Addition

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 and you learned how to add single digit numbers. (4 + 7 = 11, 3 + 6 = 9, etc). Soon after, you learned a method for adding *multiple digit* numbers, which was based on the single digit additions that you had memorized. For example, you learned to compute things like:

```
  2343
+ 4519
  ----
   ?
```

The method that you learned was a sequence of computational steps: 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 $r[\,]$.

```
        a[N-1]  .. a[0]
      + b[N-1]  .. b[0]
        ---------------
r[N] r[N-1] ..  r[0]
```

The algorithm goes 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]

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

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

The operator / is integer division, and ignores the remainder, i.e. it rounds down (often called the "floor"). The operator % denotes the "mod" operator, which is the remainder of the integer division. (By the way, note the order of the remainder and carry assignments! Switching the order would be incorrect – think why.)

Also note that the above algorithm requires that you can compute (or look up in a table or memorized) the sum of two single digit numbers with '+' operator, and also add a carry of 1 to that result. You learned those single digit sums when you were very little, and you did so by counting on your fingers.

[1] By "pseudocode", I mean something like a computer program, but less formal. Pseudocode is not written in a real programming language, but good enough for communicating between humans i.e. me and you.

Subtraction

Soon after you learned how to perform addition, you 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 and you change the 2 to a 12, then compute 12-5=7 and write down the 7.

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

The borrowing trick is similar to the carry trick in the addition algorithm. The borrowing trick allows you to perform subtraction on N digit numbers, regardless on how big N is. In Assignment 1, you will be asked to code up an algorithm for doing this.

Multiplication

Later on in grade school, you learned how to multiply two numbers. For two positive integers a and b,

$$a \times b = a + a + a + \cdots + a$$

where there are b copies on the right side. In this case, we say that a is the *multiplicand* and b is the *multiplier*.

ASIDE: The above summation can also be thought of geometrically, namely consider a rectangular grid of tiles with a rows and b columns. You understood that the number of tiles is that same if you were to write it as b rows of a tiles. This gives you the intuition that $a \times b = b \times a$, a fact which is not at all obvious if you take only the summation above.

Anyhow, the summation definition above suggests an algorithm for computing $a \times b$: I claim this algorithm is very slow. To see why, think of how long it would take you a and b each had several digits. e.g. if $a = 1234$ and $b = 6789$, you would have to perform 6789 summations!

Algorithm 2 Slow multiplication (by repeated addition).

```
product = 0
for i = 1 to b do
    product ← product + a
end for
```

To perform multiplication more efficiently, one uses a more sophisticated algorithm, which you learned in grade school. An example of this sophisticated algorithm is shown here.

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

Notice that 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). This requires that you can compute single digit multiplication, e.g. 6 $\times$ 7 = 42. As a child, you learned a "lookup table" for this, usually called a "multiplication table". The second stage of the algorithm required adding up the rows of this 2D array.

Algorithm 3 Grade school multiplication (using powers of 10)

```
for j = 0 to N − 1 do
  carry ← 0
  for i = 0 to N − 1 do
    prod ← (a[i] ∗ b[j] + carry)
    tmp[j][i + j] ← prod%10          // assume tmp[][] was array initialized to 0.
    carry ← prod/10
  end for
  tmp[j][N + j] ← carry
end for
carry ← 0
for i = 0 to 2 ∗ N − 1 do
  sum ← carry
  for j = 0 to N − 1 do
    sum ← sum + tmp[j][i]        // could be more efficient here since many tmp[][] are 0.
  end for
  r[i] ← sum%10
  carry ← sum/10
end for
```

Of course, when you were a child, your teacher did not write out this algorithm for you. Rather, you saw examples, and you learned the pattern of what to do. Your teacher explained why this algorithm did what it was supposed to, and probably you got the main idea.

Division

The fourth basic arithmetic operation you learned in grade school was division. Given two positive integers a, b where $a > b$, the integer division a/b can be thought of as the number of times (call it q) that b can be subtracted from a until the remainder is a positive number less than b. This gives

$$a = q\ b + r$$

where $0 \leq r < b$, where q is called the *quotient* and r is called the *remainder*. Note that if $a < b$ then quotient is 0 and the remainder is a. Also recall that b is called the divisor and a is called the *dividend* so

$$dividend = quotient * divisor + remainder.$$

The definition of a/b as a repeated subtraction suggests the following algorithm:

Algorithm 4 Slow division (by repeated subtraction):

```
q = 0
r = a
while r ≥ b do
  r ← r − b
  q ← q + 1
end while
```

This repeated subtraction method is very slow if the quotient is large. There is a faster algorithm which uses powers of 10, similar in flavour to what you learned for multiplication. This faster algorithm is of course called "long division".

Example of long division

Suppose $a =$ `41672542996` and $b =$ `723`. Back in grade school, you learned how to efficiently compute q and r such that $a = qb + r$. The algorithm started off like this: (please excuse the dashes used to approximate horizontal lines)

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

In this example, you asked yourself, does 723 divide into 416? The answer is No. So, then you figure out how many times 723 divides into 4167. You guess 5, by reasoning that $7 * 5 < 41$ whereas $7 * 6 > 41$. You multiply 723 by 5 and then subtract this from 4167, and you get a result between 0 and 723 so now you know that 5 was a good guess.

To continue with the algorithm beyond what I wrote above, you "bring down the 2" (that is, the underlined 2 in the dividend 4167$\underline{2}$542996) and then you figure out how many times 723 goes into 5522, where the second 2 is the one you brought down.

Why does this algorithm work? Your teacher may have explained it to you, but my guess is that you didn't understand any better than I did. How would you write this algorithm down in pseudocode, similar to what I did with multiplication? In Assignment 1, you will not only have to write down pseudocode, you will be asked to code it up in Java.

Computational Complexity of Grade School Algorithms

For the multiplication and division operations, I presented two versions each and I argued that one was fast and one was slow. We would like to be more precise about what we mean by this. In general we would like to discuss how long an algorithm takes to run. You might think that we want an answer to be in *seconds*. However, for such a real measure we would need to specify details about the programming language and the computer that we are using. We would like to avoid these real details, because languages and computers change over the years, and we would like our theory not to change with it.

The notion of speed or *complexity* of an algorithm in computer science is not measured in real time units such as seconds. Rather, it is measured as a mathematical function that depends on the *size of the problem*. In the case of arithmetic operations on two integers a, b which have N digits each, we say that the size of the problem is N.

Let's briefly compare the addition and multiplication algorithms in terms of the number of operations required, in terms of the number of digits N. The addition algorithm has some instructions which are only executed once, and it has a `for` loop which is run N times. For each pass through the loop, there is a fixed number of simple operations which include $\%, /, +$ and assignments $\leftarrow$ of values to variables. Let's say that the instructions that are executed just once take some constant time c_1 and let's say that the instructions that are executed within each pass of the `for` loop take some other constant c_2. We could try to convert these constants c_1 and c_2 to units of seconds (or nanoseconds!) if had a particular implemenetation of the algorithm in some programming language and we were running it on some particular computer. But we won't do that because this conversion is beside the main point. The main point rather is that these constants have *some* value which is independent of the variable N. To summarize, we would say that the addition algorithm requires c_1+c_2N operations, i.e. a constant c_1 plus a term that is proportional (with factor c_2) to the number N of digits. A key concept which we will elaborate in a few weeks is that, for large values of N, the dominating term will be the last term. We will say that the algorithm runs in time "order" of N, or $O(N)$.

What about the multiplication algorithm? We saw that the multiplication algorithm involves two main steps, each having a pair of `for` loops, one nested inside the other. This "nesting" leads to N^2 passes through the inner loop. There are some instructions that executed in the outer `for` loops but not in the inner `for` loops, and these instructions are executed N times. There are also some instructions that are executed outside of all the `for` loops and these are executed a constant number of times. Let c_3 be the constant amount of time (say in nanoseconds) that it takes to execute all the instructions that are executed just once, and let c_4 be the constant amount of time that it takes to execute all the instructions that are executed N times, and let c_5 be the constant amount of time that it takes to execute all the instructions that are executed N^2 times, we have that the total amount of time taken by the multiplication algorithm is

$$c_3 + c_4N + c_5N^2.$$

Unlike with the addition algorithm, for large values of N, now the dominating term will be the N^2 term. We will say that the algorithm runs in time $O(N^2)$. We will see a formal definition of $O(\ )$ in a few weeks, once we have seen more examples of different algorithms and spent more time discussing their complexity.

Binary number representation

We humans represent numbers using decimal (the ten digits from 0,1, ... 9) or "base 10". The reason we do so is that we have ten fingers. There is nothing special otherwise about the number ten. Computers don't represent numbers using decimal. Instead, they are designed to 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,

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

whereas, in binary, we represent numbers using *bits* $\{0, 1\}$, 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.

You know how to count in decimal, so 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 in the leftmost column. In the left two columns below, I only used as as many digits or bits as I needed to represent the number. In the right column, I used a fixed number of bits, namely 8. 8 bits is called a *byte*.

decimal	binary	binary (8 bits)
0	0	00000000
1	1	00000001
2	10	00000010
3	11	00000011
4	100	00000100
5	101	00000101
6	110	00000110
7	111	00000111
8	1000	00001000
9	1001	00001001
10	1010	00001010
11	1011	00001011
etc		

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 '1' bits as a power of 2 using the decimal representation you are familiar with. Then you add up these decimal numbers, e.g.

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

The other direction is more challenging, however. How do you convert a decimal number to binary? I will give a simple algorithm for doing so soon which is based on the following idea. I first explain the idea in base 10 where we have a better intuition. Let m be a positive integer which is written in decimal. Then,

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

Note that $m/10$ chops off the rightmost digit and multiplying by 10 tags on a 0. So dividing and the multiplying by 10 might not get us back to the original number. What is missing is the remainder part, which we dropped in the divison.

In binary, the same idea holds. If we represent a number m in binary and we divide by 2, then we chop off the rightmost bit which becomes the remained and we shift right each bit by one position. To multiply by 2, we shift the bits to the left by one position and put a 0 in the rightmost position. So, for example, if

$$m = (11011)_2 \;=\; 1 * 2^4 + 1 * 2^3 + 1 * 2^1 + 1 * 2^0$$

then dividing by 2 gives

$$(11011)_2/2 \;=\; (1101)_2$$

then multiplying by 2 gives

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

More generally, for any m,

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

Here is the algorithm for converting m to binary. It is so simple you could have learned it in grade school. The algorithm repeatedly divides by 2 and the "remainder" bits $b[i]$ are the bits of the binary representation. After I present the algorithm, I will explain *why* it works.

Algorithm 5 Convert decimal to binary

INPUT: a number m
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
```

Note this algorithm doesn't say anything about how m is represented. But in practice, since you are human, and so m is represented in decimal.

Example: Convert 241 to binary

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

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

Now let's apply these ideas to the algorithm for converting to binary. Representing a positive integer m in binary *means* that we write it as a sum of powers of 2:

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

where b_i is a bit, which has a value either 0 or 1. So we write m in binary as a bit sequence $(b_{n-1} \; b_{n-2} \; \ldots \; b_2 \; b_1 \; b_0)_2$. In particular,

$$\begin{aligned} m \;\%\; 2 &= b_0 \\ m \;/\; 2 &= (b_{n-1} \ldots b_2 b_1)_2 \end{aligned}$$

Thus, we can see that the algorithm for converting to binary, which just repeats the mod and division operations, essentially just 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. The algorithm just reads off the rightmost bit of m each time that we divide it by 2.

i	m	b[i]
	$(11110001)_2$	
0	$(1111000)_2$	1
1	$(111100)_2$	0
2	$(11110)_2$	0
3	$(1111)_2$	0
4	$(111)_2$	1
5	$(11)_2$	1
6	$(1)_2$	1
7	0	1

Arithmetic in binary

Let's add two numbers which are written in binary. I've written the binary representation on the left and the decimal representation on the right.

```
11010    <-carries
 11010                  26
+ 1011                 +11
------                 ----
100101                  37
```

Make sure you see how this is done, namely how the "carries" work. For example, in column 0, we have $0 + 1$ and get 1 and there is no carry. In column 1, we have $1 + 1$ (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 which represents the 2^2 terms. Make sure you understand how the rest of the carries work.

How many bits N do we need to represent m ?

Let N be the number of bits needed to represent an integer m, that is,

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

where $b_{N-1} = 1$, that is, *we only use as many bits as we need*. This is similar to the idea that in decimal we don't usually write, for example, 0000364 but rather we write 364. We don't write 0000364 because the 0's on the left don't contribute anything.

Let's derive an expression for how many bits N we *need* to represent n. I will do so by deriving a lower bound and an upper bound for N. First, the lower bound: Since $b_{N-1} = 1$ and since each of the other b_i's is either 0 or 1 for $0 \leq i < N - 1$, we have

$$m \ \leq \ 2^{N-1} + 2^{N-2} + \cdots + 2 + 1. \qquad (*)$$

To go further, I will next use of the following claim which many of you have seen from Calculus: for any real number number x,

$$\sum_{i=0}^{N-1} x^i = \frac{x^N - 1}{x - 1}. \qquad (**)$$

The proof of this fact goes as follows. Take the sum on the left and multiply by $x - 1$ and expand:

$$\sum_{i=0}^{N-1} x^i(x-1) = \sum_{i=1}^{N} x^i - \sum_{i=0}^{N-1} x^i$$

Note the indices of the first sum on the right go from 1 to N and the second go from 0 to $N - 1$. Because we are taking a difference on the right, all terms cancel except for two, namely

$$\sum_{i=1}^{N} x^i - \sum_{i=0}^{N-1} x^i = x^N - 1.$$

Thus,

$$\sum_{i=0}^{N-1} x^i(x-1) == x^N - 1.$$

which is what I claimed above.

If we consider the case $x = 2$ we get

$$\sum_{i=0}^{N-1} 2^i = 2^N - 1.$$

For example, think of $N = 4$. We are saying that $1 + 2 + 4 + 8 = 16 - 1 = 15$.

Anyhow, getting back to our problem of deriving a lower bound on N, from equation (*) above, we have

$$\begin{aligned} m &\leq 2^{N-1} + 2^{N-2} + \cdots + 2 + 1 \\ &= 2^N - 1 \quad \text{which I just proved} \\ &< 2^N \quad \text{which makes the following step cleaner} \end{aligned}$$

Taking the $\log_2$ (base 2) of both sides gives:

$$\log_2 m < N.$$

Now let's derive an upper bound. Since we are only considering the number of bits that we need, we have $b_{N-1} = 1$, and since the b_i are either 0 or 1 for any $i < N - 1$, we can conclude

$$m \geq 1 * 2^{N-1} + 0 * 2^{N-1} + ...0 * 2^1 + 0 * 2^0 = 2^{N-1}$$

and so

$$\log_2 m >= N - 1.$$

Rearranging and putting the two inequalities together gives

$$\log_2 m < N \leq \log_2 m + 1$$

Noting that N is an integer, we conclude that N is the largest integer that is less than or equal to $\log_2 m + 1$, that is N is $\log_2 m + 1$ rounded down. We write: $N = floor(\log_2 m + 1)$ where "floor" just *means* "round the number down, i.e. it is a definition. Thus, the number of bits in the binary representation of m is always:

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

This is a rather complicated expression, and I don't expect you to remember it exactly. What I do expect you to remember is that N grows roughly as $\log_2 m$.

Other number representations

Today we have considered only positive integers. Of course, sometimes we want to represent negative integers, and sometimes we want to represent numbers that are not integers, e.g. fractions. These other number representations are taught in COMP 273. If you wish to learn about them now, please have a look at the lecture notes on my COMP 273 public web page.

Arrays

As you know from COMP 202 (or equivalent), an array is a data structure that holds a set of elements that are of the same type. Each element in the array can be accessed or indexed by a unique number which is its position in the array. An array has a capacity or `length`, which is the maximum number of elements can be stored in the array. In Java, we can have arrays of primitive types or reference type. For example,

```
int[]   myInts = new int[15];
```

creates an array of type `int` which has 15 slots. Each slot is initialized to value 0. We can also define an array of objects of some class, say `Shape`.

```
Shape[]  shapes = new Shape[428];
```

This array can reference up to 428 `Shape` objects. At the time we construct the array, there will be no `Shape` objects, and each slot of the array will hold a reference value of `null` which simply means there is nothing at that slot.

We are free to assign an element to any slot of an arrays. We can write

```
myInts[12] = -3;
shape[239] =  new Shape("triangle", "blue");
```

So if each of these arrays was empty before the above instructions (i.e. after construction of the array), then after the instruction each array would have one element in it.

Arrays have constant time access

A central property of arrays is that the time it takes to access an element does not depend on the number of slots (`length`) in the array. This constant access time property holds, whether we are writing to a slot in an array,

```
a[i] = ....;
```

or reading from a slot in an array

```
...  =   a[i];
```

You will understand this property better once you have taken COMP 206 and especially COMP 273, but the basic idea can be explained now. The array is located somewhere in the computer memory and that location can specified by a number called an *address*. When I say that the array has some address, I mean that the first slot of the array has some address. Think of this as like an apartment number in a building, or an number address of a house on a street. Each array slot then sits at some location relative to that starting address and this slot location can be easily computed.

[ASIDE: To find out where `a[k]` is, you just add the address of `a[0]` to `k` times some contant, which is the amount of memory used by each slot. This works since each slot in the array has constant size. The may be different for arrays of `int` versus arrays of `double` versus arrays of `Shape` but that's not a problem for the constant access time property.]

Lists

In the next few lectures, we look at data structures for representing and manipulating *lists*. We are all familiar with the concept of a list. You have a TODO list, a grocery list, etc. A list is different from a "set". The term "list" implies that there is a positional ordering. It is meaningful to talk about the first element, the second element, the i-th element of the list, for example. For a set, we don't necessarily have an order.

What operations are performed on lists? Examples are getting or setting the i-th element of the list, or adding or removing elements from the list.

```
get(i)          //  Returns the i-th element (but doesn't remove it)
set(i,e)        //  Replaces the i-th element with e
add(i,e)        //  Inserts element e into the i-th position
add(e)          //  Inserts element e (e.g. at the end of the list)
remove(i)       //  Removes the i-th element from list
remove(e)       //  Removes element e from the list (if it is there)
clear()         //  Empties the list.
isEmpty()       //  Returns true if empty, false if not empty.
size()          //  Returns number of elements in the list
:
```

In the next several lectures, we will look at some ways of implementing lists. The first way is to use an array. The data structure we will describe is called an *array list*.

Using an array to represent a list

Suppose we have an array `a[ ]`. For now I won't specify the type because it is not the main point here. We want to use this array to represent a list. Suppose the list has `size` elements. We will keep the elements at positions 0 to `size-1` in the array, so element i in the list is at position i in the array. This is hugely important so I'll say it again. With an array list, the elements are squeezed into the lowest indices possible so that there are no holes. This property requires extra work when we add an remove elements, but the benefit is that we always know where the ith element of the list is, namely it is at the ith slot in the array.

Let's sketch out algorithms for the operations above. Here we will not worry about syntax so much, and instead just write it the algorithm as pseudocode.

We first look at how to access an element in an array list by a read (get) or write (set).

`get(i)`

To get the i-th element in a list, we can do the following:

```
if (i >= 0) &  (i < size)
   return a[i]
```

Note that we need to test that the index `i` makes sense in terms of the list definition. If the condition fails and we didn't do the test, we would get in index out of bounds exception.

We set the value at position i in the list to a new value as follows:

set(i,e)

```
if (i >= 0) &  (i < size)
   a[i] = e
```

[ASIDE: This code replaces the existing value at that position in the list. If there were a previous value in that slot, it would be lost. An alternative is to return this previous element, for example,

```
tmp = a[i]
if (i >= 0) & (i < size)
    a[i] = e
return tmp
```

In fact, the Java ArrayList method `set` does return the element that was previously at that position in the list. See
`https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html#set(int,%20E)`]

Next we "add" an element `e` to i-th position in the list. However, rather than replacing the element at that position which we did with a `set` operation, the `add` method *inserts* the element. To make room for the element, it displaces (shifts) the elements that are currently at index $i, i+1, ..., \texttt{size}-1$. Here we can assume that $i \leq \texttt{size}$. If we want to add to the end of the list then we would add at position `size`. Moreover, we also assume for the moment that the size of the list is strictly less than the number of slots of the underlying array, which is typically called the length (`a.length`) of the array. It is also called the *capacity* of the array. The capacity is greater than the size, except in the special case that the array is full.

add(i,e)

```
if ((i > 0) & (i <= size) & (size < length)){
   for (j = size; j > i; j--)
       a[j] = a[j-1]              //  shift to bigger index
   a[i] = e                       //  insert into now empty slot
   size = size + 1
}
```

The above algorithm doesn't deal with the case that the the array is full i.e. `size == length`. We need to add code to handle that case, namely we we need to make a new and bigger array. This code would go at the beginning and would insure that the condition `size < length` when the above code runs.

```
if (size == length){
   b = new array with 2 * length slots
   for (int j=0; j < length; j++)
       b[j] = a[j]                //  copy elements to bigger array
   a = b
}
```

The method `add(i, e)` only adds – that is, inserts – to a slot which holds an element in the list. What if we want to add at the end of the list? Calling `add(size, e)` will generate an index out of bounds exception, so that won't work. Instead we use just `add(e)`. See the Exercises.

Overloading

In Java, it is possible to have two methods that have the same name but whose parameters have different types. This is possible even at the code compilation stage (when the java code is translated into the lower level byte code) because variables have a declared type, so when an instruction calls a method and supplies arguments to that method, the arguments have a type and the compiler knows which version of the method is being called. We will say more about this throughout the course.

For now, note that `add` is overloaded because there is an `add(int i, E e)` and an `add(E e)` method where `e` is of type `E`. Lists in Java also have overloaded `remove` methods, where `remove(int i)` removes the element at index `i` and an `remove(E e)` removes the first occurance of an element `e`.

Adding n elements to an empty array

Suppose we wish to work with an array list. We would like to add n elements. Let's start with an array of length 1. (We wouldn't do this, but it makes the math easier to start at 1.)

```
for i = 1 to n
     add( e_i )
```

where `e_i` refers to the ith element in the list. Here I am assuming that these elements already exist somewhere else and I am just adding them to this new array list data structure. Note that the `add( e_i)` operation adds to the end of the current list.

How much work is needed to do this? In particular, how many times will we fill up a smaller array and have to create a new array of double the array size? How many copies do we need to make from the full small array to the new large arrays? It requires just a bit of math to answer this question, and it is math we will see a lot in this course. So let's do it!

If you double the array capacity k times (starting with capacity 1), then you end up with array with 2^k slots. So let's say $n = 2^k$, i.e. $k = \log_2 n$. That means we double it $k = \log_2 n$ times.

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 lower half of the bigger array (and then eventually fill the upper half of the bigger array with new elements). When $k = 1$, we copy 1 element from an array of size 1 to an array of size 2 (and then add the new element in the array of size 2). When $k = 2$ and we create a new array of size 4, and then copy 2 elements (and then eventually add the 2 new elements). The number of copies from smaller to larger arrays is:

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

Note that the last term is 2^{k-1} since we are copying into the lower half of an array with capacity $n = 2^k$. See the figures in the slides.

As I showed last lecture,

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

and so the number of copies from smaller to larger arrays is $2^k - 1$ or $n - 1$. So using the doubling array length scheme, the amount of work you need to do in order to add n items to an array is about twice what you would need to do if the capacity of the array was at least n at the beginning. *The advantage to using the doubling scheme rather than initializing the array to be huge is that you don't need to know in advance what the number of elements in the list will be.*

Removing an element from an arraylist is very similar to adding an element, but the steps go backwards. We again shift all elements by one position, but now we shift down by 1 rather than up by 1. The `for` loops goes forward from slot i to $size - 2$, rather than backward from $size$ to $i + 1$

`remove(i)`

```
if ((i >= 0) & (i < size)){
   tmp = a[i]                        // save it for later
   for (k = i; k < size-1; k++){
      a[k] = a[k+1]                  //  copy back by one position
   }
   size = size - 1
   a[size] = null      //  optional,  but perhaps cleaner
   return tmp
}
```

One final, general, and important point: Adding or removing from the other end of the list where `i == size` is fast, since few shift operations are necessary. However, if the array has many elements in it and if you add a new element to position 0 or you remove the element at position 0, then you need to do a lot of shifts which is inefficient. We will return to this point in the next few lectures when we compare array lists to linked lists.

[See the slides for the figures to accompany these lecture notes.]

Last lecture I discussed how an array can be used to represent a list, and how various list operations can be implemented using arrays. Java has an `ArrayList` class that implements the various methods such as we discussed and uses an array as its underlying data structure. You should check out what these methods are for the `ArrayList` class in the Java API:

`https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`

Whenever you construct an `ArrayList` object, you need to specify the type of the elements that will be stored in it. You can think of this as a parameter that you pass to the constructor. In Java, the syntax for specifying the type uses <> brackets. For example, to declare an ArrayList of objects that are of type `Shape`, use:

```
ArrayList<Shape>  shapes = new ArrayList<Shape>( );
```

If you look at the Java API, you'll see that the class is `ArrayList<E>` where `E` is called a *generic type*. We will see many examples of generic types later.

Just a few points to emphasize before we move on. First, although the `ArrayList` class implements a list using an underlying array, the user of this class does not index the elements of the array using the array syntax `a[ ]`. The user (the client) doesn't even know what is the name of the underlying array, since it is `private`. Instead the user accesses an array list element using a `get` or `set` or other methods.

Second, because the `ArrayList` class uses an array as its underlying data structure, if one uses `add` or `remove` for an element at the *front* of your list, this operation will take time proportional to `size` (the number of elements in the list) since all the other elements needs to shift position in the array by 1. This can be slow. Thus, although arrays allow you to get and set values in very little (and constant) time, they can be slow for adding and removing from near the front of the list because of this shifting property.

Singly Linked lists

We next look at another list data structure - called a linked list - that partly avoids the problem we just discussed that array lists have when adding or removing from the front. (Linked lists are not a panacea. They have their own problems, as we'll see).

With array lists, each element was referenced by a slot in an array. With linked lists, each element is referenced by a node. A linked list node is an object that contains:

- a reference to an element of a list
- a reference to the `next` node in the linked list.

In Java, we can define a linked list node class as follows:

```
class  SNode<T>{
   T            element;
   SNode<T>     next;
}
```

where `T` is the generic type of the object in the list, e.g. `Shape` or `Student` or some predefined Java class like `Integer`. We use the `SNode` class to define a `SLinkedList` class.

Any non-empty list has a first element and a last element. If the list has just one element, then the first and last elements are the same. A linked list thus has a first node and a last node. A linked list has variables `head` and `tail` which reference the first and last node, respectively. If there is only one node in the list, then `head` and `tail` point to the same node.

Here is a basic skeleton of an `SLinkedList` class.

```
class  SLinkedList<T>{
   SNode<T>       head;
   SNode<T>       tail;
   int            size;

   private class  SNode<T>{
      T            element;
      SNode<T>     next;
   }
}
```

We make the `SNode` class a private inner class[2] since the client of the linked list class will not ever directly manipulate the nodes. Rather the client only accesses the elements that are referenced by the nodes.

A key advantage of linked lists over array lists which I promised above is that linked lists allow you to add an element or remove an element at the front of the list in a constant amount of time. Let's look the basic algorithms for doing so. Again I will use pseudocode, rather than Java. We begin with an algorithm for adding an element to the front of a linked list.

```
addFirst( e ){                    //  add element e to front of list
   newNode = a new node
   newNode.element = e
   if (head == null){              // list is empty
      head = newNode
      tail = head
   }
   else{
       newNode.next = head
       head = newNode
   }
   size++
}
```

The order of the two instructions in the `else` block matters. If we had used the opposite order, then the `head = newNode` instruction would indeed point to the new first node. However, we would

[2]For info on inner classes (and nested classes in general), see
`https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html`

not remember where the old first node was. The `newNode.next = head` instruction would cause `newNode.next` to reference itself.

Also notice that we have considered the case that initial the list was empty. This special case ("edge case") will arise sometimes. Whenever you write methods, ask yourself what are the edge cases and make sure you test for them. I may omit the edges cases, sometimes intentionally, sometimes unintentionally. Don't hesitate to ask if notice one is missing.

Let's now look at an algorithm for removing the element at the front of the list. The idea is to advance the `head` variable. But there are a few other things to do too:

```
removeFirst(){
//  test for empty list omitted  (throw an exception)
   tmp = head             // remember first element, so we can return it
   head = head.next       // advance
   tmp.next = null        // not necessary but conceptually cleaner
   size = size - 1
   if (size == 0)
       tail = null        // edge case:  one element in list
   return tmp.element
}
```

Notice how we have used `tmp` here. If we had just started with (`head = head.next`), then the old first node in the list would still be pointing to the new first node in the list, even though the old first node isn't part of the list. (This might not be a problem. But it isn't clean, and sometimes these sorts of things can lead to other problems where you didn't expect them.) Also, in the code here, the method returns the element. Note how this is achieved by the tmp variable. In the slides, the method did not return the removed element.

Let's now discuss methods for 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( e ){
   newNode = a new node
   newNode.element = e
   tail.next = newNode
   tail = tail.next
   size = size + 1
}
```

Removing the last node from a list is more complicated, however. The reason is that you need to modify the `next` reference of 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. (I do not return the element below. That code could be added.) Otherwise, it scans the list for the element that comes before the 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
   }
   size = size-1    // optional variable
}
```

This method requires about `size` steps. This is much more expensive than what we had with an array implementation, where we had a constant cost in removing the last element from a list.

Time Complexity

The table below compares the time complexity for adding/removing an element from the head/tail of an array or linked list that has size N.

	array list	singly linked list
addFirst(e)	O(N)	O(1)
removeFirst()	O(N)	O(1)
addLast(e)	O(1)	O(1)
removeLast()	O(1)	O(N)

The main problem with the singly linked list is that it is slow to remove the last element. Note that singly linked lists do just as well as array lists, except for removing an element from the end of the list. We will see one way to get around this next lecture when we discuss doubly linked lists. However, we will also see that doubly linked lists don't solve all of our problems.

How many objects are there in a singly linked list vs. array list?

As I discuss in the slides, suppose we have a linked list with `size` = 4. How many objects do we have in total? We have the four `SNode` objects. We have the four elements (objects of type `Shape, say`) that are referenced by these nodes. We also have the `SLinkedList` object which has the head and tail references. So this is 9 objects in total.

For an array list with four elements, we would have 6 objects: the four elements in the list, the arraylist object, and the underlying array object. (Yes, an array is considered to be an object in Java.)

[ASIDE: You should follow along with the figures in the slides.]

Doubly linked lists

The "S" in the `SLinkedList` class from last lecture stood for "singly", namely there was only one link from a node to another node. Today we look at "doubly linked" lists. Each node of a doubly linked list has two links rather than one, namely references to the previous node in the list and to the next node in the list. These reference variables are typically called `prev` and `next`. As with the singly linked list class, the node class is usually declared to be a private inner class. Here we define it within a `DLinkedList` class.

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

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

The key advantage of doubly linked lists over a singly linked list is that the former allows us to quickly access elements near the back of the list. For example, to remove the last element of a doubly linked list, one simply does the following:

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

Dummy nodes

When writing methods (or algorithms in general), one has to consider the "edge cases". For doubly linked lists, the edge cases are the first and last elements. These cases require special attention since `head.prev` and `tail.next` will be `null` which can cause errors in your methods if you are not careful.

To avoid such errors, it is common to define linked lists by using a "dummy" head node and a "dummy" tail node, instead of head and tail reference variables.[3] The dummy nodes are objects of type `DNode` just like the other nodes in the list. However, these nodes have a `null` element. Dummy nodes do not contribute to the `size` count, since the purpose of `size` is to indicate the number of elements in the list. See figures in slides.

[3] Dummy nodes can be defined for singly linked lists too.

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

   // constructor

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

   //  ... List methods and more
}
```

Let's now look at some `DLinkedList` methods. We'll start with a basic getter which gets the i-th element in the list:

```
   get( i ){
     node = getNode(i)
     return  node.element
   }
```

This method uses a helper method that I'll discuss below. Its worth having a helper method because we can re-use it for several other methods. For example, to remove the i-th node:

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

This code modifies the `next` reference of the node that comes before the i-th node, that is `node.prev`, and it modifies the `prev` reference of the node that comes after the i-th node, that is `node.next`. Because we are using dummy nodes, this mechanism works even if i = 0 or i = size-1. Without dummy nodes, `node.prev` is `null` when i = 0, and `node.next` is null when `i = size - 1`, so the above code would have an error if we didn't use dummy nodes.
[ASIDE: note that I am not bothering to set the `next` and `prev` references in the removed node to `null`.]

Here is an implementation of the `getNode(i)` method. This method would be *private* to the `DLinkedList` class.

```
getNode(i){
   node = dummyHead.next
   for (k = 0;  k < i; k++)
      node = node.next
   return node
 }
```

One can be more efficient than that, however. When index i is greater than size/2, then it would be faster to start at the tail and work backwards to the front, so one would need to traverse size/2 nodes in the worst case, rather than size nodes as above.

```
getNode(i){
   if (i < size/2){
       node = dummyHead.next
       for (k = 0;  k < i; k++)
           node = node.next
   }
   else {
       node = dummyTail.prev
       for (k = size-1;  k > i; k--)
          node = node.prev
       }
   return node
}
```

The `remove(i)` method still takes `size/2` operations in the worst case. Although this worst case is a factor of 2 smaller for doubly linked list than for singly linked lists, it still grows linearly with the size of the list. Thus we say that the `remove(i)` method for doubly linked lists still is $O(N)$ when N is the size of the size. This is the same time complexity for this method as we saw for array lists and for singly linked lists. Saving a factor of 2 by using the trick of starting from the tail of the list half the time is useful and does indeed speed things up, but only by a proportionality factor. It doesn't change the fact that in the worst case the time is takes grows linearly with the size of the list.

Java `LinkedList`

Java has a `LinkedList` class which is implemented as a doubly linked list. Like the `ArrayList` class, it uses a generic type `T`.

```
LinkedList<T>  list  =  new  LinkedList<T>();
```

The `LinkedList` class has more methods than the `ArrayList` class. In particular, the `LinkedList` class has `addFirst()` and `removeFirst()` methods. Recall removing elements from near the front of a linked list was expensive for an array list. So if you are doing this a lot in your code, then

you probably don't want to be using an array list for your list. So it makes sense that that the `ArrayList` class wouldn't have such methods. But adding and removing the first elements from a list is cheap for linked lists, so that's why it makes sense to have these methods in the Java `LinkedList` class. (Of course, you could just use `remove(0)` or `add(0,e)`, but a specialized implementation `addFirst()` and `removeFirst()` might be a bit faster and the code would be easier to read – both of which are worth it if the commands are used often. In addition, the `LinkedList` class has an `addLast()` and `removeLast()` method, whereas the `ArrayList` class does not have these methods.

Suppose we add N students to the front (or back) of a linked list. Adding N students to an empty linked list takes time proportional to N since adding *each* element to the front (or back) of a linked list takes a constant amount of time, i.e. independent of the size of the list

How not to iterate through a linked list

What if we wanted to print out the elements in a list. At first glance, the following pseudocode would seem to work fine.

```
for (k = 0; k < list.size();   k ++)            //  size == N
     System.out.println( list.get(  k  ));
```

For simplicity, suppose that `get` were implemented by starting at the head and then stepping through the list, following the next reference until we get to the i-the element. (See Exercises 3 for case that the `get` method uses what we did on the previous page, namely worst case is $N/2$.) Then, with a linked list as the underlying implementation, the above for loop would require

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

steps. This is $O(N^2)$ which gets large when N is large. It is obviously inefficient since we realy only need to walk through the list and visit each element once. The problem here is that each time through the loop the `get(k)` call starts again at the beginning of the linked list and walks to the k-th element.

What alternatives to we have to using repeated `get`'s ? In Java, we can use something called an *enhanced for loop*. The syntax is:

```
for  (E    e :   list){
       //  do something with element e
}
```

where `list` is our variable of type `LinkedList<E>` and `e` is a variable that will reference the elements of type `E` that are in the list, from position, 0, 1, 2, ..., size-1.

Note that in Assignment 1, you will use the `LinkedList` class. At this stage of your Java abilities and knowledge, it will be easier for you to use the regular `for` loop rather than the enhanced `for loop`. So, just plug your nose when you write out the regular `for` loops such as the in the print statements above.

Space Complexity ?

We've considered how long it takes for certain list operations, using different data structures (array list, singly linked and doubly linked lists). What about the space required? Array lists use the least amount of total space since they require just one reference for each element of the list. (These 'slots' must be adjacent in memory, so that the address of the k-th slot can be computed quickly.)

Singly linked lists take twice as much space as array lists, since for each element in a singly linked lists, we require a reference to the element and a reference to the next node in the list. Doubly linked lists require three times as much space as array lists, because they require a reference to each element and also references to the previous and next node.

All three data structures require space proportional to the number of elements N in the list, however. So we would say that all require $O(N)$ space.

Miscellaneous Java stuff

Overloading, and method signatures

When you look at the Java API for the `ArrayList` or `LinkedList` class or many other Java classes you will see that there are some methods that have multiple versions. For example, in these list classes there are several versions of the `remove` method, including `remove(E element)` and `remove(int index)`. Both of these return a reference to the removed element. The first `remove` method removes the first instance of the given element in the list. The second `remove` method removes the i-th element, whatever it happens to be.

A method that has multiple versions is said *overloaded*. When a method is overloaded, there must be some way for the Java compiler to know which version of the method the programmer wants to use. Overloaded methods are distinguished by the type(s) and number of the arguments, what is called the method *signature*. The Java compiler can usually know which method the programmer wishes to use just be examining the type of the parameter passed to the method. (There are some subtleties here, which we will return to at the end of the course, when we discuss "polymorphism".) For now, you should just note that there can exist multiple versions and they are distinguished by the types of their arguments.

[See the slides for figures to complement the discussion in these notes.]

One common problem that you need to solve in computing is to sort a list of N objects. Let's say we want the elements to be in *increasing* order. Here we are assuming that objects are comparable, in the sense that for any two objects A and B that we are considering, either $A < B$ or $A > B$. (It could also be that $A == B$, if there are multiple copies of the same element in the list.) For example, the elements in the list might be numbers, or they might be strings which can be ordered alphabetically. Today we will look a few simple algorithms. Later in the course, we'll look at more complicated algorithms which are much faster when the list is large.

We will discuss three algorithms today. These will be presented without Java code and without committing to a particular data structure e.g. array or array list or doubly or singly linked list. The algorithm could be implemented in principle with any of these, but the details would be distracting. Instead I would like you to think about the similarities and differences between the algorithms, which can be a bit subtle.

Bubble Sort

The first algorithm is perhaps the simplest to describe. You traverse through the list repeatedly, and whenever you find two neighboring elements that are out of order, you swap them. Elements gradually make their way to their correct position in the list. The algorithm is called bubblesort because one thinks of bubbles rising in a fluid [4]. Here it is:

```
ct = 0
repeat {
  continue = false
  for i = 0 to N - 2 - ct  {         //  N-1 is the last index
    if  list[ i ] > list[ i + 1 ]{
      swap( list[ i ], list[ i + 1 ] )
      continue = true
    }
  }
  ct++
} until continue == false
```

A few points to note: first, as you should have seen in COMP 202, swap is done using a temporary variable as follows. If you don't know why a temporary variable is needed, then see me.

```
swap(x,y){
   tmp = x
   x   = y
   y   = tmp
}
```

Second, what can we saw after one pass through the list? We can say that the largest element in the list will be at the end of the list. The reason is that the `for` loop will eventually hit this element and will then drag it to the end of the list via successive swaps.

[4] although this analogy doesn't really makes sense once you understand the algorithm, ...but it doesn't matter...

What can we say about the position of the smallest element in the list after the first pass? Not much, except that it won't be at the end of the list (unless all elements are equal). For example, if the smallest element of the list starts out at the end of the list, i.e. in position $N-1$, then in the first pass through the inner loop, the element will be moved only to position $N-2$.

How many passes through the list will we need to put all elements in order? We will need at most $N=1$ passes. Take the case that the smallest element starts off at the end of the list at position $N-1$. In the first pass it moves to position $N-2$. In the second pass, it moves to position $N-3$. etc. Thus, it will take $N-1$ passes for the smallest element to arrive at position 0.

The `ct` variable counts the number of times through the outer loop. Each time through the outer loop, the largest element in positions from index 0 to `N-1-ct` is moved to position `N-1-ct`. See the example in the slides.

How long does the bubblesort algorithm take in the worse case that the outer loop is executed $N-1$ times ? There are two nested loops, so the operations within the inner loop are executed $(N-2)+(N-3)+\cdots+2+1$ times. This sum is $\frac{(N-2)(N-1)}{2}$ which is roughly $N^2/2$. Convince yourself that this *only* happens when the smallest element starts out at the end of the list.

Selection sort

The second algorithm we examine is called *Selection Sort*. The algorithm repeatedly finds the smallest element and moves it to its correct position in the list. For each i, the algorithm finds the minimum element in positions i to $N-1$. If this minimum element is at a position (`index`) different from i, then it swaps this minimum element with the element at position i.

```
  for  i = 0  to N-2   {
    index = i
    minValue = list[ i ]
    for k = i+1  to  N-1  {
       if ( list[k] < minValue ){
          index = k
          minValue = list[k]
       }
    }
    if ( index != i )
        swap(  list[i], list[ index ] )
}
```

Like bubblesort, this algorithm takes repeated passes through the array, but it covers fewer elements each time.

```
for (i = 0; i < N-1; i++)
       :
     for ( k = i+1; k < N; k++)
```

When `i` is 0, it takes `N-1` steps through inner loop (from k = 1 to N-1). When `i` is 1, it takes `N-2` steps through the inner loop, namely from k = 2 to N-1. The total number of times passing through the inner loop is

$$N-1+N-2+N-3+...+3+2+1.$$

which is $\frac{N(N-1)}{2}$ which is roughly $N^2/2$.

Note a few differences with bubblesort. One difference is that in the best case bubble sort only takes one pass through the outer loop, whereas selection sort always takes $N-1$ pass through its outer loop. Thus bubble sort is faster in the best case. However, one advantage of selection sort over bubble sort is that selection sort does fewer swaps in the typical case. Indeed, selection sort does at most one swap in each pass in the outer loop, whereas bubblesort typically has to do a lot of swaps.

Insertion Sort

The third algorithm is similar to both the previous ones in that it uses nested loops. In particular, it is similar to selection sort in that it maintains a list of sorted elements at the front of the list, and then increases the size of the sorted list by one each time. However, whereas selection sort considers all the remaining unsorted elements and finding the smallest one, insertion sort considers only the next element in the list and finds where it belongs relative to the ones that have already been sorted. It then inserts this next element into the proper position. This is why it is called "insertion sort".

The algorithm goes through an outer loop $N-1$ times. In the kth pass through the loop (starting at $k=1$), the algorithm inserts element at index k into its correct position with respect to the elements up to and including position $k-1$, which are already in their correct order.

How does the algorithm put the element at index k (`list[k]`) into its correct position with respect to elements at indices 0 to $k-1$? The idea is to search backwards from index k until it finds the right place for this element. As it searches back, it shifts forward by 1 position any element that is bigger than element k. Once an element is found that less than or equal to `list[k]`, it inserts the value `list[k]` directly after that element. Here is the algorithm.

```
for k = 1 to N - 1 {      // don't need to consider k = 0
   elementK =  list[k]   //  copy current element so it doesnt get erased
   i = k
   while (i > 0) and ( elementK  < list[i - 1] ){
       list[i] =  list[i - 1]   // shift up larger elements
       i  = i -1
   }
   list[i] = elementK  // insert into correct position
}                       // (has no effect if elementK was in correct position)
```

What is the time complexity of insertion sort? As in the first two algorithms, we have nested for loops. The outer goes from k = 1 to N-1, and the inner goes from i = k down to 1 in the worst case. So the number of passes through the inner loop is:

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

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

Note that the inner loop only continues as long as the while condition is met. If `elementK` is already greater than `list[k-1]` then the inner loop ends immediately because element **k** is already in the correct position.

What is the best and worst case scenario? In the best case, the array is already sorted from smallest to largest. Then the condition tested in the **while** loop will be false every time, and so the inner loop will take constant time. Since there are N passes through the **for** loop, the time taken is proportional to N in the best case.

Summary

In trying to understand these algorithms, it is best to start at a high level descrption and work ones way eventually to the code. Don't start with the code. It may also help to look at various website that give visualizations of different sorting algorithms. For example:

`https://www.toptal.com/developers/sorting-algorithms`.

Abstract Data Types (ADT)

We began our discussion of lists a few lecture ago by defining it abstractly as a set things of a certain type and a set of 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. The idea of an ADT will come up many times in this 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`).

Note that a stack is a kind of list, in the sense that it is a finite set of ordered elements. However, it is restricted type of list since it has fewer operations you can apply on it. Unlike a list, a stack generally does not allow you to directly access the i-th element.

Data structure for a stack

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.

If you use an array list, then you should push and pop at the end of the list with `addLast()` or `removeLast`. The reason is that if you add or remove from the front of an array list, you need to shift all the other elements each time which is inefficient. If you use a singly linked list to implement a stack, then you should push and pop at the front of the list, not at the back. The reason (recall) is that removing i.e. popping from the back of a singly linked list would be 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 or at the back. Both would be good.

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. Here is how the stack evolves over time:

```
                  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.

Suppose there are multiple types of left and right parentheses, for example, (,), {, }, [,]. 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. 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.

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
```

The basic algorithm for matching parentheses is shown below. We assume the input has been already partitioned ("parsed") into disjoint *tokens*. For this example, 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)

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

 while (there are more tokens) {
   token  =  get next token
   if token is a left parenthesis
      push(token)
   else {                                          //  token is a right parenthesis
      if stack is empty
         return false
      else {
         pop left parenthesis from stack
         if  popped left parenthesis doesnt match the right parenthesis
            return false
      }
   }
 }
```

Example 3: HTML tags

The above problem of balancing different types of parentheses might seem a bit contrived. But in fact, this arises in many real situations. An example is HTML *tags*. If you have never looked at HTML markup 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.

Tags are of the form `<tag>` and `</tag>`. They correspond to left and right parentheses, respectively. For example, `<b>` and `</b>` are "begin boldface" and "end boldface". HTML tags are *supposed to be* properly nested. 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. It does make some sense, if you think of the tags as turning on or off some state (bold, italic). But the HTML language is not supposed to allow this. And writing HTML markup this way can get you into trouble since errors such as a forgotten or extra parenthesis can be very hard to find.

Many HTML authors write improply nested HTML markup. Because of this, web browsers typically will allow improper nesting. The reason is that web browser programmers (e.g. google employees

who work on Chrome) want people to use their browser and if the browser displayed junk when trying to interpret improper HTML markup, then users of the browser would give up and find another browser.

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

Example 4: 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 $(x_0 + \cos\theta, y_0 + \sin\theta)$. It moves the state position to the end of the line just drawn. Recall $\cos(0) = 1, \cos(90) = 0, \cos(180) = -1, \sin(0) = 0, \sin(90) = 1, \sin(180) = 0, ...$
- `L` - turns left (counter-clockwise) by 90 degrees
- `R` - turns right (clockwise) by 90 degrees
- `[` - pushes the current state onto the stack
- `]` - pops the stack, and current state ← popped state

See the slides for a few examples.

Note that this simple language doesn't allow one to move the pen without drawing, except by returning to a position that the pen had been in previously. This means that the language can only be used to draw connected figures. To draw disconnected figures, you would need another instruction e.g. `M` could move the pen forward by a distance one without drawing.

Example 5: the "call stack"

We have been discussing stacks of things. One can also have a stacks of tasks. 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. While chatting, the phone rings and you answer it. You finish the phone conversation and go back to the person in your office. Then maybe there is another interruption which you take care of, return to work, etc. In each case, when you are done with a task, you ask yourself "what was I doing just before I began this task?".

A similar stack of tasks occurs when a computer program runs. The program starts with a `main` method. 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 keep track of methods and to return to the 'caller' is to use 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.

```
Class    Demo {
   void   mA( ) {
      mB( );
      mC( );
   }
   void  mB( ) {  }
   void  mC( ) {  }
   void  main( ){
       mA(  );
   }
}
```

The stack evolves as follows:

```
                        B             C
                  A     A      A      A      A
         main    main   main   main   main   main  main
----     ----    ----   ----   ----   ----   ----  ----  ----
```

Also see the slides for an example using the `SLinkedList` code from the linked list exercises. I briefly showed how the `TestSLinkedList` calls the `addLast()` method of the LinkedList class, and I show the Eclipse call stack. When you use Eclipse in debugger mode, and you set breakpoints in the middle of methods, there is a panel that shows you the call stack.

Queue

Last lecture we looked at a abstract data type called a "stack". I introduced the idea of a stack by saying that it was a kind of list, but with a restricted set of operations, `push` and `pop`. Today we will consider another kinds of abstract data type called a "queue". A queue can also be thought of a list. However, a queue is again a restricted type of list since it has a limited set of operations.

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 that gets served next. There are many examples of queues in a computer system. When you type on your keyboard, the key values enter a queue (or 'buffer'). Normally don't notice the queue because each keystroke value gets read and removed from the queue before the next one is entered. But sometimes the computer is busy doing something else, and you do notice the queue. There is a pause where nothing you type gets echoed to your screen (e.g. to your text editor), and then suddenly some sequence of characters you typed gets processed very quickly. Other examples of queues are printer jobs, CPU processes, client requests to a web server, etc.

The fundamental property of a queue is that, among those things currently in the queue, the one that is removed next is the one that first entered the queue, i.e. the one that was least recently added. This is different from a stack, where the one that is removed next is the newest one, that is, 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.

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.

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 AFTER OPERATION
	(initially empty)
enqueue(a)	a
enqueue(b)	ab
dequeue()	b
enqueue(c)	bc
enqueue(d)	bcd
enqueue(e)	bcde
dequeue()	cde
enqueue(f)	cdef
enqueue(g)	cdefg

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 implemented with `addLast(E)` and `removeFirst()` operations, respectively, when a singly linked list is used.

Array list

What if we implement a queue using an array list? In this case, `enqueue(element)` would be `addLast(element)` and `dequeue()` would be `removeFirst()`. The `enqueue()` with array list is fine, and the only issue to note is that we need to use a larger underlying array in the case that the array is full. The problem is the `dequeue()` since `removeFirst()` is inefficient for array lists since we have to shift all the remaining elements.

Expanding Array

A better way to use an array to implement a queue would be to relax the requirement that the front of the queue is at position 0. Instead of shifting when we dequeue, we keep track of an index `head` which is the index of the next item to be removed.[5] We also keep track of `size`. Then, `tail = head + size - 1`. Note that when the queue is initialized, `tail == -1`. Also note that we need to expand the array when it is full. Finally, 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.

	01234567	head	tail	size
	----	0	-1	0
enqueue(a)	a---	0	0	1
enqueue(b)	ab--	0	1	2
remove()	-b--	1	1	1
enqueue(c)	-bc-	1	2	2
enqueue(d)	-bcd	1	3	3
enqueue(e)	-bcde---	1	4	4
remove()	--cde---	2	4	3
enqueue(f)	--cdef--	2	5	4
enqueue(g)	--cdefg-	2	6	5

The problem with this approach is that, when we remove an element from the array, we never use that array position again. This is an inefficient usage of space.

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

Circular array

A better approach is treat the array as *circular*, so that the last array position (`length-1`) is 'followed' by position 0. The relationship between indices becomes

$$\texttt{tail} = (\texttt{head} + \texttt{size} - 1) \ \% \ \texttt{length}.$$

Note that in the initial state, we have `size == 0` and `head == 0`, and so the formula implies that `tail` has value `length - 1`. See example below.

The algorithm for dequeueing is as follows. Note that it does not change `tail`.

```
dequeue(){          //  check that size > 0  (omitted)
   element = queue[head]
   head = (head + 1) % length
   size = size - 1
   return element
}
```

One subtlety here is that if there is just one element in the queue, then `head == tail` before we remove that element. So if remove this single element, then we advance the `head` index which will leave `head` with a *bigger* value than `tail` (unless `head` and `tail` were `length-1` in which case `head` becomes 0).

What about enqueueing? The next available position is `(head + size) % length`, but only if `size < length` since if `size == length` then the array is full. In the case that the array is full, the length of the array needs to be increased before we can add the element.

Take the above example and suppose that the array has `length = 4`. As always, we have

$$\texttt{tail} = (\texttt{head} + \texttt{size} - 1) \ \% \ \texttt{length}.$$

```
                  ----------------------------------------
                  0123                head    tail   size

                  ----                  0       3      0    tail == -1 % 4 == 3
enqueue(a)        a---                  0       0      1
enqueue(b)        ab--                  0       1      2
remove()          -b--                  1       1      1
enqueue(c)        -bc-                  1       2      2
enqueue(d)        -bcd                  1       3      3
enqueue(e)        ebcd                  1       0      4
dequeue()         e-cd                  2       0      3
enqueue(f)        efcd                  2       1      4
```

The next instruction is `enqueue(g)`. If we were simply to double the length of the array

```
efcd----
```

then it would be awkward to add the `g` since it is supposed to go after `f`. Instead, we copy the elements to the larger array such the head is at position 0 in the array.

```
cdef----
```

so that `head == 0, tail == 3`.

So the algorithm for enqueueing would go like this:

```
 enqueue( element ){
     if ( size == length)  {    //  increase length of array
        create a bigger array tmp[ ]    //  e.g. 2*length
        for i = 0  to length  - 1       //  'length' of queue (not of tmp)
            tmp[i] = queue[ (head + i) % length ]
        head = 0
        queue = tmp     //   'length' of queue is now twice as big as before
     }
     queue[size] = element
     queue.size = queue.size + 1
}
```

There was some discussion in class about whether this is the best way to copy the elements to the bigger array. Other suggestions were made. I am not claiming that the method above is the only way to do it. Rather, I am claiming that you need to copy all the elements of the queue from the full small array to the bigger array anyhow, so there no gain in efficiency is doing it any other way than what I suggested above. (I also claimed that other ways of doing it that seem right, in fact have subtle problems with it.)

ADT's, the Java API's, and interfaces

I finished the lecture by discussing some terminology and three related concepts.

ADT's

I have discussed three ADTs: lists, stacks, and queues. ADTs are not formal mathematical things, and they are not programming language specific. Rather, they are abstract things that exist in our heads only. They are a tool for thinking about algorithms and writing them out in pseudocode. This is helpful because algorithms can be complicated and subtle, even at a high level and having to think about coding them correctly in some particular programming language just makes things more complicated. If the implementation details don't matter because we are thinking and working at a highl level, then it is better to leave them aside and work with ADTs. Note that you are free to include whatever detail you like in an ADT. Certain operations might be $O(N)$ with one implementation of an ADT but $O(1)$ with another implementation and these details might be part of the description of the ADT too. The point is that the ADT gives the 'user' the information that they need, but doesn't burden the user with implementation details that they don't need.

Java API

By this point in the course, you should have checked out the Java API for various classes, such as the `ArrayList` or `LinkedList` classes. The Java API for a class is similar to an ADT in that that the Java API specifies the methods for the class, and what the methods do, but it doesn't

specify the details on the implementation. The "I" in API stands for interface (application program interface) and the meaning is that the API for a class contains all the information that the user gets to know about that class.

Java `interface`

In Java, the word `interface` is also a reserved word (or key word) and the meaning of interface is quite different there. An `interface` is like a class in that it contains a set of method definitions. Specifically a method is defined formally by a return type, and by a method *signature* (a method name, and method argument types in a particular order). An `interface` does not and cannot include an implementation of the methods, however. Rather, in Java one needs to define a class that `implements` the interface where `implements` is another reserved word in Java.

For example, there is a `List` interface (`https://docs.oracle.com/javase/7/docs/api/java/util/List.html`) which is implemented by the `ArrayList` and `LinkedList` classes. For example, the definition of the `ArrayList` class starts out:

`class ArrayList<T> implements List<T>`

and similarly for the `LinkedList` class.

For each method defined in the `List` interface, there is an implemented method in the `ArrayList` and `LinkedList` class. 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()`

Since `ArrayList<T>` and `LinkedList<T>` implement this interface, they must implement all the methods in this interface. These two classes have other methods as well, which are not defined in the `List` 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 may not care which. You want to be flexible, so that the code will work regardless of which type is used. In this case, you can use the generic Java interface `List<T>`. For example,

```
void myMethod( List<String> list ){
   :
   list.add("hello");
   :
}
```

Java allows you to do this. The compiler will see the `List` 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. As long as `list` only invokes methods from the `List` interface, the compiler will not complain. I will say more about this works at the end of the course when we discuss object oriented design in Java.

Java Stack and Queue

Java has a `Stack` class `https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html` It has the usual stack methods (push, pop) as well as common ones that make it more useful (isEmpty and peek). It also has one which is not stacklike, namely `search`. This method checks the stack for a particular object and if it is there then it returns the position in the stack. (Frankly I don't know why the people who invented the Java language chose to include this non-stack method, but they must have had their good reason.)

Java does not have a `Queue` class. Rather, `Queue` is an interface
`https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html`
which is implemented by several different queue classes. It has methods for enqueueing and dequeueing but these are given other names than *enqueue* and `dequeue`. In the lecture I used the terms *enqueue* and `dequeue` since they are traditional in CS.

I mention the Java `Stack` class and `Queue` interface for your curiosity. We will not be discussing them further in the course.

The next core topic in the course is *recursion*. We will look at a number of recursive algorithms and we will analyze how long they take to run. Recursion can be a bit confusing when one first learns about it. One way to understand recursion is to relate it to a proof technique in mathematics called *mathematical induction*, which is what I'll cover today.

Before I introduce induction, let me give an example of a statement that you have seen before, along with a proof. The proof is slightly different from the one I gave in the slides.

$$\text{For all } n \geq 1, \qquad 1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2}.$$

One trick for showing this statement is true is to add up two copies of the left hand side, one forward and one backward:

$$1 + 2 + 3 + \cdots + (n-1) + n$$
$$n + (n-1) + ... + 3 + 2 + 1.$$

Then pair up terms:

$$1 + n, 2 + (n-1), 3 + (n-2), ...(n-1) + 2, n + 1$$

and note that each pair sums to $n+1$ and there are n pairs, which gives $(n+1) * n$. We then divide by 2 because we added up two copies. This proves the statement above.

Mathematical induction

The above proof requires a trick, and many proofs in mathematics are like that – they use a specific trick that seems to work only in a few cases. Mathematical induction is different in that it is a general type of proof technique. To understand a proof by mathematic induction, you need to understand the logic of the proof technique *in general*.

Mathematical induction allows one to prove statements about positive integers. We have some proposition $P(n)$ which is either true or false and the truth value may depends on n. We want to prove that: "for all $n \geq n_0$, $P(n)$ is true", where n_0 is some constant that we state explicitly. In the above example, this constant is 1, but sometimes we have some other constant that is greater than 1.

A proof by *mathematical induction* has two parts, and one needs to prove both parts.

1. a *base case*: the statement $P(n)$ is true for $n = n_0$.

2. *induction step*: for any $k \geq n_0$, *if* $P(k)$ is true, *then* $P(k+1)$ must also be true.

   When we talk about $P(k)$ in step 2, we refer to it as the "induction hypothesis". Note that $P(k)$ is just $P(n)$ with $n = k$, *i.e.* it is the same proposition, but we are using parameter k instead of n to emphasize that we're in the context of proving the induction step.

The logic of a proof by mathematical induction goes like this. Let's say we can prove both the base case and induction step. Then, $P(n)$ is true for the base case $n = n_0$, and the induction step implies that $P(n)$ is true for $n = n_0 + 1$, and applying the induction step again implies that the statement is true for $n = n_0 + 2$, and so on forever for all $n \geq n_0$.

Example 1

Statement: for all $n \geq 1$,

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

Proof: The base case, $n_0 = 1$, is true, since

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

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

$$\begin{aligned}
\sum_{i=1}^{k+1} i &= (\sum_{i=1}^{k} i) + (k+1) \\
&= \frac{k(k+1)}{2} + (k+1), \text{ by induction hypothesis that P(k) is true} \\
&= (k+1)(\frac{k}{2} + 1) \\
&= \frac{1}{2}(k+1)(k+2)
\end{aligned}$$

and so $P(k+1)$ is also true. This proves the induction step. The proof is complete, since we have proven the base case and the induction step.

Example 2

Statement: for all $n \geq 3$,

$$2n + 1 < 2^n.$$

Proof: The base case $n_0 = 3$ is easy to prove, i.e. $7 < 8$. (Note that the base case $n_0 = 2$ would not be correct, nor would $n_0 = 1$. That's why we chose $n_0 = 3$.)
To prove the induction step, let k be any integer such that $k \geq 3$. We hypothesize that $P(k)$ is true and show that it would follow that $P(k+1)$ is also true. Note that $P(k)$ is the inequality

$$2k + 1 < 2^k$$

and $P(k+1)$ is the inequality

$$2(k+1) + 1 < 2^{k+1}.$$

To prove $P(k+1)$ we work with the expression on the left side of the inequality:

$$\begin{aligned}
2(k+1) + 1 &= 2k + 3 \\
&= (2k+1) + 2 \\
&< 2^k + 2, \quad \text{by induction hypothesis that } P(k) \text{ is true} \\
&< 2^k + 2^k, \quad \text{since } 2 < 2^k, \text{when } k \geq 3 \\
&= 2^{k+1} .
\end{aligned}$$

Thus, if $P(k)$ is true, then it must be that $P(k+1)$ is true also.

[ASIDE: You might be asking yourself, how did I know to use the inequality $2 < 2^k$? The answer is that I knew what inequality I eventually wanted to have, namely $P(k+1)$. Experience told me how to get there.

Example 3

Statement: 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 that inequality $k^2 < 2^k$ holds, where $k \geq 5$. We show that if $P(k)$ is true, then $P(k+1)$ must also be true, namely $(k+1)^2 < 2^{k+1}$. We start with the left side of $P(k+1)$

$$\begin{aligned}(k+1)^2 &= k^2 + 2k + 1 \\ &< 2^k + 2k + 1, \text{ by induction hypothesis, for } k \geq 5 \\ &< 2^k + 2^k, \text{ from Example 2} \\ &= 2^{k+1}\end{aligned}$$

which proves the induction step.

Note that the base case choice is crucial here. The statement $P(n)$ is just not true for $n = 0, 1, 2, 3, 4$. Also, note that the induction step happens to be valid for a larger range of k, namely $k \geq 3$ rather than $k \geq 5$. But we only needed it for $k \geq 5$.

Example 4: upper bound on Fibonacci numbers

Consider the Fibonnacci[6] sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ... where

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

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

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

Statement:

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

Proof:

The statement $P(k)$ is "F(k) $< 2^k$". We take the base case to be two values $n_0 = 0, 1$. By definition, $F(0) = 0, F(1) = 1$. Since $F(0) = 0 < 2^0$ and $F(1) = 1 < 2^1$, $P(n)$ is true for both base cases.

The induction hypothesis $P(k)$ is that $F(k) < 2^k$ where $k \geq 2$. For the induction step, we hypothesize that $P(k)$ is true for some k and we show this would imply $P(k+1)$ must also be true,

[6] http://en.wikipedia.org/wiki/Fibonacci_number

that is, $F(k+1) < 2^{k+1}$. Again, we start with the left side of this inequality:

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

and so $P(k)$ is true indeed implies that $P(k+1)$ is true, and so we are done.

Next lecture, we will look at the technique of recursion and show how it is related to the idea of mathematical induction.

Recursion

Recursion is a technique for solving problems in which the solution to the problem of size n is based on solutions to versions of the problem of size smaller than n. Many problems can be solved either by a recursive technique or non-recursive (e.g. iterative) technique, and often these techniques are closely related. In the next few lectures, we'll look at several examples. We'll also argue that recursion is closely related to mathematical induction.

Example 1: Factorial

The factorial function is defined as follows:

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

Unlike the sum of numbers to n which we discussed last class, there is no formula that gives us the answer of taking the product of numbers from 1 to n, and we need to compute it. Here is an algorithm (written in Java) for computing it.

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

Here is another way to define and compute $n!$ which is more subtle, namely if $n > 1$, then

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

and here is the corresponding algorithm coded in Java which is *recursive*. Note that the method `factorial` calls itself.

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

Recursive algorithms can't keep called themselves *ad infinitum*. Rather, they need to have a condition which says when to stop. This is called a *base condition*. For the `factorial` function, the base condition is that the argument is 1. Anytime you write a recursive algorithm, make sure you have a base condition and make sure you reach it. Typically this is ensured by having the parameter of the recursive call be smaller, *e.g.* `n-1` rather than `n` in the case of `factorial`.

Let's use mathematical induction to convince ourselves that the factorial algorithm is correct.

Claim: The recursive `factorial` algorithm indeed 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.

Second, the induction step: The induction hypothesis is that `factorial(k)` indeed returns $k!$. We want to show it follows that `factorial(k+1)` returns $(k+1)$!. But this is easy to see by inspection, since the induction hypothesis implies that the algorithm returns $(k+1) * k!$, which is just $(k+1)!$.

Example 2: 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 function $F(n)$ is defined in terms of itself, and so it is recursive.

Here is an iterative algorithm for computing the n-th Fibonacci number. We start at $n = 0, 1$ and move forward (assuming $n > 0$).

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

```
fibonacci(n){    //   assume n > 0
   if ((n == 0) || (n == 1))
      return n
   else
      return fibonacci(n-1) + fibonacci(n-2)
}
```

Here is a proof that this recursive algorithm for computing the nth Fibonaccie number is correct. First, the base case: If the parameter `n` is 0 or 1, then the algorithm returns 0 or 1, respectively, which is correct.

Second, the induction step: the induction hypothesis is that `fibonacci(k)` and `fibonacci(k-1)` returns the kth and $(k-1)$-th Fibonacci number, for any $k \geq 1$. We want to show that this implies `fibonacci(k+1)` returns the $(k+1)$th Fibonacci number. But again this is easy to see by inspection, since the algorithm returns the sum of `k-1` and `k` Fibonacci numbers which is the `k+1` Fibonacci number.

The recursive version turns out to be very slow since it ends up calling `fibonacci` on the same parameter n *many* times, which is unnecessary. For example, suppose you are asked to compute the 247-th Fibonacci number. `fibonacci(247)` calls `fibonacci(246)` and `fibonacci(245)`, and `fibonacci(246)` calls `fibonacci(245)` and `fibonacci(244)`. But now notice that `fibonacci(245)` is called twice. The problem here is that every time you want to compute `fibonacci(k)` where `k > 1`, you need to do *two* recursive calls. This leads to a combinatorial explosion in the number of calls. (I haven't provided a formal calculation here in exactly how many calls would be made, but hopefully you get the idea that many repetitions of the same calls occur. If not, see the picture in the slides.)

Example 3: reversing a list

Let's next revisit a few algorithms for lists, and examine recursive versions. The first example is to reverse a list (see linked list exercises for an iterative version). The idea can be conveyed with the following picture. To reverse the list,

```
(a b c d e f g)
```

we can remove the first element `a` and reverse the remaining elements,

```
a   (b c d e f g)  ---->   a  (g f e d c b)
```

and then add the removed element at the end of the (reversed) list.

```
(g f e d c b a)
```

Here is the pseudocode. I have written it so that `list` is an argument to the various methods that are called, which is different notation from what we used in the list lectures.

```
reverse(list){     //   assume n > 0
   if list.size == 1     // base case
      return  list
   else{
      firstElement = removeFirst(list)
      list = reverse(list)        // list has only n-1 elements
      return addLast(list, firstElement)
   }
}
```

And here is some Java code for a class that implements the List methods:

```
public void reverse(){
    if (this.size > 1){
       E e = this.removeFirst();
       this.reverse();
       this.addLast(e);
    }
}
```

Example 4: sorting a list

Recall the selection sort algorithm. The basic idea was to maintain two lists: a sorted list, and a 'rest' list which is unsorted. The algorithm loops repeatedly through the rest list, removes the minimum element each time, and adds it to the end of the sorted list. (Since the sorted list consists of elements that are all smaller than or equal to elements in the rest list, all the elements of the rest list will eventually be added after elements in the sorted list.) The recursive algorithm below is essentially the same idea. Remove the minimum element, sort the rest list (recursively), and add the minimum element to the front of the sorted rest list (that is, the sorted rest list will be after any minimum elements that have been removed). That may sound complicated, but look how simple the code is:

```
sort(list){  //  assumes list.size >= 1
   if list.size == 1
      return list         // base case
   else{
      minElement = removeMin(list)
      list = sort(list)
      return addFirst(list, minElement)
   }
}
```

Tower of Hanoi

Let's now turn to an example of a problem in which a recursive solution is very easy to express, and a non-recursive solution is very difficult to express (and I won't even both with the latter). 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 increases from top to bottom. 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.
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 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
```

Here I will label the stacks `A, B, C`. For example, `tower(1,A,B,C)` would produce to the following sequence of instructions:

```
tower(0,A,C,B)       //  don't do anything
move from A to B
tower(0,C,B,A)       //  don't do anything
```

The two calls `tower(0,*,*,*)` would do nothing since the condition $n > 0$ is not met.
What about `tower(2,A,B,C)` ? This would produce the following sequence of instructions:

```
tower(1,A,C,B)
move from A to B
tower(1,C,B,A)
```

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 A to C
move from A to B
move from C to B
```

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

```
tower(2,A,C,B)
move from A to B
tower(2,C,B,A)
```

The initial state is:

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

So first we do `tower(2,A,C,B)`, which takes 3 moves:

```
      **
      ***       *
      ---       ---         ---      (after moving disk from A to B)



      ***       *           **       (after moving disk from A to C)
      ---       ---         ---


                            *
      ***                   **       (after moving from B to C)
      ---       ---         ---

  Next we do "move from A to B":

                            *
                ***         **
      ---       ---         ---      (after moving from A to B)

Then we call  tower(2, C, B, A) which does the following 3 moves:

      *         ***         **
      ---       ---         ---      (after moving from C to A)


                **
      *         ***
      ---       ---         ---      (after moving from C to B)


                 *
                 **
                 ***
      ---        ---       ---      (after moving from A to B)

   and we are done!
```

Claim: 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 we move one disk at a time, and that the n disks are eventually moved from the `start` to `finish`. The proof is by mathematical 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 rules for these k disks. (This is 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 rules (again, by the induction hypothesis). This completes the proof.

[ASIDE: In Sec. 001 lecture, I mistakenly skipped the slides for the following calculation.]

How many moves does `tower(n, ...)` take? `tower(1, ...)` takes 1 move. `tower(2, ...)` takes 3 moves, namely two recursive calls to `tower(1, ...)` which take 1 move each, plus one move. `tower(3, ...)` makes two recursive calls to `tower(2, ...)` which we just said takes 3 moves each, plus one move, for a total of 3*2 + 1 = 7. Similarly, `tower(4, ...)` makes two recursive calls to `tower(3, ...)` which we just said takes 7 moves each, plus one move, for a total of 7*2 + 1 = 15. And so on... `tower(5, ...)` takes 2*15 + 1 = 31 moves, and `tower(6, ...)` takes 2*31 + 1 = 63 moves. In general, `tower(n, ...)` makes two recursive calls to `tower(n-1, ...)` plus one move. So one can prove by induction that `tower(n, ...)` takes $2^n - 1$ moves.

Recursion and the Call Stack

Back in lecture 8, we discussed stacks and I mentioned the "call stack". 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". You will learn more about this in COMP 273, but it is worth mentioning now to take some of the mystery out of how recursion is implemented in the computer.

As an example, suppose the program calls `factorial(6)`. This leads to a sequence of recursive 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 call 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
```

Using the Eclipse debugger and setting breakpoint within the `factorial()` method, you can see how the stack evolves. I strongly recommend that you do this and verify how this works.

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

I would do the same for the Tower of Hanoi.

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

See slides for screen shots.

I should emphasize that the call stack is not some abstract idea, but rather it is a real data structure that the software that runs your Java program (called the "Java Virtual Machine"). The stack consists of *stack frames*, one for each method that is called. In the case of recursion, there is one stack frame for each time the method is called.

The stack frame contains all the information that is needed for that method. This includes local variables declared and used by that method, parameters that are passed to the method, and information about where the method returns when it is done, that is, who called the method.

You will learn about the call stack and stack frames work in much more detail in COMP 273. I mention it here because I want you to get familiar with the idea, and because I want you to be aware that the call stack and stack frame really exist. Indeed most decent IDEs will allow you to examine the call stack (see slides) and at least the current stack frame, i.e. the frame on top of the call stack.

Converting a number to its binary representation

Back in lecture 2, we saw how to convert a decimal number $n \geq 1$ to binary. Here I'll write the algorithm slightly differently so we are printing out the bits from low to high.

```
toBinary(n){  //  iterative
    i = 0
    while n > 0 {
      print  n % 2     //  bit i
      n = n/2
      i = i+1
    }
}
```

Next we write the algorithm recursively.

```
   toBinary(n){            //  algorithm assumes input n >= 1
        if  n >= 1{               //  otherwise base case, and do nothing
           print  n % 2
           toBinary( n/2 )
        }
   }
```

Note that this indeed prints the bits from the lowest order to highest order. However, if we were two switch the print statement with the `toBinary` call, then we would print from highest order to low. In that case, the printing wouldn't start until we have exited the `toBinary(n)` where `n` is either 1 or 2.

Finally, recall from lecture 2 that the iterative version of the algorithm loops about $\log_2 n$ times, where n is the original number. For the same reason, the recursive version has about $\log_2 n$ recursive calls, one for each bit of the binary representation of the number. Today we will see a few more algorithms that have an $O(\log_2 n)$ property.

Power (x^n)

Let x be some number (say a positive integer for simplicity) and consider how to compute x raised to some power n. An iterative (non-recursive) method for doing it is:

```
power(x,n){    //  iterative
    result = 1
    for i = 1 to n
       result = result * x
    return  result
}
```

A more interesting way to compute x^n is to use recursion. We could do what we did last lecture and rite $x^n = x^{n-1}x$, but let's do it in a way that is more interesting. 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 to compute x^9.

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).

```
power(x,n){    //  recursive
   if n == 0
        return 1
   else if n == 1
      return x
   else{
       tmp = power(x,n/2)
       if n is even
          return  tmp*tmp
       else
          return  tmp*tmp*x
    }
}
```

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. Note the similarity to converting a number (in this case 18) to binary where we decide if each bit is either 0 or 1. (In the slides, I used 243 as an example.)

Notice that the number of recursive calls will be $O(\log_2 n)$, 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.

For the same reason, the number of multiplies that are executed will be between $\log_2 n$ and $2\log_2 n$. Why? If the original n's binary representation has all 1's, e.g. $63 = (111111)_2$, then two multiplications will be executed at each recursive call since the parameter n will always be odd. If the original n's binary representation is all 0's (except the high order bit), e.g. $n = (100000000)_2$, then only one multiplication will be computed at each recursive call. [ASIDE: Note that the highest order bit is a 1. This bit corresponds to the base case of the recursion, and no multiplication is computed in that case. In the Sec. 001 lecture, I mistakenly forgot about that.]

As I discussed in the lecture slides and I will discuss in an exercise, we should not be fooled by the above argument into thinking that we can compute x^n in time $O(\log_2 n)$. The issue here is

that we have only discussed the number of multiplications. But the time taken for a multiplication depends on the number of digits in the two numbers being multiplied. It turns out that the recursive algorithm – although quite interesting – takes just as long as the if we compute x^n by successive multiplications by x. But as I as briefly will mention in the exercise, we can use this recursion trick to compute something similar which is very useful, and which we can indeed compute in time $O(\log_2 n)$.

Let's now turn to a different problem, which also is $O(\log_2 n)$ for similar reasons as what we've discussed today.

Binary search in a sorted (array) list

Suppose we have an array list of n elements which are *already* sorted from smallest to largest. These could be numbers or strings sorted alphabetically. Consider the problem of searching for a particular element in the list, and returning the index in $0, \ldots, n-1$ of that element, if it is present in the list. If the element is not present in the list, then we return -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 *linear search* is wasteful since it doesn't take advantage of the fact that the array is already sorted.

A much faster method, called *binary search* takes advantage of the fact that the array is sorted. You are familiar with this idea. Think of when you look up a word in an index in the back of a book. Since the index is sorted alphabetically, you don't start from the beginning and scan. Instead, you jump to somewhere in the middle.[7] If the word you are looking for comes before those on the page, then you jump to some index roughly in the middle of those elements that come before the one you jumped to, and otherwise you jump to a position in the middle of those that come after the one you jumped to. The *binary search* algorithm does essentially what I just described. Let's first present an iterative (non-recursive) version of binary search algorithm.

```
binarySearch(list, value){         // iterative
    low = 0
    high = list.size - 1
    while  low <= high  {
       mid = (low + high)/ 2    //  so mid == low, if high - low == 1
       if list[mid] == value
          return mid
       else{ if value < list[mid]
               high = mid - 1       //  high can become less than low
            else
               low  = mid + 1  }
    }
    return  -1     // value not found
}
```

[7] Of course, if you are looking for a word that starts with "b", then you don't just into the middle, but rather you start near the beginning. But let's ignore that little detail here.

For each pass through the `while` loop, the number of elements in the array that still need to be examined is cut by at least half. Specifically, if `[low, high]` has an odd number of elements (2k+1), then the new `[low, high]` will have k elements, or less than half. If `[low, high]` has an even number of elements (2k), then the new `[low, high]` has either k or k-1 elements, whih is at most half. Note that the new `[low, high]` does not contain the `mid` element.

It follows that for an input array with n elements, there are about $\log_2 n$ passes through the loop. This is the same idea as converting a number n to binary, which takes about $\log_2 n$ steps to do, i.e. the number of times we can divide n by 2 until we get 0.

The details of the above algorithm are a bit tricky and it is common to make errors when coding it up. There are two inequality tests: one is $\leq$ and one is $<$, and the way the `low` and `high` variables are updated is also rather subtle. For example, note that if the item is not found then the while loop exits when `high < low`. There is no explicit check for the case that `high == low`.

Here is a recursive version. Notice how the iterative and recursive versions of the algorithm are nearly identical. One difference is that the recursive version passes in the `low` and `high` variables.

```
binarySearch( list, value, low, high ){   // recursive
    if  low > high {
       return -1
    else{
       mid = (low + high) / 2
       if value == list[mid]
          return mid
       else if value < list[mid]
          return binarySearch(list, value, low, mid - 1 )
       else
          return binarySearch(list, value,  mid+1, high)
    }
}
```

If you were to implement this in Java, you probably wouldn't pass the `list` in as parameter since that would mean you were creating many new lists as you go. Let's not concern ourselves with this issue now, since I want you to focus here more on how `low` and `high` work and how the size of the list gets chopped in half each time. These are the main points to understand.

Today we looked at a few problems that all had $O(\log_2 n)$ behavior. The base 2 of the logarithm is due to chopping something in *half*, whether it is a number we are converting to binary or the exponent of a power, or the size of a sorted list. It is nice to see problems that having some similarities in their behavior but that also have some subtle differences. Next class we will look at another way in which we can chop a problem in half and we will see that a $\log_2 n$ factor arises there as well.

Mergesort

In lecture 7, we saw three algorithms for sorting a list of n items. We saw that, in the worst case, all of these algorithm required $O(n^2)$ operations. Such algorithms will be unacceptably slow if n is large.

To make this claim more concrete, consider that if $n = 2^{20} \approx 10^6$ i.e one million, then $n^2 \approx 10^{12}$. How long would it take a program to run that many instructions? Typical processors run at about 10^9 basic operations per second (i.e. GHz). So a problem that takes in the order of 10^{12} operations would require thousands of seconds of processing time. (Having a multicore machine with say 4 processors only can speed things up by a factor of 4 – max – which doesn't change the argument here.)

Today we consider an alternative sorting algorithm that is much faster that these earlier $O(n^2)$ algorithms. This algorithm is called *mergesort*. Here is the idea. If the list has just one number ($n = 1$), 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 (recursively, 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 to get

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

Here is pseudocode for the algorithm. Note that it uses a helper method `merge` which in fact does most of the work.

```
mergesort(list){
   if  list.length == 1
      return list
   else{
      mid = (list.size - 1) / 2
      list1 =  list.getElements(0,mid)
      list2 =  list.getElements(mid+1, list.size-1)
      list1 = mergesort(list1)
      list2 = mergesort(list2)
      return  merge( list1, list2 )
   }
}
```

Below is the merge algorithm. Note that it has two phases. The first phase initializes a new list (empty), steps through the two lists, (`list1`) and (`list2`), compares the front element of each list and removes the smaller of the two, and this removed element to to the back of the merged list. *See the detailed example in the slides for an illustration.*

The second phase of the algorithm starts after one of `list1` or `list2` becomes empty. In this case, the remaining elements from the non-empty list are moved to `list`. This second phase uses two `while` loops in the above pseudocode, and note that only one of these two loops will be used since we only reach phase two when one of `list1` or `list2` is already empty.

```
merge( list1,  list2){
   new empty list
   while  (list1 is not empty) & (list2 is not empty){
       if  (list1.first < list2.first)
          list.addlast( list1.removeFirst() )
       else
          list.addlast( list2.removeFirst() )
   }
   while list1 is not empty
       list.addlast( list1.removeFirst() )
   while list2 is not empty
       list.addlast( list2.removeFirst() )
   return  list
}
```

I have written the `mergesort` and `merge` algorithms using abstract list operations only, rather than specifying how exactly it is implemented (array list versus linked list). Staying at an abstract level has the advantage of getting us quickly to the main ideas of the algorithm: what is being computed and in which sequence? However, be aware that there are disadvantages of hiding the implementation details, i.e. the data structures. As we have seen, sometimes the choice of data structure can be important for performance.

mergesort is $O(n \log n)$

There are $\log n$ levels of the recursion, namely the number of levels is the number of times that you can divide the list size n by 2 until you reach 1 element per list. The number of instructions that must be executed at each level of the recursion is proportional to the number n of items in the list. This is most easily visualized using the example given in the slides, when I went through how all of the partitions and merges worked. There were $2 \log_2 n$ columns, half corresponding to the partitioning in to smaller lists and half corresponding to merges. The total number of elements in each columns is n. Making the columns in the merge steps require time roughly proportional to $n \log_2 n$ since there are $\log_2 n$ columns to be built. I will discuss this again a few lectures from now, when we study *recurrences*.

To appreciate the difference between the number of operations for the earlier $O(n^2)$ sorting algorithms versus $O(n \log n)$ for mergesort, 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 is significantly less than the time it takes to run bubble/selection/insertion sort, when n becomes large. Very roughly speaking, on a computer that runs 10^9 operations per second running mergesort on a list of size $n = 10^9$ would take in the order of minutes, whereas running insertion sort would take centuries.

Quicksort

Another well-known recursive algorithm for sorting a list is *quicksort*. This algorithm is quite similar to mergesort but there are important differences that I will highlight.

At each call of quicksort, one sorts a list as follows. An element known as `pivot` is removed from the current list. For example, the pivot might be the first element. The remaining elements in the list are then partitioned into two lists: `list1` which contains those smaller than the pivot, and `list2` which contains those elements that are greater than or equal to the pivot. The two lists `list1` and `list2` are recursively sorted. Then the two sorted lists and the pivot are concatenated into a sorted list containing the original elements (specifically, `list1` followed by `pivot` followed by `list2`).

```
quicksort(list){
   if  list.length <= 1
      return list
   else{
      pivot = list.removeFirst()   // or some other element
      list1 = list.getElementsLessThan(pivot)
      list2 = list.getElementsNotLessThan(pivot) // i.e. the rest
      list1 = quicksort(list1)
      list2 = quicksort(list2)
      return  concatenate( list1, pivot, list2 )
   }
}
```

Unlike mergesort, most of the work in quicksort is done prior to the recursive calls. Given a pivot element that has been removed from the list, the algorithm goes through the rest of the list and compares each element to `e`. This takes time proportional to the size of `list` in that recursive call, since one needs to examine each of the elements in the list and decide whether to put it into either `list1` or `list2`. The concatenation that is done after sorting `list1` or `list2` might also involve some work, depending on the data structure used.

In the lecture, I briefly discussed how the performance of quicksort can be bad in the worst case. I will fill in that discussion a few lectures from now when we discuss "recurrences".

There is lots more I could say about quicksort. For example, one common and simple implementation of quicksort uses a single array. That is, one can implement quicksort with no extra space required for copying the elements (unlike in mergesort, where it is very natural to use an extra array for the `merge` step). Sorting without using any extra space is called *in place* sorting.[8] If you would like to see how quicksort can be done in-place with an array, see `https://en.wikipedia.org/wiki/Quicksort`. I will not examine you on that, since I didn't present it in the lecture, but those of you who are Majoring in CS might want to give it 20-30 minutes so you get the basic idea.

[8]Bubble sort, selection sort, and insertion sort were also "in place" algorithms.

Recurrences

We have seen many algorithms thus far. For each one we have tried to express how many basic operations are required as a function of some parameter n which is typically the size of the input e.g. the number of elements in a list.

For algorithms that involve `for` loops, we can often write the number of operations of the loop component as a power of n, which corresponds to 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. The sorting algorithms from lecture 6 and the grade school multiplication algorithm are a few examples.

For recursive algorithms, it is less obvious how to express the number of operations as a function of the size of the input. We have given some examples of recursive algorithms in the last few lectures. For these examples and others, we would like to express in a more general way how the time it takes to solve the problem depends on n. That is what we will do next and next lectuer. In each case, we express a function $t(n)$ 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: reversing a list

Let $t(n)$ be the time it takes to reverse a list with n elements. Recall how this is done. You remove the first element of the list. Then, you take the remaining $n-1$ element list and recursively reverse them. Then you add the element you removed to the end of the reversed list.

Each recursive call reduces the problem from size n to size $n-1$. This suggests a relationship:

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

where the constant c is the time it takes in total to remove the first element from a list plus the time it takes to add that same element to the end of a list. We are not saying what c is. All that matters is that it is constant: it doesn't depend on n. (By the way, I am making an assumption here that the first element can be removed in constant time. If we are using an array list, then this is not so.)

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

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

This method is called *backwards substitution.* Note $t(0)$ is the base case of the recursion and done in constant time.

Informally, we say that $t(n)$ is $O(n)$ since the time it takes is roughly proportional to n. A few lectures from now, we'll say more formally what we mean by $O(\ )$.

[ASIDE: 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: sorting a list by finding the minimum element

Recall the recursive algorithm for sorting which found the smallest element in a list and removed it, then sorted the remaining $n-1$ elements, and finally added the removed element to the front of the list. We express the time taken, using the recurrence:

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

As I discussed in the lecture, we could write the recurrence more precisely as

$$t(n) = c_1 + c_2\ n + t(n-1)$$

since in each recursive call there is some constant c_1 amount of work, plus some amount of work $c_1 n$ that depends linearly on the size n of the list in that call. But let's keep it simple and consider just the first recurrence.

Solving by back substitution:

$$\begin{aligned}
t(n) &= c\ n + t(n-1) \\
&= c\ n + c \cdot (n-1) + t(n-2) \\
&= \ldots \\
&= c\ \{\ n + (n-1) + (n-2)\ + \cdots + (n-k)\} + t(n-k-1) \\
&= c\ \{\ n + (n-1) + (n-2)\ + \cdots + 2 + 1\} + t(0) \\
&= \frac{cn(n+1)}{2} + t(0)
\end{aligned}$$

This is $O(n^2)$ since the largest term here that depends on n is n^2.

Example 3: Tower of Hanoi

Recall the Tower of Hanoi problem. Let $t(n)$ be the number of disk moves. The recurrence relation is:

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

The c on the right side refers to the work that is done with the 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 for each

recursive call (making a stack frame, pushing it on the call stack, and then popping the call stack when the method exits). All this constant time work is bundled together as a single constant c. 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

$$\begin{aligned} t(n) &= c + 2t(n-1) \\ &= c + 2(c + 2t(n-2)) \\ &= c(1+2) + 4t(n-2) \\ &= c(1+2) + 4(c + 2t(n-3)) \\ &= c(1+2+4) + 8t(n-3) \\ &= ... \\ &= c(1+2+4+8+\cdots+2^{k-1}) + 2^k\ t(n-k) \\ &= c(1+2+4+8+\cdots+2^{n-1}) + 2^n t(0) \\ &= c(2^n - 1) + 2^n t(0) \end{aligned}$$

where we have used the familiar geometric series (recall lecture 2)

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

for the case that $x = 2$.

Since the largest term that depends on n is 2^n, we say that $t(n)$ is $O(2^n)$.

Example 4: binary search

Recall the recursive 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. Since the recursive call is on a list that is only half the size, we can express the time using the recurrence:

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

We suppose that n is a power of 2, i.e. $n = 2^k$, where $k = \log_2 n$.

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

So we say that binary search is $O(\log_2 n)$ since the largest term that depends on n is $\log_2 n$.

Logarithms: some review, some new

When you signed up for COMP 250, you probably didn't expect that you would need to think so much about logarithms. Surprise! As we saw last lecture, logarithms are very important in understanding how long different algorithsm take. Because you may be rusty on logarithms, here I will review some basic and important properties that you will need to know or at least be familiar with in COMP 250.

Let's begin with the definition. The logarithm is the inverse function of the exponential. If we consider take the exponential function for some base $b > 0$, namely

$$y = b^x,$$

then the inverse function

$$\log_b y \equiv x$$

is called the *logarithm* of that base b. Thus,

$$\log_b(b^x) \equiv x$$

$$b^{\log_b y} \equiv y.$$

Note that this is the *definition* of the log function, but this definition assumes that b^x is meaningful. I think you all agree that b^x is meaningful when x is an integer, but you are probably not so clear on exactly what it means when x is not an integer. For the latter case, indeed it is not so clear and you would need to arguments from MATH 242 Real Analysis to say exactly what it means. Alas, those not taking that course will just have to just trust the good people in the math department.

Here are a few basic properties of logs. We assume $a, b, c > 0$.

- $a^{b+c} = a^b a^c$
- $(a^b)^c = a^{bc}$
- $\log_b(a^c) = c \log_b a$
- $\log_b(ac) = \log_b a + \log_b c$

The next two properties compare logarithms of two different bases, that is, $\log_b x$ versus $\log_a x$.

- $\log_b x = (\log_b a)(\log_a x)$ Here is the proof:

  $$\log_b x = \log_b(a^{\log_a x}) = (\log_a x)(\log_b a)$$

- $a^{\log_b c} = c^{\log_b a}$ Here is the proof:

  $$(\log_b c)\ (\log_b a) = (\log_b a)\ (\log_b c)$$

  and so

  $$\log_b(a^{\log_b c}) \quad = \quad \log_b(c^{\log_b a})$$

  and so we can cancel the $\log_b$ on both sides and we're done.

Here we examine the recurrences for mergesort and quicksort.

Mergesort

Recall the mergesort 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 the merging requires iterating through the elements of each list. If n is even, then there are $\frac{n}{2} + \frac{n}{2} = n$ elements in the two lists. If n is odd then one of the lists has size one greater than the other, but there are still n steps to the merge.

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:

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

[ASIDE: If we were to consider a general n, then the correct way to write the recurrence would be:

$$t(n) = t(\ \lfloor \frac{n}{2} \rfloor\ ) + t(\ \lceil \frac{n}{2} \rceil\ ) + cn$$

where the $\lfloor \frac{n}{2} \rfloor$ means $floor(n/2)$ and $\lceil \frac{n}{2} \rceil$ means $ceiling(n/2)$, or "round down" and "round up", respectively. That is, rather than treating $n/2$ as an integer division and ignoring the remainder (rounding down always), we would be treating it as $n/2.0$ and either rounding up or down. In the lecture, I gave the example of $n = 13$ so $\lfloor \frac{n}{2} \rfloor = 6$ and $\lceil \frac{n}{2} \rceil = 7$. In COMP 250, we don't concern ourselves with this level of detail since nothing particualrly interesting happens in the general case, and we would just be getting bogged down with notation. The interesting aspect of mergesort is most simply expressed by considering the case that n is a power of 2.]

Let's solve the mergesort recurrent using backwards substitution:

$$\begin{aligned}
t(n) &= c\ n + 2\ t(\frac{n}{2}) \\
&= c\ n + 2\ (\ c\frac{n}{2} + \ 2t(\frac{n}{4})) \\
&= c\ n + c\ n + 4\ t(\frac{n}{4}) \\
&= c\ n + c\ n + 4\ (c\frac{n}{4}\ +\ 2\ t(\ \frac{n}{8}\ )) \\
&= c\ n + \ c\ n + \ c\ n + 8\ t(\frac{n}{8}) \\
&= c\ n\ k\ +\ 2^k\ t(\frac{n}{2^k}) \\
&= c\ n \log_2 n + \ n\ t(1), \quad \text{when } n = 2^k
\end{aligned}$$

which is $O(n\ \log_2 n)$ since the dominant term that depends on n is $n\ \log_2 n$.

What is the intuition for why mergesort is $O(n \log_2 n)$? Think of the merge phases. The list of size n is ultimately partitioned down into n lists of size 1. If n is a power of 2, then these n lists are merged into $\frac{n}{2}$ lists of size 2, which are merged into $\frac{n}{4}$ lists of size 4, etc. So there are $\log_2 n$ "levels" of merging, and each require $O(n)$ work. Hence,

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.

On the base case of mergesort

In the mergesort algorithm last week, we had a base case of $n = 1$. What if we had stopped the recursion at a larger base case? For example, suppose that when the list size has been reduced to 4 or less, we switch to running bubble sort instead of mergesort. Since bubble sort is $O(n^2)$, one might ask whether this would cause the mergesort algorithm to increase from $O(n \log_2 n)$ to $O(n^2)$. Let's solve the recurrence for mergesort by assuming $t(n) = 2t(\frac{n}{2}) + c_1 n$ when $n > 4$ and but that some other $t(n)$ holds for $n \leq 4$.

Assume n is a power of 2 (to simplify the argument). We want to stop the backsubstitution at $t(4)$ on the right side. So we let k be such that $\frac{n}{2^k} = 4$, that is, $2^k = \frac{n}{4}$.

$$\begin{aligned} t(n) &= c\ n + 2\ t(\frac{n}{2}) \\ &= \ldots \\ &= c\ n\ k\ +\ 2^k\ t(\frac{n}{2^k}), \quad \text{and letting } 2^k = \frac{n}{4} \text{ gives...} \\ &= c\ n(\log_2 n - 2) + \frac{n}{4}\ t(4) \\ &= c\ n \log_2 n\ -\ 2cn + \frac{n}{4}\ t(4) \end{aligned}$$

which is still $O(n \log_2 n)$ since the dominant term that depends on n is $n \log_2 n$.

The subtle part part of this problem is that you may be thinking that if we are doing say bubblesort when $n \leq 4$ then we need to somehow put an n^2 dependence in somehwhere. But you don't need to do this! Since we are only switching to bubblesort when $n \leq 4$, the term $t(4)$ is a constant. This is the time it takes to solve a problem of size 4.

Note: while this might seem like a toy problem, it makes an important point. Sometimes when we write recursive methods, we find that the base case can be tricky to encode. If there is a slower method available that can be used for small instances of the problem, and this slower method is easy to encode, then use it!

Quicksort

Let's now turn to the recurrence for quicksort. Recall the main idea of quicksort. We remove some element called the pivot, and then partition the remaining elements based on whether they are smaller than or greater than the pivot, recursively quicksort these two lists, and then concatentate the two, putting the pivot in between.

In the best case, the partition produces two roughly equal sized lists. This is the best case because then one only needs about $\log n$ levels of the recursion and approximately the same recurrence as mergesort can be written and solved.

What about the worst case performance? If the element chosen as the pivot happens to be smaller then all the elements in the list, or larger then all the elements in the list, then the two lists are of size 0 and $n-1$. If this poor splitting happens at every level of the recursion, then performance degenerates to that of the $O(n^2)$ sorting algorithms we saw earlier, namely the recurrence becomes

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

Solving this by backstitution (see last lecture) gives

$$t(n) = c\frac{n(n+1)}{2} + t(0)n$$

which is $O(n^2)$.

Why is quicksort called "quick" when its worst case is $O(n^2)$? In particular, it would seem that mergesort would be quicker since mergesort is $O(n \log n)$, regardless of best or worst case.

There are two basic reasons why quicksort is "quick". One reason is that the first case is easy to avoid in practice. For example, if one is a bit more clever about choosing the pivot, then one can make the worst case situation happen with very low probability One idea for choosing a good pivot is to examine three particular elements in the list: the first element, the middle element, and the last element (`list[0], list[mid], list[size-1]`. For the pivot, one sorts these three elements and takes the middle value (the median) as the pivot. The idea is that it is very unlikely for *all three* of these elements to be among the three smallest (or three largest). In particlar, if the list happens to be close to sorted (or sorted in the wrong direction) then the "median of three" will tend to be close the median of the entire list. *Note that the best split occurs if we take the pivot to be the median of the whole list.* In practice, such a simple idea works very well, and partitions have close to even size.

The second reason that quicksort is quick is that, if one uses an array list to represent the list, then it is possible to do the partition *in place*, that is, without using extra space. One needs to be a bit clever to set this up, but it is straightforward to implement. I've decided not to cover the details this year, but if you feel like you are missing out, then do check it out: `https://en.wikipedia.org/wiki/Quicksort`

The "in place" property of quicksort is a big advantage, since it reduces the number of copies one needs to do. By contrast, the straightforward implementation of mergesort requires that we use a second array and merge the elements into it. This leads to lots of copying, which tends to be slow. There are clever ways to implement mergesort which make it run faster, but the details are way beyond the scope of the course. Besides, experimental results have shown that, in practice, quicksort tends to be faster than mergesort even if the 'in place' mergesort method is used. Quicksort truly is very quick (if done in place).

Exercise

In the lecture, I asked you to solve this one:

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

This is similar to binary search, but now we have to do n operations during each call.

What do you predict? Is this $O(\log_2 n)$ or $O(n)$ or $O(n^2)$ or what?

Here is the solution:

$$
\begin{aligned}
t(n) &= n + t(\frac{n}{2}) \\
&= n + \frac{n}{2} + t(\frac{n}{4}) \\
&= n + \frac{n}{2} + \frac{n}{4} + t(\frac{n}{8}) \\
&= n + \frac{n}{2} + \frac{n}{4} + \cdots \frac{n}{2^{k-1}} + t(\frac{n}{2^k}) \\
&= n + \frac{n}{2} + \frac{n}{4} + \cdots + 4 + 2 + t(1), \quad \text{when } 2^k = n \\
&= n + \frac{n}{2} + \frac{n}{4} + \cdots + 4 + 2 + 1 - 1 + t(1) \\
&= n \sum_{i=0}^{\log_2 n} 2^i + t(1) - 1 \\
&= (2^{(\log_2 n)+1} - 1)/(2-1) + t(1) - 1 \\
&= 2n - 1 \quad + \ t(1) - 1
\end{aligned}
$$

We have seen several algorithms in the course, and we have loosely characterized their runtimes $t(n)$ in terms of the size n of the input. We say that the algorithm takes time $O(n)$ or $O(\log_2 n)$ or $O(n \log_2 n)$ or $O(n^2)$, etc, by considering how the runtime grows with n, ignoring constants. This level of understanding of $O(\ )$ is usually good enough for characterizing algorithms. However, we would like to be more formal about what this means, and in particular, what it means to ignore constants.

An analogy from Calculus: limits

A good analogy for informal versus formal definitions is one of the most fundamental ideas of Calculus: the limit. In your Cal courses, you learned about various types of limits and you learned methods and rules for calculating limits e.g. squeeze rule, ratio test, l'Hopital's rule, etc.

You were also given the formal definition of a limit. This formal definition didn't play much role in your Calculus class, which was more concerned with using rules about limits than understanding where these rules come from. But if you look back at your Calculus textbook then you'll see the formatl definitions were there. Moreover if you go further ahead in your study of mathematics then you will find this formal definition comes up again[9].

Here is the formal definition of the limit of a sequence. *A sequence $t(n)$ of real numbers converges to (or has a limit of) a real number t_∞ means that the following holds: for any $\epsilon > 0$, there exists an n_0 such that for all $n \geq n_0$, $\mid t(n) - t_\infty \mid \ < \epsilon$.*

This definition is subtle. There are two "for all " logical quantifiers and there is one "there exists" quantifier, and the three quantifiers have to be ordered in just the right way. The statement is saying that if you take any finite interval centered at the t_∞, namely $(t_\infty - \epsilon, t_\infty + \epsilon)$, then the values $t(n)$ of the sequence will all fall in that interval once n exceeds some finite value n_0.

Here is the analogy to big O. We will have a set of rules that we can use to say that some function $t(n)$ is big O of some other function e.g. $t(n)$ is $O(\log_2 n)$. These rules come from a formal definition of big O. We'll look at that formal definition next. This formal definition has a similar flavour to the formal definition of the limit of a sequence in Calculus. But the formal definition of big O says something quite different. Next lecture, I'll discuss the connection to limits again.

Big O

Let $t(n)$ be a well-defined sequence of integers. In the last several lectures, such a sequence represented the time or number of steps it takes an algorithm to run as a function of n which is the size of the input. However, today we put this interpretation aside and we just consider $t(n)$ to be a sequence of numbers, without any meaning. We will look at the behavior of this sequence $t(n)$ as n becomes large.

[9] in particular, starting in Real Analysis (e.g. MATH 242 at McGill)

Definition (preliminary)

Let $t(n)$ and $g(n)$ be two sequences of integers[10], where $n \geq 0$. We say that $t(n)$ *is asymptotically bounded above by* $g(n)$ if there exists a positive number n_0 such that,

$$\text{for all } \; n \geq n_0, \;\; t(n) \; \leq g(n).$$

That is, $t(n)$ becomes less than or equal to $g(n)$ once n becomes sufficiently large.

Example

Consider the function $t(n) = 5n + 70$. It is never less than n, so for sure $t(n)$ is not asymptotically bounded above by n. It is also never less than $5n$, so it is not asymptotically bounded above by $5n$ either. But $t(n) = 5n + 70$ is less than $6n$ for sufficiently large n, namely $n \geq 12$, so $t(n)$ is asymptotically bounded above by $6n$. The constant 6 in $6n$ is one of infinitely many that works here. Any constant greater than 5 would do. For example, $t(n)$ is also asymptotically bounded above by $g(n) = 5.00001n$, although n needs to be quite large before $5n + 70 \leq 5.00001n$.

The formal definition of big O below is slightly different. It allows us to define an asymptotic upper bound on $t(n)$ in terms of a *simpler* function $g(n)$, e.g. :

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

without having a constant coefficient. To do so, one puts the constant coefficient into the definition.

Definition (big O):

Let $t(n)$ and $g(n)$ be well-defined sequences of integers. We say $t(n)$ is $O(g(n))$ if there exist 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: this definition allows us to keep the $g(n)$ simple by not having a constant factor as part of the $g(n)$.

A few notes about this definition: First, the definition still is valid if $g(n)$ is a complicated function, with lots of terms and constants. But the whole point of the definition is that we keep $g(n)$ simple. So that is what we will do for the rest of the course. Second, the condition the $n \geq n_0$ is also important. It allows us to ignore how $t(n)$ compares with $g(n)$ when n is small. This is why we are talking about an *asymptotic* upper bound.

[10] Usually we are thinking of positive integers, but the defition is general. The $t(n)$ could be real numbers, positive or negative

Example 1

The function $t(n) = 5n + 70$ is $O(n)$. Here are a few different proofs:

First,

$$\begin{aligned} t(n) &= 5n + 70 \\ &\leq 5n + 70n, \text{ when } n \geq 1 \\ &= 75n \end{aligned}$$

and so $n_0 = 1$ and $c = 75$ satisfies the definition.

Here is a second proof:

$$\begin{aligned} t(n) &= 5n + 70 \\ &\leq 5n + 6n, \text{ for } n \geq 12 \\ &= 11n \end{aligned}$$

and so $n_0 = 12$ and $c = 11$ also satisfies the definition.

Here is a third proof:

$$\begin{aligned} t(n) &= 5n + 70 \\ &\leq 5n + n, \text{ for } n \geq 70 \\ &= 6n \end{aligned}$$

and so $n_0 = 70$ and $c = 6$ 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 or n_0 or both, and these constants with satisfy the definition also. So, don't think of the c and n_0 as being uniquely defined. Don't even think of them as the "best" c and n_0. The big O definition says nothing about "best".
- 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? No. If we were to change the definitions to be strict inequalities, then we just might have to increase the c or n slightly to make the definition work.
- We generally want our $g(n)$ to be simple. We also generally want it to be small. But the definition doesn't require this. For example, in the above example, $t(n) = 5n + 70$ is $O(n)$ but it is also $O(n \log n)$ and $O(n^2)$ and $O(n^3)$, etc. I will return to this in the next few lectures.

An example of an incorrect big O proof

Many of you are learning how to do proofs for the first time. It is important to distinguish a formal proof from a "back of the envelope" calculation. For a formal proof, you need to be clear on what you are trying to prove, what your assumptions are, and what are the logical steps that take you from your assumptions to your conclusions. Sometimes a proof requires a calculation, but there is more to the proof than calculating the "right answer".

For example, here is a typical example of an incorrect "proof" of the above:

$$\begin{aligned} 5n + 70 &\leq cn \\ 5n + 70n &\leq cn, \quad n \geq 1 \\ 75n &\leq cn \\ \text{Thus,} \quad c > 75, &\quad n_0 = 1 \quad \text{works.} \end{aligned}$$

This proof contains all the calculation elements of the previous proof. But this proof is wrong, since it isn't clear which statement implies which. The first inequality may be true or false, possibly depending on n and c. The second inequality is different than the first. It also may be true or false, depending on c. And which implies which? The reader will assume (by default) that the second inequality *follows from* the first. But does it? Or does the second inequality imply the first? Who knows? Not me. Such proofs tend to get grades of 0. This is not the big O that you want. Let's turn to another example.

Example 2

Claim: The function $t(n) = 8n^2 - 17n + 46$ is $O(n^2)$.

Proof: We need to show there exists positive c and n_0 such that, for all $n \geq n_0$,

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

$$\begin{aligned} t(n) &= 8n^2 - 17n + 46 \\ &\leq 8n^2 + 46n^2, \qquad \text{for } n \geq 1 \\ &= 54n^2 \end{aligned}$$

and so $n_0 = 1$ and $c = 54$ does the job.

Here is a second proof:

$$\begin{aligned} t(n) &= 8n^2 - 17n + 46 \\ &\leq 8n^2, \quad n \geq 3 \end{aligned}$$

and so $c = 8$ and $n_0 = 3$ does the job.

What does O(1) mean?

We sometimes say that a function $t(n)$ is $O(1)$. What does this mean? Applying the definition, it means that there exists constants c and n_0 such that, for all $n \geq n_0$, $t(n) \leq c$. That is, $t(n)$ is bounded above by a constant.

Some Big O Rules

Just like with limits in Calculus, there are some simple rules for proving that some functions are big O of other functions. Here are a few examples.

Constant Factors Rule

If $f(n)$ is $O(g(n))$ and $a > 0$ is a positive constant, then $af(n)$ is $O(g(n))$.

Proof: There exists a c such $f(n) \leq cg(n)$ for all $n \geq n_0$, and so $af(n) \leq acg(n)$ for all $n \geq n_0$. Thus we use ac and n_0 as constants to show $a\ f(n)$ is $O(g(n))$.

Sum Rule

If $f_1(n)$ is $O(g(n))$ and $f_2(n)$ is $O(g(n)$, then $f_1(n) + f_2(n)$ is $O(g(n))$.

Proof: There exists constants c_1, c_2, n_1, n_2 such $f_1(n) \leq c_1 g(n)$ for all $n \geq n_1$, and $f_2(n) \leq c_2 g(n)$ for all $n \geq n_2$. Thus, $f_1(n) + f_2(n) \leq c_1 g(n) + c_2 g(n)$ for all $n \geq max(n_1, n_2)$. So we can take $c_1 + c_2$ and $max(n_1, n_2)$ as our two constants.

Product Rule

If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n)$, then $f_1(n)f_2(n)$ is $O(g_1(n)\ g_2(n))$.

Proof: We can use similar constants as in the sum rule, except that now we have $g_1(n)$ and $g_2(n)$ instead of $g(n)$. Specifically, there exists constants c_1, c_2, n_1, n_2 such $f_1(n) \leq c_1 g_1(n)$ for all $n \geq n_1$, and $f_2(n) \leq c_2 g_2(n)$ for all $n \geq n_2$. Thus, $f_1(n) * f_2(n) \leq c_1 g_1(n) * c_2 g_2(n)$ for all $n \geq max(n_1, n_2)$. So we can take $c_1 * c_2$ and $max(n_1, n_2)$ as our two constants.

Transitivity Rule

If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.

Proof: There exists constants c_1, c_2, n_1, n_2 such $f(n) \leq c_1 g(n)$ for all $n \geq n_1$, and $g(n) \leq c_2 h(n)$ for all $n \geq n_2$. Plugging $g(n)$ from the second inequality into $g(n)$ in the first inequality gives that $f(n) \leq c_1 c_2 h(n)$ for all $n \geq max(n_1, n_2)$.

A related observation is that each of the following strict inequalities hold for sufficiently large n:

$$1 < \log_2 n < n < n \log_2 n < n^2 < n^3 < ... < 2^n < n! < n^n$$

Note that each successive pair of inequalities doesn't hold for all n. For example, $2^n < n!$ holds only for $n \geq 4$. Also note $n^3 < 2^n$ only holds for $n \geq 10$.

The point of the above rules is that it allows us to make immediate statements about the $O(\ )$ behavior of some rather complicated functions. For example, we can just look at the following function

$$t(n) = 5 + 8 \log_2 n + 16n + \frac{n(n-1)}{25}$$

and observe it is $O(n^2)$ by noting that the largest term is n^2. The rules above *justify* this observation.

Big O and sets of functions

Generally when we talk about $O()$ of some function $t(n)$, we use the "tightest" (smallest) upper bound we can. For example, if we observe that a function $f(n)$ is $O(\log_2 n)$, then generally we would not say that that $f(n)$ is $O(n)$, even though technically $f(n)$ *would* be $O(n)$, and it would also be $O(n^2)$, etc.

For a given simple $g(n)$ such as listed in the sequence of inequalities above, there are infinitely many functions $f(n)$ that are $O(g(n))$. So let's think about the set of functions that are $O(g(n))$. Up to now, we have been saying that some function $t(n)$ *is* $O(g(n))$. But sometimes we say that $t(n)$ *is a member of the set of functions that are* $O(g(n))$, or more simply $t(n)$ "belongs to" $O(g(n))$. In set notation, one writes "$t(n) \in O(g(n))$" where $\in$ is notation for set membership. With this notation in mind, and thinking of various $O(g(n))$ as sets of functions, the discussion in the paragraphs above implies that we have strict containment relations on these sets:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \ \cdots \subset O(2^n) \subset O(n!) \ldots$$

For example, any function $f(n)$ that is $O(1)$ must also be $O(\log_2 n)$, etc. I will occasionally use this set notation in the course and say "$t(n) \in O(g(n))$" instead of "$t(n)$ is $O(g(n))$".

Big Omega (asymptotic lower bound)

With big O, we defined an asymptotic upper bound. We said that one function grows at most as fast as another function. There is a similar definition for an asymptotic lower bound. Here we say that one function grows *at least* as fast as another function. The lower bound is called "big Omega". [ASIDE: In the slides, I led up to this definition by defining an asymptotic bound. Here I'll just cut to the chase give the definition of big Omega.]

Definition (big Omega): We say that $t(n)$ is $\Omega(g(n))$ 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.

Example

Claim: Let $t(n) = \frac{n(n-1)}{2}$. Then $t(n)$ is $\Omega(n^2)$.

To prove this claim, first note that $t(n)$ is less than $\frac{n^2}{2}$ for all n, so since we want a *lower* bound we need to choose a smaller c than $\frac{1}{2}$. Let's try something smaller, say $c = \frac{1}{4}$.

$$\begin{aligned} & \frac{n(n-1)}{2} && \geq && \frac{n^2}{4} \\ \iff \quad & 2n(n-1) && \geq && n^2 \\ \iff \quad & n^2 && \geq && 2n \\ \iff \quad & n && \geq && 2 \end{aligned}$$

Note that the "if and only if" symbols $\iff$ are crucial here. For any n, the first inequality is either true or false. We don't know which, until we check. But putting the $\iff$ in there, we are say that the inequalities in the different lines have the same truth value.

The last lines says $n \geq 2$, this means that the first inequality is true if and only if $n \geq 2$. Thus, we can use $c = \frac{1}{4}$ and $n_0 = 2$.

Are these the only constants we can use? No. Let's try $c = \frac{1}{3}$.

$$\begin{aligned} & \frac{n(n-1)}{2} && \geq && \frac{n^2}{3} \\ \iff \quad & \frac{3}{2}n(n-1) && \geq && n^2 \\ \iff \quad & \frac{1}{2}n^2 && \geq && \frac{3}{2}n \\ \iff \quad & n && \geq && 3 \end{aligned}$$

So, we can use $c = \frac{1}{3}$ and $n_0 = 3$.

Finally, a few notes:

- You should be able to easily show that the constant, sum, product, transitivity rules all hold for big Omega also.
- We can say that if $f(n)$ *is* $\Omega(g(n))$ then $f(n)$ is a member of the set of functions that are $\Omega(g(n))$. The set relationship is different from what we saw with $O()$, i.e. note that the set membership symbols are in the opposite direction:

  $$\Omega(1) \supset \Omega(\log n) \supset \Omega(n) \supset \Omega(n \log n) \supset \Omega(n^2) \ \cdots \supset \Omega(2^n) \supset \Omega(n!) \ldots$$

  For example, any positive function that is increasing with n will automatically $\Omega(1)$ but there are many increasing functions that are *not* bounded above by a constant i.e. $O(1)$.

Incorrect proofs

When one is first learning to write proofs, it is common to leave out certain important information. Let's look at a few examples of how this happens.

Claim:

For all $n \geq 1$, $2n^2 \leq (n+1)^2$.

If you are like me, you probably can't just look at that claim and evaluate whether it is true or false. You need to carefully reason about it. Here is the sort of incorrect "proof" you might be tempted to write, given the sort of manipulations I've been doing in the course:

$$\begin{aligned} 2n^2 & \leq (n+1)^2 \\ & \leq (n+n)^2, \text{ where } n \geq 1 \\ & \leq 4n^2 \end{aligned}$$

which is true, i.e. $2n^2 \leq 4n^2$. Therefore, you might conclude that the claim you started with is true.

Unfortunately, the claim is false. Take $n = 3$ and note the inequality fails since $2 \cdot 3^2 > 4^2$. The proof is therefore wrong. What went wrong is that the first line of the proof *assumes what we are trying to prove*. This is a remarkably common mistake.

Here is another example.

Claim:

For all $n \geq 4$, $2^n \leq n!$

This claim is true, and I asked students to prove it by induction on a midterm exam in a previous year. Many students gave "proofs" by verifying that the base case ($n = 4$) is true, and then trying to prove the induction step as follows:

$$\begin{aligned} 2^{k+1} &\leq (k+1)! \\ 2 \cdot 2^k &\leq (k+1) \cdot k! \\ 2 \cdot 2^k &\leq (k+1) \cdot 2^k \quad \text{(induction hypothesis)} \\ 2 &\leq k+1 \end{aligned}$$

and observing that the last line is indeed true for $k \geq 4$.

However, this proof is *incorrect*. It doesn't specify which statement implies which, or which statements are logically equivalent.

Here is proof of the induction step that is correct:

$$\begin{aligned} 2^{k+1} &= 2 \cdot 2^k \\ &\leq 2 \cdot k! \quad \text{(by induction hypothesis)} \\ &\leq (k+1) \cdot k! \quad \text{when } 2 \leq k+1, \text{ i.e. } k \geq 1 \\ &= (k+1)! \end{aligned}$$

Note that the induction step is proven for $k \geq 1$. But the base case is $n = 4$. So the claim only holds for $n \geq 4$, not $k \geq 1$. Indeed, the inequality of the claim is false for $k = 1, 2, 3$.

Big Theta

For all functions $t(n)$ that we deal with in this course, there will be a function $g(n)$ such 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 "big theta" of n, or $\Theta(g(n))$.

Definition (big theta): We say that $t(n)$ is $\Theta(g(n))$ if $t(n)$ is both $O(g(n))$ and $\Omega(g(n))$ for some $g(n)$. An equivalent definition is that there exists three positive constants n_0 and c_1 and c_2 such that, for all $n \geq n_0$,

$$c_1\ g(n)\ \leq\ t(n)\ \leq\ c_2\ g(n).$$

Obviously, we would need $c_1 \leq c_2$ for this to be possible.

It is possible to construct functions $t(n)$ for which there is no function $g(n)$ such that $t(n)$ is $\Theta(g(n))$. For example, consider a function that has a constant value, say 1, when n is even and has value n when n is odd. Such a function is $\Omega(1)$ and $O(n)$ but it is neither $\Theta(1)$ nor $\Theta(n)$. Obviously one can come up with many such functions. But these functions are contrived and they rarely come up in real computer science problems.

Best and worst case

The time it takes to run an algorithm depends on the size n of the input, but it also depends on the values of the input. If there are parameters passed to the algorithm then the time also depends on these parameters. For example, to remove an element from an arraylist takes constant time (fast) if one is removing the last element in the list, but it takes time proportion to the size of the list if one is removing the first element. Similarly, quicksort is very quick if one happens to choose good pivots in each call of the recursion but it is slow if one happens to choose poor pivots in each call of the recursion.

One often talks about *best case* and *worst case* for an algorithm. Here one is restricting the analysis of the the algorithm to a particular set of inputs in which the algorithm takes a minimum number of steps or a maximum number of steps, respectively. With this restriction, one writes the best or worst case as $t_{best}(n)$ or $t_{worst}(n)$ respectively.

The $t_{best}(n)$ or $t_{worst}(n)$ functions usually *each* can be characterized by some $\Theta(\ g(n)\ )$. However, the $g(n)$ may be different for $t_{best}(n)$ and $t_{worst}(n)$. I discussed two examples above, namely removing an element from an arraylist, and quicksort. These examples are in the first and the last row in the table below, and two other examples are in the 2nd and 3rd rows.

List Algorithms	$t_{best}(n)$	$t_{worst}(n)$
add, remove element (array list)	$\Theta(1)$	$\Theta(n)$
add, remove an element (doubly linked list)	$\Theta(1)$	$\Theta(n)$
insertion sort	$\Theta(n)$	$\Theta(n^2)$
selection sort	$\Theta(n^2)$	$\Theta(n^2)$
binary search (sorted array)	$\Theta(\log n)$	$\Theta(\log n)$
mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$
quick sort	$\Theta(n \log n)$	$\Theta(n^2)$

ASIDE: Binary search has the same best and worst case performance in the table below. If we were to change the binary search algorithm to check for an exact match at each call, then the best case performance would drop to $\Theta(1)$.

For a given algorithm, one often characterizes $t_{best}(n)$ using a $\Omega(\ )$ bound and one often characterizes $t_{worst}(n)$ using a $O(\ )$ bound. The reason is that when discussing the best case one often wants to express how good (lower bound) this best case is. Similarly when discussing the worst case one often wants to express how bad this worst case is. However, it also makes sense sometimes to characterize $t_{best}(n)$ using a $O(\ )$ bound and $t_{worst}(n)$ using a $\Omega(\ )$ bound. For example, if we say that the best case of removing an element from an arraylist takes time $O(1)$, we mean that it is *not worse than* constant time. Similarly, we might say the worst case of quicksort is $\Omega(n^2)$, we mean that the worst case takes *at least* that amount of time.

So, to briefly summarize the main points:

- $O(\ )$ is used for asymptotic upper bounds and $\Omega(\ )$ is used for asymptotic lower bounds.
- If we want to be mathematically correct, then we need to be talking about a specific function $t(n)$ when we discuss asymptotic bounds, whether $O(\ ), \Omega(\ )$ or $\Theta(\ )$.
- It is sometimes not possible to have one function $t(n)$ to describe the performance of an algorithm, since diffrent functions are needed for different inputs. In that case, we often consider $t_{best}(n)$ or $t_{worst}(n)$.
- Typically $t_{best}(n)$ and $t_{worst}(n)$ for some algorithm each can be expressed as $\Theta(\ g(n)\ )$ for some $g(n)$.

That said, often people want to say that *some algorithm* (rather than some $t(n)$ has certain asymptotic performance. For example if the best and worst cases of the algorithm take time $t_{best}(n)$ and t_{worst} and these functions are of the same $\Theta(\ )$ set, then one can say that the algorithm has that $\Theta(\ )$ behavior. If, however, the $t_{best}(n)$ bound is strictly less than the $t_{worst}(n)$ bound, then one often says that the algorithm is $\Omega(g_1(n))$ and $O(g_2(n))$ where $t_{best}(n)$ is $\Omega(g_1(n))$ and $t_{worst}(n)$ is $\Omega(g_2(n))$. This is loose talk, but it is common.

Using "limits" for asymptotic complexity

The formal definitions of big O, Omega, and Theta require that we give specific constants n_0 and c. But in practice for a given $t(n)$ and $g(n)$ there many possible choices of constants and typically we don't care what the constants are. We only care that they exist. This is similar to the formal definition of a limit of a sequence from Calculus, which I mentioned at the beginning of lecture 16. In your Calculus courses, you are asked whether a limit exists and what the limit is, but you are not asked for formal proofs. Instead you are given certain rules that you follow for determing the limits (l'Hopitals rule, etc).

Below I will state several rules that use limits for showing that one function is big O, big Omega, or big Theta of another function. In some cases, I will use the formal definition of a limit, so I will repeat it here (recall lecture 16).

A sequence $t(n)$ of real numbers converges to (or has a limit of) a real number t_∞ if, for any $\epsilon > 0$, there exists an n_0 such that, for all $n \geq n_0$, $\mid t(n) - t_\infty \mid < \epsilon$.

Let's now write the definition of big O slightly differently than before, namely we consider the ratio of $t(n)$ to $g(n)$. The definition now is that $t(n)$ is $O(g(n))$ if there exist two positive numbers n_0 and c such that, for all $n \geq n_0$,

$$\frac{t(n)}{g(n)} \leq c \, .$$

So, suppose that $t(n)$ and $g(n)$ are two sequences and that $lim_{n\to\infty}\frac{t(n)}{g(n)} = 0$. What can we say about their big relationship ? From the formal definition of a limit, we can say that for any $c > 0$, there exists an $n_0 > 0$ such that, for all $n \geq n_0$,

$$| \frac{t(n)}{g(n)} - 0 | < c$$

and in particular

$$\frac{t(n)}{g(n)} < c \, .$$

Since this is true for any $c > 0$, it is obviously true for some $c > 0$, and so it follows immediately that $t(n)$ is $O(g(n))$. This observation gives us our first rule.

Limit rule for big O:

If $\lim_{n\to\infty} \frac{t(n)}{g(n)} = 0$, then $t(n)$ is $O(g(n))$.

Note that this rule is not equivalent to the definition of big O, since the implication (if then) does not work in the opposite direction, that is, if $t(n)$ is $O(g(n))$ then we cannot conclude that $\lim_{n\to\infty} \frac{t(n)}{g(n)} = 0$. A simple failure of this type is the case that $t(n) = g(n)$ since $t(n)$ would be $O(g(n))$ but $\frac{t(n)}{g(n)} = 1$ for all n and so the limit would not be 0.

Also note that the above rule for big O is weak in the sense we *only* conclude that $t(n)$ is $O(g(n))$. For example, the function $t(n) = 5n + 6$ is $O(n^2)$, but this is not so interesting since in fact $t(n)$ has a tighter big O bound, namely $O(n)$.

Here is a stronger statement than the above limit rule for the case that the limit is 0.

Limit rule for big O (stronger):

If $\lim_{n\to\infty} \frac{t(n)}{g(n)} = 0$, then $t(n)$ is $O(g(n))$ but $t(n)$ is not $\Omega(g(n))$.

To prove this rule, we just rewrite the definition for big Omega in terms of the ratio $\frac{t(n)}{g(n)}$ as we did with big O, namely $t(n)$ is $\Omega(g(n))$ if there exist two positive numbers n_0 and c such that, for all $n \geq n_0$,

$$\frac{t(n)}{g(n)} \geq c \, .$$

Note that the definition requires that $c > 0$ since if we were to allow $c = 0$, then the definition would hold for *any* positive sequence $g(n)$, which wouldn't capture the idea of that $g(n)$ is an asymptotic lower bound on $t(n)$.

So how do we prove the stronger rule for big O ? We simply observe that it is impossible for *both* the limit of $\frac{t(n)}{g(n)}$ to be 0 *and* also for $\frac{t(n)}{g(n)} \geq\ c$ for all $n \geq n_0$, where $c > 0$ is a fixed constant.

We can prove analogous rules for the lower bound big Omega.

Limit rule for big Omega (Ω):

If $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \infty$, then $t(n)$ is $\Omega(g(n))$.

Saying that a limit of a sequence is infinity just means that the sequence not bounded above, namely that you cannot find constants c and n_0 such the sequence has values less than c for all $n \geq n_0$. No matter how big you make c and n_0, you can always find some $n \geq n_0$ such that the sequence takes a value greater than c for that n. Note that the sequence in question here is $\frac{t(n)}{g(n)}$.

Also note that this rule is not equivalent to the definition of big Omega, and the same counterexample above holds, namely $t(n) = g(n)$. In this case, $t(n)$ is $\Omega(g(n))$ but $\frac{t(n)}{g(n)} = 1$ is obviously bounded.

Finally, as in the big O limit rule, here we can also show that a stronger version of the big Omega rule holds.

Limit rule for big Omega (Ω): (stronger)

If $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \infty$, then $t(n)$ is $\Omega(g(n))$ but then $t(n)$ is not $O(g(n))$.

To prove this rule, we use the same argument as above. Suppose $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \infty$. In order for $t(n)$ to be $O(g(n))$ we would need a $c > 0$ and n_0 such that

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

But this can't happen since we have asumed the limit is infinity. Thus, $t(n)$ is not $O(g(n))$.

Limit rule for big Theta Θ:

If $\lim_{n\to\infty} \frac{t(n)}{g(n)} = c$ and c is strictly greater than 0 and finite, then $t(n)$ is $\Theta(g(n))$.

To formally prove this rule, we need to use the definition of a limit. Suppose $\lim_{n\to\infty} \frac{t(n)}{g(n)} = c$. Than, by definition, for any $\epsilon > 0$, there exists an n_0 such that, for all $n \geq n_0$,

$$\mid \frac{t(n)}{g(n)} - c \mid < \epsilon$$

or equivalently

$$-\epsilon <\ \frac{t(n)}{g(n)} - c\ < \epsilon$$

Immediately we see that, for all $n \geq n_0$, we have the big O bound,

$$\frac{t(n)}{g(n)} < c + \epsilon$$

and the big Omega bound

$$c - \epsilon < \frac{t(n)}{g(n)}.$$

where our constants are $c + \epsilon$ and $c - \epsilon$ respectively. Note that we can ensure that $c - \epsilon > 0$ since c is fixed $c > 0$ and we are free to choose any ϵ, so we just choose it to be any number between 0 and c. And we're done.

This rule is not equivalent to the definition of big Theta, since the implication does not work in the opposite direction. In particular, it can happen that $t(n)$ is $\Theta(g(n))$ but the limit does not exist and so the rule does not apply. For example, take the contrived example that

$$\frac{t(n)}{g(n)} = a(n)$$

where $a(n)$ is the sequence $1, 2, 1, 2, 1, 2 \ldots$. In this case, $lim_{n\to} \frac{t(n)}{g(n)}$ does not exist. Yet $t(n)$ is still $\Theta(g(n))$ in this case. That is, $t(n)$ is $O(g(n))$ since the constant $c = 2$ works for all n, and $t(n)$ is $\Omega(g(n))$ since the constant $c = 1$ works for all n.

"What do I need to know for the final exam?"

The answer to this question depends on what your background is, how important grades are to you, and how far you want to go in a Computer Science program. I'm not going to build up a table of possibilities. But if you have a good math background (meaning at least B's in your 100 level math courses) then you should be able to understand the main ideas above, and if grades are important to you then you will need to understand these arguments to answer the questions on this stuff (lectures 16-18) on the final exam, and if you want to continue to COMP 251 and COMP 330 and possibly other theoretical courses, then yes you need to know all of it (or risk not understanding things at the next level).

At the other extreme, if you only are taking COMP 250 to learn enough to clear the bar on job interviews for software positions, then I suggest you spend less time on what we covered this week and more time on the algorithms and data structures parts of the course. From what I've, job interviewers rarely ask big O questions, and if they do, it is only at a high level. Correct me if I'm wrong, after you do interviews, ok?

(Rooted) Trees

Thus far we have been working mostly with "linear" collections, namely *lists*. For each element in a list, it makes sense to talk about the previous element (if it exists) and the next element (if it exists). It is often useful to organizes a collection of elements 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. The key difference here 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, see the McGill organizational chart. `http://www.mcgill.ca/orgchart/` which is (almost) a tree. Note that the "lowest" level in this chart contains the Deans but of course there are thousands of employees at McGill "below" the Deans, which are not shown. For example, as a McGill professor in the School of Computer Science, I report to my department Chair (Professor Bettina Kemme), who reports to the Dean of Science, who reports to the McGill Provost, who 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. The first defines the parent/child relation literally, so that the tree children of a node correspond to the actual children (sons and daughters) of the person represented by the node. The second tree is less conventional. It defines each person's mother and father as its "children". In such a tree, each person has two children (the person's real parents), and they each of two children (the person's four grandparents), etc.

- A file directory on a MS Windows or UNIX/LINUX operating system. For example, this lecture notes file is stored at

  `C:\Users\Michael\Dropbox\TEACHING\250\LECTURENOTES\17-trees.pdf`

[ASIDE: The lecture slides used many figures to illustrate the ideas that I will be presenting in today's notes. I am not including the figures in these notes. But you should definitely consult those slides when reading these notes, or you will have a tough time!]

Terminology for rooted trees

A (rooted) tree consists of a set of *nodes* or *vertices*, and *edges* which are ordered pairs of nodes or vertices. When we write an edge (v_1, v_2), we mean that the edge goes from node v_1 to v_2. The "node" vs. "vertex" terminology is perhaps a bit confusing at first. When one is discussing data

structures, it is more common to use the term "node" and when one is talking about trees as abstract data types one typically uses the term "vertex."

In COMP 250 we will only deal with a special type of tree called a *rooted* tree. A rooted tree has special node called the *root node*. Rooted trees have the following properties:

- For any node v in the tree (except the root node), there is a unique node p such that (p, v) is an edge in the tree. The node p is called the *parent* of v, . Naturally, v is called a *child* of p. A parent can have multiple children.

  Notice that a tree with n nodes has $n - 1$ edges. Why? Because for each node v except the root node ($n - 1$ of these), there is a unique edge (p, v) and these $n - 1$ edges are exactly the set of edges in the tree.

- *sibling relation*: Two nodes are siblings if they have the same parent.

- *leaf*: A node with no children is called a leaf node. A more complicated way of saying a "node with no children" is "a node v such that there does *not* exist an edge (v, w) where w is a 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 linux 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 may be 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 in a tree (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.

- *height* of a node v in a tree: 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

- *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".

- A subtree is a subset of nodes and edges in a tree which itself is a tree. In particular, a subtree has its own root which may or may not be the same as the root of the original tree. Every node in a tree defines a subtree, namely the tree defined by this node and all its children. In particular, any tree is a subtree of itself. Every node in a tree also defines a subtree in which that node is the only node.

- *recursive definition of a rooted tree:* A rooted tree T either has no nodes (empty tree), or it consists of a root node r together with a set of zero or more non-empty subtrees $T_1, \ldots, T_k$ whose roots are the children of r.

We can also define operations on trees recursively. Here are a few common examples. The first is to compute the depth of a node.

```
depth(v){
  if (v is a root node)    //  that is,  v.parent == null
    return 0
  else
    return 1 + depth(v.parent)
}
```

Notice that this method requires that we can access the parent of a node. We have defined edges to be of the form (parent, child). If one implements an edge by putting a child reference in a parent node then, given a node v, we can only reference the child of this node, not the parent, and so we would not be able to compute the depth. Therefore, to use the above method for computing depth, we would need a node to have a reference to its parent. (Whether we need references to children too depends on what we will use the tree for. In most figures in the slides today, I drew trees that had references only from parents to children.)

Another example operation is to compute the height of a node. Just like the depth, the height can be computed recursively.

```
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 implementation in Java

The main decision one needs to make in implementing trees is how to represent the set of children of a node. If node can have at most 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 a node can have many children, then a more flexible approach is needed.

One common approach is to define the children by an array list or by a linked list.

```
class  TreeNode<T>{
    T   element;
    ArrayList<TreeNode<T>>   children;

    TreeNode<T>   parent;   // optional
     :
     :                     //  methods
}
```

This approach is perhaps overkill though, since one typicallly doesn't need to index the children by their number. Using a `LinkedList` (doubly linked list in Java) is also perhaps overkill if one doesn't need to be able to go both directions in the list of children. A singly linked list is often enough, and that is the approach below.

The *"first child, next sibling"* implementation of a tree uses the following node definition.

```
  class  TreeNode<T>{
      T   element;
      TreeNode<T>   firstChild;
      TreeNode<T>   nextSibling;
        :
        :                      //  methods
  }
```

It defines a singly linked list for the siblings, where the head is the `firstChild` and the next is the `nextSibling`. It is simpler than the arraylist implementation since it uses just two fixed references at each node (or 3, if one also has a parent link)

Finally, 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. The `TreeNode` class would then be defined as an inner class inside this `Tree` class.

Exercise: Representing trees using lists

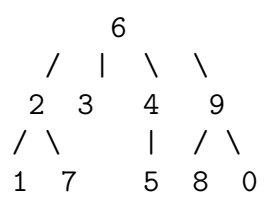
A tree can be represented using lists, as follows:

```
tree             =   root | ( root listOfSubTrees )
listOfSubTrees   =   tree |   tree listOfSubTrees
```

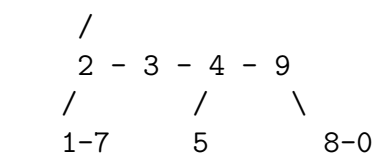
For example, let's draw the tree that corresponds to the following list, where the `root` elements are single digits. (Apologies for the ASCII art below. See the slides for prettier pictures.)

```
( 6 ( 2 1 7 ) 3 ( 4 5 ) ( 9 8 0 ) )
```

The first uses a separate edge for each parent/child pair.



The second uses the "first child, next sibling" representation.



Tree traversal

Often we wish to iterate through or "traverse" all the nodes of the tree. We generally use the term *tree traversal* for this. 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 may need to do something at each node. I will use the term "visit" for the latter. Visiting a node means doing some computation at that node. We will see some examples later.

Depth first traversal using recursion: pre- and post-order

The first two traversals that we consider 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.

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

See the lectures slides for an example of a preorder traversal and a number of the ordering of nodes visited.

As a second example, suppose we have a file system. The directories and files define a tree whose internal nodes are directories and whose leaves are either empty directories or 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 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)
      :                         (file)
   Work                         (directory)
      COMP250                   (directory)
          :
```

For a postorder traversal, one visits a node after having visited all the children of the node.

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

Let's look at an example. Suppose we want to calculate how many bytes are stored in all the files within some directory including all its sub-directories. This is post-order because in order to know the total bytes in some directory we first need to know the total number of bytes in all the subdirectories. Hence, we need to visit the subdirectories first. Here is an algorithm for computing the number of bytes. 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. (Frankly, in this example it is a big vague what I mean by "visit", since computing `sum` doesn't just happen after visiting the children but rather involves steps that occur before, during, and after visiting the children. The way I think of this is that if we were to store `sum` as a field in the node, then we could only do this after visiting the children.)

```
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 already in this course, recursive algorithms are implemented using a call stack which keep track of information needed in each call. (Recall the stack lecture.) You can sometimes avoid recursion by using an explicit stack instead. Here is an algorithm for doing a depth first traversal which uses a stack rather than recursion. As you can see by running an example (see lecture slides), this algorithm visits the list of children of a node in the opposite order to that defined by the `for` loop.

Is this algorithm preorder or postorder? The `visit cur` statement occurs prior to the `for` loop. You might think this automatically makes it preorder. However, the situation is more subtle than that. If we were to move the `visit cur` statement to be after the `for` loop, the order of the visits of the nodes would be the same. Moreover, nodes would still be visited before their children. So, this would still be a preorder traversal.

```
treeTraversalUsingStack(root){
  s.push(root)
  while !s.isEmpty(){
    cur = s.pop()
    visit cur
    for each child of cur
       s.push(child)
    //  'visit cur' could be put here instead
  }
}
```

Breadth first traversal

What happens if we use a queue instead of a stack in the previous algorithm?

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

As shown in the example in the lecture, this algorithm visits all the nodes at each depth, before proceeding to the next depth. This is called *breadth first* traversal. The queue-based algorithm effectively does the following:

```
for i = 0 to height
    visit all nodes at level i
```

You should work through the example in the slides to make sure you understand why using queue here is diffrent from using a stack.

A note about implementation

Recall first-child/next-sibling data structure for representing a tree, which we saw last lecture. Using this implementation, you can replace the line

```
for each child of cur
   ...
```

with the following. Here we are iterating through a (singly linked) list of children.

```
child = child.firstChild
while (child != null){
   ...                          //  maybe do something at that child
  child = child.nextSibling
}
```

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*.

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.

How many nodes can a binary tree have at each level? The root has one node. Level 1 can have two nodes (the two children of the root). Level 2 can have 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 if node has at most one child and there is one leaf, which by definition has no children. It follows that

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

ASIDE: We can rearrange this equation to give

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

Binary tree nodes: Java

We have seen the first-child/next-sibling data structure for general trees. For binary trees, one typically uses the following data structure instead:

```
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. Recall that 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:

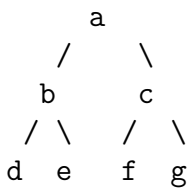
```
preorderBT(root){
    if (root is not null){              // base case
       visit root
       preorderBT(root.left)
       preorderBT(root.right)
    }
}

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

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

```
inorderBT(root){
    if (root is not null){              // base case
       inorderBT(root.left)
       visit root
       inorderBT(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 (different from one given in slides)




```
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  =  variable | integer
operator        =  + | - | * | / | ^
expression      =  baseExpression |  expression operator expression
```

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.

Expression trees

[In the slides, I used slightly different examples from what I use below.]

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:

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, although as I mentioned in the lecture, my MS Windows calculator ignores the precedence ordering of * over +. There is also a convention that `6 ^ z ^ 8` means `6 ^ (z ^ 8)` rather than `(6 ^ z) ^ 8`. See the example that I gave in the slides.

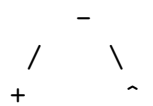
The above precedence order implies that an expression such as

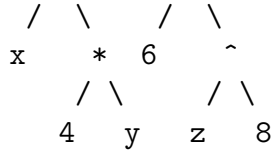
$$\texttt{x} + \texttt{4} * \texttt{y} - \texttt{6} \hat{} \texttt{z} \hat{} \texttt{8}$$

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:





You may be tempted to think that the + operator should be in the root rather than the - operator being in the root because the + operator is to the left. In fact, it works the other way: higher precedence means it is evaluated first, which means it is deeper in the 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
prefixExpression  =  baseExpression |  operator prefixExpression prefixExpression
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" or RPN. Many calculators, in particular, Hewlett-Packard calculators require that users enter expressions using RPN. Apparently one learns very quickly how to do it.]

For computer science, the advantage of postfix (and prefix) expressions over infix expressions is that with postfix expressions you do not need a precedence rule to define the order of operations. In particular, below is a simple stack-based algorithm for evaluating a postfix expression. The algorithm does not need to know about precedence orderings, since these are "built in" to the postfix expression. See the slides for an example of how the stack evolves over time for a particular expression.

```
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
     operand2 = s.pop()    // opposite order to push
     operand1 = s.pop()    //   "
     operator = cur.element
     s.push( evaluate( operand1 operator operand2 ) )
   }
   cur = cur.next
}
```

Binary Search Trees

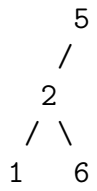
Today we consider a specific type of binary tree in which there happens to be an ordering defined on the set of elements the nodes. If the elements are numbers then there is obviously an ordering. If the elements are strings, then there is also a natural ordering, namely the dictionary ordering, also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

Definition: binary search tree

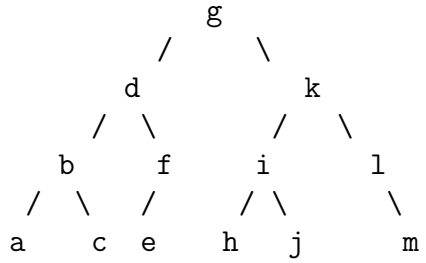
A *binary search tree* is a binary tree such that

- each node contains an element, called a *key*, such that the keys are comparable, namely there is a strict ordering relation < between keys of different nodes
- any two nodes have different keys (i.e. no repeats, i.e. duplicates)
- 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.

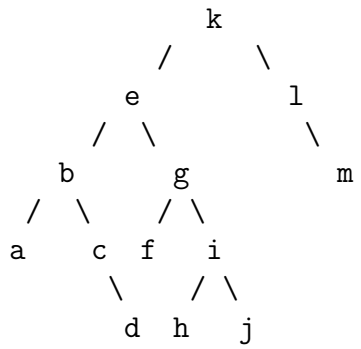
  Note this is stronger than just saying that the left child's key is less than the node's key which is less than the right child's key. For example, the following is not a binary search tree.



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.



Here is another example with the same set of keys:



BST as an abstract data type (ADT)

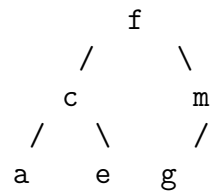
One performs several common operations on binary search trees:

- `find(key)`: given a key, return a reference to the node containing that key (or null if key is not in tree)
- `findMin()` or `findMax()`: 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)

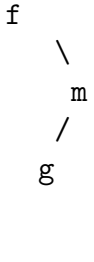
Here are algorithms for implementing these operations.

```
find(root,key){                  //    returns a node
  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)
}

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

For example, here the minimum key is a.



Notice however 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. Here the minimum key is c:



The reasoning and method is similar for findMax.

```
findMax(root){                  // returns a node
   if (root == null)
      return null
   else if (root.right == null))
      return root
   else
      return findMax(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.

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

The new node is always added at a leaf and so the base case is always reached eventually. For the non-base case, the situation is subtle. The code says that a new node is added to either the left or right subtree and then the reference to the left or right subtree is re-assigned. It is reassigned

to the root of the left or right subtree, which has the new node added it. Why is it necessary to reassign the reference like that?

[ASIDE: I had forgotten to review this detail before the Sec. 001 class (lecture recording) and I promised I would fill it into the lecture notes: here we are!]
Suppose we add the new node to the left subtree. If the new node is a descendent of the root node of the left subtree , then the assignment `root.left = add(root.left,key)` doesn't change the reference `root.left` in the `root` node; it just assigns it to the same node it was referencing beforehand. *However,* if `root.left` was null and then the new node was added to the left subtree of `root`, then the call `add(root.left,key)` will create and return the new node, namely it is the root of a subtree with one node. If we don't assign that node to `root.left` as the code says, then the new node that is created will not in fact be added to the tree.

Next, consider the problem of removing a node from a binary search tree.

```
remove(root, key){                  // returns root
    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.

The more challenging case is that the key that we want to remove is at the root (perhaps after a sequence of recursive calls). In this case we need to consider four possibilities:

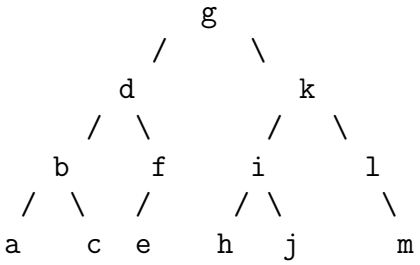
- the root has no left child
- the root has no right child
- the root has no children at all
- the root has both a left child and a right child

In the first two cases – see (**) in the algorithm – we just replace the root node with the subtree in the non-empty child. Note that the third case is accounted for here as well; since both children are null, the root will become null.

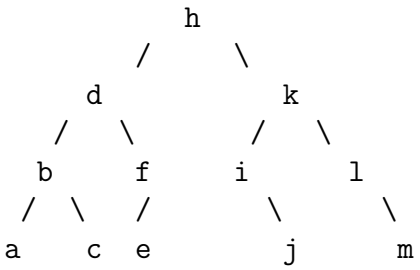
In the fourth case – (***) – we take the following approach. We replace the root with the minimum node in the right subtree. It does so in two steps. It copies the key from the smallest node in the right subtree into the root node, and then it removes the smallest node node from the the right subtree.

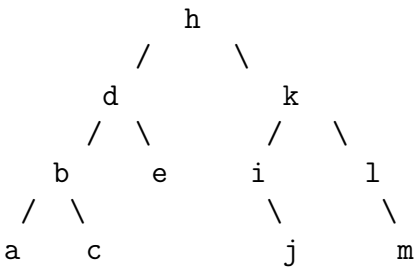
Example (remove)

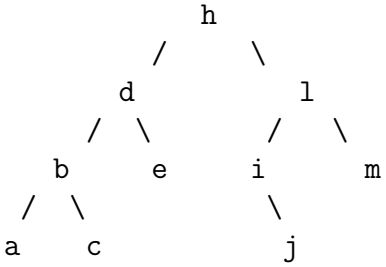
Take the following example with nodes `abcdefghijklm`:



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







Let's consider the best and worst case performance. The best case performance for this data structure tends to occur when the keys are arranged such that the tree is balanced, so that all leaves are at the same depth (or depths of two leaves differ by at most one). However, if one adds keys into the binary tree and doesn't rearrange the tree to keep it balanced, then there is no guarentee that the tree will be anywhere close to balanced, and in the worst case a BST with n nodes could have height $n-1$. This implies the following best and worst cases:

Operations/Algorithms for Lists	Best case, $t_{best}(n)$	Worst case, $t_{worst}(n)$
findMax(), findMin()	$\Theta(1)$	$\Theta(n)$
find(key)	$\Theta(1)$	$\Theta(n)$
add(key)	$\Theta(1)$	$\Theta(n)$
remove(key)	$\Theta(1)$	$\Theta(n)$

In COMP 251, you will learn about balanced binary search trees, e.g. AVL trees or red-black trees. If a tree is balanced, then the operations are all $\Theta(\log n)$. This is an interesting topic, but technically involved so I won't say anything more about it here.

Priority Queue

Recall the definition of a queue. It is a collection where we remove the element that has been in the collection for the longest time. Alternatively stated, we remove the element that first entered the collection. 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 may also or instead be determined by the urgency of the case. 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. Once a comparison method is chosen for determining priority, *the next element to be removed is the one with greatest priority.* Heads up: with priority queues, one typically assigns low numerical values to high priorities. Think "my number one priority", "my number 2 priority", etc.

One way to implement a priority queue of elements (often called *keys*) is to maintain a sorted list. This could be done with a linked list or array list. Each time a element is added, it would need to be inserted into the sorted list. If the number of elements were huge, however, then this would be an inefficient representation since in the worst case the adds and removes would be $\Theta(n)$. [ASIDE: in future, you will find people saying $O(n)$ to express what I wrote, loosely identifying 'worst case' with $O(\ )$.]

A second way to implement a priority queue would be to use a binary search tree. The element that is removed next is found by the `findMinimum()` operation. This would be a better way to implement a priority queue than the linear list method, since add and remove tend to take $\log n$ steps rather than n steps, if the tree is balanced. (I mentioned *balanced* binary search trees briefly in the last lecture. This topic will be covered in more depth – pun intended – in COMP 251.) One problem with using a balanced binary search tree for a priority queue is that it is overkill. In the next few lectures, we will look at a simpler data structure, called a *heap*.

Heaps

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 and satisfy the property that *each node is less than its children*. (To be precise, when I say that the nodes are comparable, I mean that the *elements* are comparable. And by that I mean that there is an ordering on all the elements.[11]) 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 a min heap in the next few lectures. Note that it follows from the definition that the smallest element in a heap is stored at the root.

[11] In Section 002, one student asked about the rock-paper-scissors problem where elements are pairwise ordered but there is no global ordering. That is NOT what is happening here. Here we have a global ordering. I will talk about what I mean by `Comparable` later in the course.

As with stacks and queues, the two main operations we perform on heaps are `add` and `remove`.

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 leaf. 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 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 element 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. This process of moving a node up the heap, is often called "upheap".

```
add(element){
   cur = new node at next leaf position
   cur.element = element
   while (cur != root) && (cur.element < cur.parent.element){
       swapElement(cur, parent)
       cur = cur.parent
   }
}
```

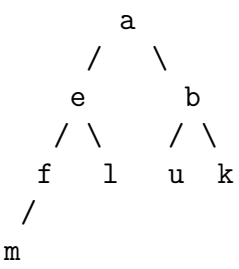
You might ask whether swapping the element at a node with its parent's element can cause a problem with the node's sibling (if it exists). It is easy to see that no problem exists though. Before the swap, the parent is less than the sibling.[12] So if the current node is less than its parent, then the current node must be less than the sibling. So, swapping the node's element with its parent's element preserves the heap property with respect to the node's current sibling.

For example, suppose we have a heap with two elements `e` and `g`. Then we add an element to the `*` position below and we find that `* < e`. So we swap them. But if `* < e` then `* < g`.

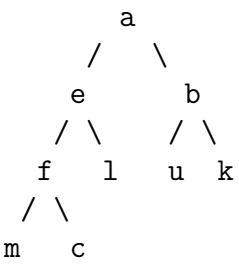
```
  e
/   \
g    *
```

[12] Here I say that one node is less than another, but what I really mean is that the element at one node is less than the element at the other node.

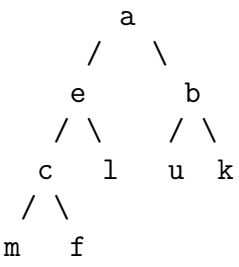
Here is a bigger example. Suppose we add element `c` to the following heap.



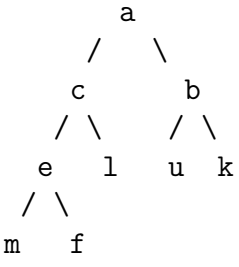
We add a node which is a sibling to `m` and assign `c` as the element of the new node.



Then we observe that `c` is less than the element `f` of its parent, so we swap `c,f` to get:



Now we continue up the tree. We compare `c` with the element in its new parent `e`, see that the elements need to be swapped, and swap them to get:



Again we compare `c` to its parent. Since `c` 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.

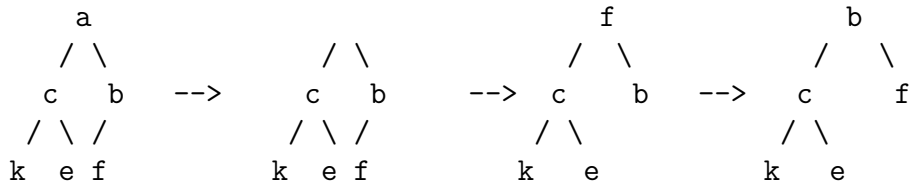
We start at the root, which now contains an element that was previously the rightmost leaf in level h. We compare the root to its two children. If the root is greater than at least one of the children, we swap the root with the smaller child. Moving the smaller child to the root does not create a problem with the other child and with the heap property, since by definition the smaller child is greater than the larger child.

Here is a sketch of the algorithm:

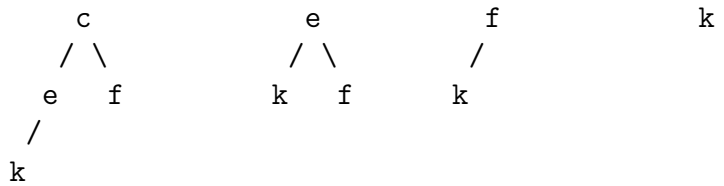
```
removeMin(){                // returns smallest element
   tmp = root.element
   remove last leaf node and put its element into the root
   cur = root
   while ((cur has a left child) and
          ((cur.element > cur.left.element) or
           (cur has a right child and cur.element > cur.right.element)))
       minChild = child with the smaller element
       swapElement(cur, minChild)
       cur = minChild
   }
   return tmp
}
```

The condition in the while loop is rather complicated, and you may have just skipped it. Don't. There are several possible events that can happen and you need to consider each of them. One is that the current node has no children. In that case, there is nothing to do. The second is that the current node has one child, in which case it is the left child. The potential problem here is that the left child might be smaller. The third is that the current node might have two children and one of these two children *has to be* smaller. (Are we guarenteed that one of the two children is smaller? See Exercises.)

Here is an example:

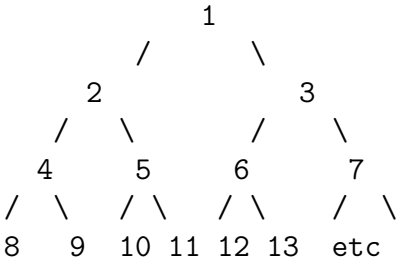


If we apply `removeMin()` again and again until all the elements are gone, we get the following sequence of heaps with elements removed in the following order: `b, c, e, f, k`.



Implementing a heap using an array

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.

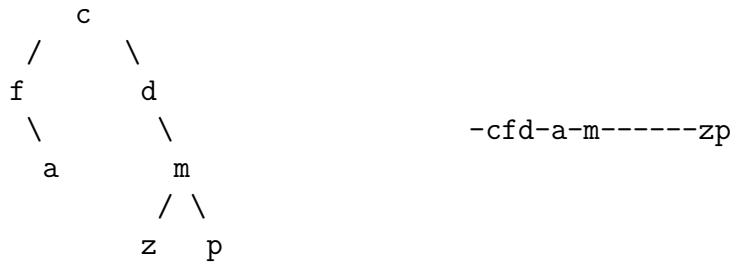


These numbers are NOT the elements stored at the node, rather we are just numbering the nodes so we can index them. I will discuss the more next lecture.

At the end of last lecture, I showed how to represent a heap using an array. The idea is that an array representation defines a simple relationship between a tree 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 $i/2$.

ASIDE: General binary trees can be represented as arrays

You can always use an array to represent a binary tree if you want, by using the above indexing scheme. However, there are costs to doing so when the binary tree is NOT a complete binary tree, namely there maybe large gaps in the array. We would just have null references at these gap points. For example, below is a binary tree (left) and its array representation (right) where - in the array indicates null. Note that there is a `null` in the 0-th element. It is only there to make the parent-child indexing formula simpler to think about. If you were really implementing this array and the various methods, you could use the 0-th slot and adjust the parent-child index relationship appropriately.



Let's next revisit the `add` and `removeMin` methods and rewrite them in terms of this array.

add(element)

Suppose we have a heap with `k` elements which is represented using an array, and now we want to add a `k+1`-th element. Last lecture we sketched an algorithm doing so. Here I'll re-write that algorithm using the simple array indexing scheme. Let `size` be the number of elements in the heap. These elements are stored in array slots 1 to `size`, i.e. recall that slot 0 is unused so that we can use the simple relationship between a child and parent index.

```
add(element ){
   size = size + 1          // number of elements in heap
   heap[ size ] = element   // assuming array has room for another element
   i = size

   //  the following is sometimes called "upHeap"  -- see below

   while (i > 1  and heap[i] < heap[ i/2 ]){
     swapElements( i, i/2 )
     i = i/2
   }
}
```

Here is an example. Let's add `c` to a heap with 8 elements.

```
1   2   3   4   5   6   7   8   9
-----------------------------------
a   e   b   f   l   u   k   m   c
a   e   b   c   l   u   k   m   f  <----  c swapped with f (slots 9 & 4)
a   c   b   e   l   u   k   m   f  <----  c swapped with e (slots 4 & 2)
```

Building a heap

We can use the `add` method to build a heap as follows. Suppose we have a list of `size` elements and we want to build a heap.

```
buildHeap(list){
   create new heap array         //  size == 0,  length > list.size
   for (k = 0; k < list.size; k++)
        add( list[k] )                    //  add to heap[ ]
}
```

We can write this in a slightly different way.

```
buildHeap(list){
   create new heap array      //  size == 0,  length > list.size
   for (k = 1; k <= list.size; k++){
       heap[k] = list[k-1]       //  say list index is 0, .. ,size-1
       upHeap(k)
   }
}
```

where `upHeap(k)` is defined as follows.

```
upHeap(k){    //  assumes we have an underlying array structure
    i = k
    while (i > 1  and heap[i] < heap[ i/2 ]){
       swapElements( i, i/2 )
       i = i/2
    }
}
```

Are we sure that the buildHeap() method indeed builds a heap? Yes, and the argument is basic mathematical induction. Adding the first element gives a heap with one element. If adding the first k elements results in a heap, then adding the $k+1$-th element also results in a heap since the upHeap method ensures this is so.

Time complexity

How long does it take to build a heap in the best and worst case? Before answering this, let's recall some notation. We have seen the "floor" operation a few times. Recall that it rounds down to the nearest integer. If the argument is already an integer then it does nothing. We also can define the ceiling, which rounds up. It is common to use the following notation:

- $\lfloor x \rfloor$ 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.

Let i be the index in the array representation of elements/nodes in a heap, then i is found at level *level* in the corresponding binary tree representation. The level of the corresponding node i in the tree is such that

$$2^{level} \leq i < 2^{level+1}$$

or

$$level \leq \log_2 i < level + 1,$$

and so

$$level = \lfloor \log_2 i \rfloor.$$

We can use this to examine the best and worst cases for building a heap.

In the best case, the node i that we add to the heap satisfies the heap property immediately, and no swapping with parents is necessary. In this case, building a heap takes time proportional to the number of nodes n. So, best case is $\Theta(n)$.

What about the worst case? Since $level = \lfloor \log_2 i \rfloor$, when we add element i to the heap, in the *worst case* we need to do $\lfloor \log_2 i \rfloor$ swaps up the tree to bring 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. If we are adding n nodes in total, the worst case number of swaps is:

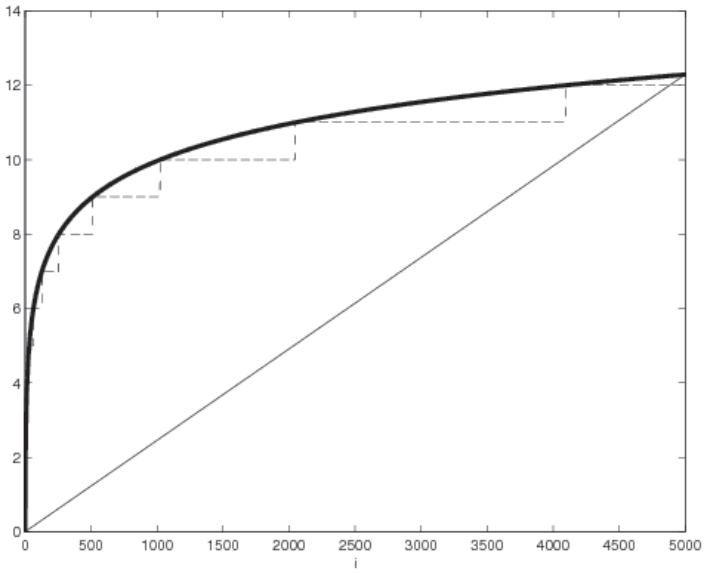
$$t(n) = \sum_{i=1}^{n} \lfloor \log_2 i \rfloor$$

To visualize this sum, consider the plot below which show the functions $\log_2 i$ (thick) and $\lfloor \log_2 i \rfloor$ (dashed) curves up to $i = 5000$. In this figure, $n = 5000$.

The area under the dashed curve is the above summation. It should be visually obvious from the figures that

$$\frac{1}{2} n \log_2 n < t(n) < n \log_2 n$$

where the left side of the inequality is the area under the diagonal line from (0,0) to $(n, \log_2 n)$ and the right side $(n \log_2 n)$ is the area under the rectangle of height $\log_2 n$. From the above inequalities, we conclude that in the worst of building a heap is $\Theta(n \log_2 n)$. I



removeMin

Next, recall the `removeMin()` algorithm from last lecture. We can write this algorithm using the array representation of heaps as follows.

```
removeMin(){
    element = heap[1]            //   heap[0] not used.
    heap[1] = heap[size]
    heap[size] = null
    size = size - 1
    downHeap(1, size)       //   see next page
    return element
}
```

This algorithm saves the root element to be returned later, and then moves the element at position `size` to the root. The situation now is that the two children of the root (node 2 and node 3) and their respective descendents each define a heap. But the tree itself typically won't satisfy the heap property: the new root will be greater than one of its children. In this typical case, the root needs to move down in the heap.

The `downHeap` helper method moves an element from a starting position in the array down to some maximum position in the heap. I will use this helper method in a few ways in this lecture.

```
downHeap(start,maxIndex){     //  move element from starting position
                              //  down to at most position maxIndex
   i = start
   while (2*i <= maxIndex){              // if there is a left child
      child = 2*i
      if (child < maxIndex) {           // if there is a right sibling
         if (heap[child + 1] < heap[child])
                                        //  if rightchild < leftchild ?
         child = child + 1
      }
      if (heap[child] < heap[ i ]){   // swap with child?
         swapElements(i , child)
         i = child
      }
      else  break    // exit while loop
   }
}
```

This is essentially the same algorithm we saw last lecture. What is new here is that (1) I am expressing it in terms of the array indices, and (2) there are parameters that allow the downHeap to start and stop at particular indices.

Faster algorithm for building a heap

Last lecture I showed you an $O(n \log_2 n)$ algorithm for building a heap. I will next present algorithm that runs in time $O(n)$. The faster algorithm is based on the `downHeap()` method from last lecture, where the two parameters are `startIndex` and `maxIndex` in the heap array. The input is a list with `size` elements. The output is a heap.

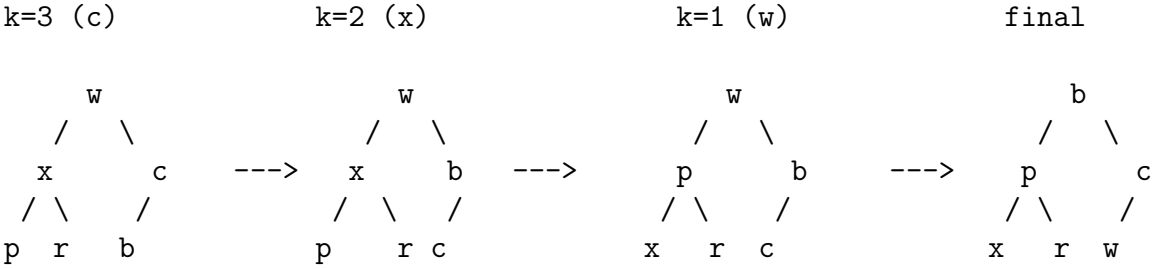
```
buildHeapFast(list){
  create new heap array //  size == 0,  length > list.size
  for (k = size/2; k >= 1; k--)
     downHeap( k, size )
}
```

The algorithm begins at node `k = n/2` and decrements the index down to the root node `k = 1`. For each `k`, it downHeaps, that is, it swaps the element from starting position `k` with the smaller of its children and repeats this until it is less than both its children (if it has any children).

The reason that the algorithm starts at `k = n/2` is that the nodes `size/2+1` to `size` have no children to compare with. So we don't bother downHeaping them.

Example

An initial arrangement of $n = 6$ keys is shown on the left. I show the state of the tree before the `k`th node is downHeaped, and the final state.



Worst case analysis for `buildHeapFast`

For each `k` of the `buildHeap` algorithm, the *worst case* number of swaps done by `downHeap()` is the height of the node `k` 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.* Recall that the height of a node in a tree is the maximum path length from the node to a leaf.

Let h be the height of the tree i.e. the height of the root node. Let's assume for mathematical analysis that we have a complete binary tree of height h and that level h is full. (All other levels are full by definition.) In this case, you can see by inspection that the height of every node at level l will be $h - l$. That is, the height of the root node (level 0) is h, the height of the two children of the root are $h - 1$, etc, and the height of all leaf nodes is $h - h = 0$.

Define $t_{worstcase}(n)$ be the sum of heights of all nodes. We write it in terms of h and sum over levels l:

$$\begin{aligned} t_{worstcase}(h) &= \sum_{l=0}^{h} (h-l)\ 2^l \\ &= h\sum_{l=0}^{h} 2^l - \sum_{l=0}^{h} l\ 2^l \end{aligned}$$

The first term is $h(2^{h+1}-1)$. The second term is the sum of the depths (or levels) of all the nodes. It is a bit trickier to solve.

I show in the Appendix (end of today's lecture notes) that:

$$\sum_{l=0}^{h} l\ 2^l = (h-1)2^{h+1}+2$$

Plugging into the term terms above, we get

$$t_{worstcase}(h) = h(2^{h+1}-1) - (h-1)2^{h+1} - 2$$

which we can simplify to

$$t_{worstcase}(h) = 2^{h+1} - h - 2$$

To write $t_{worstcase}(n)$ in terms of n rather than h, we recall that we are assuming *all* levels of the tree are full, i.e. including level $l = h$ which is the height of the tree. So,

$$n = 2^{h+1} - 1$$

and so

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

Substituting for h, we get

$$t_{worstcase}(n) = n - \log(n+1).$$

Remarkably, this is less than n. In particular, $t_{worstcase}(n)$ is $\Theta(n)$.

The intuition here is that most of the nodes in the tree are near the leaves, 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.

Heapsort

A heap can be used to sort a set of elements. The idea is simple. Just repeatedly remove the minimum element by calling `removeMin()`. This naturally gives the elements in their proper order.

Here I give an algorithm for sorting "in place". We repeatedly remove the minimum element the heap, reducing the size of the heap by one each time. This frees up a slot in the array, and so we insert the removed element into that freed up slot.

The pseudocode below does exactly what I just described, although it doesn't say "`removeMin()`. Instead, it says to swap the root element i.e. `heap[1]` with the last element in the heap i.e. `heap[size+1-i]`. After `i` times through the loop, the remaining heap has `size - i` elements, and the last `i` elements in the array hold the smallest `i` elements in the original list. So, we only downheap to index `size - i` .

```
heapsort(list){
  buildheap(list)
  for i = 1 to size-1{
    swapElements( heap[1], heap[size + 1 - i])
    downHeap( 1,  size - i)
  }
  return reverse(heap)
}
```

The end result is an array that is sorted from largest to smallest. Since we want to order from smallest to largest, we must `reverse` the order of the elements. This can be done in $\Theta(n)$ time, 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 elements (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)
```

Note that the last pass through the loop doesn't do anything since the heap has only one element left (`w` in this example), which is the largest element. We could have made the loop go from `i = 1 to size - 1`.

Appendix

$$\begin{aligned}
\sum_{l=0}^{h} l\ 2^l &= \sum_{l=0}^{h} l\ (2^{l+1} - 2^l) \qquad \text{(trick)} \\
&= \sum_{l=0}^{h} l\ 2^{l+1} - \sum_{l=0}^{h} l\ 2^l \\
&= \sum_{l=0}^{h} l\ 2^{l+1} - \sum_{l=0}^{h-1} (l+1)\ 2^{l+1} \qquad \text{second goes to h} - 1 \text{ only} \\
&= h\ 2^{h+1} + 2\sum_{l=0}^{h-1} (l - (l+1))\ 2^l \\
&= h\ 2^{h+1} - 2\sum_{l=0}^{h-1} 2^l \\
&= h\ 2^{h+1} - 2(2^h - 1) \\
&= (h-1)2^{h+1} + 2
\end{aligned}$$

Maps

The last few lectures, we have looked at ways of organizing a set of elements that are comparable. Today we will begin looking a different way of organizing things which doesn't require that they are comparable. We will mainly be looking at *maps*.

You are familiar with maps already. In high school math and in Calculus and linear algebra, you have seen functions that go from (say) $\Re^n$ to $\Re^m$. In COMP 250, we have seen functions $t(n)$ that describe how the number of operations performed by some algorithm depends on the input size n. In general, a map is a set of ordered pairs $\{(x, f(x))\}$ where x belongs to some set called the *domain* of the map, and $f(x)$ belongs to a set called the *co-domain*. The word *range* is used specifically for the set $\{f(x) : (x, f(x)) \text{ is in the map}\}$. That is, some values in the co-domain might not be reached by the map.

Maps as (key,value) pairs

You are also 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 care number or student number for the index, which maps to a person's employment record, health file, or student record, respectively.

Maps are defined as follows. Suppose we have two sets: a set of keys K, and a set of values V. A *map* is a set of ordered pairs

$$M = \{(k, v) : k \in K, v \in V\} \subseteq K \times V.$$

The pairs are called *entries* in the map. A map cannot just be any set of (key, value) pairs. Rather, for any key $k \in K$, there is *at most* one value v such that (k, v) is in the map. We allow two different keys to map to the same value, but we do not allow one key to map to more than one value. Also note that not all elements of K need to be keys in the map.

For example, let K be the set of integers and let V be the set of strings. Then the following is a map:

$$\{(3, \texttt{cat}), (18, \texttt{dog}), (35446, \texttt{meatball}), (5, \texttt{dog}), \}$$

whereas the following is not a map,

$$\{(3, \texttt{cat}), (18, \texttt{dog}), (35446, \texttt{meatball}), (5, \texttt{dog}), (3, \texttt{fish})\}$$

because the key 3 has two values associated with it.

Map ADT

The basic operations that we perform on a map are:

- put(key, value) – this adds a new entry to the map
- get(key) – this gets the entry (key, value), and it would return null if the given key did not have an entry in the map.
- remove(key) – this removes the entry (key, value) – this might return the value, or else return null if the key wasn't present

There are other methods such as `contains(key)` or `contains(value)` as well but the above are the main ones.

Map data structures

How can we represent a map using data structures that we have seen in the course? 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`. We could use an arraylist or a linked list of entries, for example. This would mean that the operations defined above would be slow in the worst case, however, namely the worst case would be $\Theta(n)$ if the map has n entries.

What if we made a stronger assumption, namely what if the keys of map were comparable? In this case, to organize the entries of the map, we could use a sorted arraylist or a binary search tree. If we used a sorted arraylist, then we could find a key in $O(\log n)$ steps, where n is the number of pairs in the map. Each slot of the sorted arraylist would hold a map entry. We would use binary search to find the entry, based on the key. However, with a sorted array it can be relatively slow to `put` or `remove` a (key,value) pair, namely in the worst case it is $\Theta(n)$.

We could instead use a binary search tree (BST) to store the (k, v) pairs, namely we index by comparing keys. Although the binary search trees we have covered have $\Theta(n)$ worst case behavior, I have told you that there are fancier versions of binary search trees that are balanced that allow adding, removing, finding in $O(\log n)$ time. You'll learn about these in COMP 251.

What about a heap? A heap would not be an appropriate data structure for a map because it only allows you to efficiently get or remove the minimum element. However, for a general `get`

operation, a heap would be $\Theta(n)$, since one would have to traverse the entire heap. Note that the heap is represented using an array, so this would just be a loop through the elements of the array.

Another special case to consider is that the keys `K` are positive integers in a small range. In this case, we could just use an array and have the key be an index into the array and the value be the thing stored in the array. Note that this typically will *not* be an arraylist, since there may be gaps in the array between entries. Using an array would give us access to map entries in $O(1)$ time. However, this would only work well if the integer values are within a small range. For example, if the keys were social insurance numbers (in Canada), which have 9 digits, we would need to define an array of size 10^9 which is not feasible since this is too big.

Despite this being infeasbile, let's make sure we understand the main idea here. Suppose we are a company and we want to keep track of employee records. We use social insurance number as a key for accessing a record. Then we could use a (unfeasibly large) array whose entries would be of type `Employee`. That is, you would use someone's social insurance number to index directly into the array, and that indexed array slot would hold a reference to an `Employee` object associated with that social insurance number. Of course this would only retrieve a record if there were an an `Employee` with that social insurance number; otherwise the reference would be null and the `find` call would return null.

For the rest of today, we consider the case that the keys map to a set of integers, without requiring these integers to be a small range. Next lecture we will address the problem of mapping to a small range of integers and treating these integers as an index into an array.

Example: Object.hashCode()

Let's jump right in and consider a map that you may be less comfortable with at this point but which is hugely useful when programming in Java. When a Java programming is running, every object is located somewhere in the memory of the Java Virtual Machine (JVM). This location is called an *address*. In particular, each Java object has a unique 32 bit number associated with it which by default is the number returned when the object calls `hashCode()` method. (What exactly this 32 bit number means may depend on the implementation of the JVM. We will just assume that the number is the object's (starting) address in the JVM.) Since different objects must be non-overlapping in memory, it follows that they have different addresses. In particular, when the object's `hashCode()` is the object's address, the expression "`obj1 == obj2`" means the same thing as "`obj1.hashCode() == obj2.hashCode()`". The statement can be either true or false, depending on whether the variables `obj1` and `obj2` reference the same object or not.

Example: String.hashCode()

The above discussion about the `hashCode()` method only considers the default implementation. Some classes override this default `hashCode()` method. For example, each String object is also located somewhere in memory but the String class uses a different `hashCode()` method.

To define the `hashCode()` method of the `String` class, let's first consider a simple map from `String` to positive integers. Suppose s is a string which consists of characters $s[0]s[1]...s[s.length-1]$. Consider the function

$$h(s) = \sum_{i=0}^{s.length-1} s[i]$$

which is just the sum of the codes of the individual characters. Notice that two strings that consist of the same letters but in different order would have the same $h()$ value. For example, "eat", "ate", "tea" all would have the same code. Since the codes of `a, e, t` are 97, 101, and 116, the code of each of these strings would be 97+101+116.

In Java, the `String.hashCode()` method is defined[13] :

$$h(s) = \sum_{i=0}^{s.length-1} s[i]\ x^{s.length-i-1}$$

where $x = 31$. So, for example,

$$h(\text{"}eat\text{"}) = 101 * 31^2 + 97 * 31 + 116$$

and

$$h(\text{"}ate\text{"}) = 97 * 31^2 + 116 * 31 + 101$$

Thus, here we have an example of how strings that have the same letters do not have the same hashCode.

[13]You might wonder why Java uses the value $x = 31$. Why not some other value? There are explanations given on the website stackoverflow, but I am not going to repeat them here. Other values would work fine too.

What about if two strings have the same hashcode? Can we infer that the two strings are equal? No, we cannot. I will include this as an exercise so that you can see why.

ASIDE: Horner's rule

Suppose you wish to evaluate a polynomial

$$h(s) = \sum_{k=0}^{N} a_k \; x^k$$

It should be obvious that you don't want to separately calculate $x^2, x^3, x^4, ..., x^N$ since there would be redundancies in doing so. We only should use $O(N)$ multiplications, not $O(N^2)$. Horner's Rule describes how to do so.

The following example gives the idea of Horner's rule for the case of hashcodes for a `String` object with four characters:

$$s[0] * 31^3 + s[1] * 31^2 + s[2] * 31 + s[3] = ((s[0] * 31^1 + s[1]) * 31 + s[2]) * 31 + s[3]$$

For a `String` object of arbitary length, Horner's rule is the following algorithm :

```
h = 0
for  (i = 0;  i < s.length;  i++)
    h = h*31 +  s[i]
```

In the lecture, I ran out of time and did not present Horner's rule. But I would like you to see it and understand it since it is a simple idea and worth knowing.

Recall the scenario from last lecture: 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 quickly access the associated value v. If the keys were integers in a small range, say $0, 1, \ldots, m-1$, then we could just use an array of size m and the keys could indices into the array. The locations k in the array would correspond to the (integer) keys in the map and the slots in the array would hold references to the corresponding values v.

In most situations, the keys are not integers in some small range, but rather they are in a large range, or the keys are not integers at all – they may be strings, or something else. In the more general case, we define a function – called a *hash function* – that maps the keys to the range $0, 1, \ldots, m-1$. Then, we put the two maps together: the hash function maps from keys to a small range of integer values, and from this small range of integer values to the corresponding values v. Let's now say more about how we make hash functions.

Hash function: hash code followed by compression

Given a space of keys K, a *hash function* is a mapping:

$$h : K \to \{0, 1, 2, m-1\}$$

where m is some positive integer. That is, 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*.

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 map is called a *hash code*, 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 mod m. (Often one takes m to be a prime number, though this is not necessary.) The mod function can be defined for negative numbers, such it returns a value in 0 to $m-1$. However, the Java mod function doesn't work like that. e.g. In Java, the expression "-4 mod 3" evaluates to -1 (rather than 2, which is what one would expect after learning about "modulo arithmetic"). To define a compression function using mod in Java, we need to be a bit more careful since we want the result to be in $\{0, 1, \ldots, m-1\}$.

The Java `hashCode()` method returns an `int`, and `int` values can be either positive or negative, specifically, they are in $\{-2^{31}, \ldots -1, 0, 1, \ldots 2^{31}-1\}$. (You will learn how this works in COMP 273.) The compression function that is used in Java is

$$| \texttt{hashCode}() \mid \text{ mod } m,$$

i.e. gives a number in $\{0, 1, \ldots, m-1\}$, as desired.

To summarize, a hash function is typically composed of two functions:

$$\text{hash function } h \; : \quad \text{compression} \; \circ \; \text{hash code}$$

where $\circ$ denotes the composition of two functions, and

$$\text{hash code} \quad : \quad \text{keys K} \to \text{integers}$$

$$\text{compression} \quad : \quad \text{integers} \rightarrow \{0, .., \text{m} - 1\}$$

and so

$$h : \text{keys } K \rightarrow \{\text{hash values}\},$$

i.e. the set of hash values is $\{0, 1, \ldots, m-1\}$. Note that a hash function is itself a map!

It can happen that two keys k_1 and k_2 have the same hash value. This is called a *collision*. There are two ways a collision can happen. First, two keys might have the same hash code. Second, two keys might have different hash codes, but these two different hash codes map (compress) to the same hash value. In either case we say that a *collision* has occured. We will deal with collisions next.

Hash map (or Hash table)

Let's return to our original problem, in which we have a set of keys of type K and values of type 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: Keep in mind there are a few different maps being used here, and in particular the word "value" is being used in two different ways. The values $v \in$ `V` of the map we are ultimately trying to represent 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 $\{0, 1, \ldots, m-1\}$.]

To represent the (k, v) pairs in our map, we use an array. The number of slots m in the array is typically bigger than the number of (k, v) pairs in the map.

As we discussed above, we say that a collision occurs when two keys map to the same hash value. Since the hash values are indices in the array, a collision leads to two keys mapping to the same slot in the array. For example, if $m = 5$ then hash codes 34327 and 83322 produce a collision since $34327 \bmod 5 = 2$, and $83322 \bmod 5 = 2$, and in particular both map to index 2 in the array.

To allow for collisions, we use a linked list of pairs (k, v) at each slot of our hash table array. These linked lists are called *buckets*.[14] Note that we need to store a linked list of pairs (k, v), not just values v. The reason is that 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 (bucket) in the hash table. We then try to find the corresponding value v in the list. We examine each entry (k, v) and check if the search key equals the key in that entry.

Good vs. bad hash functions

Linked lists are used to deal with collisions. We don't want collisions to happen, but they do happen sometimes. A good hash function will distribute the key/value pairs over the buckets such that, if possible, there is at most one pair per bucket. This is only possible if the number of entries in the hash table is no greater than the number of buckets. We define the *load factor* of a hash table to be the number of entries (k, v) in the table divided by the number of slots in the table (m). A load factor that is slightly less than one is recommended for good performance.

[14] Storing a linked list of (k, v) pairs in each hash bucket is called *chaining*.

Having a load factor less than one does not guarentee good performance, however. In the worst case that all they keys in the collection hash to the same bucket, then we have one linked list only and access is $O(n)$ where n is the number of entries. This is undesirable obviously. To avoid having such long lists, we want to choose a hash function so that there are few, if any, collisions.

The word *hash* means to "chop/mix up".[15] We are free to choose whatever hash function we want and we are free to choose the size m of the array. So, it isn't difficult to choose a hash function that performs well in practice, that is, a hash function that keeps the list lengths short. In this sense, one typically regards hash tables as giving $O(1)$ access. To prove that the performance of hash tables really is this good, one needs to do some math that is beyond COMP 250.

As an example of a good versus bad hash function, consider McGill student ID's which are 9 digits. Many start with the digits 260. If we were to have a hash table of size say $m = 100,000$, then it would be bad to use the first five digits as the hash function since most students IDs would map to one of those two buckets. Instead, using the last five digits of the ID would be better.

Java `HashMap` and `HashSet`

In Java, the `HashMap<K,V>` class implements the hash map that we have been describing. The `hashCode()` method for the key class `K` is composed with the "mod m" (and absolute value) compression function where m is the capacity of an underlying array, and it is an 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 default maximum load factor for the hash table is than 0.75 and there is a default array capacity as well. The `HashMap` constructor allows you specify the initial capacity of the array, and you can also specify the maximum load factor. 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 (or the value you specify), 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.

Java also has a `HashSet<T>` class. This is similar to a HashMap except that it only holds objects of type `<T>`, not key/value pairs. It uses the `hashCode()` method of the type `T`. Why would this class be useful? Sometimes you want to keep track of a set of elements and you just want to ask questions such as, "does some element e belong to my set, or not?" You can add elements to a set, remove elements from a set, compute intersections of sets or unions of sets." What is nice about the `HashSet` class is that it give you quick access to elements. Unlike a list, which requires $\Theta(n)$ operations to check if an element is present in the worst case, a hash set allows you to check in time $O(1)$ – assuming a good hash function.

[15] It should not be confused with the # symbol which is often the "hash" symbol i.e. hash tag on Twitter, or with other meanings of the word hash that you might have in mind.

Cryptographic hashing

Hash functions comes up in many problems in computer science, not just in hash table data structures. Another common use is in password authentication. For example, when you log in to a website, you typically provide a username or password. On a banking site, you might provide your ATM bank card number or your credit card number, again along with a password. What happens when you do so?

You might think it works like this. The web server has a file that lists all the user names and corresponding passwords (or perhaps a hash map of key-value pairs, namely usernames and passwords, respectively). When you log in, the web server takes your user name, goes to the file, retrieves the password stored there for that username, and compares it with the password that you entered. This is *not* how it works however. The reason is that this method would not be secure. If someone (an employee, or an external hacker) were to break in an steal the file then he would have access to all the passwords and would be able to access everyone's data.

Instead, the way it typically works is that the webserver stores the user name and the *hashed* passwords. Then, when a user logs in, the web server takes the password that the user enters, hashes it (that is, applies the hash function), throws away the password entered by the user, and compares the hashed password to the hashed password that is stored in the file.

Why is then any better? First, notice that if a hacker could access the hashed password, this itself would not be good enough to log in, since the process of logging in requires entering a password, not a hashed password. Second and more interesting, you might wonder if it is possible to compute an *inverse hash map*: given a hashed password, can we compute the original password or perhaps some other password that is hashmapped to the given hashed password. If such a computation were feasible, then this inverse hashmap could be used again to hack in to a user's account.

However, "cryptographic" hashing functions are designed so that such a computation is *not* feasible. These hashing functions have a mixing property that two strings that differ only slightly will map to completely different hash values, and given a hash value, one can say almost nothing about a keyword that could produce that hash function.

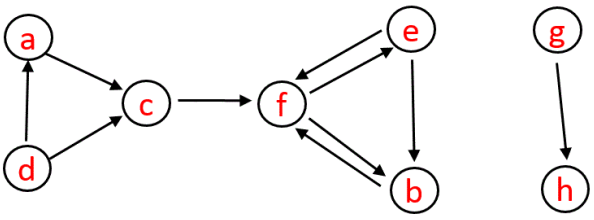
A few final observations. First, a cryptographic hashing function does not need to be secret. Indeed there are standard cryptographic hashing functions. One example is MD5 `http://www.miraclesalad.com/webtools/md5.php`. This maps any string to a sequence of 128 bits.[16]

Second, cryptographic hashing is not the same as encryption. With the latter, one tries to encode a string so that it is not possible for someone who is not allowed to know the string cannot decrypt the coded string and have the original one again. However, it *should* be possible for someone who *is* allowed to know the original string to be able to "decrypt" the encrypted string to get the original message back. So, the main dfference is that with encryption it *is* possible to invert the "hash" function. (It isn't called hashing, when one is doing encryption, but the idea is similar.) You will learn about encryption/decryption if you take MATH 240 Discrete Structures.

[16] If you check out that link, you'll see that the hash values are represented not as 128 bits there, but rather as 32 *hexadecimal* digits. I haven't discussed hexadecimal in the course, but the idea is simple: it is just a code for 4-bit strings. The digits 0 to 9 are used for 0000 to 1001 in the obvious way, namely binary coding. The letters 'a' to 'f' are used for the other five combinations of bits, which correspond to binary codes for numbers 10 to 15).

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. Here is an example.



Like in many previous data structures, a graph contains a set nodes. Each node has a reference to other nodes. For 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 to children nodes or siblings or parent nodes. In a 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 basic results go back a few hundred years, to mathematicians like Euler. Mathematically, a graph consists of a set V called "vertices" and a set of edges $E \subseteq V \times V$. The edges E in a graph is a set of *ordered* pairs of vertices.

When the ordering of the vertices in an edge is important, we say that we have a *directed graph*. One can also define graphs in which the edges do not have "arrows", that is, each edge is a pair of vertices but we don't care about the order. This is called an *undirected* graph, and in this case we would draw the edges as line segments with no arrows.

Examples of graphs include transportation networks. For example, V might be a set of airports and E might be direct flights between airports. There are many other examples. V might be a set of html documents and E would be the URLs (links) between documents. Another example is that V might be a set of objects in a running Java program, and E would be a set of references, namely when an object has a field (reference variable) that references another object. In a future lecture, I will discuss the graph of html documents and examine how google search works. I will also discuss the graph of objects referencing each other and examine how garbage collection works.

Terminology

Here is some basic graph terminology that you should become familiar with, and that is heavily used in discussing properties of graphs. Please see the slides for examples.

- *weighted graph* – a graph that has a number (weight) associated with each edge; for example, in a flight network where the vertices are airports, the weight might be the time it takes to fly between two airports
- *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 vertices $(v_1, \ldots, v_m)$ where $(v_i, v_{i+1}) \in E$ for each i. The length of a path in a graph is the number of edges in the path (not the number of vertices). The definition of path is essentially the same for graphs as it was 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 called loops.
- *directed acyclic graph* (DAG) – a directed graph that has no cycles. Such graphs are used to capture dependencies between objects or events. For example, the graph of prerequisite relationships in McGill courses is directed and acyclic.

In the lecture itself, I briefly discussed a few important graph problems that you will see in subsequent courses.

Adjacency List

A graph is a generalization of a tree. Each node in a tree has a list of children. Similarly, each graph vertex v has a list of other vertices w that are adjacent to it. We call this an *adjacency list*, namely for each vertex $v \in V$, we represent a list of vertices w such that $(v, w) \in E$. For example, here is the adjacency lists for the graph on the previous page.

```
a - c
b - f
c - f
d - a,c
e - b,f
f - b,e
g - h
h -
```

I have represented vertices in alphabetic order, but this is not necessary.

Data structures for graphs

Java does not have a `Graph` class since there are too many different types of graphs and no one size fits all. So one needs to implement one's own `Graph` class. A very basic `Graph` class might be as simple as this:

```
class Graph<T>{

   class Vertex<T> {
      LinkedList<Vertex<T>>  adjList;
      T                      element;
   }
}
```

However, it is common to have other vertex attributes and also to have attributes for the edges, in particular, edge weights. So a more common graph would be like this:

```
class Graph<T>{

   class Vertex<T> {
       LinkedList<Edge<T>>  adjList;
       T                    element;
   }

   class Edge<T> {
      Vertex<T>            endVertex;
      double               weight;
      :
}
```

Now the adjacency list for a vertex is a list of `Edge` objects and each edge is represented only by the end vertex of the edge. The start vertex of each edge does not need to be represented explicitly because it is the vertex that has the edge in its adjacency list.

An important difference between rooted trees and graphs is that rooted trees have a special node (the root) where methods between. For graphs, we may wish to access any node. For this, we need a map.

We will have a label (key) for each of the vertices, and we will use the key to access the vertices by using a hash map. The key might be a string, for example.

```
class  Graph<T>{
     HashMap< String, Vertex<T> >    vertexMap;
       :
        //  Vertex and Edge inner classes as above
}
```

Using the classes above, how many Java objects are there for the example graph on page 1? There are eight vertices and hence eight `Vertex` objects. There is one `HashMap` for identifying the keys with the vertices. (The HashMap itself has an underlying array and linked list structure which we ignore here.) Each `Vertex` object has an adjacency list, which is a `LinkedList<Edge>` object. There are also ten `Edge` objects, and these are referenced by the nodes in the `LinkedList` (and these linked list nodes are also objects, but I ignore them here).

Adjacency Matrix

A different data structure for representing the edges in a graph is 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 (called *loops*). An example is given in the slides.

I finished the lecture by comparing adjacency lists and adjacency matrices. When would you use one rather than another. See the Exercises.

Graph traversal

One problem we often need to solve when working with graphs is to decide if there is a sequence of edges (a path) from one vertex to another, or to find such a sequence. There are various versions of this problem, and the most familiar one to you is probably Google Maps which finds the shortest path from one location to another. You will learn more about shortest path problems in COMP 251.

Today we will consider the problem of finding the set of all vertices that can be reached from a given vertex v, or equivalently, the set of all vertices w for which there is a path from v to w. In the shortest path problem you will see in COMP 251, "shortest" refers to the sum of the edge weights and one tries to find the path from v to w that minimizes this sum.

Depth First Traversal

Recall the depth first traversal algorithm for trees.

```
depthfirst_Tree(root){
   if (root is not empty){
      visit root                        // preorder
      for each child of root
         depthfirst_Tree(child)
   }
}
```

This algorithm generalizes to graphs as follows.

```
depthfirst_Graph(v){
   v.visited = true
   for each w such that (v,w) is in E
      if !w.visited                              //  avoids cycles
         depthfirst_Graph(w)
}
```

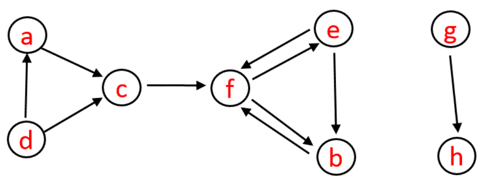
A few notes: first, I call this algorithm a "traversal" but in fact it only reaches the nodes in the graph for which there is a path from the input vertex. This algorithm is sometimes called (depth first) "search" since we are searching for all vertices that can be reached from the input vertex.

Second, before running this algorithm, we need to set the `visited` field to false for all vertices in the graph. To do so, we 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 were to use a linked list to represent all the vertices in the graph, then you would traverse this linked list and set the `visited` field to false, before you called the above traversal algorithm. 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.

Example *(see slides for another example)*—

Let's run the above preorder depth first traversal algorithm on the graph shown below. Also shown is the *call stack* and how it evolves over time, and the *call tree*. (A node in the call tree represents one "call" of the `depthfirst_Graph` 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               CALL TREE
           b   e                      a
         f f f f f                    |
       c c c c c c c                  c
     a a a a a a a a a                |
                                      f
   ------------------->  time        / \
                                    b   e
```

Notice that nodes `d`, `g`, `h` are not visited.

Non-recursive depth first traversal

We do not need recursion to do a depth first traversal. We can do depth first traversal using a stack. Our algorithm here generalizes the non-recursive tree traversal algorithm that used a stack.

The tree traversal algorithm using a stack went like this:

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

Here we visited each node after popping it from the stack. Let's rewrite this slightly so that we visit each node *before* pushing it onto the stack.


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

Now let's generalize this to graphs. In the tree case, we pushed the children of a node onto the stack. In the graph case, we will push the adjacent vertices (nodes) onto the stack. *We only do so, however, if the adjacent vertex has not yet been visited.* The reason is that the graph may contain a cycle and we want to ensure that a vertex does not get pushed onto the stack more than once, since this would lead to an infinite loop.

```
 depthFirst(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. We visit a vertex before we visit any of the children vertices.

Breadth first traversal

Like depth first traversal, breadth first traversal finds all vertices that can be reached from a given vertex v. However, breadth first traversal visits all vertices that are one edge away, before it visits any vertices are two vertices away, etc. We have already seen breadth first traversal in trees, also known as level order traversal. Breadth traversal in graphs is more general: the levels correspond to vertices that are a certain distance away (in terms of number of edges i.e. path length) from a given vertex.

Since we are working with a graph rather than a tree, we visit the nodes before enqueueing them.

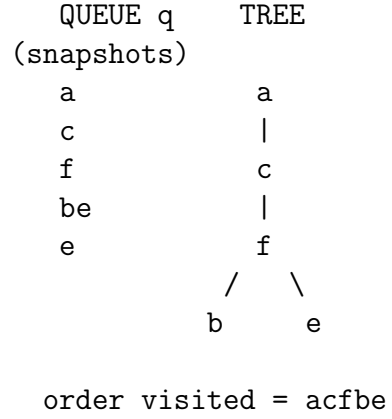
```
breathFirst(u){
  initialize empty queue q
  u.visited = true
  q.enqueue(u)
  while ( !q.empty) {
    v = dequeue()
    for each w in adjList(v)
      if !w.visited{
        w.visited = true
        enqueue(w)
      }
}
```

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 that we are searching from.

Note how the queue evolves over time, and the order in which the nodes are visited. Also note that three in this case happens to be the same as the tree defined by the depth first traversal.



order visited = acfbe

Another Example

Here is another example, which better illustrates the difference between `dft` and `bft`. For the sake of simplicity (drawing!), I suppose here that the graph is undirected. In the slides, I did a directed version of this.

```
  GRAPH      ADJACENCY LIST    depth first       depth first         breadth first
                               (recursive)         (stack)            (queue)

a - b - c    a - (b,d)         a - b - c         a - b   c           a - b - c
|   |   |    b - (a,c,e)              |         |       |           |   |   |
d - e - f    c - (b,f)         d - e - 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           g   h   i
             f - (c,e,i)
             g - (d,h)
             h - (e,g,i)
             i - (f,h)

           order visited:      abcfedghi         abdeghifc           abdcegfhi
```

[ADDED Nov. 22]

For the depth first stack case, the order listed in the originally posted notes and slides was incorrect. We visit before we push. (I had notice this mistake during one of the two sections during the lecture, but I must have forgotten to correct it. Sorry for the confusion!)

Note that in both the depth first stack and the breadth first case, the order in which vertices are visited here cannot be directly inferred from the structure of the tree.

Inheritance

In our daily lives, we classify the many things around us. 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 all 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. 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 "super" class that the subclass "extends". 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          birthDate
   Date          deathDate
   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 later.

Constructor chaining

When an object of a subclass is instantiated using one of the subclass's constructors, 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 superclass'es constructor with no parameters to be executed.

Regardless of whether they have parameters or not, superclasses may have fields and the superclass constructors may set these fields to some value. These superclass fields and values are inherited by the subclass.

Note that the superclass may have its own `super(..)` statement, or it may not – and in that case it gets the default `super()` – and so on, which causes the fields of *all* the ancestor classes automatically to be inherited and initialized.

The following example illustrates some of the details of constructor chaining. The superclass `Animal` has two constructors. The subclass `Dog` constructor chooses among them by including parameters of the `Dog` constructors `super()` calls to match the signature (number and types of arguments) of the superclass constructor. (That's just how it is done for this example, but it is not

required.) Specifically, the class `Dog` has a `String` field that specifies the owner. The `Dog(Place, String)` constructor could in principle have used either the `Animal()` or `Animal(Place)` constructor. It does the latter by calling `super(home)`. (It could have done the former by not having a `super(..)` call which would have defaulted to a `super()` call.

```
class Animal {
  Place  home;

  Animal() { ... }

  Animal( Place home) {
     this.home  =  home;
  }
}

class  Dog  extends  Animal {
  String owner;
  String name;

  Dog() { ...  }   //  This constructor automatically calls super() which creates
                   //    fields  that are inherited from the superclass

  Dog(Place home,  String  owner) {
     super(home);           //   Here we need to explicitly write which
                            //    super constructor to use.
     this.owner  = owner;
  }
  :
}
```

A few more details:

- Java does not allow you to write `super.super`. There is no way for a sub-class to explicitly invoke a method from the superclass'es superclass.
- 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.

If you want to learn more, see online tutorials
`docs.oracle.com/javase/tutorial/java/IandI/subclasses.html`

Overriding ≠ overloading

Overriding a method is different from overloading it. When a subclass method and superclass method have the same method name and the same number, types, and order of parameters, then we say that the subtype method *overrides* the supertype method. When the method name is the same but the type, number, or order of parameters changes, then we say the method is *overloaded*. Overriding can only occur from a child class (subclass) to parent class (superclass). Overloading can occur either within classes or between a child and parent class.

Note that the return type doesn't play a role in either case. The reason is that, whether you are overriding or overloading, you are defining a new method and so you can define whatever return type you want. This distinction is consistent with the definition of a method's *signature*: the "signature" is the method name and the types and order of the method parameters. The return type is not part of the method signature. Thus, one overrides a method when the signature is the same, and one overloads a method when the signature changes. (Of course, understanding this distinction depends on your understanding the definition of signature!)

Overloading within a class

Let's first consider overloading of a method *within* a class. We have seen examples of this already, such as the `add` and `remove` methods of the ArrayList and LinkedList classes.

Another example is 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 only know the dog's name. Other times I may know the dog's name and the owner's name, and other ties neither. So I may have different constructors for 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`.

Notice that the following constructors are actually the same and including them both will generate a compiler error. i.e. The parameter identifiers (`ownerName` vs. `dogName`) are not part of the method signature. Only the parameter types matter.

```
public  Dog(String ownerName){
        :
}
public  Dog(String dogName){
```

```
        :
}
```

Overloading between classes

As explained above, we use the term *overloading* if we have a method that is defined in a subclass and in a superclass and 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 of the these new fields as a parameter in the method's signature in the subclass. Or, the superclass type might be a parameter type in the superclass method, and in the corresponding subclass method we might replace the type of the corresponding parameter with the subclass type. For example, `greet(Animal a)` in the `Animal` class might be overloaded by having `greet(Dog dog)` in the `Dog` class.

Java `Object` class

Java allows any class to directly extend at most one other class. 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` (or some subclass of `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 that `Object` is an ancestor. It is the root of the class tree.

Java `equals( Object )` method

In natural languages such as English, when we talk about particular instances of classes e.g. 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 `Object` class, the `equals(Object)` method does the same thing as the "==" operator, namely it checks if two referenced *objects* are the same.

For many other classes, we may want to override the `equals(Object)` method, namely use a less restrictive version of the `equals` method. We may also wish to overload it, for example, by defining an `equals(Dog)` method in the `Dog` class. For today, I will not get into why we might want to override versus overload. I'll try to say something about it in a future lecture.

We have an intuitive notion of what we mean by 'equals'. But since the *equals()* method is so fundamental in Java, the designers of the Java language specified quite formally how the method should behave. It is very similar to the mathematic definition of *equivalence classes* which you learn about in MATH 240. When writing your own classes and overriding this method, you should be aware of this. See details here:
`https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-`

For example,

- `x.equals(x)` should always be true
- `x.equals(y)` should have the same true or false value as `y.equals(x)`
- if `x.equals(y)` and `y.equals(z)` are both true, then `x.equals(z)` should be true.

The `String` class overrides the `equals(Object)` method. You may have been 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 where as a user you would think the result should be true in each case. Not so: note that some cases are `false`.

```
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 should be fine.

Another example mentioned in the lecture is the Java `LinkedList` class which also overrides the `equals(Object)` method. If a `LinkedList` object invokes this method on some other object, the result will be true only if that other object is also a LinkedList object and each of the elements in

the two lists are "equal", in particular, the elements of the two lists are "equal" according to the `equals` method of these elements. The Java API gives only a short description of this condition and (in my opinion) a few details are left unspecified. Buyer beware.

Java `hashCode()` method

The class `Object` has a method called `hashCode()` which returns an integer, that is, a 32 bit number between -2^{31} and $2^{31}-1$. The `String` class overrides this method. We have discussed this in the hashing lecture. Have a look at this definition in
`https://docs.oracle.com/javase/7/docs/api/java/lang/String.html`
and you'll see it is what we discussed.

Also have a look at the `LinkedList` class, which also overrides the `hashCode()` method.
`https://docs.oracle.com/javase/7/docs/api/java/util/AbstractList.html`
In fact this `hashCode()` method is inherited from an "abstract class called an `AbstractList`. We discuss abstract classes a few lectures from now.

[BEGIN – ADDED Nov. 20, 2017]

One important property of the `hashCode()` method is that if two objects are considered equal by the `equals(Object)` method, then they should return the same `hashCode()`. I will come back to this point a few times in the coming lectures.

[END]

`Java toString() method`

The `Object.toString()` method returns the object's `hashCode()` written as a string. The hashcode is a 32 bit integer. Interestingly, the `toString()` method doesn't convert this 32 bit `int` into the base 10 string representation you might expect. Rather it converts it into a string representation of the 32 bit int, written in base 16. This representation is known as hexadecimal. (See Appendix below.)

Specifically, the `Object.toString()` method returns a `String` object that has three parts: the class name of the object, the @ symbol, and the hexadecimal representation of the `int` returned by the `Object.hashCode()` method. Check it out by simplying testing the instruction: `System.out.print( new Object() )`.

Notice that if you define your own class 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 a string like "`Doberman@2934a212`."

More commonly, one overrides the `Object.toString()` method and prints out a description of an object, such as the values of its fields. As an author of the class, you are free to define `toString()` however you wish, as long as it return a `String`.

Appendix: Hexadecimal representation of binary strings

When we write down binary strings with lots of bits, we can quickly get lost. No one wants to look at 16 bit strings, and certainly not at 32 bit strings. A common solution is to use *hexadecimal*, which is essentially a base 16 representation. We group bits into 4-tuples ($2^4 = 16$). Each 4-bit group can code 16 combinations and we typically write them down as: `0,1,...,9,a,b,c,d,e,f`. The symbol `a` represents 1010, `b` represents 1011, `c` represents 1100, ..., `f` represents 1111.

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	a
1011	11	b
1100	12	c
1101	13	d
1110	14	e
1111	15	f

We commonly (but not always) write hexadecimal numbers as `0x_____` where the underline is filled with characters from `0,...,9,a,b,c,d,e,f`. For example,

`0x2fa3` = 0010 1111 1010 0011.

Sometimes hexadecimal numbers are written with capital letters. In that case, a large `X` is used as in the example `0X2FA3`.

If the number of bits is not a multiple of 4, then you group starting at the rightmost bit (the least significant bit). For example, if we have six bits string 101011 , then we represent it as `0x2b`. Note that looking at this hexadecimal representation, we don't know if it represents 6 bits or 7 or 8, that is, 101011 or 0101011 or 00101011. But usually this is clear from the context.

A few more details about `Object` class methods from last lecture

One point I didn't mention last lecture is that there is a relationship between the `equals(Object)` and `hashCode()` methods. The Java API for Object.hashCode() recommends that if `o1.equals(o2)` is true, then `o1.hashCode() == o2.hashCode()` also should be true.[17] Why? One reason is that the `hashCode()` method is used in the `HashMap<K,V>` class and other classes such as `HashSet<E>` as well. In particular, for the `HashMap<K,V>` class, suppose we have two variables `key1` and `key2` of type `K`. Suppose we put an entry into the hashmap using `put(key1, value)` and then later we have the instruction `get(key2)`. If `key1.equals(key2)` is true, then we would expect `get(key1)` to return the same value as `get(key2)`. Otherwise, what would be the point of saying that two keys are equal!? But we can only ensure that they get the same value if indeed they have the same hash codes.

Java `Object clone()` method

Another commonly used method in class `Object` is `clone()`. Recall this method from Assignment 1, when you cloned large `NaturalNumber` objects, for example, in the subtraction method. In general, the `clone()` 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()` should return false.
- The expression `x.equals( x.clone())` should returns true (suggested, but not required).

These two conditions make intuitive sense. The point of cloning is to create a different object instance (first condition), but the clone is suppose to be the same as the original in whatever sense we define "same" to mean for that object's class. Note that the second condition doesn't hold for `Object` objects. But that's ok, as one rarely clones `Object` objects.

One subtle aspect of cloning is that objects can reference other objects. So if we clone an object which has reference fields, then should we also clone the objects that are referenced? For example, if we clone a `LinkedList` object, then do we want the list and the objects in the list to be cloned? This is called a *deep copy*. Or do we just want to the list to be cloned but the objects that the list references should not be cloned? This is called a *shallow copy* . If you look up the Java API for `LinkedList`, you'll see that it specifies to make a shallow copy. But in general, there is no correct answer to what one should do. It is an issue of design, and what is appropriate for the situation in question.

ADTs versus APIs

[ASIDE: This discussion reiterates point made at the end of lecture 9.]

We have seen many abstract data types (ADTs): lists, stacks, queues, trees, binary search trees, priority queues, graphs, maps. Each ADT consists of a set of operations that one performs on the data. These operations are defined independently of the implementation in some programming

[17] The converse need not hold. E.g. Two different Java strings might have the same hashCode().

language. It is sometimes useful to keep the implementation details hidden, since they may just be distracting and irrelevant.

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 classes and methods (and possibly fields) and comments on what these methods do.

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 within each class and to specify (as comments) what operations the methods should perform. Eventually, you implement the methods.

A user (client) of the class should not need to see the implementation of a method to be able to use it, however. The user only sees the API. If the design is good, then all the client needs is a description of what the method does, and the method signatures: the return type, method name, and parameters with their types. (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 implements each method that is defined in the interface. In particular, the method signatures must be the same as in the interface. So, if we say a class `C` implements an interface `I`, then `C` must implement all the methods from interface `I`, which means that `C` specifies the body of these methods. In addition, the class `C` can have other methods.

e.g. Java `List` interface

Consider the two classes `ArrayList<T>` and `LinkedList<T>` which are used to implement lists. As such, these two classes share many method signatures. Of course, the underlying implementations of the methods 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 result. This new list should not depend on whether the original list was implemented with a linked list or with an array.

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.

The `ArrayList` class also has other methods that are not part of the `List` interface. For example, the `ensureCapacity(int)` method will expand the underlying array to the number of slots of the input argument if the current array has fewer than that many slots. The `trimToSize( )` method does the opposite. It will shrink the length of the underlying array so that the number of slots is equal to the current number of elements in the list. Note that these two methods make no sense for an `LinkedList`.

There are also methods for the `LinkedList` class that would not be suitable for `ArrayList` class, namely `addFirst` and `removeFirst`. There is nothing special about these operations for array lists. They would be implemented exactly the same as on any other index and they would be expensive because of the shifts necessary. If these operations are commonly needed, then one would tend to use a linked list instead since these operations are inexpensive for linked lists.

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 may not care which. You want to be flexible, so that the code will work regardless of which type is used. In this case, you can use the generic Java interface `List<T>`. For example,

```
void myMethod( List<String> list ){
       :
    list.add("hello");
       :
}
```

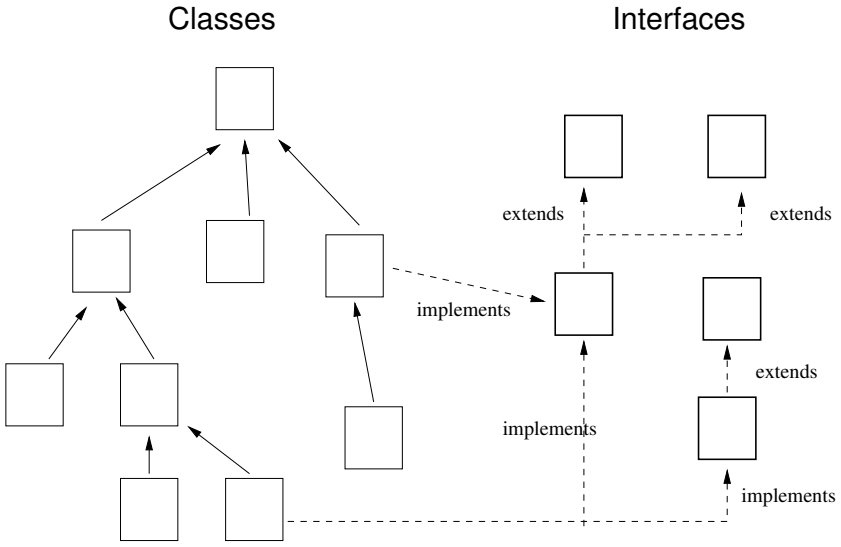
Java allows you to do this. The compiler will see the `List` type in the parameter, and it will infer that the argument passed to this method will be an object of some class that implements the `List` interface. As long as `list` only invokes methods from the `List` interface, the compiler will not complain. You can also do thinks such as the following, namely have the same variable `list` reference different types of lists at different times in the program.

```
List<String>        list;

list  =  new  ArrayList<String>();
list.add("hello");
:
list  =  new  LinkedList<String>();
list.add( new String(hi) );
```

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.

Interfaces and the Java class hierarchy

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. Java classes define a tree, with each node having a reference to its parent i.e. superclass. (Java classes do not have references to their subclasses. e.g. You are allowed to extend the `LinkedList` class. In doing so, you don't change the `LinkedList` class.)



How 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`.

- Each class (other than `Object`) directly extends exactly one other class. Why? 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 might be a 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?

  However, 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`

A Motivating Example for Java Abstract Classes: Circular

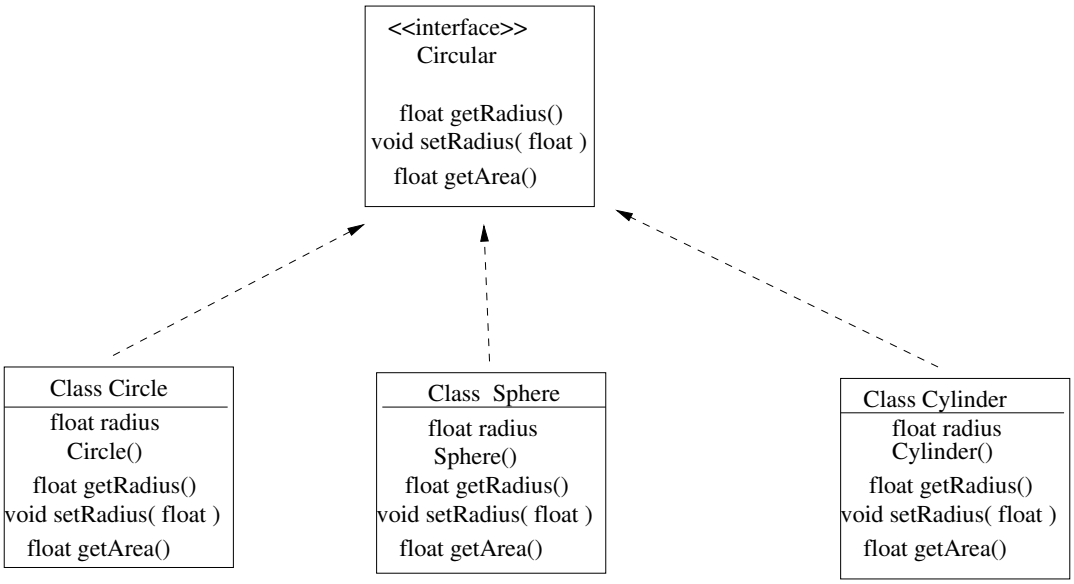
Here is an example to illustrate one of the limitations of interfaces, and motivates the use of abstract classes.

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` since perimeter is defined for 2D shapes, not 3D shapes. 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)

A class can implement more than one interface. However, a class cannot extend two abstract classes. The reason for this policy is the same for why a class cannot extend two classes – namely if the two superclasses were to contain two different versions of a method with the same signature then it wouldn't be clear which of these two methods gets inherited by the subclass.

Finally, 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.

Java `Comparable` interface

Recall that to define a binary search tree, the elements that we are considering must be comparable to each other, meaning that there must be a well-defined ordering. For strings and numbers, there is a natural ordering, but 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`. By definition, any class that implements the interface `Comparable<T>` must have a method `compareTo( T )`.

The `compareTo()` method returns an integer and it is defined as follows. Suppose we have variables

```
T  a1, a2;
```

Then the Java API specifies that `a1.compareTo(a2)` should return:

- a negative integer, if the object referenced by `a1` is "less than" the object referenced by `a2`,
- 0, if `a1.equals(a2)` returns true
- a positive integer, if the object referenced by `a1` is "greater than" the object referenced by `a2`.

By "less than" or "greater than" here, I just mean the desired ordering – what you want if the elements are to be sorted or put in a binary search tree or priority queue, etc.

Example: `Circle`

To get a flavour for this, let's define a Java class `Circle` that implements the `Comparable` interface. Let `Circle` have a fields `radius` and perhaps other fields methods such as mentioned last lecture. How should we implement the `compareTo()` method for the class `Circle`? We could compare circles by the comparing their radii.

```
public class Circle implements Comparable<Circle>{
   private radius;

   public Circle(double radius){    // constructor
        this.radius = radius;
   }

    public boolean  equals(Circle c) {
       return radius == c.getRadius();
    }

    public int  compareTo(Circle c) {
         return radius - c.getRadius();
    }
}
```

These definitions make `compareTo()` consistent with `equals()`.

Example: `Rectangle`

The usage of the `Comparable` interface sometimes can be a bit subtle. To get a flavour for this, let's define a Java class `Rectangle`, which has two fields `height` and `width`, along with getters and setters and a method `getArea()`.

When should two `Rectangle` objects be equal? One intuitive definition of equality is that they should have the same `width` and same `height`. The problem with this definition is that it becomes difficult to say if one rectangle is bigger or smaller than another. If one rectangle has bigger width *and* height, then it is obvious how to order them. But if one rectangle has a bigger width but a smaller height, then it is not obvious how to order them. In this sense, it is not obvious how to define a `compareTo` method for rectangles.

There are options. For example, we could compare rectangles by their area. Or we could compare them by only their height, or only their width. In the code below, we choose to compare by area.

```
public class Rectangle implements Comparable<Rectangle>{  // ignore Shape
   private width;
   private height;

   public Rectangle(double width, double height){    // constructor
      this.width = width;
      this.height = height;
   }

   public double getArea(){
      return width * height;
   }

   public boolean equals(Rectangle other){

     //    return  (this.height ==  other.height) && (this.width == other.width);
     //
     //    Not consistent with the compareTo method below.

      return getArea == other.getArea();
   }

   public int  compareTo(Rectangle r) {
      return getArea() - r.getArea;
   }
}
```

Note that I have used the `equals(Rectangle)` method (overloading) rather than `equals(Object)` method (overriding), and I did the same thing in the `Circle` example. Java recommends not doing this, but rather it recommends overriding rather than overloading. This takes extra work because you need to cast from `Object` to `Rectangle` and there are other subtleties that I won't go into.

`Iterator` interface

We have seen many data structures for representing collections of objects including lists, trees, hashtables, graphs. Often we would like to visit all the objects in a collection. We have seen algorithms for doing this, and implementaions for some data structures such as linked lists.

Because iterating through a collection is so common, Java defines an interface `Iterator<E>` that makes this a bit easier to do.

```
interface Iterator<E>{
   boolean   hasNext();
   E         next();    //  returns the next element and advances
}
```

In fact there are a few different motivations for using iterators. One is that you sometimes are writing methods that do certain things to all of the objects of a collections, but the details of how the collection is organized are just not relevant. For example, you might want to set the value of a particular boolean field to true for each object of the collection. You would like to be to this with one or two lines of code.

A related motivation is that for some classes, like the Java `LinkedList` class, if you iterate through by using the `get(i)` method for all `i`, then this is very inefficient namely it would be $O(n^2)$ instead of $O(n)$. You can use an enhanced for loop for this, although as I'll mention again below, the enhanced for loop in fact is implemented "under the hood" with an `Iterator` object.

Another motivation is that you might want to use several different iterators at the same time. This would be awkward to do with a loop, whether it is an enhanced for loop or some other loop. As an analogy in the real world, consider a collection of quizzes that need to be marked (assuming they were written on paper). 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. Different TAs might be grading different quizzes at any time.

Consider how the `Iterator` interface might be implemented for a singly linked list class. There would need to be a private field (say `cur`) which would be initialized to reference the same node as `head`. The constructor of the iterator would do that initialization. The `hasNext()` method would checks if `cur == null`. The `next()` method advances to the next element in the list. Naturally, you would only call `next()` if `iterator.hasNext()` returns `true`.

[ASIDE: *The method names* `next` *and* `hasNext()` *are admittedly confusing in the context of linked lists where we think of "next" as the node that comes next, rather than the currrent node. You'll simply have to accept that in the context of Iterators the word "next" will be used more like cur. In particular, if the list has just one element then at the initial stage,* `hasNext` *would return true, namely the* `cur` *variable (in my implementation) would reference that element's node.*]

```
private class  SLL_Iterator<E>   implements   Iterator<E>{

    private SNode<E>    cur;

    SLL_Iterator( SLinkedList<E>     list){            //   constructor
        cur = list.getHead();
    }

    public   boolean   hasNext() {
        return (cur != null);
    }

    public   E   next() {
        E    element  =  cur.getElement;
        cur = cur.getNext();
        return element;
    }
}
```

`Iterable` interface

The `SLinkedList` class does not implement the `Iterator` interface. Rather, a `LinkedList` construct objects that implement the `Iterator` interface. How is this done? Iterator objects are constructed by a method `iterator()` which is a method in a interface called `Iterable`.

```
interface Iterable<T>{
     Iterator<T>  iterator();
}
```

This means that the `SLinkedList` has a method called `iterator()` that returns an object whose class type implements the `Iterator` interface. 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) and, because you can do this, you would say that the collection is "iterable". This idea applies not just to linked lists, but to any collection you want to iterate over (trees, graphs, heaps, ...).

The following singly linked list class was given to you way back in the exercises when I first covered singly linked lists. Did you check it out then? If not, then now you have another opportunity! Note in particular the inner private class `SLL_Iterator` which I already described above, and also the `iterator` method which has a very simple implementation.

See slides for illustrations which hopefully will be helpful.

```
class   SLinkedList<E> implements Iterable<E> {

    SNode<E>   head;

    private   class   SNode<E> {
        SNode<E>   next;
        E                   element;
        .
    }

    private   class  SLL_Iterator<E>    implements   Iterator<E>{
        ..   //  see previous page
    }

    SLL_Iterator<E>   iterator()   {
        return  new SLL_Iterator( this );
    } ;
}
```

Think of the `iterator()` method as a "constructor" for `Iterator` objects. Note that `Iterator` is not a class it is an interface. Different `Iterable` classes require quite different iterators, since the underlying data structures are so different.

Java "enhanced for loop"

The "killer app" of the `Iterable` interface is the enhanced for loop. For example, if you have list of `String` objects, you can iterate through them with:

```
for (String   s   :   list)  {
   System.out.println( s );
}
```

The enhanced for loop will work for any class that implements `Iterable` interface. Indeed, this enhanced for loop just gets compiled into equivalent statements that use iterators.

Type Conversion

You should already be familiar with the basics of primitive types and how conversions can occur between them. Primitive types are ordered from "narrow" to "wide".

`byte`, `char`, `short`, `int`, `long`, `float`, `double`

and the number of bytes used by each is

$$1, 2, 2, 4, 8, 4, 8$$

respectively.

Widening conversions occur automatically. Narrowing conversions require an explicit *cast* in the code; otherwise you will get a compiler error. Here are some examples.

```
int    i = 3;
double d = 4.2;

d = i;                   // widening (in assignment)
d = 5.3 * i;             // widening in a binary expression (by "promotion")
i = (int) d;             // narrowing (by casting)
float f = (float) d;     //  "

char   c = 'g';
int    index = c;       // widening
c = (char) index;       // narrowing
```

These type conversions all change the bit representations of the values. In many cases, this leads to an approximation of the value that is represented. You will learn the details of these representations in COMP 273. (See my 273 lecture notes if you are interested.)

We use similar concepts of "narrowing" and "widening" for reference types 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.

Notice 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 the narrower object can be thought of as 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. (Even with primitive types, it is difficult to order by number of bytes since `float` is wider than `long`, yet a `float` has fewer bytes).

Conversions can also occur between reference types. However, *reference type conversions do not change the referenced object.* Rather, the conversion only tells the compiler that you (the programmer) expect or allow the object to be a certain type at runtime. Widening conversions from a subclass to superclass occur automatically. Here we say that we are casting *upwards* (upcasting). Upcasting is sometimes called *implicit casting.* We cast *downwards* ("downcasting") when we are

casting from a superclass to a subclass. Like with primitive types, we need to be explicit when we downcast reference types.

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 for reference types. A few examples are given below.

```
Dog  myDog = new Beagle();  // Upcasting.
:
Poodle  myPoodle = myDog;     // Compiler error.
                            // (implicit downcast Dog to Poodle not   allowed).

myDog.show();                // Gives a compiler error, since show()
                             //  is not defined in Dog class.


Poodle  myPoodle = (Poodle) myDog;   // Allowed (explicit downcast)

myPoodle.show()                   //  runtime error if myPoodle referenced
                            //  a Dog object that has no show() method

((Poodle) myDog).show();  //  Explicit down cast ok: no compiler error.
```

In the last example, if `myDog` references a `Doberman` at runtime, then you get a runtime error since Dobermans aren't show dogs.

Polymorphism (introduction)

We have seen that the declared type of a reference variable does not entirely determine the class of object that the variable can reference at runtime. At runtime, a variable can reference an object of its declared type, or it can also reference an object that is a subtype of the variable's declared type. This property, that the object type can be narrower than the declared type, is called *polymorphism*[18].

There are three separate cases to consider, depending on whether a variable's declared type is a class, an interface, or an abstract class. Suppose a reference variable has a declared type that is a class C. At runtime, that variable can reference any object of class C or any object of a class that extends C. If a variable has a declared type that is an abstract class A, then at runtime that variable can reference any object whose class extends A. If a variable has a declared type that is an interface I, then at runtime that variable can reference any object whose class implements A.

When we discussed type conversion above, we concentrated on the type checking that is done by the compiler. When we discuss polymorphism, we assume a program has compiled fine, and we are concerned with which method is invoked at runtime. The method is determined by the class that the object belongs to. Consider, for example:

[18] from Greek: poly means "many" and "morph" means forms

```
boolean  b;
Object  obj;
   :                       //  some code not specified here
if (b)
   obj = new float[23];   // an array of floats
else
   obj = new Dog();
System.out.print(obj);   //  invokes the toString() method (*)
```

The compiler cannot say for sure which `toString()` method will be invoked since the compiler doesn't know for sure what the value of `b` will be when the `if (b)` condition is evaluated. Rather, the `toString()` method must be determined at runtime, when (*) is executed and the variable `obj` references either a `float[]` or a `Dog`. In each case, there will be a `toString()` method used which is appropriate for the object. (Recall that every class has a `toString()` method.)

To understand more generally how polymorphism works, we need to understand how classes are represented in a running program.

The Class class

When you define a class by typing some ASCII code into a `.java` file, your 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, modifiers such as `public`, and other info. When you compile the class, the compiler makes a `.class` file, which is stored in a directory on your computer. That directory is determined by the package name that you write in the first line of the `.java` file.

When a Java program uses a particular class, it loads this class file and uses the information in this class file to makes an object, which I will call a " class descriptor." This class descriptor is different from a `.class` file generated by a compiler, since a class descriptor is an object in a running program rather than a file. That said, the information in a class descriptor is the same (for our purposes, anyhow) as the the information in a class file. In particular, class descriptors contain the name of the class, an array of fields and types, an array of methods including the instructions of each method, a reference to the superclass, etc.

Class descriptors are objects in a running program, just like the objects that are generated by `new` commands. What class do these class descriptor objects belong to? Answer: the `Class` class.

In lecture 30, I mentioned several methods in the `Object` class, such as `hashCode(), toString(), equals(), clone()`. Another one is `getClass()`. It returns the class descriptor of the class that this object belongs to, that is, the class whose constructor invoked this object. So, the return type of `getClass()` is `Class`. For example, `myDog.getClass()` would return a reference to the `Dog` class descriptor, which is an object of type `Class`.

The `Class` class has several methods. For example, `getSuperClass()` returns the class descriptor of the superclass. The return type is `Class`. So, if `myDog` were a `Beagle`, then `myDog.getClass()` would return the `Beagle` class descriptor, and `myDog.getClass().getSuperClass()` would return the `Dog` class descriptor.

Note that the `Object` class does not have a superclass, so the following returns `null`.

```
Object obj = new Object();
System.out.println(obj.getClass().getSuperclass());
```

Polymorphism (continued)

Last lecture I introduced the idea of a "class descriptor" which is an object of the `Class` class. Today I will continue that discussion and elaborate on the details.

When you run a Java program, you are running the `main` method of some class. A stack frame for the `main` method is put on the call stack. As the instructions in the `main` method are executed, objects may be created. The `main` method may also call other methods. Each time another method is called, a new stack frame goes on the call stack, and when the method returns, the frame of that method gets popped from the call stack. I have discussed this idea several times throughout the course so you should be familiar with it.

Each method including `main` can have local variables. When a method is called, a copy of the local variables for that method are in the stack frame for that method call. I say "a copy" because there may be multiple stack frames for that method, such as in the case of recursion. Each call to a method requires local variables for that call.

The stack frame for a method call also keeps a copy of the parameters that are passed to it. Each stack frame also keeps track of the `this` "variable" which references the object that called the method. (As we will see next lecture, some methods are `static` and aren't called by an object, so the stack frame of these `static` methods don't have a `this`.)

Finally, all of the variables in the call stack can be either primitive types or they can be reference types. In the former case, the values of the variable are written in the stack frame itself. In the latter case, the reference value i.e. `Object.hashCode()` is stored in the stack frame and the object that is referenced is stored somewhere else, namely some "object area" space. All you need to know about this latter space is that:

- each object occupies a consecutive sequence of bytes in that space, and the size of each object depends on the class that this object is an instance of;
- the bytes are numbered and we refer to the numbers as "addresses"; each address is a 32 bit number;
- each object starts at some address and when we use the `==` operator on two objects, we are testing if the starting addresses are the same.

Example

In the slides, I walked through a simple example. Suppose we have a `TestDog` class which has a `main` method with the following instructions:

```
Dog    myDog = new Beagle();
       myDog.bark()
       myDog.getOwner()
```

See the illustrations on the slides of what happens when these instructions are executed. In particular, what happens on the call stack? The first stack frame is `main` and it has a local variable `myDog` that is initialized to `null`. Then the `Beagle` constructor is called which creates a new object. The variable `myDog` references this new object, and the `Beagle` stack frame is popped.

The `main` method continues and the `bark()` method is called by `myDog`, so the `bark` stack frame is pushed on the stack. But which `bark` method is called? Note that the answer must be determined before the `bark` stack frame is put on the call stack. The answer is determined by "asking" the object referenced `myDog`: which class do you belong to. The answer is `Beagle`. One (the JVM) then checks if `Beagle`'s class descriptor contains a `bark()` method; if not, it proceeds to the `Beagle`'s superclass and upwards until the `bark()` method is found. Since the `Beagle` class descriptor does indeed have a `bark()` method, the JVM uses it.

In the next instruction, `myDog` calls `getOwner()` which simply returns a person object.[19] The point here is that this method is in the `Dog` class and so this method is inherited by the `Beagle` class. So, when the method `getOwner` is invoked, the JVM finds this method by first checking the `Beagle` class descriptor (method not found) and then proceeds to the `Dog` class descriptor (method found). A stack frame for the method `getOwner()` can go onto the call stack and the method can be executed.

I mentioned that local variables in a method (including parameters passed to the method) are stored in the stack frame. What about the variables that are fields in the class (such as `owner` or the `name` of a `Dog`). Where are these fields stored?

When a new `Beagle` object is created as in the above example, it has fields that are defined in the `Beagle` class (if there are any) and fields that are inherited from the `Dog` class, such as `owner` and `name`. Since the values of these variables are specific to the object (rather than to the class, the superclass,or particular method that is being run), the values are stored in the Dog object. When I say "stored" here, I just mean that these reference variables are in the object. The objects that they reference such as the `String` object that `name` references or the `Person` object referenced by `owner` would be separate objects. My figures in the slides did not show these objects.

One final note about where things are located: the Java instructions for each method are in the form of compiled "byte code". This code is located in the class descriptor for the class that defines the method. (Although each object has a set of methods that it can invoke, do not think of these methods as part of the object. It would be wasteful to repeat the same instructions in each object – imagine you had thousands of instances !)

Garbage collection – the "mark and sweep" algorithm

It often happens that we create objects and use them, but at some point we no longer need them and indeed we no longer reference them. The example I gave in the lecture was:

```
Dog myDog = new Beagle(Bob);
    myDog = new Terrier(Tim);
```

In this case, there will be no way to access the beagle Bob in the program. We say that Bob the beagle is *garbage*. We would like to be able to reuse the space that this object is taking up and write another object in that space. To do so, we must free up that space and remove the object.

[19] I had mentioned earlier that all dogs have owners and so it makes sense to have a `Person owner` field in the `Dog` class. You can imagine all domestic dogs need a license from the city and the owner has to be listed on the license.)

In the slides, I illustrate that the JVM needs to keep track of all the objects in a program. For simplicity I used a linked list data structure of all the objects, but there is nothing particularly list-like about the object collection and we could use any collection data structure. I use a linked list because it is easy to illustrate.

Garbage collection is done by the JVM when the object space fills up.[20] To remove the objects that are garbage, they must be identified. An object is garbage if there is not way to reference that object from the program. This means that there are no references to the object from variables in the call stack, and there are no references to the object from objects that are referenced from variables in the call stack, and there are no references to the object from objects that are referenced by objects that are referenced by variables in the call stack, etc.

The JVM defines a garbage object in terms of a *graph* whose vertices are one of the following: (1) a variable in the call stack or (2) a variable in an object or (3) an object itself. The edges in the graph are of the form (reference variable, object) where the reference variable vertex can either be in the call stack or in an object. See the slides for an illustration of these edges.

Garbage collection is done by finding all objects that are *not* garbage, that is, objects that are reachable in the graph starting at vertices in the call stack. To find such reachable object, the JVM builds a graph described above and then traverses it from each vertex that corresponds to a reference variable in the call stack. We say that reached objects are *marked* as *live*. The JVM then removes objects that are not marked. This removal is called the *sweep* step. The basic idea for sweep is to just remove the corresonding node(s) in the linked list of live objects. See the illustrations in the slides.

In the slides, I defined a second type of list which keeps track of the gaps between the objects. When an object is removed, it leaves a gap and if that object was adjacent a gap then when the object is removed it increases the size of the gap. As the slides show, theJVM keeps track of a list of the gaps as well as the objects, i.e. two different lists.[21] After garbage collection, when the program resumes running, new object instances can be created and if a big enough gap is found, the new object can be put into the gap.

Garbage collection takes time. If a program is running and needs garbage collection because there is not enough space left for new objects, then the program will need to pause while garbage colletion occurs. To avoid that this disruption takes a long time, garbage collection should be done relatively frequently. (Of course, it has to be done efficiently too. Details omitted for lack of time – and because these "systems" issues are not the main point here.)

[20] As far as I know, the Java language does not specify exactly when garbage collection should be done, so I won't discuss that here

[21] In class, students asked for details on how this is done, such as: does the JVM rearrange the objects in memory so that it has a small number of bigger gaps? As far as I know, there is no specification of this, and it depends on the implementation.

Visibility Modifiers

You should be 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 usually are defined to be `private`, and so someone using the class has to access or modify the information in the fields using getter and setter methods. Later today I will discuss why this is good practice. [Note that in the assignments we often made fields and methods public, some of which normally should be private. This was done to simplify the tester code so that we could have access to these fields and methods.]

The question of `public` vs. `private` gets a bit more complicated when you consider inheritance relationships. Packages come into this too, because they can determine whether classes can extend each other, and whether they can reference and invoke methods from each other. Today I'll go over what visibility modifiers do, and then introduce a few other important modifiers that you may have seen but have not yet learned about.

This lecture has many examples, which I do not reproduce here. You will need to go through the slides, in addition to reading these lecture notes.

Packages

To understand visibility modifiers, we first need to understand what packages are. A *package* is a particular set of classes. Go to the Java API

`http://docs.oracle.com/javase/7/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.

As you know from your assignments, you can make your own packages. For me, the classes `Beagle` and `Dog` belong to package `comp250.lectures`, and so the full name of these classes would be `comp250.lectures.Beagle` and `comp250.lectures.Dog`. My Assignment 3 package is `comp250.assignments2017.a3`

Packages correspond to file directories, and as such packages have a tree structure. The classes within a package are leaves. Packages typically are internal nodes of a directory tree (the only exception being if you have an empty package in which case it would be a leaf).

The `package` directory in the first line of the `.java` file specifies which directory the file should be in. When the `.java` file gets compiled into a `.class` file, the `package` definition specifies the directory where this `.class` file goes. Usually the `.java` file goes in a `src/` directory and `.class` files go in a `bin/` directory. Your computer finds the packages using a CLASSPATH environment variable `https://en.wikipedia.org/wiki/Classpath_(Java)`.

Visibility modifiers for classes

If you want a class `A` to be visible to all other classes, regardless of which package the other classes are in, declare `public class A {...}` . If you want a class `A` to be visible only to classes in the same package, then do not put a visibility modifier in the declaration of the class, i.e. just use `class A {...}` That is, the default is that the class is visible to other classes in teh same package. Note that there is no reserved word `package`.

A somewhat obscure point: 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 `A` to be `private`. This is only way it makes sense to have a private class.

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 be extended only by a class `B` within the same package. If you want a class `B` to extend a class `A` from a different package, then this is only possible if `A` is defined as `public`.

[ASIDE: there is one other visibility modifier – `protected` – which allows `A` to be visible to a class in the same package and also allows `A` to be extended by classes in the same or other packages. I will not discuss the `protected` modifier further in these lecture notes and you don't need to know about 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 other 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. It is 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 `B` 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.

```
_____  void   mB(  A  v ){

      v.mA();    //  or attempt to access field v.fA in class A
}
```

Note that I am not specifying the visiblity of method `mB` here since it is irrelevant.

The left column of the table below shows three possible method visibilitiess for method `mA` in class `A`. The other two columns indicate whether the compiler allows `mB` to invoke method `mA`, as in the code above. A `1` means yes and `0` means no.

mA modifier	A&B same package (B could be a subclass of A)	A and B in diff packages
public	1	1
(package)	1	0
private	0	0

Encapsulation – getters and setters

If you look at various classes the Java API, you hardly ever see class fields listed – even though you know from your experience making your own classes that it is almost impossible to do anything interesting in a class without having fields. The reason that fields are not listed in the Java API for more classes is that the fields are typically private.

To access the information in class fields, one generally uses getter and setter methods. The reason why is best seen for the setter methods. Often there are restrictions in values that the fields can take. A field may have type `String` but the author really doesn't want to allow *any* String. For example, someone's `firstName` should not be allowed to be the String "`9*@!+`", and for a McGill Student ID (say it is `String`), a string such as `150912664` should be allowed, but the string "`1234troll`" should not.

[ASIDE: Getter methods e.g. `getFirstName()` are examples of "accessors", and setter methods `setFirstName(String s)` are examples of "mutators". Accessors retrieve information but don't change the values of any fields. Mutators change the values of fields, but don't return anything. You can define methods that are both accessors and mutators e.g. `HashTable.put(K,V)` .]

Why don't method variables have a visibility modifier ?

Often methods have local variables. These are not given a visibility modifier. Why not? Methods themselves have a visibility, and so do the classes in which they are defined. However, the variables that are defined within a method are not, and the reason is that these variable (names) have no meaning in code outside the method. So it is impossible for another method in another class to use these variabl names.

While we are talking about these variables, remember where they are in the running program. They are in the stack frame of the method which is on the call stack. See illustration in the slides.

"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, so let's just deal with static methods and variables here.) Static methods and fields are often used, for example, to keep track of the number of instances of a class.

```
static  int  numberOfDogs = 0;

static  int  getNumberOfDogs(){        //  this variable.
    return numberOfDogs;
}

Dog(){
   numberOfDogs ++;
}
```

Static fields are not stored in objects (instances) of the class. Rather, static fields are stored in the class descriptor for that class, which is an object of type `Class`. One can have make static variables in a class. For example, if Dog objects were organized into a monarchy, there could be a `queenDog` and `kingDog` which would be static variables in the `Dog` class.

Another example of a `static` method is `main()`. As an aside, note that you invoke this method when you "run the class" and you do this without instantiating the class. How exactly this works in the JVM is outside the scope of the present discussion.

A `static` method or field is often called a "class method" or "class 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 )`. 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.

  The example I gave in class is that when I was child and teenager I had a beagle named "Buddy". After he passed away, I swore I would never get another dog. It was as if I declared:

  ```
  final Dog  myDog = new Beagle(``Buddy'');
  ```

  But life is not a computer program. Many years later, my wife and kids overruled me, so then:

  ```
  myDog = new Poodle(``Willie'');
  ```

  In a computer program, this instruction would have yielded a compiler (not runtime) error, since the `final` modifier for `myDog` only allows one assignment for that variable.
- `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.

[ASIDE: Please see the lecture slides for examples of what was discussed today. For this lecture, in particular, I would ignore the lecture recordings as some of the slides were messed up, and others were redundant. I cleaned up the slides from Sec. 001 to 002, and then made a few small further changes before posting them to the public web page.]