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.

Preparation

You must perform the tasks listed below before the start of the lab session *Lab3 HA*.

- Download the files for this exercise from the course website. Similar to previous labs, you can then use CMake to create a project for this lab.
- Compile, link, and execute the program. The `main` is in the file `lab3.cpp` and the first batch of tests to run are in the file `test1.cpp`. The expected output is provided in the file `test0_out.txt`.
- Review lecture 7, where binary search trees were introduced.
- 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).
- 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.
- Study the template classes `Node` and `BinarySearchTree`. These classes are also described in section 4.3 of the course book.
- Do exercise 1.
- Read "Writing a custom iterator in modern C++" before starting exercise 2.

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

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

During this lab, it is required to modify the code for the given class `BinarySearchTree`. You can modify non-public member functions. The signature of the given public member functions cannot be modified.

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. Thus, `count_nodes` should not be used in the implementation of the required functions, besides testing.

A template class to represent a binary search tree

You are requested to add extra functionality to the (template) class `BinarySearchTree` presented in the course book, section 4.3. 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 modify the tree representation such that each node stores a pointer to its parent node. The parent pointers will then be used in exercise 2.
  - 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 C++ standard with respect to iterators' validity.

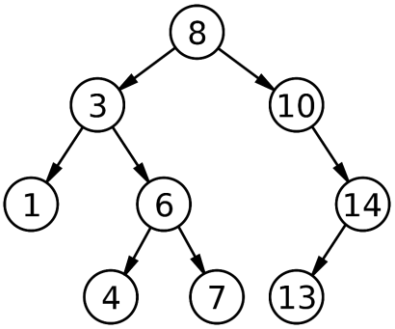
During this lab, it is required to modify the code for the given class `BinarySearchTree`. You can modify non-public member functions. Public member functions signature should not be modified, though the functions body may be modified.

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

Exercise 1

For this exercise, you should add your code to the file `BinarySearchTree.h`.

One of the tasks in this exercise is to add to class `BinarySearchTree` a public member function `find_pred_succ` that, given a value x, returns a pair of 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) `t` below.



Then,

- `t.find_pred_succ(7)` should return `<6, 8>`.
- `t.find_pred_succ(12)` should return `<10 , 13>`.

Note that the function has to handle some special cases: x is less than or equal to the smallest key in the tree; or, x is larger than or equal to the largest key in the tree. The following examples, using the tree `t` above, illustrate what the function should return in these special cases.

- `t.find_pred_succ(1)` should return `<1 , 3>`. Note that $x = 1$ is the smallest key in `t`. In this case, the first value of the returned pair is x, while the second value is the successor of x.
- `t.find_pred_succ(14)` should return `<13 , 14>`. Note that $x = 14$ is the largest key in `t`. In this case, the first value of the returned pair is the predecessor of x, while the second value is x.
- `t.find_pred_succ(50)` should return `<14 , 50>`. Note that $x = 50$ is larger than any value in the given tree. In this case, the first value of the returned pair is the largest value in `t`, while the second value of the returned pair is x.
- `t.find_pred_succ(-5)` should return `<-5 , 1>`. Note that $x = -5$ is smaller than any value in the given tree. In this case, the first value of the returned pair is x, while the second value of the returned pair is the smallest value in `t`.

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

[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.

found and it has a parent node. Otherwise, the default object `Comparable{}` is returned, where `Comparable` is the type of x.

- Uncomment the line "`//#define TEST_PARENT`" at the top of the file `test1.cpp`, so that the compiler can "*see*" and compile the code for the tests in `PHASE 5`, `PHASE 7` and `PHASE 8` of the function `test1` defined in this file.
- Compile, link, and execute the program.
- 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;
  ```

- Uncomment the line "`//#define TEST_PRED_SUCC`" at the top of the file `test2.cpp`, so that the compiler can "*see*" and compile the tests in the body of function `test2` defined in this file.
- Compile, link, and execute the program.

Feel free to add other tests to `test1.cpp` and `test2.cpp`. However, you cannot remove any of the given tests.

Exercise 2

The aim of this exercise is to equip class `BinarySearchTree` with a public class named `Iterator` that represents bidirectional iterators for BSTs. The parent pointer added to each in the previous exercise is useful for implementing the functionality requested in this exercise.

Perform the tasks listed below, by the indicated order.

- Add to class `BinarySearchTree` 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 a null pointer is returned. This function is useful to implement the increment operators (`operator++`) of the `Iterator` class.

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

- Add to class `BinarySearchTree` 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 `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) const;
  ```

- Add the definition of the template class `Iterator` to file `iterator.h`. As discussed in seminar 2, instances of this class should store internally 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`, named `begin` and `end`. Function `begin` returns an `Iterator` to the node storing the smallest value of the tree, while `end` returns `Iterator()`. These functions should be added to file `BinarySearchTree.h`.
- Add a public member function of class `BinarySearchTree`, named `find`, that returns an `Iterator` pointing to the node storing `x`. If key `x` is not found in the tree then iterator `end()` should be returned.

```
Iterator find(const Comparable& x);
```

- Uncomment the line "`//#define TEST_ITERATOR`" at the top of the file `test3.cpp`, so that the compiler can "*see*" and compile the tests in the body of function `test3` defined in this file.
- Compile, link, and execute the program.

Feel free to add other tests to `test3.cpp`. However, you cannot remove any of the given tests.

Exercise 3

For this exercise, you should add your code to the file `frequency_table.cpp` and implement function `exercise3()`.

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 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 new data type `Row` that represents each row of the table. This data type should also overload `operator<` (why?).

To test your code make sure to first uncomment the line "`//#define TEST_EXERCISE3`" at the top of the file `frequency_table.cpp`. Two text files `text.txt` and `text_long.txt` can be used to test the code. The expected frequency tables are available in the files `frequency_table.txt` and `frequency_table_long.txt`, respectively.

To be easier to check that the frequency table your program has built has the correct content, you should do as follows.

- Copy the contents of the table built by your program into a vector.

- Load the contents of the file `frequency_table.txt` (or `frequency_table_long.txt`) into another vector.
- Use an assertion to check that both vectors are equal.

Exercise 4

It is important to be aware that the implementation suggested by this lab of a container based on a BST, with support for bidirectional iterators, has its limitations (this is just an exercise).

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 with a concrete example. You'll need to answer this question orally during the RE lab session.

Presenting lab and deadlines

The exercises in this lab are compulsory and you should demonstrate your solutions during the lab session *Lab3 RE*. Read the instructions given in the labs webpage and consult the course schedule.

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.
- You must be able to answer the question posed in exercise 4.
- Readable and well-indented code. Note that complicated functions and over-repeated code make programs quite unreadable and prone to bugs.
- 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 may add extra private members to the class, if needed.

If your solution for lab 3 has not been approved in the scheduled lab session *Lab3 RE* then it is considered a late lab. Late labs can be presented provided there is time in a another RE lab session. All groups have the possibility to present one late lab on the extra RE lab session scheduled in the end of the course.

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