Week 6 Studio Sheet

(To be completed during the Week 6 studio class)

Objectives: The tutorials, in general, give practice in problem solving, in analysis of algorithms and data structures, and in mathematics and logic useful in the above.

Instructions to the class: Aim to attempt these questions before the tutorial! It will probably not be possible to cover all questions unless the class has prepared them in advance. There are marks allocated towards active participation during the class. You **must** attempt the problems under **Assessed Preparation** section **before** your tutorial class and give your worked out solutions to your tutor at the start of the class – this is a hurdle and failing to attempt these problems before your tutorial will result in 0 mark for that class even if you actively participate in the class.

Instructions to Tutors:

1. The purpose of the tutorials is not to solve the practical exercises!
2. The purpose is to check answers, and to discuss particular sticking points, not to simply make answers available.

Supplementary problems: The supplementary problems provide additional practice for you to complete after your tutorial class, or as pre-exam revision. Problems that are marked as **(Advanced)** difficulty are beyond the difficulty that you would be expected to complete in the exam, but are nonetheless useful practice problems as they will teach you skills and concepts that you can apply to other problems.

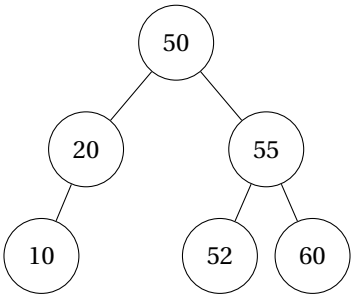
Assessed Preparation

Problem 1. A hash table can be used to store different types of elements such as integers, strings, tuples, or even arrays. Assume we want to use a hash table to store arrays of integers (i.e., each element is an array of integers). Consider the following potential hash functions for hashing the arrays of positive integers. Rank them in terms of quality and give a brief explanation of the problems with each of them. Assume that they are all taken mod m, where m is the table size.

- Return the first number in the array (e.g., if `array=[10,5,7,2]`, hash index will be `10%m`)
- Return a random number in the array (e.g., if `array=[10,5,7,2]`, hash index will be a randomly chosen element from the array mod m)
- Return the sum of the numbers in the array (e.g., if `array=[10,5,7,2]`, hash index will be `24%m`)

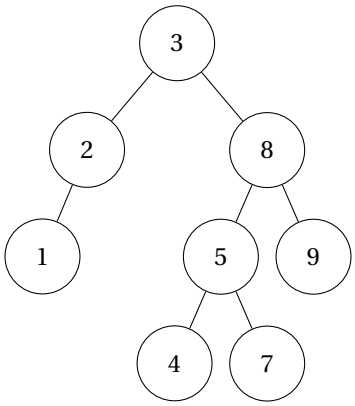
Give a better hash function than these three and explain why it is an improvement.

Problem 2. Insert 5 into the following AVL tree and show the rebalancing procedure step by step.



Studio Problems

Problem 3. Insert 6 into the following AVL tree and show the rebalancing procedure step by step.



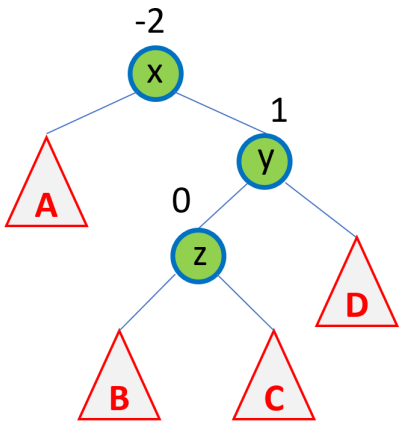
Problem 4. Placeholder to retain correct question numbering

Problem 5. Recall the algorithm for deleting an element from a binary search tree. If that element has no children, we do nothing. If it has one child, we can simply remove it and move its child upwards. Otherwise, if the node has two children, we swap it with its *successor* and then delete it. This algorithm works because of the following fact: The successor of a node with two children has no left child. Prove this fact.

(a) First prove that the successor of a node x with two children must be contained in the right subtree of x

(b) Use this fact to prove that the successor of x can not have a left child

Problem 6. Consider the binary search tree shown below, with balance factors as indicated. Assume that the subtrees A, B, C and D are AVL trees. Show that performing the rotation algorithm for the right-left case results in an AVL tree.



Problem 7. Consider the following potential hash functions for hashing integers. Rank them in terms of quality and give a brief explanation. Assume that they are all taken mod m, where m is the table size.

- Return $x \mod 2^p$ for some prime p
- Return $(ax+b)$ for some positive, and randomly chosen, a, b
- Return a random integer

Problem 8. Consider the following probing schemes and for each of them, explain whether they do or do not suffer from primary or secondary clustering. In each problem, h returns an index where the item k is to be inserted when probing for the i^{th} time. h', h_1, h_2 are hash functions.

(a) $h(k,i) = (h'(k) + 5i) \mod m$

(b) $h(k,i) = (h'(k) + i^{\frac{3}{2}}) \mod m$

(c) $h(k,i) = (h_1(k) \times (h_2(k)^i) \mod m$

(d) $h(k,i) = (h'(k) + 2^{i+1}) \mod m$

(e) $h(k,i) = (h_1(k) \times h_2(k) + i) \mod m$

Problem 9. You are given a set of distinct keys $x_1, x_2, ..., x_n$.

(a) Design an algorithm that creates a binary search tree of minimal height containing those keys in $O(n \log(n))$ time.

(b) Prove that your algorithm produces a BST of height at most $\log(n)$.

(c) Prove that $O(n \log(n))$ is the fastest algorithm for this problem in the comparison model

Problem 10. We know that when inserting an element into a binary search tree, there is only one valid place to put that item in the tree. Let's prove this fact rigorously. Let T be a binary search tree and let x be an integer not contained in T. Prove that exactly one of the following statements is true:

- The successor of x has no left child
- The predecessor of x has no right child

That is, prove that it can not be the case that both of these statements are true simultaneously, nor that both of these statements are false simultaneously. This implies that there is a unique insertion point for x since upon insertion x must either become the left child of its successor or the right child of its predecessor.

Supplementary Problems

Problem 11. Implement hashtables that use chaining, linear probing, and quadratic probing for collision resolution. You may reuse hashtable code from previous units if you have it. Compare these hashtables on randomly generated integers and compare their performance. Try adjusting the hash function, the table size, and the number of keys inserted and see how this affects the results.

Problem 12. In lectures we claimed that AVL trees are good because the balance property guarantees that the tree always has height $O(\log(n))$. Let's prove this.

(a) Write a recurrence relation for $n(h)$, the **minimum** number of nodes in an AVL tree of height h. [Hint: It should be related to the Fibonacci numbers]

(b) Find an exact solution to this recurrence relationship in terms of the Fibonacci numbers[1]. [Hint: Compare the sequence with the Fibonacci sequence, find a pattern, and then prove that your pattern is right using induction]

(c) Prove using induction that the Fibonacci numbers satisfy $F(n) \geq 1.5^{n-1}$ for all $n \geq 0$

(d) Using (a), (b), and (c), prove that a valid AVL tree with n elements has height at most $O(\log(n))$

Problem 13. (Advanced) Consider a hashtable implementing linear probing, with size $m = 17$ using the hash function

$$h(k) = (7k + 11) \mod m.$$

[1] For this problem, index the Fibonacci numbers from zero with the base case $F(0) = F(1) = 1$.

Give a sequence of keys to insert into the table that will cause its worst-case behaviour.

Problem 14. (Advanced) Suppose that we insert n keys into a hashtable with m slots using a totally random hash function. What is the expected number of pairs of colliding elements? The pairs (k, k') and (k', k) are considered the same and should not be counted twice.

Problem 15. (Advanced) Recall the algorithm for insertion into a hashtable using Cuckoo hashing. Assume that $m \geq 2n$ and MaxLoop $= O(\log(n))$. Given that the probability that an insertion triggers a rebuild is $O(n^{-2})$, and insertions succeed in expected constant time when a rebuild is not triggered:

(a) Prove that the probability that a sequence of n insertions succeeds is $1 - O\left(\frac{1}{n}\right)$

(b) Prove that the expected time complexity of insertion is $O(1)$

Problem 16. (Advanced) In this problem we consider a solution to the *static hashing* problem. We are given a set of n keys and want to build a hashtable on those keys that supports $O(1)$ worst-case lookup. No further keys will be added or removed.

A strategy similar to hashing with chaining is FKS hashing[2]. In FKS hashing, we implement a two-level hashtable. The top-level hashtable of size $m = n$ stores colliding elements in inner hashtables. To support $O(1)$ lookup, we want the inner hashtables to have no collisions.

We start out by selecting a random hash function $h(k)$ for the top-level hashtable from a *universal family*. This means that we have for all $x \neq y$,

$$\Pr_{h \in \mathscr{H}}\{h(x) = h(y)\} \leq \frac{1}{m}.$$

We compute the hash values for all n keys and count the number of collisions in each slot. The inner hashtable for slot v is initialised with size n_v^2 where n_v is the number of keys that hashed to v. If the total size of all inner hashtables is too large, we pick a new hash function and try again.

Once we are happy with the total size of the hashtable, we select hash functions for the inner hashtables (also randomly from a universal family). If any of them have collisions, we select a new hash function for those until there are no longer any collisions.

It remains to prove that finding suitable hash functions can be done fast in expectation. Let's show that building the inner tables can be done fast.

(a) Prove that the probability that a collision occurs in a given inner hash table of size $m_v = n_v^2$ containing n_v elements is less than $1/2$

(b) Hence argue that we can find a hash function that yields no collisions on this inner table in a constant number of tries in expectation

Next, let's show that we can find a hash function for the outer table fast.

(c) Argue that

$$\sum_{v=0}^{m-1} n_v^2 = \sum_{i=1}^{n}\sum_{j=1}^{n} I_{i,j},$$

where $I_{i,j} = 1$ if $h(k_i) = h(k_j)$ or 0 otherwise.

(d) Using (c), prove that

$$\mathrm{E}\left[\sum_{v=0}^{m-1} n_v^2\right] \leq 2n,$$

[2]Fredman, Komlós, Szemerédi, Storing a Sparse Table with $O(1)$ Worst Case Access Time, *Journal of the ACM* 1984

(e) Using (d) and the fact that $\Pr[X \geq a] \leq \frac{\mathrm{E}[X]}{a}$ (this is called *Markov's inequality*), deduce that

$$\Pr\left[\sum_{v=0}^{m-1} n_v^2 \geq 4n\right] \leq \frac{1}{2}.$$

(f) Use (f) to deduce that we can find a hash function that yields a total table size of $O(n)$ in a constant number of tries in expectation

Now we have all of the ingredients to conclude the analysis.

(g) Finally, deduce that we can build the entire hashtable in $O(n)$ expected time