
https://algs4.cs.princeton.edu

# 2.4 PRIORITY QUEUES

- *APIs*
- *elementary implementations*
- *binary heaps*
- *heapsort*


https://algs4.cs.princeton.edu

# 2.4 Priority Queues

- *APIs*
- *elementary implementations*
- *binary heaps*
- *heapsort*

# 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* <br> *resizable array* |
| **queue** | ENQUEUE, DEQUEUE | |
| **deque** | ADD-FIRST, REMOVE-FIRST, ADD-LAST, REMOVE-LAST | *doubly linked list* <br> *resizable array* |
| **priority queue** | INSERT, DELETE-MAX | *binary heap* |
| **symbol table** | PUT, GET, DELETE | *binary search tree* <br> *hash table* |
| **set** | ADD, CONTAINS, DELETE | |

# Priority queue

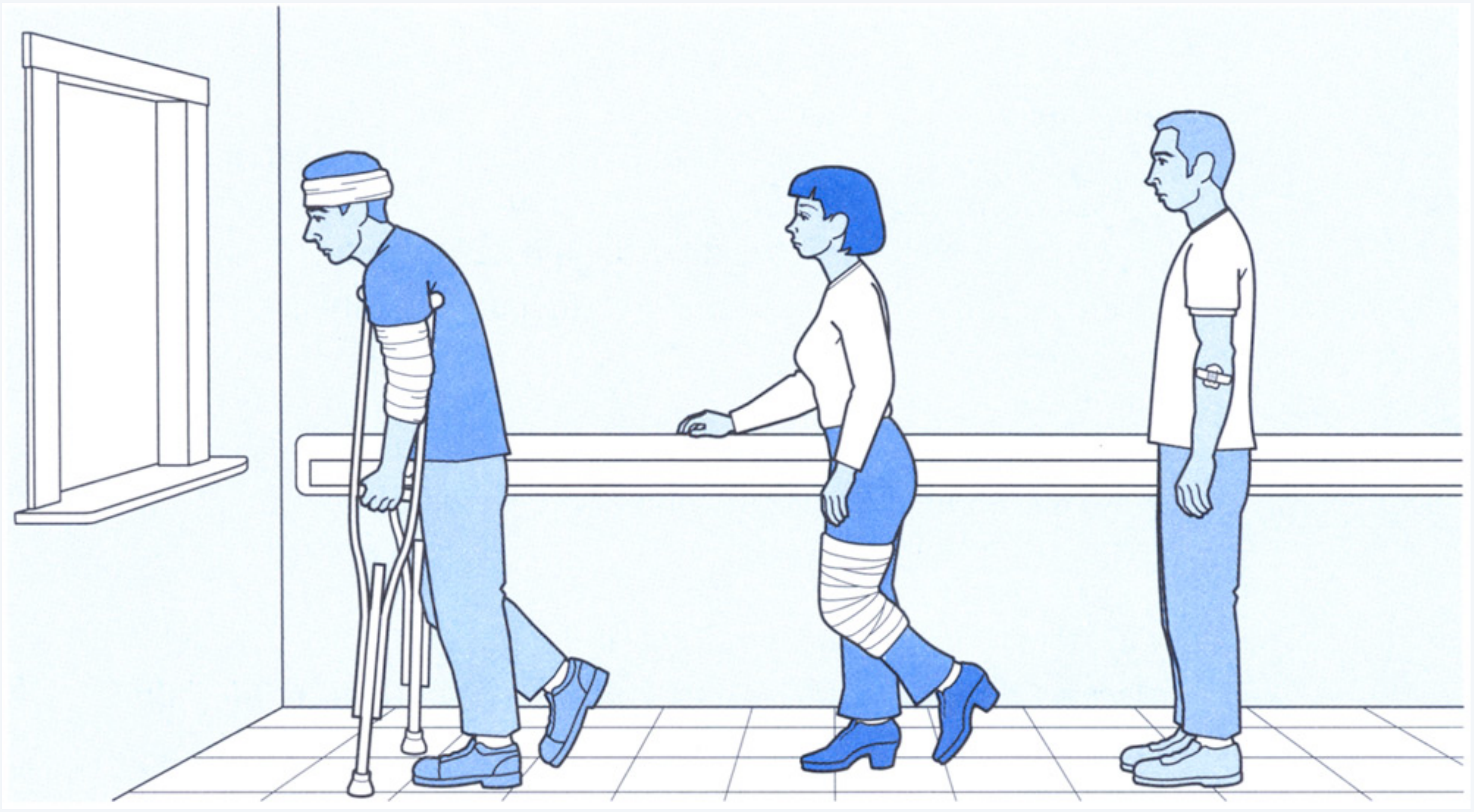
Collections. Insert and remove items. Which item to remove?

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.



**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

*"bounded type parameter"*

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

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

Note 1. Keys are generic, but must be `Comparable`.

Note 2. Duplicate keys allowed; `delMax()` removes and returns any largest key.

Performance goal. All ops take $O(\log n)$ time; use $\Theta(n)$ space. ← *$n$ = # elements in PQ*

# Min-oriented priority queue API

Analogous to `MaxPQ`.

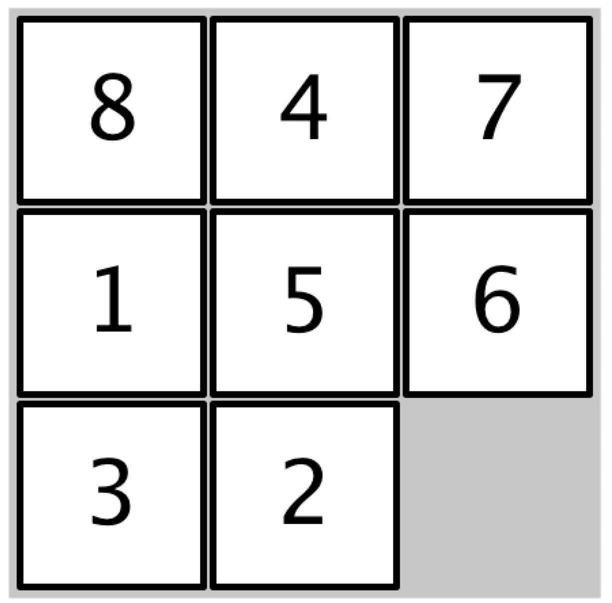
| public class MinPQ<Key extends Comparable<Key>> | |
|---|---|
| `MinPQ()` | *create an empty priority queue* |
| `void insert(Key key)` | *insert a key* |
| `Key delMin()` | *return and remove a smallest key* |
| `Key min()` | *return a smallest key* |
| `boolean isEmpty()` | *is the priority queue empty?* |
| `int size()` | *number of keys in the priority queue* |

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

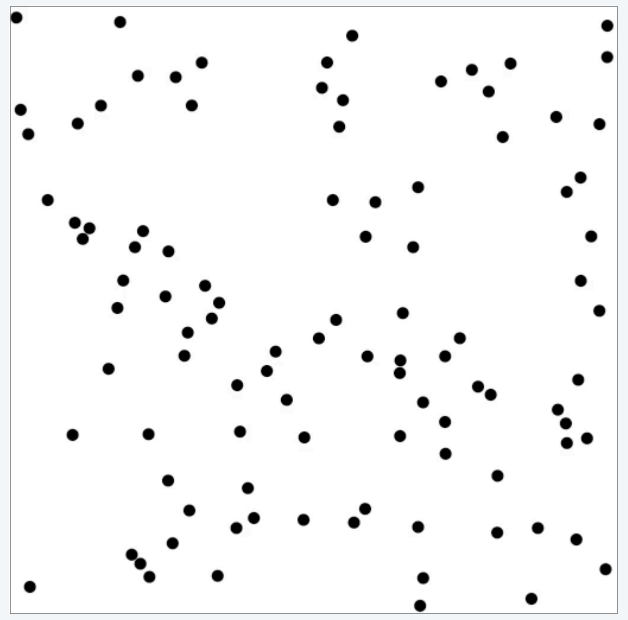
# Priority queue: applications

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


priority = length of best known path




priority = "distance" to goal board



priority = event time

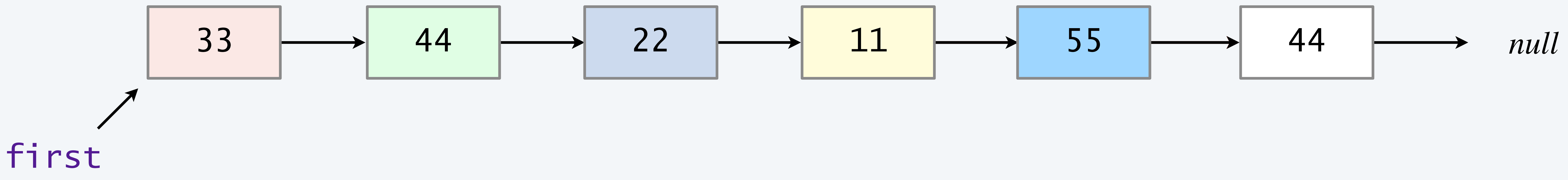

https://algs4.cs.princeton.edu

# 2.4 Priority Queues

- APIs
- *elementary implementations*
- binary heaps
- heapsort

# Priority queue: elementary implementations

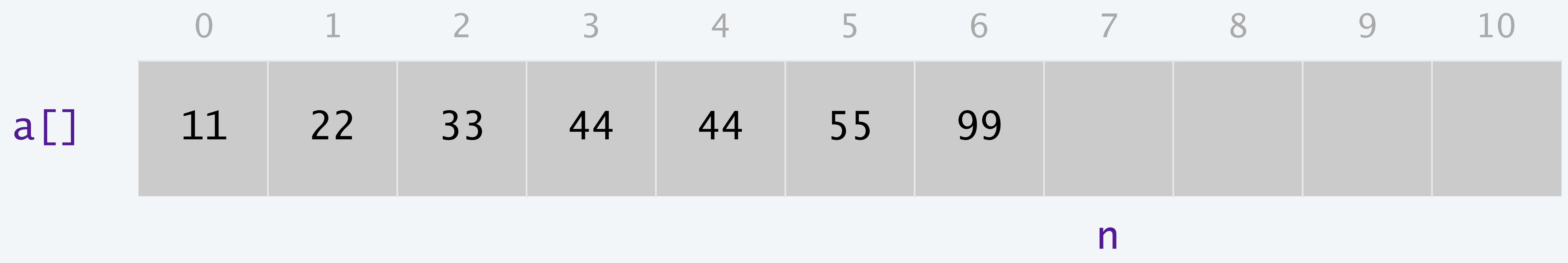
Unordered list. Store keys in a singly linked list.




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.




**ordered array implementation of a MaxPQ**

# Priority queues: poll 1

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

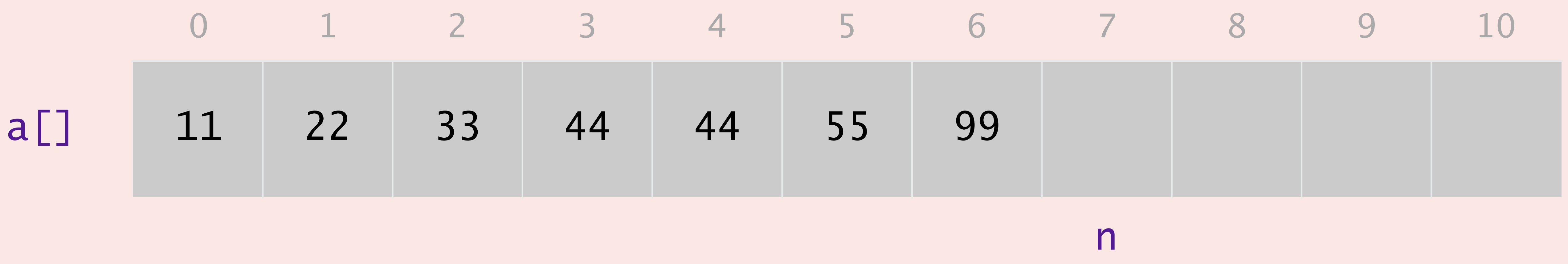
*ignore array resizing*

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

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

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

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




**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 | $\Theta(1)$ | $\Theta(n)$ |
| ordered array | $\Theta(n)$ | $\Theta(1)$ |
| goal | $\Theta(\log n)$ | $\Theta(\log n)$ |

worst-case running time for MaxPQ with n keys

Challenge. Implement both INSERT and DELETE–MAX efficiently.

Solution. "Somewhat-ordered" array.


https://algs4.cs.princeton.edu

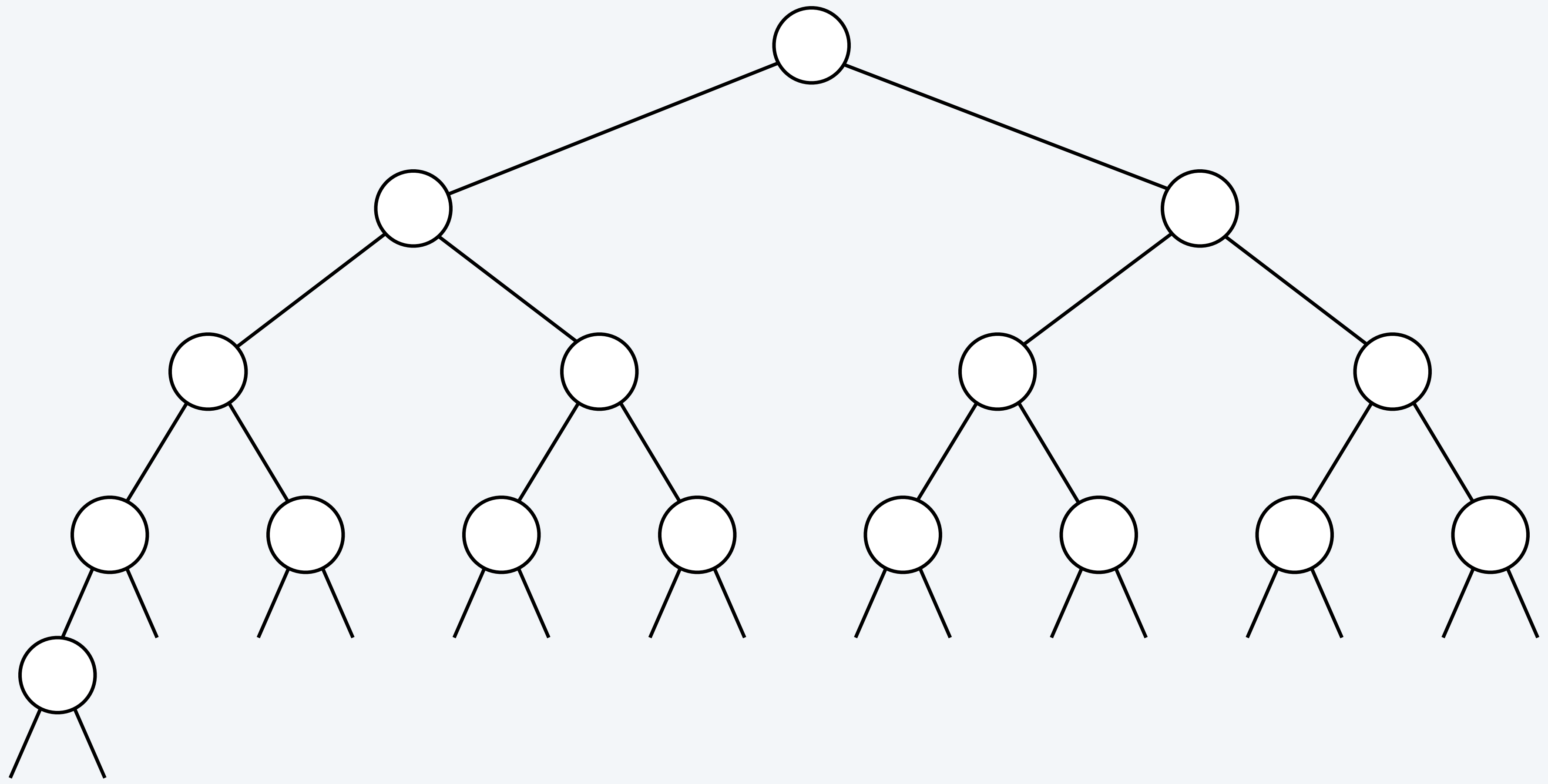
# 2.4 Priority Queues

- *APIs*
- *elementary implementations*
- *binary heaps*
- *heapsort*

# 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**
**(n = 16 nodes, height = 4)**

*floor function: largest integer* $\le \log_2 n$

**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 $2$.

# Priority queues:  poll 2

## Which is your favorite tree?

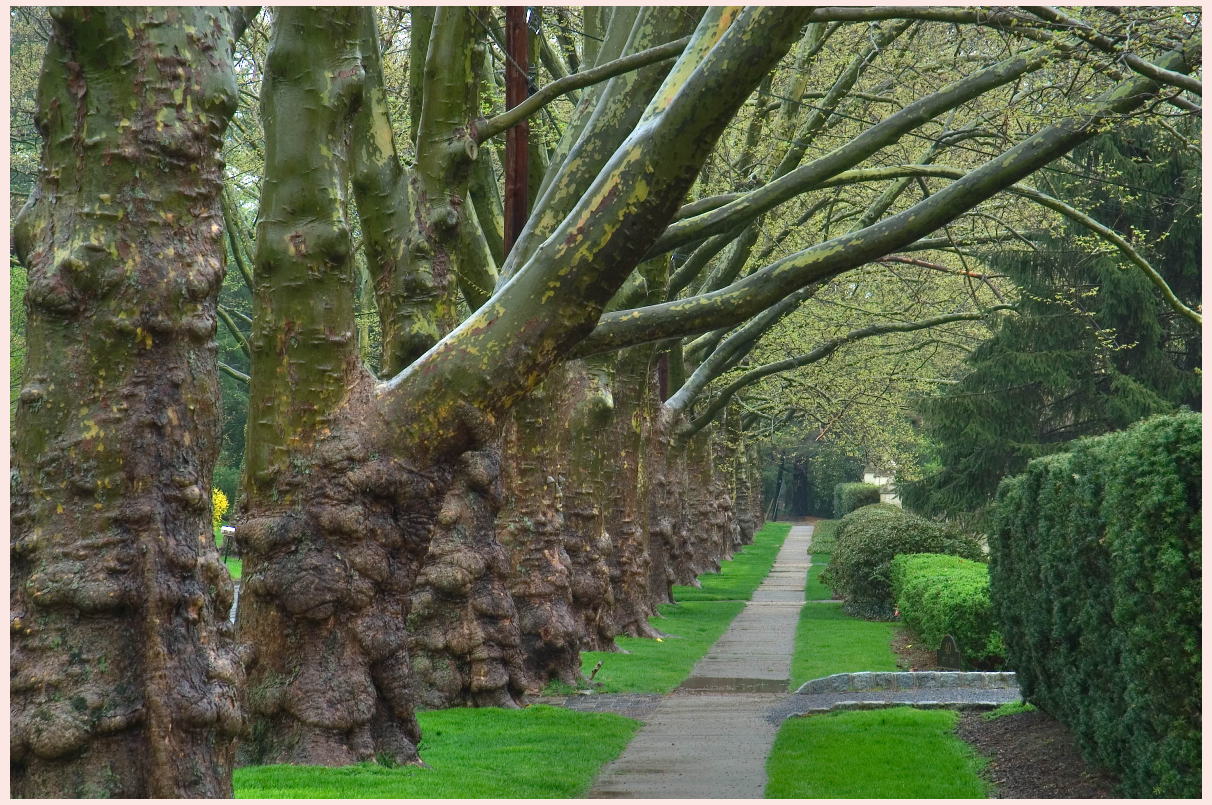
**A.**

**Joshua**

**B.**


**East African Doum Palm**

**C.**



**Sycamore**

**D.**

**Weirwood**

# A complete binary tree in nature (of height 4)

# Binary heap: representation

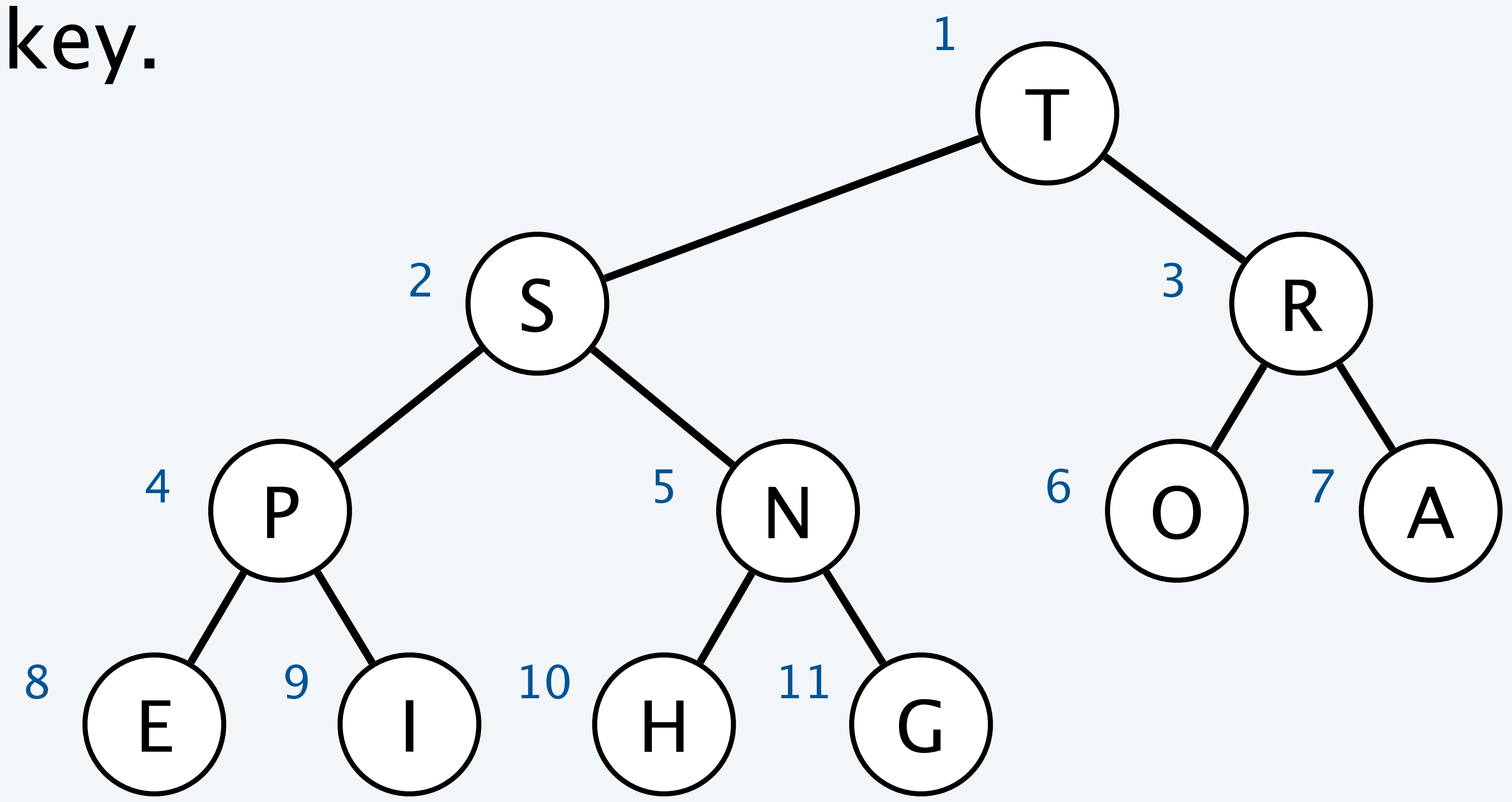
Binary heap. Array representation of a heap-ordered complete binary tree.

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: poll 3

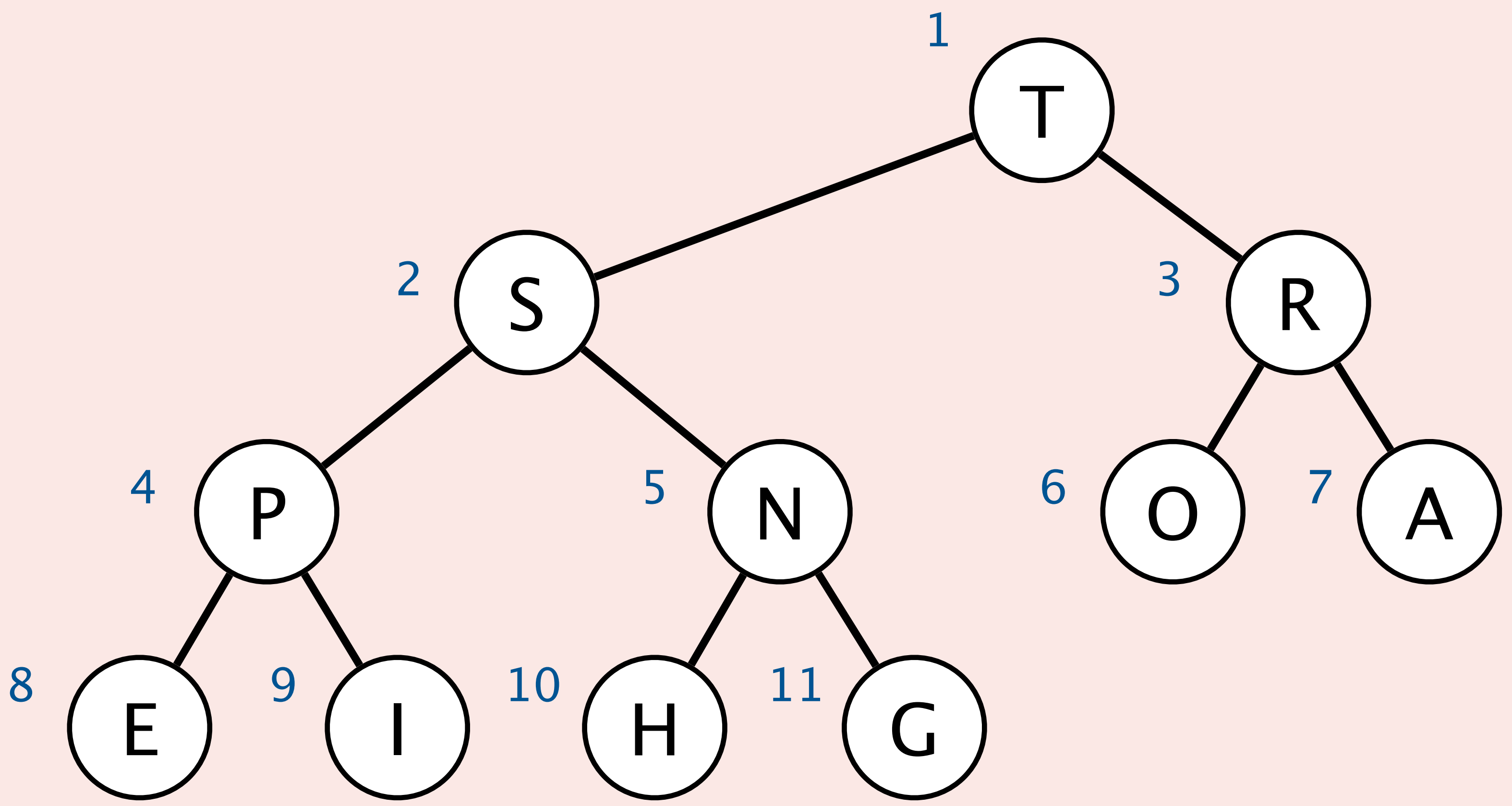
**Consider the node at index k in a binary heap. Which Java expression produces the index of its parent?**

**A.** `(k - 1) / 2`

**B.** `k / 2`

**C.** `(k + 1) / 2`

**D.** `2 * k`



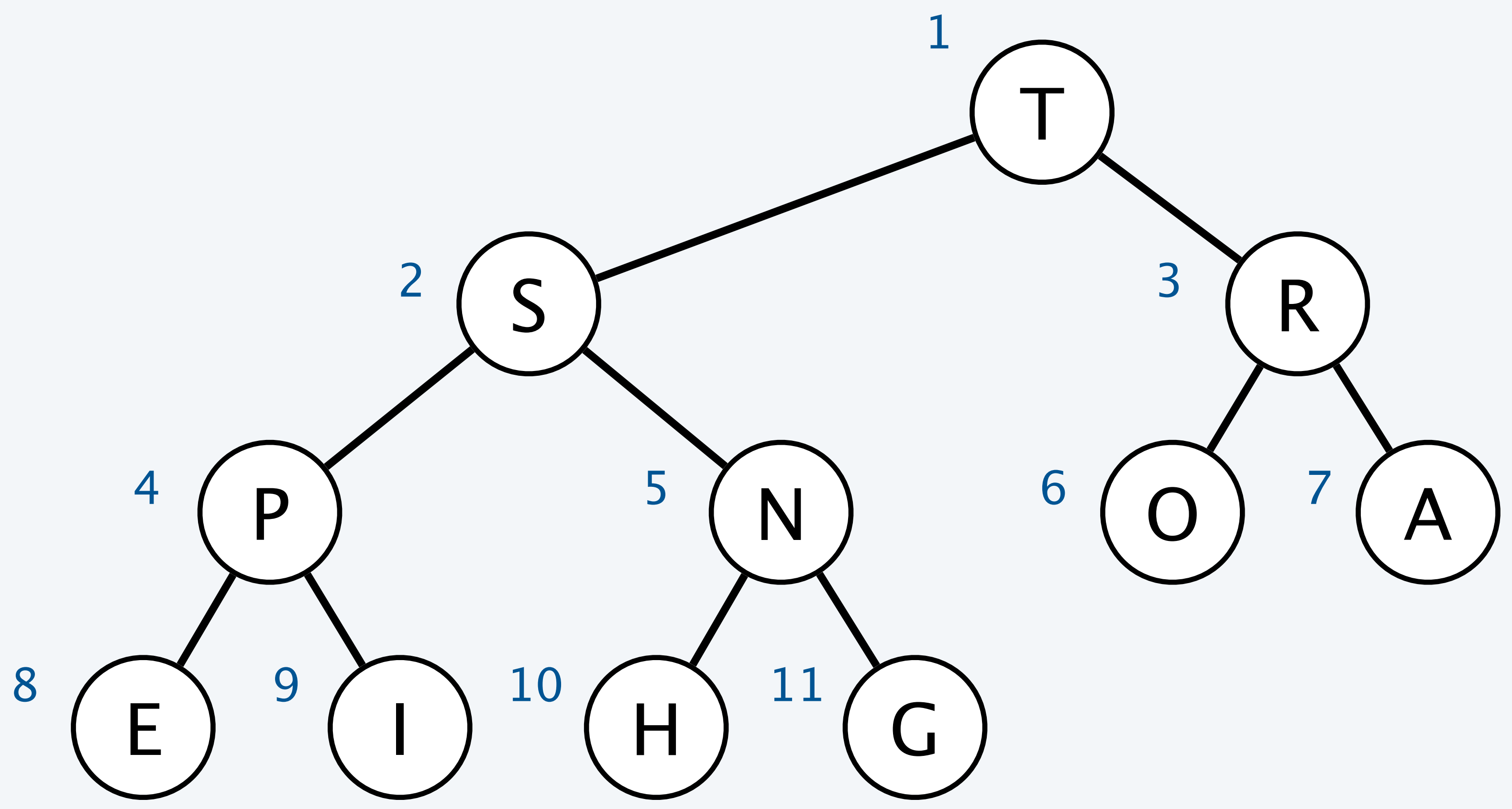

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a[] | - | T | S | R | P | N | O | A | E | I | H | G |

# Binary heap: properties

Proposition. Largest key is at index `1`, which corresponds to 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`.



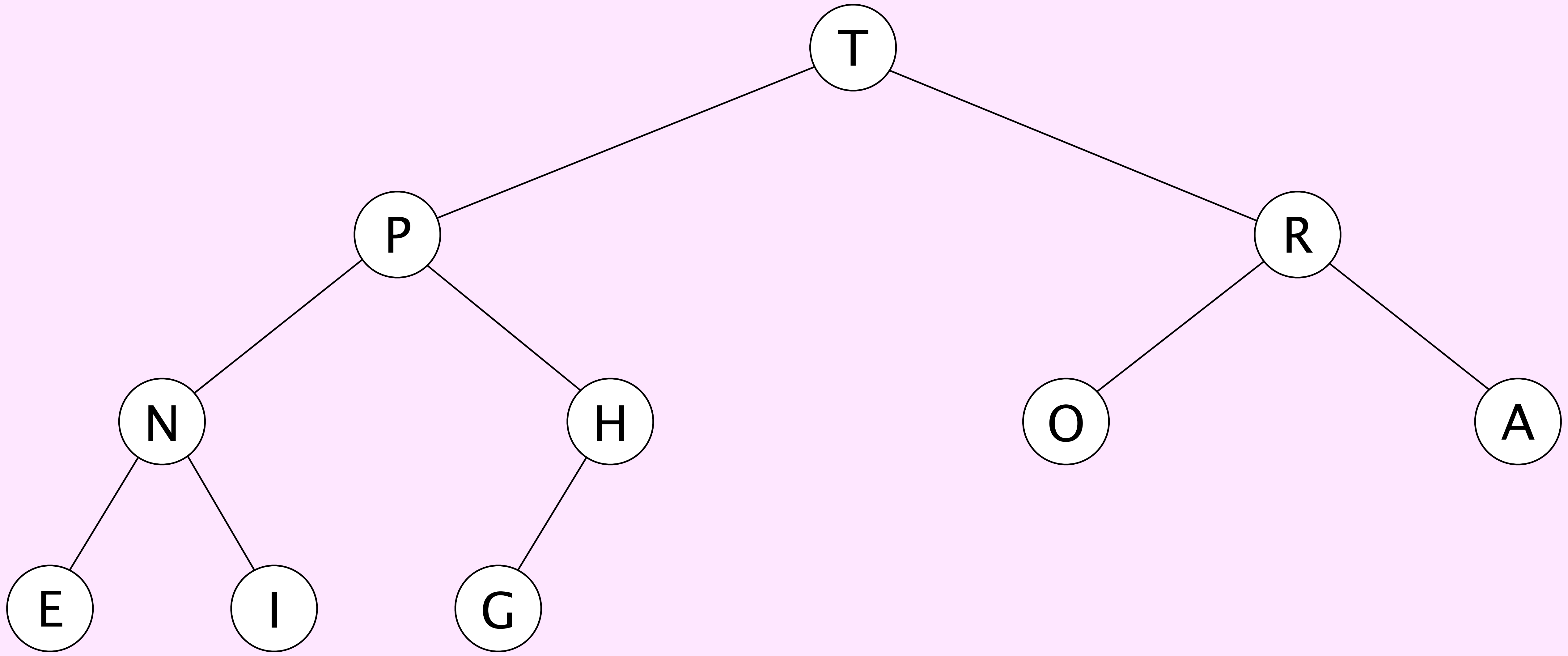

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a[] | - | T | S | R | P | N | O | A | E | I | H | G |

# Binary heap demo

**Insertion.** Create new node at end of bottom level, then swim it up.

**Deletion of the maximum.** Exchange key in root node with key in last node, then sink it down.

**heap ordered**




| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| T | P | R | N | H | O | A | E | I | G | |

# Binary heap: promotion

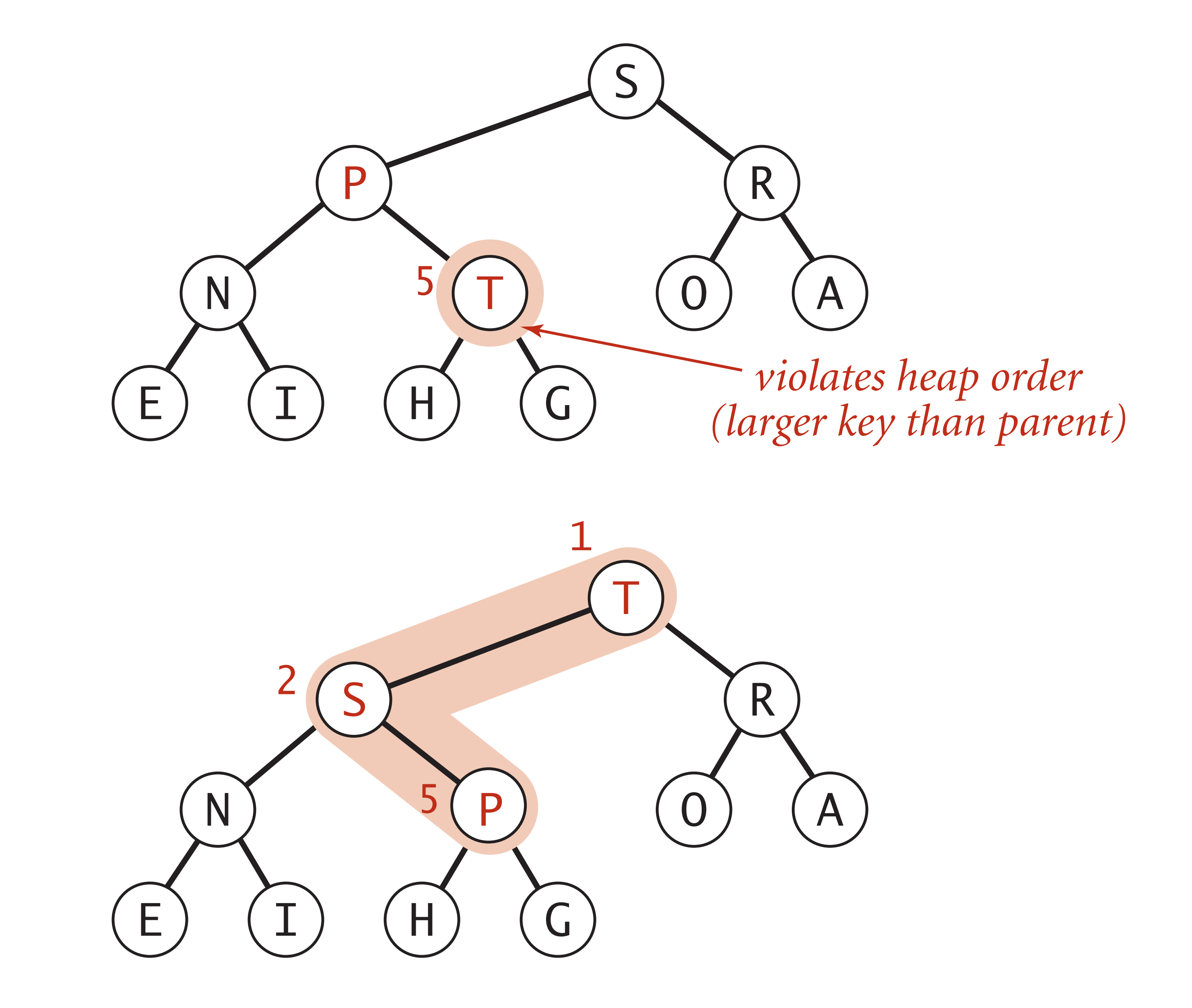
Scenario. Key in node becomes larger than key in parent's node.

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

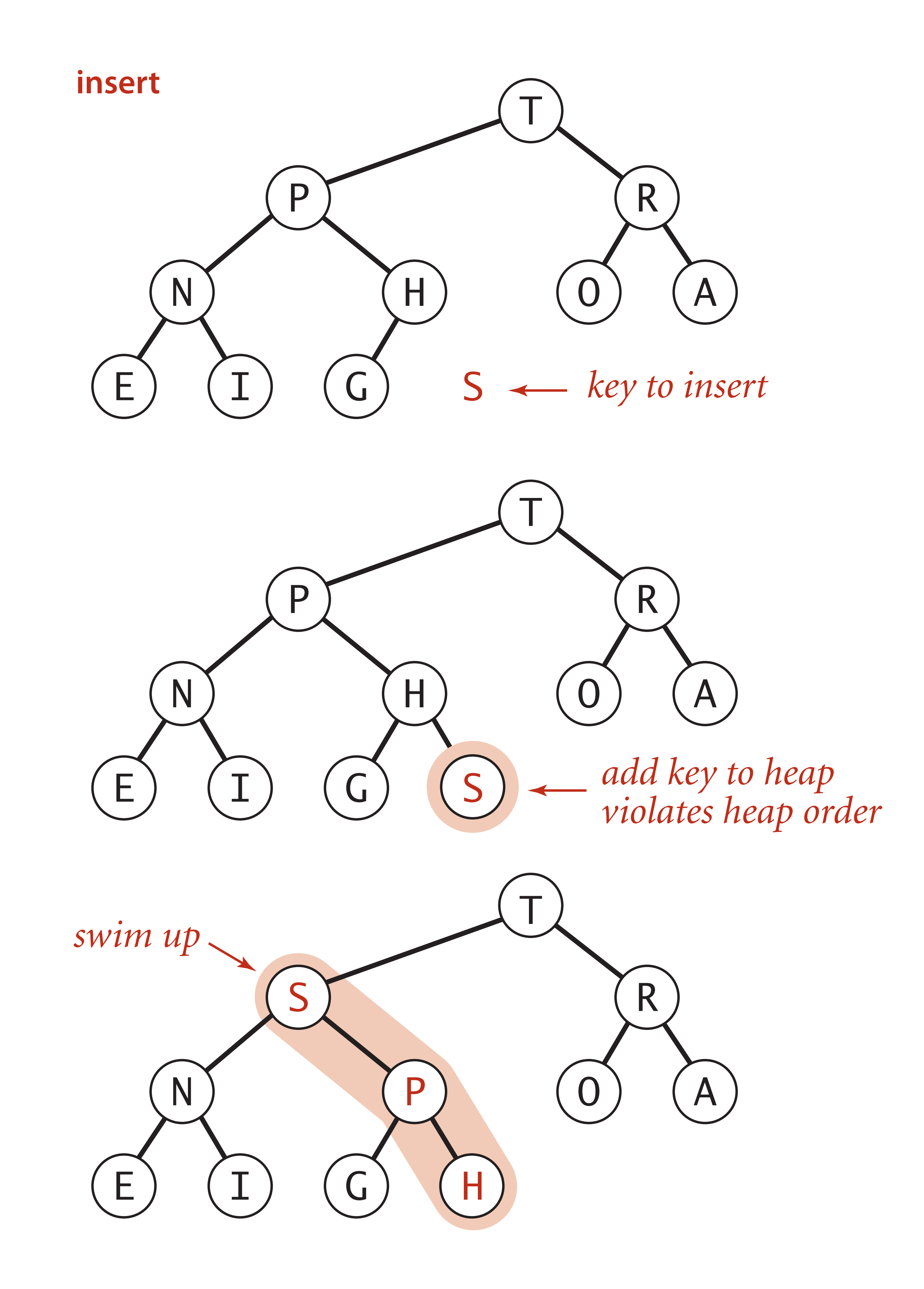


# Binary heap: insertion

Algorithm. Create new node at end of 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:

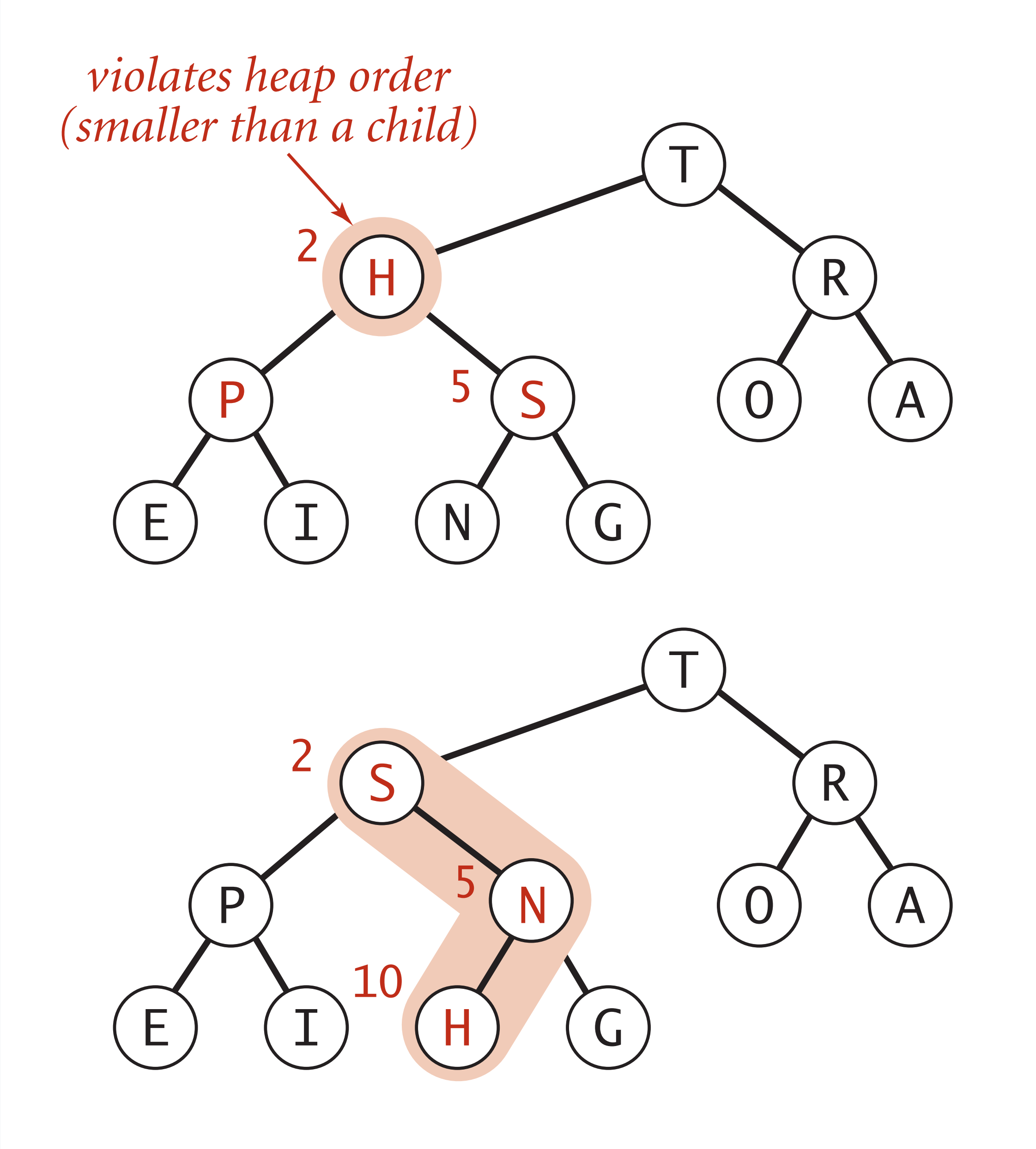
*why not smaller child ?*

- 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*

*j is now the index of the larger child*

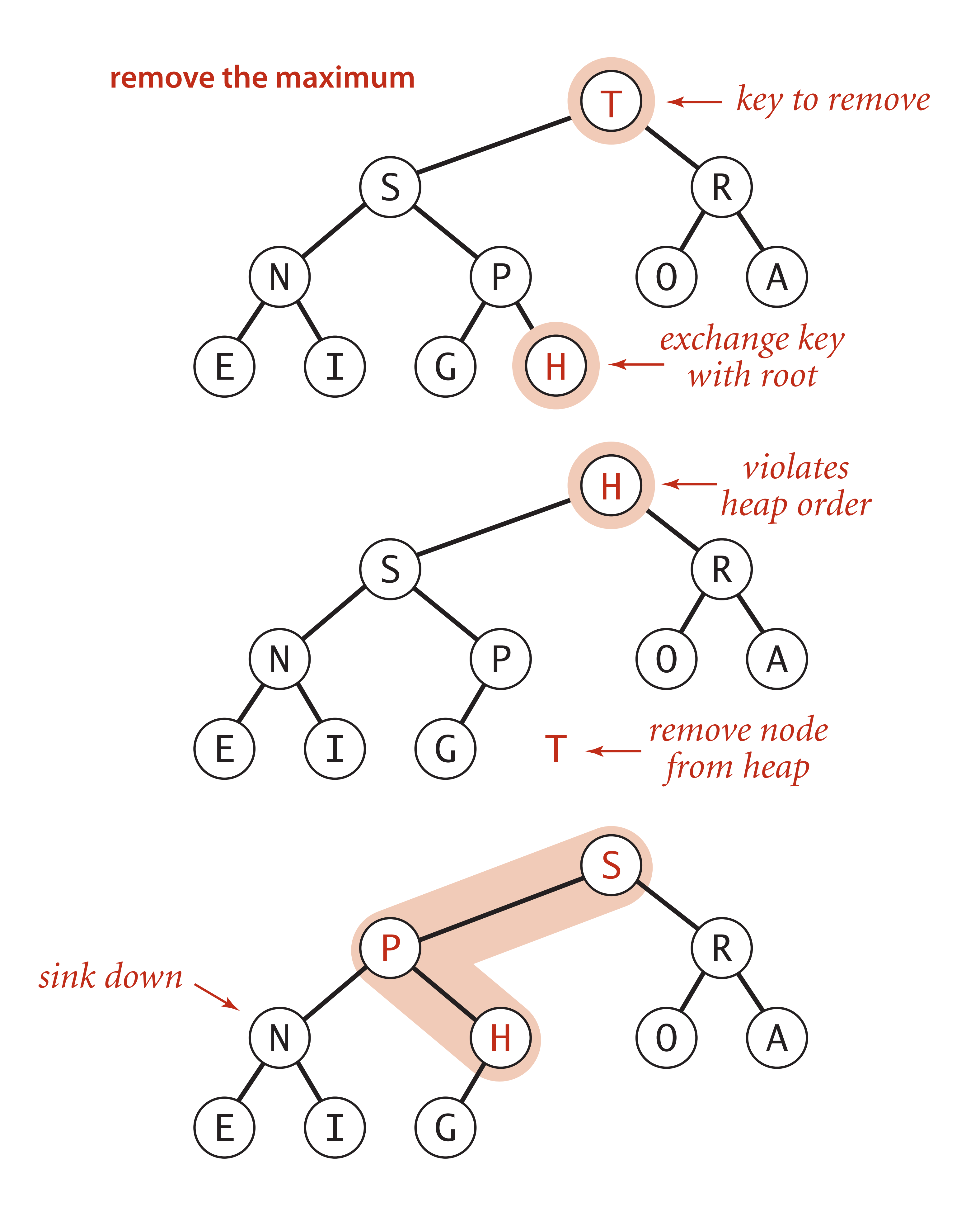


# Binary heap: deletion of the maximum

**Algorithm.** Exchange key in root node with key in last node, 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 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** | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ |
| **ordered array** | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ |
| **goal** | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(1)$ |

**worst-case running time for MaxPQ with n keys**

# Binary heap: considerations

## Underflow and overflow.

- Underflow: throw exception if deleting from empty PQ.
- Overflow: add no-arg constructor and use a resizable 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 heap operations.

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

*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

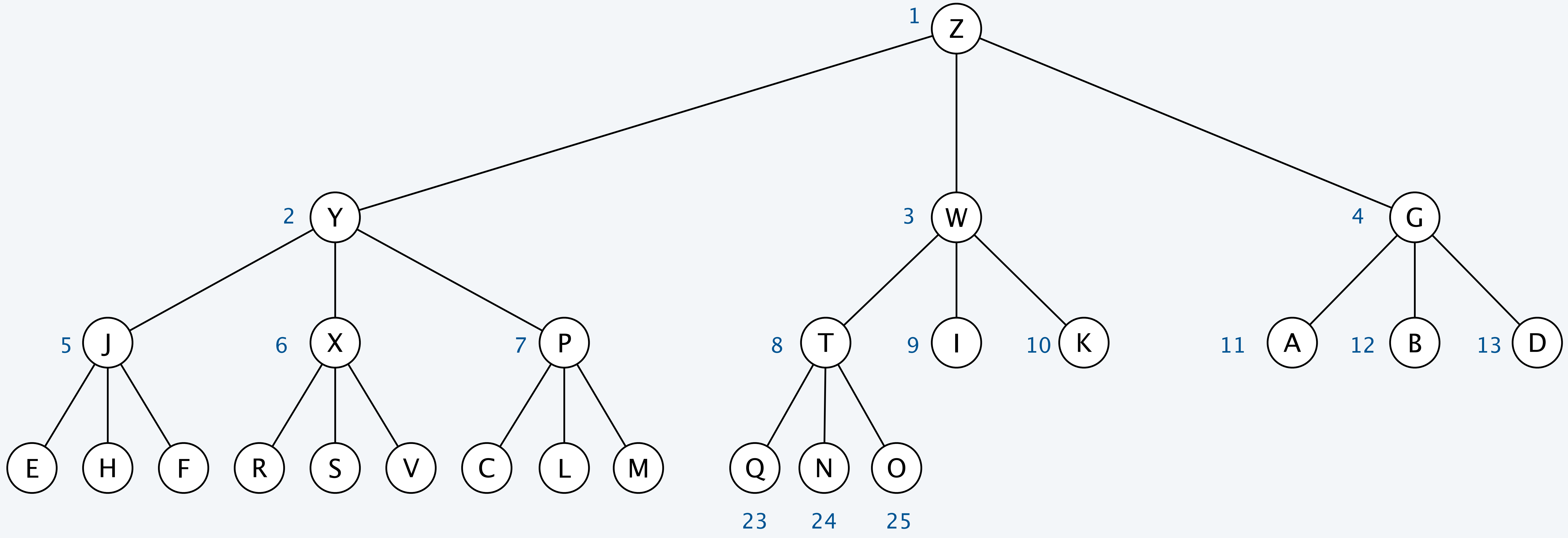
# Multiway heaps

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 $k$ at indices $3k - 1$, $3k$, and $3k + 1$; parent at index $\lfloor (k + 1) / 3 \rfloor$.




3-way heap

## Priority queues: poll 5

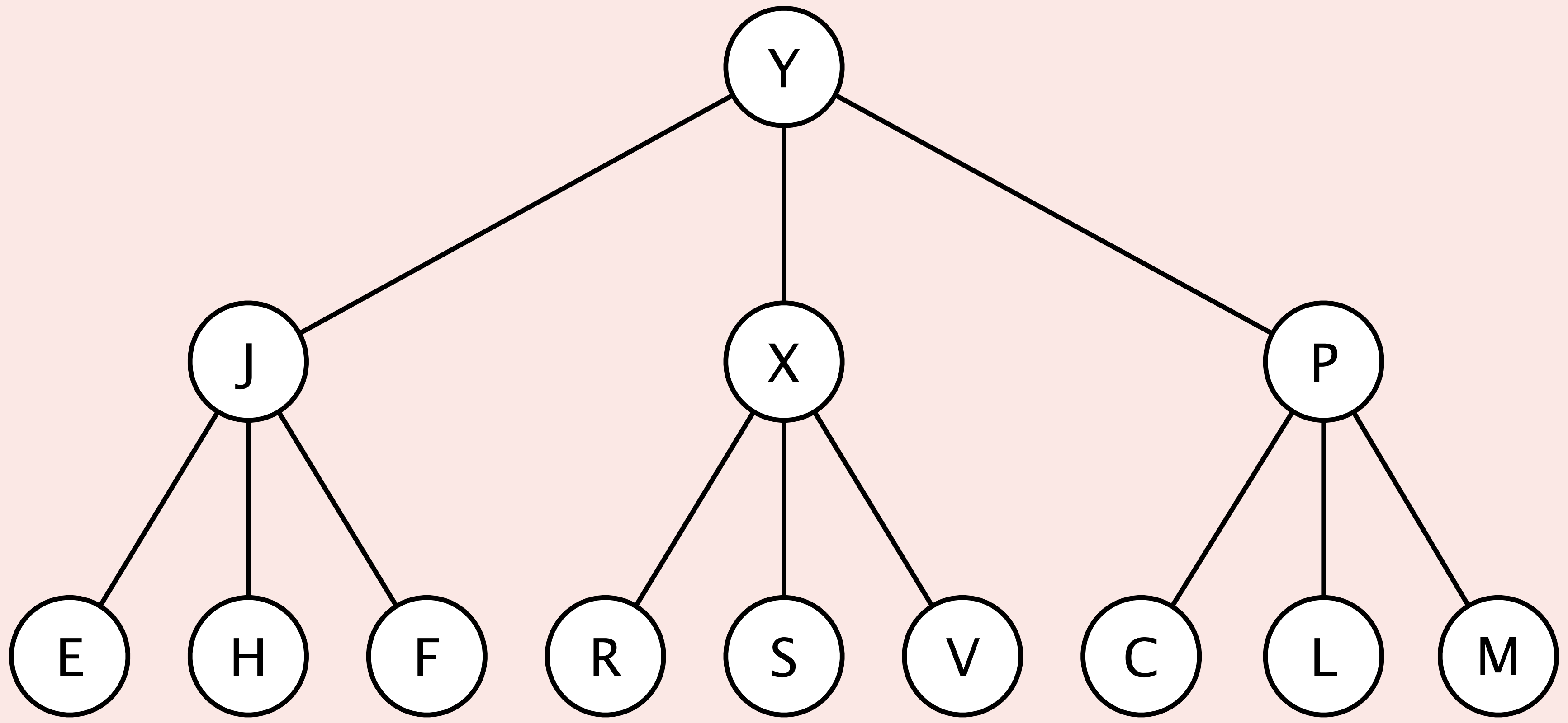
**In the worst case, how many compares to INSERT and DELETE-MAX in a $d$-way heap as function of both $n$ and $d$?**

**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 | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ | |
| ordered array | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ | |
| binary heap | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(1)$ | |
| d-ary heap | $\Theta(\log_d n)$ | $\Theta(d \log_d n)$ | $\Theta(1)$ | ← *sweet spot*: $d = 4$ |
| Fibonacci † | $\Theta(1)$ | $\Theta(\log n)$ | $\Theta(1)$ | ← *see* COS 423 |
| impossible | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ | ← *why impossible ?* |

worst-case running time for MaxPQ with n keys

† *amortized*


https://algs4.cs.princeton.edu

# 2.4 Priority Queues

- APIs
- elementary implementations
- binary heaps
- **heapsort**

**Which of the following are 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.

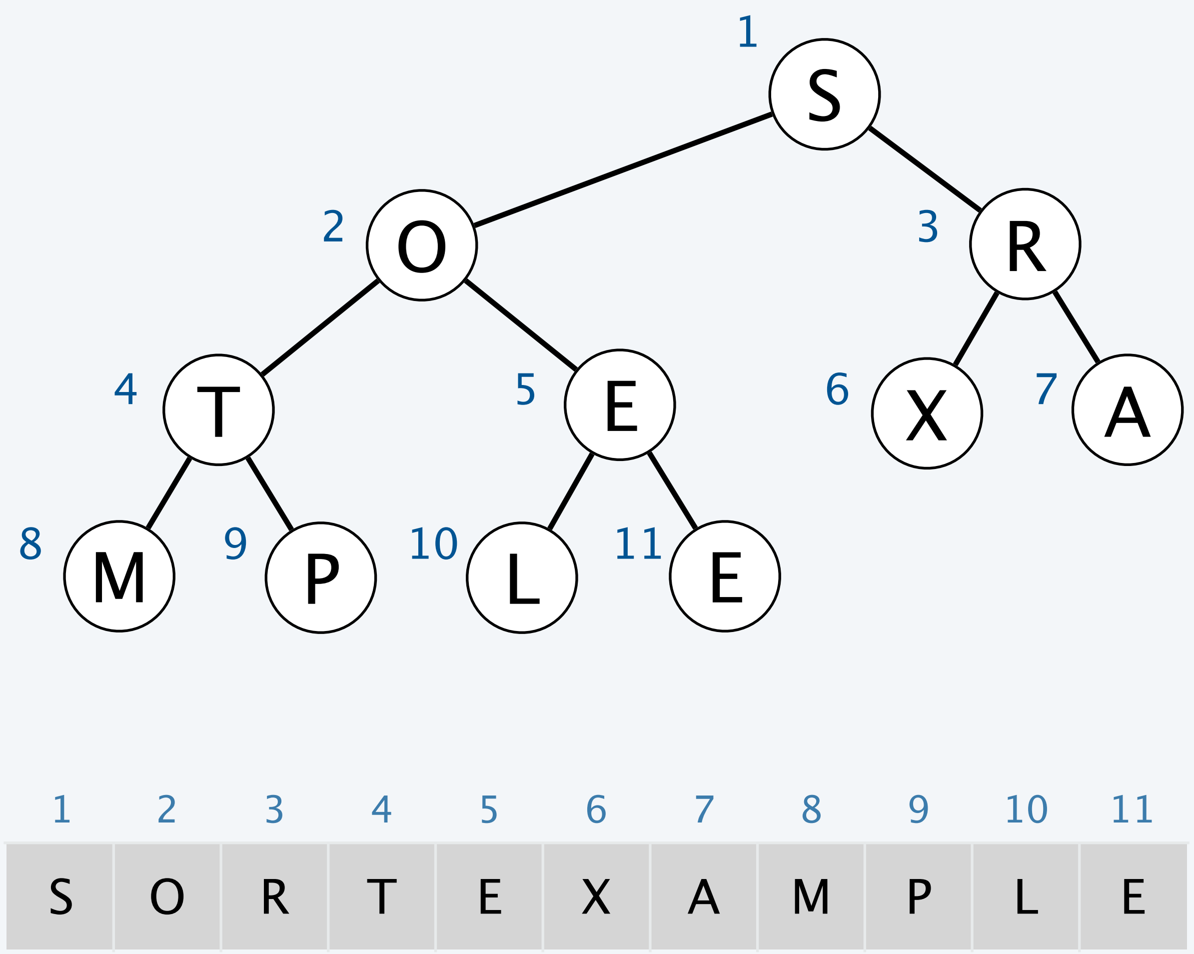
**D.** *All of the above.*

# Heapsort

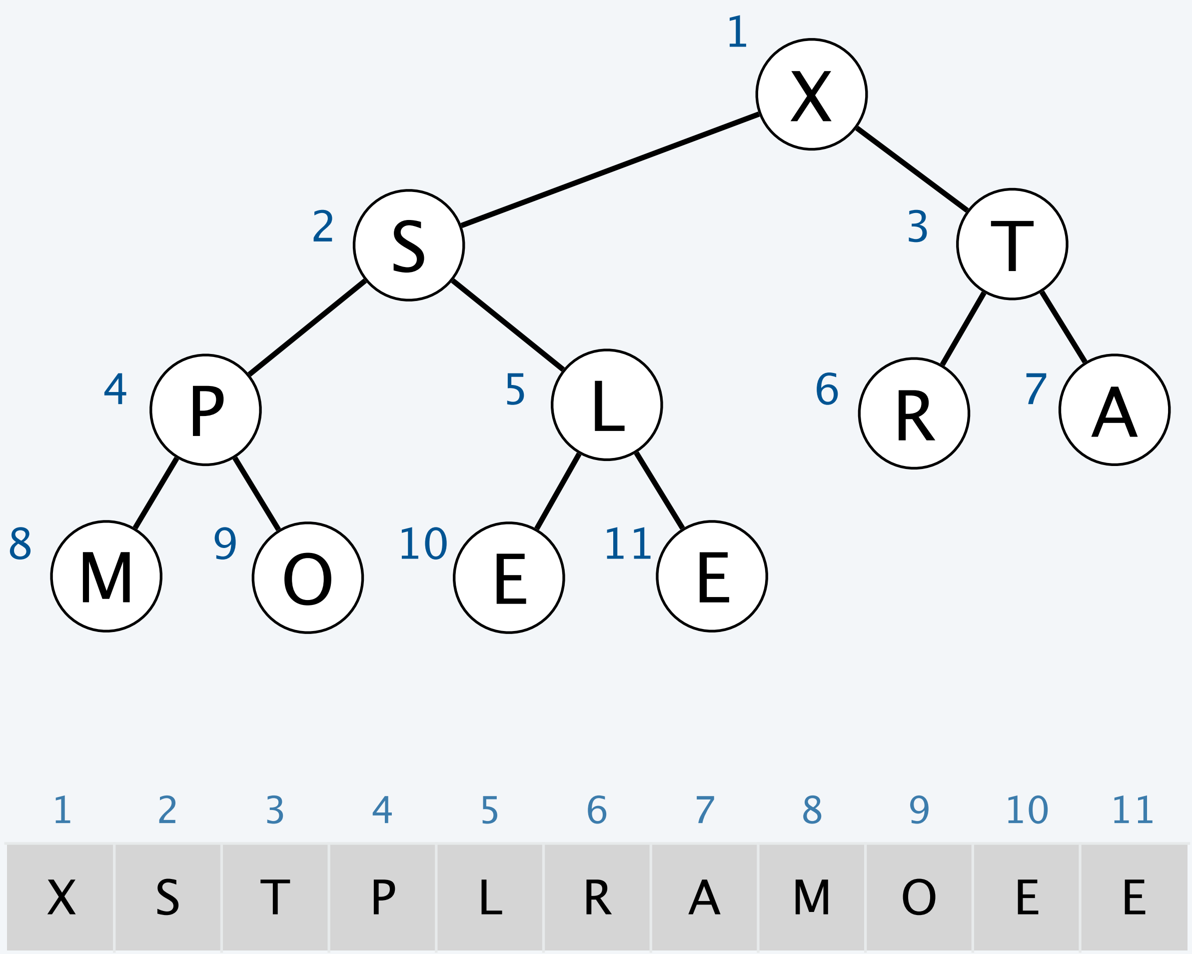
## Basic plan for in-place sort.

- View input array as a complete binary tree. ← *we'll assume 1-indexed for now*
- Phase 1 (heap construction): build a max-oriented heap.
- Phase 2 (sortdown): repeatedly remove the maximum key. ← *a version of selection sort*

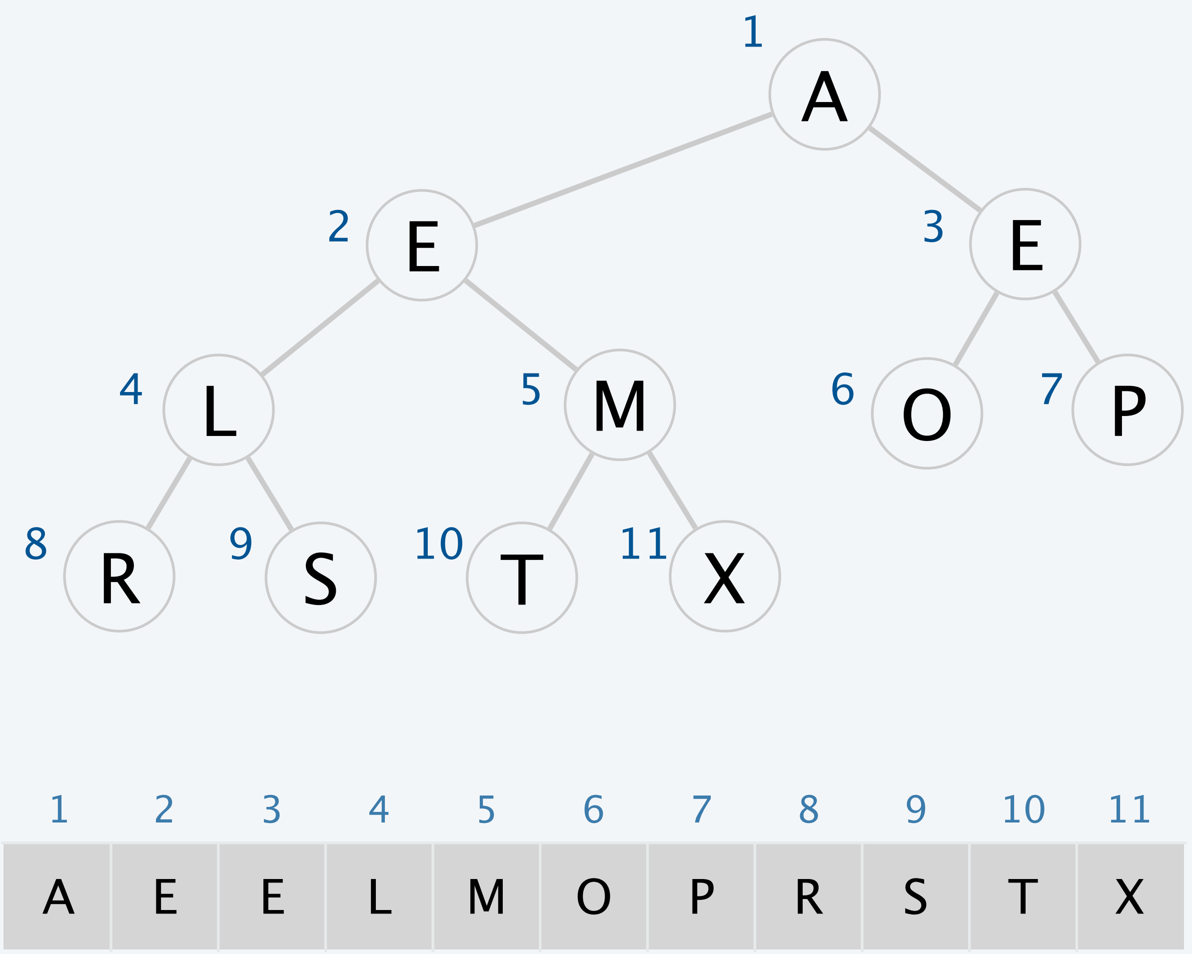
**keys in arbitrary order**




**build max heap (in place)**




**sorted result (in place)**

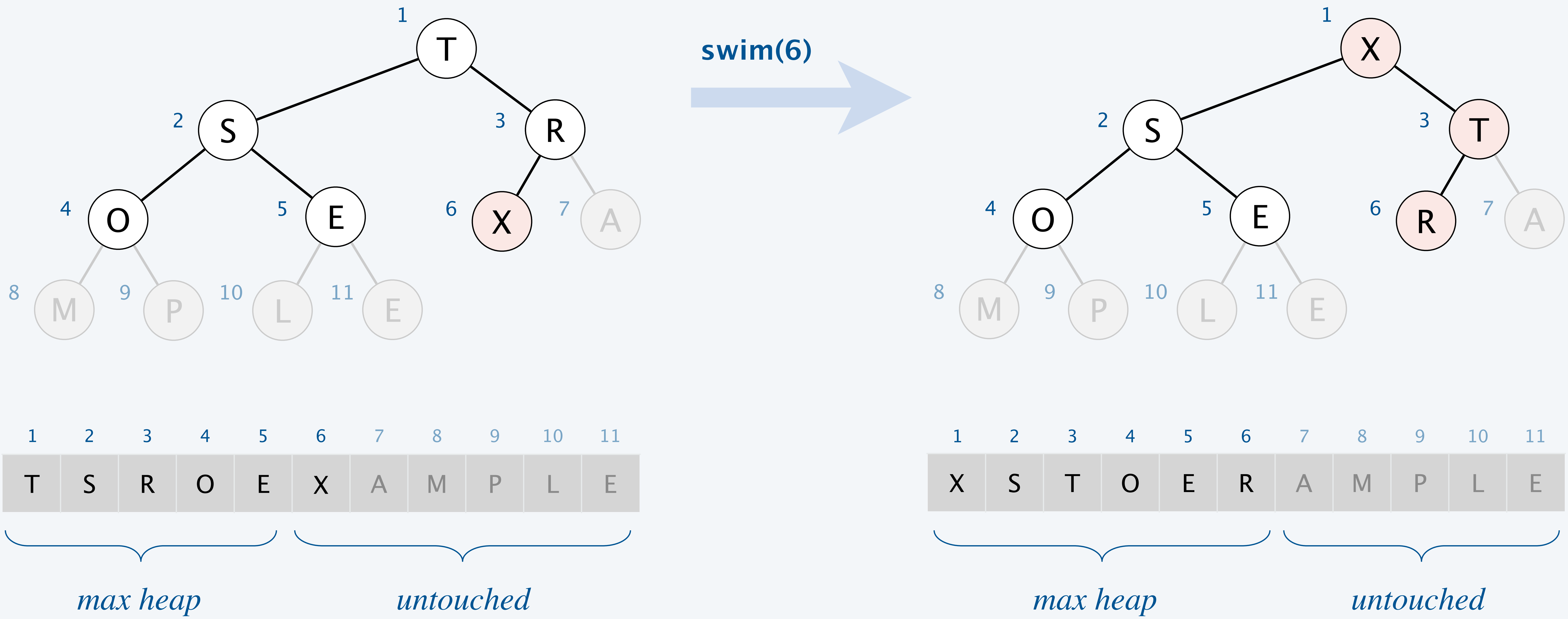


# Heapsort:  top-down heap construction

## Phase 1 (top-down heap construction).

- View input array as complete binary tree.
- Insert keys into a max heap, one at a time.

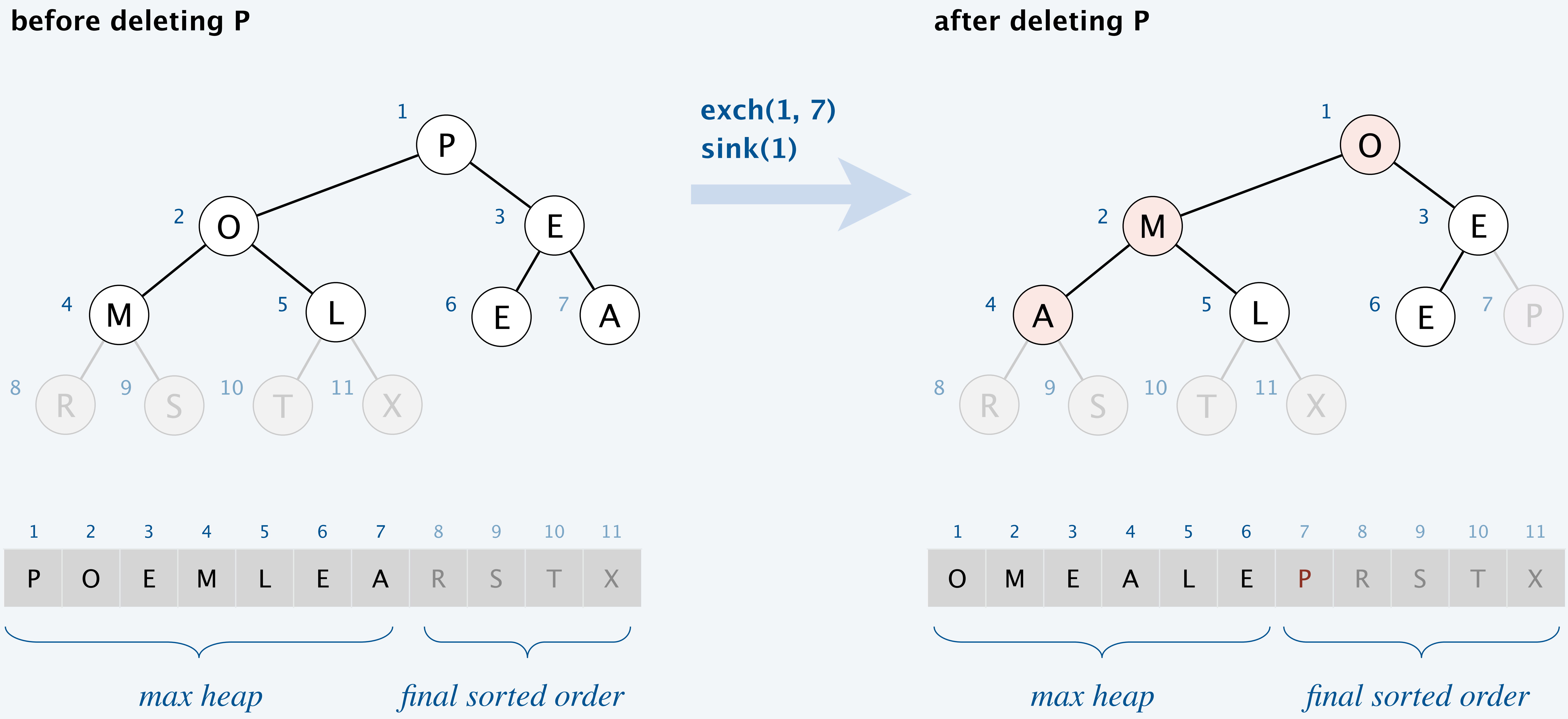
**before inserting X** | **after inserting X**



# Heapsort: sortdown

## Phase 2 (sortdown).

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



# Heapsort: Java implementation

```java
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

```java
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* a[] *and* n*)*

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

# Heapsort: mathematical analysis

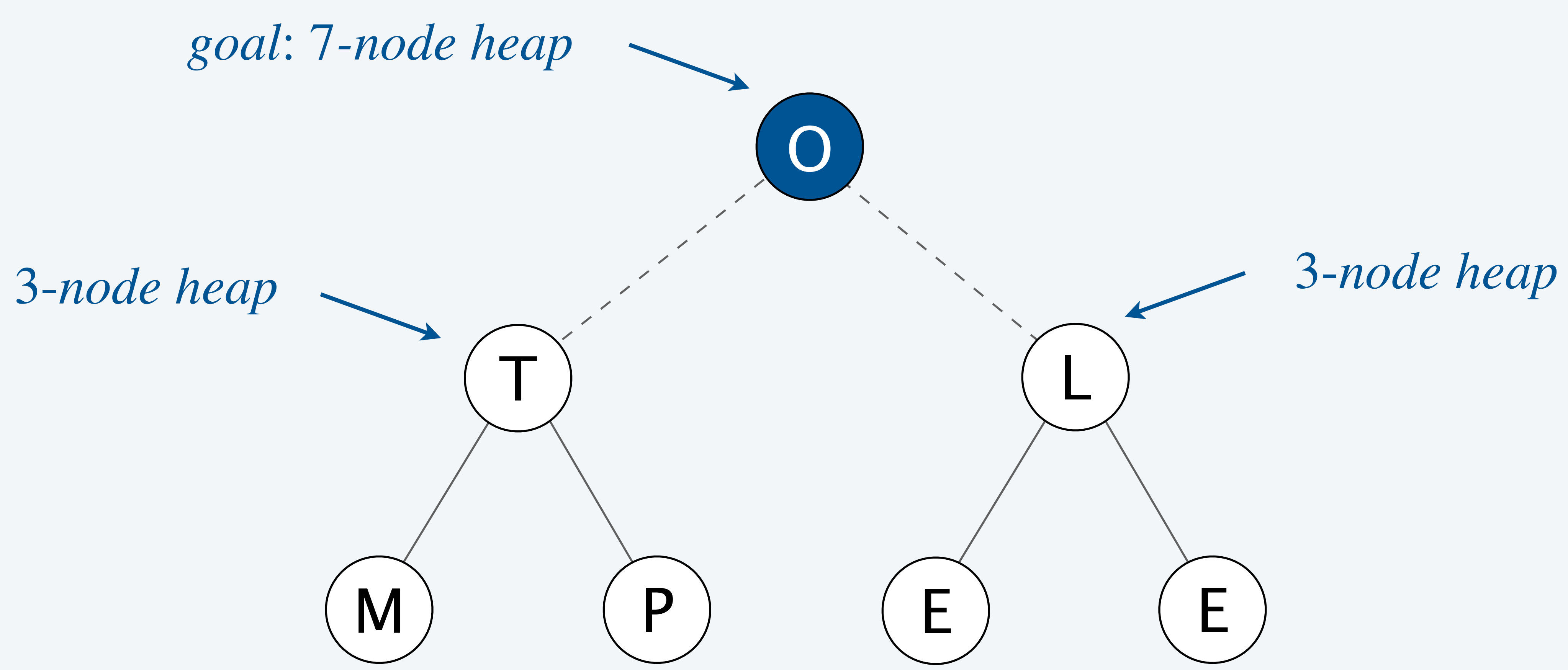
Proposition. Heapsort uses only $\Theta(1)$ extra space.

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

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

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

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



# Heapsort: context

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

- 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 possible for quicksort, but not practical*
- Heapsort: yes!

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

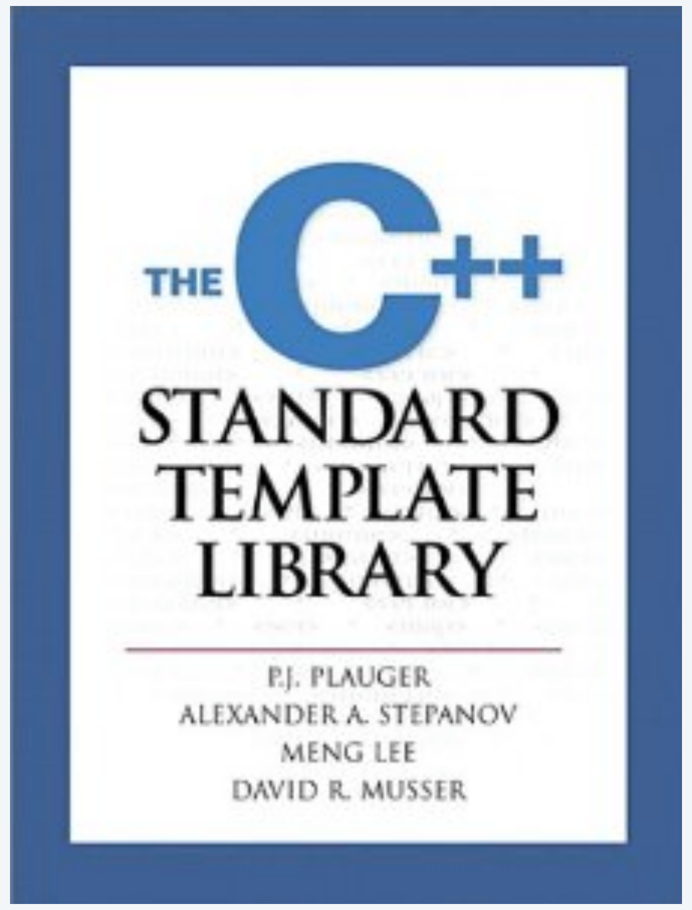
- Inner loop longer than quicksort's.
- Not stable.

# Introsort

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

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 | typical | 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 (tilde notation)**

# Credits

| image | source | license |
| --- | --- | --- |
| *Emergency Room Triage* | unknown | |
| *Car GPS* | Adobe Stock | Education License |
| *Joshua Trees* | Adobe Stock | Education License |
| *Sycamore Trees* | Alexey Sergeev | by author |
| *Weirwood Tree* | AziKun's Anime | |
| *East African Doum Palm* | Shlomit Pinter | by author |
| *The Peter Principle* | Sketchplanations | CC BY-NC 4.0 |
| *Computer and Supercomputer* | New York Times | |

# A final thought