CS 91 USACO

Bronze Division

Unit 2: 1-D Data Structures

LECTURE 7: STACKS

DR. ERIC CHOU

IEEE SENIOR MEMBER

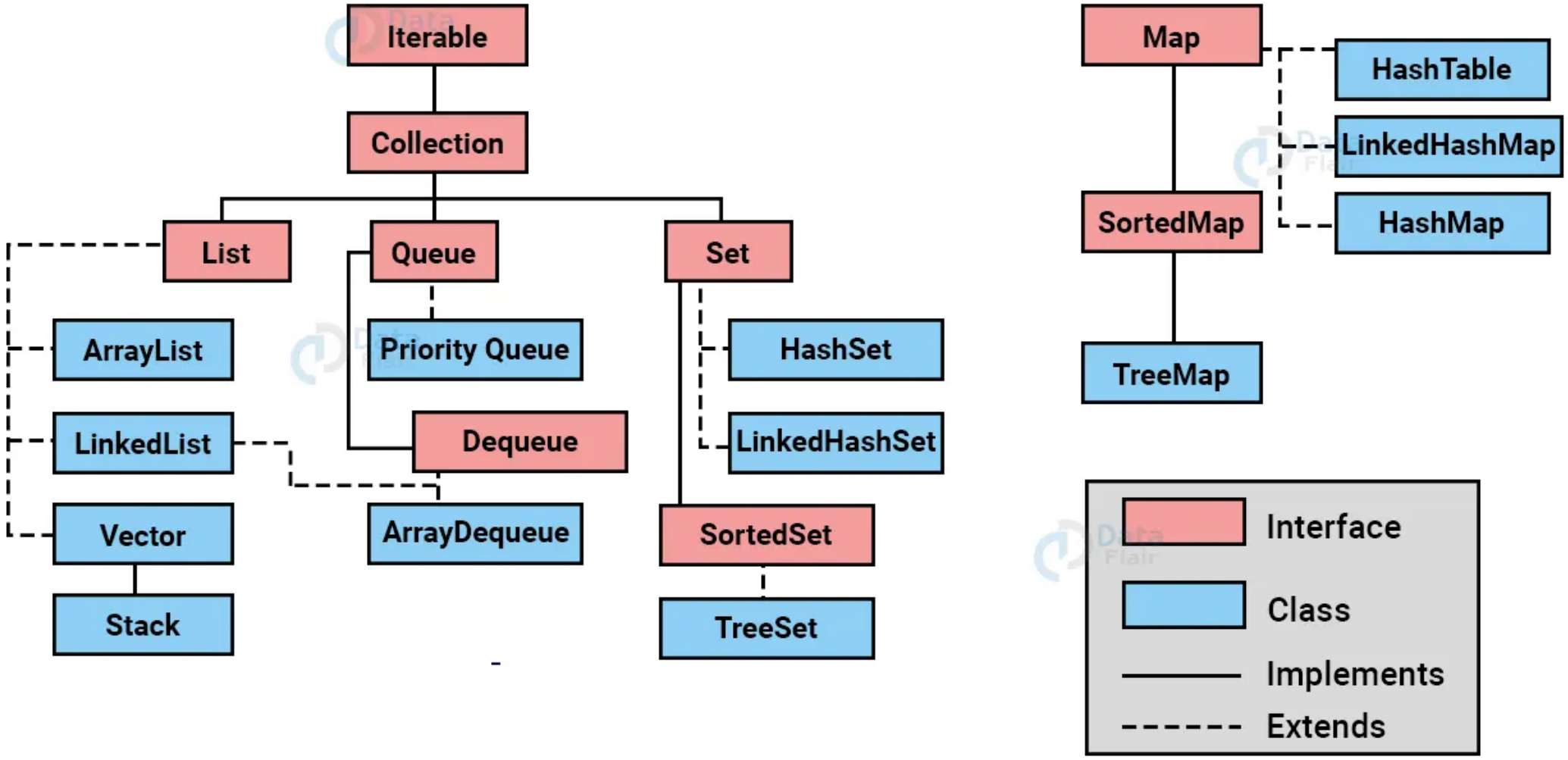
Objectives

- Study the conceptual idea about stack
- Backtracking with Stack
- Infix-postfix-prefix
- Postfix evaluation
- Conversion among algebraic expressions

Java Collections

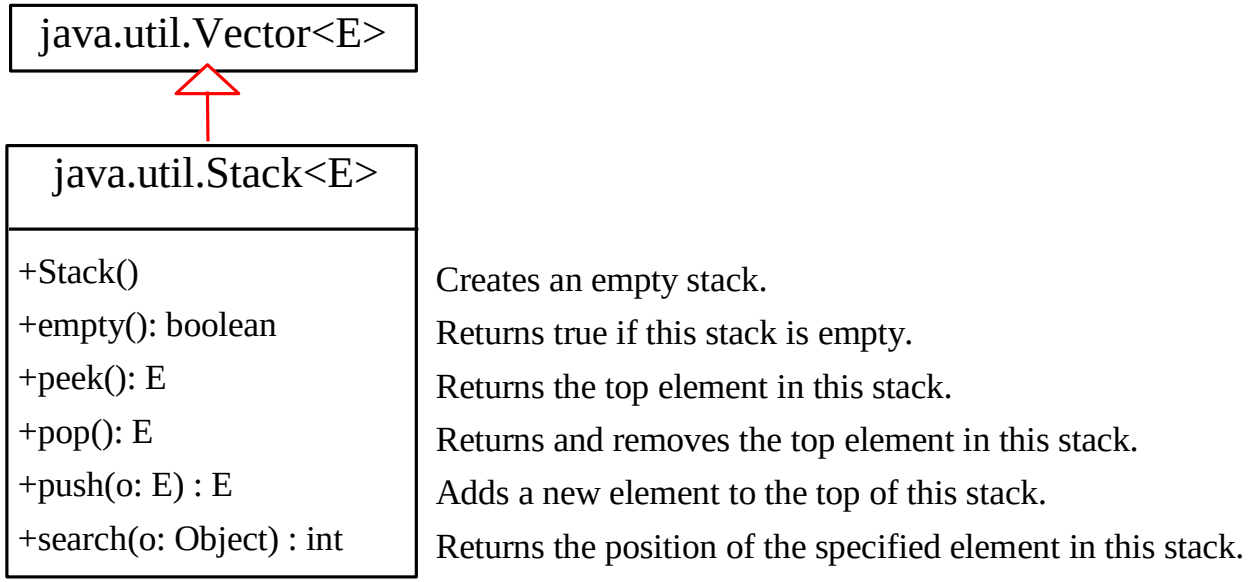
SECTION 1

Hierarchy of Collection Framework in Java



java.util.Stack

- The Stack class represents a last-in-first-out stack of objects. The elements are accessed only from the top of the stack. You can retrieve, insert, or remove an element from the top of the stack.



The Stack Class

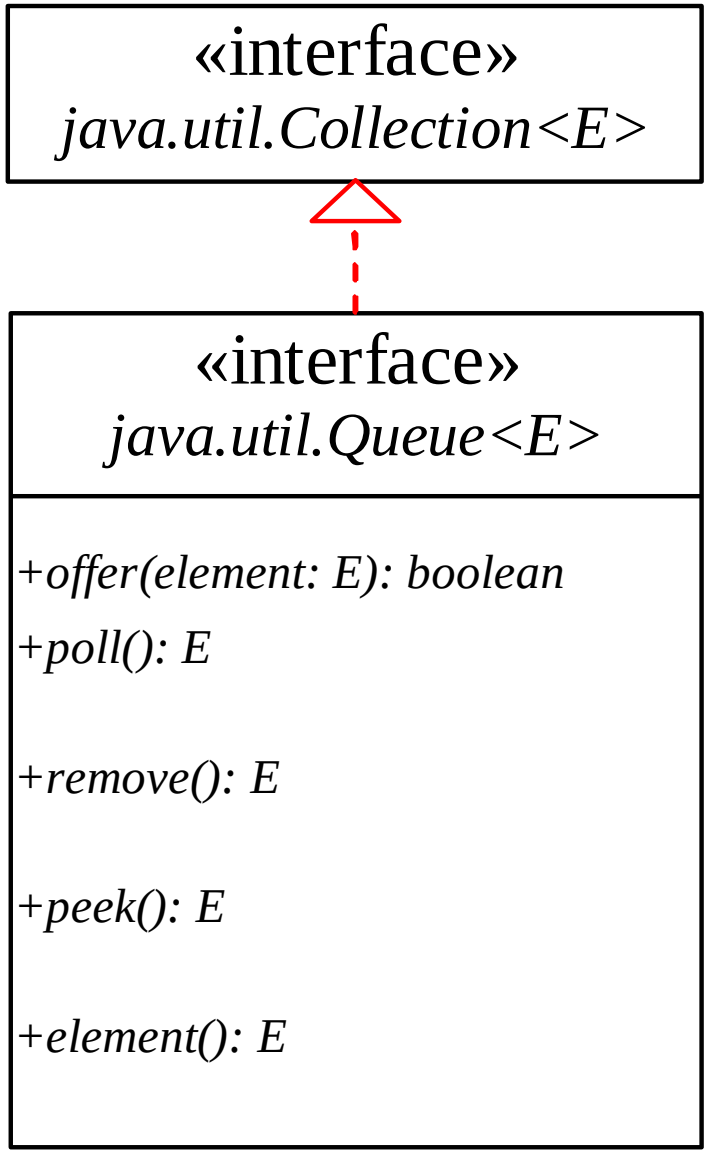
java.util.Stack

- Stack is a subclass of Vector that implements a standard last-in, first-out stack.
- Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

Demonstration Program

STACKDEMO.JAVA

java.util.Queue



Inserts an element to the queue.

Retrieves and removes the head of this queue, or null if this queue is empty.

Retrieves and removes the head of this queue and throws an exception if this queue is empty.

Retrieves, but does not remove, the head of this queue, returning null if this queue is empty.

Retrieves, but does not remove, the head of this queue, throwing an exception if this queue is empty.

The Queue Interface

java.Collection.Queue

- Being a Collection subtype all methods in the Collection interface are also available in the Queue interface.
- Since Queue is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following Queue implementations in the Java Collections API:
  - **java.util.LinkedList**
  - **java.util.PriorityQueue**

LinkedList (Concrete Class)
PriorityQueue (Concrete Class)

Both implements Queue

LinkedList is a pretty standard queue implementation.
PriorityQueue stores its elements internally according to their natural order (if they implement Comparable), or according to a Comparator passed to the **PriorityQueue**.
There are also **Queue** implementations in the **java.util.concurrent** package, but not our topic.

Here are a few examples of how to create a Queue instance:

- Queue queueA = new LinkedList();
- Queue queueB = new PriorityQueue();

Demonstration Program

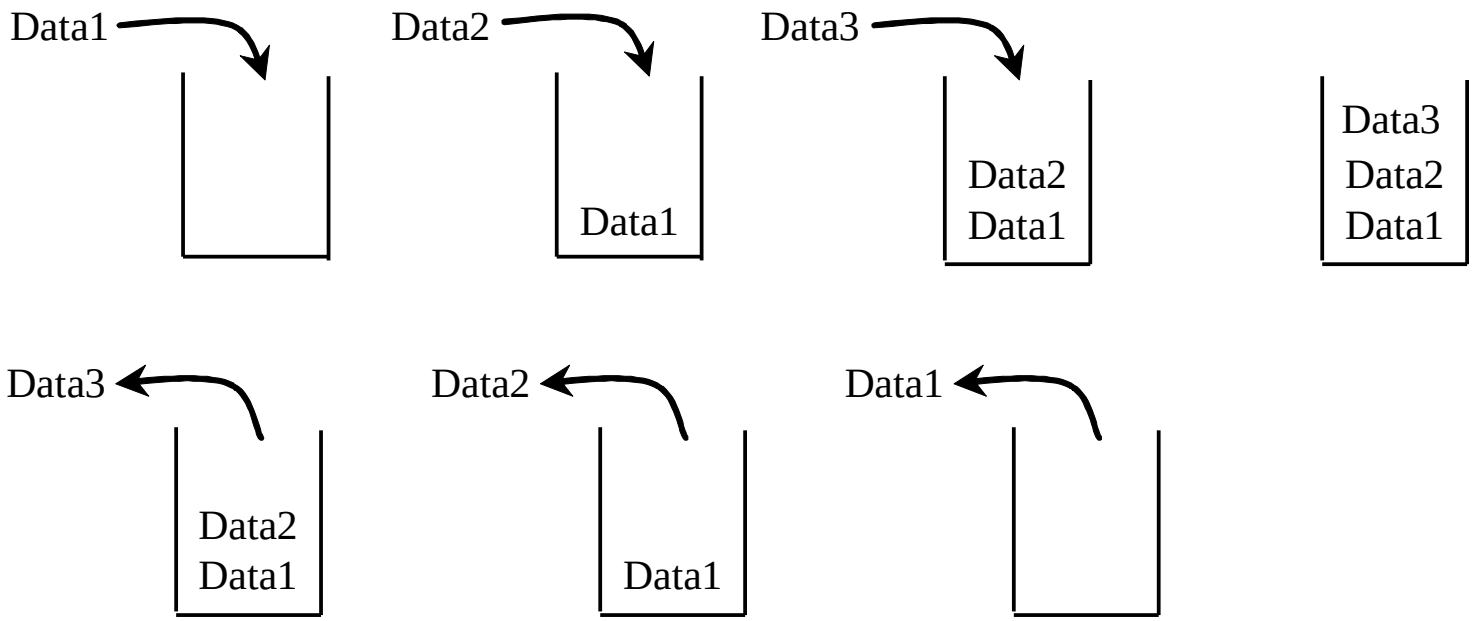
QUEUEDEMO.JAVA

Stacks Class and Queue Interface

SECTION 2

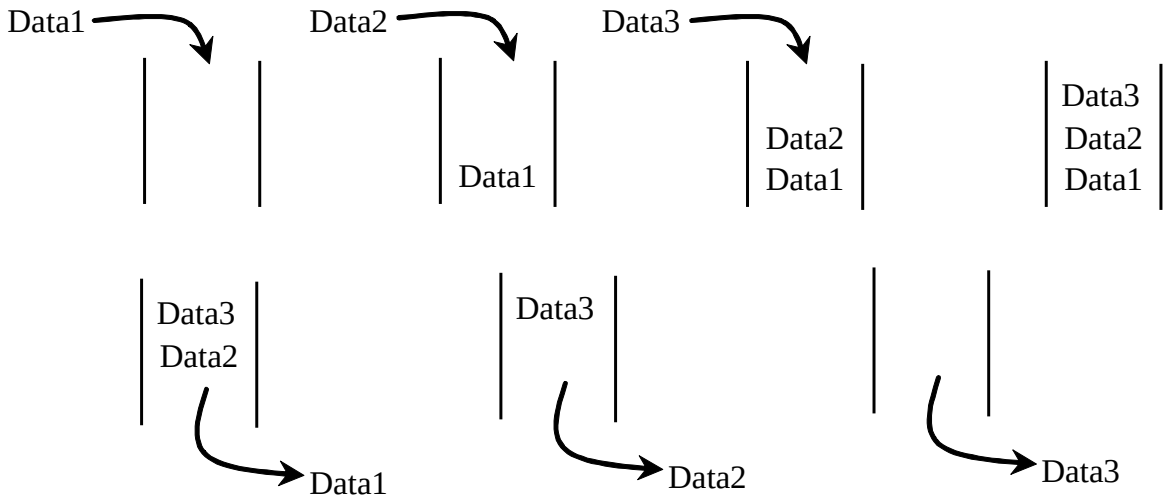
Stacks
Array Version of MyStack shown in Chapter 10

A stack can be viewed as a special type of list, where the elements are accessed, inserted, and deleted only from the end, called the top, of the stack. (FILO)



Queue

A queue represents a waiting list. A queue can be viewed as a special type of list, where the elements are inserted into the end (tail) of the queue, and are accessed and deleted from the beginning (head) of the queue. (FIFO)



Implementing Stack and Queue Classes

- Using an array list to implement Stack
- Use a linked list to implement Queue

Since the insertion and deletion operations on a stack are made only at the end of the stack, using an array list to implement a stack is more efficient than a linked list. Since deletions are made at the beginning of the list, it is more efficient to implement a queue using a linked list than an array list. This section implements a stack class using an array list and a queue using a linked list.

Design of the Stack and Queue

There are two ways to design the stack and queue classes:

- Using inheritance: You can define the stack class by extending the array list class, and the queue class by extending the linked list class.

(a) Using inheritance

- Using composition: You can define an array list as a data field in the stack class, and a linked list as a data field in the queue class.

(b) Using composition

Composition is Better

Both designs are fine, but using composition is better because it enables you to define a complete new stack class and queue class without inheriting the unnecessary and inappropriate methods from the array list and linked list.

MyStack and MyQueue

GenericStack<E>	
-list: java.util.ArrayList<E>	An array list to store elements.
+GenericStack()	Creates an empty stack.
+getSize(): int	Returns the number of elements in this stack.
+peek(): E	Returns the top element in this stack.
+pop(): E	Returns and removes the top element in this stack.
+push(o: E): void	Adds a new element to the top of this stack.
+isEmpty(): boolean	Returns true if the stack is empty.

GenericQueue<E>	
-list: LinkedList<E>	
+enqueue(e: E): void	Adds an element to this queue.
+dequeue(): E	Removes an element from this queue.
+getSize(): int	Returns the number of elements from this queue.

Using Stacks and Queues

- Write a program that creates a stack using MyStack and a queue using MyQueue. It then uses the push (enqueue) method to add strings to the stack (queue) and the pop (dequeue) method to remove strings from the stack (queue).

Demonstration Program

TESTSTACKQUEUE.JAVA

GENERICSTACK.JAVA

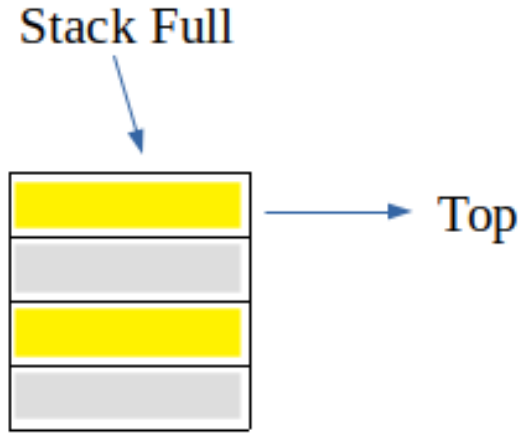
GENERICQUEUE.JAVA

Stack Implementation

SECTION 3

A Simple Array-Based Stack Implementation

- Recalling that arrays start at index 0 in Java, when the stack holds elements from data[0] to data[t] inclusive, it has size t +1. By convention, when the stack is empty it will have t equal to −1 (and thus has size t +1, which is 0).
- A complete Java implementation based on this strategy(with Javadoc comments omitted due to space considerations).



```
1   public class ArrayStack<E> implements Stack<E> {
2   public static final int CAPACITY=1000; // default array capacity
3   private E[ ] data; // generic array used for storage
4   private int t = −1; // index of the top element in stack
5   public ArrayStack( ) { this(CAPACITY); } // constructs stack with default capacity
6   public ArrayStack(int capacity) { // constructs stack with given capacity
7       data = (E[ ]) new Object[capacity]; // safe cast; compiler may give warning
8   }
9   public int size( ) { return (t + 1); }
10  public boolean isEmpty( ) { return (t == −1); }
11  public void push(E e) throws IllegalStateException {
12      if (size( ) == data.length) throw new IllegalStateException("Stack is full");
13      data[++t] = e; // increment t before storing new item
14  }
15  public E top( ) {
16      if (isEmpty( )) return null;
17      return data[t];
18  }
19  public E pop( ) {
20      if (isEmpty( )) return null;
21      E answer = data[t];
22      data[t] = null; // dereference to help garbage collection
23      t−−;
24      return answer;
25  }
26 }
```

```
public class TestArrayStack
{
    public static void main(String[] args){
        System.out.print("\f");
        Stack<Integer> S = new ArrayStack<>();  // contents: ()
        S.push(5);                              // contents: (5)
        S.push(3);                              // contents: (5, 3)
        System.out.println(S.size());           // contents: (5, 3) outputs 2
        System.out.println(S.pop());            // contents: (5) outputs 3
        System.out.println(S.isEmpty());        // contents: (5) outputs false
        System.out.println(S.pop());            // contents: () outputs 5
        System.out.println(S.isEmpty());        // contents: () outputs true
        System.out.println(S.pop());            // contents: () outputs null
        S.push(7);                              // contents: (7)
        S.push(9);                              // contents: (7, 9)
        System.out.println(S.top());            // contents: (7, 9) outputs 9
        S.push(4);                              // contents: (7, 9, 4)
        System.out.println(S.size());           // contents: (7, 9, 4) outputs 3
        System.out.println(S.pop());            // contents: (7, 9) outputs 4
        S.push(6);                              // contents: (7, 9, 6)
        S.push(8);                              // contents: (7, 9, 6, 8)
        System.out.println(S.pop());            // contents: (7, 9, 6) outputs 8
    }
}
```

LinkedStack<E>

- Stack Built by Linked List

Demonstration Program

LINKEDSTACK.JAVA

Stack Applications

SECTION 4

Matching Parentheses and HTML Tags

- In this subsection, we explore two related applications of stacks, both of which involve testing for pairs of matching delimiters. In our first application, w consider arithmetic expressions that may contain various pairs of grouping symbols, such as
  - Parentheses: “(” and “)”
  - Braces: “{” and “}”
  - Brackets: “[” and “]”
- Each opening symbol must match its corresponding closing symbol. For example, a left bracket, “[,” must match a corresponding right bracket, “],” as in the following expression
- $[(5+x)-(y+z)]$.

Matching Parentheses and HTML Tags

- The following examples further illustrate this concept:
  - Correct: ( )(( )){([( )])}
  - Correct: ((( )(( )){([( )])}))
  - Incorrect:)(( )){([( )])}
  - Incorrect: ({[ ])}
  - Incorrect: (
- We leave the precise definition of a matching group of symbols to Exercise R-6.6.

Matching Delimiters

```
1 /** Tests if delimiters in the given expression are properly matched. */
2 public static boolean isMatched(String expression) {
3       final String opening = "({["; // opening delimiters
4       final String closing = ")}]"; // respective closing delimiters
5       Stack<Character> buffer = new LinkedStack<>( );
6       for (char c : expression.toCharArray( )) {
7           if (opening.indexOf(c) != −1) // this is a left delimiter
8               buffer.push(c);
9           else if (closing.indexOf(c) != −1) { // this is a right delimiter
10              if (buffer.isEmpty( )) // nothing to match with
11                  return false;
12              if (closing.indexOf(c) != opening.indexOf(buffer.pop( )))
13                  return false; // mismatched delimiter
14          }
15      }
16      return buffer.isEmpty( ); // were all opening delimiters matched?
17 }
```

```
1 /** Tests if every opening tag has a matching closing tag in HTML string. */
2 public static boolean isHTMLMatched(String html) {
3     Stack<String> buffer = new LinkedStack<>( );
4     int j = html.indexOf('<'); // find first ’<’ character (if any)
5     while (j != −1) {
6         int k = html.indexOf('>', j+1); // find next ’>’ character
7         if (k == −1)
8             return false; // invalid tag
9         String tag = html.substring(j+1, k); // strip away < >
10        if (!tag.startsWith("/")) // this is an opening tag
11            buffer.push(tag);
12        else { // this is a closing tag
13            if (buffer.isEmpty( ))
14                return false; // no tag to match
15            if (!tag.substring(1).equals(buffer.pop( )))
16                return false; // mismatched tag
17        }
18        j = html.indexOf('<', k+1); // find next ’<’ character (if any)
19    }
20    return buffer.isEmpty( ); // were all opening tags matched?
21 }
```

Parentheses

SECTION 5

Simple Balanced Parentheses

- We now turn our attention to using stacks to solve real computer science problems. You have no doubt written arithmetic expressions such as

```
(5+6)*(7+8)/(4+3)(5+6)*(7+8)/(4+3)
```

- where parentheses are used to order the performance of operations. You may also have some experience programming in a language such as Lisp with constructs like

```
( defun square(n) (* n n))
```

- This defines a function called **square** that will return the square of its argument n. Lisp is notorious for using lots and lots of parentheses.

Balanced parentheses

- In both of these examples, parentheses must appear in a balanced fashion. Balanced parentheses means that each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested. Consider the following correctly balanced strings of parentheses:

```
(( )( )( )( ))
((( )))
(( )((( ))( )))
```

Balanced parentheses

- Compare those with the following, which are not balanced:

```
(((((((( ))
( )))
(( )( )(( )
```

- The ability to differentiate between parentheses that are correctly balanced and those that are unbalanced is an important part of recognizing many programming language structures.

Balanced parentheses

- The challenge then is to write an algorithm that will read a string of parentheses from left to right and decide whether the symbols are balanced. To solve this problem we need to make an important observation. As you process symbols from left to right, the most recent opening parenthesis must match the next closing symbol (see Figure 4).
- Also, the first opening symbol processed may have to wait until the very last symbol for its match. Closing symbols match opening symbols in the reverse order of their appearance; they match from the inside out. This is a clue that stacks can be used to solve the problem.

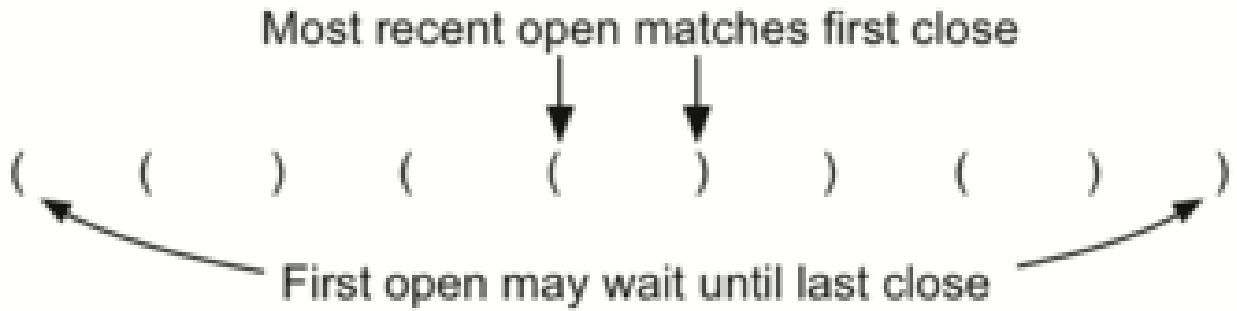


Figure 4

Balanced parentheses

- Once you agree that a stack is the appropriate data structure for keeping the parentheses, the statement of the algorithm is straightforward.
- Starting with an empty stack, process the parenthesis strings from left to right. If a symbol is an opening parenthesis, push it on the stack as a signal that a corresponding closing symbol needs to appear later. If, on the other hand, a symbol is a closing parenthesis, pop the stack.
- As long as it is possible to pop the stack to match every closing symbol, the parentheses remain balanced. If at any time there is no opening symbol on the stack to match a closing symbol, the string is not balanced properly.
- At the end of the string, when all symbols have been processed, the stack should be empty. The Python code to implement this algorithm is shown in **parcheckers.py**.

Demo Program: Parenthesis.java

- In no stack is used, we can only detect the matching of a single kind of parenthesis using counters.

```java
import java.util.*;
public class Parenthesis
{
   public static boolean match(String x){
      int count = 0;
      for (int i=0; i<x.length(); i++){
          char ch = x.charAt(i);
          if (ch=='(') count++;
          if (ch==')') count--;
          if (count < 0) return false;
        }
      return count==0;
    }
   public static void main(String[] args){
        System.out.print("\f");

        Scanner input = new Scanner(System.in);
        System.out.print("Enter an Expression: ");
        String x = input.nextLine().trim();

        System.out.printf("%s is matched is %b\n", x, match(x));
    }
}
```

True
False

Demo Program: ParenthesisWithStack.java

- This function, **match**, assumes that a **Stack** class is available and returns a boolean result as to whether the string of parentheses is balanced. Note that the boolean variable balanced is initialized to **True** as there is no reason to assume otherwise at the start. If the current symbol is (, then it is pushed on the stack.
- Note also that pop simply removes a symbol from the stack. The returned value is not used since we know it must be an opening symbol seen earlier.
- At the end, as long as the expression is balanced and the stack has been completely cleaned off, the string represents a correctly balanced sequence of parentheses.

```java
import java.util.*;
public class ParenthesisWithStack
{
   public static boolean match(String x){
       Stack<String> st = new Stack<String>();
       for (int i=0; i<x.length(); i++){
          String ch = x.substring(i, i+1);
          if (ch.equals("(")) st.push(ch);
          if (ch.equals(")")){
              if (st.isEmpty()) return false;
              st.pop();
            }
        }
       return st.isEmpty();
    }
   public static void main(String[] args){
        System.out.print("\f");

        Scanner input = new Scanner(System.in);
        System.out.print("Enter an Expression: ");
        String x = input.nextLine().trim();

        System.out.printf("%s is matched is %b\n", x, match(x));
    }
}
```

True
False

Algebraic Expressions

SECTION 7

Algebraic Expressions

- An algebraic expression can be defined as follows...

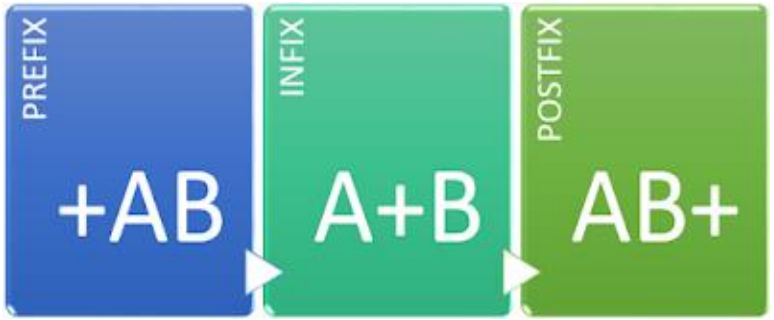
An algebraic expression is a combination of operands and operators that represents a specific value.

- Example : A = B + C; denote an expression in which there are 3 operands A, B, C and two operator + and =
- An **operator** is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in program to manipulate data and variables
- **Operands** are the values on which the operators can perform the task.

Types of Algebraic Expression

An algebraic expression can be represented in three different ways based on the operator position. They are as follows...

1.Prefix Expression
2.Infix Expression
3.Postfix Expression



Infix, Prefix and Postfix Expressions

Infix	Prefix	Postfix
(2 + 8) * (7 % 3)	* + 2 8 % 7 3	2 8 + 7 3 % *
((2 * 3) + 5) % 4	% + * 2 3 5 4	2 3 * 5 + 4 %
((2 * 5) % (6 / 4)) + (2 * 3)	+ % * 2 5 / 6 4 * 2 3	2 5 * 6 4 / % 2 3 * +
1 + (2 + (3 + 4))	+ 1 + 2 + 3 4	1 2 3 4 + + +
((1 + 2) + 3) + 4	+ + + 1 2 3 4	1 2 + 3 + 4 +

Expression with Precedence Levels

SECTION 8

Precedence Level

- Each operator has a precedence level.
- Operators of higher precedence are used before operators of lower precedence.
- The only thing that can change that order is the presence of parentheses. The precedence order for arithmetic operators places multiplication and division above addition and subtraction. If two operators of equal precedence appear, then a left-to-right ordering or associativity is used.
- Let's interpret the troublesome expression A + B * C using operator precedence.
- B and C are multiplied first, and A is then added to that result. (A + B) * C would force the addition of A and B to be done first before the multiplication.
- In expression A + B + C, by precedence (via associativity), the leftmost + would be done first.

Fully Parenthesized Expressions

- One way to write an expression that guarantees there will be no confusion with respect to the order of operations is to create what is called a fully parenthesized expression.
- This type of expression uses one pair of parentheses for each operator. The parentheses dictate the order of operations; there is no ambiguity. There is also no need to remember any precedence rules.
- The expression A + B * C + D can be rewritten as ((A + (B * C)) + D) to show that the multiplication happens first, followed by the leftmost addition. A + B + C + D can be written as (((A + B) + C) + D) since the addition operations associate from left to right.

ad hoc Method: Infix to Postfix Conversion

SECTION 9

infix to postfix conversion using Stack

Step 1: Scan all the symbols one by one from left to right in the given Infix Expression and repeat steps 2-5

Step 2: If the scanned symbol is an **operand**, then place directly in the postfix expression

Step 3: If the scanned symbol is **left parenthesis '('**, push it onto the **STACK**.

Step 4: If the scanned symbol is an **operator (✕)**, then:

Repeatedly pop from **STACK** and add to postfix expression each operator(On the top of STACK) , which has the same precedence as or higher precedence than (✕)

Add (✕) to Stack

Step 5: If the symbol scanned is a **right parenthesis ')'**

Repeatedly pop from STACK and add to postfix expression each operator(On the top of STACK) , till we get the matching left parenthesis.

Remove the left parenthesis [Do not add the left parenthesis to postfix expression]

Step 6: EXIT

Example

Consider the following Infix Expression...

(A + B) * (C - D)

- The given infix expression can be converted into postfix expression using Stack data Structure as follows...

Reading Character	STACK	Postfix Expression
Initially	Stack is EMPTY	EMPTY
(	Push '('	EMPTY
A	No operation Since 'A' is OPERAND	A
+	'+' has low priority than '(' so, PUSH '+'	A
B	No operation Since 'B' is OPERAND	A B
)	POP all elements till we reach '(' POP '+' POP '('	A B +
*	Stack is EMPTY & '*' is Operator PUSH '*'	A B +
(	PUSH '('	A B +
C	No operation Since 'C' is OPERAND	A B + C
-	'-' has low priority than '(' so, PUSH '-'	A B + C
D	No operation Since 'D' is OPERAND	A B + C D
)	POP all elements till we reach '(' POP '-' POP '('	A B + C D -
$	POP all elements till Stack becomes Empty	A B + C D - *

Systematical Methods:

Conversion of Infix Expressions to Prefix

SECTION 10

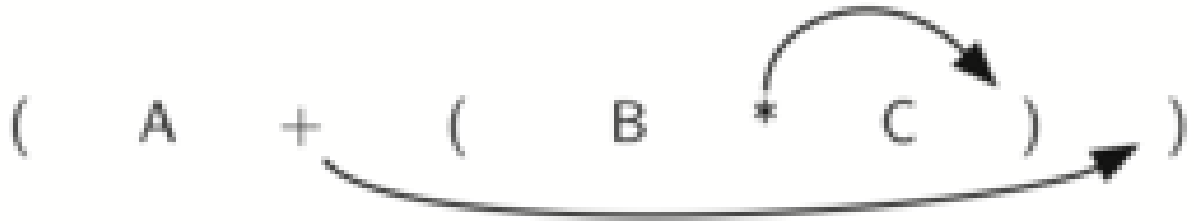
Fully Parenthesized Expression

- So far, we have used ad hoc methods to convert between infix expressions and the equivalent prefix and postfix expression notations. As you might expect, there are algorithmic ways to perform the conversion that allow any expression of any complexity to be correctly transformed.
- The first technique that we will consider uses the notion of a **fully parenthesized** expression that was discussed earlier. Recall that A + B * C can be written as (A + (B * C)) to show explicitly that the multiplication has precedence over the addition. On closer observation, however, you can see that each parenthesis pair also denotes the beginning and the end of an operand pair with the corresponding operator in the middle.

Example

infix to postfix

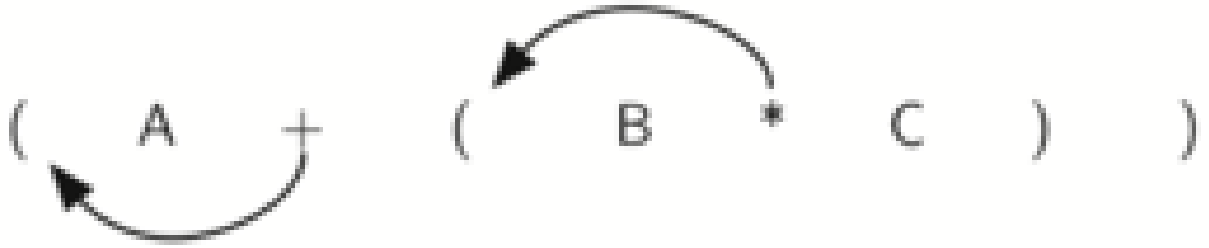
- Look at the right parenthesis in the subexpression (B * C) above. If we were to move the multiplication symbol to that position and remove the matching left parenthesis, giving us B C *, we would in effect have converted the subexpression to postfix notation.
- If the addition operator were also moved to its corresponding right parenthesis position and the matching left parenthesis were removed, the complete postfix expression would result (see Figure below).



Example

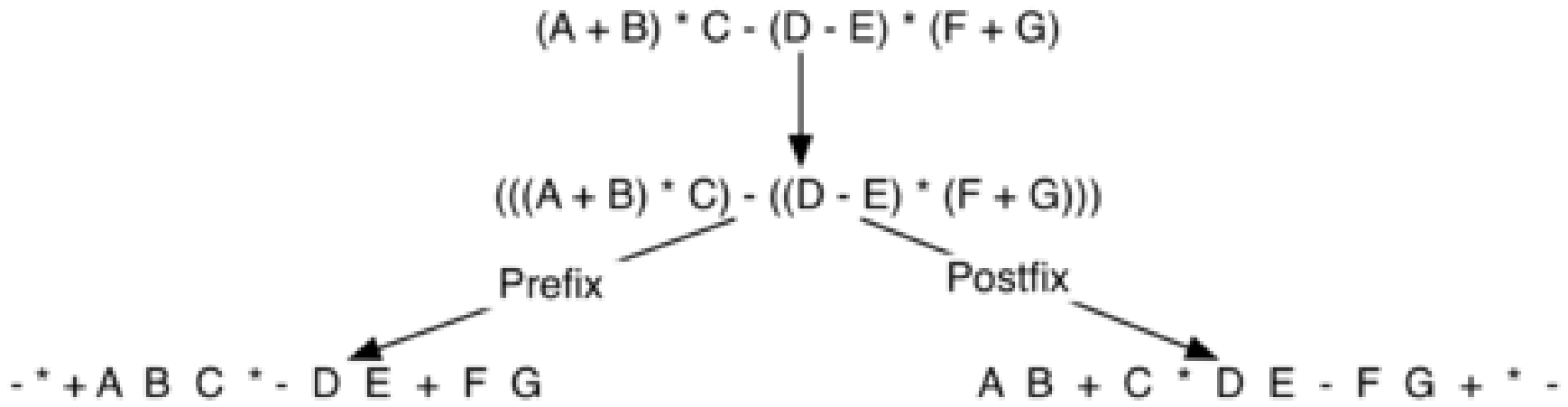
postfix to infix

- If we do the same thing but instead of moving the symbol to the position of the right parenthesis, we move it to the left, we get prefix notation (see Figure below).
- The position of the parenthesis pair is actually a clue to the final position of the enclosed operator.



Systematical Conversion

- So in order to convert an expression, no matter how complex, to either prefix or postfix notation, fully parenthesize the expression using the order of operations. Then move the enclosed operator to the position of either the left or the right parenthesis depending on whether you want prefix or postfix notation.
- Here is a more complex expression: (A + B) * C - (D - E) * (F + G). Figure Below shows the conversion to postfix and prefix notations.



Methods 1: infix to postfix

SECTION 11

General Infix-to-Postfix Conversion

- Consider once again the expression A + B * C. As shown above, A B C * + is the postfix equivalent.
- We have already noted that the operands A, B, and C stay in their relative positions.
- **It is only the operators that change position.**
- Let’s look again at the operators in the infix expression. The first operator that appears from left to right is +. However, in the postfix expression, + is at the end since the next operator, *, has precedence over addition. The order of the operators in the original expression is reversed in the resulting postfix expression.

Stack is Used for the Storage of Operators

- As we process the expression, the operators have to be saved somewhere since their corresponding right operands are not seen yet. Also, the order of these saved operators may need to be reversed due to their precedence.
- This is the case with the addition and the multiplication in this example.
- **Since the addition operator comes before the multiplication operator and has lower precedence, it needs to appear after the multiplication operator is used.**
- Because of this reversal of order, it makes sense to consider using a stack to keep the operators until they are needed.

Deal With Parentheses

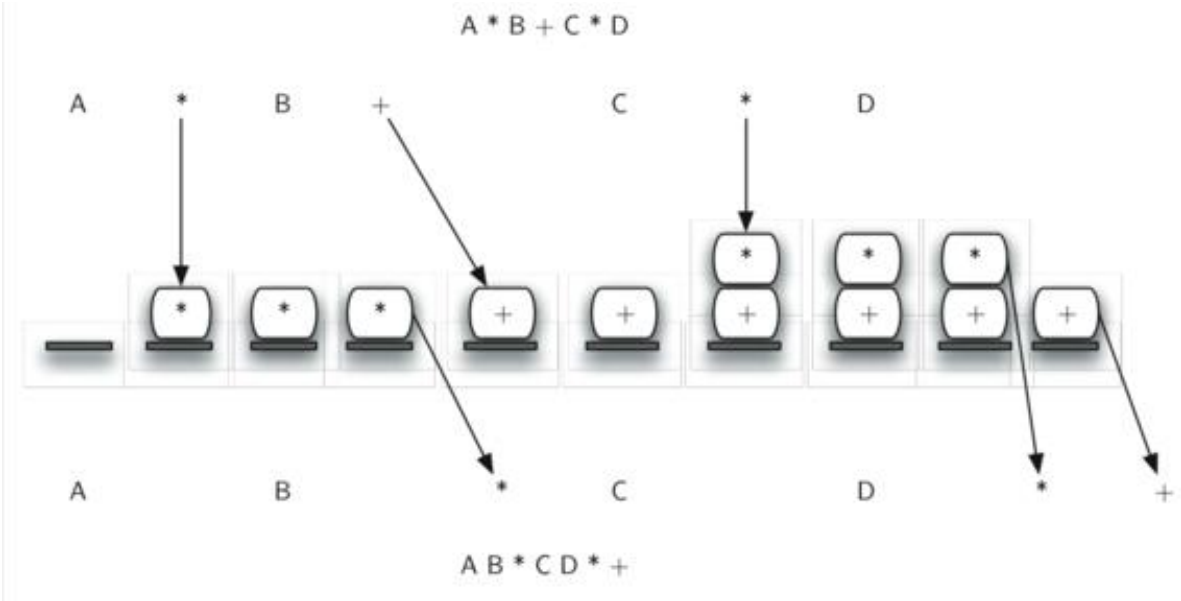
- What about (A + B) * C? Recall that A B + C * is the postfix equivalent. Again, processing this infix expression from left to right, we see + first. In this case, when we see *, + has already been placed in the result expression because it has precedence over * by virtue of the parentheses. We can now start to see how the conversion algorithm will work.
  1. When we see a left parenthesis, we will save it to denote that another operator of high precedence will be coming.
  2. That operator will need to wait until the corresponding right parenthesis appears to denote its position (recall the fully parenthesized technique).
  3. When that right parenthesis does appear, the operator can be popped from the stack.

Stack and Parentheses (I)

- As we scan the **infix expression** from left to right, we will use a **stack** to keep the operators.
- This will provide the reversal that we noted in the first example.
- The top of the stack will always be the most recently saved operator.
- Whenever we read a new operator, we will need to consider how that operator compares in precedence with the operators, if any, already on the stack.
- Assume the infix expression is a string of tokens delimited by spaces. The operator tokens are *, /, +, and -, along with the left and right parentheses, (and). The operand tokens are the single-character identifiers A, B, C, and so on.
- The following steps will produce a string of tokens in postfix order. (Next Page)

Stack and Parentheses (II)

1. Create an empty stack called **opstack** for keeping operators. Create an empty list for output.
2. Convert the input infix string to a list by using the string method **split**.
3. Scan the token list from left to right.
   - If the token is an operand, append it to the end of the output list.
   - If the token is a left parenthesis, push it on the **opstack**.
   - If the token is a right parenthesis, pop the **opstack** until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
   - **If the token is an operator, *, /, +, or -, push it on the opstack. However, first remove any operators already on the opstack that have higher or equal precedence and append them to the output list.**
4. When the input expression has been completely processed, check the opstack. Any operators still on the stack can be removed and appended to the end of the output list.



Stack and Parentheses (III)

- Figure on the left shows the conversion algorithm working on the expression A * B + C * D. Note that the first * operator is removed upon seeing the + operator. Also, + stays on the stack when the second * occurs, since multiplication has precedence over addition.
- At the end of the infix expression the stack is popped twice, removing both operators and placing + as the last operator in the postfix expression.

Demo Program: InfixToPostfix.java

In order to code the algorithm in Python, we will use a dictionary called prec to hold the precedence values for the operators. This dictionary will map each operator to an integer that can be compared against the precedence levels of other operators (we have arbitrarily used the integers 3, 2, and 1). The left parenthesis will receive the lowest value possible. This way any operator that is compared against it will have higher precedence and will be placed on top of it. Line 15 defines the operands to be any upper-case character or digit. The complete conversion function is shown in ActiveCode 1.

```java
import java.util.*;
public class InfixToPostFix
{
    public static String infixToPostfix(String expr){
       Map<String, Integer> prec = new HashMap<String, Integer>();
       prec.put("*", 3); prec.put("/", 3);
       prec.put("+", 2); prec.put("-", 2);
       prec.put("(", 1);

       Stack<String> opStack = new Stack<String>();
       ArrayList<String> postfixList = new ArrayList<String>();
       String[] tokens = expr.split(" ");

       for (String t: tokens){
           t = t.trim();
           if (t.length()==0) continue;
           char first = t.charAt(0);
```

```
        if (Character.isLetter(first) || Character.isDigit(first)){
            postfixList.add(t);
          }
         else if (t.equals("(")){
                opStack.push(t);
              }
         else if (t.equals(")")){
                String top = opStack.pop();
                while (!top.equals("(")){
                     postfixList.add(top);
                     top = opStack.pop();
                 }

              }
         else{
              while (!opStack.isEmpty() && prec.get(opStack.peek()) >= prec.get(t)){
                  postfixList.add(opStack.pop());
              }
              opStack.push(t);
          }
    }
```

```
    while (!opStack.isEmpty()){
        postfixList.add(opStack.pop());
     }

    String r = "";
    for (int i=0; i<postfixList.size(); i++){
        if (i==0) r += postfixList.get(i);
        else r += " "+ postfixList.get(i);
     }

    return r;
}
```

```
    public static void main(String[] args){
      System.out.print("\f");
      System.out.println(infixToPostfix("A * B + C * D"));
      System.out.println(infixToPostfix("( A + B ) * C - ( D - E ) * ( F + G )"));
      System.out.println(infixToPostfix("( A + B ) * ( C + D )"));
      System.out.println(infixToPostfix("( A + B ) * C"));
      System.out.println(infixToPostfix("A + B * C"));
    }
}

[Running] cd "c:\Eric_Chou\USACO\CS91 Bronze\BlueJ\U7 Stack\" && javac
InfixToPostFix.java && java InfixToPostFix
A B * C D * +
A B + C * D E - F G + * -
A B + C D + *
A B + C *
A B C * +
```

Methods 2: Postfix Evaluation

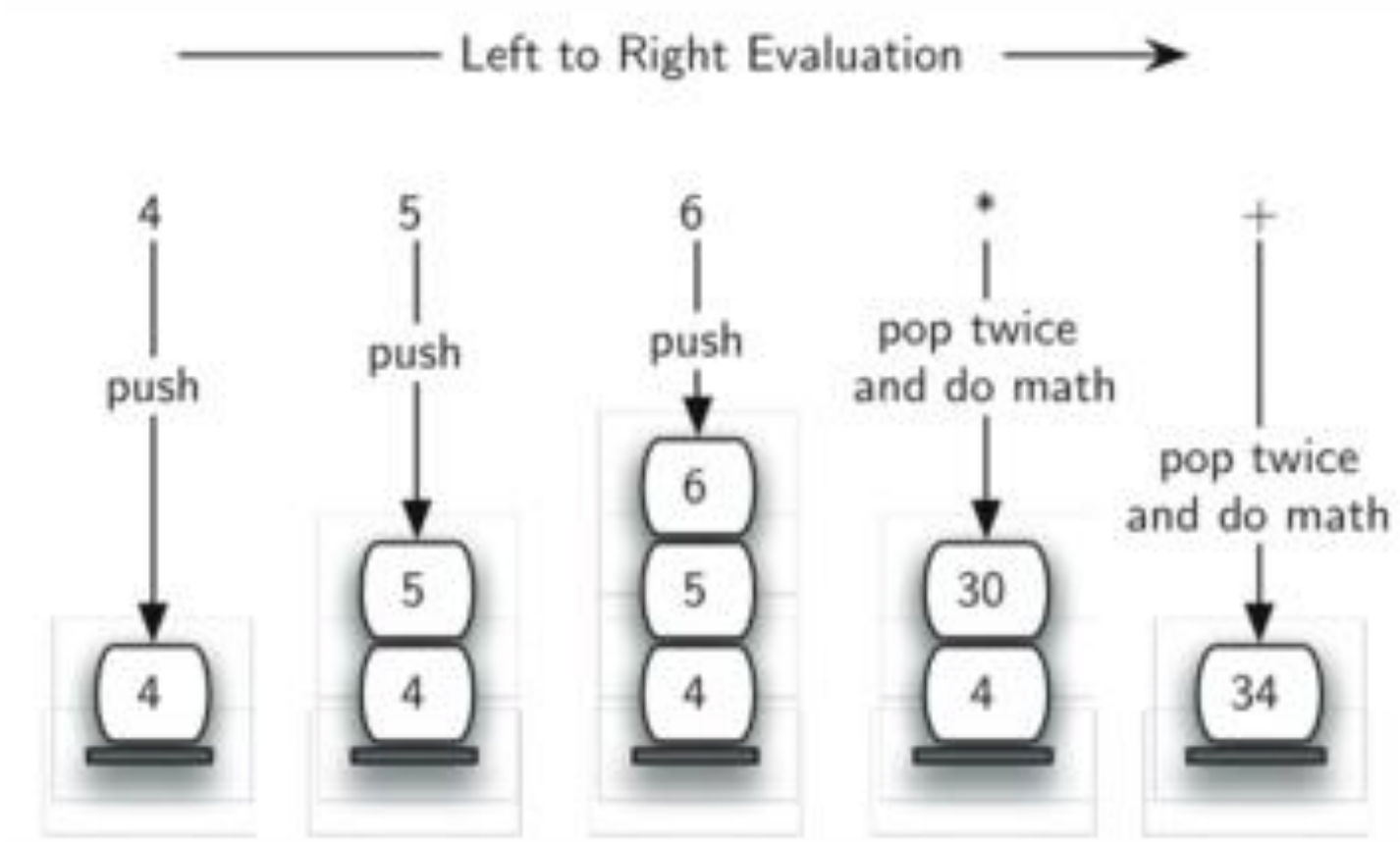
SECTION 12

Postfix Evaluation (I)

- As a final stack example, we will consider the evaluation of an expression that is already in postfix notation. In this case, a stack is again the data structure of choice. However, as you scan the postfix expression, it is the operands that must wait, not the operators as in the conversion algorithm above. Another way to think about the solution is that whenever an operator is seen on the input, the two most recent operands will be used in the evaluation.
- To see this in more detail, consider the postfix expression 4 5 6 * +. As you scan the expression from left to right, you first encounter the operands 4 and 5. At this point, you are still unsure what to do with them until you see the next symbol. Placing each on the stack ensures that they are available if an operator comes next.

Postfix Evaluation (II)

- In this case, the next symbol is another operand. So, as before, push it and check the next symbol. Now we see an operator, *. This means that the two most recent operands need to be used in a multiplication operation. By popping the stack twice, we can get the proper operands and then perform the multiplication (in this case getting the result 30).
- We can now handle this result by placing it back on the stack so that it can be used as an operand for the later operators in the expression. When the final operator is processed, there will be only one value left on the stack. Pop and return it as the result of the expression. Figure next page shows the stack contents as this entire example expression is being processed.

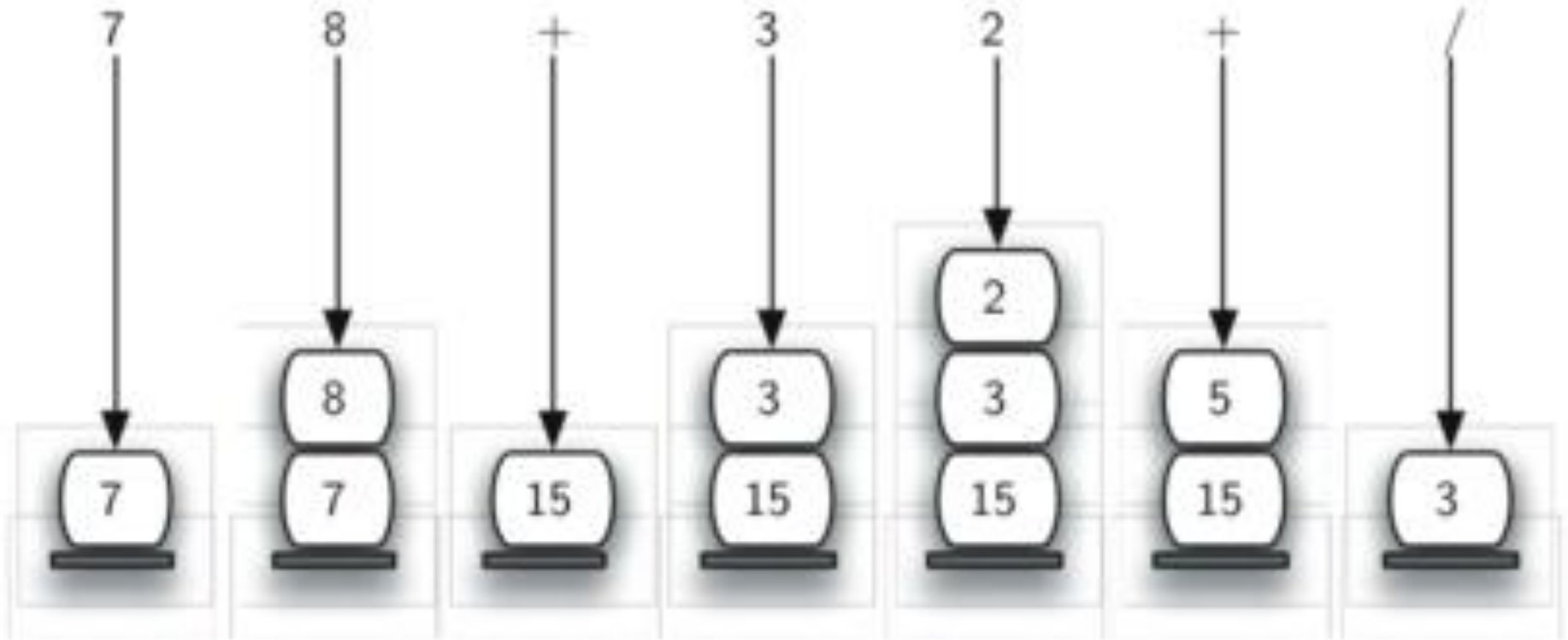


Postfix Evaluation (III)

Postfix Evaluation (IV)

- Figure 11 shows a slightly more complex example, 7 8 + 3 2 + /.
- There are two things to note in this example.
  - First, the stack size grows, shrinks, and then grows again as the subexpressions are evaluated.
  - Second, the division operation needs to be handled carefully.
- Recall that the operands in the postfix expression are in their original order since postfix changes only the placement of operators.
- When the operands for the division are popped from the stack, they are reversed. Since division is not a commutative operator, in other words 15/5 is not the same as 5/15, we must be sure that the order of the operands is not switched.

Postfix Evaluation (V)



Algorithm for Evaluation of Expressions

Assume the postfix expression is a string of tokens delimited by spaces. The operators are *, /, +, and - and the operands are assumed to be single-digit integer values. The output will be an integer result.

1. Create an empty stack called operandStack.
2. Convert the string to a list by using the string method split.
3. Scan the token list from left to right.
   - If the token is an operand, convert it from a string to an integer and push the value onto the operandStack.
   - If the token is an operator, *, /, +, or -, it will need two operands. Pop the operandStack twice. The first pop is the second operand and the second pop is the first operand. Perform the arithmetic operation. Push the result back on the operandStack.
4. When the input expression has been completely processed, the result is on the stack. Pop the operandStack and return the value.

Demo Program: PostfixEval.java

- The complete function for the evaluation of postfix expressions is shown in PostfixEval.java.
- To assist with the arithmetic, a helper function doMath is defined that will take two operands and an operator and then perform the proper arithmetic operation.

```
import java.util.*;
public class PostfixEval
{   public static double postfixEval(String expr){
        Stack<String> operandStack = new Stack<String>();
        String[] tokens = expr.split(" ");
        for (String t: tokens){
            t = t.trim();
            if (t.length()==0) continue;
            if (t.length()>0){
               boolean isNumber = false;
               try{
                    double d = Double.parseDouble(t);
                    isNumber = true;
                }
               catch(Exception e){}
               if (isNumber){
                    operandStack.push(t);
                }
               else{
                   String op2 = operandStack.pop();
                   String op1 = operandStack.pop();
                   double result = doMath(t, op1, op2);
                   operandStack.push(""+result);
                }
            }
        }
        return Double.parseDouble(operandStack.pop());
    }
```

```
   public static double doMath(String op, String op1, String op2){
       double x = Double.parseDouble(op1);
       double y = Double.parseDouble(op2);

       switch(op){
          case "*": return x*y;
          case "/": return x/y;
          case "+": return x+y;
          case "-": return x-y;
        }
       return 0;
    }
    public static void main(String[] args){
       System.out.print("\f");
       System.out.println(postfixEval("1 2 +"));
    }
}

[Running] cd "c:\Eric_Chou\USACO\CS91 Bronze\BlueJ\U7 Stack\" && javac PostfixEval.java
&& java PostfixEval
3.0
```

Postfix to Infix Conversion

SECTION 13

Conversion of a Postfix Expression into an Infix Expression

Step 1: Read the postfix expression from left to right .

Step 2: If the symbol is an operand, then insert it onto the front of a queue.

Step 3: If the symbol is an operator,
pop two symbols from the front of the queue and create it as a string by placing the operator in between the operands and insert the resulted string to the front of the queue.

Step 4: Repeat steps 2 and 3 till the end of the postfix expression.

Step 5: EXIT

Example

•Consider the following Prefix Expression...

A B - C D * +

- The above prefix expression can be converted into infix expression using Stack data Structure as follows...

•The equivalent infix Expression is as follows...

((A - B) + (C * D))

Reading Character	Infix Expression
A	A
B	B A
-	(A-B)
C	C (A-B)
D	D C (A-B)
*	(C*D) (A-B)
+	(A-B)+(C*D)

```java
import java.util.*;
public class PostfixToInfix {
    public static boolean isOperand(char c) {
        return Character.isLetter(c);
    }
    public static String postfixToInfix(String postfix) {
        ArrayList<String> s = new ArrayList<String>();
        for (int i = 0; i < postfix.length(); i++) {
            if (isOperand(postfix.charAt(i))) {
                s.add(0, "" + postfix.charAt(i));
            } else {
                String op1 = s.get(0);
                s.remove(0);
                String op2 = s.get(0);
                s.remove(0);
                s.add(0, "(" + op2 + postfix.charAt(i) + op1 + ")");
            }
        }
        return s.get(0);
    }
    public static void main(String[] args) {
        // System.out.print("\f");
        System.out.println(postfixToInfix("ab-c+de*-"));
    }
}
```

```
[Running] cd "c:\Eric_Chou\USACO\CS91 Bronze\BlueJ\U7 Stack\" && javac
PostfixToInfix.java && java PostfixToInfix
(((a-b)+c)-(d*e))
```

Prefix to Infix Conversion

SECTION 14

Conversion of a Prefix Expression into an Infix Expression

Step 1: Read the prefix expression from right to left .

Step 2: If the symbol is an operand, then push it onto the stack.

Step 3: If the symbol is an operator,

pop two symbols from the stack and create it as a string by placing the operator in between the operands and push the resulted string back to stack.

Step 4: Repeat steps 2 and 3 till the end of the prefix expression.

Step 5: EXIT

Example

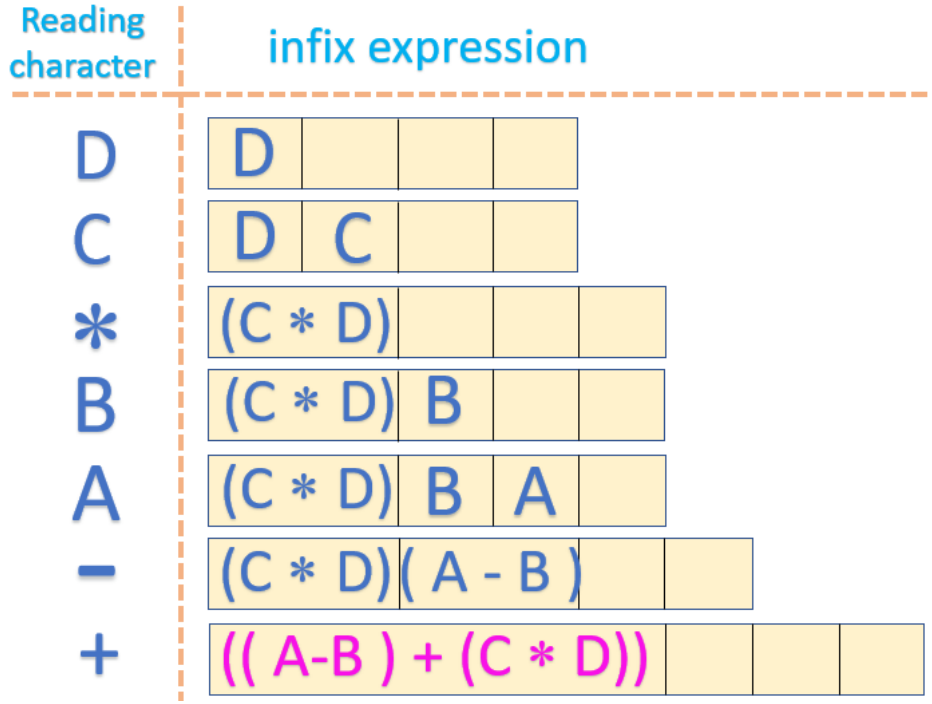
- Consider the following Prefix Expression...

  + - A B * C D

- The above prefix expression can be converted into infix expression using Stack data Structure as follows...

- The equivalent infix Expression is as follows...

  ((A - B) + (C * D))



```java
import java.util.*;
public class PrefixToInfix {
    public static boolean isOperator(char c) {
        return "*+-/!()".contains("" + c);
    }
    public static String prefixToInfix(String prefix) {
        Stack<String> s = new Stack<String>();
        int i = prefix.length() - 1;
        while (i >= 0) {
            if (!isOperator(prefix.charAt(i))) {
                s.push("" + prefix.charAt(i));
                i--;
            } else {
                String str = "(" + s.pop() + prefix.charAt(i) + s.pop() + ")";
                s.push(str);
                i--;
            }

        }
        return s.pop();
    }
    public static void main(String[] args) {
        System.out.println(prefixToInfix("*-A/BC-/AKL"));
    }
}
```

```
[Running] cd "c:\Eric_Chou\USACO\CS91 Bronze\BlueJ\U7 Stack\" && javac
PrefixToInfix.java && java PrefixToInfix
((A-(B/C))*((A/K)-L))
```

Prefix to Postfix Conversion

SECTION 15

Conversion of a Prefix Expression into an Postfix Expression

Step 1: Read the prefix expression from right to left .

Step 2: If the symbol is an operand, then push it onto the stack.

Step 3: If the symbol is an operator,
pop two symbols from the stack and create it as a string by placing the operator after the operands and push the resulted string back to stack.

Step 4: Repeat steps 2 and 3 till the end of the prefix expression.

Step 5: EXIT

Example

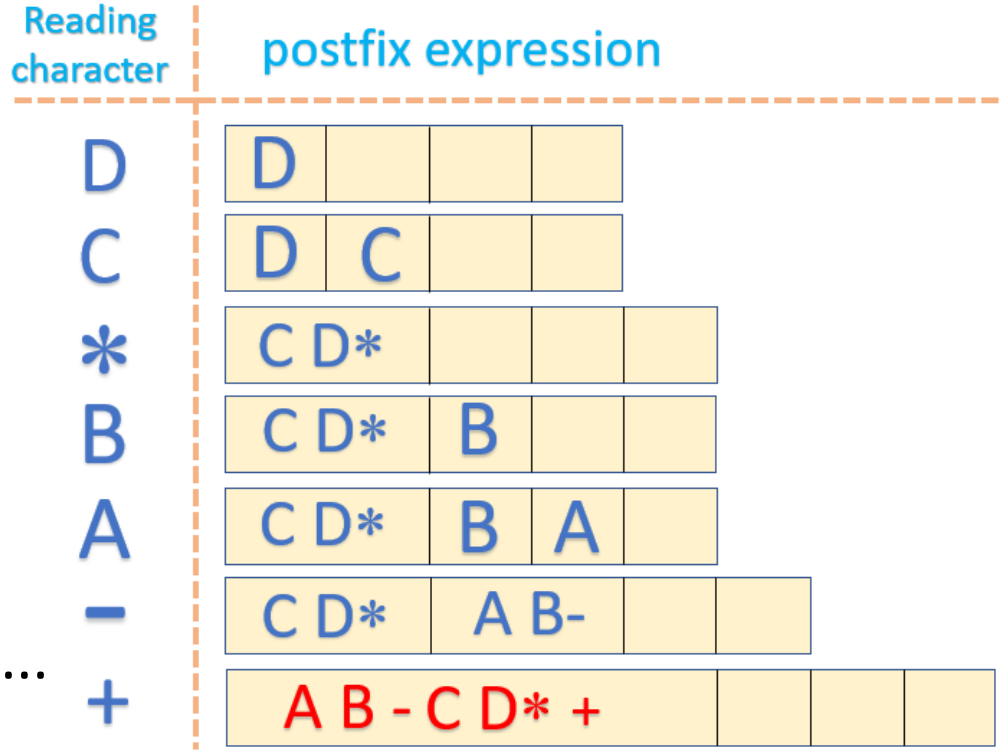
- Consider the following Prefix Expression...

+ - A B * C D

- The above prefix expression can be converted into postfix expression using Stack data Structure as follows...

- The equivalent postfix Expression is as follows...

A B - C D * +



PrefixToPostfix.py

```
def prefixToPostfix(prefix):
    # Reversing the order
    s = prefix[::-1]
    stack = []

    for i in s:
        if i in "+-*/^":
            a = stack.pop()
            b = stack.pop()
            temp = a+b+i
            stack.append(temp)
        else:
            stack.append(i)

    p = "".join(stack)
    return p

print(prefixToPostfix("*-A/BC-/AKL"))
```

ABC/-AK/L-*

Summary

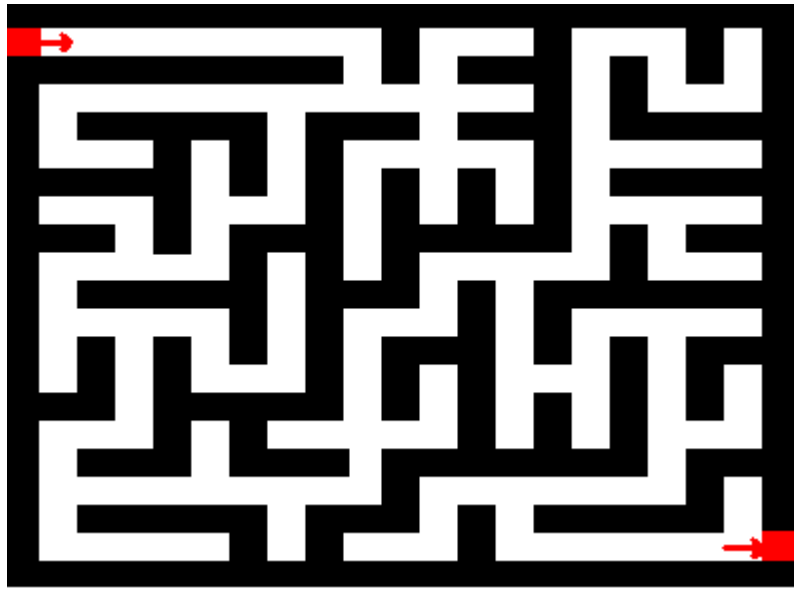
SECTION 16

Summary

- It is important to note that in both the postfix conversion and the postfix evaluation programs we assumed that there were no errors in the input expression.
- Using these programs as a starting point, you can easily see how error detection and reporting can be included. We leave this as an exercise at the end of the chapter.

Applications

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.



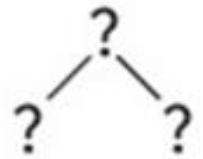
Backtracking. This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit? Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

- Language processing:
  - space for parameters and local variables is created internally using a stack.
  - compiler's syntax check for matching braces is implemented by using stack.
  - support for recursion

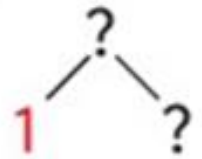
Syntax Tree, Stack, and Expression Evaluation

- In this lecture, we briefly go over functional programming and its relationship with recursion.
- Then, we work on the tree construction and various type of trees.
- After that, we talk about parsing tree.
- From an expression to a evaluation tree using stack and the different expression representation conversion and evaluation.
- More detailed syntax tree evaluation will be covered in the parsing and the interpreter design chapters.

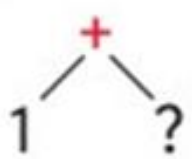
0 1 + 3 + 5 + 7



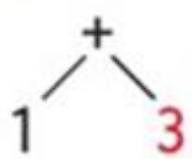
1 + 3 + 5 + 7



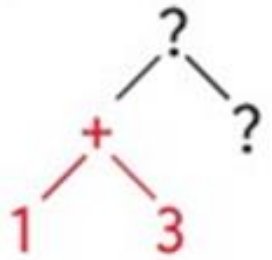
2 3 + 5 + 7



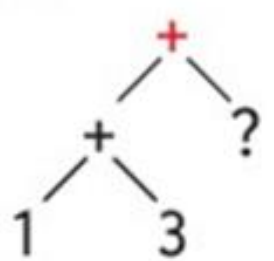
3 + 5 + 7



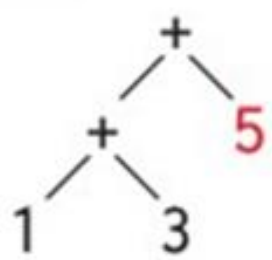
4 + 5 + 7



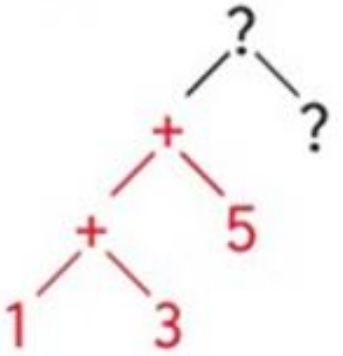
5 5 + 7



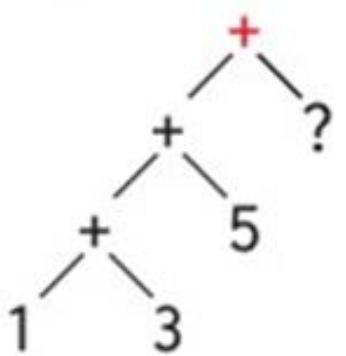
6 + 7



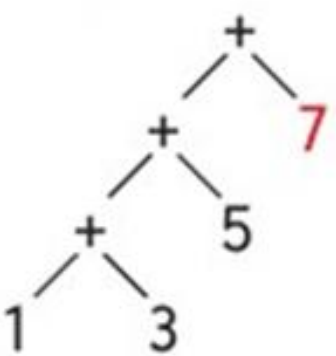
7 + 7

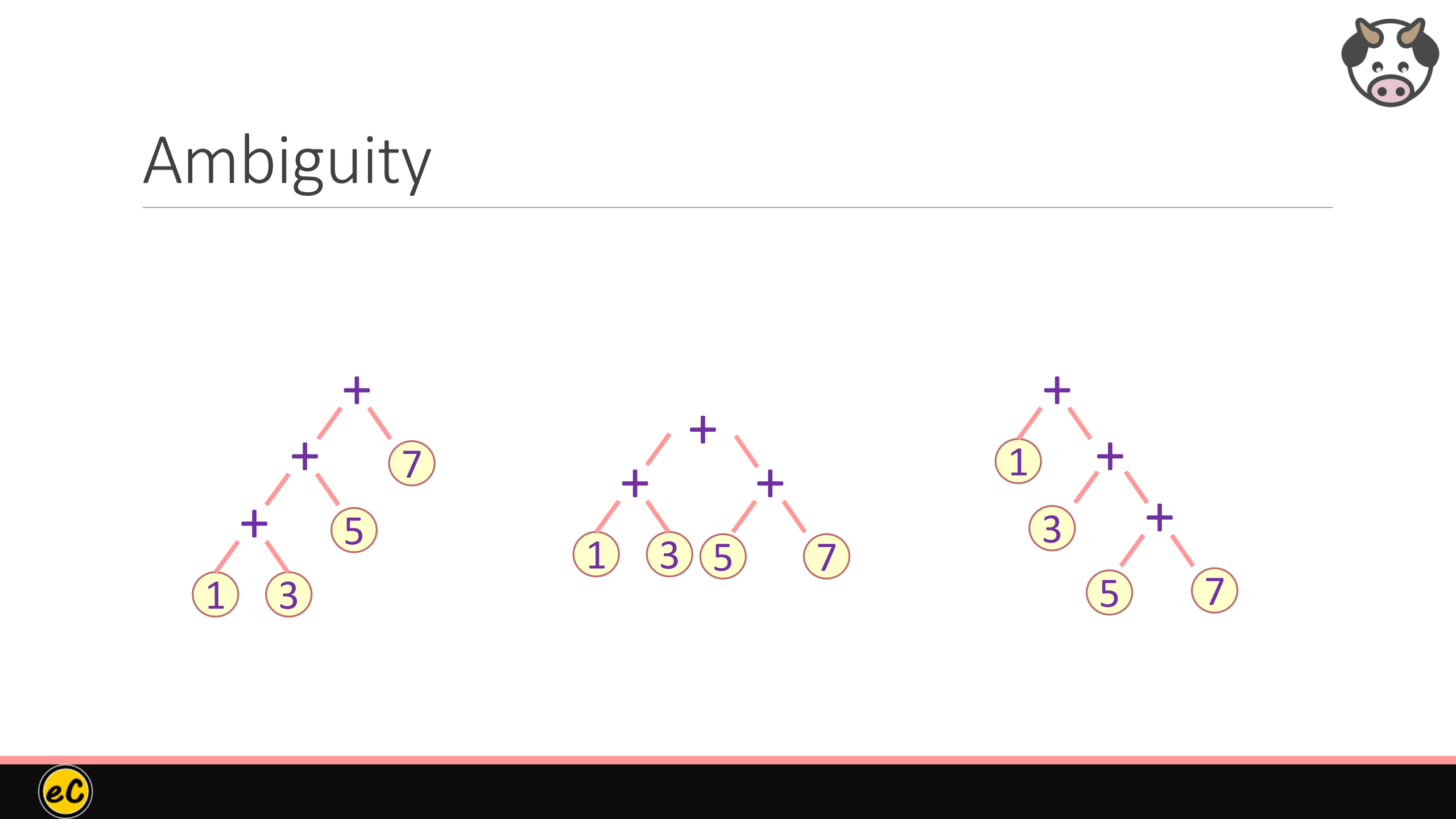


8 7



9





Ambiguity