Institut for Matematik og Datalogi
Syddansk Universitet, Odense

07. April 2025

Project: Part II

DSK814

This project consists of three parts: each part has its own deadline, so that the work is spread over the entire semester. The deadline for part II is *Sunday, April 27th at 11:59 p.m.* The three parts I/II/III are not equal in size, but are approximately 15/25/60 in size.

The project must be completed in groups of two or three.

Goal

The overall goal of the project is training the transfer of the course's knowledge of algorithms and data structures to programming. The project and the written exam complement each other, and the project is not intended as preparation for the written exam.

The specific goal of Part II is to implement the data structure *ordered dictionary* and use it to sort numbers. The work will serve as preparation for Part III of the project.

Task 1

In short, the task is to transfer the book's pseudo-code for unbalanced search trees to a Python program. Your program should be called `DictBinTree.py` and must implement the data structure *binary search tree* with numbers as keys. You do this by creating a Python class `DictBinTree`, which offers the following method calls on an object `T` of the type `DictBinTree`:

- `T.search(k)`, which returns a Boolean indicating whether the key `k` is in the tree `T`

- `T.insert(k)`, which inserts the key `k` in the tree `T`
- `T.orderedTraversal()`, which returns a list of the keys in the tree `T` in sorted order (rather than printing them to the screen as in the book's pseudo-code)

New trees are created with a call `DictBinTree()`, which should return an empty tree.

The implementation must follow the description and pseudo-code in Cormen et al., 4. Edition kapitel 12. You only need to implement only insertion (pseudo-code page 321 (294)), search (pseudo-code page 316 (290/1)), and inorder traversal (pseudo-code page 314 (288)) The implementation *must* be based on this pseudo code. The tree should not be balanced (do not use methods from chapter 13).

You need to define a separate class `BinNode` to represent nodes in trees (you should *not* use an array structure to represent the tree, such as for a heap in Part I of the project). An object of type `BinNode` must contain two other `BinNode`'s (the node's left child and right children), with value `None` if that child does not exist. It must must also contain a key `k`, but in this part does not need to contain the parent.

An object of type `DictBinTree` must contain the `BinNode` that is the root of the tree (if the tree is empty, this value is `None`). This means that a `DictBinTree` object and its root correspond to T and $T.root$ in the book. See Cormen et al., 4. Edition: Figure 10.6 (page 266) [Cormen et al., 3. Edition: Figure 10.9 (page 247)] for an illustration.

For each tree, there is exactly one object of type `DictBinTree` and zero or more objects of type `BinNode`. With `insert`, one new object of type `BinNode` must be created.

Note that you should *not* represent the tree in one big list as in part I of the project (where the navigation in the tree could be done by calculations on list indexes). This method only works method only works for highly balanced trees.

For internal use in the program, there will be a need to implement features in addition to the above. For example, recursion-based functions will need to have *two* versions: the "public" function described above at the beginning of this section, and a function that does the real work (and corresponds to the pseudo-code). The first one is not recursive, but simply calls the second with relevant values as parameters (e.g. that the node being called is the root of

the tree). For iterative functions, these parameters can simply be created before the loop starts. In an inorder traversal, a list will need to be passed as an argument. Instead of printing a node element in an inorder traversal, you should add the element to the list with `append()`.

Remember to test your program thoroughly (including testing on empty trees, testing key insertion, as well as testing for searching for both existing and existing and non-existing keys) before moving on to the next task.

Task 2

You need to implement a sorting algorithm called TREESORT based on the the functions in the program `DictBinTree.py`. The algorithm consists of making repeated `insert`'s in a dictionary, followed by a call to `orderedTraversal`. The numbers returned from this call should simply be printed.

The algorithm should be implemented in a program called `Treesort.py`. This program should use the functions from your program `DictBinTree.py` developed above.

Analogously to the provided program `PQSort.py` from Part I of the project, the program `Treesort.py` reads via the file object `sys.stdin` from the standard input (which by default is the keyboard) and write to the standard output (which by default is the screen). The input to `Treesort.py` is a sequence of `char`'s consisting of integers separated by newlines, and as output the program writes the numbers in sorted order, separated by newlines. Consider the following example of calling `Treesort.py` in a command prompt:

```
python Treesort.py
34
645
-45
1
34
0
Control-D
```

(Control-D indicates end of data under Linux and Mac, under Windows use

Ctrl-Z and then Enter) This should give the following output on the screen:

```
-45
0
1
34
34
645
```

Using *redirection*[1] of standard input and output, you can use *the same* program also on files like this:

```
python Treesort.py   < inputfile   > outputfile
```

To test `Treesort.py` you can run it on the test files from Part I.

An important reason why you should test the above method (with redirection in a command prompt) is that the programs must be able to be be tested automatically after delivery.

Formalities

You only need to submit your Python source files. Make sure that your code is well commented. It must contain the names and SDU logins of the group members.

The file must be submitted electronically in `itslearning` in the Project folder under the Resources tab. The submission module is also inserted into a `itslearning` plan in the course. The files must be submitted either as individual files or as onezip-archive (with all files at the top level, i.e. without any directory structure).

During the submission, you must declare the group by stating the names of all members. You only need to submit once per group. Please note that during the submission, you can create temporary "drafts", but you can only submit once.

Submit by:

[1] Please read about redirection at William Shotts' website (with more details here), Wikipedia or Unix Power Tools

Sunday, April 27th at 11:59 p.m.

Please note that submitting someone else's code or text, whether copied from fellow students, from the internet, or in other ways, is exam cheating, and will be treated very seriously according to current SDU rules. You will also not learn anything. In short: only people whose names are mentioned in the submitted file may have contributed to the code.