Notes - Algorithms & Data Structures - Binary Search Trees

- Dr Nick Hayward

An initial consideration of binary search trees relative to algorithms and data structures.

Contents

Intro

A binary search tree (BST) is a binary tree.

Each node has a Comparable key (and an associated value), which satisfies the restriction,

- the key in any node is
    - larger than the keys in all nodes in that node's left subtree
    - smaller than the keys in all nodes in that node's right subtree

Comparison is context specific relative to the current node.

With binary search, we need to work with sorted data sets.

When we add or update the data, we need to re-sort the list before using binary search.

If we're working with sorted lists of data, e.g. an array of books, we quickly encounter a problem. A list may need to be updated, and a item in the array is deleted. However, we then need to add a value to the index where we stored the last deleted item.

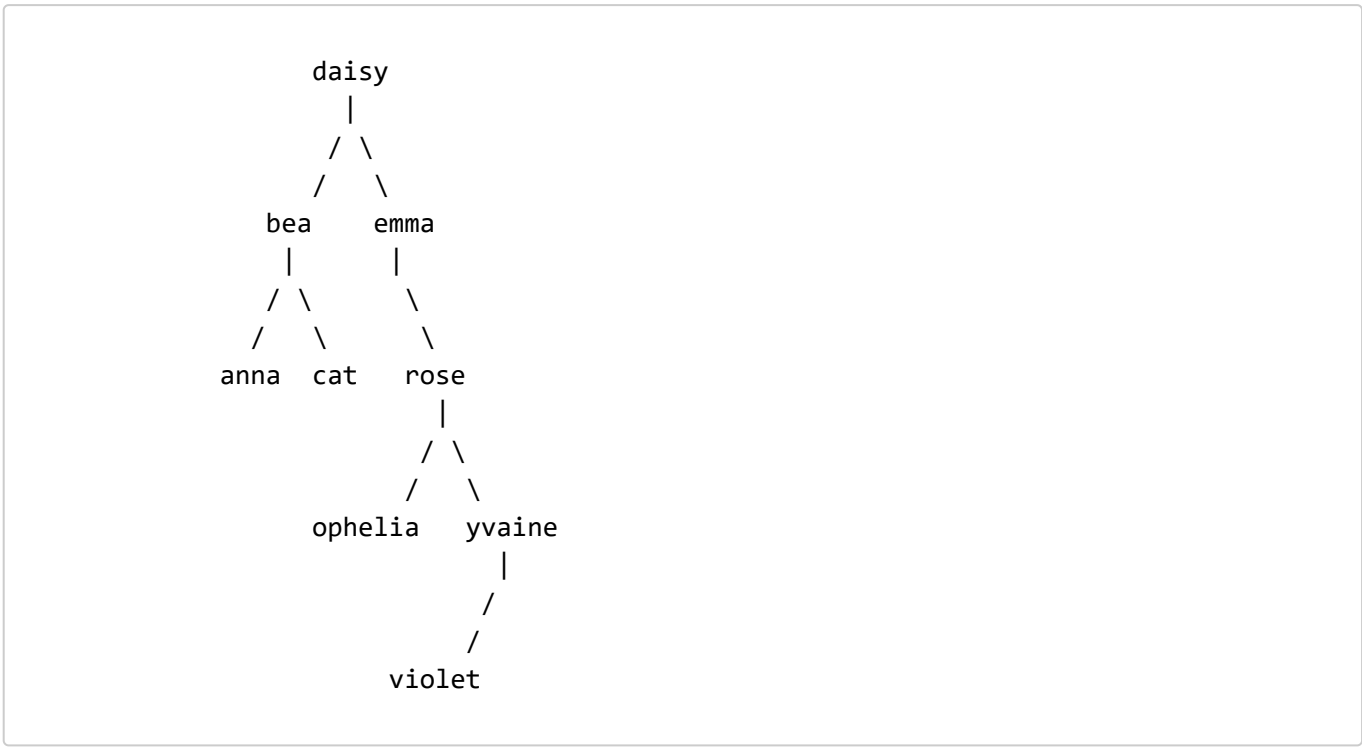
So, we may now update the list to meet specific criteria thereby removing the need to repeatedly sort the dataset.

This is the binary search tree data structure.

A basic tree data structure may be represented as follows,

Binary search tree

A sample binary search tree may be structured as follows



For every node in this tree, the nodes to the left of the current node are smaller. Nodes to the right will now be larger.

If we want to search for `violet`, we begin at the root. We may then use the following path,

- V is after D, so we traverse the right side of the tree
  - current node = emma
- V is after E, so we continue down the right side
  - current node = rose
- V is before Y, so we continue down the left side
  - `violet` node found...

Searching for a node in a binary search tree takes `O(log n)` on average, and `O(n)` for worst cases. A sorted array, by contrast, takes `O(log n)` in worst case scenarios.

Initially, we might consider arrays as the preferable option. However, a sorted binary search tree is, on average, faster for insertions and deletions.

Issues with binary search trees

Binary search trees do not provide random access.

Performance times are averages, and they rely on a balanced tree.

There are specialist trees, which provide self-balancing mechanisms. For example, we might use a *red-black* tree.

Basic logic implementation

A common option for implementing this type of traversal and search is using the *symbol-table API* for a binary search tree.

A symbol-table is also known as a map, dictionary, associative array &c. in various programming languages. The general concept is as follows.

- abstraction of key/value pairs
    - e.g. insert a value with a specified key
    - given the key, search for the corresponding value

Sample usage may include the following,

application	search	key	value
dictionary	search for a definition	word	definition
index	search for a given reference, e.g book page	term	e.g. list of page numbers for a book
compiler	search for props of variables	variable name	type and value

In JavaScript, we may use the `Map` collection to define a dictionary for the Symbol-table.

For example,

```
const symTable = new Map();
```

For the binary search tree logic, we may begin with a custom function to define nodes. This function includes props required for a node, e.g.

- key
- value
- left link
- right link
- node count

Order-based methods

A common reason for working with binary search trees is that they keep the keys in order. So, we may use BSTs in many disparate API contexts to ensure consistent I/O structure. For example, we might consider a custom *symbol-table* API.

Some sample methods and usage,

- `min & max`
    - if left link of root is `null` - smallest key in the BST must be the root node

    - if left link is not `null` - smallest key is in subtree referenced by left link (this repeats for each left link in each subtree...)
- `floor`
    - if a given key is less than the key at the root of the BST - floor of the key must now be added to the left subtree...
    - if key is greater than root - floor can be in the right link but only if there is a smaller or equal existing key - if not, floor of key is root...
- `ceiling`
    - same pattern as floor - except check relative to right link
- `selection`
    - suppose we seek the key of rank `k` (the key such that precisely `k` other keys in the BST are smaller)
    - if the number of keys `t` in the left subtree is larger than `k`, we look (recursively) for the key of rank `k` in the left subtree; if `t` is equal to `k`, we return the key at the root; and if `t` is smaller than `k`, we look (recursively) for the key of rank `k - t - 1` in the right subtree.
- `rank`
    - if the given key is equal to the key at the root, we return the number of keys `t` in the left subtree; if the given key is less than the key at the root, we return the rank of the key in the left subtree; and if the given key is larger than the key at the root, we return `t` plus one (to count the key at the root) plus the rank of the key in the right subtree.
- `delete min & max`
    - for delete the minimum, we go left until finding a node that has a null left link and then replace the link to that node by its right link. The symmetric method works for delete the maximum.
- `delete`
    - proceed in a similar manner to delete a node with one or null childrem
    - for two or more - start by replacing the current node with its successor...
    - successor is the node with the smallest key in its right subtree

We may accomplish the task of replacing `x` by its successor in four easy steps:

1. save a link to the node to be deleted in `t`
2. set `x` to point to its successor `min(t.right)`
3. set the right link of `x` (which is supposed to point to the BST containing all the keys larger than `x.key`) to `deleteMin(t.right)`, the link to the BST containing all the keys that are larger than `x.key` after the deletion.
4. set the left link of `x` (which was null) to `t.left` (all the keys that are less than both the deleted key and its successor).

- `range search`
    - to implement ths `keys()` method - returns all keys in a given range
    - begin with a basic recursive BST traversal method - known as `inorder traversal`
    - e.g. to show this order traversal
        - first print all keys in the left side of the BST - all less than root
        - then print root key
        - then print all keys in the right side of BST
    - `keys()` method
        - define code to add each key that is in the range to a Queue

- skip recursive calls for subtrees that cannot contain keys in the range