CS2040S Tutorial 7

Group T41

Week 9

Picture of the Day

Problem 1: Hashing Basics

Problem 1a

Try hashing these items $[42, 24, 18, 36, 52, 0, 47, 45, 60, 27, 32, 7]$ with the following hash function $h(x) = x \bmod 7$. Each row in the table corresponds to the bucket of $h(x)$. Fill in the table below with your answer!

Problem 1a

$h(x)$	x_1	x_2	x_3	x_4
0				
1				
2				
3				
4				
5				
6				

Solution

See answer sheet

Problem 1b

We typically use Linked Lists to store the items in a bucket! But... what if instead of a Linked List, we use an AVL Tree to store the items in a bucket?

What are the advantages or disadvantages of such a solution?

Solution

- AVL tree is faster in terms of asymptotic runtime when it comes to insertion and deletion (assuming insertion takes $\mathcal{O}(1)$ time with linked list)
- AVL tree has higher overhead, as it keeps track of more data (i.e. left child and right child, balance factor). Not to mention comparisons done when rotation occurs. On the other hand, linked list only needs to keep track of the next node.
- AVL tree is more complex
- Conclusion: AVL tree improves the ***worst case*** at cost of increase in overhead and complexity
- Bonus: Java's implementation of `HashMap` use linked list, and red-black tree as fallback after certain threshold.

Problem 1c

The goal of Hash Tables are to store (key, value) pairs. Here’s a question, at each bucket, is storing just the ***(value)*** sufficient? Or do we need to store the entire ***(key, value)*** pair? Why do you think so?

For example, for a (key, value) pair of (17, 200). At the bucket h(17), is storing (200) sufficient, or do you need to store (17, 200)?

Solution

Not sufficient. Remember that collisions are possible when we use hash functions. We need to be able to differentiate two different keys k_1 and k_2 when $h(k_1) = h(k_2)$, i.e. when they are hashed into the same bucket. Otherwise, when we retrieve k_1, we wouldn't know which value to return.

Problem 2: The Missing Element

Given an array of n integers without any particular order, but possibly with duplicates. Find the first missing number (as if we were counting from 1), or output "all present" if all values 1 to n were present in the input.

Example: given the array $[8, 5, 3, 3, 2, 1, 5, 4, 2, 3, 3, 2, 1, 9]$, the first missing number is 6.

State the space and time complexity of your solution.

Solution

- Use array or hash set
- For each number x in the array, we indicate that x is present in a hash table H, e.g. $H[x] = true$
- We iterate i from 1 to n and find whether $H[i] = true$. If it's false, then it's the first missing number.
- Time and space complexity: $\mathcal{O}(n)$

$\mathcal{O}(1)$ Additional Space?

$[8, 5, 3, 3, 2, 1, 5, 4, 2, 3, 3, 2, 1, 9]$

- For all elements, we try to put it to its correct position
- We skip once it's in the correct position or the value is larger than n
- In this example, try put the first element 8 to index 8. We see we have to "kick out" 4 and place it to the correct position.
- Repeat until the number is in the correct position.
- After that, iterate throught the array once again. The first element that's not equal to its index will be the answer.

Will it stop "kicking"?

- Draw a graph with N nodes, where node i corresponds to the index i.
- Draw an edge from node i to node $A[i] \leq n$, where A is the given array.
- You will stop if you revisit the same index again
  - if the last traversed node is self-loop
  - if the path forms cycle
- After one round of series of kicking, the traversed node will be self-loop.
- In total, we expect to traverse $\mathcal{O}(n)$ time.

Problem 3: Coupon Chaos

Given n coupons of t distinct types. Sort the coupon in ascending order.

State your running time and (additional) space complexity.

Solution 1

- Quicksort with 3-way partitioning. Runs in $O(n \log t)$ with $O(1)$* additional space

* In this module, we do not care about *hidden space* incurred like stack. If you want to take this into account, then it takes $O(\log t)$ space.

Solution 2

- AVL Tree
- Duplicate will be keep tracked as `count` property of the node
- Traverse the node in inorder traversal to generate the sorted array
- Runs in $O(n \log t)$ with $O(t)$ additional space

Solution 3

- Counting sort
- Use hash table instead of array
- Ordering the elements must be from the t distinct coupons. Sort it first
- Runs in $O(n + t \log t)$ time with $O(t)$ additional space.

Problem 4: Data Structure 2.0

Implement a data structure called `RandomizedSet` that supports the following operations:

- `RandomizedSet()`: initializes the data structure
- `insert(val)`: inserts an item `val` into the set if *not* present
- `remove(val)`: removes the item `val` from the set if *present*
- `getRandom()`: returns a random element from the current set of elements. Every element must have have an ***equal probability*** of being returned

All these operations must work in *expected* $\mathcal{O}(1)$ time!

Assume that the maximum number of elements present in `RandomizedSet` will never exceed a reasonable number n.

Solution

- You need fast insert and fast remove
- Hashset is a good candidate
- Stores the values
- How to get random item?

Solution

- Use an array `A` to store the elements in the set
- Store how many elements in the array (call it `size`)
- Should use hashmap instead of hashset
- Key: `val` inserted; Value: index in the array
- We can get random value easily: take randomly the index from 0 to `size` - 1
- Insertion: add element into `A[size]`, then increase `size` by 1
- What about deletion?

Solution

- Swap with last index
- Decrease `size` by one
- Oof! :o

Problem 5: Data Structure 3.0

Implement ADT with the following properties:

- Insert in $O(\log n)$ time
- Delete in $O(\log n)$ time
- Lookup in $O(1)$ time
- Find successor and predecessor in $O(1)$ time

Solution

- AVL + Hash Table
- A general hash table that hashes given value to the corresponding *node* in AVL
- *Each node* in AVL will have a *hash table* to indicate its successor and predecessor
- When we insert, update the successor and predecessor of:
    - the inserted node
    - the successor of inserted node
    - the predecessor of inserted node
- When we delete, we do the same thing