Amortization

- An analysis technique; *not* a design technique
- Why include it in discussion of design techniques?
  - Amortization has a strong influence on data structure design
- Use of amortization often leads to
  - Simpler algorithms
  - Improved overall runtime

Basic idea

- It doesn't matter how long an individual operation takes if the total time summed over all operations is low

A Simple Example: Clearable Stack

- ADT ClearableStack
  - Standard operations: push(), pop(), isEmpty()
  - Plus a new operation: clear() = remove everything from the Stack
  - Suppose we implement clear() by using multiple pops
- What is the worst-case time for n ClearableStack operations starting with an empty Stack?
- An analysis:
  - clear() can take O(n) time in the worst-case
  - Thus, worst-case time for n operations is $O(n^2)$
- A better analysis
  - There is at most one pop() for each insert()
  - Thus total time is proportional to number of inserts = O(n)
- Thus the *amortized* time per operation is O(1)

You've Already Seen Amortization

- Hashtable doubling
  - Some insert() operations take time O(n) due to table doubling
  - Amortized expected time per op is still O(1)
- Union/Find with weighted union and path compression
  - Worst-case time for union() is O(1)
  - Worst-case time for find() is O(log n)
  - Worst-case time for a sequence of n operations (starting from singletons) is $O(n\ \alpha(n))$
  - *Amortized* time per operation is $O(\alpha(n))$

Amortization Methods

- Aggregate Method
  - Determine total time T(n) for all n operations
  - Amortized time is T(n)/n
- Accounting Method
  - Uses a scheme of *credits* charged to each operation
  - Credits "reside" within data structure and are available to pay for operations
- Potential Method
  - Maintains a "potential energy" of the data structure
- Amortization has no effect on the algorithm itself
  - Strictly an analysis technique
- We'll mostly use the Accounting Method
  - No credits appear in the algorithm
  - Credits are strictly an analysis technique
  - Credits provide a way of trading running-time between operations

Clearable Stack using the Accounting Method

- Each operation (push, pop, clear) is *charged* some number of credits
- The credits are used to *pay* for computer time
  - It costs one credit to do a constant time operation
  - Total time is proportional to total number of credits
  - Amortized time for an operation is determined by the credits charged for that operation

Analysis

- Each operation is charged 2 credits
  - O(1) amortized time per op
- For push()
  - Costs one credit to do the push()
  - The other credit is "stored" with the item on the Stack
- For clear()
  - The credits in the Stack are used to pay for clearing the Stack

Another Example: Java Vectors

- java.util.Vector
  - Intuitively, an expandable array
  - Operations: addElement(), elementAt(), setElementAt(), others
  - addElement() increases the size of the Vector
- Implementation: uses arrays with table doubling
  - When the array gets full, allocate a new array, twice as large, and copy all the data to the new array
- addElement() can take O(n) worst-case time
- Claim: the amortized time per operation is O(1)

Analysis of Java Vectors

- Charge 7 credits for each operation
  - Amortized time per operation is O(1)
- addElement() (when no doubling occurs) uses one credit to pay for itself and stores 6 credits with the item
- The other operations use their 7 credits to pay for themselves with the extra 6 credits thrown away
- When a table doubling occurs (say at size 2^i)
  - It takes $3*2^i$ credits to pay for creating the new array and copying all the data
  - We use the credits stored at positions 2^{i-1} to 2^i
- Note that the previous doubling occurred when the size was 2^{i-1}
  - Thus the credits stored at 2^{i-1} to 2^i are unspent

Remember Union/Find

- Worst-case time for a sequence of n operations (starting from singletons) is $O(n\ \alpha(n))$
- This bound is too hard to prove for this course, but we can use amortization to prove a bound that is almost as good:
  - Worst-case time for a sequence of n operations (starting from singletons) is O(n log*n)
  - log*n does not grow as slowly as $\alpha(n)$, but it is less than about 5 for all reasonable inputs

log*(n)

- Define tower(k) =
  1 if k is 0
  $2^{\text{tower}(k-1)}$ if k is > 0
- Define log*(n) = min{k: tower(k) ≥ n}
  - log*(n) is always an integer
  - log*(n) is the number of times we have to apply log (rounding down) to get to 1
  - log*n = k iff tower(k-1) < n ≤ tower(k)
- There are small variations in the way log*(n) is defined, differing usually by ±1

k	**tower(*k*)**
0	1
1	2
2	$2^2 = 4$
3	$2^4 = 16$
4	$2^{16} = 65{,}536$
5	$2^{65{,}536}$

Rank Groups and the Rank of a Node

- Recall that we kept track of the size (weight) of each tree

Definition of *rank(v)*
- For a root node v, rank(v) is defined to be $\lfloor \log(\text{size}(v)) \rfloor$
- For a nonroot node w, rank(w) is defined to be the largest rank achieved by w before it became a nonroot

Definition: Nodes u and v are in *rank group* g if log*(rank(u)) = log*(rank(v)) = g

- Note that
  - nodes can be in the same rank group even when their ranks are different
  - the largest possible rank group is $\log^*(\lfloor \log n \rfloor) = \log^* n - 1$

Rank Properties

Lemma: If node v is the parent of node w then rank(v) > rank(w)

Proof:
- A node becomes parent of another node only via union
- During union:
  - if one tree's rank is larger then it becomes the root
  - if the ranks are tied then the new root's rank will increase
- In either case the parent ends up with larger rank

Lemma: There are at most $n/2^r$ nodes of rank r where n is the number of nodes

Proof:
- When node v achieved rank r, v had at least 2^r nodes in its tree
- Observe that those nodes cannot be part of any other tree that achieves rank r
- Thus there are at most $n/2^r$ nodes that achieve rank r

Amortization Scheme for Union/Find

Credits charged
- one credit is charged for each Union
- log*(n) credits are charged for each Find
- n log*(n) credits are initially charged to the set of nodes
- It's clear that a total of O(n log*n) credits are charged for n operations

Credits paid
- one credit is paid to do a Union
- for Find, we follow a path, moving from each node v to the parent of v
  - Pay a credit from the Find if v is the root, if the parent of v is the root, or if the parent of v is not in v's *rank-group*
  - Pay a credit from v otherwise

Do We Have Enough Credits?

How many credits are paid from a Find?

- We pay from the Find when we (1) change rank groups as we move up the path or (2) reach the root
- How many rank groups are there?
  - There are at most log*n - 1 rank-groups (not counting group 0, the singletons)
- Thus Find pays at most log*n credits

How many credits are paid from nodes?

- If node v pays then we know
  - v is not a root (thus it's rank will no longer change)
  - v's parent is in the same rank-group
  - v is going to have a new parent (with a higher rank) due to path compression
- Thus v pays at most a number of credits equal to the number of different ranks in its rank group

How Many Credits are Paid in Total from All the Nodes?

- We know each node pays an amount ≤ the number of ranks in its rank group
- How many ranks in a rank group?
  - A rank r is in rank-group g iff log*r = g
  - This is equivalent to saying tower(g-1) < r ≤ tower(g)
  - Thus the number of ranks in rank-group g is tower(g) – tower(g-1)
- Thus a node in rank-group g pays at most tower(g)-tower(g-1) credits
- How many nodes in rank-group g?
  - There are $n/2^r$ nodes of rank r (see the earlier lemma)
  - # nodes in rank-group g
    $= n/2^{tower(g-1)+1} + n/2^{tower(g-1)+2} +... + n/2^{tower(g)}$
    $< n/2^{tower(g-1)} (1/2 + 1/4 +...)$
    $= n/2^{tower(g-1)}$
    $= n/tower(g)$

Total Node-Credits Paid

- We know a node in rank-group g pays less than tower(g) credits
  - It actually pays at most tower(g)-tower(g-1) credits
- Further, we know there are at most n/tower(g) nodes in rank-group g
- Thus, each rank-group pays at most n credits
- Note that there are log*n different rank groups
- Thus, the nodes pay at most n log* n credits
- We now know that the number of credits paid is less than the number of credits charged
  - The proof is complete
  - Total time for n Union/Find operations starting from singletons is O(n log*n)