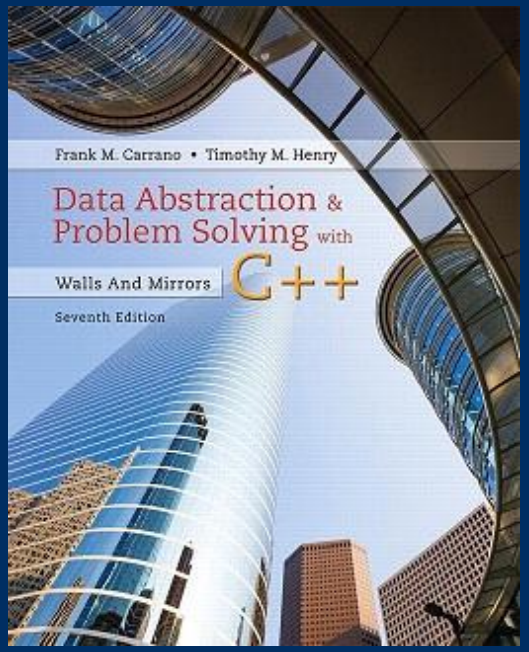
Chapter 6
Stacks

CS 302 - Data Structures

M. Abdullah Canbaz



Modified from authors' slides

Reminders

- Assignment 2 is available
  - Due Feb 14th at 2pm

- TA
  - Shehryar Khattak,
    Email: *shehryar [at] nevada {dot} unr {dot} edu*,
    Office Hours: Friday, 11:00 am - 1:00 pm at ARF 116

- Quiz 3 on Wednesday

“Manners maketh man.”

No attendance in this class!

Respect to your peers!

The Abstract Data Type Stack

- Operations on a stack
  - Last-in,
  - First-out behavior.
- Applications demonstrated
  - Evaluating algebraic expressions
  - Searching for a path between two points

Developing an ADT During the Design of a Solution

- Consider typing a line of text on a keyboard
  - Use of backspace key to make corrections
  - You type

    abcc←ddde←←←eg←fg

  - Corrected input will be

    abcdefg

- Must decide how to store the input line.

Developing an ADT During the Design of a Solution

```
// Read the line, correcting mistakes along the way
while (not end of line)
{
    Read a new character ch
    if (ch is not a '←')
        Add ch to the ADT
    else
        Remove from the ADT (and discard) the item that was added most recently
}
```

- Initial draft of solution
- Two required operations
  - Add new item to ADT
  - Remove item added most recently

Developing an ADT During the Design of a Solution

```
// Read the line, correcting mistakes along the way
while (not end of line)
{
    Read a new character ch
    if (ch is not a '←')
        Add ch to the ADT
    else if (the ADT is not empty)
        Remove from the ADT and discard the item that was added most recently
    else
        Ignore the '←'
}
```

- Read and correct algorithm
- Third operation required
  - See whether ADT is empty

Developing an ADT During the Design of a Solution

```
// Display the line in reverse order
while (the ADT is not empty)
{
    Get a copy of the item that was added to the ADT most recently and assign it to ch
    Display ch
    Remove from the ADT and discard the item that was added most recently
}
```

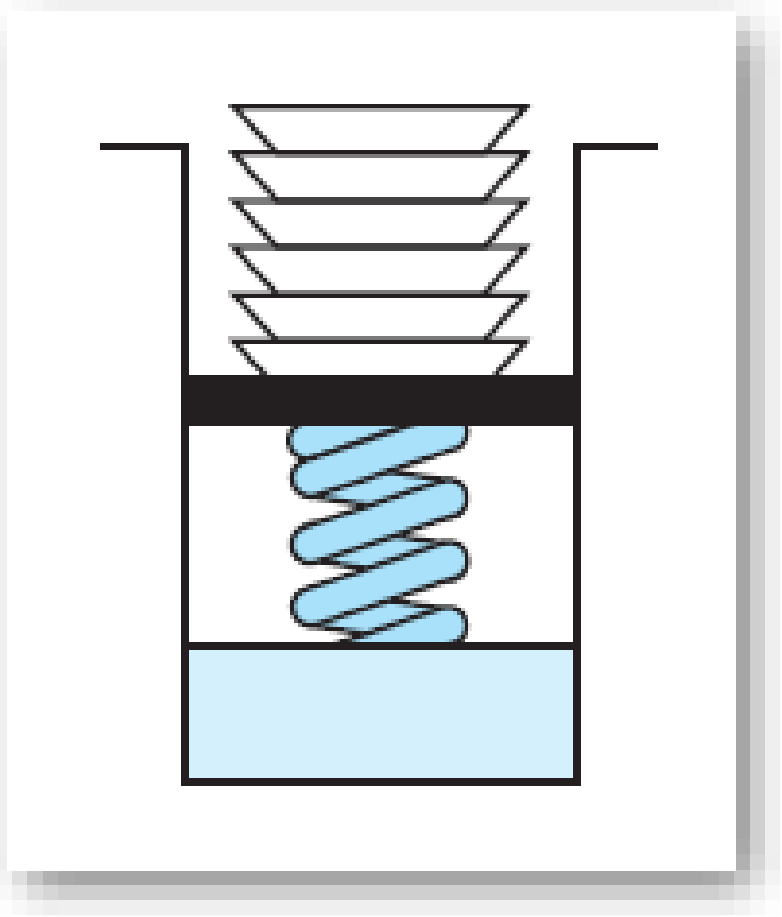
- Write-backward algorithm
- Fourth operation required
  - Get item that was added to ADT most recently.

Specifications for the ADT Stack

- See whether stack is empty.
- Add new item to the stack.
- Remove from and discard stack item that was added most recently.
- Get copy of item that was added to stack most recently.

Specifications for the ADT Stack

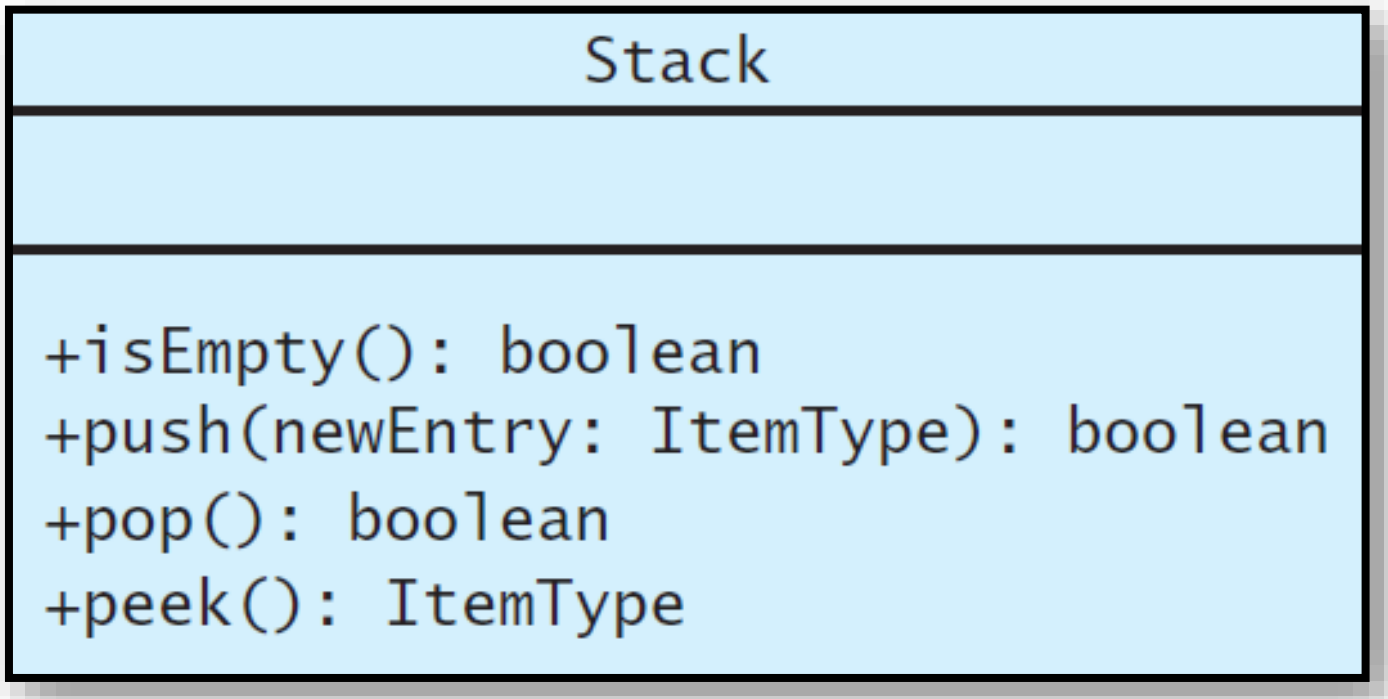
- A stack of cafeteria plates



LIFO: The last item inserted onto a stack is the first item out

Specifications for the ADT Stack

- UML diagram for the class Stack



Specifications for the ADT Stack

- A C++ interface for stacks

```
1  /** @file StackInterface.h */
2  #ifndef STACK_INTERFACE_
3  #define STACK_INTERFACE_
4
5  template<class ItemType>
6  class StackInterface
7  {
8  public:
9     /** Sees whether this stack is empty.
10     @return  True if the stack is empty, or false if not. */
11    virtual bool isEmpty() const = 0;
12
13    /** Adds a new entry to the top of this stack.
14     @post  If the operation was successful, newEntry is at the top of the stack.
15     @param newEntry  The object to be added as a new entry.
16     @return  True if the addition is successful or false if not. */
17    virtual bool push(const ItemType& newEntry) = 0;
18
```

Specifications for the ADT Stack

```
18
19      /** Removes the top of this stack.
20       @post  If the operation was successful, the top of the stack
21          has been removed.
22       @return  True if the removal is successful or false if not. */
23      virtual bool pop() = 0;
24
25      /** Returns a copy of the top of this stack.
26       @pre  The stack is not empty.
27       @post  A copy of the top of the stack has been returned, and
28          the stack is unchanged.
29       @return  A copy of the top of the stack. */
30      virtual ItemType peek() const = 0;
31
32      /** Destroys this stack and frees its assigned memory. */
33      virtual ~StackInterface() { }
34  }; // end StackInterface
35  #endif
```

Specifications for the ADT Stack

- Axioms for multiplication

$$(a \times b) \times c = a \times (b \times c)$$
$$a \times b = b \times a$$
$$a \times 1 = a$$
$$a \times 0 = 0$$

- Axioms for ADT stack

```
(Stack()).isEmpty() = true
(Stack()).pop() = false
(Stack()).peek() = error
(aStack.push(item)).isEmpty() = false
(aStack.push(item)).peek() = item
(aStack.push(item)).pop() = true
(aStack.push(item)).pop() ⇒ aStack
```

Checking for Balanced Braces

- Example of curly braces in C++ language
  - Balanced

```
abc{defg{ijk}{l{mn}}op}qr
```

  - Not balanced

```
abc{def}}{ghij{kl}m
```

- Requirements for balanced braces
  - For each **}**, must match an already encountered **{**
  - At end of string, must have matched each **{**

Checking for Balanced Braces

- Initial draft of a solution.

```
for (each character in the string)
{
    if (the character is a '{')
        aStack.push('{')
    else if (the character is a '}')
        aStack.pop()
}
```

Checking for Balanced Braces

- Detailed pseudocode solution.

```
// Checks the string aString to verify that braces match.
// Returns true if aString contains matching braces, false otherwise.
checkBraces(aString: string): boolean
{
   aStack = a new empty stack
   balancedSoFar = true
   i = 0                              // Tracks character position in string

   while (balancedSoFar and i < length of aString)
   {
      ch = character at position i in aString
      i++

      // Push an open brace
      if (ch is a '{')
         aStack.push('{')

      // Close brace
      else if (ch is a '}')
      {
```

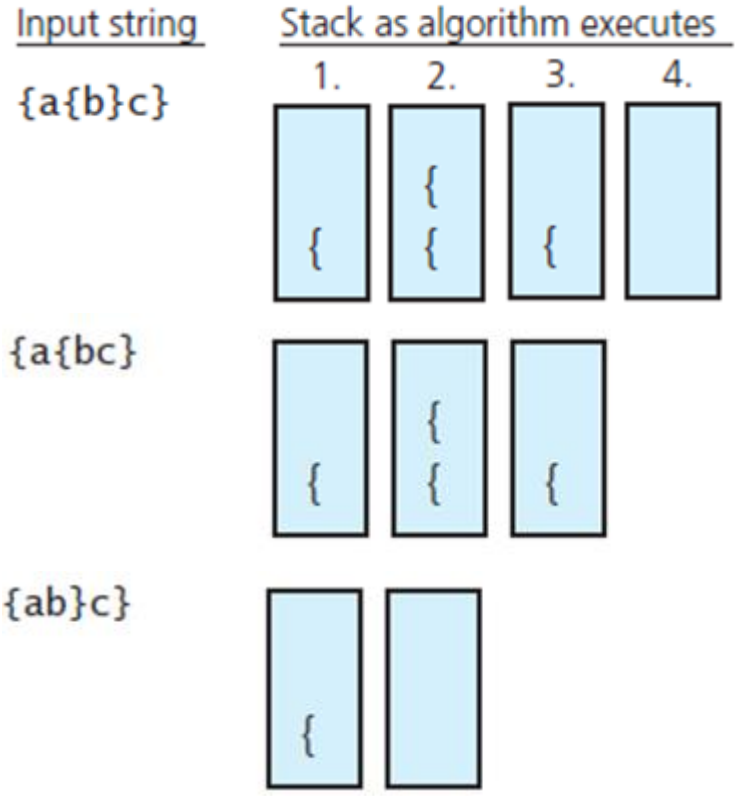
Checking for Balanced Braces

- Detailed pseudocode solution.

```
        // Close brace
        else if (ch is a '}')
        {
            if (!aStack.isEmpty())
                aStack.pop()    // Pop a matching open brace
            else                // No matching open brace
                balancedSoFar = false
        }
        // Ignore all characters other than braces
    }

    if (balancedSoFar and aStack.isEmpty())
        aString has balanced braces
    else
        aString does not have balanced braces
}
```

Checking for Balanced Braces

- Traces of algorithm that checks for balanced braces



1. push {
2. push {
3. pop
4. pop

stack empty -> balanced

1. push {
2. push {
3. pop

Stack not empty ⟹ not balanced

1. push {
2. pop

Stack empty when "}" encountered ⟹ not balanced

Recognizing Strings in a Language

- Given a definition of a language, *L*
  - Special palindromes
  - Special middle character $
  - Example ABC$CBA ε *L*, but AB$AB ∉ *L*
- A stack is useful in determining whether a given string is in a language
  - Traverse first half of string
  - Push each character onto stack
  - Reach $, undo, pop character, match or not

Recognizing Strings in a Language

- Algorithm to recognize string in language *L*

```
// Checks the string aString to verify that it is in language L.
// Returns true if aString is in L, false otherwise.
recognizeString(aString: string): boolean
{
   aStack = a new empty stack

   // Push the characters that are before the $ (that is, the characters in s) onto the stack
   i = 0                                  // Tracks character position in string
   ch = character at position i in aString
   while (ch is not a '$')
   {
      aStack.push(ch)
      i++
      ch = character at position i in aString
   }

   // Skip the $
   i++

   // Match the reverse of s
   inLanguage = true                      // Assume string is in language
   while (inLanguage and i < length of aString)
   {
```

Recognizing Strings in a Language

- Algorithm to recognize string in language *L*

```
inLanguage = true              // Assume string is in language
while (inLanguage and i < length of aString)
{
   if (!aStack.isEmpty())
   {
      stackTop = aStack.peek()
      aStack.pop()
      ch = character at position i in aString
      if (stackTop equals ch)
         i++                   // Characters match
      else
         inLanguage = false    // Characters do not match (top of stack is not ch)
   }
   else
      inLanguage = false       // Stack is empty (first half of string is shorter
                               // than second half)
}
if (inLanguage and aStack.isEmpty())
   aString is in language
else
   aString is not in language
}
```

Longest Palindromes

- The longest palindromic word
  - in the *Oxford English Dictionary* is ***tattarrattat***,
  - The *Guinness Book of Records* gives the title to ***detartrated***,
  - **Rotavator**, ***redivider*** ,***Malayalam****(* a language of southern India, is of equal length)
- A palindromic novel published: ***Satire: Veritas***
- the 224-word long poem "**Dammit I'm Mad**" by Demetri Martin.
- According to Guinness World Records, the Finnish 19-letter word ***saippuakivikauppias*** (a soapstone vendor), is the world's longest palindromic word in everyday use.

Using Stacks with Algebraic Expressions

- Strategy
  - Develop algorithm to evaluate postfix
  - Develop algorithm to transform infix to postfix
- These give us capability to evaluate infix expressions
  - This strategy easier than *directly* evaluating infix expression

Evaluating Postfix Expressions

- Infix expression 2 * (3 + 4)
- Equivalent postfix 2 3 4 + *
  - Operator in postfix applies to two operands immediately preceding

- Assumptions for our algorithm
  - Given string is correct postfix
  - No unary, no exponentiation operators
  - Operands are single lowercase letters, integers

Evaluating Postfix Expressions

- The effect of a postfix calculator on a stack when evaluating the expression 2 3 4 + *

Key entered	Calculator action		Stack (bottom to top):
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = peek	(4)	2 3 4
	pop		2 3
	operand1 = peek	(3)	2 3
	pop		2
	result = operand1 + operand2	(7)	
	push result		2 7
*	operand2 = peek	(7)	2 7
	pop		2
	operand1 = peek	(2)	2
	pop		
	result = operand1 * operand2	(14)	
	push result		14

Evaluating Postfix Expressions

- A pseudocode algorithm that evaluates postfix expressions

```
for (each character ch in the string)
{
    if (ch is an operand)
        Push the value of the operand ch onto the stack
    else // ch is an operator named op
    {
        // Evaluate and push the result
        operand2 = top of stack
        Pop the stack

        operand1 = top of stack
        Pop the stack

        result = operand1 op operand2
        Push result onto the stack
    }
}
```

Infix to Postfix

- Important facts
  - Operands always stay in same order with respect to one another.
  - Operator will move only “to the right” with respect to the operands;
    - If in the infix expression the operand *x* precedes the operator *op,*
    - Also true that in the postfix expression the operand *x* precedes the operator *op* .
  - All parentheses are removed.

Infix to Postfix

- First draft of algorithm to convert infix to postfix

```
Initialize postfixExp to the empty string
for (each character ch in the infix expression)
{
    switch (ch)
    {
        case ch is an operand:
            Append ch to the end of postfixExp
            break
        case ch is an operator:
            Save ch until you know where to place it
            break
        case ch is a '(' or a ')':
            Discard ch
            break
    }
}
```

Infix to Postfix

- Determining where to place operators in postfix expression
  - Parentheses
  - Operator precedence
  - Left-to-right association
- Note difficulty
  - Infix expression not always fully parenthesized
  - Precedence and left-to-right association also affect results

Infix to Postfix

ch	operatorStack (top to bottom)	postfixExp	
a		a	
–	–	a	
(	(–	a	
b	(–	a b	
+	+ (–	a b	
c	+ (–	a b c	
*	* + (–	a b c	
d	* + (–	a b c d	
)	+ (–	a b c d *	Move operators from stack to `postfixExp` until "("
	(–	a b c d * +	
	–	a b c d * +	
/	/ –	a b c d * +	
e	/ –	a b c d * + e	
	–	a b c d * + e /	Copy operators from stack to `postfixExp`
		a b c d * + e / –	

- A trace of the algorithm that converts the infix expression a – (b + c * d) /e to postfix form

Infix to Postfix

```
for (each character ch in the infix expression)
{
   switch (ch)
   {
      case operand:        // Append operand to end of postfix expression—step 1
         postfixExp = postfixExp • ch
         break
      case '(':            // Save '(' on stack—step 2
         operatorStack.push(ch)
         break
      case operator:       // Process stack operators of greater precedence—step 3
         while (!operatorStack.isEmpty() and operatorStack.peek() is not a '(' and
                precedence(ch) <= precedence(operatorStack.peek()))
         {
            Append operatorStack.peek() to the end of postfixExp
            operatorStack.pop()
         }
         operatorStack.push(ch) // Save the operator
         break
      case ')':                 // Pop stack until matching '('—step 4
```

- Pseudocode algorithm that converts infix to postfix

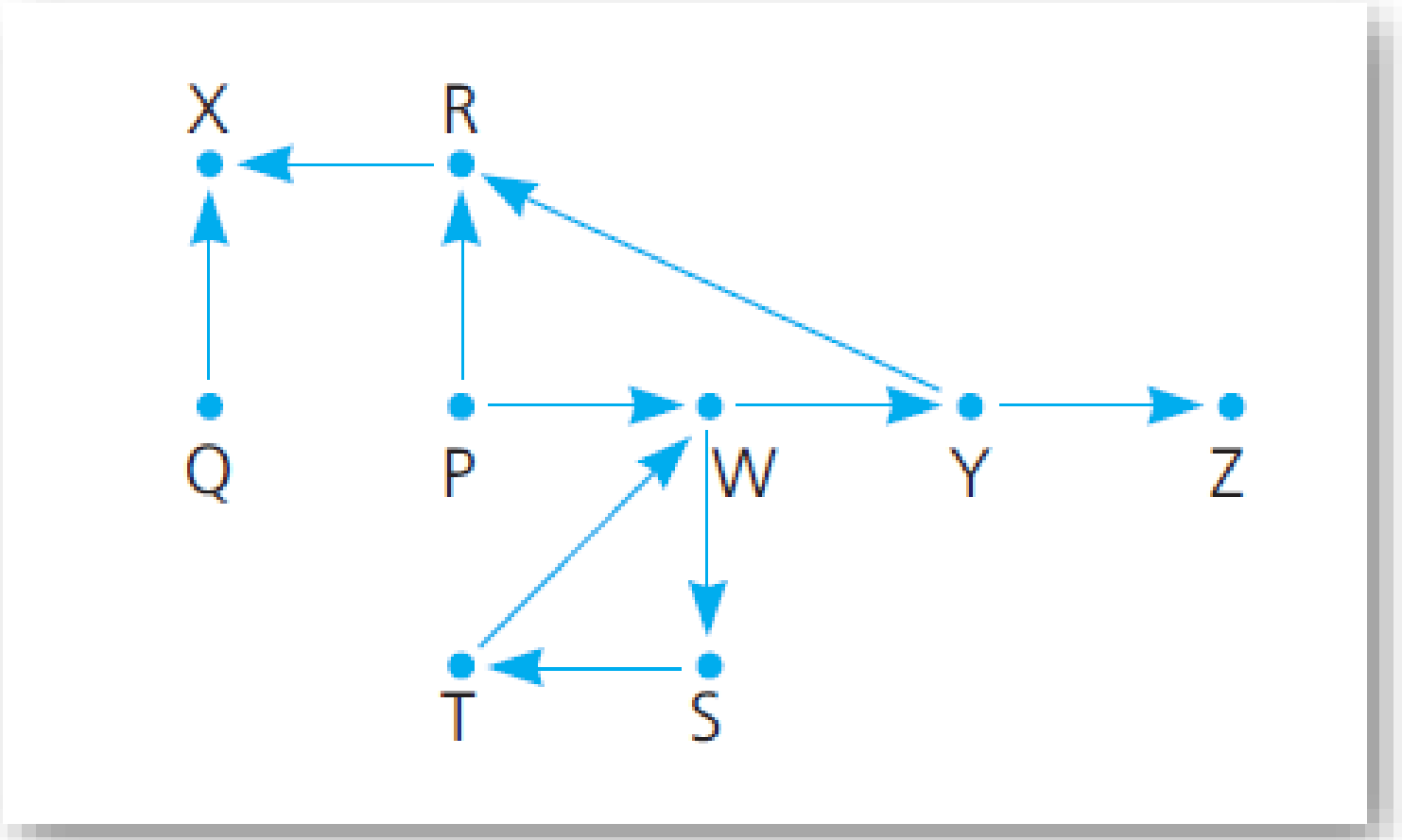
Infix to Postfix

```
            break
        case ')':                    // Pop stack until matching '('—step 4
            while (operatorStack.peek() is not a '(')
            {
                Append operatorStack.peek() to the end of postfixExp
                operatorStack.pop()
            }
            operatorStack.pop()      // Remove the open parenthesis
            break
    }
}
// Append to postfixExp the operators remaining in the stack—step 5
while (!operatorStack.isEmpty())
{
    Append operatorStack.peek() to the end of postfixExp
    operatorStack.pop()
}
```

- Pseudocode algorithm that converts infix to postfix

Using Stack to Search a Flight Map

- A flight map



Using Stack to Search a Flight Map

- Recall recursive search strategy.

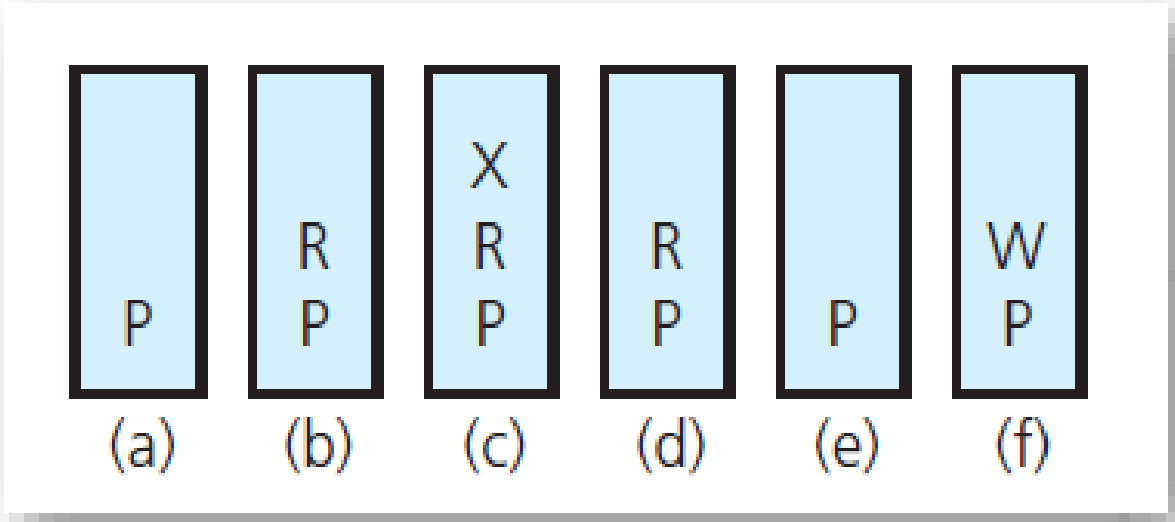
```
To fly from the origin to the destination
{
    Select a city C adjacent to the origin
    Fly from the origin to city C
    if (C is the destination city)
        Terminate— the destination is reached
    else
        Fly from city C to the destination
}
```

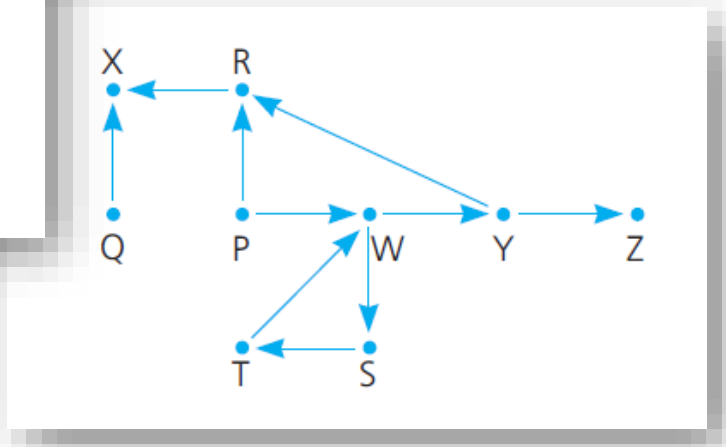
Using Stack to Search a Flight Map

- Possible outcomes of exhaustive search strategy
  1. Reach destination city, decide possible to fly from origin to destination
  2. Reach a city, *C* from which no departing flights
  3. You go around in circles
- Use backtracking to recover from a wrong choice (2 or 3)

Using Stack to Search a Flight Map

- Strategy requires information about order in which it visits cities





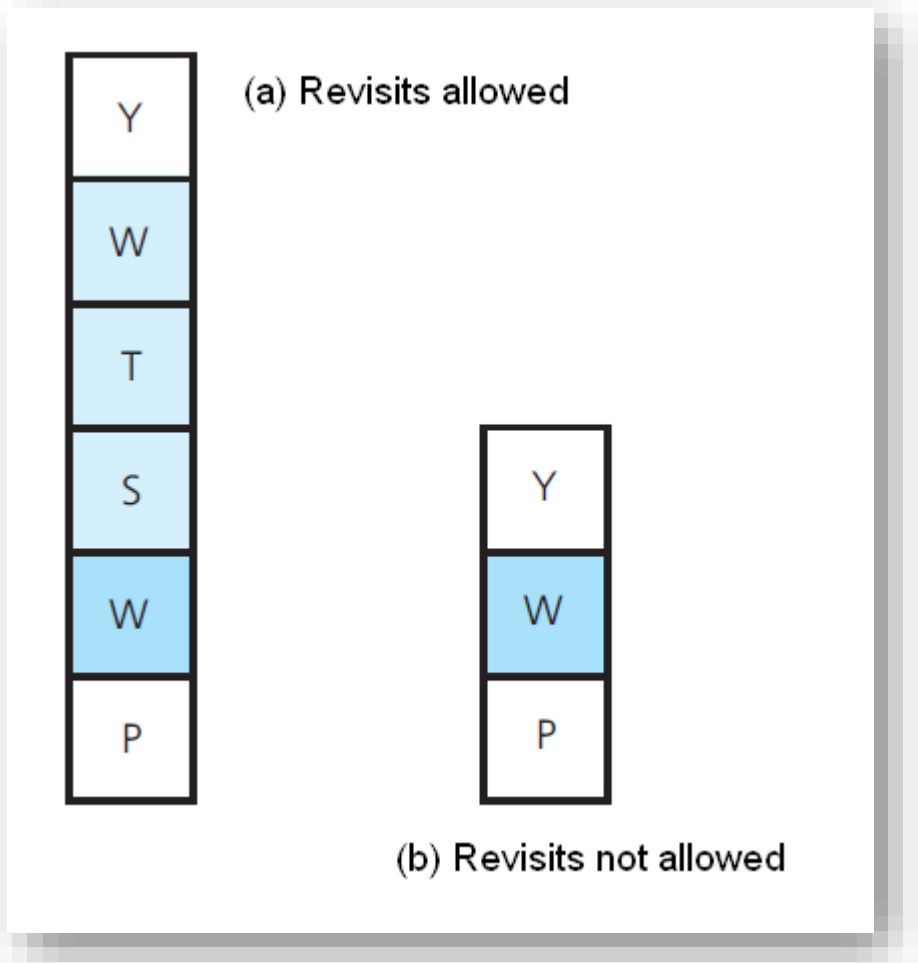
The stack of cities as you travel from P to W

Using Stack to Search a Flight Map

- Stack will contain directed path from
  - Origin city at bottom to …
  - Current visited city at top
- When to backtrack
  - No flights out of current city
  - Top of stack city already somewhere in the stack

Using Stack to Search a Flight Map

- The effect of revisits on the stack of cities



Using Stack to Search a Flight Map

- Final draft of algorithm.

```
// Searches for a sequence of flights from originCity to destinationCity
searchS(originCity: City, destinationCity: City): boolean
{
   cityStack = a new empty stack
   Clear marks on all cities

   cityStack.push(originCity) // Push origin onto the stack
   Mark the origin as visited

   while (!cityStack.isEmpty() and destinationCity is not at the top of the stack)
   {
      // Loop invariant: The stack contains a directed path from the origin city at
      // the bottom of the stack to the city at the top of the stack
      if (no flights exist from the city on the top of the stack to unvisited cities)
         cityStack.pop()          // Backtrack
      else
```

Using Stack to Search a Flight Map

- Final draft of algorithm.

```
            cityStack.pop()          // Backtrack
        else
        {
            Select an unvisited destination city C for a flight from the city on the top of the stack
            cityStack.push(C)
            Mark C as visited
        }
    }
    if (cityStack.isEmpty())
        return false // No path exists
    else
        return true  // Path exists
}
```

Using Stack to Search a Flight Map

Action	Reason	Contents of stack (bottom to top)
Push P	Initialize	P
Push R	Next unvisited adjacent city	P R
Push X	Next unvisited adjacent city	P R X
Pop X	No unvisited adjacent city	P R
Pop R	No unvisited adjacent city	P
Push W	Next unvisited adjacent city	P W
Push S	Next unvisited adjacent city	P W S
Push T	Next unvisited adjacent city	P W S T
Pop T	No unvisited adjacent city	P W S
Pop S	No unvisited adjacent city	P W
Push Y	Next unvisited adjacent city	P W Y
Push Z	Next unvisited adjacent city	P W Y Z

- A trace of the search algorithm, given the flight map

Using Stack to Search a Flight Map

```
bool Map::isPath(City originCity, City destinationCity)
{
   Stack cityStack;

   unvisitAll(); // Clear marks on all cities

   // Push origin city onto cityStack and mark it as visited
   cityStack.push(originCity);
   markVisited(originCity);

   City topCity = cityStack.peek();
   while (!cityStack.isEmpty() && (topCity != destinationCity))
   {
      // The stack contains a directed path from the origin city
      // at the bottom of the stack to the city at the top of the stack

      // Find an unvisited city adjacent to the city on the top of the stack
      City nextCity = getNextCity(topCity);

      if (nextCity == NO_CITY)
```

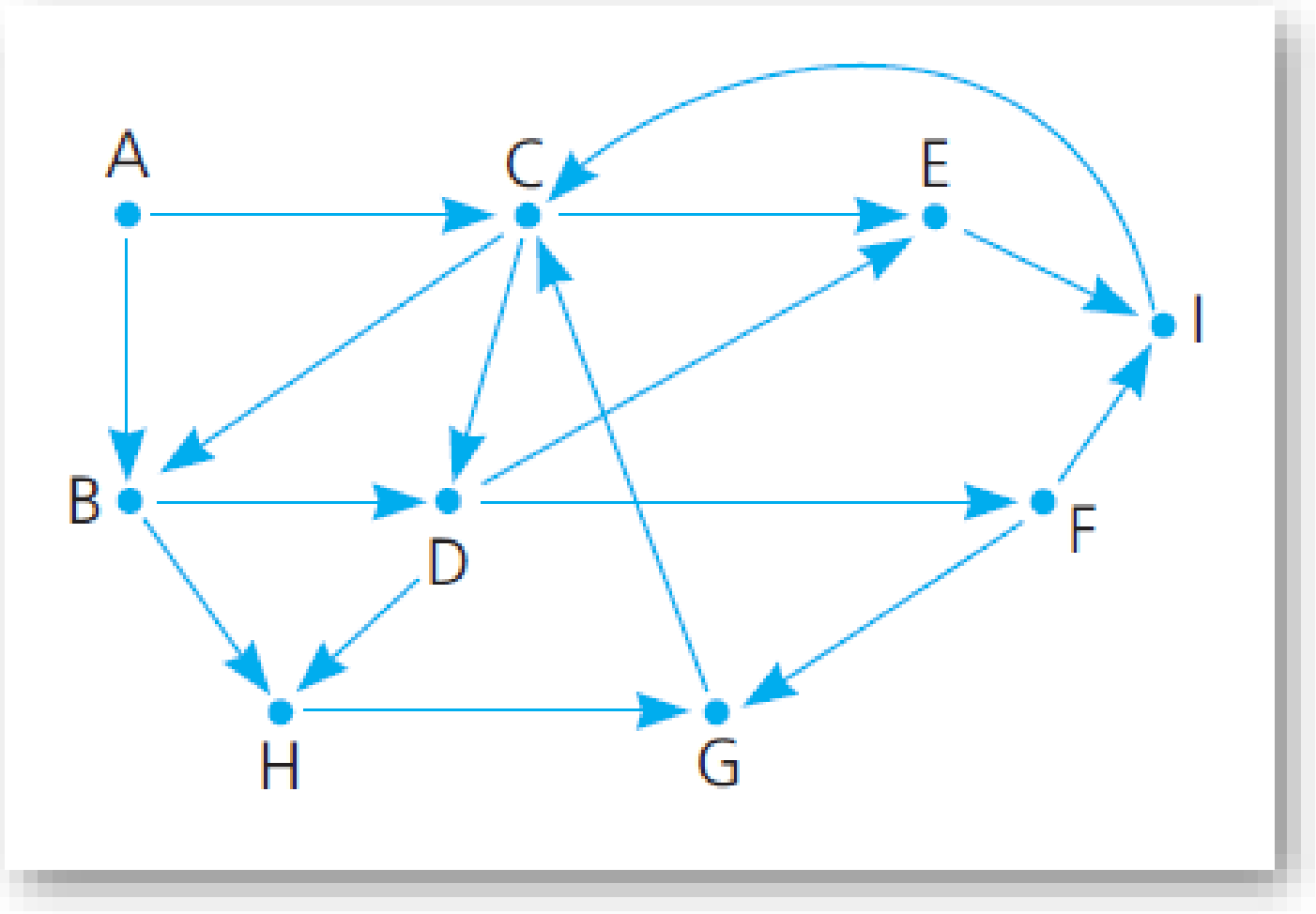
- C++ implementation of searchS

Using Stack to Search a Flight Map

```
        City nextCity = getNextCity(topCity);

        if (nextCity == NO_CITY)
           cityStack.pop(); // No city found; backtrack
        else                // Visit city
        {
           cityStack.push(nextCity);
           markVisited(nextCity);
        } // end if

        if (!cityStack.isEmpty())
           topCity = cityStack.peek();
     } // end while

     return !cityStack.isEmpty();
  } // end isPath
```

- C++ implementation of searchS

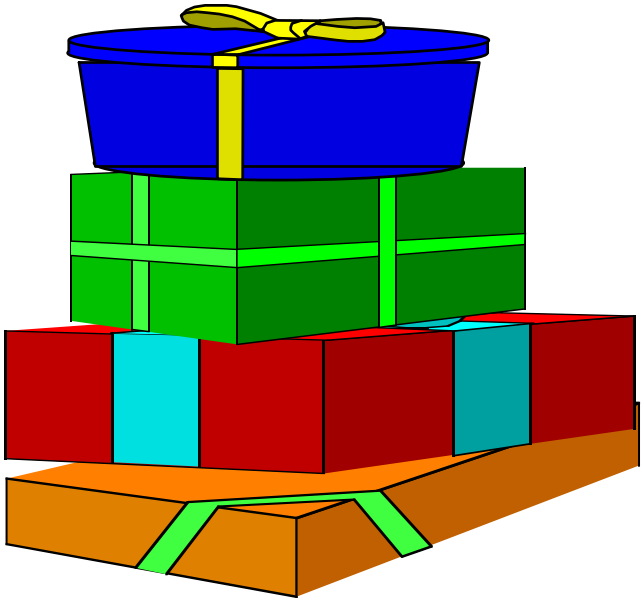
Using Stack to Search a Flight Map



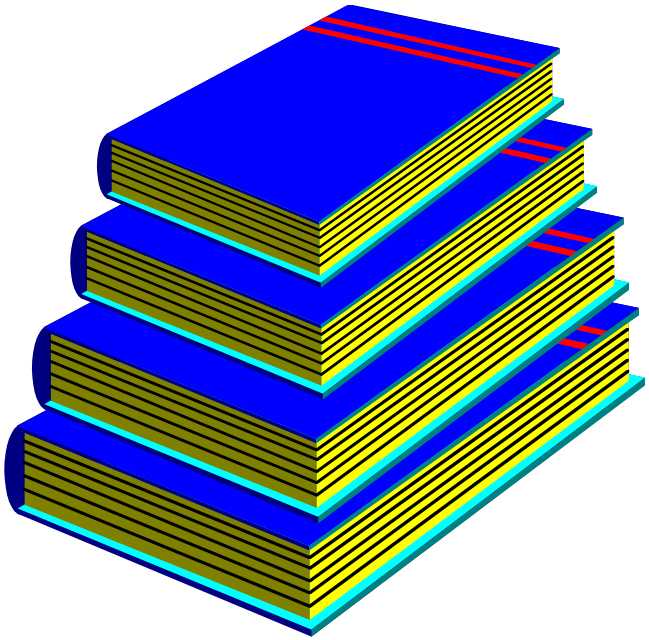
- Flight map for Checkpoint Question 8

Stacks of Boxes and Books

TOP OF THE STACK



TOP OF THE STACK

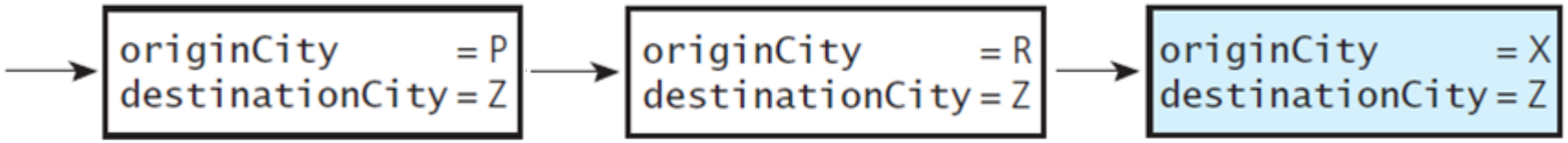


Relationship Between Stacks and Recursion

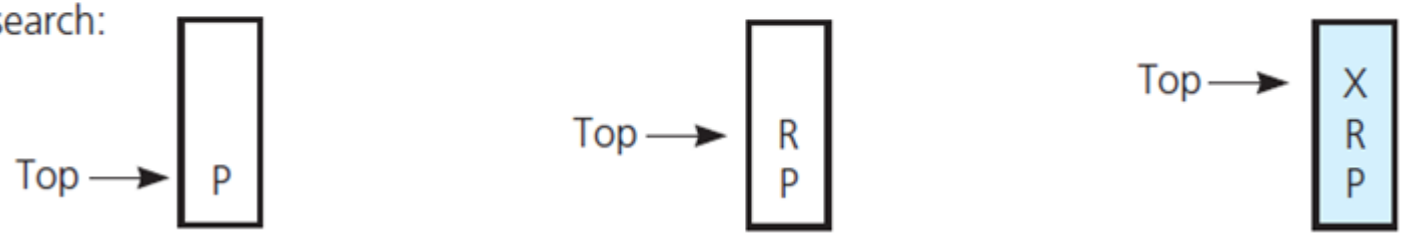
- Key aspects of common strategy
  - Visiting a new city
  - Backtracking
  - Termination

Relationship Between Stacks and Recursion

(a) Box trace of recursive search:

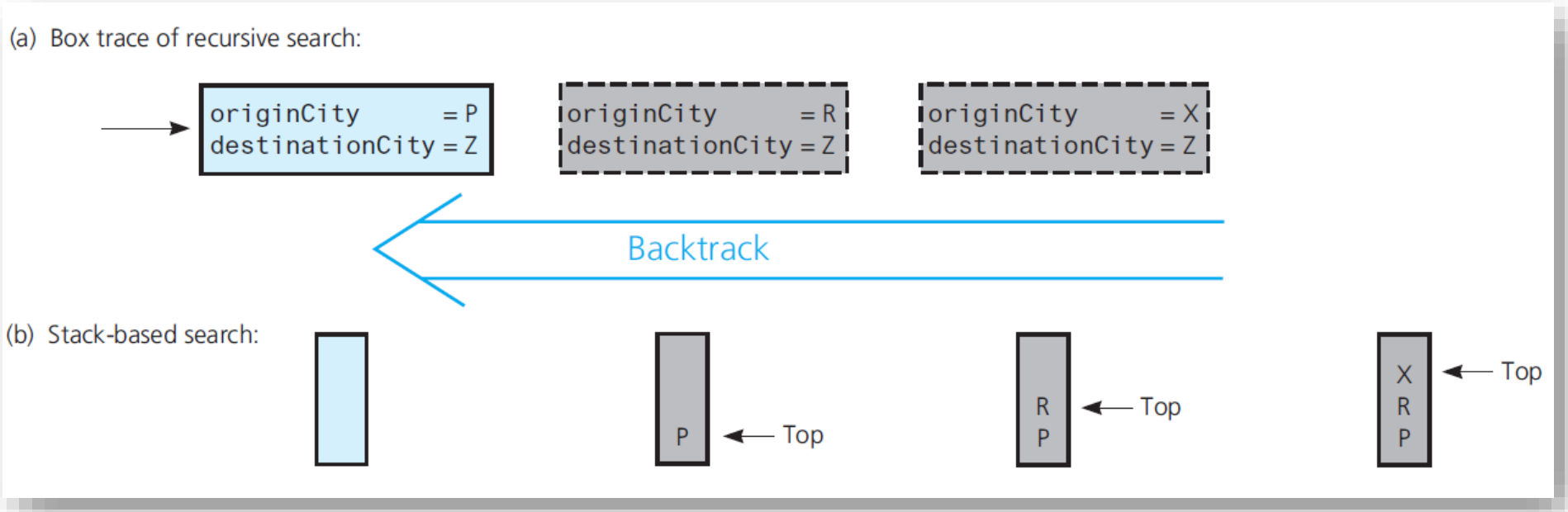


(b) Stack-based search:



- Visiting city P , then R , then X :
  (a) box trace versus (b) stack

Relationship Between Stacks and Recursion



- Backtracking from city X to R to P :
  (a) box trace versus (b) stack

Stack Activation Frames

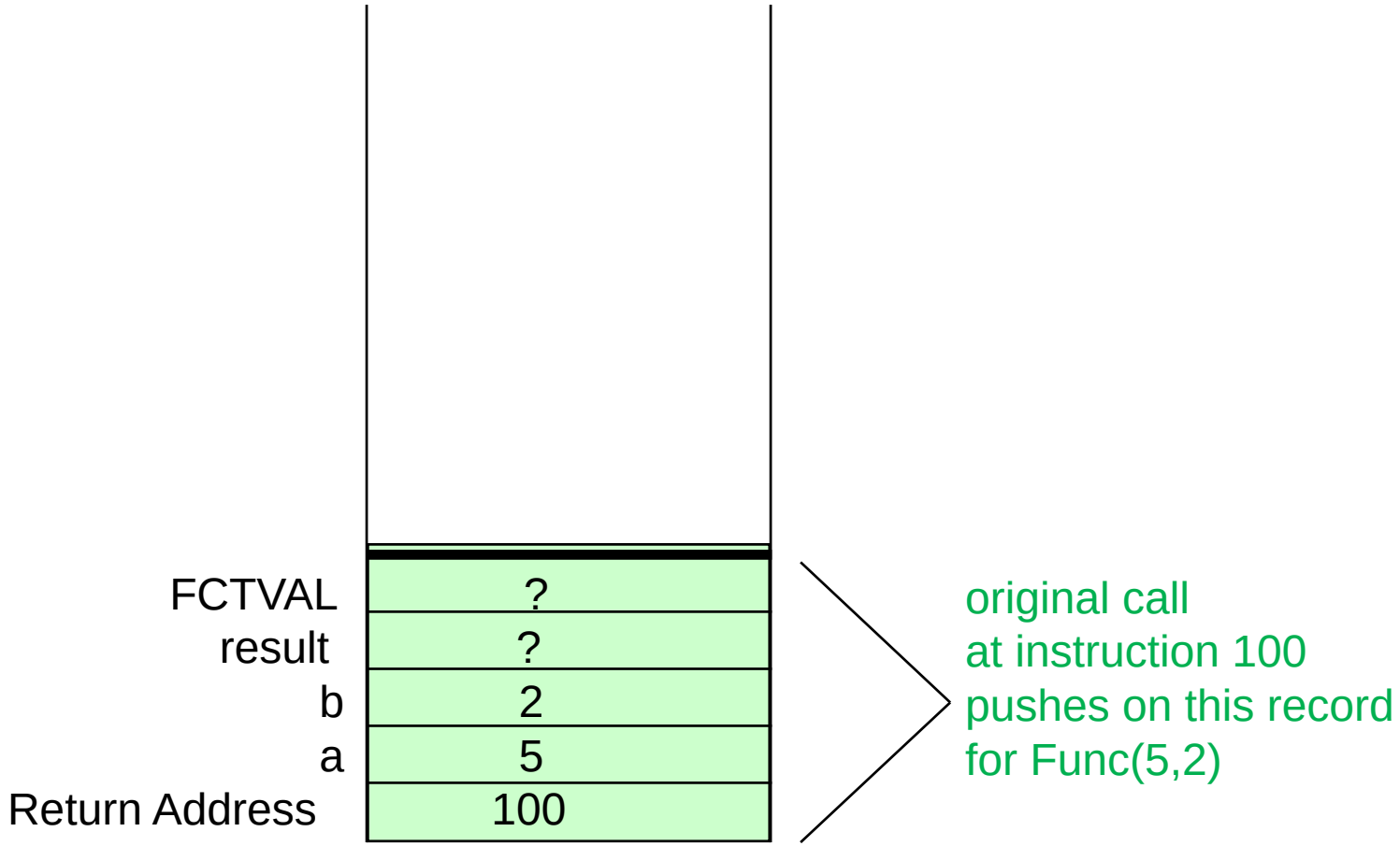
- The activation record stores
  - the return address for this function call,
  - the parameters,
  - local variables, and
  - the function's return value, if non-void.
- The activation record for a particular function call is popped off the run-time stack when
  - the final closing brace in the function code is reached, or
  - when a return statement is reached in the function code.
- At this time the function's return value, if non-void, is brought back to the calling block return address for use there.

```
// Another recursive function
int  Func  (  int  a, int b )
   //  Pre:    a and  b have been assigned values
   //  Post:   Function value = ??
{
   int  result;
   if  ( b == 0 )                              //  base case
        result = 0;
   else
   {
     if  ( b > 0 )                           //  first  general  case
        result = a +  Func ( a , b - 1 ) );    // instruction 50

     else                                      //  second general case
        result = Func ( - a , - b );            // instruction 70
   }
   return  result;
}
```

What operation does Func(a, b) simulate?

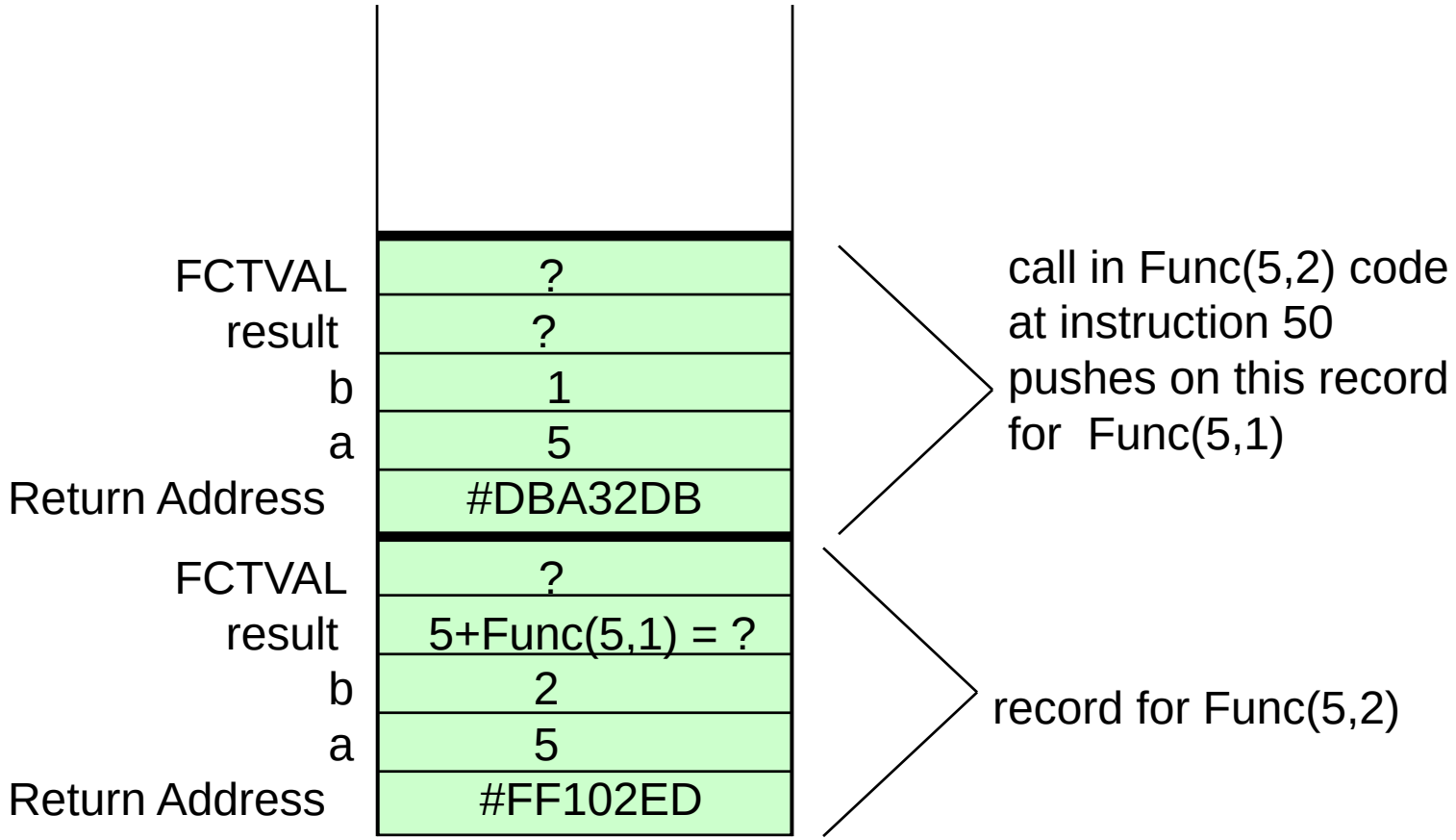
Run-Time Stack Activation Records

```
x = Func(5, 2);          // original call is instruction 100
```

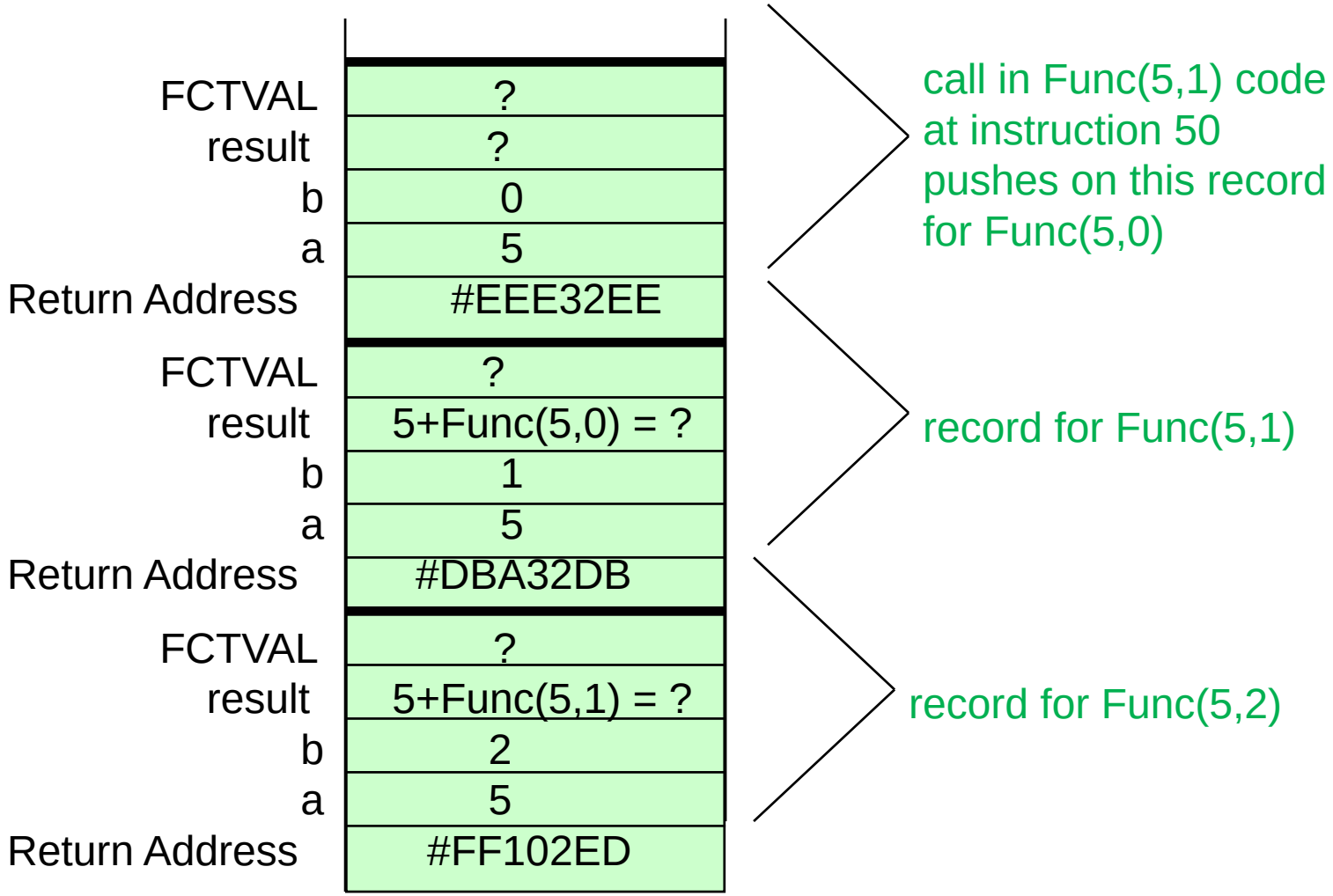


Run-Time Stack Activation Records

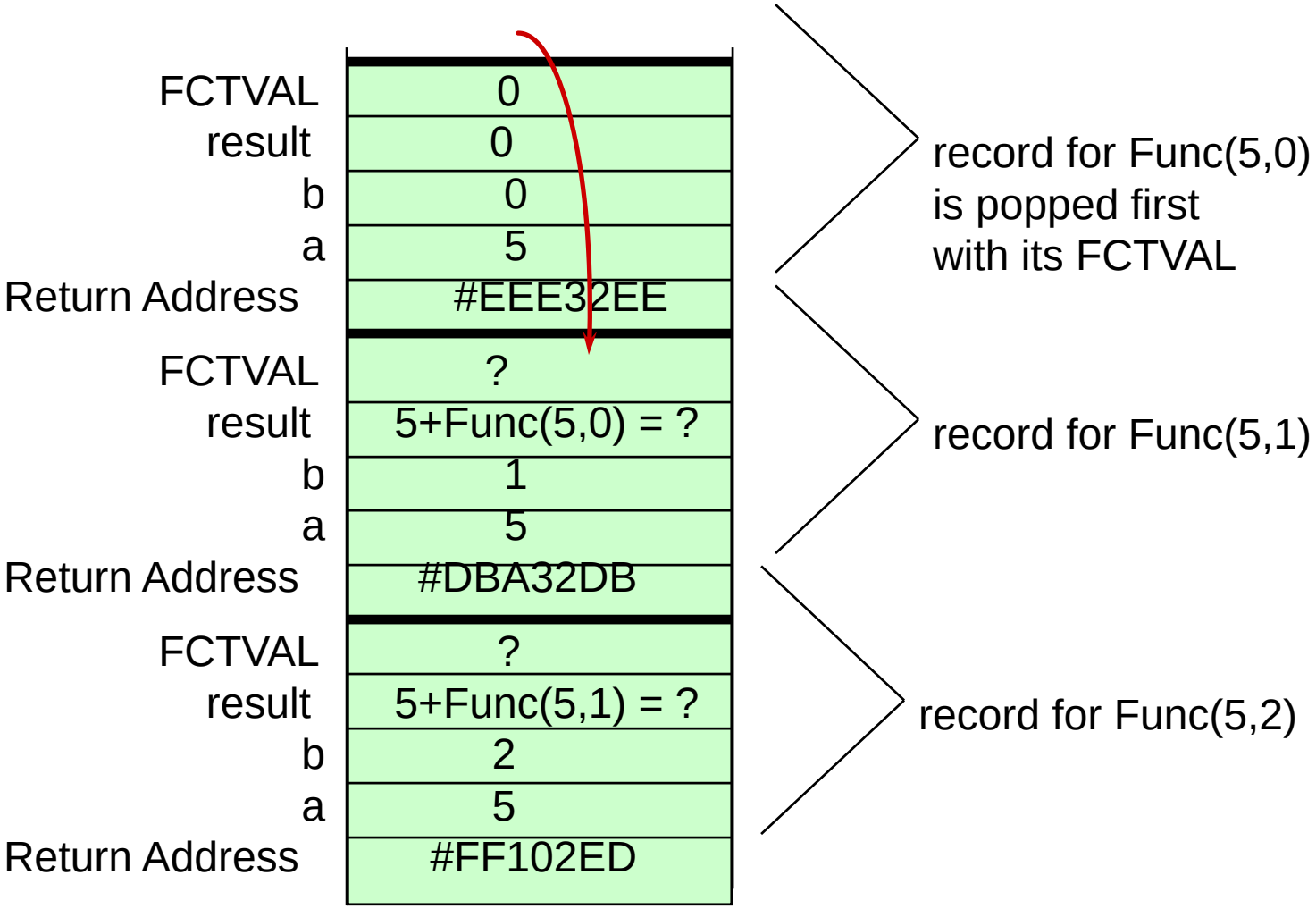
```
x = Func(5, 2);
```
// original call at instruction 100



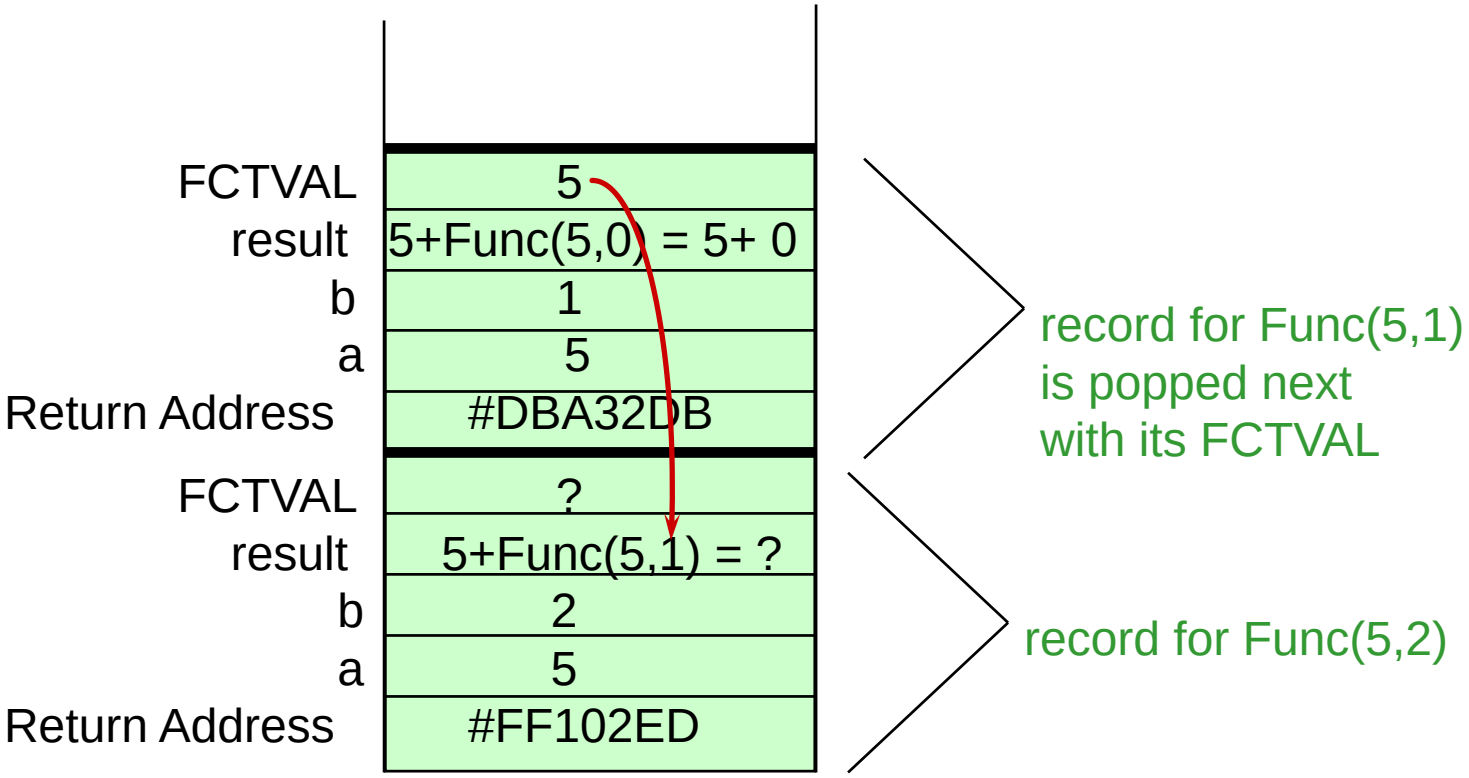
Run-Time Stack Activation Records



Run-Time Stack Activation Records

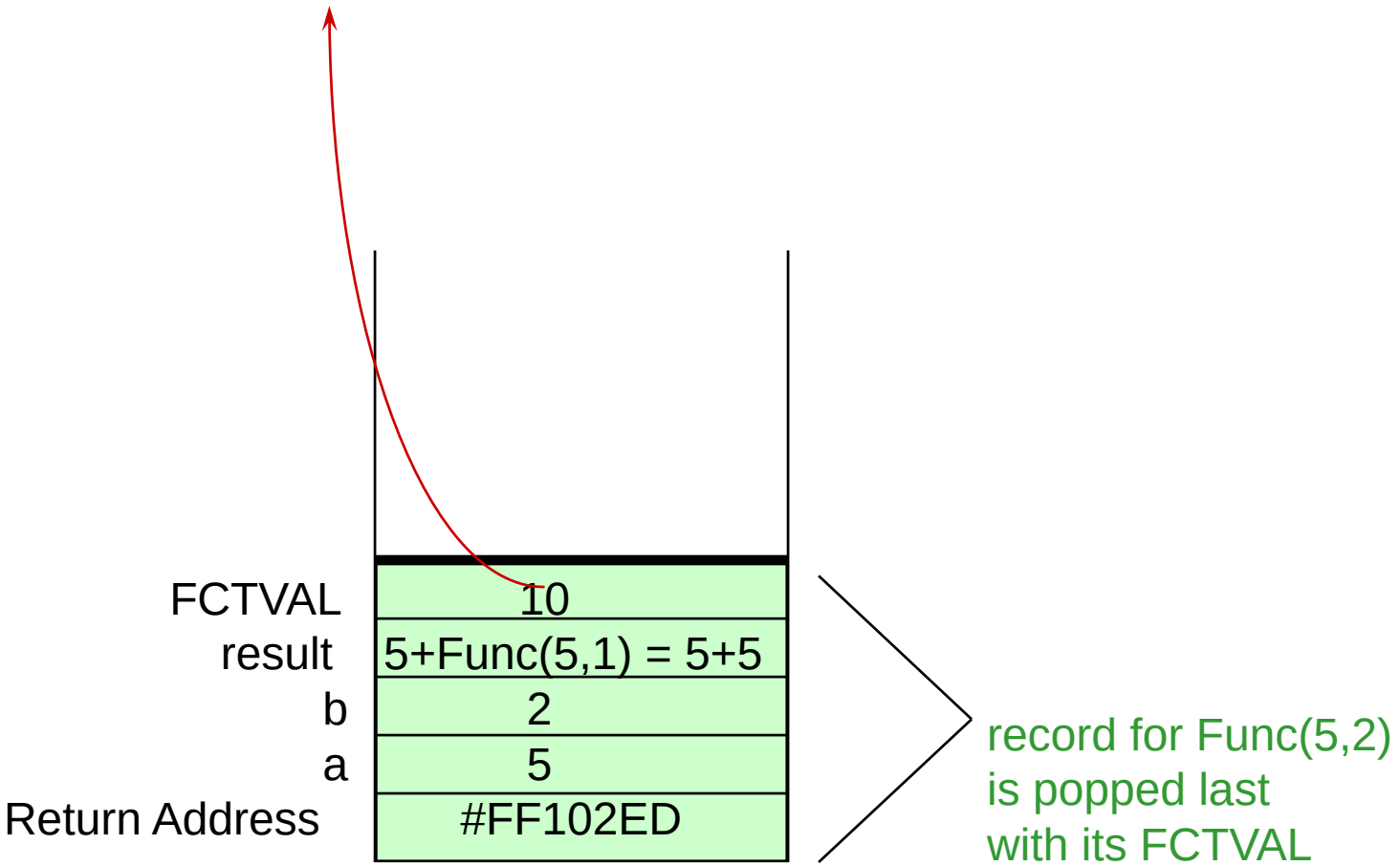


Run-Time Stack Activation Records

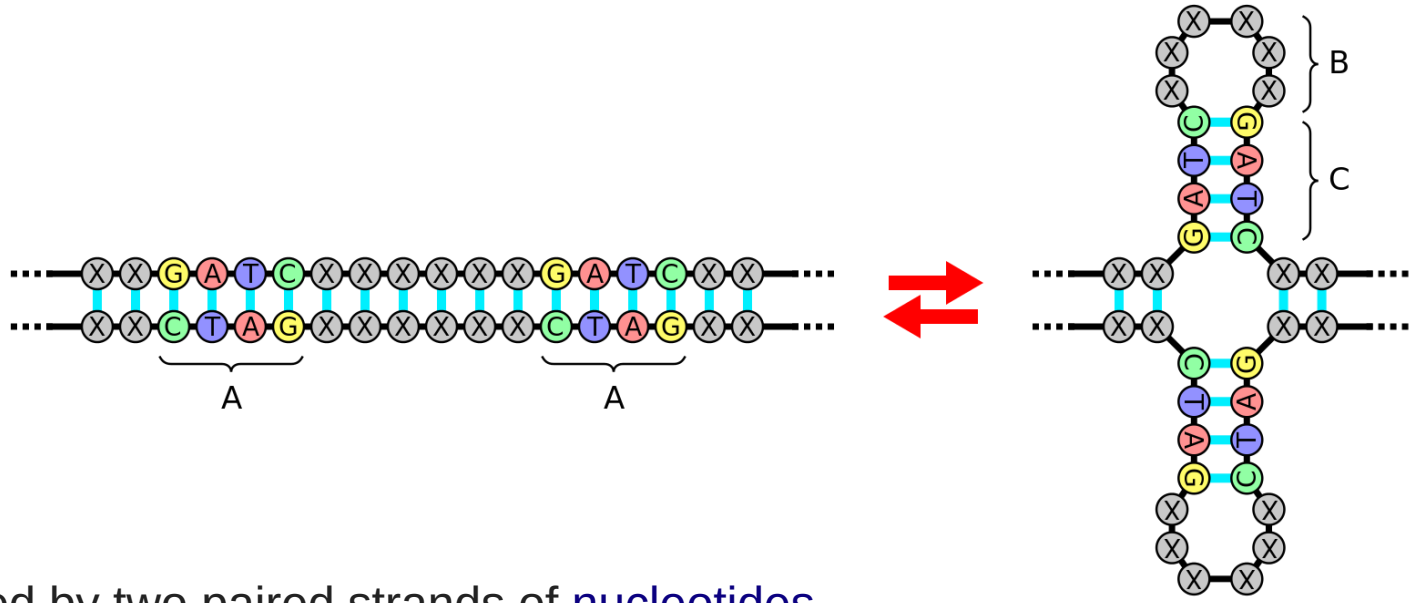


Run-Time Stack Activation Records

```
x = Func(5, 2);          // original call at line 100
```



Biological Structures

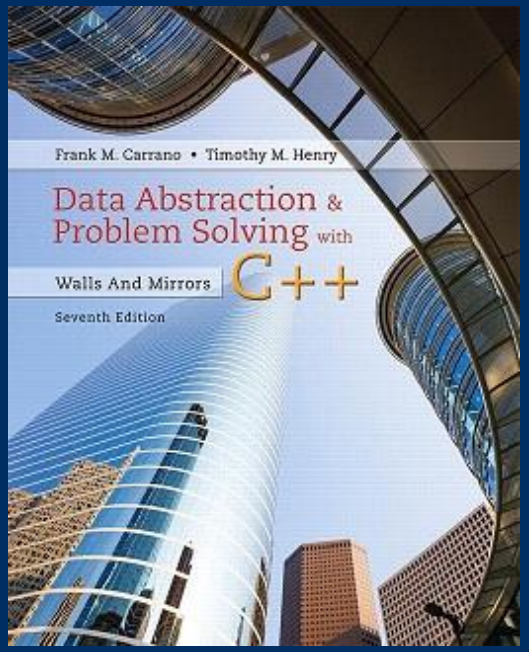


- DNA is formed by two paired strands of nucleotides,
- The nucleotides always pair in the same way (Adenine (A) with Thymine (T), Cytosine (C) with Guanine (G)),
  - a (single-stranded) sequence of DNA is said to be a palindrome if it is equal to its complementary sequence read backward.

- For example, the sequence ACCTAGGT is palindromic because its complement is TGGATCCA, which is equal to the original sequence in reverse complement.

Chapter 6
Stacks

To be continued!



Modified from authors' slides

Out of the Box

- Regular Expression Matching {Hard}

  https://leetcode.com/problems/regular-expression-matching/description/

- Generate Parentheses {Medium}

  https://leetcode.com/problems/generate-parentheses/description/

- Valid Parentheses {Easy}

  https://leetcode.com/problems/valid-parentheses/description/