Lecture 15
Dictionaries, Symbol Tables, and Hashing

EECS 281: Data Structures & Algorithms

Outline

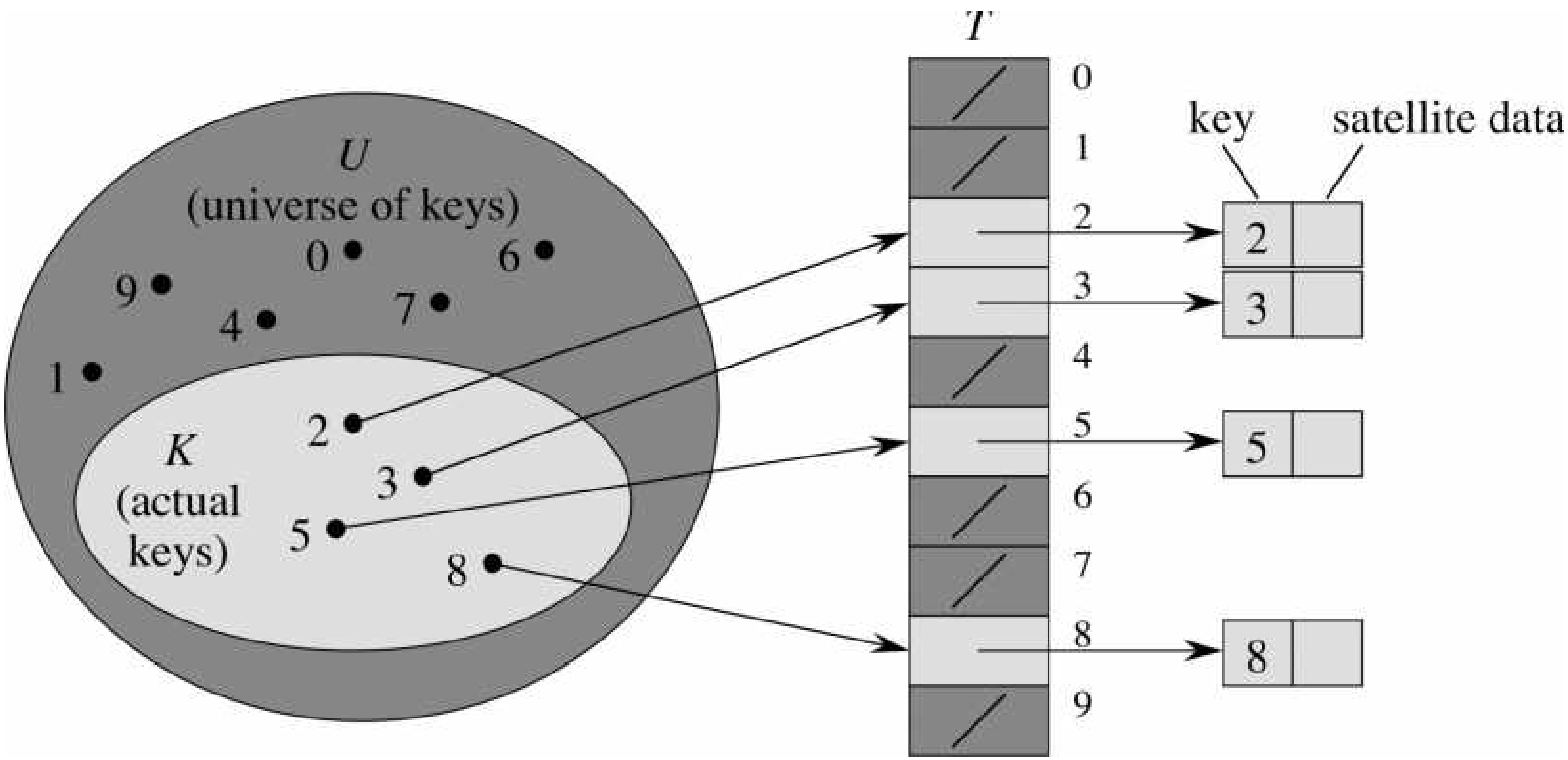
- Dictionary (symbol table) ADT
- Containers with look-up by keys
  - Bucket-based data structures
- Hash Functions
  - Hash Code
  - Compression Map
- Collision Resolution

Search and Insert

- Retrieval of a particular piece of information from large volumes of previously stored data
- Purpose is typically to access information within the item (not just the key)
- Recall that arrays, linked lists are worst-case O(N) for either searching or inserting

Need data structure with optimal efficiency for searching and inserting

Direct Addressing



Dictionary ADT

Definition: an abstract data structure of items with keys that supports two basic operations: *insert* a new item, and *return* an item with a given key

Dictionary ADT

- *Insert* a new item
- *Search* for an item (items) having a given key
- *Remove* a specified item
- *Sort* the symbol table (aka dictionary)
- *Select* the k^{th} largest item in a symbol table
- *Join* two symbol tables

Also may want *construct*, *test if empty*, *destroy*, *copy*...

What if the set of keys is large?

- Example: calendar for 1..N days
  - N could be 365 or 366
  - Can look up a particular day in $O(1)$ time
  - Every day is represented by a bucket, i.e., some container
- If we have a range of integers that fits into memory, everything is easy
  - ***What if we don't?***

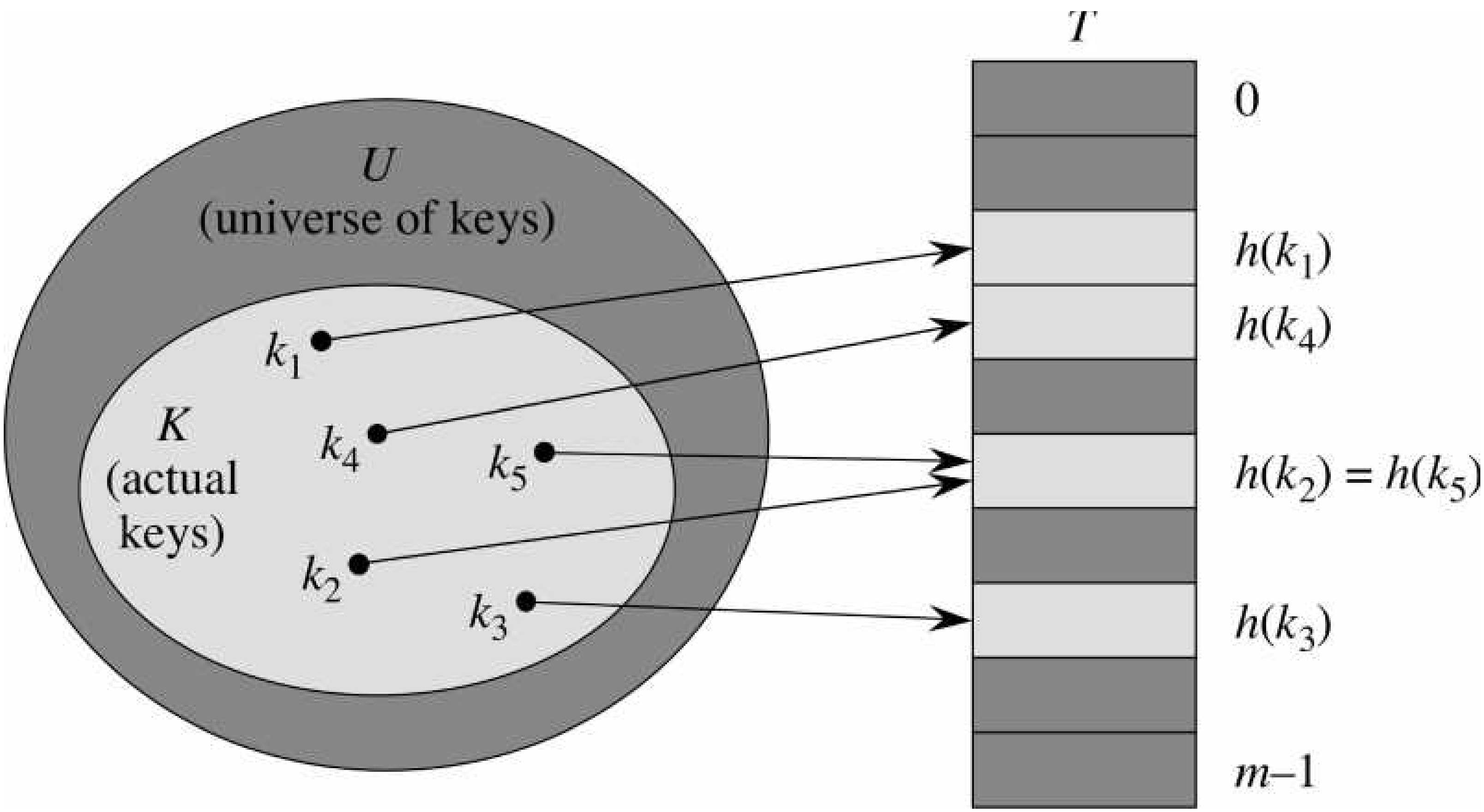
Hashing

- Reference items in a table by keys
  - Use arithmetic operations to transform keys into table addresses (buckets)

Need:

- Hash function: transforms the search key into a table address
- Collision resolution: dealing with search keys that hash to same table address

Hash Tables



T
U
(universe of keys)
K
(actual
keys)
k_1
k_4
k_5
k_2
k_3
0
$h(k_1)$
$h(k_4)$
$h(k_2) = h(k_5)$
$h(k_3)$
$m-1$

How Important are Symbol Tables?

- Databases are symbol tables
- Symbol tables are supported in the C and C++ standard libraries
  - bsearch() in stdlibc
  - binary_search() in STL
  - Associative containers in STL: set<>, map<>, multiset<>, multimap<>
  - C++11 STL: unordered_set<>, unordered_map<>, unordered_multiset<>, unordered_multimap<>

Dictionary ADT & Hashing

Hashing is an efficient implementation of:

- *Insert* a new item
- *Search* for an item (or items) having a given key
- *Remove* a specified item

Hashing is an inefficient implementation of:

- *Select* the k^{th} largest item in a symbol table
- *Sort* the symbol table

Hash Function

Two Parts

Hash Code: $t(\text{key}) \Rightarrow \text{intmap}$

- Maps the key into an integer

Compression Map: $c(\text{intmap}) \Rightarrow \text{address}$

- Maps the integer into the range $[0, M)$

Given key:

$h(\text{key}) \Rightarrow c(t(\text{key})) \Rightarrow \text{address}$

Good hash functions

- Benefits of hash tables depend on having good hash functions
- Must be easy to compute
  - Will compute a hash for every key
  - Will compute same hash for same key
- Should distribute keys evenly in table
  - Will minimize *collisions*
    - Collision: two keys map to same address
  - Trivial, poor hash function: `h(key) { return 0; }`
    - Easy to compute, maximizes collisions

Hash Function

Definition: transforms *key* into table address

- Table of size M
  - About the number of elements expected
  - If unsure, guess high
- Function that transforms keys into integers in range $[0, M)$
- That is, $h(key) \Rightarrow 0..M - 1$

Hash Function: Floats in fixed range

- *key* between 0 and 1: [0, 1)

  $h(key) = \lfloor key * M \rfloor$

- *key* between *s* and *t*: [*s*, *t*)

  $h(key) = \lfloor (key - s) / (t - s) * M \rfloor$

- Try this: range of [1.38, 6.75), M = 13, find h(3.65)

Hash Function: Division Method
w-bit integers

Modular hash function

$h(key) = key \bmod M$

- Great if keys randomly distributed
  - Often, keys are not randomly distributed
  - Example: midterm scores cluster on 75
- Don’t want to pick a bad *M*, where bad:
  - *M* and *key* have common factors

Hash Function: Mult Method
w-bit integers

- Combination modular and multiply

  $h(key) = \lfloor key * \alpha \rfloor \bmod M$

  – Say $\alpha = 0.618033 = (\sqrt{5} - 1)/2$

  – And M can be prime or not (if not prime, α does all the work of attempting to prevent collisions)

Hash Code: strings

- Consider the following strings:
  - stop, tops, pots, spot
- ASCII sum of each is equivalent
- All will map to same hash table address
  - i.e., will cause collision
- Position is important

Hash Code: strings

Polynomial Hash Code for a string x with k letters

- $x_0 a^{k-1} + x_1 a^{k-2} + \ldots + x_{k-2} a + x_{k-1} a^0$

If $a = 31$ then

- t("tops") $= 116*31^3+111*31^2+112*31+115$
  $= 3,566,014$
- t("pots") $= 112*31^3+111*31^2+116*31+115$
  $= 3,446,974$

Compression Mapping

$c(intmap) \Rightarrow$ range $[0, M)$

- *intmap* may be < 0 or >= *M*

Division Method

$|\text{intmap}| \bmod M$, where *M* is prime

MAD (multiply and divide) Method

$|a * \text{intmap} + b| \bmod M$, where *M* is prime and *a* and *b* are non-negative integers

Or choose *a* and *b* prime, don't restrict *M*

Note: $a \bmod M$ must not equal 0!

Complexity of Hashing

For simplicity, assume perfect hashing (no collisions)

- What is cost of insertion? O( )
- What is cost of search? O( )
- What is cost of removal? O( )

Wouldn't it be nice to live in a perfect world?

Summary: Hash Tables

- Efficient ADT for insert, search, and remove
- Hash Function $h(key) \Rightarrow addr$
  - Maps key to table address
  - Hash code $t(key) \Rightarrow intmap$
    - Translates key into integer
  - Compression map $c(intmap) \Rightarrow addr$
    - Maps integer into range of 0 to M - 1 addresses
- Therefore, $h(key) \equiv c(t(key))$