Today

- Hash tables (cont)

Assignments

- HW8 out (due Tues)
Approach 4: String-Valued Keys

Q: What should we do if we have string-valued keys?
   - Convert strings to ints using their character ASCII codes
   - Then apply mod to the integer values

As an example, just add up each ASCII character value ...

"NOTE" = 78 + 79 + 84 + 69 = 310

Q: Do you see a problem with this?
   - Anagrams ... "TONE" == "NOTE"
   - Other words that add up to 310?

Better approach is to “weight” by position ... for example:

"NOTE" = 1 \cdot 78 + 2 \cdot 79 + 3 \cdot 84 + 4 \cdot 69 = 764

"TONE" = 1 \cdot 84 + 2 \cdot 79 + 3 \cdot 78 + 4 \cdot 69 = 752

In Java, e.g., weight is $31^{n-1-i}$ where $i$ is character index position

"NOTE" = 78 \cdot 31^3 + 79 \cdot 31^2 + 84 \cdot 31^1 + 69 = 2,402,290

Note on terminology:
   - Usually the value is the “hash code” (e.g., weighted string value)
   - And the mapping to an index is the “hash function”
Review: Properties of “Good” Hash Functions

They should be functions!

- Same key should always result in the same index

Be fast to compute

- Slow hash functions add overhead in search, insertion, remove operations
- Modulo is generally a simple division operation to calculate
- Summing string values depends on string size

Should try scatter data evenly throughout the hash table

- Minimize collisions as possible ... i.e., when two keys mapped to same index
- Beware “skewed” key distributions
Hash functions for HW 7

Since our KV Collections allow any key type, defining a hash function is tricky.

C++ helps with this by providing a generic version of a `hash` function

```cpp
#include <functional>

...

std::hash<K> hash_fun; // K-based hash function object
size_t value = hash_fun(key); // get int-based value for key
size_t index = value % capacity; // calculate the index

...
```

C++ provides default implementations for a number of base types

- also possible to add custom implementations (e.g., for a new class)
- `std::hash` is really a struct type, with overridden `operator(const T&)`
Resolving Collisions – Insertion Operations

Assume the hash function is \( \text{hascode} \ % \ 97 \) for a 97-element table

- \( h(236) = 42 \)
- \( h(270) = 76 \)
- \( h(211) = 17 \)
- \( h(601) = 19 \)
- \( h(527) = 42 \) ... collision!

There are two general approaches for dealing with collisions:

**Open Addressing:**
- if index location is occupied, find another available location
- we’ll look at different ways to “find another location”

**Restructure the Table:**
- add more room to the table to fit the new item
- we’ll look at “chaining” and table “resizing”
Open Addressing Approach 1: Linear Probing

Basic Idea:

- hash to “bucket” index
- bucket may be empty or not empty
- If occupied, move to next empty bucket
- If get to end of table, keep searching at 0

Linearity probing can result in large “primary” clusters of keys
- i.e., large sequences of non-empty buckets

Maintain two types of empty “buckets” (for searching)
- empty-since-start ... empty since start of hash table
- empty-after-removal ... item removed making bucket empty
How linear probing works ...

Insert:
- Linear probe (sequentially) until first empty bucket found (either type)
- Insert at first empty bucket
- Note: The table could be full! (which means we can’t insert)

Removal:
- Use similar approach to find key item
- Stop at empty-since-start bucket (item not in table)
- Mark item as empty-after-removal

Search:
- Go to hashed index
- Start searching sequentially for key
- Stop when an empty-since-start bucket is found or searched entire table
Open Addressing Approach 2: Quadratic Probing

Basic Idea:

- similar to linear probing
- but probe “quadratic” sequences
- \( i + 1, i + 2^2, i + 3^2, i + 4^2 \), and so on
- helps “spread out” primary clusters

Creates “secondary” clusters since collisions use same quadratic sequences
Open Addressing Approach 3: Double Hashing

**Basic Idea:**

- use a secondary hash function $h_2$ to determine size of sequence “steps”
- where number of steps depend on the key
- can further help to reduce clustering

![Diagram of a table with keys and hash function values]
Alternative Approach: Restructuring the Table

Instead of probing, allow multiple items to be stored at each index

“Separate Chaining”

- Each array index has its own **linked list** of elements (the “chain”)

- The list grows and shrinks with collisions and removals
Benefits of Separate Chaining

Ideally, insert, remove, search are $O(1)$

However, collisions **increase** the cost

- Cost depends on the **load factor** ... how full the table is
  
  \[
  \text{load\_factor} = \frac{\text{collection\_size}}{\text{table\_capacity}}
  \]

- As table fills, chance of collision increases

- And hashing efficiency decreases (e.g., using open addressing)

Cost of **separate chaining**

- Insertion is still $O(1)$ (e.g., insert at front of linked list chains)

- Removal and search require navigating linked list chain ...
  
  - cost to search chains depends on number of collisions
  - assuming a “good” hash function (distributes keys)
  - \text{load\_factor} < 100\% implies at most length 1 chains
  - \text{load\_factor} > 100\% implies average chain length over 1

- Worst case search and remove cost is $O(n)$ since ...
  
  \[
  \text{avg\_chain\_len} = \text{load\_factor} = \frac{n}{\text{table\_capacity}}
  \]

- But in practice (like with quick sort), hash tables provide efficient search
Resizing and Rehashing

For HW 8 we are going to implement a “resizable” Hash Table

Our implementaton is inspired by Java’s HashMap data structure

- Like with HashMap:
  - the load factor threshold is 75%
  - the table array is doubled when the load factor goes over 75%
  - the table array is never decreased in size
  - the initial size of the table array is 16
  - separate chaining is used

When the load factor goes over 75% the array is resized and rehashed

- a new table array is created with twice the size
- (re-)hash elements in current table into new array (via new $h'$ function) mini
- the old table (and linked list chains) are de-allocated / deleted

Q: What are advantages and disadvantages?

More implementation details are provided in HW 8