**1singlylinkedlist\_logdiagnostics**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// Define the Node structure

struct Node {

int data;

struct Node\* next;

};

// Define the Stack structure with a top pointer

struct Stack {

struct Node\* top;

};

// Function to create a new stack

struct Stack\* createStack() {

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

stack->top = NULL;

return stack;

}

// Check if the stack is empty

int isEmpty(struct Stack\* stack) {

return stack->top == NULL;

}

// Push an element onto the stack

void push(struct Stack\* stack, int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

if (newNode == NULL) {

printf("Error: Memory allocation failed.\n");

return;

}

newNode->data = value;

newNode->next = stack->top;

stack->top = newNode;

printf("Pushed %d onto the stack.\n", value);

}

// Pop an element from the stack with error handling for an empty stack

int pop(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Error: Attempted to pop from an empty stack.\n");

return -1; // Returning -1 as an error code

}

struct Node\* temp = stack->top;

int popValue = temp->data;

stack->top = stack->top->next;

free(temp);

printf("Popped %d from the stack.\n", popValue);

return popValue;

}

// Peek at the top element of the stack without removing it

int peek(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Error: Attempted to peek on an empty stack.\n");

return -1; // Returning -1 as an error code

}

return stack->top->data;

}

// Free the entire stack

void freeStack(struct Stack\* stack) {

while (!isEmpty(stack)) {

pop(stack);

}

free(stack);

}

// Main function to demonstrate the stack operations

int main() {

struct Stack\* stack = createStack();

printf("Is stack empty? %s\n", isEmpty(stack) ? "Yes" : "No");

push(stack, 10);

push(stack, 20);

push(stack, 30);

printf("Top element is %d\n", peek(stack));

pop(stack);

printf("Top element after pop is %d\n", peek(stack));

pop(stack);

pop(stack);

// Attempt to pop from an empty stack

pop(stack);

freeStack(stack); // Free all allocated memory for the stack

return 0;

}

**2singlylinkedlist-fifo**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a queue node

struct Node {

int data;

int priority; // Optional priority field for prioritization

struct Node\* next;

};

// Define the structure for the queue

struct Queue {

struct Node\* front;

struct Node\* rear;

};

// Function to create a new queue

struct Queue\* createQueue() {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

queue->front = queue->rear = NULL;

return queue;

}

// Check if the queue is empty

int isEmpty(struct Queue\* queue) {

return queue->front == NULL;

}

// Enqueue operation: Add an element to the queue

void enqueue(struct Queue\* queue, int data, int priority) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->priority = priority;

newNode->next = NULL;

if (queue->rear == NULL) {

queue->front = queue->rear = newNode;

printf("Enqueued %d with priority %d.\n", data, priority);

return;

}

queue->rear->next = newNode;

queue->rear = newNode;

printf("Enqueued %d with priority %d.\n", data, priority);

}

// Dequeue operation: Remove an element from the queue with optional prioritization

int dequeue(struct Queue\* queue, int prioritize) {

if (isEmpty(queue)) {

printf("Error: Attempted to dequeue from an empty queue.\n");

return -1;

}

struct Node\* temp = queue->front;

struct Node\* prev = NULL;

struct Node\* selected = temp;

if (prioritize) {

// Prioritize based on custom criteria, for example, highest priority value

while (temp != NULL) {

if (temp->priority > selected->priority) {

selected = temp;

prev = prev ? prev->next : NULL;

}

temp = temp->next;

}

} else {

// FIFO behavior: dequeue the first element

prev = NULL;

}

// If selected node is not the front, find its predecessor

if (selected != queue->front) {

temp = queue->front;

while (temp != selected) {

prev = temp;

temp = temp->next;

}

}

// Remove the selected node from the queue

int dequeuedValue = selected->data;

if (prev != NULL) {

prev->next = selected->next;

} else {

queue->front = selected->next;

}

if (selected == queue->rear) {

queue->rear = prev;

}

free(selected);

printf("Dequeued %d.\n", dequeuedValue);

return dequeuedValue;

}

// Peek operation: View the front element without removing it

int peek(struct Queue\* queue) {

if (isEmpty(queue)) {

printf("Error: Attempted to peek on an empty queue.\n");

return -1;

}

return queue->front->data;

}

// Main function to demonstrate the queue operations

int main() {

struct Queue\* queue = createQueue();

// Enqueue elements with different priorities

enqueue(queue, 10, 1); // Data 10, priority 1

enqueue(queue, 20, 3); // Data 20, priority 3

enqueue(queue, 30, 2); // Data 30, priority 2

// Peek at the front element

printf("Front element is: %d\n", peek(queue));

// Dequeue with priority-based selection

dequeue(queue, 1); // Prioritize dequeue by priority level

// Dequeue with FIFO behavior

dequeue(queue, 0); // FIFO dequeue

// Dequeue remaining elements

dequeue(queue, 0);

// Attempt to dequeue from an empty queue

dequeue(queue, 0);

// Clean up and free memory

free(queue);

return 0;

}

**3bst\_rangesearch**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a BST node

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Function to create a new node

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

// Insert a node into the BST

struct Node\* insert(struct Node\* root, int data) {

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

root->left = insert(root->left, data);

} else if (data > root->data) {

root->right = insert(root->right, data);

} else {

printf("Duplicate value %d not allowed in BST\n", data); // Handle duplicates

}

return root;

}

// Search for a node in the BST

struct Node\* search(struct Node\* root, int data) {

if (root == NULL || root->data == data) {

return root;

}

if (data < root->data) {

return search(root->left, data);

}

return search(root->right, data);

}

// Find the minimum node in a given subtree

struct Node\* findMin(struct Node\* root) {

while (root->left != NULL) {

root = root->left;

}

return root;

}

// Delete a node from the BST

struct Node\* delete(struct Node\* root, int data) {

if (root == NULL) {

return root;

}

// Search for the node to be deleted

if (data < root->data) {

root->left = delete(root->left, data);

} else if (data > root->data) {

root->right = delete(root->right, data);

} else {

// Node to be deleted found

// Case 1: Node has no children (leaf node)

if (root->left == NULL && root->right == NULL) {

free(root);

return NULL;

}

// Case 2: Node has one child

else if (root->left == NULL) {

struct Node\* temp = root;

root = root->right;

free(temp);

} else if (root->right == NULL) {

struct Node\* temp = root;

root = root->left;

free(temp);

}

// Case 3: Node has two children

else {

struct Node\* temp = findMin(root->right); // Find the minimum in the right subtree

root->data = temp->data; // Replace root with the inorder successor

root->right = delete(root->right, temp->data); // Delete the inorder successor

}

}

return root;

}

// Function to print all nodes within a range [low, high]

void rangeSearch(struct Node\* root, int low, int high) {

if (root == NULL) {

return;

}

// If current node's data is within the range, print it

if (low <= root->data && root->data <= high) {

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

}

// Recursively search in the left and right subtrees

if (low < root->data) {

rangeSearch(root->left, low, high); // Left subtree may contain nodes within range

}

if (high > root->data) {

rangeSearch(root->right, low, high); // Right subtree may contain nodes within range

}

}

// Inorder traversal to print the BST

void inorder(struct Node\* root) {

if (root == NULL) {

return;

}

inorder(root->left);

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

inorder(root->right);

}

int main() {

struct Node\* root = NULL;

// Insert values into the BST

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 70);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 60);

root = insert(root, 80);

// Print the inorder traversal of the BST

printf("Inorder traversal: ");

inorder(root);

printf("\n");

// Search for a value

int searchValue = 40;

if (search(root, searchValue) != NULL) {

printf("Node with value %d found in the BST.\n", searchValue);

} else {

printf("Node with value %d not found in the BST.\n", searchValue);

}

// Delete a node

root = delete(root, 20); // Delete a node with no children

root = delete(root, 30); // Delete a node with one child

root = delete(root, 50); // Delete a node with two children

// Print the inorder traversal after deletions

printf("Inorder traversal after deletions: ");

inorder(root);

printf("\n");

// Range search

printf("Nodes with values in range [30, 70]: ");

rangeSearch(root, 30, 70);

printf("\n");

return 0;

}

**4avl\_balancefactor**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// Define the structure of a node in the AVL tree

struct Node {

int data;

struct Node\* left;

struct Node\* right;

int height;

};

// Function to create a new node

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->left = newNode->right = NULL;

newNode->height = 1;

return newNode;

}

// Function to get the height of a node

int getHeight(struct Node\* node) {

if (node == NULL)

return 0;

return node->height;

}

// Function to calculate the balance factor of a node

int getBalanceFactor(struct Node\* node) {

if (node == NULL)

return 0;

return getHeight(node->left) - getHeight(node->right);

}

// Right rotation for balancing the tree

struct Node\* rightRotate(struct Node\* y) {

struct Node\* x = y->left;

struct Node\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = 1 + (getHeight(y->left) > getHeight(y->right) ? getHeight(y->left) : getHeight(y->right));

x->height = 1 + (getHeight(x->left) > getHeight(x->right) ? getHeight(x->left) : getHeight(x->right));

// Return new root

return x;

}

// Left rotation for balancing the tree

struct Node\* leftRotate(struct Node\* x) {

struct Node\* y = x->right;

struct Node\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = 1 + (getHeight(x->left) > getHeight(x->right) ? getHeight(x->left) : getHeight(x->right));

y->height = 1 + (getHeight(y->left) > getHeight(y->right) ? getHeight(y->left) : getHeight(y->right));

// Return new root

return y;

}

// Function to insert a node into the AVL tree

struct Node\* insert(struct Node\* node, int data) {

// Step 1: Perform the normal BST insert

if (node == NULL)

return createNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else if (data > node->data)

node->right = insert(node->right, data);

else // Duplicate values are not allowed

return node;

// Step 2: Update height of the current node

node->height = 1 + (getHeight(node->left) > getHeight(node->right) ? getHeight(node->left) : getHeight(node->right));

// Step 3: Get the balance factor and perform necessary rotations

int balance = getBalanceFactor(node);

// Left Left Case

if (balance > 1 && data < node->left->data)

return rightRotate(node);

// Right Right Case

if (balance < -1 && data > node->right->data)

return leftRotate(node);

// Left Right Case

if (balance > 1 && data > node->left->data) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

// Right Left Case

if (balance < -1 && data < node->right->data) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

// Return the (unchanged) node pointer

return node;

}

// Function to find the node with the minimum value

struct Node\* findMin(struct Node\* node) {

struct Node\* current = node;

while (current->left != NULL)

current = current->left;

return current;

}

// Function to delete a node from the AVL tree

struct Node\* delete(struct Node\* root, int data) {

if (root == NULL)

return root;

// Step 1: Perform the normal BST delete

if (data < root->data)

root->left = delete(root->left, data);

else if (data > root->data)

root->right = delete(root->right, data);

else {

// Node to be deleted found

// Case 1: Node has no children (leaf node)

if (root->left == NULL && root->right == NULL) {

free(root);

return NULL;

}

// Case 2: Node has one child

else if (root->left == NULL) {

struct Node\* temp = root;

root = root->right;

free(temp);

} else if (root->right == NULL) {

struct Node\* temp = root;

root = root->left;

free(temp);

}

// Case 3: Node has two children

else {

struct Node\* temp = findMin(root->right); // Find the inorder successor

root->data = temp->data;

root->right = delete(root->right, temp->data); // Delete the inorder successor

}

}

// Step 2: Update height of the current node

if (root == NULL)

return root;

root->height = 1 + (getHeight(root->left) > getHeight(root->right) ? getHeight(root->left) : getHeight(root->right));

// Step 3: Get the balance factor and perform necessary rotations

int balance = getBalanceFactor(root);

// Left Left Case

if (balance > 1 && getBalanceFactor(root->left) >= 0)

return rightRotate(root);

// Right Right Case

if (balance < -1 && getBalanceFactor(root->right) <= 0)

return leftRotate(root);

// Left Right Case

if (balance > 1 && getBalanceFactor(root->left) < 0) {

root->left = leftRotate(root->left);

return rightRotate(root);

}

// Right Left Case

if (balance < -1 && getBalanceFactor(root->right) > 0) {

root->right = rightRotate(root->right);

return leftRotate(root);

}

// Return the (unchanged) node pointer

return root;

}

// Function to search for a node in the AVL tree

struct Node\* search(struct Node\* root, int data) {

if (root == NULL || root->data == data)

return root;

if (data < root->data)

return search(root->left, data);

return search(root->right, data);

}

// Function to print the tree in order

void inorder(struct Node\* root) {

if (root == NULL)

return;

inorder(root->left);

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

inorder(root->right);

}

// Main function

int main() {

struct Node\* root = NULL;

// Insert nodes

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 25);

root = insert(root, 5);

root = insert(root, 15);

printf("Inorder traversal of AVL tree: ");

inorder(root);

printf("\n");

// Search for a value

int value = 25;

struct Node\* result = search(root, value);

if (result)

printf("Node with value %d found.\n", value);

else

printf("Node with value %d not found.\n", value);

// Delete a node

root = delete(root, 20);

printf("Inorder traversal after deletion: ");

inorder(root);

printf("\n");

// Insert more nodes

root = insert(root, 50);

root = insert(root, 40);

printf("Inorder traversal after additional insertions: ");

inorder(root);

printf("\n");

return 0;

}

**5minmaxheap\_validafterupdate**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// Define the structure of the Min-Max Heap

struct MinMaxHeap {

int\* arr;

int size;

int capacity;

};

// Function to create a new Min-Max Heap

struct MinMaxHeap\* createMinMaxHeap(int capacity) {

struct MinMaxHeap\* heap = (struct MinMaxHeap\*)malloc(sizeof(struct MinMaxHeap));

heap->arr = (int\*)malloc(capacity \* sizeof(int));

heap->size = 0;

heap->capacity = capacity;

return heap;

}

// Function to swap two elements in the array

void swap(struct MinMaxHeap\* heap, int i, int j) {

int temp = heap->arr[i];

heap->arr[i] = heap->arr[j];

heap->arr[j] = temp;

}

// Function to get the index of the parent

int parent(int i) {

return (i - 1) / 2;

}

// Function to get the index of the left child

int leftChild(int i) {

return 2 \* i + 1;

}

// Function to get the index of the right child

int rightChild(int i) {

return 2 \* i + 2;

}

// Function to check if a node is at an even level (min-level) or odd level (max-level)

int isMinLevel(int i) {

return (i % 2 == 0);

}

// Function to heapify down at a specific index to maintain Min-Max Heap properties

void heapify(struct MinMaxHeap\* heap, int i) {

if (i >= heap->size) return;

int left = leftChild(i);

int right = rightChild(i);

int smallest = i;

int largest = i;

if (isMinLevel(i)) { // Min level, ensure heap property with children

if (left < heap->size && heap->arr[left] < heap->arr[smallest])

smallest = left;

if (right < heap->size && heap->arr[right] < heap->arr[smallest])

smallest = right;

if (smallest != i) {

swap(heap, i, smallest);

heapify(heap, smallest); // Recursive call

}

} else { // Max level, ensure heap property with children

if (left < heap->size && heap->arr[left] > heap->arr[largest])

largest = left;

if (right < heap->size && heap->arr[right] > heap->arr[largest])

largest = right;

if (largest != i) {

swap(heap, i, largest);

heapify(heap, largest); // Recursive call

}

}

}

// Function to insert a new element in the Min-Max Heap

void insert(struct MinMaxHeap\* heap, int value) {

if (heap->size == heap->capacity) {

printf("Heap is full, cannot insert\n");

return;

}

heap->arr[heap->size] = value;

int i = heap->size;

heap->size++;

// Fix the Min-Max Heap properties by bubbling up

while (i != 0 && heap->arr[parent(i)] > heap->arr[i]) {

swap(heap, i, parent(i));

i = parent(i);

}

heapify(heap, i); // Heapify the tree starting from the inserted element

}

// Function to delete the minimum element (root element)

int deleteMin(struct MinMaxHeap\* heap) {

if (heap->size == 0) {

printf("Heap is empty, cannot deleteMin\n");

return -1;

}

int min = heap->arr[0];

heap->arr[0] = heap->arr[heap->size - 1];

heap->size--;

heapify(heap, 0); // Restore the heap properties

return min;

}

// Function to delete the maximum element

int deleteMax(struct MinMaxHeap\* heap) {

if (heap->size == 0) {

printf("Heap is empty, cannot deleteMax\n");

return -1;

}

int maxIndex = 0;

for (int i = 1; i < heap->size; i++) {

if (!isMinLevel(i) && heap->arr[i] > heap->arr[maxIndex]) {

maxIndex = i;

}

}

int max = heap->arr[maxIndex];

heap->arr[maxIndex] = heap->arr[heap->size - 1];

heap->size--;

heapify(heap, maxIndex); // Restore the heap properties

return max;

}

// Function to get the minimum element (root element)

int getMin(struct MinMaxHeap\* heap) {

if (heap->size == 0) {

printf("Heap is empty, no min element\n");

return -1;

}

return heap->arr[0];

}

// Function to get the maximum element

int getMax(struct MinMaxHeap\* heap) {

if (heap->size == 0) {

printf("Heap is empty, no max element\n");

return -1;

}

int max = heap->arr[0];

for (int i = 1; i < heap->size; i++) {

if (!isMinLevel(i) && heap->arr[i] > max) {

max = heap->arr[i];

}

}

return max;

}

// Function to update the priority (key value) of an element in the Min-Max Heap

void updatePriority(struct MinMaxHeap\* heap, int index, int newValue) {

if (index < 0 || index >= heap->size) {

printf("Invalid index\n");

return;

}

int oldValue = heap->arr[index];

heap->arr[index] = newValue;

// Check if the heap is violated and fix the heap

if (newValue > oldValue) {

// If new value is larger, move it down

heapify(heap, index);

} else {

// If new value is smaller, move it up

while (index != 0 && heap->arr[parent(index)] > heap->arr[index]) {

swap(heap, index, parent(index));

index = parent(index);

}

heapify(heap, index);

}

}

// Main function to test the Min-Max Heap operations

int main() {

struct MinMaxHeap\* heap = createMinMaxHeap(10);

insert(heap, 3);

insert(heap, 2);

insert(heap, 8);

insert(heap, 5);

insert(heap, 1);

printf("Min: %d\n", getMin(heap)); // Output: 1

printf("Max: %d\n", getMax(heap)); // Output: 8

printf("Deleted Min: %d\n", deleteMin(heap)); // Output: 1

printf("Deleted Max: %d\n", deleteMax(heap)); // Output: 8

insert(heap, 7);

insert(heap, 6);

printf("Min: %d\n", getMin(heap)); // Output: 2

printf("Max: %d\n", getMax(heap)); // Output: 7

// Update the priority of an element (for example, index 1)

updatePriority(heap, 1, 9);

printf("Updated Max: %d\n", getMax(heap)); // Output: 9

return 0;

}

**6RBT\_minmaxlevelorder**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// Define color constants for Red-Black Tree nodes

#define RED 0

#define BLACK 1

// Define structure for Red-Black Tree Node

struct RBTreeNode {

int data;

int color; // RED or BLACK

struct RBTreeNode \*left, \*right, \*parent;

};

// Define structure for Red-Black Tree

struct RBTree {

struct RBTreeNode \*root;

struct RBTreeNode \*TNULL; // Sentinel node for NIL

};

// Function to create a new node with a given data value

struct RBTreeNode\* createNode(int data) {

struct RBTreeNode\* node = (struct RBTreeNode\*)malloc(sizeof(struct RBTreeNode));

node->data = data;

node->left = node->right = node->parent = NULL;

node->color = RED; // New nodes are initially red

return node;

}

// Function to initialize the Red-Black Tree

struct RBTree\* initializeRBTree() {

struct RBTree\* tree = (struct RBTree\*)malloc(sizeof(struct RBTree));

tree->TNULL = createNode(0); // Create TNULL node (sentinel)

tree->TNULL->color = BLACK;

tree->root = tree->TNULL; // Initially the tree is empty

return tree;

}

// Left rotate function to maintain Red-Black Tree balance

void leftRotate(struct RBTree\* tree, struct RBTreeNode\* x) {

struct RBTreeNode\* y = x->right;

x->right = y->left;

if (y->left != tree->TNULL) {

y->left->parent = x;

}

y->parent = x->parent;

if (x->parent == tree->TNULL) {

tree->root = y;

} else if (x == x->parent->left) {

x->parent->left = y;

} else {

x->parent->right = y;

}

y->left = x;

x->parent = y;

}

// Right rotate function to maintain Red-Black Tree balance

void rightRotate(struct RBTree\* tree, struct RBTreeNode\* x) {

struct RBTreeNode\* y = x->left;

x->left = y->right;

if (y->right != tree->TNULL) {

y->right->parent = x;

}

y->parent = x->parent;

if (x->parent == tree->TNULL) {

tree->root = y;

} else if (x == x->parent->right) {

x->parent->right = y;

} else {

x->parent->left = y;

}

y->right = x;

x->parent = y;

}

// Fix the Red-Black Tree properties after insertion

void fixInsert(struct RBTree\* tree, struct RBTreeNode\* k) {

struct RBTreeNode\* u;

while (k->parent->color == RED) {

if (k->parent == k->parent->parent->right) {

u = k->parent->parent->left;

if (u->color == RED) {

u->color = BLACK;

k->parent->color = BLACK;

k->parent->parent->color = RED;

k = k->parent->parent;

} else {

if (k == k->parent->left) {

k = k->parent;

rightRotate(tree, k);

}

k->parent->color = BLACK;

k->parent->parent->color = RED;

leftRotate(tree, k->parent->parent);

}

} else {

u = k->parent->parent->right;

if (u->color == RED) {

u->color = BLACK;

k->parent->color = BLACK;

k->parent->parent->color = RED;

k = k->parent->parent;

} else {

if (k == k->parent->right) {

k = k->parent;

leftRotate(tree, k);

}

k->parent->color = BLACK;

k->parent->parent->color = RED;

rightRotate(tree, k->parent->parent);

}

}

if (k == tree->root) {

break;

}

}

tree->root->color = BLACK;

}

// Insert a node into the Red-Black Tree

void insert(struct RBTree\* tree, int data) {

struct RBTreeNode\* node = createNode(data);

struct RBTreeNode\* y = tree->TNULL;

struct RBTreeNode\* x = tree->root;

while (x != tree->TNULL) {

y = x;

if (node->data < x->data) {

x = x->left;

} else {

x = x->right;

}

}

node->parent = y;

if (y == tree->TNULL) {

tree->root = node;

} else if (node->data < y->data) {

y->left = node;

} else {

y->right = node;

}

node->left = node->right = tree->TNULL;

node->color = RED;

fixInsert(tree, node);

}

// Search for a node with the given value

struct RBTreeNode\* search(struct RBTree\* tree, int key) {

struct RBTreeNode\* node = tree->root;

while (node != tree->TNULL) {

if (key == node->data) {

return node;

}

if (key < node->data) {

node = node->left;

} else {

node = node->right;

}

}

return tree->TNULL; // If not found

}

// Function to find the minimum value node in the Red-Black Tree

struct RBTreeNode\* minimum(struct RBTree\* tree, struct RBTreeNode\* node) {

while (node->left != tree->TNULL) {

node = node->left;

}

return node;

}

// Function to find the maximum value node in the Red-Black Tree

struct RBTreeNode\* maximum(struct RBTree\* tree, struct RBTreeNode\* node) {

while (node->right != tree->TNULL) {

node = node->right;

}

return node;

}

// Level-order traversal of the Red-Black Tree (Breadth-first traversal)

void levelOrderTraversal(struct RBTree\* tree) {

if (tree->root == tree->TNULL) {

printf("Tree is empty.\n");

return;

}

struct RBTreeNode\* queue[100];

int front = 0, rear = 0;

queue[rear++] = tree->root;

while (front != rear) {

struct RBTreeNode\* currentNode = queue[front++];

printf("%d(%s) ", currentNode->data, currentNode->color == RED ? "R" : "B");

if (currentNode->left != tree->TNULL) {

queue[rear++] = currentNode->left;

}

if (currentNode->right != tree->TNULL) {

queue[rear++] = currentNode->right;

}

}

printf("\n");

}

// Main function to test the Red-Black Tree operations

int main() {

struct RBTree\* tree = initializeRBTree();

// Insert elements into the Red-Black Tree

int values[] = {20, 15, 25, 10, 5, 30, 35};

for (int i = 0; i < 7; i++) {

insert(tree, values[i]);

}

// Search for a value

struct RBTreeNode\* foundNode = search(tree, 15);

if (foundNode != tree->TNULL) {

printf("Found: %d\n", foundNode->data);

} else {

printf("Value not found in the tree.\n");

}

// Find the minimum and maximum values

struct RBTreeNode\* minNode = minimum(tree, tree->root);

struct RBTreeNode\* maxNode = maximum(tree, tree->root);

printf("Minimum: %d\n", minNode->data);

printf("Maximum: %d\n", maxNode->data);

// Perform a level-order traversal

printf("Level Order Traversal: \n");

levelOrderTraversal(tree);

return 0;

}

**7splaytree\_rangesearch**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the Splay Tree

struct SplayTreeNode {

int data;

struct SplayTreeNode \*left, \*right;

};

// Function to create a new node

struct SplayTreeNode\* createNode(int data) {

struct SplayTreeNode\* newNode = (struct SplayTreeNode\*)malloc(sizeof(struct SplayTreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

// Function for right rotation (zig)

struct SplayTreeNode\* rightRotate(struct SplayTreeNode\* root) {

struct SplayTreeNode\* leftChild = root->left;

root->left = leftChild->right;

leftChild->right = root;

return leftChild;

}

// Function for left rotation (zag)

struct SplayTreeNode\* leftRotate(struct SplayTreeNode\* root) {

struct SplayTreeNode\* rightChild = root->right;

root->right = rightChild->left;

rightChild->left = root;

return rightChild;

}

// Splay function to move the accessed node to the root

struct SplayTreeNode\* splay(struct SplayTreeNode\* root, int key) {

if (root == NULL || root->data == key) {

return root;

}

// Key is in the left subtree

if (key < root->data) {

// Key is not present in the tree

if (root->left == NULL) return root;

// Zig-Zig (Left Left)

if (key < root->left->data) {

root = rightRotate(root);

if (root->data == key) return root;

}

// Zig-Zag (Left Right)

else if (key > root->left->data) {

root->left = leftRotate(root->left);

root = rightRotate(root);

if (root->data == key) return root;

}

}

// Key is in the right subtree

else {

// Key is not present in the tree

if (root->right == NULL) return root;

// Zag-Zag (Right Right)

if (key > root->right->data) {

root = leftRotate(root);

if (root->data == key) return root;

}

// Zag-Zig (Right Left)

else if (key < root->right->data) {

root->right = rightRotate(root->right);

root = leftRotate(root);

if (root->data == key) return root;

}

}

return root;

}

// Insert a node into the Splay Tree

struct SplayTreeNode\* insert(struct SplayTreeNode\* root, int key) {

if (root == NULL) return createNode(key);

root = splay(root, key);

// If the key is already present, return the root (no duplicates in Splay Tree)

if (root->data == key) return root;

struct SplayTreeNode\* newNode = createNode(key);

if (key < root->data) {

newNode->right = root;

newNode->left = root->left;

root->left = NULL;

} else {

newNode->left = root;

newNode->right = root->right;

root->right = NULL;

}

return newNode;

}

// Search for a node with the given key

struct SplayTreeNode\* search(struct SplayTreeNode\* root, int key) {

return splay(root, key);

}

// Function to find the minimum value node

struct SplayTreeNode\* minimum(struct SplayTreeNode\* root) {

while (root && root->left != NULL) {

root = root->left;

}

return root;

}

// Function to delete the node with the given key

struct SplayTreeNode\* delete(struct SplayTreeNode\* root, int key) {

if (root == NULL) return NULL;

root = splay(root, key);

// If key is not found, return root

if (root->data != key) return root;

// Key found, now delete the root node

if (root->left == NULL) {

struct SplayTreeNode\* temp = root;

root = root->right;

free(temp);

} else {

struct SplayTreeNode\* temp = root;

root = splay(root->left, key);

root->right = temp->right;

free(temp);

}

return root;

}

// Function to perform in-order traversal of the tree

void inorder(struct SplayTreeNode\* root) {

if (root == NULL) return;

inorder(root->left);

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

inorder(root->right);

}

// Function to perform range search, finding all nodes within the range [low, high]

void rangeSearch(struct SplayTreeNode\* root, int low, int high) {

if (root == NULL) return;

if (root->data >= low) {

rangeSearch(root->left, low, high);

}

if (root->data >= low && root->data <= high) {

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

}

if (root->data <= high) {

rangeSearch(root->right, low, high);

}

}

// Function to display the tree structure (in-order traversal)

void displayTree(struct SplayTreeNode\* root) {

if (root != NULL) {

inorder(root);

printf("\n");

} else {

printf("Tree is empty.\n");

}

}

int main() {

struct SplayTreeNode\* root = NULL;

// Inserting elements into the Splay Tree

int elements[] = {40, 20, 60, 10, 30, 50, 70};

for (int i = 0; i < 7; i++) {

root = insert(root, elements[i]);

}

printf("Tree structure after insertions:\n");

displayTree(root);

// Searching for an element (30) and splaying it to the root

printf("\nSearching for 30:\n");

root = search(root, 30);

displayTree(root);

// Deleting node with value 40 and splaying the necessary nodes

printf("\nDeleting node 40:\n");

root = delete(root, 40);

displayTree(root);

// Range Search for values between 15 and 65

printf("\nRange search for values between 15 and 65:\n");

rangeSearch(root, 15, 65);

printf("\n");

return 0;

}

**8binomialheap\_union**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// Structure for a binomial heap node

struct BinomialHeapNode {

int data;

int degree;

struct BinomialHeapNode \*child;

struct BinomialHeapNode \*sibling;

};

// Function to create a new binomial heap node

struct BinomialHeapNode\* newNode(int data) {

struct BinomialHeapNode\* node = (struct BinomialHeapNode\*)malloc(sizeof(struct BinomialHeapNode));

node->data = data;

node->degree = 0;

node->child = NULL;

node->sibling = NULL;

return node;

}

// Function to merge two binomial trees

struct BinomialHeapNode\* mergeBinomialTrees(struct BinomialHeapNode\* t1, struct BinomialHeapNode\* t2) {

if (t1->data > t2->data) {

struct BinomialHeapNode\* temp = t1;

t1 = t2;

t2 = temp;

}

t2->sibling = t1->child;

t1->child = t2;

t1->degree++;

return t1;

}

// Function to merge two binomial heaps

struct BinomialHeapNode\* unionHeaps(struct BinomialHeapNode\* h1, struct BinomialHeapNode\* h2) {

struct BinomialHeapNode\* newHeap = NULL;

struct BinomialHeapNode \*prev = NULL, \*curr = NULL, \*next = NULL;

// Merge the two heaps by degree

while (h1 != NULL && h2 != NULL) {

if (h1->degree < h2->degree) {

if (newHeap == NULL) newHeap = h1;

else prev->sibling = h1;

prev = h1;

h1 = h1->sibling;

} else if (h1->degree > h2->degree) {

if (newHeap == NULL) newHeap = h2;

else prev->sibling = h2;

prev = h2;

h2 = h2->sibling;

} else {

next = h1->sibling;

h1 = mergeBinomialTrees(h1, h2);

if (newHeap == NULL) newHeap = h1;

else prev->sibling = h1;

prev = h1;

h1 = next;

h2 = h2->sibling;

}

}

if (h1 != NULL) prev->sibling = h1;

else if (h2 != NULL) prev->sibling = h2;

return newHeap;

}

// Function to insert a new key into a binomial heap

struct BinomialHeapNode\* insert(struct BinomialHeapNode\* heap, int data) {

struct BinomialHeapNode\* newNode = newNode(data);

return unionHeaps(heap, newNode);

}

// Function to perform a binary search to find a node with the given key

int search(struct BinomialHeapNode\* heap, int key) {

struct BinomialHeapNode\* curr = heap;

while (curr != NULL) {

if (curr->data == key)

return 1; // Key found

if (search(curr->child, key)) // Search in the child subtree

return 1;

curr = curr->sibling;

}

return 0; // Key not found

}

// Function to remove the minimum element (root of the heap)

struct BinomialHeapNode\* deleteMin(struct BinomialHeapNode\* heap) {

if (heap == NULL) return NULL;

struct BinomialHeapNode \*minNode = heap, \*minPrev = NULL, \*prev = NULL, \*curr = heap;

// Find the minimum node

while (curr != NULL) {

if (curr->data < minNode->data) {

minNode = curr;

minPrev = prev;

}

prev = curr;

curr = curr->sibling;

}

if (minPrev == NULL)

heap = minNode->sibling;

else

minPrev->sibling = minNode->sibling;

struct BinomialHeapNode\* child = minNode->child;

struct BinomialHeapNode\* nextSibling;

while (child != NULL) {

nextSibling = child->sibling;

child->sibling = heap;

heap = child;

child = nextSibling;

}

free(minNode);

return heap;

}

// Function to display the binomial heap (in-order traversal of binomial trees)

void displayBinomialHeap(struct BinomialHeapNode\* heap) {

if (heap == NULL) {

printf("Heap is empty.\n");

return;

}

struct BinomialHeapNode\* curr = heap;

while (curr != NULL) {

printf("Degree: %d, Data: %d\n", curr->degree, curr->data);

struct BinomialHeapNode\* child = curr->child;

while (child != NULL) {

printf("\tChild: %d\n", child->data);

child = child->sibling;

}

curr = curr->sibling;

}

}

// Main function to test the binomial heap operations

int main() {

struct BinomialHeapNode\* heap1 = NULL;

struct BinomialHeapNode\* heap2 = NULL;

// Insert elements into the first binomial heap

int elements1[] = {23, 17, 8, 45, 91, 34, 29, 76, 58};

for (int i = 0; i < 9; i++) {

heap1 = insert(heap1, elements1[i]);

}

printf("Heap 1 structure after insertions:\n");

displayBinomialHeap(heap1);

// Insert elements into the second binomial heap

int elements2[] = {12, 38, 54, 67, 82};

for (int i = 0; i < 5; i++) {

heap2 = insert(heap2, elements2[i]);

}

printf("\nHeap 2 structure after insertions:\n");

displayBinomialHeap(heap2);

// Perform a union operation to merge the two heaps

struct BinomialHeapNode\* mergedHeap = unionHeaps(heap1, heap2);

printf("\nFinal structure of the merged heap:\n");

displayBinomialHeap(mergedHeap);

// Perform search operation for a specific key

int searchKey = 45;

if (search(mergedHeap, searchKey)) {

printf("\nElement %d found in the heap.\n", searchKey);

} else {

printf("\nElement %d not found in the heap.\n", searchKey);

}

// Delete the minimum element

mergedHeap = deleteMin(mergedHeap);

printf("\nHeap after deleting the minimum element:\n");

displayBinomialHeap(mergedHeap);

return 0;

}

**9leftistheap\_mergeandremoveminele**

**Code:;**

class LeftistHeapNode:

def \_\_init\_\_(self, data):

self.data = data

self.left = None

self.right = None

self.rank = 0 # Null path length (NPL)

class LeftistHeap:

def \_\_init\_\_(self):

self.root = None

# Helper function to merge two leftist heaps

def merge(self, other):

if not other.root:

return

self.root = self.\_merge(self.root, other.root)

other.root = None

# Merge two roots of leftist heaps

def \_merge(self, h1, h2):

if not h1:

return h2

if not h2:

return h1

if h1.data > h2.data:

h1, h2 = h2, h1

# Merge the right subtree of h1 with h2

h1.right = self.\_merge(h1.right, h2)

# Ensure leftist property: the left subtree should be taller

if not h1.left or h1.left.rank < h1.right.rank:

h1.left, h1.right = h1.right, h1.left

# Update the rank of the current node

if not h1.right:

h1.rank = 0

else:

h1.rank = h1.right.rank + 1

return h1

# Insert a new element into the leftist heap

def insert(self, data):

new\_node = LeftistHeapNode(data)

self.root = self.\_merge(self.root, new\_node)

# Find the minimum element in the heap

def find\_min(self):

if self.root:

return self.root.data

return None

# Remove the minimum element from the heap

def delete\_min(self):

if not self.root:

return None

min\_data = self.root.data

self.root = self.\_merge(self.root.left, self.root.right)

return min\_data

# Display the heap structure (in-order traversal)

def display(self):

nodes = []

self.\_inorder\_traversal(self.root, nodes)

return nodes

# In-order traversal to collect node data

def \_inorder\_traversal(self, node, nodes):

if node:

self.\_inorder\_traversal(node.left, nodes)

nodes.append(node.data)

self.\_inorder\_traversal(node.right, nodes)

# Testing the Leftist Heap operations

if \_\_name\_\_ == "\_\_main\_\_":

# Initialize two heaps

heap1 = LeftistHeap()

heap2 = LeftistHeap()

# Insert elements into the first heap

for elem in [12, 7, 25, 15, 20, 10, 5]:

heap1.insert(elem)

print("Heap 1 after insertions:", heap1.display())

# Insert elements into the second heap

for elem in [30, 17, 8, 21]:

heap2.insert(elem)

print("Heap 2 after insertions:", heap2.display())

# Merge heap1 and heap2

heap1.merge(heap2)

print("Heap 1 after merging with Heap 2:", heap1.display())

# Find the minimum element in the merged heap

print("Minimum element in the merged heap:", heap1.find\_min())

# Remove the minimum element and display the updated heap

removed\_min = heap1.delete\_min()

print("Removed minimum element:", removed\_min)

print("Heap 1 after deleting the minimum element:", heap1.display())