CSE 332

JULY 12TH – HASHING AND EXAM REVIEW

ADMINISTRIVIA

- **Exam review session**
  - CSE 403: Thursday 3:30 – 5:00

ADMINISTRIVIA

- **Exam review session**
  - CSE 403: Thursday 3:30 – 5:00
- **P2 is out**

ADMINISTRIVIA

- **Exam review session**
  - CSE 403: Thursday 3:30 – 5:00
- **P2 is out**
  - Checkpoint next Wednesday

ADMINISTRIVIA

- **Exam review session**
  - CSE 403: Thursday 3:30 – 5:00
- **P2 is out**
  - Checkpoint next Wednesday
  - Definitely have Ckpt1 passing

ADMINISTRIVIA

- **Exam review session**
  - CSE 403: Thursday 3:30 – 5:00
- **P2 is out**
  - Checkpoint next Wednesday
  - Definitely have Ckpt1 passing
  - Chpt2 is a reasonable goal

TODAY'S LECTURE

- Hashing

TODAY'S LECTURE

- **Hashing**
  - Double hashing

TODAY'S LECTURE

- **Hashing**
  - Double hashing
  - Conclusion

TODAY'S LECTURE

- **Hashing**
  - Double hashing
  - Conclusion
- **Exam Review**

TODAY'S LECTURE

- **Hashing**
  - Double hashing
  - Conclusion
- **Exam Review**
  - List of topics and things to know

HASHING

- **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

HASHING

- **Memory: The Hash Table**
  - Consider an array of size **c * 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.

HASH FUNCTIONS

- **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

- **You will not have to produce hash functions, but you should recognize good ones**
  - They run in constant time
  - They evenly distribute the data
  - They return an integer
- **These hash functions are chosen in advance, you should not pick a hash function relative to your data**

COLLISIONS

- **Hash table methods are defined by how they handle collisions**

COLLISIONS

- **Hash table methods are defined by how they handle collisions**
- **Two main approaches**
  - Probing
  - Chaining

COLLISIONS

- **Probing**
  - Linear probing

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

COLLISIONS

- **Probing**
  - Quadratic Probing

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
    - Need to be careful, possible for infinite loops with a very empty array

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

COLLISIONS

- Chaining

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

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

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 λ, 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 λ, what is the expected number of elements in a single bin? **λ**

CHAINING

- **What about chaining? What is a good data structure to use?**
  - Many implement with a simple linked list
  - If the load factor is λ, what is the expected number of elements in a single bin? **λ**
  - 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 λ, what is the expected number of elements in a single bin? **λ**
  - 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 λ, what is the expected number of elements in a single bin? **λ**
  - 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? $\mathbf{N^2}$
  - 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? $\mathbf{N^2}$
  - 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? $\mathbf{N^2}$
  - 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? $\mathbf{N^2}$
  - 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? $\mathbf{N^2}$ 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? $\mathbf{N^2}$
  - 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? $\mathbf{N^2}$ but N is almost always a constant
    - Some constant number have log n memory, but this is O(n) memory usage overall!

HASHTABLE IMPLEMENTATION

- Hashtables implement dictionaries

HASHTABLE IMPLEMENTATION

- **Hashtables implement dictionaries**
  - <Key, Value> pairs

HASHTABLE IMPLEMENTATION

- **Hashtables implement dictionaries**
  - <Key, Value> pairs
  - Don’t allow duplicate keys

HASHTABLE IMPLEMENTATION

- **Hashtables implement dictionaries**
  - <Key, Value> pairs
  - Don’t allow duplicate keys
  - Keys with the same “value” must have the same hash code

HASHTABLE IMPLEMENTATION

- **Hashtables implement dictionaries**
  - <Key, Value> pairs
  - Don’t allow duplicate keys
  - Keys with the same “value” must have the same hash code
  - For open addressing, stored either as an array of <key,value> class objects, or as two parallel arrays, one of keys and the other of values

HASHTABLE IMPLEMENTATION

- Resizing

HASHTABLE IMPLEMENTATION

- **Resizing**
  - Only get O(1) operations if the table is well-maintained

HASHTABLE IMPLEMENTATION

- **Resizing**
  - Only get O(1) operations if the table is well-maintained
  - Easy to get good runtimes, if you don't consider memory

HASHTABLE IMPLEMENTATION

- **Resizing**
  - Only get O(1) operations if the table is well-maintained
  - Easy to get good runtimes, if you don't consider memory
  - bigO analysis can apply to memory consumption in the same way it applies to clock cycles

HASHTABLE IMPLEMENTATION

- **Resizing**
  - Only get O(1) operations if the table is well-maintained
  - Easy to get good runtimes, if you don't consider memory
  - bigO analysis can apply to memory consumption in the same way it applies to clock cycles
  - Resizing takes O(n) extra memory, because you need to maintain the original hash table while you build the second.

HASHTABLE IMPLEMENTATION

- **Resizing**
  - Iterate through the table (these are not in any meaningful order)

HASHTABLE IMPLEMENTATION

- **Resizing**
  - Iterate through the table (these are not in any meaningful order)
  - Insert each of the <k,v> pairs into the new hashtable (which may be larger or smaller)
  - Move pointers to new hash table

HASHTABLE IMPLEMENTATION

- Assorted things

HASHTABLE IMPLEMENTATION

- **Assorted things**
  - Prime table sizes
    - Usually keep an array of all the primes that roughly double in size precalculated

HASHTABLE IMPLEMENTATION

- **Assorted things**
  - Prime table sizes
    - Usually keep an array of all the primes that roughly double in size precalculated
    - Finding primes is actually very computationally difficult,

HASHTABLE IMPLEMENTATION

- **Assorted things**
  - Prime table sizes
    - Usually keep an array of all the primes that roughly double in size precalculated
    - Finding primes is actually very computationally difficult,
    - If n is large enough, finding the new prime can be the most consuming portion of the resize

HASHTABLE IMPLEMENTATION

- **Assorted things**
  - Prime table sizes
    - Usually keep an array of all the primes that roughly double in size precalculated
    - Finding primes is actually very computationally difficult,
    - If n is large enough, finding the new prime can be the most consuming portion of the resize
  - If a.equals(b) then a.hashcode() == b.hashcode()

HASHTABLE IMPLEMENTATION

- **Assorted things**
  - Prime table sizes
    - Usually keep an array of all the primes that roughly double in size precalculated
    - Finding primes is actually very computationally difficult,
    - If n is large enough, finding the new prime can be the most consuming portion of the resize
  - If a.equals(b) then a.hashcode() == b.hashcode()
  - Hardware constraints, even if you have lots of memory, over allocating fails to take advantage of spatial locality and can be problematic

HASHTABLE TAKEAWAYS

- Provide constant time find(k), insert(k,v) and delete(k) provided the structure is well maintained

HASHTABLE TAKEAWAYS

- Provide constant time find(k), insert(k,v) and delete(k) provided the structure is well maintained
- Load factor is the primary determinant of runtime

HASHTABLE TAKEAWAYS

- Provide constant time find(k), insert(k,v) and delete(k) provided the structure is well maintained
- Load factor is the primary determinant of runtime
- Two approaches, probing v. chaining

HASHTABLE TAKEAWAYS

- Provide constant time find(k), insert(k,v) and delete(k) provided the structure is well maintained
- Load factor is the primary determinant of runtime
- Two approaches, probing v. chaining
- Primary and Secondary clustering

HASHTABLE TAKEAWAYS

- Provide constant time find(k), insert(k,v) and delete(k) provided the structure is well maintained
- Load factor is the primary determinant of runtime
- Two approaches, probing v. chaining
- Primary and Secondary clustering
- Which chaining data structure do you use?

HASHTABLE TAKEAWAYS

- Provide constant time find(k), insert(k,v) and delete(k) provided the structure is well maintained
- Load factor is the primary determinant of runtime
- Two approaches, probing v. chaining
- Primary and Secondary clustering
- Which chaining data structure do you use?
- Easy interview question answer, just be ready to explain how your data structure reacts to memory constraints

EXAM FRIDAY

- **Topics**
  - Definitions
  - Stacks and Queues
  - Heaps
  - Runtime Analysis
  - Dictionaries
  - BSTs
  - B-Trees
  - AVL Trees
  - Hash Tables
  - Tries

DEFINITIONS

- **Important terms**

DEFINITIONS

- **Important terms**
  - Abstract Data Type

DEFINITIONS

- **Important terms**
  - Abstract Data Type
    - Example: Dictionary

DEFINITIONS

- **Important terms**
  - Abstract Data Type
    - Example: Dictionary
      - Supports functions: insert, find, delete
      - Has expected behavior

DEFINITIONS

- **Important terms**
  - Abstract Data Type
    - Example: Dictionary
      - Supports functions: insert, find, delete
      - Has expected behavior
  - Data Structure

DEFINITIONS

- **Important terms**
  - Abstract Data Type
    - Example: Dictionary
      - Supports functions: insert, find, delete
      - Has expected behavior
  - Data Structure
    - Language independent structure which implements an ADT

DEFINITIONS

- **Important terms**
  - Abstract Data Type
    - Example: Dictionary
      - Supports functions: insert, find, delete
      - Has expected behavior
  - Data Structure
    - Language independent structure which implements an ADT
      - Example: AVL tree

DEFINITIONS

- **Important terms**
  - Abstract Data Type
    - Example: Dictionary
      - Supports functions: insert, find, delete
      - Has expected behavior
  - Data Structure
    - Language independent structure which implements an ADT
      - Example: AVL tree
      - Can be analyzed asymptotically

DEFINITIONS

- **Important terms**
  - Implementation
    - Low-level design decisions

DEFINITIONS

- **Important terms**
  - Implementation
    - Low-level design decisions
    - Language specific

DEFINITIONS

- **Important terms**
  - Implementation
    - Low-level design decisions
    - Language specific
- **Example**

DEFINITIONS

- **Important terms**
  - Implementation
    - Low-level design decisions
    - Language specific
- **Example**
  - The Queue ADT supports enqueue, dequeue and front.

DEFINITIONS

- **Important terms**
  - Implementation
    - Low-level design decisions
    - Language specific
- **Example**
  - The Queue ADT supports enqueue, dequeue and front.
  - Arrays and Linked Lists are examples of the data structures

DEFINITIONS

- **Important terms**
  - Implementation
    - Low-level design decisions
    - Language specific
- **Example**
  - The Queue ADT supports enqueue, dequeue and front.
  - Arrays and Linked Lists are examples of the data structures
  - Implementation: front and back pointers

STACKS AND QUEUES

- Our first two ADTs

STACKS AND QUEUES

- **Our first two ADTs**
  - Stack:

STACKS AND QUEUES

- **Our first two ADTs**
  - Stack:
    - Supports: push(), pop(), top()

STACKS AND QUEUES

- **Our first two ADTs**
  - Stack:
    - Supports: push(), pop(), top()
    - LIFO order

STACKS AND QUEUES

- **Our first two ADTs**
  - Stack:
    - Supports: push(), pop(), top()
    - LIFO order
  - Queue:

STACKS AND QUEUES

- **Our first two ADTs**
  - Stack:
    - Supports: push(), pop(), top()
    - LIFO order
  - Queue:
    - Supports: enqueue(), dequeue(), front()

STACKS AND QUEUES

- **Our first two ADTs**
  - Stack:
    - Supports: push(), pop(), top()
    - LIFO order
  - Queue:
    - Supports: enqueue(), dequeue(), front()
    - FIFO order

STACKS AND QUEUES

- **Data structure choices**

STACKS AND QUEUES

- **Data structure choices**
  - Arrays and Linked Lists

STACKS AND QUEUES

- **Data structure choices**
  - Arrays and Linked Lists
  - Considerations

STACKS AND QUEUES

- **Data structure choices**
  - Arrays and Linked Lists
  - Considerations
    - Memory usage

STACKS AND QUEUES

- **Data structure choices**
  - Arrays and Linked Lists
  - Considerations
    - Memory usage
    - Ease of implementation

STACKS AND QUEUES

- **Data structure choices**
  - Arrays and Linked Lists
  - Considerations
    - Memory usage
    - Ease of implementation
    - Resizing time

STACKS AND QUEUES

- **Data structure choices**
  - Arrays and Linked Lists
  - Considerations
    - Memory usage
    - Ease of implementation
    - Resizing time
  - Runtimes:

STACKS AND QUEUES

- **Data structure choices**
  - Arrays and Linked Lists
  - Considerations
    - Memory usage
    - Ease of implementation
    - Resizing time
  - Runtimes:
    - O(1) for all functions

HEAPS

- Priority Queue ADT

HEAPS

- **Priority Queue ADT**
  - Supports: insert(), findMin(), deleteMin(), changePriority()

HEAPS

- **Priority Queue ADT**
  - Supports: insert(), findMin(), deleteMin(), changePriority()
  - Data is stored in priority, value pairs

HEAPS

- **Priority Queue ADT**
  - Supports: insert(), findMin(), deleteMin(), changePriority()
  - Data is stored in priority, value pairs
  - In this class, we use the min-heap, where a lower value means it should dequeue first

HEAPS

- **Data Structure**
  - Heap

HEAPS

- **Data Structure**
  - Heap
    - Complete binary tree

HEAPS

- **Data Structure**
  - Heap
    - Complete binary tree
    - Heap property

HEAPS

- **Data Structure**
  - Heap
    - Complete binary tree
    - Heap property
  - Implementation

HEAPS

- **Data Structure**
  - Heap
    - Complete binary tree
    - Heap property
  - Implementation
    - Array

HEAPS

- **Data Structure**
  - Heap
    - Complete binary tree
    - Heap property
  - Implementation
    - Array
    - Find parents/children arithmetically

HEAPS

- **Data Structure**
  - Heap
    - Complete binary tree
    - Heap property
  - Implementation
    - Array
    - Find parents/children arithmetically
  - Runtimes

HEAPS

- **Data Structure**
  - Heap
    - Complete binary tree
    - Heap property
  - Implementation
    - Array
    - Find parents/children arithmetically
  - Runtimes
    - Insert: O(log n), findMin: O(1), deleteMin O(log n)
    - ChangePriority: O(log n)

HEAPS

- **Data Structure**
  - Heap
    - Complete binary tree
    - Heap property
  - Implementation
    - Array
    - Find parents/children arithmetically
  - Runtimes
    - Insert: O(log n), findMin: O(1), deleteMin O(log n)
    - ChangePriority: O(log n)
    - buildHeap, O(n)

RUNTIME ANALYSIS

- Counting the number of operations

RUNTIME ANALYSIS

- **Counting the number of operations**
  - Comparisons, mathematical operations, assignments

RUNTIME ANALYSIS

- **Counting the number of operations**
  - Comparisons, mathematical operations, assignments
- **For loops and while statements**

RUNTIME ANALYSIS

- **Counting the number of operations**
  - Comparisons, mathematical operations, assignments
- **For loops and while statements**
  - Count the number of times relevant code is executed

RUNTIME ANALYSIS

- **Counting the number of operations**
  - Comparisons, mathematical operations, assignments
- **For loops and while statements**
  - Count the number of times relevant code is executed
- **Important summations**

RUNTIME ANALYSIS

- **Counting the number of operations**
  - Comparisons, mathematical operations, assignments
- **For loops and while statements**
  - Count the number of times relevant code is executed
- **Important summations**
  - Sum of all numbers from 1 to n

RUNTIME ANALYSIS

- **Counting the number of operations**
  - Comparisons, mathematical operations, assignments
- **For loops and while statements**
  - Count the number of times relevant code is executed
- **Important summations**
  - Sum of all numbers from 1 to n
  - Sum of the powers of two

RUNTIME ANALYSIS

- **Asymptotic Analysis**

RUNTIME ANALYSIS

- **Asymptotic Analysis**
  - Best-case, worst-case, average-case

RUNTIME ANALYSIS

- **Asymptotic Analysis**
  - Best-case, worst-case, average-case
  - Usually we discuss worst-case complexity

RUNTIME ANALYSIS

- **Asymptotic Analysis**
  - Best-case, worst-case, average-case
  - Usually we discuss worst-case complexity
  - If we increase the input size, how does the computation time change

RUNTIME ANALYSIS

- **Asymptotic Analysis**
  - Best-case, worst-case, average-case
  - Usually we discuss worst-case complexity
  - If we increase the input size, how does the computation time change
- **BigO notation**

RUNTIME ANALYSIS

- **Asymptotic Analysis**
  - Best-case, worst-case, average-case
  - Usually we discuss worst-case complexity
  - If we increase the input size, how does the computation time change
- **BigO notation**
  - Upper bound for a given function

RUNTIME ANALYSIS

- **Asymptotic Analysis**
  - Best-case, worst-case, average-case
  - Usually we discuss worst-case complexity
  - If we increase the input size, how does the computation time change
- **BigO notation**
  - Upper bound for a given function
  - $f(n) = O(g(n)$ if there exists a c and n_0 for which $f(n) \leq c*g(n)$ for all $n \geq n_0$

RUNTIME ANALYSIS

- **Recurrences**

RUNTIME ANALYSIS

- **Recurrences**
  - Way in which we approach recursive functions

RUNTIME ANALYSIS

- **Recurrences**
  - Way in which we approach recursive functions
  - Separate into recursive and non-recursive

RUNTIME ANALYSIS

- **Recurrences**
  - Way in which we approach recursive functions
  - Separate into recursive and non-recursive
  - Calculate the runtimes for non-recursive and base cases

RUNTIME ANALYSIS

- **Recurrences**
  - Way in which we approach recursive functions
  - Separate into recursive and non-recursive
  - Calculate the runtimes for non-recursive and base cases
  - Produce the recurrence

RUNTIME ANALYSIS

- **Recurrences**
  - Way in which we approach recursive functions
  - Separate into recursive and non-recursive
  - Calculate the runtimes for non-recursive and base cases
  - Produce the recurrence
  - Solve the recurrence by rolling out, using a graphical tree or using the master theorem

RUNTIME ANALYSIS

- **Recurrences**
  - Way in which we approach recursive functions
  - Separate into recursive and non-recursive
  - Calculate the runtimes for non-recursive and base cases
  - Produce the recurrence
  - Solve the recurrence by rolling out, using a graphical tree or using the master theorem
  - Provide the bigO asymptotic bounds

RUNTIME ANALYSIS

- **Amortized analysis**

RUNTIME ANALYSIS

- **Amortized analysis**
  - When computations come at predictable times but are very expensive

RUNTIME ANALYSIS

- **Amortized analysis**
  - When computations come at predictable times but are very expensive
  - The amortized runtime is the time a method takes to run n consecutive operations divided by n.

RUNTIME ANALYSIS

- **Amortized analysis**
  - When computations come at predictable times but are very expensive
  - The amortized runtime is the time a method takes to run n consecutive operations divided by n.
  - This is different than best-case/worst-case

RUNTIME ANALYSIS

- **Amortized analysis**
  - When computations come at predictable times but are very expensive
  - The amortized runtime is the time a method takes to run n consecutive operations divided by n.
  - This is different than best-case/worst-case
  - Array resizing was the prominent example

RUNTIME ANALYSIS

- **Basic ideas**
  - O(1): Input size has no effect on runtime

RUNTIME ANALYSIS

- **Basic ideas**
  - O(1): Input size has no effect on runtime
  - O(log n): doubling the input increases the runtime by some constant amount

RUNTIME ANALYSIS

- **Basic ideas**
  - O(1): Input size has no effect on runtime
  - O(log n): doubling the input increases the runtime by some constant amount
  - O(n): linear time, each additional input increases execution time by a constant amount

RUNTIME ANALYSIS

- **Basic ideas**
  - $O(1)$: Input size has no effect on runtime
  - $O(\log n)$: doubling the input increases the runtime by some constant amount
  - $O(n)$: linear time, each additional input increases execution time by a constant amount
  - $O(n^2)$: doubling the input increases the runtime by a factor of 4.

RUNTIME ANALYSIS

- **Basic ideas**
  - $O(1)$: Input size has no effect on runtime
  - $O(\log n)$: doubling the input increases the runtime by some constant amount
  - $O(n)$: linear time, each additional input increases execution time by a constant amount
  - $O(n^2)$: doubling the input increases the runtime by a factor of 4.
  - $O(2^n)$: exponential, increasing the input by one doublies the runtime

DICTIONARIES

- ADT

DICTIONARIES

- **ADT**
  - Supports the following functions

DICTIONARIES

- **ADT**
  - Supports the following functions
    - Insert(key k, value v)

DICTIONARIES

- **ADT**
  - Supports the following functions
    - Insert(key k, value v)
    - find(key k)
    - delete(key k)

DICTIONARIES

- **ADT**
  - Supports the following functions
    - Insert(key k, value v)
    - find(key k)
    - delete(key k)
  - Data is stored in key, value pairs

DICTIONARIES

- **ADT**
  - Supports the following functions
    - Insert(key k, value v)
    - find(key k)
    - delete(key k)
  - Data is stored in key, value pairs
  - In this course, duplicate keys are not allowed

DICTIONARIES

- **ADT**
  - Supports the following functions
    - Insert(key k, value v)
    - find(key k)
    - delete(key k)
  - Data is stored in key, value pairs
  - In this course, duplicate keys are not allowed
  - Most data structures can implement a dictionary

BINARY SEARCH TREES

- Binary trees

BINARY SEARCH TREES

- Binary trees
- Nodes with two children

BINARY SEARCH TREES

- Binary trees
- Nodes with two children
- Maintains search property

BINARY SEARCH TREES

- **Binary trees**
- **Nodes with two children**
- **Maintains search property**
  - All values in the left subtree must be less than the parent
  - All values in the right subtree must be greater than the parent

BINARY SEARCH TREES

- **Binary trees**
- **Nodes with two children**
- **Maintains search property**
  - All values in the left subtree must be less than the parent
  - All values in the right subtree must be greater than the parent
- **With each increase in height, the number of nodes in a tree roughly doubles**

BINARY SEARCH TREES

- **Binary trees**
- **Nodes with two children**
- **Maintains search property**
  - All values in the left subtree must be less than the parent
  - All values in the right subtree must be greater than the parent
- **With each increase in height, the number of nodes in a tree roughly doubles**
- **A completely full tree has 2^h-1 nodes**

BINARY SEARCH TREES

- **Binary trees**
- **Nodes with two children**
- **Maintains search property**
  - All values in the left subtree must be less than the parent
  - All values in the right subtree must be greater than the parent
- **With each increase in height, the number of nodes in a tree roughly doubles**
- **A completely full tree has 2^h-1 nodes**
- **Roughly half of a binary search tree are nodes**

AVL TREES

- Specific type of binary search tree

AVL TREES

- Specific type of binary search tree
- Still must implement binary search

AVL TREES

- Specific type of binary search tree
- Still must implement binary search
- Nodes in AVL trees have two extra fields, height and balance

AVL TREES

- Specific type of binary search tree
- Still must implement binary search
- Nodes in AVL trees have two extra fields, height and balance
- Balance = | height(left) – height(right) |

AVL TREES

- Specific type of binary search tree
- Still must implement binary search
- Nodes in AVL trees have two extra fields, height and balance
- Balance = | height(left) – height(right) |
- Balance for each node must be less than or equal to 1

AVL TREES

- Specific type of binary search tree
- Still must implement binary search
- Nodes in AVL trees have two extra fields, height and balance
- Balance = | height(left) – height(right) |
- Balance for each node must be less than or equal to 1
- Trees with this condition still have O(log n) height

AVL TREES

- Specific type of binary search tree
- Still must implement binary search
- Nodes in AVL trees have two extra fields, height and balance
- Balance = | height(left) – height(right) |
- Balance for each node must be less than or equal to 1
- Trees with this condition still have O(log n) height
- No covering delete in this course

AVL TREES

- Specific type of binary search tree
- Still must implement binary search
- Nodes in AVL trees have two extra fields, height and balance
- Balance = | height(left) – height(right) |
- Balance for each node must be less than or equal to 1
- Trees with this condition still have O(log n) height
- No covering delete in this course
- Find: O(log n): Insert O(log n)

AVL ROTATIONS

- **AVL Rotations occur when an insertion makes a node out of balance**

AVL ROTATIONS

- **AVL Rotations occur when an insertion makes a node out of balance**
  - Relative to the node that is unbalanced, there are four rotations depending on which grandchild received the new node.

AVL ROTATIONS

- **AVL Rotations occur when an insertion makes a node out of balance**
    - Relative to the node that is unbalanced, there are four rotations depending on which grandchild received the new node.
    - Left-left and right right rotations involve the child of the affected node being rotated up into position

AVL ROTATIONS

- **AVL Rotations occur when an insertion makes a node out of balance**
  - Relative to the node that is unbalanced, there are four rotations depending on which grandchild received the new node.
  - Left-left and right right rotations involve the child of the affected node being rotated up into position
  - Left-right and right-left rotations involve the grandchild being rotated up into position. The grandparent and parent become the two children

AVL ROTATIONS

- **AVL Rotations occur when an insertion makes a node out of balance**
  - Relative to the node that is unbalanced, there are four rotations depending on which grandchild received the new node.
  - Left-left and right right rotations involve the child of the affected node being rotated up into position
  - Left-right and right-left rotations involve the grandchild being rotated up into position. The grandparent and parent become the two children
  - It is important that these rotations preserve BST property

B-PLUS TREES

- **Memory is not the equal access object that traditional theory discusses**

B-PLUS TREES

- **Memory is not the equal access object that traditional theory discusses**
  - Memory is broken up into pages

B-PLUS TREES

- **Memory is not the equal access object that traditional theory discusses**
  - Memory is broken up into pages
  - Some pages are on disk, others are in cache

B-PLUS TREES

- **Memory is not the equal access object that traditional theory discusses**
  - Memory is broken up into pages
  - Some pages are on disk, others are in cache
  - Need a data structure to minimize disk accesses

B-PLUS TREES

- **Memory is not the equal access object that traditional theory discusses**
  - Memory is broken up into pages
  - Some pages are on disk, others are in cache
  - Need a data structure to minimize disk accesses
- **Data structure**

B-PLUS TREES

- **Memory is not the equal access object that traditional theory discusses**
  - Memory is broken up into pages
  - Some pages are on disk, others are in cache
  - Need a data structure to minimize disk accesses
- **Data structure**
  - Two types of nodes, signposts and leaves

B-PLUS TREES

- **Memory is not the equal access object that traditional theory discusses**
  - Memory is broken up into pages
  - Some pages are on disk, others are in cache
  - Need a data structure to minimize disk accesses
- **Data structure**
  - Two types of nodes, signposts and leaves
  - Signposts have between M/2 and M children, where M makes the signpost object as large as possible while still fitting in one page

B-PLUS TREES

- **Memory is not the equal access object that traditional theory discusses**
  - Memory is broken up into pages
  - Some pages are on disk, others are in cache
  - Need a data structure to minimize disk accesses
- **Data structure**
  - Two types of nodes, signposts and leaves
  - Signposts have between M/2 and M children, where M makes the signpost object as large as possible while still fitting in one page
  - Leaves have between L/2 and L pieces of sorted data and a pointer to the next leaf

B-PLUS TREES

- **Memory is not the equal access object that traditional theory discusses**
  - Memory is broken up into pages
  - Some pages are on disk, others are in cache
  - Need a data structure to minimize disk accesses
- **Data structure**
  - Two types of nodes, signposts and leaves
  - Signposts have between M/2 and M children, where M makes the signpost object as large as possible while still fitting in one page
  - Leaves have between L/2 and L pieces of sorted data and a pointer to the next leaf
  - Root is exempt from minimums

B-PLUS TREES

- Inserting

B-PLUS TREES

- **Inserting**
  - Add in sorted order

B-PLUS TREES

- **Inserting**
  - Add in sorted order
  - If you fail, break the leaf into two

B-PLUS TREES

- **Inserting**
  - Add in sorted order
  - If you fail, break the leaf into two
  - If the signpost cannot fit another node, recursively try to add nodes back up to the root until a signpost has room

B-PLUS TREES

- **Inserting**
  - Add in sorted order
  - If you fail, break the leaf into two
  - If the signpost cannot fit another node, recursively try to add nodes back up to the root until a signpost has room
- **Find**

B-PLUS TREES

- **Inserting**
  - Add in sorted order
  - If you fail, break the leaf into two
  - If the signpost cannot fit another node, recursively try to add nodes back up to the root until a signpost has room
- **Find**
  - Signposts indicate where key,value pairs are by markers in their node, a child is between two values

B-PLUS TREES

- **Inserting**
  - Add in sorted order
  - If you fail, break the leaf into two
  - If the signpost cannot fit another node, recursively try to add nodes back up to the root until a signpost has room
- **Find**
  - Signposts indicate where key,value pairs are by markers in their node, a child is between two values
  - Traverse down the tree to the bottom

B-PLUS TREES

- Delete

B-PLUS TREES

- **Delete**
  - If a deletion causes a leaf to go less than L/2 in size

B-PLUS TREES

- **Delete**
  - If a deletion causes a leaf to go less than L/2 in size
  - Try to adopt if we can (changing signposts if necessary)

B-PLUS TREES

- **Delete**
  - If a deletion causes a leaf to go less than L/2 in size
  - Try to adopt if we can (changing signposts if necessary)
  - If not, merge leaves together

B-PLUS TREES

- **Delete**
  - If a deletion causes a leaf to go less than L/2 in size
  - Try to adopt if we can (changing signposts if necessary)
  - If not, merge leaves together
  - Recursively merge signposts together as necessary in the path back to the root

B-PLUS TREES

- **Delete**
  - If a deletion causes a leaf to go less than L/2 in size
  - Try to adopt if we can (changing signposts if necessary)
  - If not, merge leaves together
  - Recursively merge signposts together as necessary in the path back to the root
- **Gives us the most use out of a single disk access**

B-PLUS TREES

- **Delete**
  - If a deletion causes a leaf to go less than L/2 in size
  - Try to adopt if we can (changing signposts if necessary)
  - If not, merge leaves together
  - Recursively merge signposts together as necessary in the path back to the root
- **Gives us the most use out of a single disk access**
- **Commonly used for databases because it allows good disk storage and easy retrieval of keys in a range**

HASH TABLES

- A large data set M with a smaller set that should be saved, D

HASH TABLES

- A large data set M with a smaller set that should be saved, D
- A hash function maps M onto D

HASH TABLES

- **A large data set M with a smaller set that should be saved, D**
- **A hash function maps M onto D**
  - It should run in O(1) time

HASH TABLES

- **A large data set M with a smaller set that should be saved, D**
- **A hash function maps M onto D**
    - It should run in O(1) time
    - It should distribute into all of the available spots evenly
- **Hashtables provide O(1) runtime IF**

HASH TABLES

- **A large data set M with a smaller set that should be saved, D**
- **A hash function maps M onto D**
  - It should run in O(1) time
  - It should distribute into all of the available spots evenly
- **Hashtables provide O(1) runtime IF**
  - Collisions are not a problem
  - Decrease the chance of collisions by increasing the amount of memory

HASH TABLES

- **A large data set M with a smaller set that should be saved, D**
- **A hash function maps M onto D**
  - It should run in O(1) time
  - It should distribute into all of the available spots evenly
- **Hashtables provide O(1) runtime IF**
  - Collisions are not a problem
  - Decrease the chance of collisions by increasing the amount of memory
    - Resizing is costly

DESIGN DECISION PROBLEM

- Think about runtime

DESIGN DECISION PROBLEM

- Think about runtime
- Memory constraints

DESIGN DECISION PROBLEM

- Think about runtime
- Memory constraints
- Function prioritizing

DESIGN DECISION PROBLEM

- Think about runtime
- Memory constraints
- Function prioritizing
- Experimental considerations

NEXT CLASS

- Exam!