CMSC 420: Spring 2021

Homework 2: Hashing and Geometric Search

Handed out Thu, Apr 15. Due **Mon, Apr 26, 11pm**. (Solutions will be discussed in class on Tue, Apr 27, so late submissions will not be accepted after the start of class.)

Problem 1. (12 points) In this problem, you will show the result of inserting a sequence of three keys into a hash table, using linear and quadratic probing. (Each is inserted one after the other.) In each case, at a minimum you should indicate the following:

- Was the insertion successful? (The insertion fails if the probe sequence loops infinitely without finding an empty slot.)
- Show contents of the hash table after inserting all three keys.
- For each case, give a count of the number of *probes*, that is, the number of entries in the hash table that were accessed in order to find an empty slot in which to perform the insertion. (The initial access counts as a probe, so this number is at least 1. For example, in Fig. 3 in the Lecture 14 LaTeX lecture notes, `insert("z")` makes 1 probe and `insert("t")` makes 4 probes.)

For the purposes of assigning partial credit, you can illustrate the actual probes that were performed, as we did in Fig. 4 from Lecture 14. But be sure to also list the probe count.

(1.1) (6 points) Show the results of inserting the keys "`X`", "`Y`", and "`Z`" into the hash table shown in Fig. 1(a), assuming that conflicts are resolved using *linear probing*.

(a) Linear probing

```
insert("X")  h("X") = 13
insert("Y")  h("Y") = 15
insert("Z")  h("Z") = 9
```

(b) Quadratic probing

```
insert("M")  h("M") = 3
insert("D")  h("D") = 6
insert("Q")  h("Q") = 9
```

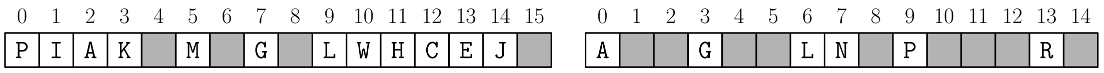

Figure 1: Hashing with linear and quadratic probing.

(1.2) (6 points) Show the results of inserting the keys "`M`", "`D`", and "`Q`" into the hash table shown in Fig. 1(b) using *quadratic probing*. (Hint: If you are unsure whether quadratic probing has gone into an infinite loop, it may be useful to note that for any positive integer i, $i^2 \bmod 15 \in \{0, 1, 4, 6, 9, 10\}$.)

(Intermediate results are not required, but may be given to help assigning partial credit.)

Problem 2. (8 points) You have a hash table of size m into which you insert n keys using separate chaining (as described in class). The keys are integers from the set $U = \{1, 2, \ldots, nm\}$.

(2.1) (5 points) Prove that, no matter how ingeniously you design your hash function, there *must exist* a subset S of U of size at least n such that every key in S hashes to the same location in the hash table.

(2.2) (3 points) What does (2.1) imply about the *worst-case* running time needed to insert n keys drawn from U into your hash table (again, assuming separate chaining).

Problem 3. (20 points) You are given a 2-dimensional point kd-tree, as described in class, where we assume that the cutting dimension alternates between x and y with each level of the tree. This tree stores a set P of n points in $\mathbb{R}^2$. For each of the following parts, the query object is a square, represented by its center point $q = (q_x, q_y)$ and radius $r > 0$. Let $S(q, r)$ denote a square consisting of the points that lie within a square of side length $2r$ that is centered at q (see Fig. 2(a)). Formally, $S(q, r)$ defined to be the set of points $p \in \mathbb{R}^2$ such that

$$q_x - r \;\leq\; p_x \leq q_x + r \quad \text{and} \quad q_y - r \;\leq\; p_y \leq q_y + r.$$

(Note that if a point is on the boundary of the square, it is considered as lying within the square.) Equivalently, we can define the *max distance* between two points p and q, denoted max-dist(p, q) to be $\max(|p_x - q_x|, |p_y - q_y|)$. The square $S(q, r)$ consist of all p such that max-dist$(p, q) \leq r$.

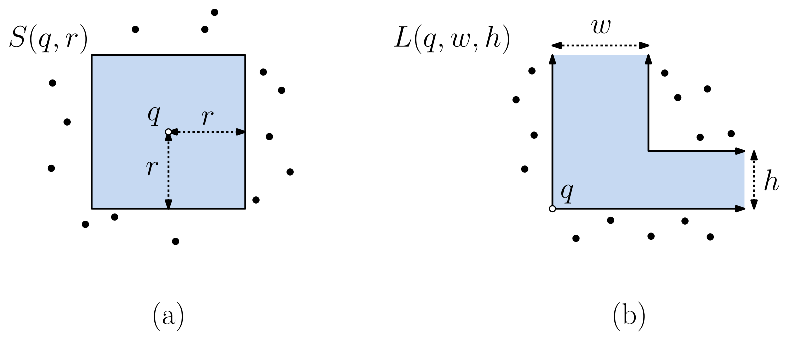

Figure 2: Square and L-shaped emptiness queries in a kd-tree.

(3.1) (6 points) Derive an efficient function

```
boolean emptySquare(Point2D q, float r)
```

which determines whether $P \cap S(q, r) = \emptyset$ (see Fig. 2(a)). That is, it returns `true` if no point of P lies within $S(q, r)$ and false otherwise. By "efficient" we mean that the query time should be $O(\sqrt{n})$, assuming that the tree is balanced, where n is the number of points in the tree. I would suggest using a recursive helper function:

```
boolean emptySquare(Point2D q, float r, KDNode p, Rectangle2D cell)
```

You may assume that any primitive geometric operations involving points, squares, and rectangles can be computed in constant time. For example, if your pseudocode, you can write "if square $S(q, r)$ and rectangle R are disjoint then ..." without explaining how to determine this.

Briefly explain your algorithm and present pseudocode.

(3.2) (4 points) Derive the running time of your function from (3.1), under the assumptions that the tree contains n points, the splitting dimension alternates between x and y, and the tree is balanced.

Hint: It may help to first review the analysis of the orthogonal range-search algorithm from the latex lecture notes. As we did in class, you may make the idealized assumption that the left and right subtrees of each node have equal numbers of points.

(3.3) (6 points) Given a point $q \in \mathbb{R}^2$ and two positive floats w and h, define the *L-shaped region* $L(q, w, h)$ to be the set of points lying within the "L"-shaped region whose lower left corner is q and which extends infinitely upwards and rightwards. The width of the vertical part is w, and the height of the horizontal part is h (see Fig. 2(b)). As above, points lying on the boundary of the region are considered to lie within the region.

Present pseudocode for an efficient function

```
boolean emptyL(Point2D q, float w, float h)
```

which determines whether $P \cap L(q, w, h) = \emptyset$.

Briefly explain your algorithm and present pseudocode. I would suggest the recursive helper function:

```
boolean emptyL(Point2D q, float w, float h, KDNode p, Rectangle2D cell)
```

(3.4) (4 points) Derive the running time of your function from (2.3), under the same assumptions as in (2.2). A complete analysis is not needed. You can explain what modifications are needed to the analysis of (2.2).

Note: You may assume the results proven in class, without the need to prove them again. If you need to modify these results, you need only explain the modifications.

Problem 4. (10 points) In this problem, we will consider how to use/modify range trees to answer two related queries. Throughout, the input set P is a set of n points in $\mathbb{R}^2$ (Fig. 3(a)). While the answer should be based on range trees, you may need to make modifications including possibly transforming the points and even adding additional coordinates. In each case, describe the points that are stored in the range tree and how the search process works. An English explanation (as opposed to pseudocode) is sufficient. Justify your algorithm's correctness and derive its running time.

(4.1) (5 points) In a *vertical segment sliding*, the query is given a vertical line segment, specified by its lower endpoint $q = (q_x, q_y)$ and its height h (see Fig. 3(b)). The query returns the first point $p_i \in P$ that is first hit if we slide the segment to its right. If no point of P is hit, the query returns `null`.

Describe how to preprocess the point set P in a data structure so that given any query (q, h), segment sliding queries can be answered efficiently. Your data structure should use $O(n \log n)$ storage and answer queries in $O(\log^2 n)$ time.

(4.2) (5 points) A *min-V query* is defined by $q = (q_x, q_y)$. Consider the V-shape defined between two rays emanating upwards from q, one with slope $+1$ and one with slope -1 (see Fig. 3(c)). Among all the points of P lying within this V-shape, the answer is the one with the smallest y-coordinate. If no point of P lies within the shape, the query returns `null`. Your data structure should use $O(n \log^2 n)$ storage and answer queries in $O(\log^3 n)$ time.

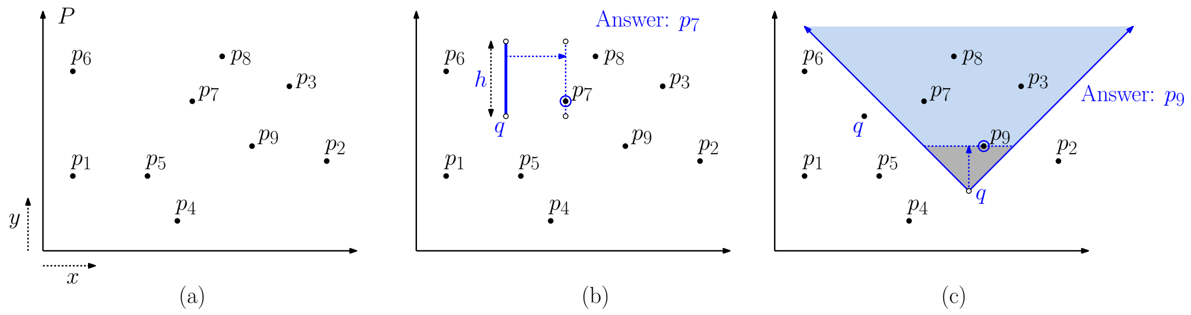

Figure 3: Vertical segment-sliding and min-V queries.

Challenge Problem: You just started work at your new company, "Dilbert's Pretty Good Data Structures". Your team has been tasked to implement a new "automatically initializing" array. The array `table` is specified by its size $m > 0$ and an initialization function f. For simplicity, let's assume that the entries of `table` are integers. The initial value of `table[i]` is defined to be $f(i)$. For example, if $f(i) = i^2$, the array's initial "virtual" contents are as shown in Fig. 4. You may assume that f can be computed in constant time.

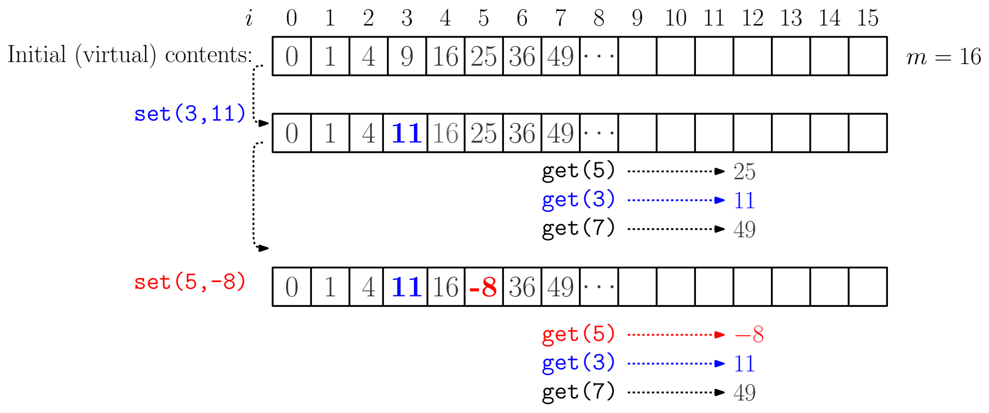

Figure 4: Auto-initializing array for $m = 16$ and $f(i) = i^2$. The initial actual (not virtual) contents are entirely unpredictable.

Your part of the project is to implement two accessor functions `void set(int i, int x)` and `int get(i)`. The first function sets `table[i] = x`. For the second function, if entry `table[i]` has been set by a previous `set` command, then its (latest) set value is returned. If its value has not yet been set, it returns $f(i)$. In both cases, you may assume that $0 \leq i < m$.

This would be easy, if you were allowed $O(m)$ time to initialize the array, but you are not. The initial contents of the array are entirely unpredictable (and may have even been set maliciously to trick your algorithm). In spite of this, each function must run in worst-case $O(1)$ time, independent of the value of m or the number n of elements that have already been set.

Your team leader has informed you that you may use additional storage of size up to $O(m)$ in which you may store *any* auxiliary data structure you like (limited to primitive data structures such as queues and stacks or one of the data structures we have seen this semester). But, as with the table, you cannot assume that this additional storage comes initialized. It too may contain garbage.

Explain how to produce a data structure to solve this problem in the required time and space. Explain your algorithm and derive its running time.

Hint: You will definitely need the additional storage to solve the problem in the stated time bound. For full credit, your answer should be deterministic (not randomized) and the worst-case running time is $O(1)$. For partial credit, show how to perform the operations in amortized $O(1)$ time, or randomized $O(1)$ time (correct with high probability) or alternatively in deterministic $O(\log m)$ time. Note that if you attempt to apply a solution based on hashing, you need to take into consideration the time needed to initialize the hash table.