University of Illinois at Urbana-Champaign
Department of Computer Science

MP8: Exceptions and Disunion

CS 225 Data Structures
Spring Semester, 1999

Handed out: Thursday, April 8, 1999
Due date: Wednesday, April 14, 1999 at 11:59 PM

1 Introduction

In this MP, you will get to apply the ideas about exceptions that you learned in section, and also to add an "Undo" operation to the uptree implementation of disjoint sets from the course library.

2 Exceptions

Exceptions are a language feature designed primarily for error handling. So far in this course, we have generally dealt with errors by printing out a message and exiting the program. While this may be the appropriate response for some applications, it could be disastrous for others. Imagine what would happen if a life-support machine suddenly printed out the message

```
Trying to read top element from empty stack!
```

and stopped functioning. The problem of error handling is especially troublesome for library code since it may be used in many different types of applications. Library writers can detect errors, but need to let the rest of the program decide on the appropriate response to them, be it printing out an error message, terminating the program, or paging technical support. Exceptions are designed to allow this type of flexible error handling.

The basic terminology for exceptions is that the code which detects the error *throws* an exception and then the error handling code catches it and deals with the error. In order to throw an exception simply use the keyword `throw`:

```
throw exception;
```

where *exception* is a variable to be "thrown". Typically it is a member of a special class or struct intended only for exceptions. For example, if we had declared the `CaffeineError` class to denote dangerously low caffeine levels, then when this condition is detected we throw the exception with the following command:

```
throw CaffeineError();
```

and the error handler tries to fix this situation, presumably by seeking appropriate beverages. (Notice the parentheses after `CaffeineError`. This line is actually calling the default constructor for type `CaffeineError` since `throw` takes a variable, not a type. If we had previously declared a variable of type `CaffeineError`, we could use this as the argument for `throw`, but most often a constructor is used as above since exception classes are typically declared only for error handling.)

Not surprisingly, exceptions are caught with the `catch` keyword. However, before you can use `catch`, you need to put the code which may throw the exception into a block preceded by the keyword `try` and specify the type of error you wish to catch. The format for this is as follows:

```
...
try
  {
  //potentially dangerous code here
  }
catch( error-type  parameter-name)
  {
  //error handling code here
  }
...
```

For the example above, *error-type* would be `CaffeineError`. The *parameter-name* is optional, but if it is included then the exception will assigned to the name of the variable placed there. This can be used to pass information about the error to the error-handling code. See the sample programs in ~`cs225/src/mp8/examples`, in particular `parameter.C`, for more information about this and other features of exceptions.

3 Your assignment

To begin with, you will need to copy the given files into your own directory. The files are located in:

```
~cs225/src/mp8           (MP code)
~cs225/src/mp8/test      (test files)
~cs225/src/mp8/examples  (sample code using exceptions)
```

Your task is to make the following changes to the given files:

1. Add the class `StackError` to the top of `stack.h` (outside the declaration for `Stack`). This class should include one public member variable, a `const char*` (array/string of constant characters) called `message`. This variable will be set to an error message describing the error which has occurred. Also create a constructor which takes a `const char*` and initializes this variable.

   Next remove the calls to `Assert` from `stack.C`. Replace these with code which checks if the corresponding error has occurred (either removing or viewing the top of an empty stack) and throws a `StackError` exception if appropriate. Use the error message which `Assert` would print as the error description in the exception you throw.

2. The second part of the MP involves using your new `Stack` class to implement a function `Undo` in the `DisjointSets` class defined in the `nodedisjoint.*` files. This function takes no arguments. It reverses the effect of the last effective call to `Union`. Since `Union` combined two previously-separate sets, `Undo` will separate the combined set back into the sets from which it was formed. If `Undo` is successful it returns a 1, but if no calls to `Union` remain to be reversed it returns a 0.

In order to support this operation, you will need to store a record of the changes made to the nodes in the `DisjointSets` class. Note that this implementation uses path compression so `Undo` actually has to reverse the effects of all `Find` operations since the most recent `Union` as well as the `Union` operation itself. It is recommended that these changes be stored in a stack or a stack of lists. The minimum knowledge required to reverse a pointer adjustment is which node's parent pointer was changed and the previous value of that parent pointer. Feel free to store additional information if necessary. Obviously you will need to distinguish the changes which occur during a `Union` from changes made during path compression. In addition, `Undo` must reverse the changes made to the uptree's size (stored in the variable `count`) so that subsequent unions occur properly.

Since this MP is partially about exception handling, use the newly-added `StackError` exception to tell when the `Stack` is empty. Specifically, don't use the `Is_Empty` function in `Undo`, but rather catch the exception from the `Stack`. (Feel free to use `Is_Empty` in other functions, however, as long as those aren't part of `Undo`.) Note that using exceptions this way is somewhat wasteful since exception handling is relatively slow, but it's justified here so that you get practice with exception handling.

Lastly, make sure that the other member functions of `DisjointSets` handle the additional "undo information" properly. In particular, it may need to be managed in the constructors and the destructor. You do not have to copy it in the copy constructor or `operator=`, however.

Don't forget your PSP work!

4 Handing in your code

To handin your MP8 code, use the command

```
handin cs225 mp8 stack.h stack.C nodedisjoint.h nodedisjoint.C
```