Linked Lists

ALGORITHMS & DATA STRUCTURES – I

COMP 221

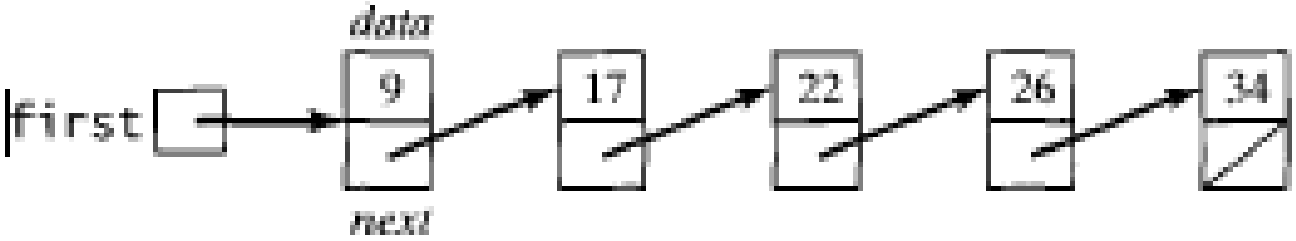
A **linked list** is a sequence of elements called nodes, each of which has two parts:

1. A ***data*** *part that stores an element of the list*
2. A ***next*** *part that stores a link (or pointer) that indicates the location of the* node containing the next list element.

If there is no next element, then a special **null** value is used.

Also, the location of the node storing the first list element must be maintained. This will be the null value, if the list is empty.

- To illustrate, a linked list storing the integers 9, 17, 22, 26, 34 might be pictured as



- first points to the first node in the list. The *data part of each node* stores one of the integers in the list and
- the arrow in the *next part represents a link to* the next node.
- The diagonal line in the *next part of the last node represents a null link* and indicates that this list element has no successor.

Basic List Operations:

1. Constructing an empty list
2. Checking whether list is empty or not
3. Traversing the list
4. Inserting a new element in the list
5. Deleting an element from the list.

1. **Constructor:**

   To construct an empty list, we simply make first a null link to indicate that it does not refer to any node:

   first = *null-value;*

2. **IsEmpty:**

   Determining whether a list is empty is simply checking whether first is null:

   first == *null-value*

3. Traverse:

- To traverse a linked list , we begin by initializing an auxiliary variable ptr to point to the first node:

Step 1 : Initialize ptr to first.

Step 2 : Process the data part of the node referred to by ptr.

Step 3 : ptr= next part of the node currently referred to by ptr.

Repeat from step2 until ptr becomes null.

- In summary, a linked list can be traversed as follows:

```
ptr = first;
while (ptr != null_value)
{
   Process data part of node referred to by ptr
   ptr = next part of node referred to by ptr
}
```

- To display the list elements, simply output the data part of the node in the processing step.

4. Insertion:

- To insert a new data value into a linked list, we must first obtain a new node and store the value in its data part.
- The second step is to connect this new node to the existing list, and for this, there are two cases to consider:

  (1) insertion after some node in the list and

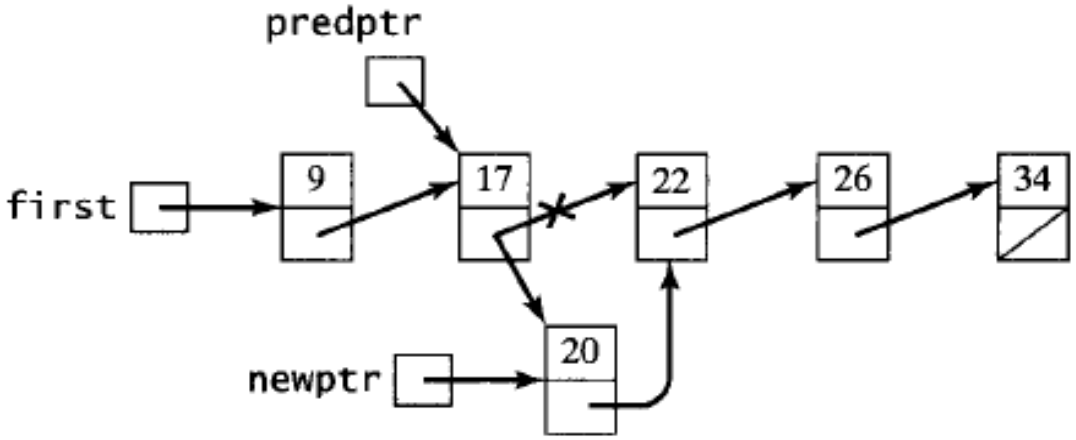
  (2) insertion at the beginning of the list.

Case (1):

Get a new node pointed to by ***newptr*** and store the item to be inserted in its data part.

And insert this node into the list after some node pointed to by **predptr** as follows:

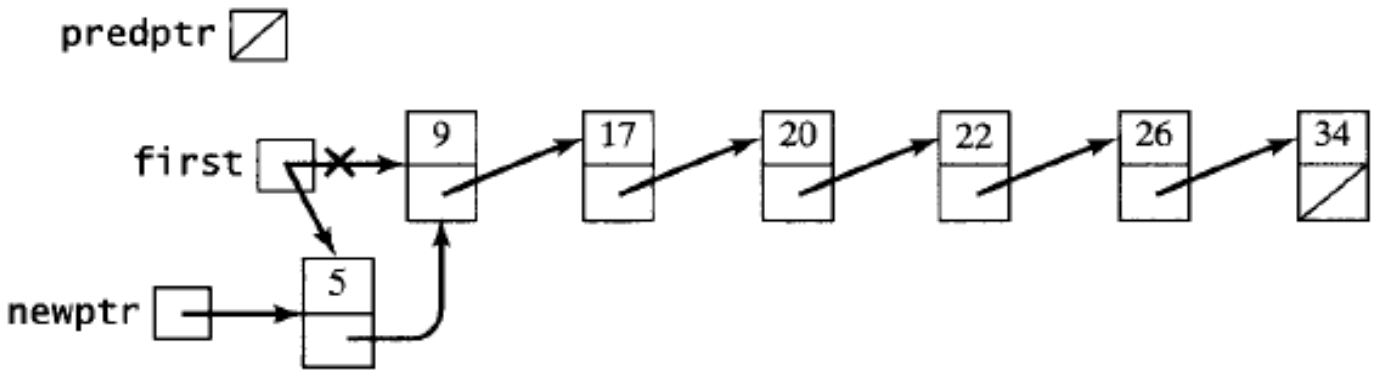
```
newptr->next = predptr->next;
predptr->next = newptr;
```



Case (2) :

There is no predecessor for the item being inserted and we need to change the pointer to the first node so that it now points to this new first node in the list:

```
newptr->next = first->next;
first = newptr;
```



5. Deletion:

- For deletion, there are also two cases to consider:

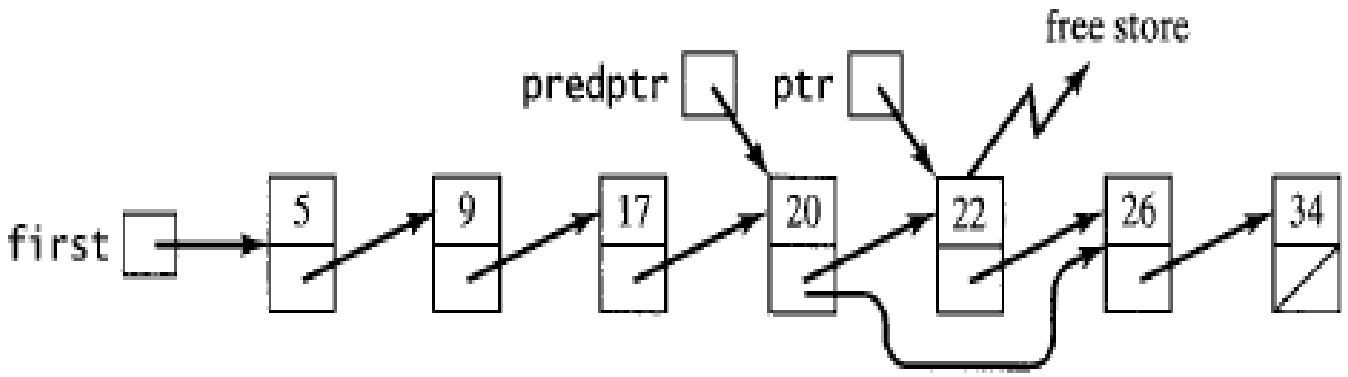
  (1) deleting an element that has a predecessor and

  (2) deleting the first element in the list.

Case (1):

In case 1, we bypass the node by changing the next link from its predecessor and then deallocating the deleted node:

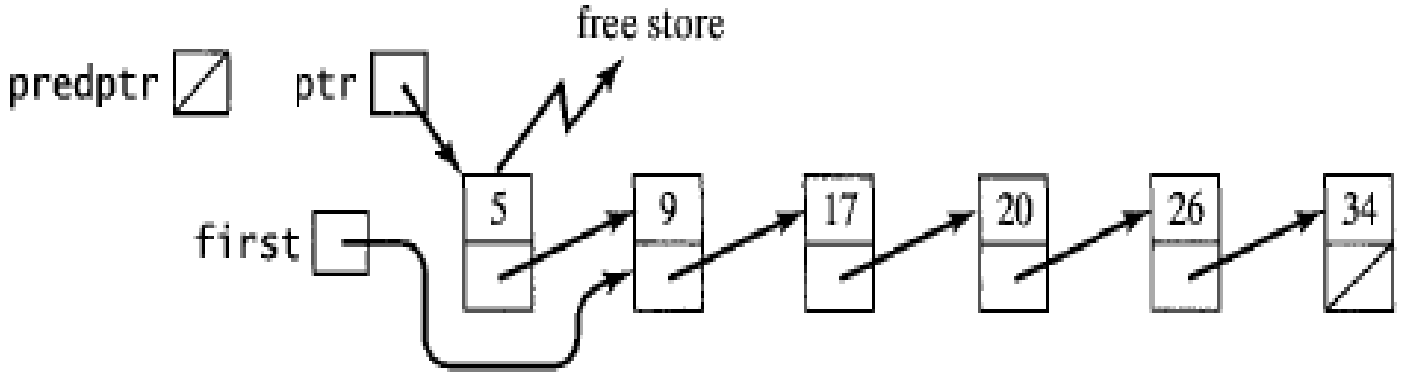
```
predptr->next = ptr->next;
delete ptr;
```



Case (2):

In case 2, there is no predecessor, so we bypass the node by changing the node pointed to by first:

```
first = ptr->next;
delete ptr;
```



Singly Linked List:

Linked lists that are processed sequentially in the order in which the nodes are linked together are referred to as **singly linked lists**.

Some Variants of Singly Linked list:

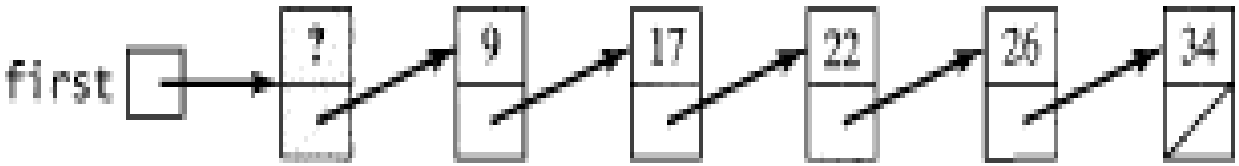
1. Linked Lists with Head Nodes
2. Circular Linked Lists

1. Linked Lists with Head Nodes:

A linked list with a dummy first node at the beginning , called a **head node**, is a linked list with head nodes.

No actual list element is stored in the data part of this head node; instead, it serves as a predecessor of the node that stores the actual first element, because its link field points to this "real" first node.

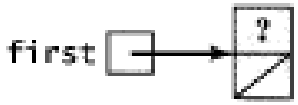
For example, the list of integers 9, 17, 22, 26, 34 can be stored in a linked list with a head node as follows:



Operations on Linked list with Head nodes:

Constructor:

An empty list will have a head node that is pointed to by *first* and has a null link.



IsEmpty:

Determining whether a list is empty is checking

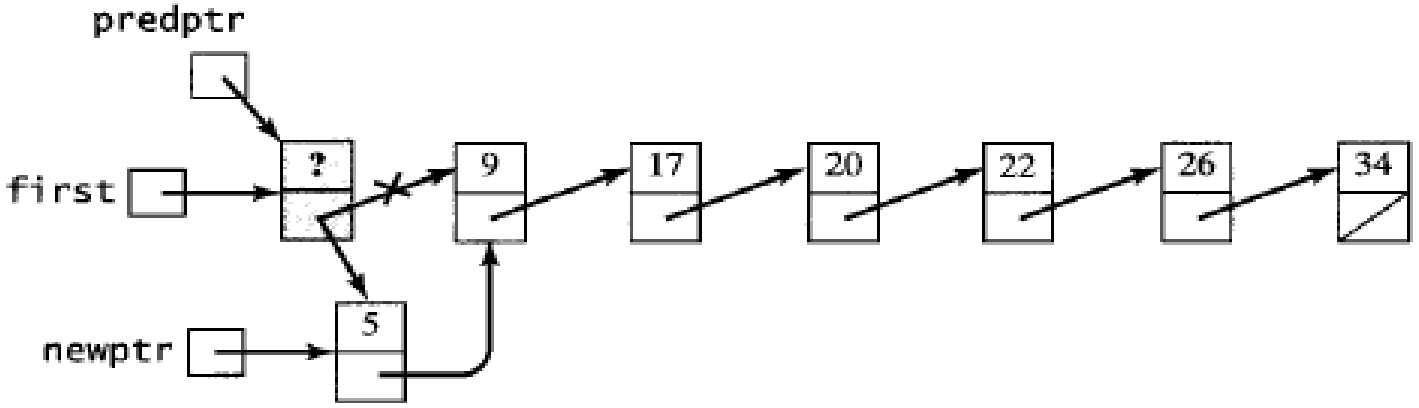
first->next == 0 rather than first == 0.

Operations on Linked list with Head nodes(contd.)

Insertion:

If the list has a head node, then inserting at the beginning of the list is not a special case because the head node serves as a predecessor for the new node:

```
newptr->next = predptr->next;
predptr->next = newptr;
```

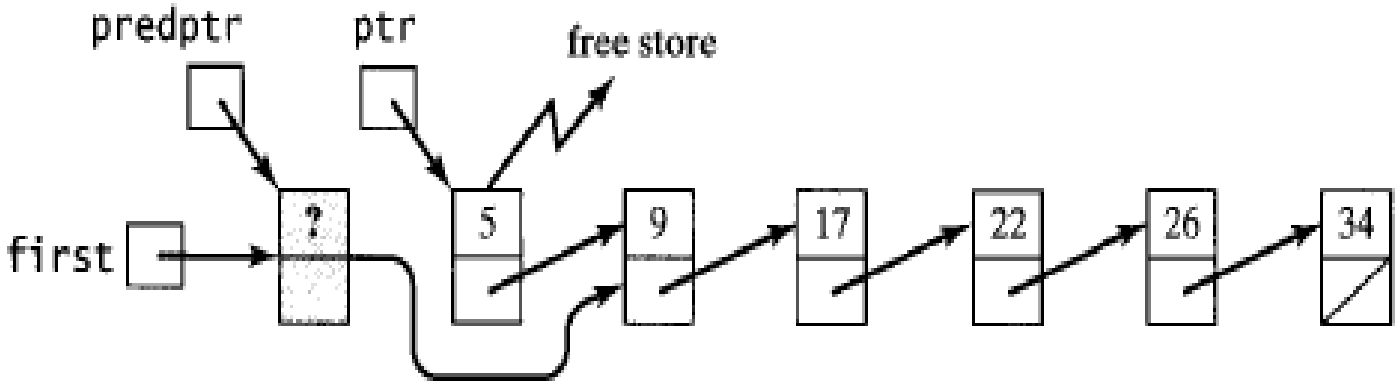


Operations on Linked list with Head nodes (contd.)

Deletion:

If the list has a head node, then deleting an element at the beginning of the list is not a special case because the head node serves as a predecessor for the new node:

```
predptr->next = ptr->next;
delete ptr;
```

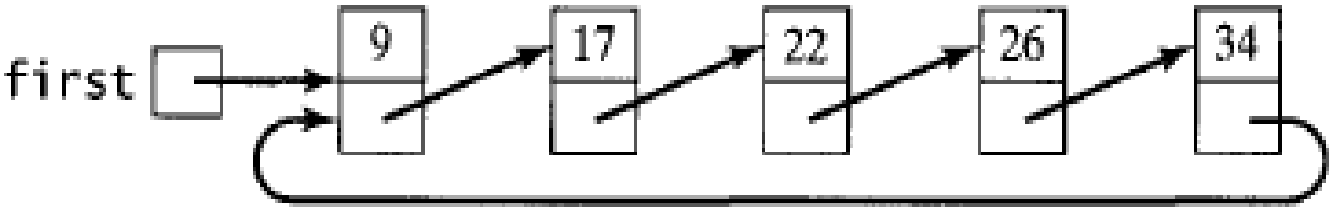


2. Circular Linked Lists:

A ***circular linked list*** is obtained by setting the link of the last node in a standard linear linked list to point to the first node.

Each node in a circular linked list has a predecessor (and a successor), provided that the list is nonempty.

For example, the list of integers 9, 17, 22, 26, 34 can be stored in a circular linked list as follows:

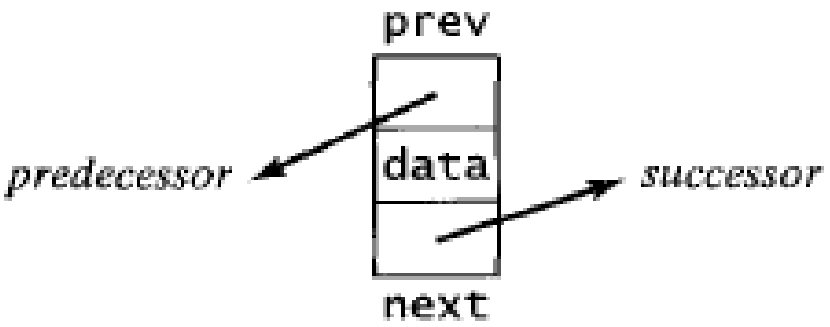


Doubly Linked Lists:

Bidirectional lists constructed using nodes that contain,

- data part,

- two links: a forward link ***next*** pointing to the successor of the node and a backward link ***prev*** pointing to its predecessor, are referred to as Doubly linked lists.



Doubly Linked Lists (contd.)

To facilitate both forward and backward traversal, a pointer ***first*** provides access to the first node and another pointer ***last*** provides access to the last node.

For example, a doubly-linked list of integers 9,17, 22, 26, 34
might be pictured as follows: