About This Lecture

• In this lecture we will learn about an implementation of a linear data called a Queue.

Outline

• Queue in the Structure Container Hierarchy
• Queue Interface
• Maze - An example of using a Queue
• List Implementation of Queue
• Vector Implementation of Queue

Queue

• An abstract data structure with the following properties
  – A list of data items
  – First in, first out operation

Abstract Data Type (ADT)

• A mathematic (or programming) model that specifies
  – The type of data stored
  – The operations supported on the data
• An ADT needs not to specify how to implement it

ADT (continued)

• Java interface is used to specify an ADT
• Java class is used to implement an ADT
ADT (continued)

- How are ADTs supported in other languages?
  - C++ is an object-oriented language and thus supports ADT with classes
  - C is a procedural language but also supports this concept/practical tool with struct

Structural Interface - Queue

```java
public interface Queue extends Linear {
    public void enqueue(Object object); // post: the object is added at the tail. It will be removed after all elements in front of it.
    public Object dequeue(); // pre: queue is not empty
    public Object peek(); // pre: queue is not empty
    public Object get(int i); // post: returns the element at the front of the queue, but does not remove it
}
```

Queue Example - Maze Algorithm

- Like Stacks, Queues can also be used to store unsearched paths.
- Repeat as long as the current square is not null and is not the finish square:
  - “Visit” the square and mark it as visited.
  - Enqueue one square on the queue for each unvisited legal move from the current square.
  - Dequeue the queue into the current square or bind the current square to null if the queue is empty.
- If the current square is the goal we are successful, otherwise there is no solution

Queue Example - Maze Trace 1

```
```

Queue Example - Searching

- The algorithm seems the same so what is the difference between using a Stack and a Queue?
- When a Stack is used, the search goes as deep along a single path as possible, before trying another path.
- When a Queue is used, the search expands a frontier of nodes equidistant from the start.

```
Stack: 5 1 2 4 10 7 8
Queue: 5 1 2 4 10 7 8 1 3 5 6 2 5 8 4 7 10
```

```
Success!
```
Maze Program

- Same program as in the Stack lecture, except replace the line:
  todo = new StackList(); // A class that implements Stack
by:
  todo = new QueueList(); // A class that implements Queue

List-Based Queues

- Which linked list should we use for the queue?
  - doubly-linked list

- What are time complexities for all three basic methods?
  - enqueue
  - dequeue
  - peek

List-Based Queues

- We can implement a Queue using a DoublyLinkedList or a CircularLinkedList.
- We do not use a SinglyLinkedList since operations on the tail take time proportional to the length of the list.
- As the Queue grows and shrinks, so does the list.
- The time for adding or removing a single element is always the same constant time.
- There are two extra link spaces required for each element (using DoublyLinkedList).
- The implementation is simple, so few mistakes can be made.

QueueList - State and Constructors

```java
public class QueueList implements Queue {
   /*
      An implementation of the Queue Interface that uses a
      DoublyLinkedList to store the Stack elements.
   */

   // Instance Variables //
   protected List data;

   public QueueList() {
      // post: initialize the Stack to be empty
      this.data = new DoublyLinkedList(); // or CircularLinkedList
   }
}
```

QueueList - Linear Interface

```java
public void enqueue(Object anObject) {
   // post: the object is added at the tail. It will be
   // removed after all elements in front of it.
   this.data.addToTail(anObject);
}

public Object dequeue() {
   // pre: queue is not empty
   // post: the head of the queue is removed and returned
   return this.data.removeFromHead();
}
```

QueueList - Linear Interface continued

```java
public Object peek() {
   // pre: queue is not empty
   // post: returns the element at the head of the queue,
   // but does not remove it
   return this.data.peek();
}
```
Vector-Based Queues
- We can implement a Queue using a single Vector.
- However, this implementation is not time efficient.
- Adding an element to the Queue results in adding an element to the end of the Vector which takes constant time (except when the Vector must grow).
- However, removing an element from the Queue results in removing the zeroth element of the Vector which takes time proportional to the length of the Vector (since all elements after the zeroth must be moved to the left).
- Since this implementation has such poor performance, we will not use it.

Array-Based Queues
- We can implement a Queue using a single Array if we know in advance the maximum size of the Queue.
- How to achieve efficient access operations.

Array-Based Queues
- Instead, we maintain two indexes, the head and tail index and let them “slide” towards the end of the Array and then wrap around.

Array-Based Queues - Empty and Full
- We leave one empty entry in the Queue.
- The condition for an empty Queue is: head == tail.
- The condition for a full Queue is: tail is one “behind” head.