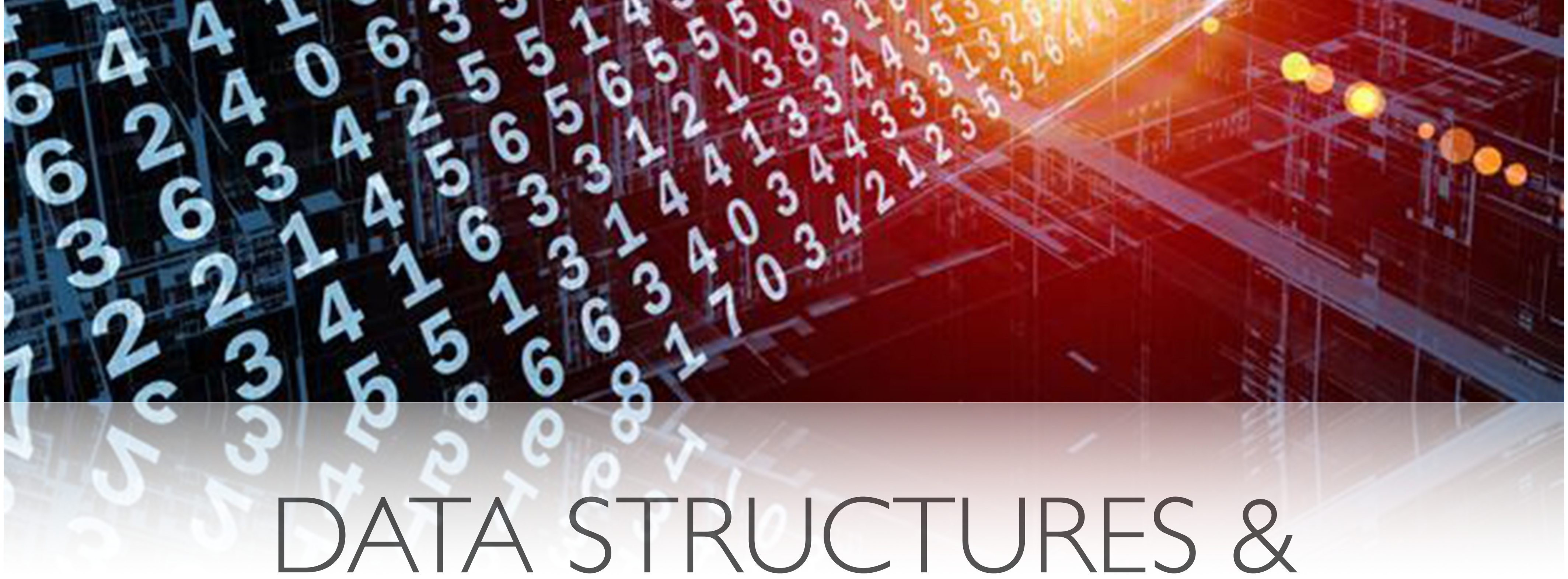


DATA STRUCTURES & ALGORITHMS

COMP H3025

Lecture 4: Stacks

Lecturer: Stephen Sheridan

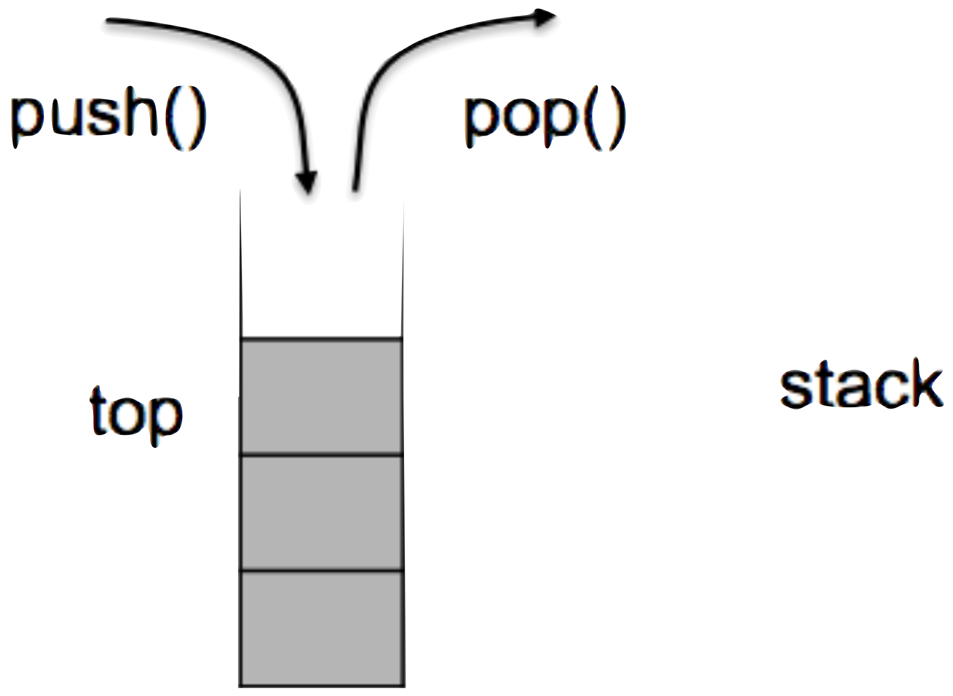
STACKS

- The term **stack** is one that is familiar to us from everyday life.
  - A *stack* dishes waiting to be washed
  - A *stack* of books on your desk
  - A *stack* of assignments waiting to be corrected

STACKS

- An important property of a **stack** is that it operates a **last in, first out** or **LIFO** policy.
- This means that the last item placed on the stack will be the first to be removed. So the item at the top of the stack is the one that must be removed next.
- The ADT **stack** is appropriate for many real world problems where the **LIFO** property holds true.
- **NOTE**: operations on a **stack** only involve the **top** item on the stack.

STACKS



STACK OPERATIONS

- Typical stack operations include:

```
public boolean isEmpty()
//determines if the stack is empty

public void push(Object newItem) throws StackException
// adds newItem to the top of the stack
// throws StackException if the insertion is not successful

public Object pop() throws StackException
// retrieves and removes the top of the stack
// throws StackException if deletion not successful

public void popAll() throws StackException
// removes all items from the stack

public Object top() throws StackException
// retrieves the top of the stack
// retrieval does not change the stack in any way
// throws StackException if retrieval not successful
```

EXAMPLE PROBLEM (BACKSPACE TXT)

- When you type a piece of text on a keyboard, you are likely to make mistakes. If you use the BACKSPACE key to correct these mistakes, each BACKSPACE erases the previously entered character.

- Consecutive BACKSPACES are applied in sequence and therefore erase several characters.

  - If you type in the line: **abcc←ddde←←←ef←fg**

  - Where ← represents the BACKSPACE key, the corrected input should be **abcdefg**

EXAMPLE PROBLEM (BACKSPACE TXT)

- Often times, the specification of an ADT can emerge while trying to solve a problem.
- In designing a solution to this problem you must eventually make a decision on how to store the input line.
- If we store the input line as a **stack**, we could ***push*** each normal character on to the stack and when we encounter a BACKSPACE, we could ***pop*** the previous character off the stack.
- When we reach the end of the input line our stack should contain the corrected text.

EXAMPLE PROBLEM (BACKSPACE TXT)

- A potential problem exists with this solution and that is if a BACKSPACE is entered and the stack is empty.

- If this situation occurs, we can:

  1. Have the program terminate with an error message

  2. Have the program ignore the BACKSPACE and continue

- We will assume the **second** option is more reasonable for our situation.

EXAMPLE PROBLEM (BACKSPACE TXT)

abcc←ddde←←←ef←fg

								e								g
							d	d	d				f		f	f
						d	d	d	d	d		e	e	e	e	e
			c		d	d	d	d	d	d	d	d	d	d	d	d
		c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a

STACKS: EXAMPLE PROBLEM (A)

- Pseudocode solution using that uses a **stack**

```
readAndCorrect()
// reads input line & returns the corrected version as a stack
// for each character read, either enters it into the stack or,
// if "←", corrects the contents of the stack

Stack stack = new Stack()
while (newChar is not the end of line symbol)
{
   if (newChar is not "←")
   {
      stack.push(newChar)
   }
   else if (!stack.isEmpty())
   {
      oldChar = stack.pop()
   }// end if

   read newChar

} //end while
return stack
```

EXAMPLE PROBLEM (BALANCED BRACKETS)

- Java uses curly brackets (braces) "{" and "}" to delimit groups of statements, blocks of code, and method's body.

- If you treat a Java program as a string of characters, you can use a stack to verify that a program contains balanced brackets of the correct type.

- For example the brackets in the following string are **balanced**:

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

- The brackets in this string are **not balanced**:

  abc**{**def**}}{**ghij**{**kl**}**m

EXAMPLE PROBLEM (BALANCED BRACKETS)

- The solution to the balanced bracket problem requires that you keep track of each unmatched "{" and discard one each time you encounter a "}".

- Once way to perform this task is to *push* each "{" encountered onto a stack and *pop* one off each time you encounter a "}".

- If the input contains **balanced brackets** the stack should be **empty** at the end of the input.

EXAMPLE PROBLEM (BALANCED BRACKETS)

- A first attempt at coding this solution might look as follows:

```
while (not at the end of the input)
{
    if (the next character is a '{')
    {
        stack.push('{')
    }
    else if (the character is a '}')
    {
        openBrace = stack.pop()
     } //end if
} //end while
```

EXAMPLE PROBLEM (BALANCED BRACKETS)

- While this code does track the brackets, it does not check if the brackets are balanced.
- To verify this, you must check to see whether the stack is empty before *popping* from it.
- If its empty, you terminate the loop and report that the string is not balanced.
- We must also verify that the stack is empty when the end of the input has been reached.

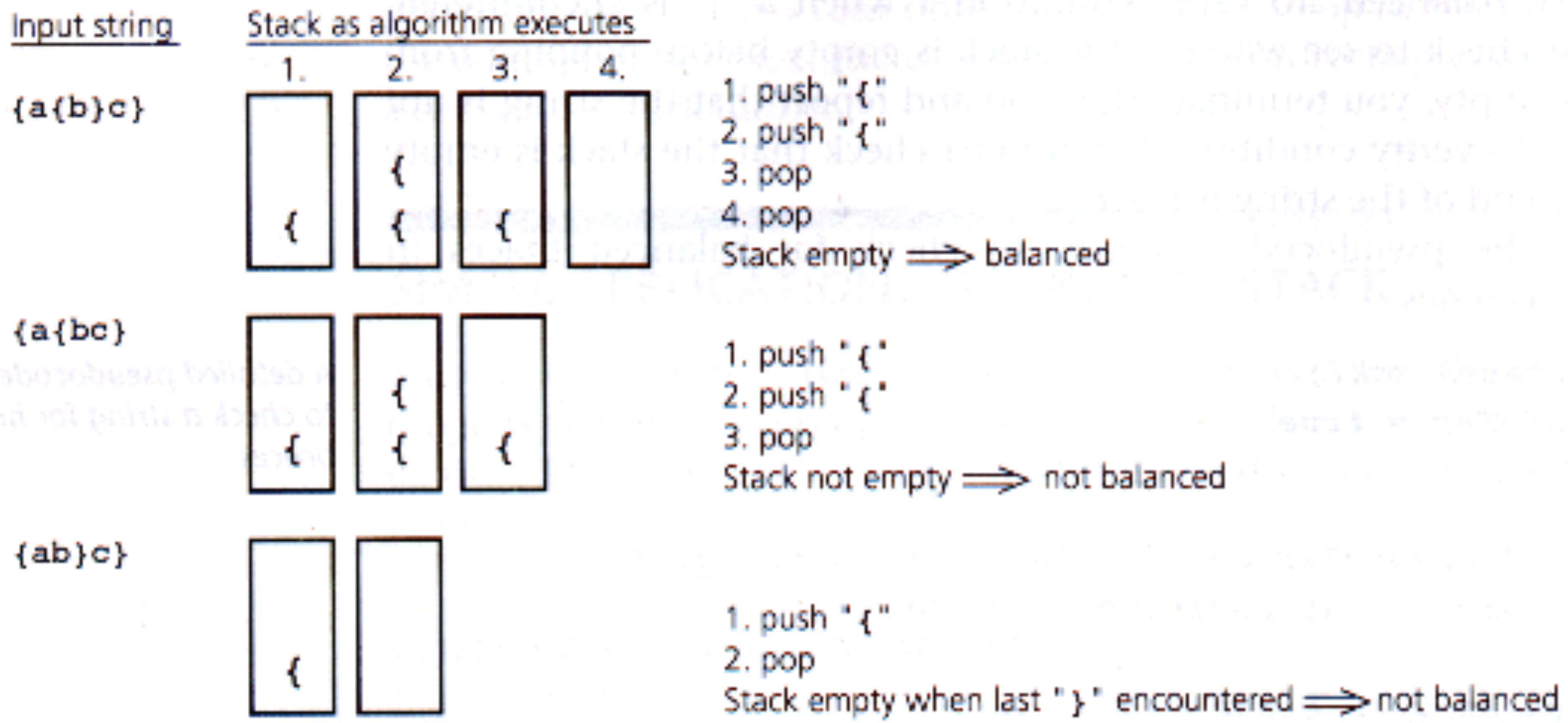
EXAMPLE PROBLEM (BALANCED BRACKETS)

```
Stack stack = new Stack()
balancedSoFar=true
k=0
while (balancedSoFar and k < length of input)
{
    ch = character at position k in input
    k++
    if (ch is '{' )
    {
        stack.push('{') // push an open brace
    }
    else if (ch is '}')
    {
        if (!stack.isEmpty() )
        {
            openBrace = stack.pop()  //pop a matching open brace
        }
        else
        {
            balancedSoFar = false // no matching open brace
        }
    }
}

if (balancedSoFar and stack.isEmpty() )
{
    input has balanced braces
}
else
{
    input does not have balanced braces
} //end if
```

EXAMPLE PROBLEM (BALANCED BRACKETS)

- The diagram bellow shows the stacks that result when this algorithm is applied to several simple examples.



STACK EXCEPTIONS

- The pseudocode for the stack ADT ignores exceptions. However, the implementation of the stack ADT should not.

- Any stack implementation should either take precautions to avoid an exception or provide try/catch blocks to handle possible exceptions.

- Suppose that **pop()** or **push()** throws StackException, how do you interpret this event?

- For example, lets refine the **pop()** section of the previous pseudocode.

STACK EXCEPTIONS

```
else if (ch is '}')
{
   try
   {
      //pop a matching open brace
      openBrace = stack.pop()

   }
   catch (StackException e)
   {
      balancedSoFar = false
   }

} //end if
```

- **NOTE**: the **push()** operation can fail for implementation-dependant reasons.
- For example, in an array based implementation, push() should throw a StackException if the array is full.

STACK ADT IMPLEMENTATION

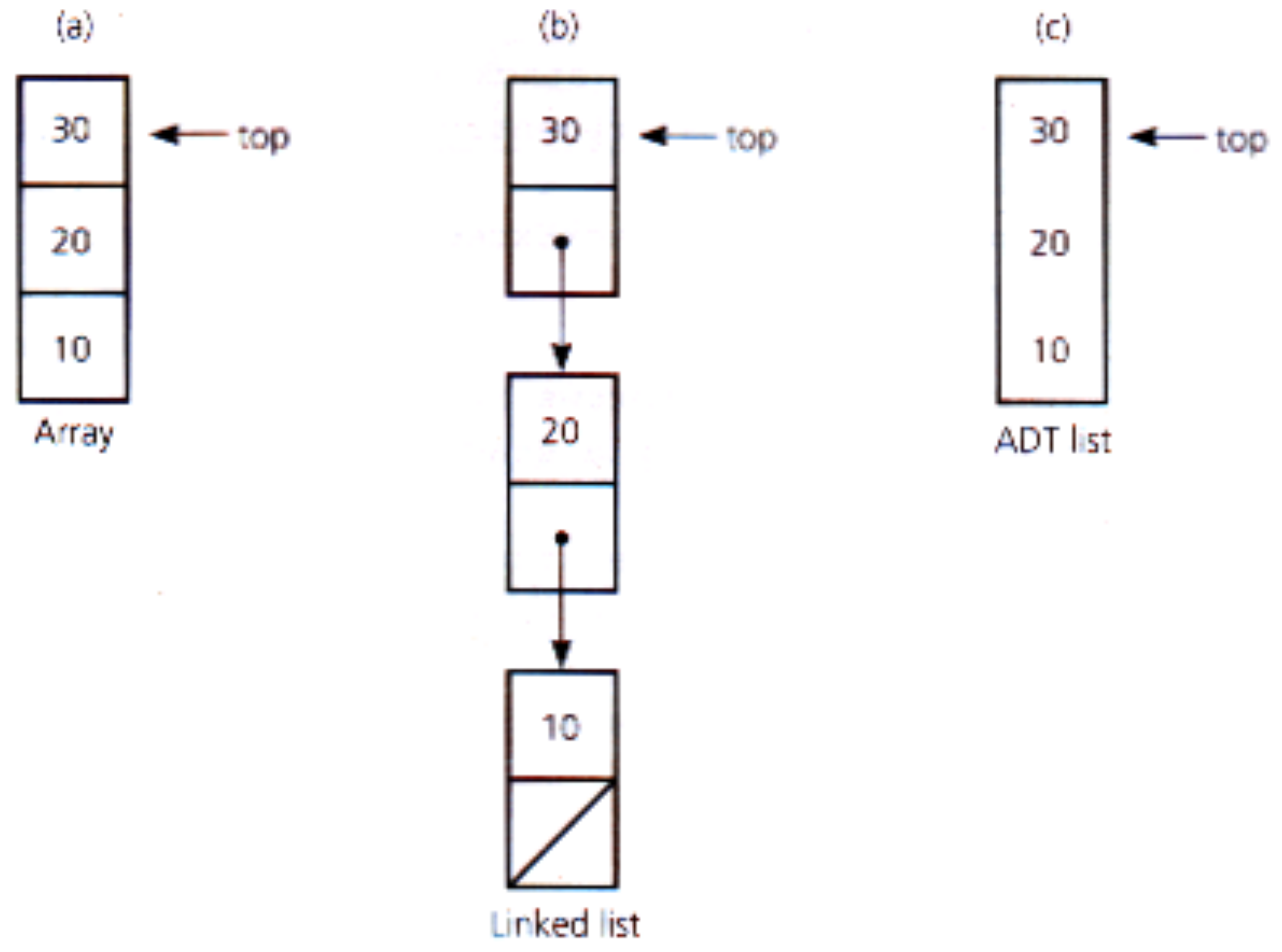
- At this point in the course we actually have three options when it comes to implementing the ADT stack.

  A) Array based implementation

  B) Referenced based linked list implementation

  C) ADT List implementation

STACK ADT IMPLEMENTATION



NOTE: the stack interface provides a common contract for all three implementations of the stack ADT.

TODO - WEEK 4

- Assignment on MOODLE!!