Exam II Review

CS 008 Data Structures & Algorithms
Pasadena City College

May 20, 2024

Contents

Chapter 1

Introduction to Trees

1.1 Trees

Trees are a series of nodes where one node is the root node to the tree. Every other node apart from the root node will have a parent node. All nodes are connected in some way and the connections between the nodes go in one direction.

Definitions:

- **Parent**: A node that points to another node.
- **Child**: A node that is pointed to by a parent node.
- **Leaf**: These are nodes that do not point to another node.
- **Height & Depth**: These are the height and depth of the tree which represent the amount of nodes followed to reach the bottom for the height or a certain node for the depth. The root node is not included in the height and depth of a tree.
- **Subtree**: A portion of the tree can be also be a tree (ignoring the parent nodes of the root of the subtree to create the subtree).

1.1.1 Taxonomy

There are many different types of trees that exist. Some of them include:

- Binary Trees
- N-ary trees
- Heaps

1.2 Binary Trees

A binary tree is a tree where the nodes in the tree would have 2 or fewer child nodes. There are two adjectives that can be used to describe a binary tree.

1.2.1 Full binary trees

A full binary tree is a binary tree where every leaf is at the same path and every non-leaf has two children.

1.2.2 Complete binary trees

All leaves have the same or atleast within one depth from each other in a complete binary tree. All non-leaves of a complete binary tree (with the exception of maybe one non-leaf) have two children. Any level of the tree that is not full has nodes that are arranged in the left most spots. With this, a complete binary tree would have a right and a left child node for parent nodes.

1.2.3 Implementation

A binary tree can be implemented as either an array or a linked list of nodes. The array implementation of the binary tree is simply just an ordered array of the nodes by number (so if we had a tree where the root is considered 1 and the children are 2 and 3, the array representation would be 1, 2, 3).

Operations for Binary Trees:

- first_in_subtree(node): find first node in the subtree defined by node
- last_in_subtree(node)
- successor(node): find the next node in traversal order after node
- predecessor(node)
- insert_after(existing_node, new_node): insert new_node in traversal order
- insert_before(existing_node, new_node)
- delete(node): remove node from the tree while maintaining traversal order

For the array implementation, the time complexity of getting a node or getting a root of the tree would be O(1). With the linked list implementation, looking for a specific node in the linked list implementation would be O(n). Looking for the value of the root is still O(1) even in the linked list implemenation.

Traversing the tree to look for a value is of the time complexity $O(\log_2(n))$, since the most amount of pointers is actually related to the depth of the value in the tree. Traversal of a tree is not linear but it is also not a constant time since the time complexity is still dependant on the amount of nodes. However, since the amount of nodes grows exponentially larger than the worst case number of operations.

1.2.4 Traversals

A traversal is bidirectional, it is the idea of following the pointers in the nodes. There are three algorithms for traversing a tree consisting of:

- **Pre-order:** (Parent, Left sub-tree, Right sub-tree)
- **In-order:** (Left sub-tree, Parent, Right sub-tree)
- **Post-order:** (Left sub-tree, Right sub-tree, Parent)

1.3 Trees Review

Trees are a series of nodes where one node is the root node to the tree. Every other node apart from the root node will have a parent node. All nodes are connected in some way and the connections between the nodes can be in any direction. Also, nodes usually don't know the other child nodes of the parent.

1.4 Binary Search Tree

Recall the **Binary Search** for a sorted array. With a sorted array, you can use binary search where you start in the middle of the array and divide the array each time by half until you find your desired result.

Question:

What is the time complexity of **Binary Search** on a sorted array?

Answer: It would have a time complexity of $O(log(n))$.

Note that a binary search only works in an ordered array (keep this in mind). For a binary search tree, the left node in a parent node would be smaller than the parent node but the right node would be the largest. So, in other words, instead of checking all nodes in the tree, we could simply look at values of the node we are currently in and move through the tree based off of the value we are searching for. This significantly shortens the time complexity of searching for a value and excludes parts of the tree we know will not contain the value.

Binary Search Trees are a specific type of a binary tree where the right child of a node is greater than the parent node and the left child of the node is lesser than the parent. (It could also work with the places switched). The time complexity of searching for a value in a **binary search tree** is $O(h)$ where the variable h is the height of the tree itself which is the largest number of pointers followed from the top to the leaf node. In the case of a **balanced binary search tree**, the time complexity will be $O(h) = O(log(n))$.

1.4.1 Insert operation

When thinking about the insert operation, consider which nodes are smaller or larger than the value x?

Remember that in a binary search tree, the children of each node has a specific pattern where the left child is smaller than the parent and the right side would be larger than the parent.

1.4.2 Predecessor and Successor

The predecessor is the node with the largest value less than the input value. The successor is the node with the smallest value larger than the input value. If you think of the nodes as being "ordered", these are just the nodes "before" and "after" the input value x. However, there are no specific rules to a binary search tree where the successor or predecessor nodes have to be at a specific place, they technically be anywhere in the tree.

1.4.3 Remove operation

The main concern of the remove operation is to preserve the *BST* property of the tree. To remove in a BST tree, we would need to look for successors and predecessors in the tree. If the node being deleted is a leaf node, we can just simply remove it. In the case of a node with one child, we can simply delete the node and replace it with it's child. In the case of 2 children nodes, depending on how the tree is ordered, we can replace it with the node's in-order successor or predecessor.

1.5 Data structure theory

Notice how with some functions of certain implementations, we see a 1(a) time complexity. The a stands for the amortized (aka: average) time because for example, if we had a dynamic array, there is a chance we need to extend the dynamic array.

1.5.1 Set Binary Tree (BST)

One property of sets is the ability to also have keys. A binary search tree is considered a 'keyed' set where it is sorted by the key / value to determine it's position in the tree. Using a binary search tree, you can implement an entire set interface by just adding a key.

1.5.2 Sequence Binary Tree

Similarly to how a BST can be used to implement a set, the traversal order of the binary tree is also the sequence order. In summary, for BST Trees, we can implement a sequence or a set with the BST Tree because of the properties the BST Tree has that can be used for these purposes. Using specific properties for a BST Tree, we can implement certain abstract data types.

1.6 AVL Trees (NEED TO FINISH)

AVL Trees are a specific type of binary trees and have a specific property where it "self-balances" itself. Sometimes, trees will have an issue where

Chapter 2

Time complexity

2.1 Big-O, Big-Ω, Big-θ

Big-O:
The Big-O notation is a representation of the time complexity of the worst case scenario. Remember, if $f(x) \in O(g(x))$, we can multiply a function $g(x)$ by some constant where it will be greater than or equal to $f(x)$.

Big-Ω:
The Big-Ω notation is a representation of the best case scenario. It would be the opposite of the Big-O notation so $f(x) \geq c \cdot g(x)$

Big-θ:
The Big-θ notation is a representation of the average case, so it would be a representation of two constants multiplied by $g(x)$ where $c_1 \cdot g(x) \leq f(x) \leq c_2 \cdot g(x)$

In summary, consider Big-O to be the upper bound for $f(x)$ and the Big-Ω to be the lower bound where Big-θ is where the function $f(x)$ is bounded by both an upper and lower bound.

2.2 Space vs Time

In the same way we classify algorithms according to their time complexity, we can also classify them by their space complexity. **Spatial complexity** represents the memory requirements of a program. Often, computational efficiency comes at a cost of spatial complexity and the same vise versa. We're often more concerned about computational complexity than the spatial complexity.

Chapter 3

Direct Access Arrays

3.1 Pre-lecture questions

Important question to keep in mind for the lecture:

> **Big lecture question:** Is it possible to execute the function $find(k)$ any more quickly than $O(log(n))$?

3.2 Word RAM model

Any region of memory can be accessed in $O(1)$ complexity time. In reality it is not but for the most part this statement is true. In the word RAM model, anything that is within 64-bits is within $O(1)$. When we work with a data structure with n elements, $n < 2^w$. For most computers, $w = 64$ and we call this a **word**. Memory is divided into w-bit chunks and each chunk can be read and written in $O(1)$.

3.3 Comparison model of computation

The comparison model of computation is more restrictive than the Word RAM Model. Like what the name implies, the comparison model can only perform comparisons (==, !=, , , $\leq$, $\geq$)

3.4 Direct access arrays

Going back to our lecture question:

Is it possible to execute a $find(k)$ function any more quickly than $O(log(n))$?

Using direct access arrays, you can execute a find function in $O(1)$.
A direct access array is an array where every index in the array is associated with a specific element. Each element has it's own place in the array. Direct access arrays are very fast but have a major flaw where it takes up too much space.

Operations:

As previously stated, direct access arrays are fast however they are very spacially inefficient.

- $build() = O(n)$
- $find(k) = O(1)$
- $add(k) = O(1)$
- $delete(k) = O(1)$

These are fast! However very inefficient space wise.

In summary, direct access arrays are great for speed however for larger ranges of values, the size of the array would be too big. With smaller ranges of numbers, it is OK. Before making a direct access array, you need to know the largest potential value you can have. It is dependent on the largest *potential* element that can be in the array and contains an index for every single element up until that point.

3.5 Hashmaps

As said before, the biggest flaw to using direct access arrays is space. When working with direct access arrays, you have an index for every possible element.

However, we can reduce the hashmap to a smaller direct access array that leads to lists. We call this a **hash function** but the basic gist of it is that it reduces the large direct access array to a smaller array where the index represents a certain pattern / group of elements.

Consider:

Direct access array: $[a_0, a_1, a_2, \ldots, a_{n-1}]$ where a_i represents a potential value.

Now, consider a hash function using modulus (%). We can define some hash function k where $h(k) = k\%n$. Each group would be called a **key**. So, for example, we could say any value of the direct access array where $h(k) = k\%n \equiv 0$ has a key of 0.

The main advantage of a hash table is how efficient it is for space and it is also a lot less intense to build compared to a direct access array where the array needs an index for every *potential* element. In summary, hash tables are used as a more space efficient way to store values than direct access array while still being fast.

Chapter 4

Sorting

4.1 Why do we care?

Sorting and searching algorithms are both really important because sorted arrays are a lot nicer to work with. You off load some of the work by sorting an array beforehand to create more efficient work.

There are tradeoffs for this (like everything in computer science) and this all depends on what's needed for the task.

When we look back at sets, we can see that the find function can be a lot more efficient if the set / array is sorted.

4.2 Search algorithms

Sequential search is when you go through every element in the array (or until you get to your desired element) to look for your element. However, a sequential search has a time complexity of O(n) and if you have a sorted array, you can shorten this search by using a binary search. A binary search has been mentioned before in earlier lectures and has a benefit of being faster than a sequential search but as mentioned before, everything has tradeoffs and a binary search can only be performed on a sorted array.

4.3 Sorting algorithms

Sorting algorithms are characterized by the following:

- Computational complexity
- Spatial complexity
- Destructiveness
- "In-place" (how much extra spatial complexity the algorithm

uses)

- Stable or unstable

4.3.1 Stability

Stability in sorting algorithms imply that the order of "equal" items are in the same order as before. Equal-valued items are not exactly equal to each other and having a stable sorting algorithm means that the order before the sorting algorithm between equal value items are not changed after the sorting algorithm (which can be useful in some cases).

4.3.2 Sorting algorithms

As mentioned before, there are tradeoffs for everything and this is highlighted in sorting algorithms. Each have their pros and cons but there is no "best" sorting algorithm in general. Each sorting algorithm have their own strengths and weaknesses depending on the use case.

4.3.3 Permutation sort

As the name implies, this sorting algorithm uses permutations of the array itself. What this means is that this algorithm creates permutations (specific combinations of the array) and then checks if it's sorted. This algorithm is a bit of a joke and has a time complexity of $O(n! * n)$. This algorithm is not stable.

4.3.4 Selection sort

Selection sort is an algorithm that sorts an array one piece at a time and swaps values to create a sorted array. The time complexity for this is $O(n^2)$ and is stable depending on the implementation but is in-place.

4.3.5 Insertion sort

Insertion sort is an algorithm that sorts an array by sorting an initially small chunk of the array and then growing that chunk and sorting it until that chunk becomes the entire array. The time complexity of this algorithm is $O(n^2)$ and stability depends on implementation and is in-place.

4.3.6 Merge sort (WIP)

... IMPLEMENT LATER

4.3.7 Direct access array sorting

Sorting an array using a direct access is a lot simpler than the algorithms talked about but they do not handle duplicates well, which we call counting sort and is defined to be $O(n + U)$. However, we learned before that with direct access arrays, we could use a hash table to counter the issue of duplicate collisions. Using a hash table and a queue, we can still maintain stability with the sorted array. Which introduces us to Radix Sort.

4.3.8 Radix Sort

Radix Sort builds on counting sort. Using keys that are not just the values themselves but an ordered series of keys, we can use these series of keys to sort the array with counting sort. In general, the amount of times needed to do counting sort is $log_n(U)$

4.4 Review on Direct Access Array sorting

4.4.1 Counting Sort

Last lecture, we covered the idea of sorting using direct access arrays. The sorting algorithm that creates a direct access array of all possible elements that the array will sort and then using an $O(n)$ iteration through all elements in the array and putting it inside the direct access array created. To get the sorted array, you then iterate through the direct access array looking for slots that have been used (which indicate the element is inside the array) to get the sorted array. This adds an extra complexity of $O(U)$ where U represents the largest possible element in the array and which when combined with the iteration of the array with a complexity of $O(n)$ gives us a combined time complexity of $O(n + U)$.

4.4.2 Radix Sort

Radix sort is essentially an iterative approach to counting sort. Instead of using keys that represent the values in the range of $[0, 1, \cdots, U]$, we can approach the values as being an ordered set of keys. For example, we can sort a large range of numbers using their digits. Instead of creating a direct access array of all possible values, there can be "buckets" that contain groups of elements.

Consider an array containing the values: $[0, 10, 100, 999]$. In this example, we'll create a direct access array containing all 10 possible digits for the ones digit. Each slot in the direct access array contains a queue of possible values.

Round 1.
Create a direct access array with 10 digits, each for a possible number for the ones digit. Iterating through the array, we put each element into the direct access array (D), and then grab the array after the the filtering to obtain a new array.

Round 2.
For this round, we'll do the same thing as Round 1 but filtering through the array by the element's tenth digit.

Round 3.
This is the same as previous rounds, except we filter by their hundreds place. After this round the array is sorted. This isn't the most intuitive explanation but, through rounds of filtering we can obtain a sorted array without directly comparing values.

Chapter 5

Priority Queues

5.1 Priority Queues

The basic gist of a priority queue is that it is a data structure when elements that go in are automatically sorted into the queue.

5.1.1 Interface (IN PROGRESS)

Functions

5.1.2 Heaps

A heap is a complete binary tree and a complete binary tree is where every layer is full, except for the last layer where nodes in the last layer are to the left. Heaps are trees that are implemented as arrays.

With the heap property:

- array[i].left = array[2 * i + 1]
- array[i].right = array[2 * i + 2]
- array[i].parent = array[floor($\frac{i-1}{2}$)]

Because of the formulae above, the heap has to be a complete binary tree otherwise these properties will not work. For a heap, there does not need to be a sense of ordering and as long as it represents a complete binary tree, the given heap array formulae will work.

Heaps are trees, implemented as arrays, with the heap property. The heap property is where the parent inside the tree is bigger than or equal to their children and this property enables heap sort.

5.1.3 Review

So, unlike a "regular" queue that we learned before, a priority queue is similar in that it has a front and back but the queue is a sorting queue where elements inserted into a priority queue are sorted into the queue. In summary, the fundamental difference between a priority queue and a "regular" queue is what values are popped from the queue.

There are several ways that can be used to implement a priority queue such as using a dynamic array, sorted dynamic array or an AVL Tree. Something important to note is that the array does not have to be sorted, but having a sorted array will have different time complexities for a priority queue as an unsorted array. A priority queue is a simplification of what we are already used to with these data structures. Depending on which data structure is used, we can emulate a certain sorting algorithm.

5.2 Heaps

5.2.1 Heap Property

Heaps are a complete binary tree in principle, but is actually implemented as an array where the parent is always larger than the children. Note that a heap needs to be sorted by this property for it to maintain this heap property.

5.2.2 Implementation

Consider an implementation with a max heap (the name explains itself):

- **build(x)**: Consider an array X such that $X = 15, 13, 11, 12, 14$. The build will output a sorted array Q such that $Q = 15, 14, 13, 12, 11$. Remember, a priority queue does not have to be a sorted array but a heap needs to be sorted.[1]
- **insert(k)** : When inserting a value to a heap, we add it to the end of the heap. However, when inserting, there is a potential issue that the new value is a large value that ruins the heap property. To counter this we would need to reorder the values in the heap and we would use the function **max_heapify_up(n - 1)**.
- **max_heapify_up(i)**: So consider the same array Q which we will say is a heap, and then add a new largest value to the end of the dynamic array. However, since that value violates the property of a heap, we

[1] When you add a bunch of elements unsorted and then max_heapify_up values from the end to the beginning of the array, you will sort the array in $O(n)$ time.

swap parent and child until we maintain the heap property. It does not have to be a perfected sorted array but simply needs to follow the heap principle in the end where the parent is larger than the children. (Time complexity: $O(log(n))$)[2]

- **delete_max()**: In the case of a max heap, we would swap the last value in the heap with the first value (our maximum value in the heap) and then delete the last element in the heap which is our maximum value and remove it to obtain a constant time removal. Similarly to inserting a value, we would heapify down the heap to maintain the heap property.

- **max_heapify_down(i)**: Very similarly to the up function we defined, there is also a function for down where you compare the parent with the two children and then swap. Since we check both children, the swap will not affect the order for the other child of the tree. This function is used to maintain the heap property after deletion of the max value. (Time complexity: $O(log(n))$)

5.2.3 Heapsort

Consider a priority queue implemented using a heap called Q such that $Q = 16, 14, 15, 12, 11, 13$. We can actually sort the array associated with this by using the function $delete\_max()$. By iteratively deleting the maximum value and storing that maximum value popped, it eventually results in a sorted array. Each $delete\_max()$ is an $O(log(n))$ operation, so since the function is called iteratively (n times), we finally result in a time complexity of $O(nlog(n))$. One important thing to note is that heapsort can be in-

place based off of the implementation of the delete_max. If the delete does not actually remove the value and simply switches values and resizes[3], we eventually result in the same array that is used in the heap that is sorted at the end.

[2]The time complexity is log(n) because a heap is based off of a tree and there are at most log(n) (base 2) swaps needed to restore the property of the heap

[3]This means you keep in track of the size of the heap with a variable