CMSC 420:Fall 2019 Dave Mount

Homework 3: Hashing, kd-Trees and More

Handed out Thu, Nov 14. Deadline: **11am, Tue, Nov 19**. This is a **hard deadline**, since solutions will be discussed in class. You may submit your homework through Gradescope (preferred) or bring hard copy to Tuesday's class. In either case, please write all answers in the *solution template* (provided on the class handouts page).

Problem 1. Consider the B-trees of order 4 shown in Fig. 1 below. Recall that each non-leaf node has between 2 and 4 children, and every node has between 1 and 3 keys. Let us assume two conventions. First, key rotation (when possible) has precedence over splitting/merging. Second, when splitting a node, if the number of keys shared by the two new nodes is an odd number, the leftmost node receives the larger number of keys.

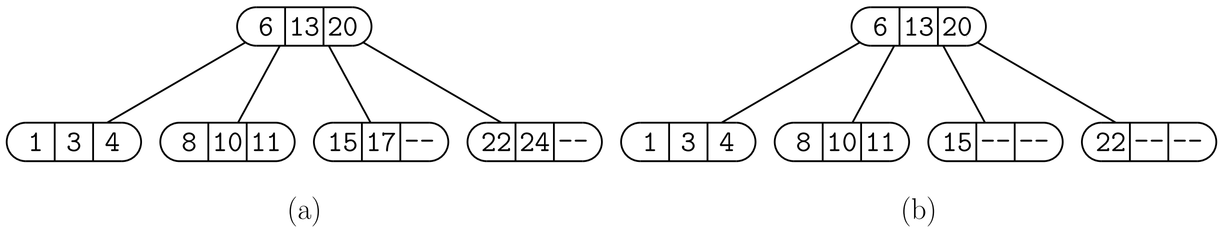

Figure 1: B-tree operations.

(1.1) Show the B-tree that results after inserting the key 9 into the tree of Fig. 1(a).

(1.2) Show the B-tree that results after inserting the key 2 into the (original) tree of Fig. 1(a).

(1.3) Show the B-tree that results after deleting the key 22 from the tree of Fig. 1(b).

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

Problem 2. Show the results of inserting the key "X" into the hash table shown in Fig. 2, assuming open addressing and using each of the following collision resolution methods. Assume that `h("X") = 2`. (If any of the methods goes into an infinite loop, indicate this.)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`table`:	▨	▨	E	H	A	M	J	L	▨	▨	P	C	▨	K	I	▨	D

Figure 2: Hashing with open addressing.

(2.1) Use *linear probing.* As in the lecture notes (Lecture 11, Slides 17 and 21), indicate which cells were probed until finding the insertion location.

(2.2) Use *quadratic probing.* (Starting with the original table.) Again, indicate which cells were probed.

(2.3) Use *double hashing.* (Starting with the original table.) The second hash function is `g("X") = 5`. Again, indicate which cells were probed.

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

Problem 3. Consider the kd-tree shown in Fig. 3. Assume that the cutting dimensions alternate between x and y.

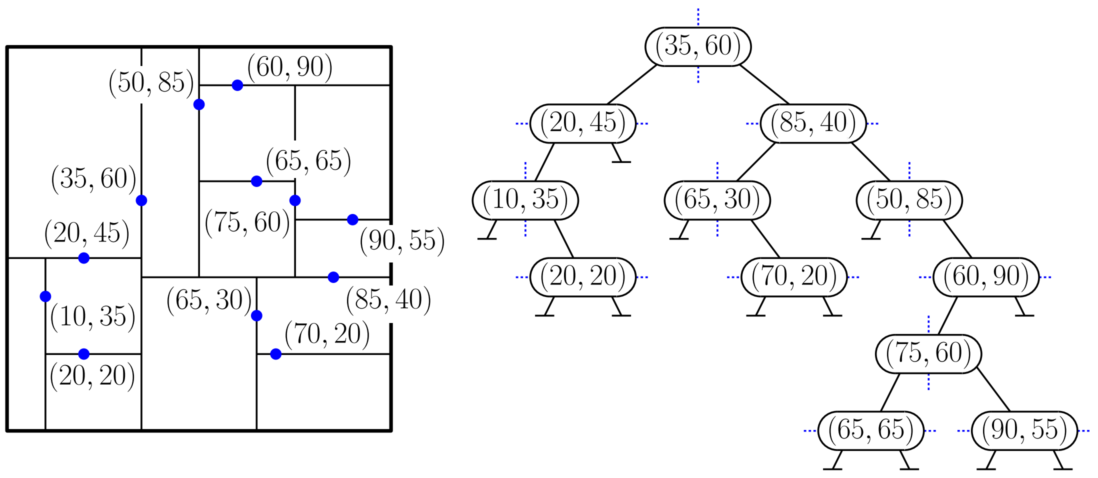

Figure 3: kd-tree operations.

(3.1) Show the result of inserting $(80, 10)$ into this tree. As in Fig. 3 show both the tree structure (with cutting directions indicated) and the subdivision of space.

(3.2) Show the kd-tree that results after deleting the point $(35, 60)$ from the *original* tree. Again, show both the tree structure and the subdivision of space.

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

Problem 4. In our study of Scapegoat trees, we explored the question of how to build a binary search tree containing a given set of keys that is as balanced as possible. In this problem, we will investigate how to do this in the context of 2-3 trees. (Recall Lecture 6.)

Given an array $A[0 \ldots n-1]$ containing $n \geq 1$ key values sorted in ascending order, present pseudocode for an algorithm that builds a valid 2-3 tree of *minimum height* containing these keys. Recall that each node of a 2-3 tree holds either one or two keys. If it holds one key, it has two children (both of which might be `null` if this is a leaf). If it holds two keys, it has three children (all of which might be `null` if this is a leaf). All the leaves of a 2-3 tree are at the same level.

When writing your pseudo-code, assume that a node in the 2-3 tree has the class structure given below. If the node is a leaf, all three child pointers are `null`. If it is a 2-node, then `key2` and `right` are both `null`. All that you may need are the constructor calls for creating 2- and 3-nodes, and these are shown in Fig. 4.

```
class Node23 {                          // a node in the 2-3 tree
  Node23   left;                        // left child
  Key      key1;                        // first key
  Node23   middle;                      // middle child
  Key      key2;                    // second key - set to null for a 2-node
  Node23   right;                   // right child - set to null for a 2-node
                                    // 3-node constructor
  Node23(Node23 l, Key x1, Node23 m, Key x2, Node23 r)
    { left = l; key1 = x1; middle = m; key2 = x2; right = r;  }

  Node23(Node23 l, Key x1, Node23 m) // 2-node constructor
    { left = l; key1 = x1; middle = m; key2 = null; right = null;  }
}
```

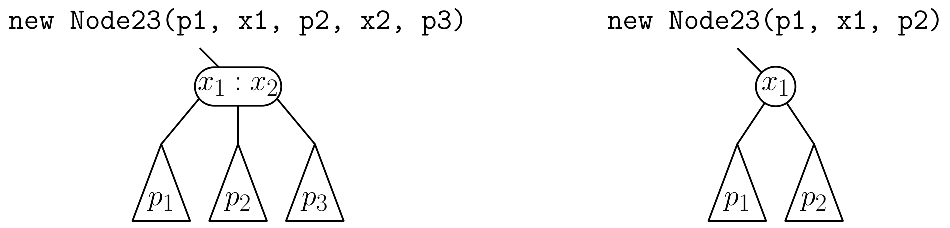

Figure 4: Problem 4: Balanced 2-3 trees.

Following the approach described on Slide 12 of Lecture 12, present pseudo-code for a recursive function `Node23 buildSubtree(Key A[], int i, int k, int h)`, which builds a minimum height 2-3 tree of height h for the k-element subarray $A[i \ldots i+k-1]$ and returns a reference to the root of this subtree. (Its arguments satisfy $0 \le i \le n-1$, $0 \le k \le n-i$, and $h \ge \lfloor \log_3 k \rfloor$.) A formal proof of correctness is not required, but please provide a short English explanation of how your program works.

When using integer division, please be careful to specify whether you are taking the floor or ceiling (e.g., `floor(k/2)` or `ceil(k/2)`.)

Hint 1: For any $n \ge 1$, the minimum height of a 2-3 storing n keys is $\lfloor \log_3 n \rfloor$.

Hint 2: This problem is a bit trickier than I thought when I first handed out the assignment. The difficulty is getting all the leaves to have the same heights. To better understand the issue, observe that a 2-3 tree with 9 keys must have a height of at least 2, and a 2-3 tree with 8 keys may have height of either 1 or 2. If you were asked to build a 2-3 tree with 28 keys, the root would be a 3-node (which uses 2 keys), and the remaining 26 keys would be distributed into three subtrees of sizes 9, 9, and 8, respectively. If you built the minimum height tree for each of these, the leaves in third subtree would be one level higher than the leaves in the first two. The purpose of the parameter h `buildSubtree` is to force a subtree to have a prescribed height, which might be greater than its minimum possible height.

We will give 75% credit for this problem if your algorithm produces a minimum-height tree, even though the leaves are not guaranteed to be at the same level. In this case, you can eliminate the h argument from `buildTree`.