ADMINISTRIVIA

• Written homework due individually tonight
  • Broken into 5 problems
  • Please resubmit if you have already, this will make grading easier for us
• Project 2 is out tonight
  • This is a one week project (no checkpoint, so no opportunity for regrading)
  • Start early!
  • Implementing Hashtables and Hashsets
  • Canvas should be configured to allow you to make your own groups, hopefully this will make grade assignments easier
Project 1 EC and Part 1 regrades out Friday

- If you got a different grade than your partner, let me know
- EC is calculated separately from the rest of the course
TODAY’S LECTURE

• Hashtables
  • Review of probing methods
  • Separate Chaining
  • Implementation considerations
• Introduction
  • Suppose there is a set of data $M$
  • Any data we might want to store is a member of this set. For example, $M$ might be the set of all strings
  • There is a set of data that we actually care about storing $D$, where $D << M$
  • For an English Dictionary, $D$ might be the set of English words
What is our ideal data structure?

- The data structure should use $O(D)$ memory
  - No extra memory is allocated
- The operation should run in $O(1)$ time
  - Accesses should be as fast as possible
HASHING

• Memory: The Hash Table
  • Consider an array of size $c \times D$
  • Each index in the array corresponds to some element in $M$ that we want to store.
  • The data in $D$ does not need any particular ordering.
The Hash Function maps the large space $M$ to our target space $D$.

We want our hash function to do the following:

- Be repeatable: $H(x) = H(x)$ every run
- Be equally distributed: For all $y, z$ in $D$, $P(H(y)) = P(H(z))$
- Run in constant time: $H(x) = O(1)$
HASH FUNCTION

• In reality, good hash functions are difficult to produce
  • We want a hash that distributes our data evenly throughout the space
  • Usually, our hash function returns some integer, which must then be modded to our table size
  • Needs to incorporate all the data in the keys
HASH EXAMPLE

• Possible solutions:
  • Store in the next available space
  • Store both in the same space
  • Try a different hash
  • Resize the array
COLLISIONS

• Hash table methods are defined by how they handle collisions

• Two main approaches
  • Probing
  • Chaining
COLLISIONS

• Probing
COLLISIONS

• Probing
  • Linear probing
COLLISIONS

• Probing
  • Linear probing
    • Try the appropriate hash table row first
COLLISIONS

• Probing
  • Linear probing
    • Try the appropriate hash table row first
    • Increase the index by one until a spot is found
COLLISIONS

• Probing
  • Linear probing
    • Try the appropriate hash table row first
    • Increase the index by one until a spot is found
    • Guaranteed to find a spot if it is available
COLLISIONS

• Probing
  • Linear probing
    • Try the appropriate hash table row first
    • Increase the index by one until a spot is found
    • Guaranteed to find a spot if it is available
    • If the array is too full, its operations reach $O(n)$ time. Primary clustering
COLLISIONS

- Probing
  - Quadratic Probing
COLLISIONS

• Probing
  • Quadratic Probing
    • Rather than increasing by one each time, we increase by the squares
COLLISIONS

• Probing
  • Quadratic Probing
    • Rather than increasing by one each time, we increase by the squares
    • $k+1$, $k+4$, $k+9$, $k+16$, $k+25$
COLLISIONS

• Probing
  • Quadratic Probing
    • Rather than increasing by one each time, we increase by the squares
    • $k+1$, $k+4$, $k+9$, $k+16$, $k+25$
    • Certain tables can cause secondary clustering
COLLISIONS

• Probing
  • Quadratic Probing
    • Rather than increasing by one each time, we increase by the squares
    • $k+1, k+4, k+9, k+16, k+25$
    • Certain tables can cause secondary clustering
COLLISIONS

• Probing
  • Quadratic Probing
    • Rather than increasing by one each time, we increase by the squares
    • $k+1, k+4, k+9, k+16, k+25$
    • Certain tables can cause secondary clustering
    • Can fail to insert if the table is over half full
COLLISIONS

• Probing
  • Secondary Hashing
COLLISIONS

• Probing
  • Secondary Hashing
    • If two keys collide in the hash table, then a secondary hash indicates the probing size
COLLISIONS

• Probing
  • Secondary Hashing
    • If two keys collide in the hash table, then a secondary hash indicates the probing size
    • Need to be careful, possible for infinite loops with a very empty array
    • If the secondary hash value and the table size are coprime (they share no factors), then secondary hashing will succeed if there is an open space
COLLISIONS

• Probing
  • Secondary Hashing
    • If two keys collide in the hash table, then a secondary hash indicates the probing size
    • Need to be careful, possible for infinite loops with a very empty array
    • If the secondary hash value and the table size are coprime (they share no factors), then secondary hashing will succeed if there is an open space
    • If table size is prime, only need to check if hash is a multiple
PRIMALITY

• Array sizes
PRIMALITY

• Array sizes
  • We normally choose our hash tables to have prime size
PRIMALITY

• Array sizes
  • We normally choose our hash tables to have prime size
  • Why?
PRIMALITY

• **Array sizes**
  
  • We normally choose our hash tables to have prime size
  • This is because for any number we pick, so long as it is not a multiple of our table size, they must be coprime
PRIMALITY

• Array sizes
  • We normally choose our hash tables to have prime size
  • This is because for any number we pick, so long as it is not a multiple of our table size, they must be coprime
  • Two numbers $x$ and $y$ are coprime if they do not share any common factors.
PRIMALITY

• **Array sizes**
  
  • We normally choose our hash tables to have prime size
  • This is because for any number we pick, so long as it is not a multiple of our table size, they must be coprime
  • Two numbers x and y are **coprime** if they do not share any common factors.
  • If the hash table size and the secondary hash value are coprime, then the search will succeed if there is space available
PRIMALITY

• **Array sizes**
  • We normally choose our hash tables to have prime size
  • This is because for any number we pick, so long as it is not a multiple of our table size, they must be coprime
  • Two numbers $x$ and $y$ are **coprime** if they do not share any common factors.
  • If the hash table size and the secondary hash value are coprime, then the search will succeed if there is space available
  • However, many primes cause secondary clustering when used with quadratic probing
COLLISIONS

• Chaining
COLLISIONS

• Chaining
  • Rather than probing for an open position, we could just save multiple objects in the same position
COLLISIONS

• Chaining
  • Rather than probing for an open position, we could just save multiple objects in the same position
  • Some data structure is necessary here
COLLISIONS

• Chaining
  • Rather than probing for an open position, we could just save multiple objects in the same position
  • Some data structure is necessary here
  • Commonly: a linked list, AVL tree or secondary hash table.
**COLLISIONS**

- **Chaining**
  - Rather than probing for an open position, we could just save multiple objects in the same position
  - Some data structure is necessary here
  - Commonly: a linked list, AVL tree or secondary hash table.
  - Resizing isn’t **necessary**, but if you don’t, you will get $O(n)$ runtime.
LOAD FACTOR

• When discussing hash table efficiency, we call the proportion of stored data to table size the load factor. It is represented by the Greek character lambda (\( \lambda \)).
When discussing hash table efficiency, we call the proportion of stored data to table size the *load factor*. It is represented by the Greek character lambda (\(\lambda\)).
LOAD FACTOR

• When discussing hash table efficiency, we call the proportion of stored data to table size the *load factor*. It is represented by the Greek character lambda (\(\lambda\)).
  - We’ve discussed this a bit implicitly before
  - What are good load-factor (\(\lambda\)) values for each of our collision techniques?
LOAD FACTOR

• Linear Probing?
• Quadratic Probing?
• Secondary Hashing?
• Chaining?
LOAD FACTOR

- Linear Probing?
- Quadratic Probing?
- Secondary Hashing?
- Chaining?
- What are the tradeoffs?
LOAD FACTOR

- Linear Probing?
- Quadratic Probing?
- Secondary Hashing?
- Chaining?

- What are the tradeoffs?
  - Memory efficiency
LOAD FACTOR

• Linear Probing?
• Quadratic Probing?
• Secondary Hashing?
• Chaining?
• What are the tradeoffs?
  • Memory efficiency
  • Failure rate
LOAD FACTOR

• Linear Probing?
• Quadratic Probing?
• Secondary Hashing?
• Chaining?
• What are the tradeoffs?
  • Memory efficiency
  • Failure rate
  • Access times?
LOAD FACTOR

• Linear Probing? $0.25 < \lambda < 0.5$
• Quadratic Probing?
• Secondary Hashing?
• Chaining?
LOAD FACTOR

- Linear Probing? $0.25 < \lambda < 0.5$
- Quadratic Probing? $0.10 < \lambda < 0.30$
- Secondary Hashing?
- Chaining?
LOAD FACTOR

• **Linear Probing?** 0.25 < \( \lambda \) < 0.5

• **Quadratic Probing?** 0.10 < \( \lambda \) < 0.30
  - If it gets to 0.5, then there is a chance of failure, and a high chance of O(n) runtime

• **Secondary Hashing?**

• **Chaining?**
LOAD FACTOR

- Linear Probing? $0.25 < \lambda < 0.5$
- Quadratic Probing? $0.10 < \lambda < 0.30$
- Secondary Hashing? $0.25 < \lambda < 0.5$
- Chaining?
LOAD FACTOR

- Linear Probing? $0.25 < \lambda < 0.5$
- Quadratic Probing? $0.10 < \lambda < 0.30$
- Secondary Hashing? $0.25 < \lambda < 0.5$
  - But we’ve eliminated primary clustering
- Chaining?
LOAD FACTOR

- **Linear Probing?** $0.25 < \lambda < 0.5$
- **Quadratic Probing?** $0.10 < \lambda < 0.30$
- **Secondary Hashing?** $0.25 < \lambda < 0.5$
- **Chaining?** $3.0 < \lambda < 10$
LOAD FACTOR

• Linear Probing? $0.25 < \lambda < 0.5$
• Quadratic Probing? $0.10 < \lambda < 0.30$
• Secondary Hashing? $0.25 < \lambda < 0.5$
• Chaining? $3.0 < \lambda < 10$

  • Because we allow multiple items in each space, we can increase memory efficiency by taking advantage
LOAD FACTOR

- Linear Probing? \(0.25 < \lambda < 0.5\)
- Quadratic Probing? \(0.10 < \lambda < 0.30\)
- Secondary Hashing? \(0.25 < \lambda < 0.5\)
- Chaining? \(3.0 < \lambda < 10\)
  - Because we allow multiple items in each space, we can increase memory efficiency by taking advantage
  - As long as there are a constant number in each space, we get \(O(1)\) runtimes.
LOAD FACTOR

• As with most array data structures, you will need to resize when they get too full
**LOAD FACTOR**

- As with most array data structures, you will need to resize when they get too full
  - Here, these resizes are often for performance, rather than failure.
LOAD FACTOR

• As with most array data structures, you will need to resize when they get too full
  • Here, these resizes are often for performance, rather than failure.
  • Hash table maintenance is important
LOAD FACTOR

• As with most array data structures, you will need to resize when they get too full
  • Here, these resizes are often for performance, rather than failure.
  • Hash table maintenance is important
  • Resizing is costly (but still O(n)) because you have to resize the array and rehash every element into the new table.
DELETION

• How to delete from a hash table?
DELETION

• How to delete from a hash table?
  • Chaining: just remove the object from the underlying data structure
DELETION

• How to delete from a hash table?
  • Chaining: just remove the object from the underlying data structure
  • Probing:
DELETION

• How to delete from a hash table?
  • Chaining: just remove the object from the underlying data structure
  • Probing: Must be able to follow the path in order to find elements that have been added later
DELETION

• How to delete from a hash table?
  • Chaining: just remove the object from the underlying data structure
  • Probing: Must be able to follow the path in order to find elements that have been added later
  • Need to mark as deleted, but not treat as completely empty
LAZY DELETION

- Common strategy in difficult-to-delete data structures
LAZY DELETION

• Common strategy in difficult-to-delete data structures
  • When you delete, mark the element as deleted, but maintain the data structure as-is
LAZY DELETION

• Common strategy in difficult-to-delete data structures
  • When you delete, mark the element as deleted, but maintain the data structure as-is
  • Works well for AVL as well
LAZY DELETION

• Common strategy in difficult-to-delete data structures
  • When you delete, mark the element as deleted, but maintain the data structure as-is
  • Works well for AVL as well
  • Can insert values into place if reinserted, just cannot return the associated value on a call to find
LAZY DELETION

• Common strategy in difficult-to-delete data structures
  • When you delete, mark the element as deleted, but maintain the data structure as-is
  • Works well for AVL as well
  • Can insert values into place if reinserted, just cannot return the associated value on a call to find
  • Necessary for Probing (aka Open Addressing) collision methods
CHAINING

• What about chaining? What is a good data structure to use?
CHAINING

• What about chaining? What is a good data structure to use?
  • Many implement with a simple linked list
**CHAINING**

- What about chaining? What is a good data structure to use?
  - Many implement with a simple linked list
  - If the load factor is \( \lambda \), what is the expected number of elements in a single bin?
CHAINING

• What about chaining? What is a good data structure to use?
  • Many implement with a simple linked list
  • If the load factor is $\lambda$, what is the expected number of elements in a single bin? $\lambda$
CHAINING

• What about chaining? What is a good data structure to use?
  • Many implement with a simple linked list
  • If the load factor is $\lambda$, what is the expected number of elements in a single bin? $\lambda$
  • However, the expected maximum actually grows (roughly) logarithmically with table length
CHAINING

• What about chaining? What is a good data structure to use?
  • Many implement with a simple linked list
  • If the load factor is $\lambda$, what is the expected number of elements in a single bin? $\lambda$
  • However, the expected maximum actually grows (roughly) logarithmically with table length
  • The more elements we add, the higher chance that there is one bad bin
CHAINING

• Solutions
  • Can perform resize when any bin reaches a certain size
CHAINING

• Solutions
  • Can perform resize when any bin reaches a certain size
    • Overallocates memory, if unlucky
CHAINING

• **Solutions**
  • Can perform resize when any bin reaches a certain size
    • Overallocates memory, if unlucky
    • Preserves $O(1)$ guarantee, however
CHAINING

• Solutions
  • Can perform resize when any bin reaches a certain size
    • Overallocates memory, if unlucky
    • Preserves O(1) guarantee, however
    • Downsizing is also difficult to calculate
CHAINING

• Solutions
  • Can perform resize when any bin reaches a certain size
    • Overallocates memory, if unlucky
    • Preserves $O(1)$ guarantee, however
    • Downsizing is also difficult to calculate
  • Make the underlying data structure more efficient
CHAINING

• Solutions
  • Can perform resize when any bin reaches a certain size
    • Overallocates memory, if unlucky
    • Preserves O(1) guarantee, however
    • Downsizing is also difficult to calculate
  • Make the underlying data structure more efficient
    • AVL is surprisingly common
CHAINING

• Solutions
  • Can perform resize when any bin reaches a certain size
    • Overallocates memory, if unlucky
    • Preserves O(1) guarantee, however
    • Downsizing is also difficult to calculate
  • Make the underlying data structure more efficient
    • AVL is surprisingly common
    • Hash table is also common
CHAINING

• Hash of hashes
CHAINING

• Hash of hashes
  • Suppose we want a collision with probability $1/N$
CHAINING

• Hash of hashes
  • Suppose we want a collision with probability 1/N
  • How big would our table need to be for open addressing?
CHAINING

- Hash of hashes
  - Suppose we want a collision with probability 1/N
  - How big would our table need to be for open addressing? $N^2$
CHAINING

• Hash of hashes
  • Suppose we want a collision with probability 1/N
  • How big would our table need to be for open addressing? $N^2$
  • What if we use a hashtable of hashtables
CHAINING

• Hash of hashes
  • Suppose we want a collision with probability $1/N$
  • How big would our table need to be for open addressing? $N^2$
  • What if we use a hashtable of hashtables
    • Let the first table size be $N$
CHAINING

• Hash of hashes
  • Suppose we want a collision with probability 1/N
  • How big would our table need to be for open addressing? $N^2$
  • What if we use a hashtable of hashtables
    • Let the first table size be N
    • Second tables are dynamically allocated (they will grow if they’re a heavy-hitter)
CHAINING

• Hash of hashes
  • Suppose we want a collision with probability 1/N
  • How big would our table need to be for open addressing? $N^2$
  • What if we use a hashtable of hashtables
    • Let the first table size be $N$
    • Second tables are dynamically allocated (they will grow if they’re a heavy-hitter)
    • If we still want 1/N collision probability, how large is the table?
CHAINING

• Hash of hashes
  • Suppose we want a collision with probability $1/N$
  • How big would our table need to be for open addressing? $N^2$
  • What if we use a hashtable of hashtables
    • Let the first table size be $N$
    • Second tables are dynamically allocated (they will grow if they’re a heavy-hitter)
    • If we still want $1/N$ collision probability, how large is the table? $N^2$ but $N$ is almost always a constant
CHAINING

• Hash of hashes
  • Suppose we want a collision with probability 1/N
  • How big would our table need to be for open addressing? \( N^2 \)
  • What if we use a hashtable of hashtables
    • Let the first table size be N
    • Second tables are dynamically allocated (they will grow if they’re a heavy-hitter)
    • If we still want 1/N collision probability, how large is the table? \( N^2 \) but N is almost always a constant
    • Some constant number have log n memory, but this is \( O(n) \) memory usage overall!