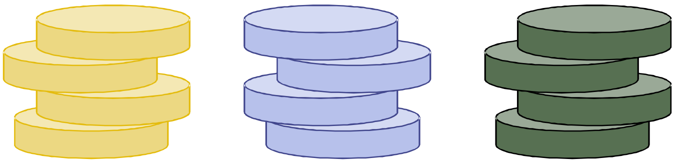
Stacks



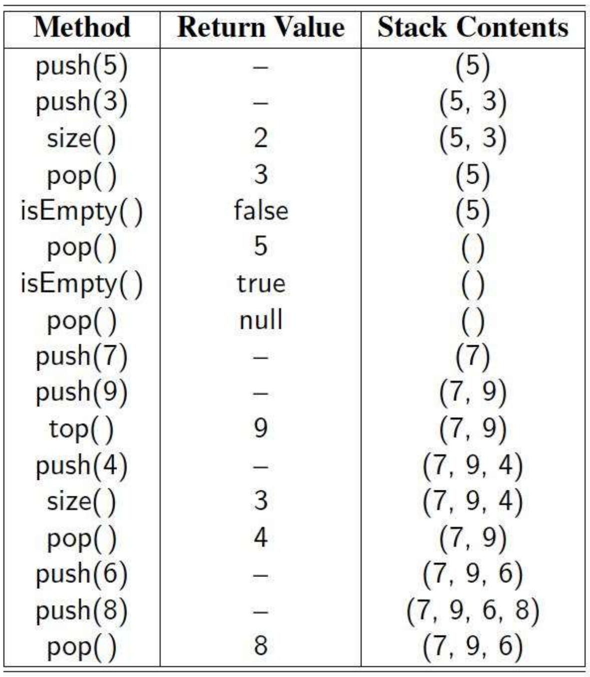
The Stack ADT(Abstract Data Type)

- A Stack is an ordered collection of homogeneous elements, in which all insertions and deletions are made at one end of the list called the "**top**" of the stack
- A stack has a **LIFO** ""last in, first out"" structure
- Think of a spring-loaded plate dispenser

Stack ADT (cont.)

- Main stack operations:
  - push(item): inserts an element
  - item pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - item 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

Example



Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size()	2	(5, 3)
pop()	3	(5)
isEmpty()	false	(5)
pop()	5	()
isEmpty()	true	()
pop()	null	()
push(7)	–	(7)
push(9)	–	(7, 9)
top()	9	(7, 9)
push(4)	–	(7, 9, 4)
size()	3	(7, 9, 4)
pop()	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop()	8	(7, 9, 6)

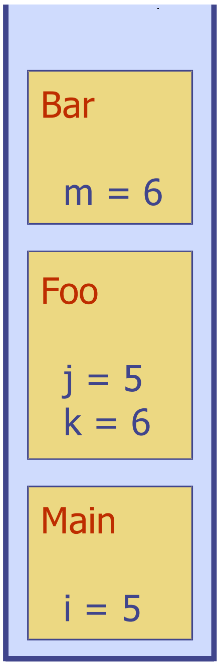
Applications of Stacks

- 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 stack frame (or activation record) for the called method
- 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) {
  …
  }
```



Stack Interface in Java

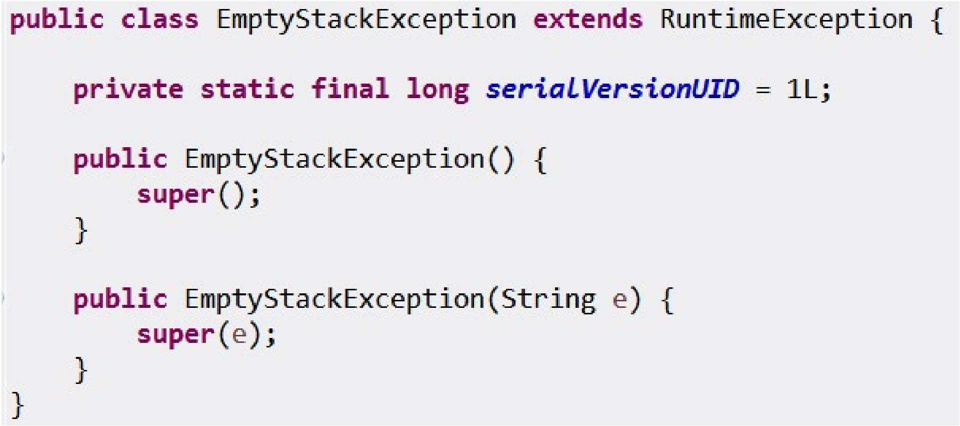
- Java interface corresponding to our Stack ADT
- Requires the definition of class EmptyStackException
- Different from the built-in Java class java.util.Stack

```
public interface Stack<E> {

   public int size();

   public boolean isEmpty();

   public E top()
         throws EmptyStackException;

   public void push(E element);

   public E 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

EmptyStackException

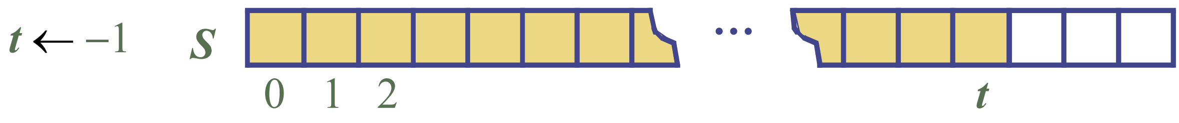


```
public class EmptyStackException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public EmptyStackException() {
        super();
    }

    public EmptyStackException(String e) {
        super(e);
    }
}
```

Array-based Stack

- 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

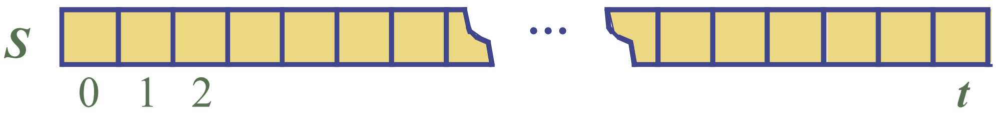
```
Algorithm size()
  return $t + 1$

Algorithm pop()
  if isEmpty() then
    throw EmptyStackException
  else
    $t \leftarrow t - 1$
    return $S[t + 1]$
```



Array-based Stack (cont.)

- 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 $t = S.length - 1$ then
    throw FullStackException
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$
```



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

Array-based Stack in Java

```
public class ArrayStack<E>
      implements Stack<E> {

   // holds the stack elements
   private E S[ ];

   // index to top element
   private int top = -1;

   // constructor
   public ArrayStack(int capacity) {
       S = (E[]) new Object[capacity]);
    }
```

```
    public E pop()
          throws EmptyStackException {
      if isEmpty()
        throw new EmptyStackException
              ("Empty stack: cannot pop");
        E temp = S[top];
        // facilitate garbage collection:
        S[top] = null;
        top = top – 1;
        return temp;
     }
…  (other methods of Stack interface)
```

Example use in Java

```
public class Tester {

   // … other methods
   public intReverse(Integer a[]) {
       Stack<Integer> s;
       s = new ArrayStack<Integer>();

       … (code to reverse array a) …
    }
```

```
    public floatReverse(Float f[]) {
        Stack<Float> s;
        s = new ArrayStack<Float>();

        … (code to reverse array f) …
     }
```

Linked-Based Implmentation

- In this section we study a link-based implementation of the Stack ADT.
- To support this we first define a **LLObjectNode** class
- After discussing the link-based approach we compare our stack implementation approaches.

The Node class

- Our stacks hold elements of type E.

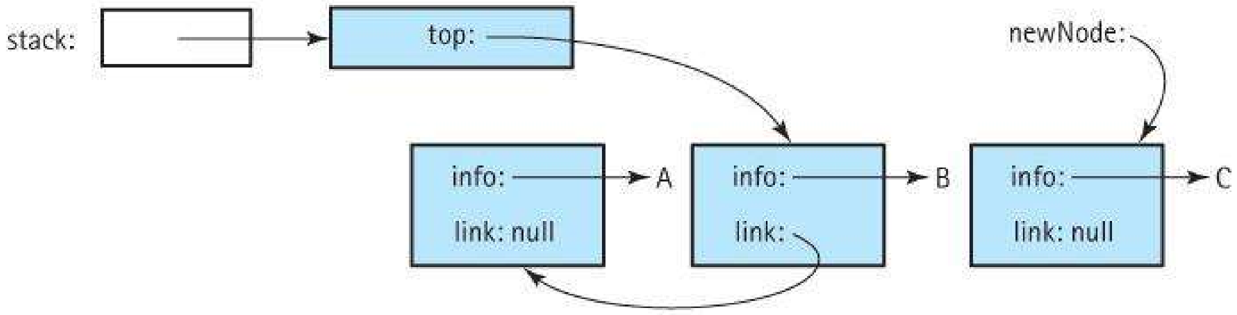
```
class Node<E> {
  private Node<E> link;
  private E info;

  Node(E info) {…}

  void setInfo(E info) {…}
  E     getInfo() {…}

  void    setLink(Node<E> link){…}
  Node<E> getLink() {…}
}
```

The LinkedStack Class

```
public class LinkedStack<E> implements Stack<E>
{
  // reference to the top of this stack
  protected LLNode<E> top;

  public LinkedStack()
  {
    top = null;
  }
. . .
```

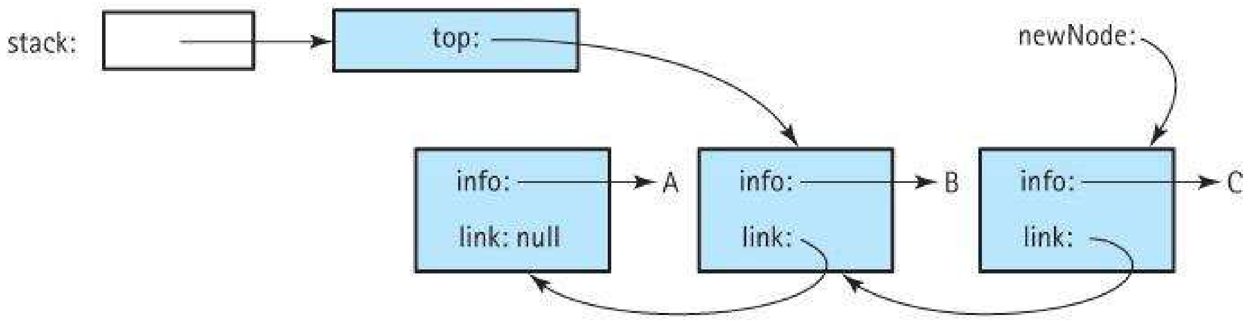
The push(C) operation (step 1)

- **Allocate space for the next stack node and set the node info to element**
- Set the node link to the previous top of stack
- Set the top of stack to the new stack node



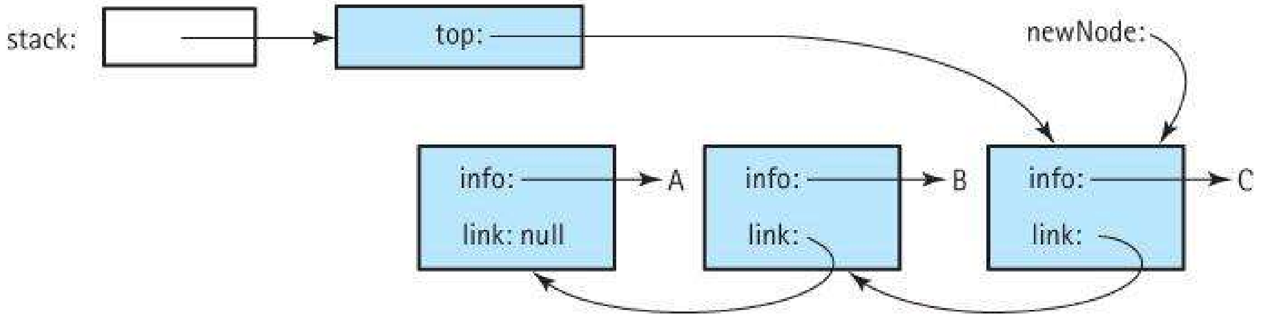
The push(C) operation (step 2)

- Allocate space for the next stack node
  and set the node info to element
- **Set the node link to the previous top of stack**
- Set the top of stack to the new stack node



The push(C) operation (step 3)

- Allocate space for the next stack node
  and set the node info to element
- Set the node link to the previous top of stack
- **Set the top of stack to the new stack node**



Code for the push method

```
public void push(E element)
// Places element at the top of this stack.
{
  LLNode<E> newNode = new LLNode<T>(element);
  newNode.setLink(top);
  top = newNode;
}
```

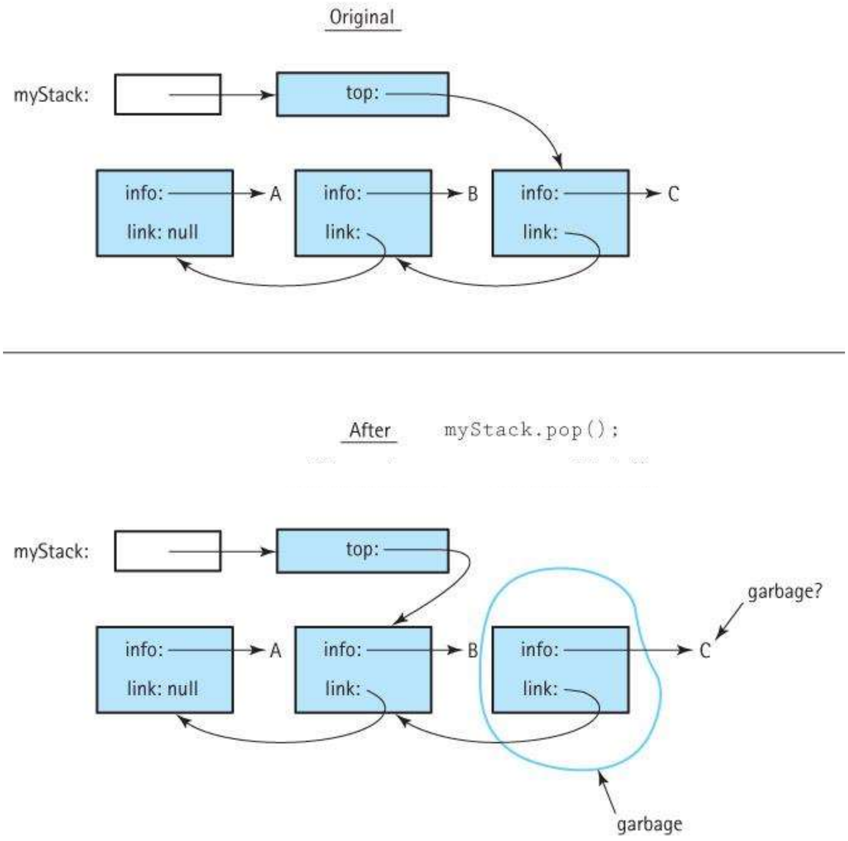
Code for the pop method

```
public E pop()
// Throws EmptyStackException if this stack is empty,
// otherwise removes top element from this stack.
{
  E temp;
  if (!isEmpty())
  {
    temp = top.getInfo();
    top = top.getLink();
  }
  else
    throw new EmptyStackException(
                    "Pop attempted on an empty stack.");
  return temp;
}
```

(Better)
Code for the pop method

```
public E pop()
// Throws EmptyStackException if this stack is empty,
// otherwise removes top element from this stack.
{
  if (isEmpty())
      throw new EmptyStackException(
           "Pop attempted on an empty stack.");

  E temp = top.getInfo();
  top = top.getLink();

  return temp;
}
```

Pop from a stack with three elements



The remaining operations

```
public E top()
// Throws EmptyStackException if this stack is empty,
// otherwise returns top element from this stack.
{
  if (!isEmpty())
    return top.getInfo();
  else
    throw new StackUnderflowException(
                       "Top attempted on an empty stack.");
}

public boolean isEmpty()
// Returns true if this stack is empty, otherwise returns false.
{
  if (top == null)
    return true;
  else
    return false;
}
```

Comparing Stack Implementations

- Storage Size
  - Array-based: takes the same amount of memory, no matter how many array slots are actually used, proportional to maximum size
  - Link-based: takes space proportional to actual size of the stack (but each element requires more space than with array approach)

- Operation efficiency
  - All operations, for each approach, are O(1)
  - Except for the Constructors:
    - Array-based: O(N)
    - Link-based: O(1)

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 to n-1 do
    if X[i] is an opening grouping symbol then
          S.push(X[i])
    else if X[i] is a closing grouping symbol then
          if S.isEmpty() then
                    return false {nothing to match with}
          if S.pop() does not match the type of X[i] then
                    return false {wrong type}
if S.isEmpty() then
    return true {every symbol matched}
else return false {some symbols were never matched}
```

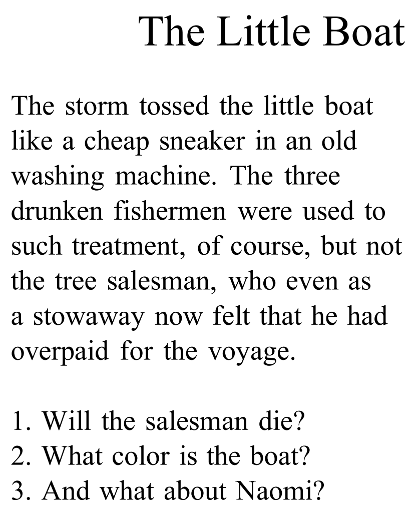
Parenthesis Matching (Java)

```
public static boolean isMatched(String expression) {
  final String opening = "({["; // opening delimiters
  final String closing = ")}]"; // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>( );
  for (char c : expression.toCharArray( )) {
    if (opening.indexOf(c) != −1) // this is a left delimiter
      buffer.push(c);
   else if (closing.indexOf(c) != −1) { // this is a right delimiter
     if (buffer.isEmpty( )) // nothing to match with
       return false;
     if (closing.indexOf(c) != opening.indexOf(buffer.pop( )))
       return false; // mismatched delimiter
     }
   }
  return buffer.isEmpty( ); // were all opening delimiters 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?

HTML Tag Matching (Java)

```
public static boolean isHTMLMatched(String html) {
  Stack<String> buffer =  new LinkedStack<>( );
  int j = html.indexOf('<'); // find first ’<’ character (if any)
  while (j != −1) {
    int k = html.indexOf('>', j+1); // find next ’>’ character
    if (k == −1)
      return false; // invalid tag
    String tag = html.substring(j+1, k); // strip away < >
    if (!tag.startsWith("/")) // this is an opening tag
      buffer.push(tag);
    else { // this is a closing tag
      if (buffer.isEmpty( ))
        return false; // no tag to match
      if (!tag.substring(1).equals(buffer.pop( )))
        return false; // mismatched tag
    }
    j = html.indexOf('<', k+1); // find next ’<’ character (if any)
  }
  return buffer.isEmpty( ); // were all opening tags matched?
}
```

Evaluating Arithmetic Expressions

14 – 3 * 2 + 7 = (14 – (3 * 2)) + 7

Operator precedence

* has precedence over +/–

Associativity

operators of the same precedence group
evaluated from left to right
Example: (x – y) + z rather than x – (y + z)

Idea: push each operator on the stack, but first pop and perform higher and *equal* precedence operations.

Algorithm for Evaluating Expressions

Two stacks:

- opStk holds operators
- valStk holds values
- Use $ as special "end of input" token with lowest precedence

```
Algorithm doOp()
    x ← valStk.pop();
    y ← valStk.pop();
    op ← opStk.pop();
    valStk.push( y op x )
```

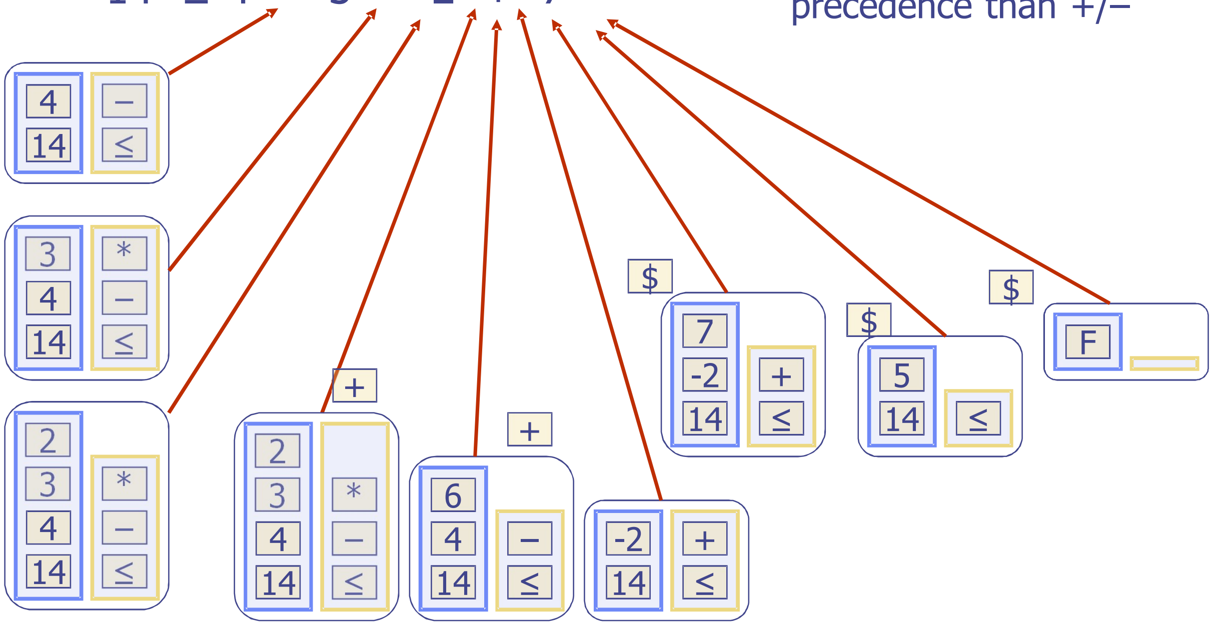
```
Algorithm repeatOps( refOp ):
  while ( valStk.size() > 1 ∧
          prec(refOp) ≤
          prec(opStk.top())
    doOp()
```

```
Algorithm EvalExp()
    Input: a stream of tokens representing
        an arithmetic expression (with
        numbers)
    Output: the value of the expression

while there's another token z
    if isNumber(z) then
        valStk.push(z)
    else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

Algorithm on an Example Expression

14 ≤ 4 – 3 * 2 + 7

Operator ≤ has lower precedence than +/–


4
14
–
≤
3
4
14
*
–
≤
2
3
4
14
*
–
≤
+
2
3
4
14
*
–
≤
+
6
4
14
–
≤
-2
14
+
≤
$
7
-2
14
+
≤
$
5
14
≤
$
F