CS062

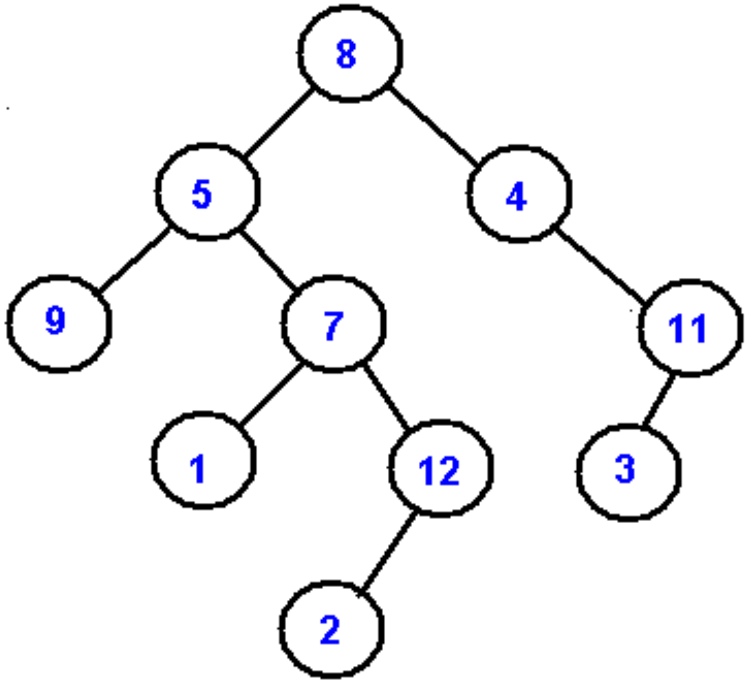
DATA STRUCTURES AND ADVANCED PROGRAMMING

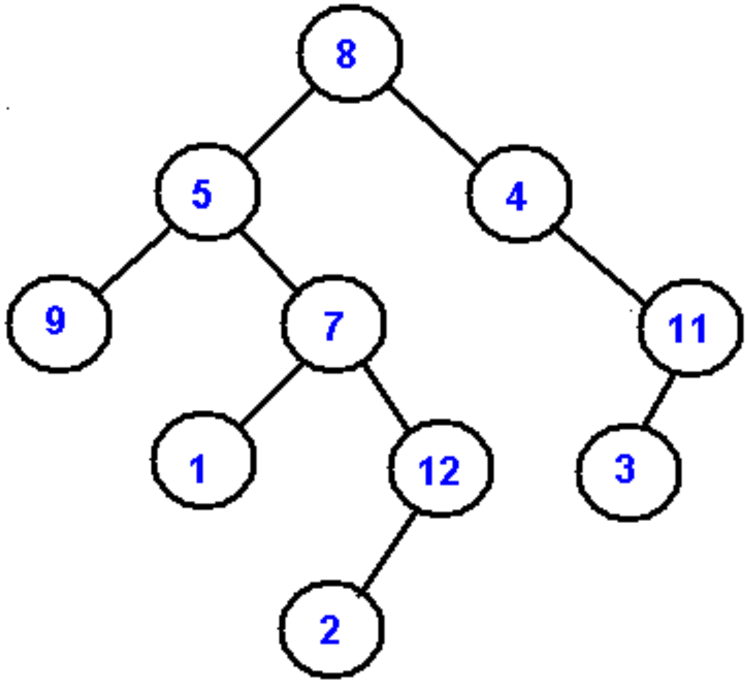
17: Heaps, Priority Queue, Heap Sort

Tom Yeh
he/him/his

Recap

- Binary Tree
- Tree Traversal: pre-order, in-order, post-order, and level order:





- Pre-order: 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
- In-order: 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
- Post-order: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
- Level-order: 8, 5, 4, 9, 7, 11, 1, 12, 3, 2

Lecture 17: Heaps, Priority Queues and Heapsort

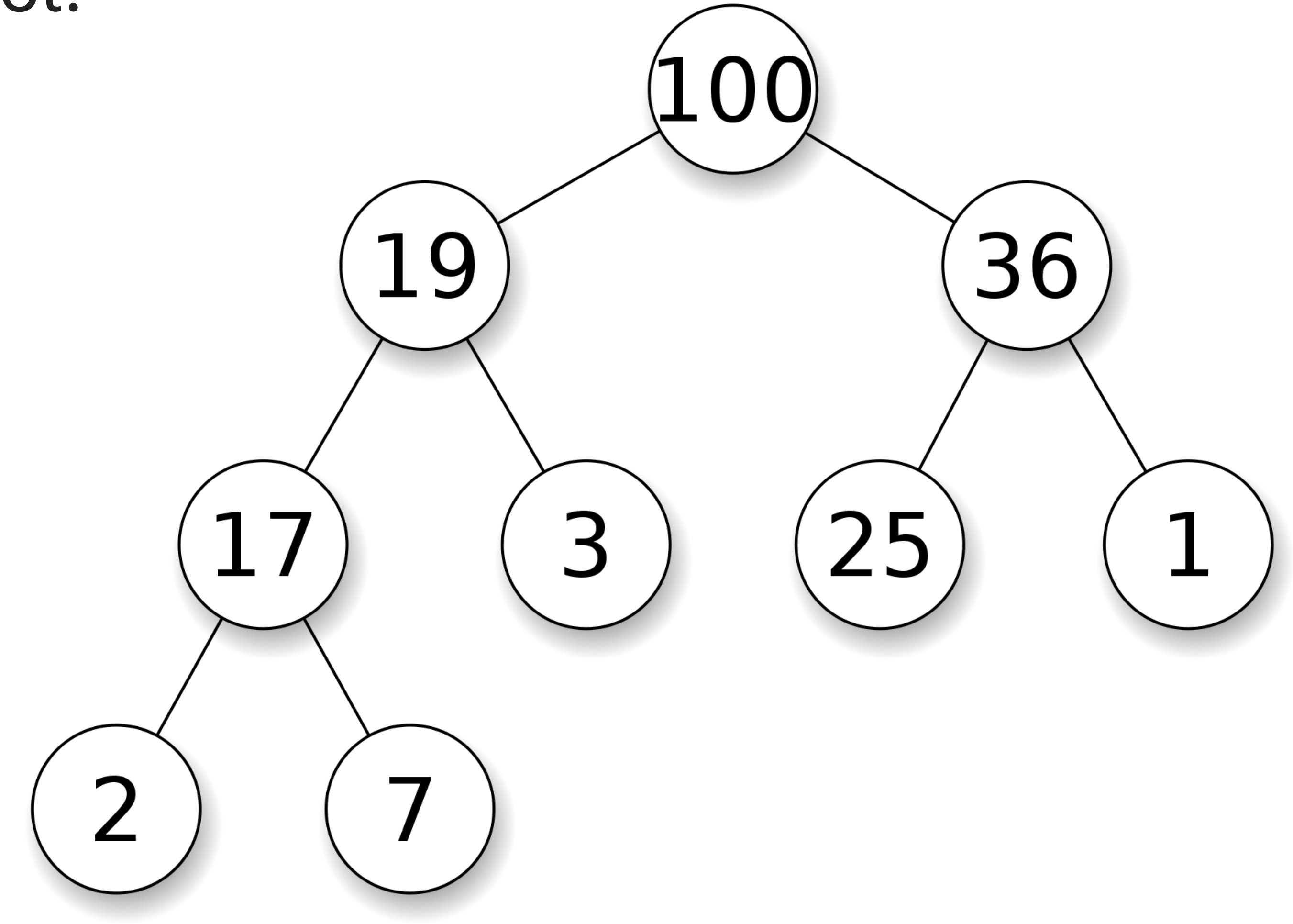
- Binary Heaps
- Priority Queue
- Heapsort

Heap-ordered binary trees

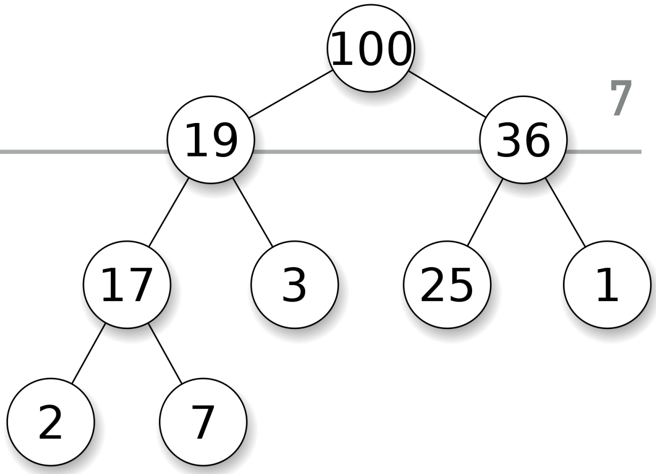
- A binary tree is heap-ordered if the key in each node is larger than or equal to the keys in that node's two children (if any).
- Equivalently, the key in each node of a heap-ordered binary tree is smaller than or equal to the key in that node's parent (if any).
- No assumption of which child is smaller.
- 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!



Binary heap representation



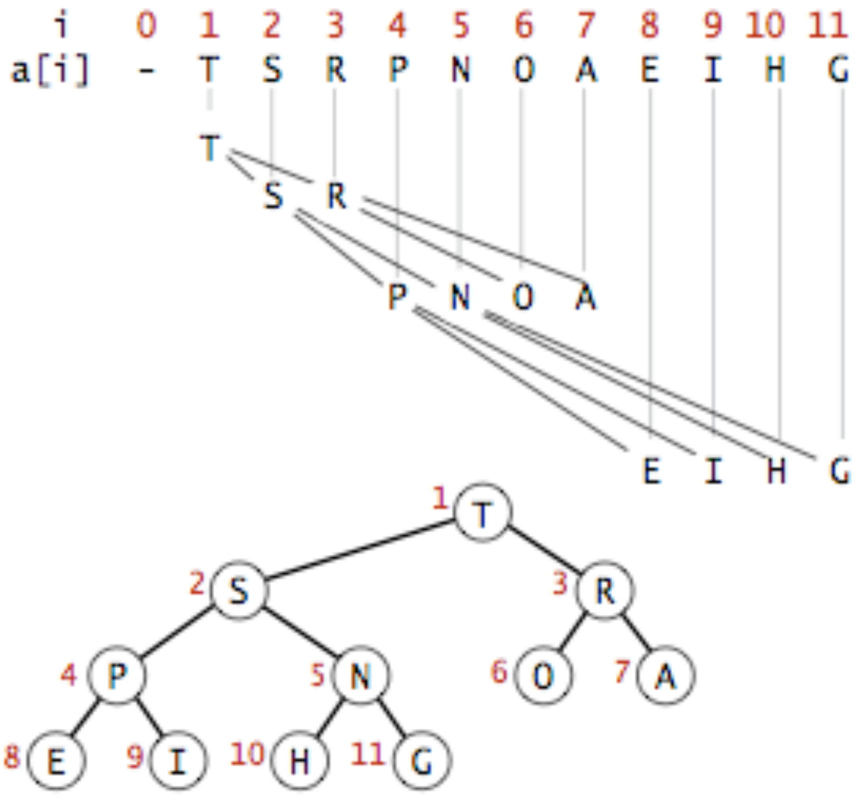
- We could use a linked representation but we would need three links for every node (one for parent, one for left subtree, one for right subtree).
- If we use **complete binary trees**, we can use an array instead.
  - Compact arrays vs explicit links means memory savings and faster execution!
    - Array access is much faster than chasing down pointers

Binary heaps

- Binary heap: the array representation of a **complete heap-ordered binary tree.**
    - Parent’s key is not smaller than children’s keys.
    - Children’s keys are not bigger than parent’s key.
- Max-heap but there are min-heaps, too.

Array representation of heaps

- Nothing is placed at index 0.
- Root is placed at index 1.
    - Easy indexing between parent/child
- Rest of nodes are placed in level order.
- No unnecessary indices and no wasted space because it’s complete.
- What's the relationship between node index and 2 children?



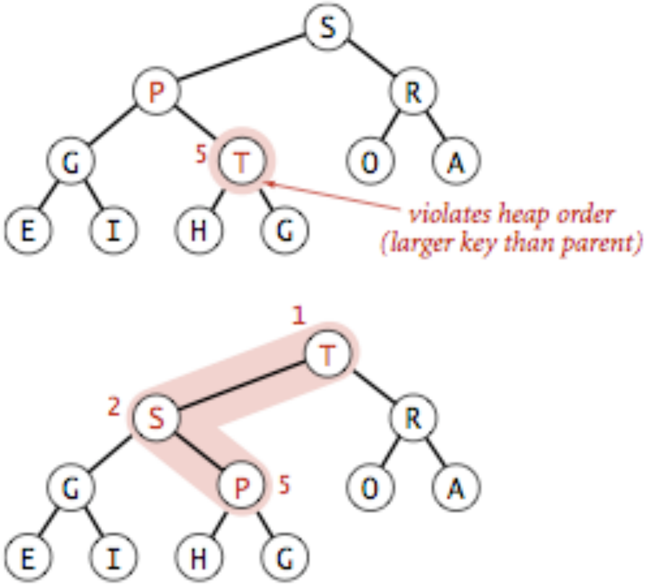
Heap representations

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 heap by using this simple arithmetic on array indices.
- Accesses using indices are much faster than using pointers/references

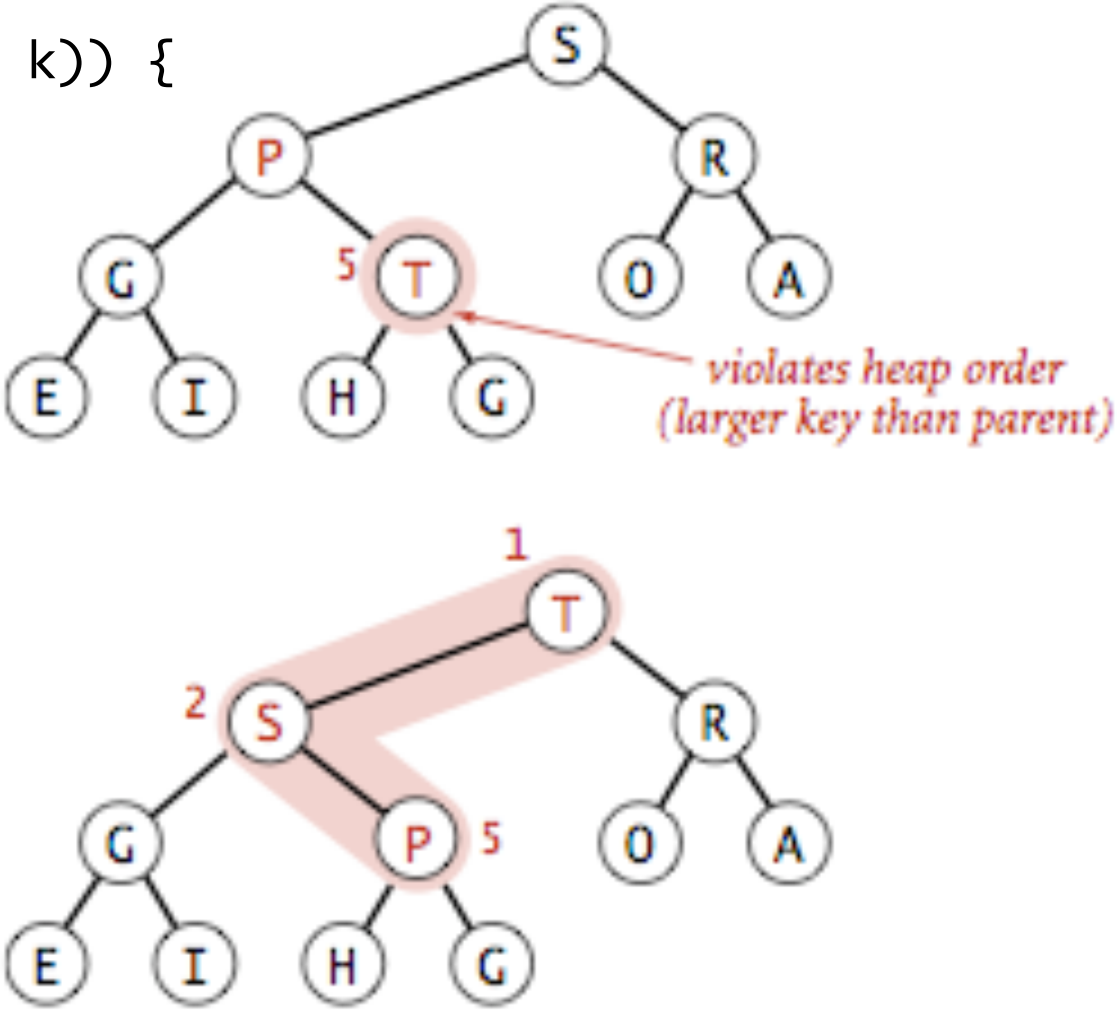
Swim/promote/percolate up/bottom up reheapify

- 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;
   }
}
```

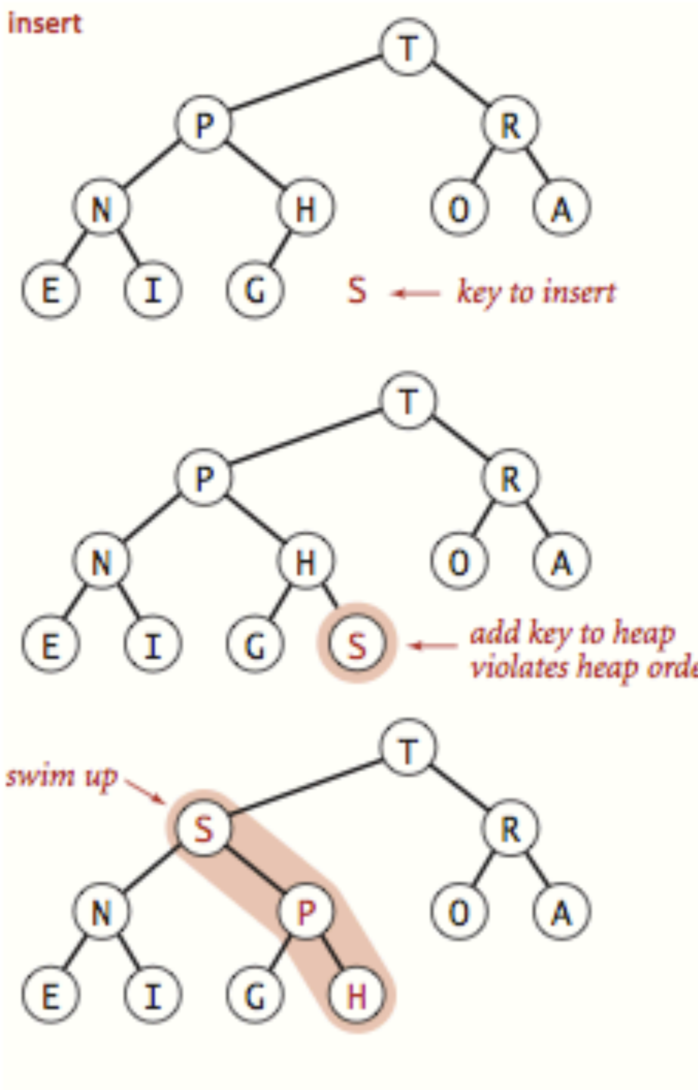


violates heap order
(larger key than parent)

Binary heap: insertion

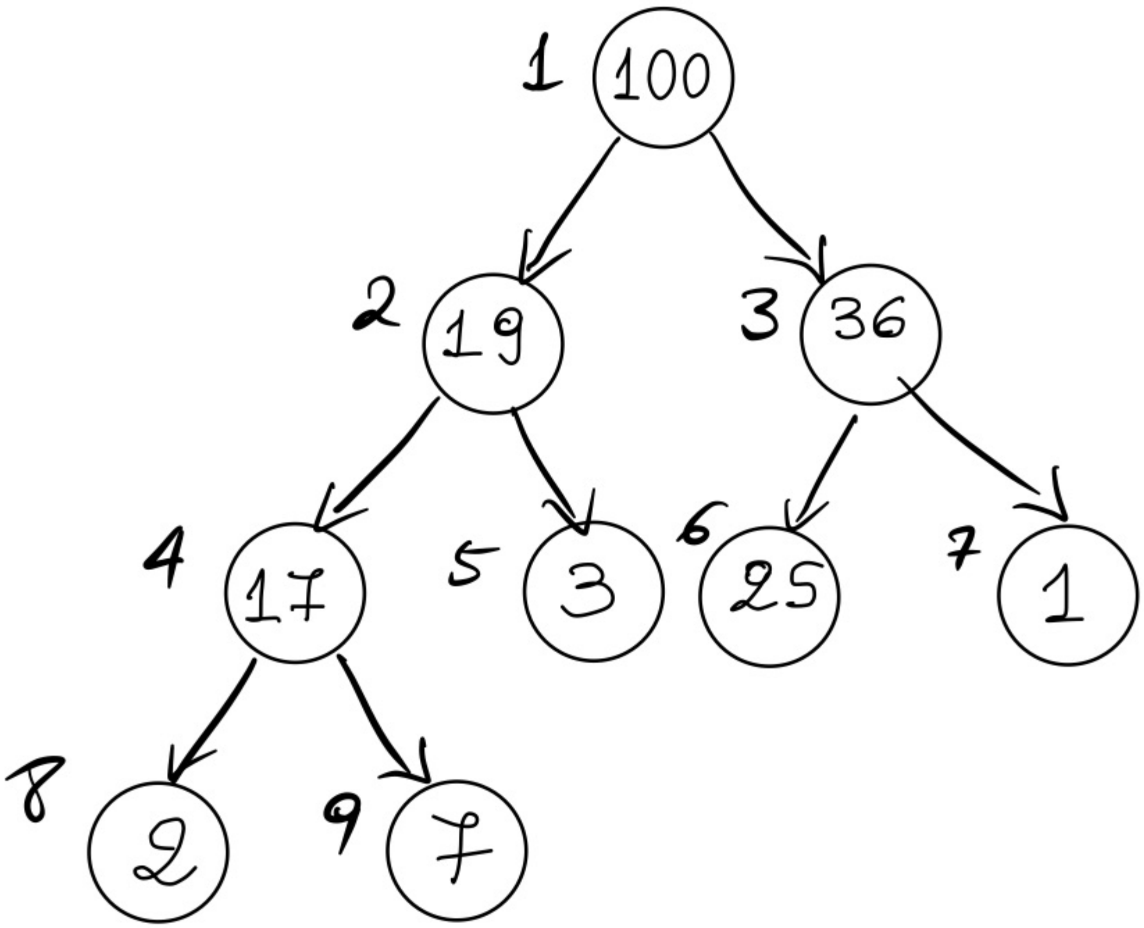
- Insert: Add node **at end in bottom level**, then **swim it up**.
- Cost: At most $\log n + 1$ compares.

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

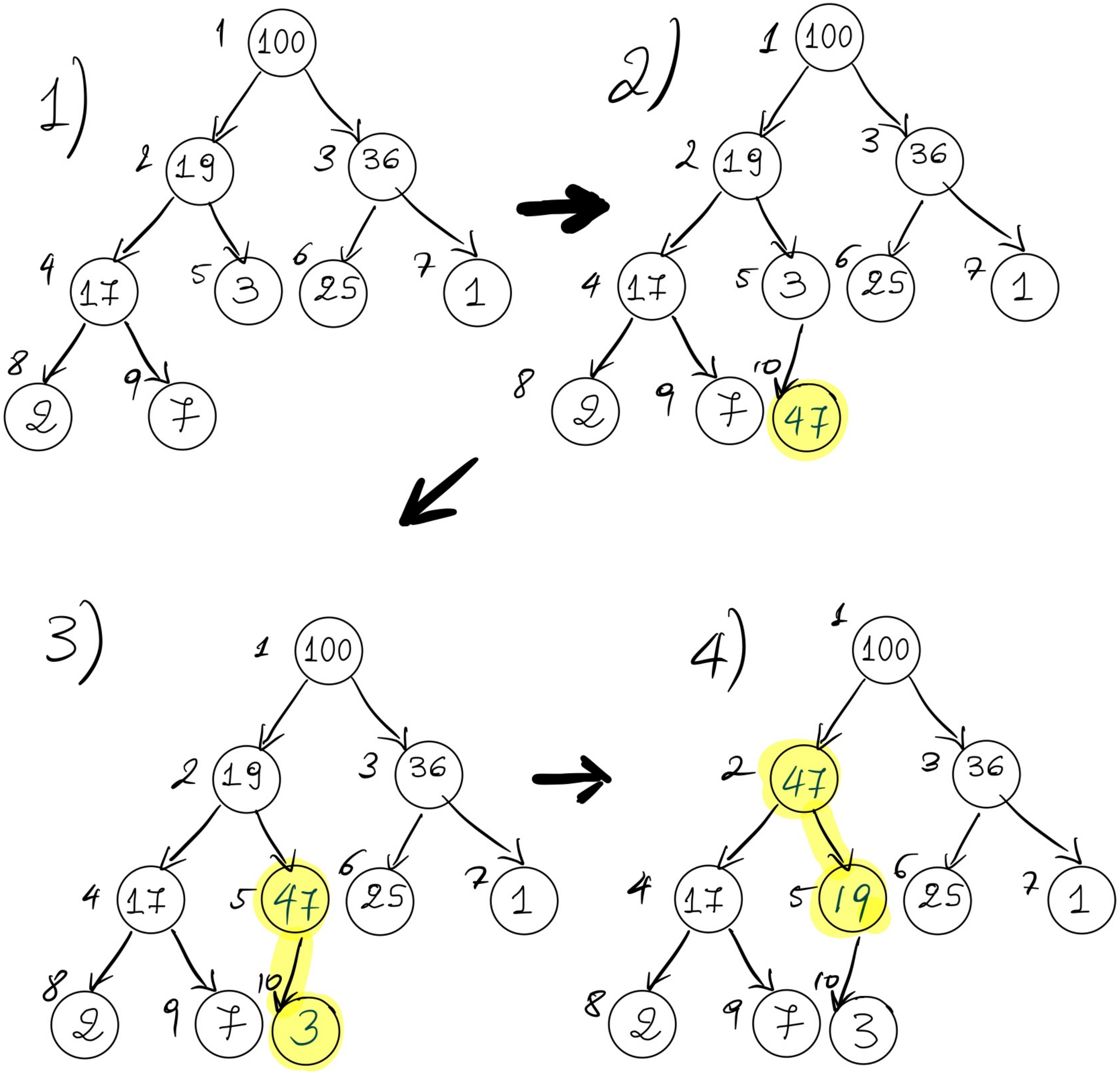


Practice Time

- Insert 47 in this binary heap.

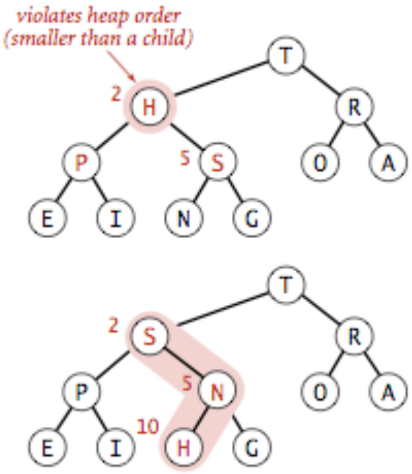


Answer


1)
1 100
2 19
3 36
4 17
5 3
6 25
7 1
8 2
9 7
2)
1 100
2 19
3 36
4 17
5 3
6 25
7 1
8 2
9 7
10 47
3)
1 100
2 19
3 36
4 17
5 47
6 25
7 1
8 2
9 7
10 3
4)
1 100
2 47
3 36
4 17
5 19
6 25
7 1
8 2
9 7
10 3

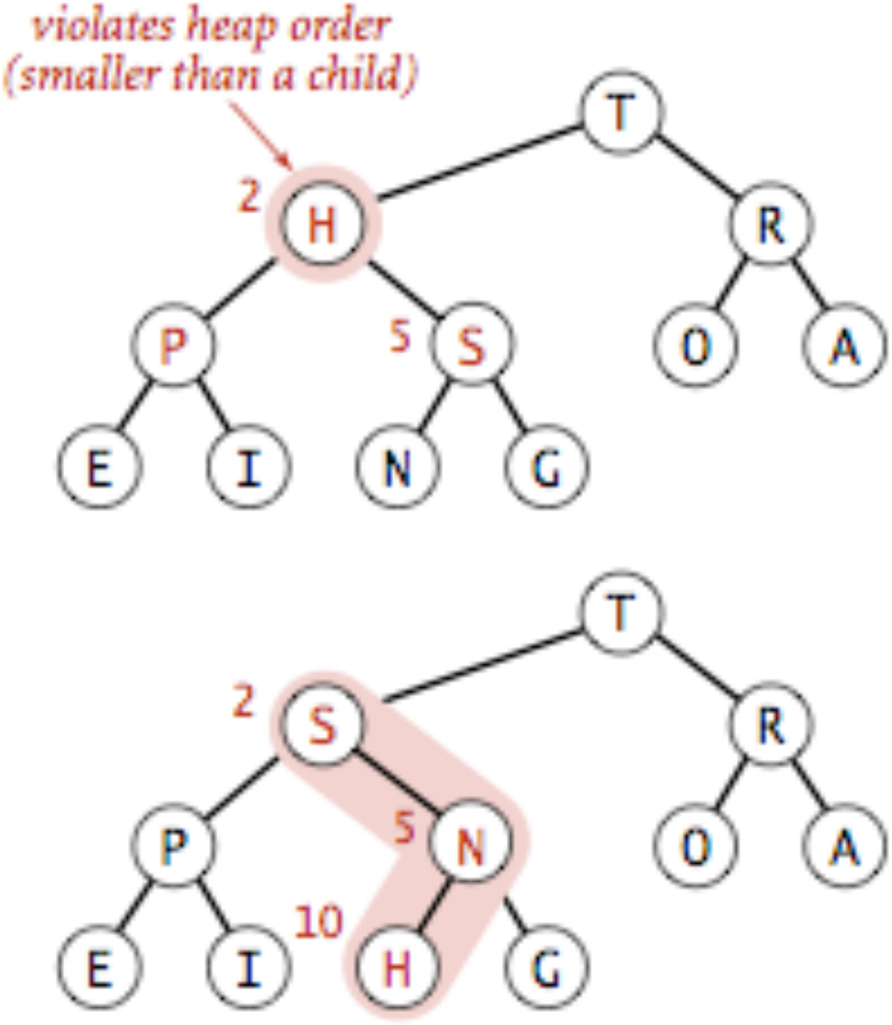
Sink/demote/top down heapify

- 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 is restored.



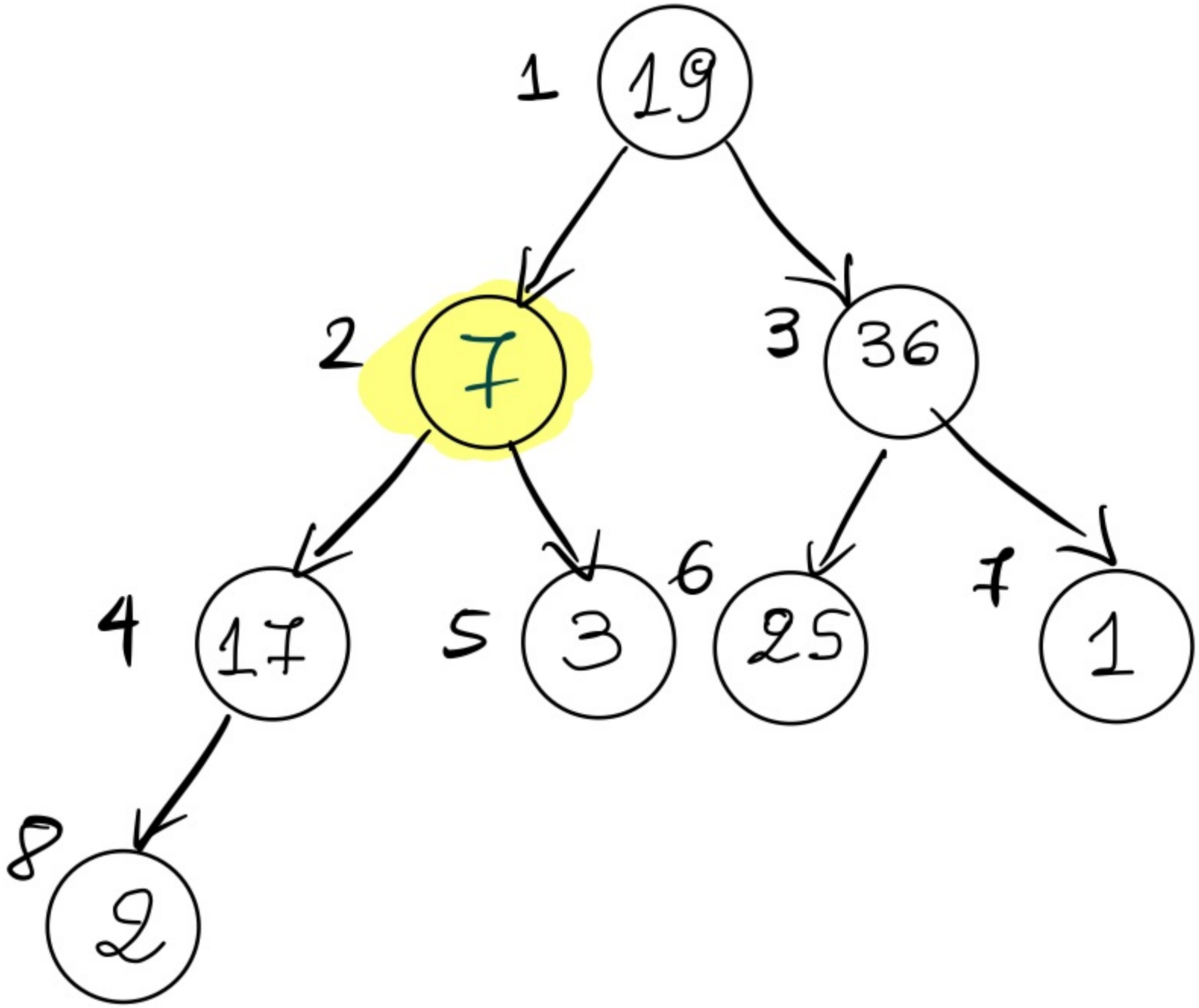
Sink/demote/top down heapify

```
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;
    }
}
```

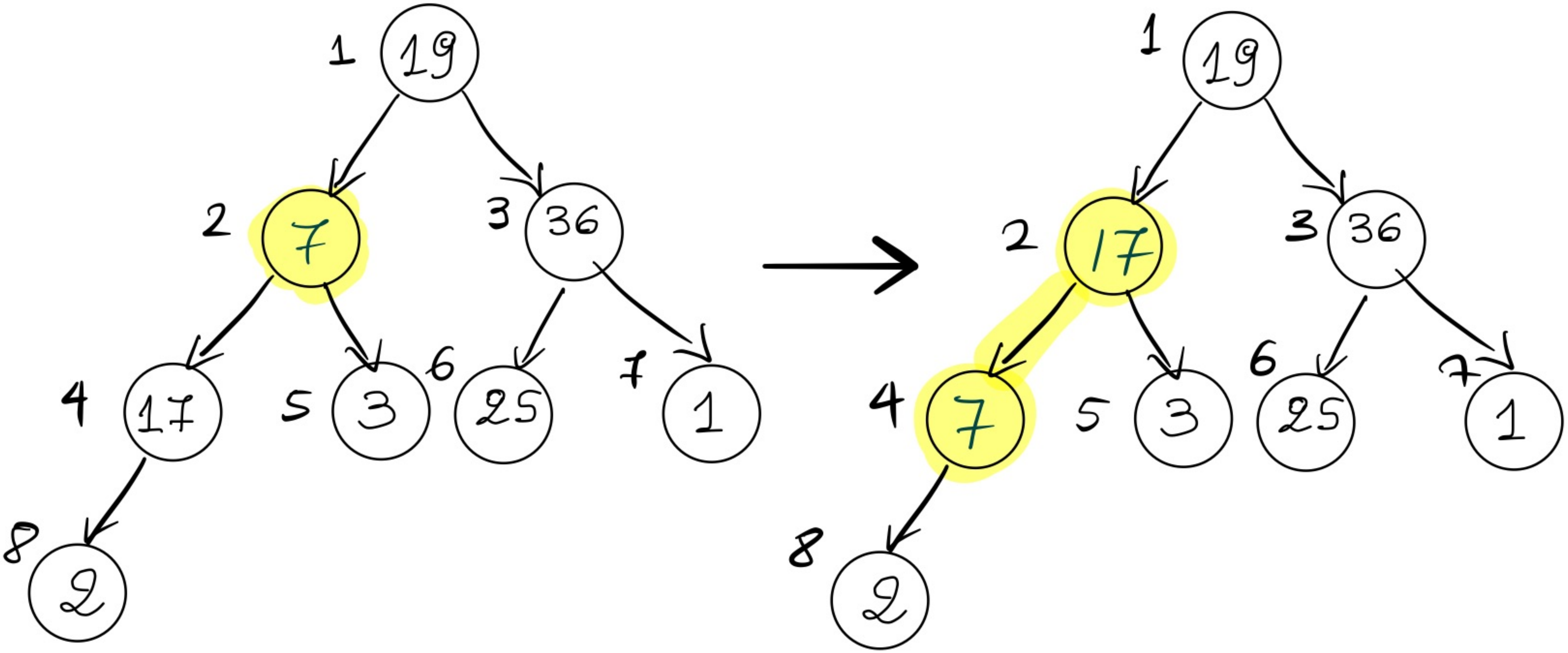


Practice Time

- Sink 7 to its appropriate place in this binary heap.



Answer



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. Why?

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

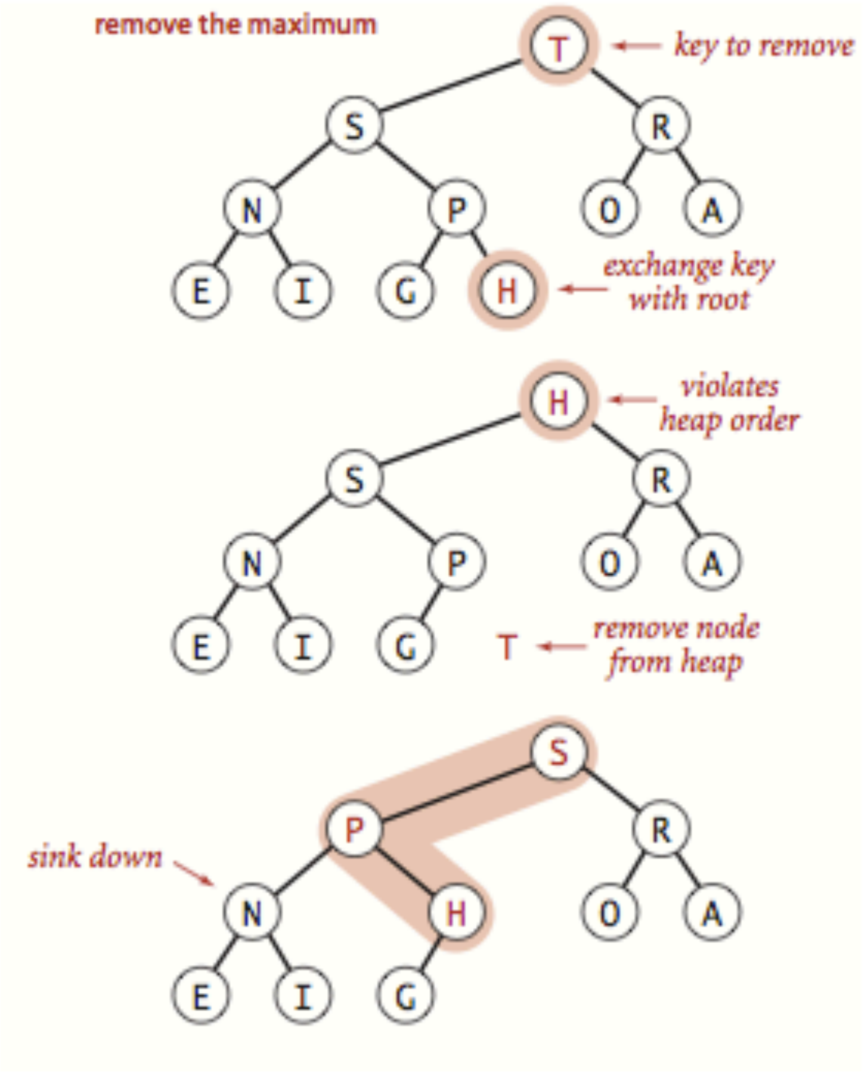
Binary heap: return (and delete) the maximum

- 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. Why?

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

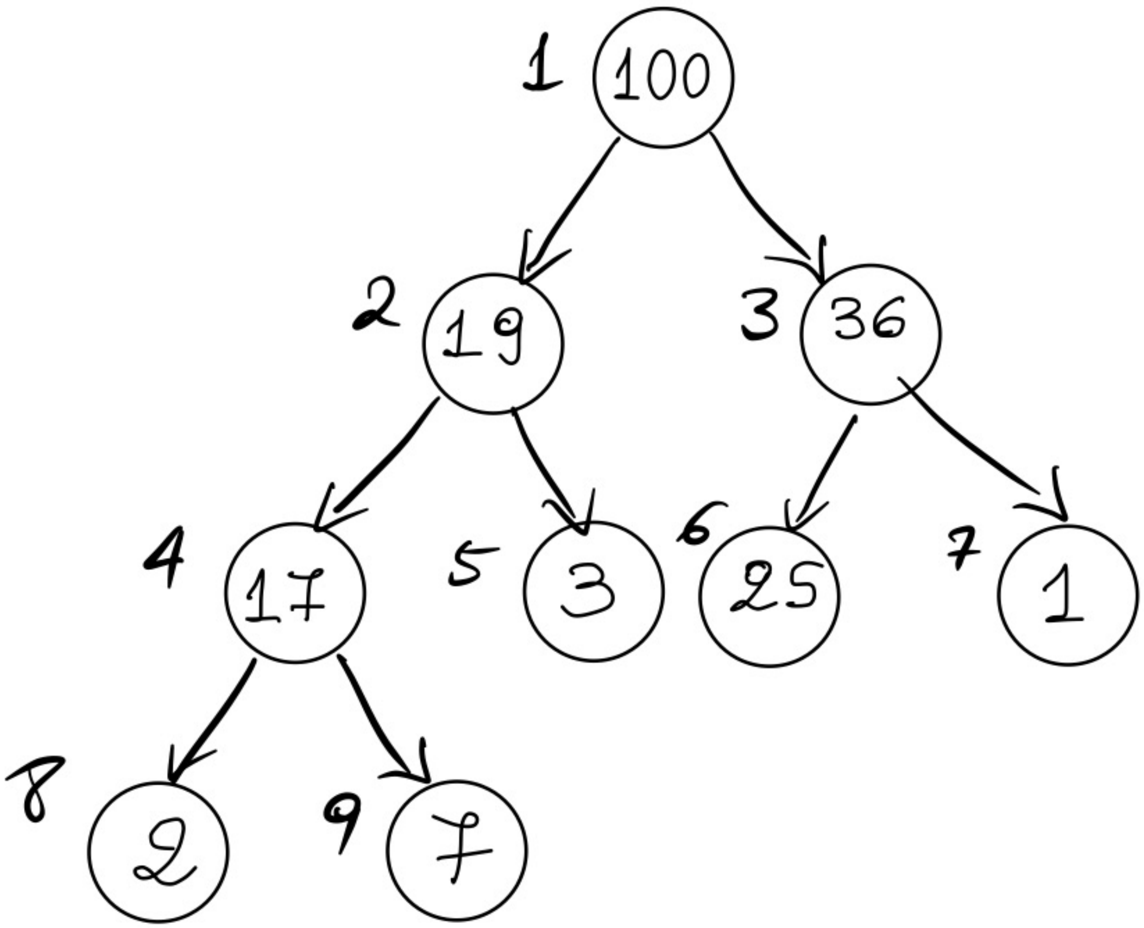
```
private void sink(int k) {
    while (2*k <= n) {
        int j = 2*k;
        if (j < n && less(j, j+1))
            j++;
        if (!less(k, j))
            break;
        exch(k, j);
        k = j;
    }
}
```

Binary heap: delete and return maximum

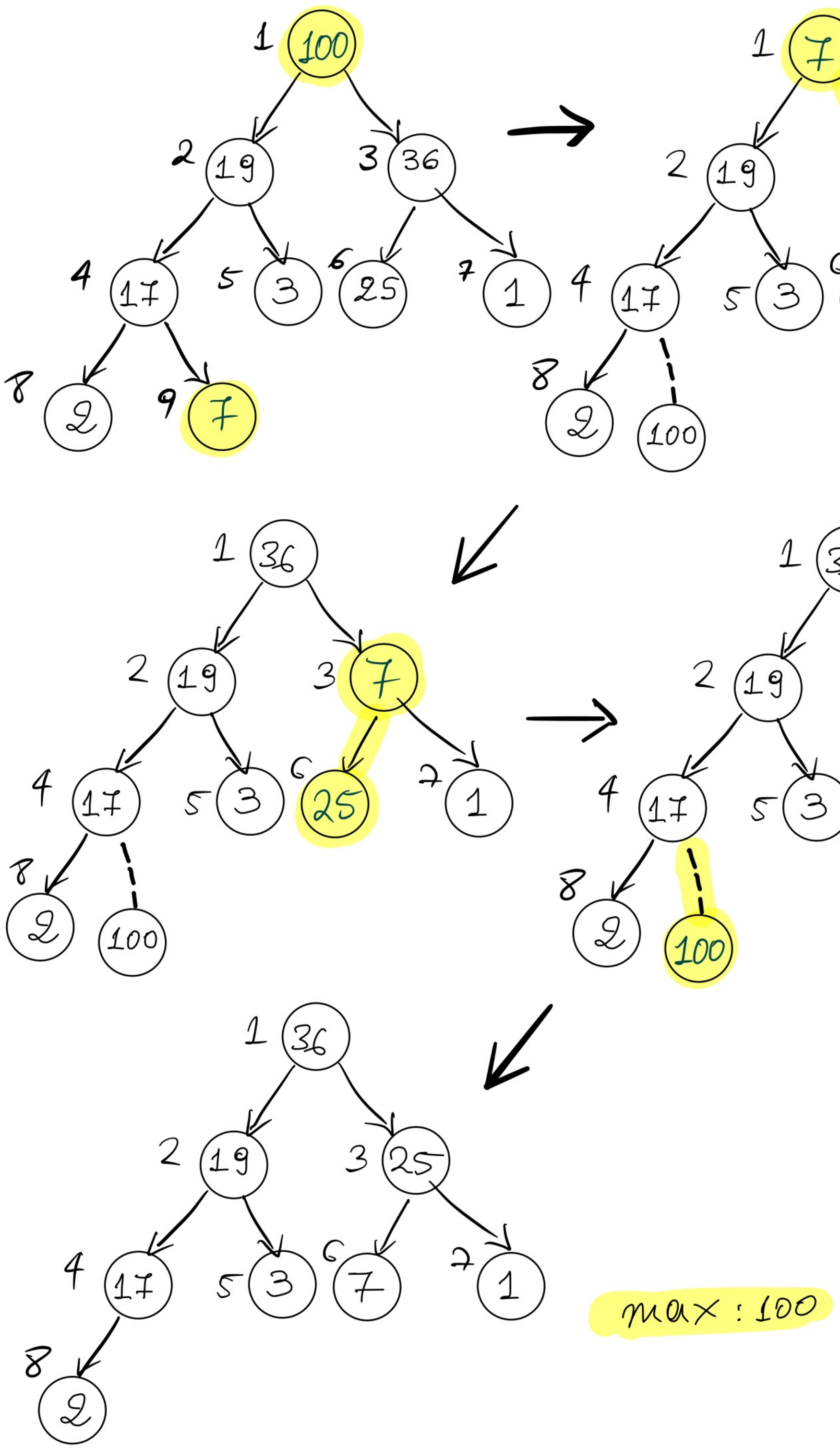


Practice Time

- Delete max (and return it!)



Answer



max : 100

Things to remember about runtime complexity of heaps

- Insertion is $O(\log n)$.
- Delete max is $O(\log n)$.
- Space efficiency is $O(n)$.

Things to remember about runtime complexity of heaps

- Insertion is $O(\log n)$.
- Delete max is $O(\log n)$.
- Space efficiency is $O(n)$.
  - Array with complete tree

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

2.4 BINARY HEAP DEMO

Lecture 17: Heaps, Priority Queues and Heapsort

- Binary Heaps
- Priority Queue
- Heapsort

Priority Queue ADT

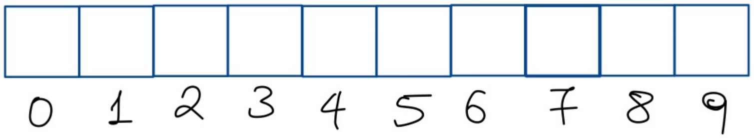
- Service best element first
    - Compared to FIFO or LIFO
- Two operations:
    - Delete (return) the maximum
    - Insert
- Applications: load balancing and interruption handling in OS, Huffman codes for compression, A* search for AI, Dijkstra's and Prim's algorithm for graph search, etc.
- How can we implement a priority queue efficiently?
    - Unordered array, Ordered array, Binary Heap

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).

PRIORITY QUEUE

```
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;   }   // Insert into index n

    public Key delMax() {
        int max = 0;
        for (int i = 1; i < n; i++)
            if (less(max, i)) max = I;    // Find max element
        exch(max, n-1);                   // Exchange max with last element

        return pq[–n];                    // Return last element
    }
    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;
    }
}
```



Practice Time

- Given an empty array of capacity 10, perform the following operations in a priority queue based on an unordered array (lazy approach):

1. Insert P
2. Insert Q
3. Insert E
4. Delete max
5. Insert X
6. Insert A
7. Insert M
8. Delete max
9. Insert P
10. Insert L
11. Insert E
12. Delete max

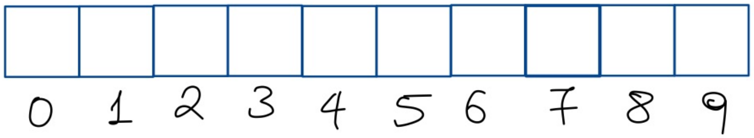
Answer

0	1	2	3	4	5	6	7	8	9	
P										insert P
P	Q									insert Q
P	Q	E								insert E
P	E	~~Q~~								delete-max → Q
P	E	X								insert X
P	E	X	A							insert A
P	E	X	A	M						insert M
P	E	M	A	~~X~~						delete-max → X
P	E	M	A	P						insert P
P	E	M	A	P	L					insert L
P	E	M	A	P	L	E				insert E
E	E	M	A	P	L	~~E~~				delete-max → P

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).

```
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];                   // Empty element is at index i
            i--;
        }
        pq[i+1] = key;                         // I+1 to get to the empty element
        n++;
    }

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



Practice Time

- Given an empty array of capacity 10, perform the following operations in a priority queue based on an ordered array (eager approach):

1. Insert P
2. Insert Q
3. Insert E
4. Delete max
5. Insert X
6. Insert A
7. Insert M
8. Delete max
9. Insert P
10. Insert L
11. Insert E
12. Delete max

Answer

0	1	2	3	4	5	6	7	8	9	
P										insert P
P	Q									insert Q
E	P	Q								insert E
E	P	~~Q~~								delete-max → Q
E	P	X								insert X
A	E	P	X							insert A
A	E	M	P	X						insert M
A	E	M	P	~~X~~						delete-max → X
A	E	M	P	P						insert P
A	E	L	M	P	P					insert L
A	E	E	L	M	P	P				insert E
A	E	E	L	M	P	~~P~~				delete-max → P

Option 3: Binary heap

- 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 delete max can be achieved in $O(1)$ running time.
- Priority queues are synonyms to binary heaps.

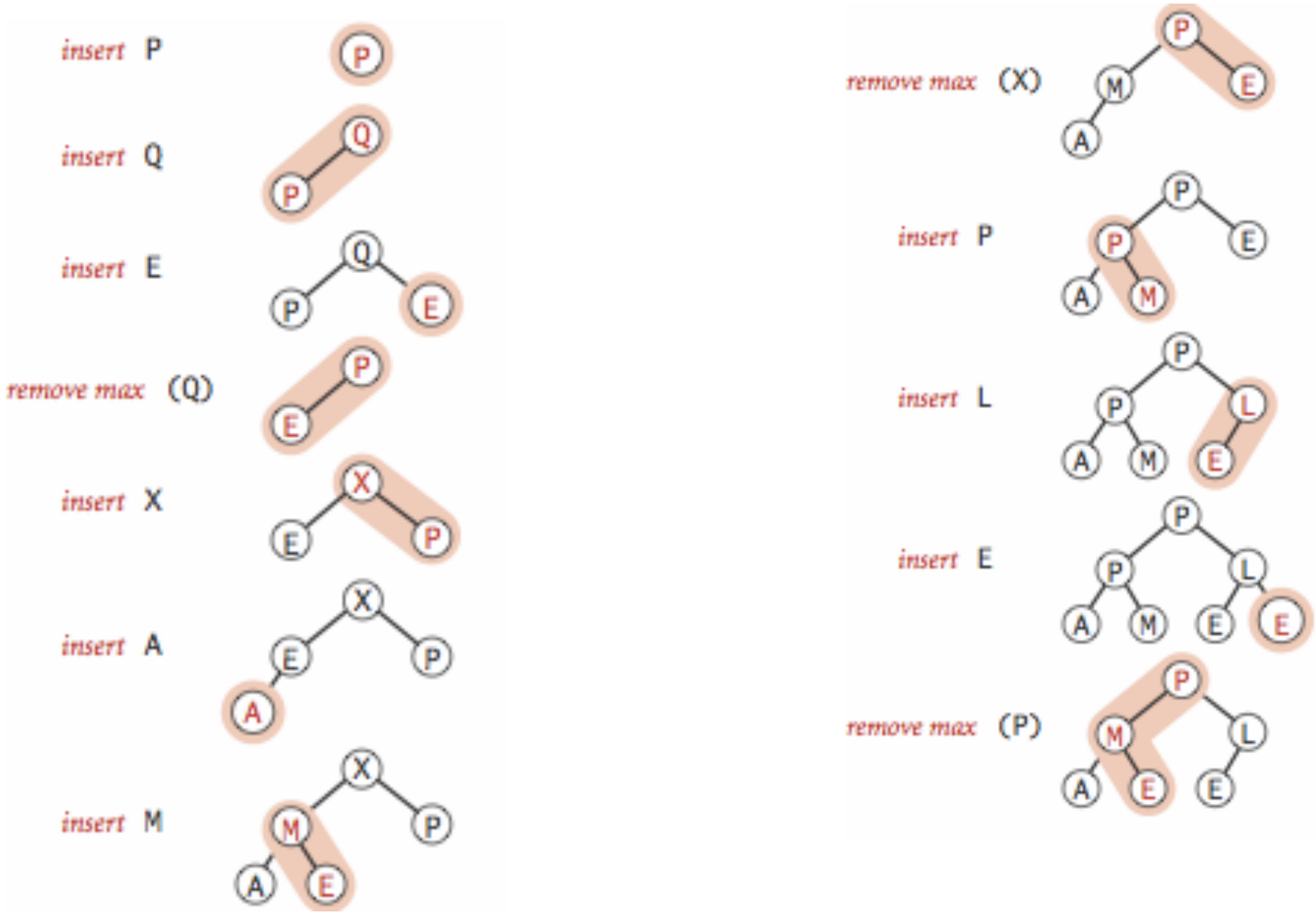
Stopped here

Practice Time

- Given an empty binary heap that represents a priority queue, perform the following operations:

1. Insert P
2. Insert Q
3. Insert E
4. Delete max
5. Insert X
6. Insert A
7. Insert M
8. Delete max
9. Insert P
10. Insert L
11. Insert E
12. Delete max

Answer


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)

Lecture 22: Priority Queues and Heapsort

- Priority Queue
- Heapsort

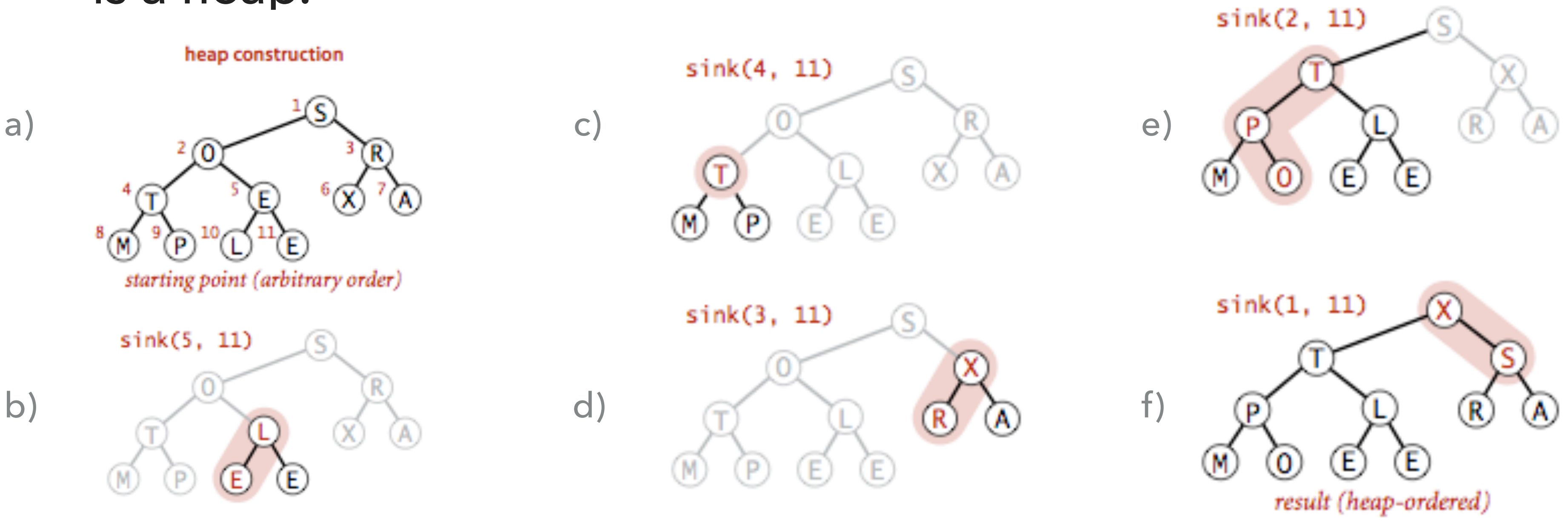
Basic plan for heap sort

- Use a priority queue to develop a sorting method that works in two steps:
- 1) Heap construction: build a binary heap with all n keys that need to be sorted.
- 2) Sortdown: repeatedly remove and return the maximum key.

$O(n)$ Heap construction

- Construct complete binary tree with elements
- Ignore all leaves (indices n/2+1,…,n).

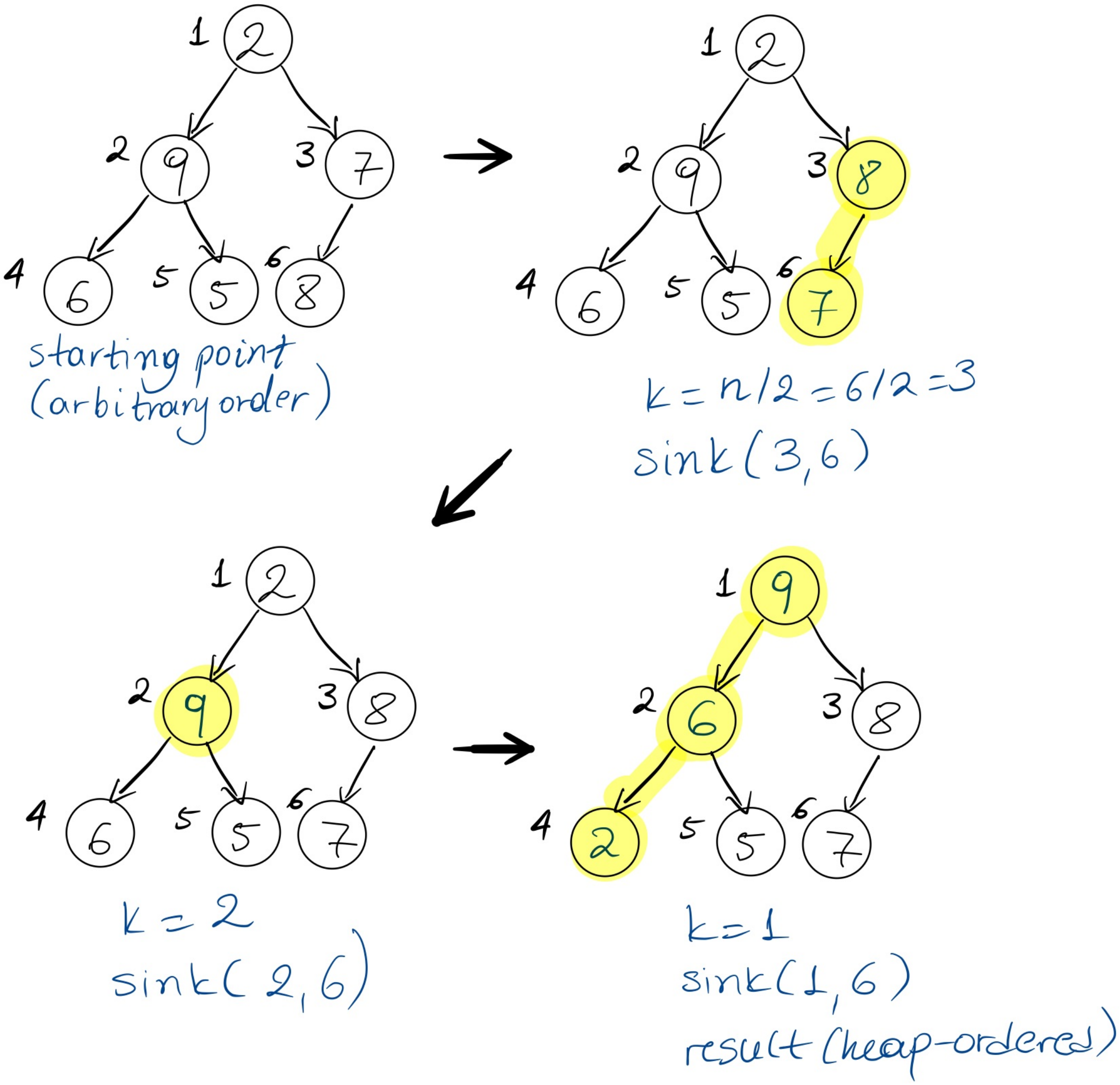
-
  ```
  for(int k = n/2; k >= 1; k--)
          sink(a, k, n);
  ```
- **Key insight**: After `sink(a,k,n)` completes, the subtree rooted at k is a heap.



Practice Time

- Run the first step of heapsort, heap construction, on the array [2,9,7,6,5,8].

Answer: Heap construction



starting point (arbitrary order)

k = n/2 = 6/2 = 3

sink(3,6)

k = 2

sink(2,6)

k = 1

sink(1,6)

result (heap-ordered)

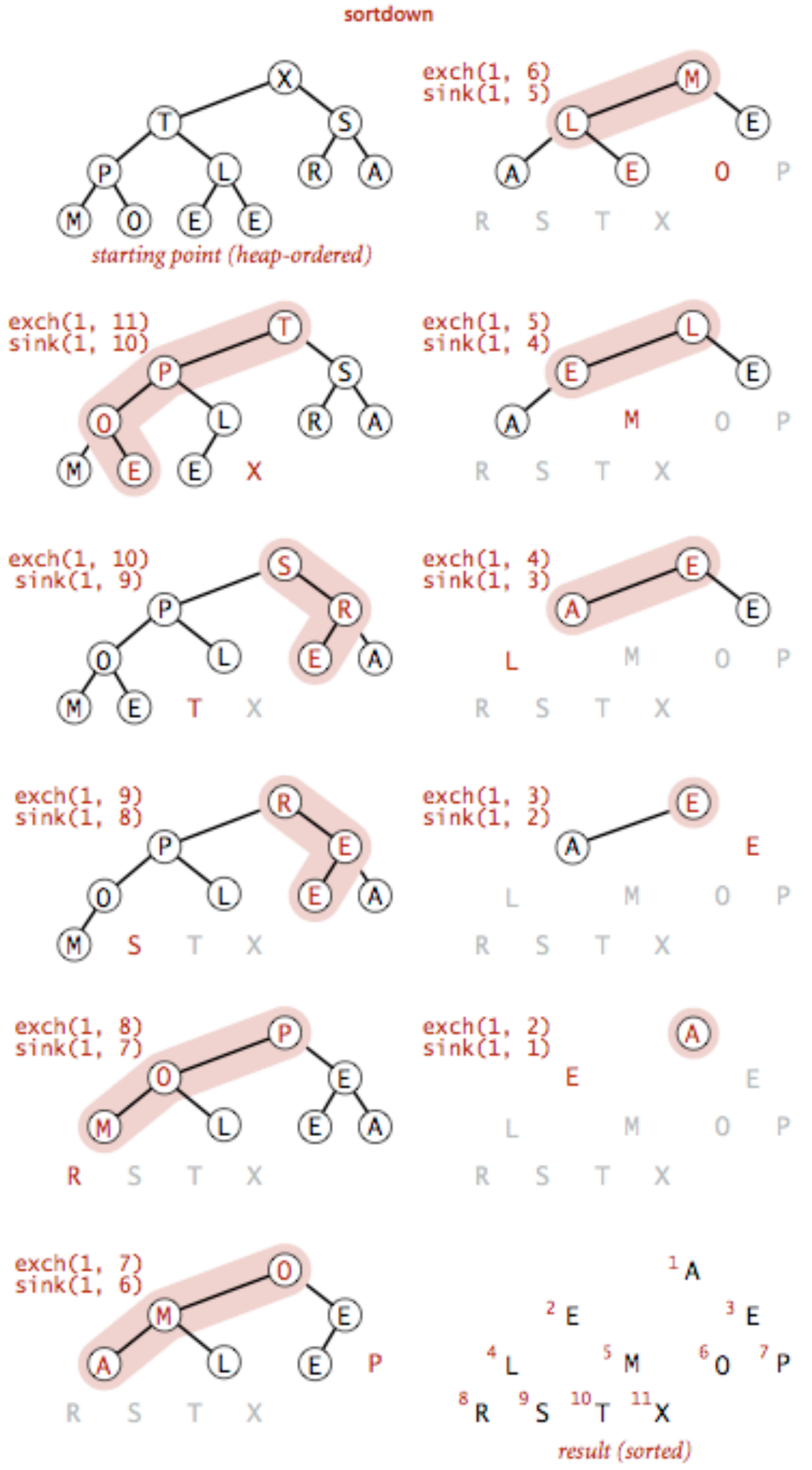
Sortdown

- 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.

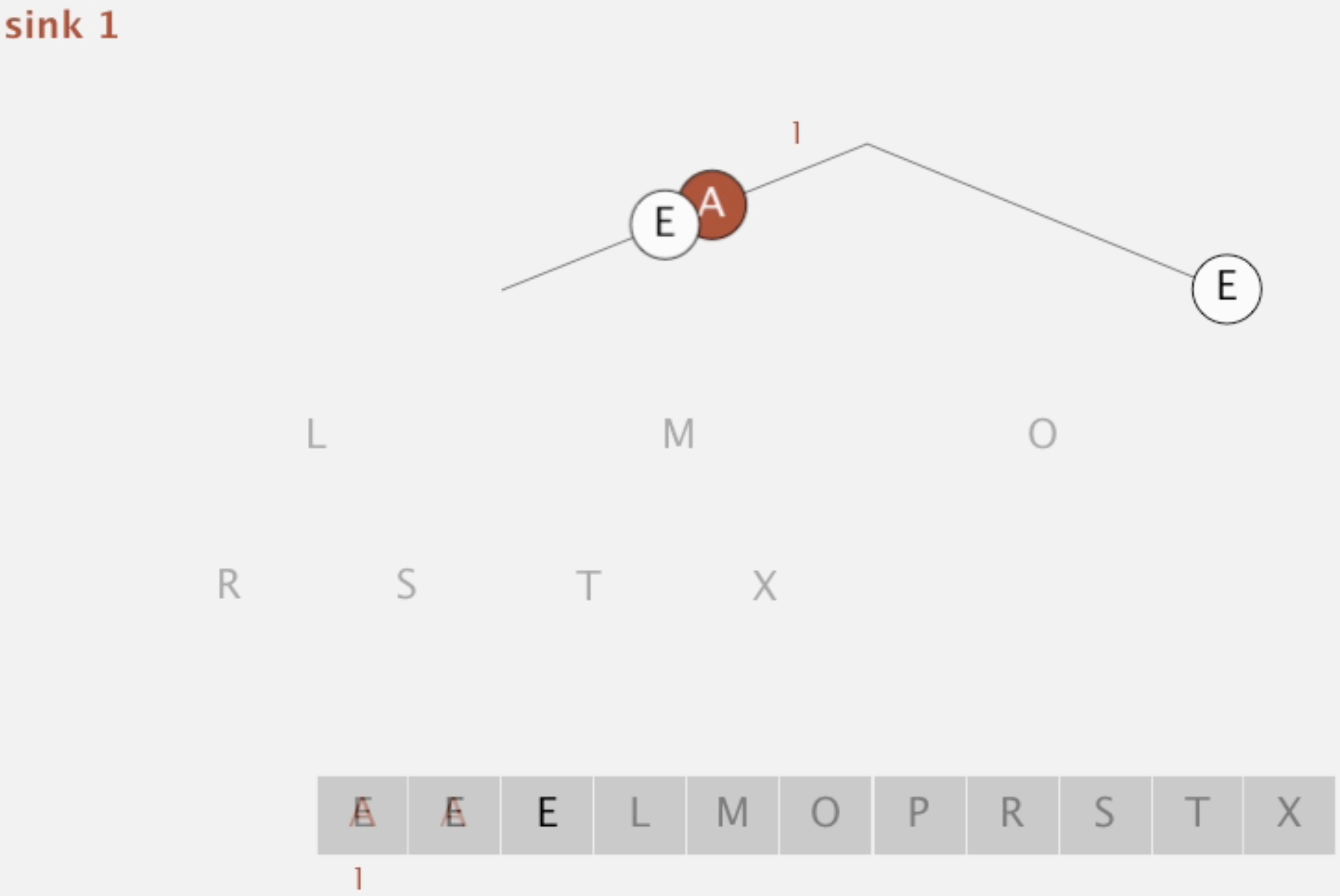
Sortdown

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



Heapsort demo

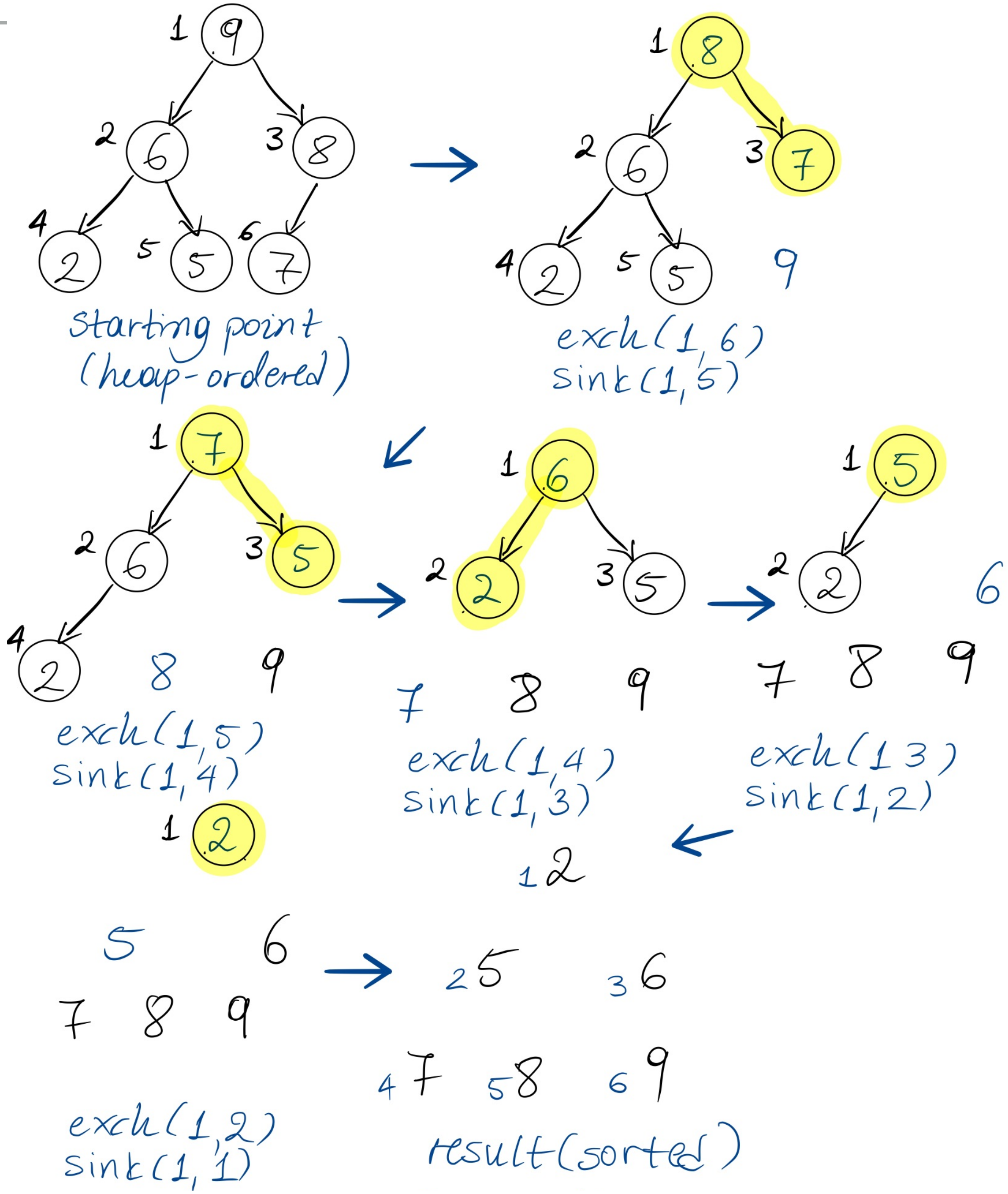
Sortdown. Repeatedly delete the largest remaining item.



Practice Time

- Given the heap you constructed before, run the second step of heapsort, sortdown, to sort the array [2,9,7,6,5,8].

Answer: Sortdown



Heapsort analysis

- Heap construction makes $O(n)$ exchanges and $O(n)$ compares.
- Sortdown and therefore the entire heap sort $O(n \log n)$ 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 in terms of Big-O, but:
  - Inner loop longer than quick sort.
  - Poor use of cache. Why?
  - Not stable.

Sorting: Everything you need to remember about it!

Which Sort	In place	Stable	Best	Average	Worst	Remarks
Selection	X		$O(n^2)$	$O(n^2)$	$O(n^2)$	n exchanges
Insertion	X	X	$O(n)$	$O(n^2)$	$O(n^2)$	Use for small arrays or partially ordered
Merge		X	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Guaranteed performance; stable
Quick	X		$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$n \log n$ probabilistic guarantee; fastest!
Heap	X		$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Guaranteed performance; in place

Lecture 22: Priority Queues and Heapsort

- Priority Queue
- Heapsort

Readings:

- Textbook:
  - Chapter 2.4 (Pages 308-327), 2.5 (336-344)
- Website:
  - Priority Queues: https://algs4.cs.princeton.edu/24pq/
- Visualization:
  - Create (nlogn) and heapsort: https://visualgo.net/en/heap

Practice Problems:

- 2.4.1-2.4.11. Also try some creative problems.

Readings:

- Textbook:
  - Chapter 2.4 (Pages 308-327)
- Website:
  - Priority Queues: https://algs4.cs.princeton.edu/24pq/
- Visualization:
  - Insert and ExtractMax: https://visualgo.net/en/heap

Practice Problems:

- Practice with traversals of trees and insertions and deletions in binary heaps