10.2 hash tables

Thursday, November 10, 2022

TODAY

review hash table data structure
Comparison of Dictionary ADT implementations
Hash table implementation

HASH TABLE REVIEW

Key idea: tradeoff where we use more space in order to have efficient operations

operations insert, remove, get, and update are $O(1)$ * terms and conditions apply

use a very big array

array indexing is $O(1)$

map key ⟶ array index using a hash function

hash(key, capacity) returns index

- convert key to integer
- use $\sum$, ASCII values of each character (x a prime number)
- use mod (%) to get an index in the right range

collisions occur if multiple keys hash to the same index. Two methods to handle this:

- probing: find next available array location, put it there
- chaining: store all collisions in the same index ⟵ We'll use this approach

load factor is size/capacity

- when this gets too high, the array is getting full, so we should increase capacity

Dictionary implementations we've seen:

- BST operations cost $O(h)$, and h can be $\leq n$
- AVL trees $O(\log_2 n)$
- Hash table $O(1)$ * terms and conditions apply

Note: C++ manages vectors as dynamically-allocated memory. We don't have to manage them; they are built-in.

new implementation: **LinearDictionary** (stores its data in a C++ vector of type vector<pair<k,v>>)

runtime	operation	implementation
$O(n)$	insert(k,v)	linear search of vector, if key is already there then throw an error, otherwise push_back

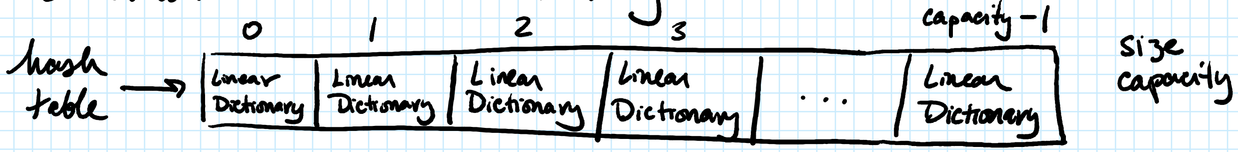
$O(n)$	remove(k)	linear search of vector, if key is there then remove it else throw error
$O(n)$	get(k)	linear search, throw error if not found
$O(n)$	update(k,v)	linear search, throw error if not found
$O(n)$	contains(k)	linear search
$O(n)$	getKeys()	iterate through vector, building a vector of keys
$O(1)$	getItems()	return vector
$O(1)$	getSize()	return vector's size
$O(1)$	isEmpty()	return whether vector's size is 0

Ok, so LinearDictionary is not a better implementation.
BUT it is simpler! If we know beforehand that n will be small, we might prefer a simpler implementation.

HASH TABLE IMPLEMENTATION

- private data:
  size
  capacity
  dynamic array of static LinearDictionaries

Each chain will be a LinearDictionary.



We expect each chain to be short, so $O(n)$-cost operations on each LinearDictionary will be $O(1)$ for the hash table.

Allocating the LinearDictionaries statically means that they will automatically get cleaned up, and we only need to delete the array.

example:

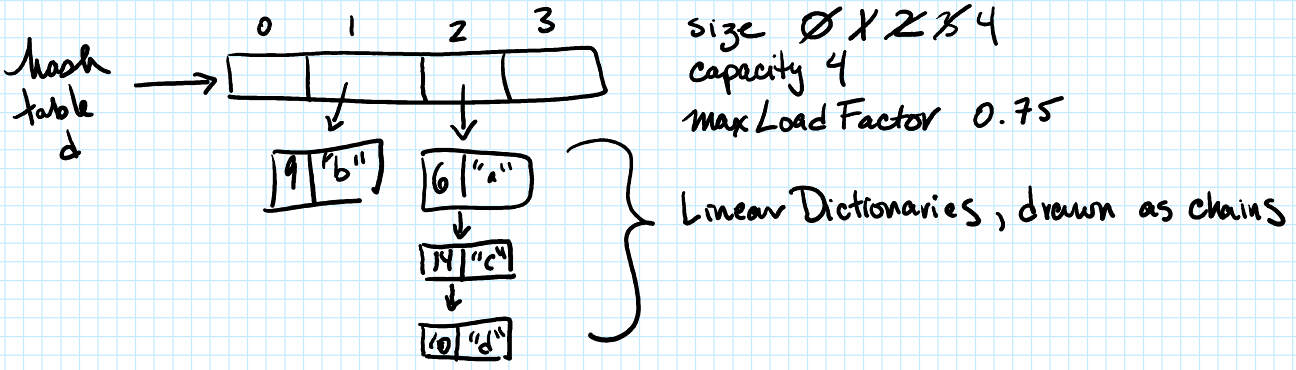
Dictionary<int, string> *d = new HashTable<int, string>();
d → insert(6, "a") $6 \% 4 = 2$ updated size = 1, load $= \frac{\text{size}}{\text{capacity}} = \frac{1}{4} = 0.25 \leq 0.75$
d → insert(9, "b") $9 \% 4 = 1$ size = 2   load $= \frac{2}{4} = 0.5 \leq 0.75$ no resize

capacity 4

d → insert(9, "b") $9 \% 4 = 1$ size=2 load = $\frac{2}{4} = 0.5 \leq 0.75$ no resize is needed

d → insert(14, "c") $14 \% 4 = 2$ size=3 load = $\frac{3}{4} = 0.75 \leq 0.75$ no resize YET

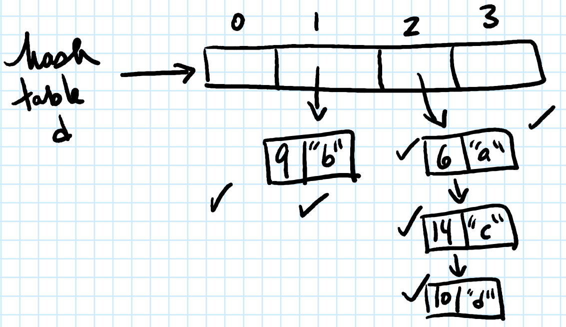
d → insert(10, "d") $10 \% 4 = 2$ size=4 load = $\frac{4}{4} = 1 > 0.75$ time to resize!



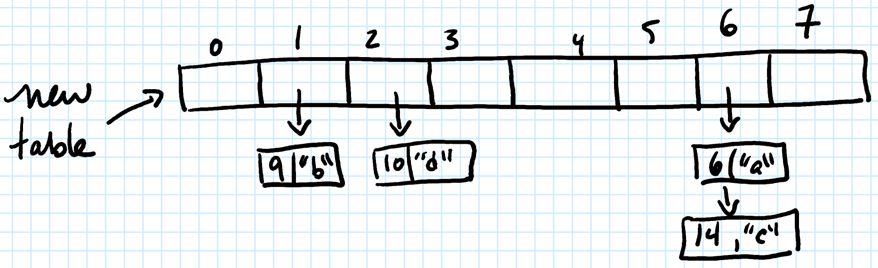
size ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4
capacity 4
maxLoadFactor 0.75

HashTable will have a private helper method called EnsureCapacity.
If the load factor gets > maxLoadFactor, then
EnsureCapacity does:

- doubles capacity by creating new array
- rehashes all keys and adds them to new array
- deletes old array, sets table to new array
- sets capacity to new capacity



size 4
capacity 4
max Load Factor 0.75
load 1



size 4
capacity 8
maxLoadFactor 0.75

After running EnsureCapacity, the load factor is $\frac{\text{size}}{\text{capacity}} = \frac{4}{8} = 0.5 < 0.75$

We improved it!

IMPLEMENTING HASH TABLE METHODS

Each HashTable is an array (called "table") of LinearDictionaries.

```
V get(K key)
    int index = hash(key, capacity) // find the index within table
    return table[index].get(key) // use the LinearDictionary's get method

void insert(K key, V value)
    int index = hash(key, capacity)
    table[index].insert(key, value) // use the LinearDictionary's insert method
    size ++
    float load = float(size) / capacity // cast the numerator as a float
                                        // to ensure float division
    if load > maxLoadFactor
        ensureCapacity()

void remove(K key)
    int index = hash(key, capacity)
    table[index].remove(key)      // use the LinearDictionary's method
    size --

vector<K> getKeys()
    create a result vector
    for i=0 to capacity-1
        table[i].getKeys()      // use the LinearDictionary's method
        append that vector to the result vector
    return result vector
```

Running time of all Dictionary methods when implemented with a HashTable:

Assumptions:
- we have a good hash function
- on average, each LinearDictionary is small

void insert(k, v)   O(1) amortized

v get(k)	O(1)	
v remove(k)	O(1)	
void update(k, v)	O(1)	
bool contains(k)	O(1)	
bool isEmpty()	O(1)	
int getSize()	O(1)	
vector<k> getKeys()	O(capacity)	Two slow methods, even with our helpful assumptions.
vector<pair<k, v>> getItems()	O(capacity)	