ECE 590.01
C++ Programming, Data structures, and Algorithms
Hash tables

Admin

- Reading
  - Chapter 5
- Midterm Friday
- Wednesday:
  - Review/catch-up on calendar
  - All caught up so review or extra practice
    - Unless we don’t finish today’s lecture (if not, it’s first Wed)
    - Feel free to discuss specific suggestions/requests on Piazza

What have we been talking about?

- What did we talk about last time?
  - Balanced BSTs
  - AVL
  - Red/Black

Advantage of these:
  - Guarantee \( O(\log N) \) access

Can we do better?

- Theoretical answer:
  - An infinite number of things
  - Most of them aren’t very likely to come up
    - Probably not going to find many algorithms that run in
      - \( O(\sqrt{\log N}) \)
      - \( O(\sin N) \)
      - \( O(1 / N) \)
      - Etc.
  - Practically speaking:
    - \( O(\log^* N) \)—count how many times you have to \( \log \) to get to 1
    - \( \log^*(N) = N \leq 1 \) if \( \log \) to 1
    - \( O(1) \)
    - Constant time

What is better than \( O(\log N) \)

- \( O(\log N) \) is pretty good
  - Billions of items? ~30 steps? Not bad

- Can we do better?
  - Question 1: what is better than \( O(\log N) \)
  - Question 2: can we do it?
### O(1): Can we do it?

- O(1) sounds great
  - 1000 things? K steps
  - 10,000 things? Still K steps
  - 100,000,000,000 things? Still K steps.
  - Nice...
  - But sounds hard
- So, can we do it?

### So how do we do this?

- First: what has O(1) access?
  - Arrays

- Ok, so lets suppose were are doing a set of ints
  - Make an array of booleans
  - Index into the array directly
    - boolean contains(n) { return array[n]; }

- Sounds great.. How big of an array do we need?

- First: what has O(1) access?
  - Arrays

- Ok, so lets suppose were are doing a set of ints
  - Make an array of booleans
  - Index into the array directly
    - boolean contains(n) { return array[n]; }

- Sounds great.. How big of an array do we need?
  - 32-bit ints: 4GB (assuming 1 byte per boolean)
  - Size doesn't scale with elements in data structure
  - Scales with number of possible keys
  - Would be very sparse (not many elements for space used)
Use the Array Differently

• Indexing the array directly is problematic:
  boolean contains(n) { return array[n]; }

• What if we index the array with some function of n instead
  boolean contains(n) { return array[f(n)]; }

• What would this function f have to do?
  * Need to map all possible input values (~4B) down to array size
  * Also need to not have collisions
  * f(x) != f(y) if x != y
  * Sadly this is impossible
    * Pigeon hole principle

• No collisions: impossible
  * Instead: aim for infrequent collisions
  * And have a way to detect/deal with them

Hashing

• Terminology: Hashing function
  * "f" in prior slides is a hashing function
  * Maps from a larger set down to a smaller set
  * Implicitly throws away information
  * Good hashing functions
    * Spread different inputs out roughly uniformly over output space
  * Note hashing has applications outside of hash tables

Collisions: Step 1

• Let's start with "mod array_size"
  * Generally good, unless some specific pattern in keys

• Need a way to check for collisions
  * Keep array of ints instead of array of booleans
    * May want (int + valid)
    * Or to use impossible numbers as invalid
  * Use to track what is there

Example

• 7 element array
  * Access at N % 7

Add 73

73 % 7 = 3

Check if the set contains 22?

22 mod 7 is 1, but 15 is in that slot, not 22: so no
Example

Add 10
10 mod 7 is 3
Problem: 73 is in that spot
Can’t just kick it out
So what to do?

Handling Collisions

- Many options:
  - Option 1: Linear probing
    - Spot X is taken? Try X + 1, then X + 2, ...
  - Option 2: Quadratic probing
    - Spot X is taken? Try X + 1, then X + 4, then X + 9, ...
    - Spread things out more if a cluster

- Pros and Cons?
  - Pro: relatively simple
  - Con: items can be anywhere now
  - Need O(array_size) to verify something isn’t there

External Chaining

- Now add 10

External Chaining

- Note that in order to keep O(1) access
  - List must stay very small
  - Not just be N/C for some constant C: that’s O(N)
Externally Chained Hash Tables

- Externally Chained Hash Tables: incredibly common
  - Easy: array of linked lists
  - Fast: generally O(1)
    - Just have to keep lists very short

- Terminology:
  - Buckets: spots in the array
  - Load factor: number of elements / array size

- Difficulty: birthday paradox
  - Important problem to understand for a lot of reasons
    - Hash table collisions
    - Security
    - ...

Birthday Paradox

- Given N people, how likely are at least 2 to have same birthday (month/day, not year)
  - Assuming no twins
  - Each birthday uniformly distributed
  - Ignore leap days (Feb 29)

- 10 people?
- 20 people?
- 30 people?
- 50 people?

Relevance to Hash tables

- In hash tables: want different items to end up in different "buckets"

- Chances of "doubling up": generalized birthday problem
  - May not be 365 buckets, same principles apply
  - A collision or two: not so bad...
  - But want to avoid many (especially on same bucket)

- How do we fix it?
  - **Rehash** when load factor gets too high

Rehashing

- Rehashing a table:
  - Create new array of size 2 * array size
  - Insert all elements from old table into new array
  - Free up old array memory

- Why 2x the size?
Rehashing

• Rehashing a table:
  - Create new array of size 2 * array_size
  - Insert all elements from old table into new array
  - Free up old array memory

• Why 2x the size?
  - Expensive operation: O(N)
  - Want to do it rarely
  - Amortize the cost across many adds
    - What is amortized cost of rehashing this way when needed?

Amortized add cost

• Add now:
  - Do normal O(1) add
    - if (need_rehashing) { rehash(); } //O(N) sometimes

• Worst case is O(N)
• Amortized cost, however, consider how often we do O(N) step
  - Each time we do it, it "buys us" N more adds
    - So N cost every N times -> N/N => 1
    - O(1) + O(1) = O(1)
  - Still have O(1) access!

When to rehash

• Rehashing: when to do it?
  - Load factor (N / array_size) > threshold

• What threshold to use?
  - Space (memory) vs time tradeoff:
    - Smaller threshold: chain length is lower, more space
    - Higher threshold: chain length is higher, less space
  - Generally somewhere in the 0.5 to 0.75 range

Chain with balanced trees?

• Externally chained with LLs: good
• Externally chained with AVL trees: better?
  - Probably not
  - Tree are good on LARGE data sets
    - O(N) vs O(lg N) when N is 4?
  - If your bucket is large, you are already losing
  - Constant factors probably higher
  - More work per "step"

Beyond ints

• So far, have talked about ints
  - What about other types?
  - Need a way to turn it into an int
  - Quite possible
    - Remember: everything is a number

• When we do this, just need to make sure its good
  - Hash a string by adding its letters?
    - "abc" = 3, "bac" = 3, "cab" = 3 —not sooo great
  - Might do something more complex/better

Hashing more broadly

• Hashing: its useful for more than just tables

• Many applications in security
  - Don’t store passwords, store hashes of passwords
  - User enters password
  - Hash it
  - Compare hash to stored hash to check

• No way for anyone to (easily) get original passwords
  - Employees with access to server
  - Hackers who get password file
Sophisticated Hashing Algorithms

- Crytographers come up with good algorithms
  - SHA
- MD5 is somewhat out-dated now
- Not the sort of thing to make up yourself if security matters
  - Easy to get wrong
  - Not going into math/crypto here

Birthday Paradox and Security

- As with hash tables, the birthday paradox can work against us with hashes for security
- This is not a class on crypto, so we won't go into it here
  - Good thing to learn about/understand

Wrap up

- Hashtables:
  - Good, fast, data structure for maps + sets
- See also:
  - http://www.cis.upenn.edu/~adhilton/cse399/hashtable.html