Homework Batch II: Trees and Algorithms

Tibor Racman

April 29, 2021

1. All the heaps are implemented using arrays.

   (a)
```
procedure RetriveMax($H$)
    $max \leftarrow -\infty$
    if $H = NIL$ then
        return $NIL$
    else
        for $i \leftarrow \lceil \frac{n}{2} \rceil$ to $n$ do
            if $H[i] > max$ then
                $max \leftarrow H[i]$
    return $max$
```

   For the heap property we have that the maximum value will be for sure in one of the leafs. Since the number of leafs is half of the nodes, they will be all in the second half of the underlying array structure. In order to find the maximum value we have to scan through this part, which will take time $\Theta(\lceil \frac{n}{2} \rceil) \simeq \Theta(n)$.

   (b)
```
procedure DeleteMax($H$)
    $max \leftarrow$RetriveMax($H$)
    if $H[n] = max$ then
        $H.size \leftarrow H.size - 1$
        return
    else
        for $i \leftarrow \lceil \frac{n}{2} \rceil$ to $n$ do
            if $H[i] = max$ then
                DecreaseKey($H, i, H[i] - H[n]$)
                $H.size \leftarrow H.size - 1$
                return
```

   If the maximum value is the rightest leaf then removing that node takes time $\Theta(1)$, otherwise we will first decrease the maximum value to the rightest value, using *DecreaseKey* which has complexity of $O(\log(n))$ and then we will proceed to remove the rightest node. However in order to find the maximum value we have to call *RetriveMax* which is $\Theta(n)$ and dominates the other complexities. The overall complexity is

$$\Theta(n) + \Theta(1) + O(\log(n)) \simeq \Theta(n)$$

   Notice that *DecreaseKey* is not called for each element of the $foor$ loop, but just for the first satisfying the maximum condition, avoiding a possible complexity of $O(n \log(n))$.

(c) As explained in b there is no worst scenario from an asymptotic point of view, since $RetriveMax$ has to be called in any case.
However, in practice (i.e execution time) the "worst case" occurs when the maximum is not in the rightest position and $DecreaseKey$ will be called, which will subsequentially call $Heapify$ in order to maintain the binary tree structure when decreasing the maximum node. The process is shown below on tree with 8 nodes:

i.

1	2	3	4	5	6	7	8
1	4	8	5	9	11	10	6

ii.

1	2	3	4	5	6	7	8
1	4	8	5	9	11	10	6

iii.

1	2	3	4	5	6	7	8
1	4	8	5	9	6	10	6

iv.

1	2	3	4	5	6	7	8
1	4	6	5	9	8	10	6

v.

1	2	3	4	5	6	7	
1	4	6	5	9	8	10	

2. (a) B =

4	0	5	3	0	0	2	0	1	0

(b) By rewriting the definition in pseudocode terms we obtain:

```
procedure SmallerCounterNaive($A$)
    for $i \leftarrow 1$ to $|A|$ do
        for $z \leftarrow i + 1$ to $|A|$ do
            if $A[z] < A[z]$ then
                $B[i] + +$
    return $B$
```

which has complexity of $O(n^2)$ since we have two nested for loops on an operation $\Theta(1)$. The correctness is guaranteed by the fact that we just applied the definition.

(c) In this case we could use a simple augmented Binary Search Tree in which each node has two extra fields:

- the number of elements on the left side of the node
- the frequency of the node itself

We start building the node inserting each element of A starting from the right. Once the value is inserted we can compute the relative B value by scanning the tree looking for the element we have just inserted: when we go right we add $1 + \mathit{node.counter} + \mathit{node.frequency}$ to keep in consideration all the nodes that are smaller then the one that we have just passed. Moreover storing the frequency of each node, helps us to decrease the node height in our case since we will have one node representing many instances of the same value (i.e 0). The procedure is the following:

```
procedure countSmaller($A$)
    $B \leftarrow$allocateArray($|A|, defaultValue = 0$)
    $T \leftarrow$BST()
    for $i \leftarrow |A|$ to 1 do
        $x \leftarrow T$.insertAugmentedBST($A[i]$)
        $B[i] \leftarrow T$.searchAugmentedBST($A[i]$)
    return $B$
```

where

```
procedure insertAugmentedBST($v$)
    $x \leftarrow root$
    $y \leftarrow NIL$
    while $x \neq NIL$ do
        $y \leftarrow x$
        if $v \leq x.key$ then
            if $x.key \leq v$ then
                $x.frequency++$
                return
            else
                $x.count++$
                $x \leftarrow x.left$
        else
            $x \leftarrow x.right$
    $x \leftarrow$newNode($v$)
    if $root \neq NIL$ then
        if $v \leq y.key$ then
            setChild($y, left, x$)
        else
            setChild($y, right, x$)
    else
        $root \leftarrow x$
```

```
procedure searchAugmentedBST($A[i]$)
    $tmp \leftarrow 0$
    $x \leftarrow root$
    while $x \neq NIL$ do
        if $x.key \leq v$ then
            if $v \leq x.key$ then
                return tmp
            else
                $tmp \leftarrow x.count + x.frequency + 1$
                $x \leftarrow x.right$
        else
            $x \leftarrow x.left$
    return $tmp$
```

```
procedure newNode($v$)
    $x.key \leftarrow v$
    $x.frequency \leftarrow 1$
    $x.count \leftarrow 0$
    $x.parent \leftarrow NIL$
    $x.right \leftarrow NIL$
    $x.left \leftarrow NIL$
```

Since we have used just one node for all duplicated values (with the help on the field $frequency$) we have that in the worst case (i.e when the Binary Search Tree is unbalanced) we will have $O(nk)$, while in the average case we will have $\Theta(nlog(k))$. The correctness is given by the Binary Search Tree structure that puts the smaller nodes on the left and the greater ones to the right.

3. (a) The Red Black Tree is a Binary Search Tree which satisfies the following conditions:
    - each node is either a RED or a BLACK node
    - the tree's root is BLACK
    - all the leaves are BLACK NIL nodes
    - all the RED nodes must have BLACK children
    - for each node x, all the branches from x contain the same number of black nodes

(b)

```
procedure treeHeight($T$)
    return treeHeightAt($T.root$)
```

where:

```
procedure treeHeightAt($n$)
    if $n = NIL$ then
        return 0
    else
        return max(treeHeightAt($n.left$), treeHeightAt($n.right$)) + 1;
```

This visits every node exactly once and does $\Theta(1)$ work per node, so the total time complexity is $\Theta(n)$. The correctness is given by the definition of height of a tree, the distance between the root and the most far node, which is exactly what the procedure is computing.

(c)

```
procedure treeBlackHeight($T$)
    $counter \leftarrow 0$
    $n \leftarrow T.root$
    while $n \neq NIL$ do
        if $n = BLACK$ then
            counter++
        $n \leftarrow n.left$
    return counter
```

The maximum height of a Red Black Tree is $2\log(n+1)$ and consequently the algorithm is $\Theta(2\log(n+1)) \simeq \Theta(\log(n))$ because it needs to explore just one random path (in our case the left one) since all of them, will produce the same black height. This fact give us also the correctness of the procedure.

4. (a) In order to lexicographical sort the pairs we will use the same conceptual approach as we used in Radix sort. First we will sort the pairs in respect to the second element and then we will sort them according to the first element. In order to satisfy the second condition of the lexical order, when sorting through the first elements (i.e second sorting) we have to use a stable sorting algorithm. An opportune data structure could be an array of pairs and a possible procedure could be:

```
procedure lexicographicalSort($A$)
    if $A \neq NIL$ then
        heapSortSecondElement($A$)
        insertionSortFirstElement($A$)
```

where:

```
procedure insertionSortFirstElement($A$)
    for $i \leftarrow 2$ to $|A|$ do
        $j \leftarrow i$
        while $j > 1 \wedge A[j].first < A[j-1].first$ do
            swap($A, j-1, j$)
            $j \leftarrow j - 1$
```

and

```
procedure heapSortSecondElement($A$)
    $H \leftarrow$buildMaxHeapPairs($A$)
    for $i \leftarrow |A|$ to 2 do
        $A[i] \leftarrow$extractMinSecondElement($H$)
```

and buildMaxHeapPairs(A) build a max heap, where each node is a pair of integers while extractMinSecondElement(H) extracts the minimum of these nodes comparing them, using the second integer.

The first sort takes time $O(n\log(n)$, however the second sort takes time $O(n^2)$ and the overall complexity is

$$O(n\log(n)) + O(n^2) \simeq O(n^2)$$

(b) With this assumption we could apply a faster but still stable algorithm on the first elements such as counting sort (i.e countingSortFirstElement(A, B, k)). The overall complexity in this case would be:

$$O(n\log(n)) + O(n+k) \simeq O(n\log(n))$$

```
procedure lexicographicalSort($A$)
    if $A \neq NIL$ then
        $B \leftarrow$allocateArray($k, defaultValue = 0$)
        heapSortSecondElement($A$)
        countingSortFirstElement($A, B, k$)
```

where

```
procedure countingSortFirstElement($A, B, k$)
    $C \leftarrow$allocateArray($k, defaultValue = 0$)
    for $i \leftarrow 1$ to $|A|$ do
```

```
        $C[A[i].first] \leftarrow C[A[i].first] + 1$
    **for** $j \leftarrow 2$ to $|C|$ **do**
        $C[j] \leftarrow C[j-1] + C[j]$
    **for** $i \leftarrow |A|$ to $1$ **do**
        $B[C[A[i].first]] \leftarrow A[i]$
        $C[A[i].first] \leftarrow C[A[i].first] - 1$
```

(c) With the previous assumption being satisfied also by the second elements of the pairs we could use counting sort twice which would result in

$$O(n+h) + O(n+k) \simeq O(n+k+h)$$

```
**procedure** LEXICOGRAPHICALSORT($A$)
    **if** $A \neq NIL$ **then**
        $B_1 \leftarrow$allocateArray($h, defaultValue = 0$)
        $B_2 \leftarrow$allocateArray($k, defaultValue = 0$)
        countingSortSecondElement($A, B_1, h$)
        countingSortFirstElement($A, B_2, k$)
```

where

```
**procedure** COUNTINGSORTSECONDELEMENT($A, B, k$)
    $C \leftarrow$allocateArray($k, defaultValue = 0$)
    **for** $i \leftarrow 1$ to $|A|$ **do**
        $C[A[i].second] \leftarrow C[A[i].second] + 1$
    **for** $j \leftarrow 2$ to $|C|$ **do**
        $C[j] \leftarrow C[j-1] + C[j]$
    **for** $i \leftarrow |A|$ to $1$ **do**
        $B[C[A[i].second]] \leftarrow A[i]$
        $C[A[i].second] \leftarrow C[A[i].second] - 1$
```

5. (a) The assumption is not strictly necessary as long as it concerns the algorithm's output, however it might affect its complexity. Let us pass an array A of length 11, in which all the values are the same. In this case even if we pick the "median of medians", the element in the middle, the sub routine partition would give us always an empty G and a full S (i.e A.length - 1) partitions. In our implementation this would result in calling partition $n - i$ times, where i is the input index and this would result in $O(n(n-i)) \simeq O(n^2)$.

(b) We could avoid the previous scenario by generalizing the $Partition(A, lower, higher, pivot)$ procedure. Its output would now result in:

- a group S of elements smaller than the *pivot*
- a group E of elements equal to the *pivot*
- a group G of elements greater than the *pivot*

and then we should check if the index i, given to the select algorithm, belongs to E, S or G and in case reiterate. A possible implementation could be:

```
procedure generalizedMedianOfMedians($A, l = 1, r = |A|, i$)
    if $r - l \leq 10$ then
        sort($A, l, r$)
        return i
    $pivot$ ←selectPivot($A, l, r$)
    $(k, m)$ ←threePartition($A, l, r, pivot$)
    if $i \geq k \wedge i \leq m$ then
        return k
    else if $i < k$ then
        return generalizedMedianOfMedians($A, l, k - 1, i$)
    else
        return generalizedMedianOfMedians($A, m + 1, r, i$)
```

while the threePartition (also known as the *Dutch national flag problem*) could be solved as follows:

```
procedure threePartition($A, lower, higher, pivot$)
    $l \leftarrow lower$
    $r \leftarrow lower$
    $u \leftarrow higher$
    while $r \leq u$ do
        if $A[r] < pivot$ then
            swap($A[l], A[r]$)
            $l \leftarrow l + 1$
            $r \leftarrow r + 1$
        else if $A[r] > pivot$ then
            swap($A[r], A[u]$)
            $u \leftarrow u + 1$
        else
            $r \leftarrow r + 1$
    return $(l, r)$
```

In this case, since threePartition($A, lower, higher, pivot$) still belongs to $\in O(n)$ and since the generalized version of select performs just few more $\Theta(1)$ operations (to check if k belongs to E) compared to the default one, we can conclude that there are no variations in complexity and that the algorithm remains in $O(n)$.