Stacks & Queues

Data structures and Algorithms

Acknowledgement:
These slides are adapted from slides provided with *Data Structures and Algorithms in C++* Goodrich, Tamassia and Mount (Wiley, 2004)
Outline and Reading

• The Stack ADT (§5.1.1)
  • Applications of Stacks (§5.1.5)
  • Array-based implementation (§5.1.2)
  • List-based stack (§5.1.3)
  • Applications (§5.1.5)

• The Queue ADT (§5.2.1)
  • Implementation with a circular array (§5.2.2)
  • List-based queue (§5.2.3)
  • Round Robin schedulers (§5.2.4)
Stacks
The Stack ADT

- Stack ADT stores arbitrary objects
- Insertions and deletions follow last-in first-out (LIFO) scheme
- Main stack operations:
  - `push(object)`: inserts an element
  - `pop()`: removes and returns the last inserted element
- Auxiliary stack operations:
  - `top()`: returns the last inserted element without removing it
  - `size()`: returns the number of elements stored
  - `isEmpty()`: returns a Boolean value indicating whether no elements are stored
## Stack Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>output</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(8)</td>
<td>-</td>
<td>(8)</td>
</tr>
<tr>
<td>push(3)</td>
<td>-</td>
<td>(3, 8)</td>
</tr>
<tr>
<td>pop()</td>
<td>3</td>
<td>(8)</td>
</tr>
<tr>
<td>push(2)</td>
<td>-</td>
<td>(2, 8)</td>
</tr>
<tr>
<td>push(5)</td>
<td>-</td>
<td>(5, 2, 8)</td>
</tr>
<tr>
<td>top()</td>
<td>5</td>
<td>(5, 2, 8)</td>
</tr>
<tr>
<td>pop()</td>
<td>5</td>
<td>(2, 8)</td>
</tr>
<tr>
<td>pop()</td>
<td>2</td>
<td>(8)</td>
</tr>
<tr>
<td>pop()</td>
<td>8</td>
<td>()</td>
</tr>
<tr>
<td>pop()</td>
<td>&quot;error&quot;</td>
<td>()</td>
</tr>
<tr>
<td>push(9)</td>
<td>-</td>
<td>(9)</td>
</tr>
<tr>
<td>push(1)</td>
<td>-</td>
<td>(1, 9)</td>
</tr>
</tbody>
</table>
Stack Interface in C++

• Interface corresponding to our Stack ADT
• Requires the definition of class EmptyStackException
• Corresponding STL construct: stack

template <typename Object>
class Stack {
public:
    int size() const;
    bool isEmpty() const;
    Object& top() throw(EmptyStackException);
    void push(const Object& o);
    Object pop() throw(EmptyStackException);
};
Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception.
- Exceptions are said to be “thrown” by an operation that cannot be executed.
- In the Stack ADT, operations \texttt{pop} and \texttt{top} cannot be performed if the stack is empty.
- Attempting the execution of \texttt{pop} or \texttt{top} on an empty stack throws an \texttt{EmptyStackException}. 
Applications of Stacks

• Direct applications
  ■ Page-visited history in a Web browser
  ■ Undo sequence in a text editor
  ■ Saving local variables when one function calls another, and this one calls another, and so on.

• Indirect applications
  ■ Auxiliary data structure for algorithms
  ■ Component of other data structures
C++ Run-time Stack

- The C++ run-time system keeps track of the chain of active functions with a stack.
- When a function is called, the run-time system pushes on the stack a frame containing:
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack.

```cpp
main() {
    int i;
    i = 5;
    foo(i);
}

foo(int j) {
    int k;
    k = j+1;
    bar(k);
}

bar(int m) {
    ...
}
```
Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

Algorithm `size()`
return $t + 1$

Algorithm `pop()`
if `isEmpty()` then
    throw `EmptyStackException`
else
    $t \leftarrow t - 1$
    return $S[t + 1]$
Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a `FullStackException`
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm `push(o)`

```plaintext
if t = S.length - 1 then
  throw FullStackException
else
  t ← t + 1
  S[t] ← o
```

Stacks & Queues
Performance and Limitations
- array-based implementation of stack ADT

• **Performance**
  - Let $n$ be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$

• **Limitations**
  - The maximum size of the stack must be defined *a priori* and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception
Array-based Stack in C++

template <typename Object>
class ArrayStack {
private:
    int capacity;       // stack capacity
    Object *S;          // stack array
    int t;              // top of stack
public:
    ArrayStack(int c) {
        capacity = c;       // stack capacity
        S = new Object[capacity];
        t = -1;             // top of stack
    }
    bool isEmpty() {
        return (t < 0);
    }
    Object pop() throw(EmptyStackException) {
        if(isEmpty())
            throw EmptyStackException
                ("Access to empty stack");
        return S[t--];
    }
    // … (other functions omitted)
Stack with a Singly Linked List

- We can implement a stack with a singly linked list
- The front element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time
Parentheses Matching

Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”

- correct: ( )(( )){([ ( )])}
- incorrect: ((( )(( ))){{( ( )])}}
- incorrect: )(( )){([ ( )])}
- incorrect: ({{[ ]})
- incorrect: ( 
Parentheses Matching Algorithm

Algorithm \( \text{ParenMatch}(X, n) \):

**Input:** An array \( X \) of \( n \) tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

**Output:** \text{true} if and only if all the grouping symbols in \( X \) match

Let \( S \) be an empty stack

\[ \text{for } i = 0 \text{ to } n-1 \text{ do} \]

\[ \text{if } X[i] \text{ is an opening grouping symbol then} \]

\[ S.\text{push}(X[i]) \]

\[ \text{else if } X[i] \text{ is a closing grouping symbol then} \]

\[ \text{if } S.\text{isEmpty}() \text{ then} \]

\[ \text{return false} \{ \text{nothing to match with} \} \]

\[ \text{if } S.\text{pop}() \text{ does not match the type of } X[i] \text{ then} \]

\[ \text{return false} \{ \text{wrong type} \} \]

\[ \text{if } S.\text{isEmpty}() \text{ then} \]

\[ \text{return true} \{ \text{every symbol matched} \} \]

\[ \text{else} \]

\[ \text{return false} \{ \text{some symbols were never matched} \} \]
The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?
Queues
The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out (FIFO) scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
The Queue ADT (cont.)

- Main queue operations:
  - enqueue(o): inserts element o at the end of the queue
  - dequeue(): removes and returns the element at the front of the queue

- Auxiliary queue operations:
  - front(): returns the element at the front without removing it
  - size(): returns the number of elements stored
  - isEmpty(): returns a Boolean value indicating whether no elements are stored

- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException
## Queue Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Output</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue(5)</td>
<td>-</td>
<td>(5)</td>
</tr>
<tr>
<td>enqueue(3)</td>
<td>-</td>
<td>(5, 3)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>5</td>
<td>(3)</td>
</tr>
<tr>
<td>enqueue(7)</td>
<td>-</td>
<td>(3, 7)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>3</td>
<td>(7)</td>
</tr>
<tr>
<td>front()</td>
<td>7</td>
<td>(7)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>7</td>
<td>()</td>
</tr>
<tr>
<td>dequeue()</td>
<td>&quot;error&quot;</td>
<td>()</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>true</td>
<td>()</td>
</tr>
<tr>
<td>enqueue(9)</td>
<td>-</td>
<td>(9)</td>
</tr>
<tr>
<td>size()</td>
<td>1</td>
<td>(9)</td>
</tr>
</tbody>
</table>
Informal C++ Queue Interface

- Informal C++ interface for our Queue ADT
- Requires the definition of class EmptyQueueException
- Corresponding built-in STL class: queue

```cpp
template <typename Object>
class Queue {
public:
    int size();
    bool isEmpty();
    Object& front() throw(EmptyQueueException);
    void enqueue(Object o) throw(EmptyQueueException);
    Object dequeue() throw(EmptyQueueException);
};
```
Applications of Queues

• Direct applications
  • Waiting lists
  • Access to shared resources (e.g., printer)
  • Multiprogramming

• Indirect applications
  • Auxiliary data structure for algorithms
  • Component of other data structures
Array-based Queue

- Use an array of size $N$ in a circular fashion
- Two variables keep track of the front and rear
  - $f$ index of the front element
  - $r$ index immediately past the rear element
- Array location $r$ is kept empty

---

**normal configuration**

<table>
<thead>
<tr>
<th>$Q$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>$f$</th>
<th>$r$</th>
</tr>
</thead>
</table>

**wrapped-around configuration**

| $Q$ | 0 | 1 | 2 | $r$ | $f$ |
Queue Operations

- We use the modulo operator (remainder of division)

**Algorithm size()**

```python
return (N - f + r) mod N
```

**Algorithm isEmpty()**

```python
return (f = r)
```

![Queue Diagram](image)
Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

Algorithm $enqueue(o)$

```plaintext
if size() = N - 1 then
    throw FullQueueException
else
    Q[r] ← o
    r ← (r + 1) mod N
```

![Queue Diagram](image)
Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty.
- This exception is specified in the queue ADT.

Algorithm `dequeue()`

```plaintext
if isEmpty() then
    throw EmptyQueueException
else
    o ← Q[f]
    f ← (f + 1) mod N
    return o
```

Diagram:

```
Q: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
  0 1 2  f  r
```

```
Q: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
  0 1 2  r  f
```
Performance and Limitations
- array-based implementation of queue ADT

**Performance**
- Let \( n \) be the number of elements in the queue
- The space used is \( O(n) \)
- Each operation runs in time \( O(1) \)

**Limitations**
- The maximum size of the queue must be defined *a priori*, and cannot be changed
- Trying to push a new element into a full queue causes an implementation-specific specific exception
Queue with a Singly Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time
- NOTE: we do not have the size-limitation of the array based implementation, i.e., the queue is NEVER full.
Application: Round Robin Schedulers

We can implement a round robin scheduler using a queue, $Q$, by repeatedly performing the following steps:

1. $e = Q.dequeue()$
2. Service element $e$
3. $Q.enqueue(e)$