Name: Shah Siddh Tejaskumar

Reg.NO.: 20BCE1937

Slot: L55 + L56

Assignment (4-6 )  
------------------------------------------------------------------------------------------------

1. Design and develop an algorithm that can perform binary search in the given set of elements. Initially all the elements are present in a linked list and are not in sorted order. You are supposed to handle the DS so that, you can ensure that, the DS can provide a search in O(log n) in average case. [Hint: You can select the Median element of the list to avoid Skews]

**Source Code:**

// in below program the merge sort is performed on list

// and then binary search is performed to find data

#include <iostream>

using namespace std;

class node  // list strurct

{

    public:

        int data;

        node\* next;

};

void mid (node \* cur, node\*\* first, node\*\* second)  // function used to find middle element

{

    node \* slow ;

    node \* fast ;

    slow = cur;

    fast  = cur->next;

    while (fast != NULL)  // slow will be the middle ele. after iterating

    {

        fast = fast->next;

        if (fast != NULL)

        {

            slow = slow->next;

            fast = fast->next;

        }

    }

    \*first = cur;

    \*second = slow->next;

    slow->next = NULL;  // breaking the list from middle

}

node\* merge (node\* first , node\*second)  // merging the sub-list

{

    node\* ans = NULL;

    if (!first)  // base condition

        return second;

    else if (!second)

        return first;

    if (first->data <= second->data)

    {

        ans = first;

        ans->next = merge (first->next, second);

    }

    else

    {

        ans = second;

        ans->next = merge (first, second->next);

    }

    return ans;

}

void mergesort (node\*\* head)  // merge sort

{

    node \* cur = \*head;

    node\* first; node \* second;

    if (!cur || !cur->next)  // if there is only one ele.

        return;

    mid (cur, &first, &second);

    mergesort(&first);

    mergesort(&second);

    \* head = merge (first, second);

}

node\* mid\_search (node \* front , node\* rear)  // function to find the mid element for binary serach

{

    node \*slow = front;

    node \* fast = front->next;

    while (fast != rear)

    {

        fast = fast->next;

        if (fast != rear)

        {

            slow = slow->next;

            fast = fast->next;

        }

    }

    return slow;  // slow = middle ele.

}

void binary\_search (node\* head, int value)

{

    node\* front = head;

    node\* temp = head;

    node\* rear;

    while (temp->next != NULL)

    {

        temp = temp->next;

    }

    rear = temp;

    free (temp);

    while (front->data <= rear->data)

    {

        node \* mid = mid\_search(front, rear);

        if (mid->data = value)

        {

            cout << "data found";

            break;

        }

        if (value > mid->data)

        {

            front = mid->next;

        }

        if (value < mid->data)

        {

            rear = mid;

        }

        if (front == rear || value != mid->data)

        {

            cout << "No such data";

            break;

        }

    }

}

int main()

{

    int n;

    cout << "Enter the size of linked list: ";

    cin >> n;

    node \*block = new node;

    block = (node \*)malloc(n\* sizeof(node));

    for (int i = 0; i < n; i++)

    {

        if (i == n - 1)

        {

            cin >> (block + i)->data;

            (block + i)->next = NULL;

        }

        else

        {

            cin >> (block + i)->data;

            (block + i)->next = (block + i + 1);

        }

    }

    mergesort(&block);

    int search ;

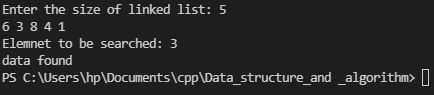
    cout << "Elemnet to be searched: ";

    cin >> search;

    binary\_search (block , search);

    return 0;

}

Sample Output:  


**2**. Implement the single source shortest Path algorithm using dijkstra's algorithm. Your program should be able to receive an adjacency matrix as the input and it should provide the shortest path exists between them.

**Source Code:**

#include <iostream>

#include <vector>

#include <utility>

using namespace std;

int findMinVertex(int \*distance, bool \*visited, int n)   // function to find vertex with min distance

{

    int minVertex = -1;

    for (int i = 0; i < n; i++)

        if (!visited[i] && (minVertex == -1 || distance[i] < distance[minVertex]))

            minVertex = i;

    return minVertex;

}

void dijkstra(int \*\*graph, int n)

{

    int \*distance = new int[n];   // array to maintain all nodes weights

    bool \*visited = new bool[n];  // array to check whether the nodes are visited or not

    for (int i = 0; i < n; i++)

    {

        distance[i] = INT\_MAX;   // initiallizing all distace as infinity except source, done below

        visited[i] = false;      // as initial all nodes will not visited, so false

    }

    int start\_vertex;   // taking the source vertex

    cout << "Enter the start vertex: ";

    cin >> start\_vertex;

    distance[start\_vertex] = 0;  // initiallizing its distance as 0, so to take as first node to be proccessed

    for (int i = 0; i < n - 1; i++)

    {

        int minVertex = findMinVertex(distance, visited, n);   // finding the node with carrying the minimum distance

        visited[minVertex] = true;  // as we visit node, mark in list as true

        for (int j = 0; j < n; j++)  // iterating to relax nodes

        {

            if (graph[minVertex][j] != 0 && !visited[j])  // checking for adjacent edges AND if node visited or not

            {

                int dist = distance[minVertex] + graph[minVertex][j];  // calculate distance from min vertex to that node

                if (dist < distance[j])  // if above distance less than already existing wait, relax it

                {

                    distance[j] = dist;

                }

            }

        }

    }

    for (int i=1 ;i<n; i++)  // preinting the min. distance from strat\_vertex

    {

        cout << distance[i] << " ";

    }

    // freeing the memory

    delete[] visited;

    delete[] distance;

}

int main()

{

    int n; // no\_of\_vertices

    int e; // no\_of\_edges

    cout << "Enter the no. of vertices: ";

    cin >> n;

    cout << "Enter the no. of edges: ";

    cin >> e;

    int \*\*graph = new int \*[n]; // creating the dynamic 2D array

    for (int i = 0; i < n; i++)

    {

        graph[i] = new int[n];

        for (int j = 0; j < n; j++)

        {

            graph[i][j] = 0;

        }

    }

    for (int i = 0; i < e; i++) // creating the graph (using adjancency matrix form)

    {

        int source, dest, weight;

        cout << "Enter the source: ";

        cin >> source;

        cout << "Enter the destination: ";

        cin >> dest;

        cout << "Enter the weight: ";

        cin >> weight;

        graph[source][dest] = weight;

    }

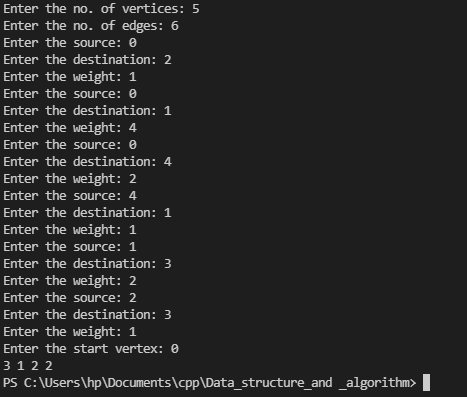
    dijkstra(graph, n);

    for (int i = 0; i < n; i++)

        delete[] graph[i];

    return 0;

}

**Sample Output:  
**

**3**. Write a program to create a subtree from a Binary search tree, by considering the median element of the tree as a root, smallest element as the left child and largest element as the Right child.

**Source Code:**

#include <iostream>

using namespace std;

class tree  // tree structure

{

public:

    int data;

    tree \*left;

    tree \*right;

};

tree\* createNode(int data)  // creating the nodes

{

    tree \*node;

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

    node->data = data;

    node->left = NULL;

    node->right = NULL;

    return node;

}

int count = 0;

int \* arr;

void median (tree\* root)   // func() to find middle ele. in BST

{

    if (root != NULL)

    {

        median(root->left);

        \*(arr + count) = root->data;

        count ++;

        median(root->right);

    }

    cout <<  \*(arr + (count/2)) << endl;

}

tree\* smallest (tree\* root)     // func() to find smallest ele. in BST

{

    while (root->left != NULL){

        root = root->left;

    }

    cout << root->data << endl;

    return root;

}

tree\* largest (tree\* root)  // func() to find largest ele. in BST

{

    while (root->right != NULL){

        root = root->right;

    }

    cout << root->data << endl;

    return root;

}

int main()

{

    tree \*root, \*l, \*r, \*ll, \*lr, \*llr, \*lll, \*rlr, \*rl;

    root = createNode(9);

    l = createNode(7);

    r = createNode(13);

    ll = createNode(2);

    lr = createNode(8);

    llr = createNode(5);

    lll = createNode(1);

    rl = createNode(10);

    rlr = createNode(11);

    // creating a BST

    //      9

    //    /  \

    //   7    13

    //  / \   /

    // 2   8  10

    /// \      \

   //1   5      11

    root->left = l;

    root->right = r;

    l->left = ll;

    l->right = lr;

    ll->right = llr;

    ll->left = lll;

    r->left = rl;

    rl->right = rlr;

    tree\* new\_root;  // root of new tree

    new\_root = createNode (\*(arr + (count/2)));

    cout << new\_root->data;

    new\_root->left = smallest(root); // smallest element of old BST as its left

    new\_root->right = largest (root);  // largest element of old BST as its right

    cout << new\_root->data << " ";

    cout << new\_root->left->data << " ";

    cout << new\_root->right->data << " ";

    return 0;

}