Prof. Dr. Goran Glavaš,
M.Sc. Fabian David Schmidt
M.Sc. Benedikt Ebing
Lecture Chair XII for Natural Language Processing, Universität Würzburg

1. Exercise for "Algorithmen, KI & Data Science 1"

24.11.2023

1 Heap(sort) & Priority Queue

1. What are the minimum and maximum numbers of elements in a heap of height h?

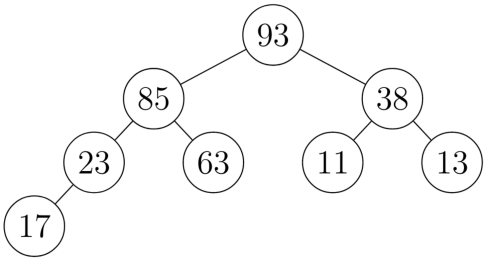
   The height h of node n is the number of edges along the longest simple upward path up to n. For instance, the height of a single node is 0 and for node 0 in a min-heap $A = [0, 1, 2, 3]$ is 2.

   Consider the complete min-heap with array representation $A = [0, ..., 6]$ that has a height $h = 2$. The ith level adds at least 1 (i.e. 1 node in bottom left of heap) and at most 2^i nodes. For $h = 2$, we therefore have $\sum_{i=0}^{1} 2^i = 3$ nodes in the complete binary tree at $h - 1 = 1$. Consequently, we have at least $(\sum_{i=0}^{1} 2^i) + 1$ and at most $(\sum_{i=0}^{2} 2^i)$ nodes in a min heap of height h. These series denoting the upper and lower bound of the number of nodes in a min heap of height h can be simplified to 2^h and $2^{h+1} - 1$, respectively.

2. Explain briefly why the array with values $[23; 17; 14; 6; 13; 10; 1; 5; 7; 12]$ is (is not) a max-heap?

   If we write down the max-heap for the above array, we see that the node with key 7 is a child of the node with key 6, violating the max-heap priority. Therefore, the array is not a valid max-heap.

3. The tree below contains 8 items, where each stored item is an integer which is its own key.



Suppose the tree drawn above is the implicit tree of a binary max-heap H. State the array representation of H, first **before** and then **after** performing the operation `H.delete_max()`.

We initially write down the operations required to delete the maximum.

a) We swap 93 with 17 and pop 93, the root node now is 17. The tree violates the max-heap property. We perform `H.heapify`.

b) In the first step, 17 needs to be swapped with 85, as 85 is the maximum of the keys in consideration $[17, 38, 85]$. The new tree still violates the max-heap priority.

c) In the second step, 17 needs to be swapped with 85, as 63 is the maximum of the keys in consideration $[17, 38, 63]$. The new tree now fulfills the max-heap priority.

d) We now deduce the array representation of the new tree, reading the elements in left-to-right by level top-to-bottom from the binary tree.

The new array representation is $[85; 63; 38; 23; 17; 11; 13]$.

4. Implement a `MinHeap(Queue)` in the `exercise03.ipynb` notebook as per the requirements laid out in the notebook.

2 Hashing

1. Insert integer keys $A = [67, 13, 49, 24, 40, 33, 58]$ in order into a hash table of size 9 using the hash function $h(k) = (11k + 4)\ mod\ 9$. Collisions should be resolved via chaining, where collisions are stored at the end of a chain. Elaborate how the resulting hash table looks like.

```
A = [67, 13, 13, 49, 24, 40, 33, 58 ]
hash_fn = lambda k: (11 * k + 9) % 9
```

```
hash_table = {}
for a in A:
    hash = hash_fn(a)
    if not hash in hash_table:
        hash_table[hash] = []
    hash_table[hash].append(a)

>> hash_table
>> # form of {hash: chain}, where has is int and chain is list
>> {3: [67, 13, 49, 40, 58], 7: [24, 33]}
```

2. Explain why the use of the following functions f_1 and f_2 as hashing functions for a hash table of size m of whole numbers (ganze Zahlen) is problematic. m is prime.

   $f_1(x) = \bar{x}$ where $\bar{x}$ is the digit sum (Quersumme) of x
   $f_2(x) = \lfloor \frac{x}{m} \rfloor$ mod m where $\lfloor y \rfloor$ denotes the floor or y, e.g. $\lfloor 4.3 \rfloor = 4$

   a) The digit sum of whole numbers is unbounded and therefore grows for abritrarily large numbers indefinitely. Thus, values larger than m cannot be used for hashing. Furthermore, the hash of many nearby numbers is identical, e.g. 12 and 21 which both have a digit sum of 3.

   b) The floor division causes many neighboring numbers to have identical hashes, causing frequent collisions. Even more so, the mod operation now causes even more conflicts.

3. The following two Python functions correctly solve the problem: given an array X of n positive integers, where the maximum integer in X is k, return the integer that appears the most times in X. Assume: a Python `list` is implemented using a dynamic array; a Python `dict` is implemented using a hash table which randomly chooses hash functions from a universal hash family; and `max(X)` returns the maximum integer in array X in worst-case $O(|X|)$ time. For each function, state its worst-case and expected running times in terms of n and k.

```
def frequenttest(X):
    H = {}
    best = X[0]
    for x in X:
        if not x in H:
            H[x] = 0
        H[x] += 1
        if H[x] > H[best]:
```

```
            best = x
    return best
```

Consider the inner workings of this function.

```
for x in X:
    H[x] = 0
```

We initially build a dictionary of (zero) counts, by assigning each unique number in our array (keys) an initial value of 0 (values).

```
best = X[0]
```

The above line sets the initially maximum value to the first element in X. This may or may not be true, but we will have to check all unique elements anyways.

```
for x in X:
    H[x] += 1
    if H[x] > H[best]:
        best = x
```

We now iterate over our array. For each value x in X, we increment the value of the key x in H by 1. Should the counts of x now exceed the counts of our current `best`, we set `best` to x. This algorithm works, as we keep track of the current best and only the counts of x can exceed the counts of `best` in the current loop.

The dictionary (i.e. has table) implementation has worst case $O(n^2)$ and expected runtimes $O(n)$. The worst case stems from the fact that in the very worst scenario, for every access of a key there is a hash collision, which is why we'd have to iterate over the chain which at most could be of length n. Each `H[key]` denotes an access into the hash table. Even more so, for each access, in the worst case we'd have to iterate on average to half the length of the chain, resulting in quadratic complexity. In practice, the length of the chain of course matters, though in the *worst* case, we'd always have hash collisions with n unique numbers, resulting in iterating n times over chains of lenght $n/2$, on average. The expected case would be not to have hash collisions. In this scenario, the runtime is linear in n and thus $O(n)$.

```
def frequenttest(X):
    k = max(X)
    A = []
    for i in range(k + 1):
        A.append(0)
    best = X[0]
    for x in X:
        A[x] += 1
```

```
        if A[x] > A[best]:
            best = x
    return best
```

The dynamic array implementation has expected and worst case runtime of $0(n+k)$.

```
A = []
for i in range(k + 1):
    A.append(0)
```

k cost stems from building the initial dynamic array where the offset corresponds to a unique number and its value reflects the count in X. Note that we may construct unnecessarily long arrays as the length solely depends on the maximum value. For instance, X may only comprise 3 values, e.g. $[1, 10^6, 10^12]$. The algorithm however builds a dynamic array of length $|10^{12}|$

```
best = X[0]
for x in X:
    A[x] += 1
    if A[x] > A[best]:
        best = x
```

Identifying the maximum now mirrors the dictionary implementation, but each access into `A` is always $O(1)$. As we do that n times, these comparisions are $O(n)$.