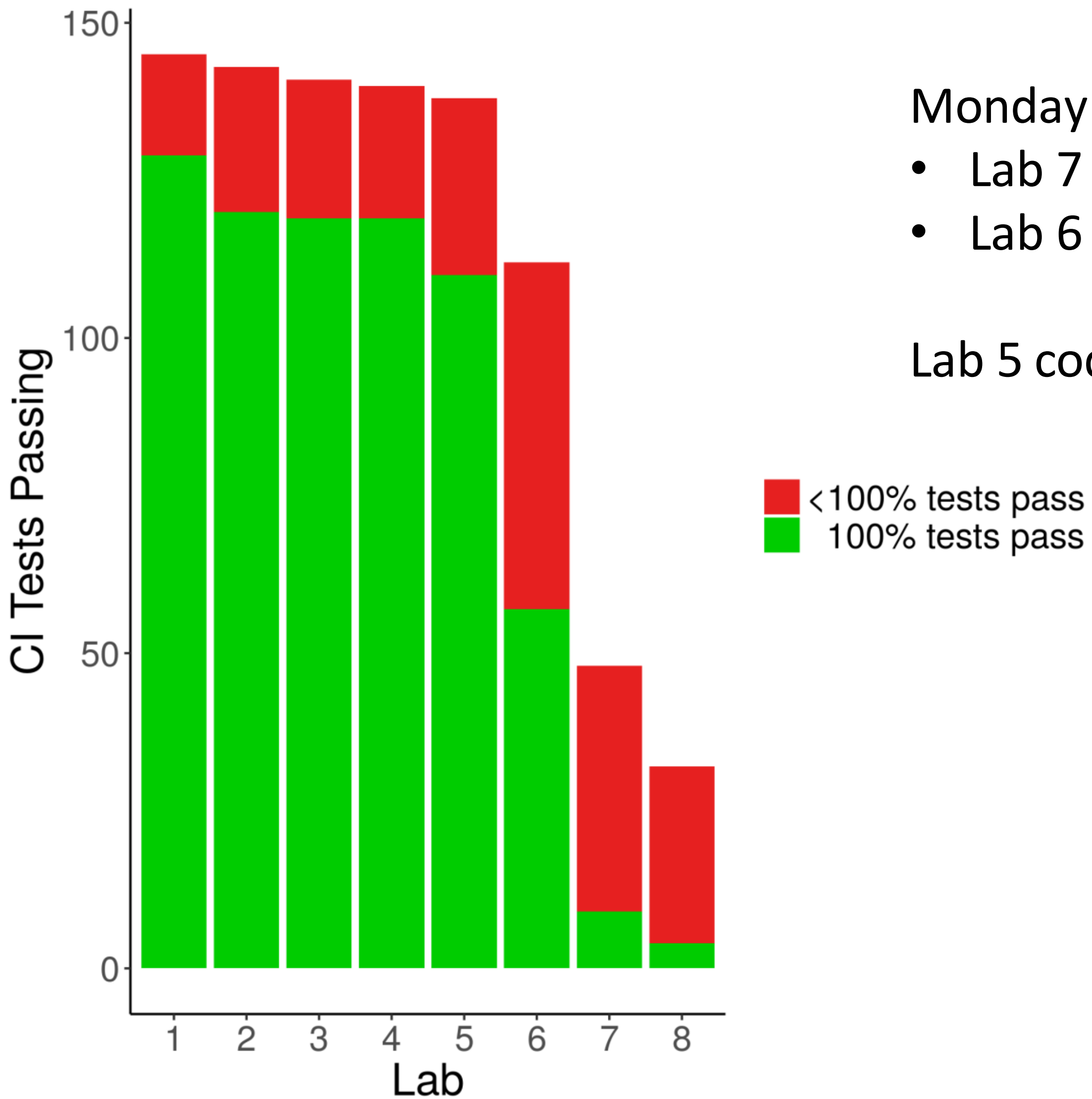


Friday 1st March, group 1:

- Lab 7
- Lab 6 deadline

Monday 4th, group 2:

- Lab 7
- Lab 6 deadline

Lab 5 code reviews on Monday/Tuesday

Software Development 3 (F27SG)

Lecture 14

Priority Queues

Rob Stewart

Overview

- Covered a number of Dynamic Data Structures
  - Linked-Lists
  - Trees
- These are Abstract Data Types
  - Different ways to implement them
    - arrays
    - dynamic creation and arrangement of objects
- Today we return to **Queues**
- **Priority queue**
  - Implemented with **array** based **min heap**

FIFO Queues Recap

- We have already covered Queues
- Supermarket, airports, printers

FIFO Queue Operations

- **Key property**: elements enter a queue at the rear and are removed from the front
- **Operations**
  - **enqueue(object)** - adds object to rear of the queue
  - **dequeue()** - remove and returns element at the top
  - **front()** - return the front element
  - **size()** - return number of elements
  - **isEmpty()** - check if empty
- Error conditions: dequeue/front of empty queue

Priority Queues

Priority Queues

- Each element in Queue has an associated "priority"
  - Usually a key (e.g. integer)
- Like FIFO (queue) and LIFO (stack) structures
  - elements **added in any order**.
- Unlike FIFO and LIFO structures
  - **elements** are **returned** in a **priority order**
- Either
  - highest priority, or
  - lowest priority
- There is no exposed notion of position
  - (e.g. 1st, 2nd 3rd element)

Uses of Priority Queues

- Air Traffic Control
  - What is the "priority" value here?
    - Fuel Level, Airline, Flight hours remaining?
- Computer Process Scheduling
    - Processes are "time sliced"
    - How do we pick the next process to get processor time?
- Airline Denied Boarding
  - Who gets kicked off the plane?

Priority Queue ADT

- Each entry has a key and value
  - **Key**: could be price, miles, age, …
  - **Value**: The thing that we store in the queue
- Operations
  - **size()** - how many entries
  - **isEmpty()** - is it empty or not?
  - **insert(key,value)** - Insert a key/value pair
  - **removeMin()** - Removes and returns minimum element
  - **min()** - Returns the minimum element (but keeps it)

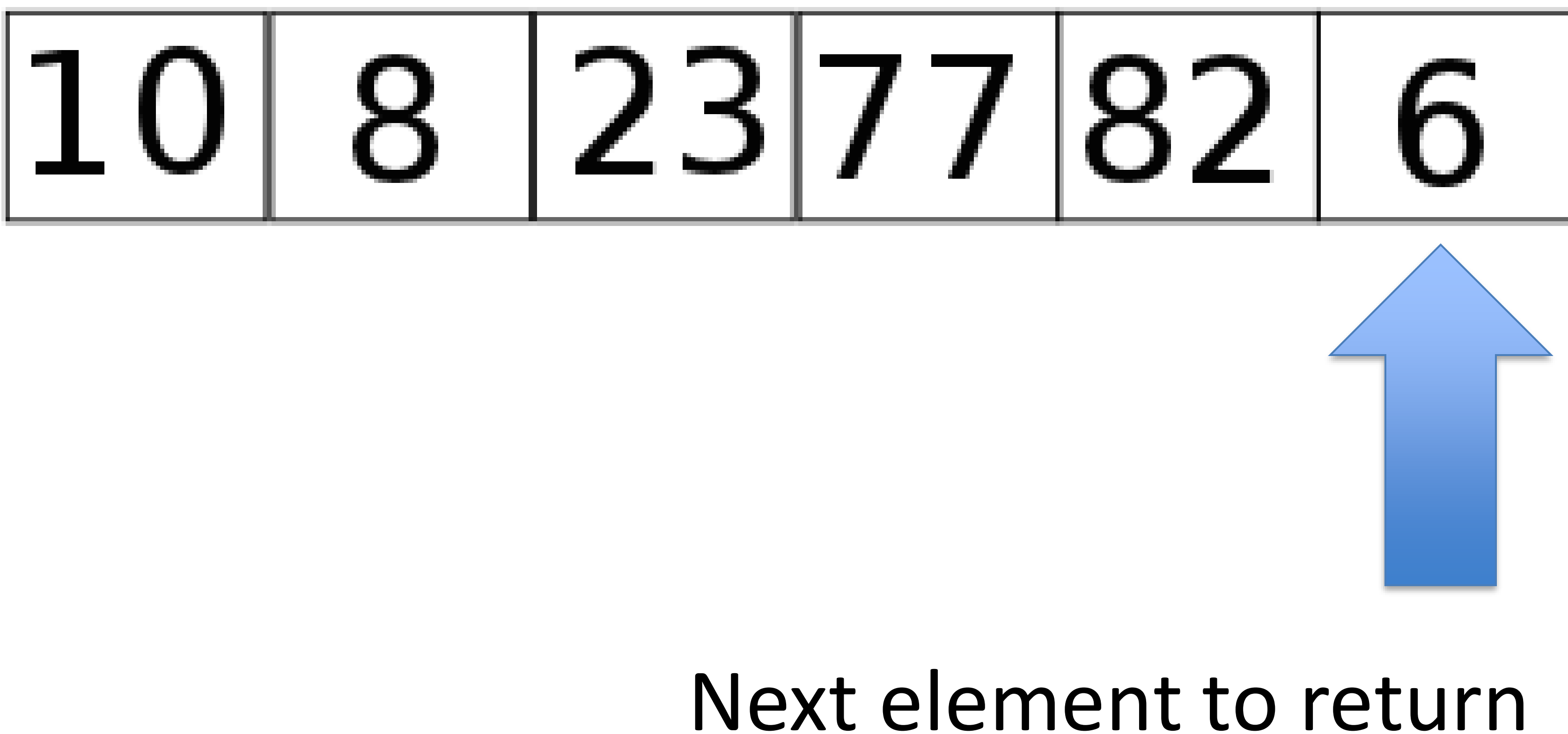
Priority Queue ADT (simplified)

- We will make a simplification
- We are storing numbers in the priority queue
  - Meaning key = value
- Operations
  - size()
  - isEmpty()
  - **insert(value)**
  - removeMin()
  - min()

Implementing a Priority Queue

- Fairly simple - use a linked list
- Add each element to the head
- Linear Search
  - to find the highest priority element
- But! also **naïve**:
  - O(1) insert
  - O(n) removeMin()

Naïve Priority Queue



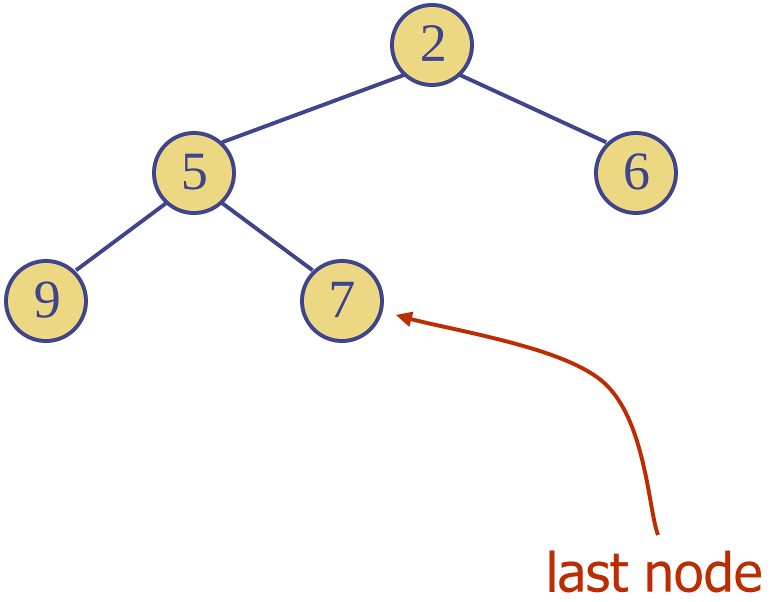
Implementing a Priority Queue (2)

- How about we keep the list sorted?
  - removeMin() now becomes O(1)
  - … but insert is O(n)
- So
  - Unsorted list: **O(1) insert**, O(n) removeMin
  - Sorted list: O(n) insert, **O(1) removeMin**
- Can we get the best of both worlds?
  - Use the **heap** data structure

Solution: Min Heaps

Min Heaps

- A **min 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 is filled up before moving to next
  - From left to right
- The last node is the rightmost node at the last level



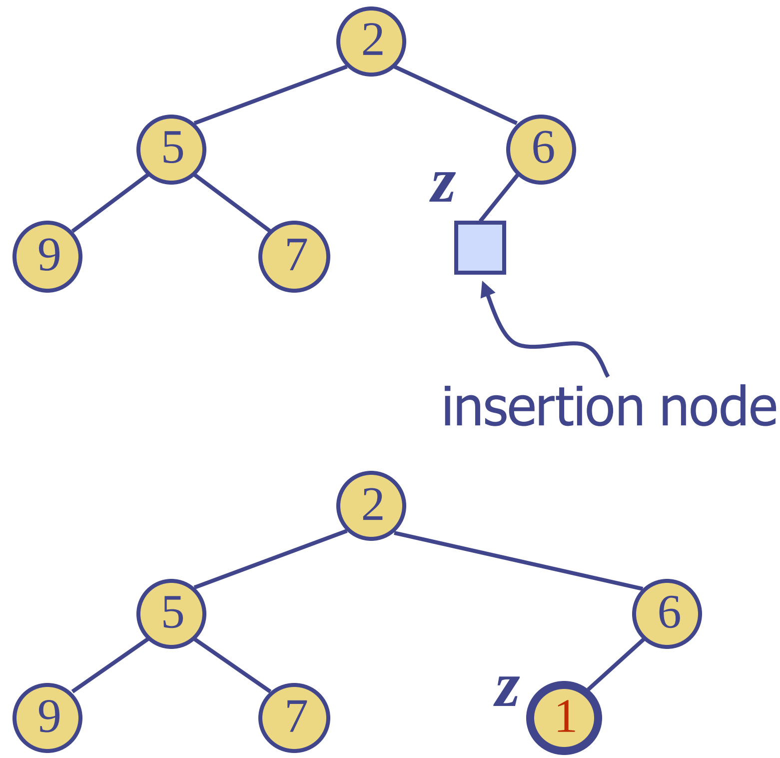
Heap operation

- The main operations on a priority queue are
  - removeMin
  - insert
- We need the heap to support these

Heap insert

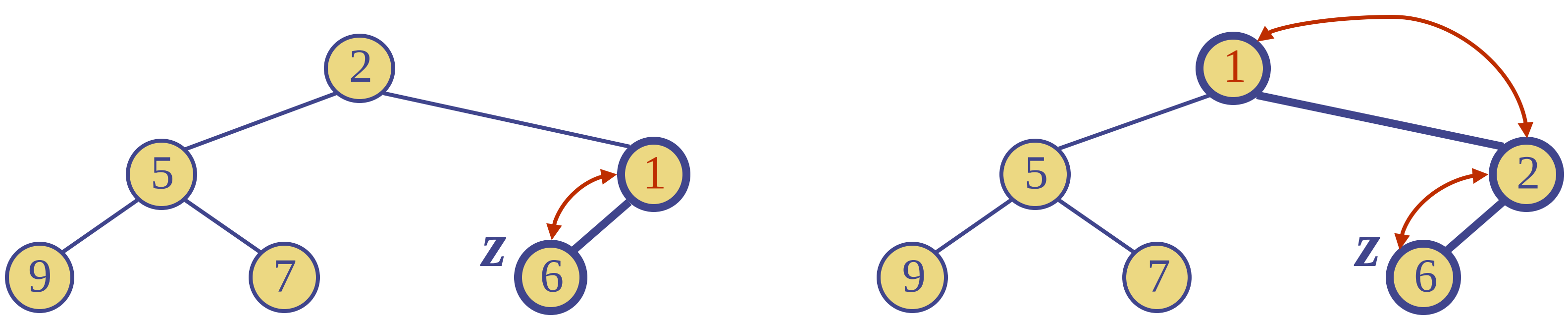
3 steps

1. Find the insertion node
   – The next free node
2. Insert node
3. Restore heap property



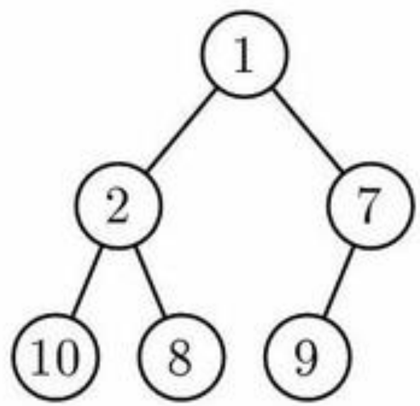
Heap insert restore heap property

- If node is smaller than parent
  - Swap value
- Continue until
  - Node is larger than parent, or node is at root
- This is called **upheap**



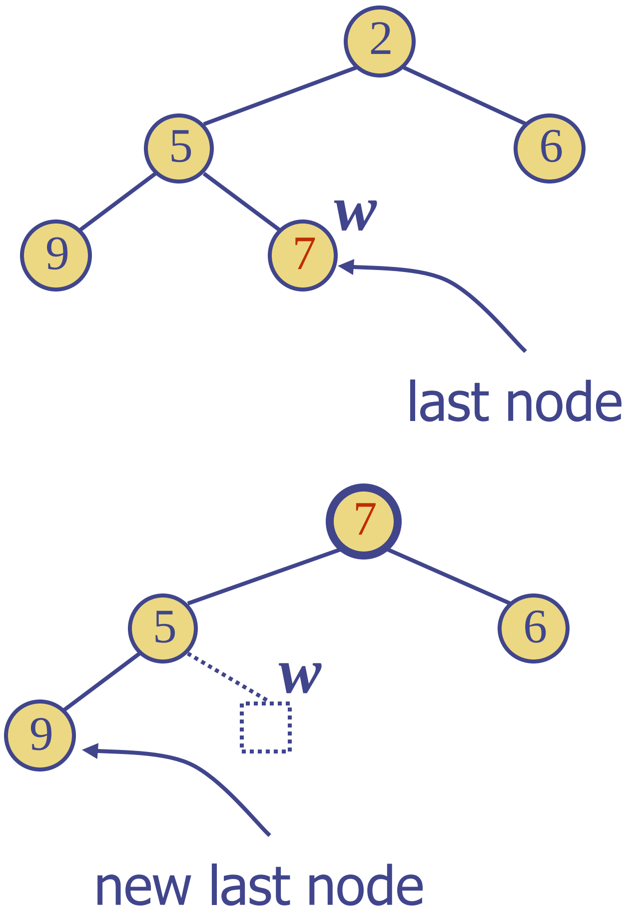
Exercise

- Apply **insert(5)** to the following heap:



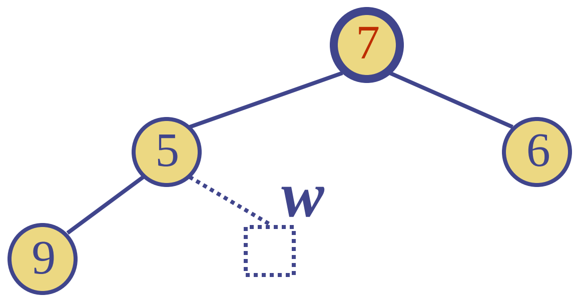
Heap removal (removeMin)

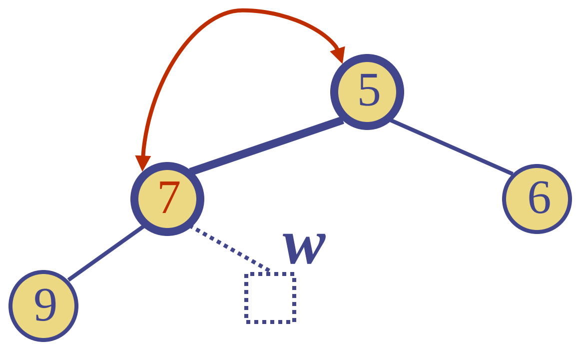
- Removal has 3 steps
  1. Replace (and return) root with last node (w)
  2. Remove last node
  3. Restore heap property



Heap removal
restore heap property

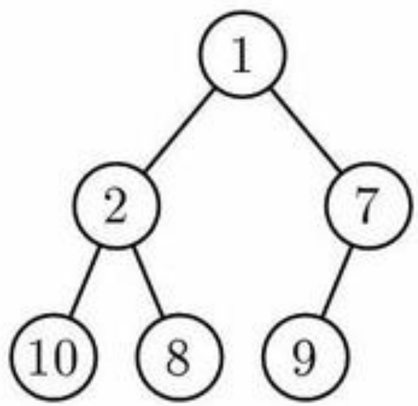
- Start at root and find the smallest child
  - If smaller than swap
- Continue until
  - Node is smaller than smallest child
  - or node is at leaf
- This is called **downheap**





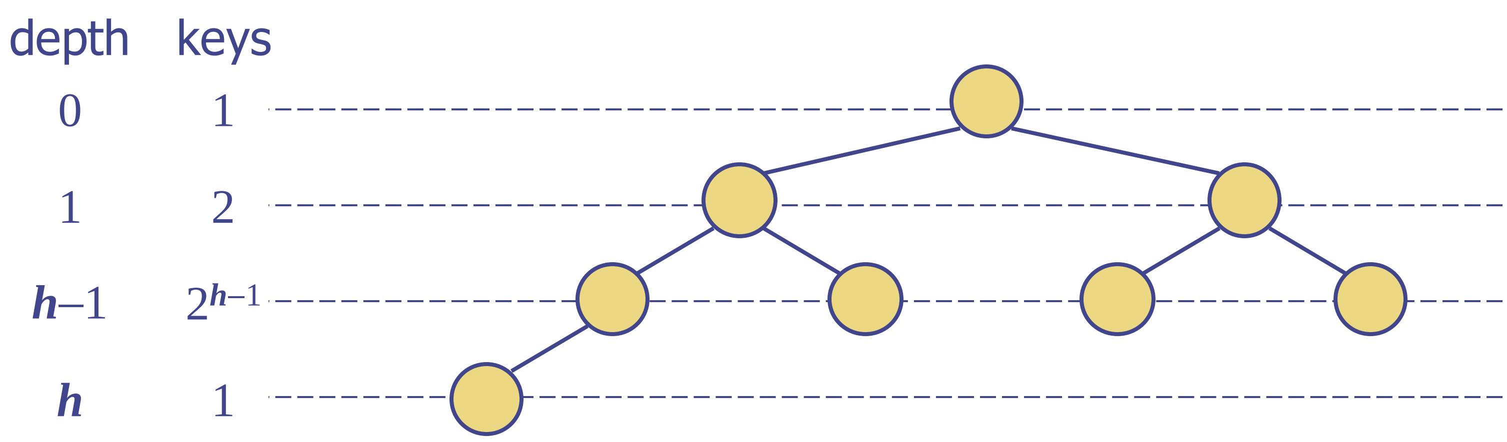
Exercise

- Apply **removeMin**() to the following heap:



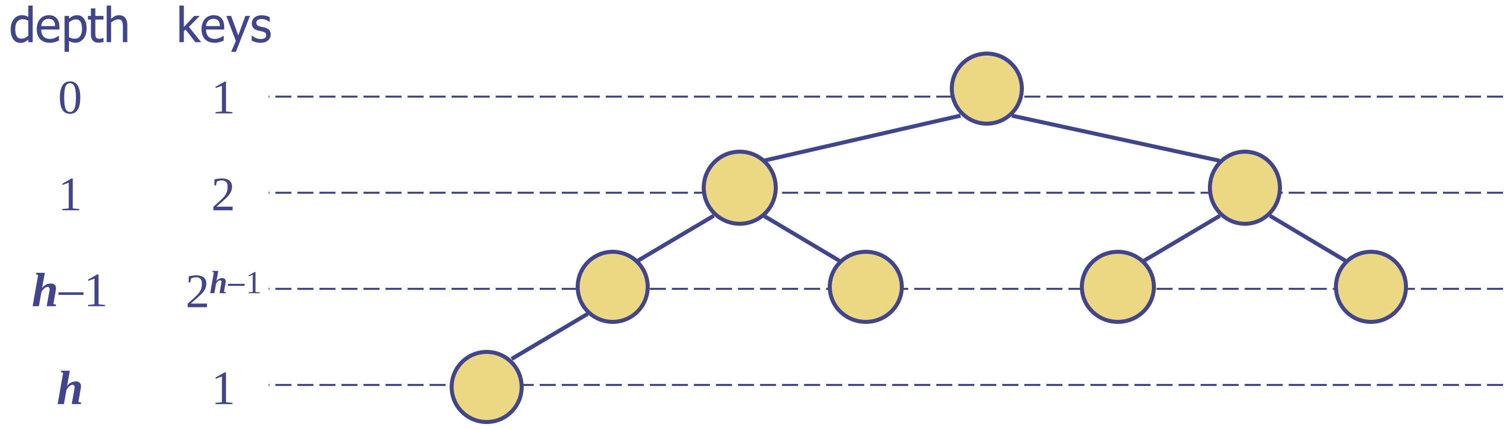
Heap complexity

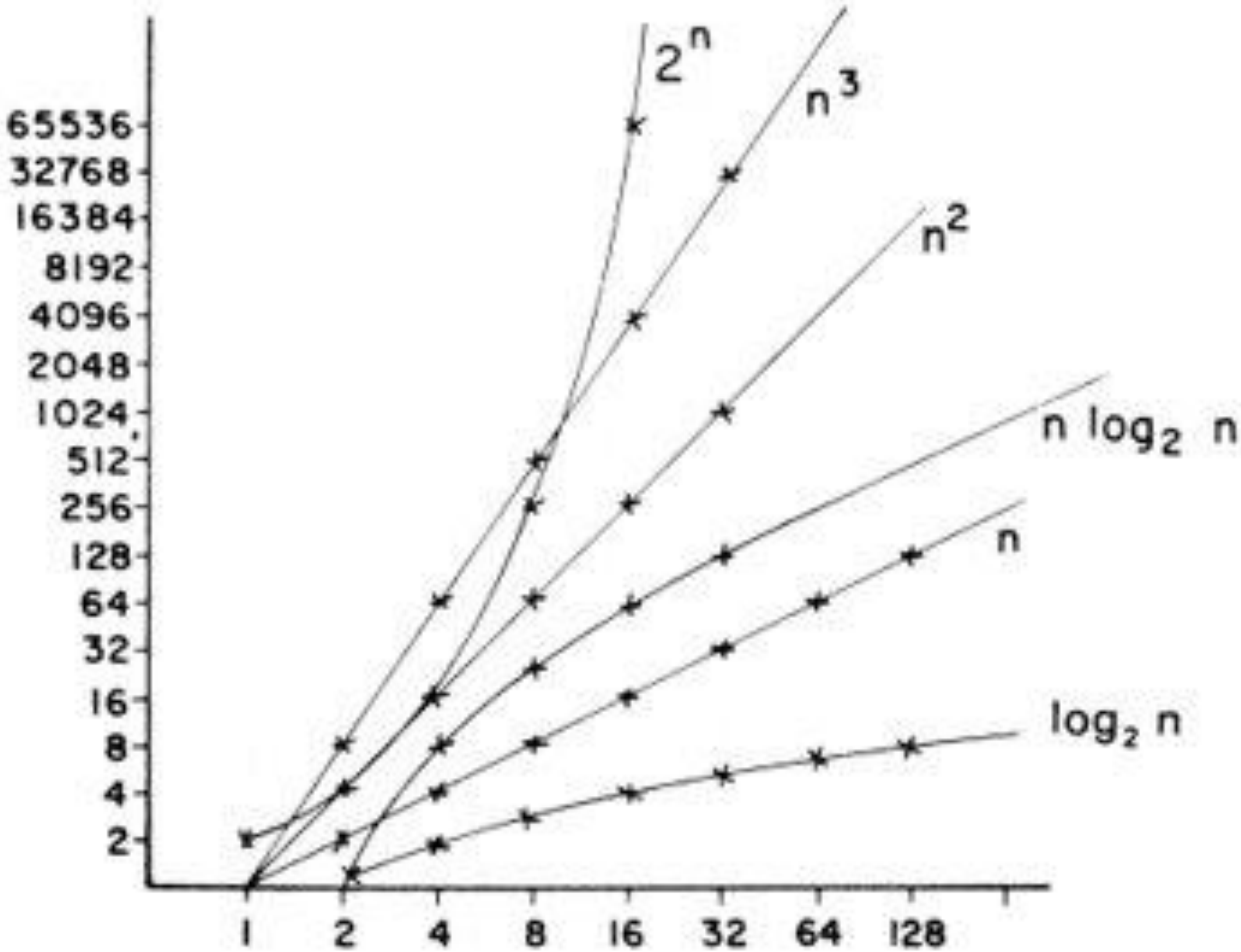
- Our goal was to improve on the **O(n)** complexity for insert/removeMin
- Tree is balanced, so the height is **log n**



Heap complexity

- Since the height is **log n**
  - *upheap* and *downheap* both runs in O(log N)
- Thus, **insert** and **removeMin**
  - are both **O(log N)**





65536
32768
16384
8192
4096
2048
1024
512
256
128
64
32
16
8
4
2
1
2
4
8
16
32
64
128
2^n
n^3
n^2
$n \log_2 n$
n
$\log_2 n$

Other Priority Queue operations

- min()
  - Involves returning the root of the heap: O(1)
- size()
  - we can keep a separate variable with the size: O(1)
    - as we did with e.g. stacks as linked lists

- isEmpty()
  - size == 0 : O(1)

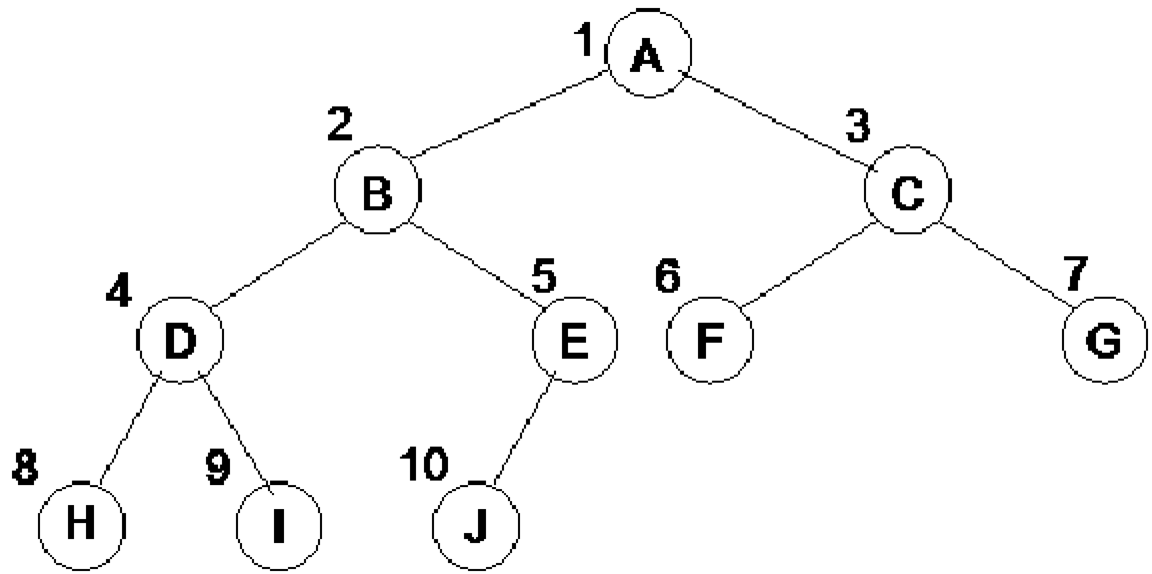
Implementing Heaps

- For min heaps, a binary tree fills up from
  - Top-to-bottom
  - Left to right

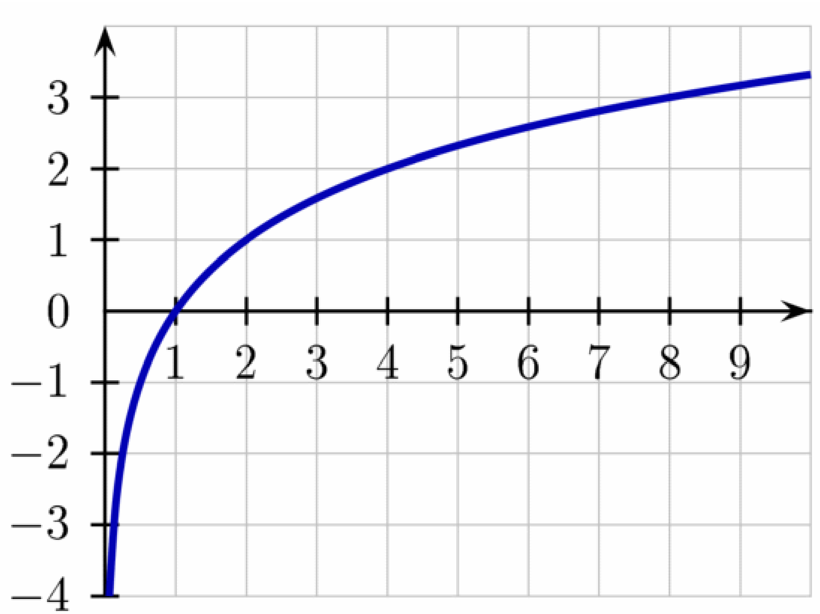
- This can in fact be represented as an **array**
  - Root at index 1

Implementing Heaps

- For a node at index n
  - It's left child will be at index 2*n
  - It's right child will be at index (2*n) + 1



	A	B	C	D	E	F	G	H	I	J				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14



- Log 2 = 1
- Log 4 = 2
- Log 8 = 3
- Log 16 = 4
- Log 32 = 5
- Log 64 = 6
- Log 128 = 7
- Log 256 = 8
- …

For example...

If you want **6 levels** in min heap, **array size** must be **64**

Min heap example

Eclipse Demo:

implementing a priority queue

Implementing Heaps

```
public class PriorityQueue {
  private int[] heap;
   int last;

public PriorityQueue(int max){
  heap = new int[max+1];
  last = 0;
}

 public int size(){
   return last;
}
```

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

public int min(){
  if(isEmpty())
       throw new
            PriorityQueueException
           ("Empty Queue");
  return heap[1];
}
```

Implementing Heaps – insertion

```
public void insert(int j){
  if(last == heap.length-1)
     throw new
        PriorityQueueException
              ("Full Queue");
  heap[++last] = j;
  upHeap(); }

private void swap(int i,int j)
{  int tmp = heap[i];
  heap[i] = heap[j];
  heap[j] = tmp; }
```

```
private void upHeap(){
  int index = last;
  int parent;
  while(index > 1){
     parent = index/2;
     if(heap[index] >= heap[parent])
       break;
     swap(index,parent);
     index = parent;
   }
}
```

Implementing Heaps – removeMin

```
public void removeMin(){
 if(isEmpty())
      throw new  PriorityQueueException (“Empty Queue");
 int min = heap[1];
  heap[1] = heap[last--];
  downHeap();
  return min;
}
```

Summary

- Introduced the **priority queue** ADT
- Shown issue with using linear linked list
- Introduced the **heap** data structure
  - insert/removeMin are O(log N)
- Shown how to **implement** priority queues using heaps with array storage

- Next lecture we will cover the final data structure in this course: **Tries**