1.Find is smallString is a part of largerString

#include <iostream>

#include <string>

using namespace std;

int main() {

    string largerString, smallerString;

    cout << "Enter the larger string: ";

    cin >> largerString;

    cout << "Enter the smaller string to check if it's a subset: ";

    cin >> smallerString;

    // Check if smallerString is a subset of largerString

    if (largerString.find(smallerString)) {

        cout << "The smaller string is not subset of the larger string." << endl;

    } else {

        cout << "The smaller string is a subset of the larger string." << endl;

    }

    return 0;

}

1.Basic implementation of Queue

#include <iostream>

#include <stack>

#include <queue>

using namespace std;

void revK\_Queue(int K,queue<int> &Queue){

    if(Queue.empty() || K<=0 || K>Queue.size())

    {

        return ;

    }

    stack<int> myStack;

    for(int i=0 ; i< K ;i++)

    {

        myStack.push(Queue.front());

        Queue.pop();

    }

    while(!myStack.empty())

    {

        Queue.push(myStack.top());

        myStack.pop();

    }

    for(int i=0 ; i<Queue.size()-K ;i++)

    {

        Queue.push(Queue.front());

        Queue.pop();

    }

}

int main()

{

    queue<int> myQueue;

    int n,k;

    cout << "Original queue size: ";

    cin >> n ;

    cout << "number of front elements to be reversed: ";

    cin >> k;

    for(int i=0 ; i< n;i++)

    {

        int element;

        cin >> element ;

        myQueue.push(element);

    }

    revK\_Queue(k,myQueue);

    for(int i=0 ; i < n ;i++)

    {

        cout << myQueue.front() << " ";

        myQueue.pop();

    }

    return 0;

}

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a Node

typedef struct Node {

    int INFO;

    struct Node\* NEXT;

} Node;

// SLL insert�at�beginning

Node\* InsertBeg\_List(Node\* START, int info) {

    Node\* New\_Node = (Node\*)malloc(sizeof(Node)); // Allocate memory for a new node

    if (New\_Node == NULL) {

        printf("Memory allocation failed.\n");

        return START; // Return the original list if memory allocation fails

    }

    New\_Node->INFO = info; // Put info in INFO field of new node

    New\_Node->NEXT = START; // Set the NEXT of the new node to the current START

    START = New\_Node; // Update START to point to the new node

    return START; // Return the updated START

}

//SLL delete at beginning

Node\* DeleteFirstNode(Node\* START) {

    Node\* Temp = START;

    if (START == NULL) {

        printf("Empty Linked List\n");

        return START;

    }

    START = START->NEXT; // Update START to point to the next node

    free(Temp); // Free the memory of the first node

    return START; // Return the updated START

}

//SLL insert�at�the�end

Node\* Append\_List(Node\* START, int info) {

    Node\* New\_Node = (Node\*)malloc(sizeof(Node)); // Allocate memory for a new node

    if (New\_Node == NULL) {

        printf("Memory allocation failed.\n");

        return START; // Return the original list if memory allocation fails

    }

    New\_Node->INFO = info; // Put info in INFO field of new node

    New\_Node->NEXT = NULL; // Make it the last node by keeping NULL in NEXT field

    if (START == NULL) {

        START = New\_Node; // If the list is empty, make the new node the START

    } else {

        Node\* Temp = START;

        while (Temp->NEXT != NULL) {

            Temp = Temp->NEXT;

        }

        Temp->NEXT = New\_Node; // Append the new node at the end

    }

    return START; // Return the updated START

}

//SLL delete last node

Node\* DeleteEnd\_List(Node\* START) {

    Node\* PREV = NULL;

    Node\* Temp = START;

    if (START == NULL) {

        printf("Empty Linked List\n");

        return START;

    }

    if (START->NEXT == NULL) {

        // If there's only one node in the list, delete it

        free(START);

        START = NULL;

        return START;

    }

    while (Temp->NEXT != NULL) {

        PREV = Temp;

        Temp = Temp->NEXT;

    }

    PREV->NEXT = NULL;

    free(Temp);

    return START;

}

//SLL insert at specific�location

Node\* InsertAt\_Given\_Loc(Node\* START, int info, int Loc) {

    Node\* Temp = START;

    Node\* New\_Node;

    if (Loc < 1) {

        printf("Invalid location!\n");

        return START;

    }

    New\_Node = (Node\*)malloc(sizeof(Node)); // Allocate memory for a new node

    if (New\_Node == NULL) {

        printf("Memory allocation failed.\n");

        return START; // Return the original list if memory allocation fails

    }

    New\_Node->INFO = info; // Put info in INFO field of the new node

    if (Loc == 1) {

        New\_Node->NEXT = START; // If Loc is 1, insert at the first position

        START = New\_Node;

    } else {

        int i;

        for (i = 1; i < Loc - 1 && Temp != NULL; i++) {

            Temp = Temp->NEXT;

        }

        if (Temp == NULL) {

            printf("Loc is greater than the number of nodes + 1\n");

            free(New\_Node); // Delete the memory for the new node

        } else {

            New\_Node->NEXT = Temp->NEXT;

            Temp->NEXT = New\_Node;

        }

    }

    return START; // Return the updated START

}

//SLL delete at specific location

Node\* DeleteFromLoc\_List(Node\* START, int Loc) {

    Node\* Prev = START;

    Node\* Temp = START;

    if (START == NULL || Loc <= 0) {

        printf("Empty Linked List or invalid location\n");

        return START;

    }

    if (Loc == 1) {

        START = START->NEXT;

        free(Temp);

        return START;

    }

    int i;

    for (i = 1; i <= Loc - 1 && Temp != NULL; i++) {

        Prev = Temp;

        Temp = Temp->NEXT;

    }

    if (Temp == NULL) {

        printf("Loc is greater than the total number of nodes\n");

        return START;

    }

    Prev->NEXT = Temp->NEXT;

    free(Temp);

    return START;

}

//SLL searching element

Node\* Search\_List(Node\* START, int info) {

    Node\* Temp = START;

    while (Temp != NULL) {

        if (Temp->INFO == info) {

            return Temp; // Return the node if the value is found

        }

        Temp = Temp->NEXT;

    }

    return NULL; // Return NULL if the value is not found in the list

}

//SLL Reversing

Node\* Reverse\_List(Node\* START) {

    Node\* Cnode = START;

    Node\* Pnode = NULL;

    Node\* Nnode;

    if (START == NULL) {

        return START; // Return the empty list if it's already empty

    }

    Cnode = START;

    Nnode = Cnode->NEXT;

    Cnode->NEXT = NULL;

    while (Nnode != NULL) {

        Pnode = Cnode;

        Cnode = Nnode;

        Nnode = Nnode->NEXT;

        Cnode->NEXT = Pnode;

    }

    START = Cnode; // Update START to the new head of the reversed list

    return START;

}

int main() {

    Node\* START = NULL; // Initialize an empty linked list

    // Insert a few nodes at the beginning of the list

    START = InsertBeg\_List(START, 3);

    START = InsertBeg\_List(START, 2);

    START = InsertBeg\_List(START, 1);

    START = Append\_List(START, 30);

    START = Append\_List(START, 20);

    START = Append\_List(START, 10);

    START = InsertAt\_Given\_Loc(START, 1, 1);

    START = InsertAt\_Given\_Loc(START, 3, 2);

    START = InsertAt\_Given\_Loc(START, 2, 2);

    START = DeleteFirstNode(START);

    START = DeleteEnd\_List(START);

    START = DeleteFromLoc\_List(START, 2);

    printf("\n");

    // Searching for a specific value (e.g., info = 2)

    Node\* result = Search\_List(START, 2);

    if (result != NULL) {

        printf("Value found: %d\n", result->INFO);

    } else {

        printf("Value not found\n");

    }

     printf("\n");

    // Reversing the linked list

    START = Reverse\_List(START);

    // Print the linked list

    Node\* current = START;

    while (current != NULL) {

        printf("%d -> ", current->INFO);

        current = current->NEXT;

    }

    printf("NULL\n");

    return 0;

}

/\*

Value found: 2

20 -> 30 -> 3 -> 2 -> 1 -> 2 -> NULL

Process returned 0 (0x0)   execution time : 9.994 s

Press any key to continue.

\*/