Single Link List

//Node.h file

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

#ifndef NODE_H
#define NODE_H



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




#endif /* NODE_H */
```

Linklist.h

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

```
public:
        Sll() { 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(Sll ob);    //MERGES TWO LISTS(FIRST LIST CHANGES)
        Sll<T> get_union(Sll ob);   //NO LIST CHANGES
        Sll<T> get_intersection(Sll ob); //NO LIST CHANGES
        Node<T>* mid_elem();     //reach the middle element in single transeversal
        int nodes_count();      //counts the no of nodes
        Sll<T> operator+(Sll<T>& r);   //OVERLOADED + OP, MERGES TWO LISTS TO CREATE A NEW LIST(ORIGINAL LISTS DONT CHANGE)
 Node<T>* getHead();         //RETURN HEAD OF LIST
 void remove_duplicates();    //REMOVES DUPLICATE ELEMENTS(CHANGES LIST)
        bool isPresent(T data,Sll ob); //checks whether data is present in the list or not
};


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

```
        tmp->nxt = NULL;
        return tmp;
}

template<typename T> Sll<T> Sll<T>::newCopy(Sll ob){
   Sll<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 Sll<T>::insert_beg(T n){
        Node<T> *tmp = create(n);    //creating a node
        if(head == NULL) head = tmp;   //in case list is empty
        else{
                //create a node that points to the head
                Node<T> *tmp = new Node<T>;
                tmp->data = n;
                tmp->nxt = head; //points to head(became the first element)
                head = tmp; //updates head
        }
}

template<typename T> void Sll<T>::insert_end(T n){
        Node<T> *tmp = create(n);    //creating a node
        if(head == NULL) head = tmp;   //in case list is empty
        else{
```

```
                //forward traversal
                Node<T>* cpy = head;  //upadated to tail node in while loop
                while(cpy->nxt != NULL) cpy = cpy->nxt; //reach the tail of list(cpy is the tail node at
last)
                cpy->nxt = tmp;   //add the data at the node pointed by cpy->nxt
        }
}

template<typename T> void Sll<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-1){ //forward traversal
                                count++;
                                tvs = tvs->nxt;      //advances pointer
                        }
                        tmp->nxt = tvs->nxt; //sets the next node after it
                        tvs->nxt = tmp;  //adds the data to required pos
                }
                else cout<<"error: position out of bound"<<endl;
        }
        else if(pos == 1) insert_beg(n); //if at pos 1
        else cout<<"error: postition doesn't exist(starts from 1)"<<endl;
}

template<typename T> void Sll<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;
                delete tmp;
        }
}

template<typename T> void Sll<T>::del_end(){
        Node<T>* tmp = head;
        for(int i = nodes_count();i > 2;i--) tmp = tmp->nxt; //forward traversal
        delete tmp->nxt;
        tmp->nxt = NULL;
}

template<typename T> void Sll<T>::del_pos(int pos){
   if( (pos > 0) && (pos <= nodes_count() )){
        if(pos == 1) del_beg();
      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)
        Node<T>* tmp2 = tmp->nxt; //node to be deleted
        tmp->nxt = tmp->nxt->nxt; //updating pointer(node after deleted node)
        delete tmp2;
      }
   }
   else cout<<"error: node doesn't exist."<<endl;
}

template<typename T> int Sll<T>::del_first_elem(T n){
   Node<T>* cur = head;    //current node
   Node<T>* prev = head;   //previous 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
      prev = cur;        //previous 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 Sll<T>::del_elements(T n){
      int count = 1;
      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;
            if(i > 0) count++;
         }
      }
      else cout<<"ERROR: no matching elements found."<<endl;
      if(count != 0) cout<<"Done: "<<count<<" nodes deleted."<<endl;
}

template<typename T> Node<T>* Sll<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 Sll<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  Sll<T>::reverse(){
    if (head == NULL){
        cout <<"LIST IS EMPTY.";
```

```
        return;
    }
    Node<T>* current = head;
    Node<T>* prev = NULL;
    Node<T>* next = current->nxt;
    while(current != NULL){
        current->nxt = prev; //reverse current node's pointer
        //move pointers one position ahead
        prev = current;
        current = next;
             if(next != NULL){
           next = current->nxt;    //store next
             }
    }
    head = prev;
}

template<typename T> void Sll<T>::merge(Sll ob){
    Node<T>* first = head;
    Node<T>* second = ob.getHead();
    if(first == NULL && second != NULL) head = second;
    else if(second == NULL && first != NULL) {}//do nothing
    else if(first == NULL && second == NULL) {} //do nothing
    else{
        while(first->nxt != NULL) first = first->nxt;
        first->nxt = second;   //points to head of second list
    }
}

template<typename T> Sll<T> Sll<T>::get_union(Sll ob){
```

```
   Sll<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> Sll<T>  Sll<T>::get_intersection(Sll ob){

   Sll<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>*  Sll<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 Sll<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> Sll<T> Sll<T>::operator+(Sll<T>& r){
    Sll<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>* Sll<T>::getHead(){
   return head;
}

template<typename T> void Sll<T>::remove_duplicates(){
  Node<T>* tmp = head;
   Node<T>* tmp2,*dup;
   while( tmp != NULL && tmp->nxt != NULL){ //doesnt checks tail(no need) --->selects elements (one at a time)
       tmp2 = tmp;
      while(tmp2->nxt != NULL){ //checks tail also ---> comparing with rest(only elements after that)
          if(tmp->data == tmp2->nxt->data){
             //delete_at_pos
             dup = tmp2->nxt;
             tmp2->nxt = tmp2->nxt->nxt; //updates pointer
             delete dup;
          }
          else tmp2 = tmp2->nxt;
      }
                    tmp = tmp->nxt;
   }
}

template<typename T> bool Sll<T>::isPresent(T data,Sll 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 /* SLL_H */
```

Linklist.cpp

```
#include "linklist.h"

int main() {
   //LIST 1
   Sll<int> sli;  //1 50 20 12 20 20 35
   sli.insert_end(20);
   sli.insert_beg(1);
   sli.insert_end(12);
   sli.insert_pos(2,50);
   sli.insert_end(20);
   sli.insert_end(20);
   sli.insert_end(35);
   cout<<"list 1 :";
   sli.show_list();
   cout << "--------" << endl;
   //LIST 2
   Sll<int> ob1; //11 20 12
   ob1.insert_end(11);
   ob1.insert_end(20);
   ob1.insert_end(12);
   cout<<"list 2 :";
   ob1.show_list();
```

```
cout << "--------" << endl;

//LIST 3,4 => UNION AND INTERSECTION OF LIST1 AND LIST2
Sll<int> l3,l4;
l3 = sli.get_union(ob1);
l4 = sli.get_intersection(ob1);
cout<<"union of List :";
l3.show_list();
cout<<"intersection of List :";
l4.show_list();
cout << "--------" << endl;

//REVERSE L3,MID OF L3 AND SEARCH IN L3
Node<int>* mid = l3.mid_elem();
mid = l3.search(101);
l3.reverse();
cout<<"REVERSE OF List after Union :";
l3.show_list();
cout << "--------" << endl;

//MERGE() ON L3 AND L4
cout<<"L1(new) AND L2(new) BEFORE MERGING ->"<<endl;
l3.show_list();
l4.show_list();
l3.merge(l4);
cout<<"MERGE OF L1(new) AND L2{new} :";
l3.show_list();
cout<<"L1{new} AND L2{new} AFTER MERGING ->"<<endl;
l3.show_list();
l4.show_list();
cout << "--------" << endl;
```

```
//MERGING USING +
cout<<"L1{new} AND L2{new} BEFORE MERGING ->"<<endl;
l3.show_list();
l4.show_list();
cout<<"MERGE OF L1{new} AND L2{new} (L'):";
Sll<int> l5;
l5 = l3 + l4;  //MERGE IN NEW
l5.show_list();
cout<<"L1 {new} AND L2 {new} AFTER MERGING ->"<<endl;
l3.show_list();
l4.show_list();
cout << "--------" << endl;

//DELETION ON L3
cout<<"<------- PERFORMING DELETE OP ON L1{new}-------->"<<endl;;
l3.del_pos(3);
cout<<"after del_pos(2) : ";
l3.show_list();

l3.del_beg();
cout<<"after del_beg() : ";
l3.show_list();

l3.del_end();
cout<<"after del_end() : ";
l3.show_list();

l3.del_elements(10);
cout<<"after del_elements(10) : ";
l3.show_list();
```

```
    return 0;
}
```

Output