DSA: Exercise 3

David Sermoneta

May 2, 2024

Exercise 1:

Exercise 1.a)

```
: 5 9
: 5
: 9
: LR 5 9
: 3 8 2
: 3
: 8 2
: 8
: 2
: LR 8 2
: LR 3 8
: LR 9 8
```

Exercise 1.b)

Since we call the function twice, that sets $a = 2$, and each call halves n by 2, this gives us $b = 2$. Further, we have 3 `PRINT` statements, as well as 2 operations that take constant time, thus $d = 0$. We get the following form:

$$T(n) = 2T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + 5\Theta(1).$$

Since $2 > 2^0$, we have that

$$T(n) = \Theta(n^{\log_b(a)}) = \Theta(n),$$

and we are done.

Exercise 1.c)

Apart from the obvious: getting rid of the print statements, there isn't anything else to do. We can explain this via the following reasoning. apart from the print statements, there are:

1. Base case of the recursion formula,
2. initializing of the variable k that is used in the rest of the code,
3. calling the function on the sliced list `A[1 : k` and `A[k : A.length]`.

Removing any of these, would result either in the function not ending, a code error, or a function that does not find the maximum of a list, but rather, a shorter version of the list. So the answer is, one can only remove the three print stamements to have the algorithm still find the maximum element of a given array.

Exercise 2:

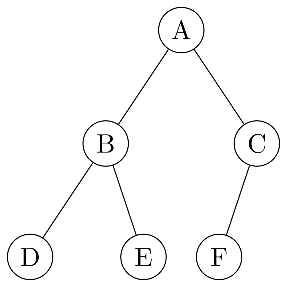


Figure 1: Graph with both inorder and preorder traversal

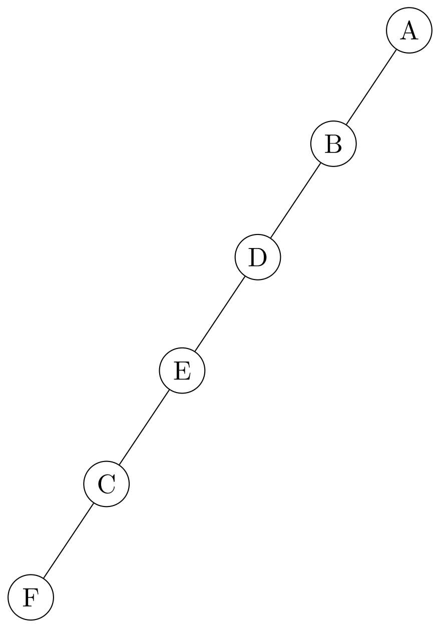


Figure 2: Graph with only preorder traversal specified, it is not the same as the one in figure 1, so the preorder traversal does not determine T uniquely.

Exercise 3:

For graph (i), we have height$(\mathsf{T}_i) = 3$, depth$(3) = 2$, and it neither FB, NC nor C.
For graph (ii), we have height$(\mathsf{T}_{ii}) = 3$, depth$(3) = 1$, and it is all of the above.
For graph (iii), we have height$(\mathsf{T}_{iii}) = 2$, depth$(3) = 2$, and it is only NC.
For graph (iv), we have height$(\mathsf{T}_{iv}) = 2$, depth$(3) = 0$, and it is only FB.

Exercise 4:

```
MERGESORT(A,1,3)
MERGESORT(A,1,2)
MERGESORT(A,1,1)
MERGESORT(A,2,2)
```

```
MERGE(A,1,1,2) → [2,6]
MERGESORT(A,3,3)
MERGE(A,1,2,3) → [2,3,6]
```

Exercise 5:

Solution to 5.a

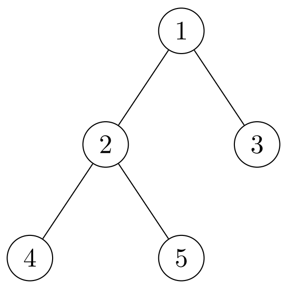


Figure 3: Array A as a binary heap tree.

Solution to 5.b

The first time upon calling `Build_Max_Heap(A)`, we get that the indices that violate the max-heap property are 2 and 3, since the parent of 2 and 3 is smaller than both respectively. That is, `A[parent(2)]`$<$ `A [2]` and same for 3. Calling on `Build_Max_Heap(A)` forces us to call on `Max_Heapify(A, 2`, and later, `Max_heapify(A,1)`, which gives us the following graphs representations of the subtrees rooted at the index we are calling (specified in the figures below): The calling

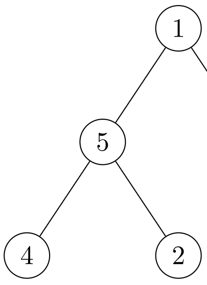
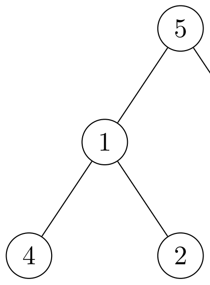
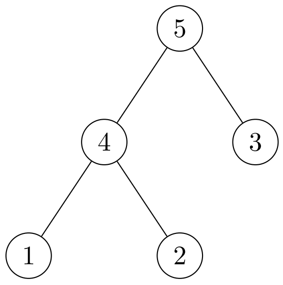


Figure 4: Array A as a binary heap tree after `Max_Heapify(A,2)`.

of `Max_Heapify(A,1)` gives us another call, namely on index 2, which will be the element 1:

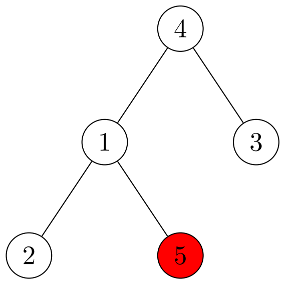


(a) Binary heap representation after call of `Max_Heapify(A,2)`. Now the index 4 and 5 violate the Max heap property, that is, the elements 4 and 2.

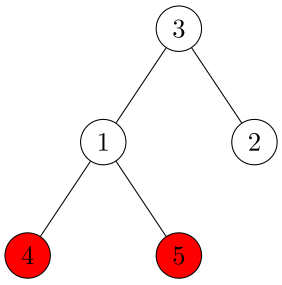


(b) Binary heap representation after call of `Max_Heapify(A,2)`. Now there are no nodes that violate the Max Heap property and thus we are done.

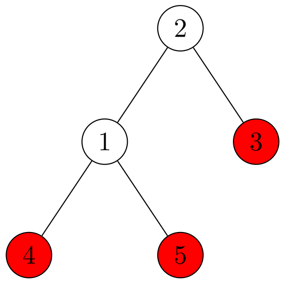
Solution to 5.c



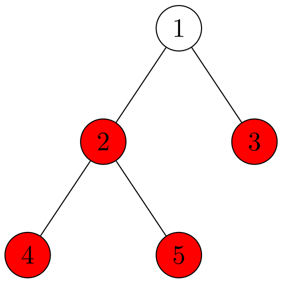
(a) `i = 5`



(b) `i = 4`



(c) `i = 3`



(d) `i = 2`

Exercise 6:

Let `A`$= \{a_1, a_2, \dots, a_n\}$ be the set of coins. If we chose to weigh half of the coins from $a_1, \dots, a_k$, where $k = \left\lfloor \frac{n}{2} \right\rfloor$, we'd expect it to weigh $k \cdot w$ if our CF coin wasn't there, and so we'll know in which half of the set it is by seeing if the weight deviates from it. Repeating this process should yield us our counterfeit (CF) coin after a finite number of iterations. In pseudocode, this becomes

```
def SCF(A, left, right)
    IF left == right: return left
    middle = left  + right // 2
    expected_value = (middle - left + 1)*$w$
    weight = Weigh(A[left : middle])
    IF expected_value != weight
        return SCF(A, left, middle)
    ELSE: return SCF(A, middle+1, right)
```

Here `Weigh(A[left : middle]` just returns the weight of the elements left to (including) middle of `A`.
When we first call the algorithm, we have that the CF coin is either in (1.) `A[left:middle]`, or in (2.) `A[middle+1:right]`. So our loop invariant is just to show that for each recursion, the CF with index i is always between `left` and `middle` if the last `IF` statement is satisfied, or between `middle+1` and `right` otherwise. The `IF` statement is only satisfied when the weight of `A[left:middle]` deviates from the expected value, meaning the CF coin is in there. So calling `SCF` with `right=middle` gives us that the loop invariant is satisfied.

If not, then that means that `expected_value == weight`, and so $\mathtt{i}$ must be in the second half of `A`, which means that in the next recursion, the CF coin will be between `middle+1` and `right`, and so when we call `SCF` with `left==middle+1`, we get that the loop invariant is also satisfied.
To find out the amount of weighings this algorithm has, suppose $|\mathtt{A}| = n$, and that $\mathtt{d}$ is the amount of weighings we did before finding the CF coin. Then since we divide `A` in two after each weighing, that must mean that we have divided n by 2^{d} times. meaning that $\frac{n}{2^{d}} = 1$ since that is when the algorithms terminates, and we get afters ome manipulation that $\mathtt{d} = \log_2(n)$. So the amount of weighings made is $\log_2(n)$ for an array of size n.