CS3310 Data and File Structures with Java
Instructor: Ajay K Gupta
Lab TA: Yu Guo, Zijiang J Yang
Assignment: 2

Jonah Kubath

Software Life Cycle Report – Assignment 2

Phase 1: Specification

- Work with linked lists
- Create stacks
- Create Queues
- Use the stacks / queues to solve problems

Program Output is at the end of this file.

Phase 2 Design:

Modules and Basic structure

Module 1: Class Main
Fields:

1. StackQueueDemo demo – object to run through the tests of balanced parentheses.

Methods – none

Module 2: CharNode
Fields:

1. Char myData – holds the data for the node
2. CharNode nextNode – points to the next node in the list
3. CharNode previousNode – points to the previous node in the list

Methods:

1. CharNode(char setData) – Overloaded constructor to set node data as setData value
2. Appropriate Getters and Setters for the 3 fields

Module 3: CharList
Fields:

1. CharNode head – points to the head of the list
2. CharNode tail – points to the tail of the list

Methods:

1. CharList() – default constructor that builds a head node and sets the tail to point to the head.
2. Void insertTail() – creates and insert a node at the tail of the list
3. Void insertTail(char newChar) – creates a new node with newChar set as data and inserts it at the end of the list
4. CharNode deleteTail() – removes the last node in the list and returns the object.
5. CharNode deleteHead() – removes the head node in the list and returns the object.

Module 4: CharQueue

Fields:

1. CharList list – holds a list object

Methods:

1. Void enqueue() – inserts a node at the tail of the list
2. Void enqueue(char newChar) – inserts a node at the tail of the list with data set to newChar.
3. CharNode dequeuer() – deletes the head node and returns the object

Module 5: CharStack

Fields:

1. CharList list – holds a list object

Methods:

2. Void push() – inserts a node at the tail
3. Void push(char newChar) – inserts a node at the tail of the list with data set to newChar.
4. CharNode pop() – deletes the tail node and returns the object

Module 6: QueueCheckBalancedParentheses

Fields:

1. String text – holds the text line to check
2. CharQueue que – holds a queue object

Methods:

1. QueueCheckBalancedParentheses – default constructor that does nothing
2. QueueCheckBalancedParentheses(String userText) – sets the text field to userText.

3. Int CheckBalancedParentheses() – enqueues when "(" is found and dequeues when ")" is found. Returns 0 if the parentheses were balanced.
4. Void setText(String newText) – removes any characters that are not "(" or ")" and sets the fields text to the new String.

Module 7: StackCheckBalancedParentheses
Fields:
1. String text – holds the text line to check
2. CharStack stack – holds a stack object

Methods:
1. StackCheckBalancedParentheses – default constructor that does nothing
2. StackCheckBalancedParentheses(String userText) – sets the text field to userText.
3. Int CheckBalancedParentheses() – pushes when "(" is found and pops when ")" is found. Returns 0 if the parentheses were balanced.
4. Void setText(String newText) – removes any characters that are not "(" or ")" and sets the fields text to the new String.

Module 8: StackQueueDemo
Fields:
1. String userInput – holds the input string from the user / file
2. QueueCheckBalancedParentheses que – holds a queue object
3. StackCheckBalancedParentheses stack – holds a stack object
4. Scanner scan – holds a scanner object for user input

Methods:
1. Void checkDemo() – Reads while there is still input. Calls QueueCheckBalancedParentheses and StackCheckBalancedParentheses to check the string and outputs the results.
2. Void getUserInput() – scans for a new line of input
3. Void openFile() – Sets the scanner to read from the file. If an error is found, the user will be asked for a new file to read from.

Pseudocode

```
Main()
        Create a StackQueueDemo object
        Call the checkDemo() of the object
```

StackQueueDemo()

1. checkDemo()
   calls the openFile()
   while(scan.hasNext)
   call getUserInput to save a line
   create a stack and queue object with the String as data
   call the CheckBalancedParentheses() from both of the objects
   save the returned integer to output the results
   if(count != 0) output the string is not balanced
   else output the string is balanced
   Output the average time to run the program

2. getUserInput()
   save the userInput field to sacn.nextLine of the file

3. openFile()
   set a default file to “balancedParenCheckInputs.txt”
   create a boolean to hold whether or not the file was found
   while(!goodFile)
   try
   open the file
   set the scanner to the file object
   set the goodFile = true
   catch (FileNotFoundException e)
   print out the file was not found
   print out the user should enter a new file to read from
   set the default file to the new input file
   //End of while loop
   close the System.in scanner

QueueCheckBalancedParentheses()

1. QueueCheckBalancedParentheses()
   Take in a string and set the field text to the string
2. CheckBalancedParentheses()
   Create count = 0
   For the length of the saved string
   Enqueue when “(“ is found and count++
   Dequeue when “)” is found and count—
   If count is ever negative
   Return -1
   Return an integer
3. setText()
   take in a string and set the field text to the string

StackCheckBalancedParentheses

1. StackCheckBalancedParentheses()
   Take in a string and set the field text to the string
2. CheckBalancedParentheses()
   Create count = 0
   For the length of the saved string
   Enqueue when "(" is found and count++
   Dequeue when ")" is found and count—
   If count is ever negative
   Return -1
   Return an integer
3. setText()
   take in a string and set the field text to the string

CharQueue()

1. enqueu()
   call insertTail() from the CharList object
2. dequeue()
   call deleteHead() from the CharList object
   return the node

CharStack()

1. push()
   call insertTail() from the list object
2. pop()
   call the deleteTail() from CharList object
   return the node

CharList()

1. CharList()
   Create a head node
   Set the tail to point to the head node
2. insertTail()
   Create a new node
   Set the new node previous field to point to the current tail
   Set the tail.next field to point to the new node
   Set the tail to point to the new node
3. deleteTail()
   if(tail == head)
   do nothing as the list is empty
   else
   Create a node to hold the tail node

Set the tail to point to the tail.previous node
Return the temp node

4. deleteHead()
   if(tail == head)
   do nothing as the list is empty
   else
   Create a node to hold the tail node
   Set the head node to the head.next node
   Return the temp node

CharNode()

1. Getters
   return the data for the given field
2. Setters
   Take in a value and set the field to that value

Summary Analysis

Stack

The stack is fairly easy to implement for a problem like this. When a left parenthesis is given, it is pushed on to the stack. When a right parenthesis is found, a left parenthesis is popped from the stack to match with it. After all this is done, if our integer to hold a count of how many times a value was pushed (value++) and how many times a value was popped (value--) returns anything other than 0, we know that there were un-balanced parentheses. 0 is returned when for every popped right parenthesis there was a previously pushed left parenthesis.

Queue

The queue is also a good implementation for this problem as values are handled linearly. The data is handled very closely to the stack implementation. The only difference is that a queue will insert at one end and delete from the other. In this problem, it does not make a difference as the left parenthesis are not unique. The method used to enqueue the left parenthesis and dequeue the right parenthesis does not make a difference in the outcome.

How can I apply this to production software?

I can apply this to production software based on how I want to handle my data. If I want to take in the information and then handle it as a first come first serve, then a queue would be appropriate. Some applications may require me to handle the most recent information first and in this case a stack would be appropriate.

Stack vs Queue

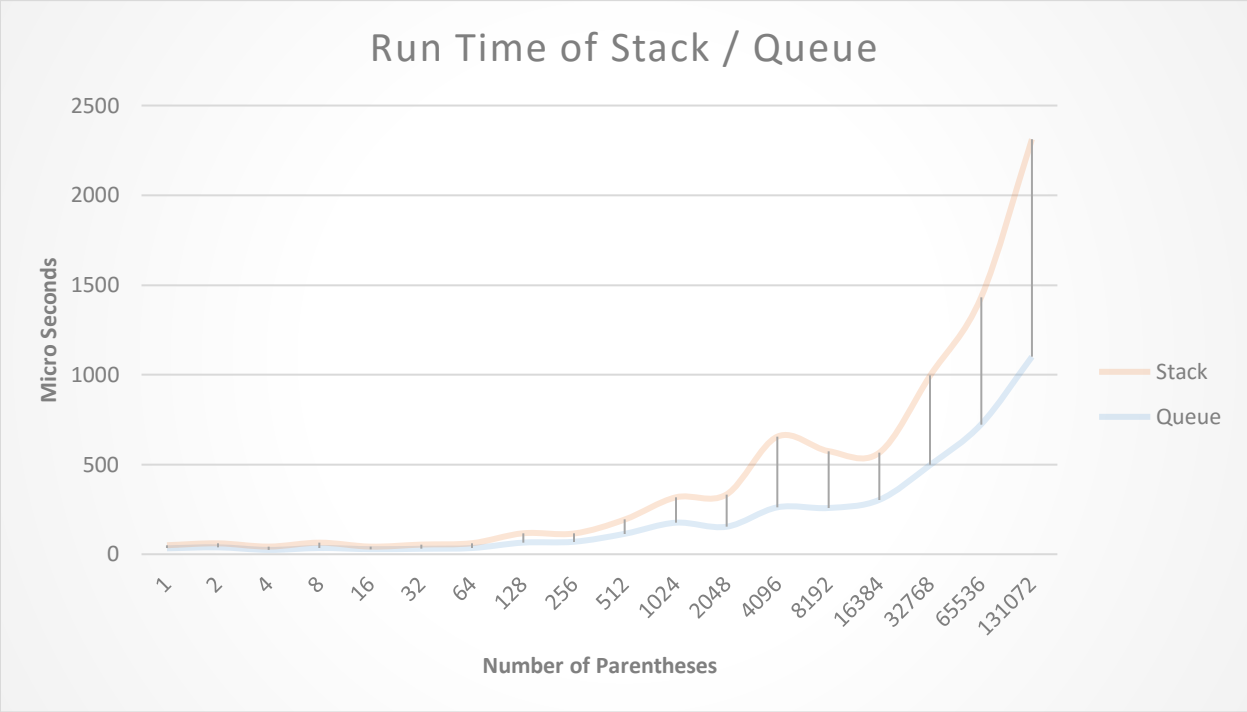
The empirical data for the stack and queue show that they grow at the same rate. There is a little difference in the exact times, but this is such as small amount it can happen because of other processing running on my machine. The general trend of these data structures are the same.

Time Complexity

Main() – O(N) The number of lines / strings to check
StackQueueDemo() –
CharNode() – O(1) only interaction is getters and setters
CharList() – O(1) only deal with 1 node at a time
CharQueue() – O(1) either enqueue or dequeue
CharStack() – O(1) either push or pop
QueueCheckBalancedParentheses() – O(p) one for-loop to go through every left or right parenthesis
StackCheckBalancedParentheses() – O(p) one for-loop to go through every left or right parenthesis

Time of Stack vs Queue

Our total time will be the total number of parentheses that need to be check. Since this is a constant time to handle each node, our time complexity for the application would be constant or O(1).



Phase 3: Risk Analysis

The only risk for this application is having a file that is not found. If this happens, the user is prompted for a new file.

Phase 4: Verification

The application has been run through with many variations to test.

Phase 5: Coding

The code of this program is included in the zip file. The code is explained with comments and line breaks to make reading easier. A Javadoc is also made to explain the use of each method.

Phase 6: Testing

Test runs have been included at the end of this file.

Phase 7: Refining the program

When the stacks and queues are testing for balanced parentheses, if they find more right parentheses that left, the program will terminate early because the string will have unbalanced parentheses. This eliminates the rest of the work on the string.

Phase 8: Production

A zip file including the source files from eclipse and the output of the program have been included.

Phase 9: Maintenance

Changes can be made once feedback is received.

Program Output

“sdb(erlj)((djre)lejr)”
Queue is balanced
Time for Queue - 67 micro s
Stack is balanced
Time for Stack - 48 micro s

“4+(5*4))”
Queue Unbalanced
Time for Queue - 42 micro s
Stack Unbalanced
Time for Stack - 43 micro s

3
Queue is balanced
Time for Queue - 27 micro s
Stack is balanced
Time for Stack - 22 micro s

“ere)(jre”
Queue Unbalanced
Time for Queue - 41 micro s
Stack Unbalanced
Time for Stack - 146 micro s

“((()))((”
Queue Unbalanced
Time for Queue - 155 micro s
Stack Unbalanced
Time for Stack - 81 micro s

)(
Queue Unbalanced
Time for Queue - 35 micro s
Stack Unbalanced
Time for Stack - 20 micro s

Average Queue - 56.285714 micro s
Average Stack - 55.571429 micro s