# LINKED LIST ①

## Insertion Sort

```
Node* insertionSort(Node* head) {
    Node* dummy = new Node(10000);
    while (head){
        Node* next = head->next;
        Node* temp = dummy;
        while (temp->next && temp->next->val < head->val){
            temp = temp->next;
        }
        head->next = temp->next;
        temp->next = head;
        head = next;
    }
    return dummy->next;
}
```

```
~linkedlist()        //DESTRUCTOR
{
    while (head!=nullptr)
    {
        deleteTail();
    }
}
```

```
void insertTail(int data){
    Node* newNode = new Node{data};
    // list empty
    if (head==nullptr)
    {
        head=tail=newNode;
    }
    // list =1 or >1
    else
    {
        tail->next=newNode;
        tail=newNode;
    }
}
```

```
void print(){
    Node* current=head;
    while (current!=nullptr){
        std::cout<<current->data<<" ";
        current=current->next;
    }
    std:cout<<std::endl;
}
```

PRINT LL

```
void deleteTail()
{
    Node* current=head;
    if (head==nullptr && tail==nullptr)
    {
        return; // list empty.
    }
    else if (head==tail){
        delete head;
        head=nullptr;
        tail=nullptr;
    }
    else {
        Node * current=head
        while (current->next!=tail)
        {
            current=current->next;
        }
        delete tail;
        tail=current;
        tail->next=nullptr;
    }
}
```

Delete from Tail

```
Node* ReverseLL(Node* head)
{
    Node* prev=NULL;
    Node* curr=head;
    Node* next;
    while (curr!=NULL)
    {
        next=curr->next;
        curr->next=prev;
        prev=curr;
        curr=next;
    }
    return prev;
}
```

Reverse LL

## CIRCULAR QUEUE

```
class MyCircularQueue {
    int * arr;
    int qfront;
    int rear;
    int size;
public:
    MyCircularQueue(int n)
    {
        arr = new int[n];
        rear = -1;
        qfront = -1;
        size = n;
    }
    bool enQueue(int value){
        if (isFull()){
            return false;
        }
        if (isEmpty()){
            qfront = rear = 0;
        } else {
            rear = (rear+1) % size;
        }
        arr[rear] = value;
        return true;
    }
    bool Dequeue(){
        if (isEmpty()){
            return false;
        }
        if (qfront == rear){
            qfront = rear = -1;
        }else {
            qfront = (qfront+1) % size;
        }
        return true;
    }
    int Front(){
        if (isEmpty()){
            return -1;
        }
        return arr[qfront];
    }
    int Rear(){
        if (isEmpty()){
            return -1;
        }
        return arr[rear];
    }
    bool isEmpty(){
        return qfront == -1;
    }
    bool isFull(){
        return (qfront==0 && rear==size-1) || (qfront==rear+1);
    }
};
```

```
Node* removeElements(Node* head, int val)
{
    while (head && (head->val==val)){
        Node* tempNode = head;
        head = head->next;
        delete tempNode;
    }
    Node* prev = head, *curr = head;
    while (curr){
        if (curr->val==val){
            Node* tempNode = curr;
            prev->next = curr->next;
            curr = curr->next;
            delete tempNode;
        }
        else{
            prev = curr;
            curr = curr->next;
        }
    }
    return head;
}
```

REMOVE ALL REPEATED NODES.

```
void removeConsecutiveRepetition()
{
    Node* current=root;
    while (current->next)
    {
      if (current->x==current->next->x)
      {
        Node* temp=current->next;
        current->next=current->next->next;
        delete temp;
        temp=nullptr;
      }
      else
      {
       current=current->next;
      }
}
```

Duplication Removal

```
void mergeLLs(LL&A,LL&B,LL& merged){
    Node* head1=A.gethead();
    Node* head2=B.gethead();
    while (head1!=nullptr && head2!=nullptr){
      if (head1->m_data<head2->m_data){
          merged.insertTail(head1->m_data);
          head1=head1->m_next;
      }
      else{
          merged.insertTail(head2->m_data);
          head2=head2->m_next;
      }
}
while (head!=nullptr)|| head2!=nullptr){
    if (head1!=nullptr){
        merged.insertTail(head1->m_data);
        head1=head1->m_next;
    }
    if (head2!=nullptr){
        merged.insertTail(head2->m_data);
        head2=head2->m_next;
    }
}
```

merge Sort

* if shows error

```
void insertRoot(int x){
    Node* current=new Node(x);
    current->next=root;
    root=current;
}
```

Insertion Front

→ for every consecutive 0's merge all the nodes lying in b/w them into single node whose value is sum of all merged nodes.

```
Node* mergeNodes(Node* head){
    Node* curr=head;
    Node* newNode=NULL;
    int sum=0;
    while (curr!=NULL){
      if (curr->val!=0){
          sum+=curr->val;
      }
      else if (sum!=0 && curr->val=0){
          Node* node=new Node(sum);
          if (newNode==NULL){
              newNode=node;
              head=node;
              sum=0;
          }
          else{
              newNode->next=node;
              newNode=newNode->next;
              sum=0;
          }
      }
      curr=curr->next;
    }
    return head
}
};
```

```
void duplicateZeros (Node* curr)
{
  while (curr != nullptr)
  {
    if (curr->data == 0)
    {
      Node* newNode = new Node{0};
      newNode->next = curr->next;
      curr->next = newNode;
      curr = newNode;
    }
    curr = curr->next;
  }
}
```

duplication

```
SinglyLLNode* cancluse (SinglyLLNode* head)
{
  SinglyLL list;
  while (head != nullptr)
  {
    bool found = false;
    SinglyLLNode* current = list.head;
    while (current != nullptr)
    {
      if (current->data == head->data)
      {
        found = true;
        break;
      }
      if (found == false)
      {
        list.insert_node(head->data);
      }
      head = head->next;
    }
  }
```

-> delete all the nodes which have duplicate.

```
Node* deleteDuplicates(Node* head){
    Node* dummy = new Node{0};
    dummy->next = head;
    Node* prev = dummy;
    Node* temp = head;
    while(temp != NULL && temp->next != NULL){
      if (temp->val == temp->next->val){
        int n = temp->val;
        while (temp != NULL && temp->val == n){
            temp = temp->next;
        }
        prev->next = temp;
      }
      else{
        prev = temp;
        temp = temp->next;
      }
    }
    return dummy->next;
}
```

```
Node* reverseBetween(Node* head, int left, int righ
{
if (left==right || head==NULL || head->next==NULL)
      return head;
  }
  int n=right-left;
  Node* dummy= new Node{-1};
dummy->next= head;
Node* prev= dummy;
for (int i=0; i<left-1; i++){
      prev=prev->next;
  }
  Node* curr= prev->next;
  Node* nex;
  while (n--){
     nex = curr->next;
     curr->next = nex->next;
     nex->next = prev->next;
     prev->next = nex;
  }
return dummy->next;
}.
```

Return_Type operator*(const ClassName& obj) / operator*(ClassName obj)

→ OPERATOR OVERLOADING.

```
std::ostream& operator<<(std::ostream& out, const Point& point){
    out << "Point (" << point.getX() << ')';
    return out;
}
```

→ OUTPUT OVERLOADING.

```
std::istream& operator>>(std::istream& in, Point& point){
    in >> point.m_x;
    return in;
}
```

→ INPUT OVERLOADING

* if it's out of class then do FRIENDS

```
int** p{new int*[4]}; // this creates the array of int*
p[i] = new int[len]; // this creates the ith array of int.
```

p[i] is identical to *(p+i)

---

```
int main(){
    Fraction fcopy{f}; //Calls copy constructor
    return 0;
}

class Fraction{
    //etc
    public:
        //Copy constructor
        Fraction(const Fraction& frac)
            : m_num{frac.m_num}, m_den{frac.m_den}
        {
        }
};
```

→ copy constructor.

---

```
Fraction operator*(const Fraction& fr2)const
{
    int NumRes = getNum() * fr.getNum();
    int denRes = getDen() * fr.getDen();
    Fraction res{numRes, denRes};
    return res;
}
```

COPY CONSTRUCTOR

fraction1.operator*(fraction2)

#include <cmath>
- → max()
- → min()
- → sqrt()
- → ceil()
- → floor()
- → pow(x,y) → $x^y$

#include <cstring>
- → strcpy(s1,s2)
- → strcat(s1,s2)
- → strlen(s1)
- → strcmp(s1,s2)
- → strchr(s1,ch)
- → strstr(s1,s2)

```
void sort(char** arr, int size)
{
  for (int j=0 ; j<size;j++){
    for (int i=0 ; i<size-j-1 ; i++){
        int index=0;
        while (arr[i][index] != '\0')
        {
          if (arr[i][index]<arr[i+1][index])
          {
            break;
          }
          else if (arr[i][index]>arr[i+1][index])
          {
            char* temp = arr[i];
            arr[i] = arr[i+1];
            arr[i+1]= temp;
            break;
          }
          index++;
        }
    }
}
```

```
delete secondLastNode (Node*& head)
  while (current -> next -> next != nullptr){
        current = current->next;
  }
  Node* secondlast = current->next;
  current -> next = current->next->next;
  delete secondlast;
```

RULE OF THREE

```
~Person() --> DESTRUCTOR
{
  delete [] m_naam;
  m_naam = nullptr;
}
Person(const Person& obj)= delete --> COPY CTOR
Person& operator = (const Person& obj)= delete
```

WITHOUT USING <CSTRING>

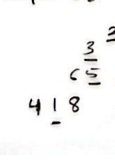
BOOL OP

```
Person(const char* naam)
{
  int i=0;
  int size=0
  while (naam[i] != '\0'){
      size++;
      i++;
  }
  m_naam = new char [size+1] {};
  for (int j=0; j<size+1;j++){
      m_naam[j]= naam[j];
  }
  m_naam[size+1] = '\0';
}
```

```
int minimumTotal (vector<vector<int>>&triangle){
  int n=triangle.size();
  vector<vector<int>>dp=triangle;
  for (int i=0 ; i<n-1; i++){
    for (int j=0 ; j<dp[i].size(); j++){
      dp[i][j] = 0;
      cout<<dp[i][j]<<" ";
    } cout<<endl;
  }
  for (int i=n-2 ; i>=0; i--){
    for (int j=0 ; j<dp[i].size(); j++){
      dp[i][j]= min(dp[i+1][j],dp[i+1][j+1]) + triangle[i][j];
    }
  }
  return dp[0][0];
}
```

e.g. [[2],[3,4],[6,5,7],[4,1,8,3]]

output 11



# CHARACTER ARRAYS

→ String Concatenation

```
char * strConcat(char*str1, char*str2):
{
  int i=0;
  int j=0;
  int size=0;
  int size2=0;
  while (str1[i] != '\0')
  {
    i++;
    size++;
  }
  while (str2[j] != '\0')
  {
    j++;
    size2++;
  }
  char *result = new char[size+size2+1];
  for (int i=0; i<size; i++){
    result[i] = str1[i];
  }
  for (int j=0; j<size2; j++){
    result[j+size] = str2[j];
  }
  result[size+size2] = '\0';
  return result;
}
```

→ Comparing Strings.

```
bool areEqual(const char*str1, const char*str2){
  int i=0;
  int j=0;
  int size=0;
  int size2=0;
  while (str1[i] != '\0')
  {
    i++;
    size++;
  }
  while (str2[j] != '\0')
  {
    j++;
    size2++;
  }
  if (size != size2)
  {
    return false;
  }
  for (int i=0; i<size; i++)
  {
    if (str1[i] != str2[j])
    {
      return false;
    }
  }
  return true;
}
```

# TEMPLATE FUNCTION

**Defining a function.**

```
template <typename T>
T functionName (T parameter1, T parameter, ...){
}       //code
```

**Calling a function.**

```
functionName <data-type> (parameter1, parameter2);
```