History
- C'99: \(O(\log^{1+\varepsilon} n)\)
- Brodal-Jacob'00: \(O(\log n \log \log n)\)
- Brodal-Jacob'02: \(O(\log n)\)

Potential area: \(O(\varepsilon)\)

100 pages!

General Dynamicization Techniques

Def: A query problem is decomposable if the answer for input set \(S_1 \cup S_2\) can be computed from the answer for \(S_1\) & answer for \(S_2\) in \(O(1)\) time.

Ex: Range counting, range emptiness, nearest neighbor search, extreme pt in \(\mathcal{C}(t)\)

Technique 1: The "Logarithmic Method" (Bentley-Saxe '80)

If there is a static DS \(\mathcal{S}\) with \(P(n)\) preprocess time, \(S(n)\) space, \(Q(n)\) query time & problem is decomposable, then there is a semi-dynamic DS \(\mathcal{S}'\) with \(O(S(n))\) space, \(O\left(\frac{P(n) \log n}{n}\right)\) insert time amortized,

\(O(Q(n) \log n)\) query time

(i.e. total time for \(n\) inserts is \(O(P(n) \log n)\))

Ex: Triangle range counting in 2D

\(P(n) = O(n \log n)\), \(S(n) = O(n)\), \(Q(n) = O(n^{1/2+\varepsilon})\)

\(\Rightarrow\) \(O(n)\) space, \(O(\log^2 n)\) insert (amort.), \(O(n^{1/2+\varepsilon} \log n)\) query time

\(O(\log n)\) delete
CH extreme pt query in 2D

\[ P(n) = O(n \log n), \quad S(n) = O(n), \quad Q(n) = O(\log n) \]

\( \Rightarrow \ O(n) \) space, \ insert \( O(\log^2 n) \) \ query \( O(\log n) \)

\( \text{worse than Preparata '80!} \quad O(\log n) \)

(delete \( O(\log n) \))

nearest neighbor search in 2D

\[ P(n) = O(n \log n), \quad S(n) = O(n), \quad Q(n) = O(\log n) \]

(Voronoi diagram)

\( \Rightarrow \ O(n) \) space, \ insert \( O(\log^2 n) \), \ query \( O(\log n) \)

Proof: divide input set \( S \) into \( O(\log n) \) subsets

Invaraint: for each \( i = 0, \ldots, \log n \),

at most 1 subset at level \( i \) of size \( 2^i \)

\( \text{insert}(p) : \)

create \( \{p\} \) at level 0.

whenever \( \exists \) 2 subsets \( S_1, S_2 \) at same level \( i \),

destroy \( S_1, S_2 \),

build \( S \cup S_2 \) at level \( i+1 \)

\( \text{e.g.} \)

\[ \begin{array}{cc}
8 & \text{\( n = 11 \)} \\
\end{array} \]

\[ \begin{array}{cc}
2 & 1 \\
\end{array} \]

\( \text{\( \uparrow \) insert} \)

\[ \begin{array}{cc}
2 & 2 \\
\end{array} \]

\[ \text{\( n = 12 \)} \]

\[ \begin{array}{cc}
8 & 4 \\
\end{array} \]
(Similar to binary counter)

Viewed as a tree, over time:

level 3
level 2
level 1
level 0

(similar to merge sort)

Space \[ \sum_{i=0}^{\log n} S(2^i) = \mathcal{O}(S(n)) \] [assume \( S(n) \) is nondecreasing]

query \[ \leq \sum_{i=0}^{\log n} \mathcal{O}(2^i) \] by decomposability

\[ \leq \mathcal{O}(\mathcal{O}(n) \log n) \]

Total time for \( n \) inserts

\[ \leq \sum_{i=0}^{\log n} P(2^i) \frac{n}{2^i} \] (# builds at level \( i \) is \( \frac{n}{2^i} \))

\[ \leq \sum_{i=0}^{\log n} P(n) \frac{n}{n} \] [assume \( P(n) \) is nondecreasing]

\[ = \mathcal{O}(P(n) \log n) \]

Remarks: (i) amortized \( \rightarrow \) worst case

(idea - spread rebuild over multiple updates messy!)

(ii) Insert time improves to \( \mathcal{O}(\frac{M(n)}{n} \log n) \)

where \( M(n) = \) merging time
(iii) tradeoffs: base-b version...

(iv) if & supports delete in \(D(n)\) time,
   & also supports delete in \(O(D(n))\) time
   (if a drop by const factor, rebuild entire DS...)

Technique 2: The "Square Root Method" (also Bentley-Saxe)
Under same assumption,
there is a fully dynamic DS with
\(O(S(n))\) space, \(O(P(V(n))\) update time,
\(O(V(n)O(V(n))\) query time.

Proof: divide \(S\) into \(V(n)\) subsets of size \(V(n)
insert(p):
   let \(S_i = \text{smallest subset}\)
   destroy \(S_i\)
   build \(S_i - \{p\}\)
delete(p):
   let \(S_i = \text{subset containing} p\)
   destroy \(S_i\)
   build \(S_i - \{p\}\) 

Ex: nearest neighbor in 2D
   update \(O(V(n) \log n)\)
   query \(O(V(n) \log n)\) \([C'19: O(\log^4 n)\) update\]
   \(O(\log n)\) query

Technique 3:
Under same assumption,
there is an offline dynamic DS &'
entire update sequence is given in advance
\(O(S(n) \log n)\) space, \(O(\frac{P(n)}{n} \log n)\) amort. update time,
\(O(\frac{Q(n)}{\log n})\) query time.
Proof: for each element, consider its lifespan/time interval

build segment tree for these n intervals!

for each node, store all its long segs in a DS &

Space $O(S(n) \log n)$, total update time $O(P(n) \log n)$,
query $O(\Theta(n) \log n)$

Ranks - generalizes to semi-online
(insert is online; when element is inserted, told its delete time)

- don't need decomposability
  (if $S$ supports insert in worst-case $I(n)$ time;
  then $S$ has $O(I(n) \log n)$ amortized update time
  $O(\Theta(n))$ query time)

undo
inserts
v
new inserts

/09