Jordan Laidig, Dror Manor and Christian Meza

Jose Coria

CSS-2B

4 May 2018

Data Structure Algorithms

This report will discuss some of the types of the abstract data structures we have studied about in the CSS-2B class . It will explain the methods used by each data structure to access data elements, search for an elements, and also insert and delete these elements. In addition, the time complexity of accessing the elements of each data structure evaluated.

An Array is a fixed size, sequential collection of elements of the same type placed in memory locations that can be individually referenced by adding an index. An element is accessed by indexing the array name. This is done by placing the index of the element in square brackets after the name of the array. For example, int array[i]. You can insert values in an array when first initializing, for example, int array[size] = { 1, 2, 3 }. To delete an e array, delete array[i]. When talking about accessing an array, the Big-O is O(n). Unless the array values are sorted, the searching time for a value in an array will be O(n) since we potentially have to go through each element of the array. If the array is sorted, the search time is , since we can use binary search instead of a linear search.

A Bag data structure is a container class that is capable of holding a collection of items (such as numbers). Container classes can be implemented as a class, along with member functions to add, remove, and access items. The Bag container class should include an insert

function, which checks if there is enough room, adds the inputted value into the data structure and increments number of items in bag (used has to be updated). Erase_one function which erases one value from the array. Erase function which erases all contents of the bag (sets used to 0). The Big-O for the bag is different depending on whether the bag is implemented with an array on the inside or using Nodes. If it is an array on the inside then the Big-O will match that of an array. If the bag is implemented using Nodes then the Big-O changes. For accessing the Node version, you may have to traverse through the entire bag inorder to find the Item you want. This means the Big-O is O(n). For searching, it is similar to accessing since you still might have to search through the entire bag. When inserting items into the Bag, since order doesn't matter, the Big-O is just O(1) because you just insert an item at the front. When deleting an item, the Big- O is just O(1) because you can just switch links around.

The Linked List is the third Data Structure we talked about in class. It uses a class called the Node class. This Node class has two main parts to it: a data field and a link field. The data field can be anything that you want to use such as an int, a double, a string, or another class that you create. The link field is a way for all the Node to be linked together into a chain. The Linked List uses the links in the Nodes to link multiple Nodes together into a list. The Linked list also has a head pointer and a tail pointer which allow for access into the list. The links in the Nodes link together going from the head pointer towards the tail pointer. Accessing the different Nodes in the list can take a little effort because in order to find the one you want you have to start at the head pointer and traverse all the way through the list. This makes the accessing function have O(n). The search function is very similar because it also has to search all the way through the list so it also has O(n). Inserting a Node is actually quite easy, as long as you have a pointer to a

previous Node. Once you've created the Node you want to insert, you just need to change a couple links in the list's Nodes and then the Node has been inserted. This makes the insertion function have O(1). The deletion function is just the reverse of the insertion so it is very similar in how it works. You just change a few links in the Nodes and then delete the Node, so it has O(1).

The Doubly Linked List is very similar to the Linked List except for one change: The Nodes have two link field instead of one. These two link fields allow the Nodes to link forwards (from the head pointer to the tail pointer) and backwards (from the tail pointer to the head pointer). This makes the tail pointer much more important than in the normal Linked List. Accessing and searching through the list are very similar to the functions from the Linked List. The only difference in the Double Linked List is that you can choose to search through starting at the back, near the tail pointer, instead of starting at the front. Because of this, both of these functions have O(n). Inserting Nodes into the Doubly Linked List is almost just as easy as in the Linked List. The only difference is that in the Doubly Linked List there are a few more links that need to be changed because of the backwards links. So the insertion is also O(1). The deletion is still the reverse of the insertion and in the Doubly Linked List the deletion still just requires a few links to be changed in order to get rid of the Node. So deletion has O(1).

Stack is a type of data structure of ordered entries. In other words, there is an order by which the entries are added and removed. Items can only be inserted and removed at the top. The last data in is the first data to come out of the structure. This is known as a LIFO, which means last in, first out. Some of the Stack functions are, top() which allows you to access the top element without removing it, size() which returns the number of elements in the Stack, push()

which inserts an element at the top of the Stack, and pop() which removes the element at the top of the Stack. Since you can only access the items at the top of the Stack, in order to access something near the bottom, you have to move all the way through the stack. This means the Big-O is O(n). Searching is the same way since we can only access it from the top. Inserting items into the stack is very simple since you just add a Node at the top meaning the Big-O is O(1). Deleting items from the stack just removes them from the top so it has the same Big-O as the insertion.

A queue data structure is used to track a group of node entries in a specific order. The queue uses the “first in first out” order, the first node that was added is the first one that can be accessed using the front() method or removed from the queue using the pop() method . Only after the first node was removed, the second node can be accessed using the front() method. For example, on an escalator the first person to go on is the first to get off. To insert a node we use the push() method, which will place the node at the end of the line. To search the queue we need to copy the queue while inspecting one node at a time using the front() and pop() methods. The time complexity for searching, copying and deleting all the nodes from the queue is O(n). Since we only have direct access to front node, we potentially need to process each node.[1]

The Binary Search Tree uses Nodes to create an upside, tree like structure. The Nodes have two different link fields that link either left or right. The way the tree is populated is it takes a value and checks it against the root value. It then puts the value on the left subtree, the part of the binary tree that is to the left, if the inserted one is less than or equal to the root you are checking. Otherwise it will move the inserted value towards the right. Inserting values into the tree can be easier than the lists that we previously saw. When you first check the root after you

pick the direction to move, you eliminate half, on average, of the Nodes you need to check. Doing this every time will make the insertion O(log(n)). Searching through the list will be very similar to the insertion because of the elimination of half the possibilities making the search function also O(log(n)). Accessing and deletion all have the same elimination conditions because you can still check whether what you want to access or delete is less than or equal to or greater than making both of them also O(log(n)). This Big-O of O(log(n)) is only an average. The worst-case scenario would be if the Search Tree was made to look almost exactly like a Linked List. This would make all of the four functions move back up to O(n). The reason that the insertion and deletion would also be O(n) is because you have to find the specific spot to insert the value you want or you have to find the specific value you want to delete.

Hash tables are stored in arrays of either a specific data type or head pointers to a linked list. A hash function is used to convert a data element to an integer. The integer can range from zero to size of the array minus one and will be used as the index of the array. When an element is added to the hash table, the hash function will return the index of where the data will be stored in the hash table. There is a chance that different pieces of data will hash to the same index and create a collision, this depends heavily on the size of the array and the quality of the hash function. For example, a hash function for strings, which will calculate the remainder of dividing the sum of a string's characters ASCII values by the size of the table. In the case that the hash table size is 10, the hash of "John" will be 9 and the hash of "Fred" will be 5, yet the hash of "Bob" is 5 as well. To avoid collision we can store the data in a linked list and the head pointer will be stored in the hash table, using the example, "Bob" and "Fred" will be stored as nodes in a linked list and the head pointer for that list will be stored at index 5 of the hash table. In reality, the hash table will be much bigger and the chances of collision are not as high as in a

table of only 10 elements, which makes the best case and probably also the average case for searching, inserting and deleting elements in a hash table O(1). However, the worst case is O(n) since there is a possibility (with small hash tables and or a bad hash functions) that all the elements will be stored in one linked list. [2]

Works Cited

[1] M. Main and W. Savitch, *Data Structures & Other Objects Using C++, Fourth Edition.* Boston: Addison-Wesley, 2018.

[2] SparkNotes Editors. "SparkNote on Hash Tables." SparkNotes LLC. n.d.. http://www.sparknotes.com/cs/searching/hashtables/ (accessed May 1, 2018).

[3] E. Rowell and Q. Plepl, "Big-O cheat sheet." http://bigocheatsheet.com/ (accessed May 1, 2018).