NJIT CS114 Final Review

Tuesday, April 30, 2024   2:53 PM

By Arsh Bhamla

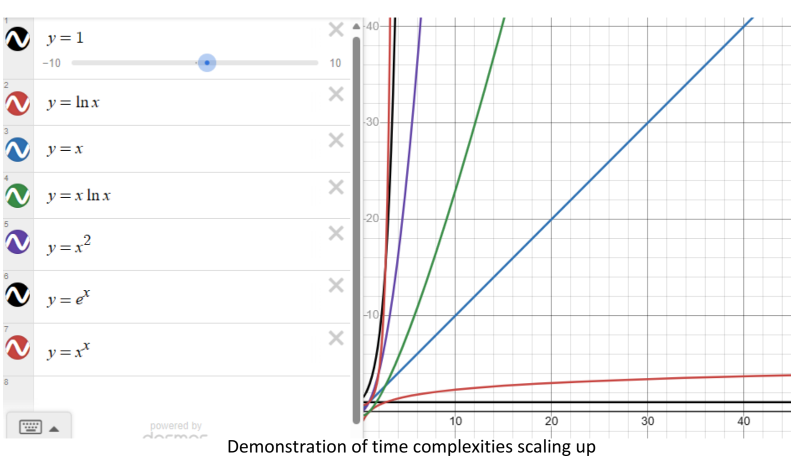
Introduction
- Every data structure runs off of tradeoffs between cost and benefits.
- Analyze a problem, and pick a data structure which fits.
- Abstract Data Type (ADT)
  - Talks about data and operations on those data, but doesn't specify how the operations are done.
    - Just look at .equals(), we know that it checks if they're equal, but we don't know how the method actually runs.
  - ADTs use Interfaces which show usable methods, but not the stuff inside.

Recursion
- Recursion is calling the same function again and again with changing parameters to get an answer.
- Any recursive function must have:
  - A base case - At least one statement which would end the recursion
  - Returning itself
    - You need to cut the problem size into smaller and smaller bits until it could hit the base case.
- Recursive functions use a runtime stack for recursion.
  - Any new recursive call is added to something akin to a stack, and when you reach the base case, it goes backwards until the first function.
  - Without a base case, this will go on forever (StackOverflowError)
- Example: Backtrack Search in Maze
  - Remember our maze problem, we started from (0,0) and had to go to (5,5).
  - We can recursively break down this problem from (x,y) to (5,5), and the value of x,y keeps changing.
  - The base case would be that we actually reach (5,5), or that we go out of bounds and have to terminate that path.
  - In this way, in our lab and homework we were able to find the route by recursively calling this method going in every single direction.
- Special Types of Recursion
  - Backtracking Search
    - We go down one path until we reach the goal/dead-end.
    - If we are at a dead-end, we backtrack one level, and see possibilities.
  - Greedy Method
    - A specific approach to recursion where we commit to decisions.
    - When looking to optimize for example, once you agree to use a piece of data, you can't later not use it.

Math Background
- Boolean Logic uses TRUE and FALSE for values.
- You should know how to use AND, OR, XOR, and NOT. (Not for the exam though)
- Induction
  - A process where we prove a property of P(x)
  - Start by making a basis step where we prove P(a1)
  - Once that's proved, or given, we can use P(a_n+1), which proves all other values of P.
- Postfix and Infix Notation
  - Methods of how we write out math
  - Infix: Infix is the traditional way of how to do math. For example, we have 1 + 1, which has the operator in between the two values that are being added.
  - Postfix is different in that the operator comes after. The operation is 1 1 +, and we see that the plus sign comes after the two numbers.
    - We can use a stack to solve for postfix notation, where you iterate through an equation, leaving any numbers where they are but pushing operands (+,-,*,/) into a stack.
      - If a higher priority item comes, add it to the stack, and if not, push the current item from the stack.
      - ** is used for powers, so 2**2 is 2^2 = 4.

Postfix Expression	Infix Expression
4 7 *	4 * 7
4 7 2 + *	4 * (7 + 2)
4 7 * 20 -	(4 * 7) - 20
3 4 7 * 2 / +	3 + ((4 * 7) / 2)

Algorithm Complexity
- Used to determine how complex an algorithm is in terms of space (like RAM) and time.
- Big Oh Complexity O(n)
  - The worst case scenario
- Big Omega Complexity Ω(n)
  - Best case scenario
  - This constitutes the lower bound of time complexity
- Big Theta Complexity Θ(n)
  - This is a "tight bound"
  - Basically, it's the middle value of the two
- Amortized Time Complexity
  - The average time required to complete a program.
  - We often has algorithms with expensive worst cases, so we take the average.
  - For example, in an insertion function, we might need to expand an array, which is expensive, but normally inserting a value is O(1).
    - The amortized time complexity here is O(1), even though the worst case is O(n^2).
- Asymptotic Time Complexity
  - We normally prefer asymptotic time complexity since it scales good. O(logn) is asymptotic.
  - The order of hierarchy is as follows:
    - O(1)
      - This is the best case scenario, it's a constant value and only runs once
    - O(logn)
    - O(n^m)
      - The value of m can be anything, but it's still more than logn
    - O(a^n)
    - O(n!)
    - O(n^n)
      - This is the worst complexity to scale
  - We could use limits and then L'Hopitals rule to solve for this.
  - Any combination of the hierarchy works in much the same way, nlogn is higher than n and logn but not higher than n^2.



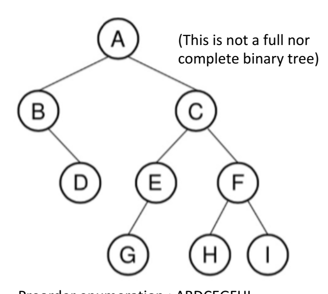
Demonstration of time complexities scaling up

Java Collections API
- Framework
  - Java has a framework (Java.util.*) which has a bunch of interfaces and classes.
    - This includes algorithms for manipulating data such as filtering, searching, sorting and aggregation.
  - Has 2 Interface Hierarchies:
    1. Collection
       1) Collection has Set, List, and Queue
    2. Map
- Java Generics
  - Basically default settings for Java.
  - Arrays.sort() for example doesn't care about what the object type is.
  - Always use generic code when available.
- List ADT
  - ArrayLists and LinkedLists are two types of implementations of List ADTs.
  - They have some methods in common, like size(), isEmpty(), get(), and remove().
  - ArrayList
    - Has a dynamic array that expands/contracts
      - Doubles or halves usually
      - This is relatively rare so the amortized time complexity is still pretty good
    - Can use an iterator which hides the details from a user
    - The time complexity depends on the operation.
      - Simple operations like getting or setting is O(1)
      - More complex operations like add and remove have O(n) time
  - LinkedList
    - Has nodes which are attached by storing the next/previous node in that node.
      - Forms a chain of nodes.
    - We know the first and last nodes, and from any node we can get the next or previous.
    - In any addition/removal, we must update the next and previous of the surrounding nodes for it to make one chain.
    - For time complexity, finding the node in the chain is O(n), while doing anything to that node is O(1).
    - Can use a ListIterator, which has the ability to change the list while it's going through it.

Data Structures
- Stacks
  - Last In-First Out
    - You can only alter or add to the top of a stack.
  - Operations
    - pop(): Removes/Returns the top element
    - peek(): Returns the top element without removing
    - push(E object): Adds a new element to the stack
      - Only operation with an obvious parameter, you need to give it what it wants to add
    - empty(): Returns a boolean
  - The time complexity of a stack is always O(1)
- Queue
  - First In-First Out
    - You add values to the back of the queue, and take from the front.
  - Operations:
    - enqueue(object): Adds an element to the end of the queue
    - dequeue(): Removes and returns the first object
    - front(): Returns the first item without removing it
    - size(): Returns the length of the queue
    - isEmpty(): Returns a boolean of if the queue is empty or not
  - Can be implemented using a Doubly/Singly LinkedList or a circular Array as well.
    - A circular array is an array where the next of the last node is the first node.
- Dictionaries
  - Interface which uses keys to access data
  - Specifically, a hash map is a type of dictionary which has keys to values.
  - Operations:
    - insertElement(k,e): Adds a value at a key area. Replaces another value if there was one.
    - removeElement(k): Removes the key and the associated information
    - findElement(k): Returns the value at the key, or null if there is none.
  - A hash table is a way of organizing keys into a table system, but it could have some problems.
    - Multiple keys could be at the same index, so a good hash table should aim to keep them separate while minimizing the table's size.
    - A hash table has two methods:
    1. A Hashcode function puts a value to a key. Keys should be non-repetitive with minimal chance of collision.
    2. A compression map or a hash function optimizes a way to put these values into a map.
       - Think of it like you have a lot of balls to sort into a few boxes. Even though every ball has a unique number, you can't just give them their own box. A compression map should be able to put multiple similar balls in one box.
       - MAD Method: A good hash function. You take your value ((i*a) + b) / N, where a and b are constants, and N is the number of buckets.
    - Multiple keys could still point to the same place despite our efforts, so we can use a list which stores all these values.
    - Time Complexity
      - Worst Time Complexity is normally O(n) for operations like insert/remove
      - Load Factor (L): In a good hashfunction the complexity is O(n/N) or O(L)
        - N is the table's size, n is the number of elements (this is the load factor)
      - If the load factor is less than one, we say the complexity is O(1)

```
//Hashcode method
public int hashCode() {
int hash = 17 + firstName.hashCode();
hash = hash * 31 + lastName.hashCode();
return hash;
}
```

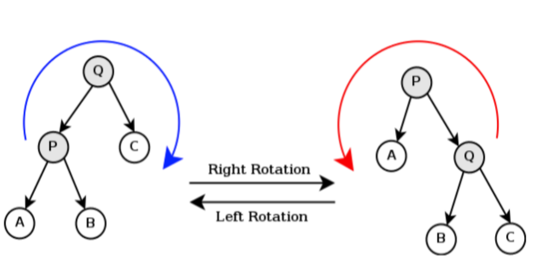
- Binary Trees
  - A recursive data structure which can access left and right nodes in a root/child relationship.
  - Terms:
    - Root: Has nodes that extend from it
    - Child: Comes from a node (root)
    - Internal node: has at least one child
    - Leaf: Has no children
    - Path: A set of nodes to go from one to another
    - Full Binary Tree: Every node has either 0 or 2 children.
    - Complete Binary Tree: Every level is filled before going to the next level.
  - Operations:
    - leftChild(): Returns the left child node
    - rightChild(): Returns the right child node
    - isLeaf(): Checks if the node has children or not
  - Theorems:
    - Number of Internal Nodes = # of leaves - 1 (for full binary trees)
    - Number of leaves of a tree with height h is at most 2^h and at least h + 1
      - It's log2, which is why we have 2^h, and minimum of one leaf is needed to start a new level
    - Number of nodes of a tree with height h is at most 2^(h+1) − 1 and at least 2h + 1
  - Binary Tree Traversal
    - Preorder Traversal: Read node, left child, right child
    - Inorder Traversal: Left child, read node, right child
    - Postorder Traversal: Left child, right child, read node



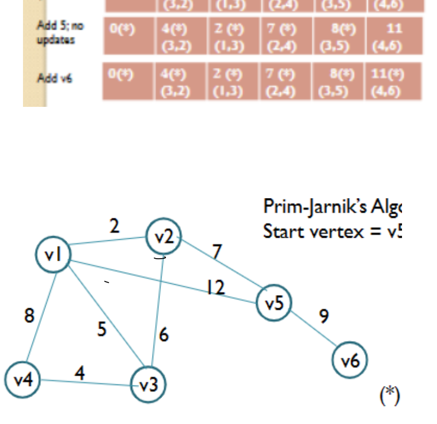
(This is not a full nor complete binary tree)

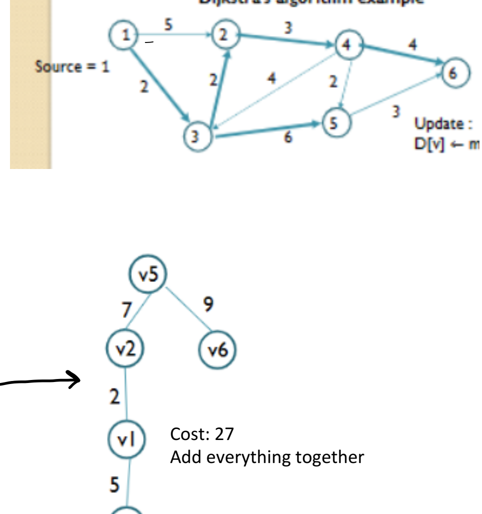
Preorder enumeration : ABDCEGFHI
Inorder enumeration : BDAGECHFI
Postorder enumeration : DBGEHIFCA

  - Binary Search Trees
    - A specific type of binary tree used to store information by order of weight or value.
    - All items to the left are smaller than the root, all items to the right are bigger
      - This is true for the main root or smaller subsections
    - Operations:
      - find(): Finds if a key exists in a binary search tree
      - insert(): Inserts a value into the correct location in a BST and then adjusts the values around it
      - delete(): Removes a value from BST and then adjusts the values around it
      - For both insert and delete, the item is put at the bottom and is sifted upwards by switching place with the parent until it reaches where it's supposed to be.
    - Balancing
      - We want our functions to be balanced to achieve a complete BST, which has a good time complexity. We use height balancing
      - Balance Factor: Height of right subtree - height of left subtree
        - This should be -1, 0, or 1 for it to be a good BF.
        - If it becomes greater than 2 or less than -2, you need to do a rotation.
      - Left Rotation: Moves the left child to become the new local root
      - Right Rotation: Moves the right child to become the new local root
      - Double Rotation: A binary tree can be doubly rotated (either the same type or even opposite directions) to balance the whole of a tree.

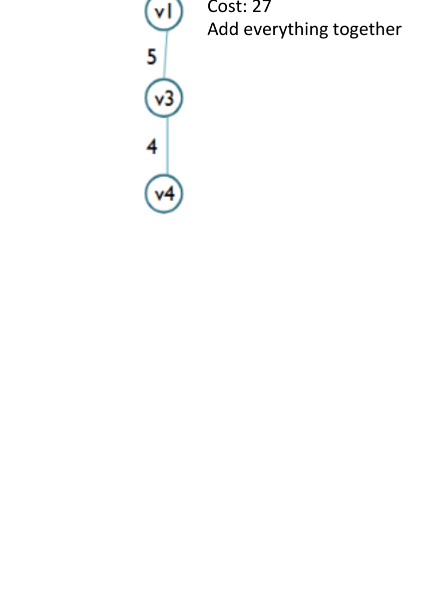


- Max-Priority Queue
  - Similar to a regular queue, but can prioritize certain elements to be removed from the queue quicker.
    - Could also be done as a minimum-priority queue, works the same way but takes the least weighted item first.
  - Operations:
    - insertElement(): insert element in queue
    - removeMax(): remove and return largest
    - maxElement(): return max element
  - Can use a heap to organize data, as well as a binary tree.
    - Is it a complete binary tree with an array implementation
    - Positions in Array
      - Parent(R) = [(R-1)/2]
      - Left Child(R) = 2R + 1
      - Right Child(R) = 2R + 2
  - Heap Implementation
    - The root must always be bigger than the child, but the children themselves do not relate to one another.
    - Root element is the maximum.
    - The height is at most log(n) (cause of the doubling property)
      - That's why it's O(logn) time complexity
    - Operations:
      - maxElement(): returns the root
      - insertion(): Puts an element at the very bottom, and sifts it up until it reaches its correct area.
      - removeMax(): Swaps the root to the lowest value, removes it, then corrects the heap by promoting the bigger of the two children on each level.
    - Heaps are usually stored in arrays. You can take a normal array and make it into a heap array.
      - Time to build a heap with n elements is O(nlogn)
      - Better to start with small heaps of a smaller height, then add the root to combine the two of them together.
      - Start with the leaves (subheaps of height one), then add their parent as a root, then sift down until it makes sense.
      - Time complexity doing it with smaller heaps is O(n), which is better.
- Graphs
  - Have vertices and edges
  - Vertices: points of data
  - Edges: relationships between the edges
    - There's in-degrees and out-degrees.
    - An edge can also direct into itself
    - You can assign some weight to edges as a cost, distance, or something else.
  - Graphs can be directed or undirected
    - Directed: Edges have directions, so one vertex can influence another but the reverse isn't necessarily true.
    - Undirected: Edges are bilateral, it's like arrows in both directions.
  - Graph's Data Structures
    - Adjacency Matrix
      - Use a matrix with n number of rows and columns.
      - Name the rows and columns as each vertex.
      - It's useful for some path problems, but is inefficient for large values.
    - Adjacency List
      - Use a list which contains all of the vertices.
  - Paths/Cycles
    - Paths: A sequence of vertices to get from one to another.
    - Cycle: A path that connects a vertex back to itself (so comes about more than once).
    - Simple paths are when there's no cycles.
    - Simple cycles are when the only repetition is the first and last vertex.
  - Traversal
    - Breath-First Search
      - Uses a queue which adds all of the outgoing edges into the queue, and adds the children of the dequeued values.
    - Depth-First Search
      - Continually commits to one path until it reaches a dead-end, where after it'll backtrack and do it again.
  - Shortest Path Algorithm
    - Dijkstra's Shortest Path
      - Used for directed graphs
      - Breaks the data set into the start vertex in one side, and the rest of the vertices on the other.
      - Evaluates the edges going from the start vertex to adjacent vertices, and commits when it confirms a shortest path.
      - Adds confirmed shortest path vertices to the side with the start index until all of the vertices have a confirmed shortest path.
      - Time complexity is O(mlogn)





    - Prim-Jarnik's MST algorithm
      - Minimum cost standing tree
      - Used for undirected graphs by creating a tree with the minimum cost of edges (trying to minimize how many edges we use).
      - Start from the start vertex then add edges and take minimums.
      - It should look very similar to Dijkstra's algorithm.
      - Total time complexity is also O(mlogn)



Sorting Algorithms
- Decision Tree Sorting / Comparison-Based Sorting
  - Literally true false questions until you get an answer
  - The lower bound is Ω(h) with height log_2(n!)
- O(n^2) Sorting
  - Insertion Sort
    - Every time you go to a new item in the array, you put it in its correct sorted position.
  - Selection Sort
    - Passes through the array several times, sorting one at a time
    - Good for small arrays
- O(nlogn) Sorting
  - Heap Sort
    - Uses property of heap (higher values are roots, lower are children) to sort
    - Start by making small heaps then merge them together by bigger root values
  - Merge Sort
    - Iterate through two lists and take the lower into a new system.
    - You assume the two separate lists are already sorted.
    - Merge time is O(n)
  - Quick Sort
    - Chose a pivot element and then make three buckets (smaller than pivot, equal to pivot, bigger than pivot).
    - Recursively sort the smaller and bigger than buckets until everything is sorted.
- Distribution Based Sorting
  - Bucket/Bin Sort
    - Sorts elements into a set of buckets which are easier to solve for.
    - Time complexity is O(n+N), where n is the number of elements, and N is the number of buckets.
  - Radix Sort
    - Sorting by the least significant digit all the way to the most significant digit.
    - Not a comparison based sorting
    - O(m(n+N)) --> O(n)
      - N and m are constants, not variables
      - The time is linear because it's not really sorting
- Selection Finding
  - Looking for the kth smallest element using a recursive structure
  - Finding the smallest is just O(n), since you have to check every element anyways.
  - The best way is to use heaps, make a miniheap and use localized extractMin() operations
    - This has a better time complexity than sorting with O(n + k*logn) when k is small.
      - K from kth smallest. 3rd smallest for example.