Link List

PAGE NO.
DATE

Disadvantage of Link list.

1) can't access random node from the list. we have to go sequentially for access that node.

2) Need extra space for pointers.

Referential class.

It is the class using to create ~~specially~~ specially in Link list or tree.

It is known as referential class because it declare a data member as a pointer to an object of the same class.

In here,

```
node *next;
```

main code:

```
#include <bits/stdc++.h>
using namespace std;

class node {
  public:
  int data;
  node *next;   // self pointer.

// constructor of node class
```

```
node (int data){
    this -> data = data;
    this -> next = NULL;
}
};

void insert at start (&node * & head, int data){
    node * new_node = new node(data);
    new_node -> next = head;
    head = new_node;
}

int main () {
    node * node1 = new node(4);
    cout << node1 -> data << endl;
    node * head = node1;
    insert at start (head, 1);
    print (head);
    return 0;
}
```

Output

1 4

define element in class.

we define element in class because after that we don't need to define data & pointer again and again. It is useful when we build custom list.

Link list

overview.

Link list is a type of list which contain the data and the address of the next node.

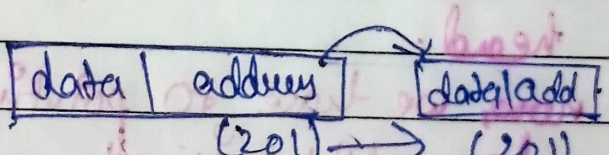
→ advantages.

1) no need to create new space
2) It is a dynamic memory.
3) It has no fixed size, we can add more nodes.
4) No shifting needed.

→ Types.

1) Singly Link list.

it contain single direction or one way direction with one data and one address section in a node.



operations on Link list

1) create a node.

```
class node {          // define using class
    public:
    int data;
    node * next;
```

node (int data) { // assign value thro
// store some data this -> data = data; // constru
// pointing to next this -> next = NULL;
node.
}
};

2) Insert node at beginning.

void (int data, node *&head) {
① —— node * temp = new node(data);
② —— temp -> next = head;
③ —— head = temp;

}

1) create a new node space for inserting
2) new nodes next pointing to the existing node head.
3) now, we have to pointing head to first node, which is the new node.

3) Insert a node at the end.

void (int data, node *&tail) {

node * new_data = new node(data);
① —— tail -> next = new_data;
② —— tail = new_data;
}

1) tail [which is last node], now tail's next pointing to the new node added.

2) And tail pointing the last node which is the new node I just added].

4) print the node or list

```
void print(*node * &head){
① —— node *temp = head;
② —— while (temp != NULL){
        cout << temp -> data << " " << endl;
③ —— temp = temp -> next;
    }
    cout << endl;
}
```

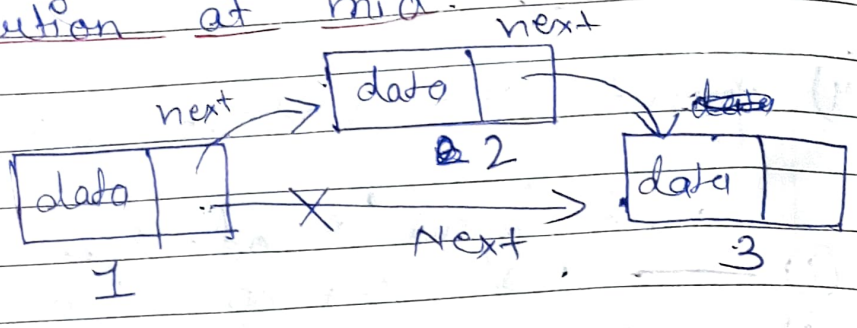
1) create a new pointer who point to the first node which indicate by "head".

2) run the loop while the temp is not equal to "NULL". [NULL represent the last node].

3) as we do in normal loop, increment the position by "1"; same we increament the temp value to the next node.

```
temp = temp -> next;
```

5) Insertion at mid.



for inserting new node,

1) the next of new node is pointing the node 3 which is pointed by the node 1.

2) And remove the next pointer of node 1 from node 3 to node 2.

code

```
void insertatmid (&node *&tail, node *&head,
int position , int data) {
    node *temp = head;
(1) if (position == 1) {
        insertat start (head, data)
        return;
    }
(2) node * temp = head;
    int count = 1;
```

```
while (count < position-1)
    {
        temp = temp -> next;
        count ++;
    }
// first insert at end
if ( tail -> next = NULL)
    {
        insert at last (tail, data);
    }
node * new_node = new node(data);
new_node -> next = temp -> next;
temp -> next = new_data;
}
```

(3)

(4)

(5)

(6)

1) if the position is 1 then we call the insertatstart function for that.
because it doesn't ~~start~~ insert at first position, it need a previous node for that and also we insert the node from one.

2) create a temp who point to the head of the list. from starting point in purpose of traversing.

3) In this portion we traverse the list because we can't access the random node.

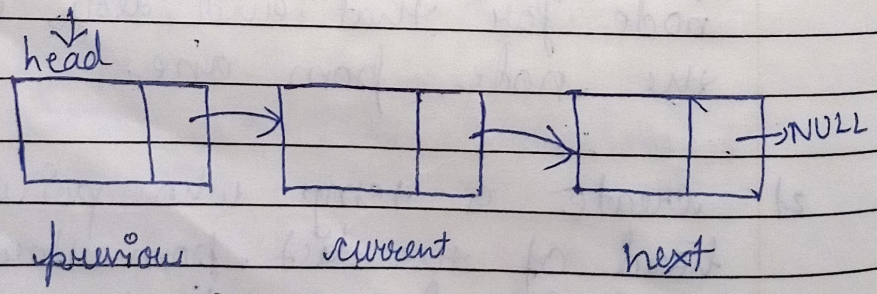
4) It is the case if the position is the last of the node. In this case we call the lastinsert function. It will work properly without this function but the tail is still not update to the next node. for that we need to call this function.

5) here we point the new node's next point to the node where previous next pointer point.

6) the previous pointer pointing to the new node who is going to be added.

deletion of link list

1) deletion from mid and last node.

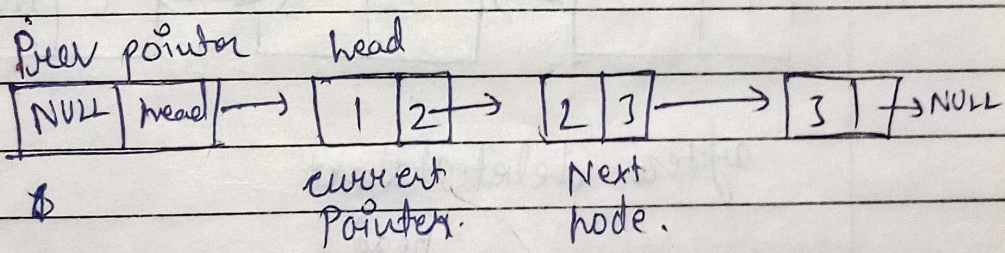


→ for deleting the current node firstly

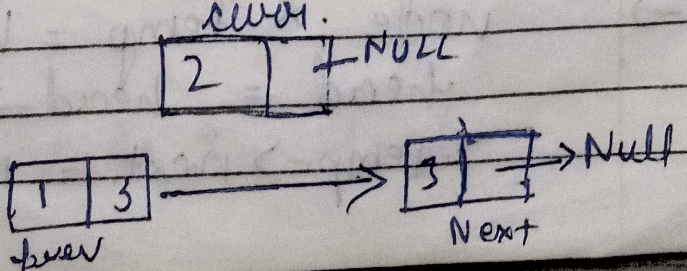
we have to pointing the previous's next to the next node and remove the current node add. from it.

→ And for doing this pointer part first we have to initilize two pointer (previous, current) and traverse the list and increament the previous and current pointer till (count == address).

* for starting from first node, previous pointer should be Null at starting. than move forward and pointing to the next node which is Head.



→ when (count == address) is arrived we do the first step remove curr add. from previous next pointer and locate to next node addres. and also NULL the pointer of curr. node.

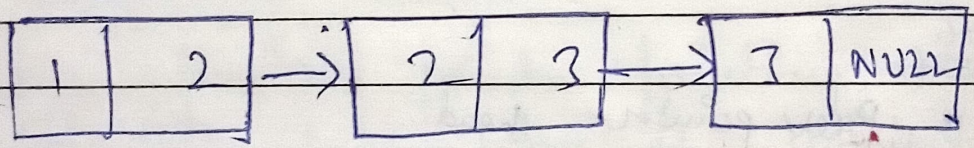


~~curr -> Next = NULL~~

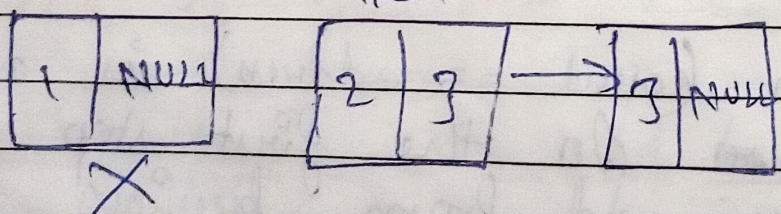
prev -> Next = curr -> Next;
curr -> Next = NULL;
delete curr;

* Deletion at starting.

(1) Deletion is not done by this processor, because we need a previous node for that, thats why we call deleteatfirst function where, we point the head the head -> next and make head->next null.



after deleteatstart



~~node * head~~

→ node * temp = head;
head = head -> next;
temp -> next = NULL;

delete temp;

main code.

```
void deletion (node *&head, int add)
{
    if (add == 1)
    {
        node *temp = head;
        head = head->next;
        temp->next = null;
        delete temp;
    }
    else{
        node *prev = NULL;
        node *curr = head;
        int count = 1;
        while (count <= add)
        {
            prev = curr;
            curr = curr->Next;
            count++;
        }

        prev->next = curr->next;
        curr->next = NULL;
        delete curr;
}
```

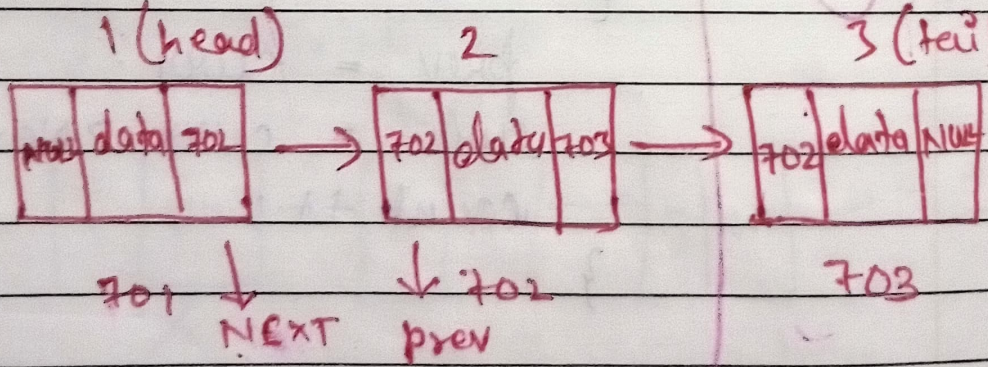
→ Destructor code in class.

```
~Node()
{
  if(this->next!=NULL)
  {
    delete next;
    next = NULL;
  }
}
```

Doubly Linked List

In double link list there is two pointer with every node

1) previous.
   These pointer pointing to the previous node.
2) next
   These pointing to the next node.



→ In this list the prev value of head is NULL.
and the next value of tail node is also NULL.

Insertion code:-

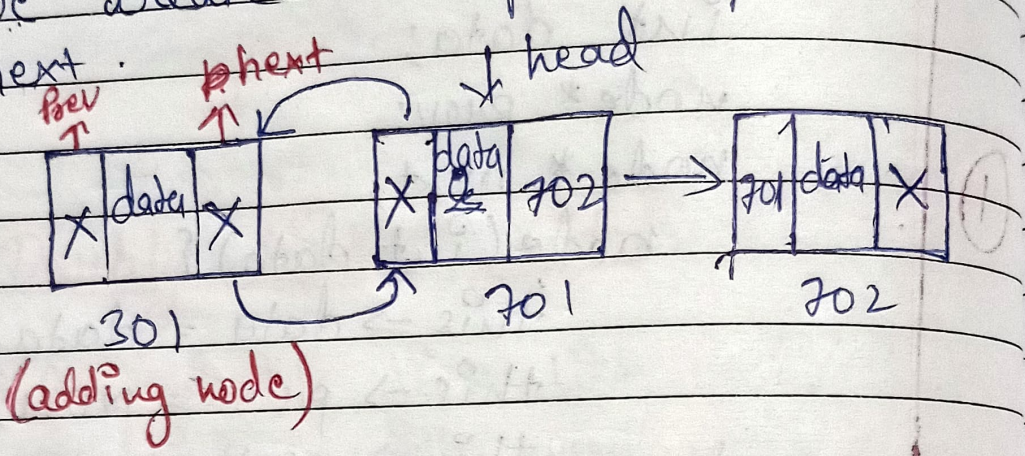
```
#include<bits/stdc++.h>
using namespace std;

class node {
    int data;
    node* prev;
    node* next;
    node(int data) {
        this -> data = data;
        this -> prev = NULL;
        this -> next = NULL;
    }
};

int main() {
    node* node1 = new node(5);
    node* head = node1;
    node* tail = node1;
    print(head);

    return 0;
}
```

① It contain three items, data for store data, prev pointer for previous node, next pointer for next node.

operations :

1) Insertion at start.

- Insert at start we create two pointer prev and next.



(adding node)

new node's next should pointing to the head.

and the head → prev = new node.

→ code.

```
void insertatstart (node * head, int data)
    {
①——  node * new_node = new node(data);

②——  new_node -> next = head;
③——  head -> prev = new_node;
④——  head = new_node;

    }
```

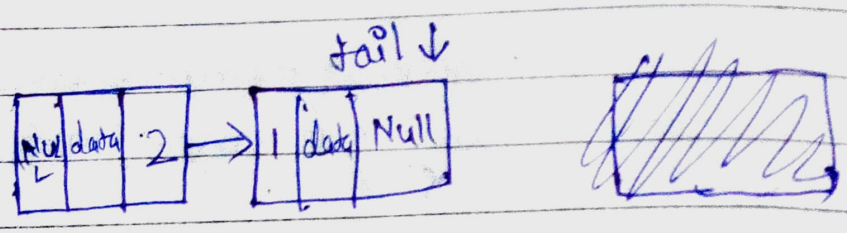
```
void print(node * & head)
{
    node *temp = head;
    while (temp->next != NULL)
    {
        cout << temp -> data << " ";
        temp = temp -> next;
    }
    cout << endl;
}

int main() {
    node *node1 = new node(4);
    node * head = node1;
    insert atstart (head, 40);
    insert atstart (head, 64);
    print(head);
    return 0;
}
```

→ output

64 40 4

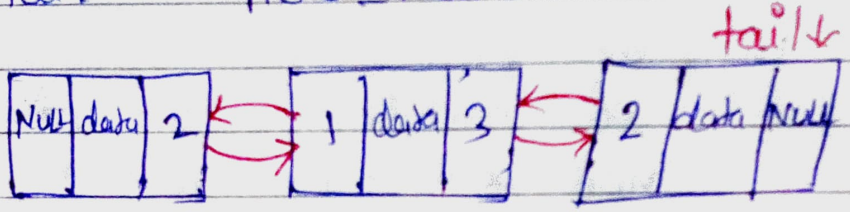
1) create new node space in heap.
2) new_node -> next point to the head.
4) ~~head~~ move head to the newly added node.
3) head -> prev pointing to the new node.

2) Insertion at end.



for inserting new node at end.

tail -> next = new_node
new_node -> prev = tail
tail = new_node.



→ code.

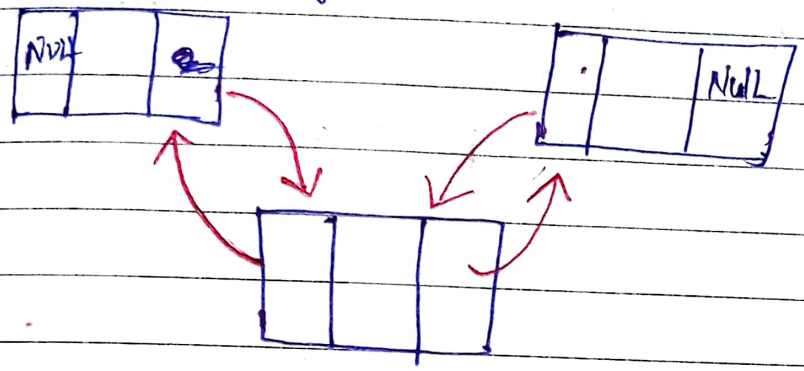
```
void insertatend (node *&tail, int d)
  {
     int temp = new node(d);
      tail -> next = temp;
      temp -> prev = tail;
       tail = temp;
  }

int main () {
   node * node1 = new node (4);
   node * tail = node1;
   insertatend (tail, 6);
   insertatend (tail, 12);
   return 0;
}
```

output

4 , 6 , 12

3) Insertion at mid.

for inserting in mid.



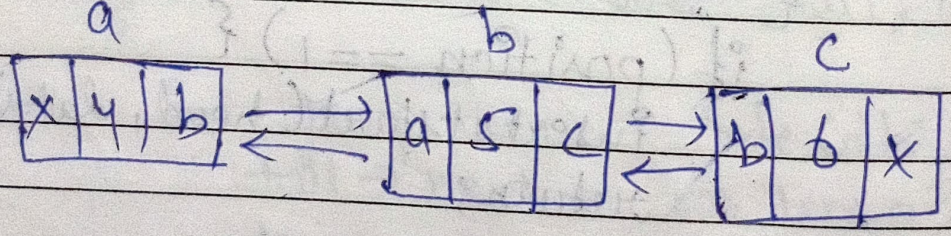
→ code

```
void insertatmid (node *& tail, node *& head, int position, int data)
    {
      if (position == 1) {
        insertatstart( head, data);
        return;
      }
      node * temp = head;
      int count = 1;
      while (count < position - 1)
      {
        temp = temp -> next;
        count ++;
      }
```

```
if (temp -> next == NULL){
    insertatend (tail, data);
    return;
}
node * new_node = new node(data);
new_node -> next = temp -> next;
new_node -> prev = temp;
temp -> next = new_node;
temp->next->prev = new_node;
}
```

Deletion in Doubly Linked List.

for deletion in doubly linked list let understand the logic first.

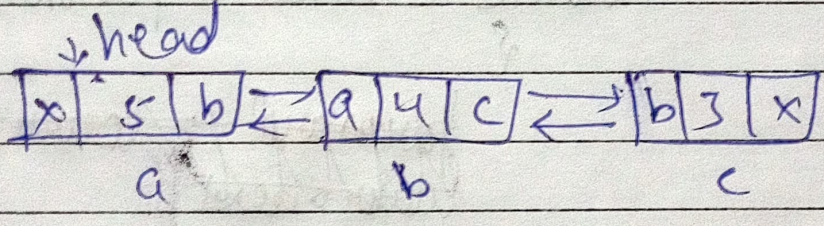


if we want to delete node b. what we have to do it.

1) the next of node a is equal to the node c
2) the prev of the node c is equal to the node a and delete the node b.

3) and before deletion you have to null the next and prev pointer of the deletion node, otherwise it give segmentation fault.

→ exception case
if it is first node in list then simply null the next of node, and also null the next node's prev pointer.
and delete the node



→ code

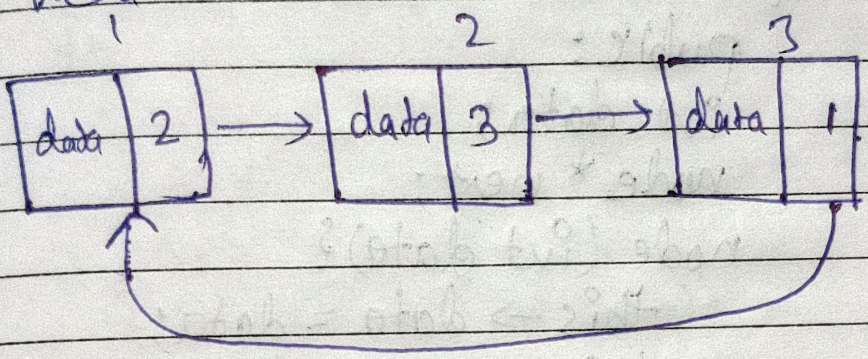
```
void deletion(node *&head, int position){
if(position == 1){
    node *temp = head;
    head->next->prev = NULL;
    head = head->next;
    head->next = NULL;
    delete
```

```
void deletion(node *&head, int position){
if(position == 1){
    node *temp = head;
```

```
temp -> next -> prev = NULL;
head = temp -> next;
temp -> next = NULL;
delete temp;
}
else {
node * prev = NULL;
node * curr = head;
~~node * forward = NULL;~~
int index = 1;

while (index < position)
{
~~forward = curr -> next;~~
~~curr -> next~~
prev = curr;
curr = curr -> next;
index += 1;
}
curr -> prev = NULL
prev -> next = curr -> next;
curr -> next -> prev = prev.
curr -> next = NULL
delete curr;
}
}
```

Circular link list.

Circular link list is list in which the last node's pointer pointing to the first node. and form a circular link list.



→ Insertion in CLL

we use a tail pointer which point to the last node always and its pointer point to the head. or first node

→ It has two cases.

1) where the tail is NULL which means the empty list.
In that case we create a node store data and represent it as tail. it next pointer pointing to itself. because it has none other node.

2) another case is non-empty list where new node store the data

; represent as tail and pointing to the first node.

⟹ code

create node

```
class node
{
public:
int data;
node * next;
node (int data){
    this -> data = data;
    this -> next = NULL;
}
```

insert in list

```
void insertion(node * tail , int element, int
data) {
if (tail == NULL)
    {
        node *new_node = new node (data);
        tail = new_node;
        new_node -> next = new_node;
    }
    else {
    node * temp = tail;
    while (temp -> next != element)
        {
            temp = temp -> next;
        }
    node * new_node = new node (data);
```

```
        new_node -> next = temp -> next;
        temp -> next = new_node;
    }
}
```

print list

```
void print (node *tail) {
    node * temp = tail;
    do
    {
        cout << tail -> data << " ";
        tail = tail -> next;
    } while (tail != temp);

    cout << endl;
}
```

main

```
int main() {
    node * tail = NULL;
    insertion( tail, 5, 10);
    insertion ( tail, 10, 22);
    insertion (tail, 22, 14);
    insertion ( tail, 14, 20);
    print (tail);
    return 0;
}
```

Output

10 22 14 20

→ deletion in CLL.

create two pointer previous and current. we just have to do is pointing the previous pointer to the current next pointer.
and delete that node.

code

delete node function.

```
~node(){
    int value = this -> data;
    if (this -> next != NULL){
        delete next;
        this -> next = NULL;
    }
}
```

deletion

```
void deletion (node * &tail, int
  value){
  if (tail == NULL){
     cout << "404 error" << endl;
  }
  else{
     node * prev = tail;
     node * curr = prev -> next;
  while(curr -> data != value)
     {  prev = curr;
        curr = curr -> next;
     }
```

```
prev -> next = curr -> next;

if (prev == curr)   // single node
{
    tail = NULL;
}

if (tail == curr)   // last node
{
    tail = prev;
}
curr -> next = NULL;
delete curr;
}
```

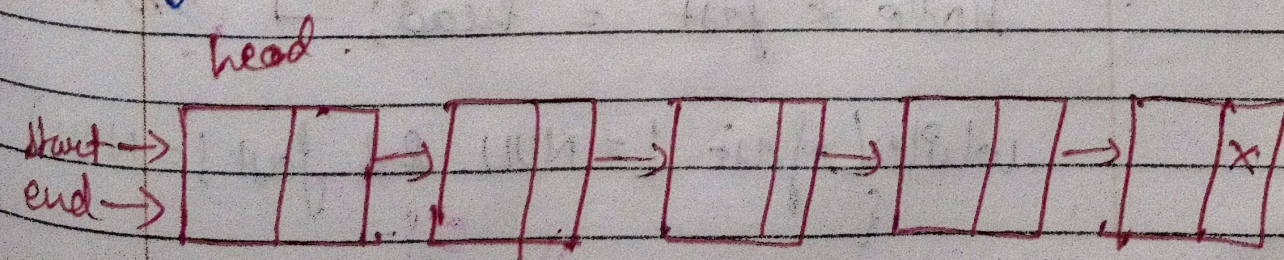
single node deletion.

last node deletion

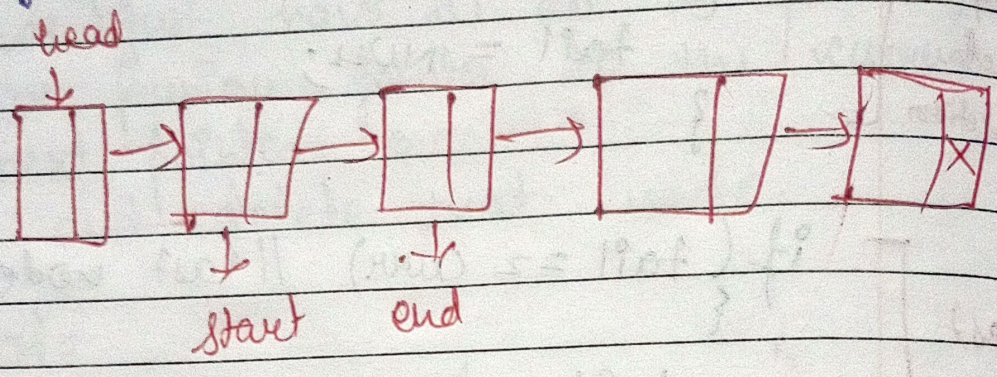
Floyd cycle Detection

This algorithm is specially for detecting the loop in a link list.

Where two pointers are use. and both move with different speed. means, the other pointer is move twice time faster than the first one.



now start move one step while end move twice or two step forward.



it traverse all the node in same time twice a time and if we find any loop or compare the node we can use this algorithm for that.

→ it use in many question.
1) find mid of a link list.
2) detect loop in link list. and so on.

⇒ example
find loop in link list.

```
bool isloop (Node * head){

    node * slow = head;
    node * fast = head;   ] — pointing to head.

while (fast != NULL && fast != NULL)
{
```