COE428 Lab 5: XML-based Heap

Use of Stacks

1. IMPORTANT: Two week lab

Note that you have two weeks to complete this lab. This lab must be submitted at least 48 hours before the beginning of your next lab period.

2. Prelab preparation

Before coming to the lab you should:

- Read the lab. Try to prepare any questions you may have about the lab.
- Refer to **Lab Guide**.
- Part 1: Create your lab directory for lab5. (i.e. use **mkdir lab5** within your coe428 directory.)
- Change to your coe428/lab5 and unzip the lab5.zip file with the command:
  unzip /home/courses/coe428/lab5/lab5.zip

- Ensure that you have downloaded properly by entering the command: `make`

  No errors should be reported.

Requirements

The requirements to complete the lab are summarized below.

1. Complete Requirements 1 and 2.
2. Answer the Question (see below) in the README file.

Theory

The eXtended Markup Language (XML) is a widely used format for describing structured data.

For this lab, we only examine a simplified form of XML: An XML document is a collection of matching start-tags and end-tags. A start-tag is a string of alphabetic characters preceded by a < and terminated with a >. For example <foo> is a start-tag of type "foo".

An end-tag is an alphabetic string preceded by </ and terminated by a >. For example </foo> is an end-tag of type "foo".

Well-formed XML requires that start-tags and end-tags match (i.e. have the same type.)

Examples		
XML	**Valid?**	**Explanation**
<a>	Yes	"a" tags balance
<a>	Yes	"a" outer tags and "b" inner tags balance
<a>	No	"a" end-tags does not match start-tag ("b")
<a><a>	Yes	all tags balance
<able><baker></Baker></able>	No	"Baker" end-tag does not match start-tag ("baker") (i.e. the tag names are *case-sensitive*.)

Requirement 1:

The algorithm to determine if the start and end-tags balance uses a Stack data structure to keep track of previously read start-tags. The algorithm is:

1. Read the input until the beginning of a tag is detected. (i.e. tags begin with <: if the next character is a / (slash), then it is an end-tag; otherwise it is a start-tag).
2. Read the tag's identity. (e.g. both tags <x> and </x> have the same identity: 'x').
3. If the tag was a start-tag, push it onto the Stack.
4. Otherwise, it is an end-tag. In this case, pop the Stack (which contains a previously pushed start-tag identity) and verify that the popped identity is the same as the the end-tag just processed. **Note:** if the stack cannot be popped (because it is empty), the input is invalid; the algorithm should STOP.
5. If the identities do **not** match, the XML expression is invalid. STOP.
6. If they do match, then no error has been detected (yet!).
7. If there is still input that has not yet been processed, go back to the first step.

8. Otherwise (no more input) then the input is valid **unless** the Stack is not empty. Indicate whether the input is valid (Stack empty) or invalid and STOP.

Note: The XML tag names can be of arbitrary length and composed of upper- and lower-case alphabetic characters. You should use the following files:

Implementation Notes

To implement the algorithm in C, you need a main() function that reads *stdin* and processes each tag names according to the algorithm. You also need to implement a string Stack (including push(), pop() and isEmpty()).

You must use separate C source code files for each of these. The files are:

part1Main.c

The main() function in this file reads *stdin* and implements the algorithm. You are provided with a skeleton implementation of main().

stringStack.c

Again, you are provided with a skeletal version of this file. You need to implement the 3 functions (push, pop and isEmpty). The comments describing what these functions do must not be modified.

Part 2: XML tree representation

The XML language is a textual encoding of a tree data structure. The first start-tag matches the last end-tag and represents the root of the tree. Everything between these tags represent the root's children (which may be empty or be trees).

The table below gives some examples.

Description	XML
Tree with root node only	<node> </node>
Tree with root node and 1 child	<node> <node></node> </node>
Tree with root node and 3 children	<node> <node></node> <node></node> <node></node> </node>
Root with 2 children, 2nd child has 1 child	<node> <node></node> <node> <node></node> </node> </node>

Node identification

The XML tree description above does not include information about the nodes. This can be fixed by allowing start-tags to have additional information associated with them.

For example, a tree with a single root node with the information "foo" associated with it can be expressed in XML as <node id="foo"></node>.

Similarly, a tree with a root (value 5) and two children (values 2 and 7) can be expressed in XML as:

```
<node id="5"><node id="2"></node><node id="7"></node></node>
```

(Note: this is also a valid Binary Search Tree.)

Requirement 2:

Your program must:

1. Read integers from *stdin* and add each one to a Heap.
2. Print the Heap tree structure as XML.
3. Print the integers in both sorted and reverse-sorted order.

Example

Suppose that the input to the program is:

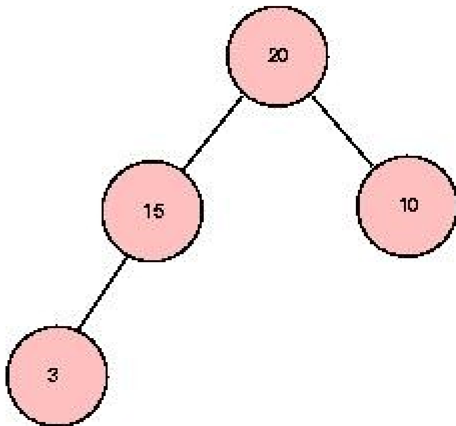
```
15

20

10

3
```

The tree heap data structure will look like:



The program then prints the tree structure of the heap as an XML expression. The output is:

```
<node id="20"><node id="15"><node id="3"></node></node><node
id="10"></node></node>
```

Next, the program deletes items one-by-one from the heap. As each item is deleted, its value is printed and also pushed onto a stack. The output is the sorted numbers (descending):

```
20

15

10

3
```

Finally, the program pops the stack and prints each item producing the output:

```
3

10

15

20
```

Implementation Notes

To implement the algorithm in C, you need a main() function that reads *stdin* and processes each line. You also need to implement a Heap and a Stack.

You must use separate C source code files for each of these. The files are:

part2Main.c

The main() function in this file reads *stdin* one line at a time. You are provided with a skeleton implementation of main() that handles the "read integers from *stdin* loop".

However, you have to code the actual algorithm. You will need the Heap and the Stack for this.

intStack.c

Again, you are provided with a skeletal version of this file. You need to implement the 3 functions (push, pop and isEmpty). The comments describing what these functions do must not be modified.

intHeap.c

Again, you are provided with a skeletal version of this file. You need to implement the 3 functions (heapAdd, heapDelete and heapSize). The comments describing what these functions do must not be modified.

Question

In your README file, you need to answer the following question:

Another legal XML tag not used in this lab is the "stand-alone" tag. This kind of tag combines both a start-tag and end-tag in one. It is identified with a '/' (slash) preceding the final >. (For example, the <foo/> is a stand-alone tag that is "self balancing".

Describe briefly how you would modify Requirement 1 to allow this kind of tag.

Submit your lab

1. Go to your **coe428** directory
2. Zip your **lab5** directory by using the following command:
   zip -r lab5.zip lab5/
3. Submit the lab6.zip file using the following command:

   submit coe428 lab5 lab5.zip