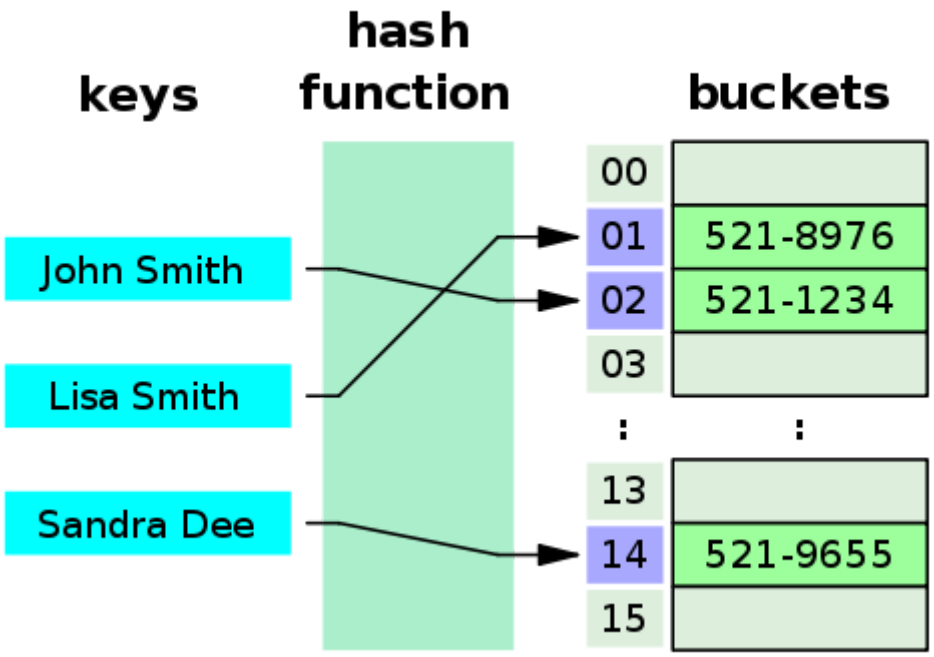
Hash Tables

What is Hashing?

Hashing is a way to take our data and run it through a function, which will return a small, simplified reference generated from that original data. Hashing is one way; you can't convert the hashed data into it's original version. Thus, we can take a complex object, and hash it down to a single integer representation used as an index in the array of buckets.



Hashing has three key components:

1. Hash Functions
2. Hash Table
3. Collisions
    - Collision Resolution Techniques

Hash Function

It takes the key and produce an integer (used as an array index).

Our goal is to take any key and hash it to an array index. We have three primary requirements in implementing a good hash function for a data type:

- Equal keys must produce the same hash value.
- Hashing in efficient amount of time.
- Each key has the same probability to be hashed into each array index.

This is what's called **Uniform Hashing Assumption**.
In the real world, we will develop a hash function for every key type. In Java, types like Strings, and Integers already have their own hash function implemented.

Java's Hash Function
All Java classes inherits hashCode() method, which returns 32-bit integer.

Java requires that if x.equals(y), then their hashCode() should be equal too. That is, if a.equals(b) is true, then a.hashCode() must have the same numerical value as b.hashCode().
It's highly desirable that if !x.equals(y), then their hashCode() is not equal, but, you can't always guarantee that since collision may occur.

The default implementation returns memory address of the current object. Java has customized implementation for standard data types, such as:

Integer
It returns the value of the integer itself.

```
public final class Integer {
    private final int value;
    // ...

    public int hashCode() { return value; }
}
```

Boolean
It returns 1231 if true, and 1237 if false.

```
public final class Boolean {
    private final boolean value;
    // ...

    public int hashCode() {
        if (value) return 1231;
        else       return 1237;
    }
}
```

Double
It returns a 32-bit integer after alternating the binary representation of the double value (64-bit)

```
public final class Double {
    private final double value;
    // ...

    public int hashCode() {
        long bits = doubleToLongBits(value);
        return (int) (bits ^ (bits >>> 32));
    }
}
```

String
Returns a hash code for this string. The hash code for a String object is computed as: s[0]*31^(n-1) + s[1]*31^(n-2) + … + s[n-1].
The hash value of the empty string is zero.

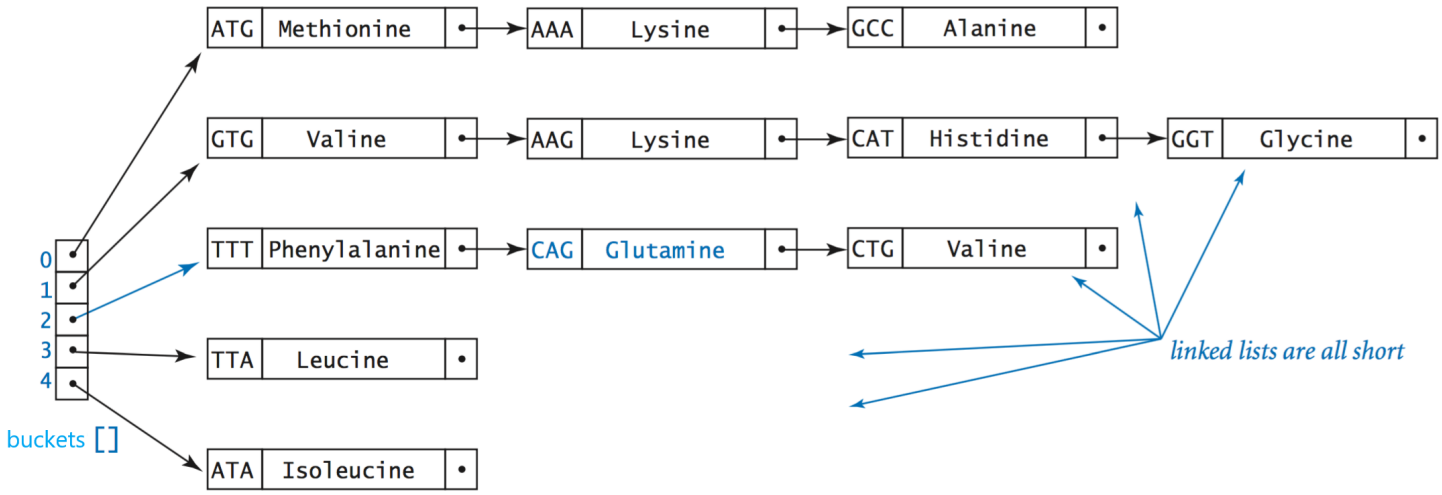
```
public final class String {
    private final char[] s;
    private int hash = 0;    // cache the hashed value
    // ...

    public int hashCode() {
        int h = hash;
        if (h != 0) return h;       // return the cached value
        for (int i = 0; i < length(); i++)
            h = s[i] + (31 * h);
        hash = h;                   // store the hashed value
        return hash;
    }
}
```

User-defined Hash Function: We can also implement a hash function for user's defined data types.

Hash Tables Concepts

A hash table is a data structure in which we use a hash function to divide the keys into m groups, which we expect to be able equal in size. For each group, we keep the keys in an unordered linked list and use sequential search.

Hash table is a generalization of array. Given a key k, we find the element whose key is k by just looking in the kth position of the array. This is called direct addressing.



Load Factor

```
load factor = (number of items stored) / size of the table
```

This is the decision parameter used when we want to re-hash or expand the existing hash table entries. This also helps us in determining the efficiency of the hashing function. That means, it tells whether the hash function is distributing the keys uniformly or not.

Collisions

Collisions is having two distinct keys hashing to same index. Collisions in hash tables are unavoidable.

Collision Resolution Techniques

- **Direct Chaining:** An array of linked list application
  - Separate chaining / aka - open hashing
- **Open Addressing:** Array-based implementation / aka - closed hashing
  - Linear probing (linear search)
  - Quadratic probing (non-linear search)
  - Double hashing (use two hash functions)
  - Cuckoo hashing

1. *Separate Chaining:*
   We maintain for each array entry, a linked list of the key-value pairs, in a hash table of size M. When two or more records hash to the same location, these records are constituted into a singly-linked list called a chain.
2. *Open Addressing:*
   Open addressing is a method for handling collisions. In Open Addressing, all elements are stored

in the hash table itself. So at any point, size of the table must be greater than or equal to the total number of keys.

Below are generic logic for open addressing:

- Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.
- Delete(k): (**slightly different**) If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted".
  Insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

a. **Linear Probing**
The interval between probes is fixed at 1. In linear probing, we search the hash table sequentially, starting from the original hash location. If a location is occupied, we check the next location. We wrap around from the last table location to the first table location if necessary.

The function for rehashing is the following:

```
h(k, i) = (h'(k) + i) % m

h'(k): U -> {0, 1 ... m - 1}  ordinary hash function
for i = 0, 1, .... m - 1
```

Given key k, we first probe T[h'(k)], i.e. the slot given by the auxiliary hash function. We next probe slot T [h'(k) + 1], and so on up to slot T[m - 1]. Then we wrap around to slots T[0], T[1], ... until we finally probe slot T [h'(k) - 1]

Linear probing is easy to implement, but it suffers from a problem known as *primary clustering*. Long runs of occupied slots build up, increasing the average search time. Clusters arise because an empty slot preceded by i full slots gets filled next with probability (i + 1)/m. Long runs of occupied slots tend to get longer, and the average search time increases.

> Knuth was able to show if N = M/2 , then mean displacement is ~ 3/2, but, as the array gets full, it's much higher. The displacement is the number of steps to find a free index in array starting from index i = hash(key).

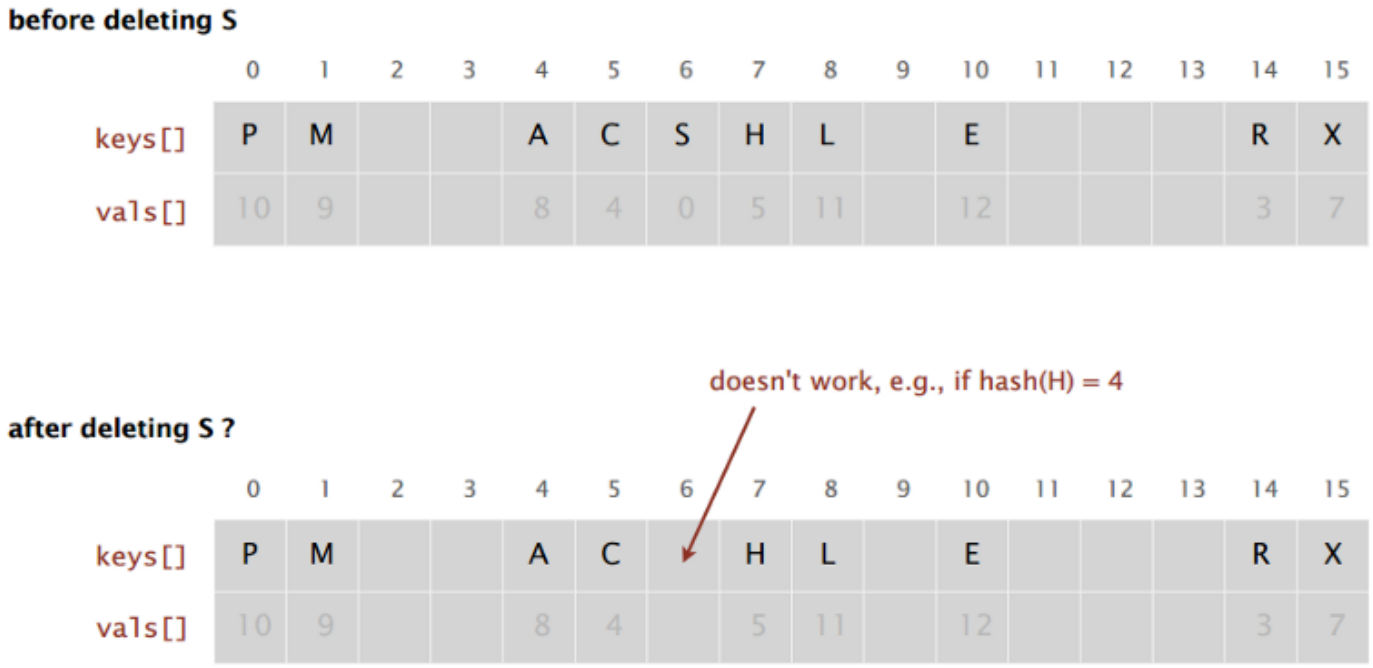
Another approach is to be probe by the step-size (more than one increment) are used. The step-size should be relatively prime to the table size, if we choose the table size to be a prime number, then any step-size is relatively prime to the table size. Clustering cannot be avoided by larger step-sizes.

In number theory, two integers a and b are said to be relatively prime, mutually prime, or co-prime if the only positive integer (factor) that divides both of them is 1.

deletion of a key in linear probing

Setting the key's table position to null will not work, because that might prematurely terminate the search for a key that was inserted into the table later. As a consequence, we need to re-hash/re-insert into the table all of the keys in the cluster to the right of the deleted key.

check code for steps.



b. **Quadratic Probing**

The interval between probes increases proportionally to the hash value (the interval thus increasing linearly, and the indices are described by a quadratic function). The problem of Clustering can be eliminated if we use the quadratic probing method.

In quadratic probing, we start from the original hash location i. If a location is occupied, we check the locations i + 1^2 , i + 2^2, i + 3^2, i + 4^2... We wrap around from the last table location to the first table location if necessary.

The function for rehashing is the following

```
rehash(key) = ( n + k^2 ) % tableSize
```

```
Insert keys in hash table of size 11:
31 mod 11 = 9
19 mod 11 = 8
2 mod 11 = 2
13 mod 11 = 2 → 2 + 1^2 = 3
25 mod 11 = 3 → 3 + 1^2 = 4
24 mod 11 = 2 → 2 + 1^2, 2 + 2^2 = 6
21 mod 11 = 10
9 mod 11 = 9 → 9 + 1^2 mod 11, 9 + 2^2 mod 11, 9 + 3^2 mod 11 = 7
```

Even though clustering is avoided by quadratic probing, still there are chances of clustering.

c. **Double Hashing**

The interval between probes is computed by another hash function. Double hashing reduces clustering in a better way. The increments for the probing sequence are computed by using a second hash function.

The second hash function h2 should be:

```
(hash1(key) + i * hash2(key)) % TABLE_SIZE
Here hash1() and hash2() are hash functions
hash2(key) != 0 , and hash2 != hash1
```

We first probe the location h1(key). If the location is occupied, we probe the location h1(key) + h2(key) % table-size, h1(key) + 2 * h1(key) % table-size... and so on. A popular second hash function is : hash2(key) = PRIME – (key % PRIME) where PRIME is a prime smaller than the TABLE_SIZE.

```
Hash Function: h1(key) = key mod 11 and h2(key) = 7 – (key mod 7) & table size is 11

Insert keys:
58 mod 11 = 3
14 mod 11 = 3 → ( 3 + 7 ) % 11 = 10
91 mod 11 = 3 → ( 3 + 7 ) % 11 = 10, ( 3 + 2 * 7 ) % 11 = 6
25 mod 11 = 3 → 3, ( 3 + 2 * 3 ) % 11 = 9
```

Separate Chaining	Open Addressing
Chaining is Simpler to implement	Open Addressing requires more computation.
In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care for to avoid clustering and load factor.

Separate Chaining	Open Addressing
Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn’t map to it.
Chaining uses extra space for links.	No links in Open addressing

How Hashing Gets O(1) Complexity?
The O(1) claim while technically false, is approximately true for many real world situations. The position of each key in the array is determined by the hash function. We shall assume that the hash function produces output in O(1).

So when we insert something into the hash table, we use the hash function (let's call it h) to find the location where to put it, and put it there. Each time we insert data, it takes O(1) time to insert it (since the hash function is O(1). Looking up data is the same.
If we want to find a value, x, we have only to find out h(x), which tells us where x is located in the hash table. So we can look up any hash value in O(1) as well.

Why it is technically false?
What if we insert data and there is already something in that position of the array? There is nothing which guarantees that the hash function won't produce the same output for two different inputs. Therefore, when we insert we need to take one of two strategies:

- Store multiple values at each spot in the array (say, each array slot has a linked list). Now when you do a lookup, it is still O(1) to arrive at the correct place in the array, but potentially a linear search down a (hopefully short) linked list. This is called "separate chaining".
- If you find something is already there, hash again and find another location. Repeat until you find an empty spot, and put it there.
  The lookup procedure can follow the same rules to find the data. Now it's still O(1) to get to the first location, but there is a potentially (hopefully short) linear search to bounce around the table till you find the data you are after. This is called "open addressing".

Basically, both approaches are still mostly O(1) but with a hopefully-short linear sequence. We can assume for most purposes that it is O(1). If the hash table is getting too full, those linear searches can

become longer and longer, and then it is time to "re-hash" which means to create a new hash table of a much bigger size and insert all the data back into it.