CS1102: Data Structures and Algorithms

Part 4

Stack

Zoltan KATO

S16 06-12

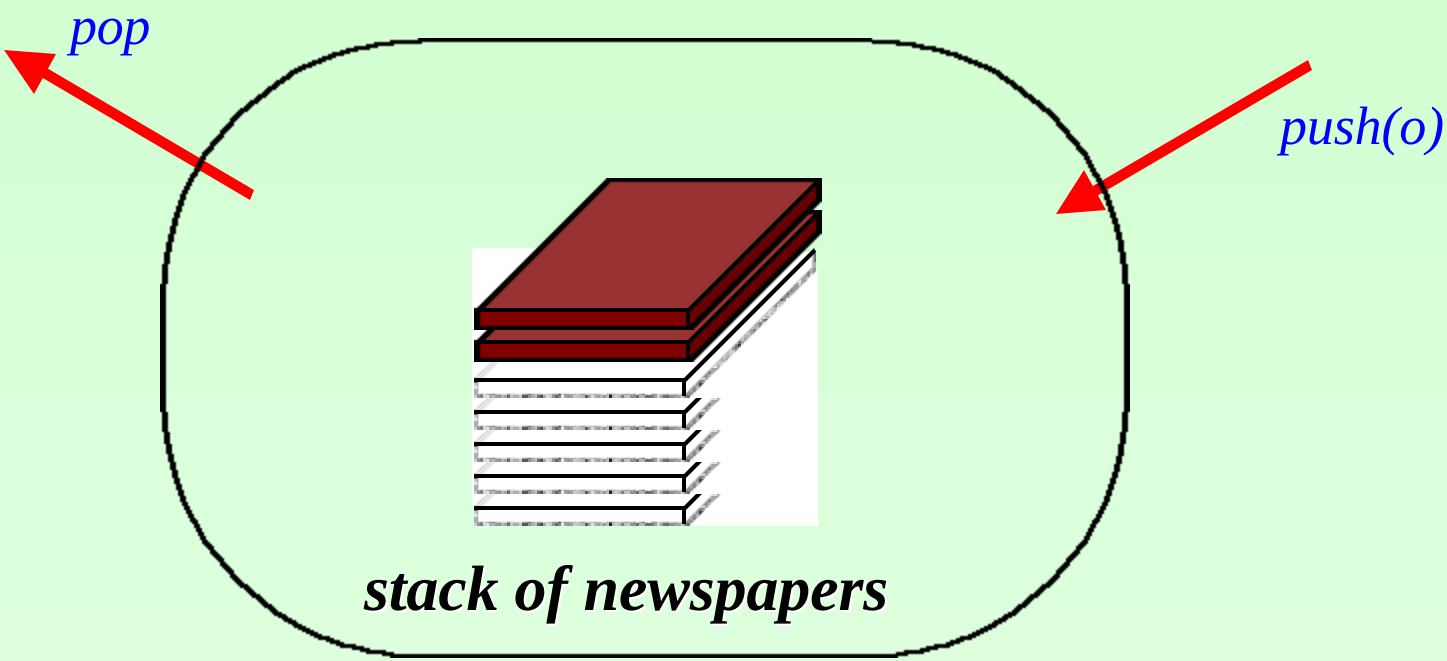
Adopted from Chin Wei Ngan's cs1102 lecture notes

Stacks

- What is a Stack?
- Stack ADT
- Applications
  - Line Editing
  - Bracket Matching
  - Postfix Calculation
- Implementation of Stack (Linked-List)
- Implementation of Stack (Array)

What is a Stack?

- Stacks can be implemented efficiently and are very useful in computing.
- Stacks exhibit the LIFO behavior.



Stack ADT Interface

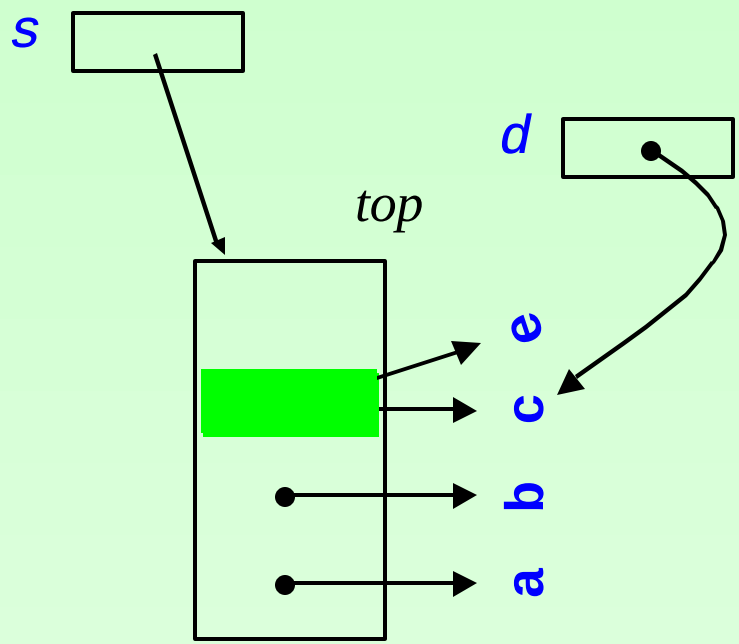
We can use Java Interface to specify Stack ADT Interface

```
interface Stack {

 boolean isEmpty();                    // return true if empty

 void push(Object o);                  // insert o into stack

 void pop() throws Underflow;          // remove most recent item

 Object top() throws Underflow;        // retrieve most recent item

 Object topAndPop() throws Underflow;  // return & remove most recent item

}
```

Sample Code

Stack ADT

```
Stack s = makeStack();
s.push("a");
s.push("b");
s.push("c");
d=s.top();
s.pop();
s.push("e");
s.pop();
```



Applications

Many application areas use stacks:

- *line editing*
- *bracket matching*
- *postfix calculation*
- *function call stack*

Line Editing

A line editor would place the characters read into a buffer but may use a backspace symbol (denoted by ←) to do error correction.

Refined Task

- read in a line
- correct the errors via backspace
- print the corrected line in reverse

<u>Example:</u>

Input : `abc_defgh~~h~~←2klp~~q~~q~~r~~←←wxyz`

Corrected Input : `abc_defg2klpwxyz`

Reversed Output : `zyxwplk2gfed_cba`

Informal Procedure

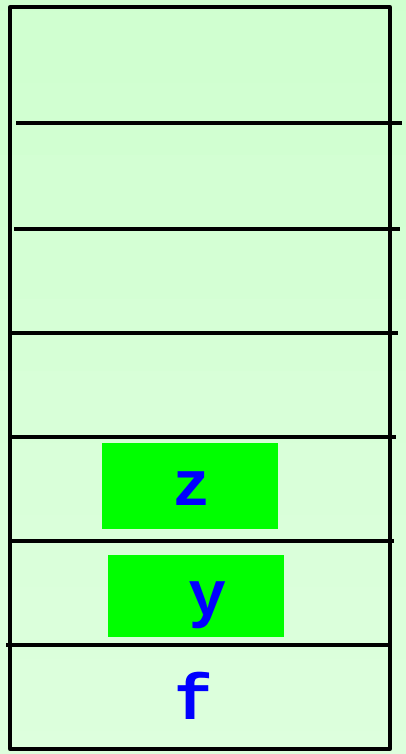
Line Editing

- Initialise a new stack.
- For each character read:
  - if it is a backspace, *pop out last char entered*
  - if not a backspace, *push the char into stack*
- To print in reverse, pop out each char for output.

Input : `fgh←r←←yz`

Corrected Input : `fyz`

Reversed Output : `zyf`



Stack

Code

Line Editing

```
Stack s = StackLL.makeStack();
while (not end of line) do
 { read a new char ch
   if (ch != '←')
     {s.push(new Character(ch));}
   else {if (!s.isEmpty()) s.pop();}
  };
// print char in reverse order
while (!s.isEmpty()) do
 { d = s.topAndPop();
   print out d ;
 }
```

Bracket Matching Problem

Ensures that pairs of brackets are properly matched.

- An Example: `{a,(b+f[4])*3,d+f[5]}`

- Bad Examples:

```
(..)..)          // too many closing brackets

(..(..)          // too many open brackets

[..(..]..)       // mismatched brackets
```

Informal Procedure

Bracket Matching

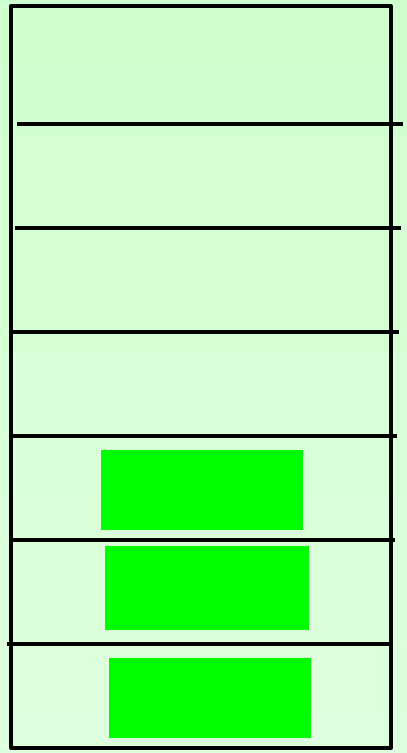
Initialise the stack to empty.

For every char read.

- if open bracket then *push onto stack*
- if close bracket, then
  - *topAndPop from the stack*
  - if doesn't match then *flag error*
- if non-bracket, *skip the char read*

Example

```
{a,(b+f[4])*3,d+f[5]}
```



Stack

Code — Bracket Matching

```
public static boolean balanced() {
 Stack s = StackAr.makeStack();
 boolean failflag = false;
 while ((not end of line) && !failflag) do
  { read a new char ch
    if ((ch == '(') || (ch == '{') ||  (ch == '['))
        {s.push(new Character(ch));}
    else if ((ch == ')') || (ch == '}') ||  (ch == ']'))
                {if (s.isEmpty()) { failflag = true;
                                    status = "No matching open brace"; }
                 else {d = (Character)s.topAndPop();
                       if !(match(ch,d.charValue())
                          { failflag = true;
                            status="Wrong pair of matching brace";
                          }
    else { // do nothing }
   };
  if failflag return false;
  else {if (!s.isempty()) {status = "Too many open parentheses";
                          return false; }
        else return true;};
 };
```

Postfix Calculator

Computation of arithmetic expressions can be efficiently carried out in Postfix notation with the help of a stack.

Infix	-	`arg1` **`op`** `arg2`
Prefix	-	**`op`** `arg1 arg2`
Postfix	-	`arg1 arg2` **`op`**

postfix

infix

`2*3+4`

`(2*3)+4` → `2 3 * 4 +`

`2*(3+4)` → `2 3 4 + *`

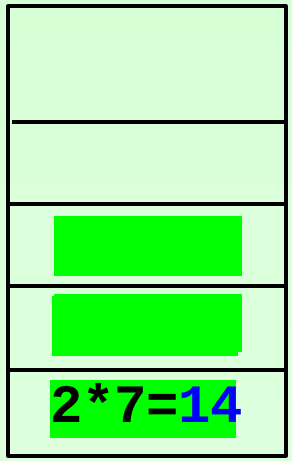
Informal Procedure *Postfix Calculator*

Initialise stack
For each item read.
    If it is an operand,
        push on the stack
    If it is an operator,
        pop arguments from stack;
        perform operation;
        push result onto the stack

Expr	
2	`s.push(2)`
3	`s.push(3)`
4	`s.push(4)`

+	`arg2=s.topAndPop()`
	`arg1=s.topAndPop()`
	`s.push(arg1+arg2)`

*	`arg2=s.topAndPop()`
	`arg1=s.topAndPop()`
	`s.push(arg1*arg2)`



Stack

Code

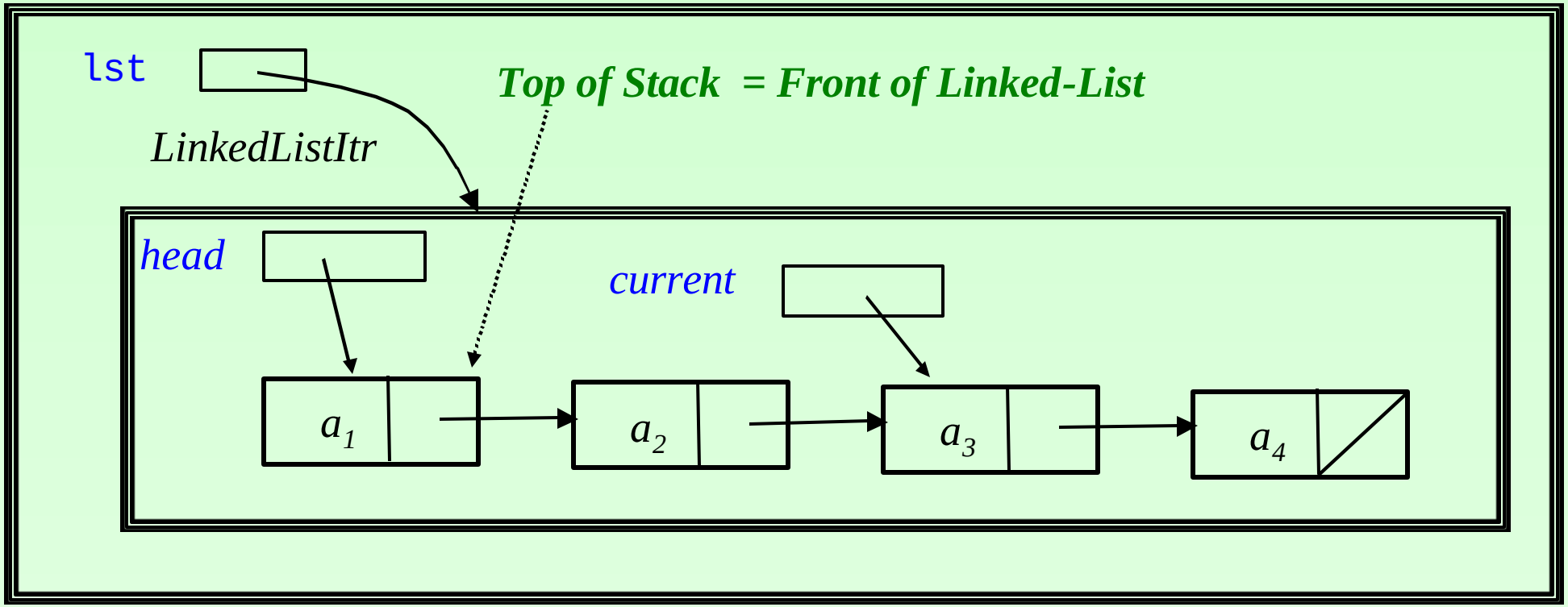
Postfix Calculator

```
Stack s = StackAr.makeStack();
while (not end of line) do
 { read a new item ch
   if (isOperand(ch)) {s.push(valueof(ch));
   else {  arg2 = (Integer) s.topAndPop() ;
           arg1 = (Integer) s.topAndPop() ;
           Integer res = compute(ch,arg1,arg2);
           s.push(res);   };
  };
```

Implementation of Stack (Linked-List)

Can use `LinkedListItr` as implementation of stack

StackLL



Code Implementation of Stack (Linked-List)

```
Class StackLL implements Stack {

 private LinkedListItr lst;

 public StackLL() { lst = new LinkedListItr(); }

 public Stack makeStack() { return new StackLL(); }

 public boolean isEmpty()                    // return true if empty
  { return lst.isEmpty();  };

 public void push(Object o)                  // add o into the stack
  { lst.addHead(o); }

 public void pop() throws Underflow          // remove most recent item
  { try {lst.deleteHead();}
    catch (ItemNotFound e)
     {throw new Underflow("pop fails - empty stack")};
  }
```

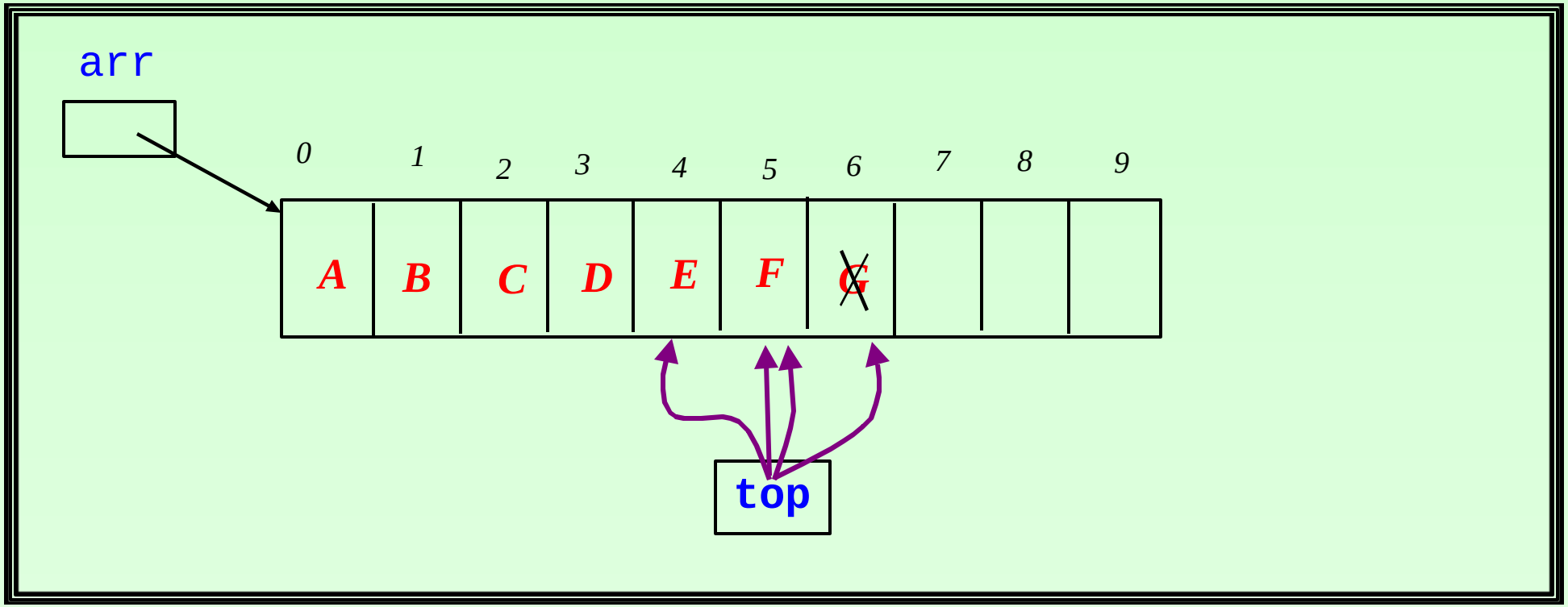
Code Implementation of Stack (Linked-List)

```
public Object top() throws Underflow;       // retrieve most recent item
 { try { lst.first();
         return lst.retrieve();
   } catch (ItemNotFound e)
       {throw new Underflow("top fails - empty stack");};
 }

public Object topAndPop() throws Underflow;
                                     // return & remove most recent item
 { try {  lst.first();
          Object p=lst.retrieve();
          lst.deleteHead();
          return p;
   } catch (ItemNotFound e)
       {throw new Underflow("topAndPop fails - empty stack");};
 }
```

Implementation of Stack (Array)

Can use `Array` with a `top` index pointer as an implementation of stack

StackAr



Code — Implementation of Stack (Array)

```
class StackAr implements Stack {

 private Object [] arr;
 private int top;
 private int maxSize;
 private final int initSize = 1000;
 private final int increment = 1000;

 public StackAr() { arr = new Object[initSize];
                    top = -1;
                    maxSize=initSize }

 public Stack makeStack() { return new StackAr(); }

 public boolean isEmpty()
  { return (top<0);  }

 private boolean isFull()
  { return (top>=maxSize);  }
```

allows resizing of array

Code — Implementation of Stack (Array)

```
public void push(Object o)        // insert o
 { top++;
   if (this.isFull()) this.enlargeArr() ;
   arr[top]=o; }

private void enlargeArr()         // enlarge the array
 {int newSize = maxSize+increment;
  Object [] barr = new Object[newSize];
  for (int j=0;j<maxSize;j++) {barr[j]=arr[j];};
  maxSize = newSize;   arr = barr;
 }
```

Code — Implementation of Stack (Array)

```
public void pop() throws Underflow
  { if (!this.isEmpty()) {top--;}
    else {throw new Underflow("pop fails - empty stack");};
  }

public Object top() throws Underflow
 { if (!this.isEmpty()) {return arr[top];}
   else {throw new Underflow("top fails - empty stack");};
 }

public Object topAndPop() throws Underflow
 { if (!this.isEmpty()) { Object t = arr[top];
                          top--;
                          return t;
                          };
  else {throw new Underflow("top&pop fails - empty stack");};
 }
```