جامعة بيرزيت
BIRZEIT UNIVERSITY

Lists

Dr. Abdallah Karakra

Computer Science Department

COMP242

Friday, March 15, 2024

List

- **What is a list?**

  An ordered sequence of elements A1, A2, …, AN

- **Two types of implementation:**
  1. **Array-Based**
  2. **Linked List**

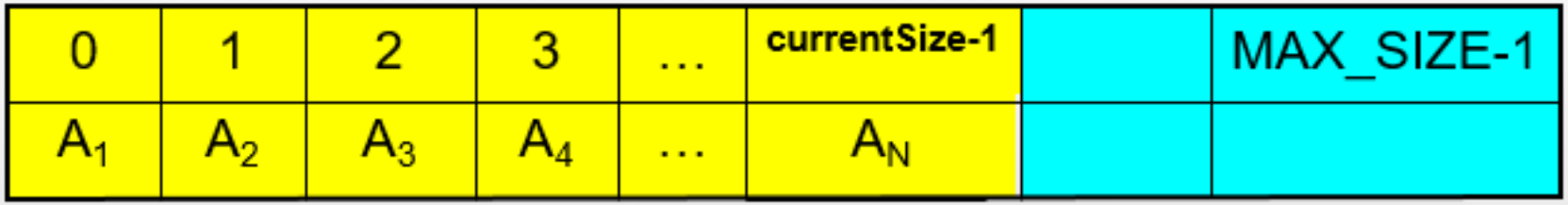
List

- **Common operations:**
  - **Insert**
  - **Find**
  - **Delete**
  - **Print list**
  - **Print element**
  - **IsEmpty**
  - **IsLast**
  - **FindPrevious**
  - **First**
  - **Kth**
  - **Last**

Lists: Array-Based Implementation

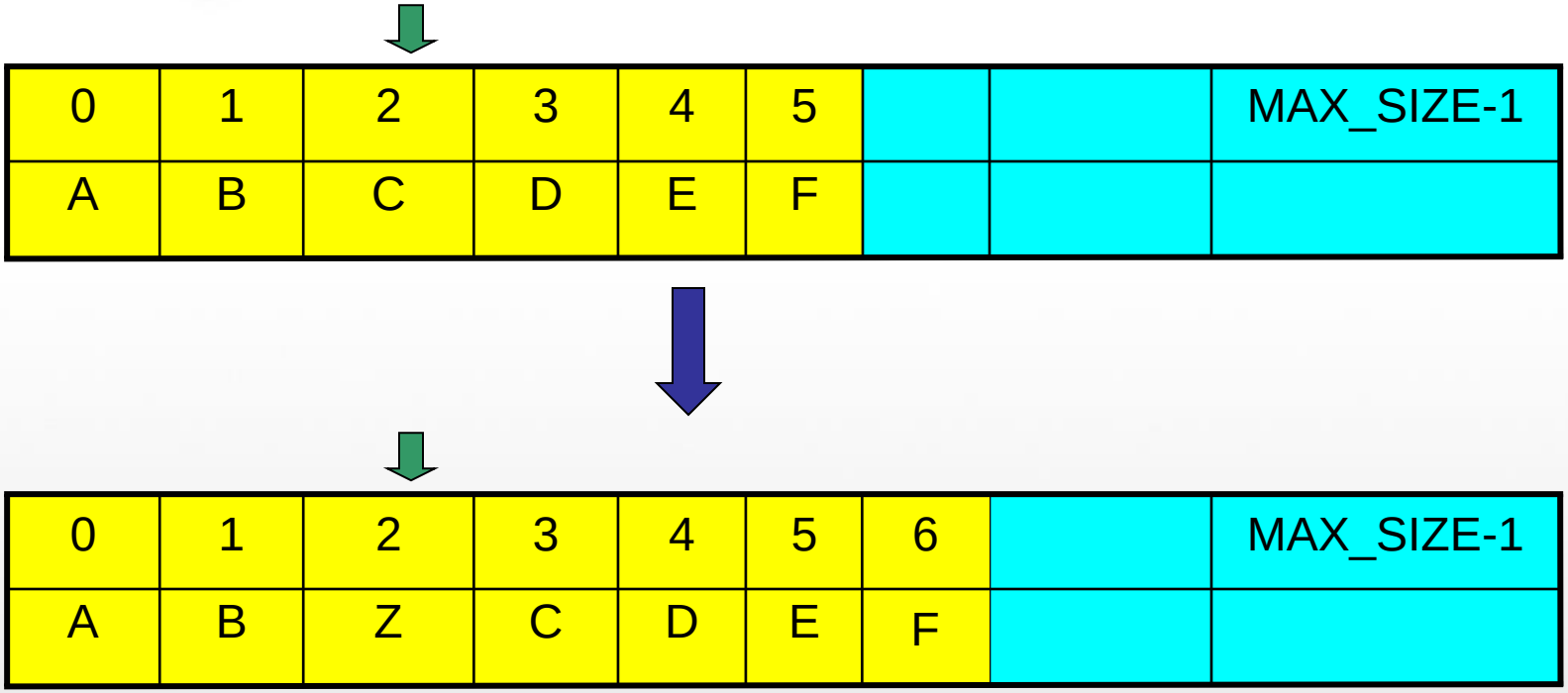
Basic Idea:

- Pre-allocate a big array of size MAX_SIZE
- Keep track of current size using a variable **`currentSize`**
- Shift elements when you have to insert or delete



Lists: Array-Based Implementation

Insert Z in kth position



Running time for N elements?

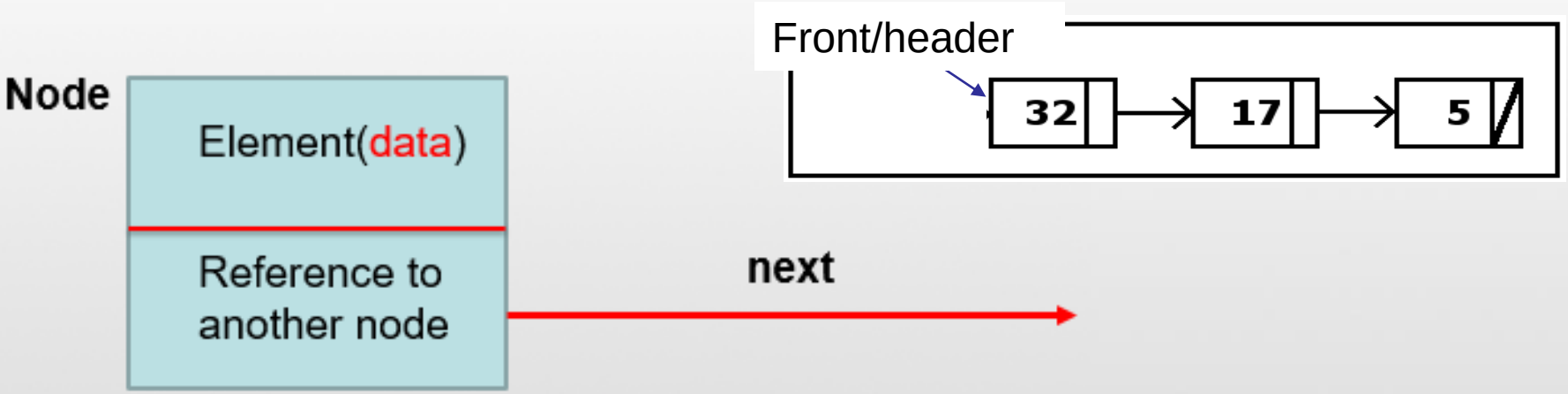
Worst case is insert at position 0. Must move all N items one position before the insert

This is O(N) running time. Probably too slow

Lists: Linked List Implementation

Definition: A **linked list** is a collection of **nodes** that together form a linear.

node: A compound object that stores the contents of the node (Element) and a pointer or reference to the next node in the list (next).



Node implementation

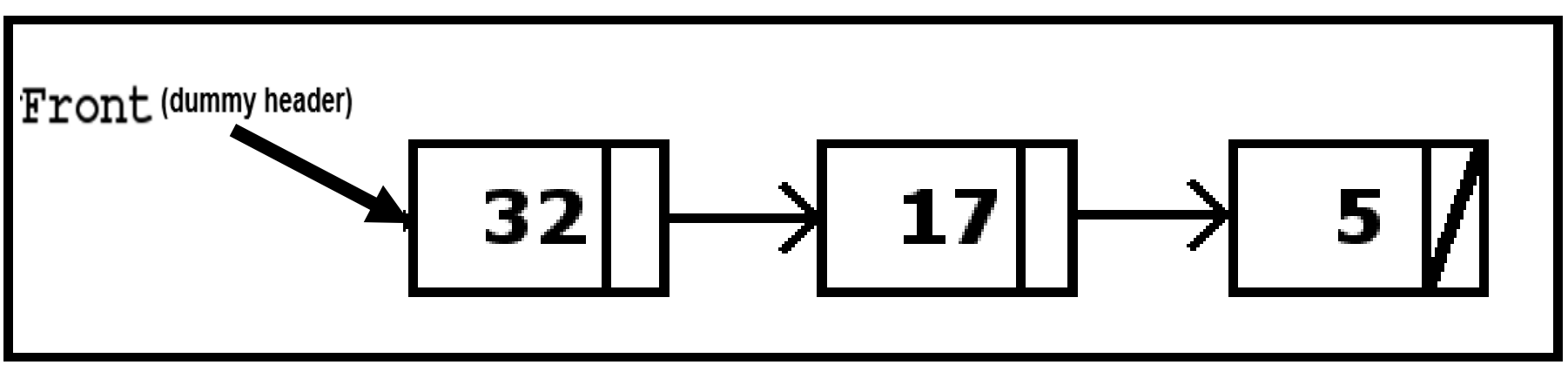
```
/* Stores one element of a linked list. */
public class Node {
  public Object element;// data
  public Node next;

  public Node(Object element) {
    this(element, null);
  }

  public Node(Object element, Node next) {
    this.element = element;
    this.next = next;
  }
}
```

linked list

a list implemented using a linked sequence of nodes

- the list only needs to keep a reference to the first node (we might name it Front)

- in Java: **java.util.LinkedList** (but we'll **write our own**)

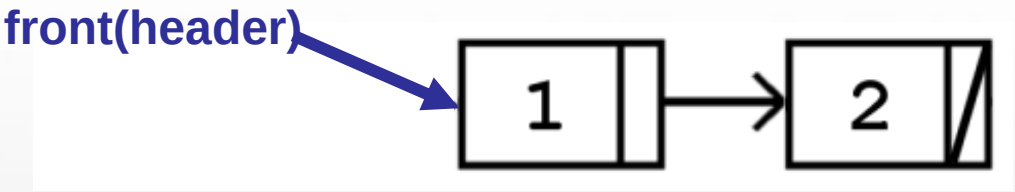


Linked node

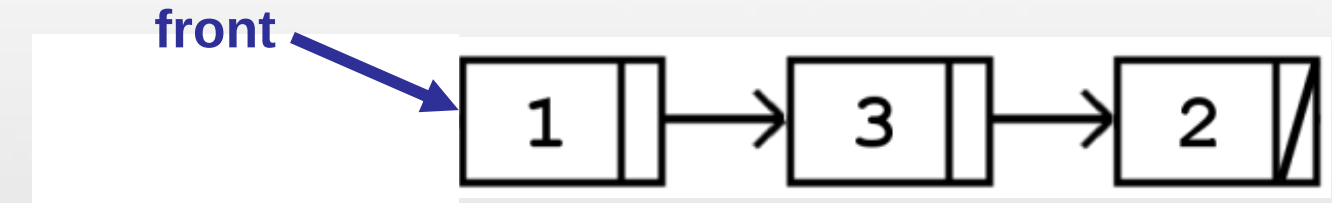
Let's examine sample chains of nodes together, and try to write the correct code for each.

- Each Node stores an `Integer` object

- Before:



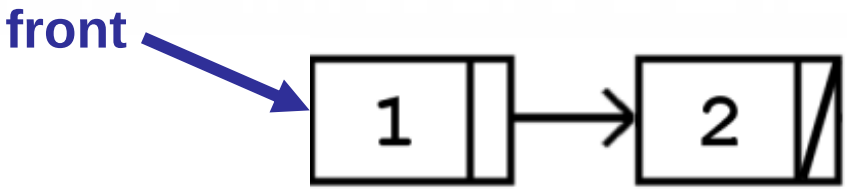
- After:



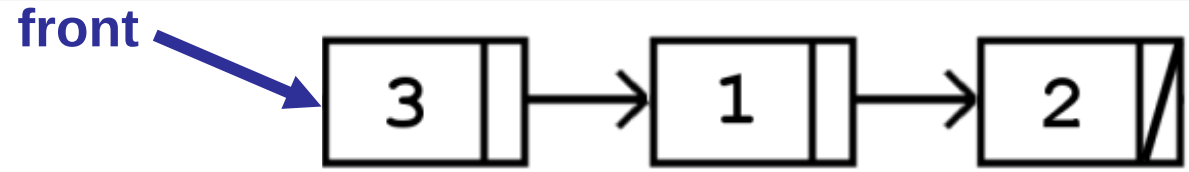

```
front.next = new Node(new Integer(3), front.next);
```

Linked node

- Before:



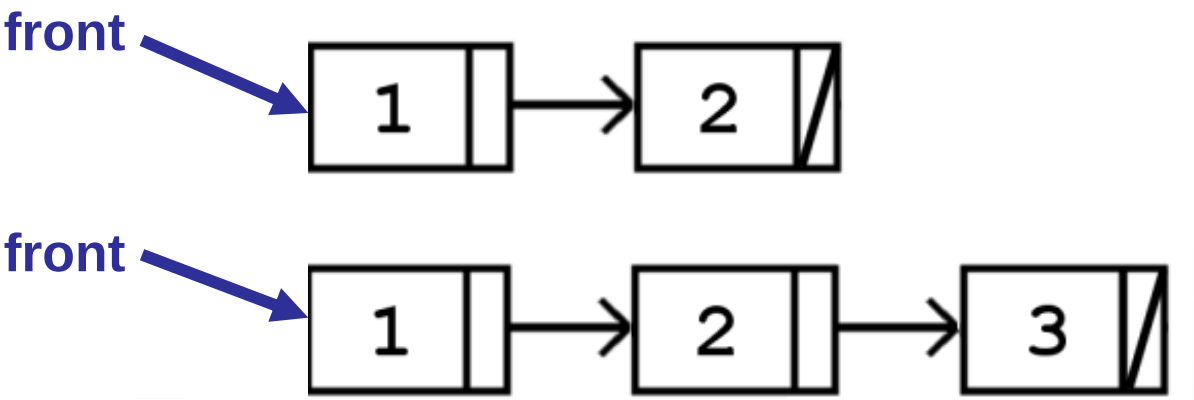
- After:




```
front = new Node(new Integer(3), front);
```

Linked node

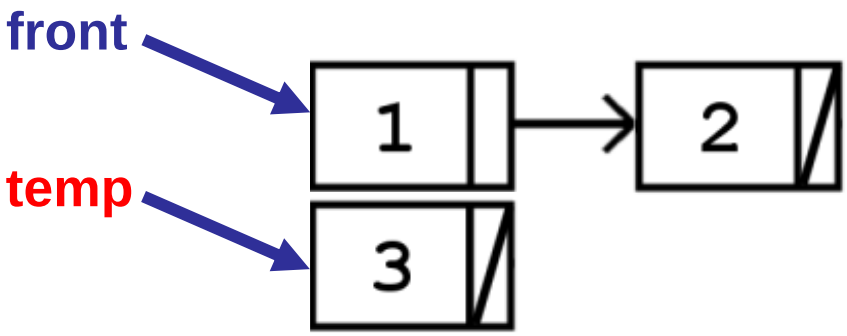
- Before:
- After:



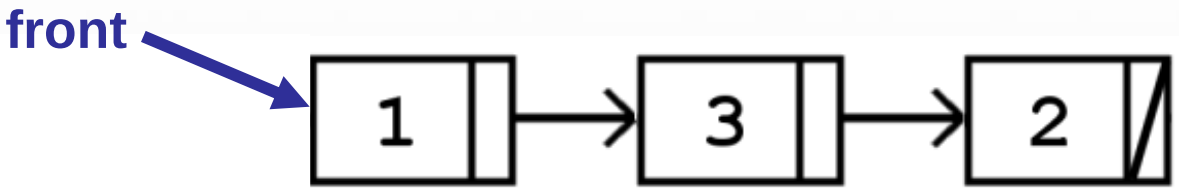

```
front.next.next = new Node(new Integer(3));
```

Linked node

- Before:



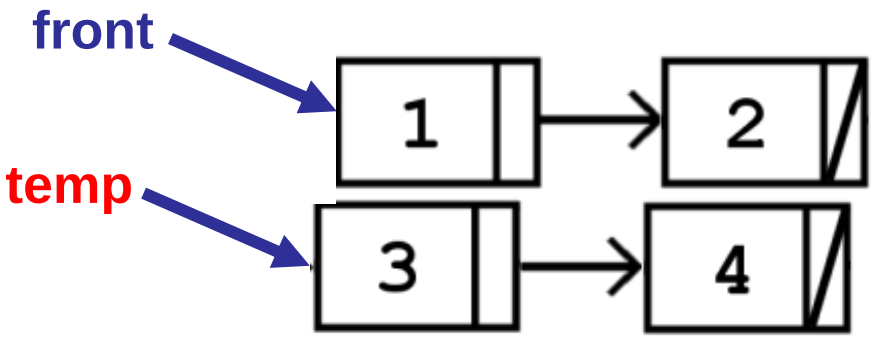
- After:



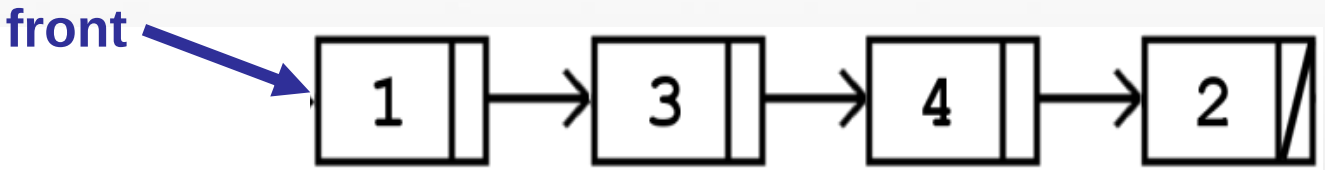

```
temp.next = front.next;  // 3 -> 2
front.next = temp;  // 1 -> 3
```

Linked node

- Before:



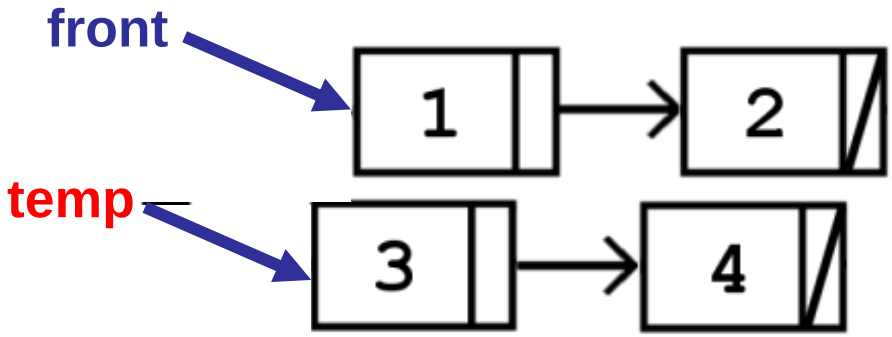
- After:



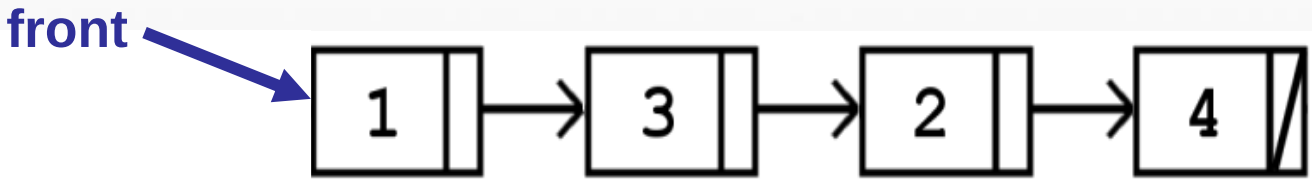

```
temp.next.next = front.next;  // 4 -> 2
front.next = temp;  // 1 -> 3
```

Linked node

- Before:



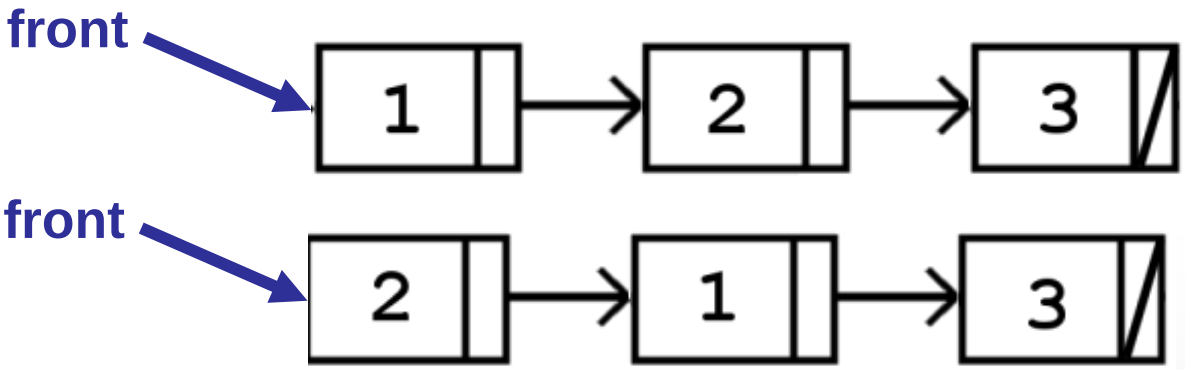
- After:




```
front.next.next = temp.next;  // 2 -> 4
temp.next = front.next;  // 3 -> 2
front.next = temp;  // 1 -> 3
```

Linked node

- Before:
- After:



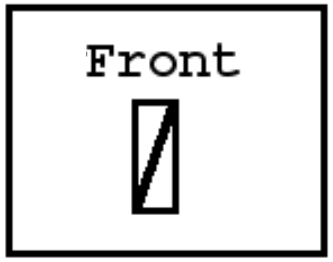

```
Node temp = front;  // temp -> 1
front = front.next;  // front -> 2
temp.next = front.next;  // 1 -> 3
front.next = temp;  // 2 -> 1
```

Linked list implementation

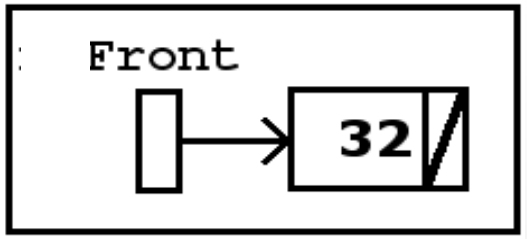
```
/* Models an entire linked list. */
public class LinkedList {
   private Node Front;// Dummy header

   public LinkedList() {
      Front = null;
   }

   /* Methods go here */
}
```

You have to know

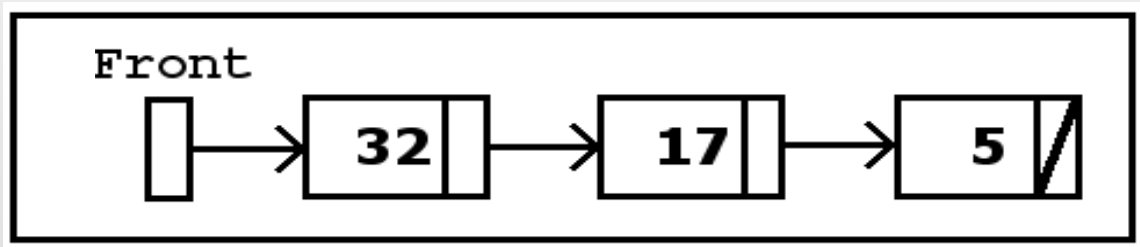
- empty list (`Front == null`)



- list with one element



- list with many elements



Analysis of LinkedList runtime

OPERATION	RUNTIME (Big-Oh)
add to start of list	**O(1)**
add to end of list	**O(*n*)**
add at given index	**O(*n*)**
clear	**O(1)**
get	**O(*n*)**
find index of an object	**O(*n*)**
remove first element	**O(1)**
remove last element	**O(*n*)**
remove at given index	**O(*n*)**
set	**O(*n*)**
size	**O(*n*)**
toString	**O(*n*)**

List Implementation (Array VS Linked List)

Operation	Array Complexity	Singly-linked list Complexity
Insert at beginning	O(n)	O(1)
Insert at end	O(1)	O(n)
Insert at middle*	O(n)	O(n)
Delete at beginning	O(n)	O(1)
Delete at end	O(1)	O(n)
Get last element	O(1)	O(n)
Delete at middle*	O(n): O(1) access followed by O(n) shift	O(n): O(n) search, followed by O(1) delete
Search	O(n) linear search	O(n)
Indexing: What is the element at a given position k?	O(1)	O(n)
Print list	O(n)	O(n)
size	O(1)	O(n)

*** middle: neither at the beginning nor at the end**

List Implementation (Array VS Linked List)

Operation	Array Complexity	Singly-linked list Complexity
Insert at beginning	O(n)	O(1)
Insert at end	O(1)	O(n)
Insert at middle*	O(n)	O(n)
Delete at beginning	O(n)	O(1)
Delete at end		
Get Last elem		
Delete at mid		) delete
Search	O(n) linear search	O(n)
Indexing: What is the element at a given position k?	O(1)	O(n)
Print list	O(n)	O(n)
size	O(1)	O(n)

Is there way to optimize the operations that in red color

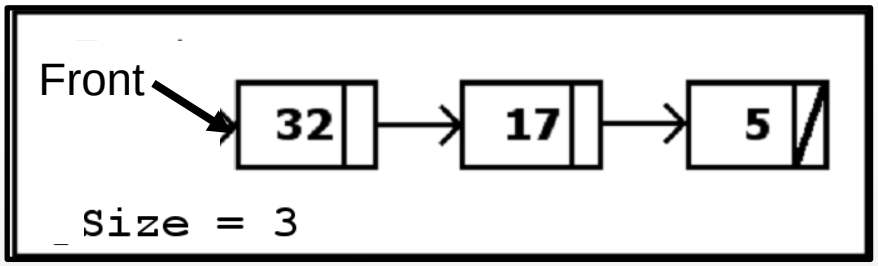
????

*** middle: neither at the beginning nor at the end**

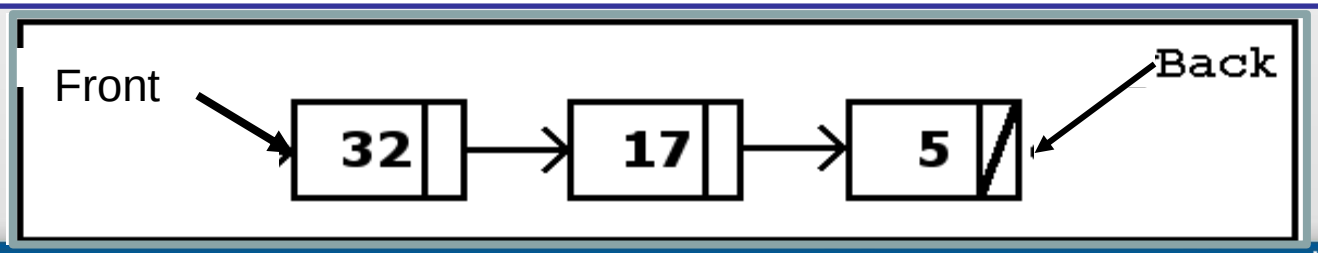
List Implementation (Array VS Linked List) optimization

problem: array list has a O(1) `size` method, but the linked list needs O(*n*) time

solution: add a `Size` field to the linked list
Big-Oh runtime for size improve to O(1)



problem: array list has O(1) get/remove of last **element**, but the linked list needs O(*n*)
solution: add a `Back` pointer to the last node
Big-Oh runtime improve the get/remove of last **element** to O(1)



Linked list implementation

```
/* Models an entire linked list. */
public class LinkedList {
   private Node Front,Back;
   private int Size;

   public LinkedList() {
      Front = Back = null;
      Size=0;
   }
   /* Methods go here */
}
```

Linked list implementation

```
/* void addFirst(Object o)
 * Inserts the given element at the beginning of the list. */
public void addFirst(Object element){
  Node newNode;
  newNode=new Node(element);
  if (Size==0) {// Empty List
    Front=Back=newNode;
  }
  else{
    newNode.next=Front;
    Front=newNode;
  }
   Size++;// update Size
}
```

Linked list implementation

```
/*Object getFirst()
  Returns the first element in the list. */

public Object getFirst() {
   if (Size == 0)
     return null;
   else
      return Front.element;
}
```

Linked list implementation

```
/*void addLast(Object o)
  Appends the given element to the end of the list.*/
public void addLast (Object element){
   Node newNode;
   newNode=new Node(element);
   if (Size==0){// Empty List
      Front=Back=newNode;
   }
   else{
     Back.next=newNode;
     Back=newNode;       // Or Back=Back.next;
   }
    Size++;// update Size
}
```

Linked list implementation

```
/* Object getLast()
   Returns the last element in the list.*/

    public Object getLast(){
     if (Size==0)
        return null;
     else
        return Back.element;
    }
```

Linked list implementation

```
/*Object get(int index)
  Returns the element at the specified position in the list.*/
 public Object get(int index){
    if (Size==0) return null; //empty list
    else if (index==0) return getFirst(); //first element
    else if (index==Size-1)return getLast();  //last element
    else if (index >0 && index<Size-1){
        Node current=Front;
        for (int i=0;i<index;i++)
           current=current.next;

        return current.element;
     }
     else
       return null; //out of boundary
 }
```

Linked list implementation

```
/*void add(int index, Object element) Inserts the specified
element  at the specified position index in this list.*/
  public void add(int index, Object element) {
    if (index==0) addFirst(element);
    else if (index>=size())addLast(element);
    else{
         Node newNode= new Node(element);
         Node current=Front;  //used to advance to proper position
         for (int i=0;i<index-1;i++)
           current=current.next;

         newNode.next=current.next;
         current.next=newNode;
         Size++;// update size
        }

  }
```

Linked list implementation

```
/*int size()
Returns the number of elements in the list.*/

 public int size(){
   return Size;
}


/*void add(Object o)
Appends the specified element to the end of the list*/
public void add(Object element)
{
     add(size(), element);
}
```