COMP9024: Data Structures and Algorithms

Week Four: Stacks and Queues

Hui Wu

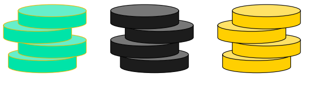
Session 1, 2015

http://www.cse.unsw.edu.au/~cs9024

Outline

- Stacks
- Queues

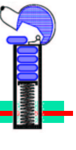
Stacks



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

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

```
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

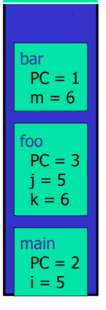
Applications of Stacks

- 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**

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



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

Algorithm *size*()
{ **return** *t* + 1; }

Algorithm *pop*()
{ **if** (*isEmpty*())
  throw *EmptyStackException;*
 else
  t = *t* − 1;
 return *S*[*t* + 1];
}

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.

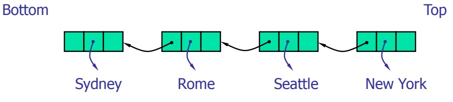
Array-based Stack in Java

```
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];
      // facilitates 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.



Linked List-based Stack (2/4)

The node class:

```
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)

```
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.getElement(); }
  public E pop() throws EmptyStackException {
      if (isEmpty()) throw new EmptyStackException("Stack is empty.");
      E temp = top.getElement();
      top = top.getNext();   // 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 "[" must be paired with a matching ")", "}", or "["
  - correct: ( )(( )){([( )])}
  - correct: (( )( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Parentheses Matching Algorithm

```
Algorithm 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: true if and only if all the grouping symbols in X match
  Let S be an empty stack;
  for ( i=0; i < n; i++)
      if ( X[i] is an opening grouping symbol )
           S.push(X[i]);
      else if ( X[i] is a closing grouping symbol )
           {
             if ( S.isEmpty() )
                 return false;   // nothing to match with
             if ( S.pop() does not match the type of X[i] )
                 return false;   // wrong type
           }
  if ( S.isEmpty() )
       return true;   // every symbol matched
  else
       return false;  // some symbols were never matched
}
```


HTML Tag Matching

- For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> 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. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

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?

Tag Matching Algorithm (1/3)

- Is similar to parentheses matching:

```
import java.io.*;
import java.util.Scanner;
import net.datastructures.*;
/** Simplified test of matching tags in an HTML document. */
public class HTML {
/** Strip the first and last characters off a <tag> string. */
  public static String stripEnds(String t) {
     if (t.length() <= 2) return null; // this is a degenerate tag
     return t.substring(1,t.length()-1); }
/** Test if a stripped tag string is empty or a true opening tag. */
  public static boolean isOpeningTag(String tag) {
     return (tag.length() == 0) || (tag.charAt(0) != '/'); }
```


Tag Matching Algorithm (2/3)

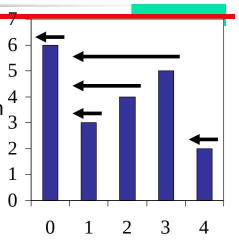
```
/** Test if stripped tag1 matches closing tag2 (first character is '/'). */
public static boolean areMatchingTags(String tag1, String tag2) {
   return tag1.equals(tag2.substring(1)); // test against name after '/' }

 /** Test if every opening tag has a matching closing tag. */
public static boolean isHTMLMatched(String[] tag) {
   Stack<String> S = new NodeStack<String>(); // Stack for matching
  tags
   for (int i = 0; (i < tag.length) && (tag[i] != null); i++)
    {
     if (isOpeningTag(tag[i])) S.push(tag[i]); // opening tag; push it on the
  stack
     else { if (S.isEmpty()) return false; // nothing to match
            if (!areMatchingTags(S.pop(), tag[i])) return false; // wrong
  match
          }
    }
    if (S.isEmpty()) return true; // we matched everything
       return false; // we have some tags that never were matched }
```


Tag Matching Algorithm (3/3)

```
public final static int CAPACITY = 1000; // Tag array size
/* Parse an HTML document into an array of html tags */
 public static String[] parseHTML(Scanner s) {
    String[] tag = new String[CAPACITY]; // our tag array (initially all null)
    int count = 0; // tag counter
    String token; // token returned by the scanner s
    while (s.hasNextLine())
       { while ((token = s.findInLine("<[^>]*>")) != null) // find the next tag
            tag[count++] = stripEnds(token); // strip the ends off this tag
         s.nextLine(); // go to the next line
       }
     return tag; // our array of (stripped) tags }

  public static void main(String[] args) throws IOException { // tester
     if (isHTMLMatched(parseHTML(new Scanner(System.in))))
        System.out.println("The input file is a matched HTML document.");
     else
        System.out.println("The input file is not a matched HTML
  document."); } }
```


Computing Spans (not in book)

- We show how to use a stack as an auxiliary data structure in an algorithm
- Given an 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 $X[j] \le X[i]$
- Spans have applications to financial analysis
  - E.g., stock at 52-week high



X	6	3	4	5	2
S	1	1	2	3	1

Quadratic Algorithm

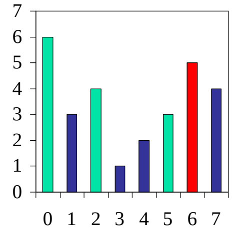
```
Algorithm spans1(X, n)
{
  Input array X of n integers
  Output array S of spans of X               #
  S = new array of n integers;               n
  for ( i = 0; i < n; i++)                   n
    { s = 1;                                 n
      while ( s ≤ i ∧ X[i – s] ≤ X[i] )      1 + 2 + …+ (n – 1)
          s = s + 1;                         1 + 2 + …+ (n – 1)
      S[i] = s;                              n
    }
  return S;                                  1
}
```

- Algorithm ***spans1*** runs in $O(n^2)$ time

25

Computing Spans with a Stack

- 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 $X[i] < X[j]$
  - We set $S[i] \leftarrow i - j$
  - We push i onto the stack



26

Linear Algorithm

- Each index of the array
  - Is pushed into the stack exactly one
  - Is popped from the stack at most once
- The statements in the while-loop are executed at most n times
- Algorithm ***spans2*** runs in $O(n)$ time

```
Algorithm spans2(X, n)                       #
{ S = new array of n integers;               n
  A = new empty stack;                       1
  for ( i = 0; i < n, i ++ )                 n
  {
    while (¬A.isEmpty() ∧
             X[A.top()] ≤ X[i] )             n
             A.pop();                        n
    if ( A.isEmpty() )                       n
        S[i] = i + 1;                        n
    else
        S[i] = i – A.top();                  n
    A.push(i);                               n
  }
  return S;                                  1
}
```

27

Queues

28

The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - object front(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

29

Queue Example

Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
front()	*7*	(7)
dequeue()	*7*	()
dequeue()	*"error"*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

30

Applications of Queues

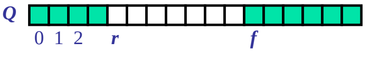
- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

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

normal configuration

wrapped-around configuration

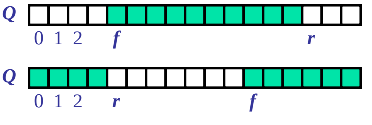


Queue Operations (1/3)

- We use the modulo operator (remainder of division)

Algorithm *size*()
{ **return** $(N - f + r) \bmod N$; }

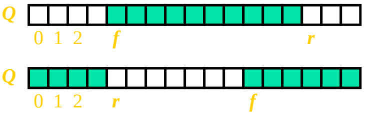
Algorithm *isEmpty*()
{ **return** $(f = r)$; }



Queue Operations (2/3)

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

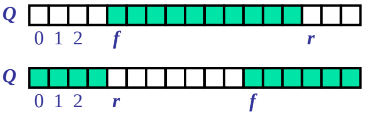
Algorithm *enqueue*(o)
{ **if** (*size*() = $N - 1$)
  throw ***FullQueueException;***
 else
  { $Q[r] = o$;
   $r = (r + 1) \bmod N$;
  }
}



Queue Operations (3/3)

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

Algorithm *dequeue*()
{ **if** (*isEmpty*())
  throw ***EmptyQueueException***
 else
  { $o = Q[f]$;
   $f = (f + 1) \bmod N$;
   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

```
public interface Queue {
  public int size();
  public boolean isEmpty();
  public Object front()
      throws EmptyQueueException;
  public void enqueue(Object o);
  public Object dequeue()
      throws EmptyQueueException;
}
```

Linked List-based Implementation of Queue (1/2)

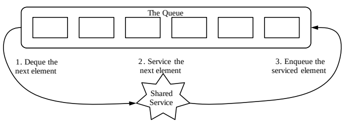
- A generic singly linked list is used to implement queue.
- The front of the queue is the head of the linked list and the rear of the queue is the tail of the linked list.
- The queue class needs to maintain references to both head and tail nodes in the list.
- Each method of the singly linked list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

Linked List-based Implementation of Queue (2/2)

```
public void enqueue(E elem) {
   Node<E> node = new Node<E>();
   node.setElement(elem);
   node.setNext(null); // node will be new tail node
   if (size == 0) head = node; // special case of a previously empty queue
   else tail.setNext(node); // add node at the tail of the list
   tail = node; // update the reference to the tail node
   size++;  }

public E dequeue() throws EmptyQueueException {
   if (size == 0) throw new EmptyQueueException("Queue is empty.");
   E tmp = head.getElement();
   head = head.getNext();
   size--;
   if (size == 0) tail = null; // the queue is now empty
   return tmp; }
```


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.

Application 2: The Josephus Problem (3/4)

```
import net.datastructures.*;
public class Josephus { /** Solution of the Josephus problem using a queue.
  */ public static <E> E Josephus(Queue<E> Q, int k) {
    if (Q.isEmpty()) return null;
    while (Q.size() > 1) {
      System.out.println(" Queue: " + Q + " k = " + k);
      for (int i=0; i < k; i++)
        Q.enqueue(Q.dequeue()); // move the front element to the end
      E e = Q.dequeue(); // remove the front element from the collection
      System.out.println(" " + e + " is out"); }
    return Q.dequeue(); // the winner }
```

Application 2: The Josephus Problem (4/4)

```
/** 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.