University of Waterloo
CS240E, Spring 2025
Assignment 5

Due Date: Tuesday, July 29, 2025 at 5pm

Question 1 [20 marks]

Update: 2025-07-23: A leaf should have height 0, not 1.

Deletion is hard in any tree, but especially hard in B trees and its variants, as the tree may need to be rebalanced after deletion. In this question we will discuss the variant of B trees in which keys are only stored at the leaves, *the B+ tree*.

In practice, database and filesystem implementations relax the B+ tree invariant to allow for underfilled nodes after deletion, such as the popular Berkeley Database. We will call this a *relaxed B+ tree*. A relaxed *B+ tree* of order $2b$ will always have at least b items and at most $2b$ items in each leaf and node except after deletion, which is allowed to underfill nodes.

Insertion and searching proceed as before in a standard B+ tree.

Deletion in such a tree deletes keys and values at leaves, leaving leaves underfilled until they are empty, at which point that leaf is deleted. When such a leaf is deleted one of the corresponding comparators in its parent is deleted, until it is empty, in which case its comparator in its parent is deleted, and so on, and so forth. An empty node with 1 child is simply skipped over directly to its child while inserting, searching, and deleting.

Question 1.1 Part 1

Argue briefly that *tombstoning*, i.e lazy deletion, can be inappropriate in a B+ tree used for implementing databases or filesystems.

Question 1.2 Part 2

Argue that a B+ tree of order $2b$ that has had m insertions and d deletions has height at most:

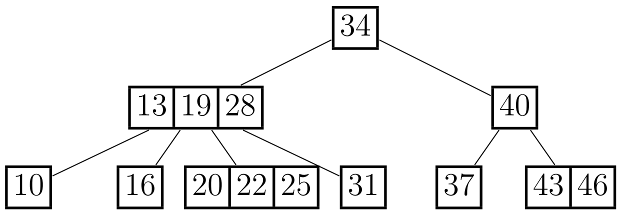
$$\log_b(\frac{m}{b}) + 1.$$

You will want to use the potential function given by summing up $\max\{0, j-b\}b^{h-1}$ over each node, where h is the height of that node and j is the number of items in that node (a leaf has height 0).

(This is a recent result of Sen and Tarjan. You are welcome to read the original research paper, but like many research papers many of the details are left to the reader. I expect an analysis that is well explained and has the details filled in).

Question 2 [20 marks]

Consider the following 2-4-tree of height 2:



1. Show the 2-4 tree that results after doing *insert(26)*. No need to show empty subtrees, and no justification required.
2. Show the red-black tree that corresponds to the given 2-4-tree (**not** your answer to (a)). No need to show empty subtrees, no justification required.

   Clearly indicate the color of nodes, by labelling them 'red' or 'black' or by using different shapes as in Question 1(i).
3. Let T be a red-black tree of even height $h \geq 0$. Show that levels $0, 1, \ldots, \frac{h}{2}$ are completely full.

Question 3 [20 marks]

One trick to make searching in a dictionary faster is to add an auxiliary data structure that can tell you with high probablity when an item is *not* in the dictionary. This works especially well if the keys are numbers and we can do hashing.

Thus, you would like to create a data structure that supports the operation IsIn(k), which returns a boolean value as follows:

- If it returns false, then k is definitely not in the dictionary.
- If it returns true, then k is in the dictionary with probability $> \frac{1}{2}$.

You will also need the following two operations:

- UpdateOnInsert(k): An item with key k is inserted into the main dictionary, and your auxiliary data structure should therefore also be updated.

- UpdateOnDelete(k): An item with key k is deleted from the main dictionary, and your auxiliary data structure should therefore also be updated. This function will not be called unless k was indeed a key in the dictionary.

Describe an implementation of such a data structure that supports these three operations in $O(1)$ **worst-case** time, and that uses $O(N)$ space, where N is a upper bound on the maximum number of items that will be in the dictionary. You may assume that you know N beforehand and that you have uniform hash-functions available.

Question 4 [20 marks]

A *MultiMap* is an ADT that allows to store multiple values associated with keys. Thus, `Insert`(k, v) may be called even if key k already existed in data structure, and it adds the value v as one of the values associated with k. (You may assume that v was not previously associated with k.) `Search`(k) returns *all* values that are associated with key k, and `Delete`(k) deletes all of them.

Assume that the keys are uniformly distributed integers. Let N be the number of keys and n be the total number of values that are stored in the dictionary, noting that $n \gg N$ is possible. Describe a realization of ADT MultiMap with expected space $O(n)$, where the operations have the following run-time: *insert(k, v)* has expected time $O(1)$ *search(k)* and *delete*(k) both have worst-case run-time $O(1+s)$, where s is the number of values associated with k. You do not need to give pseudo-code for the methods, but describe the idea in detail.

Question 5 [20 marks]

a) (Warm-up.) Consider the text $ACAGATATACACAACG$ over alphabet $\Sigma = \{A, C, G, T\}$.

What is the cost of the corresponding Huffman-encoding? Show how you obtained your answer, and also write the length of the code-word for each character.

b) Given some probabilities $p_1, \ldots, p_s$ (with $0 < p_i < 1$ and $\sum_{i=1}^{s} p_i = 1$), the *entropy* is defined to be

$$H(p_1, \ldots, p_s) = -\sum_{i=1}^{s} p_i \log_2(p_i).$$

For a text S over alphabet $\Sigma = \{x_1, \ldots, x_s\}$, we define the entropy $H(S)$ to be

$$|S| \cdot H(p_1, \ldots, p_s),$$

where $p_i = \frac{1}{|S|}(\text{frequency of } x_i)$ is used as "probability" of character x_i for $i = 1, \ldots, s$.

Compute $H(S)$ for the text from part (a). Show how you obtained the answer (in particular, list the probabilities).

c) Let S be a text such that the length of S and the frequency of the characters $x_1, \ldots, x_k$ in S are powers of 2. In other words, there exist integers $\ell_0, \ldots, \ell_k$ such that $|S| = 2^{\ell_0}$ and x_i has frequency $f_i = 2^{\ell_i}$ for $i = 1, \ldots, k$. Show that the Huffman-encoding of S has cost $H(S)$.

Hint: The text from (a) satisfies the assumption. Study its Huffman-trie: what can you say about the length of the encoding of x_i? (For ease of description, assume that the naming is such that $\ell_1 \geq \cdots \geq \ell_k$.)

Motivation: Based on Shannon's information-theoretic lower bound, one can argue that *any* encoding of S as bit-string (whether obtained via prefix-free binary encoding or otherwise) has length at least $H(S)$. So in the special case where the frequencies are powers of 2, Huffman-encoding gives the minimum-length encoding that is possible.