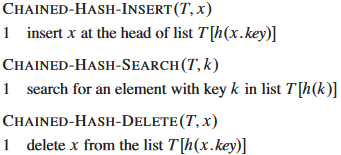
**Hash tables:**

Ultra basic description can be found in the last lesson. Some parts of the book have been copied in after the notes for other shit which might be relevant during the exam, but I cannot be bothered to read through and understand them at the time of writing.

**Avoiding hash collisions**

One problem with hash tables: Multiple keys can generate the same index through a hash function. One way of making this tolerable is by making all slots in a hash table linked lists, and then making entries which share the same slot/index be part of the same linked list. Some operations on how to manipulate such a scenario can be seen to the right.

**Hash Functions**

Now you need to figure out how to compute the index of an entry based on it’s key. There are 3 main methods for doing this, the first being division, the second being multiplication, and the third being randomization.

**Division function**

When doing division functions, you generally take the modulus of the key and the total size of the hash table. If the division function is used, it would be smartest to set the size of the hash table to be equal to a prime number, as that maximizes the number of possible slots which can accessed.

**Multiplication function**

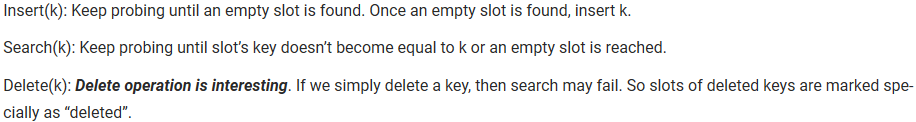
Multiply the key with a decimal number between 0 and 1, then multiply it again with the max length/size of the hash table, and round down. Note that after the first multiplication (key times 0<X<1) extract the decimal number away from the whole number (can be done by modding the result by 1), which is then what is multiplied the length/size.

As for selecting the number 0<X<1, the book suggests . Don’t ask why, there were too many numbers in the book for me to read through.

**Universal function**

I understood absolutely nothing of this.

**Open Addressing**

So in case this wasn’t pointed out before, using linked lists to store the satellite data means the hash table doesn’t store jack squat other than a pointer to each linked list. With open addressing, we take that concept and beat the ever-living shit out of it by instead choosing to store the keys directly in the table instead of just pointing to them. This has the consequence of multiple keys not being storable in the same slot. The functions below are supported in an open addressing hash table. 

With regards to searching and inserting values, the table calculates the hash of the key, and if the key at the calculated index does not match the input key it just checks the next entry and repeats with all the following entries until it comes across the desired key or it reaches a blank space in which the insert function does it’s thing and inserts the key while the search function goes on to assume that the key does not exist in the table at all.

**More advanced search/insert functions**

Because that’s always fun.

*Linear Probing*

Instead of just saying slot + 1, we accept an extra input on the hash function in the form of a number. Normally after performing the hash function on the key, the table mods the result by the size of the table, but with linear probing it adds the number input onto the result before modding. This allows for iterative searching for empty slots in a way where keys with the same hash aren’t stored right next to each other.

*Quadratic Probing*

Same as linear probing, except the number input is multiplied by some constant, after which we add the input to the power of 2 multiplied by yet another constant. See the image below for clarification as this explanation can get a little muddy.

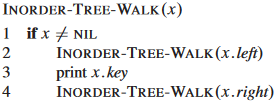


*Double hashing*

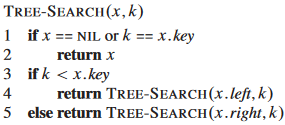
Same as linear probing, except the number input is multiplied by a second hash function run on the key.

**Binary Search Trees**

A binary search tree is a tree wherein the left child of any given node contains a value smaller than or equal to the parent’s value, while the right child contains a value that is larger than or equal to the parent’s value.

The advantage of such a tree is that you can use a very simple algorithm to print out the values in a somewhat ordered manner, as seen in the image to the right.

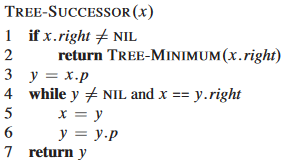
**Searching the Binary Search Tree**

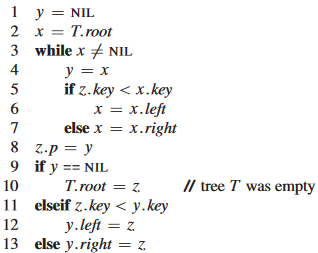
The way one is supposed to search through this tree is by taking the value, compare it to a node, if it’s smaller you go to the left side, if it’s larger you check on the right side, rinse repeat until you either find the value or hit NIL. See image to the right. The problem with this is, that this is not guaranteed to work, as any value expected to be on the left side of a given node could be risiding on the right side and vice versa. I’m gonna have to assume that the insertion method operates under the same rules as the search method, as the search method would otherwise be faulty. In fact, based on how the rest of the methods seem to work, I will assume that the insertion method follows these extra rules.

**Finding minimum and maximum values**

Purely follow either left or right path respectively.

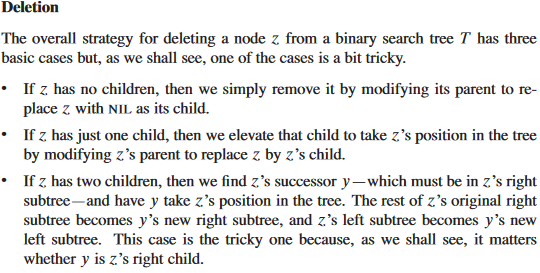
**Succeeding values**

Assume you have a value, and you want to find another value which is one step larger than your current value (as in, larger than the current, but smaller than all other values which are also larger than the current). Start by checking if the node has a right child. If it does, run the minimum function on it to find your successor. If it does not, check if the node is the right child of the parent node. If it is, repeat that process with the parent node until you come across a node without a parent, or until you come across a node that is the left child of a parent node.

**Insertion**

Functions as one would expect, taking the value, comparing it to the root, and then bouncing between left and right children until it finds a suitable empty node to put it in.

**Deletion**

A node without any children can be deleted by removing it from its parent. If it has a single child, you just overwrite the node with the child, ez pz. If 2 children are present, find a successor to the node in the node’s right subtree, and have it replace the node, inheriting the original node’s subtrees in the process. The following 2 functions are used for this exact purpose:

