600-152 Informatics 2 Lecture Notes. Binary Search Trees.

Bernie Pope. Last updated 29/3/09.

Introduction

In the previous two lectures we looked at indexing data for efficient search in the context of the SearchMe application. Our index was a mapping from terms (strings) to their definitions (strings). We noted that the natural way to represent such an index in Python is to use a dictionary.

In this lecture we are going to consider how we might actually implement dictionaries ourselves. That is, we are going to try to make a data structure which enables efficient search, without relying on the dictionaries which are built into Python.

The reason for doing this is to learn about some useful (and interesting) concepts in algorithms and data structures.

It is important to note that it is not our intention to merely copy the way Python implements dictionaries. Instead, we want to replicate their essential functionality. In practice, our approach is quite different to the one used in the Python interpreter. Ours is a lot simpler, which makes it easier to understand (and teach). Nonetheless, as you will see, our approach is reasonably competitive in its performance, and, in some cases, it is a lot better than linear search.

The essential functionality of dictionaries

Python's dictionaries offer a very rich set of features, but they have four essential operations:

- 1. Creating new empty dictionaries.
- 2. Searching for a key in an existing dictionary and, if found, returning its corresponding value.
- 3. Inserting new key/value mappings into existing dictionaries.
- 4. Modifying the value corresponding to a particular key in an existing dictionary.

Our implementation will provide these four operations.

Actually, Python's dictionaries also allow entries to be deleted, and this may be considered their fifth essential operation. We will not implement deletion in this lecture (it is somewhat complicated, and we will defer the topic to subjects dedicated to the study of algorithms).

Exercise 1

List some *other* operations that can be performed on Python's dictionaries.

The problem with linear search

The problem with linear search is that it fails to take advantage of the underlying structure of our data. This means that a search for a particular key will consider many elements of the data unnecessarily, making it inefficient. As we shall see, Binary Search Trees (BSTs) offer a solution to this problem, providing that the key values can be totally ordered.

What is a BST?

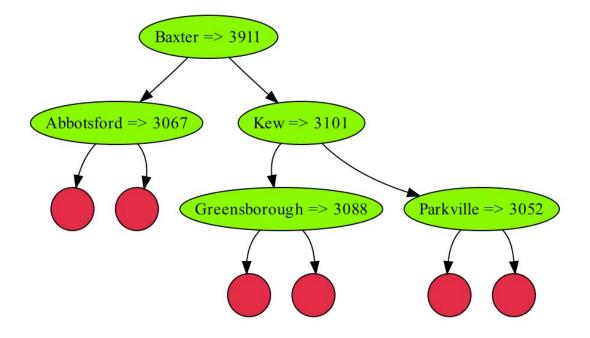
A BST is a tree which is built from two types of nodes:

- 1. Empty nodes.
- 2. Branch nodes.

Empty nodes are rather dull. They do not contain data, and they don't have children. Empty nodes only appear as leaves of the tree. Branch nodes are much more interesting. They do contain data (a key and a value), and they have exactly two children. The children of a branch node are themselves BSTs. The BST defined by the left child is called the "left subtree" of the parent, and the BST defined by the right child is called the "right sub-tree" of the parent.

The most important property of BSTs is that the keys are ordered. In this lecture we will also require the keys to be unique, but some definitions of BSTs will allow duplicate keys.

The ordering of keys is easiest to explain with an example. Here is a diagram of a BST:



Branch nodes are green, and empty nodes are red. The keys are strings representing the names of Melbourne suburbs. The values are numbers representing the postal codes of the corresponding suburbs.

If you look closely at the diagram you will notice that the keys are ordered as they would be in a dictionary book. The technical term for this is **lexicographic ordering**. Consider the root node, which has "Baxter" as its key. Every node in its left sub-tree has a key which is less than "Baxter". Also, every node in its right sub-tree has a key which is greater than "Baxter".

It is important to note that the ordering of keys in a BST applies to *every* branch node, not just the root. For example, consider the node with "Kew" as its key. "Kew" is greater than "Greensborough", and "Kew" is less than "Parkville".

In summary, every sub-tree of a BST must also be a BST. Empty nodes are just trivial cases.

Exercise 2

Extend the above example BST so that it contains the suburb (and its postcode) that you live in. Make sure you put it in the right place.

How to make an empty BST

It is trivial to make a new empty BST, you just construct an empty node. (As noted above, an empty node is a valid BST).

How to find something in a BST

BSTs are easy to search because the keys are ordered. Here is the algorithm in English:

The item that we are searching for is called the **target key**. We traverse the tree one node at a time. At each step, the node that we are considering is called the **current node**. The current node is updated as the algorithm progresses. Initially the current node is the root of the tree.

- If the current node is empty, then the search has failed to find the target key.
- Else, if the current node is not empty, then:
 - If the target key is equal to the key of the current node, then the search has succeeded, and we return the corresponding value of the current node.
 - Else, if the target key is less than the key of the current node, we search in the left sub-tree of the current node. The left child of the current node becomes the new current node.
 - Else, we search in the right sub-tree of the current node. The right child of the current node becomes the new current node.

Here is the search algorithm in Python, which assumes that the following functions are defined: is empty(), key of(), value of(), left child of() and right child of().

```
def search(current_node, target_key):
    while not is_empty(current_node):
        if target_key == key_of(current_node):
            return value_of(current_node)
        elif target_key < key_of(current_node):
            current_node = left_child_of(current_node)
        else:
            current_node = right_child_of(current_node)
        return None</pre>
```

Insertion and Modification

Inserting a new entry into a BST and modifying the value of an existing entry can be done using the same algorithm. The algorithm is similar to search; here it is in English:

The key item that we are inserting (or updating) is called the **target key**. The value item that we are inserting (or updating) is called the **target value**. We traverse the tree one node at a time. At each step, the node that we are considering is called the **current node**. The current node is updated as the algorithm progresses. Initially the current node is the root of the tree.

• If the current node is empty, then construct a new node containing the target key and target value. The children of the new node are empty.

- Else, if the current node is not empty, then:
 - If the target key is equal to the key of the current node, then we update the value of the current node to be equal to the target value.
 - Else, if the target key is less than the key of the current node, we insert/ update in the left sub-tree of the current node. The left child of the current node becomes the new current node.
 - Else, we insert/update in the right sub-tree of the current node. The right child of the current node becomes the new current node.

Here is the insert/update algorithm in Python, which assumes that the following functions are defined: is_empty(), key_of(), value_of(), left_child_of(), right_child_of(), update value(), and update empty node().

```
def insert(current_node, target_key, target_value):
    while not is_empty(current_node):
        if target_key == key_of(current_node):
            update_value(current_node, target_value)
            return
        elif target_key < key_of(current_node):
            current_node = left_child_of(current_node)
        else:
            current_node = right_child_of(current_node)
        update_empty_node(current_node, target_key, target_value)</pre>
```

Exercise 3

Why doesn't the insert function contain a return statement on the last line?

Run-time performance

It is interesting to compare the run-time performance of BSTs against Python's dictionaries and linear search. One way to do this is by experiment. The procedure I chose goes like this:

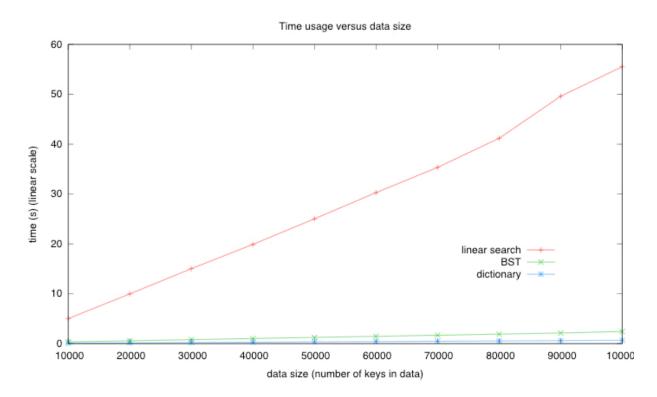
- Implement search in three ways: using BSTs (as described in this lecture), using Python's built in dictionaries, and using linear search over a list of key-value pairs (tuples).
- Create test data files for each case of various sizes.
- Time the execution of each implementation for each data size.
- Plot the results on a graph.

To make it simple, the keys of the data are just numbers (integers), and the values are also numbers. I chose to make the values equal to the square of their corresponding keys, but this detail is rather unimportant. All that really matters is that we have data files of different sizes. To get nice plots on a graph it is useful to make the data files increase in size by

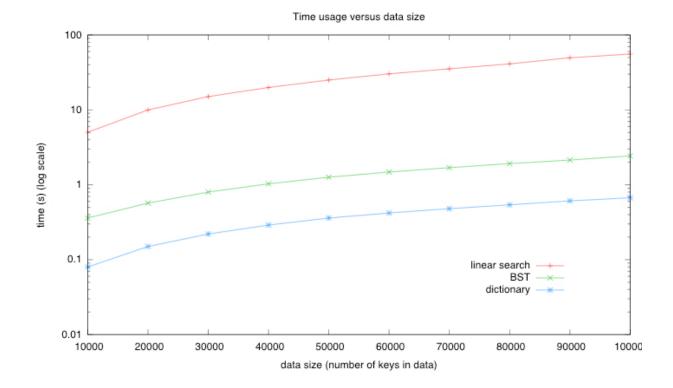
regular amounts. In this instance I chose increments of 10,000 key-value entries. The smallest data file has 10,000 entries, and the largest has 100,000 entries. Each implementation requires the data to be stored in a different way (as a BST, or a dictionary or a list of pairs). It takes time to build each data representation, but we are really only interested in the search time. To avoid counting the time taken to create each data representation, I wrote a separate program to generate the data and then stored a pickled representation in a file. The search programs that I tested just read the pickled data structures from those files. It still takes some time to read the pickled files, but the significance of that was diminished by executing a large number of searches in each test case. In this particular instance, each test case searches for 10,000 unique numbers, which were carefully generated to ensure that they appear in the test data. It may also be useful to time the how long it takes to find keys which are not in the data files, but that is left as an exercise for enthusiastic students.

One very important detail is that I randomised the order in which the key-value pairs are inserted into the data structures. The reason for this is described in the next section.

The graph below shows the results of the tests. The vertical axis plots time in seconds, and the horizontal axis plots the size of the data files (number of key-value entries, which is equal to the number of keys). Clearly BSTs and dictionaries are superior to linear search.



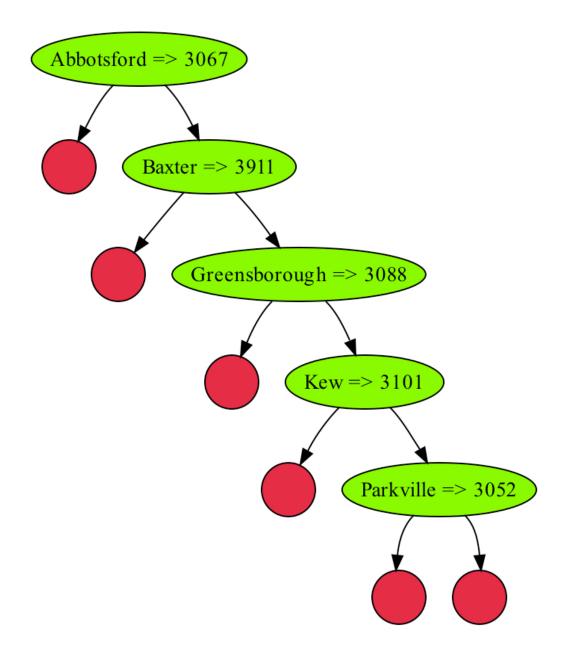
The previous graph makes it difficult to see how BSTs compare with dictionaries, because linear search is so much worse than both of them. We can get a better idea by changing the scale of the vertical axis, from a linear scale to a logarithmic scale. This tends to compress the space at the top of the graph and expand the space at the bottom.



The importance of being balanced

The performance tests above suggest that our BSTs are quite competitive with Python's dictionaries. But they do have one very unfortunate characteristic: they are sensitive to the order in which the key-value entries are inserted.

Consider what happens if we insert the suburb-postcode data from the earlier example into a BST in order sorted on the keys. We end up with this BST:



The result is a valid BST, but every left sub-tree is empty. Trees with this shape are called **sticks**. Searching for data in a stick is comparable to linear search (in fact a stick is just a list in disguise).

In the run-time testing discussed in the previous section it was pointed out that the test data was randomised before being inserted into the BST. Randomising the data makes it very unlikely that we will end up with a stick. If we inserted the data in key-sorted order then the BST would have very poor search performance.

This suggests a serious problem with our BST design: in many cases we cannot predetermine the order in which key-value entries are inserted into the BST.

This problem has been known for a long time, and has caused a lot of research to be conducted into more robust search tree implementations. One of the most important observations is that BSTs give good search performance when they are **balanced**. There are various definitions of what it means to be balanced, but in rough terms it means that the left and right sub-trees of each node are roughly the same size.

Maintaining balance in a BST is beyond the scope of Informatics 2, but the topic is covered in later year Algorithms and Data Structures subjects in Computer Science.

If you are curious to learn more, the following Wikipedia entry is a good starting point:

Self-balancing binary search trees

Implementation details

We now turn our attention to how we might actually represent a BST in Python using the features of the language that you already know.

We will represent the nodes of the BST using lists. Empty nodes are represented by empty lists. Branch nodes are represented with lists of length 4. The first and last elements of the list contain the children of the node. The second and third elements of the lists contain the key and value.

For example, the following list represents a single branch node with 'Greensborough' as the key, 3088 as the value, and empty nodes for the left and right child.

```
[[], 'Greensborough', 3088, []]
```

The complete BST described in the 'What is a BST?' section above is implemented by this list:

```
[[[], 'Abbotsford', 3067, []], 'Baxter', 3911, [[[], 'Greensborough',
3088, []], 'Kew', 3101, [[], 'Parkville', 3052, []]]]
```

Exercise 4

Write down the Python list which represents the stick BST from the previous section of these lecture notes.

We already have the <code>search()</code> and <code>insert()</code> functions defined, so it is just a matter of filling in the details. Below is a complete implementation of a BST in Python using lists to represent the nodes:

```
# list index values for accessing the components of a branch
node
left child index = 0
key index = 1
val index = 2
right child index = 3
# build an empty bst
def empty():
    return []
# search for a key in a bst
def search (current node, target key):
    while not is empty(current node):
       if target key == key of(current node):
           return value of (current node)
       elif target key < key of(current node):
           current node = left child of(current node)
       else:
           current node = right child of(current node)
    return None
# insert a new entry into a bst or update an existing one
def insert(current node, target key, target value):
    while not is empty(current node):
       if target key == key of (current node):
           update value(current node, target value)
           return
       elif target key < key of(current node):</pre>
           current node = left child of(current node)
       else:
           current node = right child of(current node)
    update empty node (current node, target key, target value)
# build a bst from a list of (key, value) pairs
def from list(list):
    t = empty()
    for key, val in list:
        insert(t, key, val)
    return t
# test if a bst is empty
```

```
def is empty(bst):
   return len(bst) == 0
# get the key from a branch node
def key of (branch node):
    return branch node[key index]
# get the value from a branch node
def value of (branch node):
   return branch node[val index]
# get the left child of a branch node
def left child of (branch node):
    return branch node[left child index]
# get the right child of a branch node
def right child of (branch node):
   return branch node[right child index]
# update an empty node with a key and a value
# thus turning it into a branch node
# the children of the new branch node are empty
def update empty node (node, key, value):
   node.extend([empty(), key, value, empty()])
# update the value stored in a branch node
def update value (branch node, value):
   branch node[val index] = value
# example association list of suburb to postcodes
suburb code = [('Baxter', 3911), ('Kew', 3101), ('Abbotsford',
3067), ('Parkville', 3052), ('Greensborough', 3088)]
# an example BST made from the suburb code list
example tree = from list(suburb code)
```