https://algs4.cs.princeton.edu

2.4 Priority Queues

- *APIs*
- *elementary implementations*
- *binary heaps*
- *heapsort*
- *event-driven simulation* ← see videos

https://algs4.cs.princeton.edu

2.4 Priority Queues

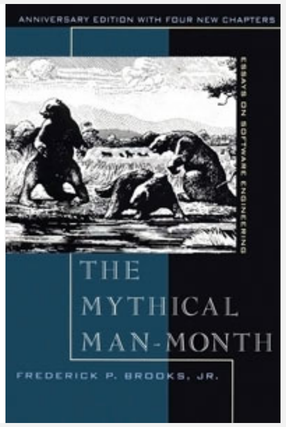
- *APIs*
- *elementary implementations*
- *binary heaps*
- *heapsort*
- *event-driven simulation*

Collections

A collection is a data type that stores a group of items.

data type	core operations	data structure
stack	Push, Pop	*linked list* *resizing array*
queue	Enqueue, Dequeue	
priority queue	Insert, Delete-Max	*binary heap*
symbol table	Put, Get, Delete	*binary search tree* *hash table*
set	Add, Contains, Delete	

" Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious." — *Fred Brooks*

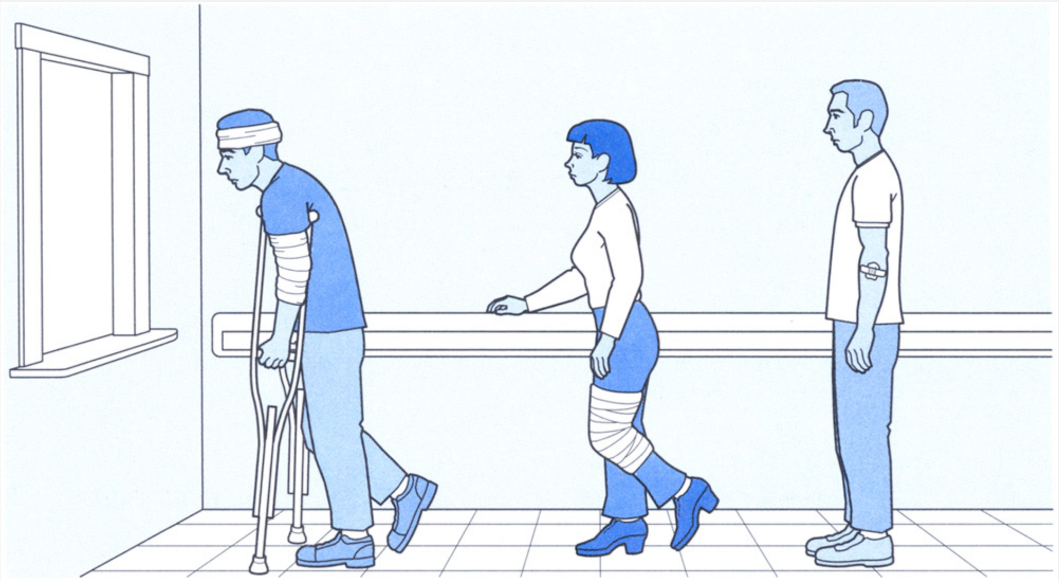


Priority queue

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.
Generalizes: stack, queue, randomized queue.



triage in an emergency room
(priority = urgency of wound/illness)

operation	*argument*	*return value*
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

Max-oriented priority queue API

Requirement. Must insert keys of the same (generic) type; moreover, keys must be `Comparable`.

"bounded type parameter"

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

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

Note. Duplicate keys allowed; `delMax()` picks any maximum key.

Min-oriented priority queue API

Analogous to `MaxPQ`.

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

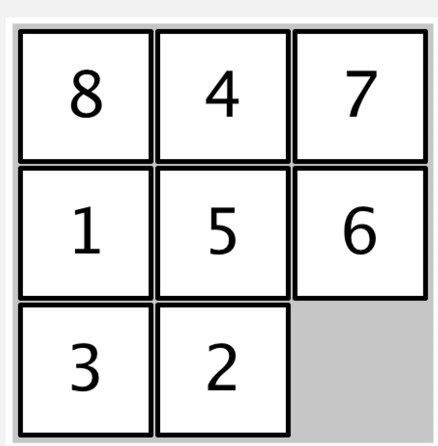
	`MinPQ()`	*create an empty priority queue*
`void`	`insert(Key v)`	*insert a key*
`Key`	`delMin()`	*return and remove a smallest key*
`boolean`	`isEmpty()`	*is the priority queue empty?*
`Key`	`min()`	*return a smallest key*
`int`	`size()`	*number of entries in the priority queue*

Warmup client. Sort a stream of integers from standard input.

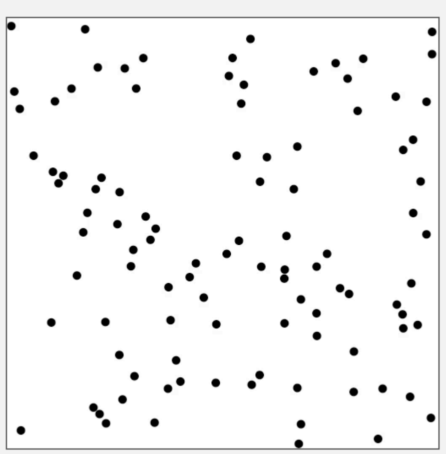
Priority queue: applications

- Event-driven simulation. [customers in a line, colliding particles]
- Discrete optimization. [bin packing, scheduling]
- Artificial intelligence. [A* search]
- Computer networks. [web cache]
- Data compression. [Huffman codes]
- Operating systems. [load balancing, interrupt handling]
- Graph searching. [Dijkstra's algorithm, Prim's algorithm]
- Number theory. [sum of powers]
- Spam filtering. [Bayesian spam filter]
- Statistics. [online median in data stream]

priority = length of best known path



priority = "distance" to goal board



priority = event time

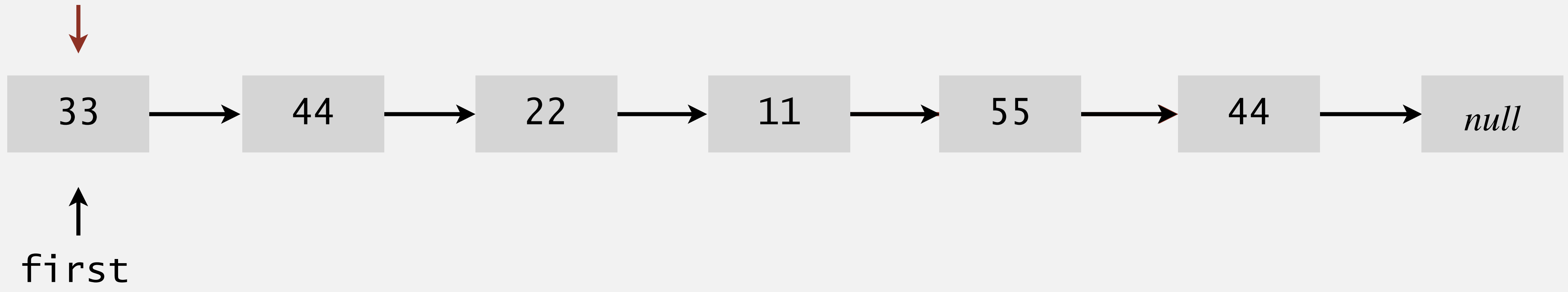
https://algs4.cs.princeton.edu

2.4 Priority Queues

- APIs
- *elementary implementations*
- binary heaps
- heapsort
- event-driven simulation

Priority queue: elementary implementations

Unordered list. Store keys in a linked list.



Performance. INSERT takes $\Theta(1)$ time; DELETE-MAX takes $\Theta(n)$ time.

Priority queue: elementary implementations

Ordered array. Store keys in an array in ascending (or descending) order.

	0	1	2	3	4	5	6	7	8	9	10
a[]	11	22	33	33	44	55	99				
								n			

ordered array implementation of a MaxPQ

Priority queues: quiz 1

What are the worst-case running times for INSERT and DELETE-MAX, respectively, for a MaxPQ implemented with an ordered array?

ignore array resizing

A. $\Theta(1)$ and $\Theta(n)$

B. $\Theta(1)$ and $\Theta(\log n)$

C. $\Theta(\log n)$ and $\Theta(1)$

D. $\Theta(n)$ and $\Theta(1)$

	0	1	2	3	4	5	6	7	8	9	10
a[]	11	22	33	33	44	55	99				
								n			

ordered array implementation of a MaxPQ

Priority queue: implementations cost summary

Elementary implementations. Either INSERT or DELETE-MAX takes $\Theta(n)$ time.

implementation	INSERT	DELETE–MAX	MAX
unordered list	1	n	n
ordered array	n	1	1
goal	$\boxed{\log n}$	$\boxed{\log n}$	$\log n$

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

Challenge. Implement both core operations efficiently.

Solution. "Somewhat-ordered" array.

https://algs4.cs.princeton.edu

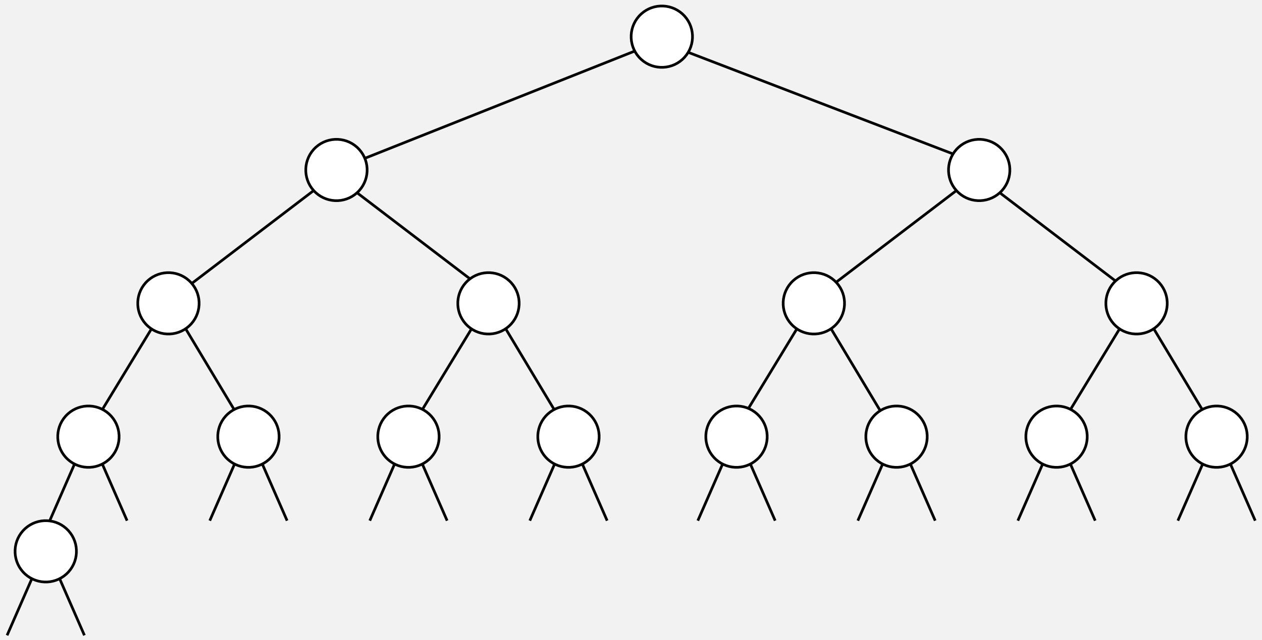
2.4 Priority Queues

- API
- elementary implementations
- *binary heaps*
- heapsort
- event-driven simulation

Complete binary tree

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

Complete tree. Every level (except possibly the last) is completely filled; the last level is filled from left to right.



complete binary tree with n = 16 nodes (height = 4)

Property. Height of complete binary tree with n nodes is $\lfloor \lg n \rfloor$.

Pf. Height increases (by 1) only when n is a power of 2.

A complete binary tree in nature

Binary heap: representation

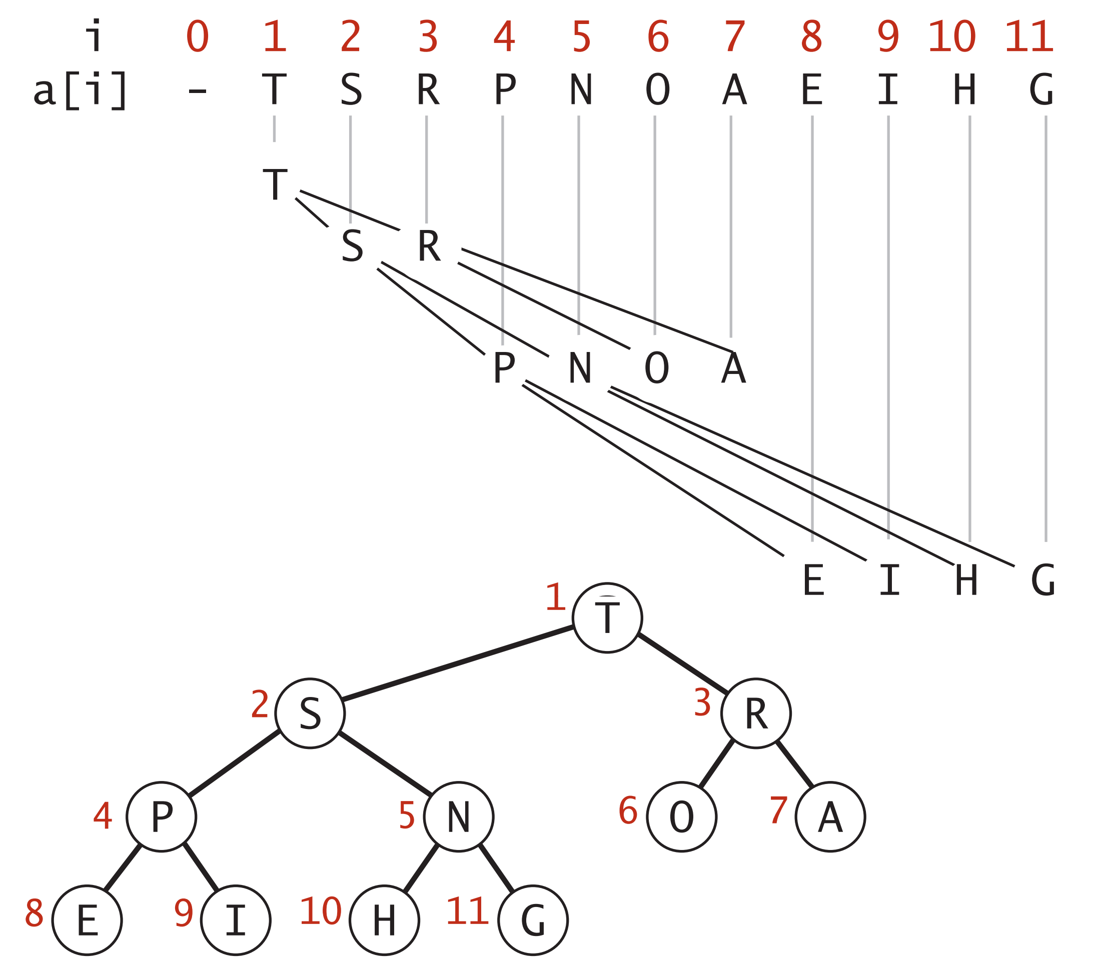
Binary heap. Array representation of a heap-ordered complete binary tree.

Heap-ordered tree.

- Keys in nodes.
- Child's key no larger than parent's key.

Array representation.

- Indices start at 1.
- Take nodes in level order.
- No explicit links!



Heap representations

Priority queues: quiz 2

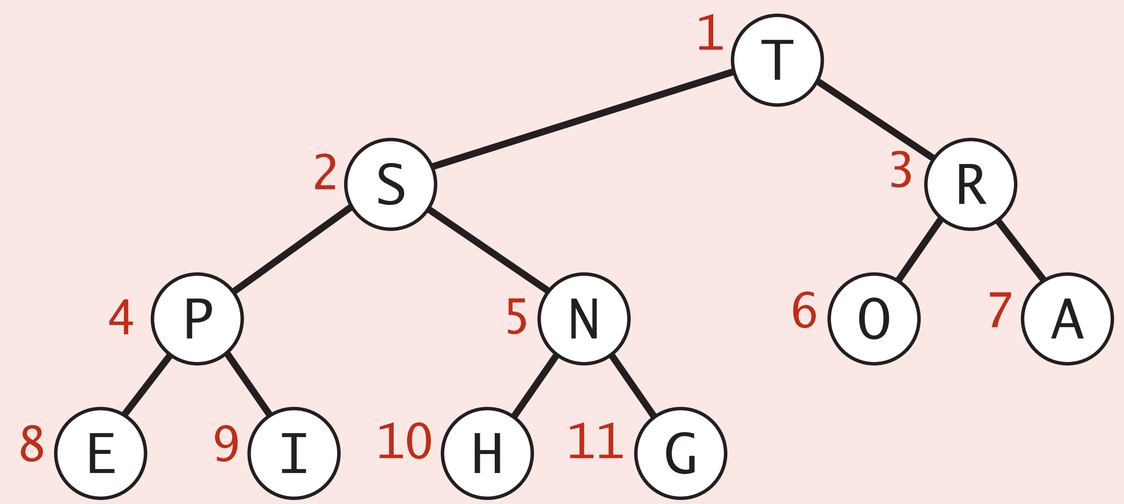
Consider the node at index k in a binary heap. Which Java expression gives the index of its parent?

A. `(k - 1) / 2`

B. `k / 2`

C. `(k + 1) / 2`

D. `2 * k`



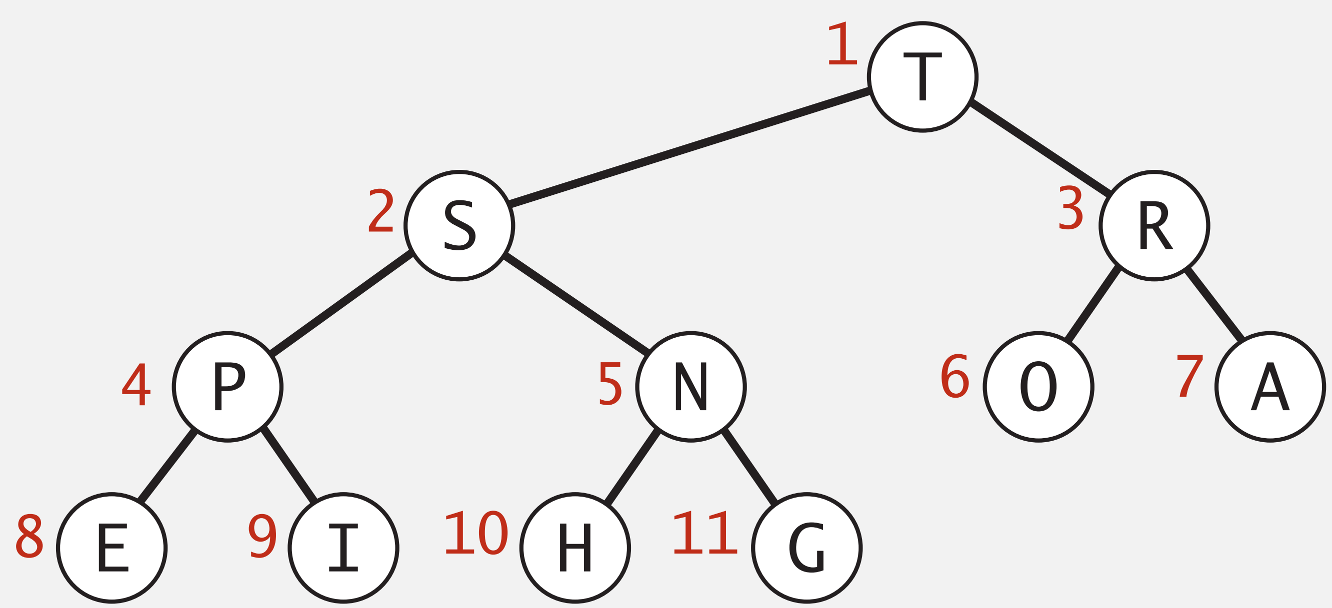
i	0	1	2	3	4	5	6	7	8	9	10	11
a[i]	-	T	S	R	P	N	O	A	E	I	H	G

Binary heap: properties

Proposition. Largest key is at index `1`, which is root of binary tree.

Proposition. Can use array indices to move through tree.

- Parent of key at index `k` is at index `k/2`.
- Children of key at index `k` are at indices `2*k` and `2*k + 1`.

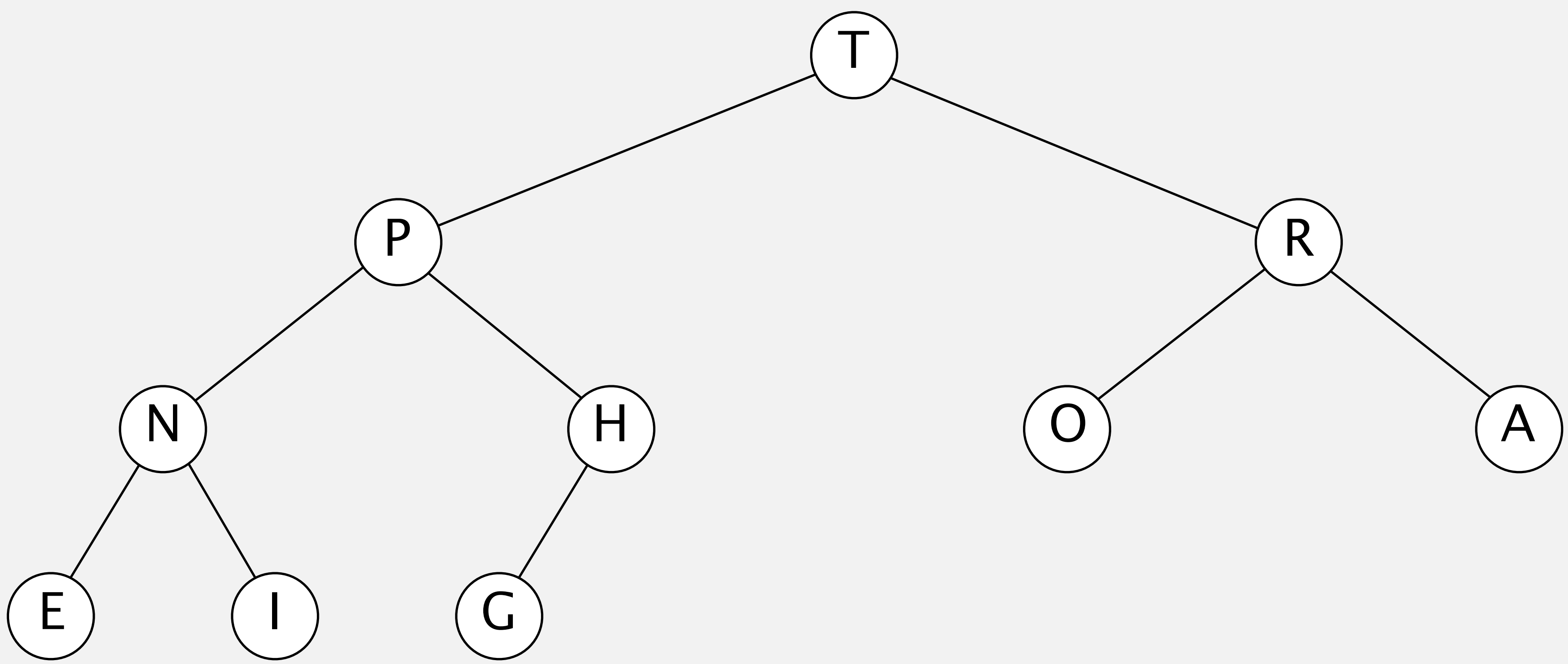


Binary heap demo

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

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered



T	P	R	N	H	O	A	E	I	G	

Binary heap: promotion

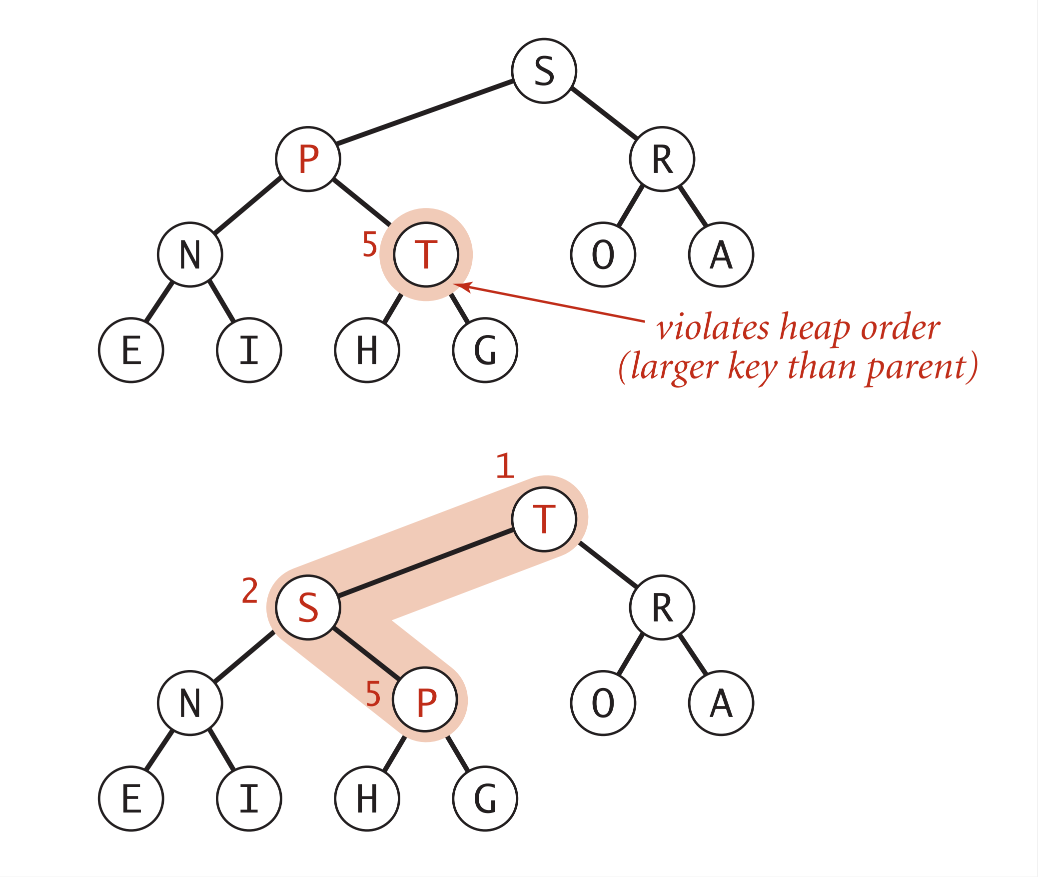
Scenario. A key becomes larger than its parent's key.

To eliminate the violation:

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

```
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



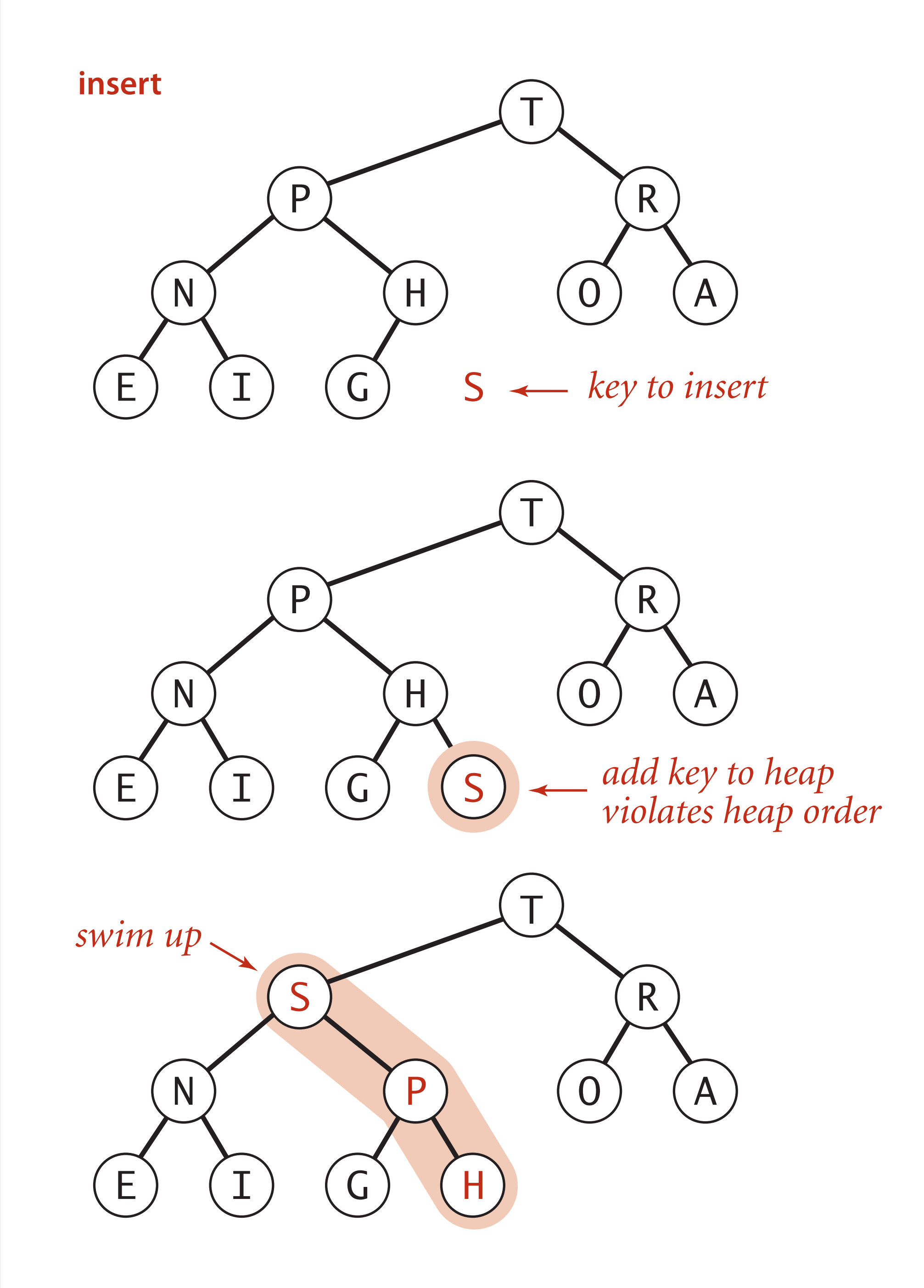
Peter principle. Node promoted to level of incompetence.

Binary heap: insertion

Insert. Add node at end in bottom level; then, swim it up.

Cost. At most $1 + \lg n$ compares.

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



Binary heap: demotion

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

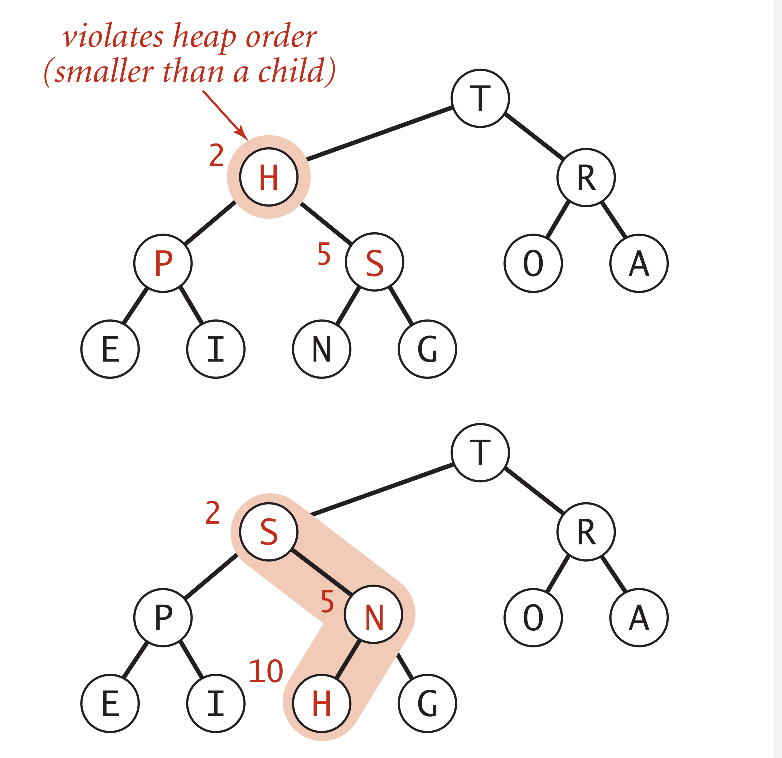
To eliminate the violation:

why not smaller child?

- Exchange key in parent 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 at 2*k and 2*k+1



Top-down reheapify (sink)

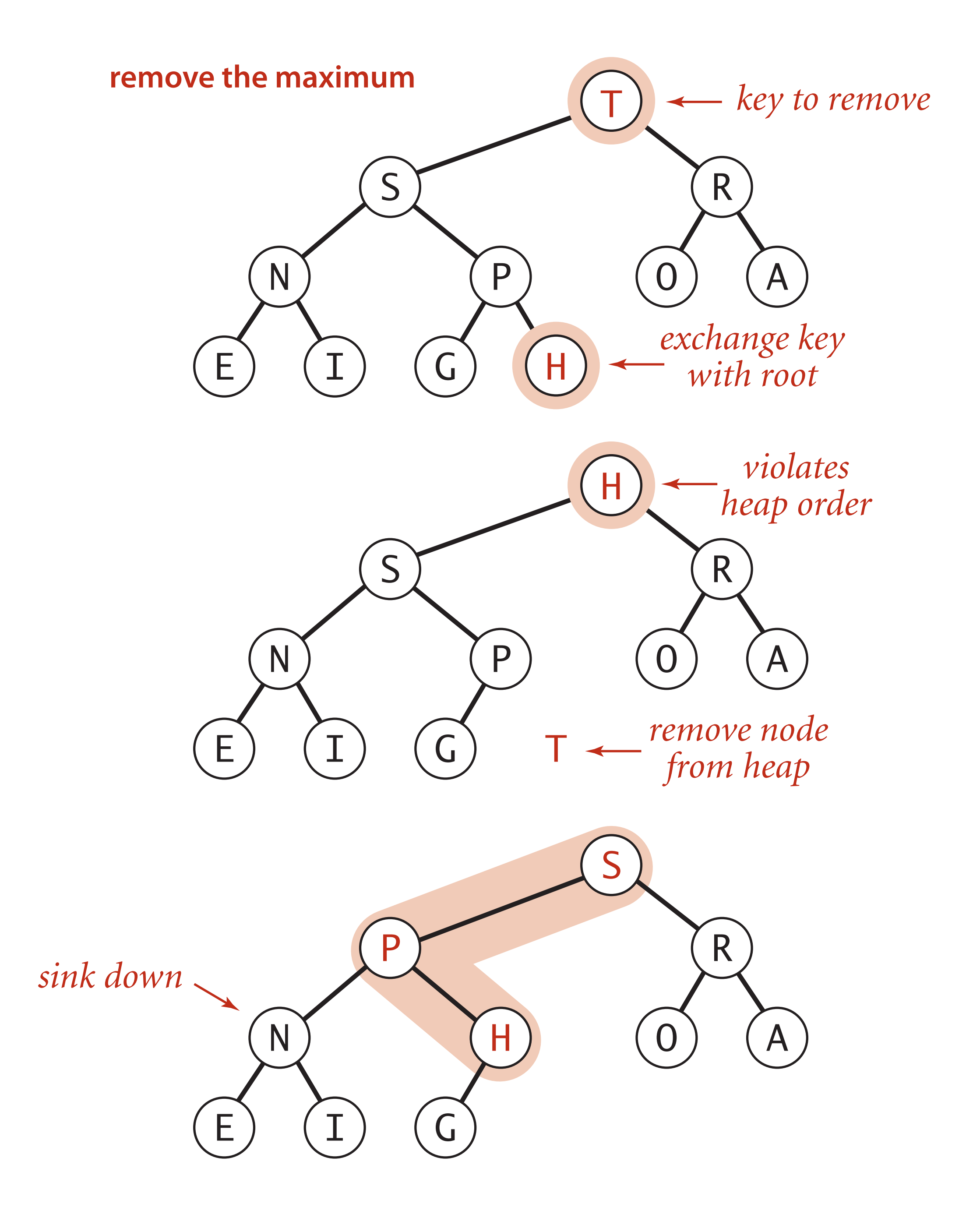
Power struggle. Better subordinate promoted.

Binary heap: delete the maximum

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;
}
```



Binary heap: Java implementation

```
public class MaxPQ<Key extends Comparable<Key>>
{
   private Key[] a;
   private int n;

   public MaxPQ(int capacity)
   {  a = (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

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

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

fixed capacity (for simplicity)

PQ ops

heap helper functions

array helper functions

https://algs4.cs.princeton.edu/24pq/MaxPQ.java.html

Priority queue: implementations cost summary

Goal. Implement both INSERT and DELETE-MAX in $\Theta(\log n)$ time.

implementation	INSERT	DELETE-MAX	MAX
unordered list	1	n	n
ordered array	n	1	1
goal	$\log n$	$\log n$	1

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

Binary heap: considerations

Underflow and overflow.

- Underflow: throw exception if deleting from empty PQ.
- Overflow: add no-arg constructor and use resizing array.

leads to $\Theta(\log n)$
amortized time per op
(how to make worst case?)

Minimum-oriented priority queue.

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

Other operations.

- Remove an arbitrary item.
- Change the priority of an item.

can implement efficiently with `sink()` and `swim()`
[stay tuned for Prim/Dijkstra]

Immutability of keys.

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

immutable in Java: `String`, `Integer`, `Double`

Priority Queue with Delete-Random

Goal. Design an efficient data structure to support the following ops:

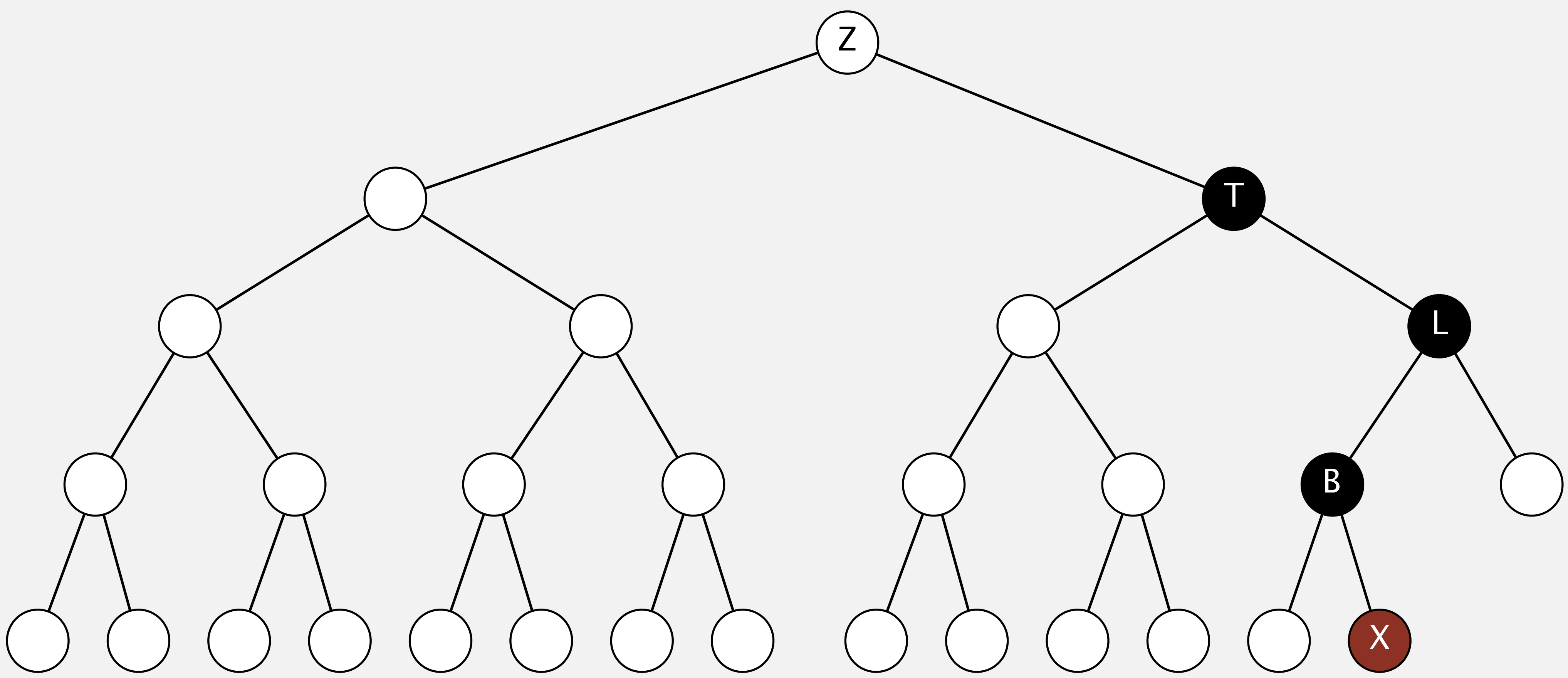
- Insert: insert a specified key.
- Delete-Max: delete and return a max key.
- Sample: return a random key.
- Delete-Random: delete and return a random key.

Midterm exam
Spring 2012

Binary heap: practical improvements

Do "half exchanges" in sink and swim.

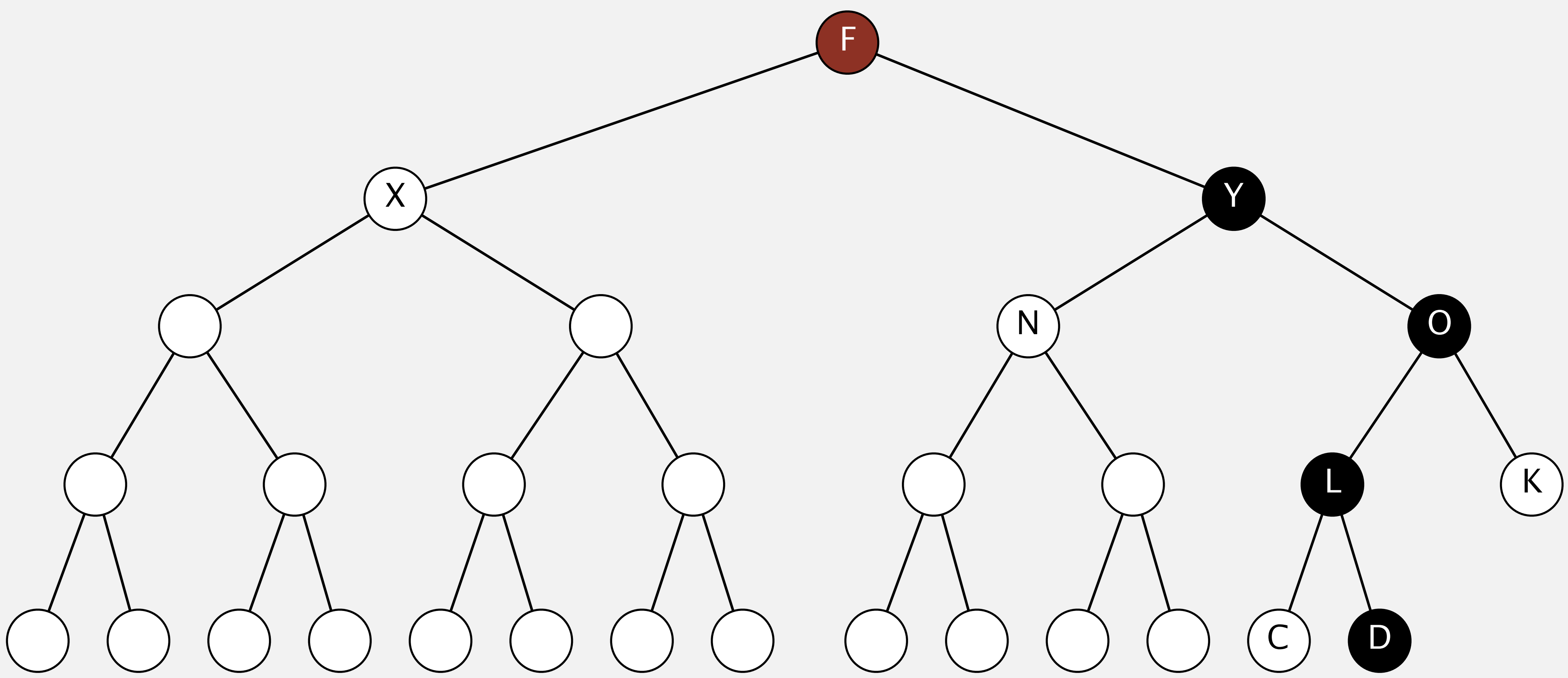
- Reduces number of array accesses.
- Worth doing.



Binary heap: practical improvements

Floyd's "bounce" heuristic.

- Sink key at root all the way to bottom. ⟵ only 1 compare per node
- Swim key back up. ⟵ some extra compares and exchanges
- Overall, fewer compares; more exchanges.

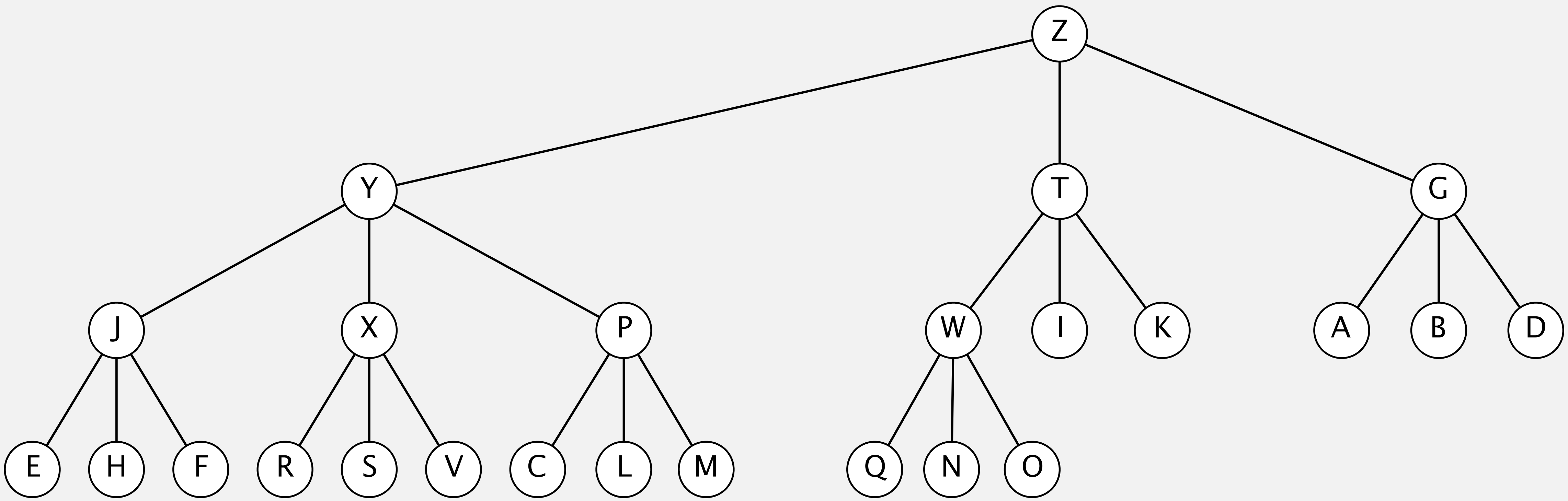


Binary heap: practical improvements

Multiway heaps.

- Complete d-way tree.
- Child's key no larger than parent's key.

Fact. Height of complete d-way tree on n nodes is $\sim \log_d n$.



3-way heap

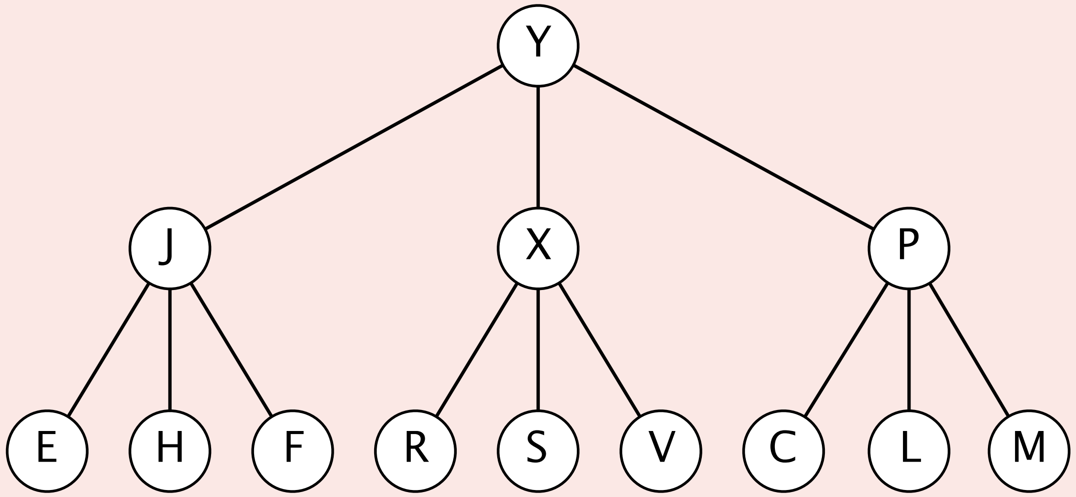
In the worst case, how many compares to INSERT and DELETE-MAX in a d-way heap as function of n and d?

A. $\sim \log_d n$ and $\sim \log_d n$

B. $\sim \log_d n$ and $\sim d \log_d n$

C. $\sim d \log_d n$ and $\sim \log_d n$

D. $\sim d \log_d n$ and $\sim d \log_d n$



Priority queue: implementation cost summary

implementation	INSERT	DELETE-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	← sweet spot: $d = 4$
Fibonacci	1	$\log n$ †	1	← see COS 423
Brodal queue	1	$\log n$	1	
impossible	1	1	1	← why impossible?

† amortized

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

https://algs4.cs.princeton.edu

2.4 Priority Queues

- *APIs*
- *elementary implementations*
- *binary heaps*
- ***heapsort***
- *event-driven simulation*

What are the properties of this sorting algorithm?

```
public void sort(String[] a)
{
    int n = a.length;
    MaxPQ<String> pq = new MaxPQ<String>();

    for (int i = 0; i < n; i++)
        pq.insert(a[i]);

    for (int i = n-1; i >= 0; i--)
        a[i] = pq.delMax();
}
```

A. $\Theta(n \log n)$ compares in the worst case.

B. In-place.

C. Stable.

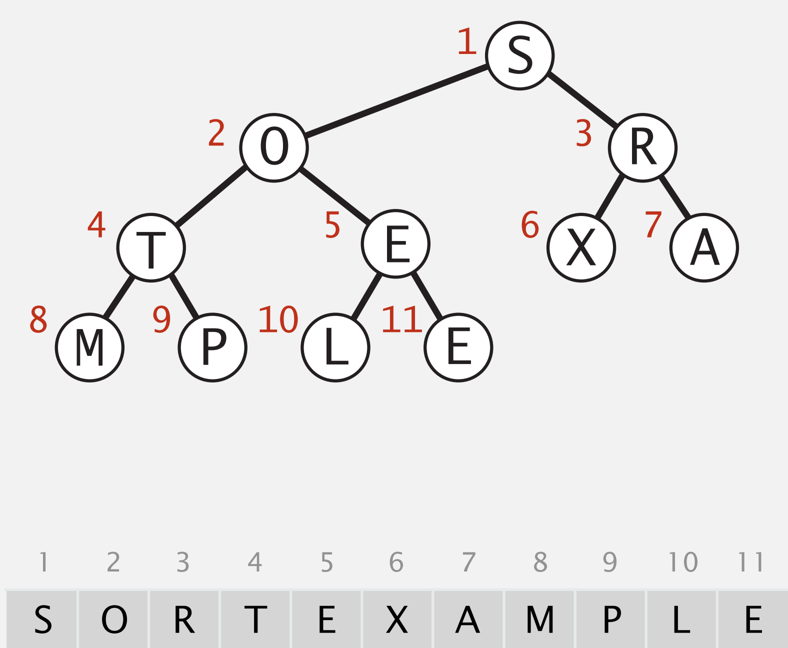
D. *All of the above.*

Heapsort

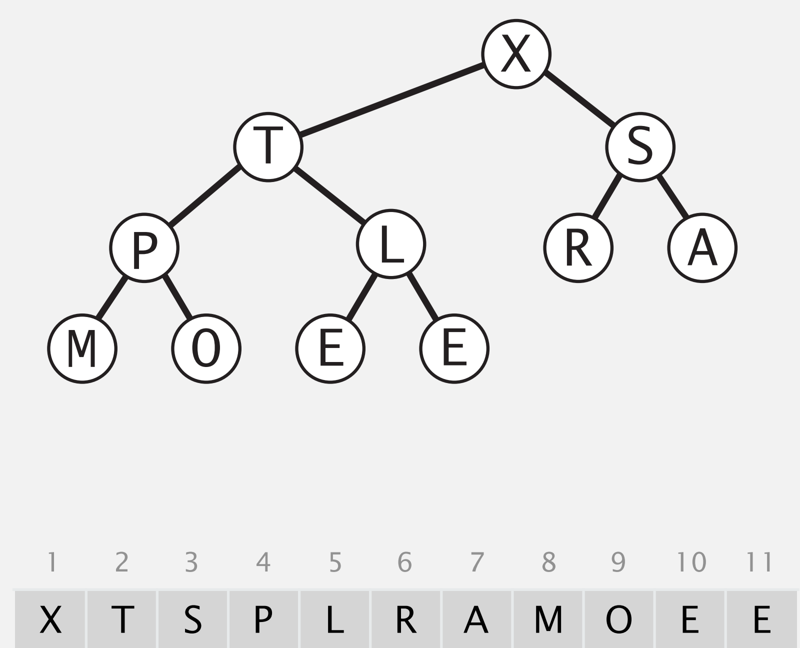
Basic plan for in-place sort.

- View input array as a complete binary tree.
- Heap construction: build a max-heap with all n keys.
- Sortdown: repeatedly remove the maximum key.

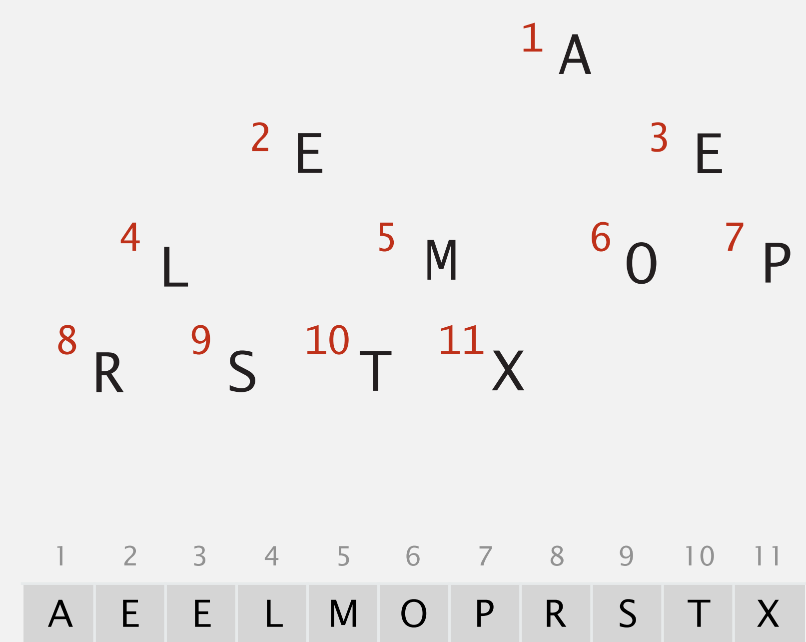
keys in arbitrary order



build max heap
(in place)



sorted result
(in place)



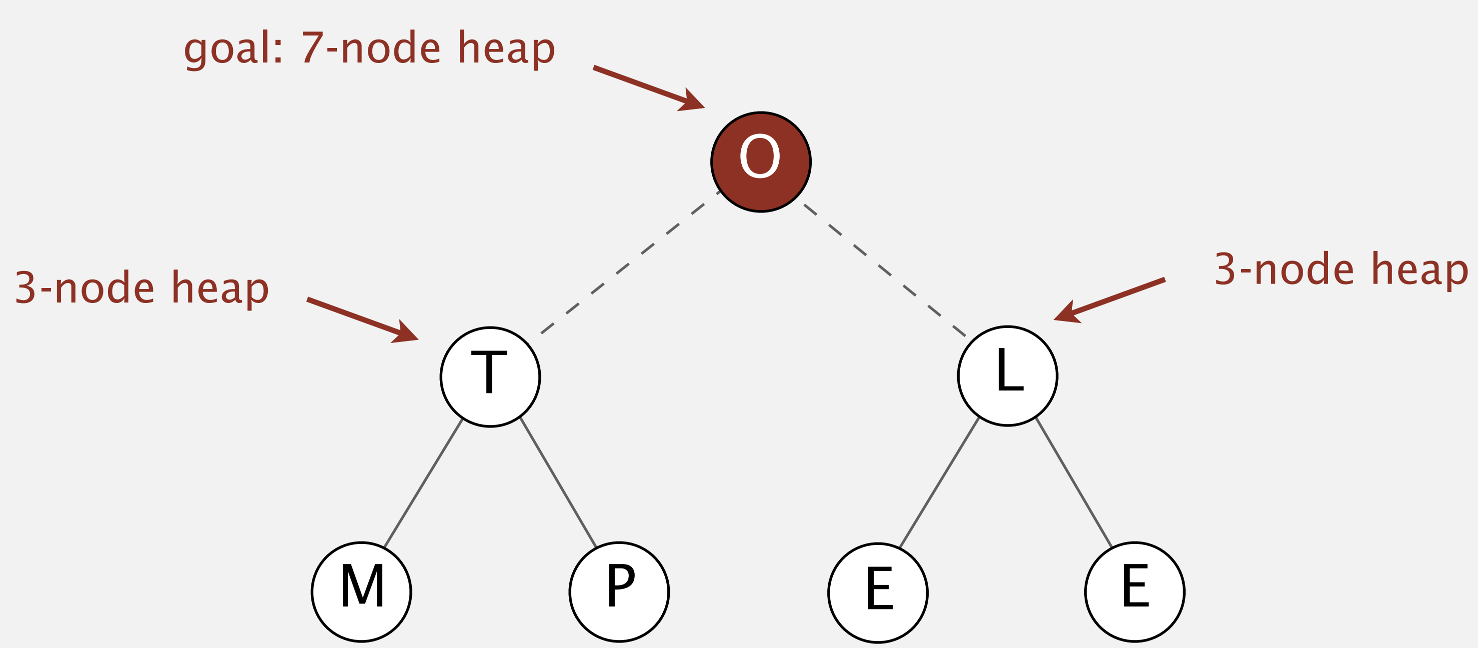
Heap construction

Top-down approach. Insert keys into a max-oriented heap, one at a time.

- Intuitive swim-based approach.
- $\Theta(n \log n)$ compares in worst case.

Bottom-up approach. Successively build larger heap from smaller ones.

- Clever sink-based alternative.
- $\Theta(n)$ compares. [stay tuned]

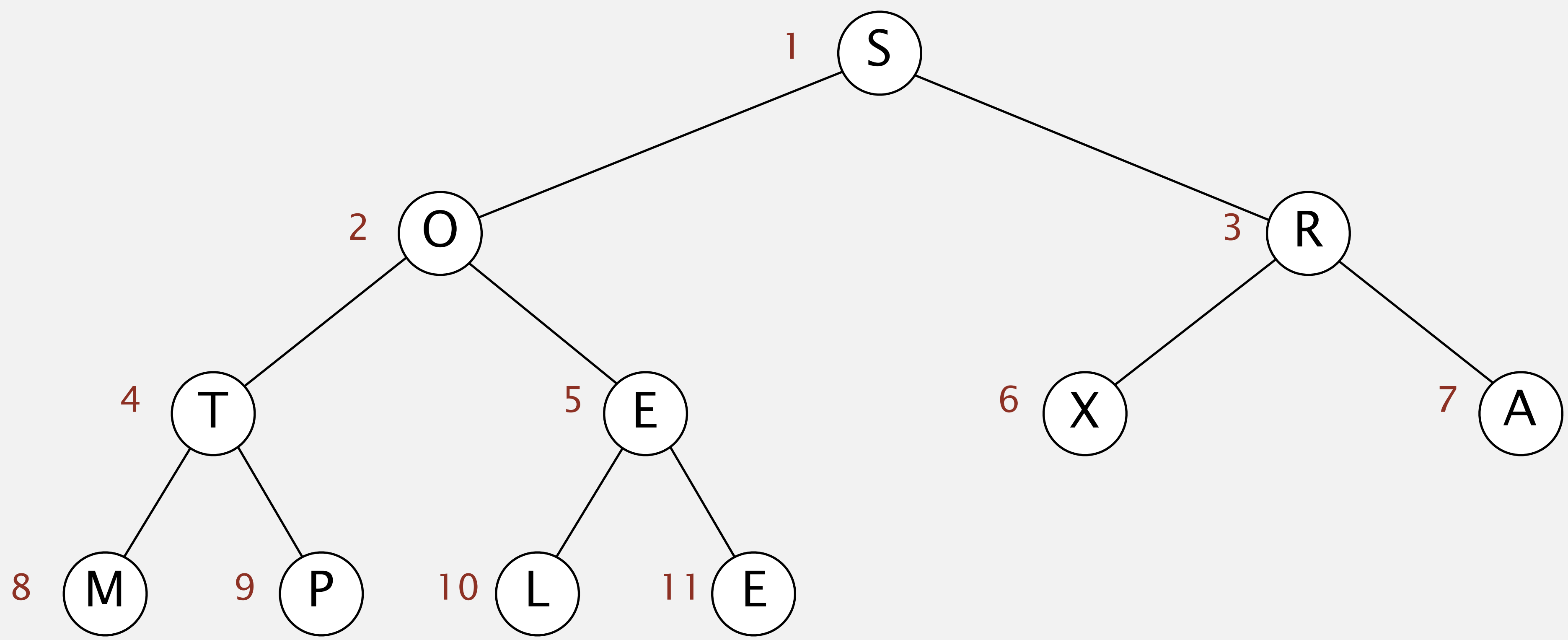


Heapsort demo

Heap construction. Build max heap using bottom-up method.

for now, assume array entries are indexed 1 to n

array in arbitrary order

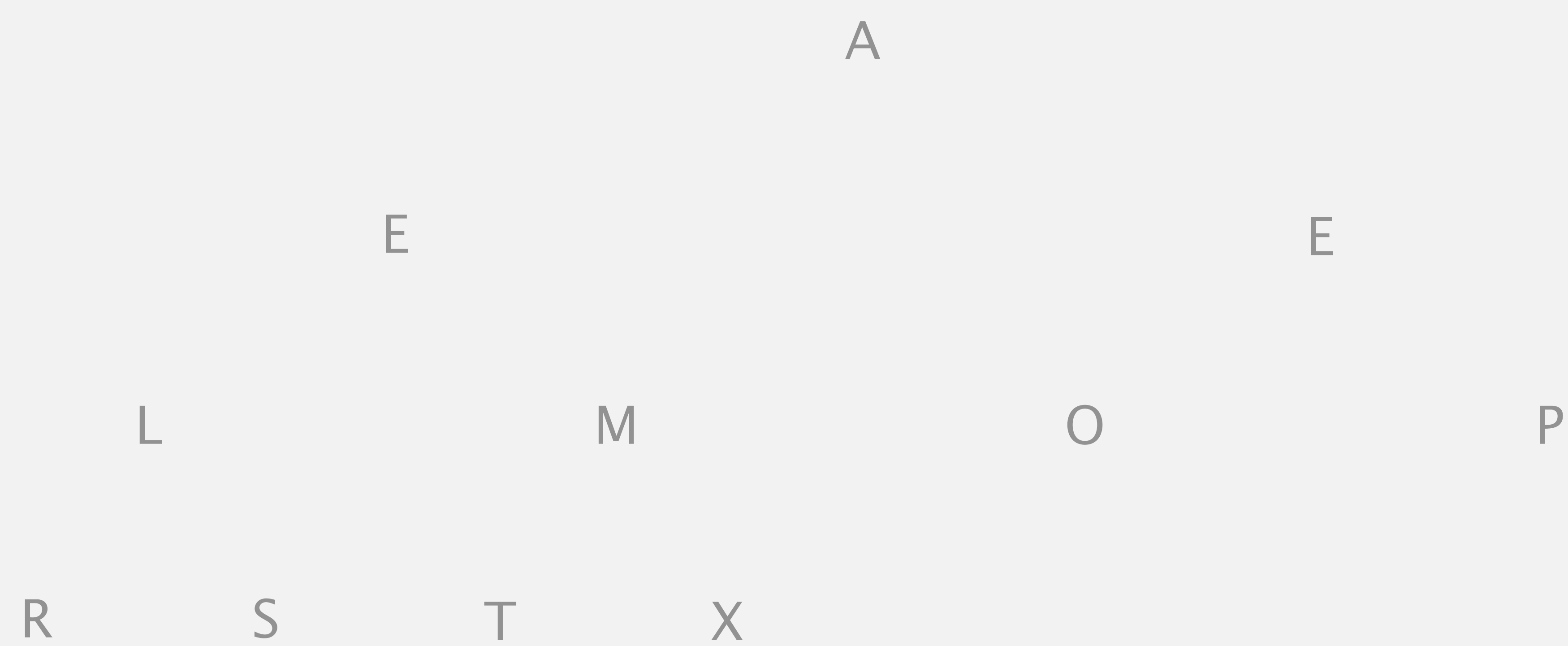


S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

array in sorted order



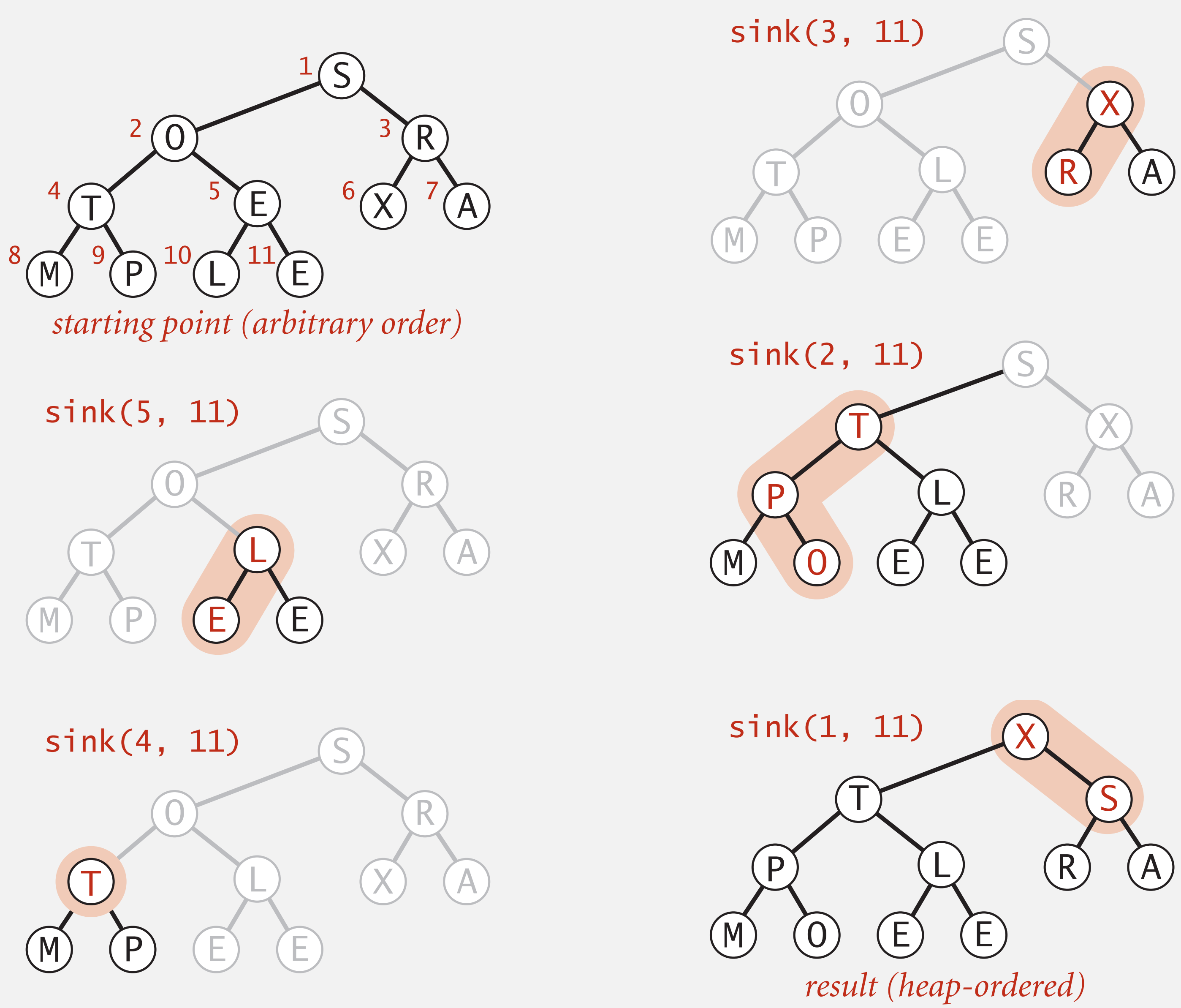
1	2	3	4	5	6	7	8	9	10	11
A	E	E	L	M	O	P	R	S	T	X

Heapsort: heap construction

First pass. Build heap using bottom-up approach.

Invariant. After calling `sink(a, k, n)`, trees rooted at `k` to `n` are heap-ordered.

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



Heapsort: sortdown

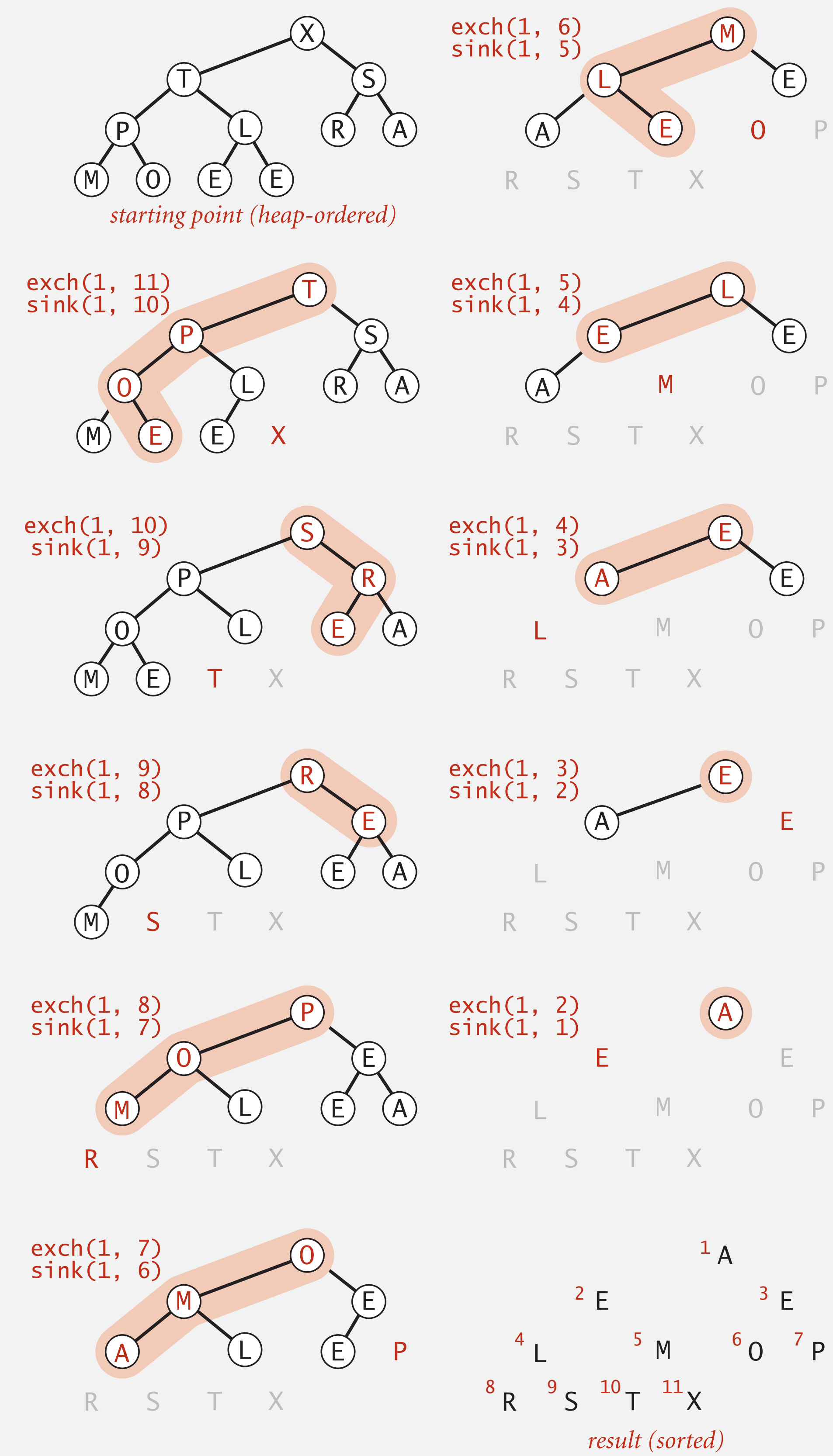
Second pass.

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

Invariants. After calling `sink(a, 1, k)`

- `a[k..n]` are in final sorted order.
- `a[1..k-1]` is a heap.

```
int k = n;
while (k > 1)
{
   exch(a, 1, k--);
   sink(a, 1, k);
}
```



Heapsort: 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);
      int k = n;
      while (k > 1)
      {
         exch(a, 1, k--);
         sink(a, 1, k);
      }
   }

   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(Object[] a, int i, int j)
   {  /* as before */  }

}
```

but make static (and pass arguments)

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

https://algs4.cs.princeton.edu/24pq/Heap.java.html

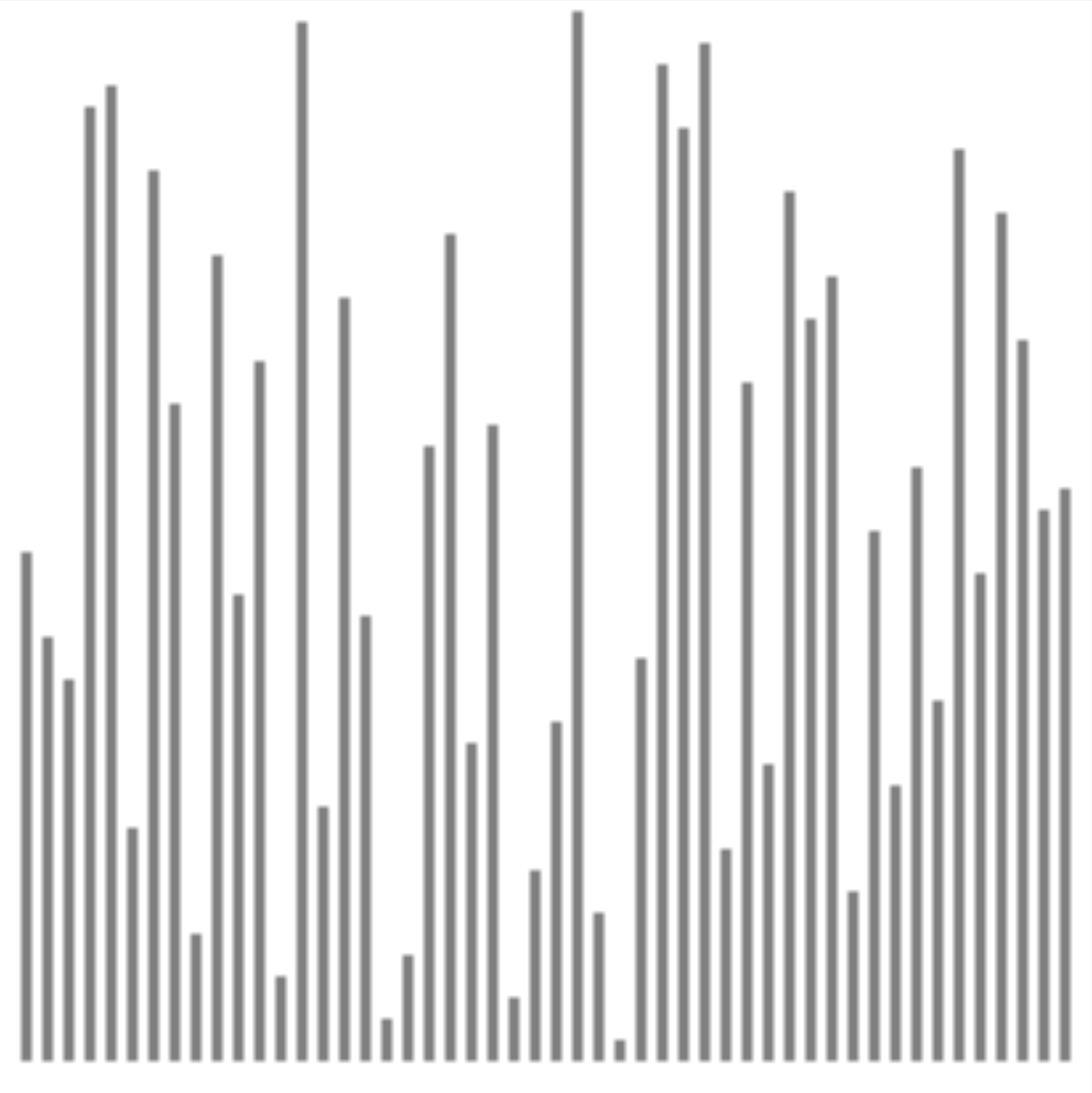
Heapsort: trace

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 animation

50 random items



https://www.toptal.com/developers/sorting-algorithms/heap-sort

▲ algorithm position

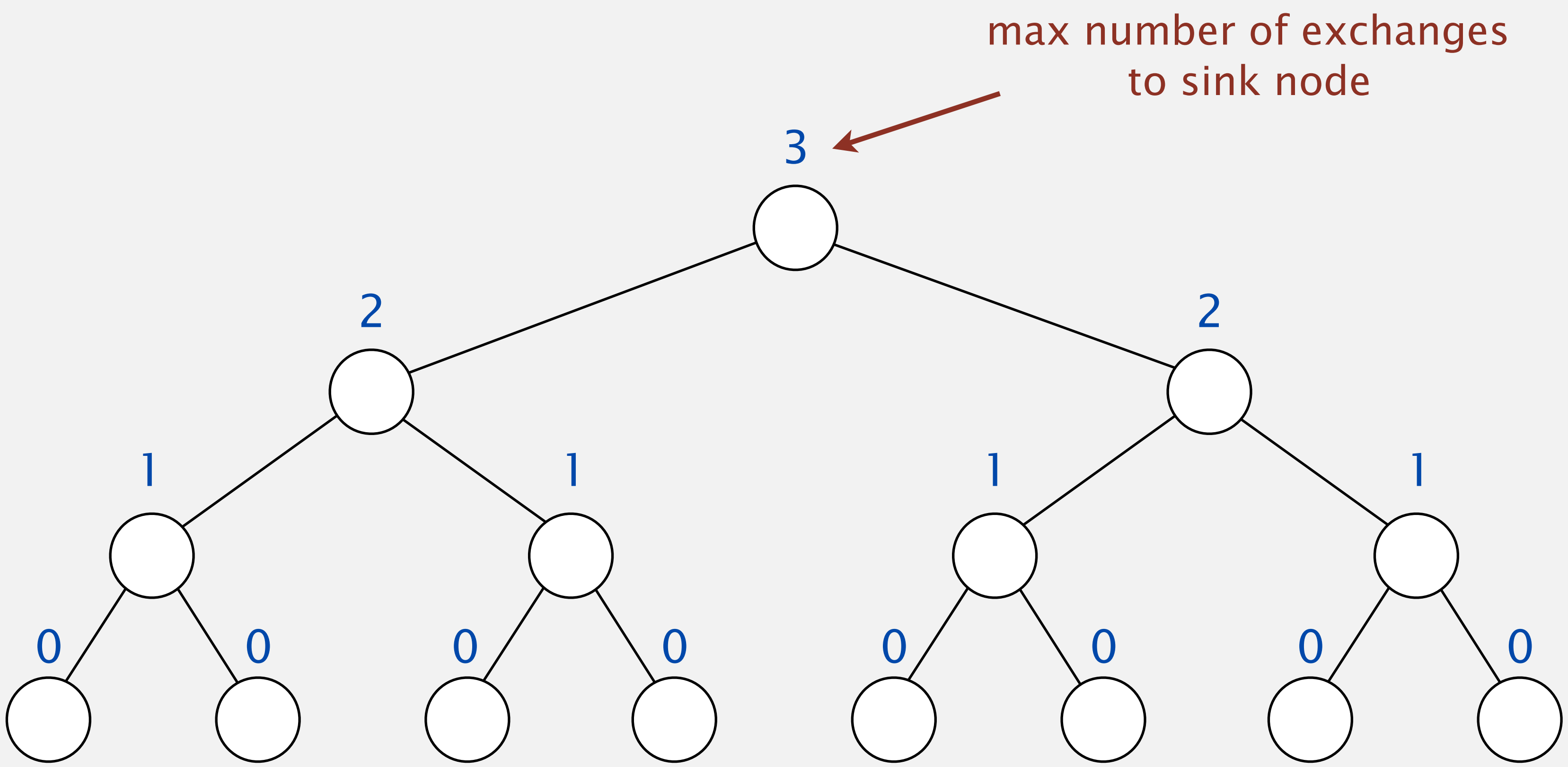
in order

not in order

Heapsort: mathematical analysis

Proposition. Heap construction makes $\leq n$ exchanges and $\leq 2\,n$ compares.

Pf sketch. [assume $n = 2^{h+1} - 1$]



binary heap of height h = 3

a tricky sum
(see COS 340)

$$\begin{aligned} h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots + 2^h(0) &= 2^{h+1} - h - 2 \\ &= n - (h-1) \\ &\leq n \end{aligned}$$

Heapsort: mathematical analysis

Proposition. Heap construction makes $\leq n$ exchanges and $\leq 2\,n$ compares.

Proposition. Heapsort makes $\leq 2\,n \lg n$ compares and exchanges.

algorithm can be improved to $\sim n \lg n$
(but no such variant is known to be practical)

Significance. In-place sorting algorithm with $\Theta(n \log n)$ worst-case.

- Mergesort: no, $\Theta(n)$ extra space. ← in-place merge possible, not practical
- Quicksort: no, $\Theta(n^2)$ time in worst case. ← $\Theta(n \log n)$ 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.
- Not stable.

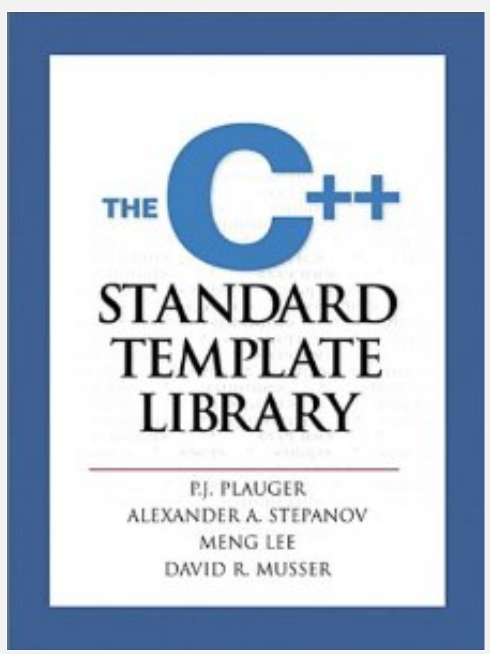
can be improved using
advanced caching tricks

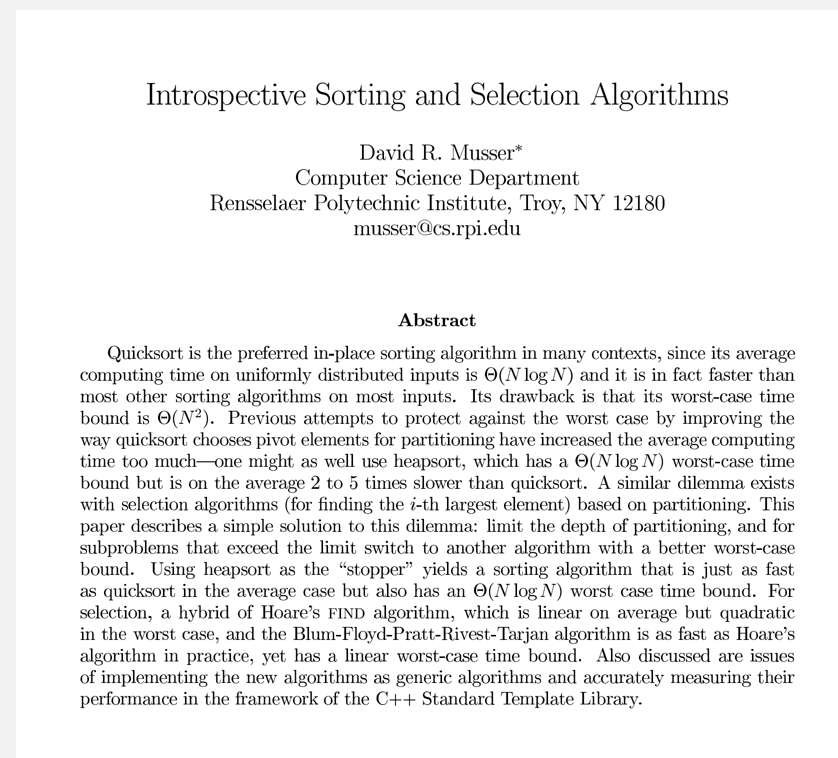
Introsort

Goal. As fast as quicksort in practice; $\Theta(n \log n)$ worst case; in place.

Introsort.

- Run quicksort.
- Cutoff to heapsort if stack depth exceeds $2 \lg n$.
- Cutoff to insertion sort for $n = 16$.





Introspective Sorting and Selection Algorithms

David R. Musser*
Computer Science Department
Rensselaer Polytechnic Institute, Troy, NY 12180
musser@cs.rpi.edu

Abstract

Quicksort is the preferred in-place sorting algorithm in many contexts, since its average computing time on uniformly distributed inputs is $\Theta(N \log N)$ and it is in fact faster than most other sorting algorithms on most inputs. Its drawback is that its worst-case time bound is $\Theta(N^2)$. Previous attempts to protect against the worst case by improving the way quicksort chooses pivot elements for partitioning have increased the average computing time too much—one might as well use heapsort, which has a $\Theta(N \log N)$ worst-case time bound but is on the average 2 to 5 times slower than quicksort. A similar dilemma exists with selection algorithms (for finding the i-th largest element) based on partitioning. This paper describes a simple solution to this dilemma: limit the depth of partitioning, and for subproblems that exceed the limit switch to another algorithm with a better worst-case bound. Using heapsort as the "stopper" yields a sorting algorithm that is just as fast as quicksort in the average case but also has an $\Theta(N \log N)$ worst case time bound. For selection, a hybrid of Hoare's FIND algorithm, which is linear on average but quadratic in the worst case, and the Blum-Floyd-Pratt-Rivest-Tarjan algorithm is as fast as Hoare's algorithm in practice, yet has a linear worst-case time bound. Also discussed are issues of implementing the new algorithms as generic algorithms and accurately measuring their performance in the framework of the C++ Standard Template Library.

In the wild. C++ STL, Microsoft .NET Framework.

Sorting algorithms: summary

	inplace?	stable?	best	average	worst	remarks
selection	✔		$\frac{1}{2}\ n^2$	$\frac{1}{2}\ n^2$	$\frac{1}{2}\ n^2$	n exchanges
insertion	✔	✔	n	$\frac{1}{4}\ n^2$	$\frac{1}{2}\ n^2$	use for small n or partially ordered
merge		✔	$\frac{1}{2}\ n \lg n$	$n \lg n$	$n \lg n$	$n \log n$ guarantee; stable
timsort		✔	n	$n \lg n$	$n \lg n$	improves mergesort when preexisting order
quick	✔		$n \lg n$	$2\ n \ln n$	$\frac{1}{2}\ n^2$	$n \log n$ probabilistic guarantee; fastest in practice
3-way quick	✔		n	$2\ n \ln n$	$\frac{1}{2}\ n^2$	improves quicksort when duplicate keys
heap	✔		$3\ n$	$2\ n \lg n$	$2\ n \lg n$	$n \log n$ guarantee; in-place
?	✔	✔	n	$n \lg n$	$n \lg n$	holy sorting grail