COM 201 – Data Structures and Algorithms

Abstract Data Types – Stack II

Assist. Prof. Özge ÖZTİMUR KARADAĞ

Department of Computer Engineering – ALKÜ

Alanya

Previously

- Stack
  - LIFO
    - Push(), pop() etc.
  - Sample applications with stack

Sample Applications with Stack ADT

- Evaluating an arithmetic expression in postfix notation;
  - Presedence of Binary operations:
    - Exponentiation ↑
    - Multiplication (*) and division (/)
    - Addition (+) and subtraction (-)
  - Notations for artihmetic expressions
    - Infix A + B
    - Prefix + A B
    - Postfix A B +

Sample Applications with Stack ADT

- Translate following infix expressions to postfix
  - (A+B) * C
    - (A+B) * C = [AB+]*C = A B+ C*
  - (A + B) / (C – D)
    - (A + B) / (C – D) = [AB+]/[CD-] = A B+ C D-/
  - We do not need parantheses in the postfix and prefix notations.
  - Computers usually evaluate arithmetic expressions in infix notations in two steps:
    - Step 1 - Convert the expression to postfix notation
    - Step 2 - Evaluate the postfix expressions
    - Stack is the main tool at each step

Application: Algebraic Expressions

- When the ADT stack is used to solve a problem, the use of the ADT's operations should not depend on its implementation
- To evaluate an infix expression *//**in**fix: operator **in** b/w operands*
  - Convert the infix expression to postfix form
  - Evaluate the postfix expression *//**post**fix: operator **after** operands; similarly we have **pre**fix: operator **before** operands*

Infix Expression	Postfix Expression	Prefix Expression
5 + 2 * 3		
5 * 2 + 3		
5 * (2 + 3) - 4		

Application: Algebraic Expressions

- Infix notation is easy to read for humans, whereas pre-/postfix notation is easier to parse for a machine. The big advantage in pre-/postfix notation is that there never arise any questions like operator precedence
- Or, to put it in more general terms: it is possible to restore the original (parse) tree from a pre-/postfix expression without any additional knowledge, but the same isn't true for infix expressions

- For example, consider the infix expression 1 # 2 $ 3. Now, we don't know what those operators mean, so there are two possible corresponding postfix expressions: 1 2 # 3 $ and 1 2 3 $ #. Without knowing the rules governing the use of these operators, the infix expression is essentially worthless.

Evaluating Postfix Expressions

- When an operand is entered, the calculator
  - Pushes it onto a stack

- When an operator is entered, the calculator
  - Applies it to the top two operands of the stack
  - Pops the operands from the stack
  - Pushes the result of the operation on the stack

Evaluating Postfix Expressions: 2 3 4 + *

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

Converting Infix Expressions to Postfix Expressions

- An infix expression can be evaluated by first being converted into an equivalent postfix expression
- Facts about converting from infix to postfix
  - Operands always stay in the same order with respect to one another
  - An operator will move only "to the right" with respect to the operands
  - All parentheses are removed

Converting Infix Expressions to Postfix Expressions

ch	Stack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators
	– (	abcd*+	from stack to
	–	abcd*+	postfixExp until " ("
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from
		abcd*+e/–	stack to postfixExp

a – (b + c * d)/ e ➔ a b c d * + e / –

Converting Infix Expr. to Postfix Expr. -- Algorithm

```
for (each character ch in the infix expression) {
    switch (ch) {
      case operand:      // append operand to end of postfixExpr
          postfixExpr=postfixExpr+ch;   break;
      case '(':          // save '(' on stack
          aStack.push(ch);   break;
      case ')':          // pop stack until matching '(', and remove '('
          while (top of stack is not '(') {
              postfixExpr=postfixExpr+(top of stack);   aStack.pop();
          }
          aStack.pop();   break;
```

Converting Infix Expr. to Postfix Expr. -- Algorithm

```
  case operator:
      aStack.push();   break;          // save new operator
} } // end of switch and for

// append the operators in the stack to postfixExpr
while (!isStack.isEmpty()) {
    postfixExpr=postfixExpr+(top of stack);
     aStack(pop);
}
```

Sample Applications with Stack ADT

- Transforming Infix Expressions into Postfix Expressions
  - If there are no paranthesis, need to consider operator presedence

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push "(" onto STACK, and add ")" to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator ⊗ is encountered, then:
   (*a*) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than ⊗.
   (*b*) Add ⊗ to STACK.
   [End of If structure.]
6. If a right parenthesis is encountered, then:
   (*a*) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
   (*b*) Remove the left parenthesis. [Do not add the left parenthesis to P.]
   [End of If structure.]
   [End of Step 2 loop.]
7. Exit.

Sample Applications with Stack ADT

- Transforming Infix Expressions into Postfix Expressions
  - Example: A + (B * C – (D / E ↑ F) * G) * H

Symbol Scanned		STACK	Expression P
(1)	A	(	A
(2)	+	(+	A
(3)	(	(+ (	A
(4)	B	(+ (	A B
(5)	*	(+ (*	A B
(6)	C	(+ (*	A B C
(7)	−	(+ (−	A B C *
(8)	(	(+ (− (	A B C *
(9)	D	(+ (− (	A B C * D
(10)	/	(+ (− (/	A B C * D
(11)	E	(+ (− (/	A B C * D E
(12)	↑	(+ (− (/ ↑	A B C * D E
(13)	F	(+ (− (/ ↑	A B C * D E F
(14)	)	(+ (−	A B C * D E F ↑ /
(15)	*	(+ (− *	A B C * D E F ↑ /
(16)	G	(+ (− *	A B C * D E F ↑ / G
(17)	)	(+	A B C * D E F ↑ / G * −
(18)	*	(+ *	A B C * D E F ↑ / G * −
(19)	H	(+ *	A B C * D E F ↑ / G * − H
(20)	)		A B C * D E F ↑ / G * − H * +

The Relationship Between Stacks and Recursion

- Recursion:
  - P is a recursive procedure if it contains a call statement to itself or a call to a second procedure that may eventually result in a call statement back to the original procedure P.
  - P must have the two properties to make sure that the program will not run indefinitely.
    - There must be a base criteria, for which the procedure does not call itself.
    - Each time the procedure does call itself (directly or indirectly) it must be closer to the base criteria.

The Relationship Between Stacks and Recursion

- A strong relationship exists between recursion and stacks
- Typically, stacks are used by compilers to implement recursive methods
  - During execution, each recursive call generates an activation record that is pushed onto a stack

  That's why we can get **stack overflow** error if a function makes too many recursive calls
- Stacks can be used to implement a nonrecursive version of a recursive algorithm

An activation record (AR) is **a private block of memory associated with an invocation of a procedure**. It is a runtime structure used to manage a procedure call. An AR is used to map a set of arguments, or parameters, from the caller's name space to the callee's name space.

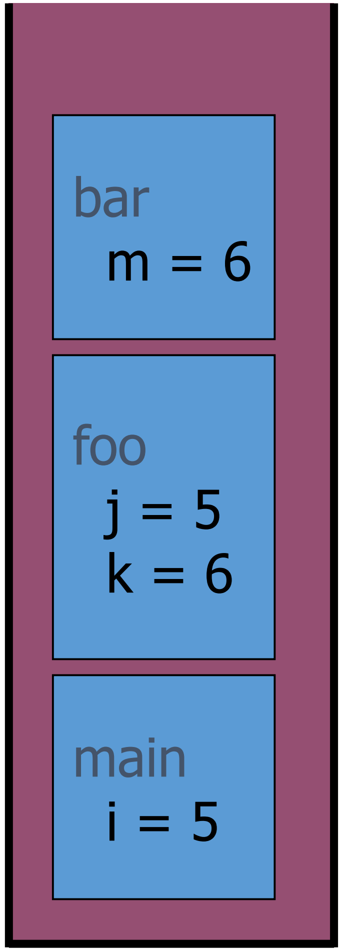
C++ Run-time Stack

- The C++ run-time system keeps track of the chain of active functions with a stack.
- When a function is called, the run-time system pushes on the stack a frame containing
  - –Local variables and return value
- When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack

```
main() {
  int i = 5;
  foo(i);
  }

foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }

bar(int m) {
  …
  }
```

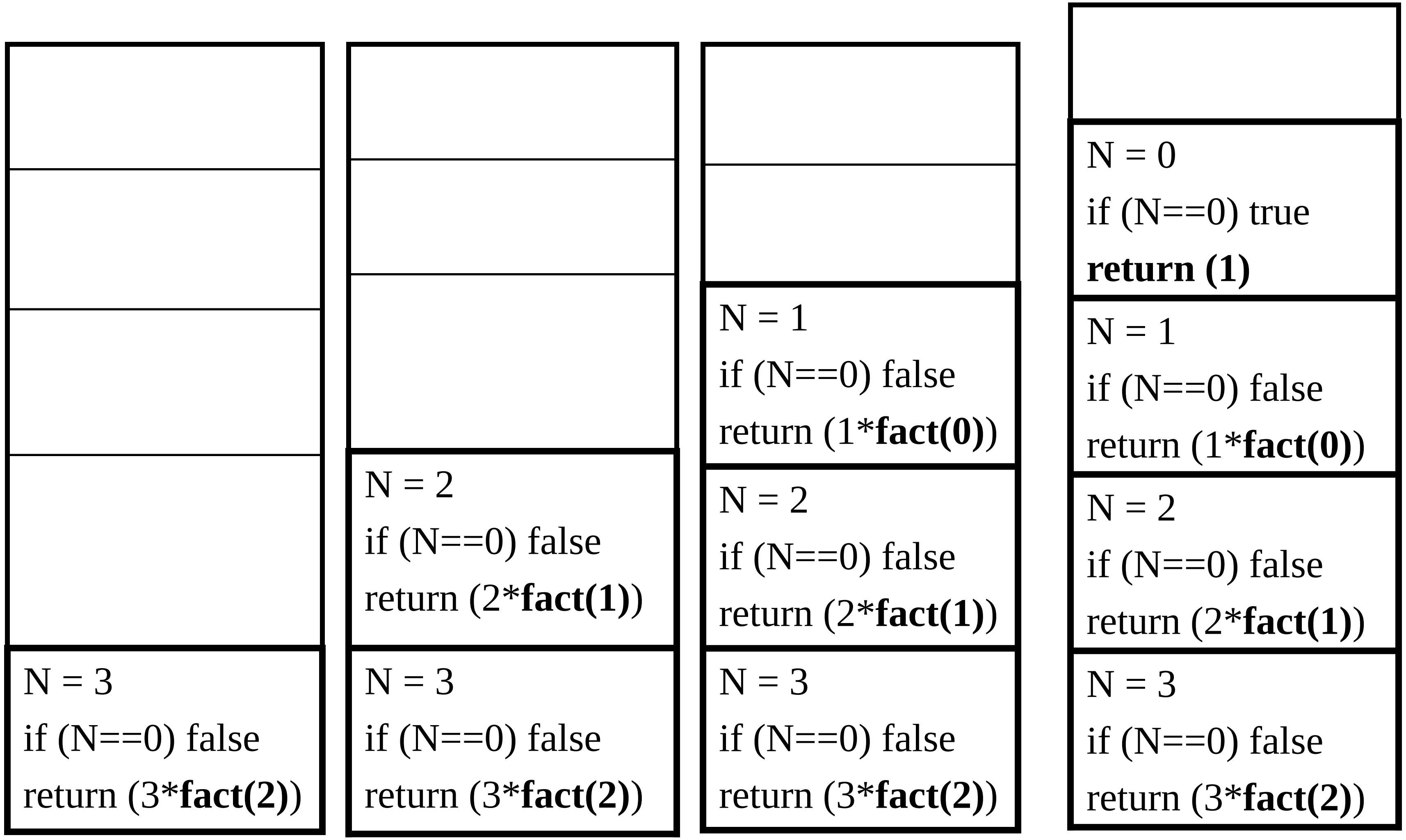


Run-time Stack

Example: Factorial function

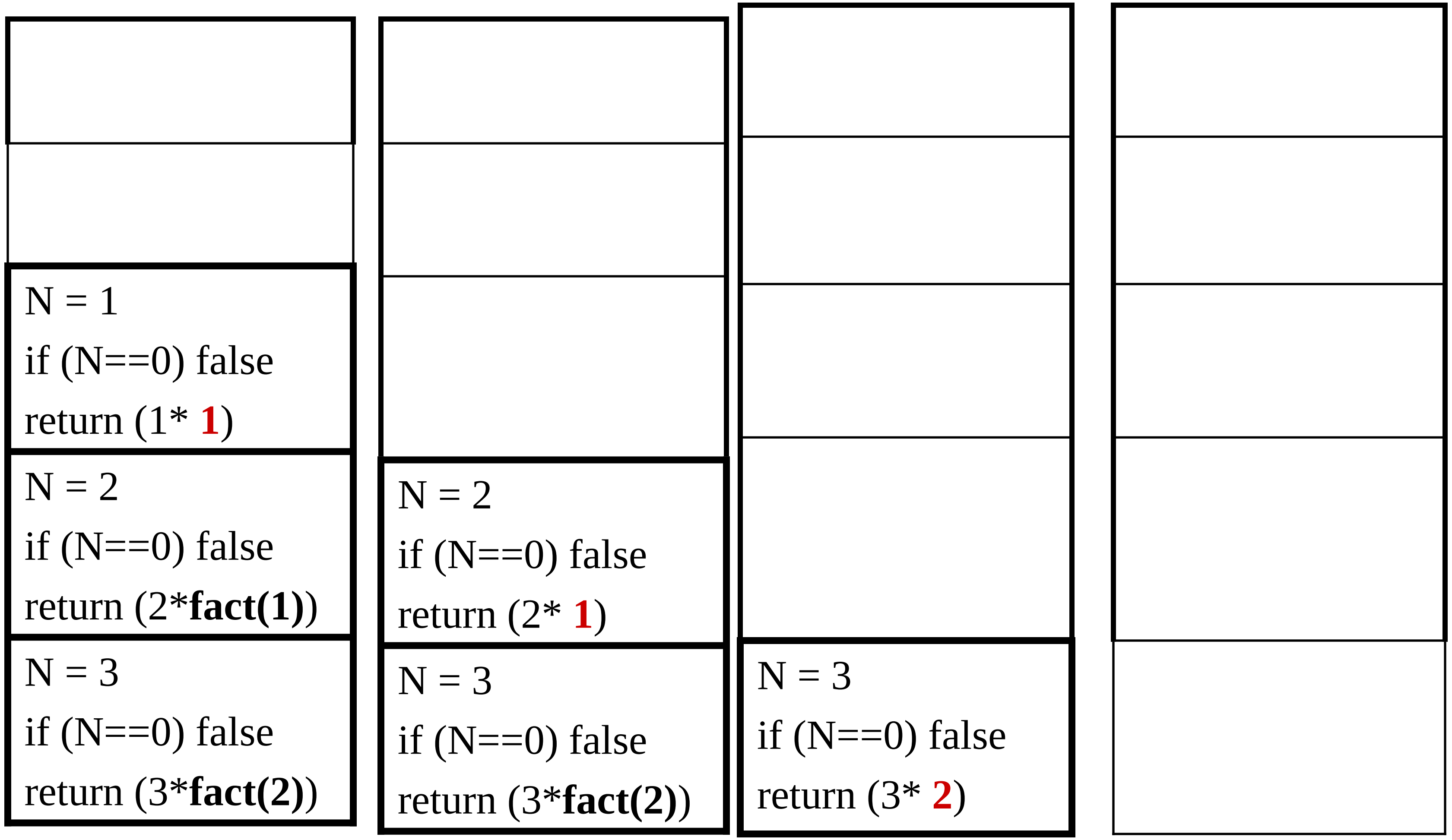
```
int fact(int n)
{
 if (n ==0)
     return (1);
 else
      return (n * fact(n-1));
}
```

Tracing the call `fact (3)`



After original call | After 1st call | After 2nd call | After 3rd call

Tracing the call `fact (3)`



After return from 3rd call

After return from 2nd call

After return from 1st call

return **6**

Example: Reversing a string

```
void printReverse(const char* str)
{

    ????????



}
```

Example: Reversing a string

```
void printReverse(const char* str)
{
    if (*str) {
        printReverse(str + 1)
        cout << *str << endl;
    }
}
```

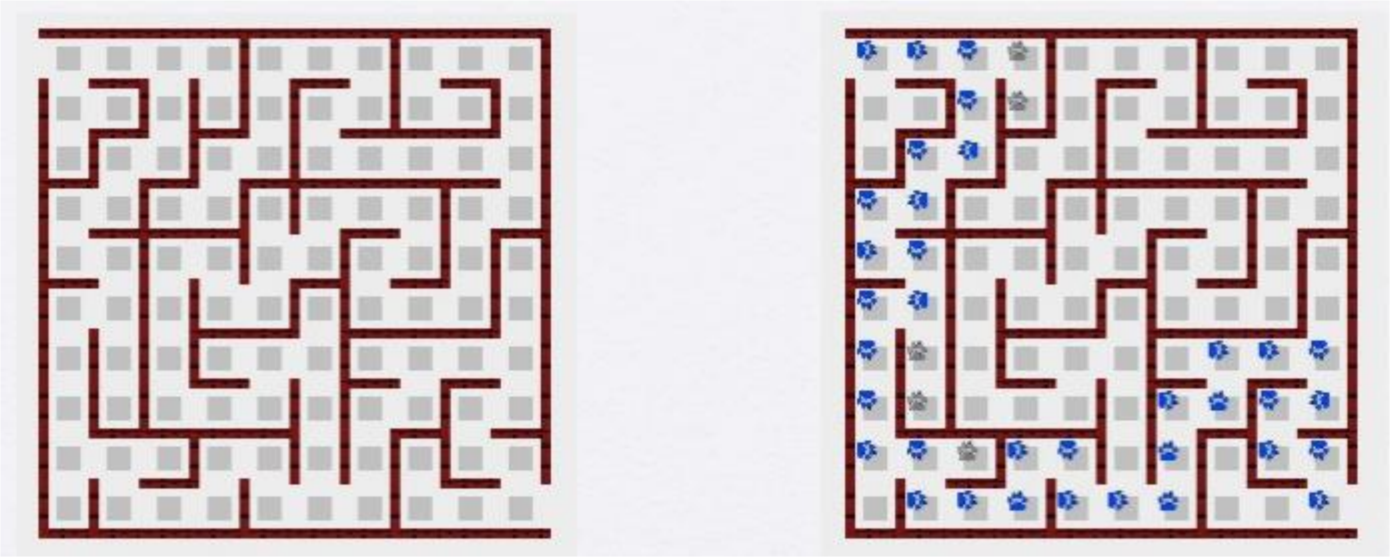
Example: Reversing a string

```
void printReverseStack(const char* str)
{
    Stack<char> s;
    for (int i = 0; str[i] != '\0'; ++i)
        s.push(str[i]);

    while(!s.isEmpty()) {
        char c;
        s.topAndPop(c);
        cout << c;
    }
}
```

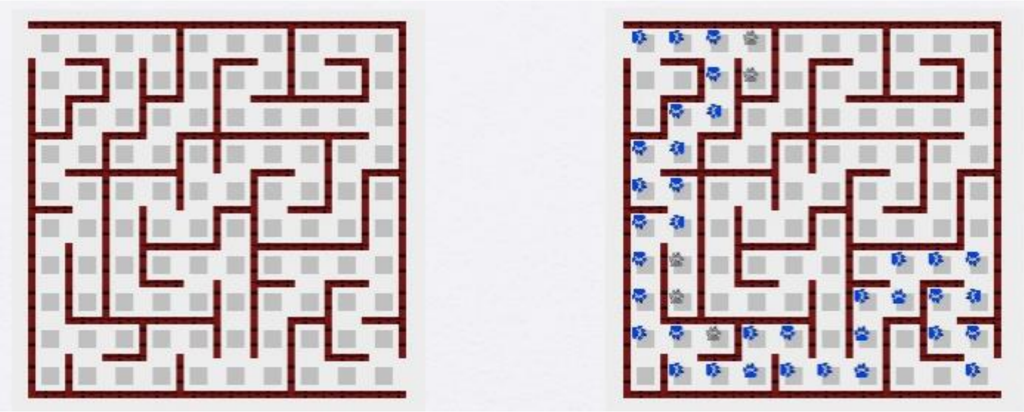
Example: Maze Solving

- Find a path on the maze represented by 2D array

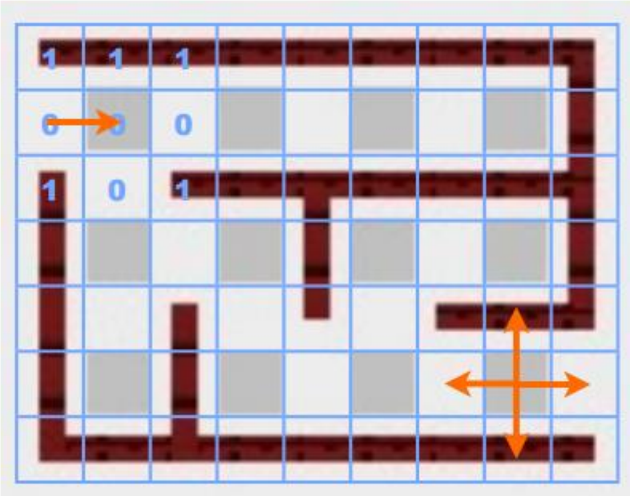


Example: Maze Solving

- Find a path on the maze represented by 2D array

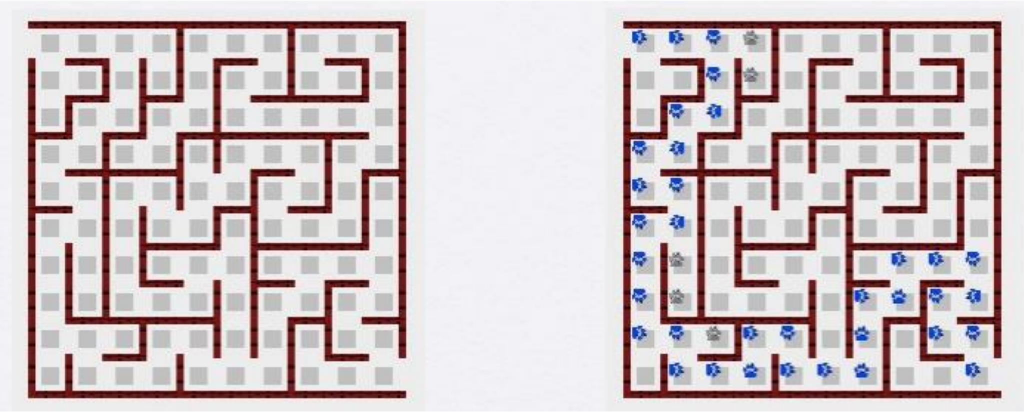


- Push the traced points into a stack

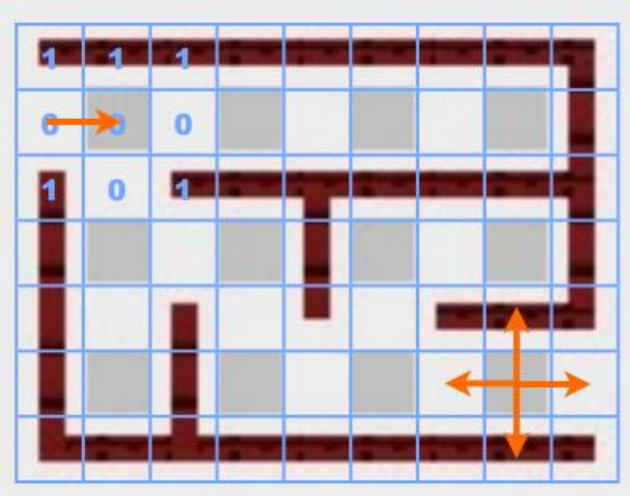


Example: Maze Solving

- Find a path on the maze represented by 2D array



- Move forward if possible. If not, pop until a new direction move is possible.



Example: Integer to Binary

- Binary to integer is easy; how about the reverse?
- What is 233 in binary?

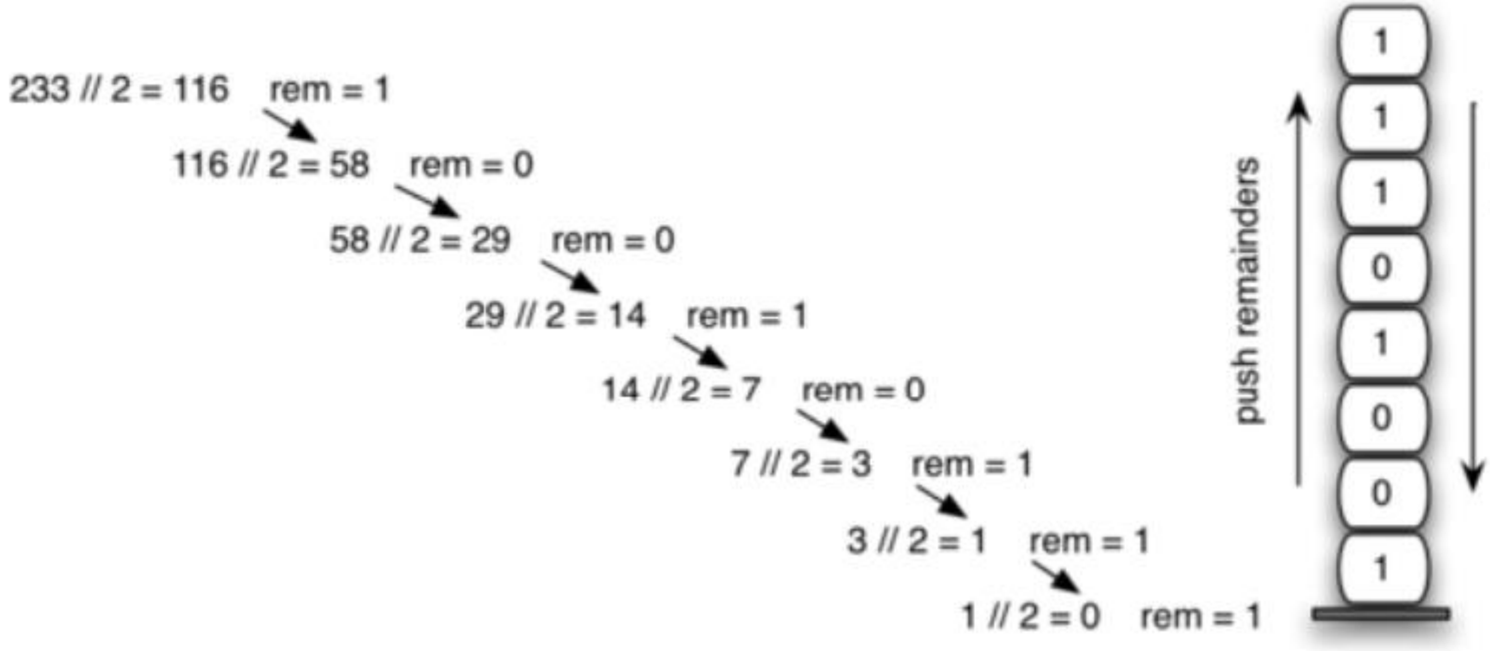
Example: Integer to Binary

- Binary to integer is easy; how about the reverse?
- What is 233 in binary?

233 // 2 = 116 rem = 1
116 // 2 = 58 rem = 0
58 // 2 = 29 rem = 0
29 // 2 = 14 rem = 1
14 // 2 = 7 rem = 0
7 // 2 = 3 rem = 1
3 // 2 = 1 rem = 1
1 // 2 = 0 rem = 1

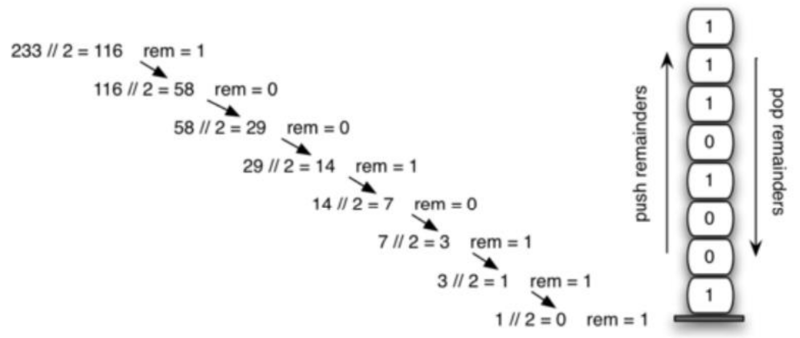
Example: Integer to Binary

- Binary to integer is easy; how about the reverse?
- What is 233 in binary?



Example: Integer to Binary

- Binary to integer is easy; how about the reverse?
- What is 233 in binary?




```
while dec_number > 0:
    rem = dec_number % 2
    rem_stack.push(rem)
    dec_number = dec_number / 2

bin_string = ""
while not rem_stack.is_empty():
    bin_string = bin_string + str(rem_stack.pop())

return bin_string
```

Example: Sort a Stack

- Using no more than one additional stack
- Put top to temp; check before moving temp to 2nd stack

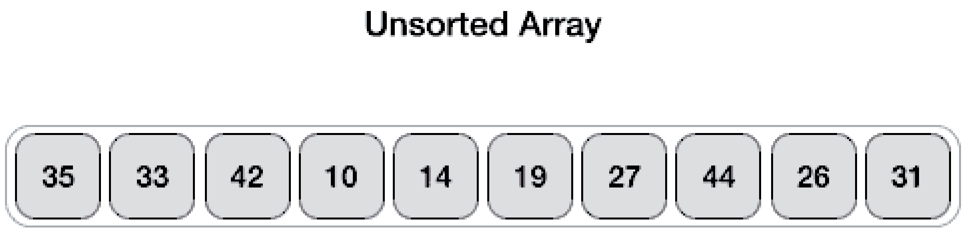
Sample Applications with Stack ADT

- Quicksort
  - 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

Sample Applications with Stack ADT

- Quicksort
  - 35 33 42 10 14 19 27 44 26 31

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot



Sample Applications with Stack ADT

- Quicksort
  - 44 33 11 55 77 90 40 60 99 22 88 66
    - Lower: 1 Upper: 12
    - Lower: Empty Upper: Empty
    - Lower: 1, 6 Upper: 4, 12 22 33 11 40 44 90 77 60 99 55 88 66
    - Lower: 1 Upper: 4 22 33 11 40 44 66 77 60 88 55 90 99
    - Lower: 1,6 Upper: 4, 10 22 33 11 40 44 55 60 66 70 88 90 99
    - Lower: 1,6,9 Upper: 4,7,10
    - Lower: Empty Upper:Empty 22 33 11 40 44 55 60 66 70 88 90 99

Checking for Balanced Braces

- A stack can be used to verify whether a program contains balanced braces

- An example of balanced braces

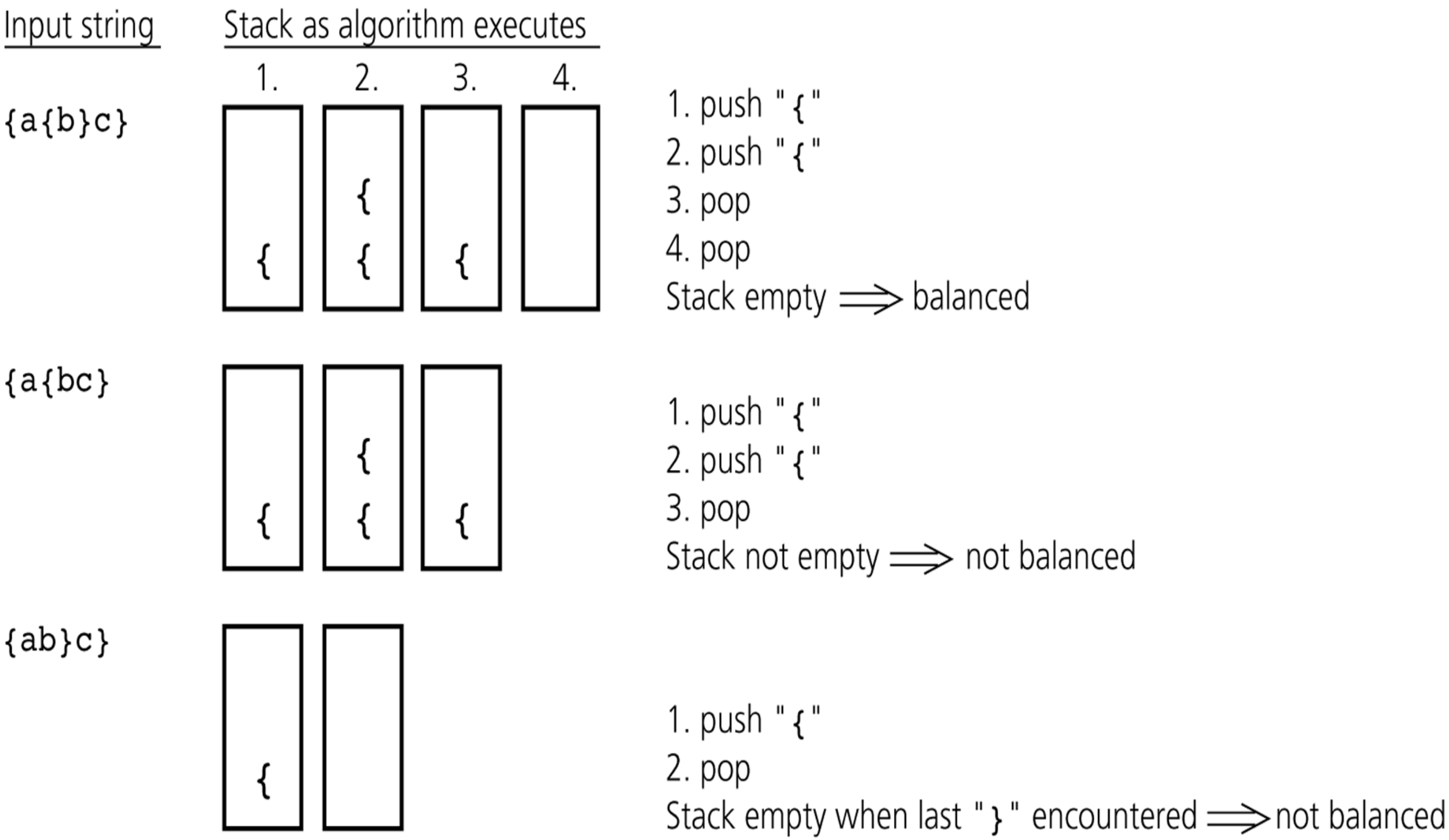
  `abc{defg{ijk}{l{mn}}op}qr`

- An example of unbalanced braces

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

- Requirements for balanced braces
  - Each time we encounter a "}", it matches an already encountered "{"
  - When we reach the end of the string, we have matched each "{"

Checking for Balanced Braces -- Traces



Checking for Balanced Braces -- Algorithm

```
aStack.createStack();        balancedSoFar = true;        i=0;
while (balancedSoFar and i < length of aString) {
    ch = character at position i in aString;    i++;
    if (ch is ‘{‘)                          // push an open brace
        aStack.push(‘{‘);
    else if (ch is ‘}’)                     // close brace
        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
```