COMP2860 Data structures

Week 4:
- Amortized Analysis

A list problem

• A list with a loop (cycle)
• Singly linked
• Identical nodes
• Unknown number of nodes
• Find the list layout
  – number of nodes in the loop
  – number of nodes before the loop

A list problem

• Use constant amount of extra memory
• You should not add fields on the nodes
• You can use any FIXED number of pointers
• You are given a pointer to the start node

A list problem

• Much easier version of the problem
• Only a loop is given
• Count the number of nodes
• Using only two pointers
A list problem

better names for the pointers:

Run-time complexity

- Running time, scalability, complexity
- Worst case running time
- Worst case number of steps
- Example: bubble sort

Analyzing bubble sort

```pseudocode
array elements[1..N]
for j:= 1 to N do
    for k := j+1 to N do
        if elements[k] > elements[k+1] then
            swap(k,k+1, elements)
        end // if
    end // for k loop
end // for j loop
```

Different bubble-sort

```pseudocode
array elements[1..N]
swapDone = true
while swapDone do
    swapDone = false
    for k := 1 to N-1 do
        if elements[k] > elements[k+1] then
            swap(k,k+1, elements)
            swapDone = true
        end // if
    end // for k loop
end // while loop
```

Bubble-sort

- Need to look at the algorithm
- Find properties for finishing the loop
  - no swaps done
- Why is the worst case complexity still $O(n^2)$?
- Is this a better solution than nested for loops?

Stack operations

- ADT of a Parser Stack
- push() is $O(1)$
- new operation multi-pop(k) is $O(k)$
  - pops k top elements from the stack
- Let $n$ is the size of the stack
- multi-pop(k) is worst case $O(n)$
- What is the total complexity of
  - $m_1$ push operations and
  - $m_2$ multi-pop(k) operations
Stack operations

• Worst case complexity of $m_1 + m_2 k$
• Worst-case upper bound
• Better upper bound?
  – still worst case

Better upper bound

• Not possible to do more pops than the number of push operations before that
• If we’ve only done $m_1$ push operations, we can pop at most $m_1$ elements
• Worst case upper bound $O(m_1)$

Better upper bound

• pop($k$) worst case bound is $O(n)$
• Notice that a sequence of worst-case pop operations are not possible
• Once you pop($n$) there is nothing left to pop
• The total worst case is therefore bounded by the number of push operations
• every element we pop must have been push-ed on the stack first

Analyzing operations

• Expensive operations
  – large worst-case times
• Is it possible to have a series of these expensive operations?
• In order to have enough elements in the data structure to cause expensive operations we need other operations in between

Amortized analysis

• Average performance of each operation in the worst case
• Multi-pop stack example:
  – a single multi-pop($k$) could be $O(n)$
  – on the average, a multi-pop($k$) operation is bounded by the number of previous
    • push operations and
    • pop operations

Amortized analysis

• 3 ways to find the amortized cost
  – Aggregate analysis
  – Accounting method
  – Potential method
Aggregate analysis

• Amortized cost of a single operation
• Consider a sequence of N such operations
• Compute an upper bound \( T(N) \) for the total cost
• Amortized cost for the operation is \( \frac{T(N)}{N} \)
• This is the average cost of the operation
  – worst-case average cost

Aggregate analysis

• For any N, find the worst case time \( T(N) \) for any sequence of N operations
• Amortized cost is \( \frac{T(N)}{N} \)
• All operations are assigned the same amortized cost
• Other methods of amortizations may assign different costs to different operations

Aggregate analysis of a stack

• Multi-pop stack
• push() and pop() operations are O(1)
• multi-pop(k) implementation
  multi-pop(k)
  while stack is not empty
  pop()
  \( k := k - 1 \)
• multi-pop(k) requires O(k) worst case
  – or min{n,k} where n is the size of the stack

Aggregate analysis of a stack

• Initially empty stack
• Consider a sequence of n operations
  – push, pop, multi-pop
  – any mix of these operations
• worst case of a multi-pop is O(n)
• worst case of the whole sequence is O(n^2)
• This bound is not tight

Aggregate analysis of a stack

• multi-pops cannot keep having worst case cost
• Each item can be popped at most once for each time it is pushed
• Number of actual pops is at most equal to the number of push operations
  – including pops in multi-pops
• A sequence of n push, pop and multi-pops has worst case complexity O(n)

Aggregate analysis of a stack

• ANY sequence of n push, pop and multi-pops has worst case complexity O(n)
• The average cost of an operation is \( \frac{O(n)}{n} = O(1) \)
• Average worst case cost
• No probabilistic analysis
• This is worst case analysis
  – and an average upper bound
<table>
<thead>
<tr>
<th>Accounting method</th>
<th>Accounting method</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Charge every operation</td>
<td>• Amount charged versus real cost</td>
</tr>
<tr>
<td>• Different operations get different charge</td>
<td>• An operation may be charged more than its real cost</td>
</tr>
<tr>
<td>• Amount charged is the amortized cost</td>
<td>• This creates a ‘credit’ in the data structure</td>
</tr>
<tr>
<td>• Total amount charged must be enough to</td>
<td>• Credit can be used to pay for subsequent</td>
</tr>
<tr>
<td>pay for the entire sequence of operations</td>
<td>expensive operations</td>
</tr>
<tr>
<td>• Different from Aggregate analysis</td>
<td>– that were charged less than their cost</td>
</tr>
<tr>
<td>– which assigns same costs to all operations</td>
<td>• Distributing the cost over all operations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accounting method</th>
<th>Accounting of a stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Amortized costs (charges) must be chosen carefully</td>
<td>• Real costs of stack operations</td>
</tr>
<tr>
<td>• Sum of amortized costs must cover total</td>
<td>– push 1</td>
</tr>
<tr>
<td>cost of the sequence of operations</td>
<td>– pop 1</td>
</tr>
<tr>
<td>• The bound must hold for all possible sequences</td>
<td>– multi-pop(k) ( \min(k,n) )</td>
</tr>
<tr>
<td>• We must not create a deficit at any time</td>
<td>• Amortized costs</td>
</tr>
<tr>
<td>– total credit must be non-negative at all times</td>
<td>– push 2</td>
</tr>
<tr>
<td></td>
<td>– pop 0</td>
</tr>
<tr>
<td></td>
<td>– multi-pop(k) 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accounting of a stack</th>
<th>Accounting of a stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Amortized costs</td>
<td>• Each operation has amortized cost ( O(1) )</td>
</tr>
<tr>
<td>– push 2 ( O(1) )</td>
<td>worst case</td>
</tr>
<tr>
<td>– pop 0 ( O(1) )</td>
<td>• Total worst case cost of ANY sequence of n stack</td>
</tr>
<tr>
<td>– multi-pop(k) 0 ( O(1) )</td>
<td>operations is ( O(n) )</td>
</tr>
<tr>
<td>• push is over-charged</td>
<td></td>
</tr>
<tr>
<td>– It may cause extra work for pop</td>
<td></td>
</tr>
<tr>
<td>• Enough credit to pay for the possibility that</td>
<td></td>
</tr>
<tr>
<td>the item is later popped</td>
<td></td>
</tr>
</tbody>
</table>


Potential method

- Accounting:
  - Prepaid work is credit in the data structure
- Potential:
  - Prepaid credit is 'potential'
    - that can be released to pay for future operations
  - Potential is associated with the data structure as a whole
    - not with specific operations

Potential method

- Start with initial data structure $D_0$
  - on which $n$ operations will be performed
- For $i=1,2,...,n$ let $c_i$ be the real cost of the $i$-th operation in the sequence
- Potential function $\Phi(D_i)$
- Amortized cost of $i$-th operation is
  - $ac_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - Amortized cost = actual cost + increase in potential due to the operation

Potential method

- Total amortized cost: sum over all $i=1...n$
  - $\sum ac_i = \sum c_i + \Phi(D_n) - \Phi(D_0)$
- Total amortized cost is an upper bound for total cost if $\Phi(D_i) \geq \Phi(D_0)$ at all times.
- Keep $\Phi(D_i) \geq \Phi(D_0)$ for all $i$
  - usually $\Phi(D_0) = 0$ and all $\Phi(D_i) \geq 0$

Potential for a stack

- Define the potential to be the number of elements on the stack
- $D_0$ is the empty stack and therefore $\Phi(D_0) = 0$
- $\Phi$ will always be non-negative
  - therefore total amortized cost is an upper bound for the total cost

Potential for a stack

- Amortized cost of a push operation
- If the $i$-th operation is a push and there are $s$ elements on the stack
  - $\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1$
- Amortized cost of a push is
  - $ac_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 2$
Potential of a stack

- Let the \( i \)-th operation be a multi-pop and there are \( s \) elements on the stack
- Amortized cost of a multi-pop(k)
  - Let \( k' = \min(s, k) \)
- Actual cost is \( k' \), and
  \[ \Phi(D_i) - \Phi(D_{i-1}) = -k' \]
- Amortized cost of multi-pop
  \[ a_c_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 0 \]

Potential for a stack

- Amortized cost for all operations is \( O(1) \)
- Since total amortized cost is upper bound to the total worst case cost...
- Any sequence of \( n \) operations is \( O(n) \) worst-case

Amortized analysis of a counter

- \( k \)-bit binary counter
- Counts up from 0
- One operation, Increment
- Implementation:
  - Array, \( A[0..k-1] \) to hold the bits
  - low order bit in \( A[0] \)
  - number in decimal is \( x = \sum A[i] 2^i, i=0..k-1 \)
  - initially all bits are zero

Binary counter

- Operation Increment is as follows:

  ```
  Increment(A)
  j:=0
  while j < length(A) and A[j] = 1
      A[j] := 0
      j := j+1
  if j < length(A) then A[j] := 1
  ```

Binary counter

- Operation Increment is \( O(k) \) worst case
  - for a \( k \)-bit counter
  - counting bit flips
- worst case: have to flip all bits
- A sequence of \( n \) Increment operations is \( O(nk) \)
- Worst case cannot happen all the time:
  - only when number happens to have many 1s

<table>
<thead>
<tr>
<th>counter</th>
<th>binary</th>
<th>total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>00010</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>00011</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>00100</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>00101</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>00110</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>00111</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>01000</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>01001</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>01010</td>
<td>18</td>
</tr>
<tr>
<td>11</td>
<td>01011</td>
<td>19</td>
</tr>
<tr>
<td>12</td>
<td>01100</td>
<td>22</td>
</tr>
<tr>
<td>13</td>
<td>01101</td>
<td>23</td>
</tr>
<tr>
<td>14</td>
<td>01110</td>
<td>25</td>
</tr>
<tr>
<td>15</td>
<td>01111</td>
<td>26</td>
</tr>
<tr>
<td>16</td>
<td>10000</td>
<td>31</td>
</tr>
</tbody>
</table>
### Binary counter - aggregate

- How often do bits flip?
  - $A[0]$ flips every time
  - $A[1]$ flips every other time
- For $n$ increment operations
  - $A[0]$ flips $n$ times
  - $A[1]$ flips $n/2$ times
  - $A[j]$ flips $n/2^j$ times

### Binary counter - aggregate

- For $j > \log_2 n$, $A[j]$ never flips
- Total number of flips
  \[
  \sum_{j=0}^{\log_2 n} \frac{n}{2^j} < n \sum_{j=0}^{\infty} \frac{1}{2^j} = 2n
  \]
- Total worst-case for $n$ operations $O(n)$
- Amortized cost per operation $O(n)/n = O(1)$

### Binary counter - accounting

- Charge $2$ every time a bit is set to 1
  - $1$ to be used to pay for setting it to 1
  - Extra credit of $1$ to be used when it is flipped back to 0
- Increment while-loop
  - At most one bit is set to 1 for the whole loop
- Amortized cost is $O(1)$

### Binary counter - potential

- Potential method can analyze the counter even when it does not start from zero
- Potential $= ?$
  - Number of ones in the counter?