UNIVERSITY OF SASKATCHEWAN

Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephone: (306) 966-4886, Facimile: (306) 966-4884

CMPT 145
Winter 2016-17
Principles of Computer Science

Assignment 6

Primitive Binary Search Trees, and Tables – Solutions

Date Due: March 16, 2017, 7pm **Total Marks: 47**

Question 1 (10 points):

Purpose: To work with primitive binary search trees.

Degree of Difficulty: Easy

On Moodle you can find the file `primbst.py`, which has the code for primitive binary search trees that we covered in Lecture 13. Read the file carefully, especially the comments at the top of the file. There are 2 to-do items, namely to implement the function `delete_prim`, and do some testing of it. The function has the following interface:

```python
def delete_prim(tnode, value):
    """
    Delete a value from the binary tree.
    Preconditions:
        :param tnode: a binary search tree, created by create()
        :param value: a value
    Postconditions:
        If the value is in the tree, it is deleted.
        If the value is not there, there is no change to the tree.
    Return:
        :return: (True, tnode) is the value was deleted, tree changed
                 (False, tnode) otherwise (tnode unchanged)
    """
    # TODO: complete this function
    #       See BST Lecture slides
    return False, tnode
```

Currently the function does nothing. It should search for the value in the tree starting at `tnode`, and if it is there, should reconnect the tree so that the tree rooted at `tnode` no longer contains the value to be deleted. Notice that the return value is a pair; the first value in the pair indicates whether the value was there, or not. The second value in the returned pair is the resulting tree, changed (if the value was deleted) or unchanged (if the value was not in the tree).

Using the algorithms from Lecture 13, complete the function `delete_prim`.

The starter file has a lot of testing code in it, for your use in testing `delete_prim`.

What to Hand In

Submit a file named `a6q1.py` containing all the functions and testing, including your code for `delete_prim`. Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 3 marks: You submitted an implementation of `delete_prim`, which looks like a reasonable attempt.
- 7 marks: Your implementation allows the testing to run without error.

Solution: Here's a pretty literal translation of the algorithms:

```
def delete_prim(tnode, value):
    """
    Delete a value from the binary tree.
    Preconditions:
        :param tnode: a binary search tree, created by create()
        :param value: a value
    Postconditions:
        If the value is in the tree, it is deleted.
        If the value is not there, there is no change to the tree.
    Return:
        :return: (True, tnode) is the value was deleted, tree changed
                 (False, tnode) otherwise (tnode unchanged)
    """

    def delete_bst(tnode):
        if tnode is None:
            return False, tnode
        else:
            cval = tn.get_data(tnode)
            if cval == value:
                return reconnect(tnode)
            elif value < cval:
                flag, subtree = delete_bst(tn.get_left(tnode))
                if flag:
                    tn.set_left(tnode,subtree)
                return flag, tnode
            else:
                flag, subtree = delete_bst(tn.get_right(tnode))
                if flag:
                    tn.set_right(tnode,subtree)
                return flag, tnode

    def reconnect(delthis):
        if tn.get_left(delthis) is None \
                and tn.get_right(delthis) is None:
            # the deleted node has no children
            return True, None
        elif tn.get_left(delthis) == None:
            # the deleted node has one right child
            return True, tn.get_right(delthis)
        elif tn.get_right(delthis) == None:
            # the deleted node has one left child
            return True, tn.get_left(delthis)
        else:
            # the deleted node has 2 children
            left = tn.get_left(delthis)
            right = tn.get_right(delthis)
            walker = left
            # walk all the way to the right from left
            while tn.get_right(walker) != None:
                walker = tn.get_right(walker)
            tn.set_right(walker,right)
            return True, left

    flag, tree = delete_bst(tnode)

    return flag, tree
```

This is of course not the only way to accomplish deletion, even given the strong guidance of the pseudo-code.

Notes for markers:

- The model solution uses two internal functions. That's literal, but not necessary.
- The model solution uses a function called `reconnect`, but students can name it something else. The model solution returns a tuple, but since `reconnect` always affects the tree, the first value is always `True`. Returning a tuple from reconnect is not absolutely necessary; returning the tree can be quite acceptable.
- The `delete_prim` should return a tuple, as it is needed by the testing.
- To grade this question, open the `a6q1.py` file, and check that the implementation looks okay. Give 3 marks if the program looks reasonable.
- Run the code (the command-line is probably most convenient).
  1. If there are no errors, give 7/7.
  2. If there Integration errors, but no Unit error, give 3/7.
  3. If there are Unit errors and Integration errors, give 0/7.

Question 2 (10 points):

Purpose: To exercise the notions of test coverage.

Degree of Difficulty: Moderate

The starter file for Question 1 has a lot of testing code in it. If you haven't studied it closely, take the opportunity now. Answer the following questions:

1. There are three functions whose name begins with `unit`, and whose name includes one of the primitive BST function names. For each of these test functions, decide whether it actually tests what the function does. Write a brief (2 or 3 sentence) comment for each of them.

   Solution: member is fine, since it only returns one value, and the test function checks it against what is expected. The other two primitive BST functions return a pair of values, and the unit test functions only test one of them. This is a deficiency, but in my opinion, not a colossal deficiency, because it would be hard to test the resulting primitive tree, without using one of the other primitive tree functions.

   Note for markers: Give full marks for any answer that recognizes that the unit test functions don't actually look at the resulting tree. Any opinion on that issue is acceptable as well.

2. The three `unit` functions are called multiple times on three kinds of trees. Consider whether the tests, taken together, provide adequate coverage of the primitive BST functions. For each of the functions `member_prim`, `insert_prim`, and `delete_prim`, write a brief comment (2 or 3 sentences) about the test coverage.

   Solution: It's good that three kinds of initial trees are being tested: empty, a single leaf, and a tree with 3 nodes and 2 leaf nodes. However, it would be better to have a tree with more varieties of nodes, e.g., a node with a left child, but no right child, and a node with a right child, but no left child. This kind of tree would test some of the special cases more closely.

   As for coverage, the unit tests try values that are in the tree, and values that are not in the tree. For the tree with 3 nodes, the testing looks for values on every possible subtree. The test coverage seems pretty good.

   Note for markers: Give full marks for an answer that suggests an understanding of test coverage. Some responses my give a count of tests, or lines tested, or maybe just a comment about "enough " or not "enough." Some students may differ on their opinion about coverage. Their opinion really shouldn't be considered correct or incorrect, as long as they seem to have given the matter a little thought. I'm hoping students will notice the weak set of example trees, but no penalty if they don't.

3. There is a rather larger function named `integration` in the starter file. It uses the primitive BST functions together in an integration test. You will notice that the integration test inserts and deletes values, but never deletes a value immediately after inserting it. Do you think the integration is adequate, or should some code be added to test deletion immediately after insertion? Give a brief (2 or 3 sentence) answer.

   Solution: It's probably okay not to test deletion immediately after insertion, especially if the test script produces no errors. But it's not too hard to add it in.

   Note for markers: Students may reply with either an assertion that the testing should include deletions after insertions, or a statement that the current testing is good enough. Full marks for any response that seems to have given the matter some thought.

4. The integration test is called several times, on data of various sizes. Do you think this is enough? Give a brief (1 or 2 sentence) answer.

> **Solution:** The testing includes random and sorted data, and quite a large number of nodes. We can probably be confident that the functions work, until someone finds a bug we didn't test for.
>
> **Note for markers:** The only thing to watch out for any response that suggests that the testing proves correctness absolutely. Deduct 2 marks only for believing that testing can assure us of the absence of errors.

What to Hand In

The brief answers to the questions above, in a file named `a6q2.txt` (other formats, such as PDF, RTF, DOC, DOCX are acceptable).

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 3 marks: Part 1: you understand what the test function does, and you can identify the weaknesses in the test functions.
- 3 marks: Part 2: you understand what the test script is testing, and you understand the notion of test coverage.
- 2 marks: Part 3: you have expressed common sense.
- 2 marks: Part 4: you have expressed common sense.

Question 3 (19 points):

Purpose: To adapt some working code to a slightly modified purpose.

Degree of Difficulty: Moderate

In this question, you will adapt the `primbst.py` file to use the key-value treenode ADT. You can start by making a copy of your solution to Question 1. **You'll hand in both versions, Q1 and Q3, in separate files.**

As we discussed in class, the `kvtreenode` ADT is variant of the `treenode` ADT. The key-value treenode allows us to organize the data according to a key, and store a data value associated with it. You can find the implementation of key-value treenode ADT in file `kvtreenode.py`.

Adapt the primitive BST functions from Question 1 to use the `kvtreenode` ADT. The nodes in the tree should have the binary search tree property on the keys, not the values. The functions you need to adapt are as follows:

member_prim(t,k) Returns the pair `True, v`, if the key `k` appears in the tree, with associated value `v`.

insert_prim(t,k,v) Stores the value `v` with the key `k` in the Table `t`.
14/03/2017: Clarified the behaviour of `insert_prim`:

- If the key is already in the tree, the value `v` replaces the current value associated with it. Returns the pair `(False, t)` even though `t` did not change structure in this case.
- If the key is not already in the tree, the pair `k,v` is added to the tree. In this case the function returns the pair `(True, t)`, in which case the tree `t` has changed.

delete_prim(t,k) If the key `k` is in the tree `t`, delete the key-value pair from the tree and return the pair `(True, t)` where `t` is the changed treenode; return the pair `(False, t)` if the key `k` is not in the tree `t`.

To be effective, you'll want to modify the testing code as well. Take care to modify the tests so that the differences in the interface are reflected in the test code.

Hint: Change the interface for each function first. Once you have the interface figured out, adapted the primitive BST functions. Once you have those figured out, adapt the testing script. You'll need to add another argument to every call for insert, giving a value to store with a key. It might be best if you store strings for values, rather than numbers, so that you can tell if your functions actually work. Something like `str(key)+'-value'`, so that you can know which key it should be related to, but so that you can't by accident look at a value and mistake it for a key.

What to Hand In

Your implementation of the primitive BST functions adapted to use kvtreenodes, and the test script adapted to your functions, in a file named `a6q3.py`.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 3 marks: `member_prim` is correctly adapted.
- 3 marks: `insert_prim` is correctly adapted.
- 3 marks: `delete_prim` is correctly adapted.
- 10 marks: The test script is correctly adapted to the new interface of the functions.

Solution: This is a straight-forward exercise, made difficult only by normal human error, and time. See the file `kvprimbst.py`.

Some students may have misunderstood the purpose of the key-value nodes, and may have modified `member` to check for equality of value as well as (or instead of) the key.

The hardest part of the question was modifying the test script. There are two ways this could be done acceptably. One is to modify every part of the script to account for a key and value being used. That would have been a lot of work, but it's fine. The second way is as shown in the model solution, in which the values are based on the keys, for testing purposes. This approach required a bit more thought, but less actual programming.

Note for markers:

- The functions should all return tuples. Full marks for using the key to search in all three functions.
- Testing needs to account for keys and values. Give full marks if the script runs, even if it has errors produced by the script's console output.
- If the test script produces a run-time error, deduct 5 marks.

Question 4 (8 points):

Purpose: To turn the primitive binary search tree functions into a practical ADT.

Degree of Difficulty: Easy

In this question, you'll implement the full Table ADT, as we discussed in class. The Table ADT has the following operations:

create() Creates an empty Table.

size(t) Returns the number of key-value pairs in the Table `t`.

is_empty(t) Returns `True` if the Table `t` is empty, `False` otherwise.

insert(t,k,v) Stores the value `v` with the key `k` in the Table `t`

retrieve(t,k) If the key `k` is in the Table `t`, return the associated value; otherwise return False.

delete(t,k) If the key `k` is in the Table `t`, delete the key-value pair from the table and return True; return false if the key `k` is not in the Table `t`.

A starter file has been provided for you, which you can find on Moodle named `Table.py`. The `create` and `retrieve` functions are already implemented for you, and all of the others have an interface and a trivial do-nothing definition. In addition, a test script exercises the Table operations.

To complete this question, you should import your solution to Question 3 into the Table ADT. Your ADT operations can call the primitive BST functions. Your Table operations may have to do a few house-keeping items, but most of the work is done by your solution to Question 3.

What to Hand In

Your implementation of the Table ADT in a file named `a6q4.py`.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 1 mark: `size`: Your implementation is correct.
- 1 mark: `is_empty`: Your implementation is correct.
- 3 marks: `insert`: Your implementation correctly adds a key-value pair to the table.
- 3 marks: `delete`: Your implementation correctly removes a key-value pair from the table.

Solution: This is a straight-forward exercise.

See the file `Table2.py`.