Introduction to Data Structures and Algorithms (F28SG)

Lecture 5

Linked lists

Rob Stewart

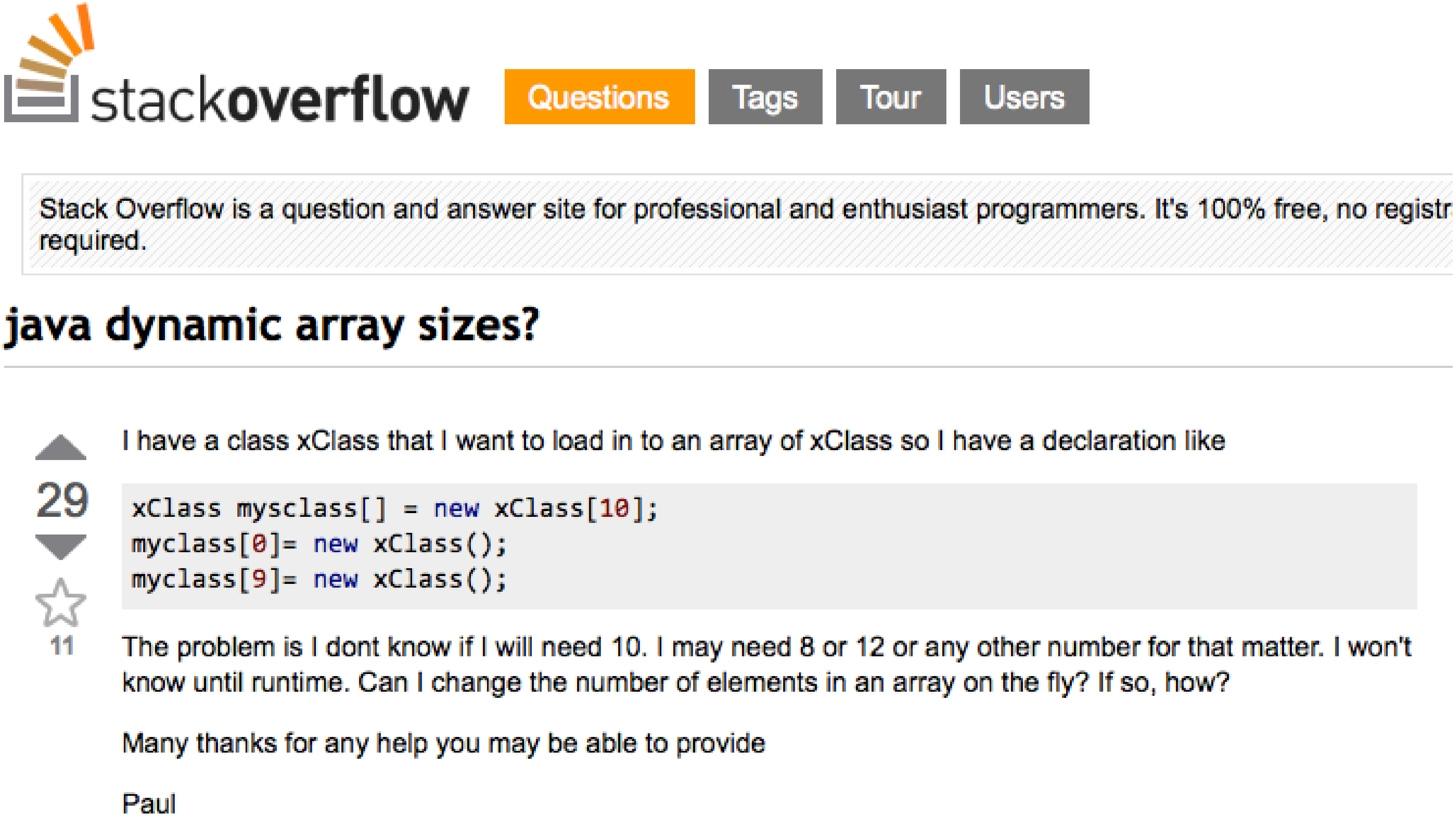
Outline

- By the end of the lecture you should
  - understand the concept of dynamic data structures
  - Understand object references in Java
  - Be familiar with linked lists

Arrays

- Last week we used arrays to implement stacks
- Many problems with arrays
- Harder to implement some ADTs efficiently
    - e.g. queues
  - Not possible to implement some at all
    - But one big issue in most languages….

Arrays



Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free, no registr required.

java dynamic array sizes?

I have a class xClass that I want to load in to an array of xClass so I have a declaration like

```
xClass mysclass[] = new xClass[10];
myclass[0]= new xClass();
myclass[9]= new xClass();
```

The problem is I dont know if I will need 10. I may need 8 or 12 or any other number for that matter. I won't know until runtime. Can I change the number of elements in an array on the fly? If so, how?

Many thanks for any help you may be able to provide

Paul

Problems with Arrays

- Array capacity is fixed
- **Exercise:** what is the growth rate using Big-O for this updated push operation for stacks

```
public void push(Object e){
  if(size()  == capacity){
      capacity *= 2;
      Object[] tmp = new Object[capacity];
      for(int i = 0; i <= top; i++)
         tmp[i] = S[i];
         S = tmp;}
S[++top] = e;}
```

Big-O: the Linear Function O(N)

- The linear function:

  $f(n) = n$

- For Big-O this means that

1. number of primitive operations increases at **the same rate** as the size of the input
2. Doubling the input size doubles the worst-case computational time complexity

- e.g. iterating over an array:

```
for (int i = 0; i < arr.length; i++) { .. }
```

Arrays

- Arrays are fine if you know how many elements needed
- Wasteful of space

```
Object[] arr = new Object arr[1000000000000000000];
```

- Many times we don’t know how many elements a data structure will store
- **Solution**: *Dynamic Data Structures*

Dynamic Data Structures

- Data structures that **dynamically** contract and expand as the program executes
  - Take up as much space as is necessary
    - and no more!
  - Won’t run out of space
    - (unless you run out of memory)
- We will cover two types
  - Linked Lists (lectures 6-11)
  - Trees (lectures 12-15)
- You need to get one important thing….

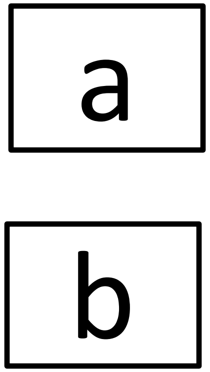
Java object references

Shared Object References

- Almost everything in Java is an Object
  - Excluding basic data types
    - int, float, double, char, boolean……..
- Need objects to be passed around efficiently
  - In space and time

Shared Object References

```
public class Number{
   public int value;
   public boolean isUsed;

  public static void main(String[] args){
            Number a,b;
            a = new Number();
            a.value = 12;
            b = new Number ();
            b.value = 10;

            a = b;
            b.value = 34;
            a.value = 20;
            System.out.println("The value of b is: ”+b.value);
            }
};
```

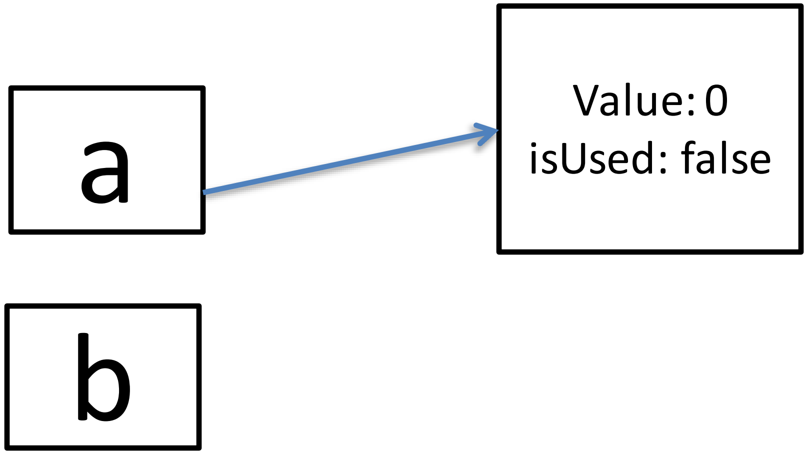


a

b

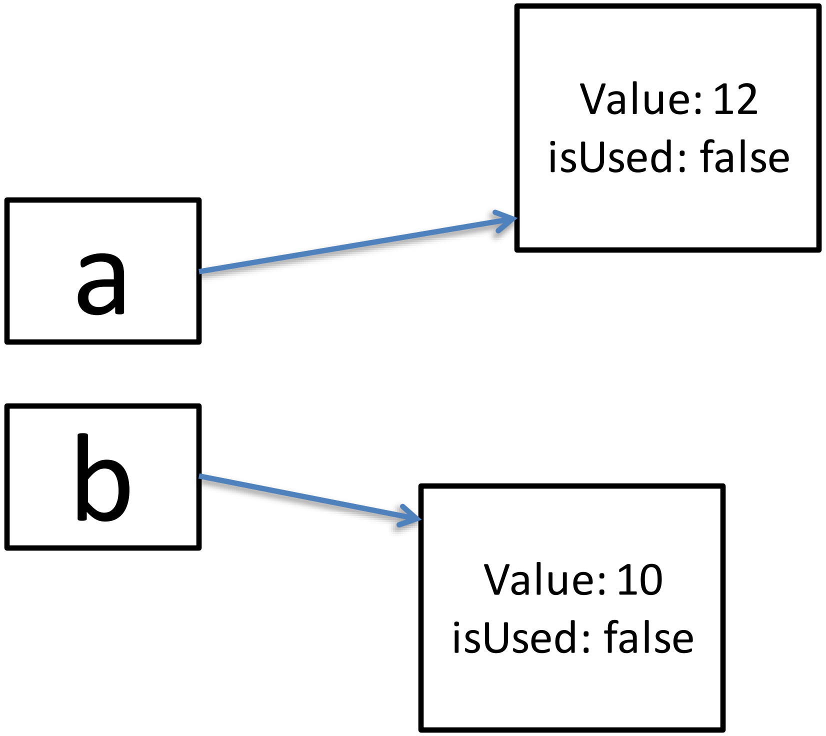
Shared Object References

```
public class Number{
   public int value;
   public boolean isUsed;

  public static void main(String[] args){
            Number a,b;
            a = new Number();
            a.value = 12;
            b = new Number ();
            b.value = 10;

            a = b;
            b.value = 34;
            a.value = 20;
            System.out.println("The value of b is: ”+b.value);
            }
};
```



Shared Object References

```
public class Number{
   public int value;
   public boolean isUsed;

  public static void main(String[] args){
             Number a,b;
             a = new Number();
             a.value = 12;
             b = new Number ();
             b.value = 10;

             a = b;
             b.value = 34;
             a.value = 20;
             System.out.println("The value of b is: ”+b.value);
             }
};
```



a

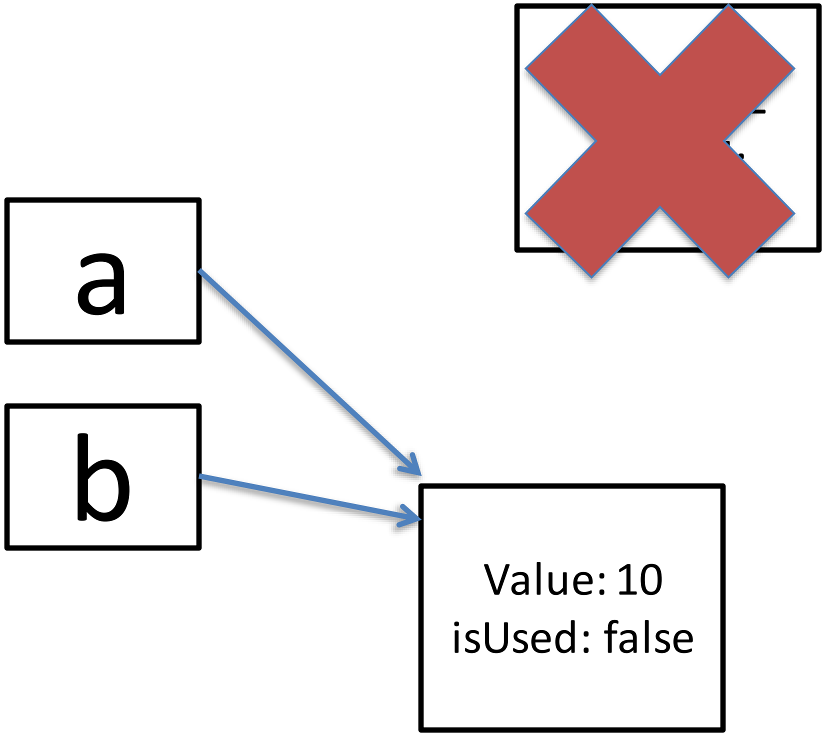
Value: 12
isUsed: false

b

Value: 10
isUsed: false

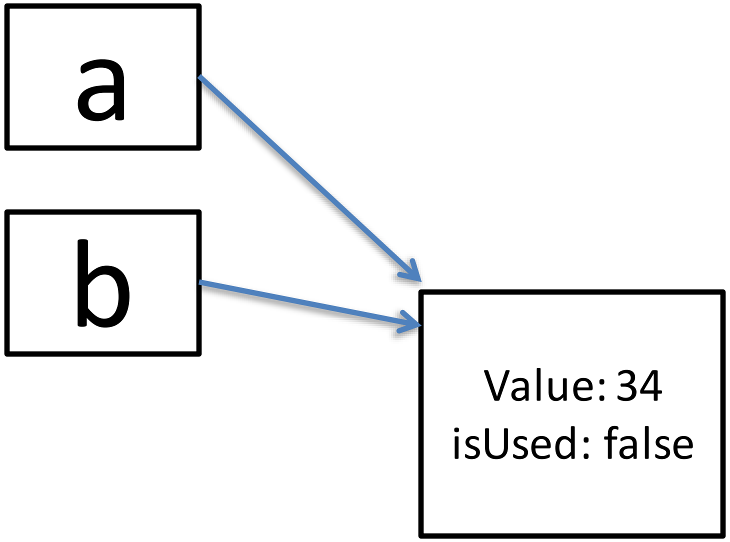
Shared Object References

```
public class Number{
   public int value;
   public boolean isUsed;

  public static void main(String[] args){
            Number a,b;
            a = new Number();
            a.value = 12;
            b = new Number ();
            b.value = 10;

            a = b;
            b.value = 34;
            a.value = 20;
            System.out.println("The value of b is: ”+b.value);
            }
};
```



Shared Object References

```
public class Number{
   public int value;
   public boolean isUsed;

  public static void main(String[] args){
           Number a,b;
           a = new Number();
           a.value = 12;
           b = new Number ();
           b.value = 10;

           a = b;
           b.value = 34;
           a.value = 20;
           System.out.println("The value of b is: ”+b.value);
           }
};
```



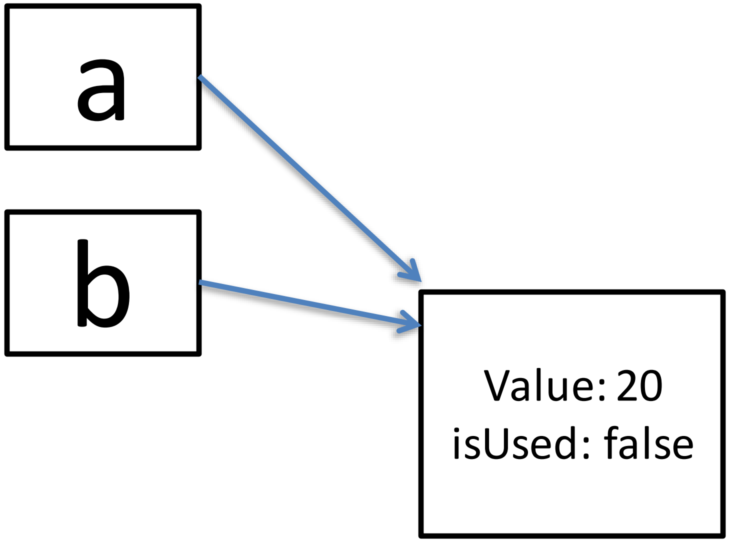
a

b

Value: 34
isUsed: false

What number is printed (held in b.value)?

```
public class Number{
   public int value;
   public boolean isUsed;

  public static void main(String[] args){
            Number a,b;
            a = new Number();
            a.value = 12;
            b = new Number ();
            b.value = 10;

            a = b;
            b.value = 34;
            a.value = 20;
            System.out.println("The value of b is: ”+b.value);
            }
};
```



a

b

Value: 20
isUsed: false

Solution

20

Why?

Try for yourself

object-reference-demo: `src/Number.java`

Shared Object References

- Object o;
  - Declares a variable of type Object
  - Only allocates enough space to hold a memory address
  - o holds a **reference**
    - In this case a null reference
- o = **new** Object();
  - Allocates memory space to hold an instance of type Object
  - Returns the address of that memory location
  - and stores it in the variable o
    - Hence the assignment operator (=)
- o is not the Object, but tells us where the object lives

Shared Object References

- When you pass o around
  - You are not giving the actual object
    - but a reference to it
- Object p = o;
  - p is assigned the **same object reference** as o
  - They point to the same object
    - i.e. they point to data in the **same memory location**
  - It **does not** copy the object

Shared Object References

- Good News
  - Java does all this for you
- Bad News
  - If you don’t understand it….
  - Then it will bite you at some point…
- **Referencing is the most important thing you need to know to understand Dynamic Data Structures**

Dynamic Data Structures (DDS)

- DDS are formed by exploiting object references
  - to allow an object to
  - **reference** another object
  - of the **same class**.
- This works recursively
- *Recursive algorithms are good*

Dynamic Data Structures (DDS)

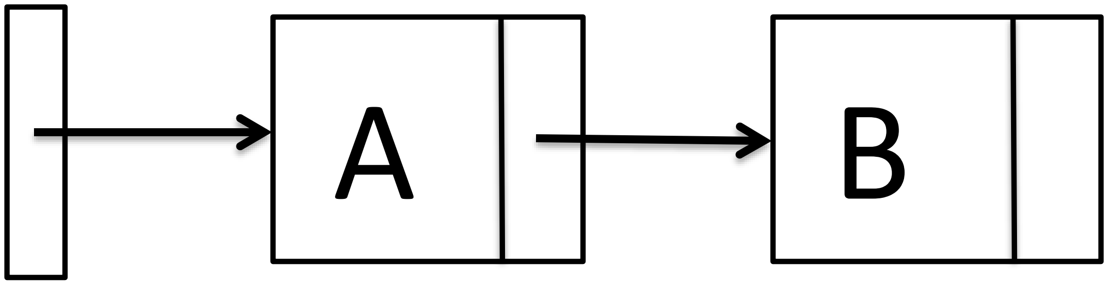
- DDS can grow and shrink in size as required
- Efficient in space and in operations

```
Collection c = new Collection();
```

	Empty collection	**Large collection**	**Read/write access**
Static Arrays	Large space cost	Large space cost	Very fast
Dynamic linked lists	**Tiny space cost**	Large space cost	**It depends**

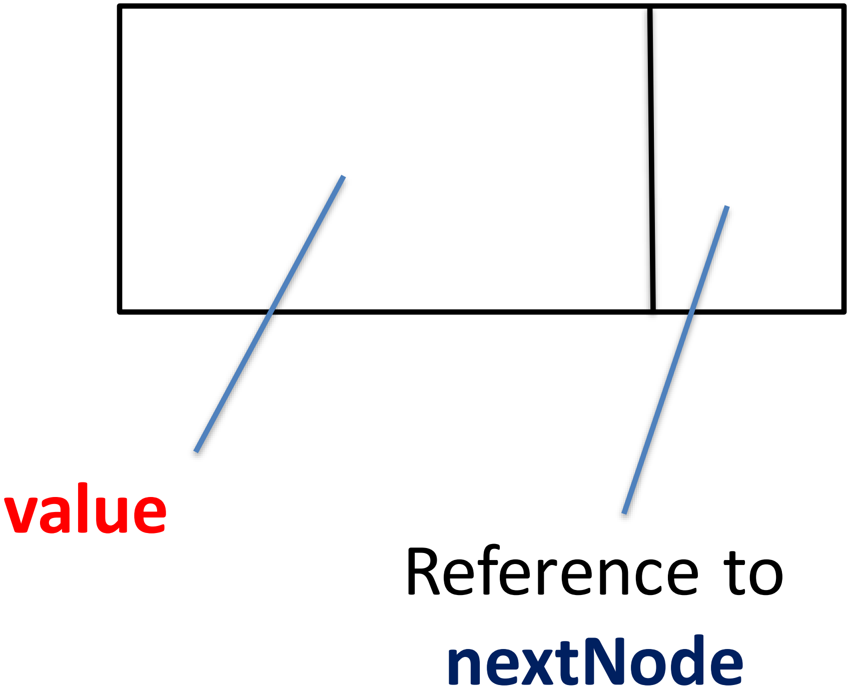
Linked Lists

- Linked List is **linearly** ordered sequence of Nodes
- We can step along the sequence to access the value in each node



Nodes

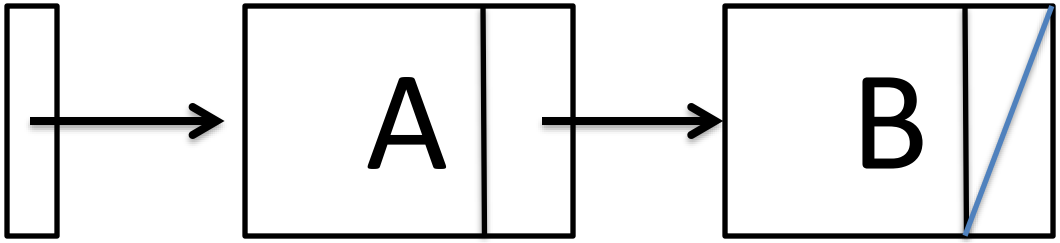
```
public class Node{
  public Object value;
  public Node nextNode;

  public Node(Object val){
          value = val;
          nextNode = null;
  }
};
```



Terminology

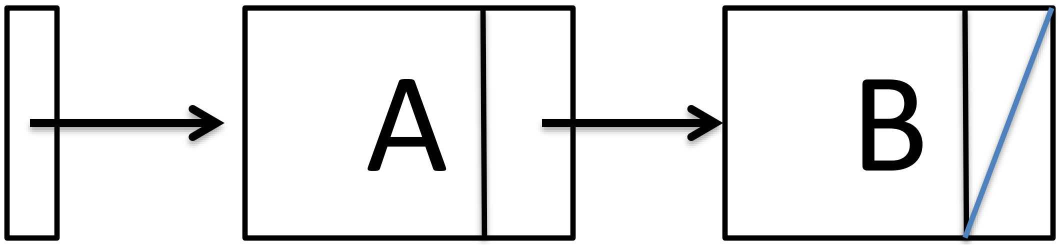
- All DDS are formed from Nodes
  - Formed from
    - a **Value** and
    - (at least) one **reference** to the **same type of Object**
  - Values can be anything
    - Values of type int or String, or another object
    - Usually need some sort of comparison function
  - **Reference**(s) allow us to get **to the other Nodes**
    - means we can't randomly access values like arrays
    - Nodes have no idea where they are in the Data Structure

Nodes



- This is how we diagrammatically link nodes
  - There is always **1 root node**
    - this is our way in
  - Nodes are only **aware** of **other Nodes** whose **references they hold**
- We can’t jump from the root node to B without going through A

Nodes



- This is how we diagrammatically link nodes
  - We can only go in the direction of the arrows
    - A holds a reference to B
    - B doesn't hold a reference to A
  - A Node without sibling has its reference set to **null**
    - We need to be explicit and careful about this in code
    - This represents the end of dynamic data structure

Dynamic Data Structures (DDS)

- Congratulations! You now know everything about Dynamic Data Structures. Honestly!
- Three common data structures
  - **Linked-Lists, Trees** and **Graphs**
  - Many variations of each
- The only difference between them is:
  1. the number of references each Node holds
  2. how Nodes can be linked to each other.
- This is called the **Topology**

Node traversal

Three exercises

1. Get last node, print its reference and value
2. Print list node values with a loop
3. Print list node values with recursion

singly-linked-list-demo project

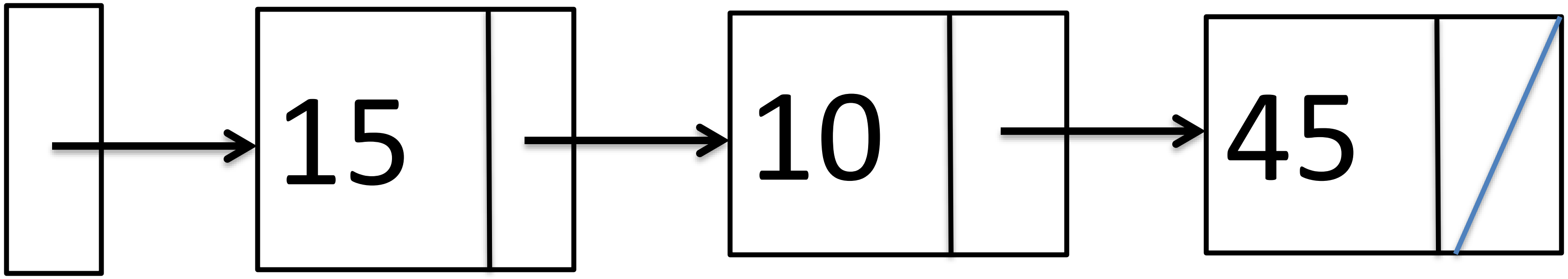
```
src/NodeTraversals.java
```

Eclipse demonstration

Example:

Print out the contents of a linked list

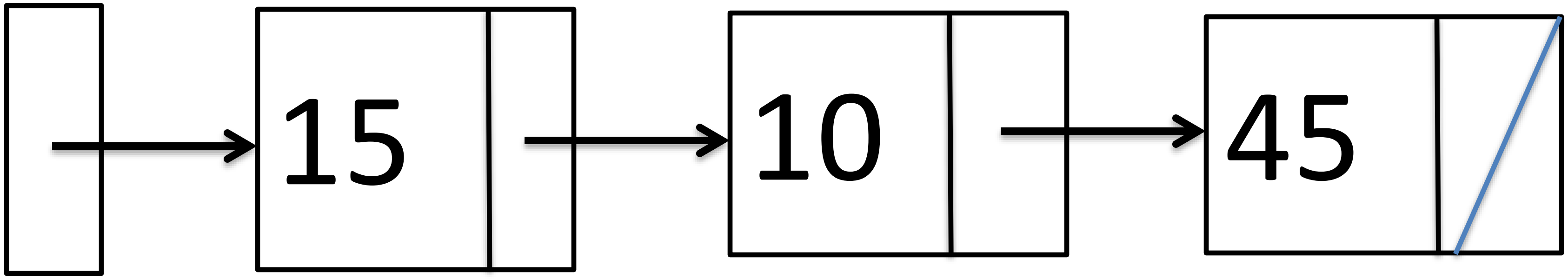
```
public class Node {
  public int value;
  public Node nextNode;

  public Node(int val){
    value = val;
    nextNode = null; //always do this
  }

  public static void main(String[] args){
           Node root; //the head nodede




}}
```



Example:

Print out the contents of a linked list

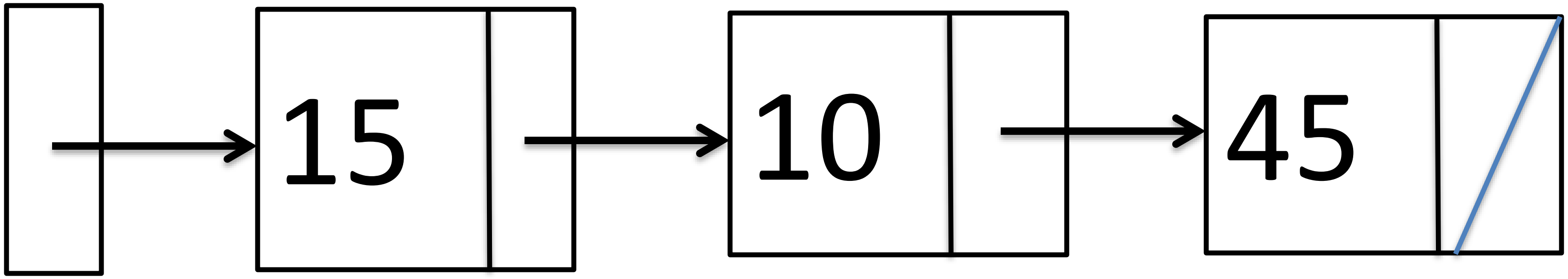
```
public class Node {
  public int value;
  public Node nextNode;

  public Node(int val){
    value = val;
    nextNode = null; //always do this
  }

  public static void main(String[] args){
             Node root; //the head node
             Node currentNode = root;
            // work through the linked list and print off each value in turn
            while (currentNode != null){


            }
}}
```



Example:

Print out the contents of a linked list

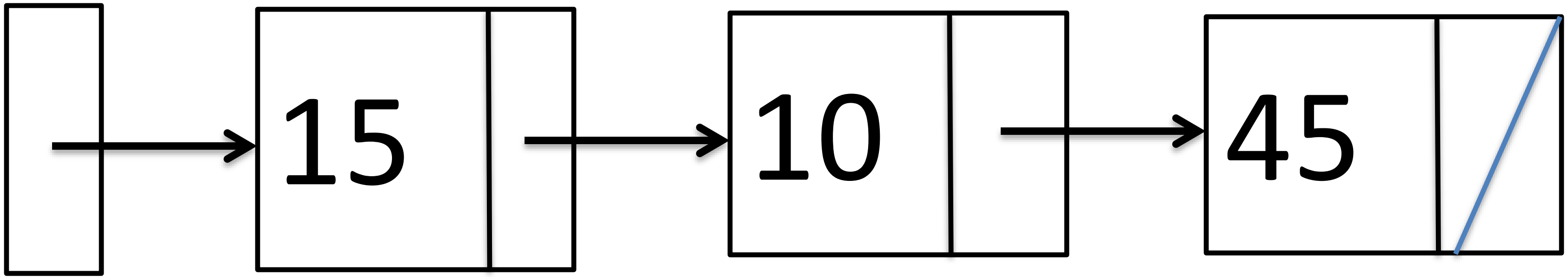
```
public class Node {
  public int value;
  public Node nextNode;

  public Node(int val){
    value = val;
    nextNode = null; //always do this
  }

  public static void main(String[] args){
              Node root; //the head node
              Node currentNode = root;
             // work through the linked list and print off each value in turn
             while (currentNode != null){
                          System.out.print(currentNode.value+" ");

             }
}}
```



Example:

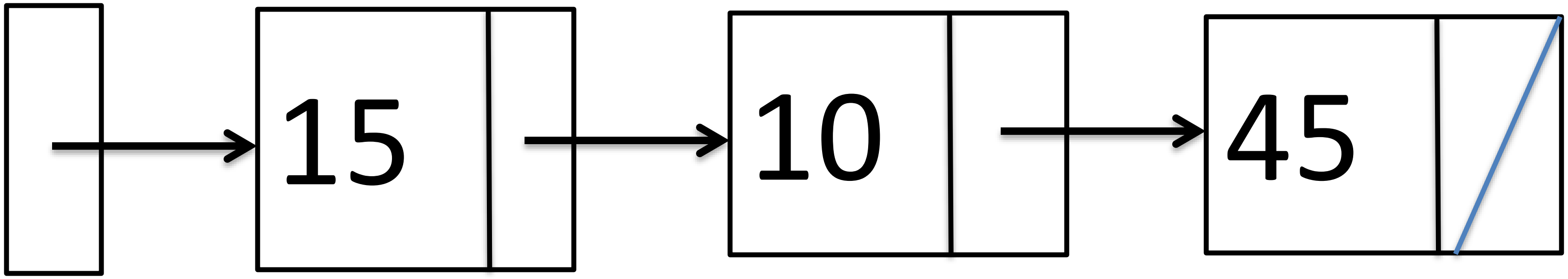
Print out the contents of a linked list

```
public class Node {
  public int value;
  public Node nextNode;

  public Node(int val){
    value = val;
    nextNode = null; //always do this
  }

  public static void main(String[] args){
               Node root; //the head node
               Node currentNode = root;
              // work through the linked list and print off each value in turn
              while (currentNode != null){
                              System.out.print(currentNode.value+" ");
                              currentNode = currentNode.nextNode;
              }
}}
```



```
public class Node {
  public int value;
  public Node nextNode;

  public Node(int val){
    value = val;
    nextNode = null; //always do this
  }

  public static void main(String[] args){
            Node root; //the head node
            Node currentNode = root;

            //  your code goes here

}}
```

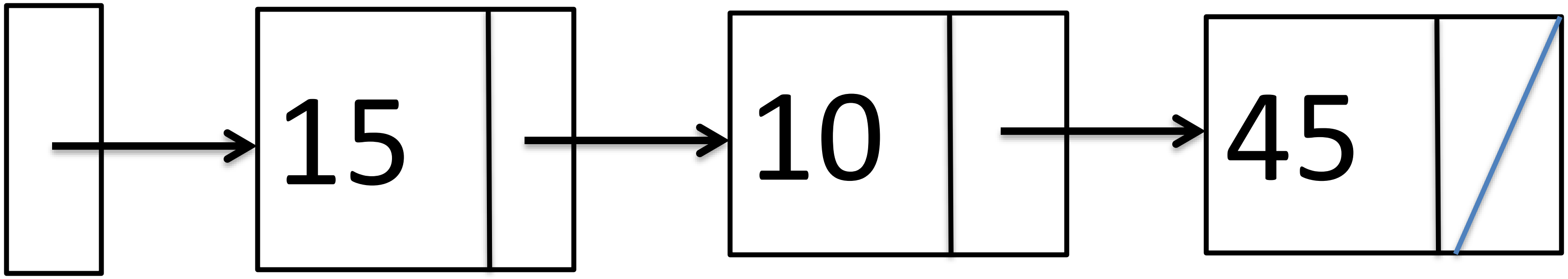
<u>Exercise</u>: Make currentNode point to last element of linked list



```
public class Node {
  public int value;
  public Node nextNode;

  public Node(int val){
    value = val;
    nextNode = null; //always do this
  }

  public static void main(String[] args){
            Node root; //the head node
            Node currentNode = root;

            If (currentNode != null) {
                while (currentNode.nextNode != null){
                        // your code here
                }
            }
}}
```

SOLUTION:

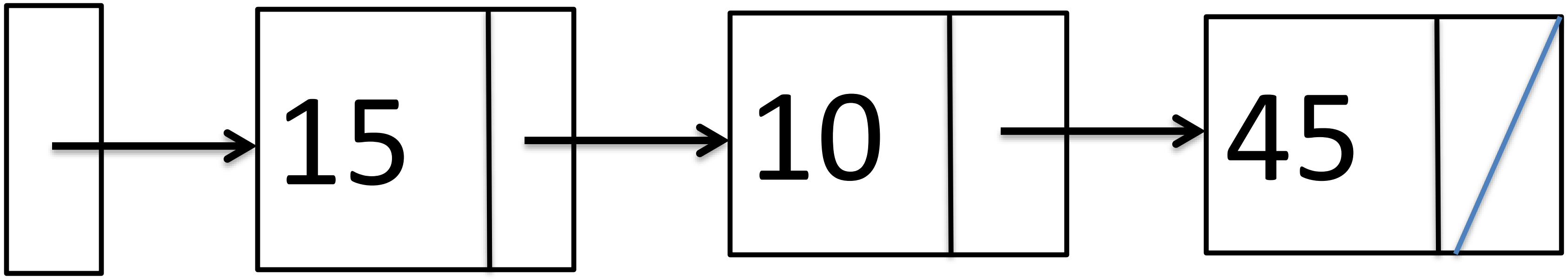
Make currentNode point to last element of linked list



```
public class Node {
  public int value;
  public Node nextNode;

  public Node(int val){
    value = val;
    nextNode = null; //always do this
  }

  public static void main(String[] args){
            Node root; //the head node
            Node currentNode = root;

            If (currentNode != null) {
                while (currentNode.nextNode != null){
                            currentNode = currentNode.nextNode;
                }
            }
}}
```

SOLUTION:

Make currentNode point to last element of linked list



Recursion

```
public class Node {
  public int value;
  public Node nextNode;
…
```

- All Dynamic Data Structures are **recursive**
  - Removing Nodes from a Linked List
    - just leaves a much simpler Linked List
  - Recursive Algorithms work really well here
    - fits with our logical model of the DDS.
  - E.g.

```
root.printList();
```

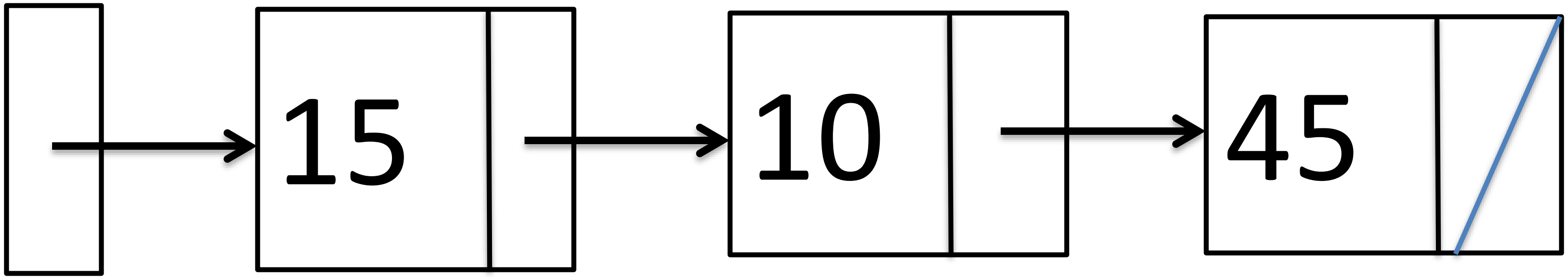
```
public void printList(){
		System.out.print(value + " ");
		if(nextNode != null){
			nextNode.printList();
		}
}
```

```
public class Node {
 public int value;
 public nextNode;

 public Node (int val){
  value = val;
  nextNode = null; //always do this
 }

public static int lastValue(Node l){
   // your code goes here
 }

 public static void main(String[] args){
        Node root; //the head node
        Node currentNode = root;
        int lastNumber = lastValue(currentNode);
        System.out.println(lastNumber);
}
```

Exercise:

Make recursive version to get the last element of linked list

Demo in Eclipse



Exercise

Given the linked list: { A , B , C , E , F }

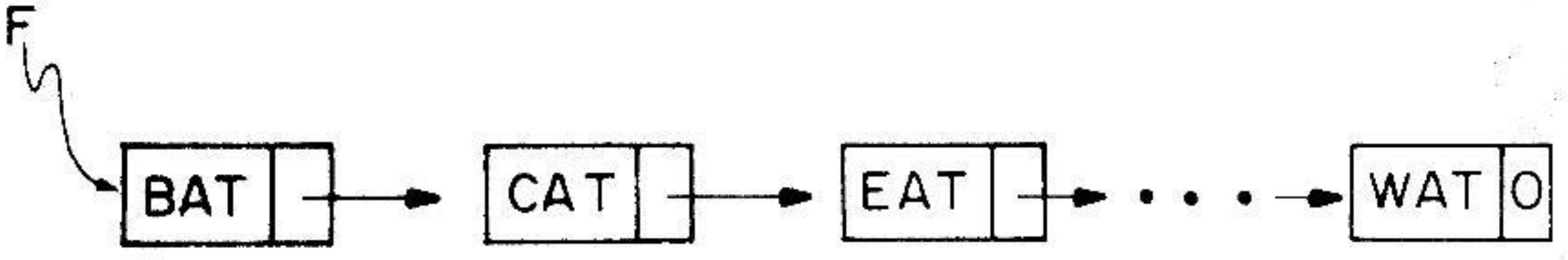
1. Insert D between elements C and E

2. Remove element E

Preview into the next lecture

LINKED LISTS: INSERT AND DELETE

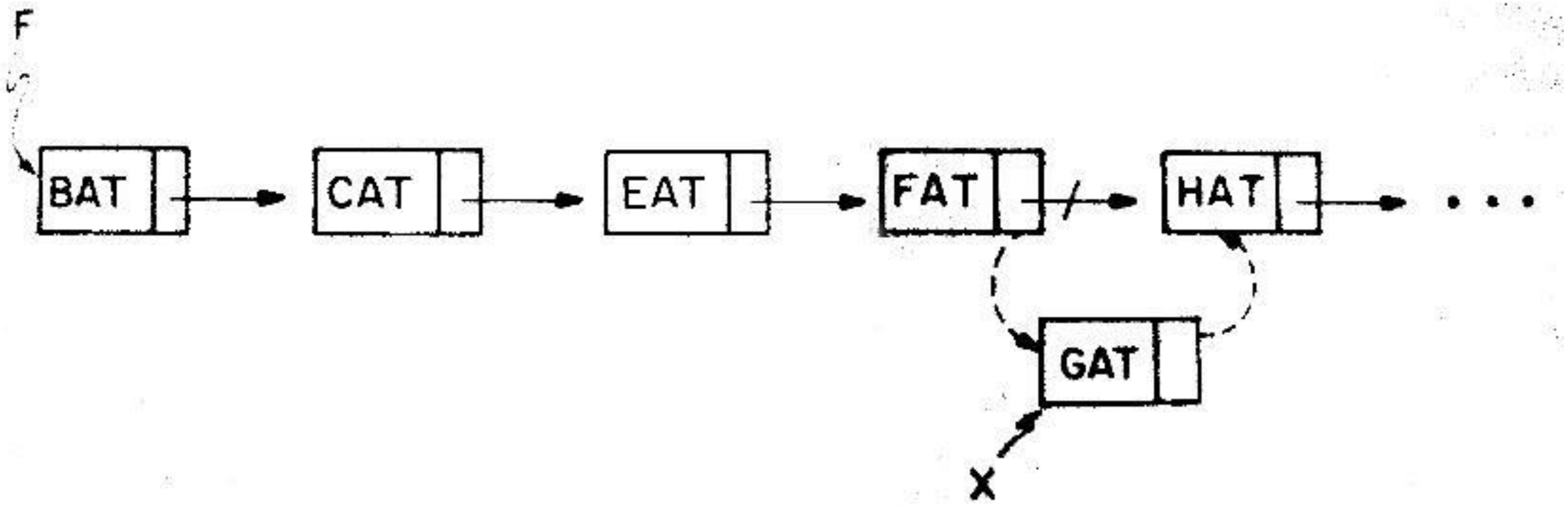
Linked List Operations



Fundamentals of Data Structures, E Horowitz & S Sohni, 2007.

Linked List Operations

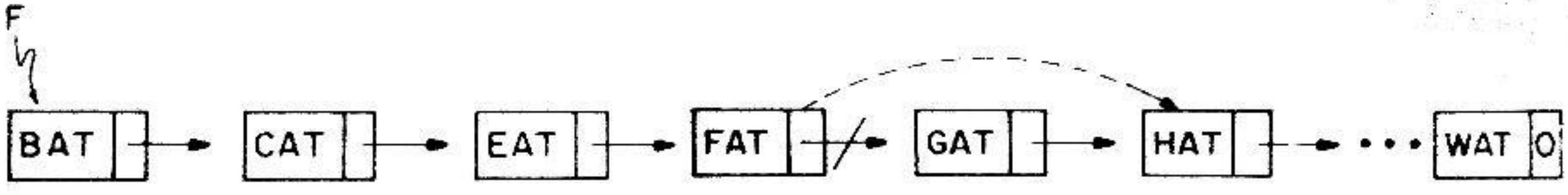
Insertion:



Fundamentals of Data Structures, E Horowitz & S Sohni, 2007.

Linked List Operations

Deletion:



Fundamentals of Data Structures, E Horowitz & S Sohni, 2007.

Summary

- Dynamic Data Structures
  - Can grow and contract
  - Are efficient in shuffling data around
- DDS also have some limitations
  - Not randomly accessible
  - Need to step through in order
- **Next lecture:** operations on dynamic linked lists