


Hash Tables



Adapted from the CLRS book slides

OVERVIEW

Many applications require a dynamic set that supports only the ***dictionary operations*** INSERT, SEARCH, and DELETE. Example: a symbol table in a compiler.

A hash table is effective for implementing a dictionary.

The expected time to search for an element in a hash table is $O(1)$, under some reasonable assumptions.

Worst-case search time is $\Theta(n)$, however.

OVERVIEW (continued)

A hash table is a generalization of an ordinary array.

With an ordinary array, store the element whose key is k in position k of the array.

Given a key k , to find the element whose key is k , just look in the k th position of the array. This is called ***direct addressing***.

Direct addressing is applicable when you can afford to allocate an array with one position for every possible key.

OVERVIEW (continued)

Use a hash table when do not want to (or cannot) allocate an array with one position per possible key.

Use a hash table when the number of keys actually stored is small relative to the number of possible keys.

A hash table is an array, but it typically uses a size proportional to the number of keys to be stored (rather than the number of possible keys).

Given a key k , don't just use k as the index into the array.

Instead, compute a function of k , and use that value to index into the array. We call this function a ***hash function***.

OVERVIEW (continued)

Issues that we'll explore in hash tables:

How to compute hash functions. We'll look at several approaches.

What to do when the hash function maps multiple keys to the same table entry. We'll look at chaining and open addressing.

DIRECT-ADDRESS TABLES

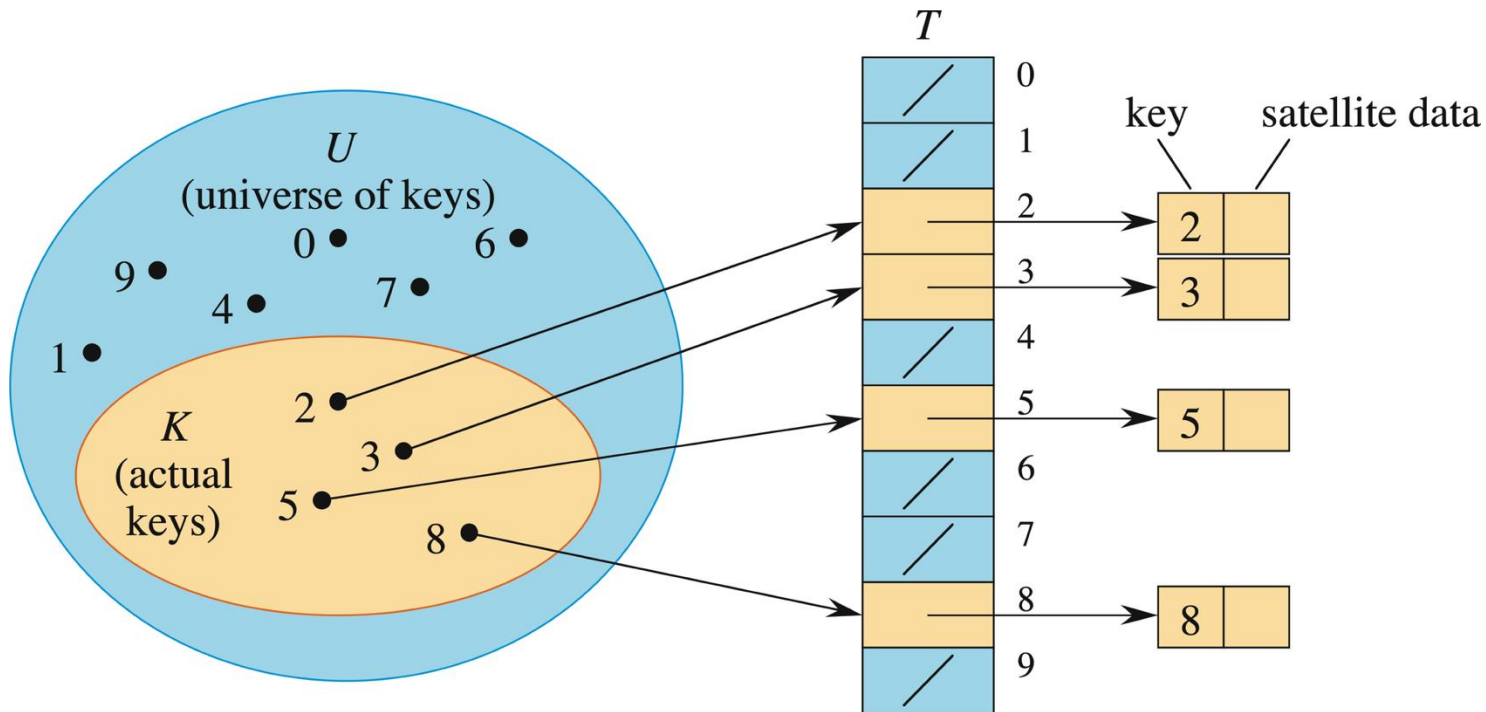
Scenario

- Maintain a dynamic set.
- Each element has a key drawn from a universe $U = \{0, 1, \dots, m - 1\}$ where m isn't too large.
- No two elements have the same key.

Represent by a *direct-address table*, or array, $T[0 \dots m - 1]$:

- Each *slot*, or position, corresponds to a key in U .
- If there's an element x with key k , then $T[k]$ contains a pointer to x .
- Otherwise, $T[k]$ is empty, represented by NIL.

DIRECT-ADDRESS TABLES (continued)



Dictionary operations are trivial and take $O(1)$ time each:

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 $T[x.key] = \text{NIL}$

The problem with direct addressing is if the universe U is large, storing a table of size $|U|$ may be impractical or impossible.



HASH TABLES

Often, the set K of keys actually stored is small, compared to U , so that most of the space allocated for T is wasted.

- When K is much smaller than U , a hash table requires much less space than a direct-address table.
- Can reduce storage requirements to $\Theta(|K|)$.
- Can still get $O(1)$ search time, but in the *average case*, not the *worst case*.

Idea

Instead of storing an element with key k in slot k , use a function h and store the element in slot $h(k)$.

- We call h a **hash function** and T a **hash table**.
- $h : U \rightarrow \{0, 1, \dots, m - 1\}$, so that $h(k)$ is a legal slot number in T .
- We say that k **hashes** to slot $h(k)$.

HASH TABLES (continued)

Collision

When two or more keys hash to the same slot.

- Can happen when there are more possible keys than slots ($|U| > m$).
- For a given set K of keys with $|K| \leq m$, may or may not happen. Definitely happens if $|K| > m$.
- Therefore, must be prepared to handle collisions in all cases.
- Use two methods: chaining and open addressing. We'll examine both.

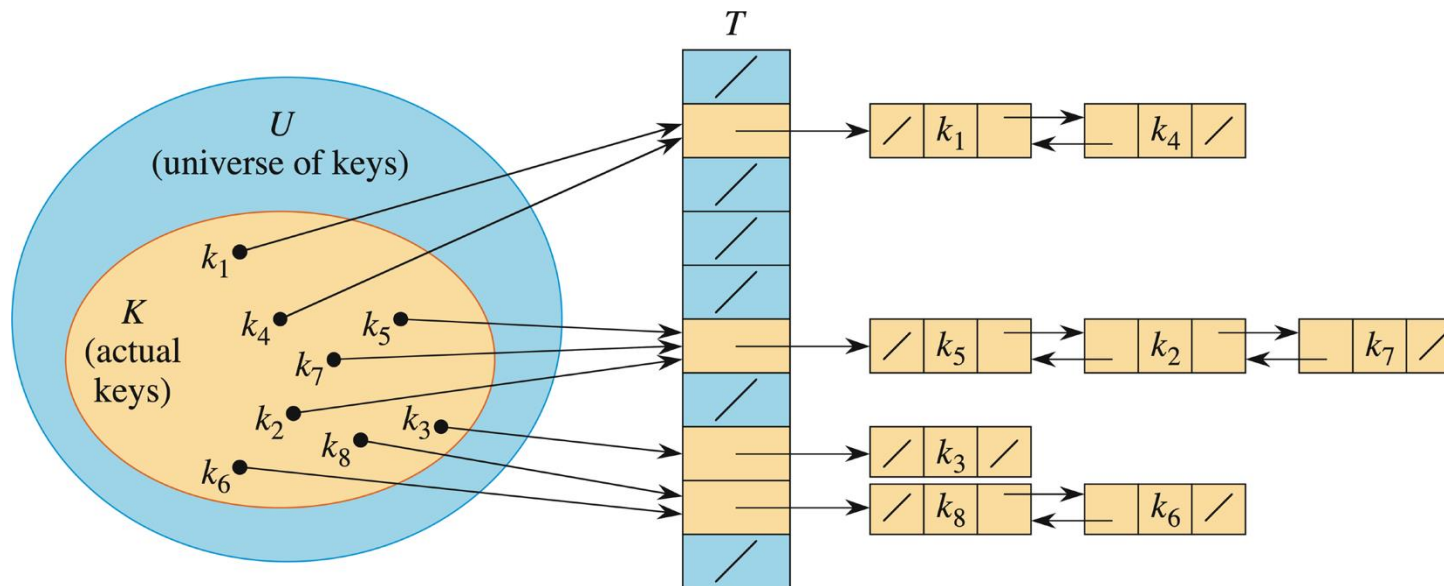
HASH TABLES (continued)

Independent uniform hashing

- Ideally, $h(k)$ would be randomly and independently chosen uniformly from $\{0, 1, \dots, m - 1\}$. Once $h(k)$ is chosen, each subsequent evaluation of $h(k)$ must yield the same result.
- Such an idea hash function is an *independent uniform hash function*, or *random oracle*.
- It's an ideal theoretical abstraction. Cannot be reasonably implemented in practice. Use it to analyze hashing behavior, and find practical approximations to the ideal.

COLLISION RESOLUTION BY CHAINING

Like a nonrecursive type of divide-and-conquer: use the hash function to divide the n elements randomly into m subsets, each with approximately $|n/m|$ elements. Manage each subset independently as a linked list.



COLLISION RESOLUTION BY CHAINING (continued)

Insertion:

- Worst-case running time is $O(1)$.

CHAINED-HASH-INSERT(T, x)

1 LIST-PREPEND($T[h(x.key)], x$)

Search:

- Worst-case running time is proportional to the length of the list of elements in slot $h(k)$.

CHAINED-HASH-SEARCH(T, k)

1 **return** LIST-SEARCH($T[h(k)], k$)

Deletion:

- Given pointer x to the element to delete, so no search is needed to find this element.
- Worst-case running time is $O(1)$ time if the lists are doubly linked.

CHAINED-HASH-DELETE(T, x)

1 LIST-DELETE($T[h(x.key)], x$)

ANALYSIS OF HASHING WITH CHAINING

Given a key, how long does it take to find an element with that key, or to determine that there is no element with that key?

- Analysis is in terms of the *load factor* $\alpha = n/m$:
 - n = # of elements in the table.
 - m = # of slots in the table = # of (possibly empty) linked lists.
 - Load factor is average number of elements per linked list.
 - Can have $\alpha < 1$, $\alpha = 1$, or $\alpha > 1$.
- Worst case is when all n keys hash to the same slot \Rightarrow get a single list of length n \Rightarrow worst-case time to search is $\Theta(n)$, plus time to compute hash function.
- Average case depends on how well the hash function distributes the keys among the slots.

ANALYSIS OF HASHING WITH CHAINING (continued)

Focus on average-case performance of hashing with chaining.

- Assume *independent uniform hashing*: any given element is equally likely to hash into any of the m slots, independent of where any other elements hash to.
- Independent uniform hashing is *universal*: probability that any two distinct keys collide is $1/m$.
- For $j = 0, 1, \dots, m - 1$, denote the length of list $T[j]$ by n_j , so that $n = n_0 + n_1 + \dots + n_{m-1}$.
- Expected value of n_j is $E[n_j] = \alpha = n/m$.
- Assume that the hash function takes $O(1)$ time to compute, so that the time required to search for the element with key k depends on the length $n_{h(k)}$ of the list $T[h(k)]$.

UNSUCCESSFUL SEARCH

Theorem

An unsuccessful search takes average-case time $\Theta(1 + \alpha)$.

Proof Independent uniform hashing \Rightarrow any key not already in the table is equally likely to hash to any of the m slots.

To search unsuccessfully for any key k , need to search to the end of list $T[h(k)]$. This list has expected length $E[n_{h(k)}] = \alpha$. Therefore, the expected number of elements examined in an unsuccessful search is α .

Adding in the time to compute the hash function, the total time required is $\Theta(1 + \alpha)$.

SUCCESSFUL SEARCH

- The circumstances are slightly different from an unsuccessful search.
- The probability that each list is searched is proportional to the number of elements it contains.

Theorem

A successful search takes expected time $\Theta(1 + \alpha)$.

SEARCH ANALYSIS

If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$, which means that searching takes constant time on average.

Since insertion takes $O(1)$ worst-case time and deletion takes $O(1)$ worst-case time when the lists are doubly linked, all dictionary operations take $O(1)$ time on average.