

***Welcome To ...***

# Advanced Data Structures

吕建华

[lujianhua@seu.edu.cn](mailto:lujianhua@seu.edu.cn)

Office: 计算机楼230

课程QQ群: 881909438



Many Thanks to Sartaj Sahni

# What The Course Is About



- Study data structures for:
  - Space efficient applications
  - External sorting
  - Dictionaries
  - Pattern Matching
  - Single and double ended priority queues
  - Multidimensional search
  - Computational geometry
  - Image processing
  - Packet routing and classification
  - ...

# What The Course Is About



- Concerned with:
  - Worst-case complexity
  - Average complexity
  - Amortized complexity

# Prerequisites

- ✓ C++
- ✓ Asymptotic Complexity
  - Big Oh, Theta, and Omega notations
- ✓ Undergraduate data structures
  - Stacks and Queues
  - Linked lists
  - Trees
  - Graphs

# Kinds Of Complexity

- ✓ Worst-case complexity.
- ✓ Average complexity.
- Amortized complexity.

# Quick Sort

- Sort  $n$  distinct numbers.
- Worst-case time is (say)  $10n^2$  microseconds on some computer.
- This means that for every  $n$ , there is a sequence of  $n$  numbers for which quick sort will take  $10n^2$  microseconds to complete.
- Also, there is no sequence of  $n$  numbers for which quick sort will take more than  $10n^2$  microseconds to complete.

# Quick Sort

- Average time is (say)  $5n \log_2 n$  microseconds on some computer.
- Consider any  $n$ , say  $n = 1000$ .
- Add up the time taken to sort each of the  $1000!$  possible  $1000$  element sequences.
- Divide by  $1000!$ .
- The result is  $5000 \log_2 1000$ .

# Quick Sort

- What if we sort only 500 of these 1000! sequences?
- We can only conclude that the total time for these 500 sequences will be
$$\leq 500 * (\text{worst-case time})$$
$$= 500 * (10n^2)$$
- We cannot conclude that the time will be  $500 * (\text{average time})$ .



# Task Sequence

- Suppose that a sequence of  $n$  tasks is performed.
- The worst-case cost of a task is  $c_{wc}$ .
- Let  $c_i$  be the (actual) cost of the  $i^{\text{th}}$  task in this sequence.
- So,  $c_i \leq c_{wc}$ ,  $1 \leq i \leq n$ .
- $n * c_{wc}$  is an upper bound on the cost of the sequence.
- $j * c_{wc}$  is an upper bound on the cost of the first  $j$  tasks.

# Task Sequence

- Let  $c_{avg}$  be the average cost of a task in this sequence.
- So,  $c_{avg} = \sum c_i / n$ .
- $n * c_{avg}$  is the cost of the sequence.
- $j * c_{avg}$  is not an upper bound on the cost of the first  $j$  tasks.
- Usually, determining  $c_{avg}$  is quite hard.

# Task Sequence

- At times, a better upper bound than  $j * c_{wc}$  or  $n * c_{wc}$  on sequence cost is obtained using amortized complexity.

# Amortized Complexity

- The **amortized complexity** of a task is the amount you charge the task.
- The conventional way to bound the cost of doing a task  $n$  times is to use one of the expressions
  - $n * (\text{worst-case cost of task})$
  - $\Sigma(\text{worst-case cost of task } i)$
- The **amortized complexity** way to bound the cost of doing a task  $n$  times is to use one of the expressions
  - $n * (\text{amortized cost of task})$
  - $\Sigma(\text{amortized cost of task } i)$

# Amortized Complexity

- The amortized complexity/cost of individual tasks in any task sequence must satisfy:  
 $\Sigma(\text{actual cost of task } i)$   
 $\leq \Sigma(\text{amortized cost of task } i)$
- So, we can use  
 $\Sigma(\text{amortized cost of task } i)$   
as a bound on the actual complexity of the task sequence.

# Amortized Complexity

- The amortized complexity of a task may bear no direct relationship to the actual complexity of the task.

# Amortized Complexity

- In worst-case complexity analysis, each task is charged an amount that is  $\geq$  its cost.

$$\Sigma(\text{actual cost of task } i)$$

$$\leq \Sigma(\text{worst-case cost of task } i)$$

- In amortized analysis, some tasks may be charged an amount that is  $<$  their cost.

$$\Sigma(\text{actual cost of task } i)$$

$$\leq \Sigma(\text{amortized cost of task } i)$$

# Arithmetic Statements

- Rewrite an arithmetic statement as a sequence of statements that do not use parentheses.
- $a = x + ((a + b) * c + d) + y;$   
is equivalent to the sequence:  
 $z1 = a + b;$   
 $z2 = z1 * c + d;$   
 $a = x + z2 + y;$



# Arithmetic Statements

$a = x + ((a + b) * c + d) + y;$

- The rewriting is done using a stack and a method `processNextSymbol`.

- create an empty stack;

for (int i = 1; i <= n; i++)

// n is number of symbols in statement

`processNextSymbol();`

# Arithmetic Statements

$$a = x + ((a + b) * c + d) + y;$$

- `processNextSymbol` extracts the next symbol from the input statement.
- Symbols other than `)` and `;` are simply pushed on to the stack.

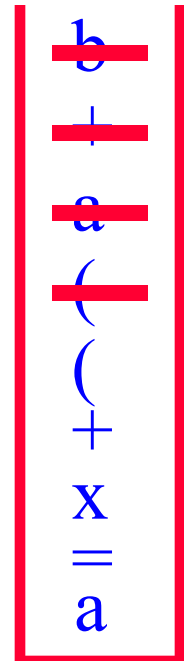
b  
+  
a  
(  
(  
+  
x  
=  
a

# Arithmetic Statements

$$a = x + ((a + b) * c + d) + y;$$

- If the next symbol is `)`, symbols are popped from the stack up to and including the first `(`, an assignment statement is generated, and the left hand symbol is added to the stack.

$$z1 = a + b;$$

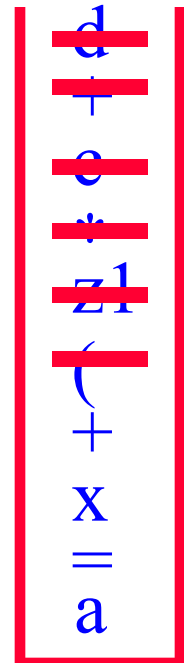


# Arithmetic Statements

$$a = x + ((a + b) * c + d) + y;$$

- If the next symbol is `)`, symbols are popped from the stack up to and including the first `(`, an assignment statement is generated, and the left hand symbol is added to the stack.

$$\begin{aligned} z1 &= a + b; \\ z2 &= z1 * c + d; \end{aligned}$$



# Arithmetic Statements

$$a = x + ((a + b) * c + d) + y;$$

- If the next symbol is  $)$ , symbols are popped from the stack up to and including the first  $($ , an assignment statement is generated, and the left hand symbol is added to the stack.

$$\begin{aligned} z1 &= a + b; \\ z2 &= z1 * c + d; \end{aligned}$$

y  
+  
z2  
+  
x  
=  
a

# Arithmetic Statements

$$a = x + ((a + b) * c + d) + y;$$

- If the next symbol is `;`, symbols are popped from the stack until the stack becomes empty. The final assignment statement

$$a = x + z2 + y;$$

is generated.

$$z1 = a + b;$$
$$z2 = z1 * c + d;$$

y  
+  
z2  
+  
x  
=  
a

# Complexity Of processNextSymbol

$a = x + ((a + b) * c + d) + y;$

- $O(\text{number of symbols that get popped from stack})$
- $O(i)$ , where  $i$  is for loop index.

# Overall Complexity (Conventional Analysis)

```
create an empty stack;
```

```
for (int i = 1; i <= n; i++)
```

```
// n is number of symbols in statement
```

```
processNextSymbol();
```

- So, overall complexity is  $O(\sum i) = O(n^2)$ .
- Alternatively,  $O(n * n) = O(n^2)$ .
- Although correct, a more careful analysis permits us to conclude that the complexity is  $O(n)$ .



# Ways To Determine Amortized Complexity

- Aggregate method.
- Accounting method.
- Potential function method.

# Aggregate Method

- Somehow obtain a good upper bound on the actual cost of the **n** invocations of `processNextSymbol()`
- Divide this bound by **n** to get the amortized cost of one invocation of `processNextSymbol()`
- Easy to see that
$$\Sigma(\text{actual cost}) \leq \Sigma(\text{amortized cost})$$

# Aggregate Method

- The actual cost of the **n** invocations of `processNextSymbol()` equals number of stack pop and push operations.
- The **n** invocations cause at most **n** symbols to be pushed on to the stack.
- This count includes the symbols for new variables, because each new variable is the result of a `)` being processed. Note that no `)`s get pushed on to the stack.

# Aggregate Method

- The actual cost of the  $n$  invocations of `processNextSymbol()` is at most  $2n$ .
- So, using  $2n/n = 2$  as the amortized cost of `processNextSymbol()` is OK, because this cost results in  $\Sigma(\text{actual cost}) \leq \Sigma(\text{amortized cost})$
- Since the amortized cost of `processNextSymbol()` is  $2$ , the actual cost of all  $n$  invocations is at most  $2n$ .

# Aggregate Method

- The aggregate method isn't very useful, because to figure out the amortized cost we must first obtain a good bound on the aggregate cost of a sequence of invocations.
- Since our objective was to use amortized complexity to get a better bound on the cost of a sequence of invocations, if we can obtain this better bound through other techniques, we can omit dividing the bound by  $n$  to obtain the amortized cost.