Module 7: Hashing

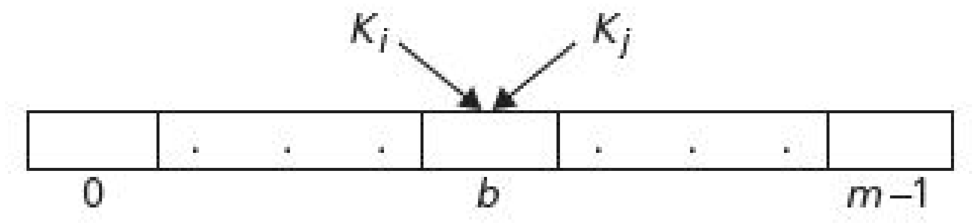
Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Hashing

- A very efficient method for implementing a *dictionary,* i.e., a set with the operations: find, insert and delete
- Based on representation-change and space-for-time tradeoff ideas
- We consider the problem of implementing a dictionary of *n* records with keys K_1, K_2, …, K_n.
- Hashing is based on the idea of distributing keys among a one-dimensional array H[0…m-1] called a hash table.
  - The distribution is done by computing, for each of the keys, the value of some pre-defined function *h* called the ***hash function***.
  - The hash function assigns an integer between 0 and *m*-1, called the hash address to a key.
  - The size of a hash table *m* is typically a prime integer.
- Typical hash functions
  - For non-negative integers as key, a hash function could be h(K)=K mod m;
  - If the keys are letters of some alphabet, the position of the letter in the alphabet (for example, A is at position 1 in alphabet A – Z) could be used as the key for the hash function defined above.
  - If the key is a character string $c_0\ c_1\ \dots\ c_{s-1}$ of characters from an alphabet, then, the hash function could be: $(\sum_{i=0}^{s-1} ord(c_i)) \bmod m$

Collisions and Collision Resolution

If $h(K_1) = h(K_2)$, there is a *collision*



- Good hash functions result in fewer collisions but some collisions should be expected
- In this module, we will look at open hashing that works for arrays of any size, irrespective of the hash function.

Open Hashing

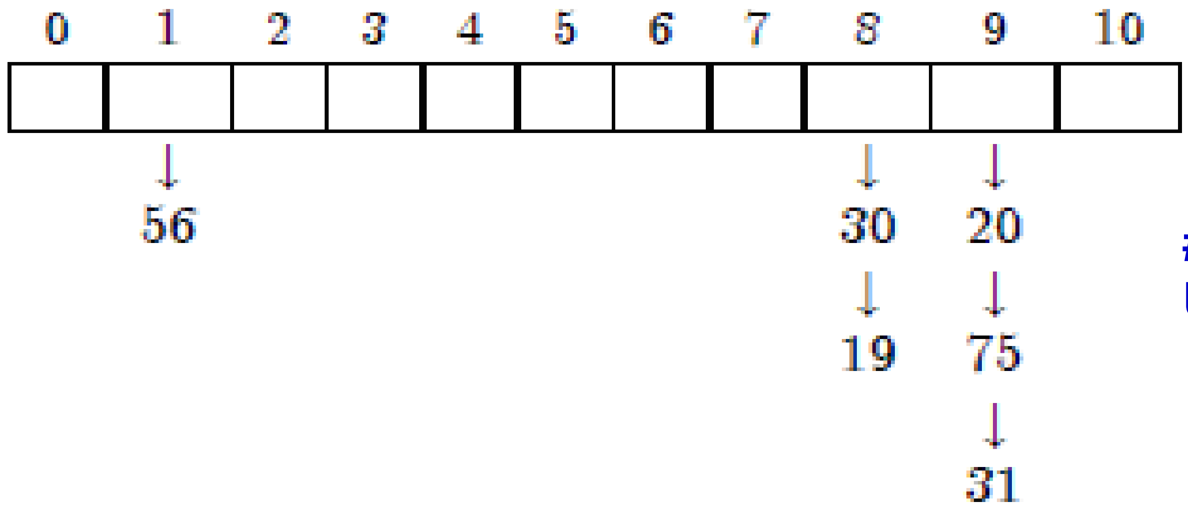
The list of keys: 30, 20, 56, 75, 31, 19

The hash function: $h(K) = K \bmod 11$

The hash addresses:

K	30	20	56	75	31	19
$h(K)$	8	9	1	9	9	8

The open hash table:



# comparisons for an Unsuccessful search is 3.

The largest number of key comparisons in a successful search in this table is 3 (in searching for $K = 31$).

The average number of key comparisons in a successful search in this table, assuming that a search for each of the six keys is equally likely, is

$$\frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 2 = \frac{10}{6} \approx 1.7.$$

Hashing: Space-time Tradeoff

- Hashing is another example for space-time tradeoff.
- A hash table takes space corresponding to the size of the array, but the average time-complexity for a successful search and unsuccessful search are much lower than that of an array.

- In the previous example:
  - In case of an array, the average number of comparisons for a successful search is (n+1)/2 = (6+1)/2 = 3.5; the number of comparisons for an unsuccessful search is 6.

Open Hashing

- Inserting and Deleting from the hash table is of the same complexity as searching.
- If hash function distributes keys uniformly, average length of linked list will be $\alpha = n/m$. This ratio is called *load factor*.
- Average-case number of key comparisons for a successful search is $\alpha/2$; Average-case number of key comparisons for an unsuccessful search is α.
- Worst-case number of key comparisons is $\Theta(n)$ – occurs if we get a linked list containing all the n elements hashing to the same index. To avoid this, we need to be careful in selecting a proper hashing function.
  – Mod-based hashing functions with a prime integer as the divisor are more likely to result in hash values that are evenly distributed across the keys.
- Open hashing still works if the number of keys, $n >$ the size of the hash table, m.

Applications of Hashing (1)

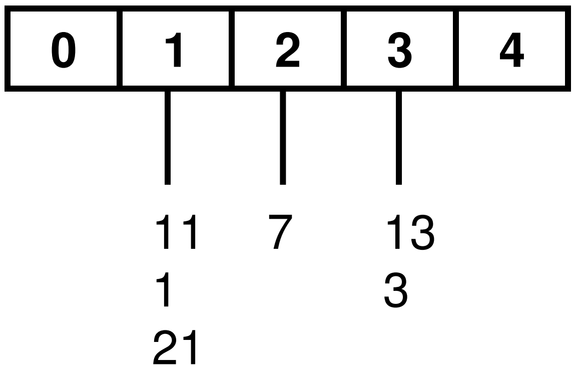
Finding whether an array is a Subset of another array

- Given two arrays A_L (larger array) and A_S (smaller array) of distinct elements, we want to find whether A_S is a subset of A_L.
- Example: $A_L = \{11, 1, 13, 21, 3, 7\}$; $A_S = \{11, 3, 7, 1\}$; A_S is a subset of A_L.
- Solution: Use (open) hashing. Hash the elements of the larger array, and for each element in the smaller array: search if it is in the hash table for the larger array. If even one element in the smaller array is not there in the larger array, we could stop!
- Time-complexity:
  - $\Theta(n)$ to construct the hash table on the larger array of size n, and another $\Theta(n)$ to search the elements of the smaller array.
  - A brute-force approach would have taken $\Theta(n^2)$ time.
- Space-complexity: $\Theta(n)$ with the hash table approach and $\Theta(1)$ with the brute-force approach.
- Note: The above solution could also be used to find whether two sets are disjoint or not. Even if one element in the smaller array is there in the larger array, we could stop!

Applications of Hashing (1)

Finding whether an array is a Subset of another array

- **Example 1**: A_L = {11, 1, 13, 21, 3, 7};
- A_S = {11, 3, 7, 1}; A_S is a subset of A_L.
- Let H(K) = K mod 5.



Hash table approach

# comparisons = 1 (for 11) + 2 (for 3) +
1 (for 7) + 2 (for 1) = 6

Brute-force approach: Pick every element in the smaller array and do a linear search for it in the larger array.

# comparisons = 1 (for 11) + 5 (for 3) +
6 (for 7) + 2 (for 1) = 14

- **Example 2**: A_L = {11, 1, 13, 21, 3, 7};
- A_S = {11, 3, 7, 4}; A_S is NOT a subset of A_L.
- Let H(K) = K mod 5.

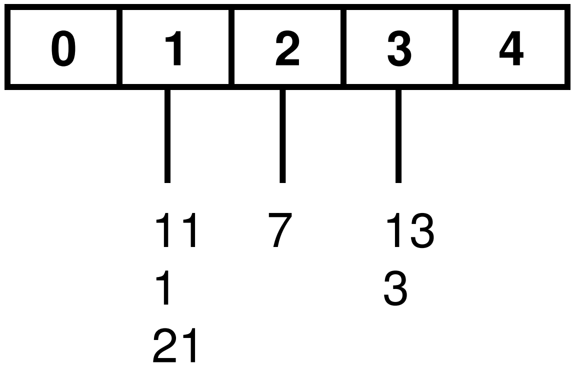
The **hash table approach** would take just 1 (for 11) + 2 (for 3) + 1 (for 7) + 0 (for 4) = 4 comparisons

The **brute-force approach** would take: 1 (for 11) + 5 (for 3) + 6 (for 7) + 6 (for 4) = 18 comparisons.

Applications of Hashing (1)

Finding whether two arrays are disjoint are not

- **Example 1**: A_L = {11, 1, 13, 21, 3, 7};
- A_S = {22, 25, 27, 28}; They are disjoint.
- Let H(K) = K mod 5.



Hash table approach

# comparisons = 1 (for 22) + 0 (for 25) + 1 (for 27) + 3 (for 28) = 5

Brute-force approach: Pick every element in the smaller array and do a linear search for it in the larger array.

# comparisons = 6 comparisons for each element * 4 = 24

- **Example 2**: A_L = {11, 1, 13, 21, 3, 7};
- A_S = {22, 25, 27, 1}; They are NOT disjoint.
- Let H(K) = K mod 5.

The **hash table approach** would take just 1 (for 22) + 0 (for 25) + 1 (for 27) + 2 (for 1) = 4 comparisons

The **brute-force approach** would take: 6 (for 22) + 6 (for 25) + 6 (for 27) + 2 (for 1) = 20 comparisons.

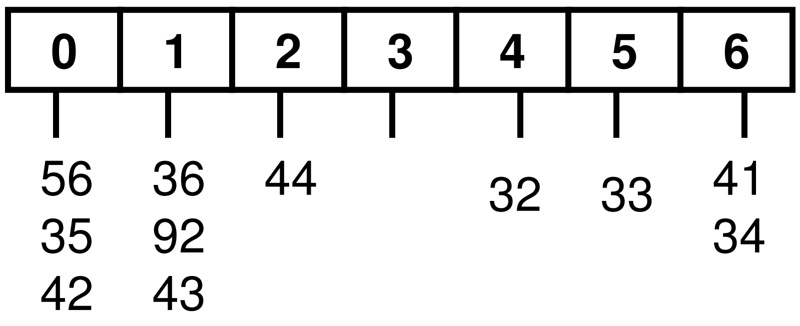
Applications of Hashing (2)

Finding Consecutive Subsequences in an Array

- Given an array A of unique integers, we want to find the contiguous subsequences of length 2 or above as well as the length of the largest subsequence.
- Assume it takes Θ(1) time to insert or search for an element in the hash table.

36 41 56 35 44 33
34 92 43 32 42

H(K) = K mod 7



36	41	56	35	44	33	34	92	43	32	42
35	40	55	34	43	32	33	91	42	31	41
	42	57					93		33	
	43								34	
	44								35	
	45								36	
									37	
	41								**32**	
	42								**33**	
	43								**34**	
	44								**35**	
									36	

Applications of Hashing (1)

Finding Consecutive Subsequences in an Array

- Algorithm

```
Insert the elements of A in a hash table H
Largest Length = 0
for i = 0 to n-1 do
    if (A[i] – 1 is not in H) then
        j = A[i]   // A[i] is the first element of a possible cont. sub seq.
        j = j + 1
        while ( j  is in H) do
            j = j + 1
        end while
        if ( j – A[i] > 1) then  // we have found a cont. sub seq. of length > 1
            Print all integers from A[i] … (j-1)
            if (Largest Length < j – A[i]) then
                Largest Length = j – A[i]
            end if
        end if
    end if
end for
```

L searches in the Hash table H for sub sequences of length L

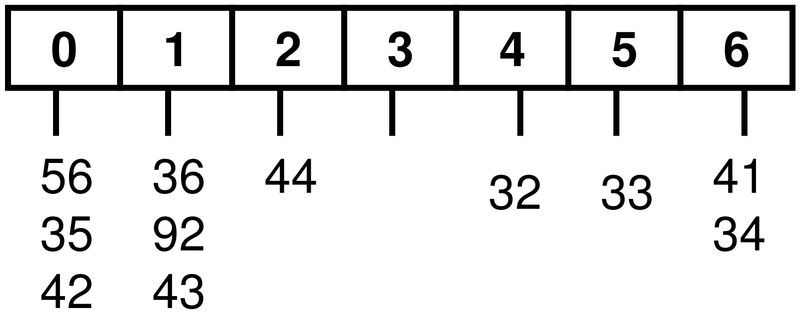
Applications of Hashing (2)

Finding Consecutive Subsequences in an Array

- **Time Complexity Analysis**
- For each element at index i in the array A we do at least one search (for element A[i] – 1) in the hash table.
- For every element that is the first element of a sub seq. of length 1 or above (say length L), we do L searches in the Hash table.
- The sum of all such Ls should be n.
- For an array of size n, we do n + n = 2n = Θ(n) hash searches. The first 'n' corresponds to the sum of all the lengths of the contiguous sub sequences and the second 'n' is the sum of all the 1s (one 1 for each element in the array)

36 41 56 35 44 33
34 92 43 32 42

H(K) = K mod 7



36	**41**	**56**	**35**	**44**	**33**	**34**	**92**	**43**	**32**	**42**
35	**40**	**55**	**34**	**43**	**32**	**33**	**91**	**42**	**31**	**41**
	42	**57**					**93**		**33**	
	43								**34**	
	44								**35**	
	45								**36**	
									37	