**Singly Linked List**

#include<iostream>

using namespace std;

struct node {

int data;

node\* next;

node() :data(0), next(nullptr){};

};

class BaseSingly {

protected:

int count;

node\* head;

node\* current;

public:

BaseSingly() {

count = 0;

head = nullptr;

current = nullptr;

}

virtual void insert\_at\_front(int val) = 0;

virtual void insert\_at\_end(int val) = 0;

virtual void insert\_at\_index(int index, int val) = 0;

virtual void delete\_at\_front() = 0;

virtual void delete\_at\_end() = 0;

virtual void delete\_at\_index(int index) = 0;

virtual void rotate(int k) = 0;

virtual void print() = 0;

};

class singlylist : public BaseSingly {

public:

singlylist():BaseSingly() {}

~singlylist(){

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

delete\_at\_end();

}

}

void insert\_at\_front(int val) {

if (head == nullptr) {

head = new node;

current = head;

head->data = val;

count++;

}

else {

node\* temp = new node;

temp->data = val;

temp->next = head;

head = temp;

count++;

}

}

void insert\_at\_end(int val) {

if (head == nullptr) {

insert\_at\_front(val);

}

else {

current = head;

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

current = current->next;

}

node\* temp = new node;

temp->data = val;

current->next = temp;

count++;

}

}

void insert\_at\_index(int index, int val) {

if (index == 1) {

insert\_at\_front(val);

}

else if (index == count) {

insert\_at\_end(val);

}

else if (index < count && index > 1) {

current = head->next;

for (int i = 2; i < index - 1; i++) {

current = current->next;

}

node\* temp = new node;

temp->data = val;

temp->next = current->next;

current->next = temp;

count++;

}

else {

cout << "Invalid index" << endl;

}

}

void delete\_at\_front() {

current = head;

head = current->next;

delete current;

current = head;

count--;

}

void delete\_at\_end() {

current = head;

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

current = current->next;

}

delete current;

count--;

current = head;

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

current = current->next;

}

current->next = nullptr;

}

void delete\_at\_index(int index) {

current = head;

if (index == 1) {

delete\_at\_front();

}

else if (index == count) {

delete\_at\_end();

}

else if (index > 1 && index < count) {

current = head->next;

for (int i = 2; i < index - 1; i++) {

current = current->next;

}

node\* temp = new node;

temp = current->next;

current->next = current->next->next;

delete temp;

count--;

}

else {

cout << "Invalid index" << endl;

}

}

void rotate(int k) {

current = head;

node\* temp = current;

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

current = current->next;

if (i == k - 1) {

temp = current;

}

}

current->next = head;

head = temp->next;

temp->next = nullptr;

}

void print() {

current = head;

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

cout << "[" << current->data << "]";

current = current->next;

}

cout << endl;

}

};

int main() {

singlylist obj;

cout << "Insert at End" << endl;

for (int i = 1; i <= 5; i++) {

obj.insert\_at\_end(i);

}

obj.print();

cout << "Rotate" << endl;

obj.rotate(4);

obj.print();

cout << "Insert at Front" << endl;

obj.insert\_at\_front(1);

obj.insert\_at\_front(2);

obj.print();

cout << "Insert at Index" << endl;

obj.insert\_at\_index(4, 69);

obj.print();

cout << "Delete at Front" << endl;

obj.delete\_at\_front();

obj.print();

cout << "Delete at End" << endl;

obj.delete\_at\_end();

obj.print();

cout << "Delete at Index" << endl;

obj.delete\_at\_index(3);

obj.print();

obj.~singlylist();

return 0;

} ****

**Doubly Linked List**

#include<iostream>

using namespace std;

struct node {

int data;

node\* next;

node\* prev;

node() :next(nullptr),prev(nullptr),data(0){}

};

class BaseDoubly {

protected:

node\* head;

node\* current;

node\* tail;

int count;

public:

BaseDoubly() {

head = nullptr;

tail = nullptr;

count = 0;

}

virtual void insert\_at\_front(int val) = 0;

virtual void insert\_at\_end(int val) = 0;

virtual void insert\_at\_index(int index, int val) = 0;

virtual void delete\_at\_front() = 0;

virtual void delete\_at\_back() = 0;

virtual void delete\_at\_index(int index) = 0;

virtual void reverseDoubly() = 0;

virtual void middleDoubly() = 0;

virtual bool palindromeDoubly() = 0;

virtual void print() = 0;

};

class doublyList :public BaseDoubly{

private:

node\* head;

node\* current;

node\* tail;

int count;

public:

doublyList():BaseDoubly() {}

~doublyList() {

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

delete\_at\_back();

}

}

void insert\_at\_front(int val){

if (head == nullptr) {

head = new node;

head->next = nullptr;

head->prev = nullptr;

head->data = val;

current = head;

count++;

}

else {

node\* temp = new node;

temp->data = val;

temp->next = head;

head->prev = temp;

tail = current;

head = temp;

count++;

}

}

void insert\_at\_end(int val) {

if (head == nullptr) {

insert\_at\_front(val);

}

else {

node\* temp = new node;

temp->data = val;

current = head;

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

current = current->next;

}

current->next = temp;

temp->prev = current;

tail = current->next;

count++;

}

}

void insert\_at\_index(int index, int val) {

if (index == 1) {

insert\_at\_front(val);

}

else if (index == count) {

insert\_at\_end(val);

}

else if (index > 1 && index < count) {

current = head->next;

for (int i = 2; i < index; i++) {

current = current->next;

}

node\* temp = new node;

temp->data = val;

temp->next = current;

temp->prev = current->prev;

current->prev->next = temp;

current->next->prev = temp;

count++;

}

else {

cout << "Invalid Index" << endl;

}

}

void delete\_at\_front() {

current = head;

head = current->next;

head->prev = nullptr;

delete current;

current = head;

count--;

}

void delete\_at\_back() {

current = tail;

tail = current->prev;

tail->next = nullptr;

delete current;

current = head;

count--;

}

void delete\_at\_index(int index) {

if (index == 1) {

delete\_at\_front();

}

else if (index == count) {

delete\_at\_back();

}

else if (index > 1 && index < count) {

current = head->next;

for (int i = 2; i < index; i++) {

current = current->next;

}

current->prev->next = current->next;

current->next->prev = current->prev;

delete current;

count--;

}

else {

cout << "Invalid Index" << endl;

}

}

void reverseDoubly() {

current = head;

node\* tempPrev = tail;

for (int i = 1; i <= count / 2; i++) {

int temp = current->data;

current->data = tempPrev->data;

tempPrev->data = temp;

current = current->next;

tempPrev = tempPrev->prev;

}

}

void middleDoubly() {

current = head;

node\* temp = tail;

for (int i = 1; i <= count/2; i++) {

current = current->next;

temp = temp->prev;

}

cout << "Middle Element : " << current->data << endl;

cout << "Middle Element : " << temp->data << endl;

}

bool palindromeDoubly() {

current = head;

node\* temp = tail;

for (int i = 1; i <= count / 2; i++) {

current = current->next;

temp = temp->prev;

if (current->data != temp->data) {

return false;

}

}

return true;

}

void print() {

current = head;

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

cout << "[" << current->data << "]";

current = current->next;

}

cout << endl;

}

};

int main() {

doublyList obj;

cout << "Insert at Front" << endl;

for (int i = 5; i > 0; i--) {

obj.insert\_at\_front(i);

}

obj.print();

cout << "Insert at End" << endl;

for (int i = 1; i < 6; i++) {

obj.insert\_at\_end(i);

}

obj.print();

cout << "Reverse Doubly" << endl;

obj.reverseDoubly();

obj.print();

cout << "Delete at Back" << endl;

obj.delete\_at\_back();

obj.print();

cout << "Delete at Front" << endl;

obj.delete\_at\_front();

obj.print();

cout << "Delete at Index" << endl;

obj.delete\_at\_index(2);

obj.print();

cout << "Middle Doubly" << endl;

obj.middleDoubly();

cout << "Insert at Index" << endl;

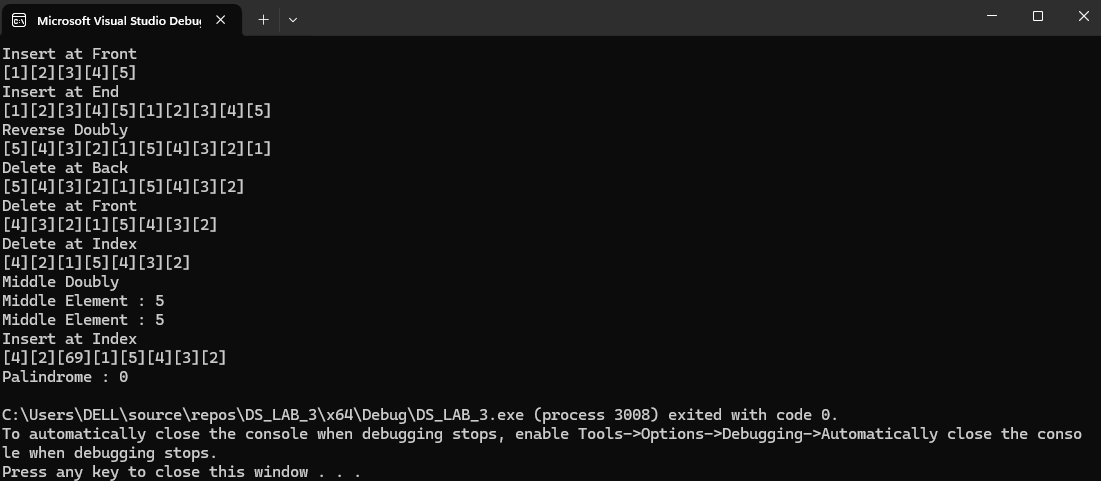
obj.insert\_at\_index(3, 69);

obj.print();

cout << "Palindrome : " << obj.palindromeDoubly() << endl;

return 0;

}

****

**Sub-Linked List**

#include<iostream>

using namespace std;

struct node {

int data;

node\* next;

node\* bottom;

int subNodeCount;

node() :data(0), next(nullptr), bottom(nullptr),subNodeCount(0) {};

};

class SubLinkedList {

private:

node\* current;

node\* head;

int count;

public:

SubLinkedList() {

current = nullptr;

head = nullptr;

count = 0;

}

void insert\_at\_end(int val) {

if (head == nullptr) {

head = new node;

current = head;

head->data = val;

head->next = nullptr;

count++;

}

else {

current = head;

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

current = current->next;

}

node\* temp = new node;

temp->data = val;

temp->next = nullptr;

current->next = temp;

count++;

}

}

void insert\_at\_bottom(int index, int val) {

current = head;

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

current = current->next;

}

current->subNodeCount++;

if (current->bottom == nullptr) {

current->bottom = new node;

current->bottom->data = val;

}

else {

current = current->bottom;

while (current->next != nullptr) {

current = current->next;

}

node \*temp= new node;

temp->data = val;

current->next = temp;

}

}

void sort(node \*temp,int n) {

current = temp;

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

current = temp;

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

if (current->data >= current->next->data) {

int tempData = current->data;

current->data = current->next->data;

current->next->data = tempData;

}

current = current->next;

}

}

}

void sortList() {

current = head;

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

if (current->bottom != nullptr) {

sort(current->bottom, current->subNodeCount);

}

current = current->next;

}

sort(head, count);

}

void printList() {

current = head;

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

cout << "[" << current->data << "]->";

current = current->next;

}

cout << "X" << endl;

}

void printSorted() {

current = head;

SubLinkedList obj;

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

obj.insert\_at\_end(current->data);

if (current->bottom != nullptr) {

node\* temp = current->bottom;

for (int j = 1; j <= current->subNodeCount; j++) {

obj.insert\_at\_end(temp->data);

temp = temp->next;

}

}

current = current->next;

}

obj.sortList();

obj.sortList();

obj.printList();

}

};

int main() {

SubLinkedList obj;

for (int i = 1; i < 4; i++) {

obj.insert\_at\_end(i);

}

for (int i = 7; i >= 3; i--) {

obj.insert\_at\_bottom(2, i);

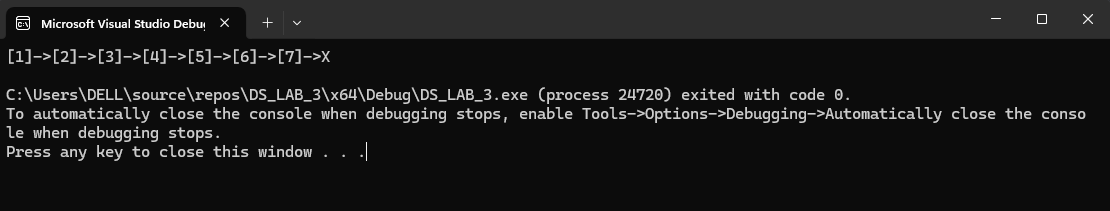
}

obj.sortList();

obj.printSorted();

return 0;

}

****