**Aim:** Introduction to linked list and its types.

**Introduction:**

A linked list is a linear data structure that consists of a sequence of nodes, where each node contains a data element and a reference (link) to the next node in the sequence. The first node is called the head, and the last node points to null. Linked lists can be used to implement dynamic data structures, where the size of the data structure can change during runtime, unlike arrays, which have a fixed size. Linked lists are useful for many applications, such as implementing stacks, queues, and memory management.

**There are several types of linked lists:**

1. Singly linked list: A linked list in which each node has a single reference to the next node in the list.
2. Doubly linked list: A linked list in which each node has two references, one to the next node and one to the previous node.
3. Circular linked list: A linked list in which the last node points back to the first node, forming a circular loop.
4. Multiply linked list: A linked list in which each node has multiple references to other nodes.
5. Skip list: A linked list in which some nodes have additional references to other nodes, skipping over intermediate nodes for faster access to elements.
6. XOR linked list: A linked list in which each node has an XOR reference to the next and previous node.

These different types of linked lists have different use cases and trade-offs in terms of space and time complexity, making them useful in different situations.

**Application of linked list:**

1. Dynamic memory allocation: Linked lists can be used to manage memory dynamically, allocating and freeing memory as needed.
2. Stack and queue implementation: Linked lists can be used to implement the basic operations of a stack (push, pop) or a queue (enqueue, dequeue) efficiently.
3. Hash tables: Linked lists can be used to implement hash tables, which are used to store and retrieve elements based on a key.
4. Sparse matrices: Linked lists can be used to implement sparse matrices, where most elements are zero, to save space.
5. Graphs: Linked lists can be used to represent graph data structures, where nodes in the linked list represent vertices and the links between nodes represent edges.
6. Polynomials: Linked lists can be used to represent polynomials, with nodes representing terms in the polynomial and links representing the relationships between terms.
7. Operating system memory management: Linked lists can be used in operating systems to keep track of memory allocation and to manage the allocation and freeing of memory.
8. File systems: Linked lists can be used in file systems to keep track of files and the relationships between files.

* Singly Linked Lists:

#include <iostream>

using namespace std;

template <typename T>

class SinglyList

{

class Node

{

public:

T data;

Node \*next;

Node(int data)

{

this->data = data;

this->next = NULL;

}

};

public:

Node \*head = NULL;

int cnt = 0;

bool isEmpty() {

return this->head == NULL ? true : false;

}

int size() {

return this->cnt;

}

void add(T data) {

Node \*n = new Node(data);

if (this->isEmpty())

{

this->head = n;

}

else{

Node \*temp = this->head;

while (temp->next != NULL)

{

temp = temp->next;

}

temp->next = n;

}

cout << "Element Added Successfully!" << endl;

cnt++;

}

void remove(){

Node \*temp = this->head;

while ((temp->next)->next != NULL)

{

temp = temp->next;

}

Node \*n = temp->next;

cout << n->data << " removed from the list" << endl;

delete n;

temp->next = NULL;

this->cnt--;

}

void reverse(){

Node \*prev = nullptr;

Node \*curr = head;

Node \*next = nullptr;

while (curr != nullptr){

next = curr->next;

curr->next = prev;

prev = curr;

curr = next;

}

head = prev;

cout << "List Reversed!" << endl;

}

string toString()

{

if (this->isEmpty())

{

return "List is empty!\n";

}

Node \*temp = head;

string str = "List [";

while (temp != NULL)

{

str = str.append(to\_string(temp->data));

str = str.append(", ");

temp = temp->next;

}

str = str.substr(0, str.length() - 2);

str.append("]\n");

return str;

}

void search(T data)

{

Node \*temp = this->head;

int k = 0;

while (temp != NULL)

{

k++;

if (temp->data == data)

{

cout << "Element found at index " << k - 1 << endl;

return;

}

temp = temp->next;

}

cout << "Element is not found!" << endl;

}

};

void util(){

int choice = 1, data;

SinglyList<int> \*list = new SinglyList<int>();

while (choice == 1 || choice == 2 || choice == 3 || choice == 4 || choice == 5 || choice == 6)

{

cout << "List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}\nchoice : ";

cin >> choice;

switch (choice)

{

case 1:

cout << "Enter Element : ";

cin >> data;

list->add(data);

break;

case 2:

list->remove();

break;

case 3:

cout << list->toString();

break;

case 4:

cout << "Size of List are : " << list->size() << endl;

break;

case 5:

cout << "Enter Element to search: ";

cin >> data;

list->search(data);

break;

case 6:

list->reverse();

break;

}

}

cout << "closing the application..." << endl;

}

int main(int argc, char const \*argv[]){

util();

return 0;

}

| List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 1  Enter Element : 12  Element Added Successfully!  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 1  Enter Element : 32  Element Added Successfully!  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 1  Enter Element : 45  Element Added Successfully!  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 1  Enter Element : 67  Element Added Successfully!  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 3  List [12, 32, 45, 67]  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 6  List Reversed!  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 3  List [67, 45, 32, 12]  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 2  12 removed from the list  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 4  Size of List are : 3  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 9  closing the application... |
| --- |

* Circular Linked List

#include <iostream>

using namespace std;

class Node

{

public:

Node \*next = nullptr;

int data;

Node(int data) : data(data) {}

};

class CircularList

{

Node \*head;

int count = 0;

public:

void add(int data)

{

Node \*newNode = new Node(data);

if (head == nullptr)

{

head = newNode;

}

else

{

Node \*temp = head;

while (temp->next != head)

{

temp = temp->next;

}

temp->next = newNode;

}

newNode->next = head;

count++;

cout << "Element added : " << data << endl;

}

void remove()

{

if (head == nullptr)

{

cout << "empty list" << endl;

return;

}

Node \*oldNode = head, \*temp = head;

while (temp->next != head)

temp = temp->next;

temp->next = head->next;

if(head->next == head)

head = nullptr;

else

head = head->next;

cout << "Element removed : " << oldNode->data << endl;

delete oldNode;

count--;

}

void reverseList()

{

if (!head){

cout << "List is empty!"<<endl;

return;

}

Node \*prev = head, \*curr = head->next, \*temp = head->next;

while (curr != head)

{

temp = curr->next;

curr->next = prev;

prev = curr;

curr = temp;

}

head->next = prev;

head = prev;

cout << "list reversed!" << endl;

}

void size()

{

cout << "size of list : " << this->count << endl;

}

void search(int data) // searching element in a list

{

if(head == nullptr){

cout << "list is empty" << endl;

return;

}

Node \*temp = head;

do{

if(temp->data == data){

cout << "Element is found!"<<endl;

return;

}

temp = temp->next;

}while (temp != head);

cout << "Element Not found!" << endl;

}

void print()

{

if (this->head == nullptr)

{

cout << "list is empty" << endl;

return;

}

cout << "list => ";

Node \*temp = head;

while (true)

{

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

temp = temp->next;

if (temp == head)

break;

};

cout << endl;

}

};

int main(int argc, char const \*argv[])

{

CircularList \*list = new CircularList();

int choice, data;

while (true) {

cout << "List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}\nchoice : ";

cin >> choice;

switch (choice)

{

case 1:

cout << "Enter Element : ";

cin >> data;

list->add(data);

break;

case 2:

list->remove();

break;

case 3:

list->print();

break;

case 4:

list->size();

break;

case 5:

cout << "Enter Element to search: ";

cin >> data;

list->search(data);

break;

case 6:

list->reverseList();

break;

default:

return 0;

}

}

return 0;

}

| List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 1  Enter Element : 23  Element added : 23  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 2  Element removed : 23  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 2  empty list  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 1  Enter Element : 23  Element added : 23  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 1  Enter Element : 43  Element added : 43  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 1  Enter Element : 56  Element added : 56  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 3  list => 23 43 56  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 3  list => 23 43 56  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 6  list reversed!  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 3  list => 56 43 23  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 2  Element removed : 56  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 5  Enter Element to search: 23  Element is found!  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 4  size of list : 2  List Operations {1 : add, 2 : remove, 3 : display, 4 : count, 5 : search, 6 : reverse a list, other : exit}  choice : 45 |
| --- |

* Doubly Linked Lists

#include <iostream>

#include <cstring>

using namespace std;

template <typename T>

class Node{

public:

T data;

Node \*prev = nullptr;

Node \*next = nullptr;

Node(T data) : data(data) {}

};

template <typename K>

class DoublyList{

Node<K> \*head, \*tail;

int cnt = 0;

public:

int size() // get size of the list

{

return this->cnt;

}

void addFirst(K data) // adding element at the start

{

Node<K> \*newNode = new Node<K>(data); // creating new node

if (head == nullptr){

head = tail = newNode; // making new node tail as well as head

} else {

newNode->next = head; // newNode's next pointer is pointing to current head

head->prev = newNode; // current head's previous pointer is now pointing to newNode

head = newNode; // changing newNode as head

}

cout << "Element added at the beginning : " << data << endl;

cnt++;

}

void addLast(K data) // adding element at the last

{

Node<K> \*newNode = new Node<K>(data); // creating new node

if (tail == nullptr) // if list is empty

{

head = tail = newNode; // head and tail now pointing to new node

}

else

{

newNode->prev = tail; // previous pointer of newNode is pointing to old tail

tail->next = newNode; // old tails next pointer is pointing to new node

tail = newNode; // newNode is now out new tail

}

cout << "Element added at the end : " << data << endl;

cnt++;

}

void removeFirst() // removing first node fro list

{

if (this->head == nullptr) // checking list is empty or not

{

cout << "List is empty!" << endl;

}

Node<K> \*oldNode = head; // getting the first node

head = head->next; // now making second node as a head node

head->prev = nullptr; // removing reference of old first node

cout << "Element removed from front : " << (oldNode->data) << endl;

delete oldNode; // deleting the memory

cnt--;

}

void removeLast() // removing last node fro list

{

if (this->head == nullptr) // checking list is empty or not

{

cout << "List is empty!!" << endl;

return;

}

Node<K> \*oldNode = tail; // getting last node from list

tail = tail->prev; // pointing second last node as tail node

tail->next = nullptr; // removing second last nodes next reference

cout << "Element removed from last : " << (oldNode->data) << endl;

delete oldNode; // deleting memory

cnt--;

}

void search(K data) // searching element in a list

{

Node<K> \*temp = head;

while (temp != nullptr)

{

if (data == temp->data)

{

cout << "Element found!" << endl;

return;

}

temp = temp->next;

}

cout << "Element Not found!" << endl;

}

void print()

{

if (this->head == nullptr)

{

cout << "list is empty" << endl;

return;

}

cout << "list => ";

Node<K> \*temp = head;

while (temp != NULL)

{

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

temp = temp->next;

}

cout << endl;

}

void printRev()

{

if (this->head == nullptr)

{

cout << "list is empty" << endl;

return;

}

cout << "list => ";

Node<K> \*temp = tail;

while (temp != NULL)

{

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

temp = temp->prev;

}

cout << endl;

}

};

void utilStack()

{

int choice, data;

DoublyList<int> \*list = new DoublyList<int>();

cout << "List Operations {11 : add first, 12 : add last, 21 : remove first, 22 : remove last, 31 : display, 32 : display Reverse, 4 : count, 5 : search, other : exit}";

do

{

cout << "\nchoice : ";

cin >> choice;

switch (choice)

{

case 11:

cout << "Enter Element : ";

cin >> data;

list->addFirst(data);

break;

case 12:

cout << "Enter Element : ";

cin >> data;

list->addLast(data);

break;

case 21:

list->removeFirst();

break;

case 22:

list->removeLast();

break;

case 31:

list->print();

break;

case 32:

list->printRev();

break;

case 4:

cout << "Size of List are : " << list->size() << endl;

break;

case 5:

cout << "Enter Element to search: ";

cin >> data;

list->search(data);

break;

default:

return;

}

} while (true);

}

int main(int argc, char const \*argv[])

{

utilStack();

return 0;

}

| List Operations {11 : add first, 12 : add last, 21 : remove first, 22 : remove last, 31 : display, 32 : display Reverse, 4  : count, 5 : search, other : exit}  choice : 11  Enter Element : 32  Element added at the beginning : 32  choice : 11  Enter Element : 43  Element added at the beginning : 43  choice : 12  Enter Element : 778  Element added at the end : 778  choice : 12  Enter Element : 98  Element added at the end : 98  choice : 31  list => 43 32 778 98  choice : 5  Enter Element to search: 778  Element found!  choice : 32  list => 98 778 32 43  choice : 21  Element removed from front : 43  choice : 22  Element removed from last : 98  choice : 31  list => 32 778 |
| --- |

**Conclusion:** I have learned about Linked List and its various types.