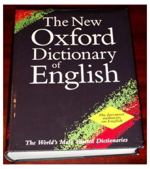
Lecture 15
Dictionaries and Hash Tables



EECS 281: Data Structures & Algorithms

Dictionary ADT

A container of items (key/value pairs) that supports two basic operations

- ***Insert*** a new item
- ***Search (retrieve)*** an item with a given key

Two primary uses

- A set of things: check if something is in the set
- Key-Value storage: look up values by keys

Dictionary ADT Operations

- Desirable Operations
  - **Insert** a new item
  - **Search** for item(s) having a given key
  - **Remove** a specified item
  - **Sort** the dictionary
  - **Select** the k^{th} largest item in a dictionary
  - **Join** two dictionaries
- Other basic container operations: construct, test if empty, destroy, copy...

Containers with key lookup

Identifying a container with <u>*fast search*</u> **and** <u>*fast insert*</u> for arbitrary `<key, value>` pairs

- Sorted Vectors: Insert will be ***O*(*n*)**
- Unsorted Vectors: Search will be ***O*(*n*)**
- Sorted/Unsorted Linked Lists: Search is ***O*(*n*)**
- Binary Search Tree (`std::map<>`)
  - Search: average ***O*(log *n*)**, worst-case ***O*(*n*)**
  - Insert: average ***O*(log *n*)**, worst-case ***O*(*n*)**
- Hash Table (`std::unordered_map<>`)
  - Search: average ***O*(1)**, worst case ***O*(*n*)**
  - Insert: average ***O*(1)**, worst case ***O*(*n*)**

What if the set of keys is small?

- 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 Defined

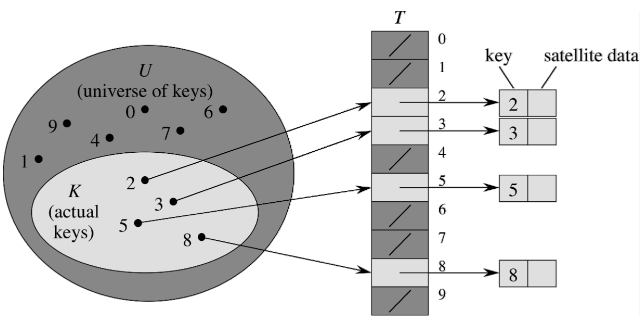
Locate items in a table by key

- Use arithmetic operations to calculate a table index (bucket) from a given key

Need

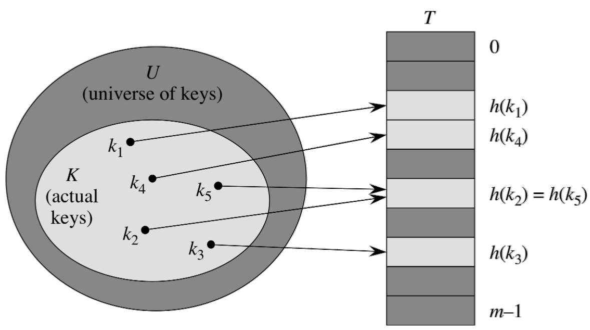
- **Translation**: converts a search key into an integer
- **Compression**: limits an integer to a valid index
- **Collision resolution**: resolves search keys that hash to same table index

Direct Addressing



Each key maps to an element of the array

Hashed Addressing



Only actual keys map to an element of the array

Hash Function

Translation: $t(key) \Rightarrow hashint$
- Converts the key into an integer

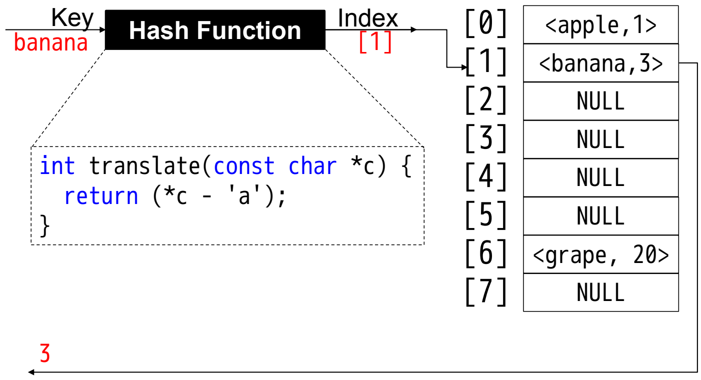
Compression: $c(hashint) \Rightarrow$ table index
- Maps the hashed integer into the range $[0, M)$

A hash function combines translation and compression:

$h(key) \Rightarrow c(t(key)) \Rightarrow$ table index

Note: `std::hash<>` provides only translation, not compression

Hashing Example



Translating Floating Points

- *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: range = [1.38, 6.75), $M = 13$
  compute $h(3.65)$

Translating Integers

Modular hash function

$t(key) = key$

$h(key) = c(t(key)) = key \bmod M$

- Great if keys randomly distributed
  - Often, keys are not randomly distributed
  - Example: midterm bubbles scores all multiples of 2.5
  - Example: pick a number from 1 to 10
- Don't want to pick a bad *M*
  - BAD: *M* and *key* have common factors

Translating Strings

- Simple hash sums ASCII character codes
- Problem Case: stop, tops, pots, spot
  - Sum of each is equivalent
  - All will map to same hash table address
  - Multiple collisions
- Solution: Character position is important
  - Consider decimal numbers

    $123 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0$

    $321 = 3 * 10^2 + 2 * 10^1 + 1 * 10^0$

Better Translation: Rabin Fingerprint

- Instead of adding up character codes, view strings as *decimal numbers*
  - Characters: `'T', 'O', 'M', ' ', 'M', …`
  - ASCII codes: `84, 79, 77, 32, 77, …`
  - Running fingerprints:

    T **84**

    T**O** 10 * 84 + **79**

    TO**M** 10 * (10 * 84 + 79) + **77**

    TOM█ 10 * (10 * (10 * 84 + 79) + 77) + **32**, …
  - Base 10 is used for illustration only (use larger numbers)
- **Shuffling the chars usually changes result**

Compression

$c(hashint) \Rightarrow$ index in range $[0, M)$

– *hashint* may be < 0 or >= *M*

Division Method

$|hashint| \bmod M$, where *M* is prime

MAD (multiply and divide) Method

$|a * hashint + b| \bmod M$, where *a* and *b* are prime

Use this method when you can't control *M*

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

Hash Function

Translation: $t(key) \Rightarrow hashint$
- Converts the key into an integer

Compression: $c(hashint) \Rightarrow$ table index
- Maps the hashed integer into the range $[0, M)$

A hash function combines translation and compression:

$h(key) \Rightarrow c(t(key)) \Rightarrow$ table index

Hashing Example

Key — lemon → **Hash Function** → Index [11]

```
int translate(const char *c) {
  return (*c - 'a');
}
```

[0]	<apple,1>
[1]	<banana,3>
[2]	NULL
[3]	NULL
[4]	NULL
[5]	NULL
[6]	<grape, 20>
[7]	NULL

Need compression!

Hash Table Size

- Table of size *M*
  - About the number of elements expected
  - If unsure, guess high
- Hash function must return keys as integers in range [0, *M*)

Index Compression

Key — lemon → **Hash Function** → Index [3]

```
int translate(const char *c) {
  return (*c - 'a');
}
int compression(int i) {
  return i % table_size;
}
```

[0]	<apple,1>
[1]	<banana,3>
[2]	NULL
[3]	NULL
[4]	NULL
[5]	NULL
[6]	<grape, 20>
[7]	NULL

Collision

Key — orange → **Hash Function** → Index [6]

```
int translate(const char *c) {
  return (*c - 'a');
}
int compression(int i) {
  return i % table_size;
}
```

[0]	<apple,1>	
[1]	<banana,3>	
[2]	NULL	
[3]	<lemon, 2>	
[4]	<mango, 4>	
[5]	NULL	
[6]	<grape, 20>	[!!!]
[7]	<pear, 1>	

Hash Tables Summarized

- Efficient ADT for insert, search, and remove
- Hashing a key *h*(*key*) ⇒ *table index*
  - Maps key to table index in two steps
  - Translates key into an integer
    t(*key*) ⇒ *hashint*
  - Compression maps integer into range [0, *M*)
    c(*hashint*) ⇒ *table index*
- Therefore, *h*(*key*) ≡ *c*(*t*(*key*))

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 index
- Trivial, poor hash function: `h(key) { return 0; }`
  - Easy to compute, but poor distribution maximizes collisions

Complexity of Hashing

For simplicity, assume perfect hashing (no collisions)

- What is cost of **insertion**? ***O***(1)
- What is cost of **search**? ***O***(1)
- What is cost of **removal**? ***O***(1)

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

- Recall the Pigeonhole Principle

Real-World Hash Tables

- C++ hash table containers (STL, C++11+)
  - `unordered_set<>, unordered_multiset<>`
  - `unordered_map<>, unordered_multimap<>`
- Database indices are built on hash tables
- Compilers use a hash table for identifiers

Major Uses of Hash Tables

- Sets of ints, strings, images, class objects
  - Check set membership
  - Detect/filter duplicates
  - Find unique elements
  - Find matching elements, e.g., $a + b = 0$

- Key-value storage: look up values by keys
  - Count things: value_type = int
  - Maintain *sparse* vectors: key_type = int
  - Maintain linked structures: key_type = value_type

Sorting a Hash Table?

Key → **Hash Function** → Index

```
int translate(const char *c) {
  return (*c - 'a');
}
int compression(int i) {
  return i % table_size;
}
```

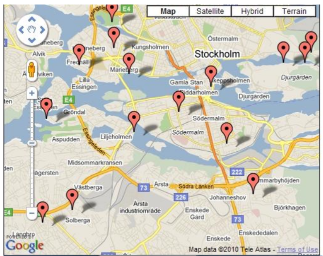
[0]	<apple,1>
[1]	<banana,3>
[2]	NULL
[3]	<lemon, 2>
[4]	<mango, 4>
[5]	NULL
[6]	<grape, 20>
[7]	<pear, 1>

Dictionary ADTs & Hashing

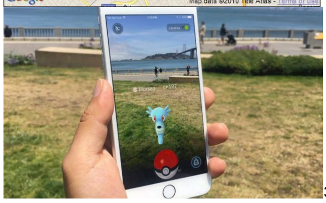
Hashing is an **efficient** implementation of:
- *Insert* a new item
- *Search* for an item (or items) having a given key
- *Remove* a specified item

Hashing is an **inefficient** implementation of:
- *Select* the k^{th} largest item in a dictionary
- *Sort* the items in the dictionary

Composite Hash Functions



- Example: hang some info over geo-locations
  - Hash {long, lat} pairs
- **How do we combine two hash functions?**
- $H(\{x, y\}) = (h(x) + p * h(y)) \% M$
- How do we hash class objects?



Is (h_1 + p * h_2) % M a good hash combiner?

- Recall properties of good hash functions
  - Fast computation
  - Even distribution of values
  - **No easily-predictable collisions**

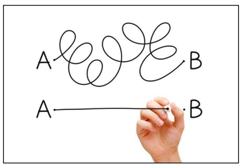
$((h_1 + p*h_2) + p*h_3) \% M = ((h_1 + p*h_3) + p*h_2) \% M$

- A better combiner for n values ($i = 0 .. n - 1$)

`seed ^= h_i + C + (seed << 6) + (seed >> 2)`

When Not To Use Hash Tables

- In many applications, it is tempting to use `unordered_set<>` or `unordered_map<>`, but
  - Significant space overhead of nested containers
  - Every access computes hash function
  - ***O*(*n*)** worst-case time (STL implementations)



When Not To Use Hash Tables

- When keys are small ints, use **bucket arrays**
  - Some keys can be coerced to small integers: enums, simple fractions, months

- For static data, set membership, or lookup
  - Consider sorting + binary search
- For key-value storage where traversals are needed but not lookup (e.g., sparse vec)
  - Store key-value pairs in a list (or vector)

Word Count Demo

VIDEO

From a web browser:

`bit.ly/eecs281-wordcount-demo`

From a terminal:

`wget bit.ly/eecs281-wordcount-demo -O wordcount.cpp`

At the command line:

```
g++ -std=c++1z wordcount.cpp -o wordcount
./wordcount
Enter filename: wordcount.cpp
```