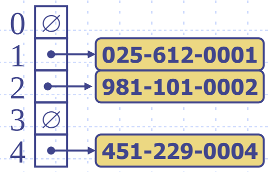
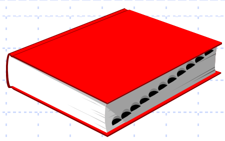
Dictionaries and Hash Tables

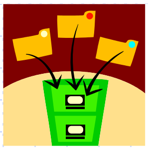


Dictionary ADT (§2.5.1)



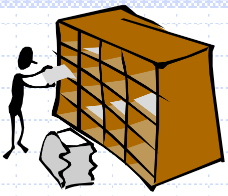
- The dictionary ADT models a searchable collection of key-element items
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key are allowed
- Applications:
  - address book
  - credit card authorization
  - mapping host names (e.g., cs16.net) to internet addresses (e.g., 128.148.34.101)
- Dictionary ADT methods:
  - findElement(k): if the dictionary has an item with key k, returns its element, else, returns the special element NO_SUCH_KEY
  - insertItem(k, o): inserts item (k, o) into the dictionary
  - removeElement(k): if the dictionary has an item with key k, removes it from the dictionary and returns its element, else returns the special element NO_SUCH_KEY
  - size(), isEmpty()
  - keys(), elements()

Log File (§2.5.1)



- A log file is a dictionary implemented by means of an unsorted sequence
  - We store the items of the dictionary in a sequence (based on a doubly-linked lists or a circular array), in arbitrary order
- Performance:
  - insertItem takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - findElement and removeElement take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

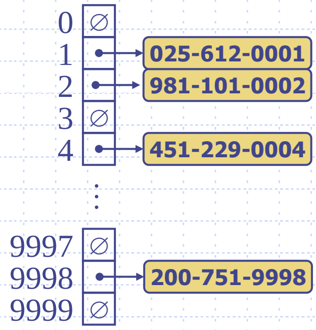
Hash Functions and Hash Tables (§2.5.2)



- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:
  $h(x) = x \bmod N$
  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a dictionary with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Example

- We design a hash table for a dictionary storing items (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x)$ = last four digits of x



Hash Functions (§ 2.5.3)

- A hash function is usually specified as the composition of two functions:

  Hash code map:
  h_1: keys $\rightarrow$ integers

  Compression map:
  h_2: integers $\rightarrow [0, N - 1]$
- The hash code map is applied first, and the compression map is applied next on the result, i.e.,
  $$h(x) = h_2(h_1(x))$$
- The goal of the hash function is to "disperse" the keys in an apparently random way

Hash Code Maps (§2.5.3)

- **Memory address:**
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys
- **Integer cast:**
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
- **Component sum:**
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

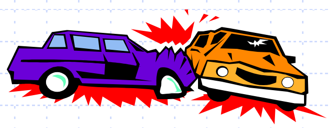
Hash Code Maps (cont.)

- **Polynomial accumulation:**
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
    $$a_0\, a_1 \,\ldots\, a_{n-1}$$
  - We evaluate the polynomial
    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots$$
    $$\ldots + a_{n-1} z^{n-1}$$
    at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)
- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time
    $$p_0(z) = a_{n-1}$$
    $$p_i(z) = a_{n-i-1} + z p_{i-1}(z)$$
    $$(i = 1, 2, \ldots, n-1)$$
- We have $p(z) = p_{n-1}(z)$

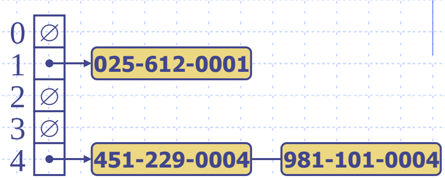
Compression Maps (§2.5.4)

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Collision Handling (§ 2.5.5)

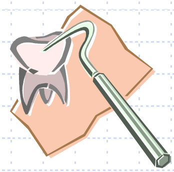


- Collisions occur when different elements are mapped to the same cell



- **Chaining**: let each cell in the table point to a linked list of elements that map there
- Chaining is simple, but requires additional memory outside the table

Linear Probing (§2.5.5)



- Open addressing: the colliding item is placed in a different cell of the table
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

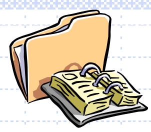
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Search with Linear Probing



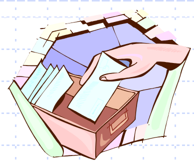
- Consider a hash table A that uses linear probing
- findElement(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm findElement(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return NO_SUCH_KEY
        else if c.key () = k
            return c.element()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until  p = N
    return NO_SUCH_KEY
```

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called ***AVAILABLE***, which replaces deleted elements
- removeElement(k)
  - We search for an item with key k
  - If such an item (k, o) is found, we replace it with the special item ***AVAILABLE*** and we return element o
  - Else, we return ***NO_SUCH_KEY***
- insert Item(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores ***AVAILABLE***, or
    - N cells have been unsuccessfully probed
  - We store item (k, o) in cell i

Double Hashing



- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
  $$(i + jd(k)) \bmod N$$
  for $j = 0, 1, \ldots, N - 1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells
- Common choice of compression map for the secondary hash function:
  $$d_2(k) = q - k \bmod q$$
  where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are
  $$1, 2, \ldots, q$$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

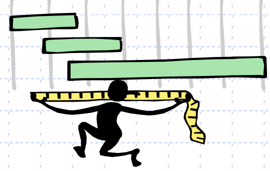
k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

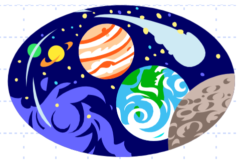
31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Performance of Hashing



- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the dictionary collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is
  $1 / (1 - \alpha)$
- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

Universal Hashing (§ 2.5.6)



- A family of hash functions is **universal** if, for any $0 \le i,j \le M-1$,
  $Pr(h(j)=h(k)) \le 1/N$.
- Choose p as a prime between M and 2M.
- Randomly select $0<a<p$ and $0 \le b<p$, and define $h(k)=(ak+b \bmod p) \bmod N$
- Theorem: The set of all functions, h, as defined here, is universal.

Proof of Universality (Part 1)

- Let $f(k) = ak+b \bmod p$
- Let $g(k) = k \bmod N$
- So $h(k) = g(f(k))$.
- f causes no collisions:
  - Let $f(k) = f(j)$.
  - Suppose $k<j$. Then

$$aj+b-\left\lfloor \frac{aj+b}{p} \right\rfloor p = ak+b-\left\lfloor \frac{ak+b}{p} \right\rfloor p$$

$$a(j-k)=\left(\left\lfloor \frac{aj+b}{p} \right\rfloor - \left\lfloor \frac{ak+b}{p} \right\rfloor\right)p$$

- So $a(j-k)$ is a multiple of p
- But both are less than p
- So $a(j-k) = 0$. I.e., $j=k$. (contradiction)
- Thus, f causes no collisions.

Proof of Universality (Part 2)

- If f causes no collisions, only g can make h cause collisions.
- Fix a number x. Of the p integers y=f(k), different from x, the number such that g(y)=g(x) is at most $\lceil p/N \rceil - 1$
- Since there are p choices for x, the number of h's that will cause a collision between j and k is at most

$$p(\lceil p/N \rceil - 1) \le \frac{p(p-1)}{N}$$

- There are p(p-1) functions h. So probability of collision is at most $\frac{p(p-1)/N}{p(p-1)} = \frac{1}{N}$
- Therefore, the set of possible h functions is universal.