Tutorial 6

Hash Tables

Lecturer: Dr Andrew Hines *TA: Esri Ni*

6.1 Hash Tables

Python's `dict` class is arguably the most significant data structure in the language. It represents an abstraction known as a dictionary in which unique keys are mapped to associated values. Because of the relationship they express between keys and values, dictionaries are commonly known as associative arrays or maps.

6.1.1 ADT of a symbol table or map

For an **unordered symbol table** the ADT has the following operations:

`put(key, value)`	put key-value pair into the table
`get(key)`	value paired with key (null if key is absent)
`delete(key)`	remove key from table and value paired with key
`contains(key)`	is there a value paired with key?
`isEmpty()`	is the table empty?
`size()`	number of key-value pairs in the table
`keys()`	all the keys in the table

6.1.2 Definitions

- **Hash Tables** save items in a key-indexed table (index is a function of the key)
- A **Hash Function** is a method for computing array index from a key.
- **Uniform Hashing Assumption** is that each key is equally likely to hash to an integer between 0 and $M-1$
- **Collision resolution** requires an algorithm and data structure to handle two keys that hash to the same array index
- Ideally Scramble the keys uniformly to produce **equally computable** table indices where each table index is **equally likely** for each key.
- To implement a hash table, **Chaining** and **Linear Probing** are two algorithms that produce different data structures.

6.1.3 Expected Complexity

Operation	List	Hash Table (expected)	Hash Table (worst case)
put	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
get	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
delete	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
size	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
keys	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

6.2 Exercises

Question 1. Chaining and Linear Probing are two hash table implementation strategies. Compare them in terms of complexity for `insert` and `search` and `delete` operations.

If the array is length m chaining will be $\mathcal{O}(1)$ for the array index + $\mathcal{O}(n/m) \approx \mathcal{O}(1)$ to step through the linked list with uniform hashing. Linear probing will also generally be $\mathcal{O}(1)$ to achieve lookups. Chaining will handle deletions easier (no need to rehash) but linear probing will handle insertions better (no need to allocate more memory for another link in the chain as the array is defined up front.)

Question 2.
Suppose we declare a Python dictionary:

```
cities = {'Paris':40, 'Dublin':10}.
```

What is the Python command to:

a. obtain the number of entries in dictionary

b. delete the entry for 'Dublin'

c. add key 'Cork' with a value of 60

d. merge together `cities` dictionary with a second dictionary:

```
towns = {'Wicklow':6, 'Skerries':2}
```

```
cities = 'Paris':40, 'Dublin':10
len(cities)
del cities['Dublin']
cities.update('Cork':60)
cities.update(towns)
```

Question 3. Computing the storage locations for a hash table by hand. We want to take a set of keys, k, and compute the hash values $h(k)$ using chaining and linear probing and insert the value in their appropriate locations.
For the keys we will use integers:

$$h(i) = 4i + 3 \mod 11$$

Build an answer table in the form:

i	0	1	2	3	4	5	6	7	8	9	10
k	3	33	28	16	45	5	99	21	16	1	8
h(k)											
linear probe											
chain lvl 0											
chain lvl 1											
chain lvl 2											

Solution

i	0	1	2	3	4	5	6	7	8	9	10
k	3	33	28	16	45	5	99	21	16	1	8
h(k)	4	3	5	1	7	1	3	10	1	7	2
linear probe	8	16	5	33	3	28	99	45	16	1	21
chain lvl 0		16	8	33	3	28		45			21
chain lvl 1		5		99				1			
chain lvl 2		16									