|  |  |
| --- | --- |
| **QuickSort on singly linked list**  **// Partitions the list taking the last element as the pivot**  **struct node \*partition(struct node \*head, struct node \*end,**  **struct node \*\*newHead, struct node \*\*newEnd)**  **{**  **struct node \*pivot = end;**  **struct node \*prev = NULL, \*cur = head, \*tail = pivot;**    **// During partition, both the head and end of the list might change**  **// which is updated in the newHead and newEnd variables**  **while (cur != pivot)**  **{**  **if (cur->data < pivot->data)**  **{**  **// First node that has a value less than the pivot - becomes**  **// the new head**  **if ((\*newHead) == NULL)**  **(\*newHead) = cur;**    **prev = cur;**  **cur = cur->next;**  **}**  **else // If cur node is greater than pivot**  **{**  **// Move cur node to next of tail, and change tail**  **if (prev)**  **prev->next = cur->next;**  **struct node \*tmp = cur->next;**  **cur->next = NULL;**  **tail->next = cur;**  **tail = cur;**  **cur = tmp;**  **}**  **}**    **// If the pivot data is the smallest element in the current list,**  **// pivot becomes the head**  **if ((\*newHead) == NULL)**  **(\*newHead) = pivot;**    **// Update newEnd to the current last node**  **(\*newEnd) = tail;**    **// Return the pivot node**  **return pivot;**  **}**      **//here the sorting happens exclusive of the end node**  **struct node \*quickSortRecur(struct node \*head, struct node \*end)**  **{**  **// base condition**  **if (!head || head == end)**  **return head;**    **node \*newHead = NULL, \*newEnd = NULL;**    **// Partition the list, newHead and newEnd will be updated**  **// by the partition function**  **struct node \*pivot = partition(head, end, &newHead, &newEnd);**    **// If pivot is the smallest element - no need to recur for**  **// the left part.**  **if (newHead != pivot)**  **{**  **// Set the node before the pivot node as NULL**  **struct node \*tmp = newHead;**  **while (tmp->next != pivot)**  **tmp = tmp->next;**  **tmp->next = NULL;**    **// Recur for the list before pivot**  **newHead = quickSortRecur(newHead, tmp);**    **// Change next of last node of the left half to pivot**  **tmp = getTail(newHead);**  **tmp->next =  pivot;**  **}**    **// Recur for the list after the pivot element**  **pivot->next = quickSortRecur(pivot->next, newEnd);**    **return newHead;**  **}**    **// The main function for quick sort. This is a wrapper over recursive**  **// function quickSortRecur()**  **void quickSort(struct node \*\*headRef)**  **{**  **(\*headRef) = quickSortRecur(\*headRef, getTail(\*headRef));**  **return;**  **}**  **QUICKSORT 3 WAY**  **/\* This function partitions a[] in three parts**  **a) a[l..i] contains all elements smaller than pivot**  **b) a[i+1..j-1] contains all occurrences of pivot**  **c) a[j..r] contains all elements greater than pivot \*/**  **void partition(int a[], int l, int r, int &i, int &j)**  **{**  **i = l-1, j = r;**  **int p = l-1, q = r;**  **int v = a[r];**    **while (true)**  **{**  **// From left, find the first element greater than**  **// or equal to v. This loop will definitely terminate**  **// as v is last element**  **while (a[++i] < v);**    **// From right, find the first element smaller than or**  **// equal to v**  **while (v < a[--j])**  **if (j == l)**  **break;**    **// If i and j cross, then we are done**  **if (i >= j) break;**    **// Swap, so that smaller goes on left greater goes on right**  **swap(a[i], a[j]);**    **// Move all same left occurrence of pivot to beginning of**  **// array and keep count using p**  **if (a[i] == v)**  **{**  **p++;**  **swap(a[p], a[i]);**  **}**    **// Move all same right occurrence of pivot to end of array**  **// and keep count using q**  **if (a[j] == v)**  **{**  **q--;**  **swap(a[j], a[q]);**  **}**  **}**    **// Move pivot element to its correct index**  **swap(a[i], a[r]);**    **// Move all left same occurrences from beginning**  **// to adjacent to arr[i]**  **j = i-1;**  **for (int k = l; k < p; k++, j--)**  **swap(a[k], a[j]);**    **// Move all right same occurrences from end**  **// to adjacent to arr[i]**  **i = i+1;**  **for (int k = r-1; k > q; k--, i++)**  **swap(a[i], a[k]);**  **}**    **// 3-way partition based quick sort**  **void quicksort(int a[], int l, int r)**  **{**  **if (r <= l) return;**    **int i, j;**    **// Note that i and j are passed as reference**  **partition(a, l, r, i, j);**    **// Recur**  **quicksort(a, l, j);**  **quicksort(a, i, r);**  **}**  **QUICKSORT 3 WAY (Another method Dutch)**  **/\* This function partitions a[] in three parts**  **a) a[l..i] contains all elements smaller than pivot**  **b) a[i+1..j-1] contains all occurrences of pivot**  **c) a[j..r] contains all elements greater than pivot \*/**    **//It uses Dutch National Flag Algorithm**  **void partition(int a[], int low, int high, int &i, int &j)**  **{**  **// To handle 2 elements**  **if (high - low <= 1)**  **{**  **if (a[high] < a[low])**  **swap(&a[high], &a[low]);**  **i = low;**  **j = high;**  **return;**  **}**    **int mid = low;**  **int pivot = a[high];**  **while (mid <= high)**  **{**  **if (a[mid]<pivot)**  **swap(&a[low++], &a[mid++]);**  **else if (a[mid]==pivot)**  **mid++;**  **else if (a[mid]>pivot)**  **swap(&a[mid], &a[high--]);**  **}**    **//update i and j**  **i = low-1;**  **j = mid; //or high-1**  **}**  **ITERATIVE MERGESORT**  **/\* Function to merge the two haves arr[l..m] and arr[m+1..r] of array arr[] \*/**  **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;**  **j = 0;**  **k = l;**  **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++;**  **}**  **}**  **UNION AN DINTERSECTION OF 2 LINKED LIST**  **// C/C++ program to find union and intersection of two unsorted**  **// linked lists**  **#include<stdio.h>**  **#include<stdlib.h>**    **/\* Link list node \*/**  **struct node**  **{**  **int data;**  **struct node\* next;**  **};**    **/\* A utility function to insert a node at the beginning of**  **a linked list\*/**  **void push(struct node\*\* head\_ref, int new\_data);**    **/\* A utility function to check if given data is present in a list \*/**  **bool isPresent(struct node \*head, int data);**    **/\* Function to get union of two linked lists head1 and head2 \*/**  **struct node \*getUnion(struct node \*head1, struct node \*head2)**  **{**  **struct node \*result = NULL;**  **struct node \*t1 = head1, \*t2 = head2;**    **// Insert all elements of list1 to the result list**  **while (t1 != NULL)**  **{**  **push(&result, t1->data);**  **t1 = t1->next;**  **}**    **// Insert those elements of list2 which are not**  **// present in result list**  **while (t2 != NULL)**  **{**  **if (!isPresent(result, t2->data))**  **push(&result, t2->data);**  **t2 = t2->next;**  **}**    **return result;**  **}**    **/\* Function to get intersection of two linked lists**  **head1 and head2 \*/**  **struct node \*getIntersection(struct node \*head1,**  **struct node \*head2)**  **{**  **struct node \*result = NULL;**  **struct node \*t1 = head1;**    **// Traverse list1 and search each element of it in**  **// list2. If the element is present in list 2, then**  **// insert the element to result**  **while (t1 != NULL)**  **{**  **if (isPresent(head2, t1->data))**  **push (&result, t1->data);**  **t1 = t1->next;**  **}**    **return result;**  **}**    **/\* A utility function to insert a node at the begining of a linked list\*/**  **void push (struct node\*\* head\_ref, int new\_data)**  **{**  **/\* allocate node \*/**  **struct node\* new\_node =**  **(struct node\*) malloc(sizeof(struct node));**    **/\* put in the data \*/**  **new\_node->data = new\_data;**    **/\* link the old list off the new node \*/**  **new\_node->next = (\*head\_ref);**    **/\* move the head to point to the new node \*/**  **(\*head\_ref) = new\_node;**  **}**    **/\* A utility function to print a linked list\*/**  **void printList (struct node \*node)**  **{**  **while (node != NULL)**  **{**  **printf ("%d ", node->data);**  **node = node->next;**  **}**  **}**    **/\* A utility function that returns true if data is**  **present in linked list else return false \*/**  **bool isPresent (struct node \*head, int data)**  **{**  **struct node \*t = head;**  **while (t != NULL)**  **{**  **if (t->data == data)**  **return 1;**  **t = t->next;**  **}**  **return 0;**  **}**  **CUCKOO REHASH**  **void place(int key, int tableID, int cnt, int n)**  **{**  **if (cnt==n)**  **{**  **printf("%d unpositioned\n", key);**  **printf("Cycle present. REHASH.\n");**  **return;**  **}**  **for (int i=0; i<ver; i++)**  **{**  **pos[i] = hash(i+1, key);**  **if (hashtable[i][pos[i]] == key)**  **return;**  **}**    **if (hashtable[tableID][pos[tableID]]!=INT\_MIN)**  **{**  **int dis = hashtable[tableID][pos[tableID]];**  **hashtable[tableID][pos[tableID]] = key;**  **place(dis, (tableID+1)%ver, cnt+1, n);**  **}**  **else //else: place the new key in its position**  **hashtable[tableID][pos[tableID]] = key;**  **}**    **/\* function to print hash table contents \*/**  **void printTable()**  **{**  **printf("Final hash tables:\n");**    **for (int i=0; i<ver; i++, printf("\n"))**  **for (int j=0; j<MAXN; j++)**  **(hashtable[i][j]==INT\_MIN)? printf("- "):**  **printf("%d ", hashtable[i][j]);**    **printf("\n");**  **}**  **void cuckoo(int keys[], int n)**  **{**  **for (int i=0, cnt=0; i<n; i++, cnt=0)**  **place(keys[i], 0, cnt, n);**  **printTable();**  **}**  **REARRANGE CHARACTERS IN STRINGS(PRIOIRTY QUEUE)**  **struct Key**  **{**  **int freq; // store frequency of character**  **char ch;**    **// function for priority\_queue to store Key**  **// according to freq**  **bool operator<(const Key &k) const**  **{**  **return freq < k.freq;**  **}**  **};**    **// Function to rearrange character of a string**  **// so that no char repeat twice**  **void rearrangeString(string str)**  **{**  **int n = str.length();**    **// Store frequencies of all characters in string**  **int count[MAX\_CHAR] = {0};**  **for (int i = 0 ; i < n ; i++)**  **count[str[i]-'a']++;**    **// Insert all characters with their frequencies**  **// into a priority\_queue**  **priority\_queue< Key > pq;**  **for (char c = 'a' ; c <= 'z' ; c++)**  **if (count[c-'a'])**  **pq.push( Key { count[c-'a'], c} );**    **// 'str' that will store resultant value**  **str = "" ;**    **// work as the previous visited element**  **// initial previous element be. ( '#' and**  **// it's frequency '-1' )**  **Key prev {-1, '#'} ;**    **// traverse queue**  **while (!pq.empty())**  **{**  **// pop top element from queue and add it**  **// to string.**  **Key k = pq.top();**  **pq.pop();**  **str = str + k.ch;**    **// IF frequency of previous character is less**  **// than zero that means it is useless, we**  **// need not to push it**  **if (prev.freq > 0)**  **pq.push(prev);**    **// make current character as the previous 'char'**  **// decrease frequency by 'one'**  **(k.freq)--;**  **prev = k;**  **}**    **// If length of the resultant string and original**  **// string is not same then string is not valid**  **if (n != str.length())**  **cout << " Not valid String " << endl;**    **else // valid string**  **cout << str << endl;**  **}**  **OPENING FILE**  **FILE \*fopen( const char \* filename, const char \* mode );**  **int fclose( FILE \*fp );**  **int fputc( int c, FILE \*fp );**  **int fputs( const char \*s, FILE \*fp );**  **int fgetc( FILE \* fp );**  **char \*fgets( char \*buf, int n, FILE \*fp );**  **Example:**  **FILE \*fp;**  **char buff[255];**  **fp = fopen("/tmp/test.txt", "r");**  **fscanf(fp, "%s", buff);**  **printf("1 : %s\n", buff );**  **fgets(buff, 255, (FILE\*)fp);**  **printf("2: %s\n", buff );**    **fgets(buff, 255, (FILE\*)fp);**  **printf("3: %s\n", buff );**  **fclose(fp);**  **fprintf(fp, "This is testing for fprintf...\n");**  **fputs("This is testing for fputs...\n", fp);** | **QuickSort on Doubly Linked List /\* Considers last element as pivot, places the pivot element at its**  **correct position in sorted array, and places all smaller (smaller than**  **pivot) to left of pivot and all greater elements to right of pivot \*/**  **node\* partition(node \*l, node \*h)**  **{**  **// set pivot as h element**  **int x  = h->data;**    **// similar to i = l-1 for array implementation**  **node \*i = l->prev;**    **// Similar to "for (int j = l; j <= h- 1; j++)"**  **for (node \*j = l; j != h; j = j->next)**  **{**  **if (j->data <= x)**  **{**  **// Similar to i++ for array**  **i = (i == NULL)? l : i->next;**    **swap(&(i->data), &(j->data));**  **}**  **}**  **i = (i == NULL)? l : i->next; // Similar to i++**  **swap(&(i->data), &(h->data));**  **return i;**  **}**    **/\* A recursive implementation of quicksort for linked list \*/**  **void \_quickSort(struct node\* l, struct node \*h)**  **{**  **if (h != NULL && l != h && l != h->next)**  **{**  **struct node \*p = partition(l, h);**  **\_quickSort(l, p->prev);**  **\_quickSort(p->next, h);**  **}**  **}**    **// The main function to sort a linked list. It mainly calls \_quickSort()**  **void quickSort(struct node \*head)**  **{**  **// Find last node**  **struct node \*h = lastNode(head);**    **// Call the recursive QuickSort**  **\_quickSort(head, h);**  **}**  **QUICKSORT ITERATIVE**  **/\* A[] --> Array to be sorted,**  **l  --> Starting index,**  **h  --> Ending index \*/**  **void quickSortIterative (int arr[], int l, int h)**  **{**  **// Create an auxiliary stack**  **int stack[ h - l + 1 ];**    **// initialize top of stack**  **int top = -1;**    **// push initial values of l and h to stack**  **stack[ ++top ] = l;**  **stack[ ++top ] = h;**    **// Keep popping from stack while is not empty**  **while ( top >= 0 )**  **{**  **// Pop h and l**  **h = stack[ top-- ];**  **l = stack[ top-- ];**    **// Set pivot element at its correct position**  **// in sorted array**  **int p = partition( arr, l, h );**    **// If there are elements on left side of pivot,**  **// then push left side to stack**  **if ( p-1 > l )**  **{**  **stack[ ++top ] = l;**  **stack[ ++top ] = p - 1;**  **}**    **// If there are elements on right side of pivot,**  **// then push right side to stack**  **if ( p+1 < h )**  **{**  **stack[ ++top ] = p + 1;**  **stack[ ++top ] = h;**  **}**  **}**  **}**  **DELETE(ND)- BST --- SLIDES**  **BinTree deleteNE(BinTree nd)**  **{ /\*Precondition: nd contains the element to be deleted**  **if (nd->right==NULL && nd->left==NULL) { free(nd); return NULL;**  **} else if (nd->right==NULL) { temp=nd->left; free(nd); return temp;**  **} else if (nd->left==NULL) { temp=nd->right; free(nd); return temp;**  **} else { par=nd; suc=nd->right;**  **while (suc->left!=NULL) { par=suc; suc=suc->left; }**  **/\* Postcondition: suc points to in-order successor of nd \*/**  **nd->rootVal = suc->rootVal;**  **if (par->left==suc) { par->left= suc->right; }**  **else /\* par->right==suc \*/ { par->right= suc->right; }**  **free(suc); return nd;**  **}**  **}**  **Merge K Sorted array**  **// This function takes an array of arrays as an argument and**  **// All arrays are assumed to be sorted. It merges them together**  **// and prints the final sorted output.**  **int \*mergeKArrays(int arr[][n], int k)**  **{**  **int \*output = new int[n\*k];  // To store output array**    **// Create a min heap with k heap nodes.  Every heap node**  **// has first element of an array**  **MinHeapNode \*harr = new MinHeapNode[k];**  **for (int i = 0; i < k; i++)**  **{**  **harr[i].element = arr[i][0]; // Store the first element**  **harr[i].i = i;  // index of array**  **harr[i].j = 1;  // Index of next element to be stored from array**  **}**  **MinHeap hp(harr, k); // Create the heap**    **// Now one by one get the minimum element from min**  **// heap and replace it with next element of its array**  **for (int count = 0; count < n\*k; count++)**  **{**  **// Get the minimum element and store it in output**  **MinHeapNode root = hp.getMin();**  **output[count] = root.element;**    **// Find the next elelement that will replace current**  **// root of heap. The next element belongs to same**  **// array as the current root.**  **if (root.j < n)**  **{**  **root.element = arr[root.i][root.j];**  **root.j += 1;**  **}**  **// If root was the last element of its array**  **else root.element =  INT\_MAX; //INT\_MAX is for infinite**    **// Replace root with next element of array**  **hp.replaceMin(root);**  **}**    **return output;**  **}**  **RED BLACK TREES**  **#include <stdio.h>  #include <stdlib.h>  enum nodeColor {         RED,         BLACK  };   struct rbNode {         int data, color;         struct rbNode \*link[2];   };   struct rbNode \*root = NULL;   struct rbNode \* createNode(int data) {         struct rbNode \*newnode;         newnode = (struct rbNode \*)malloc(sizeof(struct rbNode));         newnode->data = data;         newnode->color = RED;         newnode->link[0] = newnode->link[1] = NULL;         return newnode;   }**  **void insertion (int data) {**  **struct rbNode \*stack[98], \*ptr, \*newnode, \*xPtr, \*yPtr;**  **int dir[98], ht = 0, index;**  **ptr = root;**  **if (!root) {**  **root = createNode(data);**  **return;**  **}**  **stack[ht] = root;**  **dir[ht++] = 0;**  **/\* find the place to insert the new node \*/**  **while (ptr != NULL) {**  **if (ptr->data == data) {**  **printf("Duplicates Not Allowed!!\n");**  **return;**  **}**  **index = (data - ptr->data) > 0 ? 1 : 0;**  **stack[ht] = ptr;**  **ptr = ptr->link[index];**  **dir[ht++] = index;**  **}**  **/\* insert the new node \*/**  **stack[ht - 1]->link[index] = newnode = createNode(data);**  **while ((ht >= 3) && (stack[ht - 1]->color == RED)) {**  **if (dir[ht - 2] == 0) {**  **yPtr = stack[ht - 2]->link[1];**  **if (yPtr != NULL && yPtr->color == RED) {**  **stack[ht - 2]->color = RED;**  **stack[ht - 1]->color = yPtr->color = BLACK;**  **ht = ht -2;**  **} else {**  **if (dir[ht - 1] == 0) {**  **yPtr = stack[ht - 1];**  **} else {**  **xPtr = stack[ht - 1];**  **yPtr = xPtr->link[1];**  **xPtr->link[1] = yPtr->link[0];**  **yPtr->link[0] = xPtr;**  **stack[ht - 2]->link[0] = yPtr;**  **}**  **xPtr = stack[ht - 2];**  **xPtr->color = RED;**  **yPtr->color = BLACK;**  **xPtr->link[0] = yPtr->link[1];**  **yPtr->link[1] = xPtr;**  **if (xPtr == root) {**  **root = yPtr;**  **} else {**  **stack[ht - 3]->link[dir[ht - 3]] = yPtr;**  **}**  **break;**  **}**  **} else {**  **yPtr = stack[ht - 2]->link[0];**  **if ((yPtr != NULL) && (yPtr->color == RED)) {**  **stack[ht - 2]->color = RED;**  **stack[ht - 1]->color = yPtr->color = BLACK;**  **ht = ht - 2;**  **} else {**  **if (dir[ht - 1] == 1) {**  **yPtr = stack[ht - 1];**  **} else {**  **xPtr = stack[ht - 1];**  **yPtr = xPtr->link[0];**  **xPtr->link[0] = yPtr->link[1];**  **yPtr->link[1] = xPtr;**  **stack[ht - 2]->link[1] = yPtr;**  **}**  **xPtr = stack[ht - 2];**  **yPtr->color = BLACK;**  **xPtr->color = RED;**  **xPtr->link[1] = yPtr->link[0];**  **yPtr->link[0] = xPtr;**  **if (xPtr == root) {**  **root = yPtr;**  **} else {**  **stack[ht - 3]->link[dir[ht - 3]] = yPtr;**  **}**  **break;**  **}**  **}**  **}**  **root->color = BLACK;**  **}**  **void deletion(int data) {**  **struct rbNode \*stack[98], \*ptr, \*xPtr, \*yPtr;**  **struct rbNode \*pPtr, \*qPtr, \*rPtr;**  **int dir[98], ht = 0, diff, i;**  **enum nodeColor color;**  **if (!root) {**  **printf("Tree not available\n");**  **return;**  **}**  **ptr = root;**  **while (ptr != NULL) {**  **if ((data - ptr->data) == 0)**  **break;**  **diff = (data - ptr->data) > 0 ? 1 : 0;**  **stack[ht] = ptr;**  **dir[ht++] = diff;**  **ptr = ptr->link[diff];**  **}**  **if (ptr->link[1] == NULL) {**  **if ((ptr == root) && (ptr->link[0] == NULL)) {**  **free(ptr);**  **root = NULL;**  **} else if (ptr == root) {**  **root = ptr->link[0];**  **free(ptr);**  **} else {**  **stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];**  **}**  **} else {**  **xPtr = ptr->link[1];**  **if (xPtr->link[0] == NULL) {**  **xPtr->link[0] = ptr->link[0];**  **color = xPtr->color;**  **xPtr->color = ptr->color;**  **ptr->color = color;**  **if (ptr == root) {**  **root = xPtr;**  **} else {**  **stack[ht - 1]->link[dir[ht - 1]] = xPtr;**  **}**  **dir[ht] = 1;**  **stack[ht++] = xPtr;**  **} else {**  **/\* deleting node with 2 children \*/**  **i = ht++;**  **while (1) {**  **dir[ht] = 0;**  **stack[ht++] = xPtr;**  **yPtr = xPtr->link[0];**  **if (!yPtr->link[0])**  **break;**  **xPtr = yPtr;**  **}**  **dir[i] = 1;**  **stack[i] = yPtr;**  **if (i > 0)**  **stack[i - 1]->link[dir[i - 1]] = yPtr;**  **yPtr->link[0] = ptr->link[0];**  **xPtr->link[0] = yPtr->link[1];**  **yPtr->link[1] = ptr->link[1];**  **if (ptr == root) {**  **root = yPtr;**  **}**  **color = yPtr->color;**  **yPtr->color = ptr->color;**  **ptr->color = color;**  **}**  **}**  **if (ht < 1)**  **return;**  **if (ptr->color == BLACK) {**  **while (1) {**  **pPtr = stack[ht - 1]->link[dir[ht - 1]];**  **if (pPtr && pPtr->color == RED) {**  **pPtr->color = BLACK;**  **break;**  **}**  **if (ht < 2)**  **break;**  **if (dir[ht - 2] == 0) {**  **rPtr = stack[ht - 1]->link[1];**  **if (!rPtr)**  **break;**  **if (rPtr->color == RED) {**  **stack[ht - 1]->color = RED;**  **rPtr->color = BLACK;**  **stack[ht - 1]->link[1] = rPtr->link[0];**  **rPtr->link[0] = stack[ht - 1];**  **if (stack[ht - 1] == root) {**  **root = rPtr;**  **} else {**  **stack[ht - 2]->link[dir[ht - 2]] = rPtr;**  **}**  **dir[ht] = 0;**  **stack[ht] = stack[ht - 1];**  **stack[ht - 1] = rPtr;**  **ht++;**  **rPtr = stack[ht - 1]->link[1];**  **}**  **if ( (!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&**  **(!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {**  **rPtr->color = RED;**  **} else {**  **if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {**  **qPtr = rPtr->link[0];**  **rPtr->color = RED;**  **qPtr->color = BLACK;**  **rPtr->link[0] = qPtr->link[1];**  **qPtr->link[1] = rPtr;**  **rPtr = stack[ht - 1]->link[1] = qPtr;**  **}**  **rPtr->color = stack[ht - 1]->color;**  **stack[ht - 1]->color = BLACK;**  **rPtr->link[1]->color = BLACK;**  **stack[ht - 1]->link[1] = rPtr->link[0];**  **rPtr->link[0] = stack[ht - 1];**  **if (stack[ht - 1] == root) {**  **root = rPtr;**  **} else {**  **stack[ht - 2]->link[dir[ht - 2]] = rPtr;**  **}**  **break;**  **}**  **} else {**  **rPtr = stack[ht - 1]->link[0];**  **if (!rPtr)**  **break;**  **if (rPtr->color == RED) {**  **stack[ht - 1]->color = RED;**  **rPtr->color = BLACK;**  **stack[ht - 1]->link[0] = rPtr->link[1];**  **rPtr->link[1] = stack[ht - 1];**  **if (stack[ht - 1] == root) {**  **root = rPtr;**  **} else {**  **stack[ht - 2]->link[dir[ht - 2]] = rPtr;**  **}**  **dir[ht] = 1;**  **stack[ht] = stack[ht - 1];**  **stack[ht - 1] = rPtr;**  **ht++;**  **rPtr = stack[ht - 1]->link[0];**  **}**  **if ( (!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&**  **(!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {**  **rPtr->color = RED;**  **} else {**  **if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {**  **qPtr = rPtr->link[1];**  **rPtr->color = RED;**  **qPtr->color = BLACK;**  **rPtr->link[1] = qPtr->link[0];**  **qPtr->link[0] = rPtr;**  **rPtr = stack[ht - 1]->link[0] = qPtr;**  **}**  **rPtr->color = stack[ht - 1]->color;**  **stack[ht - 1]->color = BLACK;**  **rPtr->link[0]->color = BLACK;**  **stack[ht - 1]->link[0] = rPtr->link[1];**  **rPtr->link[1] = stack[ht - 1];**  **if (stack[ht - 1] == root) {**  **root = rPtr;**  **} else {**  **stack[ht - 2]->link[dir[ht - 2]] = rPtr;**  **}**  **break;**  **}**  **}**  **ht--;**  **}**  **}**  **}**  **void searchElement(int data) {**  **struct rbNode \*temp = root;**  **int diff;**  **while (temp != NULL) {**  **diff = data - temp->data;**  **if (diff > 0) {**  **temp = temp->link[1];**  **} else if (diff < 0) {**  **temp = temp->link[0];**  **} else {**  **printf("Search Element Found!!\n");**  **return;**  **}**  **}**  **printf("Given Data Not Found in RB Tree!!\n");**  **return;**  **}**  **void inorderTraversal(struct rbNode \*node) {**  **if (node) {**  **inorderTraversal(node->link[0]);**  **printf("%d  ", node->data);**  **inorderTraversal(node->link[1]);**  **}**  **return;**  **}**  **STRING.h**  [**char \*strcat(char \*dest, const char \*src)**](https://www.tutorialspoint.com/c_standard_library/c_function_strcat.htm)  [**int strcmp(const char \*str1, const char \*str2)**](https://www.tutorialspoint.com/c_standard_library/c_function_strcmp.htm)  [**char \*strcpy(char \*dest, const char \*src)**](https://www.tutorialspoint.com/c_standard_library/c_function_strcpy.htm)  [**char \*strtok(char \*str, const char \*delim)**](https://www.tutorialspoint.com/c_standard_library/c_function_strtok.htm)  **char str[80] = "This is - www.tutorialspoint.com - website";**  **const char s[2] = "-";**  **char \*token;**    **/\* get the first token \*/**  **token = strtok(str, s);**    **/\* walk through other tokens \*/**  **while( token != NULL )**  **{**  **printf( " %s\n", token );**    **token = strtok(NULL, s);**  **}** |