COSC 302: Analysis of Algorithms Lecture — Spring 2018
Prof. Darren Strash
Colgate University

Worksheet 6 — Red-black trees and hashing (with solutions)

1. Describe how to modify Bucket-Sort to have worst-case $\Theta(n \lg n)$ time.

   Solution: Instead of sorting each bucket using Insertion-Sort, sort each bucket with Merge-Sort. Then the overall run time for sorting buckets is

   $$\sum_{i=0}^{n-1} O(n_i \lg n_i) = O\left(\sum_{i=0}^{n-1} n_i \lg n_i\right) = O\left(\sum_{i=0}^{n-1} n_i \lg n\right) = O\left((\lg n)\sum_{i=0}^{n-1} n_i\right) = O(n \lg n).$$

   And therefore the worst-case running time is $O(n + n \lg n) = O(n \lg n)$. Note that the worst-case running time is also $\Omega(n \lg n)$, since if all n elements map to the same bucket, then Merge-Sort takes time $\Theta(n \lg n)$ time.

2. Let $a, b \in \mathbb{R}$. Describe how to modify Bucket-Sort to sort elements drawn from $[a, b)$ uniformly at random.

   Solution: We remap a given value $x \in [a, b)$ to a new value $y \in [0, 1)$, which we can then use to map to a bucket in bucket sort. We remap by first subtracting a, giving us a value in $[0, b - a)$, and then normalizing by dividing by $b - a$, to give a value in $[0, 1)$. Then $y = \frac{x-a}{b-a} \in [0, 1)$.

3. *(Dynamic selection.)* Describe how to maintain a dynamic collection of elements so that querying for the i-th smallest number in the collection is fast. What is the worst-case query time of your data structure?

   Solution: We store the elements in a red-black tree. For each node in the tree, we store the number of elements in its subtree. For a node x, let $n(x)$ be the number of elements in x's subtree and if x is an external node Nil, then $n(x) = 0$. Then we search for the i-th smallest value as follows:

Algorithm 1 Compute the weighted median.

proc Dynamic-Select(x, i)
while $n(x.\text{left}) \neq i - 1$
    if $n(x.\text{left}) > i - 1$ **then**
        $x = x.\text{left}$
    else
        $i = i - (n(x.\text{left}) + 1)$
        $x = x.\text{right}$
return x.value

Note that inserting or deleting in a red-black tree only involves at most 3 rotations, and each rotation can update the subtree counts in $O(1)$ time.

4. Let K be a set of $k > 1$ (non-integer) numbers, and suppose you are given an array $A[1..n]$ of numbers from K. Describe an algorithm to sort A in $O(n \lg k)$ worst-case time.

   Solution: We iterate over elements of A, and insert them into a modified red-black tree. In this red-black tree, we insert keys along with the number of occurrences of that key in A. This way, the tree contains $\Theta(\min\{n, k\})$ elements. This takes time $O(n \lg k)$. We then perform an inorder traversal on the tree, copying over each key in the tree to an output array B, inserting a number of copies equivalent to the number of occurrences of that key in A. This final step takes time $O(n)$ time since there is no more than n elements in the tree. The overall running time is $O(n \log k)$

5. Suppose you have a hash table with load factor $\alpha = 0.999999$. Which method of collision resolution would you expect to be best in practice, open addressing or chaining?

   Solution: We would expect chaining to be better in practice. We expect a chain to have length $1+\alpha < 2$, however, a probing sequence has expected length $\frac{1}{1-\alpha} = \frac{1}{0.000001} = 1000000$.

6. Describe why linear and quadratic probing do not meet the qualifications for uniform hashing.

   Solution: The uniform hashing assumption is that any given key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1\rangle$ as its probe sequence. For both linear probing and quadratic probing, there are only m different permutations, and therefore not all $m!$ permutations are equally likely. To illustrated, linear probing has permutations $\langle 0, 1, \ldots, m-2, m-1\rangle$, $\langle 1, 2, \ldots, m-1, 0\rangle$, ..., $\langle m-1, 0, \ldots m-2\rangle$.

7. Describe how to modify chaining to ensure worst-case $O(\lg n)$ search time.

   Solution: Instead of chaining with linked lists, store colliding keys in red-black trees. Then when searching, we no longer have to iterate through a linked list, but perform a search in a red-black tree. In the worse case, all keys collide to the same slot, and therefore search takes $\Theta(\lg n)$ time.

8. What is the expected insert and delete time of your method described above?

   Solution: The expected number of keys that hash to the same slot is $O(1)$; therefore, insertions and deletions are performed on red-black trees of expected size $O(1)$, which take $O(1)$ time.