# Linear Lists – Linked Representation

Data structures

Fall 2018

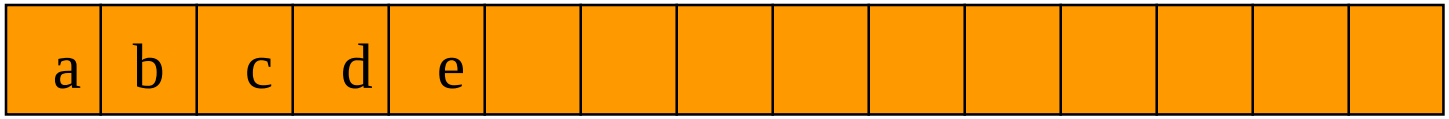
# Linked Representation

- list elements are stored, in memory, in an arbitrary order

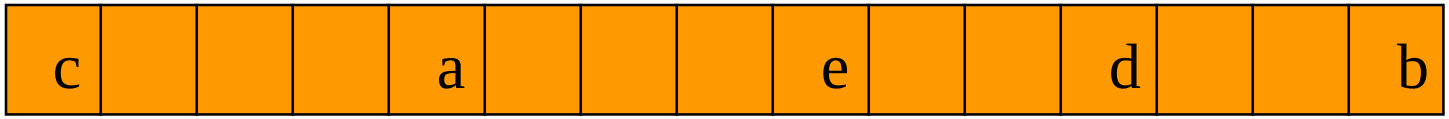
- explicit information (called a ‘link’) is used to go from one element to the next

# Memory Layout

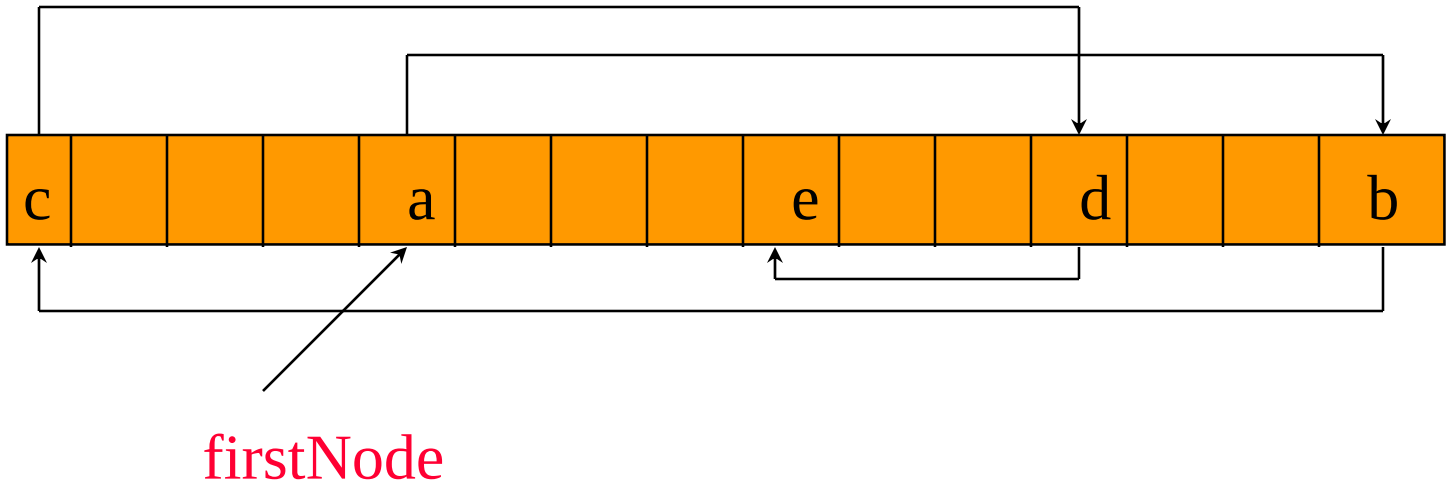
Layout of L = (a,b,c,d,e) using an array representation.




A linked representation uses an arbitrary layout.



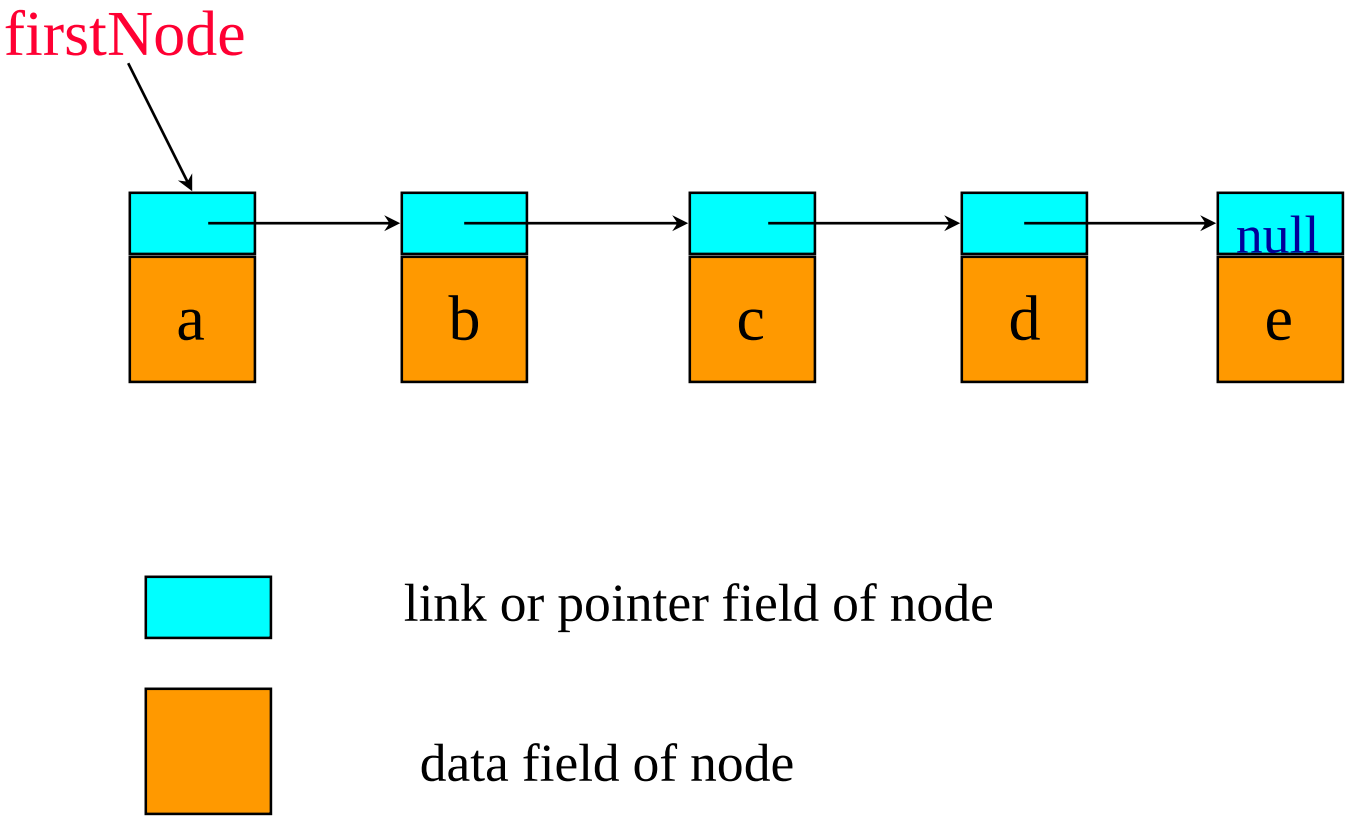
# Linked Representation



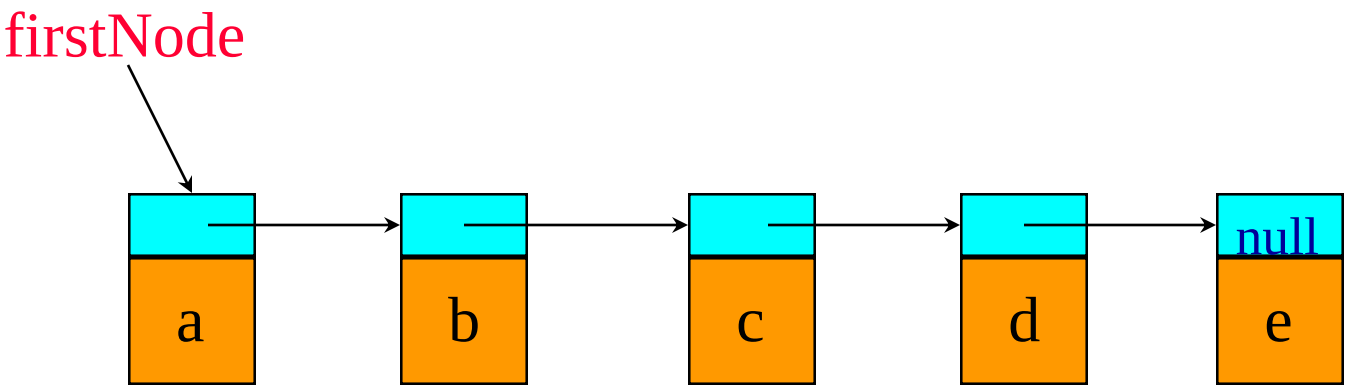

pointer (or link) in e is null

use a variable firstNode to get to the first element a

# Normal Way To Draw A Linked List



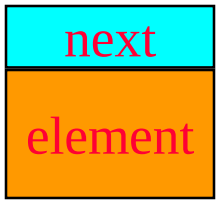
# Chain




- 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 pointer.

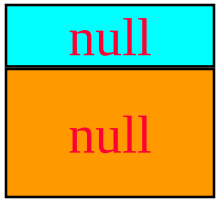
# Node Representation

```
package dataStructures;

class ChainNode
{
   // package visible data members
   Object element;
   ChainNode next;

   // constructors come here
}
```

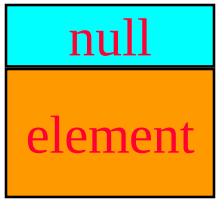


# Constructors Of ChainNode

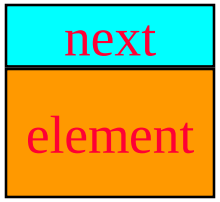
ChainNode() {}



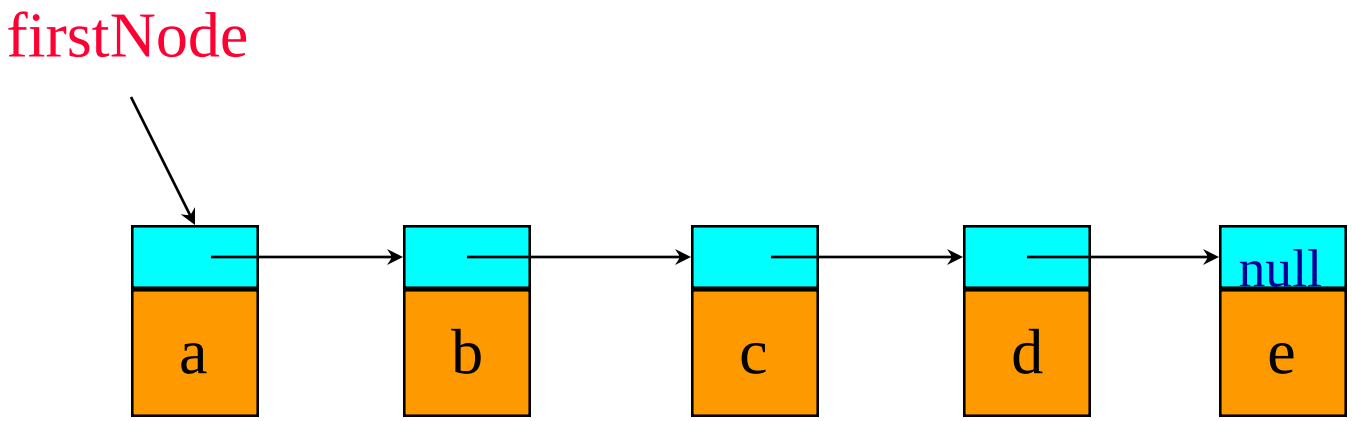

ChainNode(Object element)
{this.element = element;}




ChainNode(Object element, ChainNode next)
{this.element = element;
this.next = next;}

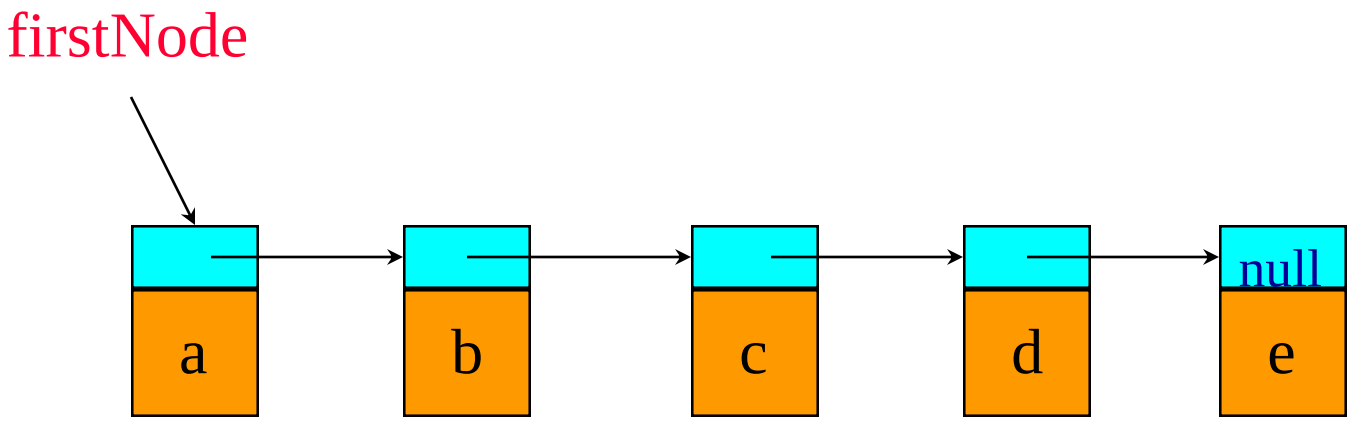


# get(0)




```
checkIndex(0);
desiredNode = firstNode; // gets you to first node
return desiredNode.element;
```

# get(1)

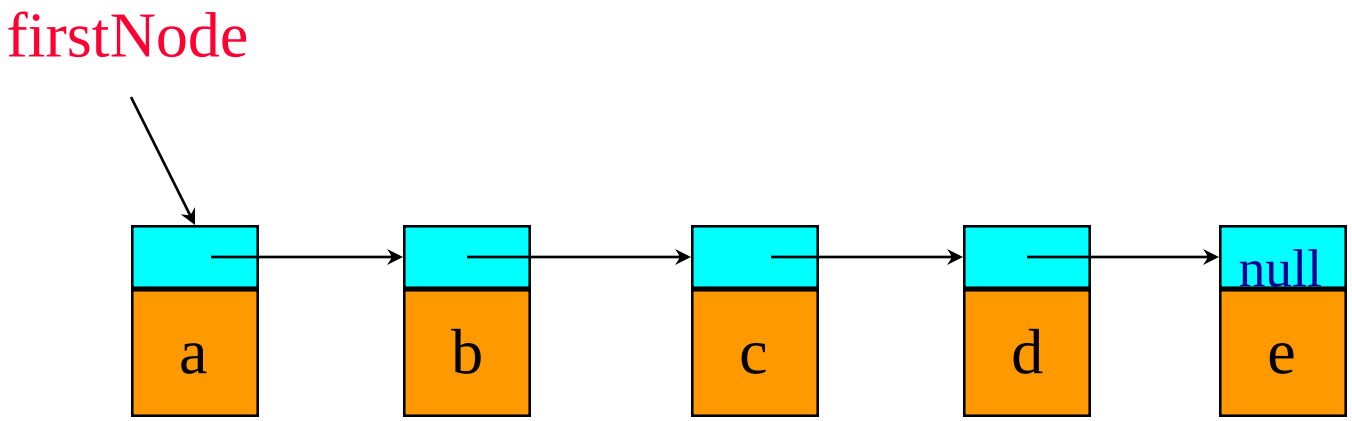



checkIndex(1);

desiredNode = firstNode.next; // gets you to second node

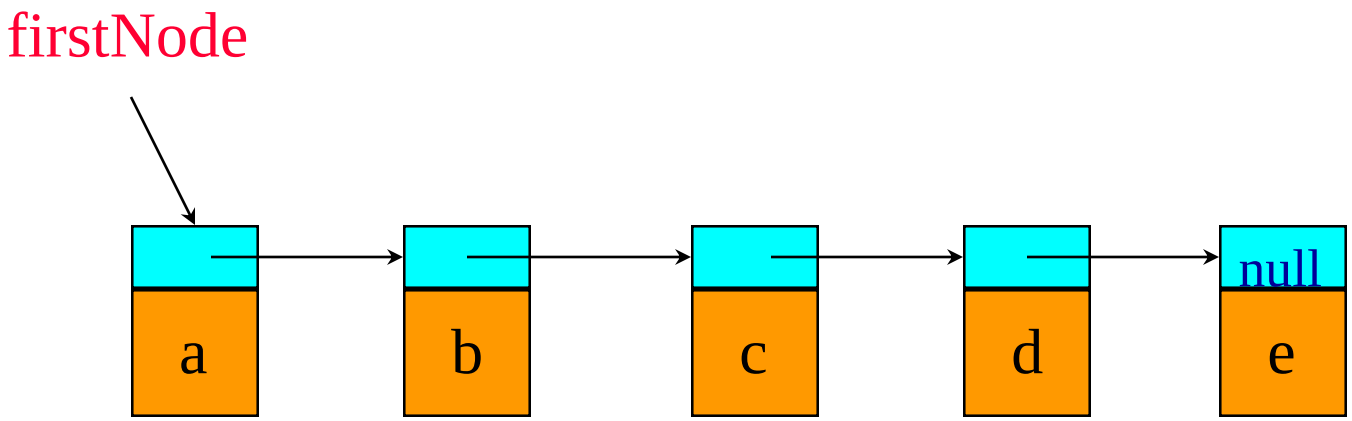
return desiredNode.element;

# get(2)



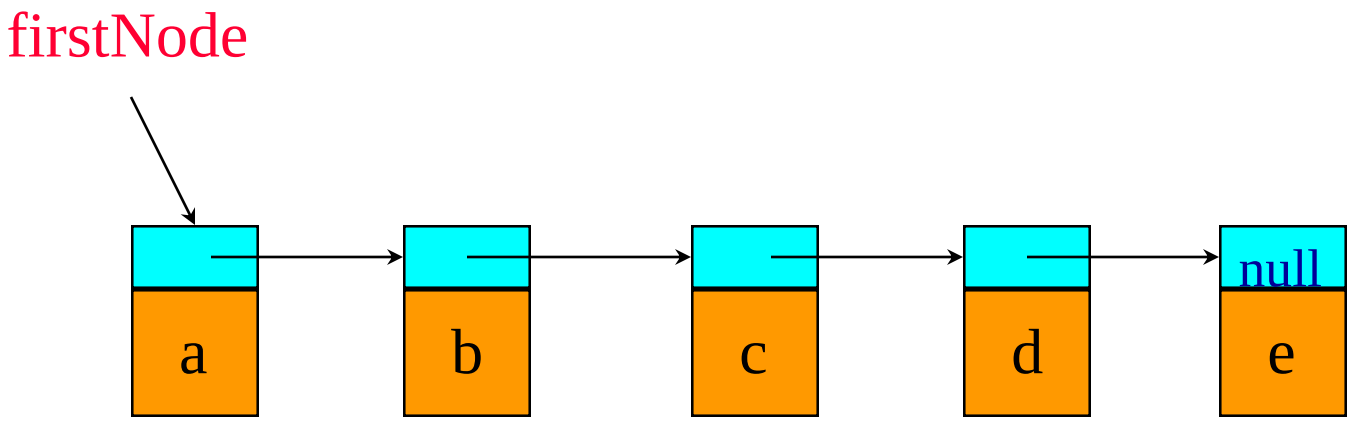

```
checkIndex(2);
desiredNode = firstNode.next.next; // gets you to third node
return desiredNode.element;
```

# get(5)



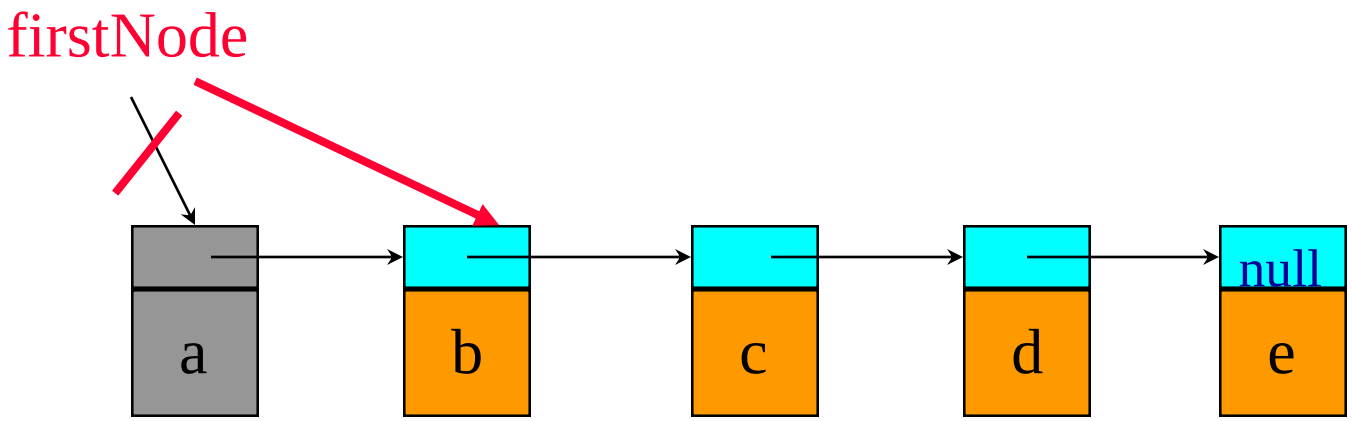

```
checkIndex(5);              // throws exception
desiredNode = firstNode.next.next.next.next.next;
                            // desiredNode = null
return desiredNode.element;   // null.element
```

# NullPointerException




```
desiredNode =
    firstNode.next.next.next.next.next.next;
      // you get a NullPointerException
```

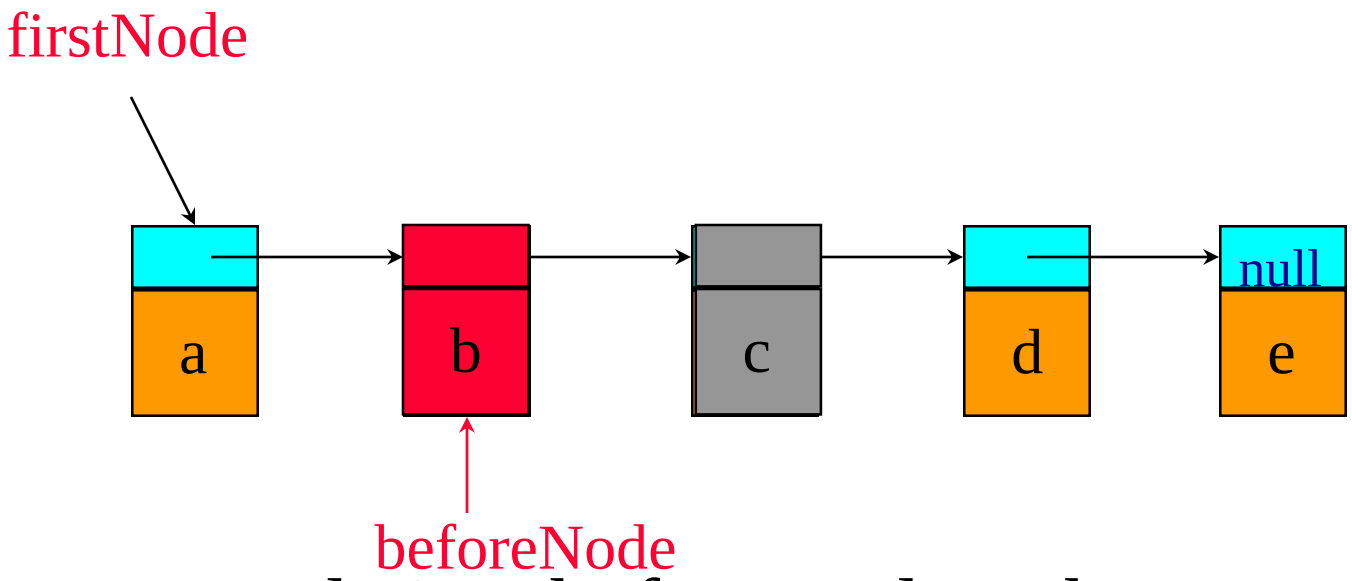
# Remove An Element




remove(0):

firstNode = firstNode.next;

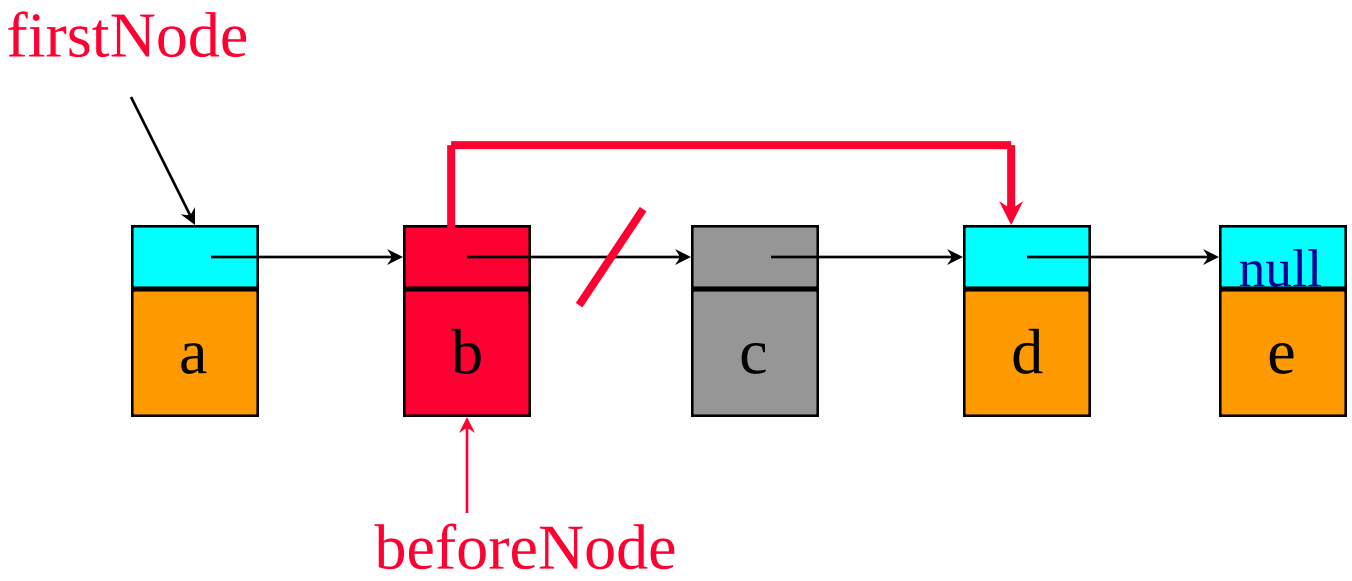
# remove(2)




first get to node just before node to be removed

beforeNode = firstNode.next;

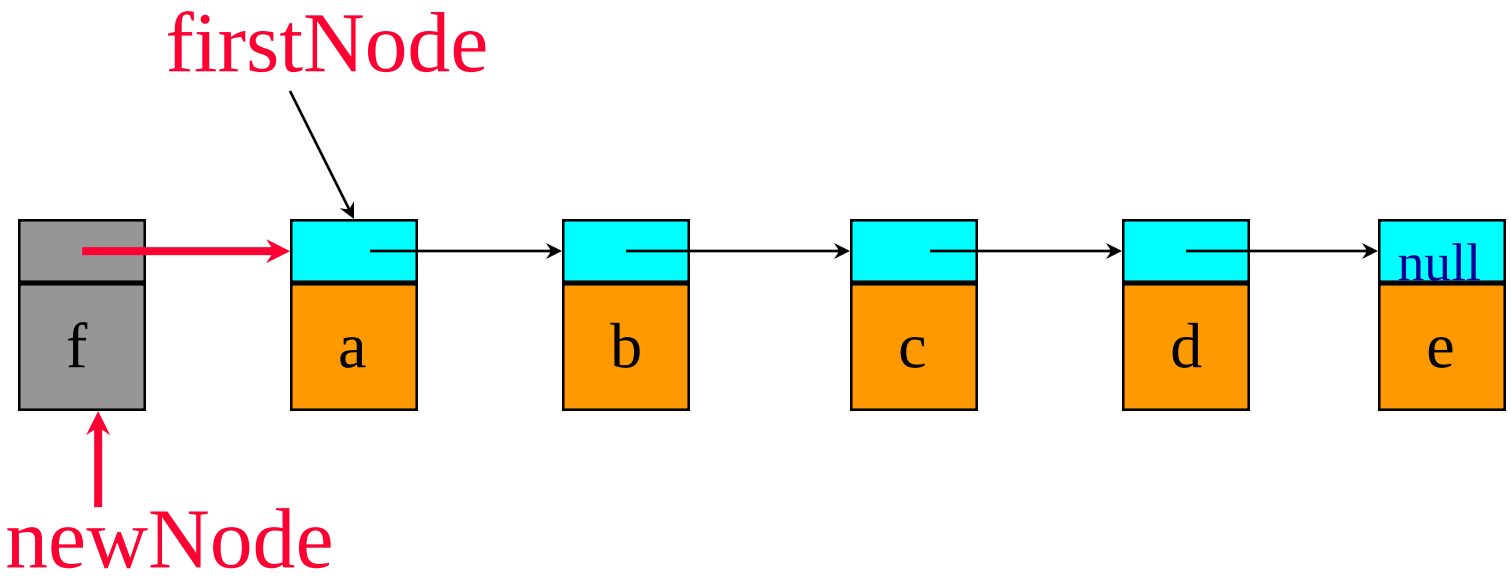
# remove(2)




now change pointer in beforeNode

beforeNode.next = beforeNode.next.next;

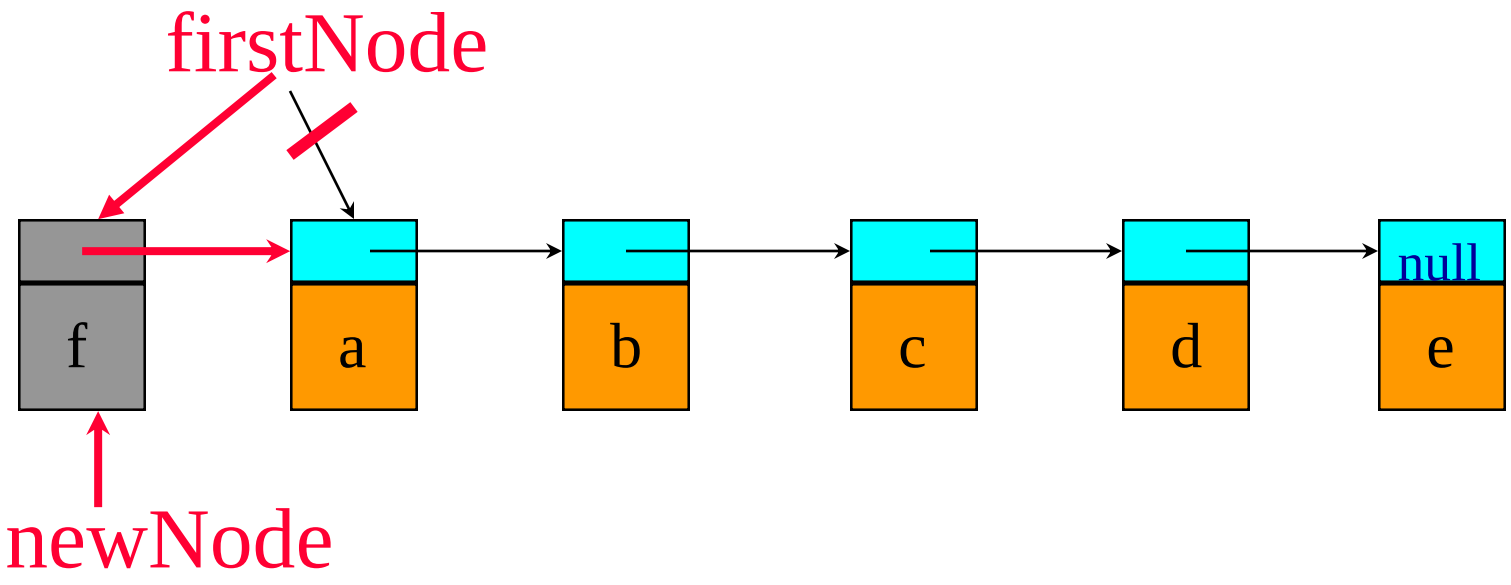
# add(0,'f')




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

```
ChainNode newNode =
    new ChainNode(new Character('f'), firstNode);
```

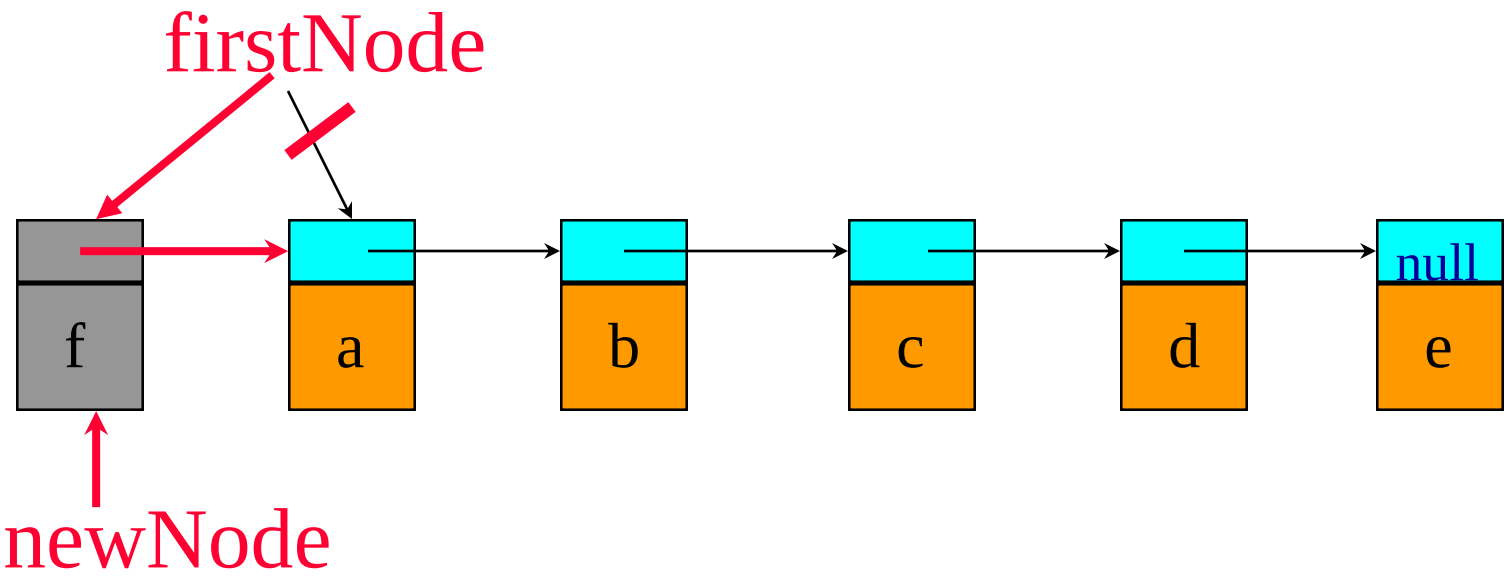
# add(0,'f')




Step 2: update firstNode

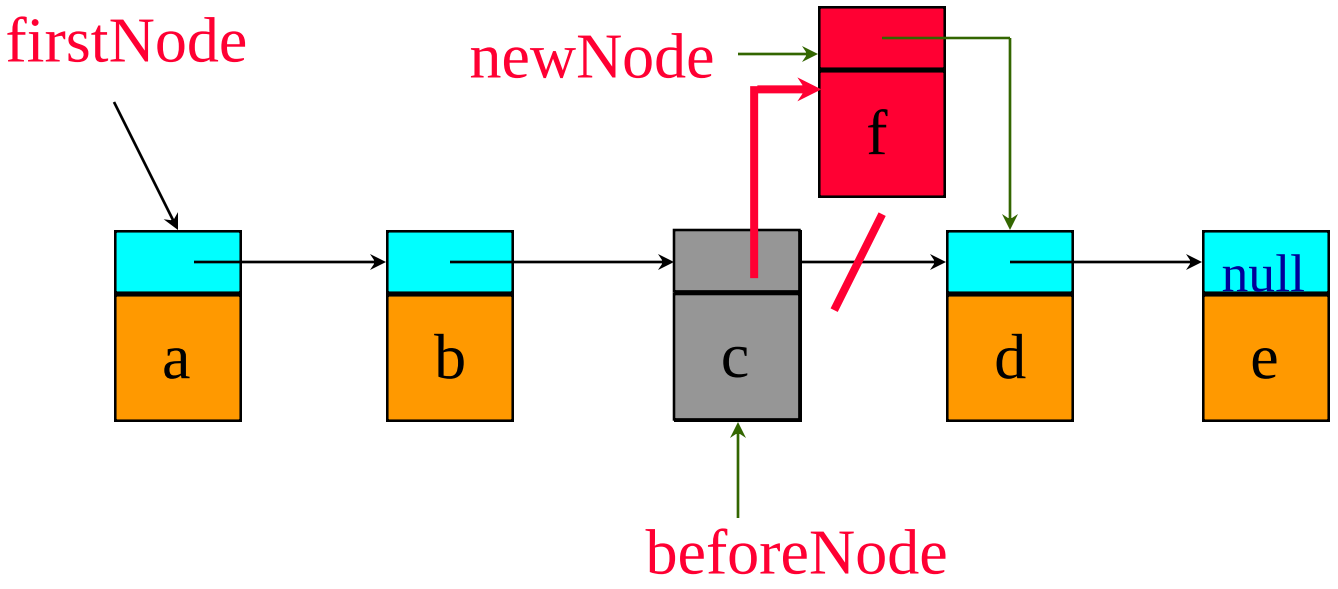
firstNode = newNode;

# One-Step add(0,'f')




firstNode = new ChainNode(new Character('f'),
firstNode);

# add(3,'f')



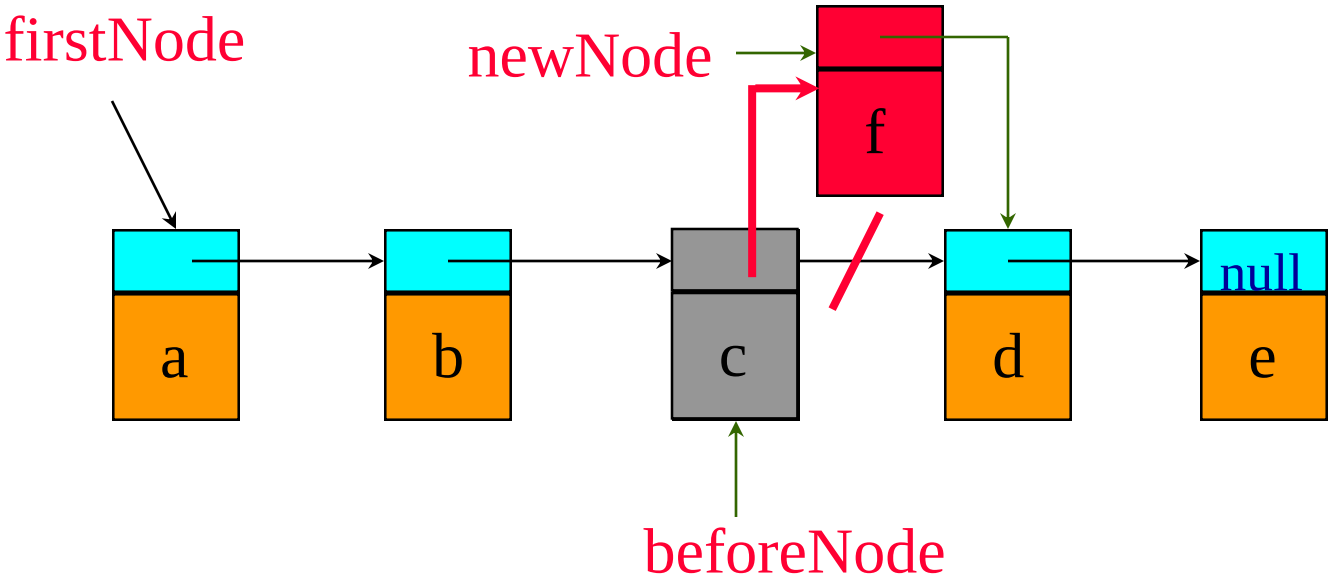

- first find node whose index is 2
- next create a node and set its data and link fields

```
ChainNode newNode = new ChainNode(new Character('f'),
                                  beforeNode.next);
```

- finally link beforeNode to newNode

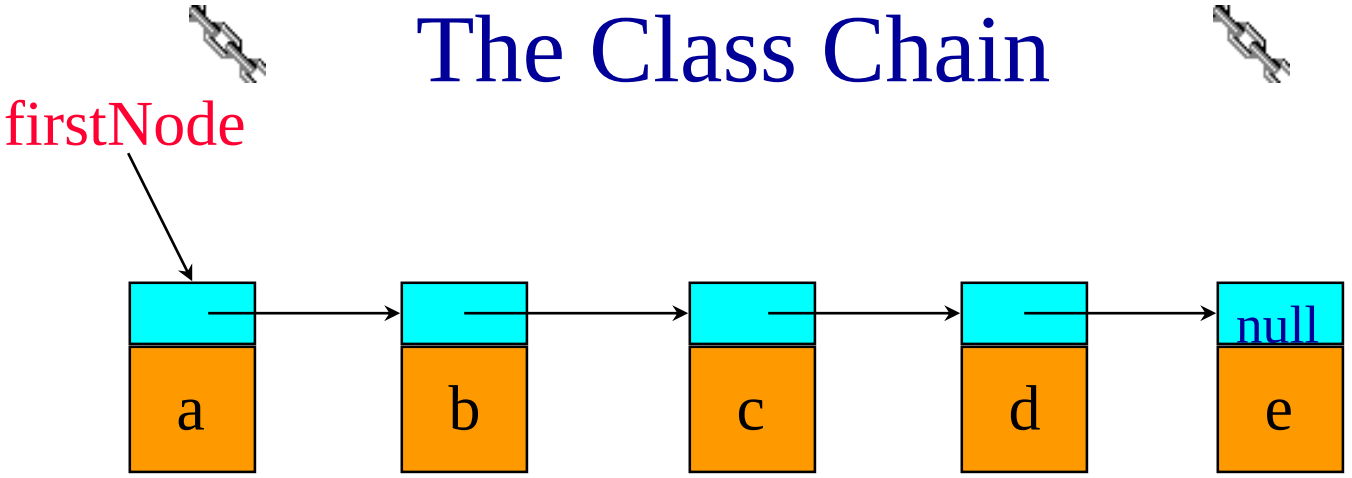
```
beforeNode.next = newNode;
```

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




```
beforeNode = firstNode.next.next;
beforeNode.next = new ChainNode(new Character('f'),
                                beforeNode.next);
```

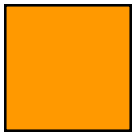
# The Class Chain




size = number of elements

Use ChainNode

next (datatype ChainNode)



element (datatype Object)

# The Class Chain

```
/** linked implementation of LinearList */
package dataStructures;
import java.util.*;  // has Iterator
public class Chain implements LinearList
{
   // data members
   protected ChainNode firstNode;
   protected int size;

   // methods of Chain come here
}
```

# Constructors

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

public Chain()
   {this(0);}
```

# The Method isEmpty

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

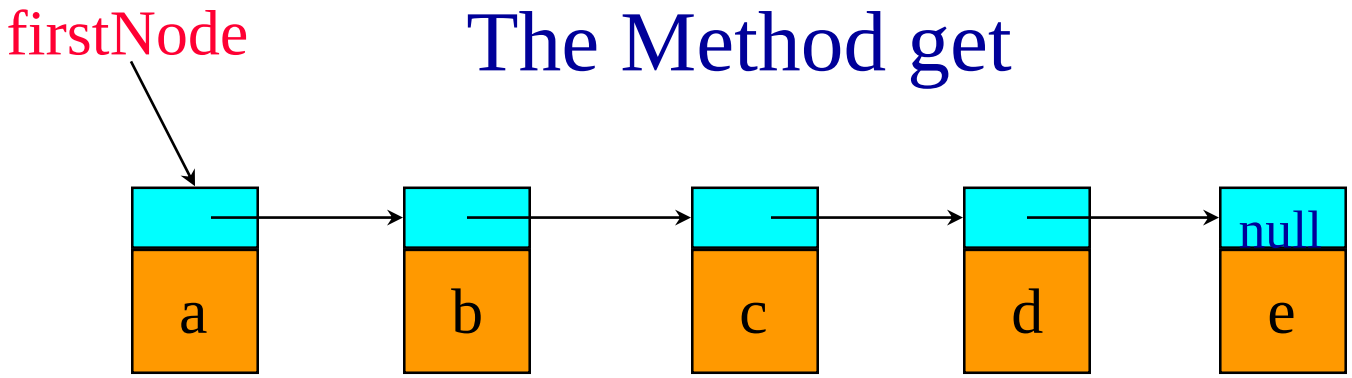
# The Method size()

```
/** @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
   ChainNode currentNode = firstNode;
   for (int i = 0; i < index; i++)
      currentNode = currentNode.next;

   return currentNode.element;
}
```

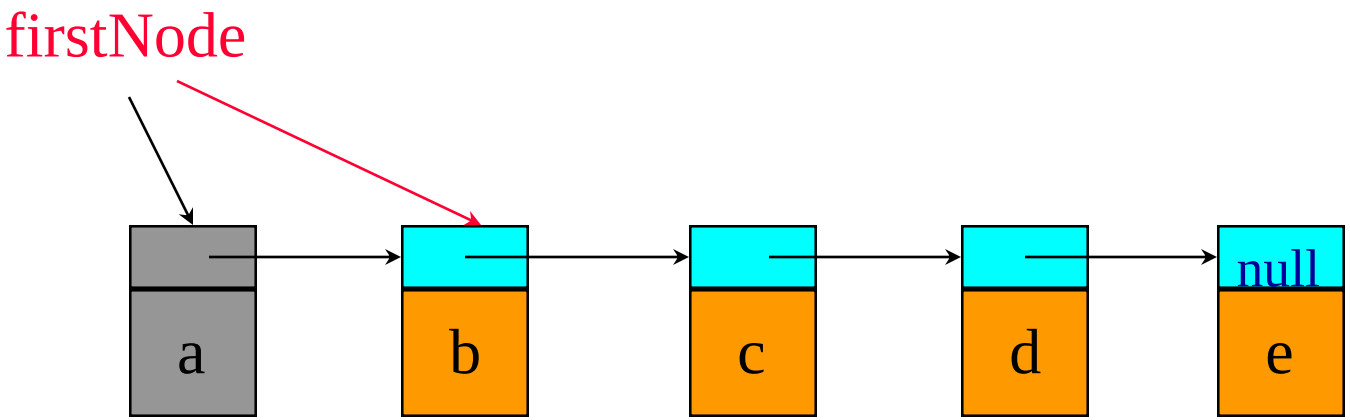
# The Method indexOf

```
public int indexOf(Object theElement)
 {
   // search the chain for theElement
   ChainNode currentNode = firstNode;
   int index = 0;  // index of currentNode
   while (currentNode != null &&
          !currentNode.element.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



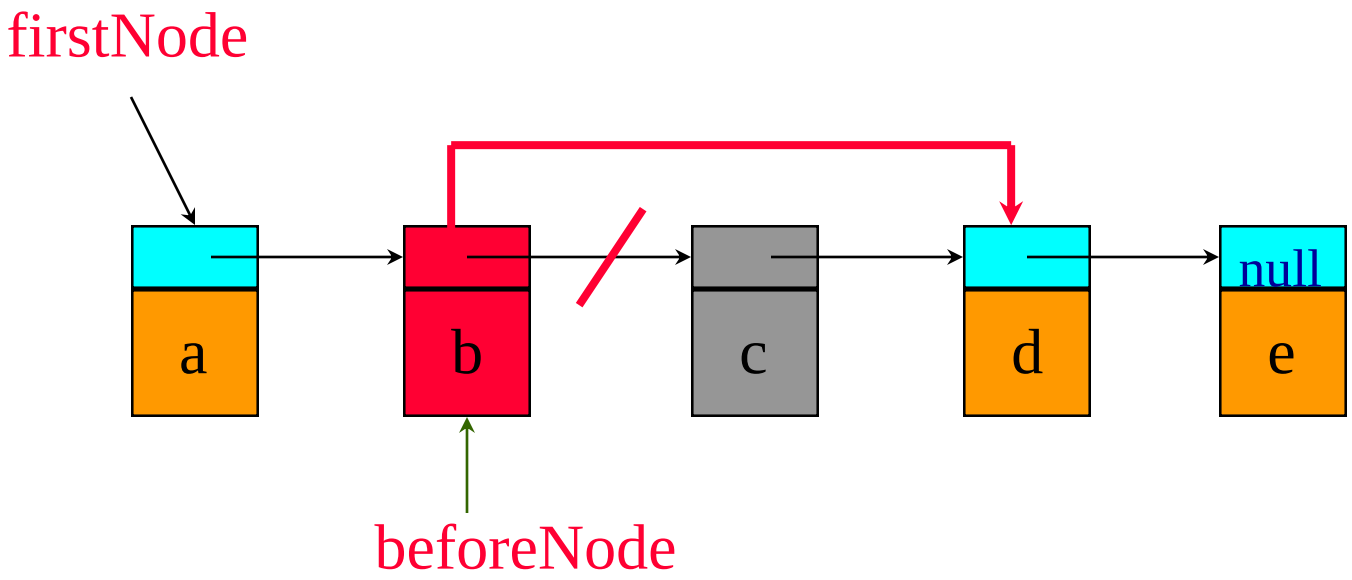

remove(0)

firstNode = firstNode.next;

# Remove An Element

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

  Object removedElement;
  if (index == 0) // remove first node
  {
    removedElement = firstNode.element;
    firstNode = firstNode.next;
  }
```

# remove(2)



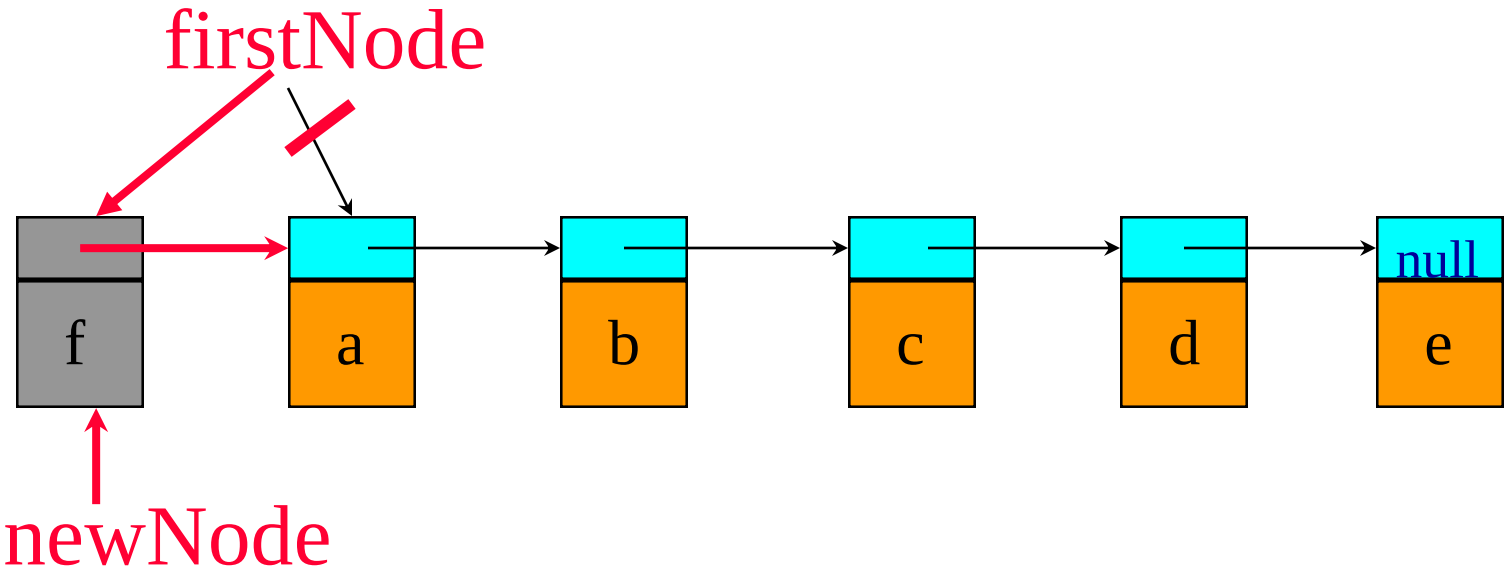

Find beforeNode and change its pointer.

beforeNode.next = beforeNode.next.next;

# Remove An Element

```
else
   {  // use q to get to predecessor of desired node
      ChainNode q = firstNode;
      for (int i = 0; i < index - 1; i++)
         q = q.next;

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

# One-Step add(0,'f')

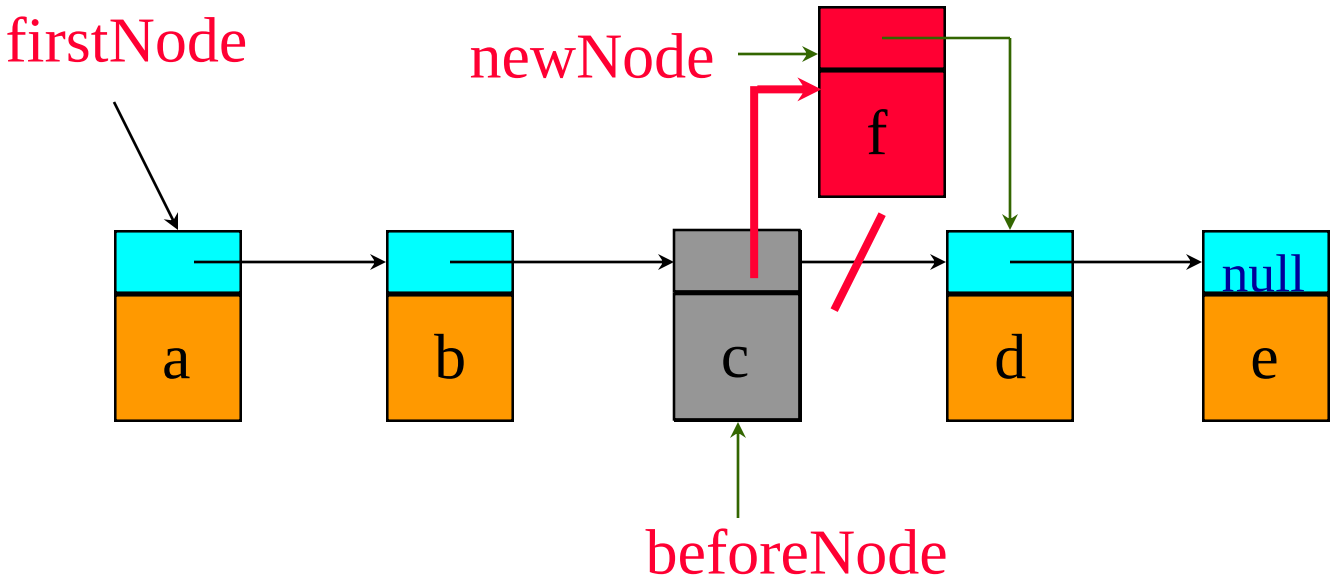



firstNode = new ChainNode('f', firstNode);

# 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 ChainNode(theElement, firstNode);
```

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

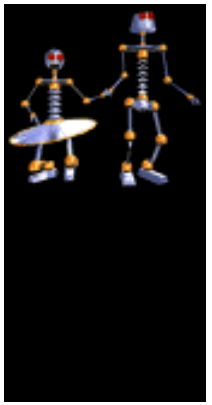



```
beforeNode = firstNode.next.next;
beforeNode.next = new ChainNode('f', beforeNode.next);
```

# Adding An Element

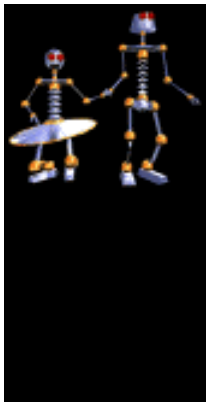
```
else
   { // find predecessor of new element
     ChainNode p = firstNode;
     for (int i = 0; i < index - 1; i++)
        p = p.next;

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

# Performance



40,000 operations of each type

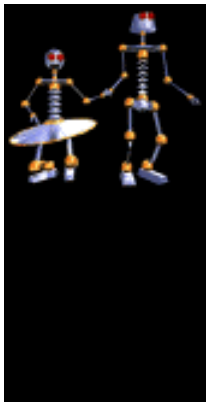
# Performance



40,000 operations of each type

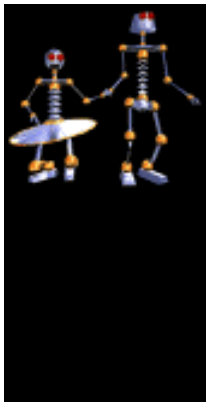
| Operation | FastArrayLinearList | Chain |
|---|---|---|
| get | 5.6ms | 157sec |
| best-case adds | 31.2ms | 304ms |
| average adds | 5.8sec | 115sec |
| worst-case adds | 11.8sec | 157sec |
| best-case removes | 8.6ms | 13.2ms |
| average removes | 5.8sec | 149sec |
| worst-case removes | 11.7sec | 157sec |

# Performance



Indexed AVL Tree (IAVL)

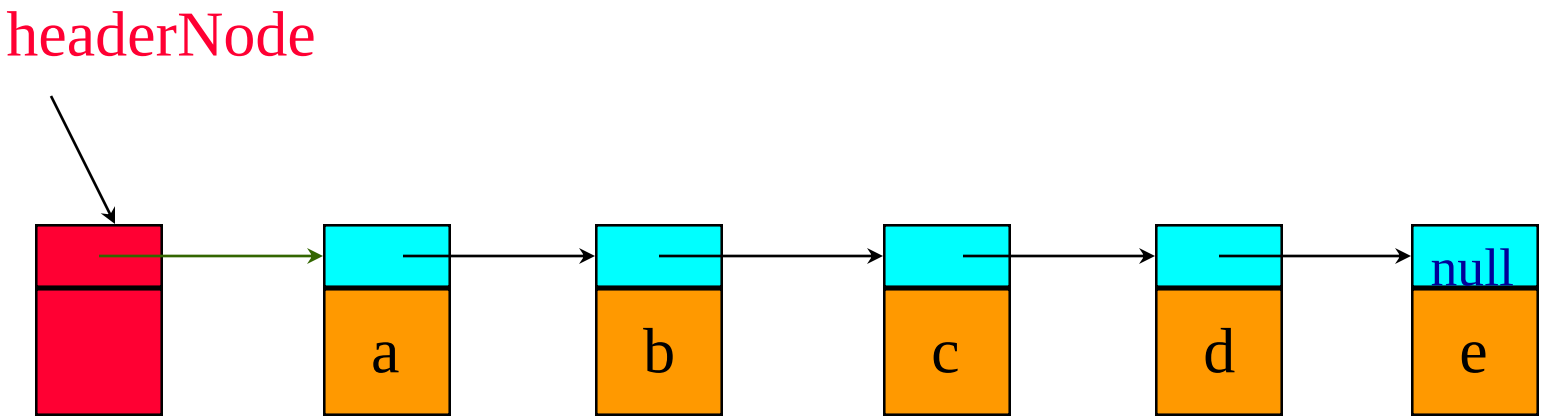
# Performance



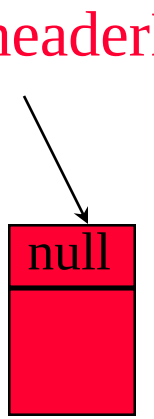
Indexed AVL Tree (IAVL)

| Operation | FastArrayLinearList | Chain | IAVL |
|---|---|---|---|
| get | 5.6ms | 157sec | 63ms |
| best-case adds | 31.2ms | 304ms | 253ms |
| average adds | 5.8sec | 115sec | 392ms |
| worst-case adds | 11.8sec | 157sec | 544ms |
| best-case removes | 8.6ms | 13.2ms | 1.3sec |
| average removes | 5.8sec | 149sec | 1.5sec |
| worst-case removes | 11.7sec | 157sec | 1.6sec |

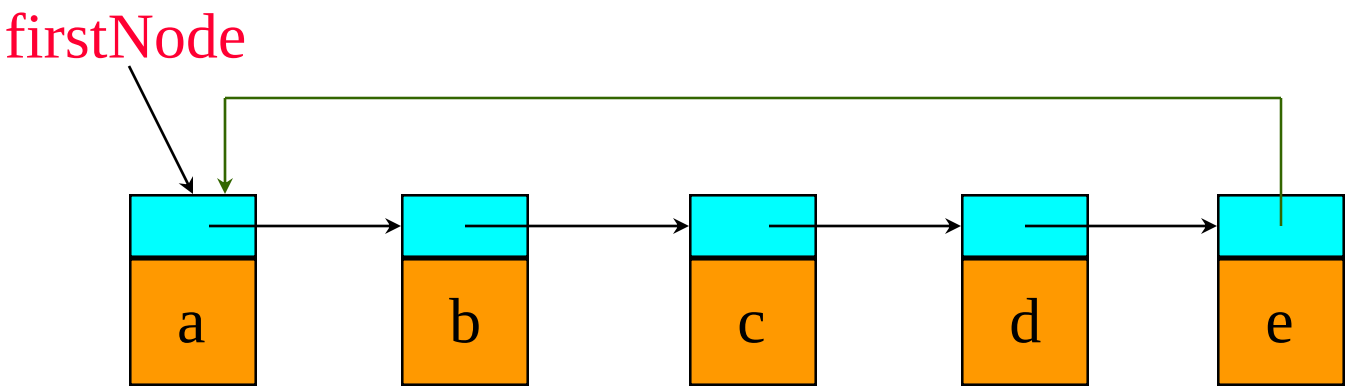
# Chain With Header Node



# Empty Chain With Header Node

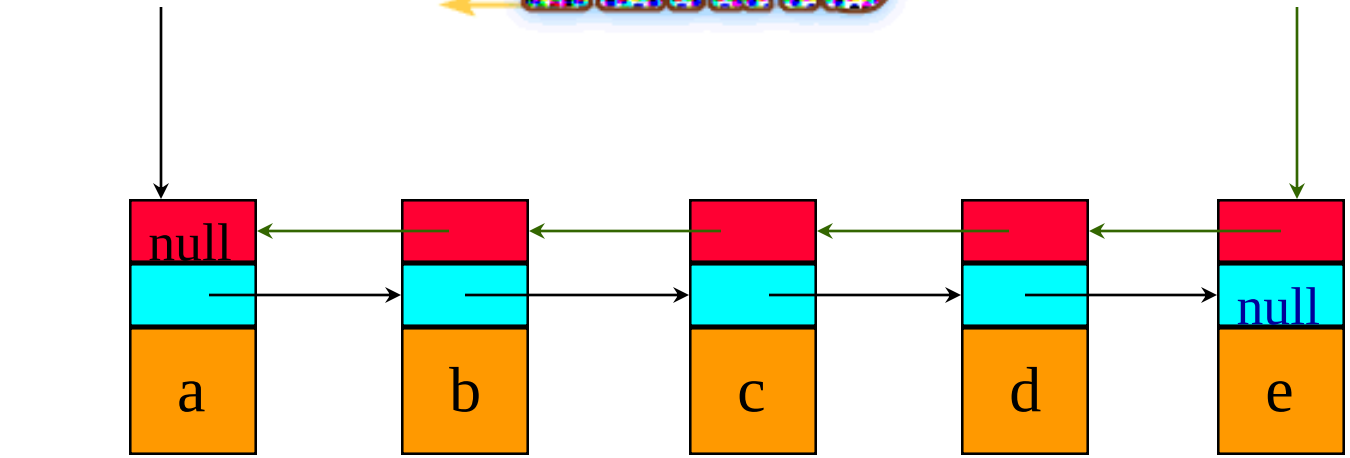



# Circular List

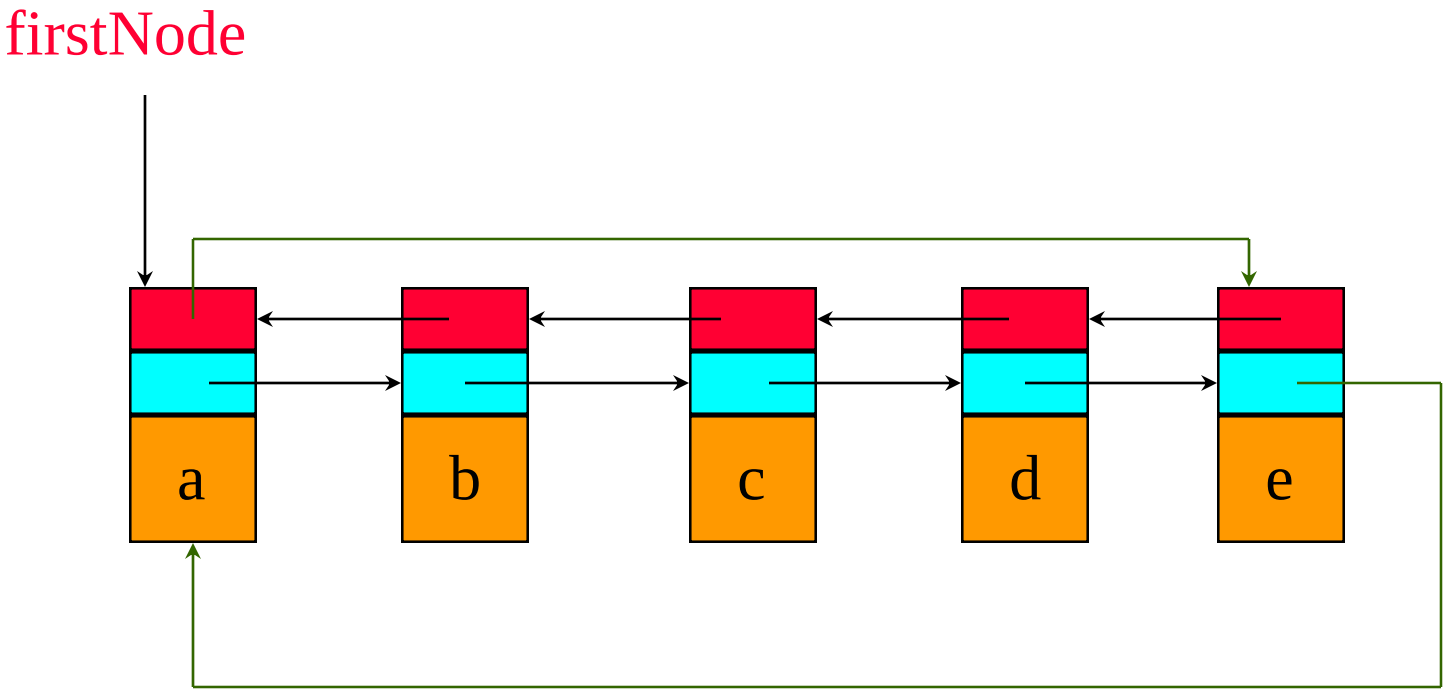




# Doubly Linked List

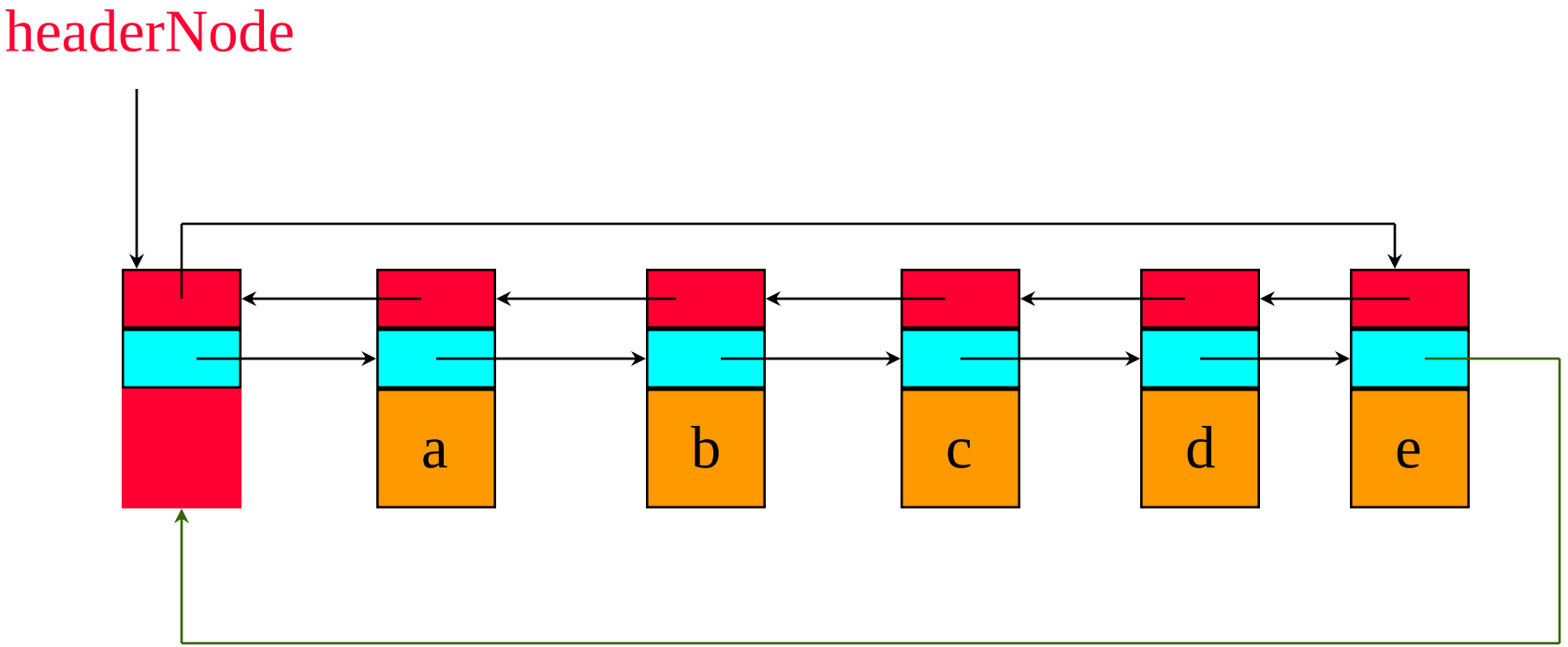




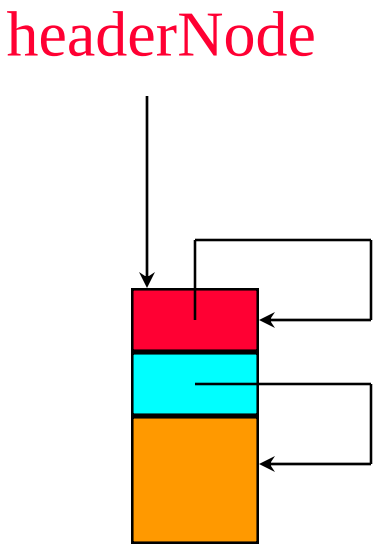
# Doubly Linked Circular List



# Doubly Linked Circular List With Header Node



# Empty Doubly Linked Circular List With Header Node



#  java.util.LinkedList 

- Linked implementation of a linear list.
- Doubly linked circular list with header node.
- Has all methods of LinearList plus many more.