Weijia Xu

Professor Arun

Algorithms & Structured Programming CS455

Oct.17, 2023

1. **(7 pts) Consider a generalization of a MAX-Heap in which the tree is a nearly-balanced k-ary tree. (k=2 is the (binary) MAX-Heap done in class. Now k can be 2, 3, 4, ...) As with binary max-heaps the term "nearly-balanced" means that the tree is filled except possibly in the last level, in which the leftmost values may be filled. Suppose the MAX-Heap is "linearized" into an array, just as for k=2 (see class slides).**

* **(2 pts) Write out the formulae to compute--for a given node represented by index i in the array--its parent, and its jth child, j=1, ..., k.**

Parent of node i: (i-1)/k – integer division

jth child of node i(j = 1,2,….,k): (k \* i) + j

* **(3 pts) What is the worst-case running time of MAX-Heapify(A, 1)? For this analysis, interpret k as a possibly increasing function of n, the number of elements in the heap. Explain your answer.**

In the MAX – Heapify operation, we campare the value of a node with its children and potentially swap it with one of ite children to maintain the heap property. In a k-ary heap, we have k children for each node, and we need to compare the node with all its children to determine the maximum and perform the swap if necessary. So, in the worst case, we perform comparisons with all k children for each node down the height of the tree, which is log\_k(n). Therefore, the number of comparisons can be expressed as T(n) = O(k\*log\_k(n)).

As we can see the interpreting k as a function of n, if k grows slowly with respect to n, then the time complexity of MAX-Heapify will be dominated by the logarithmic term, and the worst-case running time will be similar to that of a binary heap(k=2).

* **(2 pts) Based on your analysis, is there any value of k, possibly a function of n, for which the worst-case time for MAX-Heapify(A, 1) is asymptotically better than the worst-case time for MAX-Heapify(A,1) when k is 2? Explain which value and why if so, and why not if not.**

To find a value of k for which the worst-case time for MAX-Heapify is asymptotically better than k=2, we need to consider the growth rate of k relative to n.

If k grows at a slower rate than n, then the worst-case time complexity T(n) = O(log\_k(n)) would be better than T(n) = O(log\_2(n)). In this case, for large values of n, the k-ary heap would have a shallower height compared to the binary heap,leading to faster MAX- Heapity times. However, if k grows at a rate equal to or faster then n, then the worst-case time complexity T(n) = O(log\_k(n)) would not provide an asymptotic improvement over T(n) = O (log\_2(n)). Both k-ary and binary heaps would have similar heights, resulting in similar MAX-Heapify times.

1. **(8 pts) Imagine nonnegative integers arriving one by one in an input stream forever. When the next integer arrives, you need to output the number of times this integer has been in the stream till and including then. So when a number arrives for the first time, you should output 1, for the second time output 2 and so on.**

* **(3) Describe a good approach for this problem.**

One of the most intuitive and efficient approaches would be to use a dictionary or a hash table. Each time a number arrives: 1. Check if the number is in the dictionary. 2. If it’s not, add it with a value of 1. 3. If it is, increment the value associated with this number. When we output the frequency of an arriving number, simply look it up in the dictionary and return its associated value.

* **(2) Explain what is good about this approach.**

Speed: Most of the time, the average case time complexity for insertions and lookups in a hash table is O(1), which means constant time irrespective of how many elements are in the hash. Thus, even if millions of numbers have arrived, determining the frequency of the next one can often be done in constant time.

Flexibility: Hash tables are inherently designed to handle cases where you have unique keys and need to associate them with some values.

* **(3) Consider a batch version of this problem in which the numbers don't arrive in a stream but are stored in an array containing n elements. The goal in this batch version is to be able to process this array in such a way that subsequently, any number can be input and the approach outputs its frequency in the array sufficiently fast. Can you exploit the fact that now you know all the numbers upfront (even though n may be very large) to improve your approach. If so, explain in what ways your approach is an improvement.**

Knowing all the numbers upfront does give us the ability to optimize our approach. For instance:

Preprocessing: the first iterate through the array to compute the frequency of each number. This can be done in O(n) time. Once this is done, queries can be answered in O(1) average time using the precomputed hash table.

Optimized memory: if the array has a limited range or notice that there’s a large chunk of numbers that aren’t used, we can create a more compact representation of our hash table or even consider using other data structures like balanced binary search trees. It would allow for O(log n) insertion and lookup times, but they can be more memory efficient in cases where the numbers are sparse.

1. **(13 pts) Consider an array of n nonnegative integers. n can be arbitrarily large but assume that the number of distinct integers in the array cannot exceed a fixed constant K. (This does not constrain the integer values, they can be any.) We are interested in finding the number which has the highest frequency in this array.**
   1. **(3 pts) Describe an approach that is good for this problem. It must work correctly.**

Initialize an empty dictionary. Iterate through each integer in the array: if the integer is not in the dictionary, add it with a value of 1.If the integer is in the dictionary, increment its value by 1. After processing all integers in the array, find the key with the maximum value in the dictionary. This key is the integer with the highest frequency.

* 1. **(2 pts) Explain what is good about your approach.**

Speed and efficiency: Dictionaries in Python are implemented as hash tables.Inserting a new key value pair and updating an existing key’s value both operate in average case O(1) time complexity.

Scalability: Even as n becomes very large, as long as K remains constant, the dictionary will only have at most K key-value pairs. This makes our approach memory efficient.

Direct Retrieval: Dictionaries allow for direct retrieval of frequency by just querying the number key.

* 1. **(3 pts) Implement this approach in Python. You can use any packages that Python supports for the type of data structures that we have done in class.**

1. def most\_frequent(arr):
2. freq\_dict = {}
3. for num in arr:
4. if num in freq\_dict:
5. freq\_dict[num] += 1
6. else:
7. freq\_dict[num] = 1
8. # Find the number with the highest frequency
9. return max(freq\_dict, key=freq\_dict.get)
   1. **(5 pts) Test your implementation on a simulated array of n nonnegative integers, for sufficiently large n. Ensure that the number with the highest-frequency in your data set is unique.**
      1. **(2) Explain how you generated the synthetic data of the type we want in the array.**

Use Python’s random module to generate a list of nonnegative integers. Assume K = 10 and n = 106

* + 1. **(3) Create one array, run your method on it to find the most frequent element, and run a much simpler method, e.g. based on Python dictionaries to find the same. Assert that both give the same results. Also time the running times of the two methods and report them.**

