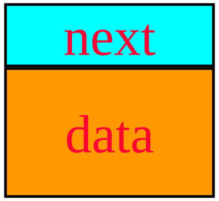
The Class LinkedList

Node representation

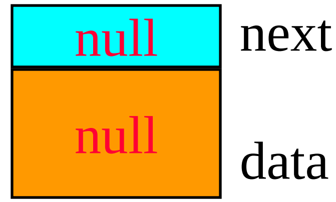
```
class Node
{
  // data members
  Object data;  // data field
  Node next;  // link field

  // constructors come here
}
```



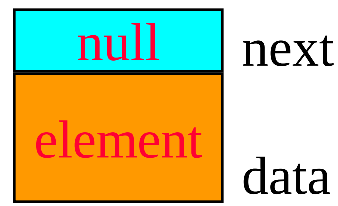
Constructors of LinkedList Node

Node() {}



Node(Object element)

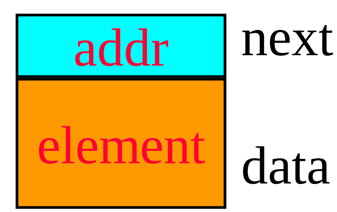
{this.data= element;}



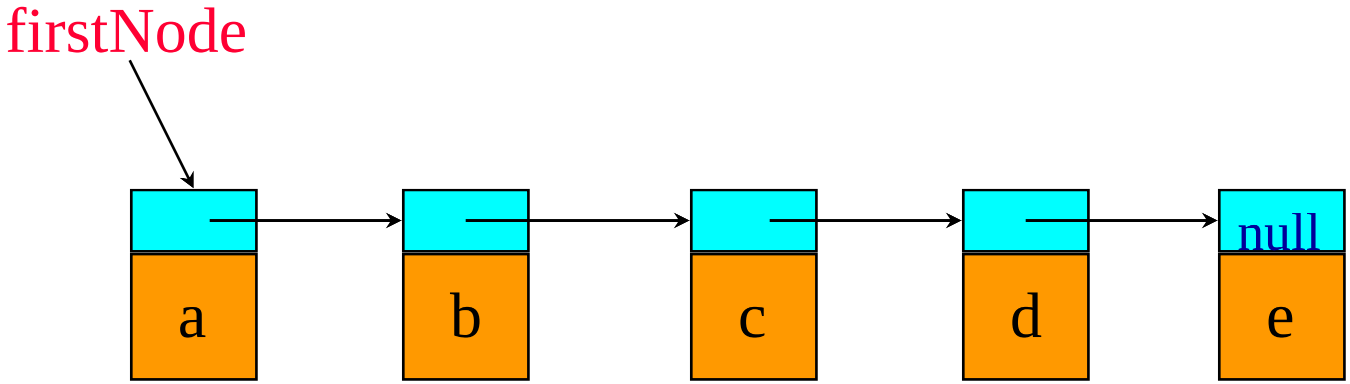
Node(Object element, Node addr)

{this.data = element;

this.next = addr;}

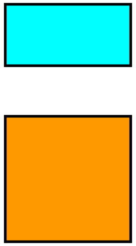


The Class LinkedList



size = number of elements

Use Node



next (datatype Node)

data (datatype Object)

The Class LinkedList

```
/** linked implementation of LinearList */

public class LinkedList  ~~implements LinearList~~
{
  // data members
  protected Node firstNode;
  protected int size;

  // constructors and methods of LinkedList come here
}
```

Constructors

```
/** create a list that is empty */
public LinkedList(int initialCapacity)
{
        // the default initial value of firstNode is null, and
        // size is 0
}

public LinkedList()
 { firstNode=null;
   size=0;
   }
```

The Method isEmpty

```
/** @return true iff list is empty */
public boolean isEmpty()
   {return size == 0;}
```

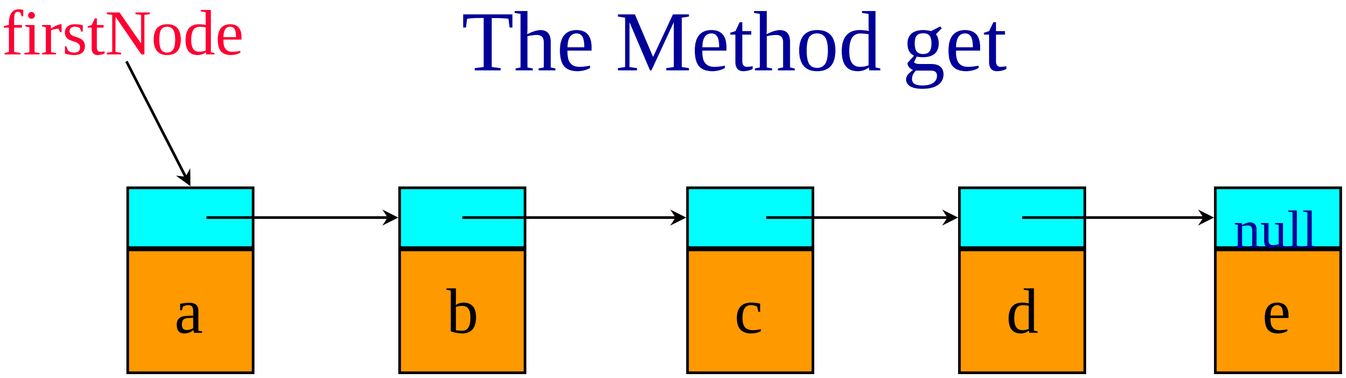
The Method size()

```
/** @return current number of elements in list */
public int size()
   {return size;}
```

The Method checkIndex

```
/** @throws IndexOutOfBoundsException when
  * index is not between 0 and size - 1 */
void checkIndex(int index)
{
   if (index < 0 || index >= size)
      throw new IndexOutOfBoundsException
            ("index = " + index + "  size = " + size);
}
```

The Method get




```
public Object get(int index)
{
   checkIndex(index);

   // move to desired node
   Node  currentNode = firstNode;
   for (int i = 0; i < index; i++)
      currentNode = currentNode.next;
  return currentNode.data;
}
```

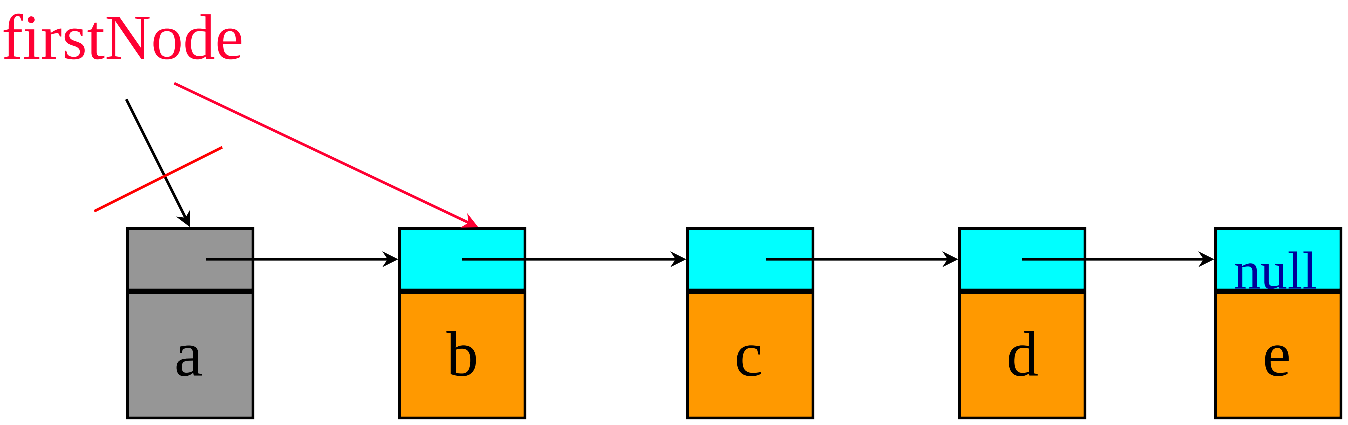
The Method indexOf

```
public int indexOf(Object theElement)
 {
   // search the LinkedList for theElement
   Node currentNode = firstNode;
   int index = 0;  // index of currentNode
   while (currentNode != null &&
         !currentNode.data.equals(theElement))
   {
     // move to next node
     currentNode = currentNode.next;
     index++;
   }
```

The Method indexOf

```
    // make sure we found matching element
    if (currentNode == null)
      return -1;
    else
      return index;
}
```

Removing An Element



remove(0) // remove first node

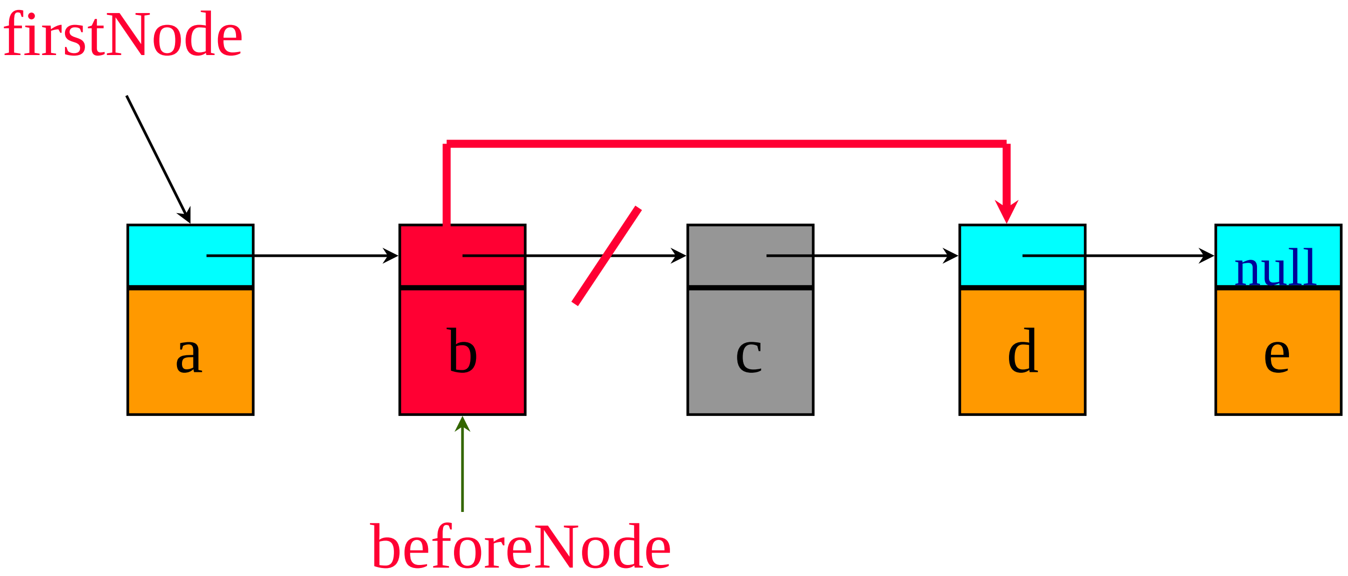
firstNode = firstNode.next;

Remove An Element

```
public Object remove(int index)
{
  checkIndex(index);
  Object removedElement;

  if (index == 0) // remove first node
  {
    removedElement = firstNode.data;
    firstNode = firstNode.next;
  }
  else ……?
}
```

remove(2)



Find beforeNode and change its pointer.

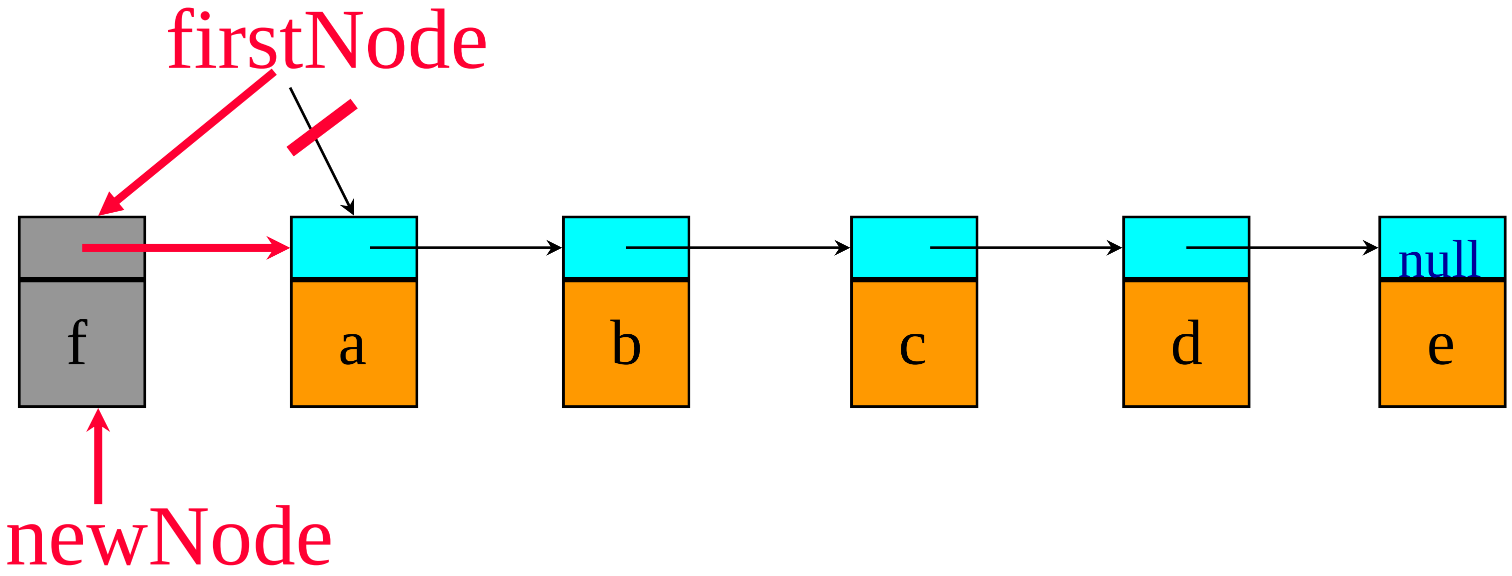
beforeNode.next = beforeNode.next.next;

Remove An Element

```
else
  {  // use q to get to beforeNode of desired node
     Node q = firstNode;
     for (int i = 0; i ≤ index - 2; i++)
        q = q.next;

     removedElement = q.next.data;
     q.next = q.next.next; // remove desired node
  }
  size--;
  return removedElement;
}
```

One-Step add(0,’f’)



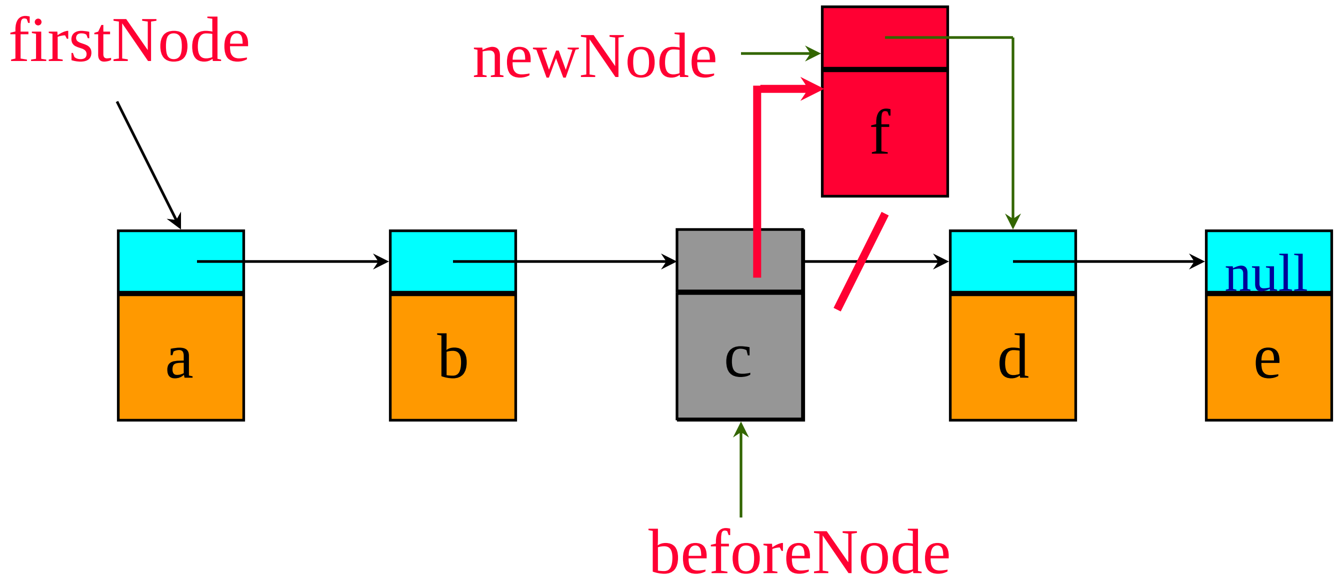
firstNode = new Node(‘f’, firstNode);

Add an element

```
public void add(int index, Object theElement)
 {
   if (index < 0 || index > size)
     // invalid list position
     throw new IndexOutOfBoundsException
          ("index = " + index + "  size = " + size);

   if (index == 0)
     // insert at front
     firstNode = new Node(theElement, firstNode);
     else ……?
  }
```

Two-Step add(3,’f’)



beforeNode = firstNode.next.next; // find beforeNode

beforeNode.next = new Node(‘f’, beforeNode.next);

OR

~~beforeNode.next = newNode;~~

~~newNode.next = beforeNode.next ;~~ // Is this correct

Adding an element

```
else
   { // find beforeNode of new element
     Node p = firstNode;
     for (int i = 0; i ≤ index - 2; i++)
        p = p.next;

     // insert after p
     p.next = new Node(theElement, p.next);
   }
   size++;
 }
```

Linked Lists

<u>Advantages</u>

Quick insertion –O(1)

Dynamic size

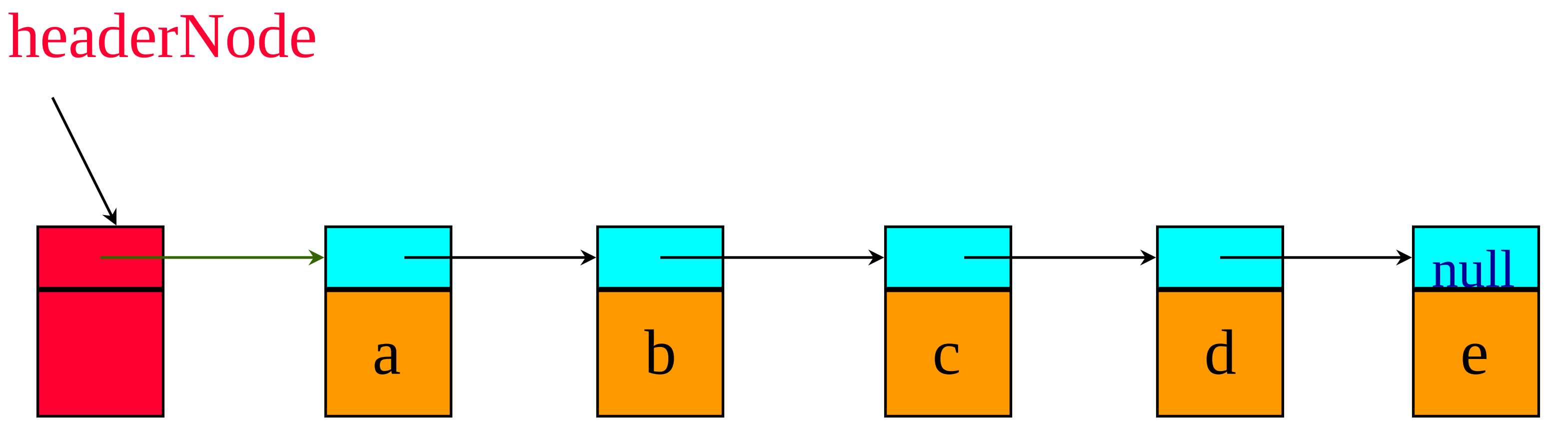
<u>Disadvantages</u>

Slow search - O(n)

Slow deletion - O(n)

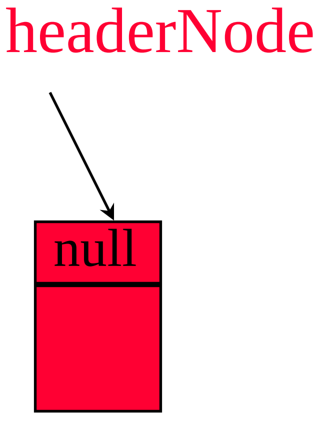
	Advantages	**Disadvantages**
Unordered array list	Insertion - O(1)	Search – O(n) Deletion – O(n) Fixed array size
Ordered array list	search –O($\log_2 n$)	Insertion - O(n) Deletion - O(n) Fixed array size
Linked list	Insertion - O(1) **Dynamic size**	Search – O(n) Deletion – O(n)

Linked List With Header Node

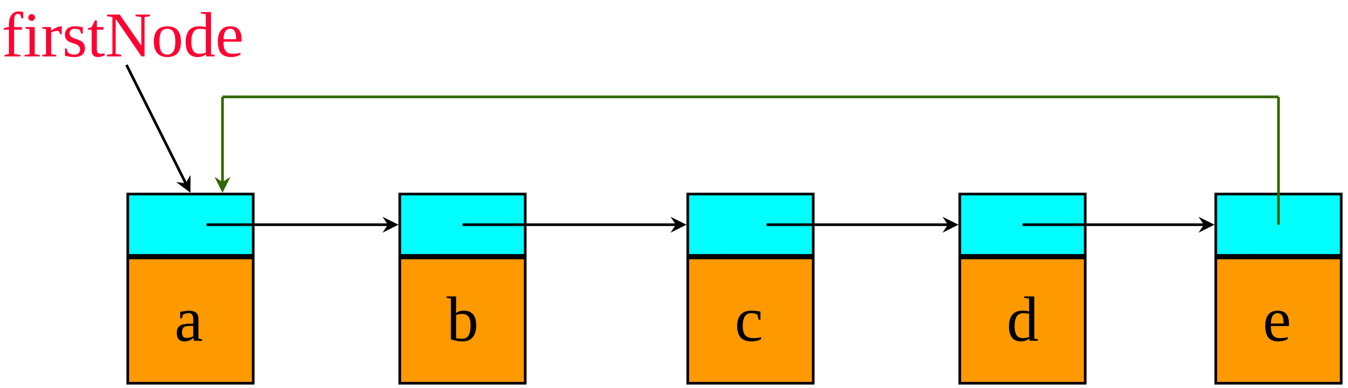


Now add/remove at left end (i.e., index = 0) are no different from any other add/remove. So add/remove code is simplified.

Empty Linked List With Header Node



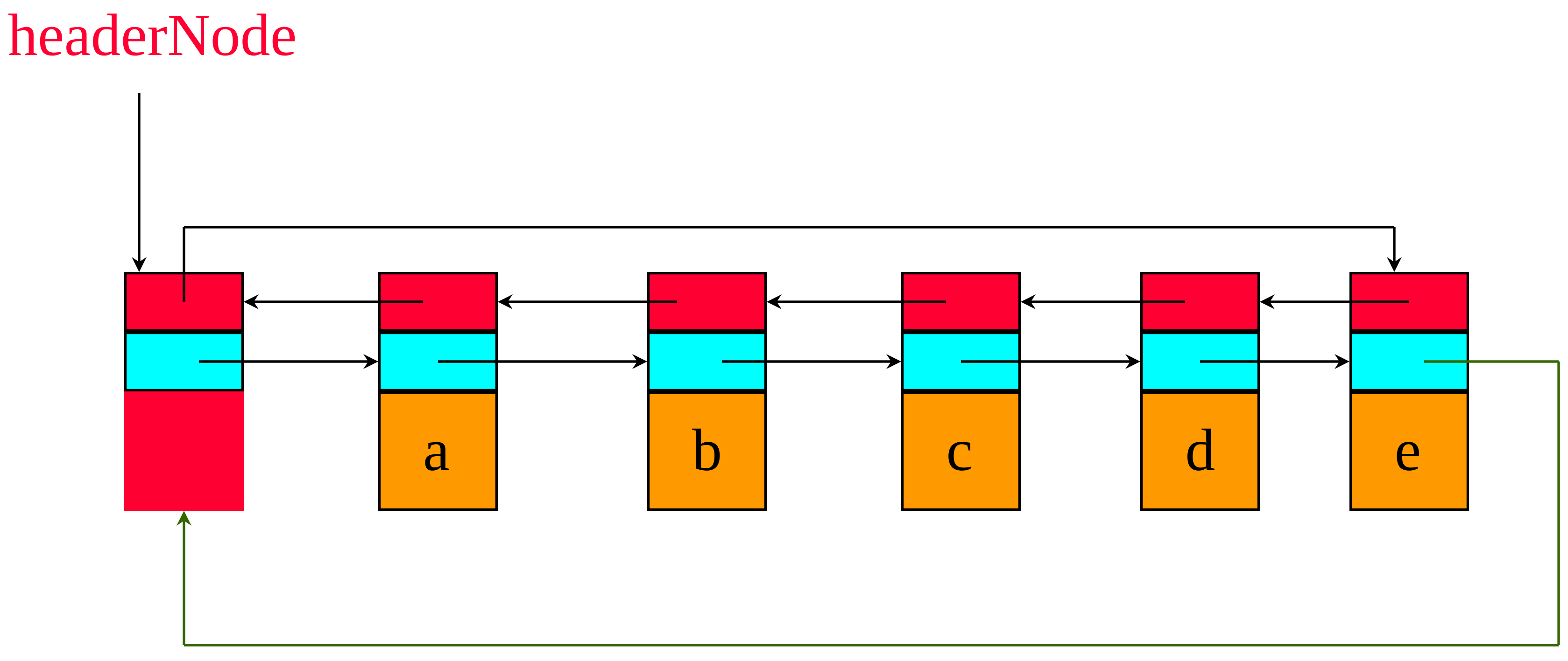
Circular List



Useful, for example, when each node represents a supplier and you want each supplier to be called on in round-robin order.

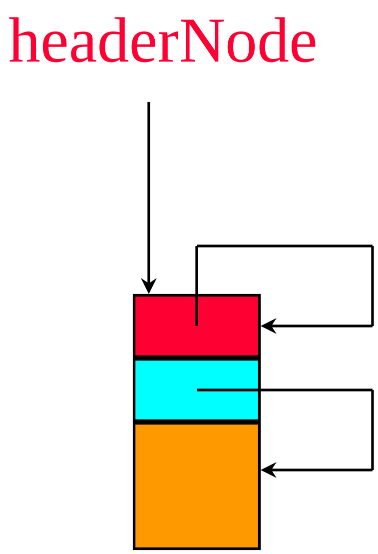
A variant of this has a header node—circular list with header node.

Doubly Linked Circular List With Header Node



It is often useful to have a last node pointer rather than a first node pointer. By doing this, additions to the front and end become O(1).

Empty Doubly Linked Circular List With Header Node



java.util.LinkedList

- Linked implementation of a linear list.
- Doubly linked circular list with header node.