# Linked Lists

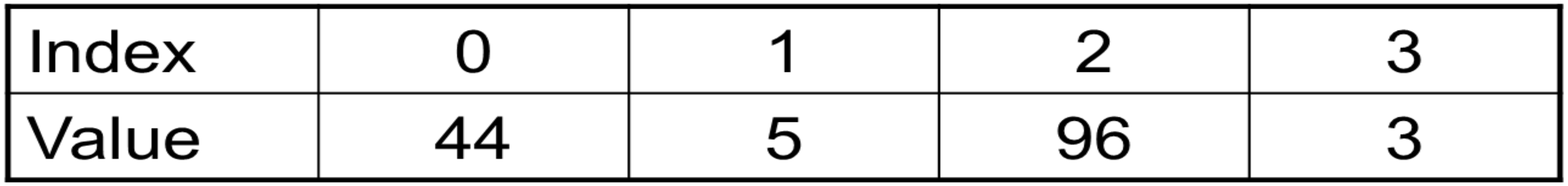
Chapter - 6

## Advantages of Arrays

- Simple and easy to use
- Faster access to the elements (constant access)

## Disadvantages of Arrays

- Preallocates all needed memory up front and wastes memory space for indices in the array that are empty.
- **Fixed size**



|  |  |  |  |  |
|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 |
| Value | 44 | 5 | 96 | 3 |

# Linked Lists

- list elements are stored, in memory, in an arbitrary order.
- In a linked representation, each element of an instance of a data object is represented in a cell or node.
- Each node keeps explicit information (called a link) about the location of other relevant nodes.

## Linked Lists

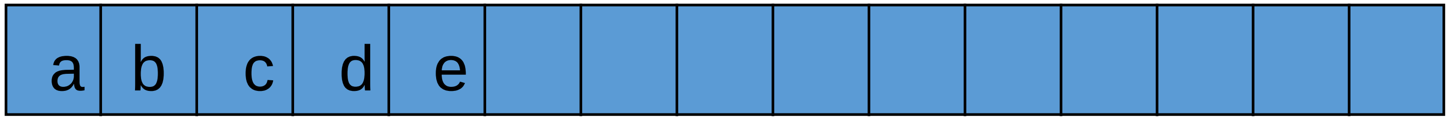
A linked list is a data structure used for storing collections of data.

A linked list has the following properties.

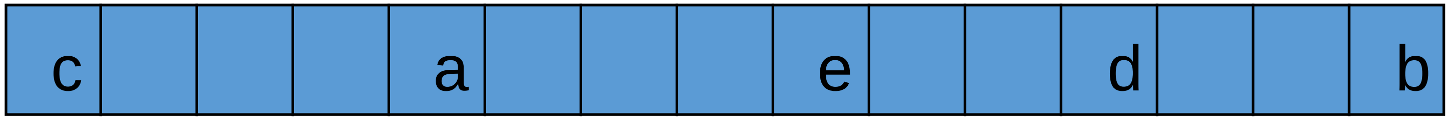
- Successive elements are connected by pointers
- The last element points to NULL
- Can grow or shrink in size during execution of a program
- Can be made just as long as required (until systems memory exhausts)
- Does not waste memory space (but takes some extra memory for pointers). It allocates memory as list grows.

# Memory Layout

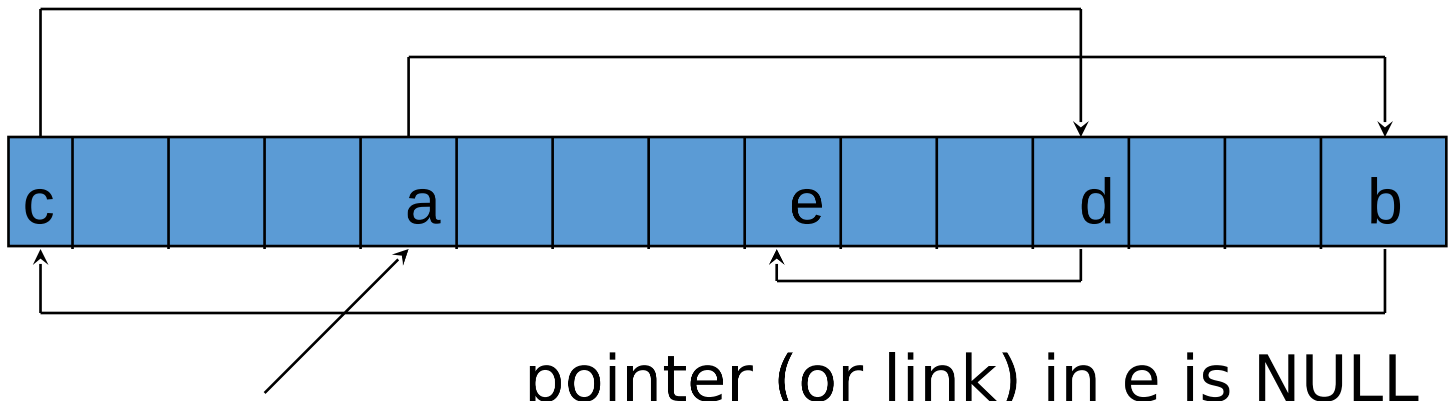
Layout of L = (a,b,c,d,e) using an array representation.



A linked representation uses an arbitrary layout.



# Linked Representation

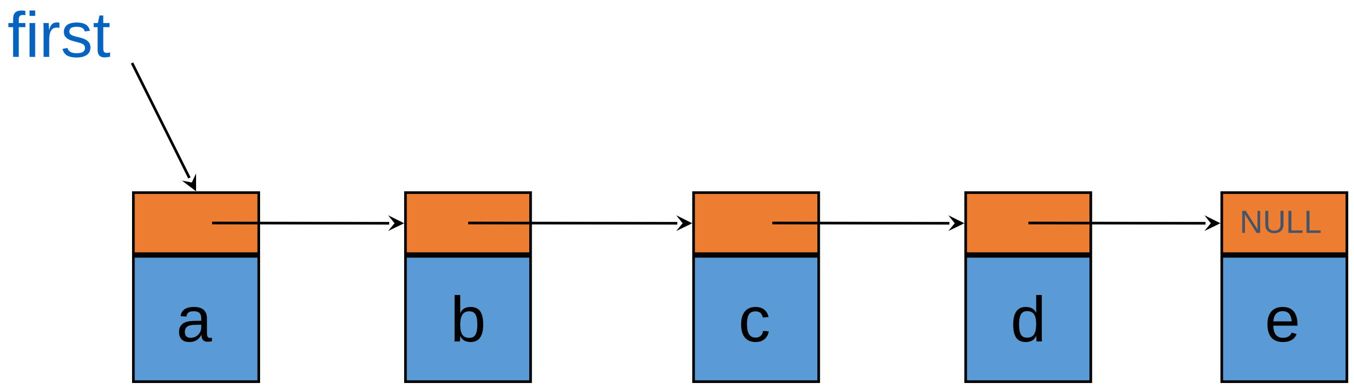


pointer (or link) in e is NULL

first

use a variable first to get to the first element a

# Normal Way To Draw A Linked List

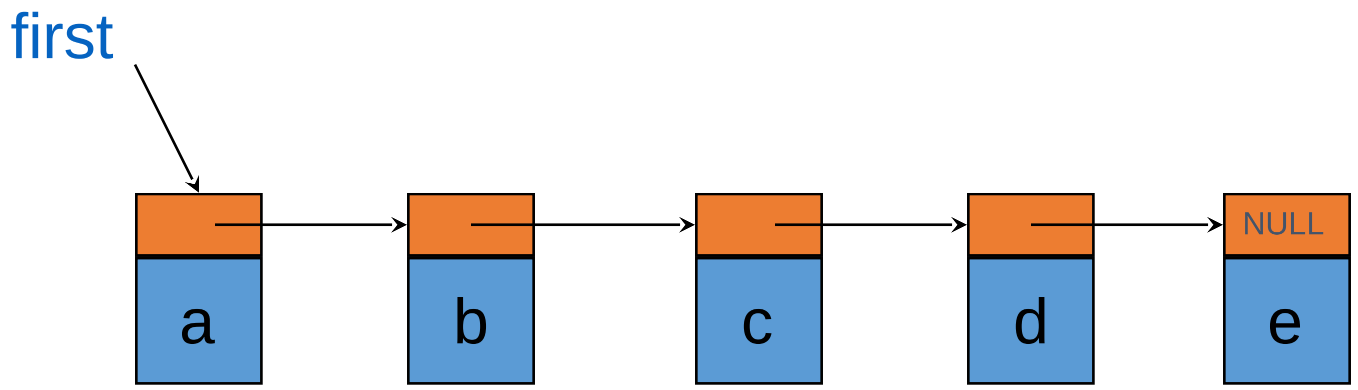


🟧 link or pointer field of node

🟦 data field of node

Since each node in this linked representation has exactly one link, the structure is called singly linked list.

# Chain

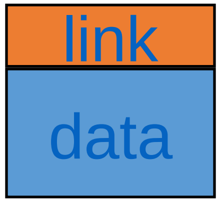


- A chain is a linked list in which each node represents one element.
- There is a link or pointer from one element to the next.
- The last node has a NULL (or 0) pointer.

# Node Representation

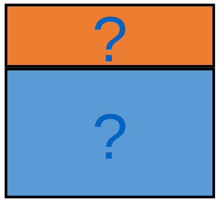
```
          <      T>
      ChainNode
{
          :
   T data;
   ChainNode<T> *link;

   // constructors come here
};
```

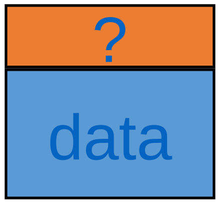


# Constructors Of ChainNode

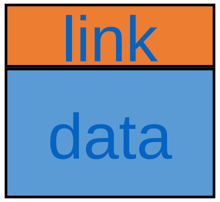
```
ChainNode() {}
```



```
ChainNode(const T& data)
     {this->data = data;}
```



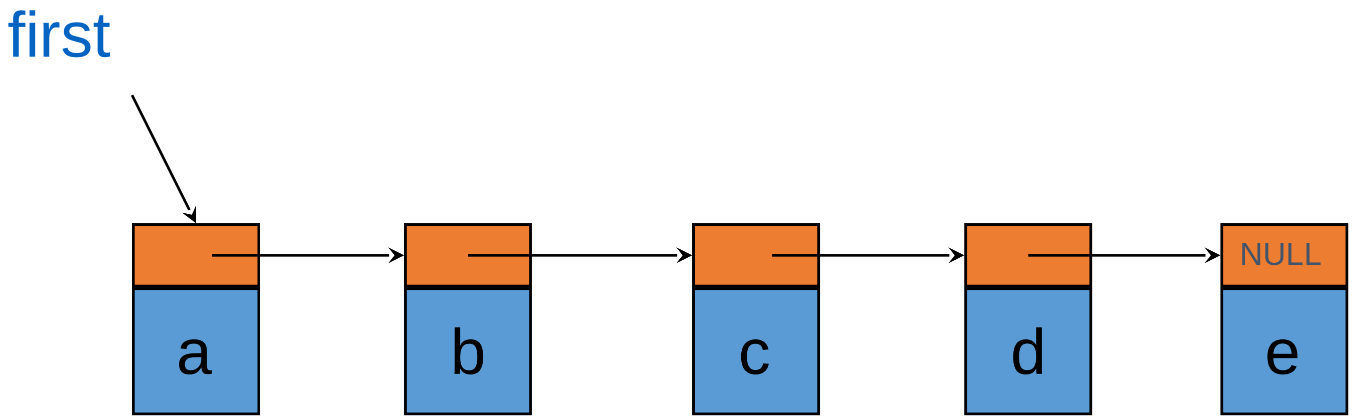
```
ChainNode(const T& data, chainNode<T>* link)
     {this->data = data;
      this->link = link;}
```



## Operations

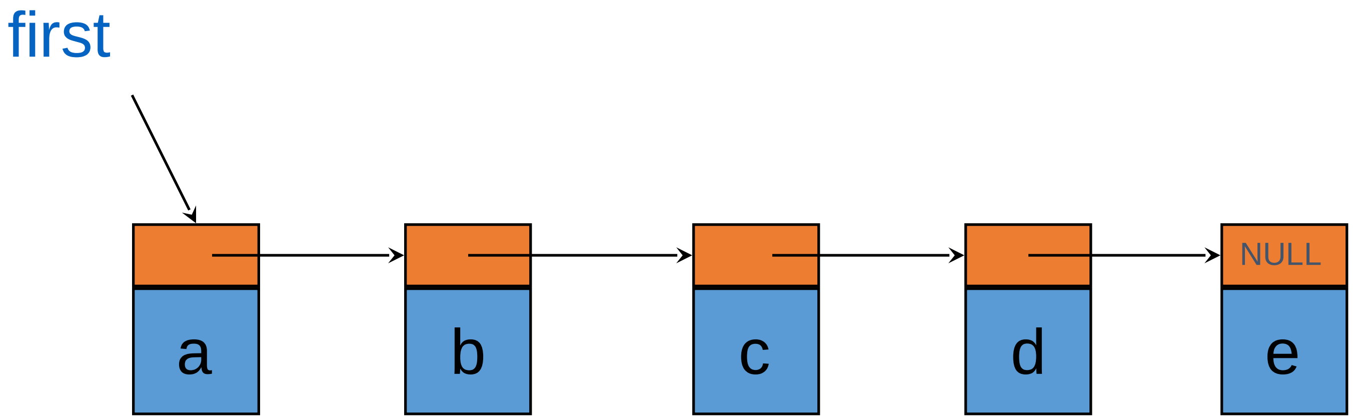
- Insert: inserts an element into the list
- Delete: removes and returns the specified position element from the list
- Delete List: removes all elements of the list (disposes the list)
- Count: returns the number of elements in the list
- Get: Find nth node from the end of the list

## Get(0)



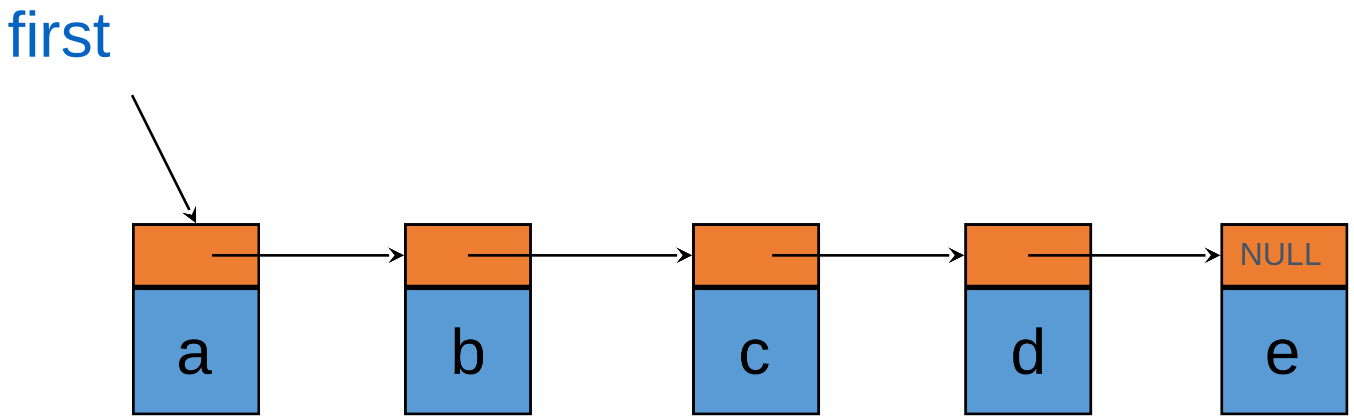
```
desiredNode = first; // gets you to first node
return desiredNode–>data;
```

# Get(1)



```
desiredNode = first–>link; // gets you to second node
return desiredNode–>data;
```

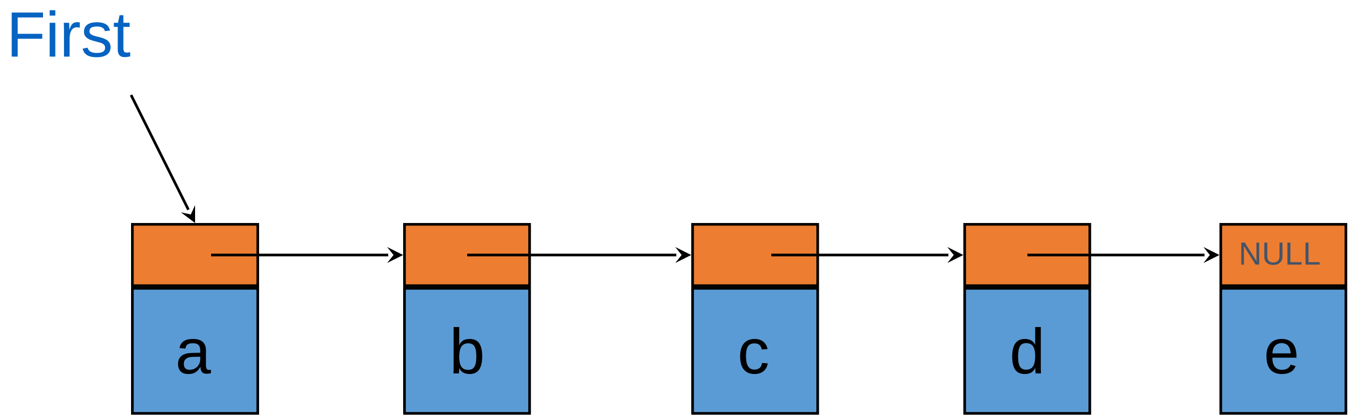
## Get(2)



desiredNode = first–>link–>link; // gets you to third node

return desiredNode–>data;

## Get(5)

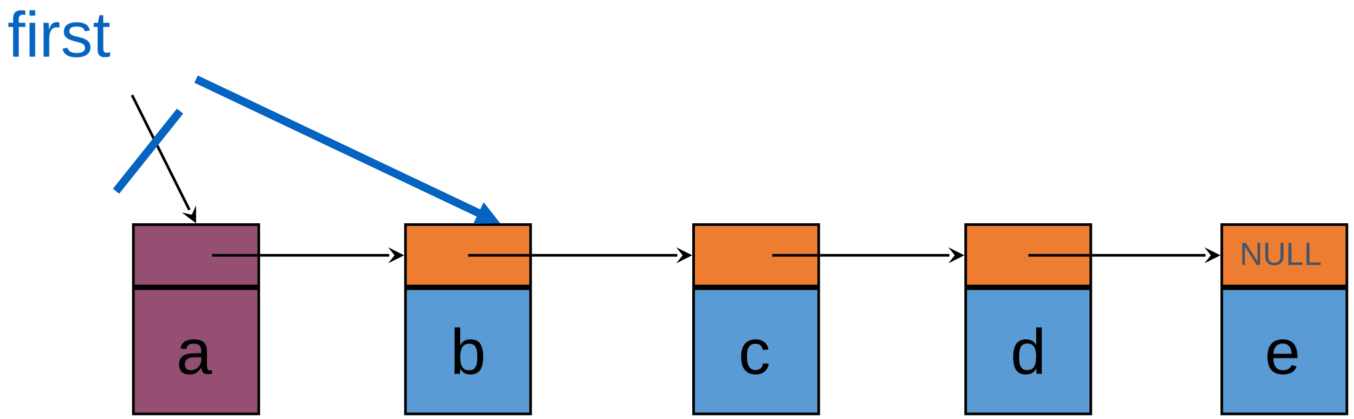


desiredNode = first–>link–>link–>link–>link–>link;

// desiredNode = NULL

return desiredNode–>data; // NULL.element

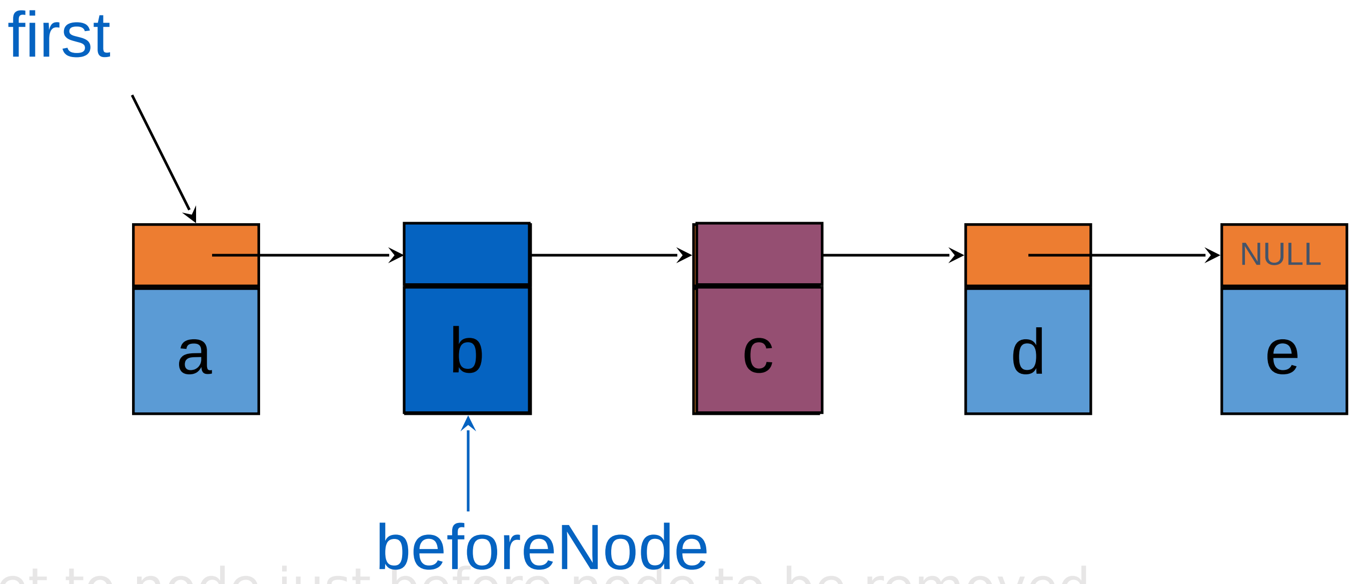
# Delete An Element



Delete(0)

```
deleteNode = first;
first = first–>link;
          deleteNode;
```

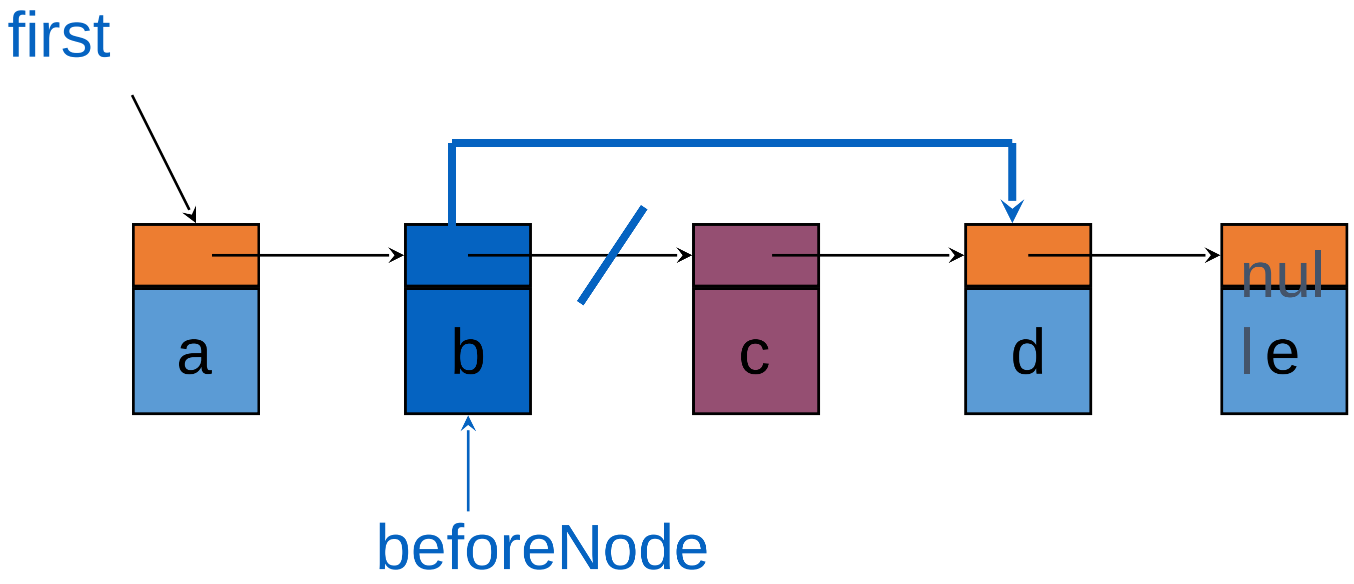
# Delete(2)



first get to node just before node to be removed

beforeNode = first–>link;

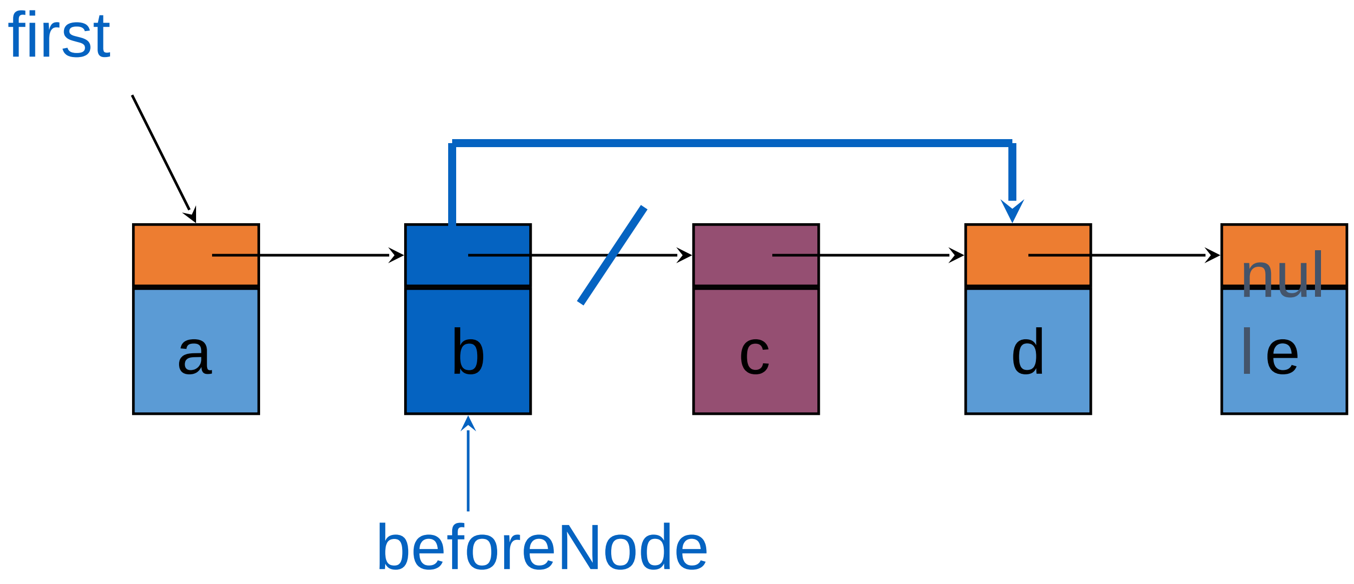
# Delete(2)



save pointer to node that will be deleted

deleteNode = beforeNode–>link;

# Delete(2)

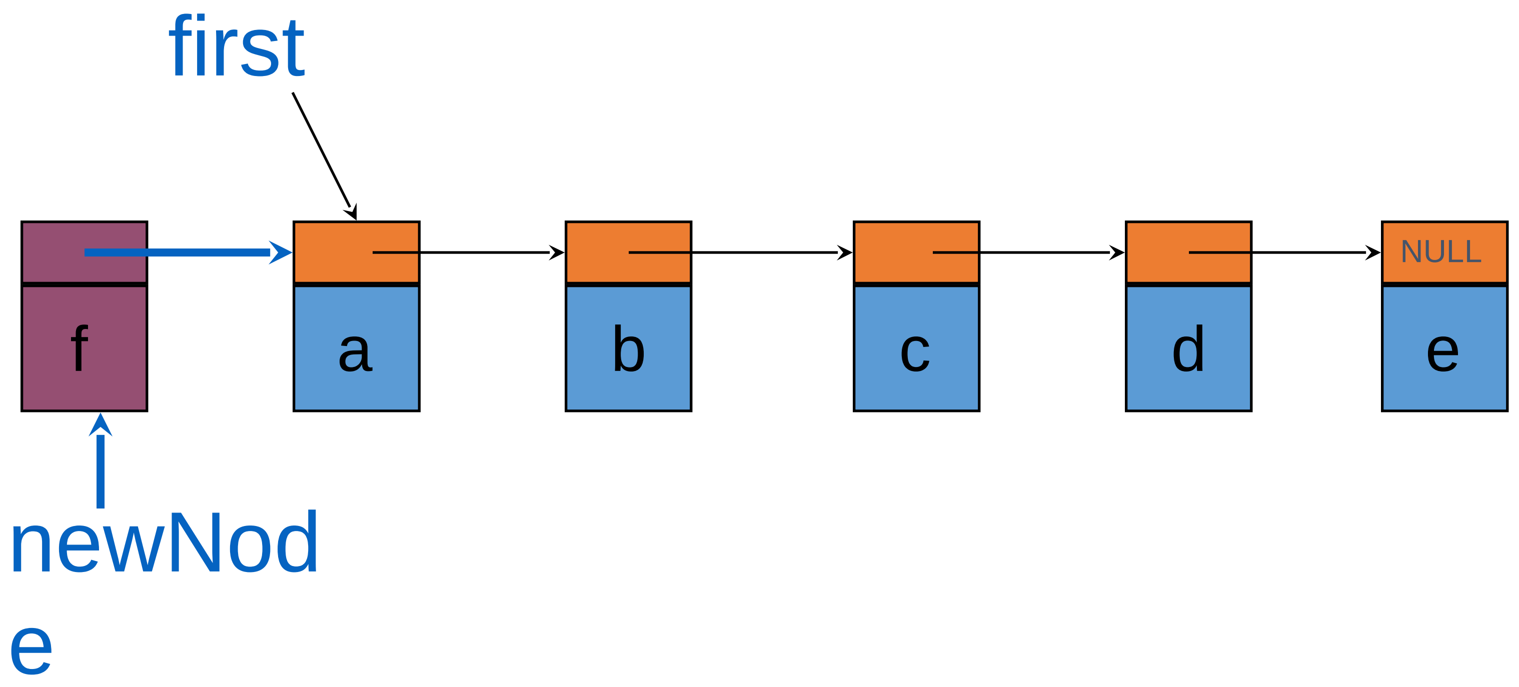


now change pointer in beforeNode

beforeNode–>link = beforeNode–>link–>link;

delete deleteNode;

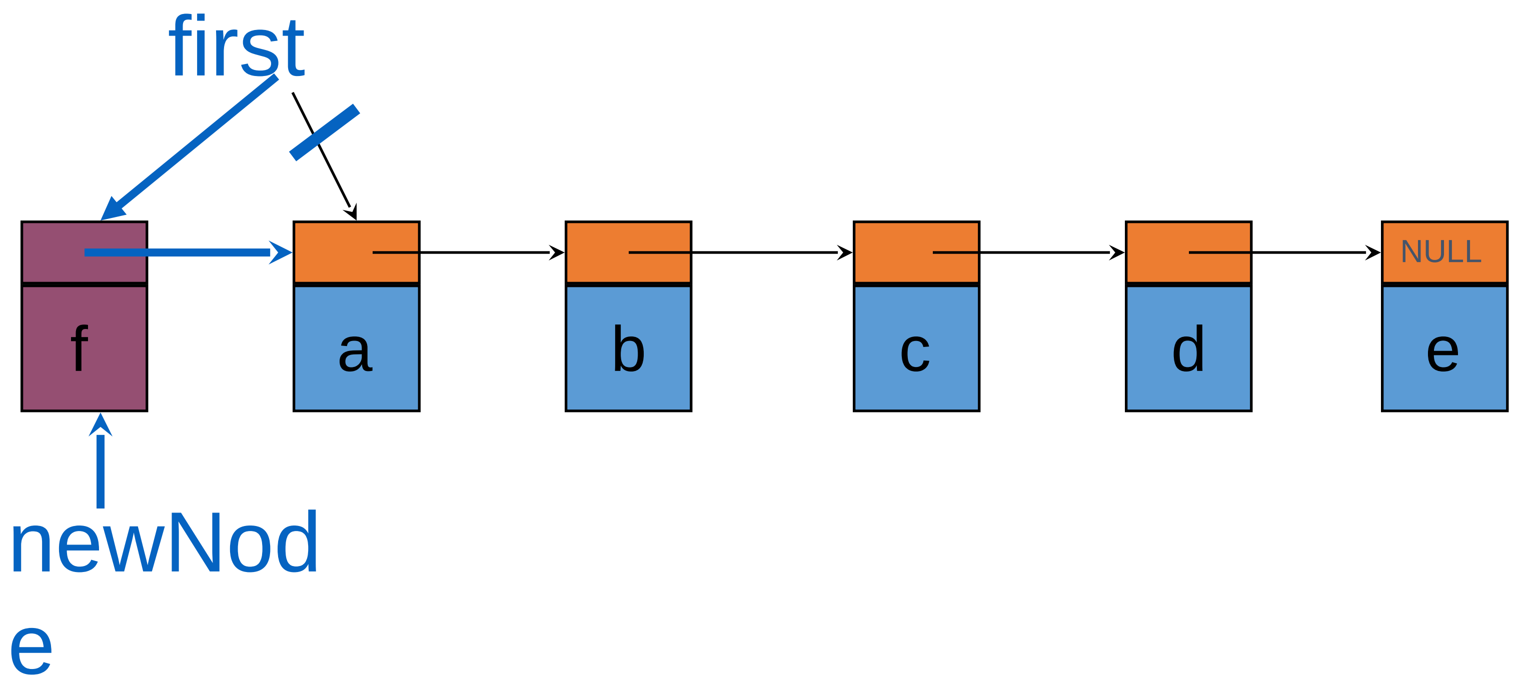
## Insert(0,’f’) : At beginning



Step 1: get a node, set its data and link fields

newNode = new ChainNode<char>(theElement,first);

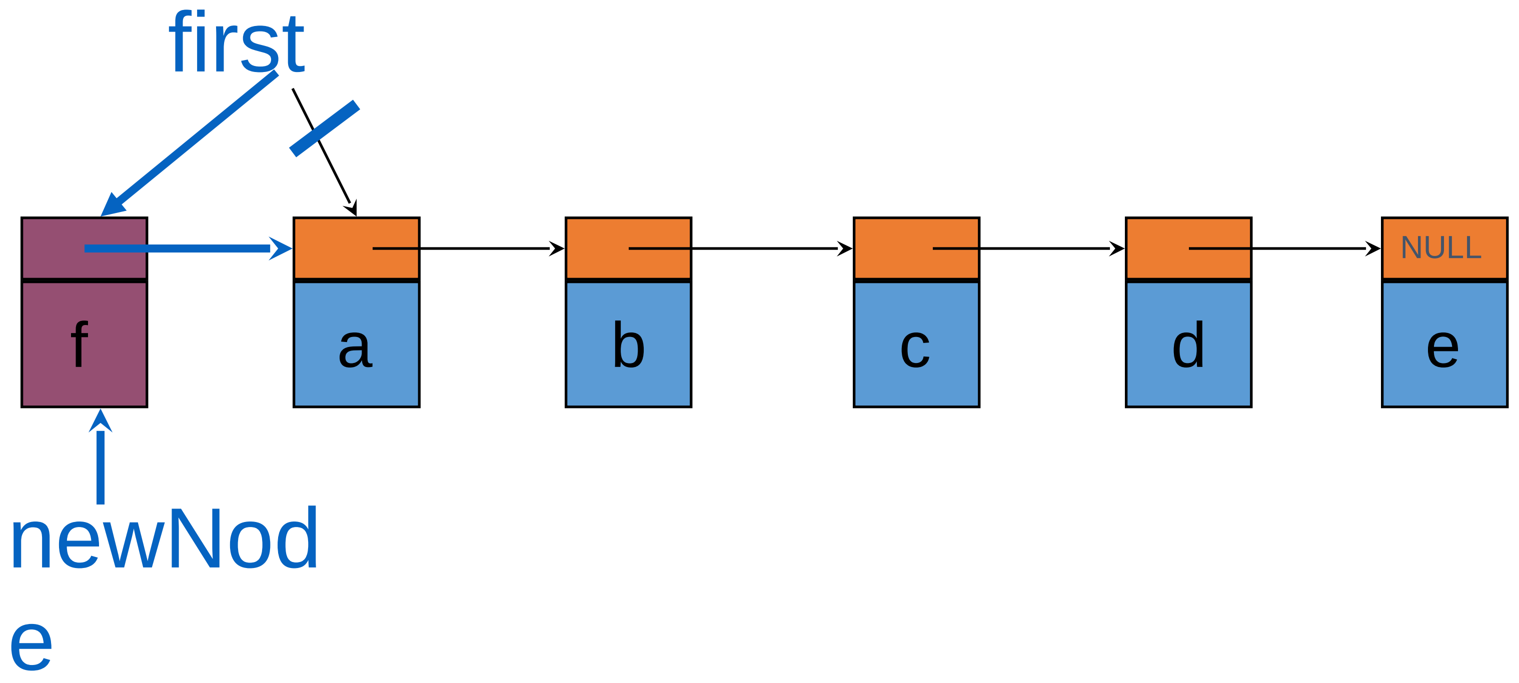
# Insert(0,’f’)



Step 2: update first

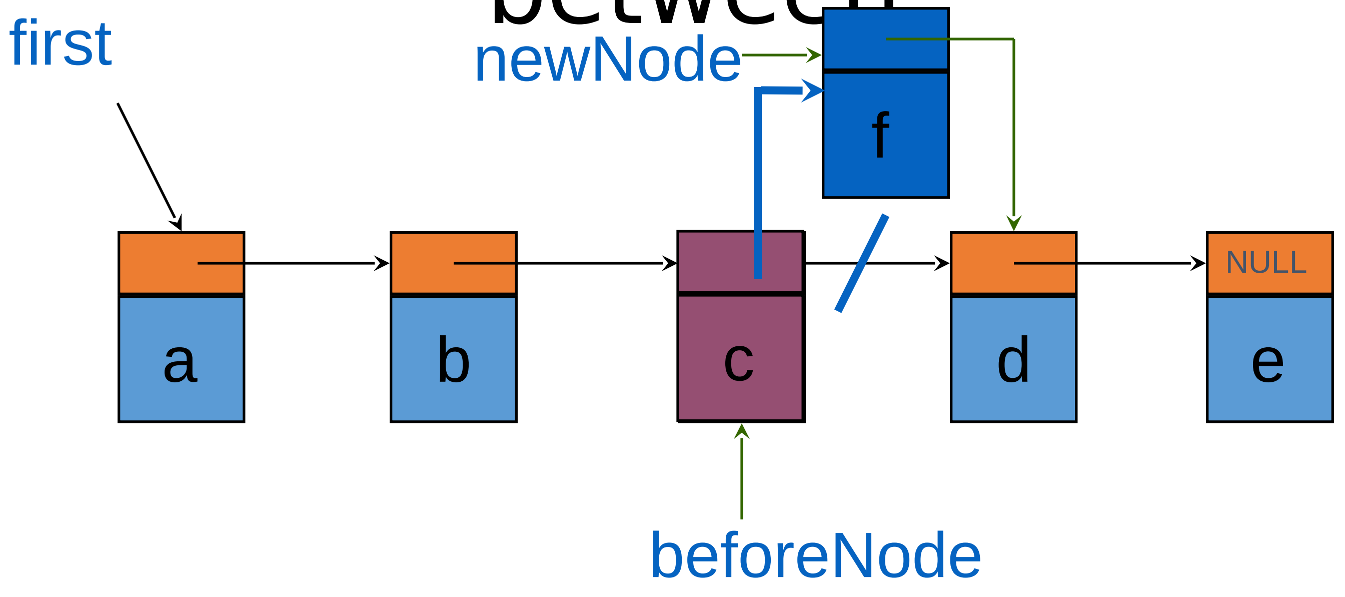
first = newNode;

# One-Step Insert(0,’f’)



```
first = new chainNode<char>(‘f’, first);
```

## Insert(3,’f’) : Anywhere in between



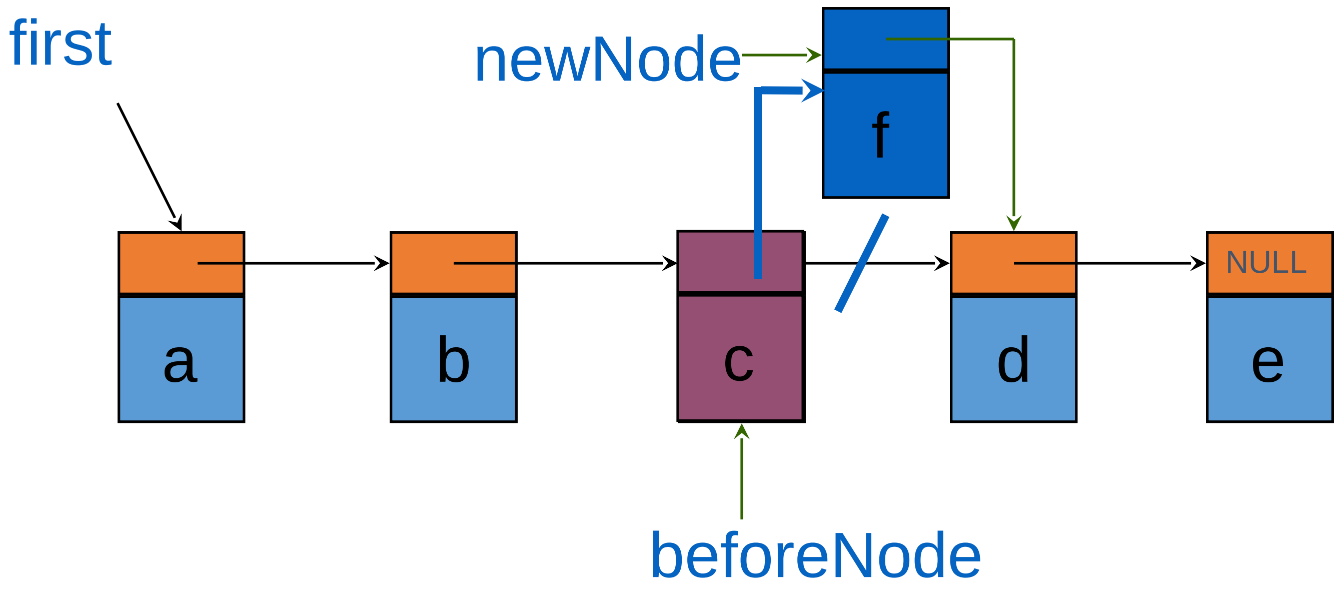
- first find node whose index is 2
- next create a node and set its data and link fields

ChainNode<char>* newNode = new ChainNode<char>( ‘f’,
beforeNode–>link);

- finally link beforeNode to newNode

beforeNode–>link = newNode;

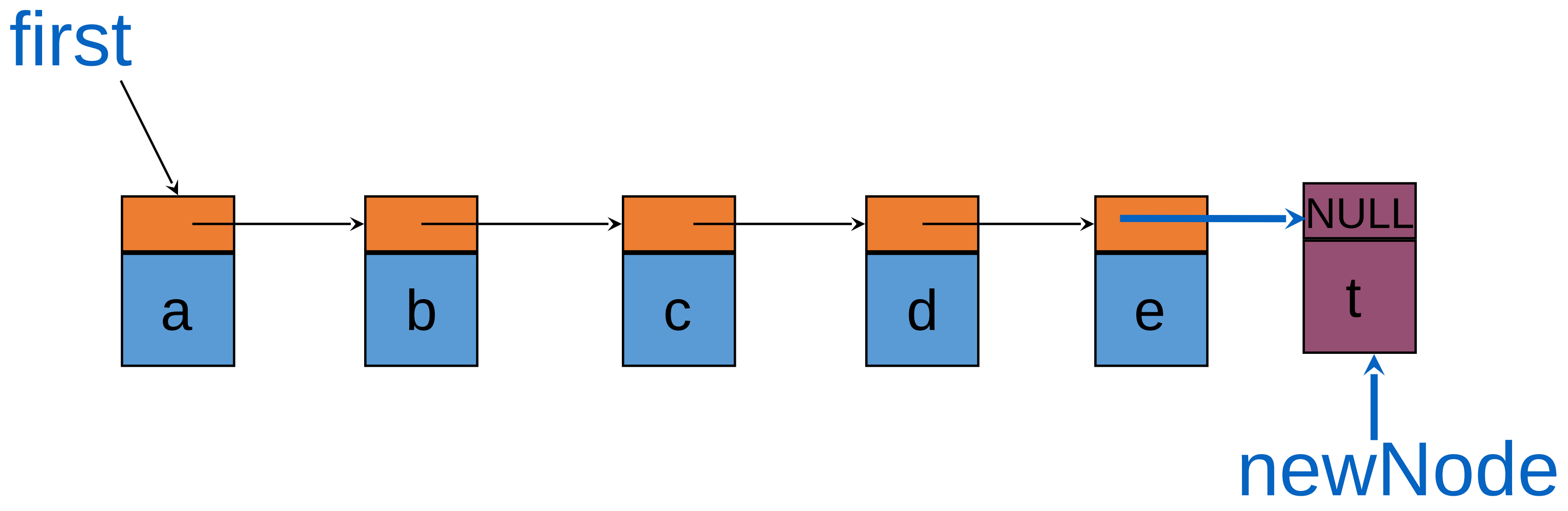
# Two-Step Insert(3,'f')



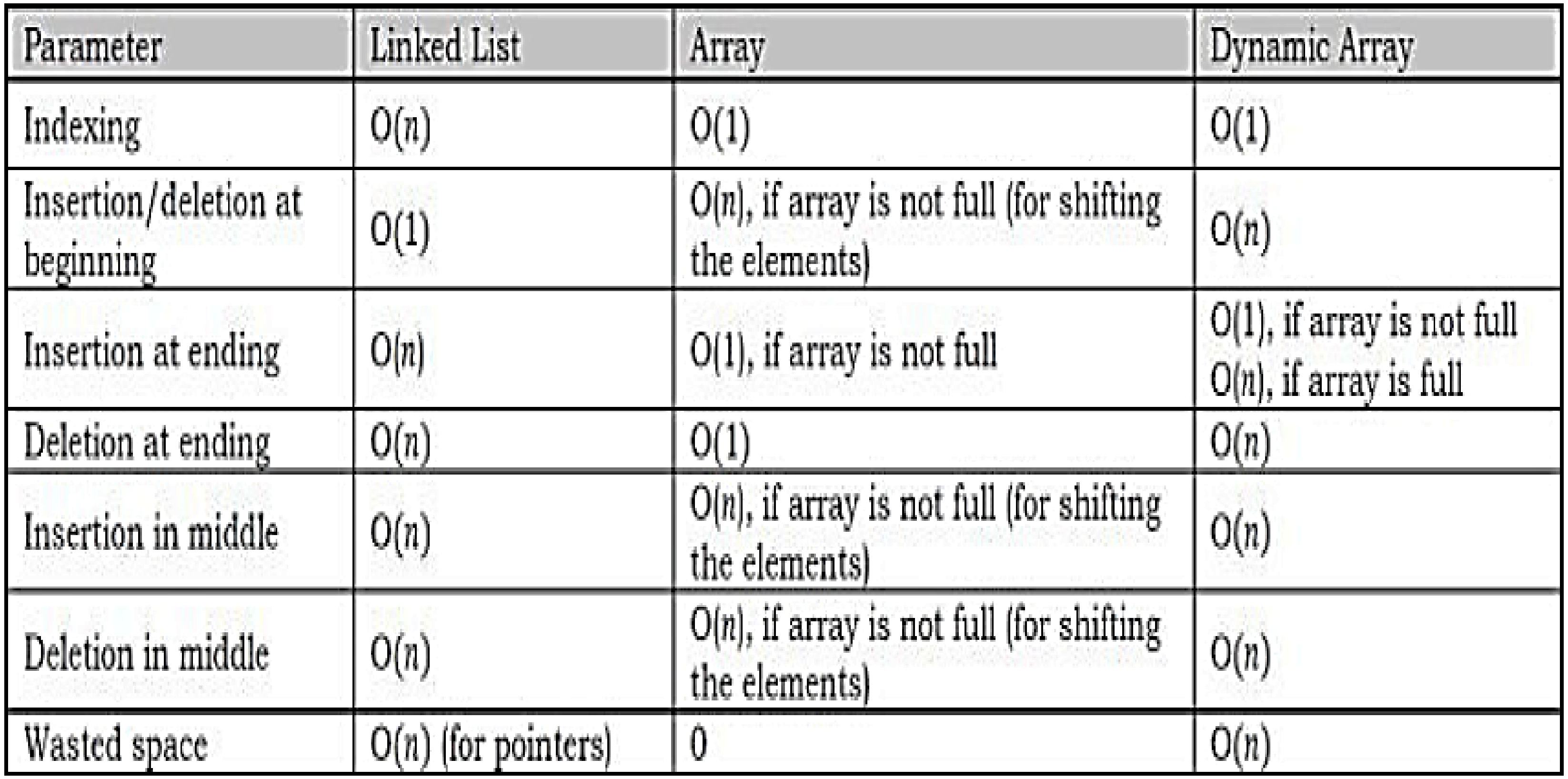
```
beforeNode = first–>link–>link;
beforeNode–>link = new ChainNode<char>
                              ('f', beforeNode–>link);
```

## Insert (5, ‘t’) : In the end

- New nodes next pointer points to NULL.



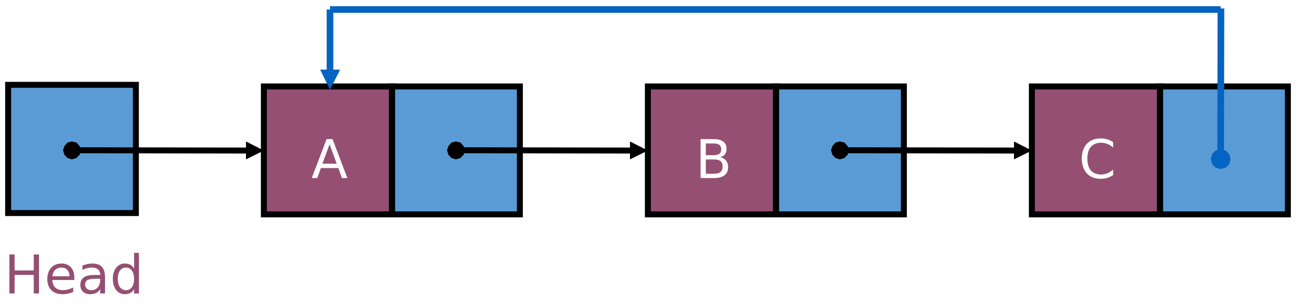
# Comparing Lists and Arrays



| Parameter | Linked List | Array | Dynamic Array |
|---|---|---|---|
| Indexing | O($n$) | O(1) | O(1) |
| Insertion/deletion at beginning | O(1) | O($n$), if array is not full (for shifting the elements) | O($n$) |
| Insertion at ending | O($n$) | O(1), if array is not full | O(1), if array is not full<br>O($n$), if array is full |
| Deletion at ending | O($n$) | O(1) | O($n$) |
| Insertion in middle | O($n$) | O($n$), if array is not full (for shifting the elements) | O($n$) |
| Deletion in middle | O($n$) | O($n$), if array is not full (for shifting the elements) | O($n$) |
| Wasted space | O($n$) (for pointers) | 0 | O($n$) |

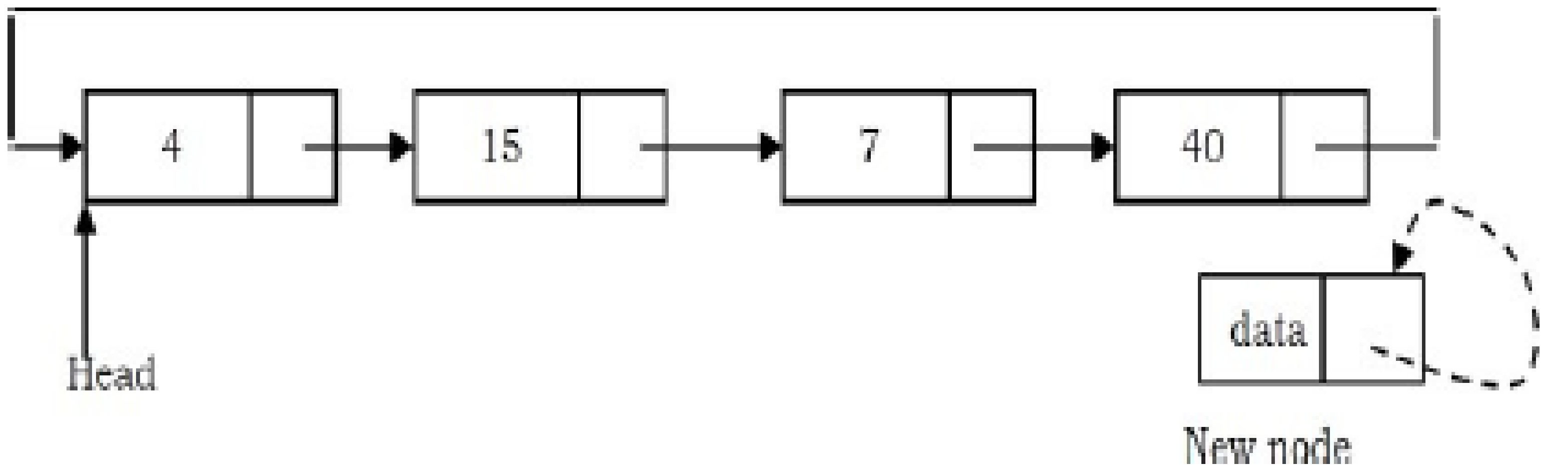
## Circular linked lists

- The last node points to the first node of the list

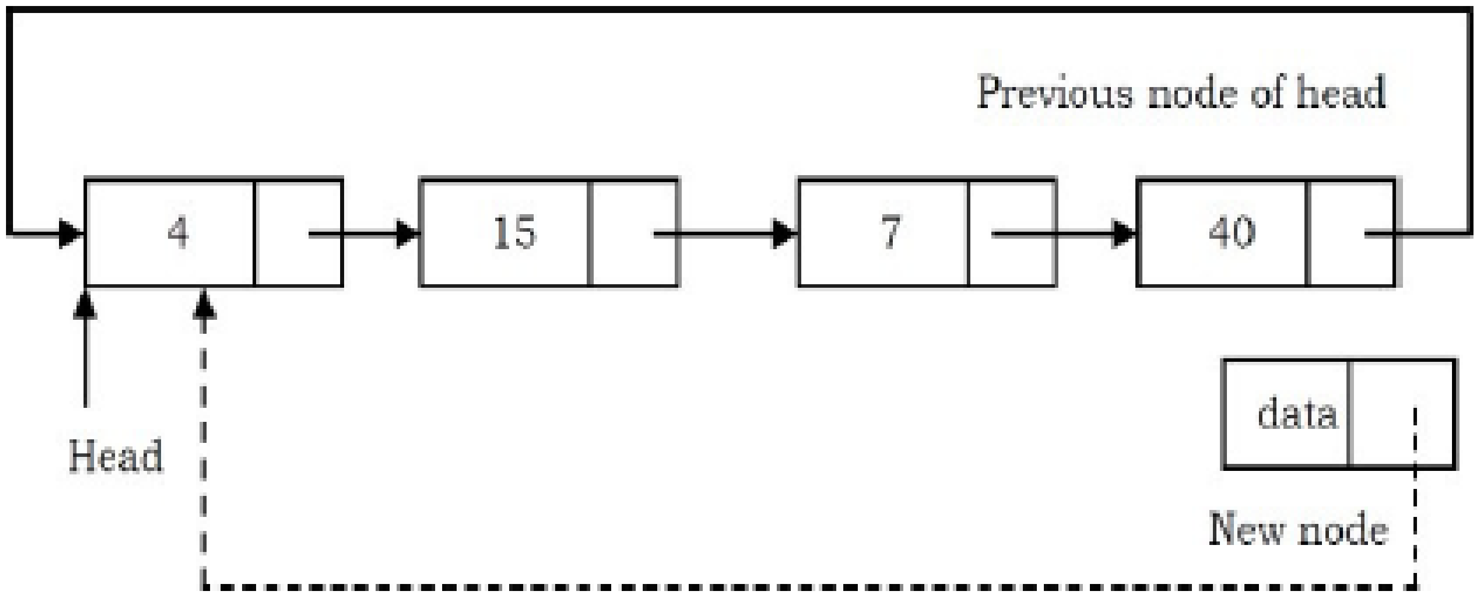
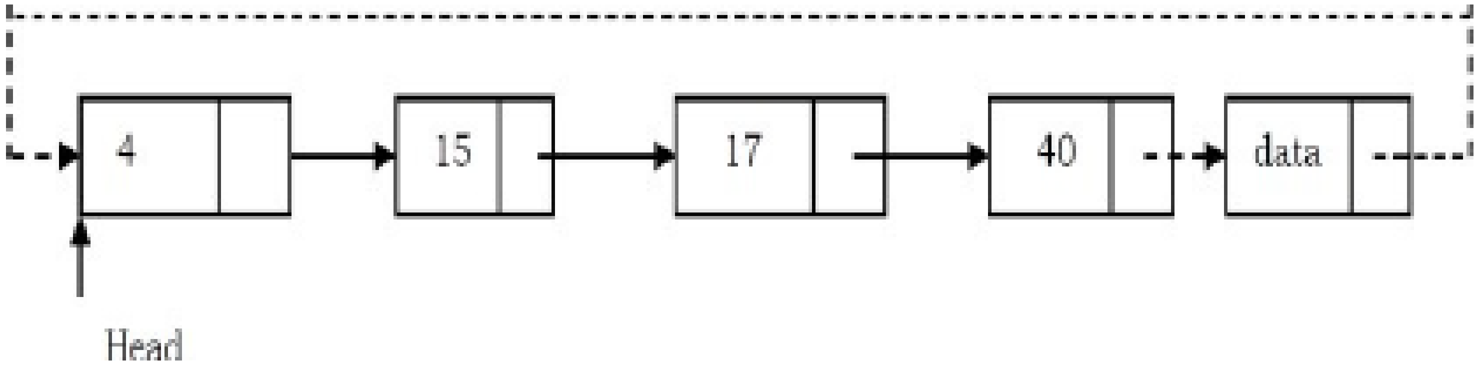


- To know when we have finished traversing the list, check if the pointer of the current

## Inserting a Node at the End of a Circular Linked List

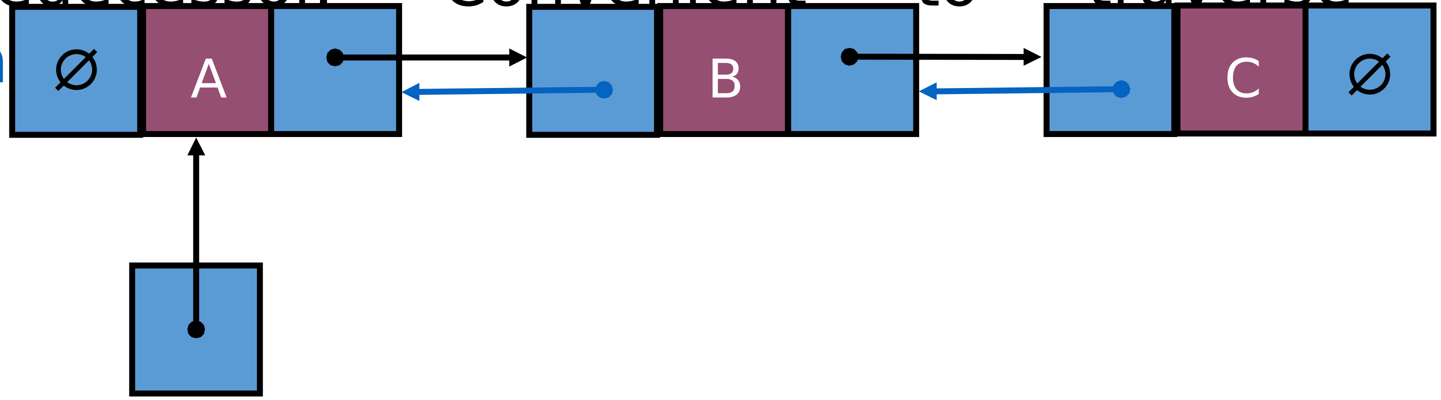


- Create a new node and initially keep its next pointer pointing to itself.
- Update the next pointer of the new node with the head node and also traverse the list to the tail. That means in a circular list we should stop at the node whose next node is head.
- Update the next pointer of the previous node to point to the new node.


Previous node of head
4
15
7
40
data
Head
New node

4
15
17
40
data
Head

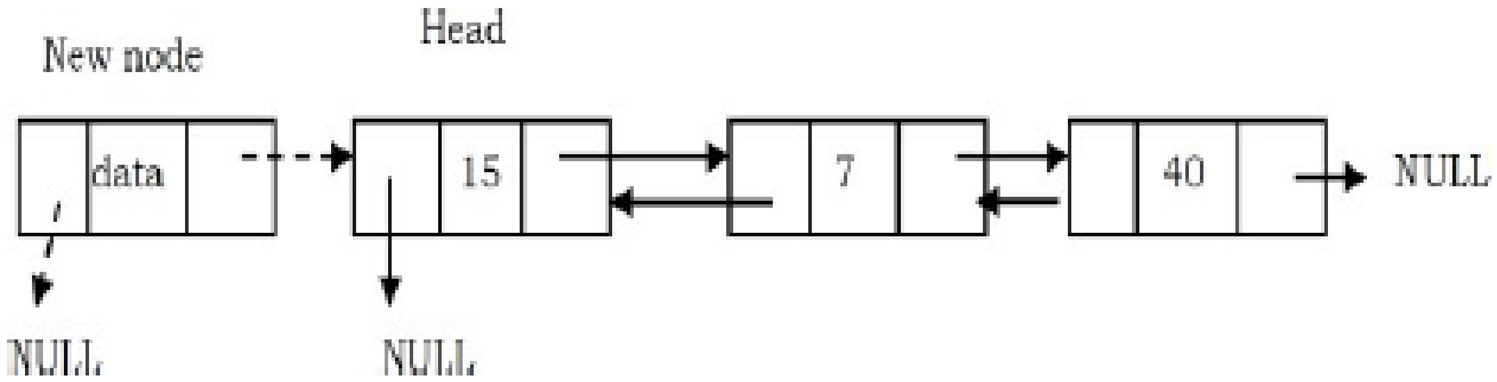
# Doubly linked lists

- Each node points to not only successor but the predecessor
- There are two NULL: at the first and last nodes in the list
- Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists ba

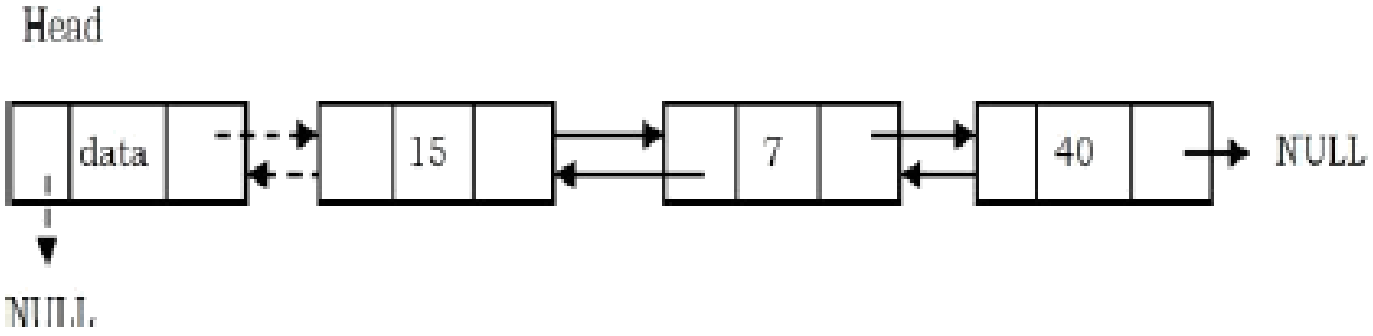


The primary disadvantages of doubly linked lists are:
• Each node requires an extra pointer, requiring more space.
• The insertion or deletion of a node takes a bit longer

# Inserting a Node in Doubly Linked List at the Beginning



Update the right pointer of the new node to point to the current head node and also make left pointer of new node as NULL.
Update head node’s left pointer to point to the new node and make new node as head. Head

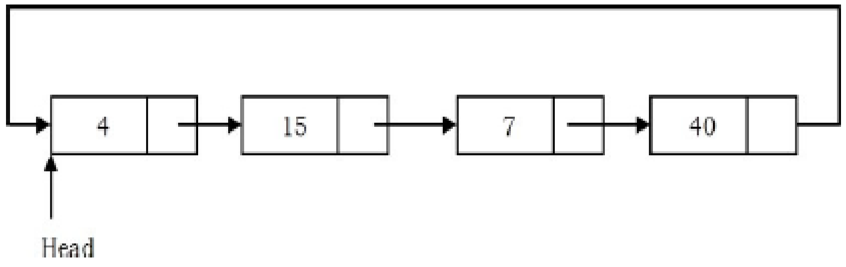


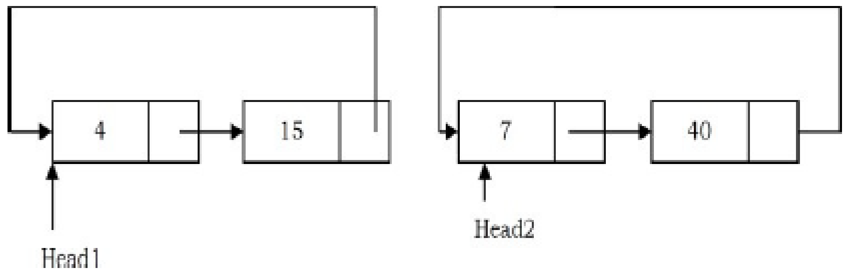
# Exercise

Split a Circular Linked List into two equal parts. If the number of nodes in the list are odd then make first list one node extra than second list.

Solution – Algorithm (Time Complexity: O(*n*))

- Store the mid and last pointers of the circular linked list
- Make the second half circular.
- Make the first half circular.
- Set head pointers of the two linked lists.





# Exercise

Check if the linked list is palindrome or not

**Algorithm:**

1. Get the middle of the linked list.
2. Reverse the second half of the linked list.
3. Compare the first half and second half.
4. Construct the original linked list by reversing the second half again and attaching it back to the first half.

Time Complexity: O($n$).

## Applications of Linked Lists

- Bin Sort
- Radix Sort
- Convex Hull