Queues and Stacks

Sandra Batista, Mark Redekopp, and David Kempe
Orders collection of items,

- Each item has an index and there is a front and back (start and end)
- Duplicates allowed (i.e. in a list of integers, the value 0 could appear multiple times)
- Accessed based on their position (list[0], list[1], etc.)

What are some operations you perform on a list?
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Input(s)</th>
<th>Output(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>Add a new value at a particular location shifting others back</td>
<td>Index</td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td>Remove value at the given location</td>
<td>Index</td>
<td>Value at location</td>
</tr>
<tr>
<td>get</td>
<td>Get value at given location</td>
<td>Index</td>
<td>Value at location</td>
</tr>
<tr>
<td>set</td>
<td>Changes the value at a given location</td>
<td>Index &amp; Value</td>
<td></td>
</tr>
<tr>
<td>empty</td>
<td>Returns true if there are no values in the list</td>
<td></td>
<td>bool</td>
</tr>
<tr>
<td>size</td>
<td>Returns the number of values in the list</td>
<td></td>
<td>Size_t</td>
</tr>
<tr>
<td>push_back</td>
<td>Add a new value to the end of the list</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>find</td>
<td>Return the location of a given value</td>
<td>Value</td>
<td>Index</td>
</tr>
</tbody>
</table>
Queues and Stacks

Two specialized List ADTs

**Queue (FIFO)**
- Items leave from the front (pop_front)
- Items enter at the back (push_back)

**Stack (LIFO)**
- Items enter and leave from the same side (i.e. the top)
- Items enter at the back (push_back)
- Items leave from the front (pop_front)
# Queue & Stack Operations

<table>
<thead>
<tr>
<th><strong>Queues</strong></th>
<th><strong>Notes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations Relative to Lists</td>
<td><strong>insert</strong></td>
</tr>
<tr>
<td></td>
<td><strong>front</strong></td>
</tr>
<tr>
<td></td>
<td><strong>set</strong></td>
</tr>
<tr>
<td></td>
<td><strong>size</strong></td>
</tr>
<tr>
<td></td>
<td><strong>pop_front</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Stacks</strong></td>
<td><strong>Notes</strong></td>
</tr>
<tr>
<td>Operations Relative to Lists</td>
<td><strong>insert</strong></td>
</tr>
<tr>
<td></td>
<td><strong>top</strong></td>
</tr>
<tr>
<td></td>
<td><strong>set</strong></td>
</tr>
<tr>
<td></td>
<td><strong>size</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
First-In, First-Out (FIFOs)

QUEUE ADT
Queue ADT

Queue – A list of items where insertion only occurs at the back of the list and removal only occurs at the front of the list

Queues are FIFO (First In, First Out)
- Items at the back of the queue are the newest
- Items at the front of the queue are the oldest
- Elements are processed in the order they arrive

Examples from everyday life:
- Printing jobs in a printer
- Customers in line for cashier at store
- Diners waiting to be seated at restaurant
- Helpful to decouple producers and consumers
Queue Operations

- `push_back(item)` – Add an item to the back of the Queue
- `pop_front()` - Remove the front item from the Queue
- `front()` - Get a reference to the front item of the Queue (don't remove it though!)
- `size()` - Number of items in the Queue
- `empty()` - Check if the Queue is empty
A Queue Interface

This abstract Queue class specifies the Queue ADT operations

Any derived implementation must implement these public member functions to be instantiated

Queue Error Conditions

- **Queue Underflow** – The condition when `pop_front` is called on an empty Queue

- **Queue Overflow** – The condition when a queue has limited capacity and `push_back` is called when full

```cpp
class IntQueue {
    public:
        virtual int front() = 0;
        virtual bool empty() = 0;
        virtual size_t size() = 0;
        virtual void push_back(int v) = 0;
        virtual void pop_front() = 0;
};
```
This class uses STL List to implement a queue.

It is an example of implementing a queue with a doubly linked list.

It is basically a wrapper class for calls directly to STL List.

```
// This class implements the queue ADT interface using STL list.
// Remember STL list is a doubly linked list implementation.

class ListIntQueue : public IntQueue {
    public:
        int front() {
            // Add proper error handling
            if (queue.empty()) return -1;
            return queue.front();
        }
        bool empty() { return queue.empty(); }  
        size_t size() { return queue.size(); }  
        void push_back(int value) { queue.push_back(value); }  
        void pop_front() {
            if (!empty()) {
                queue.pop_front();
            }
        }
    }

    private:
        std::list<int> queue;
};
```
A STL Vector Queue Class

This class uses STL Vector to implement a queue.

It is an example of implementing a queue with a dynamically sized array.

It is basically a wrapper class for calls directly to STL Vector.

```cpp
// This class implements the queue ADT interface using STL vector.
// Remember STL vector is a dynamically sized array

class VectorIntQueue : public IntQueue {

public:
    int front() {
        // add appropriate error handling
        if (this->empty()) { return -1; }
        return queue.front();
    }

    bool empty() { return queue.empty(); }
    size_t size() { return queue.size(); }
    void push_back(int value) { queue.push_back(v); }
    void pop_front() {
        if (!empty()) {
            queue.erase(queue.begin());
        }
    }

private:
    std::vector<int> queue;
};
```
STACK ADT

Last-In, First-Out (LIFOs)
Stack ADT

Stack: A list of items where insertion and removal only occurs at one end of the list

Everyday examples:
- A stack of boxes
- A PEZ dispenser
- Your e-mail inbox

Stacks are LIFO
- Newest item at top
- Oldest item at bottom
Stack Operations

- `push(item)` - Add an item to the top of the Stack
- `pop()` - Remove the top item from the Stack
- `top()` - View the top item on the Stack
- `size()` - Get the number of items in the Stack
- `empty()` - Check if stack has items

A stack implementation needs at least the following member data:
- A list of items
- Top Pointer/Index
Stack Axioms

For all stacks, s:

• $s.\text{push}(\text{item}).\text{top()} = \text{item}$
• $s.\text{push}(\text{item}).\text{pop()} = s$

Let’s draw the stack for these operations:

• $s.\text{push}(5).\text{push}(4).\text{pop()}.\text{top()}$
A Stack Class

A sample class interface for a Stack

Stack Error Conditions

• Stack Underflow – The condition when pop is called on an empty stack
• Stack Overflow – The condition when push is called on a full stack

```cpp
#ifndef STACKINT_H
#define STACKINT_H

class StackInt {
  public:
  StackInt();
  ~StackInt();
  size_t size() const;
  bool empty() const;
  void push(const int& value);
  void pop();
  int const& top() const;

  private:
    // Use a dynamically sized array
    // or linked list
    list<int> items; // or
    // vector<int> items
    size_t top_index;
};
#endif
```
Stack Example: Reversing a string

Reverse a string trace:

```cpp
#include <iostream>
#include <string>
#include <stack>
using namespace std;

int main()
{
    stack<char> s;

    string word;
    cout << "Enter a word: ";
    getline(cin, word);

    for(int i=0; i < word.size(); i++)
        s.push(word.at(i));

    while(!s.empty())
    {
        cout << s.top();
        s.pop();
    }
}
```

Type in: "hello"
Output: "olleh"
Stack Example: Checking Parentheses

➢ Check whether an expression is properly parenthesized with '(', '[', '{', '}', ']', ')'
   - Correct: \((7 * [8 + [9/{5-2}]]))
   - Incorrect: \(7*8\)
   - Incorrect: \(7*8]\)
➢ Note: The last parentheses started should be the first one completed
➢ Approach
  ▪ Scan character by character of the expression string
  ▪ For each current character that is an open-paren '(', '[', '{': push it on the stack.
  ▪ When current character is close-paren a ')', ']', '}':
    if top equals matching open paren, pop from stack.
    otherwise  ERROR!
  ▪ If no characters left in string and stack empty, accept.
  ▪ If no characters left in string and stack not empty, reject.
Stack Example: Checking Parentheses Trace

- Check whether an expression is properly parenthesized with 
  
  `'(', '[', '{', '}', ']', ')'`

  - Incorrect: (7*8
  - Incorrect: (7*8]

- Let’s trace for (7*8 and (7*8] below:

  (7 * [8 + [9/{5-2}]])
  
  ( [ [ { } ] ] )
Stack Example: Checking Parentheses Trace

- Check whether an expression is properly parenthesized with '(', '[', '{', '}', ']', ')'
  - Correct: \((7 * [8 + [9/{5-2}]])\)
- Let’s trace for \((7 * [8 + [9/{5-2}]])\)
Stack Example: Evaluating mathematical expressions

➢ How do we modify our approach if we also want to evaluate the mathematical expression?

➢ Approach Modifications
- Scan character by character of the expression string
- **push all chars** on the stack
- When you encounter a '}', ']', ')', pop until find matching opening paren.
- Evaluate expression popped and push value on stack.
- (If malformed expressions permitted and no matching opening paren found, then reject expression.)

\[(7 \times [4 + 2 + 3])\]
Queue with two stacks

To enqueue(x), push x on stack 1

To dequeue()

- If stack 2 empty, pop everything from stack 1 and push onto stack 2.
- Pop stack 2