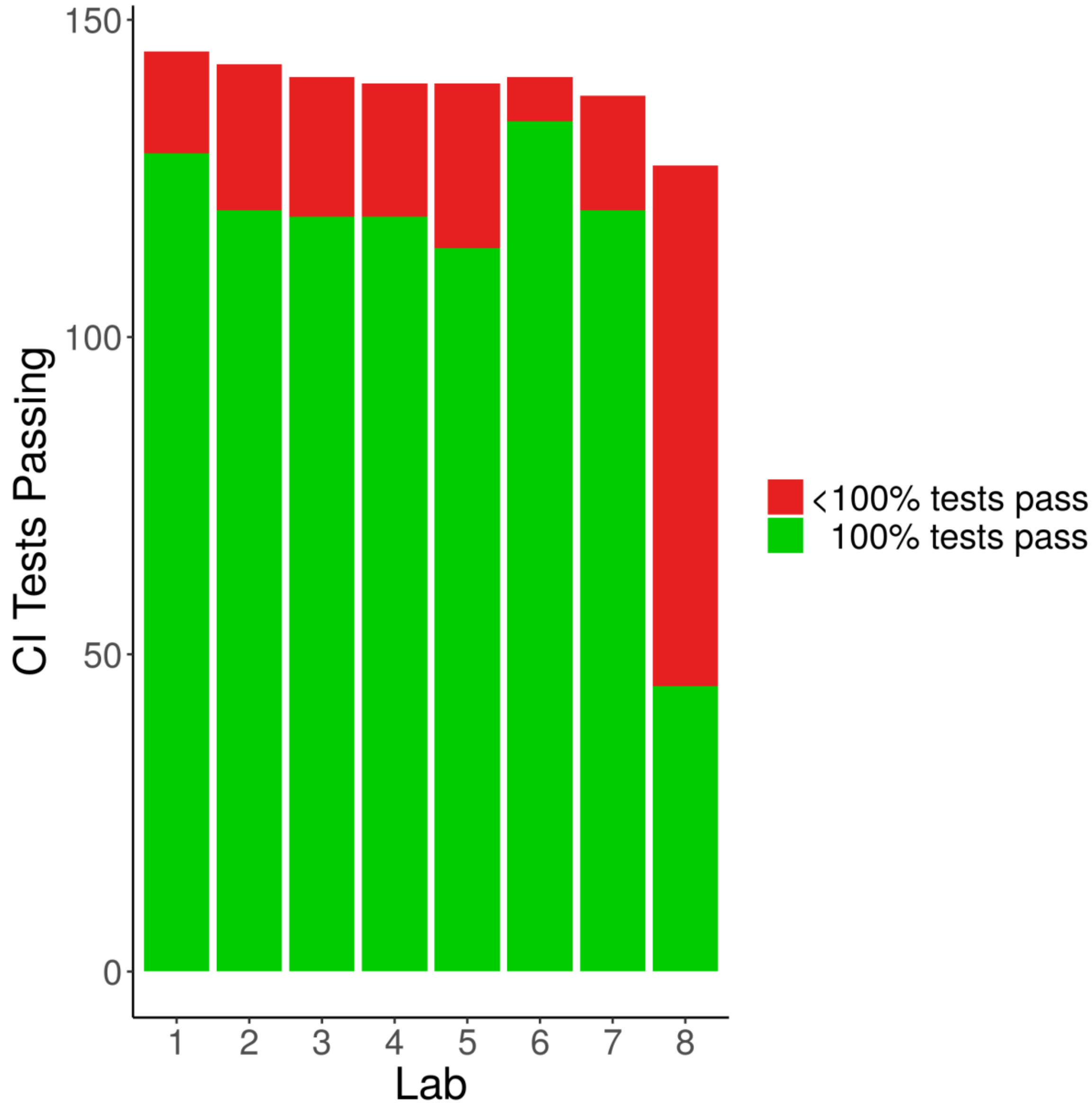
Lab 8 deadline

- Group 1
  - Friday 22nd
- Group 2
  - Monday 25th



Software Development 3 {F27SG}

Revision

Rob Stewart

Overview

Topics for the **exam** are covered in lectures 3 to 17 (and corresponding labs):

- *Concepts & implementation*
- Complexity (big-O)
- Recursion
- Stacks & queues
- Linked & doubly-linked lists
- Trees (binary, BST, k-ary, Tries)
- Priority queues and heaps
- Search (linear and binary)
- Sorting
  - insertionSort and bubbleSort
  - mergeSort and quicksort
- ***Programming exercises are particularly relevant***
  - ***Review your labs 1-8 code as part of revision***

You can **ignore**:

- Unit testing (lecture 1)
- I/O (lecture 2)
- Advanced Java (lecture 18)
  - Generics
  - Anonymous class
  - Lambda expressions
- Security (lecture 19)

Note that past exams may have questions about topics we have not covered (or you can ignore). E.g

- AVL trees
- Lambda expressions

You can ignore these questions.

The exam

- Electronic exam is worth 50%
- You answer **all** questions
- Exam question types
  1. Multiple choice
     - Exactly one answer
     - Multiple correct answers (select all that are correct)
  2. Fill in the box with
     - A number
     - One line of Java code
  3. Drag and drop text into correct box
- Bring a pen for working

Revision materials

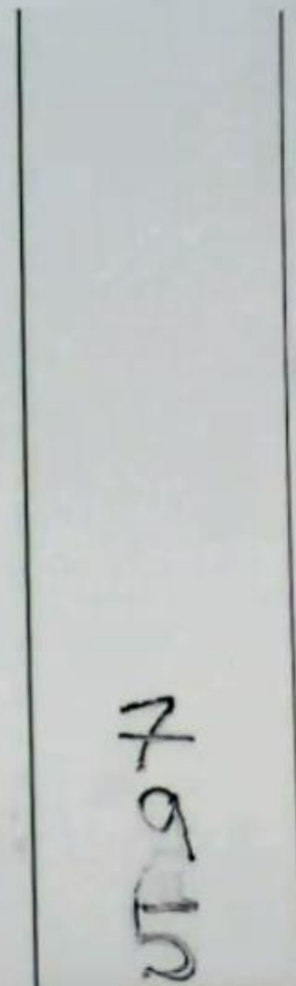
- Lecture slides
  - Learning Materials -> Edinburgh -> lecture slides
- Lecture capture
  - Leearning Materials -> Edinbugh -> lecture capture
- Model code solutions
  - https://gitlab-student.macs.hw.ac.uk/rs46

The Stack ADT

- The data is the type of elements stored
- Operations
  - **push**(object), **pop**(), **top**() , **size**() , **isEmpty**()
- Error conditions: pop/top of empty stack
- Implementation
  - As arrays
  - As Linked list

Lecture capture - *Lecture 4: stacks*

Operations	return value
push(5)	void
push(3)	void
pop()	3
push(2)	void
push(8)	void
pop()	8
pop()	2
push(9)	void
push(1)	void
pop()	1
push(7)	void
push(6)	
pop()	
pop()	
push(4)	
pop()	



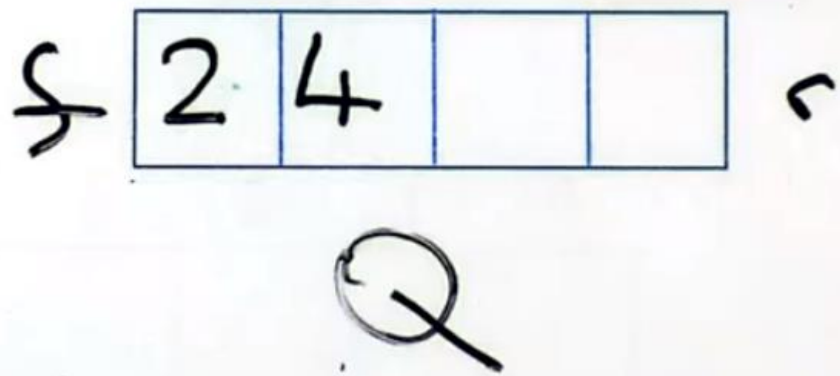
Stack

The Queue ADT

- The data is the type of elements stored
- Operations
  - **enqueue**(object), **dequeue**(), **front**() , **size**() , **isEmpty**()
- Error conditions: pop/top of empty stack
- Implementation
  - As arrays
    - Remember “wrap around”
  - As Linked list
    - Add to tail, remove from head; keep reference to tail

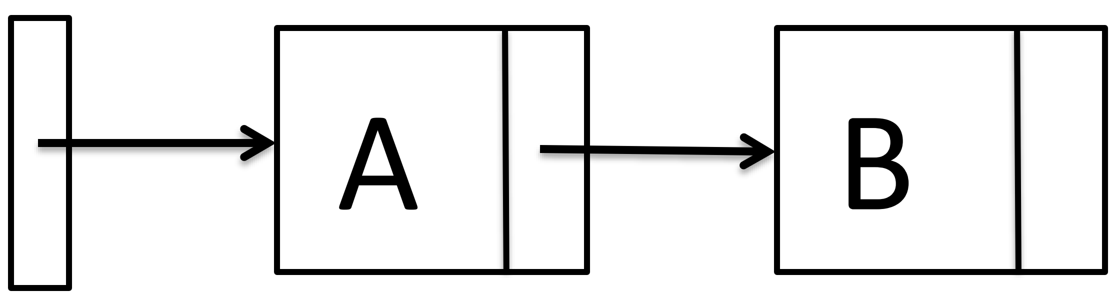
Lecture capture - *Lecture 9: queues*

operation	returned
enqueue(2)	void
enqueue(4)	void
front()	2
isEmpty()	false
dequeue()	2
dequeue()	
dequeue()	

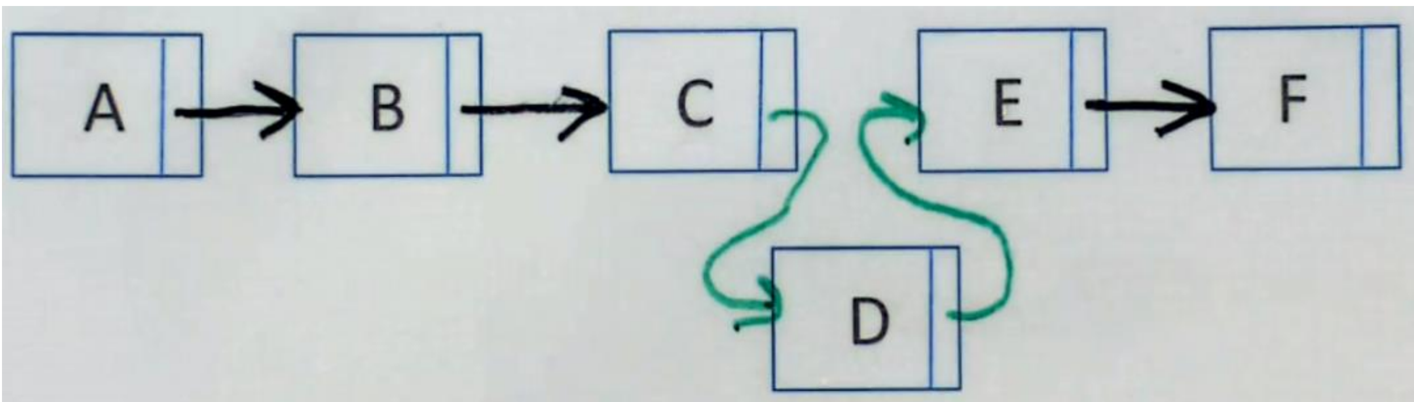


Linked Lists

- A Linked List is a **linearly** ordered sequence of Nodes
- We can step along the sequence to access the value in each node
- Operations:
  - Search, insert and delete (at head, tail, or middle)

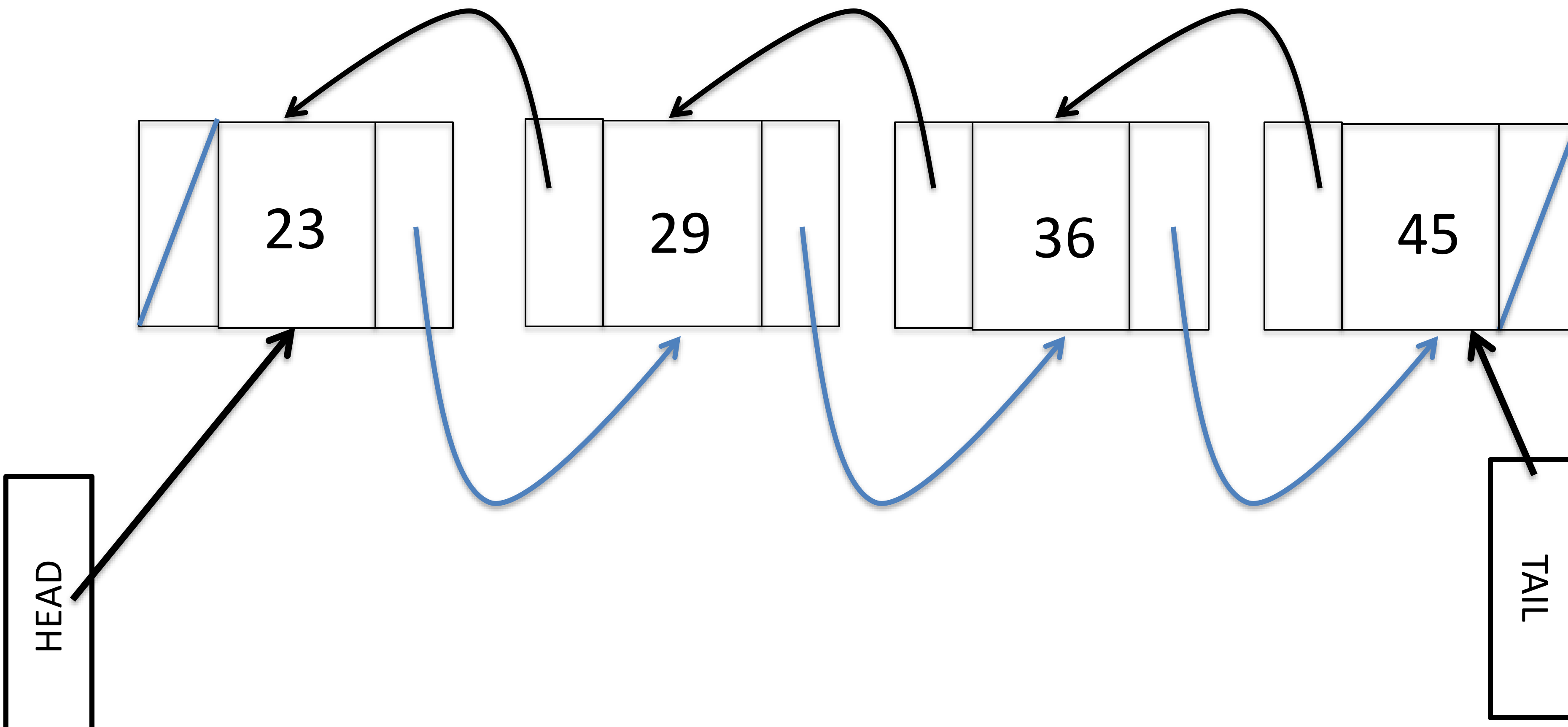


Lecture capture - *Lecture 6: linked lists*

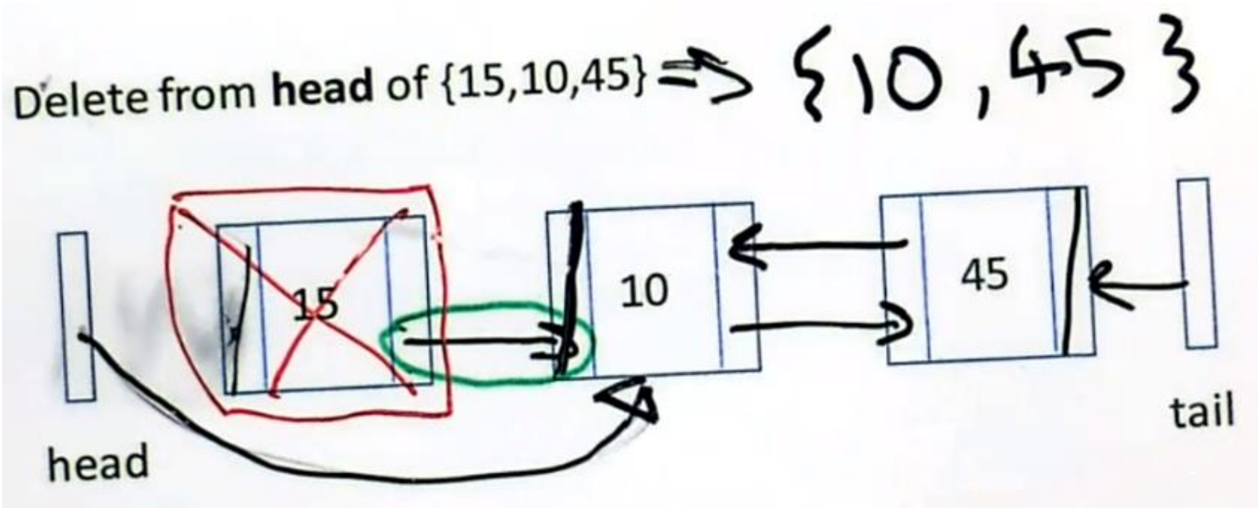


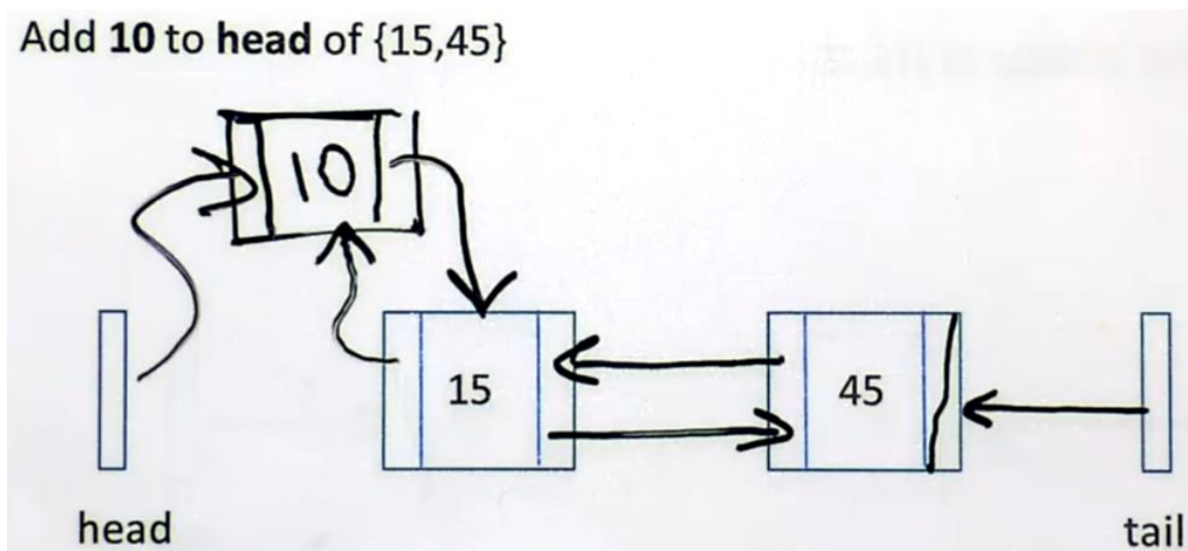
Doubly Linked Lists

- Can Traverse the List in Both Directions
  - Still not Random access, but can get Previous Node
  - Makes removal at end and Sorting lists easier



Lecture capture - *Lecture 10: doubly linked lists*

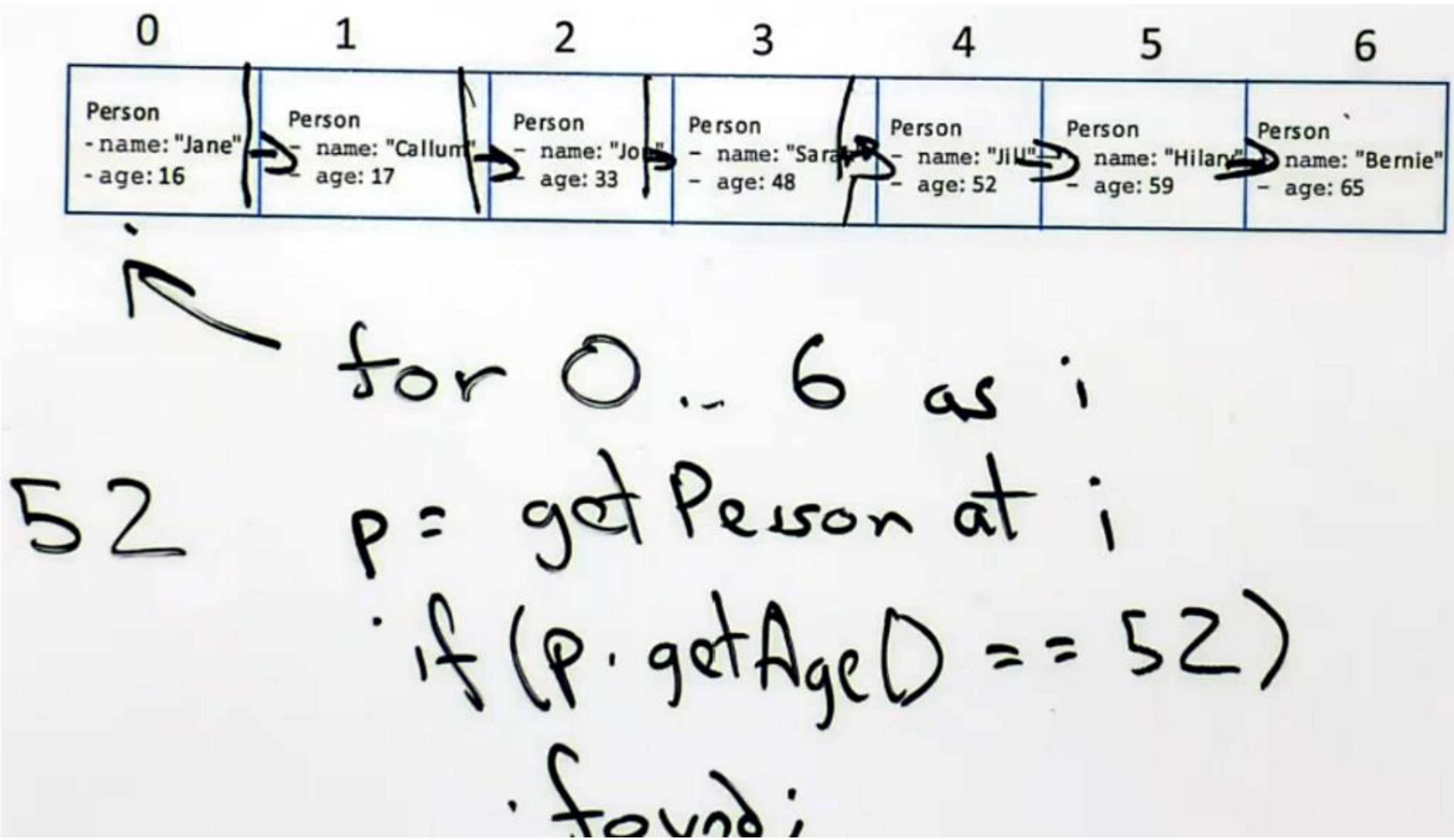




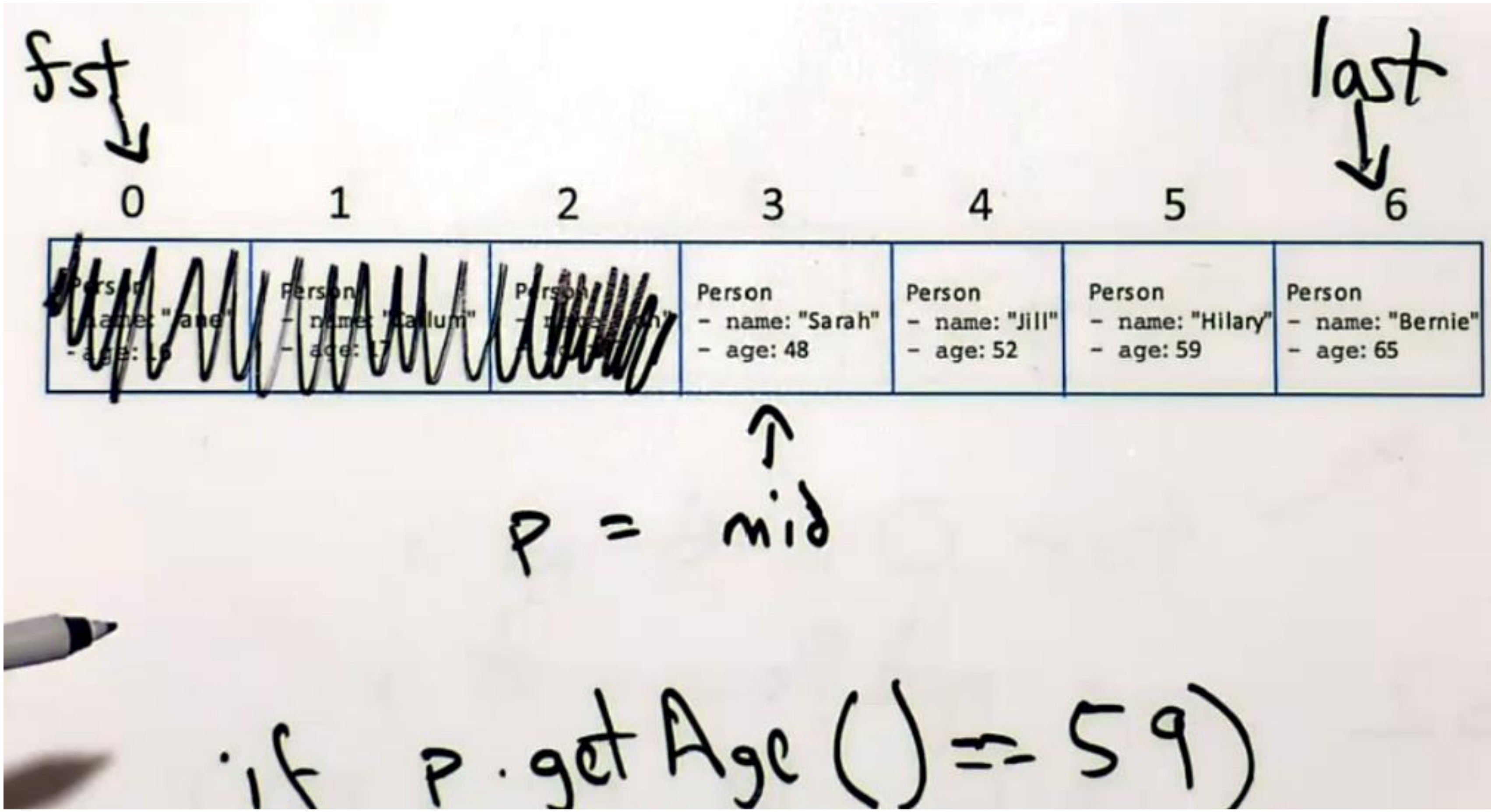
Search

- In **linear search** we
  - start at the beginning of the list/array
    - and compare until we find a match, which we return
- **Binary search** starts in the middle
  - if we get a match the value is returned
  - if the key searched for is greater than the middle we search in the top half
  - if the key searched for is smaller than the middle we search in the bottom half
  - if we end up with an empty list search has failed!
  - Binary search assumes a sorted list

Lecture capture - *Lecture 11: linear and binary search*



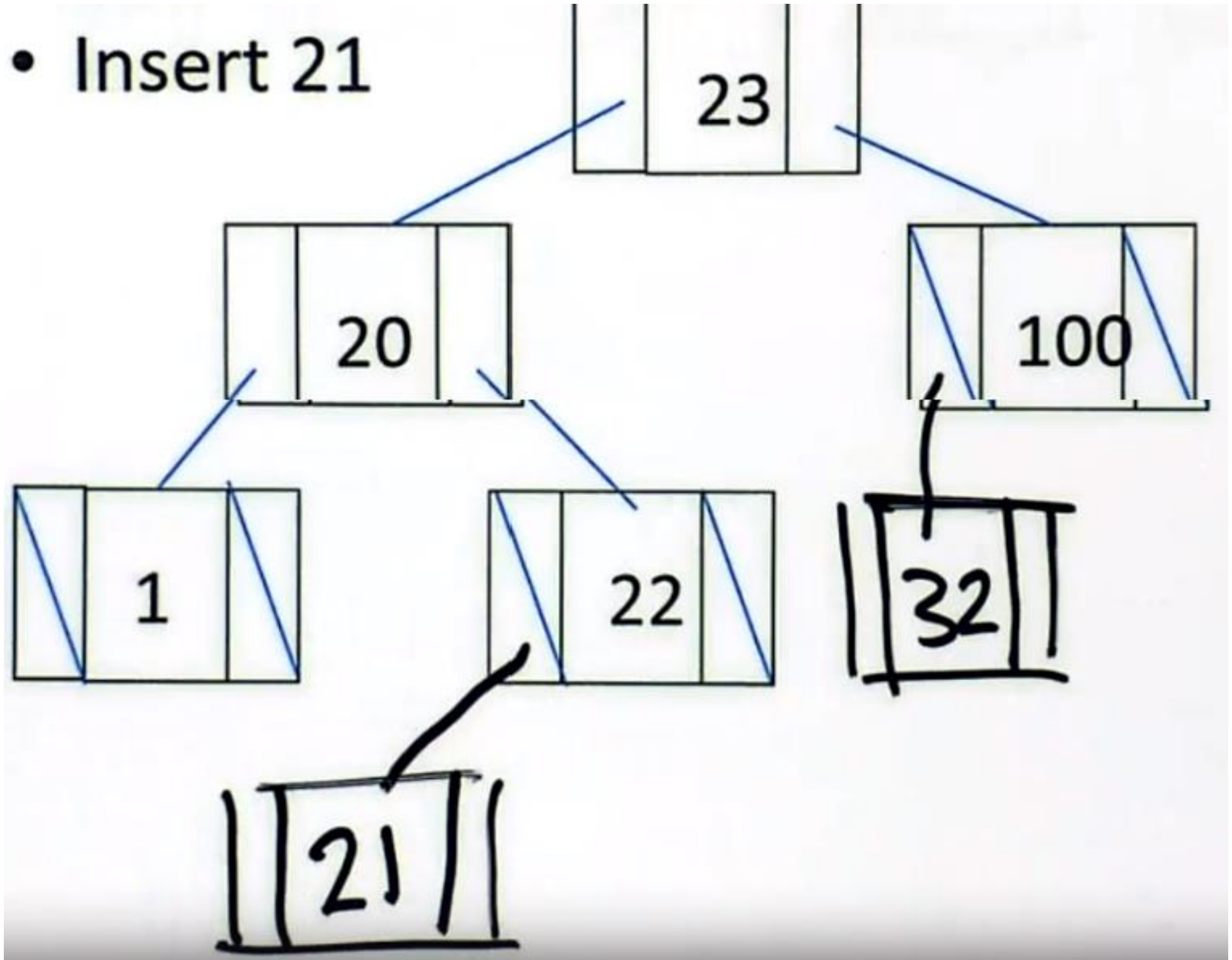
Lecture capture - *Lecture 11: linear and binary search*



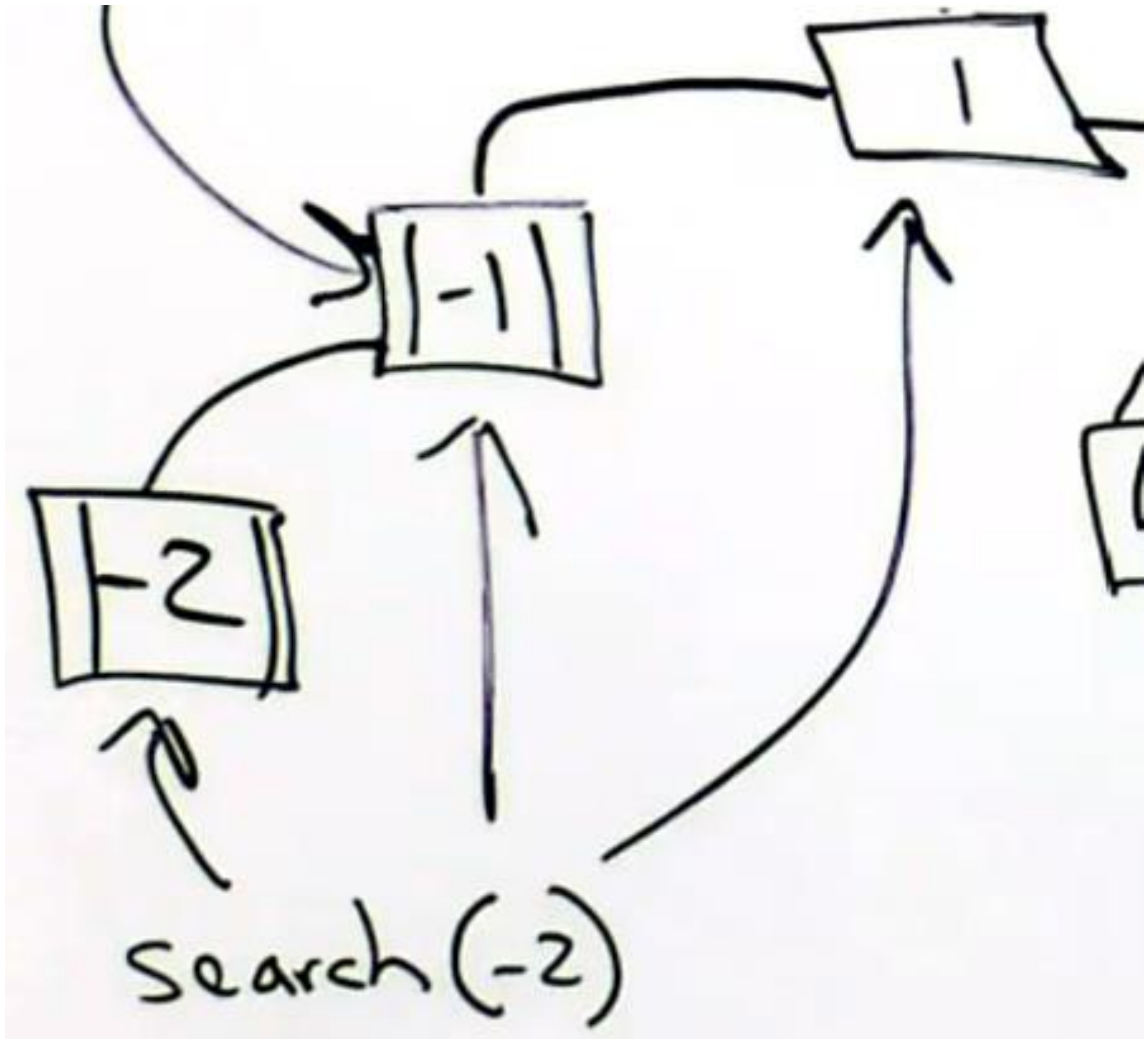
Binary Search Trees

- Binary tree – max 2 child trees
- Rules
  - For the value (index) in the current node
    - The left node value (index) is less than it
    - The right node value (index) is greater than it
  - There is no = (i.e. no duplicate values allowed)
- Traversal
  - Pre-order, post-order, in-order
- Operations
  - Add, remove and search

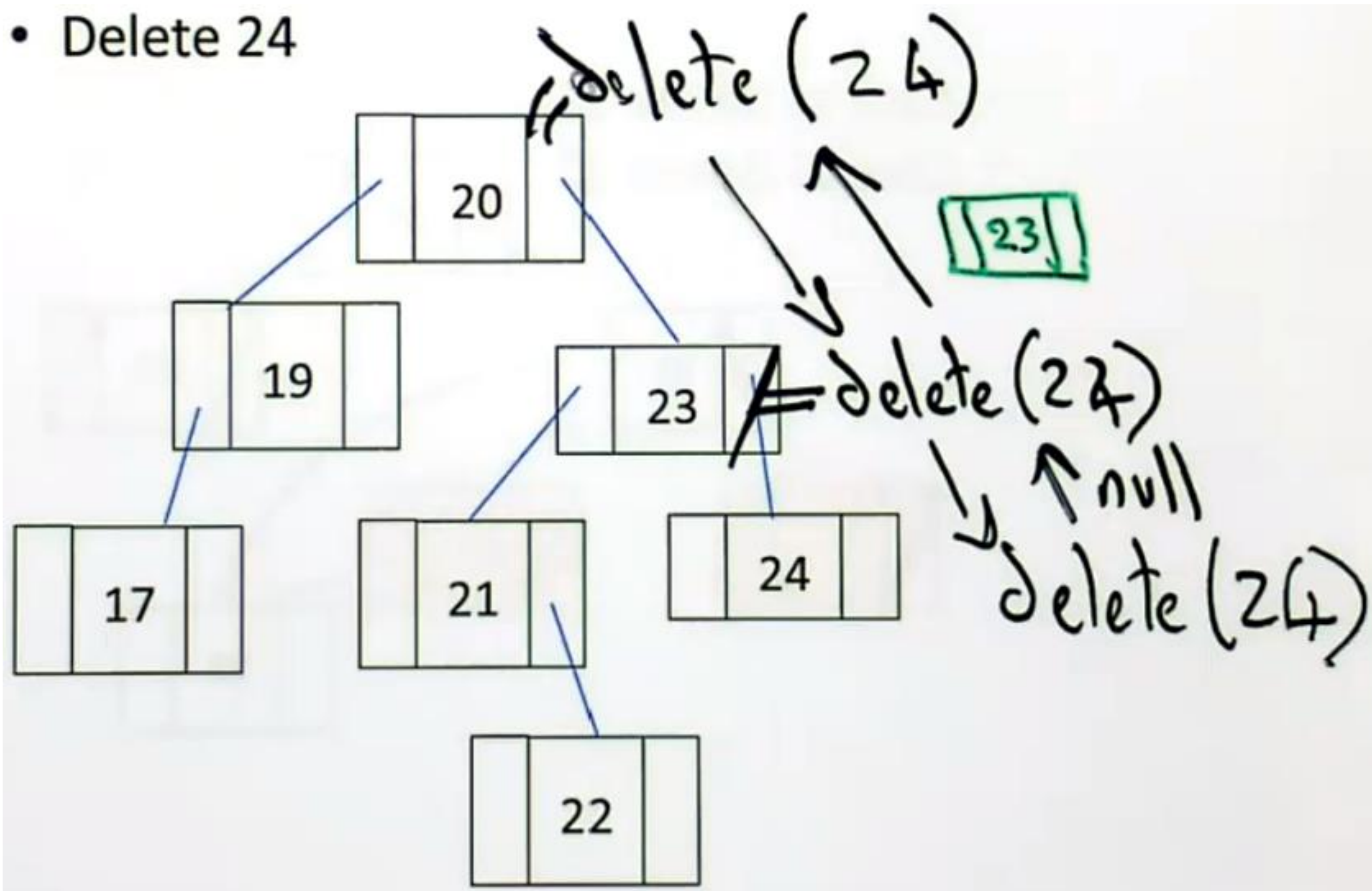
Lecture capture - *Lecture 13: implementing binary search trees*



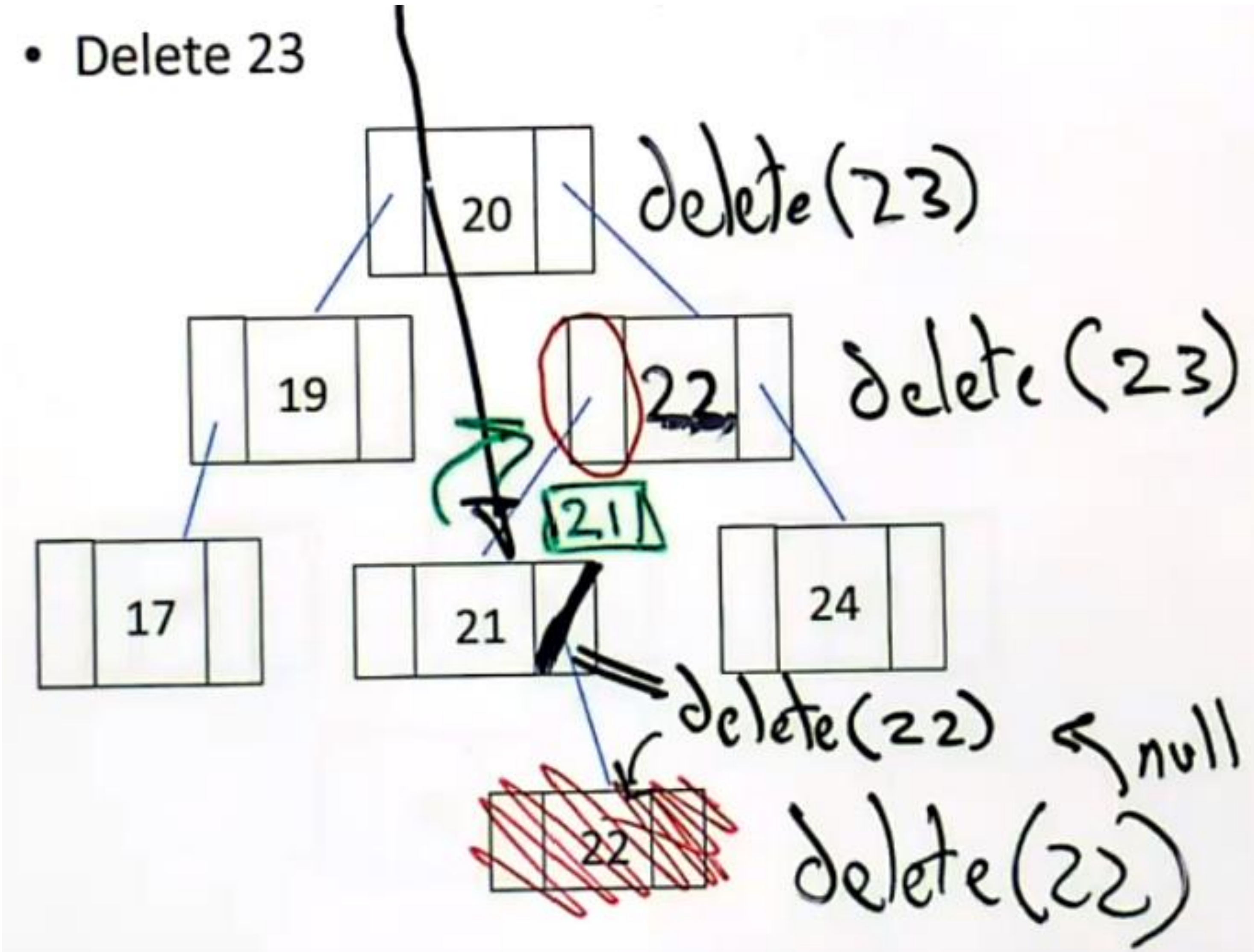
Lecture capture - *Lecture 13: implementing binary search trees*



Lecture capture - *Lecture 13: implementing binary search trees*

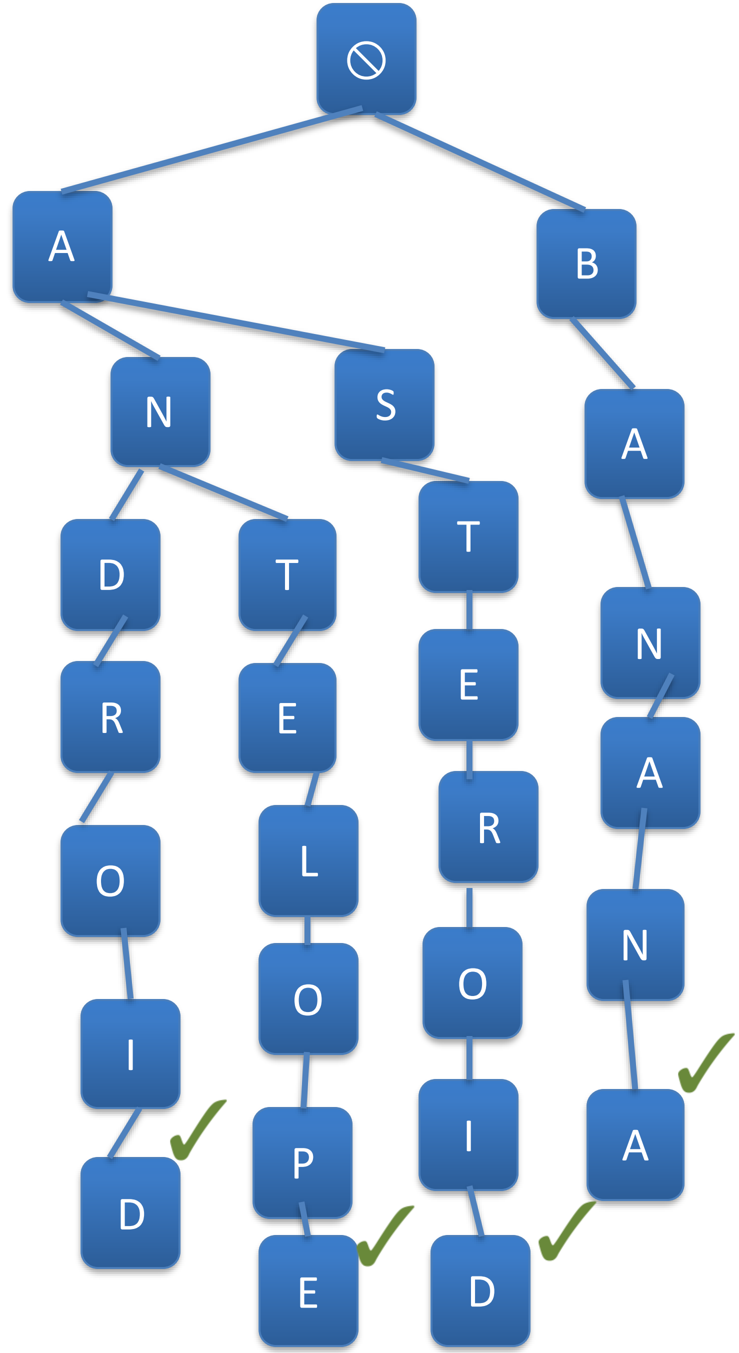


Lecture capture - *Lecture 13: implementing binary search trees*

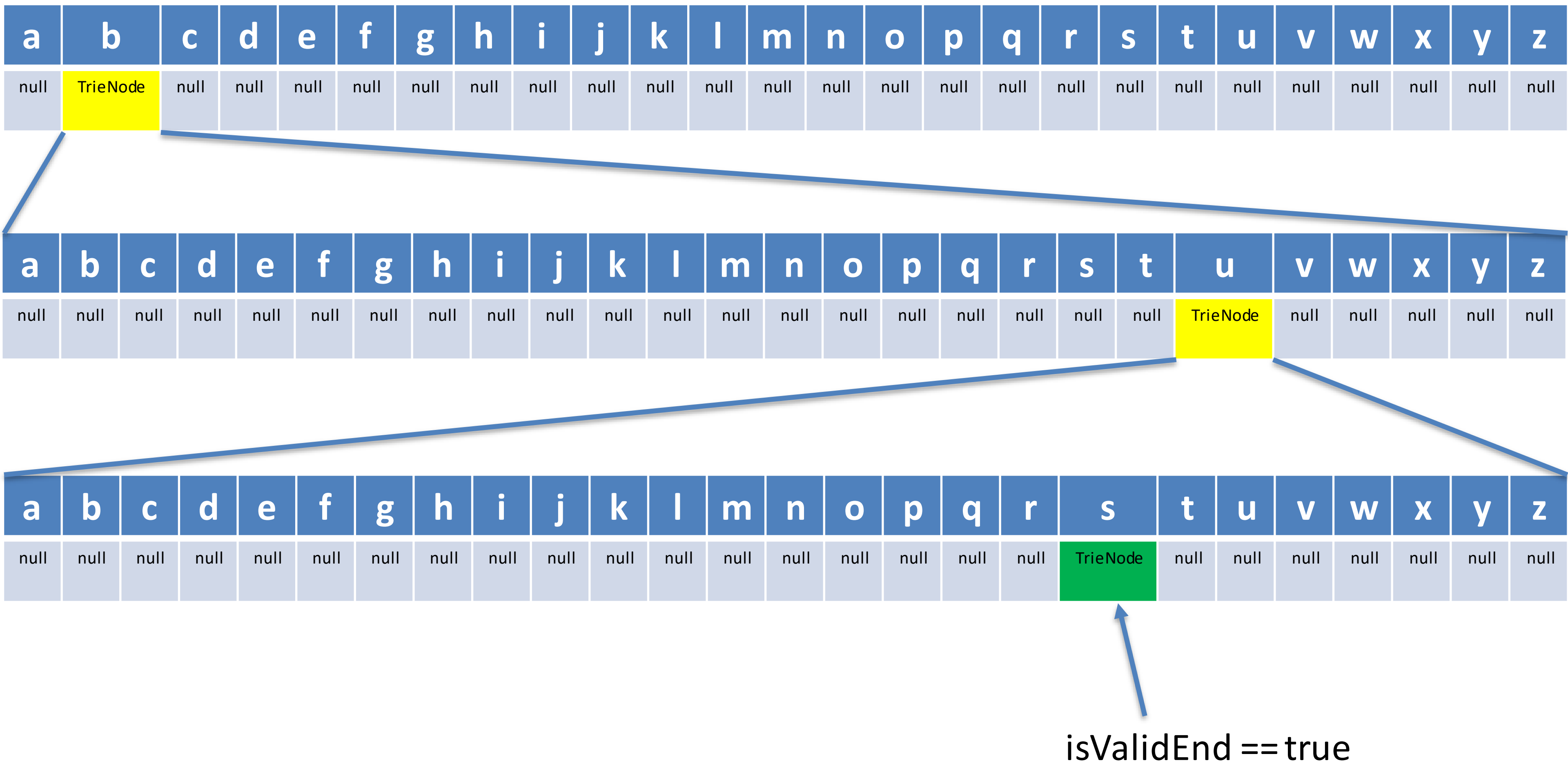


Tries

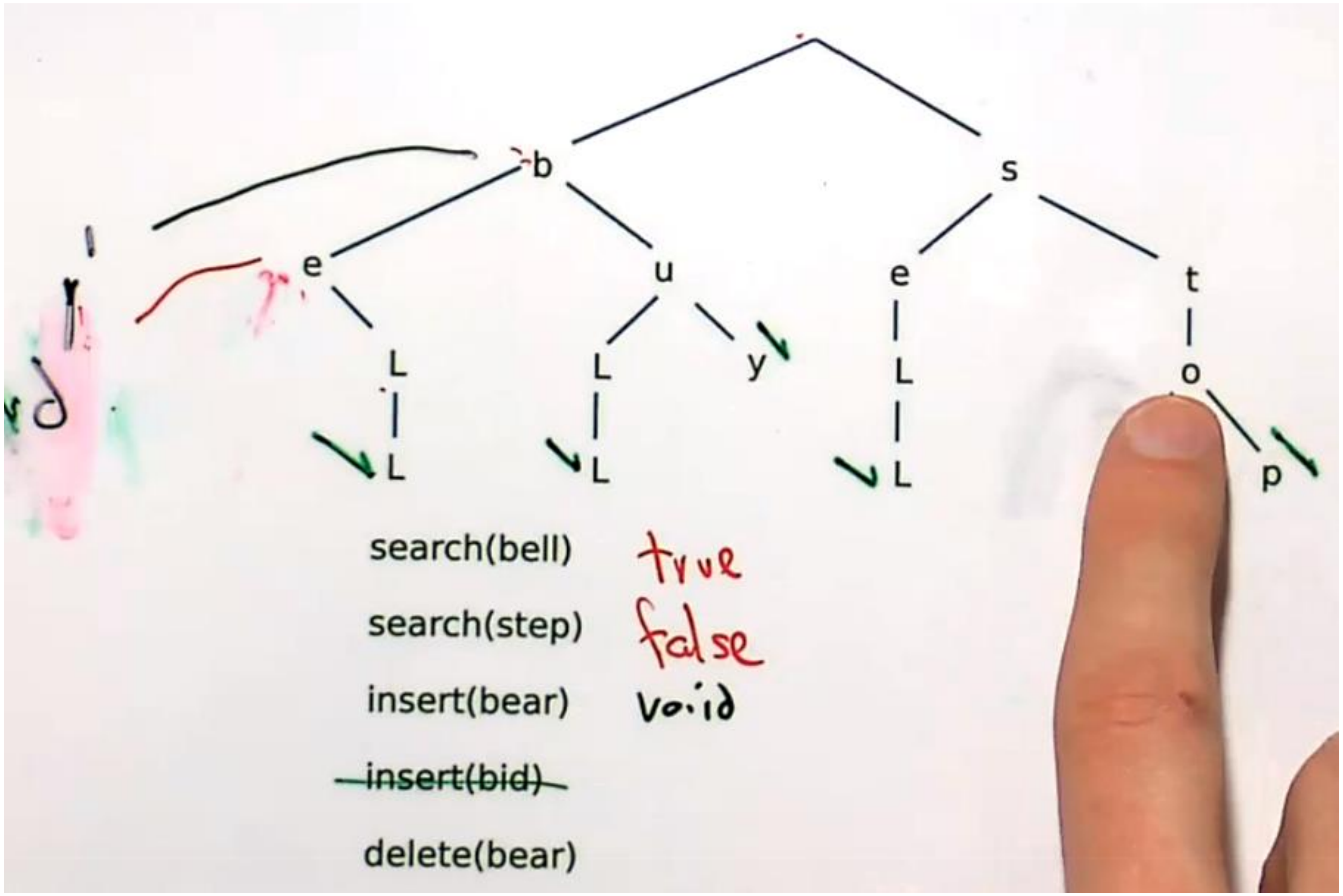
- N-ary tree (pronounced try)
- An efficient way to store a dictionary
- Each level in the tree stores a character position
  - Nth level stores the nth character of a word
  - A word is valid iff
    - Each character in it appears at the right level of the tree
    - The node containing the final character is either a leaf (or marked as a valid word)



“bus” in a trie


a b c d e f g h i j k l m n o p q r s t u v w x y z
null TrieNode null null null null null null null null null null null null null null null null null null null null null null null null
a b c d e f g h i j k l m n o p q r s t u v w x y z
null null null null null null null null null null null null null null null null null null null null TrieNode null null null null null
a b c d e f g h i j k l m n o p q r s t u v w x y z
null null null null null null null null null null null null null null null null null null TrieNode null null null null null null null
isValidEnd == true

Lecture capture - *Lecture 15: tries*

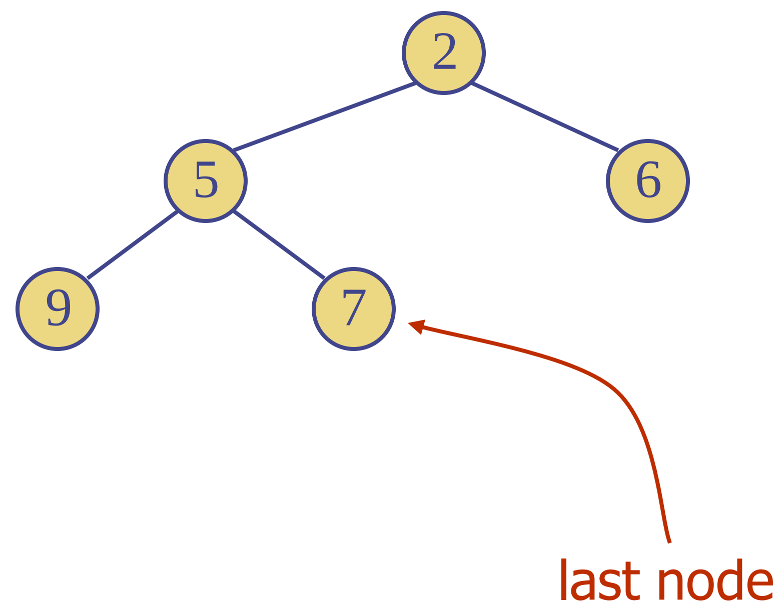


Priority Queue ADT

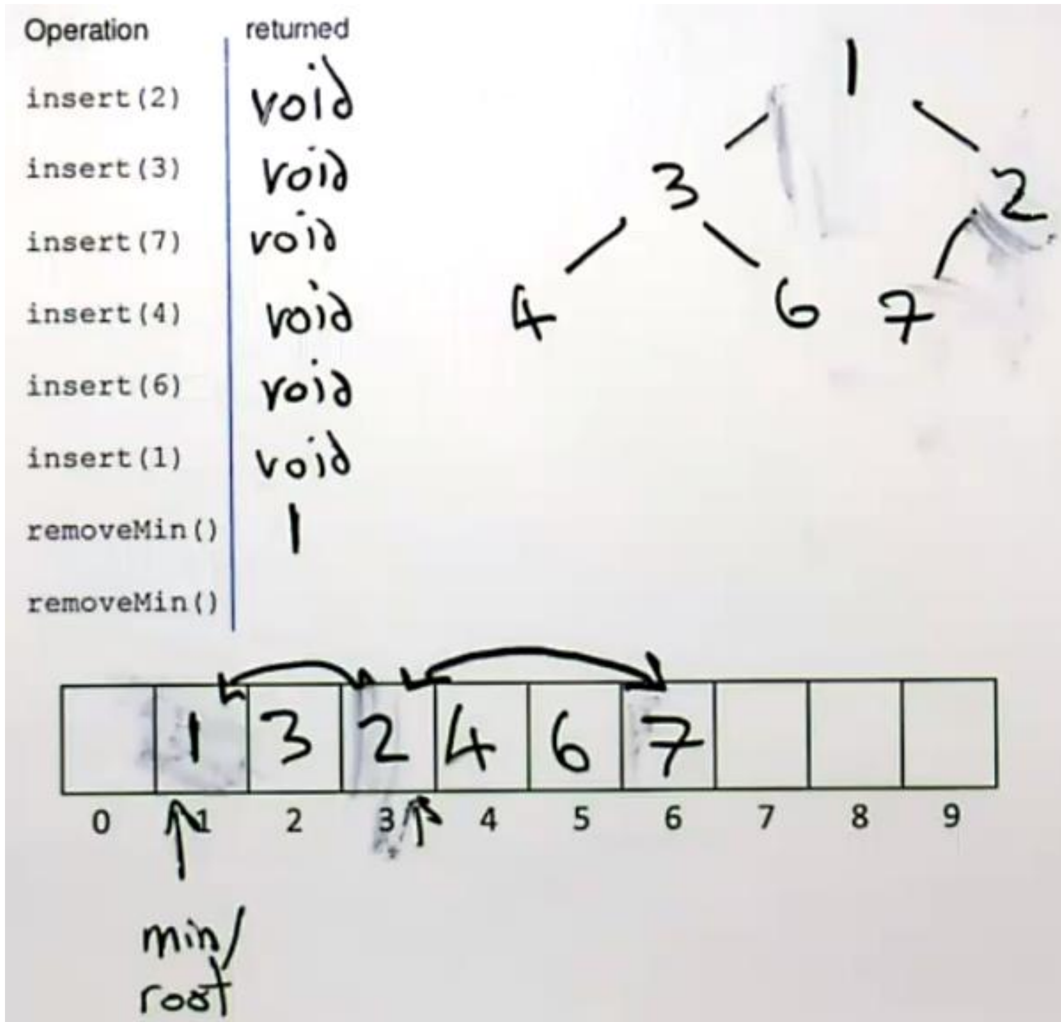
- Each entry has a key and value
  - Assumes to be the same (numbers)
- Operations
  - **size(), isEmpty(),insert(value)**,**removeMin()**, **min()**
- Implemented using a **heap**

Heaps

- A heap is a binary tree that satisfies the following properties:
  - Each node except root has a value that is <u>greater than </u>(or equal to) its parent
  - Each level are filled up before moving to next (total)
    - From left to right
- Operations
  - **removeMin**
    - Requires downheap() operation
  - **insert**
    - Requires upheap() operation
- Implemented using array



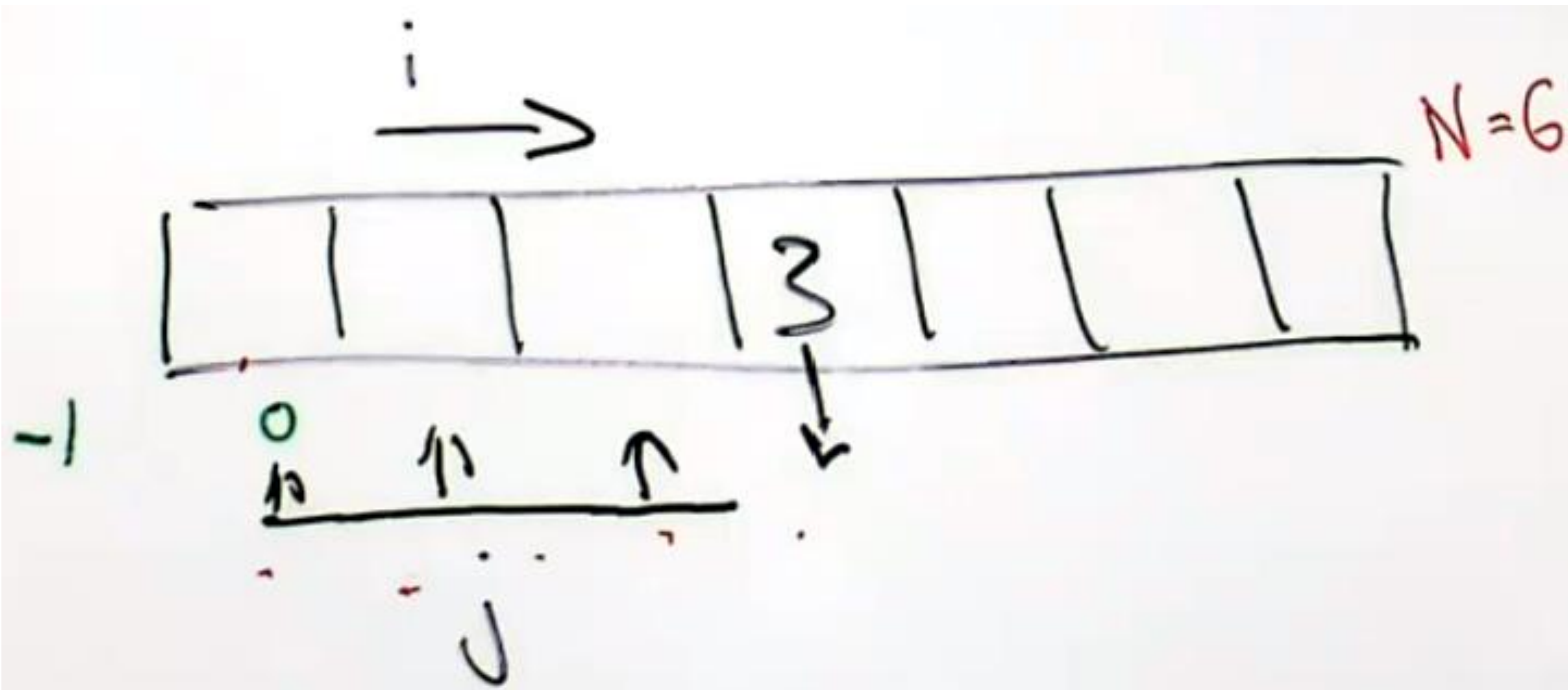
Lecture capture - *Lecture 14: priority queues*



Insertion-Sort

- **Insertion-Sort** iterates through an array, starting at the beginning
  - in the n^{th} iteration:
    - the 0..n-1 indexes of the array are sorted (the left of n)
    - while the n..length-1 indexes are not (the right of n)
  - at this step the process is:
    - the n^{th} element is moved to the correct place on the left side
      - so this remains sorted
    - all elements larger than this are shifted one place to the right
  - this continues until the end of the list is reached

Lecture capture - *Lecture 16: insertion sort and bubble sort*

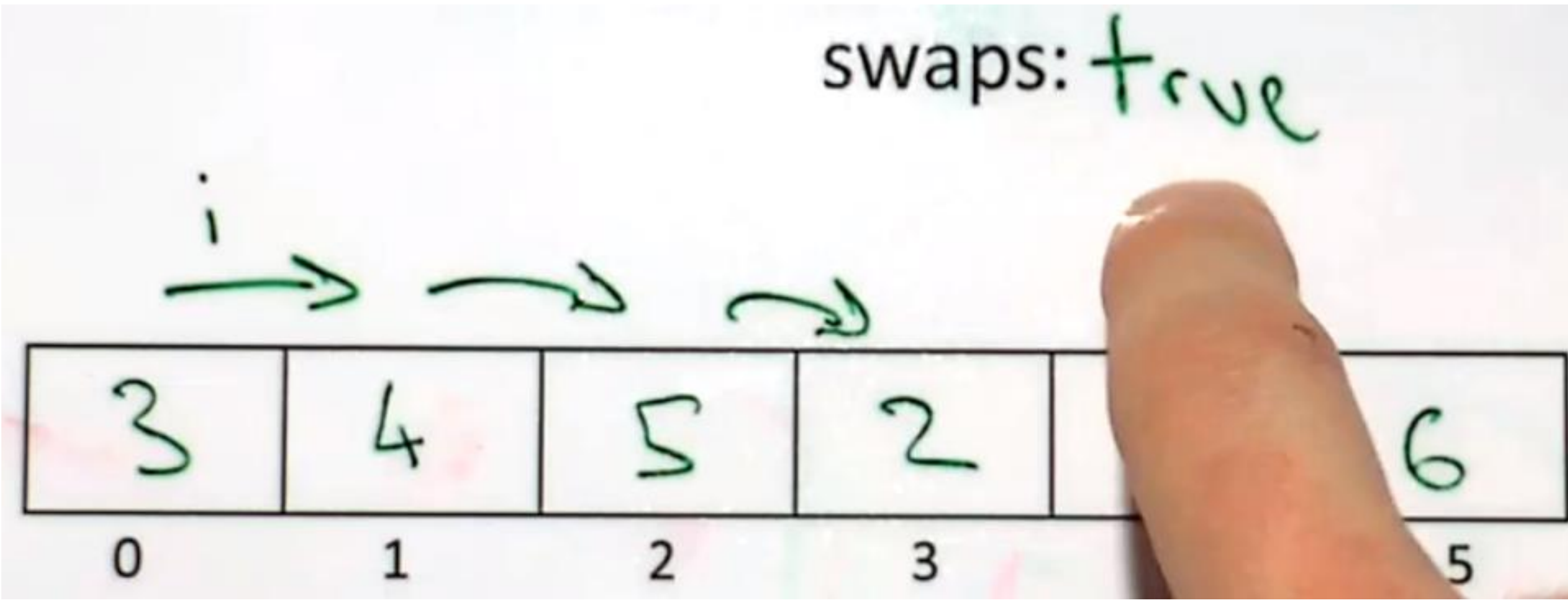


Bubble-Sort

In **Bubble-Sort** we start at the beginning of the list and iterate through the list

1. First we set a boolean variable **swaps** to **false**
2. For each step we compare with the next element
   - if the next element is smaller then we swap them and set **swaps** to **true**
3. When we reach the end of the array then
   - if **swaps** is **true** we go to 1
   - else if **swaps** is **false** we terminate

Lecture capture - *Lecture 16: insertion sort and bubble sort*

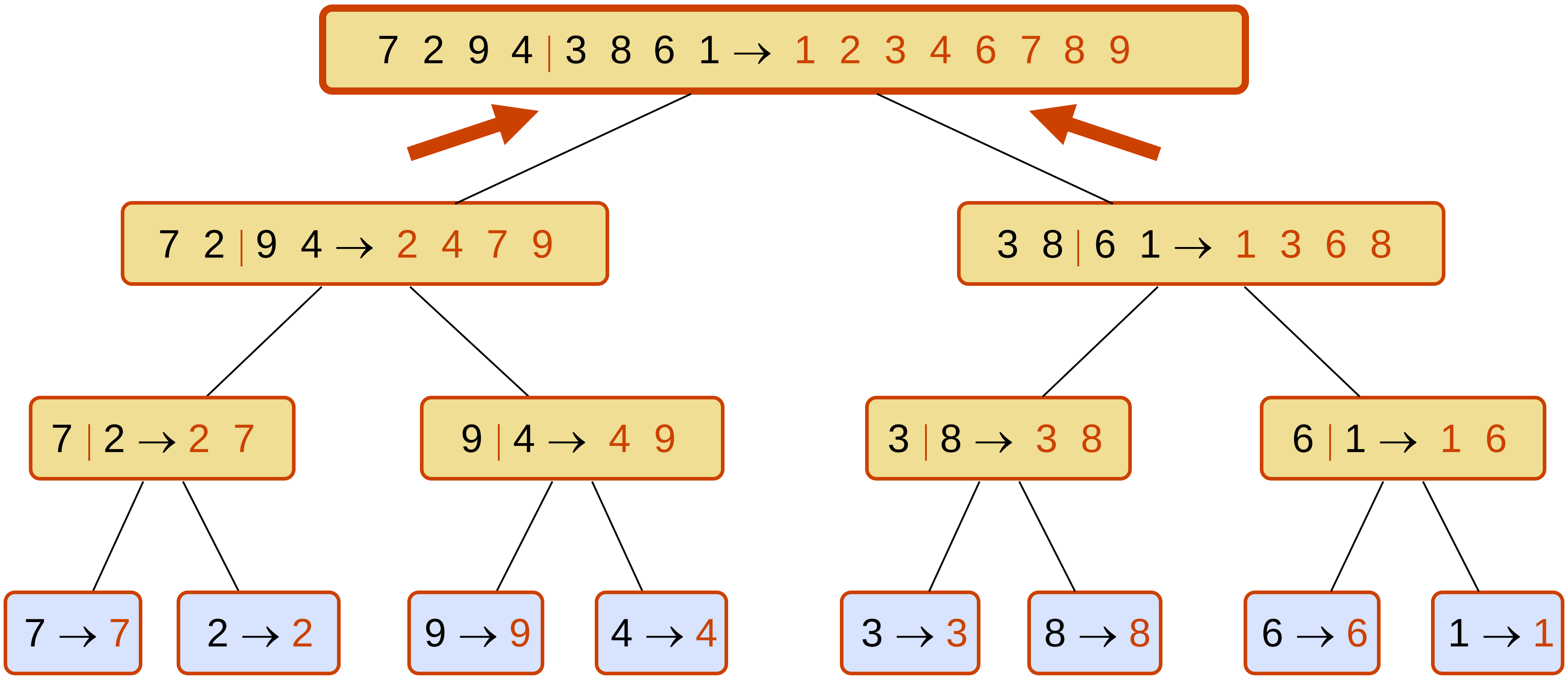


Merge-Sort

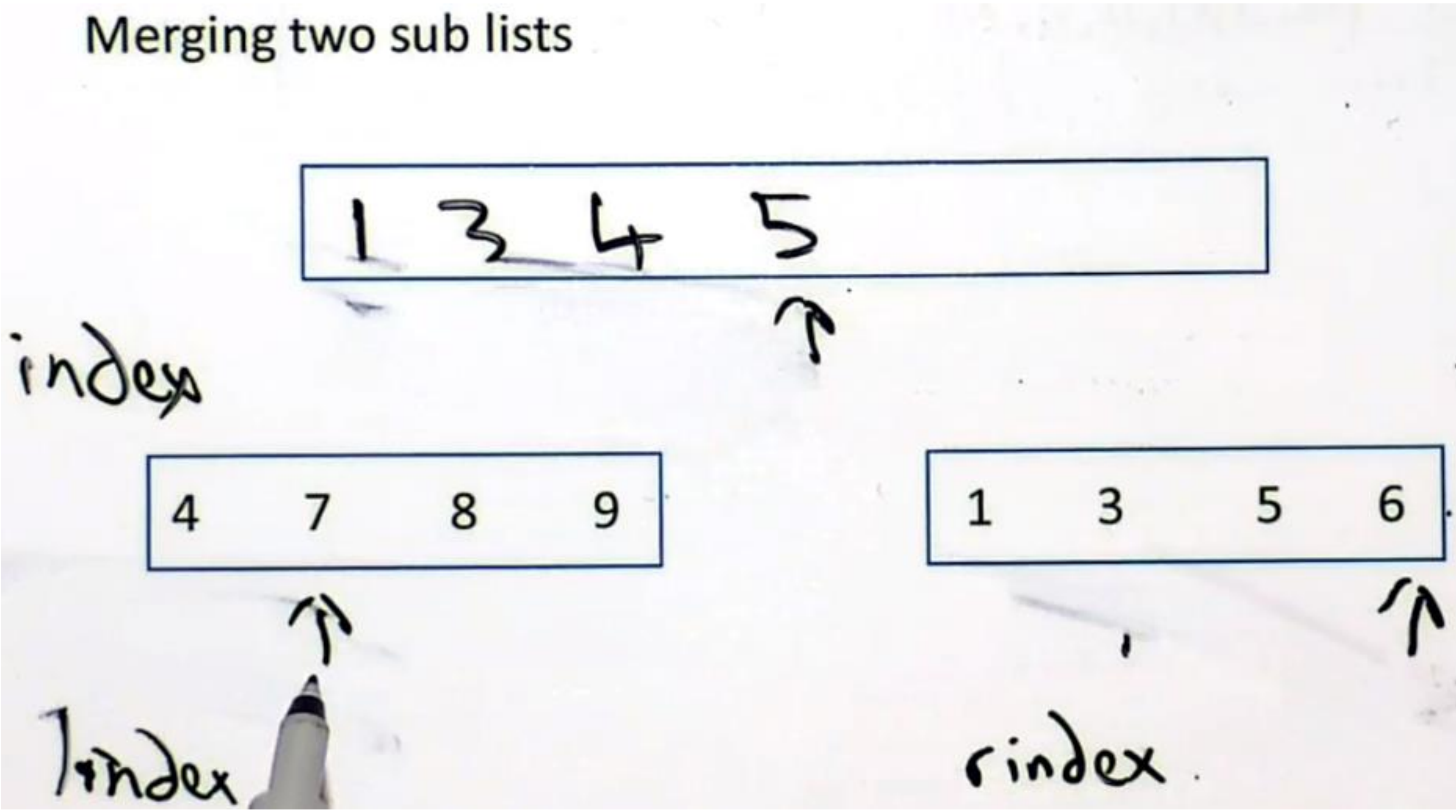
- **Base case**:
  - if the size is less than 1 return
- **Step case:**

1. split the input list into two equal halves
2. *recursively* sort the left half
   recursively sort the right half
3. merge the two sorted halves into the original list
   - always pick the smallest element from the “current” positions

Merge-Sort Example (10)



Lecture capture - *Lecture 17: merge sort and quick sort*



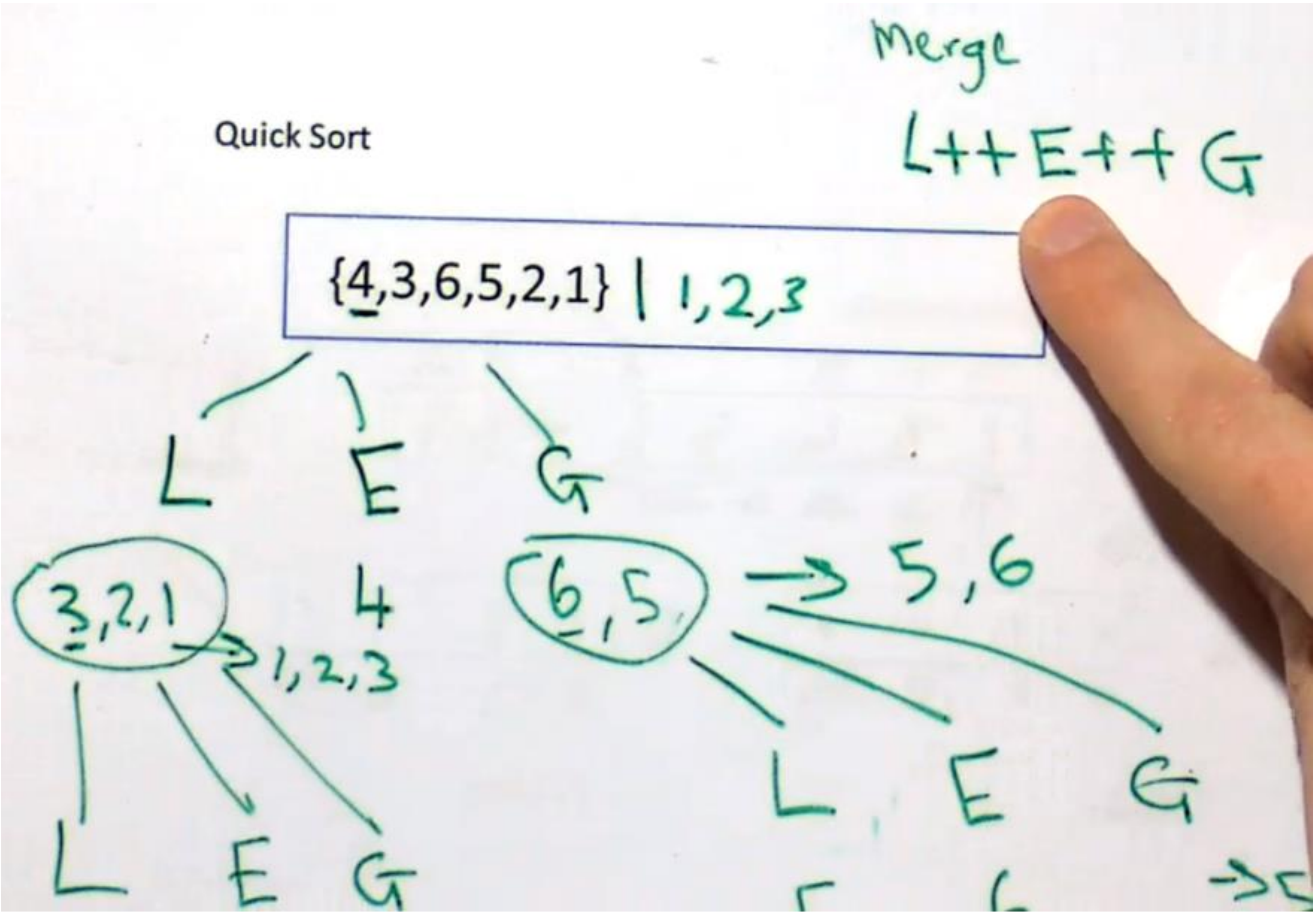
Quick-Sort

1. Select an element x from S
   - which is called the pivot
     - » E.g. first, middle or last element of S

   Divide S into 3 sub-lists:
   - L storing elements of S less than x
   - E storing elements of S equal to x
   - G storing elements of S larger than x

1. Recursively **quickSort L** and **quickSort G**
2. Put back elements in the order of
   - first elements of L,
   - then elements of E,
   - then elements of G.

Lecture capture - *Lecture 17: merge sort and quick sort*



Recursive methods

- Methods that calls itself
- The parameters are changed for each call
- We separate between two cases
  - base cases
    - where the method does not call itself
  - step cases (recursive cases)
    - where the method call itself
- You always need both!!
    - Every chain of call must eventually reach a base case

Recursion example

- The classic example of recursion is the factorial function:

  $$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$$

- Recursively, this is written:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n-1) & else \end{cases}$$

- In Java we can write this as

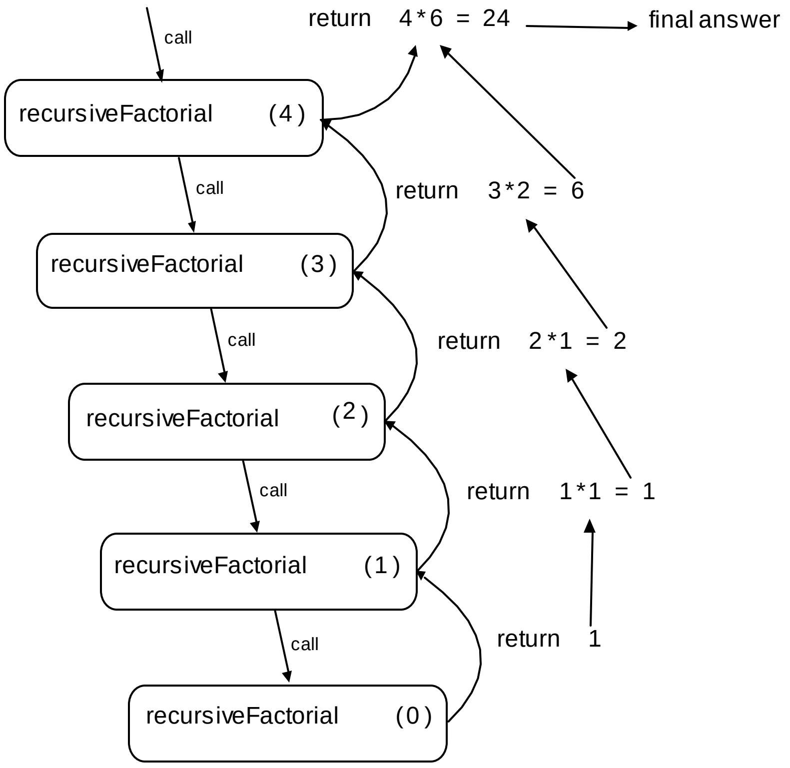
```
public static int recursiveFactorial(int n){
  if (n == 0) return 1;
  else return n * recursiveFactorial(n-1);
}
```

base

step

Recursion visualised

- The example shows how 4! is computed
  - each box is a method call
  - a (down) arrow from a caller to a callee
  - an (up) arrow from a callee to a caller with the return value




```
public static int recursiveFactorial(int n){
  if (n == 0) return 1;
  else return n * recursiveFactorial(n-1);
}
```

The Big-O Approach

A set of **primitive operations** are defined

- each assumed to have the same running time

1. We find (count) the **worst-case** number of primitive operations
   - expressed as a function on the input size
2. We then **simplify** this function to Big-Oh notation

Big-O Exercise

```
public int count(int [] arr){
  int MAX = arr.length;
  int total = 0;
  int i = 0;
  while (i < MAX){
      total = total + arr[i];
      i++;
  }
return total
}
```

- **Primitives**
  - assignment
  - calling a method
  - arithmetic operation
  - comparison
  - index into an array
  - following an object reference
  - returning from a method

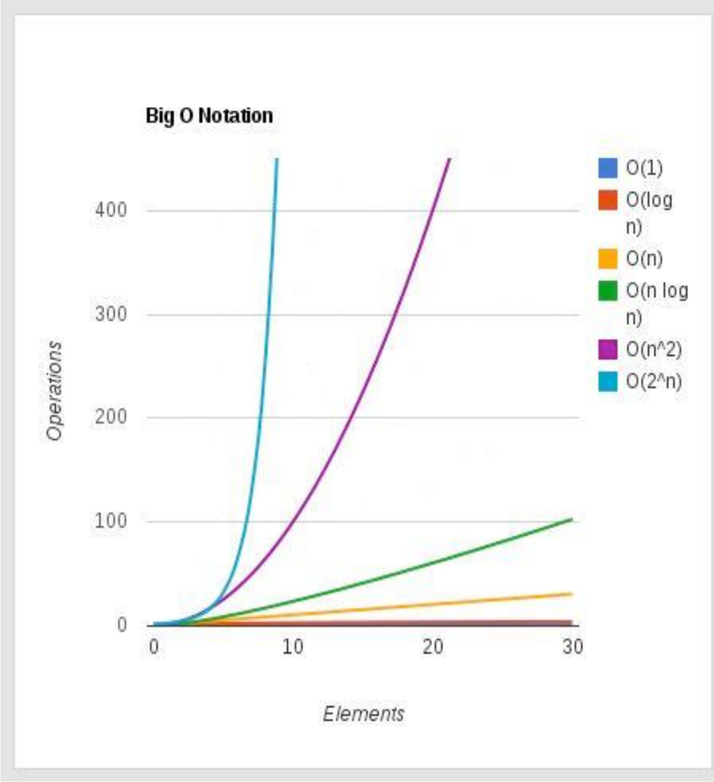
Big-O Exercise

```
public int count(int [] arr){
  int MAX = arr.length; // assignment + follow ref = 2
  int total = 0;  // assignment = 1
  int i = 0; // assignment = 1
  while (i < MAX){ I/ compare = 1
      total = total + arr[i]; // arith + array lookup + assign = 3
      i++ // (i = i+1) : assign + arith = 2
  }
return total // return = 1
}
```

- Total: 6N + 5 (where N is size of array)
- **Simplify** terms:
  - *drop lower-order terms*
  - *drop constant factors*
- We drop the lower-order term 5 (left with 6N) and constant 6 (left with N):
  - linear function O(N)

Big-O: Common Functions

- We then need to simplify the function
- The following functions are very commonly used:
  - **O(1) - the constant function**
  - **O(log n) - the logarithmic function**
  - **O(n) - the linear function**
  - **O(n log n) - the n-log-n function**
  - **O(n^2) - the quadratic function**
- This are listed in *order of complexity*
  - *O(1)* is the simplest, while *O(2^n)* is the most complex



Revision materials

- Lecture slides
  - Learning Materials -> Edinburgh -> lecture slides
- Lecture capture
  - Learning Materials -> Edinbugh -> lecture capture
- Model code solutions
  - https://gitlab-student.macs.hw.ac.uk/rs46
- **Electronic exam practise**
  - **Learning Materials -> Revision quizzes**

Guest lecture tomorrow

- **Introduction to Software Security in Java**
  - Manuel Maarek, Assistant Professor, MACS
  - James Watt 1
  - 12:15 – 13:15

… and finally …

Go through the lab solutions!

Go through lectures 3 to 17!

Good luck!

Don't panic!