Homework Batch 2

Marco Sicklinger

April 2021

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:

Exercise 1.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.

Answer

Observation. *The **heap** and the **topological properties** imply that the maximum value in a heap* `H` *can only be found among the leaves of* `H` *itself, that is among those nodes which have no children.*

Proposition. *Given a heap* `H` *with n nodes, the number of leaves (nodes with no children) is*

$$l = \left\lfloor \frac{n+1}{2} \right\rfloor.$$

Proof. Let n be the number of nodes in `H`, and suppose r is the number of nodes that would make `H` a complete binary tree, if added.
If h is the height of the corresponding complete binary tree (obtained by completing the last level of the heap `H`), this tree has

$$\sum_{k=0}^{h} 2^k = \frac{2^{h+1}-1}{2-1} = 2^{h+1} - 1$$

nodes, and 2^h represents the number of leaves of this complete binary tree, which, in terms of the aforementioned quantities, becomes

$$2^h = \frac{n+r+1}{2}.$$

At this point, it is possible to distinguish two cases:

- the nearly complete binary tree `H` is obtained by the removal of an even number r of nodes from the complete binary tree; a possible situation is pictured below, where the nodes colored in red are the removed nodes.
  In this case, $r/2$ new leaves are created: since r is a multiple of 2 and every node has at most two children and because it is necessary to respect the topological properties (the heap `H` is *nearly complete binary tree*) when removing nodes from the last level. Then, the number of leaves now present in the tree is

$$l_{even} = \frac{n+r+1}{2} - r + \frac{r}{2} = \frac{n+1}{2}.$$

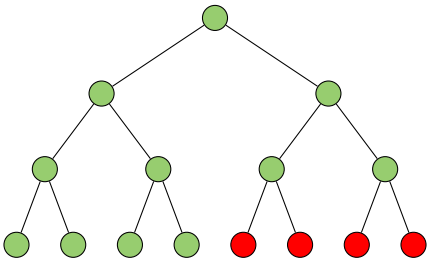


Figure 1: Removal of an even number of nodes from the last level of a complete binary tree

- the nearly complete binary tree `H` is obtained by the removal of an odd number r of nodes from the complete binary tree; a possible situation is pictured below, where the nodes colored in red are the removed nodes.
  In this case, $(r-1)/2$ new leaves are created: the removal of an even number $(r-1)$ of nodes is performed as explained above and the last one must be removed respecting the topological property of the heap `H`. However the parent node x of the removed one, by definition of heap itself (again, a *nearly complete binary tree*), must have another child, which does not make x a leaf. Then, the number of leaves now present in the tree is
  $$l_{even} = \frac{n+r+1}{2} - r + \frac{r-1}{2} = \frac{n}{2}.$$

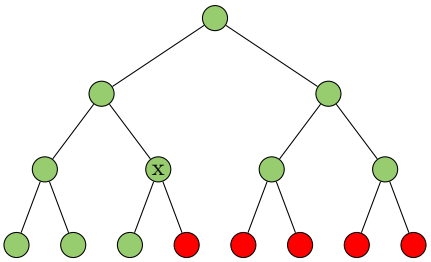


Figure 2: Removal of an odd number of nodes from the last level of a complete binary tree

Since n is a positive integer number, the two results above can be written as a single relation:

$$l = \left\lfloor \frac{n+1}{2} \right\rfloor.$$

□

Then, in order to find the maximum one has to check at most l nodes. If the array representation is used, these nodes are represented by the last l cells of the array which represents the heap `H` itself.
The function can be written as follows: the variable storing the maximum value `max` is initialized with the last value contained in the heap; then, with the help of a loop, an iterative comparison between `max` and the last l values of the heap is performed.

```
1   def RetrieveMax(H):
2       # initialize max with last element saved in the
3       # array representation of the heap H
4       max ← H[H.size]
5       # loop over all the leaves
6       for i in floor((H.size+1)/2)..1:
7           # if a bigger value is found, assign it to max
8           if total_order(max, H[H.size-i]):
9               max ← H[H.size-i]
10          endif
```

```
        endfor
        return max
    enddef
```

The `total_order()` function has the same meaning as the one used during classes. Moreover, we are assuming that `H` is an istantiation of a class representing the binary heap in a generic programming language; in such a class members such as the `size` of the heap are defined.

Complexity of the algorithm

The complexity of the inner part of the `for` loop belongs to $O(1)$, and the same holds for the assignment at line 6: since the temporary variable storing the `max` has fixed size (depending on the used architecture, programming language, etc., but always fixed) the procedure is in-place. Then, the complexity of the above algorithm is

$$T(n) = \sum_{i=0}^{l} C = \sum_{i=0}^{\lfloor \frac{n+1}{2} \rfloor} C \leq \sum_{i=0}^{\frac{n+1}{2}} C = C\frac{n+1}{2},$$

where C is the constant that take into account the expressions inside the `for` loop. This proves that $T(n) \in O(n)$. Moreover,

$$T(n) \geq \sum_{i=0}^{\frac{n}{2}} C = C\frac{n}{2},$$

which proves that $T(n) \in \Omega(n)$. From the results above, one can say that $T(n) \in \Theta(n)$.

Exercise 1.b

Write the pseudo-code of an in-place procedure `DeleteMax(H)` to efficiently deletes the `maximum` value from `H` and evaluate its complexity.

Answer

Observation. *The heap property implies the node which stores the maximum value of the heap* `H` *cannot have any children, which makes it a leaf.*

From the previous observation it follows that it is possible to swap the node storing the maximum value with the last node in the array representation of the heap `H`. Let b be the branch of the heap ending with the node that previously stored the maximum value, and now storing what was the last element of the heap `H`. This procedure may damage the heap property but it does not break the topological property of the heap `H`. After the aforementioned swap is performed, the size of the heap must be decreased, so to exclude the node storing the maximum from the heap itself.
Then it is necessary to restore the heap property, if broken. In order to do that, one can check if such a property is satisfied between the yellow-colored node and its parent, and if it is not the case, swap the two. Then, proceed in the same way until a root's child is reached. Going upper is useless since the orange-colored node is for sure "bigger" than the root.
The implementation might be planned as follows.

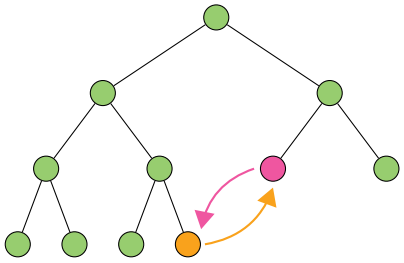
```
    def DeleteMax(H):
        # initialize max with last element saved in the
        # array representation of the heap H
        max ← H[H.size]
        # loop over all the leaves
        for i in (H.size+1)//2..1:
            # if a bigger value is found, store corresponding index
            if total_order(max, H[|H|-i]):
                i_max ← H.size-i
            endif
        endfor
        # rewrite the index corresponding to the maximum so that
```

```
        # it stores the position of the maximum itself wrt the
        # beginning of the array
        i_max ← H.size - i_max
        # swap the node related to the maximum with the last one
        # in the array representation
        H.swap(i_max, H.size)
        # decrease size by one
        H.size ← H.size - 1
        # fix heap property
        while i_max > 1:
            # if heap property is not satisfied, swap the node
            # that breaks it with its parent
            if total_order(H[i_max], H[parent(i_max)]):
                H.swap(i_max, parent(i_max))
            else:
                # if parent and child satisfy heap property, end the function
                return
            endif
            i_max ← i_max - 1
        endwhile
    enddef
```

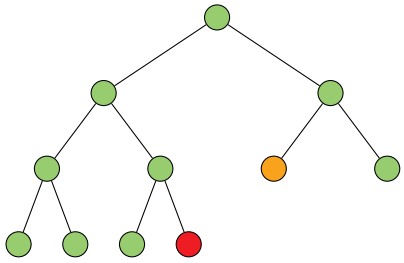
The `parent()` function can also be definined as a member function of the class representing the binary heap type as follows.

```
    def parent(node):
        return floor(node/2)
    enddef
```

As defined in class. Function `swap()` is a suitably defined function for swapping the heap's elements.



(a) Assume that the node colored in violet stores the maximum value, and the orange-colored node is the last value stored in the array representation of the heap; the first step consist in swapping those two nodes.



(b) Once swapped, the maximum value, now stored as the last element in the array representation of the heap, is removed from the heap (red-colored), while the orange-colored node may cause problems as far as the heap property is concerned.

Figure 3: Deletion of the maximum

Complexity of the algorithm

From Exercise 1.a, the complexity required to localize the maximum value, i.e. retrieve the index in the array representing the heap which corresponds to the maximum value, belongs to $O(n)$. The redefinition of maximum element's index and heap size and the swap procedure have a cost of $\Theta(1)$. Finally, the block inside the `while` loop has, again, a cost of $\Theta(1)$, and it is repeated untill the root's child is reached, that is at worst for a length of $\log_2(n) - 1$. Overall complexity is then

$$T(n) = Cn + C_1 + \sum_{i=1}^{\log_2(n)-1} C_2 \leq Cn + C_1 + \sum_{i=1}^{\log_2 n} C_2 = Cn + C_2 \log_2 n + C_1,$$

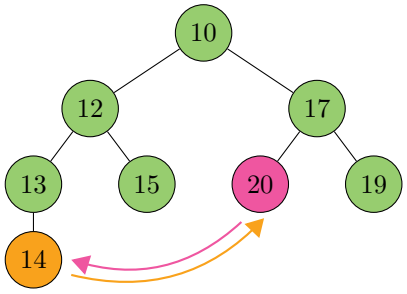
where C is the constant involved in the retrieval of the maximum's index. This proves that $T(n) \in O(n)$.

Exercise 1.c

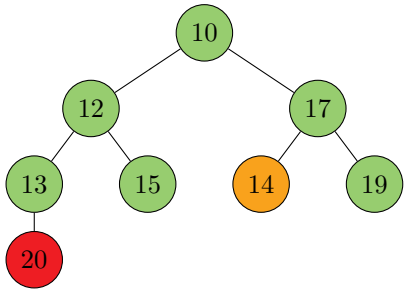
Provide a working example for the worst case scenario of the procedure `DeleteMax(H)` (see Exercise 1b) on a heap `H` consisting in 8 nodes and simulate the execution of the function itself.

Answer

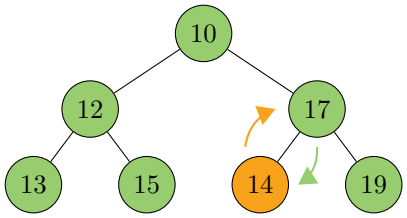
The worst case scenario happens when the swapping procedure is needed up to the root's child in order to fix the heap property. A possible example is the following one.
Given a 8-noded heap, a possible representation of the worst case scenario is the displayed in figure 4. In the case represented above, it is necessary to restore the heap property, swapping



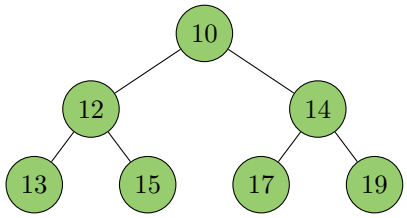
(a) The first step of the `DeleteMax()` procedure is to swap the maximum node with the last node of the heap.



(b) Once swapped, the maximum value, now stored as the last element in the array representation of the heap, is removed from the heap. It is possible to notice that node 14 now causes problems as far as the heap property is concerned.



(c) Then, in order to restore the heap property, it is necessary to swap node 14 with node 17.



(d) Now the procedure is completed: the resulting heap has a new maximum (node 19) and satisfy both the heap and the topological properties.

Figure 4: Deletion of the maximum from the 8-noded heap.

nodes until a root's child is reached. This is the worst case scenario, whose complexity still belongs to $O(n)$, since the procedure of finding the maximum is the one that brings the greater contribution to the overall complexity of `DeleteMax()`.

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:

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

Exercise 2.a

Evaluate the array B corresponding to $A = [2, -7, 8, 3, -5, -5, 9, 1, 12, 4]$.

Answer

Let's compute B element by element, using the above definition.

$$B[1] = |\{z \in [2, 10] : \ A[z] < A[1] = 2\}| = 4;$$

$$B[2] = |\{z \in [3, 10] : \ A[z] < A[2] = -7\}| = 0;$$

$$B[3] = |\{z \in [4, 10] : \ A[z] < A[3] = 8\}| = 5;$$

$$B[4] = |\{z \in [5, 10] : \ A[z] < A[4] = 3\}| = 3;$$

$$B[5] = |\{z \in [6, 10] : \ A[z] < A[5] = -5\}| = 0;$$

$$B[6] = |\{z \in [7, 10] : \ A[z] < A[6] = -5\}| = 0;$$

$$B[7] = |\{z \in [8, 10] : \ A[z] < A[7] = 9\}| = 2;$$

$$B[8] = |\{z \in [9, 10] : \ A[z] < A[8] = 1\}| = 0;$$

$$B[9] = |\{z \in [10, 10] : \ A[z] < A[9] = 12\}| = 1;$$

$$B[10] = |\{z \in \varnothing : \ A[z] < A[10] = 4\}| = 0.$$

The result is $B = [4, 0, 5, 3, 0, 0, 2, 0, 1, 0]$.

Exercise 2.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.

Answer

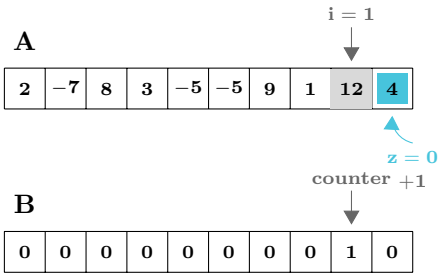
A straightforward approach consists in counting how many elements are smaller than a given element, for each element of A, with a double loop. In order to make the computation of the complexity simpler, the inner loop is performed in such a way that the elements of A are checked going from the last to the first. The pseudo-code may be written as follows.

```
1    def compute_B(A):
2        # allocate memory for array B
3        # by setting the default value to zero, the last element
4        # of B already has its correct value
5        B ← allocate_array(|A|, default_value = 0)
6        # procedure makes sense if A has more than one element,
7        # otherwise B has just one zero element
8        if |A| > 1:
9            # check if A is the null vector
10           counter_0 ← 0
11           for i in 1..|A|:
12               if A[i] = 0:
13                   counter_0 ← counter_0+1
14               endif
15           endfor
16           # if A is the null vector, then B is equal to A
17           if counter_0 = |A|:
```

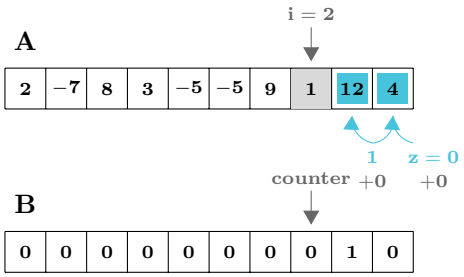
```
                return A
            endif
            # otherwise...
            for i in 1..|A|-1:
                # for each element of A the counter must be set to zero
                counter ← 0
                # check how many element between positions |A| and
                # |A|-i excluded satisfy condition
                for z in 0..(i-1)
                    if A[|A|-z] < A[|A|-i]:
                        counter ← counter + 1
                    endif
                endfor
                # assign computed count to corresponding element in B
                B[|A|-i] ← counter
            endfor
        endif
        return B
    enddef
```

Correctness of the algorithm

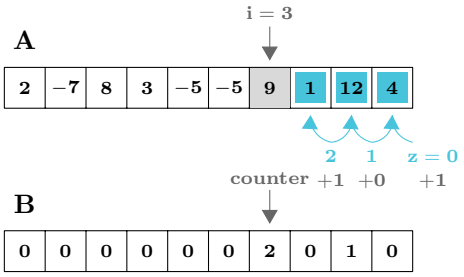
As a proof of correctness we will consider the array $A = [2, -7, 8, 3, -5, -5, 9, 1, 12, 4]$ and compute B following the procedure written above.



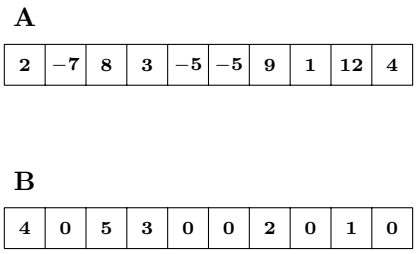
(a) The procedure starts with setting $i = 1$. Then z can assume only one value $z = 0$. At this point the condition $A[10] < A[9]$ is checked. Since it is true, `counter` is incremented by one unit and saved in the corresponding i-th position in B.



(b) In the next step i is incremented by one, to $i = 2$. Now $z \in [0, 1] \cap \mathbb{Z}$. First $A[10] < A[8]$ is checked, which is false, then $A[9] < A[8]$, which is again false. Total counts amount to zero, and the value is stored in the corresponding eighth position in B.



(c) Taking another step means increasing i to $i = 3$. As a consequence, $z \in [0, 2] \cap \mathbb{Z}$. In this case, conditions $A[10] < A[7]$, $A[9] < A[7]$, $A[8] < A[7]$ are checked. Only the second one is not true, so the value 2 is stored in $B[7]$.



(d) Following for all the remaining $i \in [0, 6] \cap \mathbb{N}$ the procedure above, one manages to fill the corresponding B's elements. Notice that the last one is already set to zero.

Figure 5: A scheme representing the procedure followed by `compute_B()` function.

It has just been proved that the algorithm works for constant sized array. Given an array of general size $n > 1$, supposed not to be the null vector, one can assume that the last $m \in [1, n) \cap \mathbb{N}$ elements of B are computed correctly and prove the correctness of the $(n - m - 1)$-th element of the array, that is of the $(m + 1)$-th element of B from its end.

At this point, the condition $A[\,n-j\,] < A[\,n-m-1\,]\ \forall j \in [\,1,m\,] \cap \mathbb{N}$ is checked: is it false? Then `counter` is not updated. Is it correct? Then `counter` is increased by one. $B[\,n-m-1\,]$ will then store the correct count.

Moreover, it is possible to distinguish a few pathological scenarios:

- $|A| = 1$: in this case the array B is allocated with size 1 and the only value is 0. Successively, the outermost `if` statement is ignored, and the returned value is simply $B = [\,0\,]$, which is the correct solution since there is no further element than $i = 1$ in A.
- $A[\,i\,] = 0\ \forall i \in [\,1,n\,] \cap \mathbb{N}$, that is, A is the *null vector*: in this case the condition at line 17 of the pseudo-code is satisfied, making the function return the null vector itself without going through the standard procedure. It is the correct solution since no element in A satisfy the condition that defines B.

Complexity of the algorithm

The code inside the first for loop takes complexity belonging to $\Theta(1)$. Then, the complexity of the first for loop is

$$T_1(n) = \sum_{i=1}^{n} C_1 = C_1 n,$$

where C_1 is a constant value. As a consequence, $T_1 \in \Theta(n)$. Although this piece of code is not necessary, it reduces the complexity of the entire algorithm if A is the null vector.
The code inside the inner `for` loop (inside the second loop) requires a complexity belonging in $\Theta(1)$. The same is valid for the other assignments in the outer `for` loop. The complexity of the inner for loop amounts to

$$T_{in}(i) = \sum_{z=0}^{i-1} C_2 = \sum_{z=1}^{i} C_2 = C_2 i,$$

where C is a constant value. As a consequence, $T_{in} \in \Theta(i)$. The overall complexity is then

$$T_{tot}(n) = \sum_{i=1}^{n-1} T_{in}(i) + T_1(n) + C_3 = C_2 \sum_{i=1}^{n-1} i + C_1 n + C_3 = \frac{C_2}{2} n^2 + \left(C_1 - \frac{C_2}{2} \right) n + C_3,$$

where C_3 is a constant value. This proves that $T_{tot} \in O(n^2)$ as requested.

Exercise 2.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.

Answer

If there are $\mathbb{N} \ni k < n$ values different from zero in A, with k constant with respect to n, the remaining non-constant $n - k$ values are zeroes. This fact could help in exploiting a constant amount of memory for saving the position of the non-zero elements, so to reduce the complexity of the algorithm which used for the necessary comparisons that allows the user to have B computed.

Observation. *When inspecting the elements* $\mathrm{A}[\,\mathrm{i}\,]$, $\mathrm{i} \in [\,0,\mathrm{n}\,] \cap \mathbb{N}$, *one of the following three cases may occur:*

- $\mathrm{A}[\,\mathrm{i}\,] = 0$*: since for all the* $\mathrm{z} \in [\,\mathrm{i}+1,\mathrm{n}\,] \cap \mathbb{N}$ *such that* $\mathrm{A}[\,\mathrm{z}\,] = 0$*, the condition* $\mathrm{A}[\,\mathrm{z}\,] < \mathrm{A}[\,\mathrm{i}\,]$ *is not satisfied, in this case one should only take into account a constant number* (*w.r.t.* n) *of values, different from zero and whose positions* (*greater than* i) *are somehow stored, that actually satisfy the condition above.*

- $A[i] < 0$*: an argument similar to the one above holds in this case.*
- $A[i] > 0$*: again, one should check a constant number of non-zero element and count those that satisfy* $A[z] < A[i]$*, such that* $z > i$*. Then one must add to the obtained count the number of zero elements with position* $z > i$*, which amounts to* $|A| - i - a$*, where* a *is the number of non-zero elements after position* i*.*

It is also possible to notice that, whatever the value of $A[i]$*, the procedure to check the positions of the non-zero elements is always the same.*

The algorithm can be designed as follows.

```
def new_compute_B(A):
    # allocate memory for array B
    # by setting the default value to zero, the last
    # element of B already has its correct value
    B ← allocate_array(|A|, default_value = 0)
    # build array containing positions of
    # non-zero elements in A
    positions = get_non_zero_pos(A)
    # procedure makes sense if A has more than one element,
    # otherwise B has just one zero element
    if |A| > 1:
        # check if A is the null vector, and if it is, return it
        if |positions| = 1 and positions[1] = 0:
            return A
        endif
        # otherwise...
        for i in 1..|A|:
            # initialize counter variable and variable
            # needed for storing the number of non-zero
            # elements up to position i
            counter ← 0
            non_zeroes_before ← 0
            # update counter when conditions are satisfied
            for j in 1..|positions|:
                # non_zeroes_before is useful only if element is positive
                if positions[j] <= i and A[i] > 0:
                    non_zeroes_before = non_zeroes_before + 1
                endif
                if positions[j] > i and A[positions[j]] < A[i]:
                    counter ← counter + 1
                endif
            endfor
            # if element is positive, add number of zeroes after i
            if A[i] > 0:
                counter ← counter +
                          + (|A|-i) - (|positions| - non_zeroes_before)
            endif
            # update corresponding B's element
            B[i] ← counter
        endfor
    endif
    return B
enddef
```

The function `get_non_zero_pos()` can be written as follows.

```
def get_non_zero_pos(A):
    # allocate array for positions
    positions ← allocate_array(1, default_value = 0)
    # evaluate first position
    if A[1] < 0 or A[1] > 0:
        positions[1] ← 1
    endif
    for i in 2..|A|:
        # if non-zero element is found, store its and
        # previous non-zero elements' positions in the array
        if A[i] < 0 or A[i] > 0:
            tmp_pos ← allocate_array(|positions|+1, default_value = NIL)
```

```
13                  for j in 1..|positions|:
14                      tmp_pos[j] ← positions[j]
15                  endfor
16                  tmp_pos[|tmp_pos|] ← i
17                  # make memory available again
18                  deallocate_array(positions)
19                  positions ← allocate_array(|tmp_pos|, default_value = NIL)
20                  for j in 1..|positions|:
21                      positions[j] ← tmp_pos[j]
22                  endfor
23                  # make memory available again
24                  deallocate_array(tmp_pos)
25              endif
26          endfor
27          return positions
28      enddef
```

Correctness of the algorithm

Let us prove the correctness for $|A| = 1$: it is a trivial case under the considered hypothesis; the outermost `if` statement is ignored, and $B = [\,0\,]$ is returned, which is the correct answer. A general case, with $n > 1$, can be represented as follows.

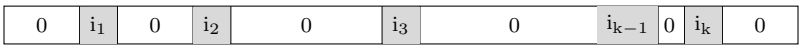


Figure 6: Array with constant (w.r.t. n) non-zero values.

The first step of the procedure is to store the positions of the non-zero elements, $[\,i_1, i_2, ..., i_{k-1}, i_k\,]$, using `get_non_zero_pos()`. Suppose that A is not the null vector, then `position` must have at least an element different from 0.
Assume now that the first $m \in [1, n) \cap \mathbb{N}$ element of the array B have been correctly computed. Let us prove the correctness for the $(m+1)$-th element:

- If $(m+1)$-th element is zero, then every time that the `positions` array returns an index such that the corresponding element of the array is negative and with position greater than $m+1$, `counter` is updated. In this case the update of the value of `non_zeroes_before` and the `if` statement at line 33 are not performed. The result is then stored in $B[\,m+1\,]$.

- if $(m+1)$-th element is negative, the `counter` gets to be updated only when smaller values are found using `positions`'s elements greater that $m+1$ itself. Again, the procedures which require the $(m+1)$-th element to be positive are ignored. The correct count is then stored in $B[\,m+1\,]$.

- if $(m+1)$-th element is positive, the `counter` gets updated only when smaller values are found using `positions`'s elements greater that $m+1$ itself. Moreover, the number of zeroes `non_zeroes_before` placed before this position in A is computed, so when entering the `if` statement at line 33 this value is suitably taken into account. The correct count is then stored in $B[\,m+1\,]$.

Again, it is possible distinguish a few pathological scenarios:

- $|A| = 1$: it is a trivial case under the considered hypothesis. However, in such a case, the outermost `if` statement is ignored, and $B = [\,0\,]$ is returned.

- $A[\,i\,] = 0 \ \forall i \in [\,1, n\,] \cap \mathbb{N}$, that is, A is the *null vector*: in this case, the condition at line 13 is satisfied and a null vector is returned.

- The case in which $k = n$, that is the case in which no zero elements are present, brings back to the point 2.b of the exercise. Indeed, all the advantages of having a constant number of element different form zero (that is, of having a non constant number of elements all equal to each other) are lost. This argument holds also for the `get_non_zero_pos()` function.

Complexity of the algorithm

The complexity T_p of `get_non_zero_pos()` can be evaluated as follows: the first `if` statement has complexity which belongs to $\Theta(1)$. Then there is a loop over $n-1$ steps. The code inside it has complexity which belongs to $\Theta(1)$, since the number of elements different from zero is constant with respect to n. As a consequence, at each step `positions` and `tmp_pos` stores a number of values bounded by k (defined above), which is constant with respect to n. In view of this

$$T_p(n) = C_1 + \sum_{i=2}^{n} C_2 = C_1 + C_2(n-1) = C_2 n + C_1 - C_2,$$

where C_1 and C_2 are constants. This proves that $T_p \in \Theta(n)$.
In the function `new_compute_B()` the code inside the outer `for` loop again has complexity which belong to $\Theta(1)$, for the same reasons explained above. Then the overall complexity T is

$$T(n) = T_p(n) + \sum_{i=1}^{n} C_3 + C_4 = C_2 n + C_1 - C_2 + C_3 n + C_4 = C' n + C'',$$

where C_3 and C_4 are constants and $C' = C_2 + C_3$, $C'' = C_1 - C_2 + C_4$. As a consequence, $T \in \Theta(n)$.

Exercise 3

Let T be a Red-Black Tree.

Exercise 3.a

Give the definition of Red-Black Trees.

Answer

A Red-Black Tree is a binary search tree that satisfy the following *red-black properties*:

- each node is colored in *black* or *red*;
- the *root* of the tree is *black*;
- The *leaves* of the tree are `NIL` nodes and colored in *black*;
- Every red node has black children;
- Given a node of the tree, each branch of the sub-tree rooted on the given node has the same number of black nodes.

Exercise 3.b

Write the pseudo-code of an efficient procedure to compute the height of T. Prove its correctness and evaluate its asymptotic complexity.

Answer

In this solution of the exercise it is assumed that the root counts when computing the height of the tree, but not the `NIL` external nodes.
In order to solve the task, one could follow this approach:

- first, find the minimum key in the tree;
- loop over the tree nodes and whenever a leaf is found, follow the branch up to the root;
- choose the maximum length between all the branches as the height of the tree.

The pseudo-code may look as the one below.

```
1    def height(T):
2        # if root is NIL, then computing height is much easier
3        if T.root != NIL:
4            x ← minimum_in_subtree(T.root)
5            height ← 0
6            while x != NIL:
7                branch_length ← 0
8                # if a leaf is found, follow the branch up to the root
9                if x.left = NIL and x.right = NIL:
10                   y ← x
11                   while y != NIL:
12                       branch_length ← branch_length + 1
13                       y ← y.parent
14                   endwhile
15               endif
16               # get maximum branch length
17               if height < branch_length:
18                   height ← branch_length
19               endif
20               x ← successor(x)
21           endwhile
22           return height
23       endif
```

```
        # if root is NIL, height is zero
        return 0
    enddef
```

The functions `minimum_in_subtree()`, `successor()` and the members `root` and `parent` have the same meaning of those defined during classes.

Correctness of the algorithm

Consider a RBT with only one node, the root. The function `minimum_in_subtree()` will then return the root itself. In this case `T.root.left` and `T.root.right` are both `NIL` nodes, as it is `T.root.parent`. As a consequence, `branch_length` gets only to be updated once. Since initial value of `height` is smaller than `branch_length`, the latter becomes the new height of the tree. The procedure ends here since the root's successor is a `NIL` node.
Assuming now that the tree has $n > 1$ nodes and that $m \in [1, n) \cap \mathbb{N}$ have already been checked, the algorithm goes further checking the $(m+1)$-th node:

- if it has at least a non-`NIL` node, then there is no point in computing the height. In this case the algorithm keeps going with the successor of the $(m+1)$-th node.
- if both of its children are `NIL` nodes, then the branch that ends in the $(m+1)$-th node is followed up to the root, updating is length at each step. Successively, the computed `branch_length` is compared to the previously computed `height`. If the former is bigger then it becomes the new height.

Complexity of the algorithm

The function `minimum_in_subtree()` has complexity which belongs in $O(\log_2 n)$. The same holds for the complexity of `successor()`.
The inner `while` loop contains only code whose complexity belongs to $\Theta(1)$. In the worst case scenario it is repeated $\log_2 n$ times. So the complexity of this part belongs to $O(\log_2 n)$. The outer `while` loop is repeated until the node `x` has no successor, that is for n times, if n is the number of the tree nodes. The overall complexity of this part of the algorithm belongs to $O(n \log_2 n)$. Then the total comlexity belongs to $O(n \log_2 n) + O(\log_2 n)$ or, more generally, $O(n \log_2 n)$.

Exercise 3.c

Write the pseudo-code of an efficient procedure to compute the black-height of T. Prove its correctness and evaluate its asymptotic complexity.

Answer

In this solution of the exercise it is assumed that the external nodes (leaves) and the root counts when computing the black-height of the tree.
In order to solve the task, one could follow this approach: since the black-height of the RBT is equal for every branch of the tree, one could simply follow the leftmost branch (but any branch would do), increasing the count every time a black node is found.
The pseudo-code may look as the one below.

```
def black_height(T):
    # if root is NIL, then computing height is much easier
    if T.root != NIL:
        x ← T.root
        black_height ← 0
        # follow leftmost branch until leaf is found
        while x != NIL:
            # check color and update height
            if x.color = black
                black_height ← black_height + 1
            endif
            x ← x.left
```

```
            endwhile
            black_height ← black_height + 1
            return black_height
        endif
        # if root is NIL, height is zero
        return 0
    enddef
```

Correctness of the algorithm

Consider a one-noded RBT. The only node of the tree is then the root. Thus, when the color is checked, the `black_height` gets to be updated, but, since in this case `x.left = NIL`, the loop ends here, and the returned value is the `black_height` increased by one since we count in also the `NIL` leaves.
Assume now the tree is n-noded. Exploiting the fact that each branch has the same black-height, the result obtained by following the leftmost one is the same to the one obtained choosing any other branch. For each node the color is checked and if the condition `x.color = black` holds, then the `black_height` is increased by one. When `x.left = NIL` the loop ends, and since `x` ← `x.left` is now the leftmost `NIL` leaf, the black-height must again be increased by one, and then returned.

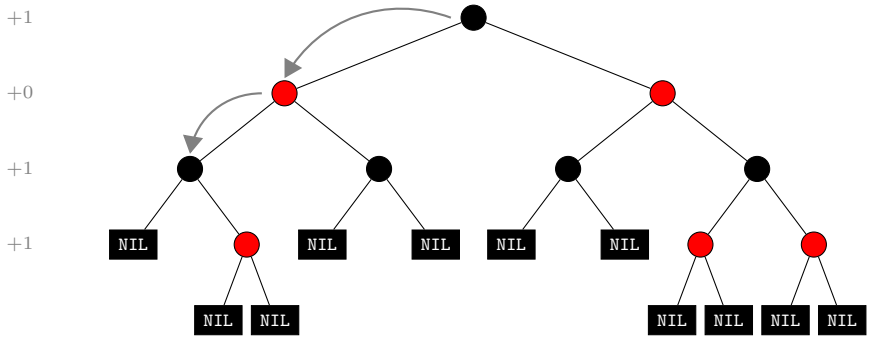


Figure 7: Computation of the black-height of a RBT.

Complexity of the algorithm

The inner `while` loop contains only code whose complexity belongs to $\Theta(1)$. In the worst case scenario the length of the branch is $\log_2 n$, where n is the number of the tree nodes. The complexity of the algorithm then belongs to $O(\log_2 n)$.

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 $b_i \leq b_{i+1}$.

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

Exercise 4.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.

Answer

One could use an array of pairs, having implemented suitably the lexicographical order and a pair class in order to handle easily the pairs of values; then use the quicksort algorithm, for example, suitably modify to sustain the new way of comparing elements.
The pairs class can be defined in such a way that, if `pair` is an instance of this class, one could retrieve the first and second element of the pair using `pair.first` and `pair.second`, respectively.
The lexicographical order function can be defined as follows.

```
1   def lexicographical_order(pair1, pair2):
2       if pair1.first = pair2.first:
3           return pair1.second <= pair2.second
4       endif
5       return pair1.first <= pair2.first
6   enddef
```

The quicksort algorithm may be modified as done below.

```
1   def quicksort(A, l = 1, r = |A|):
2       while l < r:
3           p ← partition(A, l, r, l)
4           quicksort(A, l, p-1)
5           l ← p + 1
6       endwhile
7   enddef
```

```
1   def partition(A, i, j, p):
2       swap(A, i, p)
3       (p, i) ← (i, i+1)
4
5       while i <= j:
6           if lexicographical_order(A[p], A[i]):
7               swap(A, i, j)
8               j ← j - 1
9           else:
10              i ← i + 1
11          endif
12      endwhile
13
14      swap(A, p, j)
15      return j
16  enddef
```

In the pseudo-code above it has been followed the same notation as the one used in the course material.

Complexity of the algorithm

In the worst case scenario, the `partition()` function does not split the array evenly enough: at each step, one of the two sub-arrays has null size. The complexity is then given by

$$T(n) = T(n-1) + \Theta(n).$$

Applying recursively the formula above, one obtains

$$T(n) = \sum_{i=0}^{n} \Theta(i) = \Theta\left(\sum_{i=0}^{n} i\right) = \Theta(n^2).$$

In the best case scenario, using a recursion tree, one obtains that the complexity belongs to $\Theta(n \log_2 n)$.
On average, the quicksort algorithm has complexity belonging to $\Theta(n \log_2 n)$.

Exercise 4.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 4.a? If this is the case, describe it and compute its complexity, otherwise, motivate the answer.

Answer

It is not possible to improve the efficiency of the algorithm in an effective way.
In order to make the algorithm more efficient, one could use an algorithm which exploit the fact that the integers $a_i, ..., a_n$ belong to a bounded domain. Such an algorithm could be `counting sort`, suitably modified in order to deal with pairs of integers (that is, comparison are still made between integers, first or second element, but the entire pair is stored in the new array that the algorithm returns). This helps improving the complexity (linear with respect to the dimensions of the array), but only for the first element of the pairs.
Indeed, if $a_i \in [\,1, k\,]\ \forall i \in [\,1, n\,]$ and k is constant with respect to n, in general, there will be repetitions of, at least, some of those elements. In order to correctly sort the pairs, comparisons between the b_is have to be performed only when the corresponding a_is are equal. This means that, once the modified counting sort algorithm has been applied to the first element of the pairs, hopefully a result like the one in figure 8 will be obtained.

(a_1, b_1) (a_1, b_2) (a_1, b_3)	···	(a_l, b_m)	(a_l, b_{m+1})	···	(a_k, b_{n-1})	(a_k, b_n)

Figure 8: In the above representation it is assumed that $l \in [\,1, k\,) \cap \mathbb{N}$ and $m \in [\,1, n\,) \cap \mathbb{N}$. It is also assumed that $a_1, ..., a_k$ are sorted correctly and that the a_is fill entirely their domain. Some of the first elements of the pairs are repeated, while second elements are not.

At this point it is necessary to complete the lexicographical sorting by sorting the b_is that correspond to the same first value of the pair. However, the second values of the pairs do not belong to a limited domain in general. One could use the `quicksort` algorithm again, losing generally the improvement on complexity brought by counting sort.
Alternatively, one could find the maximum and minimum for each chunk in which a value of the first element is the same apply again the counting sort algorithm. However, finding the boundaries of all the chunk will bring to the same order of complexity of the `quicksort`'s worst case, that is $O(n^2)$. In possible case is, for example, when at least one chunk has size n/c, with c constant, and the rest of the chunks have constant size (if k, the maximum number of chunks, is constant, in no way all the chunks will have constant size) but, since they occupy a $n - n/c$ long portion of the array, they must be in non-constant number.

Exercise 4.c

Assume that the condition of Exercise 4.b 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 4.a? If this is the case, describe it and compute its complexity, otherwise, motivate the answer.

Answer

In this case it is possible to improve the efficiency of the algorithm.
Since both a_is and b_is belong to a bounded domain, and assuming k and h are known quantities, a suitably modified version of the `counting sort` (like the one of the point 4.b) algorithm can be applied.
A first step would consist in sorting the array according to the first element of the pairs. This would take a complexity in $\Theta(n+k)$ as known for the `counting sort` algorithm. After this first part is completed, it is necessary to have information about the repetitions of the a_i.Moreover, it would take a linear algorithm (in $\Theta(n)$) to store, in a constant amount of memory (for example, another array of length k), the number of repetitions of each a_i. If,`rep` is an k long array for storing the repetitions, one could exploit the fact that `rep[a_i]` corresponds to the number of repetitions of the element a_i, since $a_i \in [\,1, k\,] \cap \mathbb{N}$. Thus a loop over the array, updating the number of repetitions of an element every time such element is encountered could be done as: `rep[a_i]` $\leftarrow$ `rep[a_i] + 1`, or better, following the notation used in point 4.a, `rep[A[i].first]` $\leftarrow$ `rep[A[i].first] + 1`. It is important to allocate this array with `default_value = 0` so that values in $[\,1, k\,] \cap \mathbb{N}$ that do not appear have their corresponding number of repetition set to zero.
After this, the modified counting sort could be applied on the second elements of the pairs which have same value of the first element. This procedure would take complexity in $k\Theta(n+h)$, since there are at most k chunks for which the first value of the pair does not change and k is constant with respect to n. Overall complexity would be in $\Theta([\,k+1\,]n + k[\,h+1\,])$.
The pseudo-code could possibly look as the one below.

Exercise 5

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

Exercise 5.a

Why is this assumption necessary? How relaxing this condition does affect the algorithm?

Answer

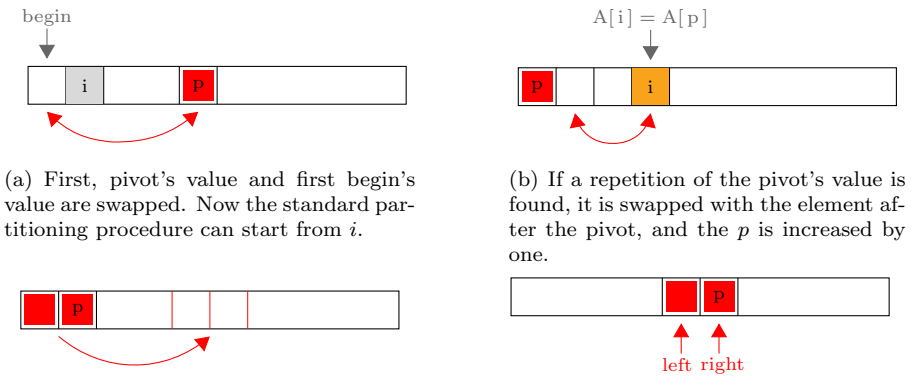
if the assumption is not respected we could end up in the case in which the pivot is a repeated value. In this case, if the element to be selected has the same value of the pivot's, unnecessary recursion calls will be performed in order to find the wanted elements. Moreover, in the worst case scenario, even if `select_pivot()` function is used, a bad choice of the pivot could be made multiple times if this value is repeated, worsening the complexity of the algorithm.

Exercise 5.b

Write the pseudo-code of an algorithm that enhance the one seen during the lessons and evaluate its complexity.

Answer

In order to improve the algorithm, both `partition()` and `select()` functions must be modified. The modification is required to make sure that during the usual steps of the `partition` algorithm procedure all the values equal to the pivot's are put at the beginning of the array, then in the middle of it, so to have all the values smaller and greater than the pivot's on the left and right, respectively, of those equal to the pivot's.



(a) First, pivot's value and first begin's value are swapped. Now the standard partitioning procedure can start from i.

(b) If a repetition of the pivot's value is found, it is swapped with the element after the pivot, and the p is increased by one.

(c) After all the values equal to the pivot's are positioned at the beginning of the array, they must be replaced after all the values smaller than the pivot's.

(d) The `partition()` function then returns the left and right boundaries of the portion of array in which all the values are equal to the pivot's.

Figure 9: Represetation of the new `partition` process.

Then, in the `select()` function, it is checked if the selected item is between `left` and `right`, and the recursive calls are made on the new portions of the array, before `left` and after `right`.
The pseudo-code may look as the one below.

```
1    def partition(A, i, j, p):
2        swap(A, i, p)
3        (p, i) ← (i, i+1)
4        reps_pivot ← 1
5        while i <= j:
```

```
            if A[i] > A[p]:
                swap(A, i, j)
                j ← j - 1
            else if A[i] < A[p]:
                i ← i + 1
            else:
                swap(A, i, p+1)
                p ← p + 1
                reps_pivot ← reps_pivot + 1
            endif
        endwhile
        right ← j
        while p > 0:
            swap(A, p, j)
            p ← p - 1
            j ← j - 1
        endwhile
        left ← j
        return left, right
    enddef
```

```
    def select(A, l = 1, r = |A|, i):
        if r - l <= 10:
            sort(A, l, r)
            return i
        endif
        j ← select_pivot(A, l, r)
        left, right ← partition(A, l, r, j)
        if i >= left and i <= right:
            return i
        endif
        if i < left:
            return select(A, l, left-1, i)
        enddef
        return select(A, right+1, r, i)
    enddef
```

In the code above the notation is the same as the one used during the lessons.

Complexity of the algorithm

The complexity of the algorithm is not different from the complexity of the original `select` algorithm, which belongs to $O(n)$. Indeed, almost all of the added and/or modified instructions have complexity in $\Theta(1)$. The only part that could not have constant complexity is the second `while` loop of the `partition()` function. In the worst case scenario, the pivot's value could be repeated a number of times which is not constant with respect to n. However, this would make the complexity of this loop belong to $\Theta(n)$ and would not affect the overall complexity.