DAALab4

Yugendra Senthil Kumar

CH.SC.U4CSE24252

CSE-C

1.Merge Sort

```c
#include<stdio.h>
#include<stdlib.h>
void merge_sorted_arrays(int arr[],int l,int m,int r){
    int left_length = m-l+1;
    int right_length = r-m;

    int temp_left[left_length];
    int temp_right[right_length];
    for(int i = 0;i<left_length;++i){
        temp_left[i] = arr[i+l];
    }
    for(int i = 0;i<right_length;++i){
        temp_right[i] = arr[m+1+i];
    }
    int i =0 ;
    int j = 0;
    for(int k=l;k<r;++k){
        if((i<left_length) && (j<right_length || temp_left[i]<temp_right[j])){
            arr[k] = temp_left[i];
            i++;
        }else{
            arr[k] = temp_right[j];
            j++;
        }
    }
}
void merge_sort_recursion(int arr[],int l, int r){

    if(l<r){
        int m = l+(r-l)/2;//getting middle
        merge_sort_recursion(arr,l,m);
        merge_sort_recursion(arr,m+1,r);
        merge_sorted_arrays(arr,l,m,r);
    }
```

```c
}
void merge_sort(int arr[],int size){
    merge_sort_recursion(arr,0,size-1);
}
int main(){
    int arr[8] = {4,1,5,1,3,52,42,12};
    merge_sort(arr,8);
    for(int i =0;i<8;++i){
        printf("%d ",arr[i]);
    }
}

/*
Time Complexity: O(nlog(n))
Space Complexity: O(n)

This sorting algorithm works on a divide and conquer strategy, by dividing up the
arrays(in this
program it is done with recursion). It then merges sorted arrays(the most recent
divided array
from the recursion call stack will be merged using merge_sorted_arrays). This
works on two principles
:merging sorted arrays are efficient and an array of one element is already
technically sorted.

*/
```

```
yugen@fountain:/mnt/c/Users/yugen/C files/Lab4$ ./merge
4 1 5 1 3 52 42 12 yugen@fountain:/mnt/c/Users/yugen/C files
```

2.Quick sort

```c
#include<stdio.h>
void swap(int* x, int* y);
void quicksort(int array[],int length);
void quicksort_recursion(int array[],int l, int r);
int partition(int array[],int l, int r);

int main(){
    int a[] = {10,11,23,44,8,15,3,9,12,45,56,45,45};
    int length = sizeof(a)/sizeof(int);
    quicksort(a,length);
    for(int i = 0;i<length;++i){
        printf("%d,",a[i]);
    }
```

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

void swap(int* x, int* y){
    int c = *x;
    *x = *y;
    *y = c;
}
void quicksort(int array[],int length){
    quicksort_recursion(array,0,length-1);
}
void quicksort_recursion(int array[],int l,int r){
    if(l<r){
        int pivot_index = partition(array,l,r);
        quicksort_recursion(array,l,pivot_index-1);
        quicksort_recursion(array,pivot_index+1,r);
    }
}
int partition(int array[],int l,int r){
    int pivot_value = array[r];
    int i = l;
    for(int j = l;j<r;++j){
        if(array[j]<=pivot_value){
            swap(&array[i],&array[j]);
            i++;
        }
    }
    swap(&array[i],&array[r]);
    return i;
}
/*
Time Complexity: O(nlogn), worst is O(n^2)
Space Complexity:O(logn), worst is O(n)


This sorting algorithm works on a divide and conquer strategy, by selecting a
pivot element
and partitioning the array around it (in this program it is done with recursion).
It then recursively
sorts the sub-arrays on either side of the pivot.
This works on two principles: partitioning places the pivot in its correct sorted
position,
and an array of one element is already technically sorted(reaching base cases).
```

```
*/
 4 1 5 1 3 52 42 12 yugen@fountain:/mnt/c/Users/yugen/C files/Lab4$ gcc quicksort.c -o quick
 yugen@fountain:/mnt/c/Users/yugen/C files/Lab4$ ./quick
 3,8,9,10,11,12,15,23,44,45,45,45,56,
```

3.Binary Search Tree

```c
#include<stdio.h>
#include<stdlib.h>
typedef struct Node{
    int data;
    struct Node* left;
    struct Node* right;
}Node;
Node* createNode(int newdata){
    Node* newnode = (Node*)malloc(sizeof(Node));
    newnode->data = newdata;
    newnode->left = NULL;
    newnode->right = NULL;
    return newnode;
}
void insert(Node** root, int data){
    if(*root == NULL){
        *root = createNode(data);
        return;
    }
    if(data < (*root)->data){
        insert(&(*root)->left, data);
    } else if(data > (*root)->data){
        insert(&(*root)->right, data);
    }
}
Node* findMin(Node* root) {
    while(root && root->left != NULL) {
        root = root->left;
    }
    return root;
}
Node* del(Node* root, int val){
    if(root == NULL){
        printf("Tree empty or value not found\n");
        return root;
    }
    if(val < root->data) {
        root->left = del(root->left, val);
    } else if(val > root->data) {
```

```
        root->right = del(root->right, val);
    } else {
        if(root->left == NULL) {
            Node* temp = root->right;
            free(root);
            return temp;
        } else if(root->right == NULL) {
            Node* temp = root->left;
            free(root);
            return temp;
        }
        Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = del(root->right, temp->data);
    }
    return root;
}
Node* search(Node* root, int val){
    if(root == NULL || root->data == val){
        return root;
    }
    if(val < root->data){
        return search(root->left, val);
    } else {
        return search(root->right, val);
    }
}
void inorderTraversal(Node* root) {
    if (root == NULL) {
        return;
    }
    inorderTraversal(root->left);
    printf("%d ", root->data);
    inorderTraversal(root->right);
}
void preorderTraversal(Node* root) {
    if (root == NULL) {
        return;
    }
    printf("%d ", root->data);
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}
void postorderTraversal(Node* root) {
```

```
    if (root == NULL) {
        return;
    }
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    printf("%d ", root->data);
}
int main() {
    Node* root = NULL;
    insert(&root, 50);
    insert(&root, 30);
    insert(&root, 70);
    insert(&root, 20);
    insert(&root, 40);
    insert(&root, 60);
    insert(&root, 80);
    printf("Inorder traversal (sorted): ");
    inorderTraversal(root);
    printf("\n");
    printf("Preorder traversal: ");
    preorderTraversal(root);
    printf("\n");
    printf("Postorder traversal: ");
    postorderTraversal(root);
    printf("\n");
    int deleteValue = 20;
    printf("After deletion of %d: ", deleteValue);
    root = del(root, deleteValue);
    inorderTraversal(root);
    printf("\n");
    int insertValue = 25;
    insert(&root, insertValue);
    printf("After insertion of %d: ", insertValue);
    inorderTraversal(root);
    printf("\n");
    int target = 25;
    Node* searchResult = search(root, target);
    if (searchResult != NULL) {
        printf("Node %d found in the BST.\n", target);
    } else {
        printf("Node %d not found in the BST.\n", target);
    }
    target = 100;
    searchResult = search(root, target);
```

```
    if (searchResult != NULL) {
        printf("Node %d found in the BST.\n", target);
    } else {
        printf("Node %d not found in the BST.\n", target);
    }
    return 0;
}

/*
Time Complexity:O(logn), worst O(n)
Space Complexity:O(logn), worst O(n)
This program implements a Binary Search Tree (BST) using a recursive approach.
The tree maintains the BST property where values smaller than a node are stored
in the left subtree
and larger values in the right subtree.
Insertion and searching work by recursively comparing the given value with the
current node and
moving left or right accordingly until the correct position or value is found.
Deletion is handled by recursively locating the target node and managing three
cases: node with no
children, node with one child, and node with two children (replaced using its
inorder successor).
Traversals operate on the principle of recursion:
inorder traversal visits left->root->right(producing sorted output),
preorder traversal visits root->left->right
postorder is left->right-root
The recursive process continues until a null pointer is reached, which acts as
the base case.
*/
```

```
yugen@fountain:/mnt/c/Users/yugen/C files/Lab4$ g++ binarysearchtree.cpp -o bst
yugen@fountain:/mnt/c/Users/yugen/C files/Lab4$ ./bst
Inorder traversal (sorted): 20 30 40 50 60 70 80
Preorder traversal: 50 30 20 40 70 60 80
Postorder traversal: 20 40 30 60 80 70 50
After deletion of 20: 30 40 50 60 70 80
After insertion of 25: 25 30 40 50 60 70 80
Node 25 found in the BST.
Node 100 not found in the BST.
```