LINKED LIST

* Problem with Array Data structure.

① Either size is fixed & pre-allocated (in both fixed & variable sized arrays), OR the worst case insertion at the end is $O(n)$.

② Insertion in the middle (or beginning) is costly.

③ Deletion from the middle (or beginning) is ~~on~~ costly.

④ Implementation of data structure like queue & deque is compler with array.

* Difficult to implement using Array.
① Cannot used in Round Robin Scheduling. (Implemetation is costly)
② We cannot implement :- (means Implementation is costly) Given a sequence of items. Whenever we see an item x in the sequence, we need to replace it with 5 instances of another item y.

I/p:- d e a x q r x p y

o/p:- deayyyyyqryyyyypy

③ We have multiple sorted sequence & we need merge them frequently.

Sequence [ ] = { {5, 10, 15, 20},
{1, 12, 18},
{3, 30, 40}
{100, 200}
⋮

merge (0, 1)
merge (1, 2)

After this function call

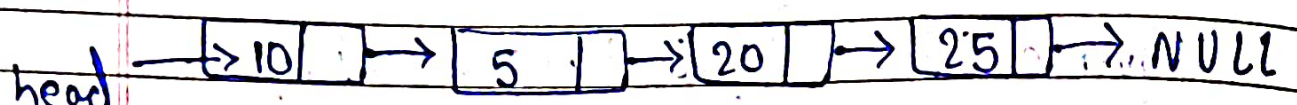
sequence[ ] = { {1, 5, 10, 12, 15, 18, 20}
{3, 30, 40}
{100, 200}

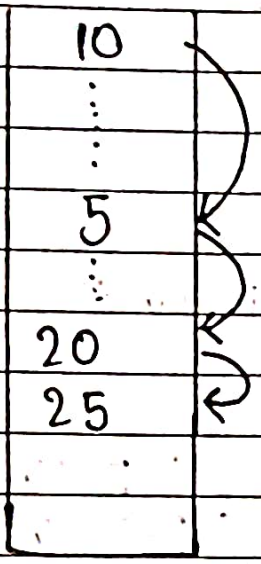
After this

Sequence[ ] = { {1, 5, 10, 12, 15, 18, 20}
{3, 30, 40, 100, 200}

④ In system level programming.

⊛ Introduction to linked list:-





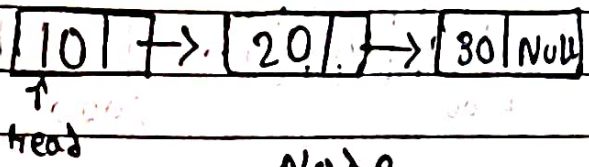
Memory Block

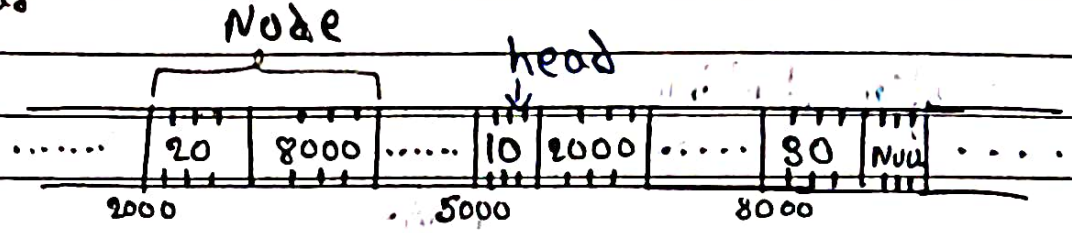
10 has memory address or pointer to 5 & 5 have address or pointer to 20 & 20 have address or pointer to 25 & 25 points to NULL.

Address are not paticular location it can be anywhere.

The idea is to drop the contiguous memory requirements so that insertions, deletion can efficiently happen at the middle also. And no need to pre-allocate the space (No extra nodes)

⊛ Simple linked list Implementation in C++:-





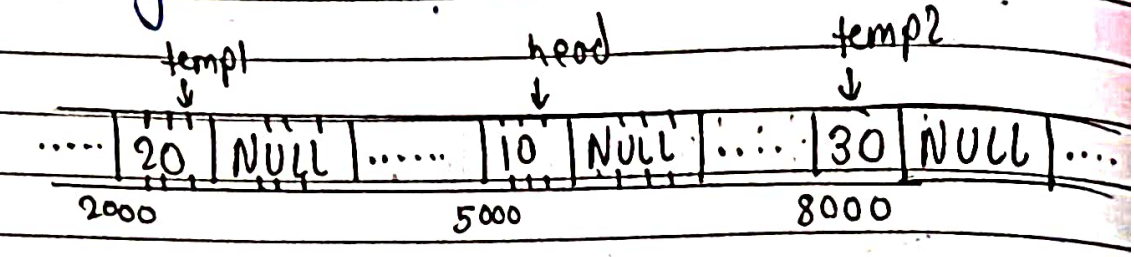
* Memory (Array of bytes) *

every node take 8 bytes in total. } Depended on compilers.
4 bytes for data.
4 bytes for Address.

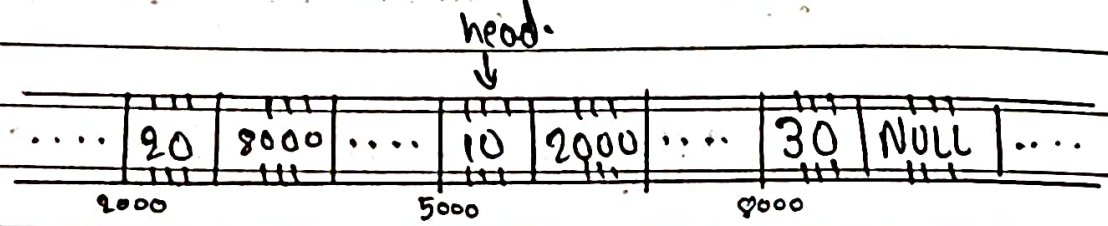
⇒ Example:-

```
struct Node {
int data;
struct *next;
 Node (int x) // Constructor.
 { data = x;
   next = NULL;
 } };

int main() {
Node *head = new Node(10); Node *temp1 = new Node(20);
Node *temp2 = new Node(30);
head -> next = temp1;
temp1 -> next = temp2;
 return 0; }
```

⇒ Initially :-



⇒ After linking :-



Alternate Implementation :-

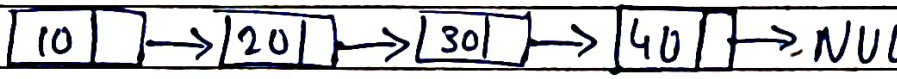
```
struct Node {
 int data;
 Node *next;
 Node (int x) {  data = x;  next = Null;}
 };

int main () {
 Node *head = new Node (10);
 head -> next = new Node (20);
 head -> next -> next = new Node (30);

 return 0;
 }
```

⑧ Application of linked list:-

1. Worst case insertion at the end & begin are O(1).
2. Worst case deletion from the beginning is O(1).
3. Insertion & deletion in middle are O(1) if we ~~y~~ ~~use~~ have reference to the previous node.
4. Round Robin Implementation.
5. Merging two sorted linked list is faster than arrays.
6. Implementation of simple memory manager where we need to link free blocks.
7. Faiser implementation of Queue & Deque data structure.

⑨ Traversing a Singly linked list in C++ :-

We have head pointer to this linked list:-

1) I/P:- 

O/P:- 10 20 30 40

2) I/P:- 

O/P:- 10

3) I/P :- NULL
O/P:-

⇒

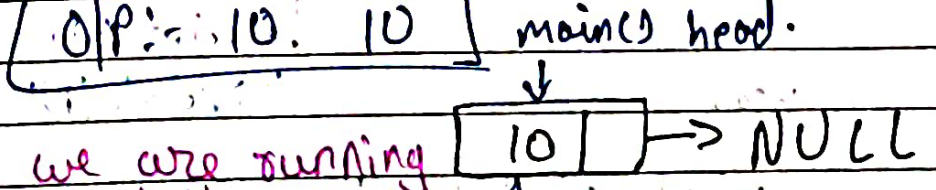
```
struct Node {
  int data;
  Node * next;
Node (int x) {
   data = x;
   next = NULL; }
};

void printList (Node *head)
{
  Node *curr = head;
   while (curr != NULL)
    {
     cout<< curr-> data << " ";
     curr = curr->next; }
}

int main () {
  Node *head; head= new Node(10);
  head->next = new Node(20);
   head->next->next = new Node(30);
  head->next->next->next = new Node(40);

  printList (head);
  return 0;
  }
```

O/P:- 10 20 30 40

⇒ function variable are also get copied when we make a function call.

```
Struct Node {
 int data;
 Node *next;
Node (int x)
 { data = x;
   next = NULL; } };

Void printList(Node *head) {
   while (head != NULL)
   { cout<< head -> data << " ";
       head = head -> next;
   }
}

int main() {
Node *head = new Node(10);
printList (head); // head get copied to printlist
printList (head); // function though both are
                  are different variable ones
return 0;         which is in main & one
}                 in function both have
                  some address.
```

O/P :- 10 10



When we are running the while loop we are changing the printlist head. (In function)

Printlist head

⊛ Recursive Display of linked list

1) I/P:- [10] → [20] → [30] → [40]

O/P:- 10 20 30 40

Time :- O(n) n is number of node.

⇒ Aux Space :- O(n)

```
void rPrint (Node *head) {
    if (head == NULL)
        {return ; }
    cout << (head -> ~~next~~ data) << " " ;

    rPrint ( head -> next );
}
```

rPrint (ref (10))

→ Print (10)

→ rPrint (ref (20))

→ Print (20)

→ rPrint (ref (30))

→ Print (30)

→ rPrint (ref (40))

→ Print (40)

→ rPrint (NULL)

head

[10] → [20] → [30] → [40]

⊛ Insert at Beginning of singly linked list.

1) I/P: 10 → 20 → 30 → NULL (head)

x = 5

O/P:- 5 → 10 → 20 → 30 → NULL (head)

2) I/P:- NULL

x = 5

O/P:- 5 → NULL.

⇒

```
struct Node {
int data;
Node *next;
Node(int x) { data = x; next = NULL; }
};

Node *insertBegin(Node *head, int x) {
Node *temp = new Node(x);
temp -> next = head;
return temp; }

int main() {
Node *head = NULL;
head = insertBegin(head, 30);
head = insertBegin(head, 20);
head = insertBegin(head, 10);
return 0; }
```

Time:-
O(1)

⊛ Insert at End of linked list :-

1) I/P :- [10] → [20] → [30]

x = 40

O/P :- [10] → [20] → [30] → [40]

2) I/P :- NULL

x = 10

O/P :- [10] → NULL

⇒ You need to change head if linked list is empty. (NULL)

```
struct Node {
 int data;
 Node * next;
 Node (int x) { data = x ; next = NULL; }
};

Node *insertEnd(Node *head, int x) {
  Node *temp = new Node(x);
  if (head == NULL)
        { return temp; }
  Node *curr = head;
  While (curr -> next != NULL)
        { curr = curr -> next; }
      curr -> next = temp;
      return head;
  }
```

```
int main() {
Node *head = NULL;
head = insertEnd (head, 10);
head = insert End (head, 20);
head = insert End (head, 30);
return 0; }
```

⊛ Insert at given position in singly linked list :-

1) I/P :- 10 → 30 → 50 → 70 , pos = 2, data = 20.
(1) (2) (3) (4)
O/P :- 10 → 20 → 30 → 50 → 70

2) I/P :- 5 → 15 → 25 → 35 , Pos = 5 , data = 10
(1) (2) (3) (4)
O/P :- 5 → 15 → 25 → 35 → 10

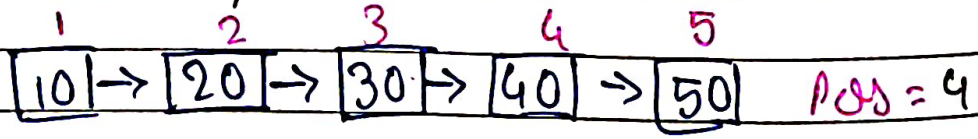
3) I/P :- 10 → 20 , pos = 4 , data = 5
O/P :- 10 → 20

Invalid position we can give positing (N+1) n is node.

Date
Page

⇒

```
Node insertPos (Node *head, int pos, int data)
{
   Node *temp = new Node (data);
    if (pass == 1)
     { temp->next = head;
        return temp; };
    Node *curr = head;
for (int i=1; i<pos-2 && curr != NULL; i++)
     { curr = curr->next; }

 if (curr == NULL)
          {return head;}

 temp -> next = curr -> next;
 curr -> next = temp;

 return head;
}
```

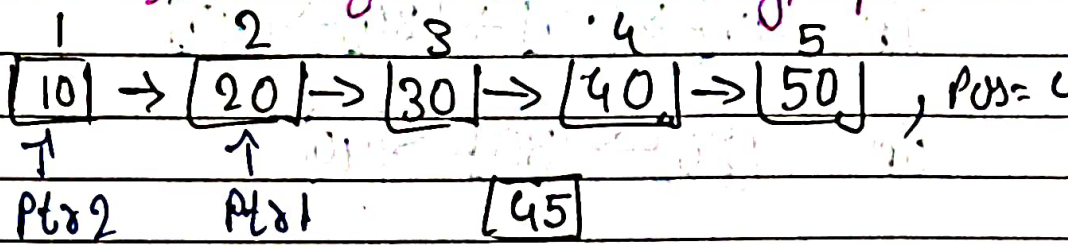


45

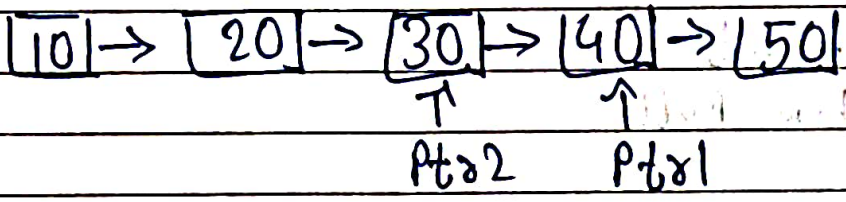
So 45 next should be alloted first

then 30 next to 45. If we allot 30 next to 45 first then we will lose the track of 40 & 50.

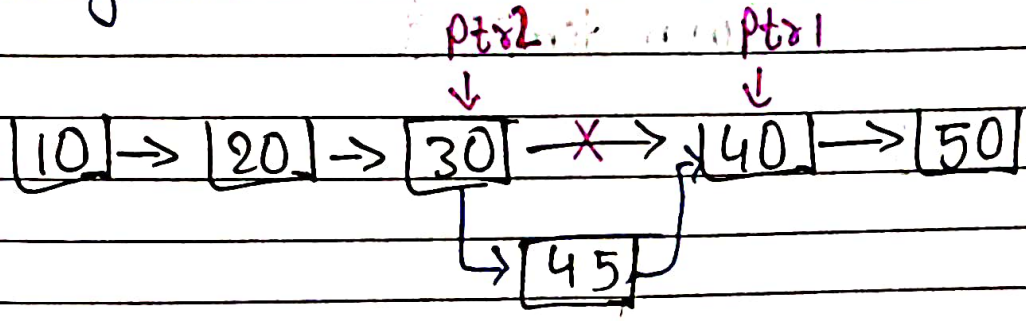
We can also use two pointer approach.
~~Ptr~~ initialization of two Node type poiters



two pointers Ptr1 & Ptr2 Both will go forward :-



Now if we make 45 next to Ptr1 or 30 next to 45 first we won't be losing rest.



⊛ Delete first Node in Singly linked list :-

I/P :- 10 → 20 → 30 → 40

O/P :- 20 → 30 → 40

I/P :- 10

O/P :- head = null.

I/P :- head = NULL

O/P :- head = NULL

⇒

```
Node *delHead (Node *head) {
    if( head == NULL)
        {return NULL; }
    else {
        Node *temp = head -> next;
        delete head;
        return temp; }
    }
```

O(1)

⊛ Delete last Node in Singly linked list :-

1) I/P :- 10 → 20 → 30 → 40
   O/P :- 10 → 20 → 30

2) I/P :- 10
   O/P :- head = NULL.

3) I/P :- head = NULL
   O/P :- head = NULL

⇒

```
Node *delLast (Node *head) {
  if (head == NULL)
    { return NULL; }
  if (head->next == NULL)
    { delete head;
      return NULL; }
  Node *curr = head;
  while (curr -> next -> next != NULL)
      { curr = curr -> next; }
  delete (curr -> next);
  curr -> next = NULL;
    return head; }
```

⊛ Search in a linked list:-

1) I/P:- [10] → [5] → [20] → [15]
x = 20
O/P:- 3

2) I/P:- [10] → [15] ; x = 20
O/P:- (-1)

3) I/P:- [3] → [20] → [5] , x = 3
O/P:- 1

⇒ Iterative Sol^n:-

```
int search (Node *head , int x) {
  int pos = 1;
  Node *curr = head ;
  while (curr != NULL)
   { if (curr -> data == x)
        return pos;
     else
        { pos++;
          curr = curr -> next; }
   }
   return -1;
}
```

Time:- O(n)

Space:- O(1) or O(n)

⇒ Recursive Soln:- |10| → |5| → |20| → |15| x=20

```
int search (Node *head, int x) {

if (head == NULL) return -1;
if (head -> data == x)                    [Time:- O(n)]
    return 1;                             [Space:- O(n)]
else
  { int res = search (head -> next, x);
    if (res == -1) {return -1;}
    else { return (res+1); }
  }                                       [Space:- O(n)]
}
```

3 → Search (ref(10), 20)

2 → search (ref(5), 20)

1 → Search (ref(20), 20).

If Array also the search time :- O(n) (unsorted)

In Sorted Array we use Binary array But in Linked list we cannot implement Binary search in O(log n). we cannot find middle element in O(1) time.
But there are variation of linked list which increase the search in sorted linked list.
& that are know as

* SKIP list *