Topic 3 Fundamentals of Algorithms

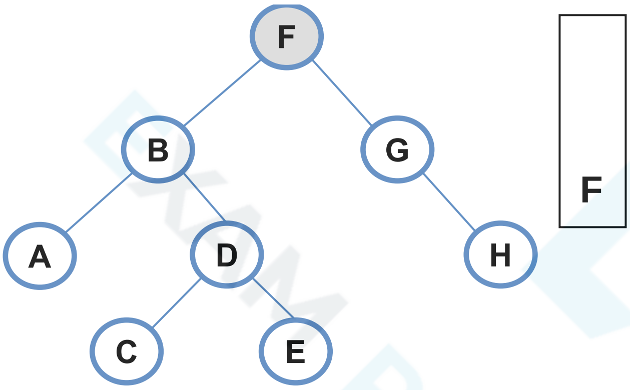
Graph Traversal

This process visits each vertex in a graph. There are two algorithms which can be used to do this; depth-first and breadth-first.

Depth-First

This approach is often used for navigating a maze and uses a stack.

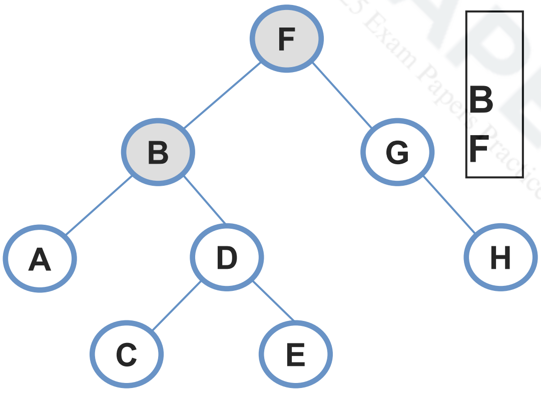
This approach works as far possible down one branch before moving on the next branch, then the next, then the next until all branches have been traversed. The various branches can be traversed in any order and a stack is used to manage backtracking as shown in the example below.



RESULT: F

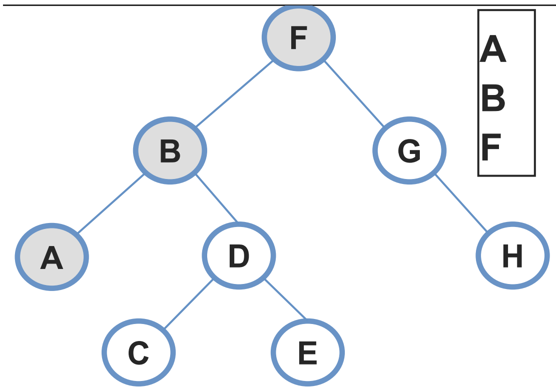
Nodes can be explored in any example, but in this case we will start from F.

The algorithm adds the first node (F) to the stack and checks the nodes connected to it.



RESULT: F B

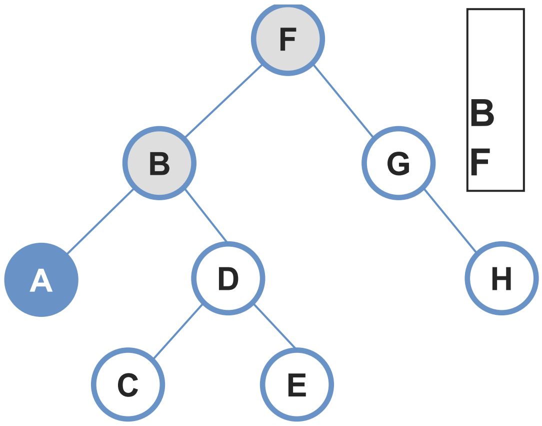
Nodes are always processed with the lowest first, since B lower than G, it is discovered first and added to the stack.



RESULT: F B A

The nodes adjacent to node B are observed, and with A being the lowest it is the first to be discovered.

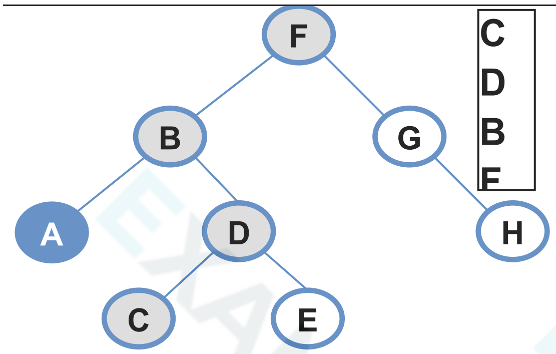
The algorithm moves to node A and adds it to the stack.



RESULT: F B A

Node A has no child nodes, and so can be thought of as fully discovered, meaning this particular branch has been fully explored.

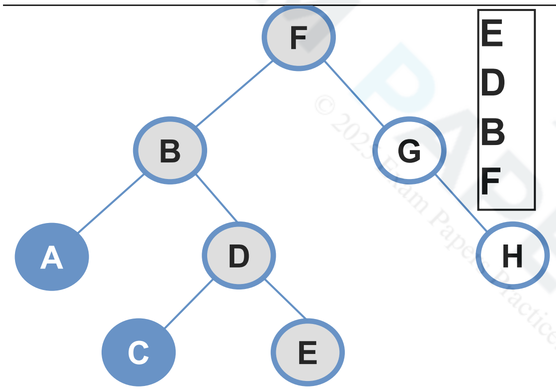
Node A is popped (removed) from the stack, and the algorithm moves back to node B



RESULT: F B A D C

The algorithm moves to node D, adding it to the stack and discovering child nodes.

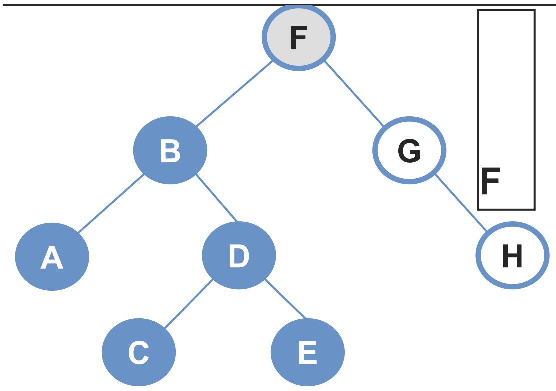
As before, they are processed lowest first and so node C is processed and added to the stack.



RESULT: F B A D C E

There are no child nodes connected to node C so it is marked as fully discovered and popped from the stack.

The algorithm moves back to node D followed by node E. Node E is added to the stack in place of Node C

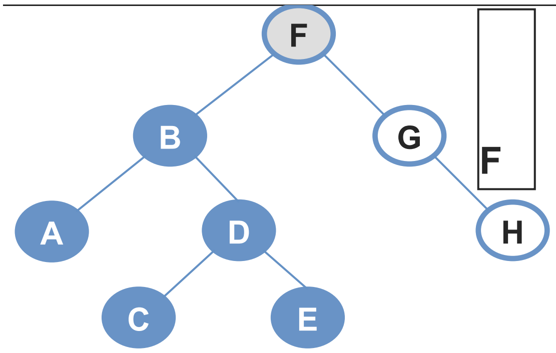


RESULT: F B A D C E

Since E has no child nodes it is popped from the stack and marked fully discovered.

This process is repeated for nodes D then B since both have no child nodes which have not been discovered already.

This leaves us back at node F

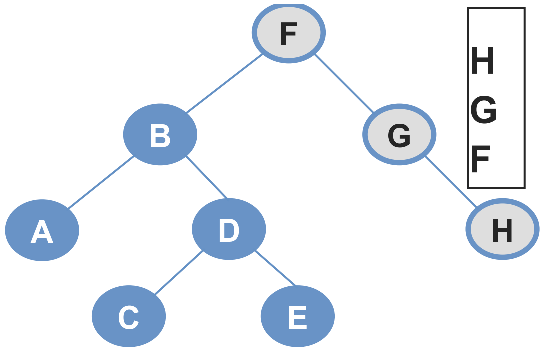


RESULT: F B A D C E

Since E has no child nodes it is popped from the stack and marked fully discovered.

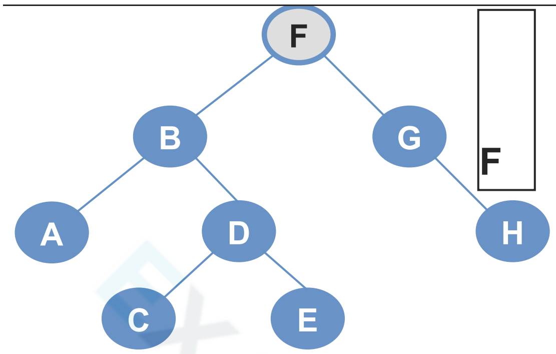
This process is repeated for nodes D then B since both have no child nodes which have not been discovered already.

This leaves us back at node F



RESULT: F B A D C E G H

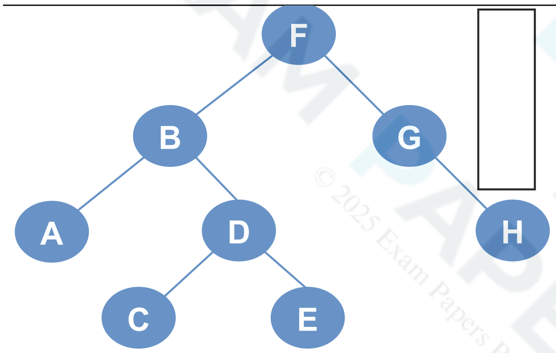
The algorithm now repeats the discovery process discovering first G then H



RESULT: F B A D C E G H

Node H has no child nodes and so is fully explored and popped from the stack.

The algorithm moves back to node G, it too has no further child nodes and so is popped from the stack.



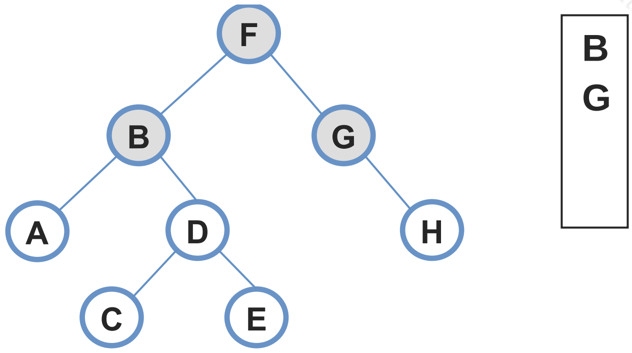
RESULT: F B A D C E G H

Finally, the algorithm discovers there are no further child nodes to node F so it is marked fully discovered and is popped from the stack.

The stack is now empty, meaning the tree has been fully discovered and the algorithm terminates.

Breadth-First

This approach works down the tree one level at a time, exploring all nodes at that level before moving onto the next. A queue is used to manage the search. This approach is useful for determining the shortest path.

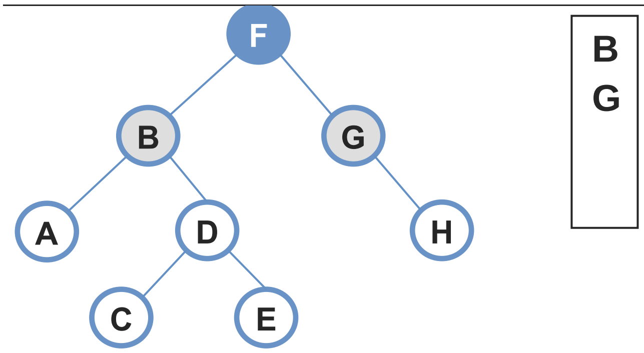


RESULT: F B G

Nodes can be explored in any example, but in this case we will start from F.

F is discovered.

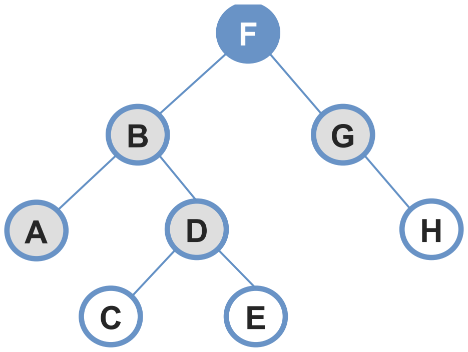
The nodes directly adjacent to F are then discovered then added to the queue and result in alphabetical order.



RESULT: F B G

Node F and its directly adjacent child nodes have all been discovered, and as such it can be described as fully discovered.

The algorithm moves onto the node at the top of the queue, in this case B

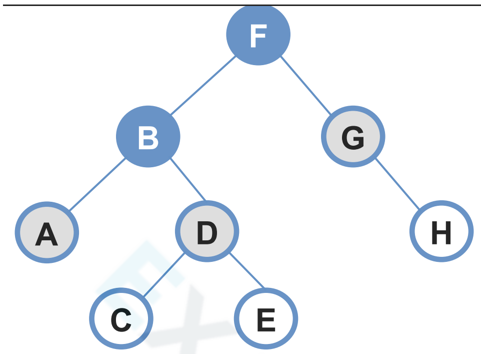


RESULT: F B G A D

G
A
D

Node B is removed from the queue.

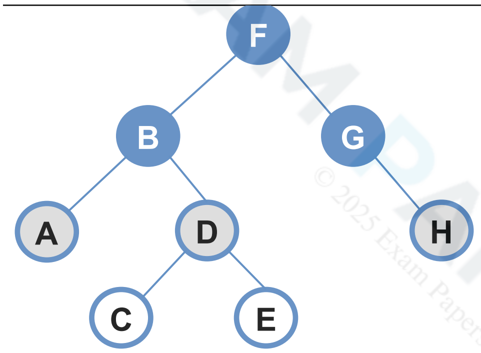
Node B’s child nodes are discovered in alphabetical order, they are added to the queue and the result



RESULT: F B G A D

G
A
D

Node B is now fully discovered.



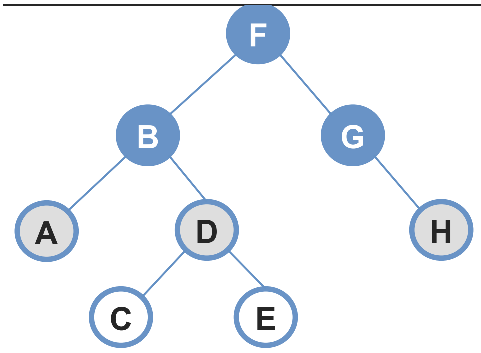
RESULT: F B G A D H

A
D
H

The algorithm again looks at the node at the top of the tree, in this case G.

The node is removed from the queue, and its child node H is discovered and added to the queue and result.

Node G is now fully explored since it has no child nodes which are not in the queue or result.



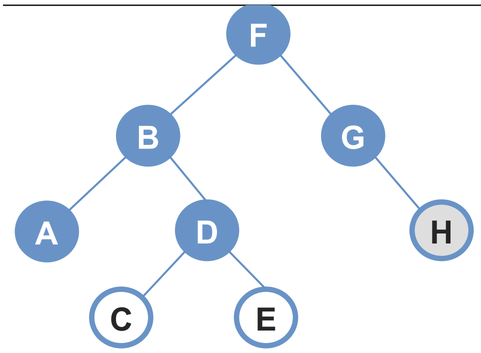
RESULT: F B G A D H

D
H

The algorithm again looks at the node at the top of the tree, in this case A.

It is removed from the queue but has no child nodes.

As such it is fully discovered.



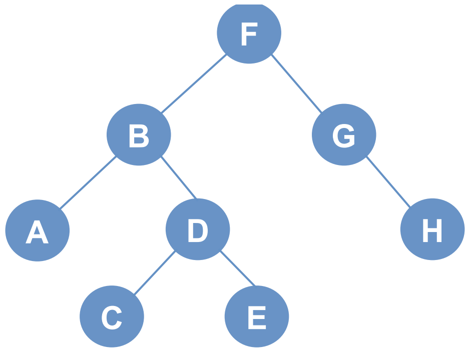
RESULT: F B G A D H C E

H
C
E

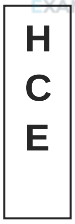
D is the next item the queue.

It had two child nodes, C and E which are added to the queue and result in alphabetical order.

D is now fully discovered and so removed from the queue.



RESULT: F B G A D H C E



The remaining items in the queue are explored in order.

Each has no child nodes so they are removed from the queue.

The tree is now fully discovered.

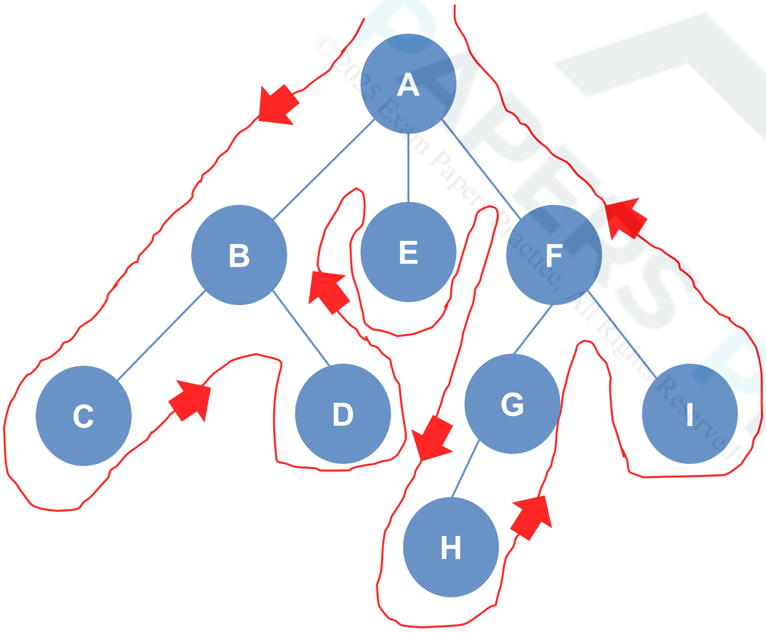
Tree Traversal

A Tree traversal algorithm works through a tree to discover and output all nodes within the tree. Tree traversal algorithms must start at the top of the tree and travel down the tree always keeping to the left hand side of the tree. There are three types of tree traversals:

- Pre Order
- In Order
- Post Order

Pre order and post order traversal can be performed on any tree, but an in order traversal can only be performed on a binary tree.

This example shows the route a tree traversal would take:



Pre Order Traversal

Pre order traversal can be performed on any tree, it is used to create a copy of the tree. The process follows these steps:

1) Mark the left hand side of every node in the tree.
2) Start at the root node and keep to the left hand side of the tree.
3) Whenever a mark (created in step 1) is passed, output the information contained in the node.
4) For each node, work as far down the tree as possible keeping to the left hand side of the tree.
5) Repeat the process moving to the next node to the right.
6) Once all child nodes have been explored, move up one level and repeat the process.

In Order Traversal

In order traversal can only be used on binary trees and will output the contents of the binary tree in ascending order. The process follows these steps:

1) The bottom of each node is marked.
2) The process starts from the left and works around the tree.
3) When a mark is reached, the details of the node are outputted.
4) For each node, work as far down the tree as possible keeping to the left hand side of the tree.
5) Repeat the process moving to the next node to the right.
6) Once all child nodes have been explored, move up one level and repeat the process.

Post Order Traversal

Post order traversal is used to make an infix to RPN (Reverse Polish Notation) conversion or to make a postfix expression from an expression tree.

Infix Notation

People will most commonly use infix notation, where the operand is either side of the opcode, but this can cause confusion as to the order of operations when working with longer equations.

For example, in the equation **9 + 1** 9 and 1 are the operand whilst + is the opcode. For a simple equation like this, the order of operations is clear and the answer is 10.

In a more complex equation, for example **4 + 6 / 2** 4, 6 and 2 are the operand whilst + and / are the opcode. Following the mathematical BODMAS order of operations, we would process the division first then the addition, giving an answer of 6.

However, we could instead add brackets to make the equation **(4 + 6) / 2** which would instead give the answer of 5.

Reverse Polish Notation (RPN)

Reverse polish notation uses postfix notation to write expressions. In reverse Polish notation operators are placed after the operands on which they operate. This removes the need for brackets and any confusion over the order of operations.

A postfix expression writes the opcode after the operand. When the opcode has both pieces of operand immediately proceeding it the operation proceeds.

RPN can be executed on a stack, making it ideal for interpreters such as Bytecode or Postscript which are based on stacks.

Infix Notation: 9 + 1
Postfix Notation: 9 1 +
Both the equations above give the answer 10.

Infix Notation: 4 + 6 / 2
Postfix Notation: 4 6 2 / +
Both the equations above give the answer 6.

Converting Infix to Postfix Notation

The Dijkstra's Shunting Yard algorithm uses a queue and a stack to convert expressions from Infix to Postfix notation. The example below shows how this works for the expression 2 + 4 x 8:

1) Use the rules of BIDMAS to add brackets to the expression: (2 + (4 x 8)
2) Move each infix operator to the very end of its set of brackets: (2 (4 8 x)+)

3) Remove the brackets: 2 4 8 x +

The key point is that the order of the operands must not change.

Converting from Postfix to Infix Notation

This process follows the reverse approach as shown in the below example based on the expression 2 4 8 x +

1) Work from left to right to find the first operator, move it one place to the left and place brackets around the resulting expression: 2 (4 x 8) +
2) Working left to right, move all the remaining expressions one by one following the same steps: (2 + (4 x 8)

Searching Algorithms

Algorithms are sets of instructions to complete a given task within a finite time. Searching algorithms are designed to check whether a list contains a given item, and at what position in the list the item is.

<u>Linear Search</u>

Linear search is the most straightforward algorithm but has relatively high time complexity and so is rarely used. It has one loop, giving a time complexity of O(N). The algorithm can be used against any unordered list.

The algorithm works by checking the first each item in the list one at a time against the target until either the target has been found or all items have been checked.

When coding a linear search, it is best practice to use a Do Until loop rather than a For Next loop to improve efficiency. If the target was at the beginning of the array, a Do Until loop would quickly locate the target then exit, whilst a For Next loop would continue to search despite the target having been found.

<u>Binary Search</u>

The binary search algorithm can only be used on sorted lists. Unsorted lists must first be sorted using a sorting algorithm before using the binary search.

The algorithm is more complex than a linear search, but has an improved time complexity of O(logN). It works by starting at the mid point of the list and following the steps below:

1) The algorithm compares the target to the item in the middle of the list.
   a. If the target item is the midpoint, the item has been located and the algorithm terminates.
   b. If the target item is higher than the midpoint, the algorithm discards the first half of the list.
   c. If the target item is lower than the midpoint, the algorithm discards the second half of the list.
2) The algorithm repeats the steps above, splitting the list in half each time until the target is found or no items remain.

<u>Binary Tree Search</u>

A binary tree search follows the same principals as the binary search but uses a binary tree. Once the list has been put into a tree, the tree is split following the same steps as above until the target is found.

Sorting Algorithms

Sorting algorithms are used to put the elements of an array in a specific order. This not only makes the list easier to read and work with, but also certain functions such as binary sort can only be completed against an ordered list.

<u>Bubble Sort</u>

The bubble sort algorithm works by swapping the position of adjacent items to place them in order. It is very inefficient with a time complexity of O(n^2).

A basic algorithm works through the steps below to sort data:

1) Compare items 1 and 2.
    a. If the items are in the correct order no swaps are made.
    b. If the items are in the incorrect order they are swapped.
2) Compare items 2 and 3.
    a. If the items are in the correct order no swaps are made.
    b. If the items are in the incorrect order they are swapped.
3) The algorithm completes the steps above until the last item is reached. This is known as one pass having been made.
4) The algorithm repeats the steps above to complete a second pass of the list.
5) This process is repeated until a full pass has been completed without any swaps being made.

As we can see from the steps above, the bubble sort algorithm is quite inefficient. In a well coded algorithm there will be an additional step to improve efficiency. At the end of the first pass, the last item in the list is guaranteed to be in the correct position and so it is locked and not checked on subsequent passes. At the end of the second pass, the second from last item is guaranteed to be in the correct place so it too is locked and not checked on subsequent passes. This reduces the number of items which need to be checked by 1 each pass.

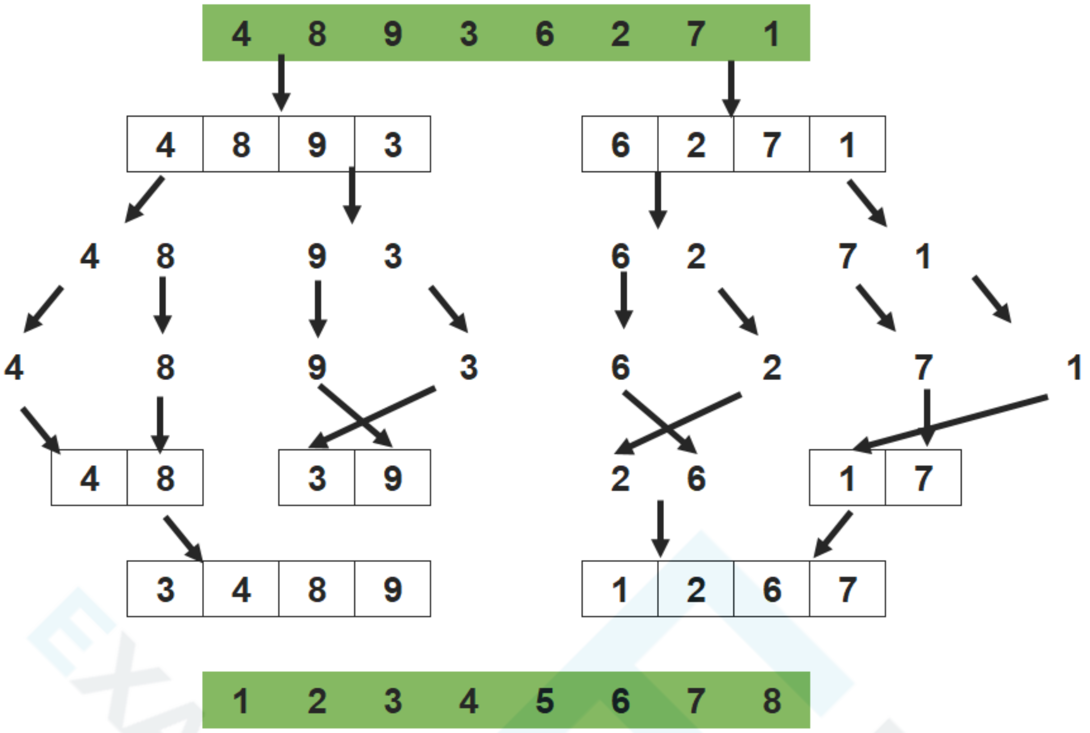
Merge Sort

A merge sort uses a 'divide and conquer' approach, splitting lists down into smaller lists to improve efficiency. It is therefore much quicker than the bubble sort with a time complexity if O(nlogn).

The algorithm follows the steps below:

1) The list is split into two halves as many times as is needed to result in each item being in its own list. For example, a list with 8 members would be split in 3 stages:
    a. First, split into 2 lists each with 4 members.
    b. Second, split into 4 lists each with 2 members.
    c. Lastly, split into 8 lists each with 1 member
2) The algorithm compares lists 1 and 2, joining them in to a new list in the correct order.
    a. This is repeated for lists 3 and 4, 5 and 6, and 7 and 8. At this point, there are now 4 ordered lists, each having two members.
3) The algorithm repeats the steps above, comparing the new lists until a single, ordered list is formed.

The example below shows how the merge sort algorithm might work.



Dijkstra's Algorithm

Dijkstra's Algorithm is an optimisation algorithm. Optimisation algorithms are designed to find the best possible solution to a problem posed.

Dijkstra's Algorithm is used to find the shortest path between two nodes in a graph. It is similar to the breadth first search algorithm, however, it records the nodes it has visited with a priority queue rather than a standard queue. It is heavily used in applications such as satellite navigation systems or network routing where there is a need to find the shortest route.

The cost of a path through the graph between two nodes is calculated by adding together the weights of all edges along that path. The shortest path is the path between two nodes with the minimum cost to travel between the start and end nodes.

The algorithm produces a list which holds the label of each node, the cost of the shortest path from each node to the start node and the label of the previous node in the path. This list allows you to backtrack to the start and determine the shortest path through the graph.

The algorithm uses the following steps to determine the shortest path once the start and end nodes are identified:

1) Record the distance of each node directly connected to the start node. Any nodes not connected to the start node are given a distance of infinity.
2) Mark the start note as fully explored.
3) Choose the note with the shortest distance to the start node and update the table with the distance from the start to each node. Mark this node as fully explored.
4) Repeat the step above for each node until all have been explored