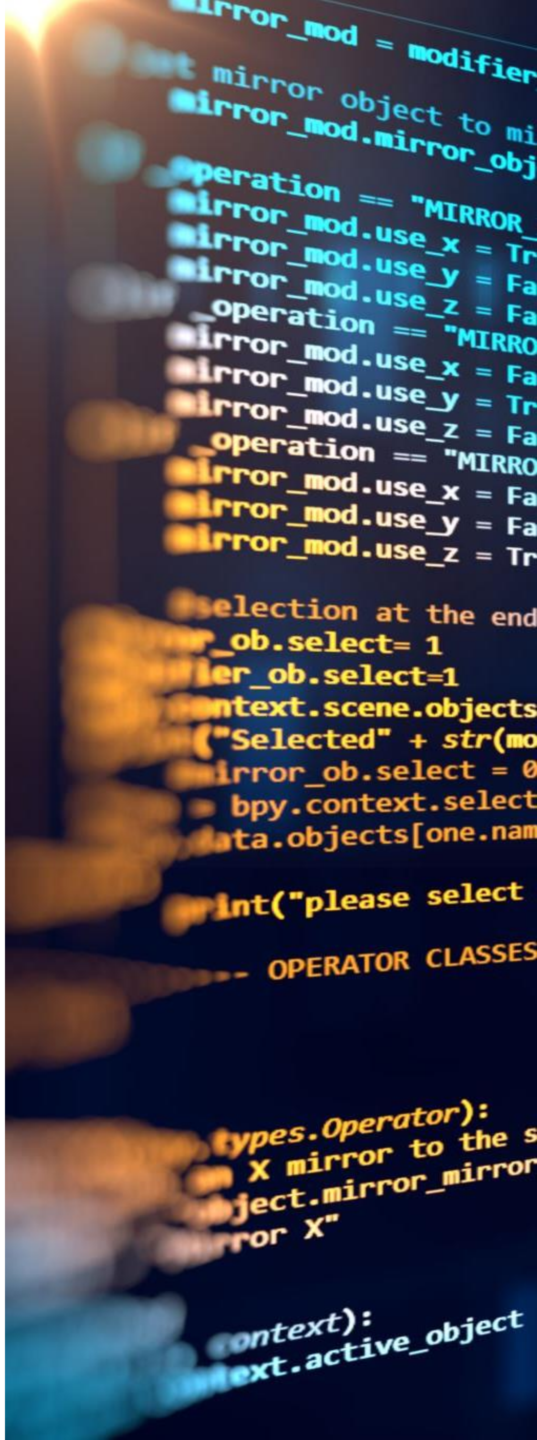
Attendance

Please use the following QR code to check in and record your attendance

CS 1027
Fundamentals of Computer Science II

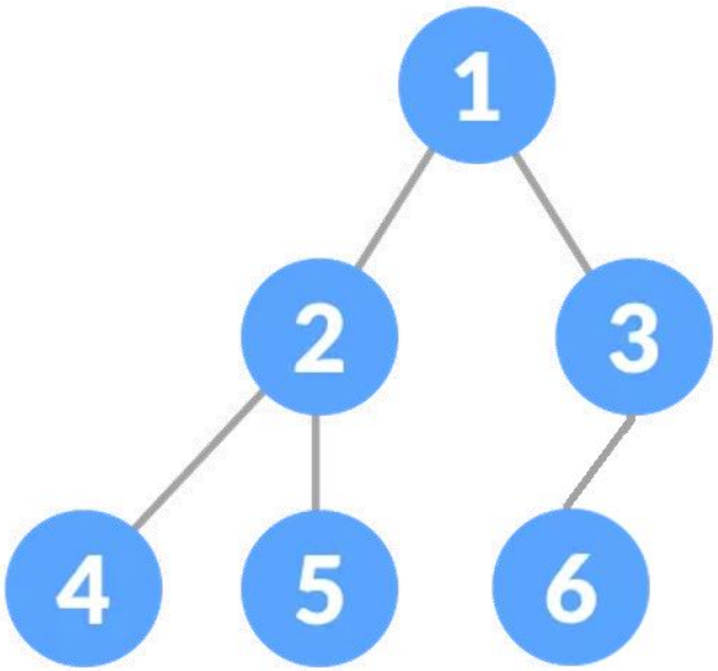
Trees ADT (cont.)

Ahmed Ibrahim



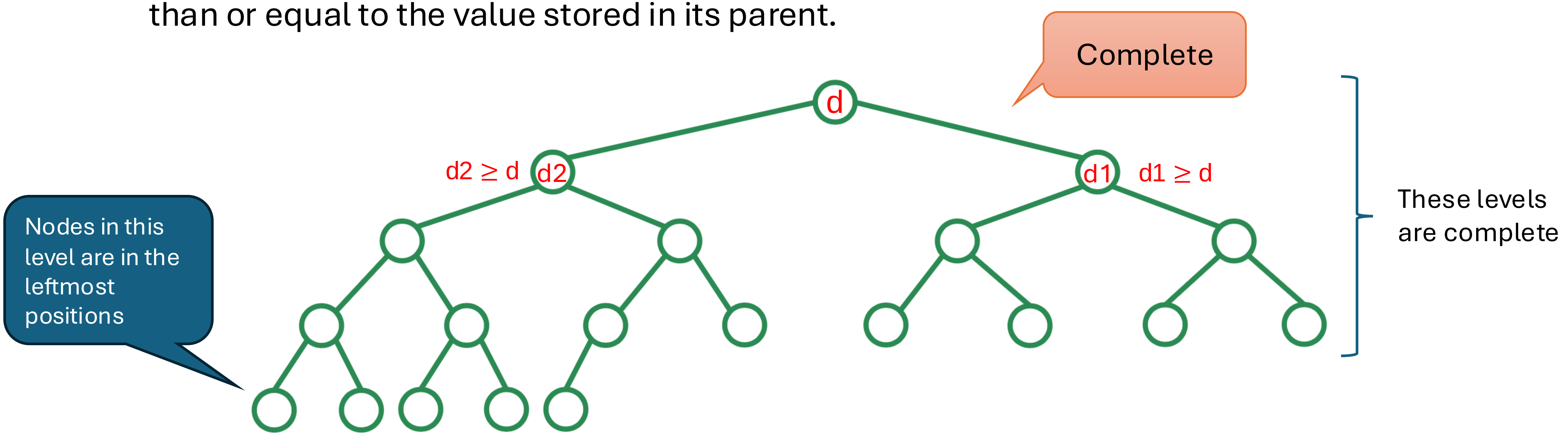
Recall: Complete Binary Tree

- A complete binary tree is a binary tree in which every level, except possibly the last, is **completely filled**, and all nodes are as far left as possible.
- This means that:
  - All levels above the last level are fully filled.
  - The last level may not be fully filled, but if it has missing nodes, those nodes are only on the right side (i.e., **all leaf nodes lean to the left**).



Min Heap

- A min-heap is a **binary tree** with the following properties:
  - The value stored in each node, except the root, is larger than or equal to the value stored in its parent.

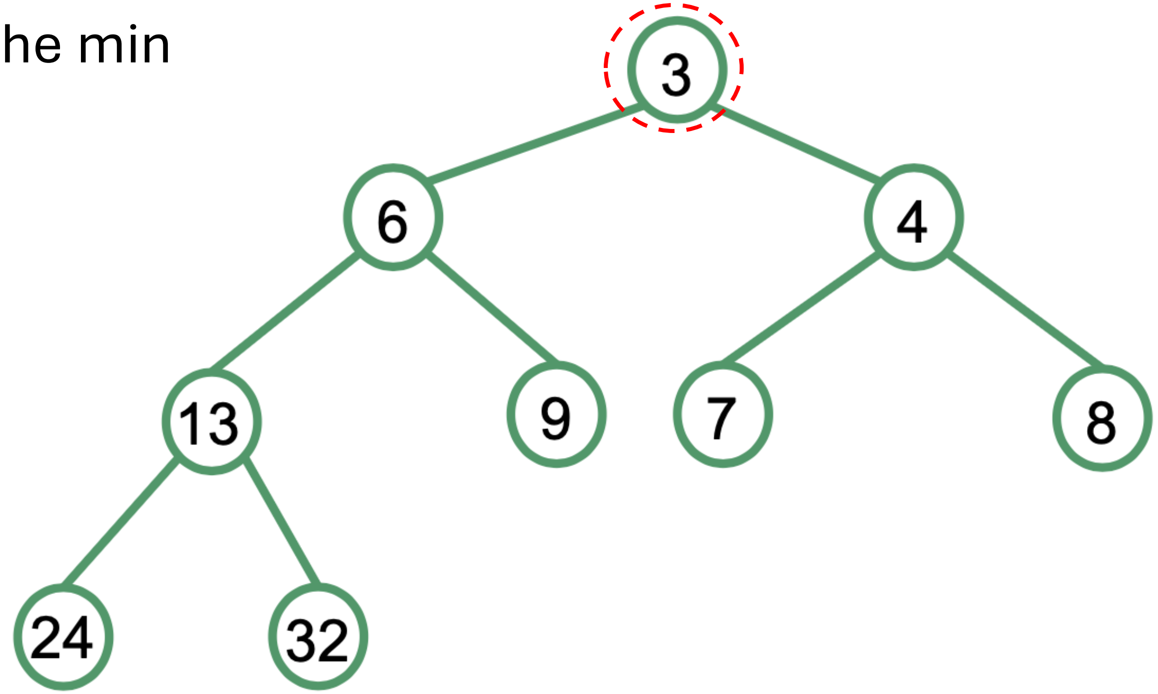


Heap Properties

- A heap is a **complete binary tree**, which means that every level of the tree is **fully filled**, except possibly the last level, which is filled from left to right.
- The **heap property** (max or min) must hold TRUE for the root and all **subtrees**.
- Heap property ensures efficient operations:
  - Elements can be inserted and deleted efficiently in O(log n) time.
  - Accessing the largest (in max-heap) or smallest (in min-heap) element is O(1) – constant time.

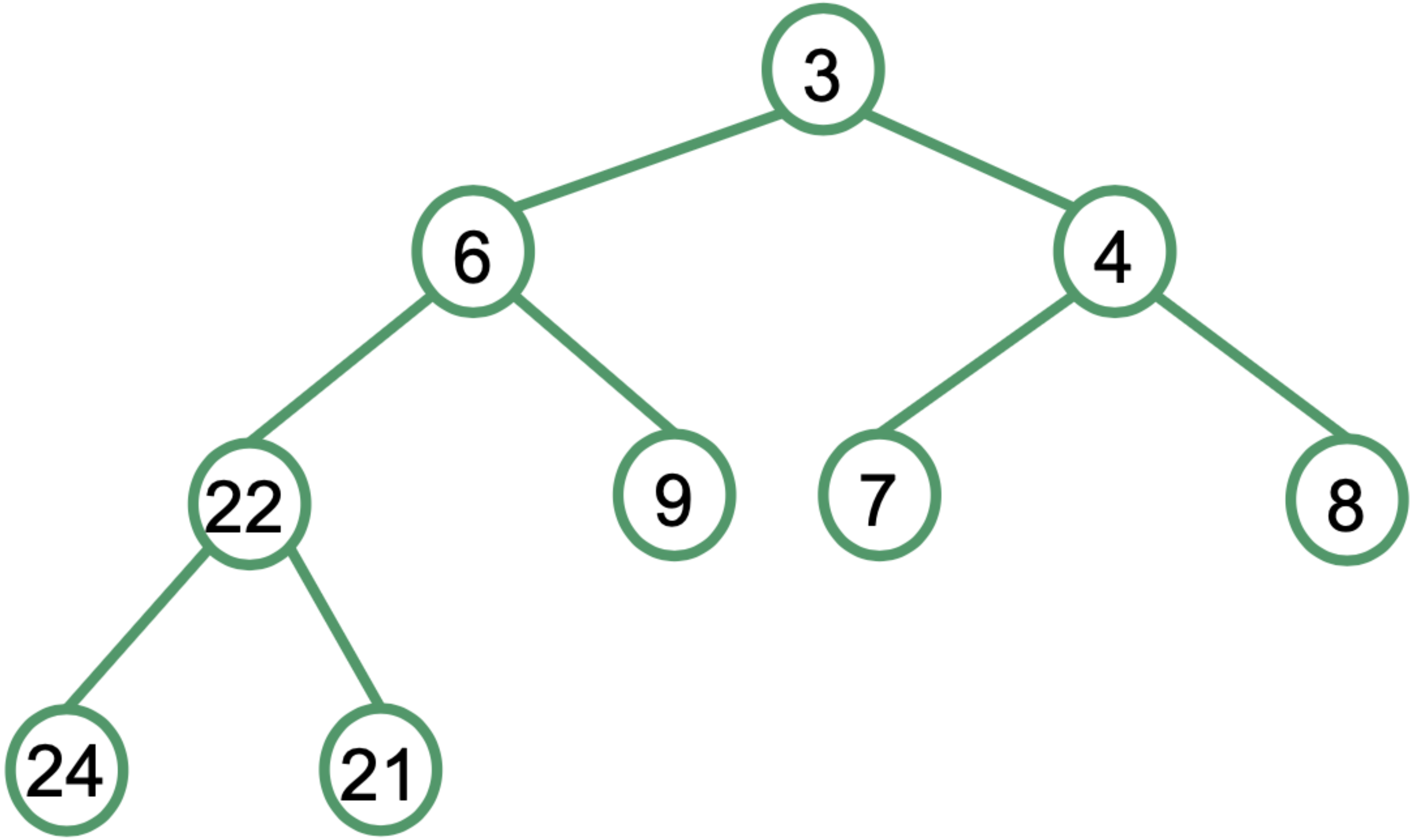
Example of Min Heap

- Note that the smallest value in the min heap is stored in the root.



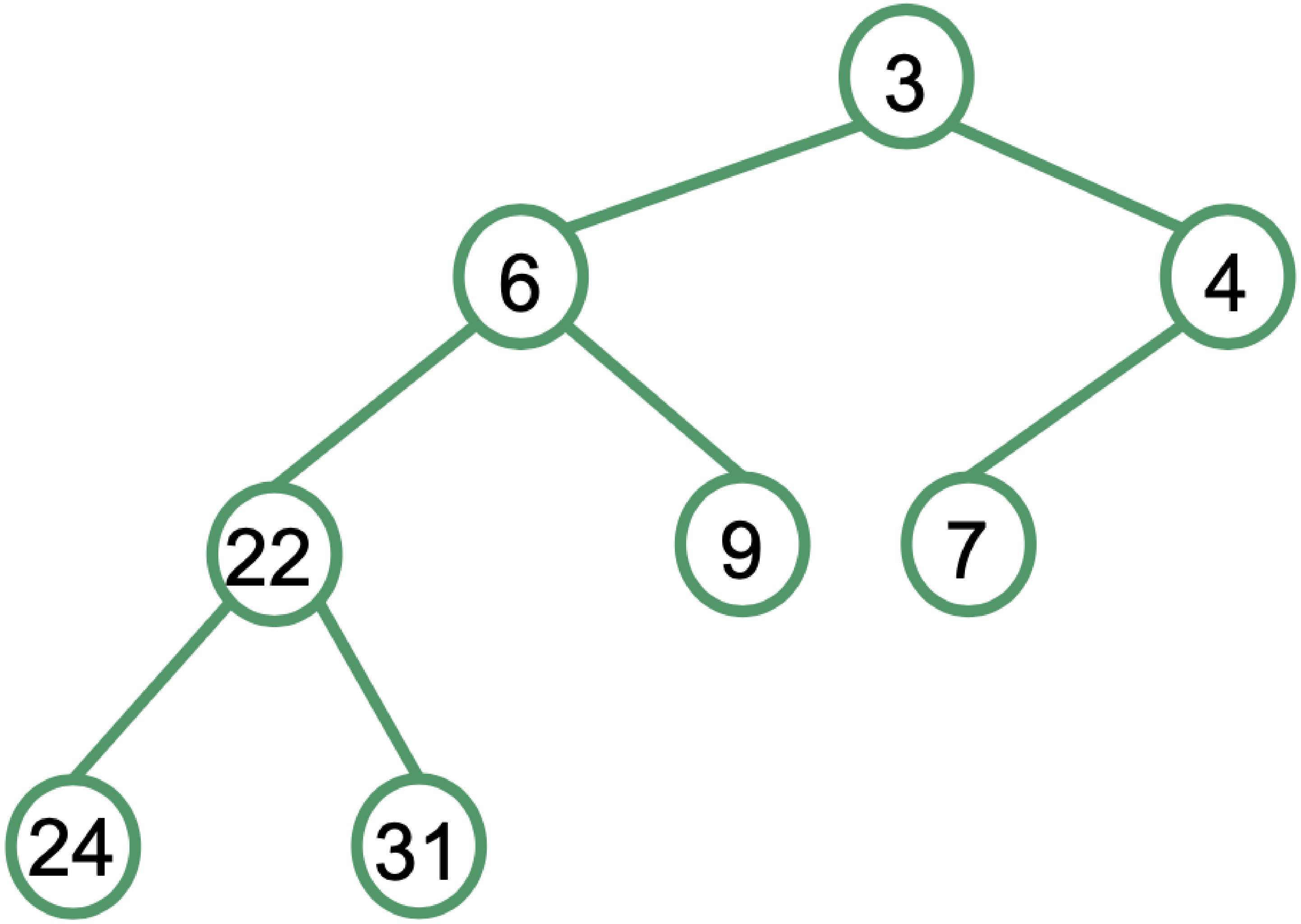
Example of Min Heap

- Is this a min heap?
- Is this a complete tree?
- Does the root have a minimum value?



Example of Min Heap

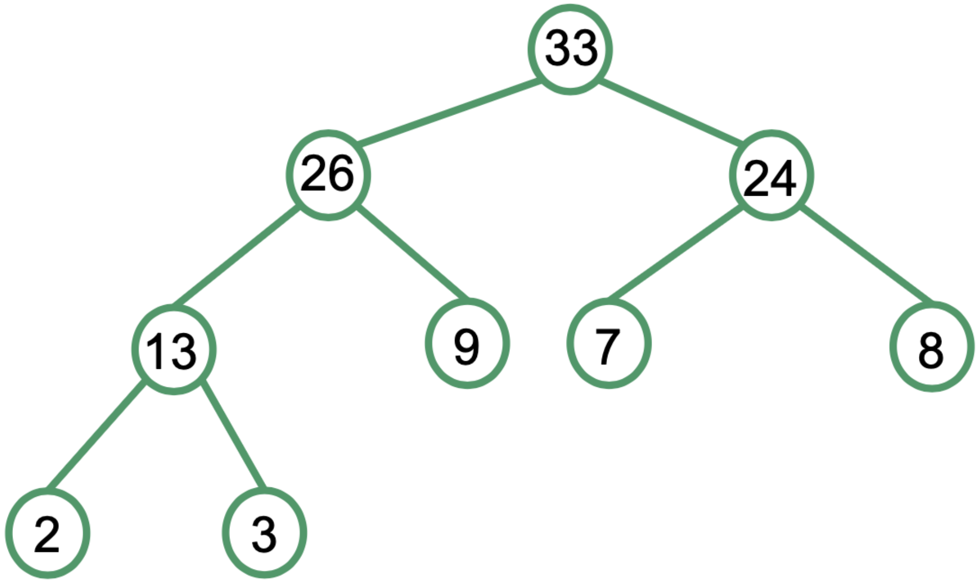
- Is this a min heap?
- Is this a complete tree?
- Does the root have a minimum value?



Max Heap

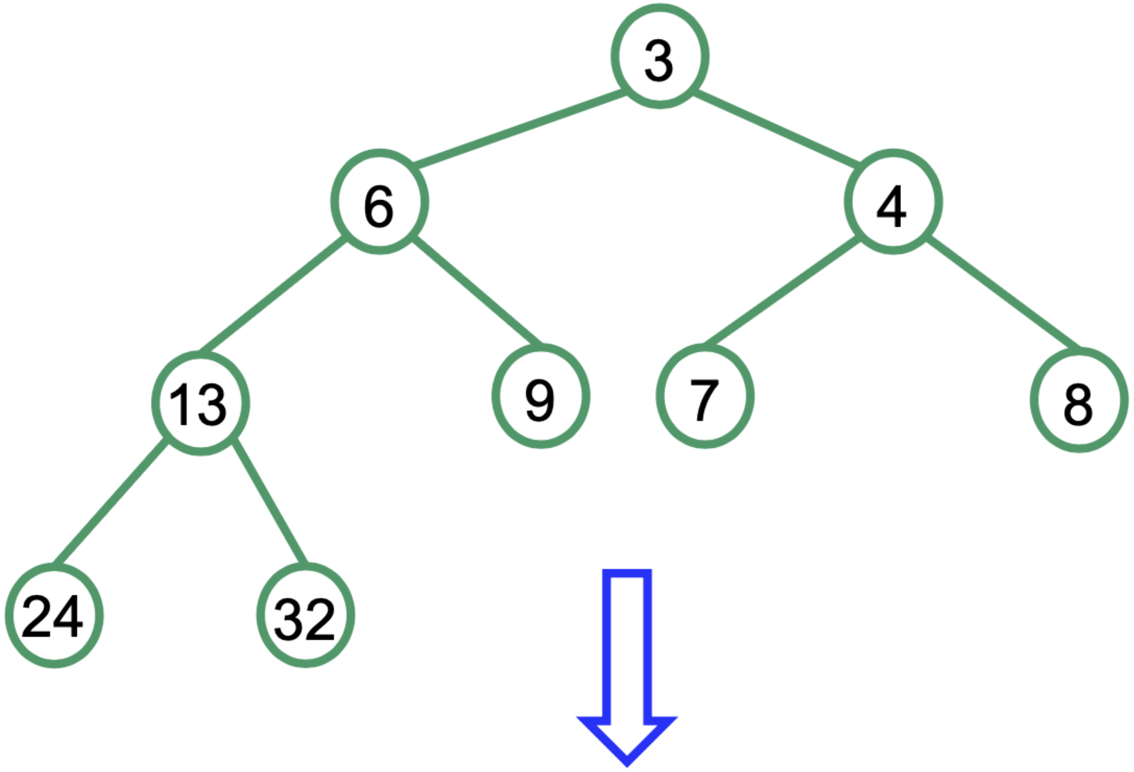
- A max-heap is similar to a min-heap, except each node stores a value, and the root is greater than or equal to the value stored in its children.
- The following is an example of a max heap

Note that the largest value in the max heap is stored in the root.



Heap Implementation with Arrays

- Since max heaps and min heaps are complete trees, then they can be efficiently implemented using arrays and without the need to use linked structures:
  - Store the root in position 0 of the array, the left child of the root is stored in position 1 and the right child in position 2
  - For a node stored in position i of the array its left child is stored in position 2*i+1 and its right child in position 2*i+2



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

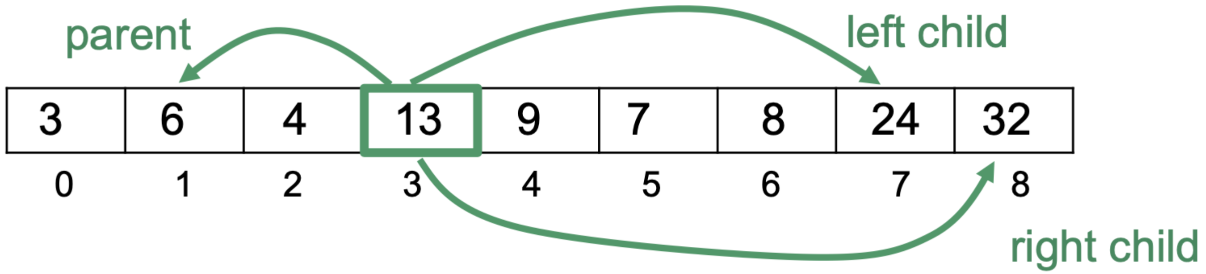
Note that the links connecting nodes to their children do not need to be implicitly stored

Heap Implementation with Arrays

- Note that in this representation, the links connecting nodes to their children and to their parents do not need to be implicitly stored. For the node stored in position i of the array:
  - its left child is in position 2*i+1
  - its right child is in position 2*i + 2
  - its parent is in position $\lfloor (i-1)/2 \rfloor$

Floor function

- This implementation is very memory efficient.



- The **floor function** takes a real number as input and returns the greatest integer less than or equal to that number. In simpler terms, it "rounds down" a number to the nearest whole number.

Making a Min Heap

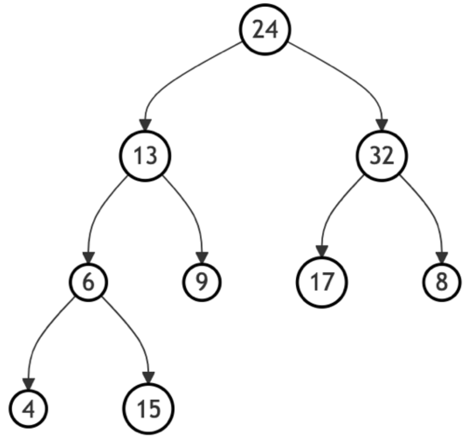
```java
public void makeHeap(T[] arr, int n) {
Comparable<T> childComp; // This variable will hold the value of the child node being compared
T swap;                  // Temporary variable for swapping values

// Start from the second element (index 1) and process each element up to index n-1
for (int i = 1; i < n; ++i) {
   int parent = (i - 1) / 2; // Find the parent index of the current element
   int child = i;            // Current child index
   childComp = (Comparable<T>) arr[child]; // The child node's value, cast to Comparable for comparison

  // Bubble up: while the child is smaller than the parent, swap them
  while (parent >= 0 && childComp.compareTo(arr[parent]) < 0) {
   swap = arr[child];
   arr[child] = arr[parent];
   arr[parent] = swap;
   child = parent; // Update child and parent indices for the next comparison
   parent = (parent - 1) / 2; } // Move up to the parent's parent
 }
}
```

24	13	32	6	9	17	8	4	15
0	1	2	3	4	5	6	7	8

Making a Min Heap

- To create a heap storing a given set of values, we first store the values in an array.

24	13	32	6	9	17	8	4	15
0	1	2	3	4	5	6	7	8



Complete Binary Tree

- Then, for each position, i:
  - Check that the parent ⌊(i−1)/2⌋ stores a value smaller, and if not swap the values and recursively check for this property for position ⌊(i−1)/2⌋

24	13	32	6	9	17	8	4	15
0	1	2	3	4	5	6	7	8

13 < 32, no swap

13	24	32	6	9	17	8	4	15
0	1	2	3	4	5	6	7	8

Making a Min Heap

```
public void makeHeap(T[] arr, int n) {
Comparable<T> childComp; // This variable will hold the value of the child node being compared
T swap;                  // Temporary variable for swapping values

// Start from the second element (index 1) and process each element up to index n-1
for (int i = 1; i < n; ++i) {
   int parent = (i - 1) / 2; // Find the parent index of the current element
   int child = i;            // Current child index
   childComp = (Comparable<T>) arr[child]; // The child node's value, cast to Comparable for comparison

  // Bubble up: while the child is smaller than the parent, swap them
  while (parent >= 0 && childComp.compareTo(arr[parent]) < 0) {
   swap = arr[child];
   arr[child] = arr[parent];
   arr[parent] = swap;
   child = parent; // Update child and parent indices for the next comparison
   parent = (parent - 1) / 2; } // Move up to the parent's parent
 }
}
```

Now: $i = 3$

The Process of Heapifying

13	24	32	6	9	17	8	4	15
0	1	2	3	4	5	6	7	8

6	13	32	24	9	17	8	4	15
0	1	2	3	4	5	6	7	8

6	9	32	24	13	17	8	4	15
0	1	2	3	4	5	6	7	8

6	9	17	24	13	32	8	4	15
0	1	2	3	4	5	6	7	8

6	9	8	24	13	32	17	4	15
0	1	2	3	4	5	6	7	8

9 < 15, no swap

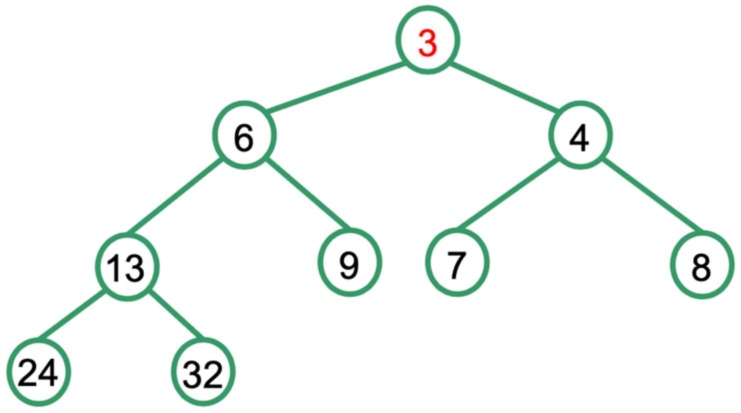
4	6	8	9	13	32	17	24	15
0	1	2	3	4	5	6	7	8

4	6	8	9	13	32	17	24	15
0	1	2	3	4	5	6	7	8

Min heap completed

Removing Minimum Value

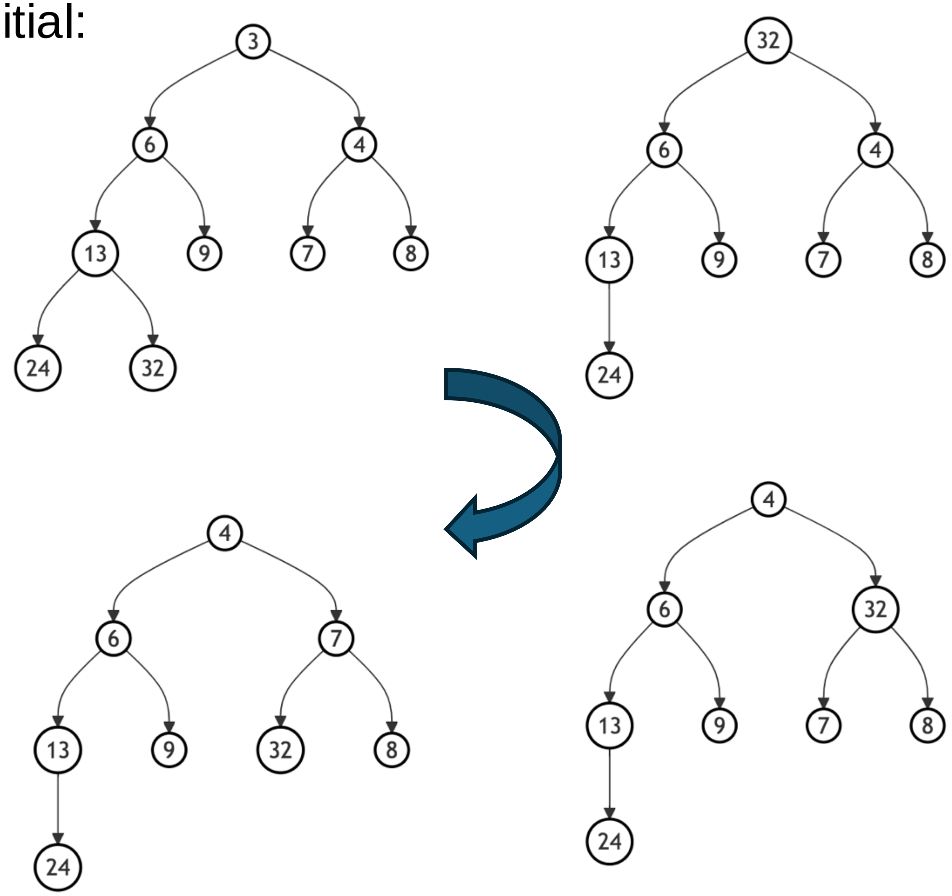
- The minimum value of a heap is stored in the root (index 0 in the array representation). To remove it:
  - Save it in some variable and replace the value in the root with the value in the last node (with index n-1)
  - Delete the last node and recursively check that the heap property holds for the root and its children.



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

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	

Removing Minimum Value Algorithm

```
public T removeMin(T[] arr, int n) {
Comparable<T> childComp; T swap;

T result = arr[0]; arr[0] = arr[n - 1]; n = n - 1;

// Start reheapifying from the root (parent) of the heap
int parent = 0; int child = 1;

while (child < n) {
   childComp = (Comparable<T>) arr[child];

  if ((child + 1 < n) && childComp.compareTo(arr[child + 1]) > 0) // Check if the parent has two children and the right child is smaller
    {child = child + 1;}

  if (childComp.compareTo(arr[parent]) < 0) {swap = arr[child]; // If the child is smaller than the parent, swap them
   arr[child] = arr[parent];
   arr[parent] = swap;
   parent = child; // Move down the heap: update parent and child indices
   child = 2 * parent + 1; // Left child of the new parent
    } else {break;}
}
return result;}
```

Sorting with Heaps

- To sort a set of values stored in an array:
  - First, convert the array into a heap
  - Repeatedly use the ***removeMin*** operation to store the values in increasing order in a second, sorted array
- This is called heapsort.
- In heapsort, we repeatedly extract the root (the largest or smallest element, depending on whether it’s a max-heap or min-heap) and reconstruct the heap.

Heapsort

Input array

24	13	32	6	9	17	8	4	15
0	1	2	3	4	5	6	7	8

$\Downarrow$ Make heap

Heap

4	6	8	9	13	32	17	24	15
0	1	2	3	4	5	6	7	8

$\Downarrow$ remove min

Heap

6	9	8	15	13	32	17	24	
0	1	2	3	4	5	6	7	8

Sorted array

4								
0	1	2	3	4	5	6	7	8

Heapsort

remove min

Heap

8	9	17	15	13	32	24		
0	1	2	3	4	5	6	7	8

Sorted array

4	6							
0	1	2	3	4	5	6	7	8

remove min

Heap

9	13	17	15	24	32			
0	1	2	3	4	5	6	7	8

Sorted array

4	6	8						
0	1	2	3	4	5	6	7	8

Heapsort

remove min

13	15	17	32	24				
0	1	2	3	4	5	6	7	8

Sorted array

4	6	8	9					
0	1	2	3	4	5	6	7	8

remove min

...

Sorted array

4	6	8	9	13	15	17	24	32
0	1	2	3	4	5	6	7	8

Sorting

Sorting Problem

- The sorting problem involves arranging the elements of a given list (or array) in a specific order, such as ascending or descending order.

9	5	19	35	8	17	23	14

5	8	9	14	17	19	23	35

Why Sorting is Important?

- Sorting is a key operation because it facilitates
  - **Searching**: Many algorithms, such as binary search, require sorted data.
  - **Data Organization**: Sorting makes datasets easier to analyze and interpret.
  - **Optimization**: In some problems (e.g., scheduling), sorting helps optimize solutions.
  - **Preprocessing**: Sorting is a prerequisite for many algorithms, such as merge-based techniques or partitioning methods.

Types of Sorting Problems

- Sorting problems can be classified into:

- **Comparison-Based Sorting**:
  - Sorting based on pairwise comparisons of elements.
  - Examples: Quick Sort, Merge Sort, Heap Sort, Insertion Sort, Selection Sort

- **Non-Comparison-Based Sorting**:
  - Sorting without directly comparing elements.
  - Uses properties like digit positions or buckets.
  - Examples: Radix Sort, Counting Sort, Bucket Sort.

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

In-Place Insertion Sort

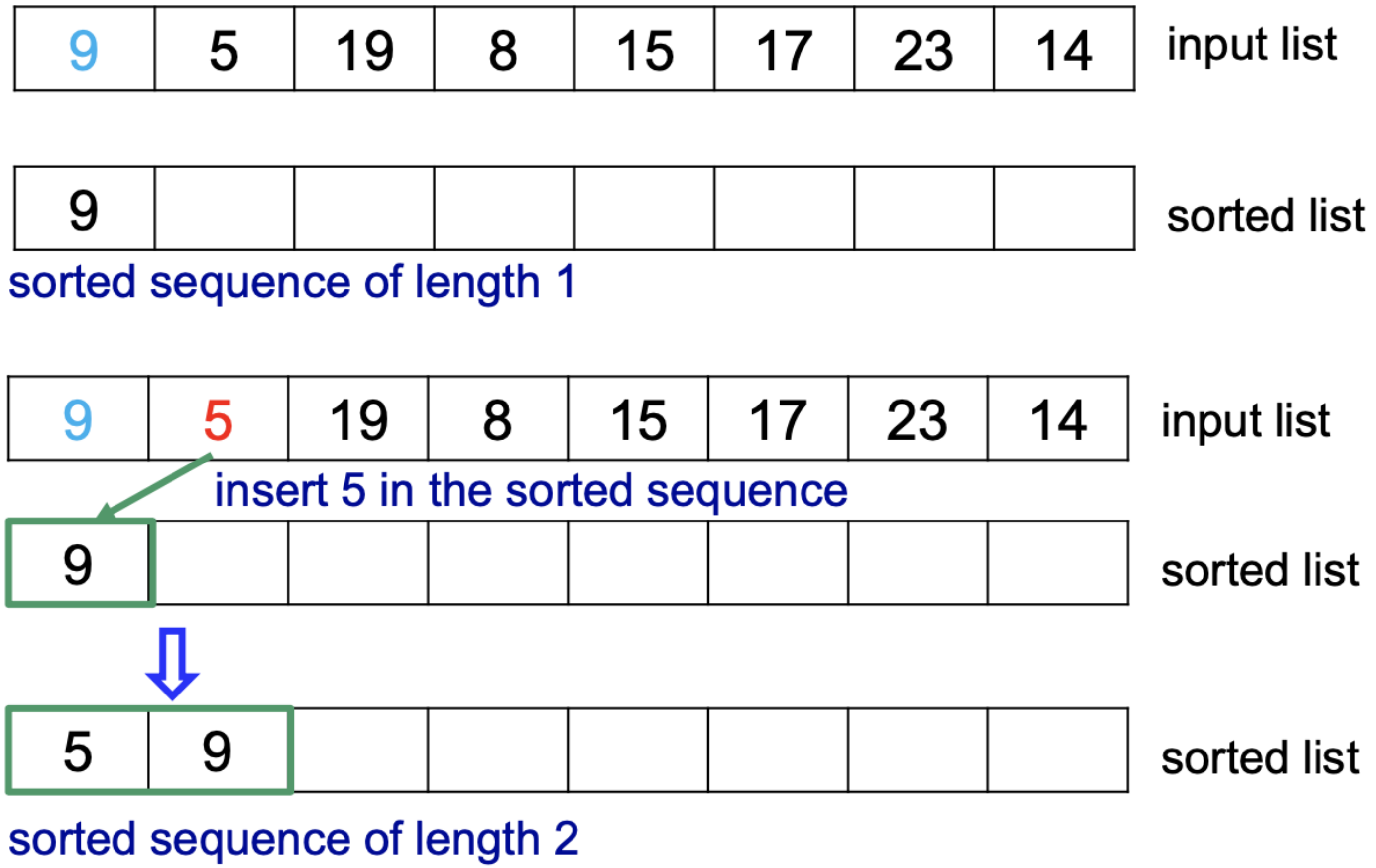
6 5 3 1 8 7 2 4

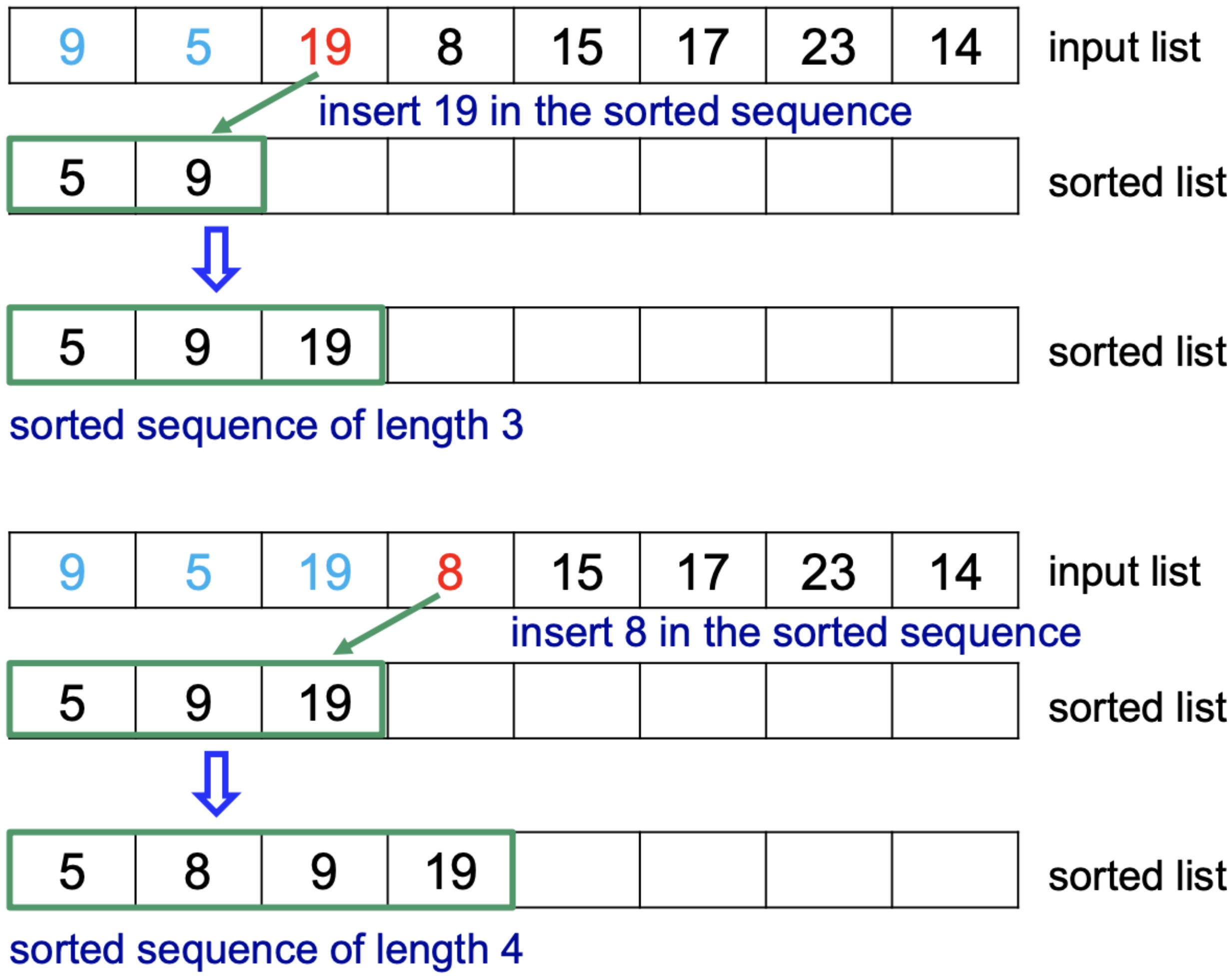
- An in-place sorting algorithm does not use auxiliary data structures, so in-place sorting algorithms are memory efficient.
- Consider that the input sequence is stored in an array.
- In-place insertion sort works as follows:
  - the sub-array containing the first value is sorted
  - add the second value to the sorted sub-array shifting values as needed to get a sorted sub-array of size 2
  - add the third value to the sorted sub-array shifting values as needed to get a sorted sub-array of size 3
  - keep doing this until the entire array is sorted

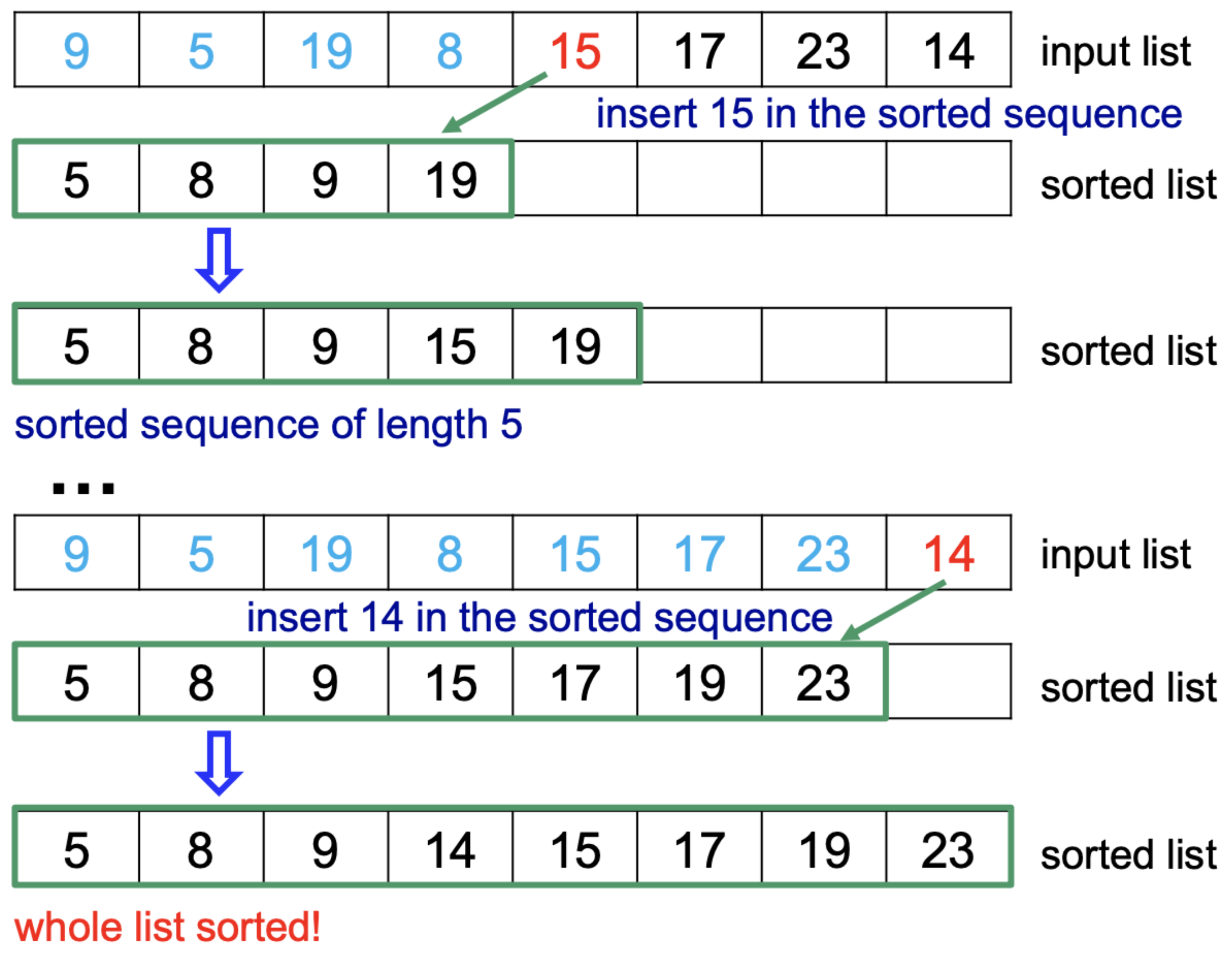
Insertion Sort

```
public void insertionSort (T[] A, int n) {
for (int i = 1; i < n; ++i) {
 // Insert A[i] in the sorted sub-array A[0..i-1]
 T temp = A[i];
 Comparable<T> tempComp = (Comparable<T>)temp;
 int j = i - 1;
 while ((j >= 0) && (tempComp.compareTo(A[j]) < 0)) {
  A[j+1] = A[j];
  j = j - 1;}
  A[j+1] = temp;
}
```

Another method for implementing insertion sort is to use an auxiliary array.

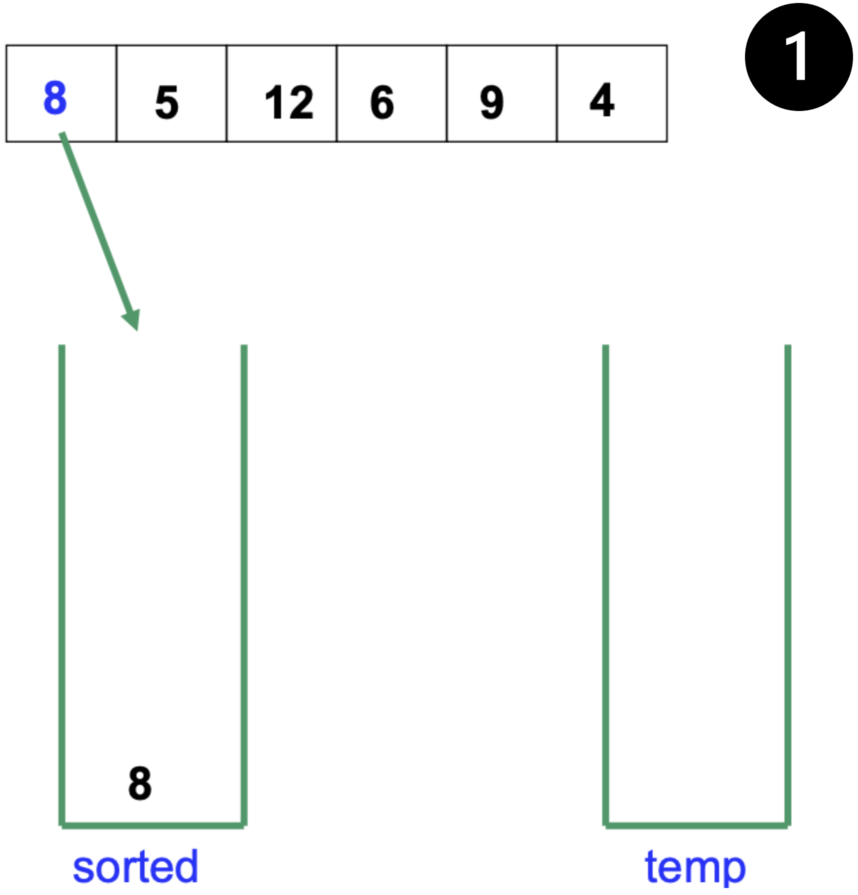



9 5 19 8 15 17 23 14
input list
insert 19 in the sorted sequence
5 9
sorted list
5 9 19
sorted list
sorted sequence of length 3
9 5 19 8 15 17 23 14
input list
insert 8 in the sorted sequence
5 9 19
sorted list
5 8 9 19
sorted list
sorted sequence of length 4

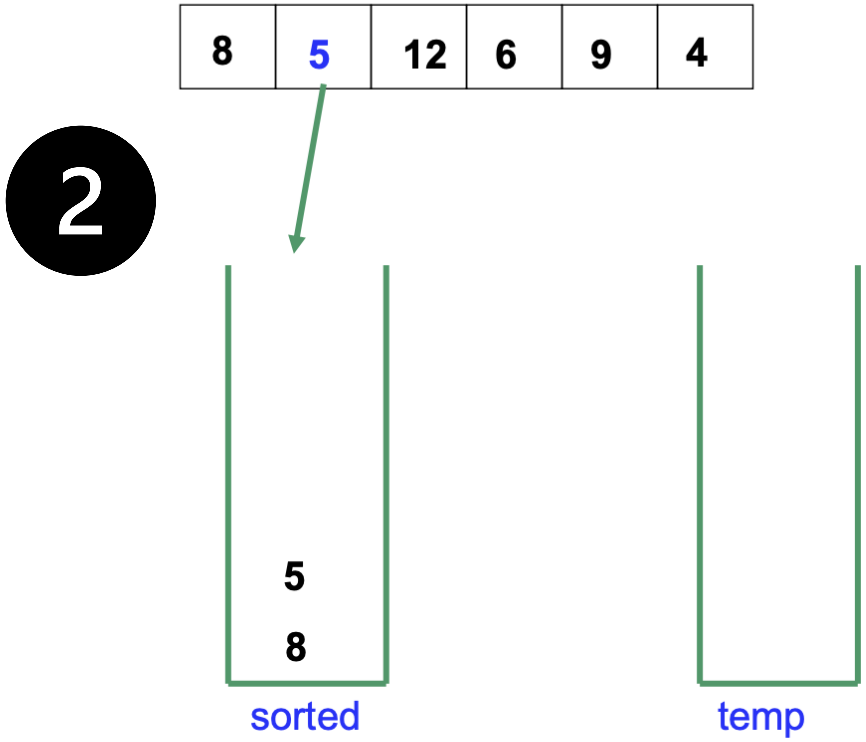

9 5 19 8 15 17 23 14
input list
insert 15 in the sorted sequence
5 8 9 19
sorted list
5 8 9 15 19
sorted list
sorted sequence of length 5
...
9 5 19 8 15 17 23 14
input list
insert 14 in the sorted sequence
5 8 9 15 17 19 23
sorted list
5 8 9 14 15 17 19 23
sorted list
whole list sorted!

Insertion Sort Using Stacks

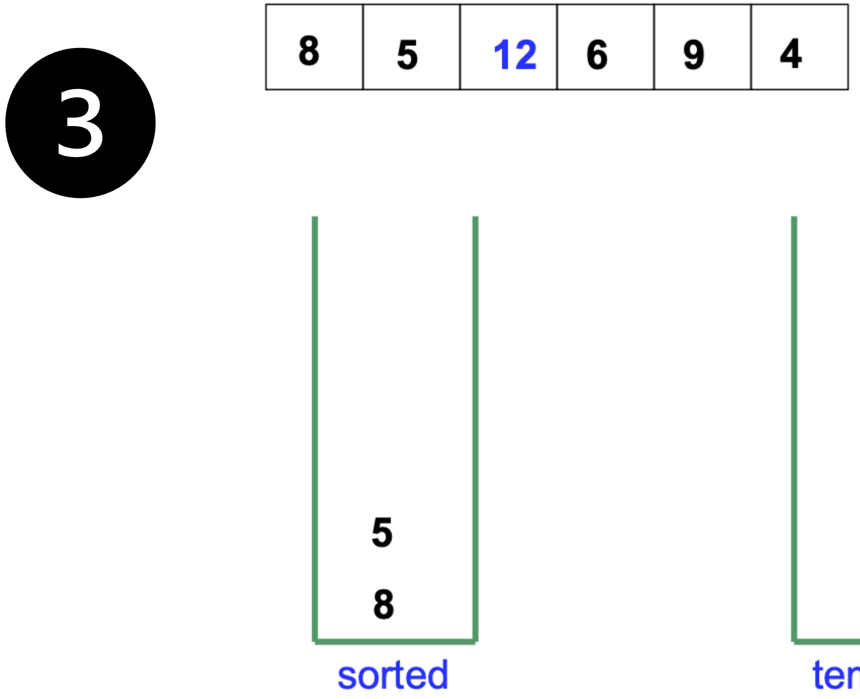
- Use two temporary stacks called sorted and temp, both of which are initially empty.
- The contents of the sorted will always be **in order**, with the smallest item on the top of the stack.
- This will be the "sorted subsequence"
- temp will temporarily hold items that need to be "shifted" out to insert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



Insertion Sort Using Stacks

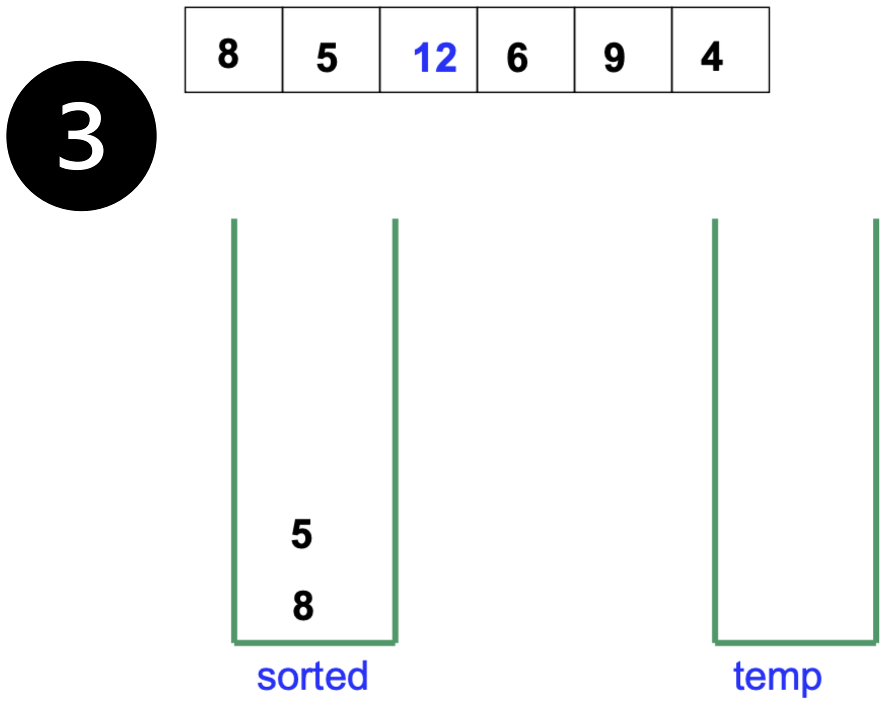


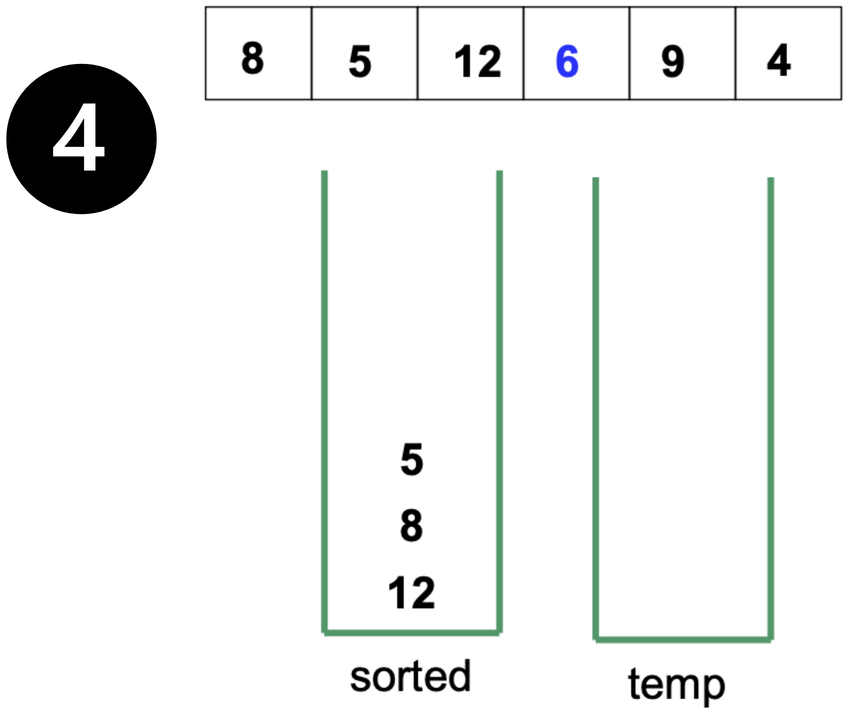
Since 12 > 5, we need to move the values from sorted to temp, push 12 into sorted and move the values back from temp to sorted



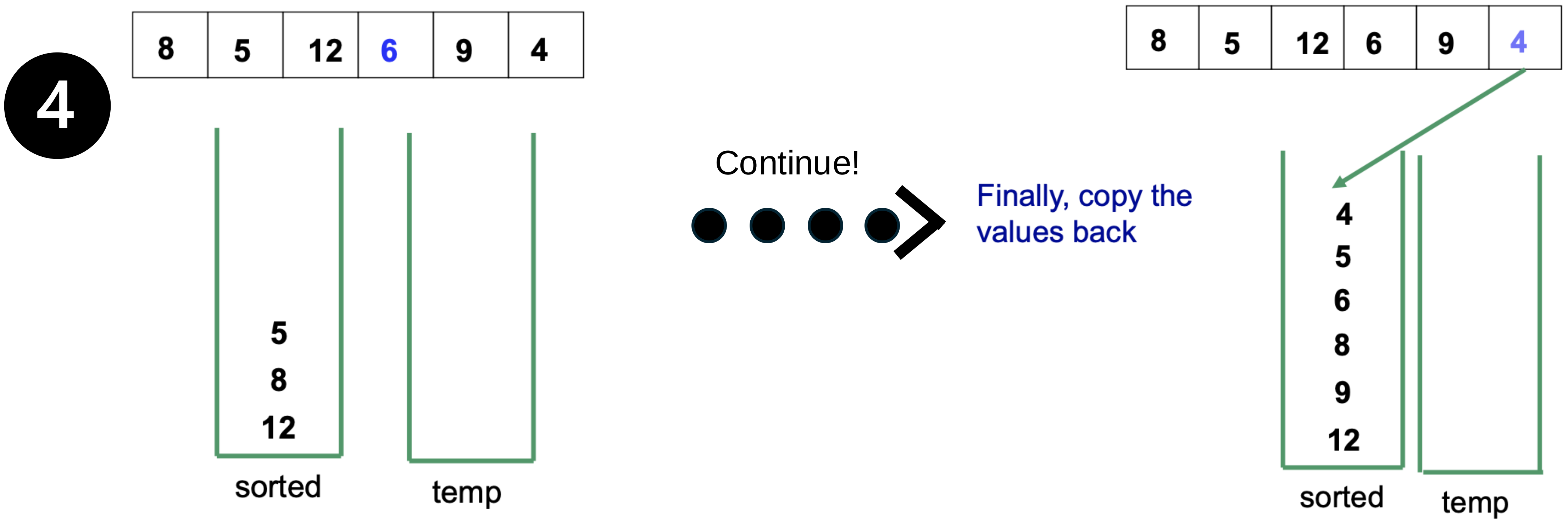
Insertion Sort Using Stacks

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



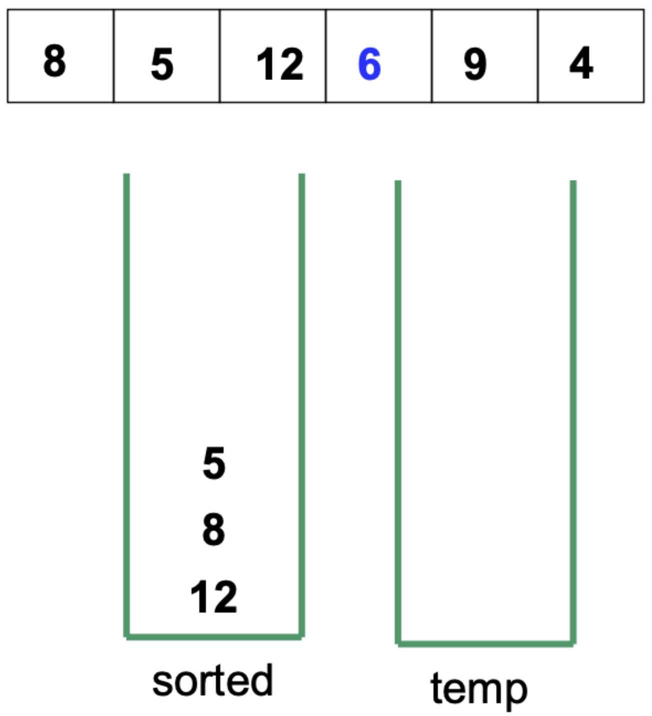


Insertion Sort Using Stacks



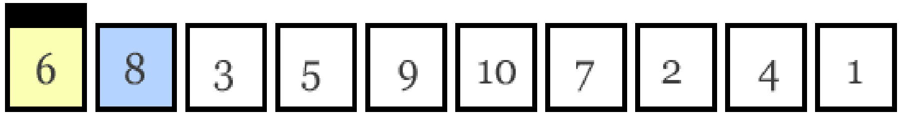
Insertion Sort using Stacks

```
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



Yellow is smallest number found
Blue is current item
Green is sorted list

In-Place Selection Sort

```
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;}
}
```


Thank you