**Question # 4**

Write code to implement quick sort. Derive time complexity of quick sort in best and worst case.

#include <iostream>

using namespace std;

void print(int \*a, int n)

{

int i=0;

while(i<n){

cout<<a[i]<<",";

i++;

}

}

void swap(int i,int j, int \*a){

int temp = a[i];

a[i] = a[j];

a[j] = temp;

}

void quicksort(int \*arr, int left, int right){

int min = (left+right)/2;

cout<<"QS:"<<left<<","<<right<<"\n";

int i = left;

int j = right;

int pivot = arr[min];

while(left<j || i<right)

{

while(arr[i]<pivot)

i++;

while(arr[j]>pivot)

j--;

if(i<=j){

swap(i,j,arr);

i++;

j--;

}

else{

if(left<j)

quicksort(arr, left, j);

if(i<right)

quicksort(arr,i,right);

return;

}

}

}

int main() {

int arr[8] = {110, 5, 10,3 ,22, 100, 1, 23};

quicksort(arr, 0, (sizeof(arr)/sizeof(arr[0]))-1);

print(arr, (sizeof(arr)/sizeof(arr[0])));

return 0;

}

**Question # 29 :**

Write code for the following functions w.r.t. AVL tree:

(i) Rotate Left

(ii) Rotate Right

#include<stdio.h>

#include<stdlib.h>

struct Node

{

int key;

struct Node \*left;

struct Node \*right;

int height;

};

int max(int a, int b);

int height(struct Node \*N)

{

if (N == NULL)

return 0;

return N->height;

}

int max(int a, int b)

{

return (a > b)? a : b;

}

struct Node\* newNode(int key)

{

struct Node\* node = (struct Node\*)

malloc(sizeof(struct Node));

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1;

return(node);

}

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

{

struct Node \*x = y->left;

struct Node \*T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->left), height(y->right))+1;

x->height = max(height(x->left), height(x->right))+1;

return x;

}

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

{

struct Node \*y = x->right;

struct Node \*T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->left), height(x->right))+1;

y->height = max(height(y->left), height(y->right))+1;

return y;

}

// Get Balance factor of node N

int getBalance(struct Node \*N)

{

if (N == NULL)

return 0;

return height(N->left) - height(N->right);

}

// Recursive function to insert key in subtree rooted

// with node and returns new root of subtree.

struct Node\* insert(struct Node\* node, int key)

{

/\* 1. Perform the normal BST insertion \*/

if (node == NULL)

return(newNode(key));

if (key < node->key)

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

else if (key > node->key)

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

else // Equal keys are not allowed in BST

return node;

/\* 2. Update height of this ancestor node \*/

node->height = 1 + max(height(node->left),

height(node->right));

/\* 3. Get the balance factor of this ancestor

node to check whether this node became

unbalanced \*/

int balance = getBalance(node);

// If this node becomes unbalanced, then

// there are 4 cases

// Left Left Case

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

return rightRotate(node);

// Right Right Case

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

return leftRotate(node);

// Left Right Case

if (balance > 1 && key > node->left->key)

{

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

return rightRotate(node);

}

// Right Left Case

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

{

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

return leftRotate(node);

}

/\* return the (unchanged) node pointer \*/

return node;

}

// A utility function to print preorder traversal

// of the tree.

// The function also prints height of every node

void preOrder(struct Node \*root)

{

if(root != NULL)

{

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

preOrder(root->left);

preOrder(root->right);

}

}

/\* Drier program to test above function\*/

int main()

{

struct Node \*root = NULL;

/\* Constructing tree given in the above figure \*/

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 40);

root = insert(root, 50);

root = insert(root, 25);

/\* The constructed AVL Tree would be

30

/ \

20 40

/ \ \

10 25 50

\*/

printf("Preorder traversal of the constructed AVL"

" tree is \n");

preOrder(root);

return 0;

}

**Question # 3**

Write a program to find k smallest element in an array num[n], where n and k are given as input from user.

// A C++ program to find k'th smallest element using min heap

#include<iostream>

#include<climits>

using namespace std;

// Prototype of a utility function to swap two integers

void swap(int \*x, int \*y);

// A class for Min Heap

class MinHeap

{

int \*harr; // pointer to array of elements in heap

int capacity; // maximum possible size of min heap

int heap\_size; // Current number of elements in min heap

public:

MinHeap(int a[], int size); // Constructor

void MinHeapify(int i); //To minheapify subtree rooted with index i

int parent(int i) { return (i-1)/2; }

int left(int i) { return (2\*i + 1); }

int right(int i) { return (2\*i + 2); }

int extractMin(); // extracts root (minimum) element

int getMin() { return harr[0]; } // Returns minimum

};

MinHeap::MinHeap(int a[], int size)

{

heap\_size = size;

harr = a; // store address of array

int i = (heap\_size - 1)/2;

while (i >= 0)

{

MinHeapify(i);

i--;

}

}

// Method to remove minimum element (or root) from min heap

int MinHeap::extractMin()

{

if (heap\_size == 0)

return INT\_MAX;

// Store the minimum vakue.

int root = harr[0];

// If there are more than 1 items, move the last item to root

// and call heapify.

if (heap\_size > 1)

{

harr[0] = harr[heap\_size-1];

MinHeapify(0);

}

heap\_size--;

return root;

}

// A recursive method to heapify a subtree with root at given index

// This method assumes that the subtrees are already heapified

void MinHeap::MinHeapify(int i)

{

int l = left(i);

int r = right(i);

int smallest = i;

if (l < heap\_size && harr[l] < harr[i])

smallest = l;

if (r < heap\_size && harr[r] < harr[smallest])

smallest = r;

if (smallest != i)

{

swap(&harr[i], &harr[smallest]);

MinHeapify(smallest);

}

}

// A utility function to swap two elements

void swap(int \*x, int \*y)

{

int temp = \*x;

\*x = \*y;

\*y = temp;

}

// Function to return k'th smallest element in a given array

int kthSmallest(int arr[], int n, int k)

{

// Build a heap of n elements: O(n) time

MinHeap mh(arr, n);

// Do extract min (k-1) times

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

mh.extractMin();

// Return root

return mh.getMin();

}

// Driver program to test above methods

int main()

{

int arr[] = {12, 3, 5, 7, 19};

int n = sizeof(arr)/sizeof(arr[0]), k = 2;

cout << "K'th smallest element is " << kthSmallest(arr, n, k);

return 0;

}

**Question # 5**

Write code to implement shell sort. Derive time complexity of quick sort in best and worst case.

// C++ implementation of Shell Sort

#include <iostream>

using namespace std;

/\* function to sort arr using shellSort \*/

int shellSort(int arr[], int n)

{

// Start with a big gap, then reduce the gap

for (int gap = n/2; gap > 0; gap /= 2)

{

// Do a gapped insertion sort for this gap size.

// The first gap elements a[0..gap-1] are already in gapped order

// keep adding one more element until the entire array is

// gap sorted

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

{

// add a[i] to the elements that have been gap sorted

// save a[i] in temp and make a hole at position i

int temp = arr[i];

// shift earlier gap-sorted elements up until the correct

// location for a[i] is found

int j;

for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)

arr[j] = arr[j - gap];

// put temp (the original a[i]) in its correct location

arr[j] = temp;

}

}

return 0;

}

void printArray(int arr[], int n)

{

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

cout << arr[i] << " ";

}

int main()

{

int arr[] = {12, 34, 54, 2, 3}, i;

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

cout << "Array before sorting: \n";

printArray(arr, n);

shellSort(arr, n);

cout << "\nArray after sorting: \n";

printArray(arr, n);

return 0;

}

**Question # 6**

Write code to implement radix sort. Derive time complexity of quick sort in best and worst case.

// C++ implementation of Radix Sort

#include<iostream>

using namespace std;

// A utility function to get maximum value in arr[]

int getMax(int arr[], int n)

{

int mx = arr[0];

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

if (arr[i] > mx)

mx = arr[i];

return mx;

}

// A function to do counting sort of arr[] according to

// the digit represented by exp.

void countSort(int arr[], int n, int exp)

{

int output[n]; // output array

int i, count[10] = {0};

// Store count of occurrences in count[]

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

count[ (arr[i]/exp)%10 ]++;

// Change count[i] so that count[i] now contains actual

// position of this digit in output[]

for (i = 1; i < 10; i++)

count[i] += count[i - 1];

// Build the output array

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

{

output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];

count[ (arr[i]/exp)%10 ]--;

}

// Copy the output array to arr[], so that arr[] now

// contains sorted numbers according to current digit

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

arr[i] = output[i];

}

// The main function to that sorts arr[] of size n using

// Radix Sort

void radixsort(int arr[], int n)

{

// Find the maximum number to know number of digits

int m = getMax(arr, n);

// Do counting sort for every digit. Note that instead

// of passing digit number, exp is passed. exp is 10^i

// where i is current digit number

for (int exp = 1; m/exp > 0; exp \*= 10)

countSort(arr, n, exp);

}

**Question # 7**

Write code to implement merge sort. Derive time complexity of quick sort in best and worst case.

/\* C program for Merge Sort \*/

#include<stdlib.h>

#include<stdio.h>

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

void merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

/\* create temp arrays \*/

int L[n1], R[n2];

/\* Copy data to temp arrays L[] and R[] \*/

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1+ j];

/\* Merge the temp arrays back into arr[l..r]\*/

i = 0; // Initial index of first subarray

j = 0; // Initial index of second subarray

k = l; // Initial index of merged subarray

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

arr[k] = L[i];

i++;

}

else

{

arr[k] = R[j];

j++;

}

k++;

}

/\* Copy the remaining elements of L[], if there

are any \*/

while (i < n1)

{

arr[k] = L[i];

i++;

k++;

}

/\* Copy the remaining elements of R[], if there

are any \*/

while (j < n2)

{

arr[k] = R[j];

j++;

k++;

}

}

/\* l is for left index and r is right index of the

sub-array of arr to be sorted \*/

void mergeSort(int arr[], int l, int r)

{

if (l < r)

{

// Same as (l+r)/2, but avoids overflow for

// large l and h

int m = l+(r-l)/2;

// Sort first and second halves

mergeSort(arr, l, m);

mergeSort(arr, m+1, r);

merge(arr, l, m, r);

}

}

/\* UTILITY FUNCTIONS \*/

/\* Function to print an array \*/

void printArray(int A[], int size)

{

int i;

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

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

printf("\n");

}

/\* Driver program to test above functions \*/

int main()

{

int arr[] = {12, 11, 13, 5, 6, 7};

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

printf("Given array is \n");

printArray(arr, arr\_size);

mergeSort(arr, 0, arr\_size - 1);

printf("\nSorted array is \n");

printArray(arr, arr\_size);

return 0;

}

**Question # 14**

Write a program to read 10 integers from keyboard and store them in the file 'My File'.

#include <stdio.h>

int main()

{

char name[50];

int marks, i, num;

printf("Enter 10 numbers");

scanf("%d", &num);

FILE \*fptr;

fptr = (fopen("C:\\student.txt", "w"));

if(fptr == NULL)

{

printf("Error!");

}

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

{

printf("For student%d\nEnter name: ", i+1);

scanf("%s", name);

printf("Enter marks: ");

scanf("%d", &marks);

fprintf(fptr,"\nName: %s \nMarks=%d \n", name, marks);

}

fclose(fptr);

return 0;

}

**Question # 16**

Describe circular queues with their applications. Write program to implement circular queues.

#include<iostream>

#include<conio.h>

using namespace std;

class cqueue

{

private :

int \*arr ;

int front, rear ;

int MAX;

public :

cqueue( int maxsize = 10 ) ;

void addq ( int item ) ;

int delq( ) ;

void display( ) ;

} ;

cqueue :: cqueue( int maxsize )

{

MAX = maxsize ;

arr = new int [ MAX ];

front = rear = -1 ;

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

arr[i] = 0 ;

}

void cqueue :: addq ( int item )

{

if ( ( rear + 1 ) % MAX == front )

{

cout << "\nQueue is full" ;

return ;

}

rear = ( rear + 1 ) % MAX;

arr[rear] = item ;

if ( front == -1 )

front = 0 ;

}

int cqueue :: delq( )

{

int data ;

if ( front == -1 )

{

cout << "\nQueue is empty" ;

return NULL ;

}

data = arr[front] ;

arr[front] = 0 ;

if ( front == rear )

{

front = -1 ;

rear = -1 ;

}

else

front = ( front + 1 ) % MAX;

return data ;

}

void cqueue :: display( )

{

cout << endl ;

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

cout << arr[i] << " " ;

cout << endl ;

}

void main( )

{

cqueue a ( 10 ) ;

a.addq ( 14 ) ;

a.addq ( 22 ) ;

a.addq ( 13 ) ;

a.addq ( -6 ) ;

a.addq ( 25 ) ;

cout << "\nElements in the circular queue: " ;

a.display( ) ;

int i = a.delq( ) ;

cout << "Item deleted: " << i ;

i = a.delq( ) ;

cout << "\nItem deleted: " << i ;

cout << "\nElements in the circular queue after deletion: " ;

a.display( ) ;

a.addq ( 21 ) ;

a.addq ( 17 ) ;

a.addq ( 18 ) ;

a.addq ( 9 ) ;

a.addq ( 20 ) ;

cout << "Elements in the circular queue after addition: " ;

a.display( ) ;

a.addq ( 32 ) ;

cout << "Elements in the circular queue after addition: " ;

a.display( ) ;

}

**Question # 19**

Write a program to create a binary tree with user input and determine the sum of all tree nodes. Also give the algorithm for this.

#include <stdio.h>

#include <stdlib.h>

/\* A binary tree node has data, pointer to left child

and a pointer to right child \*/

struct node

{

int data;

struct node\* left;

struct node\* right;

};

/\* Function to get the count of leaf nodes in a binary tree\*/

unsigned int getLeafCount(struct node\* node)

{

if(node == NULL)

return 0;

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

return 1;

else

return getLeafCount(node->left)+

getLeafCount(node->right);

}

/\* Helper function that allocates a new node with the

given data and NULL left and right pointers. \*/

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);

}

/\*Driver program to test above functions\*/

int main()

{

/\*create a tree\*/

struct node \*root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

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

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

/\*get leaf count of the above created tree\*/

printf("Leaf count of the tree is %d", getLeafCount(root));

getchar();

return 0;

}