Academic excellence for
business and the professions

IN2002 Data Structures and Algorithms

Lecture 3 – Trees, Heaps and Queues

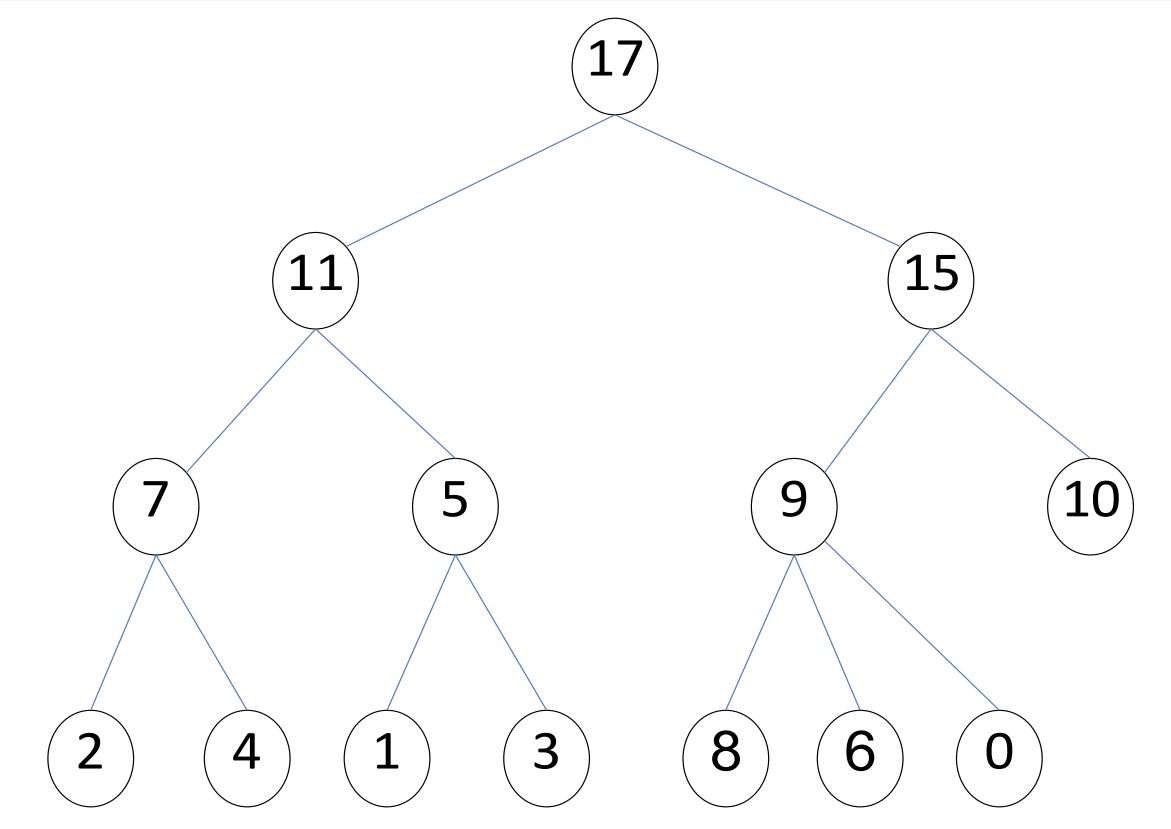
Aravin Naren
Semester 1, 2018/19

Learning Objectives

- Understand and be able to use:
  - the abstract data type queue
  - the data structures:
    - Heaps
    - Extensible arrays
- Be able to understand, apply and develop algorithms to those above, most notably:
  - Adding elements
  - Extracting elements
  - HeapSort

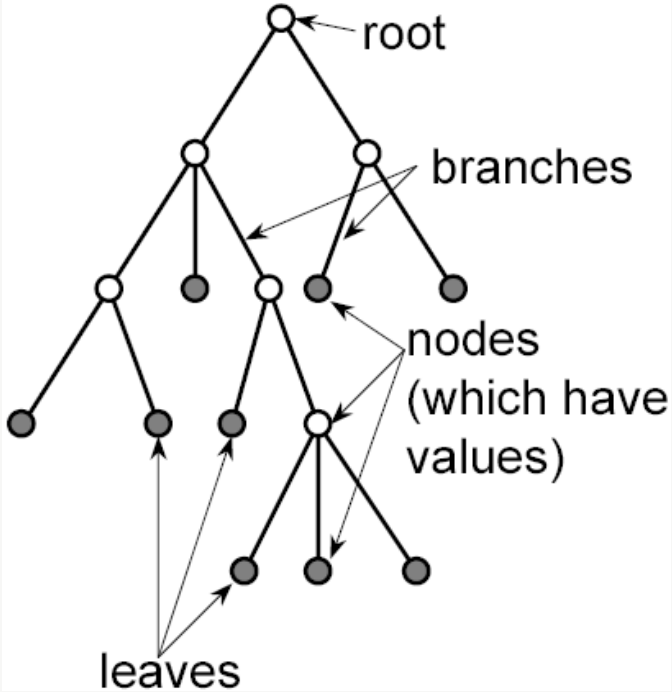
Trees

What is this?



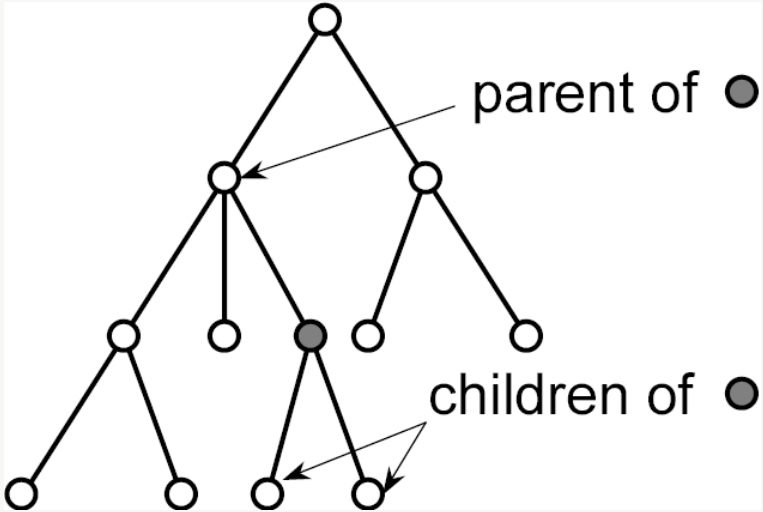
Quick introduction to trees

- Trees are data structures that have a root, nodes, branches and leaves
- Most nodes are at the end of a branch. The only exception is the root (which is the top node!)
- Branches proceed from nodes
- Nodes without branches are called “leaves”



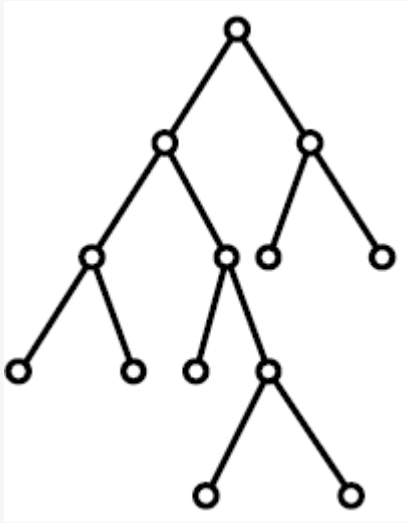
More tree terminology

- The nodes n2,n3 at the end of a node n1's branches are its children, and n1 is the parent of n2,n3.
- The root is thus a node with no parents.
- The leaves are nodes with no children.

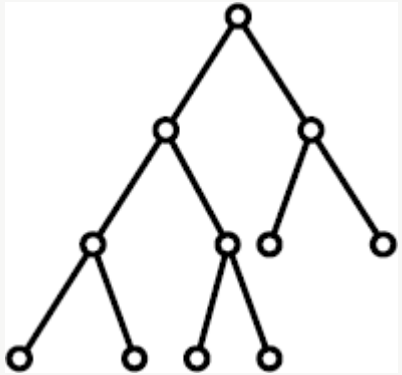


Tree types

A **binary** tree is a tree in which each node has, at most, two children



A **perfectly balanced** binary tree is a tree whose least deep leaf is no more than one level apart from the deepest one



Last week…

Priority queues

	Ordered Array	Unordered Array
isEmpty	O(?)	O(?)
add	O(?)	O(?)
extractMax	O(?)	O(?)

Is there a better implementation?

Using heaps (a kind of tree) we can distribute and reduce the load, achieving O(log n) time for both `add` and `extractMax`.

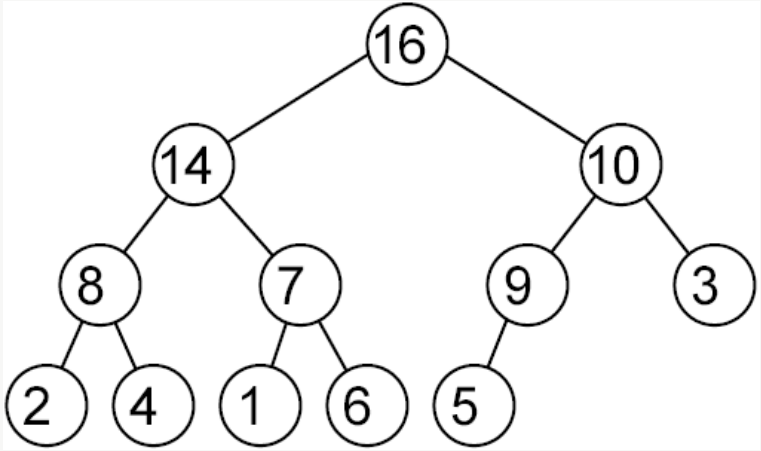
* Introduced to you in Computation & Reasoning

Heaps

Heaps

A heap is a perfectly balanced binary tree such that:

- No node is larger than its parent
- All items on the lowest level are as far to the left as possible

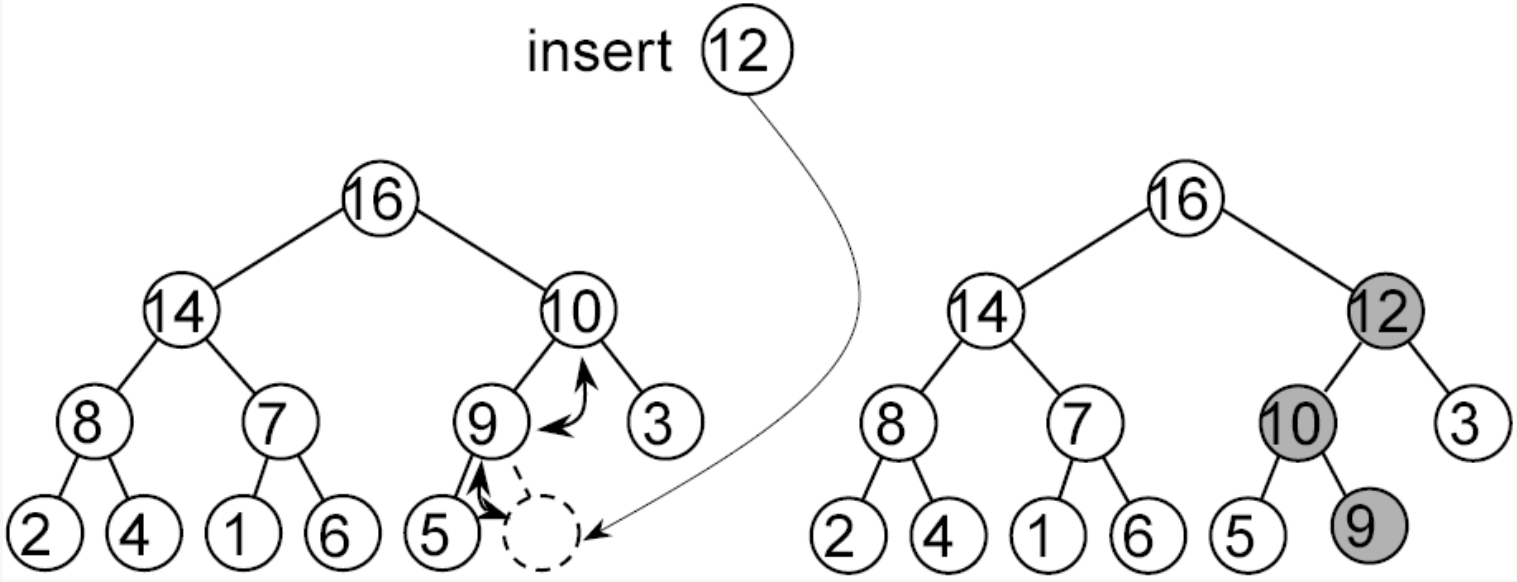


Heap Operations

- Heap is empty if the root is empty
- Heap insertion: add a node to the heap—displacing other nodes as necessary so that the heap remains a heap
- Heap extraction: remove the element with the highest value in the heap (the root), moving the other nodes as necessary so that the heap remains a heap

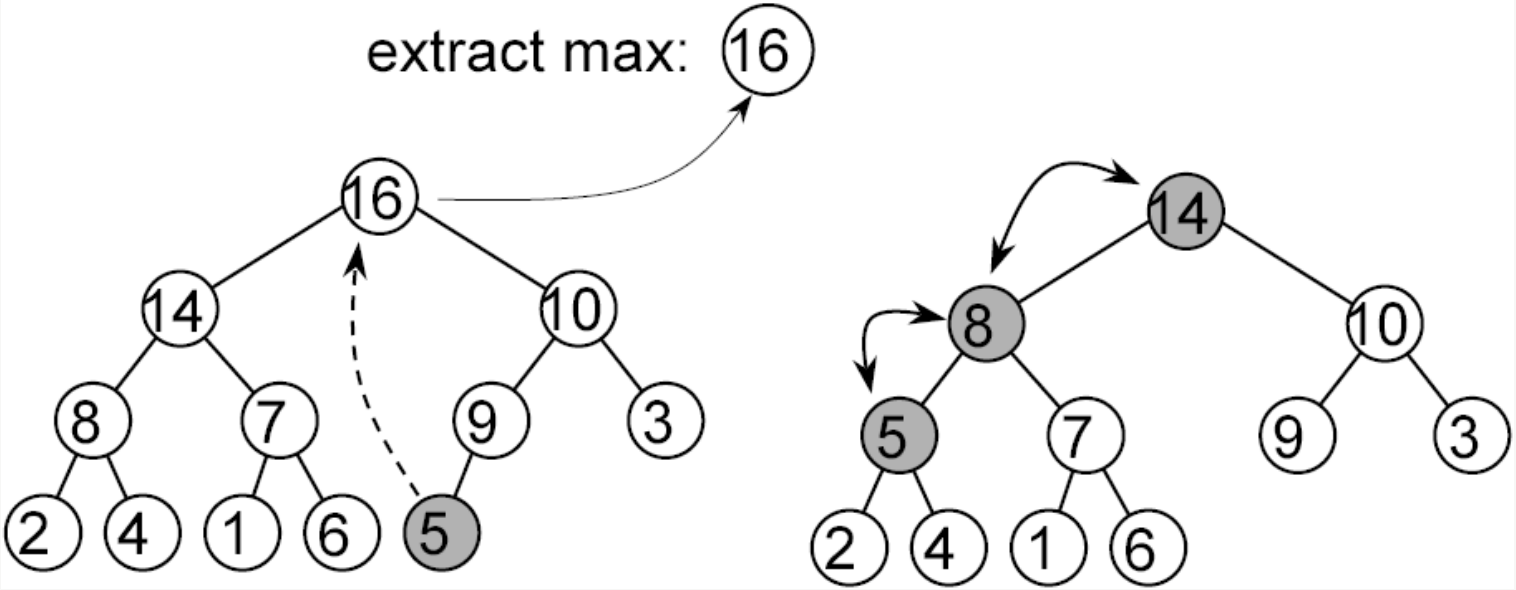
Heap Insertion

- Add the new node as a leaf
- Sift the new node up to its correct position (at most log n swaps—because the tree has at most log n levels)



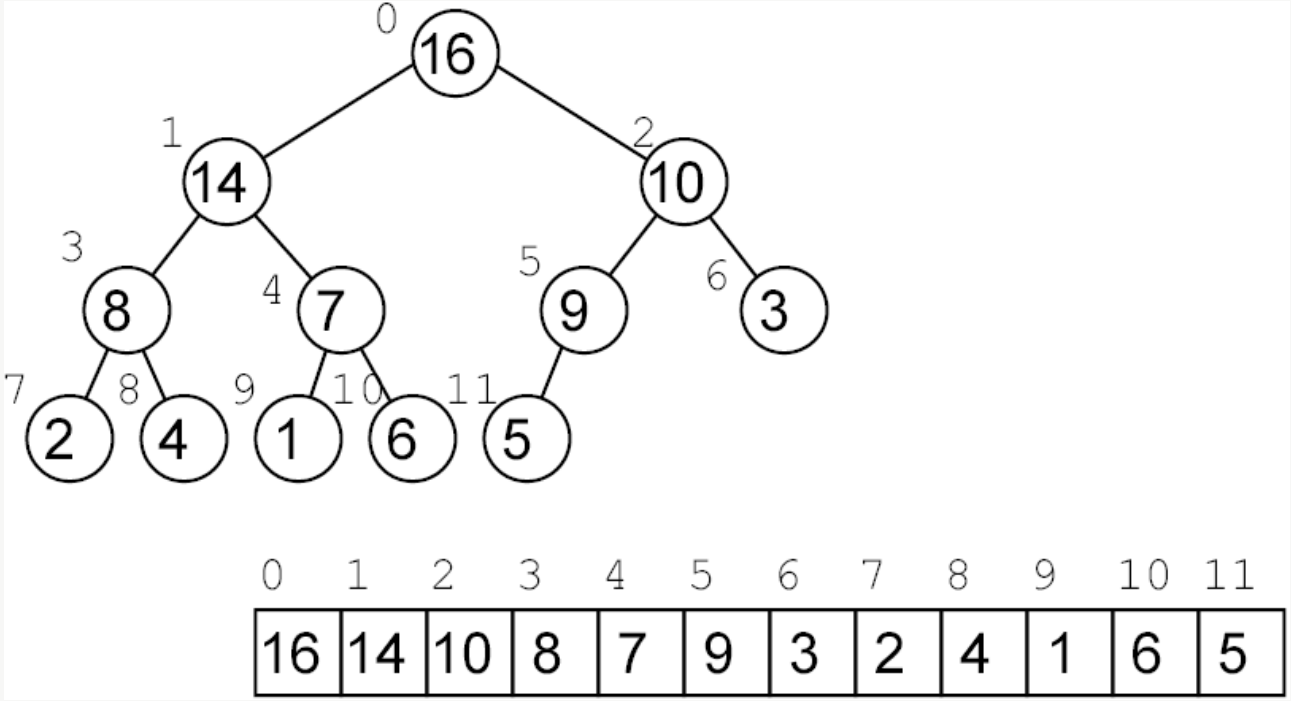
Heap Extraction

- Extract the root, and place the last leaf in the root position
- Re-establish the heap conditions (at most log n swaps)



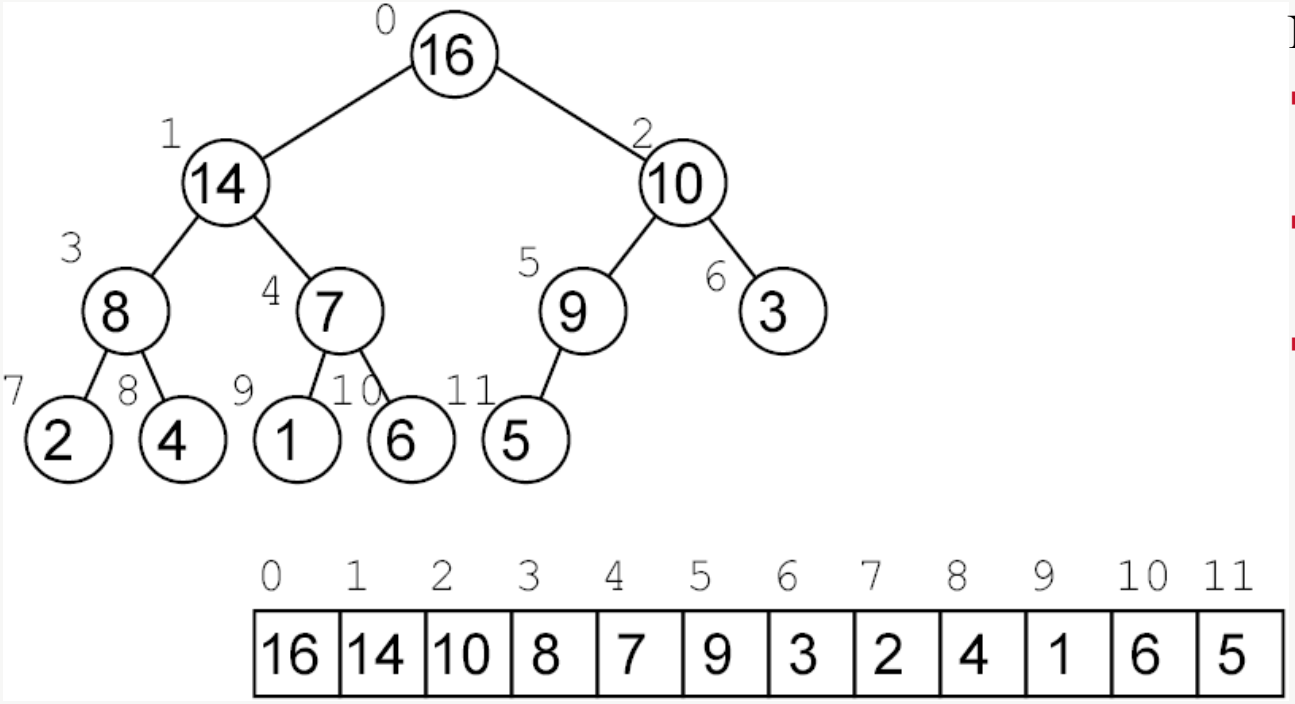
Heaps and Arrays

- Heaps can be represented as arrays:



0	1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	6	5

Relating indices



Notice that:

- the parent of node n is node (n-1)/2 (int division)
- the left child of node n is node 2n + 1
- the right child of node n is node 2n + 2

An Implementation in Java

```
public class HeapPQ implements PriorityQueue {
   private int[] data; // data[] is the heap
   private int count = 0;

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

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

... HeapPQ continues

Relating Indices in Java

You will remember that:

- the parent of node n is node (n-1)/2
- the left child of node n is node 2n + 1
- the right child of node n is node 2n + 2

```
private int parent(int n) { return (n-1)/2; }
private int left(int n) { return n*2 + 1; }
private int right(int n) { return n*2 + 2; }
```

... HeapPQ continues

Heap Insertion

Function add(elt) where data is the array containing the heap and count is the heap size:

```
pos ← count+1
WHILE pos > 1 AND data[parent of pos] < elt
    data[pos] ← data[parent of pos]
    pos ← parent of pos
data[pos] ← elt
count ← count + 1
```

Heap Extraction

Function extractMax() where "data" is the array containing the heap and "count" is the heap size:

```
max ← data[1]
data[1] ← data[count]
count ← count - 1
moveDown(data, 1, count)
Return max
```

Function *moveDown(data, first, last)* moves the element at position *first* down to its proper position

Moving a key down the heap

Function moveDown(data, first, last):

```
WHILE left child of first <= last
  larger ← position of the child with the larger value
    IF data[first] >= data[larger] THEN
        Break the while loop
    swap first and larger in data
  first ← larger
```

Remember that *first*, *larger* and *last* are positions in array *data*.
The value of *largest*, for example, is *data[largest].*

Sorting using heaps

Heap sort involves:

- adding the elements to a heap, and then
- extracting the elements from the heap

This could be done with just one array:

- in the first phase, the heap grows as the unsorted section shrinks
- in the second phase, the heap shrinks as the sorted portion grows.

HeapSort in Java

```
void heapsort(int[] data) {
    \\ first phase: heap goes from data[i+1] to
    \\ data[data.length-1]. The rest of data[] is
    \\ unsorted
    for (int i = data.length/2 - 1; i>= 0; i--)
        moveDown(data, i, data.length-1);
    \\ second phase: heap goes from data[0] to data[i]
    \\ the rest contains the largest elements, sorted
    for (int i = data.length - 1; i>= 1; i--) {
        swap(data, 0, i);
        moveDown(data, 0, i-1);
    }
}
```

HeapSort Example

- Use heapSort to sort [1 20 3 10 8 7 40]

HeapSort Analysis

- Heap sort performs O(n) heap operations, each taking O(log n) time, in total **O(n log n) time** in the **worst case**.
- Heap sort and quicksort have both **average time complexity of O(n log n)**, but heap sort is normally slower than quicksort (different constant factors).
  → Quicksort is preferred when no guaranteed response time needed.
- **Heap sort** requires only **O(1) extra memory space**, in contrast to **O(n) for merge sort** or **O(log n) for quicksort** (on average, O(n) in the worst case).
- A refinement: build the heap from the bottom up. This can be shown to take O(n) time, but the O(n log n) selection phase still dominates.

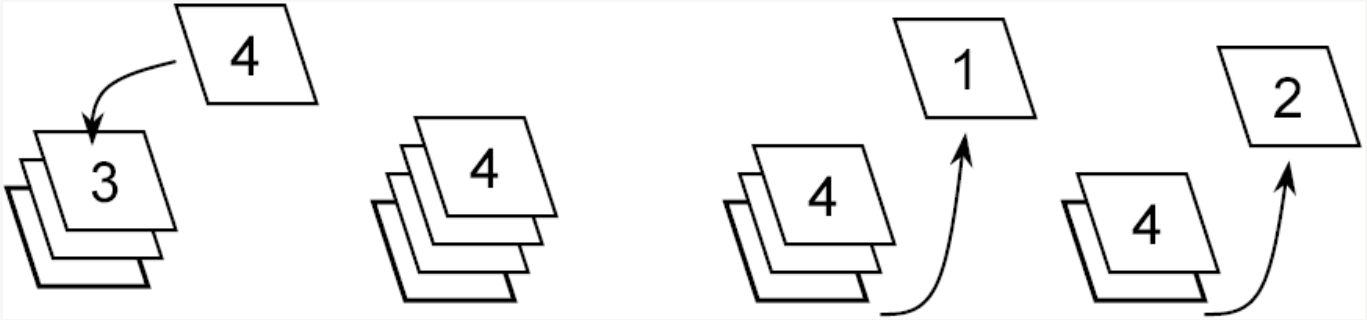
Dynamic Data Structures

Dynamic Data Structures

- Amounts of data to store often vary in size in ways we can not determine beforehand
- Reallocate-and-copy of arrays can solve this problem
- In general, linked structures are more flexible:
  - singly linked lists
  - doubly linked lists and circular lists
  - trees and graphs

Queues

- First In First Out (FIFO)
  - As opposed to last in first out (stacks)
  - And also to priority queues (in which priorities may not reflect the actual order of addition)



Queue Operations

You must always remove the element that has been in the queue the longest.

The operations are thus:

- Add an element to the queue
- Remove the element added the longest time before
- Check whether the queue is empty.

A Queue ADT

```
public interface Queue {
        // Is the queue empty?
        boolean isEmpty();
        // Add an element to the queue
        void enqueue(int elt);
        // Remove and return the earliest element of
        // the queue
        int dequeue();
}
```

A Stack ADT

```
// 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();
}
```

Extending arrays by relocate-and-copy

Because size is unpredictable:

- initial array allocation may fall short
- trying to be safe is wasteful

A solution: extensible arrays

- start with a small array
- if more space is needed, create a new, larger array and copy elements into it

... adds O(n) to space and O(n) to time per extension

Stacks with relocate-and-copy

```
public class ArrayStack implements Stack {
      private int count = 0;
      private int[] data = new int[1];

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

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

... continues

Stacks with relocate-and-copy (ctd.)

... which was the same as last week. However:

```
    public void push(int elt) {
        if (count == data.length) { // not enough space
            int[] d1 = new int[data.length + 1]; // new
            for (int i = 0; i < count; i++) // array
                d1[i] = data[i];    // copy elements
            data = d1;              // use new array
        }
        data[count] = elt;      // add new element
        count++;
    }
}
```

Amortised Analysis

Amortised complexity refers to the ***average complexity over a sequence of operations***, which normally depend on each other (usually choosing the worst case regarding data input).

This is different from what we normally call ***average case*** complexity, which ***averages over different input data*** for a single, independent operation.

E.g., the average complexity (on an array of length n) of Quick Sort is O(n log n), the amortised complexity over a sequence of Quick Sort applications is $O(n^2)$
(implying the worst case per application).

Analysis of the implementation

In a sequence of n pushes, the number of copies is

$$1 + 2 + 3 + \ldots + (n - 1) = n(n - 1)/2 = O(n^2)$$

This means that **per *push* operation**, there is an **amortised cost of O(n).**

Going up by a larger step than 1 improves things, but
only by a constant factor.

… An improvement

If we **double the size whenever we extend**, the number of elements copied for $n = 2^k$ pushes is

$$1 + 2 + 2^2 + \ldots + 2^{k-1} = 2^k - 1 = O(n)$$ (proof on next slide)

This **averages out at O(1) per operation**, i.e., the amortised cost over the sequence of operations is constant.

Proof: Sum of Powers of 2

This has been used in the last analysis and the telescopic sum.

Statement: $\Sigma_{i=0...k-1}\ 2^i = 1 + 2 + 2^2 + ... + 2^{k-1} = 2^k - 1$

Proof by in induction:

base case k=2: $2^0 + 2^1 = 1 + 2 = 3 = 4 - 1 = 2^2 - 1$

step k to k+1: $2^0 + ... + 2^{k-1} + 2^k = 2^k - 1 + 2^k$

$= 2 \cdot 2^k - 1$

$= 2^{k+1} - 1$

Statement applied here

Array-like operations in extensible arrays

Some O(1) time operations in extensible arrays:

```
public int size() {
    return count;
}
public int elementAt(int pos) {
    return data[pos];
}
public void setElementAt(int pos, int value) {
    data[pos] = value;
}
```

Summary of extensible arrays

- Extensible arrays are an implementation of the Stack ADT that use O(n) space
- With the right extension policy, these operations take O(1) time
- They also provide array-like operations at O(1) time
- Implementing the Queue ADT with extensible arrays is not so convenient (why?)
- For greater convenience, we can use pointers

Extensible arrays in Java

- Java has data structures for extensible arrays:
  Vectors and ArrayLists
- Vectors and ArrayLists are like arrays that can be extended as necessary
  - you don’t have to worry about their size
  - you don’t have to extend them explicitly
  - Vectors have Objects as elements

ArrayLists in Java

Import them:

```
import java.util.List; // ArrayList and Vector are Lists
import java.util.ArrayList;
```

Create an ArrayList:

```
List l1 = new ArrayList();
```

Programming against an Interface is more flexible.

Adding at index i moves following elements to the right:

```
al.add(i,p)
```

(Array)List and Vector use Generics since Java 5.0 (JDK 1.5):

```
List<String> stringList = new ArrayList<String>();
```

Vectors in Java

To use them you have to import them:

```
import java.util.Vector;
```

To create a vector v1:

```
Vector v1 = new Vector();
```

The size of vector v1 is:

```
v1.size()
```

To test whether v1 is empty:

```
v1.isEmpty()
```

Other vector & arrayList methods in Java

To add an element--object *p*--at the end of vector v1 (arrayList a1):

```
v1.addElement(p); al.add(p);
```

To access the element at position *n* in vector v1 (arrayList a1):

```
e_n = v1.elementAt(n); e_n = al.get(n);
```

To assign element *p* to position *n* in vector v1:

```
v1.set(n, p); \\ note that this returns the
\\ object previously in that position
```

Reading

- Weiss: Chapters 15, 17 and 20
- Drozdek: Sections 4.2, 6.1 and 9.3.2

Next week: Linked lists

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