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	*singly linked list* *resizing array*
queue	ENQUEUE, DEQUEUE	
deque	ADD-FIRST, REMOVE-FIRST, ADD-LAST, REMOVE-LAST	*doubly linked list* *resizing array*
priority queue	INSERT, DELETE-MAX	*binary heap*
symbol table	PUT, GET, DELETE	*binary search tree* *hash table*
set	ADD, CONTAINS, DELETE	

Priority queue

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

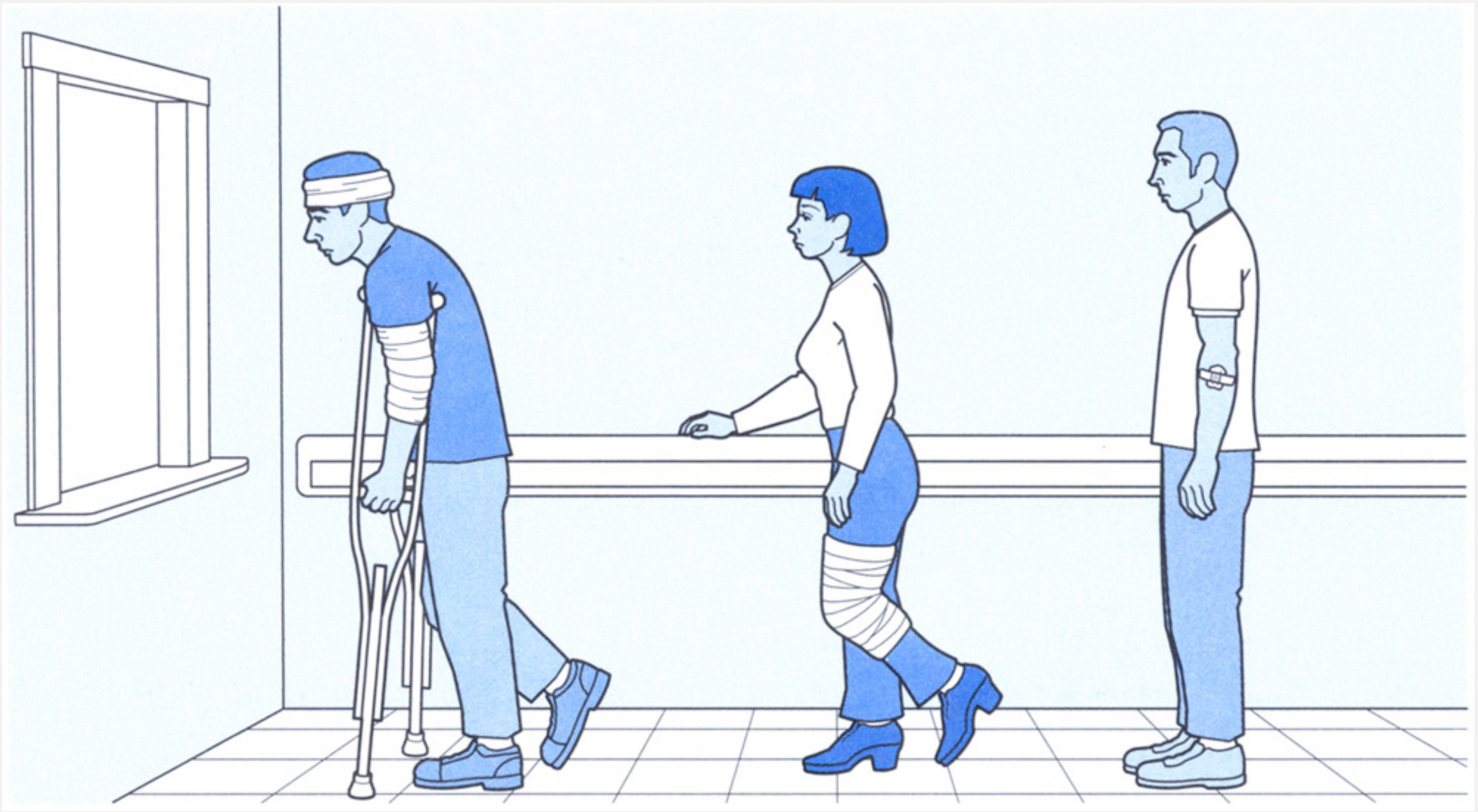
Stack. Remove the item most recently added.

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; type 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 keys in the priority queue*

Note. Duplicate keys allowed; `delMax()` removes and returns any maximum key.

Min-oriented priority queue API

Analogous to `MaxPQ`.

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

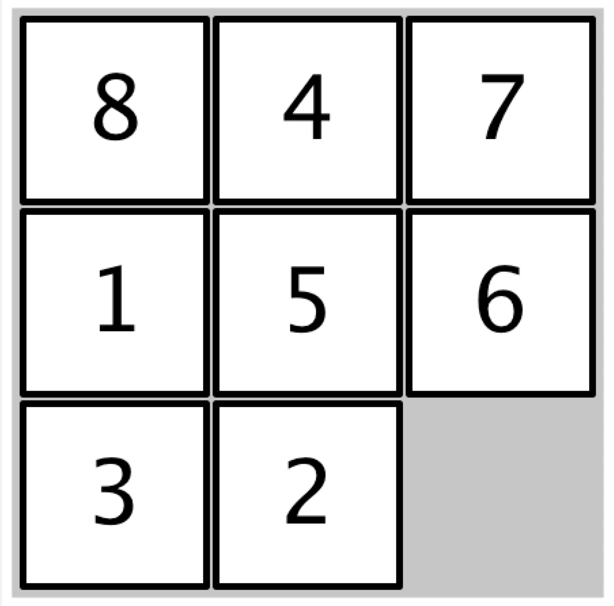
	`MinPQ()`	*create an empty priority queue*
`void`	`insert(Key v)`	*insert a key*
`Key`	`delMin()`	*return and remove a smallest key*
`boolean`	`isEmpty()`	*is the priority queue empty?*
`Key`	`min()`	*return a smallest key*
`int`	`size()`	*number of keys in the priority queue*

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

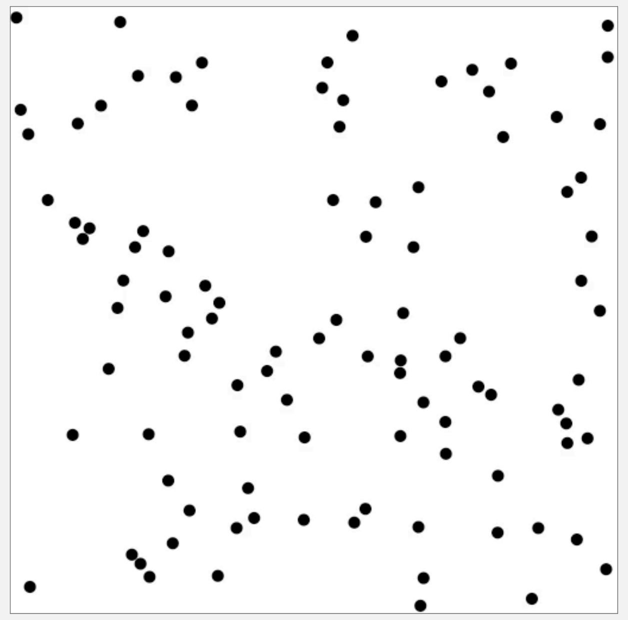
Priority queue: applications

- 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

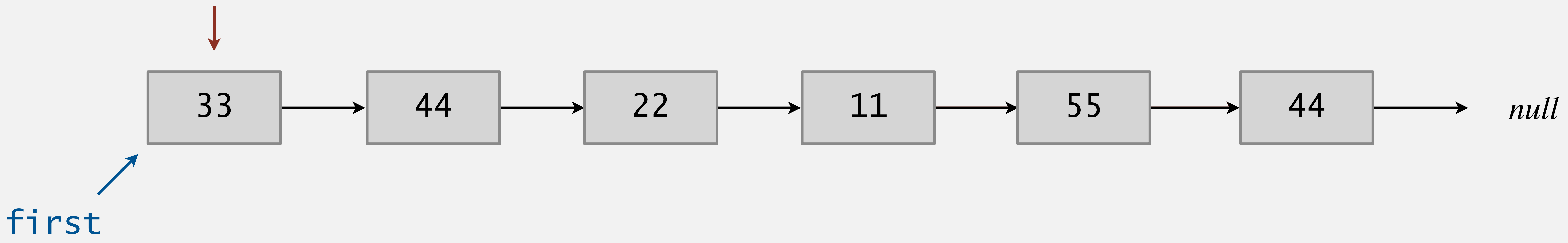
https://algs4.cs.princeton.edu

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	44	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, in 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
11	22	33	44	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
unordered list	1	n
ordered array	n	1
goal	$\log n$	$\log n$

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

Challenge. Implement both INSERT and DELETE-MAX efficiently.

Solution. "Somewhat-ordered" array.

https://algs4.cs.princeton.edu

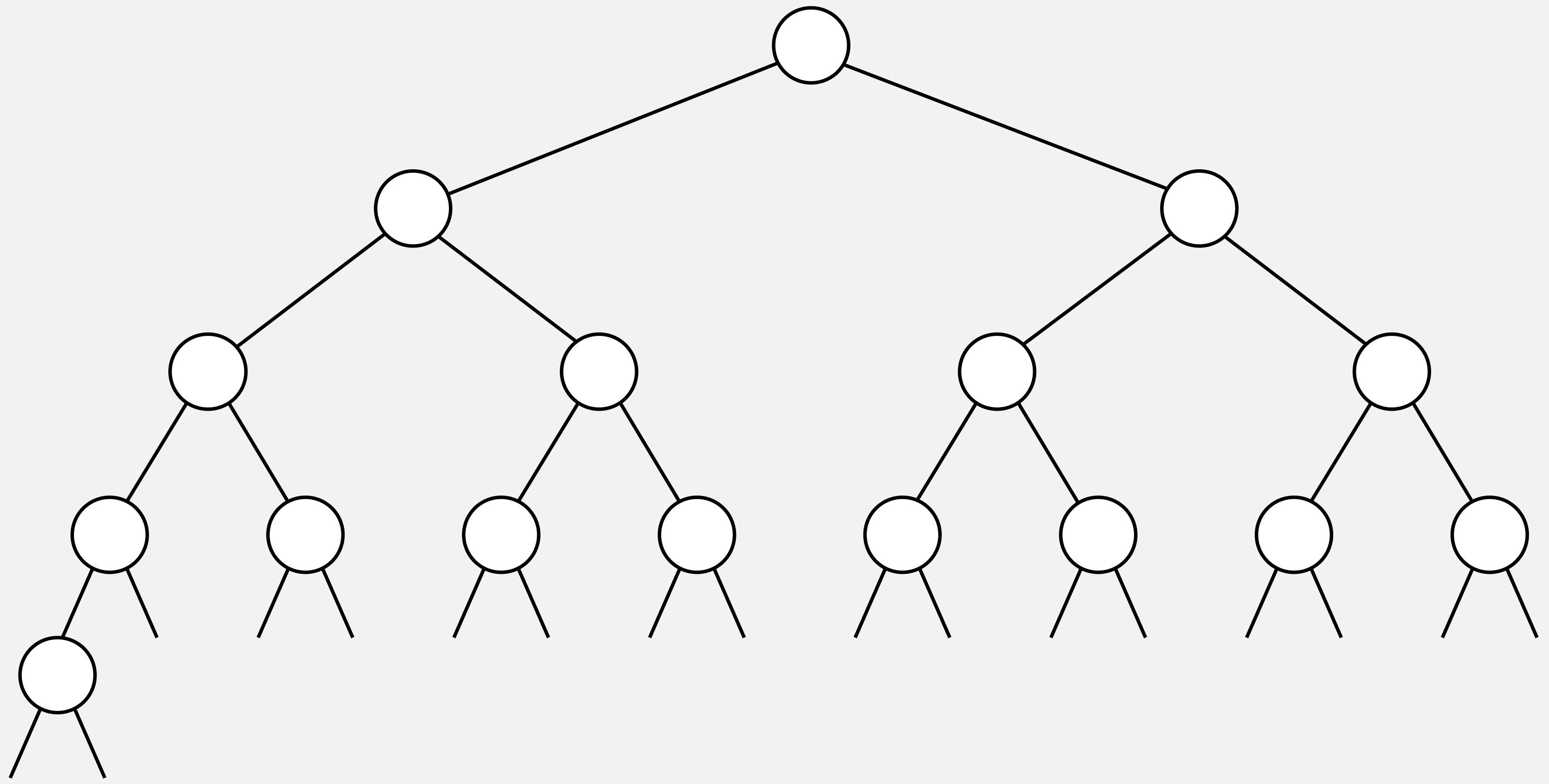
2.4 Priority Queues

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

Complete binary tree

Binary tree. Empty or node with links to two disjoint binary trees (left and right subtrees).

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 $\lfloor \log_2 n \rfloor$.

Pf. As you successively add nodes, height increases (by 1) only when n is a power of 2.

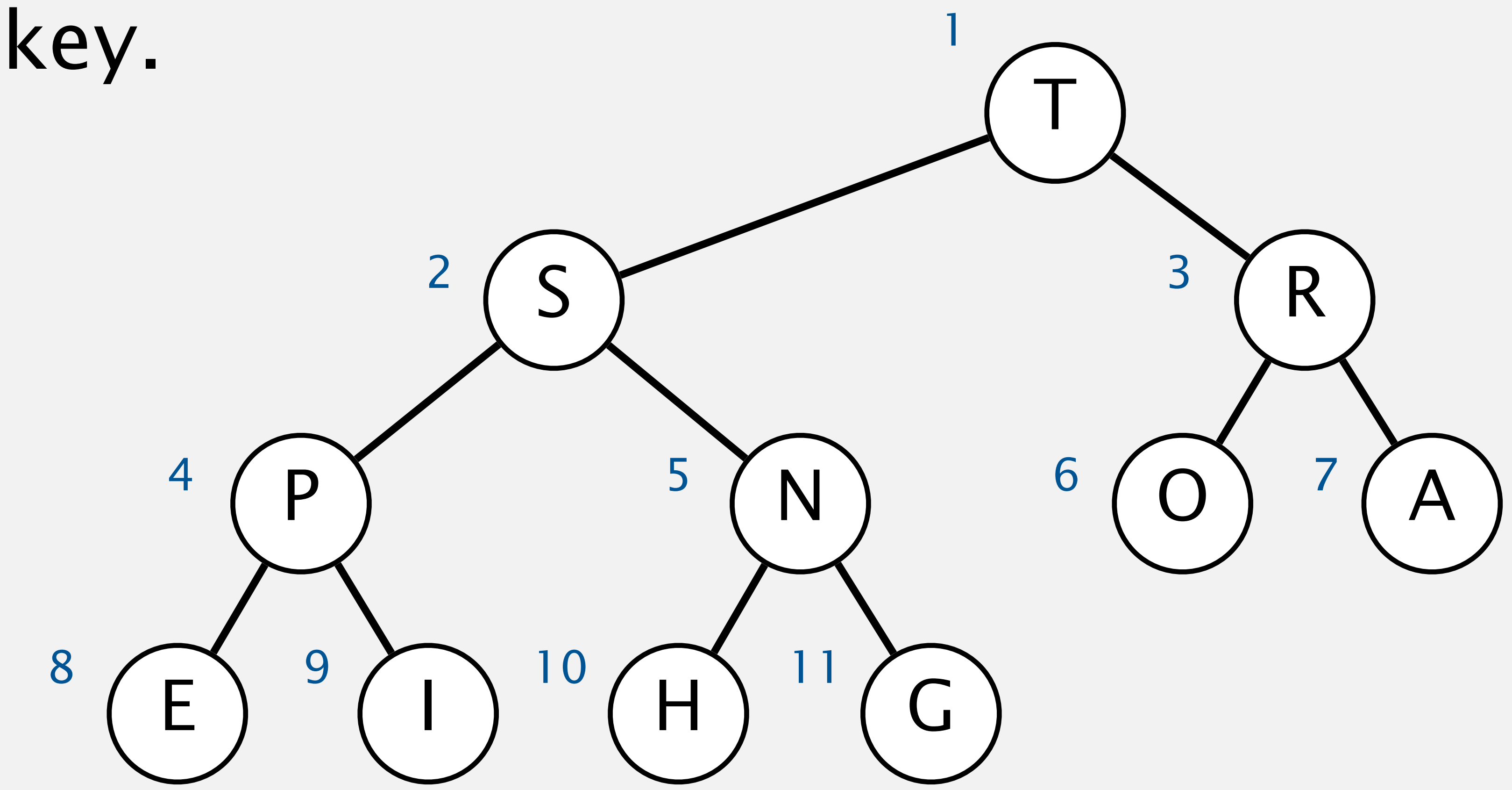
A complete binary tree in nature (of height 4)

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!

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

Priority queues: quiz 2

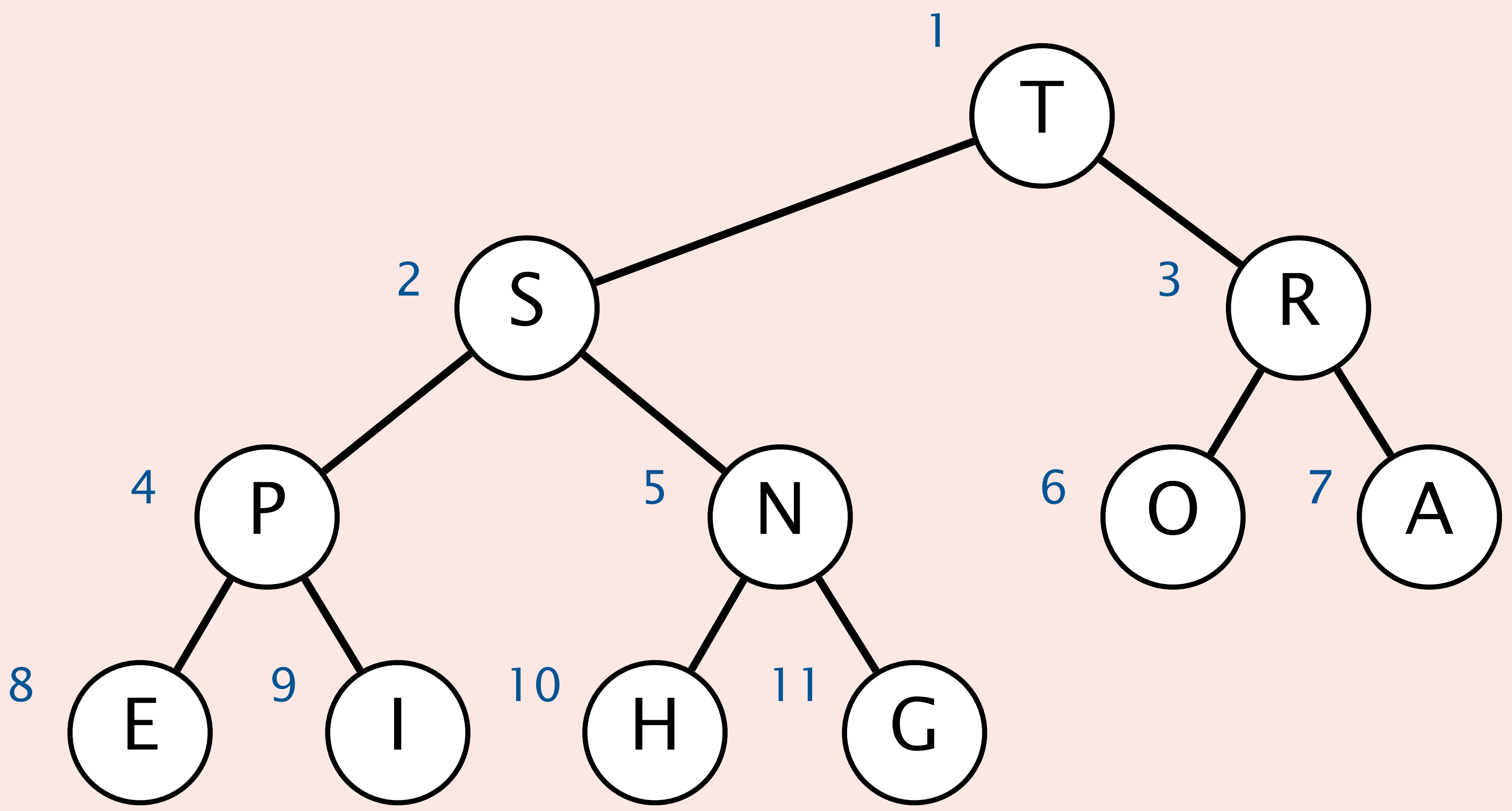
Consider the node at index k in a binary heap. Which Java expression produces the index of its parent?

A. `(k - 1) / 2`

B. `k / 2`

C. `(k + 1) / 2`

D. `2 * k`



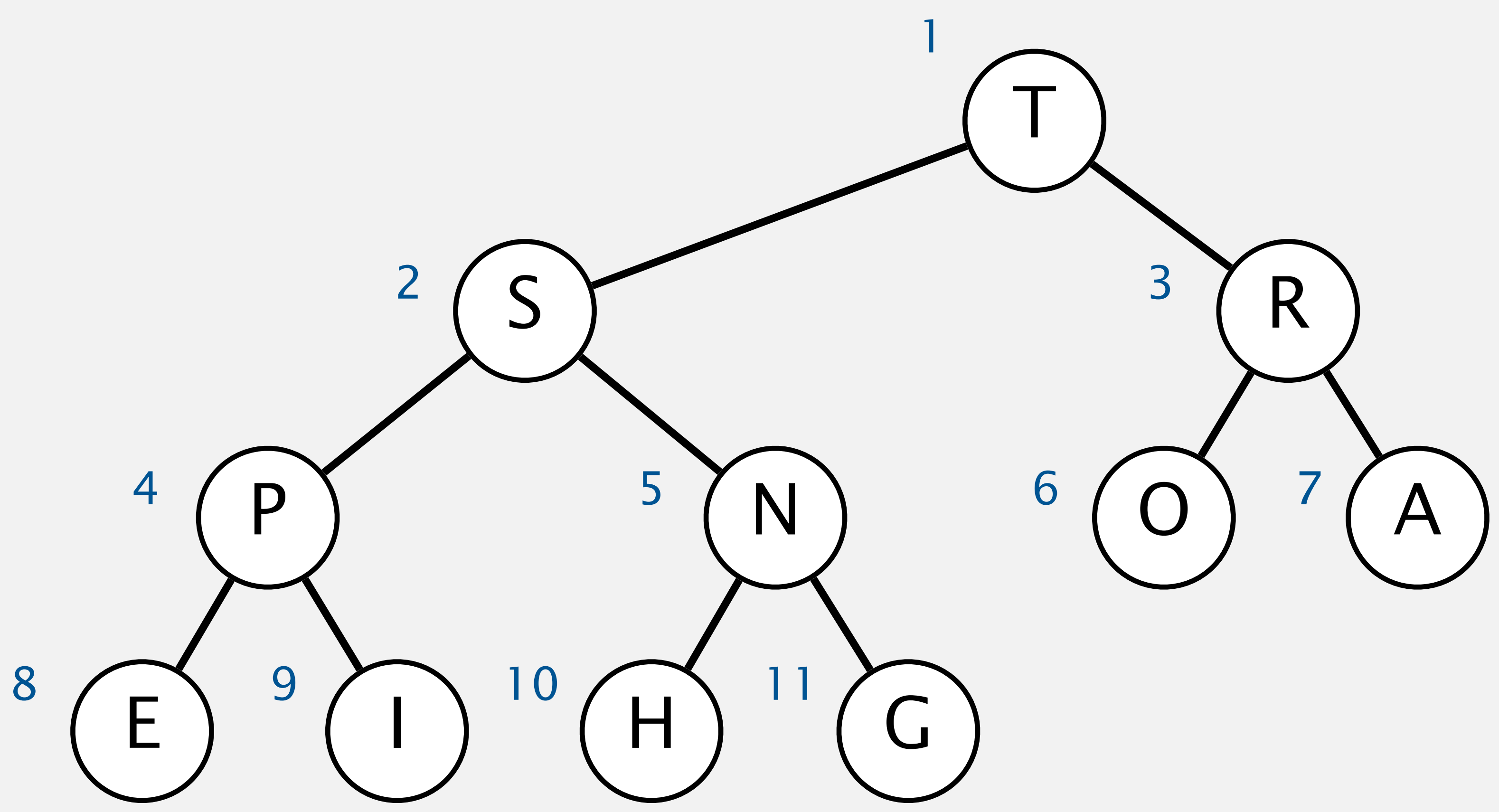
	0	1	2	3	4	5	6	7	8	9	10	11
a[]	–	T	S	R	P	N	O	A	E	I	H	G

Binary heap: properties

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

Proposition. Can use array indices to move up or down 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.

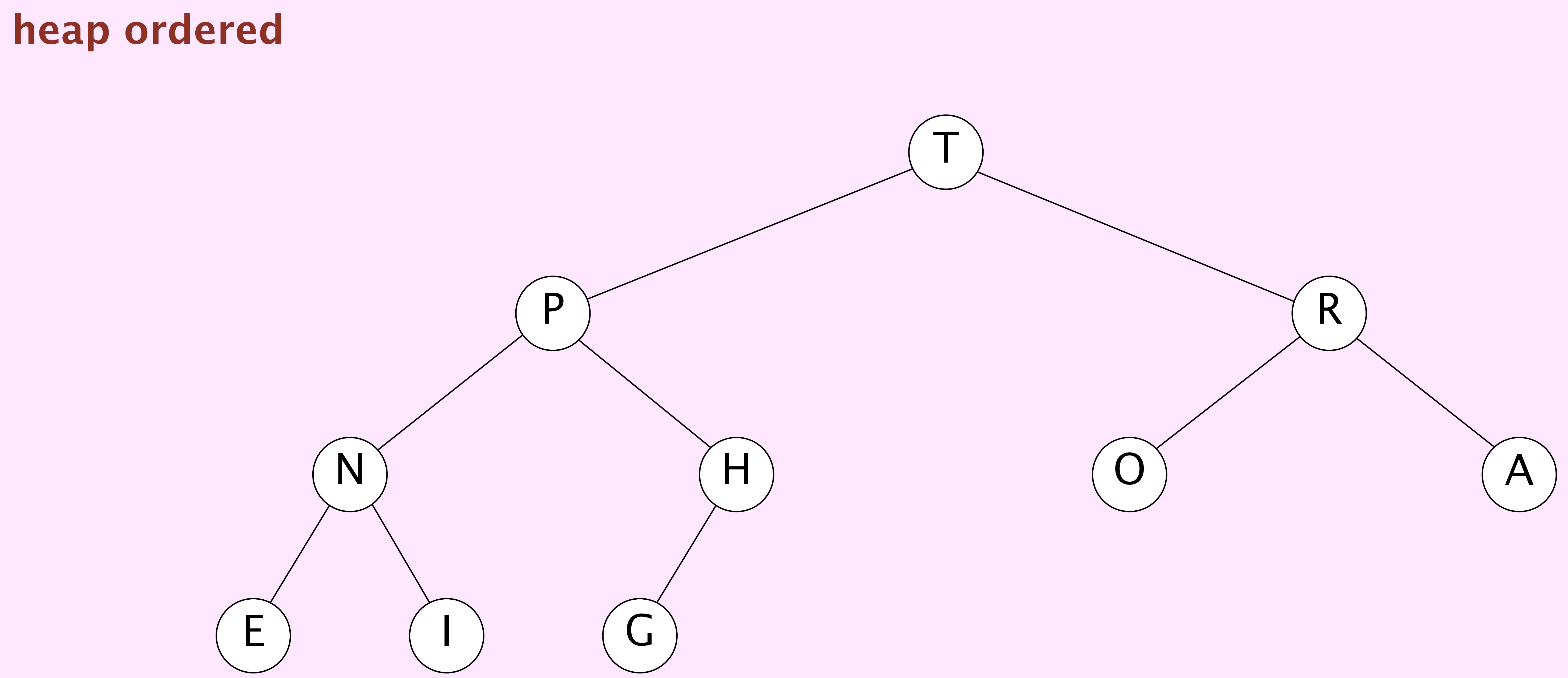


	0	1	2	3	4	5	6	7	8	9	10	11
a[]	–	T	S	R	P	N	O	A	E	I	H	G

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.



T	P	R	N	H	O	A	E	I	G	

Binary heap: promotion

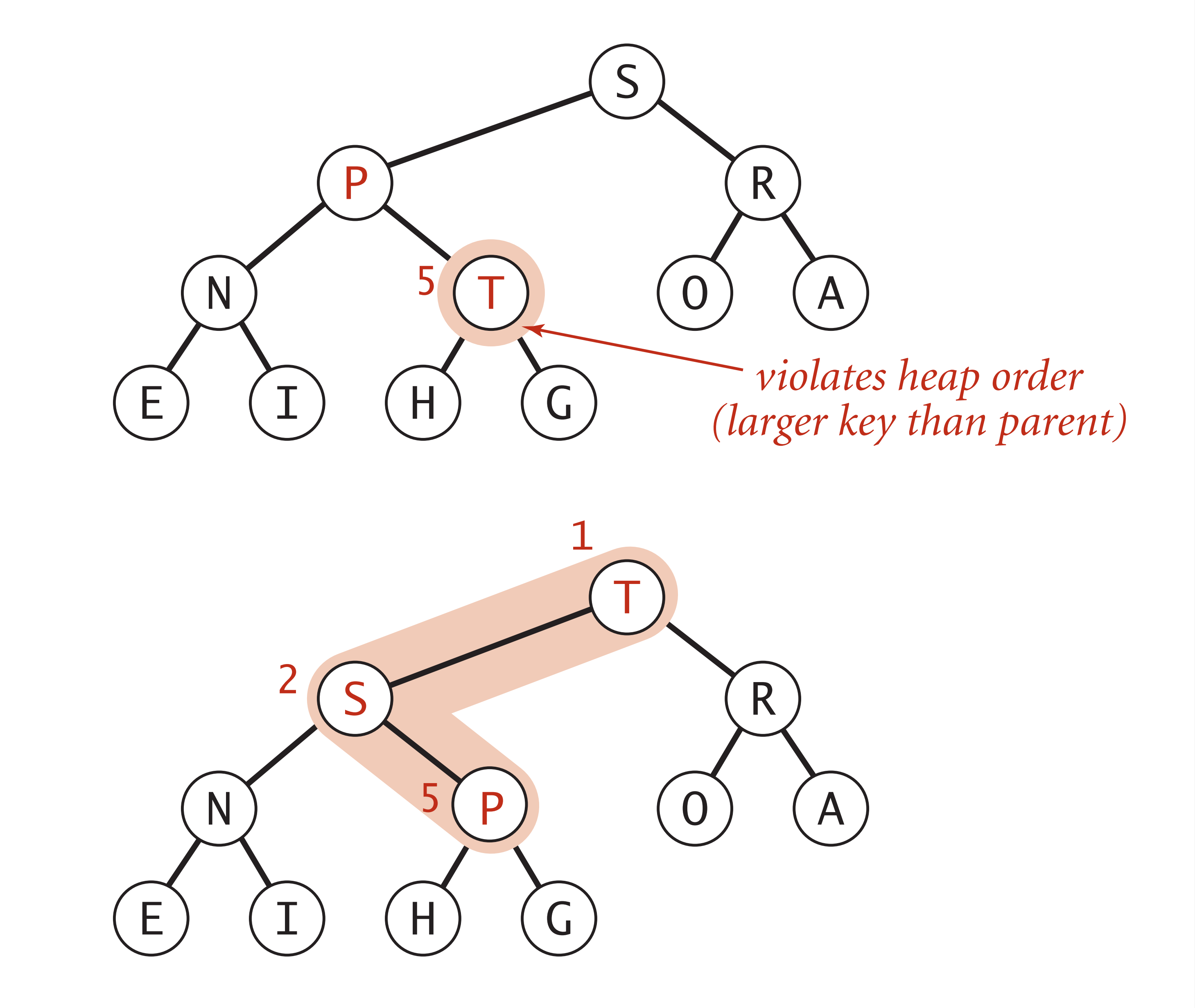
Scenario. Key in node becomes larger than key in parent's node.

To eliminate the violation:

- Exchange key in child node with key in parent node.
- 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



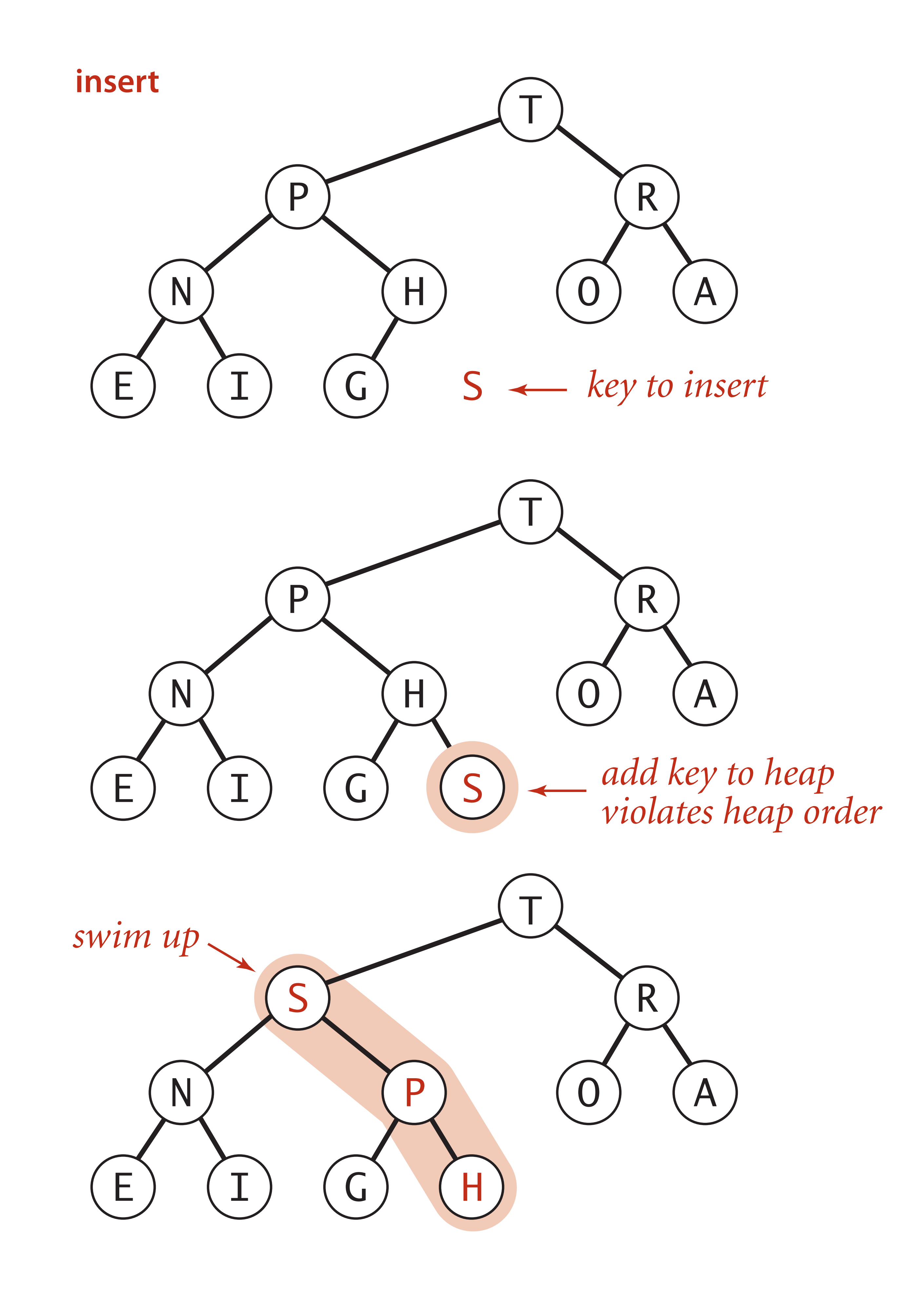
Peter principle. Node promoted to level of incompetence.

Binary heap: insertion

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

Cost. At most $1 + \log_2 n$ compares.

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



Binary heap: demotion

Scenario. Key in node becomes smaller than one (or both) of keys in childrens' nodes.

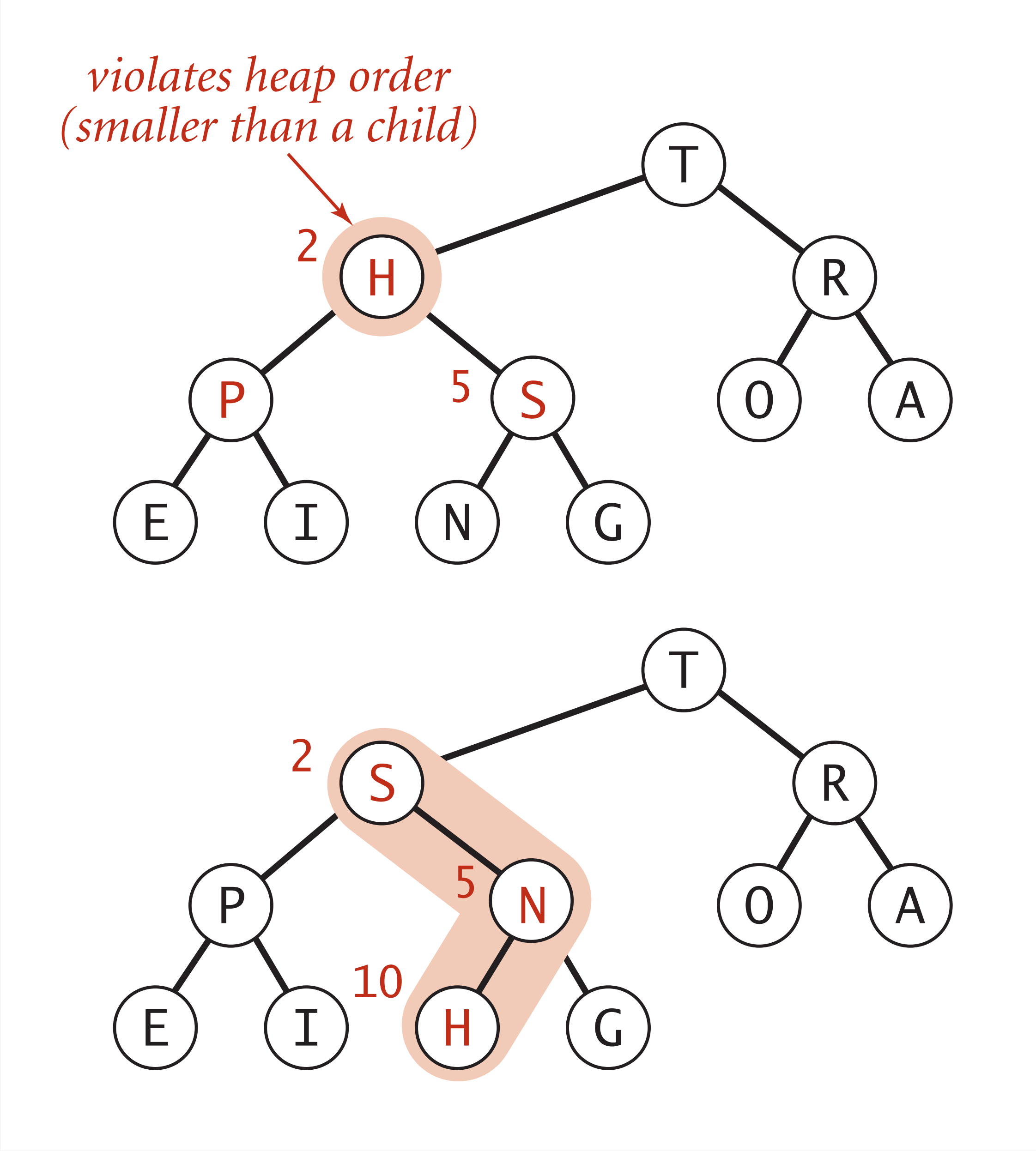
To eliminate the violation:

why not smaller child?

- Exchange key in parent node with key in larger child's node.
- 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



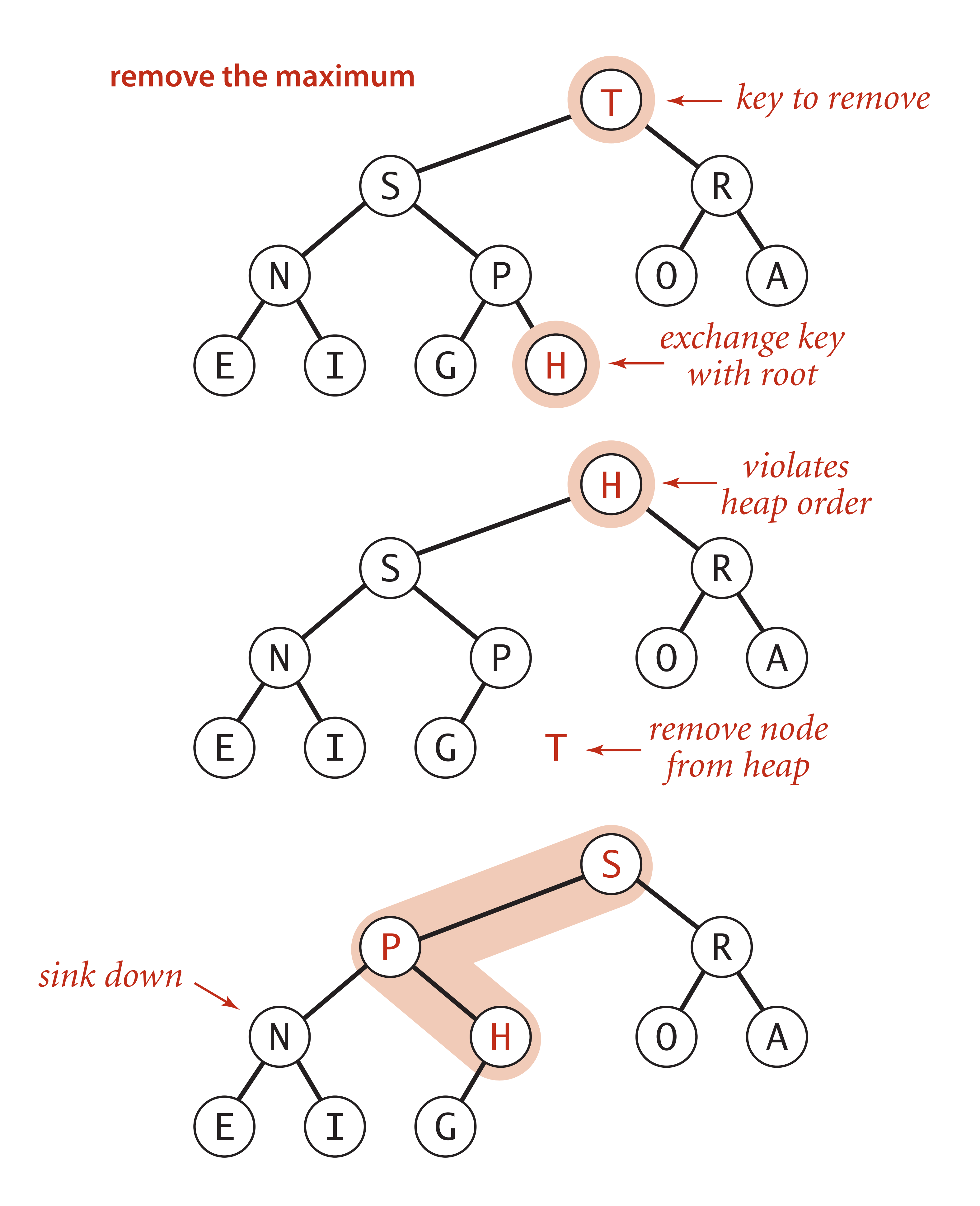
Power struggle. Better subordinate promoted.

Binary heap: delete the maximum

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

Cost. At most $2 \log_2 n$ 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 temp = a[i]; a[i] = a[j]; a[j] = temp;  }
}
```

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 $O(\log n)$ 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

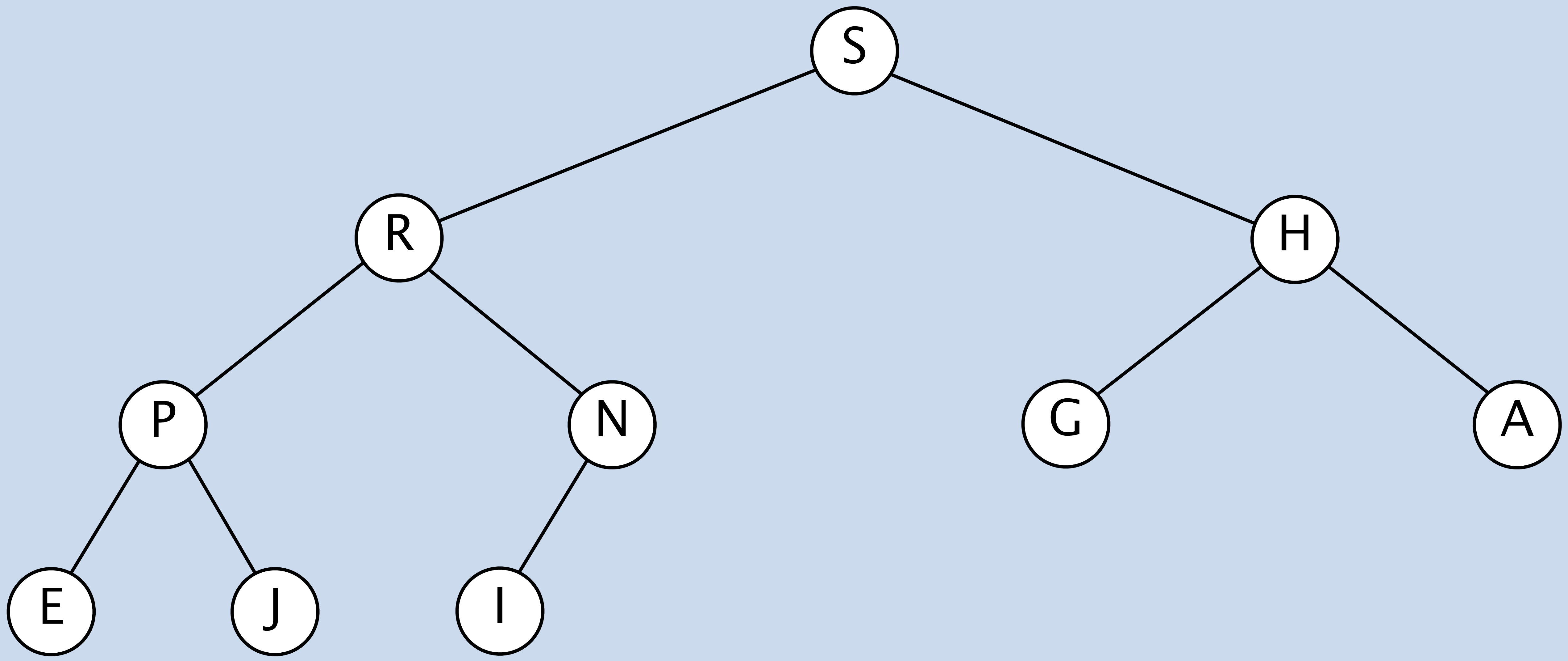
Goal. Design an efficient data structure to support the following API:

- INSERT: insert a key.
- DELETE-MAX: return and remove a largest key.
- SAMPLE: return a random key.
- DELETE-RANDOM: return and remove a random key.

Midterm exam
Spring 2012

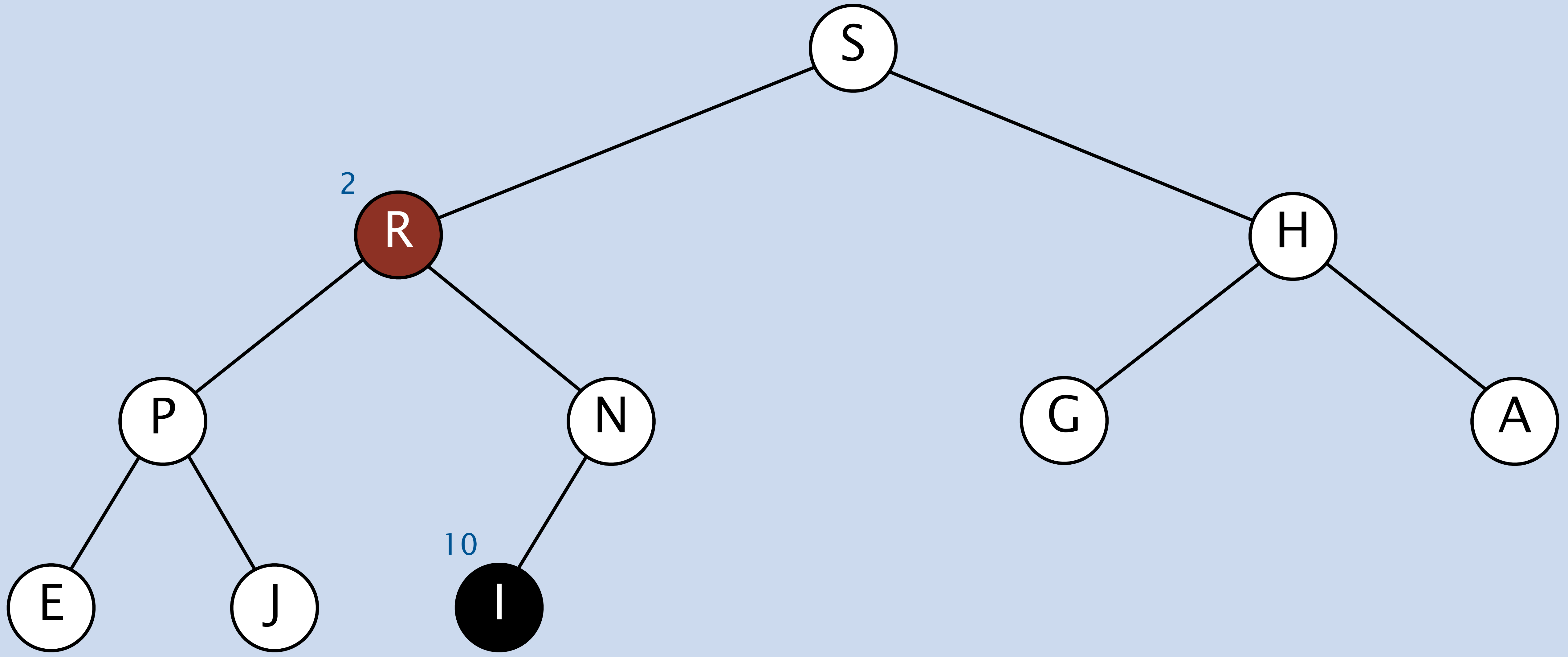
Delete-Random From a Binary Heap

Goal. Delete a random key from a binary heap in $O(\log n)$ time.



Delete-Random From a Binary Heap

Goal. Delete a random key from a binary heap in $O(\log n)$ time.



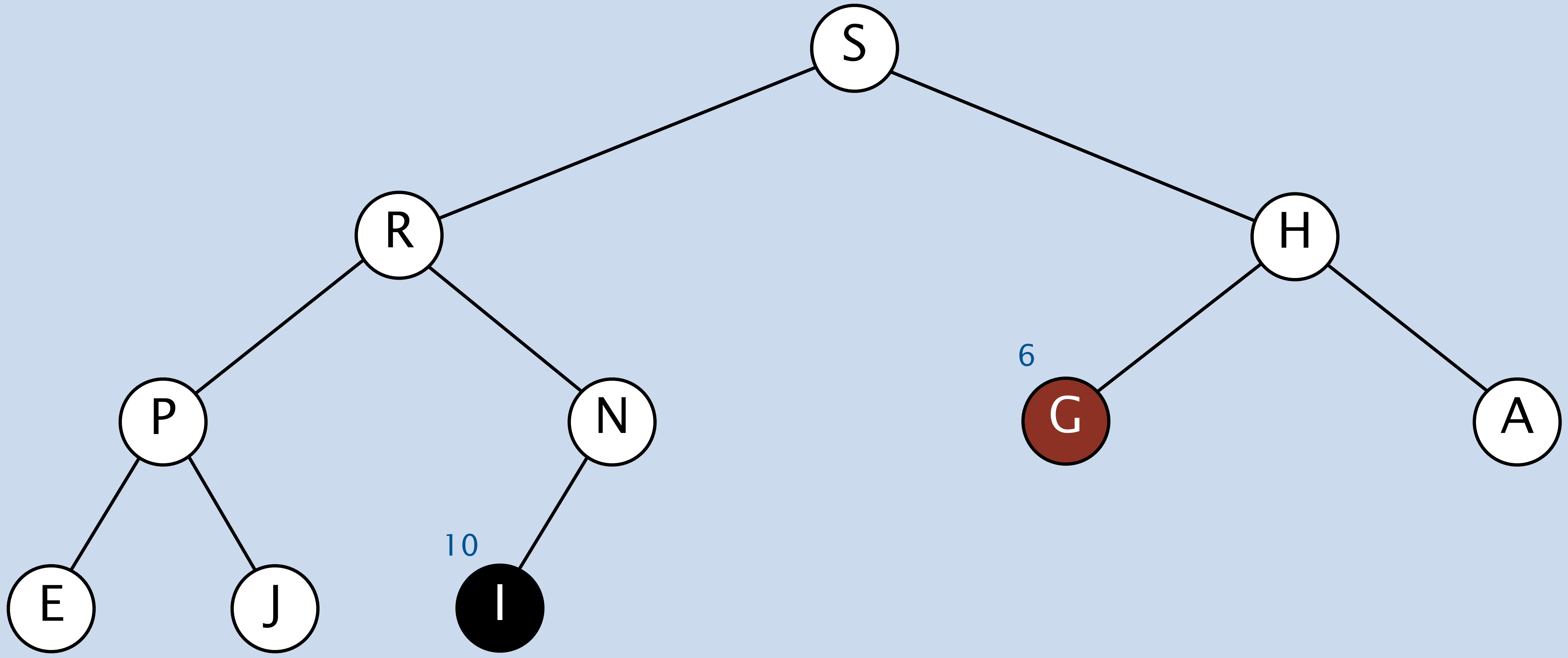
Solution.

- Pick a random index r between 1 and n.
- Perform `exch(r, n--)`.
- Perform `sink(r)`.

same trick as with dequeue() in a randomized queue

DELETE-RANDOM FROM A BINARY HEAP

Goal. Delete a random key from a binary heap in $O(\log n)$ time.



Solution.

- Pick a random index r between 1 and n.
- Perform `exch(r, n--)`.
- Or perform `swim(r)`.

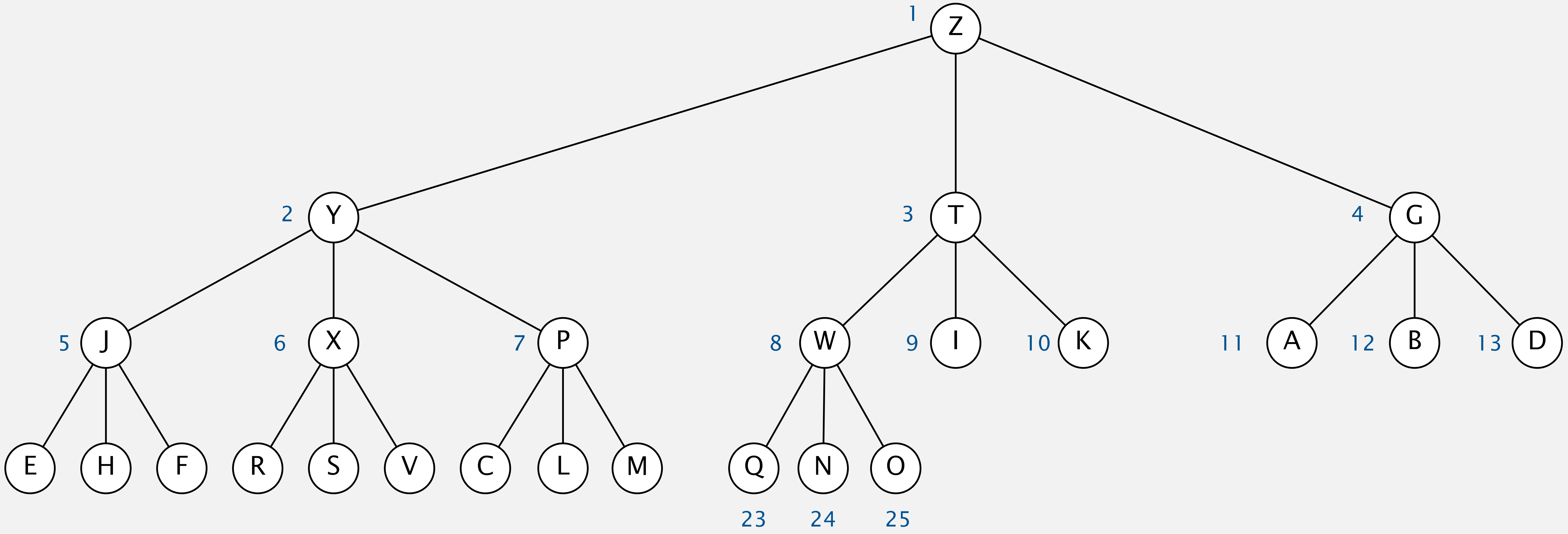
Multiway heaps

Multiway heaps.

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

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

Property. Children of key at index k are at indices $3k - 1$, $3k$, and $3k + 1$.



3-way heap

Priority queues: quiz 4

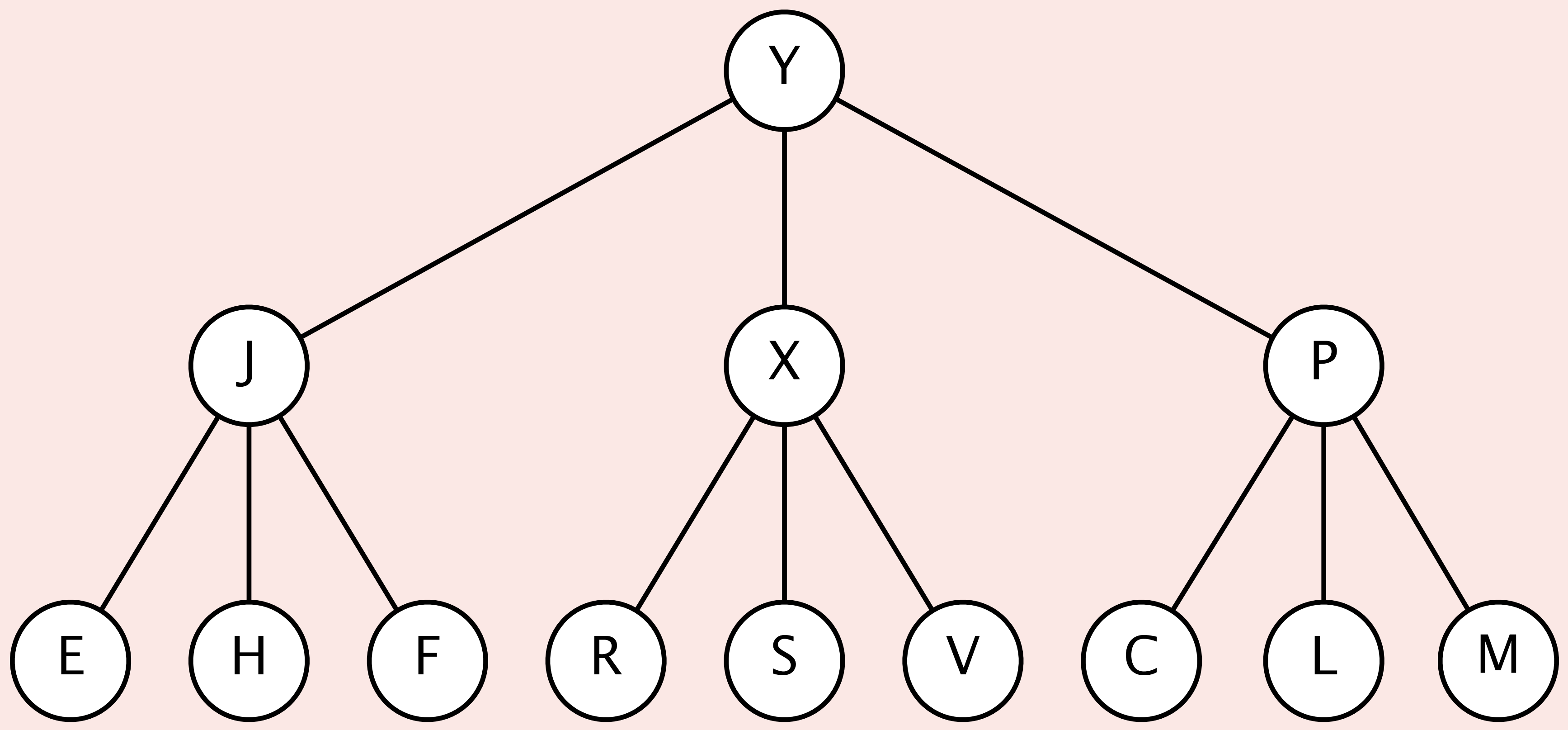
In the worst case, how many compares to INSERT and DELETE-MAX in a d-way heap as function of both n and d?

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 list	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
impossible	1	1	1	← why impossible?

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

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

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

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

B. In-place.

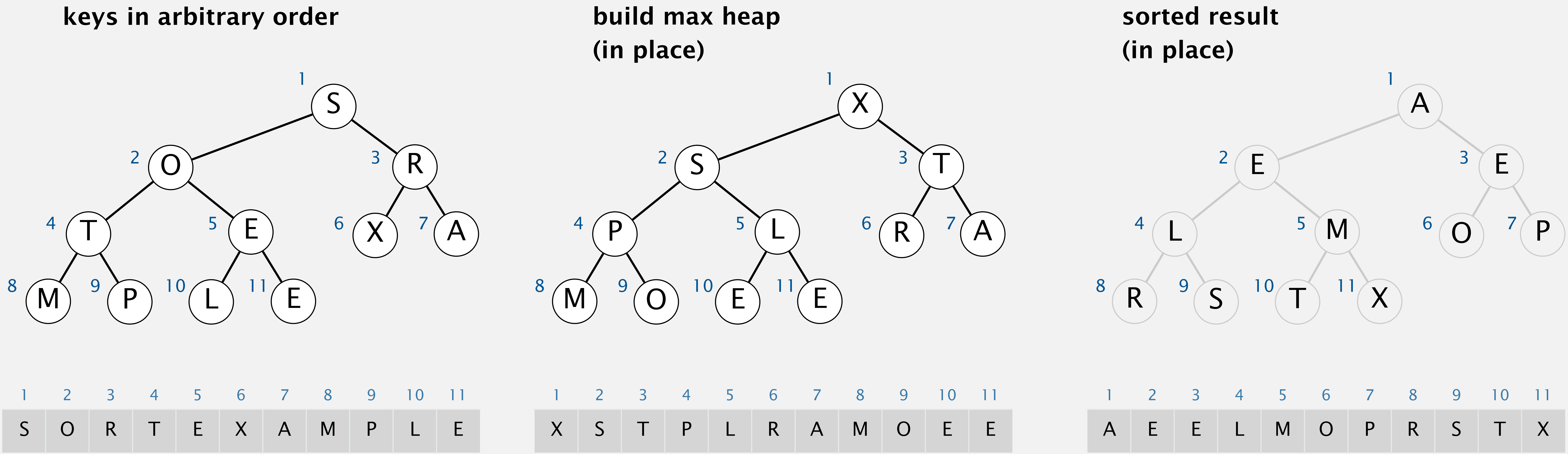
C. Stable.

D. *All of the above.*

Heapsort

Basic plan for in-place sort.

- View input array as a complete binary tree. ← we'll assume 1-indexed for now
- Heap construction: build a max-oriented heap.
- Sortdown: repeatedly remove the maximum key.



Heapsort: top-down heap construction

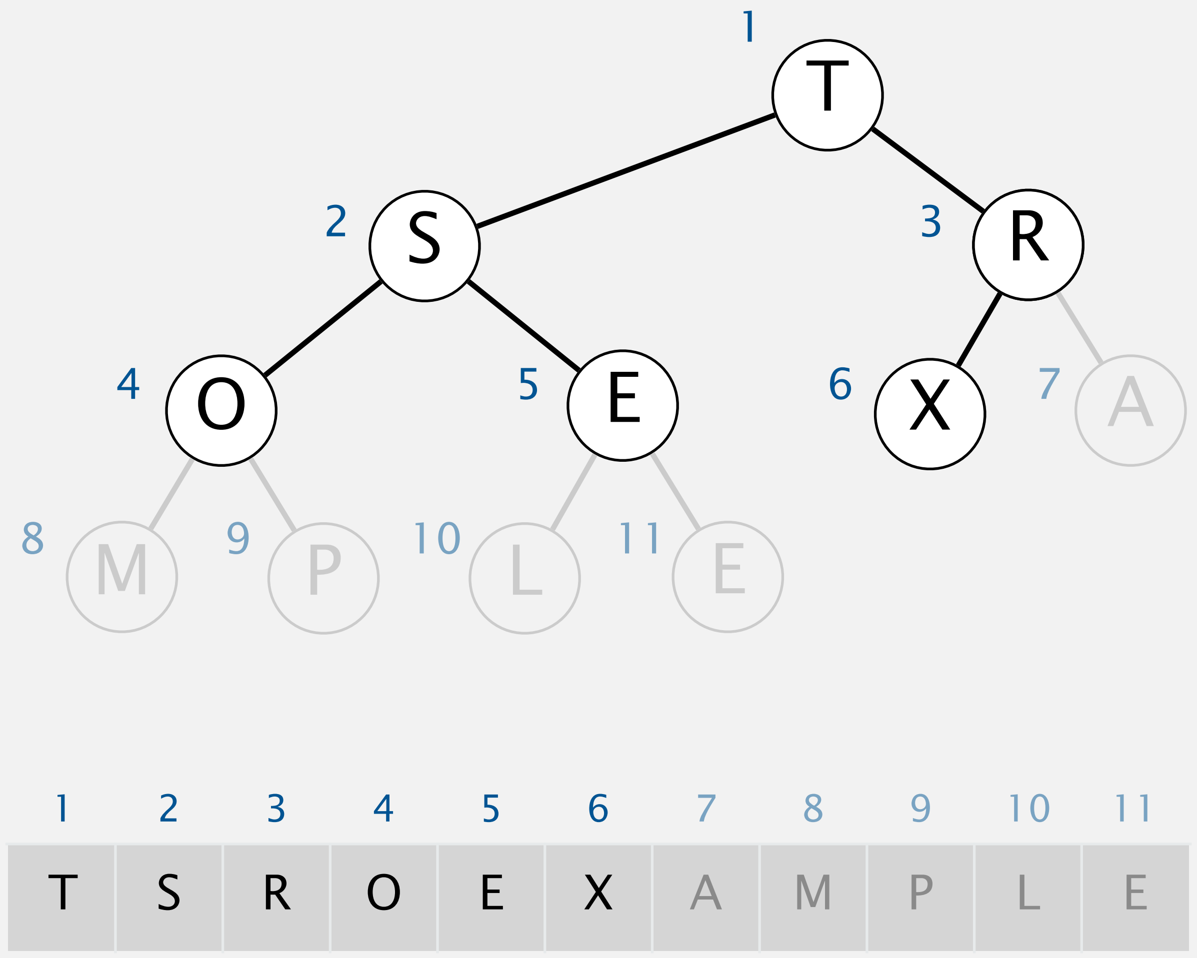
Top-down heap construction. Insert keys into a max heap, one at a time.

```
for (int k = 1; k <= n; k++)
   swim(a, k);
```

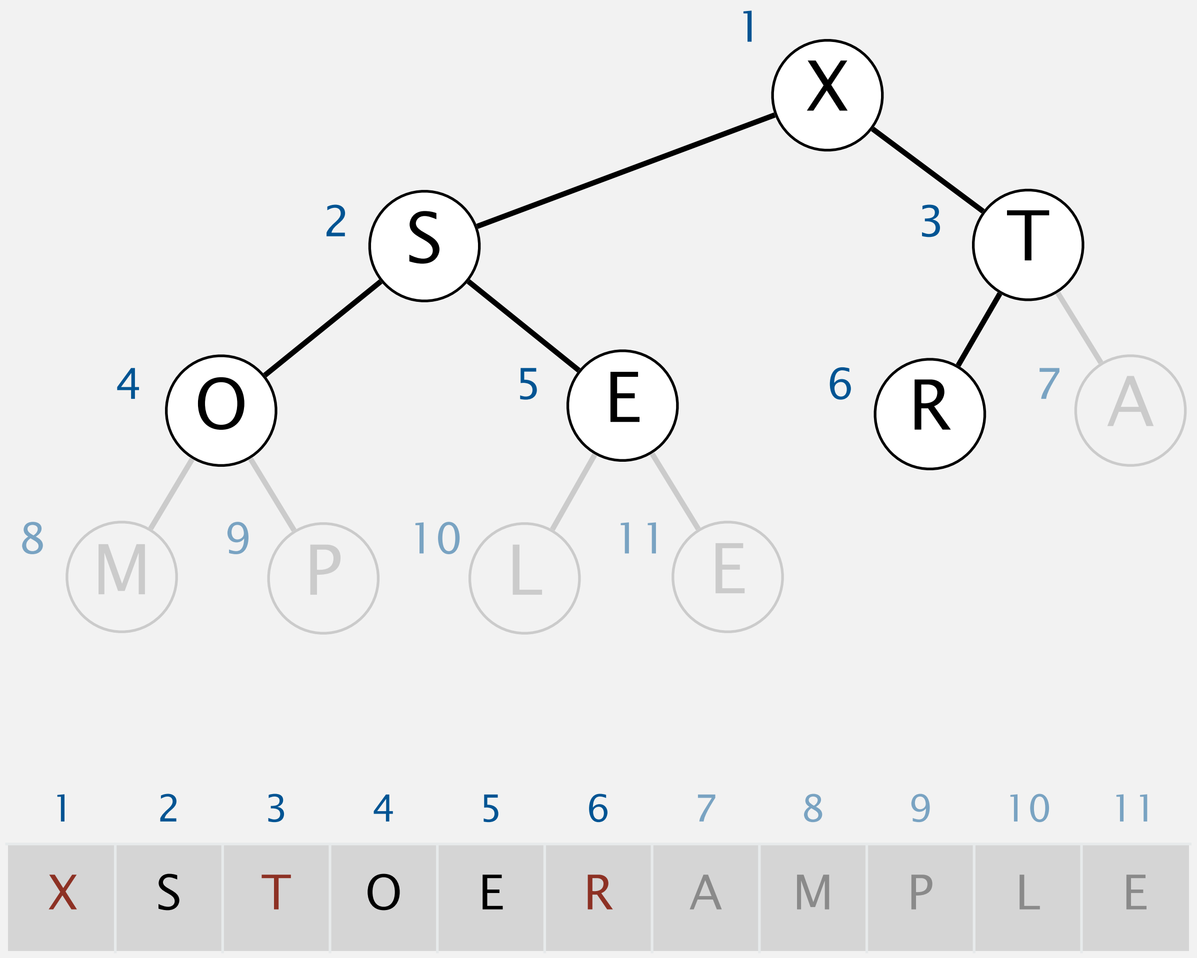
Invariants. After calling `swim(a, k)`,

- `a[1..k]` is a max heap.
- `a[k+1..n]` are untouched.

before call to swim(a, k)
k = 6



after call to swim(a, k)
k = 6



Heapsort: sortdown

Second pass.

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

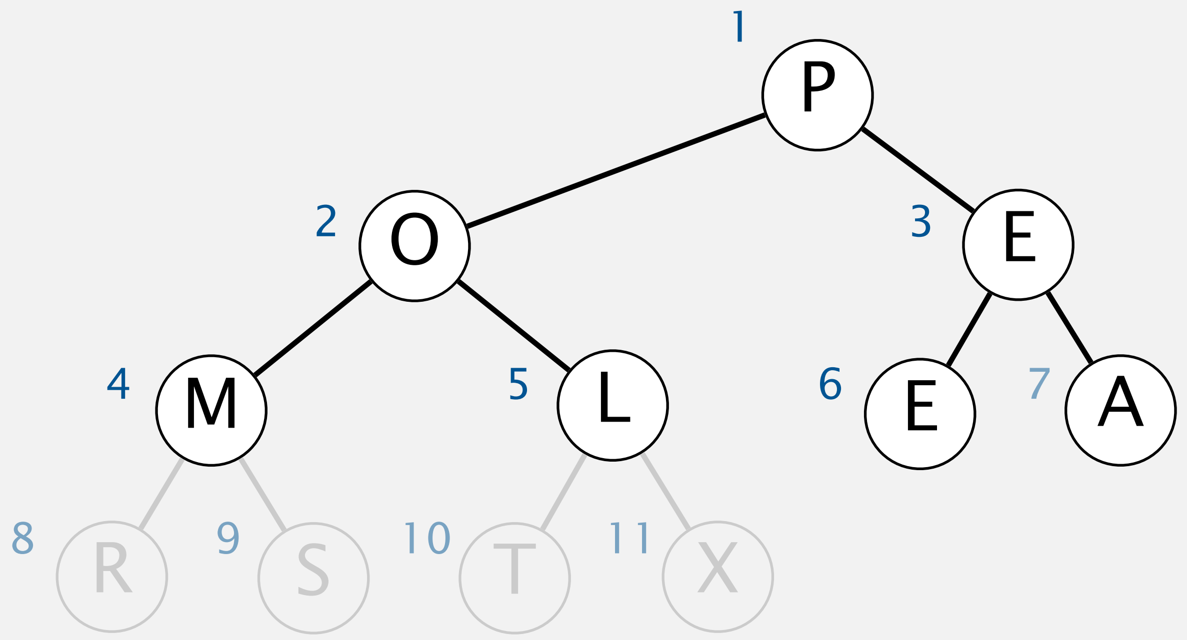
```
int k = n;
while (k > 1)
{
   exch(a, 1, k--);
   sink(a, 1, k);
}
```

delete-max
(but leave in array)

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

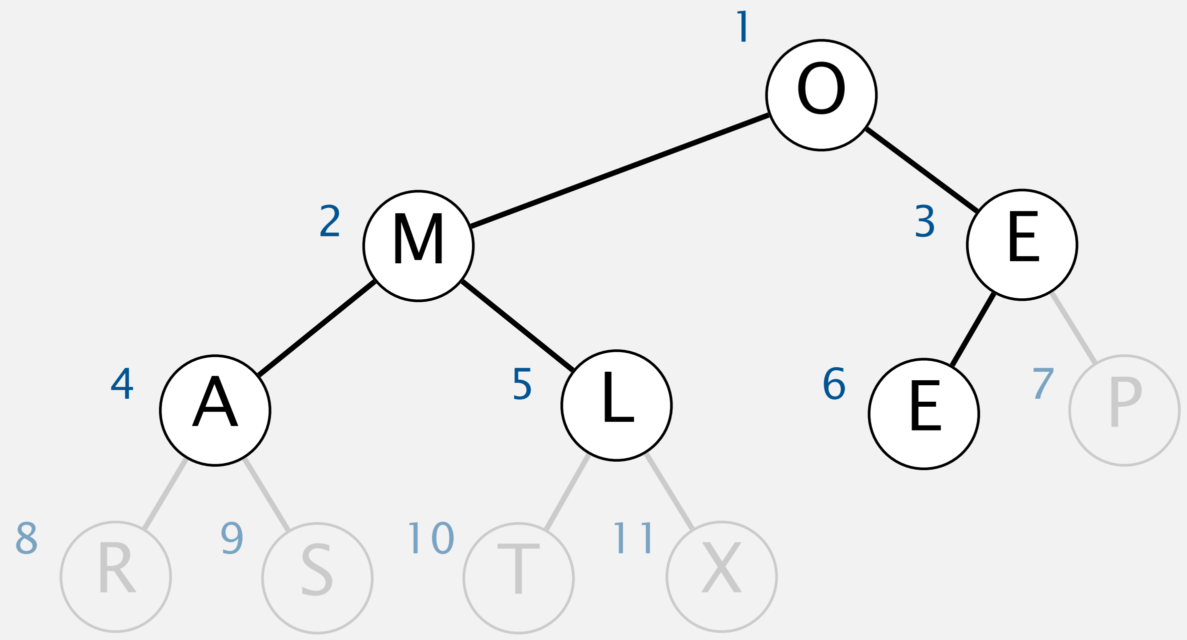
- `a[1..k-1]` is a max heap.
- `a[k..n]` are in final sorted order.

before call to sink(a, 1, k)
k = 7



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

after call to sink(a, 1, k)
k = 7



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

Heapsort: Java implementation

```
public class HeapTopDown
{
   public static void sort(Comparable[] a)
   {
      // top-down heap construction
      int n = a.length;
      for (int k = 1; k <= n; k--)
         swim(a, k);

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

   ...
}
```

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

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

private static void swim(Comparable[] a, int k)
{  /* 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

Heapsort: mathematical analysis

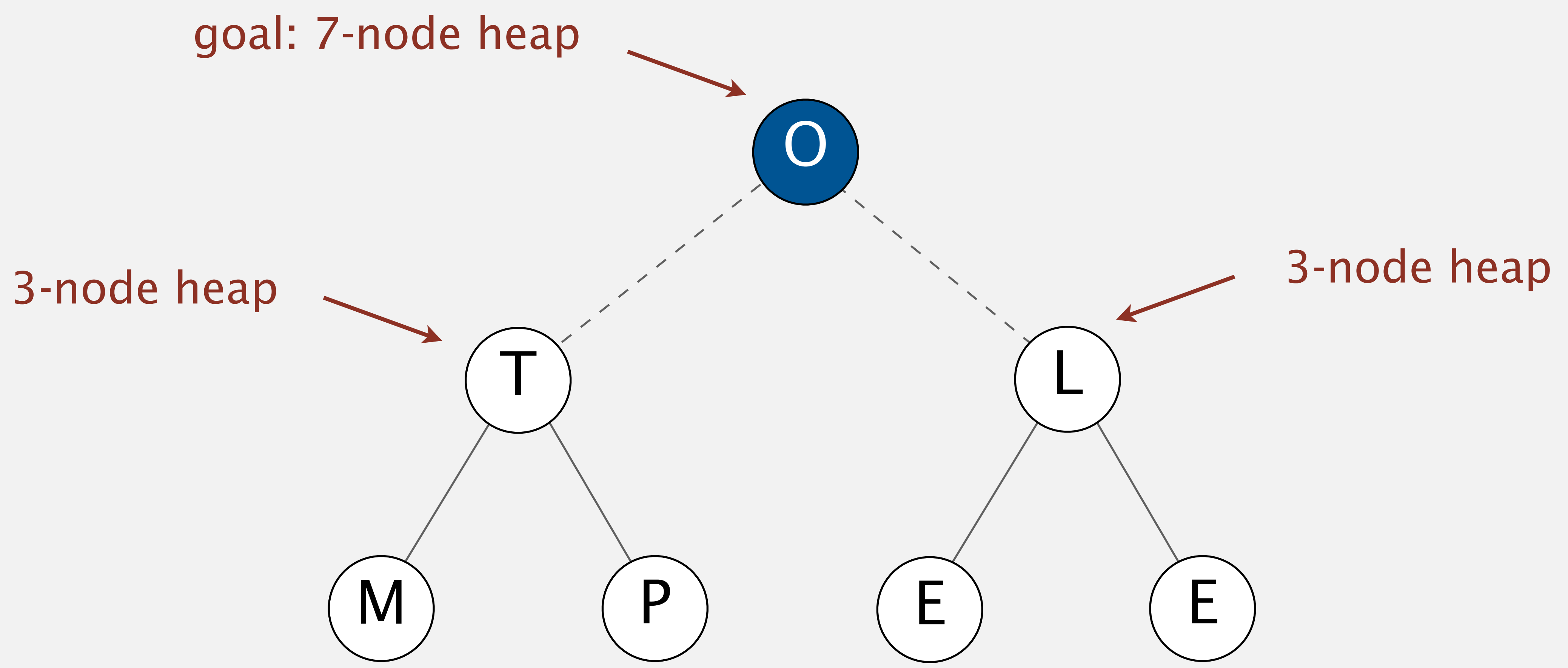
Proposition. Heapsort uses only $\Theta(1)$ extra space.

Proposition. Heapsort makes $\le 3\, n \log_2 n$ compares (and $\le 2\, n \log_2 n$ exchanges).

- Top-down heap construction: $\le n \log_2 n$ compares (and exchanges).
- Sortdown: $\le 2n \log_2 n$ compares (and $\le n \log_2 n$ exchanges).

Bottom-up heap construction. [see book] Successively building larger heap from smaller ones.

Proposition. Makes $\le 2\, n$ compares (and $\le n$ exchanges).



Heapsort: context

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:

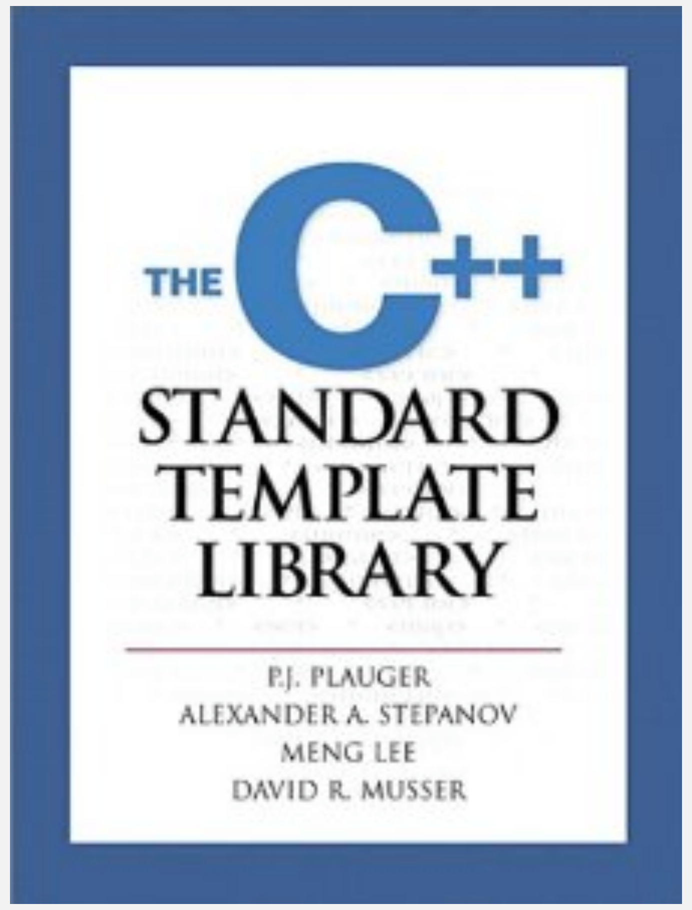
- Inner loop longer than quicksort's.
- Makes poor use of cache. ← can be improved using advanced caching tricks
- Not stable.

Introsort

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

Introsort.

- Run quicksort.
- Cutoff to heapsort if function-call stack depth exceeds $2 \log_2 n$.
- Cutoff to insertion sort for $n \leq 16$.



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

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		✔	$\frac{1}{2} n \log_2 n$	$n \log_2 n$	$n \log_2 n$	$\Theta(n \log n)$ guarantee; stable
timsort		✔	n	$n \log_2 n$	$n \log_2 n$	improves mergesort when pre-existing order
quick	✔		$n \log_2 n$	$2 n \ln n$	$\frac{1}{2} n^2$	$\Theta(n \log n)$ probabilistic guarantee; fastest in practice
3-way quick	✔		n	$2 n \ln n$	$\frac{1}{2} n^2$	improves quicksort when duplicate keys
heap	✔		$3 n$	$2 n \log_2 n$	$2 n \log_2 n$	$\Theta(n \log n)$ guarantee; in-place
?	✔	✔	n	$n \log_2 n$	$n \log_2 n$	holy sorting grail

number of compares to sort an array of n elements