CSE 203: Lists

Dr. Mohammed Eunus Ali
Professor
CSE, BUET

Definition

An Abstract List (or List ADT) is linearly ordered data where the programmer explicitly defines the ordering

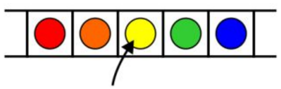
We will look at the most common operations that are usually

- The most obvious implementation is to use either an array or linked list
- These are, however, not always the most optimal

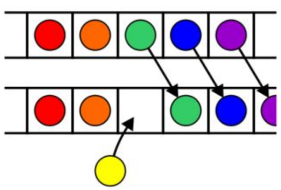
Operations

Operations at the k^{th} entry of the list include:

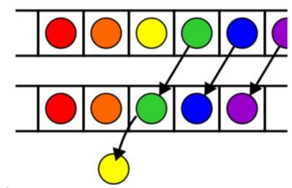
Access to the object



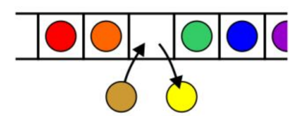
Insertion of a new object



Erasing an object

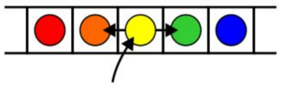


Replacement of the object



Operations

Given access to the k^{th} object, gain access to either the previous or next object



Given two abstract lists, we may want to

- Concatenate the two lists
- Determine if one is a sub-list of the other

Locations and run times

The most obvious data structures for implementing an abstract list are arrays and linked lists

- We will review the run time operations on these structures

We will consider the amount of time required to perform actions such as finding, inserting new entries before or after, or erasing entries at

- the first location (the *front*)
- an arbitrary (k^{th}) location
- the last location (the *back* or n^{th})

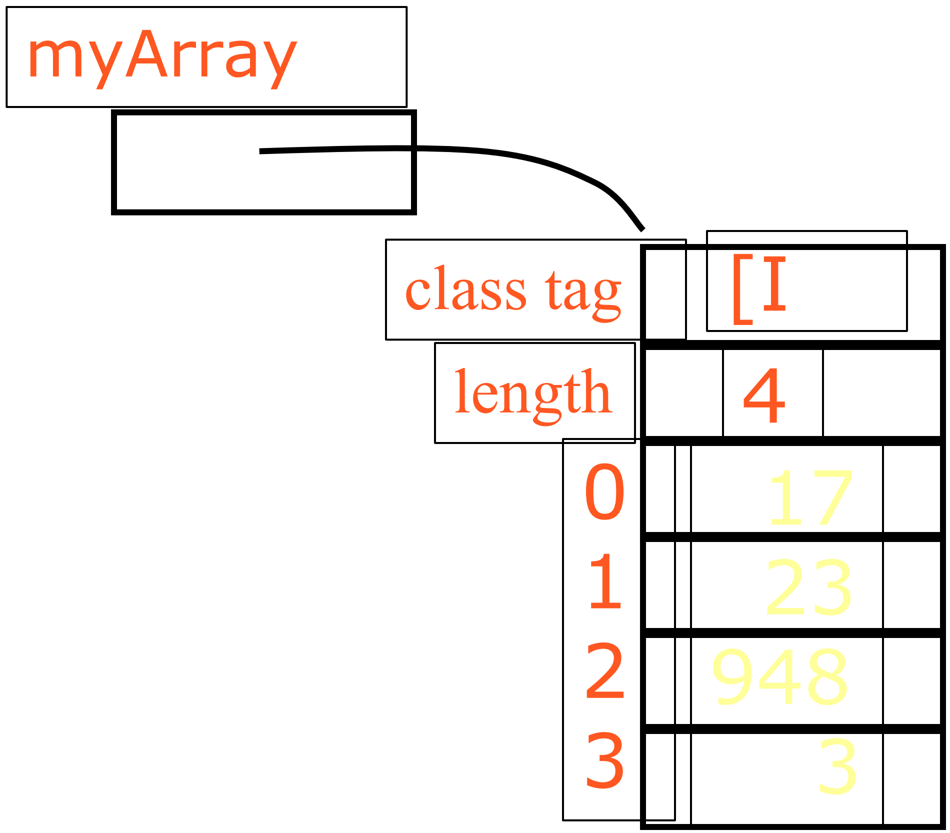
The run times will be $\Theta(1)$, $\mathrm{O}(n)$ or $\Theta(n)$

The array data structure

- An array is an indexed sequence of components
  - Typically, the array occupies sequential storage locations
  - The length of the array is determined when the array is created, and cannot be changed
  - Each component of the array has a fixed, unique index
    - Indices range from a lower bound to an upper bound
  - Any component of the array can be inspected or updated by using its index
    - This is an efficient operation: $O(1)$ = constant time

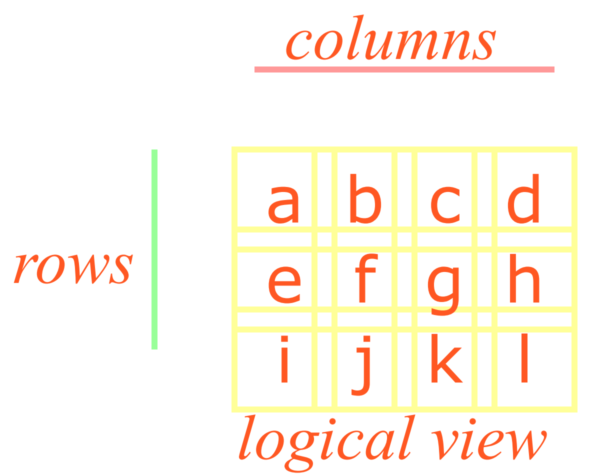
Arrays in Java III

- Here's one way to visualize an array in Java:

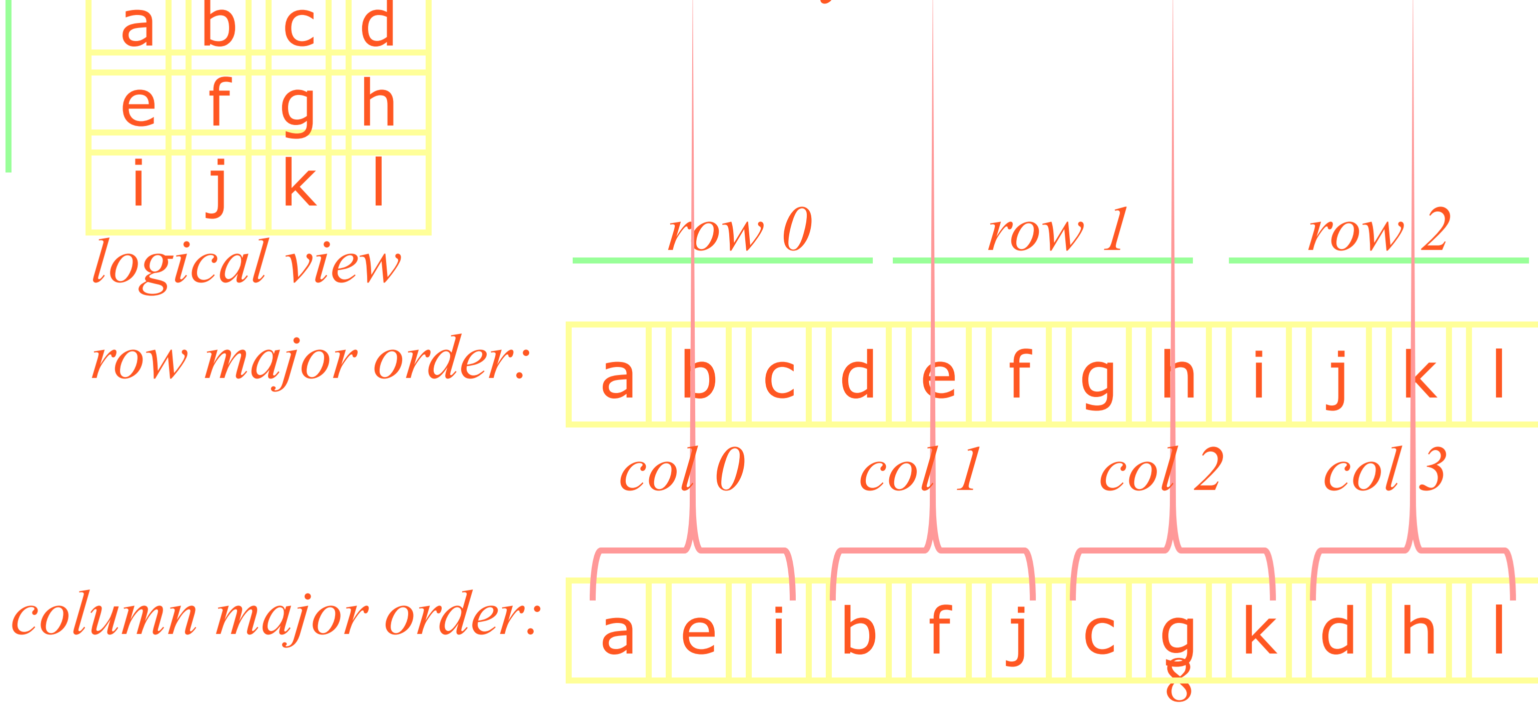


Two-dimensional arrays I

- Some languages (Fortran, Pascal) support two-dimensional (2D) arrays:



- A two-dimensional array may be stored in computer memory in either of two ways:



Summary

- Arrays have the following advantages:
  - Accessing an element by its index is very fast (constant time)
- Arrays have the following disadvantages:
  - All elements must be of the same type
  - The array size is fixed and can never be changed
  - Insertion into arrays and deletion from arrays is very slow

Case Study: List Implementations

LIST ADT

State

Set of ordered items
Count of items

Behavior

get(index) return item at index
set(item, index) replace item at index
add(item) add item to end of list
insert(item, index) add item at index
delete(index) delete item at index
size() count of items

[88.6, 26.1, 94.4]

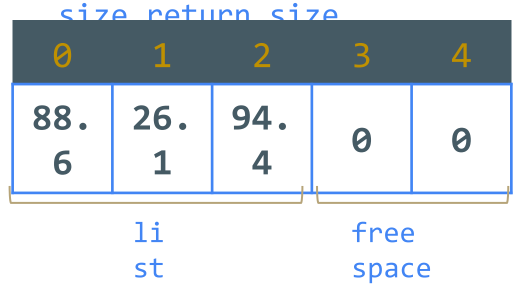
ArrayList<E>

State

data[]
size

Behavior

get return data[index]
set data[index] = value
add data[size] = value, if out of space grow data
insert shift values to make hole at index, data[index] = value, if out of space grow data
delete shift following values forward
size return size



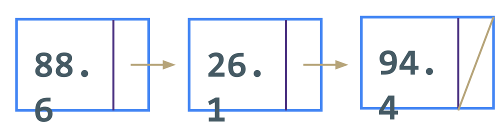
LinkedList<E>

State

Node front;
size

Behavior

get loop until index, return node's value
set loop until index, update node's value
add create new node, update next of last node
insert create new node, loop until index, update next fields
delete loop until index, skip node
size return size



Case Study: Let's Zoom In On ArrayList

- How do Java / other programming languages implement ArrayList to achieve all the List behavior?

- On the inside:

  - stores the elements **inside an array** (which has a fixed capacity) that typically has more space than currently used (For example when there is only 1 element in the actual list, the array might have 10 spaces for data),

  - stores all of these elements at the front of the array and **keeps track of how many there are** (the size) so that the implementation doesn't get confused enough to look at the empty space. This means that sometimes we will have to do a lot of work to shift the elements around.

List View

```
["Paul", "Leona", "Ryan"]
```

ArrayList View

```
["Paul", "Leona", "Ryan", null, null, null]
```

Comparing ADT Implementations: List

	ArrayList	LinkedList
`add (front)`	linear	constant
`remove (front)`	linear	constant
`add (back)`	(usually) constant	linear
`remove (back)`	constant	linear
`get`	constant	linear
`put`	linear	linear

- Important to be able to come up with this, and understand why
- But only half the story: to be able to make a design decision, need the context to understand which of these we should prioritize

Linked Lists

Definition

A linked list is a data structure where each object is stored in a *node*

As well as storing data, the node must also contains a reference/pointer to the node containing the next item of data

Linked Lists

We must dynamically create the nodes in a linked list

Thus, because `new` returns a pointer, the logical manner in which to track a linked lists is through a pointer

A `Node` class must store the data and a reference to the next node (also a pointer)

1

Node Class

The node must store data and a pointer (C++):

```
class Node {
    public:
        Node( int = 0,
Node * = nullptr );

        int value() const;
        Node *next() const;

    private:
        int node_value;
        Node *next_node;
};
```

The node must store data and a reference (Java):

```
class Node{
int node_value;
    Node next_node;

  public Node(){
      node_value= 0;
      next_node =null;
  };

    public int value();
    public Node next();

};
```

Linked List Class

The linked list class requires member variable: a pointer to a node

```
class List {
    public:
        List(){}
    private:
        Node *list_head;
    // ...
};
```

```
class List {
private Node list_head;
public List(){
   list_head=null;
}
    // ...
};
```

Structure

To begin, let us look at the internal representation of a linked list

Suppose we want a linked list to store the values

`42   95   70   81`

in this order

Structure

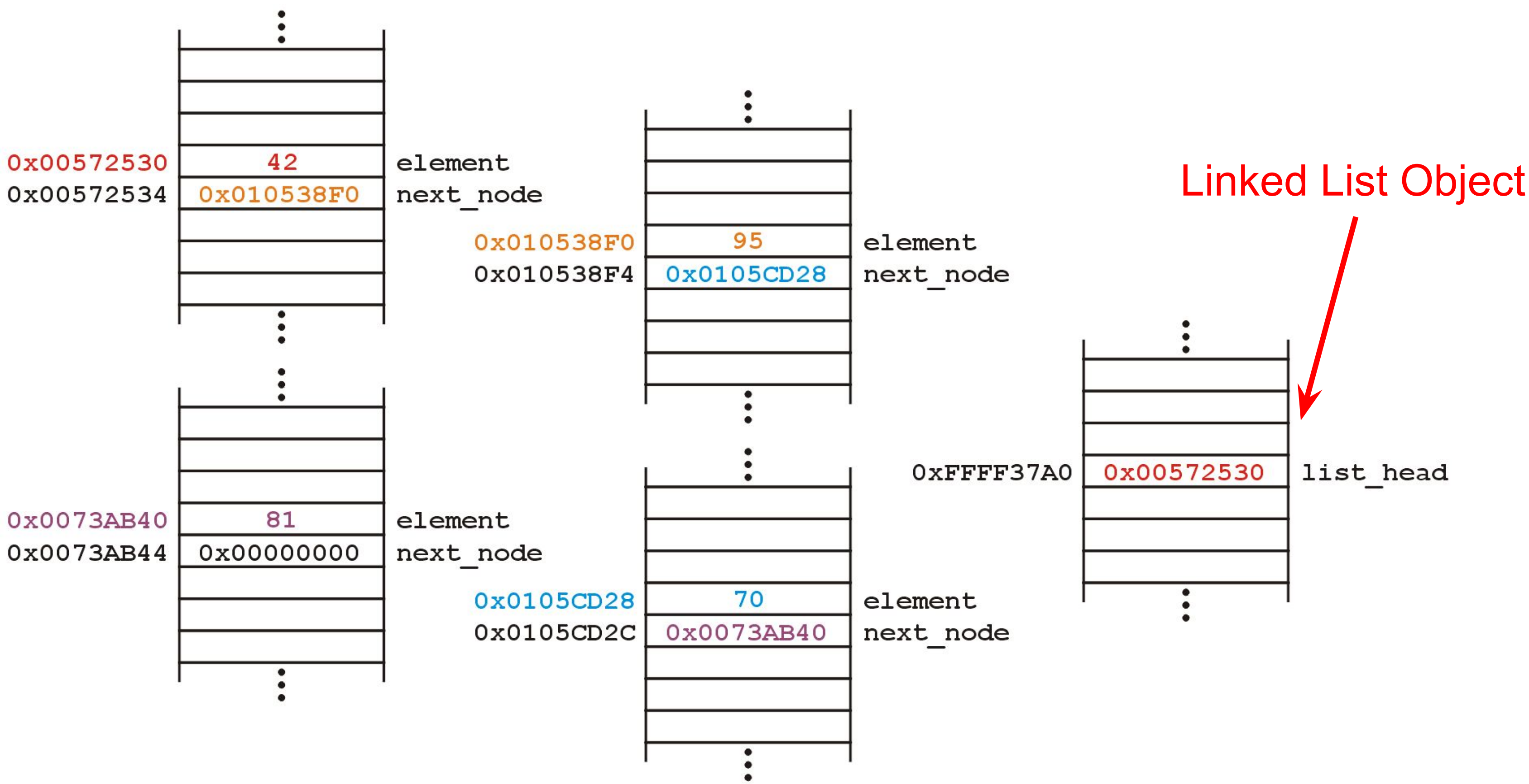
A linked list uses linked allocation, and therefore each node may appear anywhere in memory

Also the memory required for each node equals the memory required by the member variables

- 4 bytes for the linked list (a pointer)
- 8 bytes for each node (an `int` and a pointer)
  - We are assuming a 32-bit machine

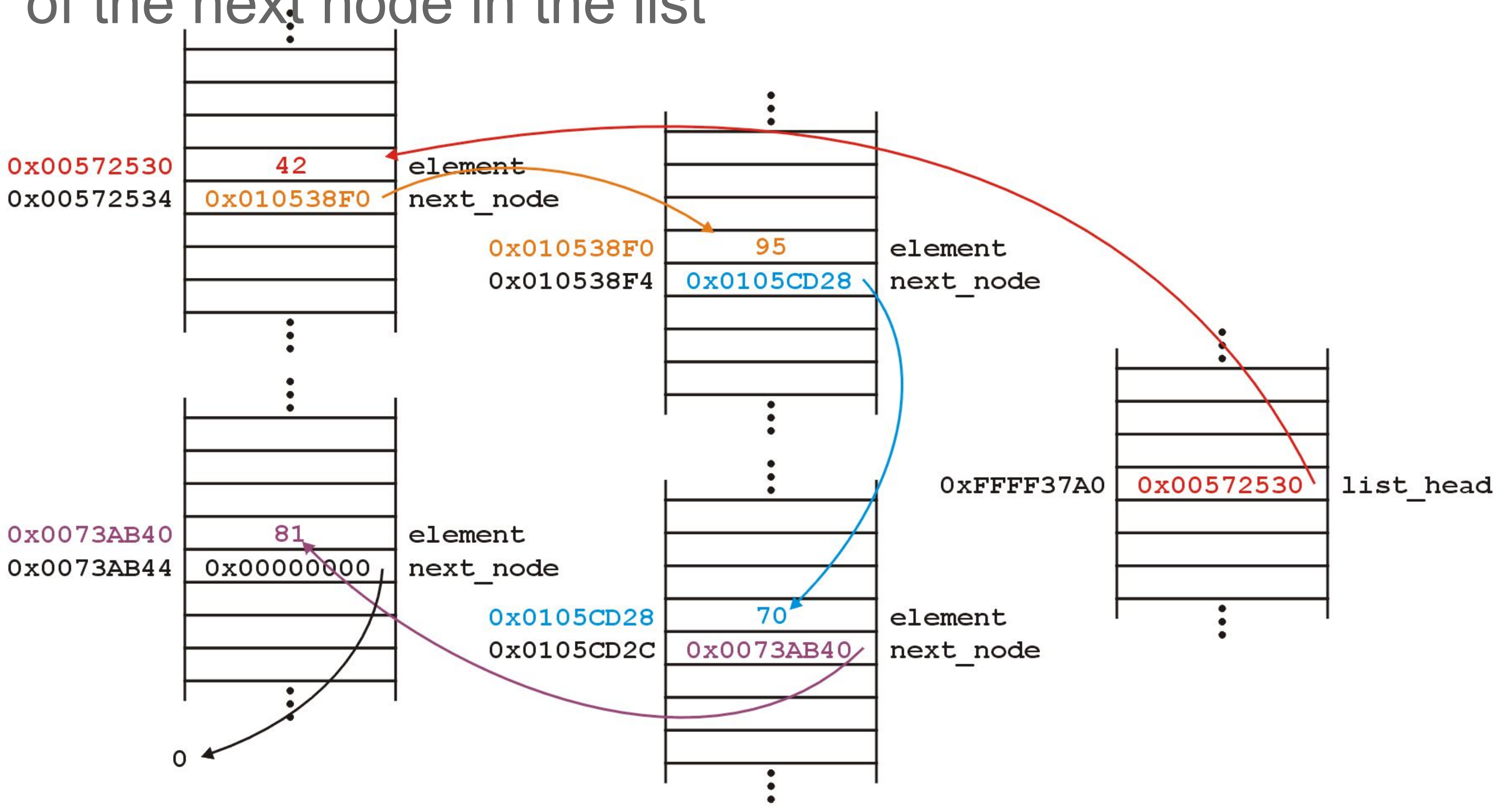
Structure

Such a list could occupy memory as follows:



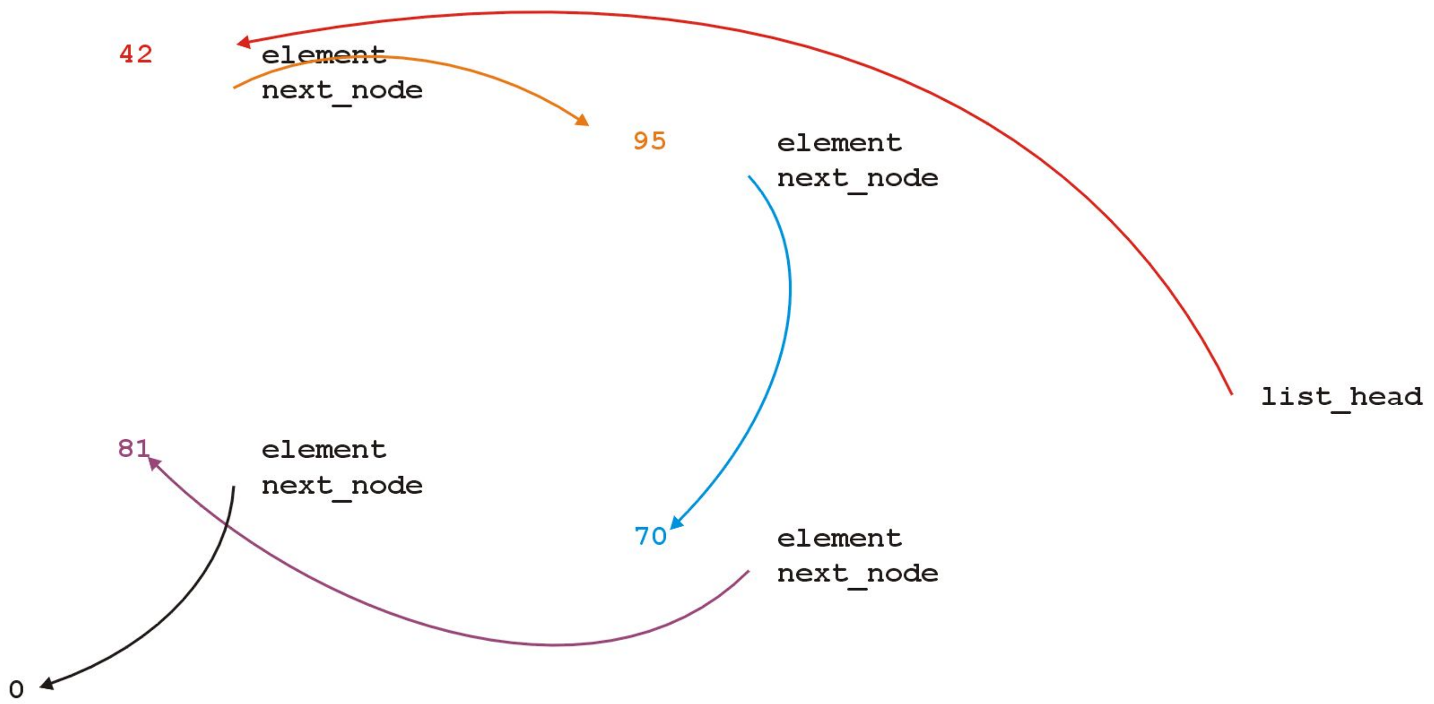
Structure

The **`next_node`** pointers store the addresses of the next node in the list



Structure

Because the addresses are arbitrary, we can remove that information:



Structure

We will clean up the representation as follows:

We do not specify the addresses because they are arbitrary and:

- The contents of the circle is the value
- The `next_node` pointer is represented by an arrow

Operations

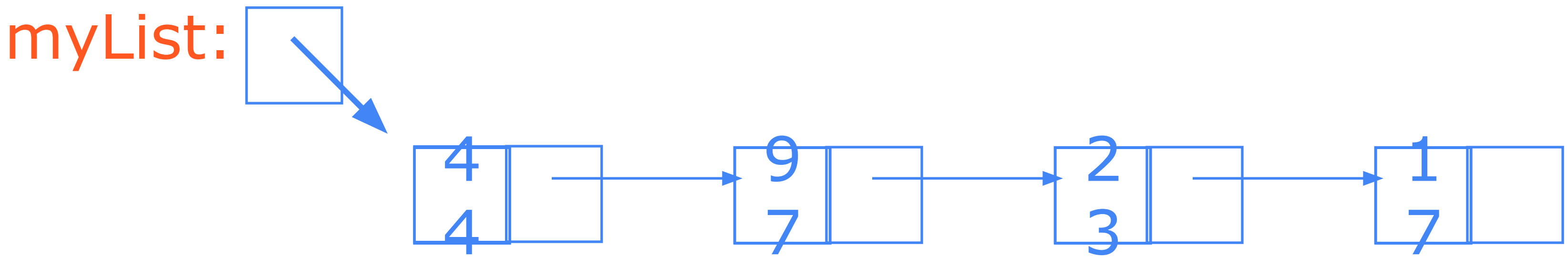
First, we want to create a linked list

We also want to be able to:

- insert into,
- access, and
- erase from

the values stored in the linked list

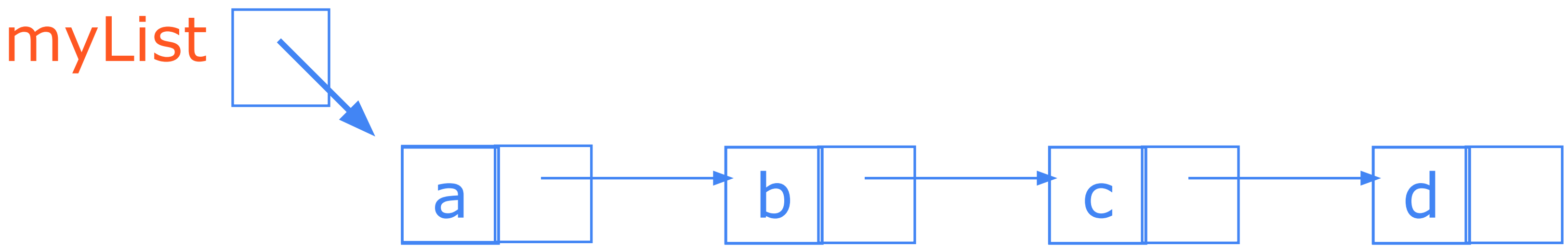
Creating links in Java




```
class Node { int value;
             Node next;
    Node(int v, Node n) { // constructor
        value = v;
        next = n;
    }
}
Node temp = new Node(17, null);
temp = new Node(23, temp);
temp = new Node(97, temp);
Node myList = new Node(44, temp);
```

Singly-linked lists

- Here is a singly-linked list (SLL):



- Each node contains a value and a link to its successor (the last node has no successor)
- The header points to the first node in the list (or contains the null link if the list is empty)

Singly-linked lists in Java

```
public class SLL {

    private SLLNode first;

    public SLL() {
        this.first = null;
    }

    // methods...
}
```

- This class actually describes the *header* of a singly-linked list
- However, the entire list is accessible from this header
- Users can think of the SLL as *being* the list
  - Users shouldn't have to worry about the actual implementation

SLL nodes in Java

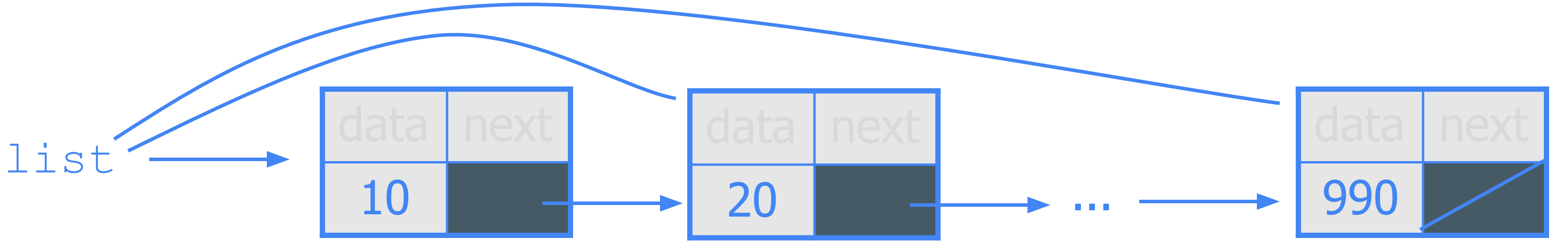
```
public class SLLNode {
    protected Object value;
    protected SLLNode next;

    protected SLLNode(Object value,
                              SLLNode next) {
        this.value = value;
        this.next = next;
    }
}
```

Traversing a list?

- One (bad) way to print every value in the list:

```
while (list != null) {
    System.out.println(list.data);
    list = list.next;     // move to next node
}
```

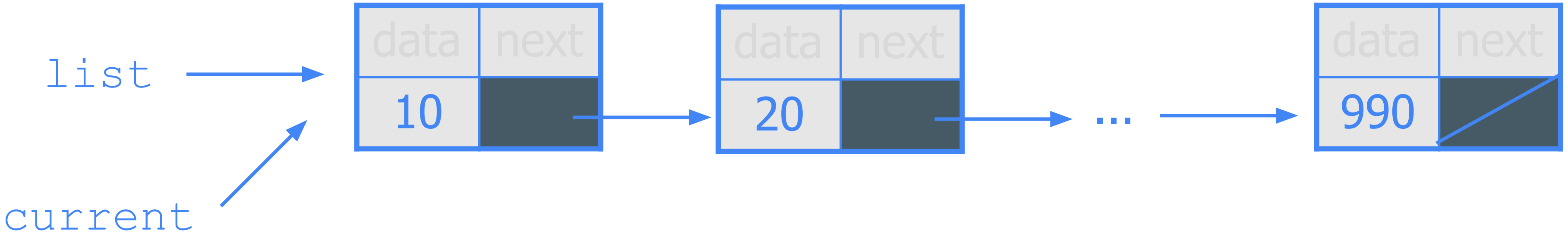
- What's wrong with this approach?
  - (It loses the linked list as it prints it!)



A `current` reference

- Don't change `list`. Make another variable, and change it.
  - A `Node` variable is NOT a `Node` object

```
Node current = list;
```



- What happens to the picture above when we write:

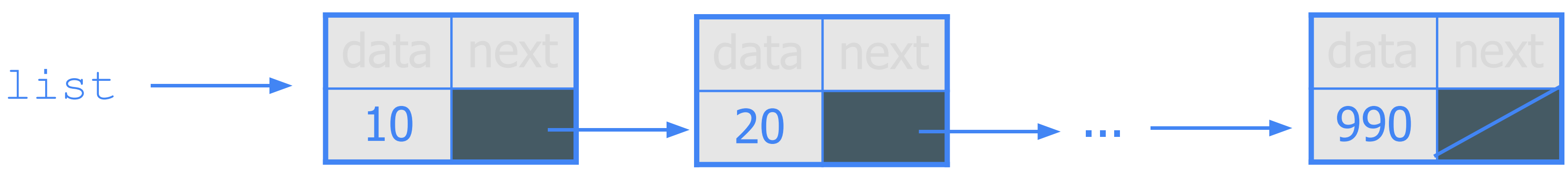
```
current = current.next;
```

Traversing a list correctly

- The correct way to print every value in the list:

```
Node current = list;
while (current != null) {
    System.out.println(current.data);
    current = current.next;  // move to next node
}
```

  - Changing `current` does not damage the list.



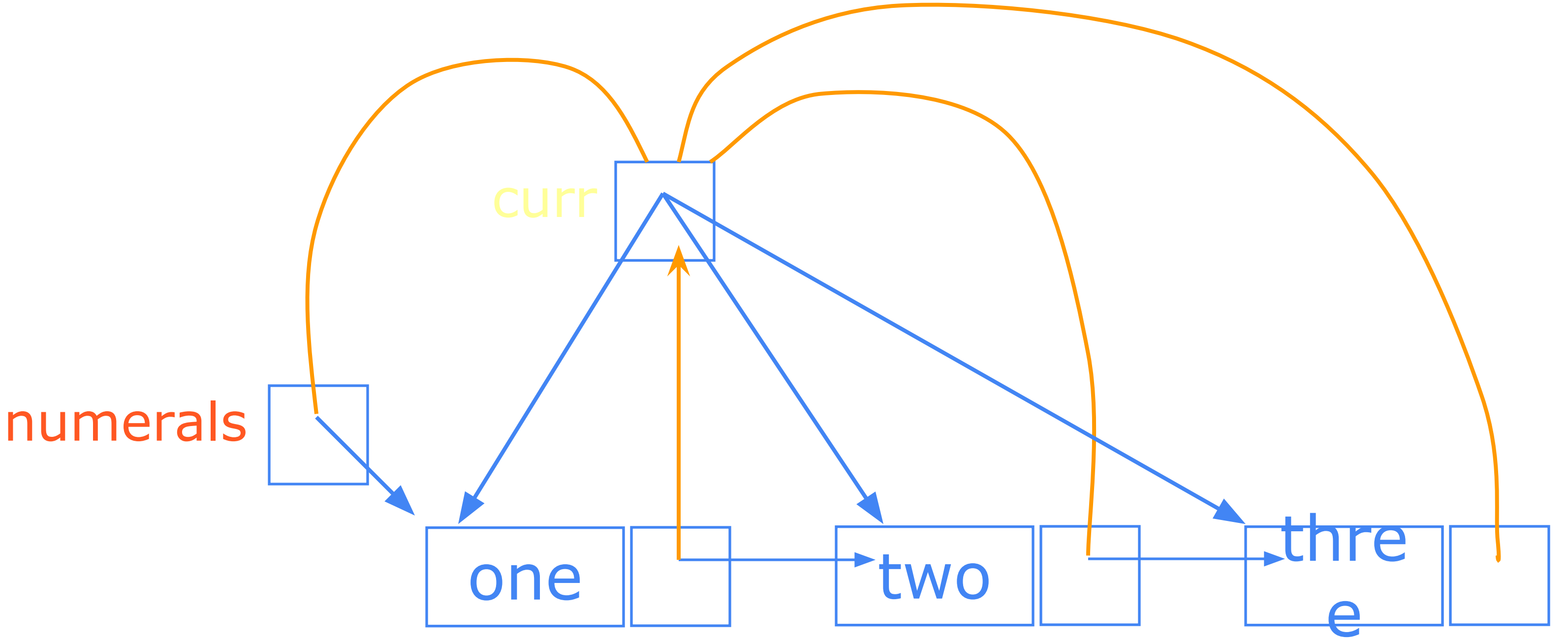
Traversing a SLL

- The following method traverses a list (and prints its elements):

```
public void printFirstToLast() {
   for (SLLNode current = first;
         current != null;
         current = curr.next) {
      System.out.print(curr.element + "  ");
   }
}
```

- You would write this as an instance method of the SLL class

Traversing a SLL (animation)



Inserting a node into a SLL

- There are many ways you might want to insert a new node into a list:
  - As the new first element
  - As the new last element
  - Before a given node (specified by a *reference*)
  - After a given node
  - Before a given value
  - After a given value
- All are possible, but differ in difficulty

Inserting as a new first element

- This is probably the easiest method to implement
- In class SLL (not SLLNode):

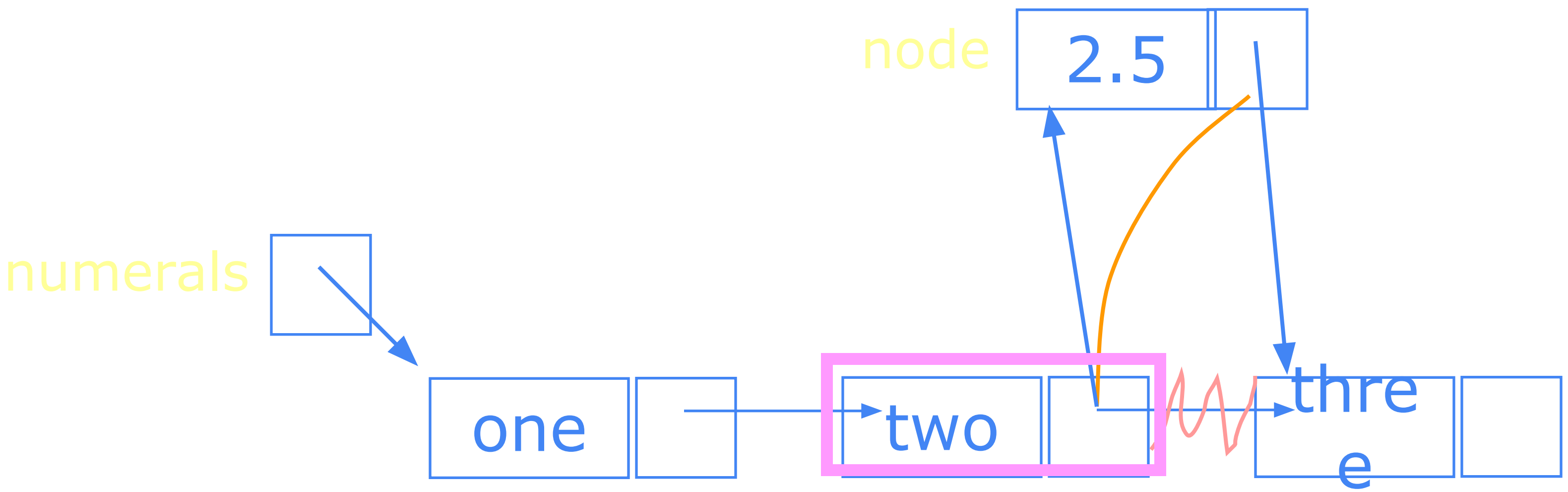
```
void insertAtFront(SLLNode node) {
 node.next = this.first;
 this.first = node;
}
```

- Notice that this method works correctly when inserting into a previously empty list

Inserting a node after a given value

```
void insertAfter(Object obj, SLLNode node) {
    for (SLLNode here = this.first;
            here != null;
            here = here.next {
        if (here.element.equals(obj)) {
            node.next = here.next;
            here.next = node;
            return;
        } // if
    } // for
    // Couldn't insert--do something reasonable!
}
```

Inserting after



Find the node you want to insert after

First, copy the link from the node that's already in the list

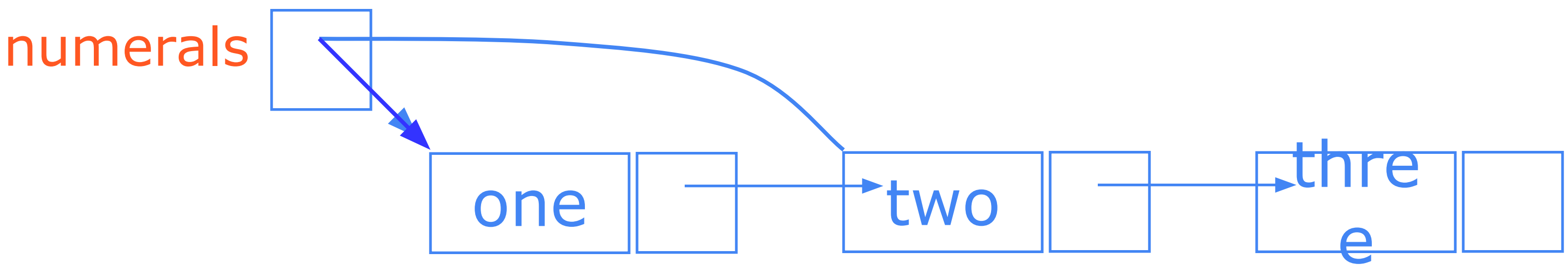
Then, change the link in the node that's already in the list

Deleting a node from a SLL

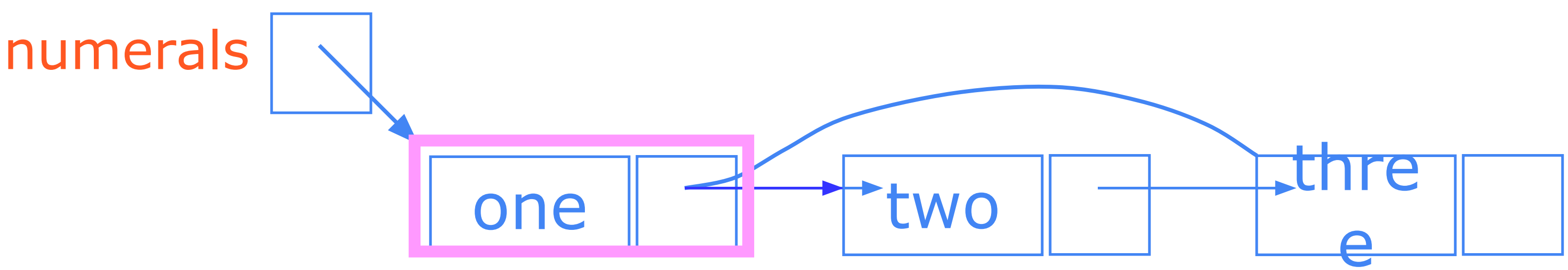
- In order to delete a node from a SLL, you have to change the link in its *predecessor*
- This is slightly tricky, because you can’t follow a pointer backwards
- Deleting the first node in a list is a special case, because the node’s predecessor is the list header

Deleting an element from a SLL

- To delete the first element, change the link in the header



- To delete some other element, change the link in its predecessor



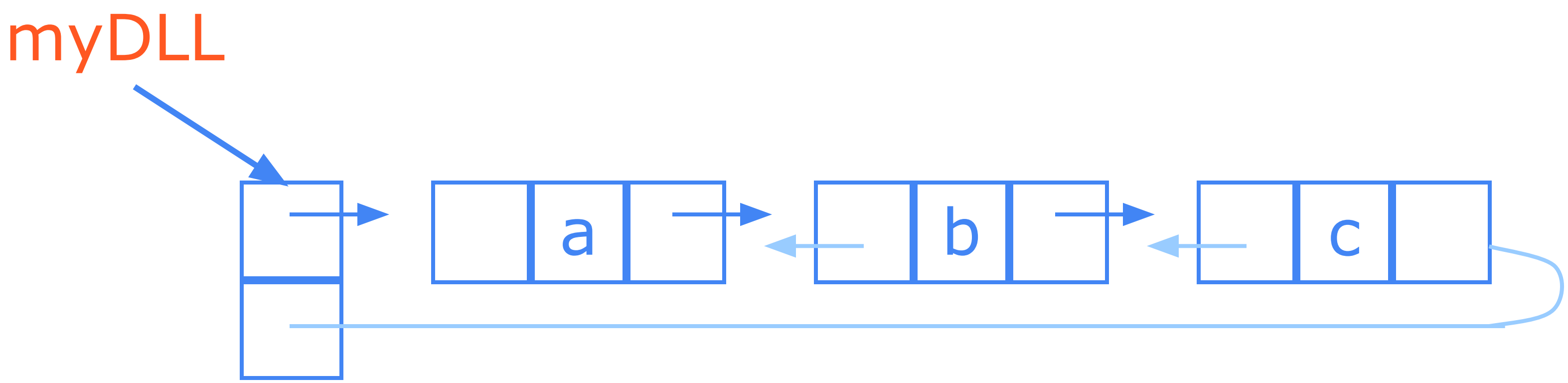
- Deleted nodes will eventually be garbage collected

Deleting from a SLL

```
public void delete(SLLNode del) {
    SLLNode succ = del.next;
    // If del is first node, change link in header
    if (del == first) first = succ;
    else { // find predecessor and change its link
        SLLNode pred = first;
        while (pred.succ != del) pred = pred.succ;
        pred.succ = succ;
    }
}
```

Doubly-linked lists

- Here is a doubly-linked list (DLL):



- Each node contains a value, a link to its successor (if any), and a link to its predecessor (if any)
- The header points to the first node in the list *and* to the last node in the list (or contains null links if the list is empty)

DLLs compared to SLLs

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

Constructing SLLs and DLLs (p. 74)

```java
public class SLL {

    private SLLNode first;

    public SLL() {
        this.first = null;
    }

    // methods...
}
```

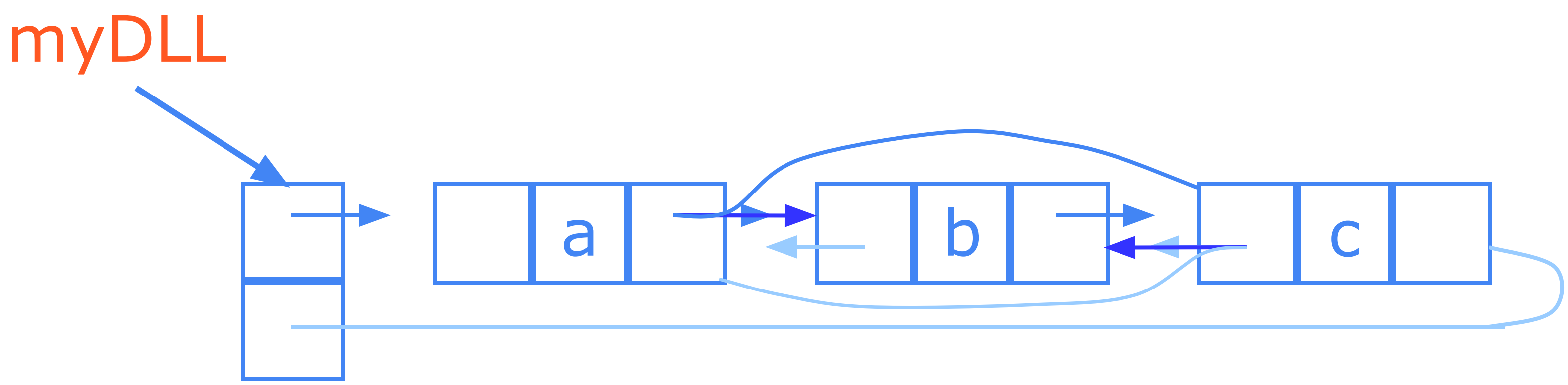
```java
public class DLL {

    private DLLNode first;
    private DLLNode last;

    public DLL() {
        this.first = null;
        this.last = null;
    }

    // methods...
}
```

DLL nodes in Java

```
public class DLLNode {
   protected Object value;
   protected DLLNode prev, next;

   protected DLLNode(Object value,
                     DLLNode prev,
                     DLLNode next) {
      this.value = value;
      this.prev = prev;
      this.next = next;
   }
}
```

Deleting a node from a DLL

- Node deletion from a DLL involves changing *two* links



- Deletion of the first node or the last node is a special case
- Garbage collection will take care of deleted nodes

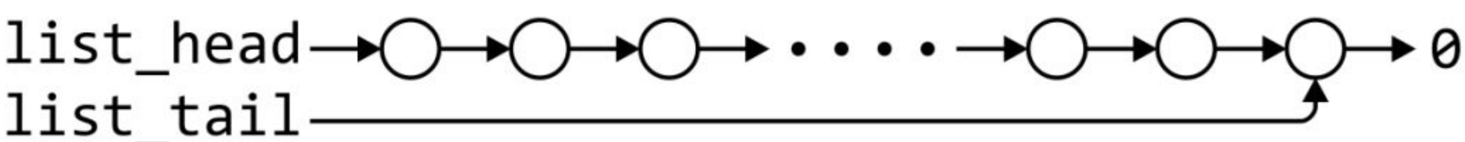
Other operations on linked lists

- Most "algorithms" on linked lists—such as insertion, deletion, and searching—are pretty obvious; you just need to be careful
- Sorting a linked list is just messy, since you can't directly access the n^{th} element—you have to count your way through a lot of other elements

Singly linked list

	Front/1^{st} node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$O(n)$	$\Theta(n)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$O(n)$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

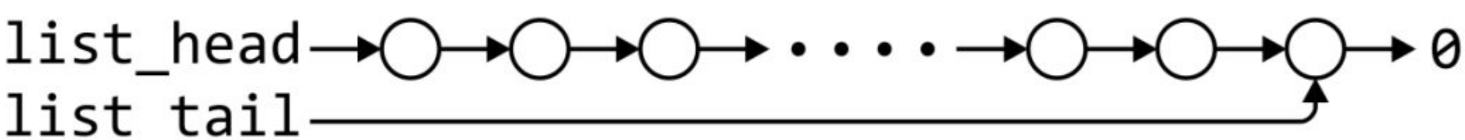
* These assume we have already accessed the k^{th} entry—an $O(n)$ operation



Singly linked list

	Front/1^{st} node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$\mathrm{O}(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$\mathrm{O}(n)$	$\Theta(n)$

By replacing the value in the node in question, we can speed things up
– useful for interviews

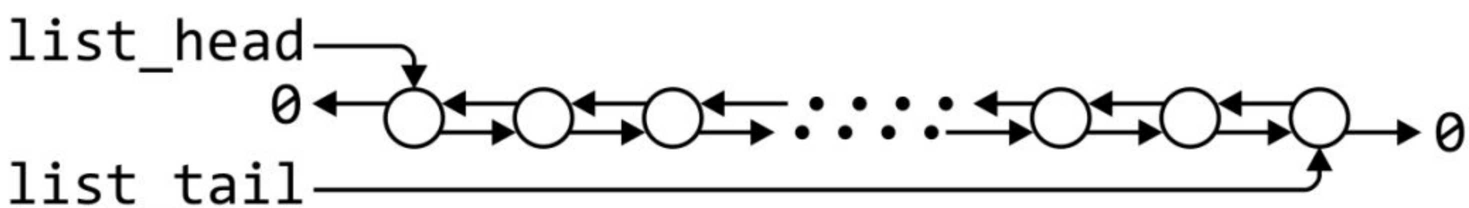


3.1.3.2

Doubly linked lists

	Front/1^{st} node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$\Theta(1)^*$	$\Theta(1)$

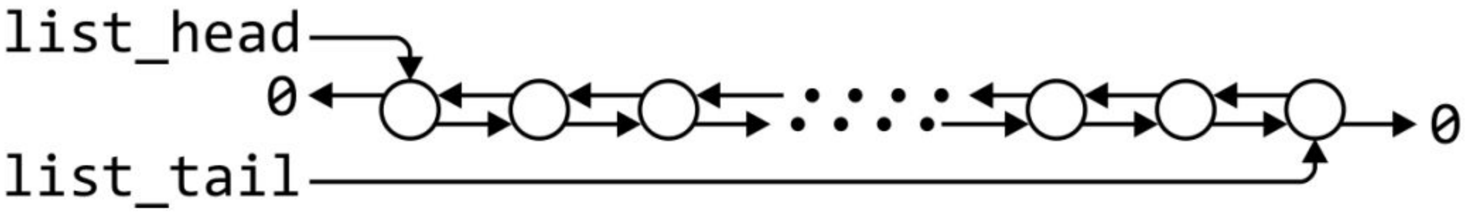
* These assume we have already accessed the k^{th} entry—an $O(n)$ operation



Doubly linked lists

Accessing the k^{th} entry is $\mathrm{O}(n)$

	k^{th} node
Insert Before	$\Theta(1)$
Insert After	$\Theta(1)$
Replace	$\Theta(1)$
Erase	$\Theta(1)$
Next	$\Theta(1)$
Previous	$\Theta(1)$



3.1.3.3

Other operations on linked lists

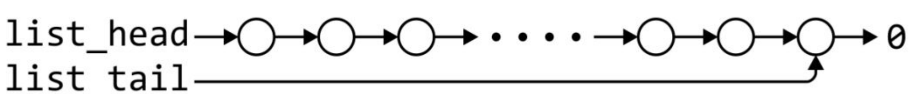
Other operations on linked lists include:

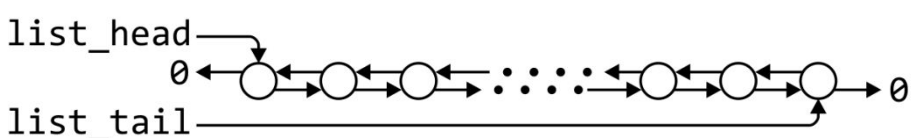
- Allocation and deallocating the memory requires $\Theta(n)$ time
- Concatenating two linked lists can be done in $\Theta(1)$
  - This requires a tail pointer

Run times

	Accessing the k^{th} entry	Insert or erase at the		
		Front	k^{th} entry	Back
Singly linked lists	$O(n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$ or $\Theta(n)$
Doubly linked lists				$\Theta(1)$
Arrays	$\Theta(1)$	$\Theta(n)$	$O(n)$	$\Theta(1)$
Two-ended arrays		$\Theta(1)$		

* Assume we have a pointer to this node





Data Structures

In general, we will only use these basic data structures if we can restrict ourselves to operations that execute in $\Theta(1)$ time, as the only alternative is $\mathbf{O}(n)$ or $\Theta(n)$

Interview question: in a singly linked list, can you speed up the two $\mathbf{O}(n)$ operations of

- Inserting before an arbitrary node?
- Erasing any node that is not the last node?

If you can replace the contents of a node, the answer is "yes"

- Replace the contents of the current node with the new entry and insert after the current node
- Copy the contents of the next node into the current node and erase the next node

Memory usage versus run times

All of these data structures require $\Theta(n)$ memory

- Using a two-ended array requires one more member variable, $\Theta(1)$, in order to significantly speed up certain operations
- Using a doubly linked list, however, required $\Theta(n)$ additional memory to speed up other operations

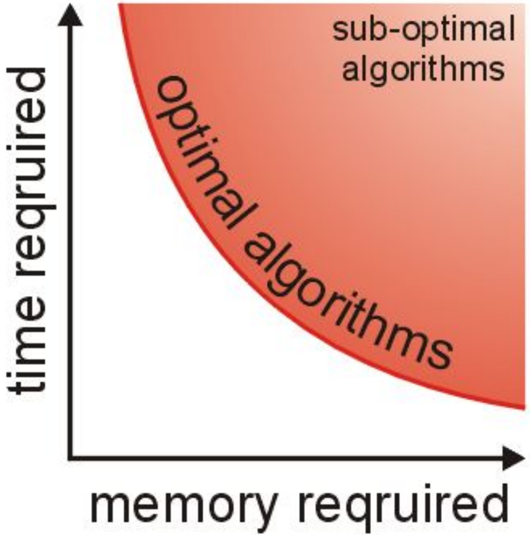
Memory usage versus run times

As well as determining run times, we are also interested in memory usage

In general, there is an interesting relationship between memory and time efficiency

For a data structure/algorithm:

- Improving the run time usually requires more memory
- Reducing the required memory usually requires more run time



Memory usage versus run times

Warning: programmers often mistake this to suggest that given any solution to a problem, any solution which may be faster must require more memory

This guideline not true in general: there may be different data structures and/or algorithms which are both faster and require less memory

- This requires thought and research

The End