CSCI 104
Queues and Stacks

Mark Redekopp
David Kempe
Lists

• Ordered collection of items, which may contain duplicate values, usually accessed based on their position (index)
  – Ordered = 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?
# List Operations

<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 : int</td>
<td>Value</td>
</tr>
<tr>
<td>remove</td>
<td>Remove value at the given location</td>
<td>Index : int</td>
<td>Value at location</td>
</tr>
<tr>
<td>get / at</td>
<td>Get value at given location</td>
<td>Index : int</td>
<td>Value at location</td>
</tr>
<tr>
<td>set</td>
<td>Changes the value at a given location</td>
<td>Index : int</td>
<td>Value</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>int</td>
</tr>
<tr>
<td>push_back / append</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>Int : Index</td>
</tr>
</tbody>
</table>
Specialized Lists

STACKS AND QUEUES
Stacks & Queues

• Lists are good for storing generic sequences of items, but they can be specialized to form other useful structures

• What if we had a List, but we restricted how insertion and removal were done?
  – Stack – Only ever insert/remove from one end of the list
  – Queue – Only ever insert at one end and remove from the other
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
  – Like waiting in line for a cashier at a store
• 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
A Queue Visual

Items leave from the front (pop_front)

Items enter at the back (push_back)
Queue Operations

• What member functions does a Queue have?
  – `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 Class

• A sample class interface for a Queue

• Queue Error Conditions
  – Queue Underflow – The name for the condition where you call pop on an empty Queue
  – Queue Overflow – The name for the condition where you call push on a full Queue (a Queue that can't grow any more)
    • This is only possible for Queues that are backed by a bounded list

```cpp
#ifndef QUEUEINT_H
#define QUEUEINT_H

class QueueInt {
public:
    QueueInt();
    ~QueueInt();
    int size() const;
    void push_back(const int& value);  // enqueue
    void pop_front();  // dequeue
    int const & front() const;
    bool empty() const;
private:
    // ???
};

#endif
```
Other Queue Details

- How should you implement a Queue?
  - Back it with an ArrayList
  - Back it with a linked list
  - Back it with a vector
  - Inherit from a linked list
  - Which is best?

<table>
<thead>
<tr>
<th></th>
<th>Push_back</th>
<th>Pop_front</th>
<th>Front()</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LinkedList</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Singly-linked w/ tail ptr)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LinkedList</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Singly-linked w/o tail ptr)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Other Queue Details

• How should you implement a Queue?
  – Back it with an ArrayList
  – Back it with a linked list
  – Back it with a vector
  – Inherit from a linked list
  – Which is best?

<table>
<thead>
<tr>
<th></th>
<th>Push_back</th>
<th>Pop_front</th>
<th>Front()</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>LinkedList (Singly-linked w/ tail ptr)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>LinkedList (Singly-linked w/o tail ptr)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Queue Applications

• Print Jobs
  – Click “Print” on the computer is much faster than actually printing (build a backlog)
  – Each job is processed in the order it's received (FIFO)
  – Why would you want a print queue rather than a print stack
• Seating customers at a restaurant
• Anything that involves "waiting in line"
• Helpful to decouple producers and consumers
Last-In, First-Out (LIFOs)

STACK ADT
Stack ADT

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

• Examples:
  – A stack of boxes where you have to move the top one to get to ones farther down
  – A spring-loaded plate dispenser at a buffet
  – A PEZ dispenser
  – Your e-mail inbox

• Stacks are LIFO
  – Newest item at top
  – Oldest item at bottom
Stack Operations

• What member functions does a Stack have?
  – push(item) – Add an item to the top of the Stack
  – pop() - Remove the top item from the Stack
  – top() - Get a reference to the top item on the Stack (don't remove it though!)
  – size() - Get the number of items in the Stack

• What member data does a Stack have?
  – A list of items
  – Top/Last Item Pointer/Index
A Stack Class

- A sample class interface for a Stack
- How should you implement a Stack?
  - Back it with an array
  - Back it with a vector
  - Back it with a linked list
  - Inherit from linked list
  - Which is best?
- Stack Error Conditions
  - Stack Underflow – The name for the condition where you call pop on an empty Stack
  - Stack Overflow – The name for the condition where you call push on a full Stack (a stack that can't grow any more)

```cpp
#ifndef STACKINT_H
#define STACKINT_H

class StackInt {
public:
    StackInt();
    ~StackInt();
    int size() const;
    bool empty() const;
    void push(const int& value);
    void pop();
    int const & top() const;
};
#endif
```
Array Based Stack

• A sample class interface for a Stack
• If using an array list, which end should you use as the "top"?
  – Front or back?
• If using a linked list, which end should you use?
  – If you just use a head pointer only?
  – If you have a head and tail pointer?

```cpp
#ifndef STACKINT_H
#define STACKINT_H

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

private:
    AListInt mylist_;  
    // or LListInt mylist_;
};
#endif
```
Stack Examples

- Reverse a string

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

int main()
{
    StackChar 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"
Another Stack Example

- Depth First Search (See Graph Traversals later in this semester)
- Use a stack whenever you encounter a decision, just pick and push decision onto stack. If you hit a dead end pop off last decision (retrace steps) and keep trying, etc.
  - Straight or Left
    - Choose straight...dead end
    - Pop straight and make next choice...left
      - Straight or Right...etc.

http://www.pbs.org/wgbh/nova/einstein/images/lrk-maze.gif
Stack Usage Example

• 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
  – Each time you hit an open-paren: '(', '[', '{' push it on the stack
  – When you encounter a '}', ']', '}' the top character on the stack should be the matching opening paren type, otherwise ERROR!
Double-ended Queues

DEQUE ADT
The Deque ADT

- **Double-ended queues**
- Equally good at pushing and popping on either end
STL Deque Class

- Similar to vector but allows for push_front() and pop_front() options
- Useful when we want to put things in one end of the list and take them out of the other

```cpp
#include <iostream>
#include <deque>
using namespace std;

int main()
{
    deque<int> my_deq;
    for(int i=0; i < 5; i++)
    {
        my_deq.push_back(i+50);
    }
    cout << "At index 2 is: " << my_deq[2] ;
    cout << endl;
    for(int i=0; i < 5; i++)
    {
        int x = my_deq.front();
        my_deq.push_back(x+10);
        my_deq.pop_front();
    }
    while( !my_deq.empty()){
        cout << my_deq.front() << " ";
        my_deq.pop_front();
    }
    cout << endl;
}
```

1. `my_deq` with contents: 50 51 52 53 54
2. After the first iteration: 51 52 53 54 60
3. After all iterations: 60 61 62 63 64
STL Vector vs. Deque

• `std::vector` is essentially a Dynamic Array List
  – Slow at removing and inserting at the front or middle
  – Fast at adding/remove from the back
  – Implies it could be used well as a (stack / queue)

• `std::deque` gives fast insertion and removal from front and back along with fast random access (i.e. `at(i)`)
  – Almost has "look and feel" of linked list with head and tail pointers providing fast addition/removal from either end
  – Implies it could be used well as a (stack / queue)
  – Practically it is likely implemented as a circular array buffer
Circular Buffers

• Take an array but imagine it wrapping into a circle to implement a deque

• Setup a head and tail pointer
  – Head points at first occupied item, tail at first free location
  – Push_front() and pop_front() update the head pointer
  – Push_back() and pop_back() update the tail pointer

• To overcome discontinuity from index 0 to MAX-1, use modulo operation
  – Index = 7; Index++ should cause index = 0
  – index = (index + 1)%MAX
  – Index = 0; Index-- should cause index = 7
  – if(--index < 0) index = MAX-1;

• Get item at index i
  – It's relative to the head pointer
  – Return item at (head + i)%MAX