**LEETCODE solutions in C**

1)Two Sum67

2)Best time to buy and sell stock

3)Find duplicates from sorted array

4)Jump Games

5)Rotate Array

6)Find Unique

7)Find the 1st positive missing number

8)Trapping Rain water

9)Stack program in C

10)Representation of stack in array

11)Representation of stack in linked list

12)Chocolate distribution

13)Next Greater Elememt(NGE)

14)Find the 1st circular tour that visits all petrol pumps

15)Delete last occurrence of an item from the linked list

16)Remove Nth node from end of a linked list

17)Reverse a linked list

18)Count ways to reach Nth stairs

19)Coin change

20)0/1 Knapsack Problem

21)Largest Increasing Subsequence

22)Longest Common Subsequence

23)Egg dropping puzzle

24)Matrix chain multiplication

25)Subset sum problem using recursion

26)pairwise Swap modes of a given linked list

27)Count for nodes in a circular linked list

28)Maximum depth / height of a binary tree

29)check if two trees are having same structure

30)Binary tree level order traversal

31)Subtree of another tree

32)Construct binary tree from inorder and preorder traversal

33)Kth Smallest Element in BST

34)Lowest Ancestor in BST

35)Search and Insert Implementation on trie

36)Search a node in BST

37)Insertion in a BST

38)Merge two sorted linked lists

39)Convert infix to postfix expression

40)Queues program in C  
41)Implementation of Queues using Arrays

42)Queues using Linked Lists

43)Reverse a Queue

44)Implement queue using two stacks

45)Circular Queue in C

46) Priority Queue – Insertion and Deletion

47) Inorder, preorder and Postorder traversal in Binary Tree

48) Lowest Common Ancestor in Binary Tree

49)Given BT is a mirrored or not

50)Sum tree or not

51)Diameter of a binary tree

52)Sum from root to leaf path

53)Ancestors of a binary tree

54)Lowest common Ancestor of a Binary tree

55)Binary Search Tree

56) Deletion in Binary Search Tree

57)Heap Sort

58)Max Heap

59)Min Heap

60)Index Mapping / Trivial Hashing

61)Separate Chaining

62)Program to implement Hash Table using Open Addressing

63)Program to implement Hash Table with Double Hashing

64)Towers of Hanoi

65)Segregate 0’s and 1’s in an array

66)Sorting of array

68)Array Rotation

1**)TWO SUM67**

#include <stdio.h>

#include <stdlib.h>

void two\_Sum67(int arr[], int res[], int n, int target) {

int i;

int \*map = (int \*)malloc(n \* sizeof(int));

for (i = 0; i < n; i++) {

if (map[target - arr[i]] != 0) {

res[1] = i;

res[0] = map[target - arr[i]];

}

map[arr[i]] = i;

}

free(map);

}

int main() {

int n, target, i;

scanf("%d", &n);

int \*arr = (int \*)malloc(n \* sizeof(int));

for (i = 0; i < n; i++) {

scanf("%d", &arr[i]);

}

scanf("%d", &target);

int \*res = (int \*)malloc(2 \* sizeof(int));

two\_Sum67(arr, res, n, target);

for (i = 0; i < 2; i++) {

printf("%d ", res[i]);

}

free(arr);

free(res);

return 0;

}

**2)BEST TIME TO BUY AND SELL STOCK**

#include <stdio.h>

int maxProfit(int price[], int start, int end)

{

if (end <= start)

return 0;

int profit = 0;

for (int i = start; i < end; i++) {

for (int j = i + 1; j <= end; j++) {

if (price[j] > price[i]) {

int curr\_profit = price[j] - price[i];

if (curr\_profit > profit) {

profit = curr\_profit;

}

}

}

}

return profit;

}

int main()

{

int n;

scanf("%d", &n);

int price[n];

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

scanf("%d", &price[i]);

}

printf("%d", maxProfit(price, 0, n - 1));

return 0;

}

**3)FIND DULPICATES FROM THE SORTED ARRAY**

#include <stdio.h>

int main() {

int n;

scanf("%d", &n);

int arr[n];

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

scanf("%d", &arr[i]);

}

int dup = 0;

for(int i=0; i<n-1; i++) {

if(arr[i] == arr[i+1]) {

dup = arr[i];

}

}

printf("%d\n", dup);

return 0;

}

**4)JUMP GAME5**

#include <stdio.h>

int main() {

int n;

scanf("%d", &n);

int arr[n];

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

scanf("%d", &arr[i]);

}

int max\_jump = 0;

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

max\_jump = (max\_jump > i+arr[i]) ? max\_jump : i+arr[i];

if(max\_jump < i+1){

break;

}

}

if(max\_jump >= n-1){

printf("true\n");

}

else{

printf("false\n");

}

return 0;

}

**5)ROTATE ARRAY**

#include <stdio.h>

int main() {

int n;

scanf("%d", &n);

int arr[n];

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

scanf("%d", &arr[i]);

}

int max\_jump = 0;

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

max\_jump = (max\_jump > i+arr[i]) ? max\_jump : i+arr[i];

if(max\_jump < i+1){

break;

}

}

if(max\_jump >= n-1){

printf("true\n");

}

else{

printf("false\n");

}

return 0;

}

**6)FIND UNIQUE**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

int main() {

int n;

scanf("%d", &n);

int \*arr = (int\*)malloc(n \* sizeof(int));

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

scanf("%d", &arr[i]);

}

int \*freqMap = (int\*)calloc(n, sizeof(int));

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

freqMap[arr[i]]++;

}

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

if (freqMap[arr[i]] == 1) {

printf("%d\n", arr[i]);

}

}

free(arr);

free(freqMap);

return 0;

}

**7)FIND THE FIRST POSITIVE MISSING NUMBER**

#include <stdio.h>

#include <stdbool.h>

int findMissing(int arr[], int n) {

bool present[n + 1];

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

if (arr[i] > 0 && arr[i] <= n) {

present[arr[i]] = true;

}

}

for (int i = 1; i <= n; i++) {

if (!present[i]) {

return i;

}

}

return n + 1;

}

int main() {

int n;

scanf("%d", &n);

int arr[n];

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

scanf("%d", &arr[i]);

}

int missing = findMissing(arr, n);

printf("%d\n", missing);

return 0;

}

**8)TRAPPING RAIN WATER**

#include <stdio.h>

#define max(x, y) (((x) > (y)) ? (x) : (y))

#define min(x, y) (((x) < (y)) ? (x) : (y))

int maxWater(int arr[], int n)

{

int res = 0;

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

int left = arr[i];

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

left = max(left, arr[j]);

}

int right = arr[i];

for (int j = i + 1; j < n; j++) {

right = max(right, arr[j]);

}

res = res + (min(left, right) - arr[i]);

}

return res;

}

int main()

{

int arr[] = { 0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1 };

int n = sizeof(arr) / sizeof(arr[0]);

printf("%d", maxWater(arr, n));

return 0;

}

**9)STACK PROGRAM IN C**

#include <limits.h>

#include <stdio.h>

#include <stdlib.h>

struct Stack

{

int top;

int maxSize;

int \*array;

};

struct Stack \*create (int max)

{

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

stack->maxSize = max;

stack->top = -1;

stack->array = (int \*) malloc (stack->maxSize \* sizeof (int));

return stack;

}

int isFull (struct Stack \*stack)

{

if (stack->top == stack->maxSize - 1)

{

printf ("Will not be able to push maxSize reached\n");

}

return stack->top == stack->maxSize - 1;

}

int isEmpty (struct Stack \*stack)

{

return stack->top == -1;

}

void push (struct Stack \*stack, int item)

{

if (isFull (stack))

return;

stack->array[++stack->top] = item;

printf ("We have pushed %d to stack\n", item);

}

int pop (struct Stack \*stack)

{

if (isEmpty (stack))

return INT\_MIN;

return stack->array[stack->top--];

}

int peek (struct Stack \*stack)

{

if (isEmpty (stack))

return INT\_MIN;

return stack->array[stack->top];

}

int main ()

{

struct Stack \*stack = create (10);

push (stack, 5);

push (stack, 10);

push (stack, 15);

int flag = 1;

while (flag)

{

if (!isEmpty (stack))

printf ("We have popped %d from stack\n", pop (stack));

else

printf ("Can't Pop stack must be empty\n");

flag = 0;

}

return 0;

}

**10)REPRESENTATION OF STACK USING AN ARRAY**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

struct Stack

{

int top;

int maxSize;

int \*array;

};

struct Stack \*create (int max)

{

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

stack->maxSize = max;

stack->top = -1;

stack->array = (int \*) malloc (stack->maxSize \* sizeof (int));

return stack;

}

int isFull (struct Stack \*stack)

{

if (stack->top == stack->maxSize - 1)

{

printf ("Will not be able to push maxSize reached\n");

}

return stack->top == stack->maxSize - 1;

}

int isEmpty (struct Stack \*stack)

{

return stack->top == -1;

}

void push (struct Stack \*stack, int item)

{

if (isFull (stack))

return;

stack->array[++stack->top] = item;

printf ("We have pushed %d to stack\n", item);

}

int pop (struct Stack \*stack)

{

if (isEmpty (stack))

return INT\_MIN;

return stack->array[stack->top--];

}

int peek (struct Stack \*stack)

{

if (isEmpty (stack))

return INT\_MIN;

return stack->array[stack->top];

}

int main ()

{

struct Stack \*stack = create (10);

push (stack, 5);

push (stack, 10);

push (stack, 15);

int flag = 1;

while (flag)

{

if (!isEmpty (stack))

printf ("We have popped %d from stack\n", pop (stack));

else

printf ("Can't Pop stack must be empty\n");

flag = 0;

}

return 0;

}

**11)REPRESENTATION OF STACK AS A LINKED LIST**

#include<stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node \*next;

};

struct Node \*head = NULL;

void push(int val)

{

//create new node

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

newNode->data = val;

newNode->next = head;

head = newNode;

}

void pop()

{

//temp is used to free the head node

struct Node \*temp;

if(head == NULL)

printf("Stack is Empty\n");

else

{

printf("Poped element = %d\n", head->data);

//backup the head node

temp = head;

head = head->next;

free(temp);

}

}

void display()

{

struct Node \*temp = head;

while(temp != NULL)

{

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

temp = temp->next;

}

printf("NULL\n");

}

int main()

{

push(10);

push(20);

push(30);

printf("Linked List\n");

display();

pop();

printf("After the pop, the new linked list\n");

display();

pop();

printf("After the pop, the new linked list\n");

display();

return 0;

}

**12)CHOCOLATE DISTRIBUTION**

#include <limits.h>

#include <stdio.h>

#include <stdlib.h>

// Compare function for qsort

int cmpfunc(const void\* a, const void\* b)

{

return (\*(int\*)a - \*(int\*)b);

}

// arr[0..n-1] represents sizes of packets m is number of students. Returns minimum difference between maximum

// and minimum values of distribution.

int findMinDiff(int arr[], int n, int m)

{

if (m == 0 || n == 0)

return 0;

// Sort the given packets

qsort(arr, n, sizeof(int), cmpfunc);

// Number of students cannot be more than number of packets

if (n < m)

return -1;

int min\_diff = INT\_MAX;

// Find the subarray of size m such that difference between last (maximum in case of sorted) and first (minimum in case of

// sorted) elements of subarray is minimum.

for (int i = 0; i + m - 1 < n; i++) {

int diff = arr[i + m - 1] - arr[i];

if (diff < min\_diff)

min\_diff = diff;

}

return min\_diff;

}

int main()

{

int arr[] = { 12, 4, 7, 9, 2, 23, 25, 41, 30,40, 28, 42, 30, 44, 48, 43, 50 };

int m = 7;

int n = sizeof(arr) / sizeof(arr[0]);

printf("Minimum difference is %d",findMinDiff(arr, n, m));

return 0;

}

**13)NEXT GREATER ELEMENT**

#include <stdio.h>

void printNGE(int arr[], int n)

{

int next, i, j;

for (i = 0; i < n; i++) {

next = -1;

for (j = i + 1; j < n; j++) {

if (arr[i] < arr[j]) {

next = arr[j];

break;

}

}

printf("%d -- %dn", arr[i], next);

}

}

int main()

{

int arr[] = { 11, 13, 21, 3 };

int n = sizeof(arr) / sizeof(arr[0]);

printNGE(arr, n);

return 0;

}

**14)FIND THE FIRST CIRCULAR TOUR THAT VISITS ALL PETROL PUMPS**

#include <stdio.h>

struct petrolPump

{

int petrol;

int distance;

};

// The function returns starting point if there is a possible solution, otherwise returns -1

int printTour(struct petrolPump arr[], int n)

{

int start = 0;

int end = 1;

int curr\_petrol = arr[start].petrol - arr[start].distance;

/\* Run a loop while all petrol pumps are not visited. And we have reached first petrol pump again with 0 or more petrol \*/

while (end != start || curr\_petrol < 0)

{

// If current amount of petrol in truck becomes less than 0, then remove the starting petrol pump from tour

while (curr\_petrol < 0 && start != end)

{

// Remove starting petrol pump. Change start

curr\_petrol -= arr[start].petrol - arr[start].distance;

start = (start + 1)%n;

// If 0 is being considered as start again, then there is no possible solution

if (start == 0)

return -1;

}

// Add a petrol pump to current tour

curr\_petrol += arr[end].petrol - arr[end].distance;

end = (end + 1)%n;

}

return start;

}

int main()

{

struct petrolPump arr[] = {{6, 4}, {3, 6}, {7, 3}};

int n = sizeof(arr)/sizeof(arr[0]);

int start = printTour(arr, n);

(start == -1)? printf("No solution"): printf("Start = %d", start);

return 0;

}

**15)DELETE LAST OCCURRENCE OF AN ITEM FROM THE LINKED LIST**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

void deleteLast(struct Node\*\* head, int x)

{

struct Node\*\* tmp1 = NULL;

while(\*head) {

if((\*head)->data == x) {

tmp1 = head;

}

head = &(\*head)->next;

}

if(tmp1) {

struct Node\* tmp = \*tmp1;

\*tmp1 = tmp->next;

free(tmp);

}

}

struct Node\* newNode(int x)

{

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

node->data = x;

node->next = NULL;

return node;

}

void display(struct Node\* head)

{

struct Node\* temp = head;

if (head == NULL) {

printf("NULL\n");

return;

}

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

int main()

{

struct Node\* head = newNode(1);

head->next = newNode(2);

head->next->next = newNode(3);

head->next->next->next = newNode(4);

head->next->next->next->next = newNode(5);

head->next->next->next->next->next = newNode(4);

head->next->next->next->next->next->next = newNode(4);

printf("Created Linked list: ");

display(head);

deleteLast(&head, 4);

printf("List after deletion of 4: ");

display(head);

return 0;

}

**16)REMOVE NTH NODE FROM THE END OF A LINKED LIST**

#include<stdio.h>

#include<stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* create(struct Node\* head, int x)

{

struct Node \*temp, \*ptr = head;

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

temp->data = x;

temp->next = NULL;

if (head == NULL)

head = temp;

else {

while (ptr->next != NULL) {

ptr = ptr->next;

}

ptr->next = temp;

}

return head;

}

struct Node\* removeNthFromEnd(struct Node\* head, int B)

{

int len = 0;

struct Node\* tmp = head;

while (tmp != NULL) {

len++;

tmp = tmp->next;

}

// B > length, then we can't remove node

if (B > len)

{

printf( "Length of the linked list is %d",len );

printf( " we can't remove %dth node from the",B);

printf(" linked list\n");

return head;

}

// We need to remove head node

else if (B == len) {

return head->next;

}

else

{

// Remove len - B th node from starting

int diff = len - B;

struct Node\* prev= NULL;

struct Node\* curr = head;

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

prev = curr;

curr = curr->next;

}

prev->next = curr->next;

return head;

}

}

void display(struct Node\* head)

{

struct Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

}

int main()

{

struct Node\* head = NULL;

head = create(head, 1);

head = create(head, 2);

head = create(head, 3);

head = create(head, 4);

head = create(head, 5);

int n = 2;

printf("Linked list before modification: \n");

display(head);

head = removeNthFromEnd(head, 2);

printf("Linked list after modification: \n");

display(head);

return 0;

}

**17)REVERSE A LINKED LIST**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

static void reverse(struct Node\*\* head\_ref)

{

struct Node\* prev = NULL;

struct Node\* current = \*head\_ref;

struct Node\* next = NULL;

while (current != NULL) {

next = current->next;

// Reverse current node's pointer

current->next = prev;

prev = current;

current = next;

}

\*head\_ref = prev;

}

void push(struct Node\*\* head\_ref, int new\_data)

{

struct Node\* new\_node= (struct Node\*)malloc(sizeof(struct Node));

new\_node->data = new\_data;

new\_node->next = (\*head\_ref);

(\*head\_ref) = new\_node;

}

void printList(struct Node\* head)

{

struct Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

}

int main()

{

struct Node\* head = NULL;

push(&head, 20);

push(&head, 4);

push(&head, 15);

push(&head, 85);

printf("Given linked list\n");

printList(head);

reverse(&head);

printf("\nReversed linked list \n");

printList(head);

getchar();

}

**18)COUNT WAYS TO REACH NTH STAIR**

#include <stdio.h>

int fib(int n)

{

if (n <= 1)

return n;

return fib(n - 1) + fib(n - 2);

}

int countWays(int s) { return fib(s + 1); }

int main()

{

int s = 4;

printf("Number of ways = %d", countWays(s));

getchar();

return 0;

}

**19)COIN CHANGE**

#include <stdio.h>

// Returns the count of ways we can sum coins[0...n-1] coins to get sum "sum"

int count(int coins[], int n, int sum)

{

// If sum is 0 then there is 1 solution (do not include any coin)

if (sum == 0)

return 1;

// If sum is less than 0 then no solution exists

if (sum < 0)

return 0;

// If there are no coins and sum is greater than 0, then no solution exist

if (n <= 0)

return 0;

// count is sum of solutions (i) including coins[n-1] (ii) excluding coins[n-1]

return count(coins, n - 1, sum)+ count(coins, n, sum - coins[n - 1]);

}

int main()

{

int i, j;

int coins[] = { 1, 2, 3 };

int n = sizeof(coins) / sizeof(coins[0]);

printf("%d ", count(coins, n, 5));

getchar();

return 0;

}

**20)0/1 KNAPSACK PROBLEM**

#include <stdio.h>

int max(int a, int b) { return (a > b) ? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n)

{

if (n == 0 || W == 0)

return 0;

// If weight of the nth item is more than Knapsack capacity W, then this item cannot be included in the optimal solution

if (wt[n - 1] > W)

return knapSack(W, wt, val, n - 1);

// Return the maximum of two cases: (1) nth item included (2) not included

else

return max(

val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1), knapSack(W, wt, val, n - 1));

}

int main()

{

int profit[] = { 60, 100, 120 };

int weight[] = { 10, 20, 30 };

int W = 50;

int n = sizeof(profit) / sizeof(profit[0]);

printf("%d", knapSack(W, weight, profit, n));

return 0;

}

**21)LARGEST INCREASING SUBSEQUENCE**

#include <stdio.h>

#include <stdlib.h>

// To make use of recursive calls, this function must return two things:1) Length of LIS ending with element arr[n-1].

// We use max\_ending\_here for this purpose 2) Overall maximum as the LIS may end with an element before arr[n-1] max\_ref is used

// The value of LIS of full array of size n stored in \*max\_ref which is our final result

int \_lis(int arr[], int n, int\* max\_ref)

{

if (n == 1)

return 1;

// 'max\_ending\_here' is length of LIS ending with arr[n-1]

int res, max\_ending\_here = 1;

// Recursively get all LIS ending with arr[0], arr[1] ... arr[n-2]. If arr[i-1] is smaller

// than arr[n-1], and max ending with arr[n-1] needs to be updated, then update it

for (int i = 1; i < n; i++) {

res = \_lis(arr, i, max\_ref);

if (arr[i - 1] < arr[n - 1]

&& res + 1 > max\_ending\_here)

max\_ending\_here = res + 1;

}

// Compare max\_ending\_here with the overall max. And update the overall max if needed

if (\*max\_ref < max\_ending\_here)

\*max\_ref = max\_ending\_here;

// Return length of LIS ending with arr[n-1]

return max\_ending\_here;

}

// The wrapper function for \_lis()

int lis(int arr[], int n)

{

int max = 1;

\_lis(arr, n, &max);

return max;

}

int main()

{

int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };

int n = sizeof(arr) / sizeof(arr[0]);

printf("Length of lis is %d", lis(arr, n));

return 0;

}

**22)LONGEST COMMON SUBSEQUENCE**

#include <stdio.h>

int max(int a, int b);

// Returns length of LCS for X[0..m-1], Y[0..n-1]

int lcs(char\* X, char\* Y, int i, int j)

{

if (X[i] == 0 || Y[j] == 0)

return 0;

if (X[i] == Y[j])

return 1 + lcs(X, Y, i + 1, j + 1);

else

return max(lcs(X, Y, i, j + 1),

lcs(X, Y, i + 1, j));

}

int max(int a, int b) { return (a > b) ? a : b; }

int main()

{

char S1[] = "BD";

char S2[] = "ABCD";

int m = strlen(S1);

int n = strlen(S2);

int i = 0, j = 0;

printf("Length of LCS is %d", lcs(S1, S2, i, j));

return 0;

}

**23)EGG DROPPING PUZZLE**

#include <limits.h>

#include <stdio.h>

int max(int a, int b) { return (a > b) ? a : b; }

/\* Function to get minimum number of trials needed in worst case with n eggs and k floors \*/

int eggDrop(int n, int k)

{

// If there are no floors, then no trials needed. OR if there is one floor, one trial needed.

if (k == 1 || k == 0)

return k;

// We need k trials for one egg and k floors

if (n == 1)

return k;

int min = INT\_MAX, x, res;

// Consider all droppings from 1st floor to kth floor and return the minimum of these values plus 1.

for (x = 1; x <= k; x++) {

res = max(eggDrop(n - 1, x - 1), eggDrop(n, k - x));

if (res < min)

min = res;

}

return min + 1;

}

int main()

{

int n = 2, k = 10;

printf("Minimum number of trials in "

"worst case with %d eggs and "

"%d floors is %d \n",

n, k, eggDrop(n, k));

return 0;

}

**24)MATRIX CHAIN MULTIPLICATION**

#include <limits.h>

#include <stdio.h>

// Matrix Ai has dimension p[i-1] x p[i] for i = 1 . . . n

int MatrixChainOrder(int p[], int i, int j)

{

if (i == j)

return 0;

int k;

int min = INT\_MAX;

int count;

// Place parenthesis at different places between first and last matrix, recursively calculate count of multiplications

// for each parenthesis placement and return the minimum count

for (k = i; k < j; k++)

{

count = MatrixChainOrder(p, i, k)

+ MatrixChainOrder(p, k + 1, j)

+ p[i - 1] \* p[k] \* p[j];

if (count < min)

min = count;

}

return min;

}

int main()

{

int arr[] = { 1, 2, 3, 4, 3 };

int N = sizeof(arr) / sizeof(arr[0]);

printf("Minimum number of multiplications is %d ",

MatrixChainOrder(arr, 1, N - 1));

getchar();

return 0;

}

**25)SUBSET SUM PROBLEM USING RECURSION**

#include <stdio.h>

#include <stdbool.h>

// Returns true if there is a subset of set[] with sum equal to given sum

bool isSubsetSum(int set[], int n, int sum)

{

if (sum == 0)

return true;

if (n == 0)

return false;

// If last element is greater than sum,then ignore it

if (set[n - 1] > sum)

return isSubsetSum(set, n - 1, sum);

// Else, check if sum can be obtained by any of the following:

// (a) including the last element (b) excluding the last element

return isSubsetSum(set, n - 1, sum)

|| isSubsetSum(set, n - 1, sum - set[n - 1]);

}

int main()

{

int set[] = { 3, 34, 4, 12, 5, 2 };

int sum = 9;

int n = sizeof(set) / sizeof(set[0]);

if (isSubsetSum(set, n, sum) == true)

printf("Found a subset with given sum");

else

printf("No subset with given sum");

return 0;

}

**26)PAIRWISE SWAP NODES OF A GIVEN LINKED LIST**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

void swap(int\* a, int\* b);

void pairWiseSwap(struct Node\* head)

{

struct Node\* temp = head;

/\* Traverse further only if there are at-least two nodes left \*/

while (temp != NULL && temp->next != NULL) {

/\* Swap data of node with its next node's data \*/

swap(&temp->data, &temp->next->data);

temp = temp->next->next;

}

}

void swap(int\* a, int\* b)

{

int temp;

temp = \*a;

\*a = \*b;

\*b = temp;

}

void push(struct Node\*\* head\_ref, int new\_data)

{

struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

new\_node->data = new\_data;

/\* link the old list of the new node \*/

new\_node->next = (\*head\_ref);

(\*head\_ref) = new\_node;

}

void printList(struct Node\* node)

{

while (node != NULL) {

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

node = node->next;

}

}

int main()

{

struct Node\* start = NULL;

push(&start, 5);

push(&start, 4);

push(&start, 3);

push(&start, 2);

push(&start, 1);

printf("Linked list before calling pairWiseSwap()\n");

printList(start);

pairWiseSwap(start);

printf("\nLinked list after calling pairWiseSwap()\n");

printList(start);

return 0;

}

**27)COUNT FOR NODES IN A CIRCULAR LINKED LIST**

#include<stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node \*next;

};

int count(struct Node \*head)//function to count number of nodes

{

int cnt = 0;

struct Node \*cur = head;

do

{

cur = cur->next;

cnt++;

} while (cur != head);

return cnt;

}

void insertStart (struct Node \*\*head, int data)

{

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

newNode->data = data;

if (\*head == NULL)

{

\*head = newNode;

(\*head)->next = \*head;

return;

}

struct Node \*curr = \*head;

while (curr->next != \*head)

{

curr = curr->next;

}

curr->next = newNode;

newNode->next = \*head;

\*head = newNode;

}

void display (struct Node \*head)

{

if (head == NULL)

return;

struct Node \*temp = head;

do

{

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

temp = temp->next;

}

while (temp != head);

printf ("\n");

}

int main ()

{

struct Node \*head = NULL;

struct Node \*head1 = NULL;

struct Node \*head2 = NULL;

insertStart (&head, 6);

insertStart (&head, 5);

insertStart (&head, 4);

insertStart (&head, 3);

insertStart (&head, 2);

insertStart (&head, 1);

printf ("Linked List is: ");

display (head);

printf("Total number of nodes are: %d",count(head));

return 0;

}

**28)MAXIMUM DEPTH / HEIGHT OF A BINARY TREE**

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node\* left;

struct node\* right;

};

int maxDepth(struct node\* node)

{

if (node == NULL)

return 0;

else {

int lDepth = maxDepth(node->left);

int rDepth = maxDepth(node->right);

if (lDepth > rDepth)

return (lDepth + 1);

else

return (rDepth + 1);

}

}

struct node\* newNode(int data)

{

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

node->data = data;

node->left = NULL;

node->right = NULL;

return (node);

}

int main()

{

struct node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

printf("Height of tree is %d", maxDepth(root));

getchar();

return 0;

}

**29)CHECK IF TWO TREES ARE HAVING SAME STRUCTURE**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* left;

struct Node\* right;

} Node;

Node\* newNode(int data)

{

Node\* node = (Node\*)malloc(sizeof(Node));

node->data = data;

node->left = NULL;

node->right = NULL;

return (node);

}

int isSameStructure(Node\* a, Node\* b)

{

// 1. both empty

if (a == NULL && b == NULL)

return 1;

// 2. both non-empty -> compare them

if (a != NULL && b != NULL) {

return (isSameStructure(a->left, b->left)&& isSameStructure(a->right, b->right));

}

// 3. one empty, one not -> false

return 0;

}

int main()

{

Node\* root1 = newNode(10);

Node\* root2 = newNode(100);

root1->left = newNode(7);

root1->right = newNode(15);

root1->left->left = newNode(4);

root1->left->right = newNode(9);

root1->right->right = newNode(20);

root2->left = newNode(70);

root2->right = newNode(150);

root2->left->left = newNode(40);

root2->left->right = newNode(90);

root2->right->right = newNode(200);

if (isSameStructure(root1, root2))

printf("Both trees have same structure");

else

printf("Trees do not have same structure");

return 0;

}

**30)BINARY TREE LEVEL ORDER TRAVERSAL**

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node \*left, \*right;

};

void printCurrentLevel(struct node\* root, int level);

int height(struct node\* node);

struct node\* newNode(int data);

void printLevelOrder(struct node\* root)

{

int h = height(root);

int i;

for (i = 1; i <= h; i++)

printCurrentLevel(root, i);

}

void printCurrentLevel(struct node\* root, int level)

{

if (root == NULL)

return;

if (level == 1)

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

else if (level > 1) {

printCurrentLevel(root->left, level - 1);

printCurrentLevel(root->right, level - 1);

}

}

int height(struct node\* node)

{

if (node == NULL)

return 0;

else {

int lheight = height(node->left);

int rheight = height(node->right);

if (lheight > rheight)

return (lheight + 1);

else

return (rheight + 1);

}

}

struct node\* newNode(int data)

{

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

node->data = data;

node->left = NULL;

node->right = NULL;

return (node);

}

int main()

{

struct node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

printf("Level Order traversal of binary tree is \n");

printLevelOrder(root);

return 0;

}

**31)SUBTREE OF ANOTHER TREE**

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node\* left;

struct node\* right;

};

bool areIdentical(struct node\* root1, struct node\* root2)

{

if (root1 == NULL && root2 == NULL)

return true;

if (root1 == NULL || root2 == NULL)

return false;

return (root1->data == root2->data && areIdentical(root1->left, root2->left)

&& areIdentical(root1->right, root2->right));

}

bool isSubtree(struct node\* T, struct node\* S)

{

/\* base cases \*/

if (S == NULL)

return true;

if (T == NULL)

return false;

if (areIdentical(T, S))

return true;

return isSubtree(T->left, S) || isSubtree(T->right, S);

}

struct node\* newNode(int data)

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

node->data = data;

node->left = NULL;

node->right = NULL;

return (node);

}

int main()

{

/\* Construct the following tree

26

/ \

10 3

/ \ \

4 6 3

\

30

\*/

struct node\* T = newNode(26);

T->right = newNode(3);

T->right->right = newNode(3);

T->left = newNode(10);

T->left->left = newNode(4);

T->left->left->right = newNode(30);

T->left->right = newNode(6);

// TREE 2

/\* Construct the following tree

10

/ \

4 6

\

30

\*/

struct node\* S = newNode(10);

S->right = newNode(6);

S->left = newNode(4);

S->left->right = newNode(30);

if (isSubtree(T, S))

printf("Tree 2 is subtree of Tree 1");

else

printf("Tree 2 is not a subtree of Tree 1");

getchar();

return 0;

}

**32)CONSTRUCT BINARY TREE FROM INORDER AND PREORDER TRAVERSAL**

#include <stdio.h>

#include <stdlib.h>

struct node {

char data;

struct node\* left;

struct node\* right;

};

int search(char arr[], int strt, int end, char value);

struct node\* newNode(char data);

struct node\* buildTree(char in[], char pre[], int inStrt, int inEnd)

{

static int preIndex = 0;

if (inStrt > inEnd)

return NULL;

struct node\* tNode = newNode(pre[preIndex++]);

if (inStrt == inEnd)

return tNode;

int inIndex = search(in, inStrt, inEnd, tNode->data);

tNode->left = buildTree(in, pre, inStrt, inIndex - 1);

tNode->right = buildTree(in, pre, inIndex + 1, inEnd);

return tNode;

}

int search(char arr[], int strt, int end, char value)

{

int i;

for (i = strt; i <= end; i++) {

if (arr[i] == value)

return i;

}

}

struct node\* newNode(char data)

{

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

node->data = data;

node->left = NULL;

node->right = NULL;

return (node);

}

void printInorder(struct node\* node)

{

if (node == NULL)

return;

/\* first recur on left child \*/

printInorder(node->left);

/\* then print the data of node \*/

printf("%c ", node->data);

/\* now recur on right child \*/

printInorder(node->right);

}

int main()

{

char in[] = { 'D', 'B', 'E', 'A', 'F', 'C' };

char pre[] = { 'A', 'B', 'D', 'E', 'C', 'F' };

int len = sizeof(in) / sizeof(in[0]);

struct node\* root = buildTree(in, pre, 0, len - 1);

printf("Inorder traversal of the constructed tree is \n");

printInorder(root);

getchar();

}

**33)KTH SMALLEST ELEMENT IN BST**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node \*left, \*right;

} Node;

struct Node\* new\_node(int x)

{

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

p->data = x;

p->left = NULL;

p->right = NULL;

return p;

}

Node\* insert(Node\* root, int x)

{

if (root == NULL)

return new\_node(x);

if (x < root->data)

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

else if (x > root->data)

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

return root;

}

int count = 0;

Node\* kthSmallest(Node\* root, int k)

{

if (root == NULL)

return NULL;

Node\* left = kthSmallest(root->left, k);

if (left != NULL)

return left;

// if current element is k'th smallest, return it

count++;

if (count == k)

return root;

return kthSmallest(root->right, k);

}

void printKthSmallest(Node\* root, int k)

{

Node\* res = kthSmallest(root, k);

if (res == NULL)

printf("There are less than k nodes in the BST");

else

printf("K-th Smallest Element is %d", res->data);

}

int main()

{

Node\* root = NULL;

int keys[] = { 20, 8, 22, 4, 12, 10, 14 };

int keys\_size = sizeof(keys) / sizeof(keys[0]);

for (int i = 0; i < keys\_size; i++)

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

int k = 3;

printKthSmallest(root, k);

return 0;

}

**34)LOWEST COMMON ANCESTOR OF BST**

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node \*left, \*right;

};

struct node\* lca(struct node\* root, int n1, int n2)

{

if (root == NULL)

return NULL;

if (root->data > n1 && root->data > n2)

return lca(root->left, n1, n2);

if (root->data < n1 && root->data < n2)

return lca(root->right, n1, n2);

return root;

}

struct node\* newNode(int data)

{

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

node->data = data;

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

return (node);

}

int main()

{

struct node\* root = newNode(20);

root->left = newNode(8);

root->right = newNode(22);

root->left->left = newNode(4);

root->left->right = newNode(12);

root->left->right->left = newNode(10);

root->left->right->right = newNode(14);

int n1 = 10, n2 = 14;

struct node\* t = lca(root, n1, n2);

printf("LCA of %d and %d is %d \n", n1, n2, t->data);

n1 = 14, n2 = 8;

t = lca(root, n1, n2);

printf("LCA of %d and %d is %d \n", n1, n2, t->data);

n1 = 10, n2 = 22;

t = lca(root, n1, n2);

printf("LCA of %d and %d is %d \n", n1, n2, t->data);

getchar();

return 0;

}

**35)SEARCH AND INSERT IMPLEMENTATION ON TRIE**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#define ARRAY\_SIZE(a) sizeof(a)/sizeof(a[0])

#define ALPHABET\_SIZE (26)

#define CHAR\_TO\_INDEX(c) ((int)c - (int)'a')

struct TrieNode

{

struct TrieNode \*children[ALPHABET\_SIZE];

bool isEndOfWord;

};

struct TrieNode \*getNode(void)

{

struct TrieNode \*pNode = NULL;

pNode = (struct TrieNode \*)malloc(sizeof(struct TrieNode));

if (pNode)

{

int i;

pNode->isEndOfWord = false;

for (i = 0; i < ALPHABET\_SIZE; i++)

pNode->children[i] = NULL;

}

return pNode;

}

void insert(struct TrieNode \*root, const char \*key)

{

int level;

int length = strlen(key);

int index;

struct TrieNode \*pCrawl = root;

for (level = 0; level < length; level++)

{

index = CHAR\_TO\_INDEX(key[level]);

if (!pCrawl->children[index])

pCrawl->children[index] = getNode();

pCrawl = pCrawl->children[index];

}

// mark last node as leaf

pCrawl->isEndOfWord = true;

}

// Returns true if key presents in trie, else false

bool search(struct TrieNode \*root, const char \*key)

{

int level;

int length = strlen(key);

int index;

struct TrieNode \*pCrawl = root;

for (level = 0; level < length; level++)

{

index = CHAR\_TO\_INDEX(key[level]);

if (!pCrawl->children[index])

return false;

pCrawl = pCrawl->children[index];

}

return (pCrawl->isEndOfWord);

}

int main()

{

char keys[][8] = {"the", "a", "there", "answer", "any","by", "bye", "their"};

char output[][32] = {"Not present in trie", "Present in trie"};

struct TrieNode \*root = getNode();

int i;

for (i = 0; i < ARRAY\_SIZE(keys); i++)

insert(root, keys[i]);

printf("%s --- %s\n", "the", output[search(root, "the")] );

printf("%s --- %s\n", "these", output[search(root, "these")] );

printf("%s --- %s\n", "their", output[search(root, "their")] );

printf("%s --- %s\n", "thaw", output[search(root, "thaw")] );

return 0;

}

**36)SEARCH A NODE IN BST**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

struct node

{

int data;

struct node \*left;

struct node \*right;

};

struct node \*newNode (int data)

{

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

node->data = data;

node->left = NULL;

node->right = NULL;

return (node);

}

bool check\_element (struct node \* root, int key)

{

// If root is null, element is not found:Backtrack

if (root == NULL)

{

return false;

}

if (root->data == key)

{

return true;

}

if (root->data > key)

{

// Traverse the left subtree

return check\_element (root->left, key);

}

// Else Traverse the right subtree

else

{

return check\_element (root->right, key);

}

}

int main ()

{

struct node \*root = newNode (80);

root->left = newNode (60);

root->right = newNode (90);

root->left->left = newNode (40);

root->left->right = newNode (70);

root->left->left->left = newNode (30);

root->left->left->right = newNode (50);

if (check\_element (root, 50))

{

printf ("Found\n");

}

else

{

printf ("Not Found\n");

}

return 0;

}

**37)INSERTION IN A BST**

#include<stdio.h>

#include<stdlib.h>

struct node

{

int val;

struct node \*left, \*right;

};

struct node\* newNode(int item)

{

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

temp->val = item;

temp->left = temp->right = NULL;

return temp;

}

void inorder(struct node\* root)

{

if (root != NULL)

{

inorder(root->left);

printf("%d \n", root->val);

inorder(root->right);

}

}

struct node\* insert(struct node\* node, int val)

{

/\* If the tree(subtree) is empty, return a new node by calling newNode func.\*/

if (node == NULL) return newNode(val);

/\* Else, we will do recursion down the tree to further subtrees \*/

if (val < node->val)

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

else if (val > node->val)

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

return node;

}

int main()

{

/\* Our BST will look like this

100

/ \

40 140

/ \ / \

40 80 120 160 \*/

struct node\* root = NULL;

root = insert(root, 100);

insert(root, 60);

insert(root, 40);

insert(root, 80);

insert(root, 140);

insert(root, 120);

insert(root, 160);

inorder(root);

return 0;

}

**38)DELETION IN A BST**

#include <stdio.h>

#include <stdlib.h>

struct node

{

int data;

struct node \*left, \*right;

};

struct node \*newNode (int value)

{

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

temp->data = value;

temp->left = temp->right = NULL;

return temp;

}

void inorderTraversal (struct node \*root)

{

if (root != NULL)

{

inorderTraversal (root->left);

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

inorderTraversal (root->right);

}

}

struct node \*deleteNode (struct node \*root, int key)

{

if (root == NULL)

return root;

if (key < root->data)

{

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

}

else if (key > root->data)

{

root->right = deleteNode (root->right, key);

}

else

{

if (root->left == NULL)

{

struct node \*temp = root->right;

free (root);

return temp;

}

else if (root->right == NULL)

{

struct node \*temp = root->left;

free (root);

return temp;

}

struct node \*temp = root->right;

while (temp->left != NULL)

{

temp = temp->left;

}

root->data = temp->data;

root->right = deleteNode (root->right, temp->data);

}

return root;

}

int main ()

{

struct node \*root = newNode (50);

root->left = newNode (30);

root->right = newNode (70);

root->left->left = newNode (20);

root->left->right = newNode (40);

root->right->left = newNode (60);

root->right->right = newNode (80);

printf ("Inorder traversal of the original tree: ");

inorderTraversal (root);

int key = 50;

root = deleteNode (root, key);

printf ("\nInorder traversal after deleting %d: ", key);

inorderTraversal (root);

return 0;

}

**39)MERGE TWO SORTED LINKED LISTS**

#include <stdio.h>

#include <stdlib.h>

struct Node

{

int data;

struct Node \*next;

} \*temp = NULL, \*first = NULL, \*second = NULL, \*third = NULL, \*last = NULL;

struct Node\* Create (int A[], int n)

{

int i;

struct Node \*t, \*last;

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

temp->data = A[0];

temp->next = NULL;

last = temp;

for (i = 1; i < n; i++)

{

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

t->data = A[i];

t->next = NULL;

last->next = t;

last = t;

}

return temp;

}

void Display(struct Node \*p)

{

while (p != NULL)

{

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

p = p->next;

}

}

void Merge(struct Node \*first, struct Node \*second)

{

if (first->data < second->data)

{

third = last = first;

first = first->next;

last->next = NULL;

}

else

{

third = last = second;

second = second->next;

last->next = NULL;

}

while (first != NULL && second != NULL)

{

if (first->data < second->data)

{

last->next = first;

last = first;

first = first->next;

last->next = NULL;

}

else

{

last->next = second;

last = second;

second = second->next;

last->next = NULL;

}

}

if (first != NULL)

last->next = first;

else

last->next = second;

}

int main()

{

int A[] = { 3, 4, 7, 9 };

int B[] = { 2, 5, 6, 8 };

first = Create (A, 4);

second = Create (B, 4);

printf ("1st Linked List: ");

Display (first);

printf ("\n2nd Linked List: ");

Display (second);

Merge (first, second);

printf ("\n\nMerged Linked List: \n");

Display (third);

return 0;

}

**40)CONVERT INFIX TO POSTFIX EXPRESSION**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_EXPR\_SIZE 100

int precedence(char operator)

{

switch (operator) {

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

case '^':

return 3;

default:

return -1;

}

}

int isOperator(char ch)

{

return (ch == '+' || ch == '-' || ch == '\*' || ch == '/'

|| ch == '^');

}

char\* infixToPostfix(char\* infix)

{

int i, j;

int len = strlen(infix);

char\* postfix = (char\*)malloc(sizeof(char) \* (len + 2));

char stack[MAX\_EXPR\_SIZE];

int top = -1;

for (i = 0, j = 0; i < len; i++) {

if (infix[i] == ' ' || infix[i] == '\t')

continue;

if (isalnum(infix[i])) {

postfix[j++] = infix[i];

}

else if (infix[i] == '(') {

stack[++top] = infix[i];

}

else if (infix[i] == ')') {

while (top > -1 && stack[top] != '(')

postfix[j++] = stack[top--];

top--;

}

else if (isOperator(infix[i])) {

while (top > -1&& precedence(stack[top])>= precedence(infix[i]))

postfix[j++] = stack[top--];

stack[++top] = infix[i];

}

}

while (top > -1) {

if (stack[top] == '(') {

return "Invalid Expression";

}

postfix[j++] = stack[top--];

}

postfix[j] = '\0';

return postfix;

}

int main()

{

char infix[MAX\_EXPR\_SIZE] = "a+b\*(c^d-e)^(f+g\*h)-i";

char\* postfix = infixToPostfix(infix);

printf("%s\n", postfix);

free(postfix);

return 0;

}

**41)QUEUES PROGRAM IN C**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<stdbool.h>

#define MAX 6

int intArray[MAX];

int front = 0;

int rear = -1;

int itemCount = 0;

int peek() {

return intArray[front];

}

bool isEmpty() {

return itemCount == 0;

}

bool isFull() {

return itemCount == MAX;

}

int size() {

return itemCount;

}

void insert(int data) {

if(!isFull()) {

if(rear == MAX-1) {

rear = -1;

}

intArray[++rear] = data;

itemCount++;

}

}

int removeData() {

int data = intArray[front++];

if(front == MAX) {

front = 0;

}

itemCount--;

return data;

}

int main() {

insert(3);

insert(5);

insert(9);

insert(1);

insert(12);

insert(15);

if(isFull()) {

printf("Queue is full!\n");

}

int num = removeData();

printf("Element removed: %d\n",num);

insert(16);

insert(17);

insert(18);

printf("Element at front: %d\n",peek());

printf("----------------------\n");

printf("index : 5 4 3 2 1 0\n");

printf("----------------------\n");

printf("Queue: ");

while(!isEmpty()) {

int n = removeData();

printf("%d ",n);

}

}

**42)IMPLEMENTATION OF QUEUES USING ARRAYS**

#include<stdio.h>

#define SIZE 5

int queue[SIZE], front = -1, rear = -1;

void enqueue(int item){

if(rear == SIZE-1){

printf("Can't enqueue as the queue is full\n");

}

else{

if(front == -1){

front = 0;

}

rear = rear + 1;

queue[rear] = item;

printf("We have enqueued %d\n",item);

}

}

void dequeue(){

if(front == -1){

printf("Can't dequeue as the queue is empty\n");

}

else{

printf("We have dequeued : %d\n", queue[front]);

front = front + 1;

if(front > rear){

front = -1;

rear = -1;

}

}

}

void printQueue(){

if(rear == -1)

printf("\nUnable to display as queue is empty");

else{

int i;

printf("\nThe queue after enqueue & dequeue ops looks like :");

for(i = front; i <= rear; i++)

printf("%d ",queue[i]);

}

}

int main()

{

enqueue(2);

enqueue(4);

enqueue(6);

enqueue(8);

dequeue();

dequeue();

printQueue();

return 0;

}

**43)QUEUES USING LINKED LIST**

#include <stdio.h>

#include <stdlib.h>

struct node

{

int data;

struct node\* next;

};

struct node \*front = NULL;

struct node \*rear = NULL;

void enqueue(int d)

{

struct node\* new\_n;

new\_n = (struct node\*)malloc(sizeof(struct node));

new\_n->data = d;

new\_n->next = NULL;

if((front == NULL)&&(rear == NULL)){

front = rear = new\_n;

}

else{

rear->next = new\_n;

rear = new\_n;

}

}

void display()

{

struct node\* temp;

if((front == NULL)&&(rear == NULL)){

printf("\nQueue is Empty");

}

else{

temp = front;

while(temp){

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

temp = temp->next;

}

}

}

void dequeue()

{

struct node \*temp;

temp = front;

if((front == NULL)&&(rear == NULL)){

printf("\nQueue is Empty");

}

else{

front = front->next;

free(temp);

}

}

int main()

{

enqueue(5);

enqueue(10);

enqueue(15);

enqueue(20);

enqueue(25);

printf("Queue:");

display();

printf("\nQueue After Dequeue:");

dequeue();

display();

}

**44)REVERSE A QUEUE USING STACK**

#include<stdio.h>

#include<stdlib.h>

#define MAX\_SIZE 100

struct Queue

{

int items[MAX\_SIZE];

int front;

int rear;

};

struct Stack

{

int items[MAX\_SIZE];

int top;

};

void enqueue (struct Queue \*q, int item)

{

if (q->rear == MAX\_SIZE - 1)

{

printf ("Queue overflow\n");

return;

}

if (q->front == -1)

{

q->front = 0;

}

q->rear++;

q->items[q->rear] = item;

}

int dequeue (struct Queue \*q)

{

if (q->front == -1)

{

printf ("Queue underflow\n");

return -1;

}

int item = q->items[q->front];

q->front++;

if (q->front > q->rear)

{

q->front = q->rear = -1;

}

return item;

}

void display (struct Queue \*q)

{

if (q->rear == -1)

{

printf ("Queue is empty\n");

return;

}

for (int i = q->front; i <= q->rear; i++)

{

printf ("%d ", q->items[i]);

}

printf ("\n");

}

void push (struct Stack \*s, int item)

{

if (s->top == MAX\_SIZE - 1)

{

printf ("Stack overflow\n");

return;

}

s->top++;

s->items[s->top] = item;

}

int pop (struct Stack \*s)

{

if (s->top == -1)

{

printf ("Stack underflow\n");

return -1;

}

int item = s->items[s->top];

s->top--;

return item;

}

int main ()

{

struct Queue q;

q.front = -1;

q.rear = -1;

struct Stack s;

s.top = -1;

enqueue (&q, 1);

enqueue (&q, 2);

enqueue (&q, 3);

enqueue (&q, 4);

printf ("Queue before reversing:\n");

display (&q);

while (q.front != -1)

{

push (&s, dequeue (&q));

}

while (s.top != -1)

{

enqueue (&q, pop (&s));

}

printf ("Queue after reversing:\n");

display (&q);

return 0;

}

**45)IMPLEMENT QUEUE USING TWO STACKS**

#include <stdio.h>

#include<stdlib.h>

#define N 100

int stack1[N], stack2[N];

int top\_stack1 = -1;

int top\_stack2 = -1;

int count = 0;

void push\_stack1 (int data)

{

if (top\_stack1 == N - 1)

{

printf ("Stack1 is overflow");

return;

}

else

{

top\_stack1++;

stack1[top\_stack1] = data;

}

return;

}

void push\_stack2 (int data)

{

if (top\_stack2 == N - 1)

{

printf ("Stack2 is overflow");

return;

}

else

{

top\_stack2++;

stack2[top\_stack2] = data;

}

return;

}

int pop\_stack1 ()

{

if (top\_stack1 == -1)

{

printf ("Stack1 is underflow\n");

return -1;

}

return stack1[top\_stack1--];

}

int pop\_stack2 ()

{

if (top\_stack2 == -1)

{

printf ("Stack2 is underflow\n");

return -1;

}

return stack2[top\_stack2--];

}

void enqueue (int data)

{

push\_stack1 (data);

count++;

}

void dequeue ()

{

if (top\_stack1 == -1 && top\_stack2 == -1)

printf ("Queue is empty\n");

else

{

for (int i = 0; i < count; i++)

{

int temp = pop\_stack1 ();

push\_stack2 (temp);

}

int x = pop\_stack2 ();

printf ("Dequeued element is %d\n", x);

count--;

for (int i = 0; i < count; i++)

{

int temp = pop\_stack2 ();

push\_stack1 (temp);

}

}

}

void display ()

{

if (top\_stack1 == -1)

{

printf ("Queue is empty \n");

return;

}

for (int i = 0; i < top\_stack1; i++)

printf ("%d ", stack1[i]);

printf ("\n");

}

void top ()

{

printf ("Top element of queue is %d ", stack1[0]);

}

int main ()

{

enqueue (3);

enqueue (45);

enqueue (-1);

display ();

dequeue ();

display ();

return 0;

}

**46)CIRCULAR QUEUE IN C**

#include <stdio.h>

#include <stdlib.h>

#define SIZE 6

struct Queue

{

int front, rear, currSize;

unsigned maxSize;

int \*a;

};

struct Queue \*createQueue (unsigned maxSize)

{

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

queue->maxSize = maxSize;

queue->front = queue->rear = -1;

queue->a = (int \*) malloc (queue->maxSize \* sizeof (int));

return queue;

}

int isFull (struct Queue \*queue)

{

if ((queue->front == queue->rear + 1) ||

(queue->front == 0 && queue->rear == queue->maxSize - 1))

{

return 1;

}

return 0;

}

int isEmpty (struct Queue \*queue)

{

if (queue->front == -1)

{

return 1;

}

return 0;

}

void enqueue (struct Queue \*queue, int value)

{

if (isFull (queue))

printf ("Can't add the queue is full \n");

else

{

if (queue->front == -1)

queue->front = 0;

queue->rear = (queue->rear + 1) % queue->maxSize;

queue->a[queue->rear] = value;

printf ("%d was added\n", value);

}

}

int dequeue (struct Queue \*queue)

{

int item;

if (isEmpty (queue))

{

printf ("Can't add the queue is empty \n");

return (-1);

}

else

{

item = queue->a[queue->front];

if (queue->front == queue->rear)

{

queue->front = queue->rear = -1;

}

else

{

queue->front = (queue->front + 1) % queue->maxSize;

}

printf ("%d dequeued\n", item);

}

}

void print (struct Queue \*queue)

{

int i;

if (isEmpty (queue))

printf ("Empty Queue\n");

else

{

printf ("\nThe items in the queue are : \n");

for (i = queue->front; i != queue->rear; i = (i + 1) % queue->maxSize)

{

printf ("%d ", queue->a[i]);

}

printf ("%d \n\n", queue->a[i]);

}

}

int main ()

{

struct Queue \*queue = createQueue (6);

dequeue (queue);

enqueue (queue, 12);

enqueue (queue, 14);

enqueue (queue, 16);

enqueue (queue, 18);

enqueue (queue, 20);

print (queue);

dequeue (queue);

dequeue (queue);

print (queue);

enqueue (queue, 22);

enqueue (queue, 24);

enqueue (queue, 26);

enqueue (queue, 28);

print (queue);

return 0;

}

**47)PRIORITY QUEUE INSERTION AND DELETION**

#include<stdio.h>

#include<limits.h>

#define MAX 100

// denotes where the last item in priority queue is initialized to -1 since no item is in queue

int idx = -1;

// pqData holds data for each index item pqPriority holds priority for each index item

int pqData[MAX];

int pqPriority[MAX];

int isEmpty ()

{

return idx == -1;

}

int isFull ()

{

return idx == MAX - 1;

}

void enqueue (int data, int priority)

{

if (!isFull ())

{

idx++;

pqData[idx] = data;

pqPriority[idx] = priority;

}

}

// returns item with highest priority

// NOTE: Low priority number means higher priority | O(N)

int peek ()

{

int maxPriority = INT\_MAX;

int indexPos = -1;

// Linear search for highest priority

for (int i = 0; i <= idx; i++)

{

// If two items have same priority choose the one with higher data value

if (maxPriority == pqPriority[i] && indexPos > -1

&& pqData[indexPos] < pqData[i])

{

maxPriority = pqPriority[i];

indexPos = i;

}

else if (maxPriority > pqPriority[i])

{

maxPriority = pqPriority[i];

indexPos = i;

}

}

return indexPos;

}

void dequeue ()

{

if (!isEmpty ())

{

int indexPos = peek ();

for (int i = indexPos; i < idx; i++)

{

pqData[i] = pqData[i + 1];

pqPriority[i] = pqPriority[i + 1];

}

idx--;

}

}

void display ()

{

for (int i = 0; i <= idx; i++)

{

printf ("(%d, %d)\n", pqData[i], pqPriority[i]);

}

}

int main ()

{

enqueue (10, 1);

enqueue (20, 3);

enqueue (30, 4);

enqueue (40, 5);

enqueue (1000, 2);

printf ("Before: \n");

display ();

dequeue ();

dequeue ();

printf ("\nAfter: \n");

display ();

return 0;

}

**48)INORDER, PREPORDER AND POSTORDER TRAVERSAL IN BINARY TREE**

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*left, \*right;

};

struct node \*newNode (int item)

{

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

temporary->data = item;

temporary->left = temporary->right = NULL;

return temporary;

}

void postorder (struct node \*root)

{

if (root != NULL)

{

postorder (root->left);

postorder (root->right);

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

}

}

void preorder (struct node \*root)

{

if (root != NULL)

{

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

preorder (root->left);

preorder (root->right);

}

}

void inorder (struct node \*root)

{

if (root != NULL)

{

inorder (root->left);

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

inorder (root->right);

}

}

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

{

if (node == NULL)

return newNode (data);

if (data < node->data)

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

else if (data > node->data)

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

return node;

}

int main ()

{

/\* What our binary search tree looks like really

9

/ \

7 14

/ \ / \

5 8 11 16 \*/

struct node \*root = NULL;

root = insert (root, 9);

insert (root, 7);

insert (root, 5);

insert (root, 8);

insert (root, 14);

insert (root, 11);

insert (root, 16);

printf ("The postorder is :\n");

postorder (root);

printf ("\nThe preorder is :\n");

preorder (root);

printf ("\nThe inorder is :\n");

inorder (root);

return 0;

}

**49)LOWEST COMMON ANCESTOR IN BINARY TREE**

#include <stdio.h>

#include <stdlib.h>

struct node

{

int data;

struct node \*left, \*right;

};

struct node \*lca (struct node \*root, int n1, int n2)

{

while (root != NULL)

{

if (root->data > n1 && root->data > n2)

root = root->left;

else if (root->data < n1 && root->data < n2)

root = root->right;

else

break;

}

return root;

}

struct node \*newNode (int data)

{

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

node->data = data;

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

return (node);

}

int main ()

{

struct node \*root = newNode (20);

root->left = newNode (8);

root->right = newNode (22);

root->left->left = newNode (4);

root->left->right = newNode (12);

root->left->right->left = newNode (10);

root->left->right->right = newNode (14);

int n1 = 10, n2 = 14;

struct node \*t = lca (root, n1, n2);

printf ("LCA of %d and %d is %d \n", n1, n2, t->data);

n1 = 14, n2 = 8;

t = lca (root, n1, n2);

printf ("LCA of %d and %d is %d \n", n1, n2, t->data);

n1 = 10, n2 = 22;

t = lca (root, n1, n2);

printf ("LCA of %d and %d is %d \n", n1, n2, t->data);

getchar ();

return 0;

}

**50)MIRRORED TREES OR NOT**

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

bool areMirrors(struct Node\* root1, struct Node\* root2) {

if (root1 == NULL && root2 == NULL) {

return true;

}

if (root1 == NULL || root2 == NULL) {

return false;

}

return (root1->data == root2->data) &&

areMirrors(root1->left, root2->right) &&

areMirrors(root1->right, root2->left);

}

int main() {

struct Node\* root1 = createNode(1);

struct Node\* root2 = createNode(1);

root1->left = createNode(2);

root1->right = createNode(3);

root1->left->left = createNode(4);

root1->left->right = createNode(5);

root2->left = createNode(3);

root2->right = createNode(2);

root2->right->left = createNode(5);

root2->right->right = createNode(4);

if (areMirrors(root1, root2)) {

printf("The two Binary Trees are mirrored.\n");

} else {

printf("The two Binary Trees are not mirrored.\n");

}

return 0;

}

**51)CHECK IF A BINARY TREE IS SUM TREE OR NOT**

#include <stdio.h>

#include <stdlib.h>

struct node

{

int data;

struct node\* left;

struct node\* right;

};

int sum(struct node \*root)

{

if(root == NULL)

return 0;

return sum(root->left) + root->data + sum(root->right);

}

int isSumTree(struct node\* node)

{

int ls, rs;

if(node == NULL ||

(node->left == NULL && node->right == NULL))

return 1;

ls = sum(node->left);

rs = sum(node->right);

if((node->data == ls + rs)&&

isSumTree(node->left) &&

isSumTree(node->right))

return 1;

return 0;

}

struct node\* newNode(int data)

{

struct node\* node =

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

node->data = data;

node->left = NULL;

node->right = NULL;

return(node);

}

int main()

{

struct node \*root = newNode(26);

root->left = newNode(10);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(6);

root->right->right = newNode(3);

if(isSumTree(root))

printf("The given tree is a SumTree.");

else

printf("The given tree is not a SumTree.");

getchar();

return 0;

}

**52)FIND THE HEIGHT OF A BINARY TREE**

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node \*left, \*right;

};

struct node\* newNode(int data);

int max(int a, int b) { return (a > b) ? a : b; }

int height(struct node\* node);

int diameter(struct node\* tree)

{

if (tree == NULL)

return 0;

int lheight = height(tree->left);

int rheight = height(tree->right);

int ldiameter = diameter(tree->left);

int rdiameter = diameter(tree->right);

// Return max of following three 1) Diameter of left subtree 2) Diameter of right subtree

// 3) Height of left subtree + height of right subtree +

return max(lheight + rheight + 1,

max(ldiameter, rdiameter));

}

int height(struct node\* node)

{

if (node == NULL)

return 0;

// If tree is not empty then height = 1 + max of leftheight and right heights

return 1 + max(height(node->left), height(node->right));

}

struct node\* newNode(int data)

{

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

node->data = data;

node->left = NULL;

node->right = NULL;

return (node);

}

int main()

{

/\* Constructed binary tree is

1

/ \

2 3

/ \

4 5

\*/

struct node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

printf("Diameter of the given binary tree is %d\n",

diameter(root));

return 0;

}

**53)TO CHECK ROOT TO LEAF PATH SUM EQUALS N**

#include <stdio.h>

struct node {

int data;

struct node \*left;

struct node \*right;

};

struct node\* getNewNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

/\*

This function returns below

1

/ \

2 3

/ \ / \

4 5 6 7

\*/

struct node\* generateBTree(){

struct node\* root = getNewNode(1);

root->left = getNewNode(2);

root->right = getNewNode(3);

root->left->left = getNewNode(4);

root->left->right = getNewNode(5);

root->right->left = getNewNode(6);

root->right->right = getNewNode(7);

return root;

}

int rootToLeafPathSum(struct node\* nodePtr, int sum, int N) {

if(nodePtr == NULL)

return 0;

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

if(nodePtr->data + sum == N)

return 1;

else

return 0;

}

s

return rootToLeafPathSum(nodePtr->left, sum + nodePtr->data, N) ||

rootToLeafPathSum(nodePtr->right, sum + nodePtr->data, N);

}

int main() {

struct node \*root = generateBTree();

if(rootToLeafPathSum(root, 0, 8)){

printf("There exist a path from root to a leaf node\n");

} else {

printf("No such path exists\n");

}

getchar();

return 0;

}

**54)ANCESTORS OF A BINARY TREE**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

int printAncestors(struct Node\* root, int key) {

if (root == NULL) {

return 0;

}

if (root->data == key) {

return 1;

}

if (printAncestors(root->left, key) || printAncestors(root->right, key)) {

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

return 1;

}

return 0;

}

int main() {

struct Node\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

root->left->left = createNode(4);

root->left->right = createNode(5);

root->right->left = createNode(6);

root->right->right = createNode(7);

int key;

printf("Enter the key to find ancestors: ");

scanf("%d", &key);

printf("Ancestors of %d are: ", key);

printAncestors(root, key);

return 0;

}

**55)BINARY SEARCH TREE**

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*left;

struct node \*right;

};

struct node \*create\_node (int data)

{

struct node \*new\_node = (struct node \*) malloc (sizeof (struct node));

new\_node->data = data;

new\_node->left = NULL;

new\_node->right = NULL;

return new\_node;

}

struct node \*insert\_node (struct node \*root, int data)

{

if (root == NULL)

{

return create\_node (data);

}

if (data < root->data)

{

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

}

else

{

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

}

return root;

}

struct node \*search\_node (struct node \*root, int data)

{

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

{

return root;

}

if (data < root->data)

{

return search\_node (root->left, data);

}

else

{

return search\_node (root->right, data);

}

}

void inorder (struct node \*root)

{

if (root != NULL)

{

inorder\_traversal (root->left);

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

inorder\_traversal (root->right);

}

}

int main ()

{

struct node \*root = NULL;

root = insert\_node (root, 50);

insert\_node (root, 30);

insert\_node (root, 20);

insert\_node (root, 40);

insert\_node (root, 70);

insert\_node (root, 60);

insert\_node (root, 80);

inorder (root);

return 0;

}

**56)DELETION IN BINARY SEARCH TREE**

#include <stdio.h>

#include <stdlib.h>

struct node

{

int data;

struct node \*left, \*right;

};

struct node \*newNode (int value)

{

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

temp->data = value;

temp->left = temp->right = NULL;

return temp;

}

void inorderTraversal (struct node \*root)

{

if (root != NULL)

{

inorderTraversal (root->left);

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

inorderTraversal (root->right);

}

}

struct node \*deleteNode (struct node \*root, int key)

{

if (root == NULL)

return root;

if (key < root->data)

{

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

}

else if (key > root->data)

{

root->right = deleteNode (root->right, key);

}

else

{

if (root->left == NULL)

{

struct node \*temp = root->right;

free (root);

return temp;

}

else if (root->right == NULL)

{

struct node \*temp = root->left;

free (root);

return temp;

}

struct node \*temp = root->right;

while (temp->left != NULL)

{

temp = temp->left;

}

root->data = temp->data;

root->right = deleteNode (root->right, temp->data);

}

return root;

}

int main ()

{

struct node \*root = newNode (50);

root->left = newNode (30);

root->right = newNode (70);

root->left->left = newNode (20);

root->left->right = newNode (40);

root->right->left = newNode (60);

root->right->right = newNode (80);

printf ("Inorder traversal of the original tree: ");

inorderTraversal (root);

int key = 50;

root = deleteNode (root, key);

printf ("\nInorder traversal after deleting %d: ", key);

inorderTraversal (root);

return 0;

}

**57)HEAP SORT**

#include<stdio.h>

int temp;

void heapify(int arr[], int size, int i)

{

int max = i;

int left = 2\*i + 1;

int right = 2\*i + 2;

if (left < size && arr[left] >arr[max])

max= left;

if (right < size && arr[right] > arr[max])

max= right;

if (max!= i)

{

temp = arr[i];

arr[i]= arr[max];

arr[max] = temp;

heapify(arr, size, max);

}

}

void heapSort(int arr[], int size)

{

int i;

for (i = size / 2 - 1; i >= 0; i--)

heapify(arr, size, i);

for (i=size-1; i>=0; i--)

{

temp = arr[0];

arr[0]= arr[i];

arr[i] = temp;

heapify(arr, i, 0);

}

}

void main()

{

int arr[] = {58, 134, 3, 67, 32, 89, 15, 10,78, 9};

int i;

int size = sizeof(arr)/sizeof(arr[0]);

heapSort(arr, size);

printf("printing sorted elements\n");

for (i=0; i<size; ++i)

printf("%d ",arr[i]);

}

**58)MAX HEAP**

#include <malloc.h>

#include <stdio.h>

struct Heap {

int\* arr;

int size;

int capacity;

};

typedef struct Heap heap;

heap\* createHeap(int capacity, int\* nums);

void insertHelper(heap\* h, int index);

void maxHeapify(heap\* h, int index);

int extractMax(heap\* h);

void insert(heap\* h, int data);

heap\* createHeap(int capacity, int\* nums)

{

heap\* h = (heap\*)malloc(sizeof(heap));

if (h == NULL) {

printf("Memory error");

return NULL;

}

h->size = 0;

h->capacity = capacity;

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

if (h->arr == NULL) {

printf("Memory error");

return NULL;

}

int i;

for (i = 0; i < capacity; i++) {

h->arr[i] = nums[i];

}

h->size = i;

i = (h->size - 2) / 2;

while (i >= 0) {

maxHeapify(h, i);

i--;

}

return h;

}

void insertHelper(heap\* h, int index)

{

int parent = (index - 1) / 2;

if (h->arr[parent] < h->arr[index]) {

int temp = h->arr[parent];

h->arr[parent] = h->arr[index];

h->arr[index] = temp;

insertHelper(h, parent);

}

}

void maxHeapify(heap\* h, int index)

{

int left = index \* 2 + 1;

int right = index \* 2 + 2;

int max = index;

if (left >= h->size || left < 0)

left = -1;

if (right >= h->size || right < 0)

right = -1;

if (left != -1 && h->arr[left] > h->arr[max])

max = left;

if (right != -1 && h->arr[right] > h->arr[max])

max = right;

if (max != index) {

int temp = h->arr[max];

h->arr[max] = h->arr[index];

h->arr[index] = temp;

maxHeapify(h, max);

}

}

int extractMax(heap\* h)

{

int deleteItem;

if (h->size == 0) {

printf("\nHeap id empty.");

return -999;

}

deleteItem = h->arr[0];

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

h->size--;

maxHeapify(h, 0);

return deleteItem;

}

void insert(heap\* h, int data)

{

if (h->size < h->capacity) {

h->arr[h->size] = data;

insertHelper(h, h->size);

h->size++;

}

}

void printHeap(heap\* h)

{

for (int i = 0; i < h->size; i++) {

printf("%d ", h->arr[i]);

}

printf("\n");

}

void main()

{

int arr[9] = {1,2,3,4,5,6,7,8,9};

heap\* hp = createHeap(9, arr);

printHeap(hp);

extractMax(hp);

printHeap(hp);

}

**59)MIN HEAP**

#include <stdio.h>

#include <stdlib.h>

struct Heap {

int\* arr;

int size;

int capacity;

};

typedef struct Heap heap;

heap\* createHeap(int capacity, int\* nums);

void insertHelper(heap\* h, int index);

void heapify(heap\* h, int index);

int extractMin(heap\* h);

void insert(heap\* h, int data);

heap\* createHeap(int capacity, int\* nums)

{

heap\* h = (heap\*)malloc(sizeof(heap));

if (h == NULL) {

printf("Memory error");

return NULL;

}

// set the values to size and capacity

h->size = 0;

h->capacity = capacity;

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

if (h->arr == NULL) {

printf("Memory error");

return NULL;

}

int i;

for (i = 0; i < capacity; i++) {

h->arr[i] = nums[i];

}

h->size = i;

i = (h->size - 2) / 2;

while (i >= 0) {

heapify(h, i);

i--;

}

return h;

}

void insertHelper(heap\* h, int index)

{

int parent = (index - 1) / 2;

if (h->arr[parent] > h->arr[index]) {

int temp = h->arr[parent];

h->arr[parent] = h->arr[index];

h->arr[index] = temp;

insertHelper(h, parent);

}

}

void heapify(heap\* h, int index)

{

int left = index \* 2 + 1;

int right = index \* 2 + 2;

int min = index;

// Checking whether our left or child element is at right index or not to avoid index error

if (left >= h->size || left < 0)

left = -1;

if (right >= h->size || right < 0)

right = -1;

// store left or right element in min if any of these is smaller that its parent

if (left != -1 && h->arr[left] < h->arr[index])

min = left;

if (right != -1 && h->arr[right] < h->arr[min])

min = right;

if (min != index) {

int temp = h->arr[min];

h->arr[min] = h->arr[index];

h->arr[index] = temp;

heapify(h, min);

}

}

int extractMin(heap\* h)

{

int deleteItem;

if (h->size == 0) {

printf("\nHeap id empty.");

return -999;

}

deleteItem = h->arr[0];

// Replace the deleted node with the last node

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

h->size--;

heapify(h, 0);

return deleteItem;

}

void insert(heap\* h, int data)

{

if (h->size < h->capacity) {

h->arr[h->size] = data;

insertHelper(h, h->size);

h->size++;

}

}

void printHeap(heap\* h)

{

for (int i = 0; i < h->size; i++) {

printf("%d ", h->arr[i]);

}

printf("\n");

}

int main()

{

int arr[9] = { 9, 8, 7, 6, 5, 4, 3, 2, 1 };

heap\* hp = createHeap(9, arr);

printHeap(hp);

extractMin(hp);

printHeap(hp);

return 0;

}

**60)INDEX MAPPING OR TRIVIAL HASHING**

#include <stdio.h>

#define MAX\_SIZE 100

int hashTable[MAX\_SIZE];

void insert(int value) {

if (hashTable[value] != -1) {

printf("Value %d is already present in the hash table.\n", value);

return;

}

hashTable[value] = value;

printf("Value %d inserted into the hash table.\n", value);

}

void search(int value) {

if (hashTable[value] != -1) {

printf("Value %d is present in the hash table.\n", value);

} else {

printf("Value %d is not present in the hash table.\n", value);

}

}

void delete(int value) {

if (hashTable[value] != -1) {

hashTable[value] = -1;

printf("Value %d deleted from the hash table.\n", value);

} else {

printf("Value %d is not present in the hash table.\n", value);

}

}

int main() {

for (int i = 0; i < MAX\_SIZE; i++) {

hashTable[i] = -1;

}

insert(5);

insert(10);

insert(15);

search(10);

search(20);

delete(10);

delete(20);

return 0;

}

**61)SEPARATE CHAINING**

#include<stdio.h>

#include<stdlib.h>

#define size 7

struct node

{

int data;

struct node \*next;

};

struct node \*chain[size];

void init()

{

int i;

for(i = 0; i < size; i++)

chain[i] = NULL;

}

void insert(int value)

{

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

newNode->data = value;

newNode->next = NULL;

int key = value % size;

//check if chain[key] is empty

if(chain[key] == NULL)

chain[key] = newNode;

//collision

else

{

//add the node at the end of chain[key].

struct node \*temp = chain[key];

while(temp->next)

{

temp = temp->next;

}

temp->next = newNode;

}

}

void print()

{

int i;

for(i = 0; i < size; i++)

{

struct node \*temp = chain[i];

printf("chain[%d]-->",i);

while(temp)

{

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

temp = temp->next;

}

printf("NULL\n");

}

}

int main()

{

//init array of list to NULL

init();

insert(7);

insert(0);

insert(3);

insert(10);

insert(4);

insert(5);

print();

return 0;

}

**62)PROGRAM TO IMPLEMENT HASH TABLE USING OPEN ADDRESSING**

#include <stdio.h>

#include <stdlib.h>

struct HashNode {

int key;

int value;

};

const int capacity = 20;

int size = 0;

struct HashNode\*\* arr;

struct HashNode\* dummy;

void insert(int key, int V)

{

struct HashNode\* temp= (struct HashNode\*)malloc(sizeof(struct HashNode));

temp->key = key;

temp->value = V;

// Apply hash function to find index for given key

int hashIndex = key % capacity;

// Find next free space

while (arr[hashIndex] != NULL&& arr[hashIndex]->key != key&& arr[hashIndex]->key != -1)

{

hashIndex++;

hashIndex %= capacity;

}

// If new node to be inserted increase the current size

if (arr[hashIndex] == NULL

|| arr[hashIndex]->key == -1)

size++;

arr[hashIndex] = temp;

}

int delete (int key)

{

// Apply hash function to find index for given key

int hashIndex = key % capacity;

// Finding the node with given key

while (arr[hashIndex] != NULL) {

// if node found

if (arr[hashIndex]->key == key) {

// Insert dummy node here for further use

arr[hashIndex] = dummy;

size--;

return 1;

}

hashIndex++;

hashIndex %= capacity;

}

return 0;

}

int find(int key)

{

int hashIndex = (key % capacity);

int counter = 0;

while (arr[hashIndex] != NULL) {

int counter = 0;

if (counter++ > capacity)

break;

if (arr[hashIndex]->key == key)

return arr[hashIndex]->value;

hashIndex++;

hashIndex %= capacity;

}

return -1;

}

int main()

{

arr = (struct HashNode\*\*)malloc(sizeof(struct HashNode\*)\* capacity);

for (int i = 0; i < capacity; i++)

arr[i] = NULL;

dummy= (struct HashNode\*)malloc(sizeof(struct HashNode));

dummy->key = -1;

dummy->value = -1;

insert(1, 5);

insert(2, 15);

insert(3, 20);

insert(4, 7);

if (find(4) != -1)

printf("Value of Key 4 = %d\n", find(4));

else

printf("Key 4 does not exists\n");

if (delete (4))

printf("Node value of key 4 is deleted "

"successfully\n");

else {

printf("Key does not exists\n");

}

if (find(4) != -1)

printf("Value of Key 4 = %d\n", find(4));

else

printf("Key 4 does not exists\n");

}

**63)DOUBLE HASHING**

#include <stdio.h>

#include<stdlib.h>

#define TABLE\_SIZE 10

int h[TABLE\_SIZE]={NULL};

void insert()

{

int key,index,i,flag=0,hkey,hash2;

printf("\nenter a value to insert into hash table\n");

scanf("%d",&key);

hkey=key%TABLE\_SIZE;

hash2 = 7-(key %7);

for(i=0;i<TABLE\_SIZE;i++)

{

index=(hkey+i\*hash2)%TABLE\_SIZE;

if(h[index] == NULL)

{

h[index]=key;

break;

}

}

if(i == TABLE\_SIZE)

printf("\nelement cannot be inserted\n");

}

void search()

{

int key,index,i,flag=0,hash2,hkey;

printf("\nenter search element\n");

scanf("%d",&key);

hkey=key%TABLE\_SIZE;

hash2 = 7-(key %7);

for(i=0;i<TABLE\_SIZE; i++)

{

index=(hkey+i\*hash2)%TABLE\_SIZE;

if(h[index]==key)

{

printf("value is found at index %d",index);

break;

}

}

if(i == TABLE\_SIZE)

printf("\n value is not found\n");

}

void display()

{

int i;

printf("\nelements in the hash table are \n");

for(i=0;i< TABLE\_SIZE; i++)

printf("\nat index %d \t value = %d",i,h[i]);

}

main()

{

int opt,i;

while(1)

{

printf("\nPress 1. Insert\t 2. Display \t3. Search \t4.Exit \n");

scanf("%d",&opt);

switch(opt)

{

case 1:

insert();

break;

case 2:

display();

break;

case 3:

search();

break;

case 4:exit(0);

}

}

}

65)TOWERS OF HANOI

#include<stdio.h>

void TH(int,char,char,char);

void main()

{

int n;

printf("Enter the number of plates: ");

scanf("%d",&n);

TH(n,'X','Y','Z');

}

void TH(int n,char a,char b,char c)

{

if(n>=1)

{

TH(n-1,a,c,b);

printf("\n%c -> %c",a,c);

TH(n-1,b,a,c);

}

}

65)SEGREGATE 0’S AND 1’S IN AN ARRAY

#include <stdio.h>

void segregate\_01(int arr[], int n)

{

int count\_0 = 0, count\_1 = 0;

for (int i = 0; i < n; i++)

{

if (arr[i] == 0)

{

count\_0++;

}

else

{

count\_1++;

}

}

for (int i = 0; i < count\_0; i++)

{

printf("0 ");

}

**67)SORTING OF ARRAY**

#include<stdio.h>

int main()

{

int a[6]= {12,5,10,9,7,6};

int temp;

int i, j;

printf("Before Sorting ");

for(i=0; i<6; i++)

{

printf("%d ",a[i]);

}

for(i=0; i<6; i++)

{

for(j=i+1; j<6; j++) { if(a[i]>a[j])

{

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

}

}

printf("\nAfter Sorting ");

for(i=0; i<6; i++)

{

printf("%d ",a[i]);

}

return 0;

}

for (int i = count\_0; i < n; i++)

{

printf("1 ");

}

}

int main()

{

int arr[] = {0, 1, 0, 0, 1, 1, 1, 0, 1, 1};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Array: ");

for (int i=0;i < n;i++)

{

printf("%d ",arr[i]);

}

printf("\nSegregated array: ");

segregate\_01(arr,n);

return 0;

}

68)ARRAY ROTATION

#include<stdio.h>

void leftRotatebyOne(int arr[], int n)

{

int temp = arr[0], i;

for (i = 0; i < n - 1; i++)

arr[i] = arr[i + 1];

arr[n-1] = temp;

}

void leftRotate(int arr[], int k, int n)

{

for (int i = 0; i < k; i++)

leftRotatebyOne(arr, n);

}

int main()

{

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

int n = sizeof(arr)/sizeof(arr[0]);

leftRotate(arr, 3, n);

for (int i = 0; i < n; i++)

printf("%d ", arr[i]);

return 0;

}