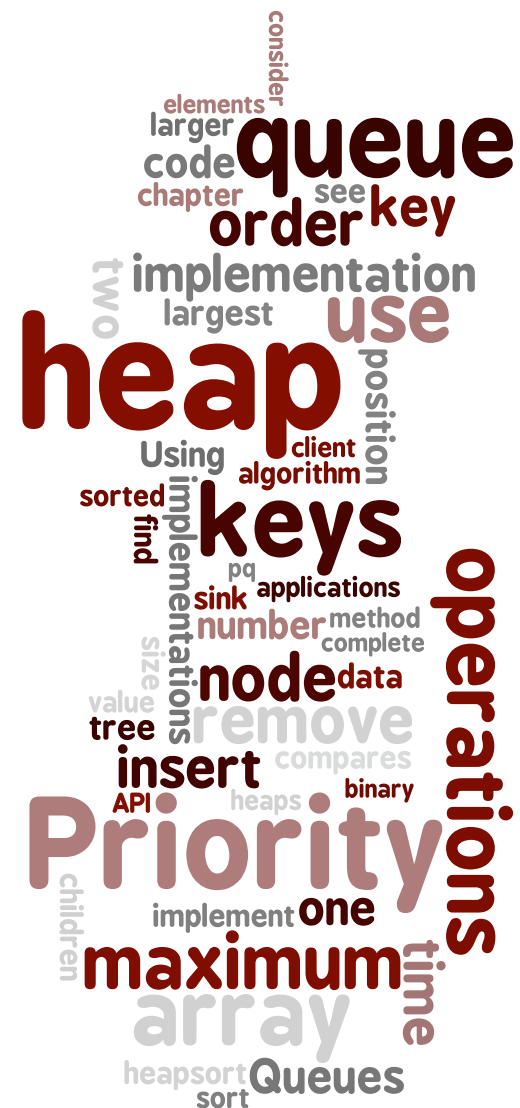


2.4 Priority Queues



- ▶ API
- ▶ elementary implementations
- ▶ binary heaps
- ▶ heapsort

Priority queue

Collections. Insert and delete items. Which item to delete?

Stack. Remove the item most recently added.

Queue. Remove the item least recently added.

Randomized queue. Remove a random item.

Priority queue. Remove the largest (or smallest) item.

<i>operation</i>	<i>argument</i>	<i>return value</i>
<i>insert</i>	P	
<i>insert</i>	Q	
<i>insert</i>	E	
<i>remove max</i>		Q
<i>insert</i>	X	
<i>insert</i>	A	
<i>insert</i>	M	
<i>remove max</i>		X
<i>insert</i>	P	
<i>insert</i>	L	
<i>insert</i>	E	
<i>remove max</i>		P

Priority queue API

Requirement. Generic items are Comparable.

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

MaxPQ()	<i>create a priority queue</i>
---------	--------------------------------

MaxPQ(maxN)	<i>create a priority queue of initial capacity maxN</i>
-------------	---------------------------------------------------------

void insert(Key v)	<i>insert a key into the priority queue</i>
--------------------	---------------------------------------------

Key max()	<i>return the largest key</i>
-----------	-------------------------------

Key delMax()	<i>return and remove the largest key</i>
--------------	------------------------------------------

boolean isEmpty()	<i>is the priority queue empty?</i>
-------------------	-------------------------------------

int size()	<i>number of entries in the priority queue</i>
------------	------------------------------------------------

API for a generic priority queue

Priority queue applications

- Event-driven simulation. [customers in a line, colliding particles]
- Numerical computation. [reducing roundoff error]
- Data compression. [Huffman codes]
- Graph searching. [Dijkstra's algorithm, Prim's algorithm]
- Computational number theory. [sum of powers]
- Artificial intelligence. [A* search]
- Statistics. [maintain largest M values in a sequence]
- Operating systems. [load balancing, interrupt handling]
- Discrete optimization. [bin packing, scheduling]
- Spam filtering. [Bayesian spam filter]

Generalizes: stack, queue, randomized queue.

Priority queue client example

Problem. Find the largest M items in a stream of N items.

- Fraud detection: isolate \$\$ transactions.
- File maintenance: find biggest files or directories.

Constraint. Not enough memory to store N items.

Solution. Use a min-oriented priority queue.

**cost of finding the largest M
in a stream of N items**

implementation	time	space
sort	$N \log N$	N
elementary PQ	$M N$	M
binary heap	$N \log M$	M
best in theory	N	M

Priority queue client example

Problem. Find the largest M items in a stream of N items.

- Fraud detection: isolate \$\$ transactions.
- File maintenance: find biggest files or directories.

Constraint. Not enough memory to store N items.

Solution. Use a min-oriented priority queue.

```
MinPQ<String> pq = new MinPQ<String>();

while (!StdIn.isEmpty())
{
    String s = StdIn.readString();
    pq.insert(s);
    if (pq.size() > M)
        pq.delMin();
}

while (!pq.isEmpty())
    System.out.println(pq.delMin());
```

**cost of finding the largest M
in a stream of N items**

implementation	time	space
sort	$N \log N$	N
elementary PQ	$M N$	M
binary heap	$N \log M$	M
best in theory	N	M

- ▶ API
- ▶ **elementary implementations**
- ▶ binary heaps
- ▶ heapsort

Priority queue: unordered and ordered array implementation

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

A sequence of operations on a priority queue

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];
    }
}
```

no generic
array creation

less() and exch()
as for sorting

Priority queue elementary implementations

Challenge. Implement **all** operations efficiently.

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

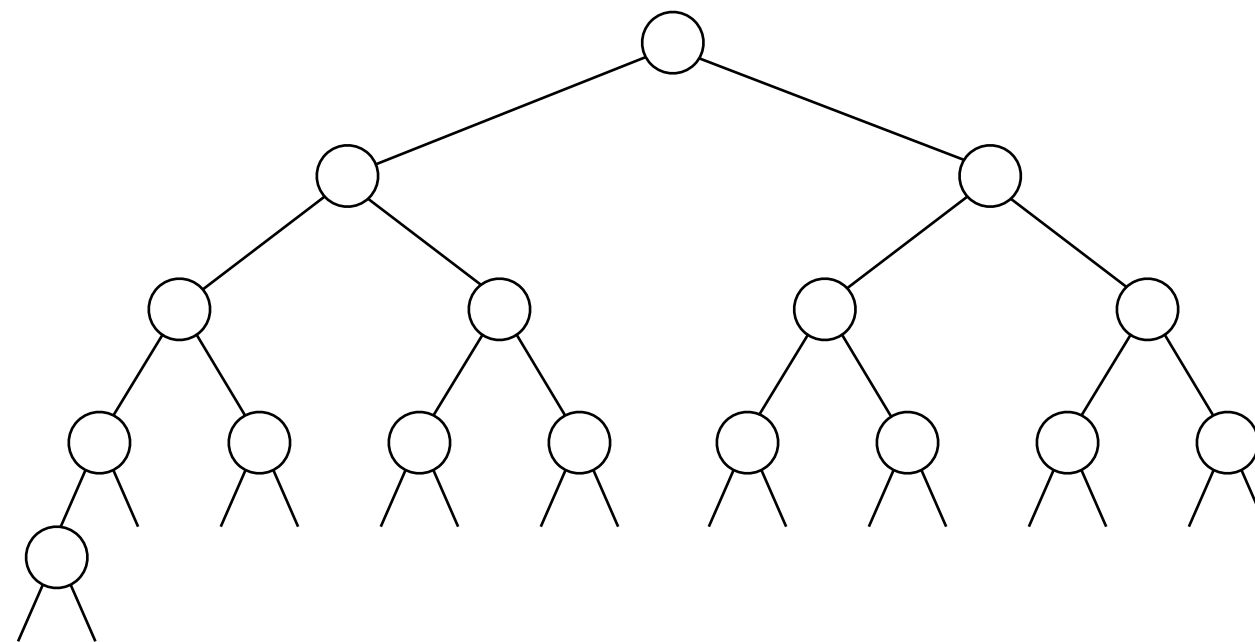
implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
goal	$\log N$	$\log N$	$\log N$

- ▶ API
- ▶ elementary implementations
- ▶ **binary heaps**
- ▶ heapsort

Binary tree

Binary tree. Empty **or** node with links to left and right binary trees.

Complete tree. Perfectly balanced, except for bottom level.



complete tree with $N = 16$ nodes (height = 5)

Property. Height of complete tree with N nodes is $1 + \lfloor \lg N \rfloor$.

Pf. Height only increases when N is a power of 2.

A complete binary tree in nature



Hyphaene Compressa - Doum Palm

© Shlomit Pinter

Binary heap representations

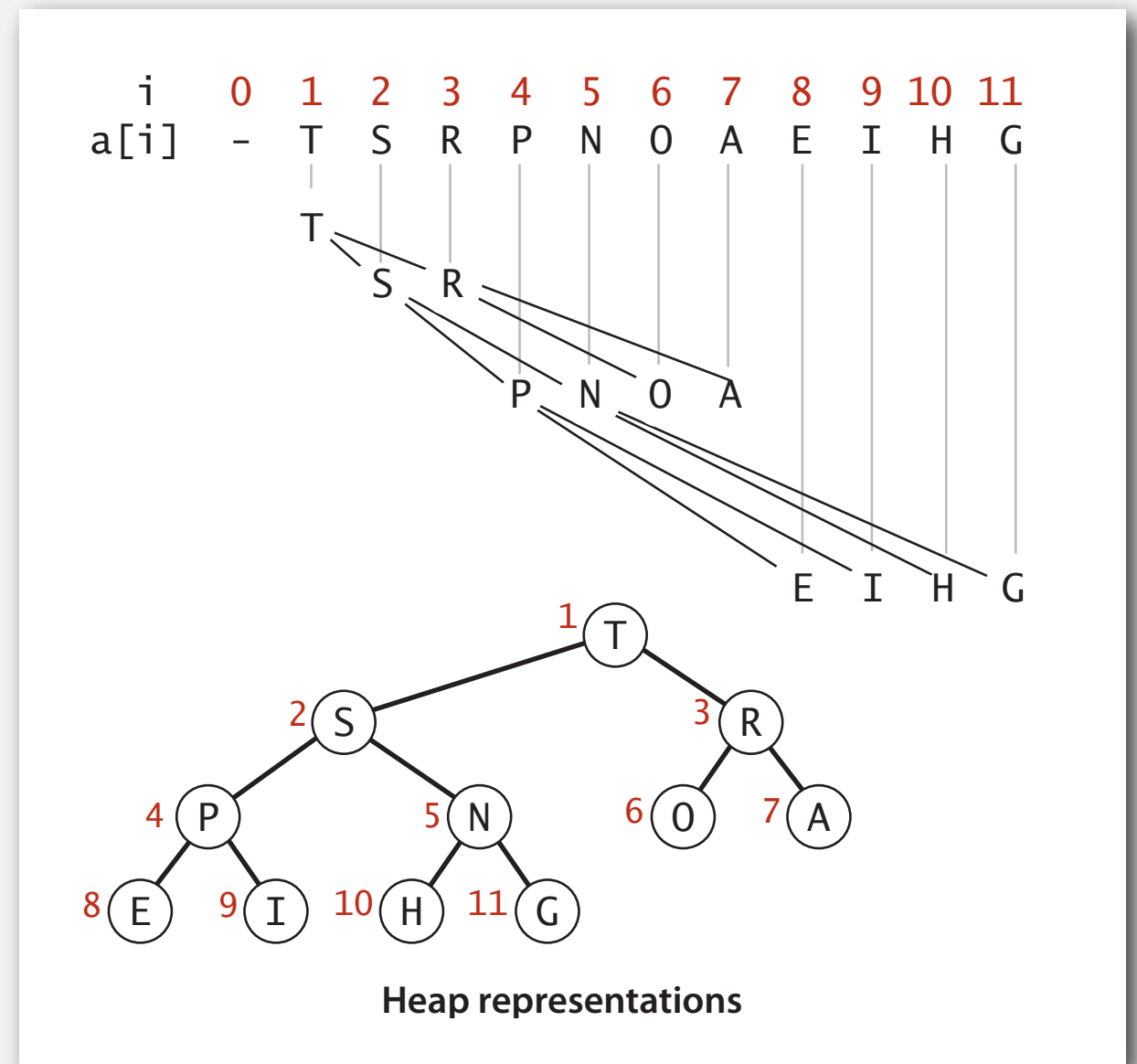
Binary heap. Array representation of a heap-ordered complete binary tree.

Heap-ordered binary tree.

- Keys in nodes.
- No smaller than children's keys.

Array representation.

- Take nodes in **level** order.
- No explicit links needed!

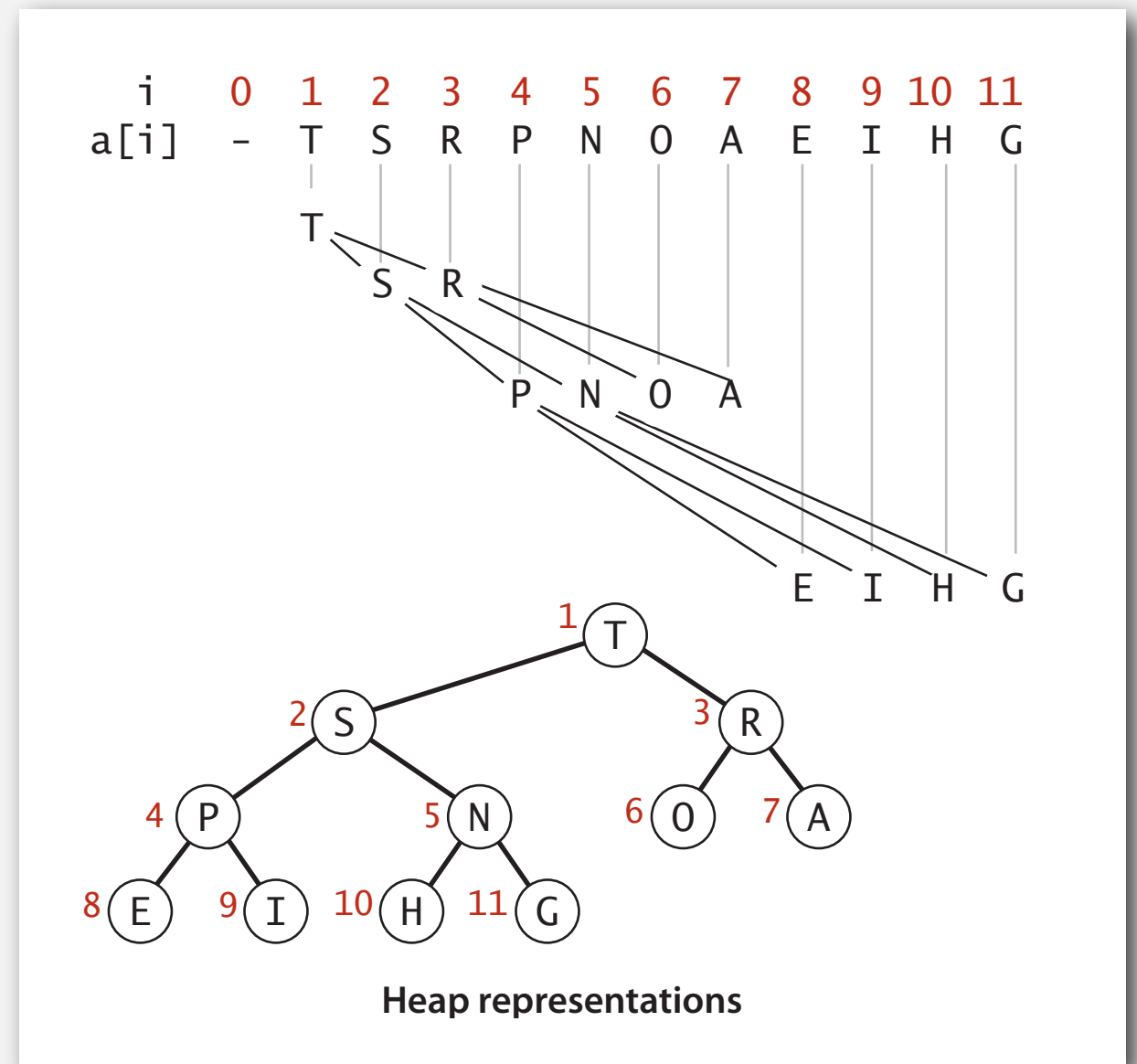


Binary heap properties

Proposition. Largest key is $a[1]$, which is root of binary tree.

Proposition. Can use array indices to move through tree. indices start at 1

- Parent of node at k is at $k/2$.
- Children of node at k are at $2k$ and $2k+1$.



Promotion in a heap

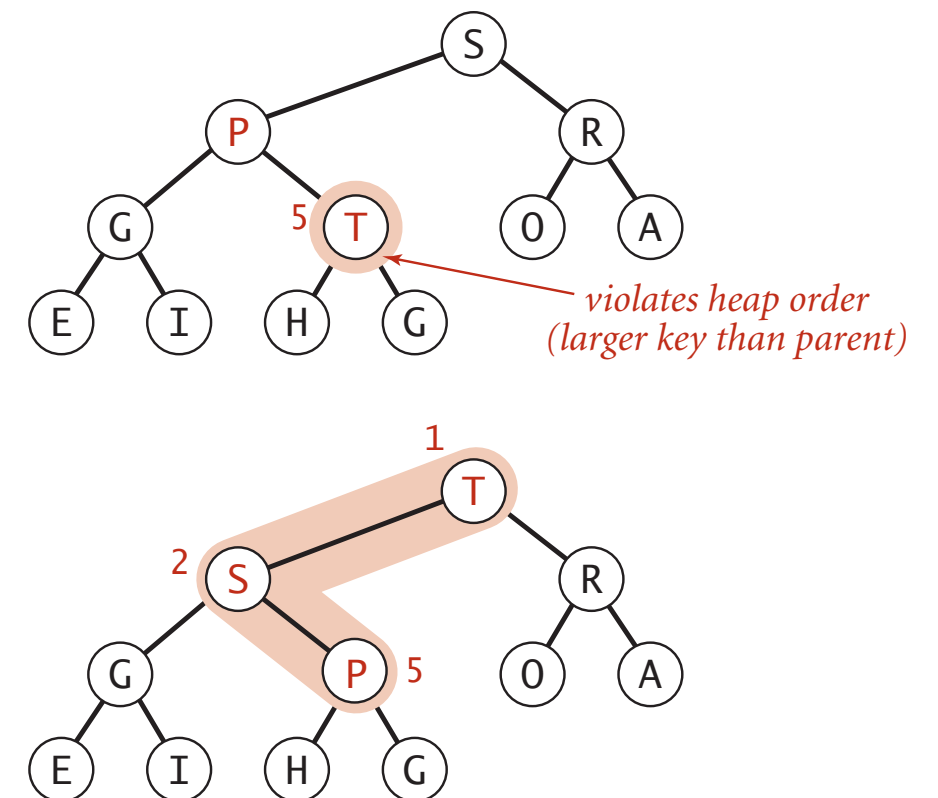
Scenario. Node's key becomes **larger** key than its parent's key.

To eliminate the violation:

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

```
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



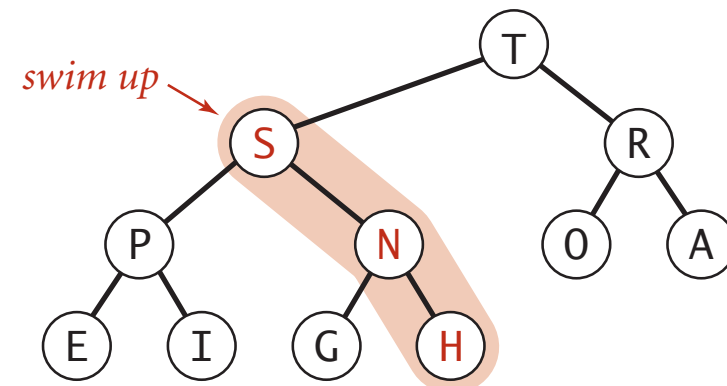
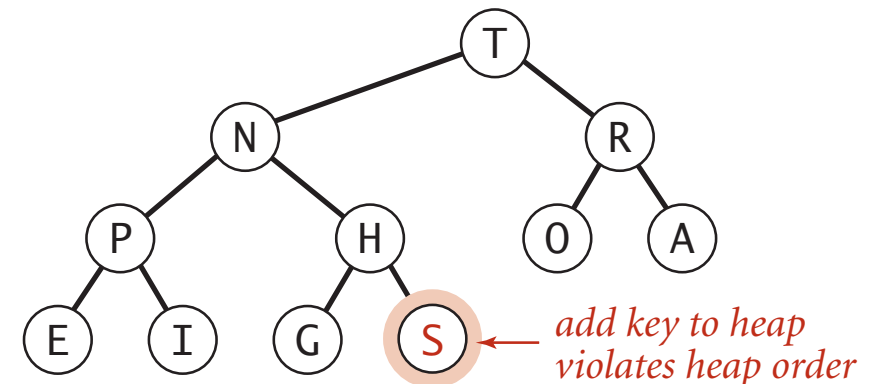
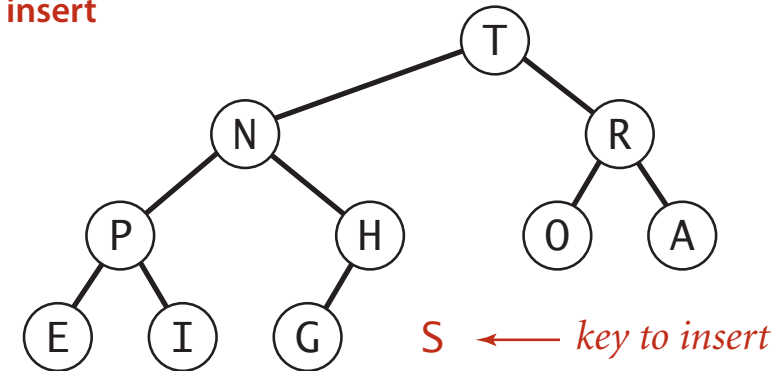
Insertion in a heap

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

Cost. At most $\lg N$ compares.

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

insert



Demotion in a heap

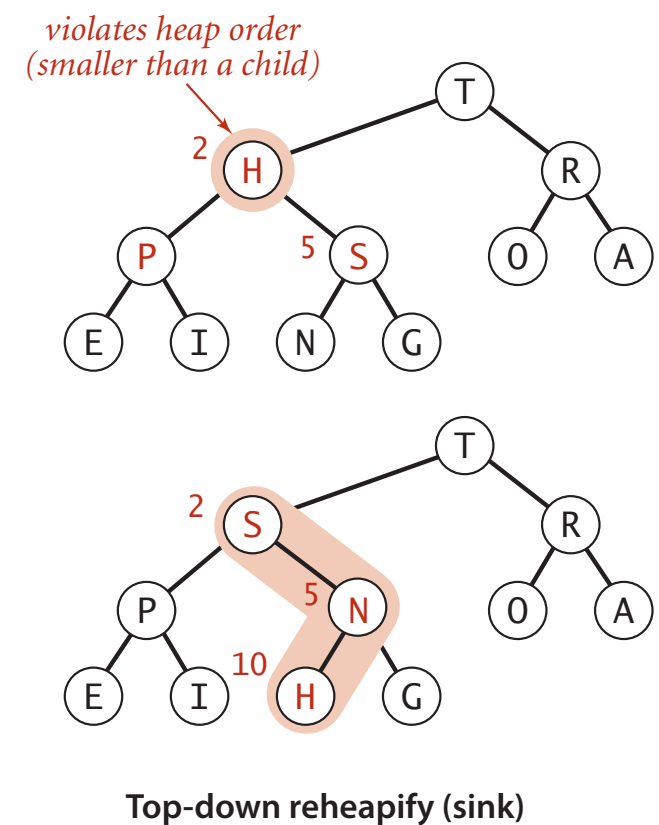
Scenario. Node's key becomes **smaller** than one (or both) of its children's keys.

To eliminate the violation:

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

```
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

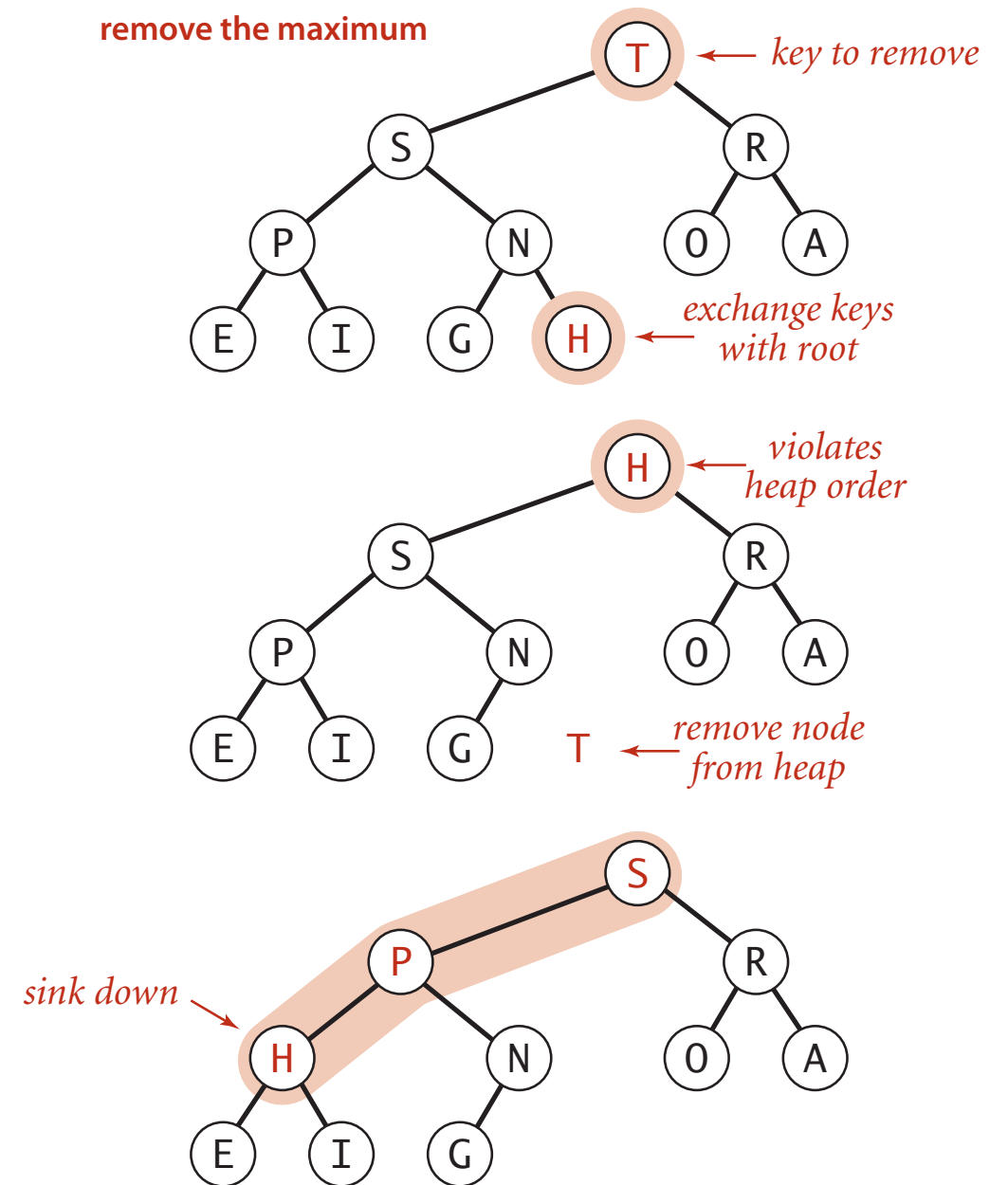


Delete the maximum in a heap

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

Cost. At most $2 \lg N$ compares.

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

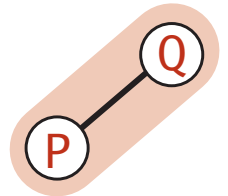


Heap operations

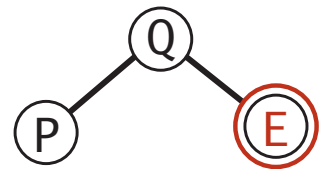
insert P



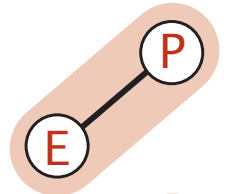
insert Q



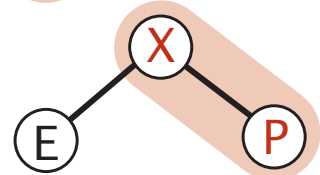
insert E



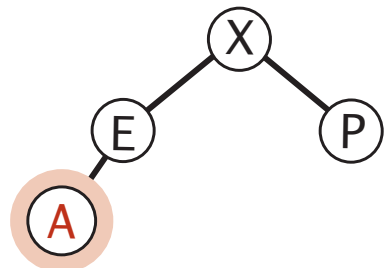
remove max (Q)



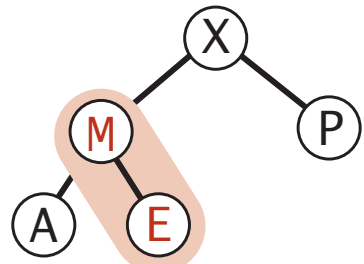
insert X



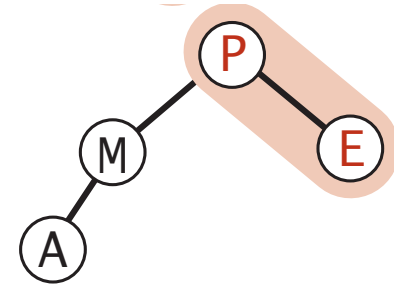
insert A



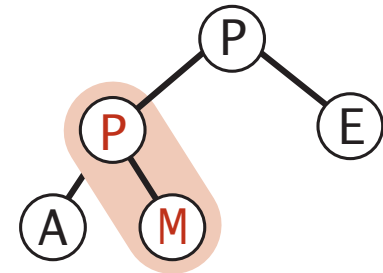
insert M



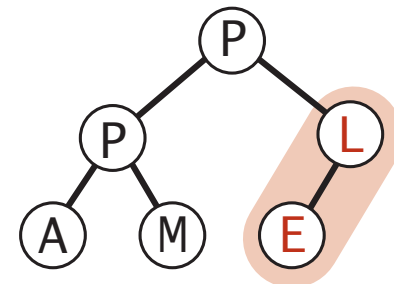
remove max (X)



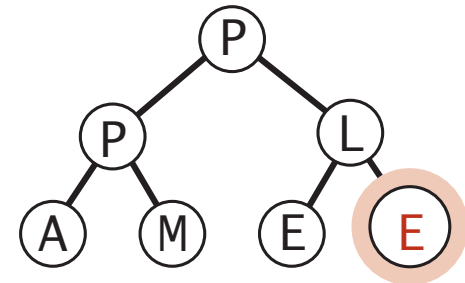
insert P



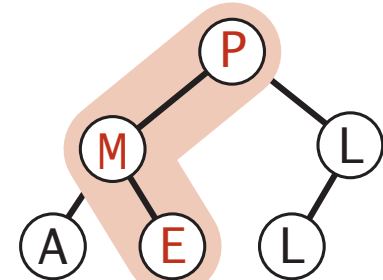
insert L



insert E



remove max (P)



Binary heap: Java 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)
    {    /* see previous code */    }
    public Key delMax()
    {    /* see previous code */    }
```

← PQ ops

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

← heap helper functions

```
    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;    }
}
```

← array helper functions

Priority queues implementation cost summary

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
binary heap	$\log N$	$\log N$	1
d-ary heap	$\log_d N$	$d \log_d N$	1
Fibonacci	1	$\log N^\dagger$	1

\dagger amortized

Hopeless challenge. Make all operations constant time.


Q. Why hopeless?

Binary heap considerations

Minimum-oriented priority queue.

- Replace `less()` with `greater()`.
- Implement `greater()`.


Dynamic-array resizing.

- Add no-arg constructor.
- Apply repeated doubling and shrinking.  leads to log N amortized time per op

Immutability of keys.

- Assumption: client does not change keys while they're on the PQ.
- Best practice: use immutable keys.

Other operations.

- Remove an arbitrary item.
 - Change the priority of an item.
- 
- easy to implement with `sink()` and `swim()` [stay tuned]

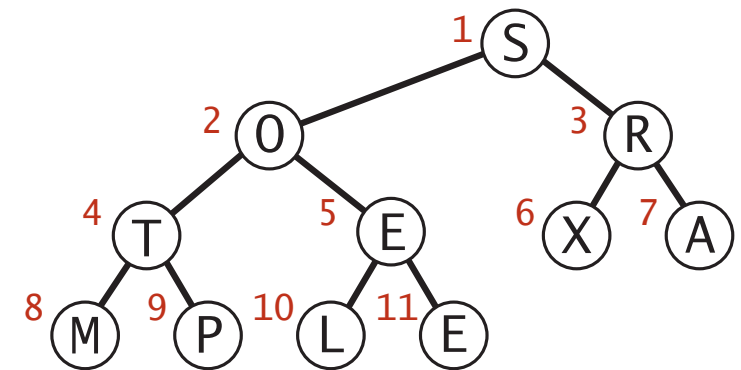
- ▶ API
- ▶ elementary implementations
- ▶ binary heaps
- ▶ **heapsort**

Heapsort

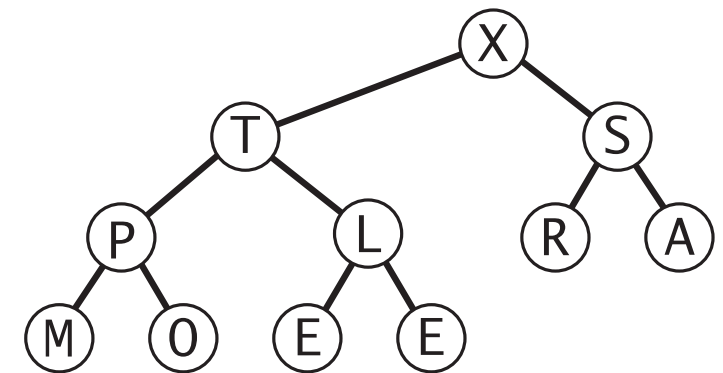
Basic plan for in-place sort.

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

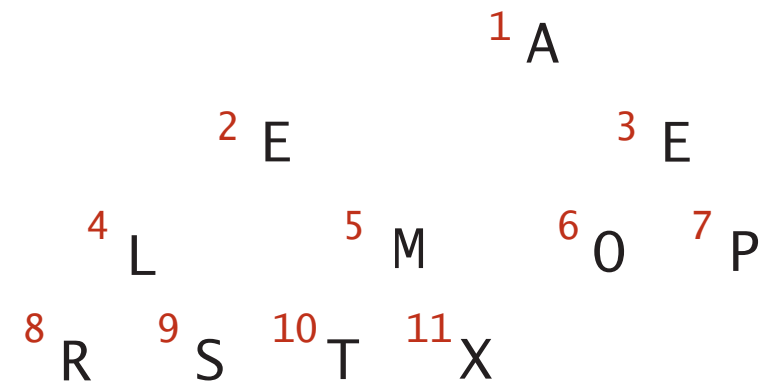
start with array of keys
in arbitrary order



build a max-heap
(in place)



sorted result
(in place)

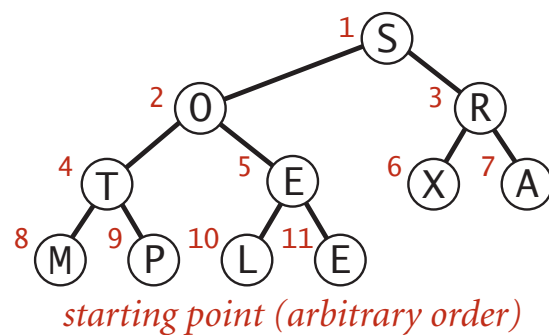


Heapsort: heap construction

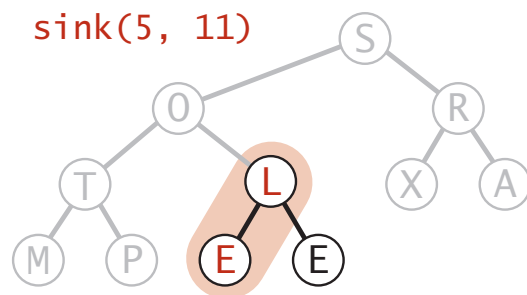
First pass. Build heap using bottom-up method.

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

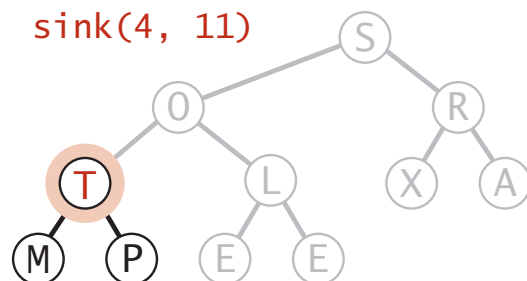
heap construction



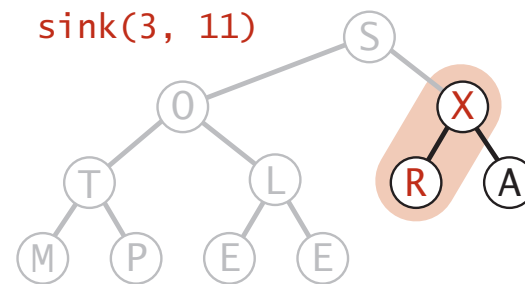
sink(5, 11)



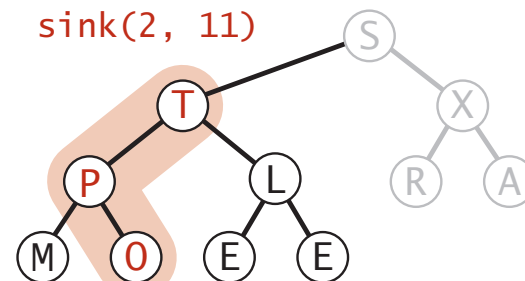
sink(4, 11)



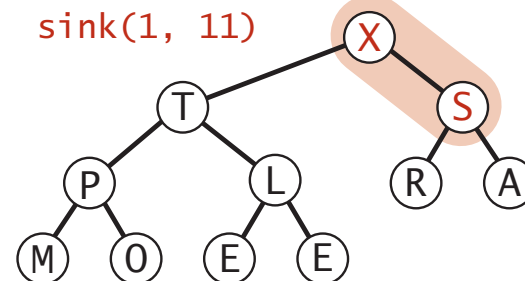
sink(3, 11)



sink(2, 11)



sink(1, 11)



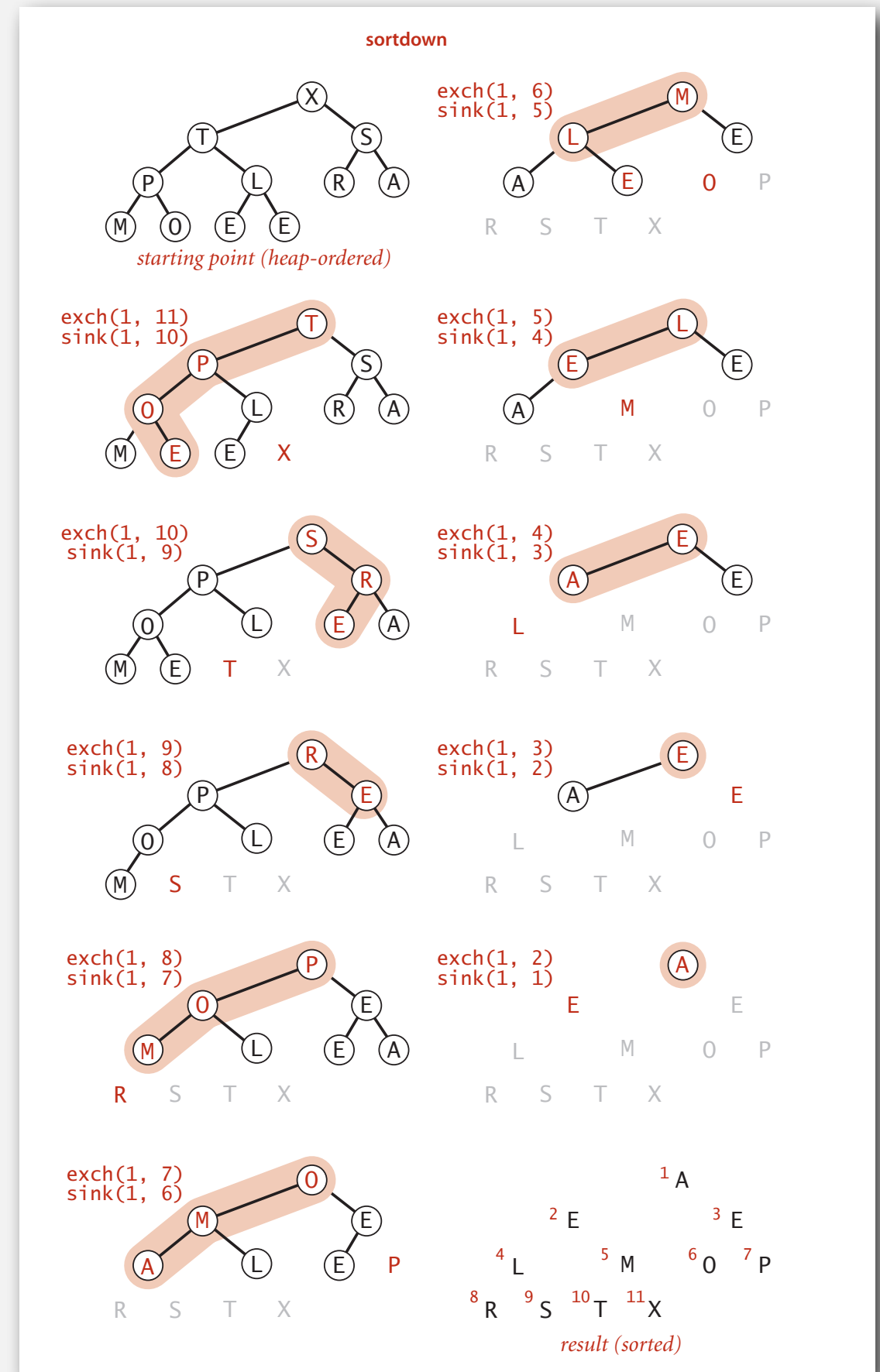
result (heap-ordered)

Heapsort: sortdown

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: Java implementation

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

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

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

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

but use 1-based indexing



Heapsort: trace


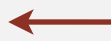
		a[i]											
N	k	0	1	2	3	4	5	6	7	8	9	10	11
<i>initial values</i>			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	L	X	A	M	P	E	E
11	4		S	O	R	T	L	X	A	M	P	E	E
11	3		S	O	X	T	L	R	A	M	P	E	E
11	2		S	T	X	P	L	R	A	M	O	E	E
11	1		X	T	S	P	L	R	A	M	O	E	E
<i>heap-ordered</i>			X	T	S	P	L	R	A	M	O	E	E
10	1		T	P	S	O	L	R	A	M	E	E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

Heapsort trace (array contents just after each sink)

Heapsort: mathematical analysis

Proposition. Heapsort uses at most $2 N \lg N$ compares and exchanges.

Significance. In-place sorting algorithm with $N \log N$ worst-case.

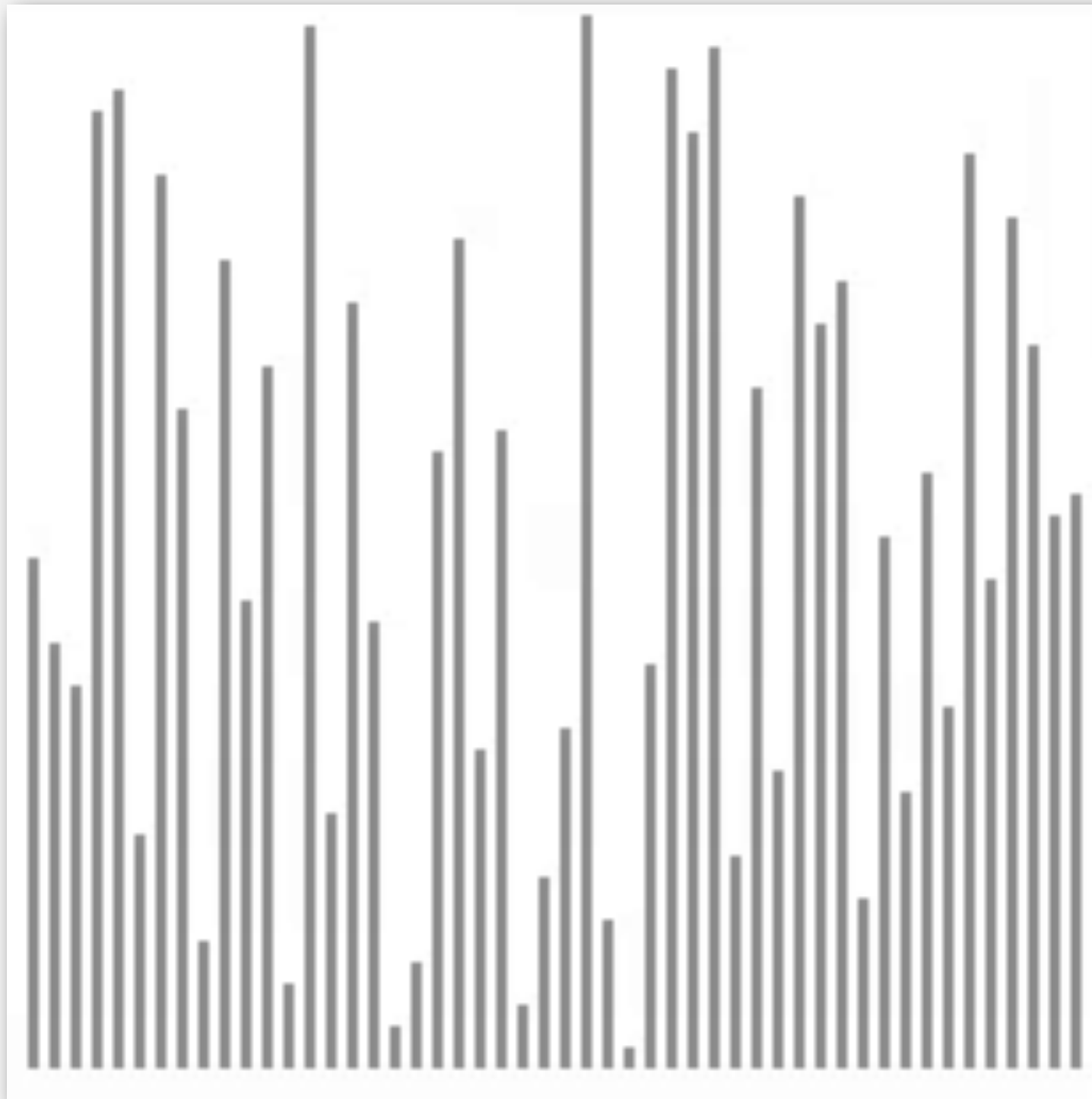
- Mergesort: no, linear extra space.  in-place merge possible, not practical
- Quicksort: no, quadratic time in worst case.  $N \log N$ worst-case quicksort possible, not practical
- Heapsort: yes!

Bottom line. Heapsort is optimal for both time and space, **but**:

- Inner loop longer than quicksort's.
- Makes poor use of cache memory.
- Not stable.

Heapsort animation

50 random elements

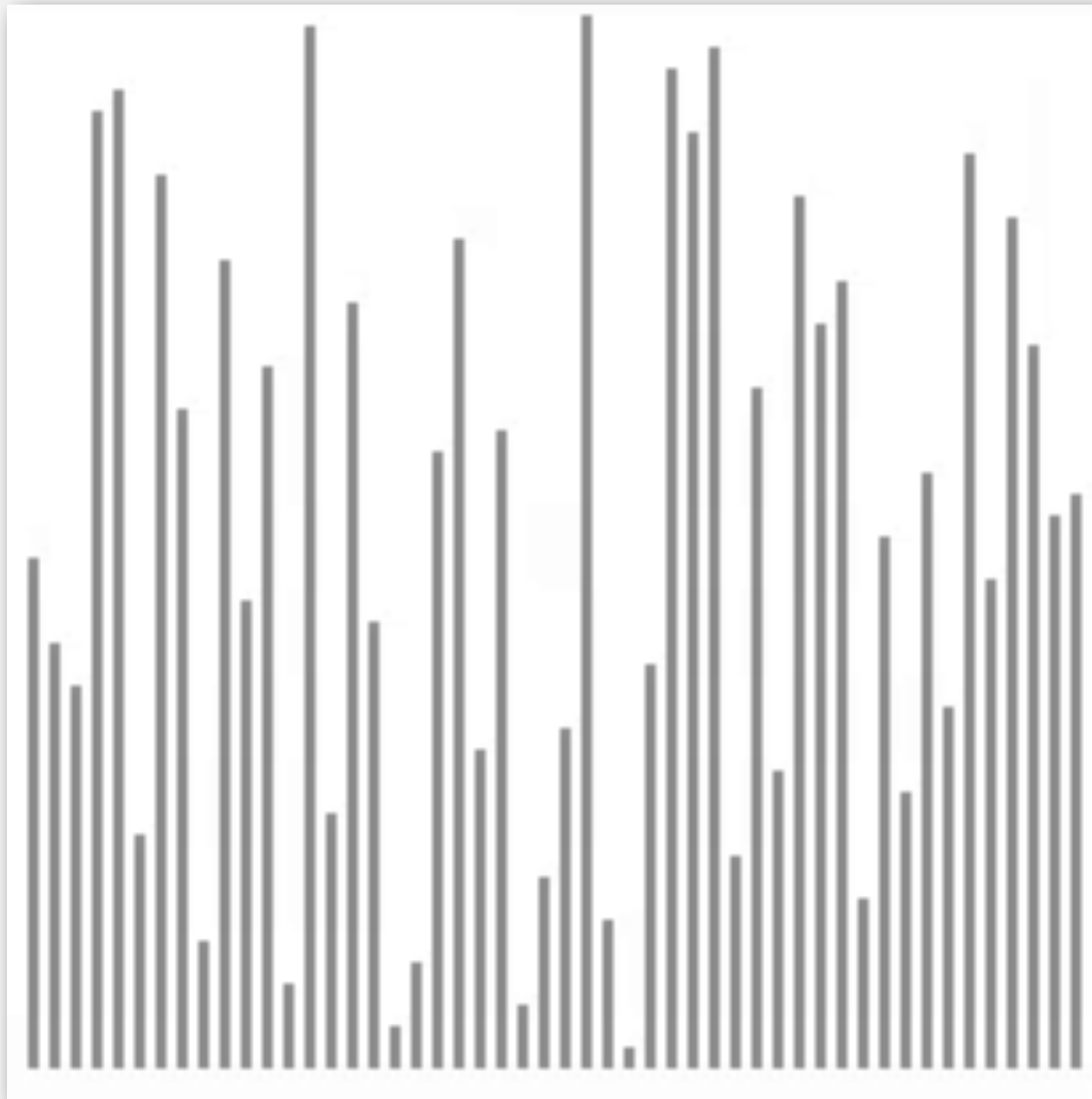


▲ algorithm position
— in order
— not in order

<https://www.cs.purdue.edu/homes/cs251/slides/media/heap-sort.mov>

Heapsort animation

50 random elements



▲ algorithm position
— in order
— not in order

<https://www.cs.purdue.edu/homes/cs251/slides/media/heap-sort.mov>

Sorting algorithms: summary

	inplace?	stable?	worst	average	best	remarks
selection	x		$N^2 / 2$	$N^2 / 2$	$N^2 / 2$	N exchanges
insertion	x	x	$N^2 / 2$	$N^2 / 4$	N	use for small N or partially ordered
shell	x		?	?	N	tight code, subquadratic
quick	x		$N^2 / 2$	$2 N \ln N$	$N \lg N$	$N \log N$ probabilistic guarantee fastest in practice
3-way quick	x		$N^2 / 2$	$2 N \ln N$	N	improves quicksort in presence of duplicate keys
merge		x	$N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee, stable
heap	x		$2 N \lg N$	$2 N \lg N$	$N \lg N$	$N \log N$ guarantee, in-place
???	x	x	$N \lg N$	$N \lg N$	$N \lg N$	holy sorting grail