Double Link List

//Dou_node.h

```
#include <iostream>
using namespace std;

#ifndef NODE_H
#define NODE_H



template<typename T> class Node{
	public:
		T data;
		Node<T>* nxt,*prev;
};




#endif /* NODE_H */
```

//Doublelist.h

```
#include "dou_node.h"
#ifndef DLL_H
#define DLL_H
template<typename T> class Dll{
	private:
		Node<T>* head,*tail;   //head AND tail of the list
		Node<T>* create(T n); //creates and returns a node
	 Dll<T> newCopy(Dll ob);  //creates and returns a copy of list
```

```
public:
        Dll() { head = NULL; }
    //INSERTION METHODS
        void insert_beg(T n);
        void insert_end(T n);
        void insert_pos(int pos,T n);
    //DELETION METHODS
        void del_beg();
        void del_end();
        void del_pos(int pos);
    int del_first_elem(T n);    //DELETS FIRST ELEM THAT MATCHES(return -1 if not found else the pos of found elements)
        void del_elements(T n); //DELETES ALL ELEMENTS THAT MATCHES
    //----------
        Node<T>* search(T n);   //SEARCHES FOR A NODE
        void show_list();       //PRINTS THE LIST
        void reverse();         //REVERSE LIST(CHANGES ORIGINAL)
        void merge(Dll ob);     //MERGES TWO LISTS(FIRST LIST CHANGES)
        Dll<T> get_union(Dll ob);   //NO LIST CHANGES
        Dll<T> get_intersection(Dll ob); //NO LIST CHANGES
        Node<T>* mid_elem();    //reach the middle element in single transeversal
        int nodes_count();      //counts the no of nodes
        Dll<T> operator+(Dll<T>& r);   //OVERLOADED + OP, MERGES TWO LISTS TO CREATE A NEW LIST(ORIGINAL LISTS DONT CHANGE)
    Node<T>* getHead();         //RETURN HEAD OF LIST
    Node<T>* getTail();         //RETURN TAIL OF LIST
    void remove_duplicates();   //REMOVES DUPLICATE ELEMENTS(CHANGES LIST)
        bool isPresent(T data,Dll ob); //checks whether data is present in the list or not
};

template<typename T> Node<T>* Dll<T>::create(T n){
    Node<T> *tmp = new Node<T>;
```

```
            tmp->data = n;
            tmp->nxt = NULL;
        tmp->prev = NULL;
            return tmp;
}

template<typename T> Dll<T> Dll<T>::newCopy(Dll ob){
   Dll<T> new_list;
   Node<T>* first = ob.getHead();
   while(first->nxt != NULL){
      new_list.insert_end(first->data);
      first = first->nxt;
   }
   new_list.insert_end(first->data);
   return new_list;
}

template<typename T> void Dll<T>::insert_beg(T n){
            Node<T> *tmp = create(n);    //creating a node
            if(head == NULL){  //in case list is empty
          //POINTS TO SAME ELEMENT
          head = tmp;
          tail = tmp;
       }
            else{
          tmp->nxt = head;
          head->prev = tmp;
          head = tmp;
            }
}
```

```
template<typename T> void Dll<T>::insert_end(T n){
          Node<T> *tmp = create(n);    //creating a node
          if(head == NULL){  //in case list is empty
        //POINTS TO SAME ELEMENT
        head = tmp;
        tail = tmp;
      }
          else{
        tail->nxt = tmp;
        tmp->prev = tail;
        tail = tmp;
          }
}

template<typename T> void Dll<T>::insert_pos(int pos,T n){
          if(head == NULL) cout<<"error: list is empty(out of bounds)"<<endl;
          else if(pos > 1){
                    if(pos < nodes_count()+2){
                              Node<T> *tmp = create(n);    //creating a node
                              Node<T> *tvs = head;
                              int count = 1;              //keeps count of postions
                              while(count < pos){ //forward traversal(REACHES TO POS)
                                        count++;
                                        tvs = tvs->nxt;     //advances pointer      (POS)
                              }
                              tmp->nxt = tvs;
                    tmp->prev = tvs->prev;
                    tvs->prev->nxt = tmp;
                    tvs->prev = tmp;
                       }
                       else cout<<"error: position out of bound"<<endl;
```

```
        }
        else if(pos == 1) insert_beg(n); //if at pos 1
        else cout<<"error: position doesn't exist(starts from 1)"<<endl;
}

template<typename T> void Dll<T>::del_beg(){
        if(head == NULL) cout<<"error:list is empty."<<endl;  //in case list is empty
        else{
                Node<T>* tmp = head;
                head = head->nxt;
         head->prev = NULL;
                delete tmp;
        }
}

template<typename T> void Dll<T>::del_end(){
   if(head == NULL) cout<<"error:list is empty."<<endl;  //in case list is empty
   else{
        Node<T>* tmp = tail;
        tail = tail->prev;
      tail->nxt = NULL;
        delete tmp;
   }
}

template<typename T> void Dll<T>::del_pos(int pos){
   if( (pos > 0) && (pos <= nodes_count() )){
        if(pos == 1) del_beg();
      else if(pos == nodes_count()) del_end();  //if last elem
      else{
         Node<T>* tmp = head;
```

```
        for(int i=0;i<pos-2;i++) tmp = tmp->nxt; //forward traversal(tmp is the node just before the node to be deleted)
        tmp->nxt = tmp->nxt->nxt;
        delete tmp->nxt->prev;
        tmp->nxt->prev = tmp;
      }
   }
   else cout<<"error: node doesn't exist."<<endl;
}

template<typename T> int Dll<T>::del_first_elem(T n){
   Node<T>* cur = head;    //current node
   int pos = 1;
   while( (cur->data != n) && (cur->nxt != NULL) ){ //if an elem matches loop stops => temp = that node(delete),prev = node before that node
      cur = cur->nxt;    //next node
      pos++;
   }
   if((cur->data != n) && (cur->nxt == NULL)) return -1;  //if no match
   else{ //deleting
      del_pos(pos);
      return pos;
   }
}

template<typename T> void Dll<T>::del_elements(T n){
      int i = del_first_elem(n); //delete the first match
      //deleting more matches
      if(i > 0){
         for(i;i <= nodes_count();i = del_first_elem(n)){//if deleted elem is not the last of the list(still possibilities for match
            if(i < 0) break;
```

```
        }
    }
    else cout<<"ERROR: no matching elements found."<<endl;
}

template<typename T> Node<T>* Dll<T>::search(T n){
   int pos = 1;
   Node<T>* tmp = head;
   while(tmp->nxt != NULL){
      if(tmp->data == n){
         cout<<"NODE FOUND! (pos : "<<pos<<" )"<<endl;
         return tmp;
      }
      tmp = tmp->nxt;
      pos++;
   }
   if(tmp->data == n){ //for tail
         cout<<"NODE FOUND! (pos : "<<pos<<" )"<<endl;
         return tmp;
   }
   else cout<<"NODE NOT FOUND!"<<endl;
}

template<typename T> void Dll<T>::show_list(){
    cout<<"list : ";
        Node<T>* tmp = head;
        if(head == NULL) cout<<"list is empty"<<endl;
        else{
                while(tmp != NULL){ //INCLUDES TAIL
                        cout<<tmp->data<<" -> ";
                        tmp = tmp->nxt;       //points to tail at end
```

```
                }
                cout<<endl;
            }
}

template<typename T> void  Dll<T>::reverse(){
    if (head == NULL){
        cout <<"LIST IS EMPTY.";
        return;
    }
    Node<T>* current = head;
    Node<T>* tmp = NULL;
    //SWAP NEXT AND PREVIOUS FOR ALL NODES
    while(current != NULL){
        tmp = current->prev;
        current->prev = current->nxt; //reverse current node's pointer
        current->nxt = tmp;
        current = current->prev;
    }
    Node<T>* temp = head;
    head = tail;
    tail = temp;
}

template<typename T> void Dll<T>::merge(Dll ob){
    if(head == NULL && ob.getHead() != NULL){
        head = ob.getHead();
        tail = ob.getTail();
    }
    else if(ob.getHead() == NULL && head != NULL) {}//do nothing
    else if(head == NULL && ob.getHead() == NULL) {} //do nothing
```

```
    else{
        tail->nxt = ob.getHead(); //tail points to next list
        ob.getHead()->prev = tail; //second list points to first list
        tail = ob.getTail();
    }
}

template<typename T> Dll<T> Dll<T>::get_union(Dll ob){
    Dll<T> res_list,first_list = newCopy(*this),second_list = newCopy(ob); //copying lists to prevent any change in original lists
    first_list.remove_duplicates();
    second_list.remove_duplicates();

    Node<T>* snd = second_list.getHead();
    res_list.merge(first_list); //ADDS ALL NODES OF FIRST IN NEW LIST
    while(snd != NULL){ //ADDS NODES OF SECOND THAT ARE NOT PRESENT IN LIST BEFORE(TAIL INCLUDED)
        if(!isPresent(snd->data,res_list)) res_list.insert_end(snd->data);
        snd = snd->nxt;
    }
    return res_list;
}

template<typename T> Dll<T>  Dll<T>::get_intersection(Dll ob){
    Dll<T> res_list,first_list = newCopy(*this),second_list = newCopy(ob); //copying lists to prevent any change in original lists
    first_list.remove_duplicates();
    second_list.remove_duplicates();
    Node<T>* fst = first_list.getHead();

    while(fst != NULL){//CHECKS THE NODES THAT ARE COMMON TO BOTH LISTS(INCLUDES TAIL ALSO)
```

```
        if(isPresent(fst->data,second_list)) res_list.insert_end(fst->data);
        fst = fst->nxt;
    }
    return res_list;
}

template<typename T> Node<T>*  Dll<T>::mid_elem(){
    //IF NO OF NODES IS EVEN => MID ELEMENT = N/2; (CAN USE THIS TO GET SECOND MID AT N/2 +
1)
    //IF NO OF NODES IS ODD => MID ELEMENT = N+1/2; (ONLY MID)
    Node<T>* slow = head;
    Node<T>* fast = head;
    int count = 0;
    while(fast != NULL && slow != NULL){
        if(fast->nxt != NULL)fast = fast->nxt->nxt;  //to prevent breakdown at tail in odd nodes list
        else fast = fast->nxt;
        slow = slow->nxt;
        count++;
    }
    cout<<"MID POSITION : "<<count<<endl;
    return slow;
}

template<typename T> int Dll<T>::nodes_count(){
            Node<T>* tmp = head;
        if(tmp == NULL) return 0;
            int count = 1;
            while(tmp->nxt != NULL){
                        tmp = tmp->nxt;
                        count++;
            }
```

```
        return count;
}

template<typename T> Dll<T> Dll<T>::operator+(Dll<T>& r){
   Dll<T> res_list,first_list = newCopy(*this),second_list = newCopy(r); //copying lists to prevent any change in original lists
   res_list.merge(first_list);
   res_list.merge(second_list);
   return res_list;
}

template<typename T> Node<T>* Dll<T>::getHead(){
   return head;
}

template<typename T> Node<T>* Dll<T>::getTail(){
   return tail;
}

template<typename T> void Dll<T>::remove_duplicates(){
   Node<T>* tmp = head;
   Node<T>* tmp2;
   while( tmp != NULL ){ //also checks tail(no need) --->selects elements (one at a time)
      tmp2 = tmp->nxt;
      while(tmp2 != NULL){
         if(tmp->data == tmp2->data){
            Node<T>* next = tmp2->nxt;
            if(head == tmp2) head = tmp2->nxt;
            if(tmp2->nxt != NULL) tmp2->nxt->prev = tmp2->prev;
            if(tmp2->prev != NULL) tmp2->prev->nxt = tmp2->nxt;
            delete tmp2;
```

```
            tmp2 = next; //UPADATE
          }
          else tmp2 = tmp2->nxt;
        }
        tmp = tmp->nxt;
    }
}

template<typename T> bool Dll<T>::isPresent(T data,Dll ob){
            Node<T>* tmp = ob.getHead();
            while(tmp->nxt != NULL){
                        if(tmp->data == data) return true;
                        tmp = tmp->nxt;
            }
            if(tmp->data == data) return true; //TAIL
            return false;
}

#endif /* DLL_H */
```

//Doublelinklist.cpp

```
#include "doublelinklist.h"

int main() {
    //LIST 1
    Dll<int> dll;
    dll.insert_end(20);
    dll.insert_beg(1);
    dll.insert_end(11);
```

```
dll.insert_pos(2,50);
dll.insert_end(20);
dll.insert_end(20);
dll.insert_end(35);
cout<<"list 1 :";
dll.show_list();
cout << "--------" << endl;
//LIST 2
Dll<int> dll1; //11 10 12
dll1.insert_end(11);
dll1.insert_end(20);
dll1.insert_end(12);
cout<<"list 2 :";
dll1.show_list();
cout << "--------" << endl;

//LIST 3,4 => UNION AND INTERSECTION OF LIST1 AND LIST2
Dll<int> L3,L4;
L3 = dll.get_union(dll1);
L4 = dll.get_intersection(dll1); //10 12
cout<<"union of List 1 AND List 2 (List 3):";
L3.show_list();
cout<<"intersection of List 1 AND List 2 (List 4):";
L4.show_list();
cout << "--------" << endl;
```

```
//REVERSE L3,MID OF L3 AND SEARCH IN L3
Node<int>* mid = L3.mid_elem(); //3
mid = L3.search(11); //2
L3.reverse();
cout<<"REVERSE OF List 3 :";
L3.show_list();
cout << "--------" << endl;

//MERGE() ON L3 AND L4
cout<<"List 3 AND List 4 BEFORE MERGING ->"<<endl;
L3.show_list();
L4.show_list();
L3.merge(L4);
cout<<"MERGE OF List 3 AND List 4 (List 3):";
L3.show_list();
cout<<"List 3 AND List 4 AFTER MERGING ->"<<endl;
L3.show_list();
L4.show_list();
cout << "--------" << endl;

//MERGING USING +
cout<<"List 3 AND List 4 BEFORE MERGING ->"<<endl;
L3.show_list();
L4.show_list();
cout<<"MERGE OF List 3 AND List 4 (List 5):";
Dll<int> l5;
```

```
l5 = L3 + L4;  //MERGE IN NEW
l5.show_list();
cout<<"List 3 AND List 4 AFTER MERGING ->"<<endl;
L3.show_list();
L4.show_list();
cout << "--------" << endl;

//DELETION ON L3
cout<<"PERFORMING DELETE OP ON List 3 "<<endl;;
L3.del_pos(3); //12 REMOVED (11 200 10 101 1 10 12)
cout<<"after del_pos(3) : ";
L3.show_list();

L3.del_beg(); //11 REMOVED (200 10 101 1 10 12)
cout<<"after del_beg() : ";
L3.show_list();

L3.del_end(); //12 REMOVED (200 10 101 1 10)
cout<<"after del_end() : ";
L3.show_list();

L3.del_elements(10); //Both 10 REMOVED (200 101 1)
cout<<"after del_elements(10) : ";
L3.show_list();

return 0;
```

}

Output

```
list 1 :list : 1 -> 50 -> 20 -> 11 -> 20 -> 20 -> 35 ->
--------
list 2 :list : 11 -> 20 -> 12 ->
--------
union of List 1 AND List 2 (List 3):list : 1 -> 50 -> 20 -> 11 -> 35 -> 12 ->
intersection of List 1 AND List 2 (List 4):list : 20 -> 11 ->
--------
MID POSITION : 3
NODE FOUND! (pos : 4 )
REVERSE OF List 3 :list : 12 -> 35 -> 11 -> 20 -> 50 -> 1 ->
--------
List 3 AND List 4 BEFORE MERGING ->
list : 12 -> 35 -> 11 -> 20 -> 50 -> 1 ->
list : 20 -> 11 ->
MERGE OF List 3 AND List 4 (List 3):list : 12 -> 35 -> 11 -> 20 -> 50 -> 1 -> 20 -> 11 ->
List 3 AND List 4 AFTER MERGING ->
list : 12 -> 35 -> 11 -> 20 -> 50 -> 1 -> 20 -> 11 ->
list : 20 -> 11 ->
--------
List 3 AND List 4 BEFORE MERGING ->
list : 12 -> 35 -> 11 -> 20 -> 50 -> 1 -> 20 -> 11 ->
list : 20 -> 11 ->
MERGE OF List 3 AND List 4 (List 5):list : 12 -> 35 -> 11 -> 20 -> 50 -> 1 -> 20 -> 11 -> 20 -> 11 ->
List 3 AND List 4 AFTER MERGING ->
list : 12 -> 35 -> 11 -> 20 -> 50 -> 1 -> 20 -> 11 ->
list : 20 -> 11 ->
--------
PERFORMING DELETE OP ON List 3
after del_pos(3) : list : 12 -> 35 -> 20 -> 50 -> 1 -> 20 -> 11 ->
after del_beg() : list : 35 -> 20 -> 50 -> 1 -> 20 -> 11 ->
after del_end() : list : 35 -> 20 -> 50 -> 1 -> 20 ->
ERROR: no matching elements found.
after del_elements(10) : list : 35 -> 20 -> 50 -> 1 -> 20 ->


...Program finished with exit code 0
Press ENTER to exit console.
```