Algorithmic Design

Andres Bermeo Marinelli

Homework II

`Exercise 1.` Let H be a `Min-Heap` containing n integer keys and let k be an integer value. Solve the following exercises by using the procedures seen during the course lessons:

(a) Write the pseudo-code of an in-place procedure RetrieveMax(H) to efficiently return the maximum value in H without deleting it and evaluate its complexity.
Solution: To fix ideas and notation, we refer to an array based representation for H. For Min-Heaps, we know that H[`PARENT(i)`]<H[`i`]. Thus, to extract the maximum, we need only to check the leaves since the heap property guarantees that the maximum won't be the parent of a node. We also know that Min-Heaps are nearly complete so there are $\lceil \frac{n}{2} \rceil$ leaves.

---

function RETRIEVEMAX(H)
  max ← H[ceil(H.size/2)]
  for i ← ceil(H.size/2)+1 `upto` H.size **do**
    if H[i]>max **then**
      max ← H[i]
    end if
  end for
  return max
end function

As mentioned above we have to check $\lceil \frac{n}{2} \rceil$ leaves and for each iteration we do $\Theta(1)$ work, therefore the overall complexity of the algorithm is $\Theta(\lceil \frac{n}{2} \rceil) = \Theta(n)$.

(b) Write the pseudo-code of an in-place procedure DeleteMax(H) to efficiently delete the maximum value from H and evaluate its complexity.
Solution: We can easily define a RetrieveMaxIndex(H) function which returns the index of the maximum of the Min-Heap. In order to delete it we must take care of two aspects:

(a) Preserving the Heap Property.

(b) Preserving the topological structure.

In order to preserve the topological structure, we must first perform a swap with the maximum key and the rightmost key (if the maximum is already located at the rightmost key then we are done and the algorithm terminates in constant time). After the swap, we can delete the rightmost node which now contains the maximum while preserving the heap topological structure. However, in doing the swap, the heap property might have been broken in the sub-tree that originally contained the maximum. In order to reestablish the heap property, we check if the parent of the newly swapped key is greater than the newly swapped key itself, and if this is the case, they are swapped. In other words, we check that the heap property is satisfied between the newly swapped key and it's parent and if its not satisfied we swap them.
However, this only restores the heap property locally and maybe the problem was moved upward. Therefore, one has to iterate the procedure in an upward fashion in order to be sure that it is restored globally. In the worst case scenario, the iteration will arrive one level below the root where we know it MUST terminate since the root is the minimum, by definition. Thus in the worst case it will iterate $h - 1 \approx \log_2 n$ times where h is the height of the tree.

```
function RetrieveMaxIndex(H)
    max ← H[roof(H.size/2)]
    maxidx← roof(H.size/2)
    for i ← roof(H.size/2)+1 downto H.size do
        if H[i]>max then
            max ← H[i]
            maxidx ← i
        end if
    end for
     return maxidx
end function

function DeleteMax(H)
    m ← RetrieveMaxIndex(H)
    if m == H.size then
        H.size ← H.size-1
     return 0
    else
        H[m]←H[H.size]                                  ▷ swap with rightmost key
        H.size ← H.size-1
        while H[PARENT(m)]>H[m] do
            par ← PARENT(m)
            swap(H, m, PARENT(m))
            m ← par
        end while
    end if
end function
```

The correctness is given by the fact that two scenarios can happen after the swap:

(a) The newly swapped key is greater than its parent, in this case we are done.

(b) The newly swapped key is smaller than the parent. Due to the heap property, the sibling of this newly swapped key is greater than the swapped key itself, thus, when we swap with the parent, the heap property is preserved also for the sibling. In other words, the heap property can only be broken "upwards" and not with siblings.

These considerations guarantee that when we terminate the algorithm, the heap property is restored everywhere.

The time complexity is as follows: $\Theta(n)$ to find the max index and $\Theta(\log_2 n)$ to execute a while loop that is executed in the worst case $\log_2(n)$ times and performs $\Theta(1)$ work at each iteration. Therefore, the overall complexity is $\Theta(n) + O(\log_2 n) = \Theta(n)$.

(c) Provide a working example for the worst case scenario of the procedure DeleteMax(H) (see Exercise 1b) on a heap H consisting in 8 nodes and simulate the execution of the function itself.
`Solution:` The worst case scenario is when the problem is not immediately fixed but is propagated upwards one level below the root. Below is a picture of an example and the simulation of the code.

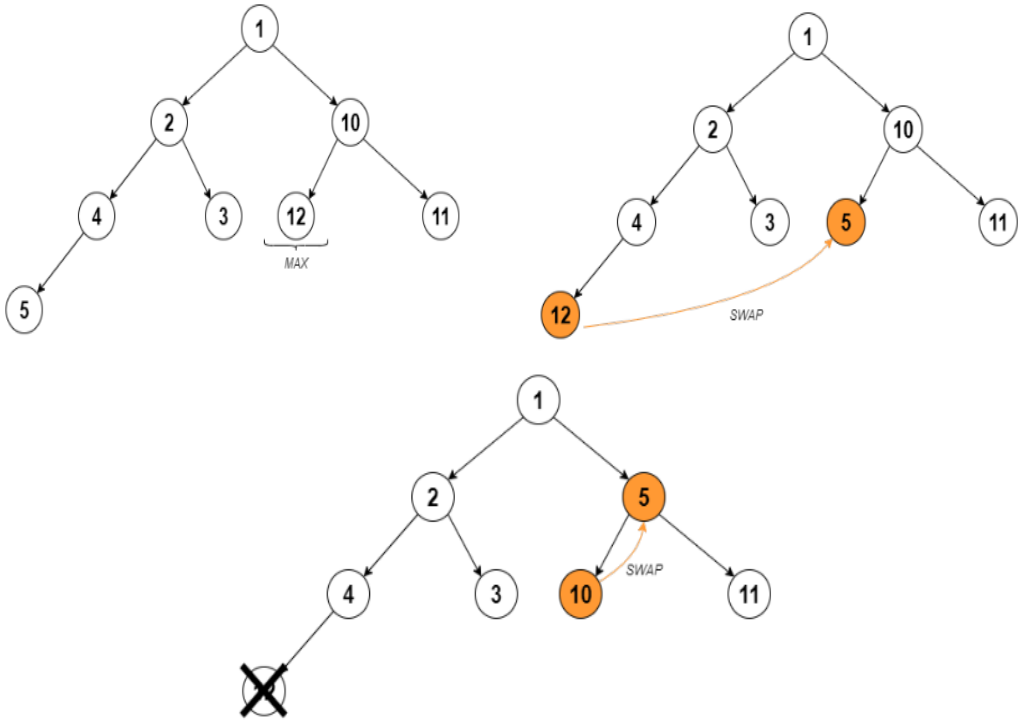

Figure 1: Working example of worst case scenario of the DeleteMax(H) procedure. After the swap, the heap property is broken and we must keep swapping upwards until we reach one level below the root. At this point we don't need to swap anymore since the root is the minimum by definition. The heap property is restored.

After having identified the maximum, we swap 12 and the rightmost node which contains 5. Then we swap 5 and its parent which contains 10. After 2 swaps we arrive at one level below the root where the algorithm terminates. At this point, the heap property is restored globally and we can delete the rightmost node which contains the maximum.

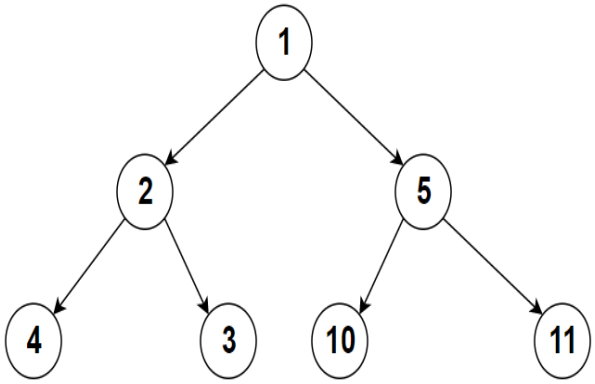

Figure 2: The final result after DeleteMax(H) is applied.

`Exercise 2.` Let A be an array of n integer values (i.e., the values belong to $\mathbb{Z}$). Consider the problem of computing a vector B such that, for all $i \in [1, n]$, $B[i]$ stores the number of elements smaller than $A[i]$ in $A[i+1, ..., n]$. More formally:

$$B[i] = |z \in [i+1, n] | A[z] < A[i]|$$

(a) Evaluate the array B corresponding to $A = [2, -7, 8, 3, -5, -5, 9, 1, 12, 4]$.
Solution: Following the definition of B we have the following:

$$B = [4, 0, 5, 3, 0, 0, 2, 0, 1, 0]$$

(b) Write the pseudo-code of an algorithm belonging to $O(n^2)$ to solve the problem. Prove the asymptotic complexity of the proposed solution and its correctness.
Solution:

```
Require: Array B of length n filled with zeroes.
  function TRANSFORM(A,B)
    for i ← 1 upto n do
      counter ← 0
      for j ← i upto n do
        if A[j]<A[i] then
          counter←counter +1
        end if
      end for
      B[i]← counter
    end for
  end function
```

The time complexity for this algorithm is $\Theta(n^2)$ since it essentially involves a linear scan for each of the n elements in the array. More formally, we have a double for loop with only $\Theta(1)$ operations inside it. At each iteration we scan $n-i$ elements so the total is $\Sigma_{i=1}^{n}(n-i) = n^2 - \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \in \Theta(n^2)$.
The correctness comes from the brute force approach of fixing an element and counting in the remaining portion of the array the number elements smaller than it by doing element by element comparisons.

(c) Assuming that there is only a constant number of values in A different from 0, write an efficient algorithm to solve the problem, evaluate its complexity and correctness.
Solution: Lets call c the number of non zero elements in A. By consequence, we know that there are $n-c$ zeroes in A, which we will denote with z. With a single linear scan, we can both copy the non zero elements into a different array D (which will have length c) and count the number of negative numbers in A which we denote with neg. Having done this, we start to scan A element by element and reduce the counters z and neg if we encounter a zero or a negative value. During the algorithm, three possible scenarios can happen:

(a) $A[i] = 0$. In this case, the number of elements less than 0 are all the negative numbers contained in the array which is given by the counter neg at that moment. So $B[i] = neg$ and we reduce the zero counter $z = z - 1$.

(b) $A[i] < 0$. In this case, we access D at the corresponding location where $D[j] = A[i]$ (note we need to remember the last place D had been read from) and scan the subsequent elements in D and count which are less than $A[i]$. We decrease the counter of negatives $neg = neg - 1$.

(c) $A[i] > 0$. In this case, we scan D in the same fashion as described above to find the elements less than $A[i]$ and then add the zero elements remaining in the array, given by the counter z at that moment.

Below is the pseudocode.

Algorithm 1 Preparation Step

```
neg← 0
j ← 1
for i←1 upto n do
  if A[i] ! = 0 then
    D[j] ← A[i]                          ▷ copy non zero elements of A to D
    j ← j + 1
  end if
  if A[i] < 0 then
    neg ← neg +1                         ▷ counter of negative elements
  end if
end for
```

Algorithm 2 Solving

```
j ← 1
for i ← 1 upto n do
    if A[i] == 0 then
        B[i]← neg
        z ← z − 1                                   ▷ Decrease counter of zeros
    else if A[i] < 0 then
        counter ← 0
        for k←j+1 upto Size(D) do
            if D[k] < D[j] then
                counter ← counter +1
            end if
        end for
        B[i] ←counter
        neg ← neg −1                                ▷ Decrease counter of negatives
        j ← j + 1                                   ▷ Index j used remember last place D was initially accessed from
    else if A[i] > 0 then
        counter ← 0
        for k←j+1 upto Size(D) do
            if D[k] < D[j] then
                counter ← counter +1
            end if
        end for
        B[i] ←counter +z
        j ← j + 1
    end if
end for
```

The complexity of the algorithm is by given different components:

- an initial linear scan to copy the non-zero elements and to count the negative values which is $\Theta(n)$.
- a second linear scan over all n elements.
  - For c of these elements, we have a limited amount of $\Theta(1)$ operations and we have an additional linear scan of an array of length c in which we do constant work.
  - For the remaining z elements, we do a few $\Theta(1)$ operations.

  The complexity of this portion is $\Theta(n) + c \cdot \Theta(c) = \Theta(n) + \Theta(c^2)$.

Thus, the overall complexity is $2\Theta(n) + \Theta(c^2) = \Theta(n) + \Theta(c^2)$. However, since c is a fixed constant, independent of n, the algorithm is $\Theta(n) + \Theta(1) = \Theta(n)$ which is a substantial improvement.
The correctness is quite easy to see since we are simply making use of counters and copying non zero elements in a shorter array but we are regardless looking through the whole array.

`Exercise 3.` Let T be a Red-Black Tree.

(a) Give the definition of Red-Black Trees.
Solution: Red Black Trees are BST's satisfying the following conditions:

- each node is either a Red or 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 (i.e they must have the same black height).

(b) Write the pseudo-code of an efficient procedure to compute the height of T. Prove its correctness and evaluate its asymptotic complexity.
Solution: We can use a recursive method to find the height of a RBT.

```
function FindHeight(node)
    if node==NIL then
     return 0
    end if
     return max(FindHeight(node.left), FindHeight(node.right))+1
end function
```

The complexity is $\Theta(n)$ since we visit each node once and perform $\Theta(1)$ operations at each recursive call. We prove the correctness by induction:

- *Base case:* For the tree composed only of the root, it is clear that the algorithm returns 1 as the height.
- *Inductive Hypothesis:* Assume that the algorithm correctly computes the height of a tree with nodes up to $n-1$.
- *Inductive Step:* We have to prove that the height is correct for the tree of n nodes.
  A tree of n nodes can be separated into its root and two subtrees T_1 and T_2.

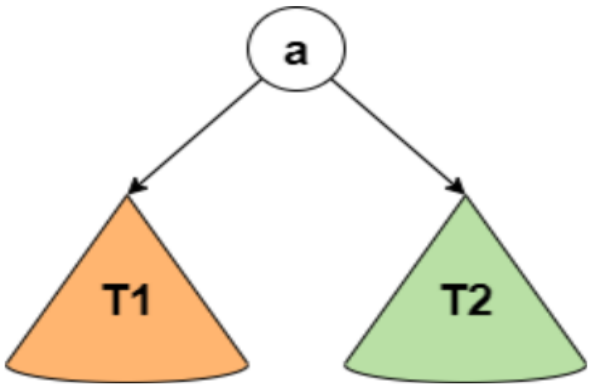

Figure 3: Tree of n nodes separated into its root and a right and left subtree.

We know that T_1 and T_2 have a number of nodes less than n and thus, by the inductive hypothesis, we can correctly calculate their height which we denote with h_1 and h_2, respectively. The height of the whole tree is given by the longest path from the root to the leaves. A simple decomposition principle tells us that any path from the root must pass first by the root, and then through either T_1 or T_2. Thus, to find the longest path we simply have to identify the maximum between h_1 and h_2 and then add one to this value which corresponds to traversing the root. However, this is corresponds to finding $max(h_1, h_2) + 1$, which is exactly the output of the algorithm. Thus, the algorithm gives the correct height of a tree with n nodes. Therefore, by induction, our algorithm is able to correctly calculate the height of all trees.

(c) Write the pseudo-code of an efficient procedure to compute the black- height of T. Prove its correctness and evaluate its asymptotic complexity.
Solution: Since the black height of the RBT is the same regardless of the path we choose, we can choose a random path (for instance, the one given by always going left) and count all the black nodes.

```
function BlackHeight(node)
    h←0
    while node != NIL do
        if node.color==BLACK then
            h← h + 1
        end if
        node ← node.left
    end while
     return h
end function
```

Since we have to traverse the whole RBT which has $\lceil \log_2 n \rceil$ levels we have to perform $\lceil \log_2 n \rceil$ iterations, each of which does $\Theta(1)$ work. Thus the overall complexity is $\Theta(\log_2 n)$. The correctness is given by the fact the Red Black

property guarantees the same BH for any path from the root to a leaf and we are simply traversing the tree always to the left and counting the black nodes.

`Exercise 4.` Let $(a_1, b_1), ..., (a_n, b_n)$ be n pairs of integer values. They are lexicographically sorted if, for all $i \in [1, n-1]$, the following conditions hold:

- $a_i \leq a_{i+1}$
- $a_i = a_{i+1}$ implies that $b_i \leq b_{i+1}$

Consider the problem of lexicographically sorting n pairs of integer values.

(a) Suggest the opportune data structure to handle the pairs, write the pseudo-code of an efficient algorithm to solve the sorting problem and compute the complexity of the proposed procedure.
Solution: A possible data structure is an array L of pairs $T_i = (a_i, b_i)$ for $i = 1, ..., n$.
To solve the sorting problem, we define a function LexOrder(T_i,T_j) which establishes whether $T_i \leq T_j$ using the definition above. Since this is a total ordering, the sorting problem is well defined and we simply call HeapSort() to sort the array using LexOrder() as the total order to be used.

```
function LexOrder(T_i,T_j)
    if a_i == a_j then
     return b_i ≤ b_j
    else
     return a_i ≤ a_j
    end if
end function
```

Require: Array of L tuples $T_i = (a_i, b_1)$ for $i = 1, ..., n$; LexOrder(T_i, T_j)

```
function LexSort(L)
    HeapSort(L, LexOrder())
end function
```

Creating the array has complexity $\Theta(n)$ and we know that HeapSort() is $\Theta(n \log_2 n)$. Thus, the overall complexity $\Theta(n \log_2 n)$

(b) Assume that there exists a natural value k, constant with respect to n, such that $a_i \in [1, k]$ for all $i \in [1, n]$. Is there an algorithm more efficient than the one proposed as solution of Exercise 4a? If this is the case, describe it and compute its complexity, otherwise, motivate the answer.
Solution: No there isn't. Since a_i is limited in a certain range we could think to use CountingSort() to sort by a_i with $\Theta(n)$ complexity. However, in this way, we first would need to define two total orderings OrderA() and OrderB() which order by a_i and b_i respectively. Then we split the sorting problem by first sorting by b_i using HeapSort() and then sorting by a_i using CountingSort() (the stability of the algorithm guarantees the correctness of the result). This would result in a total complexity $\Theta(n \log_2 n) + \Theta(n) = \Theta(n \log_2 n)$ which is the same as before. In other words, we are limited by the fact that we also need to sort by b_i for which we don't have any domain knowledge and will take $\Theta(n \log_2 n)$ time to sort. Therefore, there is no improvement.

(c) Assume that the condition of Exercise 4b holds and that there exists a natural value h, constant with respect to n, such that $b_i \in [1, h]$ for all $i \in [1, n]$. Is there an algorithm to solve the sorting problem more efficient than the one proposed as solution for Exercise 4a? If this is the case, describe it and compute its complexity, otherwise, motivate the answer.
Solution: Since a_i and b_i are limited in a certain range, one could use CountingSort() to sort by b_i and a_i. We define two total orderings OrderA() and OrderB() and then use CountingSort() to first order by b_i and then order the result by a_i. Since CountingSort() is stable the algorithm is obviously correct. The complexity of the algorithm becomes $\Theta(n) + \Theta(n)$ which is still $\Theta(n)$.

`Exercise 5.` Consider the `select` algorithm. During the lessons, we explicitly assumed that the input array does not contain duplicate values.

(a) Why is this assumption necessary? How does relaxing this condition affect the algorithm?
Solution: This assumption is necessary in order to ensure that the median of medians is a good pivot. This can be immediately seen if for some reason, we pass to the select procedure an array composed of all 1's and ask for the value, if A were sorted, in position 1. The median of medians would always be 1, so partitioning the array would result in a left sub-array filled with $n-1$ 1's and an empty right sub-array. Repeating this procedure reduces the length of the left subarray by 1 each time which means we have to repeat the partition procedure, which is $\Theta(n)$, a total of n times. Therefore, the complexity of solving the problem becomes $\Theta(n^2)$ which is much worse than simply sorting and returning the appropriate value, which takes $\Theta(n \log_2 n)$.

(b) Write the pseudo-code of an algorithm that enhances the one seen during the lessons and evaluate its complexity.
Solution: The main issue with repeating values is that the partition procedure is not sensitive to the fact that the pivot might be repeated often, this in turn doesn't guarantee the good splitting ratio of the median of medians[1]. In order to fix this, we can implement a three way partition[2] procedure which is a generalization of the partition implementation. The three way partition operates with three indices instead of two and at the end of the procedure, the array is separated in three portions:

(a) $[\texttt{begin}, ..., k.\texttt{left} - 1]$ are indexes of elements of A strictly smaller than the pivot.

(b) $[k.\texttt{left}, ..., k.\texttt{right}]$ are indexes of elements of A equal to the pivot.

(c) $[k.\texttt{right} + 1, ..., \texttt{end}]$ are elements of A strictly greater than the pivot

where k is the object returned by the three way partition procedure and has two elements, the `left` index and the `right` index.
In this way, when we do the select procedure, we have to check if the index i falls in either of the three portions. If i is in the left or right portion, we do recursion as before, however, if i falls in the middle portion, then we simply return the index since this portion contains all repeated values.
The intuition behind this whole procedure is the following: if the median of medians procedure picks a value that is repeated, we basically "group" all the repeated values together and put all of them in the correct position within the array. This in turn guarantees once again that the median of medians provides a good splitting ratio. This can be seen by the fact that we can think of symbolically substituting all the repeated values in the middle portion of the array (after the three way partition) with a single value and storing information that this value occupies a certain amount of indexes in A. In this way we re-conduce ourselves to the original problem where we know that the median of medians is smaller than at least $\frac{3n'}{10} - 6$ elements and greater than at least $\frac{7n'}{10} + 6$ where n' is the size of the array minus the repeated values of the pivot. Ultimately, this is the factor that counts since we want to be sure than when we fall on the right or left portion of the array we have a good splitting ratio.
We also notice that the worst case scenario described in (a) is solved easily since the three way partition groups all the 1's in the middle portion of the array and has empty left and right sub-arrays. The select procedure immediately identifies that the index falls in the middle portion and thus simply returns the index.
Below is the pseudocode.

Algorithm 3 Modified Select

Require: PARTITION3W()

```
function SELECT(A, l = 1, r = |A|, i)
    if r − l ≤ 10 then                                          ▷ Base Case
        SORT(A,l,r) return i
    end if

    j ←SELECT_PIVOT(A,l,r)                                      ▷ Mean of Medians Procedure
    k ←PARTITION3W(A,l,r,j)                                     ▷ Three Way Partion

    if i ≥ k.left and i ≤ k.right then
     return i
    end if

    if i < k.left then
     return SELECT(A,l,(k.left − 1),i)
    end if

  return SELECT(A,(k.right + 1), r, i)
  end function
```

Algorithm 4 Three Way Partition

```
function PARTITION3W(A, begin = 1, end = |A|, piv)
    i ← begin
    j ← begin
    k ← end
    while j ≤ k do
        if A[j] < piv then
            SWAP(A[i], A[j])
            i ← i + 1
            j ← j + 1
        else if A[j] > piv then
            SWAP(A[j], A[k])
            k ← k − 1
        else
            j ← j + 1
        end if
    end while
    bound.LEFT← i
    bound.RIGHT← k
     return bound
  end function
```

The complexity of the three way partition procedure is $\Theta(n)$ since at each iteration we scan an element of the array and move either "forward" or "backwards" (doing constant work) with sentinels i, j, k and terminate when the sentinels meet. Since the partition procedure is still $\Theta(n)$ and the median of medians using three way partition still guarantees a good splitting ratio, the overall complexity of the select algorithm is still $\Theta(n)$.

References

[1] Bi, Brian. "How Exactly to Do Linear Time Quickselect with Duplicate Elements?" B_2^3, 1 Jan. 2021, spin0r.wordpress.com/2021/01/01/how-exactly-to-do-linear-time-quickselect-with-duplicate-elements/.

[2] "Dutch National Flag Problem." Wikipedia, Wikimedia Foundation, 4 Mar. 2021, en.wikipedia.org/wiki/Dutch_national_flag_problem.