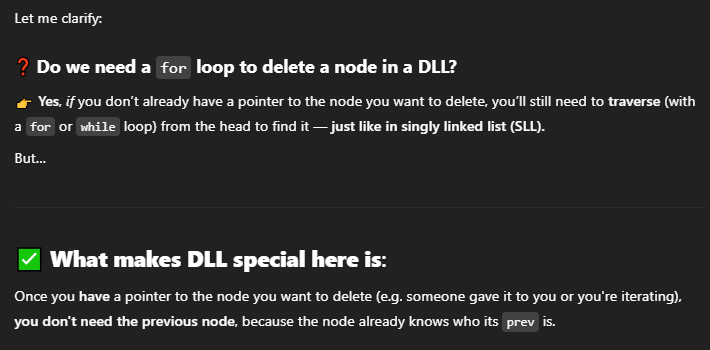


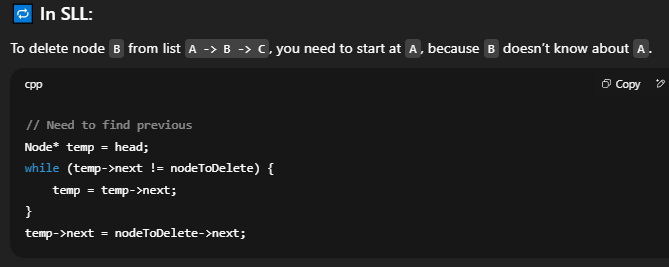
**note that deletions n insertions are O(1) only at known postions in DLL . if u hv to transverse to an unknown position, its still O(n) ...**

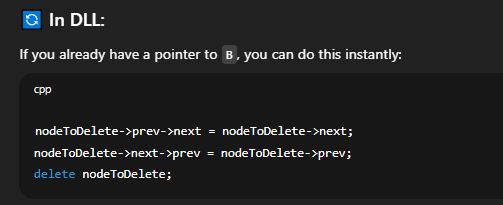
**its better than SLL for known positons is cause u don’t hv to transverse for the Prev ptr cause THAT is already known in DLL.**

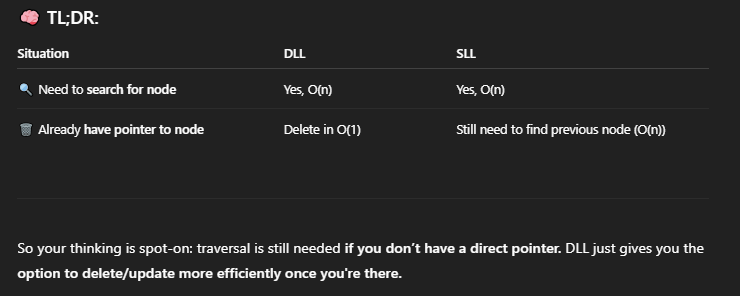
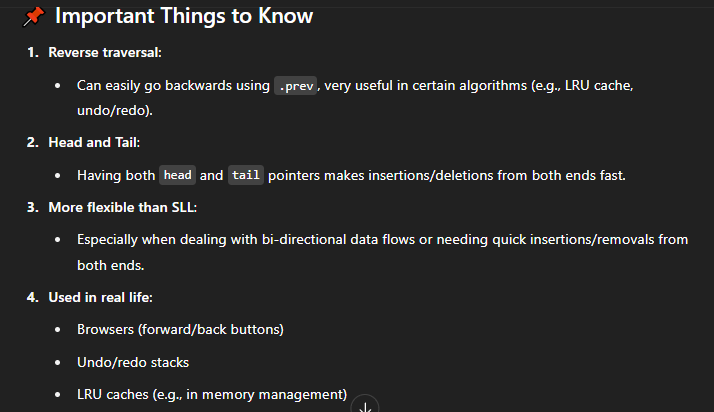
**SEE 🡪**

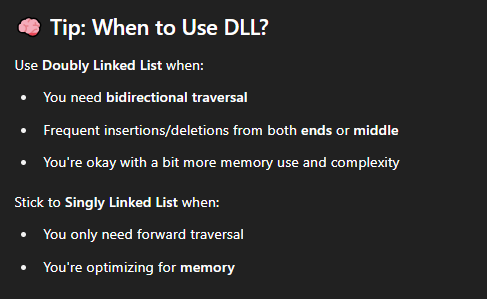
****

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

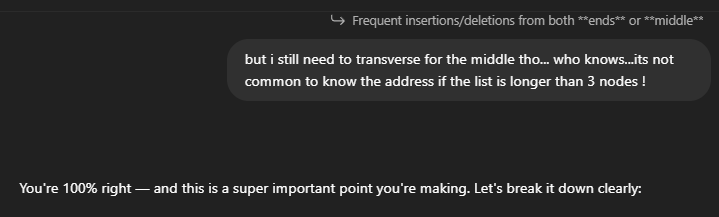
****

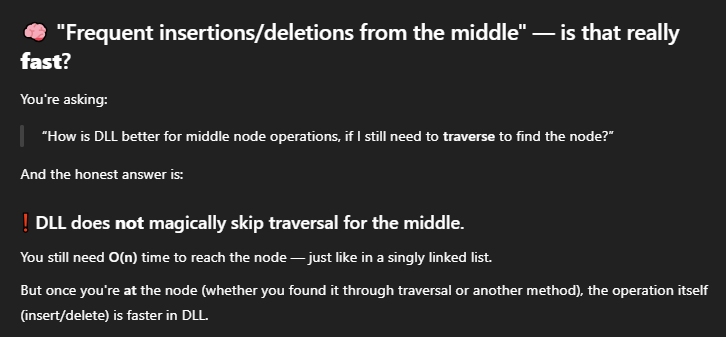
****

**TRIVIA 🡪**

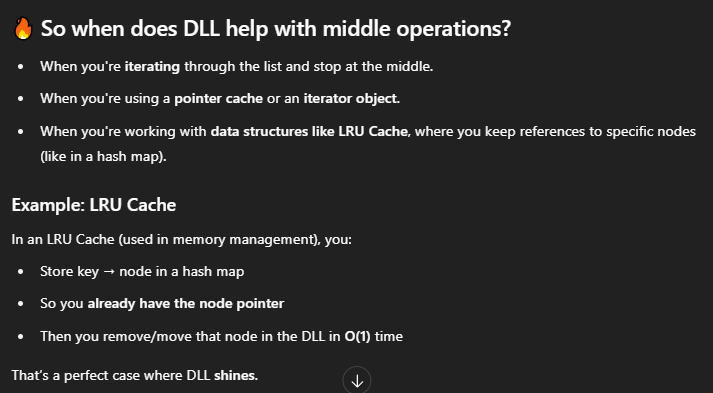
****

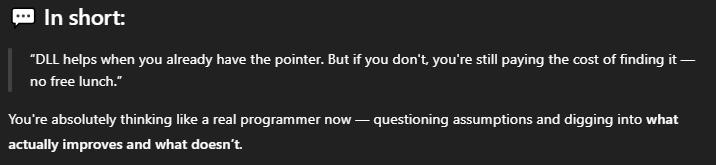
**(READ DLL 2nd POINT)**

****

****

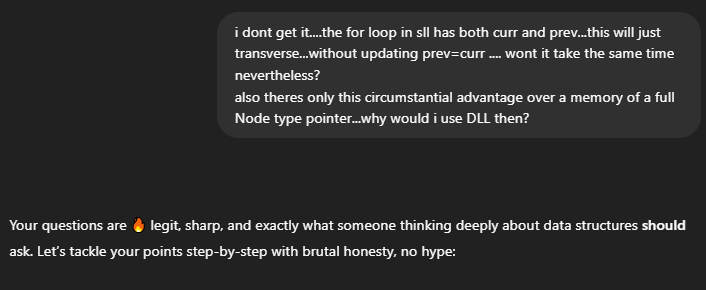
****

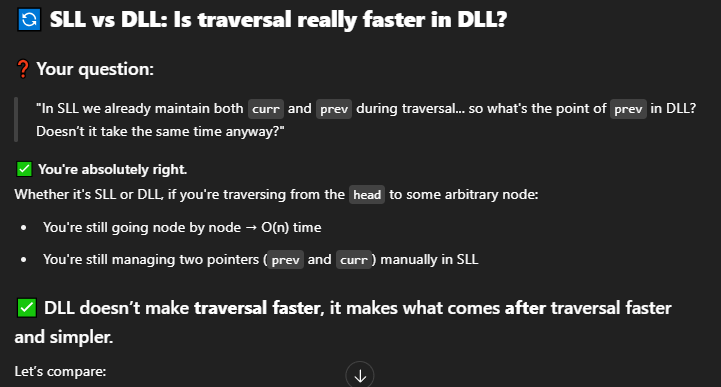
****

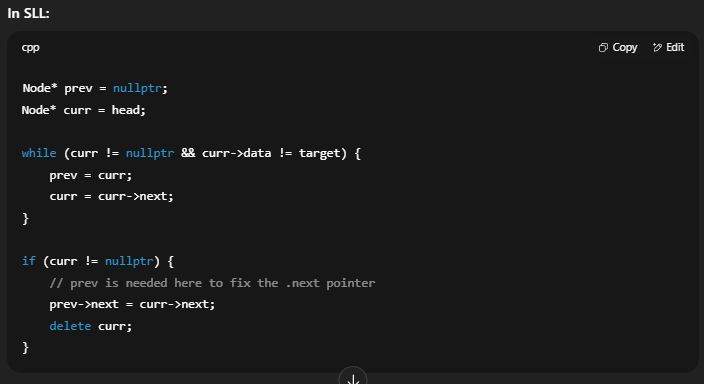
****

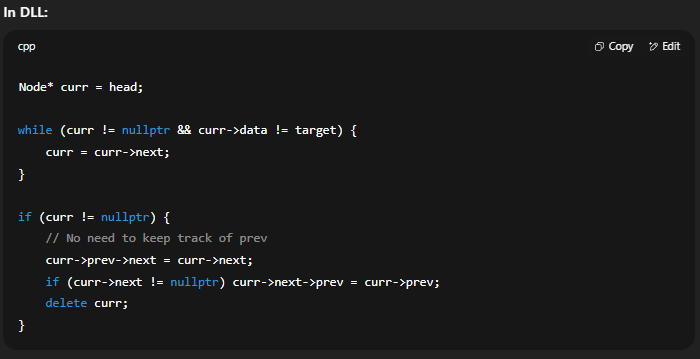
**I know it’s not clear ... and its alright dude**

**see 🡪**

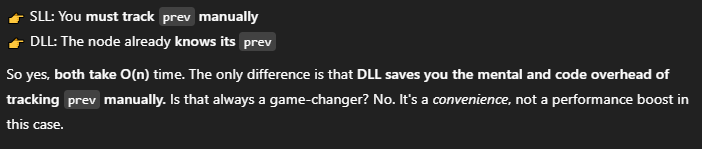
****

****

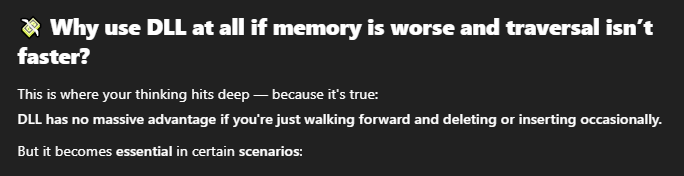
****

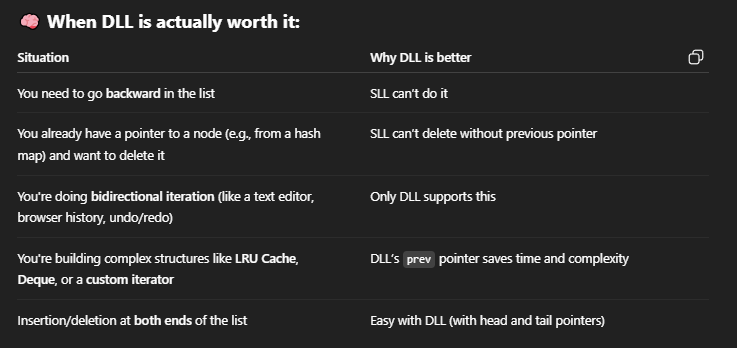
****

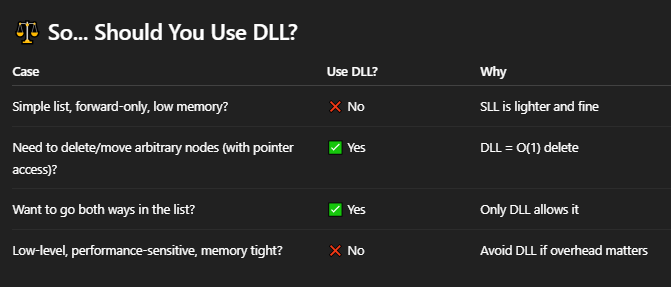
**READ THIS -- >**

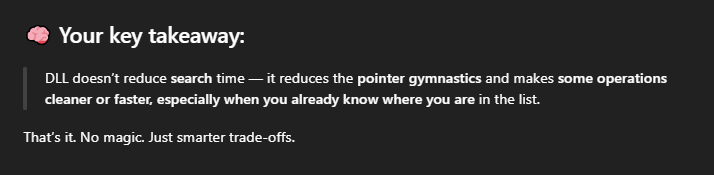
****

**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);

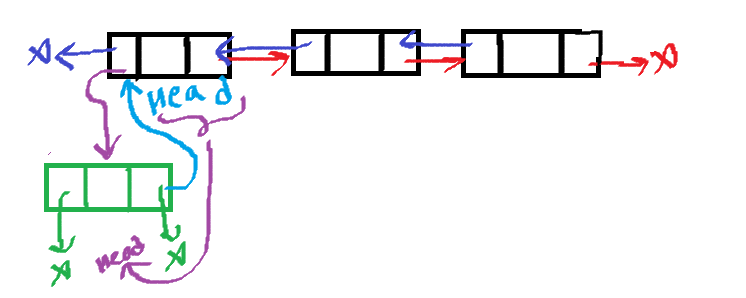
count\_list(head);

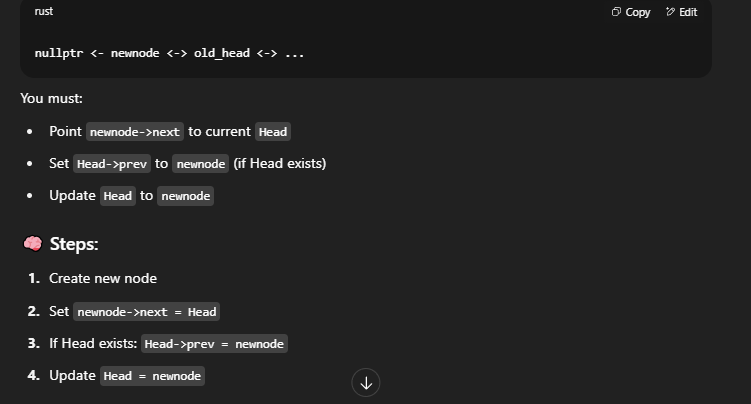
return 0;

}

**INSERTIONS !!!**

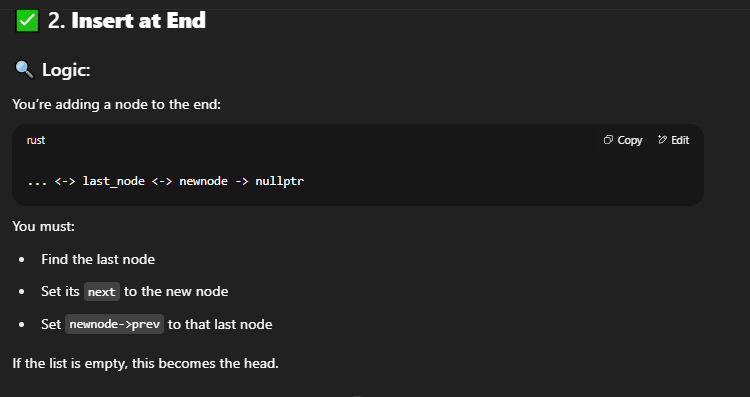
**inserting in beginning btw ->**

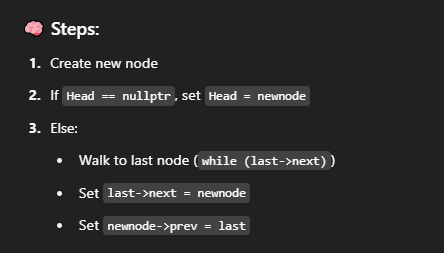
****

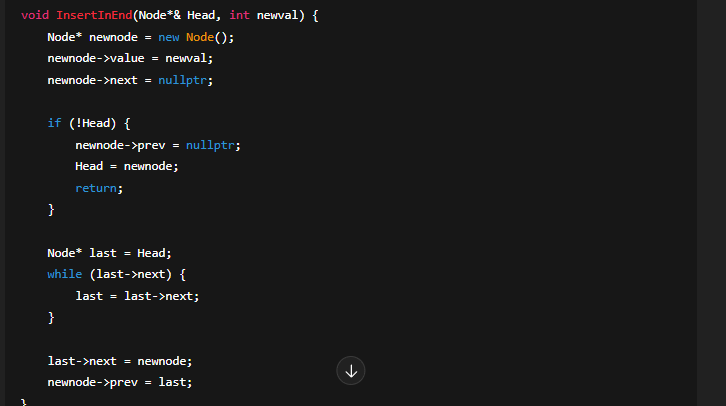
****

****

**2) INSERTING IN END 🡪**

****

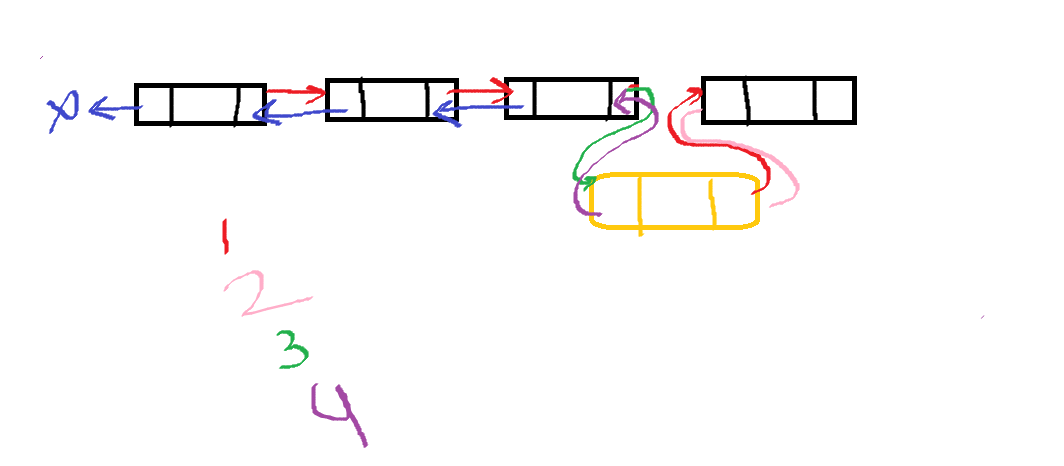
****

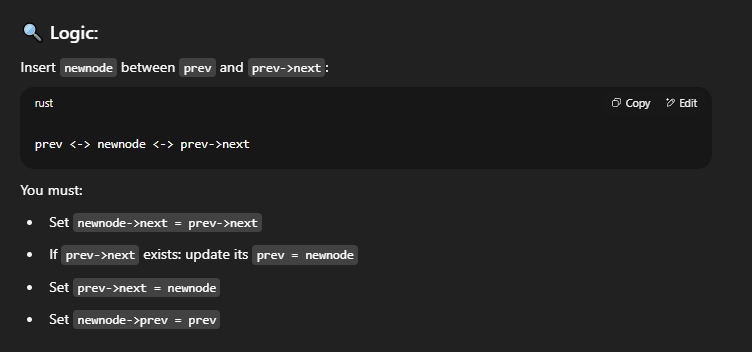
****

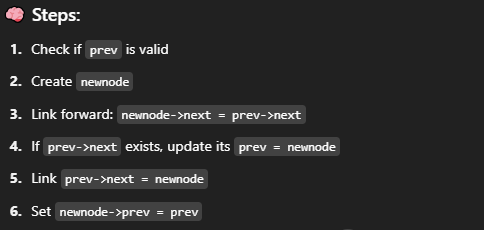
**3) Inserting After Node 🡪**

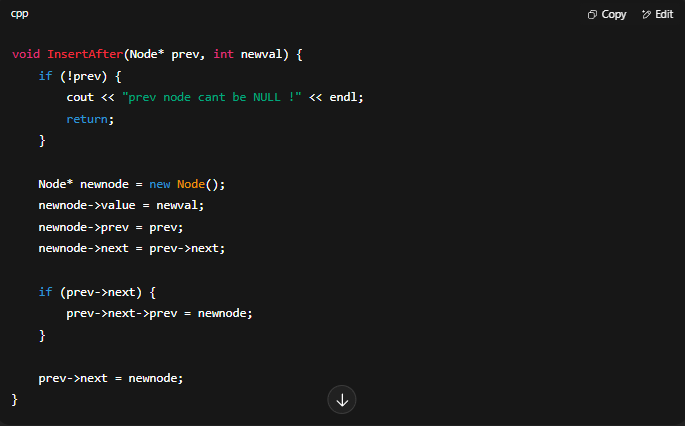
**Code Babbar order ->**

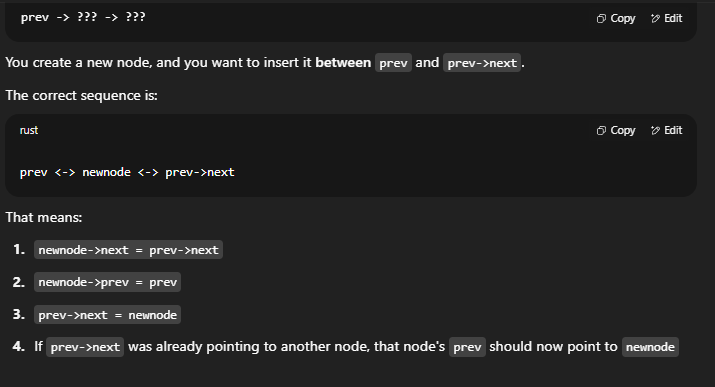
1. new node ka next to prev next
2. prev next ka prev to newnode
3. prev ka next to newnode
4. new node ka prev = temp

I DID THIS ON MY OWN ON PAINT HEHE

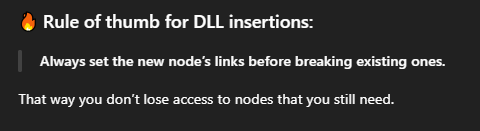
****

****

****

****

**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;

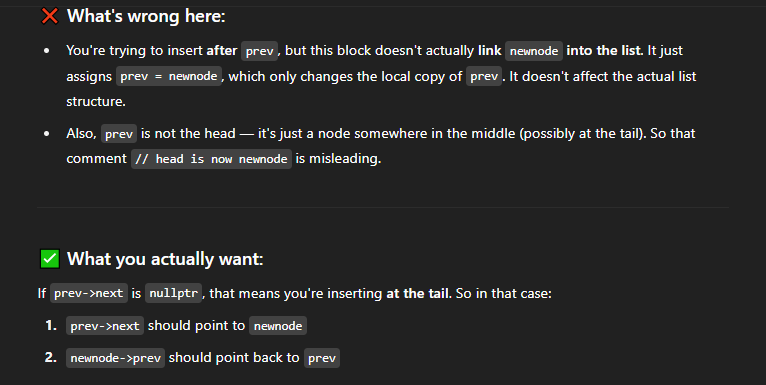
}

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

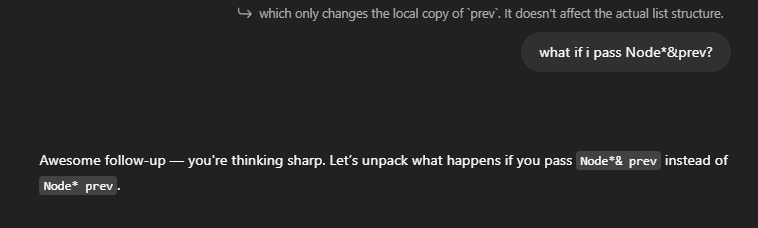
if (newnode->next != nullptr) {

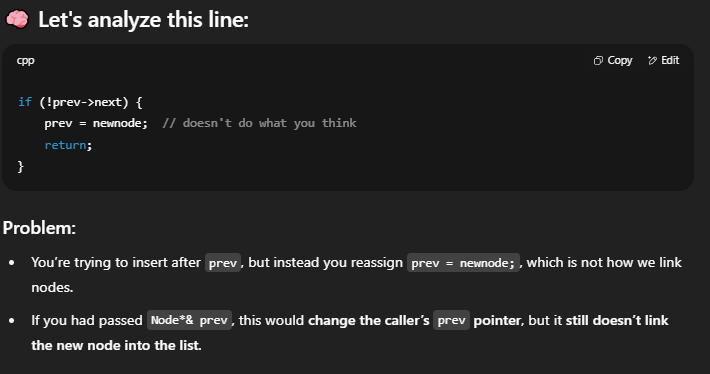
newnode->next->prev = newnode;

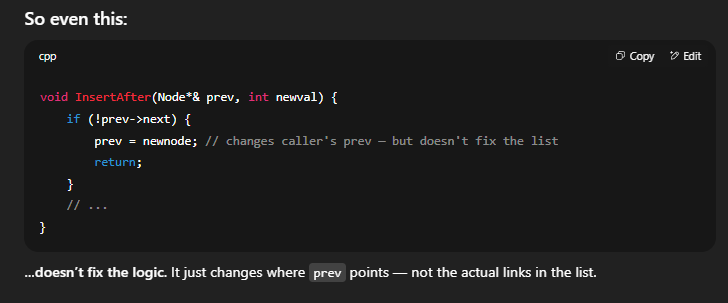
}

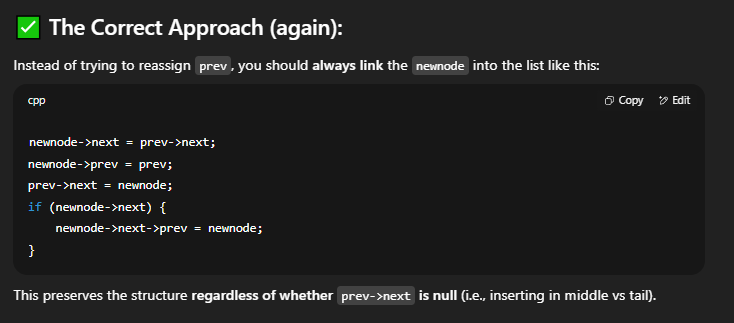
}

-- chatgpt—









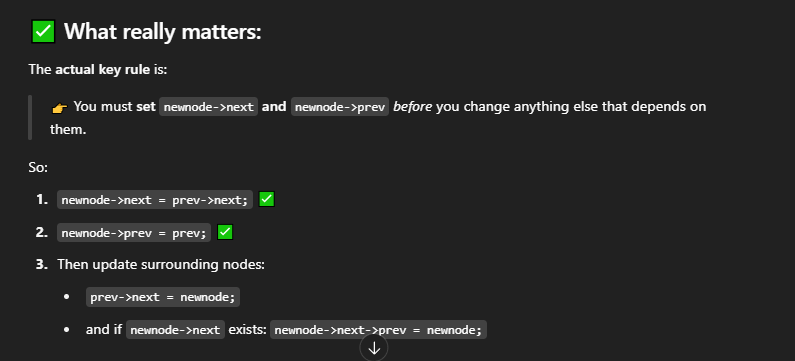
newnode->next = prev->next;

prev->next->prev = newnode;

prev->next = newnode;

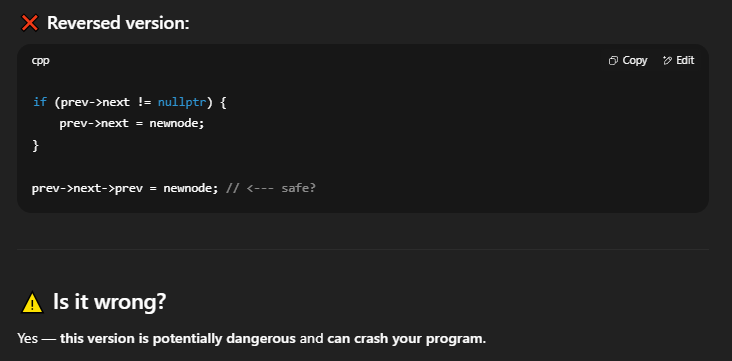
newnode->prev = prev;

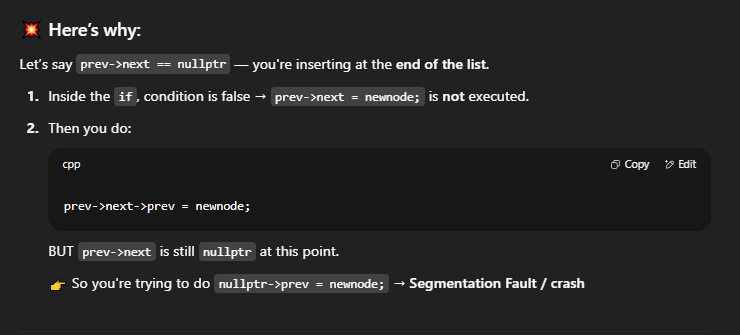
}

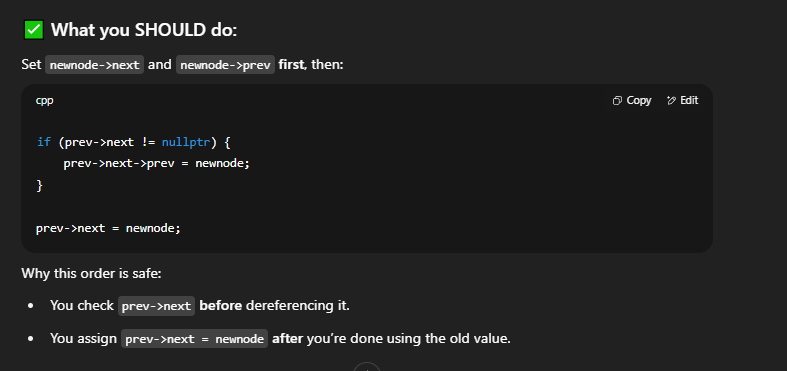


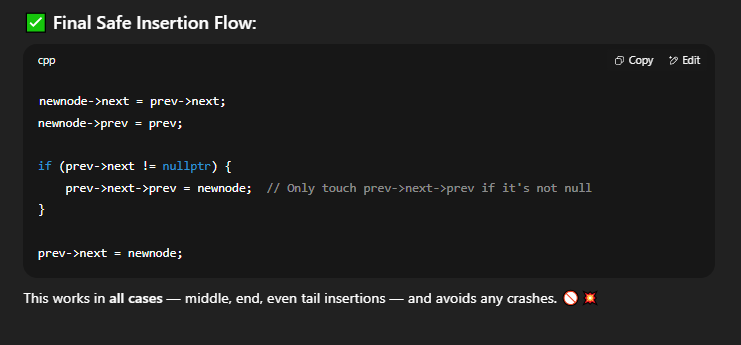


IF U REVERSE THE TWO LINES INSIDE THE IF LOOP ...









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;

}