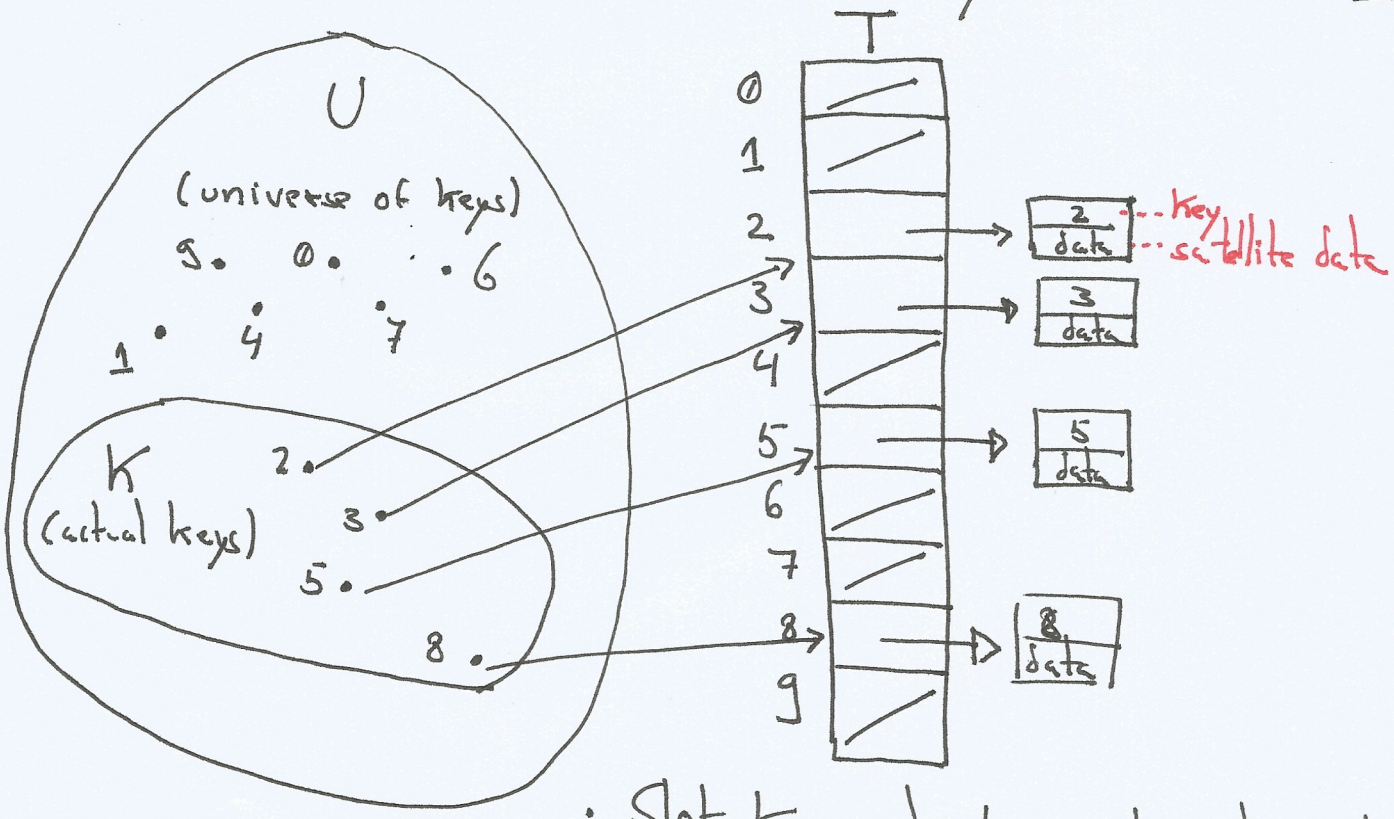
Chapter 11 - Hash Tables [cormen2007]

- Many applications require a dynamic set that supports only the dictionary ~~applications~~ operations INSERT, SEARCH and DELETE
- A hash table is an effective data structure for implementing dictionaries
- Although searching for an element in a hash table can take as long as searching for an element in a linked list ($O(n)$ time) in practice, hashing performs extremely well.
- Under reasonable assumptions, the expected time to search for an element in a hash table is $O(1)$.
- Hash table is a generalization of an ordinary array. Directly addressing into an ordinary array makes effective use of our ability to examine an arbitrary position in an array in $O(1)$ time
- Direct addressing is applicable when we can afford to allocate an array that has one position for every possible key
- When the number of keys actually stored is small relative to the total number of keys, hash tables become an effective alternative to directly addressing an array
  - This happens because a hash table typically uses an array of size proportional to the number of keys actually stored

11.1 Direct Address Tables

- Simple technique that works well when the universe U of keys is small.
- Suppose that an application needs a dynamic set in which each element has a key drawn from the universe $U = \{0, 1, \ldots, m-1\}$, where m is not too large. Assume that no two elements have the same key.
- To represent the dynamic set, we use an array, or direct-address table denoted by $T[0 \ldots m-1]$ in which each position corresponds to a key in the universe U.



- Slot k points to an element in the set with key k
- If the set contains no element with key key k, then $T[k] = NIL$
- $O(1)$ time required

11.2 Hash Tables

- Disadvantage of direct addressing:

  If U is large then it is impractical/impossible to store a table T of size $|U|$.

- Furthermore:

  Set K of keys actually stored may be so small relative to U that most of the space allocated for T would be wasted!

- When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct address table

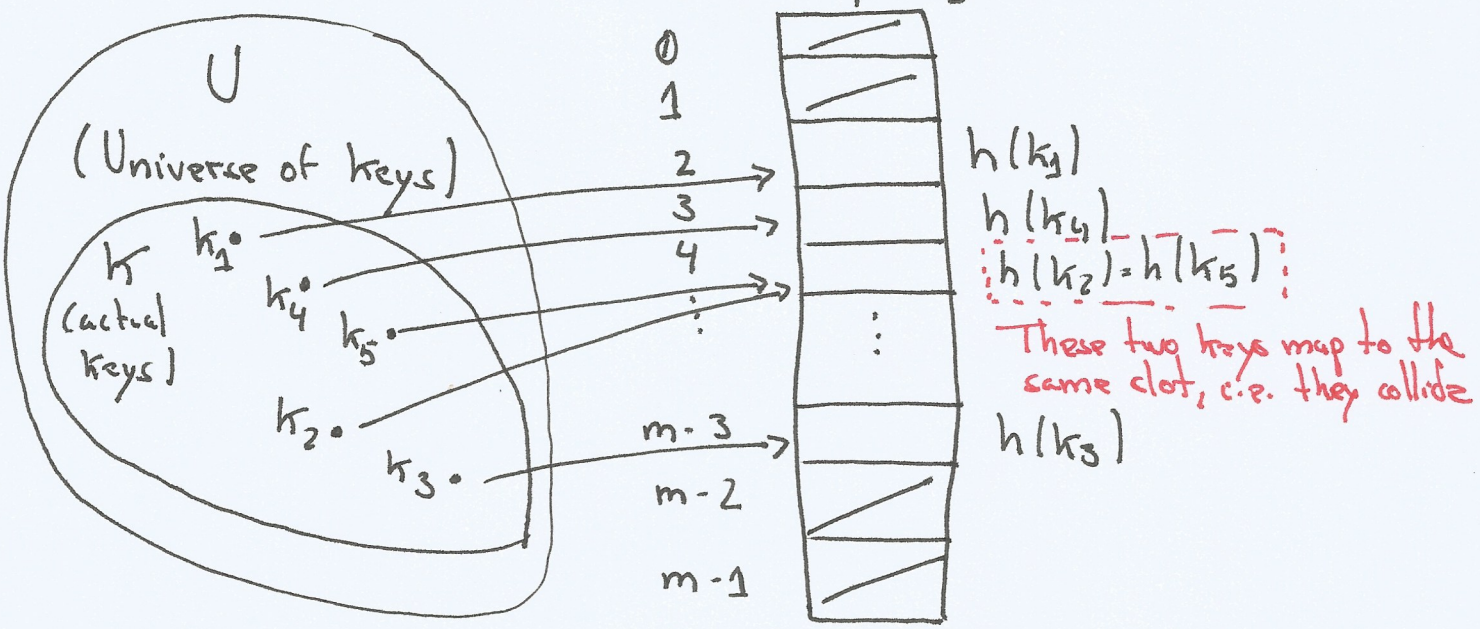
- Specifically: . storage requirements = $\Theta(|K|)$
- Whilst also maintaining $O(1)$ time for searching
- **Question:**

  How is it possible to have the best of both worlds?

  - The $O(1)$ time bound is only valid for the average case/time
  - Whereas for direct-addressing it was true for the worst-case

- With hashing an element is stored is position/index $h(k)$, where h is a hash function that computes the slot from key k.
- I.e. h maps the universe U of keys into the slots of a hash table $T[0 \dots m-1]$:

$$h: U \longrightarrow \{0, 1, \dots, m-1\}$$



- Two keys may hash to the same slot (collision). Since $|U| > m$ avoiding collisions is impossible. A good hash functions tries to minimize the number of collisions.

Question:
So how can we tackle this issue? Any ideas?

11.2.1 Collision resolution by chaining

Simple idea: place all colliding elements in a linked list

- Slot j contains a pointer to the head of the list of all stored elements that hash to j. If there are no such elements, slot j contains NIL.

- Dictionary operations:

  ChainedHashInsert (T, x):

  insert x at the head of the list $T[h(key[x])]$

  ChainedHashSearch (T, k):

  search for an element with key k in list $T[h(k)]$

  ChainedHashDelete (T, x):

  Delete x from the list $T[h(key[x])]$

- Worst-case running time for insertion is $O(1)$
  - Assumes that keys cannot be repeated

- Worst-case running time for searching is proportional to the length of the list

- Deletion takes time $O(1)$ in the worst case if we use a doubly linked list
  - Note: that the function receives element x as input, and note its key k, therefore we do not need to search the list

11.2.2 Analysis of ~~chaining~~ hashing with chaining

Question 1: How well does hashing with chaining performs?

Question 2: How can we analyze this?

- We are interested in determining how long it takes to search for an element with a given key.
  - The other operations we already know that they require $O(1)$ time.
- Given a hash table T with m slots storing n elements we define the load factor α for T as $\frac{n}{m}$
- The load factor represents the average number of elements stored in a chain.
- **Question:** What is the worst-case that you can think of?

⇓

All n-keys mapping to the same slot, creating a list of length n

- The worst-case time for searching is thus $O(n)$ plus the time to compute the hash function

⇓

Terrible performance, clearly hash tables are not used for their worst-case performance.

- Average hashing performance depends on how well the hash function distributes the set of keys to be stored among the m slots

- Assume that any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to. This assumption is called simple uniform hashing

- For $j = 0, 1, \ldots, m-1$ denote the length of the list $T[j]$ by n_j so that $n = n_0 + n_1 + \cdots + n_{m-1}$ and the average value of $n_j = \alpha = n/m$

- Assume that the hash value $h(k)$ can be computed in $O(1)$ time. This way the time required to search for an element with key k depends on $n_{h(k)}$ of the list $T[h(k)]$.

- Lets consider the expected number of elements examined by the search algorithm
  - Number of elements in list $T[h(k)]$ that are checked to see if their keys are equal to k

- We shall consider two cases:

Case 1: The search is unsuccessful: no element in the table has key k

Case 2: The search is successful: the table contains an element with key k

Theorem 11.1

In a hash table in which collisions are solved by chaining, an unsuccessful search takes time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing

Proof: ① Under the assumption of simple uniform hashing, any k not already stored in the table is equally likely to hash to any of the m slots.

② The expected time to search unsuccessfully for a key k is the expected time to search to the end of the list $T[h(k)]$ which has expected length $n_{h(k)} = \alpha$

③ Thus the expected number of elements examined is α and the time is $\Theta(1+\alpha)$ (includes the cost of calculating $h(k)$)

- The situation for a successful search is slightly different: each list is not equally likely to be searched. Instead the probability that a list is searched is proportional to the number of elements it contains.

Theorem 11.2

In a hash table in which collisions are solved by chaining, a successful search takes time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Proof: You can check the book for details, but it involves probability, expectation values, etc... All fun things !!

Question:
What does this mean?

If the number of hash-table slots is at least proportional to the number of elements in the table then we have $n = O(m)$ and consequently

$$\alpha = \frac{n}{m} = \frac{O(m)}{m} =_m O(1)$$

This searching takes constant time on average.

- All dictionary operations can be performed in $O(1)$ time (on average):
  - Insertion: $O(1)$ worst-case
  - Deletion: $O(1)$ worst-case for doubly linked lists
  - Search: $O(1)$ time on average.