SORTING

PRIORITY QUEUES

Ruben Acuña

Fall 2018

2

CONCEPTUAL BACKGROUND

PRIORITY QUEUES: BACKGROUND

- Until now we been concerned with the idea of transforming an entire collection of elements into a *sorted* order.
- This is a very useful thing to have.
- However, sometimes we don't need all the information that a sort finds: an exact order.
- Sometimes we are only interested in finding the smallest or largest element in a collection.
- To solve this problem, we'll explore new ADT called a Priority Queue that solves this problem in logarithmic time.
- Notice that the new problem we have just mentioned could be used to sort an array… indeed; there is a algorithm called Heapsort.

PRIORITY QUEUES: A SAMPLE PROBLEM

- Imagine that we have some very large (endless?) list of data elements, for example, bank transactions, and we want to know what are the M largest ones.
- Since the list is very large, we may not be able to sort them.
- We could also try comparing each element to the others already seen, but that gives an O(MN) algorithm.

- This is a sample problem we want to address. Our goals are:
  - Provide a fast way of doing this.
  - Provide a general abstraction for doing this.

Other Applications:

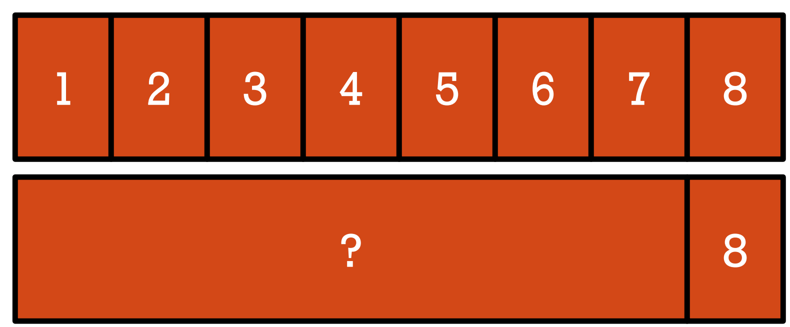
- Event Simulation
- Pathfinding
- …

PRIORITY QUEUES: KEY CONCEPTS

- The basic idea of PQs is to provide two methods:
  - insert: adds a new element to the ADT.
  - delMax: removes the largest (or something equal to it) element in the collection.

(When we say largest, we mean in terms of comparisons.)

- Keep in mind:
  - We don’t need a sorted array.
  - All we need is the highest value.



IMPLEMENTATION: LAZINESS VS EAGERNESS

operation	argument	return value	size	contents (unordered)	contents (ordered)
insert	P		1	P	P
insert	Q		2	P Q	P Q
insert	E		3	P Q E	E P Q
remove max		Q	2	P E	E P
insert	X		3	P E X	E P X
insert	A		4	P E X A	A E P X
insert	M		5	P E X A M	A E M P X
remove max		X	4	P E M A	A E M P
insert	P		5	P E M A P	A E M P P
insert	L		6	P E M A P L	A E L M P
insert	E		7	P E M A P L E	A E E L M
remove max		P	6	E E M A P L	A E E L M

A sequence of operations on a priority queue

In the context of sorting, and other algorithms we have written, we could think about implementing the heart of PQ in either **insert** or **del**.

- We could be *lazy*, and find the maximum of the data for each **del**.
- We could be *eager*, and maintain a sorted array for each **insert**.

Image from Sedgewick and Wayne.

API OVERVIEW

`public class MaxPQ<Key extends Comparable<Key>>`

	`MaxPQ()`	*create a priority queue*
	`MaxPQ(int max)`	*create a priority queue of initial capacity* `max`
	`MaxPQ(Key[] a)`	*create a priority queue from the keys in* `a[]`
`void`	`insert(Key v)`	*insert a key into the priority queue*
`Key`	`max()`	*return the largest key*
`Key`	`delMax()`	*return and remove the largest key*
`boolean`	`isEmpty()`	*is the priority queue empty?*
`int`	`size()`	*number of keys in the priority queue*

The API for PQs is shown above. For now, let us view them as black boxes.

Image from Sedgewick and Wayne.

API SAMPLE CLIENT

```
public class TopM
{
   public static void main(String[] args)
   {  // Print the top M lines in the input stream.
      int M = Integer.parseInt(args[0]);
      MinPQ<Transaction> pq = new MinPQ<Transaction>(M+1);
      while (StdIn.hasNextLine())
      {  // Create an entry from the next line and put on the PQ.
         pq.insert(new Transaction(StdIn.readLine()));
         if (pq.size() > M)
            pq.delMin();     // Remove minimum if M+1 entries on the PQ.
      }  // Top M entries are on the PQ.

      Stack<Transaction> stack = new Stack<Transaction>();
      while (!pq.isEmpty()) stack.push(pq.delMin());
      for (Transaction t : stack) StdOut.println(t);
   }
}
```

```
% more tinyBatch.txt
Turing       6/17/1990   644.08
vonNeumann   3/26/2002  4121.85
Dijkstra     8/22/2007  2678.40
vonNeumann   1/11/1999  4409.74
Dijkstra    11/18/1995   837.42
Hoare        5/10/1993  3229.27
vonNeumann   2/12/1994  4732.35
Hoare        8/18/1992  4381.21
Turing       1/11/2002    66.10
Thompson     2/27/2000  4747.08
Turing       2/11/1991  2156.86
Hoare        8/12/2003  1025.70
vonNeumann  10/13/1993  2520.97
Dijkstra     9/10/2000   708.95
Turing      10/12/1993  3532.36
Hoare        2/10/2005  4050.20
```

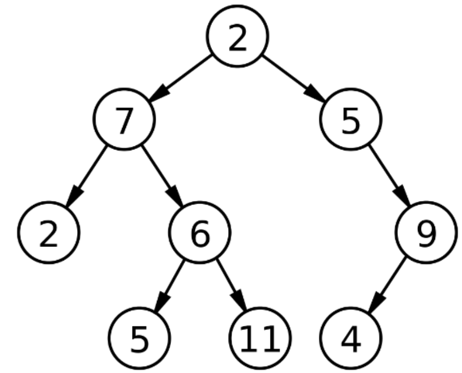
```
% java TopM 5 < tinyBatch.txt
Thompson     2/27/2000  4747.08
vonNeumann   2/12/1994  4732.35
vonNeumann   1/11/1999  4409.74
Hoare        8/18/1992  4381.21
vonNeumann   3/26/2002  4121.85
```


Image from Sedgewick and Wayne.

9 HEAP DATA STRUCTURES

HEAPS: OVERVIEW

- **Definition**: A *binary tree* is a tree where each node has at most two children.



- **Definition**: A *binary search tree* is a binary tree where each node's left child has a key less than the parent, and the right child has a key greater than the parent.

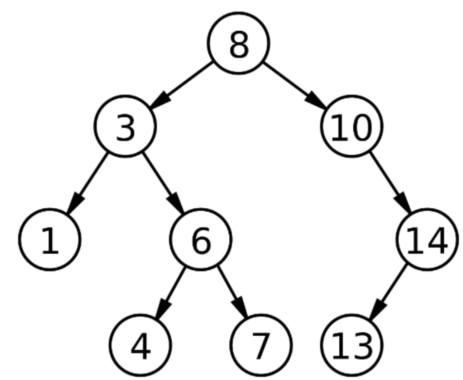
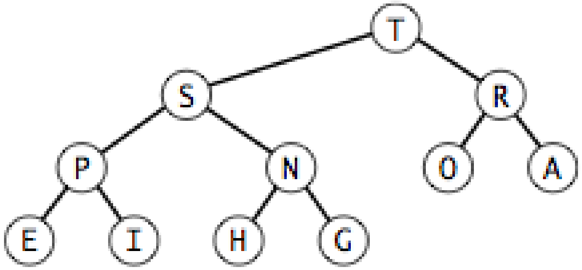


Image from Wikipedia Commons.

Picking an underlying structure is often the first step.

HEAPS: OVERVIEW

- We need a slightly looser concept:
- **Definition**: "A binary tree is *heap-ordered* if each node is larger than or equal to the keys in that node's two children (if any)."



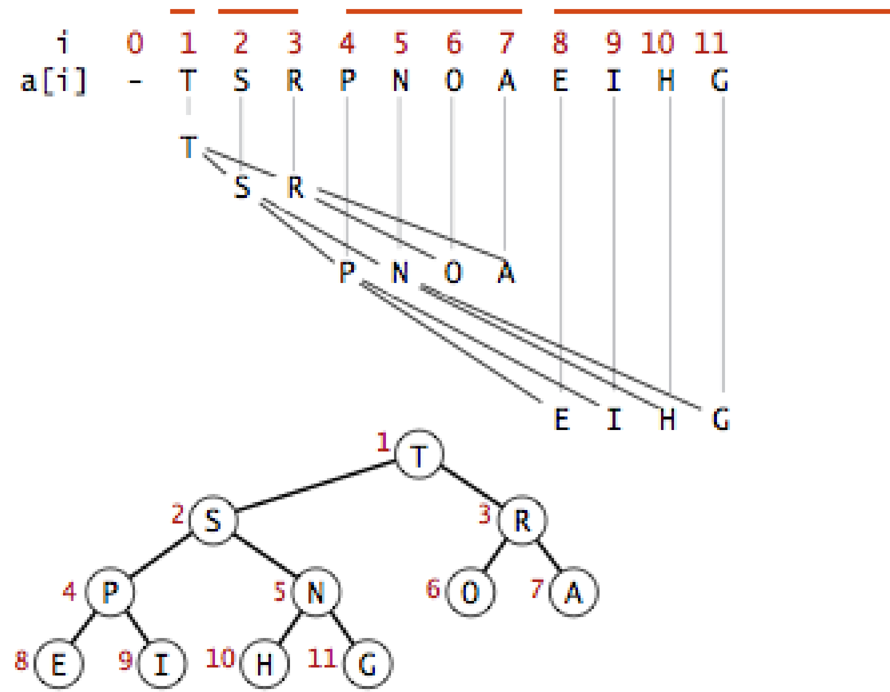
A heap-ordered complete binary tree

- Where is the largest element stored?
- For a tree with *n* elements, how many nodes are between a leaf and the root?

Image from Sedgewick and Wayne.

HEAPS: ARRAY MAPPING

- Although the tree could be represented as a linked structure, that is a little complicated. And slow!
- Instead, we can encode a tree into an array by assuming it is *complete* and flattening it.
- **Definition**: a *complete binary tree* is one where every level is full except the last.
- This will make accessing other nodes fast (constant time).
- For some node k:
  - $p = \left\lfloor \frac{k}{2} \right\rfloor$ parent
  - c=k2 first child
  - c=k2+1 second child



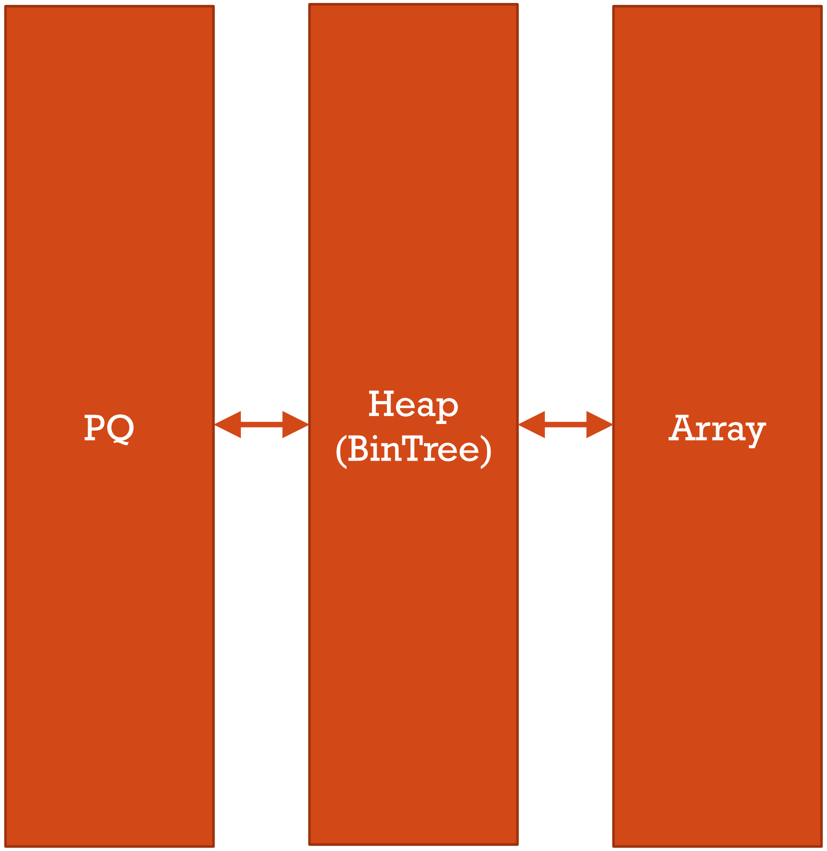
Heap representations

Image from Sedgewick and Wayne.

ADT→REPRESENTATION→STRUCTURE

- Before we continue, we should take time to acknowledge one of the key ideas that priority queues illustrate: the separation of representation and structure.
- (One might say that there is an *encoding* or *projection* step.)

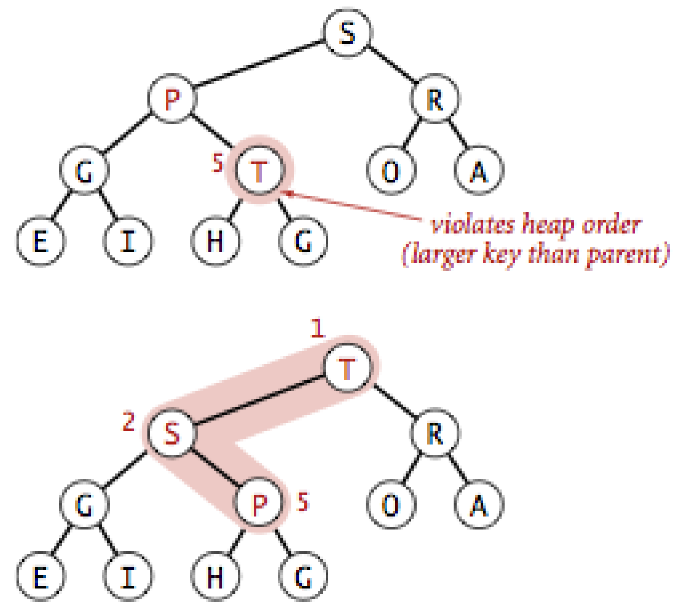
- A PQ is *represented* as a binary tree.
- A PQ is *structured* as an array.
- Remember that representation != structure.



HEAP OPERATIONS: SWIM

- So how do we manage this thing? Let's start by defining two useful operations.

- Say there's an element in the tree that is (potentially) larger than its parent.
- We need to migrate that element upward until the properties of a heap hold.

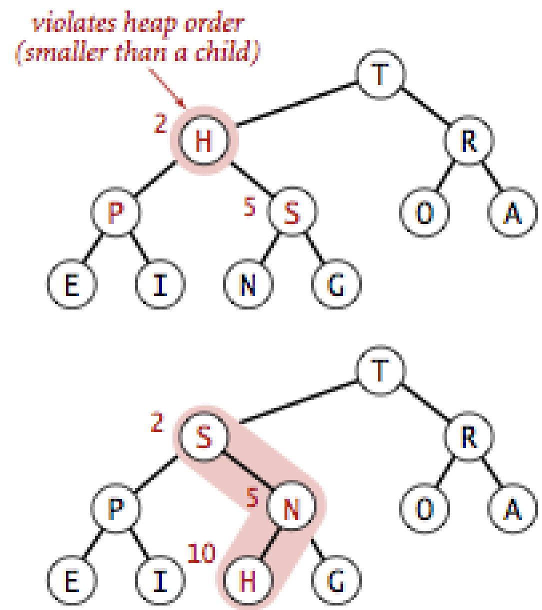



```
private void swim(int k) {
    while (k > 1 && less(k/2, k)) {
        exch(k, k/2);
        k = k/2;
    }
}
```


Image from Sedgewick and Wayne.

HEAP OPERATIONS: SINK

- Let's say there is a parent in the tree that is (potentially) smaller than one of its children.
- We need to migrate that element downward until the properties of a heap hold.




```
private void sink(int k) {
    while (2*k <= N) {
        int j = 2*k;
        if (j < N && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```


Image from Sedgewick and Wayne.

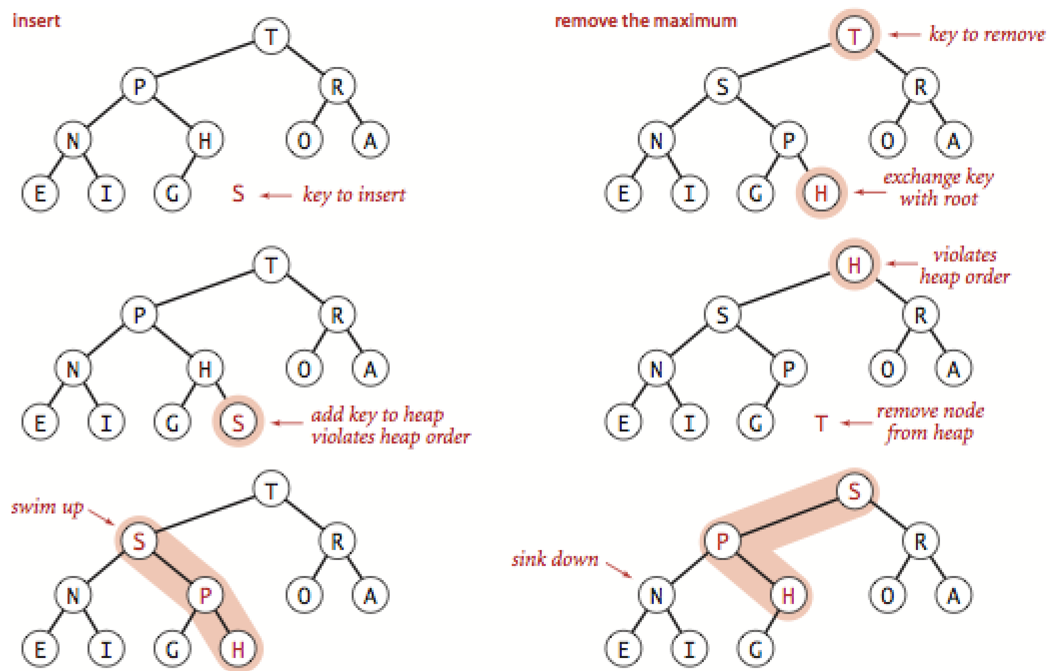
16 IMPLMENTATION

KEY OPERATIONS

- Using sink and swim, it is relatively easy to implement insert and del:

```
public void insert(Key v) {
    pq[++N] = v;
    swim(N);
}

public Key delMax() {
    Key max = pq[1];
    exch(1, N--);
    pq[N+1] = null;
    sink(1);
    return max;
}
```



Heap operations

Image from Sedgewick and Wayne.

FINAL CODE

```java
public class MaxPQ<Key extends Comparable<Key>> {
    private Key[] pq;
    private int N = 0;

    public MaxPQ(int maxN) {
        pq = (Key[]) new Comparable[maxN+1];
    }

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

    public int size() {
        return N;
    }

    public void insert(Key v) {
        pq[++N] = v;
        swim(N);
    }

    public Key delMax() {
        Key max = pq[1];
        exch(1, N--);
        pq[N+1] = null;
        sink(1);
        return max;
    }
    private void swim(int k) {
        while (k > 1 && less(k/2, k)) {
            exch(k, k/2);
            k = k/2;
        }
    }

    private void sink(int k) {
        while (2*k <= N) {
            int j = 2*k;
            if (j < N && less(j, j+1)) j++;
            if (!less(k, j)) break;
            exch(k, j);
            k = j;
        }
    }

    private boolean less(int i, int j) {
            return ((Comparable<Key>) pq[i])
                      .compareTo(pq[j]) < 0;
    }

    private void exch(int i, int j) {
        Key swap = pq[i];
        pq[i] = pq[j];
        pq[j] = swap;
    }
}
```

FINAL CODE

- Overtime, a partially sorted structure is formed and then maintained as calls to insert and delMax are made.

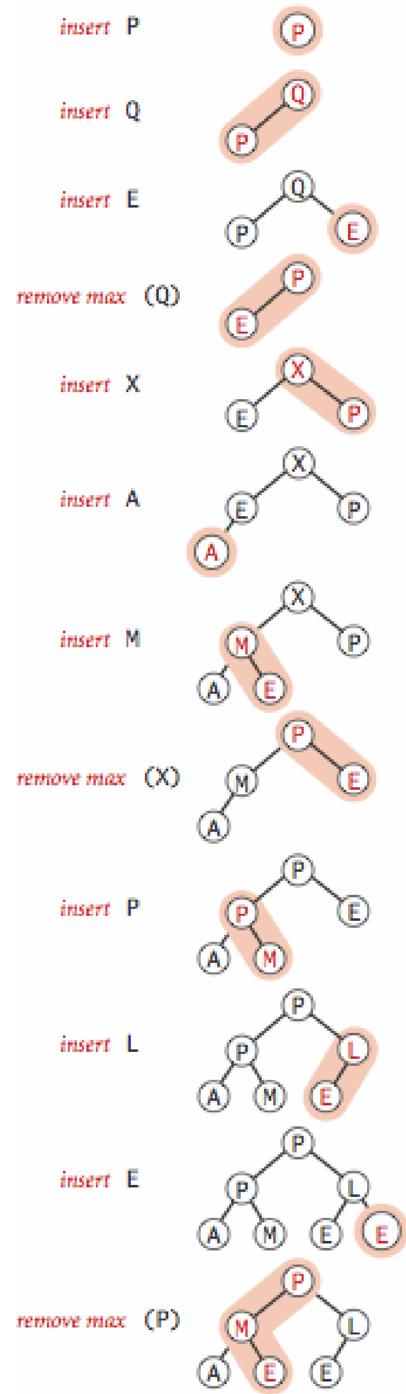


Image from Sedgewick and Wayne.

HEAPSORTS

- The simplest way to sort an array with a PQ is to insert all of the elements (nlog(n)), and then list them out (nlog(n)).

- The other way is to do it in place, treating the array as a PQ, and fixing tree issues one layer at a time. Then, taking out the largest elements and putting them in the final position.

```
public  void sort(Comparable[] a) {

        int N = a.length;
        for (int k = N/2; k >= 1; k--)
            sink(a, k, N);
        while (N > 1) {
            exch(a, 1, N--);
            sink(a, 1, N);
        }
    }
```