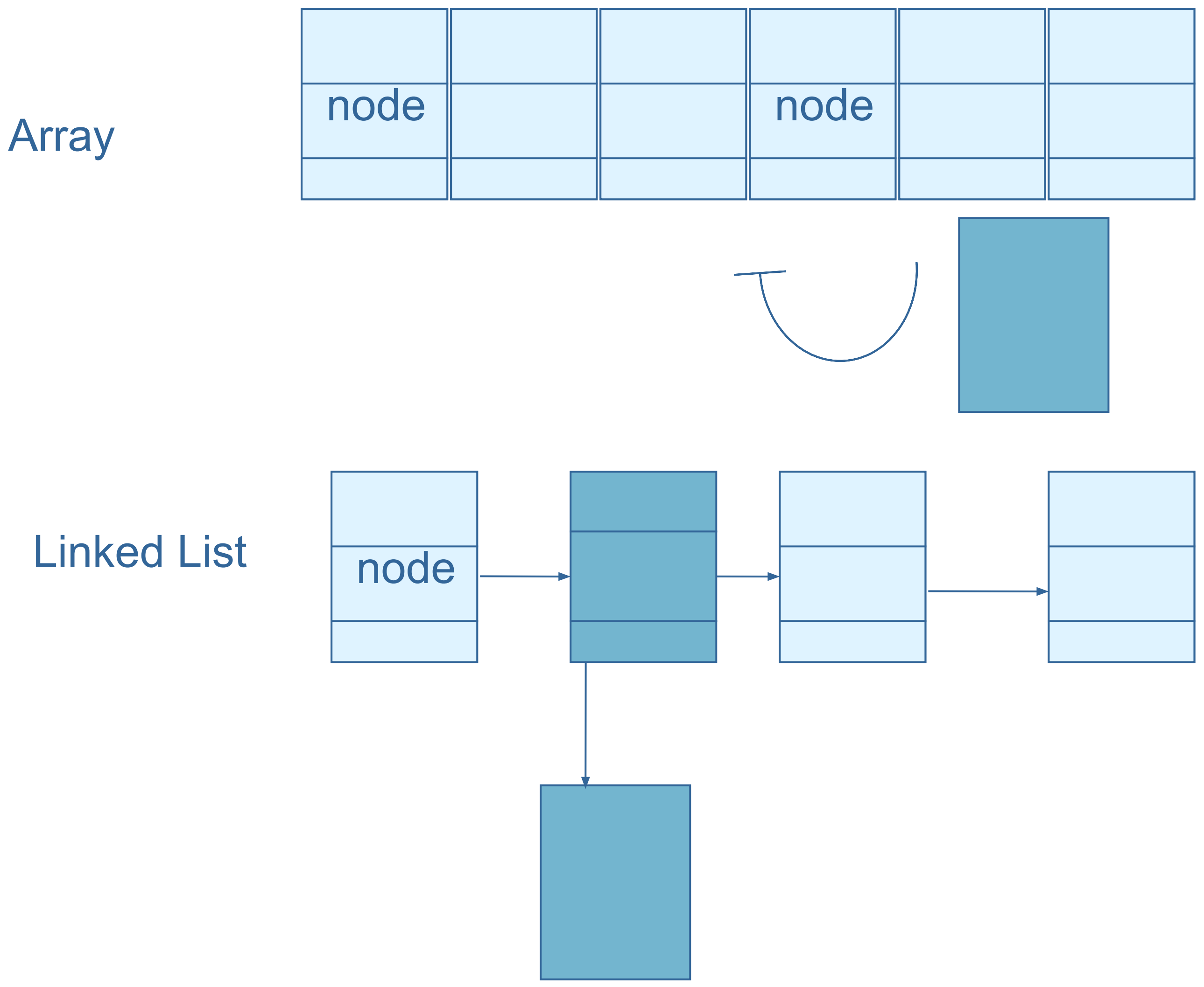
LINKED LISTS

Array vs Linked List



What’s wrong with Array and Why lists?

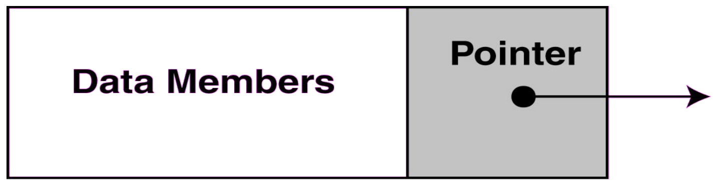
- Disadvantages of arrays as storage data structures:
  - slow searching in unordered array
  - slow insertion in ordered array
  - Fixed size
- Linked lists solve some of these problems
- Linked lists are general purpose storage data structures and are versatile.

Linked list

- Linked list an ordered collection of data in which each element contains the location of the next element.
- Each element contains two parts: ***data and link***.
- The link contains a pointer (an address) that identifies the next element in the list.
- **Singly linked list**
- The link in the last element contains a null pointer, indicating the end of the list.

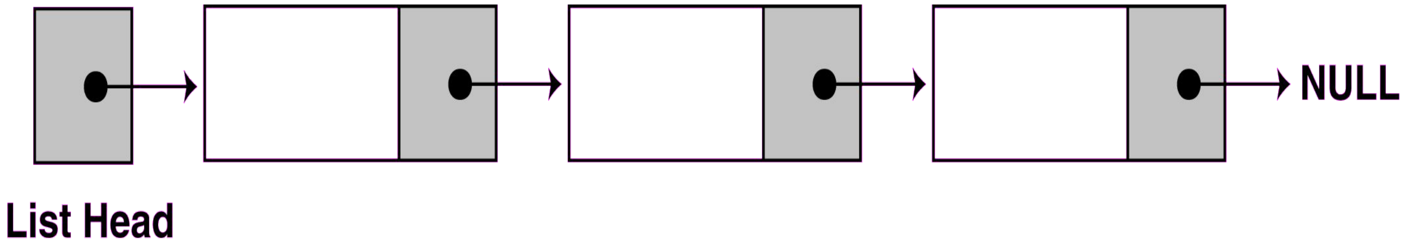
The composition of a Linked List

- Each node in a linked list contains one or more members that represent data.
- In addition to the data, each node contains a pointer, which can point to another node.



The composition of a Linked List

- A linked list is called "linked" because each node in the series has a pointer that points to the next node in the list.



Linked Lists

- Each data item is embedded in a link.
- Each Link object contains a reference to the next link in the list of items.
- In an array items have a particular position, identified by its index.
- In a list the only way to access an item is to traverse the list
- Is Linked List an ADT?

Basic Operations

Each operation is developed using before and after figures to show the changes.

- Insertion
- Deletion
- Retrieval
- Traversal

Declarations

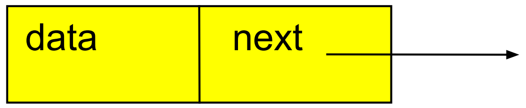
- First you must declare a data structure that will be used for the nodes.
- For example, the following `struct` could be used to create a list where each node holds a `float`:

```
struct ListNode
{
   float value;
   struct ListNode *next;
};
```

Linked List Implementation/Coding Issues in C

- We can define structures with pointer fields that refer to the structure type containing them

```
struct list {
        int data;
        struct list *next;
}
```

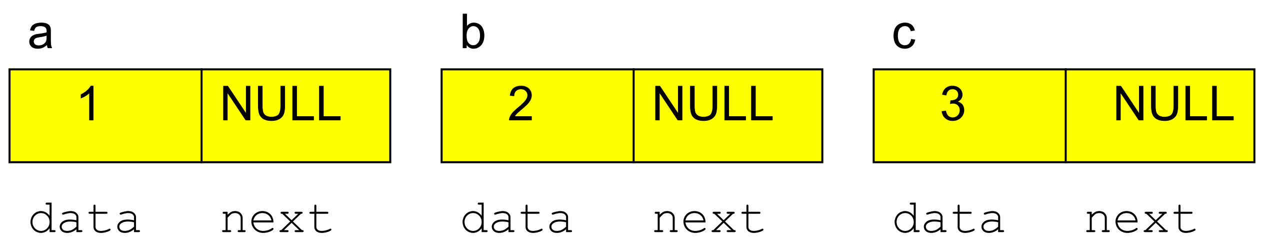


- The pointer variable next is called a *link.*
- Each structure is linked to a succeeding structure by way of the field next.
- The pointer variable next contains an address of either the location in memory of the successor struct list element or the special value NULL.

Example

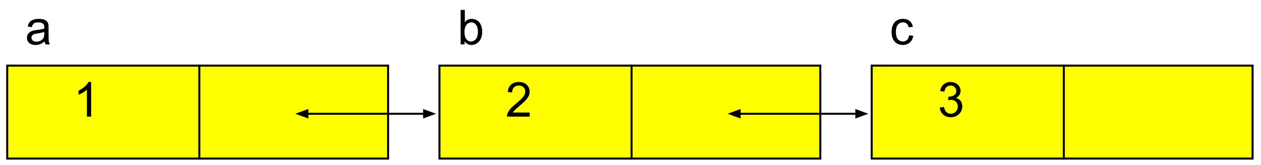
```
struct list a, b, c;

a.data = 1;
b.data = 2;
c.data = 3;
a.next = b.next = c.next = NULL;
```



Example continues

- `a.next = &b;`
- `b.next = &c;`
- `a.next -> data` has value `2`
- `a.next -> next ->data` has value `3`
- `b.next -> next -> data` error !!



Dynamic Memory Allocation

- Creating and maintaining dynamic data structures requires dynamic memory allocation – the ability for a program to obtain more memory space at execution time to hold new values, and to release space no longer needed.

- In C, functions *malloc* and *free*, and operator *sizeof* are essential to dynamic memory allocation.

Dynamic Memory Operators *sizeof* and *malloc*

- Unary operator *sizeof* is used to determine the size in bytes of any data type.

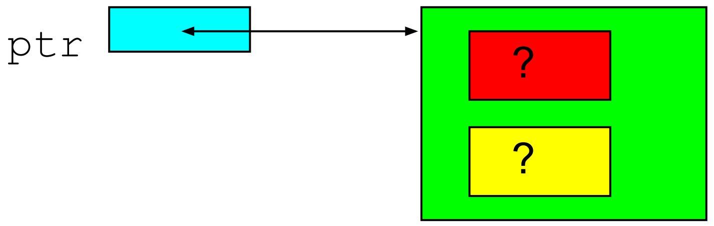
  sizeof(double)   sizeof(int)

- Function *malloc* takes as an argument the number of bytes to be allocated and return a pointer of type void * to the allocated memory. (A void * pointer may be assigned to a variable of any pointer type). It is normally used with the *sizeof* operator.

Dynamic Memory Operators in C Example

```
struct node{
    int data;
    struct node *next;
};
 struct node *ptr;

ptr = (struct node *)    /*type casting */
      malloc(sizeof(struct node));
```



The *Free* Operator in C

- Function *free* deallocates memory- i.e. the memory is returned to the system so that the memory can be reallocated in the future.

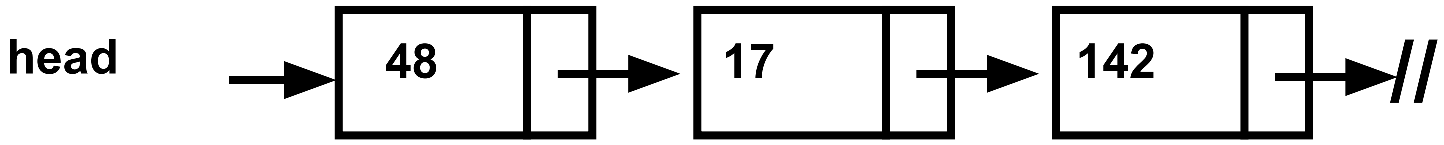
free(ptr);

ptr

Insertion of a node Into Linked Lists

The Scenario

- You have a linked list
  - Perhaps empty, perhaps not
  - Perhaps ordered, perhaps not



- You want to add an element into the linked list

Adding an Element to a Linked List

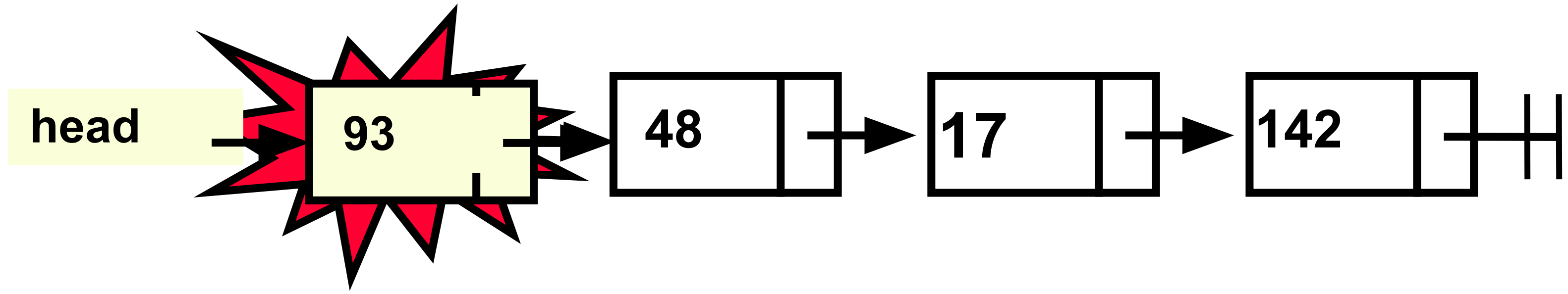
Involves two steps:

- **Finding the correct location**
- **Doing the work to add the node**

Finding the Correct Location

- **Three possible positions:**
  - **The front**
  - **The end**
  - **Somewhere in the middle**

Inserting to the Front



- There is no work to find the correct location
- Empty or not, head will point to the right location

Algorithm-Inserting a node to the front

If p is a pointer to a node, then

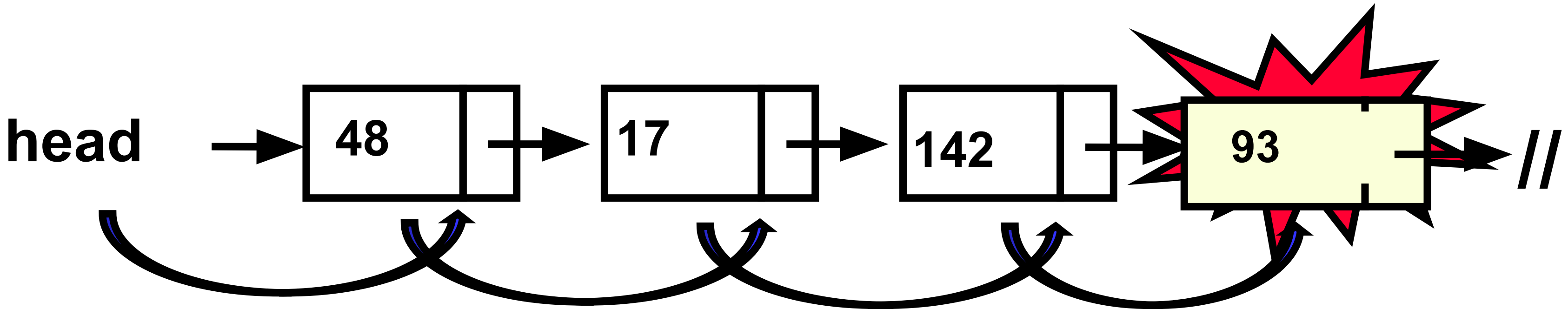
node(p)= node pointed by p

info(p)-information portion of that node

next(p)- next address portion

```
p=getnode()
info(p)=x
next(p)=list
list=p
```

Inserting to the End

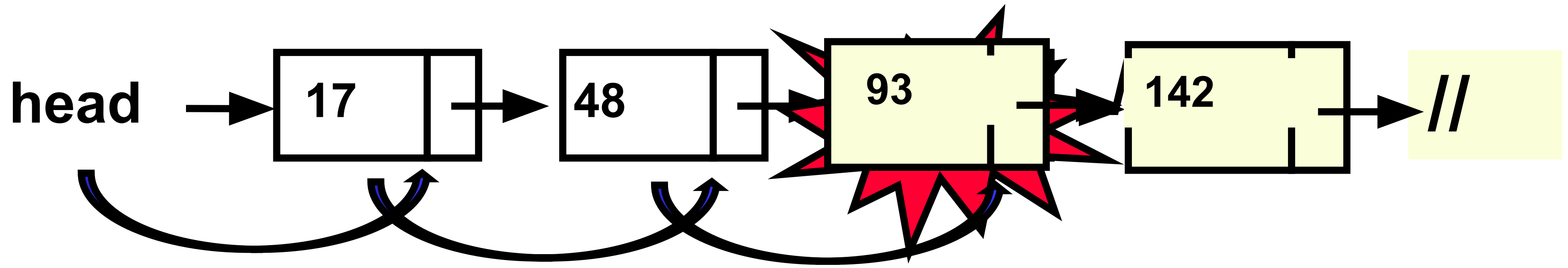


- Find the end of the list (when at NIL)
  - Recursion or iteration

Algorithm- Inserting a Node to the End

```
p=list
while(next(p)!=NULL)
p=next(p)
q=getnode()
info(q)=x
next(q)=NULL
next(p)=q
```

Inserting to the Middle

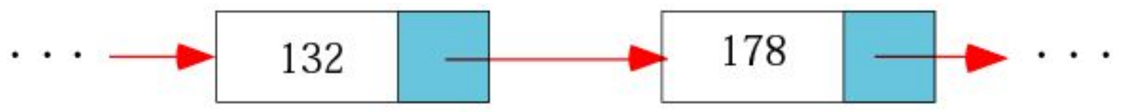


- Used when order is important
- Go to the node that should follow the one to add
  - Recursion or iteration

Algorithm- Inserting a Node at Location

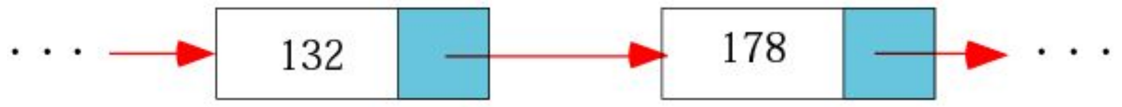
```
q=list
For i=1 to position-1
{
q=p
next(p)=p
}
newnode= getnode()
info(newnode)=x
next(newnode)=next(q)
next(q)= newnode
```

Inserting
a node to the middle

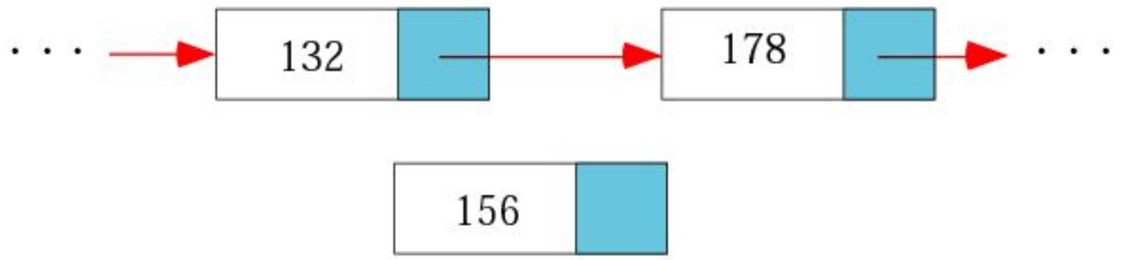


Original List

Inserting
a node to the middle

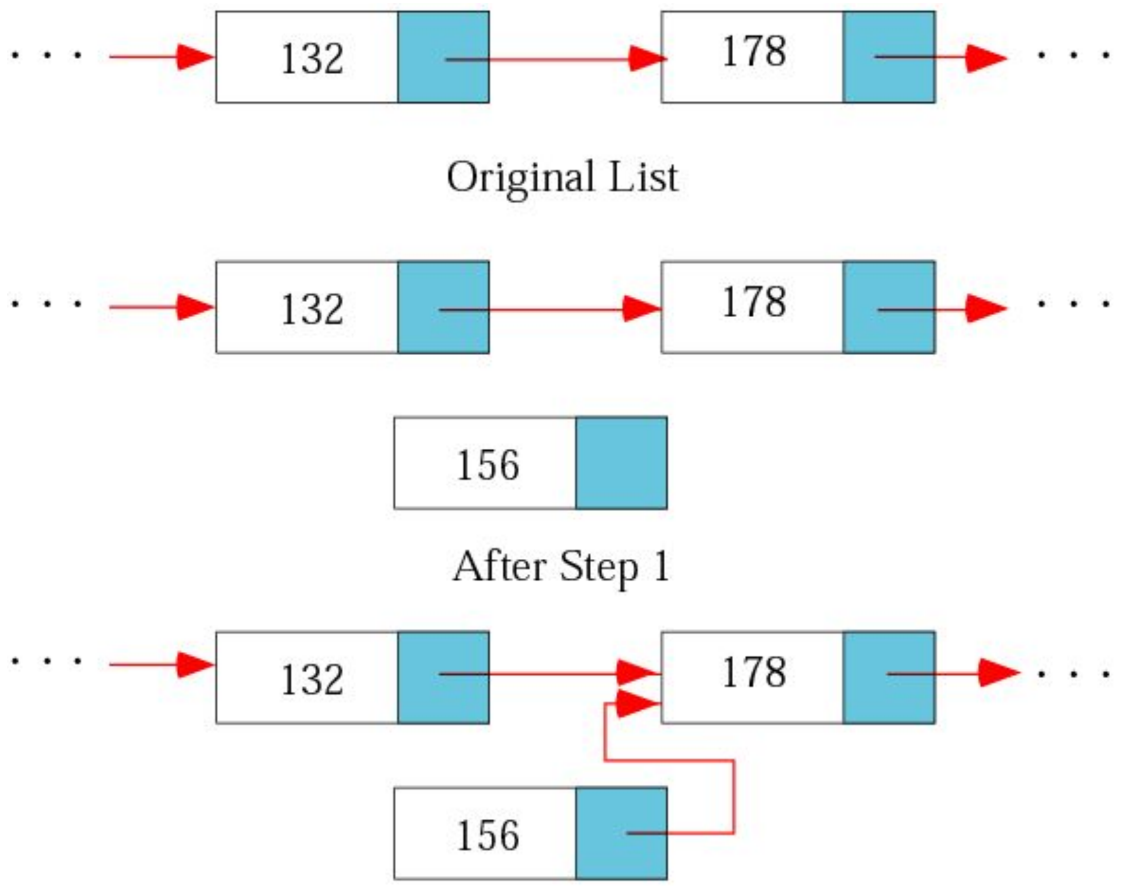


Original List



After Step 1

Inserting
a node to the middle

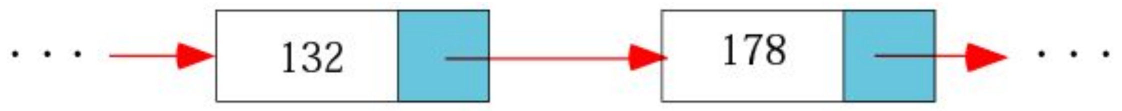


Original List

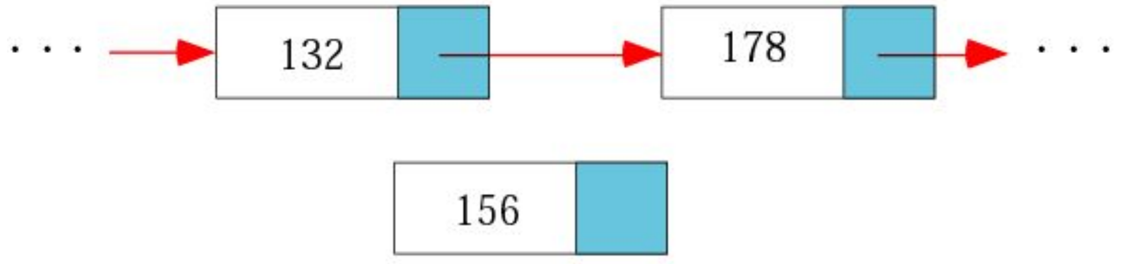
After Step 1

After Step 2

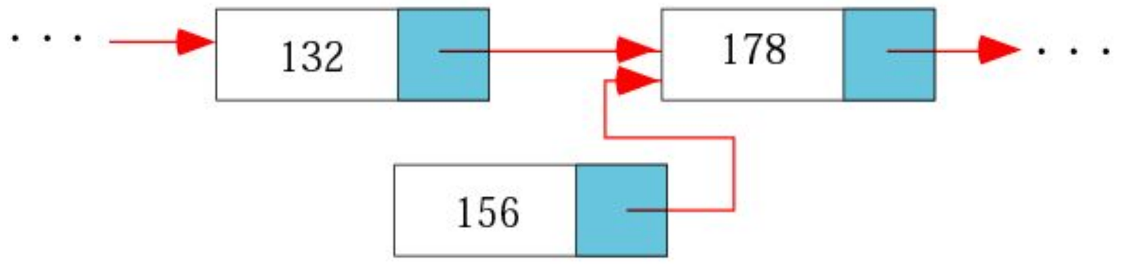
Inserting a node to the middle



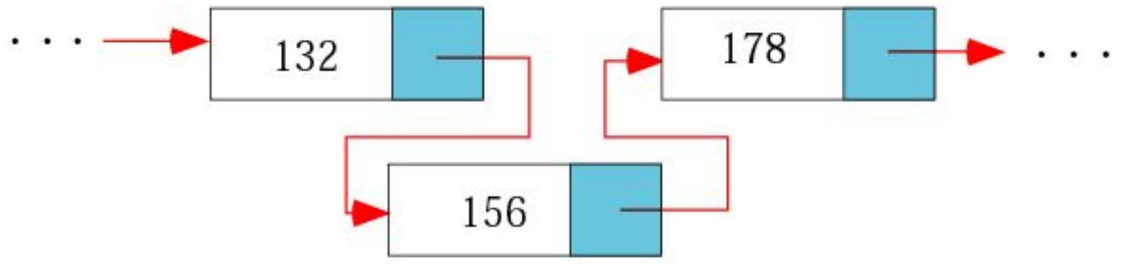
Original List



After Step 1



After Step 2



After Step 3

Deletion of a node Into Linked Lists

There are three situation for Deleting element in list.

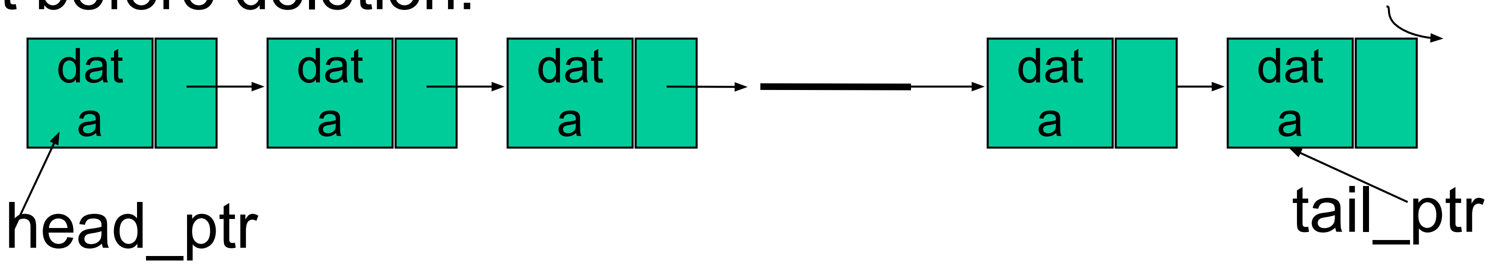
1.Deletion at beginning of the list.

2.Deletion at the middle of the list.

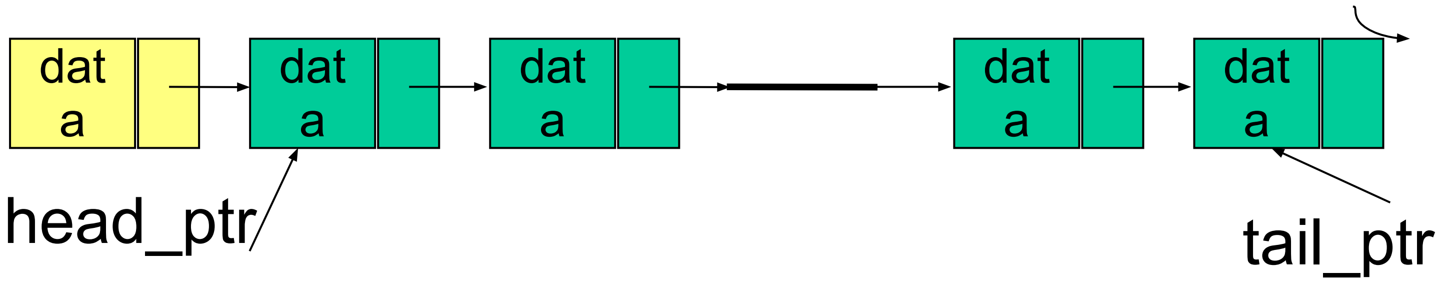
3.Deletion at the end of the list.

Delete – the first node

List before deletion:



List after deletion of the head item:



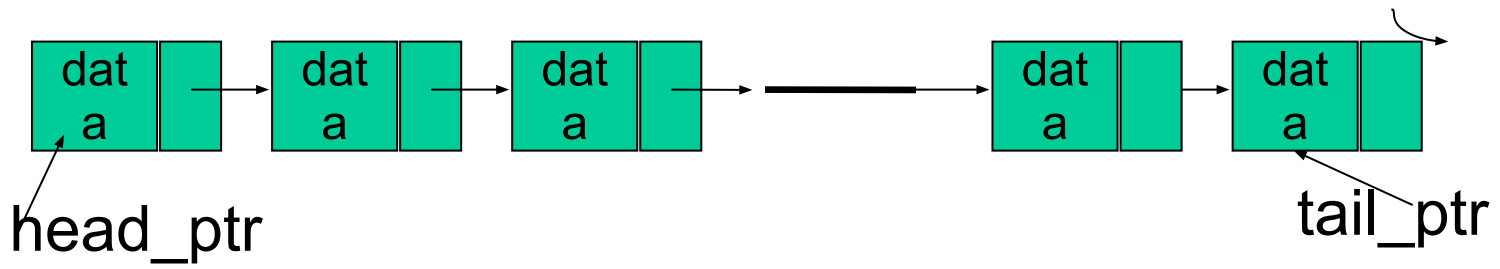
- The new value of head_ptr = link-value of the old head item
- The old head item is deleted and its memory returned

Algorithm- Deleting a Node at the front

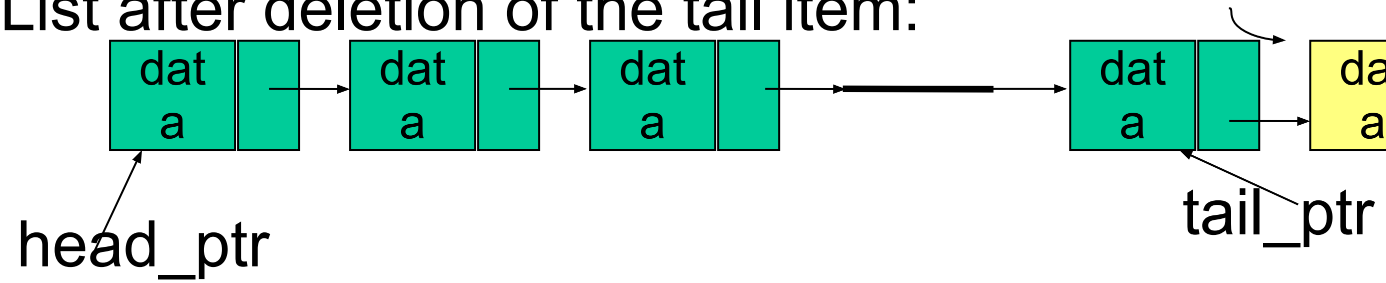
```
p=list
list=next(p)
x=info(p)
freenode(p)
```

Delete – the end node

List before deletion:



List after deletion of the tail item:



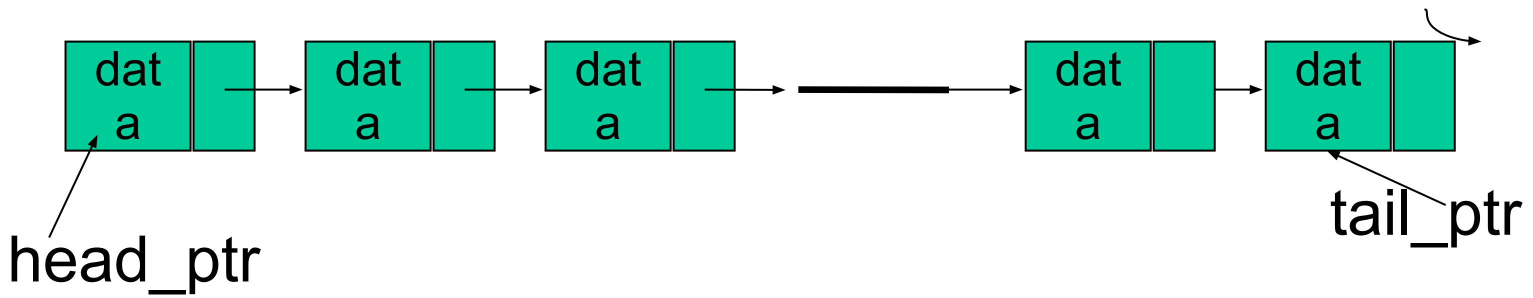
- New value of tail_ptr = link-value of the 3rd from last item
- New link-value of new last item = **NULL**.

Algorithm- Deleting a Node at the End

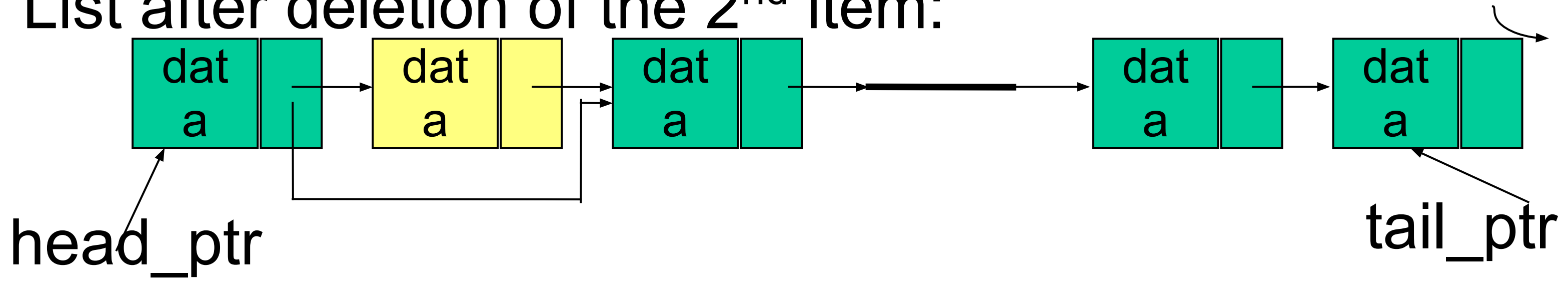
```
p=list
while(next(p)!=NULL)
{   q=p
p=next(p)
}
x=info(p)
next(q)=NULL
freenode(p)
```

Delete – an inside node

List before deletion:

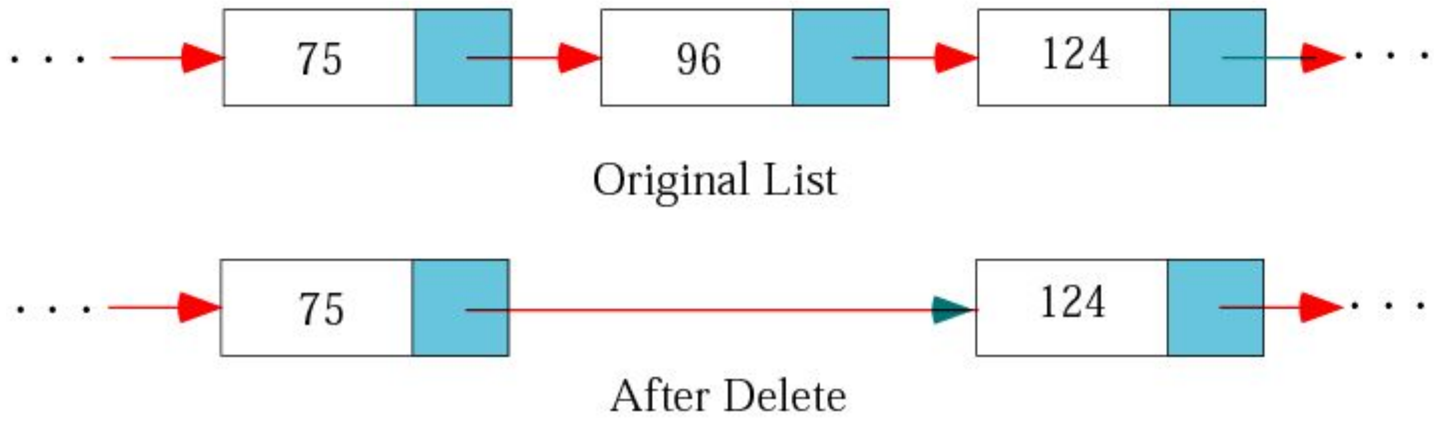


List after deletion of the 2nd item:

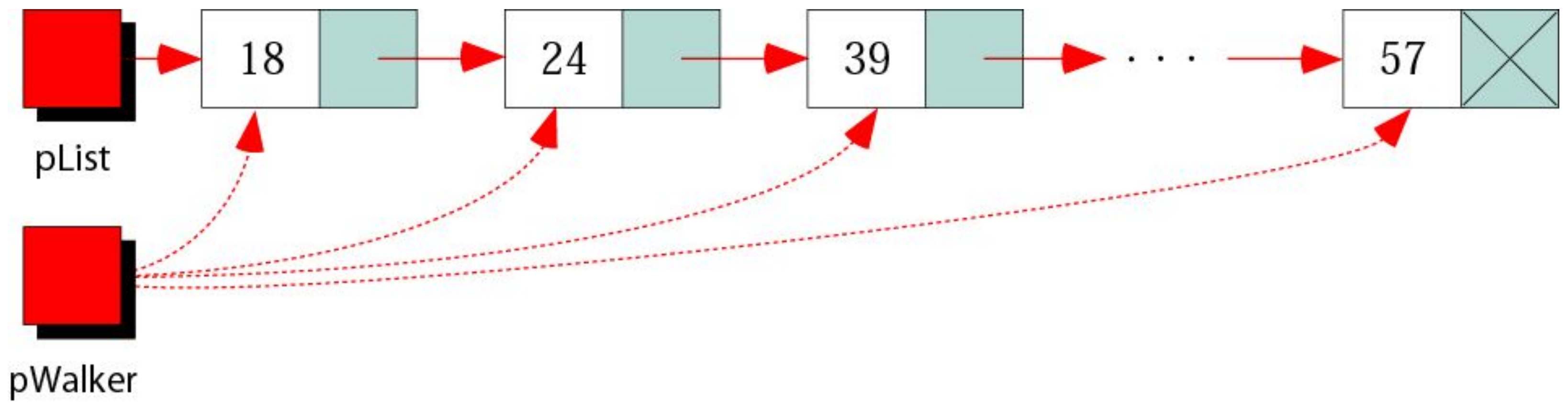


•New link-value of the item located before the deleted one = the link-value of the deleted item

Deleting a node at a Location



Traversing a list



Linked Implementation as Stack

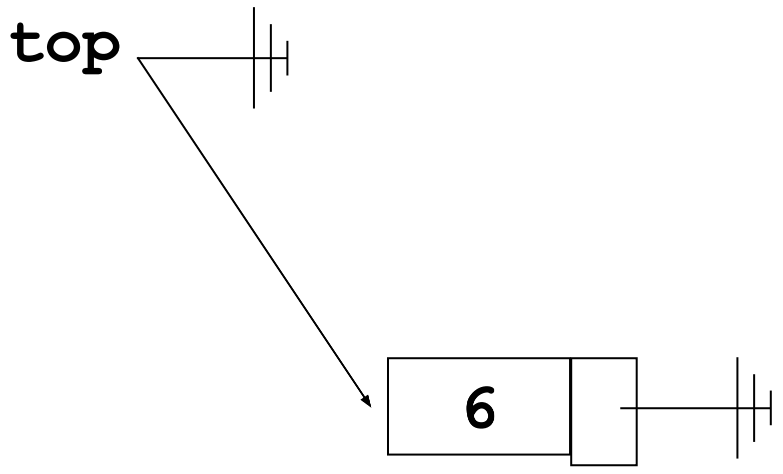
The push operation can be implemented by using the operation of adding the element/nodes at the front of the list.

The pop operation can be implemented by using the operation of removing the element/nodes at the front of the list.

List Stack Example

Java Code

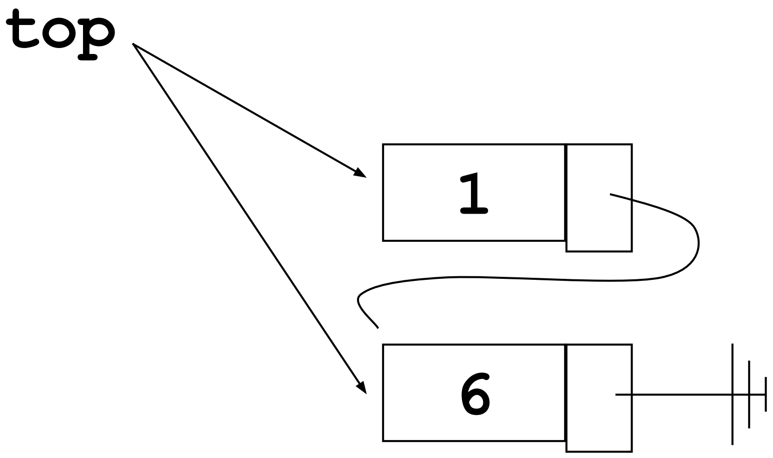
```
Stack st = new Stack();
st.push(6);
```



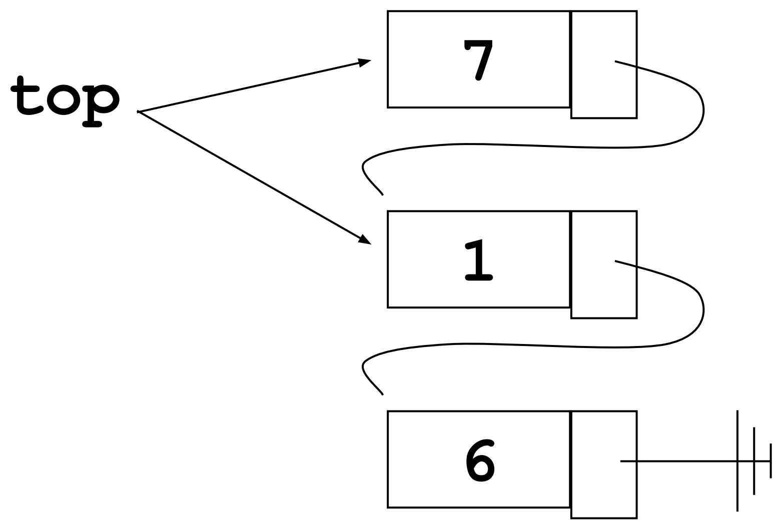
List Stack Example

Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
```



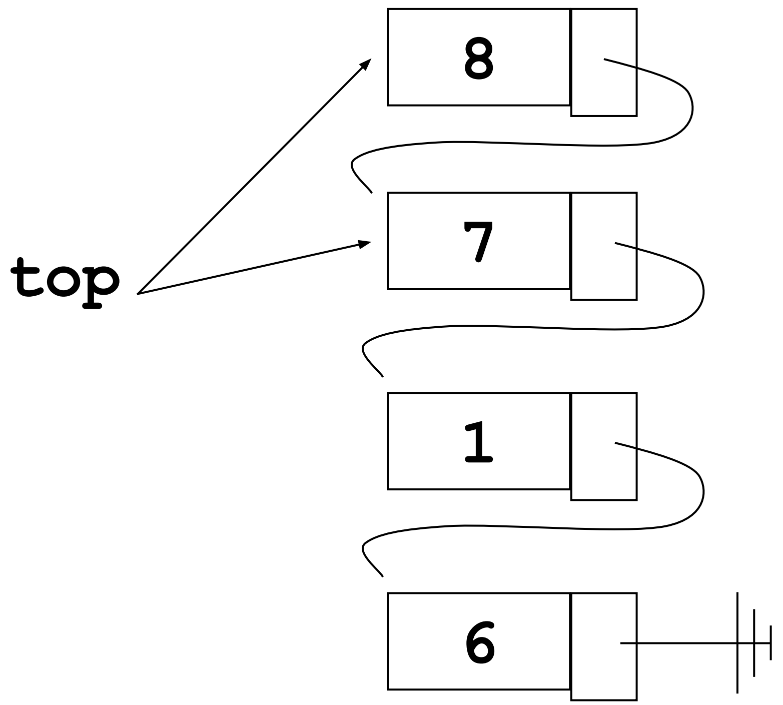
List Stack Example



Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
```

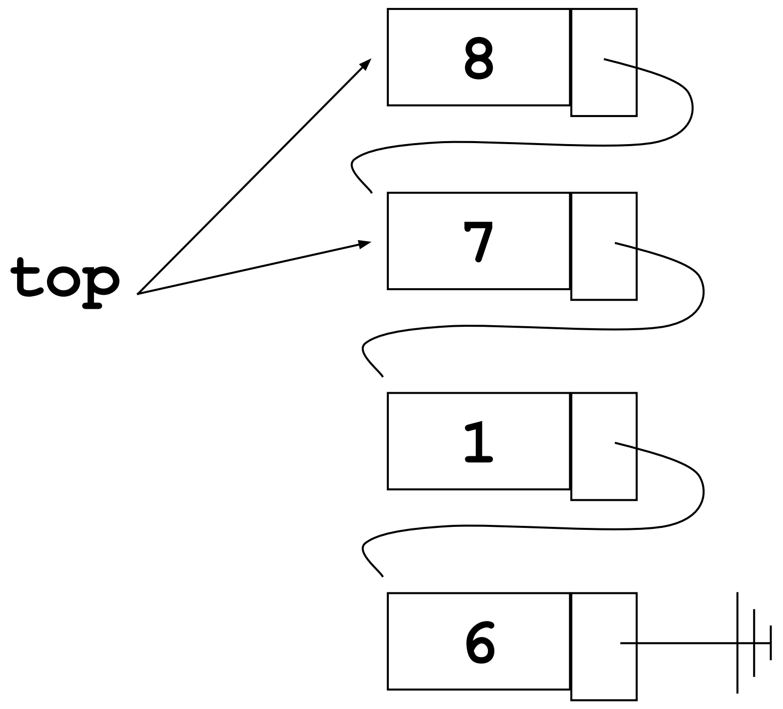
List Stack Example



Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
```

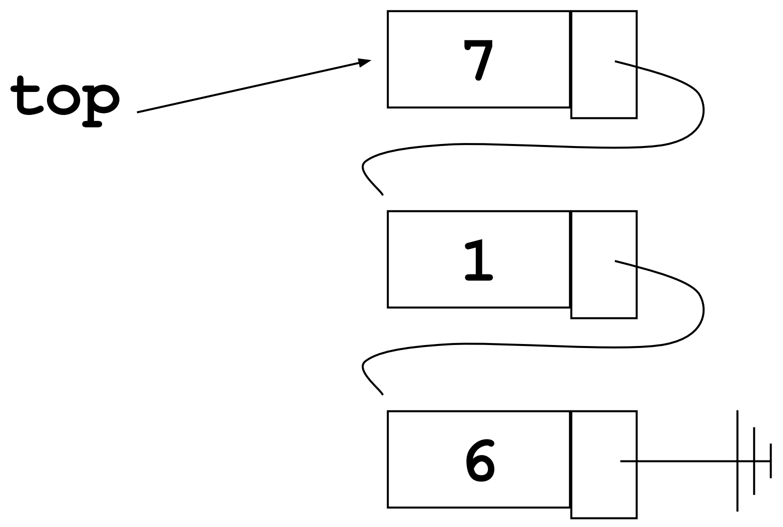
List Stack Example



Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
st.pop();
```

List Stack Example



Java Code

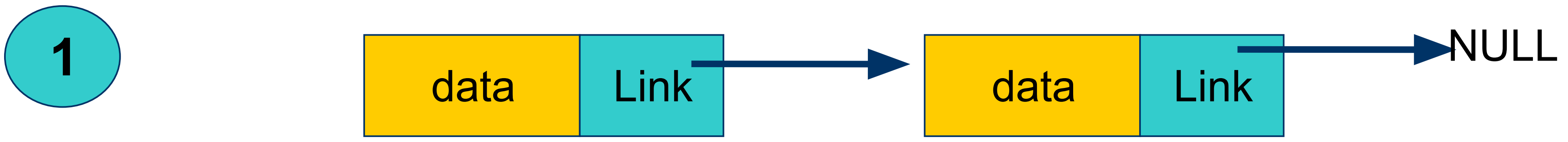
```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
st.pop();
```

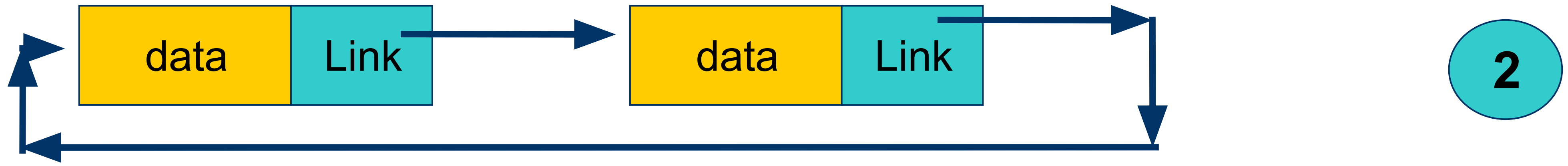
Linked Implementation as Queue

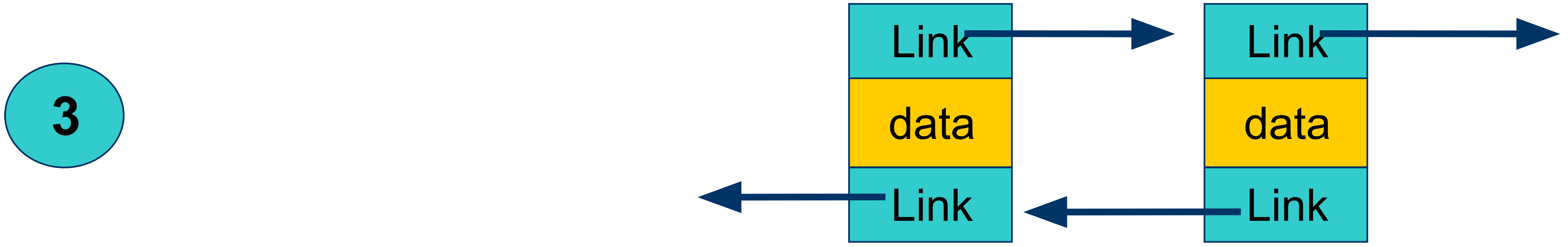
The insert operation can be implemented by using the operation of adding the element/nodes at the end of the list.

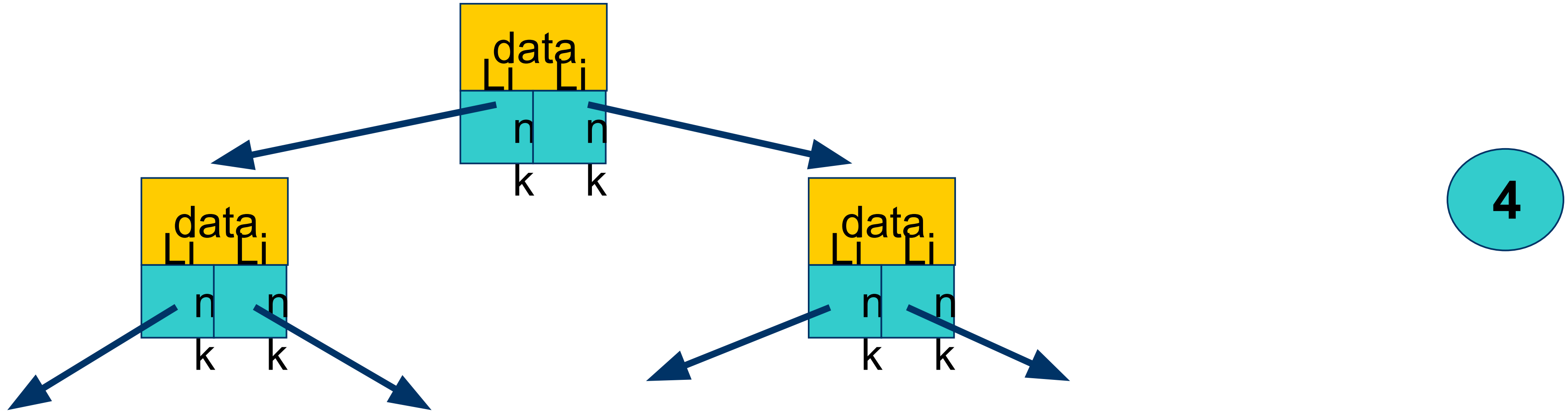
The delete operation can be implemented by using the operation of removing the element/nodes at the front of the list.

Type of Linked List









Types of linked lists

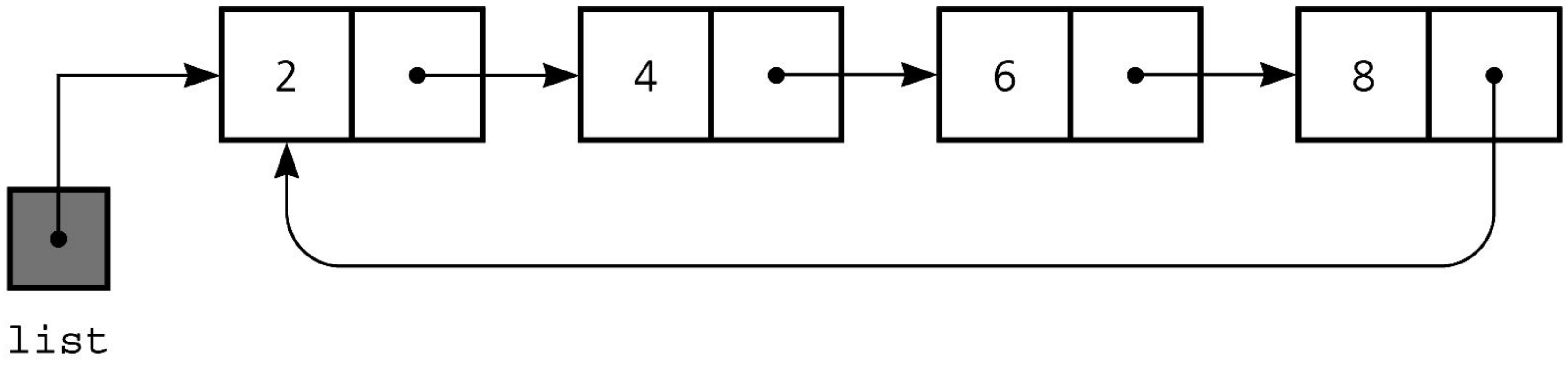
- Singly linked list
    - Begins with a pointer to the first node
    - Terminates with a null pointer
    - Only traversed in one direction
- Circular, singly linked
    - Pointer in the last node points back to the first node
- Doubly linked list
    - Two “start pointers” – first element and last element
    - Each node has a forward pointer and a backward pointer
    - Allows traversals both forwards and backwards
- Circular, doubly linked list
    - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

Circular Linked Lists

In linear linked lists if a list is traversed (all the elements visited) an external pointer to the list must be preserved in order to be able to reference the list again.

Circular linked lists can be used to help the traverse the same list again and again if needed.

A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node.



Circular Linked Lists

- Access to last node requires a traversal
- Make external pointer point to last node instead of first node
  - Can access both first and last nodes without a traversal

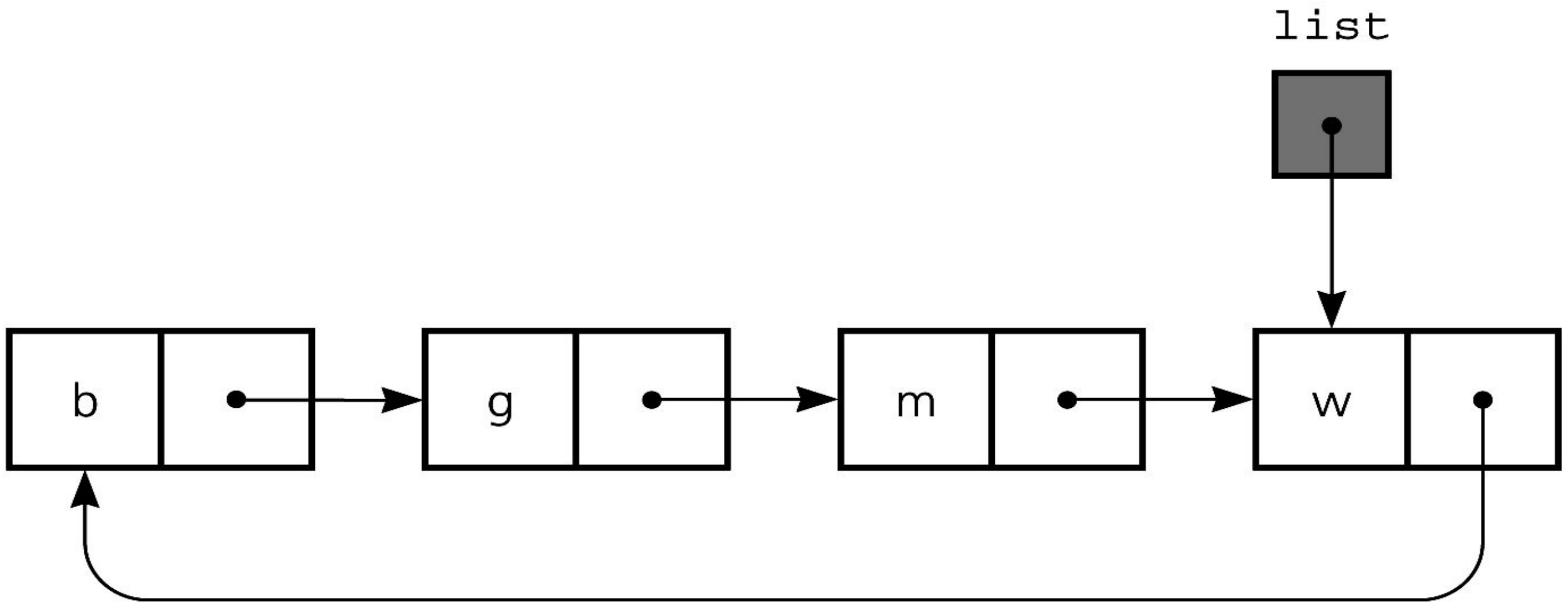


Figure A circular linked list with an external pointer to the last node

Algorithm- Inserting a Node to the front of Circular List

```
p=getnode()
info(p)=x
if(list==NULL)
    –   next(p)=p, list=p
next(p)=next(list)
next(list)=p
```

Algorithm- Inserting a Node at the end of Circular List

```
p=getnode()
info(p)=x
if(list==NULL)
        {
        next(p)=p,
        list=p
        }
next(p)=next(list)
next(list)=p
list=p
```

Doubly-Linked Lists

- It is a way of going *both* directions in a linked list, *forward* and *reverse*.
- Many applications require a quick access to the *predecessor* node of some node in list.

Advantages over Singly-linked Lists

Quick update operations:

such as: insertions, deletions at *both* ends (head and tail), and also at the middle of the list.

A node in a doubly-linked list store two references:

A ***next*** link; that points to the next node in the list, and

A ***prev*** link; that points to the previous node in the list.

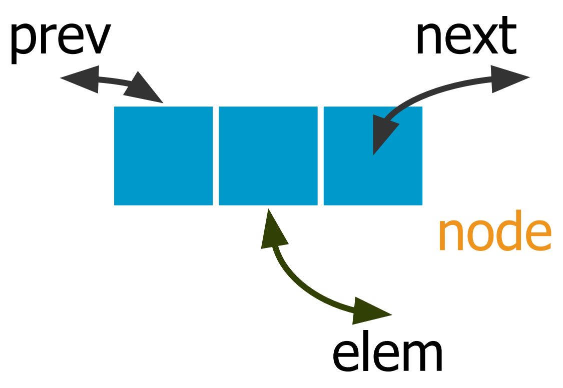
Doubly Linked List

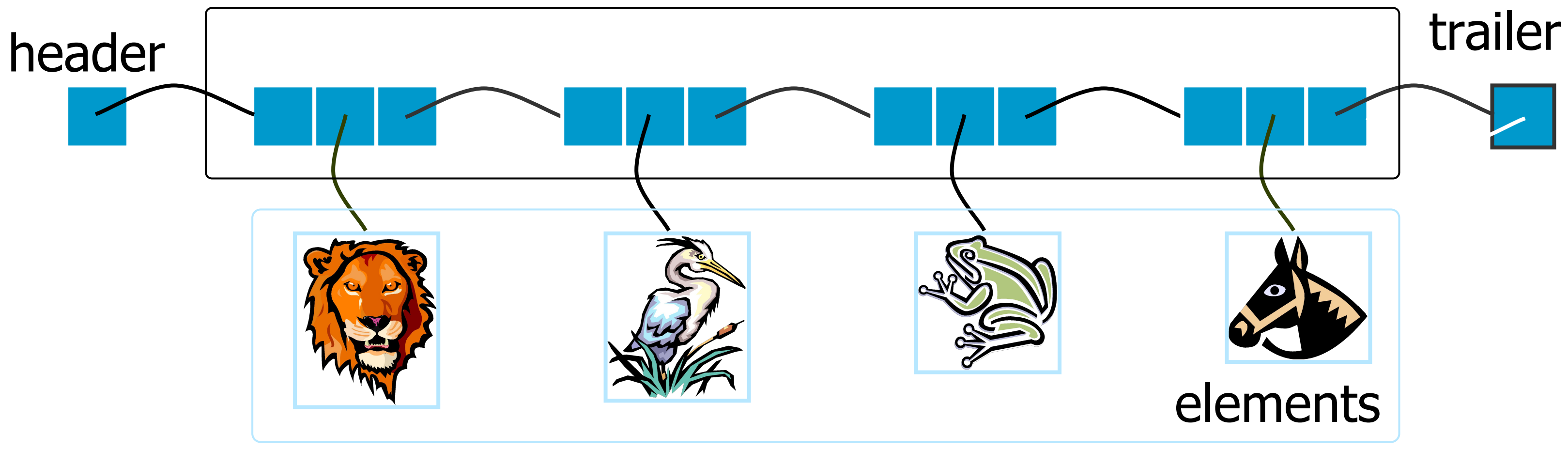
A doubly linked list provides a natural implementation of the List ADT

Nodes implement Position and store:

- **element**
- **link to the previous node**
- **link to the next node**

Special trailer and header nodes





Sentinel Nodes

To simplify programming, two special nodes have been added at both ends of the doubly-linked list.

Head and tail are dummy nodes, also called **sentinels**, do not store any data elements.

Head: header sentinel has a ***null-prev*** reference (link).

Tail: trailer sentinel has a ***null-next*** reference (link).