Amortized Analysis

Used a lot for "Online Programs" ---not Internet Online --- it means in comparison to "Offline" programs you do not know input at start time of program

- Not just consider one operation, but a sequence of operations on a given data structure.
- Average cost over a sequence of operations.
- Probabilistic analysis:
  - Average case running time: average over all possible inputs for one algorithm (operation).
  - If using probability, called expected running time.
- Amortized analysis:
  - No involvement of probability
  - Average performance on a sequence of operations, even some operation is expensive.
  - Guarantee average performance of each operation among the sequence in worst case.

Three Methods of Amortized Analysis

- Aggregate analysis:
  - Total cost of *n* operations/*n*,
- Accounting method:
  - Assign each type of operation an (different) amortized cost
  - overcharge some operations,
  - store the overcharge as credit on specific objects,
  - then use the credit for compensation for some later operations.
- Potential method:
  - Same as accounting method
  - But store the credit as "potential energy" and as a whole.

Example for amortized analysis

- Stack operations:
  - PUSH(S,x), $O(1)$
  - POP(S), $O(1)$
  - MULTIPOP(S,k), min(s,k)
    - **while** not STACK-EMPTY(S) and $k>0$
    - **do** POP(S)
    - $k=k-1$
- Let us consider a sequence of n PUSH, POP, MULTIPOP.
  - The worst case cost for MULTIPOP in the sequence is $O(n)$, since the stack size is at most n.
  - thus the cost of the sequence is $O(n^2)$. Correct, but not tight.

Aggregate Analysis

- In fact, a sequence of n operations on an initially empty stack cost at most $O(n)$. Why?

Each object can be POP only once (including in MULTIPOP) for each time it is PUSHed. #POPs is at most #PUSHs, which is at most n.

Thus the average cost of an operation is $O(n)/n = O(1)$.

Amortized cost in aggregate analysis is defined to be average cost.

Another example: increasing a binary counter

- Binary counter of length k, A[0..k-1] of bit array.
- INCREMENT(A)

1. $i \leftarrow 0$

2. **while** $i<k$ and A[i]=1

3. **do** A[i]$\leftarrow$0 (flip, reset)

4. $i \leftarrow i+1$

5. **if** $i<k$

6. **then** A[i]$\leftarrow$1 (flip, set)

Analysis of INCREMENT(A)

- Cursory analysis:
  - A single execution of INCREMENT takes $O(k)$ in the worst case (when A contains all 1s)
  - So a sequence of n executions takes $O(nk)$ in worst case (suppose initial counter is 0).
  - This bound is correct, but not tight.
- The tight bound is $O(n)$ for n executions.

Amortized (Aggregate) Analysis of INCREMENT(A)

Observation: The running time determined by #flips
but not all bits flip each time INCREMENT is called.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 17.2 An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Notice that the total cost is never more than twice the total number of INCREMENT operations.

A[0] flips every time, total n times.
A[1] flips every other time, $\lfloor n/2 \rfloor$ times.
A[2] flips every forth time, $\lfloor n/4 \rfloor$ times.
….
for i=0,1,…,k-1, A[i] flips $\lfloor n/2^i \rfloor$ times.

Thus total #flips is $\sum_{i=0}^{k-1} \lfloor n/2^i \rfloor$
$< n\sum_{i=0}^{\infty} 1/2^i$
$=2n$.

Amortized Analysis of INCREMENT(A)

- Thus the worst case running time is $O(n)$ for a sequence of n INCREMENTs.
- So the amortized cost per operation is $O(1)$.

Amortized Analysis: Accounting Method

- Idea:
  - Assign differing charges to different operations.
  - The amount of the charge is called amortized cost.
  - amortized cost is more or less than actual cost.
  - When amortized cost **>** actual cost, the difference is saved in specific objects as credits.
  - The credits can be used by later operations whose amortized cost **<** actual cost.
- As a comparison, in aggregate analysis, all operations have same amortized costs.

Accounting Method (cont.)

- Conditions:
  - suppose actual cost is c_i for the ith operation in the sequence, and amortized cost is c_i',
  - $\sum_{i=1}^{n} c_i' \geq \sum_{i=1}^{n} c_i$ should hold.
    - since we want to show the average cost per operation is small using amortized cost, we need the total amortized cost is an upper bound of total actual cost.
    - holds for all sequences of operations.
  - Total credits is $\sum_{i=1}^{n} c_i' - \sum_{i=1}^{n} c_i$, which should be nonnegative,
    - Moreover, $\sum_{i=1}^{t} c_i' - \sum_{i=1}^{t} c_i \geq 0$ for any $t>0$.

Accounting Method: Stack Operations

- Actual costs:
  - PUSH :1, POP :1, MULTIPOP: min(s,k).
- Let assign the following amortized costs:
  - PUSH:2, POP: 0, MULTIPOP: 0.
- Similar to a stack of plates in a cafeteria.
  - Suppose $1 represents a unit cost.
  - When pushing a plate, use one dollar to pay the actual cost of the push and leave one dollar on the plate as credit.
  - Whenever POPing a plate, the one dollar on the plate is used to pay the actual cost of the POP. (same for MULTIPOP).
  - By charging PUSH a little more, do not charge POP or MULTIPOP.
- The total amortized cost for n PUSH, POP, MULTIPOP is $O(n)$, thus $O(1)$ for average amortized cost for each operation.
- Conditions hold: total amortized cost ≥total actual cost, and amount of credits never becomes negative.

Accounting method: binary counter

- Let \$1 represent each unit of cost (i.e., the flip of one bit).
- Charge an amortized cost of \$2 to set a bit to 1.
- Whenever a bit is set, use \$1 to pay the actual cost, and store another \$1 on the bit as credit.
- When a bit is reset, the stored \$1 pays the cost.
- At any point, a 1 in the counter stores \$1, the number of 1’s is never negative, so is the total credits.
- At most one bit is set in each operation, so the amortized cost of an operation is at most \$2.
- Thus, total amortized cost of n operations is $O(n)$, and average is $O(1)$.

The Potential Method

- Same as accounting method: something prepaid is used later.
- Different from accounting method
  - The prepaid work not as credit, but as “potential energy”, or “potential”.
  - The potential is associated with the data structure as a whole rather than with specific objects within the data structure.

The Potential Method (cont.)

- Initial data structure D_0,
- n operations, resulting in $D_0, D_1, \ldots, D_n$ with costs $c_1, c_2, \ldots, c_n$.
- A potential function Φ: $\{D_i\} \rightarrow$ R (real numbers)
- $\Phi(D_i)$ is called the potential of D_i.
- Amortized cost c_i' of the ith operation is:
  - $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})$. (actual cost + potential change)
- $\sum_{i=1}^{n} c_i' = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$
- $= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$

The Potential Method (cont.)

- If $\Phi(D_n) \geq \Phi(D_0)$, then total amortized cost is an upper bound of total actual cost.
- But we do not know how many operations, so $\Phi(D_i) \geq \Phi(D_0)$ is required for any i.
- It is convenient to define $\Phi(D_0)=0$,and so $\Phi(D_i) \geq 0$, for all i.
- If the potential change is positive (i.e., $\Phi(D_i) - \Phi(D_{i-1})>0$), then c_i' is an overcharge (so store the increase as potential),
- otherwise, undercharge (discharge the potential to pay the actual cost).

Potential method: stack operation

- Potential for a stack is the number of objects in the stack.
- So $\Phi(D_0)=0$, and $\Phi(D_i) \geq 0$
- Amortized cost of stack operations:
  - PUSH:
    - Potential change: $\Phi(D_i)- \Phi(D_{i-1}) =(s+1)-s =1$.
    - Amortized cost: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})=1+1=2$.
  - POP:
    - Potential change: $\Phi(D_i)- \Phi(D_{i-1}) =(s-1) -s= -1$.
    - Amortized cost: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})=1+(-1)=0$.
  - MULTIPOP(S,k): $k'=\min(s,k)$
    - Potential change: $\Phi(D_i)- \Phi(D_{i-1}) = -k'$.
    - Amortized cost: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})=k'+(-k')=0$.
- So amortized cost of each operation is $O(1)$, and total amortized cost of n operations is $O(n)$.
- Since total amortized cost is an upper bound of actual cost, the worse case cost of n operations is $O(n)$.

Potential method: binary counter

- Define the potential of the counter after the *i*th INCREMENT is $\Phi(D_i) = b_i$, the number of 1's. clearly, $\Phi(D_i) \geq 0$.
- Let us compute amortized cost of an operation
  - Suppose the *i*th operation resets t_i bits.
  - Actual cost c_i of the operation is at most $t_i + 1$.
  - If $b_i=0$, then the *i*th operation resets all k bits, so $b_{i-1}=t_i=k$.
  - If $b_i>0$, then $b_i=b_{i-1}-t_i+1$
  - In either case, $b_i \leq b_{i-1}-t_i+1$.
  - So potential change is $\Phi(D_i) - \Phi(D_{i-1}) \leq b_{i-1}-t_i+1-b_{i-1}=1-t_i$.
  - So amortized cost is: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq t_i +1+1-t_i=2$.
- The total amortized cost of n operations is $O(n)$.
- Thus worst case cost is $O(n)$.

Amortized analyses: dynamic table

- A nice use of amortized analysis
- Table-insertion, table-deletion.
- Scenario:
  - A table –maybe a hash table
  - Do not know how large in advance
  - May expend with insertion
  - May contract with deletion
  - Detailed implementation is not important
- Goal:
  - $O(1)$ amortized cost.
  - Unused space always ≤ constant fraction of allocated space.

Dynamic table

- ***Load factor*** $\alpha = num/size$, where num = # items stored, $size$ = allocated size.
- If $size = 0$, then $num = 0$. Call $\alpha = 1$.
- Never allow $\alpha > 1$.
- Keep α >a constant fraction → goal (2).

Dynamic table: expansion with insertion

- **Table expansion**
- Consider only insertion.
- When the table becomes full, double its size and reinsert all existing items.
- Guarantees that $\alpha \geq 1/2$.
- Each time we actually insert an item into the table, it's an ***elementary insertion***.

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

```
TABLE-INSERT(T, x)
 1  if size[T] = 0
 2     then allocate table[T] with 1 slot
 3          size[T] ← 1
 4  if num[T] = size[T]
 5     then allocate new-table with 2 · size[T] slots
 6          insert all items in table[T] into new-table      Num[t] ele. insertion
 7          free table[T]
 8          table[T] ← new-table
 9          size[T] ← 2 · size[T]
10  insert x into table[T]                                   1 ele. insertion
11  num[T] ← num[T] + 1
```

Initially, $num[T] = size[T] = 0$.

Aggregate analysis

- ***Running time:*** Charge 1 per elementary insertion. Count only elementary insertions,
- since all other costs together are constant per call.
- c_i = actual cost of ith operation
  - If not full, $c_i = 1$.
  - If full, have $i - 1$ items in the table at the start of the ith operation. Have to copy all $i - 1$ existing items, then insert ith item, $\Rightarrow c_i = i$
- *Cursory analysis:* n operations $\Rightarrow c_i = O(n) \Rightarrow O(n^2)$ time for n operations.
- Of course, we don't always expand:
  - $c_i = \begin{cases} i & \text{if } i - 1 \text{ is exact power of 2}, \\ 1 & \text{otherwise}. \end{cases}$
- *So total cost* $= \sum_{i=1}^{n} c_i \le n + \sum_{i=0}^{\log(n)} 2^i \le n + 2n = 3n$
- Therefore, **aggregate analysis** says amortized cost per operation = 3.

Accounting analysis

- Charge \$3 per insertion of x.
  - \$1 pays for x's insertion.
  - \$1 pays for x to be moved in the future.
  - \$1 pays for some other item to be moved.
- Suppose we've just expanded, *size* = m before next expansion, *size* = $2m$ after next expansion.
- Assume that the expansion used up all the credit, so that there's no credit stored after the expansion.
- Will expand again after another m insertions.
- Each insertion will put \$1 on one of the m items that were in the table just after expansion and will put \$1 on the item inserted.
- Have \$$2m$ of credit by next expansion, when there are $2m$ items to move. Just enough to pay for the expansion, with no credit left over!

Potential method

- **Potential method**
- $\Phi(T) = 2 \cdot num[T] - size[T]$
- Initially, $num = size = 0 \Rightarrow \Phi = 0$.
- • Just after expansion, $size = 2 \cdot num \Rightarrow \Phi = 0$.
- Just before expansion, $size = num \Rightarrow \Phi = num$ $\Rightarrow$ have enough potential to pay for moving all items.
- Need $\Phi \geq 0$, always.
- Always have
  - $size \geq num \geq ½\, size \Rightarrow 2 \cdot num \geq size \Rightarrow \Phi \geq 0$.

Potential method

- ***Amortized cost of ith operation:***
  - $num_i = num$ after ith operation ,
  - $size_i = size$ after ith operation ,
  - $\Phi_i = \Phi$ after ith operation .
- If no expansion:
  - $size_i = size_{i-1}$,
  - $num_i = num_{i-1} + 1$,
  - $ci = 1$.
- Then we have
  - $C_i' = c_i + \Phi_i - \Phi_{i-1} = 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) = 3.$
- If expansion:
  - $size_i = 2size_{i-1}$,
  - $size_{i-1} = num_{i-1} = num_i - 1$,
  - $c_i = num_{i-1} + 1 = num_i$.
- Then we have
- $C_i' = c_i + \Phi_i - \Phi_{i-1} = num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) = num_i + (2num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) = num_i + 2 - (num_i - 1) = 3$

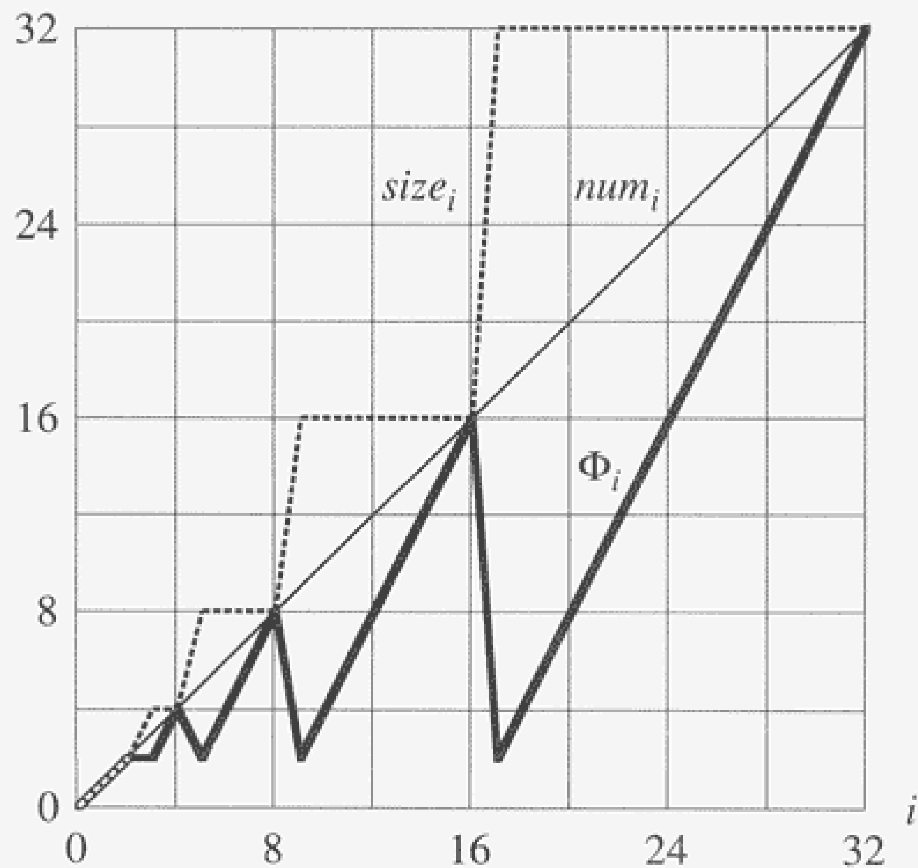


Figure 17.3 The effect of a sequence of n TABLE-INSERT operations on the number num_i of items in the table, the number $size_i$ of slots in the table, and the potential $\Phi_i = 2 \cdot num_i - size_i$, each being measured after the ith operation. The thin line shows num_i, the dashed line shows $size_i$, and the thick line shows Φ_i. Notice that immediately before an expansion, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table. Afterwards, the potential drops to 0, but it is immediately increased by 2 when the item that caused the expansion is inserted.

Expansion and contraction

- **Expansion and contraction**
- When α drops too low, contract the table.
  - Allocate a new, smaller one.
  - Copy all items.
- Still want
  - α bounded from below by a constant,
  - amortized cost per operation = $O(1)$.
- Measure cost in terms of elementary insertions and deletions.

Obvious strategy

- Double size when inserting into a full table (when $\alpha = 1$, so that after insertion α would become <1).
- Halve size when deletion would make table less than half full (when $\alpha = 1/2$, so that after deletion α would become >= 1/2).
- Then always have $1/2 \le \alpha \le 1$.
- Suppose we fill table.
  - Then insert $\Rightarrow$ double
  - 2 deletes $\Rightarrow$ halve
  - 2 inserts $\Rightarrow$ double
  - 2 deletes $\Rightarrow$ halve
  - . . .
  - Cost of each expansion or contraction is $\Theta(n)$, so total n operation will be $\Theta(n^2)$.
- Problem is that: Not performing enough operations after expansion or contraction to pay for the next one.

Simple solution

- Double as before: when inserting with $\alpha = 1 \Rightarrow$ after doubling, $\alpha = 1/2$.
- Halve size when deleting with $\alpha = 1/4 \Rightarrow$ after halving, $\alpha = 1/2$.
- Thus, immediately after either expansion or contraction, have $\alpha = 1/2$.
- Always have $1/4 \leq \alpha \leq 1$.
- ***Intuition:***
- Want to make sure that we perform enough operations between consecutive expansions/contractions to pay for the change in table size.
- Need to delete half the items before contraction.
- Need to double number of items before expansion.
- Either way, number of operations between expansions/contractions is at least a constant fraction of number of items copied.

Potential function

- $\Phi(T) = \begin{cases} 2num[T] - size[T] & \text{if } \alpha \geq 1/2 \\ size[T]/2 - num[T] & \text{if } \alpha < 1/2 \end{cases}$.
- T empty $\Rightarrow \Phi = 0$.
- $\alpha \geq 1/2 \Rightarrow num \geq 1/2 size \Rightarrow 2num \geq size \Rightarrow \Phi \geq 0$.
- $\alpha < 1/2 \Rightarrow num < 1/2 size \Rightarrow \Phi \geq 0$.

intuition

- measures how far from $\alpha = 1/2$ we are.
  - $\alpha = 1/2 \Rightarrow \Phi = 2num - 2num = 0$.
  - $\alpha = 1 \Rightarrow \Phi = 2num - num = num$.
  - $\alpha = 1/4 \Rightarrow \Phi = size/2 - num = 4num/2 - num = num$.
- Therefore, when we double or halve, have enough potential to pay for moving all *num* items.
- Potential increases linearly between $\alpha = 1/2$ and $\alpha = 1$, and it also increases linearly between $\alpha = 1/2$ and $\alpha = 1/4$.
- Since α has different distances to go to get to 1 or 1/4, starting from 1/2, rate of increase differs.
- For α to go from 1/2 to 1, *num* increases from $size/2$ to *size*, for a total increase of $size/2$. Φ increases from 0 to *size*. Thus, Φ needs to increase by 2 for each item inserted. That's why there's a coefficient of 2 on the $num[T]$ term in the formula for when $\alpha \geq 1/2$.
- For α to go from 1/2 to 1/4, *num* decreases from $size/2$ to $size/4$, for a total decrease of $size/4$. Φ increases from 0 to $size/4$. Thus, Φ needs to increase by 1 for each item deleted. That's why there's a coefficient of −1 on the $num[T]$ term in the formula for when $\alpha < 1/2$.

Amortized cost for each operation

- Amortized costs: more cases
  - insert, delete
  - $\alpha \geq 1/2$, $\alpha < 1/2$ (use α_i, since α can vary a lot)
  - *size* does/doesn’t change

Splay tree

- A binary search tree (not balanced)
- Height may be larger than log n, even n-1.
- However a sequence of n operations takes O(nlog n).
- Assumptions: data values are distinct and form a totally order set
- Operations:
  - Member(i,S)
  - Insert(i,S)
  - Delete(i,S)
  - Merge(S,S')
  - Split(i,S)
  - All based on
    - splay(i,S), reorganize tree so that *i* to be root if $i \in S$, otherwise, the new root is either $\max\{k \in S \mid k<i\}$ or $\min\{k \in S \mid k>i\}$

Splay tree (cont.)

- For examples,
  - merge(S,S')
    - Call Splay(∞, S) and then make S' the right child
  - Delete(i,S), call Splay(i,S), remove I, then merge(left(i), right(i)).
  - Similar for others.
  - Constant number of splays called.

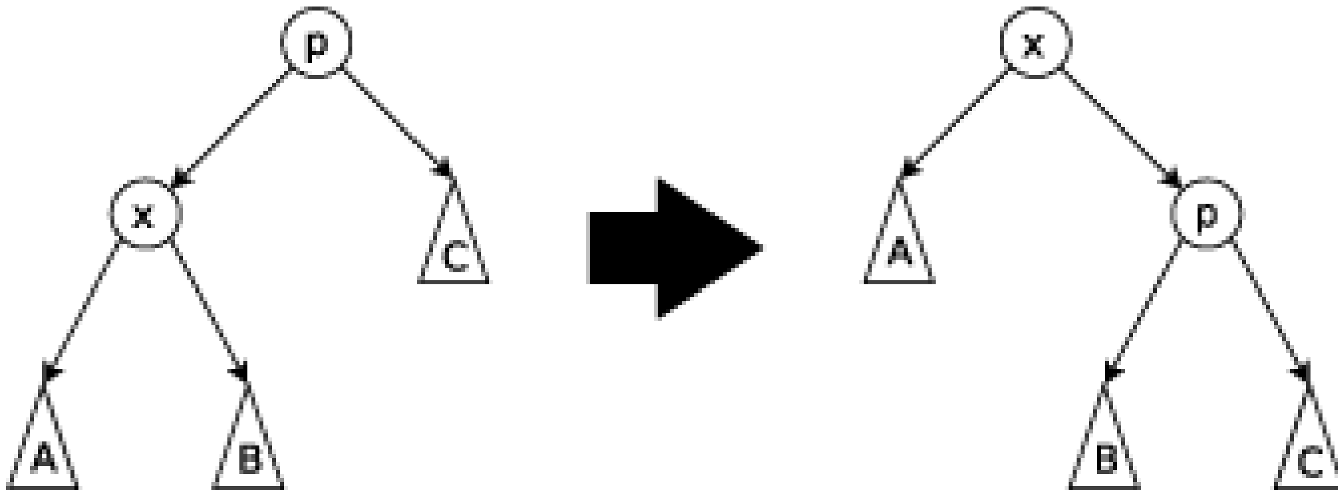
Splay tree (cont.)

- Splay operation is based on basic rotate(x) operation (either left or right).
- Three cases:
  - p is the parent of x and x has not grandparent
    - rotate(x)
  - x is the left (or right) child of p and p is the left (or right) child of g,
    - Rotate(p) and then rotate(x)\
  - x is the left (or right) child of p and p is the right (or left) child of g,
    - rotate(x) and then rotate(x)

Case 1: Zip Step

p is the parent of x and x has not grandparent

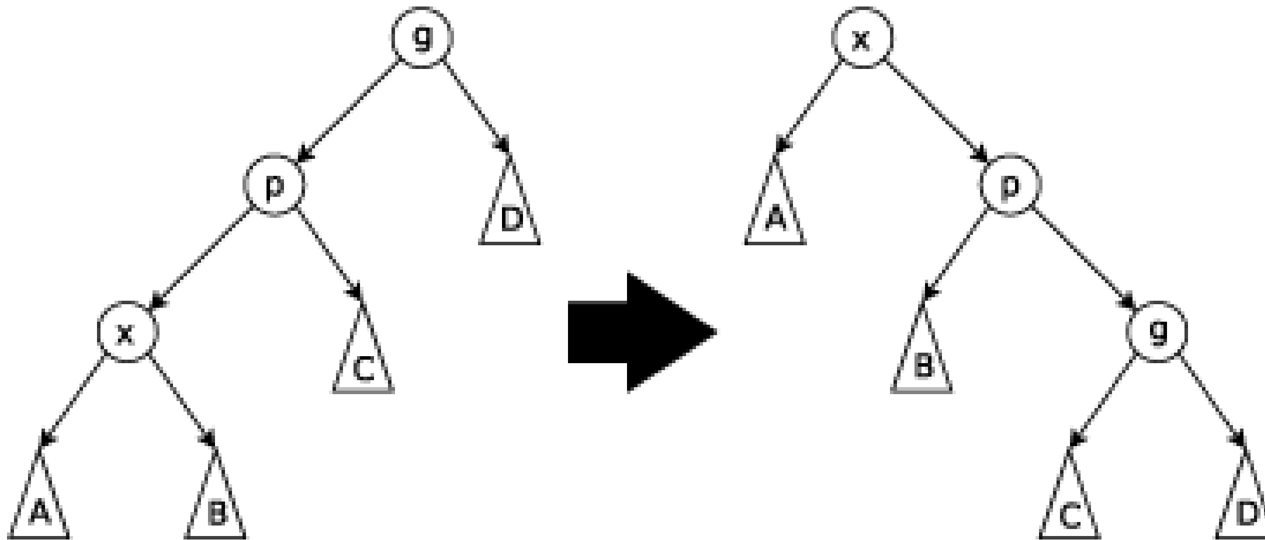
rotate(x)



Case 2: Zip-Zip Step

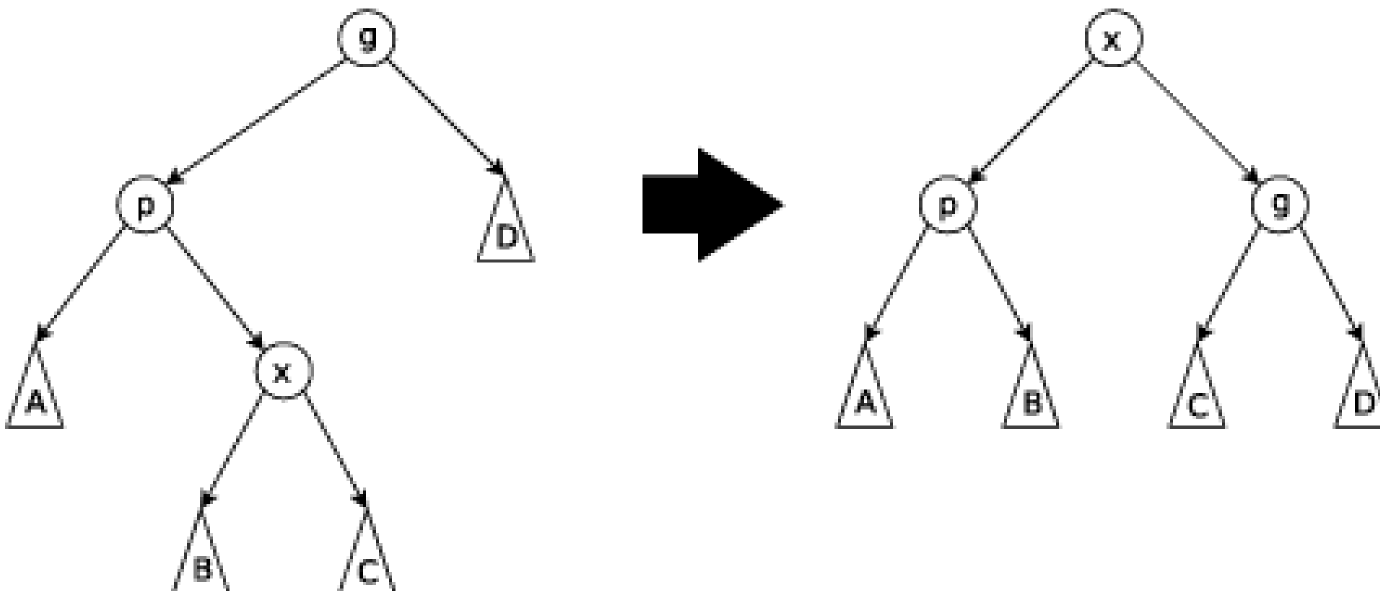
x is the left (or right) child of p and p is the left (or right) child of g,

Rotate(p) and then rotate(x)\



Case 3: Zip-Zag Step

x is the left (or right) child of p and p is the right (or left) child of g,
rotate(x) and then rotate(x)



Splay tree (cont.)

- Credit invariant: Node x always has at least log $\mu(x)$ credits on deposit.
  - Where $\mu(S)=\log(|S|)$ and $\mu(x)=\mu(S(x))$
- Lemma:
  - Each operation splay(x,S) requires no more than $3(\mu(S)-\mu(x))+1$ credits to perform the operation and maintain the credit invariant.
- Theorem:
  - A sequence of *m* operations involving *n* inserts takes time $O(m\log(n))$.
- Read more about this …

Summary

- Amortized analysis
  - Different from probabilistic analysis
- Three methods and their differences
- how to analyze