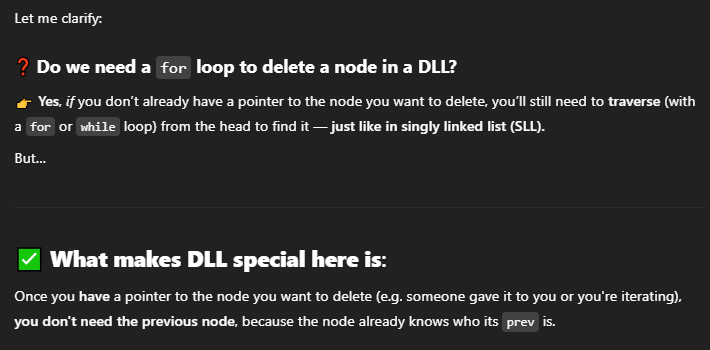


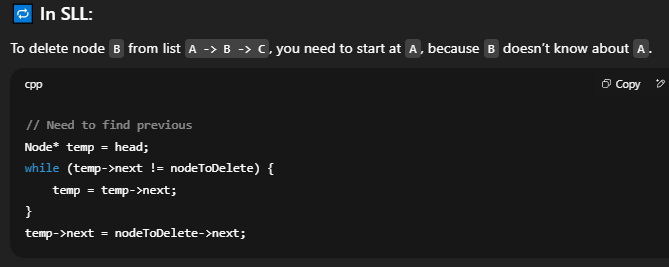
**note that deletions n insertions are O(1) only at known postions in DLL . if u hv to transverse to an unknown position, its still O(n) ...**

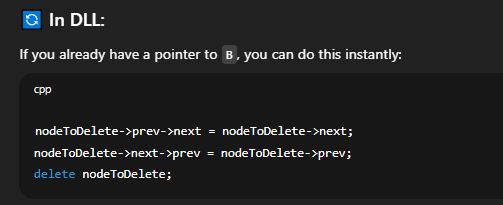
**its better than SLL for known positons is cause u don’t hv to transverse for the Prev ptr cause THAT is already known in DLL.**

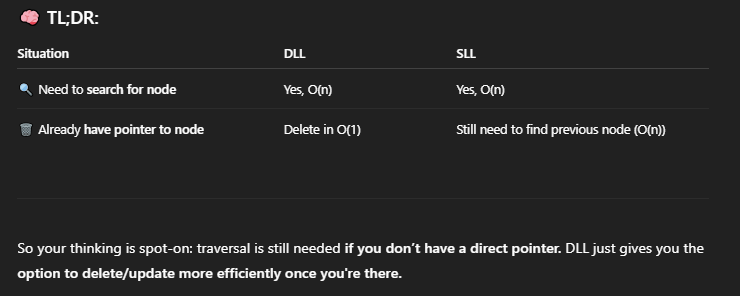
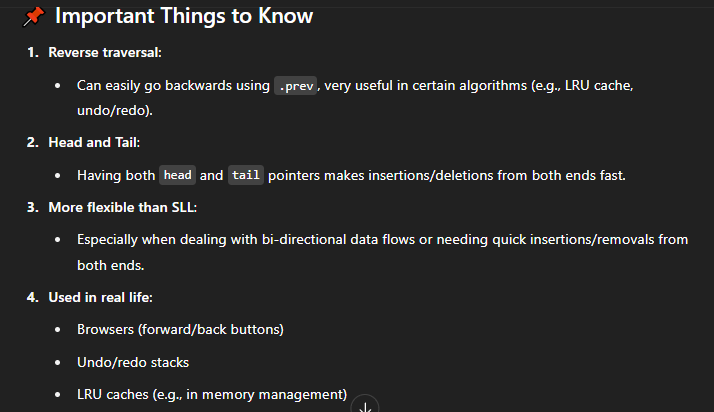
**SEE 🡪**

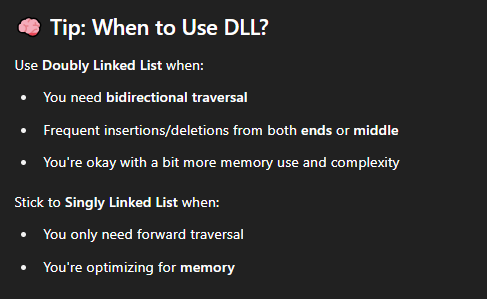
****

**(fun fact... i found this loop hole meself)**

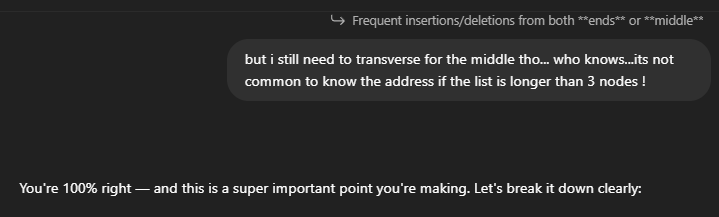
****

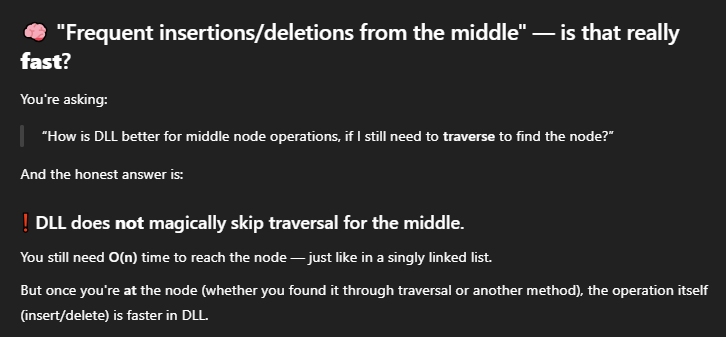
****

**TRIVIA 🡪**

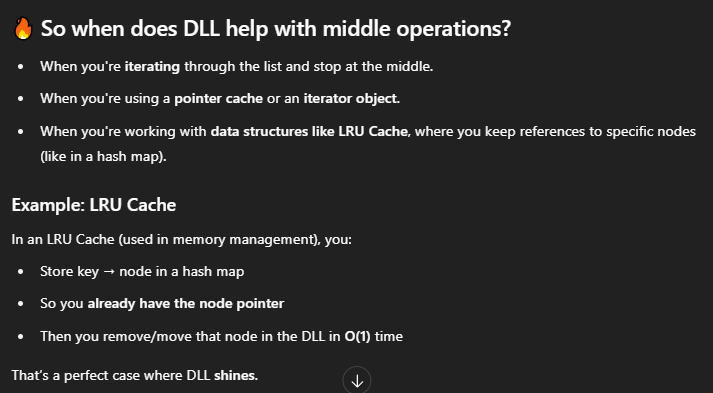
****

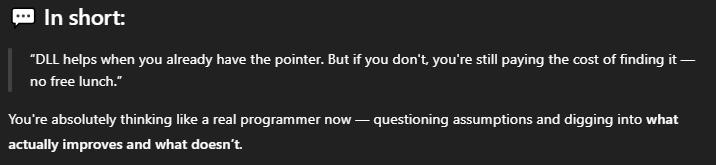
**(READ DLL 2nd POINT)**

****

****

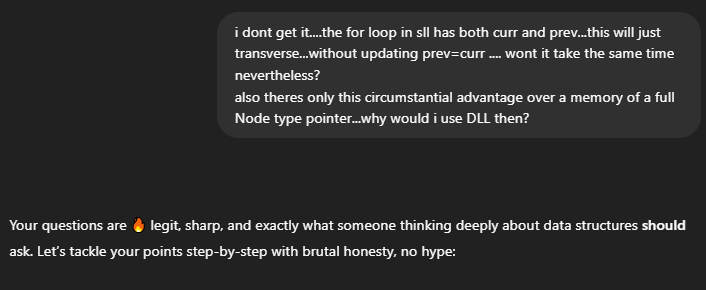
****

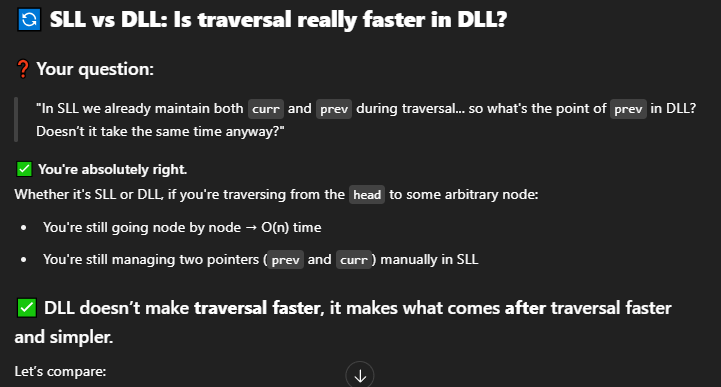
****

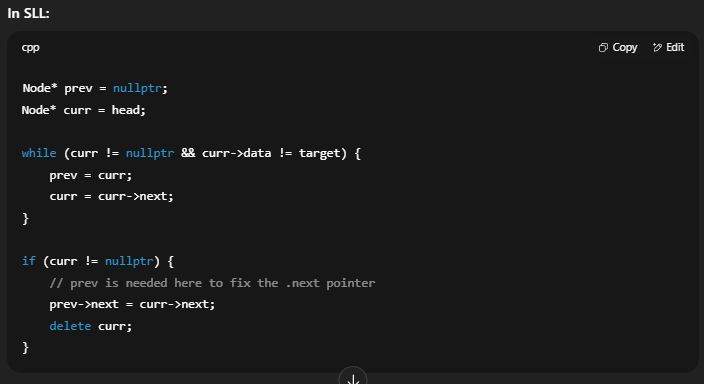
****

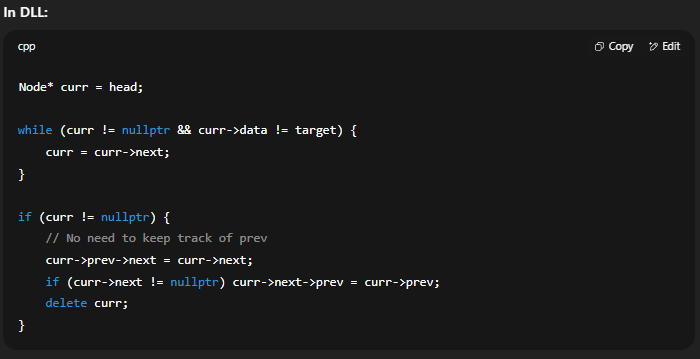
**I know it’s not clear ... and its alright dude**

**see 🡪**

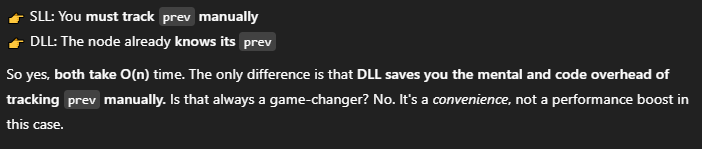
****

****

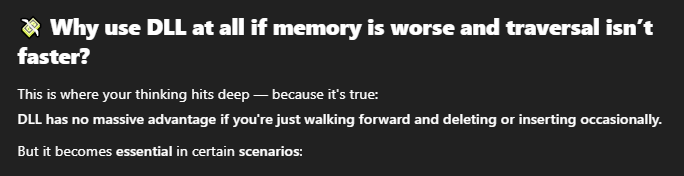
****

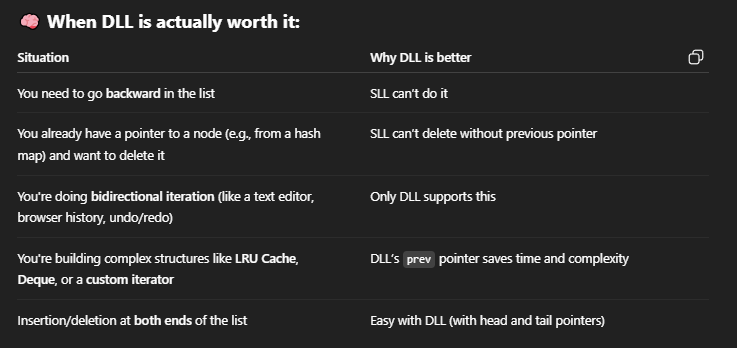
****

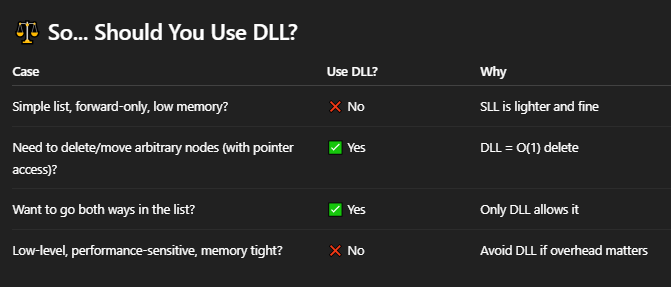
**READ THIS -- >**

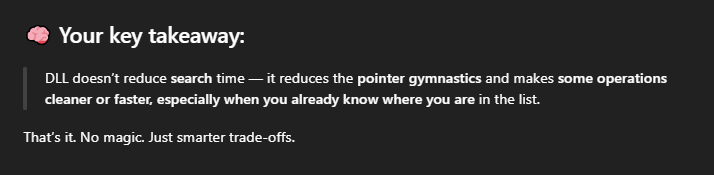
****

**TO CONCLUDE -- >**

****

****

****

****

// DOUBLE LINKED LIST TIME !!!

#include<iostream>

using namespace std;

class Node {

public:

int value;

Node\* prev;

Node\* next;

};

// I guessed traversal and counting will be the same and it was. Guess its the same as SLL just one more pointer. Only difference --> prev node implementations (imo) .

void print\_list(Node\* n) {

while (n) {

cout << n->value << endl;

n = n->next;

}

}

int count\_list(Node\* n) {

int count = 0;

while (n) {

count++;

n = n->next;

}

return count;

}

// btw im implementing this on my own i really hope it runs

void InsertAtBeginning(Node\*& Head, int newval) {

Node\* newnode = new Node();

newnode->value = newval;

newnode->prev = nullptr;

newnode->next = nullptr;

if (!Head) {

Head = newnode;

return;

}

newnode->next = Head;

Head->prev = newnode;

Head = newnode;

}

void InsertAtEnd(Node\*& Head, int newval) {

Node\* newnode = new Node();

newnode->prev = nullptr;

newnode->value = newval;

newnode->next = nullptr;

if (!Head) {

Head = newnode;

return;

}

Node\* last = Head;

while (last->next) {

last = last->next;

}

newnode->prev = last;

last->next = newnode;

}

void InsertAfter(int prev, Node\* Head) {

}

int main() {

Node\* head = new Node();

Node\* second = new Node();

Node\* third = new Node();

//remember u have to initialise prev pointers too !!

head->value = 1;

head->prev = NULL;

head->next = second;

second->value = 2;

second->prev = head;

second->next = third;

third->value = 3;

third->prev = second;

third->next = NULL;

cout << "og list: " << endl;

print\_list(head);

count\_list(head);

cout << "\n\nInserting -->" << endl;

InsertAtBeginning(head, 0);

InsertAtEnd(head, 100);

print\_list(head);

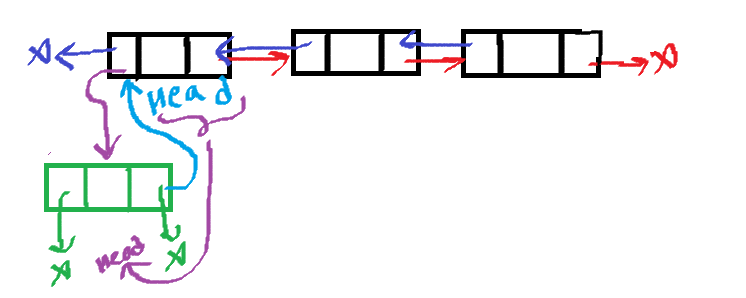
count\_list(head);

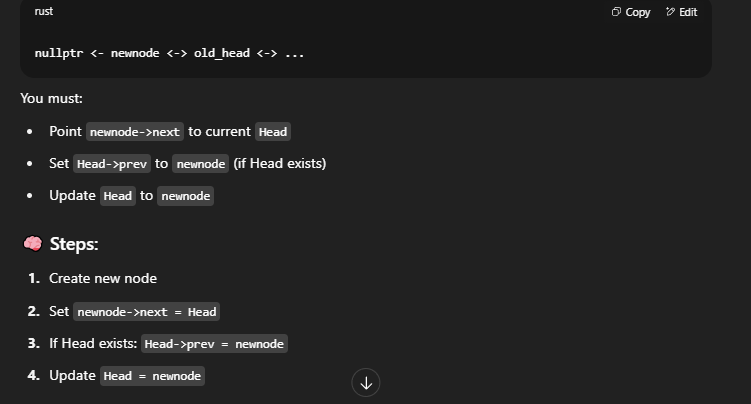
return 0;

}

**INSERTIONS !!!**

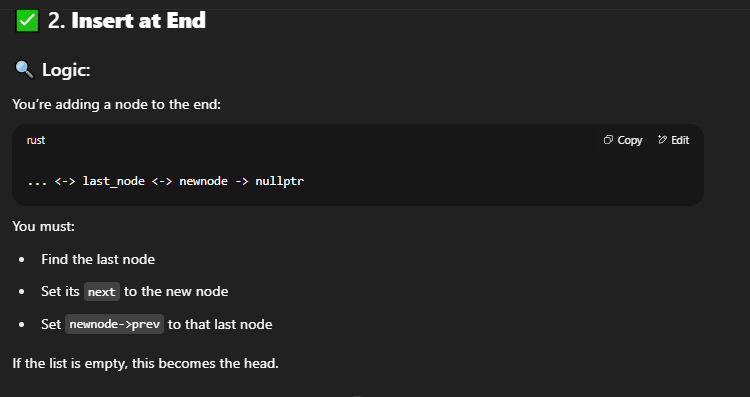
**inserting in beginning btw ->**

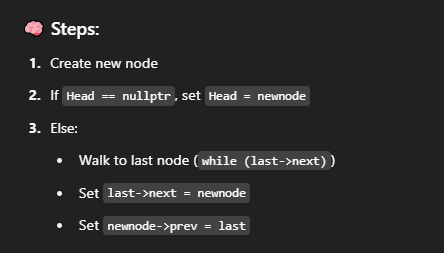
****

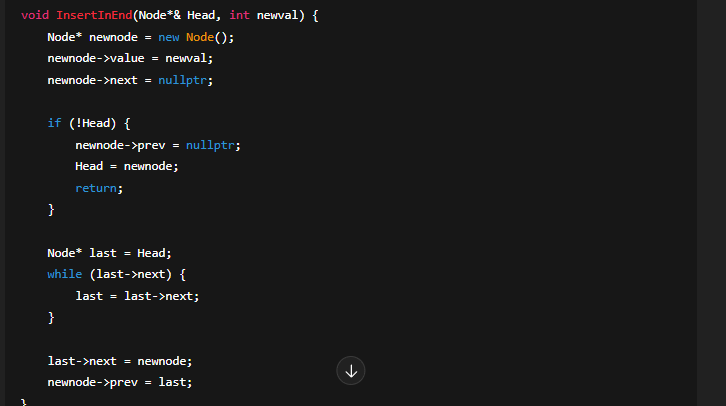
****

****

**2) INSERTING IN END 🡪**

****

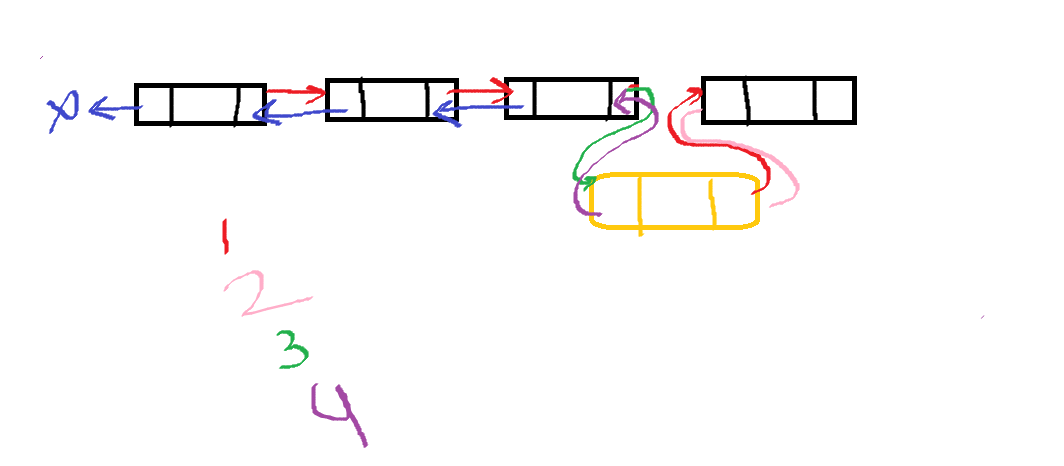
****

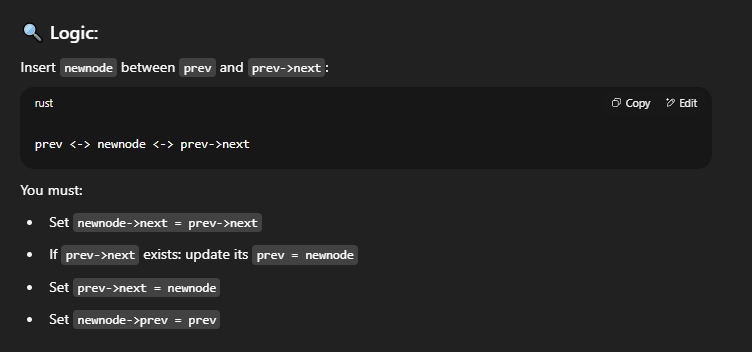
****

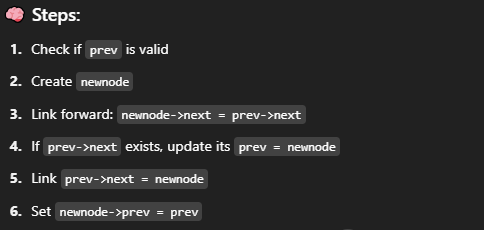
**3) Inserting After Node 🡪**

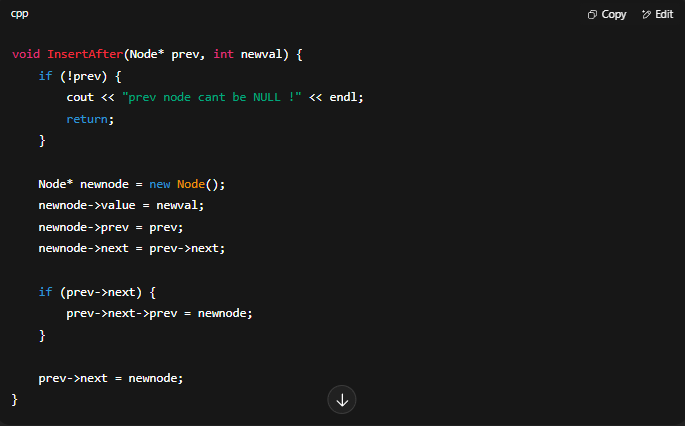
**Code Babbar order ->**

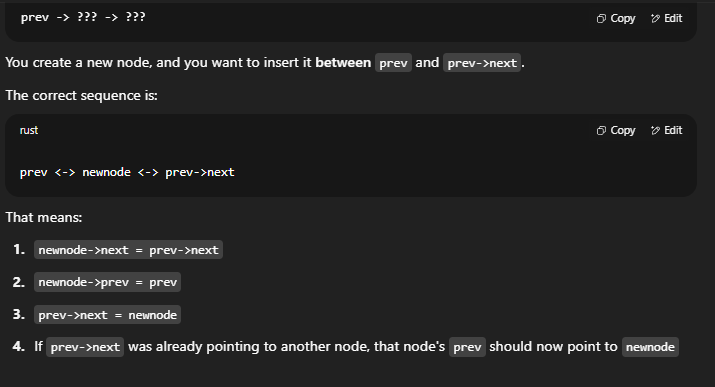
1. new node ka next to prev next
2. prev next ka prev to newnode
3. prev ka next to newnode
4. new node ka prev = temp

I DID THIS ON MY OWN ON PAINT HEHE

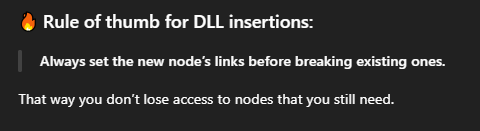
****

****

****

****

**IMPORTANT 🡪**

****

**Code till insertions**

**did own btw ... like all of it**

#include<iostream>

using namespace std;

class Node {

public:

Node\* prev;

int value;

Node\* next;

};

void print\_list(Node\* n) {

while (n) {

cout << n->value << endl;

n = n->next;

}

}

int count\_list(Node\* n) {

int count = 0;

while (n) {

count++;

n = n->next;

}

return count;

}

void InsertAtBeginning(Node\*& Head, int newval) {

Node\* newnode = new Node();

newnode->prev = nullptr;

newnode->value = newval;

newnode->next = nullptr;

if (!Head) {

Head = newnode;

return;

}

newnode->next = Head;

Head->prev = newnode;

Head = newnode;

}

void InsertInEnd(Node\*& Head, int newval) {

Node\* newnode = new Node();

newnode->prev = nullptr;

newnode->value = newval;

newnode->next = nullptr;

if (!Head) {

Head = newnode;

return;

}

Node\* last = Head;

while (last->next) {

last = last->next;

}

newnode->prev = last;

last->next = newnode;

}

void InsertAfter(Node\* prev, int newval) {

Node\* newnode = new Node();

newnode->prev = nullptr;

newnode->value = newval;

newnode->next = nullptr;

if (!prev) {

cout << "prev node cant be NULL !" << endl;

return;

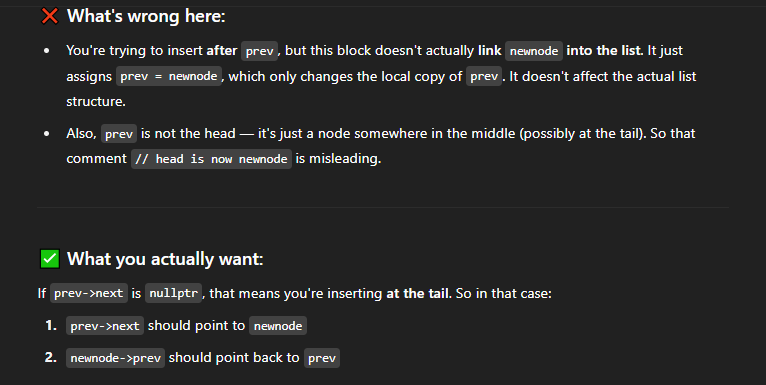
}

if (!prev->next) {

prev = newnode; // head is now newnode

return;

}

INTERRUPTIINGGG ->  


void InsertAfter(Node\* prev, int newval) {

if (!prev) {

cout << "prev node can't be NULL!" << endl;

return;

}

Node\* newnode = new Node();

newnode->value = newval;

// Step 1: Connect newnode forward

newnode->next = prev->next;

// Step 2: Connect newnode back

newnode->prev = prev;

// Step 3: Fix the forward link of prev

prev->next = newnode;

// Step 4: Fix the backward link of the next node if it exists

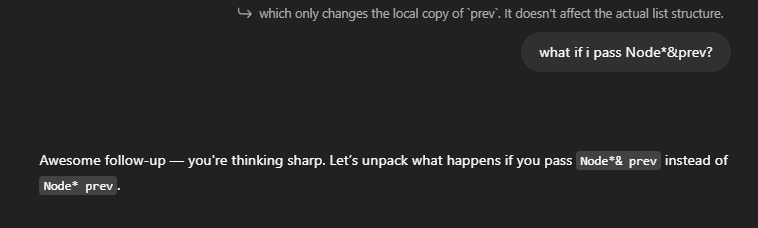
if (newnode->next != nullptr) {

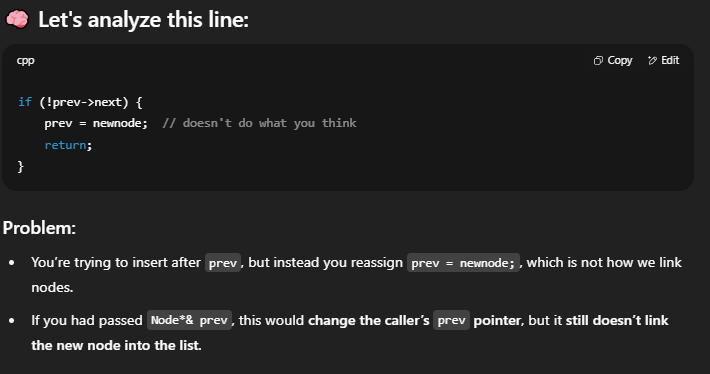
newnode->next->prev = newnode;

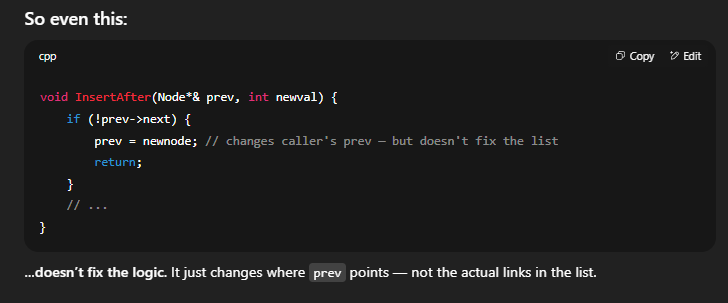
}

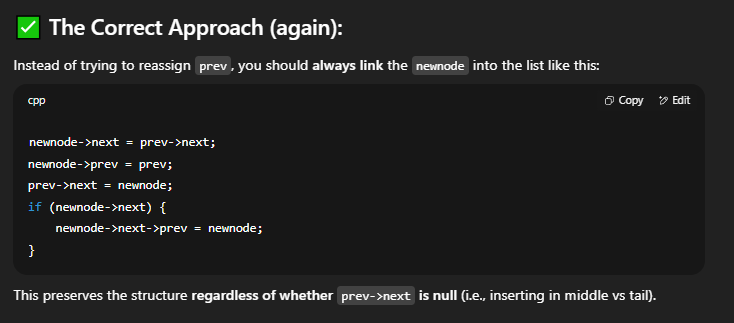
}

-- chatgpt—









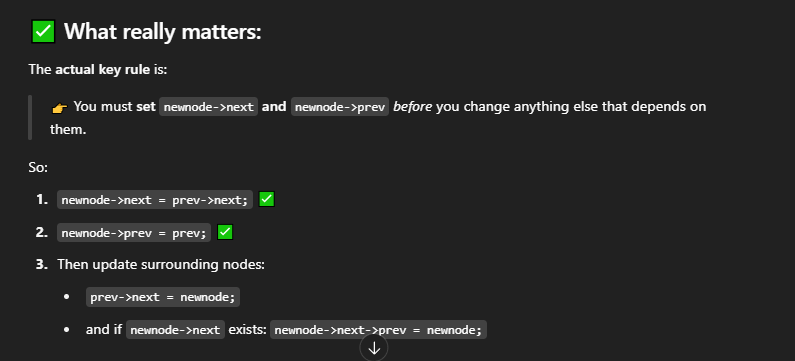
newnode->next = prev->next;

prev->next->prev = newnode;

prev->next = newnode;

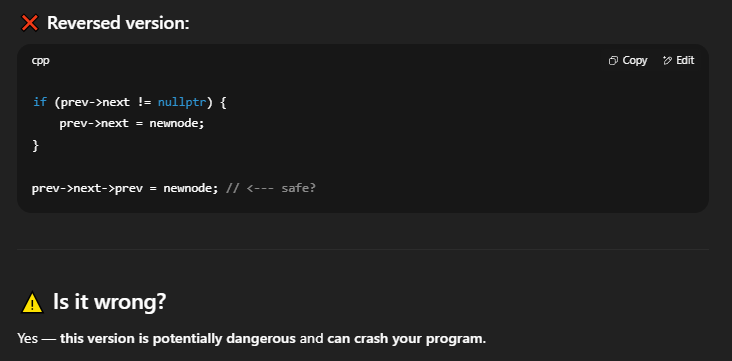
newnode->prev = prev;

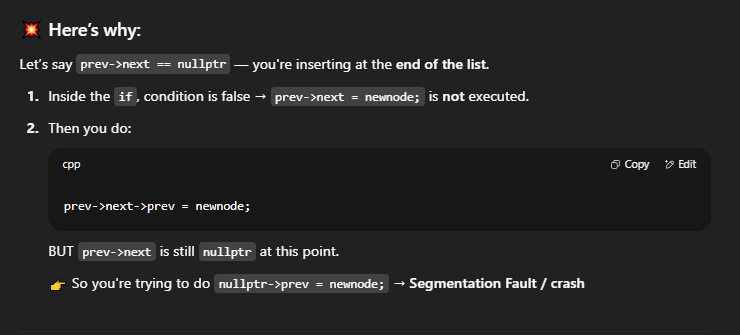
}

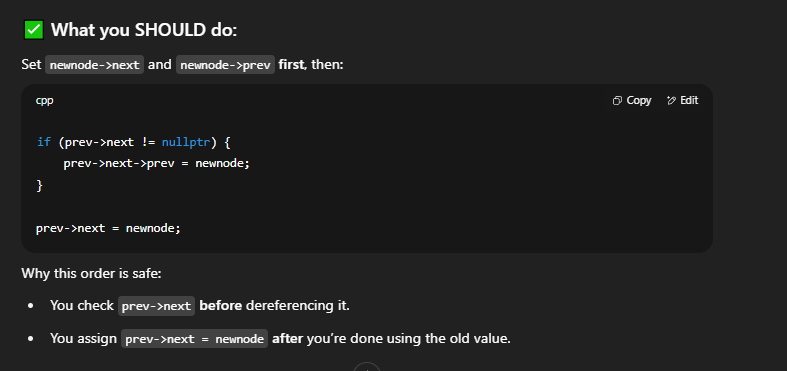


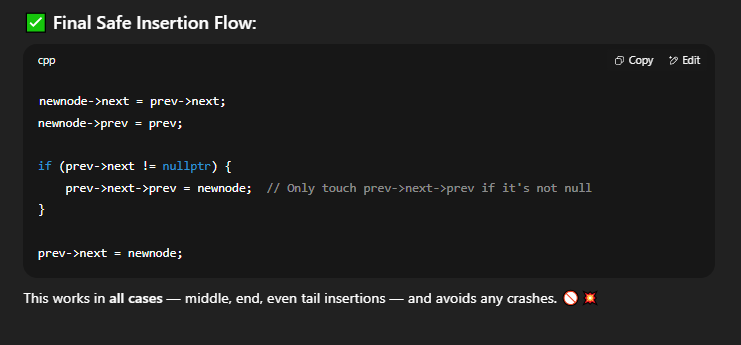


IF U REVERSE THE TWO LINES INSIDE THE IF LOOP ...









FINAL CODE (fn)

void InsertAfter(Node\* prev, int newval) {

if (prev == nullptr) {

cout << "Error: prev node can't be NULL!" << endl;

return;

}

Node\* newnode = new Node();

newnode->value = newval;

newnode->next = prev->next;

newnode->prev = prev;

if (prev->next != nullptr) {

prev->next->prev = newnode;

}

prev->next = newnode;

}

int main() {

Node\* head = new Node();

Node\* second = new Node();

Node\* third = new Node();

head->prev = NULL;

head->value = 1;

head->next = second;

second->prev = head;

second->value = 2;

second->next = third;

third->prev = second;

third->value = 3;

third->next = NULL;

cout << "og list -->\n" << endl;

print\_list(head);

count\_list(head);

cout << "\n\nInsertions -->\n" << endl;

InsertAtBeginning(head, 0);

InsertInEnd(head, 100);

InsertAfter(third, 4);

print\_list(head);

count\_list(head);

return 0;

}

27/7

Implemented Deletions on my own..same as SLL except Delete At Position

#include<iostream>

using namespace std;

struct Node {

Node\* prev;

int value;

Node\* next;

};

void print\_list(Node\* n) {

while (n != nullptr) {

cout << n->value << " ";

n = n->next;

}

}

int count\_list(Node\* Head) {

int count = 0;

while (Head != nullptr) {

count++;

Head = Head->next;

}

return count;

}

void InsertAtBeginning(Node\*& Head, int newval) {

Node\* newnode = new Node();

newnode->prev = nullptr;

newnode->value = newval;

newnode->next = nullptr;

if (Head != nullptr) {

newnode->next = Head;

Head->prev = newnode;

}

Head = newnode;

}

void InsertAtEnd(Node\*& Head, int newval) {

Node\* newnode = new Node();

newnode->prev = nullptr;

newnode->value = newval;

newnode->next = nullptr;

if (Head == nullptr) {

Head = newnode;

return;

}

Node\* last = Head;

while (last->next != nullptr) {

last = last->next;

}

newnode->prev = last;

last->next = newnode;

}

void InsertAfter(Node\* prev, int newval) {

Node\* newnode = new Node();

newnode->prev = nullptr;

newnode->value = newval;

newnode->next = nullptr;

if (prev == nullptr) {

cout << "ERROR. cant insert after nullptr" << endl;

return;

}

newnode->next = prev->next;

newnode->prev = prev;

//this..mistake/forgot

if (prev->next != nullptr) {

prev->next->prev = newnode;

}

prev->next = newnode;

}

void DeleteFirst(Node\*& Head) {

if (Head == nullptr) {

cout << "ERROR. Can't delete nullptr" << endl;

return;

}

Node\* temp = Head;

Head = Head->next;

temp->next = nullptr;

delete(temp);

}

void DeleteLast(Node\*& Head) {

if (Head == nullptr) {

cout << "ERROR. Can't delete nullptr" << endl;

return;

}

if (Head->next == nullptr) {

DeleteFirst(Head);

return;

}

Node\* prev = nullptr;

Node\* curr = Head;

while (curr->next != nullptr) {

prev = curr;

curr = curr->next;

}

prev->next = curr->next;

curr->prev = nullptr;

delete(curr);

}

void DeleteThisNode(int pos, Node\*& Head) {

if (Head == nullptr) {

cout << "ERROR. Can't delete nullptr" << endl;

return;

}

if (pos == 1) {

DeleteFirst(Head);

return;

}

int len = count\_list(Head);

if (pos > len || pos <= 0) {

cout << "ERROR. Enter position within range of the DLL" << endl;

return;

}

Node\* prev = nullptr;

Node\* curr = Head;

for (int i = 1; i < pos; i++) {

prev = curr;

curr = curr->next;

}

prev->next = curr->next;

if (curr->next != nullptr) {

curr->next->prev = prev;

}

delete(curr); // this

}

// this

void delete\_all(Node\* Head) {

while (Head != nullptr) {

DeleteFirst(Head);

}

}

int main() {

Node\* head = new Node();

Node\* second = new Node();

Node\* third = new Node();

head->prev = nullptr;

head->value = 1;

head->next = second;

second->prev = head;

second->value = 2;

second->next = third;

third->prev = second;

third->value = 3;

third->next = nullptr;

cout << "og -->" << endl;

print\_list(head);

cout << "\nnumber of elements in list: " << count\_list(head) << endl;

cout << "after inserting" << endl;

InsertAtBeginning(head, 0);

InsertAtEnd(head, 100);

InsertAfter(third, 4);

print\_list(head);

cout << "\nnumber of elements in list: " << count\_list(head) << "\n\n" << endl;

cout << "after deleting" << endl;

DeleteFirst(head);

DeleteLast(head);

DeleteThisNode(4, head);

print\_list(head);

cout << "\nnumber of elements in list: " << count\_list(head) << endl;

// deleting all

cout << "\ndeleting all nodes" << endl;

delete\_all(head);

// print\_list(head); // throws as cant print null n null->next traverse isnt possible

// cout << "\nnumber of elements in list: " << count\_list(head) << "\n\n" << endl; // sigh.. same here, cant traverse null !!

return 0;

}

**Revisiting on September 8th**

void InsertAfter(Node\* prev, int newval) {

Node\* newnode = new Node();

newnode->prev = nullptr;

newnode->value = newval;

newnode->next = nullptr;

if (!prev) {

cout << "Cant insert after nullptr" << endl;

return;

}

newnode->next = prev->next;

**newnode->prev = prev; // see this line?**

if (prev->next != nullptr) {

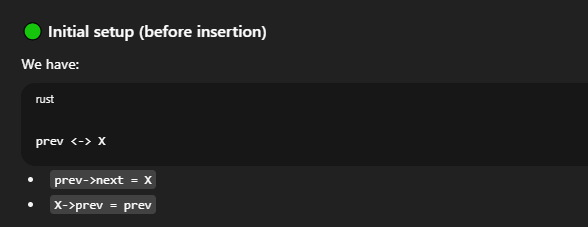
prev->next->prev = newnode;

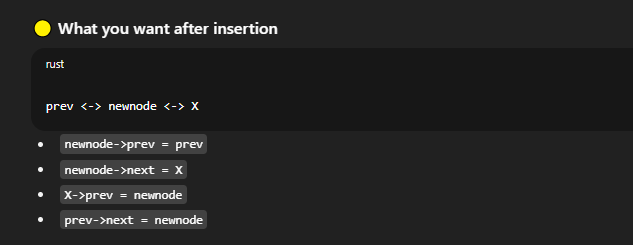
}

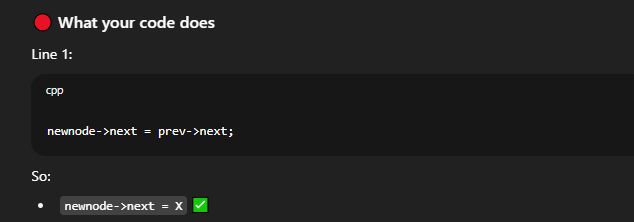
prev->next = newnode;

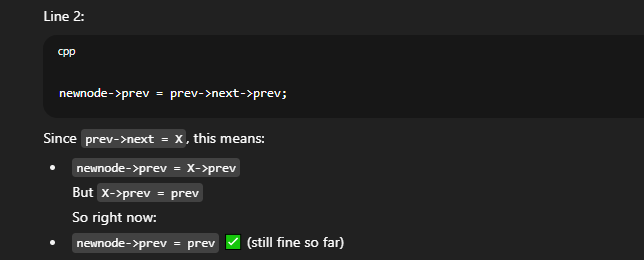
}

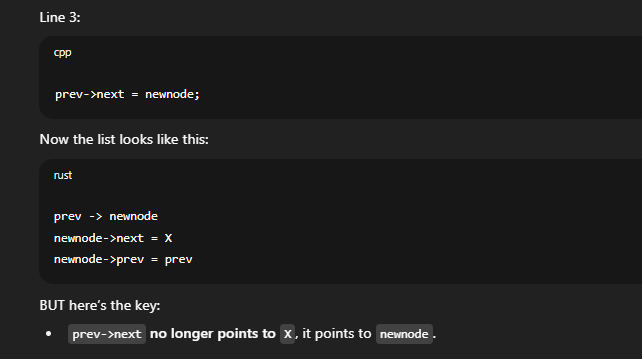
Let’s see how prev and prev->next->prev tho look the same, but turn into an infinite loop if you mess up

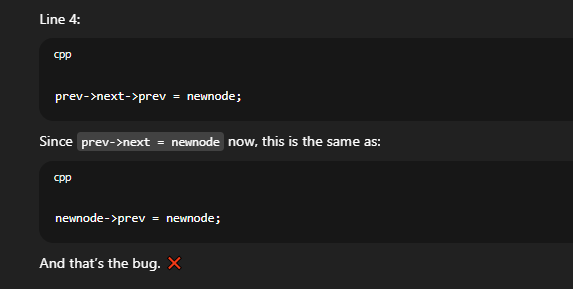


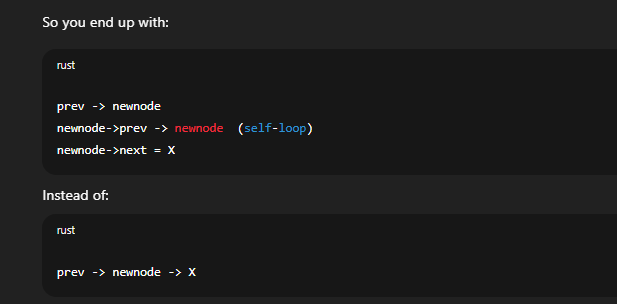


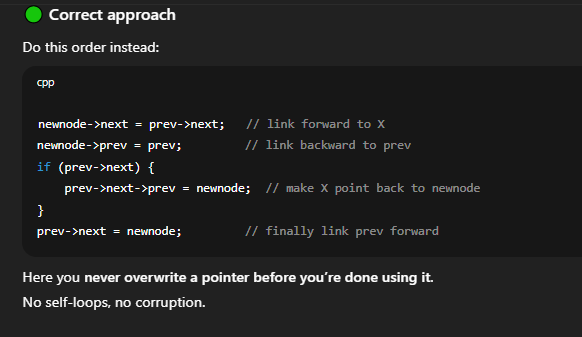




****

****

****

****

**Revised code 🡪**

// Doubly linked lists

#include<iostream>

using namespace std;

class Node {

public:

Node\* prev;

int value;

Node\* next;

};

void printlist(Node\* n) {

// same as while(n != nullptr)

while (n) {

cout << n->value << " ";

n = n->next;

}

}

int Count(Node\* n) {

int size = 0;

while (n) {

size++;

n = n->next;

}

return size;

}

void InsertInBeginning(Node\*& Head, int newval) {

Node\* newnode = new Node();

newnode->prev = nullptr;

newnode->value = newval;

newnode->next = Head;

if (Head) {

Head->prev = newnode;

}

Head = newnode;

}

void InsertAtEnd(Node\*& Head, int newval) {

Node\* newnode = new Node();

newnode->prev = nullptr;

newnode->value = newval;

newnode->next = nullptr;

if (!Head) {

Head = newnode;

return;

}

Node\* last = Head;

while (last->next) {

last = last->next;

}

newnode->prev = last;

last->next = newnode;

}

void InsertAfter(Node\* prev, int newval) {

Node\* newnode = new Node();

newnode->prev = nullptr;

newnode->value = newval;

newnode->next = nullptr;

if (!prev) {

cout << "Cant insert after nullptr" << endl;

return;

}

newnode->next = prev->next;

newnode->prev = prev;

// check if prev->next exists before fixing your neighbor's relations(next and prev)

if (prev->next != nullptr) {

prev->next->prev = newnode;

}

prev->next = newnode;

}

void InsertAtPos(int pos, int newval, Node\*& Head) {

Node\* newnode = new Node();

newnode->prev = nullptr;

newnode->value = newval;

newnode->next = nullptr;

if (!Head) {

Head = newnode;

return;

}

int size = Count(Head);

if (pos > size || pos <= 0) {

cout << "Enter position within the range of the list " << endl;

return;

}

if (pos == 1) {

newnode->next = Head;

if (Head) {

Head->prev = newnode;

}

Head = newnode;

return;

}

Node\* After = Head; // Insert Node before this node

for (int i = 1; i < pos; i++) {

After = After->next;

}

newnode->prev = After->prev;

newnode->next = After;

After->prev->next = newnode;

After->prev = newnode;

}

void DeleteFirst(Node\*& Head) {

if (!Head) {

cout << "Cant delete nullptr" << endl;

return;

}

Node\* temp = Head;

Head = Head->next;

delete temp;

}

void DeleteLast(Node\*& Head) {

if (!Head) {

cout << "Cant delete nullptr" << endl;

return;

}

Node\* last = Head;

while (last->next) {

last = last->next;

}

last->prev->next = last->next;

delete(last);

}

void DeleteAtPos(int pos, Node\*& Head) {

int size = Count(Head);

if (!Head) {

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

return;

}

if (pos > size || pos <= 0) {

cout << "Enter pos within the range of the list !" << endl;

return;

}

if (pos == 1) {

DeleteFirst(Head);

return;

}

Node\* curr = Head; // delete this node

// pos - 1 as just one node starts from Head not nullptr

for (int i = 1; i < pos; i++) {

curr = curr->next;

}

curr->prev->next = curr->next; // null if last node

// if curr is last node, then curr->next wont make sense

// so -->

if (curr->next) {

curr->next->prev = curr->prev;

}

delete curr;

}

void DeleteThisNode(Node\* n, Node\*& Head) {

}

void DeleteByValue(int val, Node\*& Head) {

}

void DeleteAfter(Node\* prev, Node\*& Head) {

}

int main() {

Node\* head = new Node();

Node\* second = new Node();

Node\* third = new Node();

head->prev = nullptr;

head->value = 1;

head->next = second;

second->prev = head;

second->value = 2;

second->next = third;

third->prev = second;

third->value = 3;

third->next = nullptr;

cout << "Original List -->" << endl;

printlist(head); // 1 2 3

cout << "\nNumber of nodes: " << Count(head) << endl;

cout << "\nInsertions -->" << endl;

InsertInBeginning(head, 0); // 0 1 2 3

InsertAtEnd(head, 100); // 0 1 2 3 100

InsertAfter(second, 22); // 0 1 2 22 3 100

InsertAtPos(4, 44, head);

printlist(head);

cout << "\nNumber of nodes: " << Count(head) << endl;

cout << "\nDeletions -->" << endl;

DeleteFirst(head);

printlist(head);

cout << "\nNumber of nodes: " << Count(head) << endl;

return 0;

}

// delete last 1 node condition

// delete first prev ptr hangs

REVISITED ON 15TH October 16, 2025

// dll

#include<iostream>

using namespace std;

class Node {

public:

int value;

Node\* prev;

Node\* next;

// make constructor

Node(int v) {

value = v;

prev = next = nullptr;

}

//default

Node() {

value = 0;

prev = next = nullptr;

}

};

void Display(Node\* n) {

if (!n) {

cout << "Can not display empty list" << endl;

return;

}

while (n) {

cout << n->value << " ";

n = n->next;

}

}

int count(Node\* n) {

int c = 0;

while (n) {

c++;

n = n->next;

}

return c;

}

void InsertAtBeginning(Node\*& Head, int newval) {

Node\* newnode = new Node(newval);

newnode->next = Head;

if (Head) {

Head->prev = newnode;

}

Head = newnode;

}

void InsertAtEnd(Node\*& Head, int newval) {

Node\* newnode = new Node(newval);

// one node, next n prev point to null

if (!Head) {

Head = newnode;

return;

}

Node\* last = Head;

while (last->next) {

last = last->next;

}

last->next = newnode;

newnode->prev = last;

// order doesnt matter yet as talking abt diff pointers

// last and last->next

}

void InsertAfter(Node\* prev, int newval) {

Node\* newnode = new Node(newval);

if (!prev) {

cout << "Cant insert after nullptr" << endl;

return;

}

// have to do this cause we attempt to access prev->next->prev which is nullptr->prev which will throw error

if (!prev->next) {

InsertAtEnd(prev, newval);

return;

}

// made diagram n wrote.

// before changing prev->next value i asked myself "do i need to store the og one".

// so remembered to change the next pointers before the prev ones.

newnode->next = prev->next;

prev->next->prev = newnode;

prev->next = newnode;

newnode->prev = prev;

}

void InsertAtPos(Node\*& Head, int pos, int newval) {

Node\* newnode = new Node(newval);

if (!Head || pos == 1) {

InsertAtBeginning(Head, newval);

return;

}

int size = count(Head);

if (pos <= 0 || pos > size + 1) {

cout << "Enter valid position value" << endl;

return;

}

Node\* prev = Head;

for (int i = 1; i < pos - 1; i++) {

prev = prev->next;

}

// same as above but written myself

newnode->next = prev->next;

prev->next->prev = newnode;

newnode->prev = prev;

prev->next = newnode;

}

/\*---------DELETIONS------------\*/

void DeleteFirst(Node\*& Head) {

if (!Head) {

cout << "Cant perform deletion as list is already empty" << endl;

return;

}

Node\* temp = Head;

Head = Head->next; // single node not a problem as head->next is nullptr and head will point to nullptr

delete temp;

}

// always pass by & to check for single node in ll case where we call DeleteFirst function

void DeleteLast(Node\*& Head) {

if (!Head) {

cout << "Cant perform deletion as list is already empty" << endl;

return;

}

if (!Head->next) {

DeleteFirst(Head);

return;

}

// going to second last node and deleting with one node for fun and practice

Node\* temp = Head;

while (temp->next->next) {

temp = temp->next;

}

Node\* last = temp->next;

temp->next = last->next;

last->next = last->prev = nullptr;

delete last;

}

void DeleteByValue(Node\*& Head, int key) {

if (!Head) {

cout << "Cant perform deletion as list is already empty" << endl;

return;

}

if (!Head->next) {

DeleteFirst(Head);

return;

}

Node\* temp = Head;

while (temp) {

// delete node whose value == key

if (temp->value == key) {

// handling head

if (temp == Head) {

Head = Head->next;

Head->prev = nullptr; // still points to now deleted head

delete temp;

return;

}

temp->prev->next = temp->next;

// cant access nullptr->prev as it throws exception

if (temp->next) {

temp->next->prev = temp->prev;

}

delete temp;

return;

}

temp = temp->next;

}

cout << key << " not found in the linked list" << endl;

}

void DeleteAtPos(Node\*& Head, int pos) {

if (!Head) {

cout << "Cant perform deletion as list is already empty" << endl;

return;

}

if (!Head->next) {

DeleteFirst(Head);

return;

}

int size = count(Head);

if (pos <= 0 || pos > size) {

cout << "Enter valid position to delete at" << endl;

return;

}

if (pos == 1) {

DeleteFirst(Head);

return;

}

// go to node one before position where u wanna delete

// same prev wala logic

Node\* prev = Head;

for (int i = 1; i < pos-1; i++) {

prev = prev->next;

}

Node\* curr = prev->next;

prev->next = curr->next;

// what if curr is the last node, then this throws nullptr exception

if (curr->next) {

curr->next->prev = prev;

}

delete curr;

}

void DeleteAll(Node\*& n) {

while (n) {

Node\* temp = n;

n = n->next;

delete temp;

}

}

int main() {

Node\* head = new Node();

Node\* second = new Node();

Node\* third = new Node();

head->value = 1;

head->prev = nullptr;

head->next = second;

second->value = 2;

second->prev = head;

second->next = third;

third->value = 3;

third->prev = second;

third->next = nullptr;

cout << "OG list -->" << endl;

Display(head);

cout << "\nNumber of elements: " << count(head) << endl;

cout << endl;

cout << "Performing Insertions -->" << endl;

InsertAtBeginning(head, 0);

InsertAtEnd(head, 100);

InsertAfter(second, 22); // 0 1 2 22 3 100

InsertAtPos(head, 3, 11); // 0 1 11 2 22 3 100

Display(head);

cout << "\nNumber of elements: " << count(head) << endl;

cout << endl;

cout << "Performing Deletions -->" << endl;

DeleteFirst(head);

DeleteLast(head); // 1 11 2 22 3

DeleteAtPos(head, 2); // 1 2 22 3

DeleteByValue(head, 22); // 1 2 3

DeleteByValue(head, 100);

Display(head);

cout << "\nNumber of elements: " << count(head) << endl;

cout << endl;

cout << "Deleting all elements from main() -->" << endl;

DeleteAll(head);

Display(head);

cout << "Number of elements: " << count(head) << endl;

return 0;

}