Linked List

Definition

→ An order collection od homogenous data item

→ Where elements can be added anywhere and removed from anywhere

Catagory	Array	Linked List
Data structure	Contiguous block od memory	Nodes with Pointers
Dynamic size	Fixed size unless manually resized	Easily grows or shinks
Insertion/ Deletions	Inefficient for frequent insertions /removals.	Efficient for insertions/rem-ovals.
Memory Allocation	May lead to fragmentation	Reduces fragmentation.
Use Case	Fast access with known size	No fixed size Frequent insortion/rem-

Types

1) Single linked List

→ Can be traversed only in one direction

→ Sorted LL

- Element are inserted in sorted order
- Deletion as per requirements.

→ Unsorted LL

- Elements are inserted & deleted as per user requirements.

→ Implementation

SNT

1) Algo LLType CreateLinkedList( )
{
CreateNode(head);
head = NULL;
}

2) LLType Insert(LLType head, NodeType NewNode)
→ { //Unsorted (int key)
if(head == NULL)
NewNode → Next = NULL;
head = NewNode;
Else {
Case 1: insertion before head
NewNode → next = head;
head = NewNode;
Case 2: insertion after the last
Struct NodeType *temp;
temp = head
While(temp → next != NULL)
temp = temp → next;
temp → next = NewNode;
Case 3: insertion before some node(Key)
Struct NodeType* temp;
temp = head; Prew = NULL;
While (temp → data != Key && temp != NULL)
Prev = temp; temp = temp → next;

if (temp == NULL) " Key not Found"
Else
NewNode -> next = temp;
prev -> next = temp;
Case4: Insertion after some node (Key)
Struct Node Type *temp
~~temp = Head; Prev = Null~~;
while (temp -> data != Key && ~~temp != Null~~
temp != NULL)
temp = temp -> next;
if (temp == NULL) "Key not Found"
else
NewNode -> next = temp -> next;
temp -> next = NewNode;
}
→ // sorted
{
if (LL = NULL) // first element
LL = NewNode; exit(0);
temp = LL, Prev = NULL;
While ((NewNode -> data > temp -> data) &&
temp != NULL)
{ prev = temp;
temp = temp -> next;
}
if (temp == NULL) // end
Prev -> next = New Node;
if (temp = = LL && NewNode -> data < temp -> data)

{NewNode -> next = temp;
LL = NewNode;
}
else
NewNode -> next = temp;
prev -> next = NewNode;
}

3) Algo ET Delete (LLType Head, ET ele, int key)
{
if (Head == NULL)
Print "Underflow";
exit;
else
{
1) Search for element
2) does not exist
3) in Unsorted List
4) in sorted List
}
}

What to do.

Deletion

{
if (Head == NULL)
{ Print "Underflow"; exit(); }
else { // search
Temp = Head, Prev = NULL;
while (Temp -> data < key)
prev = Temp; Temp = Temp -> Next;

```
if (Temp -> data != key) // not found
{ "Error"; Exit(0); }
elseif (temp -> data == Key && prev == NULL)
Head = Head -> next;    // first element
else if (temp -> data == key && temp -> next
        == NULL) // only element in LL
  Head = NULL
else
prev -> next = Temp -> next
 }
return Temp;
}
```

2) Circular Linked List
→ Last node is connected to first node
→ It supports traversing from the end of the list to the beginning by making the last node point back to the head of the list
→ Implementation
Refer (1)

2) CLL Type Insert (CLL Type head, Node Type NewN
{
if (Head == NULL) // top
Head = NewNode;
Head → next = Head;
else
NewNode → Next = Head → next;
Head → next = value;
}
// middle
{
Create Node (temp) = Head;
while (temp → element != Key && current→N
    != Head)
{
Current = Current → next;
if (Current → element == Key) {
NewNode → next = Current → next
Current → next = NewNode;
}

//end
{
createNode (temp) = Head;
while (temp -> next != head) {
current = current -> next; }
current -> next = NewNode;
NewNode -> next = head;
}

3) Algo ET Delete (CLLType Head, ET ele, int key)
{ //top
if (head == NULL)
"underflow"
if (head -> next == head)
temp = head;
Head = NULL;
else
Temp = head -> next;
head -> next = temp -> next;
return (temp -> data)
}
~~//End~~
{ //END
Temp = head; Prev = NULL;
while (temp -> next != head) {
Prev = Temp;
Temp = Temp -> next;
}
~~if (Prev == NULL)~~

```
{ //middle
Temp = head; Prev = NULL;
while (Temp->ele != key && temp->next != head)
{ prev = temp;
  temp = temp->next;
}
                    data
if (Temp-> element == key) {
Prev->next = current->next;
return (temp->data)
}
```

3) Doubly Linked List

→ Can be traversed in both ~~questio~~ direction

→ Implementation

```
struct Node {
  int data;
  Node* next;
  Node* prev;
 };
Node* head, List1, List2;
```

1) ~~Search~~

```
temp = Head;
While (NewNode->data > temp->data &&
          temp-> next != NULL)
temp = temp->next;
```

2) //Insertion

```
Create Node (New Node)
```

A) First Node

```
if (Head == NULL)
Head = NewNode;
```

B) At end

```
if (NewNode->data > temp->data && temp->next
      == NULL)
{ NewNode -> prev = temp;
  temp -> next = NewNode;
}
```

C) Before head Node

```
if (NewNode->data < temp->data && temp == head)
{ NewNode -> next = temp;
```

temp -> prev = NewNode;
head = NewNode;
}

D) inbetween
{
NewNode -> next = temp;
NewNode -> Prev = temp -> prev;
temp -> prev -> next = NewNode;
temp -> prev = NewNode;
}

3) Deletion

1) if (Head == NULL)
"Underflow"
// search
temp = head
while(temp != NULL && temp -> data < key)
temp = temp -> next;
// unsuccessful search
if (temp -> data > key || (temp -> next == NULL
&& temp -> data < key))
"Error"

2) Only Node.
if (temp -> data == key && temp == head && temp
-> next == NULL)
{ Head = NULL;
return (temp) }

3) Last Node
if (temp -> data == key && temp -> next == NULL)

{ temp → prev → next = NULL
return(temp)
}

4) <u>General</u>
{ temp → next → prev = temp → prev;
temp → prev → next = temp → next;
}

5) <u>First node</u>
if (temp == head && temp → data == key)
{ head = head → next;
temp → next → prev = NULL;
return (temp)
}

Advantage

→ Overflow can never occur unless memory actully full.

→ Insertions & deletions are easier than for contiguous (array) lists.

→ With large records, moving pointers is easier and faster than moving the items themselves

Disadvatage

→ Pointers require extra space

→ Linked Lists do not allow random access

→ Time must be spent traversing & changing the pointers

→ Programming is trickier with pointer