass fields are set by the superclass constructor (either via an explicit `super()` call, or by default). Thus, even if the superclass is abstract, it still needs a constructor.

```
public class Circle extends Circular{

    Circle(double radius){
        super(radius);
    }

    double getArea(){
        double  r = this.getRadius();
        return  Math.PI * r*r;
    }
}
```

```
public class Cylinder extends Circular{
    double height;

    Cylinder(double radius, double h){
        super(radius);
        this.height = h;
    }

    double getArea(){
        return 2* Math.PI * r * height;
    }
}
```

Abstract classes also appear in class hierarchies/diagrams, along with interfaces as above:

- a class (abstract or not) "implements" an interface
- a class (abstract or not) "extends" a class (abstract or not)

In general, that a class cannot extend two abstract classes. The reason for this policy is the same as why a class cannot extend two classes – namely if the two superclasses were to contain two versions of a method with the same signature then it wouldn't be clear which of these two methods gets inherited by the subclass.

One last fact is that one can declare variables to have a type that is an abstract class, just as one can declare a variable to be of type class or of type interface. It will become more clear why, when we discuss polymorphism in an upcoming lecture.

Casting

You are already familiar with the basics of primitive types and how conversions can occur between them. Primitive types are ordered from "narrow" to "wide", for example, `char` widens to `int` and the following go from narrow to wide:

`byte`, `short`, `int`, `long`, `float`, `double`.

Widening conversions occur automatically, but narrowing conversions requires an explicit *cast*, otherwise you will get a compiler error.

```
char   c = 'g';
int    index = c;            // widening conversion
       c = (char) index;     // narrowing conversion

int    i = 3;
double d = 4.2;
       d = i;    // widening conversion (as part of an assignment)
       d = 5.3 * i;     // widening conversion (by "promotion")
       i = (int) d;     // narrowing conversion (by casting)
       float f = (float) d;    //  "
```

Narrowing typically leads to an approximation. For example, when you convert from a `float` to an `int`, you discard the fractional part. Interestingly, though, approximations can occur with widening conversions as well. How to see this? Note that there are 32 bits used to represent an `int` and 32 bits used to represent a `float`. This means that you can represent 2^{32} different possible values of each. Most `float`'s have a fractional part[30]. It turns out that most `int` values cannot be represented exactly as `float` values either.

We use similar concepts of "narrowing" and "widening" in a class hierarchy as well. If class `Beagle` extends class `Dog`, then class `Beagle` is narrower than `Dog`, or equivalently, `Dog` is wider than `Beagle`. In general, a subclass is narrower than its superclass; the superclass is wider than the subclass.

One possible confusion is that an object of a subclass typically has more fields and methods than an object of its superclass. So if you think of the relative "size" of the object (the number of fields and methods) then you will notice that the narrower object is bigger. This is the opposite of what generally happens with primitive types, where the wider type usually uses the same or more bytes than the narrower type.

There is another important difference to keep in mind between primitive and reference types. When we convert from one primitive type to another, in fact we perform an operation in which one binary representation of a value is replaced by another binary representation, possibly with a different number of bits. For example, a double uses 64 bits whereas a float uses only 32, and so converting from a float to a double (or double to float) requires re-coding the bits. *Casting reference types is different, however, since no change occurs to the referenced object.* Rather, the cast only tells the compiler that you (the programmer) expect the object to be a certain type at runtime, and specifically that you want to apply a certain method that is defined objects of that runtime class. A few examples below will illustrate this idea.

First, though, here is a bit more terminology. We cast *downwards* ("downcasting") when we are casting from a superclass to a subclass, and we cast *upwards* (upcasting) when we cast from a subclass to a superclass. Upcasting occurs automatically, and so it is sometimes called *implicit casting*. We have seen upcasting before, e.g.

```
Dog  myDog = new Beagle();
```

This is analogous to:

```
double  myDouble = 3;   //    from int to double.
```

We have not seen downcasting before (at least not for reference types). We will see it below.

Example 1

```
Dog  myDog = new Doberman();  // Upcasting.
  :
Beagle  myBeagle = myDog;     // Compiler error.
                              // (implicit downcast Dog to Beagle not allowed).
```

[30] You will learn exactly how floats are defined in COMP 273.

```
Beagle  myBeagle = (Beagle) myDog;   // Allowed (though runtime error
                                     //  would occur if myDog references
                                     //  a Doberman object).
```

Example 2

```
 Dog  myDog = new Beagle();
 myDog.hunt();                  // Gives a compiler error, if hunt() is
                                // defined in Beagle but not in class Dog.

((Beagle) myDog).hunt();  //  Explicit cast ok and no compiler error.
                          //  But if myDog references a Doberman at
                          //  runtime, then you get a runtime error.

//  Alternatively, you could do the following which would not
//  produce a runtime error (if myDog references a Doberman) and
//  instead just wouldn't get executed.

if (myDog instanceOf Beagle){
   ((Beagle) myDog).hunt();
}
```

A question came up in class about the rules of the `instanceof` operator. This operator takes two arguments: the first is a reference type variable `var`; the second is a class `C`. The operator returns true if and only if the object referenced by `var` is an instance of the class `C` or any class that extends `C`. The idea behind this rule is simple: we use the `instanceof` operator to verify whether the object referenced by `var` indeed has access to all the methods of `C`, e.g. whether or not `myDog` can invoke `hunt()`.

A related example came up in class. One student suggested a class `HuntingDog` such that `Beagle extends HuntingDog` and `HuntingDog extends Dog`. Suppose the method `hunt()` were defined in HuntingDog.

```
Dog  myDog = new Beagle();
if (myDog instanceOf HuntingDog){  // returns true
   ((Beagle) myDog).hunt();        // compiler allows this.
}
```

Polymorphism (runtime)

In the last few lectures, we have seen that the declared type of a reference variable does not entirely determine what classes of object the variable can reference at runtime. At runtime, a variable can reference an object of its declared type, or it can also reference an objects whose class is a subclass of the variable's declared type (if the declared type is a class), or if the declared type is an interface then the variable can reference objects whose class implements the interface.

This property, that the runtime type is more general than the declared type, is called *polymorphism*[31]. We sometimes refer to the declared type, which is determined at compile time, as the *static* type), to distinguish it from the *dynamic* type which is defined at runtime.

Last class we concentrated on the issue of *type checking* which is done by the compiler (and is typically called *static type checking*. We looked at when implicit casting was good enough and when we needed to do explicit casting. We also looking at the `instanceof` operator which is used for dynamic type checking, namely checking the class of an object at runtime. Let's now assume a program has compiled and focus on the question of which method is invoked. The method used is the one defined by the object that invokes it. This choice is sometimes called *dynamic dispatch*. There are a few different cases to be aware of here.

The first case is what I discussed above: when an object invokes a method (at runtime, i.e. we are talking about an object), the method is determined by the class that the object belongs to. The object knows what class it belongs to. This information is stored in the object in the `class` field that references the class descriptor (which is an instance of the `Class` class – see end of lecture 23, where I have added notes about it). A programmer can access this field with the `Class getClass()` method, which it inherits from the `Object` class.

Consider, for example:

```
boolean  b;
Object  it;
   :
if (b)
   it = new float[23];   // an array of floats
else
   it = new Dog();
System.out.print(it.toString());                              (*)
```

At compile time, we cannot say which `toString()` method should be used, since we don't know the value of `b`. Rather, the method must be determined at runtime, when (*) is executed and `it` references either a `float[]` or a `Dog`. In each case, there will be a `toString()` method used which is appropriate for the object.

A second issue that arises in polymorphism is that subclasses can override methods from the superclass and sometimes it is not so obvious which method gets applied. Let's just look at a simple example here.

Consider a method `threaten()` in the `Dog()` class. This method calls `showTeeth()`, which is defined in the `Dog` class only, and `bark()` which is defined in both the `Dog` and subclasses of `Dog`, in particular, `Doberman`.

[31] from Latin: poly means "many" and "morph" means forms

Consider what happens when a `Doberman` object invokes the `Dog` method `threaten()`. Which version of `bark()` is defined for Doberman objects ? To reason about this, you must understand that the method `bark()` in the `Dog()` class is invoked without specifying which object is invoking it. In the code, however, there is an implicit `this` refererence which the compiler fills in, so the call to `bark()` within `threaten` is really `this.bark()`. Thus, if the caller is a `Doberman` object, then the `bark()` that is used is `Doberman.bark()` rather than `Dog.bark()`.

```
class Dog  {
   String        name

   void    Dog(){ ..  }

   void    threaten{
              showTeeth(){  .. } );    //   this.showTeeth()
              bark();                  //   this.bark()
    }

   void    bark(){
        System.out.println("woof");
   }
}

class Doberman extends Dog{

   Doberman(String name){
      super(name);          //  alternatively,  "this.name = name;"
   }

   void  bark(){
      System.out.println("GRRRR!  WO WO WO!");
   }

   void  barkLikeDog{      // generic Dog impersonator
      super.bark();
   }
}
```

Note that if you wanted a `Doberman` object to bark like a `Dog`, you could do this by explicitly writing `super.bark()`. The `Doberman` object would then go to its superclass which is `Dog` and use the `bark()` method from the `Dog` class. See the `Doberman.barkLikeDog()` above.

Finally, in the lecture slides, I discussed walked through the different steps of the `main()` method above and showed how the call stack evolves. I suggest you do it for yourself and then verify your answer by checking the slides. Note that the `Doberman(String)` constructor calls `super(String)` so don't forget about that one.

Graphical user interfaces (GUI)

Most applications that you work with provide a graphical interface. These include IDE's such as Eclipse, web browsers such as Firefox, mail programs such as Thunderbird, Word processors, etc. You are so used to interacting with GUIs that you probably rarely think about how they work and how you might you write one.

We will now spend a few lectures looking at some of the object oriented concepts that are involved in creating such interface. The topic is large and we will only cover the basics concepts and techniques. This should be enough to spark your interest and get you going.

Containers and components

The part of the display screen where the input and output for an application occurs is called a *frame*. A frame is a window which has a minimize, maximize, close icons at the top right corner. A frame often consists a number of regions called *panels*. We think of frames and panels as *containers* in that they contain *components* such as images and text labels that we want to display, buttons that we can click on, and areas where we can enter text, etc. When we say "component", we mean something that has a position and size in a given frame or panel. A component either displays information or it allows the user to input something, or both.

If you are reading this document with acroread, for example, then the frame is the acroread window. There are several panels. There is the panel containing the document. But there is also a panel with icons at the top which allow you to step through pages of the document, zoom, etc. There may be another panel that has icons that allow you to print the document, add a sticky note, highlight text. And there is the panel which has the File/Edit/View/Window/Help tabs which produce dropdown menus when you click on them.

A container can contain components, but a container itself is also a component – it is special type of component that contains other components. As such, containers and components have a tree structure such that internal nodes are containers and leaves are either components (or empty containers), and the "A contains B" relation is a (parent/A, child/B) edge. This property is not something we will emphasize, but you should be aware of it nonetheless.

When you are creating a GUI in Java, the containers and components are all objects and so they need to be instantiated. You also need to add the components objects to the containers. For example, you can add a component directly to a frame. Or you can add a component to a panel, and then add the panel (now as a component) to the frame.

Let's now turn to some of the Java classes that are involved. The Java classes that you use to write a graphical user interface are all in `javax.swing.*` and `java.awt.*` where the latter stands for "abstract windowing toolkit". We will just refer to these GUI tools simply as "swing". An excellent reference for swing is the Java tutorials
`docs.oracle.com/javase/tutorial/ui/index.html`

`JFrame`

A `JFrame` is the main container for an application. It defines the application window that you are used to, i.e. it has a little X in the top right corner that allows you to close it, as well as a max and

min icon, and a title. The frame can be moved around on your screen by dragging with the mouse, and it can also be resized.

The constructor of the `JFrame` sets the initial size and other properties of the frame. Some of the methods of `JFrame` are `setVisible, setDefaultCloseOperation, setSize, setBounds, pack`.

JPanel

Panels are instances of the `JPanel` class. Panels are used to group together a set of components. We typically extend the `JPanel` class and define a particular panel that we wish to add to our frame. For example, if we want to make a form for people to fill out, then we might write a `MyForm` class that extends `JPanel`. It may seem strange to define a class for this, since often we only have one instance of this extended `JPanel` class. But that's how it works, so we have to get used to it.

Note that we can also extend the `JFrame` class if we wish and by define a subclass that has a particular set of `JPanel` objects within it. We then instantiate the `JPanel` objects and add them to the extended frame class. We can also extend both `JFrame` and `JPanel`.

JComponent

There are many components classes that we can instantiate and add to frames and panels. Here are some examples:

- `ImageIcon` - a little graphic/picture e.g. a garbage can to represent "delete"
- `JLabel` - text or an image
- `JButton` - allows you to click on it with a mouse
- `JCheckBox` - allows you to select from several options by clicking with your mouse, which marks the box with an X (or some other symbol)
- `JRadioButton` - similar to JCheckBox but now you must choose one and only one of a set of buttons (It is called a radio button because this is how old car radios used to work. You could select buttons which were set to tune in particular radio stations.)

  Note that `JButton`, `JCheckBox` and `JRadioButton` are often grouped together to make a `ButtonGroup`.
- `JTextField` - a region where you can type one line of text
- `JPasswordField` - like a textfield, but when you type in the text, the field shows only a symbol like * for each character
- `JTextArea` - a region where you can type multiple lines of text
- `JComboBox` - a set of items in a dropdown menu from which a single item is selected

LayoutManager

You might assume that when you make a GUI, you are responsible for positioning exactly where the components go in the frame and in each panel. This is not the case, though. One reason is that it is often common for users to resize windows and the programmer doesn't want to have to specify exactly where each component goes whenever the window is resized.

The positioning of the components is done automatically by an object known as a `LayoutManager` which has private methods that decide where to position each component, based on the user's choice of window size. Specifically, each `JFrame` and `JPanel` has a private field which references a helper object called its `LayoutManager`. The layout manager for the container has methods for positioning the components in that container.

`LayoutManager` is an interface which is implemented by Java classes `FlowLayout`, `GridLayout`, `BorderLayout`. If you want a container to use a "flow layout" manager. Then, within the container class (`JFrame` or `JPanel`) you call:

```
setLayout( new FlowLayout() )
```

The `setLayout` method is method in the `JFrame` and `JPanel` class. When we extend either of these container classes, we put the above method call in the constructor of the extended class. Note that there is an implicit `this` in the above method call.

Once the `LayoutManager)` has been assigned to a container, it remains there (unless it is reset, which is not typically done). Also note that, as a programmer, you don't declare a reference variable for the layout manager of a container. A hidden (private) one is already there. Once you set the layout manager, you don't need to refer to it again.

Event driven programming

Last lecture we looked at containers and components and how they are used to define the visual elements of a graphical user interface (GUI). However, the interface did not do anything useful. It just showed how the components could be arranged in panels and a frame. We would like to use the interface to *interact* with a program. For example, when we push on a button, we would like this button press to cause the program to do something. The program might compute something and display a result in one of the text fields, for example. (A calculator is the most obvious example, but in fact nearly everything we do with computers involves interaction with the user interface.)

Some (but not all) components of an interface can have actions done to them. For example, buttons can be clicked, checkboxes can be checked, text fields can be typed into. What is the Java mechanism for noticing that such actions occur and for responding ? What classes are involved? What is the programmer responsible for specifying, and what is automatic handled by the JVM and hidden from the programmer?

Big Picture: Java application + JVM $\leftrightarrow$ OS $\leftrightarrow$ hardware

Before answering the above questions, it may be helpful to zoom out and consider the big picture. There two different perspectives on what is happening when a user interacts with a program using a GUI. The first perspective is that of the user who interacts directly with the *hardware*, namely the input devices (keyboard and mouse, typically) and the output device (display screen). The user moves the mouse in order to position a cursor on the display, and clicks the mouse or presses keys on the keybard when the cursor is located particular positions in order to send position-dependent messages to the computer. We can think of the mouse motions and clicks and key presses as input "events".

The second perspective is that of the programmer or more generally the *software*. When a Java program runs, many objects are generated. Among these are event objects that represent the user events that I mentioned above, namely the mouse motion and clicks. These event objects then affect what the Java program does in a manner I will discuss below.

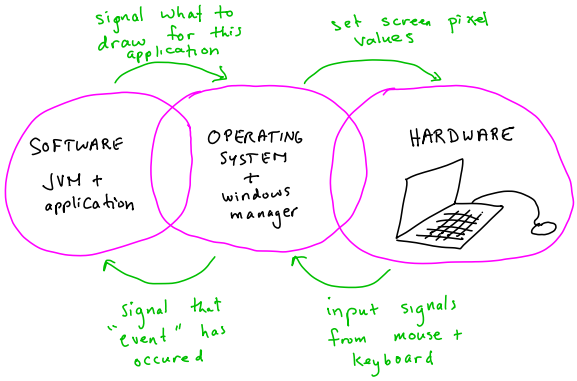
Between the hardware and software is the *operating system* (OS). You think of the operating system as software but it is more than that. At the lowest level of the computer, where the bits move from one electronic component to another, it becomes difficult to draw the boundary between operating system and hardware (as you will appreciate if you take COMP 273). Similarly, your Java application runs on a particular computer, there is a fuzzy region where the Java code must be translated into instructions that run on that particular computer and again there is a fuzzy region, now between the Java Virtual Machine software that is running the application and translating the application code into code that runs on the particular computer, and the operating system which is involved in managing these "native" machine instructions and coordinating with other applications lower level processes that might be running on the computer.

Returning now to GUIs, let's consider what happens when the user takes some action like clicking on the mouse. This physical event causes a signal to be sent from the mouse to the computer. The operating system responds to this event, by telling the Java Virtual Machine software that such an event has occured. The JVM then instantiates an event object (discussed later). Thus, there is a dual notion of an event: (1) there is a physical event i.e. the user's action, and (2) there is an object that is instantiated in a running Java application which encodes what the physical event

was, including the type of event and other parameters such as the exact time the event occured.

In the examples I presented last class, nothing happens when the user clicks on a mouse, in that there no changes changes visible on the display. Today I will tell you about Java classes that handle such events. To briefly summarize, classes that handle such events are called listeners and they have methods that are invoked as a response to events. Listeners can alter objects in the program, both component and container objects, as well as other objects that may be part of an application and have nothing directly to do with the interface.

You should also be aware of what happens at the output end (though we will not discuss this). The operating system regularly monitors the GUI component and container objectss. The OS (in particular, the "window manager") is responsible for determining the pixel values which are sent to the display i.e. the interface for the user. In a nutshell, the JVM gives instructions to the operating system of what are the objects to be drawn and what is their layout, and the OS window manager figure out how to draw them on the screen (when there are many other applications that are trying to draw to the screen too). See the sketch below which was taken from the slides.



Events

As discussed above, we have two notions of an "event". There are the physical user actions of interacting with hardware, and there are the corresponding actions in the software. For the latter, there is a class called `EventObject`. So pressing on a button causes an `EventObject` object to be instantiated. How the EventObject object is instantiated will remain hidden from you; you just have to accept that it happens automatically.

You likely won't work with the `EventObject` class. Rather, you will work with one of the many classes that extend `EventObject` such as `ActionEvent`, `MouseEvent`, `MouseMotionEvent`, `KeyEvent`. As the user manipulates the mouse or presses keys on the keyboard, events objects are automatically constructed by the JVM. (This happened with the examples from last lecture too, even though there was no response to these events.) As the event objects are instantiated, they are placed in an event queue. When the JVM has time, it removes the event objects from the event queue one-by-one and processes them, as described below.

Each event is automatically associated with a unique component or a container, known as the *source* of the event.[32] For example, when a user positions the cursor over a screen button and clicks the mouse, an `ActionEvent` object is produced and the component that is associated with that

[32]The term "source" is somewhat misleading here, since it suggests that the source component/container *caused* the event or constructed the event. From what I have read, that is *not* how it works. Rather, the term "source" is

event object is the `JButton` object that corresponds to the screen button that the user pressed. For `MouseMotionEvent` objects, it is less obvious what the source is. If the cursor happens to be over a button, then the source will be the corresponding `JButton` object. But if the cursor is in the panel area outside the button, then the `JPanel` object will be the source.

How can you, the programmer, know what the source of an event is? The `EventObject` class has a method `getSource()`. You can get access to an event's source by invoking the event object's `getSource()` method, which returns the source object. In the online example code `DemoButtonListener` that counts button presses, you will see how this can be useful.

Listeners

What happens when the event object is removed from the event queue to be handled? There are lots of other objects that exist (`Dog`, `String`, `LinkedList` objects that are part of your program) as well as component and container objects that deal with the GUI part of the program. Any object that is supposed to react when an event is processed is called a *listener*.

Java defines several listener interfaces.[33] Here are a few of the ones you are more likely to see, along with their methods. (In some cases, I have not listed all the methods of the interface.)

```
interface ActionListener{
   void actionPerformed(ActionEvent e)   //  ActionEvent is a class
}

interface MouseListener{
       void mousePressed(MouseEvent e);
       void mouseReleased(MouseEvent e);
 }

interface MouseMotionListener{
       void mouseDragged(MouseEvent e);
       void mouseMoved(MouseEvent e);
 }

interface KeyListener{
       void keyPressed(KeyEvent e);
       void keyMoved(KeyEvent e);
 }
```

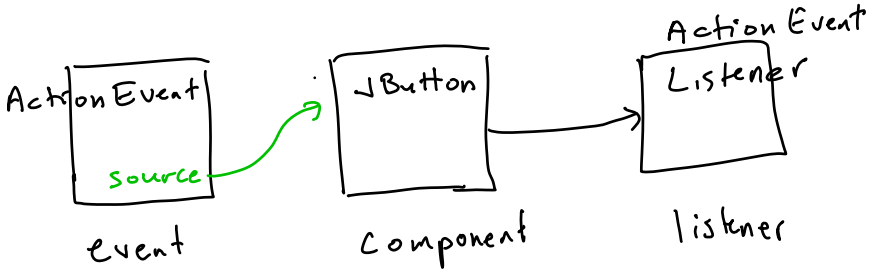
Any object can be a listener. A `Dog` can be a listener. A `JPanel` can be a listener – you just need to extend it to a class that implements a particular listener interface. You can define a separate listener class that implements some (or several) listener interface, and only does that!

here referring to the physical thing which is associated with the source object, e.g. a button or a panel on the screen, or a key press.

[33] These interfaces all extend the `EventListener` interface, which in fact is an empty interface!

Now let's turn to the mechanism by which events cause listeners do to things. This is perhaps the most conceptually difficult aspect of GUIs so read the following very carefully.

When the JVM removes an event object from the event queue to be handled, it doesn't directly invoke methods of a listener object. Instead, the JVM automatically looks up the source object for that event (using the event object's `getSource()` method) and then gets the source object to invoke the appropriate listener methods – see example below. The reason for this indirection (event → source → listener) is that the event object doesn't know what its purpose is. The event object only knows that some particular event happened (a particular button was pressed, or the mouse moved in some location, etc.) and it knows the source of that event, i.e. it has a source field and a `getSource()` method.



The next thing you need to know is that there may be multiple listeners for any event. In this case, the source of the event needs to invoke the appropriate methods of each of these listeners. Two things are needed for this to happen:

1. Each listener for an event needs to be on a special list, namely (by definition) the list of listener objects that is informed when such an event occurs;

   For example, suppose the variable `listener` references an `ActionListener` object, and suppose the variable `button` references a particular `JButton` object. Then, we can add the listener object to the listeners for the button press as follows:

   ```
   button.addActionListener(listener)
   ```

2. The source object tells each of the listeners that an event has occured. It does so by going through the list of listeners that have been added and invokes the methods that are defined for this type of event.

   For a button press, we have an `ActionEvent` and the listener is an `ActionEvenListener` and the method that is invoked is `actionPerformed(Event e)`, i.e. this is the method defined in the `ActionEventListener` interface – see previous page.

   To summarize, the JVM removes the event object from the event queue, then gets the source for that event, then goes through the list of listeners for that source, and then invokes the appropriate method. Think of it as follows:

```
ActionEvent event = ...  //  remove event from the event queue
for each listener in event.getSource()'s ActionEventListener list
     listener.actionPerformed(event)
```

Careful: the above is done by the JVM. It is automatic; as the programmer, you don't need worry about it. It is "under the hood".

Online code

At the end of the lecture, I showed several examples. The code will be made available next to these lecture notes. You should go through them and play with them a bit. Just reading these notes is not enough. You might also look through the slides, which have some figures that are not included in these notes. (And of course you can read more in any basic Java book. There are hundreds in the Schulich library.)

Java modifiers

Some of the material today is taken from the Readings BackgroundJava.pdf which I posted early in the course. I have added further details, some of which involve concepts from inheritance.

Packages

A *package* is a particular set of classes. Go to the Java API

`http://java.sun.com/j2se/1.5.0/docs/api/`.

and notice that in the upper left corner of that page is a listing of dozens of packages. A few of them that you are familiar with are:

- `java.lang` - familar classes such as `Object, String, Math, ...`
- `java.util` - has many of the data structure classes that we use in this course, such as `LinkedList`, `ArrayList`, `HashMap`

The full name of a class is the name of the package following by the name of the class, for example `java.lang.Object` or `java.util.LinkedList`. You can have packages within packages e.g. `java.lang` is a package within the `java` package.

You can also make your own packages, of course. For example, the classes `Beagle` and `Dog` might belong to package `comp250.lectures`, and so the full name of the classes would be `comp250.lectureExamples.Beagle` and `comp250.lectures.Dog`.

Packages have a tree structure. Packages are directories, typically internal nodes[34]). The classes within a package are leaves. The issue of how file systems and packages are organized is important in practice, but I did not discuss it in class. If you want to learn about package trees and how they correspond to file directories (and to www URLs) then you read through the following:
`http://cs.armstrong.edu/liang/intro6e/supplement/packages.pdf`]

Visibility Modifiers

You are familiar with defining classes to be `public` and defining methods and fields to be `public` or `private`. You learned, for example, that fields in a class are usually defined to be `private`, and the fields are accessed using `public` getter and setter methods.

The question of `public` vs. `private` gets a bit more complicated when you consider inheritance relationships and packages.

Visibility modifiers for classes

If you want a class to be visible to all other classes, regardless of which package the other classes is in, declare the class `public`. If you want a class `B` to be visible only to class `A`, then define `B` inside `A` – so that `B` is an *inner class* or *nested class* of `A` – and declare class `B` to be `private`. This is only way it makes sense to have a private class.

[34]the only exception being if you have an empty package in which case it would be a leaf

Class visibility modifiers also determine whether a class can extend another. If a class `A` is declared without a modifier (package visibility) then this class can only be extended by a class `B` within the same package. If you want a class `B` to extend a class `A` from a different package, then `A` can be defined as `public`.

In fact, there is one other possibility, namely to define `A` to be `protected`. This allows `A` to be visible to an class in the same package (default package visibility), but it also allows `A` to be extended by classes in other packages. *I will not discuss the* `protected` *modifier further in these lecture notes. You don't need to know the rules of this modifier for the final exam.*

Visibility modifiers for fields and methods

Let's next consider modifiers for class fields and methods. Let's deal with methods first. Methods contain instructions which invoke methods. Suppose that method `mA` is within class `A` and method `mB` is within class `B` and that class `A` is visible to class `B`. (Note that visiblity is *not* a symmetric relationship. Its possible for class `A` to be visible to class `B` but for `B` to not be visible to `A`, e.g. `A` might be public and `B` might be package, and `A` and `A` might be in different packages.

For an instruction in method `mB` to invoke method `mA`, method `mA` in class `A` needs to be visible to class `B`. For this, it is necessary that class `A` is visible to class `B`. But this is not sufficient, since we also require that the method `mA` itself is visible. So we have a situation as follows. *Note that I am not specifying the visiblity of method* `mB` *here since it is irrelevant to the current question.*

```
_____  void   mB(  A  v ){

      v.mA();    //  or attempt to access field v.fA in class A
}
```

The left column of the table below shows the four possible method visibility modifiers for method `mA` or field `fA` in class `A`. The other columns consider whether the compiler allows `mB` to invoke method `mA`, as in the code above. A `1` means yes and `0` means no.

fA/mA modifier	A & B same class	A&B same package, but diff classes (even if B is is subclass of A)	A and B in diff packages
public	1	1	1
(package)	1	1	0
private	1	0	0

One possible source of confusion is that these visibility relationships are between classes. (This is easy to understand if we keep in mind that these visibility relationships as defined at compile time, not at runtime.) For example, suppose the `Dog` class has a private field:

```
private  String  name
```

Then any methods that are defined with the `Dog` class can use this field. Methods defined within the dog class do not have to use the `getName()` or `setName()` methods. For the example the `Dog` class might have a method

```
public void sayNameOfDog(Dog dog){
    System.out.println( dog.name );
}
```

by which a `Dog` could say its own name or the name of another `Dog`. Such a method can be inherited by a subclass e.g. `Doberman`, even though the instructions in the method use the private field of `Dog`. This might be counter-intuitive, since any method that is defined explicitly in the subclass (say `Doberman`) is *not* allowed to explicitly use private fields like `name` from the superclass `Dog` and instead it must use the getter/setter.

Use modifiers

`static`

The modifier `static` specifies that a variable or method is associated with the class, not with particular instances. It is possible to define a `static` class, but only in the special case that the class is an inner class.

More often, the `static` modifier is used for methods and fields. It is often used, for example, to keep track of the number of instances of a class.

```
static  int  numberOfObjects;              //  A constructor would increment
static  int  getNumberOfObjects(){         //  this variable.
    return numberOfObjects;
}
```

Another example of a `static` method is `main()`. Note that you invoke this method when you "run the class" and you do this without instantiating the class.

A `static` method or field is often called a "class method" or field. Examples of static methods are `sin()` or `exp` in the `java.lang.Math` class. Note that to use such methods, in your class definition you would specify that you are using the package `java.lang` and then you would invoke the method just by stating the classes name and the method e.g. `Math.cos( 0.5 )`. Note that the class name is used instead of a reference variable (to an object) which is why we say that `static` methods are "class methods" rather than "instance methods".

`final`

The `final` modifier means different things, depending on whether it is used for a class, a method, or a variable.

- `public final class A` - means that the class cannot be extended (a compiler error results if you write `B extends A`.)
- `final double PI = 3.1415`. The value cannot be changed. Note that you need to assign a value for this to make sense.
- `final int myMethod()` - means that the method (which happens to return an `int`) cannot be overridden in a subclass.

  `String` and `Math` are example of a `final` class.

Priority Queue

Recall the definition of a queue. The fundamental property is that the next item to be removed is the one that was in the queue longest. A natural way to implement such a queue was using a linear data structure, such as a linked list or a (circular) array.

A *priority queue* is a different kind of queue, in which the next element to be removed is defined by (possibly) some other criterion. For example, in a hospital emergency room, patients are treated not in a first-come first-serve basis, but rather the order is also determined by the urgency of the case (perhaps in combination with other factors, including how long the patient has been waiting). To define the next element to be removed, it is necessary to have some way of comparing any two objects and deciding which has greater priority. Then, *the next item to be removed is the one with greatest priority.* Careful though: with priority queues, one typically assign low numerical values to high priorities. Think: "my number one priority", "my number 2 priority", etc.)

One way to implement a priority queue is to maintain a sorted list of the elements in the queue. This could be done with a linked list or array. Each time an item is added, it would need to be inserted into the sorted list. If the number of items is huge, then this would be an inefficient representation since inserts and removals are $O(n)$. A second way to implement a priority queue would be to use a binary search tree. The item that is removed next is found by the `findMinimum()` operation. This would be a better way to implement a priority queue than the linear method, since an insertion and deletion tend to take $\log n$ steps rather than n steps, if the tree is balanced. However, there is no guarentee on that. A binary search tree can have long paths.

ASIDE: Java's `PriorityQueue` class

The Java API defines a class `PriorityQueue<T>` where type `T` implements `Comparable`. The `PriorityQueue<T>` class has several methods, including an `add` and `remove` method. You can check out the Java API for the details. If you do, you'll note that there are some subtle differences between these methods. I won't go into them here, since there are more fundamental things to learn now, namely.....

Heaps

The most common way to implement a priority queue is to use a special kind of binary tree, called a *heap*. To define a heap, we first need to define a complete binary tree. We say a binary tree of height h is *complete* if every level l less than h has the maximum number (2^l) of nodes, and in level h all nodes are as far to the left as possible. A *heap* is a complete binary tree, whose nodes are comparable[35] and satisfy the property that *each node is less than its children.* This is the default definition of a heap, and is sometimes called a *min heap.* (A *max heap* is defined similarly, except that the element stored at each node is greater than the elements stored at the children of that node. Unless otherwise specified, we will assume the definition of a min heap in the next few lectures.) It follows from the definition that the smallest element in a heap is stored at the root.

As with stacks and queues, the two main operations we perform on heaps are `add` and `remove`.

[35] I am not distinguishing the nodes being comparable from the keys/elements at the nodes being comparable.

How many heaps are there?

Suppose we have letters `a, b, c, d, e, f, g`. How many heaps can we make with these letters?

The root has to be `a` since it is the minimum element. The letter `b` has to be in level 1 all elements other than `a` and `b` are greater than `b` and hence none of them can be `b`'s parent. Letter `c` does not have to be the sibling of `b`, though. Letter `c` could be a child of `b`, and in that case the sibling of **b** could be either `d` or `e`. *See the slides for several examples.*

Note that if you swap any two sibling nodes, you automatically preserve the heap property. *Note that swapping sibling nodes is different from swapping sibling keys.* If you swap nodes, you take the children with. For example, here I swap nodes b and c in the heap.

add

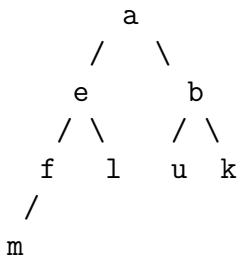
To add an element to a heap, we create a new node and insert it in the next available position of the (complete) tree. If level h is not full, then we insert it next to the rightmost element. If level h is full, then we start a new level at height $h + 1$.

Once we have inserted the new node, we need to be sure that the heap property is satisfied. The only problem could be that the parent of the node is greater than the node. This problem is easy to solve. We can just swap the elements of the node and its parent. We then need to repeat the same test on the new parent node, etc, until we reach either the root, or until the parent node is less than the current node.

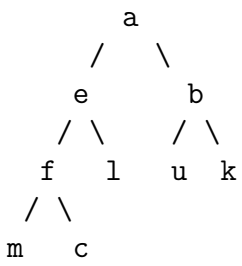
```
add(key){
   create a new node at next available leaf position
   cur = new node
   cur.key = key
   while (cur != root) && (cur.key < cur.parent.key){
       swapKey(cur, parent)
       cur = cur.parent
   }
}
```

You might ask whether swapping the key at a node with its parent's key can cause a problem with the node's sibling (if it exists). It is easy to see that no problem exists though. The sibling must be greater than or equal to its arent (since we have a heap), and so if the current node is strictly less than its parent, then (by transitivity) the node must be less than the sibling. So, swapping the node's key with its parent's key preserves the heap property with respect to the node's current sibling.

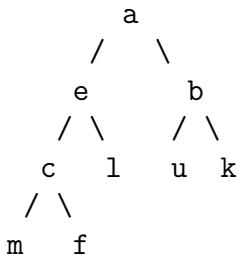
Here is an example. Suppose we add key `c` to the following heap.



We add a node which is a sibling to `m` and assign `c` as the key of the new node.

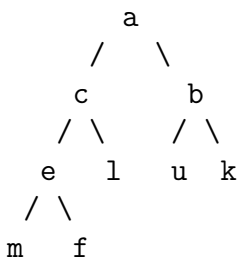


Then we observe that `c` is less than the key `f` of its parent, so we swap keys to get:



Could this move have violated the heap property, with respect to its former sibling `m` ? No. The sibling `m` was a child of `f` beforehand, and so we know that the sibling `m` was greater than or equal to `f`. But we are swapping `c` and `f` because `c` is strictly less than `f`. Thus, `c` is less than its former sibling `m`.

Now we continue up the tree. We compare `c` with the key in its new parent `e`, see that the keys need to be swapped, and swap them to get:



Again we compare `c` to its parent's key – but since is greater than `a`, we stop and we're done.

removeMin

Next, let's look at how we remove elements from a heap. Since the heap is used to represent a priority queue, we remove the minimum element, which is the root.

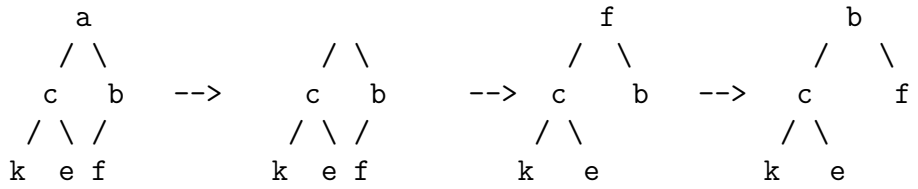
How do we fill the hole that is left by the element we removed ? We first copy the last element in the heap (the rightmost element in level h) into the root, and delete the node containing this last element. We then need to manipulate the elements in the tree to preserve the heap property that each parent is less than its children.

Starting at the root (which contains an element that was previously a leaf), we compare the root to its two children. If the root is less than its two children, then we are done. Otherwise, the root is greater than at least one of the children. In this case, we swap the root with the smaller child. Moving the smaller child to the root does not create a problem, since by definition the smaller child will be greater than the larger child.

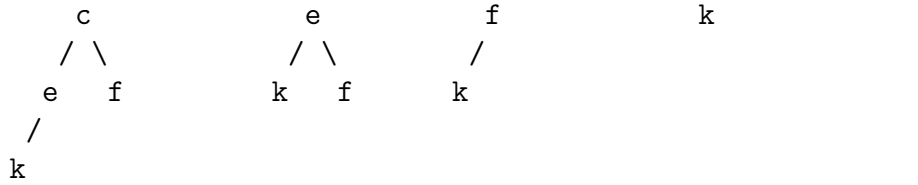
Here is a sketch of the algorithm:

```
removeMin(){
   remove last leaf node and put its key into the root
   node = root
   while ((node has at least one child) &&
          (node.key > leftchild.key) &&
          (node.key > rightchild.key))
       c = child with the smaller key
       swapKey(node, c)
       node = c
   }
}
```

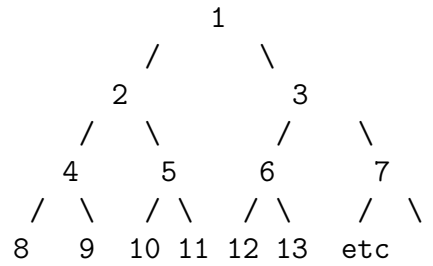
Here is an example:



If we continue applying `removeMin()` again until all the keys are gone, we get the following sequence of heaps (with keys removed in the following order `b, c, e, f, k`).



A heap is defined to be a complete binary tree. If we number the nodes of a heap by a level order traversal and start with index 1, rather than 0, then we get an indexing scheme as shown below.



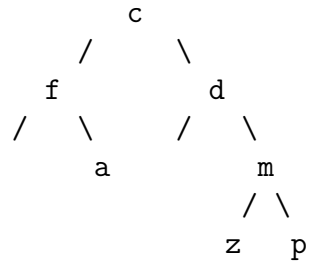
These numbers are NOT the keys stored at the node, rather we are just numbering the nodes so we can index them.

This indexing scheme gives a simple relationship between a node's index and its children's index. If the node index is i, then its children have indices $2i$ and $2i + 1$. Similarly, if a non-root node has index i then its parent has index $i/2$.

Implementing a heap using an array

It is very common to use an array to represent a heap, rather than a binary tree, and to use the parent/child indexing scheme described above rather than using nodes and parent, leftchild, and right child references. Note that the heap is still a complete binary tree. We will still be talking about "parent", "left child", and "right child". However, we will be using an array to represent the binary tree.

Also note that you can always use an array to represent a binary tree if you want, and use the above indexing scheme. However, the benefits of doing so are questionable when the binary tree is NOT a complete binary tree. For example, below is a binary tree (left) and its array representation (right) where - indicates null.




```
cfd-a-m------zp
```

Adding an element to a heap (array representation)

Suppose we have a heap with `k` keys which is represented using an array and we want to add a new key. Last class we sketched an algorithm doing so. Now we re-write that algorithm using the simple array indexing scheme. Let `a.size` be the number of keys in the array. These keys are stored in array slots 1 to `a.size`. (Recall that index/slot 0 is unused.)

```
add( a, key ){
   a.size = a.size + 1
   a[ a.size ] = key     // assuming array has room for another key
   upheap( a, a.size )
}
```

where the `upHeap()` helper method is:

```
upHeap(a, j){    //  assumes
   i = j
   while (i > 1  and a[i] < a[ i/2 ]){
      swapKeys( i, i/2 )
      i = i/2
   }
}
```

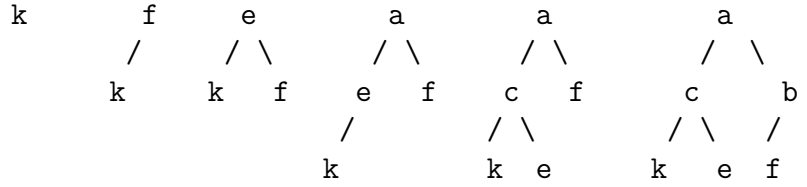
Building a heap

We can use the above `upHeap` operation to make a heap as follows. Begin with an array `a` whose keys are in positions 1 to `a.size`. The keys are in no particular order. If we just take the key `a[1]`, then this single key defines a heap. Then, given a heap with k keys, we upHeap the key k+1, giving us a heap out of the first k+1 keys. By induction, we end up with a heap of size n = `a.size`.

```
buildHeap(a){

//  INPUT:   an array a of unsorted keys, indexed from 1 to size (size > 1)
//  OUTPUT:  the keys of the array rearranged so that the array is a heap

  for (k = 2; k <= a.size; k++)
     upHeap( a, k );
}
```

Example

Let's suppose the heap is inititally empty and then we add items `k,f,e,a,c,b` in that order. Here is how the heap evolves.



Notation: floor and ceiling (rounding)

Recall the following notation from lecture 21 slides.

- $\lfloor x \rfloor$ is the largest integer that is less than or equal to x. $\lfloor\ \rfloor$ is called the *floor* operator.
- $\lceil x \rceil$ is the smallest integer that is greater than or equal to x. $\lceil\ \rceil$ is called the *ceiling* operator.

Note that for any positive integer $i \geq 1$, there is a unique integer l such that

$$2^l \leq i < 2^{l+1}$$

or

$$l \leq \log i < l + 1,$$

and $l = \lfloor \log i \rfloor$. In particular, if i is the index in the array representation of keys/nodes in a heap, then l is the level where you find i in the corresponding binary tree representation.

Let's have a look at the worse and base cases. If node i is at level l (the root is at level $l = 0$), then $l = \lfloor \log i \rfloor$. Thus, when we add node i to the heap, in the worst case we need to do $\lfloor \log i \rfloor$ swaps up the tree to bring the new element i to a position where it is less than its parent, namely we may need to swap it all the way up to the root. Since we are adding n nodes in total, the worst case number of swaps is:

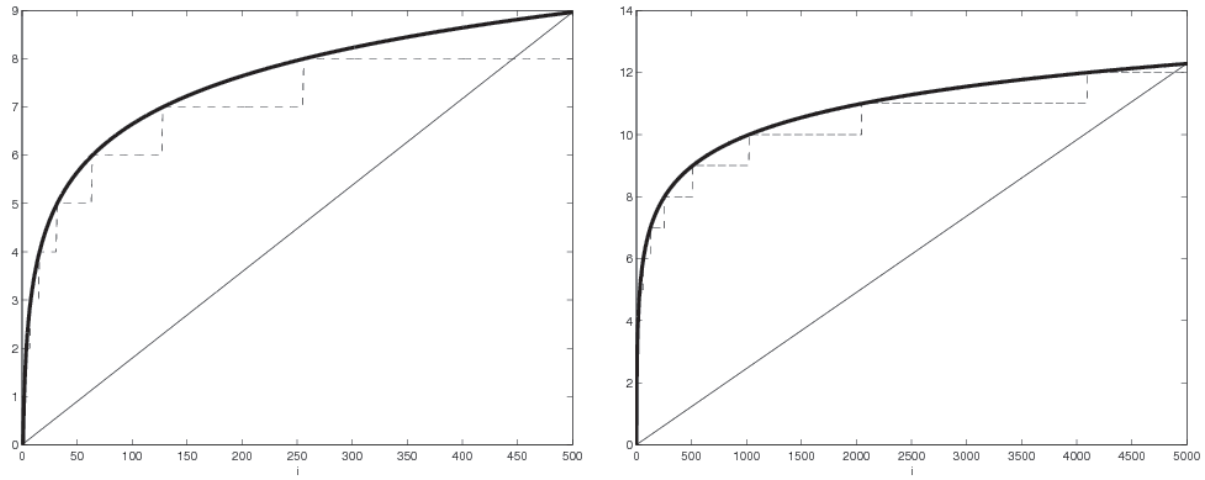
$$t(n) = \sum_{i=1}^{n} \lfloor \log i \rfloor$$

To visualize this sum, consider the plot below which show the $\log i$ (thick) and $\lfloor \log i \rfloor$ (dashed) curves up to $i = 500$ (left) and $i = 5000$ (right). The area under the dashed curve is the above summation. It should be visually obvious from the figures that

$$\frac{1}{2} n \log n < t(n) < n \log n$$

where the left side of the inequality is the area under the diagonal line from (0,0) to $(n, \log n)$ and the right side $(n \log n)$ is the area under the rectangle of height $\log n$. From the above inequalities, we conclude that $t(n)$ is both $O(n \log n)$ and $\Omega(n \log n)$. Note the reasoning here. I used a figure to convince you that the the above inequalities are true for large n, i.e. the upper and lower bound of $t(n)$. I then used the inequalities to make the big O and Ω statements about $t(n)$.

The *best case* for building a heap is that the initial array is already a heap! In this case, the $t(n)$ for the algorithm is proportional to n. The `buildheap(a)` algorithm still goes through the "for loop". For the k-th key, it checks if the parent of that key is smaller. The answer is "no" in each case, since we are considering the best case here. Thus, $t(n) = n$ which is both $O(n)$ and $\Omega(n)$. I emphasize that the $t(n)$ for the best and worst cases are different! You could think of them as $t_{worst}(n)$ and $t_{best}(n)$, if that helps.



removeMin and downHeap

Recall the `removeMin()` algorithm from last lecture. We can write this algorithm using the array representation of heaps as follows.

```
removeMin(){
  n = a.size
  key = a[1]      //  to be returned
  a[1] = a[n]
  downHeap(a, n-1)
  a.size = a.size - 1
  return key
}
```

This algorithm takes a heap as input, saves the root to be returned later, and then moves the key at position `a.size` to the root. The situation now is that node 2 and its descendants define a heap, and node 3 and its descendents define a heap. But the tree itself typically won't satisfy the heap property.

The `downHeap` method is then applied to move the root key down.

```
downHeap(a, n){
  i = 1
  while (2*i <= n){                 // There is a left child
    child = 2*i
    if child < n {                  // There is also a right child
      if (a[child + 1] < a[child])  // Is rightchild < leftchild ?
          child++
    }
    if (a[child] < a[ i ]){           // Do we need to swap with child?
       swapKeys(i , child)
       i = child
```

```
    }
  }
}
```

We already went over this algorithm last lecture. What is new here is that I am expressing it in terms of the array indices.

Heapsort

A heap can be used to sort a set of keys. First, we build a heap. Then, we are sure that the minimum key is the root. The idea of heapsort is to remove the minimum key from the heap, replace it by the last key and then downheap that key from the root. (Note the size of the heap is reduced by 1.) We use the freed slot in the array (the last key) to store the removed (smallest) key.

```
INPUT:  heap  a[1 .. n] i.e. minimum is a[1]
OUTPUT: a sorted array a[1 .. n]   (sorted from maximum to minimum)

for i = 1 to n{
    swapKeys( a[1], a[n+1 - i])
    downHeap( a,  n-i)
}
```

Note that after i times through the loop, the remaining heap is of size $n - i$ only, and the last i keys in the array stored the smallest i keys in the set.

Note that the final array will be sorted from largest to smallest. Is this a problem? Not at all, since we can reverse the keys in an array in $O(n)$ time, i.e. by swapping i and $n + 1 - i$ for $i = 1$ to $\frac{n}{2}$.

Example

The example below shows the state of the array after each pass through the for loop. The vertical line marks the boundary between the remaining heap (on the left) and the sorted keys (on the right).

```
  1   2   3   4   5   6   7   8   9
-------------------------------------
  a   d   b   e   l   u   k   f   w |
  b   d   k   e   l   u   w   f | a   (removed a, put w at root, ...)
  d   e   k   f   l   u   w | b   a   (removed b, put f at root, ...)
  e   f   k   w   l   u | d   b   a   (removed d, put w at root, ...)
  f   l   k   w   u | e   d   b   a   (removed e, put u at root, ...)
  k   l   u   w | f   e   d   b   a   (removed f, put u at root, ...)
  l   w   u | k   f   e   d   b   a   (removed k, put w at root, ...)
  u   w | l   k   f   e   d   b   a   (removed l, put u at root, ...)
  w | u   l   k   f   e   d   b   a   (removed u, put w at root, ...)
  w   u   l   k   f   e   d   b   a   (removed w, and done)
```

Some properties of complete binary trees

I begin today by considering some interesting properties of complete binary trees. At the end of the lecture, I will these properties to show you a remarkably fast algorithm for building a heap. *The slides have many pictures which should help you seen what's going on in this lecture.*

Sum of the depths of all nodes

One quantity that often comes up in tree algorithms is the average depth of nodes. The average depth is just the sum of the depths of all nodes, divided by the number of nodes. So we will ask the more basic question, what is the sum of the depths of all nodes? We specifically address the case of complete binary trees. To simplify the math slightly, let's assume that *all* levels of the tree are full, i.e. including level $l = h$ which is the height of the tree. Then,

$$n = 2^{h+1} - 1$$

and so

$$h = \log(n+1) - 1.$$

Let $t(n)$ be the sum of the depths of all nodes, Then (*see the slides*),

$$t(n) = \sum_{i=1}^{n} \lfloor \log i \rfloor.$$

We can rewrite $t(n)$ as a function of h and solve:

$$\begin{aligned}
t(h) &= \sum_{l=0}^{h} l\ 2^l \\
&= \sum_{l=0}^{h} l x^l, \qquad x = 2 \\
&= x \sum_{l=0}^{h} l\ x^{l-1} \\
&= x \sum_{l=0}^{h} \frac{dx^l}{dx} \\
&= x\ \frac{d}{dx} \sum_{l=0}^{h} x^l \\
&= x\ \frac{d}{dx} \frac{x^{h+1}-1}{x-1} \\
&= x\ \frac{(h+1)x^h(x-1) - (x^{h+1}-1)}{(x-1)^2} \\
&= 2\ ((h+1)2^h - 2^{h+1} + 1), \qquad \text{since } x = 2 \\
&= 2\ ((h+1)2^h - 2 \cdot 2^h + 1) \\
&= 2\ ((h-1)2^h + 1) \\
&= (h-1)2^{h+1} + 2
\end{aligned}$$

Substituting for h, we get

$$t(n) = (\log(n+1) - 2)(n+1) + 2.$$

Thus, $t(n)$ is both $O(n \log n)$ and $\Omega(n \log n)$. The intuition here is that most of the nodes in the tree are near the leaves and since the height of the tree is $\lfloor \log n \rfloor$, most of the leaves have depth which is either $\lfloor \log n \rfloor$ or very close to it.

[ASIDE: you may recognize the sum of depths from Assignment 3 Question 2.6. The total number of edges you need to follow to add n nodes to an empty binary search tree is the sum of depths of nodes in the final tree. When you add n nodes to the binary search tree, in the best case the tree is nearly balanced. Insertion of the ith node in this case requires traversing a branch of length $\lfloor \log i \rfloor$, as in the first summation on the previous page.]

Sum of the heights of all nodes

Recall that the height of a node in a tree is the maximum path length from the node to a leaf. For a complete binary tree of with $n = 2^{h+1} - 1$ nodes (that is, *all* the levels including h are full) the height of every node at level l will be $h - l$. I emphasize that h here refers to the height of the tree i.e. the height of the root node.

```
height     number of nodes of that height
------     ------------------------------
  h                1  (namely the root of the tree)
  h-1              2  (namely the children of the root)
   :               :
  i               2^{h-i}
  :
  1               2^{h-1}
  0               2^h      (or less, if last level is not full)
```

Define $t(n)$ now to be sum of heights of all nodes. We write it first in terms of h:

$$\begin{aligned} t(h) &= \sum_{l=0}^{h} (h-l)\, 2^l \\ &= h \sum_{l=0}^{h} 2^l - \sum_{l=0}^{h} l\, 2^l \\ &= h(2^{h+1} - 1) - (h-1)2^{h+1} - 2, \quad \text{from above} \\ &= 2^{h+1} - h - 2 \end{aligned}$$

In terms of n, we have

$$t(n) = n - \log(n+1)$$

which is $O(n)$ and $\Omega(n)$.

Generalizing the above results to k-ary trees

Consider a tree which could have up to k children at each node. We define a complete k-ary tree in the same way as we did for a binary tree, namely for all levels (except possibly the last one), each node has k children. Let's consider the case that all levels are full. In this case, the sum of depths of all nodes will be

$$t(h) = \sum_{l=0}^{h} lk^l$$

and the sum of heights of all nodes will be

$$t(h) = \sum_{l=0}^{h} (h-l)k^l.$$

We can apply the same derivation as above to express these in terms of n without a summation. (Basically you substitute $x = k$.) I will leave the derivations to you. But if you work it out, you will find that the sum of depths in $O(n \log_k n)$ and $\Omega(n \log_k n)$ and the sum of the heights is again $O(n)$ and $\Omega(n)$. [Note: you will recognize this situation. It is the situation of the trie in Assignment 3, but here we are considering the best case situation.]

Parent-child indexing scheme (revisited)

Last lecture, we considered how to represent a binary tree using an array. For node index i in the array, the children have indices $2i$ and $2i + 1$ and the parent has index $i/2$. I showed examples to convince you of the relationship, but I did not give a mathematical proof. Let me do that now. I will show that the left child of node i is $2i$. The other properties of the right child and parent follow immediately from that.

First note that the total number of nodes in levels 0 to $l - 1$ is

$$\sum_{i=0}^{l-1} 2^i = 2^l - 1.$$

The first node at level l is 2^l. There are 2^l nodes at level l. So the last node at level l is $2^{l+1} - 1$.

The index i of any node at level l can be written

$$i = 2^l - 1 + m$$

for some m such that $1 \leq m \leq 2^l$. But the index of the right child of node i is the number of nodes up to and including level l, which is $2^{l+1} - 1$, plus the $2m$ children (at level $l + 1$) of the first m nodes at level l. Thus, the index of the right child of node i is $2^{l+1} - 1 + 2m$. Hence the left child has index one less than that, namely $2^{l+1} - 2 + 2m$. But this quantity is just $2(2^l - 1 + m)$ which is $2i$. So we're done.

Notice that if there are n nodes in the heap, then the parent of the last node is $\frac{n}{2}$. In particular, nodes $\frac{n}{2} + 1, \cdots, n$ are leaves in the tree. We will use this fact in the following.

Building a heap in time $O(n)$

We now apply what we have learned this lecture to the problem of building a heap in time $O(n)$, which is faster than the buildHeap algorithm from last class.

Having a fast algorithm for building a heap would be useful if n were large and if we wanted to change how keys/nodes were compared from time to time. We would rebuild the heap, based on the new "compare to" definition. For example, in an investment banking application, the comparison might be based on a calculation of monetary costs of many interrelated things which might fluctuate over time.

Our new method for building a heap is based on the `downHeap()` method. The algorithm goes like this:

```
  buildHeap(a, n){
//    INPUT:  an array a of size n (with elements in any order)
//    OUTPUT: a heap

    for (k = n/2; k >= 0; k--)
       downHeap( a, n, k  )
    }
```

Here we are using a slightly more general `downHeap()` method than we used last lecture. Here we can `downHeap`, not just from the root node, but from any node. The only line that changed is the first one.

The method takes and array of size `n` and downHeaps from node `k`. It assumes that the trees rooted at the children of `k` – namely with roots at `2*k` and `2*k+1` – are heaps. It then changes the tree so that the subtree rooted at node `k` is also a heap.

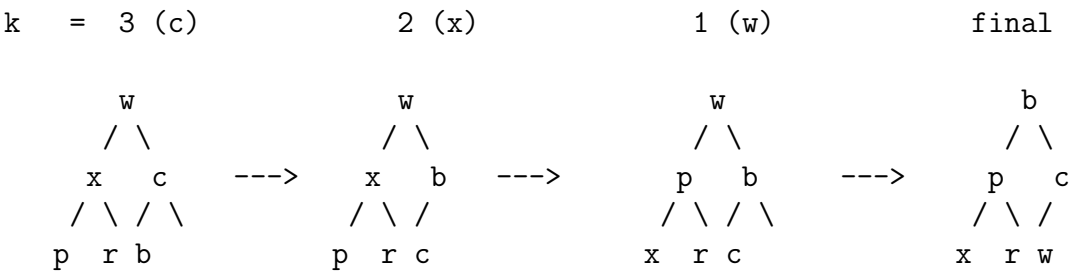
```
downHeap(a, n, k){
  i = k                              // THE ONLY LINE THAT CHANGED !
  while (2*i <= n){
    child = 2*i
    if child < n {
      if (a[child + 1] < a[child])
          child++
    }
    if (a[child] < a[ i ]){
       swapKeys(i , child)
       i = child
    }
  }
}
```

The new `buildHeap(a,n)` algorithm begins at node $k = \frac{n}{2}$. Notice that since each of the nodes $\frac{n}{2} + 1$ to n is a leaf, it must be a heap, namely a heap with one element.

For each `k`, the number of swaps done by `downHeap()` is at most `h - k`, that is, the height of the node in the tree. Thus the total number of swaps that we need to do is the total of the heights of the nodes in the tree. But we saw earlier in the lecture that this is $O(n)$!

Example

An initial arrangement of $n = 6$ keys is shown on the left. We show the state of the tree before the `k`th node is downHeaped, and the final state.



Maps

Today we will begin looking at collections of data that are called maps. You are familiar with the idea of *maps* in your daily life. You might have an address book which you use to look up addresses, telephone numbers, or emails. You index this information with a name. So the mapping is from name to address/phone/email. A related example is "Caller ID" on your phone. Someone calls from a phone number (the index) and the phone tells you the name of the person. Many other traditional examples are social security number or health cared number or student number for the index, which maps to a person's employment record, health file, or student record, respectively.

For a more formal definition of a map, suppose we have two sets: a set of keys K, and a set of values V. A *map* is a set of ordered pairs – also called *entries* –

$$M = \{(k, v) : k \in K, v \in V\} \subseteq K \times V,$$

such that, for any key k in our set of possible keys, there is *at most* one value v such that (k, v) is in the map. It is possible to have two keys map to the same value.

For example, let K be the set of integers and let V be the set of strings. Then the following is an example of a map:

$$\{(3, \texttt{cat}), (18, \texttt{dog}), (35446, \texttt{blablabla}), (5, \texttt{dog}), \}$$

whereas the following is not a map,

$$\{(3, \texttt{cat}), (18, \texttt{dog}), (35446, \texttt{blablabla}), (5, \texttt{dog}), (3, \texttt{fish})\}$$

because the key 3 has two values associated with it.

Data structures for maps

How can we represent a map using data structures that we have seen in the course? For example, we might have a key type `K` and a value type `V` and we would like our map data structure to hold a set of object pairs $\{ (k, v) \}$, where k is of type `K` and v is of type `V`. In the context of Java programming, we would have to decide whether these are primitive types or reference types, but that is not the point now.

In general, if the keys of map are comparable, then we can use one of the data structures we discussed earlier (sorted array, binary search tree) to organize the pairs $\{ (k, v) \}$ so that we can quickly access a pair by its key k. *(See slides for an illustration of the ideas in this paragraph.)* If we use a sorted array, then we can find a key in $O(\log n)$ steps, where n is the number of pairs in the map. Once we have found the entry with that key, we can find the key's associated value v in $O(1)$ steps, since a reference to the value is stored together with the key (as a pair). However, with a sorted array it is relatively slow to `add` or `remove` a (key,value) pair, namely $O(n)$. To get around this worst case behavior, we could instead use a binary search tree (BST) to store the (k, v) pairs, namely we index by comparing keys. The reason is that with a BST you can index with $O(\log n)$ steps. (You will learn in COMP 251 how to maintain a binary search tree so that it is balanced.)

Next lecture, we will discuss another scheme for representing maps, called *hashing*. Before I can explain hashing, though, I need to introduce a more few concepts.

Direct addressing

Suppose the keys `K` are positive integers. For example, the keys might be social insurance numbers (9 digit integers – in Canada anyhow) and the values be `Employee` objects (which hold varies pieces of information about a person working in a company). Social insurance numbers have 9 digits, so you could in principle define an array of size 10^9 of type `Employee`. That is, you would use someone's social insurance number to index directly into the array, and access the address of an `Employee` object associated with that social insurance number (if indeed there was an an `Employee` object with that social insurance number. Otherwise, the reference would be null and the `find` call would return null). See the sketch in the slides. This scheme for representing a map is called *direct addressing*. You provide the key, and in time $O(1)$ – namely an array access – you have the reference to the `Employee` object with that social insurance number.

Direct addressing works fine when the number of possible integers keys is relatively small, but not when the number of possible keys is large. For example, a typical company will not have anywhere near 10^9 employees, so an array of size 10^9 holds references to `Employee` objects would be mostly empty, i.e. contain `null`s. This is clearly a waste of memory. Next class, we will see how to deal with this problem.

Other examples of maps

(address, object)

When a Java programming is running, every object must sit somewhere in memory and thus has a (starting) memory address. For example, in many computer operating systems, addresses are either 32 bit numbers or 64 bit numbers. Another example is that in many implementations of the Java Virtual Machine, objects have a unique 24 bit number which is determined (in an unspecified way) by the object's address in the JVM, such that different objects have different numbers. This is the number returned by the `Object.hashCode()` method.

A set of objects together with their addresses, give us a set of $\{(address, object)\}$ pairs. This mapping from addresses to objects is exactly what the Java Virtual Machines does when it runs your program. Reference variables hold object addresses (keys) and the JVM takes an address and looks up the object (value) at that address in memory.

What's new here is that we are thinking for the first time of using the object's address as a key for comparisons. Indeed, if you think about it, that's not so different from what that a social insurance number (or phone number) does. Its just a number that is used to index other information. The number itself has no special meaning.

(string, big integer)

Another interesting map is from the set of strings to the set of integers, that is, for each string we define some integer. For example, consider strings made up of ASCII characters which are 8 bits (one byte) each. Let s be a string $s[0]s[1]\ldots s[n-1]$ with $s.length$ characters. A mapping from (ASCII) strings to integers could be defined using base 2^8, as follows:

$$mapping\ m\ :\ s \rightarrow \sum_{i=0}^{s.length-1} s[i]\ (2^8)^i\ .$$

If the characters that make up a string were in UNICODE (16 bits, i.e. 2 bytes each) then for any string we could define a positive integer:

$$s \rightarrow \sum_{i=0}^{s.length-1} s[i]\ (2^{16})^i$$

where $s[i]$ represents a number from 0 to $2^{16}-1$. So the integer defined by string s is being expressed as a number with base 2^{16}. That is, not base 2, not base 10, not base 16 (like hexadecimal), not base $2^8 = 256$ like in the string example above, but rather base 2^{16}.

Notice that this mapping would not produce preserve the lexicographic ordering of strings of a given length, however, since strings $s1$ and $s2$ should be compared first by their $s1[0]$ vs $s2[0]$ characters but that is not what would happen here. Instead, one could change the definition slightly to:

$$mapping\ m\ :\ s \rightarrow \sum_{i=0}^{s.length-1} s[i]\ (2^{16})^{length-i-1}\ .$$

This now would preserves the lexicographic ordering of strings of a given length. That is, if $s1 < s2$ then $m(s1) < m(s2)$ where $m(s)$ is the mapped (integer) value.

SEE THE SLIDES FOR AN EXAMPLE – "hello" vs. "prime".

The same argument holds for base 256 (ASCII), of course. If we define the mapping

$$mapping\ m\ :\ s \rightarrow \sum_{i=0}^{s.length-1} s[i]\ (2^{8})^{length-i-1}\ .$$

then the string of my first name, "Mike", would be mapped to the integer

$$77 * 256^3 + 105 * 256 * 256^2 + 107 * 256^1 + 101$$

since the ASCII codes of the characters 'M', 'i', 'k', 'e' are 77, 105, 107, and 101, respectively.
See `http://www.asciitable.com`.

Does this given a unique mapping from the set of `all` strings to the set of positive integers ? Not quite. The ASCII or UNICODE value of 0 stands for the symbol `null` – indeed, when you have a `null` reference in your program, it means that the value stored in that variable has all 0 bits. Thus "Mike" would have the same encoding as "Mike" with an arbitrary number of `null` characters appended to the end of it. Other than this ambiguity, no two strings map to the same value (and for every value, there is a unique string – up to the ambuiguity of trailing `null` characters).

Today we will look at a technique called *hashing* which is very commonly used in computer science. We will concentrate on *hash maps*. At the end of the lecture, I will mention a specific case of this, which in Java are known as *hash sets*. (In the lecture itself, I did not get to hash sets. I have included that material here and on the slide, in case you are interested.)

Suppose we have a map, that is, a set of ordered pairs $\{(k, v)\}$. We want a data structure such that, given a key k, we can access the associated value v. The main idea is as follows. If the keys are integers in a small range, say $0, 1, \ldots, m-1$, then we can just use an array of size m and the keys simply would index the array. The locations k in the array that are keys in the map would hold references to their corresponding values v. This was the "direct addressing" or "direct mapping" approach I mentioned last lecture. (See the slides for an illustration.) If the keys are more general, which is usually the case, then we will define another function – called a hash function – which maps the keys to the range $0, 1, \ldots, m-1$. Then, we put the two maps together. This gives us a mapping from the keys k to their corresponding values v. The result of this lecture will elaborate on this idea.

Hash function: hash code, compression

Given a space of keys K, define a *hash function* to be a mapping:

$$h : K \to \{0, 1, 2, m-1\}$$

where m is some positive integer. For each key $k \in K$, the hash function specifies some integer $h(k)$ between 0 and $m-1$. The $h(k)$ values in $0, \ldots, m-1$ are called *hash values*.

Typically m is smaller than the number of keys in K. For example, if the key space is all possible social insurance numbers (10^9 of them), but our map only has a few hundred keys, then we might use as hash function where m is 1000, rather than 10^9.

Note that the hash function h is a mapping that is defined on the entire set K and not just on a subset of K. As such, it typically happens that a huge number of keys in the key space K will map to the same hash value. This is not a problem since the keys in a given map will typically be a small subset of the space of possible keys. What's important is that, for the keys *in the map*, we would like it to be rare that two keys map to the same integer.

It is very common to design hash functions by writing them as a composition of two maps. The first map takes keys K to a large set of integers. The second map takes the large set of integers to a small set of integers $\{0, 1, \ldots, m-1\}$. The first mapping is called *hash coding* and the integer chosen for a key k is called the *hash code* for that key. The second mapping is called *compression*. Compression maps the hash codes to *hash values*, namely in $\{0, 1, \ldots, m-1\}$.

A typical compression function is the "mod" function. For example, suppose i is the hash code for some key k. Then, the hash value is $i \bmod m$. Often one takes m to be a prime number, though this is not necessary.

To summarize, we have that a hash function is typically composed of two functions:

$$\text{hash function } h \; : \quad \text{compression} \; \circ \; \text{hash code}$$

where $\circ$ is the composition of two functions i.e. $f \circ g(x) \equiv f(g(x))$, and

$$\text{hash code} \quad : \quad \text{keys} \to \text{integers}$$

$$\text{compression} \quad : \quad \text{integers} \rightarrow \{0, .., \text{m} - 1\}$$

and so

$$h : \{keys\} \rightarrow \{hash\ values\},$$

i.e. the set of hash values is $\{0, 1, \ldots, m-1\}$. Note that a hash function is itself a map!

Also note it can happen that two keys k_1 and k_2 map to the same hash value. Indeed this can happen in two ways. First, two keys might have the same hash code. Second, two keys might have different hash codes, but these two different hash codes map to the same hash value. In either case we say that a *collision* has occured. We will discuss collisions below.

Example: hash codes for strings

Suppose s is a string, composed out of unicode characters (16 bits each). A hash code for strings can be defined by:

$$h(s) = \sum_{i=0}^{s.length-1} s[i]\ x^{s.length-i-1}$$

where x is some positive integer. In the last lecture, we looked at the case $x = 2^{16}$ with $x = 2^8$.

For Java strings, the hash code is defined by setting $x = 31$. That is, `String.hashCode()` method computes the above sum. A few questions about this came up during in the lecture:

- **Q:** *Does using x=31 guarentee that two different strings will return different hash codes?*

  A: No, it doesn't. To see why not, consider strings of length 2. If we are using 16 bit chars (unicode, as in Java), then there are $2^{16} \times 2^{16} = 2^{32}$ possible strings of length 2. However, the number of hash codes defined by all strings of length 2 is much less than this. Any such hash code is of the form s[0]*31 + s[1], where s[0] and s[1] are 16 bit numbers. Since $31 < 2^5$, we can observe that any hash code $s[0] * 31 + s[1]$ must be a number less than $2^6 2^{16} = 2^{22}$ which is much less than 2^{32}. But we cannot uniquely encode 2^{32} things (possible strings of 2 unicode characters) with with $16 + 6 = 22$ bits. It has to be the case that many different strings map to each such hash code.

- **Q:** *What is the return type of the Java* `hashCode()` *method ?*

  A: If you look it up, you'll find that the return type is `int`. How can we reconcile this with the above polynomial, which obviously can compute numbers that are very large and cannot be represented with only 32 bits, i.e. the `int` type. The answer is that `String.hashCode()` uses the above sum $\underline{\text{mod } 2^{32}}$, i.e. it uses the last 32 bits of the number defined above as the return value of `hashCode()`. This is implicitly stated in the Java API where it says that this method computes the above polynomial "*using* `int` *arithmetic*".

An alternative hash code for strings is to use the above polynomial but let $x = 1$, in which case the above formula reduces to

$$h(s) = \sum_{i=0}^{s.length-1} s[i].$$

In this example, two strings consisting of the same set of characters, such as "eat" and "ate" or "hello" and "lleoh" would have the same hash code. A hash function based on this hash code would thus produce a collision.

Hash map: hash function, collisions

Let's return to our original problem, in which we have spaces of keys K and values V and we wish to represent a map M which is set of ordered pairs $\{(k, v)\}$, namely some subset of all possible ordered pairs `K` $\times$ `V`.

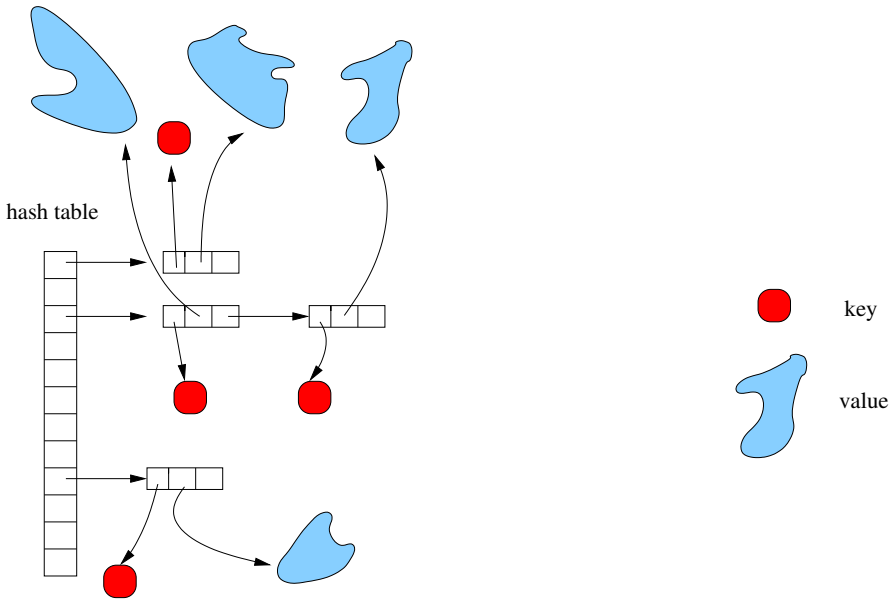
[ASIDE: Note that the "values" $v \in$`V` of a map M are not the same thing as the "hash values" $h(k)$ which are integers in $0, 1, \ldots, m-1$. Values $v \in$ `V` might be `Employee` records, or entries in an telephone book, for example, whereas hash values are indices in $\{0, 1, \ldots, m-1\}$.]

Define an array called a *hash table*, which will hold the (k, v) pairs. This is like the direct mapping array we discussed last lecture. Here, the number of slots in the array is m. This number is typically a bit bigger than the number of (k, v) pairs.

As we discussed above, we say that a collision occurs when two keys map to the same hash value. For example, if $m = 5$ then hash codes 7 and 22 produce a collision since $7 \bmod 5 = 2$, and $22 \bmod 5 = 2$.

To allow for collisions, we can use linked list of pairs (k, v) at each slot of our hash table array. These linked lists are sometimes called *buckets*. Storing a linked list of (k, v) pairs in each hash bucket is called *chaining*.[36]

Note that we need to store both the key k and the value v, because when use a key k to try to access a value v, we need to know which of the v's stored in a bucket corresponds to which k. We use the hash function to map the key k to a location/slot/bucket in the hash table. We then try to find the corresponding value v in the bucket/list. The bucket contains a linked list of (k, v) pairs. We examine each pair and try to match the search key k with the key in each pair. For example, with social insurance numbers (keys) and employee records (values), there may be multiple employee records stored in each bucket and we need to be able to check the keys of each one to see which, if any, corresponds to the given key (social insurance number).



[36] I mention these terms mainly in case you want to look them up and you need an index term.

In the worst case that all the elements in the collection hash to the same location in the array, then we have one linked list and access is $O(n)$ where n is the number of (key,value) pairs. This is undesirable since the whole point here is to represent a map so that we can access values as quickly as possible. To avoid having such long lists, we choose a hash function so that there are few, if any, collisions.[37] *If we are free to choose whatever hash function we want and we are free to choose the size m of the array, we can guarentee that the linked lists have a worst case constant.* In this sense, we say that *hash tables give $O(1)$ access.*

Is it possible to design hash functions that avoid collisions entirely? (Such hash functions are called *perfect hash functions.*) For this to be possible in any given situation, it is necessary that the number of buckets of the hash table be greater than or equal to the number of key-value pairs in the map. We define the *load factor* of a hash table to be the ratio of the number of (k, v) pairs currently in the table to the number of slots in the table (m). So a perfect hash function will need a load factor that is at most 1.

Java `HashMap`

In Java, the `HashMap<K,V>` class implements the sort of hash map that we have been describing. The `hashCode()` method for the key class `K` is composed with the "mod m" compression function where m is the capacity of an underlying array of linked lists. The linked lists hold `(K,V)` pairs. Have a look at the Java API to see some of the methods and their signatures: `put` , `get`, `remove`, `size`, `containsKey`, `containsValue`, and think of how these might be implemented.

In Java, the load factor for the hash table is never more than 0.75. If you try to add a new entry – using the `put()` method – to a hash table that would make the load factor go above 0.75, then a new hash table is generated, namely there is larger number m of slots and the (key,value) pairs are remapped to the new underlying hash table. This happens "under the hood", similar to what happens with `ArrayList` when the underlying array fills up and so the elements needs to be remapped to a larger underlying array (recall lecture 6).

`hashCode()` and `equals()`

For any class, the `hashcode()` and `equals()` method should be related as follows: if `k1.equals(k2)` is true, then `k1.hashCode() == k2.hashCode()` should be true. I did not give a good example of this in class, so let me do so here. Suppose our keys are user names in some data base and we want the user names to *not* be case sensitive i.e. we want to `Chang123` and `Chang123` and `CHANG123`, etc to be "equal". If these keys had different hash codes, then they might map to different buckets. So if we put the pair (`Chang123`, value) into the map and then we tried to get using the key `CHANG123`, we would fail to find the key and hence fail to find the value. Obviously this is not the behavior what we want.

What about the converse of the above rule? If `k1.hashCode() == k2.hashCode()` is true, then should we require that `k1.equals(k2)` returns true ? No, we shouldn't require this. For example, two different strings which we don't necessarily want to consider equal could have the same hash code, as we discussed earlier. In this case, we would get a collision. We would prefer not to have

[37] The word *hash* means to "chop/mix up". It should not be confused with the # symbol which is often the "hash" symbol i.e. hash tag on Twitter.

collisions, since it means we have to search through a linked list. But allow for collisions as the price to pay for generally quick access.

Bottom line: if you write a class which uses the `hashCode()` method and you want to override either the `Object` class'es `hashCode()` or `equals()` methods, then you should ensure this: if `k1.equals(k2)` returns true or false, then `k1.hashCode() == k2.hashCode()` should return true of false, respectively. This may require that you override *both* of them.

Java `HashSet<K>` class

The lecture ended here, and I did not have time to discuss a second interesting Java class that uses hashing. I am including a brief discussion below, and I strongly encourage you to read it. It is *not* on the final exam, but it is interesting and it will help you understand better how hashing relates to previous data structures we have seen in the course.

`HashSet<T>` is a class that stores a set of objects of some generic type `T`. Think of it as an alternative to `LinkedList<T>` or `ArrayList<T>`, for certain situations. `HashSet<T>` uses a hash table, and as its hash function it uses the `hashCode()` method for type `T`, together with the "mod m" compression function where again m is the capacity of the underlying array. Each bucket in the array holds a list of objects of types `T`. Think of a `HashSet<T>` as a `HashMap<T,V>` without the values `V`!

`HashSet` does not implement the `List` interface, but rather it implements the `Set` interface. (Check it out.) For example, suppose you simply want to keep track of a *set* of `Strings`. Suppose you don't need the strings to be ordered, i.e. you won't be asking for the i-th string in a lexicographic order (as you might for a sorted list), and you won't be asking for the string that was added last (as you would for a stack), and you won't be asking for the string that was added first (as you would for a queue). Suppose you simply want to keep track of a set of strings, and be able to ask questions like "is string x a member of this set?" or perform operations like adding a string to the set or deleting a string from the set. In this case, using a hash table would work great since it would allow you to answer these questions in time $O(1)$. The cost of doing so, of course, is that you are limited in the operations you can do.

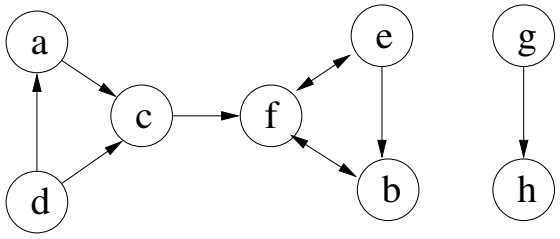
Graphs

You are familiar with data structures such as arrays and lists (linear), and trees (non-linear). We next consider a more general non-linear data structure, known as a graph. Like the previous data structures, a graph has a set nodes. Each node has a reference to other nodes. In the context of graphs, a reference from one node to another is called an edge.

In a linked list, there are references from one to the "next" (and/or "previous") node. In a (rooted) tree, there are references were to children nodes (or siblings or parent nodes). In a general graph, there is no unique notion of "next" or "prev" as in a list, and there is no unique notion of child or parent. Every node can potentially reference every other node. We will discuss data structures for graphs below.

Graphs have been studied and used for many years, and some of the main basic results go back a few hundred years. Mathematically, a graph consists of a set V called "vertices" and a set of edges $E \subseteq V \times V$, that is, the edges E in a graph is some set of ordered pairs of vertices.

Below is an example of a graph. Some of the arrows are in a single direction and some are in both directions. The latter are a convenient notation that there are two edges, namely one in each direction.



Examples of graphs include transportation networks. For example, V might be a set of airports and E might be direct flights between airports. Or V might be a set of html documents and E might be defined by URLs (links) between documents.

There are two very common data structures for graphs: adjacency lists and adjacency matrices.

Adjacency List

For each vertex v, consider a list of vertices w such that $(v, w) \in E$. For example, here is the adjacency lists for the example above.

```
a  -  c
b  -  f
c  -  f
d  -  a,c
e  -  b,f
f  -  b,e
g  -  h
h  -
```

One might represent adjacency lists as part of a graph class in Java. Take the simple case in which the vertices are characters. Then one could define a node:

```
class  GNode{
   char                vertex;
   LinkedList<GNode>   adjList;
}
```

See the example in the slides.

A more general scheme allows the vertices to be objects of some generic class `V`. One could then define

```
class  GNode<V>{
   V                   vertex;
   LinkedList<GNode>   adjList;
}
```

ASIDE: The example in the slides is sort of a compromise between these. For `V`, I use `Character`. But the sketches suggest that the characters are inside the node, i.e. a field in the `GNode` class.

The next question is how to organize these `GNode` objects into a graph. A simple way to do this is just to have a list of `GNode`s.

```
class  Graph<V>{
   LinkedList<GNode<V>>  gNodeList;
            :
            :      // methods
}
```

One would build the graph by constructing each `GNode` and adding it to the list of `GNode`'s.

A more sophisticated graph class might use a hash table:

```
class  Graph<K,V>{
   HashMap<K, GNode<V>>  graph;
}
```

The keys might be the vertices `V` themselves. Or the keys might a field within the vertices `V`. Another exmaple: in a graph of airports and flight connections, the key might be the three letter name of an airport e.g. YUL or LAX.

Adjacency Matrix

A different data structure for representing the edges in a graph is to use an *adjacency matrix* which is a $|V| \times |V|$ array of booleans, where $|V|$ is the number of elements in set V i.e. the number of vertices. The value 1 at entry $(v1, v2)$ in the array indicates that $(v1, v2)$ is an edge, that is, $(v1, v2) \in E$, and the value 0 indicates that $(v1, v2) \notin E$.

The adjacency matrix for the graph from earlier is shown below.

```
   abcdefgh
a  00100000
b  00000100
c  00000100
d  10100000
e  01000100
f  01001000
g  00000001
h  00000000
```

Note that in this example, the diagonals are all 0's, meaning that there are no edges of the form (v, v). But graphs can have such edges. An example is given in the slides.

Adjacency list versus adjacency matrix

Space considerations

If the number of entries in an adjacency matrix is n^2, and if n is large, then it is not practical to build such a matrix. For example, if the graph represents web pages on the world wide web, then $n \approx 20,000,000,000$. See `http://www.worldwidewebsize.com`. Most of the web pages point to a small number of other web pages, and so it would be much more space efficient to use an adjacency list.

Another consideration is that the adjacency matrix requires (strictly speaking) only one bit per edge, whereas an adjacency list requires much more, namely it requires a node in a linked list, plus other linked list overhead requirements.

Time considerations

Suppose you wish to know which edges a given vertex v is connected to, namely you want to know for which w's there is an edge $(v, w) \in E$. Using an adjacency matrix, it takes time $\mid V \mid$ time steps to determine this, namely you need to examine all elements in a row of the adjacency matrix and see which ones have the value 1. If you use an adjacency list, then the time it takes is proportional to the size of the adjacency list of v.

When would be advantagious to use an adjacency matrix? If you want to know if a graph contains a particular edge (v, w), then an adjacency matrix allows you to determine this is one step (an array lookup), whereas an adjacency list requires you to search through the list and this on average takes time proportional to the number of edges in the list.

Terminology

Finally, here is some basic graph terminology that you should become familiar with, and that is heavily used in discussing properties of graphs. The ideas are intuitive enough.

- *outgoing edges from* v - the set of edges of the form $(v, w) \in E$ where $w \in V$

- *incoming edges to* v - the set of edges of the form $(w, v) \in E$ where $w \in V$
- *in-degree of* v - the number incoming edges to v
- *out-degree of* v - the number outgoing edges from v
- *path* - a sequence of verticies $(v_1, \ldots, v_m)$ where $(v_i, v_{i+1}) \in E$ for each i.

  Note that the definition of path is similar to the definition of path that we saw for trees.
- *cycle* - a path such that the first vertex is the same as the last vertex

  If there were an edge (v, v), then this would be considered as a cycle. Such edges are certainly allowed in graphs and indeed are quite common.

At the end of the lecture, I mentioned a few definitions and important graph problems that you will see in subsequent courses.

Graph traversal

One problem we often need to solve when working with graphs is whether there is a sequence of edges (a path) from one vertex to another. More generally, what is the set of all vertices that can be reached from a given vertex v, that is, the set of all vertices w for which there is a path from v to w ?

Depth First Traversal

Recall the depth first traversal algorithm for trees. This algorithm generalizes to graphs as follows. The algorithm is similar to preorder traversal of a tree.

```
dft(v){
   v.visited = true
   for each w such that (v,w) is in E
     if !w.visited                          //  avoids cycles
        dft(w)
}
```

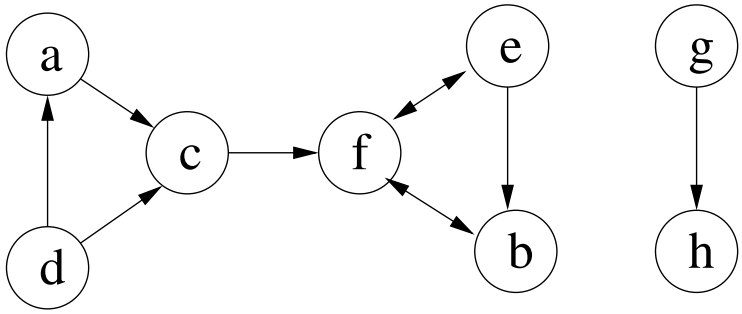
Note that we call this algorithm a traversal. But in fact it only reaches the nodes in the graph for which there is a path from the input vertex. This algorithm is sometimes called "depth first search" although that's not quite the right name either since we are not searching for a particular vertex – we're searching for all vertices that can be reached from the input vertex.

Also, before running this algorithm, you would need to set the `visited` field to false for all vertices in the graph. To do so, you would need to access *all* vertices in the graph. This is a different kind of traversal, which is independent of the edges in the graph. For example, if you are using a linked list to represent all the vertices in the graph, then you would traverse this linked list before you called the above traversal algorithm and set the `visited` field of each vertex to false. If you were using a hash table to represent the vertices in the graph, you would need to go through all buckets of the hash table by iterating through the hash table array entries and following the linked list stored at each entry. You would set the `visited` field to false on each vertex (value) in each bucket.

```
depthFirstTraversal(v){
   for each w in V
      w.visited = false
   dft(v)
}
```

Example *(see slides for a different example)—*

Let's run the above preorder depth first traversal algorithm on the graph from last lecture.
Below is the *call stack* (evolving over time) for the dft recursion. Also shown is a tree, known as the *call tree*. A node in the call tree represents one "call" of the `dft` method. Two nodes in the call tree have a parent-child relationship if the parent node calls the child node.



Note that the call stack is actually constructed when you run a program that implements this recursive algorithm, whereas the call tree is not constructed. The call tree is just a way of thinking about the recursion.

```
      CALL STACK
        b   e
      f f f f f
    c c c c c c c
  a a a a a a a a a

       ->  time
```

```
CALL TREE
     a
     |
     c
     |
     f
    / \
   b   e
```

Notice that nodes `d`, `g`, `h` are not visited.

Non-recursive depth first traversal

We can do a depth first traversal without recursion, just as we saw with tree traversal. We use a stack.

```
dft(v){
   initialize empty stack s
   v.visited = true
   s.push(v)
   while (!s.empty) {
     u =  s.pop()
     for each w in u.adjList{
        if (!w.visited){
           w.visited = true
           s.push(w)
        }
}
```

Note that this still is a preorder traversal. In the above example, we will still visit the nodes in the same order. But this is not true in general. *See the slides for an example.*

Post-order recursive depth first traversal

Is a postorder recursive traversal possible? Yes, it is, but we need to be careful. We cannot just move the `v.visited = true` assignment after the `dft` call because we could get stuck in an infinite loop, in the case of a cycle. The following will NOT work, for example, if the graph consists of a single cycle:

```
dft(v){
   for each w such that (v,w) is in E
     if !w.visited
        dft(w)
   v.visited = true
}
```

To make a post-order traversal work, we need to distinguish between "reaching" a vertex and "visiting" a vertex. Reaching a vertex could just mean that we access the adjacency list of the vertex, whereas visiting it might mean that we write into a field in the vertex. The following works – in the sense that it avoids the infinite loop from a cycle, and also allows us to visit a vertex after all the adjacent vertices have been visited.

```
dft(v){
   v.reached = true
   for each w such that (v,w) is in E
     if !w.reached
        dft(w)
   v.visited = true
}
```

Notice that for the above algorithms to work properly, the fields `visited` and `reached` should both be initialized to false.

Also, you could also come up with a non-recursive post-order traversal. Again, you would need to use a `reached` field.

Breadth first traversal

Like depth first traversal, "breadth first traversal" searches for all the vertices that can be reached from a given vertex v. The difference is that breadth first traversal attempts to create paths from a given vertex that are as short as possible. First, it visits vertices that are a distance 1 away. Then it visits vertices that are a distance 2 away, etc.

We have already seen an example of breadth first traversal, namely level order traversals in trees. Here we are considering a more general level order traversal, where the levels correspond to vertices that are a certain distance away (in terms of number of edges) from a given vertex.

Since we are working with a graph rather than a tree, we visit the nodes before enqueueing them. (Alternatively, we could use a `reached` field as we did with the post order traversal.)

```
bft(u){
  initialize empty queue q
  u.visited = true
  q.enqueue(u)
  while ( !q.empty) {
    v = dequeue()
    for each w in adjList(v)
      if !w.visited{
        w.visited = true
        enqueue(w)
      }
}
```

Notice that we enqueue a vertex only if we have not yet visited it, and so we cannot enqueue a vertex more than once. Moreover, all vertices that are enqueued are dequeued, since the algorithm doesn't terminate until the queue is empty.

Take the same example graphs as before. Below we show the queue `q` as it evolves over time, namely we show the queue at the end of each pass through the `while` loop.

Since this is not a recursive function, we don't have a "call tree". But we can still define a tree. Each time we visit a vertex – *i.e.* `w in adjList(v)` and we set `w.visited` to `true` – we get a edge `(v, w)` in the tree. We can think of the **w** vertex as a child of the `v` vertex, which is why we are calling this a tree. Indeed we have a rooted tree since we are starting the tree at some initial vertex `u` that we are searching from.

Note how the queue evolves over time, and what is the order in which the nodes are visited.

```
   QUEUE q    TREE
(snapshots)
   a           a
   c           |
   f           c
   be          |
   e           f
              /  \
             b    e

  order visited = acfbe
```

Another Example

Here is another example, which better illustrates the difference between `dft` and `bft`. We suppose the graph is undirected.

```
  GRAPH      ADJACENCY LIST       dft                 bft
                                (pre-order)

a - b - c    a - (b,d)          a - b - c           a - b - c
|   |   |    b - (a,c,e)               |           |   |   |
d - e - f    c - (b,f)          d - e - f           d   e   f
|   |   |    d - (a,e,g)       |                   |   |   |
g - h - i    e - (b,d,f,h)      g - h - i           g   h   i
             f - (c,e,i)
             g - (d,h)
             h - (e,g,i)
             i - (f,h)

            order visited:      abcfedghi            abdcegfhi
```

Note that with a postorder traversal, you would get the same tree, but you would visit the nodes in a different order. For the above example, you would *reach* the nodes in the same order as with the preorder traversal, but you would *visit* the nodes in the opposite order.