# Chapter 11: Hash Tables

Hash tables allow us to implement a dictionary (or map) with an average time of $O(1)$, but the worst-case is $\Theta(n)$. However, with careful design the worst-case can be avoided most of the time.

To search an element in a hash table:
Worst-case: $\Theta(n)$ (no better than a linked list)
Average-case: $O(1)$

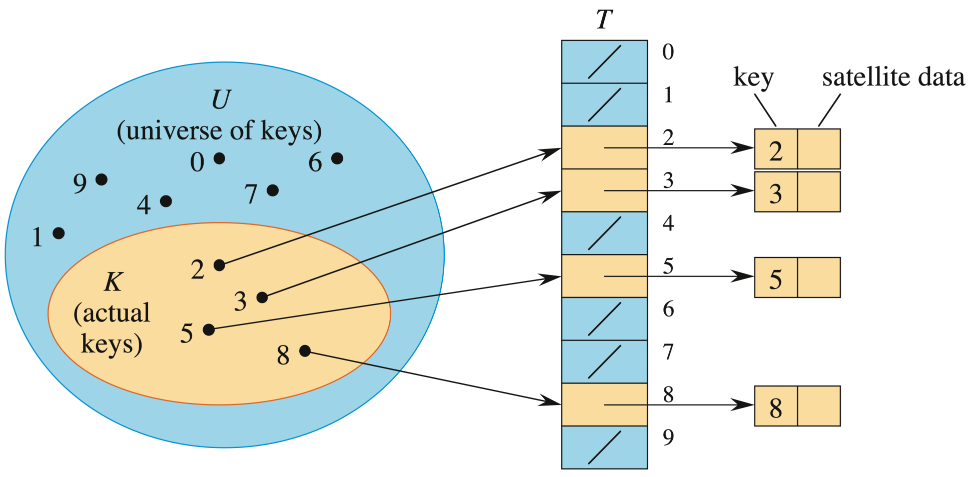
Let us motivate hash tables by first considering a simpler idea that also implements a set.

## Direct-address Tables

- Each element to be stored has a unique key.
- Suitable for applications with small set $U$ (U stands for Universe, the set of all possible keys).
- Suppose $U = \{0, 1, \ldots, m-1\}$. To store a dynamic set of elements, a direct-address table $T[0 \ldots m-1]$ is allocated with each position called a slot, where

$$T[i] = \begin{cases} x & \text{if the element with key } i \text{ is stored, where } x \text{ points to the element} \\ NIL & \text{otherwise} \end{cases}$$

Ex: $U = \{0, 1, \ldots, 9\}$ and the set of elements (keys) stored = $\{2, 3, 5, 8\}$.



With this setup, now we can provide procedures for search, insert, and delete operations.

```
Direct-Address-Search(T, k)
1. return T(k)

Direct-Address-Insert(T, x)
1. T[x.key] = x


Direct-Address-Delete(T, x)
1. T[x.key] = NIL
```

- Each of those operations takes $\Theta(1)$ worst-case runtime. This sounds amazing but we are paying the price of requiring an array of size $m$. This idea of a time-space trade-off is a common pattern in computer science (and most aspects of life!).
- It's impractical to use direct addressing if $\mid U \mid = m$ is large.

  *Example 1*: If $U$ is the set of all possible social security numbers, then $T$ requires $10^9$ = 1 billion entries.

  *Example 2*: If $U$ is the set of all possible 16-digit credit card numbers, then $T$ requires $10^{16} = 10,000$ trillion entries!!
- **Recommended Exercises**: Exercise 11.1-1, 11.1-3.

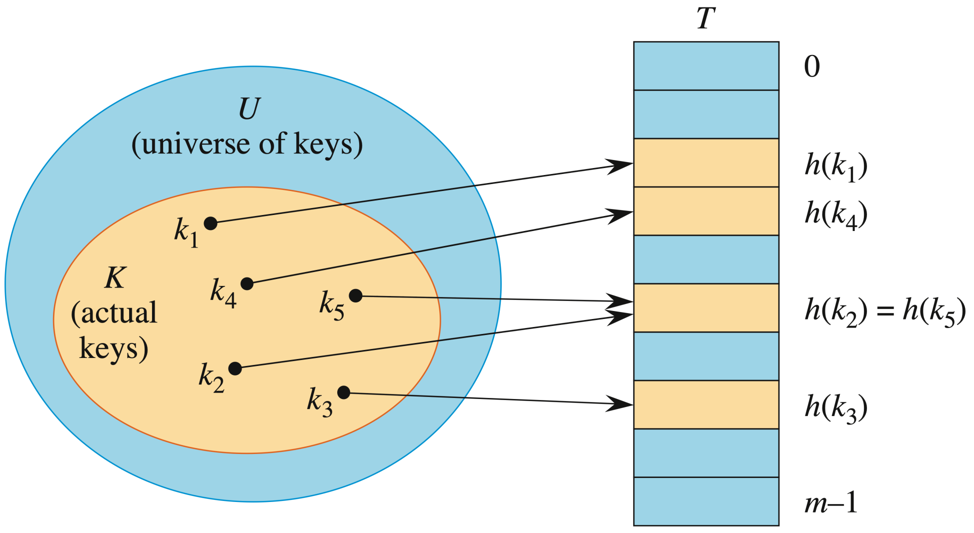
## Hash Tables

- We can reduce the memory requirement to $\Theta(n)$ and still have $\Theta(1)$ average searching time by hashing technique, where $n$ is the number of elements stored.

  **Direct-addressing**: an element with key k is stored in entry k.

  **Hashing**: an element with key $k$ is stored in entry $h(k)$, where $h$ is the **hash function** and $h(k)$ is the hash value.

  For a hash table $T[0..m-1]$, the hash function $h : U \rightarrow \{0, 1, \ldots, m-1\}$ (so it maps each possible key to some slot in the table).
- The hashing technique is useful for applications that $|U| >> m$ (the size of table $T$).
- However, now collisions may occur when we have two (or more) keys $k_1, k_2$ such that $h(k_1) = h(k_2)$). We reduced the amount of space used, so we now pay the price of having to do more work dealing with collisions!



**Solution for collision:**

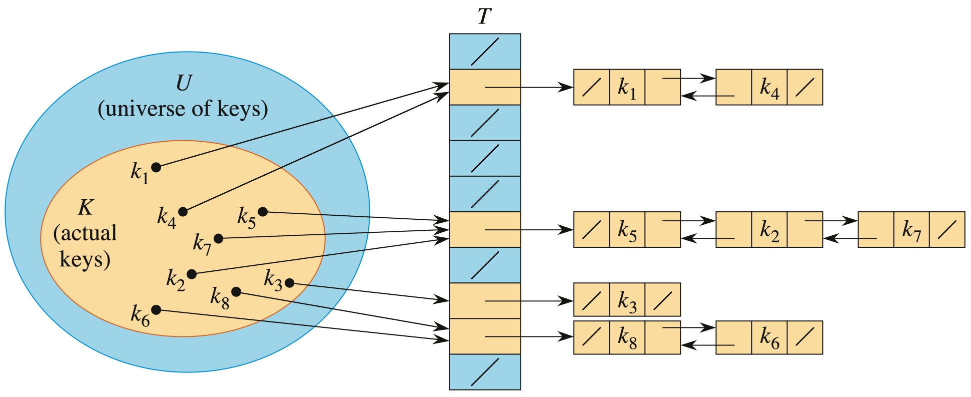
- avoid the collisions altogether $\rightarrow$ impossible unless we revert back to direct address tables
- choose $h$ to be "random" to minimize the # of collisions. Ideally, we want $h$ to be an independent uniform hash function, that is the function maps keys to any slot in the table with equal probability. We will assume such a function for the analysis and then show how to approximate it in practice.

Two topics to discuss further:

1. How to resolve the collisions?
2. What is a good choice of $h$?

## Collision resolution by chaining

We put all elements colliding on one entry into a doubly-linked list. For example:



```
Chained-Hash-Insert(T, x)
1. List-Prepend(T[h(x.key)], x)
```

The insert operation takes $\Theta(1)$ worst-case runtime.

```
Chained-Hash-Search(T, k)
1. return List-Search(T[h(k)], k)
```

The search operation takes $\Theta(n)$ worst-case runtime. However, the worst-case is very unlikely with a good hash function. Thus, in this case, we are more interested in the average case runtime.

**Average time**: $\Theta(1+\alpha)$, where $\alpha = n/m$ is the load factor of the table $T$.

This is based on *Theorem 11.1* and *Theorem 11.2* in the textbook. We will skip the proofs of those theorems though for this class. An informal argument is based on the observation that $n$ elements being distributed uniformly across $m$ lists would give us an average size of $\alpha = n/m$. We add the 1 because $n/m$ may be less than 1 and we will have at least one step.

```
Chained-Hash-Delete(T, x)
1. List-Delete(T[h(x.key)], x)
```

Using a doubly-linked list, we can do the delete operation in $\Theta(1)$ worst-case runtime.

## Hash Functions

- Good hash function: **simple uniform hashing**. Each key is equally likely to hash to any of the $m$ entries.
- Assumption: Let the domain of keys be the set of natural numbers.
- If the keys are not numbers (e.g., strings), they should be mapped to numbers before hashing.

### The division method

$h(k) = k \bmod m$.

- The division method is almost simple uniform.
- The value $m$ should not be a power of 2. If $m = 2^p$, then $h(k)$ is the $p$ low-order bits of $k$. It is better to make hash function depends on all bits of the key.

- Good choice for $m$: primes that are not too close to exact powers of 2.

*Example*: To hold 2000 keys and 3 elements examined in average for an unsuccessful search. The collision is resolved by chaining. What the table size $m$ should be?

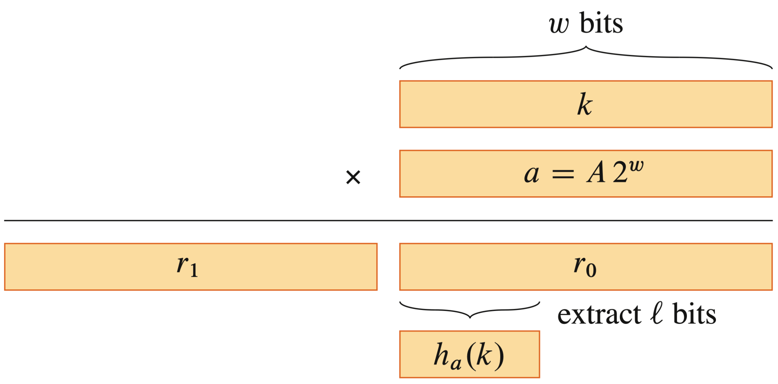
Solution: $m = 701$, since 701 is a prime and $701 \simeq 2000/\alpha$, and also 701 is not close to any power of 2.

## Multiplication method

Choose a constant $A$ such that $0 < A < 1$. Then we multiply the constant by the key $k$ and extract the fractional part of $kA$. We denote this as $(kA \bmod 1)$. Then, multiple this value by the table size $m$ and take the floor of the result. This is shown below:

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

- In the multiplication method, the value of m isn't critical so we could choose to be a power of 2 ($m = 2^p$ for some integer $p$) to make the calculation of the hash function easier.



## Random hashing

Suppose a malicious adversary chooses the keys to be hashed by some fixed hash function. Then the adversary can choose $n$ keys in such a way that all keys hash to the same slot, yielding an average run time of $\Theta(n)$. Any static hash function is vulnerable to such terrible worst-case behavior.

The effective way to improve the situation is to choose the hash function randomly in such away that is independent of the keys that are going to be actually stored.. This approach is call random hashing.

A special case of this approach. called universal hashing, can yield provably good performance on average when collisions are handled by chaining, no matter what keys the adversary chooses.

**Recommended Exercises**: 11.3-1, 11.3-2.

## Open Addressing

- Another collision resolution technique.
- Each entry in the table $T$ can store at most one element, rather than a linked list of elements in chaining. Thus, $\alpha \leq 1$. The size of the table is $m$.
- To perform insertion, we successively examine, or probe, the hash table until we find an empty entry.
- The probe sequence (the sequence of entries examined) depends on the probing techniques used. There are several types of probing techniques. We will discuss two of them later in this section.

**Pseudocode:**

To search for a given element, we search for its key by hashing the key and then using the probe function to generate successive slots to probe until we find the element or have probed the entire table since the element is not in the table.

```
Hash-Search(T, k)
// T[0:m-1] is the hash table
// k is the key value to search for
1. i = 0
2. repeat
3        probe = h(k, i)
4.       if T[probe] == k
5.               return probe
6.       i = i + 1
7. until T[probe] == NIL or i == m
8. return NIL
```

To insert a given element, we hash the key to find a slot to probe. If the slot is occupied, we use the probe function to generate successive slots to probe until we find an open slot or find that the table is full.

```
Hash-Insert(T, k)
// T[0:m-1] is the hash table
// k is the key value to insert
1. i = 0
2. repeat
3.      probe = h(k, i)
3.      if T[probe] == NIL
4.           T[probe] = k
5.           return probe
6.      else i = i + 1
7. until i == m
8. error "hash table overflow"
```

**Issues with deletion**: We can mark a deleted spot with a special tag DELETED. The insert would treat it as an empty slot so it can insert a new key here. The search passes over slots with the DELETED tag. If there are several slots with DELETED tags, it will increase the search time and it will no longer depend on the load factor. For this reason, we prefer to use hashing by chaining when keys must be deleted.

**Linear Probing:** The probe sequence is:

$$h(k,i) = (h_1(k) + i) \bmod m, \ for\ i = 0, 1, \ldots, m-1$$

That is, $T[h_1(k)],\ T[h_1(k)+1], \ldots, T[m-1],\ T[0],\ T[1], \ldots, T[h_1(k)-1]$.

This technique suffers on the **primary clustering** problem: long runs of occupied entries.

- Why does linear probing usually results in primary clustering?
- What's wrong with the primary clustering?
- **Answer**: An empty slot preceded by $i$ full slots gets filled next with probability $(i+1)/m$. So long runs tend to get longer, thus decreasing the average search time.

**Double Hashing:** The probe sequence is

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m, \ for\ i = 0, 1, \ldots, m-1$$

- The probe sequence depends in two ways upon the key $k$. In this case, if two keys have the same initial probe position, they may not have the same probe sequence. Thus, double hashing does not suffer from clustering.

- In order to fully utilize the table, given any $k$, $h_2(k)$ **must be always relatively prime to** $m$.

  If $d = \gcd\{h_2(k), m\}$, then only $1/d$th of table will be searched.

  Two different approaches to make $h_2(k)$ always relatively prime to $m$.

  1. Let $m = 2^p$, where $p$ is some positive integers.
     Design $h_2$ so that it always produces an odd number.
  2. Let $m$ be a prime number.
     Design $h_2$ so that it always produces a positive integer less than $m$.
     Example: Let $m$ be a prime and let

$$\begin{cases} h_1(k) = k \bmod m \\ h_2(k) = 1 + (k \bmod m'), \text{where } m' = m-1 \text{ or } m-2 \end{cases}$$

**Recommended Exercise**: 11.4-1, 11.4-2.

The following theorems provide analysis of chaining and open addressing. We will not do the proofs in this course but it is useful to see what the theorems state.

**Theorem 11.1** *For hashing by chaining, an unsuccessful search takes time $\Theta(1+\alpha)$ in average, under the simple uniform hashing assumption.*

**Theorem 11.2** *For hashing by chaining, a successful search takes time $\Theta(1+\alpha)$ in average, under the simple uniform hashing assumption.*

**Theorem 11.6** *For open-address hashing, the expected # of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.*

**Theorem 11.8** *For open-address hashing, the expected # of probes in a successful search is at most $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$, assuming uniform hashing.*

## Hash tables in practice

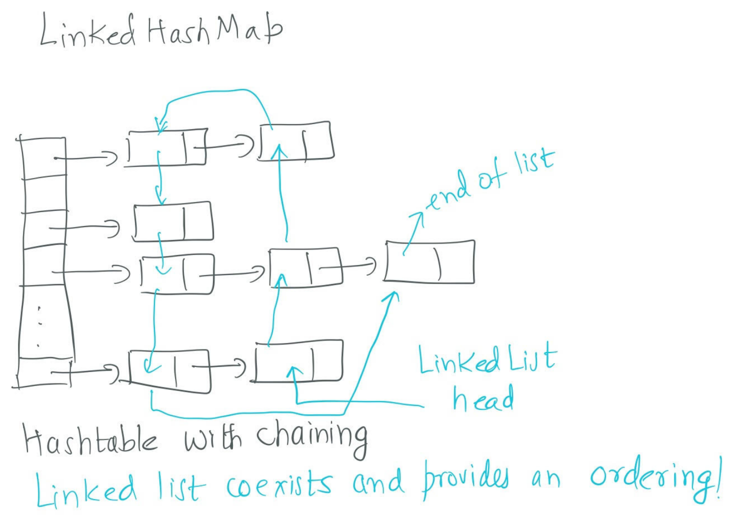
- Java provides `HashMap` (preferred and faster), and `Hashtable`.

```
// key --> String, value --> Integer
HashMap<String, Integer> map  = new HashMap<String, Integer>();

// then we can call put, get, remove etc
map.put("CS321", 1);
map.put("CS488", 2);
```

  See the hashmaps folder in the class resources repo for more examples.

- Java also provides the related `LinkedHashMap` that combines a hash table with a linked list! So we can have fast access but still be able to order the elements (like in a linked list). See the picture below for how a `LinkedhashMap` works.



- **Coding Exercise 1**: Modify the cache in Project 1 to use a `LinkedHashMap` instead of a `LinkedList`. This will make our cache faster, especially for larger cache sizes. This would be essential if we want our cache class to be usable across a large variety of applications. We will reuse the `Cache` in the last project in the course, so this is desirable to get the best speed, although not essential (as our cache sizes aren't very large in the last project).
- **Coding Exercise 2**: Modify your `MaxHeap` class to contain a `HashMap` to be able to find the index in the heap array for a given key. So your `MaxHeap` will contain a heap array and an auxiliary `HashMap` inside it!