Birkbeck
UNIVERSITY OF LONDON

Introduction to Software Development: Data Structures

December 2009

Outline of this lecture

1. References to Objects
2. Linked lists of Objects
3. Stacks
4. Queues
5. Trees

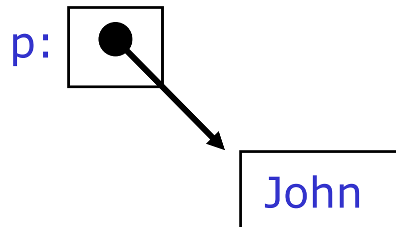
1. References to objects

- In Groovy, the keyword new creates a new object, and also returns a ***reference*** to that object: a reference is just an “address” (i.e. location) in the computer’s memory indicating where the new object has been stored, so that we can then do further things with it (languages like C and C++ support “pointers” rather than references)
- E.g. if a class Person has been defined, with a field String name, if we say:

```
def p = new Person("John“)
```

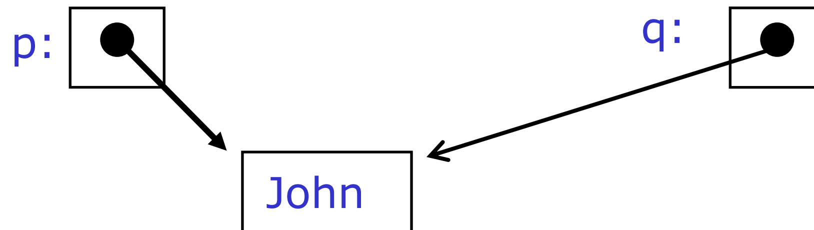
new Person("John") creates a new Person object and returns a reference to this object;

def p = … assigns this reference to the variable p (which will then have type Person):

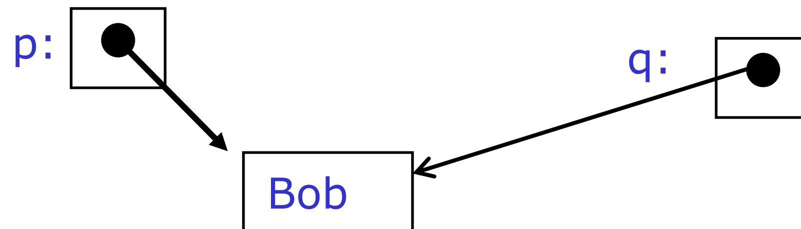


What can we do with references?

- The operations available on references include:
  - copy e.g. def q = p results in:

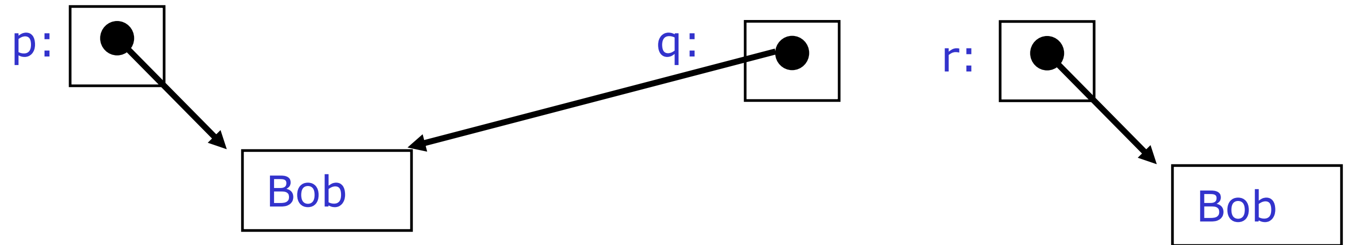


- dereference ("follow") e.g. p.name = "Bob" results in:



What can we do with references?

- Compare for equality e.g. given



- p.name == q.name returns true (first dereferences and then compares the two strings)
- p.name == r.name returns true (first dereferences and then compares the two strings)
- p == q returns true (dereferences and compares the objects: p and q point to the same object)
- p == r returns false (dereferences and compares the objects: p and r point to different objects)

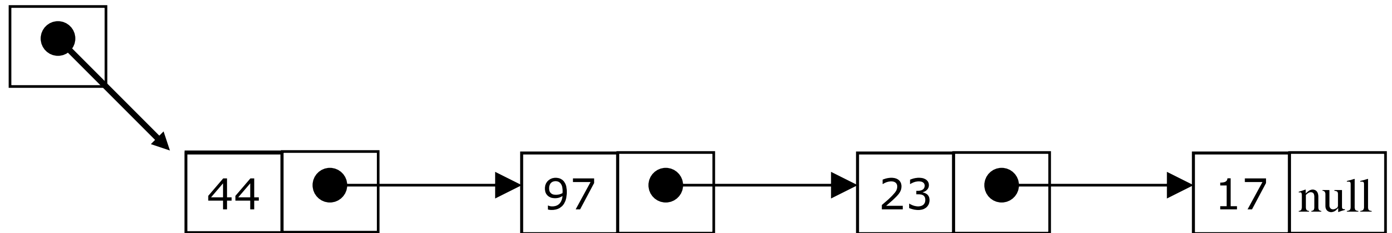
2. Linked Lists of Objects

- We can use references to create linked lists of objects in Groovy
- Let’s define a class called Cell:

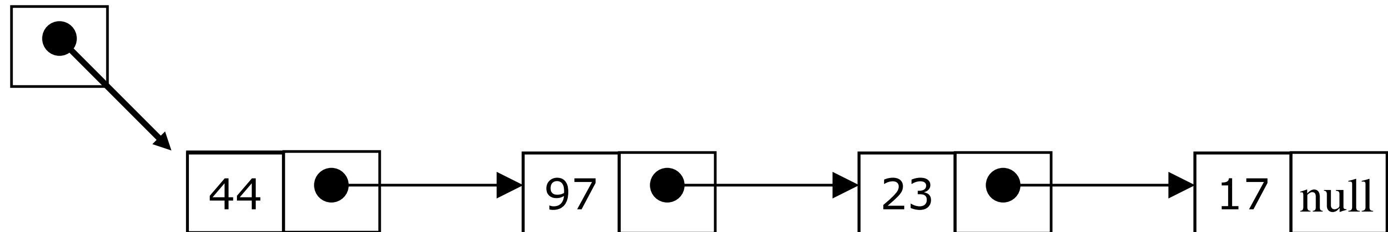
```
class Cell {
        int value;
        Cell next;

        Cell (int v, Cell c) {
                value = v;
                next = c;
        }
}
```

Example of a linked list of numbers

How can we create this linked list?



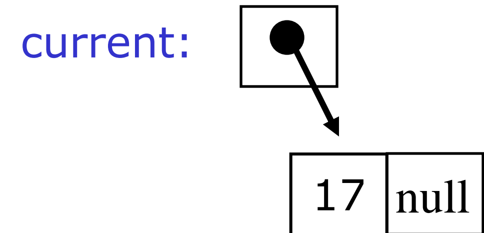
Example of a linked list of numbers



One way is starting from the end and working backwards i.e. in the *reverse* direction of the references:

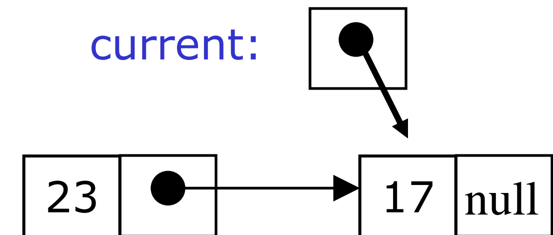
```
current = new Cell(17, null)
current = new Cell(23, current)
current = new Cell(97, currrent)
current = new Cell(44, current)
myList = current
```

starting from the end and working backwards:



current = new Cell(17, null)

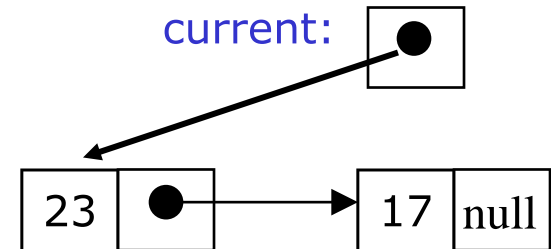
starting from the end and working backwards:



just this part of the next statement:

new Cell(23, current)

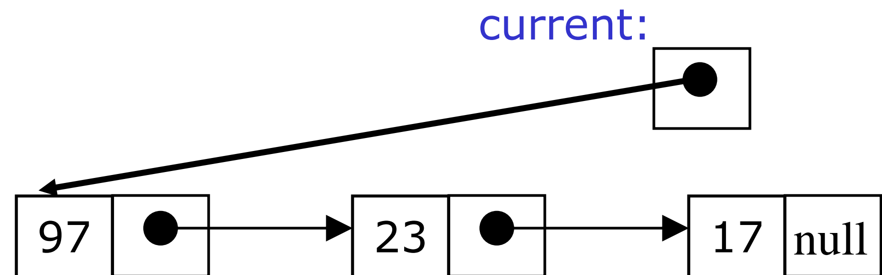
starting from the end and working backwards:



with the whole statement:

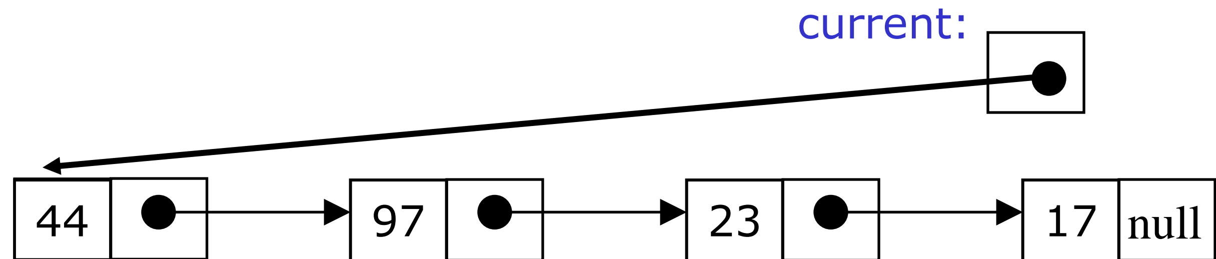
current = new Cell(23, current)

starting from the end and working backwards:



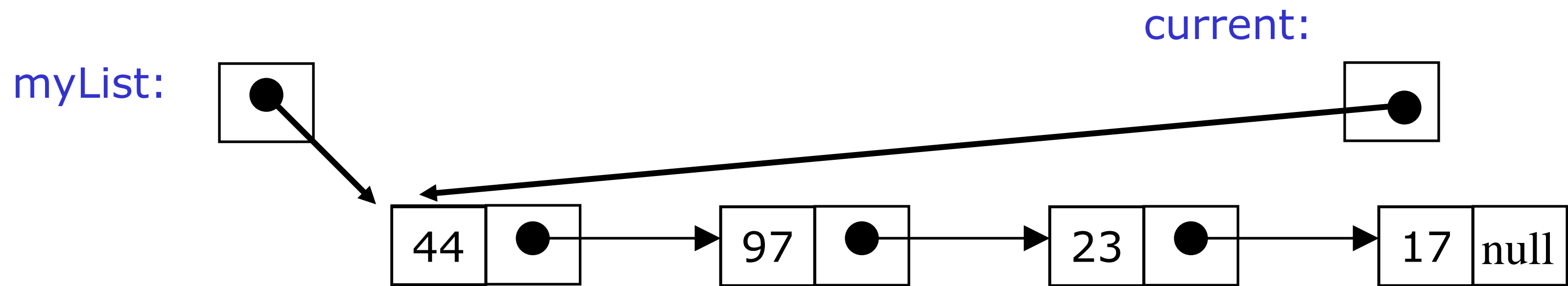
current = new Cell(97, currrent)

starting from the end and working backwards:



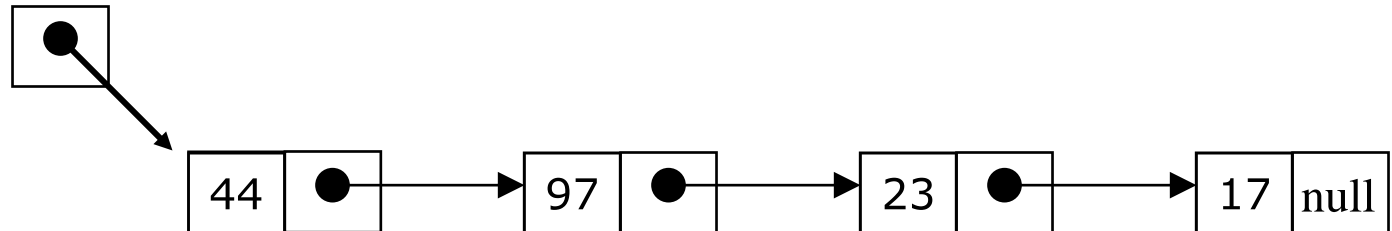
current = new Cell(44, current)

finally:



myList = current

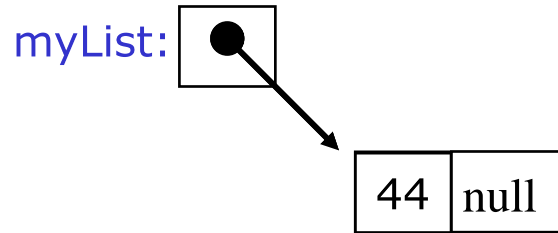
Example of a linked list of numbers



We can also create the list starting from the front and working forwards (i.e. in the *same* direction as the references):

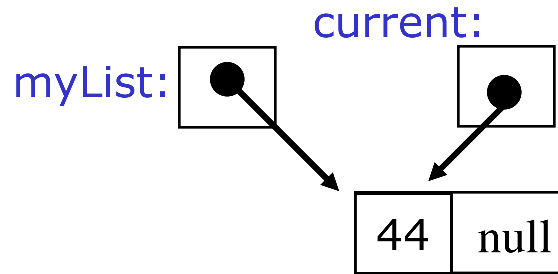
```
myList = new Cell(44,null)
current = myList
current.next = new Cell(97,null)
current = current.next
current.next = new Cell(23,null)
current = current.next
current.next = new Cell(17, null)
```

starting from the front and working forwards:



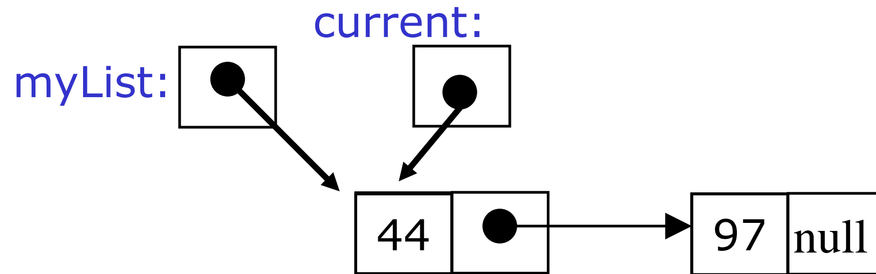
myList = new Cell(44,null)

starting from the front and working forwards:



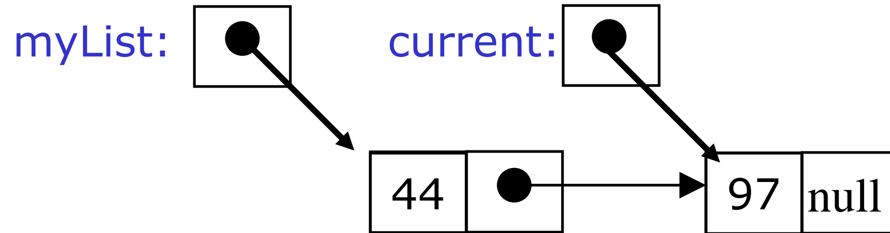
current = myList

starting from the front and working forwards:



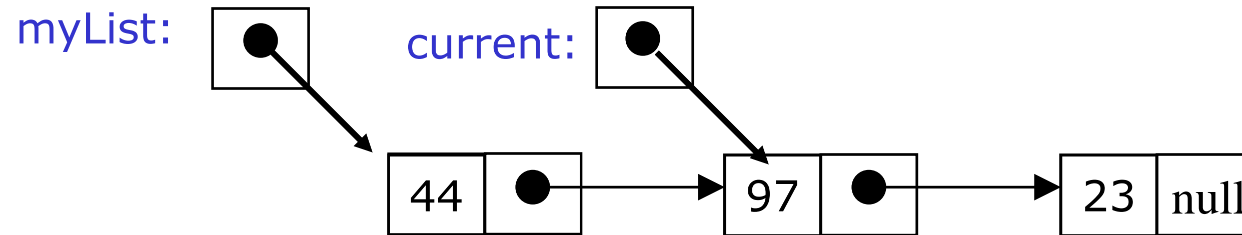
current.next = new Cell(97,null)

starting from the front and working forwards:



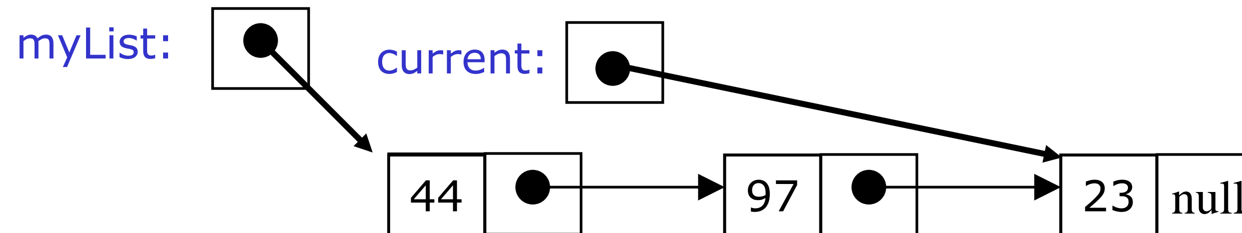
current = current.next

starting from the front and working forwards:



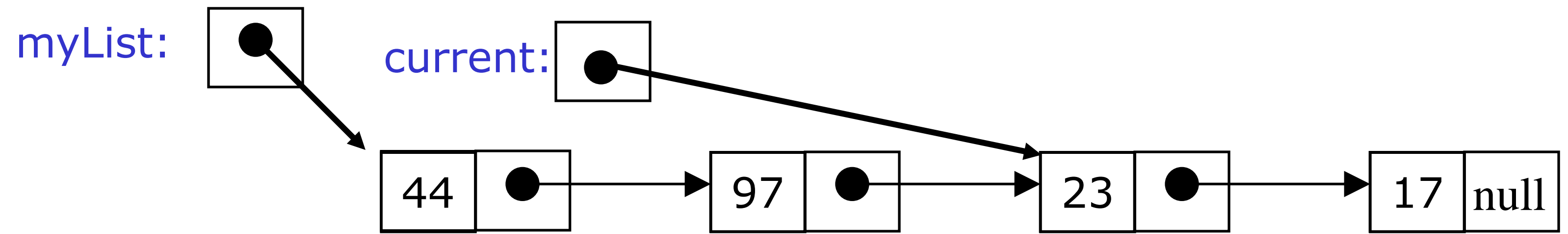
current.next = new Cell(23,null)

starting from the front and working forwards:



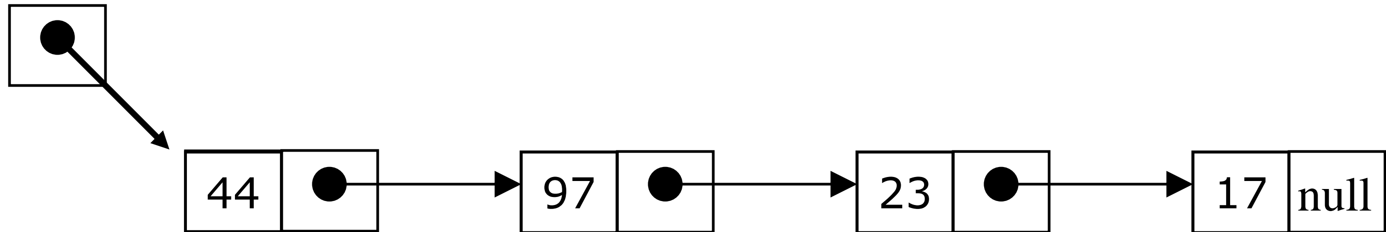
current = current.next

finally:



current.next = new Cell(17, null)

Traversing a linked list



If we define this function:

```
def print(Cell t) {
    if (t == null) return
    println(t.value)
    print(t.next)
}
```

We can print the list's contents by saying: print(myList)

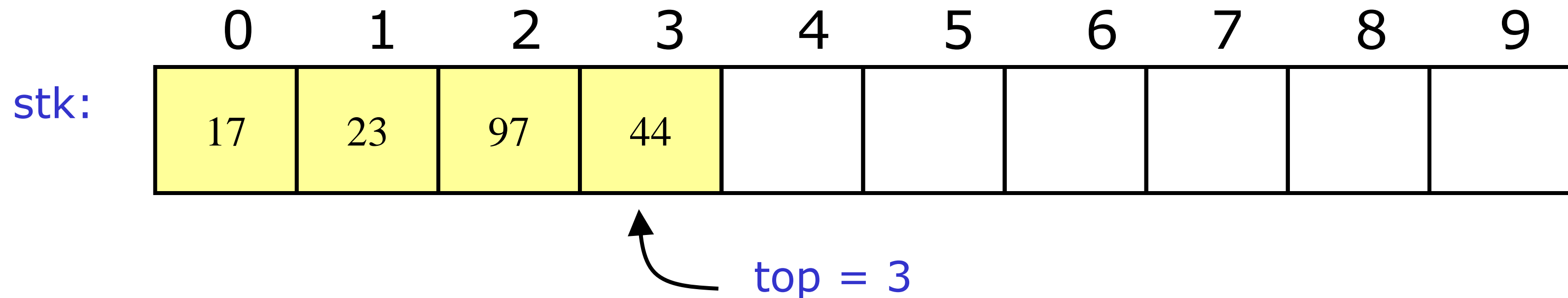
3. Stacks

- A stack is a last in, first out (LIFO) data structure:
  - items are removed from a stack in the reverse order to the way they were inserted
- This kind of data structure is useful for programming language implementation, e.g.
  - for parsing expressions
  - to support calls to methods/functions/procedures.

(i) Array implementation of stacks

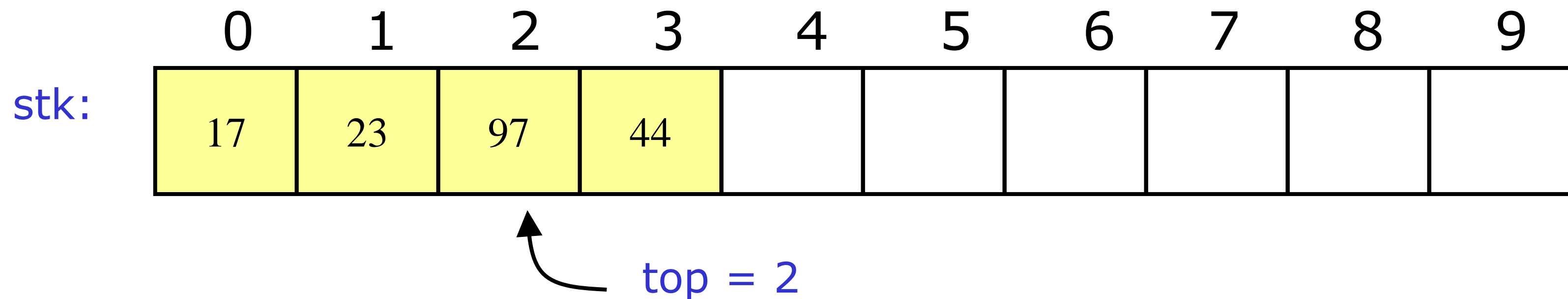
- To implement a stack, items are inserted and removed at the same end (called the top)
- To use an array to implement a stack, you need both the array itself and an integer
- The integer tells you which location is currently the top of the stack

Pushing, popping and peeping



- If the bottom of the stack is at location 0, then an empty stack is represented by top = -1
- To add ("push") a new element:
  - add 1 to top and store the new element in stk[top]
- To remove ("pop") an element:
  - Get the element from stk[top] and decrease top by 1
- To peep at the top of the stack, just look at stk[top] (but what happens if top < 0 ?)

After popping



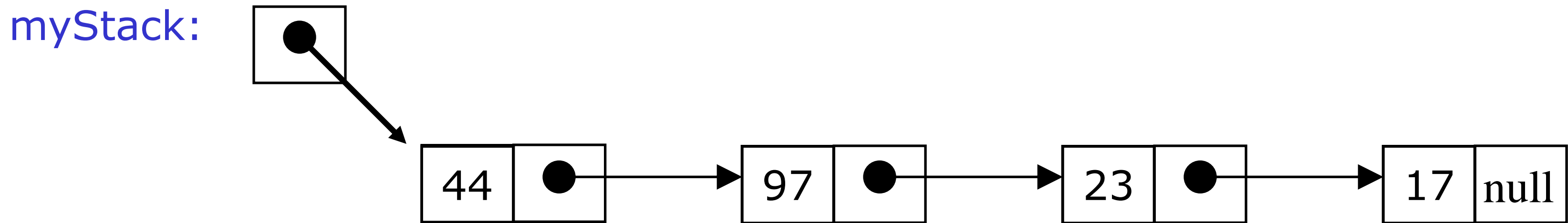
- When you pop an element, do you need to do anything to the “deleted” element sitting in the array?
- If this is an array of primitive values, there is no need to do anything
- However, if the array contains references to objects, you should set stk[top] = null before decreasing top by 1
  - Why? So that Groovy will garbage collect the “deleted” object’s storage in memory, for reuse

Error checking

- There are two stack errors that can occur:
  - Underflow: trying to pop (or peek at) an empty stack
  - Overflow: trying to push onto an already full stack
- For underflow, you should throw an exception
  - if you don't catch it yourself, Groovy will throw an ArrayIndexOutOfBounds exception
  - you could create your own, more informative exception
- For overflow, you could do the same as for underflow,
  - or you could check for your array being full as part of your push operation, and if it is copy the contents of the full stack into a new, larger array before pushing the new element onto it

(ii) Linked-list implementation of stacks

- Since all the action happens at the top of a stack, a linked list is another way to implement at stack
- The “header” variable of the list points to the top of the stack:



- pushing is done by inserting an element at the front of the list (see earlier)
- peeping is done by looking at myStack.value
- popping is done by myStack = myStack.next

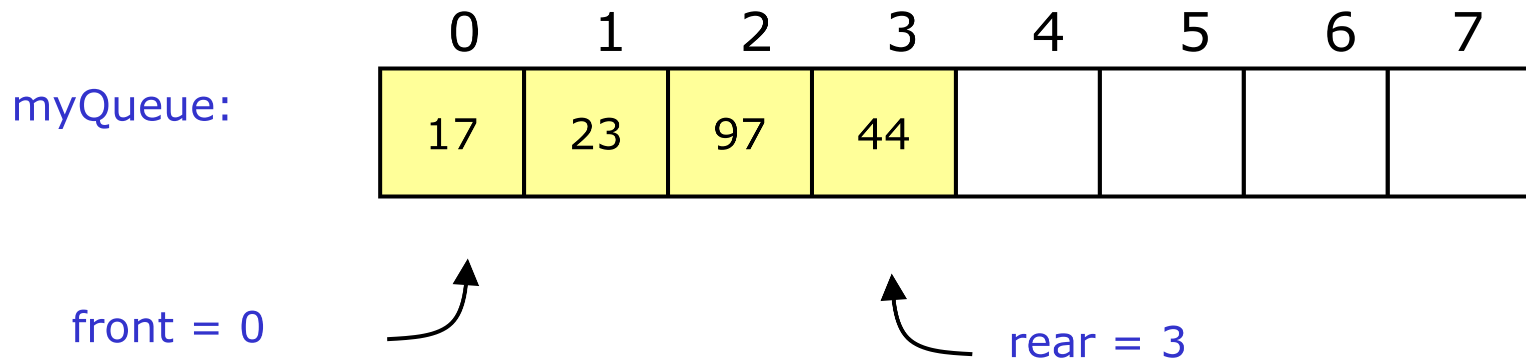
Linked-list implementation of stacks

- With a linked-list implementation, Overflow will not happen (unless the computer's memory is exhausted, which is another kind of problem!)
- Underflow can happen (trying to pop from a null stack) and should be handled the same way as for an array implementation
- When the first element is popped from the stack, if it references an object, then this does not need to be set to null
- Unlike an array implementation, you can no longer get to the first element from the rest of the linked list
- So its storage can be garbage collected as appropriate

4. Queues

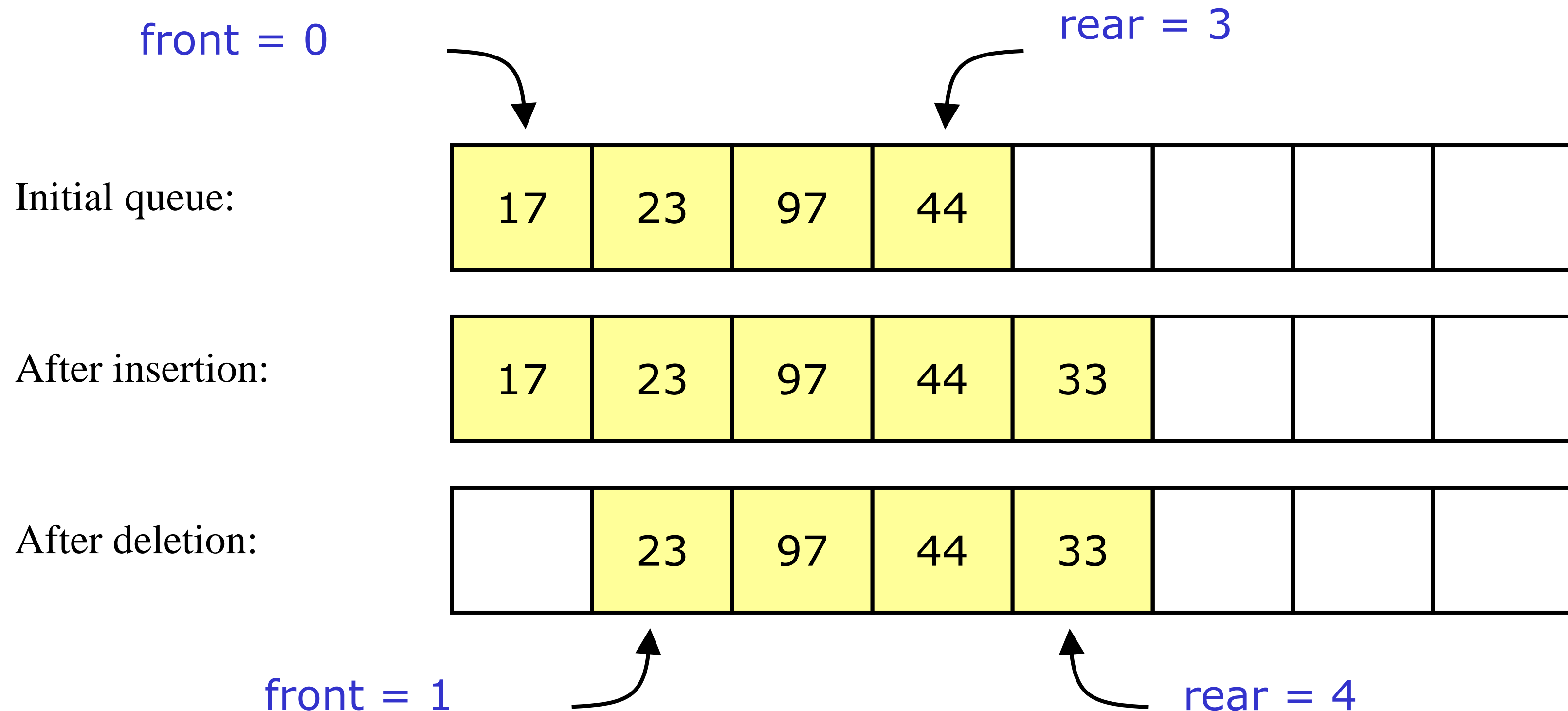
- A queue is a first in, first out (FIFO) data structure:
  - items are removed from a queue in the same order as they were inserted
- This kind of data structure is useful for “buffering” data as it’s produced so it doesn’t have to be processed immediately but can be used later on

(i) Array implementation of queues



- **To insert:** put new element in location 4, and set rear to 4
- **To delete:** take element from location 0, and set front to 1

Array implementation of queues



- Notice how the array contents "crawl" to the right as elements are inserted and deleted
- This will be a problem after a while!

Queue implementation details

- With an array implementation:
  - you can have both overflow and underflow
  - you should set deleted elements to null

Linked-list implementation of queues

- In an linked list you can easily find the successor of an element, but not its predecessor
  - remember, references are "one-way"
- If you know where the *last* element in a list is, it's easy to add a new element after it
- So:
  - use the *first* element in a linked list as the *front* of the queue, from which dequeueing will occur
  - use the *last* element in a linked as the *rear* of the queue, to which enqueueing will occur
  - keep references to *both* the front and the rear of the linked list

Enqueueing an element

last

first

44 97 23 17

Element to be enqueued

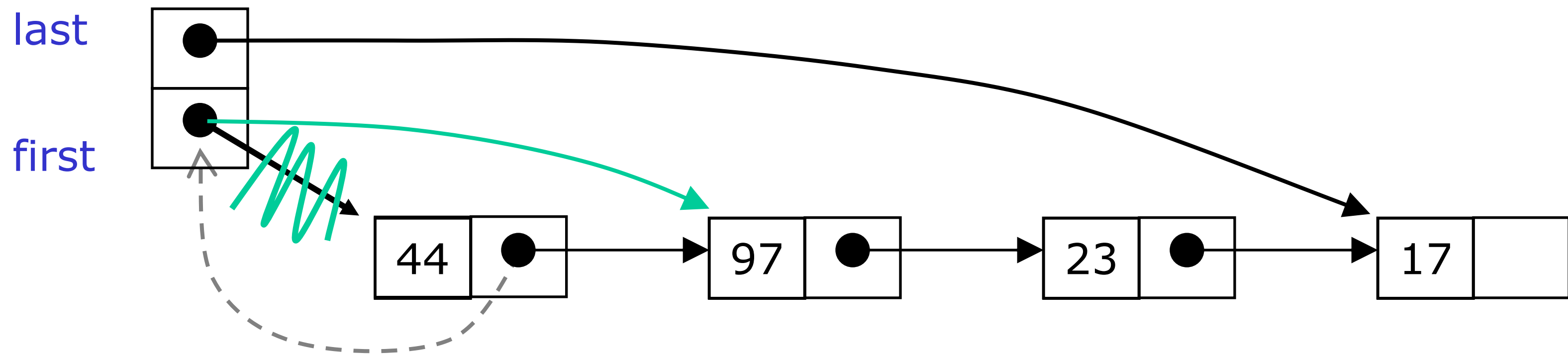
To enqueue an element:

Find the current last element

Change its reference to point to the new last element

Update the last reference to point to the new last element

Dequeueing an element



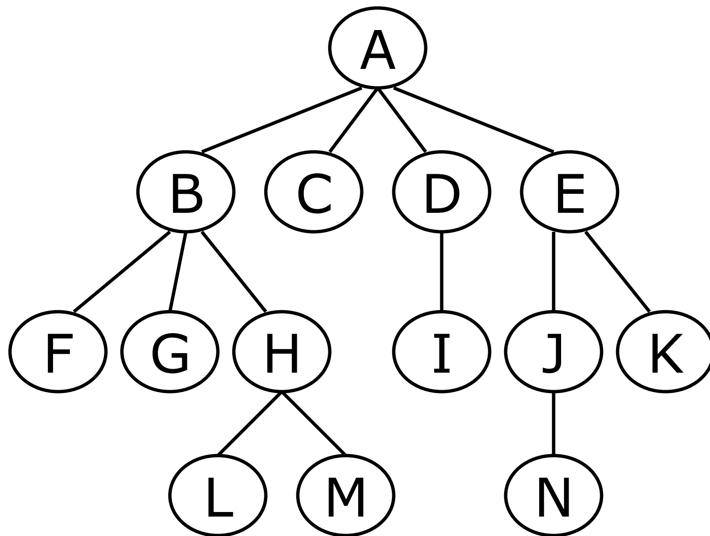
- To dequeue an element:
  - Copy the reference from the first element into first reference

Queue implementation details

- With a linked-list implementation:
  - you can have underflow (trying to dequeue from a null queue)
  - overflow is a global out-of-memory condition
  - there is no need to set deleted elements to null

5. Trees

- A tree data structure consists of a set of ***nodes*** with references between them
- Each node contains a value and references to other nodes – called its ***children*** – which may or may not be ordered, depending on the program's requirements
- There is a single ***root*** node (drawn at the top by convention!)



- Each node (except the root) has a single ***parent***
- Each node has a ***depth*** i.e. distance from the root (the root has depth 0)
- A node that has no children is called a ***leaf*** node

More definitions

- The ***descendents*** of a node are its children, its children's children, etc.
- The ***ancestors*** of a node are its parent, its parent's parents, etc.
- Two nodes are ***siblings*** if they have the same parent
- The ***subtree rooted at a node*** consists of the given node and all of its descendents
- An ***ordered tree*** is one in which the order of the children is important; an ***unordered tree*** is one in which the children of a node can be thought of as a set

Operations on trees

- It must be possible to:
  - Construct a new tree
  - Add a child to a node
  - Get the children of a node
  - Access (get and set) the value in a node
- It may also be necessary to:
  - Remove a child (and the subtree rooted at that child)
  - Get the parent of a node (this would need “upwards” references as well, from children back to their parents)

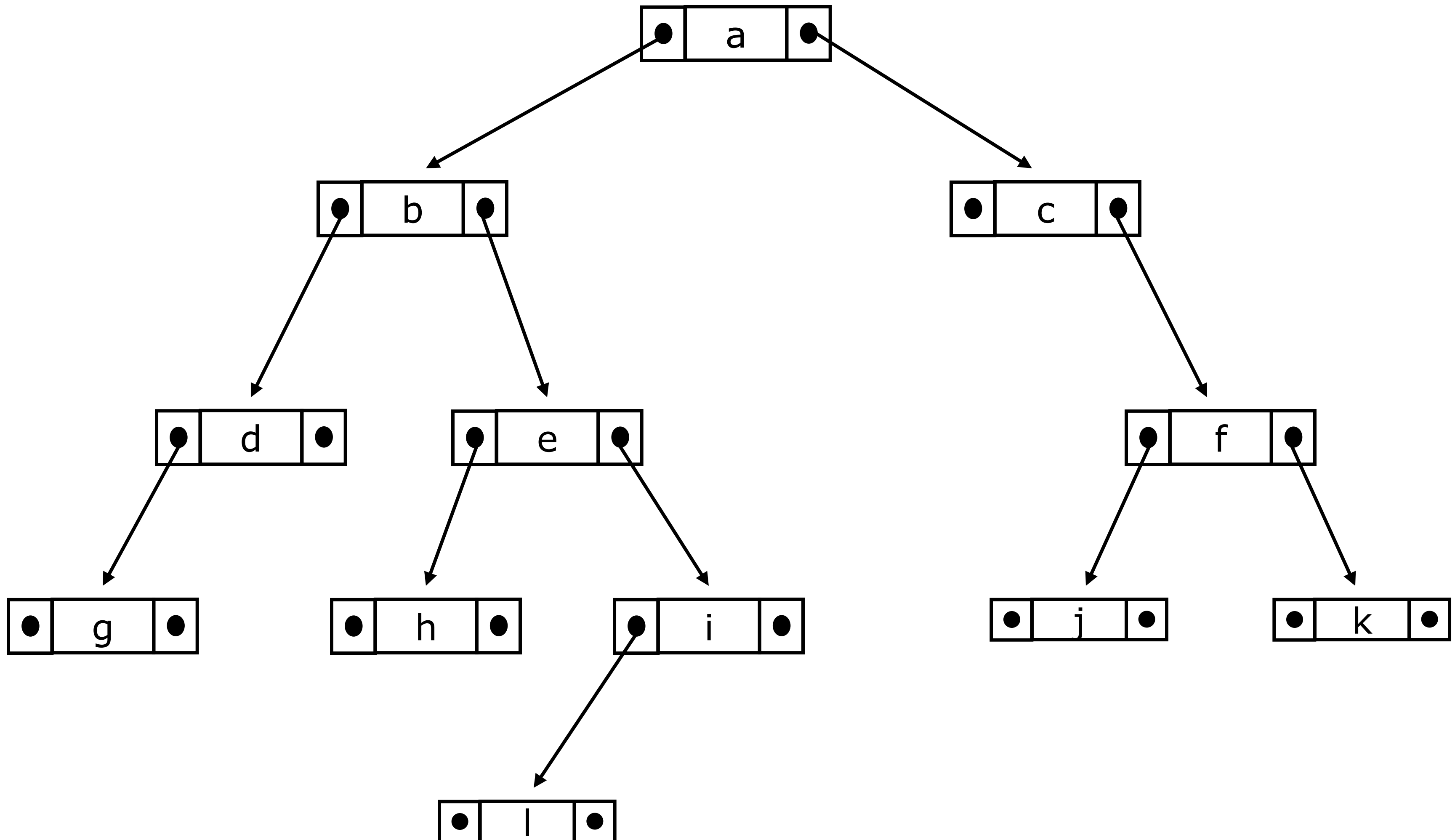
Binary Trees

- A binary tree is composed of zero or more nodes (zero nodes if the reference to it is null)
- Each node contains:
  - a value (some sort of data item)
  - a reference to a left child (which may be null)
  - a reference to a right child (which may be null)

Binary trees in Groovy

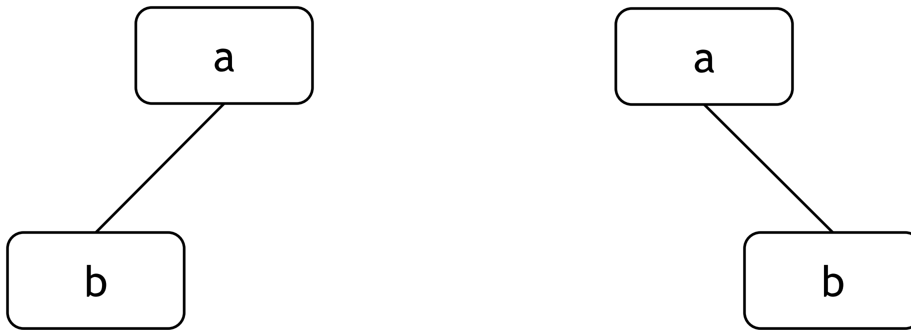
```
class BinaryTree  {
        int value;
        BinaryTree leftChild;
        BinaryTree rightChild;

        BinaryTree (int v, BinaryTree l, BinaryTree r) {
                value = v;
                leftChild = l;
                rightChild = r;
        }
}
```

Detailed picture of a binary tree



Left ≠ Right

- The following two binary trees are *different:*

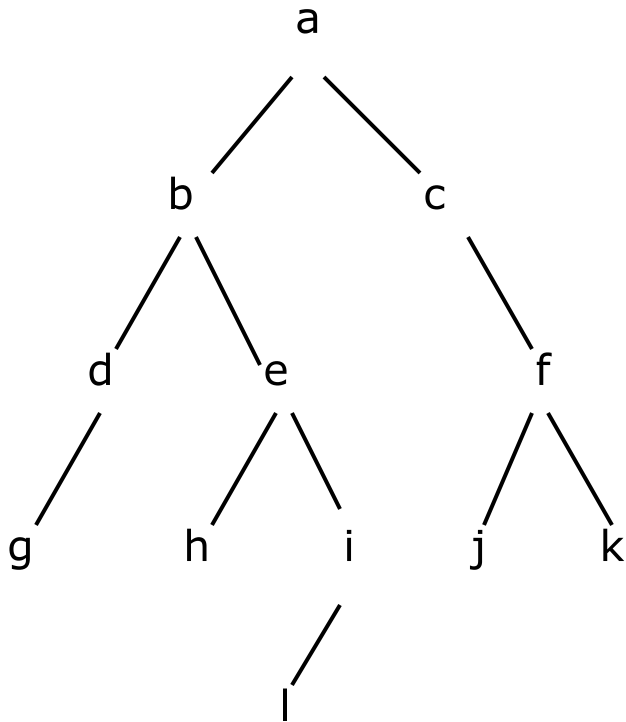


- In the first binary tree, the root node has a left child but no right child; in the second, the root node has a right child but no left child

Usages of binary trees

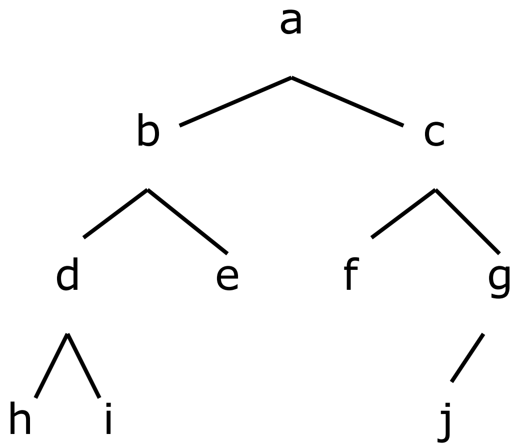
- The binary tree data structure is useful for
  - representing programming language expressions – “syntax trees”
  - organising data so that it can be search quickly – “binary search trees”

Size and depth

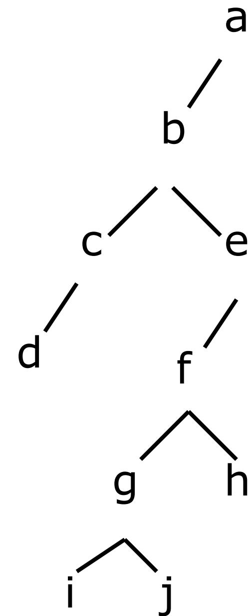


- The size of a binary tree is the number of nodes in it
  - This tree has size 12
- The depth of a node is its distance from the root
  - a is at depth zero
  - e is at depth 2
- The depth of a binary tree overall is the depth of its deepest node
  - This tree has depth 4

Balance



A balanced binary tree

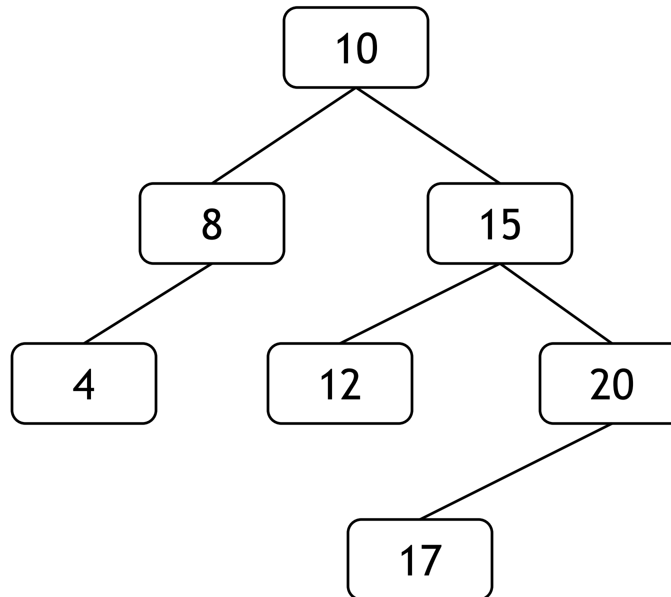


An unbalanced binary tree

- A binary tree is ***balanced*** if every level above the lowest is “full” i.e. there are no null references in nodes at that level
- In some settings, having a balanced binary tree is desirable (why?)

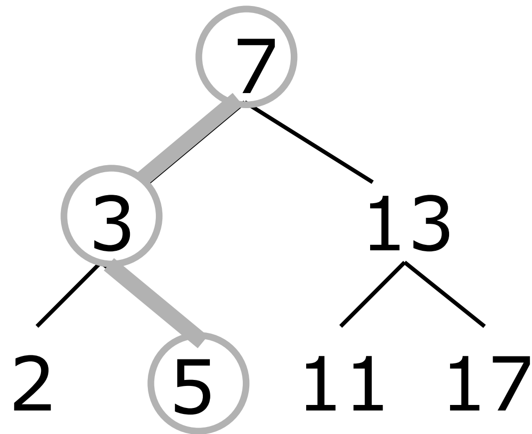
Sorted binary trees

- A binary tree is sorted if every node in the tree is larger than (or equal to) its left descendants, and smaller than (or equal to) its right descendants
- Equal nodes can go either on the left or the right (but this has to be consistent for the whole tree)



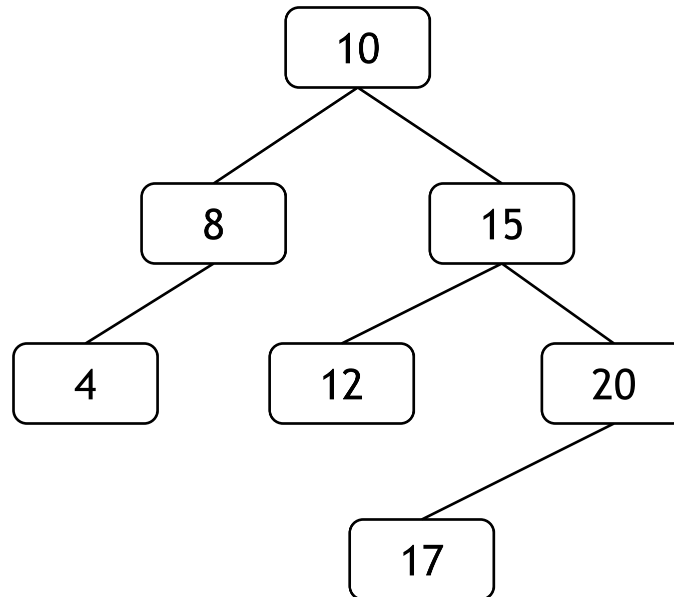
Search in a sorted binary tree

- How do we check if a number is present in a sorted binary tree? e.g. "Is 5 present in the following tree ?"



Search in a sorted binary tree

- Is 17 present in this tree ?
- Is 12 present in this tree ?
- Is 22 present in this tree ?



Traversing a binary tree

- To "traverse" the binary tree means to visit each node in the binary tree once
- A binary tree consists of a root, a left subtree and a right subtree
- So tree traversals are naturally recursive
- Since a binary tree has three "parts," there are six possible ways to traverse the binary tree:
  - root, left, right
  - left, root, right
  - left, right, root
  - root, right, left
  - right, root, left
  - right, left, root

Preorder traversal

- With preorder traversal, the root is visited *first, followed by the left and right subtrees*
- Here is a preorder traversal to print out all the elements in a binary tree:

```
def preorderPrint(BinaryTree bt) {
    if (bt == null) return
    println(bt.value)
    preorderPrint(bt.leftChild)
    preorderPrint(bt.rightChild)
}
```

Inorder traversal

- With inorder traversal, the root is visited *after the left subtree and before the right subtree*
- Here is an inorder traversal to print out all the elements in a binary tree:

```
def inorderPrint(BinaryTree bt) {
    if (bt == null) return
    inorderPrint(bt.leftChild)
    println(bt.value)
    inorderPrint(bt.rightChild)
}
```

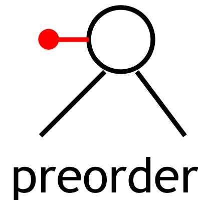
Postorder traversal

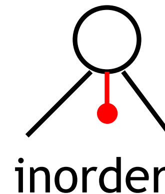
- With postorder traversal, the root is visited *last, after the left subtree and the right subtree*
- Here is a postorder traversal to print out all the elements in a binary tree:

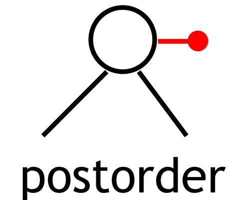
```
def postorderPrint(BinaryTree bt) {
    if (bt == null) return
    postorderPrint(bt.leftChild)
    postorderPrint(bt.rightChild)
    println(bt.value)
}
```

Visualising tree traversals using "flags"

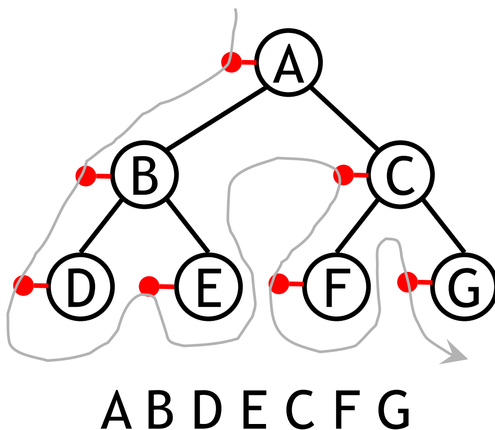
- The order in which the nodes are visited during a tree traversal can be visualised by imagining there is a "flag" attached to each node:

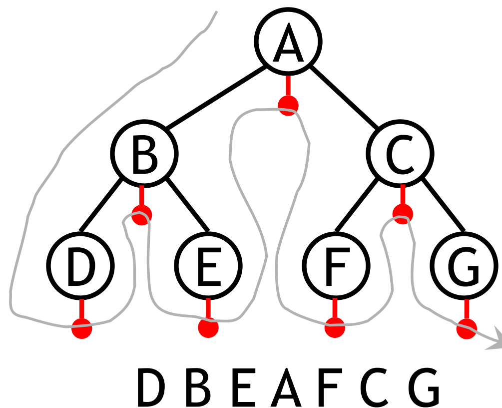


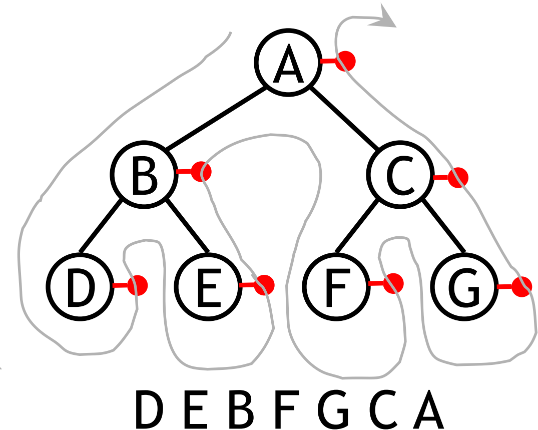




- To traverse the tree, collect the flags:







Copying a binary tree

- Here is a postorder traversal to make a complete copy of a given binary tree:

```
def copyTree(BinaryTree bt) {
    if (bt == null) return null
    def left = copyTree(bt.leftChild)
    def right = copyTree(bt.rightChild)
    return new BinaryTree(bt.value, left, right)
}
```

Summary of this lecture

- We have looked at how to use <u>references to objects</u> in Groovy to create data structures such as
  - Linked lists, Stacks, Queues, Trees

- Data structures such as stacks, queues and trees have numerous usages in computing

- We have also looked at how to implement operations such as the following in Groovy:
  - Linked lists: creation and traversal
  - Stacks: pop, push, peep
  - Queues: enqueue, dequeue
  - Binary trees: traversal (inorder, preorder, postorder), copying
  - Sorted binary trees: search