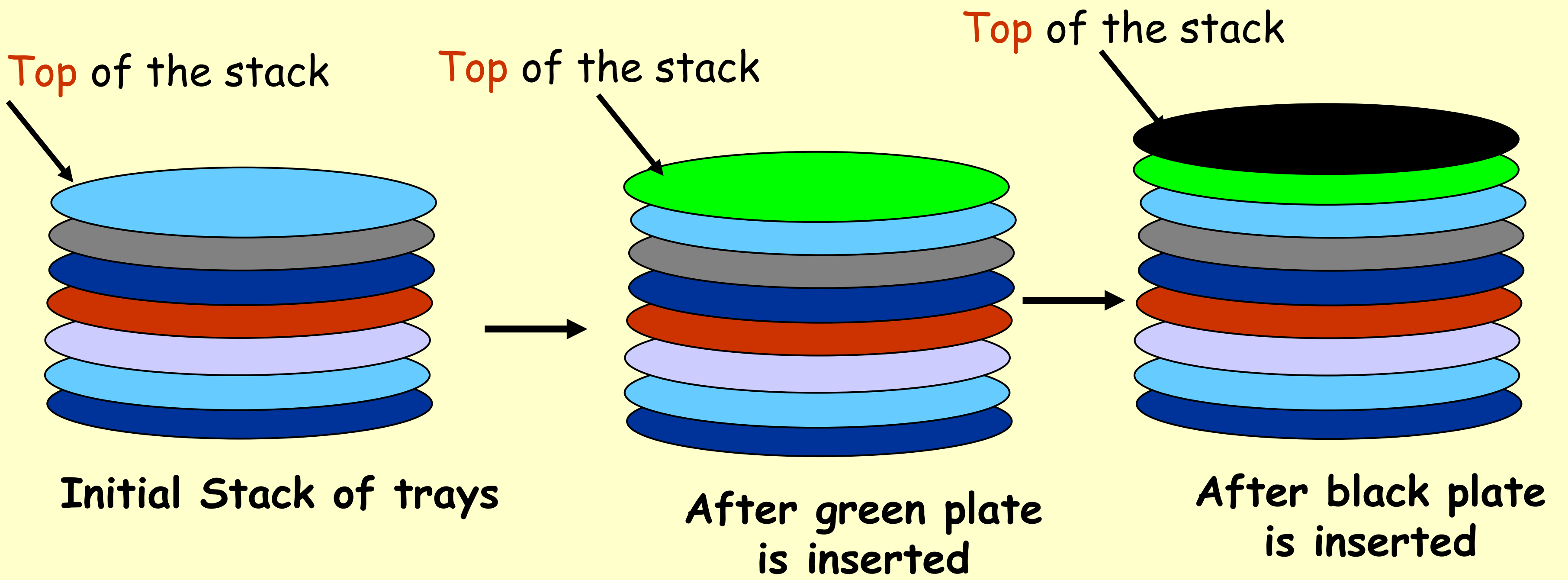
Stacks – Chapter 3

- A stack is a data structure in which all insertions and deletions of entries are made at one end, called the top of the stack.

- Alternatively, in a stack the element deleted is the most recently inserted. This is also called last-in-first-out (LIFO)

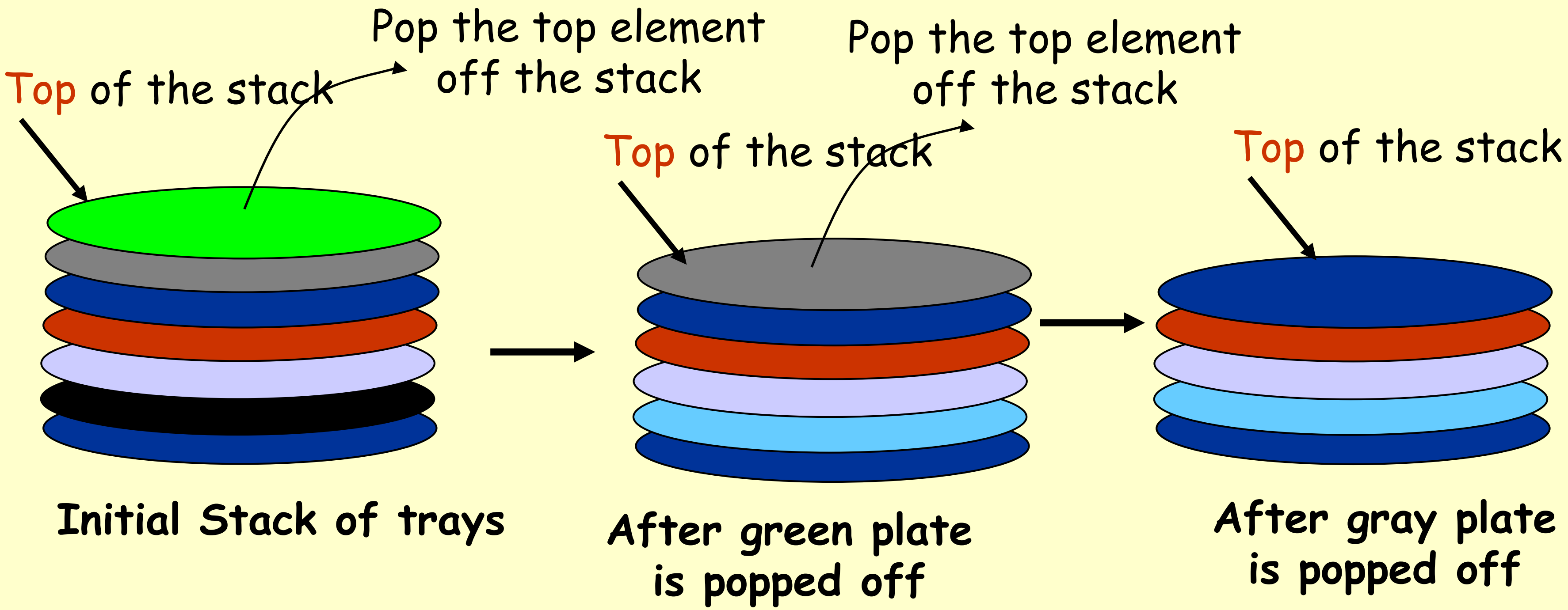
- Classical example for stacks is a stack of trays in a cafeteria

Stack Concept and Push Operation Example



- A stack has a top where insertions and deletions are made
- Insert operation in a stack is often called Push
- Notice that the element pushed to a stack is always placed at the top of the stack

Stack concept and Pop operation example



- Delete operation in a stack is often called Pop
- Notice that the element popped off the stack is always the one residing on top of the stack (LIFO)

Stack ADT

- A stack is a data structure in which all insertions and deletions of entries are made at one end, called the top of the stack.

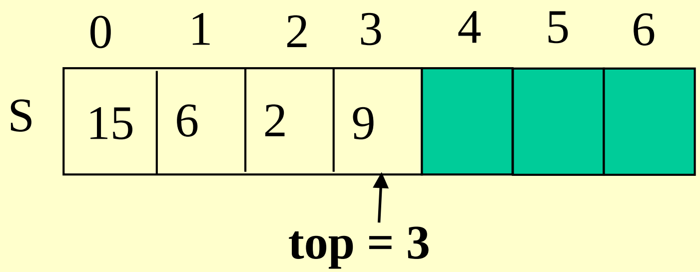
- Common stack operations:
  - Push(item) – push item to the top of the stack
  - Pop() – Remove & return the top item
  - Top() – Return the top item w/o removing it
  - isEmpty() – Return true if the stack is empty

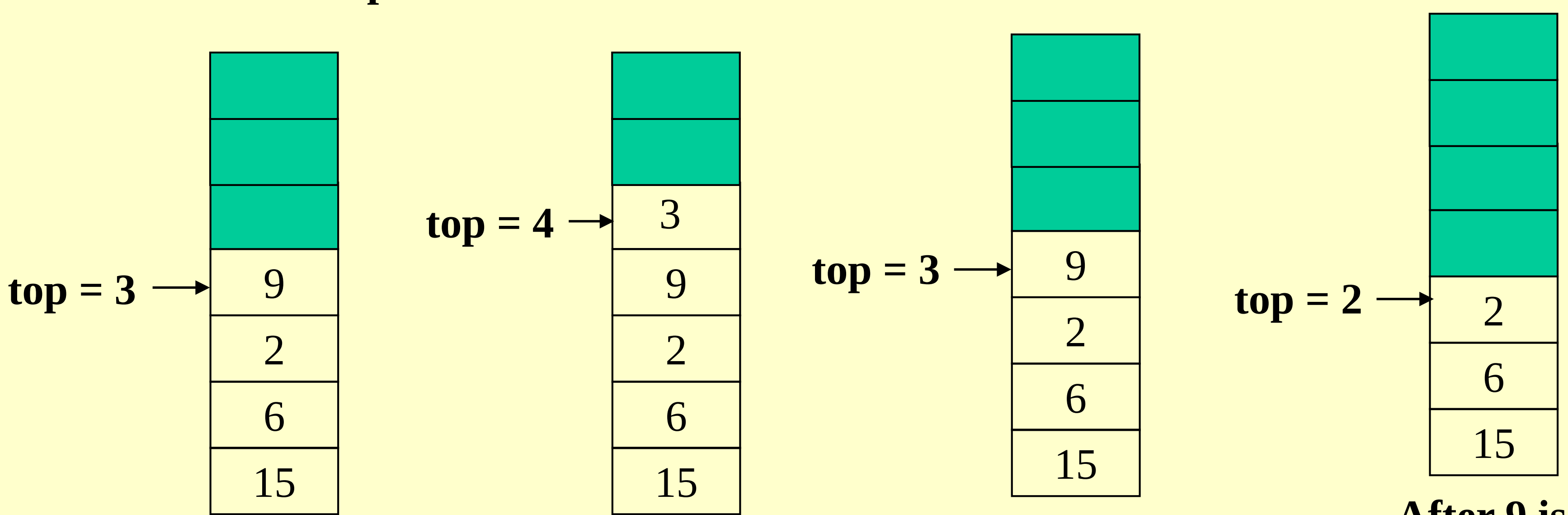
How do we implement stack ADT?

- 2 ways to implement a stack
  - Using an array
  - Using a linked list

Array Implementation of Stacks

- We can implement a stack of at most "N" elements with an array "S" as follows





Initial Stack

After 3 is inserted

After 3 is popped off

After 9 is popped off

Stack Declaration & Operations

```
public class Stack {
  private:
  static int N = 100;  // size of the stack
  int S[];  //Stack elements are positive integers
  int top;  // Current top of the stack

public:
  Stack();
  int Push(int item);
  int Pop();
  int Top();
  bool isEmpty();
  bool isFull();
};
```

Stack Operations: isEmpty, isFull

```
// Constructor
Stack(){
  S = new int[N];
  top = -1;
} // end-Stack

// Returns true if the stack is empty
bool isEmpty(){
  if (top < 0) return true;
  else return false;
} //end-isEmpty

// Returns true if the stack is full
bool isFull(){
  if (top == N-1) return true;
  else return false;
} // end-isFull
```

Stack Operations: Push

```
// Pushes an element to the top of the stack
// Returns 0 on success, -1 on failure
int Push(int newItem){
  if (isFull()){
    // Stack is full. Can’t insert the new element
    System.out.println(“Stack overflow”);
    return -1;
  } //end-if

  top++;
  S[top] = newItem;

  return 0;
} //end-Push
```

Stack Operations: Top

```
// Returns the element at the top of the stack
// If the stack is empty, returns -1
int Top(){
  if (isEmpty()){
    // Stack is empty! Return error
    System.out.println(“Stack underflow”);
    return -1;
  } //end-if

  return S[top];
} //end-Top
```

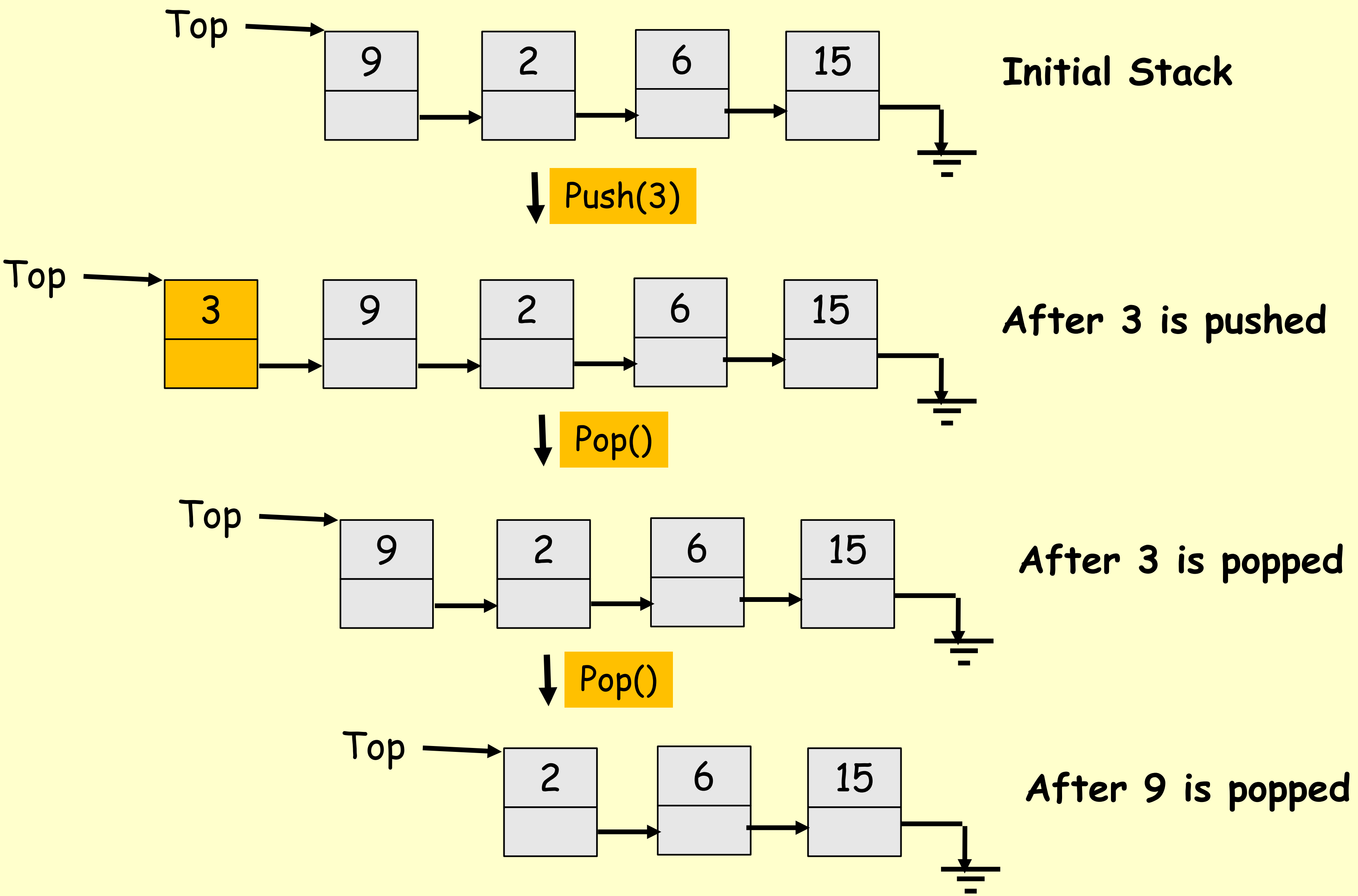
Stack Operations: Pop

```
// Pops the top element of the stack and returns it.
// If the stack is empty, returns -1
int Pop(){
  if (isEmpty()){
    // Stack is empty! Return error
    System.out.println("Stack underflow");
    return -1;
  } //end-if

  int idx = top; // Save current top
  top--;         // Remove the item

  return S[idx];
} //end-Pop
```

Stack Usage Example

```
main(){
  Stack s = new Stack();

  if (s.isEmpty()) println("Stack is empty"); // Empty stack

  s.Push(49);
  s.Push(23);

  println("Top of the stack is: " + s.Pop()); // prints 23
  s.Push(44);
  s.Push(22);

  println("Top of the stack is: " + s.Pop()); // prints 22
  println("Top of the stack is: " + s.Pop()); // prints 44
  println("Top of the stack is: " + s.Top()); // prints 49.
  println("Top of the stack is: " + s.Pop()); // prints 49.

  if (s.isEmpty()) println("Stack is empty"); // Empty stack
} //end-main
```

Linked-List implementation of Stacks



Stack using Linked List: Declarations

```
public struct StackNode {
  public int item;
  public StackNode next;

  StackNode(int e){item=e; next=null;}
};
```

```
/* Stack ADT */
public class Stack {
private:
  StackNode top;    // Stack only has a top

public:
  Stack(){top=null;}
  void Push(int item);
  int Pop();
  int Top();
  bool isEmpty();
};
```

Stack Operations: Push, isEmpty

```
// Pushes an item to the stack
void Push(int item){
  StackNode x = new StackNode(item);
  x.next = top;
  top = x;
} //end-Push

// Returns true if the stack is empty
bool isEmpty(){
  if (top == null) return true;
  else             return false;
} //end-isEmpty
```

Stack Operations: Top

```
// Returns the top of the stack
int Top(){
  if (isEmpty()){
    println("Stack underflow"); // Empty stack.
    return -1; // error
  } //end-if

  return top.item;
} //end-Top
```

Stack Operations: Pop

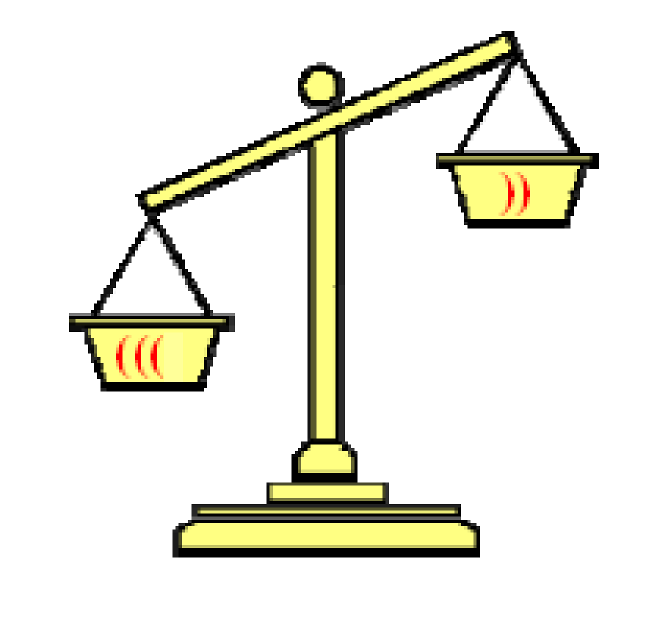
```
// Pops and returns the top of the stack
int Pop(){
  if (isEmpty()){
    println(“Stack underflow”); // Empty stack.
    return -1; // error
  } //end-if

  // Keep a pointer to the current top of the stack
  StackNode tmp = top;

  // Move the top of the stack to the next node
  top = top.next;

  // Return the item
  return tmp.item;
} //end-Pop
```

Stack Usage Example

```
main(){
  Stack s = new Stack();

  if (s.isEmpty()) println("Stack is empty");  // Empty stack

  s.Push(49);
  s.Push(23);

  println("Top of the stack is: " + s.Pop());  // prints 23
  s.Push(44);
  s.Push(22);

  println("Top of the stack is: " + s.Pop());  // prints 22
  println("Top of the stack is: " + s.Pop());  // prints 44
  println("Top of the stack is: " + s.Top());  // prints 49.
  println("Top of the stack is: " + s.Pop());  // prints 49.

  if (s.isEmpty()) println("Stack is empty");  // Empty stack
} //end-main
```

Application of Stacks I: Compilers/Word Processors

- Compilers and Word Processors: Balancing Symbols
  - E.g., 2*(i + 5*(17 – j/(6*k)) is not balanced – ")" is missing
  - Write a Balance-Checker using Stacks and analyze its running time.



Application of Stacks I: Compilers/Word Processors

- Balance-Checker using Stacks:
  1. Make an empty stack and start reading symbols
  2. If input is an opening symbol, Push onto stack
  3. If input is a closing symbol:

     If stack is empty, report error

     Else

       Pop the stack

       Report error if popped symbol is not a matching open symbol
  4. If End-of-File and stack is not empty, report error

- Example: 2*(i + 5*(17 – j/(6*k))

- Run time for N symbols in the input text: O(N)

App. Of Stacks II: Expression Evaluation

- How do we evaluate an expression?
  - 20+2*3+(2*8+5)*4

- Specify the sequence of operations (called a postfix or reverse polish notation)
  - Store 20 in accumulator A1
  - Compute 2*3 and store the result 6 in accumulator A2
  - Compute A1+A2 and store the result 26 in A1
  - Compute 2*8 and store the result in A2
  - Compute 5+A2 and store the result 21 in A2
  - Compute 4*A2 and store the result 84 in A2
  - Compute A1+A2 and store the result 110 in A1
  - Return the result, 110, stored in A1

- 20 2 3 * + 2 8 * 5 + 4 * + (postfix notation)

App. Of Stacks II: Expression Evaluation

- The advantage of the postfix notation is that the postfix notation clearly specifies the sequence of operations without the need for paranthesis
  - Therefore it is much easier to evaluate a postfix expression than an infix expression

App. Of Stacks II: Expression Evaluation

- It turns out we can easily convert an infix expression to postfix notation using a stack

(1) When an operand is encountered, output it

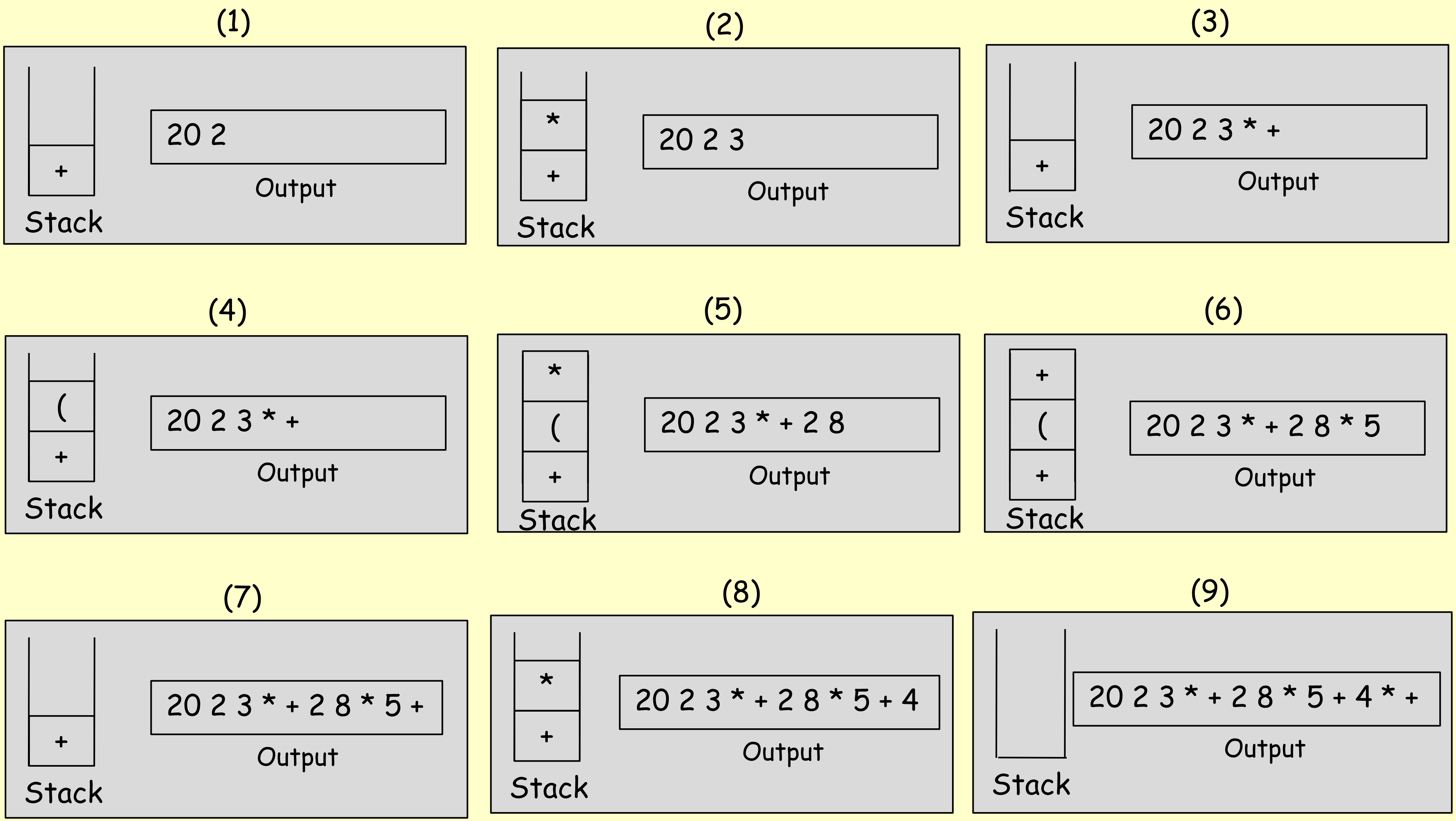
(2) When '(' is encountered, push it

(3) When ')' is encountered, pop all symbols off the stack until '(' is encountered

(4) When an operator is encountered (+, -, *, /), pop symbols off the stack until you encounter a symbol that has lower priority

(5) Push the encountered operator to the stack

Steps in converting the infix expression 20 + 2*3 + (2*8+5) *4 to postfix notation



Evaluating a postfix expression

- We can also use a stack to evaluate an expression specified in postfix notation

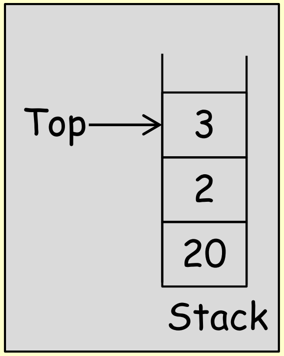
(1) When an operand is encountered, push it to the stack

(2) When an operator is encountered, pop 2 operands off the stack, compute the result and push the result back to the stack

(3) When all symbols are exhausted, the result will be the last symbol in the stack

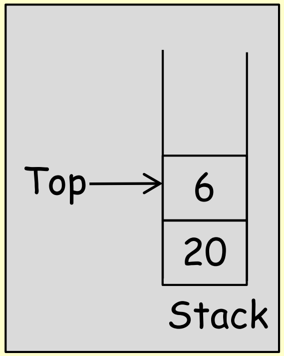
Steps in evaluating the postfix expression:
20 2 3 * + 2 8 * 5 + 4 * +

(1)



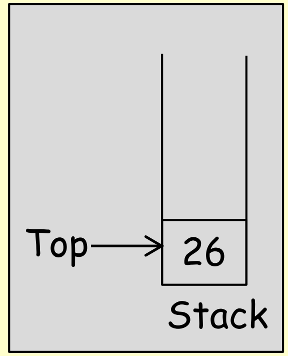
Push: 20, 2, 3

(2)



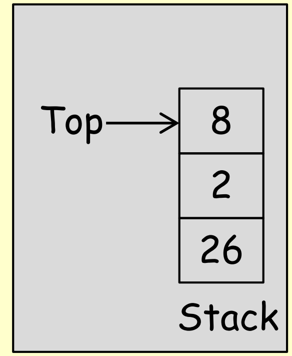
Pop 3, 2
Compute 3 * 2
Push 6

(3)



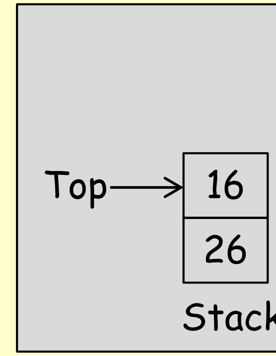
Pop 6, 20
Compute 6 + 20
Push 26

(4)



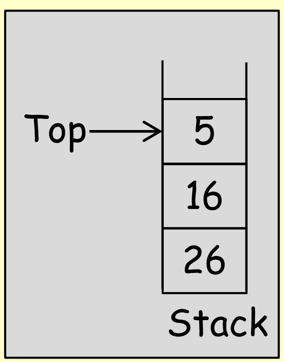
Push 8, 2

(5)



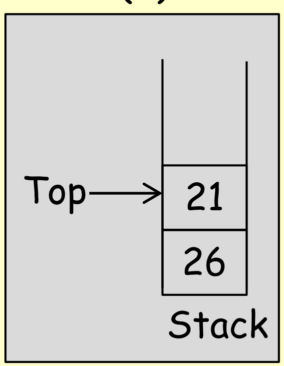
Pop 8, 2
Compute 8 * 2
Push 16

(6)



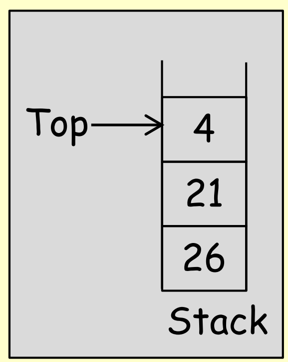
Push: 5

(7)



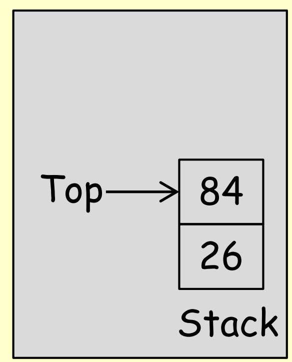
Pop 5, 16
Compute 5 + 16
Push 21

(8)



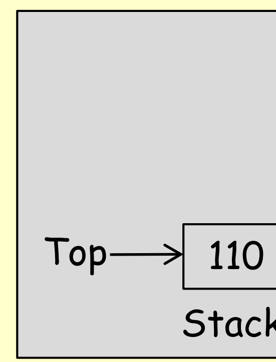
Push 4

(9)



Pop 4, 21
Compute 4 * 21
Push 84

(10)



Pop 84, 26
Compute 84 * 26
Push 110