Where we’ve been and where we are going…

Our goal so far: We want to store and retrieve data (keys) fast

Tree structures

- **BSTs**: simple, fast in the average case, but can perform poorly with ordered data (you implemented)
- **AVL trees**: Guaranteed fast, but tricky to implement (you will not implement)
- **RSTs (built on treaps)**: Simpler than AVL trees, usually fast, even with ordered data (you will not implement)
- **Red-black trees**: Guaranteed fast, built in to C++ (coming soon… but you will not implement)

Hash tables (today thru Friday):

- *Very* fast in the average case
- No sorted access to data
- Can be tricky to do well (you are using an implementation built into C++ in PA3)

Note: you are responsible for knowing how to work with all of these structures as we did in class/in the reading
Fast Lookup: Hash tables

- So far, almost all of our data structures have had performance $O(\log N)$ for insert and find.

- Operations supported by hash tables:
  - Find (key based look up)
  - Insert
  - Delete

- Average run times:

- No worst case guarantees
Hash table Motivation: Two sum problem

- Consider the 2-sum problem: Given an unsorted array (A) of N integers between 0 and 1000,000, find all pairs of elements that sum to a given number T

```
for (i = 0; i < N; i++)
    for (j = i + 1; j < N; j++)
        if ((A[i] + A[j]) == T)
            store (A[i], A[j]);
```

Q: What is the worst case run time of this method? Assume the store method (inserts the elements into an unsorted linked list)

A. O(N)
B. O(NlogN)
C. O(N^2)
Hash table Motivation: Two sum problem

• Consider the 2-sum problem: Given an unsorted array \( A \) of \( N \) integers between 0 and 1,000,000, find all pairs of elements that sum to a given number \( T \)

\[ A \]

• Method 2: Sort first then search

\[
\text{sort} \ (A); \\
\text{for} \ (i = 0; \ i < N; \ i++)\{ \\
\quad j = \text{binary_search}(T - A[i]); \\
\quad \text{store} \ (A[i], \ A[j]); \\
\}
\]

Q: What is the worst case run time of this method? Assume the store method (inserts the elements into an unsorted linked list)

A. \( O(N) \)
B. \( O(N \log N) \)
C. \( O(N^2) \)
De-tour: Finding data fast

• Recall:
The data you will store will always be integers between 0 and 1,000,000. You decide to use an array with 1,000,000 Boolean values to store your data. An entry will be true if the item is in your structure, and false otherwise.

• What is the (Big-O) running time to insert an item into this proposed structure?
  A. O(N)
  B. O(logN)
  C. O(1)

The Big O running time of a find will be:
Can we use this results to improve the performance of the two-sum problem
Hash table Motivation: Two sum problem

- Consider the 2-sum problem: Given an unsorted array (A) of N integers between 0 and 1000,000, find all pairs of elements that sum to a given number T

A

- Method 3: Use an array based lookup table
  ```cpp
  bool Hashtable[ARRAY_SIZE]={false};
  for (i = 0; i<N; i++){
      insert(Hashtable, A[i]);
  }
  for (i = 0; i<N; i++){
      if (find(Hashtable, T-A[i]) == true)
          store (A[i], A[j]);
  }
  ```

Q: What is the worst case run time of this method? Assume the store method (inserts the elements into an unsorted linked list)

A. O(N)  
B. O(NlogN)  
C. O(N^2)  

What is the problem with this method?
Fast Lookup: Hash functions

• Setup for hashing:
  • Universe of possible keys U
  • Keep track of evolving set S
  • |S| is approximately known
Hashing

• Let’s modify our array-based lookup table
• Need a hash-function $h(x)$: takes in a key, returns an index in the array
• Gold standard: random hash function

Hash function

Key

Hash Code (HC)

(index data)  

Size is proportional to # of keys (not value of keys)

(might be collisions)

Hash table (array)
Probability of Collisions

• Suppose you have a hash table that can hold 100 elements. It currently stores 9 elements (in 9 different locations in the hash table). What is the probability that your next insert will cause a collision (assuming a totally random hash function)?

A. 0.09  
B. 0.10  
C. 0.37  
D. 0.90  
E. 1.00
Probability of Collisions

• Suppose you have a hash table that can hold 100 elements. It currently stores 30 elements (in one of 30 possible different locations in the hash table). What is the probability that your next two inserts will cause at least one collision (assuming a totally random hash function)? (Choose the closest match)

A. .09
B. .30
C. .52
D. .74
E. .90
Probability of Collisions

If you have a hash table with M slots and N keys to insert in it, then the probability of at least 1 collision is:

\[ P_{N,M}(\text{collision}) = 1 - P_{N,M}(\text{no collision}) \]

\[ = 1 - \prod_{i=1}^{N} P_{N,M}(\text{ith key no collision}) \]
Hashtable collisions and the "birthday paradox"

• Suppose there are 365 slots in the hash table: M=365
• What is the probability that there will be a collision when inserting N keys?
  • For N = 10, prob\textsubscript{N,M}(collision) = 12%
  • For N = 20, prob\textsubscript{N,M}(collision) = 41%
  • For N = 30, prob\textsubscript{N,M}(collision) = 71%
  • For N = 40, prob\textsubscript{N,M}(collision) = 89%
  • For N = 50, prob\textsubscript{N,M}(collision) = 97%
  • For N = 60, prob\textsubscript{N,M}(collision) = 99+%  

• So, among 60 randomly selected people, it is almost certain that at least one pair of them have the same birthday
• On average one pair of people will share a birthday in a group of about \( \sqrt{2 \cdot 365} \approx 27 \) people
• In general: collisions are likely to happen, unless the hash table is quite sparsely filled
• So, if you want to use hashing, can’t use perfect hashing because you don’t know the keys in advance, and don’t want to waste huge amounts of storage space, you have to have a strategy for dealing with collisions
Making hashing work

- Important issues in implementing hashing are:
  - Deciding on the **hash function**
  - Deciding on the **size of the hash table**
  - Deciding on the **collision resolution strategy**

- With a good hashtable design, $O(1)$ average-case insert and find operation time costs can be achieved, with $O(1)$ space cost per key stored

- This makes hashtables a very useful and very commonly used data structure
Hash functions: desiderata

Important considerations in designing a hash function for use with a hash table
- It is fast to compute (must be O(1))
- It distributes keys evenly
- It is consistent with the equality testing function (i.e. two keys that are equal will have the same hash value)

Designing a good hash function is not easy!
We’ll look at a few, but there’s much more to explore.
Simple (and effective?) hash function for integers: \( H(K) = K \mod M \)

- When is the function \( H(K) = K \mod M \) where \( K \) is the key and \( M \) is the table size a good hash function for integer keys?
  A. Always
  B. When \( M \) is prime
  C. When values of \( K \) are evenly distributed
  D. In the case of either B or C
  E. Never
Simple (and effective?) hash function for integers: $H(K) = K \mod M$

- When is the function $H(K) = K \mod M$ where $K$ is the key and $M$ is the table size a good hash function for integer keys?

  When $M$ is prime OR When values of $K$ are evenly distributed.

Think of an example of a non-prime $M$ and a distribution of keys that would cause poor behavior from this function.
Simple (and effective?) hash function for integers: $H(K) = K \mod M$

- When is the function $H(K) = K \mod M$ where $K$ is the key and $M$ is the table size a good hash function for integer keys?

  When $M$ is prime OR When values of $K$ are evenly distributed

Because you can never count on evenly distributed keys, always use prime-size table with this hash function.
Hash table size

• By "size" of the hash table we mean how many slots or buckets it has

• Choice of hash table size depends in part on choice of hash function, and collision resolution strategy

• But a good general “rule of thumb” is:
  • The hash table should be an array with length about 1.3 times the maximum number of keys that will actually be in the table, and
  • Size of hash table array should be a prime number (especially with the simple hash function we looked at)

• So, let $M =$ the next prime larger than 1.3 times the number of keys you will want to store in the table, and create the table as an array of length $M$

• (If you underestimate the number of keys, you may have to create a larger table and rehash the entries when it gets too full; if you overestimate the number of keys, you will be wasting some space)
Next time

• Hash functions
• Resolving collisions in hash tables