Tuesday, March 13 2001

Overview

- Containers
  - Inspectable Containers
- Abstract Data Types
  - Trees, Binary Trees
- Pseudo-Code
- Array Based Implementations
  - Stack
  - Queue
- Loop Invariants (Time Permitting)

Containers (1)

- Definition
  - A data structure that stores and organizes a collection of objects called the elements of the container.
  - Provides access to them through the methods of the abstract data type.
  - Examples include: stacks, queues, dequesues, vectors, lists, sequences etc.
- Four main Categories for Methods of a Container
  - Query Methods: Return info. on the container or specific elements.
  - Accessor Methods: Return elements or positions of the container.
  - Update Methods: Change the container by adding or removing elements or altering the relation between elements.
  - Constructor Methods: Generate an instance of the container.

Inspectable Containers

- Containers that do not provide update methods.
- Support only read-only access and cannot be modified.
- Protect their elements from erroneous or malicious update attempts by other objects.
- Cannot be used when their elements are subject to updates during the life of their container.
- Using inheritance we can get protection of inspectable containers while still getting flexibility of update methods.

Inspectable Vector

- InspectableVector that contains only the query methods and the accessor method.
- Redefine a Vector as an ADT that inspects the inspectableVector ADT and adds the update methods.
- Revised vector is equivalent to the original one.
- But now we can reference an instance of the vector with a variable of type InspectableVector thus allowing only the query and accessor methods.
- Can also restructure the List and Sequence ADTs by introducing inspectable versions of them.
Trees: Terminology and Basic Properties

- Tree Abstract Data Type
  - Stores elements hierarchically
  - A set of nodes storing elements in a parent-child relationship.
  - Top element \( r \) is called the root.
  - Each node \( v \) of tree \( T \) except for \( r \) has a parent node \( u \).
  - Tree cannot be empty.
    - Will always have at least the root node

- Definitions
  - Child
    If node \( u \) is the parent of node \( v \), then \( v \) is a child of \( u \).
  - Siblings
    Two nodes which have the same parent.
  - External Node:
    A node which has no children.
    - Also known as a leaf.
  - Internal Node:
    A node which has at least one child.

Trees: Examples

- organization structure of a corporation
- table of contents of a book

Trees: Another Example

- Unix or DOS/Windows file system
  - Internal nodes are associated with directories.
  - External nodes are associated with regular files.
  - In Unix, root of the tree is called root directory.
Trees: Terminology

- A is the root node.
- B is the parent of D and E.
- C is the sibling of B
- D and E are the children of B
- D, E, F, G, I are external nodes, or leaves
- A, B, C, H are internal nodes
- The depth (level) of E is 2
- The height of the tree is 3
- The degree of node B is 2

Property: (# edges) = (#nodes) − 1

Trees: ADT (1)

- Tree ADT stores elements at positions
  - Defined relative to neighboring positions.
- Positions in a tree are its nodes
  - Neighboring positions satisfy the parent-child relationships that define a valid tree.
- Position and Node are used interchangeably for trees!
- Methods
  - A position object for a tree supports the method:
    - element()
      Return the object at this position
      Input: None, Output: Object
    - Accessor methods

Trees: ADT (2)

root()
Return the root of the tree.
Input: None, Output: Position

parent(v)
Return the parent of node v; An error occurs if v is the root.
Input: Position, Output: Position

children(v)
Return the Iterator of the children of node v.
Input: Position, Output: Iterator of Position

Query Methods

isInternal(v)
Test whether node v is internal.
Input: Position, Output: Boolean

Trees: ADT (3)

isExternal(v)
Test whether node v is external.
Input: Position, Output: Boolean

isRoot(v)
Test whether node v is the root of the tree.
Input: Position, Output: Boolean

Generic Methods
- Not necessarily related to a tree structure
  - size()
    Return the number of nodes in the tree.
    Input: None, Output: Integer
  - elements()
    Return an iterator of all elements stored in the nodes of the tree.
    Input: None, Output: Iterator of Objects
Trees: ADT (4)

positions()
Return the iterator of all nodes in the tree.
Input: None, Output: Iterator of positions

swapElements(v, w)
Swap the elements stored at nodes v and w.
Input: Two positions, Output: None

replaceElement(v, e)
Replace with e and return the element stored at node v.
Input: Position and an object, Output: Object

Trees: Java Interface (1)

public interface InspectablePositionalContainer extends InspectableContainer {
    // accessor methods
    /** return the positions in the container */
    public PositionIterator positions();
}

public interface PositionalContainer extends InspectablePositionalContainer {
    // update methods
    /** swap the elements at two nodes */
    public void swapElements(Position v, Position w);
    /** replace with and return the element at a node */
    public Object replaceElement(Position v, Object e);
}

Trees: Java Interface (2)

public interface InspectableTree extends InspectablePositionalContainer {
    // accessor methods
    /** return the root of the tree */
    public Position root();
    /** return the parent of a node */
    public Position parent(Position v);
    /** return the children of a node */
    public PositionIterator children(Position v);
    // query methods
    /** test whether a node is internal */
    public boolean isInternal(Position v);
    /** test whether a node is external */
    public boolean isExternal(Position v);
    /** test whether a node is the root of the tree */
    public boolean isRoot(Position v);
}

Trees: Java Interface (3)

public interface Tree extends InspectableTree,
    PositionalContainer {

    Additional update methods may be added depending on the application.
    • Not included in the interface!
    • Tree interface is simply the combination of two other interfaces.
Trees: Binary Trees (1)

- Binary Tree
  - Ordered tree.
  - Each node has a maximum of two children.

- Definitions:
  - Proper Binary tree:
    Each node has either zero or two children. Every internal node has exactly two children.
  - Left or Right Child
    Each child of a node is labeled as either the left or right child. Left child comes before the right child.
  - Left and Right Subtree
    The subtree rooted at a left or right child of an internal node v.

Trees: Binary Trees (2)

- Recursive Definition:
  - A binary tree is either
    - an external node (leaf), or
    - an internal node (the root) and two binary trees
      (left subtree and right subtree)

Trees: Binary Tree ADT

- Accessor Methods:
  - leftChild(Position v)
    Return left child of node v; Error occurs if v is an external node.
    \[\text{Input: Position, Output: Position}\]
  - rightChild(Position v)
    Return right child of node v; Error occurs if v is an external node.
    \[\text{Input: Position, Output: Position}\]
  - sibling(Position v)
    Return sibling of node v; Error occurs if v is the root.
    \[\text{Input: Position, Output: Position}\]

- Specialized update methods are not defined here!
- May have additional error conditions if the trees are not proper!

Trees: Binary Tree Java Interface

```java
public interfaceInspectableBinaryTree extendsInspectableTree{

  // accessor methods
  /** return the left child of a node */
  public Position leftChild(Position v);

  /** return the right child of a node */
  public Position rightChild(Position v);

  /** return the sibling of a node */
  public Position sibling(Position v);

public interface BinaryTree extendsInspectableBinaryTree, PositionalContainer{
```

COSC 2011 Section N
Winter 2001 03/13/2001
Pseudo-Code (1)

- A description of an algorithm that is more structured than usual prose but less formal than a programming language.
- Example: finding the maximum element of an array:

```
Algorithm arrayMax(A, n):
    Input: An array A storing n integers.
    Output: The maximum element in A.
    currentMax ← A[0]
    for i ← 1 to n - 1 do
        if currentMax < A[i] then
            currentMax ← A[i]
    return currentMax
```

- More compact than Java code
- Easier to read
- High level description permits high level analysis of a data structure or algorithm

Pseudo-Code (2)

- What is Pseudo-Code?
  - A mixture of natural language and high level programming concepts that describe the main ideas behind a generic implementation of a data structure or algorithm
  - Expressions: Use standard mathematical symbols to describe numeric and boolean expressions.
    - Use `=` for assignment (`==` in Java)
    - Use `==` for equality relationships (`==` in Java)
  - Method Declarations:
    - `Algorithm(param1, param2 ... )`
  - Programming Constructs:
    - Decision Structures
      - if ... then ... [else]
    - While Loops
      - while ... do
    - For Loops
      - for ... do
    - Array Indexing: A[I]

Pseudo-Code (3)

- What is Pseudo-Code?
  - Methods:
    - Calls: object.method(args)
    - Return: return value
  - When writing Pseudo Code, Keep in Mind:
    - Writing for a human reader, not a computer!
      - Want high level ideas not low level implementation details!
      - Don’t forget important steps!
    - Skill which is improved with practice!

Array Based Implementations: Stack (1)

- Specify a maximum size N for the Stack, e.g. N = 1000
- Stack consists of an N element Array S and an integer variable t, the index of the top element in the Stack.
  - Array indices start with 0 so initialize t to −1.

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

Algorithm isEmpty():
    return (t < 0)

Algorithm top():
    if isEmpty() then
        throw a StackEmptyException
    return S[t]
```

...
Array Based Implementations: Stack (2)

Algorithm push(s):
if size() = N then
    throw StackOverflowException
i ← i + 1
S(i) ← e

Algorithm pop(s):
if isEmpty() then
    throw StackEmptyException
i ← S(i)
S(i) ← null
i ← i - 1
return e

Each of the above methods runs in O(1) time.
Very simple and efficient
Upper bound to the stack size may be:
Too small
May be too large and actually waste memory

Note the exceptions!

Array Based Implementations: Stack (3)

Source Code

```java
public class ArrayStack implements Stack {
    // Implementation of the Stack interface
    // using an array.

    public static final int CAPACITY = 1024; // default
    // capacity of the stack
    private int capacity; // maximum capacity of the
    // stack
    private Object S[]; // holds the elements of
    // the stack
    private int top = -1; // the top element of the
    // stack.

    public ArrayStack() { // Initialize the Stack
        this(CAPACITY); // with default capacity
    }

    public ArrayStack(int cap) { // Initialize the
    // stack with given capacity
        capacity = cap;
        S = new Object[capacity];
    }
}
```

Array Based Implementations: Stack (4)

Source Code (continued)

```java
public int size() { // Return the current stack size
    return (top + 1); }

public boolean isEmpty() { // return true iff
    return (top < 0); }

public void push(Object obj)
    throws StackFullException { // push a new
    // element on the stack
    if (size() == capacity) {
        throw new StackFullException("Stack overflow.");
    }
    S[++top] = obj;
}

public Object top() { // Return the top stack
    // element
    throws StackEmptyException { //
    if (isEmpty()) {
        throw new StackEmptyException("Stack is empty.");
    }
    return S[top];
}
```

Array Based Implementations: Stack (5)

Source Code (continued)

```java
public Object pop() // Pop off the stack element
    throws StackEmptyException {
    Object elem;
    if (!isEmpty()) {
        throw new StackEmptyException("Stack is Empty.");
    }
    elem = S[top];
    S[top] = null; // Dereference S[top] and
    // decrement top
    return elem;
}
```
Array Based Implementations: Stack (6)

- Casting with a Generic Stack
  - Can store generic objects in the stack, each belonging to an arbitrary class.
  - Elements that are stored in it are viewed as instances of the Java Object class!
  - No trouble adding elements to the stack since every class in Java inherits from Object.
  - When removing an element from a Stack we get back a reference of type Object no matter what type of class the object may be.
  - Must cast the object to the specific class.

  ```java
  public static Integer[] reverse(Integer[] a) {
    ArrayStack S = new ArrayStack(a.length);
    Integer[] b = new Integer(a.length);
    for (int i = 0; i < a.length; i++) {
      S.push(a[i]);
    }
    for (int i = 0; i < a.length; i++) {
      b[i] = (Integer) (S.pop());
    }
  }
  ```

Array Based Implementations: Queue (1)

- Array Q to with a capacity N
  - How do we keep track of the front and rear of the Q?
    - Can use same approach as the Stack – let Q[0] be the front of the queue and let the queue grow from there.
      - Not efficient! Need to move all the elements forward one array cell after a dequeue operation.
    - To avoid moving the objects:
      - Define two integer variables f and r with the following properties:
        - f, index of the front element.
        - r, index of the rear element.
      - Initially assign f = r = 0 to indicate queue is empty.
      - When we remove an element from front of the queue, increment f to index the next cell.
      - When we add an element, increment r to index the next available cell
      - Allows for O(1) time enqueue and dequeue

Array Based Implementations: Queue (2)

- Problem with this approach!
  - What if we enqueue and dequeue a single element N different times?
    - Then f = r = N!
    - If we tried to insert another element, we get an ArrayIndexOutOfBoundsException!
    - Still plenty of room in the array!
  - Use a “Circular Array”
    - f and r indices wrap around the end of the queue.
    - Array goes from Q[0] – Q[N-1] and then immediately back to Q[0] again.

  ![Circular Array Diagram]

  * "wrapped around" configuration

Array Based Implementations: Queue (3)

- Each time we increment f or r, compute the increment as:
  - f = (f + 1) mod N
  - r = (r + 1) mod N
- Now have O(1) time enqueue and dequeue
- What if we enqueue N objects without dequeueing any of them?
  - Then f = r, same condition when queue is empty and cannot tell the difference between a full and empty queue.
  - Solution?
    - Insists that Q can never hold more than N-1 elements.
  - Computing Size of the Queue:
    - (N – f + r) mod N
  - Gives correct size both in normal configuration (f <= r) and wrapped around configuration (r < f)
Array Based Implementations: Queue (4)

- Pseudo-Code:

  Algorithm size()
  \hspace{0.5cm} return \( (N - f + r) \mod N \)

  Algorithm isEmpty()
  \hspace{0.5cm} return \( (f = r) \)

  Algorithm front()
  \hspace{0.5cm} if isEmpty then
  \hspace{1.0cm} throw a QueueEmptyException
  \hspace{0.5cm} return \( Q[f] \)

  Algorithm dequeue()
  \hspace{0.5cm} if isEmpty then
  \hspace{1.0cm} throw a QueueEmptyException
  \hspace{0.5cm} temp \( \leftarrow Q[f] \)
  \hspace{0.5cm} Q[f] \leftarrow \text{null}
  \hspace{0.5cm} f \leftarrow \( (f + 1) \mod N \)
  \hspace{0.5cm} return temp

Array Based Implementations: Queue (5)

- Pseudo-Code (continued):

  Algorithm enqueue(o)
  \hspace{0.5cm} if size() = \( N - 1 \) then
  \hspace{1.0cm} throw a QueueFullException
  \hspace{0.5cm} Q[r] \leftarrow o
  \hspace{0.5cm} r \leftarrow \( (r + 1) \mod N \)

- Disadvantages of Array based Implementations:
  - Set the capacity of the array
  - In a real application we may need more or less capacity than this.

- If we have a good estimate of the number of elements in the stack or array then array based implementations are simple and very efficient!

Stacks and Queue with Deques

<table>
<thead>
<tr>
<th>Stack Methods</th>
<th>Deque Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td>size()</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>isEmpty()</td>
</tr>
<tr>
<td>top()</td>
<td>last()</td>
</tr>
<tr>
<td>push(e)</td>
<td>insertLast(e)</td>
</tr>
<tr>
<td>pop()</td>
<td>removeLast()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Queue Method</th>
<th>Dequeue Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td>size()</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>isEmpty()</td>
</tr>
<tr>
<td>front()</td>
<td>first()</td>
</tr>
<tr>
<td>enqueue()</td>
<td>insertLast()</td>
</tr>
<tr>
<td>dequeue()</td>
<td>removeFirst()</td>
</tr>
</tbody>
</table>

Bubble Sort Algorithm

- Bubble-Sort Algorithm
  - Sorts a sequence of elements in a sequence in non-decreasing order.
  - Performs a series of passes over the sequence
  - In each pass, elements are scanned in increasing rank, from rank 0 to the end of the sequence.
  - At each position in each pass, an element is compared with its neighbour
  - If in wrong order, elements are swapped
  - Total of n passes are performed
  - In first pass, when largest element is swapped, it will be swapped until it reaches the end of the sequence.
  - In the second pass, the seconds largest element is found etc.

- Running Time:
  - \( O(n^2) \)