COMP250: Hash tables

Lecture 22

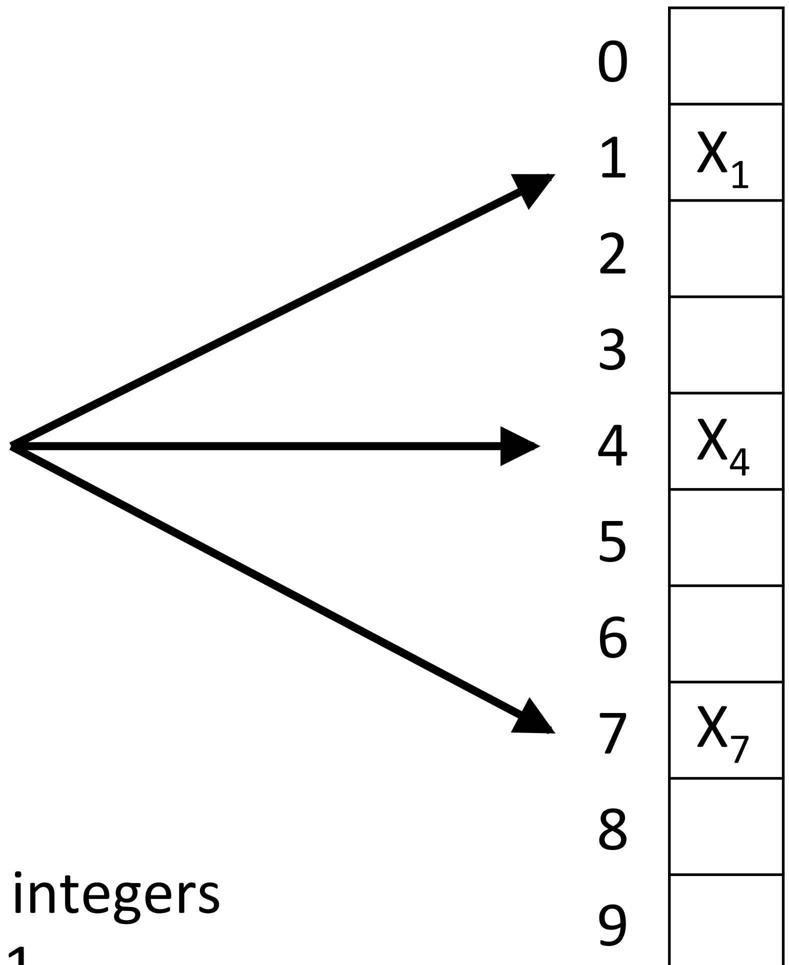
Jérôme Waldispühl
School of Computer Science
McGill University

Dictionary ADT

- Reminder: A dictionary stores pairs (key, information)
- Operations:
  - find(key k)
  - insert(key k, info i)
  - remove(key k)
- Binary Search Trees implement all these operations in time O(h), where h is the height of the tree, **which is O(log n) if we maintain the tree balanced.**
- We can sometimes do better...

Hash tables

remove(1)
insert(4,X_4)
insert(7,X)
insert(1,X_1)



Suppose keys are integers between 0 and K-1.

Hash tables

- Suppose keys are integers between 0 and K-1
- Then, use an array A[0...K-1] containing elements of type "info" to store the dictionary:
  - insert(key k, info i): A[k] = i;
  - remove(key k): A[k] = null;
  - find(key k): return A[k];
- Running time: All operations are O(1)
- It's a miracle! Except that...

Problems with direct array implementation

- If K is large, the array will be very big
  - For McGill student ID, K = 1 000 000 000
- The amount of memory needed (K) is essentially independent of the number of items in the dictionary.
- Idea: compress the array...

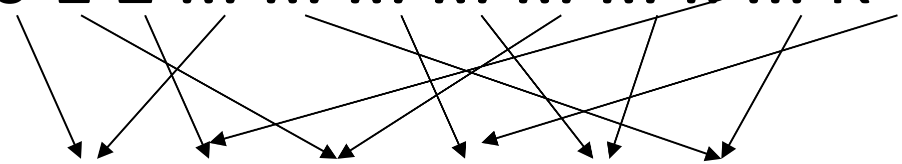
Hash functions

Idea: Map the K possible keys to N integers, with N being much smaller than K

Hash function f: [0...K-1] $\rightarrow$ [0...N-1]

Space of keys: 0 1 2 ... ... K-1

Hash function



Hashed key 0 1 2 ... ... N-1

insert(key k, info i): A[f(k)] = i;

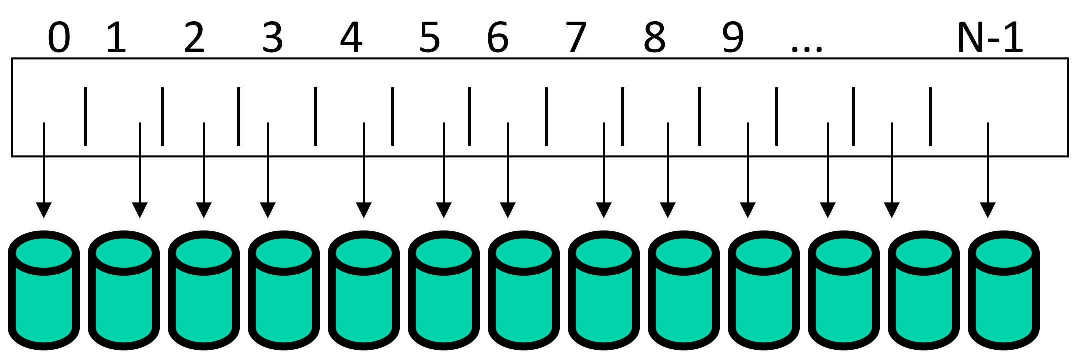
remove(key k): A[f(k)] = null;

find(key k): return A[f(k)];

Collisions

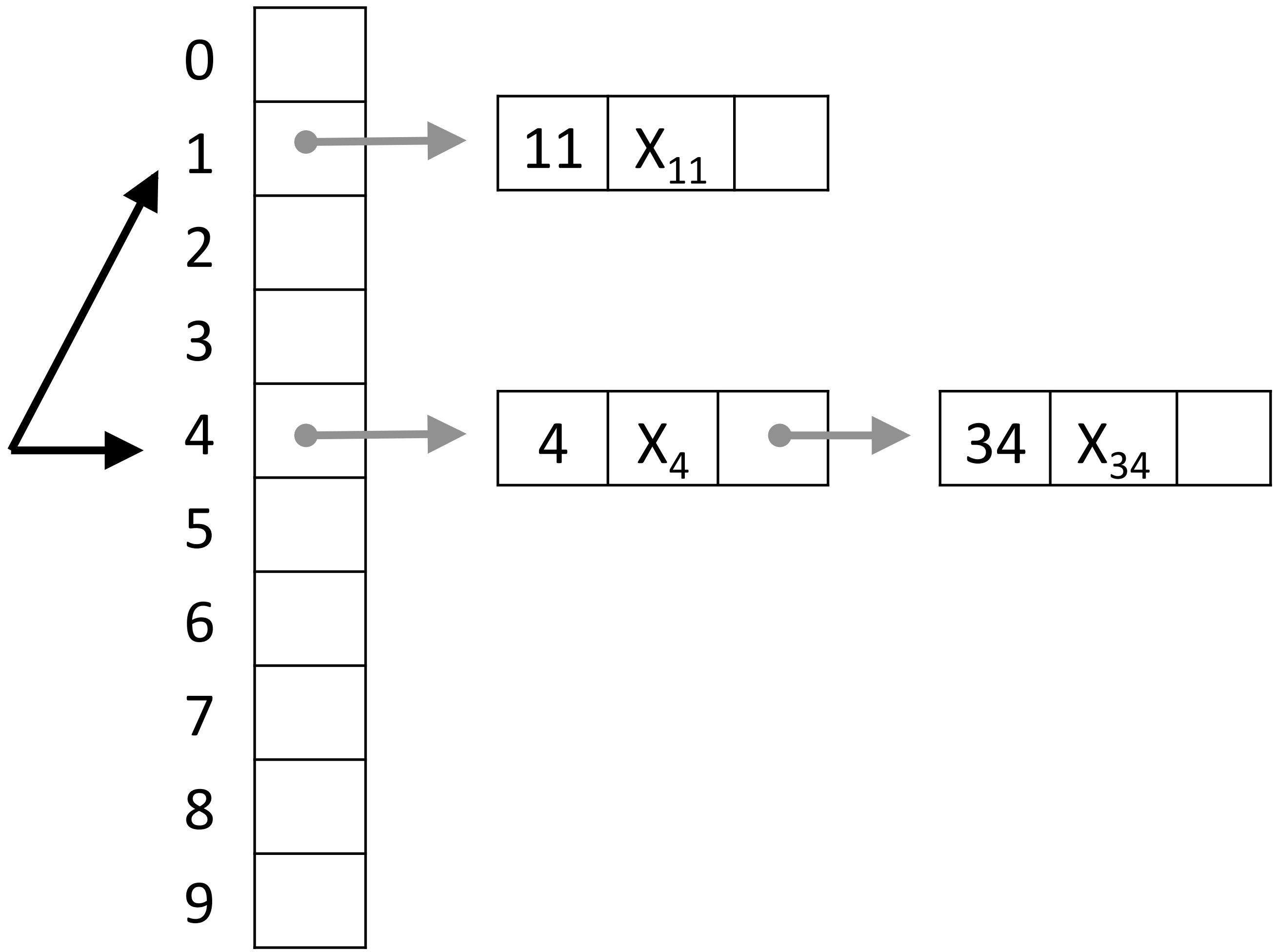
- Collisions! Many keys map to the same index
- Solution: Each element of the array is itself a dictionary (called a bucket), implemented with linked-list, binary search tree, or even a hash table!

Hash table:



Example

remove(4)
insert(4,X_4)
insert(34,X_{34})
insert(11,X_{11})



f(x) = x % 10

Resolving collision with chaining

insert(key k, info i):	A[f(k)].insert(k,i);
remove(key k, info i):	A[f(k)].remove(k);
find(key k):	return A[f(k)].find(k);

Analysis of Hashing with Chaining

Insertion: O(1) time.

Deletion: Search time + O(1) (if we use a double linked list).

Search:

Search time = compute hash function + search the list.

We assume that the time to compute hash function is O(1).

Worst time for searching happens when all keys go the same slot. We need to scan the full list => O(n).

Worst case running time of search to is O(n).

Importance of good hash functions

- Worst case complexity :
  - if all keys end up in the same bucket and we use a linked-list to store buckets??
  - if keys are evenly spread among the N buckets??

- We want a hash function that spreads the keys evenly among the buckets.

Examples of hash functions

Key: k = student ID #

Size of the hash table: N = 100

- f(key k) = $\lfloor k/10\,000\,000 \rfloor$ = first 2 digits
- f(key k) = k mod 100 = last 2 digits
- f(key k) = (sum of digits of k) mod 100

Good hash functions

- Choice of hash function depends on application
- In general, f(k) = k mod N is good choice when N is a prime number
- Example: For student Ids, choose N = 101
  - f(k) = k mod 101
- What if the key is not an integer (e.g. a String)?
  - map key to integer first with some function g(key)
  - use f() to map the integer to [0...N-1]

Hash functions on Strings

We need a function g: String $\rightarrow$ Integers that minimizes collisions

- Linear code:
  g(key k) = sum of ASCII values of each char.
  Problem?
- Polynomial code: Choose a small prime number a
  If key k = $k_0 k_1 k_2 \ldots k_e$, choose
  $g(k) = k_0 + k_1 a + k_2 a^2 + \ldots + k_e a^e$