Lab 9: Checkpoint 2 study guide

Sorting

Information

- Checkpoint 2 is Tuesday, April 8 in class.
- You can bring one hand-written (ok hand-written on tablets and then printed) back and front sheet of paper (i.e. two pages). NO slides shrunk and copy pasted.
- Review lecture slides along with slides on practice problems and links to code. Go over quizzes, labs, and assignments. Use the practice problems in this presentation. If you want to read in more depth, use the recommended textbook (extra copies for in-lab use in the dept library)
- Practice writing code on paper.

Review

Checkpoint II Review

- Sorting
- Heaps/Priority Queues
- Dictionaries
- Misc
- Practice Problems
- Answers

Sorting

- Selection sort
- Insertion sort
- Merge sort
- Quick sort
- Heap sort

Sorting

- Given an array of n items, sort them in non-descending order based on a comparable key.
- Cost model counts comparisons and exchanges (or array accesses).
- Not in place: If linear extra memory is required.
- Stable: If duplicate elements stay in the same order that they appear in the input.
- Practice: https://visualgo.net/en/sorting (minus quick sort).

Selection sort - Algorithm

```
public static <E extends Comparable<E>> void selectionSort(E[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int min = i;
            for (int j = i+1; j < n; j++) {
                if (a[j].compareTo(a[min])<0){
                    min = j;
                }
            }
            E temp = a[i];
            a[i]=a[min];
            a[min]=temp;
        }
    }
```

Selection sort – Key characteristics

- At the end of each iteration `i`:
  - `a[0…i]` is sorted.
  - no smaller item exists in `a[i+1…n-1]`.
- In-place.
- Not stable.
- $O(n^2)$ comparisons for best/average/worst case.
  - $O(n)$ exchanges.
- Slowest. Realistically, rarely used in practice unless small array and minimizing cost of exchanges is important.

Selection sort – Example

- Sort: 1,4,9,3,8,2.

i iteration	Result
0	1,4,9,3,8,2
1	1,2,9,3,8,4
2	1,2,3,9,8,4
3	1,2,3,4,8,9
4	1,2,3,4,8,9
5	1,2,3,4,8,9

Sorting

- Selection sort
- Insertion sort
- Merge sort
- Quick sort
- Heap sort

Insertion sort - Algorithm

```
public static <E extends Comparable<E>> void insertionSort(E[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
           for (int j = i; j > 0; j--) {
               if(a[j].compareTo(a[j-1])<0){
                   E temp = a[j];
                   a[j]=a[j-1];
                   a[j-1]=temp;
               }
               else{
                  break;
               }
           }
       }
 }
```

Insertion sort – Key characteristics

- At the end of each iteration `i`:
  - `a[0…i]` is partially sorted.
- In-place.
- Stable.
- $O(n^2)$ comparisons/exchanges for average/worst case.
- $O(n)$ comparisons and 0 exchanges for best case (already sorted array).
- Slow but in practice such little overhead that can be even faster than quick sort for small arrays. Often used below certain thresholds for merge sort and quick sort.

Insertion sort – Example

- Sort: 1,4,9,3,8,2.

i iteration	Result
0	1,4,9,3,8,2
1	1,4,9,3,8,2
2	1,4,9,3,8,2
3	1,3,4,9,8,2
4	1,3,4,8,9,2
5	1,2,3,4,8,9

Sorting

- Selection sort
- Insertion sort
- Merge sort
- Quick sort
- Heap sort

Merge sort - Algorithm

```java
private static <E extends Comparable<E>> void merge(E[] a, E[] aux, int lo, int mid, int hi) {
    for (int k = lo; k <= hi; k++){
        aux[k] = a[k];
    }
    int i = lo, j = mid + 1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) { // ran out of elements in the left subarray
            a[k] = aux[j++];
        } else if (j > hi) { // ran out of elements in the right subarray
            a[k] = aux[i++];
        } else if (aux[j].compareTo(aux[i]) < 0) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        }
    }
}
```

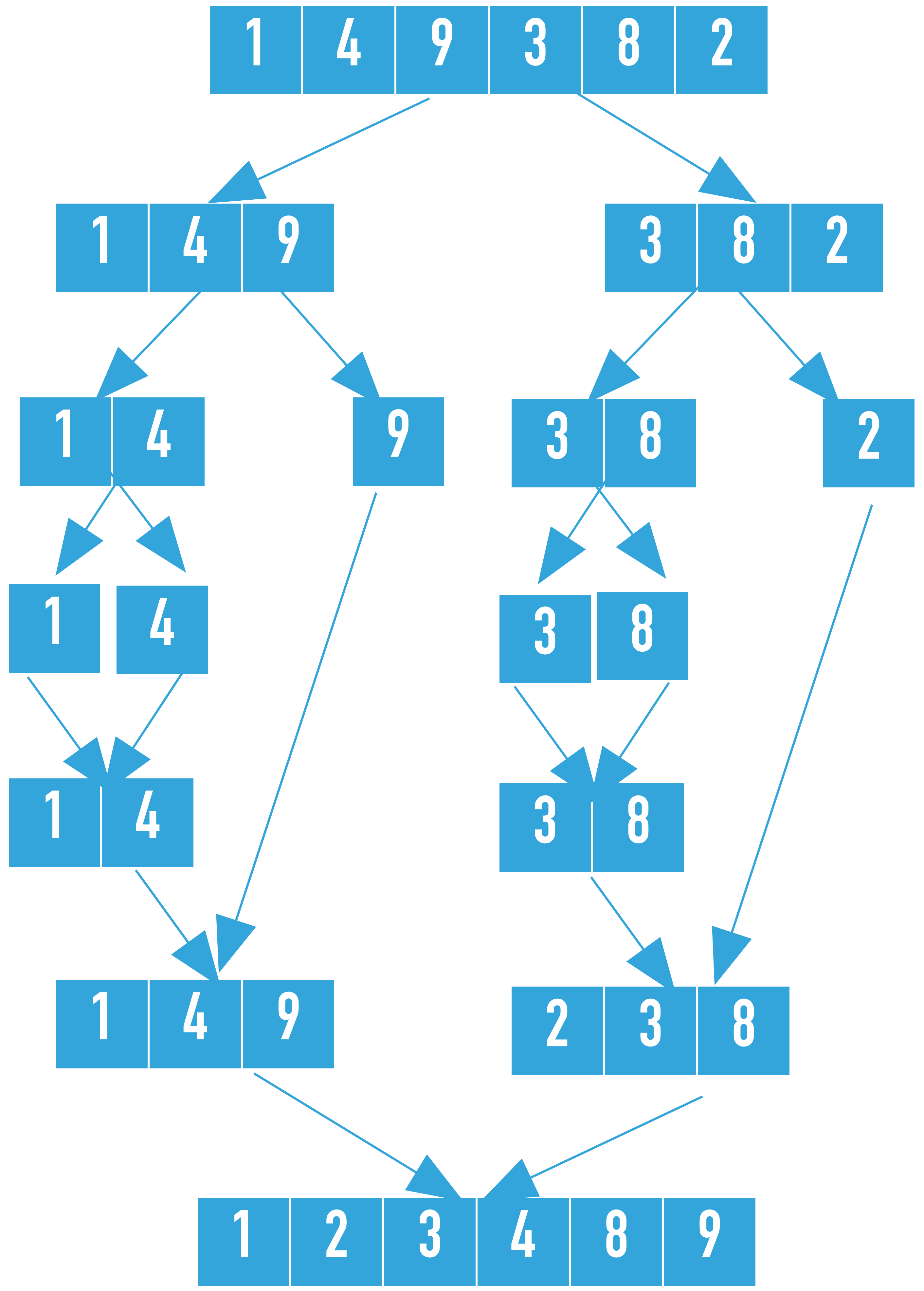
```java
public static <E extends Comparable<E>> void mergeSort(E[] a) {
    E[] aux = (E[]) new Comparable[a.length];
    mergeSort(a, aux, 0, a.length - 1);
}

private static <E extends Comparable<E>> void mergeSort(E[] a, E[] aux, int lo, int hi) {
    if (hi <= lo){
        return;
    }
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

Merge sort – Key characteristics

- Divide till you reach an array of a single element and conquer by merging two already-sorted subarrays into a sorted larger one.
- Not in-place, requires linear extra memory. On-disk sort assignment showed how to use the disk if memory is not enough.
- Stable.
- $O(n \log n)$ comparisons/array accesses for best/average/worst case.
- Stable performance, preferred for arrays of objects due to stability. Slower than quick sort on average. Not in-place so not good when memory is in short supply (e.g., embedded systems).

Merge sort - Example

- Sort: 1,4,9,3,8,2.



Sorting

- Selection sort
- Insertion sort
- Merge sort
- Quick sort
- Heap sort

Quick sort - Algorithm

```java
private static <E extends Comparable<E>> int partition(E[] a, int lo, int hi) {
        E pivot = a[lo]; // Choose leftmost element as pivot
        int i = lo + 1; // Start from the next element
        int j = hi;

        while (true) {
            // Move right until we find an element >= pivot
            while (i <= j && a[i].compareTo(pivot) <= 0) {
                i++;
            }
            // Move left until we find an element < pivot
            while (j >= i && a[j].compareTo(pivot) > 0) {
                j--;
            }
            // If pointers cross, break
            if (i > j) {
                break;
            }

            // Swap elements to ensure correct partitioning
            E temp = a[i];
            a[i] = a[j];
            a[j] = temp;

        }

        // Swap pivot into its correct position
        E temp = a[lo];
        a[lo] = a[j];
        a[j] = temp;

        return j; // Return final pivot position
    }
```

```java
public static <E extends Comparable<E>> void quickSort(E[] a) {
        quickSort(a, 0, a.length - 1);
    }

private static <E extends Comparable<E>> void quickSort(E[] a, int
lo, int hi) {
        if (lo < hi){
            int pivot = partition(a, lo, hi);
            quickSort(a, lo, pivot - 1);
            quickSort(a, pivot + 1, hi);
        }

    }
```

Quick sort - Key characteristics

- Swap smaller elements than pivot to go to left, and larger elements to go to right subarray.
- In-place.
- Not stable.
- $O(n \log n)$ comparisons/exchanges for best/average case.
- $O(n^2)$ comparisons/exchanges for worst case (already (reversely) sorted array, where pivot is always the smallest/largest element).
- Preferred for arrays of primitives since stability does not matter. Fastest on average but if unlucky quadratic (can avoid with high likelihood if shuffle first). In-place so good choice for memory efficient applications with tolerance for occasional slowdowns.

Quick sort – Example

- Sort: 4,1,9,3,8,2

Iteration 1

[4, 1, 9, 3, 8, 2]   i = 1, j = 5

swap 2 and 9 -> [4, 1, 2, 3, 8, 9]   i = 2, j = 5

swap 4 (pivot) and 3 -> [3, 1, 2, 4, 8, 9] i = 4, j = 3

new pivot is 3 for [3, 1, 2] and 8 for [8, 9]

Iteration 2

[3, 1, 2, X, X, X]   i = 1, j = 2

swap 3 (pivot) and 2 -> [2, 1, 3, X, X, X]   i = 3, j = 2

Iteration 3

[2, 1, X, X, X, X]   i = 1, j = 1

swap 2 (pivot) and 1 -> [1, 2, X, X, X, X]   i = 2, j = 1

Iteration 4

Next pivot is 1, nothing happens - single item already sorted

Iteration 5

[X, X, X, X, 8, 9]   i = 5, j = 5

Swap 8 (pivot) with itself -> [X, X, X, X, 8, 9]   i = 5, j = 4

Iteration 6

Next pivot is 9, nothing happens - single item already sorted

[1, 2, 3, 4, 8, 9]

Sorting

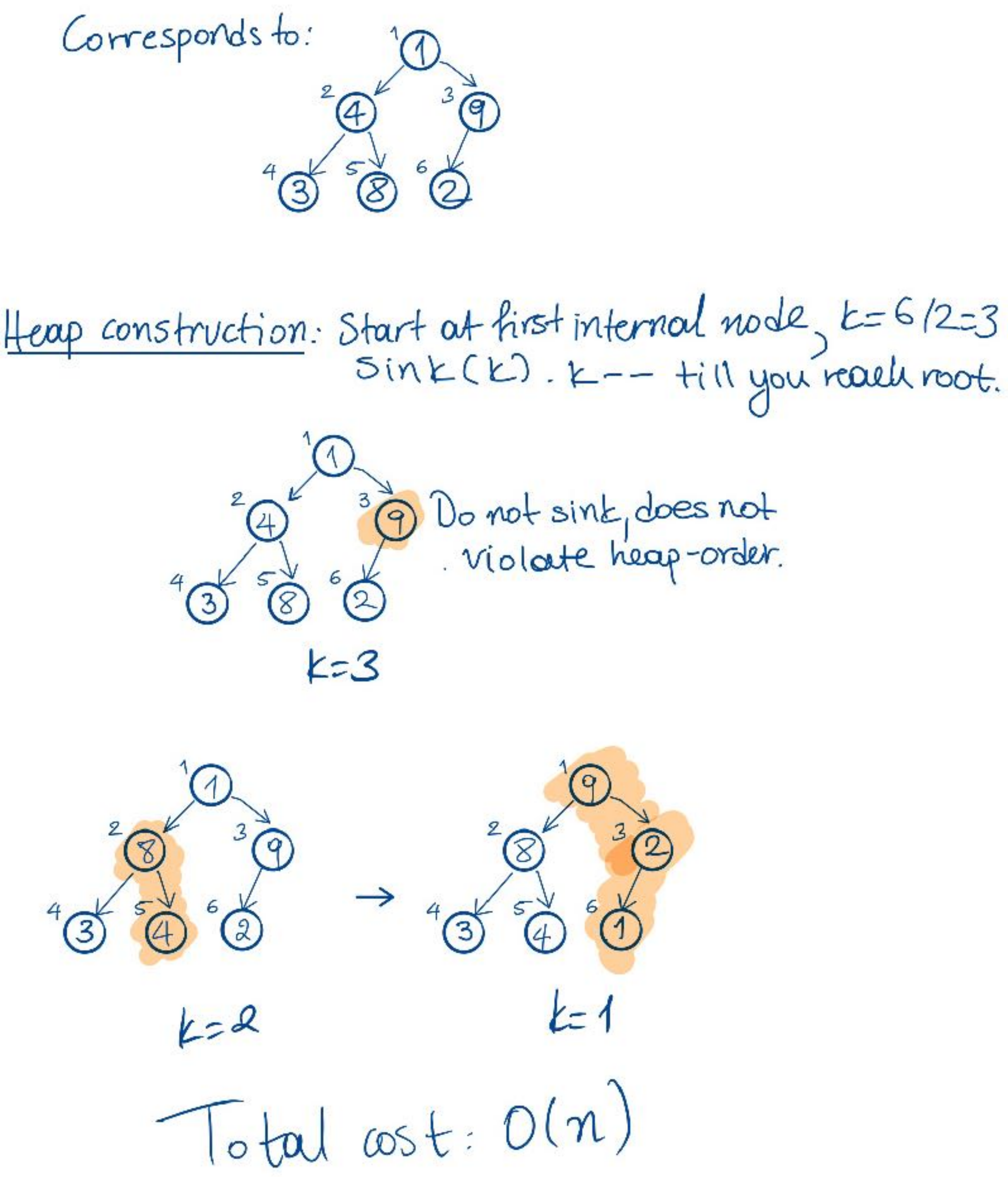
- Selection sort
- Insertion sort
- Merge sort
- Quick sort
- Heap sort

Heap sort - Key characteristics

- Heap construction in $O(n)$: heapify subtrees rooted in internal nodes in reverse order.
  - There is also a slower $O(n \log n)$ version with n insertions. Avoid it.
  - Sortdown in $O(n \log n)$: Repeat: exchange root with last element and sink.
- In-place.
- Not stable.
- $O(n \log n)$ comparisons/exchanges for best/average/worst case.
- Slower than merge sort (and quick sort) but does not require extra memory. Good choice for memory efficient applications that need stable performance.

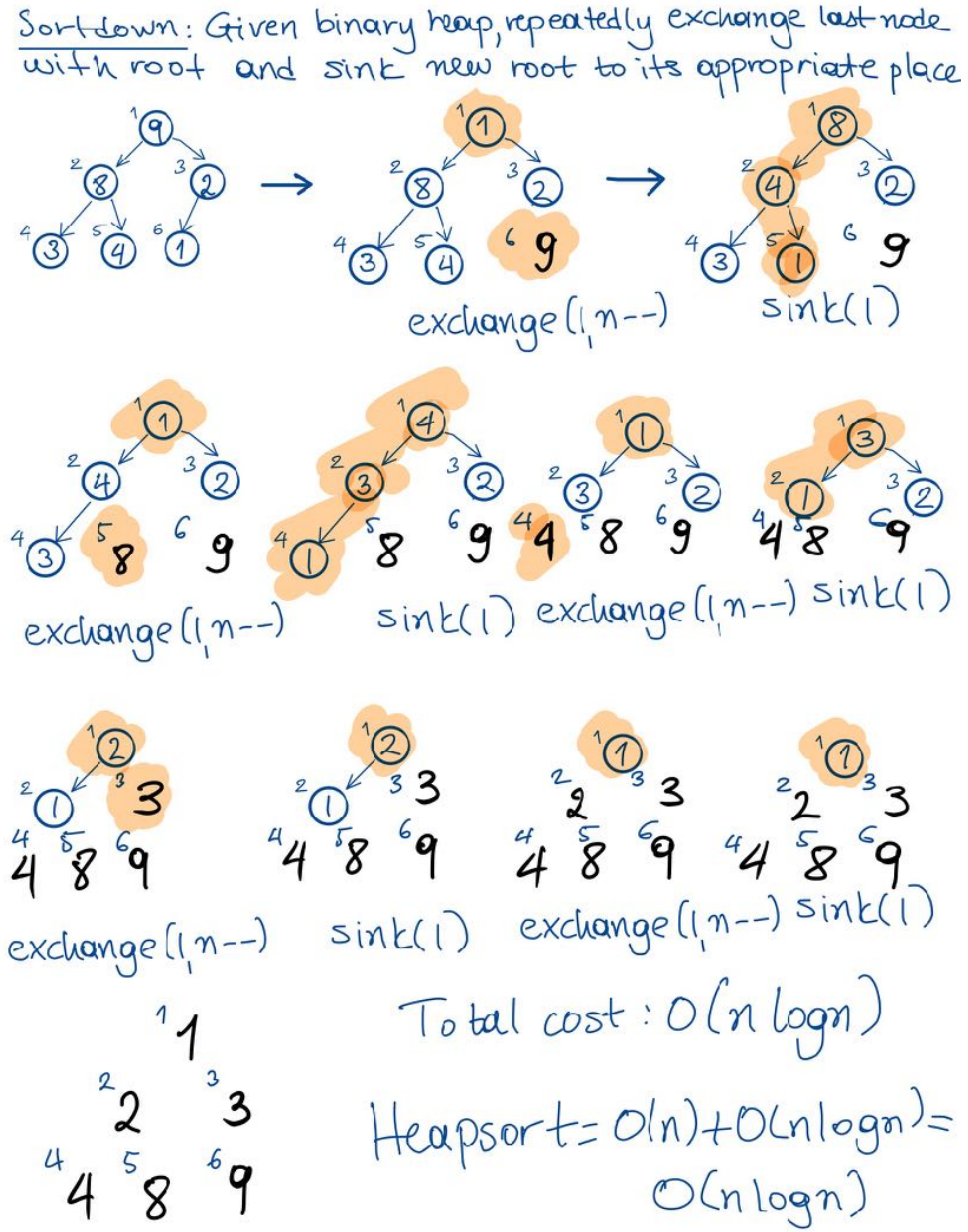
Heap sort - Example

- Sort: 1,4,9,3,8,2



Heap sort - Example

- Sort: 1,4,9,3,8,2,



Sorting: Everything you need to remember about it!

Which Sort	In place	Stable	Best	Average	Worst	Memory	Remarks
Selection	X		$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$\Theta(1)$	n exchanges
Insertion	X	X	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$\Theta(1)$	Fastest if almost sorted or small
Merge		X	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$\Theta(n)$	Guaranteed performance; stable
Quick	X		$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$\Theta(\log n)$	$n \log n$ probabilistic guarantee; fastest in practice
Heap	X		$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$\Theta(1)$	Guaranteed performance; in place

Heaps/Priority Queues

- Insertion
- Deletion

Heaps

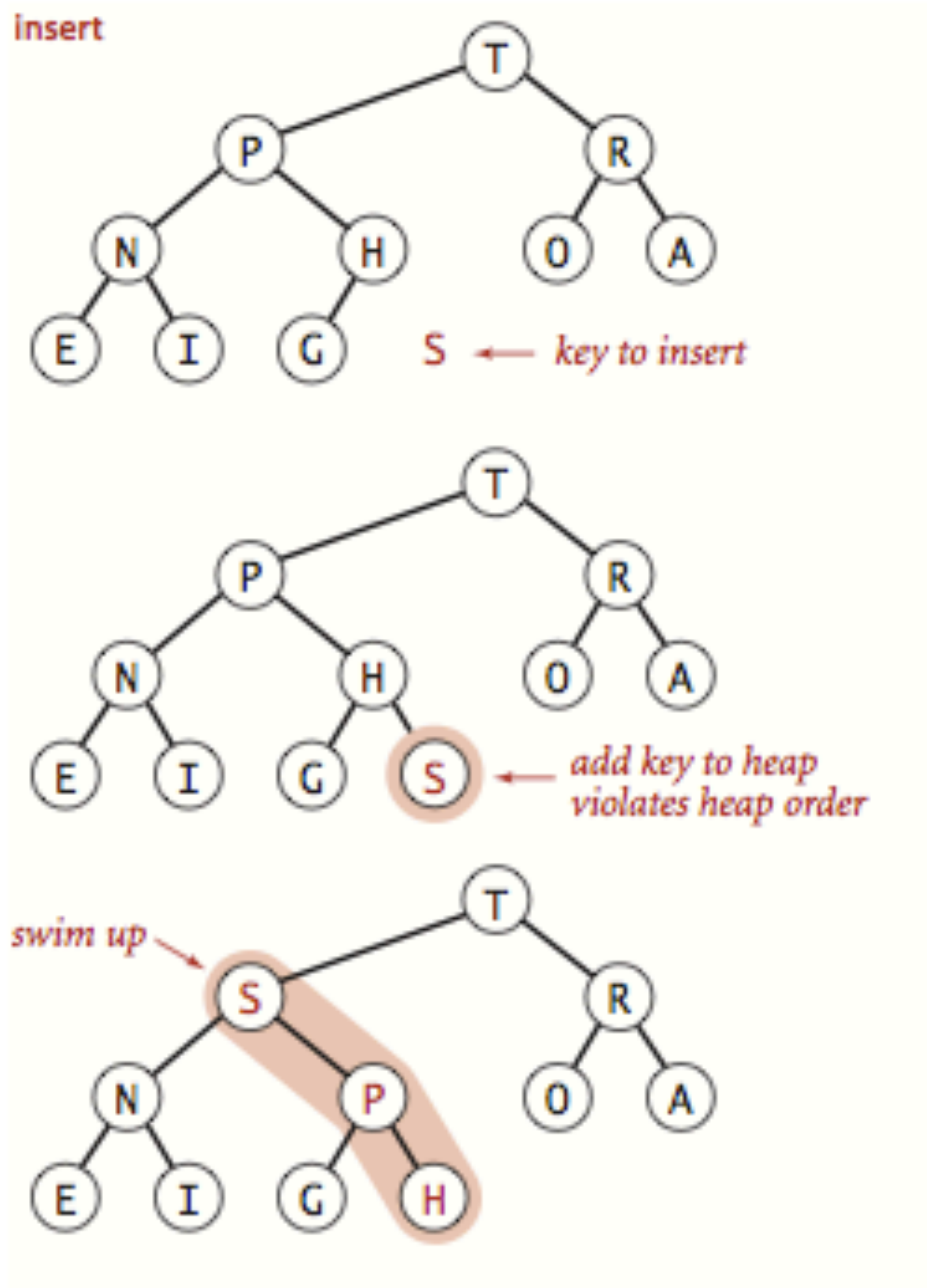
- Array representation of binary trees (at most 2 children for each node) which are complete ($O(logn)$ minimal height and nodes in last level as left as possible) and heap-ordered (every node is larger/equal to both of its children - if any).
- For node k, left child can be found at 2k, right child at 2k+1, and parent at k/2. Elements start at index 1.
- Heaps and priority queues are often considered synonyms.
- Practice: https://visualgo.net/en/heap (including heap sort).

Heaps

- Insertion
- Deletion

Heaps - Insertion

Insert node at last level, as left as possible (or create a new level if last level is full). Swim newly-added node to its proper place so that heap-ordered property is satisfied.
At most $O(\log n)$ comparisons.

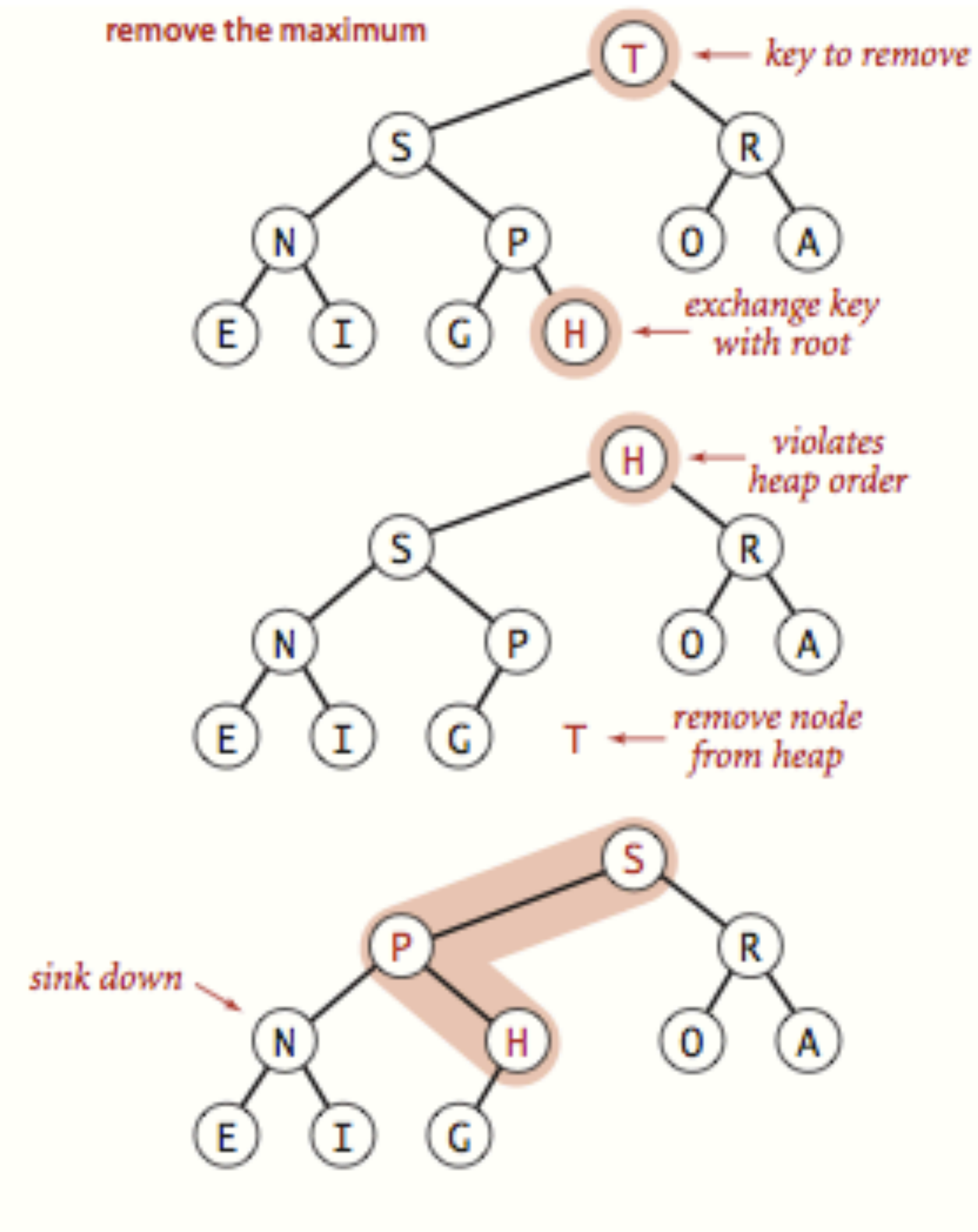


Heaps

- Insertion
- Deletion

Heaps - delete max

Exchange root with last element.
Sink down the new root to its proper place so that heap-ordered property is satisfied. Nullify index of deleted element and return it.
At most $O(\log n)$ comparisons.



Dictionaries

- Binary search trees
- B-trees

Dictionaries

- (Possibly ordered by key) collections of key-value pairs. Keys are comparable and unique. Values cannot be null.
- Ultimate goal is to achieve fast search based on key.
- Support insertion, deletion, and possibly ordered operations.

Binary search trees

- Binary trees with symmetric order (every node contains key larger than **all** keys in left subtree and smaller than **all** keys in right subtree).
- Height can vary from $O(\log n)$ (compact like complete trees) all the way to $O(n)$ (sticks/twigs).
- Practice: https://visualgo.net/en/bst

```java
public class BST<Key extends Comparable<Key>, Value> {
   private Node root;                // root of BST

   private class Node {
        private Key key;             // sorted by key
        private Value val;           // associated value
        private Node left, right;    // roots of left and right subtrees
        private int size;            // #nodes in subtree rooted at this

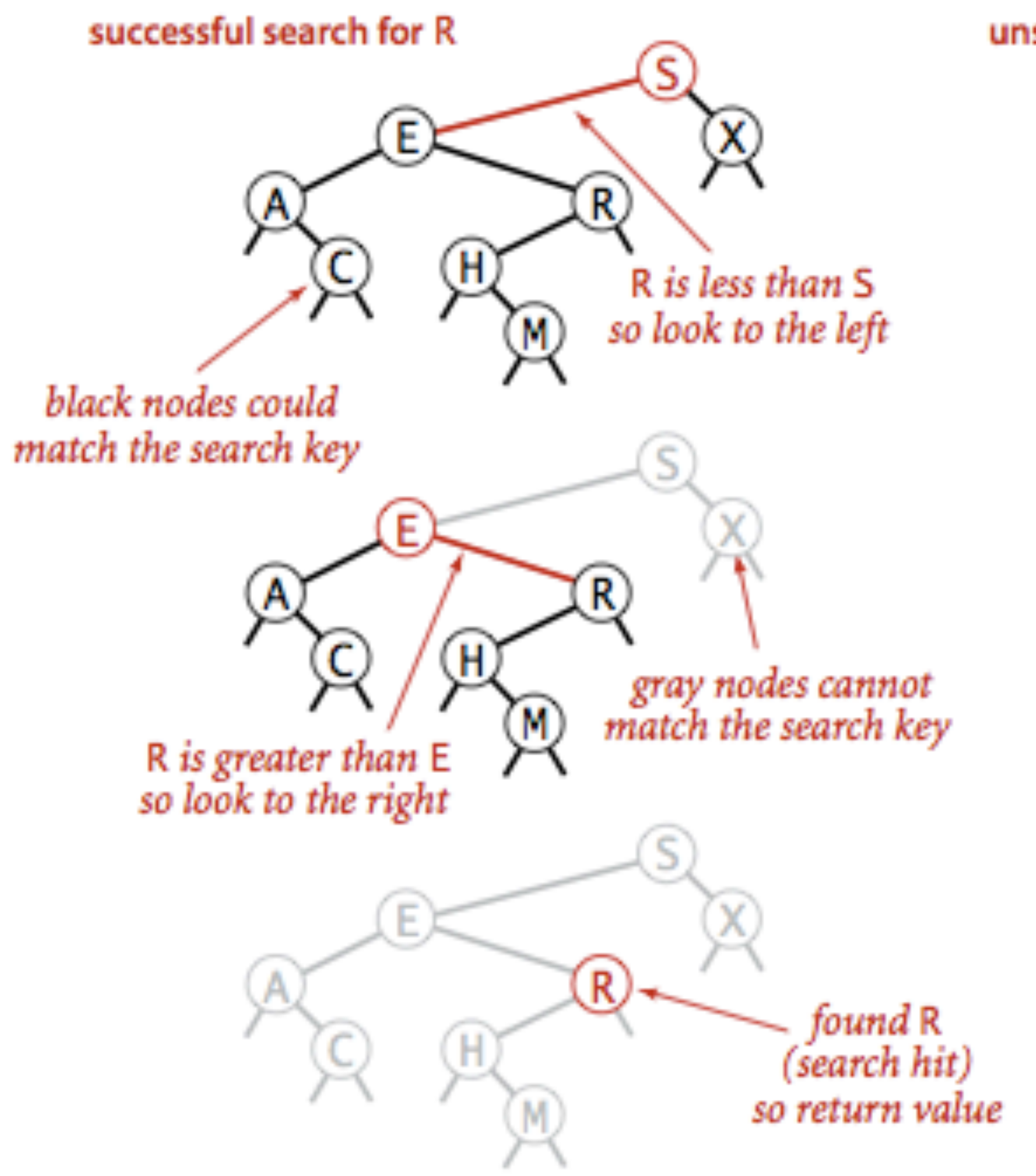
        public Node(Key key, Value val, int size) {
            this.key = key;
            this.val = val;
            this.size = size;
        }
    }
```

Binary search trees – search

- Compare key with root node. Smaller? Go left. Larger? Go right.
- Search hit: If found node with key you're looking for, return associated value.
- Search miss: reached a null node, return null.

```
private Value get(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return get(x.left, key);
    else if (cmp > 0) return get(x.right, key);
    else return x.val;
}
```

Binary search trees – search



Successful (left) and unsuccessful (right) search in a BST

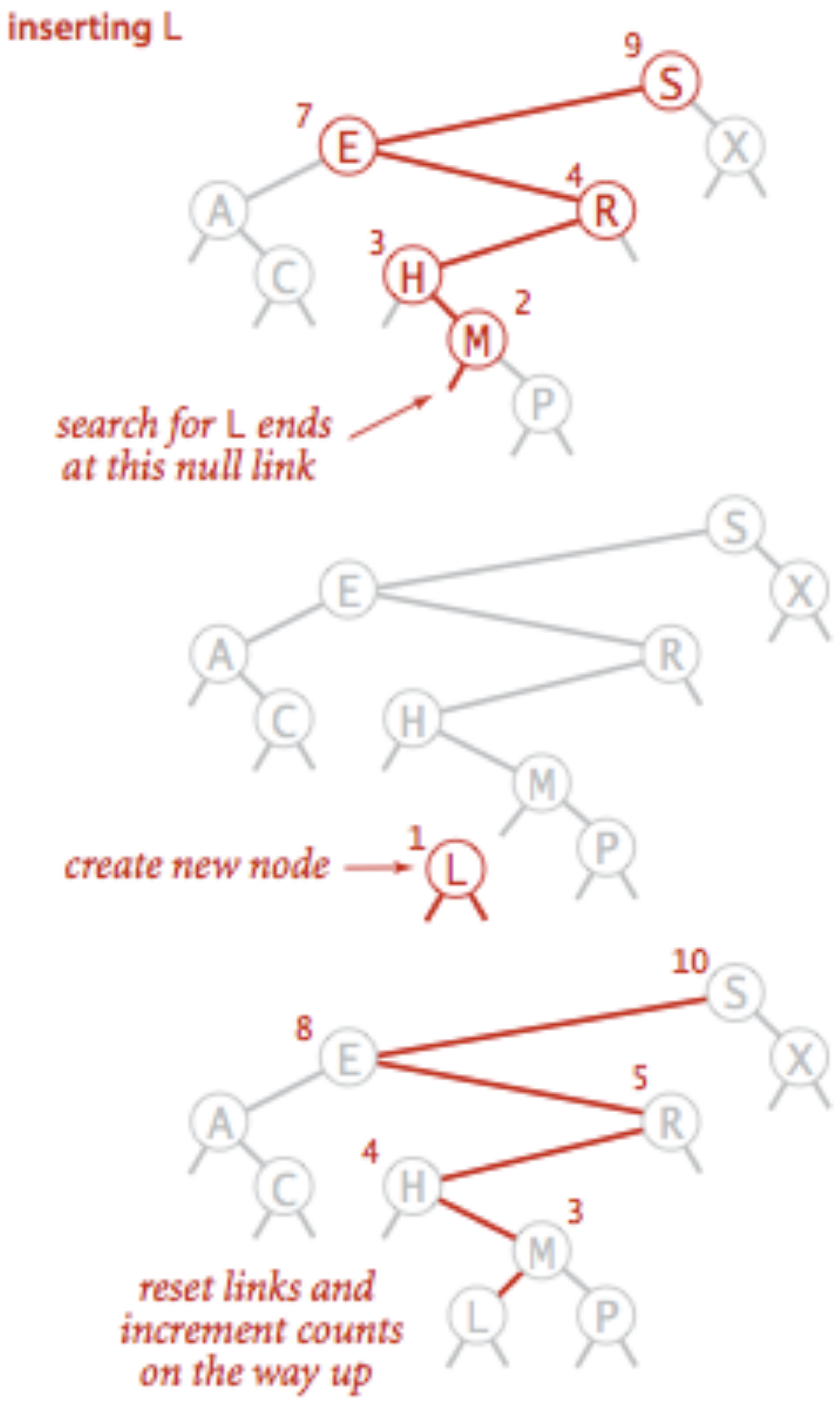
Binary search trees – insertion

- Compare key with root node. Smaller? Go left. Larger? Go right.
- If found node with same key, update value.
- If reached a null node, insert (key,value) pair.

```
public void insert(Key key, Value val) { //recursive implementation
        root = insert(root, key, val);
    }

    // helper (@returns root of subtree at x)
    private Node insert(Node x, Key key, Value val) {
        if (x == null) return new Node(key, val, 1); //empty subtree, insert new node
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x.left = insert(x.left, key, val);
        else if (cmp > 0) x.right = insert(x.right, key, val);
        else x.val = val; //update existing node
        x.size = size(x.left) + size(x.right) + 1; //update size
        return x;
    }
```

Binary search trees – insertion

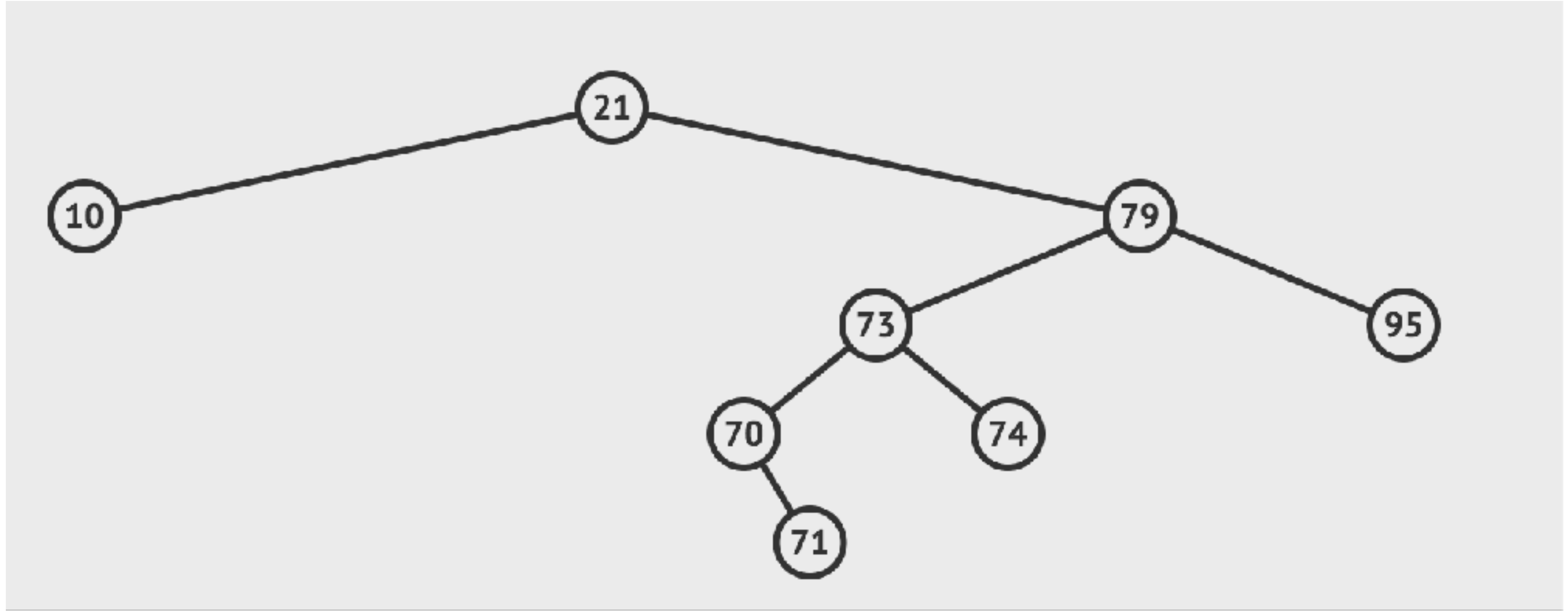


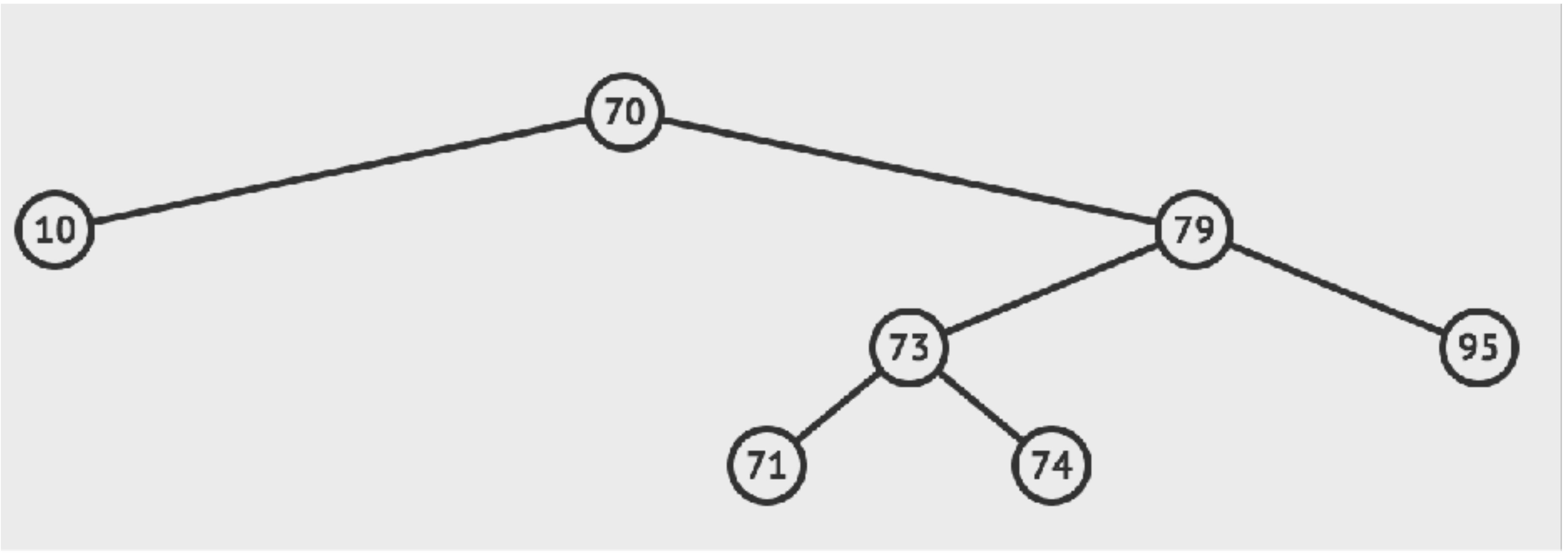
Insertion into a BST

Binary search trees – Hibbard's deletion

- Search for node:
  - Leaf? Just delete it.
  - Node with one child? Delete it and replace with child.
  - Node with two children? Delete and replace with successor (smallest of the larger keys) or predecessor. If successor/predecessor has a child, pass it to parent.

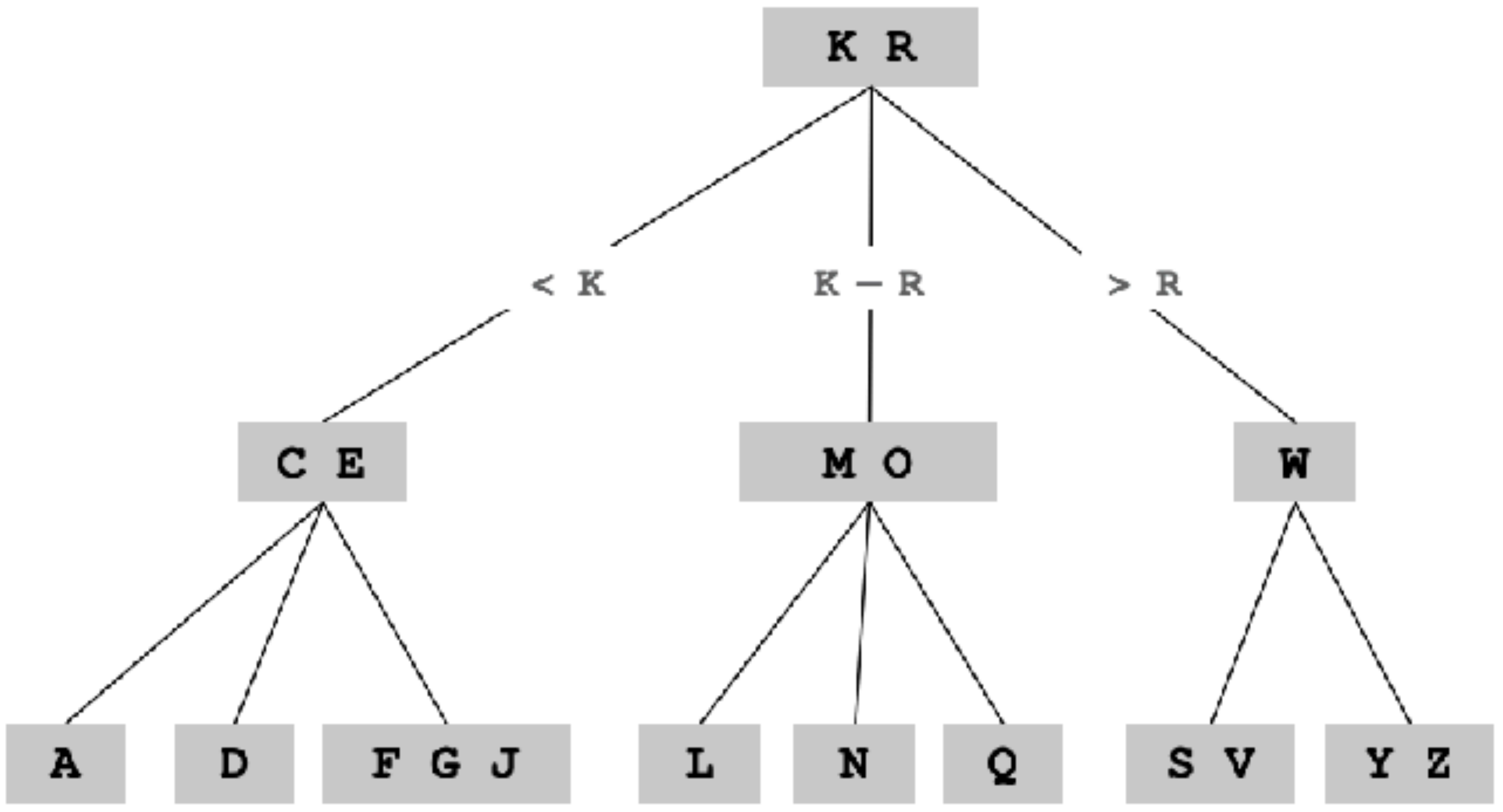
Binary search trees – delete node with key 21





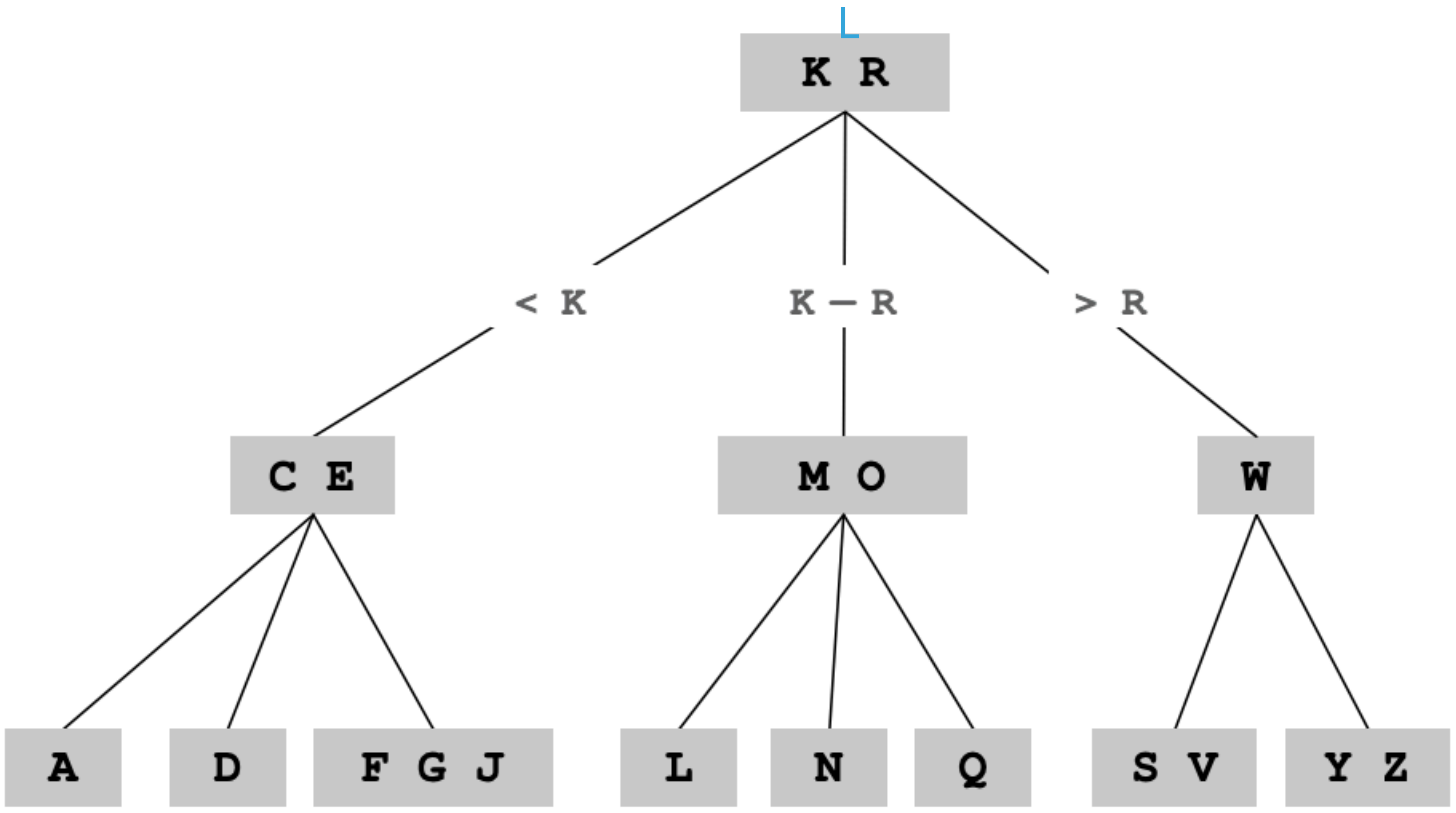
2-3-4 tree

- Definition: A 2-3-4 search tree is either empty or consists of three types of nodes: 2-node, a 3-node, or a 4-node.
  - 2-node: one key, two children
  - 3-node: two keys, three children
  - 4-node: three keys, four children
- Balanced 2-3-4 tree: A 2-3-4 search tree with with all paths from root to a null link has the same length, that is all leaves have the same depth.
  - From now on, 2-3-4 trees are assumed to be balanced.



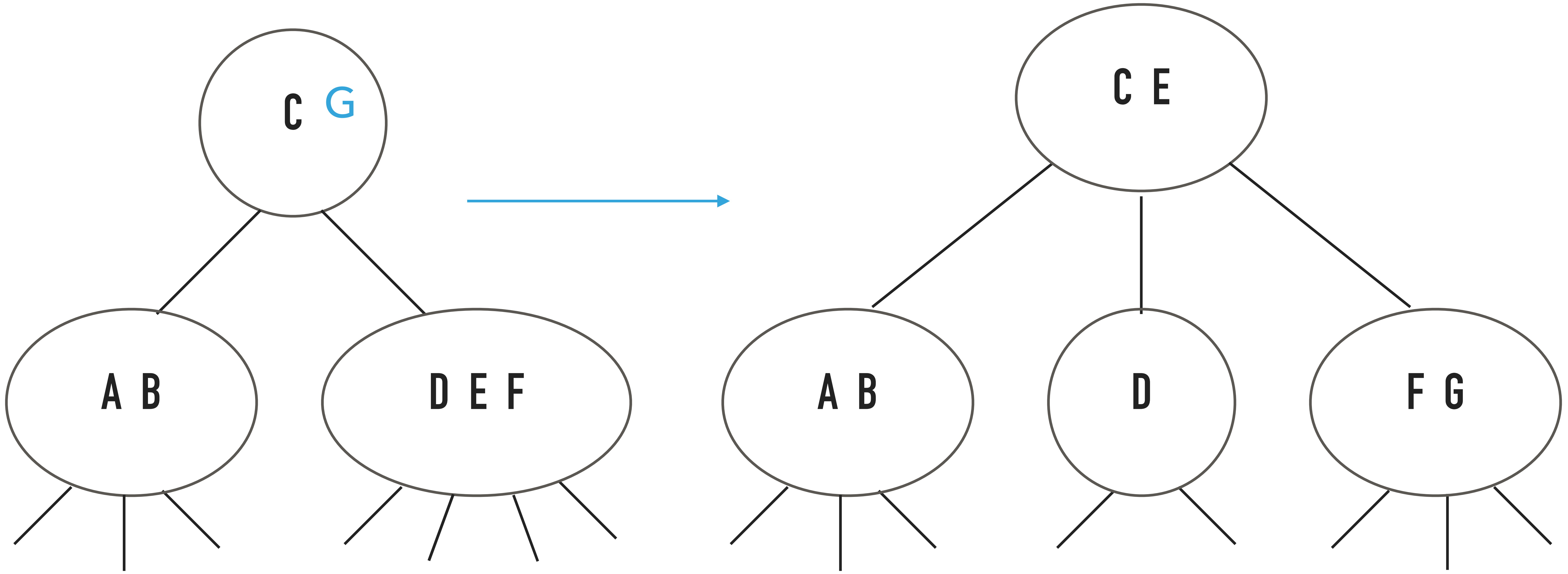
2-3-4 Search Trees - Search

- Compare search key against (every) key in node.
- Find interval containing search key (left, potentially middle, or right).
- Follow associated link, recursively.



2-3-4 Search Trees - Insertion

- Search for key to bottom. Turn 2-nodes to 3-nodes and 3-nodes to 4-nodes.
- 4-nodes are split by moving left middle key to parent.



2-3-4 Search Trees - Performance

- O(log n) search/insertion/deletion but harder to implement because of different types of nodes.

- For practice: https://yongdanielliang.github.io/animation/web/24Tree.html

Summary for dictionary operations

- Worst case search and insert are $O(n)$ for BSTs. Not great!

	Worst case			Average case		
	Search	Insert	Delete	Search	Insert	Delete
BST	n	n	n	$\log n$	$\log n$	$\sqrt{n}$
B-trees	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$

Misc

- Comparable/Comparator Interfaces
- Iterable/Iterator Interfaces
- BT Traversals

Comparable Interface

- Interface with a single method that we need to implement: `public int compareTo(T that)`
- Implement it so that `v.compareTo(w)`:
  - Returns >0 if v is greater than `w`.
  - Returns <0 if v is smaller than `w`.
  - Returns 0 if v is equal to `w`.
- Corresponds to natural ordering.

Comparator Interface

- Sometimes the natural ordering is not the type of ordering we want.
- Comparator is an interface which allows us to dictate what kind of ordering we want by implementing the method:
  `public int compare(T this, T that)`
- Implement it so that `compare(v, w)`:
  - Returns >0 if v is greater than w.
  - Returns <0 if v is smaller than w.
  - Returns 0 if v is equal to w.

```
public static Comparator<ClassName> reverseComparator(){
    return (ClassName a, ClassName b)->{return -a.compareTo(b)};
}
```

Misc

- Comparable/Comparator Interfaces
- Iterable/Iterator Interfaces
- BT Traversals

`Iterable<T>` Interface

- Interface with a single method that we need to implement: `Iterator<T> iterator()`
- Class becomes iterable, that is it can be traversed with a for-each loop.
- ```
  for (String student: students){
          System.out.println(student);
  }
  ```
  ```

`Iterator<T>` Interface

- Interface with two methods that we need to implement: `boolean hasNext()` and `T next()`.
- `hasNext()` checks whether there is any element we have not seen yet.
- `next()` returns the next available element.
- Always check if there are any available elements before returning the next one.
- Typically a comparable class, has an inner class that implements Iterator. Outer class's `iterator` method returns an instance of inner class.
- Can also be implemented in a standalone class where collection to iterate over is passed in the constructor.

Misc

- Comparable/Comparator Interfaces
- Iterable/Iterator Interfaces
- BT Traversals

BT traversals

- Pre-order: mark root visited, left subtree, right subtree.
- In-order: left subtree, mark root visited, right subtree.
- Post-order: left subtree, right subtree, mark root visited.
- Level-order: start at root, mark each node as visited level by level, from left to right.

Practice problems

Practice Problems

- Problem 1 - Sorting
- Problem 2 - Heaps
- Problem 3 - Tree traversals
- Problem 4 - Binary Trees
- Problem 5 - Binary Search Trees
- Problem 6 - Iterators
- Problem 7 - Balanced Binary Search Trees

Problem 1 – Sorting

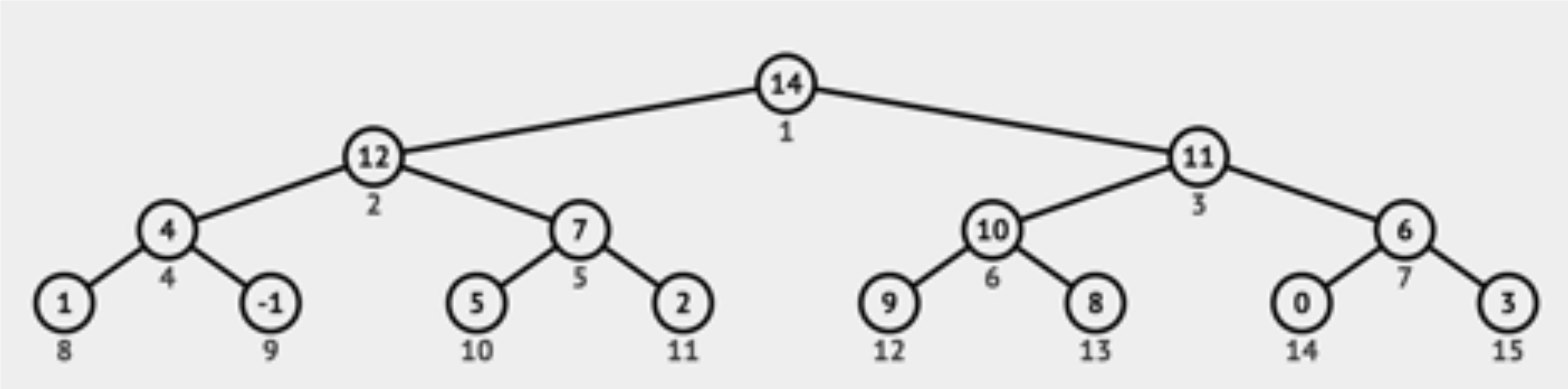
- In the next slide, you can find a table whose first row (last column 0) contains an array of 18 unsorted numbers between 1 and 50. The last row (last column 6) contains the numbers in sorted order. The other rows show the array in some intermediate state during one of these five sorting algorithms:
  - 1-Selection sort
  - 2-Insertion sort
  - 3-Mergesort
  - 4-Quicksort (one partition only)
  - 5-Heapsort
- Match each algorithm with the right row by writing its number (1-5) in the last column.

Problem 1 - Sorting

12	11	35	46	20	43	42	47	44	32	16	10	40	18	41	21	28	15	**0**
11	12	20	35	42	43	46	47	44	32	16	10	40	18	41	21	28	15	
12	11	10	15	20	43	42	47	44	32	16	35	40	18	41	21	28	46	
10	11	12	15	16	43	42	47	44	32	20	35	40	18	41	21	28	46	
43	32	42	28	20	40	41	21	15	11	16	10	35	18	12	44	46	47	
11	12	20	35	46	43	42	47	44	32	16	10	40	18	41	21	28	15	
10	11	12	15	16	18	20	21	28	32	35	40	41	42	43	44	46	47	**6**

Problem 2 – Heaps

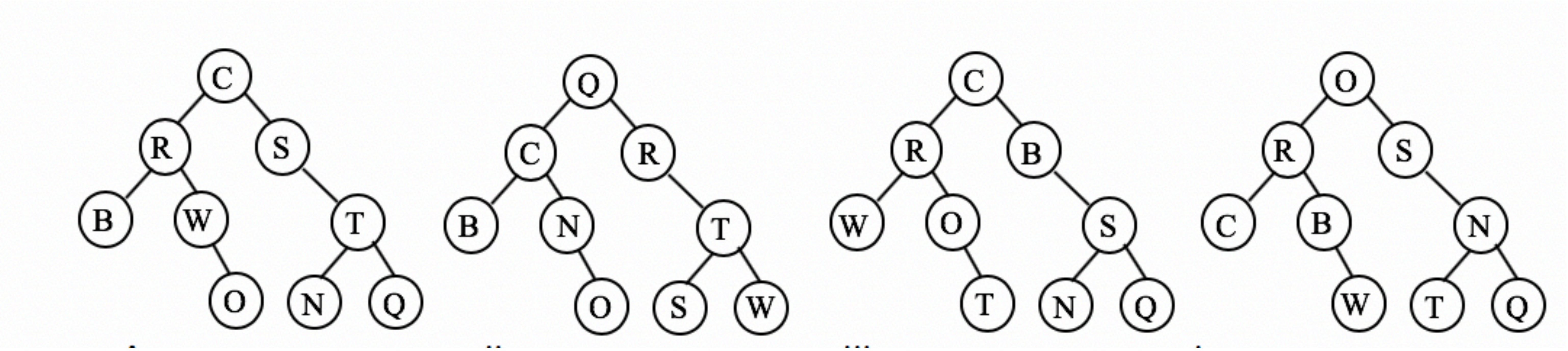
- Consider the following max-heap:



- Draw the heap after you insert key 13.
- Suppose you delete the maximum key from the original heap. Draw the heap after you delete 14.

Problem 3 – Tree Traversals

- Circle the correct binary tree(s) that would produce both of the following traversals:
  - Pre-order: C R B W O S T N Q
  - In-order: B R W O C S N T Q



Problem 4 - Binary Trees

- You are extending the functionality of the `BinaryTree` class that represents binary trees with the goal of counting the number of leaves. Remember that `BinaryTree` has a pointer to a `root Node` and the inner class `Node` has two pointers, `left` and `right` to the root nodes that correspond to its left and right subtrees.
- You are given the following public method:

```
public int sumLeafTree()
  return sumLeafTree(root);
}
```

- Please fill in the body of the following recursive method

```
private int sumLeafTree(Node x){…}
```

Problem 5 - Binary Search Trees

- You are extending the functionality of the `BST` class that represents binary search trees with the goal of counting the number of nodes whose keys fall within a given `[low, high]` range. That is you want to count how many nodes have keys that are equal or larger than `low` and equal or smaller than `high`. Remember that `BST` has a pointer to a `root Node` and the inner class `Node` has two pointers, `left` and `right` to the root nodes that correspond to its left and right subtrees and a `Comparable Key key` (please ignore the value).
- You are given the following public method:

```
public int countRange(Key low, Key high)
  return countRange(root, Key low, Key high);
}
```

- Please fill in the body of the following recursive method

```
private int countRange(Node x, Key low, Key high){…}
```

Problem 6 – Iterators

- A programmer would like to traverse an arraylist in reverse order (from last element to first element). Modify the class ArrayList we wrote together to provide such an iterator.

```
public class ArrayList<E> implements List<E>, Iterable<E> {
     //instance variables data and size
      public Iterator<E> iterator() {

             return new ArrayListIterator();
     }


    private class ArrayListIterator implements Iterator<E> {
     //your implementation

     }
 }
```

Problem 7 - Balanced Binary Search Trees

- Insert the keys 1,2, 3, 4, 5, 6, 7, 8, 9, 10 in a 2-3-4 search tree and draw it after each insertion.

Solutions

Answers

- Solution to Problem 1 - Sorting
- Solution to Problem 2 - Heaps
- Solution to Problem 3 - Tree traversals
- Solution to Problem 4 - Binary Trees
- Solution to Problem 5 - Binary Search Trees
- Solution to Problem 6 - Iterators
- Solution to Problem 7 - Balanced Search Trees

Solution to Problem 1 – Sorting

0-Starting point

1-Selection sort

2-Insertion sort

3-Mergesort

4-Quicksort (one partition only)

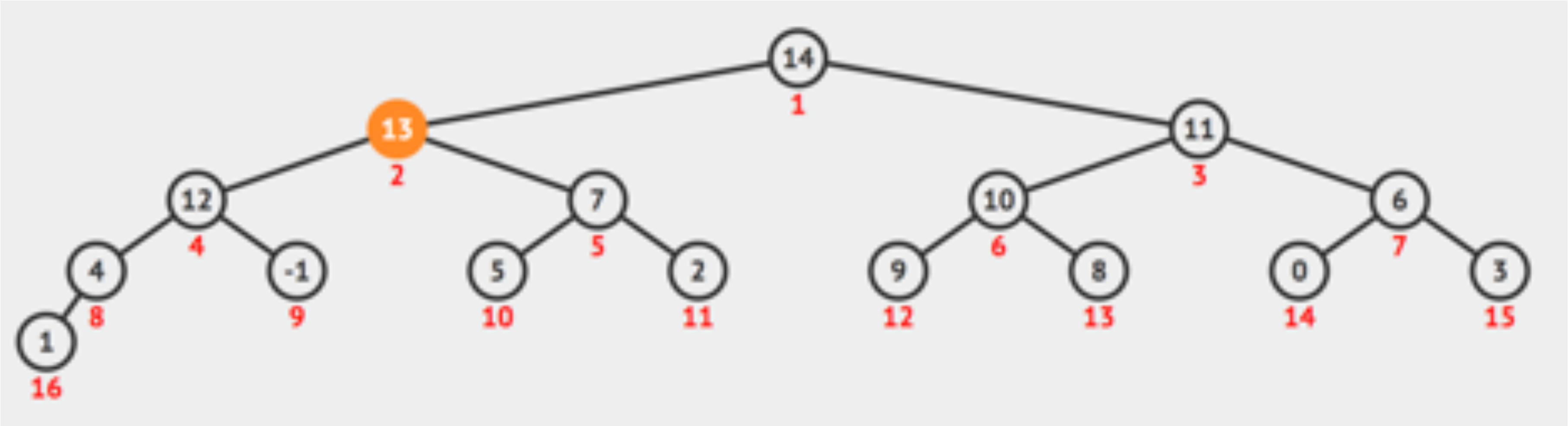
5-Heapsort

6-Final sorted result

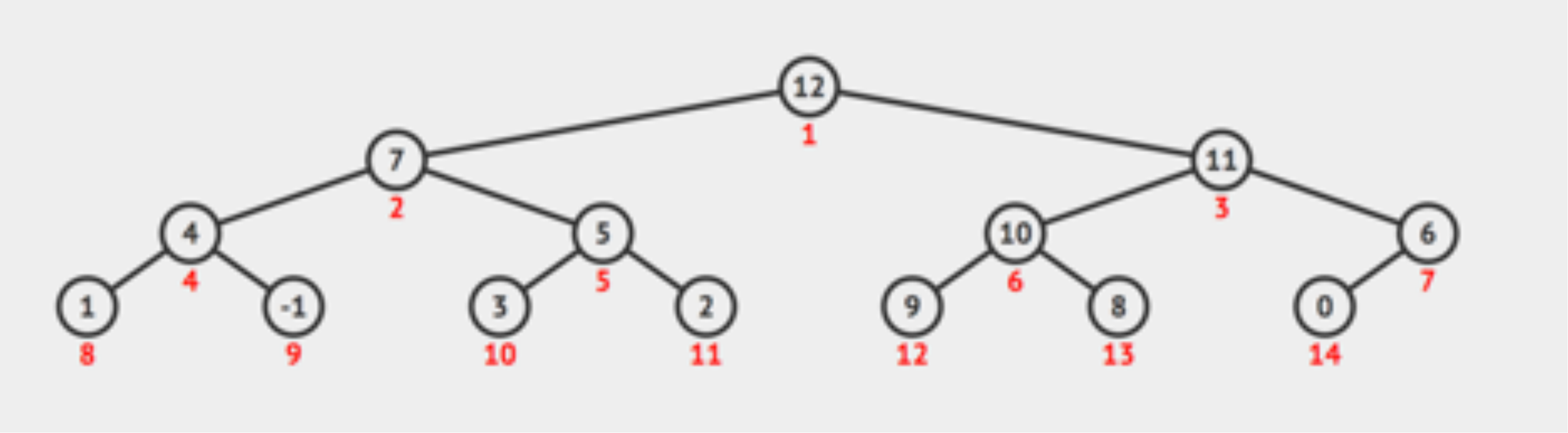
12	11	35	46	20	43	42	47	44	32	16	10	40	18	41	21	28	15	**0**
11	12	20	35	42	43	46	47	44	32	16	10	40	18	41	21	28	15	**2**
12	11	10	15	20	43	42	47	44	32	16	35	40	18	41	21	28	46	**4**
10	11	12	15	16	43	42	47	44	32	20	35	40	18	41	21	28	46	**1**
43	32	42	28	20	40	41	21	15	11	16	10	35	18	12	44	46	47	**5**
11	12	20	35	46	43	42	47	44	32	16	10	40	18	41	21	28	15	**3**
10	11	12	15	16	18	20	21	28	32	35	40	41	42	43	44	46	47	**6**

Solution to Problem 2 - Heaps

- Insert key 13:

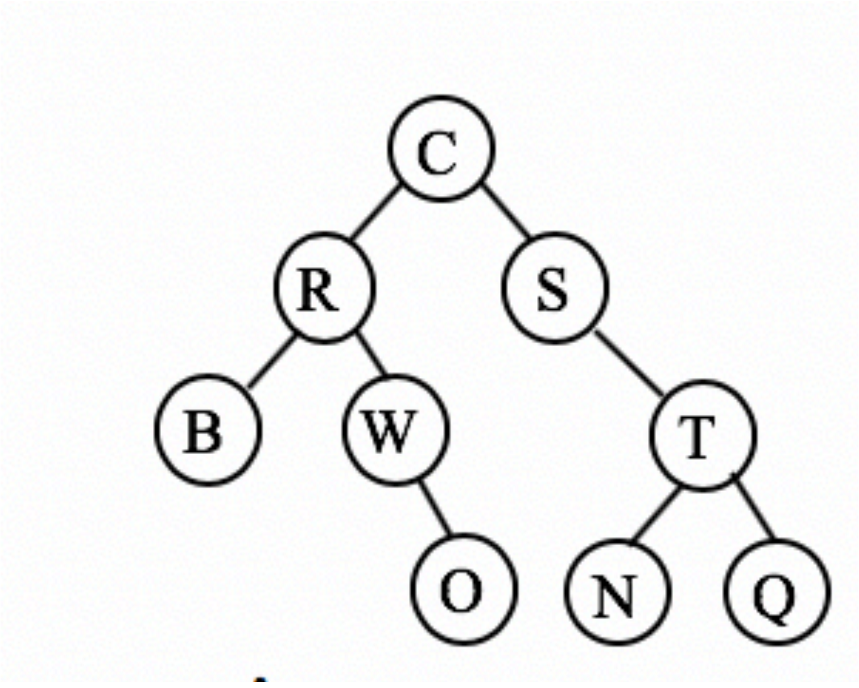


- Delete max-key (14):



Solution to Problem 3 – Tree traversals

- Pre-order: C R B W O S T N Q
- In-order: B R W O C S N T Q



Solution to Problem 4 - Binary Trees

```
private int sumLeafTree(Node x){
  if (x == null){
    return 0;
  }
  else if (x.left == null && x.right == null){
        return 1;
 }
  else{
    return sumLeafTree(x.left) + sumLeafTree(x.right);
  }
}
```

Solution to Problem 5 - Binary Search Trees

```
private int countRange(Node x, Key low, Key high){
  if (x == null){
    return 0;
  }
  if (x.key.compareTo(low)>=0 && x.key.compareTo(high)<=0){
    return 1 + countRange(x.left, low, high) + countRange(x.right, low, high);
  }
  else if (x.key.compareTo(low)<0){
    return countRange(x.right, low, high);
  }
  else{
    return countRange(x.left, low, high);
  }
}
```

Solution to Problem 6 - Iterators

- A programmer would like to traverse an arraylist in reverse order (from last element to first element). Modify the class ArrayList we wrote together to provide such an iterator.

```
public class ArrayList<E> implements List<E>, Iterable<E> {
     //instance variables data and size
       public Iterator<E> iterator() {

               return new ArrayListIterator();
     }


      private class ArrayListIterator implements Iterator<E> {
        private int i = size -1;

        public boolean hasNext() {
            return i >= 0;
        }

        public E next() {
            return data[i--];
        }

        public void remove() {
        }

     }
 }
```

Solution to Problem 7 a - Balanced Search Trees