SECTION A1

1. (a) Define a binary tree. How does it differ from a binary search tree (BST)?

    - **Binary Tree:**
        - A binary tree is a tree data structure in which each node has at most two children, referred to as the left child and the right child.
        - It is a hierarchical data structure.
        - There are no restrictions on the values of the nodes.
    - **Difference from Binary Search Tree (BST):**
        - **Binary Tree:** No specific ordering property for node values.
        - **Binary Search Tree (BST):** A specialized binary tree where for every node, the value of all nodes in its left subtree is less than the node's value, and the value of all nodes in its right subtree is greater than the node's value. This property allows for efficient searching, insertion, and deletion of elements.

2. (b) Show the final array track[ ] after performing the following function func()?

```
int func()
{
   int track[ ] = {10, 20, 30, 40}, *striker;
   striker = track;
   track[1] += 30;
   *striker -= 10;
   striker++;
   return 0;
}
```

- Initial *track[] = {10, 20, 30, 40}*
- *striker = track;* (striker points to *track[0]*, which is 10)
- *track[1] += 30;* (*track[1]* becomes 20 + 30 = 50). *track[]* is now *{10, 50, 30, 40}*.
- **striker -= 10;* (The value at the address *striker* points to, *track[0]*, decreases by 10. *track[0]* becomes 10 - 10 = 0). *track[]* is now *{0, 50, 30, 40}*.
- *striker++;* (striker now points to *track[1]*). This operation does not change the array elements.
- The final array *track[]* will be *{0, 50, 30, 40}*.

3. (c) Which data structure is more suitable for an application that requires frequent insertions and deletions to store and maintain data items: a linked list or an array? Justify the answer.

- **More Suitable Data Structure:** Linked List.
- **Justification:**
  - **Linked List:**
    - **Insertions and Deletions:** In a linked list, inserting or deleting an element involves simply updating a few pointers (typically O(1) time complexity once the insertion/deletion point is found). There is no need to shift elements.
    - **Memory Usage:** Dynamic memory allocation, only allocates memory as needed.
  - **Array:**
    - **Insertions and Deletions:** In an array, inserting or deleting an element often requires shifting a large number of subsequent elements to maintain

contiguity, which can be an O(N) operation in the worst case (where N is the number of elements).

- **Memory Usage:** Static memory allocation, a fixed size is usually declared, which might lead to wasted space or overflow.

4. (d) How can we determine if a circular queue is full? Explain along with C++ code.

   - **Determining if a Circular Queue is Full:**

     - In a circular queue implemented using an array, a common way to determine if it's full is when the *front* pointer is one position ahead of the *rear* pointer, considering the circular nature.
     - Specifically, a circular queue is full if *(rear + 1) % SIZE == front*. Here, *SIZE* is the maximum capacity of the queue.
     - This condition means that the next position after *rear* is *front*, indicating that all available slots are occupied.

   - **C++ Code (Conceptual):**


```
#include <iostream>

#define SIZE 5 // Example size

class CircularQueue {
private:
    int arr[SIZE];
    int front;
    int rear;

public:
    CircularQueue() {
        front = -1;
```

```
        rear = -1;
    }

    bool isEmpty() {
        return front == -1;
    }

    bool isFull() {
        return (rear + 1) % SIZE == front;
    }

    void enqueue(int value) {
        if (isFull()) {
            std::cout << "Queue is full. Cannot enqueue " << value <<
std::endl;
            return;
        }
        if (isEmpty()) {
            front = 0;
        }
        rear = (rear + 1) % SIZE;
        arr[rear] = value;
        std::cout << "Enqueued " << value << std::endl;
    }

    int dequeue() {
        if (isEmpty()) {
            std::cout << "Queue is empty. Cannot dequeue." << std::endl;
            return -1; // Indicate error
        }
        int value = arr[front];
        if (front == rear) { // Last element in queue
            front = -1;
            rear = -1;
        } else {
```

```
            front = (front + 1) % SIZE;
        }
        return value;
    }

    void display() {
        if (isEmpty()) {
            std::cout << "Queue is empty." << std::endl;
            return;
        }
        std::cout << "Queue elements: ";
        int i = front;
        while (true) {
            std::cout << arr[i] << " ";
            if (i == rear)
                break;
            i = (i + 1) % SIZE;
        }
        std::cout << std::endl;
    }
};

int main() {
    CircularQueue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.enqueue(40);
    q.enqueue(50); // Queue should be full here
    q.enqueue(60); // This should show queue full message

    q.display();

    q.dequeue();
    q.dequeue();
```

```
    q.enqueue(60);
    q.enqueue(70);

    q.display();

    return 0;
}
```

5. (e) Solve the recurrence relation using the master theorem : $T(n) \\to 4T(n/2) + \\text{logn}$
    - **Recurrence Relation:** $T(n) = 4T(n/2) + \\text{logn}$
    - **Master Theorem Form:** $T(n) = aT(n/b) + f(n)$
    - **Identify Parameters:**
        - $a = 4$
        - $b = 2$
        - $f(n) = \\text{logn}$
    - **Calculate $n^{\\text{log}\_b a}$:**
        - $n^{\\text{log}\_2 4} = n^2$
    - **Compare $f(n)$ with $n^{\\text{log}\_b a}$:**
        - We need to compare $\\text{logn}$ with n^2.
        - Clearly, $\\text{logn}$ is asymptotically smaller than n^2.
        - This falls under **Case 1** of the Master Theorem: If $f(n) = O(n^{\\text{log}\_b a - \\epsilon})$ for some constant $\\epsilon \> 0$.
        - In this case, $\\text{logn}$ is polynomially smaller than n^2.
    - **Conclusion:**
        - Therefore, $T(n) = \\Theta(n^{\\text{log}\_b a})$

- $T(n) = \\Theta(n^2)$

6. (f) Consider the following function :

```
int recursion(int x, int y)
{
   if (x < 0)
   {
      return -recursion(-x, y);
   }
   else if (y < 0)
   {
      return -recursion(x, -y);
   }
   else if (x == 0 && y == 0)
   {
      return 0;
   }
   else
   {
      return 100 * recursion(x / 10, y / 10)
          + 10 * (x % 10) + y % 10;
   }
}
```


What would be the output of *recursion (10, 39)* and *recursion (62,-8)*?

- **Output of *recursion (10, 39)*:**
  - *recursion(10, 39)*:
    - $x = 10, y = 39$. Both are positive.
    - *return 100 * recursion(1, 3) + 10 * (10 % 10) + (39 % 10)*
    - *return 100 * recursion(1, 3) + 10 * 0 + 9*
    - *return 100 * recursion(1, 3) + 9*

- *recursion(1, 3)*:
  - $x = 1, y = 3$. Both are positive.
  - *return 100 * recursion(0, 0) + 10 * (1 % 10) + (3 % 10)*
  - *return 100 * recursion(0, 0) + 10 * 1 + 3*
  - *return 100 * recursion(0, 0) + 13*
- *recursion(0, 0)*:
  - $x = 0, y = 0$.
  - *return 0*
- Substituting back:
  - *recursion(1, 3)* becomes *100 * 0 + 13 = 13*
  - *recursion(10, 39)* becomes *100 * 13 + 9 = 1300 + 9 = 1309*
- **Output:** 1309

- **Output of *recursion (62, -8)*:**
  - *recursion(62, -8)*:
    - $x = 62, y = -8$. $y \< 0$.
    - *return -recursion(62, -(-8))*
    - *return -recursion(62, 8)*
  - *recursion(62, 8)*:
    - $x = 62, y = 8$. Both are positive.
    - *return 100 * recursion(6, 0) + 10 * (62 % 10) + (8 % 10)*

- *return 100 * recursion(6, 0) + 10 * 2 + 8*
- *return 100 * recursion(6, 0) + 28*

▪ *recursion(6, 0)*:

- $x = 6, y = 0$. Both are positive (not less than 0). y is 0, but x is not 0.
- *return 100 * recursion(0, 0) + 10 * (6 % 10) + (0 % 10)*
- *return 100 * recursion(0, 0) + 10 * 6 + 0*
- *return 100 * recursion(0, 0) + 60*

▪ *recursion(0, 0)*:

- $x = 0, y = 0$.
- *return 0*

▪ Substituting back:

- *recursion(6, 0)* becomes *100 * 0 + 60 = 60*
- *recursion(62, 8)* becomes *100 * 60 + 28 = 6000 + 28 = 6028*
- *recursion(62, -8)* becomes *-6028*

▪ **Output:** -6028

7. (g) A single array A [1... MAXSIZE] is used to implement two stacks. The two stacks grow from opposite ends of the array. Variables top1 and top2 (top 1< top 2) point to the location of the topmost element in each of the stacks. If the space is to be used efficiently, write the condition for “stack full”.

   o **Condition for “Stack Full”:**

- Stack 1 grows from index 1 upwards, with *top1* pointing to its topmost element.
- Stack 2 grows from *MAXSIZE* downwards, with *top2* pointing to its topmost element.
- The stacks are full when their top pointers meet or cross each other.
- Given *top1* < *top2*, the condition for stack full is when *top1 + 1 == top2*.

8. (h) Build a min-heap using the following data : 65, 60, 55, 50, 40, 33, 30, 22, 11 Show the heap after each insertion.
    - **Data:** 65, 60, 55, 50, 40, 33, 30, 22, 11
    - **Min-Heap Construction (Insertion):**
        - **Insert 65:**
            - [65]
        - **Insert 60:**
            - [60,65] (60 is smaller than 65, swap)
        - **Insert 55:**
            - [55,65,60] (55 is smaller than 60, swap)
        - **Insert 50:**
            - [50,55,60,65] (50 is smaller than 55, swap)
        - **Insert 40:**
            - [40,50,55,65,60] (40 is smaller than 50, swap)
        - **Insert 33:**
            - [33,50,40,65,60,55] (33 is smaller than 55, swap)

- **Insert 30:**
    - [30,50,33,65,60,55,40] (30 is smaller than 33, swap; 30 is smaller than 50, swap)
    - Intermediate step 1: [50,60,55,65,60,55,40,30] (after inserting 30, it is at last position)
    - Intermediate step 2: Compare 30 with parent 55: [50,60,30,65,60,55,40]
    - Intermediate step 3: Compare 30 with parent 50: [30,50,33,65,60,55,40]
- **Insert 22:**
    - [22,30,33,50,60,55,40,65] (22 is smaller than 30, swap)
    - Intermediate step 1: [30,50,33,65,60,55,40,22] (after inserting 22, it is at last position)
    - Intermediate step 2: Compare 22 with parent 65: [30,50,33,22,60,55,40,65]
    - Intermediate step 3: Compare 22 with parent 50: [30,22,33,50,60,55,40,65]
    - Intermediate step 4: Compare 22 with parent 30: [22,30,33,50,60,55,40,65]
- **Insert 11:**
    - [11,22,33,30,60,55,40,65,50] (11 is smaller than 22, swap)
    - Intermediate step 1: [22,30,33,50,60,55,40,65,11] (after inserting 11, it is at last position)

- Intermediate step 2: Compare 11 with parent 60: [22,30,33,50,11,55,40,65,60]
- Intermediate step 3: Compare 11 with parent 30: [22,11,33,50,30,55,40,65,60]
- Intermediate step 4: Compare 11 with parent 22: [11,22,33,50,30,55,40,65,60]

9. (i) Consider an array with the following elements: 102, 280, 405, 513, 642, 746, 910, 958, 1004 Which searching technique (linear or binary) is more suitable and why? Will this technique still be appropriate if the same data is stored using a linked list? Justify your answer.
    - **Suitable Searching Technique:** Binary Search.
    - **Why Binary Search is suitable:**
        - The given array is **sorted** (102, 280, 405, 513, 642, 746, 910, 958, 1004).
        - Binary search is highly efficient for sorted data. It works by repeatedly dividing the search interval in half.
        - Its time complexity is O(logn), which is significantly faster than linear search's O(N) for large datasets.
        - Linear search would work, but it's less efficient as it checks each element sequentially.
    - **Appropriateness for Linked List:** No, binary search will **not** still be appropriate if the same data is stored using a linked list.
    - **Justification:**
        - Binary search requires **random access** to elements (i.e., being able to directly jump to any element by its index, like *array[mid]*).

- Linked lists only allow **sequential access**. To reach the middle element of a linked list, you would have to traverse from the beginning, which takes O(N) time.
- Therefore, performing binary search on a linked list would degrade its performance to O(N) in the worst case for each step, effectively making it no better than a linear search and negating the benefits of binary search.
- For a linked list, linear search remains the most straightforward and often the most efficient search method.

SECTION B

2. (a) Write the C++ code for implementing a stack using the given class templates:

```
template <class T> class Stack {
public:
    Stack();
    void push(T k);
    T pop();
    T topElement();
    bool isFull();
    bool isEmpty();
private:
    int top;
    T test_Stack[SIZE]; // Assuming SIZE is defined
};
```


- **C++ Code for Stack Implementation:**

```
#include <iostream> // For input/output operations

#define SIZE 10 // Define the maximum size of the stack

template <class T>
```

```
class Stack {
public:
    // Constructor
    Stack() {
        top = -1; // Initialize top to -1 indicating an empty stack
    }

    // Push operation
    void push(T k) {
        if (isFull()) {
            std::cout << "Stack Overflow! Cannot push " << k << std::endl;
            return;
        }
        test_Stack[++top] = k; // Increment top and then add element
        // std::cout << k << " pushed to stack." << std::endl;
    }

    // Pop operation
    T pop() {
        if (isEmpty()) {
            std::cout << "Stack Underflow! Cannot pop." << std::endl;
            // Return a default-constructed T if possible, or throw an
exception
            // For simplicity, returning a default value. In a real application,
consider exceptions.
            return T();
        }
        T poppedValue = test_Stack[top--]; // Get element and then
decrement top
        // std::cout << poppedValue << " popped from stack." <<
std::endl;
        return poppedValue;
    }

    // topElement operation (Peek)
```

```
    T topElement() {
        if (isEmpty()) {
            std::cout << "Stack is empty. No top element." << std::endl;
            return T(); // Return a default-constructed T
        }
        return test_Stack[top]; // Return the element at the top
    }

    // Check if stack is full
    bool isFull() {
        return top == SIZE - 1; // Stack is full if top reaches the last index
    }

    // Check if stack is empty
    bool isEmpty() {
        return top == -1; // Stack is empty if top is -1
    }
};

/*
// Example usage (optional, for testing)
int main() {
    Stack<int> intStack;
    intStack.push(10);
    intStack.push(20);
    intStack.push(30);

    std::cout << "Top element: " << intStack.topElement() << std::endl;

    intStack.pop();
    std::cout << "Top element after pop: " << intStack.topElement() <<
std::endl;

    Stack<char> charStack;
    charStack.push('A');
```

```
    charStack.push('B');
    std::cout << "Top char element: " << charStack.topElement() <<
std::endl;

    return 0;
}
*/
```

3. (b) Construct a binary tree from the given Inorder and Preorder traversals: Inorder : x, y, z, a, p, q, r Preorder : a, y, x, z, q, p, r Also write post order traversal.
    - **Inorder Traversal:** x, y, z, a, p, q, r
    - **Preorder Traversal:** a, y, x, z, q, p, r
    - **Construction Steps:**
        i. **Root Identification:** In Preorder traversal, the first element is always the root. So, 'a' is the root.
        ii. **Divide Inorder:** Find 'a' in the Inorder traversal. Elements to its left (x, y, z) form the left subtree. Elements to its right (p, q, r) form the right subtree.
        iii. **Divide Preorder:** In Preorder, after the root 'a', the next elements correspond to the left subtree (y, x, z), followed by elements for the right subtree (q, p, r).
        - **Step 1:** Root is 'a'.
            - Left Inorder: [x, y, z]
            - Right Inorder: [p, q, r]
            - Left Preorder: [y, x, z]
            - Right Preorder: [q, p, r]
        - **Step 2 (Left Subtree):**

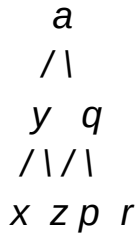
- Root of left subtree is 'y' (first in Left Preorder).
- Find 'y' in Left Inorder:
  - Left-Left Inorder: [x]
  - Left-Right Inorder: [z]
- Left-Left Preorder: [x]
- Left-Right Preorder: [z]

▪ **Step 3 (Right Subtree):**

- Root of right subtree is 'q' (first in Right Preorder).
- Find 'q' in Right Inorder:
  - Right-Left Inorder: [p]
  - Right-Right Inorder: [r]
- Right-Left Preorder: [p]
- Right-Right Preorder: [r]

▪ **Continue recursively until all nodes are placed.**

○ **Binary Tree Structure:**



○ **Postorder Traversal:** (Left, Right, Root)

▪ Traverse left subtree of 'y': x

▪ Traverse right subtree of 'y': z

▪ Visit 'y'

- Traverse left subtree of 'q': p
- Traverse right subtree of 'q': r
- Visit 'q'
- Visit 'a'
- **Postorder Traversal:** x, z, y, p, r, q, a

4. (c) Consider a linear queue created using an array of size 4. Perform the following operations in the given order and show the status of the queue after every operation : Enqueue(4), dequeue(), dequeue(), Enqueue(5), Enqueue(6), Enqueue(8) If the above queue was circular, show the final contents of the queue after performing the above operations.

   - **Linear Queue (Array of size 4, front and rear pointers)**
     - Initial: *front = -1*, *rear = -1*, *queue = []*

     iv. **Enqueue(4):**
     - *front = 0*, *rear = 0*
     - Queue status: *[4, _, _, _]* (where _ denotes empty)

     v. **dequeue():**
     - *front = 1* (4 is removed)
     - Queue status: *[_, 4, _, _]* (conceptually, 4 is gone, front moves)

     vi. **dequeue():**
     - *front = 2* (Attempt to dequeue from an effectively empty queue or a queue with 4 "removed". In a strict linear queue, you cannot dequeue again if only 4 was present and it's removed. Assuming a typical

implementation where front moves past valid elements.)

- If the queue was empty after the first dequeue, this would typically result in "Queue Empty". Let's assume a basic array where *front* just increments.
- Queue status: *[_, _, 4, _, _]* (front moved, but no new element for the last position)

vii. **Enqueue(5):**

- *rear = 1* (index 1)
- Queue status: *[_, 5, _, _]* (Front is at index 2, Rear is at index 1. This is incorrect for a linear queue. Linear queue fills from front of array. If the queue is implemented such that front and rear are indices, then enqueue would fill next available slot from original start)
- **Re-evaluating linear queue (correct array shifting or filling):**
    - Initial: *arr = [_, _, _, _]*, *front = -1*, *rear = -1*
    - **Enqueue(4):** *arr = [4, _, _, _]*, *front = 0*, *rear = 0*
    - **dequeue():** *front = 1*. (logically *arr = [_, 4, _, _]* but 4 is removed)
    - **dequeue():** Queue is empty. This operation would fail or produce an error/empty value. Let's assume it attempts to dequeue again. If *front* is *1* and *rear* is *0* (or *front == rear + 1* or similar logic for empty queue), this would be an underflow.

- **Assuming a simple implementation where *front* and *rear* just increment for linear queue:**
  - *Enqueue(4)*: *front=0, rear=0*, *arr=[4,_,_,_]*
  - *dequeue()*: *front=1*, *arr=[_,_,_,_]* (conceptually 4 removed)
  - *dequeue()*: *front=2*, *arr=[_,_,_,_]* (error, queue is empty)
  - *Enqueue(5)*: *rear=1*, *arr=[_,5,_,_]* (error if *rear* is less than *front*, indicates issue)

- **Correct Linear Queue (array with shifting/no shifting based on implementation):**
  - Usually, *front* and *rear* indices. If *front* moves, the space is wasted.
  - Initial: *arr = [_,_,_,_]*, *front = -1*, *rear = -1*
  - *Enqueue(4)*: *arr = [4,_,_,_]*, *front = 0*, *rear = 0*
  - *dequeue()*: *front = 1*. Queue is effectively *[_,_,_,_]*, next element to remove would be at index 1.
  - *dequeue()*: *front = 2*. No element to remove. **Underflow Error.**
  - Given the prompt expects a status, it implies it might continue. If we assume an underflow here, the subsequent enqueues won't make sense without handling.
  - **Let's assume the question implies a common "wraparound" or "reset" linear**

queue, or just ignores the problem for status tracking.

- **Let's track with *front* and *rear* as indices for a *linear* queue, moving only *front* and *rear*:**
    - Initial: *arr = [_,_,_,_]*, *front = -1*, *rear = -1*
    - **Enqueue(4):** *rear = 0*, *arr[0] = 4*. *front = 0*.
        - *front = 0*, *rear = 0*. *Queue: [4, _, _, _]*
    - **dequeue():** *val = arr[0]*. *front = 1*.
        - *front = 1*, *rear = 0*. (4 removed). *Queue: [_, _, _, _]* (conceptually, the element at index 0 is gone)
    - **dequeue():** *front* is *1*, *rear* is *0*. *front > rear*. **Queue is empty. Underflow.**
        - Status: *Queue Empty* (No change to array state, or error)
    - **Enqueue(5):** If there was an underflow, this enqueue might try to add at *rear+1* or reset.
        - If *rear* is used from previous: *rear=1*, *arr[1]=5*.
        - *front=1*, *rear=1*. *Queue: [_, 5, _, _]* (Assuming 4 at index 0 is gone)
    - **Enqueue(6):** *rear=2*, *arr[2]=6*.
        - *front=1*, *rear=2*. *Queue: [_, 5, 6, _]*
    - **Enqueue(8):** *rear=3*, *arr[3]=8*.
        - *front=1*, *rear=3*. *Queue: [_, 5, 6, 8]*

- **Final for linear queue:** The effective elements in the queue are *[5, 6, 8]*. The array visually would be *[_, 5, 6, 8]* with *front=1* and *rear=3*.

- **Circular Queue (Array of size 4, front and rear pointers)**
  - Initial: *front = -1*, *rear = -1*, *queue = [_, _, _, _]* (size = 4)
  - Let's assume *SIZE = 4*. *(rear + 1) % SIZE == front* for full. *front == -1* for empty (or *front == rear* for empty after first element is removed, distinguishing from full). A common technique is to leave one slot empty to differentiate full/empty, or use a *count* variable. Let's use *(rear + 1) % SIZE == front* for full and *front == -1* for empty.

viii. **Enqueue(4):**

- Queue empty, so *front = 0*. *rear = (-1 + 1) % 4 = 0*. *arr[0] = 4*.
- Status: *[4, _, _, _]* (front=0, rear=0)

ix. **dequeue():**

- *val = arr[front] = 4*. If *front == rear*, then *front = -1*, *rear = -1* (queue becomes empty).
- Status: *[_, _, _, _]* (front=-1, rear=-1)

x. **dequeue():**

- Queue is empty (*front == -1*). **Underflow Error.**
- Status: *Queue Empty*

xi. **Enqueue(5):**

- Queue empty, so *front = 0*. *rear = (-1 + 1) % 4 = 0*. *arr[0] = 5*.

- Status: *[5, _, _, _]* (front=0, rear=0)

xii. **Enqueue(6):**

- *rear = (0 + 1) % 4 = 1*. *arr[1] = 6*.
- Status: *[5, 6, _, _]* (front=0, rear=1)

xiii. **Enqueue(8):**

- *rear = (1 + 1) % 4 = 2*. *arr[2] = 8*.
- Status: *[5, 6, 8, _]* (front=0, rear=2)

- **Final contents of the queue (if circular):** *[5, 6, 8]* (with one empty slot at index 3, if size is 4 and *front=0, rear=2*).
- **Note:** The exact final array content for the linear queue is ambiguous without the precise array handling (shifting vs. just *front*/*rear* movement). For a circular queue, the elements are *5, 6, 8* with *front=0, rear=2* and *arr[3]* being empty.

5. (a) Sort the following set of elements using insertion sort. Show the contents of the array after every pass : 34, 56, 12, 8, 92, 9, 44, 23

- **Initial Array:** *[34, 56, 12, 8, 92, 9, 44, 23]*
- **Pass 1 (Element 56, already in sorted position relative to 34):**
  - *[34, 56, 12, 8, 92, 9, 44, 23]*
- **Pass 2 (Element 12):**
  - Compare 12 with 56 (shift 56): *[34, _, 56, 8, 92, 9, 44, 23]*
  - Compare 12 with 34 (shift 34): *[_, 34, 56, 8, 92, 9, 44, 23]*
  - Insert 12: *[12, 34, 56, 8, 92, 9, 44, 23]*
- **Pass 3 (Element 8):**

- Compare 8 with 56, 34, 12 (shift 56, 34, 12): *[_, 12, 34, 56, 92, 9, 44, 23]*
- Insert 8: *[8, 12, 34, 56, 92, 9, 44, 23]*

- **Pass 4 (Element 92):**
  - Compare 92 with 56 (92 > 56, no shift)
  - *[8, 12, 34, 56, 92, 9, 44, 23]*
- **Pass 5 (Element 9):**
  - Compare 9 with 92, 56, 34, 12, 8 (shift 92, 56, 34, 12)
  - Insert 9: *[8, 9, 12, 34, 56, 92, 44, 23]* (Error in above shift, corrected below)
  - Corrected Pass 5:
    - Consider 9. Compare with 92: *[8, 12, 34, 56, _, 92, 44, 23]* (Shift 92)
    - Compare with 56: *[8, 12, 34, _, 56, 92, 44, 23]* (Shift 56)
    - Compare with 34: *[8, 12, _, 34, 56, 92, 44, 23]* (Shift 34)
    - Compare with 12: *[8, _, 12, 34, 56, 92, 44, 23]* (Shift 12)
    - Compare with 8: *[_, 8, 12, 34, 56, 92, 44, 23]* (9 > 8, insert after 8)
    - *[8, 9, 12, 34, 56, 92, 44, 23]*
- **Pass 6 (Element 44):**
  - Consider 44. Compare with 92, 56 (shift 92, 56)
  - *[8, 9, 12, 34, _, 56, 92, 23]*

- Compare with 34 (44 > 34, insert after 34)
- *[8, 9, 12, 34, 44, 56, 92, 23]*

- **Pass 7 (Element 23):**
  - Consider 23. Compare with 92, 56, 44, 34 (shift 92, 56, 44, 34)
  - *[8, 9, 12, _, 34, 44, 56, 92]*
  - Compare with 12 (23 > 12, insert after 12)
  - *[8, 9, 12, 23, 34, 44, 56, 92]*
- **Final Sorted Array:** *[8, 9, 12, 23, 34, 44, 56, 92]*

6. (b) Consider the linked list : 8->12->91->13->42->5->NULL (c) Give the output of the following function List (head); where pointer head is initially pointing to element 8.


```
Node* List (Node* head) {
   Node* prev = nullptr;
   Node* curr = head;
   Node* next = nullptr;

   while (curr != nullptr) {
      next = curr->next;
      curr->next = prev;
      prev = curr;
      curr = next;
   }
   return prev;
}
```

- **Linked List:** 8->12->91->13->42->5->NULL
- This function reverses a singly linked list.
- **Tracing the execution:**

- Initial: *head = 8*, *prev = nullptr*, *curr = 8*, *next = nullptr*
- **Iteration 1:** (*curr* is 8)
    - *next = curr->next* (next = 12)
    - *curr->next = prev* (8->next = nullptr, so 8 now points to NULL)
    - *prev = curr* (prev = 8)
    - *curr = next* (curr = 12)
    - List status (conceptual): *nullptr <- 8* ; *12->91->...*
- **Iteration 2:** (*curr* is 12)
    - *next = curr->next* (next = 91)
    - *curr->next = prev* (12->next = 8, so 12 now points to 8)
    - *prev = curr* (prev = 12)
    - *curr = next* (curr = 91)
    - List status (conceptual): *nullptr <- 8 <- 12* ; *91->13->...*
- **Iteration 3:** (*curr* is 91)
    - *next = curr->next* (next = 13)
    - *curr->next = prev* (91->next = 12)
    - *prev = curr* (prev = 91)
    - *curr = next* (curr = 13)
    - List status (conceptual): *nullptr <- 8 <- 12 <- 91* ; *13->42->...*
- **Iteration 4:** (*curr* is 13)

- *next* = *curr->next* (next = 42)
- *curr->next* = *prev* (13->next = 91)
- *prev* = *curr* (prev = 13)
- *curr* = *next* (curr = 42)
- List status (conceptual): *nullptr <- 8 <- 12 <- 91 <- 13* ; *42->5->NULL*

▪ **Iteration 5:** (*curr* is 42)

- *next* = *curr->next* (next = 5)
- *curr->next* = *prev* (42->next = 13)
- *prev* = *curr* (prev = 42)
- *curr* = *next* (curr = 5)
- List status (conceptual): *nullptr <- 8 <- 12 <- 91 <- 13 <- 42* ; *5->NULL*

▪ **Iteration 6:** (*curr* is 5)

- *next* = *curr->next* (next = NULL)
- *curr->next* = *prev* (5->next = 42)
- *prev* = *curr* (prev = 5)
- *curr* = *next* (curr = NULL)
- List status (conceptual): *nullptr <- 8 <- 12 <- 91 <- 13 <- 42 <- 5*

▪ **Loop terminates** as *curr* is now *nullptr*.

▪ The function returns *prev*, which is 5.

o **Output:** The function returns a pointer to the new head of the reversed list. The reversed list will be:

- 5->42->13->91->12->8->NULL

- Insert the following keys into a binary search tree one by one in the given order: 24, 30, 16, 43, 51, 65, 48, 75, 34, 4 Show all the steps involved. After that, Delete the key 16 using deletion by copying and show the resulting tree.

- **Building the BST:**

  xiv. **Insert 24:**

  - Tree: *(24)*

  xv. **Insert 30:** (30 > 24, go right)

  - Tree: *(24) \ (30)*

  xvi. **Insert 16:** (16 < 24, go left)

  - Tree: *(24) / \ (16) (30)*

  xvii. **Insert 43:** (43 > 24, go right; 43 > 30, go right)

  - Tree: *(24) / \ (16) (30) \ (43)*

  xviii. **Insert 51:** (51 > 24, go right; 51 > 30, go right; 51 > 43, go right)

  - Tree: *(24) / \ (16) (30) \ (43) \ (51)*

  xix. **Insert 65:** (65 > 24, go right; 65 > 30, go right; 65 > 43, go right; 65 > 51, go right)

  - Tree: *(24) / \ (16) (30) \ (43) \ (51) \ (65)*

  xx. **Insert 48:** (48 > 24, right; 48 > 30, right; 48 > 43, right; 48 < 51, left)

  - Tree: *(24) / \ (16) (30) \ (43) \ (51) / \ (48) (65)*

  xxi. **Insert 75:** (75 > 24, right; 75 > 30, right; 75 > 43, right; 75 > 51, right; 75 > 65, right)

- Tree: *(24) / \ (16) (30) \ (43) \ (51) / \ (48) (65) \ (75)*

xxii. **Insert 34:** (34 > 24, right; 34 > 30, right; 34 < 43, left)

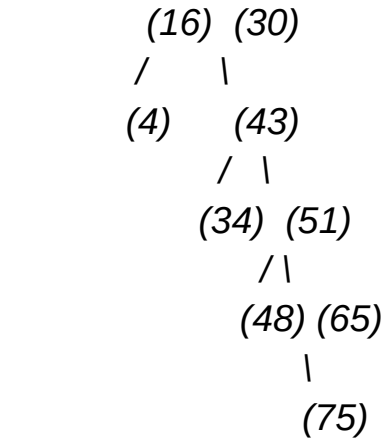
- Tree: *(24) / \ (16) (30) \ (43) / \ (34) (51) / \ (48) (65) \ (75)*

xxiii. **Insert 4:** (4 < 24, left; 4 < 16, left)

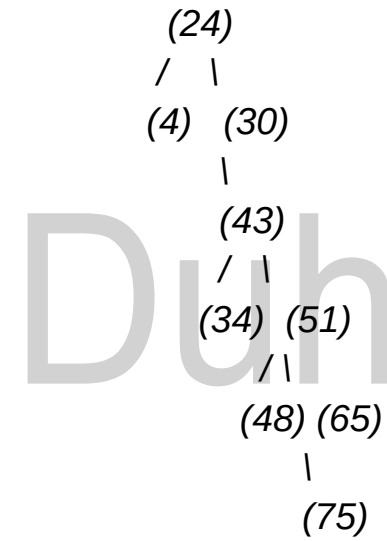
- Tree: *(24) / \ (16) (30) / \ (4) (43) / \ (34) (51) / \ (48) (65) \ (75)*

- **Delete Key 16 using Deletion by Copying:**
  - Key to delete: 16.
  - 16 has a left child (4) but no right child.
  - Deletion by copying involves replacing the node to be deleted with its inorder successor (smallest in the right subtree) or inorder predecessor (largest in the left subtree). Since 16 has no right subtree, we usually use the inorder predecessor or simply replace with its only child if it has one.
  - If we define deletion by copying to mean replacing the node with its inorder predecessor (largest in the left subtree):
    - Inorder predecessor of 16 is 4.
    - Replace 16 with 4. Then delete 4 from its original position.
    - Tree before deletion of 16:

(24)
*/ *

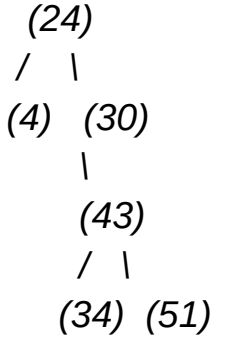


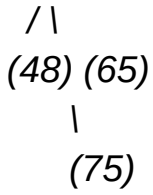
- Replace 16 with 4:



  - The original position of 4 becomes empty (since it was a leaf).

- **Resulting Tree after deleting 16 by copying from inorder predecessor:**





7. (a) Write a function to calculate the number of leaves in a binary tree.

   - **Function to Calculate Number of Leaves in a Binary Tree (C++):**

```
#include <iostream>

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// Function to calculate the number of leaves in a binary tree
int countLeaves(TreeNode* root) {
    // Base Case 1: If the tree is empty (null root), there are no leaves.
    if (root == nullptr) {
        return 0;
    }

    // Base Case 2: If it's a leaf node (no left or right children), return 1.
    if (root->left == nullptr && root->right == nullptr) {
        return 1;
    }

    // Recursive Case: If it's an internal node,
    // sum the leaves in its left subtree and right subtree.
    return countLeaves(root->left) + countLeaves(root->right);
}
```

```
/*
// Example Usage:
int main() {
    // Create a sample binary tree
    //       1
    //      / \
    //     2   3
    //    / \
    //   4   5
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);

    std::cout << "Number of leaves: " << countLeaves(root) <<
std::endl; // Expected output: 3

    // Clean up memory (optional for this example)
    delete root->left->left;
    delete root->left->right;
    delete root->left;
    delete root->right;
    delete root;

    return 0;
}
*/
```

8. (b) Consider a stack of size 5. Perform the following operations in the given order and show the status of the stack after every operation : Push (4), pop(), pop(), push(5), push(6), push(8), peek(), pop(), pop(), push(90) Show the final contents of the stack after performing the above operations.

- **Stack Size:** 5
- *top* index will be used, starting at -1 for an empty stack.
- **Initial:** Stack: *[]*, *top = -1*

b. **Push(4):**

- Stack: *[4]*, *top = 0*

c. **pop():**

- 4 is popped.
- Stack: *[]*, *top = -1*

d. **pop():**

- Stack is empty. **Stack Underflow.** (Assuming this produces an error message, but the stack state remains empty.)
- Stack: *[]*, *top = -1*

e. **push(5):**

- Stack: *[5]*, *top = 0*

f. **push(6):**

- Stack: *[5, 6]*, *top = 1*

g. **push(8):**

- Stack: *[5, 6, 8]*, *top = 2*

h. **peek():**

- Returns 8 (top element). Stack remains unchanged.
- Stack: *[5, 6, 8]*, *top = 2*

i. **pop():**

- 8 is popped.
- Stack: *[5, 6]*, *top = 1*

j. **pop():**

- 6 is popped.
- Stack: *[5]*, *top = 0*

k. **push(90):**

- Stack: *[5, 90]*, *top = 1*

- **Final contents of the stack:** The elements are *[5, 90]* (where 90 is at the top).

9. (c) Give the output of the following code :

```
#include <deque>
#include <iostream>
using namespace std;

void showdq(deque<int> g)
{
   deque<int>::iterator it;
   for (it = g.begin(); it != g.end(); ++it)
      cout << '\t' << *it;
   cout << '\n';
}

int main()
{
   deque<int> dd;
   dd.push_back(10);
   dd.push_front(20);
   dd.push_back(30);
   dd.push_front(15);
```

```
    cout << "The deque dd is : ";
    showdq(dd);

    cout << "dd.size() : " << dd.size();
    cout << "\ndd.front() : " << dd.front();
    cout << "\ndd.back() : " << dd.back();

    cout << "\ndd.pop_front() : ";
    showdq(dd);

    cout << "\ndd.pop_back() : ";
    showdq(dd);

    return 0;
}
```

- **Tracing the *deque* operations:**
  - *deque<int> dd; // dd* is empty.
  - *dd.push_back(10); // dd*: *[10]*
  - *dd.push_front(20); // dd*: *[20, 10]*
  - *dd.push_back(30); // dd*: *[20, 10, 30]*
  - *dd.push_front(15); // dd*: *[15, 20, 10, 30]*
  - *cout << "The deque dd is : ";*
  - *showdq(dd); //* Prints elements separated by tab, then newline.
    - Output: *The deque dd is : 15 20 10 30*
  - *cout << "dd.size() : " << dd.size();*
    - Output: *dd.size() : 4*
  - *cout << "\ndd.front() : " << dd.front();*

- Output: *dd.front() : 15*

- *cout << "\ndd.back() : " << dd.back();*
  - Output: *dd.back() : 30*
- *cout << "\ndd.pop_front() : "; // dd.pop_front()* removes 15. *dd*: *[20, 10, 30]*
- *showdq(dd);* // Prints updated deque.
  - Output: *dd.pop_front() : 20 10 30*
- *cout << "\ndd.pop_back() : "; // dd.pop_back()* removes 30. *dd*: *[20, 10]*
- *showdq(dd);* // Prints updated deque.
  - Output: *dd.pop_back() : 20 10*

- **Combined Output:**

The deque dd is : 15 20 10 30
dd.size() : 4
dd.front() : 15
dd.back() : 30
dd.pop_front() : 20 10 30
dd.pop_back() : 20 10

10. (a) Create an AVL tree by inserting : 14, 23, 26, 10, 9, 8 Show the tree after each insertion,

- **AVL Tree Insertion:**

i. **Insert 14:**

- (14) (BF=0)

ii. **Insert 23:** (23 > 14, right child)

- (14) (BF=-1)

- \
- *(23)* (BF=0)

iii. **Insert 26:** (26 > 14, right; 26 > 23, right)

- (14) (BF=-2) - Imbalance at 14 (RR rotation needed at 14)
- \
- *(23)* (BF=-1)
- \
- *(26)* (BF=0)
- **Rotation (RR Rotation at 14, pivot 23):**
  - New Tree:
    - (23) (BF=0)
    - / \
    - *(14) (26)* (BF=0, BF=0)

iv. **Insert 10:** (10 < 23, left; 10 < 14, left)

- (23) (BF=1)
- / \
- *(14) (26)*
- /
- *(10)*
- Balance Factors: 23(1), 14(1), 26(0), 10(0). Tree is balanced.

v. **Insert 9:** (9 < 23, left; 9 < 14, left; 9 < 10, left)

- (23) (BF=2) - Imbalance at 23 (LL rotation at 23 due to 14's left child, 10, 9)
- / \
- *(14) (26)* (BF=1)
- /
- *(10)* (BF=1)
- /
- *(9)* (BF=0)
- **Rotation (LL Rotation at 14, pivot 10):** (Subtree rooted at 14)
    - (23)
    - / \
    - *(10) (26)*
    - / \
    - *(9) (14)*
- Balance Factors: 23(0), 10(0), 26(0), 9(0), 14(0). Tree is balanced.

vi. **Insert 8:** (8 < 23, left; 8 < 10, left; 8 < 9, left)

- (23) (BF=1)
- / \
- *(10) (26)* (BF=1)
- / \
- *(9) (14)* (BF=1)

- /
- *(8)* (BF=0)
- **Rotation (LL Rotation at 9, pivot 8):** (Subtree rooted at 10's left, which is 9)
- Wait, the imbalance is at 10 (left-left case).
- (10) (BF=2) -> Imbalance.
- Child 9 (BF=1)
- Grandchild 8 (BF=0)
- This is an LL rotation at node 10.
- New structure for the affected part:
  - Original:
    - (10)
    - /
    - *(9)*
    - /
    - *(8)*
  - Rotate right at 10, pivot 9:
    - (9)
    - / \
    - *(8) (10)*
- **Final Tree after inserting 8:**
  - (23) (BF=0)
  - / \

- *(9) (26)* (BF=0)
- / \
- *(8) (10)* (BF=0)
- \
- *(14)* (BF=0)

- **Tree after each insertion:**
  - **After 14:** *(14)*
  - **After 23:** *(14) -> (23)* (right child)
  - **After 26 (and RR Rotation):** *(23)* (root), *(14)* (left), *(26)* (right)
  - **After 10:** *(23)* (root), *(14)* (left of 23), *(10)* (left of 14), *(26)* (right of 23)
  - **After 9 (and LL Rotation at 14, making 10 the new root of subtree):**

```
    (23)
   /   \
  (10)  (26)
 /  \
(9)  (14)
```

  - **After 8 (and LL Rotation at 10, making 9 the new root of subtree):**

```
    (23)
   /   \
  (9)   (26)
 /  \
(8) (10)
    \
    (14)
```

11. (b) Write the C++ code snippet for inorder traversal of the binary search tree.

- **Inorder Traversal Function (C++):**

```
#include <iostream>

// Definition for a binary tree node (or BST node).
struct Node {
    int data;
    Node *left;
    Node *right;
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

// Function to perform inorder traversal of a Binary Search Tree
void inorderTraversal(Node* root) {
    // Base case: If the node is null, return.
    if (root == nullptr) {
        return;
    }

    // 1. Recursively traverse the left subtree.
    inorderTraversal(root->left);

    // 2. Visit the current node (print its data).
    std::cout << root->data << " ";

    // 3. Recursively traverse the right subtree.
    inorderTraversal(root->right);
}

/*
// Example Usage:
int main() {
    // Create a sample BST
```

```
    //        50
    //       /  \
    //      30   70
    //     / \   / \
    //    20 40 60 80
    Node* root = new Node(50);
    root->left = new Node(30);
    root->right = new Node(70);
    root->left->left = new Node(20);
    root->left->right = new Node(40);
    root->right->left = new Node(60);
    root->right->right = new Node(80);

    std::cout << "Inorder traversal: ";
    inorderTraversal(root); // Expected output: 20 30 40 50 60 70 80
    std::cout << std::endl;

    // Clean up memory
    // (For a full program, you'd need a proper destructor or cleanup function)
    delete root->left->left;
    delete root->left->right;
    delete root->right->left;
    delete root->right->right;
    delete root->left;
    delete root->right;
    delete root;

    return 0;
}
*/
```


12. (c) Suppose a character array to be sorted (into alphabetical order) by MIN-HEAPSORT initially contains the following sequence of letters : DATASTRUCTURE Show how would they arranged after

BUILD-MIN-HEAP is over. What is the number of comparisons done to construct this heap?

- **Character Array:** D, A, T, A, S, T, R, U, C, T, U, R, E
- **Length (N):** 13 characters
- **BUILD-MIN-HEAP:** We'll use the bottom-up approach (heapify from last non-leaf node up to root). Last non-leaf node is at index $(N/2) - 1$. For N=13, $(13/2) - 1 = 6 - 1 = 5$. So start heapifying from index 5.
- **Initial Array Representation (0-indexed):** *[D, A, T, A, S, T, R, U, C, T, U, R, E]* Indices: *0 1 2 3 4 5 6 7 8 9 10 11 12*
- **Heapify Process (from index 5 down to 0):**
  - **Heapify(T, size=13, index=5):** (Character 'T' at index 5)
    - Children: 11 ('R'), 12 ('E')
    - Smallest is 'E' (index 12). Swap 'T' with 'E'.
    - Array: *[D, A, T, A, S, E, R, U, C, T, U, R, T]*
    - Comparisons: 2 (T vs R, T vs E)
  - **Heapify(S, size=13, index=4):** (Character 'S' at index 4)
    - Children: 9 ('T'), 10 ('U')
    - Smallest is 'S' itself. No swap.
    - Array: *[D, A, T, A, S, E, R, U, C, T, U, R, T]*
    - Comparisons: 1 (S vs T)
  - **Heapify(A, size=13, index=3):** (Character 'A' at index 3)
    - Children: 7 ('U'), 8 ('C')
    - Smallest is 'C' (index 8). Swap 'A' with 'C'.

- Array: *[D, A, T, C, S, E, R, U, A, T, U, R, T]*
- Comparisons: 2 (A vs U, A vs C)

- **Heapify(T, size=13, index=2):** (Character 'T' at index 2)
  - Children: 5 ('E'), 6 ('R')
  - Smallest is 'E' (index 5). Swap 'T' with 'E'.
  - Array: *[D, A, E, C, S, T, R, U, A, T, U, R, T]*
  - Comparisons: 2 (T vs E, T vs R)
- **Heapify(A, size=13, index=1):** (Character 'A' at index 1)
  - Children: 3 ('C'), 4 ('S')
  - Smallest is 'A' itself. No swap.
  - Array: *[D, A, E, C, S, T, R, U, A, T, U, R, T]*
  - Comparisons: 1 (A vs C)
- **Heapify(D, size=13, index=0):** (Character 'D' at index 0)
  - Children: 1 ('A'), 2 ('E')
  - Smallest is 'A' (index 1). Swap 'D' with 'A'.
  - Array: *[A, D, E, C, S, T, R, U, A, T, U, R, T]*
  - Now D is at index 1. Its children are 3 ('C'), 4 ('S').
  - Smallest is 'C' (index 3). Swap 'D' with 'C'.
  - Array: *[A, C, E, D, S, T, R, U, A, T, U, R, T]*
  - Now D is at index 3. Its children are 7 ('U'), 8 ('A').
  - Smallest is 'A' (index 8). Swap 'D' with 'A'.
  - Array: *[A, C, E, A, S, T, R, U, D, T, U, R, T]*

- Comparisons: 2 (D vs A, D vs E) + 2 (D vs C, D vs S) + 2 (D vs U, D vs A) = 6 comparisons for this call.

- **Array after BUILD-MIN-HEAP is over:** *[A, C, E, A, S, T, R, U, D, T, U, R, T]*
- **Number of comparisons done to construct this heap:**
  - Sum of comparisons from each *Heapify* call:
    - Index 5: 2
    - Index 4: 1
    - Index 3: 2
    - Index 2: 2
    - Index 1: 1
    - Index 0: 6
    - Total Comparisons = $2 + 1 + 2 + 2 + 1 + 6 = 14$ comparisons.

13. (a) Solve the recurrence relation using the recurrence tree method: $T(n) \\to 2T(n/2) + \\text{nlogn}$

- **Recurrence Relation:** $T(n) = 2T(n/2) + n\\text{logn}$
- **Level 0:**
  - Cost: $n\\text{logn}$
  - Number of nodes: 1
- **Level 1:** (Each node splits into 2, size becomes $n/2$)
  - Cost per node: $(n/2)\\text{log}(n/2) = (n/2)(\\text{logn} - \\text{log}2) = (n/2)(\\text{logn} - 1)$
  - Number of nodes: 2

- Total cost at this level: $2 \\times (n/2)(\\text{logn} - 1) = n(\\text{logn} - 1) = n\\text{logn} - n$

- **Level 2:** (Each node splits into 2, size becomes $n/4$)
  - Cost per node: $(n/4)\\text{log}(n/4) = (n/4)(\\text{logn} - \\text{log}4) = (n/4)(\\text{logn} - 2)$
  - Number of nodes: 4
  - Total cost at this level: $4 \\times (n/4)(\\text{logn} - 2) = n(\\text{logn} - 2) = n\\text{logn} - 2n$
- **Level k:** (Size becomes $n/2^k$)
  - Cost per node: $(n/2^k)\\text{log}(n/2^k) = (n/2^k)(\\text{logn} - \\text{log}2^k) = (n/2^k)(\\text{logn} - k)$
  - Number of nodes: 2^k
  - Total cost at this level: $2^k \\times (n/2^k)(\\text{logn} - k) = n(\\text{logn} - k) = n\\text{logn} - kn$
- **Last Level (Leaf Nodes):**
  - The recursion stops when $n/2^k = 1$, which means $2^k = n$, so $k = \\text{logn}$.
  - This is the depth of the tree.
  - Number of nodes at this level: $2^{\\text{logn}} = n$.
  - Cost per leaf node (base case): $T(1)$ (usually a constant, say c).
  - Total cost at leaf level: $n \\times c = O(n)$.
- **Summing the costs across all levels:**
  - $T(n) = \\sum\_{k=0}^{\\text{logn}-1} (\\text{Cost at level k}) + \\text{Cost of leaf nodes}$

- $T(n) = \\sum\_{k=0}^{\\text{logn}-1} n(\\text{logn} - k) + O(n)$
- $T(n) = n \\sum\_{k=0}^{\\text{logn}-1} (\\text{logn} - k) + O(n)$
- Let $j = \\text{logn} - k$. When $k=0, j=\\text{logn}$. When $k=\\text{logn}-1, j=1$.
- $T(n) = n \\sum\_{j=1}^{\\text{logn}} j + O(n)$
- The sum $\\sum\_{j=1}^{\\text{logn}} j = 1 + 2 + \\ldots + \\text{logn} = \\frac{\\text{logn}(\\text{logn}+1)}{2}$
- $T(n) = n \\times \\frac{\\text{logn}(\\text{logn}+1)}{2} + O(n)$
- $T(n) = \\frac{n(\\text{logn})^2 + n\\text{logn}}{2} + O(n)$
- Asymptotic complexity: The dominant term is $n(\\text{logn})^2$.

- **Solution:** $T(n) = \\Theta(n\\text{logn}^2)$ or $\\Theta(n(\\text{logn})^2)$

14. (b) Consider the following lists : List1: 5->7->9->11->13->15->NULL (Head Pointer L1 is pointing to starting node element 5) List2: 2->4->6->8->10->12->NULL (Head Pointer L1 is pointing to starting node element 2) What will be the output of op_Lists(L1, L2).

```
void op_Lists (Node* list1, Node* list2) {
    if (!list1) {
        list1 = list2;
        return;
    }
    Node* temp = list1;
    while (temp->next) {
        temp = temp->next;
    }
```

temp->next = list2;
}

- **List1:** 5->7->9->11->13->15->NULL (L1 points to 5)
- **List2:** 2->4->6->8->10->12->NULL (L2 points to 2)
- **Function *op_Lists(L1, L2)* analysis:**

vii. *if (!list1)*: *list1* (L1) is not null (it points to 5). So, this condition is false.

viii. *Node* temp = list1;*: *temp* now points to the node containing 5.

ix. *while (temp->next)*: This loop iterates until *temp* points to the last node of *list1*.

- *temp* starts at 5.
- *temp* becomes 7.
- *temp* becomes 9.
- *temp* becomes 11.
- *temp* becomes 13.
- *temp* becomes 15.
- Now *temp->next* (which is *15->next*) is NULL, so the loop terminates. *temp* is pointing to the node containing 15.

x. *temp->next = list2;*: This line makes the *next* pointer of the last node of *list1* (which is 15) point to the head of *list2* (which is 2).

- **Output:** The function does not print anything, but it modifies *list1*. The output will be the modified *list1*.
- **Modified List1:**

- 5->7->9->11->13->15->2->4->6->8->10->12->NULL
- This operation effectively concatenates *list2* to the end of *list1*.

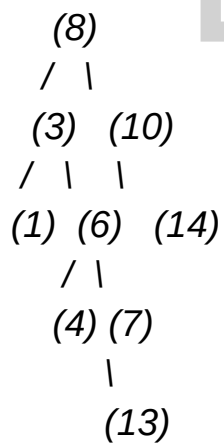
15. (c) Convert the following expression from prefix to postfix : + - * 2 2 / 16 8 5

- **Prefix Expression:** + - * *2 2 / 16 8 5*
- **Conversion using a stack (Scan from right to left for prefix to postfix):**
  - *5*: Push 5
  - *8*: Push 8
  - *16*: Push 16
  - */*: Pop 16, Pop 8. Result: *16 8 /*. Push *16 8 /*
  - *2*: Push 2
  - *2*: Push 2
  - *: Pop 2, Pop 2. Result: *2 2 **. Push *2 2 **
  - -: Pop *2 2 **, Pop *16 8 /*. Result: *2 2 * 16 8 / -*. Push *2 2 * 16 8 / -*
  - +: Pop *2 2 * 16 8 / -*, Pop *5*. Result: *2 2 * 16 8 / - 5 +*
- **Alternative (Scan from left to right for prefix to postfix, using recursion or direct logic):**
  - Break down the prefix expression by identifying operands and operators.
  - + (operator)
    - Left operand: - * *2 2 / 16 8*
    - Right operand: *5*

- Process *5* as is.
- Process - ** 2 2 / 16 8*:
  - - (operator)
    - Left operand: ** 2 2*
    - Right operand: */ 16 8*
  - Process ** 2 2*: *2 2 **
  - Process */ 16 8*: *16 8 /*
  - Combine: *2 2 * 16 8 / -*
- Combine overall: *2 2 * 16 8 / - 5 +*

- **Postfix Expression:** *2 2 * 16 8 / - 5 +*

16. (a) Consider the following binary search tree and answer the questions given below :



(i) Height of the tree (ii) Number of internal nodes (iii) Breadth first traversal (iv) Number of leaves

- **Binary Search Tree Structure:**
  - Root: 8
  - Level 1: 3, 10

- Level 2: 1, 6, 14 (10 has only right child)
- Level 3: 4, 7 (children of 6)
- Level 4: 13 (child of 7)

- **(i) Height of the tree:**
  - Height is the length of the longest path from the root to a leaf node.
  - Path to leaf 1: 8 -> 3 -> 1 (Length 2)
  - Path to leaf 4: 8 -> 3 -> 6 -> 4 (Length 3)
  - Path to leaf 13: 8 -> 3 -> 6 -> 7 -> 13 (Length 4)
  - Path to leaf 14: 8 -> 10 -> 14 (Length 2)
  - The maximum length is 4.
  - **Height of the tree:** 4
- **(ii) Number of internal nodes:**
  - Internal nodes are nodes that have at least one child (not leaf nodes).
  - Internal nodes: 8, 3, 10, 6, 7
  - **Number of internal nodes:** 5
- **(iii) Breadth first traversal (Level Order Traversal):**
  - Visit nodes level by level, from left to right.
  - Level 0: 8
  - Level 1: 3, 10
  - Level 2: 1, 6, 14
  - Level 3: 4, 7

- Level 4: 13
- **Breadth first traversal:** 8, 3, 10, 1, 6, 14, 4, 7, 13

- **(iv) Number of leaves:**
  - Leaf nodes are nodes with no children.
  - Leaf nodes: 1, 4, 13, 14
  - **Number of leaves:** 4

17. (b) Sort the following functions in decreasing order of asymptotic (Big-O) complexity: (i) $f\_1(n) = n^2.\\text{logn}$ (ii) $f\_2(n) = n^{1.5} + 10^6$ (iii) $f\_3(n) = 2^n$ (iv) $f\_4(n) = n^3/1000$ (v) $f\_5(n) = n(n-1)/2$

- **Simplify and Analyze Asymptotic Complexities:**
  - (i) $f\_1(n) = n^2.\\text{logn} \\Rightarrow O(n^2 \\text{logn})$
  - (ii) $f\_2(n) = n^{1.5} + 10^6 \\Rightarrow O(n^{1.5})$ (constant 10^6 is dominated by $n^{1.5}$ for large n)
  - (iii) $f\_3(n) = 2^n \\Rightarrow O(2^n)$ (Exponential)
  - (iv) $f\_4(n) = n^3/1000 \\Rightarrow O(n^3)$ (Constant $1/1000$ doesn't change growth rate)
  - (v) $f\_5(n) = n(n-1)/2 = (n^2 - n)/2 \\Rightarrow O(n^2)$ (Dominant term is n^2)

- **Ordering from smallest to largest growth rate for comparison:**
  - $n^{1.5}$
  - n^2
  - $n^2\\text{logn}$

- n^3
- 2^n

- **Justification for ordering:**
  - **Exponential (2^n)** grows much faster than any polynomial or polynomial-logarithmic function.
  - **Polynomials:** n^3 grows faster than $n^2\\text{logn}$, n^2, and $n^{1.5}$.
  - **Polynomial-logarithmic vs. Polynomial:** $n^2\\text{logn}$ grows faster than n^2 (because $\\text{logn}$ grows, albeit slowly).
  - **Polynomials:** n^2 grows faster than $n^{1.5}$.
- **Decreasing Order of Asymptotic Complexity:**

xi. $f\_3(n) = 2^n \\quad (O(2^n))$

xii. $f\_4(n) = n^3/1000 \\quad (O(n^3))$

xiii. $f\_1(n) = n^2.\\text{logn} \\quad (O(n^2 \\text{logn}))$

xiv. $f\_5(n) = n(n-1)/2 \\quad (O(n^2))$

xv. $f\_2(n) = n^{1.5} + 10^6 \\quad (O(n^{1.5}))$

18. (c) Suppose the following class definitions of a circular single linked list are given:

```
class Node
{
   int info;
   Node *next;
   Node(int i) {info = i; next=NULL;}
};
```

```
class IntCSLL
{
   Node *head, *tail;
   delete() // This function deletes a node from the head of the circular
linked list
   {
      Node *Temp = head;
      head->next = head;
      delete Temp;
   }

   insert (int info) // This function inserts a node at the beginning of a
single linked list
   {
      Node *pNode = new Node(info);
      pNode = head;
   }

   traverse() // This function traverses the single linked list
   {
      while (head != NULL)
      {
         cout << head->info;
         head = head->next;
      }
   }
};
```


Find the errors in *delete()*, *insert()* and *traverse()* functions given in the above code (if any). Write the corrections.

- **Errors and Corrections:**
- **1. *delete()* function:**
  - **Errors:**

- **Missing Return Type:** *delete()* is a member function, but it lacks a return type (e.g., *void*).
- **Incorrect Deletion Logic:**
    - *head->next = head;* This line attempts to make the *head* node point to itself. This is only correct if it's the *only* node left in the list after deletion. It doesn't correctly handle deletion from a circular linked list with multiple nodes.
    - If *head* is the only node, *tail* should also be set to *nullptr* (or handle the empty list case).
    - If there are multiple nodes, *tail->next* must be updated to point to the new *head*.
- **Memory Leak/Dangling Pointer:** If *head* becomes *head->next*, and *Temp* (the old head) is deleted, *head->next* (which was the next node) would be pointing to freed memory.

- **Corrections:**

```
// Assuming delete is meant to delete from the head (beginning)
void deleteNodeFromHead() // Renamed to clarify purpose and avoid keyword conflict
{
    if (head == nullptr) { // List is empty
        std::cout << "List is empty, nothing to delete." << std::endl;
        return;
    }

    Node *temp = head; // Store the current head node to be deleted

    if (head == tail) { // Only one node in the list
        head = nullptr;
```

```
        tail = nullptr;
    } else { // More than one node
        head = head->next; // Move head to the next node
        tail->next = head; // Make tail point to the new head
(circularity)
    }
    delete temp; // Free the memory of the old head node
}
```

- **2. *insert(int info)* function:**
  - **Errors:**
    - **Missing Return Type:** *insert* lacks a return type (e.g., *void*).
    - **Incorrect Insertion Logic:**
      - *pNode = head;* This line overwrites the newly created node *pNode* with the current *head* pointer. The new node created by *new Node(info)* is lost (memory leak).
      - It does not link the new node into the list at all.
      - It doesn't handle an empty list correctly (where *head* and *tail* would initially be *nullptr*).
      - For a circular linked list, the new node's *next* should point to the old *head*, and *tail->next* should point to the new node.
  - **Corrections:**

```
// Assuming insert is meant to insert at the beginning of a
circular linked list
void insert(int info)
{
    Node *pNode = new Node(info); // Create the new node
```

```
    if (head == nullptr) { // List is empty
        head = pNode;
        tail = pNode;
        tail->next = head; // Make it circular by pointing to itself
    } else { // List is not empty
        pNode->next = head; // New node points to the current
head
        head = pNode; // New node becomes the head
        tail->next = head; // Tail still needs to point to the new
head
    }
}
```

- **3. *traverse()* function:**
  - **Errors:**
    - **Missing Return Type:** *traverse* lacks a return type (e.g., *void*).
    - **Incorrect Traversal for Circular List:**
      - *while (head != NULL)*: In a circular linked list, *head* will never be *NULL* (unless the list is empty). This loop will run infinitely if the list is not empty, and *head* will keep moving.
      - The traversal logic needs to handle circularity by stopping when it returns to the starting node, or by checking if the next node is the head.
    - **Modifies *head*:** *head = head->next;* permanently changes the *head* pointer of the list, which is a major issue. A temporary pointer should be used for traversal.

- **Missing *std::cout* prefix:** *cout* is used without *std::* namespace qualifier or *using namespace std;* statement at the top of the *IntCSLL* class (though it's present in the *main* function given in a previous sub-question, it's safer to include it or qualify here).

- **Corrections:**

```
void traverse()
{
    if (head == nullptr) { // Handle empty list
        std::cout << "List is empty." << std::endl;
        return;
    }

    Node *current = head; // Use a temporary pointer for traversal
    do {
        std::cout << current->info << " ";
        current = current->next;
    } while (current != head); // Continue until we loop back to the head
    std::cout << std::endl; // New line after printing all elements
}
```