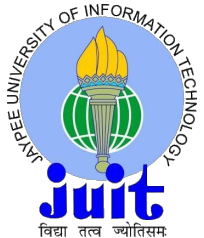


Data Structure

(Course Code: 25B11CI318)

Linked List

Mr. Sandeep Kumar Patel

Assistant Professor

Jaypee University of Information Technology

Waknaghat, Solan, HP-173234

Need for Linked Lists

Instead of storing elements in contiguous memory,
we store each element in a separate memory location.

Each element stores

- Data
- Address of the next node

Need for Linked Lists

Instead of storing elements in contiguous memory,
we store each element in a separate memory location.

Each element stores

- Data
- Address of the next node

Advantages

- Dynamic Size
- Easy Insertion
- Easy Deletion
- Better Memory Utilization

What is a Linked List?

Definition

A linked list is a linear data structure in which each element is called a **node**.
Each node contains

1. Data
2. Address of the next node

The nodes are connected using pointers.

What is a Linked List?

Definition

A linked list is a linear data structure in which each element is called a **node**.
Each node contains

1. Data
2. Address of the next node

The nodes are connected using pointers.

$$Node = (Data,\ Next)$$

Structure of a Node

Node Representation

C Structure

```c
struct Node
{
    int data;
    struct Node *next;
};
```

The pointer stores the address of the next node.

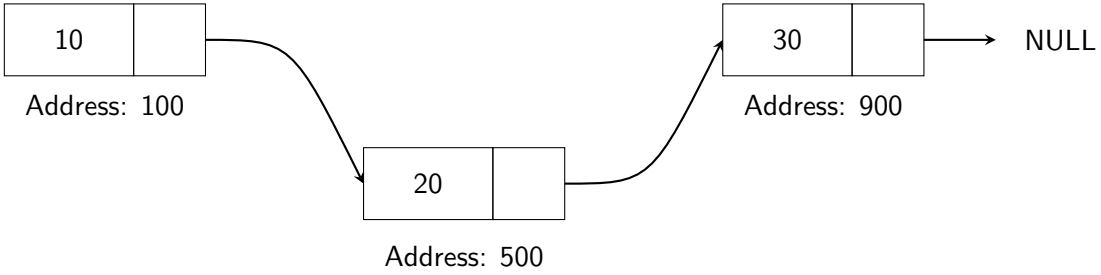
Representation of a Linked List

Representation of a Linked List

Nodes are connected using pointers.
The last node points to **NULL**.

Memory Representation of a Linked List

Unlike arrays, linked list nodes are stored at **different memory locations**.

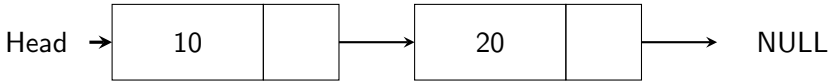


Observation

Nodes need not be stored in contiguous memory.

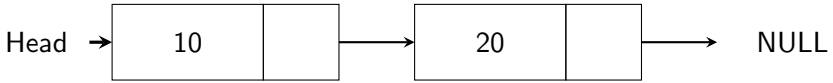
Head Pointer

The **Head Pointer** stores the address of the first node in the linked list.



Head Pointer

The **Head Pointer** stores the address of the first node in the linked list.



Important

If

$$Head = NULL$$

then the linked list is empty.

Dynamic Memory Allocation

Nodes are created during program execution.

```
#include <stdlib.h>

struct Node *newNode;

newNode =
(struct Node*)
malloc(sizeof(struct Node));
```

Dynamic Memory Allocation

Nodes are created during program execution.

```
#include <stdlib.h>

struct Node *newNode;

newNode =
(struct Node*)
malloc(sizeof(struct Node));
```

Memory is allocated from the

Heap Memory

Dynamic Memory Allocation

Nodes are created during program execution.

```
#include <stdlib.h>

struct Node *newNode;

newNode =
(struct Node*)
malloc(sizeof(struct Node));
```

Memory is allocated from the

Heap Memory

Advantages

- Memory allocated only when required.
- No fixed size limitation.

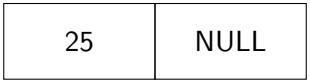
Creating a New Node

```
struct Node *newNode;

newNode =
(struct Node*)
malloc(sizeof(struct Node));

newNode->data = 25;

newNode->next = NULL;
```

Creating a New Node

```
struct Node *newNode;

newNode =
(struct Node*)
malloc(sizeof(struct Node));

newNode->data = 25;

newNode->next = NULL;
```

Result



Traversal of a Linked List

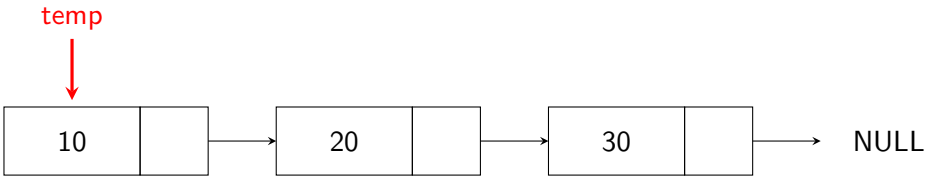
Algorithm

```
temp = head;

while(temp != NULL)
{
   printf("%d ",temp->data);

   temp = temp->next;
}
```

Traversal of a Linked List

Algorithm

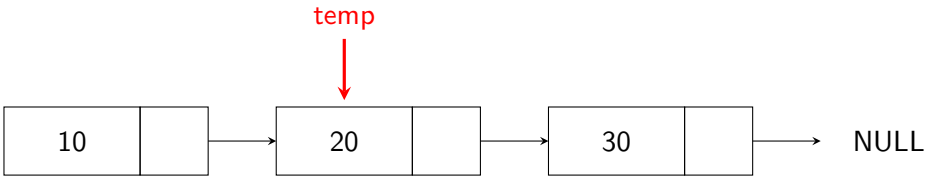
```
temp = head;

while(temp != NULL)
{
    printf("%d ",temp->data);

    temp = temp->next;
}
```



Traversal of a Linked List

Algorithm

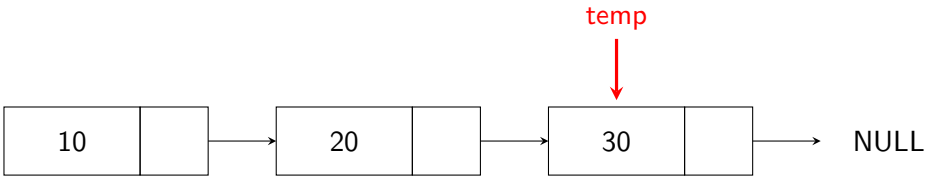
```
temp = head;

while(temp != NULL)
{
    printf("%d ",temp->data);

    temp = temp->next;
}
```



Traversal of a Linked List

Algorithm

```
temp = head;

while(temp != NULL)
{
    printf("%d ",temp->data);

    temp = temp->next;
}
```



Accessing Nodes

Unlike arrays,

No Direct Indexing

Accessing Nodes

Unlike arrays,

No Direct Indexing

To access the 4th node,we must visit

$$1^{st} \rightarrow 2^{nd} \rightarrow 3^{rd} \rightarrow 4^{th}$$

Accessing Nodes

Unlike arrays,

No Direct Indexing

To access the 4th node,we must visit

$$1^{st} \rightarrow 2^{nd} \rightarrow 3^{rd} \rightarrow 4^{th}$$

Operation	Complexity
Access	O(n)
Traversal	O(n)
Search	O(n)

Key Point

Linked Lists provide sequential access, not random access.

Insertion at the Beginning

Insert node containing **5** at the beginning.

Steps

Insertion at the Beginning

Insert node containing **5** at the beginning.

Steps

1. Create a new node.

Insertion at the Beginning

Insert node containing **5** at the beginning.

Steps

1. Create a new node.
2. Store the value in the new node.

Insertion at the Beginning

Insert node containing **5** at the beginning.

Steps

1. Create a new node.
2. Store the value in the new node.
3. New node points to the current `head`.

Insertion at the Beginning

Insert node containing **5** at the beginning.

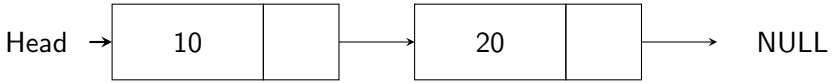
Steps

1. Create a new node.
2. Store the value in the new node.
3. New node points to the current `head`.
4. Make the new node the new `head`.

$$\boxed{\text{Time Complexity } = O(1)}$$

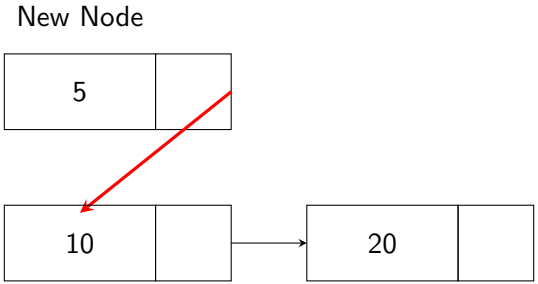
Insertion at the Beginning

Goal: Insert **5** before the first node.



Insertion at the Beginning

Goal: Insert **5** before the first node.



Time Complexity $= O(1)$

Insertion at the Beginning

Goal: Insert **5** before the first node.

Time Complexity $= O(1)$

Insertion at Beginning

```
newNode = malloc(sizeof(struct Node));

newNode->data = value;

newNode->next = head;

head = newNode;
```

Insertion at Beginning

```
newNode = malloc(sizeof(struct Node));

newNode->data = value;

newNode->next = head;

head = newNode;
```

Time Complexity $= O(1)$

Insertion at the End

Insert node containing **30**.

Steps

Insertion at the End

Insert node containing **30**.

Steps

1. Traverse to the last node.

Insertion at the End

Insert node containing **30**.

Steps

1. Traverse to the last node.
2. Create a new node.

Insertion at the End

Insert node containing **30**.

Steps

1. Traverse to the last node.
2. Create a new node.
3. Last node's next points to new node.

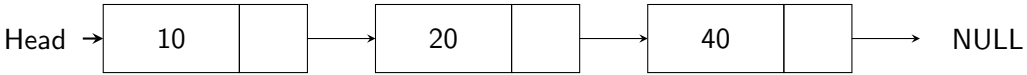
Insertion at the End

Insert node containing **30**.

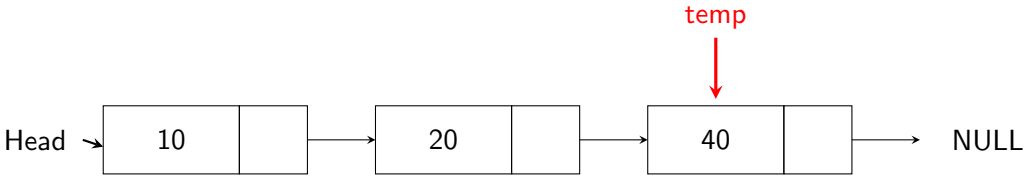
Steps

1. Traverse to the last node.
2. Create a new node.
3. Last node's next points to new node.
4. New node points to NULL.

Insertion at the End



Insertion at the End



Time Complexity $= O(n)$

Insertion at the End

Time Complexity $= O(n)$

Insertion at End

```
newNode = malloc(sizeof(struct Node));

newNode->data = value;
newNode->next = NULL;

temp = head;

while(temp->next != NULL)
{
   temp = temp->next;
}

temp->next = newNode;
```

Insertion at End

```
newNode = malloc(sizeof(struct Node));

newNode->data = value;
newNode->next = NULL;

temp = head;

while(temp->next != NULL)
{
   temp = temp->next;
}

temp->next = newNode;
```

Time Complexity $= O(n)$

Insertion After a Given Node

Insert node containing **25** after node 20.

Steps

Insertion After a Given Node

Insert node containing **25** after node 20.

Steps

1. Traverse to the node containing 20.

Insertion After a Given Node

Insert node containing **25** after node 20.

Steps

1. Traverse to the node containing 20.
2. Create a new node.

Insertion After a Given Node

Insert node containing **25** after node 20.

Steps

1. Traverse to the node containing 20.
2. Create a new node.
3. New node points to the next node of 20.

Insertion After a Given Node

Insert node containing **25** after node 20.

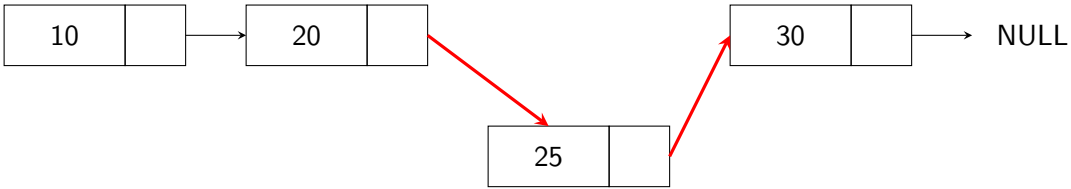
Steps

1. Traverse to the node containing 20.
2. Create a new node.
3. New node points to the next node of 20.
4. Node 20 points to the new node.

$$\boxed{\text{Time Complexity } = O(n)}$$

Insertion After a Given Node

Insert **25** after node containing 20.



Insertion After a Given Node

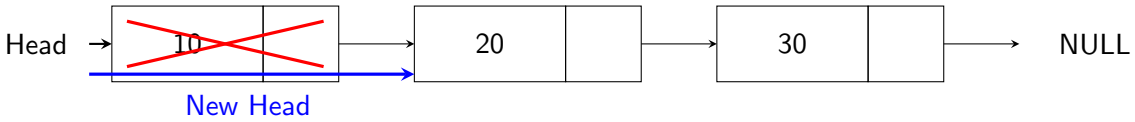
```
newNode = malloc(sizeof(struct Node));

newNode->data = value;

temp = head;

while(temp != NULL && temp->data != key)
{
   temp = temp->next;
}

if(temp != NULL)
{
   newNode->next = temp->next;
   temp->next = newNode;
}
```

Insertion After a Given Node

```
newNode = malloc(sizeof(struct Node));

newNode->data = value;

temp = head;

while(temp != NULL && temp->data != key)
{
   temp = temp->next;
}

if(temp != NULL)
{
   newNode->next = temp->next;
   temp->next = newNode;
}
```

Deletion from the Beginning

Delete the first node containing **10**.



Operation:

$$\text{Head} = \text{Head} \rightarrow \text{next}$$

The first node is removed and **20** becomes the new head.

Deletion from the Beginning

Delete the first node containing **10**.

Steps

Deletion from the Beginning

Delete the first node containing **10**.

Steps

1. Check whether the list is empty.

Deletion from the Beginning

Delete the first node containing **10**.

Steps

1. Check whether the list is empty.
2. Store the first node in a temporary pointer.

Deletion from the Beginning

Delete the first node containing **10**.

Steps

1. Check whether the list is empty.
2. Store the first node in a temporary pointer.
3. Move `head` to the next node.

Deletion from the Beginning

Delete the first node containing **10**.

Steps

1. Check whether the list is empty.
2. Store the first node in a temporary pointer.
3. Move `head` to the next node.
4. Free the temporary node.

$$\boxed{\text{Time Complexity } = O(1)}$$

Deletion from Beginning

```
temp = head;

if(head != NULL)
{
    head = head->next;
    free(temp);
}
```

Deletion from Beginning

```
temp = head;

if(head != NULL)
{
    head = head->next;
    free(temp);
}
```

Time Complexity $= O(1)$

Deletion from the End

Delete the last node containing **30**.

Operation:

Traverse to the last node's predecessor and set:

$$\text{temp} \rightarrow \text{next} = \text{NULL}$$

The last node is removed.

Deletion from the End

Delete the last node containing **30**.

Steps

Deletion from the End

Delete the last node containing **30**.

Steps

1. Check whether the list is empty.

Deletion from the End

Delete the last node containing **30**.

Steps

1. Check whether the list is empty.
2. Traverse to the second-last node.

Deletion from the End

Delete the last node containing **30**.

Steps

1. Check whether the list is empty.
2. Traverse to the second-last node.
3. Store the last node in a temporary pointer.

Deletion from the End

Delete the last node containing **30**.

Steps

1. Check whether the list is empty.
2. Traverse to the second-last node.
3. Store the last node in a temporary pointer.
4. Set the second-last node's next to `NULL`.

Deletion from the End

Delete the last node containing **30**.

Steps

1. Check whether the list is empty.
2. Traverse to the second-last node.
3. Store the last node in a temporary pointer.
4. Set the second-last node's next to `NULL`.
5. Free the last node.

$$\boxed{\text{Time Complexity } = O(n)}$$

Deletion from End

```
temp = head;

while(temp->next->next != NULL)
{
   temp = temp->next;
}

free(temp->next);
temp->next = NULL;
```

Deletion from End

```
temp = head;

while(temp->next->next != NULL)
{
    temp = temp->next;
}

free(temp->next);
temp->next = NULL;
```

Time Complexity $= O(n)$

Deletion After a Given Node

Delete the node containing **25** after node 20.

Steps

Deletion After a Given Node

Delete the node containing **25** after node 20.

Steps

1. Traverse to the given node.

Deletion After a Given Node

Delete the node containing **25** after node 20.

Steps

1. Traverse to the given node.
2. Store the node to be deleted in a temporary pointer.

Deletion After a Given Node

Delete the node containing **25** after node 20.

Steps

1. Traverse to the given node.
2. Store the node to be deleted in a temporary pointer.
3. Change the given node's next pointer.

Deletion After a Given Node

Delete the node containing **25** after node 20.

Steps

1. Traverse to the given node.
2. Store the node to be deleted in a temporary pointer.
3. Change the given node's next pointer.
4. Free the deleted node.

$$\boxed{\text{Time Complexity } = O(n)}$$

Deletion of a Given Node

```
temp = head;

while(temp->next != NULL &&
      temp->next->data != key)
{
   temp = temp->next;
}

del = temp->next;

temp->next = del->next;

free(del);
```

Deletion of a Given Node

```
temp = head;

while(temp->next != NULL &&
      temp->next->data != key)
{
   temp = temp->next;
}

del = temp->next;

temp->next = del->next;

free(del);
```

Time Complexity $= O(n)$

Finding the Middle Node

Use two pointers: **slow** and **fast**.

Steps

Finding the Middle Node

Use two pointers: **slow** and **fast**.

Steps

1. Initialize both pointers at `head`.

Finding the Middle Node

Use two pointers: **slow** and **fast**.

Steps

1. Initialize both pointers at `head`.
2. Move `slow` one node at a time.

Finding the Middle Node

Use two pointers: **slow** and **fast**.

Steps

1. Initialize both pointers at `head`.
2. Move `slow` one node at a time.
3. Move `fast` two nodes at a time.

Finding the Middle Node

Use two pointers: **slow** and **fast**.

Steps

1. Initialize both pointers at `head`.
2. Move `slow` one node at a time.
3. Move `fast` two nodes at a time.
4. When `fast` reaches the end, `slow` is at the middle.

Slow Pointer: 1 step

Fast Pointer: 2 steps

Finding the Middle Node

```
slow = head;
fast = head;

while(fast != NULL && fast->next != NULL)
{
    slow = slow->next;
    fast = fast->next->next;
}

printf("Middle = %d", slow->data);
```

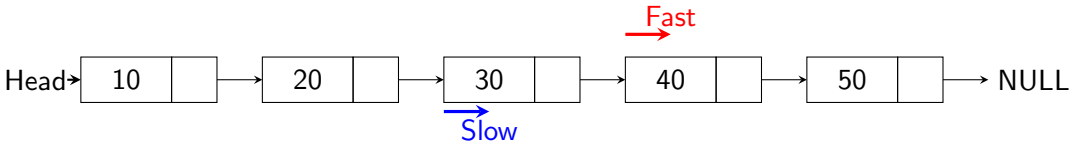
Finding the Middle Node

```
slow = head;
fast = head;

while(fast != NULL && fast->next != NULL)
{
    slow = slow->next;
    fast = fast->next->next;
}

printf("Middle = %d", slow->data);
```

Time Complexity $= O(n)$

Space Complexity $= O(1)$

Finding the Middle Node



`slow` $\rightarrow 30$ `fast` $\rightarrow 40$

Thank You

Questions?