W2 Lab Practice

Challenge 1-insert at the Front :

insert node at the start of of a linked list the complexity is O(1)
it is easier compare to inserting at index 0 because let's say the array at the 0 index is occupied we need to shift but for linked list we can just create new node and point it to the head and let new node become head and we don't need to shift the element in the front

```
void insert_front( const int& value){
 Node*n= new Node(value);
 n->next=head_;    // this new node point to old head
 head_=n;          // head is now new node
 ++size_;

 }
```

a

Challenge 2 – insert at the End

for this one insert at the end for linked list without tail we need to traverse to the last node and point that node to the new node that we want to insert at the end
so O(n)

```
void insert_end(const int& value){
   if(empty()){
    insert_front(value);
    return;
  }
  curr_=head_;
  while(curr_->next){
        curr_=curr_->next;
  }
  Node*n= new Node(value);
    curr_->next=n;
    ++size_;



}
```

Challenge 3 – insert at the middle

for inserting in the middle of the node it is easy if we have the size of linked list now for this we need to traverse to the node before the middle let's say prev , when we got it
we let the new node next pointer point to the prev next pointer and next the prev next pointer point to new node.
Compare to shifting in the array is that for array we don't need to traverse but still need shifting but linked we only need to traverse to find the position and update the two pointer no shifting is needed.

```
void insert_at(const int& value , int position){
   if(position <0 || position > size_) { throw out_of_range("insert_at: position out of range"); }
   if(position==0) { insert_front(value); return; }
   if(position==size_) { insert_end(value);  return; }  //    in case the position is equal to size we can just insert at the end

   Node* prev=head_;
   for(int i=0; i<position-1; ++i){
       prev=prev->next;
       }

  Node* n= new Node(value);
  n->next=prev->next;
  prev->next=n;
  ++size_;


}
```

Challenge 4 - Delete from the front

for delete from the front node we have like temp equal to the current head and we update head by
let it point to the next node and we delete temp which is the current head. Also for the old head memory by using delete it already free the memory. O(1)

```
void remove_front(){
 if(empty()) return ;
  Node* temp = head_; // store old head
  head_=head_->next;  // move head  forward
  delete temp;      // delete old hed
  --size_;

}
```

Challenge 5 - Delete from the end

for delete from the end we also need to traverse to find the node before the last node , let's say it curr by delete curr next pointer meaning we delete the last node of the linked list and we point the curr next pointer to nullptr . O(n)

```
void remove_end(){
 if(empty()){ return;}
 if(size_==1) {remove_front();  return;}
 curr_=head_;
 while(curr_->next->next!=nullptr){
   curr_=curr_->next;
 }
 delete curr_->next;
 curr_->next=nullptr;
 --size_;


}
```

Challenge 6 – Delete from the middle

for deleting from the middle we need to traverse to the node before the middle node and then we
create the node to store the middle node let's say victim and then we update the prev next pointer let it point to victim next pointer like letting prev next pointer skip the victim node that we want to delete and we can safely delete the victim node . O(n)
also if you forgot to free the memory it will cause memory leak like something will cause the linked list to break;

```
void remove_at(int position){
   if(position<0 || position>=size_) { throw out_of_range("remove_at:out of range");}
   if(position==0) {remove_front(); return;}
   if(position==size_-1){ remove_end(); return;}

   Node* prev=head_;
   for(int i=0;i<position-1;i++){
     prev=prev->next;
   }
   Node* victim=prev->next;
   prev->next=victim->next;
   delete victim;
   --size_;
}
```

Challenge 7 – Traverse the list

it is different because traverse in the linked list we need to start from the head every time if you want to do any operation but for array we can access the value by knowing the index which is easier from random access

Challenge 8 - Swap two node

I write 3 case just to be careful about it for but I already test it out the last one work for all case like when swap two node near each other and when swap node y come before x etc … , so let just talk about the last condition so first we check is x and y equal or not if true we can just return no need to swap and if not loop to find x and y if it is there store in currX and CurrY and also the prev node before x and y
next we let prevX connect to currY and prevY connect to currX and then swapping happen by store currY next node in temp and swap with each other

```
void Swap(const int& x , const int& y){
      if(x==y) {return;}

          Node *prevX=nullptr , *prevY=
nullptr;
          Node *currX=nullptr, *currY=nullptr
;
          curr_=head_;
          while(curr_!=nullptr){
                if(curr_->data==x){
                 currX=curr_;
                 break;
                }
                prevX=curr_;
                curr_=curr_->next;

          }
          curr_=head_;
          while(curr_!=nullptr){
               if(curr_->data==y){
                currY=curr_;
                break;
               }
               prevY=curr_;
               curr_=curr_->next;

          }
           if(currX==nullptr || currY==
nullptr){
            return;
          }

          if(currX->next==currY){

           if(prevX){
             prevX->next=currY;
           }else{
             head_=currY;

           }

           currX->next=currY->next;
           currY->next=currX;
        }

        else if(currY->next==currX){
                if(prevY){
                  prevY->next=currX;
                }
                else {
                  head_=currX;
                }
                currY->next=currX->next;
                currX->next=currY;

          }


           else {

          if(prevX!=nullptr){
                prevX->next=currY;
          }
          else{
            head_=currY;
          }

          if(prevY!=nullptr){
                prevY->next=currX;
          }
          else{
            head_=currX;
          }

          Node* temp=currY->next;
          currY->next=currX->next;
          currX->next=temp;
           }

    }
```

Challenge 9 – Search in linked

it is similar to linear search in the array because we need traverse every time when we want to access anything the best case is when something we want access is the head which make it O(1)
but what if it is in the middle or something it will be O(n) ,also the worst can be what is the value we search for is not in there . for random access array is faster because we directly access using index .

Challenge 10 – compare with array

Operation	Linked List	Array
Insert in the front	O(1)	O(1) empty O(n) occupied
Insert in the middle	O(n)	O(n)
Insert in the end	O(n) no tail , O(1) have tail	O(1)
Remove in the front	O(1)	O(n)
Remove in the middle	O(n)	O(n)
Remove in the end	O(n)	O(1)