ID1020: Priority Queues

Dr. Per Brand
perbrand@sics.se

kap 2.4

Slides adapted from Algoritms 4th Edition, Sedgewick.

Priority Queues

- Collections. Insert, delete and retrieve items.

- Stack. Retrieve the newest item (most recently added).
- Queue. Retrieve the oldest item
- Randomized queue. Retrieve a random item.
- Priority queue. Retrieve the largest (or smallest) item.

Priority Queue API

Key must be Comparable
(bounded type parameter)

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

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

How to implement?

- Naive methods
  - Unordered array representation
    - Stack
    - Insert = push
    - Remove the maximum = search array and exchange with last entry
  - Ordered array representation
    - Use insertion sort to keep array in order
- Can we do better ?

Unordered and ordered arrays

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	P	
insert	E		7	P	E	M	A	P	L	E	A	E	E	L	M	P	P
remove max		P	6	E	M	A	P	L	E		A	E	E	L	M	P	

A sequence of operations on a priority queue

Unordered array implementation

```
public class UnorderedMaxPQ<Key extends Comparable<Key>>
{
   private Key[] pq;    // pq[i] = ith element on pq
   private int N;       // number of elements on pq

   public UnorderedMaxPQ(int capacity)
   {  pq = (Key[]) new Comparable[capacity];   }

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

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

   public Key delMax()
   {
      int max = 0;
      for (int i = 1; i < N; i++)
         if (less(max, i)) max = i;
      exch(max, N-1);
      return pq[--N];
   }
}
```

Less, exch from sorting

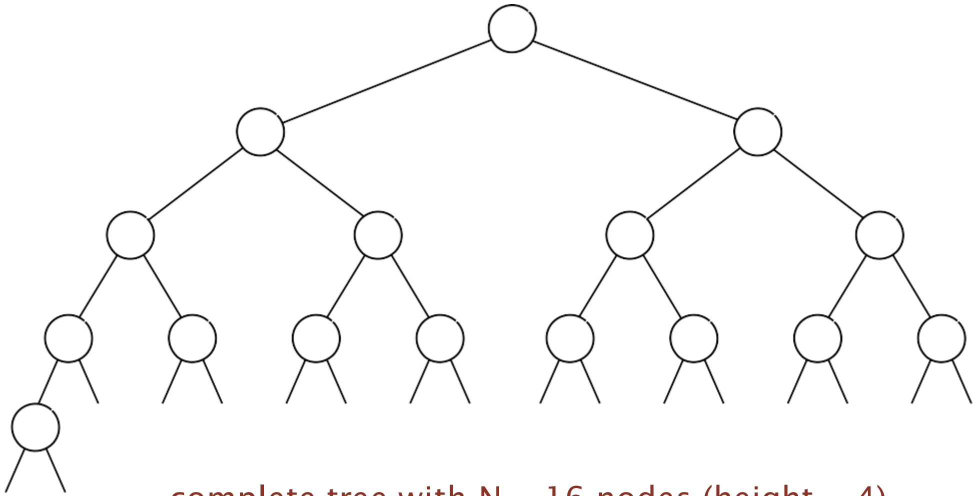
Properties

order of growth of running time for priority queue with N items

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
goal	log N	log N	log N

Binary tree

- Each node has one parent and up to two children
- Complete tree – balanced (except for bottom layer)



complete tree with N = 16 nodes (height = 4)

- Depth of tree is O(lg N)
- By convention computer scientist draw their trees upside down

Heap-ordered

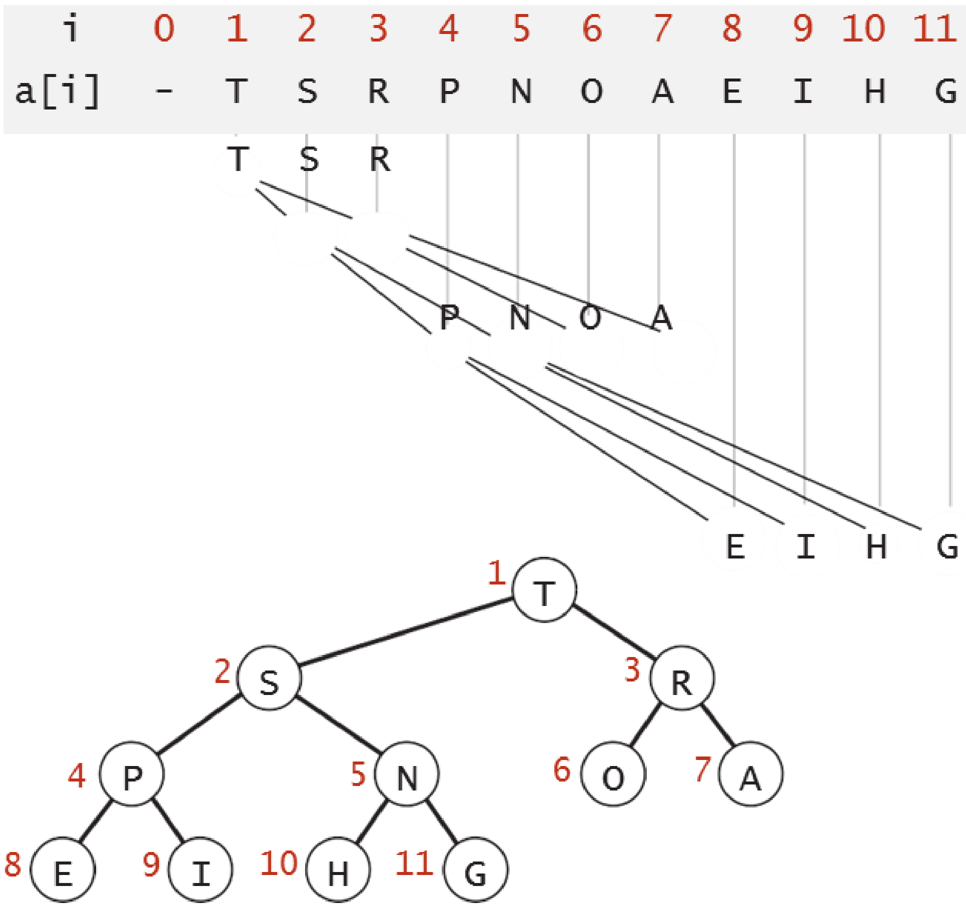
- Definition: Binary tree is heap-ordered if key in each node is larger (or equal) to the keys in its two children
  - Largest key in root

- How to implement?
  - Method 1: Linked representation.
    - Data structure is a node
    - Three links (parent, left child, right child)
  - Method 2: Use an array with implicit links
    - Children of element ***i*** are in element 2***i*** and 2***i***+1

  - Book prefers method 2. Why?

Binary heap

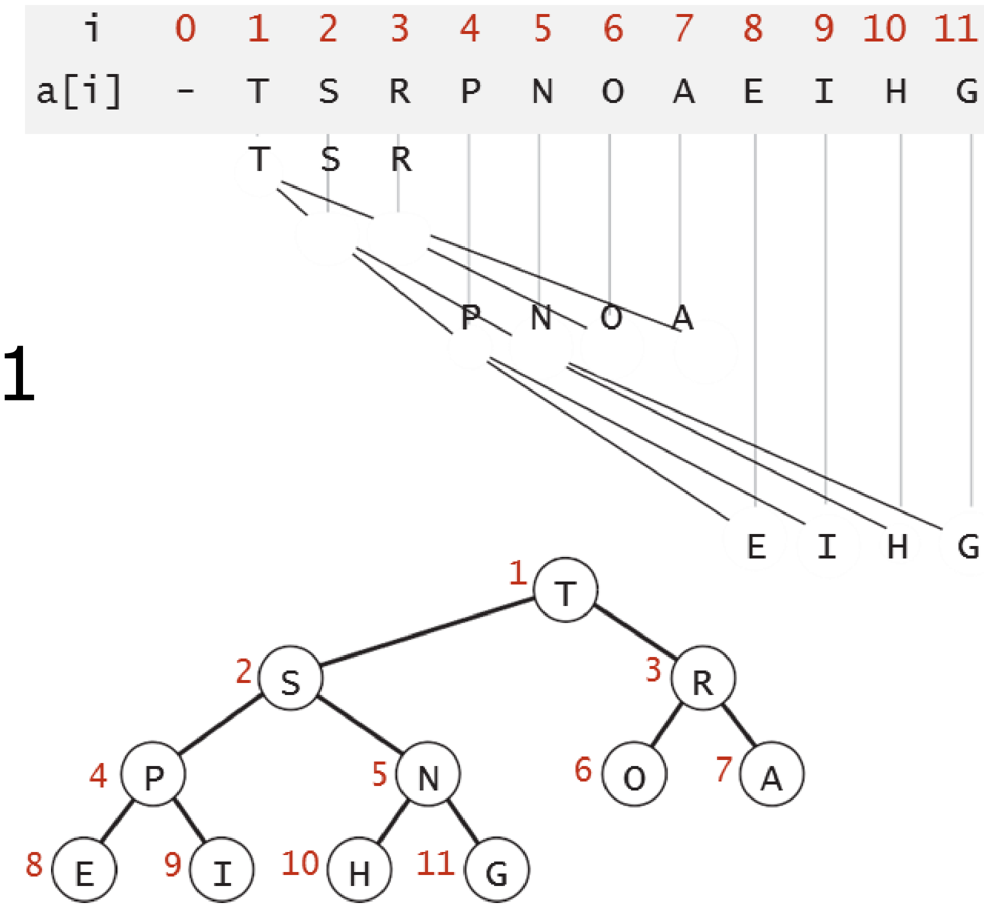
- Array representation of a heap-ordered complete binary tree
- Indices start at 1
- Order top down and left to right



Heap representations

Binary heap (2)

- Largest key at index 1
  the root
- Parent is [i/2]
- Children at $2i$ and $2i+1$



Heap representations

How to keep it ordered

- Disorder introduced by
  - Removing an item (typically the root)
    - Replace the element by the last item
    - Array kept compact
  - Adding an item (new last item)
  - Changing the key of an item

- Two violations
  - The key in a node becomes larger than parent
  - The key in a node becomes smaller than one (or both) of its children

- Question: Which operations (may) produce which violations?

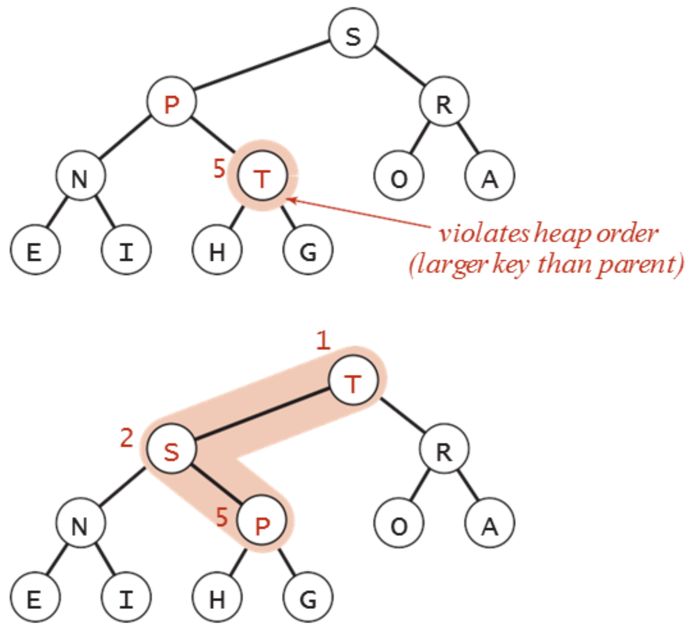
Bottom-up reheapify

- When a child’s key becomes larger than its parent’s
- Algorithm
  - Exchange child with parent
  - Then move up the tree repeating until in order
  - Possibly all the way to the root

Also called promotion or swim

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

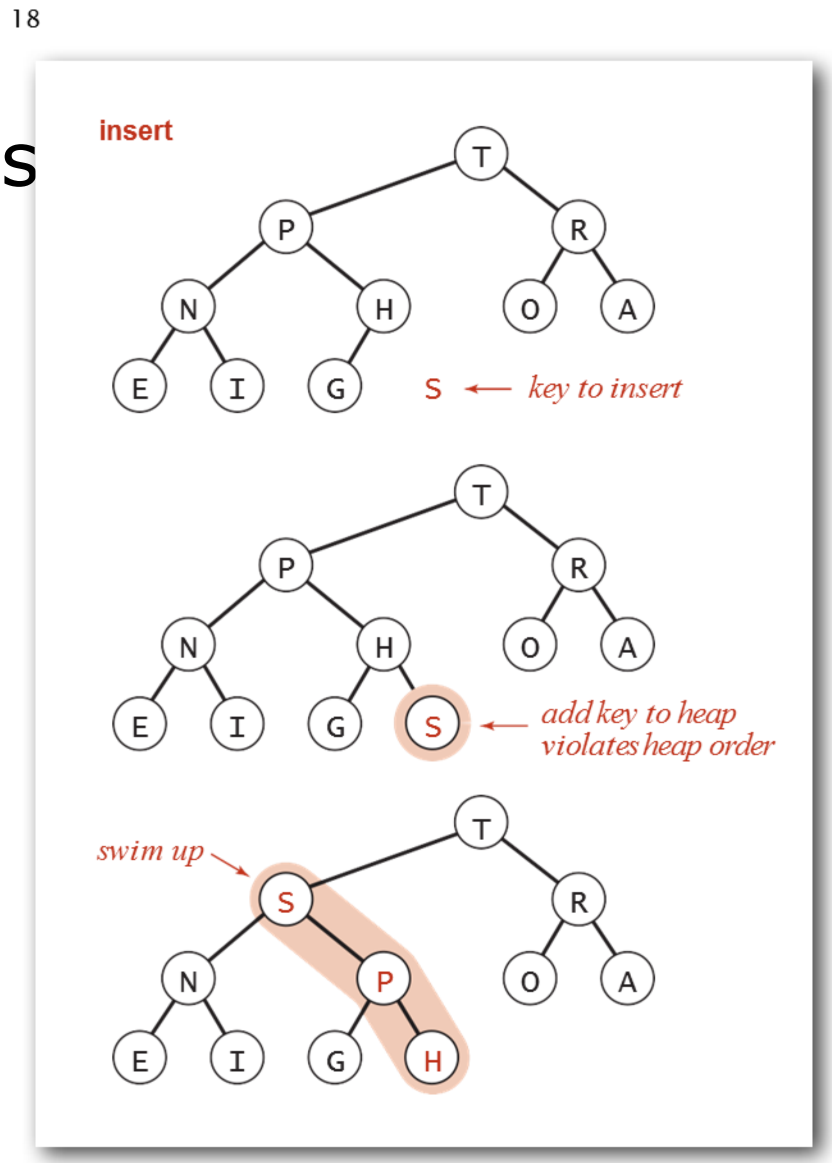
parent of node at k is at k/2



Insertion

- Add node to end and promote
- Cost: At most 1+ lgN operations

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



Top-down reheapify

- When a node becomes smaller that one or both of its children
- Algorithm:
  - Exchange with largest child
  - Repeat moving down until order is found

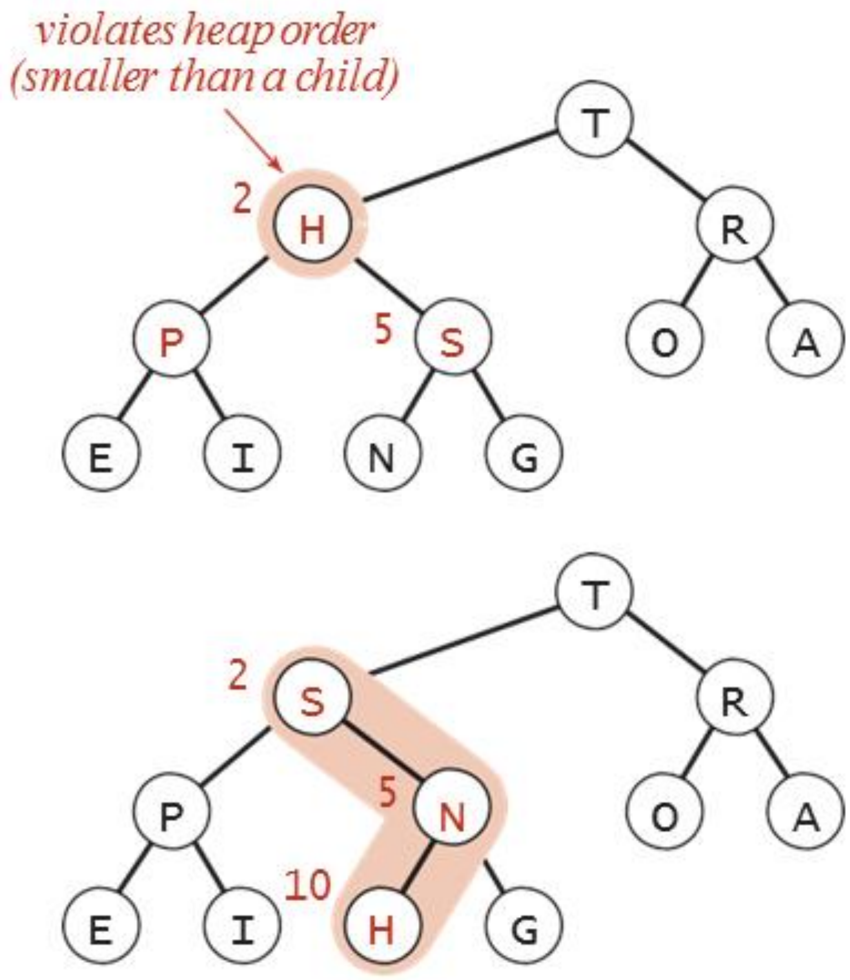
- Why with largest child?

```
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

Top-down reheapify (2)

- Also called demotion and sink



Top-down reheapify (sink)

Removing the maximum

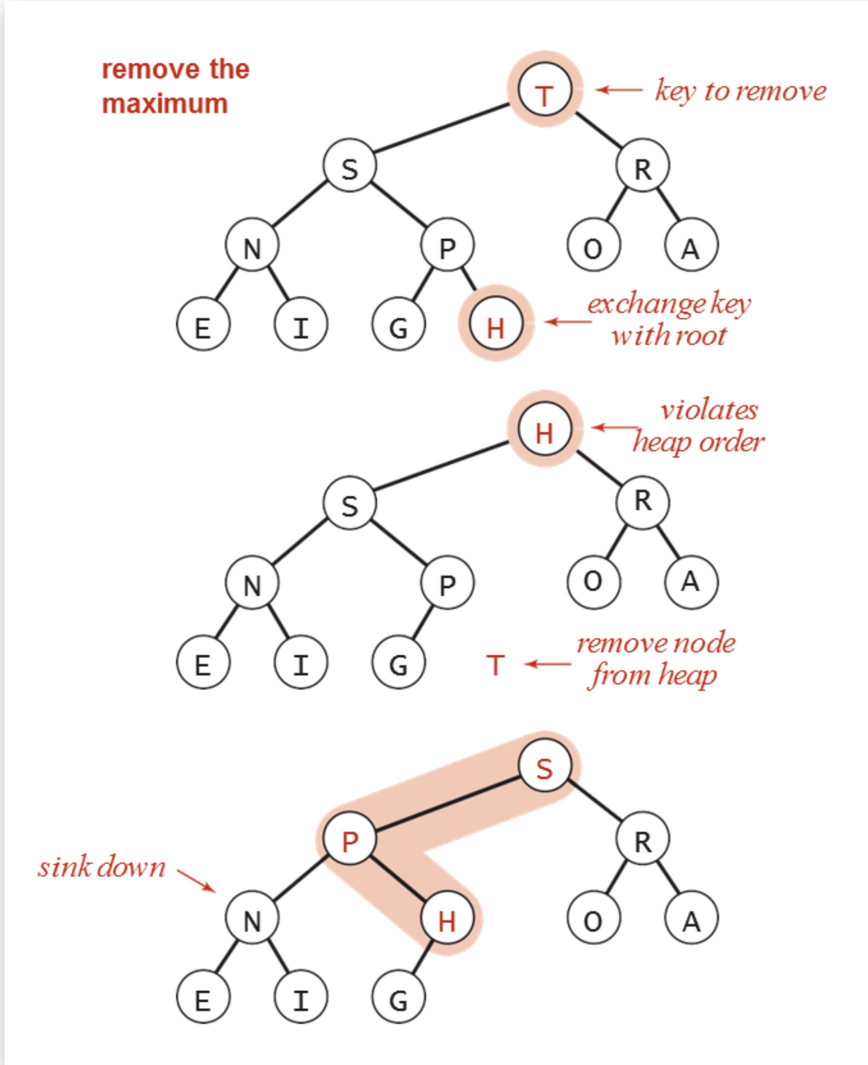
- Exchange root node with last node
  - top-down reheapify

- Cost:
  - At most 2 lg N compares
  - Why 2?
  -

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

Why ?

20



Binary heap implementation

```
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

Odds and Ends

- Ternary trees (multiway heaps)
  - Generalization: Each node has D children
  - Children of node i can be found at
    - ... Di-2, Di -1, Di, Di +1
- Array resizing
  - Just like we did we stacks
  - Invisible to client
  - Upon need doubles the heap size
  - Cost amortized
- Removing arbitrary item
  - Left as an exercise

Immutability

- Our assumption: items are immutable
  - Best practice keys are immutable
- If clients can update objects in heap
  - Could violate ordering constraints
- Especially important across interfaces/APIs

“ Classes should be immutable unless there's a very good reason to make them mutable.... If a class cannot be made immutable, you should still limit its mutability as much as possible. ”

— Joshua Bloch (Java architect)

Immutable data types

- Advantages
  - Simplifies debugging
  - Safer in the presence of hostile code
    - Hostility may be inadvertent
  - Simplifies concurrent programming
  - Safe to use as keys in data structures
    - Priority queue, symbol tables, etc.

- Disadvantage
  - Must create a new data object for each data type value
  -

Example in Java

```
public final class Vector {
   private final int N;
   private final double[] data;

   public Vector(double[] data) {
      this.N = data.length;
      this.data = new double[N];
      for (int i = 0; i < N; i++)
         this.data[i] = data[i];
   }

   ...
}
```

can't override instance methods

all instance variables private and final

defensive copy of mutable instance variables

instance methods don't change instance variables

Cost summary

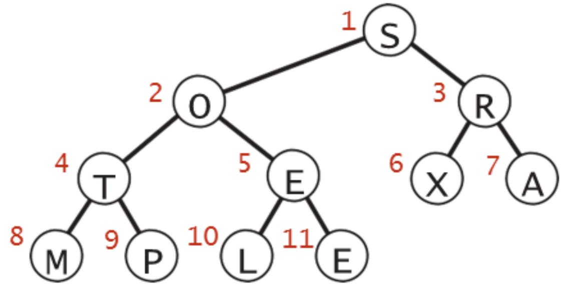
Implementation	Insert	Remove Max	Retrieve Max
Unordered array	1	N	N
Ordered array	N	1	1
Binary heap	log N	2 log N	1
D-ary heap	$\log_D N$	$D \log_D N$	1
Impossible	1	1	1

Revisit sorting

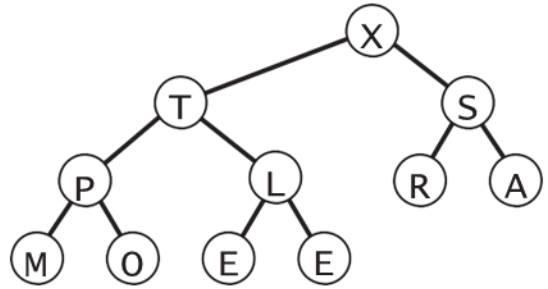
- We can use a priority queue to sort
  - Insert items one by one into a minimum-oriented priority queue
  - Then repeatedly remove the minimum
- Use an unordered array priority queue
  - Turns into selection sort
- Use an ordered array
  - Turns into insertion sort
- What happens if we use a heap?

In place sorting

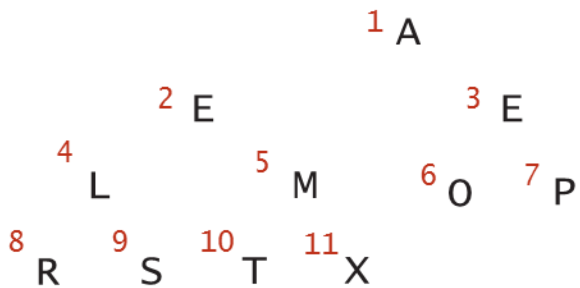
30



build a max-heap
(in place)



sorted result
(in place)



Heapsort

- Given an unordered array how can we convert it to an binary heap
- For simplicity we use maximum (reverse order sorting)
- Method 1:
  - Move down the array and use successive swims (promotions)
  - Items to left of scanning pointer are in order
  - Next item may be moved up the tree
  - O(N log N)

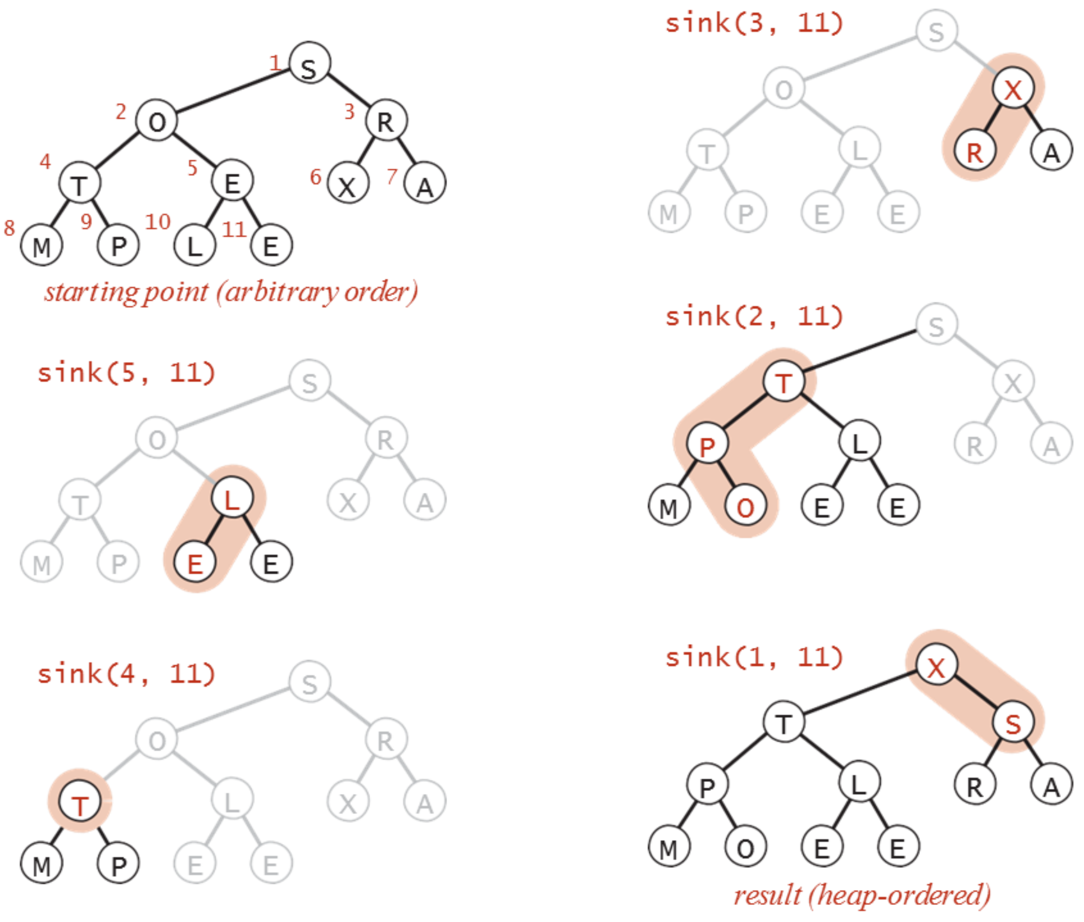
Heapsort (2)

- Method 2:
  - Build heap right to left
  - Consider parent of two ordered subheaps
  - Use sink to create an ordered larger subheap
  - Move one step to left (towards beginning)
  - Start halfway through

First pass

- Build heap bottom up

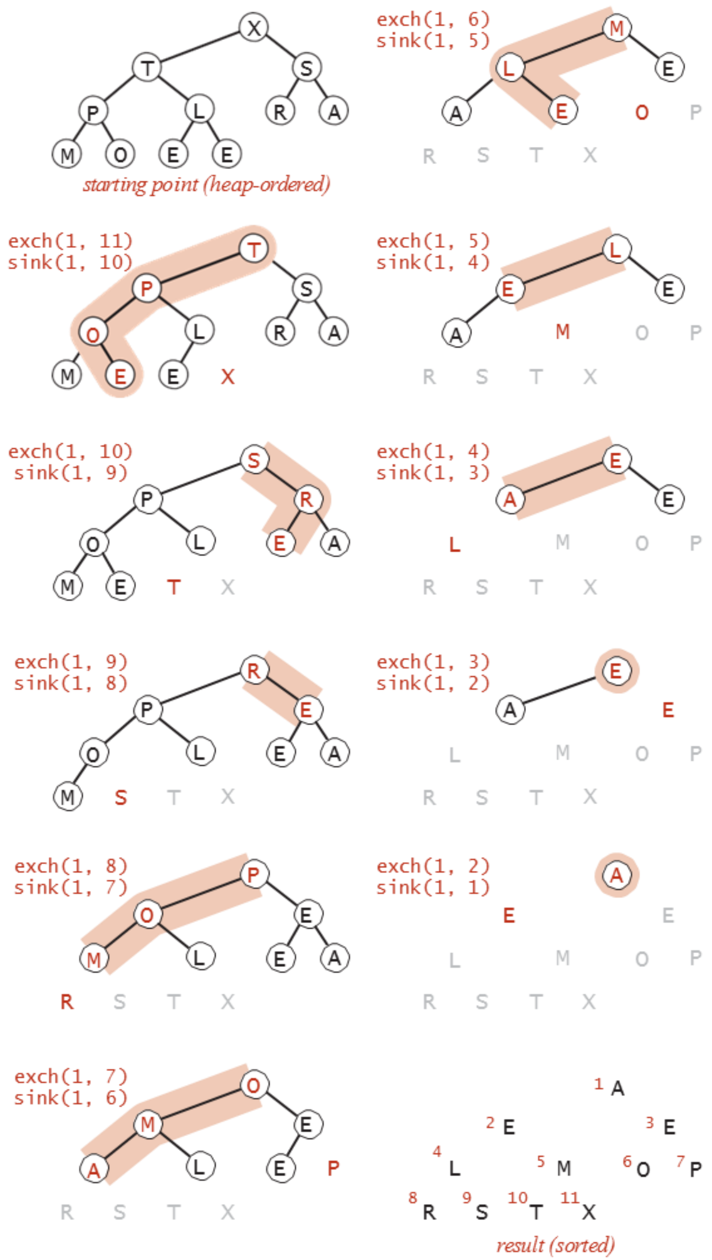
```
for (int k = N/2; k >= 1; k--)
   sink(a, k, N);
```



Second pass

- Sortdown
  - Move down
  - Exchange current item (largest) with last item
  - Use sink on the item

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



Java implementation

```
public class Heap
{
   public static 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);
      }
   }

   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 */   }
}
```

but convert from
1-based indexing to
0-base indexing

A trace

		a[i]											
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)

Properties of heapsort

- Heap building 2N compares and exchanges
- Sortdown uses 2N log N compares and exchanges

- Heapsort O(N log N) and in-place

- Used mainly in embedded systems where space is limited

- Disadvantages
  - Poor cache performance
  - Inner loop longer than quicksort

Applications of priority queues

- Just some examples

- The more obvious
  - Storing and sorting large amounts of data
  - Commercial companies, sorting by name, account number, order time, etc.

- Slightly less obvious
  - Continuously streaming massive input
  - Too large to store as is
  - Priority queue used to store, while concurrently stored information is analyzed, filtered and compactified.

Discrete Event Simulation

- Even less obvious application
- Some problems not emanable to analysis
- Modelling complex processes to understand bottlenecks
- Observing the macro from the micro
- Simulation
  - Time is discrete
  - Priority queue holds upcoming events ordered by increasing time
  - Next event at time t taken from queue and then processed.
    Typically causes one or more events at times t+X
    -These events added to priority queue.