1 Map ADT

- get(k): if the map M has an entry with key k, return its associated value.
- put(k, v): if key k is not in M, then insert (k, v) into the map M; else, replace the existing value associated to k with v.
- remove(k): if the map M has an entry with key k, remove it.
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M.
- values(): return an iterable collection of the values in M.
- keySet(): return an iterable collection of the keys in M.

1.1 Sorted map ADT (extra methods)

- firstEntry() returns the entry with the smallest key; if map is empty, returns null.
- lastEntry() returns the entry with the largest key; if map is empty, returns null.
- ceilingEntry(k) returns the entry with the least key that is greater than or equal to k (or null, if no such entry exists).
- floorEntry(k) returns the entry with the greatest key that is less than or equal to k (or null, if no such entry exists).
- lowerEntry(k) returns the entry with the greatest key that is strictly less than k (or null, if no such entry exists).
- higherEntry(k) returns the entry with least key that is strictly greater than k (or null, if no such entry exists).
- subMap(k1,k2) returns an iteration of all the entries with key greater than or equal to k1 and strictly less than k2.

1.2 List-Based (unsorted) Map

We can implement a map using an unsorted list of key-item pairs To do a get and put we may have to traverse the whole list, so those operations take O(n) time. Only feasible if map is very small or if we put things at the end and do not need to perform many gets (i.e., system log)

1.3 Tree-Based (sorted) Map

We can implement a sorted map using an AVL tree, where each node stores a key-item pair To do a get or a put we search for the key in the tree, so these operations take O(h) time, which can be O(log n) if the tree is balanced. Only feasible if there is a total ordering on the keys

2 Hash Functions and Hash Tables

2.1 Simple Map Implementation with restricted keys

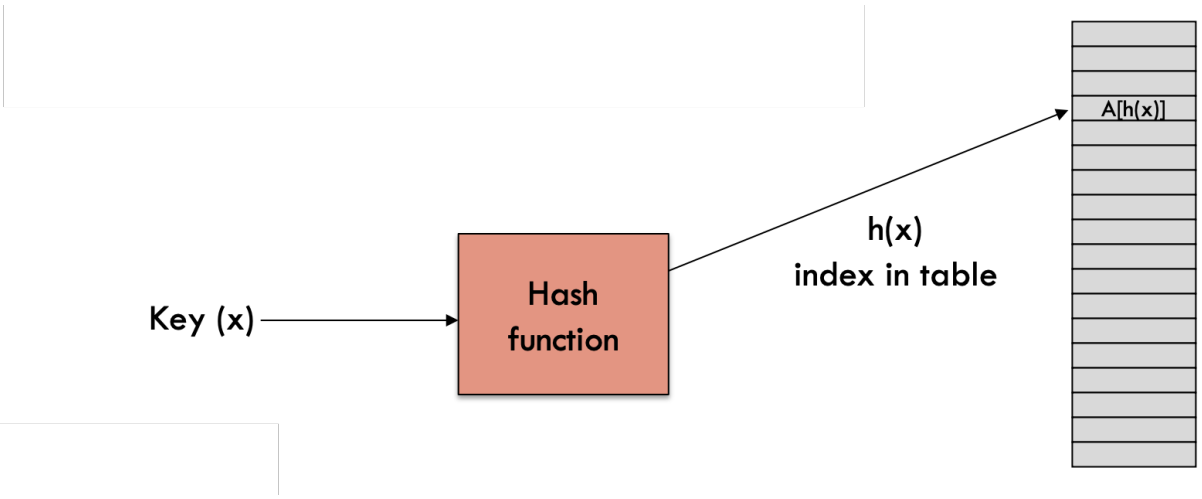
Maps support the abstraction of using keys as addresses to get items. Consider a restricted setting in which a map with n items with keys in a range from 0 to N-1, for some N >= n. This can be implemented through an array of size N. The key can be indexed such that entries can be located directly. O(1) operations (get, put, remove).

The drawback is that usually N » n, e.g. StudentID is 9 digits, so a Map with StudentID key can be stored in array of 1,000,000,000 entries (way more than the number of students). This results in bad space utilization.

2.2 Hash Functions and Hash Tables

To get around these issues, we use a hash function h to map keys to corresponding indices in an array A.

- h must be a mathematical function (always gives same answer for any particular x) and fairly efficient to compute.
- Example: h(x) = x mod N is a hash function for integer keys.
- The integer h(x) is called the hash value of key x.



A hash function h is usually the composition of two functions:

- Hash code: h1: Keys to integers (If the keys are not already integers).
- Compression function: h2: Integers to [0, N-1].

As such, h(x) = h2(h1(x)). The goal of the hash function is to "disperse" the keys in an apparently random way. In general, we want to avoid having many items being hashed to the same location.

3 General Approaches

There are two general approaches that one can take when designing a hash function:

1. **View the key k as a tuple of integers**: In this approach, the key is seen as a tuple of integers $(x_1, x_2, \ldots, x_d)$ with each integer being in the range $[0, M-1]$ for some M.
2. **View the key k as a large nonnegative integer**: In this approach, the key is seen as a large nonnegative integer that may have arbitrary length.

3.1 Summing Components Approach

The summing components approach is used for keys of the form $k = (x_1, x_2, \ldots, x_d)$.

- $h(k) = \sum_{i=1}^{d} x_i$
- $h(k) = (\sum_{i=1}^{d} x_i) \bmod p$, where p is a prime number
- $h(k) = \sum_{i=1}^{d} x_i \cdot a^{d-i}$, where a is some + integer chosen empirically to avoid collisions.

The first two options may cause problems because the hash codes are invariant under permutations of the key tuple. For example, "mate", "meat", "tame", and "team" all map to the same code. The third option solves this problem by making the hash code dependent on the order of the tuple.

3.2 Modular Division Approach

The modular division approach is used for keys that are positive integers.

- $h(k) = k \bmod N$, where N is some prime number.

If the keys are randomly and uniformly distributed in $[0, M]$ where $M \gg N$, then the probability that two keys collide is $1/N$. However, keys are usually not randomly distributed, so this approach may not work well in practice.

3.3 Universal Hash Functions

A universal hash function is a type of hash function that is designed to produce a relatively uniform distribution of hash values for different input data, regardless of the specific characteristics of the input. To achieve this, a universal hash function uses randomization or other techniques to make the mapping between inputs and hash values unpredictable and independent of the input data. Universal hash functions are commonly used in various applications that require efficient and secure data hashing, such as cryptography, data compression, and data indexing, among others.

3.4 Random Linear Hash Functions

Used on keys k that are positive integers.

- $h(k) = ((ak + b) \bmod p) \bmod N$

Where p is a prime number, and a and b are chosen uniformly at random from the interval $[1, p-1]$ with $a \neq 0$. The random choice of a and b ensures that the mapping between the keys and hash codes is unpredictable and independent of the specific characteristics of the input data. This makes the hash function resistant to certain types of attacks, such as those based on knowledge of the input data. One useful property of Random Linear Hash Functions is that if the keys are in the range $[0, M]$ and $p > M$, then the probability that two keys collide is $1/N$.

4 Collision Handling

4.1 Seperate Chaining

Let each cell in the table point to a linked list holding the entries that map there Get, put, and remove operations are delegated to the appropriate list, where put needs to search through the list to replace the existing value of the key, if present Separate chaining is simple, but requires additional memory outside the table.

```
1. def get(k):
2.     return A[h(k)].get(k)
```

```
1. def put(k,v):
2.     return A[h(k)].put(k,v)
```

```
1. def remove(k):
2.     return A[h(k)].remove(k)
```

Assume that our hash function maps n keys to independent uniform random values in the range $[0, N-1]$. Let X be a random variable representing the number of items that map to a bucket in the array A. Then, $E(X) = n/N$, where the parameter n/N is called the load factor of the hash table, usually denoted as a. The expected time for hash table operations is $O(1+a)$ when collisions are handled with separate chaining. However, the worst-case time is $O(n)$, which happens when all the items collide into a single chain.

4.2 Linear Probing

Linear Probing is a technique for handling collisions in a hash table by placing the colliding item in the next available cell (circularly). In other words, if the initial cell is already occupied, the algorithm checks the next cell, and so on until it finds an empty cell. Each cell inspected during the search for an empty cell is referred to as a probe. Colliding items can end up lumped together, causing future collisions to result in a longer sequence of probes. Linear Probing can be efficient when the load factor of the table is low, but as the table becomes more crowded, the likelihood of long sequences of probes and poor performance increases.

```
1. def get(k):
2.     i = h(k)
3.     p ← 0
4.     repeat
5.         c ← A[i]
6.         if c = null then
7.             return null
8.         else if c.key = k then
9.             return c.value
10.         else
11.             i = (i+1) mod N
12.             p = p + 1
13.    until p = N
14.    return null
```

Updates with Linear Probing
To handle insertions and deletions, we introduce a special object, called DEFUNCT, which replaces deleted elements, to tell them apart from empty cells

- get(k): must pass over cells with DEFUNCT and keep probing until the element is found, or until reaching an empty cell.
- put(k,v): search for the entry as in get(k), but we also remember the index j of the first cell we find that has DEFUNCT or empty. If we find key k, we replace the value there with v and return the previous value. If we don't find k, we store (k, v) in cell with index j Throw exception if table is full.
- remove(k): search for the entry as in get(k). If found, replace it with the special item DEFUNCT and return element v

In the worst case, get, put, and remove take O(n) time. Fact: Assuming hash values are uniformly randomly distributed, expected number of probes for each get and put is 1/(1-a) where a = n/N is the load factor of the hash table. Thus, if the load factor is a constant < 1 then the expected running time for the get and put operations is O(1).

4.3 Cuckoo hashing

Main problem with the methods we've seen so far is that operations take O(n) time in the worst-case. Cuckoo hashing achieves worst-case O(1) time for lookups and removals, and expected O(1) time for insertions. In practice Cuckoo hashing is 20-30% slower than linear probing but is still often used due to its worst case guarantees on lookups.

The Power of Two Choices

The Power of Two Choices is a technique for handling collisions in a hash table that uses two hash tables, $T1$ and $T2$, each of size N, and two hash functions, $h1$ and $h2$, for $T1$ and $T2$ respectively.

For an item with key k, there are only two possible places where we are allowed to store the item: $T1[h1(k)]$ or $T2[h2(k)]$. This restriction simplifies lookup dramatically, while still allowing worst-case $O(1)$ running time for get and remove operations.

The Power of Two Choices works by randomly selecting which of the two hash tables to use for each item, based on the hash codes generated by the two hash functions. By using two hash tables and two hash functions, the technique reduces the likelihood of collisions and spreads the items more evenly across the tables, improving performance.

```
1. def get(k):                                                    # O(1)
2.     if T1[h1(k)] != null and T1[h1(k)].key = k then
3.         return T1[h1(k)].value
4.     else if T2[h2(k)] != null and T2[h2(k)].key = k then
5.         return T2[h2(k)].value
6.     return null
```

```
1. def remove(k):                                                 # O(1)
2.     temp ← null
3.     if T1[h1(k)] != null and T1[h1(k)].key = k then
4.         temp ← T1[h1(k)].value
5.         T1[h1(k)] ← null
6.     else if T2[h2(k)] != null and T2[h2(k)].key = k then
7.         temp ← T2[h2(k)].value
8.         T2[h2(k)] ← null
9.     return temp
```

4.4 High level idea behind put

If a collision occurs in the insertion operation in the cuckoo hashing scheme, then we evict the previous item in that cell and insert the new one in its place.

This forces the evicted item to go to its alternate location in the other table and be inserted there, which may repeat the eviction process with another item, and so on. Eventually,

we either find an empty cell and stop or we repeat a previous eviction, which indicates an eviction cycle.

If we discover an eviction cycle, then we bail out or rehash all the items into larger tables

```
1. def put(k,v):                                              # O(1) expected
2.     if T1[h1(k)] != null and T1[h1(k)].key = k then        # try to fit item into T1
3.         T1[h1(k)] ← (k,v)
4.         return
5.     else if T2[h2(k)] != null and T2[h2(k)].key = k then   # try to fit item into T2
6.         T2[h2(k)] ← (k,v)
7.         return
8.     i ← 1                                                  # start eviction sequence
9.     repeat
10.        if Ti[hi(k)] = null then
11.            Ti[hi(k)] ← (k,v)
12.            return
13.        temp ← Ti[hi(k)]
14.        Ti[hi(k)] ← (k,v)
15.        (k,v) ← temp
16.        i ← i + 1
17.    until a cycle is detected or i > 100
18.    rehash all items into larger tables
```

4.5 How to detect eviction cycles

Use a counter to keep track of the number of evictions. If we evict enough times we are guaranteed to have a cycle. Keep an additional flag for each entry. Every time we evict an entry, we flag it. After a successful put, we need to unflag the entries flagged.

4.6 Performance of Cuckoo Hashing

One can show that "long eviction sequences" happen with very low probability.

Fact: Assuming hash values are uniformly randomly distributed, expected time of n put operations is O(n) provided N > 2n.

Fact: Cuckoo hashing achieves worst-case O(1) time for lookups and removals.

5 Set ADT

A set is an unordered collection of elements, without duplicates that typically supports efficient membership tests. Elements of a set are like keys of a map, but without any auxiliary values.

5.1 Set implemented via Map

- Use a Map to store the keys, and ignore the value.
- Allows `contains(k)` to be answered by `get(k)`.
- Similarly for `add` and `remove`.
- Using `HashMap` for Map, gives main Set operations that usually can be performed in $O(1)$ time.

5.2 MultiSet

- Like a Set, but allows duplicates
- Also called a Bag
- Operation `count(e)` says how many occurrences of `e` in the collection
- `remove(e)` removes ONE occurrence (provided `e` is in the collection already)
- Implement by Map where the element is the key, and the associated value is the number of occurrences.

5.3 Practice vs Theory

- In practice, hash table implementations are usually fast, and people use them as though `put`, `get`, and `remove` take $O(1)$ time.
- The analyses we covered in lecture assume uniformly random hash values, which are not possible to implement in practice. Removing these assumptions is an active area of research well beyond the scope of this class.
- In theory, we do not know of an implementation of hash tables that can perform `put`, `get`, and `remove` in $O(1)$ time in the worst case.