Linköping University
Department of Science and Technology/ITN
Aida Nordman

TND004: Data structures

Lab 3

Goals

To implement a binary search tree (BST) 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 the seminar, three possible ways to implement an 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 the seminar.

More concretely, this lab consists of four exercises.

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

During this lab, it is required to modify the code for the given class `BinarySearchTree`. You can modify non-public member functions. Public member functions parameters should not be modified, though their body can be modified.

Similar to lab 1 and 2, assertions are used to help testing your code.

Write efficient functions, i.e. functions that visit as few tree nodes as possible.

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: ...`".

Counting number of nodes

As discussed in the course, a (template) class `Node` is defined to represent a node of a BST. Each node stores an element and two pointers: a pointer to the left child node and a pointer to the right child node. In addition, class `Node` keeps track of the number of existing nodes, through a static member variable (named `count_nodes`) that counts the total number of existing nodes. The `Node` constructors increment the counter for the number of nodes, while the destructor decrements the counter.

A (static) member function named `get_count_nodes()` of class `BinarySearchTree` returns the total number of existing nodes. This function is used in the test code to help detecting possible memory leaks through the use of assertions, similar to lab 2.

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 seminar notes. The description of this lab may very likely seem quite obscure if you have not attended the seminar and reviewed its notes.

- Download the code from the course website and create a project with the following files:
  - `BinarySearchTree.h` `node.h`
  - `dsexceptions.h` `test1.cpp`
- Study the template classes `Node` and `BinarySearchTree`. These classes are 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`.

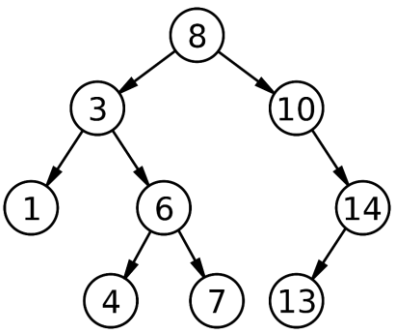
4. Do exercise 1.

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

If you want to have feedback about your code during the **HA** lab session on week 18, then you must submit your preliminary code via Lisam until **April 28th, 08:00 sharp**. Recall that any submitted files via Lisam must include the name and LiU-id of all group members. If you submit code until the given deadline then you will be invited for a 20 minutes Zoom meeting during the scheduled HA lab session.

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 a pair values `<a , 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)` should return `<6, 8>`, while
- `find_pred_succ(12)` should return `<10 , 13>`.

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

- Add to each node a pointer to the parent node. Then, 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` (instead of the current function which performs an in-order traversal of the tree).
- 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 object `Comparable{ }` is returned, where `Comparable` is the type of x.
- Test your code with the program given in `test1.cpp`. Make sure `PHASE 5` and `PHASE 7` of the code in the `main` are uncommented. The expected output is provided in the file `test1_out.txt`.
- Add to the class `BinarySearchTree` a member function `find_pred_succ` that, given a value x, returns a pair of values representing the predecessor of x and the successor of x.

  ```
  std::pair<Comparable, Comparable> find_pred_succ(const Comparable& x) const;
  ```

- Test that your code passes the tests given in `test2.cpp` (i.e. no assertion fails).

Feel free to add other tests to the test programs `test1.cpp` and `test2.cpp`. However, during redovisning you should use the original programs without modifications.

Exercise 2: To add a bidirectional iterator class

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

Perform the tasks listed below, by the indicated order.

- Add a private member function named `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. This function is useful to implement the increment operators (`operator++`) of the `Iterator` class.

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

- Add a private member function named `find_predecessor` that returns a pointer to the node storing the predecessor of the value stored in a given node

[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.

`t`. If a predecessor does not exist in the tree then `nullptr` is returned. This function is useful to implement the decrement operators (`operator--`) of the `Iterator` class.

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

- Add the file `iterator.h` to the project. Then, add the definition of the template class `Iterator`. As discussed in seminar 2, instances of this class should store a pointer to a node. Overload the usual iterator operators: `operator*`, `operator->`, `operator==`, `operator!=`, pre and pos-increment `operator++`, and pre and pos-decrement `operator--`. The class `Iterator` should also have a public default constructor and a (non-public) constructor to create an iterator given a pointer to a tree's node. The default constructor initializes the iterator's pointer with null pointer.
- Add two public member functions of class `BinarySearchTree`, called `begin` and `end`. Function `begin` returns an `Iterator` to the node storing the smallest value of the tree, while `end` returns `Iterator()`.
- Modify the public member function `BinarySearchTree::contains` such that it returns an `Iterator` (instead of a `bool`) to the node storing the searched value. If the searched valued is not found in the tree then iterator `end()` should be returned.
- Test that your code passes the tests given in `test3.cpp` (i.e. no assertion fails).

Feel free to add other tests to the test program `test3.cpp`. However, during redovisning you should use the original program without modifications.

Exercise 3: build a frequency table of words

In this exercise, you are requested to write a program that creates a frequency table for the words in a given text file. The words in this table should only contain lower-case letters and digits, but no punctuation signs (e.g. .,!?:\"();). Genitive apostrophe (' as in china's) is possible though. The frequency table should be alphabetically sorted.

The input file contains no special characters (such as å, ä, ö, ü or similar) but punctuation signs and words with upper and lower-case letters may occur in the file (e.g. China's). For every word read from the file, all upper-case letters should be transformed to lower-case letters and all punctuation signs should be removed. As usual, words are separated by white spaces.

This problem was also part of the last lab in the TNG033 course (lab3, exercise 1). 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::Iterator`) to traverse the tree. A BST is the data structure usually used to implement `std::map`.

Each entry of the frequency table contains a key (`std::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 text file `text.txt` and `text_long.txt`. The expected frequency tables are available in the files `frequency_table.txt` and `frequency_table_long.txt`, respectively.

Exercise 4: 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. You'll need to answer this question orally during the RE lab session on week 19 (so, no need to submit a written answer).

Presenting solutions and deadline

You must submit your code for exercise 1, 2, and 3 via Lisam until **May 8th, 08:00 sharp**. Recall that any submitted files via Lisam must include the name and LiU-id of all group members. If you do not submit your code until the deadline given above, then you won't be invited for a Zoom meeting during the scheduled RE lab session on week 19. During the Zoom meeting, you are given the opportunity to present your solution and answer individual questions.

Necessary requirements for approving your lab are given below.

- Use of global variables is not allowed, but global constants are accepted.
- You must be able to motivate the time complexity of the functions.
- 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 neither other memory related bugs. On the labs webpage you can find a list of tools that can help to check for memory related problems in the code.
- The only modifications allowed to the public interface of class `BinarySearchTree` are the ones explicitly described in this lab. However, you are allowed to add extra private members to the class, if needed.

If your solution for lab 3 has not been approved in the RE lab session of week 19 then it is considered a late lab. Late labs can be presented provided there is time in a RE lab. All groups have the possibility to present one late lab on the extra RE lab session on week 22.

Lycka till!

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