Linköping University
Department of Science and Technology/ITN
Aida Nordman

TND004: Data Structures

Lab 3

Goals

To implement a binary search tree class supporting bidirectional iterators.

Prologue

You are requested to add extra functionality to the (template) class `BinarySearchTree` presented in the course book, section 4.3. The code for this class can be downloaded from the course website. One of the major extensions requested is to add a class that represents bidirectional iterators for class `BinarySearchTree`.

In seminar 2, three possible ways to implement a bidirectional iterator class for BSTs are presented. In this lab, every tree's node will be extended to accommodate a pointer to its parent and the implementation of bidirectional iterators should then use the parent pointer stored in the nodes. This is one of the three solutions presented in seminar 2.

More concretely, this lab consists of four exercises.

- **Exercise 1**: to add to the class `BinarySearchTree` a member function (named find_pred_succ) that finds a pair of values belonging to the tree that correspond to the closest predecessor and successor of a given value x.
- **Exercise 2**: to equip class `BinarySearchTree` with bidirectional iterators.
- **Exercise 3**: check whether class `BinarySearchTree` satisfies the requirements with respect to iterators' validity.
- **Exercise 4**: to implement a frequency table of words by using an instance of class `BinarySearchTree`.

During this lab, it is required to modify the code for the given class `BinarySearchTree`. You should only modify non-public member functions (e.g. modify the parameters list). Public member functions parameters should not be modified.

Preparation

You can find below a list of tasks that you need to do before the **HA** lab session on week 18.

1. Review lecture 7, where binary search trees were introduced.
2. Read sections 4.1 to 4.3 of the book. Pay special attention to section 4.3 (you can safely skip reading section 4.3.6 for this lab).
3. Review the notes of seminar 2. The description of this lab may very likely seem quite obscure if you have not attended the seminar and reviewed its notes.

- Downloaded the code for the (template) class `BinarySearchTree` from the course website. Study the given class. This class is also described in section 4.3

of the course book. It is possible to test the given class with the program given in `test1.cpp`. The expected output is provided in the file `test0_out.txt`.

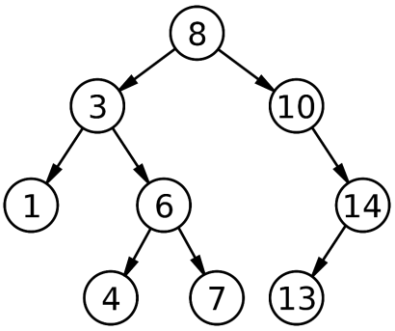
4. Do exercise 1.

Recall that the implementation of `BinarySearchTree` member functions should be added to the header (`.h`) file, since `BinarySearchTree` is a template class.

In the beginning of the **HA** lab session on week 18, the lab assistant will give feedback on your solution for exercises 1 and 2. The remaining exercises of this lab build upon these two exercises.

Exercise 1: To find the predecessor and the successor of a given value x

The aim of this exercise is to add to class `BinarySearchTree` a public member function `find_pred_succ` that, given a value x, returns two values `a` and `b`, such that `a` is the largest value stored in the tree smaller than x and `b` is the smallest value stored in the tree larger than x. The values `a` and `b` are called the **predecessor** and **successor** of x, respectively. For instance, consider the binary search tree (BST) below.



Then,

- `find_pred_succ(7, a, b)` should return `a = 6` and `b = 8`, while
- `find_pred_succ(12, a, b)` should return `a = 10` and `b = 13`.

As discussed in seminar 2, the function `find_pred_succ` can be implemented by calling the auxiliary functions `find_successor` and `find_predecessor`. These two functions, `find_successor` and `find_predecessor`, are also useful in the implementation of bidirectional iterators for class `BinarySearchTree` and they both use the parent pointer to find the successor and the predecessor of a node, respectively.

For this first exercise, perform the tasks listed below, by the indicated order.

- Add to each node a pointer to the parent node. Make the necessary changes in the given code for the insertion/removal of a value in the tree. Note that the private member function `clone`[1] also needs to be modified.
- Add a private member function to display the tree using pre-order[2] with indentation, as shown in the example `test1_out.txt`. This function should then be called from the public member function `printTree`.

[1] The private member function `clone` is called by the copy constructor to perform the actual copying work.
[2] The given code uses an in-order traversal of the tree and, consequently, it displays all values stored in the tree in increasing order.

- Add a (test) public member function `get_parent` that, given a value x, returns the value stored in the parent of the node storing x, if a node storing value x is found and it has a parent node. Otherwise, the default constructor `T()` is returned, where `T` is the type of x.
- Test your code with the program given in `test1.cpp`. Make sure `PHASE 6` test code in the `main` is uncommented. The expected output is provided in the file `test1_out.txt`.
- Add a private member function `find_successor` that returns a pointer to the node storing the successor of the value stored in a given node `t`. If a successor does not exist in the tree then `nullptr` is returned.

```
Node* find_successor(Node* t);
```

- Add a private member function `find_predecessor` that returns a pointer to the node storing the predecessor of the value stored in a given node `t`. If a predecessor does not exist in the tree then `nullptr` is returned.

```
Node* find_predecessor(Node* t);
```

- Add to the class `BinarySearchTree` a member function `find_pred_succ` that, given a value x, returns two values `a` and `b`, such that `a` is the predecessor of x and `b` is the successor of x.

```
void find_pred_succ(const Comparable& x, Comparable& pred, Comparable& suc) const;
```

- Test your code with the program given in `test2.cpp`. The expected output is provided in the file `test2_out.txt`.

Exercise 2: To add a bidirectional iterator class

The aim of this exercise is to equip class `BinarySearchTree` with a public class named `BiIterator` that represents bidirectional iterators for BSTs.

Perform the tasks listed below, by the indicated order.

- Add class `BiIterator` as a public nested class of `BinarySearchTree`. Overload the usual iterator operators: `operator*`, `operator->`, `operator==`, `operator!=`, pre and pos-increment `operator++`, and pre and pos-decrement `operator--`. The class `BiIterator` should also have a default constructor and a constructor to create a bidirectional iterator given a pointer to a tree's node.
- Add two public member functions of class `BinarySearchTree`, `begin` and `end`. Function `begin` returns a `BiIterator` to the node storing the smallest value of the tree, while `end` returns `BiIterator()`.
- Modify the member function `BinarySearchTree::contains` such that it returns a `BiIterator` (instead of a `bool`).
- Test your code with the program given in `test3.cpp`. The expected output is provided in the file `test3_out.txt`.

Exercise 3: Iterator validity

The reference manual of C++ states the following, for container `std::map`[3].

- "*References and iterators to the erased elements are invalidated. Other references and iterators are not affected.*"

Does your implementation of class `BinarySearchTree` satisfy the requirement above? Motivate your answer.

Exercise 4: build a frequency table of words

In this exercise, you are requested to write a program that displays a frequency table for the words in a given text file. For simplicity, you can assume that all words in the file consist of lower-case words and there are no punctuation signs. This problem was part of the last lab in the TNG033 course. There you used the container `std::map`.

In this lab, you must use a binary search tree, i.e. an instance of class `BinarySearchTree`, to represent the frequency table and use bidirectional iterators (i.e. instances of class `BinarySearchTree::BiIterator`) to traverse the tree. A BST is the data structure usually used to implement `std::map`.

Each row of the frequency table contains a key (`string`) and a counter. Thus, you need first to define a class that represents each row of the table. This class should also overload `operator<`.

Test your code for this exercise with the file `words.txt`. The expected frequency table is available in the file `frequency_table.txt`.

Presenting solutions and deadline

You should demonstrate your solution orally during your **RE** lab session on **week 19**. If you fail to get this lab approved on week 19 then this lab will be considered a late lab. At most, one late lab can be presented in a **RE** lab session. Moreover, labs cannot be presented during **HA** lab sessions and we only accept to do "*redovisining*" of lab 3 in the first or second lab session of week 21, provided there is time.

Necessary requirements for approving your lab are given below.

- Use of global variables is not allowed, but global constants are accepted.
- Readable and well-indented code. Note that complicated functions and over-repeated code make programs quite unreadable and prone to bugs. Always check your code and perform code refactoring whenever possible
- There are no memory leaks. Recall that you can use one of the tools suggest in the appendix lab 2.
- The only modifications allowed to the public interface of class `BinarySearchTree` are the ones explicitly described in this exercise. However, you are allowed to add extra private members to the class, if needed.

If you have any specific question about the exercises, then send us an e-mail. Be short and concrete, otherwise you won't get a quick answer. You can write your e-mail in Swedish. Add the course code to the e-mail's subject, i.e. "`TND004: ...`".

[3] The reason to mention the container `std::map` is that it is usually implemented with a binary search tree.

Lycka till!