CSC 225 - Summer 2019

Fundamental Data Structures

Bill Bird

Department of Computer Science
University of Victoria

May 21, 2019

Data Structures

A **Data Structure** is an organized collection of data with a collection of associated operations.

An **Abstract Data Type** (ADT) is a well-defined collection of operations for a particular set of data.

Abstract Data Types are used to describe the externally visible parts of data structures (similar to the role of interfaces in Java).

Stacks and Queues

A **stack** is a data structure with three operations:

PUSH(x)	Add the element x to the top of the stack.
POP()	Remove the top element from the stack (Last in, first out).
PEEK()	Return the top element without removing it.

A **queue** is a data structure with two operations:

ENQUEUE(x)	Add the element x to the end of the queue.
DEQUEUE()	Remove the element from the beginning of the queue (First in, first out).
FRONT()	Return the first element without removing it.

Lists

A **list** is a data structure with the following operations:

AddFirst(x)	Add the element x at the beginning of the list.
AddLast(x)	Add the element x at the end of the list.
AddAfter(y, x)	Insert the element x after element y.
RemoveFirst()	Remove and return the first element of the list.
RemoveLast()	Remove and return the last element of the list.
Remove(x)	Remove element x from the list.
Swap(x, y)	Swap x and y.
Before(x)	Return the element before element x.
After(x)	Return the element after element x.
Length(x)	Return the number of elements in the list.

List Implementation

Function	**Linked List**	**Array**	
		Bounded	Resizeable
AddFirst(x)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
AddLast(x)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
AddAfter(y, x)	$\Theta(1)$[1]	$\Theta(n)$	$\Theta(n)$
RemoveFirst()	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
RemoveLast()	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Remove(x)	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Swap(x, y)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Before(x)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
After(x)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Length(x)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

[1] Given a list node y, we can insert an element after y in constant time.

Stack and Queue Implementation

	Function	**Linked List**	**Array**	
			Bounded	Resizeable
Stack	PUSH(x)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
	POP()	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
	PEEK()	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Queue	ENQUEUE(x)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
	DEQUEUE()	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
	FRONT()	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Note

The remaining slides in this sequence contain some examples of algorithm design problems that benefit from stacks and queues (and some examples of how to indicate data structure use in pseudocode). These might be helpful as study material for later.

Most of these applications are review from CSC 115 (so we won't cover them directly in the lectures).

Balanced Parentheses (1)

Easy Example: Design an algorithm to check whether parentheses in an arithmetic expression are balanced (with every '(' having a matching ')' and vice versa). Assume that the expression is provided as a string of characters (and that strings can be indexed like arrays).

Balanced	**Unbalanced**
`6`	`(6`
`(6)`	`6)`
`(x + y)`	`)x + y(`
`5 + (6*10)`	`5 + 6*10)`
`(5 + (6*10))`	`(5 + (6*10)`
`(6*(5+10)*(x*y))`	`(6*5+10)*(x*y))`

Balanced Parentheses (2)

```
procedure BalancedParentheses1(str)
    n ← Length of str
    depth ← 0
    for i = 0, 1, ..., n − 1 do
        if str[i] = '(' then
            depth ← depth + 1
        else if str[i] = ')' then
            depth ← depth − 1
            if depth < 0 then
                return False
            end if
        end if
    end for
    if depth = 0 then
        return True
    else
        return False
    end if
end procedure
```

Balanced Parentheses (3)

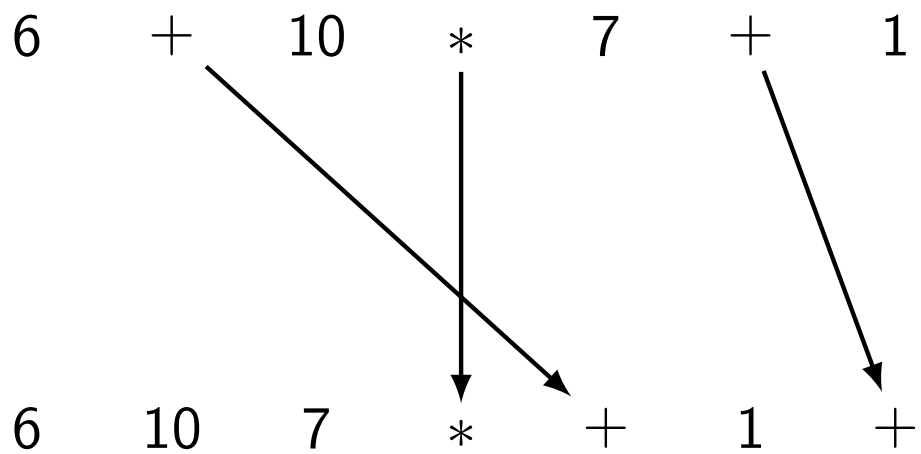
Harder Example: Modify the previous algorithm to accommodate round and square brackets such that brackets are correctly nested.

Balanced	**Unbalanced**
`[6]`	`[6`
`(6)`	`[6)`
`[(x) + y]`	`[(x] + y)`
`[[x+[y]] + (5*6)]`	`[x+(y] + [5*6)]`
`(5 + [6*10])`	`[5 + (6*10])`
`[6*(5+10(x*y))]`	`[6*(5+10]*[(x)*y)]`

Balanced Parentheses (4)

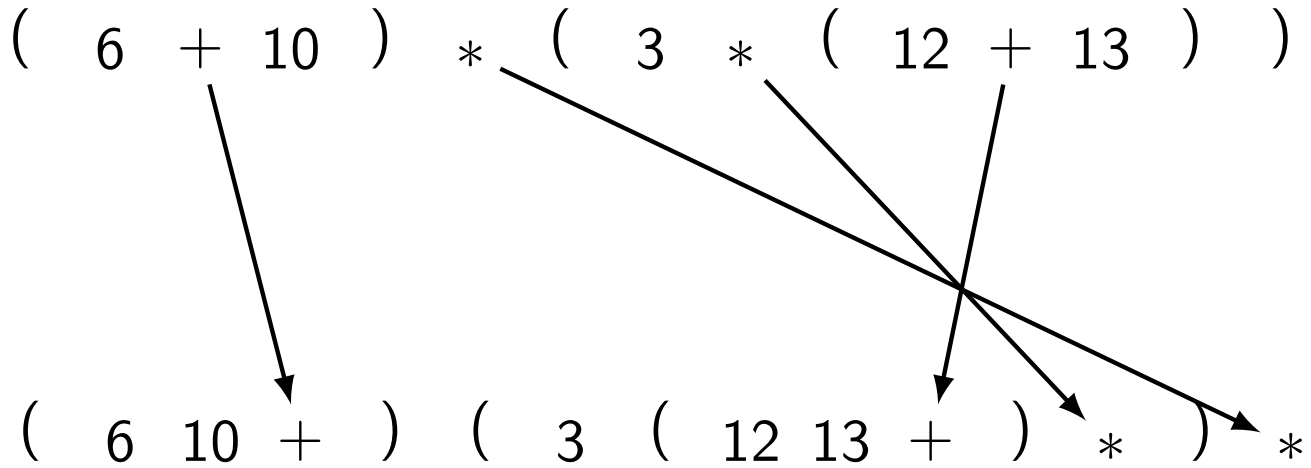
```
procedure BalancedParentheses2(str)
    n ← Length of str
    S ← Empty stack
    for i = 0, 1, ..., n − 1 do
        if str[i] is one of '(' or '[' then
            Push str[i] onto S
        else if str[i] = ')' then
            if S is empty or Pop(S) ≠ ')' then
                return False
            end if
        else if str[i] = ']' then
            if S is empty or Pop(S) ≠ '[' then
                return False
            end if
        end if
    end for
    if S is empty then
        return True
    else
        return False
    end if
end procedure
```

Postfix Arithmetic (1)



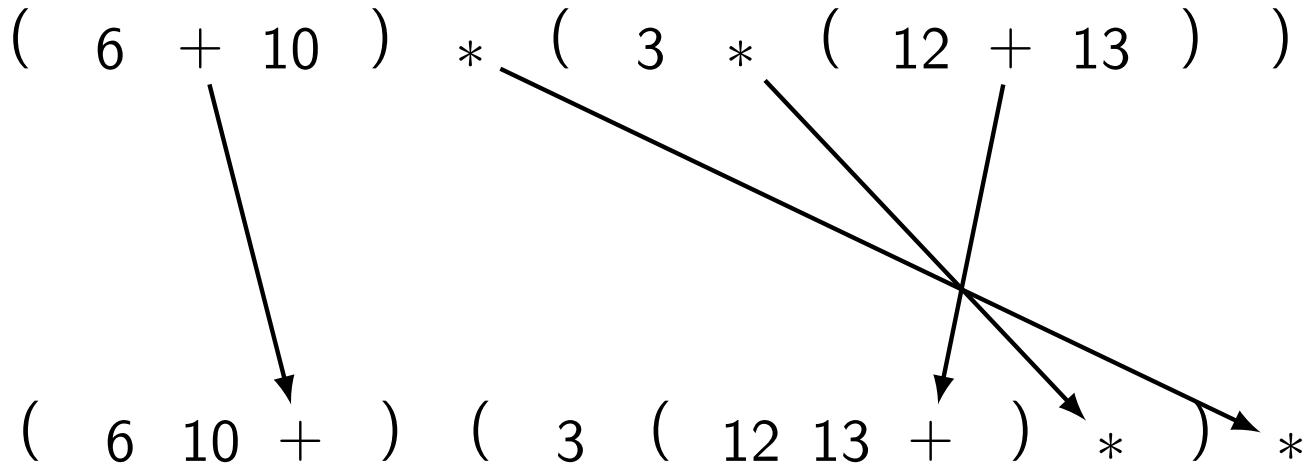
- Normally, equations are written in *infix* notation, where operators appear between operands (for example, '$5 + 6$').
- In *postfix* notation, the operator is written after all operands (for example, '56+')

Postfix Arithmetic (2)



- If the expression is parenthesized, it is easy to reorder operands into postfix (just move each operator to the end of its enclosing parentheses).
- In postfix, no parentheses are ever necessary (and order of operations is extremely simple compared to infix).

Postfix Arithmetic (3)



- The final expression above can be written

$$6 \quad 10 \quad + \quad 3 \quad 12 \quad 13 \quad + \quad * \quad *$$

Postfix Evaluation (1)

EvaluatePostfix	
Input:	A list L containing a postfix expression, using only the $+$ and $*$ operators. For example, $L = [6, 10, *, 7, +]$.
Output:	The numerical result of evaluating the expression in L.

Postfix expressions can be evaluated with a simple iterative algorithm. Infix expressions usually require a tree based structure.

Postfix Evaluation (2)

```
procedure EvaluatePostfix(L)
    S ← Empty stack
    for each token t in L do
        if t is a number then
            Push t onto S
        else if t = + then
            x ← Pop(S)
            y ← Pop(S)
            Push x + y onto S
        else if t = * then
            x ← Pop(S)
            y ← Pop(S)
            Push x · y onto S
        end if
    end for
    return Pop(S)
end procedure
```