Solutions for Data Structures and Algorithms Spring 2023 — Problem Sets

By Dmitriy Okoneshnikov, B22-DSAI-04

March 29, 2023

Week 9. Homework 2

3.1 Dynamic Programming (20 points)

Two cities A and B are connected by a highway. Your task is to place shopping centers along the highway in an efficient way, maximizing estimated revenue. Possible locations of shopping centers along the highway are given by N numbers $x_1, x_2, ..., x_N$. Each number specifies the number of kilometers from city A to that location on a highway. For each position x_i we also know in advance the estimated revenue r_i that we get if we place a shopping center there. There is a restriction that no two shopping centers can be placed within d kilometers of each other. The highway length is H kilometers and we have $0 < x_1 < x_2 < ... < x_n < H$.

For example, if we have 4 locations $x_1 = 5, x_2 = 10, x_3 = 14, x_4 = 15$ with estimated revenue $r_1 = 80, r_2 = 150, r_3 = 60, r_4 = 75$ and $d = 6$ then the maximum revenue is 155 when we place two shopping center at locations $x_1 = 80, x_4 = 15$.

Describe a general algorithm for any number N of available locations, any length of the highway H:

1. Write down pseudocode for a recursive algorithm that solves the problem.

   Answer.

2. Provide asymptotic worst-case time complexity of the recursive algorithm

   Answer. $O(2^n)$

3. Identify overlapping subproblems.

   Answer.

4. Write down pseudocode for the optimized algorithm that solves the problem using dynamic programming (top-down or bottom-up). The algorithm should compute both the maximum estimated revenue **and** the specific locations where shopping centers should be placed.

   Answer.

5. Provide asymptotic worst-case time complexity of the dynamic programming algorithm.

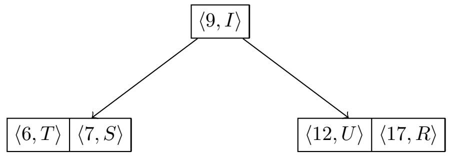
   Answer. $O(H)$

3.2 B-Trees (15 points)

Insert the following $\langle key, value\rangle$ items into an initially empty B-tree [Cormen, Chapter 18] with minimum degree $t = 2$. Show the state of the tree after every 5 insertions (your answer must contain exactly 4 trees):

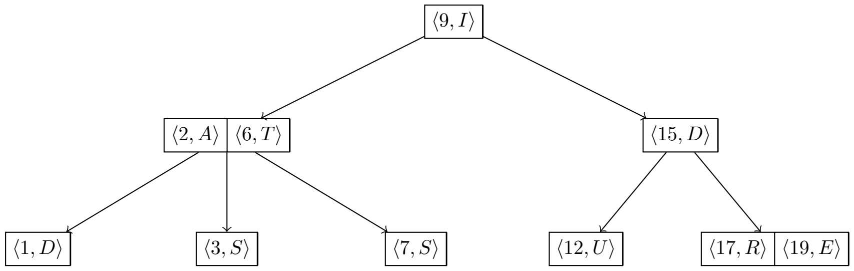
1. $\langle 17, R\rangle, \langle 6, T\rangle, \langle 9, I\rangle, \langle 7, S\rangle, \langle 12, U\rangle$

   Answer.



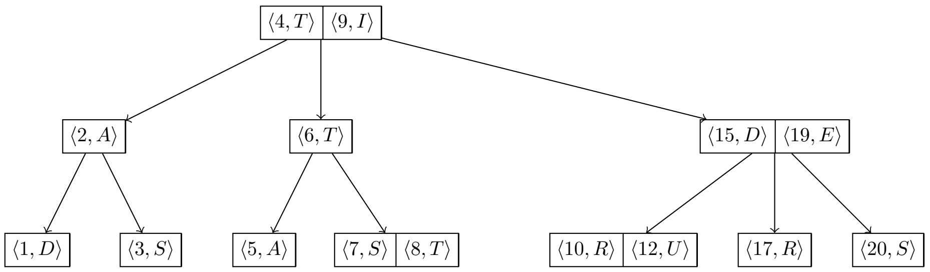
2. $\langle 1, D\rangle, \langle 15, D\rangle, \langle 19, E\rangle, \langle 2, A\rangle, \langle 3, S\rangle$

   Answer.



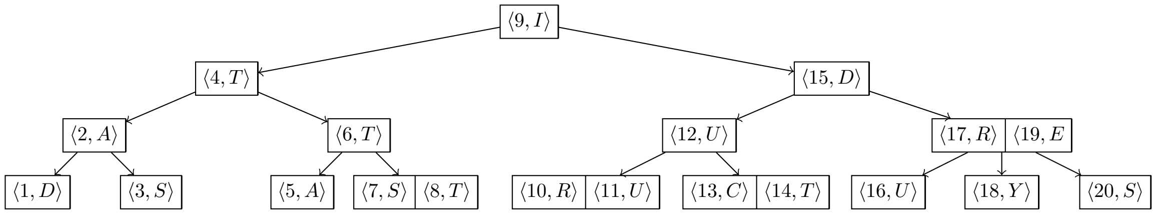
3. $\langle 10, R\rangle, \langle 8, T\rangle, \langle 4, T\rangle, \langle 5, A\rangle, \langle 20, S\rangle$

   Answer.



4. $\langle 14, T\rangle, \langle 18, Y\rangle, \langle 11, U\rangle, \langle 13, C\rangle, \langle 16, U\rangle$

   Answer.



3.3 Binary Search Trees with Equal Keys (15 points)

Equal keys pose a problem for the implementation of Binary Search Trees. We propose to improve insertion into a Binary Search Tree by testing for each node whether inserted key is equal to the key stored in the inspected nodes. If equality holds, then we employ one of the following strategies.

For each strategy, find the asymptotic time complexity of inserting n items with identical keys into an initially empty binary search tree. Justify your answer for each strategy.

1. No special strategy, use regular insertion algorithm.

   Solution.

   With no specials strategies it would be just an ordinary insertion. Every time when a new element gets inserted we would always go to one side of a tree. At the end we would have a singly linked list, so insertion of n elements would have time complexity of $O(n \cdot n) = O(n^2)$.

   Answer. $O(n^2)$

2. Keep a boolean flag at every node. This flag determines whether to go left or right when inserting an item with the key equal to the one in the node. Every time we visit the node with the key equal to node's key, we switch the flag.

   Solution.

   This strategy allows our tree to balance itself, therefore, the height of this tree is $O(\log n)$. The time complexity is $O(n \log n)$.

   Answer. $O(n \log n)$

3. Keep a list of values with equal keys in each node. Insert the item into that list.

   Solution.

   This will result our tree to have only one node with a list that will have n elements in the end. Insertion into a list can be done in $O(1)$, therefore, the time complexity would be $O(n \cdot 1) = O(n)$

   Answer. $O(n)$

4. Randomly choose either left of right. For this strategy, show both the worst-case time complexity, and informally derive the expected time complexity.

   Solution.

   In the worst case every element will be inserted into one of the side of the tree, so it will be the same as point 1: $O(n^2)$.

   The expected time complexity would be $O(n \log n)$ as it is the same as point 2.

   Answer. $O(n^2)$ and $O(n \log n)$

3.4 d-ary heaps (+2% extra credit)

A d-ary heap is similar to a binary heap, except non-leaf nodes have d children instead of 2 children (except the last non-leaf node, which is allowed to have fewer children).

1. How should a d-ary heap be represented in an array?

   Answer.

   It should be represented as an array of length n, where the root is at index 0 and the children of node at index i are located at $d \cdot i + 1, d \cdot i + 2, ..., d \cdot i + d$. The parent of node at index i is located at $\lfloor \frac{i-1}{d} \rfloor$.

2. What is the height of a d-ary heap with n elements in terms of n and d?

   Answer.

   $h = \lfloor \log_d n \rfloor + 1$

3. Give an efficient implementation of `extractMax` procedure in a d-ary max heap. Analyze the worst-case running time in terms of n and d.

   Solution.

```
heapify($H$, $i$) // $H[1:N]$ is the $d$-ary heap
    $max$ := $i$
    for $k$ := 1 to $H.d$:
        $childInd$ := $H.d$ * ($k$ - 1) + $i$ + 1
        if $childInd$ ≤ $H$.size and $H[childInd]$ > $H[i]$:
            if $H[childInd]$ > $H[max]$:
                $max$ := $childInd$
    if $max$ != $i$:
        swap($H[max]$, $H[i]$)
        heapify($H$, $max$)

extractMax($H$) // $H[1:N]$ is the $d$-ary heap
    $max$ := $H[1]$
    $H[1]$ := $H[H.\text{size}]$
    decrease size of $H$
    heapify($H$, 1)
    return $max$
```

First we get the root element [$O(1)$], which is the maximum value in max heap. Then we replace the root and the last element in the heap [$O(1)$]. We should decrease the size of the heap [$O(1)$ amortized], then run `heapify` [$O(\log_d n)$] on the new root node which is called recursively d times, therefore, the worst-case time complexity will be $O(d \log_d n)$.

4. Give an efficient implementation of `insert` procedure in a d-ary max heap. Analyze the worst-case running time in terms of n and d.

Solution.

```
parent($H$, $i$)
    return floor(($i$ - 1) / $H.d$)

insert($H$, $e$) // $H[1:N]$ is the $d$-ary heap
    increase size of $H$
    $H[H.\text{size}]$ := $e$
    $i$ := A.size
    while $i$ > 1 and $H[\text{parent}(H, i)]$ < $H[i]$:
        swap($H[i]$, $H[\text{parent}(H, i)]$)
        $i$ := parent($i$)
```

First we increase the size of array [$O(1)$ amortized]. Then we place the new element at the end of the heap [$O(1)$]. After that we move our element up which will take at worst $O(\log_d n)$, therefore, the worst-case time complexity will be $O(\log_d n)$.

5. Give an efficient implementation of `increaseKey` procedure in a d-ary max heap. Analyze the worst-case running time in terms of n and d.

Solution.

```
parent($H$, $i$)
    return floor(($i$ - 1) / $H.d$)

increaseKey($H$, $e$, $pos$) // $H[1:N]$ is the $d$-ary heap
    if $e$ < $H[pos]$:
        return
    $H[pos]$ = $e$
    $i$ := $pos$
    while $i > 1$ and $H$[parent($H$, $i$)] < $H$[$i$]:
        swap($H$[$i$], $H$[parent($H$, $i$)])
        $i$ := parent($i$)
```

First we place the new element $[O(1)]$. Then we move our element up which will take at worst $O(\log_d n)$, therefore, the worst-case time complexity will be $O(\log_d n)$.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms, Fourth Edition.* The MIT Press 2022.

[2] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser. *Data Structures and Algorithms in Java.* WILEY 2014.