CS211, Lecture 22
Hierarchical Sequence Structure

Announcements:

Overview:

- Motivation
- PQ
- Linear PQ
- Heaps
- PQ as Heap
- Array representation

1. Motivation

1.1 Sequence Structure Reminder

- want ADT to store and retrieve items
- use put and get to create and manipulate a pile of things
- not interested in search

1.2 Queues

- line things up, take them out in FIFO order
- weakness: what if some things are more important than others?
- Analogies: patients needing emergency care, small and large print jobs, office hours…

1.3 Priority Queues

- insert items in any order:
  include a ***priority*** (numerical rating of importance)
- extract items according to priority

1.4 For later…

- some algorithms need to be broken into smaller tasks
- use PQs to prioritize which tasks occur early/later

2. Interface

```
interface SeqStructure {
    void put(Object o);
    Object get();
    boolean isEmpty();
    int size();
}
```

2.1 Operations

- `put`: insert items in any order…
- `get`: remove item with highest priority
  - might see altenate ops: `findMin` or `findMax`
  - return smallest/largest item means return highest/lowest priority ("small" things usually go first)

2.2 Implementations

- linear: as array, as list
- hierarchical: as BST (better, splay tree), heap (special kind of tree)

3. PQ Elements

- PQ Element has data (`Object`) and priority (`int`)
- kind of like a key-value pair
- design elements to be compared for priorities
- code:

```
public class PQElement implements Comparable {
    private Object item;
    private int priority;
    public PQElement(Object o, int p) {
        item = o;
        priority = p;
    }
    public int getPriority() { return priority; }
    public void setPriority(int p) { priority = p; }
    public Object getItem() { return item; }
    public String toString() {
         return  "("+item+","+priority+")"; }

    // return pos means this.p > o.p
    // return nil means this.p = o.p
    // return neg means this.p < o.p
    public int compareTo(Object o) {
        if (o instanceof PQElement)
            return (priority - ((PQElement)o).priority);
        else {
            System.out.println("Crap!");
            return 0; // should really throw an Exception
        }
    }
} // Class PQElement
```

4. Array Implementation

4.1 Very similar to `SortedArray` for searching

- but, must allow for duplicates
- sort in ascending order
- item with highest priority from end of array
- so,
  - get: O(1)
  - put: O(n)
- see **`PQAsSortedArray.java`**

4.2 Get

```
// get the rightmost item,
// because sorted in ascending order
// (want highest priority item):
public Object get() {
    // check for empty PQ:
    if (size == 0) {
        System.out.println("Empty array error!");
        return null;
    }
    return a[--size];
}
```


4.3 Put

- need to set cursor (current item) with binary search
- allow for repeats

```
// insert into right place in sorted array
// (ascending order):
public void put(Object o) {
    if (size == MAXSIZE) {
        System.out.println("Overflow error!");
        return;
    }
    // set the cursor to point to insertion point:
    boolean found = search(o);

    // find leftmost object equal to o
    // (could have repeats):
    while (cursor > 0 &&
        ((PQElement)(a[cursor - 1])).compareTo(o) == 0)
        cursor--;

    // make room for new element by shifting
    // elements to left of cursor
    for (int i = size -1; i >= cursor;i--)
        a[i+1] = a[i];

    // insert new element:
        a[cursor] = (PQElement) o;
        size++;
}
```


5. List Implementation

5.1 Operations

- put: either store “randomly” (FIFO) or sort list:
  - FIFO makes **`put`** easy, but **`get`** hard
  - sorted list makes put work a lot to insert and set cursors (same kind of cursors as array PQ) but **`get`** is very easy
- **`get`**: see above

5.2 Implementations

- **`PQAsList`**: easy put, hard get
- **`PQAsListAlt`**: hard put, easy get
- either way, at least one operation is O(n)

5.3 Example from PQAsList

```
public class PQAsList implements SeqStructure{
    // fields

    public void put(Object o) {
        list.add(o); size++; }

    public Object get() {
        if (isEmpty()) {
            System.out.print("Empty!");
            return null;
        }

    // search list, starting from head:
    ListNode n = list.getHead();
    Object max = n.getItem();
    ListNode next = n.getNext();

    // find and update max:
    while (next != null) {
        Object current = next.getItem();
        int comp = ((Comparable)current).compareTo(max);
        if (comp > 0) max = current;
        next = next.getNext();
    }

    list.remove(max);
    size--;
    return max;

    // methods
}
```

6. Tree Implementation

6.1 Any other way to do this?

- array & list have O(*n*) at some point
- so maybe a search tree could help us?

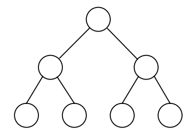
6.2 Search Trees

- designed for searching (BST gives O(log *n*) on avg
  - search is proportional to height (search path from root to leaf)
  - nodes = 2^height-1, so height = log[2](nodes+1)
  - if BST is "bushy", gives O(log *n*) time
  - if BST is "skinny", resembles list: O(*n*) time
  - actually, there's a lot more math involved here…
- problems:
  - input not randomized (queues used for simulation)
  - support more operations than really needed
  - need to have "bushiness" of tree to get O(log n)
- something called a ***splay tree*** helps sometimes
- ***heap*** to the rescue!

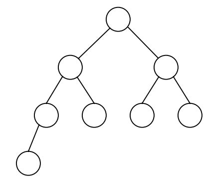
7. Binary Trees Revisited

7.1 Special types of binary tree to define heap

- we need some tree definitions
- ***full binary tree***: every node has two children
- ***complete binary tree***: full binary tree, except…
  - next-to-last level may be partially filled
  - must fill last level from left to right
- full, complete help give "bushiness" to trees



full



complete

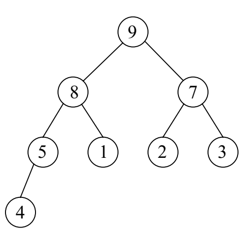
8. Heap Types

8.1 *heap*: complete binary tree with ordering

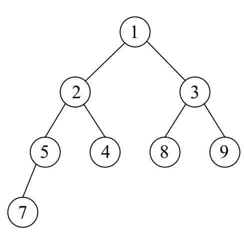
- different kinds of ordering (min, max, min-max, …)
- not related to "the heap"
- use `Comparable` to achieve ordering

8.2 Specific heap types

- ***maxheap***: object in a node ≥ all children
- ***minheap***: object in a node ≤ all children
- ***min-max heap*** (and more): http://www.diku.dk/forskning/performance-engineering/Jesper/heaplab/heapsurvey_html/node1.html



maxheap



minheap

8.3 Real Life Examples

- ages of people in family tree: parent is always older than children (max heap), but you may have an uncle/aunt younger than you
- people's salaries: bosses make more than subordinates ("pions"), but a 2nd-level manager may make more money than a 1st-level manager in a different sub-division

8.4 Min or Max for PQ?

- we'll pick maxheap (DS&SD 18.6)
- why? want to find max priority item
- minheap has analogous operations (see DS&A, Chap 11)

8.5 Methods to implement

- `put`: add something to the heap, but must preserve the heap property (min, max, min-max, …)
- `get`: remove largest item from the heap

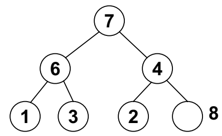
8.6 Heapsort

- If we're always pulling out the largest (or smallest) item, then technically, we could use a heap for sorting!
- DS&SD go into ***heapsort*** (section 18.6)

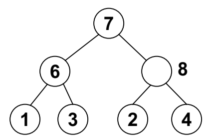
9. Heap Operations

9.1 put

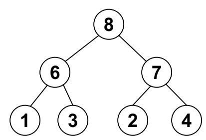
- method will insert item at first free node (left-bottom)
- must maintain heapness, so will have to reheapify



Get heap
Does it have enough space?
Find/create 1st empty location
See if item (8) goes there



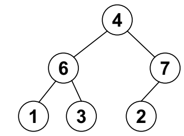
Parent (4) is < item (8)
so, move parent to that empty
Now, compare item with parent



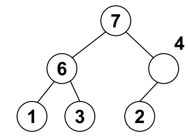
Parent (7) is < item (8)
so, move parent to that empty
Now, compare item with root
Nothing left to compare
Done.

9.2 get

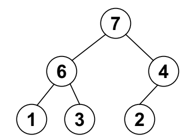
- want to remove max, which means root (uh oh!)
- tree could becomes forest! crap!
- but, we could "fix" (reheap) the heap



Get max (root, which was 8)
Replace with 1st leaf (4)
Delete (or nullify) that leaf
Want happy heap? Reheap!



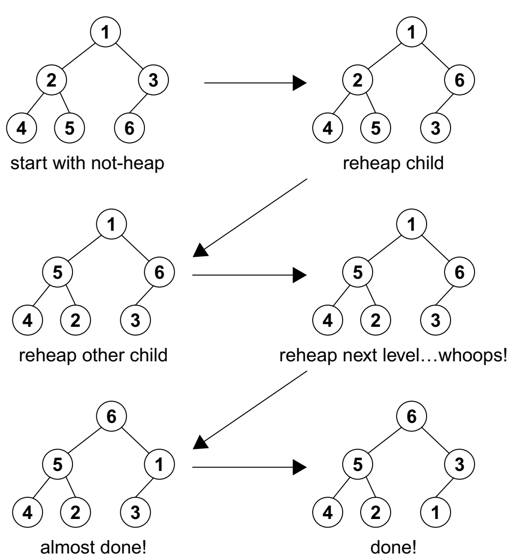
Item (4) is > children
Put item in current parent



Done!
Our heap is ready for more action.

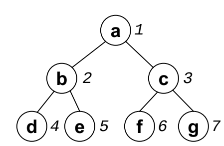
9.3 create heap

- could use **`put`**, which gives O($n \log n$)
- better: reheap each internal node until reaching root:



10. Heaps and Arrays!

10.1 Number Nodes in Heap



10.2 Now, put items in array with indicies

	a	b	c	d	e	f	g
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

10.3 Now, think BINARY TREE

- binary tree should somehow use powers of 2
- ex) size = 2^(h+1)-1 = 2^(2+1)-1 = 8-1 =7

10.4 Older files to check out:

- **`MaxHeap.java`**
- **`HeapDecoder.java`**
- **`TestMaxHeap.java`**

10.5 Interesting Features

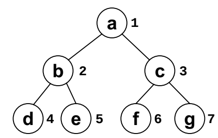
- children of node *i*: 2*i* and 2*i*+1
- parent of node *i* (not root): *i*/2
- root: *i*/2=0, so i = 1

10.6 Binary Path

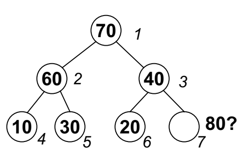
- find binary representation of a node number:
  - formula:

$$d = b_n \times 2^n + b_{n-1} \times 2^{n-1} + \ldots + b_1 \times 2^1 + b_0 \times 2^0$$

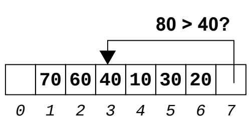
  - take number *d*, divide by 2
  - remainders are $b_0, b_1, \ldots, b_{n-1}, b_n$
  - ex) b0=6%2=0; b1=3%2=1; b2=1%2=0; so, 6→110
- drop leading 1 (if exists):
  - pattern: 1 = R, 0 = L
  - ex) 6 → 110, so to find node 6, go RK (10):



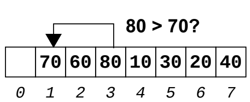
10.7 Example (put)



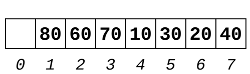
Want to put 80 into heap



Want to insert item (80)
Locate free space (index 7)
Find parent 7/2=3→40



80>40, so move 40 to index 7
Find parent: 3/2=1→70



80>70, so move 80 to index 1
Find parent: 1/2=0
So, we reached root…done!

11. Maxheap Code

From **`MaxheapAsArray.java`**

See also **`PQAsMaxheapAsArray.java`**

11.1 Fields, Constructors

```
private Object[] heap;
private int MAXSIZE;
private int size; // gives pointer to last index in heap

public MaxheapAsArray(int size) {
    MAXSIZE = size;
    heap = new Object[MAXSIZE];
}

public MaxheapAsArray(Object[] stuff) {
    // set # of elems and allotted space:
    MAXSIZE = size = stuff.length+1;

    // need to have empty 0th pos:
    heap = new Object[MAXSIZE];

    // copy stuff into unheaped heap:
    System.arraycopy(stuff,0,heap,1,MAXSIZE);

    // start at root and work "down"
    for (int i = heap.length/2; i > 0; i--) reheap(i);
}
```

11.2 Put

```
public void put(Object o) {

    size++; // increment size more this new item

    // increase array if out of space
    if (size > MAXSIZE) increaseHeapSize();

    int index = size; // index of current free location
    int parent = index/2; // parent of free location

    // Until reaching root, move the parents down
    // while item o > parents:
    while (index > 1 &&
        ((Comparable) o).compareTo(heap[parent]) > 0) {

        heap[index]=heap[parent]; // parent ->child
        index = parent;           // update index
        parent = index/2;         // update parent

    }

    // done finding appropriate location
    // to maintain heapness:
    heap[index] = o;
}
```


11.3 Get

```
public Object get() {
    Object root = null;
    if (!isEmpty()) {
        root = heap[1];      // max item to return
        heap[1]=heap[size]; // root<-recent item added
        size--;             // reduce size
        reheap(1);          // reheap entire tree
    }
    return root;
}
```

`reheap`? See **`MaxheapAsArray.java`**

11.4 PQ AS Maxheap (AsArray)

```
public class PQAsMaxheapAsArray {
    public static void main(String[] args) {
        SeqStructure pq = new MaxheapAsArray(10);
        pq.put(new PQElement("Bill",3));
        pq.put(new PQElement("Monica",1));
        pq.put(new PQElement("Hillary",4));
        pq.get());
        pq.put(new PQElement("Gennifer",3));
    }
}
```


12. Time analysis

12.1 Nodes and height

- $n = 2^h - 1$
- $h = \log(n+1)$
- so, $h = \log(n+1)$

12.2 put

- height is $\log(n+1)$
- adding does 1 level at a time, so O(log *n*)

12.3 get

- similar analysis as height
- O(log *n*)

13. Exercises

- Implement a PQ with a circular array.
- Implement a PQ with a sorted list. Allow for duplicate items.
- Write **`toTree`** for **`HeapAsArray`** that produces a text-based tree, as we did for you binary trees.
- Rewrite **`HeapAsArray`**'s **`put`** such that the unused array position (index 0) stores the item. Doing so helps to move the **`index > 1`** test in the **`while`** loop.
- Write a **`heapsort`** method inside **`HeapAsArray`**.