Algorithms
Chapter 11 Hash Tables

Assistant Professor: Ching-Chi Lin
林清池 助理教授
chingchi.lin@gmail.com

Department of Computer Science and Engineering
National Taiwan Ocean University

Outline

- **Direct-address tables**
- Hash tables
- Hash functions
- Open addressing

Overview$_{1/3}$

- 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.
- A hash table is a generalization of an ordinary array.
  - With an ordinary array, we store the element whose key is k in position k of the 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**.

Overview$_{2/3}$

- We use a hash table when we do not want to (or can't) 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**.

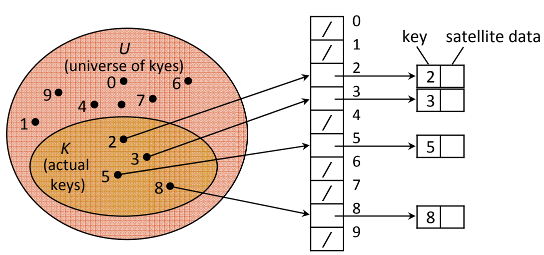
Overview$_{3/3}$

- Issues that we'll explore in hash tables:
  - How to compute hash functions?
    - The multiplication methods.
    - The division methods.
  - What to do when the hash function maps multiple keys to the same table entry?
    - Chaining.
    - Open addressing.

Direct-address tables$_{1/2}$

- Scenario:
  - Maintain a dynamic set.
  - Each element has a key drawn from a universe $U = \{0, 1, \ldots, m-1\}$ where m isn't too large.
  - No two elements have the same key.
- Represent by a **direct-address table**, or array, $T[0..m-1]$:
  - 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$_{2/2}$



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

DIRECT-ADDRESS-SEARCH(T, k)
return $T[k]$

DIRECT-ADDRESS-DELETE(T, x)
$T[key[x]] \leftarrow$ NIL

DIRECT-ADDRESS-INSERT(T, x)
$T[key[x]] \leftarrow x$

Outline

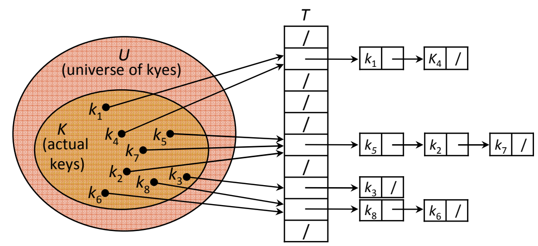
- Direct-address tables
- **Hash tables**
- Hash functions
- Open addressing

Hash tables$_{1/2}$

- **Problem:**
  - If the universe U is large, storing a table of size $|U|$ may be impractical or impossible.
  - The set K of keys actually stored is small, compared to U, so that most of the space allocated for T is wasted.
- **Solution:** Hash tables
  - When K is much smaller than U, a hash table requires much less space than a direct-address table.
  - Storage requirements can be reduced to $\Theta(|K|)$.
  - Searching for an element requires $O(1)$ time, but in the **average case**, not the **worst case**.

Hash tables$_{2/2}$

- **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**.
  - $h : U \rightarrow \{0, 1, . . . , m - 1\}$, so that $h(k)$ is a legal slot number in T.
  - We say that k **hashes** to slot $h(k)$.
  - We also say that $h(k)$ is the **hash value** of key k.



Collisions

- **Collisions:** When two or more keys hash to the same slot.
  - Can happen when there are more possible keys than slots ($|U| > m$).
  - Methods to resolve the collision problem.
    - **Chaining**
    - **Open addressing**
  - Chaining is usually better than open addressing.
- **Collision resolution by chaining**
  - Put all elements that hash to the same slot into 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$_{1/2}$

- How to implement dictionary operations with chaining:
  - CHAINED-HASH-**INSERT**(T,x):

    Insert x at the head of list $T[h(key[x])]$
    - Worst-case running time is $O(1)$.
    - Assumes that the element being inserted isn't already in the list.
    - It would take an additional search to check if it was already inserted.
  - CHAINED-HASH-**SEARCH**(T,k):

    Search for an element with key k in list $T[h(k)]$
    - Running time is proportional to the length of the list of elements in slot $h(k)$.

Dictionary Operations$_{2/2}$

- CHAINED-HASH-**DELETE**(T,x):

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

  - 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.
  - If the lists are singly linked, then deletion takes as long as searching, because we must find x's predecessor in its list.

Analysis of hashing with chaining

- Given a key, how long does it take to find an 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
  - get a single list of length n
  - 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.

Average-case performance

- Assume **simple uniform hashing**: any given element is equally likely to hash into any of the m slots.
- For $j = 0, 1, \ldots, m-1$, denote the length of the list $T[j]$ by n_j, so that $n = n_0 + n_1 + \ldots + n_{m-1}$.
- Average value of n_j is $E[n_j] = \alpha = n/m$.
- Assume that the hash value $h(k)$ can be computed in $O(1)$ time.
  - Time for the element with key k depends on the length $n_{h(k)}$ of the list $T[h(k)]$.
- We consider two cases:
  - contains no element with key k → unsuccessful.
  - contain an element with key k →successful.

Theorem 11.1

- An **unsuccessful search** takes expected time $\Theta(1+\alpha)$.
- Proof:
  - Under the assumption of simple uniform hashing, 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 the 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 $\Theta(1 + \alpha)$.

Theorem 11.2

- An **successful search** takes expected time $\Theta(1+\alpha)$.
- Proof:
  - Assume the element being searched for is equally likely to be any of the n elements in the table T.
  - During a successful search for x, the # of elements examined = # of elements in the list before x + 1.
  - The expected length of that list is $(n-i)/m$.
  - The expected # of elements examined in a successful search is

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{n-i}{m}\right)=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)=1+\frac{1}{nm}\left(\frac{n(n-1)}{2}\right)=1+\frac{\alpha}{2}-\frac{\alpha}{2n}.$$

  - The total time is $\Theta(2+\alpha/2-\alpha/2n)=\Theta(1+\alpha)$.

Outline

- Direct-address tables
- Hash tables
- **Hash functions**
- Open addressing

What makes a good hash function?

- Ideally, the hash function satisfies the assumption of simple uniform hashing.
- In practice, it's not possible.
  - We don't know in advance the probability distribution.
  - The keys may not be drawn independently.
- Often use heuristics, based on the domain of the keys, to create a hash function that performs well.

Interpreting keys as natural numbers

- Most hash functions assume that the universe of keys are natural numbers.
- Thus, if the keys are not natural numbers, a way is found to interpret them as natural numbers.
- **Example:** Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS.
  - ASCII values: C = 67, L = 76, R = 82, S = 83.
  - There are 128 basic ASCII values.
  - So interpret CLRS as $(67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0) = 141{,}764{,}947$.

Division method

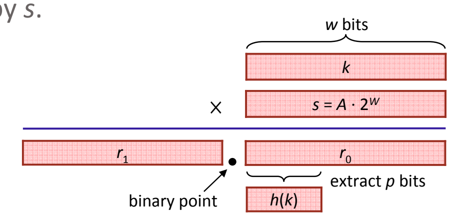
- **Method:** $h(k) = k \bmod m$.
- **Example:** $m = 20$ and $k = 91$ ➔ $h(k) = 11$.
- **Advantage:** Fast, since requires just one division operation.
- **Disadvantage:** Have to avoid certain values of m:
  - Powers of 2 are bad. If $m = 2^p$ for integer p, then $h(k)$ is just the least significant p bits of k.
  - If k is a character string interpreted in radix 2^p (as in CLRS example), then $m = 2^p - 1$ is bad: permuting characters in a string does not change its hash value. (Exercise 11.3-3).
- **Good choice:**
  - A prime not too close to an exact power of 2.

The multiplication method$_{1/4}$

- **Method:**
  - Choose constant A in the range $0 < A < 1$.
  - Multiply key k by A.
  - Extract the fractional part of kA.
  - Multiply the fractional part by m.
  - Take the floor of the result.
- In short, the hash function is $h(k) = \lfloor m\,(kA \bmod 1) \rfloor$, where $kA \bmod 1 = kA - \lfloor kA \rfloor$ = fractional part of kA.
- **Advantage:** Slower than division method.
- **Disadvantage:** Value of m is not critical.

The multiplication method$_{2/4}$

- **Easy implementation:**
  - Choose $m = 2^p$ for some integer p.
  - Let the word size of the machine be w bits.
  - Assume that k fits into a single word. (k takes w bits.)
  - Let s be an integer in the range $0 < s < 2^w$.
  - Restrict A to be of the form $s/2^w$.
  - Multiply k by s.



The multiplication method$_{3/4}$

- The result is $2w$ bits, $r_1 2^w + r_0$, where r_1 is the high-order word of the product and r_0 is the low-order word.
- r_1 holds the integer part of kA ($\lfloor kA \rfloor$). r_0 holds the fractional part of kA ($k\,A \bmod 1 = kA - \lfloor kA \rfloor$).
- The p most significant bits of r_0 holds the value $\lfloor m(kA \bmod 1) \rfloor$.

- **Example:** $m = 8$ (implies $p = 3$), $w = 5$, $k = 21$.
  Must have $0 < s < 2^5$; choose $s = 13$, so $A = 13/32$.
  - **Formula:** $h(k)$: $kA = 21 \cdot 13/32 = 273/32 = 8\,\frac{17}{32}$
    ➔ $kA \bmod 1 = 17/32 \Rightarrow m(kA \bmod 1) = 8 \cdot 17/32 = 17/4 = 4\,\frac{1}{4}$
    ➔ $\lfloor m(kA \bmod 1) \rfloor = 4$, so that $h(k) = 4$.

The multiplication method$_{4/4}$

- **Easy implementation:** $ks = 21 \cdot 13 = 273 = 8 \cdot 2^5 + 17$
  ➔ $r_1 = 8$, $r_0 = 17$. Written in $w = 5$ bits, $r_0 = 10001$.
  Take the $p = 3$ most significant bits of r_0, get 100 in binary, or 4 in decimal, so that $h(k) = 4$.
- **How to choose A:**
  - The multiplication method works with any legal value of A.
  - But it works better with some values than with others, depending on the keys being hashed.
  - Knuth suggests using $A \approx (\sqrt{5} - 1)/2$.

Outline

- Direct-address tables
- Hash tables
- Hash functions
- **Open addressing**

Open addressing

- An alternative to chaining for handling collisions.
- **Idea:**
  - Store all elements in the hash table itself.
  - When searching, we examine table slots until the desired element is found or it is clear that the element is not in the table.
  - We **compute** the sequence of slots to be examined.
- **Advantage:**
  - Avoid pointers.
  - Has a larger number of slots for the same amount of memory.
- **Disadvantage:**
  - Deletion is difficult, thus chaining is more common if keys must be deleted.

Insertion & Searching

- To perform insertion, we successively examine, or **probe**, the hash table until we find an empty slot.
- The sequence of positions probed **depends upon the key being inserted**.
- The hash function is $h : U \times \{0,1,\ldots, m-1\} \rightarrow \{0,1,\ldots, m-1\}$.
- The **probe sequence** is $h(k,0), h(k, 1), \ldots, h(k,m-1)$.

```
HASH-INSERT(T, k)
    i ← 0
    repeat j ← h(k, i)
            if T[j] = NIL
               then T[j] ← k
                      return j
               else i ← i + 1
    until i = m
    error "hash table overflow"
```

```
HASH-SEARCH(T, k)
    i ← 0
    repeat j ← h(k,i)
            if T[j] = k
               then return j
            i ← i + 1
    until T[j] = NIL or i = m
    return NIL
```

Deletion

- When we delete a key from slot i, we can't simply mark that slot as empty by storing NIL in it.
- **Solution:** Use a special value DELETED instead of NIL when marking a slot as empty during deletion.
  - Search should treat **DELETED** as though the slot holds a key that does not match the one being searched for.
  - Insertion should treat **DELETED** as though the slot were empty, so that it can be reused.

Three probing methods

- The ideal situation is **uniform hashing**: each key is equally likely to have any of the $m!$ permutations of $<0, 1, \ldots, m-1>$ as its probe sequence.
- Three commonly used probing methods:
  - Linear probing
  - Quadratic probing
  - Double hashing
- None of these techniques fulfills the assumption of uniform hashing.

Linear probing

- Given an ordinary hash function $h' : U \rightarrow \{0, 1, \ldots, m-1\}$, which we refer to as an **auxiliary hash function**, the method of **linear probing** uses the hash function

$$h(k, i) = (h'(k) + i) \bmod m$$

for $i = 0, 1, \ldots, m - 1$.

- Because the initial probe determines the entire probe sequence, there are only m distinct probe sequences.
- Linear probing suffers from **primary clustering**: long runs of occupied sequences build up.

Quadratic probing

- **Quadratic probing** uses a hash function of the form

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

where h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants, and $i = 0, 1, \ldots, m-1$.

- This method works much better than linear probing, but to make full use of the hash table, the values of c_1, c_2, and m are constrained.
- If two keys have the same initial probe position, then their probe sequences are the same. This property leads **secondary clustering.**
- Because the initial probe determines the entire probe sequence, there are only m distinct probe sequences.

Double Hashing

- **Double hashing** uses a hash function of the form

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

where h_1 and h_2 are auxiliary hash functions.

- The value $h_2(k)$ must be relatively prime to the hash-table size m for the entire hash table to be searched.
  - Let m be a power of 2 and to design h_2 so that it always produce an odd number.
  - Let m be prime and have $1 < h_2(k) < m$.
- $\Theta(m^2)$ different probe sequences, since each possible combination of $h_1(k)$ and $h_2(k)$ gives a different probe sequence.

An example for double Hashing

- Hash table size: 13.
- $h_1(k) = k \bmod 13$; $h_2(k) = 1 + (k \bmod 11)$
- Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$,

$$h(14, i) = (h_1(k) + ih_2(k)) \bmod m$$
$$= (1 + i(1 + 3)) \bmod 13$$
$$= (1 + 4i) \bmod 13.$$

- So, the key 14 is inserted into empty slot 9.