IN2002 Data Structures and Algorithms

Lecture 2 – Recursion and Abstract Data Types

Aravin Naren
Semester 1, 2018/19

Learning Objectives

- Understand recursive algorithms
  - What it means for an algorithm to be recursive
  - How to analyse a recursive algorithm in terms of time and space complexities
- Abstract data types
  - What they are
  - How they are implemented

Recursion

Divide and Conquer

Many algorithms have the form:

- If the input is not simple, **divide** the input into simpler components, apply the algorithm recursively to each part, and **combine** the results obtained
- Otherwise, **solve** it with a special algorithm

Analysing divide and conquer algorithms

You have to consider, for an input size of n:

- If the input is simple
  - the time the special algorithm takes
- Otherwise,
  - the time of dividing the input, plus
  - the time of processing the components, plus
  - the time of combining the results
  - ... and then solve the recursive definition.

Recursion *

A powerful way of programming using a function within itself

- A problem is addressed by identifying:
  - A step towards the next simpler case to which the same approach applies

  AND
  - Result in the simplest case
- Recursion is, in a way, the reverse of proof by induction

* Introduced to you in Computation & Reasoning

Example: Factorial

Problem: calculate $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

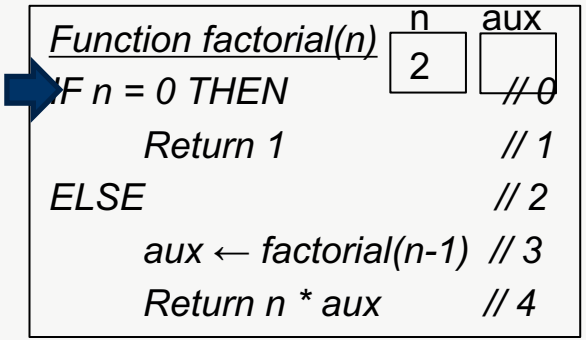
- Step (for $n>0$): $n! = n \cdot (n-1)!$
- Base (simplest) case: $0! = 1$

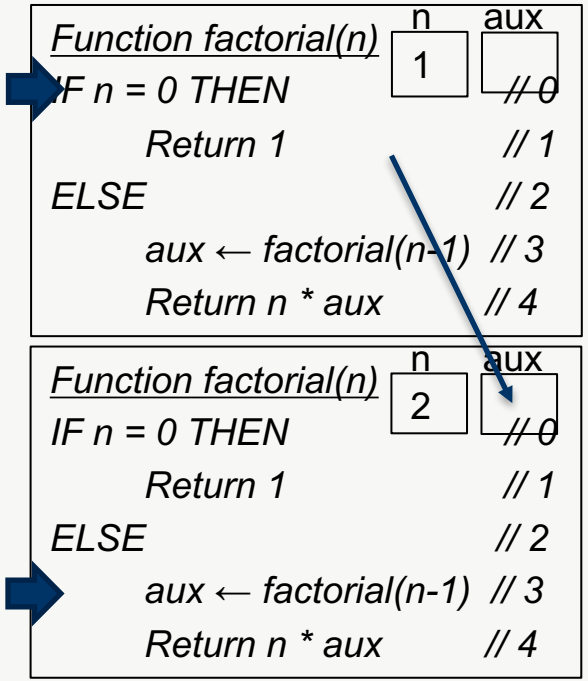
This defines the value of n! for any natural number n.

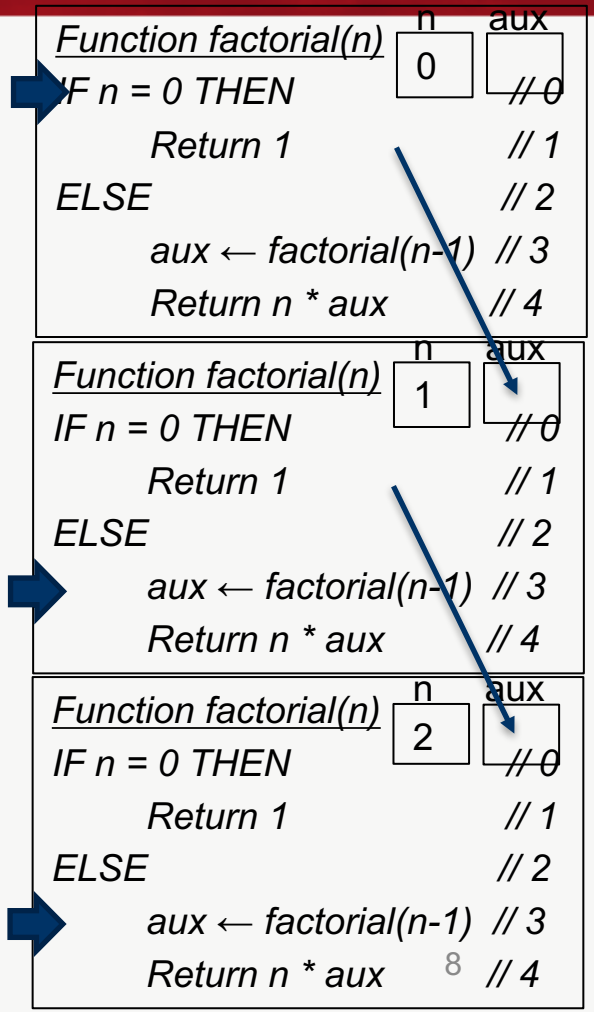
```
Function factorial(n)
IF n = 0 THEN                    // 0
    Return 1                     // 1
ELSE                             // 2
    aux ← factorial(n-1)         // 3
    Return n * aux               // 4
```

What is going on?

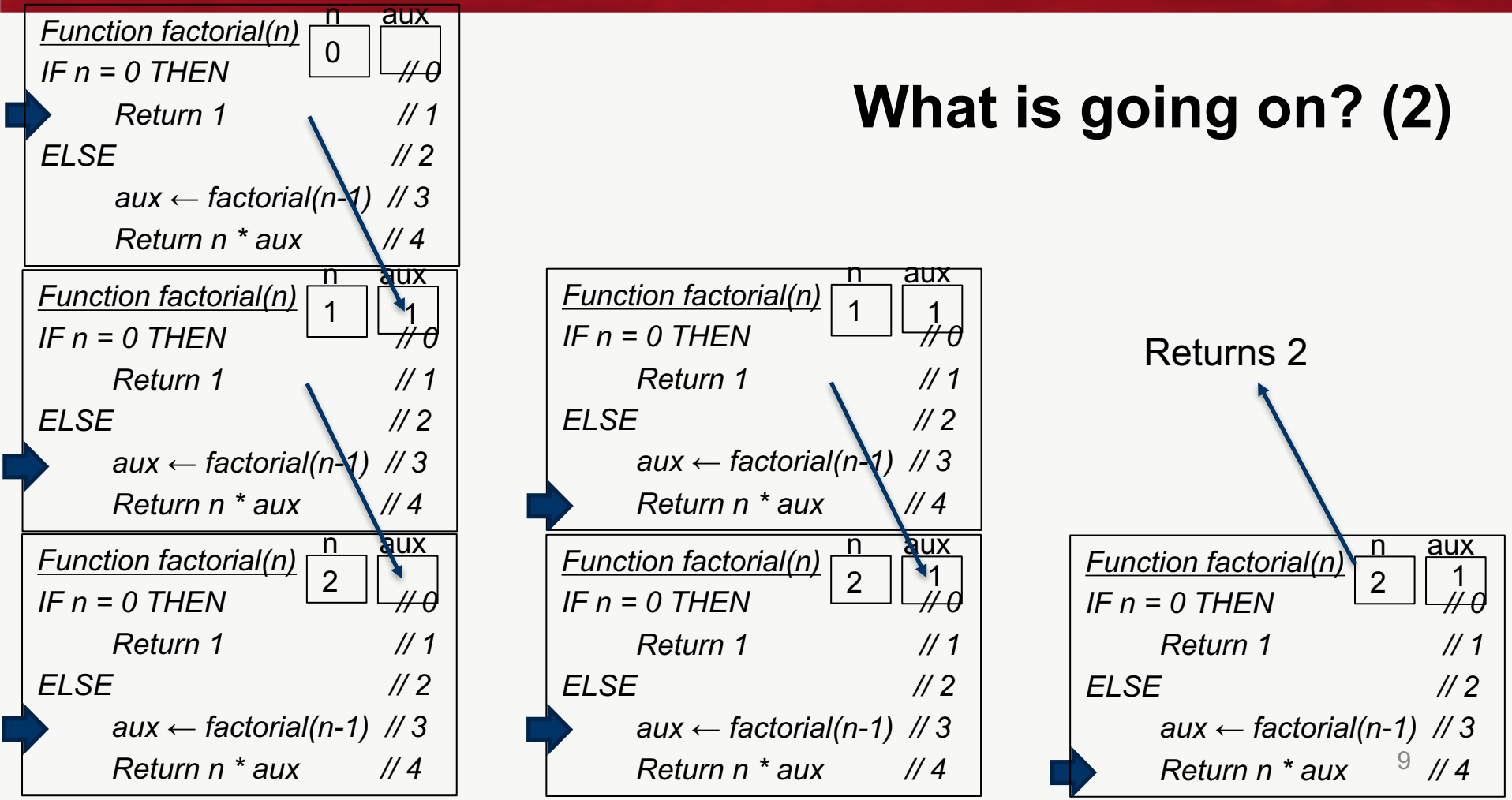
Calling factorial (2)







What is going on? (2)

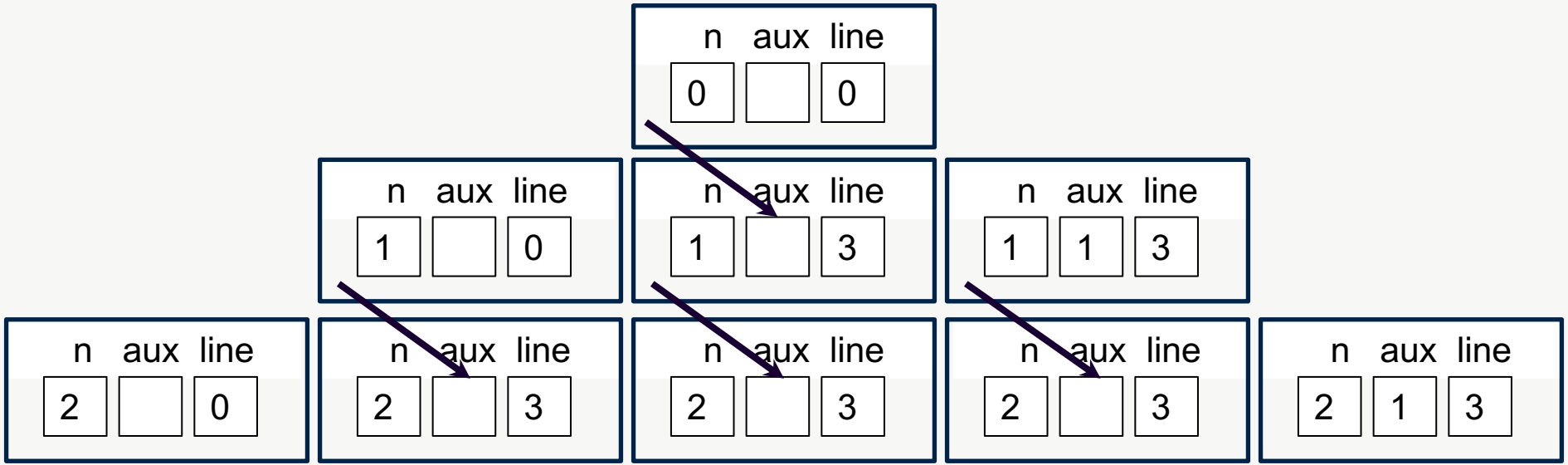


Understanding the Picture

- Recursive calls to factorial get pushed onto the stack
  - Multiple copies of function factorial, each with different arguments & local variable values, and execution at different lines of the code

- All the computer really needs to remember for each active function call is the values of arguments & local variables and the location of the next statement to be executed when control goes back

The Stack


n aux line
0 0
n aux line
1 0
n aux line
1 3
n aux line
1 1 3
n aux line
2 0
n aux line
2 3
n aux line
2 3
n aux line
2 3
n aux line
2 1 3

What are “factorial’s” the time and space complexities?

- Time complexity
  - Calls itself n times
  - All other statements constant
  - … hence O(??)
- Space complexity
  - Calls itself n times (stack)
  - … hence O(??)

Example: power

Problem: calculate x^n

- Step (for $n>0$): $x^n = x \cdot x^{n-1}$
- Base case: $x^0 = 1$

This defines the value of x^n for any natural number n.

Function power(x, n):

```
IF n = 0 THEN
    Return 1
ELSE
    Return x * power(x, n-1)
```

What are “power’s” the time and space complexities?

- Time complexity

- Space complexity

Tail Recursion

```
void tail(int i) {
    if (i > 0) {
        System.out.println(i);
        tail(i-1);
    }
}

void tailloop(int i) {
    while (i > 0) {
        System.out.println(i);
        i=i-1;
    }
}
```

Tail recursion is when the recursive call is the last thing done.

A loop can easily replace it.

Example of Tail Recursion: factorial

This factorial function is tail-recursive:

```
int factorial(int n) {
    return fact(n, 1);
}

int fact (int n, int product) {
    if (n == 0){
        return product;
    }
    return fact(n-1, n*product);
}
```

... Tail Recursive factorial

```
int fact (int n, int product) {
    if (n == 0)
        return product;
    return fact(n-1, n*product);
}
```

```
int fact(int n, int product) {
    while(n>0) {
        product = n*product;
        n = n-1;
    }
    return product;
}
```

And can easily be converted into iterative (i.e. a loop).

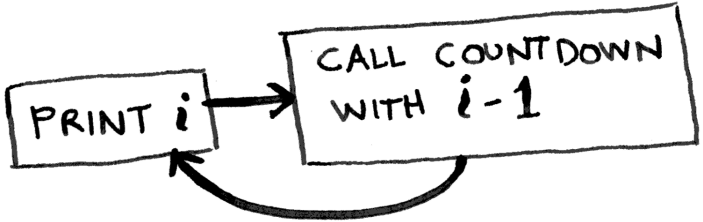
Another Example of Recursion

What does this function do?

```
void printNum(int n) {
    if (n >= 10){
        printNum(n/10);
    }
    System.out.print(n%10);
}
```

A Call Tree for printNum

```
printNum(1234)
        printNum(123)
                printNum(12)
                        printNum(1)
                                System.out.print('1')
                        System.out.print('2')
                System.out.print('3')
        System.out.print('4')
```



What are “printNum’s” the time and space complexities?

- Time complexity
  - Calls itself log n times
  - All other statements constant
  - … hence O(??)

- Space complexity
  - Calls itself log n times (stack)
  - … hence O(??)

Another example for recursion (exam 2014-15)

Recursion (15%)
You are given the code for *bla* below.

```
void bla (int i)  {
    if (i > 0) {
        System.out.print (i + " ");
        bla(i – 1);
    }
}
```

Another example for recursion (exam 2014-15) – cont'd

a) Describe what *bla* does. (5%)

b) What is its time complexity? Justify your answer. (5%)

c) What is its space complexity? Justify your answer. (5%)

Applications of Recursion

Recursion is useful for:

- designing algorithms
- traversing branching structures (like trees and graphs)
- writing parsers
- backtracking search algorithms
- divide-and-conquer algorithms

Pros and Cons of Using Recursion

- Pros
  - often elegant solutions
  - compact code
  - proving correctness often relatively easy
- Cons
  - careful algorithm design necessary
  - hard to debug (better prove correctness)
  - function call stack needs extra memory and is often limited

Common Computations

- Why do we often discuss sorting algorithms?
- How about searching?
- Insertion?
- Deletion?

Sorting Algorithms

- What are examples of sorting algorithms?

- How fast are they?

Sorting Algorithms

	Space complexity	Average time complexity	Worst case time complexity
Selection sort	O(1)	$O(n^2)$	$O(n^2)$
Insertion sort	O(1)	$O(n^2)$	$O(n^2)$
Quicksort	O(log n)	O(n log n)	$O(n^2)$
Mergesort	O(n)	O(n log n)	O(n log n)

Other Common Computations

- How about searching?
- Insertion?
- Deletion?

Deletion Algorithm

- How do we delete an element of an array?
- What if the array is sorted?
- How often do we need to delete new elements?

Food for Thought

- Should we always use arrays to store and manipulate our data?
- Should we always sort their contents?

Abstract Data Types

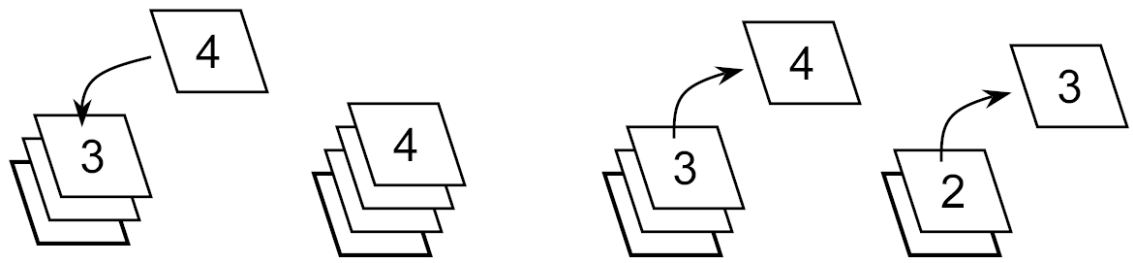
Abstract Data Types (ADTs)

- Abstract data types
  - separate functionality from implementation
  - hidden representation
  - manipulated via a limited set of operations
  - in Java, interfaces with defined method semantics
- Data structures
  - concrete implementations, often subject to an invariant preserved by the operations
  - in Java: classes implementing ADT interfaces
  - are analysed by measuring the time and space complexity of the structure and each operation

ADT Stack

Think of physical stacks:

- series of similar elements
- you add elements on the top
- you take the elements from the top



... so the last element in is the first out (LIFO)

Stack Operations

You can only remove the last element added.

The operations are thus:

- Add an element to the stack
- Remove the last element added
- Check whether the stack is empty.

A stack in Java

```
// a stack of integers
public interface Stack {

    // is the stack empty?
    boolean isEmpty();

    // add (push) an element into the stack
    void push(int elt);

    // remove and return the most recently pushed
    // element still in the stack
    int pop();
}
```

A stack implementation using an array

```
public class ArrayStack implements Stack {
    private int[] a;
    private int count = 0;

    public ArrayStack(int size) {a = new int[size];}

    public boolean isEmpty() {return count == 0; }

    public void push(int elt) { a[count++] =   elt; }

    public int pop() { return a[--count]; }
}
```

Checking Brackets with a Stack

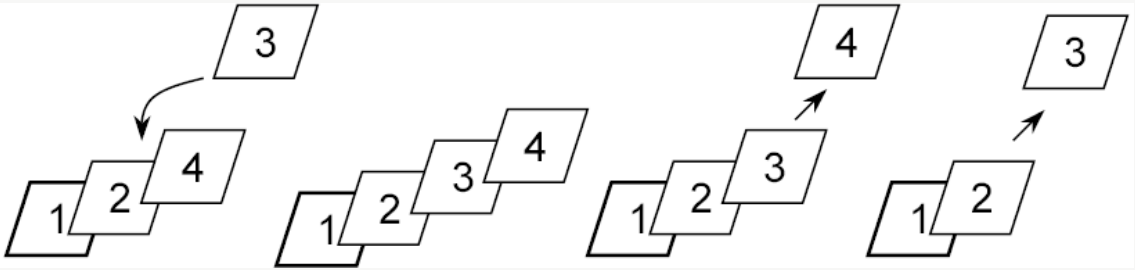
opening bracket: push on stack

closing bracket: pop from stack and match

String	Stack	Op
([ ] ([ )])		push
[ ] ([ )])	(	push
] ([ )])		
([ )])		
[ )])		
)])		

Priority Queues

- You can only remove the element with the highest priority
- Examples: printer queues, emergency rooms, council housing



Priority Queue Operations

- Add an element
- Remove an element: the element removed is the one with the highest priority
- Check whether the queue is empty

Applications of Priority Queues

- As a component of various algorithms
- Sorting: add all the items to the priority queue and then extract them one by one
- Scheduling (e.g. in operating systems, post, air traffic)
- Event-driven simulation

Scheduling

Several tasks should be performed with different priorities

For example…

We want to produce a system for parcel deliveries. The priority could be set by using the guaranteed date of delivery and the time at which the parcel was posted.

- The priority queue would be handled by storing the parcels in the depot and releasing them for delivery according to their priority.
- In busy days, probably only those parcels whose delivery is due - i.e., those that have the highest priority - could be delivered.
- In less busy days, parcels would be delivered in advance of their guaranteed date of delivery.

Event-Driven Simulation

We wish to simulate a series of events, each considered to occur at a point in simulated time.

Nothing happens between events, so maintain a collection of pending events, and repeat:

- extract the first pending event
- advance the simulation clock to its time
- simulate the event

Events may change the state, and may cause other events at later times, i.e. add further events to the collection

Useful for load analysis and prediction

e.g., for Internet nodes, road or air traffic, emergency procedures.

A priority queue ADT

```
// A priority queue of integers
public interface PriorityQueue {

    // Is the priority queue empty?
    boolean isEmpty();

    // Add an element to the priority queue
    void add(int elt);

    // Remove and return the largest value
    // currently in the priority queue
    int extractMax();
}
```

An implementation using a sorted array

A way of implementing priority queues is to keep the elements in a sorted array, so:

- *to add an element, search and insertion are used*
- *to extract an element, the last element in the array is extracted (as it will have the highest priority)*

Concrete Implementation in Java

```
public class ArrayPQ implements PriorityQueue {
    private int[] data;
    private int count = 0;

    public ArrayPQ(int size){ data = new int[size]; }

    public boolean isEmpty(){ return count == 0; }

    public void add(int elt){
        insert(data, count++, elt);
    }

    public int extractMax(){ return data[--count]; }
}
```

Analysing the use of the Data Structure

- The data structure takes O(n) space complexity

- By using the ordered array data structure:
  - `isEmpty` takes time O(??)
  - `add` takes time O(??)
  - `extractMax` takes time O(??)

- The ordering constraint is very restrictive, maintaining it by insertions costs O(n) time.

An Alternative Implementation

What would happen if the representation were an unordered array?

In that case:

- `isEmpty` takes time O(??)
- `add` takes time O(??)
- `extractMax` takes time O(??)

It is an equivalent situation, but slow at extracting.

Is there a better implementation?

Reading

- Weiss: Chapter 7, and chapter 15 section 1.1
- Drozdek: Chapter 5, and chapter 4 sections 4.1and 4.3

Next week: Trees and heapsort

City, University of London
Northampton Square
London
EC1V 0HB
United Kingdom

T: +44 (0)20 7040 8406
E: ug-compsci@city.ac.uk
www.city.ac.uk/department-computer-science