Data Structures and Algorithms

A Simple Introduction

Table of Contents

Read this first

This is a short and simple introduction to the ideas of data structure and algorithm. These are important ideas, because they are the basis for all programming.

What is an algorithm?

An algorithm is a way of doing something. It is a method, or recipe, or set of instructions to follow to solve some problem.

We start with a very simple example. Suppose we have two pieces of data, x and y, and we need to exchange their values. As a test case:

before: x=3 and y=7

after: x=7 and y=3

This needs to work with any values. Whatever x and y are before, after they must have the values changed over.

How to do this? What algorithm to use?

We might say:

x ← y // this means copy the value of y to x

y ← x

Does this work? We can check using a trace table. We work this out on paper, keeping track of the values of the variables at each step, like this..

	x	y	
Before	3	7	start values
after x ← y	7	7	the 7 was copied to x
after y ← x	7	7	the 7 was copied back to y
At the end	7	7	

So, this does not work. When we copied 7 to x, we lost the initial value of x.

One way which does work is to use an extra storage space, which we will call 'temp', because it is only for temporary use:

temp ← x

x ← y

y ← temp

We check this with a trace table:

	x	y	temp	
Before	3	7	??	It makes no difference what temp is
after temp ← x	3	7	3	the 3 was copied to temp
after x ← y	7	7	3	the 7 was copied to x
after y ← temp	7	3	3	temp (start value of x) copied to y

At the end	7	3		Values exchanged

So this algorithm will exchange two data values.

Algorithm steps

So an algorithm is a sequence of steps, each step being an 'instruction'. These instructions become instructions to the computer in the program, so we can only have very simple steps - only the following are allowed:

1. Moving a value from one location to another, or a constant. So for example x ← y or x ← 3
2. Doing arithmetic. For example x ← 2*3-5 (the * means multiply)
3. Decisions based on simple logic . For example if x > 10 then x ← 2*3-5
4. Loops. For example do 10 times x ← x+1
5. Input and output.

The input might come from the keyboard, or from a file of data. The output might be displayed on a screen, or on a printer, or written out to a file.

We cannot use instructions which require intelligence, like 'solve $x^2+3x-4=0$'. If we say something like 'test if x is a prime number' that in fact is using another algorithm, and we should give that algorithm, to say *how* to test if a number is prime.

We can only have a finite number of instructions, and the algorithm must end after a finite number of steps, with the correct result.

We write down algorithms in pseudo-code. 'pseudo' is pronounced 'sudo' - the p is silent. It means 'like, but not real'. So pseudo-code is like real program code, but not actual program code. We use pseudo-code because

1. It is easier to read and understand than actual code
2. It is less strict than code
3. It can be converted to any programming language

Algorithms and programs

Computers cannot run pseudo-code algorithms. They can only run actual programs, written in some programming language.

Algorithms should be language-agnostic. That means, they should not depend on any particular programming language. They will work in any language.

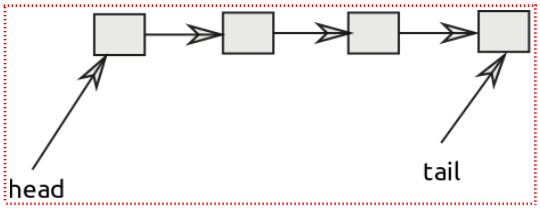
We will use Java and C to show algorithms written as actual programs. But you should focus on the actual algorithms and data structures, not the code.

Data structures

A data structure is a way of organising a set of data values. We will use diagrams to show them.

For example, a list:

Here the boxes represent data nodes, and the arrows are pointers.



Data nodes

A node is one of the items n a data structure. They might be any type - for example integers, for decimal numbers, or characters or strings, date or time, pixel colour or whatever.

Usually all the nodes in a structure and the *same* type, whatever that is.

Often the nodes are key-value pairs, in records (like a struct in C). For example in a personnel application we might have nodes which are for example

1123	John	Smith	15.1.1990	Sales
Payroll number	First name	Last Name	DateOfBirth	Department
Key	Values			

So each node has a unique key field - everybody has a different key field. We will often search a data structure for the node with a key field we want.

In examples, we will usually take a node to have a key which is an integer, and forget about value fields. This is just to make things simple. Using nodes with keys and values of different types does not alter the algorithm or the structures.

Usually a node will also have one or more fields which are pointers, to link the node to other nodes, to make a structure.

Pointers

We show pointers in a data structure as actual arrows which point at other nodes. A pointer is simply a way of 'getting to' another node. How pointers are implemented is different in different languages.

In C, a pointer is an actual address in memory.

In Java a pointer is a reference. This is not an actual memory address. It cannot be, since Java objects move around in memory as the program executes. But still, if we have a reference, we can 'get to' the object pointed at.

Some pointers are variables with names. For example in the list, we have a pointer named 'head', and this points to the first node in the list. The simplest way to think of this is that the value of 'head' is the address of the first node in memory.

We often use the idea of a null pointer. This is a pointer which points 'nowhere'. For example the last node in a list has a null pointer to the next node - because there is no next node. Often the null pointer value will be something which could not be the address of a real node - such as -1.

Time and Space Complexity

We often have a choice of using different algorithms. How to decide which is better? Remember all algorithms must 'work' - that is, end with the desired result. Otherwise they do not count as an algorithm.

An algorithm is better if

1. it is faster
2. it uses less memory

For a small number of data items, any algorithm is OK. It is only for a very large amount of data does speed and memory use matter.

The time complexity of an algorithm is how many steps it takes to finish, depending on the number of data items - normally called n. So we need to know how many steps, for very large n. Obviously any algorithm will be faster on a faster computer. So we ignore the actual time, and just count the steps. That way we can compare algorithms on different computers.

The space complexity is how much memory it uses. Some algorithms just move data around in the data structure. These work 'in place' and use no extra memory beyond what the data itself needs. Other algorithms move the data into a different structure - these use twice the memory of the initial data structure.

Usually we can trade off speed and memory, so that an algorithm which uses more memory is faster.

Example - random numbers

In many areas of computing it is useful to have random numbers. Can we find an algorithm to generate a sequence of random numbers? One way is to take some starting number and multiply by something to get the next one. For example

rand ← rand * 121

This has the problem that rand will get bigger and bigger. We can fix that by taking the modulus (the remainder):

rand ← (rand * 121) mod 1000

so then if rand * 121 is, say 1234, the modulus 1000 is just 234. We will stay in the range 0 to 999.

Here is an algorithm to make 100 numbers like this:

rand ← 1

counter ←1

while counter<100

  rand←(rand*121) mod 1000

  counter←counter+1

end loop

Here is program in C to do this:

```
#include <stdio.h>

int main(int argc, char** argv) {
    int rand, counter;
    rand = 1;
    counter = 1;
    while (counter < 100) {
        rand = (rand * 121) % 1000; // % is modulus in C
        printf("%d\n", rand);      // output rand
        counter++;
    }

    return (0);
}
```

The output starts..

```
121
641
561
881
601
721
241
161
481
201
321
841
```

```
761
81
801
```

At first that looks pretty random. But then we see the number always ends in 1. And if we look further down the output we see..

```
961
281
1
121
641
561
881
```

so we get 1, then 121, where we started, then the sequence repeats. These are not really random numbers.

In fact *no algorithm can make truly random numbers* on normal digital devices. Instead we use pseudo-random numbers. These are very mixed up and will go through an extremely long sequence before they repeat. But sooner or later they will repeat.

In our simple algorithm we started rand as

rand ← 1

This is called the seed - the first value in the sequence.

Both C and Java have standard library code to create pseudo-random numbers in a much better way (using a different algorithm). In C the function is rand(), which produces a number from 0 to RAND_MAX, which is at least 32767. This is seeded by the function srand(). The usual way to use this is using the time() function, so the sequence is seeded by the current computer time. This means we will get a different sequence every time the program runs..

```
 #include <stdio.h>
#include <stdlib.h> // needed for rand()
#include <time.h>   // needed for time())

int main(int argc, char** argv) {
    int counter;
    srand(time(0)); // seed with current time
    counter = 1;
    while (counter < 10) {
        printf("%d\n", rand()); // call rand and output it
        counter++;
    }

    return (0);
}
```

One run produces..

```
736868008
841317755
3139694
1449466950
1445477844
2003343631
1022628446
1119168331
1005563998
```

but we would get a different sequence on every run.

In Java a class named Random models a pseudo-random number generator:

```
import java.util.Random;

public class Test {
```

```
  public static void main(String[] args) {
  Random rng=new Random(); // construct a random number generator
  for (int counter=0; counter<9; counter++) // 10 times
  {
    int number=rng.nextInt(1000000); // get next number, in range 0 to one million
    System.out.println(number); // output it
    }

  }
}
```

Output:

```
534576
220608
335031
232252
713572
225434
946502
159104
268880
```

The seed here is very likely to be different on every run.

The idea of random numbers is discussed in Donald Knuth's *The Art of Computer Programming Volume 2*. This text is the foundation of Computer Science, and all should read it.

Recursion

Recursion is when an algorithm uses itself.

An example is factorial. The factorial of a number is the product of all the numbers down to 1. So the factorial of 6 = 6! = 6 X 5 X 4 X 3 X 2 X 1 = 720

We can calculate it like this:

if n=1, n!=1

else n! = n X (n-1)!

In C..

```
 #include <stdio.h>

int fact(int n)
{
    printf("In fact with n = %d\n", n); // to follow the process
    if (n==1) return 1;
    return n*fact(n-1);
}

int main(int argc, char** argv) {

    printf("%d\n", fact(6));

    return (0);
}
```

Output:

```
In fact with n = 6
In fact with n = 5
In fact with n = 4
In fact with n = 3
```

```
In fact with n = 2
In fact with n = 1
720
```

This is recursive, because the definition of the fact function includes a call to fact.

There is also an if which leads to an immediate return - that is

```
    if (n==1) return 1;
```

Recursive code will always have a 'get out' like this. Otherwise we have an infinite recursion and it will fail.

We could also do this using iteration in place of recursion:

```
 int fact(int n)
{
    int result=1;
    for (int i=n; i>1; i--)
        result*=i;
    return result;
}
```

It can be proved that any recursive code can be converted to iterative code.

A second example is the Fibonacci sequence:

1 1 2 3 5 8 13 21..

We start with 1 and 1, then the next number is the sum of the last two.

We can have a recursive definition of the nth Fibonacci number..
if n=1 or 2, Fib(n)=1
else Fib(n) = Fib(n-1) + Fib(n-2)

In C:

```
#include <stdio.h>

int fib(int n)
{
    printf("In fib with n= %d\n",n);
    if (n==1 || n==2) return 1;

    return fib(n-1) + fib(n-2);
}

int main(int argc, char** argv) {

    printf("%d\n", fib(5));

    return (0);
}
```

Output is

```
In fib with n= 5
In fib with n= 4
In fib with n= 3
In fib with n= 2
In fib with n= 1
In fib with n= 2
In fib with n= 3
In fib with n= 2
In fib with n= 1
5
```

How to explain the output?

This calls Fib(5)

which calls Fib(4)

which calls Fib(3)

which calls Fib(2) returns 1

and Fib(1) returns 1

and Fib(2) returns 1

and Fib(3)

which calls Fib(2) returns 1

and Fib(1) returns 1

Recursion is sometimes slightly tricky.

Basic array algorithms

An array is a basic data structure in which we can access a node using an index.

In pseudo-code we will show array elements with the index in square brackets.

So

values[328] ← 2398

means we have a array named values, and the assignment writes 2398 into the node with index 328. So an array is just a set of numbered boxes. We will have arrays that start with box number 0, as in Java and C and many languages.

In most languages array elements are stored next to each other in memory, and boxes are found using a simple calculation. For example in C:

```
int array[100]; // declare and create array of 100 ints
int * ptr;  // ptr is a pointer to an int
ptr=array; // ptr points to start of array
array[10]=37; // put value in the array in box 10
printf("%d\n", array[10] ); // 37 - fetch array box
printf("%d\n", *(array+10) ); // 37 - treat array as pointer to start of block
printf("%d\n", *(ptr+10) ); // 37 - use ptr in same way
printf("%p\n", ptr); // address of start of array = 0x7ffe652cb520
printf("%p\n", ptr+1); //  next int location = 0x7ffe652cb524
printf("%ld\n", sizeof(int)); // size of int = 4 bytes
```

we can access the element with index 10 as array[10]. We can also get it as *(ptr+10), when we have said ptr=array, so that ptr points to the start of the array. The value of ptr is 0x7ffe652cb520, which is the address in RAM (in hex) of where the array starts.

See that ptr+1 is 0x7ffe652cb524, not 0x7ffe652cb521. This is because we have declared ptr to be a *pointer to an int*, so ptr+1 is the address of the *next int*, not the next byte. Ints here are 4 bytes long, so it adds 4 to the address, not 1.

So it finds array[10] by adding 40 to the address of start of the array. In general

address of nth element = address of start of array + n*(size of an element in bytes)

Total of array elements

We use a running total, which we initialize to zero. Then we iterate through the array, adding in each element. For example for an array of size 6:

total ← 0

index ←0

while index<6

  total ← total+array[index]

  index ← index+1

So in Java -

```
int[] array = {1, 5, 2, 3, 4};
int total = 0;
int index = 0;
while (index < 5) {
  total += array[index];
  index++;
}
System.out.println(total); // 15
```

We get 1+5+2+3+4 = 15

Count even elements

If number is even, number%2 is zero - since % (modulus) is remainder after division. So in Java:

```
int[] array = {1, 5, 2, 3, 4};
int evenCount = 0;
int index = 0;
while (index < 5) {
  if (array[index]%2==0)
    evenCount++;
  index++;
}
System.out.println(evenCount); // 2
```

Find maximal values

Maximal means largest and smallest.

We start by finding the largest.

We use a variable named 'biggestSoFar'. We initialise this to the first array element. Then we iterate through the rest of the array, and if we find an element larger than biggestSoFar, we change biggestSoFar to that value.

biggestSoFar ← array[0]

index ←1

while index<6

if array[index]>biggestSoFar

biggestSoFar ← array[index]

index ← index+1

So in C:

```
int array[5] = {1, 5, 2, 3, 4};
int biggestSoFar = array[0];
int index = 1;
while (index < 5) {
    if (array[index] > biggestSoFar)
        biggestSoFar = array[index];
    index++;
}
printf("%d\n", biggestSoFar); // 5
```

To find the smallest value, we just have something like

```
 if (array[index] < smallest)
    smallest = array[index];
```

Reverse an array

For example to change

{9,4,8,1,3}

into

{3,1,8,4,9}

We could create an array of the same size as the first, then copy the first into the last element of the new array, the second into one from last, and so on.

But this uses extra memory, and we can do it 'in-place' using only one extra location.

We iterate half way through the array, exchanging each element with the one a corresponding distance from the end.

In Java..

```
    int[] array = {9,4,8,1,3};
    // iterate half-way through
    for (int index=0; index<array.length/2; index++)
    { // do exchange
      int t=array[index];
      array[index]=array[array.length-1-index];
      array[array.length-1-index]=t;
    }
    for(int val:array) // test
      System.out.println(val);
```

Here we use array.length, so this works for any size array.

We also use the 'for-each' loop in for(int val:array)

Key and value nodes

There seems little point to search for an int in an array. If we know what the int is - why look for it?

Usually the data we handle has some identifying unique key, and other data fields. We look for some key, to read back the data fields linked to it.

In Java, we would define a Node class like this:

```
class Node
{
  final int ID;
  String data;

  Node(int id, String val)
  {
    this.ID=id;
    this.data=val;
  }

  public String toString()
  {
    return "Node ID="+ID+" data="+data;
  }
}
```

This has just one value field, named 'data', but we could have several. The ID field is an int. It does not have to be an int. The ID identifies the node, and it would not make sense to change it, so we declare it to be final. We could create two nodes with the same ID, which would be wrong. We could make this impossible - but we do not, just to keep it simple.

Then we can modify our array reverse code to use an array of Nodes:

```
     Node[] array = new Node[5];
    array[0]=new Node(9,"one");
    array[1]=new Node(4,"two");
    array[2]=new Node(8,"three");
    array[3]=new Node(1,"four");
    array[4]=new Node(3,"five");
    // iterate half-way through
    for (int index=0; index<array.length/2; index++)
    { // do exchange
      Node t=array[index];
      array[index]=array[array.length-1-index];
      array[array.length-1-index]=t;
    }
```

```
    for(Node val:array) // test
      System.out.println(val);
```

The output is:

```
Node ID=3 data=five
Node ID=1 data=four
Node ID=8 data=three
Node ID=4 data=two
Node ID=9 data=one
```

Searching

This is about searching some data, in an array, for some given key value.

Iteration

Iteration means looping or repeating. We often iterate through an array. For example, suppose we want to store a value, 999, in every element of an array which has 100 elements - here is the pseudo-code:

```
index ← 0                       // initialise index (start value)
while index <100          // start loop
  values[index] ← 999     // write 999 into array box
  index ← index + 1             // move to next box
end loop                        // stop when get to last box
```

The // starts a comment - an explanation of what is happening

Here is that algorithm as a Java program:

```
public class Test {

  public static void main(String[] args) {
    int index;    // declare index as an integer
    int[] values = new int[100]; // declare values as an array of 100 ints, and make the array
    index = 0; // initialise
    while (index < 100) { // loop
      values[index] = 999;
      index++; // same as index=index+1
    }

  }
}
```

The same algorithm in C is very similar:

```
int main(int argc, char** argv) {
    int index;
    int values[100];
    while (index<100)
    {
        values[index]=999;
        index++;
    }

    return (0);
}
```

Linear Search

Suppose we have an array of integers, and we need to find some given value. There is the possibility it is not in the array. How to do this? What algorithm to use?

The obvious way is to start with the first element of the array. If it is equal to the target, we have finished. If not we move on to the second element and do the same. We continue until we find it, or we reach the end of the array and it is not present. In pseudo-code, searching an array with 100 elements:

```
index ← 0
```

while index<100

if array[index] = target

end - value found at index

index ← index+1

endloop

end - value not present

This is a linear search. *Linear means in a line*.

In C:

```
#include <stdio.h>
#include <stdlib.h> // needed for rand()
#include <time.h>   // needed for time())

int main(int argc, char** argv) {
    int array[100];
    // fill array with random integers 0 to 99
    int counter;
    srand(time(0));
    counter = 0;
    while (counter < 100) {
        array[counter] = rand() % 100;
        printf("%d: %d\n", counter, array[counter]); // for testing
        counter++;
    }
    // do linear search for target 56 (for example))
    int target = 56;
    counter = 0;
    while (counter < 100) {
        if (array[counter] == target) {
            printf("Found at %d\n", counter);
            return 0; // exit
        }
        counter++;
    }
    printf("Not found\n");
    return (0);
}
```

Sample output:

```
0: 40
1: 49
2: 18
3: 73
..

45: 66
46: 97
47: 77
48: 12
49: 56
50: 82
51: 4
52: 14
..
98: 12
99: 82
```

```
Found at 49
```

The same in Java:

```
import java.util.Random;

public class Test {

  public static void main(String[] args) {
    int[] array = new int[100];
    // fill array at random
    Random rng = new Random();
    for (int counter = 0; counter < 100; counter++) {
      array[counter] = rng.nextInt(100);
      System.out.println(counter + " " + array[counter]); // for testing
    }

    int target = 56; // search for 56
    for (int counter = 0; counter < 100; counter++) {
      if (array[counter] == target) {
        System.out.println("Found at" + counter);
        return;
      }
    }
    System.out.println("Not present");
  }
}
```

Time complexity of linear search

We can measure how long a program takes. For example in Java:

```
import java.util.Random;

public class Test {

  public static void main(String[] args) {
    final int N=10000000;
    int[] array = new int[N];
    // fill array at random
   ..

    int target = 1234567; // search
    long startTime=System.nanoTime();
    for (int counter = 0; counter < N; counter++) {
      if (array[counter] == target) {
        System.out.println("Found at" + counter);
        System.out.println( (System.nanoTime()-startTime)/1000000 + " milliseconds");
        return;
      }
    }
    System.out.println("Not present");
    System.out.println( (System.nanoTime()-startTime)/1000000 + " milliseconds");
  }
}
```

We are using an array of 10 million values, and calling the number of values N. System.nanoTime() gives the current system time, in nanoseconds. We remember the start time, and get the time again when we find out, or end not finding it. Typical output is:

```
Found at 7110511
17 milliseconds
```

But actual speed is not very useful. The same algorithm on a faster computer would be faster. And if the computer is doing other tasks at the same time (which it will be), this will slow things down.

A better way is to count the number of steps taken - because this is the same on slow or fast computers. In this algorithm there are 2 kinds of steps - the if statement to compare the array to the target, and the addition to move the index counter on. We can count the steps - in C, using N=1000000 :

```
#include <stdio.h>
#include <stdlib.h> // needed for rand()
#include <time.h>   // needed for time())

int main(int argc, char** argv) {
    int N=1000000;
    int array[N];
    ..
    // do linear search
    int target = 123456;
    int steps=0;
    counter = 0;
    while (counter < N) {
        steps++; // the if step
        if (array[counter] == target) {
            printf("Found after %d steps\n", steps);
            return 0;
        }
        counter++;
        steps++; // the counter step
    }
    printf("Not found after %d steps\n", steps);
    return (0);
}
```

Typical output is

```
Found after 1030845 steps
```

How many steps do we expect? We might be very lucky, and find the target in the first array element. Or, if the target is not in the array, we will need to check every element, and this will mean 2n steps, for n data values. On average we will find it half way through, and need n steps. So we have

best case - 1 step

worst case - 2n steps

on average - n steps

We will study this more later.

Searching sorted data

Suppose the data in the array was sorted in increasing order. Could we find a faster algorithm then?

The first stage is to have an array of data in order. We can do this using random values as follows, to fill an array of 100:

index ← 0

value ← random

while index<100

array[index] ← value

value ← value + random

index ← index+1

endloop

This uses 'random to mean some random value. We simply add a random amount to the last array element to get the next - so they will be in increasing order. Here is code in Java:

```
final int N=10;
int[] array = new int[N];

Random rng = new Random();
int value;
value=0;
for (int counter = 0; counter < N; counter++) {
  value+=rng.nextInt(5);
  array[counter] = value;
  System.out.println(counter + " " + array[counter]); // for testing
}
```

The random value might be 0, so we might have 2 elements next to each other which are equal.

Binary search

If the data is sorted, we can use the binary search algorithm.

We first compare the middle number with the target

If equal, we have found it.

If it is too small, we look in the 'top half'

If it is too big, we look in the bottom half

This is called a binary search because binary means two, and the data is divided into two halves. It is also called a bisection search. Bisection means cut into two.

So we have two pointers, for the top and bottom of a section. To start with this is the whole array. After the first step, it is the top half or the bottom half. And so on until we find it, or the section gets down to size one.

The algorithm to search an array of size n:

bottom ← 0 // set top and bottom pointers

top ← N-1 // whole array to start with

do

middle ← (top+bottom)/2 // find middle of section

if array[middle]= target // found it?

found it - end

if array[middle]> target // middle too big

top←middle // use bottom half

else // middle must be too small

bottom←middle // use top half

while top not equal to bottom+1// until section size 1

not found

Here's the code in Java:

```
     int target = 15; // search
    int bottom=0;
    int top=N-1;
    do
    {
      int middle =(top+bottom)/2;
      System.out.println("Bottom="+bottom+" middle="+middle+" top="+top);
      if (array[middle]== target)
      {
        System.out.println("Found at "+middle);
        return;
      }
      if (array[middle]>target)
        top=middle;
      else
        bottom=middle;
    } while (top!=bottom+1);
    System.out.println("Not found");
```

and a typical run, with the array listed first (looking for 15):

```
0 0
1 4
2 7
3 7
4 11
5 11
6 13
7 15
8 15
9 17
Bottom=0 middle=4 top=9 // first look at 4 - too small
Bottom=4 middle=6 top=9 // top half - look at 6 too small
Bottom=6 middle=7 top=9 // top quarter - found at 7
Found at 7
```

Time complexity of binary search

If we modify the program to use 1,000,000 data items, and count the loops, we get on a sample run:

```
Found at 149316
18 steps
```

by comparison, a linear search on average would take 500,000 loops.

If we are searching 1,000,000, the first check, top or bottom, means we reject 500,000 elements. The next step means we reject another 250,000. We work it out

Step	Elements left
1	500,000
2	250,000
3	125,000
4	62,500

5	31,250
6	15,625
7	7,812
8	3,906
9	1,953
10	876
11	438
12	229
13	114
14	57
15	28
16	14
17	7
18	3
19	1

so just after 19 steps, the section size reduces to 1. We might find it before that, but in the worst case it will take just 19 steps.

So the best case is 1 step, and the worst is *how many times n can be divided by 2*. This is what $\log_2 n$ is - log to base 2.

If n is large, binary search is very much faster than a linear search. But it only works if the data is sorted.

Recursive binary search

In a binary search we divide an array section into two, look at the middle element, and if its not there.. the process happens again, either on the lower or the upper half. So we can do this recursively.

We set up the array in main..

```
  public static void main(String[] args) {
    final int N = 10;
    int[] array = new int[N];

    Random rng = new Random();
    int value;
    value = 0;
    for (int counter = 0; counter < N; counter++) {
      value += rng.nextInt(5);
      array[counter] = value;
      System.out.println(counter+" "+value);
    }
    binSearch(array, 15, 0, N - 1);
  }
```

then the recursive bisection search is:

```
public static void binSearch(int[] array, int target, int bottom, int top) {

    if (top == bottom + 1) {
      System.out.println("Not found");
      return;
```

```
    }

    int middle = (top + bottom) / 2;
    System.out.println("Bottom=" + bottom + " middle=" + middle + " top=" + top);
    if (array[middle] == target) {
      System.out.println("Found at " + middle);
      return;
    }
    if (array[middle] > target) {
      top = middle;
    } else {
      bottom = middle;
    }
    binSearch(array, target, bottom, top);
  }
```

It is recursive because binSearch calls binSearch.

We have two 'get-out' ifs - if top=bottom+1, and its not found - or if it is found. So we do not get an infinite recursion.

Sorting

Suppose we have a set of numbers such as

7, 3, 8, 4, 2

and we re-arrange them into increasing order:

2, 3, 4, 7, 8

This is called 'sorting' - putting data in order of the key field.

Why do it?

Sort algorithms are very important, and a lot of computer time is spent doing it. Why?

Suppose you have a list of files in a folder, sorted into alphabetical order:

```
BST.c
build
dist
docs
DSA.h
dynArray.c
heap.c
linkedList.c
main.c
Makefile
nbproject
queue.c
stack.c
text.odt
```

Suppose I am looking for a file bugs.doc. I can read from the top, and as soon as I see dist, I know the file is not there. If the list was not sorted, I would have to search to the end, which would be slower.

Another example is about databases, with an index on a table. We might have a query like

SELECT * FROM invoices WHERE invoice_id =32756

This searches the table 'invoices' for a row with ID 32756. The ID should be a key, with an index in order, and this makes doing the search much faster, since it can use a binary search not a linear one.

Bubble sort

The bubble sort algorithm is usually the first to be taught. We introduce it in two stages.

Suppose we go through the array, compare each value with the next, and exchange hem if they are the wrong way round. What effect does that have? We try it in a trace table:

Initial numbers	7	3	8	4	2
7 and 3 swap	3	7	8	4	2
7 and 8 no swap	3	7	8	4	2
8 and 4 swap	3	7	4	8	2
8 and 2 swap	3	7	4	2	8

This has moved 8, the largest value, to the end. But it has not sorted them. The second largest, 7, is left at position 2.

Do it again:

Initial numbers	3	7	4	2	8
7 and 3 no swap	3	7	4	2	8
7 and 4 swap	3	4	7	2	8
7 and 2 swap	3	4	2	7	8
7 and 8 no swap	3	4	2	7	8

A second scan has moved the second number, 7, to its correct final position.

Each scan sweeps one more number to its correct position. So for five numbers, we need to do this five times.

In C:

```
int main(int argc, char** argv) {
    int array[5]={3,7,4,2,8}; // initial array
    for (int repeats=0; repeats<5; repeats++) // do it 5 times
        for (int index=0; index<4; index++) // scan through from
                                            // 0 to 1 from the end
            if (array[index]>array[index+1]) // compare number with next
            {
                int temp=array[index]; // exchange
                array[index]=array[index+1];
                array[index+1]=temp;
            }
    for (int index=0; index<5; index++) // output to test
        printf("%d\n",array[index]);

    return (0);
}
```

Output is

```
2
3
4
7
8
```

Time complexity of a bubble sort

Suppose there are n numbers in the array.

In each scan, we make (n-1) comparisons, and possible a swap for each one.

We make n scans.

So the total steps (comparison and possible swap) is n(n-1)

Big O notation

The important thing is how fast an algorithm is on large numbers of numbers.

Any algorithm will sort 10 or 20 numbers in a few milliseconds. It is not until we want to sort large numbers - thousands - that different algorithms have different speeds.

So we want to have the number of steps as a function of n, and

1. We want the asymptotic behaviour of this - the function for large n

2. We ignore any constant multiplier.

This is written as O[n] - so big O notation.

For the bubble sort we have worked out the step count as $n(n-1) = n^2-n$

Suppose n is 1000. Then $n^2 = 1000000$, and the step count is

1000000 - 1000

If n is large, n^2 is much bigger than n. The asymptotic value of n^2-n is just n^2.

So for a bubble sort, O[n] is n^2

Practical testing

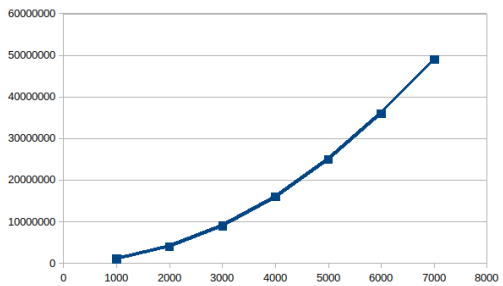
Is this correct? Can we actually check the number of steps, and compare it with n?

We can run this program repeatedly, with different values of n

```
    int N = 7000;
    int array[N];
    srand(time(0));
    int steps = 0;
    for (int repeats = 0; repeats < N; repeats++) // do it N times
        for (int index = 0; index < N - 1; index++) // scan through from
        {
            steps++;
            // 0 to 1 from the end
            if (array[index] > array[index + 1]) // compare number with next
            {
                int temp = array[index]; // exchange
                array[index] = array[index + 1];
                array[index + 1] = temp;
            }
        }

        printf("%d\n", steps);
```

If we do this and record the step count for different n we get:

n	steps
1000	999000
2000	3998000
3000	8997000
4000	15996000
5000	24995000
6000	35994000
7000	48993000



So bubble sort is $O[n^2]$

This is not a fast sorting method. To sort twice as many elements takes 4 times as long. Ten times as many elements takes one hundred times as long.

Improved bubble sort

After the first scan, the largest element is in its final place at the send.

After the second scan, the second largest is in its final place one from the end.

So the elements at the end are already in place, and the scan can reduce in length as we repeat it.

Also, suppose before the end all the values are sorted. The algorithm will continue even though no swaps are taking place. We can check - if no swaps happen after a scan, stop:

```
    int N = 7000;
    int array[N];
    srand(time(0));
    for (int index = 0; index < N; index++)
        array[index] = rand() % N;

    int steps = 0;
    for (int repeats = 0; repeats < N; repeats++) // repeat scans
    {
        int swapCount = 0;
        for (int index = 0; index < N - 1 - repeats; index++) // scan end reduces
        {
            steps++;
            // 0 to 1 from the end
            if (array[index] > array[index + 1]) // compare number with next
            {
                swapCount++; // counting swaps
                int temp = array[index];
                array[index] = array[index + 1];
                array[index + 1] = temp;
            }
        }
        if (swapCount == 0) break; // if no swap, end
    }
```

The speed increase depends on the random data, but in a run of n=7000, we get 24492930 steps, compared with 48993000 for the original bubble sort.

Bogosort

The bogosort algorithm is

repeat

swap random pairs of array elements

until array is sorted

Here is a version in C:

```
int N = 13; // global data
int array[13];

void randomSwaps() {
    int a, b; // select 2 elements at random
    a = rand() % N;
    b = rand() % N;
    int temp = array[a]; // and swap them
    array[a] = array[b];
    array[b] = temp;
    return;
}

int notSorted() // is it sorted yet?
{
    for (int index=0; index<N-1; index++)
        if ( array[index]>array[index+1] )
            return 1; // out of order : not sorted
    return 0;
```

```
}

int main(int argc, char** argv) {

    srand(time(0)); // fill array at random
    for (int index = 0; index < N; index++)
        array[index] = rand() % N;

    int steps=0;
    do { // bogosort
        randomSwaps();
        steps++;
    } while (notSorted());

    for (int index=0; index<N; index++) // check
        printf("%d\n", array[index]);
    printf("Steps = %d\n", steps);

    return (0);
}
```

Typical output is

```
1
1
3
3
4
4
4
5
7
10
11
11
12
Steps = 300322848
```

How fast is bogosort?

We are making random rearrangements of the array until we find the sorted one.

For n elements there are n! possible arrangements, and we have a chance of 1 in n! of having the sorted one. So the speed is O[n]

13! is over 6 billion, and we did it after only 300 million steps, so we were lucky.

Bogosort is a joke. It is the worst possible sort algorithm.

Quicksort

This works as follows:

We have two pointers called i and j, and a 'pivot' value, which is the first element. The i pointer starts at the second element, and moves up until it finds a value bigger than the pivot. The j pointer starts at the end and moves down until it finds a value less than the pivot. Then the values at i and j are swapped.

This continues until i and j meet (they will probably not be in the middle).

At this stage, the array now has two parts. The lower part is all less than the pivot, and the upper part is all more than the pivot.

The array has been partioned - put into two parts.

Then recursively we do quicksort on the two parts - provided a part has not reduced down to 1 element.

In Java - firstly main fills the array with random values, calls quicksort, and outputs the sorted array:

```
public static void main(String[] args) {
   final int N = 50;
   int[] array = new int[N];

   Random rng = new Random();// fill with random values
   for (int counter = 0; counter < N; counter++) {
     array[counter] = rng.nextInt(N);
     System.out.println(counter + " " + array[counter]);
   }
   quicksort(array, 0, array.length - 1); // quicksort entire array
   for (int counter = 0; counter < N; counter++) { // check result
     System.out.println(counter + " " + array[counter]);
   }
 }
```

Here is quicksort:

```
 public static void quicksort(int[] array, int lo, int hi) {
   if (lo < hi) { // if lo and hi have not become equal..
     int p = partition(array, lo, hi); // split. p is the split point
     quicksort(array, lo, p); // quicksort lower half
     quicksort(array, p + 1, hi); // quicksort upper half
   }
 }
```

and here is the partition process:

```
static int partition(int[] array, int lo, int hi) {
   // work on array from index lo to hi
   int pivot = array[lo]; // pivot is lowest element in section
   int i = lo - 1; // i and j are 2 moving pointers
   int j = hi + 1; // i will increase and j decrease
   while (true) { // we reak out by returning j
     do { // move i up until find element bigger than pivot
       i++;
     } while (array[i] < pivot);
     do { // move j down until find small element less than pivot
       j--;
     } while (array[j] > pivot);
     if (i >= j) { // if they meet..
       return j; // stop - we split at this position
     }
     int temp = array[i]; // else swap i and j elements
     array[i] = array[j];
     array[j] = temp; // then continue i up and j down

   }
 }
```

How fast is this? One the partition, applied to the whole array, i moves up and j down over every element, so n steps are involved.When the array is split into two, those two partition scans again go over every element (in two parts), so again n steps are involved. Every time there is a partition, there are n steps. So how many partitions will there be?

A partition divides the array into two, so the answer is how many times n can be divided by 2 - which is $\log_2 n$.

A note on logarithms

$\log_{10}1000 = 3$, because $10^3=1000$

$\log_{10}100 = 2$, because $10^2=100$

The log of a number to some base is what power you need to raise the base to to get the number. Or, it is how many times the number can be divided by the base. So for example 1000 can be divided by 10 3 times - so $\log_{10}1000=3$

In computer science we are often dividing things by 2. How often you can do this is the log to base 2. For example

$\log_2 2=1$

$\log_2 4=2$

$\log_2 8=3$

$\log_2 16=4$

The log function increase very slowly. You can *double* a number, and its log only increases by 1.

This is a simplification. The two parts may not be equal in size - we may get a small part and a larger part. But $\log_2 n$ is roughly correct.

So quicksort is $O[n \log_2 n]$

For large n, this is much faster than bubble sort and similar.

Quicksort was invented by Tony Hoare in 1959. It is the sorting method in C's standard library, and is the basis for Java's Arrays.sort() methods.

Suppose we ask quicksort to sort an array which is already in order. What will happen?

The pivot is the first element, so it is the smallest. i will move up to the second element, which is larger. j moves down trying to find an element smaller than the pivot - but none is, so j will move down until it meets i. The two parts will be the first element alone, and all the rest. A partition just breaks off the first element.

This will be true of all of them, so we will have n partitions, each needing n steps, giving $O[n^2]$ - just as bad as bubble sort.

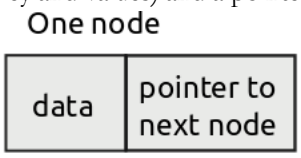
For this reason the pivot value is usually not taken as the first element, as Hoare did in the original version. But this shows the idea of quicksort.

Linked List

An array has a set of elements 'in a line', so is a type of list. An array is usually stored as a single block of memory. Elements can be accessed directly by index. However it is not possible to add or remove elements from an array, since it uses that fixed block of memory.

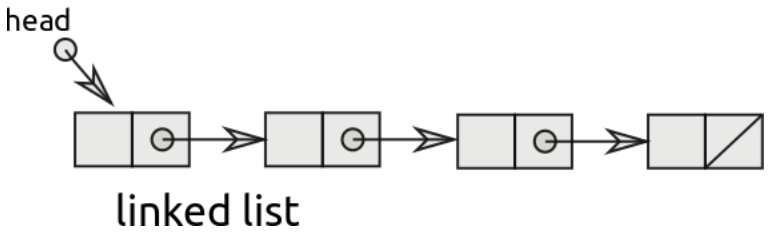
A linked list is a set of nodes, each node having a pointer to the next node. The nodes are usually not next to each other in memory - they can be anywhere. This means it is easy to add and remove nodes at runtime:

Each node contains two fields. These are some data (usually a key and values) and a pointer field, which points to the next node.



Exactly what the pointer is depends on the implementation. In C it would be a RAM address of the next node. In Java it would be a reference.

The diagram shows how linked lists can be thought of. There is a variable called 'head' (usually) which is a pointer, and it points to the first node of the list. In the last node the next pointer has a special 'null' value, meaning it points nowhere. It is usually drawn as shown.



Linked list in C

We need a way to represent a node.We can do this with a struct and a typedef

```
typedef struct NodeStruct{
    int data;
    struct NodeStruct * next; // pointer to next block
} Node;
```

We just use an int as the data. This could be other types, or a key field and value fields, or a void * as a pointer to any type. The 'next' field will be the address of the next node in teh list.

A function to construct a new node is

```
Node * newNode(int number)
{
    Node * node = (Node *) malloc(sizeof(Node));
    node->data=number;
    node->next=NULL;
return node;
}
```

We call malloc to get the memory to hold a new Node. malloc returns a void *, and we cast it to Node *. We set the 'next' field to NULL to start with, this node is not linked to any others.

We can also define a linked list pointer type:

```
typedef struct ListStruct {
    Node * head; // pointer to first node
} * ListPtr;
```

This just has one field - a pointer to the first node in the list. We could have other fields in here, such as a pointer to the last node, or a count of how many nodes there are.

and a function to construct a new list

```
ListPtr newList() {
```

```
    ListPtr llp = (ListPtr) malloc(sizeof (struct ListStruct));
    llp->head = NULL;
    return llp;
}
```

The value of the 'head' field is NULL, the pointer to nowhere. This is because this is a new list, and there are no nodes in it.

This adds a new node to a list:

```
void put(ListPtr llp, Node * newNode) {
    if (llp->head==NULL)
    {
        llp->head=newNode;
        return;
    }
    newNode->next= llp->head ;
    llp->head=newNode;
}
```

We go through this code in detail.

We need different action if the list is empty, or if there are already nodes in it.

```
if (llp->head==NULL)
```

llp->head is the address of the head of the list - that is, the location in RAM of the first node in the list. If this is NULL, this means there is no first node, becaus eteh list is empty. So..

```
        llp->head=newNode;
        return;
```

newNode is the *address* of the node we are adding to the list. Assigning llp->head to this means head points to this node. Then we return from the function, because nothing else needs to be done.

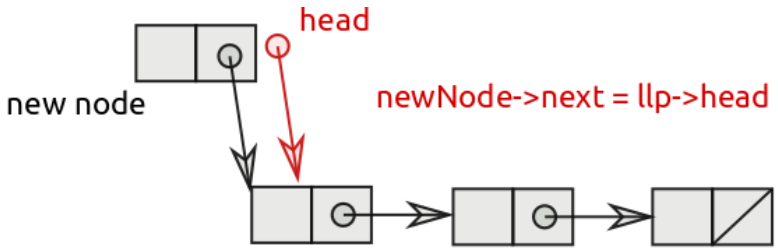
Othewise, llp->head is the address of the head of the list (before we add a new one).

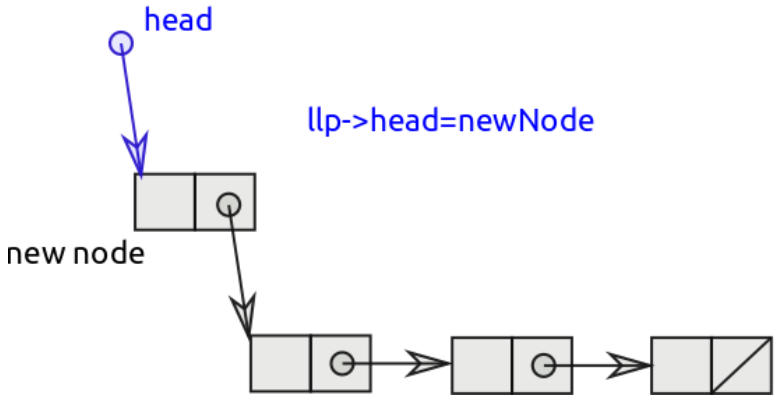
```
    newNode->next= llp->head ;
```

This means that in the new node, the 'next' field now points to what was the head. Then

```
    llp->head=newNode;
```

changes the head pointer to point to this new node. So the node is added at the head of the list. These two steps are:





We can make new nodes, construct a list and addd nodes to the list by:

```
    Node * n1 = newNode(7);
    Node * n2 = newNode(4);
    Node * n3 = newNode(3);
    ListPtr list = newList();
```

To traverse a data structure means to go through it, visiting every node, and doing something. We will just output each node:

```
void printNode (Node * whichNode)
{
    printf("Node at %p  data=%d next node at %p\n", whichNode, whichNode->data, whichNode-
>next);
}

void traverse(ListPtr llp) {
    if (llp->head==NULL) return;
    Node * where = llp->head;
    while (where != NULL) {
        printNode(where);
        where = where->next;
    }
    printf("End of list\n");
    return;
}
```

If the list is empty, traverse does nothing:

```
    if (llp->head==NULL) return;
```

Otherwise

```
    Node * where = llp->head;
```

sets up a new pointer to a node called 'where', and initialises it to point to the first node. 'where' will move along the list to the end.

```
        where = where->next;
```

where->next is the address of the next node, so this moves 'where' along from teh current node to the next.

```
     while (where != NULL) {
```

in the last node, the 'next' field is NULL. So if where=NULL this means we have reached the end of the list. printNode just prints the details of the node.

We can find out about our list like this:

```
    traverse(list);
    printf("Size of int=%ld size of pointer =%ld\n", sizeof(int), sizeof(Node *) );
```

The output (on this computer) is

```
Node at 0x55a9921ba2a0  data=3 next node at 0x55a9921ba280
Node at 0x55a9921ba280  data=4 next node at 0x55a9921ba260
Node at 0x55a9921ba260  data=7 next node at (nil)
End of list
Size of int=4 size of pointer =8
```

So on this implementation, an int is held in 4 bytes, and a pointer is in 8 bytes (this is a 64bit processor). So a node, with an int field and a pointer to the next, is 12 bytes in length. The output shows the memory contents are like this

Contents	7	nil		4			3	
Address	260	264	26A	280	284	28A	2A0	2A4
	Last node			Second node			First node	

When malloc is supplying memory, it is providing locations at addresses which get smaller - it is moving 'backwards' in memory. This list starts at RAM adddress

0x55a9921ba2a0

but we do not need to know these actual numbers, since we can write this as

llp->head

LinkedList in Java

Java has an interface LinkedList in the Collections framework, but we want to code our own.

In Java we can use classes, with fields constructors and methods, in place of C's structs and functions. We start with a class for a Node:

```
class Node {

  int data;
  Node next;

  Node(int value) { // make a new node
    data = value;
    next = null;
  }
```

Then a class for a LinkedList:

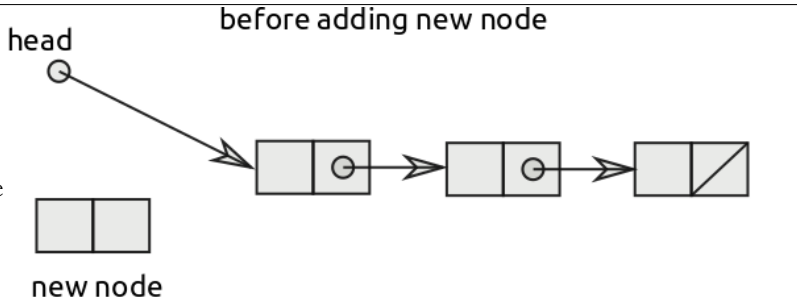
```
class LinkedList {

  Node head; // just one field

  LinkedList() { // make a new empty list, no nodes..
    head = null;
  }

  void add(Node node) {
    .. see later
  }

  void traverse() {
    .. see later
  }

}
```

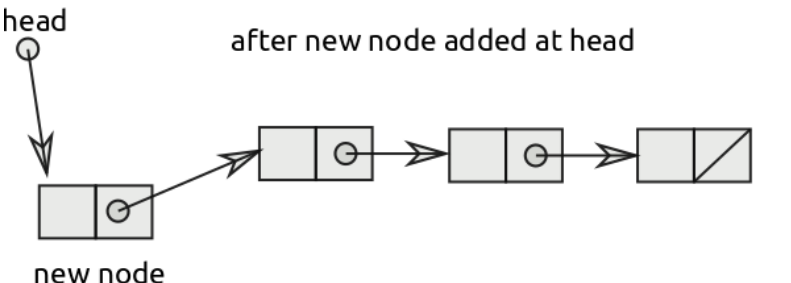
A linked list has just one data field, head, which points to the first node in the list. A new list is empty, with no nodes, so head is null.

The add method will add a new node to the list.

To traverse a data structure means to go though it somehow, going to every node, and doing something there. Our traverse method will output the node data fields.

The process of adding a new node (at the head) is as shown.





A copy of the head pointer is placed in the next field of the new node, so the new node next points to the old head. Then head is changed to point to the new node.

The Java code is

```
    void add(Node node) {
    if (head == null) { // if the list is empty
      head = node; // the new node is the head now
      return;
    }
    // else
    node.next = head;
   head=node;
  }
```

To traverse the list we have a variable named 'where' which is a moving pointer which starts at the head and moves to the end:

```
  void traverse() {
    Node where = head;
    while (where != null) {
      System.out.println(where.data);
      where = where.next;
    }
  }
```

We can use our list class like this:

```
  public static void main(String[] args) {
    LinkedList list = new LinkedList();
    Node n1 = new Node(6);
    Node n2 = new Node(68);
    Node n3 = new Node(4);
    list.add(n1);
    list.add(n2);
    list.add(n3);
    list.traverse(); // 4 68 6
  }

}
```

Variables like 'list' here are reference types, and 'list' is a reference to a LinkedList object. This reference is a pointer. But it is not an actual RAM address as pointers are in C - because Java objects move around in memory. But we can think of these pointers as if they were.

Searching a linked list

Usually this means given a key, search a list for a matching key and return the corresponding value.

The algorithm is:

start at the head

repeat

does the key match? if so return with the value

else move to the next node

until end of list reached

In our sample code we just have int keys and no values. We just return which node it is (counting the first node as 1), or -1 if not found (in C):

```
int find(ListPtr llp, int target) {
   int counter = 1;
   Node * where = llp->head;
   while (where != NULL && where->data != target  ) {
       where = where->next;
       counter++;
   }
   if (where == NULL) return -1;
   else return counter;
}
```

For example:

```
   ListPtr list = newList();
   put(list,newNode(8));
   put(list,newNode(7));
   put(list,newNode(10));
   put(list, newNode(9));

   printf("10 is at node %d\n", find(list,10) );
   printf("11 is at node %d\n", find(list,11) );
```

Output:

```
10 is at node 2
11 is at node -1
```

It is important to say

```
   while (where != NULL && where->data != target  ) {
```

and not

```
   while (where->data != target && where != NULL ) {
```

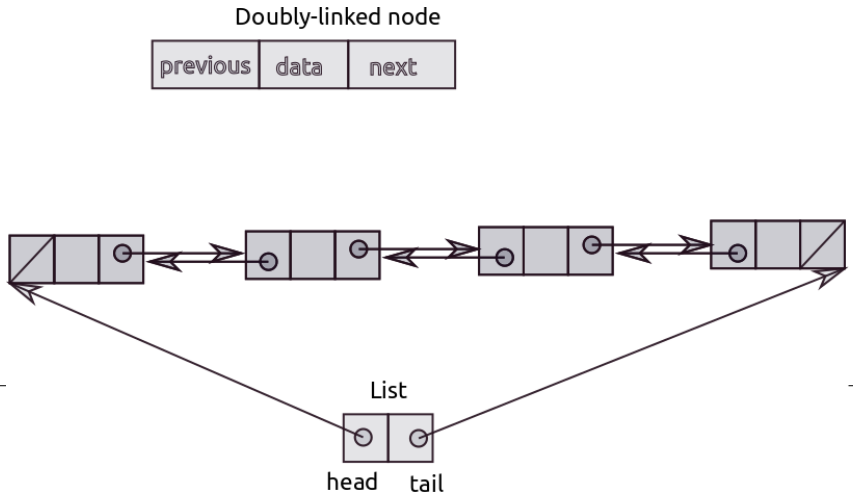
This is because at the end of the list, where ==NULL, and so in the first version where->data is not evaluated (because C does short-circuit evaluation, and when it finds the first condition is false it stops)

In the second version at the end where is NULL and where->data gives a segmentation fault. The first law of C programming is

Thou shalt not follow the null pointer

Doubly-linked lists

If the nodes in our list point both to the next and the previous node, we have a doubly-linked list. We can traverse a doubly-linked both 'forwards' and 'backwards'



We can code the node class in Java:

```
class Node {

  int data;
  Node next;
  Node previous;

  Node(int value) { // make a new node
    data = value;
    next = null;
```

```
    previous = null;
  }
}
```

and a doubly-linked list class is

```
class LinkedList {

  Node head;
  Node tail;

  LinkedList() { // make a new empty list, no nodes..
    head = null;
    tail = null;
  }

  void add(Node node) {
    if (head == null) { // if the list is empty
      head = node; // the new node is the head now
      tail = node; // so is the tail
      return;
    }
    // else insert at head
    node.next = head;
    head.previous=node;
    head = node;

  }

  void traverse() {
    Node where = head;
    while (where != null) {
      System.out.println(where.data);
      where = where.next;
    }
  }

  void traverseBack() {
    Node where = tail;
    while (where != null) {
      System.out.println(where.data);
      where = where.previous;
    }
  }

}
```

We insert a new node at the head:

```
    node.next = head; // next field of new node points to current headnode
    head.previous=node; // previous of current head points back to new one
    head = node; // change head to new node
```

We can use this as:

```
    LinkedList list = new LinkedList();
    list.add(new Node(10));
    list.add(new Node(5));
    list.add(new Node(20));
    list.traverse();
    list.traverseBack();
```

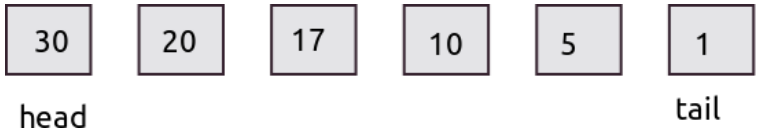
output:

```
20
```

```
5
10
10
5
20
```

Priority Queues

We could have the nodes in our linked list in order of the key field, and then we would have a priority queue. This is a list where the items are kept in order of 'importance' somehow. It might be used to model a list of passengers waiting for an aircraft seat, for example.



We use a doubly-linked list for this. The only change is the method to insert a new node. Instead of always inserting a new node at the head, we must look along the list to insert at the correct place.

There are four situations we have to handle..

```
void add(Node node) {
    if (head == null) { // if the list is empty
     ..
    // else work out where to put it
    ..
    if (where == null) {// if reached the end
      ..
    if (where == head) { // if goes before all of them
      ..
    // must insert somewhere along list
    ..
  }
```

If the list is empty, head is null, and we do this:

```
if (head == null) { // if the list is empty
      head = node; // the new node is the head now
      tail = node;
      return;
    }
```

Otherwise we have to work out where to put it. We use a moving pointer, called 'where', which starts at the head, and looks for a smaller node - and we must insert the new node just before this smaller node:

```
    Node where = head; // start at the head
    // look for a smaller node, or reach end
    while (where != null && where.data > node.data) {
      where = where.next;
    }
```

We may need to put the new node after them all, after the current tail. Or before them all, at the current head. Or somewhere along the list. Firstly at the tail:

```
    if (where == null) {// if reached the end
      tail.next = node; // link current last node to new one
      node.previous = tail; // and link it back
      tail = node; // this is the tail now
      return; // finish
    }
```

Or at the head:

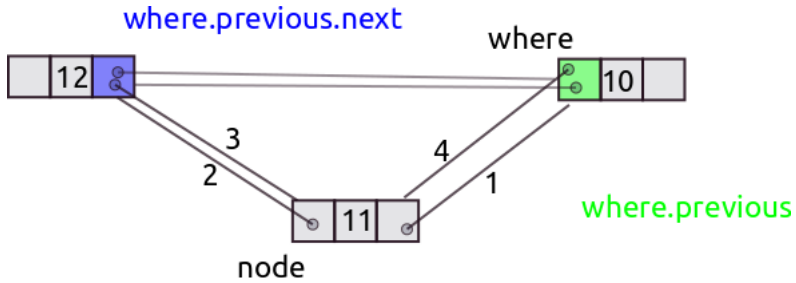
```
  if (where == head) { // if goes before all of them
```

```
      node.next = where; // link new node next to current head
      where.previous = node; // link current head back to new node
      head = node; // this is head now
      return; // finished
    }
```

or somewhere along the list:

```
    // where is first smaller node
    // new node must go before that one
1   node.next = where; // after node link to where
2   where.previous.next = node;// link one that was before where to new one
3   node.previous = where.previous; // back link of new one
4   where.previous = node; // back link of where
```

We illustrate those steps, inserting 11 between 10 and 12:



Speed of linked list

To find a node with a given key we must start at the head and follow the links until we find it or we reach the end. We might be lucky at find it at the head, in 1 step, or unlucky and find it at the end after n steps (if there are n nodes). The average would be n/2.

If its a priority queue it is slightly faster, since if we find a key smaller than the target (in an increasing list) we can stop, knowing it is not present, without searching to the end. Time is still n/2.

To insert a new node in an unordered list, at the head, takes just one step. This is called constant time (constant because it does not depend on n).

Compared with an array, access to a linked list node is slower, because we have to follow the pointers hrough the nodes from the start, while for an array we just use the index.

The advantage of a linked list is that it can grow and shrink which the program is running, while an array is fixed in size at compile-time.

Stacks

A stack is a pile. You can add extra things on top of the pile, and take things off the top. That is all. You cannot take something out part way through the pile, or everything will fall over.

The two operatitions are called push and pop. You push data onto a stack, and pop values off. For example:

Stack			7			
		10	10	10		
	9	9	9	9	9	
	4	4	4	4	4	
	2	2	2	2	2	
		Push 10	Push 7	Pop	Pop	Pop
				Get 7	Get 10	Get 9

When we push 10 on, then do a pop, we get the 10 back. The last data value in will be the first one out. A stack is also called a last-in-first-out or LIFO structure.

We will look at uses of stacks later.

Stacks in Java

We can implement a stack in Java as a linked list, pushing nd popping elements at teh head of the list. The node is:

```
class Node {
  int data;
  Node next;
  Node(int value) { // make a new node
    data = value;
    next = null;
  }
}
```

and a Stack:

```
class Stack {
  Node top;
  Stack() { // make a new empty list, no nodes..
    top = null;
  }

  void push(Node node) {
    if (top == null) { // if the list is empty
      top = node; // the new node is the head now
      return;
    }
    node.next = top;
    top = node;
  }

  int pop() {
    if (top == null) {
      throw new RuntimeException("Stack underflow");
    }
    int val = top.data;
    top = top.next;
    return val;
```

```
  }
}
```

Suppose we try to pop a value off a stack, and the stack is empty? What should we do. Java has an exception-handling feature, which we use here.

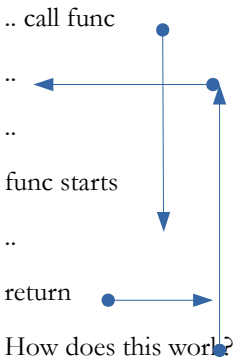
For example

```
    Stack stack = new Stack();
    stack.push(new Node(10));
    stack.push(new Node(7));

    System.out.println(stack.pop()); // 7
    System.out.println(stack.pop()); // 10
    System.out.println(stack.pop()); // stack underflow exception
```

Uses of stacks

A stack is a classical data structure and it has many uses. We will look at the use of a stack for procedure calls. This applies to most languages - C, Java, Basic, asemby language and machine code - the idea is the same.

In different languages the code blocks have different names - functions, procedures, methods, sub-routines and so on. In all languages, we can call a function. Th epath of exceution goes off to the start of the function, runs through it, then at the end of the function there is a return. This means the execution path returns to the instruction after the original call:



How does this work? It works by using a stack:

1. The call instruction pushes the address of the next instruction (after the call) onto a stack, then switches to the start of the function.
2. A return instruction pops an address off the stack, and carries on from that address - which will be after the call.

Like this:

Address	Instruction	Next instruction	Stack
100	..	101	7,2,78
101	call func	200	7,2,78, 103
103	..		
..	..		
200	func	201	7,2,78, 103
201	..		7,2,78, 103
	..		7,2,78, 103
210	return	103	7,2,78

Why do it like this? Suppose inside the function, another function is called. It still works:

Address	Instruction	Next instruction	Stack
100	..	101	7,2,78
101	call funcA	200	7,2,78, 103
103	..		
..	..		
200	funcA	201	7,2,78, 103
201	call funcB		7,2,78, 103,202
202	..		7,2,78, 103
210	return	103	7,2,78
300	funcB	301	7,2,78, 103,202
301		302	7,2,78, 103,202
302	return	202	7,2,78, 103

Whenever we have a call, the next address is pushed on the stack, and at every return, the address is popped back off. We can have function calls nested to any depth.

But not an infinite depth. Infinite recursion produces a 'stack overflow' error, because the stack grows larger than the available memory.

The stack can also be used for parameter passing, and return values:

1. The calling code pushes the return address on the stack, followed by te parameter values, then jumps to the function start
2. The function pops the parameters off the stack and uses them.
3. At the end of the function, the return address is popped off the stack, the return value pushed, then jump to the return address
4. The calling code pops the return value of the stack.

Again this still works with nested calls.

Function calls do not always work like this. Sometimes parameters are passed in processor registers. If a stack is used, the order in which parameters are pushed is important. These ideas form a calling convention - examples are cdecl (C declaration, used by many C compilers) and ABI, the application binary interface, used for Linux system calls.

Processor chips have push and pop instructions, and a stack pointer register, which has the address of the top of the stack. Applications which compile to native code will use these. Languages which have a runtime (such as Java) will have their own stack setup. Many Java bytecode instructions make use of a stack.

Stacks in expression evaluation

An expression is for example

2+3 X 4

To 'evaluate' it means to work out its value. An interpreter mus be able to do this. A compiler must create the code to do it.

The obvious method is to work left to right, processing each character in turn. But we have a problem. From left to right we get '2+3'. But we must not do the add, because we must do the 3X4 first, then do the add 2. Somehow the system must 'remember' it needs to do the 2+, after it has done the 3X4.

An operator is + - X / (and others) - things that do arithmetic.

Each operator has a precedence. Those with higher precedence must be applied first, So X has higher precedence than +.

One algorithm is as follows. We have 2 stacks - a value stack and an operator stack.

We deal with each character, left to right.

If it is a value, push it onto the value stack

If it is an operator, and the operator stack is empty, push onto operator stack

If operator stack not empty, apply higher precedence operator to values popped off stack, push result onto value stack

After input all used, pop operator off stack, apply it and push result onto value stack, until operator stack empty. We are left with the expression value as the single value on the value stack

For example, with 2+3X+6

character	value stack (top on right)	operator stack (top on right)	notes
2	2		value pushed
+	2	+	operator stack empty, so push
3	2 3	+	value pushed
*	2 3	+ *	* has higher precedence - push
4	2 3 4	+ *	value pushed
+	2 12	+ +	The * applied to the 3 and 4
6	2 12 6	+ +	value pushed
	2 18	+	input empty
	20		expression value=20

This just outlines the process.

One other issue is the need to use tokens. The expression

8.5+7.2

has seven characters, but three tokens:

<number><operator><number>

The above algorithm in fact applies to tokens not characters. A tokeniser changes a stream of characters into a stream of tokens.

Stack in C

We show a different way to implement a stack, in C

We put the stack in an array:

```
int STACK_MAX=5;
int stack[5];
int stackTop=0;
```

stackTop is the location for the next value pushed onto the stack.

```
void push(int val)
{
    if (stackTop==STACK_MAX) // array is full
        printf("Stack overflow\n");
    else
    {
    stack[stackTop]=val;
    stackTop++;
    }
}

int pop()
{
    if (stackTop==0) // array is empty
        printf("Stack underflow\n");
    else
    {
    stackTop--;
    return stack[stackTop];
    }
}
```

We can use this stack as:

```
    push(9); push(10); push(11);
    printf("%d\n",pop()); // 11
    push(9); push(10); push(11);
    push(2); // stack overflow
```

How does this compare with using a linked list?

1. We do not need memory for pointers, as needed for a linked list
2. If the array size is small, we risk getting a stackoverflow
3. If the array size is large, we use a lot of memory, which may be wasted if teh stack never grows large.

Abstract data types

Is a stack an array or a linked list? We have seen both used as a stack. A new idea is an abstract data type ADT.

An ADT is a data structure defined by *what it can do*. For example, a stack ADT can do just two things:

push a data value onto the stack, and

pop a value off the stack, getting the value most recently pushed.

An implementation of an ADT is an actual way of making such a thing work. We have seen two implementations of a stack ADT - as a linked list and as an array. As we have seen, different implementations have differnt good and bad points.

Queues

A queue data structure is like a queue of cars at traffic lights. It is a first in first out structure - FIFO, but a stack is LIFO.

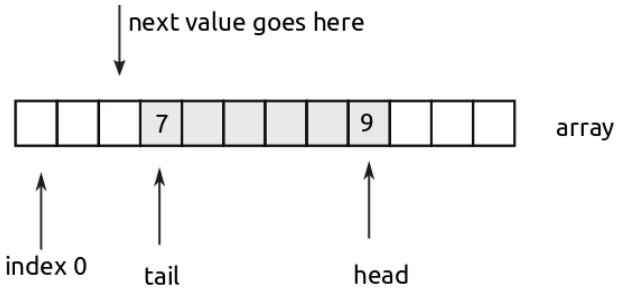
A queue is an ADT with two operations:

enQueue - put a data value into a queue

deQueue - get a value out of a queue, which is the first one put in

Circular queue in C

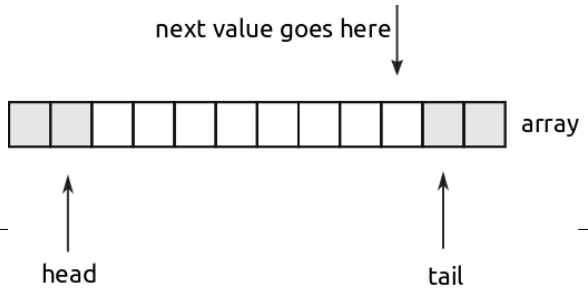
One implementation of a queue is in an array:



In this example, 7 was the most recent value added to the queue. If a value is removed, it will be 9.

As values are added, tail will move to the left, and as values are removed, head will also go left.

After a while tail will get to 0. We could then say the queue was full. But there would still be free locations at the end of the array. Instead we can 'wrap-around' the tail to the end of the array. So after some insertions and removals we might get:



This is a circular queue (also called a ring buffer). Insertions and removals from the queue make it rotate around a fixed memory block.

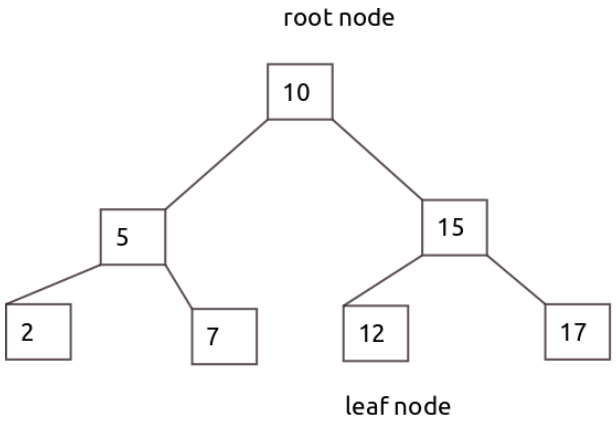
In C:

```
#define ARRAY_SIZE 5
#define LAST_INDEX 4
int array[ARRAY_SIZE];
int head = 0;
int tail = 1;
int qCount = 0; // count elements in the queue

void enqueue(int number) {
    if (qCount == ARRAY_SIZE) {
        printf("Queue full\n");
        exit(1);
    }
    tail--;
    if (tail == -1) tail = LAST_INDEX;
    array[tail] = number;
    qCount++;
}

int dequeue() {
    if (qCount == 0) {
        printf("Queue empty\n");
        exit(1);
    }
    int result = array[head];
    head--;
    qCount--;
    if (head == -1) head = LAST_INDEX;
    return result;
}
```

Trees

Binary Search Tree

Each node has two pointers, to the right and left nodes below it. The root node is the 'top' node. Each node also carries data, which is key and values sets. Here we just show keys, as integers.



In a binary search tree, each node below and to the left of a node is less than it. Every node to the right is greater - as the diagram shows.

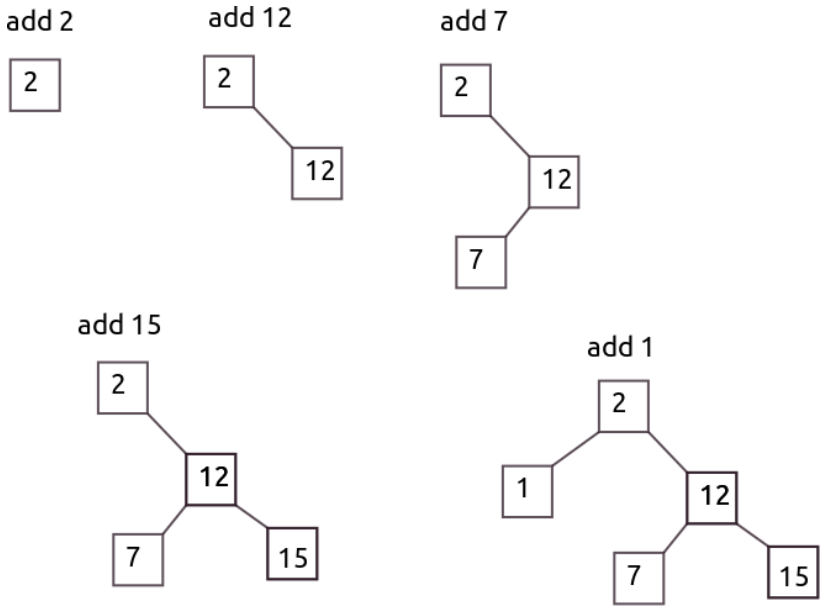
We see later why this is useful.

In Java:

```
class BST {

  Node root = null;

  void insert(Node newNode) {
    ..

  }

  void traverse() {
    ..
  }

  ..

}

class Node {

  int data;
  Node left;
  Node right;

  Node(int value) { // make a new node
    data = value;
    left = null;
    right = null;

  }
}
```

How do we insert a new node? If the tree is empty (root==null) then the root is this new node, and we have finished. Otherwise we start at the root and go left if the new value is less, and right if it is more. We continue until we reach a leaf, and link the new node in there.

Watch what happens when we we add 2,12,7,15 and 1 into an empty tree, in that order:

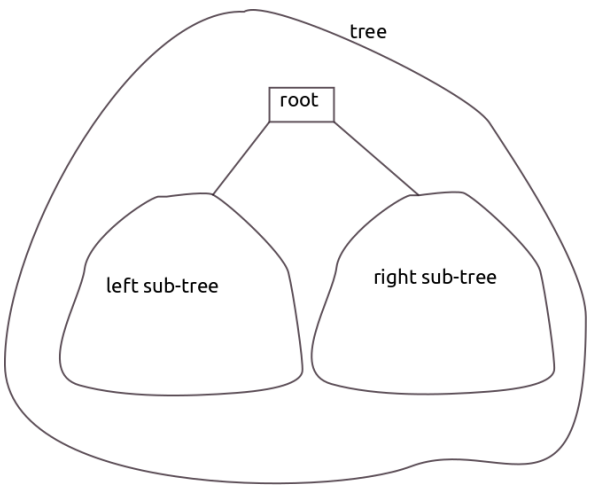


so the code for insert is:

```
void insert(Node newNode) {
  if (root == null)// empty tree
  {
    root = newNode;
    return;
  }
  Node next = root;
  Node where = next;
  // where is current position
  // next is where to go next
  while (next != null) {
    where = next;
    if (where.data > newNode.data) {
      next = next.left;
    } else {
      next = next.right;
    }
  }
  if (where.data > newNode.data) {
    where.left = newNode; // link onleft
  } else { // or right
    where.right = newNode;
  }

}
```

BST traversal

For the traversal of the tree, we use the fact that trees are recursive:



We traverse the tree by

if left sub-tree not null, traverse it

visit the root

if root sub-tree not null, traverse it

The code is

```
void traverse() {
  visit(root);
}

void visit(Node whichNode) {
  if (whichNode.left != null) {
    visit(whichNode.left);
  }
  System.out.println(whichNode.data);
  if (whichNode.right != null) {
    visit(whichNode.right);
  }
}
```

For example:

```
BST tree=new BST();
tree.insert(new Node(2));
tree.insert(new Node(12));
tree.insert(new Node(7));
tree.insert(new Node(15));
tree.insert(new Node(1));
tree.traverse();
```

Output

```
1
2
7
12
15
```

Why?

This is what happens, in sequence:

The numbers show the order. So the first thing is

1 to visit the left node of the root node. Then

2 left of the 1 node, is null

3 output 1 node

4 right is null - nothing. Traversal of left sub-tree now complete

5 output 2 node

6 visit right sub-tree of the root node

and so on.

This is called an in-order traversal. We do

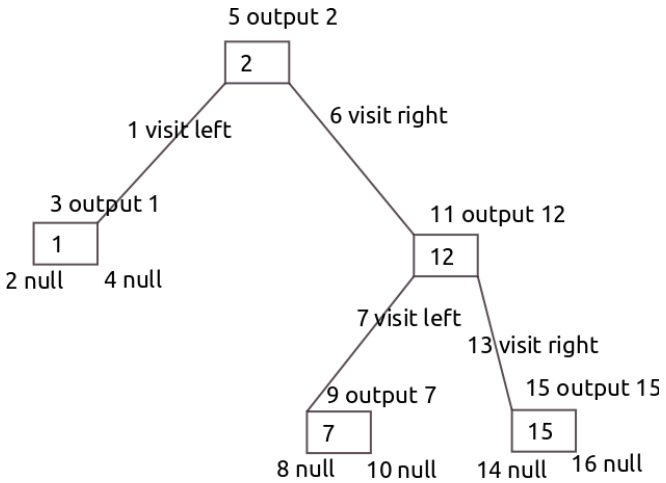
left node right

We can also do pre-order

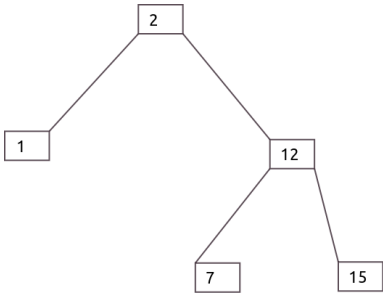
node left right

or post-order

left right node



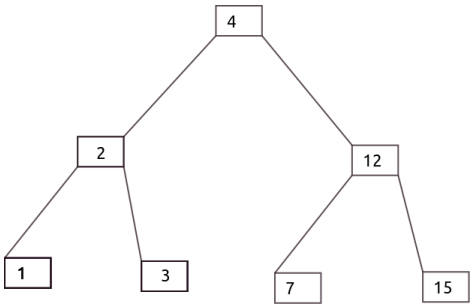
BST depth



The depth of a tree is how many levels it has. The tree shown has depth 3.

In a complete tree every level is completely filled. So this is not complete, but

this is complete (and has depth 3, 7 nodes)



How many nodes in a complete tree with different depth?

Depth	Number of nodes
1	1
2	3
3	7
4	15

So a complete tree with depth d has 2^d-1 nodes

What about the other way round? If we have n nodes, what is the depth? The answer is $\log_2(n+1)$. For example if n=16, the depth = $\log_2(16)=4$

Tree insertion sort

The output of the in-order traversal of the BST is ordered. This means we have a new sort method - insert the data into a BST, and do an in-order traversal.

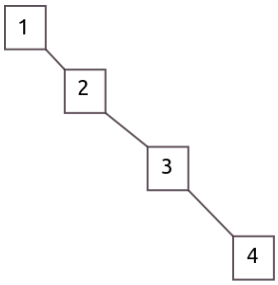
How fast is it? Inserting each node will mean starting at the root and moving to a leaf node, so the number of steps is the depth, $\log_2(n+1)$. As we insert nodes the depth increases, so this is just a rough measure, which we take as $\log_2(n)$. We have to do this for n nodes, so the total steps is n $\log_2(n)$.

The traversal s the same, we need to go down to a depth of $\log_2(n)$ n times.

So a tree insertion sort if O[n log(n+1)]

Balanced trees

Suppose we insert into an empty tree values 1,2,3,4 in that order - what will we get? The 1 will go in as the root. The root will go to the right. The 3 will go to the right of the 2. We get this:

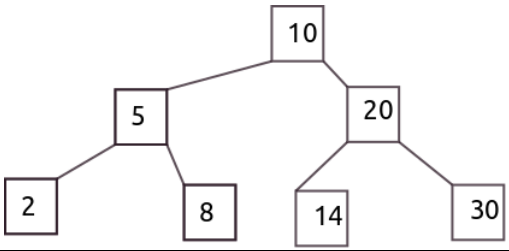


The tree is totally unbalanced, with only one node on each level. If we insert n items into an empty tree in order, we will get a depth of n, and the insertion and traversal will be $O[n^2]$. This is similar to the bad performance of the original quicksort on ordered data.

In general we want trees to be as balanced as possible, because then they have a minimum depth of log n, and searches are as fast as possible. There are two standard ways of doing this red-black trees and AVL trees. These have insertion and deletion algorithms which keep trees balanced.

Array implementation

Complete trees can be implemented as an array, with no pointers. This example:



would be this array (counting index starting at 1)

value	10	5	20	2	8	14	30
index	1	2	3	4	5	6	7

The value at index 2 (the 5) has left and right sub-nodes at 4 and 5. In general the sub-nodes of the element at n are at 2n and 2n+1

The parent of the node at n is n div 2 (eg 14 at 6 has parent at 3. 8 at 5 has parent at 2)

Trees in general

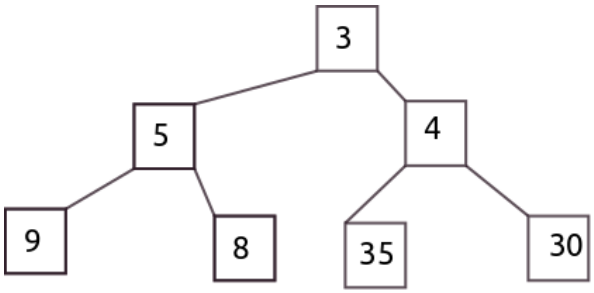
A BST is just one type of tree. In a binary tree each node has 0 1 or 2 sub-nodes, but not all trees are binary. As well as pointers to sub-nodes, each node may have a pointer 'up' to its parent node. Or in an array implementation, no pointers at all. Trees may be ordered, or not.

A tree is a type of graph (not a Cartesian x-y graph). Graphs have node linked to other nodes. One use for a graph is a road map, where the nodes are places and the links are roads. Links are weighted with the road distance (and maybe the traffic density and other data).

A tree is an acyclic graph - a graph with no loops.

Heap

An example of a different type of ordering is a heap, where every node is larger than every sub-node beneath it (in a max-heap) or smaller (in a min heap). An example min-heap:

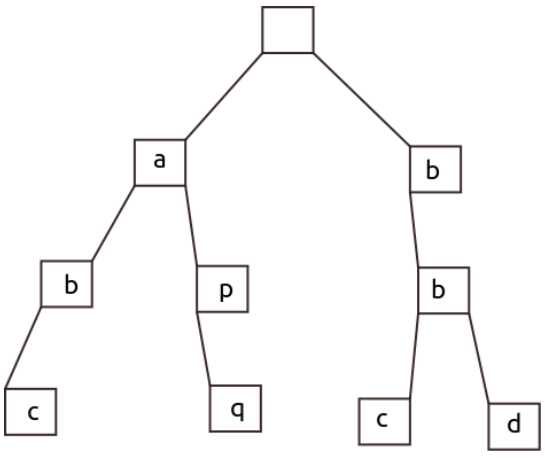


The ordering is by level. Within a level, there is no order.

Runtime systems (such as Java's JRE) often refer to a stack and a heap for memory. Usually the stack is an actual stack, used for return address, parameter ng, local variables and so on as described above. But the heap is not ordered, so is not a heap data structure. It is used for global data (if there is any - Java has none), class definitions, objects and so on. This use of 'heap' with two meanings is confusing.

Tries

A trie is a different kind of tree, usually used to hold strings of characters. Most trees hold a data item in a node. A trie holds the data down a path from the root.



Suppose we insert strings "abc", "apq", "bbc" and "bbd" into an empty trie, we get:

Tracing a path down from the root, we get a string.

This is not a binary tree, because a node might have more than 2 sub-nodes.

In Java, we might have a node

```
class Node {

  char c;
  Node[] subNodes = new Node[26];
  Node parent;

  Node(char value, Node n) { // make a new node with character value and given parent
    c = value;
    parent = n;
    for (int index = 0; index < 26; index++) {
      subNodes[index] = null;
    }
  }

  ..
}
```

So a node has a single character and an array of 26 subNodes beneath it, which are null unless they 'go somewhere'.

We will only have strings with the letters 'a' to 'z'. If we wanted to allow any character, we could have used an ArrayList in place of the array.

We also have a link to the parent of the node, so we can move 'up' the trie.

The Trie class is:

```
class Trie {

  Node root;

  Trie() { //constructor
    root = new Node((char) 0, null);
  }

  boolean contains(String str) { //does the trie contain this string?
    ..
  }

  ArrayList<String> getAll() { // get a list of all strings in the trie
    ..
  }

  void visit(Node n, ArrayList<String> list) { // visit a node - used by getAll
    ..

  }

  void insert(String str) { // insert a string into trie
    ..
  }

}
```

To start with, we look at insert:

```
  void insert(String str) { // insert a string into trie
    str = str.toLowerCase();
    Node where = root; // start at root
    for (int index = 0; index < str.length(); index++) {
      char c = str.charAt(index); // get each character in teh string
      int whichBox = c - 'a'; // which array element matches this character?
      if (whichBox < 0 || whichBox > 25) { // if not 'a' to 'z'
        throw new RuntimeException("invalid char = " + c);
      }
      if (where.subNodes[whichBox] == null) { // if this currently goes nowhere
```

```
      where.subNodes[whichBox] = new Node(c, where); // make a new node for this character
    }
    where = where.subNodes[whichBox]; // go down the tree ready for next character
  }
}
```

contains() is similar, but if we find a null link, we can return false:

```
boolean contains(String str) { //does the trie contain this string?
  str = str.toLowerCase();
  Node where = root; // start at the top
  for (int index = 0; index < str.length(); index++) {
    char c = str.charAt(index); // each character
    int whichBox = c - 'a';
    if (whichBox < 0 || whichBox > 25) {
      throw new RuntimeException("invalid char = " + c);
    }
    if (where.subNodes[whichBox] == null) {
      return false; // its not in the trie
    }
    where = where.subNodes[whichBox];
  }
  return true; // we found every character - is in the trie
}
```

Getting all the strings from the trie requires some processing. The idea is

Check each node to see if it is terminal - all subnodes null. If so, this is the last character in a string, so..

Work back up the trie from this one, adding characters to the string, until we reach the top

The string will be backwards, so reverse it

Add it to a list of strings.

We need a method of Node to check if it is terminal:

```
boolean isTerminal() // is this a terminal node?
{ // with all subNodes null?
  for (int index = 0; index < 26; index++) {
    if (subNodes[index] != null) {
      return false;
    }
  }
  return true;
}
```

and a method to work up from a terminal node to the top making a string:

```
String stringFrom() { // make a string which ends at this node
  StringBuilder s = new StringBuilder(); // empty
  Node where = this; // start here
  while (where != null) { // until we get to the top
    s.append(where.c); // stick this character on
    where = where.parent; // and move up
  }
  s = s.reverse();
  return new String(s);
}
```

String is immutable. We use a StringBuilder, and turn it into a String when we have finished.

To get all strings, Trie has a recursive visit method:

```
void visit(Node n, ArrayList<String> list) { // visit a node - used by getAll
  for (Node node : n.subNodes) { // for every subnode
    if (node==null) continue; // skip if null
    if (node.isTerminal()) { // if terminal = end of string
```

```
        list.add(node.stringFrom()); // get string, add to list
      } else {
        visit(node, list); // recursively visit the subnode
      }
    }
```

which is started by:

```
 ArrayList<String> getAll() { // get a list of all strings in the trie
    ArrayList<String> list = new ArrayList<>();
    visit(root, list);
    return list;

  }
```

Sample run:

```
    Trie trie = new Trie();
    trie.insert("abc");
    trie.insert("apq");
    trie.insert("bbc");
    trie.insert("qaac");
    System.out.println("Contains apq " + trie.contains("apq"));
    System.out.println("Contains abb " + trie.contains("abb"));
    ArrayList<String> list = trie.getAll();
    for (String str:list)
      System.out.println(str);
```

output:

```
Contains apq true
Contains abb false
abc
apq
bbc
qaac
```

Hash tables

A hash table (or hash map) is designed for *fast insertion and retrieval* of data.

The data is in the form of key-value pairs

We *calculate* where to put it.

On insertion, we use the key to calculate an address, and store it there.

On retrieval of a given key, we repeat the calculation, and look in that address.

The calculation is called a hash code or hashing function.

The data is stored as elements in an array, which is initially 'empty'

The array elements are called buckets or bins

This is the basic idea. In fact there are several other issues. We start looking at a simple version in C.

Basic hash table

The data will have an int for a key (in the range 0 to 99), and a value which is a random string:

```
#define VAL_LEN 10

typedef struct NodeStruct{
    int key;
    char value[VAL_LEN];
} Node;

Node * newNode(int keyValue) // make a new node with given key
{
    Node *ptr = (Node *) malloc(sizeof(Node));
    ptr->key = keyValue;
    // make random value
    for (int index=0; index<VAL_LEN-1; index++)
        ptr->value[index]= 'A'+rand()%26;
    ptr->value[VAL_LEN-1]=0; // terminal null
    return ptr;
}

void show(Node * ptr)
{ // output a key-value pair
    if (ptr==NULL) printf("Null\n");
    else
    printf("Key %d - value %s\n", ptr->key, ptr->value);
}
```

Here is the hash table:

```
#define TABLE_SIZE 10
Node * hashTable[TABLE_SIZE];

void initTable() // initialize the array
{
    for (int index = 0; index < TABLE_SIZE; index++)
        hashTable[index] = NULL;
}

int hashCode(int keyValue) { // the hashing function
    return keyValue / 10;
```

```
}

void put(Node * ptr) // insert node in array
{
    hashTable[hashCode(ptr->key)] = ptr;
}

char * get(int keyValue) // fetch value given a key
{
    return hashTable[hashCode(keyValue)]->value;
}

void showTable() {
    for (int index = 0; index < TABLE_SIZE; index++) {
        printf("Cell %d ", index);
        show(hashTable[index]);
    }
}
```

The array is very small - just 10 locations - to explain the idea. The hashing function is very simple - just divide teh key by 10, so that for example key 78 will be stored in cell 7.

We try it out:

```
    srand(time(0));
    initTable();
    Node * p = newNode(25);
    show(p);
    printf("%d\n", hashCode(p->key));
    put(p);
    p = newNode(45);
    show(p);
    put(p);
    printf("%d\n", hashCode(p->key));
    put(p);
    p = newNode(73);
    show(p);
    put(p);
    printf("%d\n", hashCode(p->key));

    showTable();
    printf("%s\n", get(45));
```

The output is:

```
Key 25 - value EXVOPJWDK    (make 3 key value pairs and insert into table)
2                           (the hash code)
Key 45 - value GIHHFIHGO
4
Key 73 - value ZKHLVNPMS
7
Cell 0 Null                         // the table contents now
Cell 1 Null
Cell 2 Key 25 - value EXVOPJWDK
Cell 3 Null
Cell 4 Key 45 - value GIHHFIHGO
Cell 5 Null
Cell 6 Null
Cell 7 Key 73 - value ZKHLVNPMS
Cell 8 Null
Cell 9 Null
GIHHFIHGO                           // value of key 45 read out of table
```

Collisions

A collision is when two different keys hash to the same location.

For example, using the scheme above:

```
    srand(time(0));
    initTable();
    Node * p = newNode(25);
    show(p);
    put(p);
    p = newNode(45);
    show(p);
    put(p);
    p = newNode(48);
    show(p);
    put(p);
    printf("%d\n", hashCode(p->key));

    showTable();
    printf("%s\n", get(45));
```

Output:

```
Key 25 - value PLQQFIYWR
Key 45 - value QMQTRMZHI
Key 48 - value RYIZWIKEX
4
Cell 0 Null
Cell 1 Null
Cell 2 Key 25 - value PLQQFIYWR
Cell 3 Null
Cell 4 Key 48 - value RYIZWIKEX
Cell 5 Null
Cell 6 Null
Cell 7 Null
Cell 8 Null
Cell 9 Null
RYIZWIKEX
```

Keys 45 and 48 collide - they both hash to cell 4, so the 48 over-wrote the 45, and we get the wrong value out.

So our scheme is too simple, and we must have a way to deal with collisions.

Three possibilities are

1. Having linked lists with roots at each array location. At a collision, we just insert the new key-value pair at the head of the list. On retrieval, we must do a linear search of the list (which gets slower as the list gets longer).

2. Store the value at the next empty location. In our example the 48 would have been stored at 5

3. As (2), but not the next location - make hash +5.

We modify our code using option (2). We have to change the put and get:

```
void put(Node * ptr) // insert node in array
{
    int where = hashCode(ptr->key); // use hash code
    while (hashTable[where]!=NULL) // look for next free location
        where=(where+1) % TABLE_SIZE;
    hashTable[where] = ptr; // put it there
}
```

and

```
char * get(int keyValue) // fetch value given a key
{
    int where = hashCode(keyValue);
    if (hashTable[where]==NULL) return "Not present";
    while (hashTable[where]->key != keyValue)
    {
        where=(where+1) % TABLE_SIZE;
        if (hashTable[where]==NULL) return "Not present";
    }
    return hashTable[where]->value;
}
```

Sample run:

```
    initTable();
    Node * p = newNode(25);
    show(p);
    put(p);
    p = newNode(85);
    show(p);
    put(p);
    p = newNode(87);
    show(p);
    put(p);
    p = newNode(88);
    show(p);
    put(p);

    showTable();
    printf("%s\n", get(88));
    printf("%s\n", get(24));
```

Output

```
Key 25 - value FWEXEUWEH
Key 85 - value WAJFUPRMM
Key 87 - value NMWZYJVCQ
Key 88 - value JDFBLCFII
8
Cell 0 Key 88 - value JDFBLCFII
Cell 1 Null
Cell 2 Key 25 - value FWEXEUWEH
Cell 3 Null
Cell 4 Null
Cell 5 Null
Cell 6 Null
Cell 7 Null
Cell 8 Key 85 - value WAJFUPRMM
Cell 9 Key 87 - value NMWZYJVCQ
JDFBLCFII
Not present
```

85 went in at 8, 87 at 9, and wrapped around to 0.

Choice of hashing function

We cannot avoid collisions. The set of all possible keys is usually effectively infinite, and we have limited space in any array in a real device. At best, we choose a hashing function which minimises collisions (why? because collisions slow down insertion and retrieval).

Keys will have some probability distribution. All keys may be equally likely, but some key ranges may be more likely than others. The hashing function should ideally spread these so that all locations through the array have the same likelihood of being used.

Load factor

The load factor is the fraction of array locations which are used. Initially the array is empty and the load factor is zero. As data is inserted, the load factor increases. Collisions become more likely, and insertion and retrieval become slow.

Usually when the load factor reaches some value, perhaps 50%, the following is done:

1. A new, larger array is created.

2. A new hashing function is developed using this larger address space

3. Data is copied from the old array to the new one.

4. The old array is garbage collected.

This is a lot of processing, so it is done as infrequently as possible.