Homework Batch I: Trees and Algorithms

Liberatori Benedetta

April 2021

1

Let H be a `Min-Heap` containing n integer keys.

a) A `Min-Heap` stores totally ordered values w.r.t. $\leq$. Then by construction the maximum is stored in a leaf node, being it $\geq$ than all the ancestors. In the following, the array-based representation used during lesson will be used. Then it can be exploit that the leaves nodes correspond to the elements from index $\lfloor \frac{n}{2} \rfloor + 1$ up to the last index n and an in-place procedure to retrieve the maximum consists in a scan of this sub-array.

```
def RetrieveMax(H):

    n ← H.size
    k ← H[⌊n/2⌋ + 1]

    for  i in ⌊n/2⌋ +2, ... n:

        if   H[i] > k :
            k ← H[i]
        endif
    endfor
return  k
enddef
```

The for-loop iterates on a sub-array of A which has a dimension depending on n, and the interior operations takes constant time, then the asymptotic complexity is given by:

$$T(n) = \Theta(1) + \sum_{i=\lfloor \frac{n}{2} \rfloor + 2}^{n} \Theta(1) = \Theta(n)$$

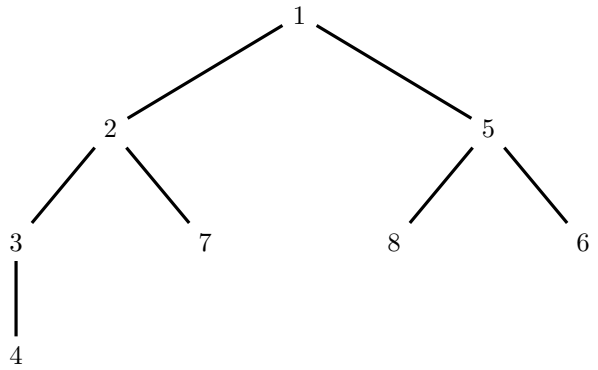
b) Once the maximum has been found, the deletion requires additional care in order to maintain the two key properties of the data structure. The first part of the algorithm is the same procedure explained in point a), a part from

the fact that the index of the maximum is needed, rather than the value. The maximum can be deleted without changing the topology of the heap in $\Theta(1)$: putting the rightmost leaf of the last level, (last element in the array with the array representation) in place of the maximum and decreasing the size of the data structure. After this step the heap property may have been lost. The new node in place of the maximum could be smaller than its new parent. If this is the case it can be fixed by exchanging the twos. Then the problem could have been pushed one level up, so the procedure must be repeated.

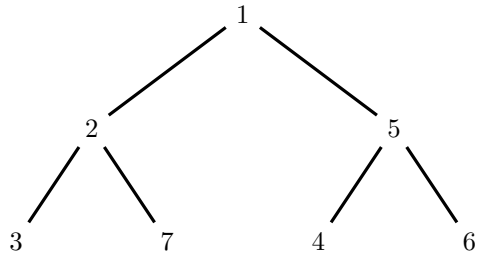
```
def DeleteMax(H):

    n ← H.size
    k ← H[⌊n/2⌋ + 1]
    for   i in ⌊n/2⌋ +2, ... n:

        if    H[i] > k:
            k ← H[i]
            j ← i
        endif
    endfor

    H[j] ← H[n]
    H.size ← H.size - 1

    while (H[Parent(i)] > H[i]) :
            swap(H, i, Parent(i))
    endwhile
return
enddef
```

Where the function Parent(i) returns the index $\lfloor \frac{i}{2} \rfloor$ which is the one of the parent and swap(H, i, j) swaps the elements $H[i]$ and $H[j]$. The first step of retrieving the index of the maximum has cost $\Theta(n)$ as already pointed out in a). The substitution with the rightmost element and the decrease of the size takes $\Theta(1)$. The while-loop is performed in the worst case for each node in the path from the parent of the maximum to a child of the root (being the root for sure $\leq$ than the new node in place of the maximum). The swap operation takes constant time, then the while takes $O(logn)$, due to the nearly completeness property. Thus the overall the complexity is still $\Theta(n)$.

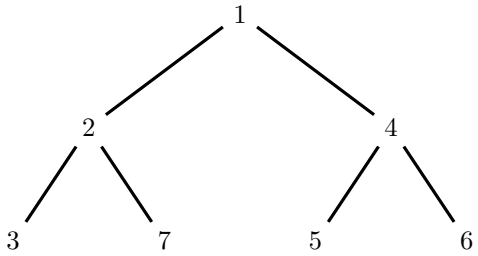
c) Worst case scenario for the DeleteMax(H) with $n = 8$:
The worst case is then a case in which after the replacement the heap property has to be restored. Here is an example:



The maximum 8 will be replaced by 4, which happens to be smaller than 5.



Then 4 and 5 must be swapped and this restore the Min-Heap, since the root is already smaller than its new child (being it a descendant from the start).



2

a) A = $[2, -7, 8, 3, -5, -5, 9, 1, 12, 4] \Rightarrow B = [4, 0, 5, 3, 0, 0, 2, 0, 1, 0]$

b) A solution for the problem with quadratic complexity consists in scanning, for each element in the array, the remaining elements on its right:

```
def NaiveSmallerOnRight(A):
    for i in 1,...n:
```

```
        B[i] ← 0
    endfor

    for i in 1,..n:
        for j in i+1,..n:
            if A[j] < A[i] :
                B[i] ← B[i] + 1
            endif
        endfor
    endfor
    return B
enddef
```

The initialization of B costs $\Theta(n)$, the nested for-loop costs $\Theta(n-i)$ and the operations inside take constant time.

$$T(n) = \Theta(n) + \sum_{i=1}^{n} \Theta(n-i) * \Theta(1) = \Theta(n) + \sum_{j=0}^{n-1} \Theta(j) = \Theta(n) + \Theta(n^2) = \Theta(n^2) = O(n^2)$$

The correctness can be proved by induction on the size of the array. It is trivial for size 1, since the inner-most loop cancels out and then the correspondent array of size 1 B remains 0. Supposed true for size $n-1$, then it follows for size n. Indeed the iteration for $i = 1, ..., n-1$ and $j = i+1, ..., n-1$ compute for the i-th element the correct number of smaller elements which are on the right, up to the second to last element, which will be $B[1, ..., n-1]$. Then for $j = n$ this number is increased by one *iff* the last element in the array is smaller than it. The updated sub-array $B[1, ..., n-1]$ is correct since adding the element $A[n]$ can at most change the previously computed B by adding 1 where the elements are greater than it. Lastly, for $i = n$ the inner-most loop never starts and $B[n] = 0$, which is correct since there are no elements left at its right.

c) With the assumption of a constant number of non-zero values in A, the problem can be solved with linear asymptotic complexity.
Since for the element in the i-th position we are interested in the elements smaller from index $i+1$ on, it makes sense to scan the array from right to left, *taking note* somehow of the elements already met. This can be done using a variable to count the number of zeros already visited (which will add +1 to all the correspondent element in B of all the positive elements on its right), a variable to count the number of negatives already visited (which similarly will add +1 to the correspondent element in B of all the zeros elements on its right) and a dynamic data structure storing the non-zeros elements already visited. The number of elements stored in this data structure will be constant at each step in the procedure. Then even using a Linked List, which has no particular property on the order of the elements stored, counting the number of elements in it smaller than a certain value has a constant asymptotic complexity.

In the following pseudocode, the function SmallerInList(L, v) returns the number of elements smaller than v in the linked list L, simply scanning it and comparing each element with v. Insert(L, v) is a function which inserts v in the list L. Lastly, EmptyList() creates an empty linked list.

```
def SmallerOnRight(A):

    for i in 1,...n:
        B[i] ← 0
    endfor
    L ← EmptyList()
    negatives ← 0
    zeros ← 0

    for i in n,...1
        if A[i] = 0 :
            zeros ← zeros + 1
            B[i] ← negatives
        else if A[i] < 0 :
            negatives ← negatives + 1
            B[i] ← SmallerInList(L, A[i])
            Insert(L, A[i])
        else:
            B[i] ← zeros + SmallerInList(L, A[i])
            Insert(L, A[i])
        endif
    endfor
return B
enddef
```

The initialization of B is linear in n, EmptyList() takes constant time, the for-loop iterates on the elements of A and all the operations inside take $\Theta(1)$ due to the assumption, then:

$$T(n) = \Theta(n) + \sum_{i=1}^{n} \Theta(1) = \Theta(n)$$

In order to prove the correctness three cases can be distinguished. The algorithm scans the array from right to left, thus when the i-th element is visited, then all the elements at its right have already been visited and consequently all the zeros and negatives have been counted. If the i-th element in A is equal to zero, then the only elements already visited which are smaller are the negative ones and $B[i]$ is correctly computed. If the i-th element in A is negative, then it smaller than zero and $B[i]$ is correctly computed as the number of elements smaller than it already in the list. If instead it is positive, $B[i]$ is the number of smaller elements already present in the list, plus the number of zeros visited up

to this step.

3

Let T be a Red-Black Tree.

a) A Red-Black Tree is a Binary-Search Tree which satisfies:

(i) Every node is either red or black.

(ii) The root is black.

(iii) Every leaf (NIL) is black.

(iv) If a node is red, then both its children are black.

(v) For each node, all branches from the node to descendant leaves contain the same number of black nodes.

b) The height of a Binary Tree is by definition the maximum distance from the root node to one of the leaves nodes. In Red-Black Trees one need to take into account the presence of the NIL nodes. Either the root node or the NIL node in the longest path must not be counted, otherwise a single node with no children (and only two NIL leaves) would have height 2. To be consistent with the fact that a NIL node and a single node should have a different height, in the following I have considered the former having height 0 and the latter 1.
A procedure to compute it is to recursively compute it on the sub-trees. In Red-Black Trees the base case is that if x is a black NIL leaf, then $h(x) = 0$. This is important because if NIL leaves are set to have height 1, then a single node would end up with height 2, in contradiction with the Trees' Data Structure. Let $h(x)$ be the height of a node x, then $h(x) = 1 + max(h(x.left), h(x.right))$. This claim can be proved ab absurdum. By definition the height of a node is the maximum possible length of a path from that node down to one of the leaves in the rooted-sub-tree. Then it must be $h(x) > max(h(x.left), h(x.right))$.
Let us assume by contradiction that $h(x) > 1 + max(h(x.left), h(x.right))$. Let the path long $h(x)$ be a path from x down to a leaf y and, whitout loss of generality, containing $x.left$. It can be decomposed into the edge between x and $x.left$ and the path from $x.left$ to y. Being an edge, the former has length 1, while the latter has length $h(x.left)$, otherwise there should exist a leaf z with smaller distance from $x.left$ and thus from x. Then the equality $h(x) = 1 + h(x.left)$ holds. The same holds if the path contains $x.right$ and then the inequality $h(x) > 1 + max(h(x.left), h(x.right))$ can not hold. The following is the pseudo-code, which can be called on $T.root$ to obtain the height of T.

def Height(x):
if x $\neq$ NIL

```
    return 1 + max(Height(x.left), Height(x.right))
endif
return 0
enddef
```

Let h be the height of T, then the complexity is given by:

$$T(h) = 2T(h-1) + \Theta(1)$$
$$T(0) = \Theta(1)$$

Solved with the iterative method:

$$T(h) = 2T(h-1) + \Theta(1) = 2(2T(h-2) + \Theta(1)) + \Theta(1) =$$

$$= 2^k T(h-k) + \sum_{i=0}^{k-1} 2^i \Theta(1)$$

Up to the base case $h - k = 0 \Leftrightarrow h = k$. Then :

$$T(h) = 2^h + \Theta\Big(\sum_{i=0}^{h-1} 2^i\Big) = 2^h + \Theta\Big(\frac{1-2^h}{1-2}\Big) = \Theta(2^h) = O(n).$$

And the last equality follows from the fact that in Red-Black Trees with n nodes the height is $O(logn)$.

c) The Black-Height of a node is the number of black nodes below it in any branch. This means that the counting do not include the node itself if black, but always include the final NIL black leaf. It can be computed counting the number of black nodes in an arbitrary branch from the root. A possible algorithm is the following:

```
def BlackHeight(x):
bh ← 1
if x ≠ NIL
    while x.left ≠ NIL
        x ← x.left
        if COLOR(x) = BLACK
            bh ← bh + 1
        endif
    endwhile
endif
return bh
enddef
```

As for the algorithm in b) it can be called on $T.root$ to obtain the black height of T. The counting strats from 1 since black NIL leaves have height 1.

The correctness follows directly from the property (v) stated in point a). If T is made of a single node, then the black height is correctly computed as 1, the NIL leaf above.
The while-loop is performed in the worst case for a number of nodes equal to the height of T. The operations within takes $\Theta(1)$ time and since the height of a Red-Black tree is $O(logn)$ the overall asymptotic complexity is $T(n) = O(logn)$.

4

Let $(a_1, b_1), ..., (a_n, b_n)$ be n pairs of values in Z.

a) A possible data structure to handle the pairs is a nx2 matrix M such that for each row $[a_j, b_j]$ of M there exist a correspondent input pair (a_j, b_j) and vice-versa. Then to perform a lexicographical sort a possible procedure is:

```
def LexSort(M):
for i in 2,1:
    StableSort(M, i)
endfor
enddef
```

Where StableSort(M,i) is a sorting algorithm satisfying the key property of stability, that sorts the rows of M w.r.t. the elements in the i-th column. The stability property is needed in order to preserve the relative order of the pairs. The for-loop is performed twice and then the asymptotic cost complexity is equal to the one of the sorting algorithm chosen: $T_{LexSort}(n) = T_{StableSort}(n)$. Among those seen in the lectures the best choice happens to be Insertion Sort ($O(n^2)$).

b) Let us assume the existence of $k \in N$, constant w.r.t n, s.t. $a_i \in [1, k]$ for all $i \in [1, n]$. This assumption (and no less important the fact that the pairs take values in Z) makes it possible to use Counting Sort, which is not based on comparisons and sorts in linear time. Yet, if the assumption is only of half of the data, we can not get rid of the dominant term (the cost to sort the b_i) and the asymptotic complexity would remain the same.

c) In addition to the condition in b), let us assume the existence of $h \in N$, constant w.r.t. n, s.t. $b_i \in [1, h]$ for all $i \in [1, n]$. Being the values in M's columns in a bounded domain, the stable algorithm Counting Sort can be used for each column. The algorithm will not be in-place, since Counting Sort is not. The complexity is given by:

$$T_{LexSort}(n) = \Theta(n + k) + \Theta(n + h) = \Theta(n + max(h, k)).$$

5

a) The `Select` procedure studied relies on the assumption that the input array does not contains duplicate values. First let us recall that, as in the reformulated version of the problem, the function should take in input a potentially unsorted array A and an index $i \in [1, |A|]$ and output $j \in [1, |A|]$ s.t. $\tilde{A}[j] = \bar{A}[i]$, where $\tilde{A}$ is A after the computation and $\bar{A}$ is A sorted.
Let us consider the case in which we do not make a distinction between equal numbers in A.
Then the assumption only affects the asymptotic complexity, which we have proved to be $O(n)$, where $n = |A|$. The proof indeed is based on a upper bound for the number of elements smaller or equal to the partitioning element. Being those at most $\frac{7n}{10} + 6$, in the worst-case the recursive call will be performed on a input of such size and then the complexity is the one proved.
Without this assumption the partition could not be balanced enough. In the worst case scenario in which all the elements in A are equal to $k \in Z$, than the number of elements smaller or equal to the pivot k will be $n - 1$, leading to over-linear complexity.

b) An possible enhanced version of the `Select` algorithm to deal with duplicates is one which uses a tripartition of the array in place of the `Partition` procedure used. Once the pivot has been selected, the array is partitioned in the three subsets of elements smaller, equal and greater than it. This can be done using `ThreePartition` which return two indexes p and j, respectively the first and the last indexes of the central subarray of elements equal to the pivot. If $p = j$ then there are no duplicates of the pivot. As in the `Partition` procedure seen, the pivot is exchanged with the first element and then the array is scanned from left to right. Elements greater than the pivot are exchanged with the last one sill to be visited. The difference is that elements smaller than the pivot are immediately exchanged with the pivot.

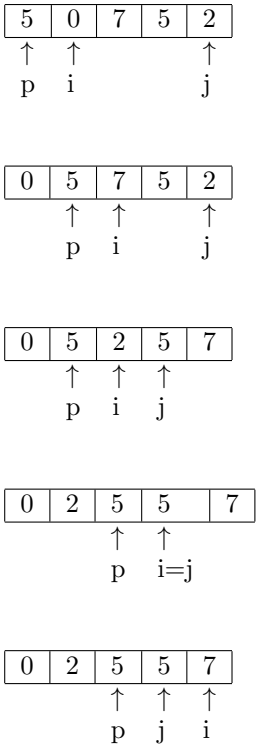
```
def ThreePartition(A,i,j,p):
    swap(A,i,p)
    (p,i) ← (i, i+1)
    while i ≤ j:
        if A[i] > A[p]:
            swap(A,i,j)
            j ← j - 1
        elseif A[i] < A[p]:
            swap(A, i, p)
            p ← p + 1
            i ← i + 1
        else:
            i ← i + 1
        endif
    endwhile
```

    return p, j
enddef

The while-loop iterates over all the elemtens between positions i and j, the operations inside take constant time. Let $i = 1$ and $j = n$, then:

$$T_P(n) = \Theta(1) + \sum_{i=1}^{n} \Theta(1) = \Theta(n).$$

Let us see how it works on a simple example:



Let `SelectPivot` be the function already used to chose the pivot as the almost-median, and `Sort` a sorting procedure, then the algorithm is:

def Select(A,l=1, r=n,i):
    if r-l$\leq$ 10:
        Sort(A,l,r)
        return
    endif

```
    p ← SelectPivot(A,l,r)
    k, j ← ThreePartition(A, l, r, p)
    if i > j:
    return Select(A, j+1, r, i)
    elseif i < k:
    return Select(A, l, k-1, i)
    else:
        return k
    endif
enddef
```

As already observed, if there are no duplicates in A then the selected pivot has no duplicates and $k = j$. In this case the algorithm is exactly the same seen during lessons and the asymptotic complexity as well, since the `ThreePartition` in place of `Partition` has the same cost.
If there are duplicates, then it could be the case that a pivot has duplicates, but these will be put inside the central part and will not change the upper bound for the number of elements in the recursive call. Then the equation for the complexity remains valid and the cost remains $T_s(n) = O(n)$.