Today

- Midterm info
- Hashing continued

Assignments

- HW 6 out
- Midterm exam Thurs
Midterm Exam

Basics:
- closed book, computer, notes
- 4 multipart questions
- worth 5% of final grade

Possible Question Topics:
- C++ programming
  - dynamic vs static function binding (virtual)
  - generics (type parameters, templates)
  - abstract classes (pure virtual)
  - operator overloading
  - essential operators
- Algorithm analysis
  - best-case vs worst-case
  - detailed analysis
  - definition of $O$, $Ω$, $Θ$
- Sorting
  - insertion, selection, bubble sort ($O(n^2)$ sorting)
  - merge sort, quick sort (arrays and linked lists)
- Data types and data structures
  - sequence and map
  - linked lists and resizable arrays
  - binary search (based map)
**Hash function examples (cont)**

**Approach 1: “Selecting Digits”**

- Select specific parts of (integer) key to use as the hash value
- Example: assuming keys are 9-digit employee numbers
  - Let \( h(k) = 4\text{th and 9th digit of } k \)
  - E.g., \( h(001364825) = 35 \)
  - Given key 001364825, insert entry at table[35]
- Fast hash function, but may not evenly distribute data ... why?
- How can we use with strings? ... replace characters with ASCII codes

**Approach 2: “Folding”**

- Add (sum) digits of the key
- Example: again assuming 9-digit employee numbers
  - \( h(k) = i_1 + i_2 + \cdots + i_9 \) where \( k = i = i_1 i_2 \cdots i_9 \)
  - Since \( h(001364825) = 29 \), insert entry at table[35]
- Also fast, but also may not evenly distribute data
- In this example, only hits ranges from 0 to 81
- Can pick different (similar) schemes ... e.g., \( i_1 i_2 i_3 + i_4 i_5 i_6 + i_7 i_8 i_9 \)
**Approach 3:** Modular Arithmetic

- Often end up with indexes **outside** of the range of table indexes
- Use modulo operator $\%$ (mod) to map values to valid table indexes
  
  $h(k) = f(k) \mod \text{tables}i\text{ze}$
  
  ... for $f : \text{Key} \rightarrow \text{int}$

  - technically, $f$ generates a "hash code" and $h$ maps it to a table index
- Example: if $f(k) = k$
  
  $h(001364825) = 1,364,825 \mod 100 = 25$

Good $h$ index distribution may require carefully chosen table sizes

- Since keys may not be random (e.g., multiples of 2 or 10 or ...)
- E.g., 110 mod 100, 210 mod 100, 310 mod 100, etc
- Often use prime numbers (to help distribute values) like 101 in this case

**Approach 4:** Weighted sum to help minimize collisions

- Sum approach can lead to collisions with different words:
  
  "NOTE" = 78 + 79 + 84 + 69 = 310
  
  "TONE" = 84 + 79 + 78 + 69 = 310

- Weighted sum by position
  
  "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, 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
“Universal” Hash functions:

- set of hash functions when randomly chosen, minimize chance of collisions
- for $m$ buckets and any two values $x \neq y$, the collision probability is (at most):
  \[ \Pr(h(x) = h(y)) \leq \frac{1}{m} \]
- an example of a universal hash function is:
  \[ h(x) = ((k \cdot x + q) \mod p) \mod m \]
  where $k > 0$ and $q$ are randomly chosen integers and $p \geq m$ is prime

Examples of hash functions “in the wild”

- Cryptographic hash functions, e.g., the “Secure Hash Algorithm” (SHA)
  - unlike for hash tables, made to be computationally hard to “invert”

- MurmurHash (non cryptographic)
  - mixes input bits to produce output (hash code) bits
  - a version of MurmurHash is used for C++ strings
  - also used in Apache Hadoop, nginx, and many other applications

- FarmHash (non cryptographic)
  - by Google for fast hashing of small strings (e.g., up to 64 bytes)
  - based on a version of MurmurHash
  - useful for “probabilistic” data structures (like Bloom Filters)

- Plus many others!
Hash functions for HW-6

Since Map implementations allow any key type, defining a hash function is tricky

C++ helps 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&)`
Resizing and Rehashing

For HW 6 we are going to implement a “resizable” Hash Table

Our implementation 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)
- the old table (and linked list chains) are de-allocated / deleted

Q: What are advantages and disadvantages?
More details on resizing and rehashing

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

1. a new table array is created with twice the size (the `new_table_capacity`)
2. (re-)hash elements in current table into new table:
   - `int new_index = hash_fun(key) % new_table_capacity;`
   - note that “table capacity” is the size of the array
   - iterate through elements in old table, and insert into new table
3. the old table (and linked list chains) are de-allocated / deleted
4. the new table replaces the old one

Note this is the basic idea ...
   - multiple ways to carry out the above

Also note this is very similar to our `ArraySeq` implementation
   - but we have to deal with the linked list chains
   - we have a different load factor threshold (75% instead of 100%)
   - and we have to rehash when we resize
Declaring the hash table data structure:

```c
struct Node {
    K key;
    V value;
    Node* next;
};

Node** hash_table;
```

- Each bucket holds a linked list (**Node**)
- We have an array of these (the second “star” ... pointer to **Node**)
- Just like the type of **int array[]** is **int** array

We declare and implement a private helper function to resize and rehash:

```c
void resize_and_rehash();
```

We have three additional member variables:

```c
// number of k-v pairs in the collection
int count = 0;

// number of buckets (array capacity)
int capacity = 16;

// load threshold to trigger resize and rehash
double load_factor_threshold = 0.75;
```

Need to implement all the essential operators/functions

Q: Why? ...
The insert, contains, operator[], and erase functions:

- For insert, hash to bucket then add at front of linked list
- For contains, hash to bucket, find node in linked list
- For operator[], hash to bucket, find node, return value
- For erase, hash to bucket, find node, remove
- Note: in insert, need to check for resize and rehash

The find-keys and sorted-keys functions:

- For both have to “iterate” through entire hash table
- For sorting, get keys, then call sort on resulting array sequence (like HW-5)