جامعة بيرزيت
BIRZEIT UNIVERSITY

COMP242

Data Structure

Lectures Note: Heaps

Prepared by: **Dr. Mamoun Nawahdah**

2016/2017

Priority Queues (Heaps)

A ***priority queue*** is a data structure that allows **at least** the following two operations:

- **Insert**: which does the obvious thing;
- **deleteMin (or deleteMax)**: which finds, returns, and removes the minimum (or maximum) element in the priority queue.

Simple Implementations:

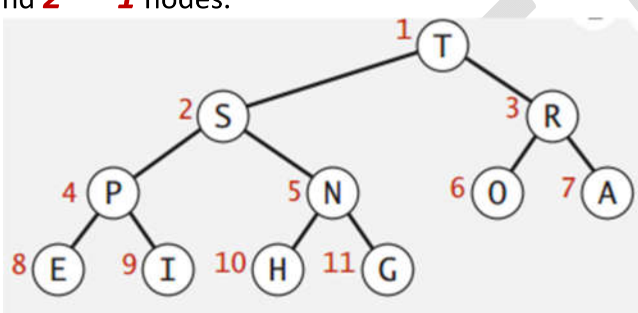
- **Unsorted Linked list**, performing insertions at the front in **O(1)** and traversing the list, which requires **O(N)** time, to delete the minimum/maximum.
- **Sorted Linked list**, performing insertions in **O(N)** and **O(1)** to delete the minimum/maximum.
- **Binary search tree**: this gives an **O(log N)** average running time for both operations.

Binary Heap

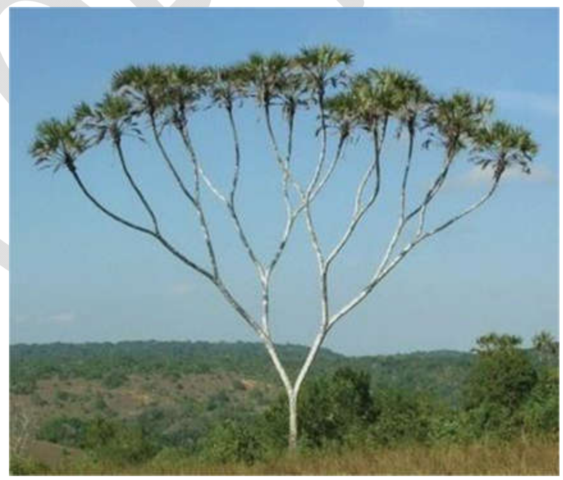
A heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.

Such a tree is known as a **complete binary tree**.

A complete binary tree of height ***h*** has between ***2^h*** and ***$2^{h+1}-1$*** nodes.



Heap representations



As complete binary tree is so **regular**, therefore, it can be represented as an array:

i	0	1	2	3	4	5	6	7	8	9	10	11
a[i]	-	T	S	R	P	N	O	A	E	I	H	G

- Parent of node at ***i*** is at ***i/2***.
- Children of node at ***i*** are at ***2i*** (left child) and ***2i+1*** (right child).

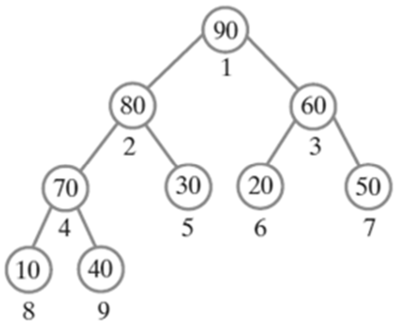
Heap-order property:

- In a **min heap**, for every node **X**, the key in the parent of **X** is smaller than (*or equal to*) the key in **X**, with the exception of the root (which has no parent). Therefore, the minimum element can always be found at the root.
- In a **max heap**, for every node **X**, the key in the parent of **X** is larger than (*or equal to*) the key in **X**, with the exception of the root (which has no parent). Therefore, the maximum element can always be found at the root.

Interface for the max-heap

```
public interface MaxHeapInterface<T extends Comparable<? super T>>
{
   public void add(T newEntry);
   public T removeMax();
   public T getMax();
   public boolean isEmpty();
   public int getSize();
   public void clear();
} // end MaxHeapInterface
```

An Array to Represent a Heap



0	1	2	3	4	5	6	7	8	9	10	11	12
	90	80	60	70	30	20	50	10	40			

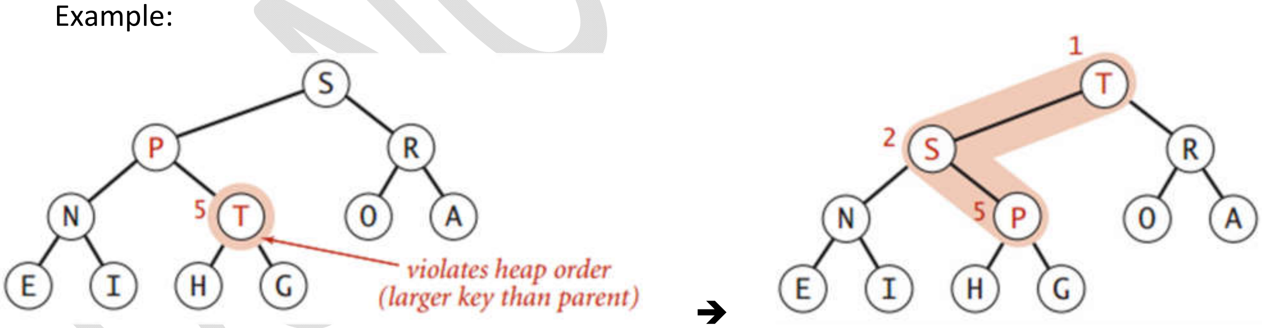
Promotion (ترفيع) in a max heap

Scenario: Child's key becomes larger than its parent's key.

To eliminate the violation:

- Exchange key in **child** with key in **parent**.
- Repeat until heap order restored.

Example:




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

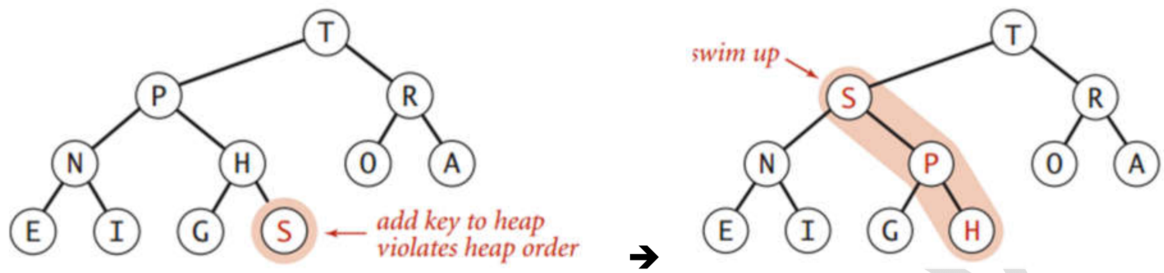
parent of node at k is at k/2

Insertion in a max heap

Insert: Add node at end, then swim it up.

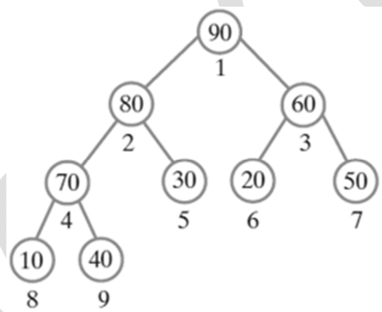
Cost: At most ***1 + log N*** compares.

Example 1: insert **S**

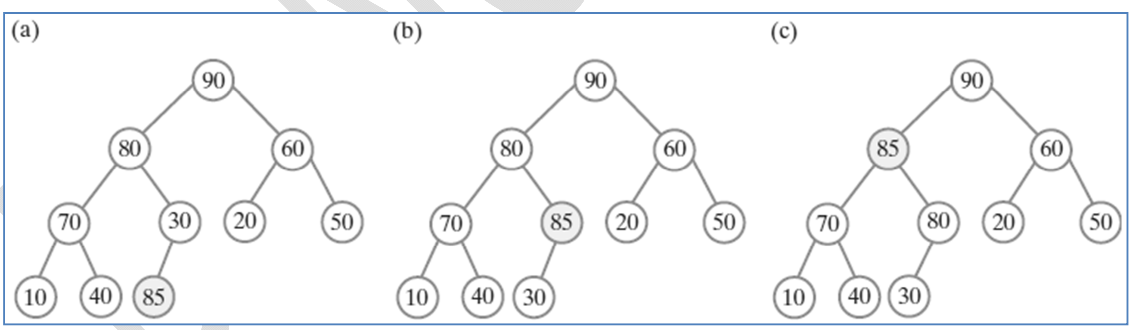



```
public void insert(Key x)
{
   pq[++N] = x;
   swim(N);
}
```

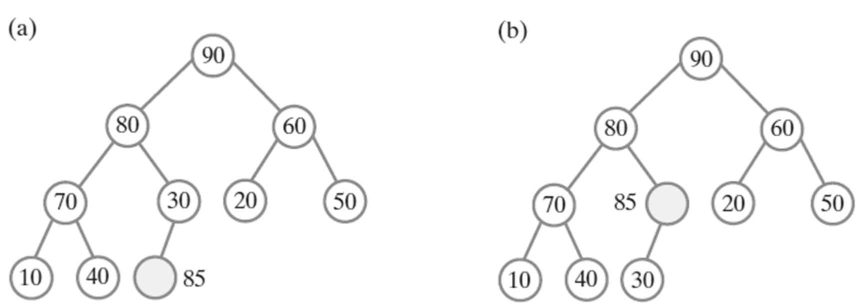
Example 2: insert **85**

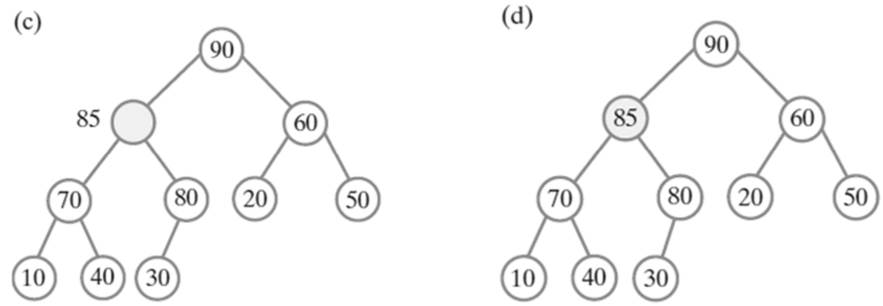


Method 1: The steps in adding 85 to the previous max-heap



Method 2: A revision of the steps shown in the previous figure, **to avoid swaps**:





The following figures shows array representation of the steps in the previous figures:

(a)

0	1	2	3	4	5 (10/2)	6	7	8	9	10	11	12	
	90	80	60	70	**30**	20	50	10	40				85 > 30

(b)

0	1	2	3	4	5	6	7	8	9	10	11	12	
	90	80	60	70		20	50	10	40	**30**			Move 30

(c)

0	1	2 (5/2)	3	4	5	6	7	8	9	10	11	12	
	90	**80**	60	70		20	50	10	40	30			85 > 80

(d)

0	1	2	3	4	5	6	7	8	9	10	11	12	
	90		60	70	**80**	20	50	10	40	30			Move 80

(e)

0	1 (2/2)	2	3	4	5	6	7	8	9	10	11	12	
	90		60	70	80	20	50	10	40	30			85 < 90

(f)

0	1	2	3	4	5	6	7	8	9	10	11	12	
	90	**85**	60	70	80	20	50	10	40	30			Insert 85

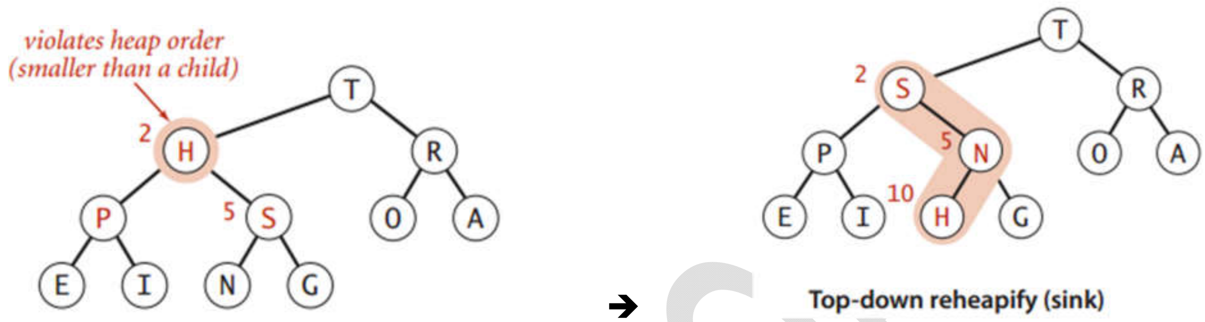
Demotion (إنزال رتبة) in a max heap

Scenario: Parent's key becomes smaller than one (or both) of its children's.

To eliminate the violation:

- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

Example 1:



Top-down reheapify (sink)

```
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;
   }
}
```

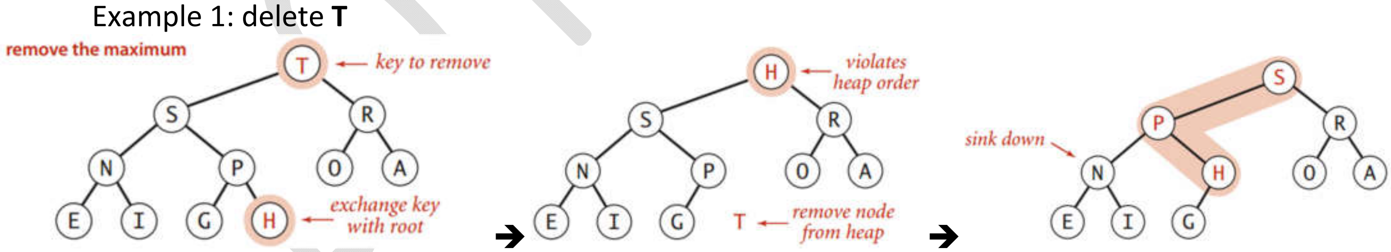
children of node at k are 2k and 2k+1

Delete the maximum in a max heap (Removing the root)

Delete max: Exchange root with node at end, and then sink it down.

Cost: At most ***2 log N*** compares.

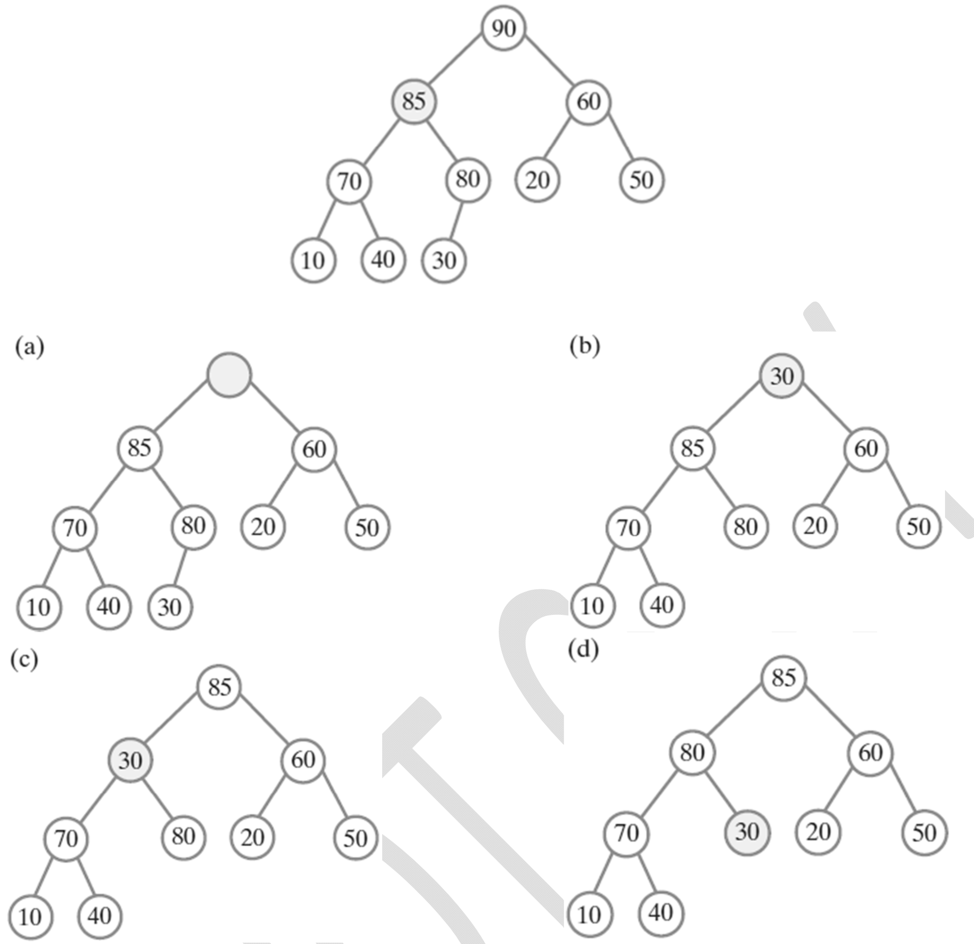
Example 1: delete **T**




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

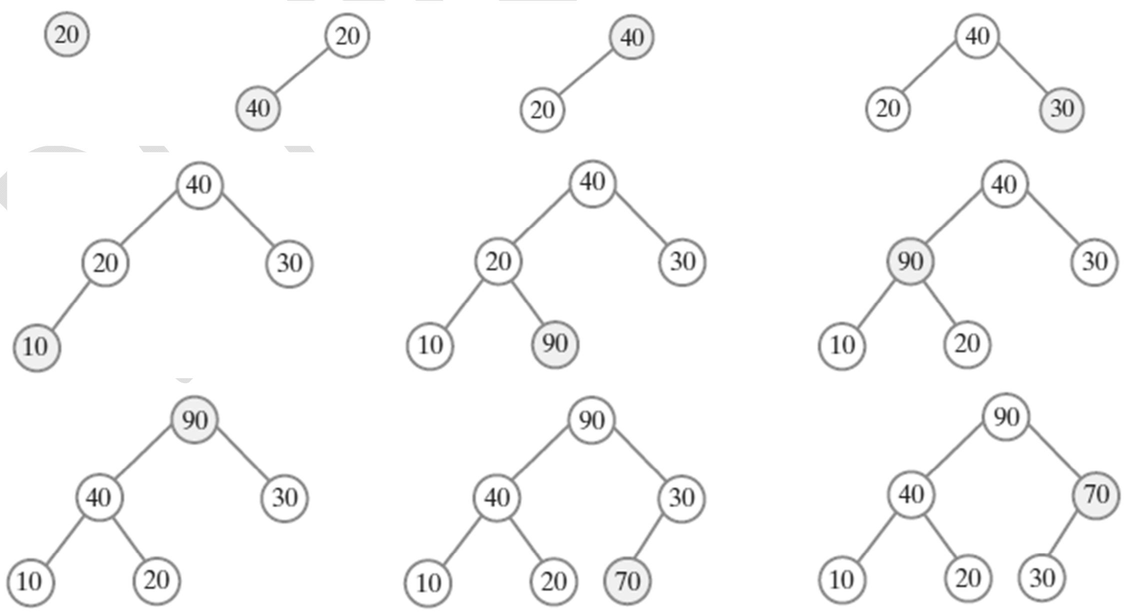
prevent loitering

Example 2: delete **root (max)**



Creating a Heap

The steps in adding 20, 40, 30, 10, 90, and 70 to an initially empty heap



	20	40	30	10	90	70
0	1	2	3	4	5	6

Binary heap: Java implementation

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

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

   public boolean isEmpty()
   {   return N == 0;    }
   public void insert(Key key)
   public Key delMax()
   {   /* see previous code */   }

   private void swim(int k)
   private void sink(int k)
   {   /* see previous code */   }

   private boolean less(int i, int j)
   {   return pq[i].compareTo(pq[j]) < 0;  }
   private void exch(int i, int j)
   {   Key t = pq[i]; pq[i] = pq[j]; pq[j] = t;  }
}
```

fixed capacity (for simplicity)

PQ ops

heap helper functions

array helper functions

HeapSort

Basic plan:

- Create max heap with all ***N*** keys.
- Repeatedly remove the maximum key.

Heapsort demo:

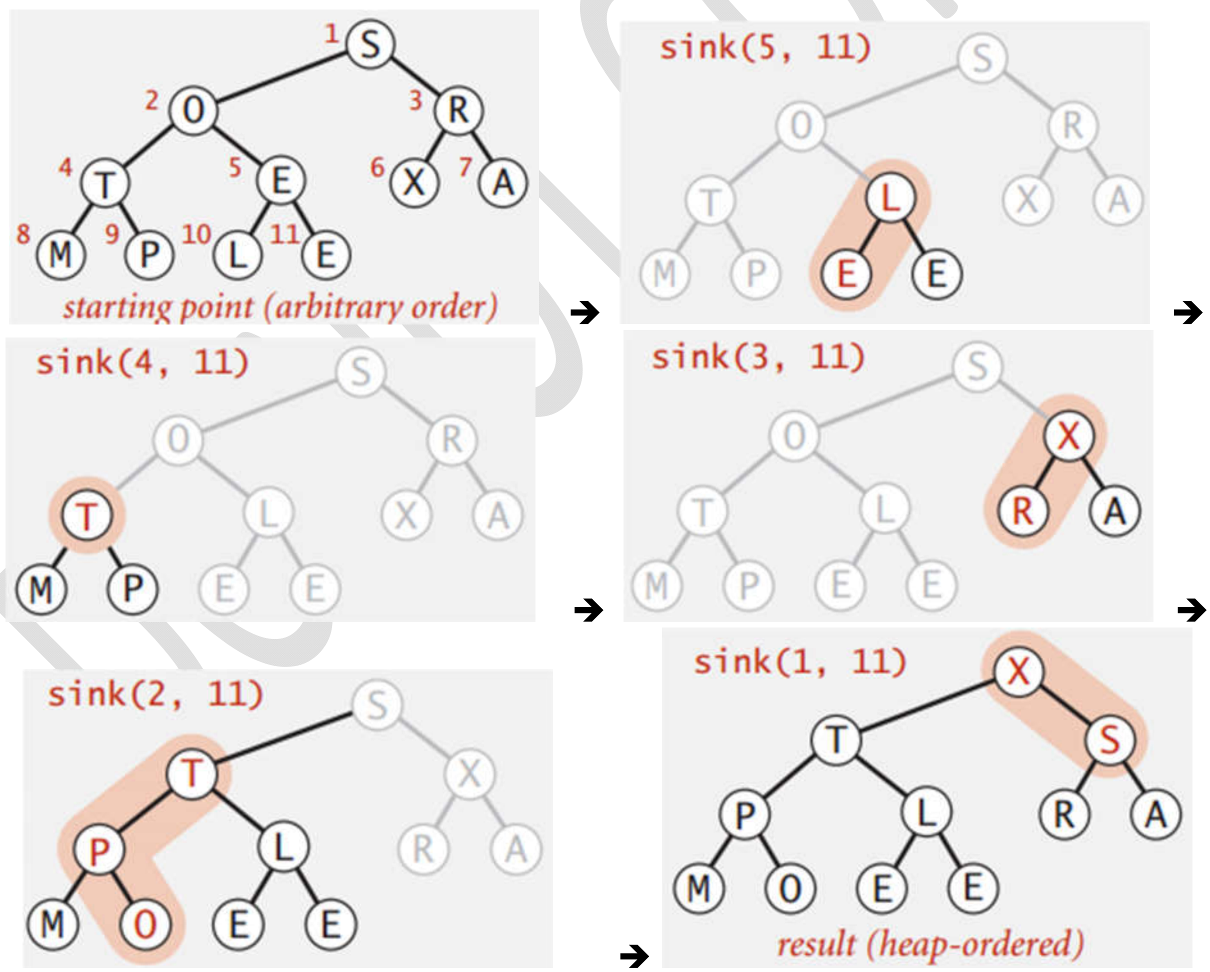
- **First pass**. Build heap using **bottom-up method**:

Array in arbitrary (random) order

S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

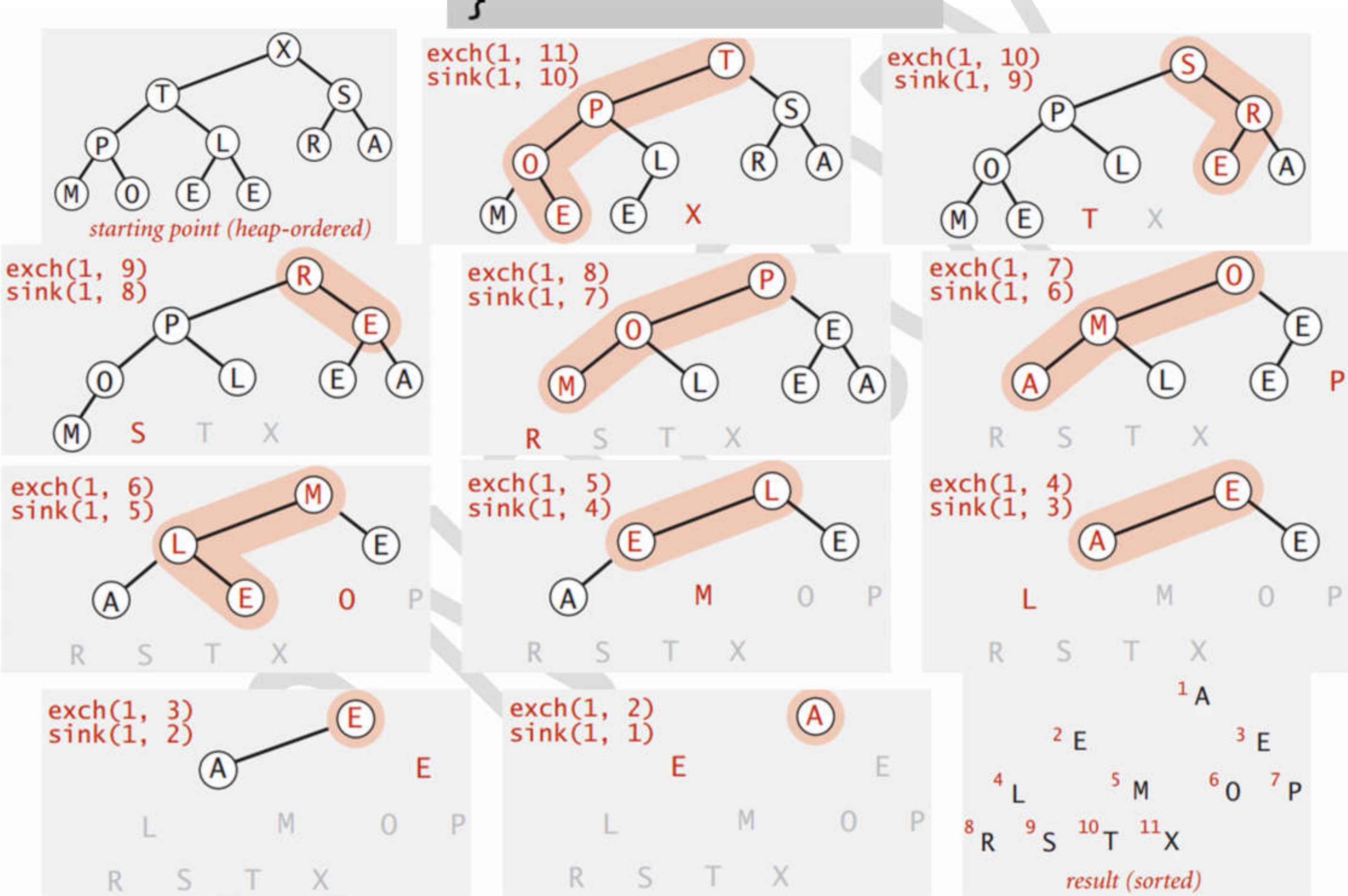
N=11

```
for (int k = N/2; k >= 1; k--)
    sink(a, k, N);
```



- **Second pass**:
  - Remove the maximum, one at a time.
  - Leave in array, instead of nulling out.

```
while (N > 1)
{
    exch(a, 1, N--);
    sink(a, 1, N);
}
```



Heapsort: trace

N	k	0	1	2	3	4	5	6	7	8	9	10	11
initial values			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	L	X	A	M	P	E	E
11	4		S	O	R	T	L	X	A	M	P	E	E
11	3		S	O	X	T	L	R	A	M	P	E	E
11	2		S	T	X	P	L	R	A	M	O	E	E
11	1		X	T	S	P	L	R	A	M	O	E	E
heap-ordered			X	T	S	P	L	R	A	M	O	E	E
10	1		T	P	S	O	L	R	A	M	E	E	X
9	1		S	P	R	O	L	E	A	M	E	T	X
8	1		R	P	E	O	L	E	A	M	S	T	X
7	1		P	O	E	M	L	E	A	R	S	T	X
6	1		O	M	E	A	L	E	P	R	S	T	X
5	1		M	L	E	A	E	O	P	R	S	T	X
4	1		L	E	E	A	M	O	P	R	S	T	X
3	1		E	A	E	L	M	O	P	R	S	T	X
2	1		E	A	E	L	M	O	P	R	S	T	X
1	1		A	E	E	L	M	O	P	R	S	T	X
sorted result			A	E	E	L	M	O	P	R	S	T	X

Heapsort trace (array contents just after each sink)

Heapsort: mathematical analysis

- Heap construction uses ***≤ 2 N*** compares and exchanges.
- Heapsort uses ***≤ 2 N lg N*** compares and exchanges.

Heapsort Significance: **In-place sorting** algorithm with ***N log N*** worst-case.
Heapsort is optimal for both time and space, but it makes poor use of cache memory and not stable.

Heapsort: Java implementation

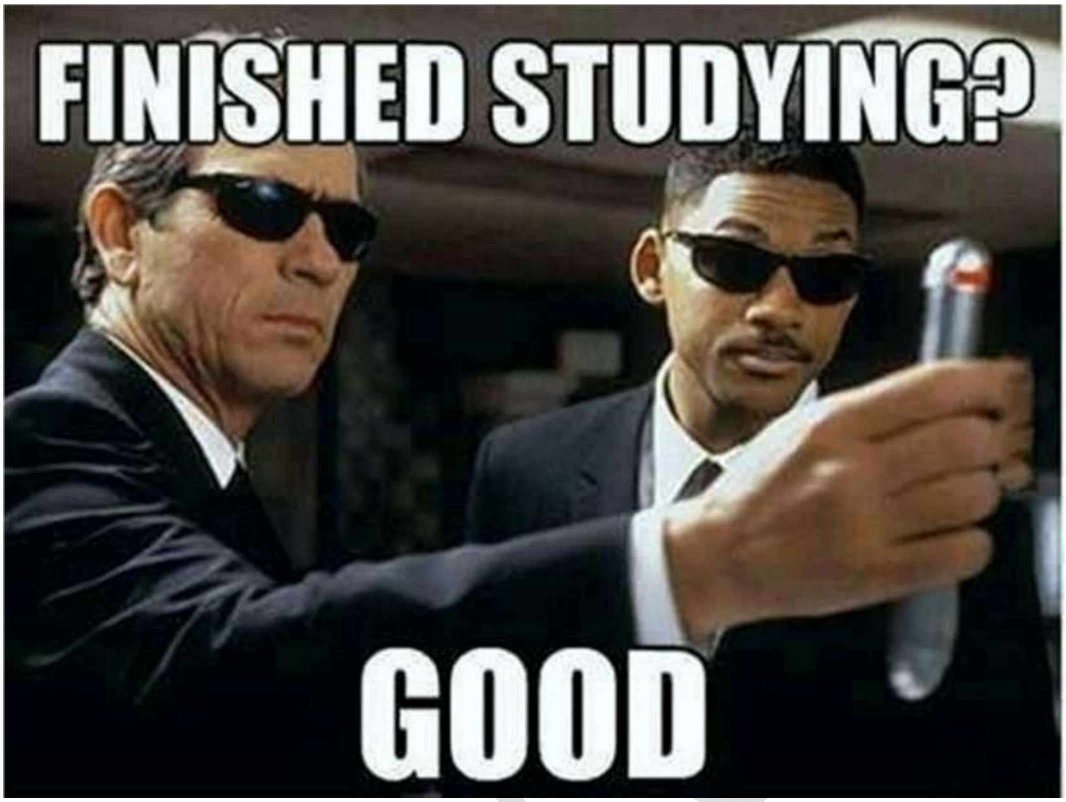
```
public class Heap
{
   public static void sort(Comparable[] a)
   {
      int N = a.length-1;
      for (int k = N/2; k >= 1; k- )
         sink(a, k, N);
      while (N > 1)
      {
         exch(a, 1, N);
         sink(a, 1, --N);
      }
   }

   private static void sink(Comparable[] a, int k, int N)
   {  /* as before */  }

   private static boolean less(Comparable[] a, int i, int j)
   {  /* as before */  }

   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */  }

}
```


FINISHED STUDYING?
GOOD