CSE 332 Notes

Contents

1 Abstract Data Types

1.1 Abstract Data Type

An abstract data type is a set of expected behaviors for a set of operations

1.2 Data Structure

A data structure is a way of organizing data points

- A data structure is an implementation of an ADT

1.3 Queues (FIFO)

- State
  - Set of elements
- Behavior
  - `insert(element)` add a new element to the collection
  - `remove()` returns the element that has been in the collection the longest and removes it
  - `peek()` find but do not remove the element that has been in the collection the longest

1.4 Min Priority Queues

- State
  - Set of comparable values
  - Ordered based on priority
- Behavior
  - `insert(element)` add a new element to the collection
  - `removeMin()` returns the element with the smallest priority and removes it
  - `peekMin()` find but do not remove the element with the smallest priority

1.5 Stacks (LIFO)

- State
  - Set of elements
- Behavior
  - `insert(element)` add a new element to the collection
  - `remove()` returns the element that has been in the collection the shortest and removes it
  - `peek()` find but do not remove the element that has been in the collection the shortest

1.6 Dictionaries

- State
  - Set of (key, value) pairs
- Behavior
  - `insert(key, value)` inserts (key, value) pair. If key is already in dictionary, overwrites the previous value
  - `find(key)` returns the stored value associated with key
  - `delete(key)` removes the key and its value from the dictionary

1.7 Sets

- State
  - Set of elements
- Behavior
  - `insert(element)` inserts elements into the set
  - `find(element)` returns true if element is in the set, false otherwise
  - `delete(key)` removes the key and its value from the dictionary

2 Trees

2.1 Binary Trees

- Height – the number of edges on the longest path from the root to a leaf
- Perfect – a tree is perfect if every row is completely filled
- A perfect binary tree with n nodes has a height of $\log_2(n+1) - 1$
- A perfect binary tree with height h has $2^{h+1} - 1$ nodes

2.2 AVL Condition

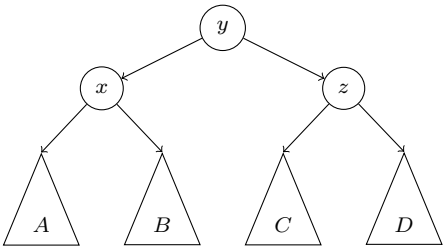
For every node, the height of its left subtree and right subtree differ by at most 1

2.3 AVL Trees

An AVL tree is a binary tree that maintains the AVL condition

- The minimum number of nodes $S(h)$ of an AVL tree of height h is given by $S(h) = S(h-1) + S(h-2) + 1$
- The maximum number of nodes $L(h)$ of an AVL tree of height h is given by $L(h) = 2^{h+1} - 1$

2.4 AVL Tree Insertion



Given that y is the lowest unbalanced node, there are four types of rotations

- A is the deepest node
  - Right rotation $A - x - y$
- B is the deepest node
  - Left rotation $A - x - B$ and right rotation $y - B - x$
- C is the deepest node
  - Right rotation $C - z - D$ and left rotation $y - C - z$
- D is the deepest node
  - Left rotation, $y - z - D$

2.5 Binary Heaps

A binary heap is a binary tree where

- Every node is less than or equal to all of its children
- The smallest element is the root
- The tree is complete
- Every level of the tree is completely filled, except possibly the last row, which is filled from left to right

2.6 Floyd's Build-Heap Algorithm

1. Arbitrarily insert the elements into a binary tree, respecting the shape property
2. Starting from the lowest level and moving upwards, percolate the node down until the heap property is restored

Floyd's build-heap algorithm is guaranteed to return a binary heap in $\Theta(n)$ time

2.7 B-Tree Dictionaries

B-Trees reduce memory accesses by only storing keys in each node

- Each internal node has M children with $M-1$ signposts
- Each leaf node has L key-value pairs
- Each node takes up exactly one page in memory
- Maximizes the number of children pointers stored in each page
- Allows each node to have a lot of children
- Makes the tree shorter
- Values are stored in leaves
- Each leaf takes up exactly one page in memory

2.8 Balancing B-Trees

- Root
  - Starts as a leaf
  - Has between 2 and M children
- Internal nodes
  - Have between $\lceil \frac{M}{2} \rceil$ and M children
- Leaf nodes
  - Have between $\lceil \frac{L}{2} \rceil$ and $L(\text{key}, \text{value})$ pairs
- Keep all leaves at the same level
- Choose L and M as large as possible while still fitting in one page in memory

3 Hash Tables

3.1 Hash Tables

Given an array of size n, a hash table uses a hash function to compute an index i where $0 \leq i \leq n-1$ in which each element can be found

3.2 Load Factor

A load factor λ is the ratio between the number of elements and the number of indices in an array

3.3 Separate Chaining

Elements that share a hash value are stored in a linked list under the hash value

3.4 Open Addressing

- Linear Probing
  Let h be the hash function. On the i^{th} probe, the new hash value is $h(\text{key}) + i$
  - If there is already an element under the new hash value, probe again
  - Suffers from primary clustering
- Quadratic Probing
  Let h be the hash function. On the i^{th} probe, the new hash value is $h(\text{key}) + i^2$
  - If there is already an element under the new hash value, probe again
  - Suffers from secondary clustering
  - If $\lambda < \frac{1}{2}$ and the table size is a prime number, then quadratic probing will find an empty slot
- Double Hashing
  Let h_1 be the primary hash function and h_2 be the secondary hash function. On the i^{th} probe, the new hash value is $h_1(\text{key}) + i * h_2(\text{key})$
  - If there is already an element under the new hash value, probe again
  - Reduces chance of primary or secondary clustering assuming table size is prime

4 Graphs

4.1 Graphs

A graph G is a pair (V, E) where V is a set of vertices/nodes and E is a set of edges

- Graphs can be directed or undirected
  - Edges have direction or no direction

4.2 Graph Terminology

- Degree: the number of edges connected to the node
- In-degree: the number of edges directed into the node
- Out-degree: the number of edges directed out of the node
- Disconnected: there exists a node in the graph that is not connected to any other node
- Walk: a sequence of adjacent vertices
- Directed Walk: a sequence of directionally adjacent vertices
- Length: the number of edges in a walk
- Path: a walk that does not repeat a vertex
- Cycle: a path that returns to its starting vertex

4.3 Adjacency Matrix

In an adjacency matrix `a`, the element `a[u][v]` has value 1 if there is an edge (u,v), and 0 otherwise

- $O(n^2)$ space complexity

4.4 Adjacency List

In an adjacency list, the i^{th} element contains a list of neighbors of u_i

- $O(n+m)$ space complexity, where n is the number of vertices and m is the number of edges

4.5 Breadth First Search and Depth First Search Algorithms

```
function search(graph, source) {
    toVisit.add(source);
    source.visited = true;

    while (toVisit.size() > 0) {
        curr = toVisit.remove();
        for (vertex : curr.neighbors) {
            if (!vertex.visited) {
                toVisit.add(vertex);
                vertex.visited = true;
            }
        }
        finished.add(curr);
    }
}
```

- In breadth first search, `toVisit` is a FIFO queue
- In depth first search, `toVisit` is a LIFO stack
- Both algorithms have $O(|V| + |E|)$ running time

4.6 Dijkstra's Shortest Path Algorithm

```
function dijkstraShortestPath(graph, source) {
    for (vertex : graph.vertices) {
        vertex.dist = infinity;
    }
    source.dist = 0;
    toVisit.add(source, 0);

    while (toVisit.size() > 0) {
        curr = toVisit.removeMin();
        for ((curr, neighbor) : curr.outEdges) {
            if (curr.dist + weight(curr, neighbor) < neighbor.dist) {
                if (neighbor.dist == infinity) {
                    toVisit.insert(neighbor, curr.dist + weight(curr, neighbor));
                } else {
                    toVisit.decrKey(neighbor, curr.dist + weight(curr, neighbor));
                }
                neighbor.dist = curr.dist + weight(curr, neighbor);
                neighbor.predecessor = curr;
            }
        }
        finished.add(curr);
    }
}
```

- In Dijkstra's algorithm, `toVisit` is a minimum priority queue
- Dijkstra's algorithm has $O(|E| \log |V| + |V| \log |V|)$ running time

4.7 Prim's Minimum Spanning Tree Algorithm

```
function primMST(graph) {
    for (vertex : graph.vertices) {
        vertex.dist = infinity;
    }
    source.dist = 0;
    source.bestEdge = (source, source);
    toVisit.add(source, 0);

    for ((source, neighbor) : source.outEdges) {
        neighbor.dist = weight(source, neighbor);
    }
    while (toVisit.size() > 0) {
        curr = toVisit.removeMin();
        spanning_tree.add(u.bestEdge);
        for ((curr, neighbor) : curr.outEdges) {
            if (weight(curr, neighbor) < neighbor.dist) {
                neighbor.dist = weight(curr, neighbor);
                neighbor.bestEdge = (curr, neighbor)
            }
        }
        finished.add(curr);
    }
}
```

- At each iteration, Prim's algorithm adds the closest vertex to the currently reachable set
- In Prim's algorithm, `toVisit` is a minimum priority queue
- Prim's algorithm has $O(|E| + |V| \log |V|)$ running time

4.8 Kruskal's Minimum Spanning Tree Algorithm

```
function kruskalMST(graph) {
    <initialize each vertex as a connected component>
    <sort edges by weight>

    for (edge (u, v) in sorted order) {
        if (u and v are in different components) {
            <add (u, v) to MST>
            <update u and v to be in the same component>
        }
    }
}
```

- At each iteration, Kruskal's algorithm adds the smallest edge that expands the currently reachable set
- Kruskal's algorithm has $O(|E| \log |V|)$ running time

4.9 Topographical Sort

A topographical sort of a directed graph is a linear ordering of its vertices such that all edges go from left to right

- Finding a topographical ordering of a directed graph takes $O(|V| + |E|)$ running time

4.10 Directed Acyclic Graph

A directed acyclic graph is a directed graph with no directed cycles

4.11 Directed Graph Connectedness

- In strongly connected graphs, you can get from every vertex to every other and back
- In weakly connected graphs, you can get from every vertex to every other and back, if you ignore the direction of the edges
- In disconnected graphs, you cannot get from every vertex to every other and back, even if you ignore the direction of the edges

4.12 Strongly Connected Component

A set of vertices C is a strongly connected component if every pair of vertices in C is connected in both directions, and there is no other vertex which is connected to every vertex of C in both directions

- Finding the strongly connected components of a directed graph takes $O(|V| + |E|)$ running time
  - Algorithm makes use of depth first search

5 Asymptotic Analysis

5.1 Basic Operations

Assume basic operations take the same constant amount of time

- Adding `ints` or `doubles`
- Assignment
- Incrementing a variable
- A return statement
- Accessing an array index or an object field

5.2 Big-O Notation

$f(n)$ is $O(g(n))$ if there exists positive constants c, n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$

- Big-O represents an upper bound for the algorithm run time

5.3 Big-Omega Notation

$f(n)$ is $\Omega(g(n))$ if there exists positive constants c, n_0 such that for all $n \geq n_0$, $f(n) \geq c \cdot g(n)$

- Big-Omega represents a lower bound for the algorithm run time

5.4 Big-Theta Notation

$f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$

- Big-Theta represents an exact bound for the algorithm run time

5.5 Worst Case Analysis

The running time for the worst state the data structure can be in, or the worst input it can receive

5.6 Asymptotic Analysis Examples

Function sizes

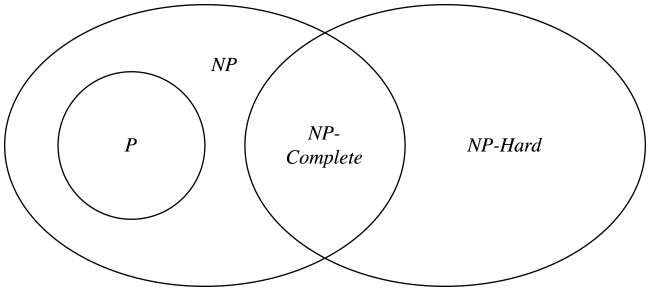
1. $\log n$
2. n
3. $n!$

Examples

- $10n \log n + 3n$ is $O(n \log n)$
- $20n \log \log n + 2n \log n$ is $O(n \log n)$
- 2^{3n} is $O(8^n)$
- $10n^2 + 15n$ is $O(n^2)$ and also $O(n^3)$
  - $O(n^2)$ is a tight big-O bound
  - $O(n^3)$ is a loose big-O bound

6 Complexity Classes

6.1 Complexity Classes



6.2 Polynomial

P is the set of all decision problems that have an algorithm that runs in time $O(n^k)$ for some constant k

- All P problems are NP problems

6.3 Non-Deterministic Polynomial

NP is the set of all decision problems such that if the answer is yes, there is a proof which can be verified in polynomial time

- NP is a superset of P

6.4 Polynomial Time Reducible

Problem A reduces to problem B in polynomial time if there exists an algorithm that, using a hypothetical polynomial time algorithm for B, solves A in polynomial time

- Problem A can be translated to problem B and solved using a polynomial time algorithm for B
- This means that problem A is easier than problem B

6.5 NP-Complete

A problem A is NP-complete if A is in NP and all problems in NP reduce to A in polynomial time

- An NP-complete problem is the hardest problem in NP
- A polynomial time algorithm for an NP-complete problem can be used to solve every problem in NP in polynomial time via reduction

6.6 NP-Hard

A problem A is NP-hard if all problems in NP reduce to A in polynomial time

- All NP-complete problems are NP-hard
- An NP-hard problem is not necessarily NP

6.7 Sample Problems

P problems

- Short Path
  Given a directed graph, is there a path from s to t of length at most k?
- Minimum Spanning Tree
  Given a weighted graph, is there a spanning tree of weight at most k?

NP problems

- Prime Numbers
  Given a number X, is X prime?
- Sudoku
  Given a Sudoku table, is it a correct solution?

NP-complete problems

- Long Path
  Given a directed graph, is there a path from s to t of length at least k?
- Traveling Salesman
  Given a weighted graph, is there a walk that visits every vertex and returns to the start of weight at most k?

NP-hard problems

- Halting Problem
  Given an arbitrary computer program, will it terminate after a finite period?
- $n \times n$ Chess
  Given an $n \times n$ chessboard, is a move the best possible one?

7 Sorting Algorithms

7.1 Goals

- In-Place Sorting
  - Only use $O(1)$ extra memory
  - Sorted array is given back in the input array
- Stable
  - If `a` appears before `b` in the initial array and `a.compareTo(b) == 0`, then `a` appears before `b` in the final array

7.2 Insertion Sort

Insertion sort builds a sorted sub-array at the front of the original array. Takes the first element of the unsorted sub-array and inserts it into the sorted sub-array

- In-place and stable
- Best case $O(n)$ time
  - When array is already sorted
- Worst case $O(n^2)$ time
  - When array is sorted in reverse

7.3 Selection Sort

Selection sort builds a sorted sub-array at the front of the original array. Finds the smallest element of the unsorted sub-array and inserts it at the end of the sorted sub-array

- In-place but not stable
- Best, worst, average case $O(n^2)$ time

7.4 Heap Sort

Heap sort builds a sorted sub-array at the rear of the original array, and maintains the unsorted sub-array as a min-heap. Finds the smallest element of the min-heap and inserts it at the front of the sorted sub-array

- In-place but not stable
- Best, worst, average case $O(n \log n)$ time

7.5 Merge Sort

Merge sort splits the array in the middle, recursively sorts the two halves, then merges the two sorted halves together. When merging the two sorted halves, it compares the smallest remaining element in each half, and inserts the smallest element into the merged array

- Not in-place but stable
- Best, worst, average case $O(n \log n)$ time

7.6 Quick Sort

Quick sort splits the array into values smaller than a pivot and values larger than a pivot, recursively sorts the two halves, and then merges the two sorted halves together. When merging the two halves, it combines the left half, the pivot, and the right half together

- In-place but not stable
- Best, average case $O(n \log n)$
  - When pivot is the median element in the array
- Worst case $O(n^2)$
  - When pivot is the largest or smallest element in the array

Pivot choosing strategies

- Pick the first element
  - Simple to implement
  - If list is already sorted then running time approaches $O(n^2)$
- Pick an element at random
  - Probability $1 - \frac{1}{n^2}$ of $O(n \log n)$ time
  - Generating random index takes too long
- Pick the median
  - Guaranteed $O(n \log n)$ time
  - Finding the median takes linear time
- Median of three (median of first, middle, and last elements in the array)
  - Likely $O(n \log n)$ time
  - Takes constant time

7.7 Comparison Sorting Lower Bound

Any sorting algorithm which only interacts with its input by comparing elements must take $\Omega(n \log n)$ time

7.8 Bucket Sort

Bucket sort distributes the elements of an array into a number of buckets, and each bucket is then sorted individually

- Not in-place but can be stable depending on the bucket sorting algorithm used
- Best, average case $O(m + n)$ time, where m is the number of buckets and n is the number of elements
- Worst case bucket sorting algorithm time
  - If insertion sort is used, then worst case $O(n^2)$ time
  - If merge sort is used, then worst case $O(n \log n)$ time

7.9 Radix Sort

Radix sort runs a bucket sort with respect to the first digit of the elements

- Not in-place but stable
- $O(d(n + r))$ time, where d is the number of digits in each entry, r is the base of the number system, and n is the number of elements

8 Memory Hierarchy

8.1 Memory Hierarchy

- CPU Register
  This is the processor's memory
  - Size: 32 bits
  - Time: 0 ns
- L1 Cache
  This is the processor's primary cache
  - Size: 128 KB
  - Time: 0.5 ns
- L2 Cache
  This is the processor's secondary cache
  - Size: 2 MB
  - Time: 7 ns
- RAM
  This is where running programs are stored
  - Size: 8 GB
  - Time: 100 ns
- Disk
  This is the computer's long term storage
  - Size: 1 TB
  - Time: 8,000,000 ns

8.2 Runtime Optimizations

- Temporal locality
  If you use some piece of memory, you are likely to use that exact data soon
  - OS keeps recently used memory in the cache
- Spatial locality
  If you use some piece of memory, you are likely to use nearby data soon
  - OS moves memory to the cache in pages

9 Parallelism

9.1 ForkJoin Library

```
1  import java.util.concurrent.ForkJoinPool;
2  import java.util.concurrent.RecursiveTask;
3  import java.util.concurrent.RecursiveAction;
```

- Generate a new thread object with `ForkJoinPool.commonPool().invoke(ThreadObject);`
- Run a thread object in a new thread with `.fork()`
- Run a thread object in the current thread with `.compute()`
- Wait for a thread to finish with `.join()`

9.2 Maps

Map operations apply some function to each element in a collection

- i.e. vector addition

9.3 Reduces

Reduce operations reduce a collection of elements into a single result

- i.e. find maximum element

9.4 Maps and Reduces

- Maps usually extend `RecursiveAction`
- Reduces usually extend `RecursiveTask<E>`

9.5 Race Conditions

A race condition is an error in parallel code where the output depends on the order of execution of the threads

- Data race

  A data race is an error where the following could happen at the same time

  - Two threads are writing to the same variable
  - One thread writes to a variable while another is reading it

- Bad interleaving

  Bad interleaving is when incorrect behavior could result from a particular sequential execution order

9.6 Locks

A lock is a concurrency control technique that allows at most one thread to own it at a time

- `acquire()` stops the thread until the lock becomes available
- `release()` allows another thread to acquire the lock

9.7 Speedup Analysis

- Work is the running time on one processor
- Span is the running time on infinitely many processors
- T_P is the big-O running time with P processors
- $T_P = O\left(\frac{n}{P} + \log n\right)$ where n is the number of threads and P is the number of processors
  - $T_1 = O(n)$
  - $T_\infty = O(\log n)$
- The speedup for P processors is $\frac{T_1}{T_P}$
- The speedup for infinite processors is $\frac{T_1}{T_\infty}$

9.8 Amdahl's Law

Let the work be 1 unit of time and let S be the fraction of the code that is nonparallelizable

- With 1 processor, the running time is $T_1 = S + (1 - S) = 1$
- With P processors, the running time is $T_P \geq S + \frac{1-S}{P}$
- With ∞ processors, the running time is $T_\infty \geq S$
- The speedup with P processors is $\frac{T_1}{T_P} \leq \frac{1}{S+\frac{1-S}{P}}$
- The speedup with ∞ processors is $\frac{T_1}{T_\infty} \leq \frac{1}{S}$