Lecture 4-1. Stacks

Sunghyun Cho
Professor
Division of Computer Science
chopro@hanyang.ac.kr

HANYANG UNIVERSITY

Review

- Data Structure
    - Data storing structure + access methods
- Asymptotic Analysis
    - Compare the performance of each data structures and algorithms
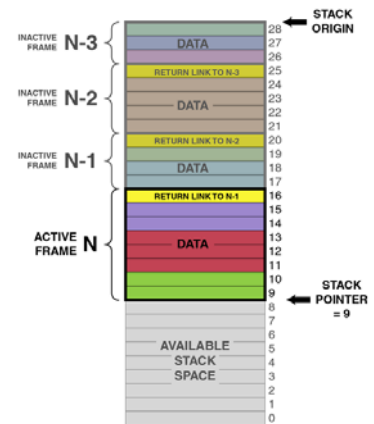   - Pseudo-code → Mathematic Exp. → Big O
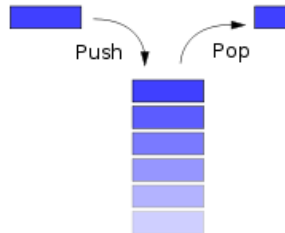- List
    - The simplest and fundamental data structure in CS
    - Array based list
    - Linked list
    - O(n) for add and remove method

HANYANG UNIVERSITY

- Definition of Stack

- Implementation of Stack

- Usages of Stack

HANYANG UNIVERSITY

Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure

- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

HANYANG UNIVERSITY

The Stack ADT

- The Stack ADT stores arbitrary objects
- **<u>Insertions and deletions follow the last-in first-out scheme</u>**
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element

- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

Example 6.3: Stack Operations (textbook p. 209)

Stack Interface in Java

- Java interface corresponding to our Stack ADT
- Requires the definition of class EmptyStackException
- Different from the built-in Java class java.util.Stack

```java
public interface Stack<E> {

    public int size();

    public boolean isEmpty();

    public E top()
         throws EmptyStackException;

    public void push(E element);

    public E pop()
          throws EmptyStackException;
}
```

Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an "**Exception**"

- Exceptions are said to be "thrown" by an operation that cannot be executed

- In the Stack ADT, operations pop and top cannot be performed if the stack is empty → Exception Cases

- Attempting the execution of pop or top on an empty stack throws an EmptyStackException

---

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine

- Indirect applications
  - Auxiliary data structure for algorithms
 - Component of other data structures

Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack
- Allows for recursion

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

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

bar(int m) {
    ...
}
```

bar
PC = 1
m = 6

foo
PC = 3
j = 5
k = 6

main
PC = 2
i = 5
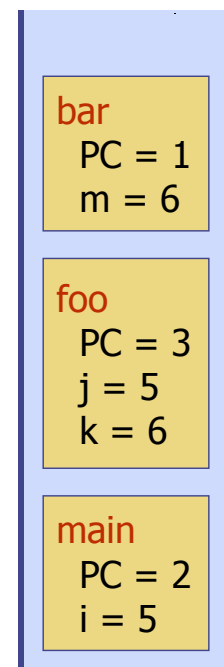
Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable t keeps track of the index of the top element

Algorithm *size()*
   return $t + 1$

Algorithm *pop()*
  if *isEmpty()* **then**
    throw *EmptyStackException*
  else
    $t \leftarrow t - 1$
    return $S[t + 1]$

      => S[t]

S

```
0   1   2                          t
```

Array-based Stack (cont.)

❑ The array storing the stack elements may become full

❑ A push operation will then throw a FullStackException

  ▪ Limitation of the array-based implementation

  ▪ Not intrinsic to the Stack ADT

> **Algorithm** *push(o)*
>   **if** $t = S.length - 1$ **then**
>   **throw** *FullStackException*
>   **else**
> $t \leftarrow t + 1$
> $S[t] \leftarrow o$

S

0 1 2

t

Performance and Limitations

▣ Performance
  – Let n be the number of elements in the stack
  – The space used is $O(N)$ $(N \neq n)$
  – Each operation runs in time $O(1)$
    n/2까지 제거됐다가 하는 경우 스택을 사용하면 안됨

▣ Limitations
  – **The maximum size of the stack** must be defined a priori and cannot be changed
  – Trying to push a new element into a full stack causes **an implementation-specific exception**

Array-based Stack in Java

```java
public class ArrayStack<E>
    implements Stack<E> {

  // holds the stack elements
  private E S[ ];

  // index to top element
  private int top = -1;

  // constructor
  public ArrayStack(int capacity) {
    S = (E[]) new Object[capacity]);
  }
}
```

```java
public E pop()
    throws EmptyStackException {
  if isEmpty()
    throw new EmptyStackException
        ("Empty stack: cannot pop");
  E temp = S[top];
  // facilitate garbage collection:
  S[top] = null;
  top = top – 1;
  return temp;
}

…  (other methods of Stack interface)
```

Example use in Java

```java
public class Tester {

  // … other methods
  public intReverse(Integer a[]) {
    Stack<Integer> s;
    s = new ArrayStack<Integer>();

    … (code to reverse array a) …
  }
}
```

```java
public floatReverse(Float f[]) {
  Stack<Float> s;
  s = new ArrayStack<Float>();

  … (code to reverse array f) …
}
```

Linked List based Stack

- Using linked list to implement the stack ADT
  - use singly linked list
  - need to decide if the top of the stack is at the head or at the tail of the list
    - to insert and delete elements within constant time, **the head should be the top of the stack**
  - in order to perform operation size in constant time, we keep track of the current number of elements in an instance variable

- JAVA code
  - refer to Code Fragment 6.4 (textbook p. 215)

Performance

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$

Parentheses Matching

■ Each ")", "{", or "[" must be paired with a matching ")", "}", or "["
- correct: ( )(( )){([( )])}
- correct: ((( )(( )){([( )])}
- incorrect:)(( )){([( )])}
- incorrect: ({[ ])}
- incorrect: (

Parentheses Matching Algorithm

Algorithm ParenMatch(*X,n*):
Input: An array *X* of *n* tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number
Output: **true** if and only if all the grouping symbols in *X* match
Let *S* be an empty stack
for *i*=0 to *n*-1 **do**
    if *X*[*i*] is an **opening grouping symbol then**
        S.**push**(*X*[*i*])
    else if *X*[*i*] is a **closing grouping symbol then**
        if *S*.isEmpty() **then**
            return false {nothing to match with}
        if *S*.**pop**() does not match the type of *X*[*i*] **then**
            return false {wrong type}
if *S*.isEmpty() **then**
    return true {every symbol matched}
else return false {some symbols were never matched}

HTML Tag Matching

◆ For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

The Little Boat

The storm tossed the little boat
like a cheap sneaker in an old
washing machine. The three
drunken fishermen were used to
such treatment, of course, but not
the tree salesman, who even as
a stowaway now felt that he had
overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

HANYANG UNIVERSITY

Tag Matching Algorithm (in Java)

```java
import java.io.*;
import java.util.Scanner;
import net.datastructures.*;
/** Simplified test of matching tags in an HTML document. */
public class HTML {
  /** Strip the first and last characters off a <tag> string. */
  public static String stripEnds(String t) {
    if (t.length() <= 2) return null;              // this is a degenerate tag
    return t.substring(1,t.length()-1);
  }
  /** Test if a stripped tag string is empty or a true opening tag. */
  public static boolean isOpeningTag(String tag) {
    return (tag.length() == 0) || (tag.charAt(0) != '/');
  }
}
```

refer to Code Fragment 6.6 (textbook p. 218)

HANYANG UNIVERSITY

Tag Matching Algorithm (cont.)

```java
/** Test if stripped tag1 matches closing tag2 (first character is '/'). */
public static boolean areMatchingTags(String tag1, String tag2) {
   return tag1.equals(tag2.substring(1)); // test against name after '/'
}
/** Test if every opening tag has a matching closing tag. */
public static boolean isHTMLMatched(String[] tag) {
  Stack<String> S = new NodeStack<String>();     // Stack for matching tags
  for (int i = 0; (i < tag.length) && (tag[i] != null); i++) {
    if (isOpeningTag(tag[i]))
    S.push(tag[i]); // opening tag; push it on the stack
    else {
    if (S.isEmpty())
      return false;               // nothing to match
    if (!areMatchingTags(S.pop(), tag[i]))
      return false;     // wrong match
    }
    }
  if (S.isEmpty()) return true; // we matched everything
  return false; // we have some tags that never were matched
}
```

Tag Matching Algorithm (cont.)

```java
public final static int CAPACITY = 1000; // Tag array size
/* Parse an HTML document into an array of html tags */
public static String[] parseHTML(Scanner s) {
  String[] tag = new String[CAPACITY]; // our tag array (initially all null)
  int count = 0;                               // tag counter
  String token;                                // token returned by the scanner s
  while (s.hasNextLine()) {
    while ((token = s.findInLine("<[^>]*>")) != null)  // find the next tag
      tag[count++] = stripEnds(token); // strip the ends off this tag
    s.nextLine(); // go to the next line
  }
  return tag; // our array of (stripped) tags
}
public static void main(String[] args) throws IOException { // tester
  if (isHTMLMatched(parseHTML(new Scanner(System.in))))
    System.out.println("The input file is a matched HTML document.");
  else
    System.out.println("The input file is not a matched HTML document.");
  }
}
```

Evaluating Arithmetic Expressions

14 – 3 * 2 + 7 = (14 – (3 * 2)) + 7

Operator precedence
    * has precedence over +/–

Associativity
    operators of the same precedence group
    evaluated from left to right
    Example: (x – y) + z rather than x – (y + z)

Idea: push each operator on the stack, but first pop and
perform higher and *equal* precedence operations.

Algorithm for Evaluating Expressions

Running Time: O(1)

Two stacks:
- ❑   opStk holds operators
- ❑   valStk holds values
- ❑   Use $ as special "end of input" 텍스트
  token with lowest precedence

Algorithm doOp()
    x ← valStk.pop();
    y ← valStk.pop();
    op ← opStk.pop();
    valStk.push(y **op** x)

Algorithm repeatOps(refOp):

  while (valStk.size() > 1 ∧
        prec(refOp) ≤
        prec(opStk.top()))
    doOp()

Algorithm EvalExp()
    Input: a stream of tokens representing
        an arithmetic expression (with
        numbers)
    Output: the value of the expression

while there's another token z
    if isNumber(z) **then**
        valStk.push(z)
    else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()

14 ≤ 4 − 3 * 2 + 7

Operator ≤ has lower precedence than +/−