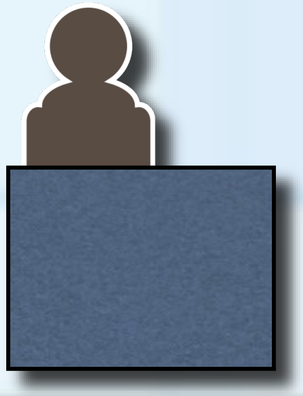
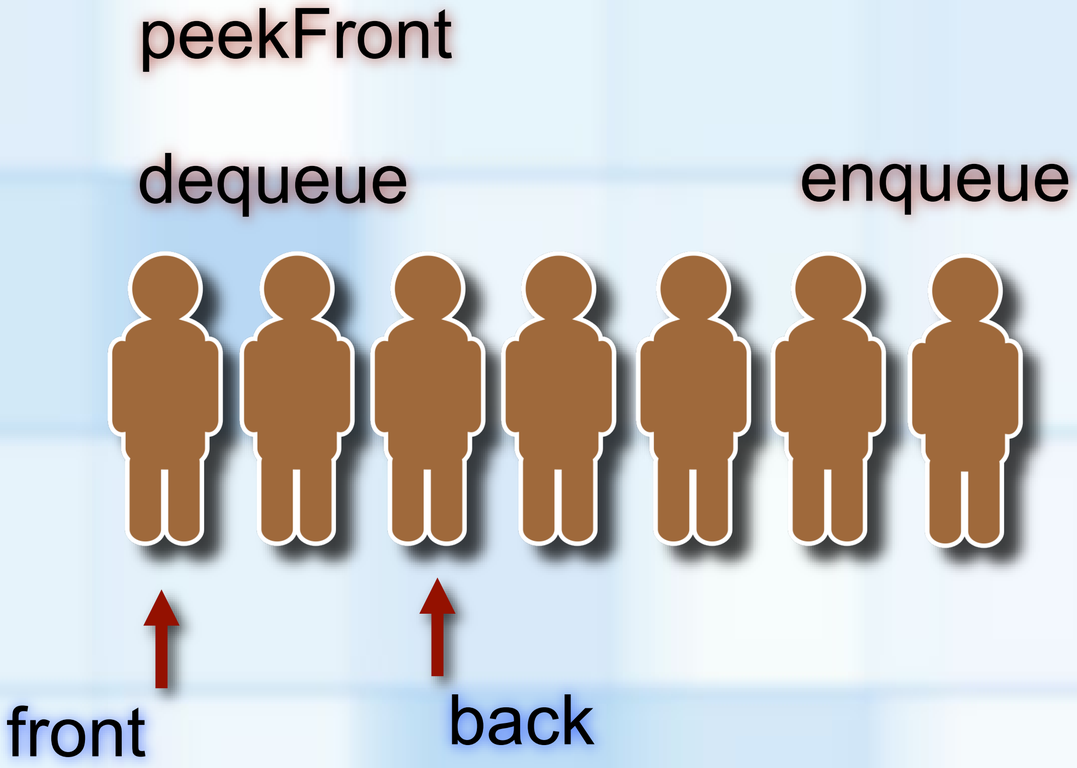
The ADT Queue

The ADT Queue

- **Organizes entries in the order added**
- **First-In, First-out (FIFO) Behavior**
  - Entries are added at the back
  - Removals occur at the front
- **Access restricted to entries**
  - Only front entry is accessible
  - Earliest entry added to queue



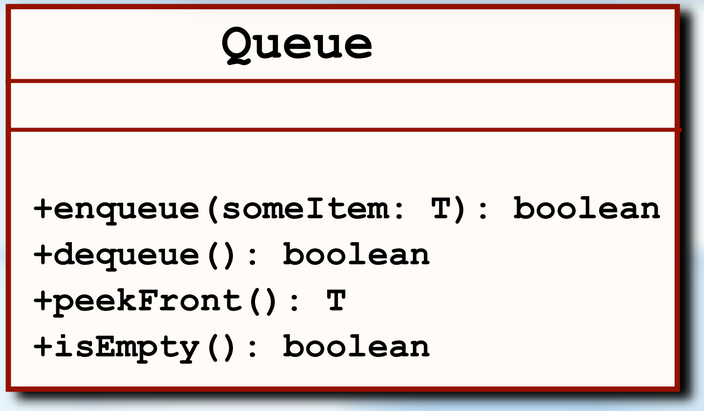


Pearson

The ADT Queue

- **Collection of objects in chronological order and having the same data type**
- **ADT Queue operations**
  - Add a new item to the queue
    - ***boolean enqueue(ItemType someItem)***
  - Remove item that was added most recently
    - ***boolean dequeue()***
  - Retrieve item that was added most recently
    - ***ItemType peekFront()***
  - Determine whether a queue is empty
    - ***boolean isEmpty()***

```
template<class ItemType>
class QueueInterface
{
public:
  /** Adds a new entry to the back of this queue. */
  virtual bool enqueue(const ItemType& someItem) = 0;

  /** Removes the front of this queue. */
  virtual bool dequeue() = 0;

  /** Returns the front of this queue. */
  virtual ItemType peekFront() const = 0;

 /** Sees whether this queue is empty. */
  virtual bool isEmpty() const = 0;
}; // end QueueInterface
```



Pearson

Using The ADT Queue

```
template<class ItemType>
class QueueInterface
{
public:
  virtual bool enqueue(const ItemType& newEntry) = 0;
  virtual bool dequeue() = 0;
  virtual ItemType peekFront() const = 0;
  virtual bool isEmpty() const = 0;
}; // end QueueInterface
```

```
Tam is at the front of the queue.
Tam is removed from the queue.
Jess is at the front of the queue.
Jess is removed from the queue.
```


```
QueueInterface<string>* stringQueue = new OurQueue<string>();
stringQueue->enqueue("Tam");
stringQueue->enqueue("Jess");
stringQueue->enqueue("Yuki");
stringQueue->enqueue("Jane");
stringQueue->enqueue("Armin");

string name = stringQueue->peekFront();
cout << name << " is at the front of the queue." << endl;

stringQueue->dequeue();
cout << name << " is removed from the queue." << endl;

name = stringQueue->peekFront();
cout << name << " is at the front of the queue." << endl;

stringQueue->dequeue();
cout << name << " is removed from the queue." << endl;
```

Pearson

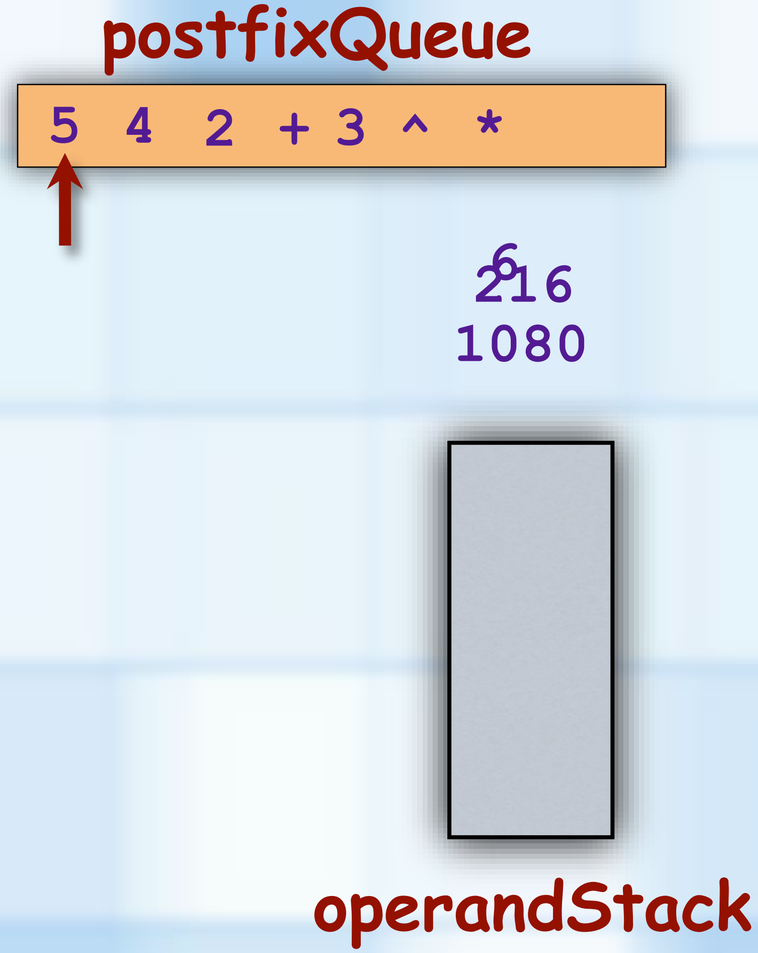
Using the ADT Queue

Evaluating Postfix Expressions

- Dequeue characters from the postfixQueue

Expression

- When an ***operand*** is entered,
  - `push` it onto the **operandStack**
- When an ***operator*** is entered,
  - apply it to the top two ***operands*** of the **operandStack**
    - `pop` the ***operands*** from the operandStack
  - `push` the result of the operation onto the **operandStack**



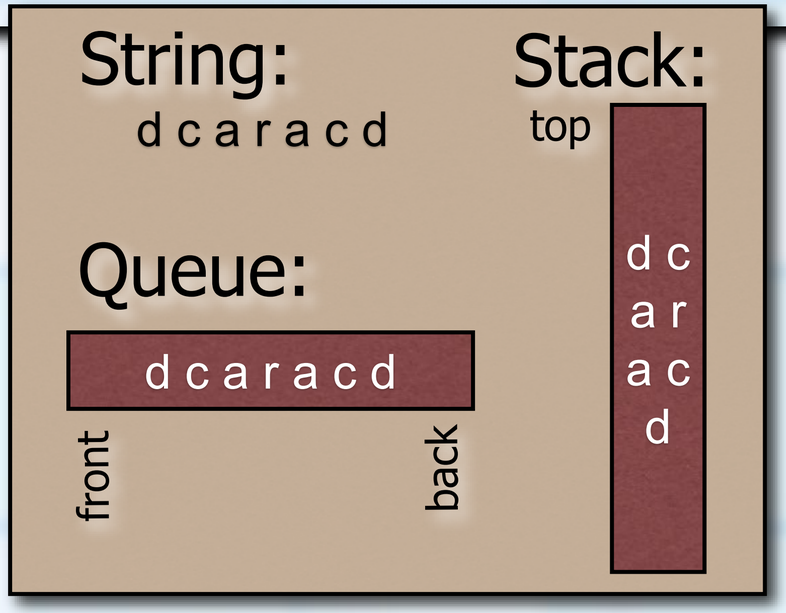
Evaluating Infix Expressions

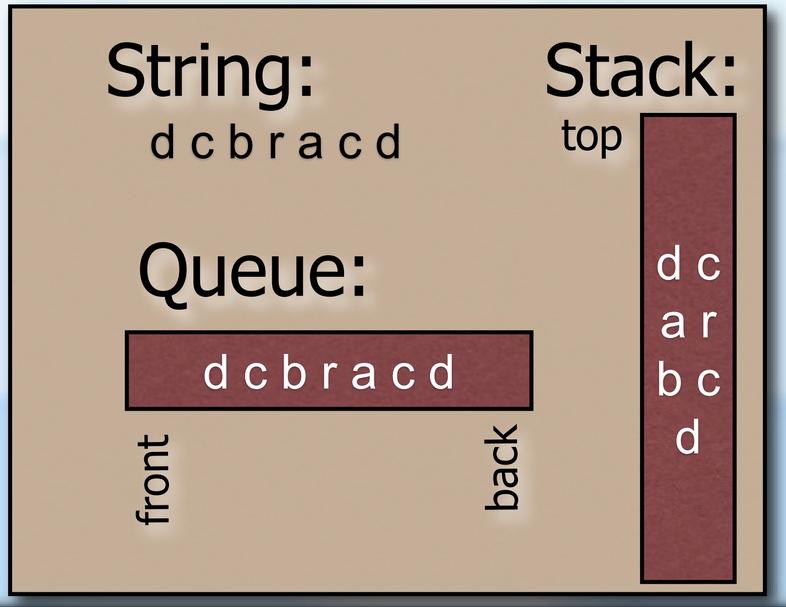
- **To evaluate an infix expression**
  - Convert the infix expression to postfix form
  - Evaluate the postfix expression

```
Algorithm evaluatePostfix(postfix)
// Evaluates a postfix expression.
valueStack = a new empty stack

while (!postfixQueue.isEmpty())
{
 {
    nextCharacter = postfixQueue.peekFront()
    postfixQueue.dequeue()
    switch (nextCharacter)
    {
       case variable:
          valueStack.push(value of the variable nextCharacter)
          break

       case '+' : case '-' : case '*' : case '/' : case '^' :
            operandTwo = valueStack.peek()
           valueStack.pop()
           operandOne = valueStack.peek()
           valueStack.pop()
           result = the result of the operation
                          in nextCharacter and its operands operandOne and operandTwo
           valueStack.push(result)
           break

        default: break
     } // end switch
} // end while

return valueStack.peek()
```

PALINDROMES

- **Recognizing Palindromes**
  - A palindrome
    - A string of characters that reads the same from left to right as its does from right to left
  - To recognize a palindrome, use a queue in conjunction with a stack
    - A stack reverses the order of occurrences
    - A queue preserves the order of occurrences
- **Non-recursive recognition algorithm**
  - Traverse the character string from left to right,
    - Insert each character into both a queue and a stack
    - Compare the characters at the front of the queue and the top of the stack





Queue Applications

- **Simulation**
  - A technique for modeling the behavior of both natural and human-made systems
  - Goal
    - Generate statistics that summarize the performance of an existing system
    - Predict the performance of a proposed system
  - Example
    - A simulation of the behavior of a bank
- **A time-driven simulation**
  - Simulated time advances by one time unit
  - The time of an event is determined randomly and compared with the simulated time
- **An event-driven simulation**
  - Simulated time advances to time of next event
  - Events are generated by using a mathematical model based on statistics and probability

QUEUE APPLICATIONS

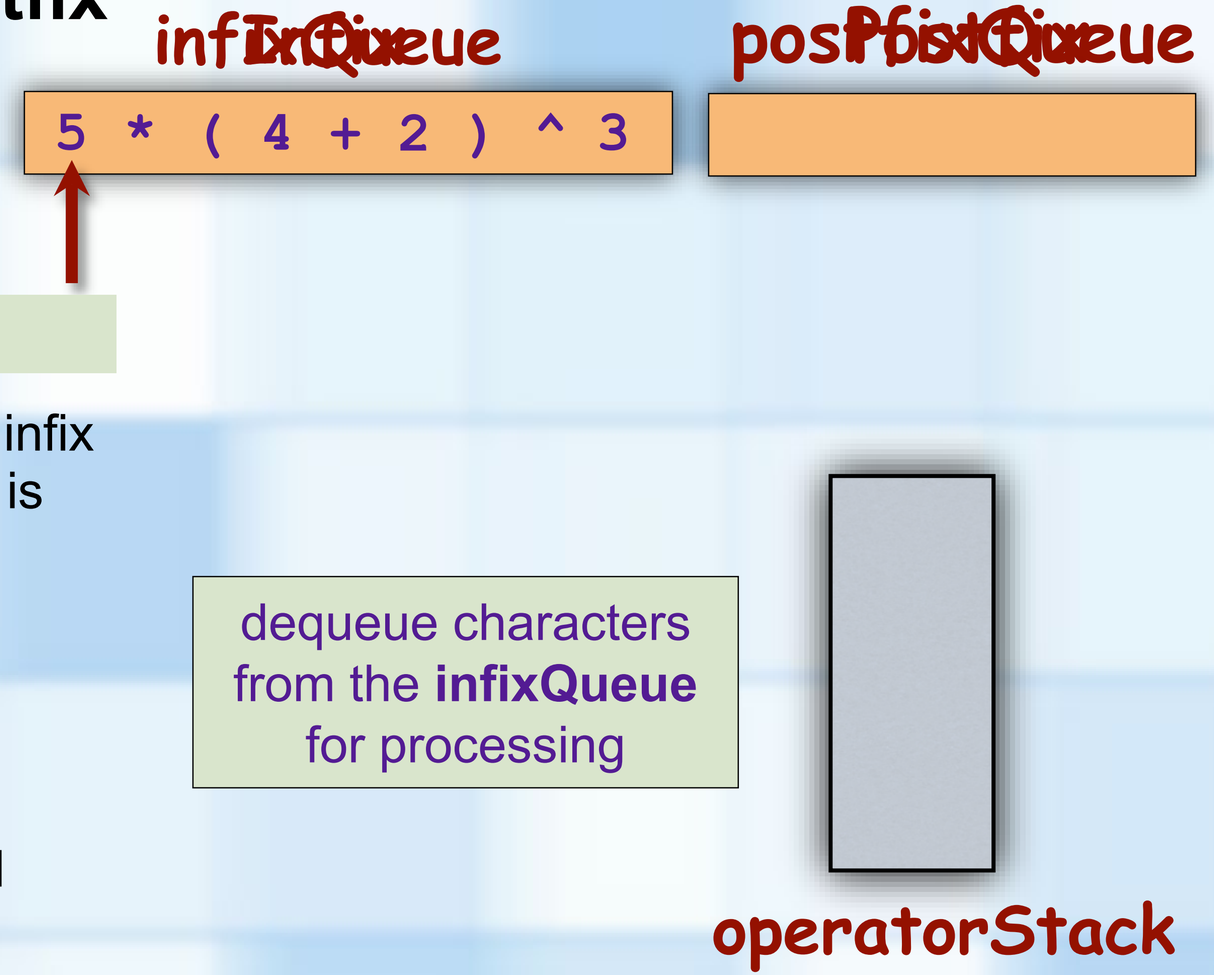
- **Event Simulation Event Categories**
  - ***External events:***
    - input file specifies times at which the events occur
    - pre-generated test data for re-running simulation with consistent input
  - ***Internal events:***
    - the simulation determines the times at which the events occur
    - can be random or based on other events
- **Bank simulation**
  - Event-driven and uses an event list
  - Arrival events are external
  - Departure events based on when customer gets to teller
  - simulation tracks events that have not occurred yet

Pearson

Ch13a The ADT Queue

Evaluating Infix Expressions

- **Converting Infix Expressions to Postfix**
  - `Operand` - a[illegible] enqueue on the **postFixQueue** [illegible]n
  - `Operators * / + -`
    - if **operatorStack** is not empty;
      - pop operators and ap[illegible] enqueue on the **postFixQueue**
      - if their precedence >= that of the operator in the infix expression until **operatorStack** is empty or a `(` is reached
      - Push operator onto the **operatorStack**,
  - `(` - Push onto **operatorStack**
  - `)` - pop operators from **operatorStack** and ap[illegible] enqueue on the **postFixQueue** until `(` is popped
  - `Operator ^`
    - Push operator onto the **operatorStack**,



Evaluating Infix Expressions

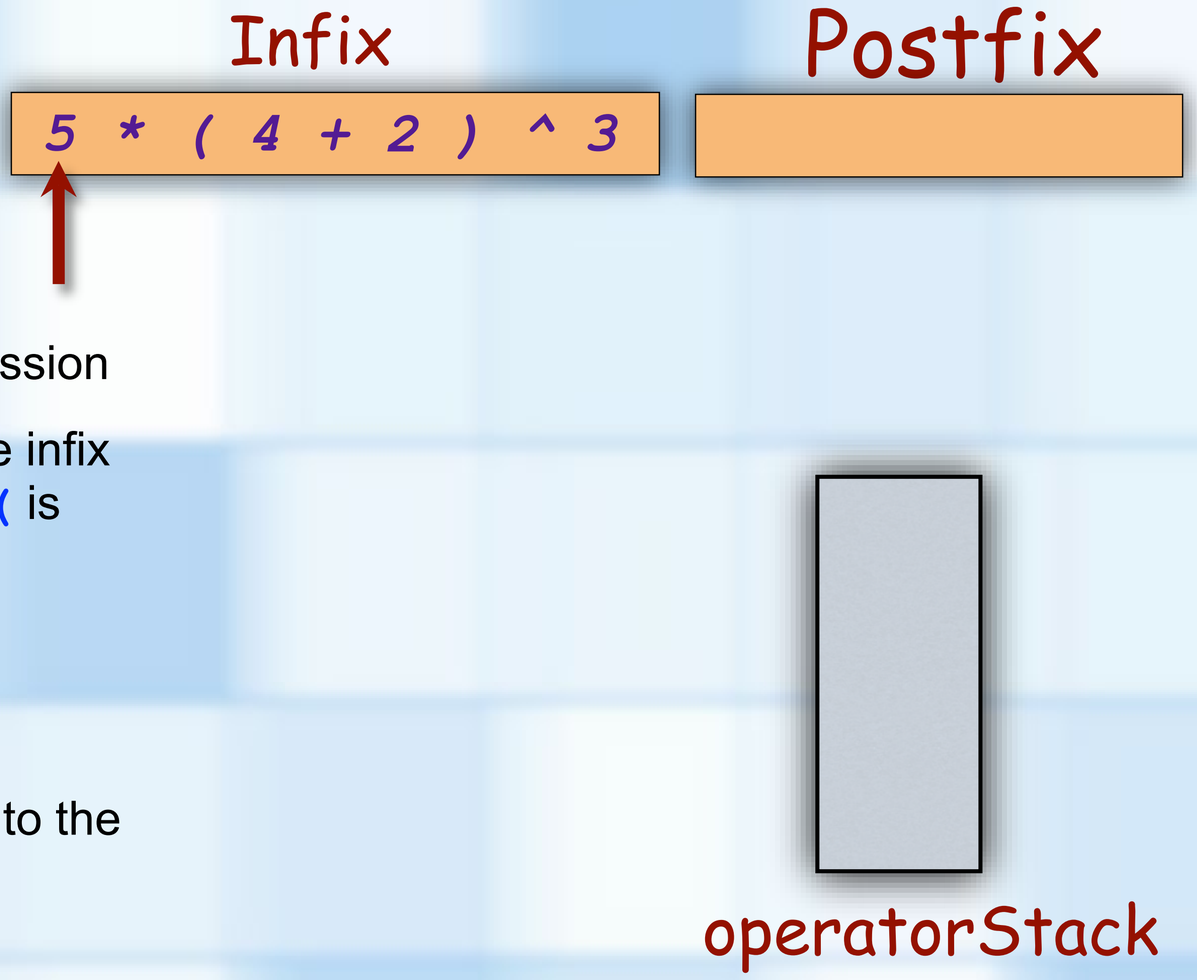
- **To evaluate an infix expression:**

Convert the infix expression to postfix form

Evaluate the postfix expression

Evaluating Infix Expressions

- **Converting Infix Expressions to Postfix**
  - `Operand` - append to the postFix expression
  - `Operators * / + -`
    - if **operatorStack** is not empty;
      - pop operators and append to the postFix expression
      - if their precedence >= that of the operator in the infix expression until **operatorStack** is empty or a `(` is reached
      - Push operator onto the **operatorStack**,
  - `(` - Push onto **operatorStack**
  - `)` - pop operators from **operatorStack** and append to the postFix expression until `(` is popped
  - `Operator ^`
    - Push operator onto the **operatorStack**,



Evaluating Infix Expressions

- **To evaluate an infix expression**
  - Convert the infix expression to postfix form
  - Evaluate the postfix expression

```
Algorithm evaluatePostfix(postfix)
// Evaluates a postfix expression.
valueStack = a new empty stack

while (postfixExpression has characters left to parse)
{
    nextCharacter = next non blank character of postfixExpression
    switch (nextCharacter)
    {
       case variable:
          valueStack.push(value of the variable nextCharacter)
          break

       case '+' : case '-' : case '*' : case '/' : case '^' :
           operandTwo = valueStack.peek()
           valueStack.pop()
           operandOne = valueStack.peek()
           valueStack.pop()
           result = the result of the operation
                        in nextCharacter and its operands operandOne and operandTwo
           valueStack.push(result)
           break

        default: break
     } // end switch
} // end while

return valueStack.peek()
```