Name : Deshmukh Mehmood Rehan

MIS No. : 612303050

SY Computer Science – Div 1

QuestionImplement ADT Binary Search Tree(BST) using a linked list of nodes of structure having Month, left pointer, right pointer, and parent pointers pointing to left sub-trees, right sub-tree, and parent of the node.

Perform a series of insertions on keys - December, January, April, March, July, August, October, February, November, May, June. Write following functions:

initBST() // to initialize the tree.

insertNode() // non-recursive function to add a new node to the BST.

removeNode() // to remove a node from a tree.

traverse() // write any of the non-recursive traversal methods.

destroyTree() // to delete all nodes of a tree.

Write a menu driven program to invoke all above functions.

Code:

header.h : This File includes the declarations of structures and function prototypes

```
/*
Implement ADT Binary Search Tree(BST) using a linked list of nodes of
structure having Month, left pointer, right pointer, and parent pointers
pointing to left sub-trees, right sub-tree, and parent of the node.

Perform a series of insertions on keys - December, January, April, March,
July, August, October, February, November, May, June.  Write following
functions:

initBST()         // to initialize the tree.

insertNode()    // non-recursive function to add a new node to the BST.

removeNode() // to remove a node from a tree.

traverse()         // write any of the non-recursive traversal methods.

destroyTree()  // to delete all nodes of a tree.
```

```c
Write a menu driven program to invoke all above functions.
*/
#ifndef HEADER_H
#define HEADER_H


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/*
This is the structure of a node in the binary search tree.
it contains:
    - month: the month name
    - left: pointer to the left child
    - right: pointer to the right child
    - parent: pointer to the parent of the node
*/

typedef struct Node {
    char *month;
    struct Node *left;
    struct Node *right;
    struct Node *parent;
} Node;

/*
This is the binary search tree structure.
it contains:
    - root: pointer to the root of the tree
*/

typedef Node * BST;

/* Function prototypes */

void init_bst(BST *root);

void recursive_insert_node(BST *root, char *month);
void iterative_insert_node(BST *root, char *month);

void recursive_remove_node(BST *root, char *month);
void iterative_remove_node(BST *root, char *month);


void recursive_inorder_traversal(BST root);
void iterative_inorder_traversal(BST root);
```

```
void recursive_preorder_traversal(BST root);
void iterative_preorder_traversal(BST root);

void recursive_postorder_traversal(BST root);
void iterative_postorder_traversal(BST root);

void level_order_traversal(BST root);

void recursive_destroy_tree(BST *root);
void iterative_destroy_tree(BST *root);

int iterative_count_leaf(BST root);
int recursive_count_leaf(BST root);

int iterative_count_non_leaf(BST root);
int recursive_count_non_leaf(BST root);

int iterative_get_height(BST root);
int recursive_get_height(BST root);

#endif
```

logic.c : contains the definition of all the functions declared in the header file along with some helper functions

```
#include "header.h"
#include "stack.h"
#include "queue.h"


/*
This function initializes the binary search tree.
It takes a pointer to the root of the tree and initializes it to NULL.
*/
void init_bst(BST *root) {
    *root = NULL;
}

/*
This function inserts a node in the binary search tree.
It takes a pointer to the root of the tree and the month to be inserted.

The function allocates memory for the new node and copies the month to the new
node.
Then it traverses the tree to find the correct position for the new node.

If the tree is empty, the new node is inserted as the root.
```

```c
/*
If the tree is not empty, the function traverses the tree to find the correct
position for the new node.
If the month already exists in the tree, the function frees the memory
allocated for the new node and returns.
*/

void iterative_insert_node(BST *root, char *month) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->month = (char *)malloc(strlen(month) + 1);
    strcpy(newNode->month, month);
    newNode->left = newNode->right = newNode->parent = NULL;


    if (*root == NULL) {
        *root = newNode;
        return;
    }

    Node *temp = *root;
    while (temp) {
        if(strcmp(month, temp->month) == 0) {
            free(newNode->month);
            free(newNode);
            return;
        }

        if (strcmp(month, temp->month) < 0) {
            if (!temp->left) {
                temp->left = newNode;
                newNode->parent = temp;
                return;
            }
            temp = temp->left;
        } else {
            if (!temp->right) {
                temp->right = newNode;
                newNode->parent = temp;
                return;
            }
            temp = temp->right;
        }
    }
}

/*
This function also inserts a node in the binary search tree.
but it does so recursively.
*/
```

```c
void recursive_insert_node(BST *root, char *month){
    Node *temp = *root;

    if(!temp){
        Node *newNode = (Node *) malloc(sizeof(Node));
        newNode->month = (char *) malloc(strlen(month) + 1);
        strcpy(newNode->month, month);
        newNode->left = newNode->right = newNode->parent = NULL;
        *root = newNode;
        return;
    }

    if(strcmp(month, temp->month) < 0) recursive_insert_node(&temp->left,
month);
    else recursive_insert_node(&temp->right, month);
}

/*
This function removes a node from the binary search tree.
It takes a pointer to the root of the tree and the month to be removed.

The function traverses the tree to find the node with the given month.
If the node is not found, the function returns.

If the node is found, the function checks the following cases:
1. If the node is a leaf node, the function frees the memory allocated for the
node and returns.

2. If the node has only one child, the function links the parent of the node
to the child of the node and frees the memory allocated for the node.

3. If the node has two children, the function finds the inorder successor of
the node, copies the month of the inorder successor to the node, and removes
the inorder successor.

The function also handles the case when the node to be removed is the root of
the tree.

*/

void iterative_remove_node(BST *root, char *month) {
    if(!*root) return;

    Node *temp = *root;
    while (temp != NULL && strcmp(month, temp->month) != 0) {
        if (strcmp(month, temp->month) < 0) {
            temp = temp->left;
        } else {
```

```
            temp = temp->right;
        }
    }
    if (!temp) {
        return;
    }

    if (!temp->left && !temp->right) {
        if(temp->parent){
            if(temp->parent->left == temp) temp->parent->left = NULL;
            else temp->parent->right = NULL;
        }else{
            *root = NULL;
        }

        free(temp->month);
        free(temp);
    } else if (!temp->left) {
        if(temp->parent){
            if(temp->parent->left == temp) temp->parent->left = temp->right;
            else temp->parent->right = temp->right;
        }else{
            *root = temp->right;
        }

        free(temp->month);
        free(temp);
    } else if (!temp->right) {
        if(temp->parent){
            if(temp->parent->left == temp) temp->parent->left = temp->left;
            else temp->parent->right = temp->left;
        } else{
            *root = temp->left;
        }

        free(temp->month);
        free(temp);
    } else {
        Node *successor = temp->right;
        while (successor->left != NULL) {
            successor = successor->left;
        }
        strcpy(temp->month, successor->month);
        free(successor->month);
        if(successor->parent->left == successor) successor->parent->left =
successor->right;
        else successor->parent->right = successor->right;
        free(successor);
```

```
    }


    return;
}

/*
Helper function to get the inorder successor of a node.
*/
Node *get_inorder_successor(Node *root){
    Node *temp = root->right;
    while(temp->left){
        temp = temp->left;
    }
    return temp;
}

/*
This function also removes a node from the binary search tree.
but it does so recursively.
*/

void recursive_remove_node(BST *root, char *month) {
    if(!*root) return;

    if(strcmp(month, (*root)->month) < 0){
        recursive_remove_node(&(*root)->left, month);
    }else if(strcmp(month, (*root)->month) > 0){
        recursive_remove_node(&(*root)->right, month);
    }else{
        Node *temp = *root;
        if(!temp->left && !temp->right){
            if(temp->parent){
                if(temp->parent->left == temp) temp->parent->left = NULL;
                else temp->parent->right = NULL;
            }else{
                *root = NULL;
            }
            free(temp->month);
            free(temp);
        }else if(!temp->left){
            if(temp->parent){
                if(temp->parent->left == temp) temp->parent->left = temp-
>right;
                else temp->parent->right = temp->right;
            }else{
                *root = temp->right;
            }
            free(temp->month);
```

```
            free(temp);
        }else if(!temp->right){
            if(temp->parent){
                if(temp->parent->left == temp) temp->parent->left = temp-
>left;
                else temp->parent->right = temp->left;
            }else{
                *root = temp->left;
            }
            free(temp->month);
            free(temp);
        }else{
            Node *successor = get_inorder_successor(temp);
            strcpy(temp->month, successor->month);
            recursive_remove_node(&temp->right, successor->month);
        }

    }
}

/*
This function traverses the binary search tree in inorder.
It takes the root of the tree as an argument.

The function traverses the left subtree, visits the root, and then traverses
the right subtree.
*/

void recursive_inorder_traversal(BST root){
    if(!root) return;

    recursive_inorder_traversal(root->left);
    printf("%s ", root->month);
    recursive_inorder_traversal(root->right);
}

/*
This function also traverses the binary search tree in inorder.
but it does so iteratively using a stack.
*/

void iterative_inorder_traversal(BST root){
    Stack s;
    init_stack(&s);
    Node *temp = root;
    while(1){
        if(temp){
            push_stack(&s, temp);
```

```
            temp = temp->left;
        }else{
            if(is_empty_stack(s)) break;
            Node *node = pop_stack(&s);
            printf("%s ", node->month);
            temp = node->right;
        }
    }

    printf("\n");
}
/*
This function traverses the binary search tree in preorder.
It takes the root of the tree as an argument.

The function visits the root, traverses the left subtree, and then traverses
the right subtree.
*/
void recursive_preorder_traversal(BST root){
    if(!root) return;

    printf("%s ", root->month);
    recursive_preorder_traversal(root->left);
    recursive_preorder_traversal(root->right);
}

/*
This function also traverses the binary search tree in preorder.
but it does so iteratively using a stack.
*/

void iterative_preorder_traversal(BST root){
    if(!root) return;

    Stack s;
    init_stack(&s);
    push_stack(&s, root);

    while(!is_empty_stack(s)){
        Node *temp = pop_stack(&s);
        printf("%s ", temp->month);
        if(temp->right) push_stack(&s, temp->right);
        if(temp->left) push_stack(&s, temp->left);
    }

    printf("\n");
}
```

```c
/*
This function traverses the binary search tree in postorder.
It takes the root of the tree as an argument.

The function traverses the left subtree, then the right subtree, and finally
visits the root.
*/
void recursive_postorder_traversal(BST root){
    if(!root) return;

    recursive_postorder_traversal(root->left);
    recursive_postorder_traversal(root->right);
    printf("%s ", root->month);
}

/*
This function also traverses the binary search tree in postorder.
but it does so iteratively using two stacks.
*/

void iterative_postorder_traversal(BST root){
    if(!root) return;

    Stack s1, s2;
    init_stack(&s1);
    init_stack(&s2);

    push_stack(&s1, root);

    while(!is_empty_stack(s1)){
        Node *temp = pop_stack(&s1);
        push_stack(&s2, temp);

        if(temp->left) push_stack(&s1, temp->left);
        if(temp->right) push_stack(&s1, temp->right);
    }

    while(!is_empty_stack(s2)){
        Node *temp = pop_stack(&s2);
        printf("%s ", temp->month);
    }

    printf("\n");
}

/*
This function traverses the binary search tree in level order.
It takes the root of the tree as an argument.
```

```
The function uses a queue to traverse the tree level by level.

It enqueues the root, then dequeues the node, prints the month, and enqueues
the left and right children of the node.
*/

void level_order_traversal(BST root){
    if(!root) return;

    Queue q;
    init_queue(&q);
    enqueue(&q, root);

    while(!is_empty_queue(q)){
        int nodeCount = get_queue_size(q);
        for(int i = 0; i < nodeCount; i++){
            Node *temp = dequeue(&q);
            printf("%s ", temp->month);
            if(temp->left) enqueue(&q, temp->left);
            if(temp->right) enqueue(&q, temp->right);
        }
    }

    printf("\n");
}

/*
This function destroys the binary search tree.
It takes a pointer to the root of the tree.

The function traverses the tree postorder and frees the memory allocated for
each node.
It also frees the memory allocated for the month of each node.
*/

void recursive_destroy_tree(BST *root) {
    if (*root == NULL) {
        return;
    }
    recursive_destroy_tree(&(*root)->left);
    recursive_destroy_tree(&(*root)->right);
    free((*root)->month);
    free(*root);

    *root = NULL;
}
```

```c
/*
This function also destroys the binary search tree.
but it does so iteratively using a stack.
*/

void iterative_destroy_tree(BST *root) {
    if(!*root) return;

    Stack s;
    init_stack(&s);
    push_stack(&s, *root);

    while (!is_empty_stack(s)) {
        Node *temp = pop_stack(&s);
        if (temp->left) push_stack(&s, temp->left);
        if (temp->right) push_stack(&s, temp->right);
        free(temp->month);
        free(temp);
    }
    *root = NULL;
}

/*
This function counts the number of leaf nodes in the binary search tree.
It takes the root of the tree as an argument.
*/

int recursive_count_leaf(BST root) {
    if(!root) return 0;

    if(!root->left && !root->right) return 1;

    return recursive_count_leaf(root->left) + recursive_count_leaf(root-
>right);
}

/*
This function also counts the number of leaf nodes in the binary search tree.
but it does so iteratively using a queue.
*/

int iterative_count_leaf(BST root) {
    if(!root) return 0;

    Queue q;
    init_queue(&q);

    enqueue(&q, root);
```

```
    int count = 0;

    while(!is_empty_queue(q)){
        int nodeCount = get_queue_size(q);
        for(int i = 0; i < nodeCount; i++){
            Node *temp = dequeue(&q);
            if(!temp->left && !temp->right) count++;
            if(temp->left) enqueue(&q, temp->left);
            if(temp->right) enqueue(&q, temp->right);
        }
    }

    return count;
}

/*
This function counts the number of non-leaf nodes in the binary search tree.
It takes the root of the tree as an argument.
*/

int recursive_count_non_leaf(BST root) {
    if(!root) return 0;

    if(!root->left && !root->right) return 0;

    return 1 + recursive_count_non_leaf(root->left) +
recursive_count_non_leaf(root->right);
}

/*
This function also counts the number of non-leaf nodes in the binary search
tree.
but it does so iteratively using a queue.
*/

int iterative_count_non_leaf(BST root) {
    if(!root) return 0;

    Queue q;
    init_queue(&q);

    enqueue(&q, root);

    int count = 0;
    while(!is_empty_queue(q)){
        int nodeCount = get_queue_size(q);
        for(int i = 0; i < nodeCount; i++){
            Node *temp = dequeue(&q);
```

```
            if(temp->left || temp->right) count++;
            if(temp->left) enqueue(&q, temp->left);
            if(temp->right) enqueue(&q, temp->right);
        }
    }

    return count;
}

/*
This function gets the height of the binary search tree.
It takes the root of the tree as an argument.

The function uses a queue to traverse the tree level by level.
It enqueues the root, then dequeues the node, and enqueues the left and right
children of the node.
The function increments the height after traversing each level.
*/

int iterative_get_height(BST root) {
    if(!root) return 0;

    Queue q;
    init_queue(&q);
    enqueue(&q, root);

    int height = 0;
    while(!is_empty_queue(q)){
        int nodeCount = get_queue_size(q);
        height++;
        for(int i = 0; i < nodeCount; i++){
            Node *temp = dequeue(&q);
            if(temp->left) enqueue(&q, temp->left);
            if(temp->right) enqueue(&q, temp->right);
        }
    }

    return height;
}

/*
This function also gets the height of the binary search tree.
but it does so recursively.
*/

int recursive_get_height(BST root) {
    if(!root) return 0;
```

```
    int leftHeight = recursive_get_height(root->left);
    int rightHeight = recursive_get_height(root->right);

    return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);
}
```

stack.h : This File includes the declarations of structures and function prototypes of the stack data structure

```
#ifndef STACK_H
#define STACK_H

#include "header.h"

typedef struct stack_node {
    Node *data;
    struct stack_node *next;
} stack_node;

typedef stack_node *Stack;

void init_stack(Stack *s);
void push_stack(Stack *s, Node *data);
Node *pop_stack(Stack *s);
Node *peek_stack(Stack s);
int is_empty_stack(Stack s);
int get_stack_size(Stack s);

#endif
```

stack.c : contains the definition of all the functions declared in the stack.h file

```
#include "./stack.h"

void init_stack(Stack *s){
    *s = NULL;
    return;
}

void push_stack(Stack *s, Node *data){
    stack_node *newNode = (stack_node *)malloc(sizeof(stack_node));
    if(!newNode) return;

    newNode->data = data;
    newNode->next = *s;

    *s = newNode;
```

```
    return;
}

Node *pop_stack(Stack *s){
    if(is_empty_stack(*s)) return NULL;

    stack_node *removedNode;
    Node *removedElement;

    removedNode = *s;
    removedElement = removedNode->data;
    *s = (*s)->next;

    free(removedNode);
    return removedElement;
}

Node *peek_stack(Stack s){
    if(is_empty_stack(s)) return NULL;

    return s->data;
}

int is_empty_stack(Stack s){
    if(!s) return 1;

    return 0;
}

int get_stack_size(Stack s){
    int count = 0;
    while(s){
        count++;
        s = s->next;
    }

    return count;
}
```

queue.h : This File includes the declarations of structures and function prototypes of the queue data structure

```
#ifndef QUEUE_H
#define QUEUE_H

#include "header.h"
```

```c
typedef struct queue_node {
    Node *data;
    struct queue_node *next;
} queue_node;

typedef struct{
    queue_node *front;
    queue_node *rear;
} Queue;

void init_queue(Queue *q);
void enqueue(Queue *q, Node *data);
Node *dequeue(Queue *q);
Node *peek_queue(Queue q);
int is_empty_queue(Queue q);
int get_queue_size(Queue q);

#endif
```

queue.c : contains the definition of all the functions declared in the queue.h file

```c
#include "./queue.h"

void init_queue(Queue *q){
    q->front = q->rear = NULL;
    return;
}

void enqueue(Queue *q, Node *data){
    queue_node *newNode = (queue_node *)malloc(sizeof(queue_node));
    if(!newNode) return;

    newNode->data = data;
    newNode->next = NULL;

    if(!q->front){
        q->front = q->rear = newNode;
        return;
    }

    q->rear->next = newNode;
    q->rear = newNode;

    return;
}

Node *dequeue(Queue *q){
    if(is_empty_queue(*q)) return NULL;
```

```
    queue_node *removedNode;
    Node *removedElement;

    removedNode = q->front;
    removedElement = removedNode->data;

    if(q->front == q->rear){
        q->front = q->rear = NULL;
    }else{
        q->front = q->front->next;
    }

    free(removedNode);
    return removedElement;
}

Node *peek_queue(Queue q){
    if(is_empty_queue(q)) return NULL;

    return q.front->data;
}

int is_empty_queue(Queue q){
    if(!q.front) return 1;

    return 0;
}

int get_queue_size(Queue q){
    int count = 0;

    queue_node *temp = q.front;
    while(temp){
        count++;
        temp = temp->next;
    }

    return count;
}
```

main.c : This contains the code to test the implementation

```
#include "header.h"

void print_menu();
void handle_choice(BST *root, int choice);
```

```c
void lower_string(char
s[]);


int main() {
    BST root;
    init_bst(&root);
    int choice;
    while(1){
        print_menu();
        printf("Enter your choice: ");
        scanf("%d", &choice);
        handle_choice(&root, choice);
    }
    return 0;
}

void lower_string(char s[]) {
    for(int i = 0; s[i]; i++){
        s[i] = tolower(s[i]);
    }
}

void print_menu() {
    printf("\n1. Insert a node\n");
    printf("2. Remove a node\n");
    printf("3. Inorder traversal\n");
    printf("4. Preorder traversal\n");
    printf("5. Postorder traversal\n");
    printf("6. Level order traversal\n");
    printf("7. Destroy tree\n");
    printf("8. Count leaf nodes\n");
    printf("9. Count non-leaf nodes\n");
    printf("10. Get height of tree\n");
    printf("11. Exit\n");
}

void handle_choice(BST *root, int choice) {
    char month[20];
    switch(choice){
        case 1:
            printf("Enter month: ");
            scanf("%s", month);
            lower_string(month);
            iterative_insert_node(root, month);
            break;
        case 2:
            printf("Enter month: ");
```

```
            scanf("%s", month);
            lower_string(month);
            iterative_remove_node(root, month);
            break;
        case 3:
            iterative_inorder_traversal(*root);
            break;
        case 4:
            iterative_preorder_traversal(*root);
            break;
        case 5:
            iterative_postorder_traversal(*root);
            break;
        case 6:
            level_order_traversal(*root);
            break;
        case 7:
            iterative_destroy_tree(root);
            break;
        case 8:
            printf("Leaf nodes: %d\n", iterative_count_leaf(*root));
            break;
        case 9:
            printf("Non-leaf nodes: %d\n", iterative_count_non_leaf(*root));
            break;
        case 10:
            printf("Height of tree: %d\n", iterative_get_height(*root));
            break;
        case 11:
            exit(0);
        default:
            printf("Invalid choice\n");
    }
}
```

Output:

Menu:

```
● → Assignment 5 git:(main) ✗ gcc ./main.c ./logic.c ./stack.c ./queue.c  -Wall -o main
○ → Assignment 5 git:(main) ✗ ./main

 1. Insert a node
 2. Remove a node
 3. Inorder traversal
 4. Preorder traversal
 5. Postorder traversal
 6. Level order traversal
 7. Destroy tree
 8. Count leaf nodes
 9. Count non-leaf nodes
 10. Get height of tree
 11. Exit
 Enter your choice:
```

Commands.txt:

Lecture Assignments > Assignment 5 > commands.txt

```
1
january
1
february
1
march
1
april
1
may
1
june
1
july
1
august
1
september
1
october
1
november
1
december
3
4
5
6
8
9
10
2
june
3
7
11
```

result:

```
● → Assignment 5 git:(main) ✗ gcc ./main.c ./logic.c ./stack.c ./queue.c  -Wall -o main
● → Assignment 5 git:(main) ✗ cat ./commands.txt | ./main
april august december february january july june march may november october september
january february april august december march june july may september october november
december august april february july june november october september may march january
january february march april june may august july september december october november
Leaf nodes: 3
Non-leaf nodes: 9
Height of tree: 6
april august december february january july march may november october september
○ → Assignment 5 git:(main) ✗
```