Data Structures

June 14, 2019

1 Elementary Data Structures

1.1 Stacks and Queues

Stacks and queues are *dynamic* data structures in which we are constantly modifying the elements of a partiular set. In a stack, if we want to remove an item, we ususally remove the last element in the stack, and we call it a *last-in-first-out* type of data structure, since the last element to ocme in is the furst element we remove.

1.1.1 Stacks

As mentioned, stacks are *first-in-first-out* data structures, which means that only the 'top' element is accessible at any given time. The operations we may perform are all in O(1), meaning that they are constant time operations. These include:

1. `Push(S,value)`: When we push a value onto the stack, we are adding an element to the array and moving the `S.top` attribute, increasing it by 1.

2. `Pop()`: When we pop an item, there is no need for an argument. We just reset `S.top` down by 1, and potentially overwrite the data item when we have to.

Essentially, a stack behaves like an array with n element, and we can only use the topmost one. If the top value exceeds n, we say that **stack overflow** (ayyy lmao)has occurred. The opposite, when we try to pop an empty stack, we say that **stack underflow** has ocurred.

1.1.2 Queues

For a queue, we can perform similar operations to a stack, except that they have different names. For example, an insert operation is called `enqueue`, and a delete operation is called a `dequeue`.

We can compare a queue to a line of customers, where the first one in line is the first one to proceed. The elements all come into the back of the queue, and elements are only taken out of the front, or *head*.

Another distinctive feature of queues is that the elements 'wrap around', that is, if the queue goes to the end of the array, the next element will be arr[0]. When the tail = head -1, the queue is full, since it wrapped around to the other side and will interfere if it grows any more. This is because we are constantly removing elements from the head of the queue, so if we didn't implement this, our queue would be decreasing in size every time we dequeue.

For queues, because of its FIFO property, whenever we `dequeue`, the first element in the queue is removed, similar to the first customer in a line that proceeds to the counter. Similarly, the element that is `enqueued` always goes at the tail of the queue.

1.2 10.2 Linked Lists

A ***linked list*** is a container-type data structure in which the elements are categorized not as indices in an array, rather each element, or *node*, has a pointer to the next object. This means that linked lists don't need to be of a specified size, they can expand and contract dynamically.

This would be an example of a singly(?) linked list. A ***doubly linked list***, however, contains a pointer to the next and previous element in the list respectively. At first, the head and the tail both point to the same element. The first node, n1, should be such that `n1.prev =null`. This would let us know that a node in the list is a header. For a tail node, we would only need to chech its *next* pointer, since the tail pointer should be null as well.

For an unsorted list, if we want to search for a specific element, we would just have to perform a linear search for our desired element.

Using a *sentinel* object might help us simplify boundary condition checking. We can place another object *before* the head object and we then have `nil.next` point to the head object. the object `tail.next` could also point to the NIL object. The main reason to even use sentinels is to simplify the code maybe within a loop, becuase they don't do that much for reducing

complexity. Also, they take up extra space, so if we are using linked lists with objects and have several of them, it might be better to not use sentinels.

1.3 10.3 Implementing Pointers and Objects

This section covers mostly the pointers implementation in objects that do not support them. Suppose we wanted to implement a linked list in this way. We could do it with three arrays: one for the key, one for the next and prev. For the next and prev arrays, we have the vaue in the array element i be the index that it corresponds to in the key array. For example, the ith value in the prev array is the index of the previous element, so if prev[i] =5, that objects prev is the one in the 5th index. However, it seems that this approach is very limited?

1.4 Representing Rooted Trees

Trees can be thought of as an extension of the idea of a linked list. We can have 3 attributes per node. We need a pointer to the parent of the node; if `x.parent` = NIL, then this is the root of the tree. Considering only binary trees, they can only have two children.

However, we can still implement trees with an arbitrary number of children. We use the *left-child right-sibling representation*. This combines the idea of a tree and linked list. What this means is that we have a pointer to the leftmost child. Then, we have a pointer to `x.right`, which is a pointer to the right sibling. Then, we have the leftmost pointer serving as the head of the list and pointing to every right sibling.

2 Hash Tables

A ***hash table*** is a data structure that mimics a 'dictionary', or one in which a *key* maps to an array index. However, unlike an array where we provide the index directly, here we *calculate* the corresponding array index based on a key value. Depending on the function we use, there might be more or less collisions, or what happens when two different keys map to the same array index.

2.1 Direct-address tables

We can use an array, called a ***direct-address table***, to keep track of the data we want to associate with the key k. Each element in the array maps to a certain key.

2.2 Hash Tables

With direct address tables, we were calculating the value in a table with —U— values, one for every element in the universe of possible keys, while really only using —K— values. Instead, we can get away with using a much smaller array, and calculating the index necessray for an element. The **hash function** maps the entire universe into an array of size n. Since —U—¿—M—, there has to be at least one collision (pigeonhole principle anyone?). One way we can deal is by having a pointer in collisioned indexes to a list of elements that map to that same index. This is called ***chaining***.

When one or more values map to the index given by h(k), we create a linked list at that index; or rather, we have a pointer to the head of a linked list. Then, if we make it a doubly-linked lsit, deletion can run more efficiently. For search , we could just do a linear search starting at the head, if we just insert an item at the end every time.

Supposing we have a hash table T with a n elements currently being stored and size m, the ***load factor*** α is the ratio n/m (how 'full' the table currently is ?). The load factor can vary between 0-1. In reality however, the performance of our hash table depends on the function we use to hash the values.

n_j can be used to describe the length of the list at index h(k). The expected length of the list is $E[n_j] = \alpha$.

Theorem 11.1

In a hash table where collisions are resolved by sorting, a search takes on average $\Theta(\alpha + 1)$.

Proof The time it takes to search for an index k, if there is nothing in the index, is constant. If there is a list in the index, the expected length of the list is α, therefore the worst-case time for searching the list is O(n). This is all assuming that every element in the array is equally likely to be chosen given the original key.

2.3 Hash Functions

A good hash function obviously minimizes the probability that two keys map to the same index. We encourage simple uniform hashing, since the hash value calculated for every index should be independent of the key itself. One popular way is the *division method*, in which we take a (unrelated) prime number, and calculate the index by dividing the index and the prime number, then taking the floor of the number.

Representing the keys as natural numbers would allow us to create functions that operate on them in some way. For example, translating the number into a certain radix(base).

2.4 Division Method

The simplest way of calculating the hash function is by using modular arithmetic. since it is jsut one division, it might be quite fast, but wouldn't there be many collisions? If

$$h(k) = kmod(m) \tag{1}$$

then there are many values which would map to the same values, right?

2.5 Multiplication Method

With the multiplication method, we multiply the key number k by the A value which is between 0 and 1. Then, we multiple it by m, the table size, and take the floor part of the result.

2.6 Universal Hashing

With universal hashing, we can select from a set of functions. This would ensure that inputs cannot be selected in such a way to guarantee a collission on every input. In theory, this would ensure that on average we have relatively good performance. The main idea with universaltiy is that we can choose from a set of functions such that the probaility of two equivalent hashes is 1/m. Basically, our ideal is to have hash tables which behave as if they indexes were randomly chosen.

Then how do we design the set of functions to be used in the hashing?

. . .

3 Binary Search Trees

3.1 What are binary search trees

3.2 Traversing a binary tree

3.3 Insertion and Deletion

When we want to insert a value into the tree, it becomes a (relatively) straightforward proccess. We first start at the root, and successively check the value of each node we land on. For example, if the value we want to insert is less than the value of the root, we choose the left subtree and check the next value. We find the value for which z is greater than the key. Then, we set our *trailing pointer* equal to that value, and then make our desired node z equal to the appropriate child node of that node.

3.4 Randomly Built Binary Search Trees

Lorem Ipsum

4 Red-Black Trees

Red-Black Trees are a special case of binary trees. Each node is either red or black, and has the same properties as a normal BST. RBTs have several important characteristics:

1. **Every Node is either Red or Black**
2. **The root is black**
3. **Every leaf(a state called NIL), is also black**
4. **For all nodes, all simple paths from the node to descendant leaves contain the same amount of black nodes**

Thus the height of a red-black tree can be thought of as the **black-height** of the tree, or the amount of black nodes from the root to a leaf, which by a property of the tree will always be the same length. Black trees make good search trees because the height of the nodes is at most 2ln(n+1).

We can replicate al of the procedures for the binary trees with the red-black trees, except insert and delete, because we cannot effectively guarantee that the resulting tree will be a RBT.

4.1 Rotation

Whenever we want to perform an insert or delete operation on the RBT, we cannot guarantee that the tree afterwards will retain the same properties. T ensure that it does, we can perform a roation on the tree.

The procedure for the rotation involves managing the values of the nodes and switching them (similar to what we did with insertions in the Linked List homework).

4.2 Insertion

(1. Thats's what she said)To insert a new node into the tree, we actually insert it as if it were a regular binary tree. We then need to call another function that 'fixes up' the values in the tree. This procedure can rotate elements as necessary and guarantees that the produced tree will be a RBT.

For the fixup procedure, we need to first guarantee that the nodes are colored in the correct manner when we first insert the node. Then, we need to ensure that the other RBT properties hold, and we call the left-rotate or right-rotate procedures if we need to rearrange the height of the tree.

5 Minimum spanning trees

What is a minimum spanning tree? It is a tree such that allows us to connect all of the vertices of a graph by using the weights of the edges(so edgy bro). if we wanted to connect all the edges by minimizing or maximizing a certain parameter, and include all of the vertices, we could use a minimum spanning tree based on the graph.

6 Iterators and Containers

The C++ standard library provides some objects (similar to pointers), called containers, which allow us to sometimes store many things of different types because of the use of templates. These provide many built in functionality and might be useful whenever we want to solve problems(looking at you, cpe!).

6.1 Introduction to Containers

Containers come in all shapes and sizes, and allow us to freely use somewhat optimized data structures to not force ut to reinvent the wheel.

6.1.1 Containers Overview: NContainers

The first type of container is called the Non-linear types of containers. These are then able to be checked inside datat structure like `array<>`,`tree<>`,`List`,...