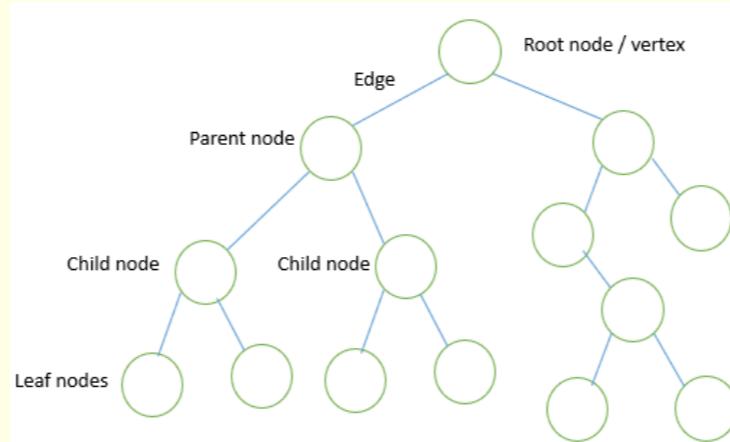


Non-Linear Data Structures

TREE *Hierarchical data structure*



Root Node stores the first data item
Node can have up to 2 descendants

Leaf Nodes have no descendants
Edge/ branches is the line

Type of Tree:

A common kind of tree is a Binary Tree
Every node can have at the most two child nodes

Depth-First Traversal

- Pre-Order
- In-Order
- Post-Order

Breadth-First

The Applications of Trees

- Manipulating hierarchical data, such as folder structures or moves in a game
 - making information easy to search
 - manipulating sorted lists of data

Tree

- A tree is a data structure used to represent **non-linear** data.
- A hierarchical data structure made up of nodes, with one root node and sub-nodes called children.
- A tree consists of **nodes** that are connected by **edges**.
- Each node contains some data, and each edge indicates a relationship between the nodes it connects.

Binary Tree

A binary tree is a rooted tree where every node has at most two child nodes. This means that a tree where every node has either two, one or none child nodes is a binary tree.

Binary Search Tree

A binary search tree (BST) is a **rooted tree** where the nodes of the tree are **ordered**.

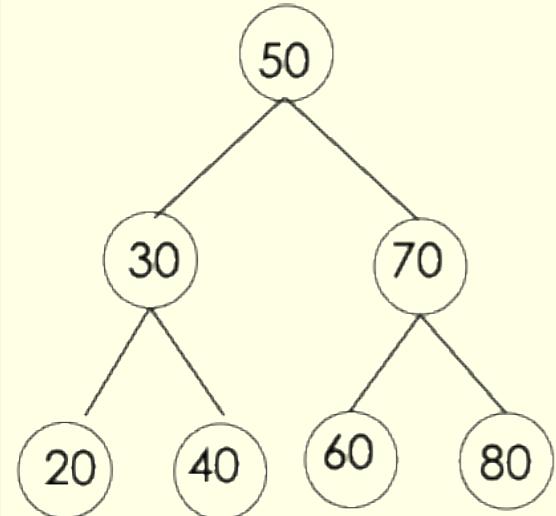
Binary Search Trees can have a **maximum of 2 children** for each parent node, but it is not a requirement that they have 2.

An advantage of storing data in a binary search tree instead of a 1-dimensional array: searching is faster ($O(\log n)$), inserting new tasks is faster and do not need to sort the structure (each time a new task is inserted)

If the order is ascending (low to high), the nodes of the left subtree have values that are lower than the root, and the nodes of the right subtree have values that are higher than the root.

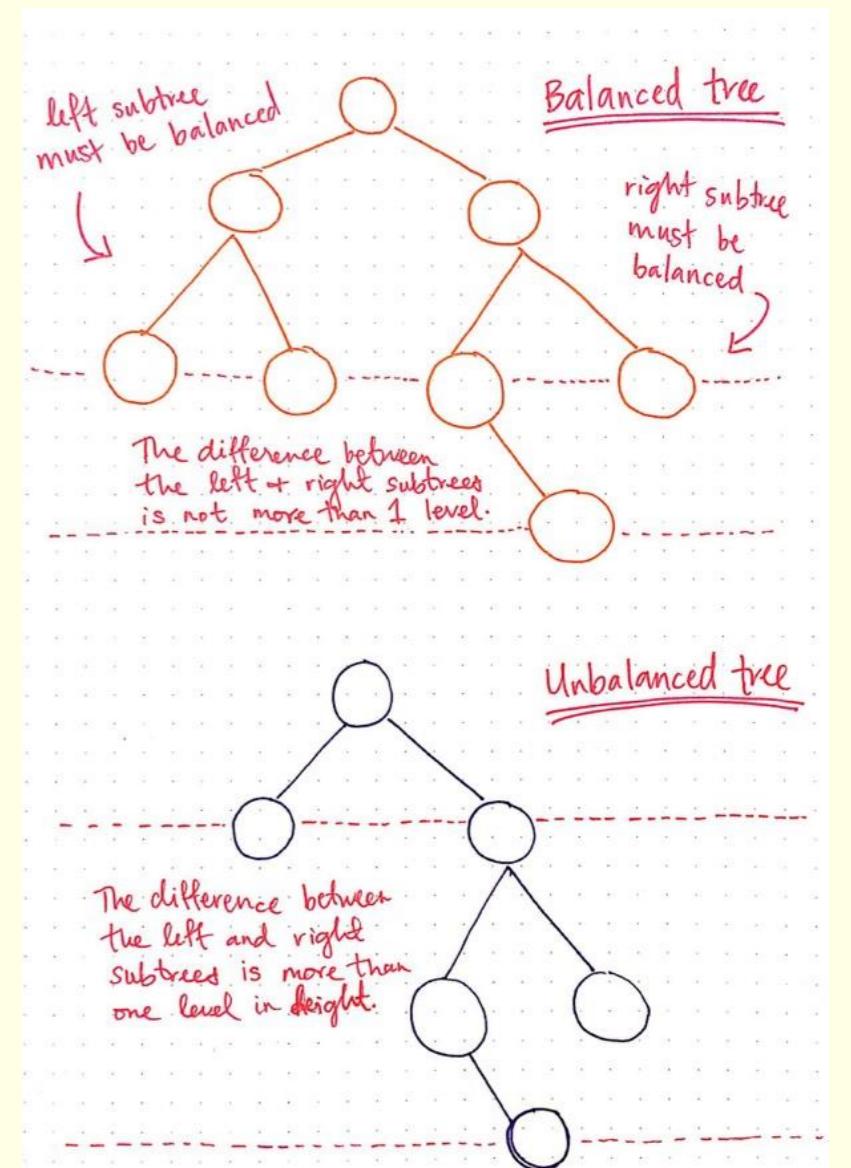
This property is true for any node of the tree; the nodes of its left subtree will have values that are lower, and the nodes of its right subtree will have values that are higher.

Imagine that you are searching for the number 40:



- Start at the root 50
- Compare 40 with 50 - 40 is lower, so check the left child
- Compare 40 with 30 - 40 is higher, so check the right child
- Compare 40 with 40 - success!

You have found the item with 3 comparisons.

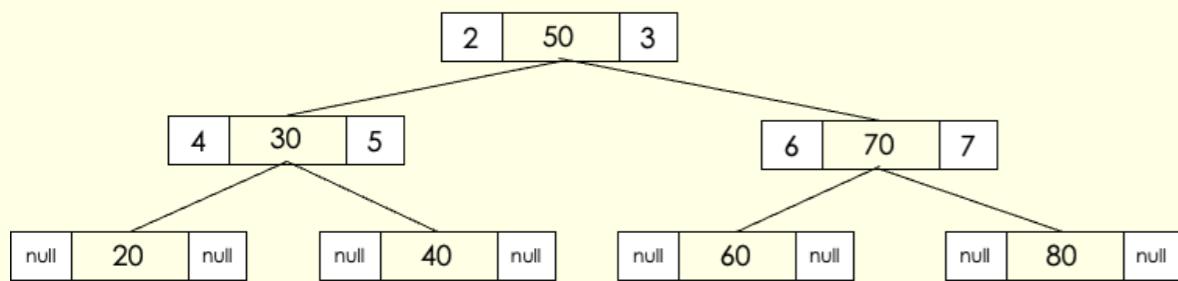


Drawback of an unbalanced binary search tree when searching for a value

- If the tree is unbalanced, nodes may form a long chain.
- **Time complexity** worsens from $O(\log n)$ (balanced) to $O(n)$ (unbalanced), meaning more comparisons are needed to find a value.
- Searching may require visiting almost every node in the tree.

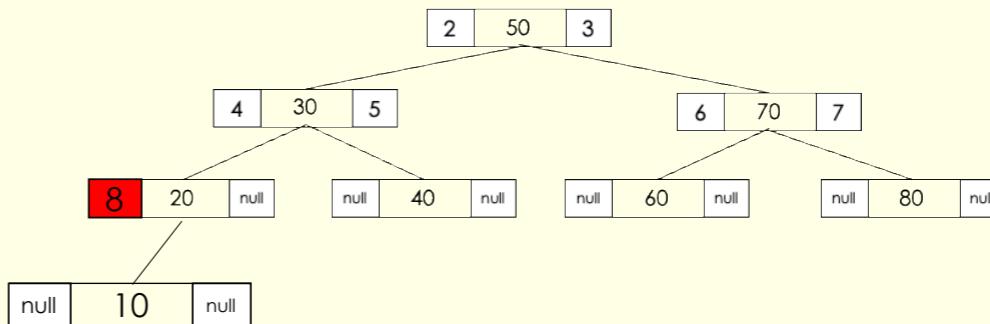
OCR Exam Abstract Representation of a Binary Tree using an array representation

Binary tree created using the data: 50, 30, 70, 20, 60, 80



Position	Left	Value	Right
1	2	50	3
2	4	30	5
3	6	70	7
4	null	20	null
5	null	40	null
6	null	60	null
7	null	80	null

After



Position	Left	Value	Right
1	2	50	3
2	4	30	5
3	6	70	7
4	8	20	null
5	null	40	null
6	null	60	null
7	null	80	null
8	null	10	null

Adding a node

- Create a new node
- Start at the root node
- Search the tree to find the location in of the new node
- Once a null pointer is found insert the new node at that position by updating the appropriate left or right of the parent node to point to the new node.

Remember that Binary Search Trees can have a maximum of 2 children for each parent node, but it is not a requirement that they have 2.

A binary tree can be stored as objects

Each node has a left pointer, a right pointer and the data being stored.
This is the node class and this is used to create a new_node object.

Node object

- **data** which in this case is a simple value attached to the node
- **right** which is a reference to the node to the right of the node
- **left** which is a reference to the node to the left of the node

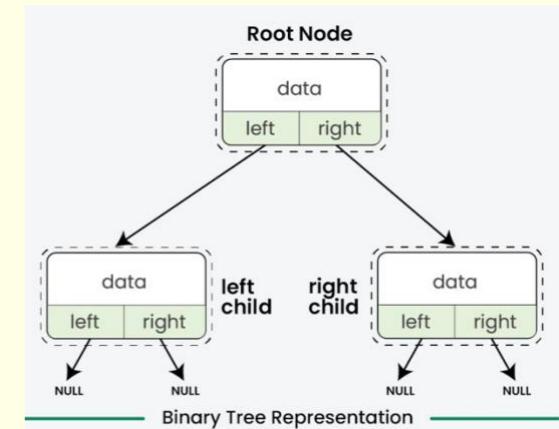
```
class: node
-----
attributes:
private data
private left: Integer
private right: Integer
-----
methods:
new (new_data)
getData()
setData()
getLeft ()
getRight ()
setLeft()
setRight ()
```

```
class Node
private data
private left
private right

public procedure new (new_data)
    data = new_data # instance variable to store the data
    left = null
    right = null
end procedure
end class
```

A binary tree has a root node

```
class BinaryTree
-----
private root
-----
public procedure new ()
    root = null
...
end class
```



This implementation is using **linked-node representation** of a binary tree instead of using an array representation.

Linked-node representation

Each node stores:

- Data value
- Pointer (reference) to left child
- Pointer (reference) to right child

The tree is connected via pointers rather than array indexes

Array representation

- The tree's nodes are stored in a sequential array.
- Each node's position is determined by its index in the array.

We can store each node's **data**, **left index**, and **right index** in a **2D array**

- Each **row** represents a node.
- **Column 0** = node data.
- **Column 1** = index of left child
- **Column 2** = index of right child

Inserting a node into binary tree

Step 1: Create a New Node

- Assign the new data to the node.
- Set the left and right of the node to null.

Step 2: Check if the Tree is Empty

- If the root is null:
 - Set the root to the new node.

Step 3: Traverse the Tree to find insertion point

- If the tree is not empty:
 - Start from the **root node**.
 - Repeat until you find **null**:
 - Compare the new data with the current node's data:

If new_data < current_node.data

- Follow the **left**

Else

- Follow the **right**

- Keep track of the **parent node** during traversal.

- Once a null pointer is found:

- Insert the new node at that position by updating the appropriate left or right of the parent node to point to the new node.

Procedure insert(new_value)

Step 1: Create a New Node

new_node = new Node(new_value)

Step 2: Check if the Tree is Empty

if root == null then

root = new_node

return

Step 3: Traverse the Tree to find insertion point

current = root

parent = null

while current != null:

parent = current

if new_value < current.getData()

current = current.getLeft()

else

current = current.getRight()

Step 4: Insert the Node

if new_value < parent.getData() then

parent.setLeft(new_node)

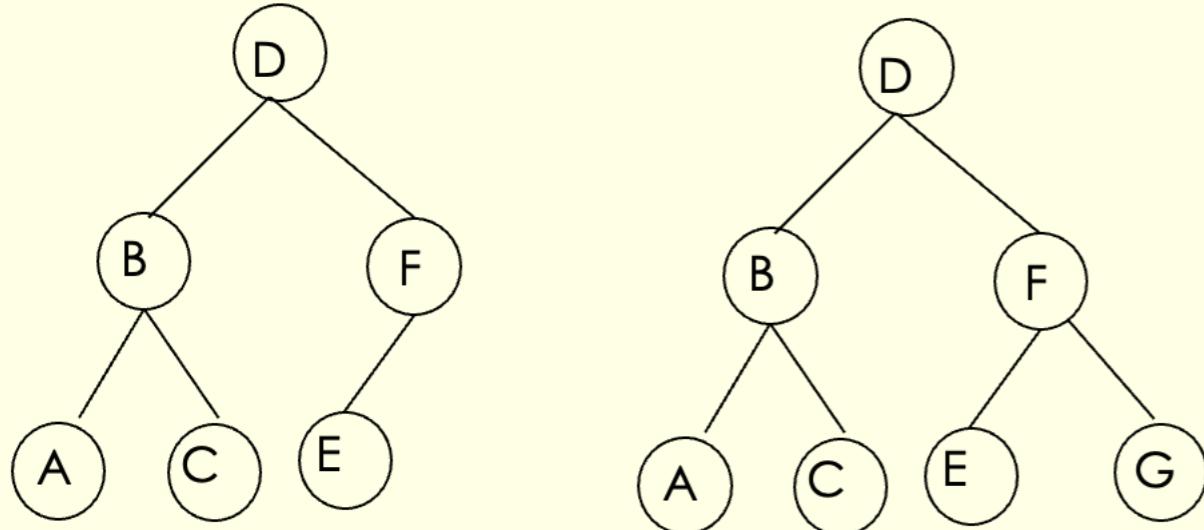
else

parent.setRight(new_node)

Deleting a Node

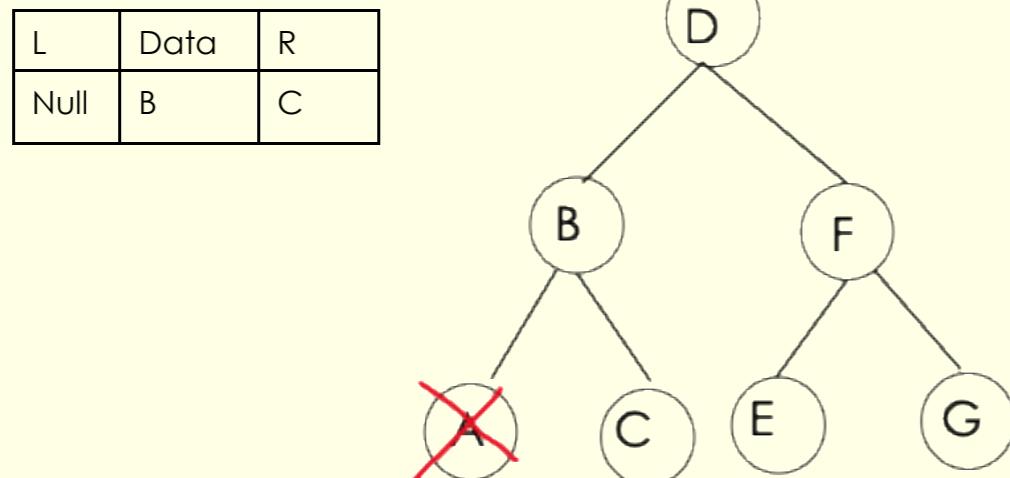
Example 1: Add node G

- Search the tree to find the location of node G
- Create a new node with value G
- Add a pointer from node F to the new node
- Make node G point to null/equivalent



Example 2 Delete a leaf node A

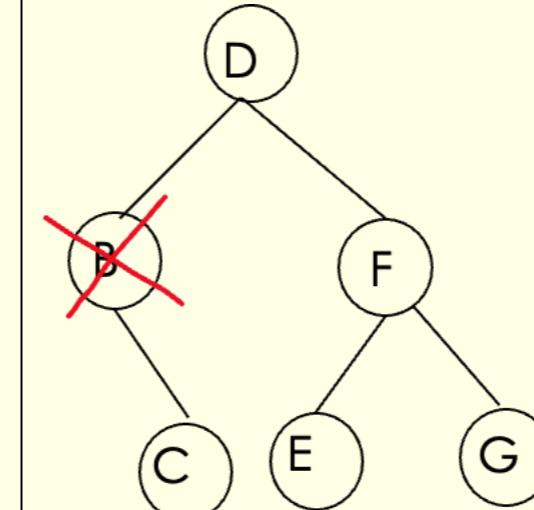
- Search/traverse tree until the required node is found
- Set the parent node pointer to the leaf node to null
- Add the deleted node to the free storage list (leave for garbage clear up)



Example 3 Delete a node with one child node B

- Search the tree to find the location of Node E
- Update the parent's pointer to bypass the node and point directly to the child.
- Add node B to the empty node list

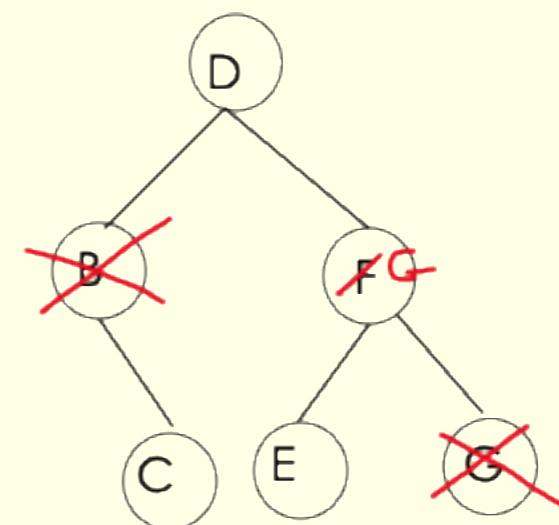
L	Data	R
C	D	F



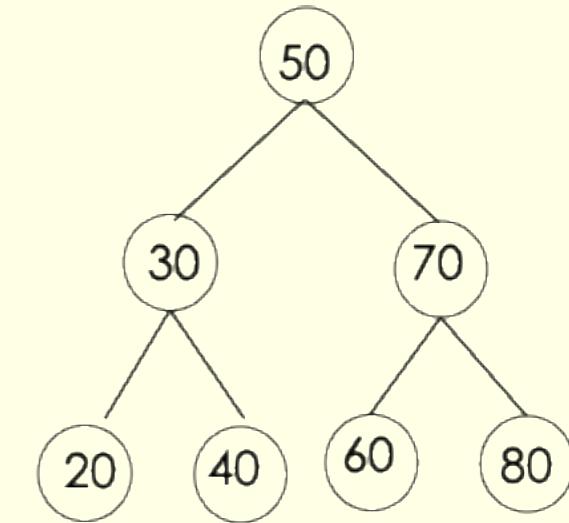
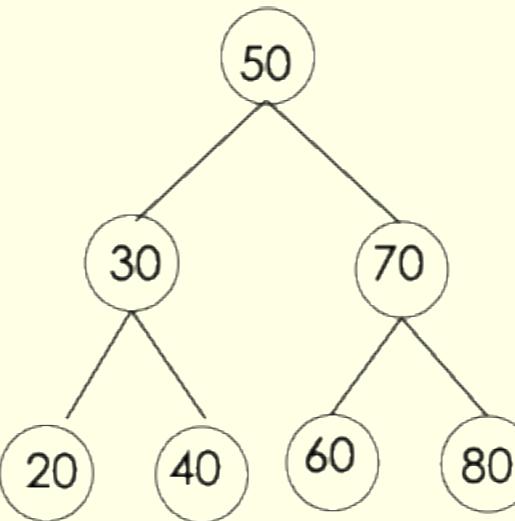
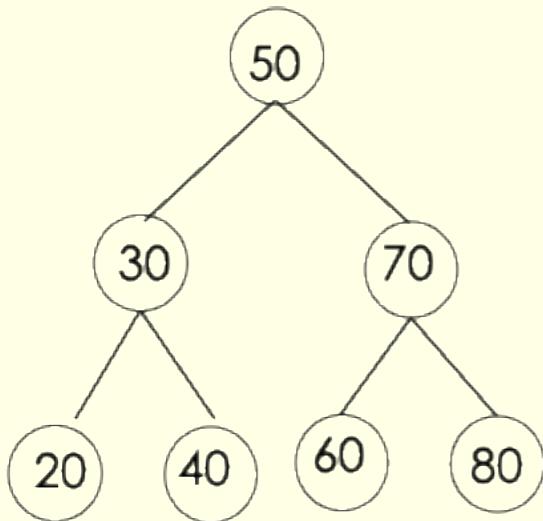
Delete a node with two child nodes

- Search the tree to find the node to delete (F).
- Find the in-order successor of F - the smallest value in the right subtree (G).
- Copy the value of G into F - F now contains G.
- Delete the original node G - G is a leaf, so it can be removed easily.
- Update pointers (Set F.right = NULL, F.left remains E)
- Add the deleted node to the free list

L	Data	R
E	F G	Null



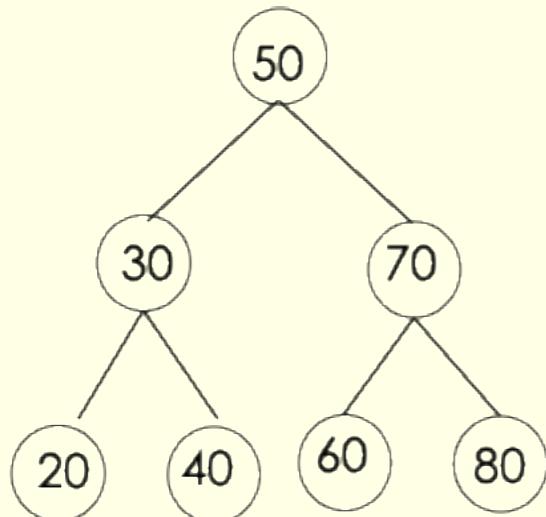
Binary Tree Traversals



Pre-order	In-order traversal	Post-order
Create a copy of a tree. 1. Visit the root. 2. Traverse the left subtree, 3. Traverse the right subtree	1. Traverse the left subtree, 2. Visit the root. 3. Traverse the right subtree	Delete a tree 1. Traverse the left subtree 2. Traverse the right subtree 3. Visit the root.
50, 30, 20, 40, 70, 60, 80	20, 30, 40, 50, 60, 70, 80	20, 40, 30, 60, 80, 70, 50
Algorithm Preorder <pre>procedure preorder(current_node) if current_node != None #Visit each node: NLR print(current_node.getData()) if current_node.getLeft() != None preorder(current_node.getLeft()) if current_node.getRight() != None preorder(current_node.getRight())</pre>	Algorithm Inorder <pre>procedure inorder(current_node) if current_node != None #Visit each node if current_node.getLeft() != None inorder(current_node.getLeft()) print(current_node.getData()) if current_node.getRight() != None inorder(current_node.getRight())</pre>	Algorithm Postorder <pre>procedure postorder(current_node) if current_node != None #Visit each node: LRN if current_node.getLeft() != None postorder(current_node.getLeft()) if current_node.getRight() != None postorder(current_node.getRight()) print(current_node.getData())</pre>
Useful for saving a tree.	Outputs values in sorted order for a BST	Useful for deleting nodes.

Breadth-first traversal

Level by level (top-down, left to right)



50 30 70 20 40 60 80

First add the root node into the **queue** while the queue is not empty.

Get the first node in the queue, and then print its value

Add both left and right children into the queue (if the current node has children)

We will print the value of each node, level by level.

```
procedure breadthFirst(currentnode)
```

```
    if (currentnode != None) {  
        q = new Queue()  
        q.enqueue(currentnode)  
        while NOT q.isEmpty()
```

Check root node is not null

```
            currentnode = q.dequeue()
```

Add root node to queue

```
            print (currentnode.getData())
```

While the queue is not empty

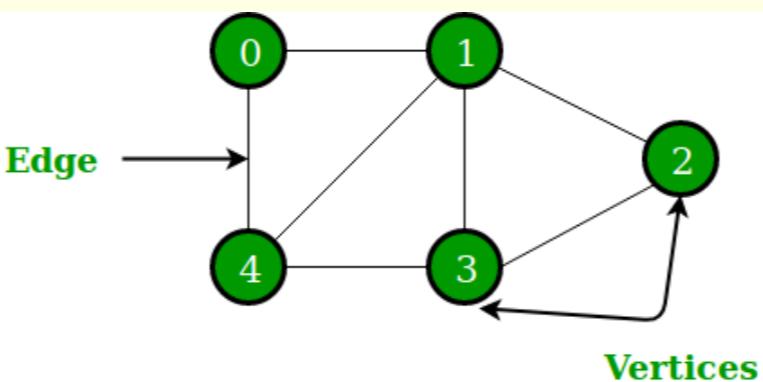
```
                if(currentnode.getLeft() !=null)  
                    q.enqueue(currentnode.getLeft())  
                if(currentnode.getRight() !=null)  
                    q.enqueue(currentnode.getRight())  
            endwhile  
    end procedure
```

print data

Add the left and right child nodes to the queue

GRAPH

Modeling connections or relationships between items



A graph is a set of vertices or nodes connected by edges or arcs.

Each vertex has a 'degree' – number of connections

No unique Root

Type of graphs:

Undirected graph

Directed graph

Weighted graph

Breadth-First and Depth-First Traversal

The Applications of Graphs

Graphs may be used to represent, for example:

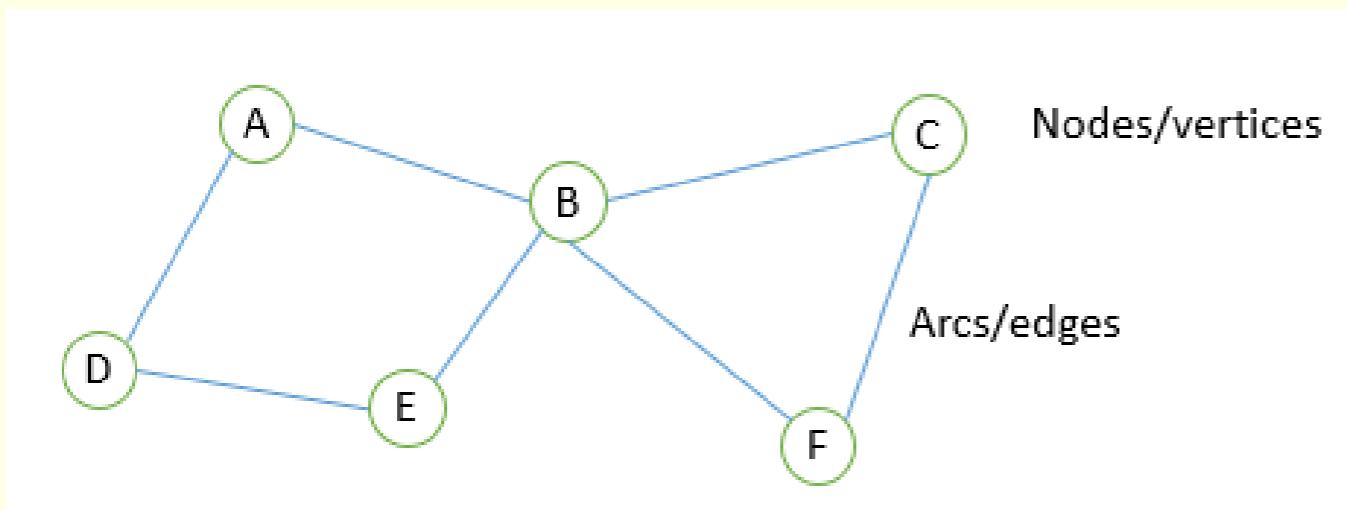
- Computer networks, with nodes representing computers and weighted edges representing the bandwidth between them
- Roads between towns, with edge weights representing distances, rail fares or journey times
 - Tasks in a project, some of which have to be completed before others

Adjacency Matrix

Implementing Graphs

Two possible implementations of a graph are the **adjacency matrix** and the **adjacency list**.

No Weighting, Undirected graph



Adjacency List

A list of all the nodes is created, and each node points to a list of all the adjacent nodes to which it is directly linked.

The adjacency list can be implemented using a dictionary, with the key being the node and the value, the edges.

A	[B, D]
B	[A, C, E, F]
C	[B, F]
D	[A, E]
E	[B, D]
F	[B, C]

The next nodes
should be in
alphabetical
order

Weighted Graph

A two-dimensional array can be used to store information about a directed or undirected graph.

Each of the rows and columns represents a node, and a value stored in the cell at the intersection of row i, column j indicates that there is an edge connecting node i and node j.

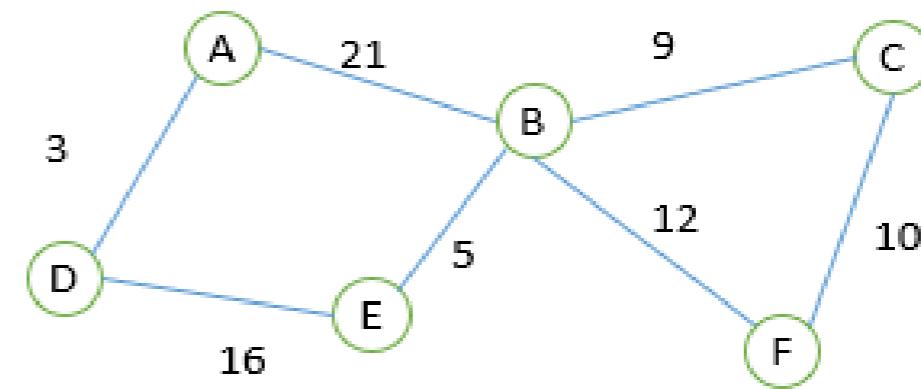
Graphs with no weights are given a value of 1 for connected nodes

In the case of an undirected graph, the adjacency matrix will be symmetric, with the same entry in (0,1) as in (1,0), for example.

	A	B	C	D	E	F
A	-	1	-	1	-	-
B	1	-	1	-	1	1
C	-	1	-	-	-	1
D	1	-	-	-	1	-
E	-	1	-	1	-	-
F	-	1	1	-	-	-

Weighted graphs add a value to an arc.

This might represent the distance between places or the time taken between train stations.



Adjacency Matrix

Adjacency List

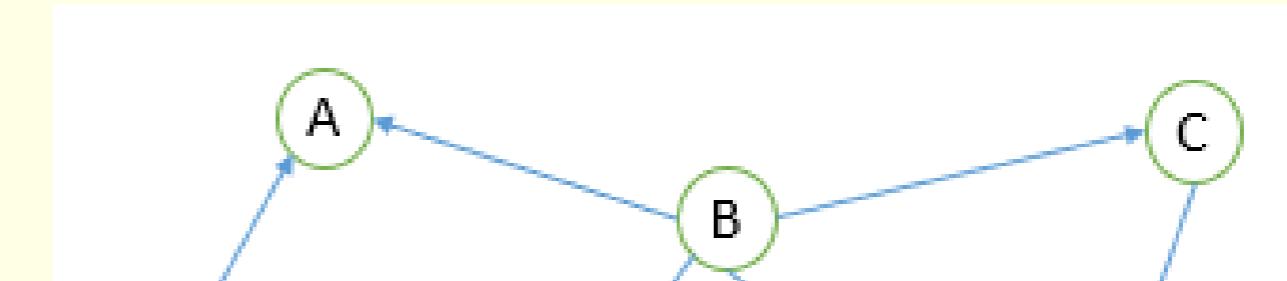
The adjacency list can be implemented as a list of dictionaries, with the key in each dictionary being the node and the value, the edge weight.

A	{B:21, D:3}
B	{A:21, C:9, E:5, F:12}
C	{B:9, F:10}
D	{A:3, E:16}
E	{B:5, D:16, }
F	{B:12, C:10}

	A	B	C	D	E	F
A	-	21	-	3	-	-
B	21	-	9	-	5	12
C	-	9	-	-	-	10
D	3	-	-	-	16	-
E	-	5	-	16	-	-
F	-	12	10	-	-	-

Directed Graph

Directed graphs only apply in one direction and are represented with edges with arrow heads on one end.



Adjacency List

A	
B	[A, C, E, F]
C	[F]
D	[A, E]
E	
F	

Adjacency Matrix

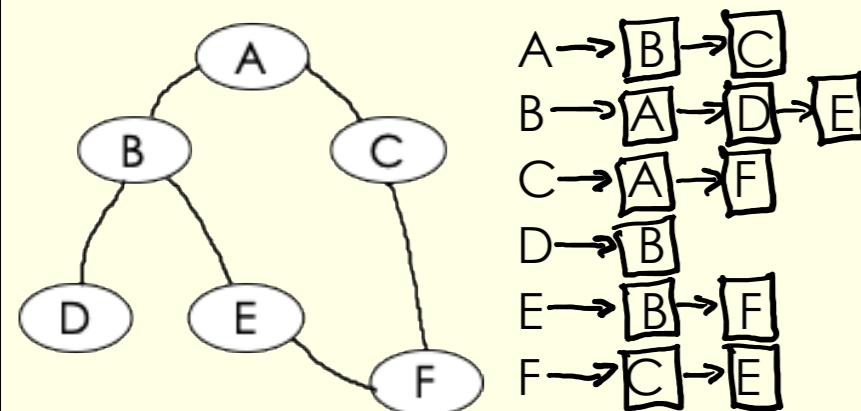
		To					
		A	B	C	D	E	F
From	A						
	B	1		1		1	1
	C						1
	D	1			1		
	E						
	F						

Depth-first search

Uses a stack to manage traversal

Visits nodes by exploring as **far down one branch** as possible before **backtracking**

Typically explores neighbours in a depth-wise manner



```
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
```

Current node	Stack	Visited
A	C B	A
B	C E D	A B
D	C E	A B D
E	C F	A B D E
F	C	A B D E F
C		A B D E F C

This method makes use of a stack data structure to push each visited node onto the stack and then pop them from the stack when there are no nodes left to visit:

1. Set the **current node** to the **start node**.
2. If the **current node** has **unvisited neighbours**, **push** them to the stack **in reverse order**.
3. If the **current node** is **not in visited**, add it to visited.
4. **Pop** the next node from the stack and set it as the new **current node**.
5. Repeat until the **stack is empty**.

Algorithm

```
Procedure depth_first_search (graph, start_node)
```

```
    visited = []
```

```
    s = Stack()
```

```
    current_vertex = start_node
```

```
    while current_vertex != None #while there is a node in the stack
```

```
        for vertex in reversed(graph[current_vertex])
```

```
            if not vertex in visited then
```

```
                s.push(vertex) #adds the vertex to the stack if not already in visited
```

```
            end if
```

```
            next vertex
```

```
            if not current_vertex in visited then
```

```
                visited.append(current_vertex) # current_vertex is added to visited
```

```
            endif
```

```
            current_vertex = s.pop() #next current_vertex
```

```
        endwhile
```

```
        print(visited)
```

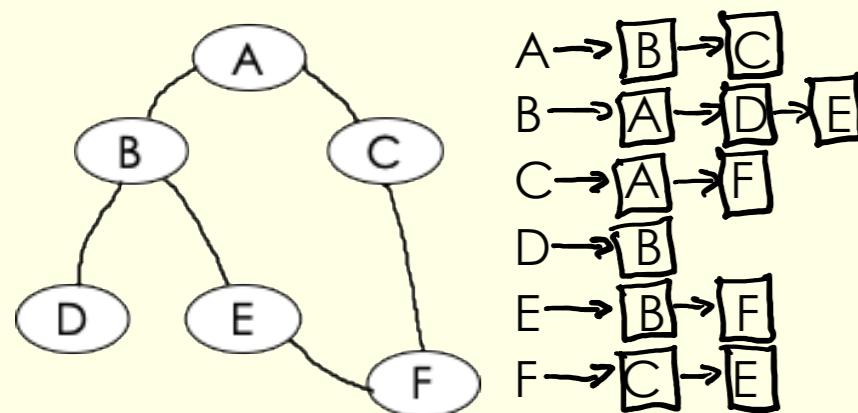
```
    end procedure
```

Breadth first Traversal

Uses a queue to manage traversal

Visits nodes **level-by-level**, exploring all neighbours before moving to the next level

Ideal for finding the shortest path in an unweighted graph.



Current node	Queue	Visited
	A	
A	BC	A
B	C D E	A B
C	D E F	A B C
D	E F	A B C D
E	F	A B C D E
F		A B C D E F

A B C D E F

1. Set **current node** to **start node**
2. If the **current node** has **unvisited neighbours**, **enqueue unvisited neighbour** nodes to the **queue**
3. If the **current node** is **not in visited**, add it to visited.
4. **Dequeue** node from queue and set to **current node**.
5. Repeat until the **queue is empty**

Algorithm

Procedure bfs(graph,start_node)

 visited = []

 q = queue() #create queue

 current_vertex = start_node# currentNode

 while current_vertex != None #while the queue is not empty

 for vertex in graph[current_vertex] #get all nodes that are adjacent nodes

 if not vertex in visited then # if vertex is in visited do not add to queue

 q.enqueue(vertex) #add vertex to queue

 next vertex

 if not current_vertex in visited then #if the current node has not been visited

 visited.append(current_vertex) #add node to visited

 current_vertex = q.dequeue() #next current_vetex

 print(visited)

End procedure

Depth-first search

Uses a stack to manage traversal

Visits nodes by exploring as far down one branch as possible before backtracking

Typically explores neighbours in a depth-wise manner

Breadth-First search

Uses a queue to manage traversal

Visits nodes level-by-level, exploring all neighbours before moving to the next level

Ideal for finding the shortest path in an unweighted graph