Non-Linear Data Structures

Binary Trees

Abstract representation of a binary tree

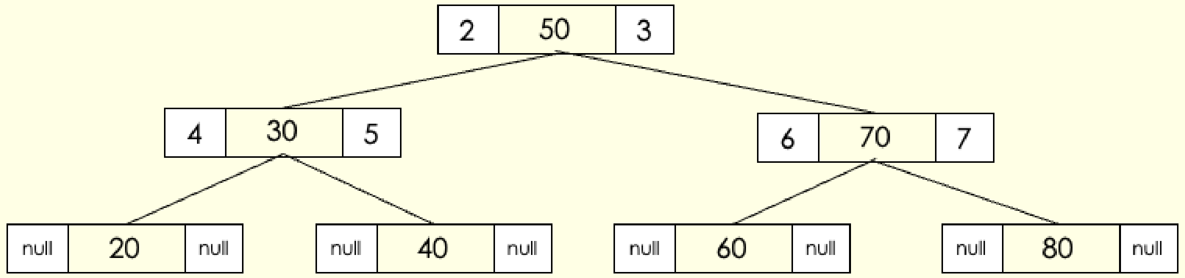
Binary trees can be stored in static array.

Each node contains:

- Left pointer
- Data
- Right pointer

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.

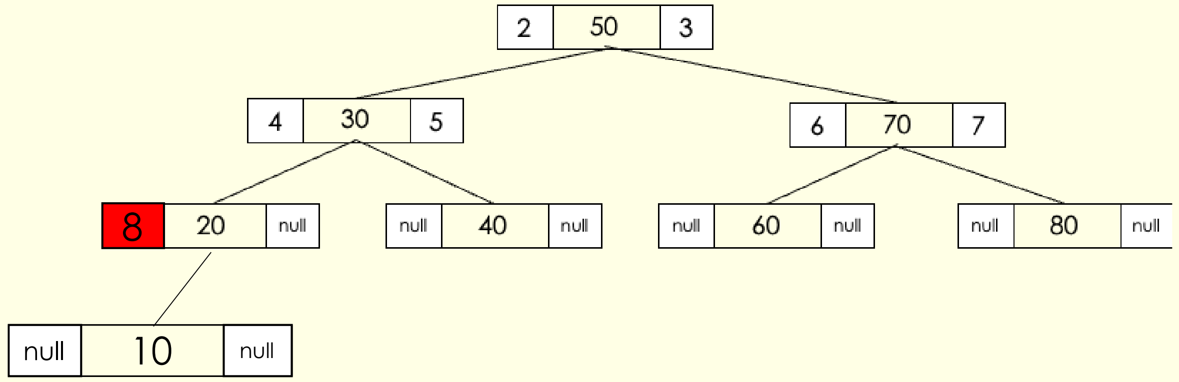
```
class node:
    data = None
    left_pointer = None
    right_pointer = None
```



Position	Left Pointer	Value	Right Pointer
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

Adding a node

- Create a new node
- Search the tree to find the location in of the new node
- Add a pointer from parent node to the new node
- Make new node point to null



Position	Left Pointer	Value	Right Pointer
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

Inserting a node to a binary tree

1. Check there is free memory for a new node – if no free memomirt output error
2. Create a new node and insert data into it

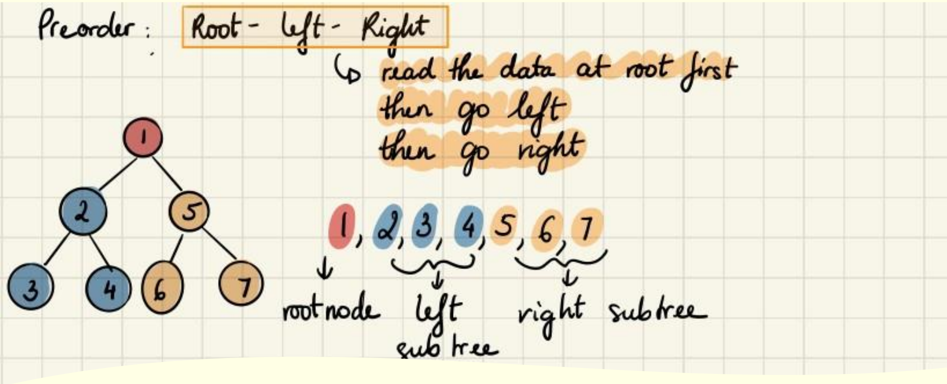
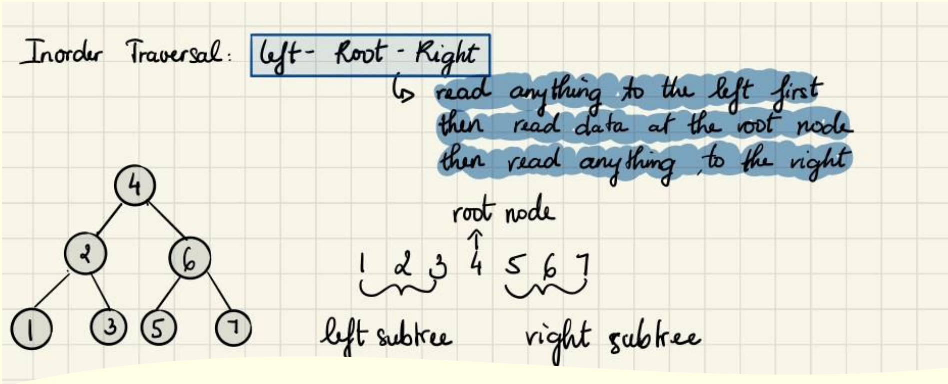
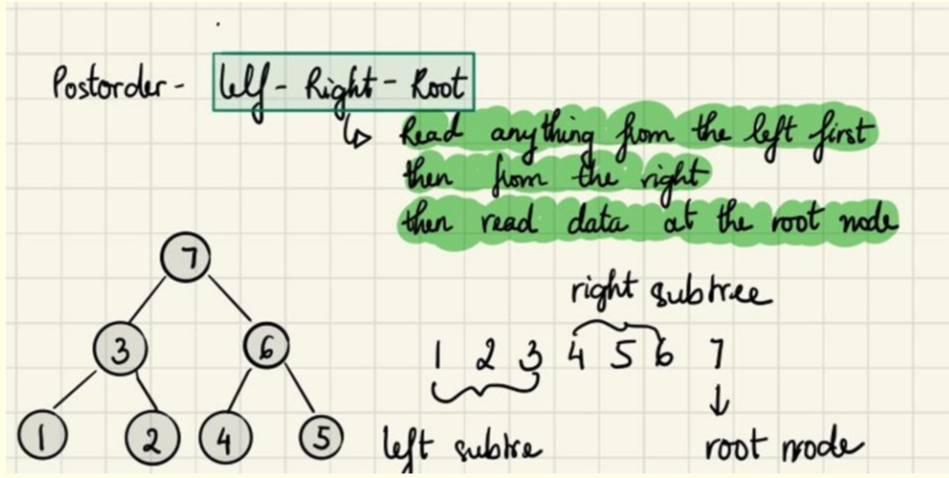
Removing an item from a binary tree

Find the node to be deleted

1. Start at the root node and set it as the current node.

3. If the binary tree is empty: • New node becomes first item • Set start pointer 4. If the binary tree is empty: • Start at the root node • If the new node should be placed before the current node, follow the left pointer • If the new node should be placed after the current node, follow the right position • Repeat until left node reached (pointer is NULL) • If the new node should be placed before the current node, set the left pointer to the be the new node • If the new node should be placed after the current node, set the right pointer to the be the new node	2. While the current node exists and it is not the one to be deleted: a. Set the previous node variable to be the same as the current node. b. If the item to be deleted is less than the current node, follow the left pointer and set the discovered node as the current node. c. If the item to be deleted is greater than the current node, follow the right pointer and set the discovered node as the current node.
	The node to be deleted has no children 3. If the previous node is greater than the current node, the previous node's left pointer is set to null. 4. If the previous node is less than the current node, the previous node's right pointer is set to null.
	The node to be deleted has one child 5. If the current node is less than the previous node: a. Set the previous node's left pointer to the current node's left child. 6. If the current node is greater than the previous node: a. Set the previous node's right pointer to the current node's right child.
	The node to be deleted has two children 7. If a right node exists and it has a left sub-tree, find the smallest leaf node in the right sub-tree: a. Change the current node to the smallest leaf node. b. Remove the smallest leaf node. 8. If a right node exists and it has no left sub-tree: a. Set the current node to be the current node's right pointer. b. Set the current node's right pointer to null.

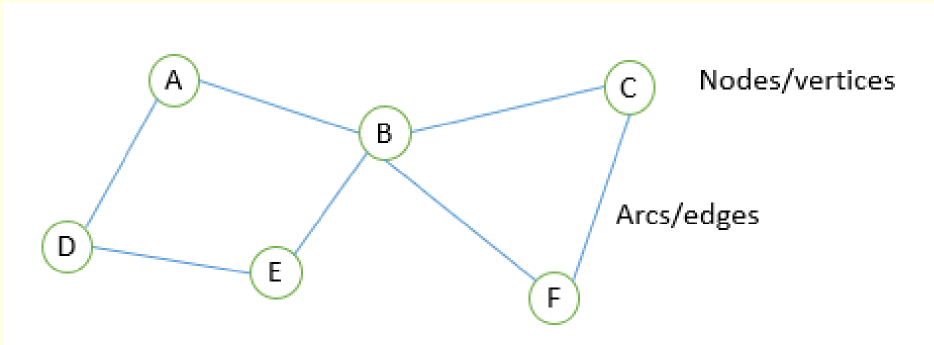
Binary Tree Traversals

<table>
<tr>
<td>Pre-order is used to create a copy of a tree.
1. Visit the root.
2. Traverse the left subtree,
3. Traverse the right subtree</td>
<td>In-order traversal gives nodes in non-decreasing order.
1. Traverse the left subtree,
2. Visit the root.
3. Traverse the right subtree</td>
<td>Post-order delete a tree
1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the root.</td>
</tr>
<tr>
<td>
</td>
<td>
</td>
<td>
</td>
</tr>
<tr>
<td>Algorithm Preorder
procedure preorder(current_node)
if current_node != None:
#Visit each node: NLR
print(current_node.data)
if current_node.left_pointer != None:
preorder(current_node.left_pointer)
if current_node.right_pointer != None:
preorder(current_node.right_pointer)</td>
<td>Algorithm Inorder
procedure inorder(current_node)
if current_node != None:
#Visit each node:
if current_node.left_pointer != None:
inorder(current_node.left_pointer)
print(current_node.data)
if current_node.right_pointer != None:
.norder(current_node.right_pointer)</td>
<td>Algorithm Postorder
procedure postorder(current_node):
if current_node != None:
#Visit each node: LRN
if current_node.left_pointer != None:
postorder(current_node.left_pointer)
if current_node.right_pointer != None:
postorder(current_node.right_pointer)
print(current_node.data)</td>
</tr>
</table>

Graphs

Adjacency List With No Weighting

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



First node is the parent/previous node

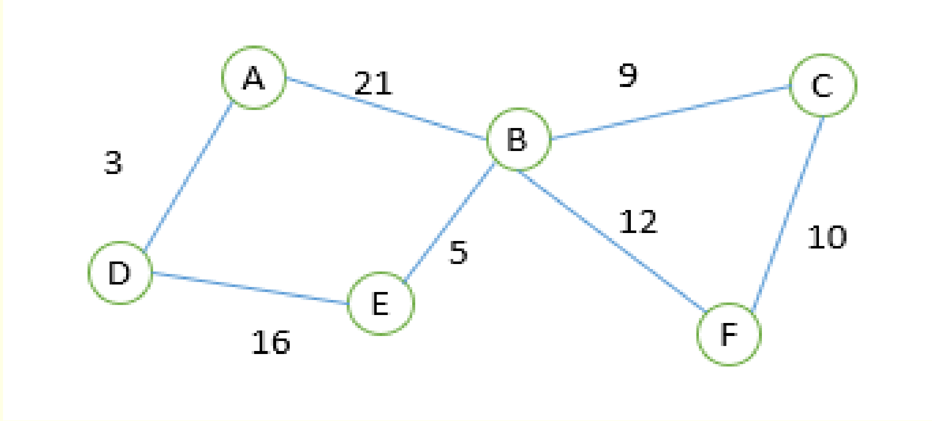
A	[D, B]
B	[A, E, C,F]
C	[B, F]
D	[A, E]
E	[D, B]
F	[B, C]

The next nodes should be in alphabetical order

	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 graph

Weighted graphs add a value to an arc.
This might represent the distance between places or the time taken between train stations



Adjacency List

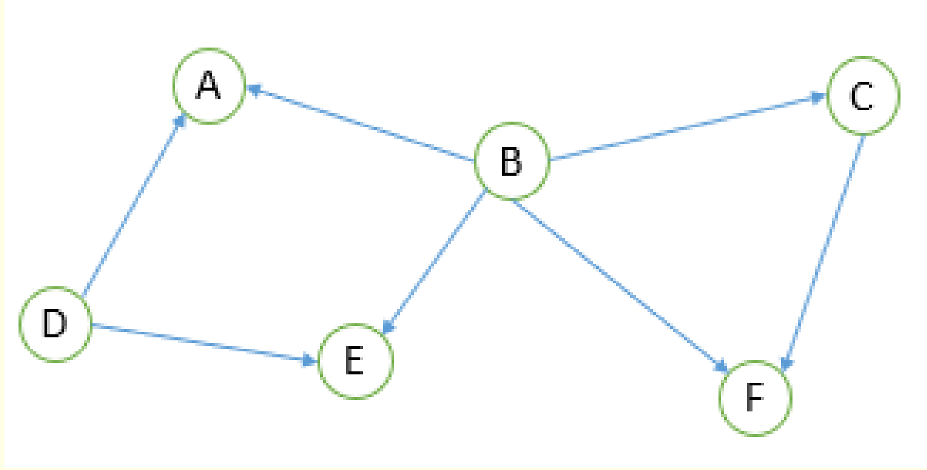
A	{D:3, B:21}
B	{A:21, E:5, C:9, F:12}
C	{B:9, F:10}
D	{A:3, E:16}
E	{D:16, B:5}
F	{B:12, C:10}

Adjacency Matrix

	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.



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

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

Depth-first search	Breadth-First search
Uses a stack to manage traversal Visits nodes by exploring as far down one branch as possible before backtracking	Uses a queue to manage traversal Visits nodes level-by-level, exploring all neighbours before moving to the next level