Hashing (Hash Table): Implementation and Runtime Analysis

The Dictionary Data Structure

- Dictionary or map is a data structure.

> **Dictionary/map**: aka. **associative array** is a data structure that consists of a collection of `(key, value)` pairs called an entry of a dictionary.
>
> - Note: the key and the value can be composite.

- Usage of a dictionary:
  - The dictionary data structure is used to store `(key, value)` pairs where the user can loop up (=search) a value using the key.
  - To support its usage, a dictionary must provide the following operations:
    - `size()` : return the number of entries stored inside this dictionary.
    - `put(key, value)` L if `key` is found in the dictionary, it updates the value with `value` . Otherwise, it inserts `(key, value)` into the dictionary.
    - `get(key)` : return the corresponding `value` if `key` is found, and return `null` otherwise.
    - `remove(key)` : remove the dictionary entry containing `key` (and return the corresponding `vaue` if `key` is found).
  - The most frequently used operation is `get(key)` , so fast lookup is required!
- The `Dictionary` interface:

```
public interface Dictionary<K,V> {
    public int size(); // return number of entires in the dictionary
    public void put(K key, V value); // insert or update (key, value) pair
    public V get(K key); // return value associated with key;
                         // return null if not found
    public V remove(K key); // remove entry with key and return corresponding value
}
```

- We can implement the dictionary data structure using:
  - An array

  - A linked list
- For simplicity, we will implement the dictionary using an array. To achieve it, we need to:
  - Definition of the `Entry` class to represent an entry in a dictionary.
  - Definition of a class to represent a dictionary using an array.
  - Implement the (support) methods of the `Dictionary` interface.
- The `Entry` class and the `ArrayMap` implementation:

```java
public class ArrayMap<K,V> implements Dictionary<K,V> {
    /* ---------------- Nested Entry class ---------------- */
    private class Entry<K,V> {
        private K key; // The key (to look up)
        private V value; // the value (corresponding to the key)

        public Entry(K k, V v) { // constructor
            key = k;
            value = v;
        }

        /** Accessor method fvor the key **/
        public K getKey() {
            return key;
        }

        /** Accessor method for the value **/
        public V getValue() {
            return value;
        }

        /** Mutator method for the value **/
        public void setValue(V v) {
            this.value = v;
        }

        @Override
        public String toString() {
            return "(`" + key + "`, `" + value + "`)";
        }
    }
    /* ---------------- End of nested Entry class ---------------- */

    Entry<K,V>[] entry; // Dictionary
    int nEntries; // number of entries in the dictionary

    public ArrayMap(int N) { // Constructor
        entry = new Entry[N];
        nEntries = 0;
    }

    @Override
    public int size() {
```

```
        return nEntries;
    }

    @Override
    public void put(K k, V v) {
        for (int i = 0; i < nEntries; i++) {
            if (entry[i].getkey().equals(k)) {
                // Found
                entry[i].setValue(v); // update the value
                return;
            }
        }
        // Key not found
        entry[nEntries] = new Entry<K,V>(k, v); // insert (k,v)
        nEntries++;
    }

    @Override
    public V get(K k) {
        for (int i = 0; i < nEntries; i++) {
            if (entry[i].getKey().equals(k)) {
                // Found
                return entry[i].getValue();
            }
        }
        // Not found
        return null;
    }

    @Override
    public V remove(K k) {
        boolean found = false; // Indicate key not found
        int loc = -1; // contains index of key
        V ret = null; // contains the return value

        for (int i = 0; i < nEntires; i++) {
            if (entry[i].getKey().equals(k)) {
                found = true; // indicates key k was found
                loc = i; // remember the index of the entry
                break;
            }
        }
```

```
        if (found) {
            // Key found
            ret = entry[loc].getValue(); // update return value
            for (int i = loc + 1; i < nEntries; i++) {
                // delete entry [loc]
                entry[i-1] = entry[i]; // shift array
            }
            nEntries--;
        }
        return ret;
    }
}
```

- Problems with the `ArrayMap` implementation:
  - A dictionary is used to look up information (= `value`) for a given `key` .
  - The loop up operation `get()` is $\mathcal{O}(n)$.
    - Can we do better than $\mathcal{O}(n)$?
      - Yes, we can sort the array and use binary search to reduce the runtime to $\mathcal{O}(\log n)$.
    - Can we do better than $\mathcal{O}(\log n)$?
      - Yes, we can use hashing to reduce the runtime to $\mathcal{O}(1)$.

The Hash Function

- Insight on how to improve the search performance of arrays.
  - Fact on arrays:
    - Array access is very fast if access uses an array index.
  - Fact on dictionaries:
    - Entries in a dictionary are looked up using its key.
  - The problem with the `ArrayMap` implementation of the dictionary is that entries of the dictionary are stored using an index that is not related to the `key` .
  - To improve the search operation for a dictionary stored in an array, we need to find a way to relate (=map) the key `k` to an index `h` of the array:

    ```
    h = hashFunction(k)
    ```

  - This way of storing data into an array is called hashing.
- Hashing functions `H()` :

    - Hash function is a function that maps a key `k` to a number `h` in the range `[0, M-1]`, where `M =` length of the array. That is, $h = H(k)$, where $h \in [0, \cdots, M-1]$.
        - `H()` is consistent: always gives the same answer for a given key.
        - `H()` is uniform: the function values are distributed evenly across `[0...M-1]`.
    - A hash function is usually specified as the composition of 2 functions: $H(k) = H_2(H_1(k))$, where
        - $H_1(k)$ is the **hash code** function that returns the integer value of the key `k`.
        - $H_2(x)$ is a compression function that maps a value x uniformly to the range $[0, M-1]$.
- The hash code of a key:
    - Fact: all data inside a computer is stored as abinary number.
    - The `Object` class in Java contains a `hashCode()` method that returns the data stored in the `Object` as an integer.
    - We can use the `hashCode()` method as our $H_1(k)$ function.
- The compression function $H_2(x)$
    - Notice from the previous discussion on the hash code $H_1(k)$: $H_1(k)$ uses the data stored in the key `k` to compute (deterministically) a hash code value.
    - The compression function $H_2(x)$ has two purposes:
        - Make sure that the return value is in the range $[0, M-1]$. (where M is the length of the array).
        - Scatter/randomize the input value $x = H_1(k)$, so that the value $H_2(x)$ is evenly/uniformly distributed over the range $[0, M-1]$.
    - Why do we use uniform randomization?
        - The array element used to stroe the diction entry is `array index = H(k) = H2(H1(k))`.
        - Uniform randomization will minimize the likelihoo/chance that 2 different keys being hashed to the same value (=array index) (a.k.a. **collision**).
    - A commonly used compression function is the **Multiply Add Divide (MAD)** function:

```
H2(x) = ( (ax + b) % p ) % M,   where p = some prime number
          ^^^^^^^^^^^^
           randomizes
```

        - In the MAD function, `a` and `b` are random numbers, and `p` is a prime number.
        - In the examples of this course, we will use `p = 109345121` (a prime number), `a = 123` and `a = 456`.
        - Note: `p` must be greater than `M` (i.e., `p > M`), otherwise, we will not use the full capacity of the array.

Hash Table

- Terminologies:
  - **Hash function** `H()` : maps a key `k` to an integer in the range `[0, M−1]`.
    `H(k) = integer in [0, M−1]`.
  - **Hash value** `h` : the value returned by the hash function `H()` : `h = H(k)`.
  - **Bucket**: the array element used to store an entry of the dictionary.
  - **Collision**: A collision occurs when 2 different keys `k1` and `k2` have the same hash value.
    `h1≠h2 but H(k1)=H(k2)`.
- If there are `n` entries in a hash table of size `M`, how likely is it that 2 entries hash into the same bucket?

$$
\begin{aligned}
\mathbf{P}(\text{all } n \text{ entries use different buckets}) &= \frac{M(M-1)\cdots(M-n+1)}{M^n} \\
&= \frac{M!}{M^n(M-n)!} \\
\mathbf{P}(2 \text{ entries use the same buckt}) &= 1 - \frac{M!}{M^n(M-n)!}.
\end{aligned}
$$

  - There are 2 techniques to handle collision in hashing:
    - Closed addressing (a.k.a. **Separable Chaining**):
      - Entries are always stored in their hash bucket.
      - Each bucket of the hash table is organized as a linked list.
    - Open addressing:
      - Entries are stored in a different bucket than their hash buckets.
      - A rehash algorithm is used to find an empty bucket.

Closed Addressing (Separate Chaining)

- Previously, we used the `Entry<K,V>` class in the `ArrayMapo<K,V>` implementation to store the dictionary entries.
  - In order to support separate chaining, the `Entry<K,V>` class must be modified to support a linked list.

```java
public class HashTableSC<K,V> implements Dictionary<K,V> {
    /* ---------------- Nested Entry class ---------------- */
    private class Entry<K,V> {
        private K key; // key
        private V value; // value
        private Entry<K,V> next; // link to create a linked list

        public Entry(K k, V V) { // constructor
            key = k;
            value = v;
        }
        /** Accessor method fvor the key **/
        public K getKey() {
            return key;
        }
        /** Accessor method for the value **/
        public V getValue() {
            return value;
        }
        /** Mutator method for the value **/
        public void setValue(V v) {
            this.value = v;
        }
        @Override
        public String toString() {
            return "(`" + key + "`, `" + value + "`)";
        }
    }
    /* ---------------- End of nested Entry class ---------------- */

    public Entry<K,V>[] bucket; // The hash table
    public int capacity; // capacity = bucket.length
    int NItems; // number of entries in the hash table

    // MAD formula: (Math.abs(a * HashCode + b) % p) % M
    public int MAD_p; // prime number in the MAD alg
    public int MAD_a; // multiplier in the MAD alg
    public int MAD_b; // offset in the MAD alg

    public HashTableSC(int M) { // create a hash table of size M
        bucket = (Entry[]) new Entry[M]; // create hash table of size M
        capacity = bucket.length; // capacity of has table
        NItems = 0; // number of entries in the hash table
```

```
    // Initialize MAD parameters
    MAD_p = 109345121; // prime number
    MAD_a = 123; // multiplier
    MAD_b = 456; // offset
}

/** Hash function H(k) **/
public int hashValue(K key) {
    int x = key.hashCode(); // hash code of the key
    return (Math.abs(x * MAD_a + MAD_b) % MAD_p) % capacity;
}

/* ---------------------------------------------
The help method findEntry(k): find the Entry containing key in the hash table
return: Entry object containing key if found
return: null if not found
--------------------------------------------- */
public Entry findEntry(K k) {
    int hashIdx = hashValue(k); // get hash index using key k
    Entry<K,V> curr = bucket[hashIdx]; // curr = first of linked list

    while (curr != null) {
        if (curr.getKey().equals(k)) {
            return curr;
        }
        curr = curr.next;
    }
    return null; // not found
}

@Override
public int size() {
    return NItems;
}

@Override
public void put(K k, V v) {
    int hashIdx = hashValue(k);
    Entry<K,V> h = findEntry(k);
    if (h != null) {
        h.setValue(v); // update value with v
    } else {
```

```java
            // Add newEntry as first element in the list at bucket[hashIdx]
            Entry<K,V> newEntry = new Entry<>(k, v); // make new entry
            newEntry.next = bucket[hashIdx]; // point to the first bucket
            bucket[hashIdx] = newEntry; // make newEntry the first bucket
            NItems++; // increment number of entries
        }
    }

    @Override
    public V get(K k) {
        Entry<K, V> h = findEntry(k);
        if (h != null) {
            return h.getValue();
        } else {
            return null;
        }
    }

    @Override
    public V remove(K k) {
        int hashIdx = hashValue(k);
        // General case delete from linked list
        Entry<K,V> previous = bucket[hashIdx];
        Entry<K,V> current = bucket[hashIdx];

        while (current != null && !current.getValue().equals(k)) {
            previous = current;
            current = current.next;
        }

        if (current != null) { // found
            previous.next = current.next; // unlink current
            NItems--; // decrement number of entries
            return current.getValue();
        }
        return null; // not found
    }
}
```

- Runtime analysis:
    - Consider a hash table using separate chaining. Due to randomization of the hash value,
        - Some entries in the hash table has no keys

        - Some entries in the hash table has exactly 1 key.
        - Some entries in the hash table has more than 1 key.
    - Operations on a hash table always uses the hash value. The hash value will select one specific hash bucket.
        - The search key will be:
            - Found in this hash bucket, or
            - Not found in this hash bucket.
    - Therefore, operations on a hash table will always examine all keys in one search bucket.
    - Therefore, the running time of operations on a hash table is equal to the number of entries stored inside one bucket in the hash table.
        - Problem: how many entries will be stored inside 1 bucket?
        - Fact: A search key that has hash value `k` is stored in the bucket `k`.
        - Therefore, number of entries in bucket `k` is the number of keys where `H(key) = k`.
        - Now, let's estimate the number of entries stored in a bucket.
        - By the uniformity assumption, the random hash value `H(key)` is uniformly distributed over the range `[0, M−1]`. Then, each outcome is equally likely with probability of `1/M`.
        - Suppose there are a total of `n` items/entries hashed and stored in the hash table. According to the theory of probability, the number of items/entries in any bucket has a binomial probability distribution of $\mathbf{BIN}\left(n, p = \frac{1}{M}\right)$.
        - Then, the average number of entries in 1 bucket is $\frac{n}{M}$. So, the average running time for hash operations is $\frac{n}{M} \sim \mathcal{O}(n)$.

Open Addressing

- Closed addressing vs Open addressing:
    - Closed addressing:
        - In closed addressing, each key is always stored in the hash bucket where the key is hashed to.
        - Close addressing must use some data structure (e.g. linked list) to store multiple entries in the same bucket.
    - Open addressing:
        - In open addressing, each hash bucket will store at most one hash table entry.
        - In open addressing, a key may be stored in different bucket than where they key was hashed to.
        - Entries used in open addressing:
            - Since in open addressing, each hash bucket will store at most one hash table entry, the entries stored in open address do not have a link variable.

        - Therefore, the `Entry<K,V>` class used in open addressing is different from the `Entry<K,V>` class used in closed addressing. In fact, we can use the `Entry<K,V>` defined the `ArrayMap<K,V>` implementation.
- Collision resolution in Open Addressing:
    - If a key is hashed to a bucket that is already occupied, we need to find another bucket to store the key. This process will be completed with an insert algorithm.
    - The insert algorithm will start at the hash index and find the next variable hash bucket that can be used to store the key.
    - The procedure to find the next available hash bucket is called **rehashing**.
        - Note: rehashing is not random but deterministic (=computable).
- Commonly used Rehashing Algorithms to Resolve Collision in Open Addressing:
    - Linear Probing: in linear probing, the hash table is searched sequentially starting from the hash index value.
        - In other words, the rehash function is `Rehash(key) = (h + i)%M`, where `h = H(key)` and `i = 1, 2,...`
    - Quadratic Probing: uses the following rehash function: `Rehash(key) = (h + i^2)%M`, where `h = H(key)` and `i = 1, 2,...`
    - Double hashing: uses the following rehash function: `Rehash(key) = (h + i*H2(key))%M`, where `h = H(key)`, `h' = H'(key)` is a second hash function, and `i = 1, 2,...`
- The code for linear probing without `remove()`:

```java
public class HashTableLP<K,V> {
    /* ---------------- Nested Entry class ---------------- */
    private class Entry<K,V> {
        private K key;   // The key (to loop up)
        private V value; // The value (corresponding to the key)
        public Entry(K k, V v) { // Constructor
            key = k;
            value = v;
        }
        public K getKey() { // Accessor method for the key
            return key;
        }
        public V getValue() {  // Accessor method for the value
            return value;
        }
        public void setValue(V value) {         // Mutator method for the value
            this.value = value;
        }
        public String toString() {
            return "(" + key + "," + value + ")";
        }
    }
    /* ---------------- End of nested Entry class ---------------- */

    public Entry<K,V>[] bucket; // The Hash table
    public int capacity;        // capacity == bucket.length
    int    NItems;              // # items in hash table
    // MAD formula: ( Math.abs(a * HashCode + b) % p ) % M
    public int MAD_p;           // Prime number in the Multiply Add Divide alg
    public int MAD_a;           // Multiplier   in the Multiply Add Divide alg
    public int MAD_b;           // Offset       in the Multiply Add Divide alg

    // Constructor
    public HashTableLP(int M) {  // Create a hash table of size M
        bucket = (Entry[]) new Entry[M]; // Create a hash table of size M
        capacity = bucket.length;        // Capacity of this hash table
        NItems = 0;                      // # items in hash table

        MAD_p = 109345121;               // We pick this prime number...
        MAD_a = 123;                     // a = non-zero random number
        MAD_b = 456;                     // b = random number
    }
```

```java
    // The hash function for the hash table
    public int hashValue(K key) {
        int x = key.hashCode(); // Uses Object.hashCode()
        return ((Math.abs(x*MAD_a + MAD_b) % MAD_p) % capacity);
    }
    public int size() {
        return NItems;
    }

    public void put(K k, V v) {
        int hashIdx = hashValue(k); // find the hash index for key k
        int i = hashIdx;
        do {
            if (bucket[i] == null) { // is entry empty?
                bucket[i] = new Entry<K,V>(k, v);
                return;
            } else if (entry[i].getKey().equals(k)) { // is entry k?
                entry[i].setValue(v); // update value
                return;
            }
            i = (i + 1) % capacity; // rehash
        } while (i != hashIdx); // all entires searched!
        System.out.println("Full");
    }

    public V get(K k) {
        int hashIdx = hashValue(k); // find the hash index for key k
        int i = hashIdx;
        do {
            if (bucket[i] == null) { // is entry empty?
                return null;
            } else if (entry[i].getKey().equals(k)) { // is entry k?
                return entry[i].getValue(); // return value
            }
            i = (i + 1) % capacity; // rehash
        } while (i != hashIdx); // all entires searched!
        return null; // not found
    }
}
```

- Now, let's consider the `remove()` method. If we remove the entry stored in `bucket[i]`, then we will not be able to find the entry stored in `bucket[i+1]`.
    - Therefore, we need to move the entry stored in `bucket[i+1]` to `bucket[i]`.

- However, if we move the entry stored in `bucket[i+1]` to `bucket[i]` , then we will not be able to find the entry stored in `bucket[i+2]` .
- That means, instead of simply moving the entry stored in `bucket[i+1]` to `bucket[i]` , we need alternative method to solve this problem.

- To solve the deletion problem, a hash table using open addressing uses a special entry called `AVAILABLE` :

```
public Entry<K,V> AVAILABLE = new Entry<>(null, null);
```

  - When an existing entry in the hash table is removed, the entry is replaced by the `AVAILABLE` entry.
  - When we are searching for key `k` , then
    - `AVAILABLE` must be treated as an empty bucket (i.e., it does not contain any key).
    - The rehash algorithm must continue with the next search location.

```java
public class HashTableLP<K,V> implements Dictionary<K,V> {
    /* ---------------- Nested Entry class ---------------- */
    private class Entry<K,V> {
        private K key;   // The key (to loop up)
        private V value; // The value (corresponding to the key)
        public Entry(K k, V v) { // Constructor
            key = k;
            value = v;
        }
        public K getKey() { // Accessor method for the key
            return key;
        }
        public V getValue() {  // Accessor method for the value
            return value;
        }
        public void setValue(V value) {         // Mutator method for the value
            this.value = value;
        }
        public String toString() {
            return "(" + key + "," + value + ")";
        }
    }
    /* ---------------- End of nested Entry class ---------------- */

    public Entry<K,V>[] bucket; // The Hash table
    public int capacity;        // capacity == bucket.length
    int    NItems;              // # items in hash table
    // MAD formula: ( Math.abs(a * HashCode + b) % p ) % M
    public int MAD_p;           // Prime number in the Multiply Add Divide alg
    public int MAD_a;           // Multiplier   in the Multiply Add Divide alg
    public int MAD_b;           // Offset       in the Multiply Add Divide alg

    public Entry<K,V> AVAILABLE = new Entry<>(null, null); // special entry for remove(

    // Constructor
    public HashTableLP(int M) {  // Create a hash table of size M
        bucket = (Entry[]) new Entry[M]; // Create a hash table of size M
        capacity = bucket.length;        // Capacity of this hash table
        NItems = 0;                      // # items in hash table

        MAD_p = 109345121;               // We pick this prime number...
        MAD_a = 123;                     // a = non-zero random number
        MAD_b = 456;                     // b = random number
```

```java
}

// The hash function for the hash table
public int hashValue(K key) {
    int x = key.hashCode(); // Uses Object.hashCode()
    return ((Math.abs(x*MAD_a + MAD_b) % MAD_p) % capacity);
}

@Override
public int size() {
    return NItems;
}

@Override
public void put(K k, V v) {
    int hashIdx = hashValue(k); // find the hash index for key k
    int i = hashIdx;
    int firstAvail = -1; // -1 means: no AVAILABLE entry found

    do { // search for key k
        if (bucket[i] == null) { // is entry empty?
            if (firstAvail = -1 ) { // No AVAILABLE entry found
                bucket[i] = new Entry<K,V>(k, v);
                // insert (k,v) in this empty bucket
            } else { // AVAILABLE entry found
                bucket[firstAvail] = new Entry<K,V>(k, v);
                // insert (k,v) in the first AVAILABLE bucket
            }
            return;
        } else if (bucket[i] == AVAILABLE) {
            if (firstAvail == -1) {
                firstAvail = i; // remember the first AVAILABLE entry
            }
        } else if (entry[i].getKey().equals(k)) { // is entry k?
            entry[i].setValue(v); // update value
            return;
        }
        i = (i + 1) % capacity; // rehash
    } while (i != hashIdx); // all entires searched!

    if (firstAvail == -1) {
        System.out.println("Full");
    } else {
```

```
            bucket[firatAvail] = new Entry<>(k,v);
        }
    }

    @Override
    public V get(K k) {
        int hashIdx = hashValue(k); // find the hash index for key k
        int i = hashIdx;
        do {
            if (bucket[i] == null) { // is entry empty?
                return null;
            } else if (bucket[i] == AVAILABLE) {
                // Do NOT Test bucket[i]
                // continue
            } else if (entry[i].getKey().equals(k)) { // is entry k?
                return entry[i].getValue(); // return value
            }
            i = (i + 1) % capacity; // rehash
        } while (i != hashIdx); // all entires searched!
        return null; // not found
    }

    @Override
    public V remove(K k) {
        int hashIdx = hashValue(k);
        int i = hashIdx;

        do {
            if (bucket[i] == null) { // Is bucket empty?
                return null; // Not found
            } else if (bucket[i] == AVAILABLE) {
                // Do NOT Test bucket[i]
                // continue
            }  else if (bucket[i].getKey().equals(k)) { // does bucket contain k?
                V retVal = bucekt[i].getValue();
                bucket[i] = AVAILABLE; // mark as deleted
                return retVal;
            }
            i = (i + 1) % capacity; // rehash
        } while (i != hashIdx); // all entires searched!
        return null; // not found
    }
}
```

- Clustering in Learning Hashing:
  - Suppose the hash table currently stores the entries as follows:

```
             0   1   2   3   4   5   6   7   8   9
entry[] = |    | A | B | C | D | E | F | G | H |    |
```

    - Then, if we want to insert a key `k` with a hash value in the range `[1...9]` m we will have to store it in the bucket 9.
    - This is called **clustering**.
  - To alleviate clustering, other rehashing methods can be used:
    - Quadratic Probing
    - Double hashing.

Running Time Analysis

- Strength and Weakness of a Hash Table
  - A hash table is fast when entries are not clustered.
  - In this case, the running time of operations such as `get()`, `pu()` and `remove()` is $\mathcal{O}(1)$.
    - The search will find the key immediately in the hash bucket.
    - Or else, the search will terminate in the next step because it finds an empty (`null`) bucket.
  - A hash table is slower when entries are clustered. In those cases, we need more comparison operations.
- Worse case running time of hashing with linear probing: when the hash table is full.
  - Then, `get()`, `pu()`, and `remove()` may need to scan the entire hash table to find the entry.
  - Therefore, worse case running time of linear probing is `n/2` : The scan will examine approximately half of all the entries.
- Average case running time analysis of linear probing:
  - Consider the `get()` algorithm using linear probing. The `get()` method will return when it find
    - an empty bucket, or
    - the key `k`
  - Consider the `put()` algorithm using linear probing. The `put()` method will return when it find
    - an empty bucket, or
    - the key `k`
  - Consider the `remove()` algorithm using linear probing. The `remove()` method will return when it find

        - an empty bucket, or
        - the key `k`
    - Simplifying assumption: to keep the running time analysis simple, we will assume that where are no `AVAILABLE` entries in the hash table.
    - From the observation of `get()`, `put()`, and `remove()` algorithms:
        - The running time of them depends on the number of entries we need to check in order to find the key `k` or an empty bucket.
        - So, the worse case running time is when the search ends by finding an empty bucket (takes longer time).
        - Therefore, average running time of `get()`, `put()`, and `remove()` = average number of compare operations to find an empty bucket.
- Load factor and the probability of finding an empty bucket.

> **Load factor**: a.k.a. **occupancy level** is defined as
>
> $$\alpha = \frac{\text{number of entries in hash table}}{\text{size of the hash table}} = \frac{n}{M}.$$
>
> - The load factor α is a measure of how full the hash table is.

    - Then, the probability (=likelihood) that a hash bucket is occuped is

$$\begin{aligned}\mathbf{P}(\text{bucket } i \text{ is occupied}) &= \frac{\text{number of entries in the hash table}}{\text{total number of buckets in the hash table}} \\ &= \alpha.\end{aligned}$$

    - So, the probability (=likelihood) that a hash bucket is empty is $\mathbf{P}(\text{buket } i \text{ is empty}) = 1 - \alpha$.
- The average running time of `get()`, `put()`, and `remove()` is found by computing:
    - How often (frequent) do we need to check 1 entry to find an empty slot (=f_1)? How many operations did we perform in this case? (=c_1)
    - How often (frequent) do we need to check 2 entry to find an empty slot (=f_2)? How many operations did we perform in this case? (=c_2)
    - ...
    - The average running time of `get()`, `put()`, and `remove()` is equal to

$$\text{Average running time} = f_1 c_1 + f_2 c_2 + f_3 c_3 + \cdots$$

- How often do we need to check 1 entry to find an empty slot?
    - The probability of finding a bucket to be empty = $1 - \alpha$.
    - We check 1 entry (=the hash bucekt) and fids an empty bucket.

$$\mathbf{P}(\text{check 1 bucket to find an empty bucket}) = 1 - \alpha = f_1$$
$$\text{number of check operations performed} = 1 = c_1$$

- Similarly, in the case of checking 2 entries to find an empty bucket, we have:

$$\mathbf{P}(\text{check 2 bucket to find an empty bucket}) = \alpha(1 - \alpha =) f_2$$
$$\text{number of check operations performed} = 2 = c_2$$

- So, we know the average running time of `get()` , `put()` , and `remove()` is equal to

$$\begin{aligned}\text{Average running time} &= f_1c_1 + f_2c_2 + \cdots + f_nc_n \\ &= (1-\alpha)\cdot 1 + \alpha(1-\alpha)\cdot 2 + \alpha^2(1-\alpha)\cdot 3 + \cdots \\ &= (1-\alpha)[1 + 2\alpha^1 + 3\alpha^2 + 4\alpha^3 + \cdots]\end{aligned}$$

  - Suppose

$$S = 1 + 2\alpha^1 + 3\alpha^2 + 4\alpha^3 + \cdots .$$

    To compute the sum, we used MATLAB

```
syms a k
assume(a > 0 & a < 1)
symsum((k+1)*(a^k), k, 0, inf)

>>> ans =
              1/(a - 1)^2
```

  - So, the average running time of `get()` , `put()` , and `remove()` is equal to

$$(1-\alpha)\cdot\frac{1}{(1-\alpha)^2} = \frac{1}{1-\alpha}.$$

- Summary:
  - α = the load factor or occupancy level.
  - The probability (=likelihood) of finding a bucket to be empty = $1 - \alpha$.
  - The average runtime of `get()` , `put()` , and `remove()` is the average number of compare operations performed to find an empty bucket. This quantity is equal to $\frac{1}{1-\alpha}$.
  - Example: If $\alpha = 10\%$, then (because 90% of the time we find an empty bucket), average number bucekts searched is `1/(1-0.1) = 1/0.9 ~= 1.1` .

Double Hashing

- Consequence of increasing/decreasing the hash table size:
  - Due to the dependency of the hash function on the array size `M` , we have the following unfortunate consequence: **Changing the array size will also change the hash function.**
  - This means: the entries stored using the old hash function cannot be found using the new hash function.
  - In other words, when we increase/decrease the hash table size, we must rehash all the entries using the new hash function.
- Naïve way to increase/decrease the hash table size.
  - Because the hash function changes with the hash table size, we must rehash all the keys and insert them into the new hash table.
  - A naïve way to do this is to create a new hash table with the new size, and then insert all the keys into the new hash table.

```
public void doubleHashTable() {
  Entry[] oldBucket = bucket; // save the old hash table

  // Double the size of the bucket
  bucket = (Entry[]) new Entry[2 * oldBucket.length];
  capacity = 2 * oldBucket.length;

  // Rehash all the entries in the old hash table
  for (int i = 0; i < oldBucket.length; i++) {
      if (oldBucket[i] != null && oldBucket[i] != AVAILABLE) {
          this.put(oldBucket[i].getKey(), oldBucket[i].getValue());
      }
  }
}
```