EX4

	Section	Page
Introduction	1	1
Main	2	2
Store values in the tree	7	4
Traverse the tree	11	5
Index	16	6

1. Introduction. This is a literate program which solves the problem set in lab four—Implementing a binary search tree.

We will start with the, by now familiar, outline of the program. This time we will not use global data and we have no functions other than *main*.

```
⟨ Headers 4 ⟩
using namespace std;
⟨ Global variables and types 6 ⟩
⟨ Prototypes for the functions 9 ⟩
⟨ The main program 2 ⟩
⟨ Implementation of the functions 10 ⟩
```

2. Main. Ok, let's start writing *main*. The skeleton of the *main* program is as follows.

```
⟨ The main program 2 ⟩ ≡
  int main( )
  {
    ⟨ Variables of main 3 ⟩
    ⟨ Open and validate the input file 5 ⟩
    ⟨ Store values into the tree 7 ⟩
    ⟨ Traverse the tree 11 ⟩
    ⟨ Finish and clean up 15 ⟩
  }
```

This code is used in section 1.

3. The first thing we need to do is declare the variables we need to input data. Let's start with the character array *filename* and the input stream *fin*.

```
⟨ Variables of main 3 ⟩ ≡
  char filename[20];
  ifstream fin;
```

See also section 8.

This code is used in section 2.

4. Hang on—we need a couple of header files *iostream* for stream-based input and *fstream* for managing files.

```
⟨ Headers 4 ⟩ ≡
#include <iostream>
#include <fstream>
```

See also section 14.

This code is used in section 1.

5. Right—now we can get the file opened, ready for input. We will prompt for the input file name using *cerr* so that we can redirect the output without getting the prompt in the output file and so that we can see the prompt even when we redirect standard output. We will then read in the file name and open an input stream. We should test for errors too, I guess.

```
⟨ Open and validate the input file 5 ⟩ ≡
  cerr ≪ "Please␣enter␣the␣name␣of␣the␣input␣file:␣";
  cin ≫ filename;
  fin.open(filename);
  if (¬fin) {
    cerr ≪ "Error␣opening␣file␣" ≪ filename ≪ ".␣Program␣will␣exit." ≪ endl;
    return 0;
  }
```

This code is used in section 2.

6. Before we start reading the file we need to declare the data structure we are going to use to store BST nodes.

⟨ Global variables and types 6 ⟩ ≡

```
struct BSTnode {
   int value;
   BSTnode *left
   {nullptr};
   BSTnode *right
   {nullptr};
};
```

This code is used in section 1.

7. Store values in the tree. We are now ready to do the main input loop. we can read the integer values from the input file and store them in the tree. We will use a recursive function, *BSTinsert*, to manage the tree. To kick things off we need a pointer to the root of the BST, which we will call, somewhat unimaginatively, *root*.

```
⟨ Store values into the tree 7 ⟩ ≡
  while (fin ≫ input) {
    root = BSTinsert(input, root);
  }
```

This code is used in section 2.

8. Better declare *input* and *root*.

```
⟨ Variables of main 3 ⟩ +≡
  int input;
  BSTnode *root = nullptr;
```

9. Ok—let's actually implement the functions as we need them this time. First the prototype.

```
⟨ Prototypes for the functions 9 ⟩ ≡
  BSTnode *BSTinsert(int, BSTnode *);
```

See also section 12.

This code is used in section 1.

10. And then the implementation. The function should test whether the *incoming* node is defined and, if not, create a new node to store the *current* data. if *current* is already in use we recursively call *BSTinsert*, using either the *left* or *right* child; depending on whether *incoming* is less than or greater than *current.value*.

```
⟨ Implementation of the functions 10 ⟩ ≡
  BSTnode *BSTinsert(int incoming, BSTnode *current)
  {
    if (current ≡ nullptr) {
      current = new BSTnode;
      current→value = incoming;
    }
    else if (incoming ≤ current→value) current→left = BSTinsert(incoming, current→left);
    else current→right = BSTinsert(incoming, current→right);
    return current;
  }
```

See also section 13.

This code is used in section 1.

11. Traverse the tree. Back to main—we now perform an in-order traversal of our BST tree printing out the node values as we go. We will use a second reursive function *BSTtraverse* to do this. We will use an integer *count* to keep track of how many values we print. This will allow us to get 10 items to a line. We also neeed to purge any pending output if there is not an exact multiple of 10 values in the input file. We will use pass by reference to allow us to change the value of *count*.

```
⟨ Traverse the tree 11 ⟩ ≡
  int count = 0;
  BSTtraverse(root, count);
  if (count % 10 ≠ 0) cout ≪ endl;
```

This code is used in section 2.

12. So now we define our traversal function;

```
⟨ Prototypes for the functions 9 ⟩ +≡
  void BSTtraverse(BSTnode ∗, int &);
```

13. The implementation of the function involves traversing the *left* and *right* subtrees of the *current* node, printing out the *current.value* between the two subtree traversals. If the *current* node is null we simply return.

```
⟨ Implementation of the functions 10 ⟩ +≡
  void BSTtraverse(BSTnode ∗current, int &count)
  {
    if (current ≡ nullptr) return;
    BSTtraverse(current→left, count);
    cout ≪ setw(5) ≪ current→value;
    if (++count % 10 ≡ 0) cout ≪ endl;
    BSTtraverse(current→right, count);
    return;
  }
```

14. If we use setw() we had better include the required header file.

```
⟨ Headers 4 ⟩ +≡
#include <iomanip>
```

15. We should also close our input file.

```
⟨ Finish and clean up 15 ⟩ ≡
  fin.close();
  return 0;
```

This code is used in section 2.

16. Index. This index is automatically created. It lists all the variables used in the program and the section(s) in which they are used. Underlined entries indicate where a variable is defined. The remaining sections of this document are also created automatically.

BSTinsert: 7, 9, 10.
BSTnode: 6, 8, 9, 10, 12, 13.
BSTtraverse: 11, 12, 13.
cerr: 5.
cin: 5.
close: 15.
count: 11, 13.
cout: 11, 13.
current: 10, 13.
endl: 5, 11, 13.
filename: 3, 5.
fin: 3, 5, 7, 15.
fstream: 4.
ifstream: 3.
incoming: 10.
input: 7, 8.
iostream: 4.
left: 6, 10, 13.
main: 1, 2.
nullptr: 6, 8, 10, 13.
open: 5.
right: 6, 10, 13.
root: 7, 8, 11.
setw: 13.
std: 1.
value: 6, 10, 13.

⟨ Finish and clean up 15 ⟩ Used in section 2.
⟨ Global variables and types 6 ⟩ Used in section 1.
⟨ Headers 4, 14 ⟩ Used in section 1.
⟨ Implementation of the functions 10, 13 ⟩ Used in section 1.
⟨ Open and validate the input file 5 ⟩ Used in section 2.
⟨ Prototypes for the functions 9, 12 ⟩ Used in section 1.
⟨ Store values into the tree 7 ⟩ Used in section 2.
⟨ The main program 2 ⟩ Used in section 1.
⟨ Traverse the tree 11 ⟩ Used in section 2.
⟨ Variables of main 3, 8 ⟩ Used in section 2.