CMSC 420:Fall 2019 Dave Mount

Practice Problems for Midterm 2

Midterm Exam 2 will be in class on **Thu, Nov 21**. The exam will be closed-book, closed-notes, but you will be allowed two sheets of notes, front and back (handwritten or typeset, your choice). Please plan to bring your university ID with you during the exam.

Disclaimer: These practice problems have been extracted from old homework assignments and exams. Material changes from semester to semester. These do **not** necessarily reflect the actual coverage, difficulty, or length of the midterm exam.

Problem 1. Short answer questions. Except where noted, explanations are not required, but may be given for helping with partial credit.

(a) Of the following dictionary structures, which could be used to answer the following query efficiently (say in $O(\log n)$ time or faster) in the worst case. Given a key x, find the three smallest items in the dictionary whose key value is greater than or equal to x. Pick any/all that apply: Sorted array, standard (unbalanced) binary search tree, balanced binary search tree (e.g., AVL, red-black, or 2-3 tree), splay tree, hashing. (Explain briefly.)

(b) In class, we mentioned that when using double hashing, it is important that the second hash function $g(x)$ should not share any common divisors with the table size m. What might go wrong if this were not the case?

(c) In the B+ tree data structure, each leaf node stores a pointer to the next leaf in the sorted sequence. Why was this done?

(d) Suppose you have a key consisting of k components $x = (c_0, c_1, \ldots, c_{k-1})$. Assuming that Horner's rule is used, how many additions and how many multiplications are needed to compute the hash function $h(x)$ assuming that polynomial hashing is used. (Express your answer as an exact, not asymptotic, formula of k. Don't include the final operation "mod m" in your count.)

(e) In hashing, what is *secondary clustering*? Among the primary collision resolution methods in open addressing (linear probing, quadratic probing, and double hashing) which one is most susceptible to secondary clustering and which one is least susceptible?

(f) What is the maximum number of points that can be stored in a 3-dimensional point quadtree of height h? Express your answer as an exact (not asymptotic) function of h. (Hint: It may be useful to recall the formula for any $c > 1$, $\sum_{i=0}^{m} c^i = (c^{m+1}) - 1)/(c - 1)$.)

Problem 2. Let T be a binary tree (not necessarily a binary search tree) with the usual fields `key`, `left` and `right`, where the keys are of some numeric type (integer, float, or double). Give pseudo-code for a procedure `T.makeSum()` that replaces the `key` field of each node in the tree with the sum of the key fields of itself and all its descendants in the tree (see Fig. 1(a)). Your algorithm should run in time $O(n)$, where n is the tree's size.

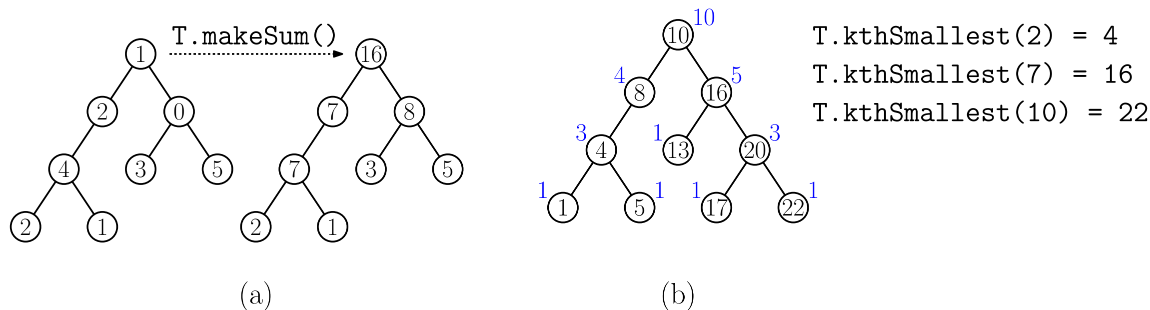

Figure 1: (a) Problem 2 (`makeSum`) and (b) Problem 3 (`kthSmallest`).

Problem 3. Let T be a binary search tree with the usual fields `key`, `left` and `right`, and in addition each node contains a field `size`, which indicates the number of nodes in the subtree rooted at this node. Give pseudo-code for a procedure `T.kthSmallest(int k)` that returns the value of the kth smallest element of T (see Fig. 1(b)). Thus, `T.kthSmallest(1)` returns the minimum element of the tree and `T.kthSmallest(n/2)` returns the median element of the keys. Your algorithm should run in time proportional to the height of the tree.

Problem 4. In class we demonstrated a simple idea for deleting keys from a hash table with open addressing. Namely, whenever a key is deleted, we stored a special value "`deleted`" in this cell of the table. It indicates that this cell contained a deleted key. The cell may be used for future insertions, but unlike "`empty`" cells, when the probe sequence searching for a key encounters such a location, it should continue the search.

Suppose that we are using *linear probing* in our hashing system. Describe an alternative approach, which does not use the "`deleted`" value. Instead it moves the table entries around to fill any holes caused by a deleted items. In addition to explaining your new method, justify that dictionary operations are still performed correctly. (For example, you have not accidentally moved any key to a cell where it cannot be found!)

Problem 5. In class we showed that for a balanced kd-tree with n points in the real plane (that is, in 2-dimensional space), any *axis-parallel line* intersects at most $O(\sqrt{n})$ cells of the tree.

The purpose of this problem is to show that does not apply to lines that are not axis-parallel. Show that for every n, there exists a set of points P in the real plane, a kd-tree of height $O(\log n)$ storing the points of P, and a line ℓ, such that *every* cell of the kd-tree intersects this line.

Problem 6. In applications where there is a trade-off to be faced, a common query involves a set called the *Pareto maxima*.[1] Given a set of 2-dimensional data, we say that a point q *dominates* another point q' if $q_x > q'_x$ and $q_y > q'_y$. The set of points of P that are not

[1] To motivate this, suppose that you are a policy maker and you have set of energy technologies to chose from a (coal, nuclear, wind, solar) where each has an associated cost of deployment and environmental impact. Some alternatives are inexpensive to deploy but have a high negative impact on the environment, and others are more expensive to deploy but have a lower impact on the environment. Clearly, we are not interested in any technology that is "dominated" by another technology that is both less expensive and has a lower environmental impact.

dominated by any other point of P are called the *Pareto maxima* (the highlighted points of Fig. 2(a)). As seen in the figure, these points naturally define a "staircase" shape.

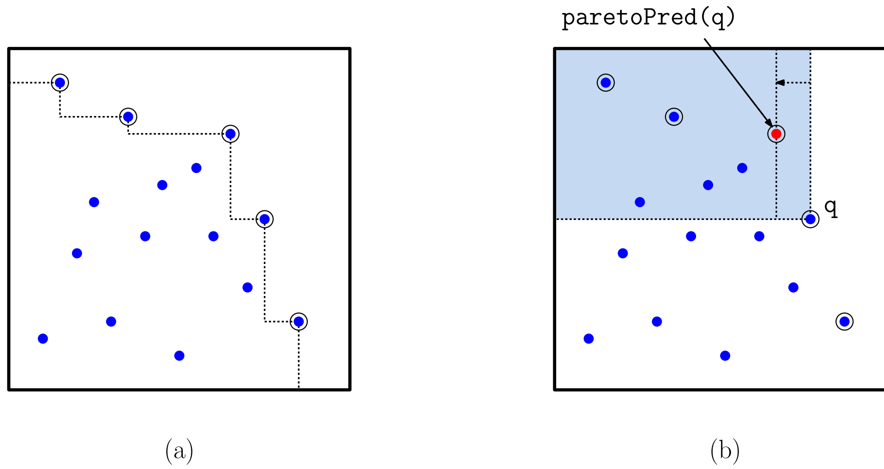

Figure 2: (a) the Pareto maxima and (b) the Pareto predecessor.

Given a 2-dimensional point set P and a query point $q = (q_x, q_y)$ define q's *Pareto predecessor* to the point $(x, y) \in P$ such that $x \leq q_x$, $y \geq q_y$, and among all such points, x is maximum. An more visual way of think about the Pareto predecessor is as the rightmost point in the subset of P lying in q's northwest quadrant (see Fig. 2(b)).

Assuming that the points of P are stored in a kd-tree T, present pseudo-code for a function `T.paretoPred(Point q)`, which returns the Pareto predecessor of a query point q.

Hint: The recursive function to compute the predecessor has the following structure:

```
Point paretoPred(Point q, KDNode p, Rectangle cell, Point best),
```

where `q` is the query point, `p` is the current node of the kd-tree being visited, `cell` is the rectangular cell associated with the current node, and `best` is the rightmost point encountered so far in the search that satisfies the Pareto criteria.