Lecture 10/07/16

Lecturer: Xiaodi Wu

Reading: Chapter 1.5, [CLRS] Chap 17, Note on Amortized Analysis
FIFO vs LIFO

FIFO implemented by 2 LIFOs
FIFO vs LIFO

FIFO implemented by 2 LIFOs

- enqueue(o): stack2.push(o).

- dequeue():
  - if (!stack1.isEmpty()) then return stack1.pop();
  - else while (!stack2.isEmpty()) do {
    - o = stack2.pop(); stack1.push(o);
  }
  - return stack1.pop();
FIFO vs LIFO

FIFO implemented by 2 LIFOs

- **enqueue(o):** stack2.push(o).
- **dequeue():**
  ```
  if (! stack1.isEmpty()) then return stack1.pop();
  else while (! stack2.isEmpty()) do
    { o=stack2.pop(); stack1.push(o); }
  return stack1.pop();
  ```
FIFO vs LIFO

FIFO implemented by 2 LIFOs

- **enqueue(o)**: stack2.push(o).
- **dequeue()**: if (! stack1.isEmpty()) then return stack1.pop(); else while (! stack2.isEmpty()) do
  \{ o=stack2.pop(); stack1.push(o); \}

**return** stack1.pop();

**Question**: LIFO implemented by 2 FIFOs?
Amortized Analysis

Stack: multi-pop()

- multi-pop(): pop out all objects in the stack by LIFO principle.
Amortized Analysis

Stack: multi-pop()

- multi-pop(): pop out all objects in the stack by LIFO principle.
- multi-pop(): while (!is_empty()) do pop();
Amortized Analysis

Stack: multi-pop()

- multi-pop(): pop out all objects in the stack by LIFO principle.
- multi-pop(): while (!isEmpty()) do pop();
- For stack with \( n \) elements, what is the time complexity of multi-pop()?
Amortized Analysis

Stack: multi-pop()

- multi-pop(): pop out all objects in the stack by LIFO principle.
- multi-pop(): while (!isEmpty()) do pop();
- For stack with $n$ elements, what is the time complexity of multi-pop()? $O(n)$. 

Time of $m$ push() and/or multi-pop() operations from an empty stack

- push() takes $O(1)$, multi-pop() takes $O(m)$, worst case $m \times O(m) = O(m^2)$.
- It is a correct $O(\cdot)$ statement, but a huge over-estimate.
Amortized Analysis

Stack: multi-pop()

- multi-pop(): pop out all objects in the stack by LIFO principle.
- multi-pop(): while (!isEmpty()) do pop();
- For stack with $n$ elements, what is the time complexity of multi-pop()? $O(n)$.

Time of $m$ push() and/or multi-pop() operations from an empty stack

- push() takes $O(1)$, multi-pop() takes $O(m)$, worst case $m \times O(m) = O(m^2)$.
Amortized Analysis

Stack: multi-pop()

- multi-pop(): pop out all objects in the stack by LIFO principle.
- multi-pop(): while (!isEmpty()) do pop();
- For stack with $n$ elements, what is the time complexity of multi-pop()? $O(n)$.

Time of $m$ push() and/or multi-pop() operations from an empty stack

- push() takes $O(1)$, multi-pop() takes $O(m)$, worst case $m \times O(m) = O(m^2)$.
- It is a correct $O(\cdot)$ statement, but a huge over-estimate.
Amortized Analysis: cont’d

Theorem (1.30 on page 34)

A series of $m$ operations on an initially empty stack takes $O(m)$ time.
Theorem (1.30 on page 34)
A series of $m$ operations on an initially empty stack takes $O(m)$ time.

Proof.
Let $M_0, \cdots, M_{m-1}$ be the series of operations, and let $M_{i_0}, \cdots, M_{i_{k-1}}$ be the $k$ multi-pop() operations. We have

$$0 \leq i_0 \leq \cdots \leq i_{k-1} \leq n-1, i_{-1} = -1.$$
Amortized Analysis: cont’d

Theorem (1.30 on page 34)

A series of \( m \) operations on an initially empty stack takes \( O(m) \) time.

Proof.

Let \( M_0, \ldots, M_{m-1} \) be the series of operations, and let \( M_{i_0}, \ldots, M_{i_k-1} \) be the \( k \) multi-pop() operations. We have

\[
0 \leq i_0 \leq \cdots \leq i_{k-1} \leq n - 1, \quad i_{-1} = -1.
\]

Time cost of \( M_{i_{j+1}} \) to \( M_{i_j} \) for each \( j = 0, \ldots, k - 1 \):

- \( i_j - i_{j-1} - 1 \) operations of push(). cost \( O(i_j - i_{j-1}) \).
Amortized Analysis: cont’d

**Theorem (1.30 on page 34)**

*A series of m operations on an initially empty stack takes* \(O(m)\) *time.*

**Proof.**

Let \(M_0, \ldots, M_{m-1}\) be the series of operations, and let \(M_{i_0}, \ldots, M_{i_{k-1}}\) be the \(k\) multi-pop() operations. We have

\[
0 \leq i_0 \leq \cdots \leq i_{k-1} \leq n - 1, \ i_{-1} = -1.
\]

**Time cost of** \(M_{i_{j+1}}\) **to** \(M_{i_j}\) **for each** \(j = 0, \ldots, k - 1:\**

- \(i_j - i_{j-1} - 1\) operations of push(). cost \(O(i_j - i_{j-1}).\)
- 1 multi-pop(): only \(i_j - i_{j-1} - 1\) elements in the stack. cost: \(O(i_j - i_{j-1}).\)

\(\square\)
Theorem (1.30 on page 34)

A series of $m$ operations on an initially empty stack takes $O(m)$ time.

Proof.
Sum up, we have the total time is (telescoping sum)

$$O \left( \sum_{j=0}^{k-1} (i_j - i_{j-1}) \right) = O(m).$$
Amortized Analysis: cont’d

Theorem (1.30 on page 34)
A series of \( m \) operations on an initially empty stack takes \( O(m) \) time.

Proof.
Sum up, we have the total time is (telescoping sum)

\[
O \left( \sum_{j=0}^{k-1} (i_j - i_{j-1}) \right) = O(m).
\]

Remark: Worst case analysis of a single operation leads to loose bounds for a series of operations!
Amortized Analysis: cont’d

For a single operation,

\[
\text{amortized running time} = \frac{\text{worst case complexity of } m \text{ operations}}{m}.
\]
Amortized Analysis: cont’d

For a single operation,

\[ \text{amortized running time} = \frac{\text{worst case complexity of } m \text{ operations}}{m}. \]

For multi-type operations, e.g., 2 types

\[ \text{worst case complexity of } m_1 \text{ op1 and } m_2 \text{ op2} \leq \text{amortized complexity op1 } \times m_1 + \text{amortized complexity op2 } \times m_2. \]
For a single operation,

$$\text{amortized running time} = \frac{\text{worst case complexity of } m \text{ operations}}{m}.$$ 

For multi-type operations, e.g., 2 types

$$\text{worst case complexity of } m_1 \text{ op1 and } m_2 \text{ op2} \leq \text{amortized complexity } \text{op1} \times m_1 + \text{amortized complexity } \text{op2} \times m_2.$$ 

Thus, push() and multi-pop() have amortized complexity $O(1)$. 
Amortized Analysis: more intuitive derivation

Question: perform amortized analysis besides by definition?
Amortized Analysis: more intuitive derivation

- **Question:** perform amortized analysis besides by definition?
- **Key:** analyze and upper bound the complexity of a series of operations!
Amortized Analysis: more intuitive derivation

- **Question:** perform amortized analysis besides by definition?
- **Key:** analyze and upper bound the complexity of a series of operations!

#primitive operations in m operations ≤ resources spent
Amortized Analysis: more intuitive derivation

- **Question**: perform amortized analysis besides by definition?
- **Key**: analyze and upper bound the complexity of a series of operations!

\[ \text{#primitive operations in m operations} \leq \text{resources spent} \]

When the resource is
- Money ⇒ **The Accounting Method**.
Amortized Analysis: more intuitive derivation

- **Question:** perform amortized analysis besides by definition?
- **Key:** analyze and upper bound the complexity of a series of operations!

\[ \#\text{primitive operations in } m \text{ operations} \leq \text{resources spent} \]

When the resource is
- Money ⇒ **The Accounting Method.**
- Energy ⇒ **The Potential Function Method**
The Accounting Method

Principle

- Every primitive operation costs 1-unit money.
The Accounting Method

Principle

- Every primitive operation costs 1-unit money.
- Deposit money whenever performing an operation (amortized complexity). Money spent after every primitive operation.

Correctness

\[
\text{all primitive ops} \leq \text{all money deposited} = \text{amortized complexity} \times \text{# ops} \leq \text{due to your balance being non-negative all the time!}
\]
The Accounting Method

Principle

- Every primitive operation costs 1-unit money.
- Deposit money whenever performing an operation (amortized complexity). Money spent after every primitive operation.
- Your bank starts with zero-balance and remains non-negative during the whole procedure. No loan!
The Accounting Method

Principle

- Every primitive operation costs 1-unit money.
- Deposit money whenever performing an operation (amortized complexity). Money spent after every primitive operation.
- Your bank starts with zero-balance and remains non-negative during the whole procedure. No loan!

Correctness

\[
\#\text{all primitive ops} \leq \#\text{all money deposited} = \text{amortized complexity} \times \#\text{ops}
\]
The Accounting Method

Principle

- Every primitive operation costs 1-unit money.
- Deposit money whenever performing an operation (amortized complexity). Money spent after every primitive operation.
- Your bank starts with zero-balance and remains non-negative during the whole procedure. No loan!

Correctness

\[
\text{all primitive ops} \leq \text{all money deposited} = \text{amortized complexity} \times \text{# ops}
\]

\[\leq \text{due to your balance being non-negative all the time!}\]
The Accounting Method: Example

Push() & Multi-pop()

- deposit 2$ for each Push(): 1$ is spent to execute the push operation, 1$ is left in the bank for later.
The Accounting Method: Example

Push() & Multi-pop()

- deposit 2$ for each Push(): 1$ is spent to execute the push operation, 1$ is left in the bank for later.
- deposit 0$ for each Multi-pop(): its cost is paid for by the deposit made at the push operation.
The Accounting Method: Example

Push() & Multi-pop()

- deposit 2$ for each Push(): 1$ is spent to execute the push operation, 1$ is left in the bank for later.
- deposit 0$ for each Multi-pop(): its cost is paid for by the deposit made at the push operation.
- A formal proof requires showing the non-negativity of your balance.
The Accounting Method: Example

Push() & Multi-pop()

- deposit 2$ for each Push(): 1$ is spent to execute the push operation, 1$ is left in the bank for later.
- deposit 0$ for each Multi-pop(): its cost is paid for by the deposit made at the push operation.
- A formal proof requires showing the **non-negativity** of your balance.

Credit Invariant

- Invariant: # of (bank) credits = # of items in the stack.
- Prove the invariant for each operation: push(), multi-pop().