Queues and Unit Testing

Shreya Shankar

Stanford CS 106B

3 July 2018

Based on slides created by Ashley Taylor, Marty Stepp, Chris Gregg, Keith Schwarz, Julie Zelenski, Jerry Cain, Eric Roberts, Mehran Sahami, Stuart Reges, Cynthia Lee, and others.
Recap
Big O

• Idea: measures algorithmic **efficiency**
  • How *long* does the program take to run?
• Algorithms are better if they take less time
• Number of statements in code can depend on how large the input is – we normally say the input size is $N$
• Only the biggest power of $N$ matters
• To calculate big O, work from innermost indented code out
Example

Q: What is the big $O$?

Vector<int> v;
for (int x = 1; x <= N; x += 2) {
    v.add(x);
}
while (! v.isEmpty()) {
    cout << v.remove(0) << endl;
}

A:
Example

Q: What is the big O?

Vector<int> v;
for (int x = 1; x <= N; x += 2) {
    v.add(x);
}
while (! v.isEmpty()) {
    cout << v.remove(0) << endl;
}

A: $O(n^2)$
Stacks

• **stack**: ADT that only allows a user to push an element and peek or pop the last element
  • ”Last-in, first-out”
  • O(1) for these operations

• **Basic stack operations**:
  • **push**: add an element to the end of the stack
  • **pop**: remove and return the last element in the stack
  • **peek**: return (but do not remove) the last element in the stack
Queues
Motivation

- There is behavior that stacks fail to model well
  - What if we want to remove from front instead of from back?
- Can we model first-in, first-out behavior using other ADTs?
  - Vector: removing from the beginning of a list is an $O(n)$ operation
  - Stack: need two stacks to get the first element (food for thought: how to do this?)
  - Grid: nah
- So, we need a new ADT...
Queues

• `queue`: ADT that retrieves elements in the order they were added
  • First-in, First-out ("FIFO")
  • Elements are stored in order of insertion, no indexes
  • Can add only to the end of a queue and can only examine or remove the front
• Basic queue operations:
  • `enqueue`: add an element to the back
  • `dequeue`: remove the front element
  • `peek`: examine the front element
Class methods

#include "queue.h"

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>q.dequeue()</td>
<td>O(1)</td>
<td>Removes front value and returns it; throws error if queue is empty</td>
</tr>
<tr>
<td>q.enqueue(value)</td>
<td>O(1)</td>
<td>Places given value at back of queue</td>
</tr>
<tr>
<td>q.isEmpty()</td>
<td>O(1)</td>
<td>Returns true if queue has no elements</td>
</tr>
<tr>
<td>q.peek()</td>
<td>O(1)</td>
<td>Returns front value without removing; throws an error if queue is empty</td>
</tr>
<tr>
<td>q.size()</td>
<td>O(1)</td>
<td>Returns number of elements in queue</td>
</tr>
</tbody>
</table>

Table 1: The Queue class in Stanford's C++ libraries.

- Food for thought: what isn't an O(1) operation?
Example

```cpp
Queue<int> q; // {} front -> back
cout << q.dequeue() << endl; // 42 (q is {-3, 17})
```

```cpp
q.enqueue(42); // {42}
q.enqueue(-3); // {42, -3}
q.enqueue(17); // {42, -3, 17}
cout << q.dequeue() << endl; // 42 (q is {-3, 17})
cout << q.peek() << endl; // -3 (q is {-3, 17})
cout << q.dequeue() << endl; // -3 (q is {17})
```
Applications

• Real-world examples
  • Middle school lunch lines (no cutting!)
  • Escalators (when someone is taking up the whole step)
  • Anything first-come-first-serve

• Computers
  • Sending jobs to a printer
  • Uploading photos on social media
  • Call services – being on hold
Exercise

Q: What is the output of the following code?

```cpp
Queue<int> queue;
for (int i = 1; i <= 6; i++) {
    queue.enqueue(i);
} // {1, 2, 3, 4, 5, 6}

for (int i = 0; i < queue.size(); i++) {
    cout << queue.dequeue() << " ";
}
```

```cpp
cout << queue << " size " << queue.size() << endl;
```

A. 1 2 3 4 5 6 {} size 0
B. 1 2 3 {4, 5, 6} size 3
C. 1 2 3 4 5 6 {1, 2, 3, 4, 5, 6} size 6
D. none of the above
Exercise

**Q:** What is the output of the following code?

```cpp
Queue<int> queue;
for (int i = 1; i <= 6; i++) {
    queue.enqueue(i);
} // {1, 2, 3, 4, 5, 6}

for (int i = 0; i < queue.size(); i++) {
    cout << queue.dequeue() << " ";
}
cout << queue << " size " << queue.size() << endl;
```

A. 1 2 3 4 5 6 {} size 0  
B. 1 2 3 {4, 5, 6} size 3  
C. 1 2 3 4 5 6 {1, 2, 3, 4, 5, 6} size 6  
D. none of the above
Exercise

Q: Write a function **stutter** that accepts a queue of integers and replaces every element with two copies of itself. For example, \{1, 2, 3\} becomes \{1, 1, 2, 2, 3, 3\}. 
Exercise

Q: Write a function `stutter` that accepts a queue of integers and replaces every element with two copies of itself. For example, \{1, 2, 3\} becomes \{1, 1, 2, 2, 3, 3\}.

```cpp
void stutter(Queue<int>& q) {
    int size = q.size();
    for (int i = 0; i < size; i++) {
        int n = q.dequeue();
        q.enqueue(n);
        q.enqueue(n);
    }
}
```
Helpful hints for queues

• Don't use size() directly

```java
int size = q.size();
for (int i = 0; i < size; i++) {
    // do something with q.dequeue();
    // (including possibly re-adding it to the queue)
}
```

• As with stacks, must pull contents out of queue to view them

```java
while (!q.isEmpty()) {
    // do something with q.dequeue();
}
```
More Queues
Mixing stacks and queues

How can we reverse the order of elements in a queue?

```cpp
Queue<int> q {1, 2, 3}; // q={1, 2, 3}
Stack<int> s;

while (!q.isEmpty()) { // transfer queue to stack
    s.push(q.dequeue()); // q={} s={1, 2, 3}
}

while (!s.isEmpty()) { // transfer stack to queue
    q.enqueue(s.pop()); // q={3, 2, 1} s={}
}

cout << q << endl; // {3, 2, 1}
```
Q: Write a function `mirror` that accepts a queue of strings and appends the queue’s contents to itself in reverse order. For example, `{"a", "b", "c"}` becomes `{"a", "b", "c", "c", "b", "a"}`.
Exercise

Q: Write a function `mirror` that accepts a queue of strings and appends the queue’s contents to itself in reverse order.

```cpp
void mirror(Queue<string>& q) {
    Stack<string> s;
    int size = q.size();
    for (int i = 0; i < size; i++) {
        string str = q.dequeue();
        s.push(str);
        q.enqueue(str);
    }
    while (!s.isEmpty()) {
        q.enqueue(s.pop());
    }
}
```
Deques

- **deque**: double-ended queue (pronounced "deck")
  - Can add/remove from either end
  - Combines many of the benefits of stack and queue
- **Basic deque operations**:
  - enqueueFront, enqueueBack
  - dequeueFront, dequeueBack
  - peekFront, peekBack
- Get queue and stack functionality in one data structure!

```
enqueuFront, dequeueFront, peekFront
```

```
enqueuBack, dequeueBack, peekBack
```

```
1 2 3
```

Unit Testing
Early Google Maps
Evolution of Google Maps: real-world technology

- Version 1: could only look up a city in the US and see roads in it – ”paper atlas in living form” (Recode article)
- Then, added functionality to calculate directions from one place to another
- Added ability to search for local businesses
- Added satellite imagery for people to view their own houses, making Maps very popular
- A couple years into development, the team completely rewrote all code to make it run faster
- Problems: How to make sure all this code is functional? How to make sure functionality persists throughout all these changes?
Motivation

- **Code in the real world:**
  - You are not the only one reading the code you write
  - Code is frequently updated to offer new functionality, fix old bugs, etc.
  - We often know what functionality we need, and we want to make sure our code hits all the "edge cases"

- **Unit testing** is a method for testing small pieces or "units" of source code for a larger piece of software

- Usually a series of functions to run

- Benefits of unit testing:
  - Limits your code to only what is necessary and finds bugs early
  - Easily preserves functionality when code is changed
Unit testing basics

- Unit tests are usually a list of functions in a file, where each function tests a **small piece of functionality**
  - We don’t want to test lots of things in one unit test function
- Each unit test (function) has a way of indicating ”pass” or ”failure”
- Each programming language has its own style of unit testing
- Unit tests are usually designed before code is written
- Unit testing is usually automated – before new code can enter the codebase, the test file is run and all unit tests must pass
Common unit test macros

• **assert**: throws error if condition inside assert statement is false

```c
void checkNumStreetsInBigCities(Location &loc) {
    int numStreets = getNumStreets(loc);
    assert(numStreets > 100); // will crash program
    if false
}
```

• **expect**: displays error when condition inside expect statement is false, but other tests continue to run

• Use **assert** when test is critical, **expect** otherwise
Exercise

Recall grids: Write a function `getPossibleMoves` that accepts a grid and a row/column pair (row, col) as parameters, and returns a vector of empty locations that a knight can legally move to from (row, col).

- Recall that a knight makes an "L" shaped move, going 2 squares in one dimension and 1 square in the other.
- `getPossibleMoves(board, 1, 2)` returns `Vector<Point>{(0, 0), (2, 0), (3, 3), (2, 4)}`
Exercise

getPossibleMoves(board, 1, 2) returns Vector<Point> 
{(0, 0), (2, 0), (3, 3), (2, 4)}
Exercise

Step one: think about all subunit functionalities (for lecture purposes, we'll list a few)

1. Need to be able to process all valid moves from a given location
2. Need to make sure our valid moves don’t have our pieces in them
3. Need to make sure our valid moves processed don’t go out of bounds
4. Need to make sure user is passing a location in bounds
Solution

Vector<Point> get Possible Moves(Grid<string>& board, int row, int col) {
    Vector<Point> validMoves;
    if (!board.inBounds(row, col) || board[row][col] != "knight") return validMoves;

    for (int xDir = -1; xDir <= 1; xDir += 2) {
        for (int yDir = -1; yDir <= 1; yDir += 2) {
            for (int xLen = 1; xLen <= 2; xLen++) {
                int yLen = 3 - xLen;
                Point point = (xLen * xDir, yLen * yDir);
                if (board.inBounds(row + point.getX(), col + point.getY())
                    && board[row + point.getX()][col + point.getY()] == "")
                    validMoves.add(point);
            }
        }
    }

    return validMoves;
}
Testing exercise

Prototype for functionality 1:

```c++
const Grid<string> TEST_BOARD;

void checkNumPossibleMoves() {
    /* Get possibleMoves: vector of possible locations knight could end up at */
    Vector<Point> possibleMoves = getPossibleMoves(
        TEST_BOARD, 1, 2);

    // No more than 8 possible locations to move to
    assert(possibleMoves.size() <= 8);
}
```
Testing exercise

Prototype for functionality 2:

```cpp
const Grid<string> TEST_BOARD;
void checkNoPiecesInPossibleMoves() {
    Vector<Point> possibleMoves = getPossibleMoves
        (TEST_BOARD, 1, 2);
    Vector<Point> currentLocs =
        getCurrentPieceLocations();

    // Check to make sure possibleMoves doesn’t contain
    // our pieces
    for (Point pLoc: possibleMoves) {
        for (Point cLoc: currentLocs) {
            assert(pLoc != cLoc);
        }
    }
}
```
Testing exercise

Prototype for functionality 3:

```cpp
const Grid<string> TEST_BOARD;

void checkPossibleMovesInBounds() {
    Vector<Point> possibleMoves = getPossibleMoves
        (TEST_BOARD, 1, 2);

    // Check to make sure possibleMoves are in bounds
    for (Point pLoc: possibleMoves) {
        assert(TEST_BOARD.inBounds(pLoc.getX(),
                                   pLoc.getY()));
    }
}
```
Testing exercise

Prototype for functionality 4:

```cpp
const Grid<string> TEST_BOARD;

void checkBadUserLoc () {
    // (-1, -1) is a bad location
    Vector<Point> possibleMoves = getPossibleMoves
        (TEST_BOARD, -1, -1);

    // There are no moves from invalid location
    assert(possibleMoves.size() == 0);
}
```
Testing takeaways

• Think about tests early, and think about them often!
• This class does not heavily focus on unit tests in assignments, but software engineering in general does.
• Unit tests are a lot of work to write! But they’re valuable when code is used in production and frequently updated.

Meme taken from me.me
Thank you!

- Any questions? Email me at shreya@cs.stanford.edu or come to my office hours today (Tuesday) from 1:30-3:30 PM in Gates B02
- **Assignment 1 (Life)** due Thursday
- Today is last day of LaIR before assignment is due
- No class tomorrow – have a great 4th of July!