CS583 Lecture 12

Jana Kosecka

Amortized/Accounting Analysis
Disjoint Sets

Previously

- Dynamic Programming
- Greedy Algorithms

Slight digression

- Amortized analysis
- Disjoint sets

Amortized Analysis

• *Amortized analysis* computes running time of a sequence of n operations

• Different then average analysis

Three techniques
• Aggregate analysis
• Accounting method
• Potential method

Aggregate analysis

• Stack example *PUSH(S,x)* and *POP(x)* each takes *O(1)*

• Add operation *MULTI-POP(S,k)* pop multiple elements

• Cost of *MULTI-POP(S,k)* *min(s,k), where* s is the number of elements in the stack

• Cost of a sequence of n operations *PUSH, POP, MULTIPOP*

• Worst case for n operation $O(n^2)$ (why ?) – not tight
• Better cost for n operations $O(n)$ (why ?)
• Amortized cost per operation $O(n)/n = O(1)$.

Accounting Analysis

- Another method for analyzing time to perform sequence of operations
- If we have more then one type of operation, each operation can have different amortized cost
- *Accounting method*
  Charge each operation an amortized cost
  Amount not used stored in "bank"
  Later operations can used stored money
  Balance must not go negative
- Book also discusses *potential method*
  But we will not discuss it here
- Example: Dynamic Tables
- Adjust the size of the table on the fly

Accounting Method Example: Dynamic Tables

- Implementing a table (e.g., hash table) for dynamic data, want to make it small as possible
- Problem: if too many items inserted, table may be too small
- Idea: allocate more memory as needed

Dynamic Tables

1. Init table size $m = 1$
2. Insert elements until number $n > m$
3. Generate new table of size $2m$ (double the size)
4. Reinsert old elements into new table (need table to be in continuous block of memory)
5. (back to step 2)

- *What is the worst-case cost of an insert?*
- One insert can be costly, but the total?

- Analyze cost on n *Insert()* 's of initially empty table

Analysis Of Dynamic Tables

- Let c_i = cost of i-th insert
- $c_i = i$ if i-1 is exact power of 2, 1 otherwise
- Example:

Operation	Table Size	Cost	
`Insert(1)`	1	1	1

Analysis Of Dynamic Tables

- Let c_i = cost of *i*-th insert
- c_i = *i* if i-1 is exact power of 2, 1 otherwise
- Example:

Operation	Table Size	Cost	
Insert(1)	1	1	1
Insert(2)	2	1 + 1	2

Analysis Of Dynamic Tables

- Let c_i = cost of *i*-th insert
- c_i = *i* if i-1 is exact power of 2, 1 otherwise
- Example:

Operation	Table Size	Cost	
Insert(1)	1	1	1
Insert(2)	2	1 + 1	2
Insert(3)	4	1 + 2	3

Analysis Of Dynamic Tables

- Let c_i = cost of *i*-th insert
- c_i = *i* if i-1 is exact power of 2, 1 otherwise
- Example:

Operation	Table Size	Cost
Insert(1)	1	1
Insert(2)	2	1 + 1
Insert(3)	4	1 + 2
Insert(4)	4	1

Analysis Of Dynamic Tables

- Let c_i = cost of *i*-th insert
- c_i = *i* if i-1 is exact power of 2, 1 otherwise
- Example:

Operation	Table Size	Cost
Insert(1)	1	1
Insert(2)	2	1 + 1
Insert(3)	4	1 + 2
Insert(4)	4	1
Insert(5)	8	1 + 4

Analysis Of Dynamic Tables

- Let c_i = cost of *i*-th insert
- c_i = *i* if i-1 is exact power of 2, 1 otherwise
- Example:

Operation	Table Size	Cost
Insert(1)	1	1
Insert(2)	2	1 + 1
Insert(3)	4	1 + 2
Insert(4)	4	1
Insert(5)	8	1 + 4
Insert(6)	8	1

Analysis Of Dynamic Tables

- Let c_i = cost of *i*-th insert
- c_i = *i* if i-1 is exact power of 2, 1 otherwise
- Example:

Operation	Table Size	Cost
Insert(1)	1	1
Insert(2)	2	1 + 1
Insert(3)	4	1 + 2
Insert(4)	4	1
Insert(5)	8	1 + 4
Insert(6)	8	1
Insert(7)	8	1

Analysis Of Dynamic Tables

- Let c_i = cost of *i*-th insert
- c_i = *i* if i-1 is exact power of 2, 1 otherwise
- Example:

Operation	Table Size	Cost
Insert(1)	1	1
Insert(2)	2	1 + 1
Insert(3)	4	1 + 2
Insert(4)	4	1
Insert(5)	8	1 + 4
Insert(6)	8	1
Insert(7)	8	1
Insert(8)	8	1

Analysis Of Dynamic Tables

- Let c_i = cost of *i*-th insert
- c_i = *i* if i-1 is exact power of 2, 1 otherwise
- Example:

Operation	Table Size	Cost
Insert(1)	1	1
Insert(2)	2	1 + 1
Insert(3)	4	1 + 2
Insert(4)	4	1
Insert(5)	8	1 + 4
Insert(6)	8	1
Insert(7)	8	1
Insert(8)	8	1
Insert(9)	16	1 + 8

Aggregate Analysis

- n Insert() operations cost

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lg n} 2^j = n + (2n - 1) < 3n$$

- At most n operations are of cost 1 + costs of expansions
- Expansion happens only where (i-1) is power of 2
- Average cost of operation
  = (total cost)/(# operations) < 3
- Asymptotically, then, a dynamic table costs the same as a fixed-size table
- Both O(1) per *Insert()* operation

Accounting Analysis

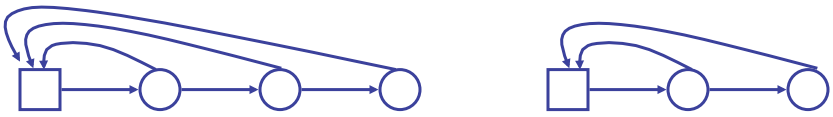
- We have shown that amortized cost is 3 per *Insert()*
- Each *Insert()* 'pays' 3
- By the time the table is full – each item paid some amount
- There is enough credit to expand the table and re-insert each element

Disjoint Sets

Chap 21

Disjoint Set Union

- So how do we implement disjoint-set union?
  Naïve implementation: use a linked list to represent each set:

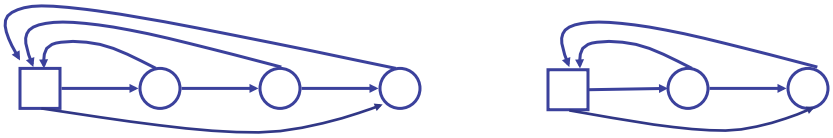


Set representative

MakeSet(): ??? time
FindSet(): ??? time
Union(A,B): "copy" elements of A into B: ??? time

Disjoint Set Union

- So how do we implement disjoint-set union?
  Naïve implementation: use a linked list to represent each set:



MakeSet(): O(1) time
FindSet(): O(1) time
Union(A,B): "copy" elements of A into B: O(A) time
Why ?
How long can a single Union() take?
How long will n Union()'s take?

Disjoint Set Union: Analysis

- Worst-case analysis: $O(n^2)$ time for n Union's

Union(S_1, S_2)	"copy"	1 element
Union(S_2, S_3)	"copy"	2 elements
…		
Union(S_{n-1}, S_n)	"copy"	n-1 elements
		$O(n^2)$

  - Appending largest set to smaller – worst case
  - Average per operation $O(n)$
  - (2n-1) operations total cost $O(n^2)$ (n MakeSet(s) and n-1 Union(s))
- Improvement: always copy smaller into larger
  Why will this make things better?
  What is the worst-case time of Union()?
- But now n Union's take only $O(n \lg n)$ time!

Amortized Analysis

- *Amortized analysis* computes average times without using probabilities
- Worst case: Each time element is copied, element in smaller set must have the pointer updated

1st time	resulting set size at least 2 members	≥ 2
2nd time		≥ 4
(lg n)-th time		≥ n

- With our new Union(), any individual element is copied at most *lg n* times when forming the complete set from 1-element sets
- After *lg n* times the resulting set will have already n numbers
- For n Union operations time spent updating objects pointers
- *O(n lg n)*

Amortized Analysis of Disjoint Sets

- Since we have n elements each copied at most lg n times, n Union()'s takes *O(n lg n)* time
- We say that each Union() takes *O(lg n) amortized time*
  Financial term: imagine paying *$(lg n)* per Union() operation

  At first we are overpaying; initial Union *$O(1)*
  But we accumulate enough $ in bank to pay for later expensive *O(n)* operation.

  Important: amount in bank never goes negative

Amortized Analysis of Disjoint Sets

- Since we have n elements each copied at most *lg n* times, n Union()'s takes *O(n lg n)* time
- Therefore we say the *amortized cost* of a Union() operation is *O(lg n)*
- This is the *aggregate method* of amortized analysis:
- n operations take time *T(n)*
- Average cost of an operation = *T(n)/n*

- In this style of analysis the amortized cost is applied to each operation, although different operations may have different costs