R-2.8: Selection Sort on Sequence (22, 15, 26, 44, 10, 3, 9, 13, 29, 25)

Selection Sort repeatedly selects the minimum element from the unsorted part of the list and swaps it with the first unsorted element.

Steps of Selection Sort:

Initial Array: (22, 15, 26, 44, 10, 3, 9, 13, 29, 25)

Step 1: Find the minimum element (3), swap with the first element (22).

Result: (3, 15, 26, 44, 10, 22, 9, 13, 29, 25)

Step 2: Find the minimum in the rest (9), swap with the second element (15).

Result: (3, 9, 26, 44, 10, 22, 15, 13, 29, 25)

Step 3: Find the minimum in the rest (10), swap with the third element (26).

Result: (3, 9, 10, 44, 26, 22, 15, 13, 29, 25)

Step 4: Find the minimum in the rest (13), swap with the fourth element (44).

Result: (3, 9, 10, 13, 26, 22, 15, 44, 29, 25)

Step 5: Find the minimum in the rest (15), swap with the fifth element (26).

Result: (3, 9, 10, 13, 15, 22, 26, 44, 29, 25)

Step 6: Find the minimum in the rest (22), no swap needed.

Result: (3, 9, 10, 13, 15, 22, 26, 44, 29, 25)

Step 7: Find the minimum in the rest (25), swap with (44).

Result: (3, 9, 10, 13, 15, 22, 25, 26, 29, 44)

Step 8: No swap needed for the rest.

Final Sorted Array: (3, 9, 10, 13, 15, 22, 25, 26, 29, 44) R-2.9: Insertion Sort on Sequence (22, 15, 26, 44, 10, 3, 9, 13, 29, 25)

Insertion Sort builds the sorted array one element at a time by repeatedly inserting the next element into its correct position in the sorted part.

Steps of Insertion Sort:

Initial Array: (22, 15, 26, 44, 10, 3, 9, 13, 29, 25)

Step 1: Insert 15 in the sorted part (22).

Result: (15, 22, 26, 44, 10, 3, 9, 13, 29, 25)

Step 2: Insert 26 (already in place).

Result: (15, 22, 26, 44, 10, 3, 9, 13, 29, 25)

Step 3: Insert 44 (already in place).

Result: (15, 22, 26, 44, 10, 3, 9, 13, 29, 25)

Step 4: Insert 10 before 15.

Result: (10, 15, 22, 26, 44, 3, 9, 13, 29, 25)

Step 5: Insert 3 before 10.

Result: (3, 10, 15, 22, 26, 44, 9, 13, 29, 25)

Step 6: Insert 9 before 10.

Result: (3, 9, 10, 15, 22, 26, 44, 13, 29, 25)

Step 7: Insert 13 before 15.

Result: (3, 9, 10, 13, 15, 22, 26, 44, 29, 25)

Step 8: Insert 29 before 44.

Result: (3, 9, 10, 13, 15, 22, 26, 29, 44, 25)

Step 9: Insert 25 before 29.

Result: (3, 9, 10, 13, 15, 22, 25, 26, 29, 44)

Final Sorted Array: (3, 9, 10, 13, 15, 22, 25, 26, 29, 44)

R-2.10: Worst-Case Sequence for Insertion Sort ($\Omega(n^2)$ Time)

The worst-case sequence for Insertion Sort occurs when the array is sorted in reverse order. In this case, each element needs to be compared with all the previous elements to find its correct position, resulting in the worst-case time complexity of $O(n^2)$.

Example of Worst-Case Sequence:

For n = 5, the worst-case sequence would be:

(5, 4, 3, 2, 1)

In this case, every insertion will require comparisons with all preceding elements, leading to $O(n^2)$ time.

R-2.13

No, the tree T is not necessarily a heap. A heap must satisfy two properties:

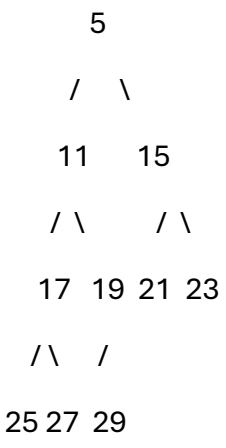
Complete Binary Tree: The tree must be complete (i.e., all levels except possibly the last are fully filled, and the last level is filled from left to right).

Heap Property: For a min-heap, every parent node must be smaller than or equal to its children. For a max-heap, every parent node must be greater than or equal to its children.

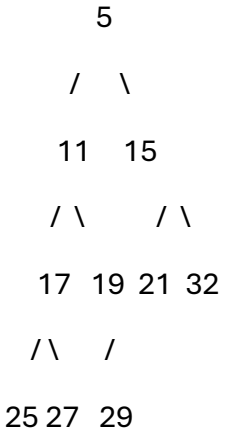
If the items are simply stored in sorted order in the vector, this does not guarantee the heap property, as the relationships between parent and child nodes may not meet the required heap conditions.

R-2-18

We need to draw a heap with keys that are all odd numbers from 1 to 49, such that inserting 32 would cause **up-heap bubbling** to proceed all the way up to a child of the root.



Now, inserting 32 causes the heap to maintain its properties by comparing 32 with its parent. Since 32 is greater than all its parents, it would replace a value near the root. After bubbling up, the final heap might look like:



C-2.32

Start at the root of the heap.

For each node, if the key is smaller than or equal to x, add it to the result and recursively check its children. Stop traversing when a key is larger than x because, in a heap, all children of that node will also be larger.

Algorithm reportKeys(T, x):

result = []

```
  reportHelper(T, T.root(), x, result)
  return result

Algorithm reportHelper(T, p, x, result):
  if T.element(p) ≤ x:
    result.append(T.element(p))
    if T.leftChild(p) != null:
      reportHelper(T, T.leftChild(p), x, result)
    if T.rightChild(p) != null:
      reportHelper(T, T.rightChild(p), x, result)
```

Time Complexity: The time complexity is O(k), where k is the number of keys reported. The algorithm only explores relevant nodes and avoids unnecessary traversal.