Assignment 4

R-2.8 Illustrate the performance of the selection-sort algorithm on the following input sequence (22, 15, 26, 44, 10, 3, 9, 13, 29, 25).

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 22 | 15 | 26 | 44 | 10 | 3 | 9 | 13 | 29 | 25 |
| 3 | 9 | 26 | 44 | 10 | 22 | 15 | 13 | 29 | 25 |
| 3 | 9 | 10 | 44 | 26 | 22 | 15 | 13 | 29 | 25 |
| 3 | 9 | 10 | 13 | 26 | 22 | 15 | 44 | 29 | 25 |
| 3 | 9 | 10 | 13 | 15 | 22 | 26 | 44 | 29 | 25 |
| 3 | 9 | 10 | 13 | 15 | 22 | 26 | 44 | 29 | 25 |
| 3 | 9 | 10 | 13 | 15 | 22 | 25 | 44 | 29 | 26 |
| 3 | 9 | 10 | 13 | 15 | 22 | 25 | 26 | 29 | 44 |
| 3 | 9 | 10 | 13 | 15 | 22 | 25 | 26 | 29 | 44 |
| 3 | 9 | 10 | 13 | 15 | 22 | 25 | 26 | 29 | 44 |

R-2.9 Illustrate the performance of the insertion-sort algorithm on the input sequence of the previous problem.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 22 | 15 | 26 | 44 | 10 | 3 | 9 | 13 | 29 | 25 |
| 15 | 22 | 26 | 44 | 10 | 3 | 9 | 13 | 29 | 25 |
| 15 | 22 | 26 | 44 | 10 | 3 | 9 | 13 | 29 | 25 |
| 15 | 22 | 26 | 44 | 10 | 3 | 9 | 13 | 29 | 25 |
| 10 | 15 | 22 | 26 | 44 | 3 | 9 | 13 | 29 | 25 |
| 3 | 10 | 15 | 22 | 26 | 44 | 9 | 13 | 29 | 25 |
| 3 | 9 | 10 | 15 | 22 | 26 | 44 | 13 | 29 | 25 |
| 3 | 9 | 10 | 13 | 15 | 22 | 26 | 44 | 29 | 25 |
| 3 | 9 | 10 | 13 | 15 | 22 | 26 | 29 | 44 | 25 |
| 3 | 9 | 10 | 13 | 15 | 22 | 25 | 26 | 29 | 44 |

R-2.10 Give an example of a worst-case sequence with n elements for insertion-sort runs in Ω(n2) time on such a sequence.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 44 | 29 | 26 | 25 | 22 | 15 | 13 | 10 | 9 | 3 |

R-2.13 Suppose a binary tree T is implemented using a vector S, as described in Section 2.3.4. If n items are stored in S in sorted order, starting with index 1, is the tree T a heap? Justify your answer.

**Answer:**

**Yes it is a heap, because S is in Sorted order.**

R-2-18 Draw an example of a heap whose keys are all the odd numbers from 1 to 49 (with no repeats), such that the insertion of an item with key 32 would cause up-heap bubbling to proceed all the way up to a child of the root (replacing that child’s key with 32).

Answer:

A picture containing line, diagram, pattern

Description automatically generated

C-2.32 Let T be a heap storing n keys. Give an efficient algorithm for reporting all the keys in T that are smaller than or equal to a given query key x (which is not necessarily in T). For example, given the heap on Figure 2.41 and query key x=7, the algorithm should report 4, 5, 6, 7. Note that the keys do not need to be reported in sorted order. Ideally, your algorithm should run in O(k) time, where k is the number of keys reported.

|  |  |
| --- | --- |
| **Algorithm findSmaller(T, x)**  **s := new Sequence  findHelper(T, x, T.root(), S)  return S Algorithm findHelper(T, x, p, S)  if T.isExternal(p) then**  **return  if p.element() > x then**  **return**  **S.insertLast(p.element())  findHelper(T, x, T.leftChild(p), S)  findHelper(T, x, T.rightChild(p), S)** | **O(1)**  **O(logn)**  **O(1)**  **O(1)**  **O(1)**  **O(1)**  **O(1)**  **O(1)**  **O(logn)**  **O(logn)**  **Total running time is O(logn)** |
| **Algorithm reporting(T, k)**  **if T.isEmpty() then return []**  **arr = []**  **non\_stop = True**  **while non\_stop do**  **r = T.removeFirst()**  **if r < k then**  **arr.add(r)**  **else:**  **non\_stop = False**  **return arr** | **O(1)**  **O(1)**  **O(1)**  **O(k)**  **O(k)**  **O(k)**  **O(k)**  **O(1)**  **Total running time is O(k)** |

Design an algorithm, isPermutation(A,B) that takes two sequences A and B and determines whether or not they are permutations of each other, i.e., same elements but possibly occurring in a different order. Hint: A and B may contain duplicates. What is the worst case time complexity of your algorithm? Justify your answer.

|  |  |
| --- | --- |
| **Algorithm isPermutation(A, B)**  **if A.size() != B.size() then**  **return false**    **H1 <- heapSort(A)**  **H2 <- heapSort(B)**    **n <- A.size()**  **for i<-0 to n-1 do**  **p1 <- H1.removeFirst()**  **p2 <- H2.removeFirst(s)**  **if p1.element() != p2.element() then**  **return false**    **return true** | **O(1)**  **O(nlogn)**  **O(nlogn)**  **O(1)**  **O(n)**  **O(n)**  **O(n)**  **O(n)**  **O(1)**  **Total time is O(nlogn)** |