Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Markus Püschel, David Steurer

5. November 2018

Algorithms & Data Structures — Homework 7 — HS 18

Exercise Class (Room & TA): _____

Submitted by: _____

Peer Feedback by: _____

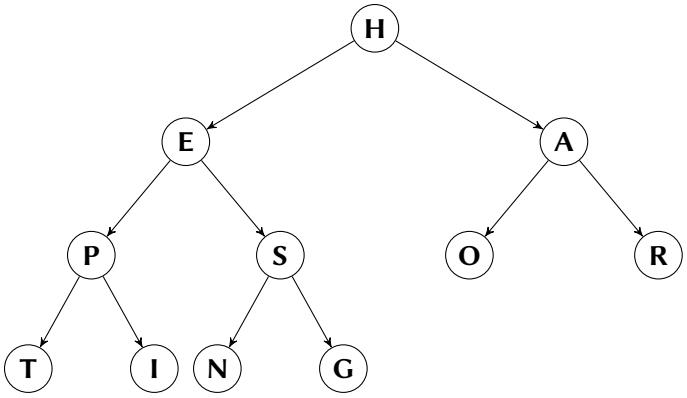
Points: ____

Exercise 7.1 *Heapsort.*

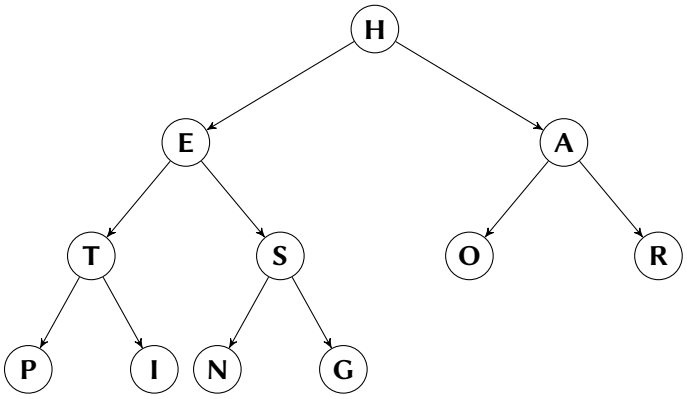
Given the array [H,E,A,P,S,O,R,T,I,N,G], we want to sort it in ascending alphabetical order using Heapsort.

1. From the lecture, you know a method to construct a heap in linear time. Draw the resulting max binary heap if this method is applied to the above array.

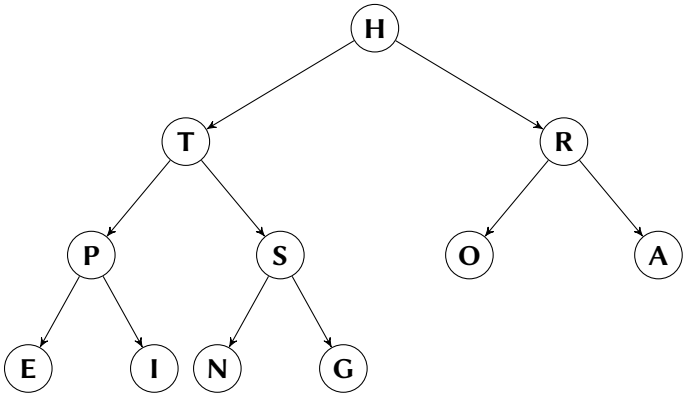
   Solution: The array [H,E,A,P,S,O,R,T,I,N,G] can be interpreted as the following binary tree:



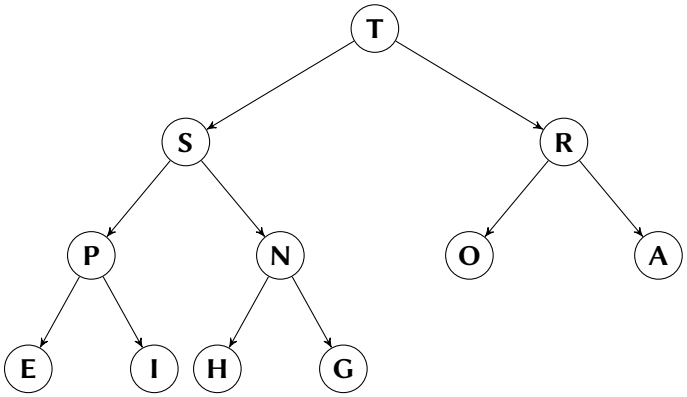
   After heapifying the subtrees with roots at level 2, the array is [H,E,A,T,S,O,R,P,I,N,G], and the binary tree looks like:



After heapifying the subtrees with roots at level 1, the array is [H,T,R,P,S,O,A,E,I,N,G], and the binary tree looks like:

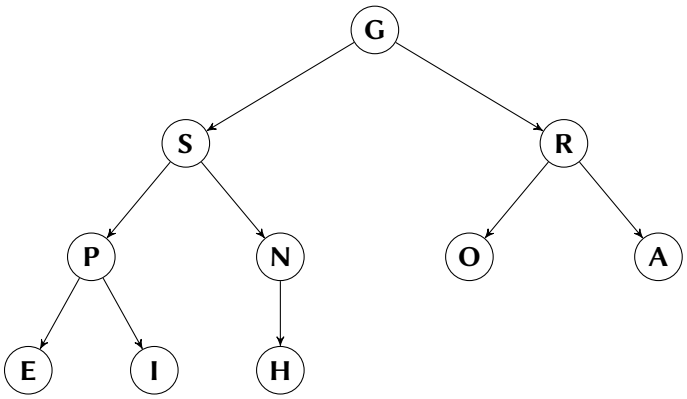


After sifting the root node down, the array is [T,S,R,P,N,O,A,E,I,H,G], and the binary tree looks like:

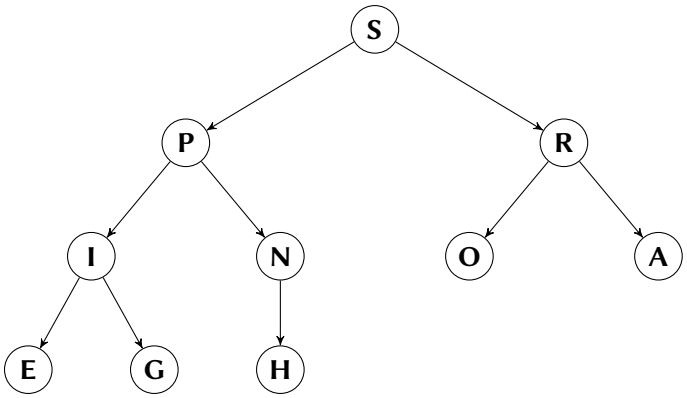


2. Sort the above array in ascending alphabetical order with heapsort, beginning with the heap that you obtained in 7.1.1. Draw the array after each intermediate step in which a key is moved to its final position.

   Solution: We begin with the max binary heap [T,S,R,P,N,O,A,E,I,H,G]. Swap T with the last element G, removing T from the heap:



   Then reheapify, sifting G downwards until it comes after its children in alphabetical order.



After reheapifying, the array is [S,P,R,I,N,O,A,E,G,H,T].

The array after the subsequent steps are as follows. Blue letters are at their final positions.

```
(a) Swap S and H:  [H,P,R,I,N,O,A,E,G,S,T]
    Sift H down:   [R,P,O,I,N,H,A,E,G,S,T]

(b) Swap R and G:  [G,P,O,I,N,H,A,E,R,S,T]
    Sift G down:   [P,N,O,I,G,H,A,E,R,S,T]

(c) Swap P and E:  [E,N,O,I,G,H,A,P,R,S,T]
    Sift E down:   [O,N,H,I,G,E,A,P,R,S,T]

(d) Swap O and A:  [A,N,H,I,G,E,O,P,R,S,T]
    Sift A down:   [N,I,H,A,G,E,O,P,R,S,T]

(e) Swap N and E:  [E,I,H,A,G,N,O,P,R,S,T]
    Sift E down:   [I,G,H,A,E,N,O,P,R,S,T]

(f) Swap I and E:  [E,G,H,A,I,N,O,P,R,S,T]
    Sift E down:   [H,G,E,A,I,N,O,P,R,S,T]

(g) Swap H and A:  [A,G,E,H,I,N,O,P,R,S,T]
    Sift A down:   [G,A,E,H,I,N,O,P,R,S,T]

(h) Swap G and E:  [E,A,G,H,I,N,O,P,R,S,T]
    Sift E down:   [E,A,G,H,I,N,O,P,R,S,T]

(i) Swap E and A:  [A,E,G,H,I,N,O,P,R,S,T]
```

We are done.

3. You are given a max binary heap with n pairwise distinct elements, stored in an array. Specify the set of array indices where the minimum element can be found. Provide a proof.

   Solution: Let A be the array storing the binary heap, with indices in the range $1, \ldots, n$. We can use the heap property from the script:

   $$(2k + 1 \leq n) \Rightarrow (A[k] \geq A[2k + 1])$$

   Since each element in the heap is distinct, this is equivalent to:

   $$(2k + 1 \leq n) \Rightarrow (A[k] > A[2k + 1])$$

Let j be the position of the minimum element within the array. Because $A[j]$ is the minimum element, $A[j] > A[2j+1]$ is always false, so $2j+1 \leq n$ must be false as well. $2j+1 > n$, so $j > \frac{n-1}{2}$.

Therefore, the minimum element of the heap must be found at an index between $\frac{n-1}{2}$ and n.

An alternate solution is that an element in a max heap must be larger than either of its children. Thus the minimum element must be a **leaf** node, so it must be in the last half of the array.

Exercise 7.2 *Quicksort* **(1 Point for 7.2.2).**

1. Given the array [Q, U, I, C, K, S, O, R, T, I, N, G], we want to sort it in ascending alphabetical order using Quicksort. When partially sorting a subarray (with procedure Aufteilen), always use the last element as the pivot. Draw the array after each intermediate step in which a key is moved to its final position.

   Solution:

   1. $[C, G, I, Q, K, S, O, R, T, I, N, U]$
   2. $[C, G, I, Q, K, S, O, R, T, I, N, U]$
   3. $[C, G, I, I, K, N, O, R, T, Q, S, U]$
   4. $[C, G, I, I, K, N, O, R, Q, S, T, U]$
   5. $[C, G, I, I, K, N, O, Q, R, S, T, U]$
   6. $[C, G, I, I, K, N, O, Q, R, S, T, U]$

2. Prove by induction on n that Quicksort (as seen in the lecture and described in the script) performs at most $2n^2$ comparisons of keys when sorting an array of n pairwise distinct numbers, where $n \geq 0$ is a non-negative integer.

   Solution: We will prove buy indiction over n that the number of comparisons is at most $2n^2$.

   Base case. If $n = 0$, $2n^2 = 0$ and Quicksort does not perform any comparisons.

   Inductive hypothesis. Assume that for all $k < n$ Quicksort performs at most $2k^2$ comparisons of keys when sorting an array of $k > 0$ elements.

   Inductive step. Let $n > 0$. The first partition performs at most $2n$ comparisons of elements:

   - There are $n - 1$ elements that need to be partitioned.
   - Whenever an element is compared to the pivot and not swapped, it will never again be compared to the pivot.
   - Otherwise, the element gets swapped and will then be compared in the other while loop as the first element, but not swapped anymore, and thus not get compared anymore to the pivot.
   - This makes so far at most $2n - 2$ comparisons.
   - The remaining two comparisons happen right before the procedure Aufteilen terminates: the last comparison in either while loop can involve an element already compared and possibly swapped earlier.

Let x and y be the sizes of the subarrays after the partition. By induction hypothesis, Quicksort performs at most $2n + 2x^2 + 2y^2$ comparisons. Since $x + y + 1 = n$,

$$\begin{aligned} 2n + 2x^2 + 2y^2 &= 2n + 2x^2 + 2(n - 1 - x)^2 \\ &= 2n + 2x^2 + 2n^2 + 2x^2 + 2 - 4n - 4nx + 4x \\ &= 2n^2 + 4x(x + 1 - n) - 4n + 2 \\ &\leq 2n^2 - 4n + 2 \\ &= 2(n - 1)^2 \leq 2n^2 \end{aligned}$$

Where the third line follows since for all natural numbers $n > 1$ and x, such that $0 \leq x \leq n - 1$, the following inequality holds:

$$x + 1 - n \leq 0.$$

Therefore, Quicksort performs at most $2n^2$ comparisons of keys when sorting an array of n elements.

3. You and your friend have a competition to see who can implement the fastest sorting algorithm. You choose to implement Heapsort, and your friend chooses to implement Quicksort. By peaking at her code, you noticed that your friend's implementation of Quicksort always chooses the element at index $\lfloor \frac{k}{2} \rfloor$ as the pivot when sorting a subarray of length k. (That is, the additional very first step of Aufteilen is to swap the elements in positions $\lfloor \frac{k}{2} \rfloor$ and r, where r is the last index of the subarray.)

   Given an array with elements $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$, what order maximizes the number of comparisons that your friend's Quicksort algorithm must perform? Give a solution for how to arrange the elements of an array of length n, so that you will always win the competition when n is large enough.

   Solution: In order to maximize the number of comparisions, we will ensure that the maximum element is always chosen to be the pivot.

   Consider our example of ten elements. The pivot positions will be 5, 4, 4, 3, 3, 2, 2, 1, 1. For easier notation, let A_i denote the array containing the elements after round i of Quicksort, and let A_0 give the input order. Hence, $A_0[5] = 10$, $A_1[4] = 9$, $A_2[4] = 8$, and so on. When starting with $A_0 = [3, 5, 7, 9, 10, 2, 6, 1, 8, 4]$ we get exactly what we need.

   We shall describe a *recursive construction* for general n. First, observe the following:

   - The only swaps we want to happen are the inital swap of Aufteilen, when the pivot is placed to the last position un the (sub)array.
   - When n is odd, the pivot positions are $\frac{(n-1)}{2}$, $\frac{(n-1)}{2}$, $\frac{(n-1)}{2} - 1$, $\frac{(n-1)}{2} - 1$, ..., 1, 1;
   - When n is even, then the positions are $\frac{n}{2}$, $\frac{n}{2} - 1$, $\frac{n}{2} - 1$, $\frac{n}{2} - 2$, $\frac{n}{2} - 2$, ..., 1, 1.
   - When a position is for the first time a pivot position, then it contains the same element as in the input A_0.
   - When a position is for the second time the pivot position, then it contains the element that was swapped with the former pivot.
   - In the first $\lceil n/2 \rceil$ rounds, every element is thus swapped at most once.

   The first $\lceil n/2 \rceil$ rounds are as follows:

Case: n is even.

We thus start with $A_0 = [3, 5, 7, 9, 11, \ldots, (n-1), n, \_, \_, \_, \ldots]$.

Then, for instance, in the third round, we want that the pivot is $n-2$. This element has been swapped with element $n-1$ in the previous iteration, and thus $A_0[n-1] = n-2$. Similarly, in the $2i+1$-th round with $i \geq 1$, we want that the pivot is $n-2i$. This element has been swapped with element $n-2i+1$ in the previous iteration, and thus $A_0[n-2i+1] = n-2i$.

Hence, $A_0 = [3, 5, 7, 9, 11, \ldots, (n-1), n, \ldots, (n-6), X_{k-2}, (n-4), X_{k-1}, (n-2), X_k]$, where $k = \lceil \frac{n}{4} \rceil$.

Thus, after $\frac{n}{2}$ rounds, we have

$A_{n/2} = [3, 5, 7, 9, 11, \ldots, (n - \frac{n}{2}), X_1, \ldots, X_k, (n - \frac{n}{2} + 1), (n - \frac{n}{2} + 2), \ldots, n]$.

Hence, it remains to continue with a subarray of size $n - n/2$, having the same structure as the input array A_0:

$A' = [3, 5, 7, 9, 11, \ldots, (n - n/2), X_1, \ldots, X_k]$.

Case: n is odd.

By the above argumentation and for the pivot positions when n is odd:

$A_0 = [3, 5, 7, 9, 11, \ldots, n, \ldots, X_{k-2}, (n-5), X_{k-1}, (n-3), X_k, (n-1)]$, where $k = \lceil \frac{n}{4} \rceil$.

Thus, after $\lceil \frac{n}{2} \rceil$ rounds, we have

$A_{\lceil \frac{n}{2} \rceil} = [3, 5, 7, 9, 11, \ldots, (n - \lceil \frac{n}{2} \rceil), X_1, \ldots, X_k, (n - \lceil \frac{n}{2} \rceil + 1), (n - \lceil \frac{n}{2} \rceil + 2), \ldots, n]$.

Hence, it remains to continue with a subarray of size $n - \lceil n/2 \rceil$, having the same structure as the input array A_0:

$A' = [3, 5, 7, 9, 11, \ldots, (n - \lceil \frac{n}{2} \rceil), X_1, \ldots, X_k]$.

Exercise 7.3 *Resistors* **(1 Point)**.

When assembling electronic circuits, it is important to make sure that the used resistors have exactly the right values in order to guarantee the correct functionality of the circuit. Since one does not always have all possible resistor values, one often uses a series connection of multiple resistors to achieve a certain total resistance. For example, one can produce a total resistance $R = 27\Omega$ by the two resistors $R_1 = 12\Omega$ and $R_2 = 15\Omega$ connected in series.

Develop an algorithm that decides for a given box full of n resistors, whether it is possible to achieve a certain resistance R by a series connection of at most 2 resistors. The algorithm should be as fast as possible, but only use concepts that you have learned in this class. Express the worst-case complexity of the algorithm in $\mathcal{O}$ notation.

Solution: Given a resistor of resistance R_1, if there is a resistor in our box of resistance $R - R_1$, then the pair of resistors can be used to achieve the desired resistance. If we have a sorted array of resistors, we can use a **binary search** to determine if such a resistor exists.

This suggests the following algorithm: First scan through the array to see if a single resistor fufills our needs. Then sort the array, and for every resistor, search to see if there is another resistor such that their sum is the target resistance. Pseudocode that implements this algorithm is as follows:

```
Boolean resistancePossible(double resistances, double R)
{
    for(int i = 0; i < resistances.length(); i++) {
        if(resistances[i] == R)
            return true;
    }

    heapSort(resistances);
    for(int i = 0; i < resistances.length(); i++) {
        //Binary search returns -1 if the element isn't found,
        //and the index otherwise.
        if(binarySearch(R - resistances[i], 0, resistances.length()-1) > -1)
            return true;
    }
    return false;
}
```

Checking to see if a single resistor works takes $\mathcal{O}(n)$ time. HeapSort, costs $\mathcal{O}(n \log n)$. Each binary search takes $\mathcal{O}(\log n)$ time, so the loop over the binary search costs $\mathcal{O}(n \log n)$. Thus, the entire operation costs $\mathcal{O}(n \log n)$

An alternate solution is to maintain two pointers: One starting at the beginning of the array, and one starting at the end. When the sum is of the elements pointed to by both pointers is less than R, we increment the left pointer, and when it is greater than R, we decrement the right pointer. This operation costs $\mathcal{O}(n)$ if the array is initially sorted, but because the array must be sorted, the algorithm still costs $\mathcal{O}(n \log n)$.

Exercise 7.4 *Coin Collecting* **(1 Point)**.

You are an avid coin collector and you have a huge bucket of coins in front of you. You are interested in getting the oldest coins out of the bucket to put into a display case.

There are n coins in your bucket, and you want to place the m oldest coins into your display case, where m is much smaller than n. Develop an algorithm to find the m oldest coins. The algorithm should be as fast as possible (in the worst case, as always), but only use concepts that you have learned in this class. Express the worst-case complexity of the algorithm in $\mathcal{O}$ notation.

Solution: The solution is to maintain a priority queue of size m.

Take m coins out of the bucket and arrange them in a **min-heap**, so each coin is **newer** than its children. Then take the rest of the coins out of the bucket one at a time. For each coin, compare its age to the coin on top of the heap. If the coin taken from the bucket is newer than the coin on top of the heap, it is newer than all of the coins in the heap, so we do nothing. If the coin taken from the bucket is older than the coin on top of the heap, replace the coin on top of the heap with the coin from the bucket, and sift the newly-found coin down into the heap.

The inital heapify operation costs $\mathcal{O}(m)$. Adding a coin to the priority queue costs $\mathcal{O}(\log m)$, since it is a heap of size m. We may add a coin to the priority queue $n - m$ times, for a total complexity of $\mathcal{O}(m + (n - m) \log m) = \mathcal{O}(n \log m - m \log m + m)$. Since we know that m is much smaller than m, we can write this as $\mathcal{O}(n \log m)$.

Submission: On Monday, 12.11.2018, hand in your solution to your TA *before* the exercise class starts.