

**Data Structures and Algorithms**

**Lab Assignment 3**

**SUBMITTED BY:**

Hasaan Ahmad SP22-BSE-017

**SUBMITTED TO: Sir Syed Ahmad Qasim**

**Linked List and All its vital methods.**

**Code:**

#include <iostream>

#include <list>

using namespace std;

struct Node

{

    int data;

    Node \*next = nullptr;

};

class LinkedList

{

public:

    Node \*first = nullptr;

    Node \*last = nullptr;

    LinkedList()

    {

        first = nullptr;

    }

    void insert\_end()

    {

        Node \*p;

        p = new Node;

        cout << "Enter the data in node:";

        cin >> p->data;

        if (first == NULL)

            first = last = p;

        else

        {

            last->next = p;

            last = p; /\* assign p to last node \*/

        }

    }

    void insert\_start()

    {

        // take the pointer to hold the address of nodetype record

        Node \*p;

        // allocate runtime memory for new record of nodetype using new operator

        p = new Node;

        cout << "Enter the data in node:";

        cin >> p->data;

        if (first == NULL) /\* l i s t i s empty \*/

                           /\* p becomes first node and first and last pointer will point to same node \*/

            first = last = p;

        else

        {

            p->next = first; /\* link the new node with first node \*/

            first = p;       /\* assign p to last node \*/

        }

    }

    Node \*search(int key)

    {

        Node \*p = first;

        // p = first;

        while (p != NULL && p->data != key)

        {

            p = p->next;

        }

        return p; /\* if p is NULL then value not found \*/

    }

    void insert\_after(int key)

    {

        Node \*p = NULL;

        p = search(key);

        if (p == NULL)

            cout << "value not found";

        else

        {

            Node \*Newnode;

            Newnode = new Node;

            if (p == last)

            {

                last->next = p;

                last = p;

            }

            else

            {

                Newnode->next = p->next;

                p->next = Newnode;

            }

            cout << "New node linked successfully";

        }

    }

    void delete\_first()

    {

        Node \*p;

        if (first == nullptr)

            cout << "\n Linked List is empty";

        else

        { /\* non−empty l i s t \*/

            p = first;

            first = first->next;

            delete (p); /\* free up memory \*/

        }

    }

    void delete\_last()

    {

        Node \*q, \*q1;

        q1 = nullptr;

        q = first;

        if (q == nullptr)

        {

            cout << "\n Linked List is empty";

        }

        else

        {

            while (q != last)

            {           /\* advance towards end \*/

                q1 = q; /\*q1 will follow the q pointer \*/

                q = q->next;

            }

            if (q == first)

            {

                first = last = nullptr;

            }

            else

            { /\* more than one node \*/

                q1->next = nullptr;

                last = q1;

            }

            delete q;

        }

    }

    void remove\_spec(int key)

    {

        Node \*q, \*q1;

        q1 = NULL; /\* i n i t i a l i z e \*/

        q = first;

        /\* search node \*/

        while (q != NULL && q->data != key)

        {

            q1 = q;

            q = q->next;

        }

        if (q == NULL)

        {

            cout << "Not found supplied key";

        }

        else if (q == first && q == last)

        {

            delete q;

            first = last = NULL;

        }

        else if (q == last)

        {

            q1->next = NULL;

            last = q1; /\* make 2nd last node as last node \*/

            delete q;

        }

        else /\* other than f i r s t node and last \*/

        {

            q1->next = q->next;

            delete q;

        }

    }

    void traverse()

    {

        Node \*newNode = first;

        while (newNode != nullptr)

        {

            cout << newNode->data << "  ";

            newNode = newNode->next;

        }

        cout << endl;

    }

    void reversePrint(Node \*p)

    {

        if (p != nullptr)

        {

            reversePrint(p->next);

            cout << p->data << "  ";

        }

    }

    void countOccurence(int key)

    {

        Node \*temp = first;

        int count = 0;

        while (temp != NULL)

        {

            if (temp->data == key)

            {

                count++;

            }

            temp = temp->next;

        }

        cout << "Count of " << key << " is " << count << endl;

    }

};

void traverseThroughHead(Node \*head)

    {

        Node \*temp = head;

        while (temp != NULL)

        {

            cout << temp->data << "  ";

            temp = temp->next;

        }

        cout << endl;

    }

Node \*mergeTwoLists(Node \*l1, Node \*l2)

{

    Node \*head = new Node();

    Node \*temp = head;

    while (l1 != NULL && l2 != NULL)

    {

        if (l1->data < l2->data)

        {

            temp->next = l1;

            l1 = l1->next;

        }

        else

        {

            temp->next = l2;

            l2 = l2->next;

        }

        temp = temp->next;

    }

    if (l1 != NULL)

    {

        temp->next = l1;

    }

    if (l2 != NULL)

    {

        temp->next = l2;

    }

    return head->next;

}

int main()

{

    LinkedList \*l1 = new LinkedList();

    LinkedList \*l2 = new LinkedList();

    l1->insert\_end();

    l1->insert\_end();

    l1->traverse();

    l1->insert\_start();

    l1->insert\_start();

    l1->traverse();

    l1->reversePrint(l1->first);

    cout << endl;

    l2->insert\_end();

    l2->insert\_start();

    l2->delete\_first();

    l2->insert\_end();

    cout<<l2->search(10);

    cout<<endl;

    l2->insert\_start();

    l2->insert\_start();

    l2->insert\_start();

    l2->traverse();

    Node \*l3 = mergeTwoLists(l1->first, l2->first);

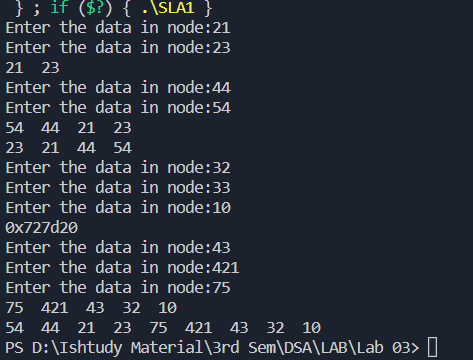
    // Testing merge two linked lists.

    traverseThroughHead(l3);

    return 0;

}

**Output:**

****

**Output 2**

