Trees

Class notes by Vibha Masti

Feedback/corrections: vibha@pesu.pes.edu

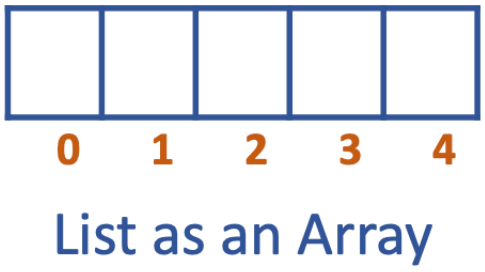
0.0 Table of Contents

1.0 Introduction

Problems with Linear Data Structures

List as an Array

- Fixed size
- Random insertion/deletion is time consuming



List as an Array

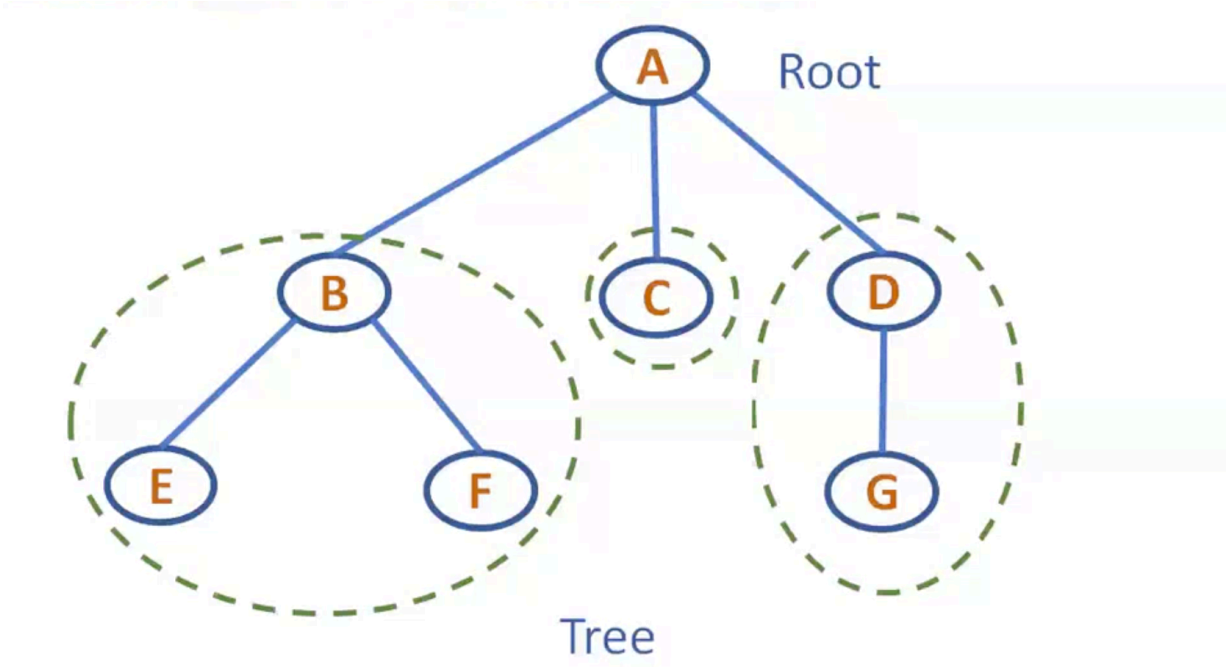
List as a Linked List

- Random access is time consuming

List as a Linked List

2.0 Trees

- Finite nonempty set of elements
- One element is the root
- Remaining elements are partitioned into disjoint subsets each of which is a tree

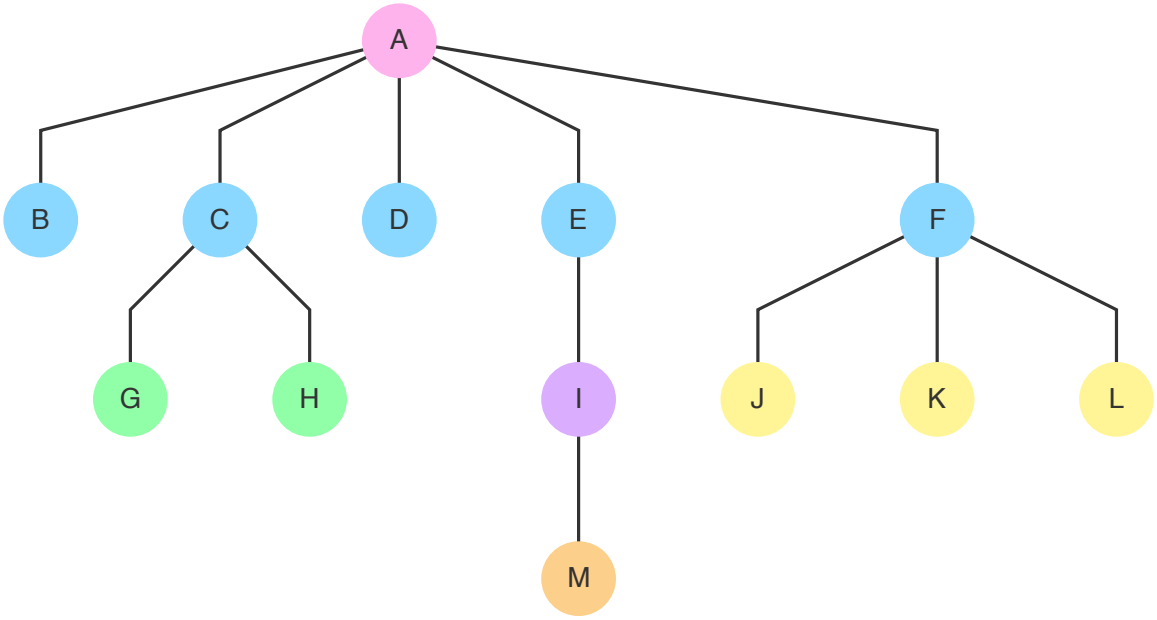


Ordered Tree

- A tree in which subtrees of each node form an ordered set
- We define the first, second, ..., last children of the nodes

n-ary Tree

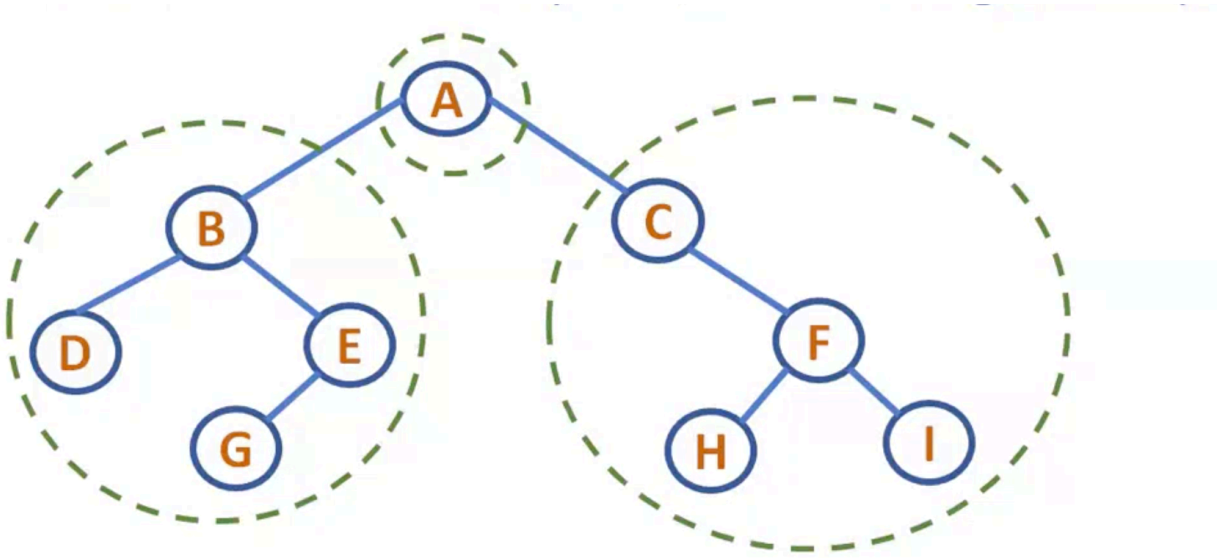
- An n-ary tree is a rooted tree in which each node has no more than n children
- Trees can be unary (n= 1), binary (n = 2), ternary (n = 3) and so forth
- Siblings are the children of the same parent (coloured the same as shown here)
- Root node is coloured pink, its children (all are siblings) are coloured blue
- This is an n-ary tree with n = 5 or pentanary tree



- Image representation: **quadtree** or **octatree** representation; very useful
- **Forest**: more than one tree

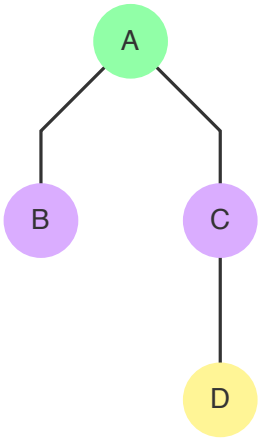
3.0 Binary Trees

- Maximum 2 child nodes in a binary tree (0, 1, 2)
- Partitioned into 3 subsets: root, left binary tree and right binary tree

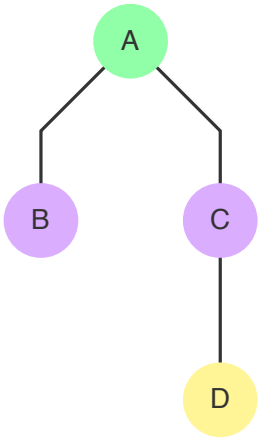


Terminology

- Each element is called a node
- Left node is called left child, right node is called right child
- **Parent** node is the one above
- A node without children is called a **leaf node** or **external node**
- All other nodes are called **non-leaf nodes** or **internal nodes**
- A node N1 is called the **ancestor** of a node N2 if
  - N1 is the parent of N2
  - N1 is the parent of some ancestor of N2
- **Level** of a node
  - Root: 0
  - Level of a child = 1 + level of its parent
- **Depth** of a tree: maximum level of any leaf node (path length from deepest leaf to root)
  - Depth of node A: 0
  - Depth of node B: 1
  - Depth of node C: 1
  - Depth of node D: 2
  - Depth of the tree: 2

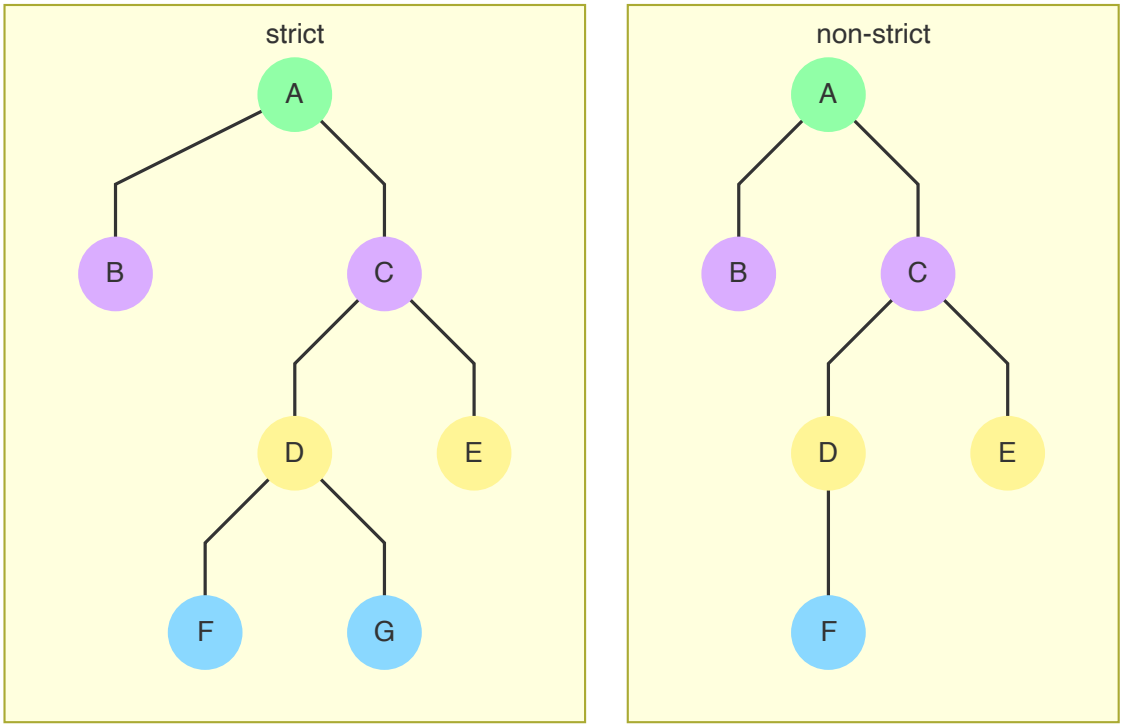


- Height of a tree: path length from deepest leaf to root
  - Height of node A: 2
  - Height of node B: 0
  - Height of node C: 1
  - Height of node D: 0
  - Height of the tree: 2



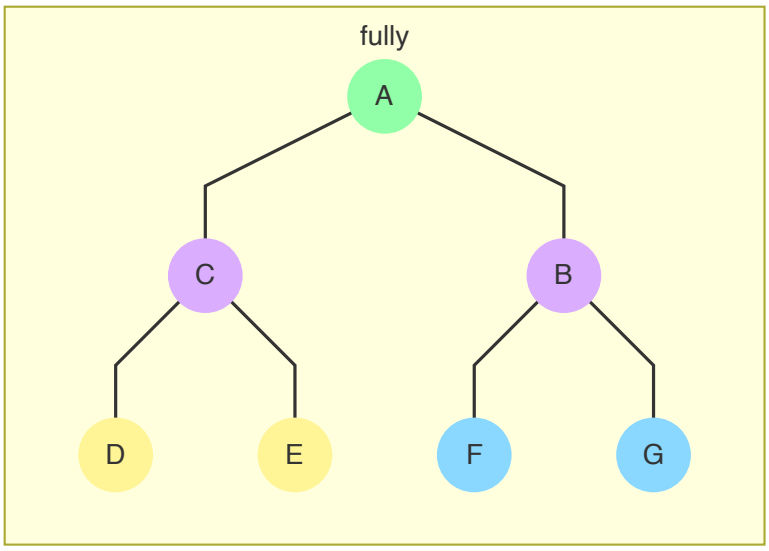
Strictly Binary Tree

- Tree where every node has either 0 or 2 children



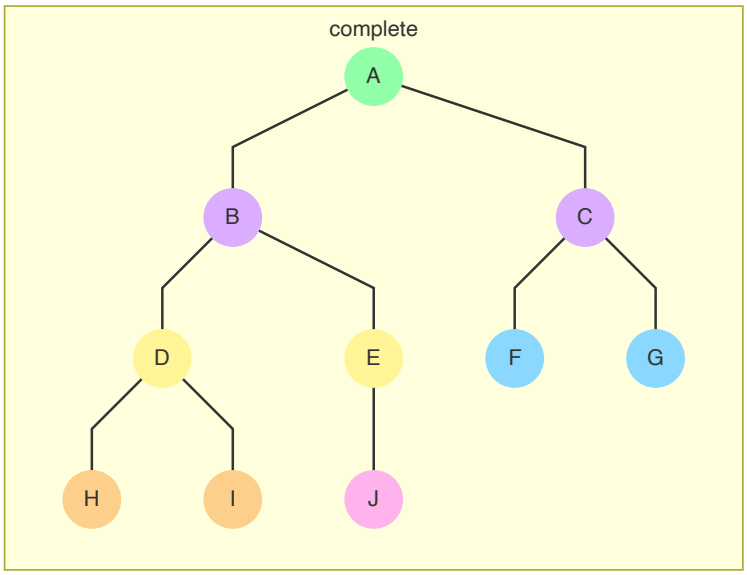
Fully Binary Tree

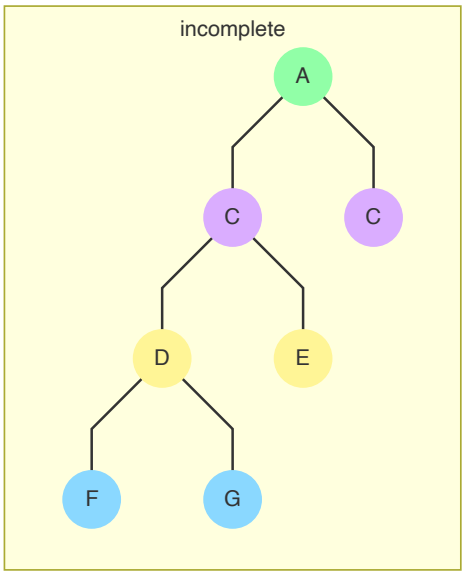
- All lead nodes at the same level
- Depth = d or 0 to d levels
- Total number of nodes = $2^0 + 2^1 + \ldots + 2^d = 2^{d+1} - 1$
- For a fully binary tree of depth 2, total number of nodes = $2^{2+1} - 1 = 2^3 - 1 = 7$

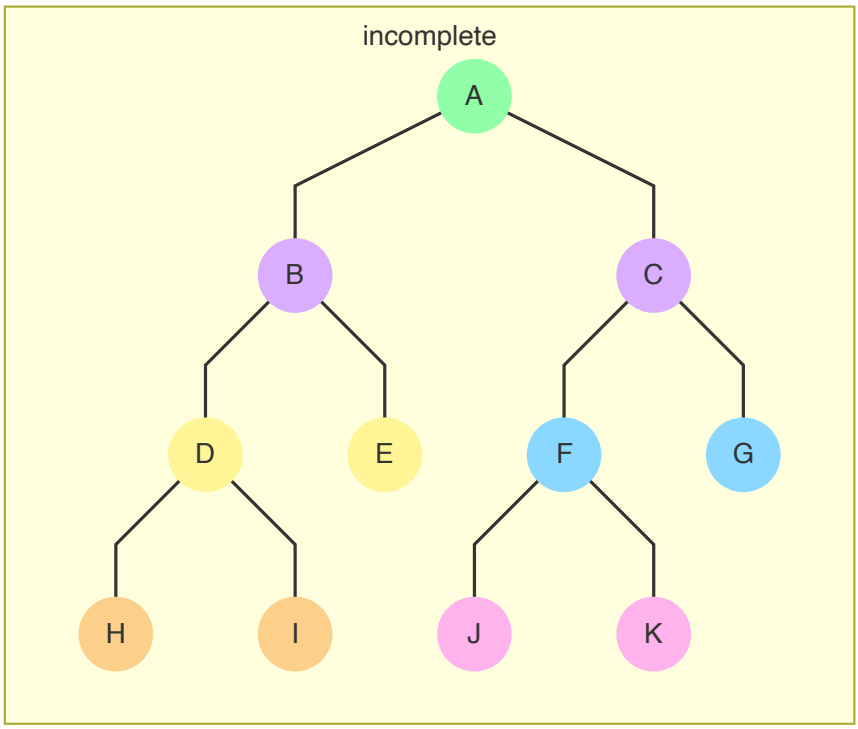


Complete Binary Tree

- All levels are completely filled except possibly the last level, and the last level has all keys as left as possible

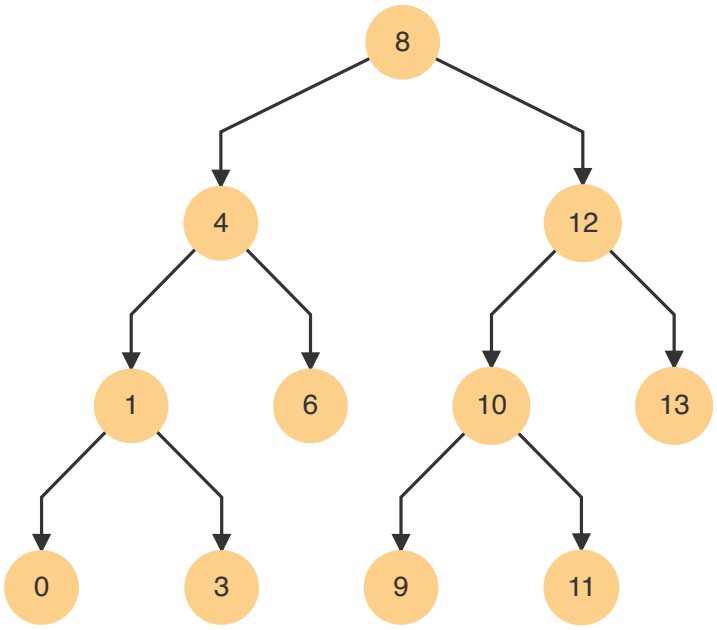






Binary Search Tree

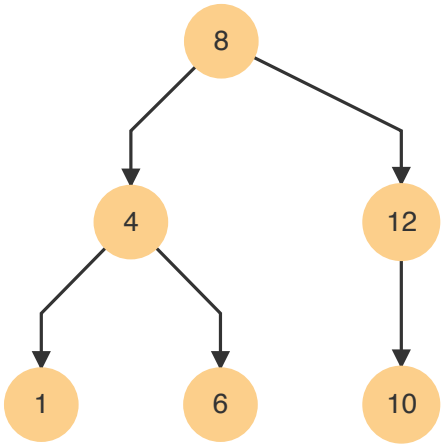
- Binary Search Tree is a node-based binary tree data structure which has the following properties
    - Left subtree of a node contains only nodes with keys lesser in value than the node's keys
    - Right subtree of a node contains only nodes with keys greater in value than the node's keys
    - Left and right subtrees are also binary search trees



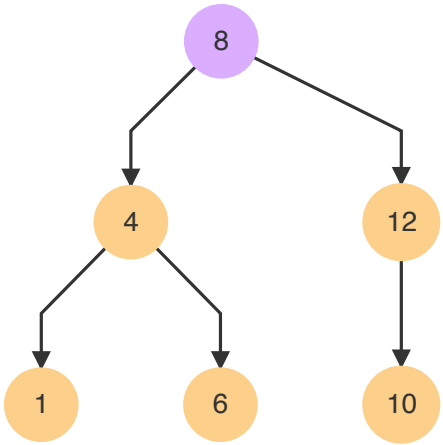
4.0 Tree Traversal

Preorder Tree Traversal

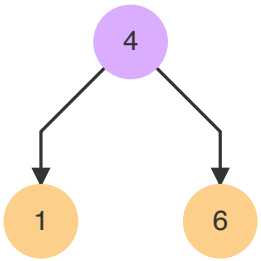
- Steps (recursive)
    1. Visit root node
    2. Traverse left subtree
    3. Traverse right subtree
- Consider the tree shown



1. Visit root node



2. Traverse left subtree
    1. Visit root node




```
2. Traverse left subtree
1. Visit root node
```


```
3. Traverse right subtree

    1. Visit root node
```

3. Traverse right subtree
    1. Visit root node

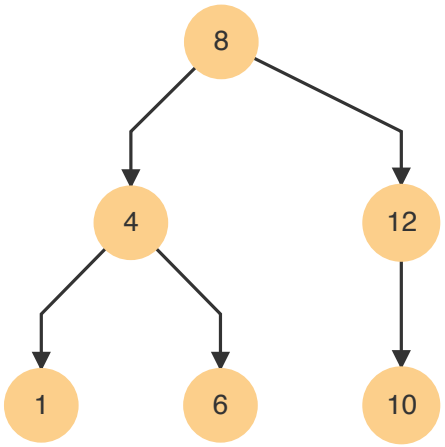
2. Traverse left subtree
    1. Visit root node

- Order of traversal: 8, 4, 1, 6, 12, 10

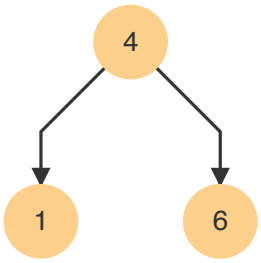
Inorder Tree Traversal

- Steps (recursive)
    1. Traverse left subtree
    2. Visit root node
    3. Traverse right subtree

- Consider the tree shown

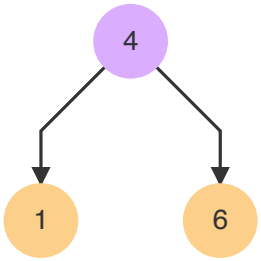


1. Traverse left subtree



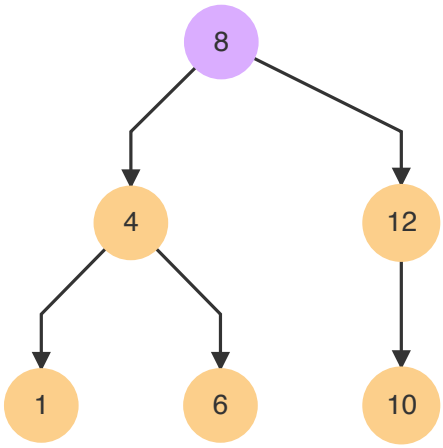
1. Traverse left subtree
    1. Traverse left subtree: `NULL`
    2. Visit root node
    3. Traverse right subtree: `NULL`

2. Visit root node



3. Traverse right subtree
    1. Traverse left subtree: `NULL`
    2. Visit root node
    3. Traverse right subtree: `NULL`

2. Visit root node

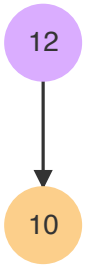


3. Traverse right subtree

1. Traverse left subtree
    1. Traverse left subtree: `NULL`
    2. Visit root node

3. Traverse right subtree: `NULL`

2. Visit root node



3. Traverse right subtree: `NULL`

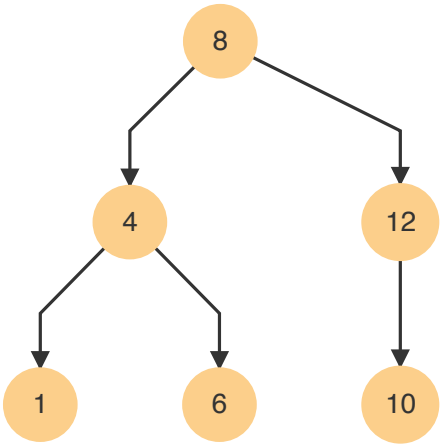
- Order of traversal: 1, 4, 6, 8, 10, 12
- Always in **ascending order**

Postorder Tree Traversal

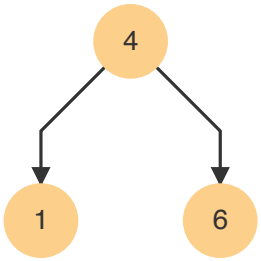
Steps (recursive)

1. Traverse left subtree
2. Traverse right subtree
3. Visit root node

- Consider the tree shown



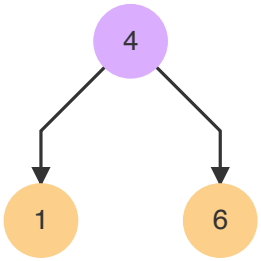
1. Traverse left subtree



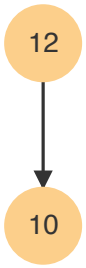
1. Traverse left subtree
    1. Traverse left subtree: `NULL`
    2. Traverse right subtree: `NULL`
    3. Visit root node

2. Traverse right subtree
    1. Traverse left subtree: `NULL`
    2. Traverse right subtree: `NULL`
    3. Visit root node

3. Visit root node

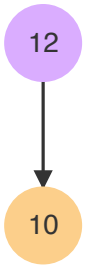


2. Traverse right subtree

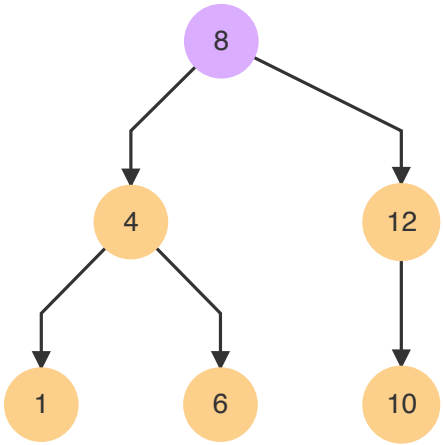


1. Traverse left subtree
    1. Traverse left subtree: `NULL`
    2. Traverse right subtree: `NULL`
    3. Visit root node

2. Traverse right subtree: `NULL`



3. Visit root node



Order of traversal: 1, 6, 4, 10, 12, 8

Code Implementation

- In C

Structures for `Tree` and `Node`

```c
// Node data structre
typedef struct node {
    int data;
    struct node *left, *right;
} Node;

// Tree data structure
typedef struct tree {
    Node *root;
} Tree;
```

Insertion (binary search tree) - recursive function

```c
// Function called once on Tree* datatype
void lex_insert(Tree *tree, int data) {

    // If no root node exists, create the first node
    if (tree->root == NULL) {
        Node *new_node = create_node(data);
        tree->root = new_node;
        return;
    }
```

```
10
11      // Otherwise, call the recursive function
12      lex_insert_leaf(tree->root, data);
13  }
14
15
16  // Recursive function
17  void lex_insert_leaf(Node *root, int data) {
18
19      // If the new data is less than the root node's data
20      if (data < root->data) {
21
22          // If the root node's left child is NULL, insert
23          if (root->left == NULL) {
24              Node *new_node = create_node(data);
25              root->left = new_node;
26          }
27
28          // Otherwise, call the recursive function on the left
29          // child as the new root
30          else {
31              lex_insert_leaf(root->left, data);
32          }
33      }
34
35      // If the new data is greater than or equal to the root node's data
36      else {
37
38          // If the root node's right child is NULL, insert
39          if (root->right == NULL) {
40              Node *new_node = create_node(data);
41              root->right = new_node;
42          }
43
44          // Otherwise, call the recursive function on the right
45          // child as the new root
46          else {
47              lex_insert_leaf(root->right, data);
48          }
49      }
50  }
```

Preorder Traversal

```c
// Function called once on Tree* datatype
void tree_preorder(Tree *tree) {
    preorder_leaves(tree->root);
    printf("\n");
}

// Recursive function
void preorder_leaves(Node *root) {
    if (root != NULL) {
        // Visit root node
        printf("%d ", root->data);

        // Traverse left subtree
        preorder_leaves(root->left);

        // Traverse right subtree
        preorder_leaves(root->right);
    }
}
```

Inorder Traversal

```c
// Function called once on Tree* datatype
void tree_inorder(Tree *tree) {
    inorder_leaves(tree->root);
    printf("\n");
}

// Recursive function
void inorder_leaves(Node *root) {
    if (root != NULL) {
        // Traverse left subtree
        inorder_leaves(root->left);

        // Visit root node
        printf("%d ", root->data);

        // Traverse right subtree
        inorder_leaves(root->right);
    }
}
```

Postorder Traversal

```c
// Function called once on Tree* datatype
void tree_postorder(Tree *tree) {
    postorder_leaves(tree->root);
    printf("\n");
}

// Recursive function
void postorder_leaves(Node *root) {
    if (root != NULL) {
        // Traverse left subtree
        postorder_leaves(root->left);

        // Traverse right subtree
        postorder_leaves(root->right);

        // Visit root node
        printf("%d ", root->data);
    }
}
```

Minimum Value

- Leftmost node (leaf)

```c
int min_val(Tree *tree) {
    if (tree->root == NULL) {
        return -1;
    }
    return min_val_leaves(tree->root);
}

int min_val_leaves(Node *root) {
    if (root->left == NULL) {
        return root->data;
    }
    return min_val_leaves(root->left);
}
```

Maximum Value

- Rightmost node (leaf)

```c
int max_val(Tree *tree) {
    if (tree->root == NULL) {
        return -1;
    }
    return max_val_leaves(tree->root);
}

int max_val_leaves(Node *root) {
    if (root->right == NULL) {
        return root->data;
    }
    return max_val_leaves(root->right);
}
```

Number of Nodes

- $1 + number\ of\ nodes\ in\ the\ left\ subtree + number\ of\ nodes\ in\ the\ right\ subtree$

```c
int no_of_nodes(Tree *tree) {
    return no_of_nodes_leaves(tree->root);
}

int no_of_nodes_leaves(Node *root) {
    if (root == NULL) {
        return 0;
    }
    return 1 + no_of_nodes_leaves(root->left) + no_of_nodes_leaves(root->right);
}
```

Number of Leaf Nodes

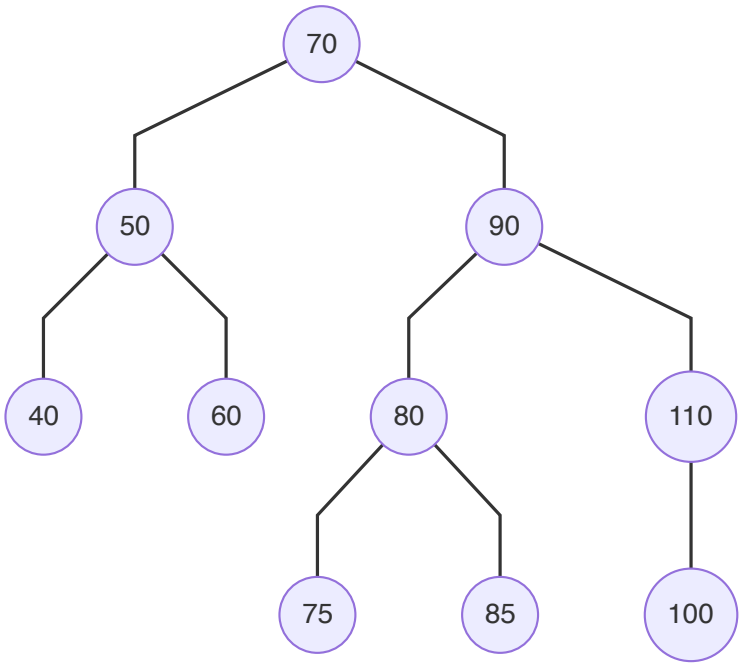
- $number\ of\ leaf\ nodes\ in\ the\ left\ subtree + number\ of\ leaf\ nodes\ in\ the\ right\ subtree$

```c
int no_of_leaf_nodes(Tree *tree) {
    return no_of_leaf_nodes_leaves(tree->root);
}

int no_of_leaf_nodes_leaves(Node *root) {
    if (root == NULL) {
        return 0;
    }
    if (root->left == NULL && root->right == NULL) {
        return 1;
    }
    return no_of_leaf_nodes_leaves(root->left) +
no_of_leaf_nodes_leaves(root->right);
}
```

5.0 Delete a Node

Algorithm

- Consider the tree shown



- To delete a node, first we must locate it in the tree
    - We declare a `temp` node that starts by pointing to the root node
    - We also declare a `parent` node that starts by pointing to `NULL`
    - The value that we are searching for in the tree is stored in `value`

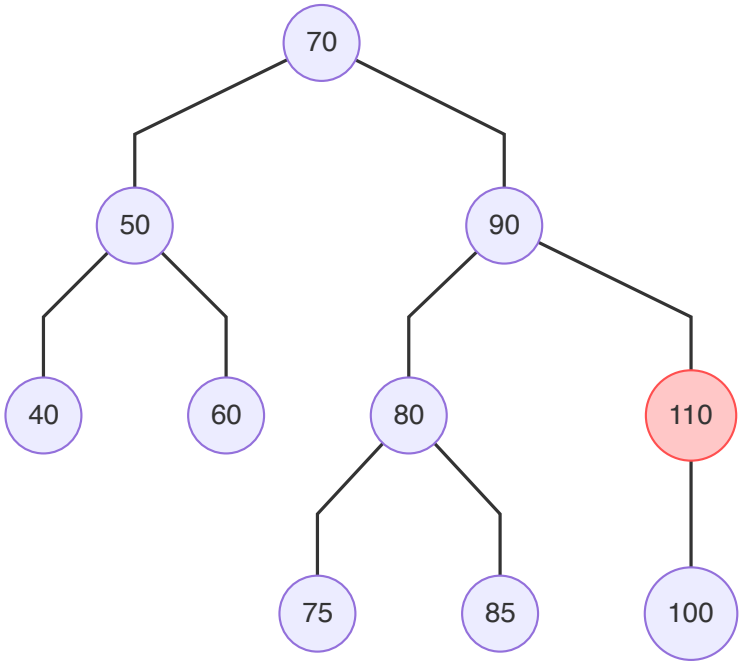
    - We declare two more node variables: `grandchild` and `inorder_successor`
- We traverse through the tree starting at the root node. If `value` is greater than the value stored in `temp`, we traverse to the right and if `value` is less than the value stored in `temp`, we traverse to the left (reassign `temp` to one of its children)
- We break out from the loop when either of the conditons have been met
    - the `temp` node points to `NULL`
    - the value of `temp` matches the `value` to be found
- After breaking out of the loop, there are four possibilities

Case 1: `temp` is pointing to `NULL`

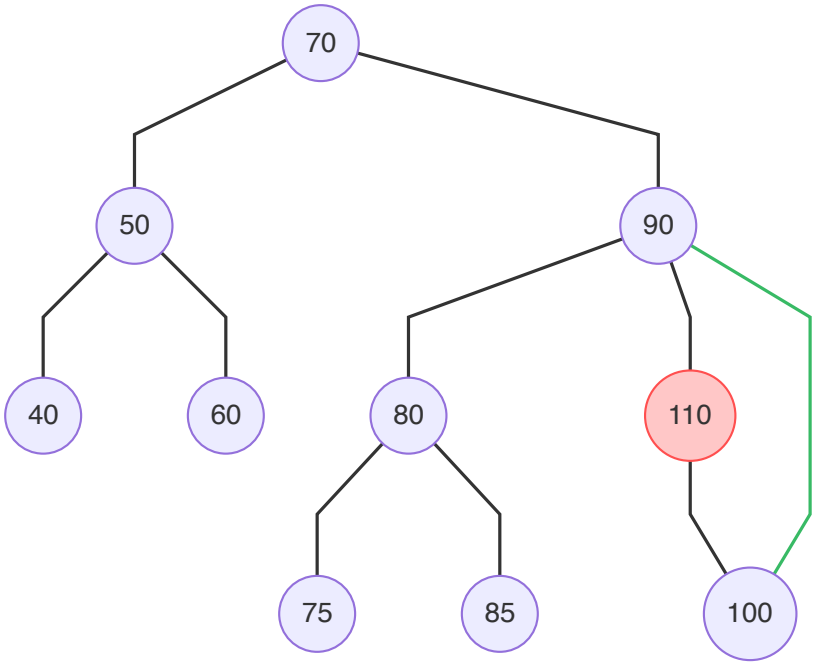
- The value was not found in the tree
- For example, if `value` = 20 in the tree shown above
- Do nothing

Case 2: `temp` has only one child

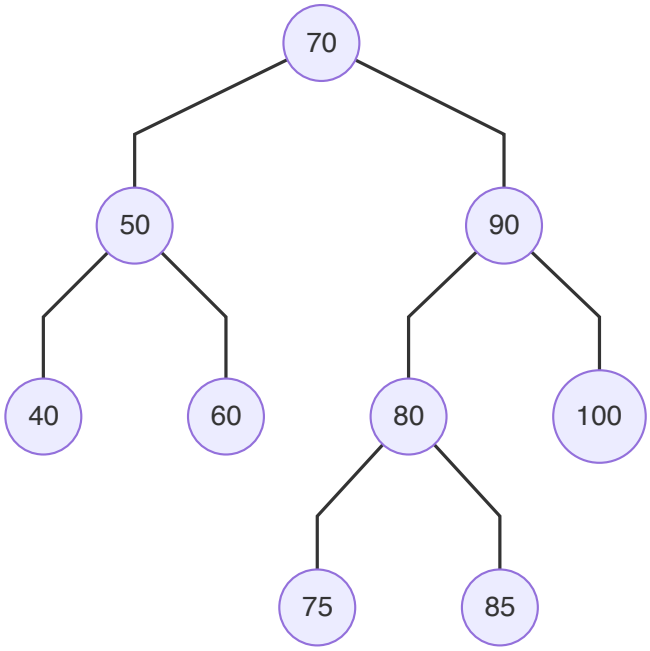
- In this case, we store the only child of `temp` in the node `grandchild`
- If `temp` was `parent`'s right child, we assign `grandchild` to be `parent`'s right child node
- If `temp` was `parent`'s left child, we assign `grandchild` to be `parent`'s left child node
- For example, if `value` = 110 in the tree shown



- We make `100` the right child of `90`



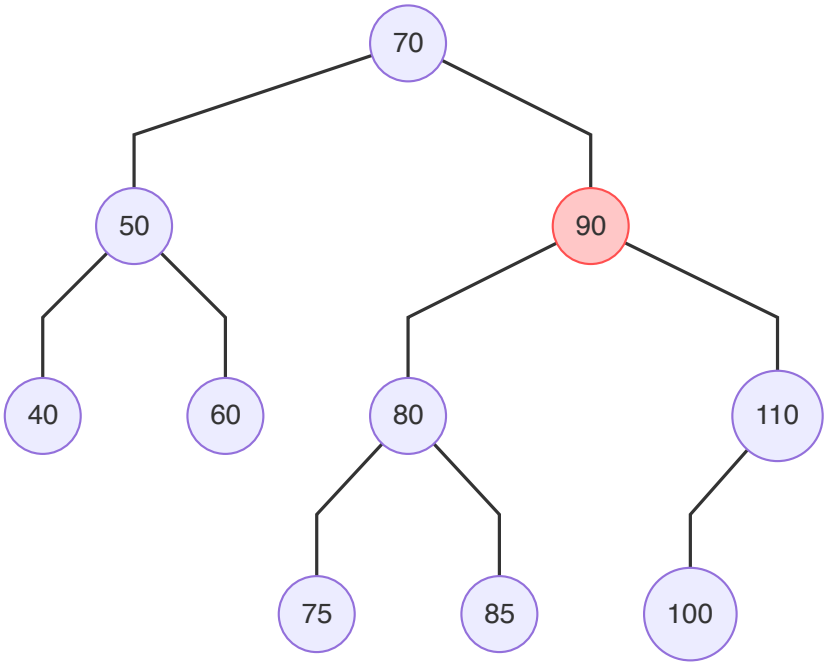
- We then delete `110`



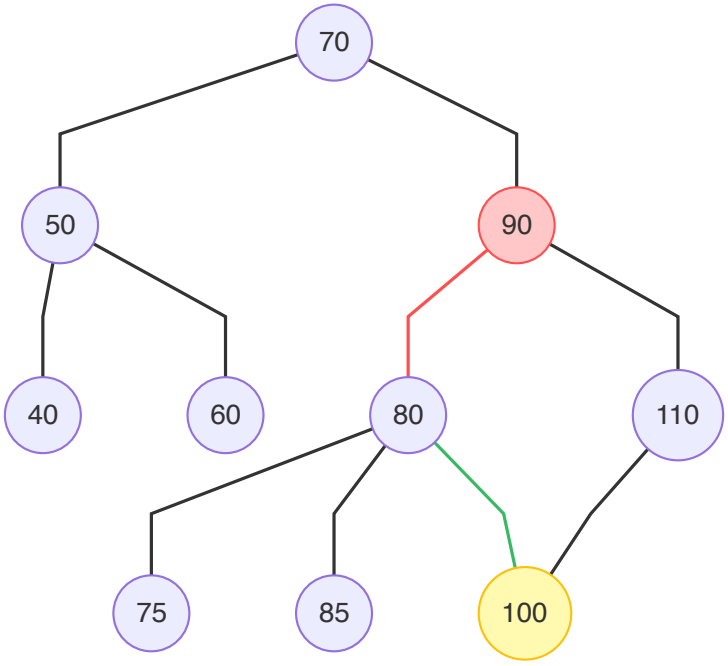
Case 3: `temp` has two children

- In this case, we find the inorder successor of `temp` (the next number in ascending order)
- The inorder successor of `temp` will be the leftmost leaf (minimum) of the right subtree
- We then assign the root node of the **left** subtree of `temp` to be the left child node of the inorder successor
- We store the root node of the **right** subtree of `temp` in `grandchild`
- We then treat it like a case 2

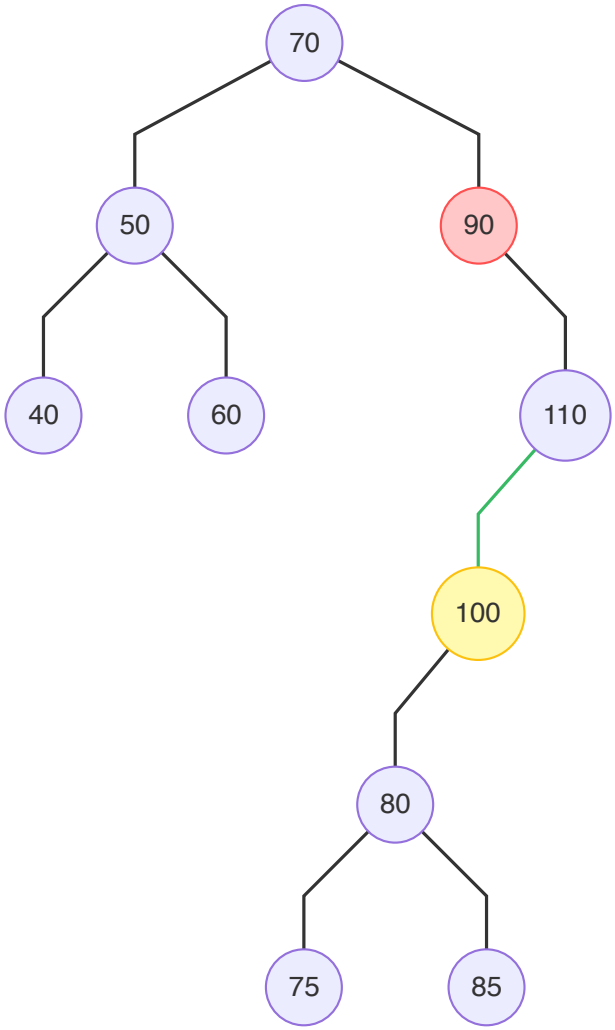
- For example, if we had to delete `90` in the tree shown below
- The parent node is `70`



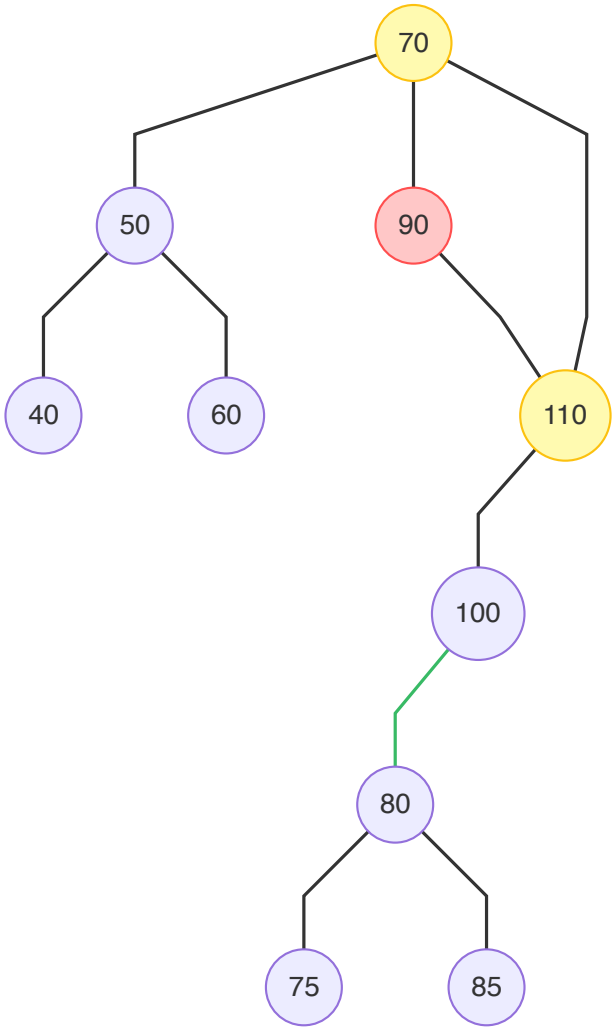
- The inorder successor would be `100`



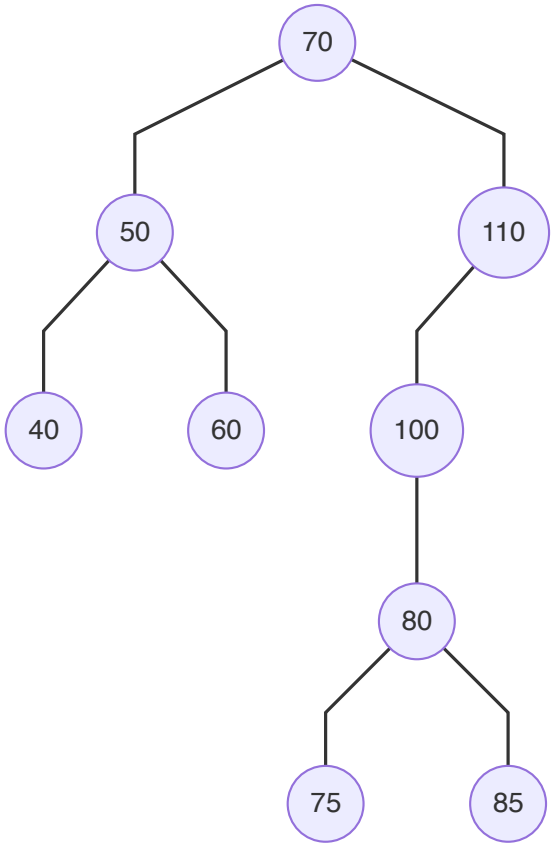
- We make `80` the left child of `100`



- The right child of `temp` is `110` and is stored in `grandchild`
- We make `110` the child of `70`



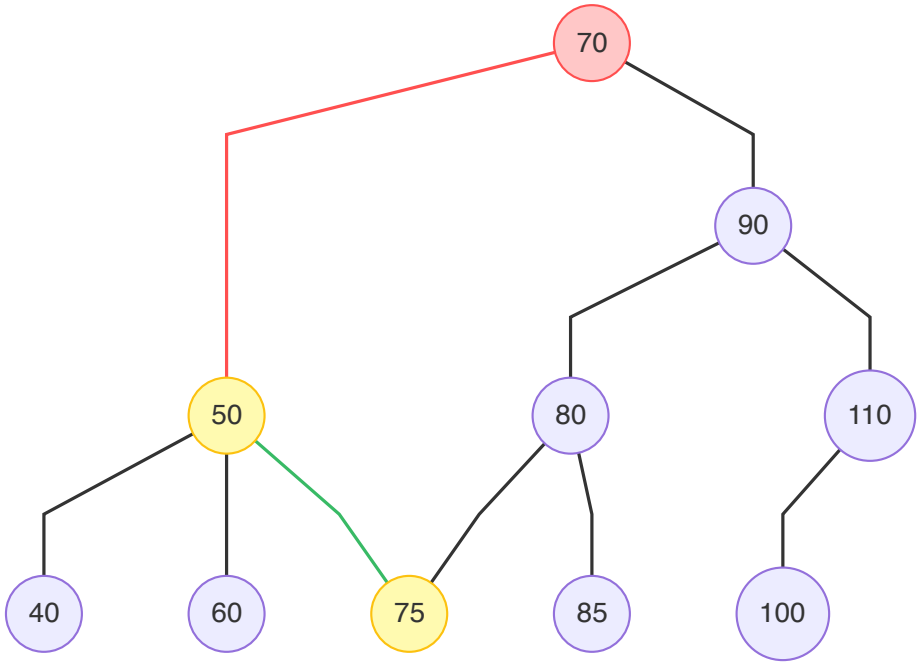
- We then delete `90`



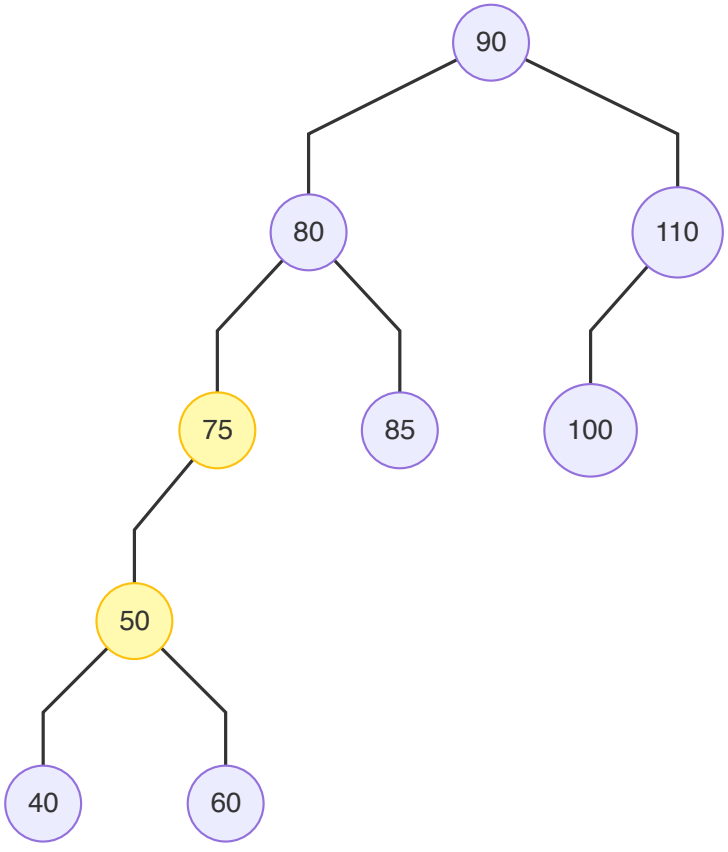
Case 4: `temp` is the tree's root node

- We treat it like a case 3
- We reassign the tree's root node to be `temp`'s right child node and we do not attach it to `temp`'s parent node (as it is `NULL`)

- For example, if we had to delete `70` in the tree shown below



- It would look like this



Code Implementation

- In C

```c
void delete_node(Tree *tree, int value) {
    Node *temp = tree->root, *parent = NULL, *grandchild = NULL,
*inorder_suc = NULL;

    // Search for a given node in a tree
    while (temp != NULL && temp->data != value) {

        // if value is greater than root node
        if (value > temp->data) {
            parent = temp;
            temp = temp->right;
        }

        // if value is less than root node
        else if (value < temp->data) {
            parent = temp;
            temp = temp->left;
        }
    }

    // value not found in tree
    if (temp == NULL) {
        printf("%d is not in the tree\n", value);
        return;
    }

    // temp has only one child (right)
    else if (temp->left == NULL) {
        grandchild = temp->right;
    }

    // temp has only one child (left)
    else if (temp->right == NULL) {
        grandchild = temp->left;
    }

    // temp has two children nodes
    else {
        // Find the inorder successor (next number in ascending
        // order). It will be the min of the right subtree
        inorder_suc = temp->right;

        while (inorder_suc->left != NULL) {
            inorder_suc = inorder_suc->left;
```

```
        }

        // Once found, the left subtree of temp gets attached
        // to the inorder successor of temp
        inorder_suc->left = temp->left;

        // grandchild becomes temp->right
        grandchild = temp->right;
    }

    // Attach grandchild

    // If root node is being deleted
    if (parent == NULL) {
        tree->root = grandchild;
    }
    else if (temp == parent->right) {
        parent->right = grandchild;
    }
    else if (temp == parent->left) {
        parent->left = grandchild;
    }

    // Delete temp
    free(temp);
}
```

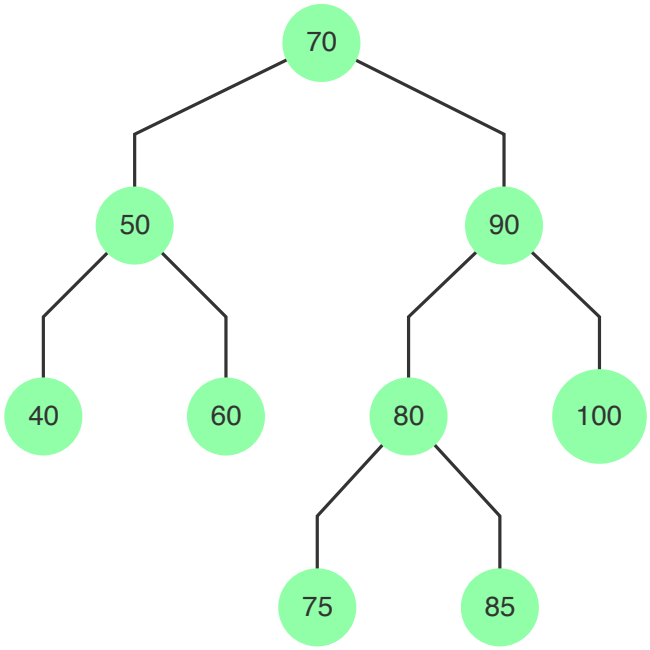
6.0 BST Using Arrays

- We store the tree in an array, not like a linked list

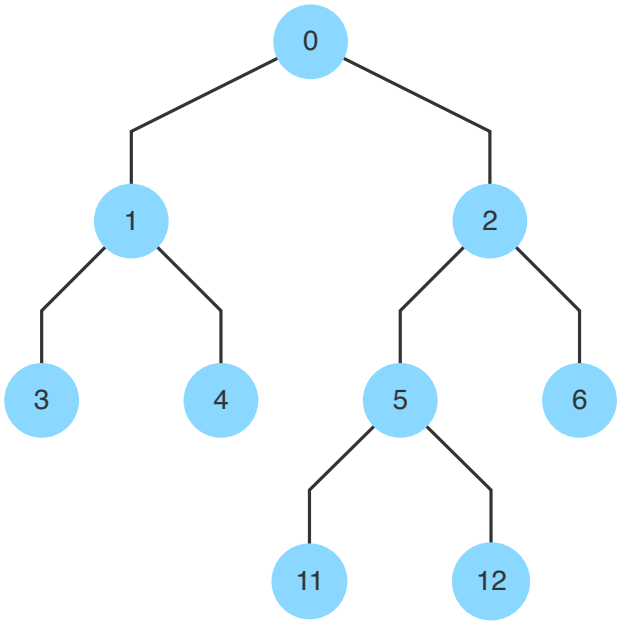
Insertion

- The index of a node is given as
  - $2 \times i + 1$, for a left child
  - $2 \times i + 2$, for a right child

  where i is the index of the root node
- The indices start at 0 with the tree's root node
- Stored in breadth first order



- The indices of the tree are as shown below



- Put into an array, it looks like (`0` is empty)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
70	50	90	40	60	80	100	0	0	0	0	75	85	0	0

Code Implementation

- The tree `bst` is an `int` array of size `TREE_SIZE`

```c
#define TREE_SIZE 40

int main() {
    int binary_tree[TREE_SIZE];
    bst_initialise(binary_tree);

    .
    .
    .
    return 0;
}


void bst_initialise(int *bst) {
    for (int i = 0; i < TREE_SIZE; ++i) {
        bst[i] = 0;
    }
}
```

Insertion

```c
void bst_insert(int *bst, int data) {
    int i = 0;

    // Empty tree
    if (bst[i] == 0) {
        bst[i] = data;
    }

    // Tree not empty
    else {
        while (i < TREE_SIZE && bst[i] != 0) {
            if (data < bst[i]) {
                i = 2*i + 1;
            }
            else {
                i = 2*i + 2;
            }
        }
        if (i >= TREE_SIZE) {
            printf("Tree is full\n");
            return;
```

```
        }
        bst[i] = data;
    }
}
```

Traversals

Preorder

```c
void bst_preorder(int *bst) {
    bst_preorder_helper(bst, 0);
}

// i is the root index of the iteration
void bst_preorder_helper(int *bst, int i) {
    if (bst[i] != 0) {
        // Root node
        printf("%d ", bst[i]);

        // Left node
        bst_preorder_helper(bst, 2*i+1);

        // Right node
        bst_preorder_helper(bst, 2*i+2);
    }
}
```

Inorder

```c
void bst_inorder(int *bst) {
    bst_inorder_helper(bst, 0);
}

// i is the root index of the iteration
void bst_inorder_helper(int *bst, int i) {
    if (bst[i] != 0) {
        // Left node
        bst_inorder_helper(bst, 2*i+1);

        // Root node
        printf("%d ", bst[i]);

        // Right node
        bst_inorder_helper(bst, 2*i+2);
```

```
    }
}
```

Postorder

```
void bst_postorder(int *bst) {
    bst_postorder_helper(bst, 0);
}

// i is the root index of the iteration
void bst_postorder_helper(int *bst, int i) {
    if (bst[i] != 0) {
        // Left node
        bst_postorder_helper(bst, 2*i+1);

        // Right node
        bst_postorder_helper(bst, 2*i+2);

        // Root node
        printf("%d ", bst[i]);
    }
}
```

7.0 Expression Tree

- An expression can be represented using the **Expression Tree** data structure
- Normally a postfix expression is used in constructing the Expression Tree
- Implemented using stack

Algorithm

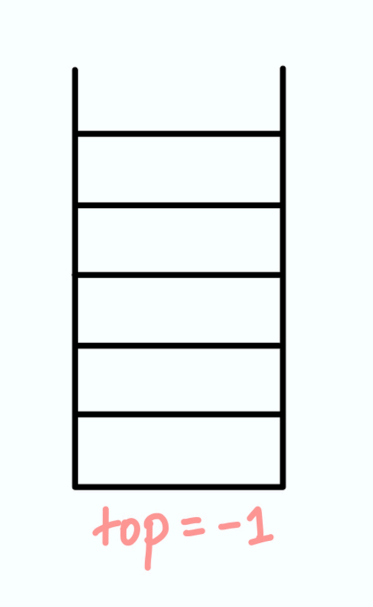
1. Convert to postfix
2. If character is an operand, push it onto stack
3. If character is an operator, create a node in the tree. Pop the top element from the stack and make it the node's right child. Pop the next element from the stack and make it the node's left child.

Example

Consider the expression $A + B * C$

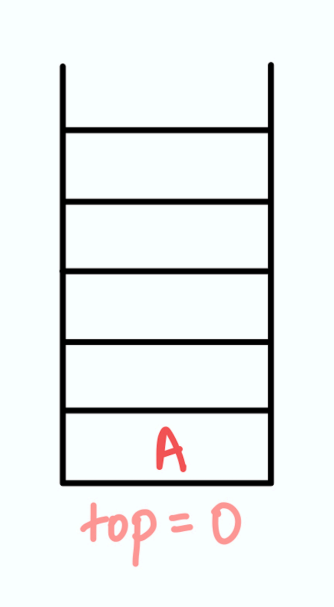
1. Convert to postfix

- $ABC * +$



2. Read character

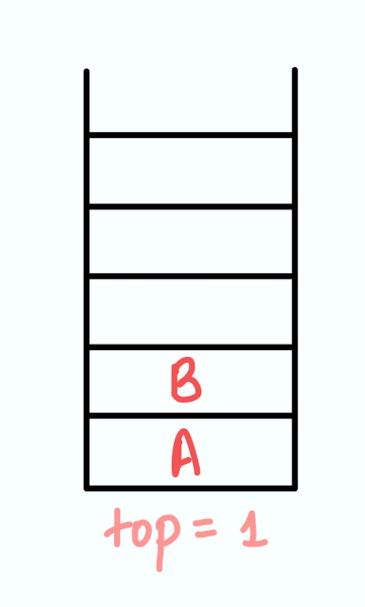
- Read A
- A is an operand
- Push to stack



3. Read character

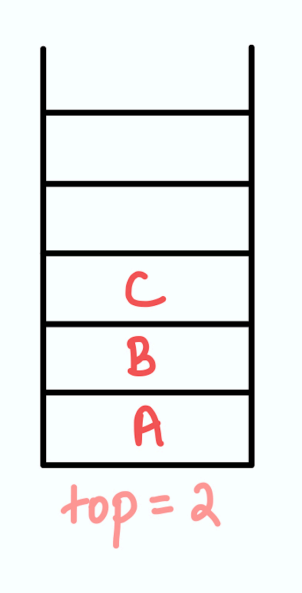
- Read B
- B is an operand

- Push to stack



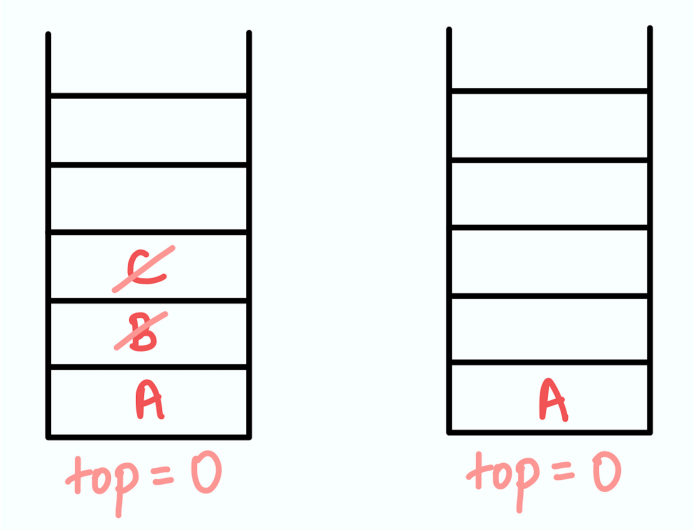
4. Read character

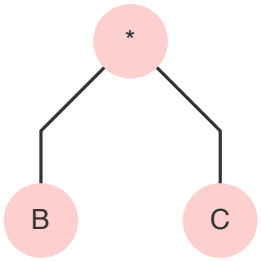
- Read C
- C is an operand
- Push to stack

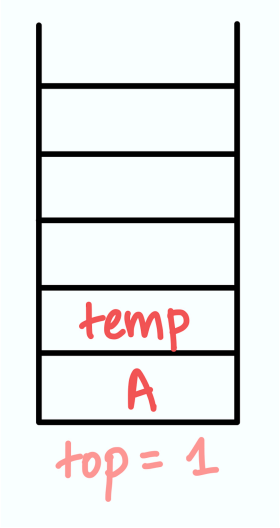


5. Read character

- Read $*$
- $*$ is an operator
- Create node (called $temp$) in tree with $*$
- Pop C and B from the stack
- Make C the right child node and B the left child node
- Push $temp$ $(stores *)$ to the stack

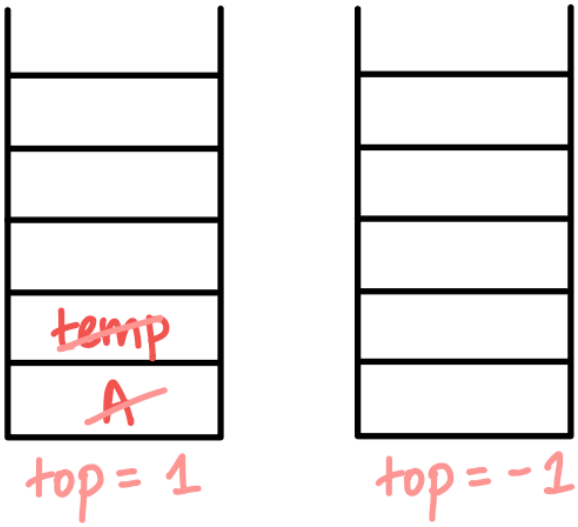


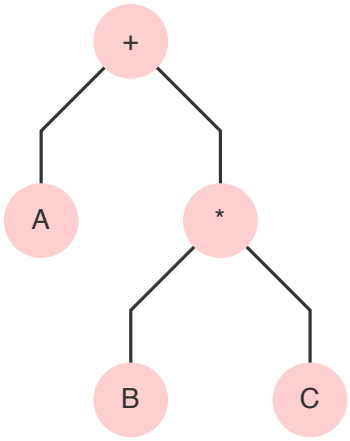




6. Read character

- Read $+$
- $+$ is an operator
- Create node (called $another\_temp$) in tree with $+$
- Pop $temp$ and A from the stack
- Make $temp$ the right child node and A the left child node





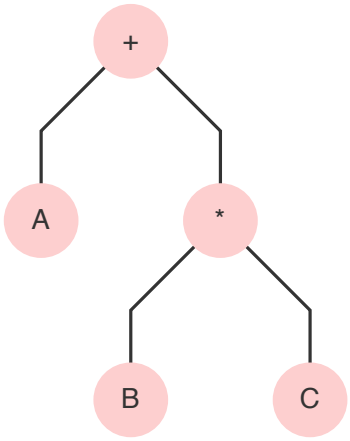
7. Read character

- Read $null$
- Exit

Traversing Expression Tree

Preorder

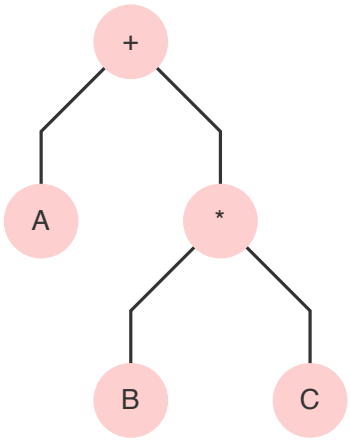
- Consider the same tree



- Preorder traversal: $+A*BC$
- Gives prefix version of expression

Inorder

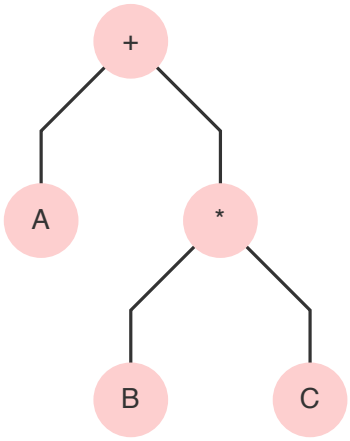
- Consider the same tree



- Inorder traversal: $A+B*C$
- Gives infix version of expression

Postorder

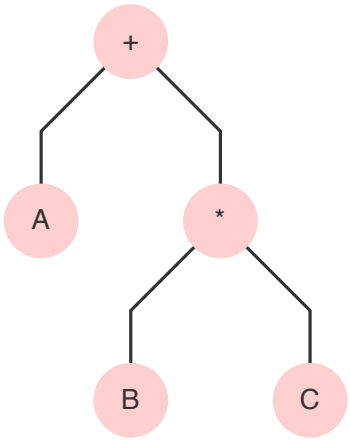
- Consider the same tree



- Postorder traversal: $ABC*+$
- Gives postfix version of expression

Evaluation

- Consider the same tree



- Recursive function `evaluate(node *)`
- If the node is an operator X, return `evaluate(node->left)` X `evaluate(node->right)` (where X can be `+`, `-`, `*`, `/`, `^`)

Code Implementation

Structures for `tree` and `node`

```c
typedef struct node {
    char ch;
    struct node *left, *right;
} Node;

typedef struct tree {
    Node *root;
} Tree;
```

Structure Helpers

```c
// Create a node
Node *create_node(char ch) {
    Node *new_node = (Node *)malloc(sizeof(Node));
    new_node->ch = ch;
    new_node->left = NULL;
    new_node->right = NULL;
    return new_node;
}

// Initialise tree
Tree *initialise_tree() {
    Tree *new_tree = (Tree *)malloc(sizeof(Tree));
    new_tree->root = NULL;
    return new_tree;
}

// Destroy tree
void destroy_tree(Tree *tree) {
    if (tree == NULL) return;
    destroy_nodes(tree->root);
    free(tree);
}

// Destroy all nodes
void destroy_nodes(Node *root) {
    if (root == NULL) return;
    if (root->left == NULL && root->right == NULL) {
        free(root);
        return;
    }
    if (root->left != NULL) {
        destroy_nodes(root->left);
    }
```

```
34
35      if (root->right != NULL) {
36          destroy_nodes(root->right);
37      }
38  }
```

Creating the Tree

```
1   void create_exp_tree(Tree *tree, char *exp) {
2       char ch;
3       Node *stack[STACK_SIZE];
4       int top = -1;
5       Node *temp = NULL;
6
7       for (int i = 0; exp[i] != 0; ++i) {
8           ch = exp[i];
9           temp = create_node(ch);
10
11          // If the char is an operator
12          if (is_oper(ch)) {
13
14              // Pop the top two from stack
15              temp->right = pop(stack, &top);
16              temp->left = pop(stack, &top);
17
18              // Push the address of the operator to stack
19              push(stack, temp, &top);
20          }
21
22          // Otherwise, push operand to stack
23          else {
24              push(stack, temp, &top);
25          }
26      }
27      tree->root = pop(stack, &top);
28      stack_destroy(stack, top);
29  }
30
31  int is_oper(char ch) {
32      switch (ch) {
33          case '+':
34          case '-':
35          case '*':
36          case '/':
37          case '^':
38              return 1;
39          default:
```

```
            return 0;
    }
}
```

Preorder

```
void exp_preorder(Tree *tree) {
    exp_preorder_helper(tree->root);
    printf("\n");
}

void exp_preorder_helper(Node *root) {
    if (root != NULL) {
        printf("%c ", root->ch);
        exp_preorder_helper(root->left);
        exp_preorder_helper(root->right);
    }
}
```

Inorder

```
void exp_inorder(Tree *tree) {
    exp_inorder_helper(tree->root);
    printf("\n");
}

void exp_inorder_helper(Node *root) {
    if (root != NULL) {
        exp_inorder_helper(root->left);
        printf("%c ", root->ch);
        exp_inorder_helper(root->right);
    }
}
```

Postorder

```c
void exp_postorder(Tree *tree) {
    exp_postorder_helper(tree->root);
    printf("\n");
}

void exp_postorder_helper(Node *root) {
    if (root != NULL) {
        exp_postorder_helper(root->left);
        exp_postorder_helper(root->right);
        printf("%c ", root->ch);
    }
}
```

Evaluation

```c
int evaluate(Tree *tree) {
    if (tree == NULL || tree->root == NULL) return 0;
    return evaluate_helper(tree->root);
}

// Recursive function
int evaluate_helper(Node *root) {
    int x;
    if (root == NULL) return 0;
    switch (root->ch) {
        case '+': {
            return evaluate_helper(root->left) + evaluate_helper(root->right);
        }
        case '-': {
            return evaluate_helper(root->left) - evaluate_helper(root->right);
        }
        case '^': {
            return powerof(evaluate_helper(root->left), evaluate_helper(root->right));
        }
        case '/': {
            return evaluate_helper(root->left) / evaluate_helper(root->right);
        }
        case '*': {
            return evaluate_helper(root->left) * evaluate_helper(root->right);
        }
        default: {
```

```
            printf("Enter value of %c: ", root->ch);
            scanf("%d", &x);
            return x;
        }
    }
}

int powerof(int a, int b) {
    int res = 1;
    for (int i = 0; i < b; ++i) {
        res *= a;
    }
    return res;
}
```

Output

```
Enter postfix expression: ABC*+
Inorder form:
A + B * C
Preorder form:
+ A * B C
Postorder form:
A B C * +
Enter value of A: 2
Enter value of B: 4
Enter value of C: 3
Evaluated expression: 14
```

8.0 Iterative Tree Traversal

- Using stack of `node` pointers
- Avoid function overheads

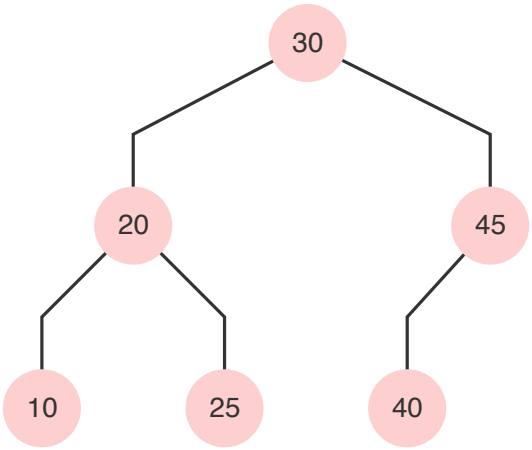
Inorder

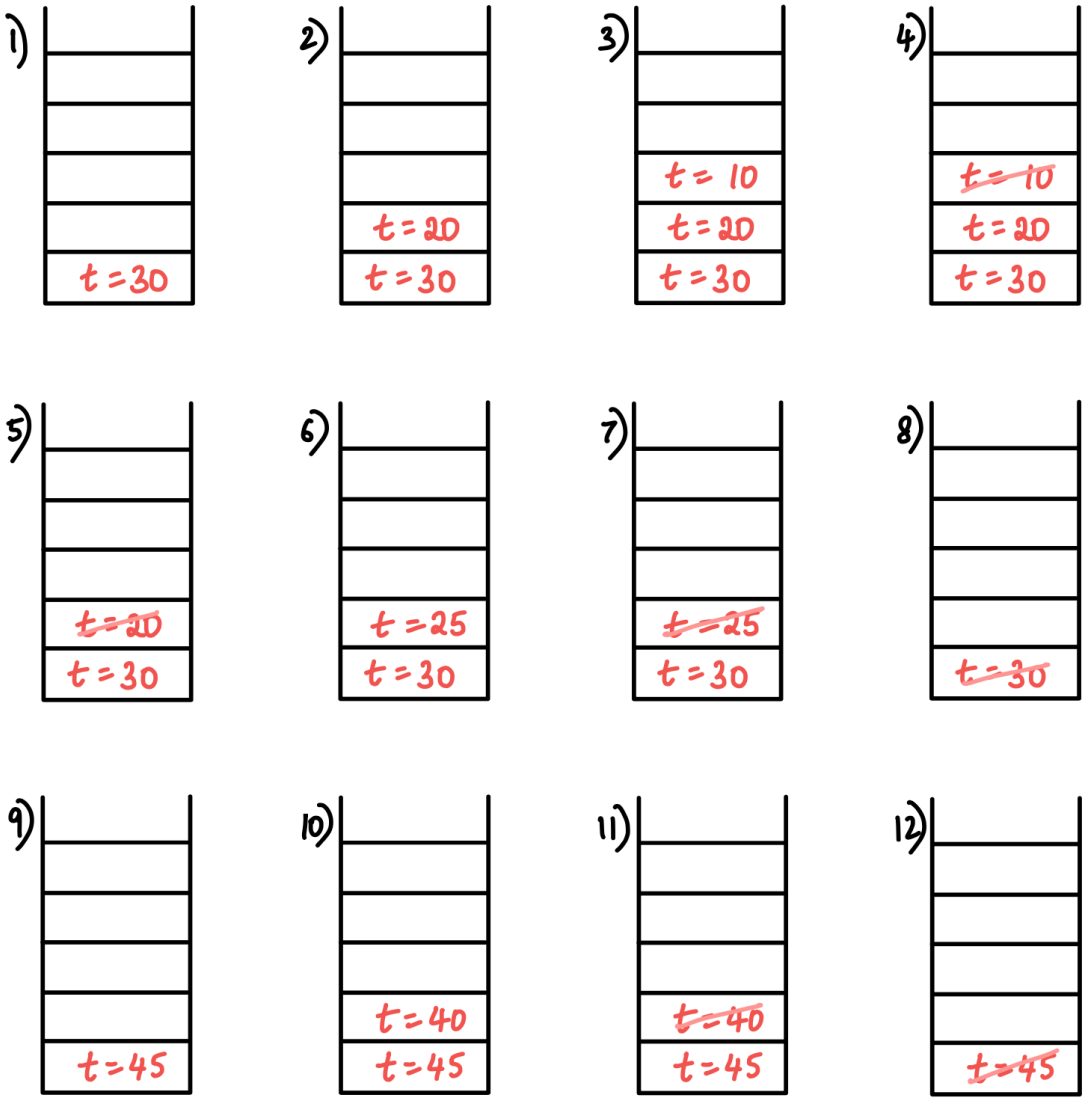
Algorithm

1. Start with `temp` as root node
2. Travel down as left as possible, pushing each node onto the stack
3. Once left subtree is empty, visit the node and pop from the stack
4. Visit the right subtree and go to step 2

Stack Sequence

- Consider the tree shown below





final sequence: 10, 20, 25, 30, 40, 45

Preorder & Postorder

- Similar to inorder
- Instead of traversing left, printing and then traversing right, the order changes
- Postorder requires two stacks
- Go through code for better understanding

Code Implementation

Structures for `tree` and `node`

```c
typedef struct node {
    int data;
    struct node *left, *right;
} Node;

typedef struct tree {
    Node *root;
} Tree;
```

Insertion

```c
void lex_insert(Tree *tree, int data) {
    Node *new_node = create_node(data);

    if (tree->root == NULL) {
        tree->root = new_node;
        return;
    }

    Node *temp = tree->root;

    while (1) {
        if (data < temp->data) {
            if (temp->left == NULL) {
                temp->left = new_node;
                return;
            }
            else {
                temp = temp->left;
            }
        }
        else {
            if (temp->right == NULL) {
                temp->right = new_node;
                return;
            }
            else {
                temp = temp->right;
            }
        }
    }
}
```

Inorder

```c
void iter_inorder(Tree *tree) {
    Node *stack[STACK_SIZE];
    int top = -1;

    Node *temp = tree->root;

    while (top != -1 || temp != NULL) {
        while (temp != NULL) {
            push(stack, temp, &top);
            temp = temp->left;
        }
        temp = pop(stack, &top);
        printf("%d ", temp->data);
        temp = temp->right;
    }
    printf("\n");
}
```

Preorder

```c
void iter_preorder(Tree *tree) {
    Node *stack[STACK_SIZE];
    int top = -1;

    Node *temp = tree->root;
    push(stack, temp, &top);

    while (top != -1) {
        temp = pop(stack, &top);
        printf("%d ", temp->data);

        // Right child is pushed first so that
        // left is processed first
        if (temp->right != NULL) {
            push(stack, temp->right, &top);
        }

        if (temp->left != NULL) {
            push(stack, temp->left, &top);
        }
    }
    printf("\n");
}
```

Postorder

```
void iter_postorder(Tree *tree) {
    Node *s1[STACK_SIZE], *s2[STACK_SIZE];
    int t1 = -1, t2 = -1;

    Node *temp = tree->root;
    push(s1, temp, &t1);

    while (t1 != -1) {
        temp = pop(s1, &t1);
        push(s2, temp, &t2);

        if (temp->left != NULL) {
            push(s1, temp->left, &t1);
        }

        if (temp->right != NULL) {
            push(s1, temp->right, &t1);
        }
    }

    while (t2 != -1) {
        temp = pop(s2, &t2);
        printf("%d ", temp->data);
    }
    printf("\n");
}
```

9.0 Threaded BST

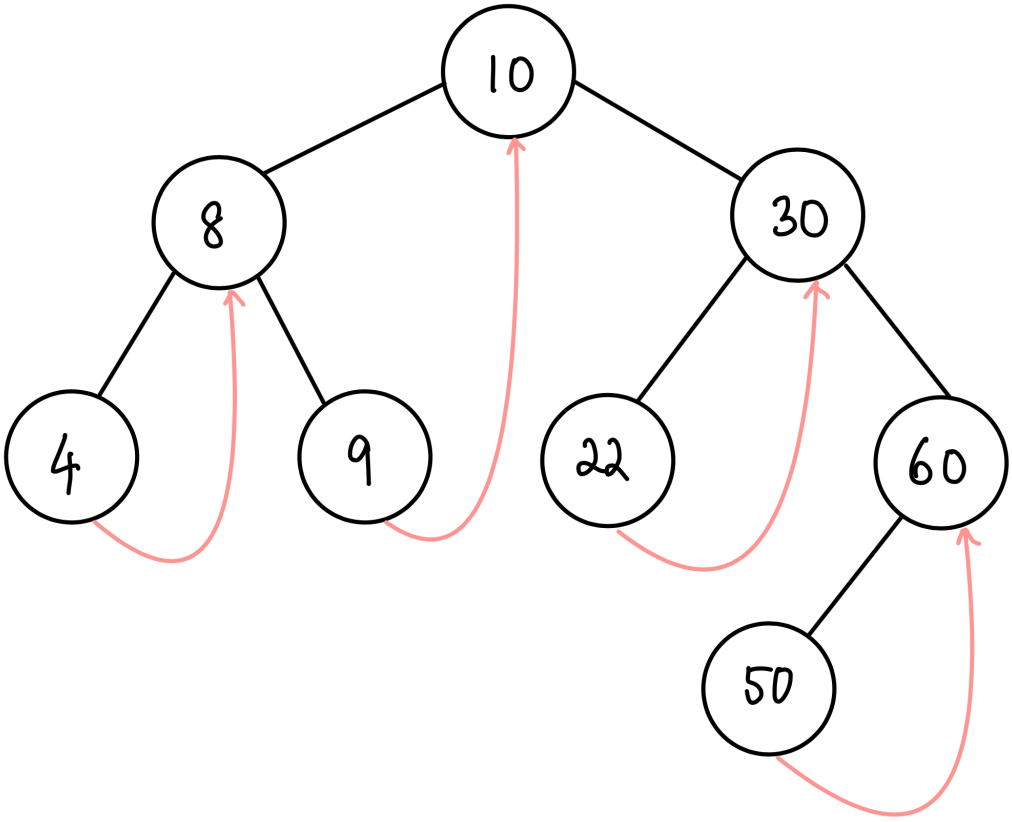
- Iterative Inorder Traversal requires explicit stack - costly
- Node addressed needed to be stored in stack
- Threaded BST uses no stack

Types

- We can use the right pointer of a node to point to the inorder successor if a right child does not exist. This is called a right thread. Such a tree is called **Right-In Threaded** Binary Tree.
- We can use the left pointer of a node to point to the inorder predecessor if a eft child does not exist. This is called a left thread. Such a tree is called **Left-In Threaded** Binary Tree.
- If we use both the pointers, the tree is called **In Threaded** Binary Tree

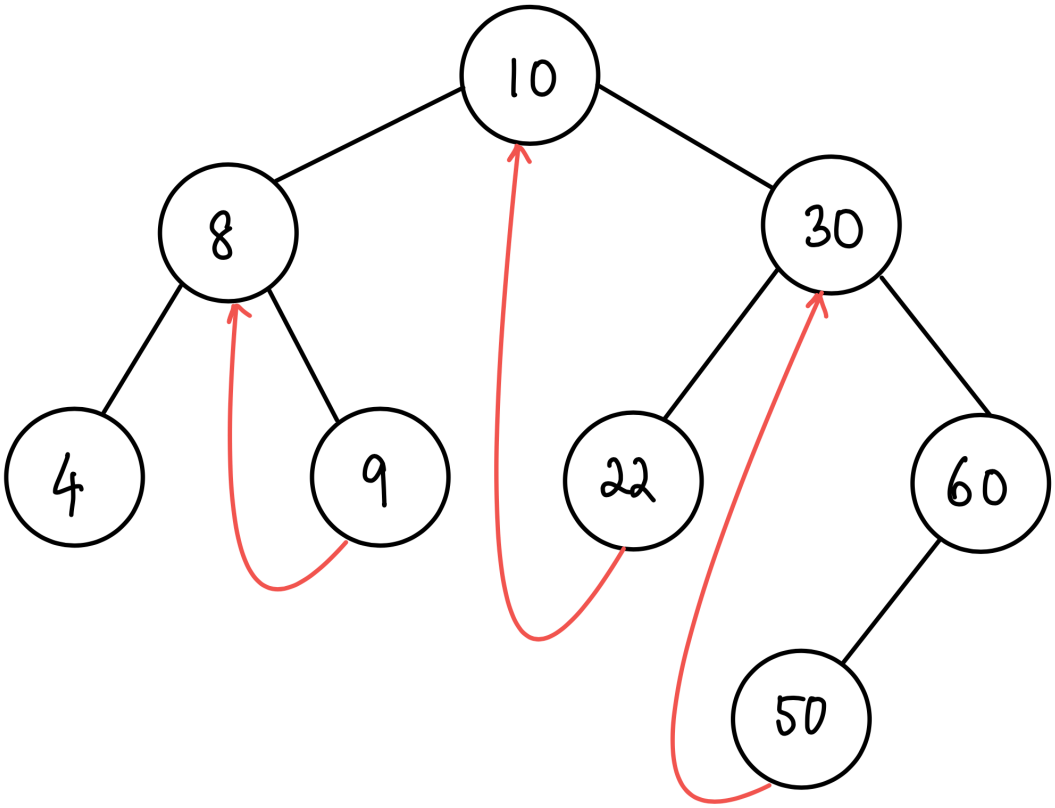
Right-In Threaded Binary Tree

- We have a boolean value in each node called `right_thread`
- `right_thread` is true if the node does not have a right child and false otherwise
- The right child stores the value of the inorder successor if `right_thread` is true



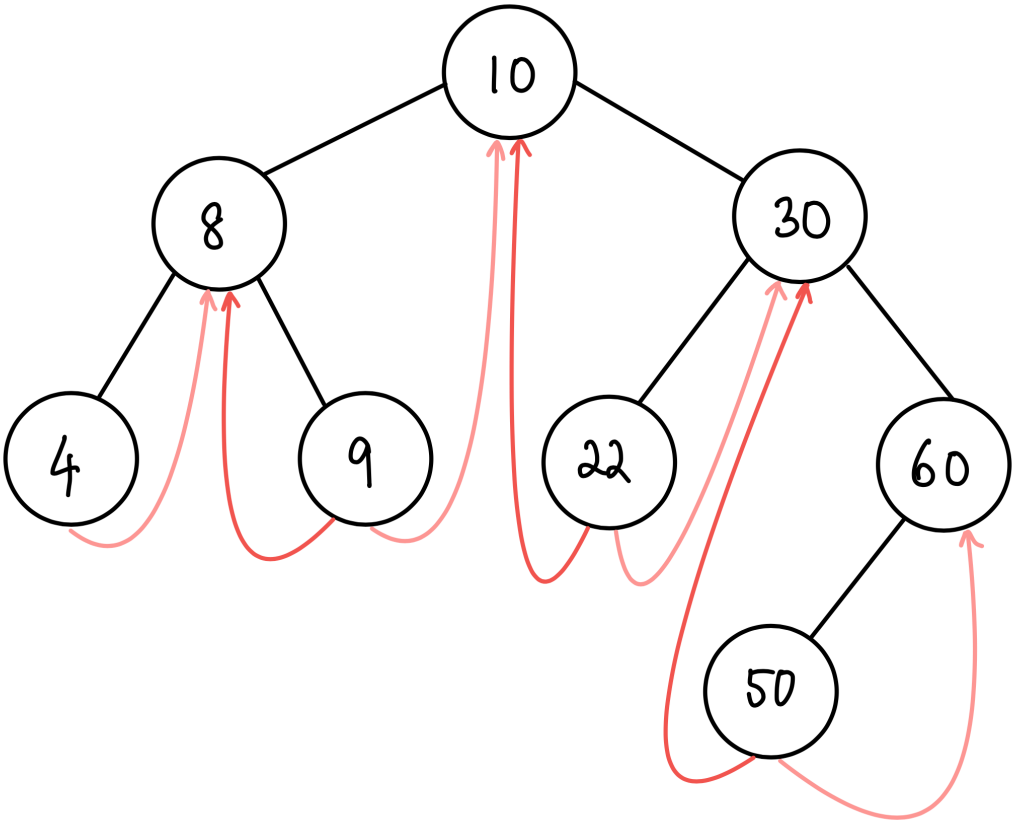
Left-In Threaded Binary Tree

- We have a boolean value in each node called `left_thread`
- `left_thread` is true if the node does not have a left child and false otherwise
- The left child stores the value of the inorder predecessor if `left_thread` is true



In Threaded Binary Tree

- We have two boolean values in each node called `right_thread` and `left_thread`
- `right_thread` is true if the node does not have a right child and false otherwise
- The right child stores the value of the inorder successor if `right_thread` is true
- `left_thread` is true if the node does not have a left child and false otherwise
- The left child stores the value of the inorder predecessor if `left_thread` is true



Code Implementation

Structures for `tree` and `node`

```c
typedef struct node {
    int data;
    struct node *left, *right;
    int left_thread, right_thread;
} Node;

typedef struct tree {
    Node *root;
} Tree;
```

Insertion

```c
void thread_insert(Tree *tree, int data) {
    Node *new_node = create_node(data), *temp = tree->root;

    if (temp == NULL) {
        tree->root = new_node;
        return;
```

```
    }

    // Search for position to insert and then break
    while (1) {
        if (data < temp->data) {
            if (temp->left_thread == 0) {
                temp = temp->left;
            }
            else {
                break;
            }

        }
        else {
            if (temp->right_thread == 0) {
                temp = temp->right;
            }
            else {
                break;
            }
        }
    }

    // Attach to either right or left
    if (data < temp->data) {
        // Assign right_thread for new_node
        new_node->left = temp->left;
        new_node->right = temp;

        temp->left_thread = 0;
        temp->left = new_node;
    }
    else {
        // Assign left_thread for new_node
        new_node->right = temp->right;
        new_node->left = temp;

        temp->right_thread = 0;
        temp->right = new_node;
    }
}
```

Inorder

```c
void thread_inorder(Tree *tree) {
    Node *temp = tree->root;

    if (temp == NULL) {
        printf("Empty tree\n");
        return;
    }

    // Go to leftmost node
    while (temp->left_thread == 0) {
        temp = temp->left;
    }

    // Print inorder successors
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = inorder_successor(temp);
    }
}

Node *inorder_successor(Node *root) {
    if (root == NULL) {
        return NULL;
    }

    if (root->right_thread == 1) {
        root = root->right;
    }
    else {
        root = root->right;
        while (root->left != NULL && root->left_thread == 0) {
            root = root->left;
        }
    }
    return root;
}
```

10.0 Constructing a Tree Given Traversals

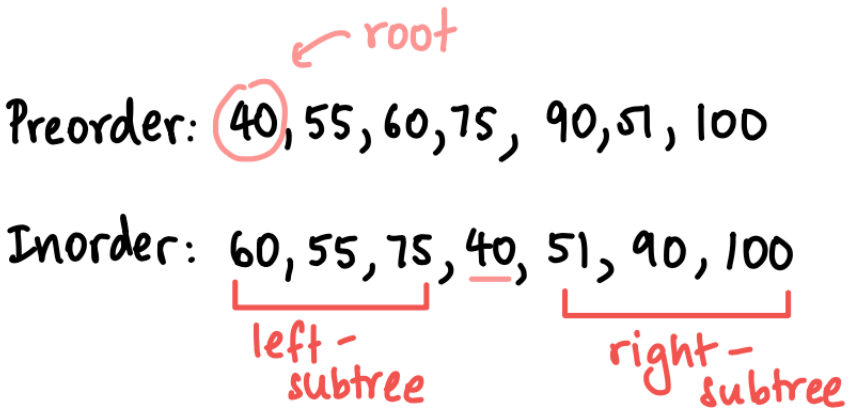
- Inorder and either postorder or preorder
- Not necessarily BST

Inorder and Preorder

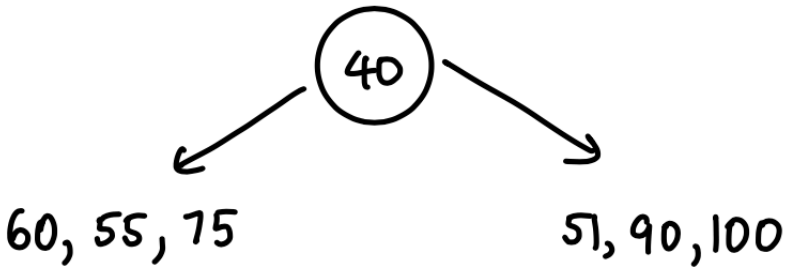
- First element of preorder is root node
- In inorder, everything left of root is left subtree and everything right of root is right subtree

Example

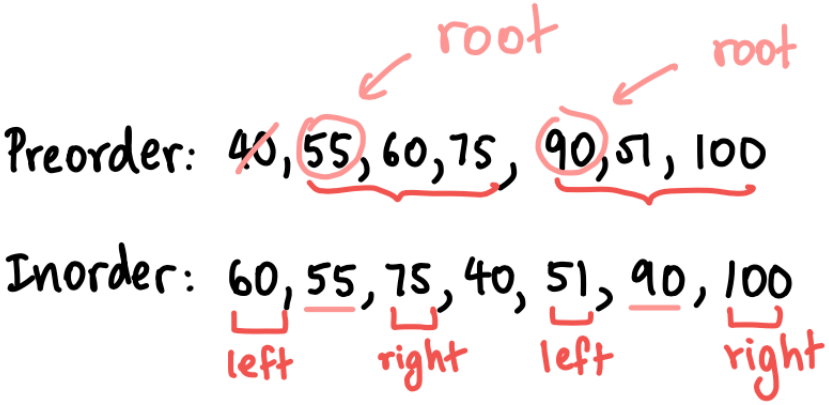
- Preorder: $40, 55, 60, 75, 90, 51, 100$
- Inorder: $60, 55, 75, 40, 51, 90, 100$

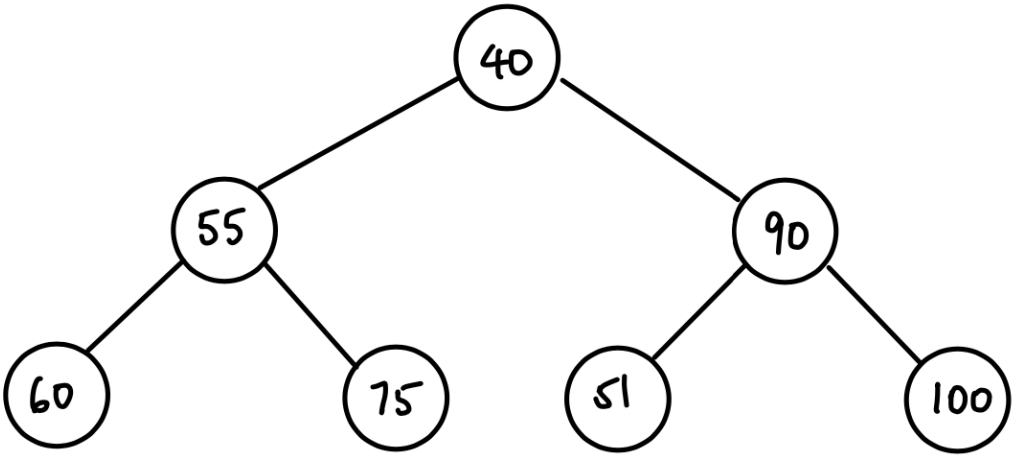


- Locate left and right subtrees



- Repeat the process



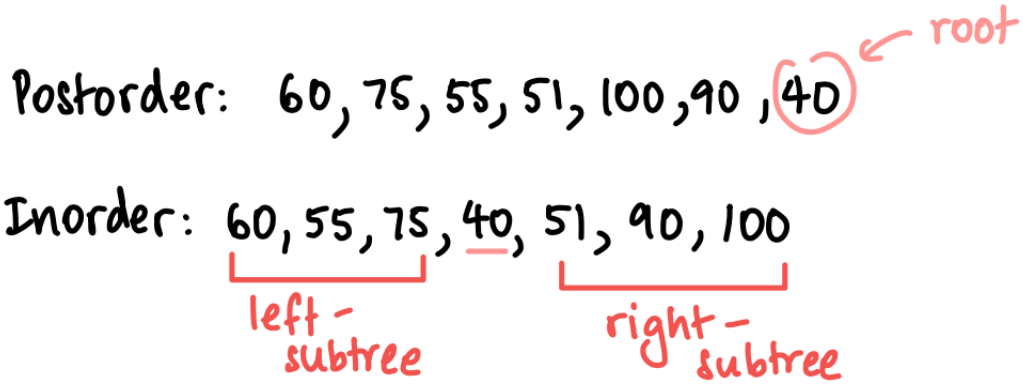


Inorder and Postorder

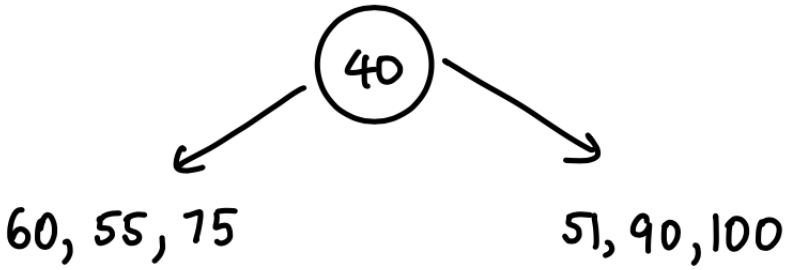
- Last node in postorder is the root node
- Same logic

Example

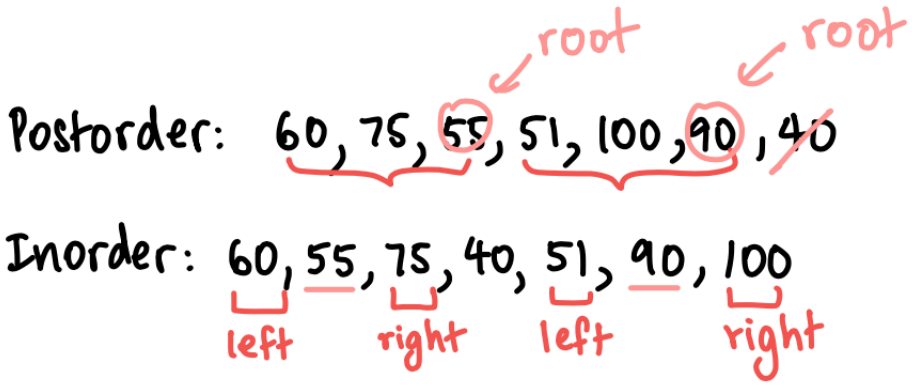
- Postorder: $60, 75, 55, 51, 100, 90, 40$
- Inorder: $60, 55, 75, 40, 51, 90, 100$

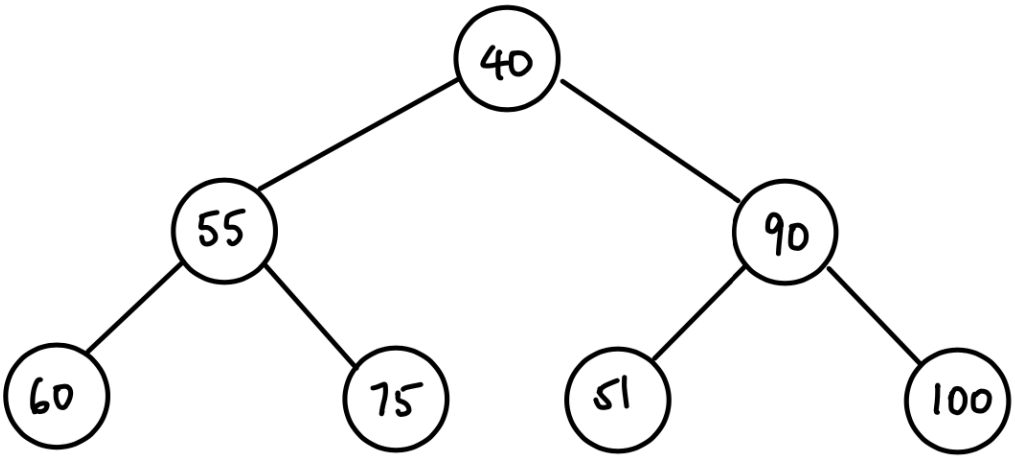


- Locate left and right subtrees



- Repeat the process





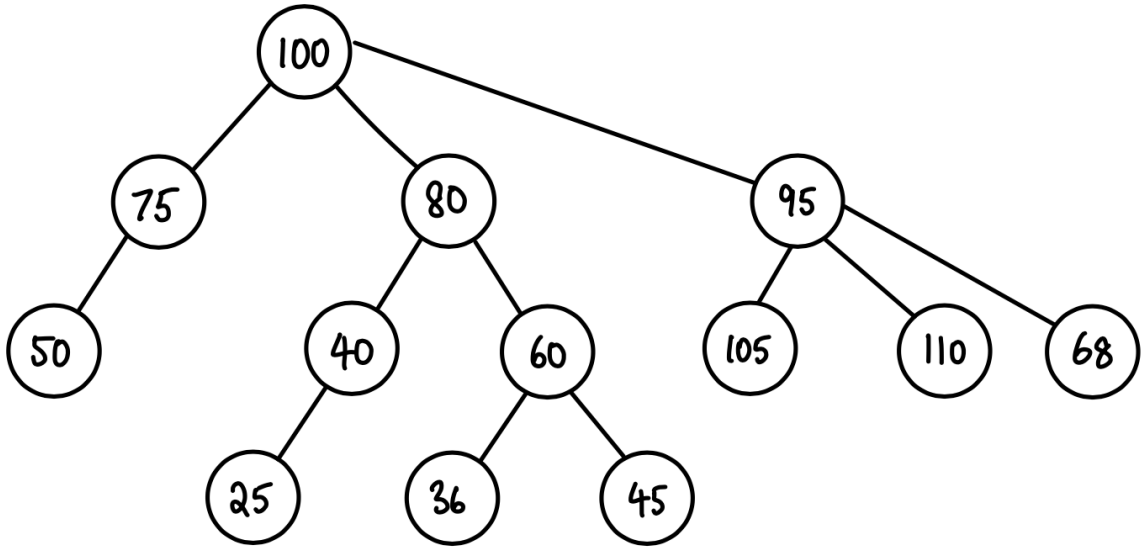
11.0 N-ary Tree to Binary Tree

Intermediary step

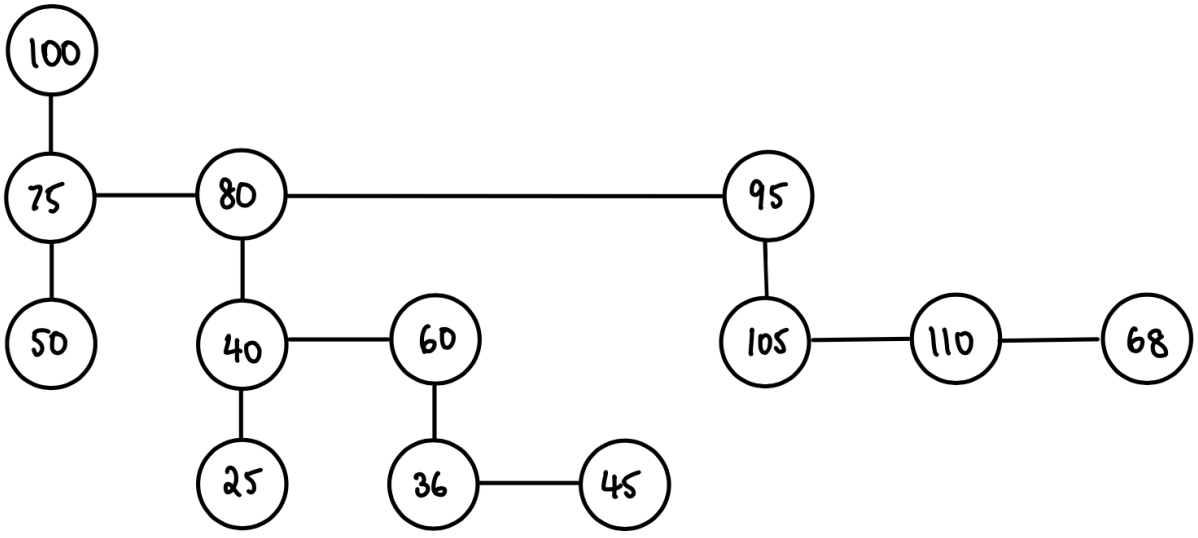
- Write siblings on the same level, connected with an edge
- Write the leftmost child node directly underneath the root

Example

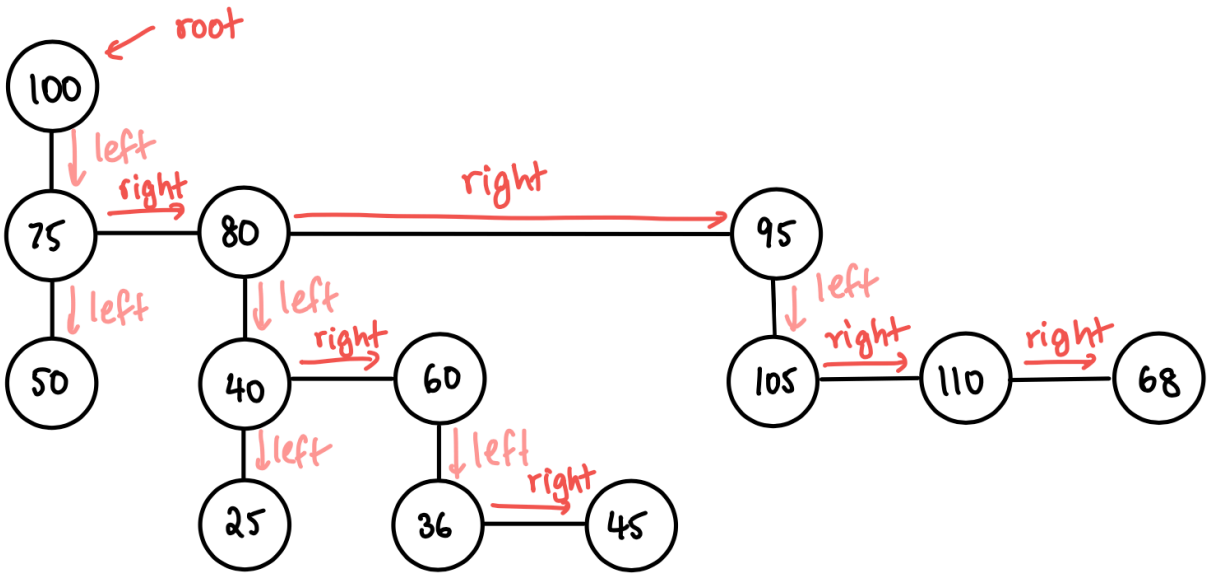
Consider the N-ary tree shown



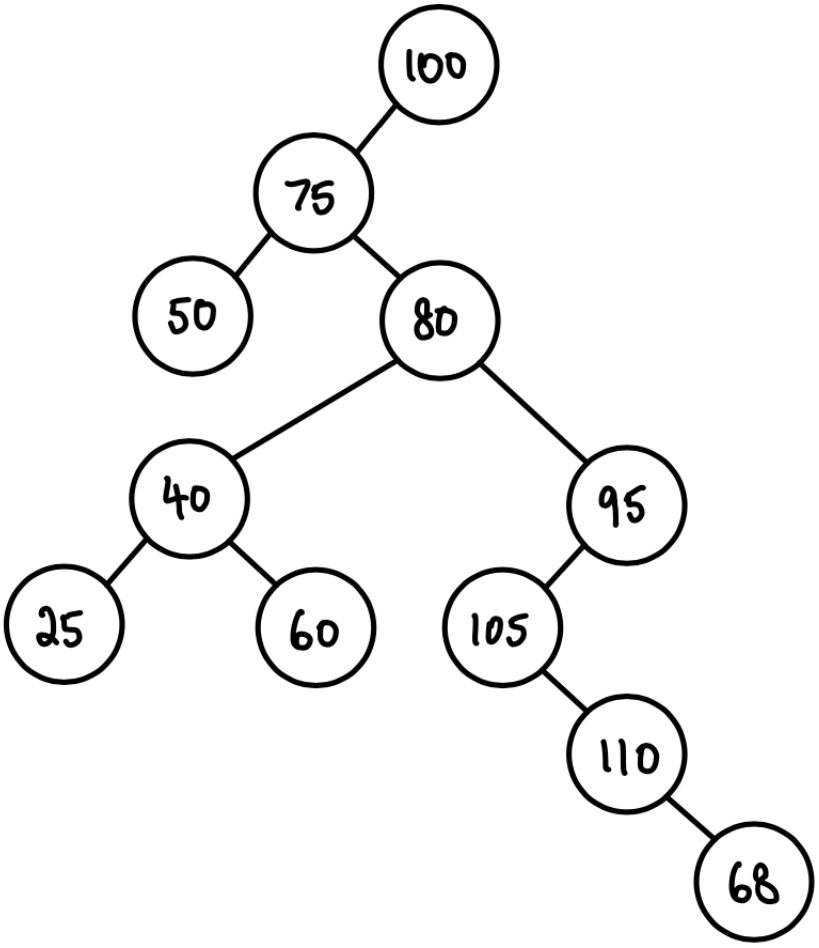
Siblings are connected horizontally



Convert to binary


root
100
left
right
75
80
right
95
left
left
left
right
right
50
40
60
105
110
68
left
left
right
25
36
45

Redraw

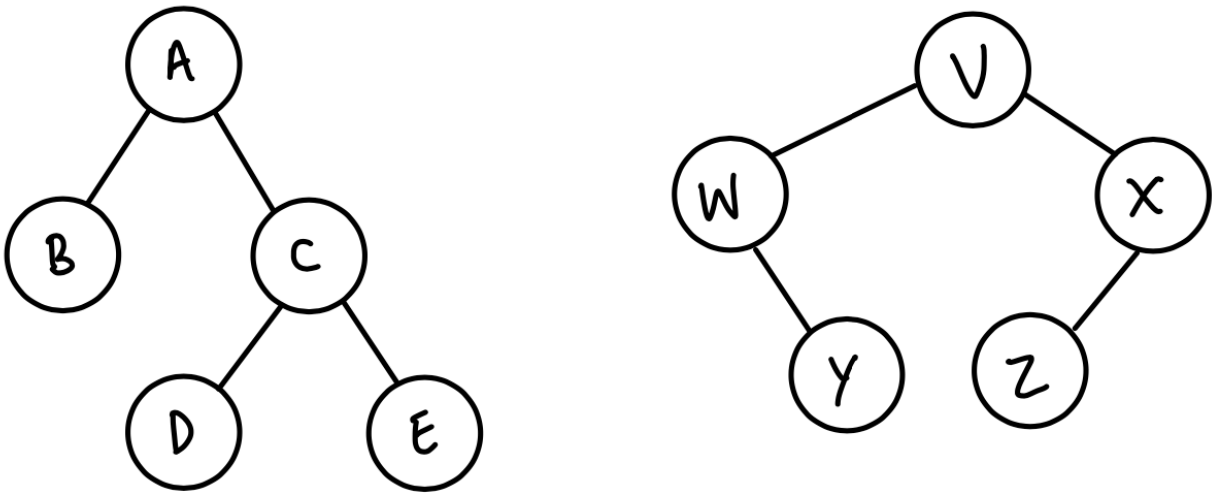

100
75
50
80
40
95
25
60
105
110
68

Convert Forest to Binary Tree

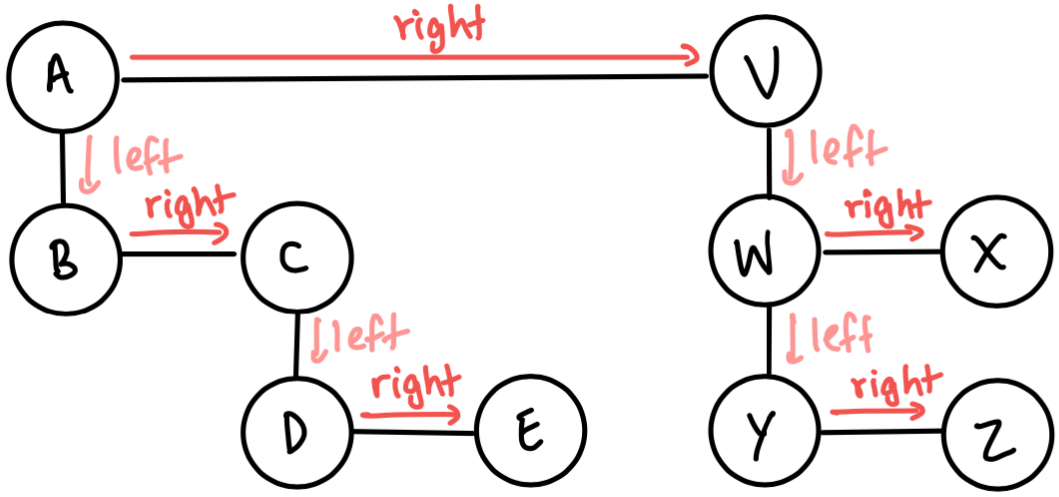
- Same steps, consider all root nodes to be siblings on the same level

Example

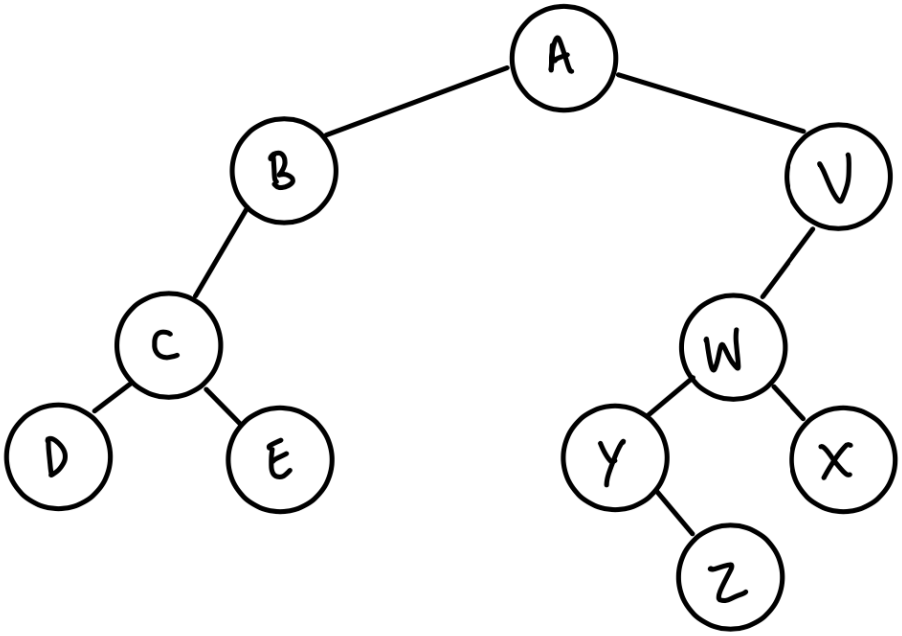
- Consider the two trees
- Assume A and V are siblings



- Intermediary step



- Convert to binary tree

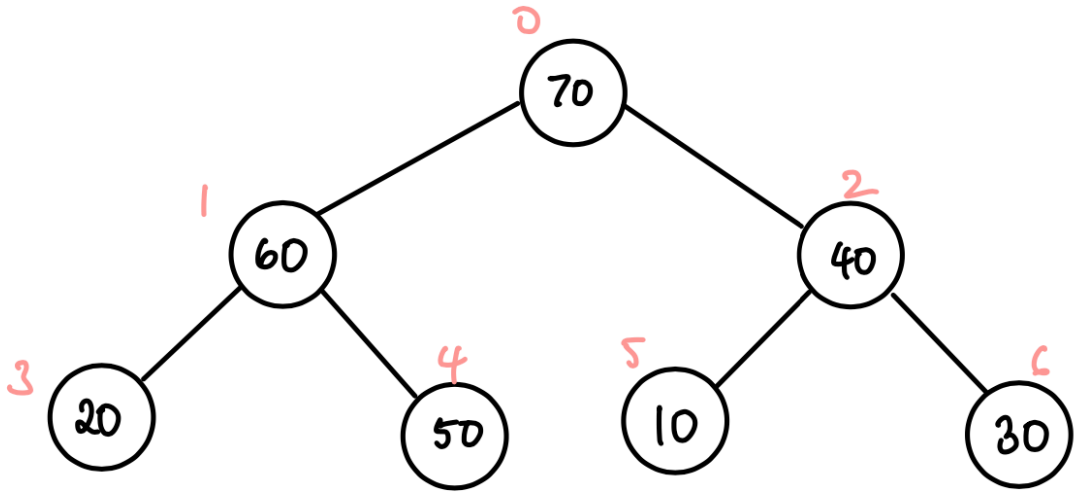


12.0 Heap

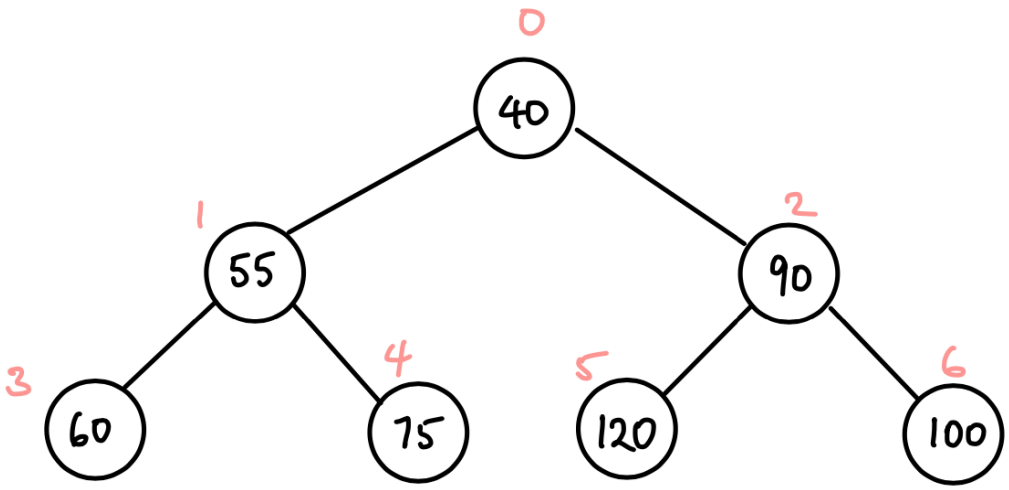
- Collection of objects/elements that are highly unorganised
- Maximum and minimum heap
- Two conditions
  - **Tree's shape** - must be a complete binary tree
  - **Parental dominance** - key at each node is greater than or equal to keys of its children (maximum heap) or lesser than or equal to its children (minimum heap)

Types of Heaps

Max Heap

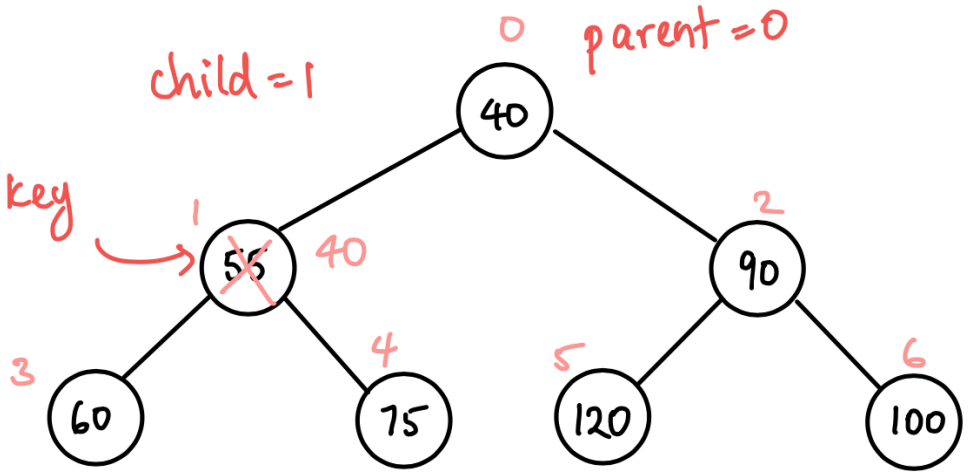


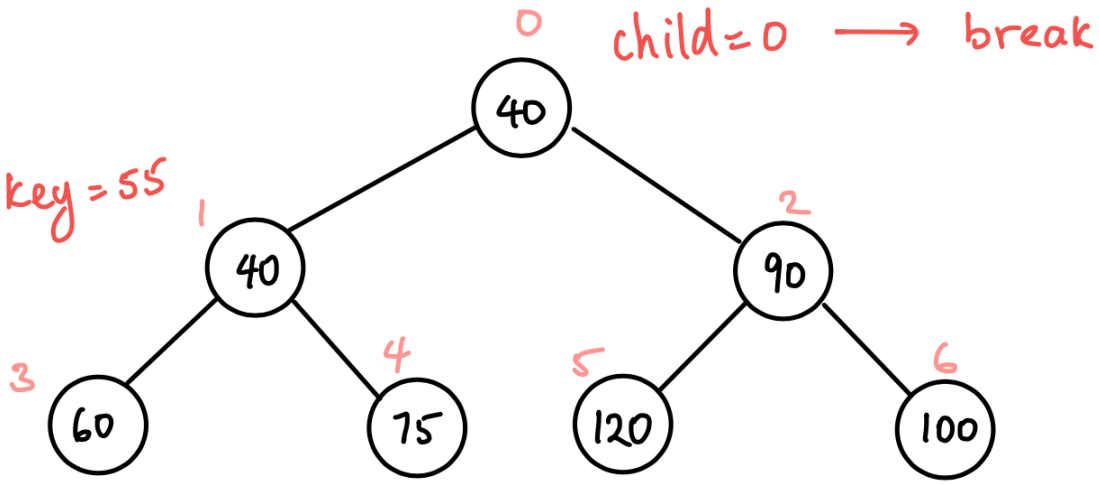
Min Heap

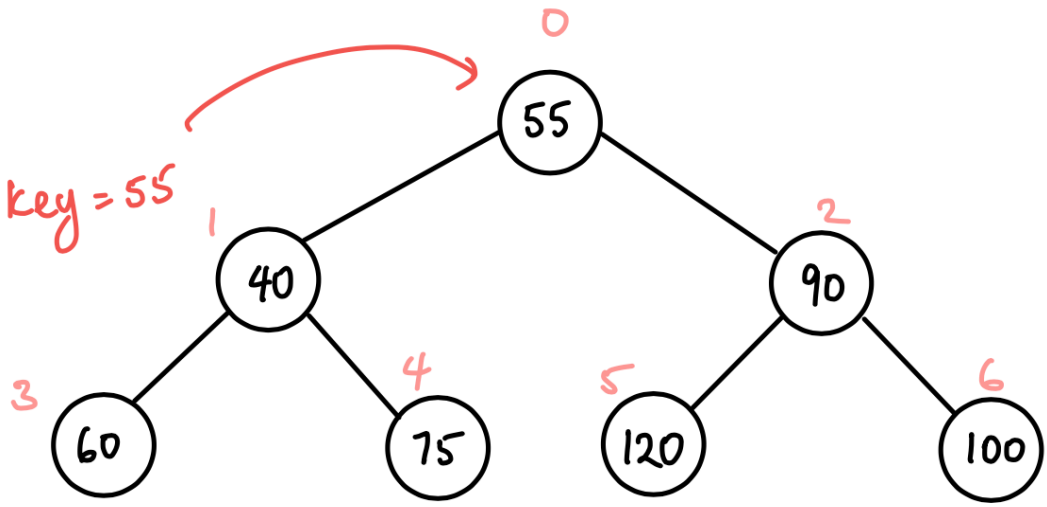


Max Heapify

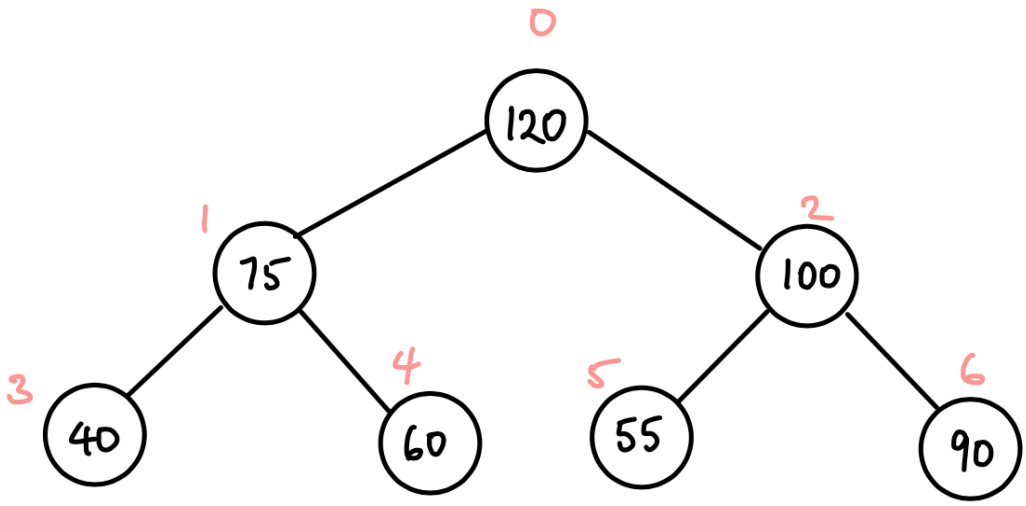
- Somewhat similar to insertion sort
- Convert a binary tree to a heap







- Repeat until fully heapified



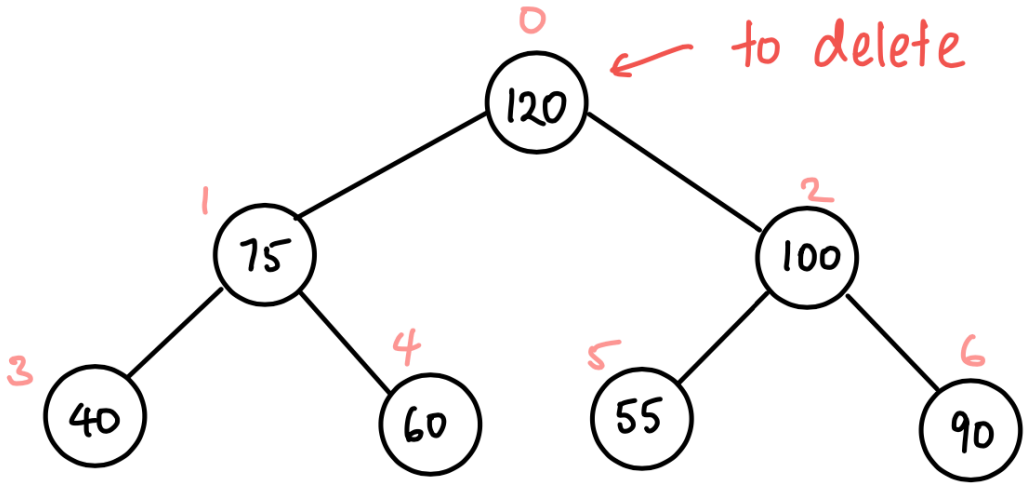
Code Implementation

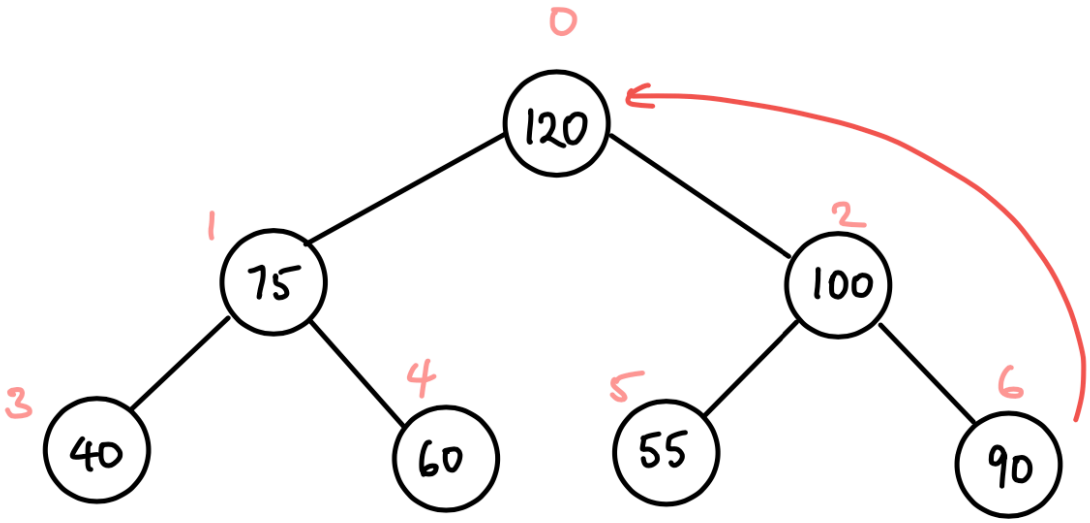
- Array implementation
- Left child - $2 \times i + 1$
- Right child - $2 \times i + 2$
- Process of creating a heap given a set of locations: **heapify**

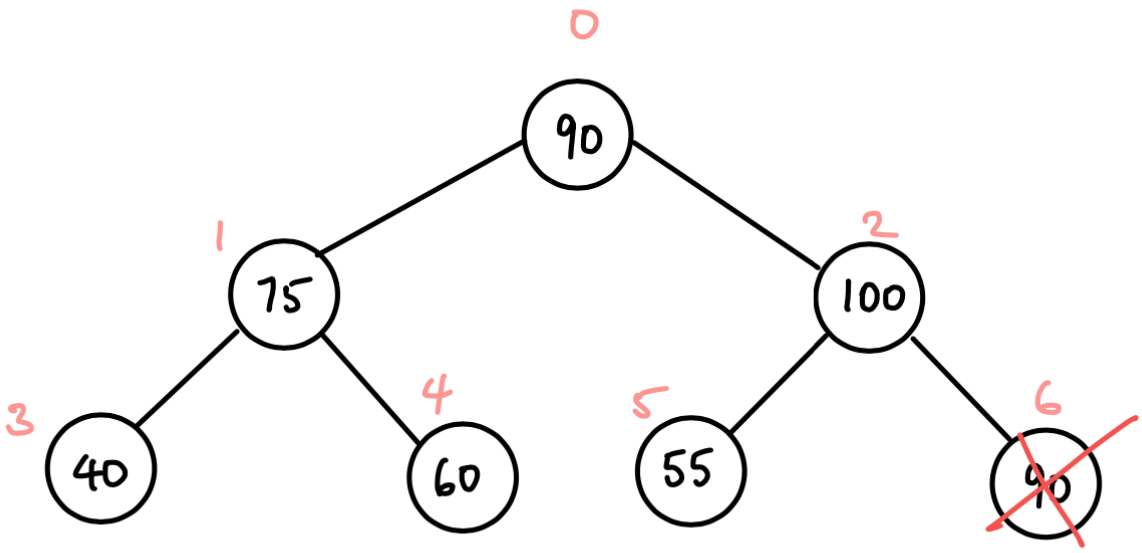
```
void max_heapify(int *heap, int n) {
    int key, child, parent, k;

    for (k = 1; k < n; ++k) {
        child = k;
        key = heap[child];

        // parent uses inverse of child formula 2*i+1
        parent = (child-1)/2;

        while ((child > 0) && heap[parent] < key) {
            heap[child] = heap[parent];

            // Go up
            child = parent;
            parent = (child-1)/2;
        }
        heap[child] = key;
    }
}
```

Deleting the Max Node

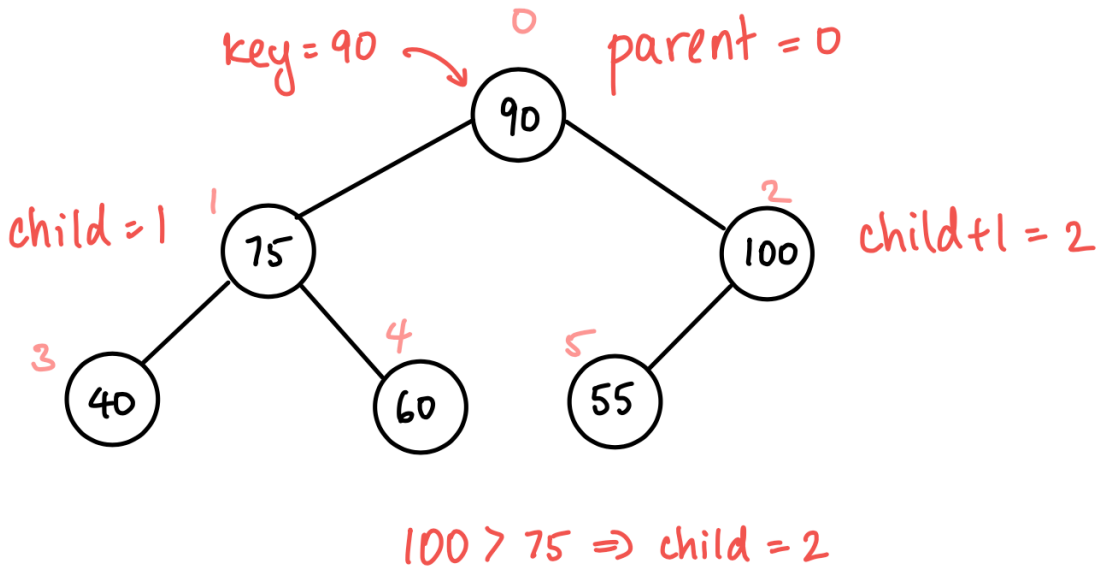
- Top down approach
- Have to remove root node
- If root node is simply removed, tree becomes a forest
- Before removing root node, it is swapped with rightmost node (here, count - 1)
- The last node (rightmost node containing old root) is then deleted and the tree is to be heapified again
- Instead of calling `heapify` once more, a more efficient `adjust` method is called that uses the top down approach

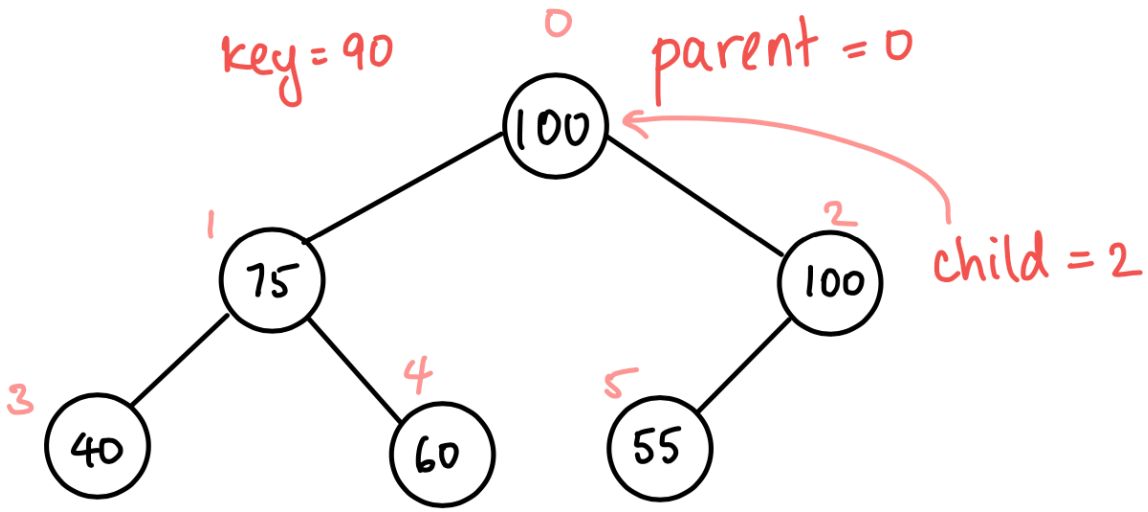


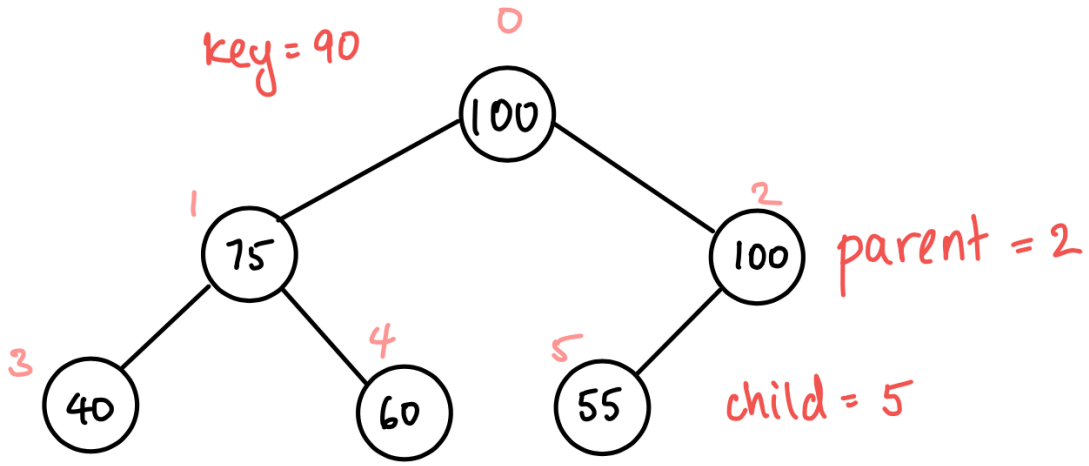


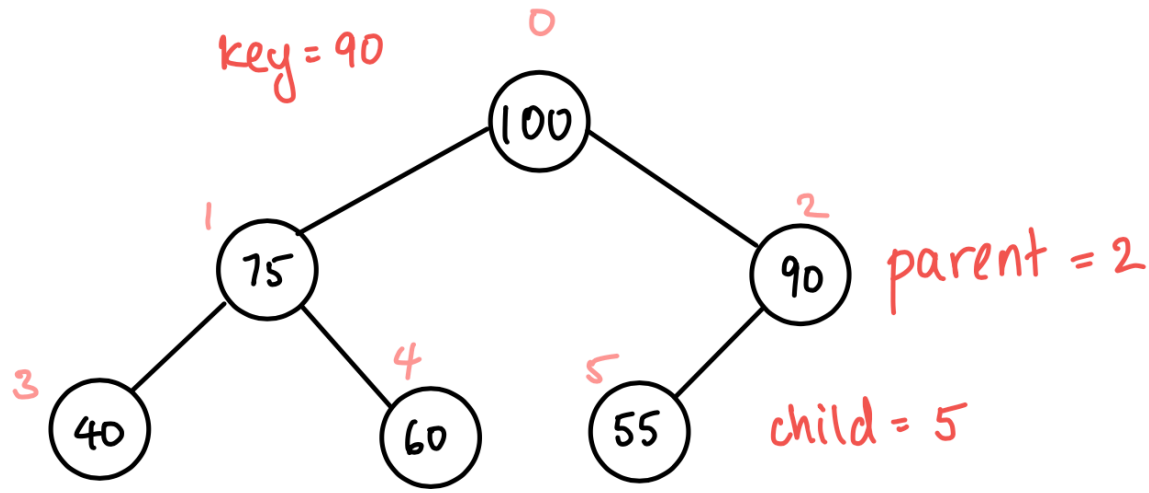


- Readjust (less time than to re-heapify)




key = 90
0
parent = 0
100
1
75
2
100
child = 2
3
40
4
60
5
55


key = 90
0
100
1
75
2
100
parent = 2
3
40
4
60
5
55
child = 5


key = 90
0
100
1
75
2
90
parent = 2
3
40
4
60
5
55
child = 5

Code Implementation

```
int max_remove(int *heap, int *pn) {
    // Root node to delete
    int top = heap[0];

    // Assign last element to first element of heap
    heap[0] = heap[*pn-1];

    // Reduce number of nodes
    --(*pn);

    // Call adjust function
    adjust(heap, *pn);
    return top;
}

// More efficient than heapify
void adjust(int *heap, int n) {
    int child, parent, key;
    parent = 0;
    key = heap[parent];

    child = 2*parent+1;

    // If left child exists
    while (child < n) {

        // Right child exists
        if (child+1 < n) {
            // If right node is greater than left
            if (heap[child+1] > heap[child]) {
                ++child;
            }
        }

        // Now child stores index of greater child
        if (key < heap[child]) {
            heap[parent] = heap[child];
            parent = child;
            child = 2*parent+1;
        }
        else {
            break;
        }
    }
    heap[parent] = key;
}
```

Priority Queue Using Heap

- Minimum heap with priorities as keys

Code Implementation

Structure for node and constants

```c
#define HEAP_SIZE 10

typedef struct node {
    int data, priority;
} Node;

// Inside main - static array (not global)
Node priority_queue[HEAP_SIZE];
```

Insert into priority queue

```c
void pq_insert(Node *heap, int data, int priority, int *pn) {
    int child, parent;

    Node temp;
    temp.data = data;
    temp.priority = priority;

    // Add new node to end of heap
    heap[*pn] = temp;
    ++(*pn);

    // Last node is the child node
    child = *pn - 1;
    // Its parent node
    parent = (child-1)/2;

    // Bottom up
    while ((child > 0) && temp.priority < heap[parent].priority) {
        heap[child] = heap[parent];
        child = parent;
        parent = (child-1)/2;
    }
    heap[child] = temp;
```

```
}
```

Delete the top node

```
Node pq_delete(Node *heap, int *pn) {
    Node temp = heap[0];
    heap[0] = heap[*pn-1];
    --(*pn);
    adjust(heap, *pn);
    return temp;
}

void adjust(Node *heap, int n) {
    int child, parent;
    Node key;
    parent = 0;
    key = heap[parent];

    child = 2*parent+1;

    // If left child exists
    while (child < n) {

        // Right child exists
        if (child+1 < n) {
            // If right node is greater than left
            if (heap[child+1].priority < heap[child].priority) {
                ++child;
            }
        }

        // Now child stores index of greater child
        if (key.priority > heap[child].priority) {
            heap[parent] = heap[child];
            parent = child;
            child = 2*parent+1;
        }
        else {
            break;
        }
    }
    heap[parent] = key;
}
```

Height of a Tree

```c
int height(struct tnode *t) {
    int l, r;

    // Empty tree
    if(t == NULL) {
        return -1;
    }

    // Single node tree
    if((t->left == NULL) && (t->right == NULL)) {
        return 0;
    }

    // Height of left subtree
    l = height(t->left);

    // Height of right subtree
    r = height(t->right);

    // If left subtree is taller
    if(l > r) {
        return 1 + l;
    }
    // If right subtree is taller
    else {
        return 1 + r;
    }
}
```