CS 32 Solutions Week 4

Concepts: Stacks, Queues

1. (5 mins) Given a string of '(', ')', '[', and ']', write a function `isValid` to check if the input string is *valid*. Validity is determined by each '(' having a corresponding ')', and each '[' having a corresponding ']', with parentheses being properly nested and brackets being properly nested.

Examples:

```
isValid("[()([])[[([][])]]]") // true
isValid("((([()])))")  // false, since not properly nested
isValid("(()))") // false, since no corresponding '(' for last ')'
isValid("()[]") // true
```

Solution:

```
// The idea here is that our stack maintains the sequence of
// opening parentheses and brackets, and removes an opening
// symbol upon seeing the matching closing one. Note that if we
// have a closing symbol, but the stack is empty or the top of
// the stack is not the the matching opening symbol, then we've
// encountered an invalid sequence of parentheses and brackets.
bool isValid(string symbols) {
  stack<char> openers;
  for (int k = 0; k != symbols.size(); k++) {
    char c = symbols[k];
    switch (c) {
      case '(':
      case '[':
        openers.push(c);
        break;
      case ')':
        if (openers.empty() || openers.top() != '(') return false;
        openers.pop();
        break;
      case ']':
        if (openers.empty() || openers.top() != '[') return false;
        openers.pop();
        break;
    }
  }
  return openers.empty();
}
```

2. (5 mins) Write a function `reverseQueue` that reverses a queue Q in place using a reference. Only the following standard operations are allowed on queue:

1) `Q.push(x)` : Add an item x to the back of the queue.
2) `Q.pop()` : Remove an item from the front of the queue.
3) `Q.front()` : Return the item at the front of the queue
4) `Q.empty()` : Check if the queue is empty or not.

You may use an additional data structure if you wish.

Example:

```
queue<int> q({10, 20, 30, 40, 50, 60, 70, 80, 90, 100});
reverseQueue(q)
// q should now be {100, 90, 80, 70, 60, 50, 40, 30, 20, 10}
```

Solution:

```
void reverseQueue(queue<int>& Q) {
  // use an auxiliary stack
  stack<int> S;

  while (!Q.empty()) {
    S.push(Q.front());
    Q.pop();
  }
  while (!S.empty()) {
    Q.push(S.top());
    S.pop();
  }
}
```

3. (5 mins) Evaluate the following postfix expression and show your work:

```
9 5 * 8 - 6 7 * 5 3 - / *
```

Solution:

```
45 8 - 42 2 / *
37 21 *
777
```

4. (15 mins) Write a function `findNextInts` that takes in two integer arrays of size *n: sequence* and *results*. This function assumes that *sequence* already contains a sequence of positive integers. For each position *i* (from 0 to *n*-1) of *sequence*, this function should find the **smallest index *j* such that *j* > *i* and *sequence[j]* > *sequence[i]*, and put *sequence[j]* in *results[i]***; if there is no such *j*, put -1 in *sequence[i]*. Try to do this without nested for loops both iterating over the array! (Hint: `#include <stack>`). In other words, we want to store the nearest value appearing later in the array than the current one that is greater than it in the result.

Example:

```
int seq[] = {2, 6, 3, 1, 9, 4, 7 }; // Only positive integers!
int res[7];
findNextInts(seq, res, 7);
for (int i = 0; i < 7; i++) { // Should print: 6 9 9 9 -1 7 -1
  cout << res[i] << " ";
}
cout << endl;
```

Notice that the last value in *results* will always be set to -1 since there are no integers in *sequence* after the last one!

Solution:

```
void findNextInts(const int sequence[], int results[], int n) {
  if (n <= 0)
    return;

  stack<int> s;

    // push the first index to stack
  s.push(0);

    // iterate for rest of the elements
  for (int i = 1; i < n; i++) {
    int current = sequence[i];

      // Fill in results for preceding unfilled items
      // that are less than current.
    while (!s.empty() && current > sequence[s.top()]) {
      results[s.top()] = current;
      s.pop();
    }

    s.push(i);
  }

    // Remaining items don't have a later greater value
  while (!s.empty()) {
    results[s.top()] = -1;
    s.pop();
  }
}
```

5. (10 mins) Implement a Stack class using only queues as data structures. This class should implement the *empty*, *size*, *top*, *push*, and *pop* member functions, as specified by the standard library's implementation of stack. (The implementation will not be very efficient.)

Solution:

```
class Stack {
  // This implementation of Stack accepts only int. See if you
  // can make an implementation with templates!
 public:
  bool empty() const;
  size_t size() const;
  int top() const;
  void push(const int& value);
  void pop();

 private:
  queue<int> storage;
};

bool Stack::empty() const { return storage.empty(); }

size_t Stack::size() const { return storage.size(); }

int Stack::top() const { return storage.back(); }

void Stack::push(const int& value) { storage.push(value); }

void Stack::pop() {
  // Note that this causes a runtime error if
  // storage is empty. This matches how calling pop() on an empty
  // C++ STL stack causes a runtime error.
  int limit = storage.size() - 1;
  for (int n = 0; n < limit; n++) {
    storage.push(storage.front());
    storage.pop();
  }  // circling the entire queue
  storage.pop();
}  // stack is LIFO, queue is LILO
```

6. (16 mins) Implement a Queue class using only stacks as data structures. This class should implement the *empty*, *size*, *front*, *back*, *push*, and *pop* member functions, as specified by the standard library's implementation of queue. (The implementation will not be very efficient.)

Solution:

```
class Queue {
  // This implementation of Queue accepts only int. See if you can
  // make an implementation with templates!
  // pushStorage is a stack that contains items when they're first
  // pushed. popStorage is another stack, and we move items from
  // pushStorage to popStorage when we want to pop from the queue
 public:
  bool empty() const;
  size_t size() const;
  int front() const;
  int back() const;
  void push(const int& value);
  void pop();

 private:
  // move items from pushStorage to popStorage while leaving back
  // item within pushStorage
  void moveItems();

  // storage for pushing items with one exception: always includes
  // back item if available
  stack<int> pushStorage;
  // storage for popping items: always includes front item
  stack<int> popStorage;
};

bool Queue::empty() const { return pushStorage.empty() &&
popStorage.empty(); }

size_t Queue::size() const { return pushStorage.size() + popStorage.size();
}

int Queue::front() const { return popStorage.top(); }

int Queue::back() const {
  if (size() == 1) return popStorage.top();
```

```
  return pushStorage.top();
}

void Queue::push(const int& value) {
  if (size() > 0)
    pushStorage.push(value);
  else
    popStorage.push(value);
}

void Queue::pop() {
  // Note that this causes a runtime error if
  // popStorage and pushStorage are empty (i.e. the Queue has no
  // items in it). This matches how calling pop() on an empty
  // C++ STL queue causes a runtime error.
  if (popStorage.size() > 0) {
    popStorage.pop();
    if (popStorage.size() == 0 && pushStorage.size() > 0) moveItems();
  } else {
    moveItems();
    popStorage.pop();
  }
}

void Queue::moveItems() {
  int temp = pushStorage.top();
  bool backExists = false;
  if (pushStorage.size() > 1) {
    pushStorage.pop();
    backExists = true;
  }

  while (pushStorage.size() > 0) {
    popStorage.push(pushStorage.top());
    pushStorage.pop();
  }

  if (backExists) pushStorage.push(temp);
}
```