# CS062

## DATA STRUCTURES AND ADVANCED PROGRAMMING

## 23: Priority queues


Alexandra Papoutsaki
Lectures

Mark Kampe
Labs

# Lecture 23: Priority Queues

- Priority Queue
- Binary heap
- Heapsort

Some slides adopted from Algorithms 4th Edition or COS226


## Priority Queue ADT

- Two operations:
  - Delete the maximum
  - Insert
- Applications: load balancing and interruption handling in OS, Huffman codes for compression, A* search for AI, Dijkstra's and Prim's algorithm for graph search, etc.
- How can we implement a priority queue efficiently?

## Option 1: Unordered array

- The *lazy* approach where we defer doing work (deleting the maximum) until necessary.
- Insert is $O(1)$ (will be implemented as push in stacks).
- Delete maximum is $O(n)$ (have to traverse the entire array to find the maximum element).

```java
public class UnorderedArrayMaxPQ<Key extends Comparable<Key>> {
    private Key[] pq;      // elements
    private int n;         // number of elements

    // set inititial size of heap to hold size elements
    public UnorderedArrayMaxPQ(int capacity) {
        pq = (Key[]) new Comparable[capacity];
        n = 0;
    }

    public boolean isEmpty()   { return n == 0; }
    public int size()          { return n;      }
    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];
    }
    private boolean less(int i, int j) {
        return pq[i].compareTo(pq[j]) < 0;
    }

    private void exch(int i, int j) {
        Key swap = pq[i];
        pq[i] = pq[j];
        pq[j] = swap;
    }
}
```

## Option 2: Ordered array

- The *eager* approach where we do the work (keeping the list sorted) up front to make later operations efficient.
- Insert is $O(n)$ (we have to find the index to insert and shift elements to perform insertion).
- Delete maximum is $O(1)$ (just take the last element which will the maximum).

```java
public class OrderedArrayMaxPQ<Key extends Comparable<Key>> {
    private Key[] pq;           // elements
    private int n;              // number of elements

    // set inititial size of heap to hold size elements
    public OrderedArrayMaxPQ(int capacity) {
        pq = (Key[]) (new Comparable[capacity]);
        n = 0;
    }


    public boolean isEmpty() { return n == 0;  }
    public int size()        { return n;       }
    public Key delMax()      { return pq[--n]; }

    public void insert(Key key) {
        int i = n-1;
        while (i >= 0 && less(key, pq[i])) {
            pq[i+1] = pq[i];
            i--;
        }
        pq[i+1] = key;
        n++;
    }

   private boolean less(Key v, Key w) {
        return v.compareTo(w) < 0;
    }
```

## Option 3: Binary heap

- A new data structure!
- Will allow us to both insert and delete max in $O(\log n)$ running time.
- There is no way to implement a priority queue in such a way that insert and remove max can be achieved in $O(1)$ running time.

## Lecture 23: Priority Queues

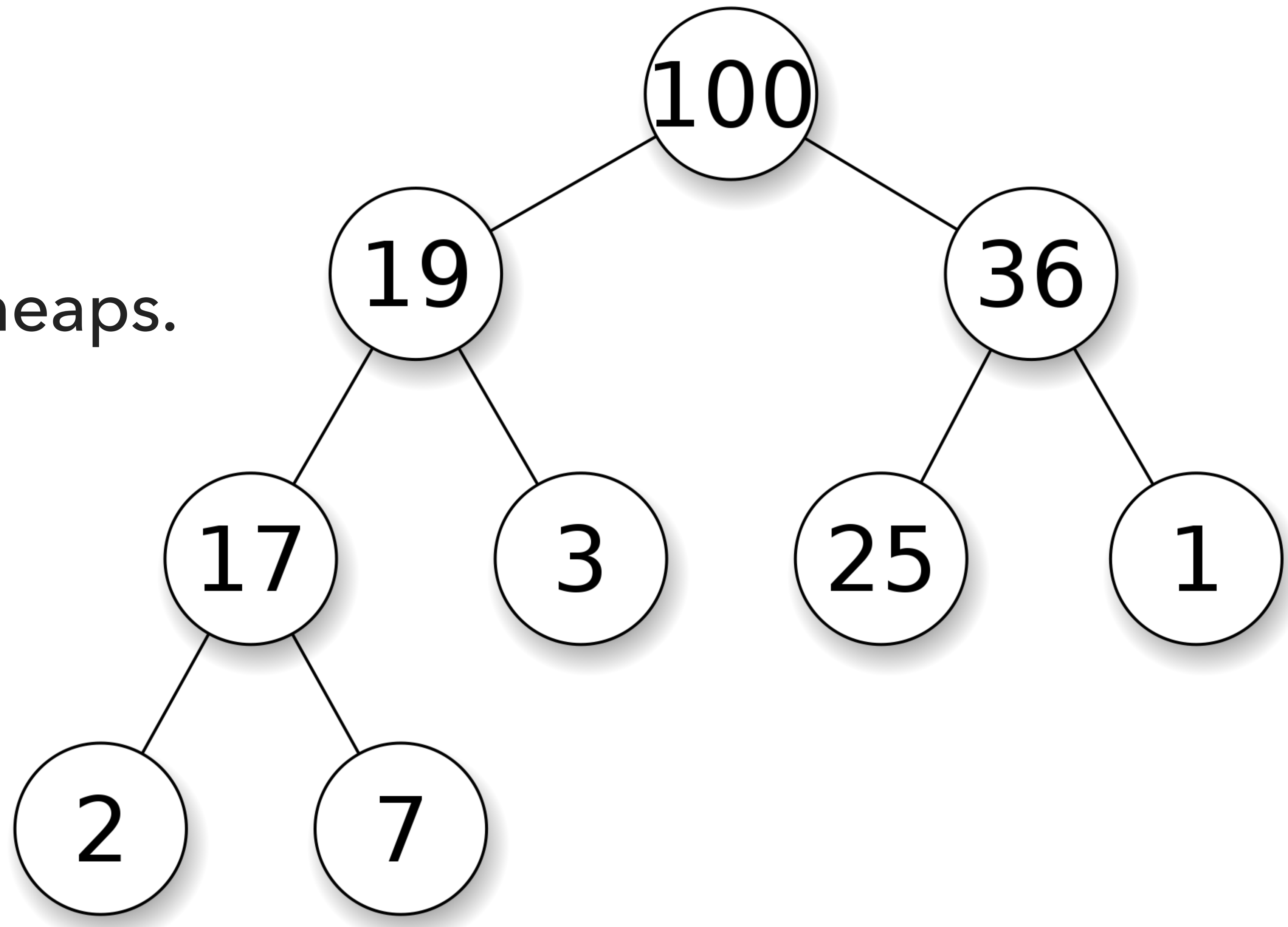
- Priority Queue
- Binary heap
- Heapsort

## Heap-ordered binary trees

- A binary tree is heap-ordered if the key in each node is larger than or equal to the keys in that node's two children (if any).
- Equivalently, the key in each node of a heap-ordered binary tree is smaller than or equal to the key in that node's parent (if any).
- Moving up from any node, we get a non-decreasing sequence of keys.
- Moving down from any node we get a non-increasing sequence of keys.

## Heap-ordered binary trees

- The largest key in a heap-ordered binary tree is found at the root!
- Max-heap.
  - There are min-heaps.



## Binary heap representation

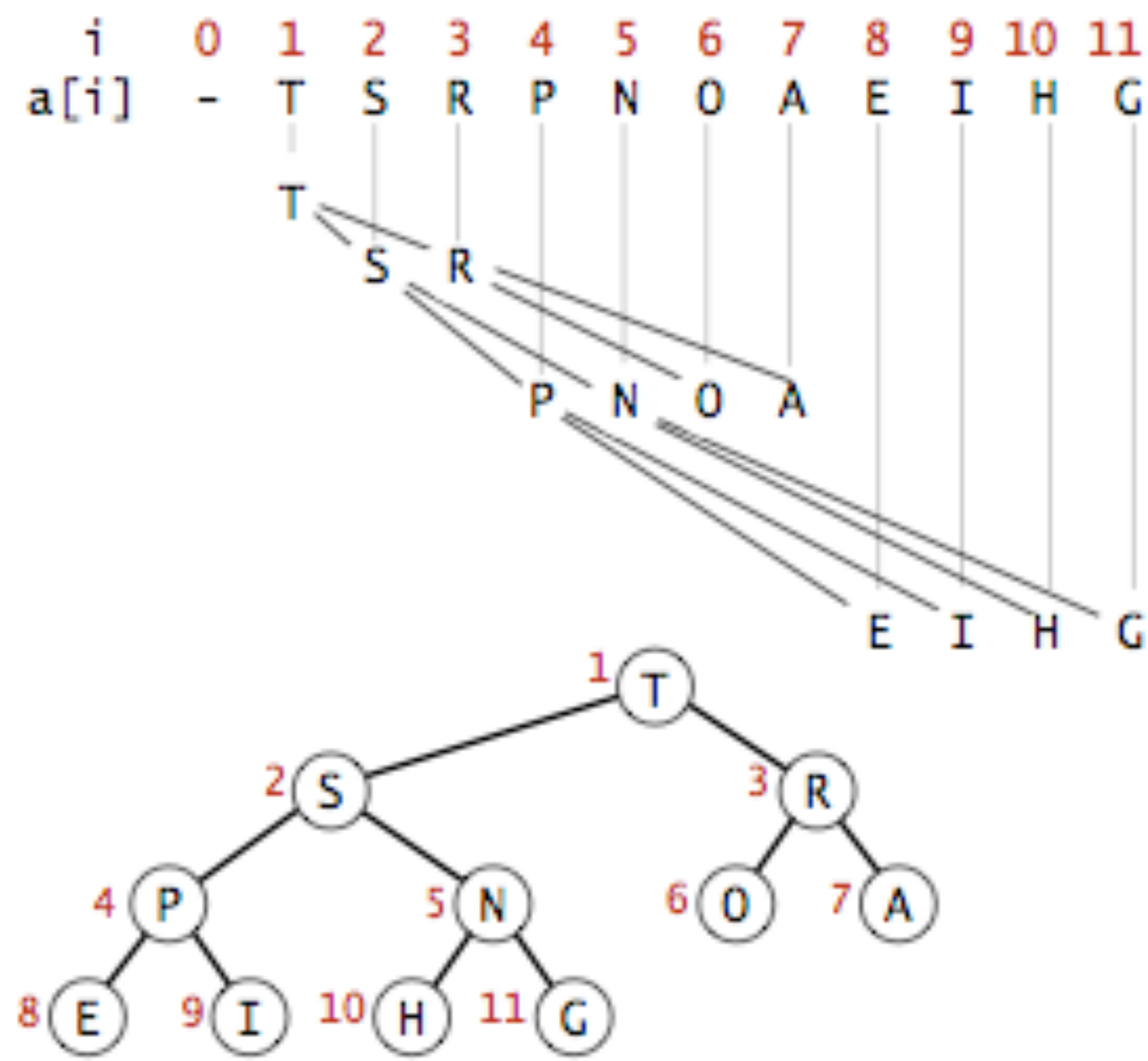
- We could use a linked representation but we would need three links for every node (one for parent, one for left subtree, one for right subtree).
- If we use complete binary trees, we can use instead an array.
  - Compact arrays vs explicit links means memory savings.

# Binary heap

- Binary heap: array representation of complete heap-ordered binary tree.
  - A data structure that can efficiently support the basic priority queue operations (insert and remove maximum).
  - Items are stored in an array such that each key is guaranteed to be larger (or equal to) than the keys at two other specific positions.

## Array representation

- Nothing is placed at index 0.
- Root is placed at index 1.
- Rest of nodes are placed in level order.
- No unnecessary indices and no wasted space because it's complete.




Heap representations

Reuniting immediate family members.

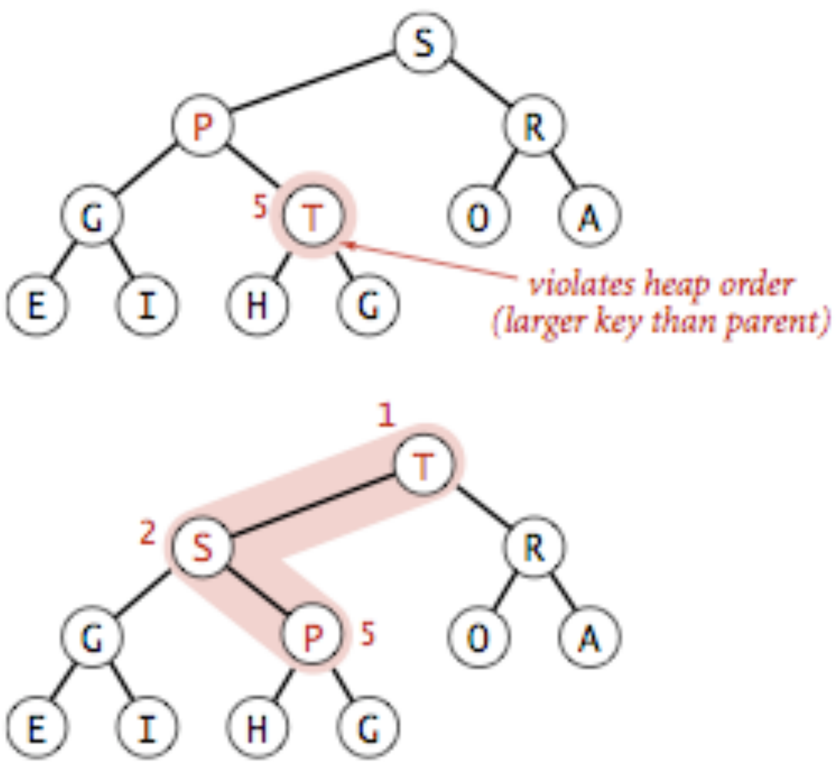
- For every node at index $k$, its parent is at index $\lfloor k/2 \rfloor$.
- Its two children are at indices $2k$ and $2k + 1$.
- We can travel up and down the tree by using this simple arithmetic on array indices.


http://algs4.cs.princeton.edu

# 2.4 BINARY HEAP DEMO

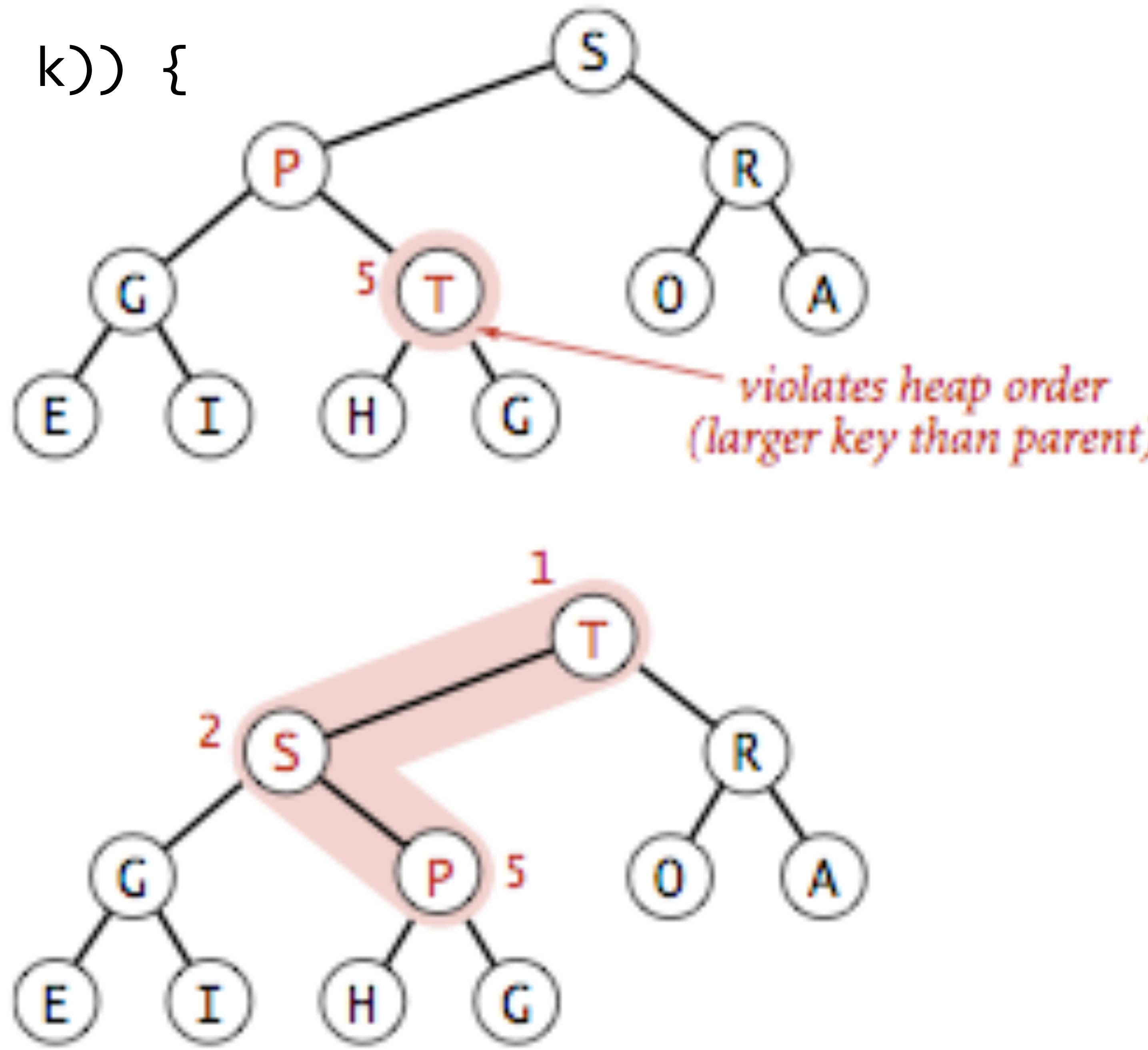
# Swim/promote/percolate up/bottom up reheapify

- Scenario: a key becomes larger than its parent therefore it violates the heap-ordered property.
- To eliminate the violation:
  - Exchange key in child with key in parent.
  - Repeat until heap order restored.



## Swim/promote/percolate up

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

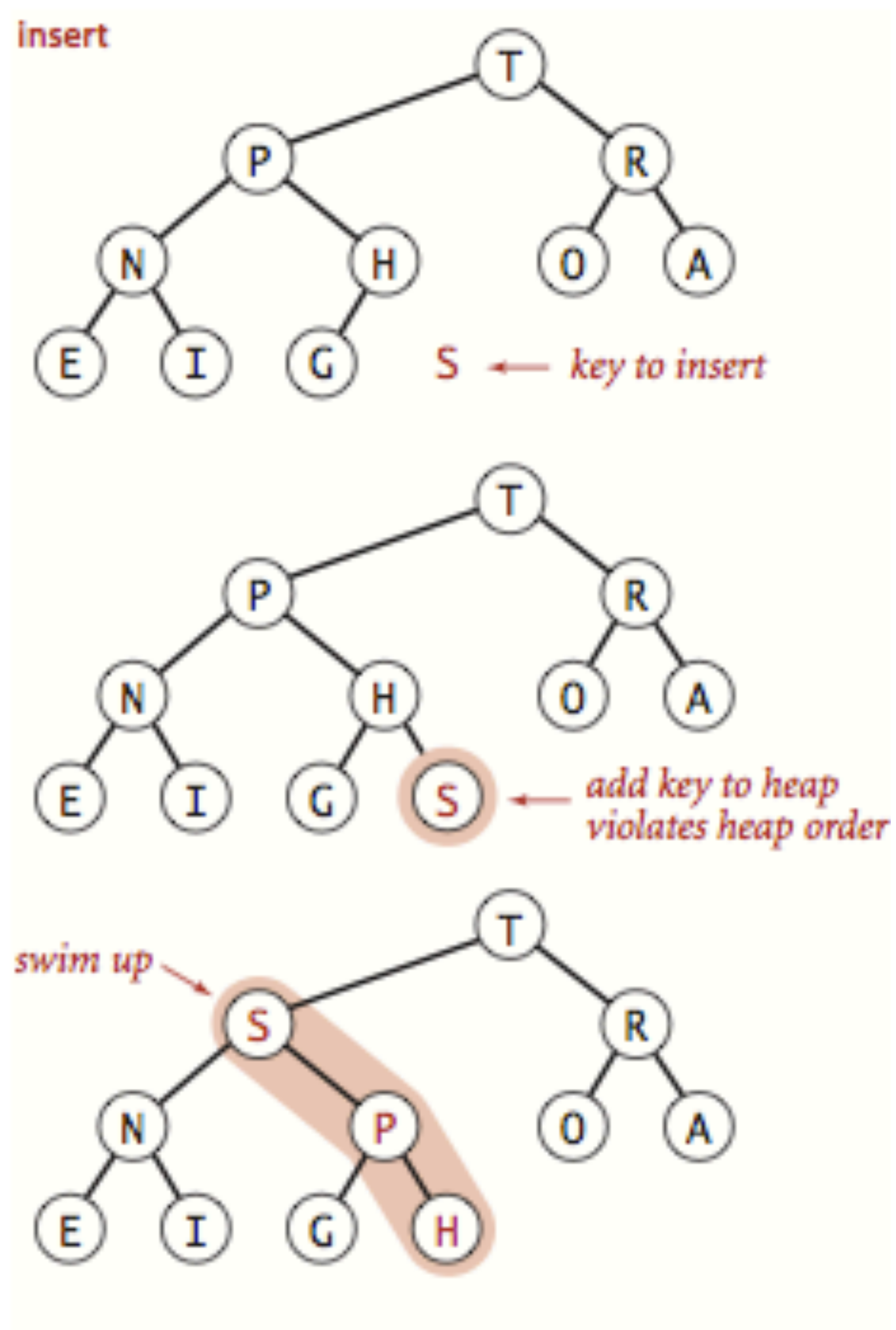


## Binary heap: insertion

- Insert: Add node at end in bottom level, then swim it up.
- Cost: At most $\log n + 1$ compares.

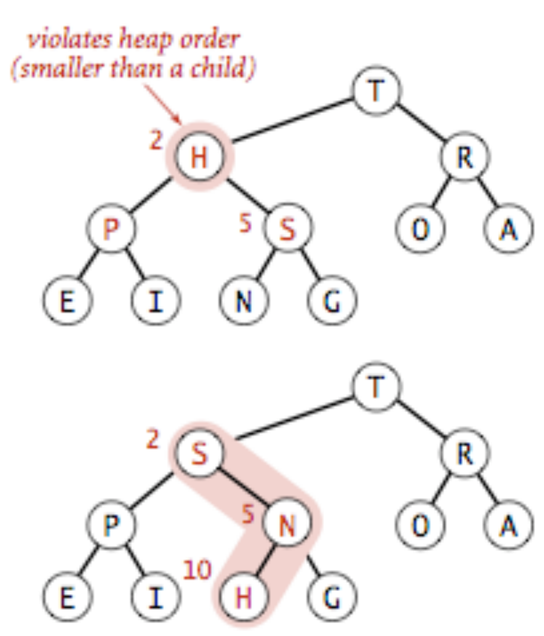
```
public void insert(Key x) {
    pq[++n] = x;
    swim(n);
}
```

# Binary heap: insertion



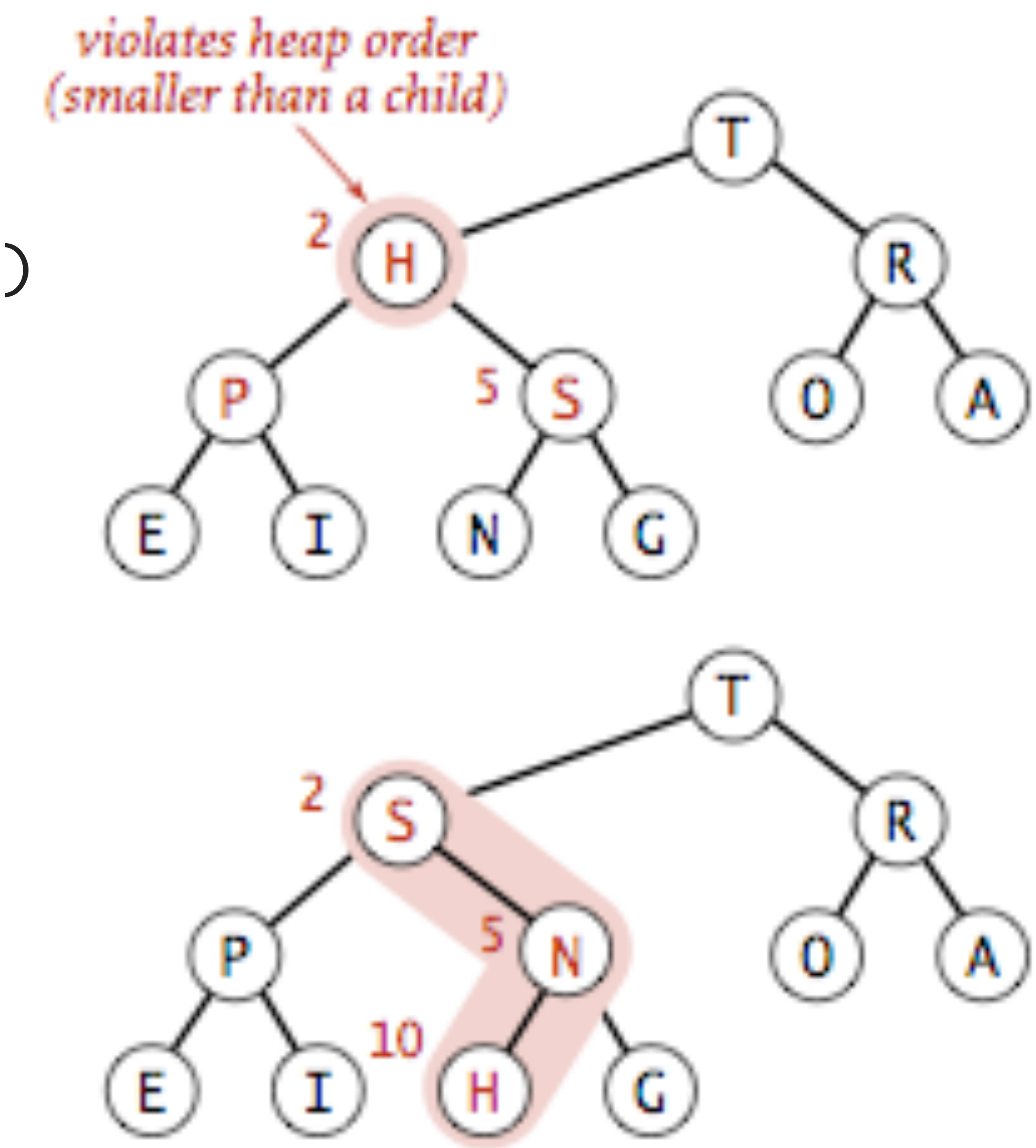
# Sink/demote/top down heapify

- Scenario: a key becomes smaller than one (or both) of its children's keys.
- To eliminate the violation:
  - Exchange key in parent with key in **larger** child.
  - Repeat until heap order restored.



# Sink/demote/top down heapify

```
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;
    }
}
```

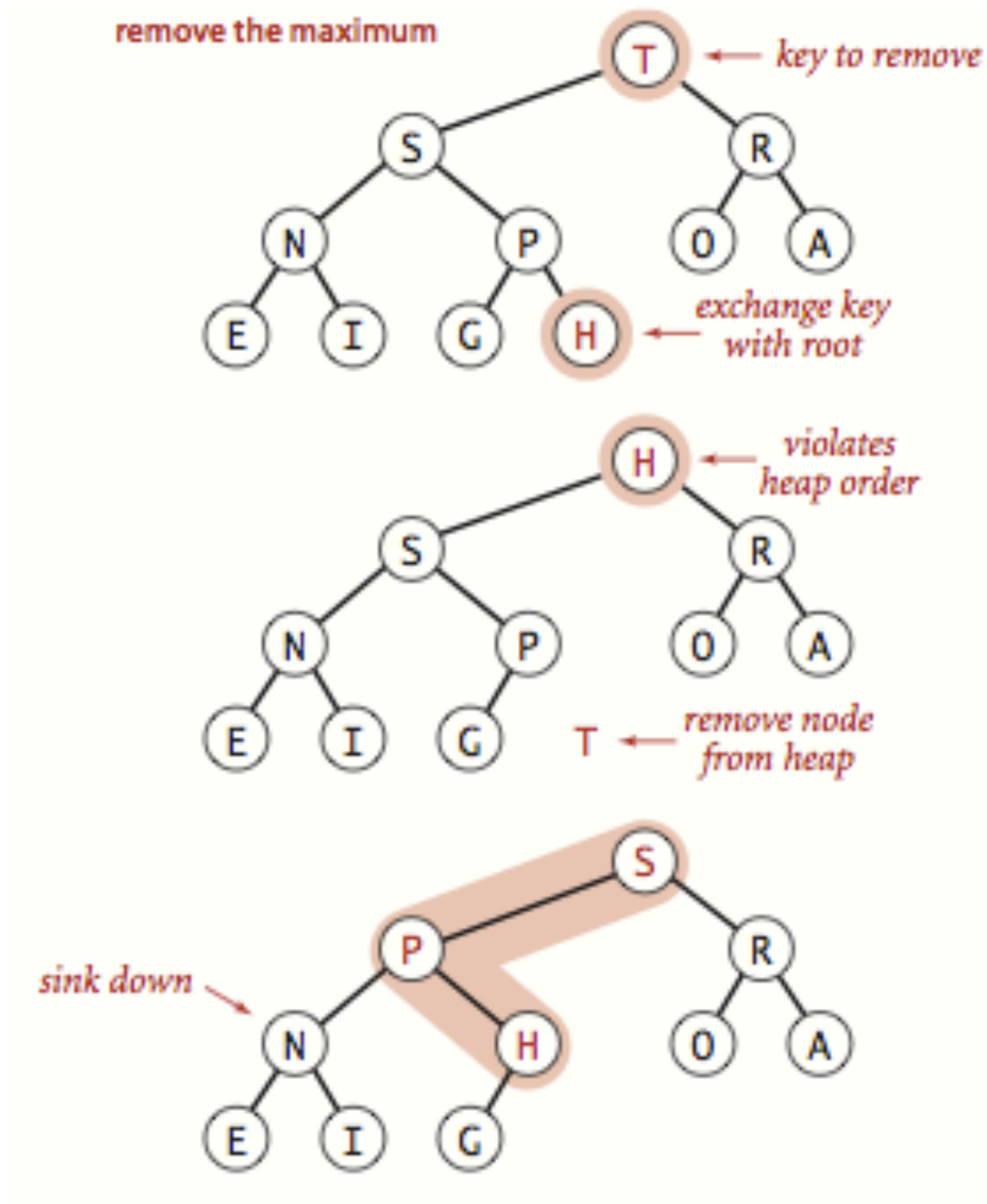


# Binary heap: return (and delete) the maximum

- Delete max: Exchange root with node at end. Return it and delete it. Sink the new root down.
- Cost: At most $2 \log n$ compares.

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

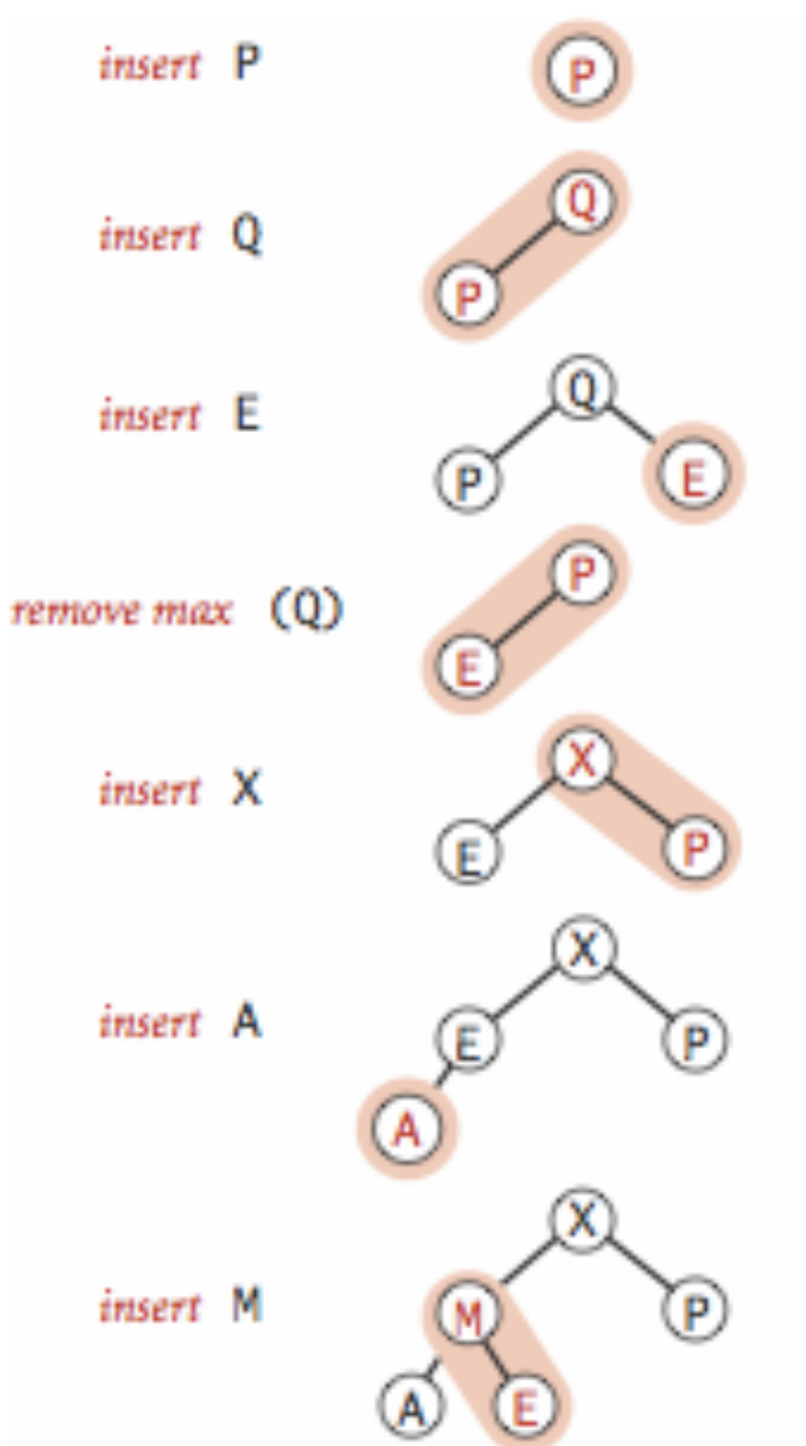
# Binary heap: delete and return maximum

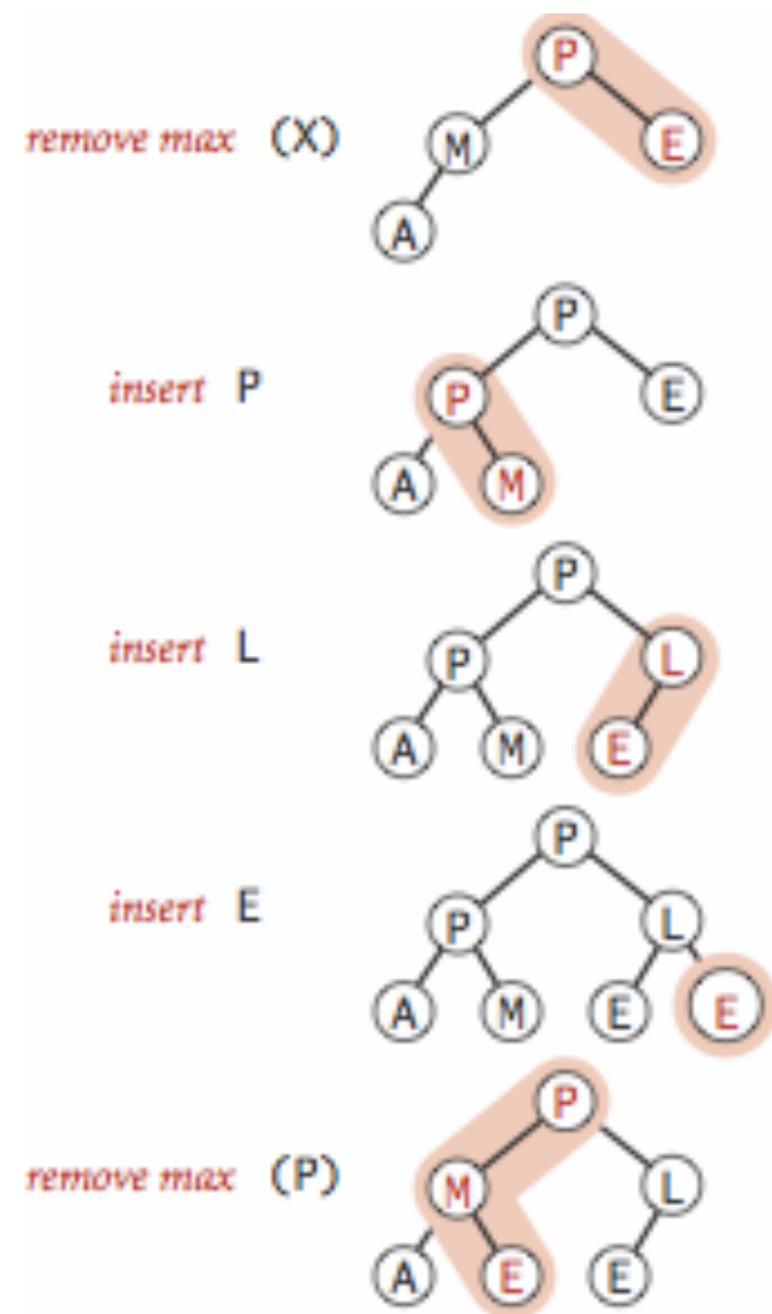


## Putting everything together

- Insert is $O(\log n)$.
- Delete max is $O(\log n)$.
- Look into MaxPQ class https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/MaxPQ.java.html

# Putting everything together



# Lecture 23: Priority Queues

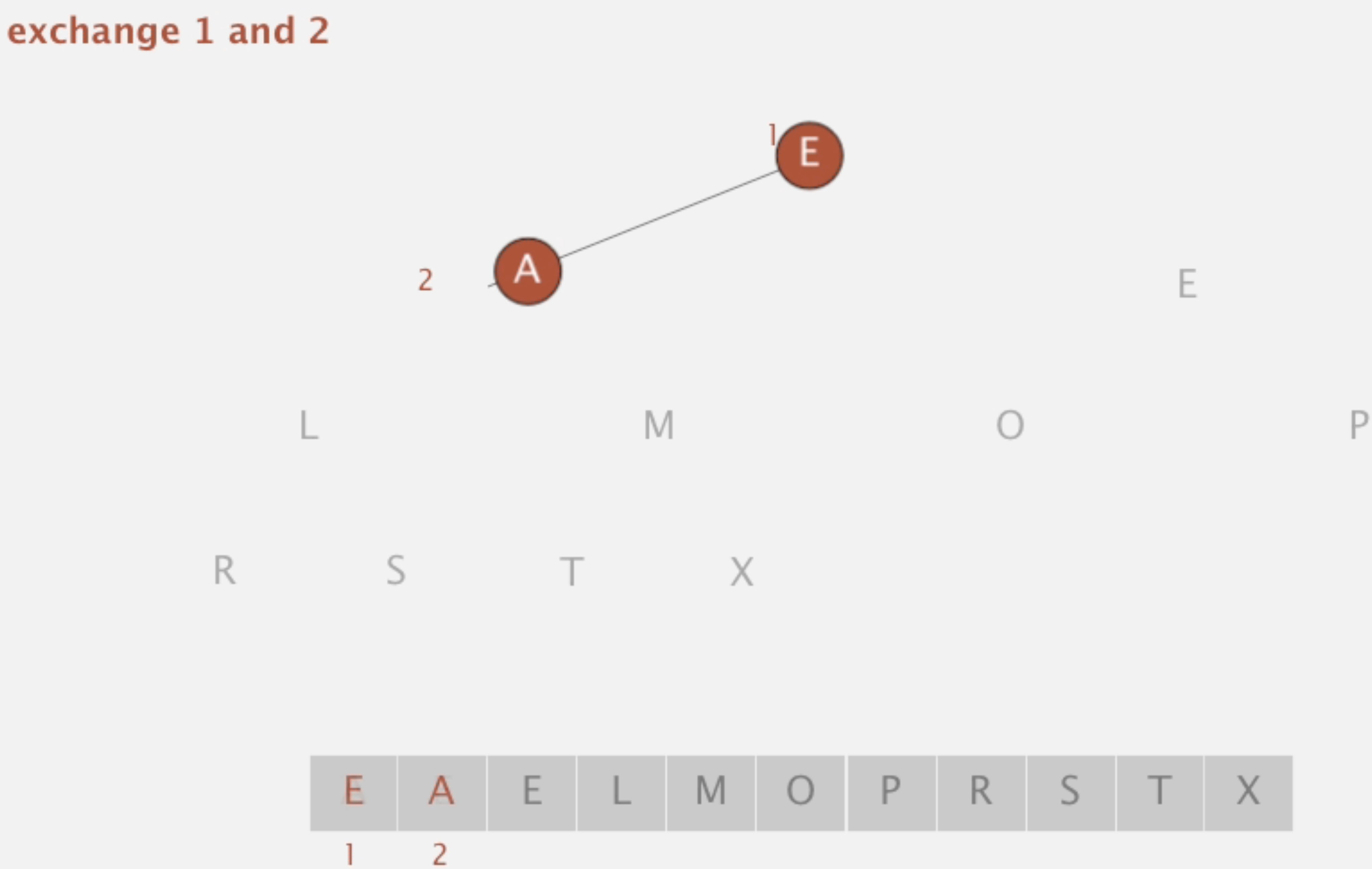
- Priority Queue
- Binary heap
- Heapsort

## 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.

# Heapsort demo

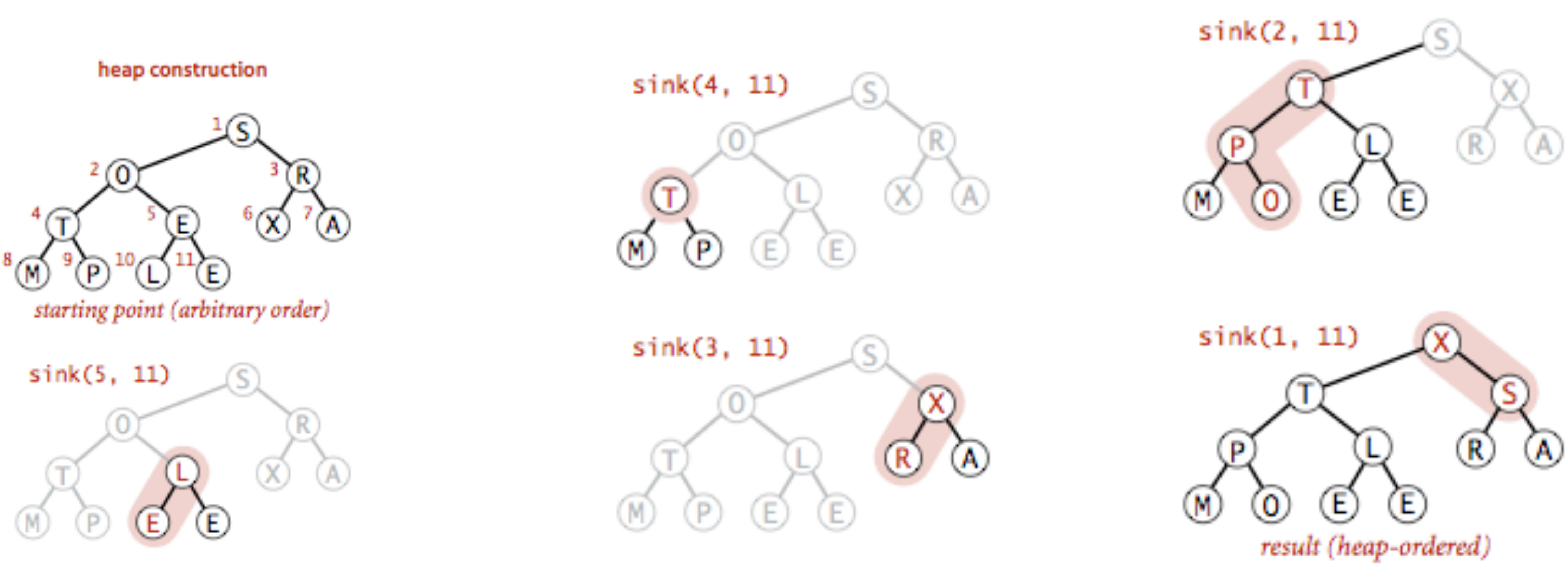
Sortdown. Repeatedly delete the largest remaining item.



# Heap construction

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

- Key insight: After `sink(a,k,n)` completes, the subtree rooted at k is a heap.

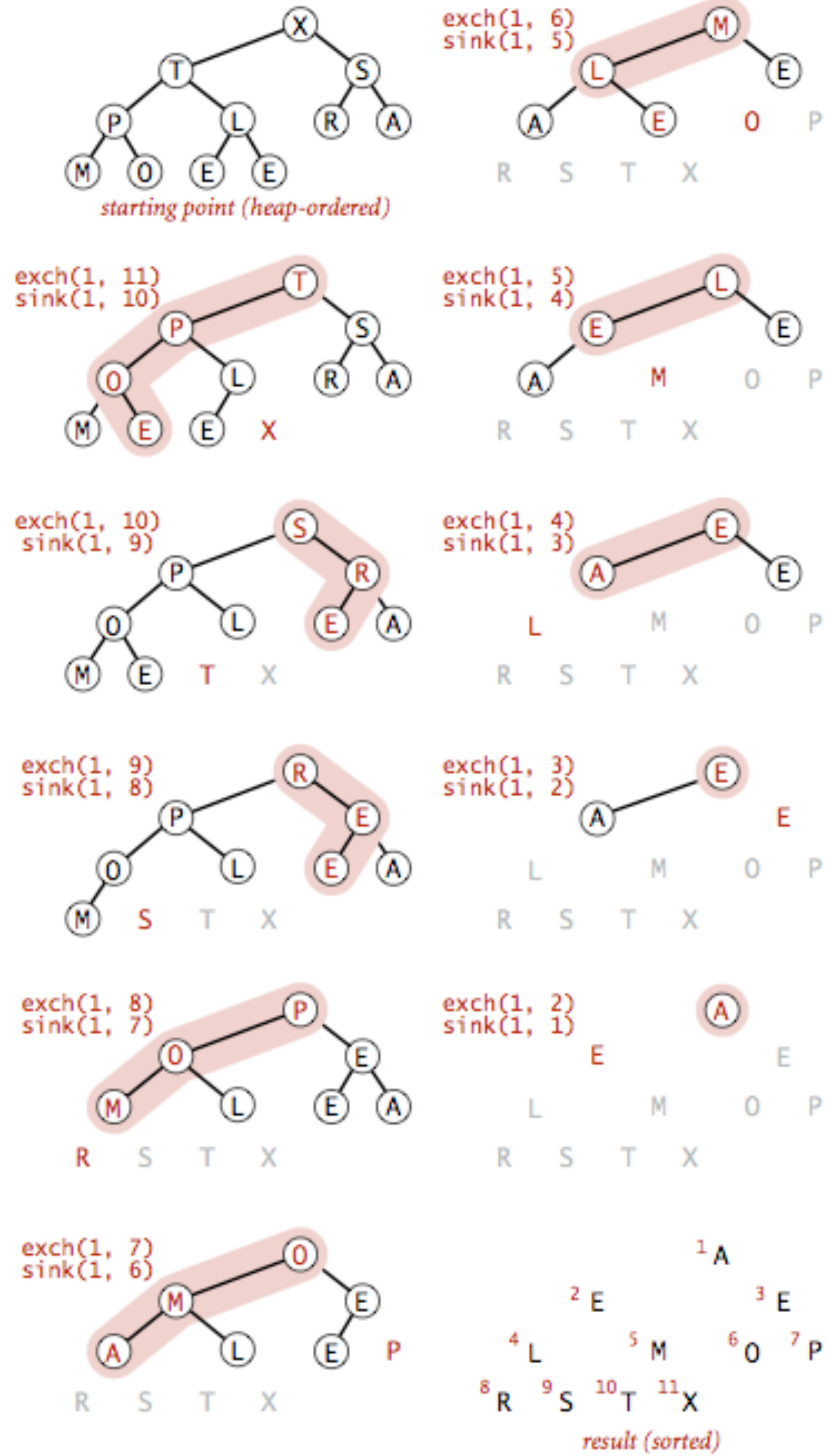

  ```

## Sortdown

- Remove the maximum, one at a time, but leave in array instead of nulling out.
- 
```
while(n>1){
   exch(a, 1, n--);
   sink(a, 1, n);
}
```
- Key insight: After each iteration the array consists of a heap-ordered subarray followed by a sub-array in final order.

Sortdown

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


  ```

# Heapsort analysis

- Heap construction makes $O(n)$ exchanges and $O(n)$ compares.
- Heapsort uses $O(n \log n)$ exchanges and compares.
- In-place sorting algorithm with $O(n \log n)$ worst-case!
- Remember:
  - mergesort: not in place, requires linear extra space.
  - quicksort: quadratic time in worst case.
- Heapsort is optimal both for time and space, but:
  - Inner loop longer than quick sort.
  - Poor use of cache.
  - Not stable.

# What you need to remember about sorting

| | In place | Stable | Best | Average | Worst | Remarks |
|---|---|---|---|---|---|---|
| Selection | X | | $1/2n^2$ | $1/2n^2$ | $1/2n^2$ | $n$ exchanges |
| Insertion | X | X | $n$ | $1/4n^2$ | $1/2n^2$ | Use for small arrays or partially ordered |
| Merge | | X | $1/2n \log n$ | $n \log n$ | $n \log n$ | Guaranteed performance; stable |
| Quick | X | | $n \log n$ | $2n \ln n$ | $1/2n^2$ | $n \log n$ probabilistic guarantee; fastest in practice |
| Heap | X | | $n \log n$ | $2n \log n$ | $2n \log n$ | $n \log n$ guarantee; in place |

## Lecture 23: Priority Queues

- Priority Queue
- Binary heap
- Heapsort

## Readings:

- Textbook:
  - Chapter 2.4 (Pages 308-327), 2.5 (336-344)
- Website:
  - Priority Queues: https://algs4.cs.princeton.edu/24pq/

## Practice Problems:

- 2.4.1-2.4.11. Also try some creative problems.