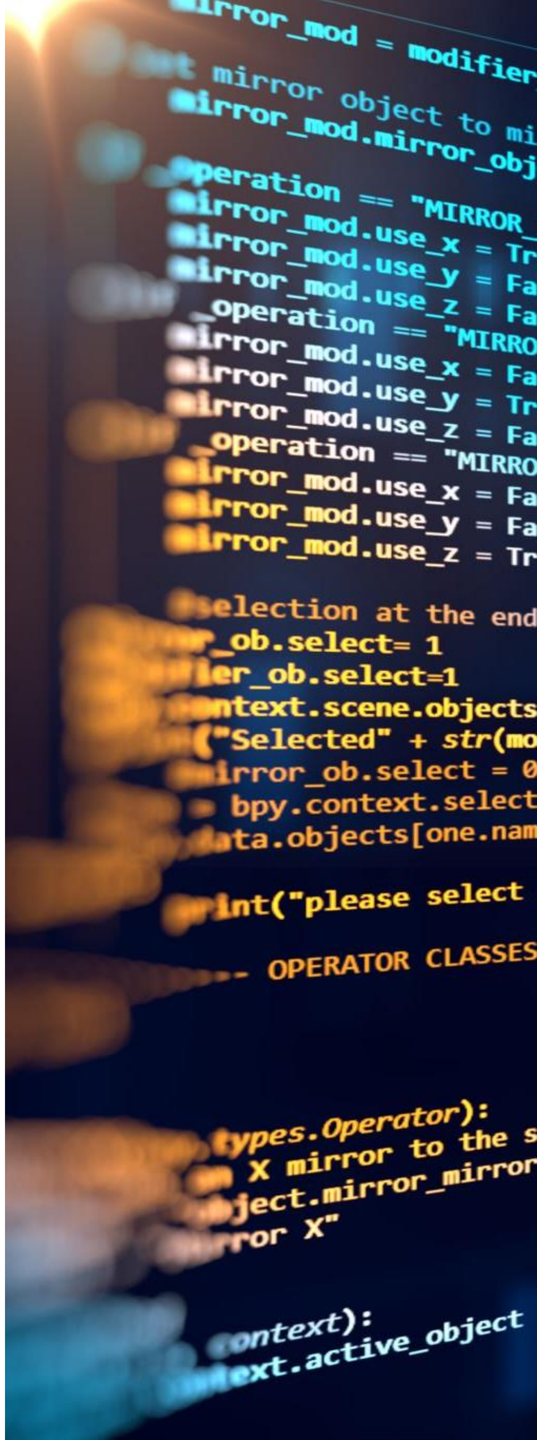
CS 1027
Fundamentals of Computer Science II

Sorting (cont.)

Ahmed Ibrahim



Insertion Sort

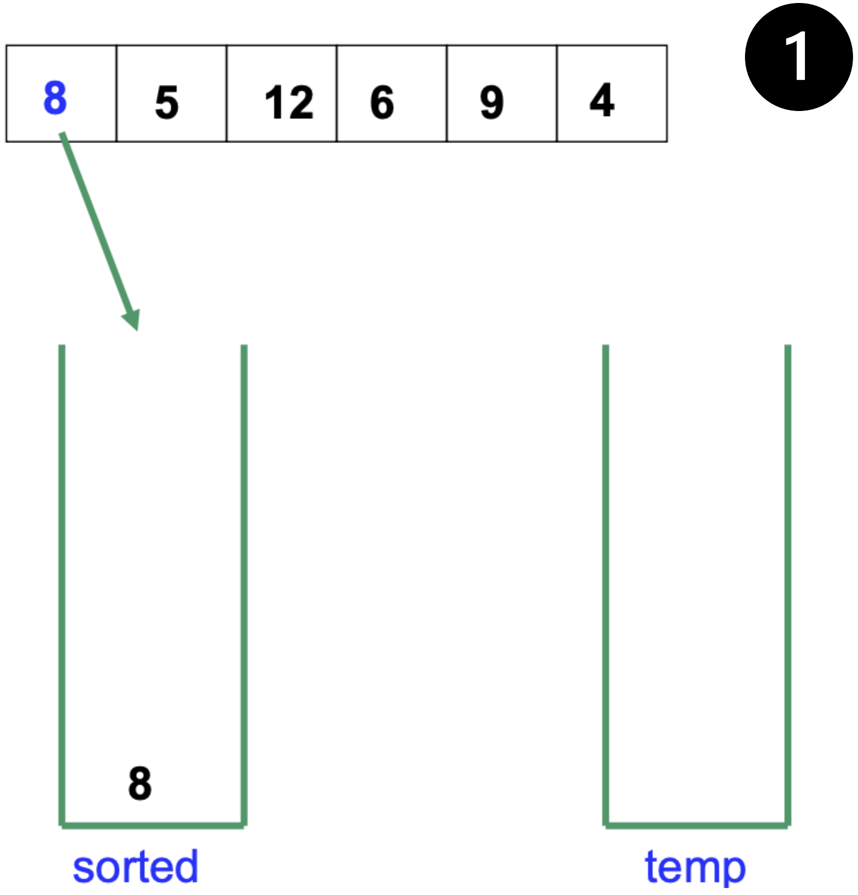
- Insertion Sort orders a sequence of values by repeatedly taking each value and inserting it in its proper position within a sorted subset of the sequence.
- More specifically:

Insertion Sort in Action

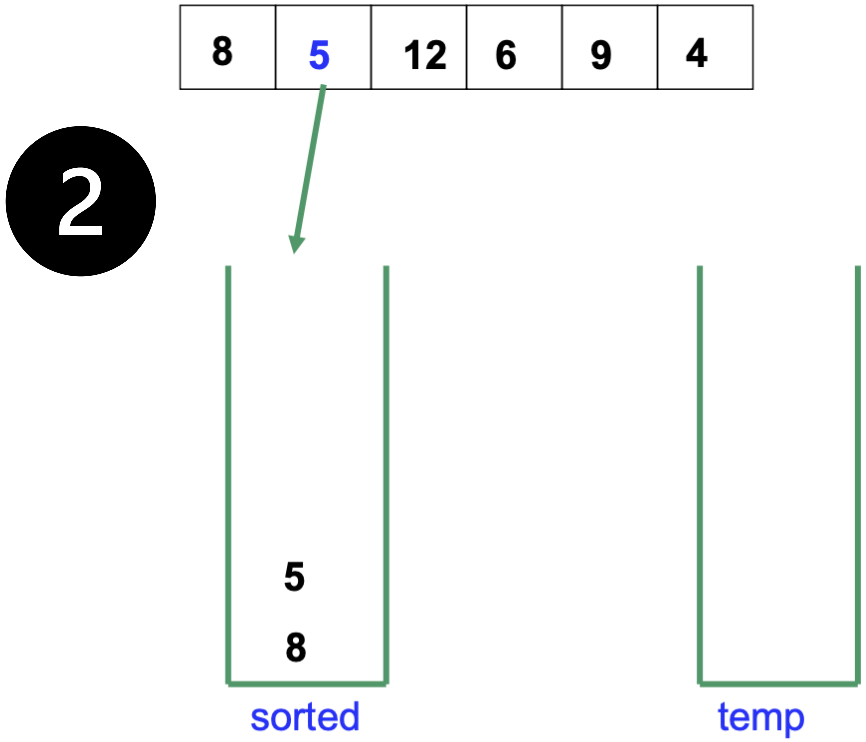
Source:https://s-satsangi.medium.com/insertion-sort-selection-sort-and-bubble-sort-5eb16d55a4de

Insertion Sort Using Stacks

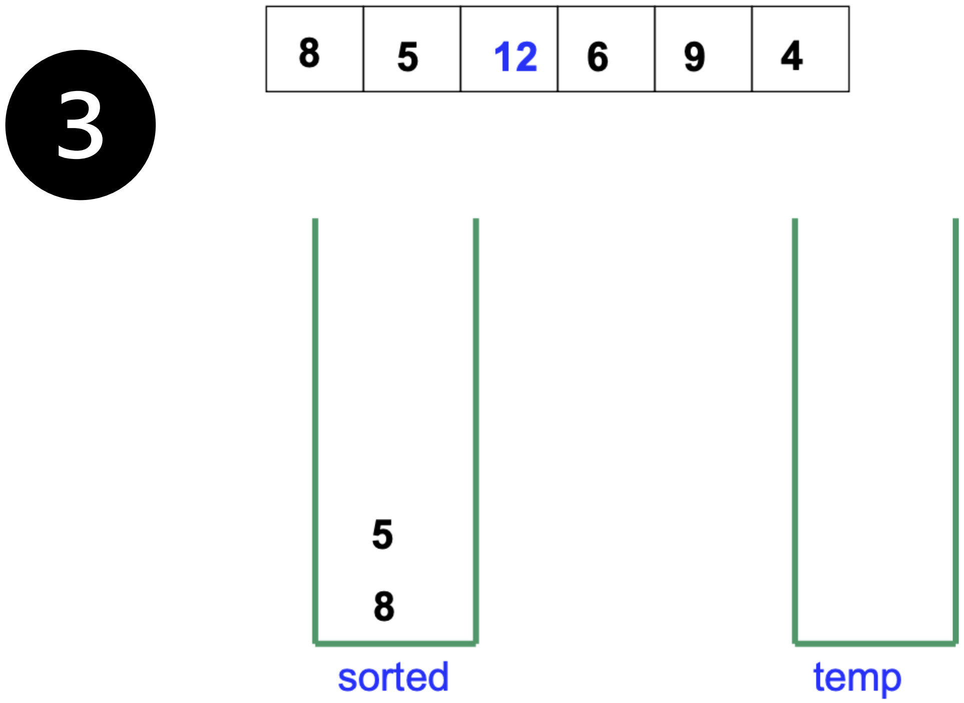
- Use two temporary stacks called sorted and temp, both of which are initially empty.
- The contents of the sorted will always be **in order**, with the smallest item on the top of the stack.
- This will be the "sorted subsequence"
- temp will temporarily hold items that need to be "shifted" out to insert the new item in the proper place in the stack sorted.



Insertion Sort Using Stacks

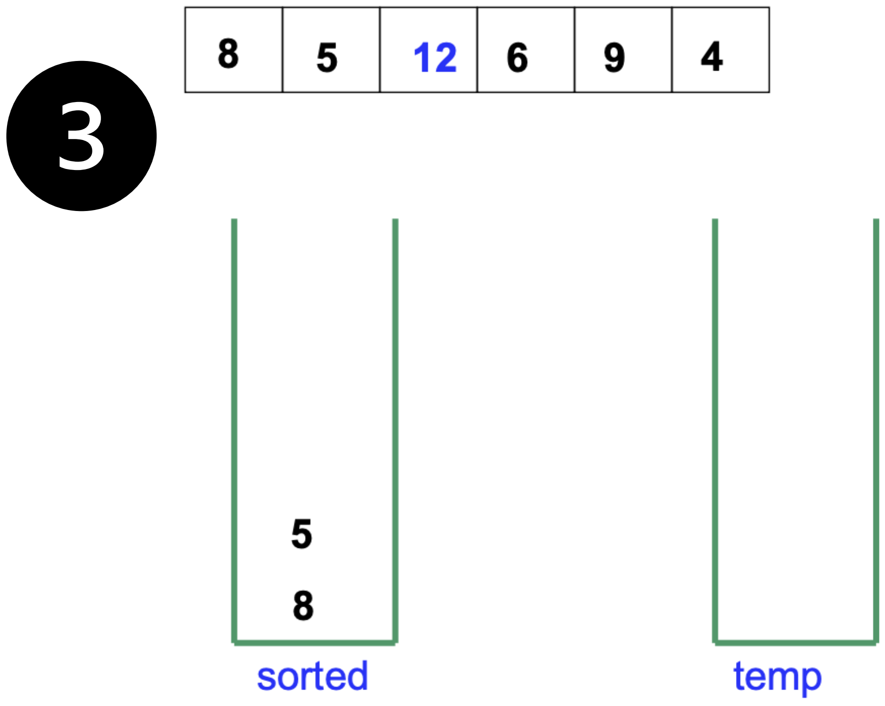


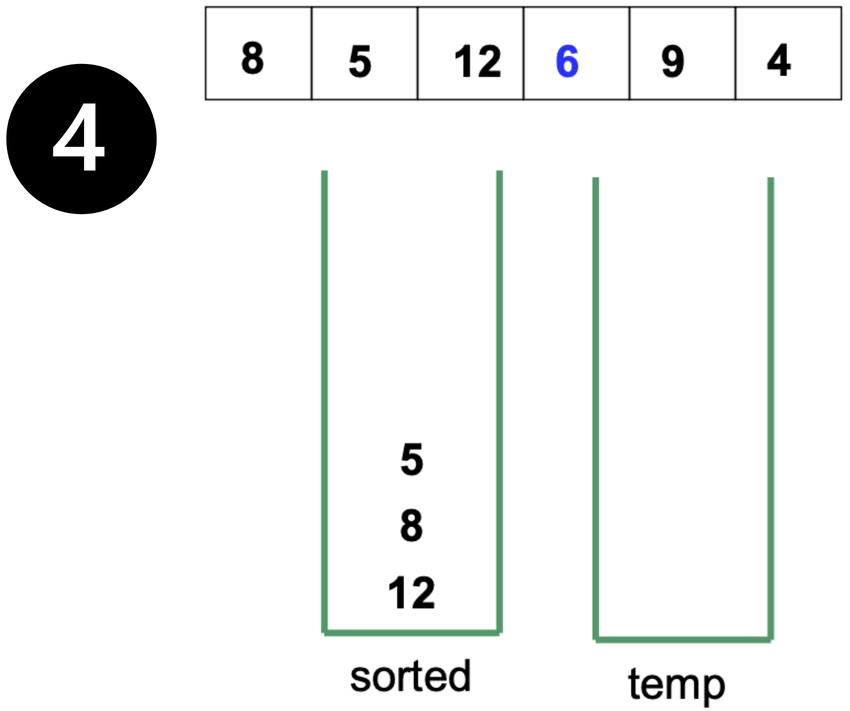
Since 12 > 5, we need to move the values from sorted to temp, push 12 into sorted and move the values back from temp to sorted



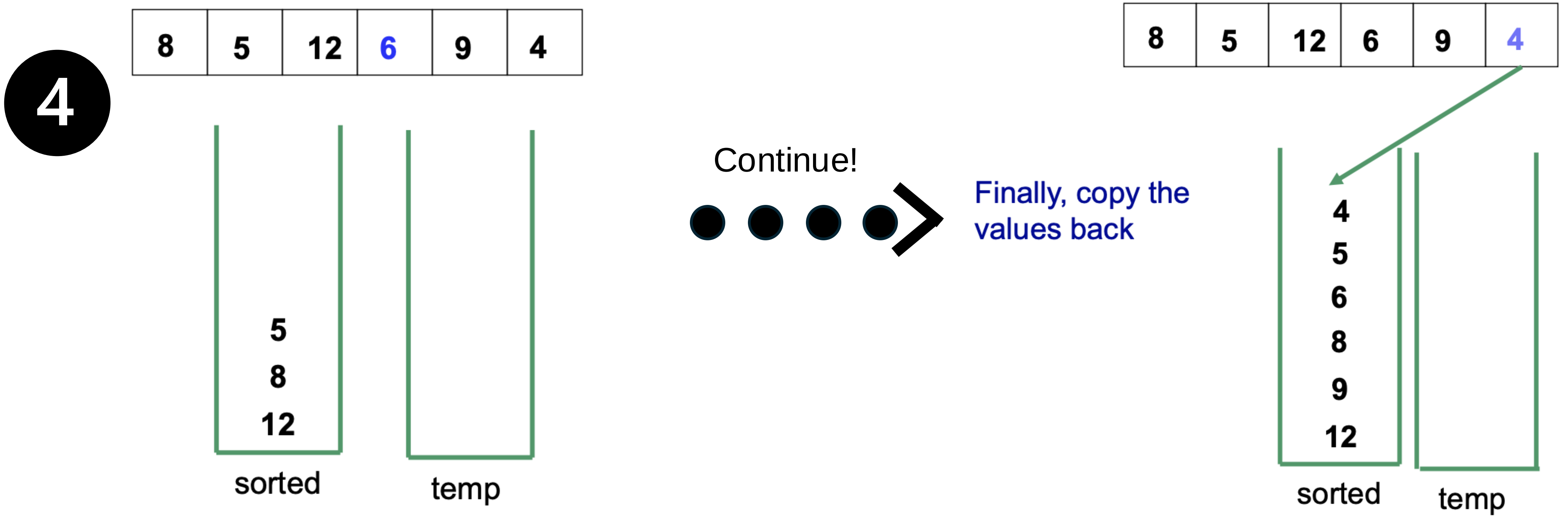
Insertion Sort Using Stacks

Since 6 > 5, we need to move 5 from sorted to temp, push 6 into sorted and move 5 back from temp to sorted





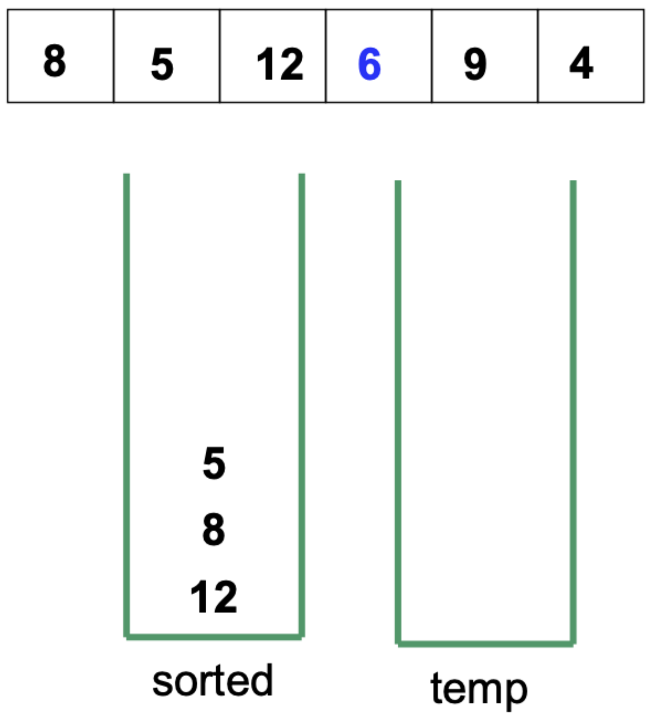
Insertion Sort Using Stacks



Insertion Sort using Stacks

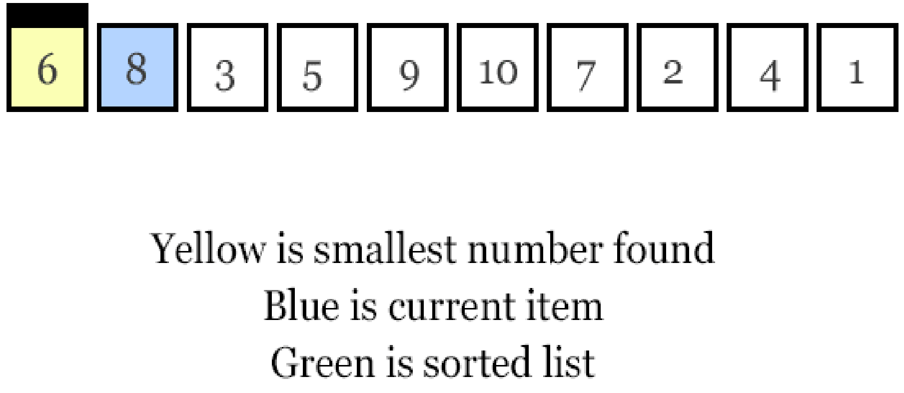
```
Algorithm insertionSort (A,n)
Input: Array A storing n elements
Output: Sorted array

sorted = empty stack
temp = empty stack
for i = 0 to n-1 do {
 while (sorted is not empty) and (sorted.peek() < A[i]) do
 temp.push (sorted.pop())
 sorted.push (A[i])
 while temp is not empty do sorted.push (temp.pop())
}
for i = 0 to n-1 do
   A[i] = sorted.pop()
return A
```



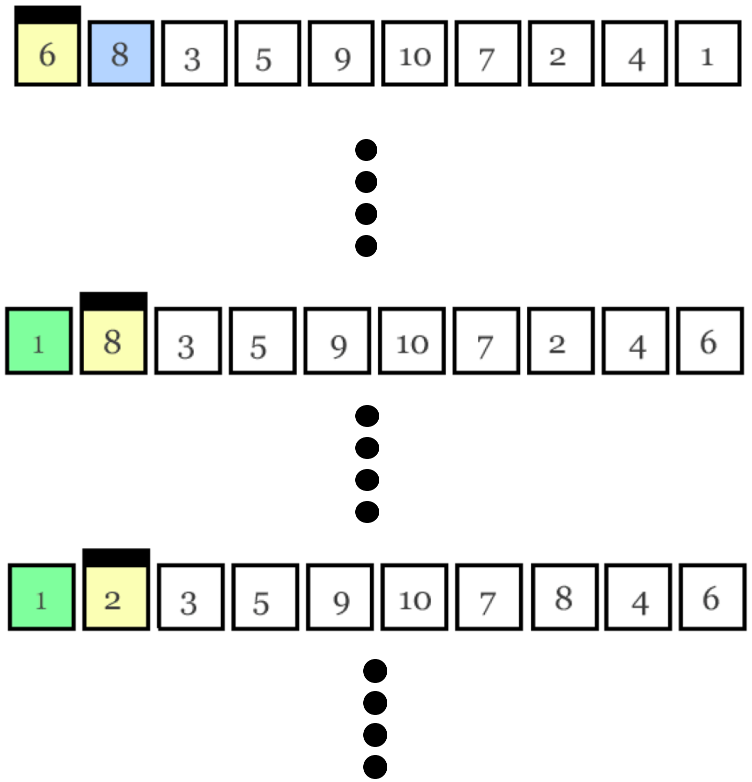
Selection Sort

- This is perhaps the most natural sorting algorithm:
  - Find the smallest value in the sequence
  - Switch it with the value in the first position
  - Find the next smallest value in the sequence
  - Switch it with the value in the second position
  - Repeat until all values are in their proper places



In-Place Selection Sort

```java
public void selectionSort (T[] A, int n) {
 for (int i = 0; i <= n-2; ++i) {
  // Find the smallest value in unsorted subarray A[i..n-1]
  int smallest = i;
  for (int j = i + 1; j <= n - 1; ++j) {
   Comparable<T> tempComp = (Comparable<T>) A[j];
   if (tempComp.compareTo(A[smallest]) < 0) smallest = j;}
   // Swap A[smallest] and A[i]
   T temp = A[smallest];
   A[smallest] = A[i];
   A[i] = temp;}
}
```



Divide-and-Conquer

Divide-and-Conquer is a technique for designing algorithms that consists of three steps:

Divide: Split the input into smaller parts unless the input is so small that the problem can be solved easily (base case)

Conquer: Recursively solve the subproblems associated with the smaller parts of the input

Combine: Combine the solutions of the subproblems to form a solution for the entire problem

Quicksort

- Quicksort orders a sequence of values by using Divide-and-Conquer:
- First, a value in the sequence is selected as the pivot or partition element. Then, the sequence is partitioned into three groups:
    - values greater than the pivot
    - values smaller than the pivot and
    - values equal to the pivot
- Each of the first two partitions is **recursively** sorted.
- The partitions are combined to get the entire sequence sorted.
- The partition element or pivot can be **any value** in the input sequence. In our quicksort implementation, we will select the pivot as the first value in the sequence we are trying to sort.

Steps of Quicksort

1. Put all the items to be sorted into a container (e.g. an array) "Stop & Think"
2. Then, we will arbitrarily choose the pivot (partition element) as the first element from the container
3. Next, we will use a container called *smaller* to hold the items smaller than the pivot, a container called *larger* to hold the items larger than the pivot, and a container called *equal* to hold the items of the same value as the pivot.
4. We then recursively sort the items in the containers smaller and larger.
5. Finally, copy the elements from *smaller* back to the original container, followed by the elements from *equal*, and finally the ones from *larger*

Quicksort

Input array:

6	3	2	6	9	4	8
0	1	2	3	4	5	6

6	3	2	6	9	4	8
0	1	2	3	4	5	6

pivot or partition element

smaller

0	1	2	3

larger

0	1	2	3

equal

0	1	2	3

Quicksort

6	3	2	6	9	4	8
0	1	2	3	4	5	6

Partition the sequence:

smaller

3	2	4	
0	1	2	3

larger

9	8		
0	1	2	3

equal

6	6		
0	1	2	3

Quicksort

Divide step:

6	**3**	**2**	**6**	**9**	**4**	**8**
0	1	2	3	4	5	6

smaller

3	**2**	**4**	
0	1	2	3

Now sort this list

equal

6	**6**		
0	1	2	3

larger

9	**8**		
0	1	2	3

Quicksort

6	3	2	6	9	4	8
0	1	2	3	4	5	6

smaller

2	3	4	
0	1	2	3

Sorted!

equal

6	6		
0	1	2	3

larger

9	8		
0	1	2	3

Quicksort

smaller

2	3	4	
0	1	2	3

equal

6	6		
0	1	2	3

larger

9	8		
0	1	2	3

Next sort this list

smaller

2	3	4	
0	1	2	3

equal

6	6		
0	1	2	3

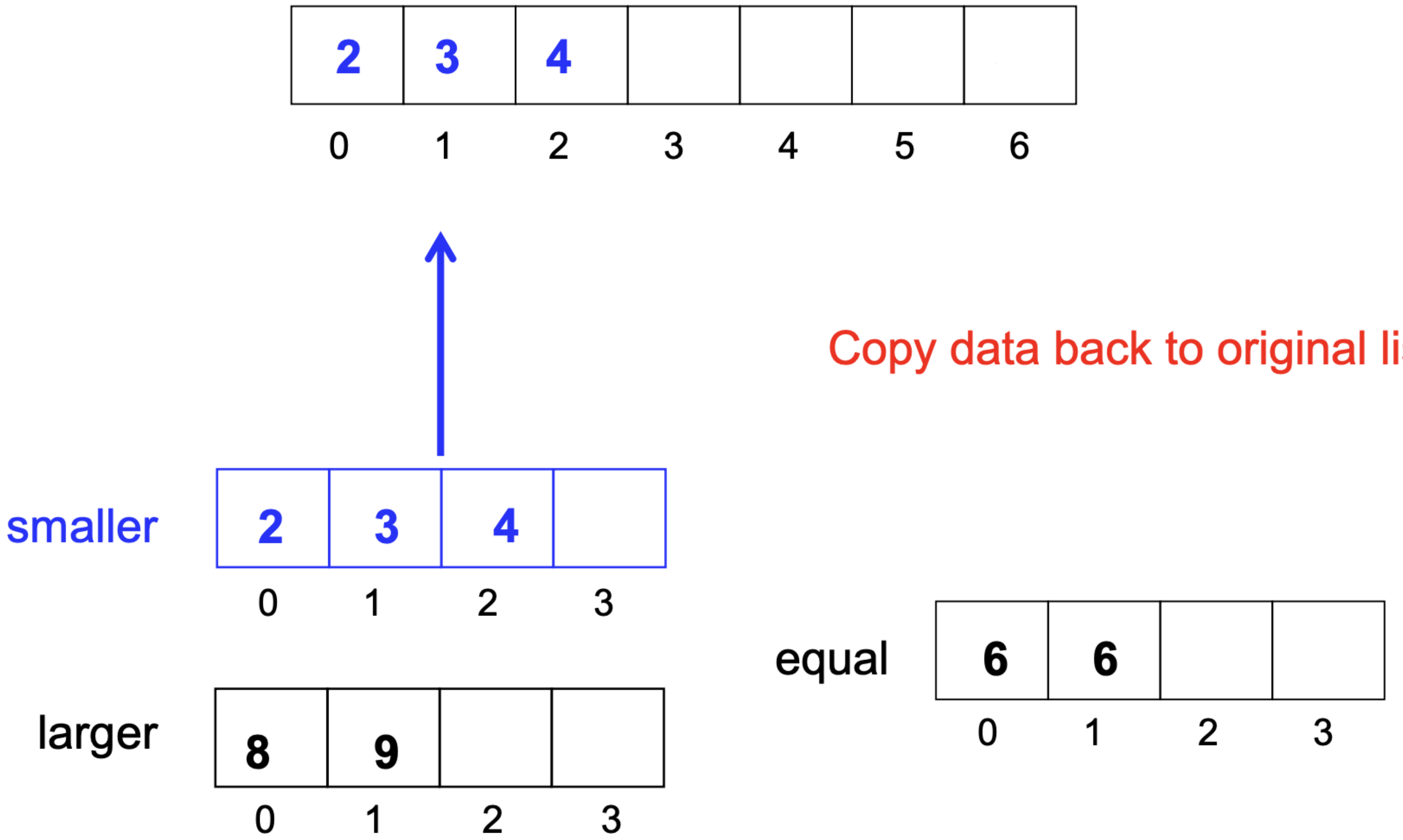
larger

8	9		
0	1	2	3

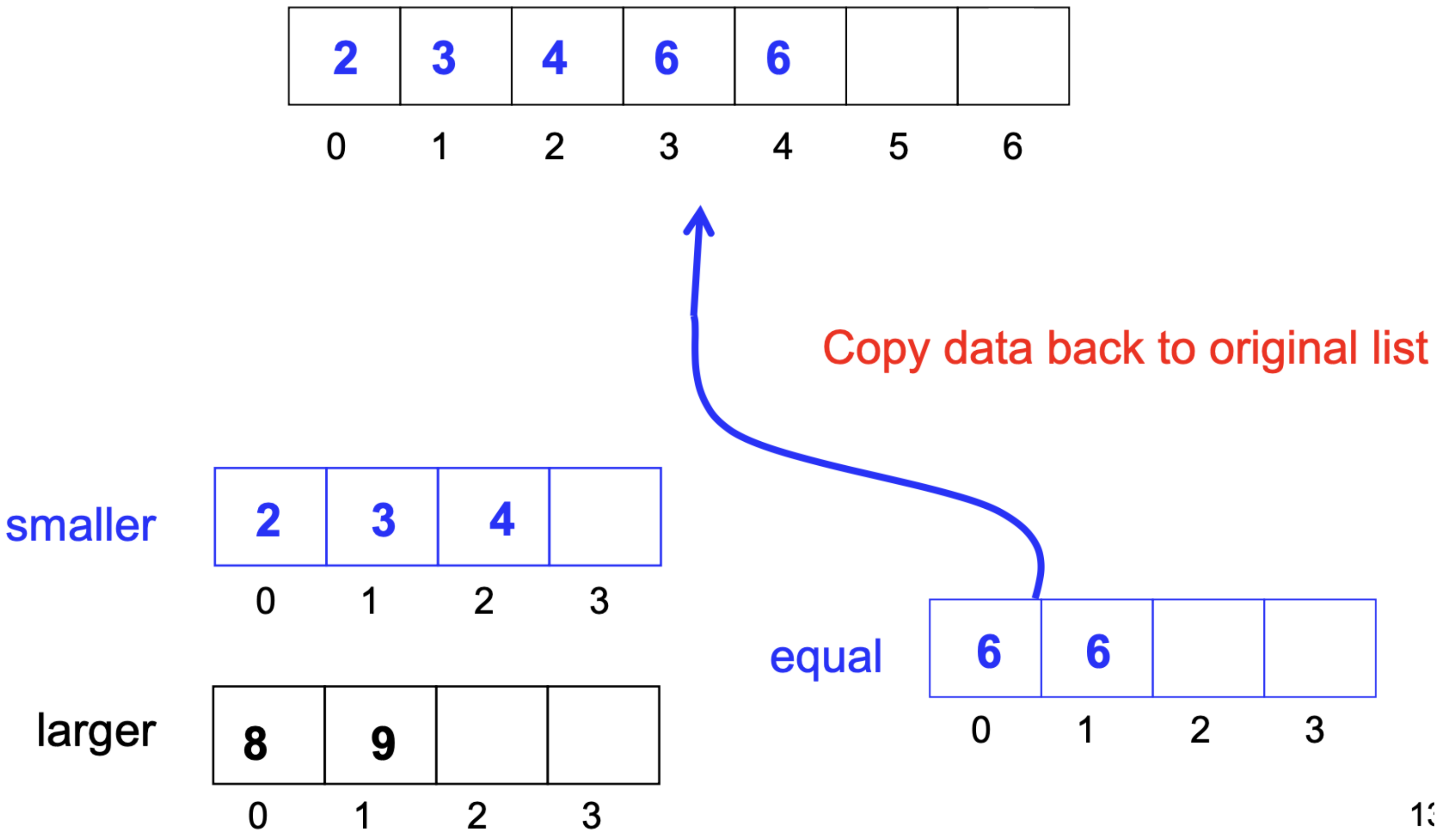
Sorted!

Quicksort

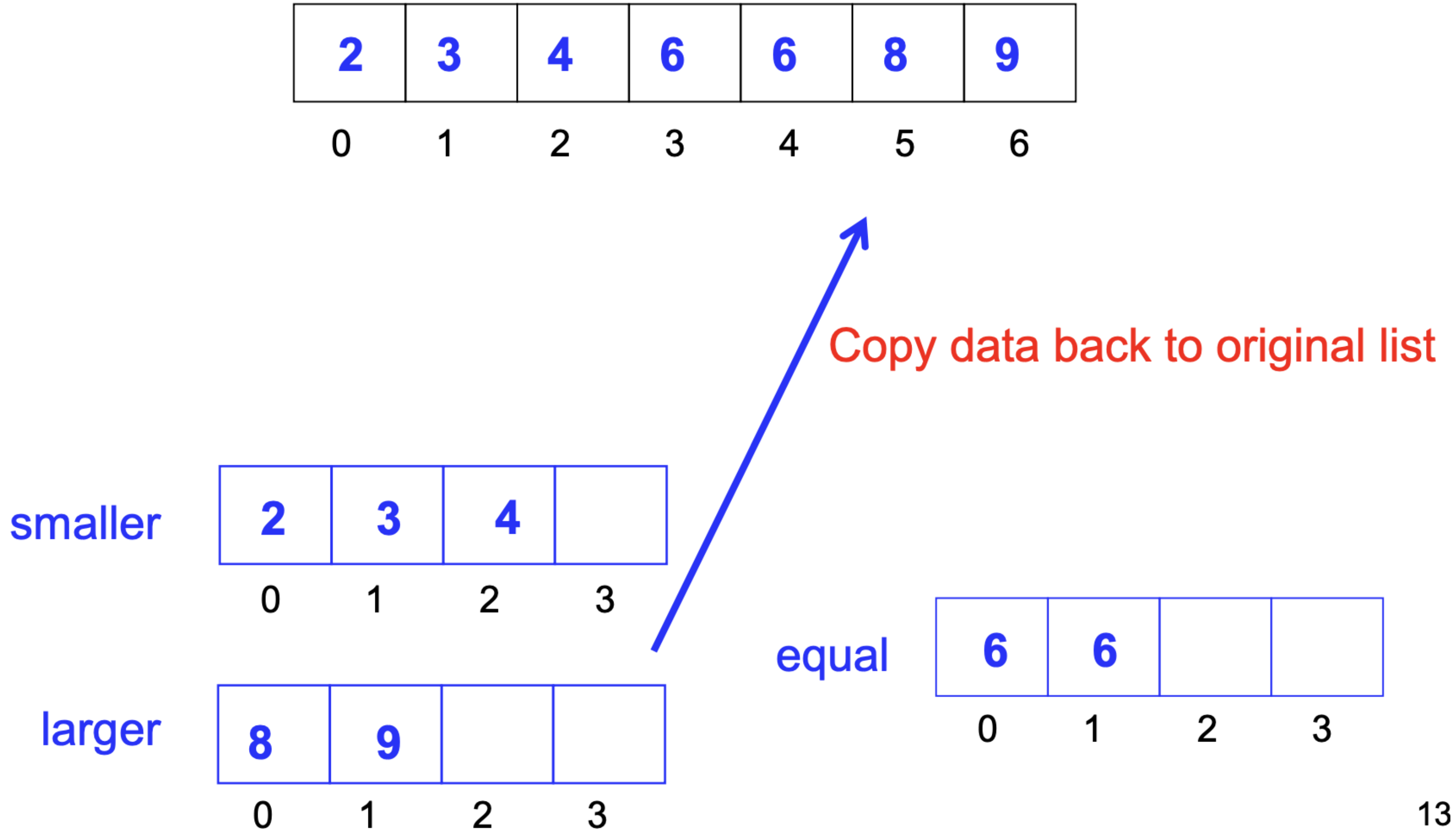
Combine step:



Quicksort


2 3 4 6 6
0 1 2 3 4 5 6
Copy data back to original list
smaller
2 3 4
0 1 2 3
equal
6 6
0 1 2 3
larger
8 9
0 1 2 3

Quicksort


2 3 4 6 6 8 9
0 1 2 3 4 5 6
Copy data back to original list
smaller
2 3 4
0 1 2 3
larger
8 9
0 1 2 3
equal
6 6
0 1 2 3
13

Quicksort

6	3	2	6	9	4	8
0	1	2	3	4	5	6

smaller

3	2	4	
0	1	2	3

How did we sort this list?

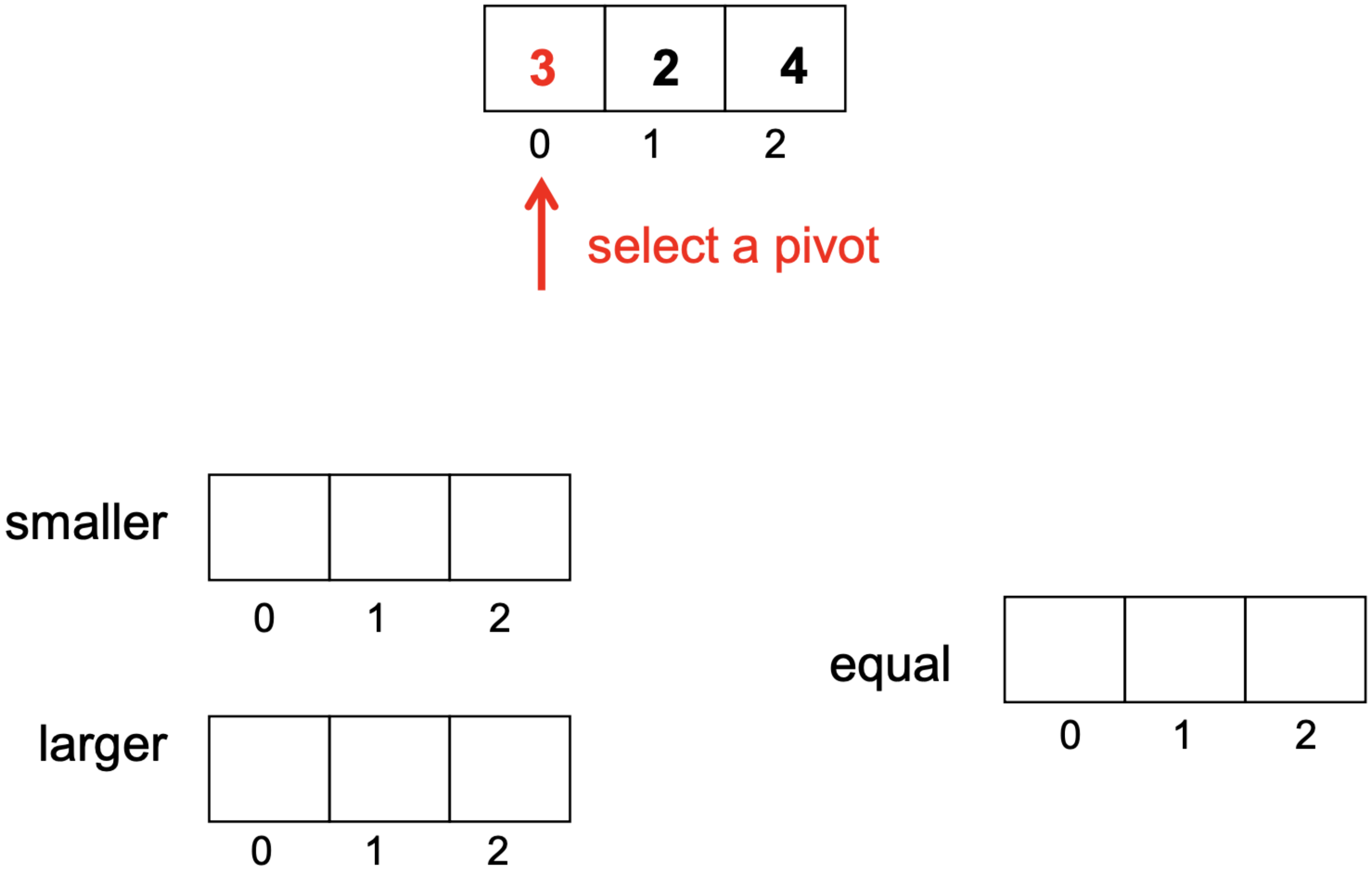
equal

6	6		
0	1	2	3

larger

9	8		
0	1	2	3

Quicksort



Quicksort

Algorithm quicksort(A,n)

In: Array A storing n values

Out: Nothing, but sort A in increasing order

```
public void quicksort(T[] A, int n) {
if (n > 1) {
    T[] smaller = (T[])(new Object[n]);
    T[] equal = (T[])(new Object[n]);
    T[] larger = (T[])(new Object[n]);
int ns, ne, nl;
ns = ne = nl = 0;
T pivot = A[0];
Comparable<T> tmp = (Comparable<T>) pivot;
```

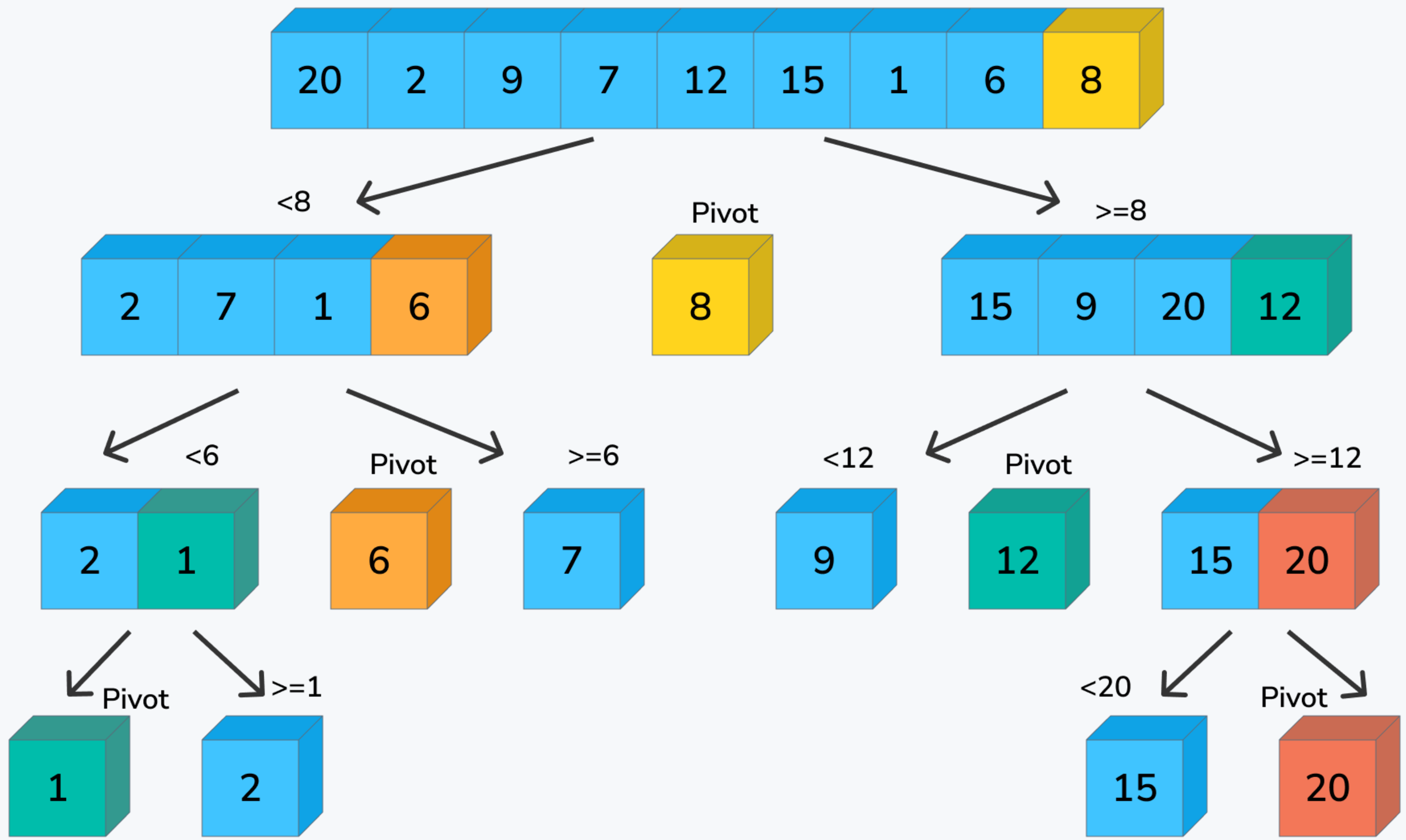
```
// Partition the values
for (int i = 0; i <= n-1; ++i)
  if (A[i].equals(pivot)) equal[ne++] = A[i];
  else if (tmp.compareTo(A[i]) > 0)
         smaller[ns++] = A[i];
      else larger[nl++] = A[i];

quicksort(smaller,ns);
quicksort(larger,nl);
int i = 0;
  for (int j = 0; j < ns; ++j) A[i++] = smaller[j];
  for (int j = 0; j < ne; ++j) A[i++] = equal[j];
  for (int j = 0; j < nl; ++j) A[i++] = larger[j];
  }
}
```

Dividing

Sorting

Combining

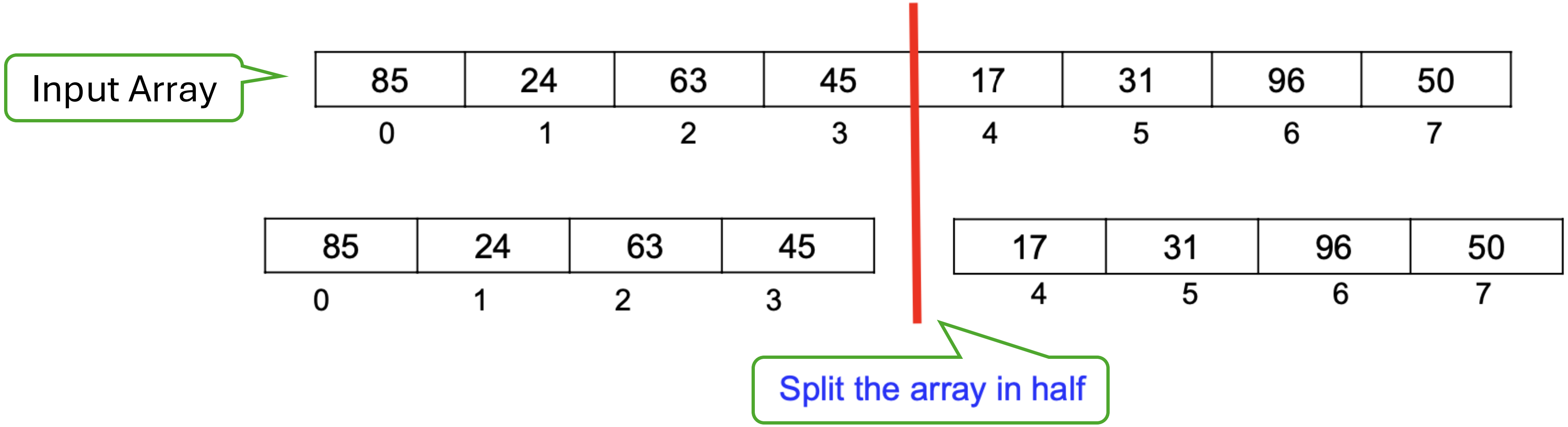


Recursion Tree for Partitions

Source: https://www.geeksforgeeks.org/implement-quicksort-with-first-element-as-pivot/

Mergesort

- Mergesort also orders a sequence of values by using Divide-and-Conquer:
  - First, the input sequence is divided in half
  - Then, each half is **recursively** sorted
  - Finally, the sorted sub-sequences are combined to get the entire sequence sorted.



Mergesort

85	24	63	45	17	31	96	50
0	1	2	3	4	5	6	7

85	24	63	45
0	1	2	3

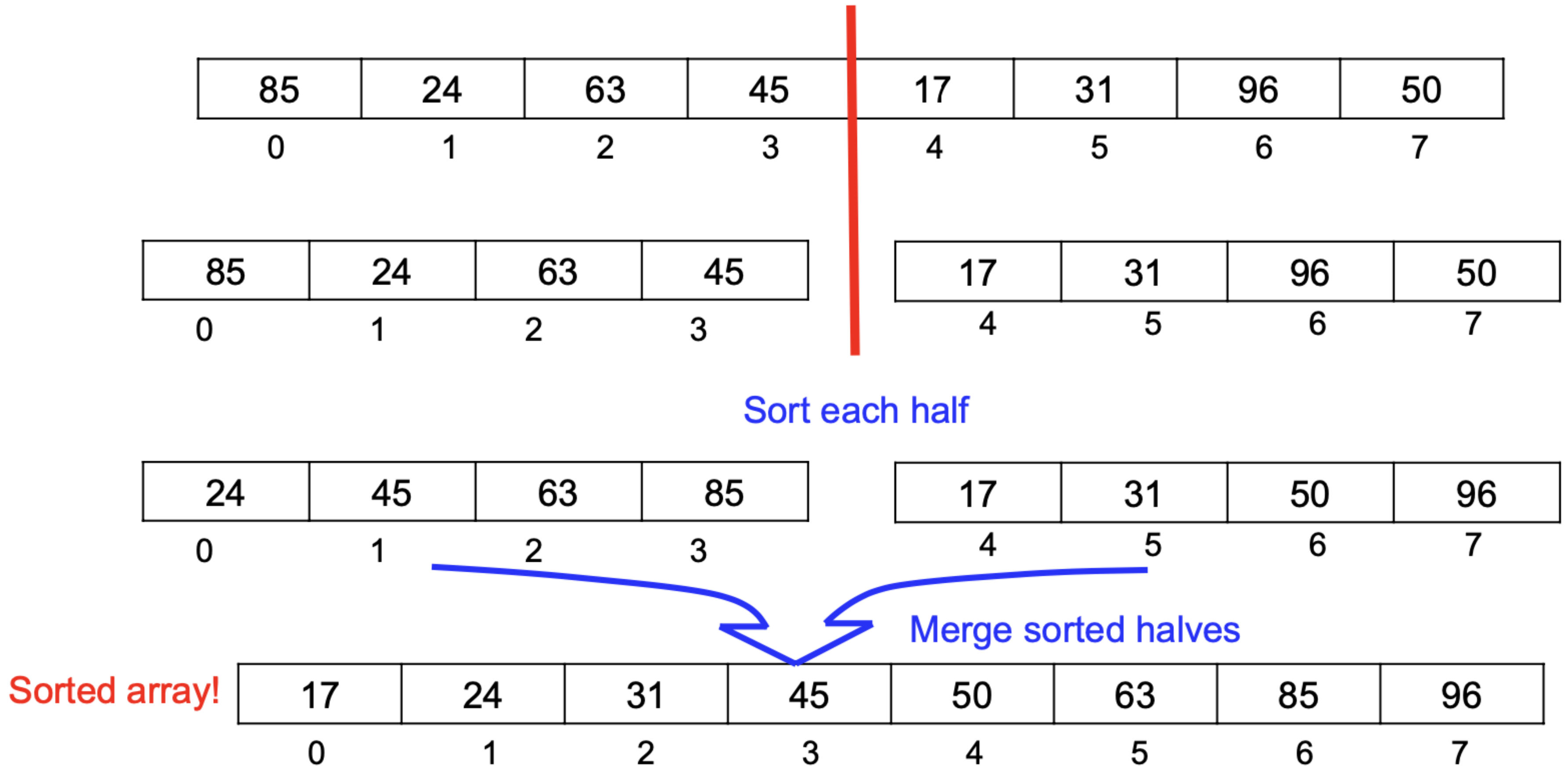
17	31	96	50
4	5	6	7

Sort each half

24	45	63	85
0	1	2	3

17	31	50	96
4	5	6	7

Mergesort


85 24 63 45 17 31 96 50
0 1 2 3 4 5 6 7
85 24 63 45
0 1 2 3
17 31 96 50
4 5 6 7
Sort each half
24 45 63 85
0 1 2 3
17 31 50 96
4 5 6 7
Merge sorted halves
Sorted array!
17 24 31 45 50 63 85 96
0 1 2 3 4 5 6 7

Mergesort

85	24	63	45	17	31	96	50
0	1	2	3	4	5	6	7

85	24	63	45
0	1	2	3

How to sort each half?

85	24	63	45
0	1	2	3

Split in half

85	24
0	1

63	45
2	3

sort each half $\Rightarrow$

24	85
0	1

45	63
2	3

Merge sorted halves

24	45	63	85
0	1	2	3

Mergesort

We will assume that the input is stored in an array.

85	24	63	45	17	31	96	50
0	1	2	3	4	5	6	7

85	24	63	45
0	1	2	3

85	24
0	1

split in half

85
0

24
1

Each half is already sorted!
(this is the base case)

Mergesort

```
public void mergesort (T[] A, int
first, int last) {
   if (first < last) {
   int mid = (first + last) / 2;
        mergesort(A,first,mid);
        mergesort(A,mid+1,last);
        merge(A,first,mid,last);
  }
}
```

```
public void merge (T[] A, int first, int mid, int last) {
T[] tmp = (T[]) (new Object[last-first+1]);
int i = first;
int j = mid + 1;
int k = 0;

while ((i <= mid) && (j <= last)) {
if (((Comparable<T>)A[i]).compareTo(A[j]) < 0 ) tmp[k] = A[i++];
else tmp[k] = A[j++]; ++k;}

while (i <= mid) tmp[k++] = A[i++];
while (j <= last) tmp[k++] = A[j++];
while (k-- > 0) arr[k+first] = tmp[k];
}
```

Review & Practice Exercises

Question!

- Consider the following queue, Q, and code corresponding to Queue Q.

$$\mathtt{Q} = \underline{\overline{\;4 \quad -1 \quad 7 \quad 5 \quad -3\;}} \longleftarrow \text{rear}$$

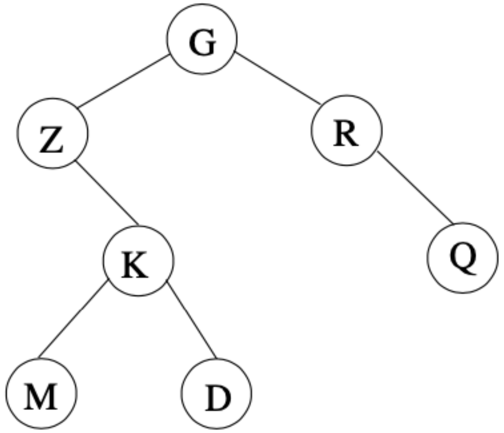
```
while (!Q.isEmpty())
if (Q.first() > 0) System.out.print(Q.dequeue() + " ");
```

Determine which of the following statements is TRUE:

a) This code would produce a compile-time error
b) This code would produce a run-time error
c) This code would result in an infinite loop
d) This code would result in the following output: 4 -1 7 5 -3
e) This code would result in the following output: 4 7 5

Question!

- Consider the following binary tree for the next three questions.



(1 mark) Determine the **preorder** traversal of the tree shown above.

- ○ Z, M, K, D, G, R, Q
- ○ G, Z, M, D, K, Q, R
- ○ G, Z, K, M, D, R, Q
- ○ G, Z, K, D, M, R, Q
- ○ G, Z, R, K, Q, M, D
- ○ M, D, K, Z, Q, R, G

Question!

(4 marks) Consider the following code.

```
public void sort(int[] a, int n) {
    int j = 0;
    while (j < n - 1) {
        int index = j;
        (**)
        int y = a[index];
        a[index] = a[j];
        a[j] = y;
        ++j;
    }
}
```

temp

Outer-loop

Which code must be inserted at the point marked (**) to sort the array in decreasing order? Array a stores n different integer values.

- ◯ `for (int i = j+1; i < n; i = i + 1) if (a[i] < a[index]) index = a[i];`
- ◯ `for (int i = j+1; i < n; i = i + 1) if (a[i] > a[index]) index = a[index];`
- ◯ `for (int i = j+1; i < n; i = i + 1) if (a[i] < a[index]) index = i;`
- ◯ `for (int i = j+1; i < n; i = i + 1) if (a[i] > a[index]) index = i;`
- ◯ `for (int i = 0; i < n; i = i + 1) if (a[i] < a[index]) index = i;`
- ◯ `for (int i = 0; i < n; i = i + 1) if (a[i] > a[index]) index = i;`

Question!

(2 marks) Consider an empty stack **s** and a queue **q** storing the following values: 3, 7, 5, and 6. The queue is implemented using a circular array **arrQueue** where public instance variable **front** is the index of the first value and **rear** is the index of the last value in the queue, as shown in the following figure.

rear = 1 front = 5

arrQueue

0	1	2	3	4	5	6
5	6				3	7

Consider the following code fragment

```
for (int i = 0; i < 4; i = i+1) s.push(i);
for (int i = 1; i <= 3; i = i + 1) {
    q.enqueue(s.pop());
    if (((q.rear + 1) % 7) != q.front) q.enqueue(s.pop());
    s.push(q.dequeue());
}
```

s
3
2
1
0

What are the values of **front** and **rear** after the above code fragment is executed?

- ◯ `front = 0, rear = 6`
- ✔ `front = 1, rear = 6`
- ◯ `front = 1, rear = 5`
- ◯ `front = 6 rear = 5`
- ◯ None. The code will throw an exception

Question! (cont.)

Iteration 1 (i = 1)

1. First Statement: `q.enqueue(s.pop());`
    1. Stack s -> (pop 3).
    2. arrQueue = [5, 6, 3, , , 3, 7]
    3. front = 5, rear = 2
2. Condition: `if (((q.rear + 1) % 7) != q.front)`
    1. (2 + 1) % 7 = 3, and q.front = 5.
    2. Condition is true.
3. Second Statement in if: `q.enqueue(s.pop());`
    1. Stack s -> (pop 2).
    2. arrQueue = [5, 6, 3, 2, , 3, 7] front = 5, rear = 3
4. Last Statement: `s.push(q.dequeue());`
    1. dequeue q.front -> 3
    2. s: [0, 1, 3]
    3. arrQueue = [5, 6, 3, 2, , , 7] front = 6, rear = 3

Iteration 1 (i = 2)

1. First Statement: `q.enqueue(s.pop());`
    1. Stack s -> (pop 3).
    2. arrQueue = [5, 6, 3, 2, 3, , 7]
    3. front = 6, rear = 4
2. Condition: `if (((q.rear + 1) % 7) != q.front)`
    1. (4 + 1) % 7 = 5, and q.front = 6
    2. Condition is true.
3. Second Statement in if: `q.enqueue(s.pop());`
    1. Stack s -> (pop 1).
    2. arrQueue = [5, 6, 3, 2, 3, 1, 7] front = 6, rear = 5
4. Last Statement: `s.push(q.dequeue());`
    1. dequeue q.front -> 7
    2. s: [0, 7].
    3. arrQueue = [5, 6, 3, 2, 3, 1,] front = 0, rear = 5

Question! (cont.)

Iteration 1 (i = 3)

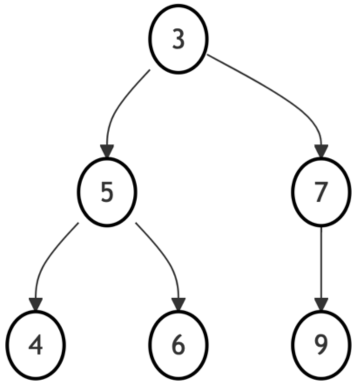
1. First Statement: `q.enqueue(s.pop());`
    1. Stack s -> (pop 7).
    2. arrQueue = [5, 6, 3, 2, 3, 1, 7]
    3. front = 0, rear = 6
2. Condition: `if (((q.rear + 1) % 7) != q.front)`
    1. (6 + 1) % 7 = 0, and q.front = 0
    2. Condition is **false**.
    3. No action is taken for the second `q.enqueue(s.pop());`
3. Last Statement: `s.push(q.dequeue());`
    1. dequeue q.front -> value 5
    2. s: [0, 5].
    3. arrQueue = [, 6, 3, 2, 3, 1, 7] front = 1, rear = 6

Question!

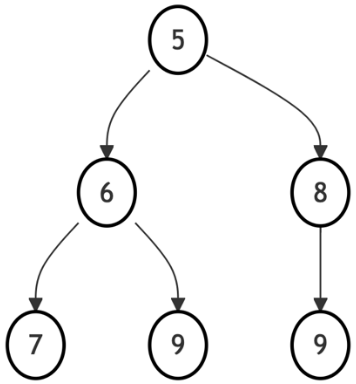
(3 marks) Which of the following arrays represents a min heap?

A1	3	5	7	4	6	9
A2	5	6	8	7	9	9
A3	4	7	6	8	9	5

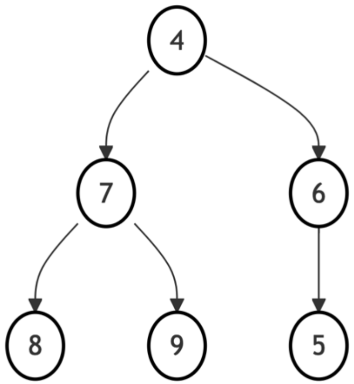
- ○ A1
- ☑ A2
- ○ A3
- ○ A1 and A2
- ○ A1 and A3
- ○ A2 and A3
- ○ None
- ○ All of them



left child



left child



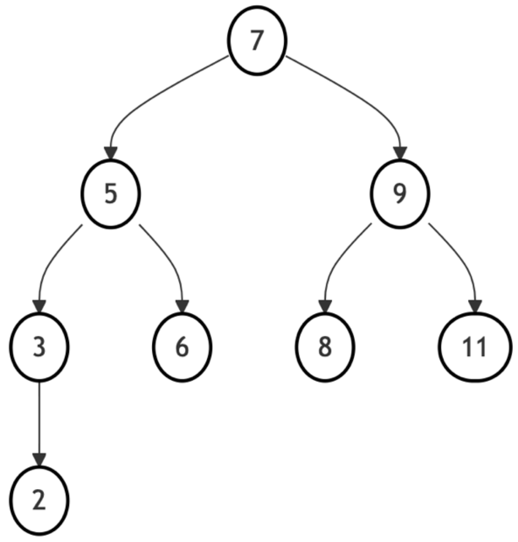
left child

Question!

(2 marks) The following array represents a complete binary tree. Does it represent a binary search tree?

◯ Yes ◯ No

7	5	9	3	6	8	11	2



left child

Question!

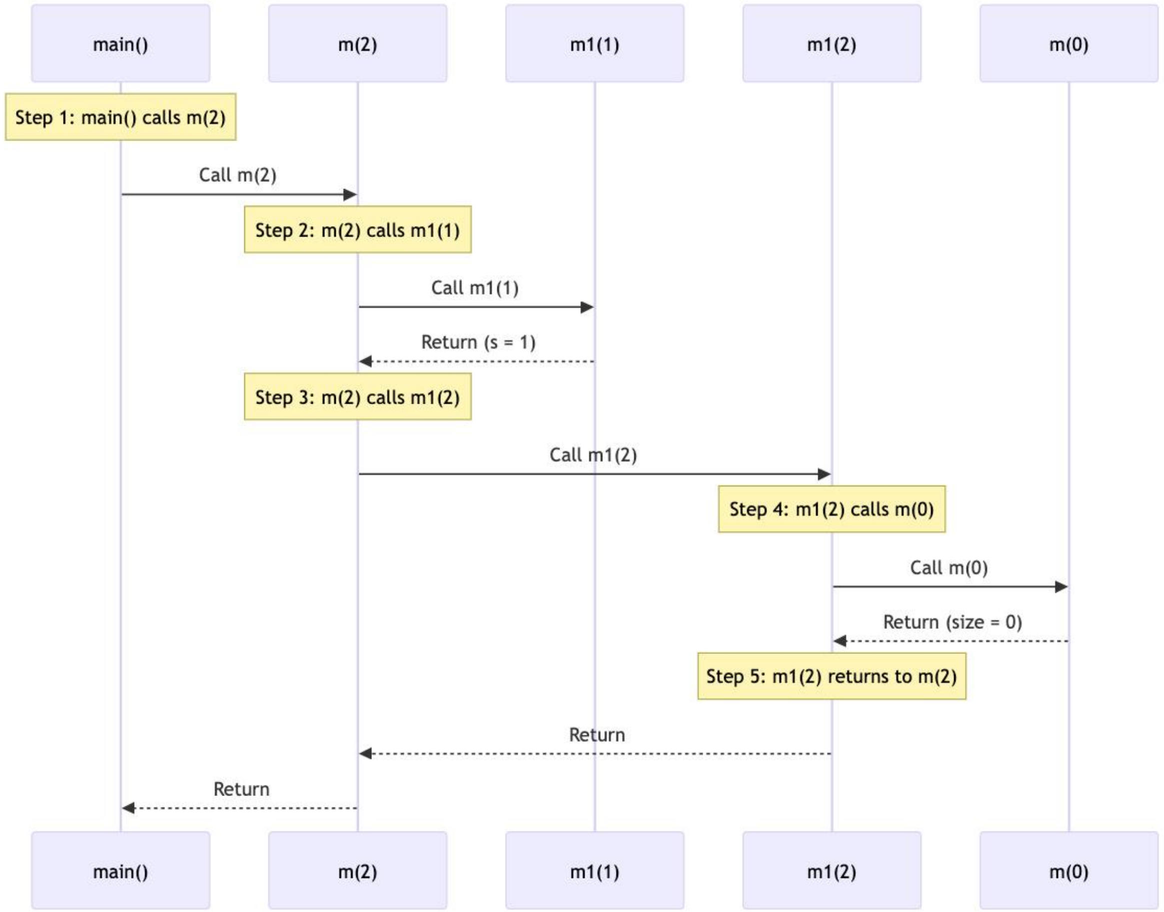
(3 marks) Consider the following Java program.

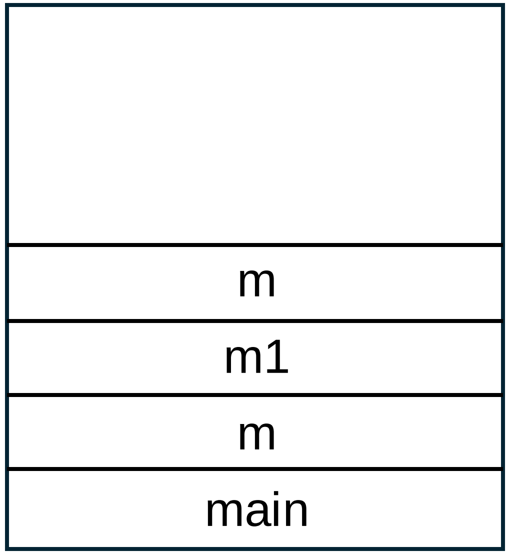
```
public static void m(int size) {
    if (size == 0) return;
    else m1(size-1);
    m1(size);
}
```

```
public static void m1(int s) {
    int i = s-2;
    if (s <= 1) return;
    else m(i);
}
public static void main(String[] args) {
    m(2);
}
```

An activation record for method **m1** uses 10 bytes, an activation record for method **m** uses 10 bytes, and an activation record for **main** uses 10 bytes. How much memory is needed for the execution stack when the program is executed? The amount of memory needed is equal to the total size of the maximum number of activation records that are in the execution stack at the same time.

- ○ 10 bytes
- ○ 20 bytes
- ○ 30 bytes
- ○ 40 bytes
- ○ 50 bytes





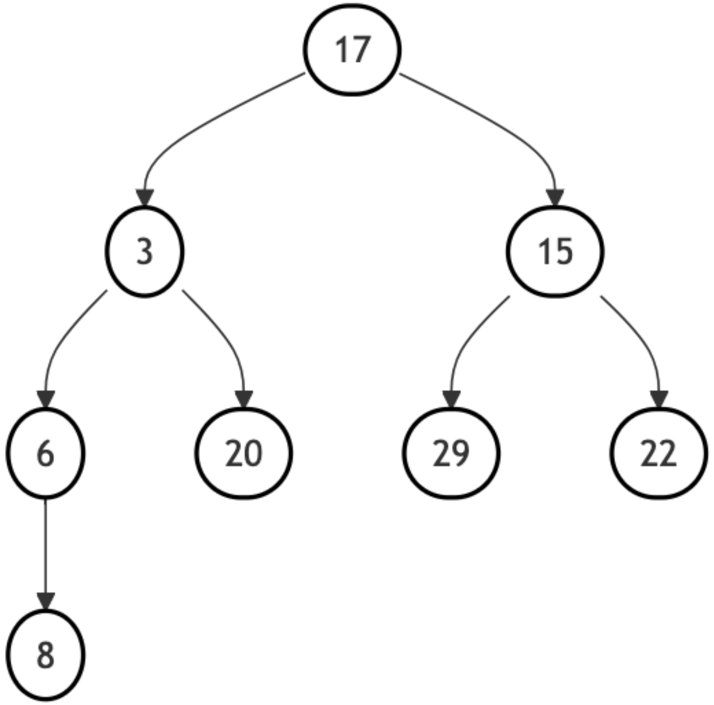
Execution Stack @ Step 4

Question!

(5 marks) Consider the following min heap. Delete the minimum value from the min heap and show the resulting min heap in the empty array provided. You must use the algorithm studied in class for removing the minimum value from a heap.

Hint. If you do not remember the algorithm, it first removes the minimum value replacing it with the last value in the min heap. Then, repeatedly it traverses down the heap comparing the value stored in a node with the smaller value in its children, swapping values if needed.

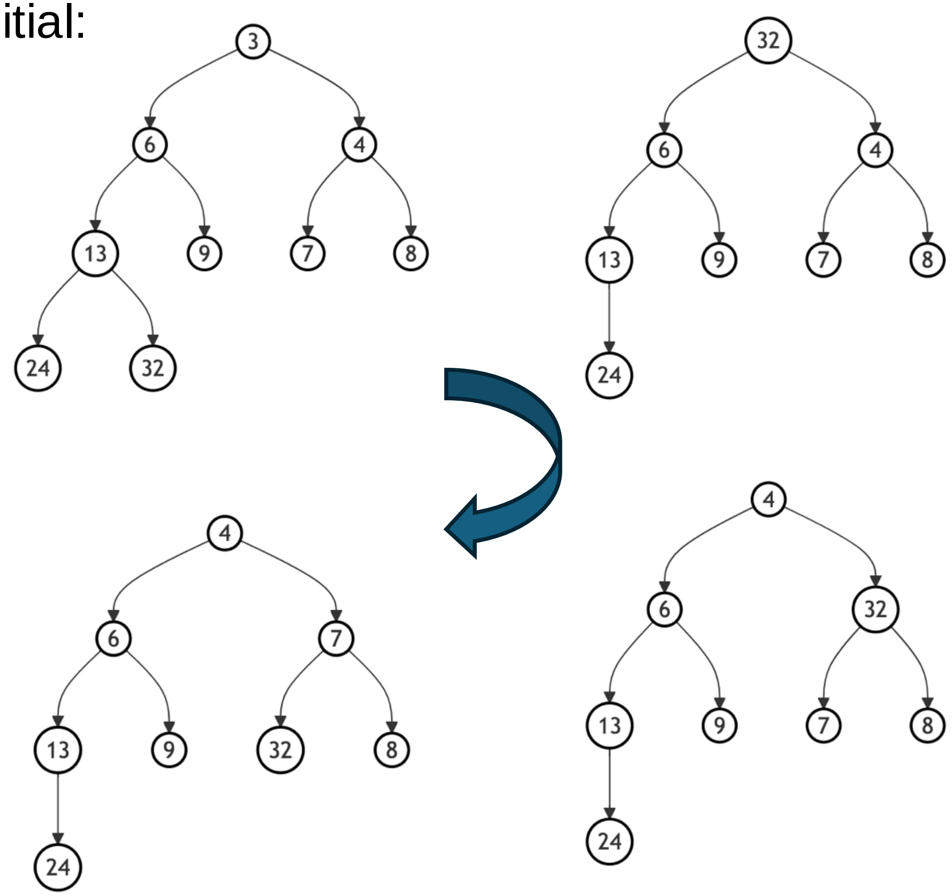
	0	1	2	3	4	5	6	7	8
Min heap	2	3	15	6	20	29	22	8	17



After Remove the Minimum Value

Recall: Removing Minimum Value

Initial:



3	6	4	13	9	7	8	24	32
0	1	2	3	4	5	6	7	8

result = 3, save minimum value

32	6	4	13	9	7	8	24	
0	1	2	3	4	5	6	7	8

swap with smallest child

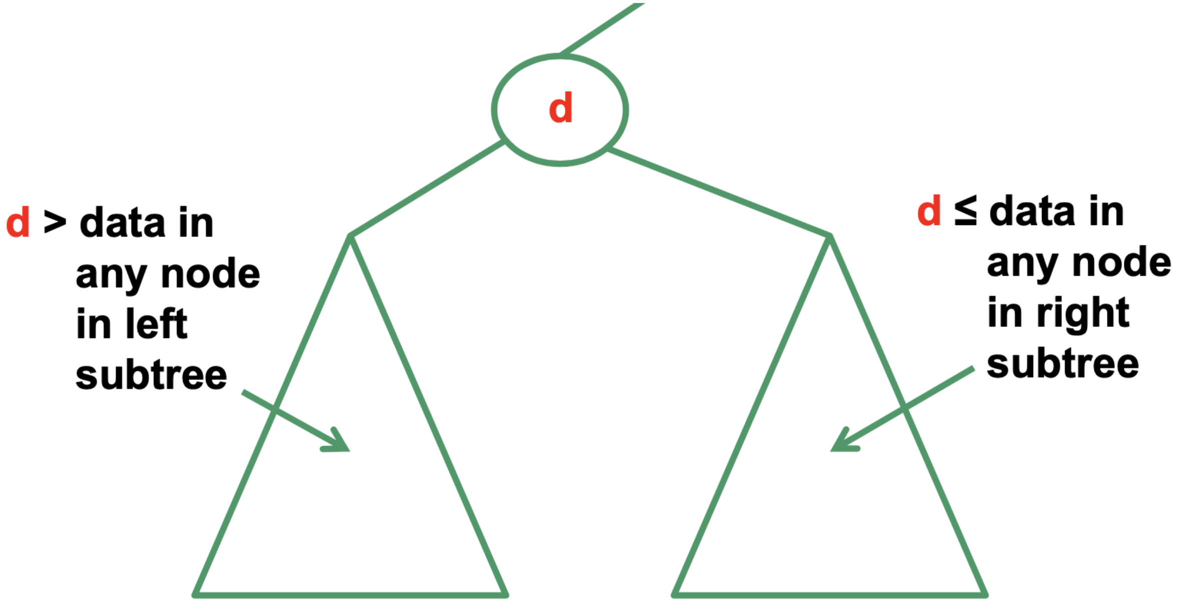
32	6	4	13	9	7	8	24	
0	1	2	3	4	5	6	7	8

4	6	32	13	9	7	8	24	
0	1	2	3	4	5	6	7	8

4	6	7	13	9	32	8	24	

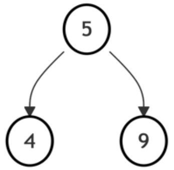
Recall: Binary Search Tree

- A binary search tree (BST) is a binary tree in which the data in every internal node is greater than the data in any node in its left subtree and smaller than or equal to the data in any node in its right subtree



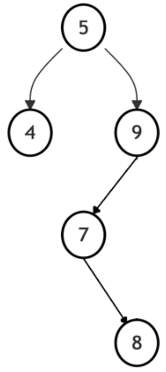
Question!

(4 marks) Draw a **binary search tree** storing the values 4, 5, 7, 8, 9 such that a preorder traversal of the tree visits the nodes in the order: 5, 4, 9, 7, 8.



What is given:

- The given preorder traversal is: **5, 4, 9, 7, 8**
- Remember: Preorder traversal visits nodes in the order: **Root → Left Subtree → Right Subtree**.



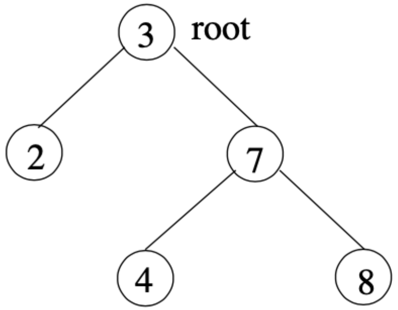
To construct the BST, follow the properties of a BST:

1. The **root** is the first element of the preorder traversal (5).
2. For each subsequent element:
    1. Insert into the **left subtree** if it is smaller than the current node.
    2. Insert into the **right subtree** if it is larger than the current node.

Question!

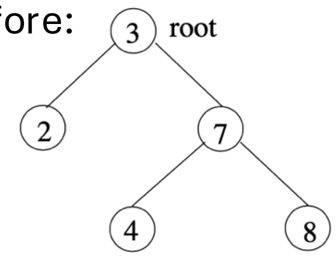
(6 marks) Consider the following algorithm.

```
public void change(BinaryTreeNode r) {
    BinaryTreeNode left = r.leftChild(), right = r.rightChild();
    if (left != null && right != null) {
        change(left);
        change(right);
        left.setRightChild(right.leftChild());
        right.setLeftChild(left);
        r.setLeftChild(null);
    }
}
```



Draw the tree produced by the above algorithm when executed on the following tree, i.e. when the algorithm is invoked as `change (root)`.

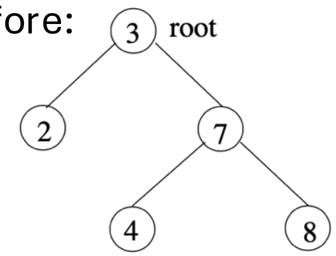
Before:



Trace Table

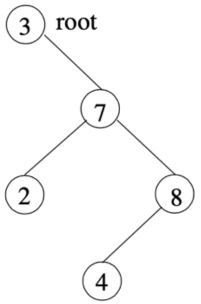
Step	Current Node (r)	Left Child (left)	Right Child (right)	Operation/Modification	Resulting Tree
1	3	2	7	Recursively call change(left = 2) and change(right = 7).	Tree remains the same.
2	2	null	null	No modification because `left == null	
3	7	4	8	Recursively call change(left = 4) and change(right = 8)	Tree remains the same.
4	4	null	null	No modification because `left == null	
5	8	null	null	No modification because `left == null	

Before:



Trace Table

After:



Step	Current Node (r)	Left Child (left)	Right Child (right)	Operation/Modification	Resulting Tree
6	7	4	8	Modify: - left.setRightChild(right.leftChild()) (sets 4.rightChild = null) - right.setLeftChild(left) (sets 8.leftChild = 4) - r.setLeftChild(null) (sets 7.leftChild = null)	Tree after modification: 7.rightChild = 8 and 8.leftChild = 4.
7	3	2	7	Modify: - left.setRightChild(right.leftChild()) (sets 2.rightChild = null) - right.setLeftChild(left) (sets 7.leftChild = 2) - r.setLeftChild(null) (sets 3.leftChild = null).	Final tree: 3.rightChild = 7 and 7.leftChild = 2.


Thank you