COMP 3760: Algorithm Analysis and Design

Lesson 10: Priority Queues

Rob Neilson

rneilson@bcit.ca

Announcement

Note: I have deviated from the posted schedule a bit. I said that if I was going to do this I would announce it in class, so, I am announcing it now.

A new schedule if posted on webct.

Homework and Reading

Reading: Chapters 6.1, 6.4

Homework: (due at start of lab in week of Oct 13 – 17, except, <set G will submit on webct before Tuesday>)

Questions:

Chapter 6.1, pg 201, question 1

Chapter 6.4, pg 222, questions 1, 2, 6, 7

Note:

- Marks will be assigned to people who actually attempt the homework;
- this means that if a questions asks (for example) … what is the maximum number of keys … I would expect not just a number (ie: 42), but also some justification, calculation, or rational for the answer.
- Similarly, if a questions asks … is this a stable … I would expect more than ‘yes’ or ‘no’. You should explain your choice of answer.

Sample Exam Question

Given 2 sets S_1 and S_2 (each of size n), and a number x, describe an *O(nlogn)* algorithm for finding whether there exists a pair of numbers, one from S_1 and one from S_2, that add up to x.

we observe:

- brute force would be $O(n^2)$ because we would have to compare each element from S_1 with each element from S_2
- O(nlogn) suggest a sort. How could a sort help us?
  - if we were searching in a sorted array (or list), the search would be O(logn)

therefore we could do something like ...

```
put S2 in an array A[]                      // O(n)
sort A[]                                    // O(nlogn)
for each item s in S1                       // n times
    t = x – s                               // t is num to find
    if (binarySearch(A[], t) ≥ 0)           // O(logn)
        return PAIR_EXISTS
end for
return DOES_NOT_EXIST
```

O(n)+O(nlogn)+n*O(logn) =
O(n)+O(nlogn)+O(nlogn) $\in$ O(nlogn)

Today's Agenda

- Priority Queues
- Heaps

What is a Priority Queue?

- A data structure with the following properties:
  - used to store items that have **ordered keys**
  - **quick insertion** and deletion of arbitrary items
  - **quick access** to the item with the **largest** or smallest key
  - *when would we use something like this?*

> ➢ need an ordered list, but, the order keeps changing as the algorithm runs
> eg: maybe it is a game, and the current 'leader' is displayed somewhere …
> but the leader always changes …

Sample Problem: computer simulation

- you need to model a real-world situation, for example you might want to know if twinning the Port Mann Bridge will actually ease traffic congestion
- as the simulation progresses, future events are identified, and queued to be processed
  - eg: a car arrives at the bridge; a car leaves the bridge, etc
- the simulation engine will do something like:
  - while there are events (eg: a car arrives)
    - analyze current situation
    - update statistics / state information
    - add/delete future events as the current situation requires
    - advance time to time of next event
    - get next event
- Solution:
  - use a priority queue to store all known future events. The highest priority event is the next one that will occur
  - in this case the ordered key (priority) is the event time

When to use a Priority Queue

- stacks/queues let us retrieve items based on ***insertion order***
- dictionaries (maps) let us retrieve items based on a ***key***
- **priority queues let us retrieve items based on a *priority***

Use a priority queue when you need quick access to the smallest or largest key (where the key indicates the relative priority of the items)

- Note: it ***is*** permissible for items to have duplicate keys. The order for items with the same key value is *not defined*.

Priority Queue Implementations (1)

Recap

- we want a structure that stores objects (items) and sorts them based on a ‘key’
- want the structure to have fast insert / retrieve of maximum (or minimum) items

Implementation 1: sorted array or list

largest item always at the end

- fast find and delete maximum (or minimum). How?
- slow insertion of new elements. Why?
- slow deletion of arbitrary elements. Why?

need to find posn, maybe need to shuffle elements

0	1	2	3	4	5	6	7
4	6	11	19	21	37	39	44

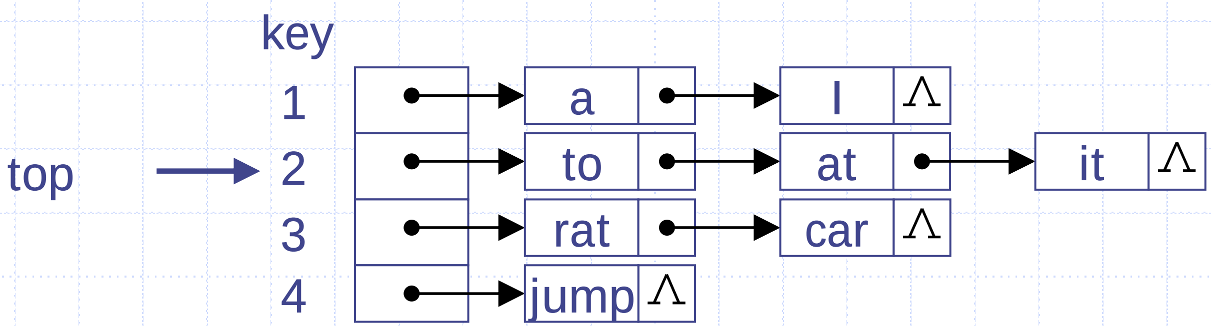
same as insert … need to locate item – and maybe shuffle elements

Priority Queue Implementations (2)

Implementation 2: bounded-height priority queue

This is a slight variation of a PQ … here we select on ‘max number of keys’

- requires a bounded range of key values
- implemented as an array of n linked lists
- i^{th} bucket is a list of all the items with key i
- maintain a pointer *top* to *smallest non-empty list*
- the next element retrieved is the first one in the list pointed to by top



- what is the key in the above pq?

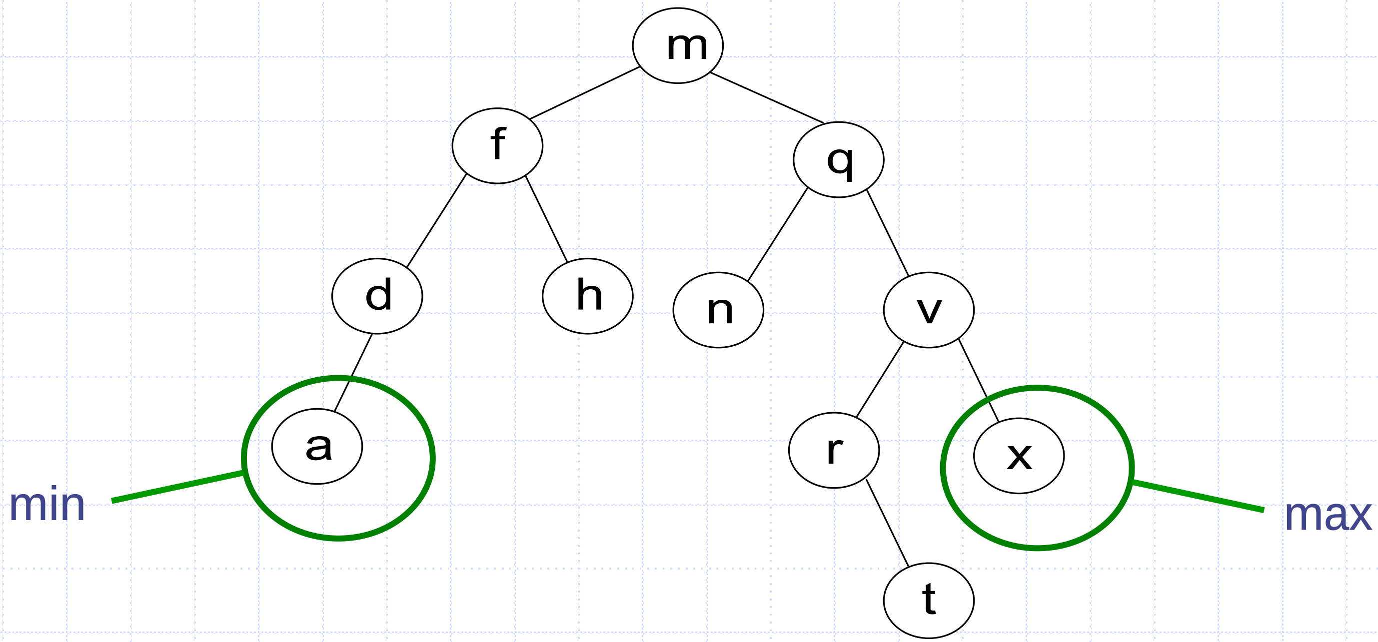
the key for top has value “2”

- use when you need to process the queue with most entries

Priority Queue Implementations (3)

Implementation 3: binary search tree

- smallest element is leftmost node
- largest is rightmost node



- need to traverse tree to get max item (worst case O(n) when tree degenerates to a list)
- we could use a balanced binary tree … *but there is a better data structure …*

Priority Queue Implementations (4)

Implementation 4: binary heap

- insert in O(log n)
- extract max value is also O(log n)

- most PQ's are implemented as Heap's, because of the fast insert and remove times

... ??but what exactly is a binary heap?? ...

OK, so what is a Heap?

A Heap is a binary tree where:

a) All leaves are on, *at most*, two adjacent levels

b) All leaves on the lowest level occur to the left

c) All levels *except* the lowest are completely filled

d) The key in the root is $\geq$ all its children

e) The left and right sub-trees of any node are also heaps (recursive defn)

Note:

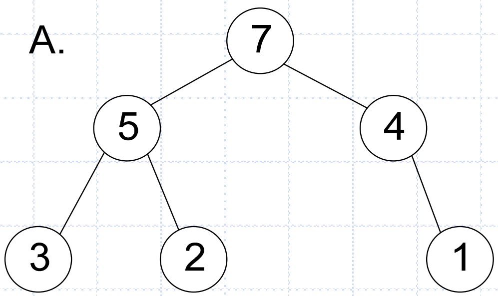
heaps are not binary search trees, but they are binary trees.

write these rules down, as you will need them for the next slide!

Heap or No Heap?

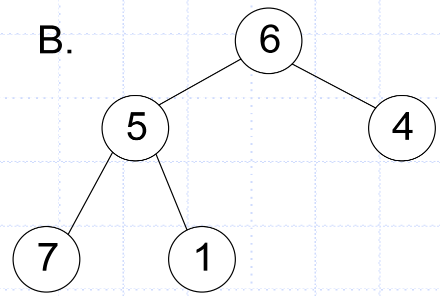
NO

A.



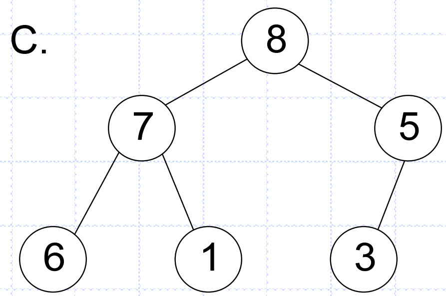
NO

B.



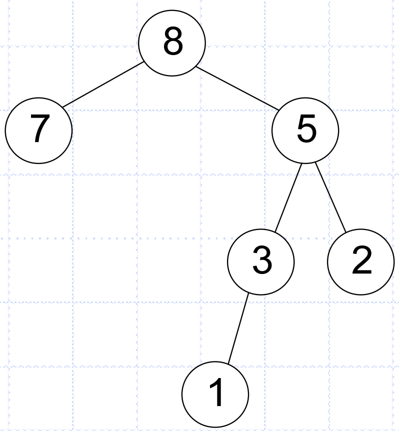
YES

C.



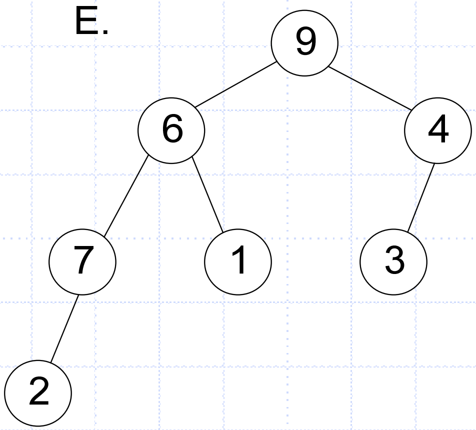
NO

D.



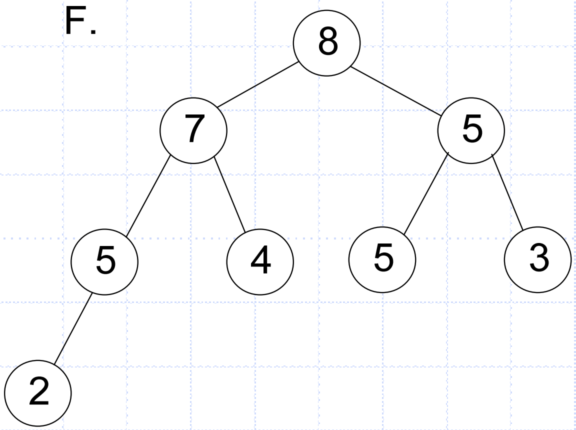
NO

E.



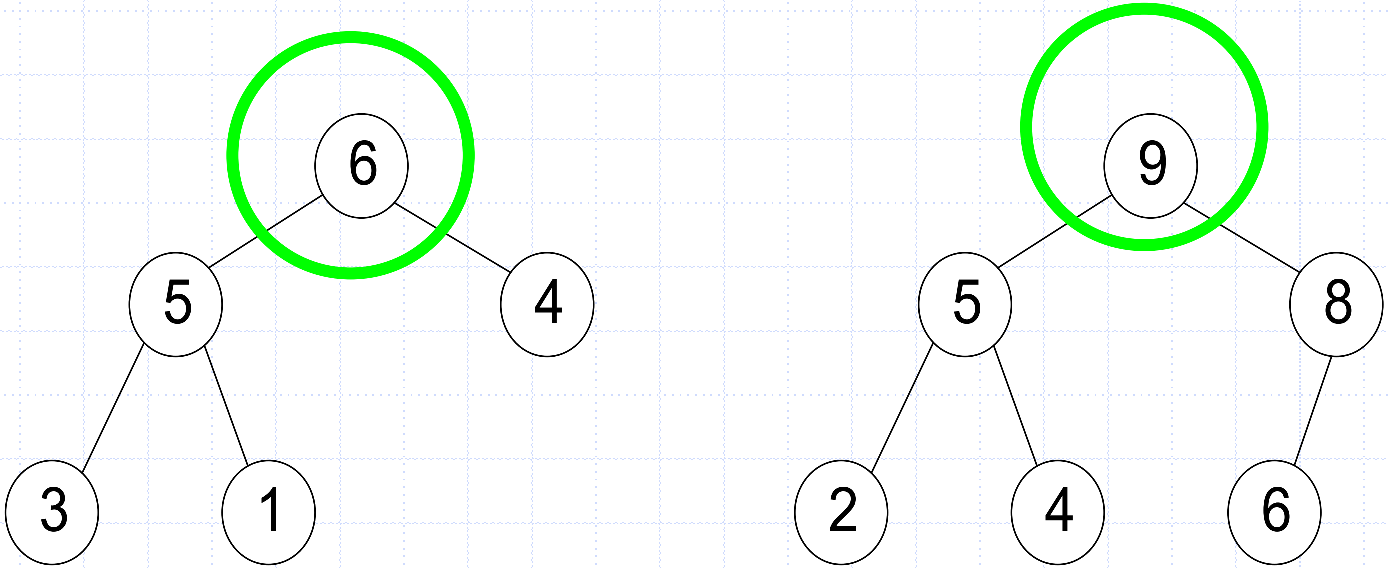
YES

F.



Heap Question 1

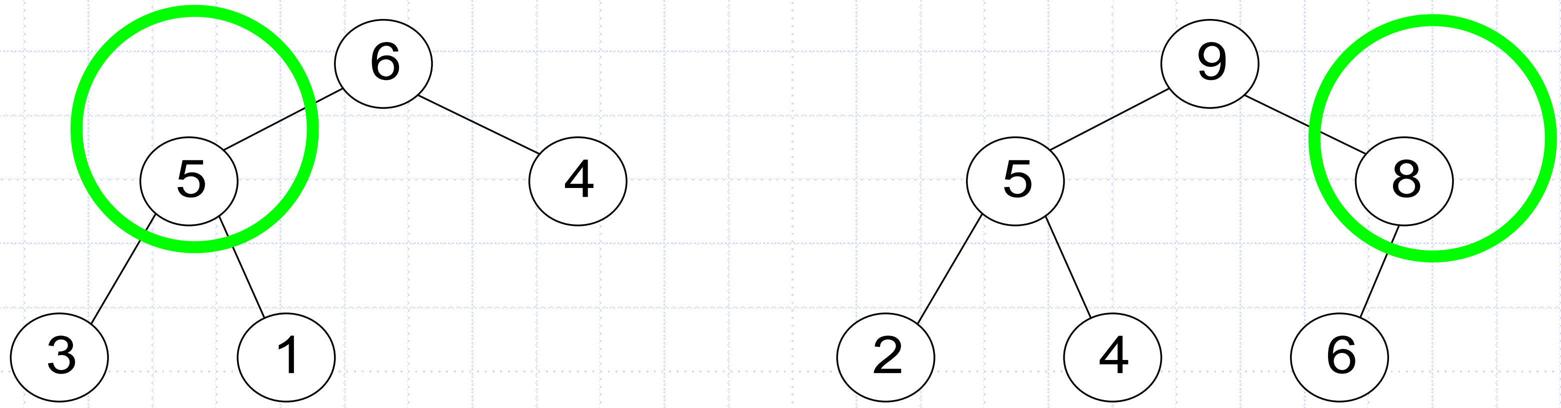
- Where is the largest element in a heap?
  Answer - the root.



Heap Question 2

- Where is the second largest element?

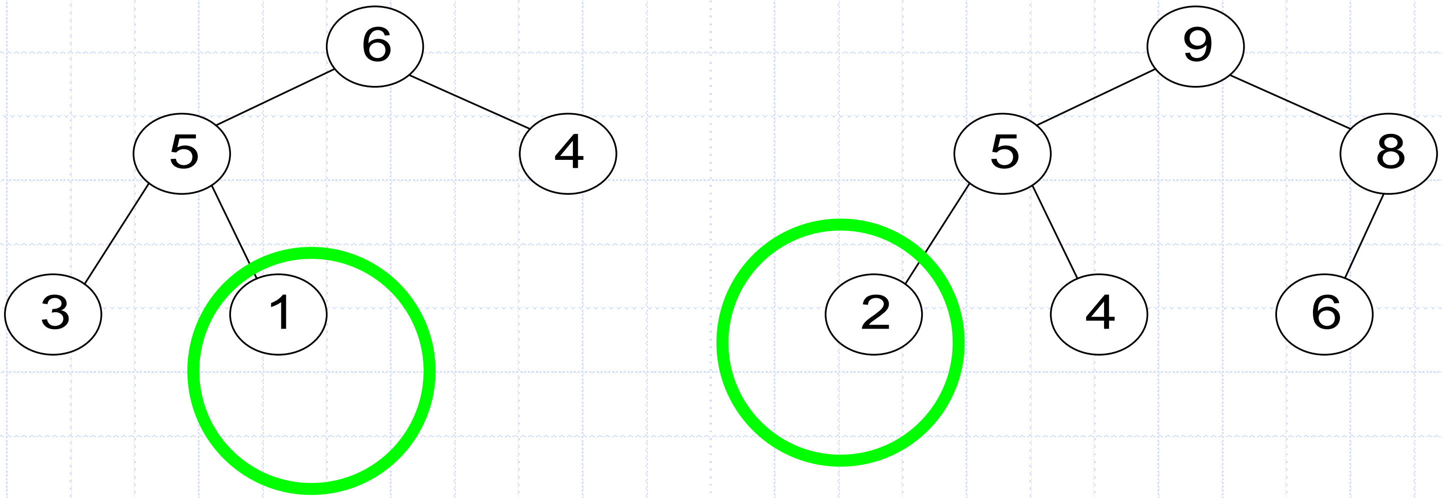
  Answer - as the root's left *or* right child.



Heap Question 3

- Where is the smallest element?

  Answer - it is *one* of the leaves (but we don't know which one).



Heap Question 4

- Can we do a binary search to find a particular key in a heap?

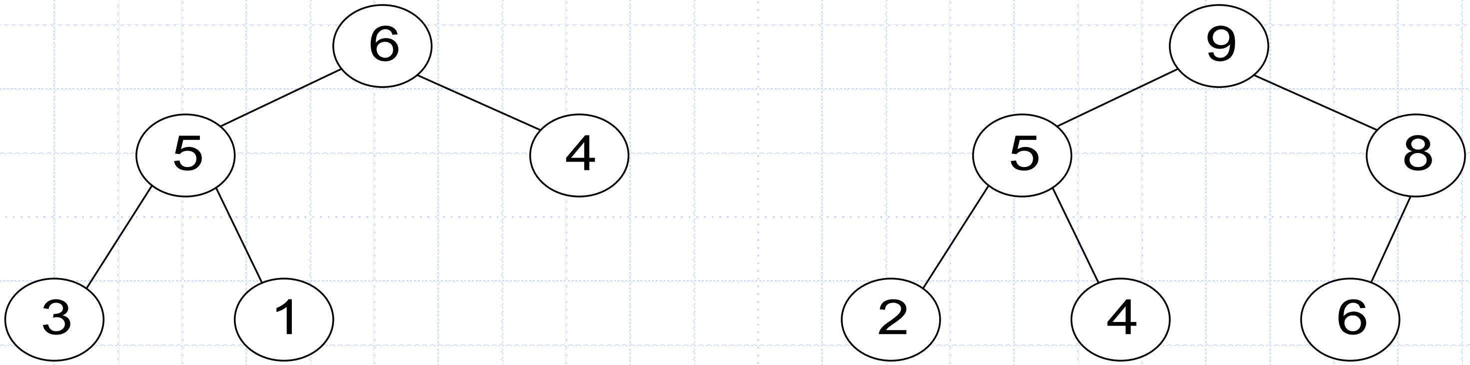
Answer - No!

A heap is not a binary search tree, and cannot be effectively used for searching.

Heap Question 5

- Why Do Heaps “Lean Left”?

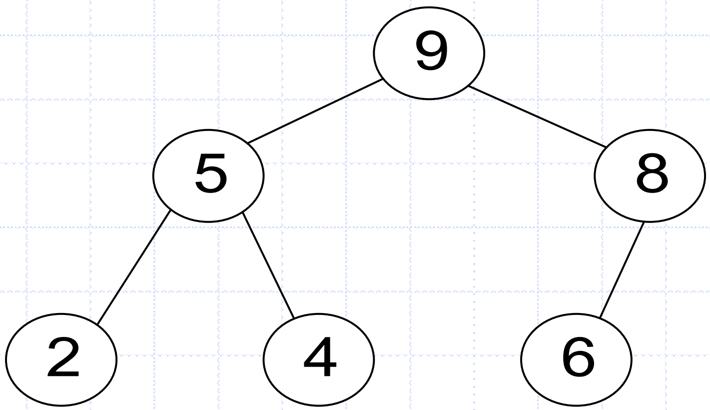
Answer – The “fill from left” rule guarantees that most heaps will lean left.



Heap Question 6

- How can we store the heap in an n element array without pointers?

  Answer - each of the *n* items can be assigned a number from 1 to *n* with the property that the *left* child of node number *k* has a number 2*k* and the right child number 2*k*+1. These numbers are used to index the array.



1	2	3	4	5	6
9	5	8	2	4	6

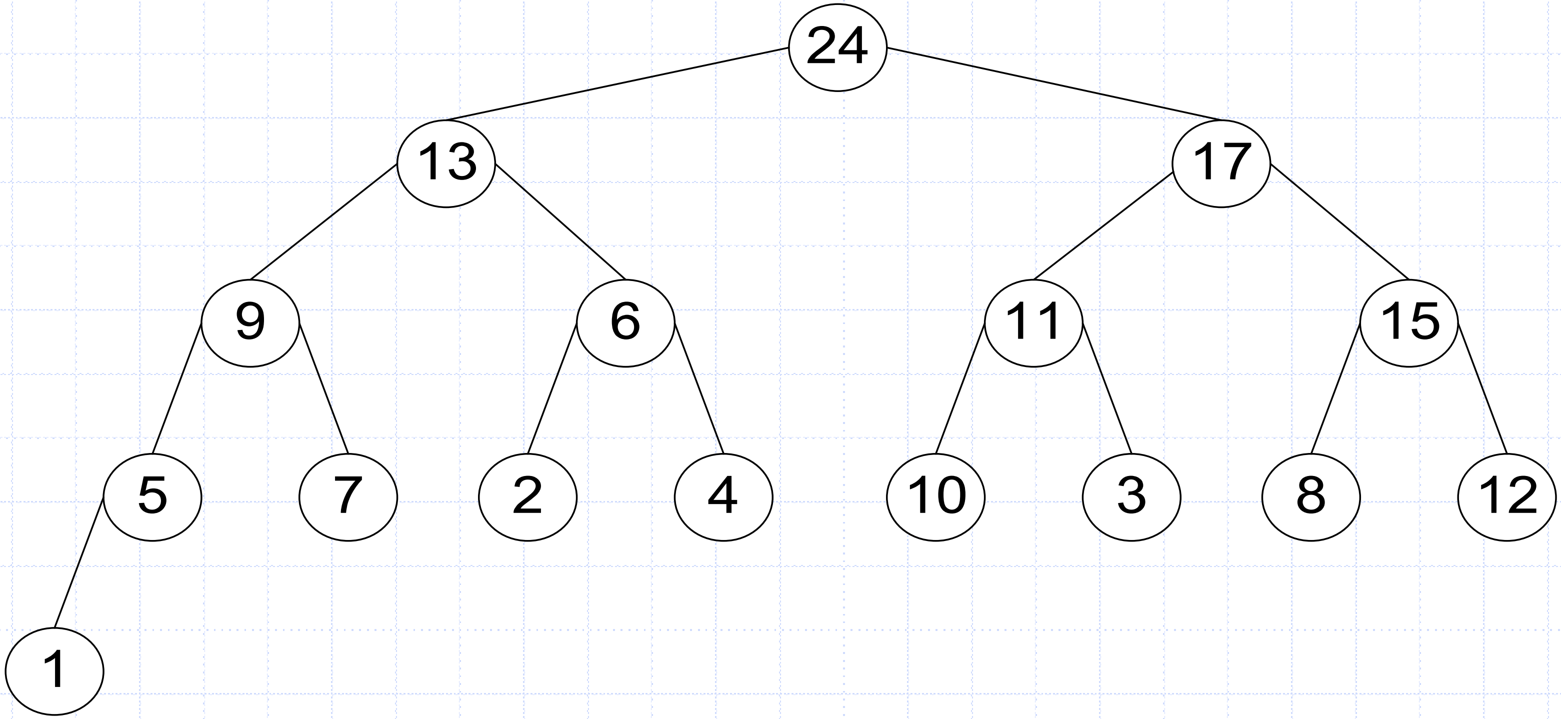
Array Representation (1)

- draw the tree representation of this heap

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
value	17	11	12	9	8	10	5	1	4	6	2	3	7

Array Representation (2)

- draw the array representation of this heap



Algorithm Heap.Insert(item)

- use this algorithm to insert a new key into an existing Heap
- can be used to create a new heap in O(nlogn) time (by putting it in a loop)

```
Insert(H[1...n], item)
      insert item into last empty location
      while item > parent AND item is not root
            swap item with parent
```

Example:

- build a heap by inserting the following numbers one at a time with the Insert method shown above

  9 8 13 12 14 2 20 18 17 14 16 24 3 6

Which Priority Queue implementation to use? (1)

Ask yourself some questions:

What other operations do I need (besides access to the maximum item)?

- do you need to search for arbitrary elements?
  - in this case a sorted array might work (binary heap would be slow)
- do you need the smallest (as well as largest)?
  - a sorted list or binary search tree would be good for this
- will you be deleting arbitrary elements?
  - if there are a lot of arbitrary deletions, consider using a sorted list
- do you know the maximum size of the structure in advance?
  - if you know this, you can pre-allocate storage – so array based structures work well

Which Priority Queue implementation to use? (2)

More questions:

- will you be changing the priority of elements already in the queue or just inserting them?
  - changing priority means you need to lookup arbitrary items by key, and then restructure the heap
    - we would use something called a *Fibonacci heap*
- is your priority based on the value of the key, or on the number of items with identical keys
  - this would imply that you need a bounded-height priority queue
- are new items inserted after the first query?
  - if not, just use a list (no need for a priority queue)

Sample Exam Question

a. Devise an algorithm that deletes an arbitrary item from a Priority Queue that is implemented as a Binary Heap.

b. Devise an algorithm that deletes an arbitrary item from a Priority Queue that is implemented as a Sorted List.

c. What is the efficiency class for the algorithms from (a) and (b)?

The End