COMP9024: Data Structures and Algorithms

Week Four: Stacks and Queues

Hui Wu

Session 1, 2015
http://www.cse.unsw.edu.au/~cs9024

Outline

- Stacks
- Queues

Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

Example: ADT modeling a simple stock trading system
- The data stored are buy/sell orders
- The operations supported are:
  - order buy(stock, shares, price)
  - order sell(stock, shares, price)
  - void cancel(order)
- Error conditions:
  - Buy/sell a nonexistent stock
  - Cancel a nonexistent order

Stacks

The Stack ADT

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in-first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

Stack Interface in Java

- Java interface corresponding to our Stack ADT
- Requires the definition of class EmptyStackException
- Different from the built-in Java class java.util.Stack

```java
public interface Stack {
    public int size();
    public boolean isEmpty();
    public Object top() throws EmptyStackException;
    public void push(Object o);
    public Object pop() throws 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 pop and top cannot be performed if the stack is empty.
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException.

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack.
- When a method is called, the JVM pushes on the stack a frame containing:
  - Local variables and return value
  - Program counter, keeping track of the statement being executed.
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack.
- Allows for recursion.

Array-based Stack (1/2)

- 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.

```java
main() {
    int i = 5;
    foo(i);
}
foo(int j) {
    int k;
    k = j + 1;
    bar(k);
}
bar(int m) {
    ...
}
```

Array-based Stack (2/2)

- 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)
if (t = S.length - 1)
    throw FullStackException;
else
    t = t + 1;
    S[t] = o;
}
```

Performance and Limitations

- 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—Overflow.
Array-based Stack in Java

```java
public class ArrayStack implements Stack {
    // holds the stack elements
    private Object S[];
    // index to top element
    private int top = -1;
    // constructor
    public ArrayStack(int capacity) {
        S = new Object[capacity]);
    }
    public Object pop() throws EmptyStackException {
        if (isEmpty())
            throw new EmptyStackException("Empty stack: cannot pop");
        Object temp = S[top];
        // facilitate garbage collection
        S[top] = null;
        top = top - 1;
        return temp;
    }
}
```

Linked List-based Stack (1/4)

- The top of the stack is the head of the linked list.
- A instance variable keeps the current number of elements.
- push: create a new node and add it at the top of the stack.
- Pop: delete the node at the top of the stack.

```
Bottom
Rome
Seattle
New York
Top
```

Linked List-based Stack (2/4)

The node class:

```java
public class Node<E> {
    // Instance variables:
    private E element;
    private Node<E> next;
    /** Creates a node with null references to its element and next node. */
    public Node() {
        this(null, null); } 
    /** Creates a node with the given element and next node. */
    public Node(E e, Node<E> n) { element = e; next = n; }
    // Accessor methods:
    public E getElement() { return element; }
    public Node<E> getNext() { return next; }
    // Modifier methods:
    public void setElement(E newElem) { element = newElem; }
    public void setNext(Node<E> newNext) { next = newNext; }
}
```

Linked List-based Stack (3/4)

```java
public class NodeStack<E> implements Stack<E> {
    protected Node<E> top; // reference to the head node
    protected int size; // number of elements in the stack
    public NodeStack() {
        // constructs an empty stack
        top = null; size = 0; }
    public int size() { return size; }
    public boolean isEmpty() {
        if (top == null)
            return true;
        return false; }
    public void push(E elem) {
        Node<E> v = new Node<E>(elem, top);
        // create and link-in a new node
        top = v; size++; }
    public E top() throws EmptyStackException {
        if (isEmpty())
            throw new EmptyStackException("Stack is empty.");
        return top.
    }
    public E pop() throws EmptyStackException {
        if (isEmpty())
            throw new EmptyStackException("Stack is empty.");
        E temp = top.
        top = top.
        // link-out the former top node
        size--; 
        return temp; }
}
```

Linked List-based Stack (4/4)

- Each of the methods of the Stack interface takes constant time.
- Space complexity is O(n), where n is the number of elements on the stack.
- No overflow problem as in array-based stack.

Parentheses Matching

- Each "(" or "{" or "[" must be paired with a matching ")", "}", or "]".
  - correct: ( ( ( ) ) )
  - correct: ( ( ) ) ( )
  - incorrect: ( ( ) ( )
  - incorrect: ( )
  - incorrect: (

```
For fully-correct HTML, each &lt;name&gt; should pair with a matching &lt;/name&gt;.

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 true seaman, who even as a stowaway now felt that he had overdosed for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?
4. What color is the boat?
5. And what about Naomi?

We show how to use a stack as an auxiliary data structure in an algorithm.

Given an array X, the span S[i] of X[i] is the maximum number of consecutive elements X[j] immediately preceding X[i] and such that X[j] ≤ X[i].

Spans have applications in financial analysis. E.g., stock at 52-week high.
We push Main queue operations:

- Each index of the integer
- We scan the array from
- Attempting the execution of
- We pop indices from the
- The statements in
- We keep in a stack the
- 2
- We set
- boolean
- object
- Insertions are at the rear of the
- Is popped from
- Let
- 7
- enqueue
- Exceptions

We keep in a stack the indices of the elements visible when “looking back”
- We scan the array from left to right
- Let \( i \) be the current index
- We pop indices from the stack until we find index \( j \) such that \( X(j) < X(i) \)
- We set \( S(i) \leftarrow i - j \)
- We push \( i \) onto the stack

Auxiliary queue operations:

- object \( \text{front}() \): returns the element at the front without removing it
- Integer \( \text{size}() \): returns the number of elements stored
- Boolean \( \text{isEmpty}() \): indicates whether no elements are stored

Exceptions

- Attempting the execution of dequeue or front on an empty queue throws an \( \text{EmptyQueueException} \)
Applications of Queues

- Direct applications
  - Waiting lists, bureaucracy
  - 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

Queue Operations (1/3)

- We use the modulo operator (remainder of division)

  Algorithm `size()`:
  ```java
  { return (N - f + r) mod N; }
  ```

  Algorithm `isEmpty()`:
  ```java
  { return (f == r); }
  ```

Queue Operations (2/3)

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

  Algorithm `enqueue(o)`:
  ```java
  { if (size() == N - 1)
    throw FullQueueException;
  else
    Q[r] = o;
    r = (r + 1) mod N;
  }
  ```

Queue Operations (3/3)

- Operation `dequeue()` throws an exception if the queue is empty
  - This exception is specified in the queue ADT

  Algorithm `dequeue()`:
  ```java
  { if (isEmpty())
    throw EmptyQueueException
  else
    o = Q[f];
    f = (f + 1) mod N;
    return o;
  }
  ```

Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Requires the definition of class `EmptyQueueException`
- No corresponding built-in Java class

  ```java
  public interface Queue {
  public int size();
  public boolean isEmpty();
  public void enqueue(Object o);
  public Object dequeue();
  throws EmptyQueueException;
  public void isEmpty();
  throws EmptyQueueException;
  ```
Linked List-based Implementation of Queue (1/2)

- A generic singly linked list is used to implement queue.
- The front of the queue is the head of the linked list and the rear of the queue is the tail of the linked list.
- The queue class needs to maintain references to both head and tail nodes in the list.
- Each method of the singly linked list implementation of queue ADT runs in O(1) time.
- Two methods, namely dequeue() and enqueue(), are given on the next slide.

Application 1: 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)

Application 2: The Josephus Problem (1/4)

- A group of children sit in a circle passing an object, called "potato", around the circle.
- The potato begins with a starting child in the circle, and the children continue passing the potato until a leader rings a bell, at which point the child holding the potato must leave the game after handing the potato to the next child in the circle.
- After the selected child leaves, the other children close up the circle.

Application 2: The Josephus Problem (2/4)

- This process then continues until there is only child remaining, who is declared the winner.
- If the leader always uses the strategy of ringing the bell after the potato has been passed k times, for some fixed k, determining the winner for a given list of children is known as the josephus problem.
/** Build a queue from an array of objects */
public static <E> Queue<E> buildQueue(E a[])
Queue<E> Q = new NodeQueue<E>);
for (int i=0; i<a.length; ++i) Q.enqueue(a[i]);
return Q;
}

/** Tester method */
public static void main(String[] args) {
String[] a1 = {"Alice", "Bob", "Cindy", "Doug", "Ed", "Fred"};
String[] a2 = {"Gene", "Hope", "Irene", "Jack", "Kim", "Lance"};
String[] a3 = {"Mike", "Roberto"};
System.out.println(First winner is " + Josephus(buildQueue(a1), 3));
System.out.println(Second winner is " + Josephus(buildQueue(a2),
10)); System.out.println(Third winner is " + Josephus(buildQueue(a3),
7));
})

References
1. Chapter 5, Data Structures and Algorithms by Goodrich and Tamassia.