#	Pri.	Story	Example	Test
1		As a programmer, I want to print out all of the strings that are passed as parameters to my program so I can see what they are.	If the program is called as "./stack this is a test, it will print: argv[1] = this argv[2] = is argv[3] = a argv[4] = test	```Stacks-story1> ./stack this is a test argv[1] = this argv[2] = is argv[3] = a argv[4] = test execute() called```
2		As a user, I want my program to notify me when I enter an incorrect parameter and tell me what the correct parameters are, and to notify me if I give a -h parameter, so I do not need to remember all of the parameters.	The call stack -h causes the help message to be printed and the program to execute; stack -s causes the help message to be printed and the program to terminate; stack causes the program to execute.xs	```Stacks-story2> ./stack -h stack: parameters are -h: print this message execute() called Stacks-story2> ./stack -x Error: unknown parameter x stack: parameters are -h: print this message Stacks-story2> ./stack execute() called```
3		As a programmer, I want to be able to push a character on a stack, so I can use the stack to match parentheses.	After function "push(stack *s, char x)" is the stack will have the value x as its top position.	```Test 3: push pushed a top: a :bottom pushed b top: b a :bottom pushed c top: c b a :bottom```
4		As a programmer, I want to pop a character from a stack so I can use the stack to match parentheses.	The function "push(stack s, char x)" will push a character onto the top of the stack; the function "char pop(stack s)" will pop an integer off the top of the stack.	```Test 4: pop pushed a top: a :bottom pushed b top: b a :bottom pushed c top: c b a :bottom popped c after pop top: b a :bottom popped b after pop top: a :bottom popped a after pop top: :bottom Error: pop(s, *val) empty stack s popped ? after pop top: :bottom```

| 5 | As a user, I want to be able to enter a set of brackets to see if they are matched at the command line, so I can use the program to match brackets. | If you enter "stack -b {[()]}" the program will print out that the bracket match. If you type "stack -b "{[()])" it will point to the mismatched bracket. | ```
Test 5: match_brackets
()
Brackets match
(]
 ^ mismatch
[]
Brackets match
{}
Brackets match
(a + b)
Brackets match
{[ x * (a + b)] }
Brackets match
{[ x * (a + b}] }
         ^ mismatch
``` |
| 6 | As a programmer, I want to reimplement a stack that contains integers, so I can write the function to evaluate postfix notation. | Calling "push(stack s, int i)" will push integer i onto the stack; calling "pop(stack s)" immediately afterwards will pop the integer and return the value | ```
Test 6: integer stack
pushed 1 top: 1 :bottom
pushed 2 top: 2 1 :bottom
pushed 3 top: 3 2 1 :bottom
popped 3 after pop top: 2 1 :bottom
popped 2 after pop top: 1 :bottom
popped 1 after pop top: :bottom
Error: pop(s, *val) empty stack s
popped -1 after pop top: :bottom
``` |
| 7 | As a programmer, I want to implement a program that will evaluate expressions based on postfix notation, so I can see how it is done. | If you enter stack -p "3 2 xs+" the program will print out 5. | ```
Test 7: evaluate_postfix
eval_postfix("1 2 +"): 3
eval_postfix("1 2 -"): -1
eval_postfix("3 2 *"): 6
eval_postfix("4 2 /"): 2
eval_postfix("20 4 +"): 24
eval_postfix("2 40 +"): 42
eval_postfix("20 40 +"): 60
eval_postfix("10 20 40 + -"): -50
eval_postfix("10 20 40 + /"): 0
eval_postfix("10 20 + 40 -"): -10
eval_postfix("10 20 * 40 -"): 160
``` |
| 8 | As a programmer, I want to be able to print a queue, so I can see what is in it. | The function "print_queue(char *msg, queue q) will print the contents of a queue. | ```
Test 8: print_queue
Test queue (a b c)  front: a b c :rear
``` |

9		As a programmer, I want to be able to enqueue a character in a queue, so I can learn to use queues.	The function "enqueue(queue q, char val) will return a queue that is identical to the one passed in except that the value of the char passed in is added to the rear of the queue	Test 9: enqueue Returning front: a :rear added 'a' front: a :rear Returning front: a b :rear added 'b' front: a b :rear Returning front: a b c :rear added 'c' front: a b c :rear
10		As a programmer, I want to be able to dequeue a character in a queue, so I can learn to use queues.	The function "dequeue(queue *q) will return the value at the front of the queue and modify the queue to remove that element.	Test 10: dequeue Test queue (a b c) front: a b c :rear Retrieved a queue is now front: b c :rear Returning front: b c d :rear added 'd' front: b c d :rear Retrieved b queue is now front: c d :rear Retrieved c queue is now front: d :rear Retrieved d queue is now front: :rear Error: dequeue of empty queue s Retrieved queue is now front: :rear