Binary Search Trees, Reading

The O(log n) binary search algorithm from the previous module suggests yet another possible data structure for associative arrays, using an array that's always in sorted order. Its **operator[ ]** and **containsKey** getters could then locate a matching key with O(log n) with a bisection algorithm.

But using a data array to store key-value pairs would be a problem for push operations. They would have to *shift positions in the array* to the right to insert a new key-value at a position that maintains sorted order. And pop operations would have to shift positions to the left to fill holes. These operations necessarily become O(n) -- *not good!*

A possible solution for getting from O(n) back to something that scales well (like O(1) or O(log n)) is to use a *linked structure* like the one we considered using for heaps in an earlier module (but didn't because it appeared more complicated than the arrayed solution and did not seem to offer any advantages to make it worthwhile). Before we explore this structure for a possible binary search based associative array, let's compare and contrast to the options we already have.

Associative Array Options

Our first associative array template, AssociativeArray, worked okay, but it's push and pop operations were O(n). So it doesn't scale very well, making it suitable only for small data sets. Of course, we didn't write functions named "push" or "pop" -- we wrote the **operator[ ]** *setter* and the **deleteKey** setter instead -- but they do the same thing.

We improved on scalability with our HashTable template, where push and pop were O(1) as long as the data array had lots of empty space in it. Since we designed it with large data sets in mind, the empty space becomes a bit of an issue.

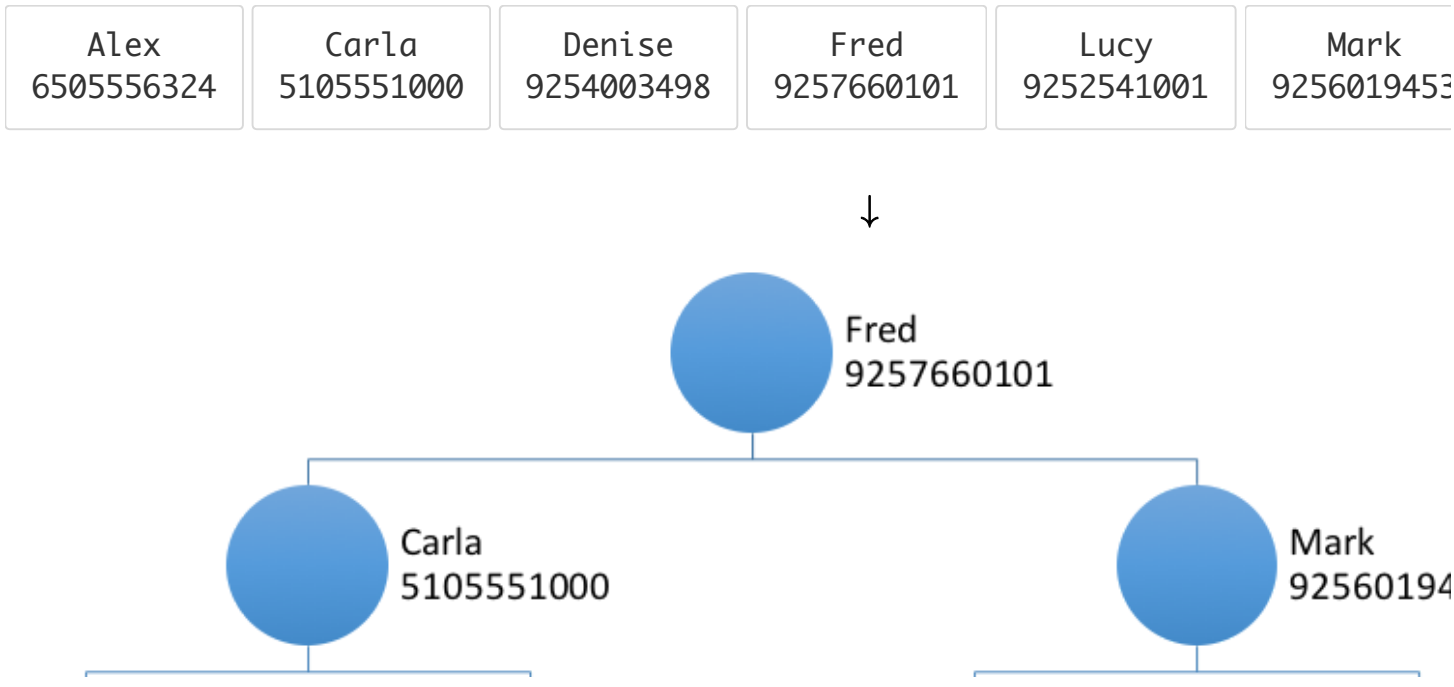
Our HashTable also does not have self-adjusting capacity. And even if it did, capacity changes would involve queuing up the stored values and rehashing them back into the resized array.

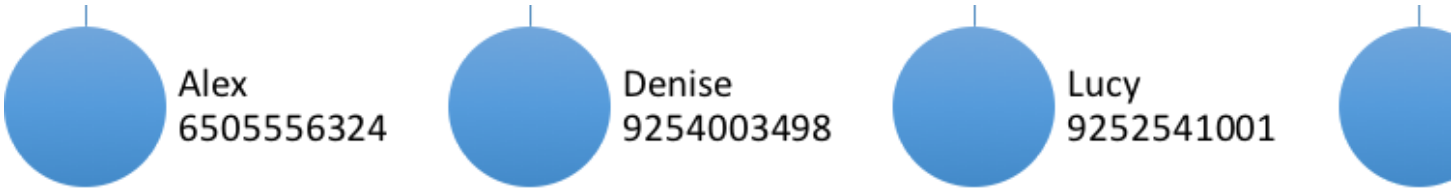
Compared to these, its big oh considerations make a binary search solution better than our AssociativeArray for large data sets. Being a linked structure of nodes, capacity would not be an issue, and even though it's big oh is slightly less favorable than the HashTable's, both scale well, and the binary search solution would have (1) no wasted data space, (2) self-adjusting capacity, and (3) no pausing for reallocating doubly-sized arrays and/or rehashing.

So it seems worth pursuing, which is what this module is all about.

Binary Search Tree Structure

Let's start by exploding an ordered array into a linked structure, with the linkages among nodes providing the exact same pathways that the binary search calculates for indexes as it traverses its array:





It's no coincidence that the arrangement of the key-value pairs in the arrayed and linked structures match in order from left to right. Note that the middle got pulled up to the very top -- that's the middle index where a binary search would start. In the remaining left and right halves, the middle index of *those* halves are pulled up to be children of the top. Those are the next indexes that would be searched. And the pattern continues to lower levels of the hierarchy.

This structure is commonly referred to as a "binary search tree", or "BST". It has the appearance of an upside-down "tree", the way we depict it above, with the node at the top called the "root" (again, it's upside-down). It's "binary" because each node can "branch to" at most two other nodes below it (still upside-down). It's "searchable" because **all the keys below and to the left of any node are less than or equal to the node's own key**, and all **below and to the right are greater or equal**. So in a linked format, the same search pathways exist that we saw in the binary search of an ordered array in the previous module.

BST Nodes

Each node has a key-value pair, and needs to know its dependent nodes -- it's left and right children:

```
struct Node
{
  K key;
  V value;
  Node* left;
  Node* right;
};
```

...as is would appear in a templated class.

BST Data Members

The root node and size are needed:

```
int siz;
Node* rootNode;
```

...and both would be initialized to zero in the main constructor. There's no concept of *capacity*.