**CPSC 261: Lab 3**

**Lab 3 - Reading and Writing Complex Data**

* [Goals](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab3.html#goals)
* [Introduction](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab3.html#introduction)
* [Overview](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab3.html#overview)
* [The Details](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab3.html#details)
* [Provided Materials](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab3.html#provided)
* [Handing It In](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab3.html#handing)

**Goals**

The goals of this lab are to:

* Follow up your experience with reading binary data from files in Lab 2 with writing and reading more complex linked data structures to and from files
* Enhance your C programming expertise
* Strengthen your understanding of pointers and memory usage in C programs.
* Deepen your understanding of UNIX File I/O.

**Introduction**

In Lab 2 you dealt with primitive data and were only responsible for reading from files. This lab follows up Lab 2 by doing the other half of the problem (writing data as well as reading data) and by increasing the complexity of the data structures that are being stored in and retrieved from files.

In this lab, you are going to write a program that reads and writes binary trees to the file system.

This lab may seem kind of long and complicated. However, you will have two weeks to complete it. See the end of this description to see what you need to complete during the first week.

**Binary Trees**

In Computer Science, the term *Tree* refers to a data structure that stores a number of values in a hierarchial manner. A Tree is a directed acyclic graph of nodes where each tree node has a value and some number of edges to other nodes. These edges are often called branches and the nodes that are pointed to by a node are often called the *children* of the node.

A *Binary Tree* is a special case of a Tree where each nodes has no more than two branches and so no more than two children. In a Binary Tree, the children of a node are called *left* and *right*. In this lab, the Tree structure is declared as follows:

struct Tree

{

valuetype value; // the value of the node

struct Tree \*left;

struct Tree \*right;

};

In general value (i.e. that value of that node in the tree) can be any data type (e.g. int, char, long[], a struct, etc.). In this lab valuetype is typedef'd to be a long. left and right are branches, or pointers to child Trees. Either or both of left and right may be NULL.

**Reading and Writing a Binary Tree from/to a file**

Storing a Tree into a file requires a function to write the Tree to the file system and a function to read the Tree back into the program. Indeed, similar read and write functions can be used for other data structures too, like Lists, Queues, Stacks, as well as complex databases.

One problem that needs to be overcome is how to store the representation of a tree node in a file. It is possible, for example, to simply write struct Tree to the file. That would cause the tree value to be written to the file along with the pointers to the left and right children. The problem is that when reading the tree back in, we don't know which tree node written into the file is associated with a particular "pointer" and certainly any dynamically allocated struct Trees won't be located at the same place in memory that the corresponding node in the source tree resided. To overcome this problem, some other strategy is needed.

One approach is to store each binary Tree node as a unique ID (TREE-ID) for that node followed by the value and then the unique TREE-IDs for the left and right children. An abstract representation of a complete binary tree consisting of 7 tree nodes would look like the following in a file:

ID01 4 ID02 ID03  
ID02 2 ID04 ID05  
ID03 6 ID06 ID07  
ID04 1 NULL NULL  
ID05 3 NULL NULL  
ID06 5 NULL NULL  
ID07 7 NULL NULL

In the above the first column is the TREE-ID of that node, it is followed by the value of that node (recall that our tree has longs as the type of value), and the TREE-IDs of the left and right children. If the child node doesn't exist then NULL (i.e., zero) is written as the TREE-ID. Note that the file will not contain text data, only the actual binary representation of the IDs and values.

An alternative way of storing a binary tree in a file avoids storing the unique ids of each node. If you look at the representation of the tree above, you will see that the ids that are assigned to the nodes are sequential and that the nodes are stored in the file in sequential order. Given that, there is no fundamental need to store the id of each node. Instead we can use the position of the node in the file as its id. This will reduce the amount of storage required to store a tree. However this is a bit more tricky to implement as you have to be more careful when writing the tree. We will not be expecting you to implement this optimization in this lab.

**Overview**

You will be reusing most of the functions that you wrote for lab2:

* swapInt, swapLong
* isLittleEndian
* readChar, readString, readInt, readLong, readPtr

This lab has a number of parts:

* First you must write a collection of functions that write primitive atomic data values to a file. These match the functions that you wrote in lab2 that read primitive atomic data from a file.
  + writeChar
  + writeString
  + writeInt
  + writeLong
  + writePtr
* Next, you must write a function called readTreeNode that uses your functions that read primitive data values from a file to read a binary tree node from a file, along with a function called readTree that reads an entire binary tree from a file, translating the ids read from the file back into pointers.
* Next, you must write a function called writeTreeNode that uses your functions that write primitive data values from a file to write a binary tree node to a file, along with a function called writeTree that writes an entire binary tree to a file. This function will have to deal with assigning ids to each of the nodes in the tree and writing ids rather than pointers.
* Finally, you will need to write driver programs called testreadtree and testwritetree that use all of the functions described above and demonstrate that your readTree and writeTree functions work correctly.

**The Details**

* [Step 1](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab3.html#step1) Implement the write data functions
* [Step 2](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab3.html#step2) Implement readTreeNode and readTree
* [Step 3](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab3.html#step3) Implement the testreadtree program
* [Step 4](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab3.html#step4) Implement writeTreeNode and writeTree
* [Step 5](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab3.html#step5) Implement the testwritetree program
* [Step 6](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab3.html#step6) Implement a script to test your programs
* Step 1 - Implement the functions that write primitive data to a file.

The prototypes for these functions are:

* + void writeChar(FILE \*f, char c);
  + void writeString(FILE \*f, char \*c);
  + void writeInt(FILE \*f, int c);
  + void writeLong(FILE \*f, long c);
  + void writePtr(FILE \*f, void \*c);

Note that these functions do not take a swap argument. Data is always written in the *natural* endian-ness of the computer that is doing the writing.

All 5 of these write functions will use the standard C output function fwrite to write the necessary data to the file. The documentation of the fwrite function can be found [here](http://linux.die.net/man/3/fwrite).

A single file named writeData.c should contain the implementation of all 5 of these functions.

**Notes**

* + A string consists of all of the characters pointed to by the pointer up to and including the null character '\0' that is used to terminate strings in C. This is the only function that will write a variable number of bytes to the file.

We have provided a test program named encode.c that uses these 5 write functions to write some data to a file. You should use the encode program to test your write functions; the decode program from Lab 2 should be able to read the files that are written by the encode program using your functions.

make encode

./encode newdata.dat

../lab2/decode newdata.dat

* Step 2 - Implement readTreeNode() and readTree()

The prototype of readTreeNode() is:

* + Tree \*readTreeNode(FILE \*file, int swap);

This function allocates space for a new tree node, and uses your functions from Lab 2 that read primitive data to read the three fields of a tree node from the file. They will be written in the order: value, left, right.

The function returns the newly created and initialized node. Note that it does not worry about converting between ids and pointers; that will be done by readTree, the higher level function that calls readTreeNode.

The prototype for readTree is:

* + Tree \*readTree(FILE \*file, int swap);

This function reads an id from the file, then uses readTreeNode to read from the file the tree node that has that id. It repeats this process until all of the tree nodes have been read from the file. The first tree node read is the root of the tree.

The tricky part is that the left and right pointers in each tree node when it is read from the file do not contain pointers, they contain the id of the corresponding node. It is also the case that the order in which tree nodes are written to the file is what is called *pre-order*. That is, each node is written to the file before its left and right subtrees are written to the file. So, when reading the collection of tree nodes from the file, we will see the ids of the left and right subtrees before we see the nodes that have those ids.

After readTree has read all the tree nodes from the file, it has to reconstruct the tree by translating all of the ids that were read into pointers to the correct tree nodes. This process is traditionally called *unswizzling*. *Swizzling* is the process on the writing side that translates the pointers in a linked data structure into ids so that the structure can be stored in a file or transmitted on a network.

After readTree has unswizzled all of the pointers, it returns the tree that it has read from the file.

It is important that readTree does not make any assumptions about the ids that will be used to identify the tree nodes. In particular, it cannot assume that the ids will be small integers in increasing order. We'll see why when we get to the writeTree function.

Your readTree function is going to have to store in some data structure the association between ids and tree nodes. For simplicity, it is ok to use a fixed size array of structures to store this mapping. You may assume that no trees will contain more than 1,000,000 nodes so you can use that as the size of the array.

Here is some code that you might find useful for this task. This code appears near the beginning of my readTree.c file.

#define TABLESIZE 1000000

struct {

long id;

Tree \*tree;

} lookupTable[TABLESIZE];

int nEntries = 0;

void rememberId(long id, Tree \*tree) {

lookupTable[nEntries].id = id;

lookupTable[nEntries].tree = tree;

nEntries ++;

}

Tree \*findNodeWithId(long id) {

int i;

for (i = 0; i < nEntries; ++i) {

if (lookupTable[i].id == id)

return lookupTable[i].tree;

}

return NULL;

}

* Step 3 - Implement a driver program for your readTreeNode and readTree functions

In a file named testreadtree.c, implement a program that uses your readTree function to ensure that it works. This program should take as its single argument the name of a file in which a tree has been previously written.

We are using the same technique as we did in lab2 to detect the endian-ness of a file that contains a tree. The first 4 bytes of the file will be the value 0x01020304 written in the natural endian-ness for the host that created the file.

We have provided you with 3 test files that contain trees.

littleendian9.tree

a small 9 node tree written on a little-endian computer

bigendian9.tree

a small 9 node tree written on a big-endian computer

bigendian31.tree

a complete binary tree with 31 nodes written on a big-endian computer

We have also provided a printTree.c file that contains two functions that print binary trees.

printTree(Tree \*tree)

This function prints a tree to standard output, providing as much information as possible about the tree. It prints the values of the pointers as well as the value stored in the node. It indents the display so that the relationship between parent and child nodes can be seen.

showTree(Tree \*tree)

This function prints a tree to standard output, printing only the values, but indenting so that the relationship between parent and child nodes can be seen.

Two trees that are isomorphic (having identical form, shape, or structure) will display identically using showTree.

Your testreadtree program should read the tree from the provided file and print it using showTree to standard output. While you are debugging your program it may be helpful to use printTree rather than showTree, as printTree provides more details.

* Step 4 - Implement writeTreeNode() and writeTree

The prototype for the writeTreeNode function is:

* + void writeTreeNode(FILE \*file, Tree \*tree);

This function writes the three fields in a tree node to the file in the order: value, left, right. This function assumes that the pointers left and right have already been *swizzled* into ids appropriate for writing.

All three fields are written in the natural endian-ness of the computer doing the writing.

The prototype for writeTree is:

* + void writeTree(FILE \*file, Tree \*tree);

This function arranges to write the id of the node followed by the node itself for each tree node in the tree, starting with the node at the root of the tree. It also must take care of assigning ids to nodes.

In this implementation, we are going to take full advantage of the flexibility of the readTree function with respect to ids. We will assign as the id of each tree node, the address in memory of the node. As the memory address of each tree node is unique, the memory address can serve as an id for the node. Child pointers that are NULL will have NULL written as their ids.

writeTree must ensure that it writes all of the tree nodes that are in the tree to the file. The root of the tree must be the first tree node writen to the file.

* Step 5 - Implement a driver program for your writeTreeNode and writeTree functions

In a file named testwritetree.c, implement a program that uses your writeTree function to ensure that it works. This program should take as first argument the name of a file to which the tree is to be written.

Your testwritetree program must mark each file that it creates with its endian-ness, so it must write the (32 bit) int 0x01020304 to the file before writing any tree data. This allows the readers of the file to determine the endian-ness of the data in the file

We have provided you with a function that will create trees of various kinds. This function is called makeTree and it is called in a manner similar to how main programs are called in C. It takes an integer indicating how many arguments are being provided to it and a pointer to an array of strings that holds the arguments. It is defined in this manner so that it is convenient for a main program to pass a sequence of its arguments to the makeTree function.

For example, the testwritetree program can call makeTree as follows:

int main(int argc, char \*argv[]) {

...

// open the file whose name is given as argv[1]

...

Tree \*t = makeTree(argc - 2, &argv[2]);

}

The first element of argv for a C program is the name of the program executable itself, which is usually boring, the second argument (argv[1]) for testwritetree will be the name of the file that is to contain the tree, and the rest of the arguments can be passed to makeTree to control the kind of tree that it creates.

The first argument that makeTree sees should be one of:

little

Make and return a fixed small tree with 9 nodes. The nodes of the tree contain the numbers from 100 to 108 in increasing order (if you traverse the tree in pre-order).

random

Make and return a random tree with the number of nodes as given by the next argument. If no size is specified, then 20 is assumed.

complete

Make and return a complete tree with the number of levels in the tree given by the next argument. If no size is specified, 6 is assumed. A complete tree has all of its leaves at the same level, and all of its interior nodes have 2 non-null children.

Your testwritetree program should construct a tree as described above, print the tree out using showTree, and write the tree to the provided file using writeTree.

* Step 6 - Implement a script that tests your programs

Write a (tcsh) shell script that generates random and complete trees of a variety of sizes, writes them to files, reads them back, and ensures that the tree read back is the same as the tree written.

You should ensure that you generate at least 3 different sizes of random tree and 3 different sizes of complete tree.

You can find the complete documentation for the tcsh shell [here](http://linux.about.com/library/cmd/blcmdl1_tcsh.htm). You will want to look at the foreach builtin command as well as output redirection and the diff external command.

To get you started consider the following little script that prints out a list of numbers:

#!/bin/tcsh

foreach i ( 23 54 99 )

echo $i

end

You should write your script in a file named testtree.csh. Once you have created this file, you can execute it by typing:

csh testtree.csh

Or you can make the script executable by typing:

chmod +x testtree.csh

after which you can execute the script as a command:

./testtree.csh

**Provided Materials**

You should use subversion to check out the lab3 directory from your repository, in the same way that you did for lab1 and lab2.

Your lab3 directory should contain the following files:

* Makefile

The provided Makefile contains instructions for compiling and linking your programs. You should not need to, and should not, change the Makefile that we have provided for you.

* encode.c

This program uses your write data functions to write a series of data values to the file whose name is provided as a command line argument.

* tree.h

This file defines the type of the structure Tree and prototypes of all of the functions that you are to implement, as well as the functions that we have provided for you. All of your source program files should:

#include "tree.h"

* printTree.c

C source file that contains the printTree() and showTree() functions.

* makeTree.c

C source file containing the makeTree() function (and a few helper functions that it calls).

* encode.c

C source file containing a program that writes data to a file.

* littleendian9.tree

a small 9 node tree written on a little-endian computer

* bigendian9.tree

a small 9 node tree written on a big-endian computer

* bigendian31.tree

a complete binary tree with 31 nodes written on a big-endian

You will need to create all of the other files.

**When is what due?**

You have two weeks to work on this lab. On the Monday between the first and second week, you must hand in at least:

* [Step 1](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab3.html#step1) The write data functions
* [Step 2](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab3.html#step2) readTreeNode and readTree
* [Step 3](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab3.html#step3) the testreadtree program

On the Monday after the two weeks, you must have completed and handed in the entire lab.