HEAPS & PRIORITY QUEUES

Lecture 16
CS2110 Spring 2018

Announcements

2

- A4 due TOMORROW. Late deadline is Sunday.
- A5 released. Due next Thursday.
- Deadline for Prelim 1 regrade requests is tomorrow.
- Remember to complete your TA evaluations by tonight.

Abstract vs concrete data structures

3

- Abstract data structures are **interfaces**
  - they specify only **interface** (method names and specs)
  - not **implementation** (method bodies, fields, …)
- Concrete data structures are **classes**. Abstract data structures can have multiple possible implementations by different concrete data structures.

Concrete data structures

4

- Array
- LinkedList (singley-linked, doubly-linked)
- Trees (binary, general, red-black)

Abstract data structures

5

- **interface** List defines an "abstract data type".
- It has methods: add, get, remove, …
- Various **classes** ("concrete data types") implement List:

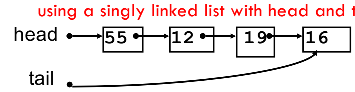
Class:	ArrayList	LinkedList
Backing storage:	array	chained nodes
add(i, val)	O(n)	O(n)
add(0, val)	O(n)	O(1)
add(n, val)	O(1)	O(1)
get(i)	O(1)	O(n)
get(0)	O(1)	O(1)
get(n)	O(1)	O(1)

Abstract data structures

6

- List (ArrayList, LinkedList)
- Set (HashSet, TreeSet)
- Map (HashMap, TreeMap)
- Stack
- Queue

Both stack and queue efficiently implementable using a singly linked list with head and tail



- PriorityQueue

Priority Queue

7

- Data structure in which data items are `Comparable`
- Elements have a priority order (smaller elements---determined by `compareTo()`---have higher priority)
- `remove()` return the element with the highest priority
- break ties arbitrarily

Many uses of priority queues

8

Surface simplification [Garland and Heckbert 1997]

- Event-driven simulation: customers in a line
- Collision detection: "next time of contact" for colliding bodies
- Graph searching: Dijkstra's algorithm, Prim's algorithm
- AI Path Planning: A* search
- Statistics: maintain largest M values in a sequence
- Operating systems: load balancing, interrupt handling
- Discrete optimization: bin packing, scheduling
- College: prioritizing assignments for multiple classes.

`java.util.PriorityQueue<E>`

9

```
interface PriorityQueue<E> {
 boolean add(E e) {...} //insert e.
 E poll() {...} //remove/return min elem.
 E peek() {...} //return min elem.
 void clear() {...} //remove all elems.
 boolean contains(E e)
 boolean remove(E e)
 int size() {...}
 Iterator<E> iterator()
}
```

Priority queues as lists

10

- Maintain as a list
  - `add()` put new element at front – O(1)
  - **poll()** must search the list – O(n)
  - **peek()** must search the list – O(n)
- Maintain as an ordered list
  - `add()` must search the list – O(n)
  - **poll()** min element at front – O(1)
  - **peek()** O(1)
- Maintain as red-black tree
  - `add()` must search the tree & rebalance – O(log n)
  - **poll()** must search the tree & rebalance – O(log n)
  - **peek()** O(log n)

Can we do better?

Heaps

11

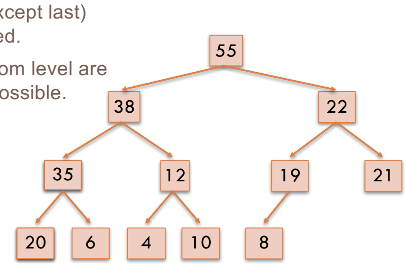
- A ***heap*** is a binary tree that satisfies two properties
  1) Completeness. Every level of the tree (except last) is completely filled.
  2) Heap Order Invariant. Every element in the tree is <= its parent

Do not confuse with heap memory, where a process dynamically allocates space–different usage of the word heap.

Completeness Property

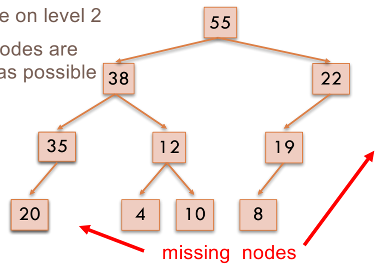
Every level (except last) completely filled.

Nodes on bottom level are as far left as possible.



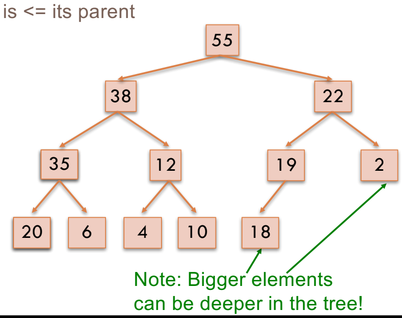
Completeness Property

Not a heap because:
- missing a node on level 2
- bottom level nodes are not as far left as possible



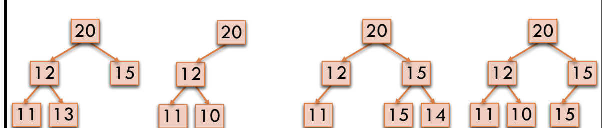
Order Property

Every element is <= its parent



Heap Quiz

Which of the following are valid heaps?



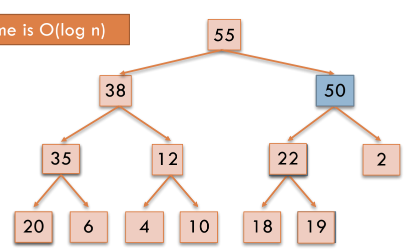
(a)	(b)	(c)	(d)
No	No	No	Yes

Heaps

- A ***heap*** is a binary tree that satisfies two properties
  1) Completeness. Every level of the tree (except last) is completely filled. All holes in last level are all the way to the right.
  2) Heap Order Invariant. Every element in the tree is <= its parent
- A heap implements three key methods:
  1) add(e): adds a new element to the heap
  2) poll(): deletes the max element and returns it
  3) peek(): returns the max element

add(e)

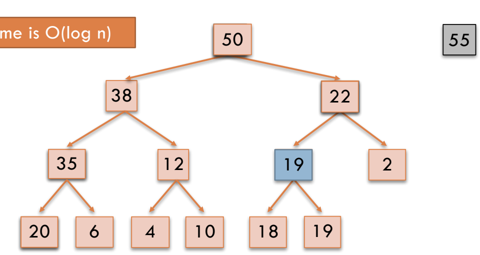
Time is O(log n)



1. Put in the new element in a new node (leftmost empty leaf)
2. Bubble new element up while greater than parent

poll()

Time is O(log n)

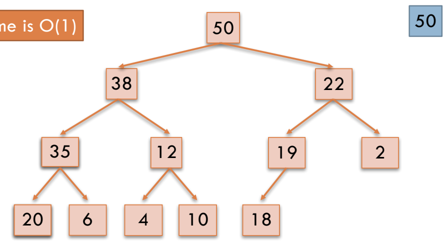


1. Save root element in a local variable
2. Assign last value to root, delete last node.
3. While less than a child, switch with bigger child (bubble down)

peek()

19

Time is O(1)

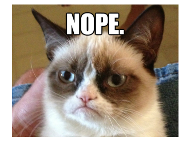


1. Return root value

Implementing Heaps

20

```
public class HeapNode<E> {
  private E value;
  private HeapNode left;
  private HeapNode right;
  ...
}
```



Implementing Heaps

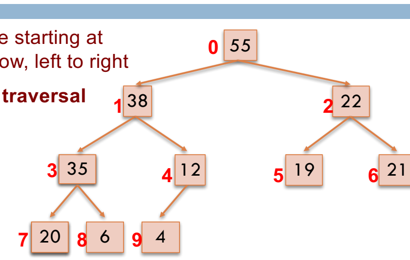
21

```
public class Heap<E> {
  private E[] heap;
  ...
}
```

Numbering the nodes

Number node starting at root row by row, left to right

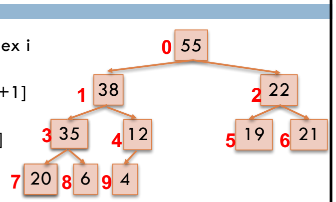
Level-order traversal

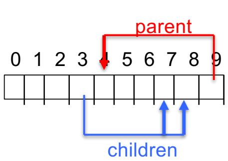


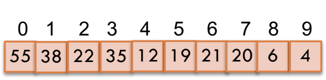
Children of node k are nodes 2k+1 and 2k+2
Parent of node k is node (k-1)/2

Storing a heap in an array

- Store node number i in index i of array b
- Children of b[k] are b[2k +1] and b[2k +2]
- Parent of b[k] is b[(k-1)/2]







add() --assuming there is space

24

```
/** An instance of a heap */
class Heap<E> {
  E[] b= new E[50];  // heap is b[0..n-1]
  int n= 0;          // heap invariant is true

  /** Add e to the heap */
  public void add(E e) {
    b[n]= e;
    n= n + 1;
    bubbleUp(n - 1); // given on next slide
  }
}
```

add(). Remember, heap is in b[0..n-1]

25

```
class Heap<E> {
  /** Bubble element #k up to its position.
    * Pre: heap inv holds except maybe for k */
  private void bubbleUp(int k) {
    int p= (k-1)/2;
    // inv: p is parent of k and every elmnt
    // except perhaps k is <= its parent
    while (k > 0  &&  b[k].compareTo(b[p]) > 0) {
       swap(b[k], b[p]);
       k= p;
       p= (k-1)/2;
    }
}
```

poll(). Remember, heap is in b[0..n-1]

26

```
/** Remove and return the largest element
  * (return null if list is empty) */
public E poll() {
    if (n == 0) return null;
    E v=  b[0];    // largest value at root.
    b[0]= b[n];    // element to root
    n= n – 1;      // move last
    bubbleDown(0);
    return v;
}
```

poll()

27

```
/** Tree has n node.
 *  Return index of bigger child of node k
    (2k+2 if k >= n) */
public int biggerChild(int k, int n) {
   int c= 2*k + 2;     // k's right child
   if (c >= n || b[c-1] > b[c])
      c= c-1;
   return c;
}
```

poll()

28

```
/** Bubble root down to its heap position.
    Pre: b[0..n-1] is a heap except maybe b[0] */
private void bubbleDown() {
   int k= 0;
   int c= biggerChild(k, n);
   // inv: b[0..n-1] is a heap except maybe b[k] AND
   //      b[c] is b[k]'s biggest child
   while ( c < n &&  b[k] < b[c] )
      swap(b[k], b[c]);
      k= c;
      c= biggerChild(k, n);
   }
}
```

peek(). Remember, heap is in b[0..n-1]

29

```
/** Return the largest element
  * (return null if list is empty) */
public E peek() {
    if (n == 0) return null;
    return b[0];   // largest value at root.
}
```

Quiz 2: Let's try it!

30

Here's a heap, stored in an array:

[9 5 2 1 2 2]

What is the state of the array after execution of add(6)? Assume the existing array is large enough to store the additional element.

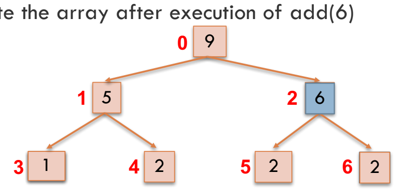
A. [9 5 2 1 2 2 6]
B. [9 5 6 1 2 2 2]
C. [9 6 5 1 2 2 2]
D. [9 6 5 2 1 2 2]

Quiz 2: Let's try it!

Here's a heap, stored in an array:

[9 5 2 1 2 2] ⇒ [9 5 6 1 2 2 2]

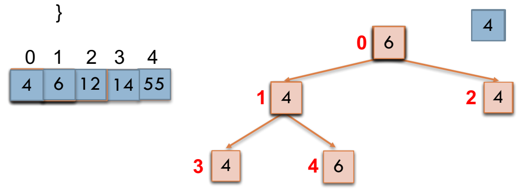
Write the array after execution of add(6)



HeapSort

32

1. Make b[0..n-1] into a max-heap (in place)
2. for (k= n-1; k > 0; k= k-1) {
   b[k]= poll –i.e. take max element out of heap.
   }



Priority queues as heaps

33

- A *heap* is can be used to implement priority queues
  - Note: need a min-heap instead of a max-heap
- Gives better complexity than either ordered or unordered list implementation:
  - **add():** O(log n) (n is the size of the heap)
  - **poll():** O(log n)
  - **peek():** O(1)

java.util.PriorityQueue<E>

34

```
interface PriorityQueue<E> {                          TIME
 boolean add(E e) {...} //insert e.                  log
 void clear() {...} //remove all elems.
 E peek() {...} //return min elem.                    constant
 E poll() {...} //remove/return min elem.             log
 boolean contains(E e)                                linear
 boolean remove(E e)                                  linear
 int size() {...}                                     constant
 Iterator<E> iterator()
}                                          IF implemented with a heap!
```

What if the priority is independent from the value?

35

Separate priority from value and do this:

add(e, p); //add element e with priority p (a double)

THIS IS EASY!

Be able to change priority

change(e, p); //change priority of e to p

THIS IS HARD!

Big question: How do we find e in the heap?
Searching heap takes time proportional to its size! No good!
Once found, change priority and bubble up or down. OKAY

Assignment A5: implement this heap! Use a second data structure to make change-priority expected log n time