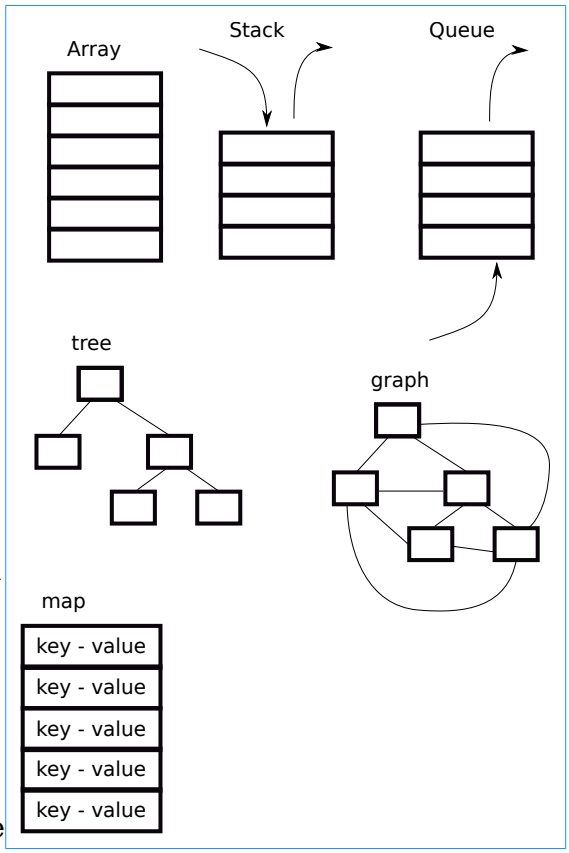
Data structures

Table of Contents

Most of the tests in this section involve writing programs about data structures. To do these, you need basic practical skills in a programming language.

Much of this is about the standard algorithms linked with data structures - so you need to know at least what algorithms are.



1 What are data structures?

A data structure is an arrangement or pattern of a set of data values. The diagram tries to illustrate some standard structures.

Many programming languages have an array as a structure built in to the language. Other structures are available in standard libraries, or coded by the programmer.

Need for data structures

Usually a program needs to deal with not just one data value, but a whole set of them. Examples are

- all the pixels in an image
- all the students in a class

- all the accounts in a bank branch
- all the roads on a map
- all the words in a word-processed document
- all the parts of a web page
- all the pages in a web site

Each of these is a collection of data. We need a way to arrange all the data items, together somehow, in some structure, that we can handle in software. Different standard structures are available, suitable for different *use cases*. The pieces of data in the structure are called *elements* or *nodes*.

A *static data structure* is fixed in size. It cannot grow or shrink during execution. For example in a card game we might want a data structure for the card deck. There are always 52 cards in the deck, so a static data structure would be good.

A *dynamic data structure* can change in size during execution. In a card game the number of cards in a hand might change as cards are dealt and played. So a dynamic data structure would be needed here.

Key-value pairs

Usually each data item 'thing' in a data structure is a *record*. The record has a key field, and other data fields, referred to as values. The key field needs to be unique, different for every item.

For example, a data structure might contain data about students. A sample record might be:

101257	Roxanna	Begum	07748 861 001	15.1.2001	..
Student number	First name	Last name	Phone	Date of birth	More fields..

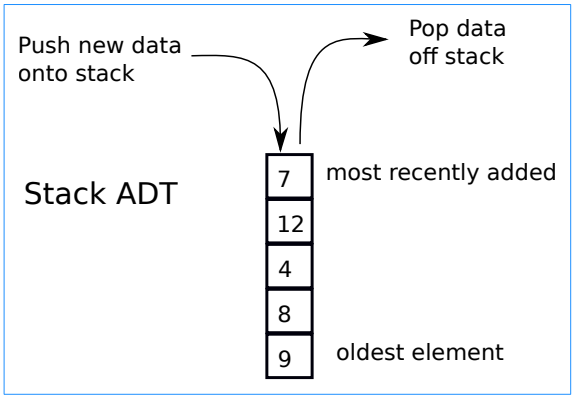
The key field is in yellow, and value fields in blue. Note we cannot use personal names as key fields, because we can have more than one student with the same name.

We usually use the data structure to find the value fields of a record with some key. For exampe we might need to know the phone number of student with ID 100753.

We will often just use a single data value – a number or a string – in our data structure examples. In the real things, these will be records.

ADTs and implementations

An *abstract data type* (ADT) is defined by what it can do - and *not* how it works. For example, a stack is like a pile. You can put things on top of the pile (push a value) and take a value off the top (pop). Push and pop are the only two things a stack can do. This is what a stack is.



For example, if we push 5 values onto a stack, 9, 8, 4, 12 and 7, with 9 first and 7 last, we get the stack as shown. Then if we pop a value off the stack, we get 7 - the one most recently added.

That example has integers as the data type. Any data type could be used.

In actual programming, we need to *implement an ADT* – actually code it. For example we implement a stack ADT by writing code to carry out push and pop operations. We can do this in different ways, such as using by an array, or a linked list. These are different implementations of the stack ADT.

In practice a language library often has implementations of standard ADTs built-in, and these are normally used in real code. But it is a useful learning task to write our own versions to see how they can work.

ADTs are an example of *abstraction* - focussing on *what* something does, not *how* it does it.

Memory and file storage

One basic aspect of implementations is whether the data structure is in memory or in a file. We can hold data structures in *memory* or in *files* on a non-volatile medium such as disc. Memory is much faster, but file storage can be much larger.

Software cannot access file data directly. It must be read into memory, and it can be processed there by program instructions, and if needed, written back out to the file. A memory area used to hold data for this is usually called a buffer. We usually do not need to read an entire file into memory at the same time - just a part.

Most of the data structures discussed here, like stacks and queues and trees, are in memory.

Fields, records and files

A file is a collection of data usually held on a *non-volatile medium* – one that keeps data when power is switched off.

Files are either ‘binary’ or ‘text’. In a text file, data is represented as characters. In a binary file, data is held some standard format. Examples of binary files are image files (like a jpg or png) or a sound file like an mp3, or a video like an avi.

Both binary and text files are digital, so both are streams of bits.

Text files often consist of a sequence of *records*, each record having a set of *fields*. For example a file might contain data on customers. A customer record might have fields for customer ID, name, address, phone number, date of birth and email address. The customer ID would be unique (each customer has a different ID), and would be the *key field*. The file might have thousands of records.

To use a file in code, we have to

1. open the file
2. read or write to it
3. close it

File access is actually done by the underlying operating system. This keeps track of different files by having a file handle or file channel for each open file. Programs in a high level language access files through the operating system facilities.

Each file handle uses memory. If we open files but do not close them, we will eventually run out of available file handles, which are always limited.

Test

1. Explain what a data structure is.

2. What is the difference between a stack and a queue?

3. What is a key field?

2 Arrays

An array is a basic type of data structure. An array is a set of storage locations or boxes, with each box having a number, called the *index*. We can read or write a box using its index.

In many languages indexes are enclosed in [square brackets], and the numbering starts at 0 not 1. Arrays are often fixed in length and cannot grow or shrink during execution.

Usually all the elements of an array have the same type.

For example

```
int myData[100];   // declare myData to be an array of 100
                   // ints
myData[0] = 23;    // store 23 in the first location
```

```
myData[9] = 17;    // store 17 in the 10th location
y = myData[7];     // read contents of index 7 and store in y
```

An array is an ADT. The special thing about an array is that we can access them through an index, usually a number. That means we can read or write them through an index.

But in many languages an array is a built-in data structure, implemented as a single block of memory, with each element next to each other. This makes it simple for the runtime to know where to find where an element is. For example, suppose an array starts at address 5000, and has 100 ints, each 4 bytes long. The array will occupy 400 bytes, from 5000 to 5400. Index 0 is at 5000. Index 10 is at address 5000 + 10 X 4 = 5040

If the elements in an array are themselves arrays, we have a two dimensional array.

A two dimensional array might be used to hold a matrix.

Beyond two, we can have multi-dimensional arrays.

Test

Write a program, in any language, which will

1. Declare an array of 100 integers
2. Use a loop to fill it with multiples of 3 - so, 1, 3, 6, 9, 12 and so on.
3. Use a loop to print it out.

3 Dynamic data structures

A static data structures is fixed in size in the program code. It stays that size while the program executes. An array is usually a static stucture.

For a dynamic data structure, we can add, delete and move nodes in the structure as the program executes. We need a way to get more memory to make a node, and a way to release the memory when we delete a node.

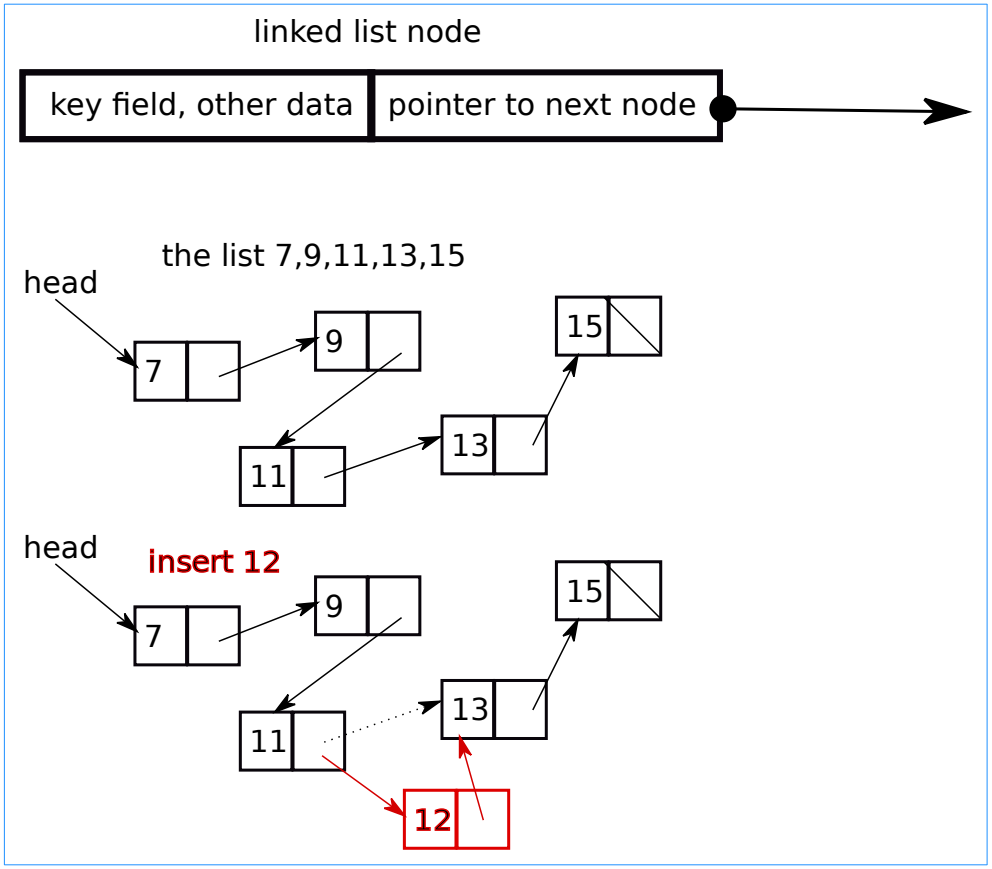
Program code needs to:

- Define a suitable structure for each node. There would be a field for a key value, and maybe other data fields. And pointer fields to other nodes, according to what data structure it is a part of. In C this would be a struct. In Java and JavaScript and Python, this would be a class.
- Have a way of creating a new node. In C this would use calloc or malloc. In Java we would say new..
- Create a new, empty structure.

- Have functions or methods to carry out the required algorithms relating to the structure - like adding a new node, searching for a node, and deleting a node.

Linked list

One dynamic structure is a linked list. This is a set of nodes, each one linked to the next in line. The nodes are not next to each other in memory. They are wherever the memory manager finds new free memory. The pointer lets us find the next node:



The last node in the list (15 in this example) has a special value of the next pointer, often called *null* - meaning a pointer to nowhere.

We have a pointer to a node, often named 'head', which points to the first node in the list. For a new empty list, head is null.

We can traverse a linked list. This means to start at the head, then follow the pointer to the next node, and the next, and so on, until we reach the end. This visits each node in turn, maybe to search for a data item, to output each node, find the smallest, or whatever. The pseudo-code would be

```
where = head // where is a pointer to a node - start at head
while where != null
  visit(where) // print it out or whatever
    where=where.next // change where to point to next node
```

We create a new, empty list by simply saying

```
head = null
```

To add a new node to a list, at the head, we make a new node, make its next field point to the current list head, then change the head to point to the new node. If the data in the new code is d, the pseudocode is:

```
newNode = new node // set up new node
newNode.data=d       // newNode is a pointer to a node
newNode.next = null
newNode.next=head    // next points to current head
head=newNode         // change head to this node
```

This also works if the list is empty.

Test

Write a program in any language which

1. Defines a suitable node type (a struct in C, a class in Java or JavaScript or Python)

2. Has a function or method to add a new node at the head of a list

3. Has a function or method to traverse a list and output the data

4. Insert values 3, 7, 2, 8, 1 nto a new list, then traverse it.

Test it.

4 Queues

A queue is like cars queueing at traffic lights. The first one to leave will be at the head of the queue - the first one to join the queue.

A *queue is a FIFO list* – first in, first out.

If we add 4, 6, 2 and 3 to an empty queue, in sequence, then remove items, we get 4, 6, 2 and 3 out. First in, first out.

A queue might be used for a keyboard input buffer. The user wants keystrokes to be processed in the order that they are typed.

A queue is an ADT. It might be implemented by using an array, or a linked list, or other methods.

In a *priority queue*, elements have a data value, and also a priority. Items with a higher priority come out of the queue first.

A *circular queue* is a type of queue implementation. A block of memory is used to store the queue. This might be an array. There are two pointers into the array, head and tail. Data added to the queue goes in at the head, which is then incremented. Data is removed from the tail, which is also incremented. So in use, the queue moves forward through the array. Eventually the head will hit the array end – when it wraps around back to the array start.

Similary for the tail. It might be that the head catches up with the tail, in which case the queue is full.

Circular queues are also called ring buffers. A keyboard input buffer might be a circular queue. Keystrokes go into the queue as keys are hit, and removed as the application ‘uses’ them.

Test

Write a program which implements a queue as a linked list, following these steps:

1. Define a node structure.

2. Define a queue type. This just needs 2 pointers to nodes - head points to the first node in the queue, and tail to the last. Data is added at the head, removed from the tail.

3. Write a method to add a new node at the head.

4. Write a method to traverse the queue, printing it out from head to tail. This is for testing purposes.

5. Write a method to remove an item from the queue - at the tail.

6. Insert 8,2,7,6 into the queue. Then remove data and print it, until the queue is empty.

5 Stacks

A stack is a LIFO structure – *last in first out*.

A stack is an ADT which has two operations -

- *push* a new item onto a stack, and
- *pop* the value off the top of the stack, removing it.

Sometimes there is also a *peek* method, which returns the top of stack, but does not remove it.

So if we push 1,2,3,4 onto a stack, then pop values off, we get 4,3,2,1.

Stacks are used in connection with subroutine calls, for passing parameters, return values and addresses.

Processors usually have push and pop native instructions, and a stack register to track where the top of stack is in memory.

The ADT could be implemented in an array or a linked list.

Test

Write a program implementing a stack in an array.

1. Have a constant STACKSIZE. Start with STACKSIZE = 10

2. Declare an array of integers with size STACKSIZE.

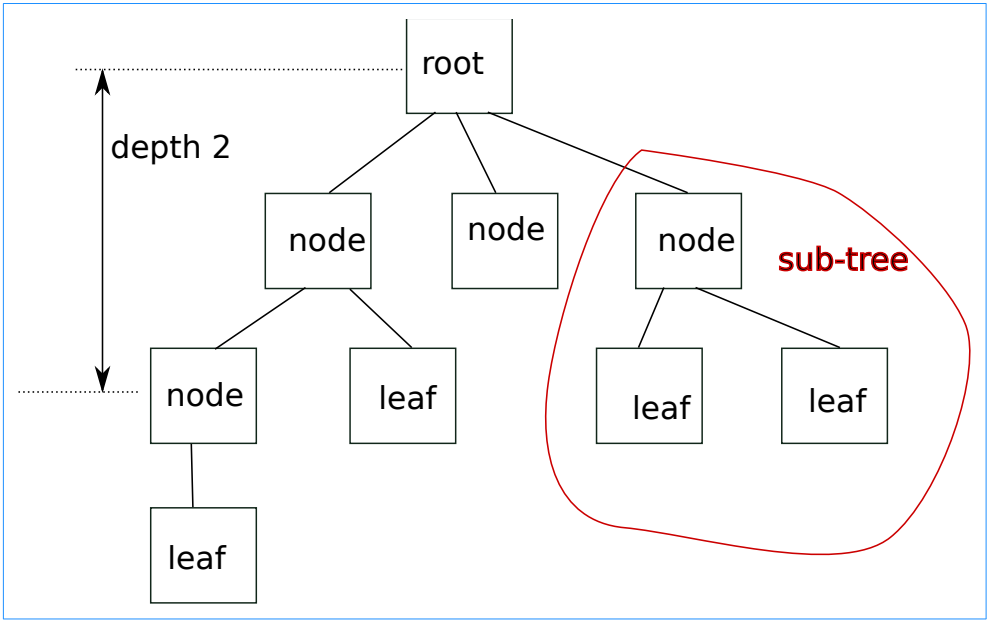
3. Have an integer variable stackTop, representing where in the array the next value will go. Initially stackTop=0, for an empty stack. If stackTop == STACKSIZE, the array is full.

4. Write push and pop methods. push may fail if the array is full. pop will fail if the array is empty.

5. Push 1,2,3,4,5. Then pop values until the stack is empty.

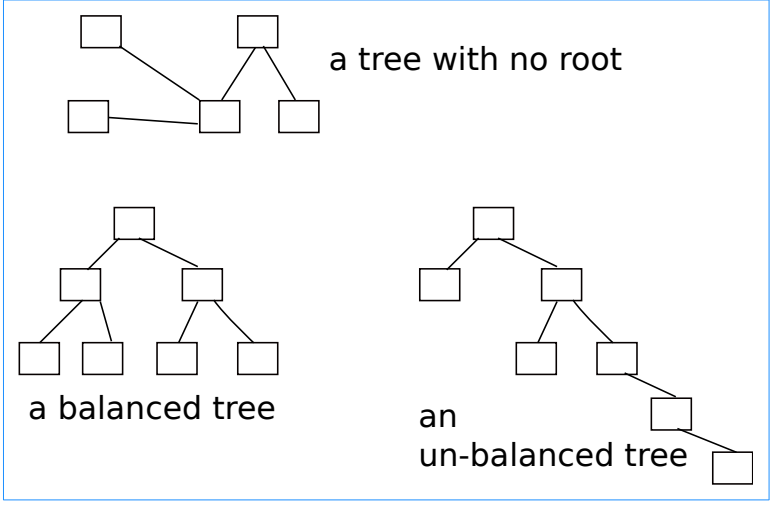
6 Trees

Here is a tree:



This is a set of nodes. Most nodes are linked to some 'lower' nodes. Nodes at the bottom, not linked to others, are called leaf nodes. The root node is at the top, and there are no nodes which link to it.

We can pick out a set of nodes, as shown, which are themselves a tree - this is a sub-tree.

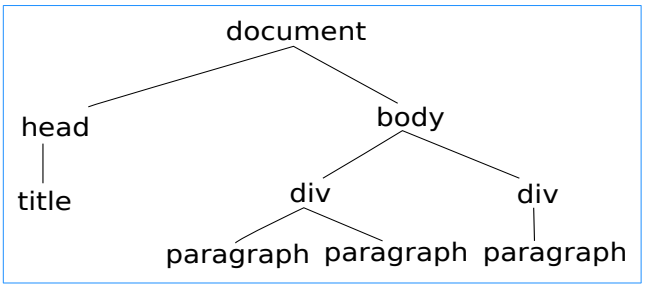


In a *binary tree*, each node has at most 2 nodes linked to it. The example above is not a binary tree, since the root has 3 sub-nodes.

There are no loops in a tree - we cannot follow links and cycle back to where we started. A tree is *acyclic* - no cycles.

The DOM

One use for a tree is the DOM - the document object model. This is for representing the structure of a web page, which will be an html file. A simple web page might have two parts – a head and a body. The head might contain the title. Inside the body we might have two divs. The first div might contain two paragraphs, and the second div one paragraph. Here is the tree for such a page:

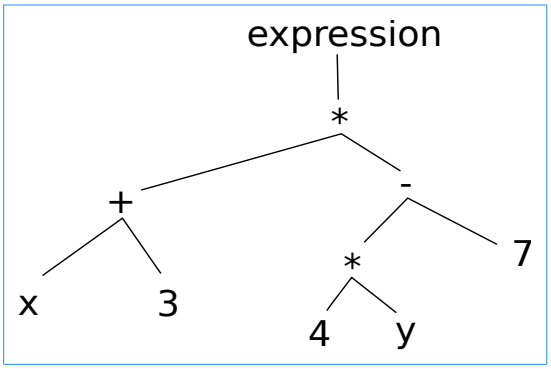


In JavaScript code we can access teh DOM tree, search it, add and change and remove nodes, and so alter the web page dynamically.

Abstract syntax tree

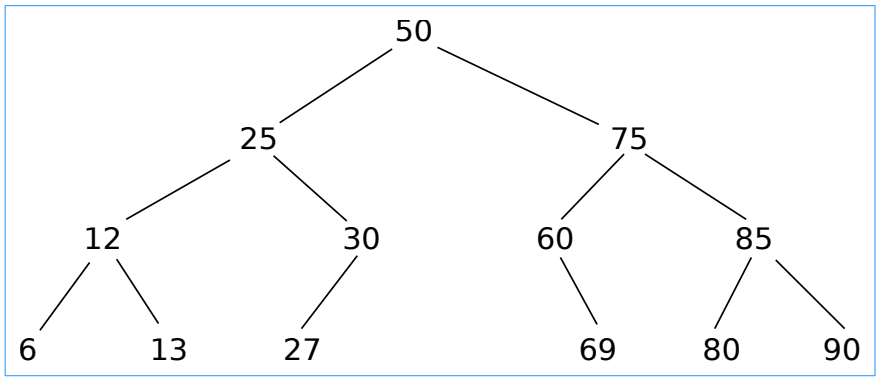
Another use is an 'abstract syntax tree' to represent some program code. For example suppose some program contains the expression

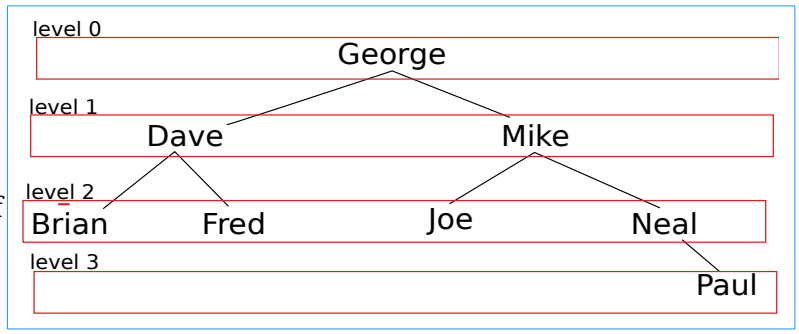
(x+3)*(4*y-7). As a tree this is as shown. An interpreter or compiler will process source code and generate trees like this to represent valid expressions.



Binary Search Tree

Another use is a *binary search tree (BST).* This is a binary tree (maximum 2 child nodes) which is *ordered*. That is, smaller data goes left and larger goes right. For example:





Here is a BST for some names, ordered alphabetically.

In a real application this might be a store of personnel data. Each node would hold a record on an employee, with fields like name, address, department, pay grade and so on. Here we just show first names.

Suppose we search this for Joe. We start at the root. Joe comes after George, so we go right.

Mike comes after Joe, so we go left.

This node matches Joe and he is found. We can read his paygrade or whatever we need.

Now search for Barry. Start at the root. Barry comes before George, so go left. Barry is before Dave, so go left. Barry is before Brian.

But there is no node to the left of Brian – the left pointer from Brian is null. So, Barry is not in the tree.

Why use a BST? Because it is *fast to search*. Each comparison (an 'if') takes us down one level. A two level tree has up to 2 nodes in the lowest level. A three level tree has 2 X 2 = 4. A four level tree has up to 8. A ten level tree has 512, and we can search it with just 9 comparisons. This is much faster than a list in an array, for example. On average we would find it half way through, after 256 ifs. So the search time for a BST is O[$\log_2 n$]

The algorith to search for target is to start at the root, and go right or left, until we either find it, or reach a leaf node:

```
where=root    // where is a pointer to a node
while where != null and where.data!=target
  if where.data < target // go right
   where =where.right
```

```
  else   // go left
   where=where.left
if where == null : not present
else target at where
```

The algorithm to insert a value is similar, but we need to track the previous node we came from. At the end, we need to link this with the new node:

```
// newNode and where and previous are
// pointers to nodes
// start by setting up the new node
newNode = new node
newNode.data = value
newNode.left=null
newNode.right=null
// find where to link it
where=root
while where!=null
       previous=where
   if where.data < target // go right
   where =where.right
  else   // go left
   where=where.left
// now finished loop - link previous to newNode
if previous.data>value
  previous.left=newNode
else
  previous.right=newNode
```

To traverse the BST, in key field order, we do this recursively. We start at the root, traverse the left sub-tree visit the root, and traverse the right sub-tree:

```
function visit(node)
  if node.left!=null
    visit(node.left)
  print(node.data)
  if (node.right!=null)
    visit(node.right)
```

and we start this by saying

```
visit(root)
```

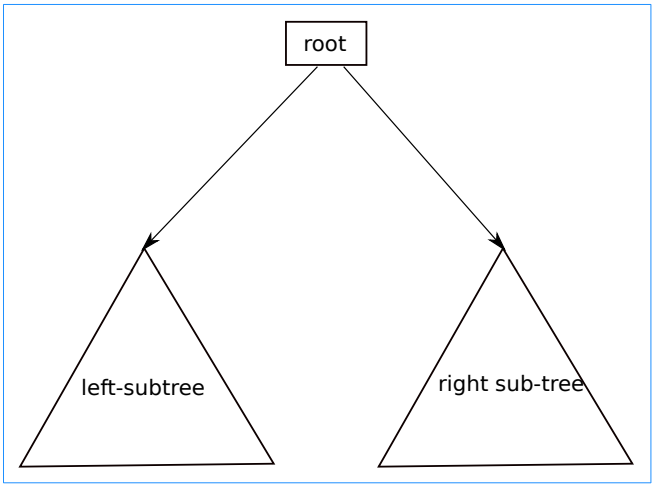
Test

In any language, implement a BST:

1. Define a tree node structure
2. Write a method to insert a value
3. Write a method to traverse the tree
4. Insert some data, and traverse it

7 Tree traversals

To traverse a data structure means to follow links from node to node until every node has been reached.



Tree traversals - in-order

With a binary tree, as shown, three ways to traverse it are

1. left - root - right (*in-order)*
2. root - left - right *(pre-order)*
3. left - right - root *(post-order)*

These are *recursive*. The left sub-tree for example, is itself a tree, and we need to traverse it.

The algorithm for an in-order traversal is to *visit* the root, where visit is:

```
vist(node)
    if node.left != null, visit(node.left)
    output node (see below)
    if node.right !=null visit(node.right)
```

The 'output node' might be replaced with whatever we want to do with the data at each node - place them in a queue, do arithmetic with them or whatever

An in-order traversal corresponds to the way a BST is ordered. So a *use for an in-order traversal* is to output a BST in order of keys.

Pre-order tree traversal

Here we visit the root, then recurse to the left sub-tree, then the right sub-tree.

The root is visited *before* the left and right.

A *use for this is to copy a tree*. We clone the root, link its left field to a clone of the left, then link its right to a clone of the right node. This is the root of the tree copy.

Post-order tree traversal

This is

left
right
root

The root is visited *after* the left and right. This is used in *abstract syntax tree* evaluation.

Test

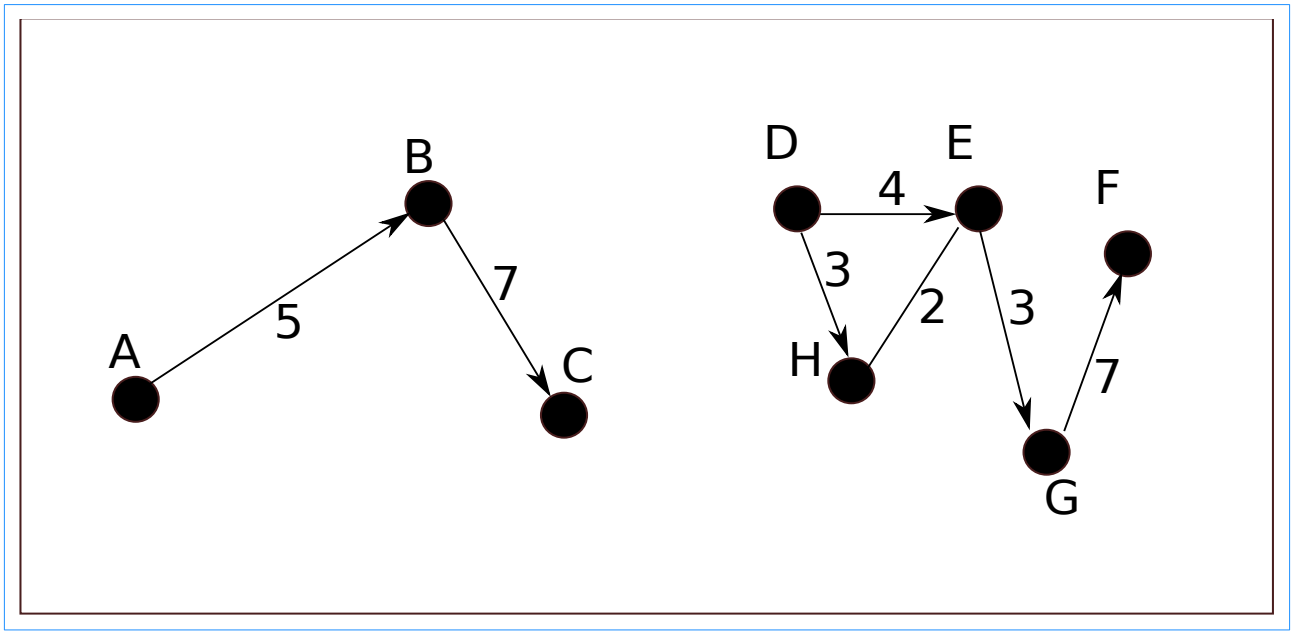
Add a 'copy' method of your BST implementation, using a pre-order traversal.

8 Graphs

This is *not* the same as an x-y graph in mathematics. That is a Cartesian graph usually showing some function of x sideways and y upwards. These graphs are different.

A graph might be used to represent roads connecting towns on Google maps. Or computers and printers on a network. Or underground stations on the Paris Metro. Or shipping routes between ports.

For example:



Graphs are not images. The diagram is a diagrammatic representation of a graph (like a Tube map is not a picture of the Tube network).

The graph has some black circles linked by lines. The circles, labelled A to H, are called *nodes* or *vertexes* (singular : vertex). The lines are called *edges* or *arcs.* These edges have arrows – they are one way links. These are called directed edges, and we have a *directed graph*. In some graphs the edges do not have arrows, and the graph is *undirected.*

The lines have numbers. These might be distance in miles, or travel times in minutes, or cost of a plane ticket, or the bandwidth on that part of a network. Whatever the number means, it is the edge weight. This is a *weighted graph*. Some graphs are not weighted.

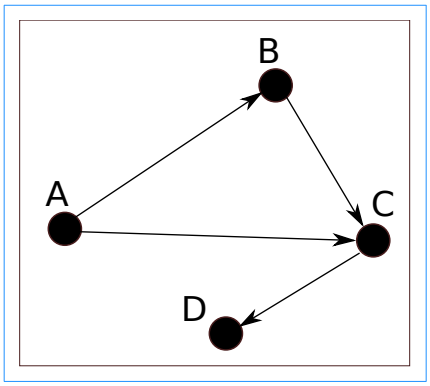
Nodes D E and H form a loop – called a cycle. A graph with no cycles is called an *acyclic graph*.

A B and C are linked, but nothing links these with D to H. This graph is not connected like road maps on two separate islands. In a *connected graph* there is a link through intermediate nodes between any two nodes.

A tree is a type of graph – a *connected undirected acyclic graph.*

How can we represent a graph in a computer? We want to be able answer questions like 'how far from x to y' or 'what is the fastest route from a to d going through p'? An actual image would not be much help.

One technique is to use an *adjacency matrix.* This is a square table, with a row and column for each edge, and the cell showing if the two edges are linked (and maybe the weighting if they are).



For example, in this graph, the adjacency matrix would be

	A	B	C	D
A		1	1	
B			1	
C				1
D				

The empty cells actually contain 0. The cell in row B column C is 1 because there is an edge B to C. If the edges are weighted, the cell can contain the weight. As a matrix, this might be held in a 2D array in memory.

An alternative is an *adjacency list*. This is a list of nodes, and for each, a list of nodes linked to it. So for the same graph:

A	B, C
B	C
C	D
D	

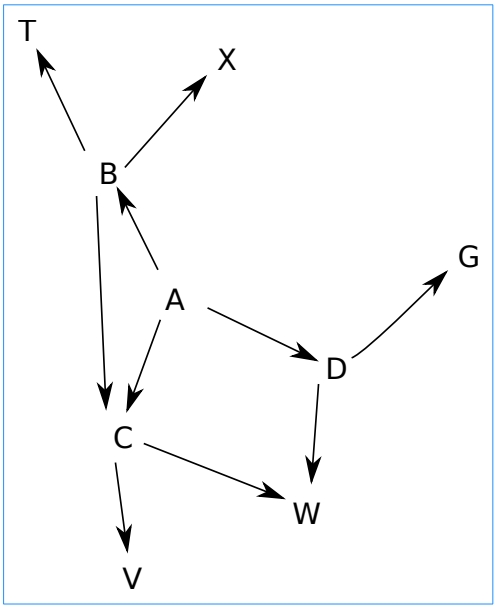
With the matrix we can quickly find out if any two nodes have an edge between. But the list uses less memory, especially for a graph with a small number of edges.

9 Graph algorithms

Graph traversals – breadth-first

One traversal method is *breadth-first.*

Suppose this is our graph, and we start from node A:



In breadth-first, we start by visiting every node next to the start. Then every node next to those, and so on.

More exactly:

push start node into a queue
while queue not empty
    pop queue head and *visit* it
    put every node next to this into the queue, if it is not already in

So here, starting at A

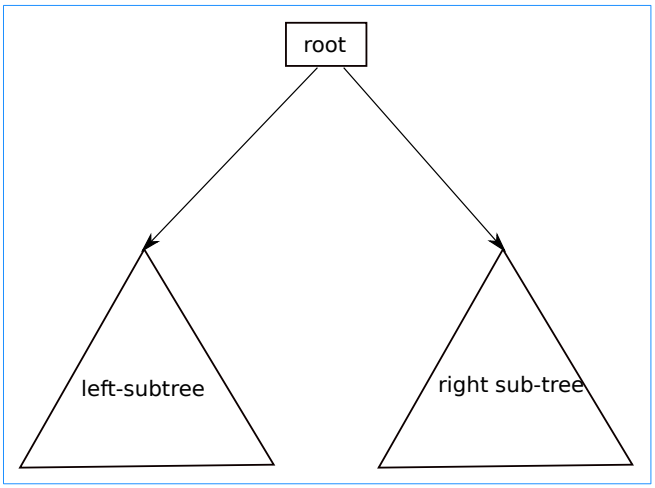
visit A
Put B, C D in queue
Pop node, get B. visit it. Add nearest to queue. Queue is now C D T X (C was already in)
Pop node, get C. Queue is D T X V W
Pop D. Queue is T X V W G
Visit T
Visit X
Visit V
Visit W
Visit G
Queue empty, so end.

The sequence was A B C D G T X V W G.

The order depends on the order nearest nodes are pushed. For example we could have pushed D, C, B as nearest to A.

This algorithm is used to *find the shortest path* from one node to another (see Dijkstra later).

Infix and reverse Polish



There are a set of ideas relating to expressions and their evaluation. Programs contain expressions, and program execution means evaluating those expressions.

A *expression* is something which can be worked out - for example 2+3*4

Evaluating an expression means working out its value

Expressions contains *operators* - like + - * and /. There might also be functions like sin and cos and log.

And *operands* - things the operators work on. These might be integers like 2 and 3, floats like 3.141, variables like x and y and so on.

Operators have *precedence levels*, which means what order they should be carried out in. For example * normally has higher precedence than +. So in 2+3*4, the * should be done first, so this is 14 not 20. The fact that the * is to the right of the + means we cannot just work left to right. So to work out 2+3*4, simply going left to right would mean 2+3 = 5, *4 = 20, which is the wrong answer.

The usual precedence levels are BODMAS - brackets division multiplication adddition subtraction.

Expressions can be written in different notations. The usual notation is called *infix*. The operator comes in between operands - like 2 + 3

In *prefix,* the operand is before (pre) the operands, like + 2 3

In *postfix,* the operator is after (post) the operands, like 2 3 +. Postfix is also called *Reverse Polish* or RPN. This is after Jan Łukasiewicz, a Polish logician.

Reverse Polish is used, internally within interpreters and compilers, because it is fast to evaluate, and brackets are not needed.

In conversion of infix to reverse Polish, the order of operands is unchanged. Some examples:

Infix	reverse Polish
2 + 3	2 3 +
2 + 3 * 4	2 3 4 * +
2 * 3 + 4	2 3 * 4 +
2*(3+4)	2 3 4 + *
(2+3)*(5-2)	2 3 + 5 2 - *

In 2 3 + 5 2 - *, we can work this out, left to right, as follows:

2 3 + : apply the + to the two things before it : 2 and 3 : and replace with result

5 5 2 - : apply the - to the 5 and 2

5 3 * : apply the * to 5 and 3

15

Check that the brackets in (2+3)*(5-2) are not needed in 2 3 + 5 2 - *

10 Maps

A map is an ADT. It contains a set of key-value pairs. We can insert a key and a value pair of data items. Then we can ask the map to fetch a value, which as paired with a given key. It has 2 basic operations - to put a key and a value, and a get, which has a key as a parameter, and returns a matching value. So for example we might say

```
myMap.put(45,"John")
myMap.put(55,"Jane")
myMap.put(23,"June")
```

This puts into the map the pair with key 45 and value 'John'. Also the pair 55 and Jane, and 23 and June.

Then if we call

```
myMap.get(55)
```

Key 55 is linked to value Jane, so that is what this would return.

In this example the key type is integer, and the value is a string, but any data types can be used.

A map corresponds to a function, before the set of keys and the set of values.

For example, this is a map where the key is a character and the value is an integer. Key s gets value 3. Key h gets value 1.

key	value
t	3
h	1
i	1
s	3
a	1
e	1

Suppose, for example, we need to count how many occurences of each letter there is in a text – which we might do, for example, in cryptography.

The algorithm would be:

for each character in the text..

if it not in the map, add it, with value 1

else (it is in) add 1 to its value

If we do this on

this is a test

we get the map shown above.

In Python, a map is called a *dictionary*.

In JavaScript, all objects are *associative arrays*. This is a type of map which looks like an array, but the index is not only an integer. For example we can say

```
var myObject=[]; // object with no fields
myObject['id']=27; // key is 'id', value 27
myObject['name']="John"; // key 'name', value 'John'
console.log( myObject['id']); // get value of key 'id'
```

In Java, Map is an interface, implemented by classes such as HashMap and TreeMap.

One way to implement a map ADT is using a BST. Nodes in the BST have a key field and a value field, and the tree is ordered on the key.

Test

Modify the BST implementation to work as a map, and write put and get methods.

11 Hash tables

A hash table is a type of data structure implementation, containing key-value pairs. It is one type of *map*, mapping keys to values.

We use a hash table if we want *very fast retrieval*. That is to *find and fetch keys very quickly*.

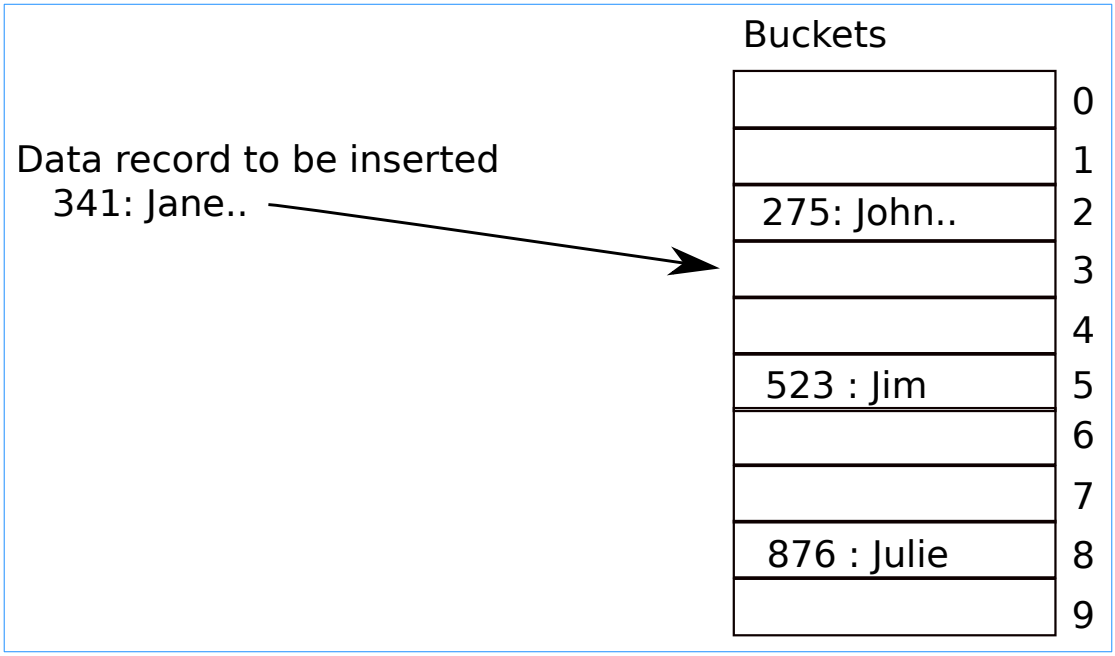
This works by *calculating where* they are. We do not search for keys. We calculate where they are.

The calculation is called the *hash function*.

Data is held in a set of storage locations, called buckets. A bucket can hold one record.

A very small simplified example is shown. We have ten buckets. The data has a key field which is a 3 digit integer, from 000 to 999. Th

ehashing function is simply to take the first digit of the key field. So key 197 hashes to bucket 1. Key 865 hashes to bucket 8:



The algorithm to obtain a record, given a key (say 523) is very simple.

1. Apply the hashing function to get the bucket (5)
2. Look in that bucket. If its empty, the data is not there. If its not empty – the record has been found

But – suppose we then insert a record with key 511. This also hashes to bucket 5. This is a *collision* – two key fields hash to the same bucket.

It is usually impossible to find a hashing function which will avoid collisions. They must be handled somehow.

One solution is to place overflow records into linked lists starting at the bucket hashed to, like this:

Then to find key 272, for example, we

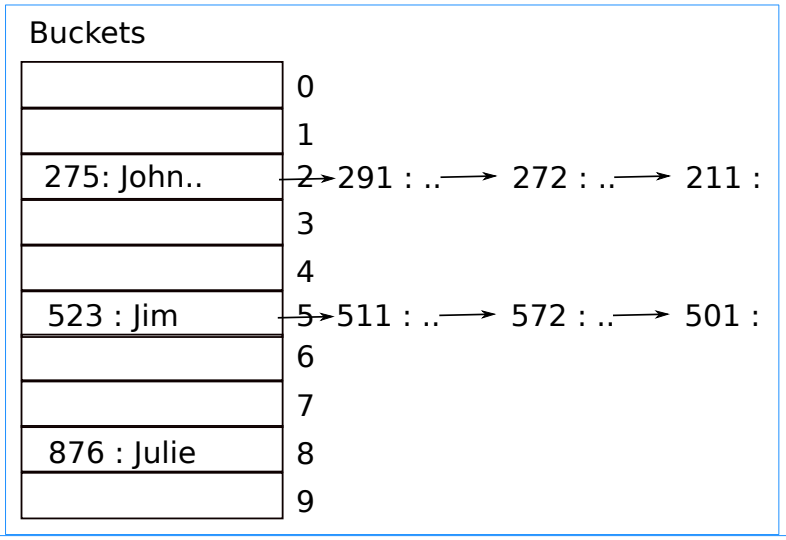
hash it – get 2

look in bucket 2

loop : if it matches, its found

else move to next node in the list

until end of list – not present



This works. But searching a linked list is *slow*. On average we find it half way along, and this might be a long list.

Another technique is to store it in the next available bucket. So to store 291, if bucket 2 is occupied, store it in bucket 3. Again retrieval involves a *slow* linear search.

The fraction of buckets occupied is called the load factor. So if half the buckets re occupied, the load factor is 0.5.

The higher the load factor is, the more likely a collision is, and the hash table will become slow.

We can fix this by *re-hashing*. This means:

1. Creating a new, larger number, of empty buckets.
2. Getting a new hashing function. This is often done using modulus (remainder) to get the hash into the correct rangle. For example if we have 1000 buckets, we have a hash mod 1000. This will be from 0 to 999 and will fit.
3. Move existing data into the new buckets.
4. Return existing buckets to free memory (garbage collect)

Since we now have more buckets, the load factor is lower, and the table faster. But doing the re-hashing takes time and memory.

Test

Write a basic hash table implementation. You need

1. A set of buckets - an array with size 100 or 1000. To start with, to simply testing, it could be size 10.

2. A hashing function, hashing key fields to bucket numbers.

3. A method of using the hashing function to insert key-value pairs. This includes a way of handling collisions.

4. A method of reading a value, given a key. This should include the possibility that the key is not in the map.