CMSC 420:Fall 2020 Dave Mount

Practice Problems for the Final Exam

Like the Midterm Exam, the Final Exam will be asynchronous and online. The exam will be made available through Gradescope for a 48-hour period starting at **12:00am the morning of Wed, Dec 16** and running through **11:59pm the evening of Thu, Dec 17** (Eastern Time). The exam is designed to be taken over a 2-hour time period, but to allow time for scanning and uploading, you will have **2.5 hours** to submit the exam through Gradescope once you start it. The exam will be open-book, open-notes, open-Internet, but it must be done on your own without the aid of other people or software. (You may use a simple arithmetic calculator, but I don't expect that you will need one.)

Disclaimer: This just reflects the material since the second midterm. 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 0. Since the exam is comprehensive, please look back over the previous homework assignments, the midterm exam, and the midterm practice problems. You should expect at least one problem that involves tracing through an algorithm or construction given in class.

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

(a) Let T be extended binary search tree (that is, one having internal and external nodes). You visit the nodes of T according to one of the standard traversals (preorder, postorder, or inorder). Which of the following statements is necessarily true? (Select all that apply.)

(i) In a *postorder traversal*, all the external nodes appear in the order *before* any of the internal nodes

(ii) In a *preorder traversal*, all the internal nodes appear in the order *after* any of the external nodes

(iii) In an *inorder traversal*, internal and external node *alternate* with each other

(iv) None of the above is true

(b) You have an AVL tree containing n keys, and you insert a new key. As a function of n, what is the maximum number of rotations that might be needed as part of this operation? (A double rotation is counted as two rotations.) Explain briefly.

(c) Repeat (b) in the case of deletion. (Give your answer as an asymptotic function of n.)

(d) Suppose you know that a very small fraction of the keys in a data structure are to be accessed most of the time, but you do not know which these keys are. Among the data structures we have seen this semester, which would be best for this situation? Explain briefly.

(e) 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?

(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 $c > 1$, $\sum_{i=0}^{m} c^i = (c^{m+1})-1)/(c-1)$.)

(g) We have n uniformly distributed points in the unit square, with no duplicate x- or y-coordinates. Suppose we insert these points into a kd-tree in *random* order (see the left side of Fig. 1). As in class, we assume that the cutting dimension alternates between x and y. As a function of n what is the expected height of the tree? (No explanation needed.)

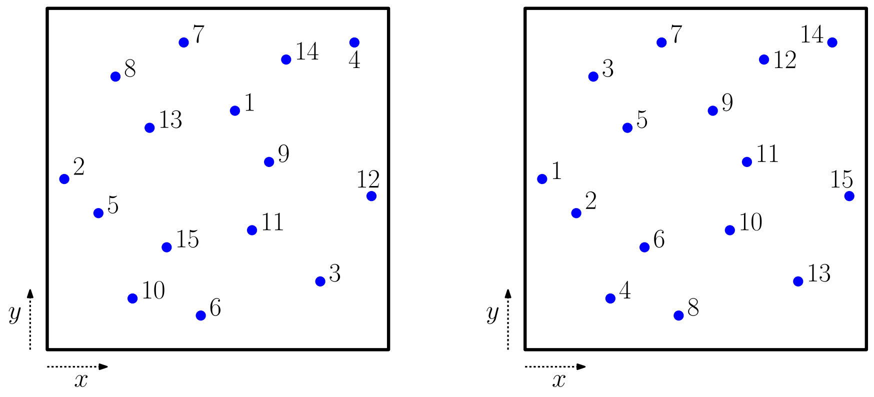


Figure 1: Height of kd-tree.

(h) Same as the previous problem, but suppose that we insert points in *ascending* order of x-coordinates, but the y-coordinates are *random* (see the right side of Fig. 1). What is the expected height of the tree? (No explanation needed.)

(i) Between the classical dynamic storage allocation algorithm (with arbitrary-sized blocks) or the buddy system (with blocks of size power of 2) which is more susceptible to *internal fragmentation*? Explain briefly.

Problem 2. In our simple binary-search tree implementations, we assumed that each node stores just a key-value pair (`p.key` and `p.value`) and pointers to the node's left and right children (`p.left` and `p.right`). In practice, it is often useful to store additional information including the following:

- `p.parent`: p's parent, or `null` if p is the root
- `p.min`: The smallest key in the subtree rooted at p
- `p.max`: The largest key in the subtree rooted at p
- `p.size`: The total number of nodes (including p) in p's subtree

Modify the *right rotation* pseudo-code so that (in addition to the rotation) all of the above associated values are updated. (**Note**: Remember that the return value is significant and should remain q.)

Problem 3. Define a new treap operation, `expose(Key x)`. It finds the key `x` in the tree (throwing an exception if not found), sets its priority to $-\infty$ (or more practically `Integer.MIN_VALUE`), and then restores the treap's priority order through rotations. (Observe that the node containing `x` will be rotated to the root of the tree.) Present pseudo-code for this operation.

Problem 4. In scapegoat trees, we showed that if `size(u.child)/size(u)` $\leq \frac{2}{3}$ for every node of a tree, then the tree's height is at most $\log_{3/2} n$. In this problem, we will generalize this condition to:

$$\frac{\texttt{size(u.child)}}{\texttt{size(u)}} \;\leq\; \alpha, \qquad (*)$$

for some constant α.

(a) Why does it **not** make sense to set α larger than 1 or smaller than $\frac{1}{2}$?

(b) If every node of an n-node tree satisfies condition $(*)$ above, what can be said about the height of the tree as a function of n and α? Briefly justify your answer.

Problem 5. We say that an extended binary search tree is *geometrically-balanced* if the splitter value stored in each internal node `p` is midway between the smallest and largest keys of its external nodes. More formally, if the smallest external node in the subtree rooted at `p` has the value $x_{\min}$ and the largest external node has the value $x_{\max}$, then `p`'s splitter is $(x_{\min}+x_{\max})/2$ (see Fig. 2).

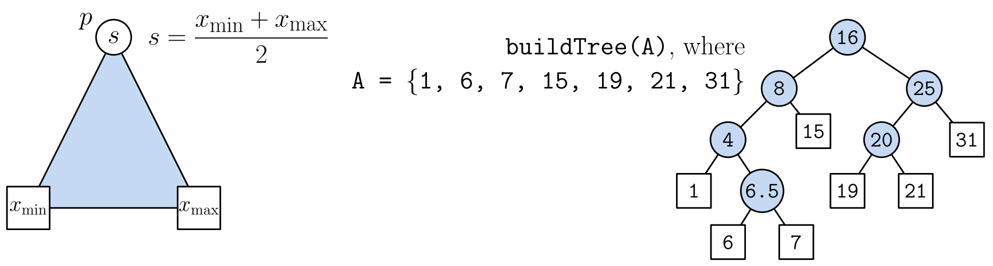


Figure 2: Geometrically balanced tree.

Given a sorted array $A[0 \ldots n-1]$ containing $n \geq 1$ numeric keys, present pseudo-code for a function that builds a geometrically-balanced extended binary search tree, whose external nodes are the elements of A. **Convention:** If a key is *equal* to an internal node's splitter value, then the key is stored in the *left subtree.*

Briefly explain any assumptions you make about underlying primitive operations (e.g., constructors for your internals and external nodes). Any running time is okay.

Problem 6. Given a set P of n points in the real plane, a *partial-range max query* is given two x-coordinates x_1 and x_2, and the problem is to find the point $p \in P$ that lies in the vertical strip bounded by x_1 and x_2 (that is, $x_1 \leq p.x \leq x_2$) and has the maximum y-coordinate (see Fig. 3).

Present pseudo-code for an efficient algorithm to solve partial-range max queries, assuming that the points are stored in a kd-tree. You may make use of any primitive operations on points and rectangles (but please explain them). You may assume that there are no duplicate coordinate values, and no coordinates are equal to x_1 and x_2. If you solve the problem recursively, indicate what the initial call is from the root level. If the tree is balanced, your algorithm should run in time $O(\sqrt{n})$.

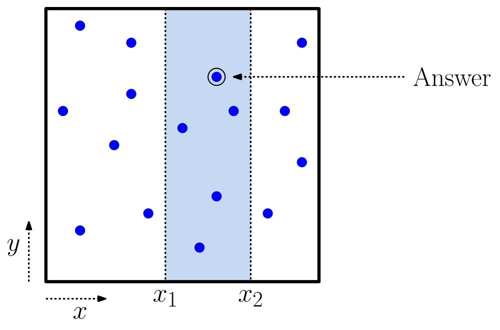


Figure 3: Partial-range max query.

Problem 7. 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!)

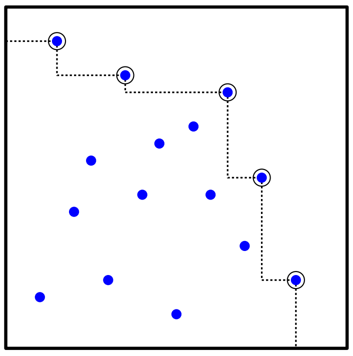
Problem 8. In class we showed that for a balanced kd-tree with n points in the real plane (that is, in 2-dimensional space), any *axis-parallel line* intersects at most $O(\sqrt{n})$ cells of the tree.

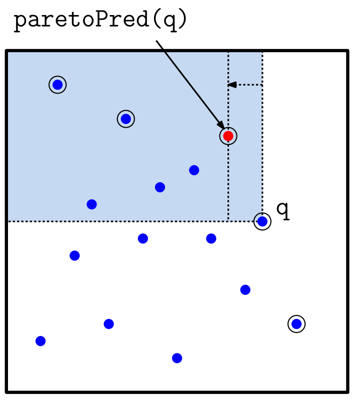
The purpose of this problem is to show that does not apply to lines that are not axis-parallel. Show that for every n, there exists a set of points P in the real plane, a kd-tree of height $O(\log n)$ storing the points of P, and a line ℓ, such that *every* cell of the kd-tree intersects this line.

Problem 9. 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 dominated by any other point of P are called the *Pareto maxima* (the highlighted points of Fig. 4(a)). As seen in the figure, these points naturally define a "staircase" shape.

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. 4(b)).

[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.


(a)



(b)

Figure 4: (a) the Pareto maxima and (b) the Pareto predecessor.

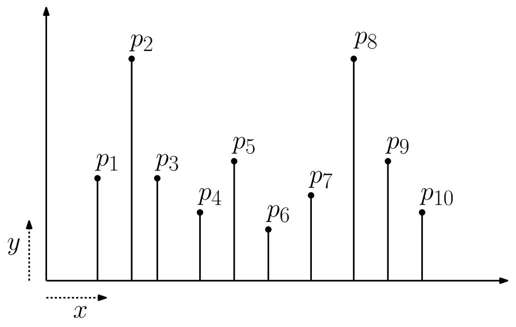
Assuming that the points of P are stored in a kd-tree T, present pseudo-code for a function `T.paretoPred(Point q)`, which returns the Pareto predecessor of a query point q.

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.

Problem 10. In this problem we will see how to use kd-trees to answer a common geometric query, called *ray shooting*. You are given a collection of vertical line segments in 2D space, each starts at the x-axis and goes up to a point in the positive quadrant. Let $P = \{p_1, \dots, p_n\}$ denote the upper endpoints of these segments (see Fig. 5). You may assume that both the x- and y-coordinates of all the points of P are strictly positive real numbers.



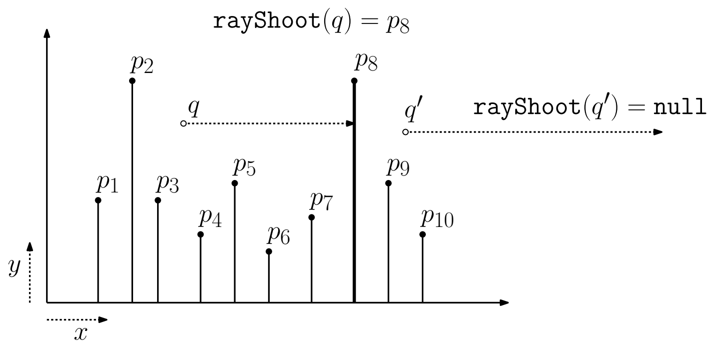


Figure 5: Ray shooting in a kd-tree.

Given a point q, we shoot a horizontal ray emanating from q to the right. This ray travels until it hits one of these segments (or perhaps misses them all). For example, in the figure above, the ray shot from q hits the segment with upper endpoint p_8. The ray shot from q' hits nothing.

In this problem we will show how to answer such queries using a standard point kd-tree for the point set P. A query is given the point $q = (q_x, q_y)$, and it returns the upper endpoint $p_i \in P$ of the segment the ray first hits, or `null` if the ray misses all the segments.

Suppose you are given a kd-tree of height $O(\log n)$ storing the points of P. (It does *not* store the segments, just the points.) Present pseudo-code for an efficient algorithm, `rayShoot(q)`, which returns an answer to the horizontal ray-shooting query (see the figure above, right).

You may assume the kd-tree structure given in class, where each node stores a point `p.point`, a cutting dimension `p.cutDim`, and left and right child pointers `p.left` and `p.right`, respectively. You may make use of any primitive operations on points and rectangles (but please explain them). You may assume that there are no duplicate coordinate values among the points of P or the query point.

Hint: `rayShoot(q)` will invoke a recursive helper function. Here is a suggested form, which you are *not* required to use:

```
Point rayShoot(Point2D q, KDNode p, Rectangle cell, Point best),
```

Be sure to indicate how `rayShoot(q)` makes its initial call to the helper function.

Problem 11. Recall the buddy system of allocating blocks of memory (see Fig. 6). Throughout this problem you may use the following standard bit-wise operators:

`&`	bit-wise "and"	`|`	bit-wise "or"
`^`	bit-wise "exclusive-or"	`~`	bit-wise "complement"
`<<`	left shift (filling with zeros)	`>>`	right shift (filling with zeros)

You may also assume that you have access to a function `bitMask(k)`, which returns a binary number whose k lowest-order bits are all 1's. For example `bitMask(3)` $= 111_2 = 7$.

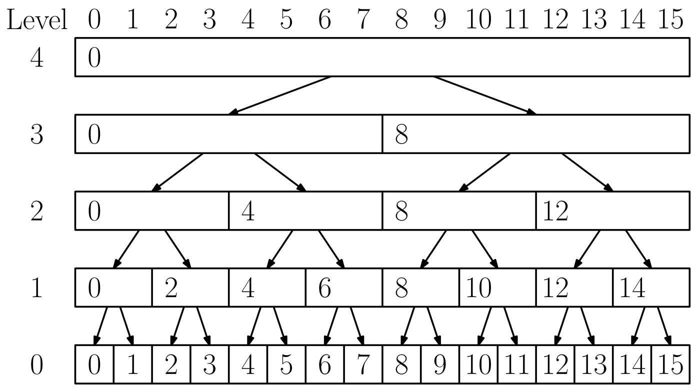


Figure 6: Buddy relatives.

Present a short (one-line) expression for each of the following functions in terms of the above bit-wise functions:

(a) `boolean isValid(int k, int x)`: True if and only if $x \geq 0$ a valid starting address for a buddy block at level $k \geq 0$.

(b) `int sibling(int k, int x)`: Given a valid buddy block of level $k \geq 0$ starting at address x, returns the starting address of its *sibling*.

(c) `int parent(int k, int x)`: Given a valid buddy block of level $k \geq 0$ starting at address x, returns the starting address of its *parent* at level $k+1$.

(d) `int left(int k, int x)`: Given a valid buddy block of level $k \geq 1$ starting at address x, returns the starting address of its *left child* at level $k-1$.

(e) `int right(int k, int x)`: Given a valid buddy block of level $k \geq 1$ starting at address x, returns the starting address of its *right child* at level $k-1$.

Problem 12. Suppose you have a large span of memory, which starts at some address `start` and ends at address `end-1` (see Fig. 7). (The variables `start` and `end` are generic pointers of type `void*`.) As the dynamic memory allocation method of Lecture 15, this span is subdivided into blocks. The block starting at address `p` is associated with the following information:

- `p.inUse` is 1 if this block is in-use (allocated) and 0 otherwise (available)
- `p.prevInUse` is 1 if the block immediately preceeding this block in memory is in-use. (It should be 1 for the first block.)
- `p.size` is the number of words in this block (including all header fields)
- `p.size2` each available block has a copy of the size stored in its last word, which is located at address `p + p.size - 1`.

(For this problem, we will ignore the available-list pointers `p.prev` and `p.next`.)

In class, we said that in real memory-allocation systems, blocks cannot be moved, because they may contain pointers. Suppose, however, that the blocks are movable. Present pseudo-code for a function that compacts memory by copying all the allocated blocks to a single contiguous span of blocks at the start of the memory span (see Fig. 7). Your function `compress(void* start, void* end)` should return a pointer to the head of the available block at the end. Following these blocks is a single available block that covers the rest of the memory's span.

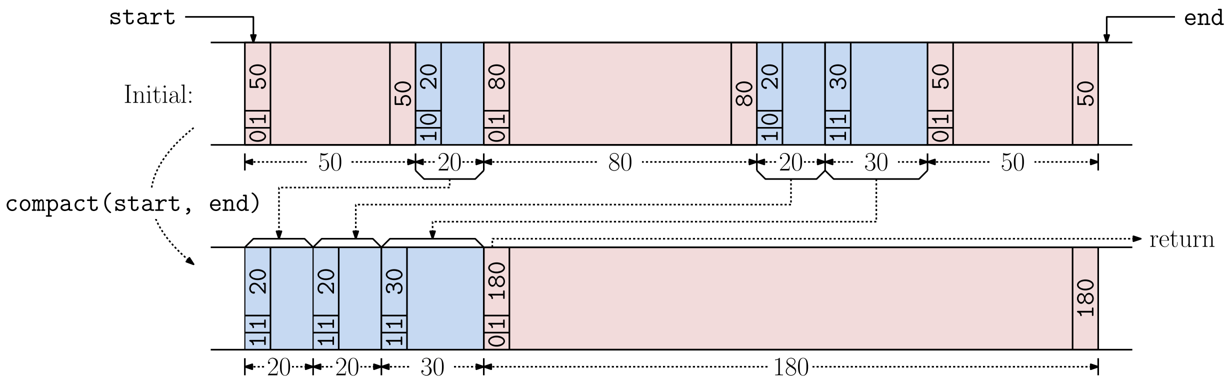


Figure 7: Memory compactor.

To help copy blocks of memory around, you may assume that you have access to a function `void* memcpy(void* dest, void* source, int num)`, which copies `num` words of memory from the address `source` to the address `dest`.