# Algorithms and Data Structures

## Exercise Session 6

https://n.ethz.ch/~ahmala/and

# Quiz

# Fancy Binary Search aka Binary Lifting

```
int ans = 0;
for (int k = /* some power of two */; k != 0; k /= 2) {
  if (condition(ans + k)) {
    ans += k;
  }
}
```

Is the runtime of Merge Sort on the input $[1,2,\ldots,n]$ $\Theta(n)$ ?

Below you see four sequences of snapshots, each obtained in consecutive steps of the execution of one of the following algorithms: `InsertionSort`, `SelectionSort`, `QuickSort`, `MergeSort`, and `BubbleSort`. For each sequence, write down the corresponding algorithm.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 6 | 5 | 1 | 2 | 4 | 8 | 7 |
| 3 | 6 | 5 | 1 | 2 | 4 | 8 | 7 |
| 3 | 5 | 6 | 1 | 2 | 4 | 8 | 7 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 6 | 5 | 1 | 2 | 4 | 8 | 7 |
| 3 | 5 | 1 | 2 | 4 | 6 | 7 | 8 |
| 3 | 1 | 2 | 4 | 5 | 6 | 7 | 8 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 6 | 5 | 1 | 2 | 4 | 8 | 7 |
| 3 | 6 | 1 | 5 | 2 | 4 | 7 | 8 |
| 1 | 3 | 5 | 6 | 2 | 4 | 7 | 8 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 6 | 5 | 1 | 2 | 4 | 8 | 7 |
| 3 | 6 | 5 | 1 | 2 | 4 | 7 | 8 |
| 3 | 6 | 5 | 1 | 2 | 4 | 7 | 8 |

**Solution:**

InsertionSort (top left) – BubbleSort (top right) – MergeSort (bottom left) – SelectionSort (bottom right).

# Stack

- pop
- push
- top

# Queue

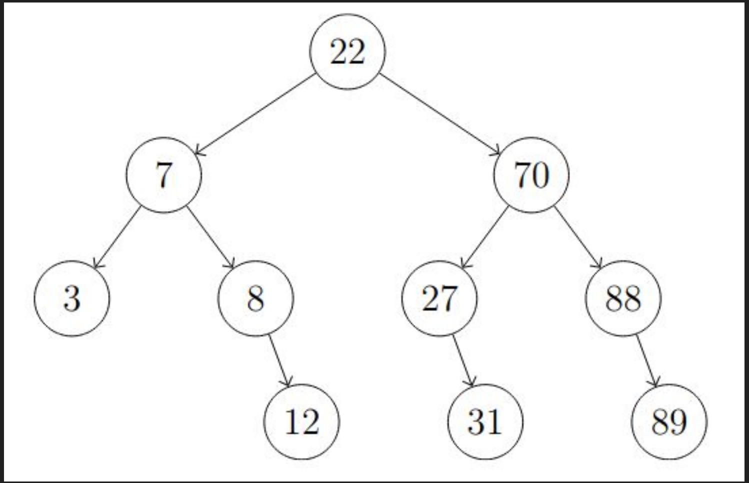
- enqueue
- dequeue

# Priority Queue

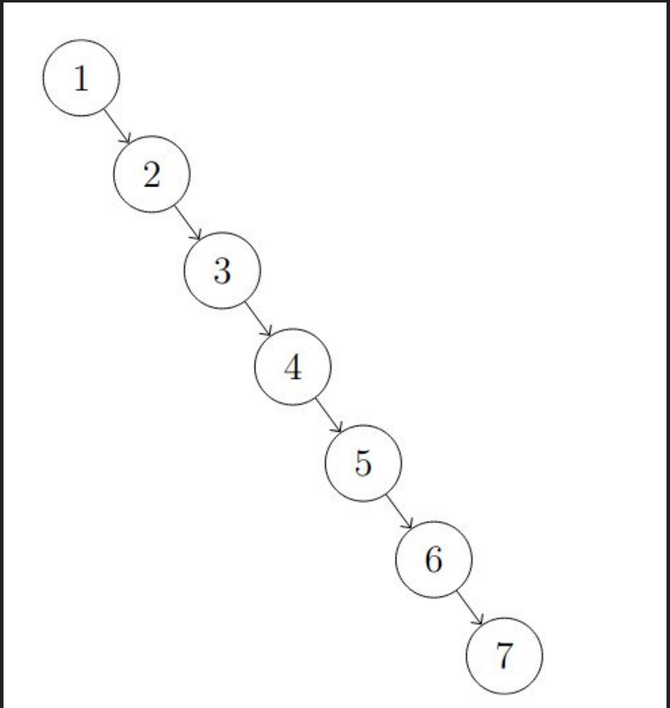
- insert
- extractMax

# Binary Search Tree

- for any node's key
  - all keys in the left subtree are smaller
  - all keys in the right subtree are larger

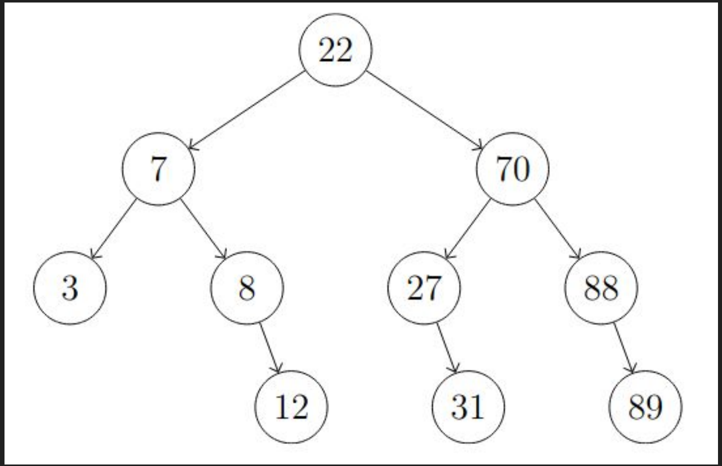


# unbalanced :(



# Binary Search Tree

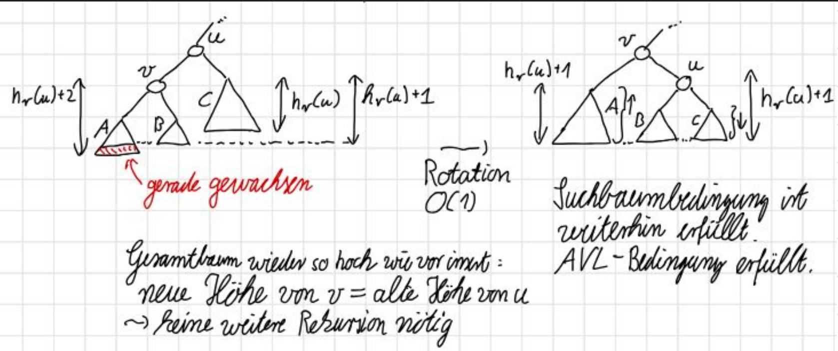
- insert(p)
- delete(p)
  - p is a leaf
  - p has a single child
  - p has two children

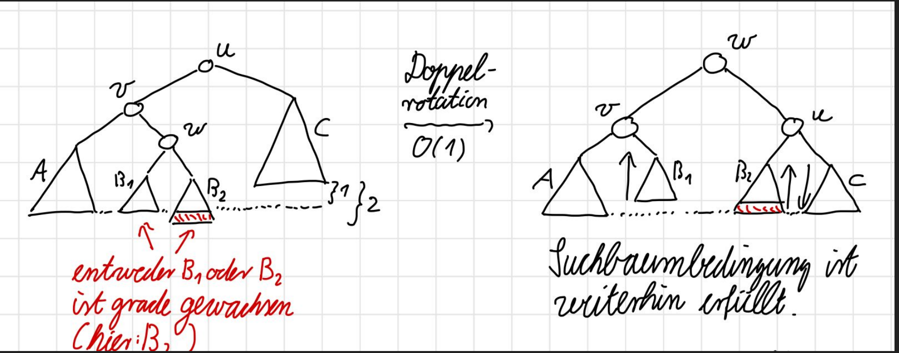


# Balanced Search Tree

- AVL Tree
  - search, insert, delete in logarithmic time

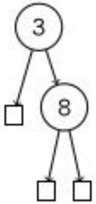
(a) Draw the tree obtained by inserting the keys 3, 8, 6, 5, 2, 9, 1 and 0 in this order into an initially empty AVL tree. Give also all the intermediate states after every insertion and before and after each rotation that is performed during the process.



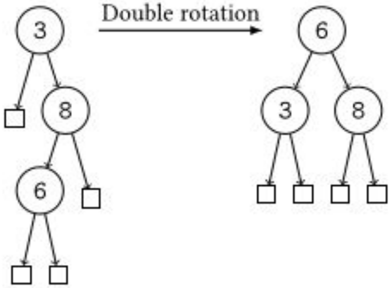



**Insert 3:**

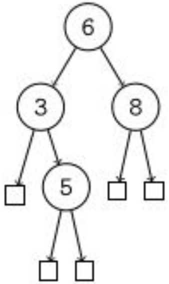

**Insert 8:**



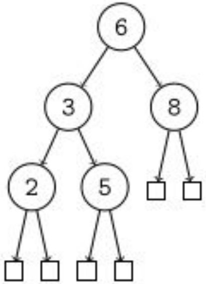

**Insert 6:**



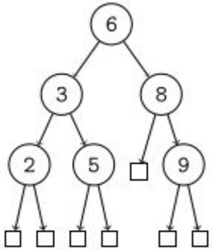

**Insert 5:**



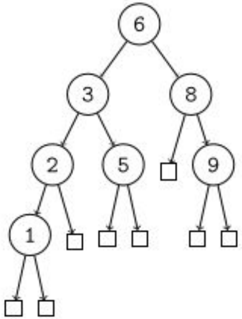

**Insert 2:**



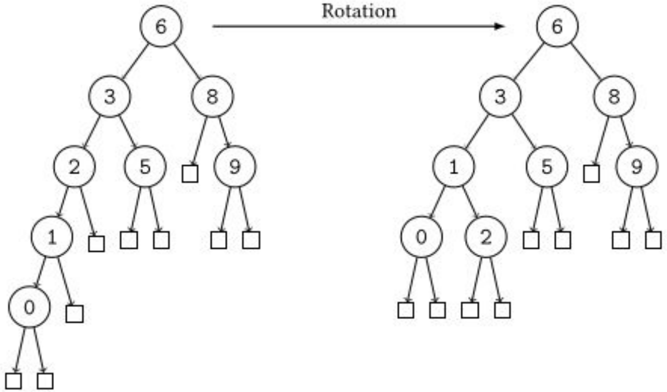
**Insert 9:**



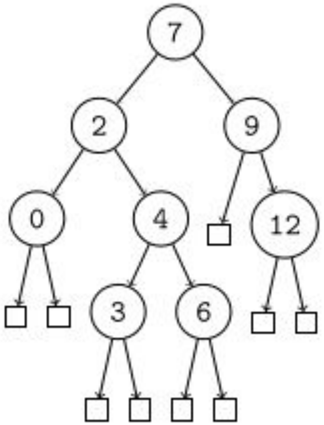

**Insert 1:**



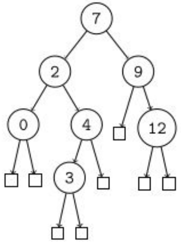

**Insert 0:**



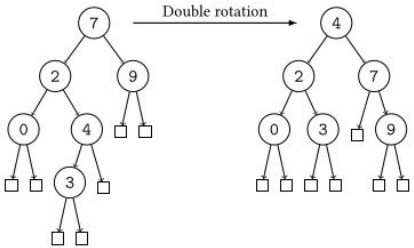
(b) Consider the following AVL tree.



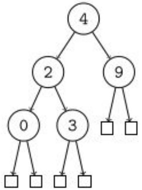

Draw the tree obtained by deleting 6, 12, 7 and 4 in this order from this tree. Give also all the intermediate states after every deletion and before and after each rotation that is performed during the process.



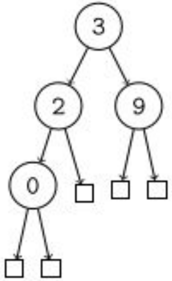
**Delete 12:**



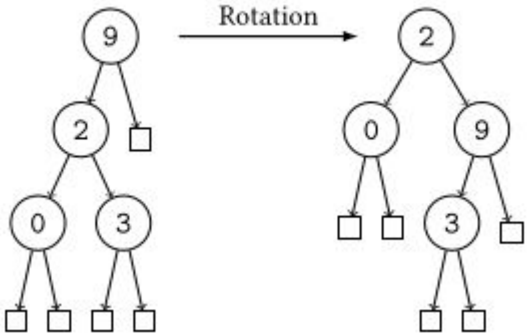

**Delete 7:**



**Delete 4:** Key 4 can either be replaced by its predecessor key, 3, or its successor key, 9. If key 4 is replaced by its predecessor:



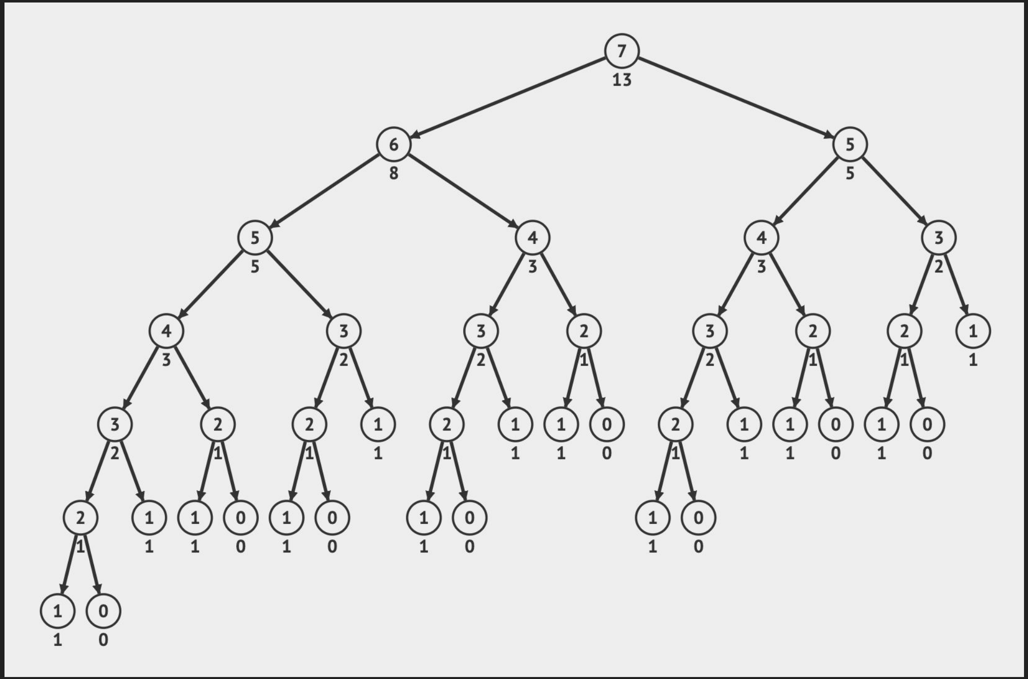

If key 4 is replaced by its successor:



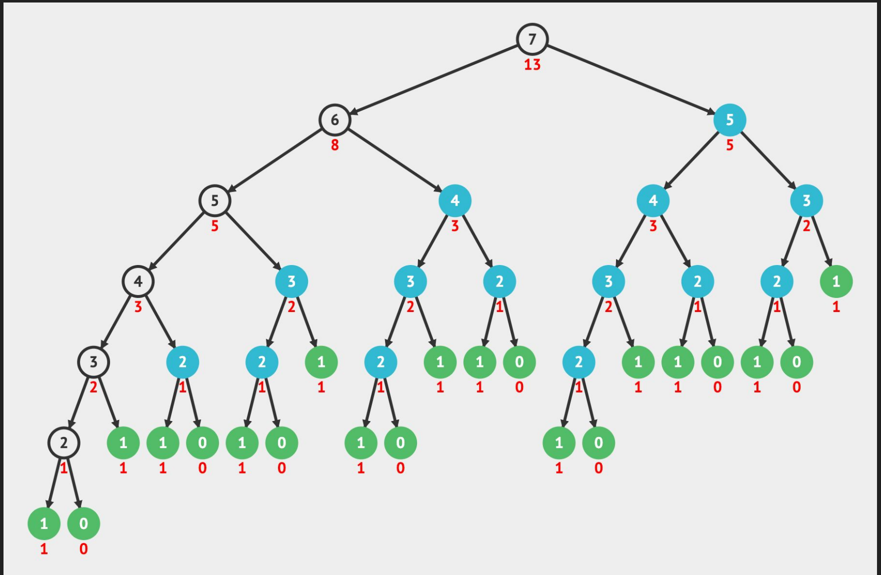
# Dynamic Programming

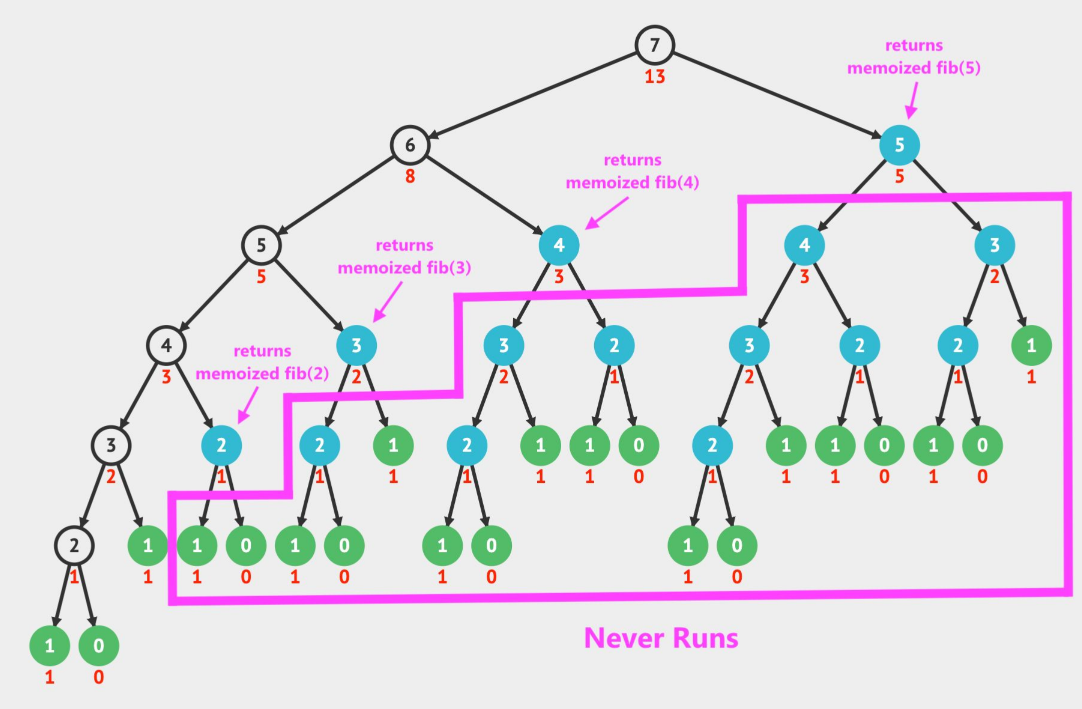
- Memoization
  - storing the result of function call
  - return the stored result when same input occurs again
- Bottom up
  - iteratively
  - starting with the smallest subproblems and building up to the main problem
- Top down
  - recursive
  - uses memoization
  - Solves subproblems as needed
  - prone to stack overflow

# Fibonacci Without Memoization



# Redundant computations





returns memoized fib(5)
returns memoized fib(4)
returns memoized fib(3)
returns memoized fib(2)
Never Runs

# Climbing Stairs

You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top? For example for n = 8 there are 8 distinct ways:

- 1 + 1 + 1 + 1 + 1
- 1 + 1 + 1 + 2
- 1 + 1 + 2 + 1
- 1 + 2 + 1 + 1
- 2 + 1 + 1 + 1
- 1 + 2 + 2
- 2 + 1 + 2
- 2 + 2 + 1

# Recursive -- (Memoization in the next slide)

```
function climbStairs(n):
    if n <= 1:
        return 1
    return climbStairs(n-1) + climbStairs(n-2)
```

```
memo = {}
function climb(i):
    if i <= 1:
        return 1
    if i in memo:
        return memo[i]
    memo[i] = climb(i-1) + climb(i-2)
    return memo[i]

return climb(n)
```

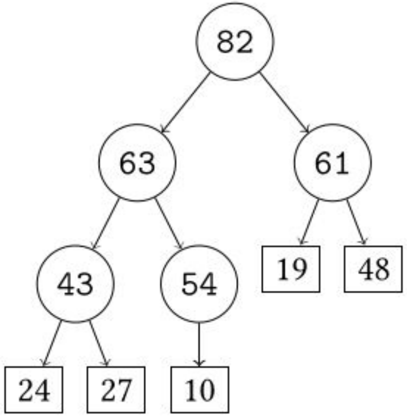
# Iterative -- Bottom Up

```
dp = [0] * (n + 1)
dp[0] = dp[1] = 1

for i in 2...n:
    dp[i] = dp[i-1] + dp[i-2]

return dp[n]
```

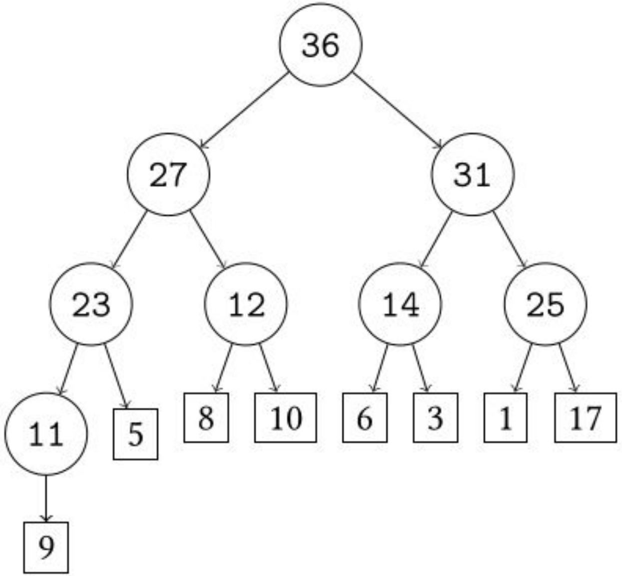
# Exercise Sheet 5

**Exercise 5.1** *Max-Heap operations* **(1 point)**.

(a) Consider the following max-heap:




Draw the max-heap after inserting the elements 70 and 51 in that order.




Draw the max-heap after two ExtractMax operations.

**Exercise 5.3** *Quick(?) sort* **(1 point)**.

Recall the pseudocode for the *quick sort* algorithm from the lecture:

---

**Algorithm 1** quick sort

---

**function** QUICKSORT($A, \ell, r$)<br>
    **if** $\ell < r$ **then**<br>
        $k =$ PARTITION$(A, \ell, r)$<br>
        QUICKSORT$(A, \ell, k-1)$<br>
        QUICKSORT$(A, k+1, r)$<br>
**function** PARTITION($A, \ell, r$)<br>
    $i \leftarrow \ell$<br>
    $j \leftarrow r - 1$<br>
    $p \leftarrow A[r]$ ▷ Choose the rightmost entry as pivot<br>
    **repeat**<br>
        **while** $i < r$ and $A[i] \leq p$ **do**<br>
            $i \leftarrow i + 1$<br>
        **while** $j \geq \ell$ and $A[j] > p$ **do**<br>
            $j \leftarrow j - 1$<br>
        **if** $i < j$ **then**<br>
            Swap $A[i]$ and $A[j]$<br>
    **until** $i > j$<br>
    Swap $A[i]$ and $A[r]$ ▷ At the end, the correct place for the pivot is $i$<br>
    Return $i$

---

We want to study the number of comparisons between array entries the quick sort algorithm performs when we apply it to an array $A[1 \ldots n]$ consisting of $n$ unique integers which is already sorted in ascending order (so $A[1] < A[2] < \ldots < A[n]$).

(a) Show that the number of comparisons $T(n)$ between array entries that QUICKSORT$(A, 1, n)$ performs when applied to a sorted array $A$ as above, and with the above rule to select the pivot satisfies the recursive relation

$$T(1) = 0, \quad T(n) = T(n-1) + (n-1) \quad \forall n \geq 2.$$

You may assume for simplicity that PARTITION$(A, \ell, r)$ always performs exactly $\ell - r$ comparisons between entries. In your argument, refer to the pseudocode above.

**Algorithm 2** Heap Construction

```
function HEAPIFY(T)
    for t = height(T) − 1, . . . , 0 do
        for nodes N at level t do
            for ℓ = t, . . . , height(T) − 1 do
                C1 ← the left child of N, if no such child exists assign it key −∞.
                C2 ← the right child of N, if no such child exists assign it key −∞.
                if key(C1) ≥ key(C2) and key(C1) > key(N) then
                    Swap the keys of nodes N and C1.
                    N ← C1
                else if key(C1) < key(C2) and key(C2) > key(N) then
                    Swap the keys of nodes N and C2.
                    N ← C2
                else
                    Exit inner for loop
```

Let $T$ be a complete binary tree consisting of $n$ nodes with $n \geq 2$. Let $H$ be the data structure that results from executing Heapify$(T)$.

(a) Prove that the executing Heapify$(T)$ returns a valid heap.

# Peer Grading

Exercise 5.3