CSCI-UA 480.4: APS
Algorithmic Problem Solving

Linear Data Structures

Instructor: Joanna Klukowska

created based on materials for this class by
Bowen Yu and materials shared by the authors of
the textbook Steven and Felix Halim

Questions

- homework 1 questions ?

Basic Linear Data Structures, Continuted

- list
  - array
  - linked
- stack
- queue

Stacks

Stack (first in last out, FILO)

- implementation provided by buil-in classes:
  - `stack` in C++ STL
  - `Stack` in Java

Stack (first in last out, FILO)

- implementation provided by buil-in classes:
  - `stack` in C++ STL
  - `Stack` in Java
- operations performed
  - add/push O(1) - adds to the top
  - remove/pop O(1) - removes from the top
  - top O(1) - access the element on the top (optional)
  - empty O(1) - determine if the stack is empty (optional)

Stack (first in last out, FILO)

- implementation provided by buil-in classes:
  - `stack` in C++ STL
  - `Stack` in Java
- operations performed
  - add/push O(1) - adds to the top
  - remove/pop O(1) - removes from the top
  - top O(1) - access the element on the top (optional)
  - empty O(1) - determine if the stack is empty (optional)
- used in many algorithms for solving problems
  - postfix, prefix calculations and conversions
  - graph algorithms

Challenge: Function Call Stack

What is the output of `fun(4)` ?

```
        (   x ) {
    (x == 0)       ;
  printf ("%d\n", x);
  fun(x-1);
}
```

Challenge: Function Call Stack

What is the output of `fun(4)` ?

```
          (   x ) {
     (x == 0)       ;
   printf ("%d\n", x);
   fun(x-1);
}
```

Output

```
4
3
2
1
```

Challenge: Function Call Stack

What is the output of `fun(4)` ?

```
        (   x ) {
    (x == 0)       ;
  fun(x-1);
  printf ("%d\n", x);
}
```

Challenge: Function Call Stack

What is the output of `fun(4)` ?

```
        (   x ) {
    (x == 0)       ;
  fun(x-1);
  printf ("%d\n", x);
}
```

Output

```
1
2
3
4
```

Challenge: Function Call Stack

What is the output of `fun(4)` ?

```
        (   x ) {
    (x == 0)       ;
  fun(x-1);
  fun(x-1);
  printf ("%d ", x);
}
```

Challenge: Function Call Stack

What is the output of `fun(4)` ?

```
        (   x ) {
    (x == 0)       ;
  fun(x-1);
  fun(x-1);
  printf ("%d ", x);
}
```

Output `1 1 2 1 1 2 3 1 1 2 1 1 2 3 4`

Challenge: Function Call Stack

What is the output of `fun(4)` ?

```
        (   x ) {
    (x == 0)       ;
  fun(x-1);
  fun(x-1);
  printf ("%d ", x);
}
```

Output `1 1 2 1 1 2 3 1 1 2 1 1 2 3 4`

How about these functions? (Try it on your own after the class.)

```
        (   x ) {
    (x == 0)       ;
  printf ("%d ", x);
  fun(x-1);
  fun(x-1);
  printf ("%d ", x);
}
```

```
        (   x,    y ) {
    ( abs(x) >= 3 || abs(y) >= 2)
          ;
  printf ("%d %d\n", x, y);
  fun(x-1, y+1);
  fun(x  , y-1);
  fun(x+1, y-1);
  printf ("%d %d\n", x, y);
}
```

Challenge: Function Call Stack

What is the output of `fun(4)` ?

```
        (   x ) {
    (x == 0)       ;
  fun(x-1);
  fun(x-1);
  printf ("%d ", x);
}
```

Output `1 1 2 1 1 2 3 1 1 2 1 1 2 3 4`

How about these functions? (Try it on your own after the class.)

```
        (   x ) {
    (x == 0)       ;
  printf ("%d ", x);
  fun(x-1);
  fun(x-1);
  printf ("%d ", x);
}
```

```
        (   x,    y ) {
    ( abs(x) >= 3 || abs(y) >= 2)
          ;
  printf ("%d %d\n", x, y);
  fun(x-1, y+1);
  fun(x  , y-1);
  fun(x+1, y-1);
  printf ("%d %d\n", x, y);
}
```

For more complicated recursive functions, draw a call stack tree.

Challenge: Brackets Matching

Given a mathematical expression containing parentheses, i.e., `(` and `)`, determine if the expression is valid.

Example:

`()(())` => valid

`)()(())(` => invalid

`()()(` => invalid

Challenge: Brackets Matching

Given a mathematical expression containing parentheses, i.e., `(` and `)`, determine if the expression is valid.

Example:

`()(())` => valid

`)()(())(` => invalid

`()()(` => invalid

- Can you solve it without using any data structures (i.e., no stack)?

Challenge: Brackets Matching

Given a mathematical expression containing parentheses, i.e., `(` and `)`, determine if the expression is valid.

Example:

`()(())` => valid

`)()(())(` => invalid

`()()(` => invalid

- Can you solve it without using any data structures (i.e., no stack)?
- Can you solve it for different types of brackets in a single expression?

  `{()()}[(())]` => valid

  `{)()(]` => invalid

  `(){(})` => invalid

Challenge: Brackets Matching

Solution with one kind of brackets:

- keep an integer that starts at zero
- for each opening bracket increment it
- for closed bracket decrement it (if less than zero, INVALID)

if the value at the end is zero, then VALID, otherwise INVALID

Challenge: Brackets Matching

Solution with one kind of brackets:

- keep an integer that starts at zero
- for each opening bracket increment it
- for closed bracket decrement it (if less than zero, INVALID)

if the value at the end is zero, then VALID, otherwise INVALID

Solution for mixed brackets:

- keep a stack of characters
- for each opening brackets, push it on the stack
- for each closing bracket,
  - if matches top of the stack, then pop the stack
  - otherwise, INVALID (this covers empty stack as well)

if the stack is empty, then VALID, otherwise INVALID

Challenge: Brackets Matching

Solution with one kind of brackets:

- keep an integer that starts at zero
- for each opening bracket increment it
- for closed bracket decrement it (if less than zero, INVALID)

if the value at the end is zero, then VALID, otherwise INVALID

Solution for mixed brackets:

- keep a stack of characters
- for each opening brackets, push it on the stack
- for each closing bracket,
  - if matches top of the stack, then pop the stack
  - otherwise, INVALID (this covers empty stack as well)

if the stack is empty, then VALID, otherwise INVALID

What is the time complexity of these algorithms?

Challenge: Brackets Matching

Solution with one kind of brackets:

- keep an integer that starts at zero
- for each opening bracket increment it
- for closed bracket decrement it (if less than zero, INVALID)

if the value at the end is zero, then VALID, otherwise INVALID

Solution for mixed brackets:

- keep a stack of characters
- for each opening brackets, push it on the stack
- for each closing bracket,
  - if matches top of the stack, then pop the stack
  - otherwise, INVALID (this covers empty stack as well)

if the stack is empty, then VALID, otherwise INVALID

What is the time complexity of these algorithms?

- Linear in the length of the input string expression.

Exercise

Can we use dynamic/resizable array (`vector` in C++ or `Vector` in Java) to provide efficient implementation of a stack?

- If so, figure out how to do it (i.e., determine which functions in those classes give provide the functionality that is required by the stack).

Evaluating Mathematical Expressions

Evaluate an arithmetic expression with only operators and numbers:

- `1 + 2 + 3 => 6`
- `1 + 2 * 3 => 7`
- `1 * 2 * 3 => 6`
- `1 * 2 + 3 => 5`

Evaluating Mathematical Expressions

Evaluate an arithmetic expression with only operators and numbers:

- `1 + 2 + 3 => 6`
- `1 + 2 * 3 => 7`
- `1 * 2 * 3 => 6`
- `1 * 2 + 3 => 5`

Do the same with added parenthesis:

- `1 + 2 * 3 => 7`
- `(1 + 2) * 3 => 9`
- `1 * 2 + 3 => 5`
- `1 * (2 + 3) => 6`

Evaluating Mathematical Expressions

Evaluate an arithmetic expression with only operators and numbers:

- `1 + 2 + 3 => 6`
- `1 + 2 * 3 => 7`
- `1 * 2 * 3 => 6`
- `1 * 2 + 3 => 5`

Do the same with added parenthesis:

- `1 + 2 * 3 => 7`
- `(1 + 2) * 3 => 9`
- `1 * 2 + 3 => 5`
- `1 * (2 + 3) => 6`

The code that can evaluate such expressions has to:

- find and evaluate all subexpressions
- for each operator figure out what its operands are

It is easier if we can ignore the parenthesis and not have to worry about the operator precedence.

Prefix and Postfix Notations
(a.k.a. Polish and Reverse Polish Notation)

Infix notation is a notation for writing human readable arithmetic expressions in which the operator appears *in-between* its operands.

Prefix notation is a notation for writing arithmetic expressions in which the operator comes *before* its operands.

Postfix notation is a notation for writing arithmetic expressions in which the operator comes *after* its operands.

infix	prefix	postfix
`2 + 5`	`+ 2 5`	`2 5 +`
`(2 + 4) * 5`	`* + 2 4 5`	`2 4 + 5 *`
`2 + 4 * 5`	`+ 2 * 4 5`	`2 4 5 * +`

Evaluate Prefix Expressions

- scan the given prefix expression from right to left
- for each token in the input prefix expression
  - if the token is an operand then
    - push onto a stack
  - else if the token is an operator then
    - operand1 = pop stack
    - operand2 = pop stack
    - compute operand1 operator operand2
    - push result onto stack
- return top of stack as result

Evaluate Postfix Expressions

- scan the given postfix expression from left to right
- for each token in the input postfix expression
  - if the token is an operand
    - push it (its value) onto a stack
  - else if the token is an operator
    - operand2 = pop stack (!!!)
    - operand1 = pop stack
    - compute operand1 operator operand2
    - push result onto stack
- return top of the stack as result

Convert Infix to Postfix

- for each tokn in the input infix string expression
  - if the token is an operand
    - append to postfix string expression
  - else if the token is a left brace
    - push it onto the operator stack
  - else if the token is an operator
    - if the stack is not empty
      - while top element on the stack is not a left brace AND has higher or equal precedence
        - pop the stack and append to postfix string expression
    - push it (the current operator) onto the operator stack
  - else if the token is a right brace
    - while the operator stack is not empty
      - if the top of the operator stack is not a matching left brace
        - pop the operator stack and append to postfix string expression
      - else
        - pop the left brace and discard
        - break
- while the operator stack is not empty
  - pop the operator stack and append to postfix string expression

Exercise

Apply the infix to postfix conversion to the following expression:

Show the content of the operator stack and the postfix expression after each iteration of the outermost for loop.

Exercise

Apply the infix to postfix conversion to the following expression:

Show the content of the operator stack and the postfix expression after each iteration of the outermost for loop.

You should end up with `4 5 6 * 1 2 + 3 + / +`.

(full solution on the next slide)

Infix	Stack	Postfix	Comments
4 + (5 * 6) / (1 + 2 + 3)		4	
4 **+** (5 * 6) / (1 + 2 + 3)	+	4	
4 + **(**5 * 6) / (1 + 2 + 3)	+(	4	
4 + (**5** * 6) / (1 + 2 + 3)	+(	4 5	
4 + (5 ***** 6) / (1 + 2 + 3)	+(*	4 5	
4 + (5 * **6**) / (1 + 2 + 3)	+(*	4 5 6	
4 + (5 * 6**)** / (1 + 2 + 3)	+	4 5 6 *	A
4 + (5 * 6) **/** (1 + 2 + 3)	/	4 5 6 * +	B
4 + (5 * 6) / **(**1 + 2 + 3)	/(	4 5 6 * +	
4 + (5 * 6) / (**1** + 2 + 3)	/(	4 5 6 * + 1	
4 + (5 * 6) / (1 **+**$_A$ 2 + 3)	/(+$_A$	4 5 6 * + 1	C
4 + (5 * 6) / (1 + **2** + 3)	/(+$_A$	4 5 6 * + 1 2	
4 + (5 * 6) / (1 + 2 **+**$_B$ 3)	/(+$_B$	4 5 6 * + 1 2 +$_A$	
4 + (5 * 6) / (1 + 2 + **3**)	/(+$_B$	4 5 6 * + 1 2 +$_A$ 3	
4 + (5 * 6) / (1 + 2 + 3**)**	/	4 5 6 * + 1 2 +$_A$ 3 +$_B$	D
4 + (5 * 6) / (1 + 2 + 3)		4 5 6 * + 1 2 +$_A$ 3 +$_B$ /	E

A: pop operators down to (and then remove the bracket

B: / has higher precedence than +

C: + operator marked with A to distinguish from the next one

D: pop everything up to and including the left bracket

E: end of the expression, pop operator and append them to the postfix