Algorithms

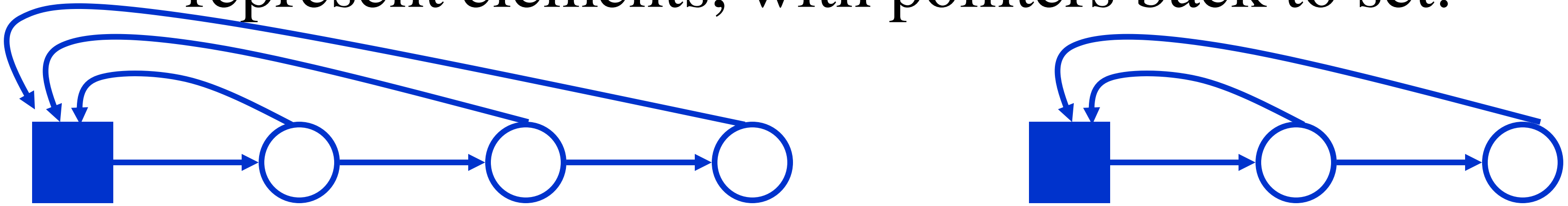
Amortized Analysis

Review: Running Time of Kruskal's Algorithm

- Expensive operations:
  - Sort edges: O(E lg E)
  - O(V) MakeSet()'s
  - O(E) FindSet()'s
  - O(V) Union()'s
- Upshot:
  - Comes down to efficiency of disjoint-set operations, particularly Union()

Review: Disjoint Set Union

- So how do we represent disjoint sets?
  - Naïve implementation: use a linked list to represent elements, with pointers back to set:



    - MakeSet(): O(1)
    - FindSet(): O(1)
    - Union(A,B): "Copy" elements of A into set B by adjusting elements of A to point to B: O(A)
  - *How long could n Union()s take?*

Review: 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)

- Improvement: always copy smaller into larger
  - *How long would above sequence of Union’s take?*
  - Worst case: n Union’s take O(n lg n) time
  - Proof uses amortized analysis

Review:
Amortized Analysis of Disjoint Sets

- If elements are copied from the smaller set into the larger set, an element can be copied at most *lg n* times
  - Worst case: Each time copied, element in smaller set

1st time	resulting set size	≥ 2
2nd time		≥ 4
…		
(lg n)th time		$\geq n$

Review:
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

Amortized Analysis: Accounting Method

- *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 won’t worry about it here

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

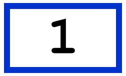
1. Init table size $m = 1$
2. Insert elements until number $n > m$
3. Generate new table of size $2m$
4. Reinsert old elements into new table
5. (back to step 2)

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

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:

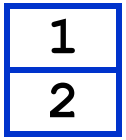
Operation	Table Size	Cost
`Insert(1)`	`1`	`1`



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:

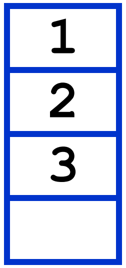
Operation	Table Size	Cost
`Insert(1)`	`1`	`1`
`Insert(2)`	`2`	`1 + 1`



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:

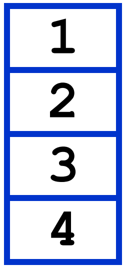
Operation	Table Size	Cost
`Insert(1)`	1	1
`Insert(2)`	2	1 + 1
`Insert(3)`	4	1 + 2



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:

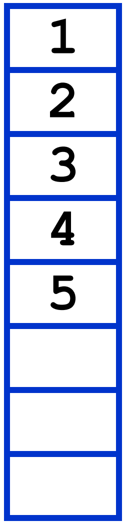
Operation	Table Size	Cost
`Insert(1)`	`1`	`1`
`Insert(2)`	`2`	`1 + 1`
`Insert(3)`	`4`	`1 + 2`
`Insert(4)`	`4`	`1`



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:

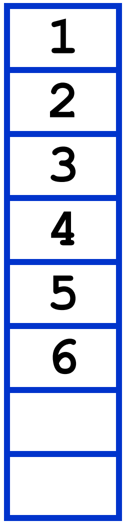
Operation	Table Size	Cost
`Insert(1)`	`1`	`1`
`Insert(2)`	`2`	`1 + 1`
`Insert(3)`	`4`	`1 + 2`
`Insert(4)`	`4`	`1`
`Insert(5)`	`8`	`1 + 4`



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:

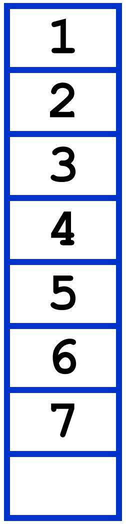
Operation	Table Size	Cost
`Insert(1)`	1	1
`Insert(2)`	2	1 + 1
`Insert(3)`	4	1 + 2
`Insert(4)`	4	1
`Insert(5)`	8	1 + 4
`Insert(6)`	8	1



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:

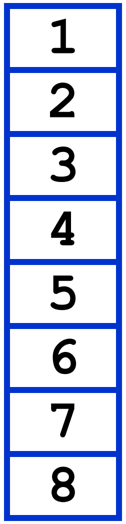
Operation	Table Size	Cost
`Insert(1)`	1	1
`Insert(2)`	2	1 + 1
`Insert(3)`	4	1 + 2
`Insert(4)`	4	1
`Insert(5)`	8	1 + 4
`Insert(6)`	8	1
`Insert(7)`	8	1



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:

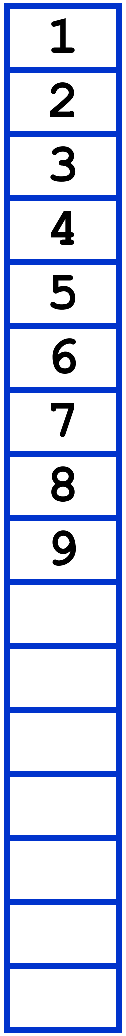
Operation	Table Size	Cost
`Insert(1)`	1	1
`Insert(2)`	2	1 + 1
`Insert(3)`	4	1 + 2
`Insert(4)`	4	1
`Insert(5)`	8	1 + 4
`Insert(6)`	8	1
`Insert(7)`	8	1
`Insert(8)`	8	1



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

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

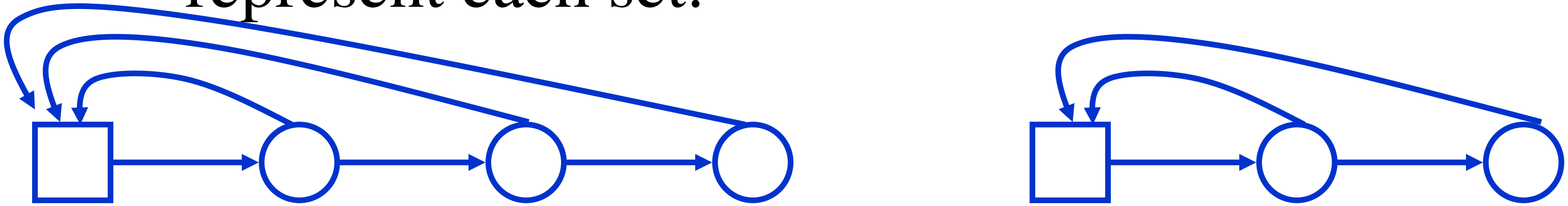
- Average cost of operation
= (total cost)/(# operations) < 3
- Asymptotically, then, a dynamic table costs the same as a fixed-size table
  - Both O(1) per Insert operation

Accounting Analysis

- Charge each operation \$3 amortized cost
  - Use \$1 to perform immediate Insert()
  - Store \$2
- When table doubles
  - \$1 reinserts old item, \$1 reinserts another old item
  - Point is, we’ve already paid these costs
  - Upshot: constant (amortized) cost per operation

Disjoint Set Union

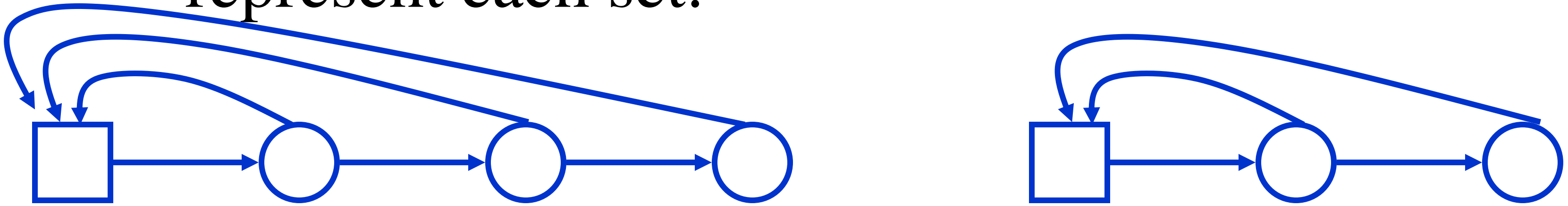
- So how do we implement disjoint-set union?
  - Naïve implementation: use a linked list to represent each set:



    - 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
  - *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
		$O(n^2)$

- 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 of Disjoint Sets

- *Amortized analysis* computes average times without using probability
- With our new Union(), any individual element is copied at most *lg n* times when forming the complete set from 1-element sets
  - Worst case: Each time copied, element in smaller set

1st time	resulting set size	≥ 2
2nd time		≥ 4
…		
(lg n)th time		$\geq n$

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
  - 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