10.7 PROJECTS

Project 10.1 Creating a searchable database

Write a program that allows for the interactive search of a data set, besides the earthquake data from Section 10.2, downloaded from the web or the book website. It may be data that we have worked with elsewhere in this book or it may be a new data set. Your program, at a minimum, should behave like the program that we developed in Section 10.2. In particular, it should:

1. Read the data into a table or parallel lists.
2. Sort the data by an appropriate key.
3. Interactively allow someone to query the data (by a key). When a key is found, the program should print the satellite data associated with that key. Use binary search to search for the key and return the results.

Project 10.2 Binary search trees

In this project, you will write a program that allows for the interactive search of a data set using an alternative data structure called a *binary search tree (BST)*. Each key in a binary search tree resides in a *node*. Each node also contains references to a *left child* node and a *right child* node. The nodes are arranged so that the key of the left child of a node is less than or equal to the key of the node and the key of the right child is greater than the key of the node. For example, Figure 1 shows a binary search tree containing the numbers that we sorted in previous sections. The node at the top of the tree is called the tree's *root*.

To insert a new item into a binary search tree, we start at the root. If the new item is less than or equal to the root, we next look at the left child. On the other hand, if the new item is greater than the root, we next look at the right child. Then we repeat this step on the next node, and continue until we arrive at a position without a node. Figure 2 illustrates how the value 35 would be inserted into the binary search tree in Figure 1. Starting at the root, since $35 < 50$, we move to the left. Next, since $35 > 30$, we next move to the right. Then, since $35 < 40$, we move to the left. Since there is no node in this position, we create a new node with 35 and insert it there, as the left child of the node containing 40.

Question 10.2.1 *How would the values 5, 25, 65, and 75 be inserted into the binary search tree in Figure 1?*

Question 10.2.2 *Does the order in which items are inserted affect what the tree looks like?*

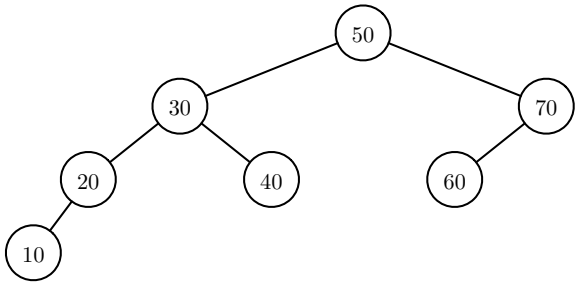


Figure 1 A binary search tree.

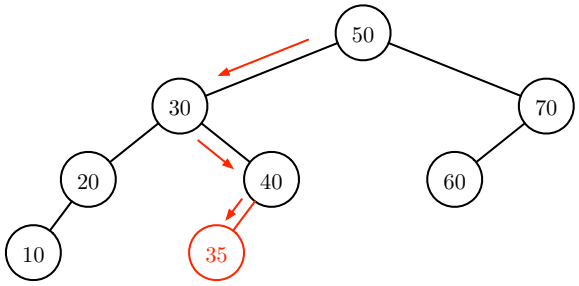


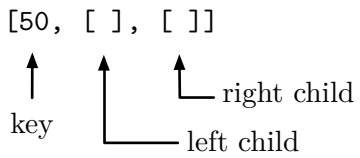
Figure 2 Insertion into a binary search tree.

After the four values in the previous question are inserted into the binary search tree, insert the value 67. Would the binary search tree be different if 67 were inserted before 65?

Searching a binary search tree follows the same process, except that we check whether the target value is equal to the key in each node that we visit. If it is, we return success. Otherwise, we move to the left or right, as we did above. If we eventually end up in a position without a node, we know that the target value was not found. For example, if we want to search for 20 in the binary search tree in Figure 1, we would start at the root and first move left because $20 < 50$. Then we move left again because $20 < 30$. Finally, we return success because we found our target. If we were searching for 25 instead, would have moved right when we arrived at node 20, but finding no node there, we would have returned failure.

Question 10.2.3 *What nodes would be visited in searches for 10, 25, 55, and 60 in the binary search tree in Figure 1?*

In Python, we can represent a node in a binary search tree with a three-item list. As illustrated below, the first item in the list is the key, the second item is a list representing the left child node, and the third item is a list representing the right child node.



The list above represents a single node with no left or right child. Or, equivalently, we can think of the two empty lists as representing "empty" left and right children. To insert a child, we simply insert into one of the empty lists the items representing the desired node. For example, to make 70 the right child of the node above, we would insert a new node containing 70 into the second list:

```
[50, [ ], [70, [ ], [ ]]]
```

To insert 60 as the left child of 70, we would insert a new node containing 60 into the first list in 70:

```
[50, [ ], [70, [60, [ ], [ ]], [ ]]]
```

The list above now represents the root and the two nodes to the right of the root in Figure 1. Notice that an entire binary search tree can be represented by its root node. The complete binary search tree in Figure 1 looks like this:

```
bst = [50, [30, [20, [10, [], []], []], [40, [], []]],
           [70, [60, [], []], []]]
```

Question 10.2.4 *Parse the list above to understand how it represents the binary search tree in Figure 1.*

This representation quickly becomes difficult to read. But, luckily, we will rely on our functions to read them instead of us.

Let's now implement the insert and search algorithms we discussed earlier, using this

list implementation. To make our code easier to read, we will define three constant values representing the indices of the key, left child, and right child in a node:

```
KEY = 0
LEFT = 1
RIGHT = 2
```

So if `node` is the name of a binary search tree node, then `node[KEY]` is the node's key, `node[LEFT]` is the node's left child, and `node[RIGHT]` is the node's right child.

The following function inserts a new node into a binary search tree:

```
def insert(root, key):
    """Insert a new key into the BST with the given root.

    Parameters:
        root: the list representing the BST
        key:  the key to insert

    Return value: None
    """

    current = root
    while current != [ ]:
        if key <= current[KEY]:
            current = current[LEFT]
        else:
            current = current[RIGHT]
    current.extend([key, [ ], [ ]])
```

The variable named `current` keeps track of where we are in the tree during the insertion process. The `while` loop proceeds to "move" `current` left or right until `current` reaches an empty node. At that point, the loop ends, and the algorithm inserts a new node containing `key` by inserting `key` and two empty lists into the empty list assigned to `current`. (Recall that the `extend` method effectively appends each item in its list argument to the end of the list.) To use this function to insert the value 35 into our binary search tree named `bst` above, as in Figure 2, we would call `insert(bst, 35)`.

The function to search a binary search tree is very similar:

```
def search(root, key):
    """Search for a target key in the BST with the given root.

    Parameters:
        root: the list representing the BST
        key:  the key to search for

    Return value: a Boolean indicating whether key was found
    """

    current = root
    while current != [ ] and current[KEY] != key:
        if key < current[KEY]:
            current = current[LEFT]
        else:
            current = current[RIGHT]
    return current != [ ]
```

The only differences in the `search` function are (a) the loop now also ends if we find the desired `key` value in the node assigned to `current`, and (b) at the end of the loop, we return `False` (failure) if `current` ends at an empty node and `True` otherwise.

In this project, you will work with a data set of your choice, downloaded from the web or the book website. It may be data that we have worked with elsewhere in this book or it may be a new data set. Your data must contain two or more attributes per entry, one of which will be an appropriate key. The remaining attributes will constitute the satellite data associated with the entry.

Part 1: Extend the BST implementation

Extend the list representation of a node so that each node can store both a key and associated satellite data. Think about how your new design will affect the `insert` and `search` functions. Then modify these functions so that they work with your new representation.

With this extension, you are actually creating a new data structure that implements a dictionary abstract data type. The dictionary abstract data type in Python defines a way to insert (key, value) pairs and retrieve the value associated with a key using indexing. As explained in Tangent 7.2, Python dictionaries are usually implemented using a data structure called a hash table. In your extended binary search tree data structure, the `insert` function will insert a new (key, value) pair into the dictionary and the `search` function will return the value associated with `key`, if it is found, or `None` if it is not. We will revisit a dictionary implementation in Section 12.5.

Part 2: Read the data

Write a function that creates an empty binary search tree, reads your data from the web or a file, and then inserts each entry into the binary search tree.

Part 3: Allow queries

Write a function, like the `queryQuakes` function from Section 10.2, that allows someone to interactively query your data. The function should prompt for a key, and then print the associated satellite data. To locate this data, search for the key in your binary search tree.

Question 10.2.5 *Is searching a binary search tree as efficient as using the binary search algorithm to search in a sorted list? In what situations might a binary search tree not be as efficient? Explain your answers.*

Write a `main` function that puts all of the pieces together to create a program that reads your data set and allows repeated queries of the data.

Part 4: Recursion

Every node in a binary search tree is the root of a `subtree`. In this way, binary search trees exhibit self-similarity. The subtrees rooted by a node's left and right children are called the node's *left subtree* and *right subtree*, respectively. Exploiting this self-similarity, we can think about inserting into (or searching) a binary search tree with root r as recursively solving one of two subproblems: inserting into the left subtree of r or inserting into the right subtree of r. Write recursive versions of the `insert` and `search` functions that use this self-similarity.

Part 5: Sorting

Once data is in a binary search tree, we have a lot of information about how it is ordered. We can use this structure to create a sorted list of the data. Notice that a sorted list of the keys in a binary search tree consists of a sorted list of the keys in the left subtree, followed by the root of the tree, followed by a sorted list of the keys in the right subtree. Using this insight, write a recursive function `bstSort(root)` that returns a sorted list of the keys in a binary search tree. Then use this function to add an option to your query function from Part 3 that prints the list of keys when requested.

Question 10.2.6 *How efficient do you think this sorting algorithm is? How do you think it compares to the sorting algorithms we discussed in this chapter? (Remember to take into account the time it takes to insert the keys into the binary search tree.)*