write your name ~~Ghazal~~ -5

100/100

30

Excellent

Question1.1:

T(n) = T(n/5) + 5, T(1) = 31

T(5) = T(5/5) + 5 = T(1) + 5 = 31 + 5 = 36 $T(5^1) = 31 + 5$

T(25) = T(25/5) + 5 = T(5) + 5 = 36 + 5 = 41 $T(5^2) = 31 + 5 + 5$

$T(5^3) = 31 + 5 + 5 + 5$

T(125) = T(125/5) + 5 = T(25) + 5 = 41 + 46 $T(5^x) = 31 + \sum_{i=0}^{x} 5 \quad 5\sum_{i=0}^{x} 1$

From these three examples, the general expression is: T(n) = 31 + $5 log_5(n)$ (31+5x)

explain How?

Let n = 5^x, then T(5^x) = 31 + 5x

Proof:

Base Case (x = 0):

$$T(5^0) = 31 + 5(0) = 31 + 0 = 31$$

Inductive step:

Assume that T(5^x) is true, then T(5^{x+1}) is true.

The expected answer should be: T(5^{x+1}) = 31 + 5(x + 1)

now for (x + 1): T(5^{x+1}) = T(5^x) + 5 = 31 + 5x + 5 = 31 + 5(x + 1)

∴ T(5^x) = T(5^{x-1}) + 5 is equivalent to T(5^x) = 31 + 5x

Using mathematical induction. ▮

Question 1.2:

T(n) = 2T(n/2) + n, T(1) = 1

T(2) = 2T(2/2) + 2 = 2T(1) + 2 = 2(1) + 2 = 2 + 2 = 4

T(4) = 2T(4/2) + 4 = 2T(2) + 4 = 2(4) + 4 = 8 + 4 = 12

T(8) = 2T(8/2) + 8 = 2T(4) + 8 = 2(12) + 8 = 24 + 8 = 32

From these three examples, the general expression is: $T(n) = n(log_2(n) + 1)$

Let $n = 2^x$, then: $T(2^x) = 2^x(x + 1)$

How ?

Proof:

Base step (x = 0):

$$T(2^0) = 2^0(0 + 1) = 1(1) = 1$$

Inductive step:

Assume that $T(2^x)$ is true, then $T(2^{x+1})$ is true

The expected answer should be: $T(2^{x+1}) = 2^{x+1}(x + 2)$

Now for (x + 1):

$$T(2^{x+1}) = 2T(2^x) + 2^{x+1} = 2(2^x(x + 1)) + 2^{x+1} = 2^{x+1}(x + 1) + 2^{x+1}$$

$$= 2^{x+1}(x + 2)$$

∴ $T(2^x) = 2T(2^{x-1}) + 2^x$ is equivalent to $T(2^x) = 2^x(x + 1)$

Using mathematical induction. ▌

Question 1.3:

Theorem: $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

T(n) = T(n − 1) + n^2 , T(1) = 23

T(2) = T(1) + 4 = 23 + 4 = 27

T(3) = T(2) + 9 = 27 + 9 = 36

T(4) = T(3) + 16 = 36 + 16 = 52

From these three examples, the general expression is: T(n) = 22 + $\sum_{i=1}^{n} i^2$

= 22 + $\frac{n(n+1)(2n+1)}{6}$

Proof:

Base step (n = 1):

$$T(1) = 22 + \frac{1(1+1)(2(1)+1)}{6} = 22 + 1 = 23$$

Inductive step:

Assume that T(n) is true, then T(n + 1) is true.

The expected answer should be: T(n + 1) = 22 + $\frac{(n+1)(n+2)(2n+3)}{6}$

Now for (n + 1):

$$T(n+1) = T(n) + (n + 1)^2 = 22 + \frac{(n+1)(n+2)(2n+3)}{6} + (n + 1)^2$$

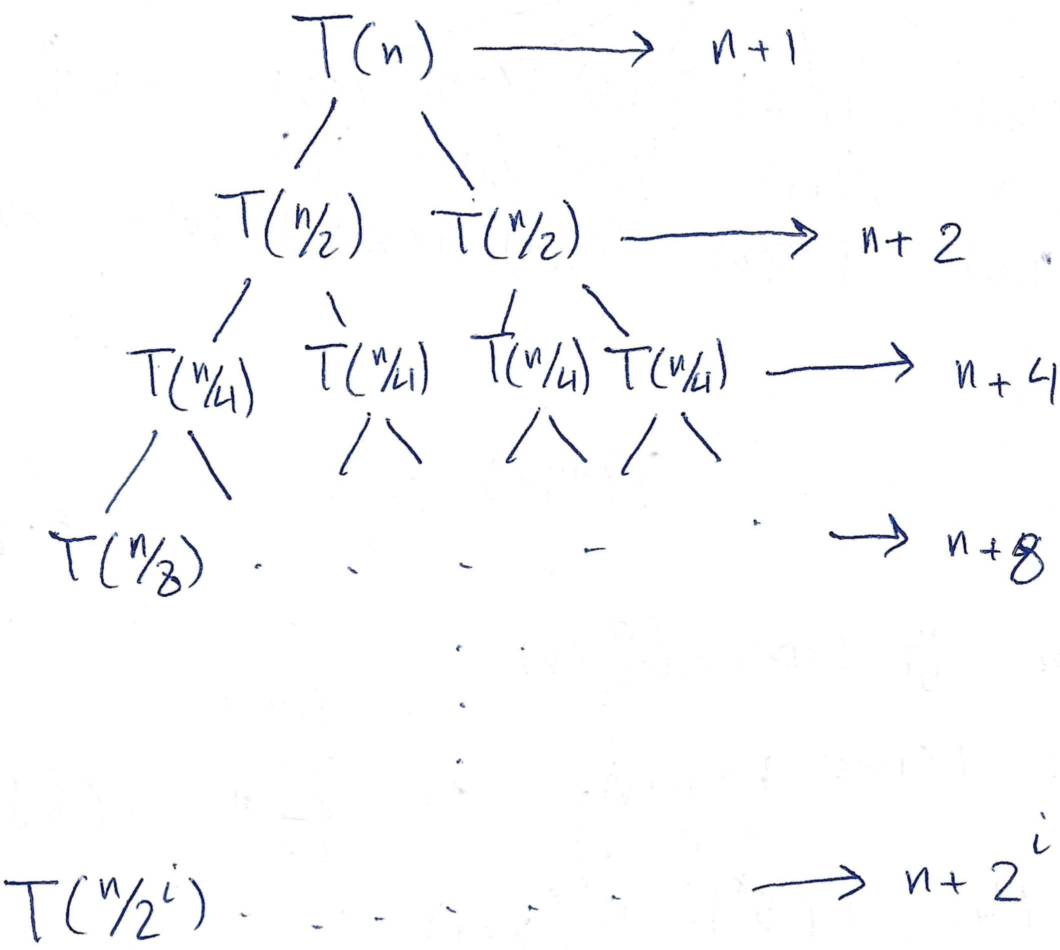
$$= 22 + \frac{(n+1)(n+2)(2n+3) + 6(n+1)^2}{6} = 22 + \frac{(n+1)(n+2)(2n+3)}{6}$$

∴ T(n) = T(n - 1) + n^2 is equivalent to T(n) = 22 + $\frac{n(n+1)(2n+1)}{6}$

Using mathematical induction. ▮

$T(n) = 2T(n/2) + n + 1, \quad T(1) = 1$

~~P2~~ 10/10



Stopping case: $\frac{n}{2^i} = 1 \Rightarrow i = \log_2(n)$

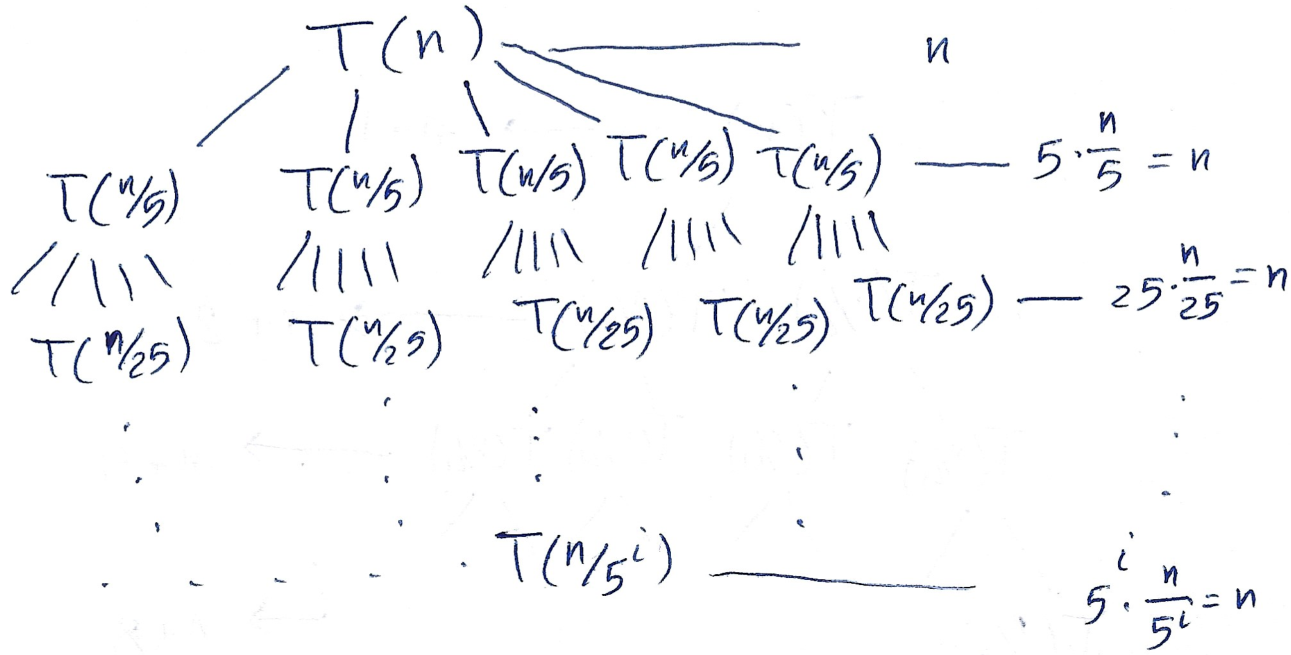
$\therefore$ The sum of terms: $(n+1) + (n+2) + \dots + (n+2^i) = \sum_{i=0}^{\log_2(n)} (n+2^i)$

$$\Rightarrow \sum_{i=0}^{\log_2(n)} (n+2^i) = \sum_{i=0}^{\log_2(n)} n + \sum_{i=0}^{\log_2(n)} 2^i = n(\log_2(n)+1) + 2^{\log_2(n)+1} - 1$$

let $n = 2^x$, then: $T(2^x) = 2^x(x+2) - 1$

so, what is the complexity?

$T(n) = 5\,T(n/5) + n, \quad T(1) = 1$



Stopping case: $\frac{n}{5^i} = 1 \Rightarrow i = \log_5(n)$

Sum of the terms: $n + n + n + \ldots + n = \sum_{i=0}^{\log_5(n)} n = n(\log_5(n) + 1)$

let $n = 5^x$, then: $T(5^x) = 5^x(x+1)$

Complexity?

```cpp
//Name: Ghazi Najeeb Al-Abbar
//ID: 2181148914
//Problems: 3 - 4 - 5 - 6
//problem-3.cpp

//Let n be the number of nodes in the binary tree

#include <iostream>
#include <queue>
using namespace std;

//Node class
struct Node{

    //Constructor
    //Makes every child of a node point to null
    //Assigns the user input as data
    Node(int Data): data(Data)
    {
        this->right = nullptr;
        this->left = nullptr;
    }

    Node* right;
    Node* left;
    int data;
};

//Binary Search Tree Class
class BST{

public:

    //Constructor
    //Makes the root point to null when instantiating a new BST object
    BST()
    {
        this->root = nullptr;
    }

//Problem 3
/************************************************************************************

    //Returns true if the input data is found within the tree
    //Time Complexity: O(log(n)) because searching goes through the nodes by cutting down
    bool Search(int key){

        Node* temp = root; //temp node to hold the root (copy of the root, not reference)

        return searchRec(key, temp); //Calls searchRec to return true or false
    }

    //Adds a node with user input data to the tree
    //Time Complexity: O(log(n)) because going through the nodes means cutting down the l
    void add(int key){addRec(key, root);} //Calls addRec to add a node to the tree. The root
```

```
    //Deletes a node from the tree
    //Time Complexity: O(log(n)) because going through the nodes means cutting down the l
    void Delete(int key){deleteRec(root, key);} //Calls deleteRec to remove a node from the t

    //Prints the tree in order
    //Time Complexity: O(n) since it goes through all the whole tree
    void printInorder(){

        Node* temp = root; //temp node to hold the root (copy of the root, not reference)

        Inorder(temp); //Calls the Inorder function to print the tree in order

        cout << endl;
    }

    //Prints the tree post order
    //Time Complexity: O(n) since it goes through all the whole tree
    void printPostorder(){

        Node* temp = root; //temp node to hold the root (copy of the root, not reference)

        Postorder(temp); //Calls the Postorder function to print the tree post order

        cout << endl;
    }

    //Prints the tree pre-order
    //Time Complexity: O(n) since it goes through all the whole tree
    void printPreorder(){

        Node* temp = root; //temp node to hold the root (copy of the root, not reference)

        Preorder(temp); //Calls the Preorder function to print the tree pre-order

        cout << endl;
    }

/*************************************************************************************

    //problem 4
    //Counts and returns the number of leaves in the binary tree
    //Time Complexity: O(n) since it goes through all the whole tree
    int LeafCount(){

        int Count = 0; //Counter that counts the number of leaves

        Node* temp = root; //temp node to hold the root (copy of the root, not reference)

        LeafCountRec(temp, Count); //Calls the function LeafCountRec to count the number of l

        return Count; //Returns the number of leaves
    }

    //problem 5
```

```cpp
    //Finds and returns the kth smallest element from the binary tree.
    //Time Complexity: O(n) since it goes through all the whole tree
    int KthSmallest(int k){

        //Checks if the root is null. If it was, the function would end and returns nothing
        if (root == nullptr)
            cout << "The tree is Empty!\n";
```

P5

25/25

```cpp
        //The case where the tree is not empty.
        else{

            //Queue to hold all the data from the tree in (in order) sequence
            queue<int> Q; //The data is positioned from the smallest to the largest
            Node* temp = root; //temp node to hold the root (copy of the root, not reference)

            traverse(temp, Q); //Calls the traverse function to store the data in the queue.

            //goes through the queue and pops the front k - 1 times
            for (int i = 1; i < k; i++)
                Q.pop();

            return Q.front(); //Returns the front (the kth element)
        }
    }

    //problem 6
    //Checks if the binary tree is complete. Returns true if it is. Otherwise, returns fa
    //Time Complexity: O(n) since it goes through all the whole tree
    bool isComplete(){
```

P6

25/25

```cpp
        //If the tree is empty, then by definition, it is complete. Returns true
        if (root == nullptr)
            return true;

        bool isNotComplete = false; //Boolean value that checks if the tree is empty or not.

        queue<Node*> Q; //Queue to store the nodes in descending order
        Q.push(root); //enqueues the root

        //Goes through the tree and checks whether it is complete or not
        while (!Q.empty()){

            Node* temp = Q.front(); //Temporary node to hold the front of the queue
            Q.pop(); //The queue is popped (handles the descending movement across the tree)

            /* Since isNotComplete starts out as false, then the queue just enqueues the left
               If it happens to find a null pointer between two non-null pointing nod
               isNotComplete switches to true and the function returns returns false.

               If the queue reached a null pointer and no other non-null pointing nod
            */

            //checks whether the temp node is null
            if (temp == nullptr)
                isNotComplete = true; //isNotComplete is switched to tue
```

P3

6/6

```

            //if temp is not empty
            else{

                //If isNotComplete is true while temp is not empty, then the tree is not comp
                if (isNotComplete)
                    return false;

                Q.push(temp->left); //enqueues the left child first
                Q.push(temp->right); //enqueues the second child next
            }
        }

        //If the end of the loop has been reached, then the tree is a complete tree. Returns
        return true;
    }

private: //Encapsulated members and functions

    Node* root; //Holds the root node for the tree

    //adds a node with data equal to key to the tree by searching for its destination rec
    //Time Complexity: O(log(n)) because going through the nodes means cutting down the l
    void addRec(int key, Node*& node){

    //Checks if the current node points to null
    if (node == nullptr){

        Node* NewNode = new Node(key); //creates new node with data equal to key

        node = NewNode; //The current node is assigned to the new node

        cout << "Node with data " << key << " is added!\n";

        return; //Leaves the function
    }

    //If the current node data is greater than or equal to key
    if (node->data >= key)
        return addRec(key, node->left); //Moves to the left node recursivly

    //If the current node data is less than key
    else
        return addRec(key, node->right); //Moves to the right node recursivly

    }

    //Checks if key is a data for a node within the tree. Returns true if the node exists
    //Time Complexity: O(log(n)) because searching goes through the nodes by cutting down
    bool searchRec(int key, Node* node){

        //if the current node is null, then the node does not exist. Returns false
        if (node == nullptr)
            return false;

```

```
        //If the current node is equal to the key value, then it returns true
        if (key == node->data)
            return true;

        //If the current node data is greater than or equal to the key value
        if ( node->data >= key)
            return searchRec(key, node->left); //Moves to the left child recursivly

        //If the current node data is less than the key value
        else
            return searchRec(key, node->right); //Moves to the right child recursivly
    }

    //Prints the tree in order
    //Time Complexity: O(n) Since it goes through the whole tree
    void Inorder(Node* node){

        //if the current node is pointing to null
        if (node == nullptr)
            return; //leaves function or recursive call

        Inorder(node->left); //recursivly move to the left child

        cout << node->data << "  "; //prints the data of the current node

        Inorder(node->right); //recursivly move to the right child
    }

    //Prints the tree in post order
    //Time Complexity: O(n) Since it goes through the whole tree
    void Postorder(Node* node){

        //if the current node is pointing to null
        if (node == nullptr)
            return; //leaves function or recursive call

        Postorder(node->left); //recursivly move to the left child

        Postorder(node->right); //recursivly move to the right child

        cout << node->data << "  "; //prints the data of the current node
    }

    //Prints the tree in post order
    //Time Complexity: O(n) Since it goes through the whole tree
    void Preorder(Node* node){

        //if the current node is pointing to null
        if (node == nullptr)
            return; //leaves function or recursive call

        cout << node->data << "  "; //prints the data of the current node

        Preorder(node->left); //recursivly move to the left child

```

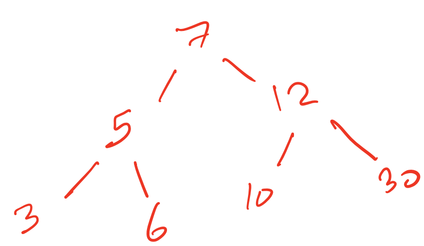
```
        Preorder(node->right); //recursivly move to the right child
    }

    //Finds the node data with the smallest value after a certain node
    //Time Complexity: O(log(n)) since it goes left using a fraction of n steps
    int MinValue(Node* node){

        Node* temp = node; //temp node to hold the root (copy of the root, not reference)

        //Moves left until it reaches the node that points to null
        while (temp->left != nullptr)
            temp = temp->left; //temp is temp's left child

        return temp->data; //Returns the node data of the far left child
    }

    //Deletes a node from the tree
    //Time Complexity: O(log(n)) because going through the nodes means cutting down the l
    void deleteRec(Node*& node, int key){

        //if the node with data equal to key is not found
        if (node == nullptr){

            cout << "No such Node exists with data " << key <<endl; //prints message

            return; //leaves function
        }

        //If the current node data is equal to the key value, then the deletion process begin
        if (node->data == key){

            //If the current node has no children
            if (node->left == nullptr && node->right == nullptr){

                cout << "node with data " << key << " succesfully deleted\n";

                Node* temp = node; //Stores the current node address in temp
                delete temp; //deletes temp (frees up the memory)

                node = nullptr; //current node points to null

                return; //leaves function
            }


            //If the current node only has a left child
            if (node->left != nullptr && node->right == nullptr){

                cout << "node with data " << key << " succesfully deleted\n";

                Node* temp = node->left; //stores the current node's left child's address in

                node = node->left; //Current is assigned to its left child

                delete temp; //deletes temp (frees up the memory)
```

```

                return; //Leaves the function
            }

            //If the current node only has a right child
            if (node->left == nullptr && node->right != nullptr){

                cout << "node with data " << key << " succesfully deleted\n";

                Node* temp = node->right; //stores the current node's right child's address i

                node = node->right; //Current is assigned to its right child

                delete temp; //Current is assigned to its left child

                return; //Leaves the function
            }


            //If the node has both a right and a left child
            if (node->left != nullptr && node->right != nullptr){

                node->data = MinValue(node->right); //Current node data is the smallest node

                deleteRec(node->right, MinValue(node->right)); //deleteRec is called to remov

                cout << "node with data " << key << " succesfully deleted\n";

                return; //Leaves function
            }
        }

        //If the node data is greater than the key value
        if (node->data > key)
            return deleteRec(node->left, key); //Recursivly move to the left child

        //If the node data is less than the key value
        if (node->data < key)
            return deleteRec(node->right, key); //Recursivly move to the right child
    }

    //problem 4
    //Counts and returns the number of leaves in the binary tree
    //Time Complexity: O(n) since it goes through all the whole tree
    void LeafCountRec(Node* node, int& Count){

        //if the current node points to null
        if (node == nullptr)
            return; //leaves the function or recursive call

        //If the node's left child and right child are both empty, then it is a leaf
        if (node->right == nullptr && node->left == nullptr){

            Count += 1; //increment the counter by one

```

P4

4 / 4

```
            return; //leaves the function or recursive call
        }

        //if the current node only has a right child
        if (node->right != nullptr && node->left == nullptr)
            LeafCountRec(node->right, Count); //Recusrivly move to the right child

        //if the current node only has a left child
        if (node->right == nullptr && node->left != nullptr)
            LeafCountRec(node->left, Count); //Recusrivly move to the right child

        //if the current node has both left and right children
        if (node->right != nullptr && node->left != nullptr){

            LeafCountRec(node->left, Count); //Recusrivly move to the left child

            LeafCountRec(node->right, Count); //Recusrivly move to the right child

            return; //leaves the function or recursive call
        }
    }

    //Function that traverses through the tree in order, and enqueues all node data into
    //By the end of the function, the queue has all the tree's node data in ascending ord
    //Time Complexity: O(n) Since it goes through the whole tree
    void traverse(Node* node, queue<int>& Q){

        //If the current node is empty
        if (node == nullptr)
            return; //leaves the function or recursive call

        traverse(node->left, Q); //Recusrivly move to the left child
        Q.push(node->data); //enqueues the current node data into the queue
        traverse(node->right, Q); //Recusrivly move to the right child
    }
};

//Beginning of program
int main(){

    BST tree;

    tree.add(7);
    tree.add(12);
    tree.add(5);
    tree.add(3);
    tree.add(6);
    tree.add(10);
    tree.add(30);

    cout << "Is the there a node with data 12 in the tree? The answer is: ";

    if (tree.Search(12))
        cout << "True\n";
    else
```



```
        cout << "False\n";

    cout << "Is the there a node with data 100 in the tree? The answer is: ";

    if (tree.Search(100))
        cout << "True\n";
    else
        cout << "False\n";

    cout << "Printing in order: ";
    tree.printInorder();

    cout << "Printing post order: ";
    tree.printPostorder();

    cout << "Printing pre-order: ";
    tree.printPreorder();

    cout << "The number of leaves is " << tree.LeafCount() << endl;

    cout << "The 4th smallest element is " << tree.KthSmallest(4) << endl;

    cout << "Is the tree complete? The answer is: ";

    if (tree.isComplete())
        cout << "True\n";
    else
        cout << "False\n";

    tree.Delete(7);

    cout << "Is the there a node with data 7 in the tree? The answer is: ";

    if (tree.Search(7))
        cout << "True\n";
    else
        cout << "False\n";

    cout << "Is the there a node with data 30 in the tree? The answer is: ";

    if (tree.Search(30))
        cout << "True\n";
    else
        cout << "False\n";

    cout << "Printing in order: ";
    tree.printInorder();

    cout << "Printing post order: ";
    tree.printPostorder();

    cout << "Printing pre-order: ";
    tree.printPreorder();

    cout << "The number of leaves is " << tree.LeafCount() << endl;
```

```

    cout << "The 4th smallest element is " << tree.KthSmallest(4) << endl;

    cout << "Is the tree complete? The answer is: ";
    if (tree.isComplete())
        cout << "True\n";
    else
        cout << "False\n";

    return 0;
}
//End of program
```