Two Easy Applications of Amortized Analysis:

1. **Making Static Data Structures semi-dynamic**
2. **Dynamic Hash Tables**

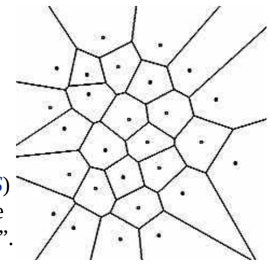
These slides are based on slides by Charles Leiserson and Carola Wenk

Making Static structre Dynamic (problem 17-2 in the text)

Last meeting we introduced the **Voronoi Diagram.**

Given: A set $S=\{s_1 \ldots s_n\}$ of points (sites) in 2D.

The Voronoi Diagram VD(S) is the plane – partition of the plane into "cell" or "regions".

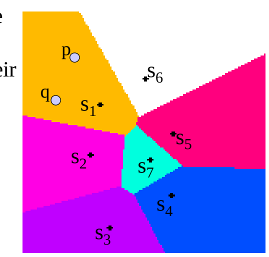


Voronoi Diagram – cont.

The Voronoi Diagram VD(S) is the subdivision of the plane so that two point lie in the same region of VD(S) **iff** their nearest site is the same.

Example:
$S=\{s_{1 \ldots} s_7\}$ set of sites.

Point p and q lie in the same cell since the nearest site for both points is the site s_1.



History

Informal use of Voronoi diagrams can be traced back to Descartes in 1644.

Dirichlet used 2-dimensional and 3-dimensional Voronoi diagrams in his study of quadratic forms in 1850.

British physician John Snow used a Voronoi diagram in 1854 to illustrate how the majority of people who died in the Soho cholera epidemic lived closer to the infected Broad Street pump than to any other water pump.

Voronoi diagrams are named after Russian mathematician Georgy Fedoseevich Voronoi

Using VD(S) – some facts

Let $n=|S|$.
VD(S) can be constructed in $O(n \log n)$.

Moreover, we can create in time $O(n \log n)$ a "point location data structure" so that once a query point q is given, we can find the nearest site to q in time $O(\log n)$.

By abusing notation, in this context, we call VD(S) to the whole structure.

This structure is **static** – we cannot add sites.

Static Dynamic and Semi-Dynamic

Consider a data structure D constructed on some input.

Def: D is **static,** if once a new point is inserted or deleted, we need to constructed D from scratch (takes super-linear time ($\Omega(n)$)).
Example: Sorted array.

D is **dynamic** if once a point is inserted **or** deleted, we can update the D in sub-linear time. (better than $O(n)$).
Example – red-black tree/AVL tree.

D is **semi-dynamic** if once a point is inserted, we can update D in sub-linear time. (better than $O(n)$).

General technique for Making static DS Semi-Dynamic

Bentley and J. B. Saxe. *Decomposable searching problems I: Static-to-dynamic transformations*. Journal of Algorithms, 1:301-358, 1980

Making the structure semi-dynamic

Need a semi-dynamic structure, so that we can

Add a new site to S in **amortized** time $O(\log^2 n)$

Find the nearest site to a query point q, in time $O(\log^2 n)$

We use Voronoi Diagram only for a demonstration – applies for many data structures.

Idea: We decompose S into a disjoint collection of sets

$S=\{ S_0 \cup S_1 \cup .. \cup S_k \}$, (some might be empty) where $|S_i| = 2^i$

So – at most one set of size 1, at most one of size 2, one of size 4, and so on. So $k=O(\log n)$.

We construct $VD(S_i)$, $\forall i$

Performing a query

Given $VD(S_0)$, $VD(S_1)$… $VD(S_k)$, to find the neatest site to a query point q, we just perform a query in each $VD(S_i)$ and find the nearest.

Time: $O(\log n)$ per VD, altogether $O(\log^2 n)$.

Handling insertions of sites: Given $S=\{S_0 \cup S_1 \cup S_k\}$, add new site s'

Rule: During the insertion process, we can temporally have 2 sets both containing 2^i sites, but then they gave to be merged.

Algorithm:

Create a new set $S'_0 = \{s'\}$.

While (there are two sets S_i , S'_i both containing 2^i sites) {

Merge S_i , S'_i to form a new set S'_{i+1} containing 2^{i+1} sites ;

Discard S_i , S'_i ;

}

Compute new *VD* for all new sets.

When done ,we have at most one set of size 2^i $\forall i$

Running time analysis

Recall that constructing new VD of m sites (where $m \leq n$) costs

$O(m \log m) \leq O(m \log n)$

When inserting a new site, we equipped it with $O(\log^2 n)$ dollars.

The constant is the same constant as the constant in the O( ) of the VD construction time.

Every time that a new VD of m sites is constructed, its sites pays for the construction. We collect $O(m \log m)$ dollars, so the construction leaves us with a positive balanced.

Every cite is involved in $\leq \log_2 n$ different VD's, so it is charged no more than $O(\log^2 n)$ dollars , as required.

Thm: Starting with an empty set, an sequence of n insertions takes time $O(n \log^2 n)$.

Running time analysis

Recall that constructing new VD of m sites (where $m \leq n$) costs

$O(m \log m) \leq O(m \log n)$

When inserting a new site, we equipped it with $O(\log^2 n)$ dollars.

The constant is the same constant as the constant in the O( ) of the VD construction time.

Every time that a new VD of m sites is constructed, its sites pays for the construction. We collect $O(m \log m)$ dollars, so the construction leaves us with a positive balanced.

Every cite is involved in $\leq \log_2 n$ different VD's, so it is charged no more than $O(\log^2 n)$ dollars , as required.

Application 2: Dynamic tables

Goal: Maintain insertions into an array, s.t. the array is small (with respect to the input)

Applications: Hashing tables, when the number of keys is not known in advanced.

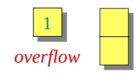
Problem: What if we don't know the number of insertions in advance?

Solution: ***Dynamic tables.***

IDEA: Whenever the table overflows, "grow" it by allocating (via `malloc` or `new`) a new, larger table. Move all items from the old table into the new one, and free the storage for the old table.

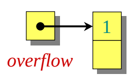
Example of a dynamic table

1. INSERT
2. INSERT



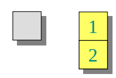
Example of a dynamic table

1. INSERT
2. INSERT



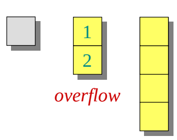
Example of a dynamic table

1. INSERT
2. INSERT



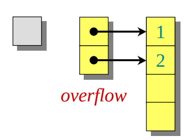
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



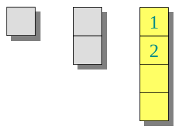
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



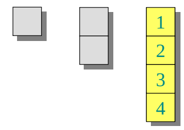
Example of a semi-dynamic table

1. INSERT
2. INSERT
3. INSERT



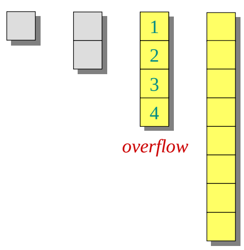
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT



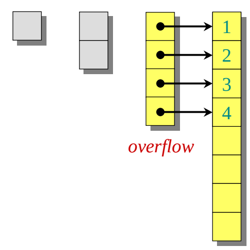
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



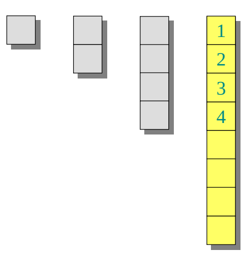
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



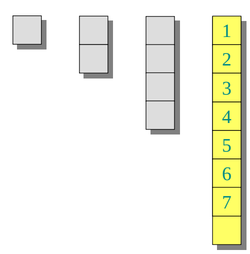
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



Worst-case analysis

Consider a sequence of n insertions. The worst-case time to execute one insertion is $\Theta(n)$. Therefore, the worst-case time for n insertions is $n \cdot \Theta(n) = \Theta(n^2)$.

WRONG! In fact, the worst-case cost for n insertions is only $\Theta(n) \neq \Theta(n^2)$.

Let's see why.

Tighter analysis

Let c_i = the cost of the ith insertion

$$= \begin{cases} i & \text{if } i-1 \text{ is an exact power of } 2, \\ 1 & \text{otherwise.} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1

Tighter analysis

Let c_i = the cost of the ith insertion

$$= \begin{cases} i & \text{if } i-1 \text{ is an exact power of } 2, \\ 1 & \text{otherwise.} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	1	1	1	1	1	1	1	1	1
		1	2		4				8	

Tighter analysis (continued)

$$\text{Cost of } n \text{ insertions} = \sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j \le 3n = \Theta(n).$$

Thus, the amortized cost of each dynamic-table insertion is $\Theta(n)/n = \Theta(1)$.

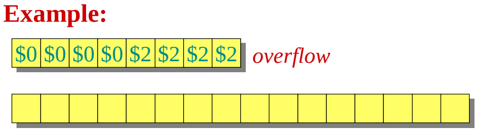
Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the ith insertion.

- \$1 pays for the immediate insertion.
- \$2 is stored for later table doubling.

When the table doubles, \$1 pays to move a recent item, and \$1 pays to move an old item.

Example:



overflow

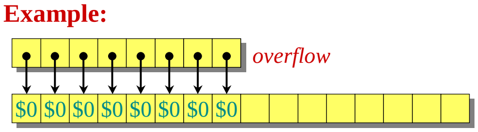
Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the ith insertion.

- \$1 pays for the immediate insertion.
- \$2 is stored for later table doubling.

When the table doubles, \$1 pays to move a recent item, and \$1 pays to move an old item.

Example:



overflow

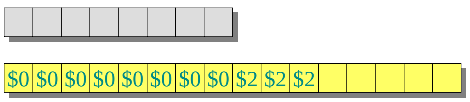
Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the i th insertion.

- $1 pays for the immediate insertion.
- $2 is stored for later table doubling.

When the table doubles, $1 pays to move a recent item, and $1 pays to move an old item.

Example:



Accounting analysis (continued)

Key invariant: Bank balance never drops below 0. Thus, the sum of the amortized costs provides an upper bound on the sum of the true costs.

What about deletions ?

If m is the size of the table containing n elements

Doubling: Copy all elements the size into a table of size $2m$, if $n>m$.

Shrinking: Copy all elements the size into a table of size $m/2$ if $n < m/4$..

Then still the amortized time per operation is $O(1)$

(homework)