# CS 209
# Data Structures and Mathematical Foundations

04 / 10 / 2024

Instructor: Michael Eckmann

# Today's Topics

- Questions?/Comments?
- Quicksort
  - speedup comments
  - Analysis
- Hash Tables

# Quicksort

- A typical speedup for Quicksort is to do the following:
  - when we get down to some small number of elements (say 10) in our list, instead of using quicksort on them, we do insertion sort.

- How would we alter the code we just wrote to do insertion sort when the number of elements to sort is small?

# Quicksort

- Running time and space analysis
- Worst case running time
  - Happens when?
- Best case running time
  - Happens when?

# Quicksort

- Analysis
- Worst case
  - Happens when list is divided into extremely unequal sizes each time
  - That is, when the pivot is less (or greater) than all the other elements
  - List size is divided into sizes n-1 and 0
- Best case
  - Happens when list is divided in as close to half as possible each time (close to be Mergesort's divide)

# Quicksort

- Analysis
- Worst case
  - Happens when list is divided into extremely unequal sizes each time
  - That is, when the pivot is less (or greater) than all the other elements
  - List size is divided into sizes n-1 and 0
  - Running time is $n^2$
- Best case
  - Happens when list is divided in as close to half as possible each time (close to be Mergesort's divide)
  - Running time is n * lg(n)
- All cases space is O(1)

# Hashing

- Hashing is used to allow very efficient insertion (add), removal (delete), and retrieval (search) of items.
- Consider retrieval (searching) with several structures
  - To find data in an unordered linear list structure
    - O(n)
  - To find data in an ordered linear array
    - O(log n)
  - To find data in a shortest-height BST
    - O(log n)
- What orders are better than log n ?

# Hashing

- Hashing is used to allow
  - inserting an item
  - removing an item
  - searching for an item

  all in constant time (in the average case).

- Hashing does not provide efficient sorting nor efficient finding of the minimum or maximum item etc.

# Hashing

- We want to handle data of any type (e.g. String, int, float, Employee records, etc.)
- Terms:
  - Hash Table (a list of references to objects(items))
    - table_size is the number of **possible** places to store data
  - Hash Function (calculates a hash value (an integer) based on some **key** data about the item we are adding to the hash table.)
  - Hash Value (the value returned by the hash function)
    - the hash value must be an integer value within the range: 0 to table_size – 1, inclusive
    - The Hash Value is then used as the index to hash table.

# Hashing

- Simple example of how to insert and retrieve items into a hash table (this does not use a good hash function)
  - Consider our items are simply integers
  - Consider our Hash Function to be f(x) = x % n
  - The hash function returns a hash value which is modded by the size of our hash table list to compute the index which is where we will store our item.
  - example on the board (assume n=8, add items 24, 3, 17, 31)
- To search for a particular item in our hash table, we first compute the hash value and look there ...

# Hashing

- In our example (assume n=8, add items 24, 3, 17, 31), what if we needed to insert item 11 into our hash?
- There'd be a collision.
- There are several strategies to handle collisions
  - Assume the hash value computed was H
  - the chosen strategy effects how retrieval is handled too
  - Open Addressing (aka Probing)
    - Place item in next open slot (linear probing)
      - H+1, or H+2 or H+3 ...
    - Place item in next open slot (quadratic probing)
      - $H+1^2$, or $H+2^2$, or $H+3^2$, or $H+4^2$, ...
  - Wraparound is allowed / required

# Hashing

- There are several strategies to handle collisions
  - Another technique besides Open Addressing, is Separate chaining
    - Each list element stores a reference to a linked list
- Examples of these techniques on the board.