Approximate Nearest Neighbor Search under ℓ_∞

Weston Jackson

April 2019

Abstract

This paper covers a variety of efficient static and dynamic data structures for c-approximate nearest neighbor (c-ANN) search under ℓ_∞. We begin by describing Indyk's recursive data structure for $O(\log_{1+\rho} \log d)$-ANN search. We then cover dynamization of said data structure via Bentley-Saxe, Overmars, and Indyk's recursive method. In the subsequent sections, we provide several alternative dynamic constructions for ℓ_∞ c-ANN search when $c \geq 3$, which can improve time or space complexity in low dimensional spaces. We show these constructions can be implemented with BSTs, Van Emde Boas trees, fusion trees, or hash tables. We then prove that these bounds can be improved on expectation using randomization. Finally, we cover the hardness of ℓ_∞ c-ANN search for $c < 3$ via reducibility to the superset query problem and orthogonal range search.

Contents

1 Introduction

Similarity search is a common problem when dealing with large data sets. Given some input object, we want our database to return the closest image, movie, or DNA sequence to our input. This problem is known as the *nearest neighbor search* problem, and it is an important optimization problem in a variety of fields.

Definition 1 (NNS)**.** *The nearest neighbor search (NNS) problem asks us, given a set of database points $x_1 \cdots x_n$ in some metric space (X, d), construct a data structure such that on a query point y, the data structure returns*

$$argmin_{x_1 \cdots x_n} d(y, x)$$

In one-dimensional space, the NNS problem has linear space and sublinear query time solutions. A binary search tree for example supports nearest neighbor search in $O(\log n)$ time and linear space. In higher dimensions however, obtaining linear space and sublinear query time is more difficult. To move past the *curse of dimensionality* barrier, we turn to approximation algorithms, which allow us to return a point that is at most c distance away from the near points. This approximate nearest neighbor problem we refer to as the c-ANN problem, and it allows for more efficient algorithms.

Definition 2 (c-ANN)**.** *The c-approximate near neighbor (c-ANN) problem asks us, given a set of points $x_1 \cdots x_n$ in some metric space (X, d), and parameter c, construct a data structure such that, given any query point y:*

- $\exists x : d(x, y) \leq 1$ *return some x where $d(x, y) \leq c$.*
- $\forall x : d(x, y) \geq c$ *return nothing.*

Optimal data structures for c-ANN problems depend on the metric space in which the algorithm is performed. The most common spaces for nearest neighbor search are often the ℓ_p spaces in $\mathbb{R}^d$. In ℓ_p space, distance between two points x and y is defined as $\|x - y\|_p = (\sum_{i=1}^{d} |x_i - y_i|^p)^{\frac{1}{p}}$. While the properties of ℓ_1 (Manhattan Distance) and ℓ_2 (Euclidean Distance) are well known, in this paper we will consider the less widely understood ℓ_∞ norm, which returns the max difference between the coordinates of x and y:

$$\|x - y\|_\infty = \max_i |x_i - y_i|$$

The ℓ_∞ norm is of interest for several reasons. First, it is a natural metric in the case where coordinates correspond with unrelated data. Additionally, the ℓ_∞ norm is also noteworthy in that it is a useful host space for subspace embeddings [1].

2 Indyk's Data Structure for ℓ_∞ ANN

2.1 Overview

In [2], Piotr Indyk describes a data structure for $O(\log_{1+\rho} \log d)$-ANN search under ℓ_∞, which supports sublinear query time with slightly super-linear space. To obtain these bounds, Indyk's data structure exploits the fact that the ℓ_∞ norm is *decomposable*. In particular, if two vectors x and y satisfy $\forall i : |x_i - y_i| \leq c$, this implies x is a c-ANN of y. Additionally if a single $|x_i - y_i| > 1$, this implies $\|x - y\|_\infty > 1$. Indeed, this property will also play a crucial role in making his data structure dynamic.

Indyk uses a divide-and-conquer strategy to split the ℓ_∞ c-ANN problem into subproblems. This gives his data structure a tree-like structure, where each node in the recursive tree is either a *separator* or *box* node. Intuitively, box nodes group together sets of points within a small diameter into a single node while separator nodes utilize decomposability to divide the nearest neighbor search into smaller subproblems.

2.2 Construction

Let $x_1 \cdots x_n$ be our data set. Assume we know that our query point y lies in some point set $P \subset \mathbb{R}^d$. To reduce our search problem into subproblems, we pick a coordinate i and value t, then divide P into two approximate near

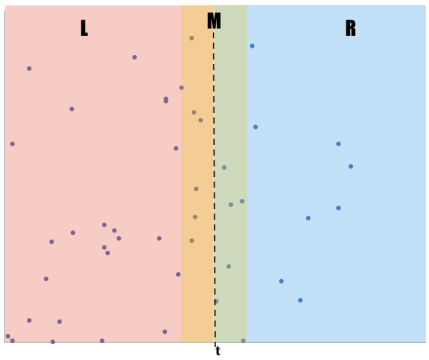

Figure 1: An example separator hyperplane. All the points in M are copied into both left and right subproblems so that we don't discard any potential near neighbors.

neighbor subproblems via t. The left subproblem handles P where the ith coordinate ranges from $(-\infty, t)$ and the right subproblem handles P where the ith coordinate ranges from $[t, \infty)$. Assuming our query point is y, we can choose which subproblem to recurse into depending on if $y_i \in (-\infty, t)$ or $y_i \in [t, \infty)$.

Unfortunately, this strategy has one major issue. There could be a near neighbor x such that $x_i = t + \epsilon$ and $y_i = t - \epsilon$, meaning $x_i \in [t, \infty)$ yet $y_i \in (-\infty, t)$. This is a problem, as x is potentially a near neighbor of y, but it is not included in the $(-\infty, t)$ subproblem. To solve this, Indyk proposes we include the points $x_i \in [t-1, t+1]$ in *both* subproblems. Thus, whenever we recurse, we are ensured that any x_i that satisfies $|y_i - x_i| \leq 1$ will not be discarded. Let

$$L := (-\infty, t-1)_i \cap P$$
$$R := (t+1, \infty)_i \cap P$$
$$M := [t-1, t+1]_i \cap P$$

Given a point set P, we define our subproblems $P_1 = L \cup M$ and $P_2 = R \cup M$. Since the choice of t defines a hyperplane separator in $\in \mathbb{R}^{d-1}$, we refer to the choice of t as a *separator* node. See Figure 1 for a visual example.

To prove that this recursive solution yields sufficient time and space bounds, we need to ensure that we reduce the search space by a substantial fraction at each iteration. Let $|L|$ be the number of points in the point set satisfying $x \in L$, and define $|R|$ and $|M|$ similarly. Our goal is to choose t and i such that we avoid the case where either $|R| \ll |M|$ or $|L| \ll |M|$. If such a separator exists, we recurse on P_1 or P_2 as specified before. However, if such a separator does not exist, then Indyk proves that a large number of points can be contained in a box $C \subset P$ which has small diameter.

Lemma 3. *Let $m = \frac{|M|}{|P|}$, $r = \frac{|R|}{|P|}$, and $l = \frac{|L|}{|P|}$ such that $l + r + m = 1$. Define*

$$L(m, r) = \log_{1/(m+r)}(\frac{1}{m} - 1)$$

For any set P and $\rho > 0$, either

- *(1) We can choose t and i such that $L(m, r) \leq \rho$ and $l, r > \frac{1}{4d}$*
- *(2) There is a $C \subset P$ of size $\frac{|P|}{2}$ having diameter $\leq 4\lceil \log_{1+\rho} \log 4d \rceil$*

Proof. Intuitively, the first condition says there exists a good separator such that $L(m, r) \leq \rho$ and $\frac{1}{4d}$ points in L and R are not replicated. Indyk proves if this condition cannot be satisfied, then half the points can be contained in a box with diameter $O(\log_{1+\rho} \log d)$. The proof is moved to Appendix A for conciseness, but the main idea is that the lack of a good hyperplane separator for some coordinate i implies that at least $1 - \frac{1}{2d}$ fraction of points have their ith index in an interval $[-\lceil \log_{1+\rho} \log 4d \rceil, \lceil \log_{1+\rho} \log 4d \rceil - 1]$. Indyk then removes all the points NOT

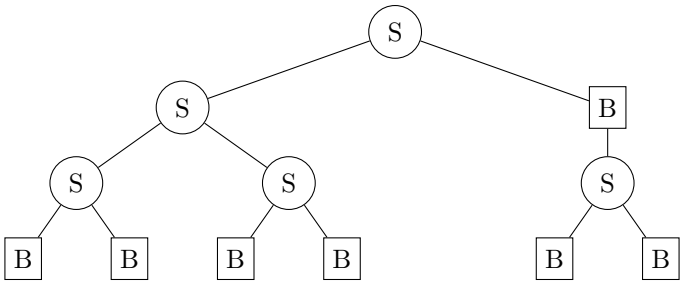

Figure 2: Tree structure of Indyk's data structure for $O(\log_{1+\rho} \log d)$-ANN under ℓ_∞. S designates a separator node while B designates a box node.

in this interval from P. Repeating this process over all d coordinates, this implies we have at least $1 - \frac{1}{2d}d = \frac{1}{2}$ fraction of points remaining. Thus, we have $\frac{|P|}{2}$ points remaining in a box with diameter:

$$|[-\lceil \log_{1+\rho} \log 4d \rceil, \lceil \log_{1+\rho} \log 4d \rceil - 1]| = O(\log_{1+\rho} \log d)$$

□

2.3 Complexity

We build the tree data structure by looking for good separators with $L(m, r) \leq \rho$ and $l, r > \frac{1}{4d}$. If such a separator exists, we store its metadata in a separator node and recursively continue on $P_1 = L \cup M$ and $P_2 = R \cup M$. If a good separator does not exist, the previous lemma implies P contains a set C of $\frac{|P|}{2}$ points with $O(\log_{1+\rho} \log d)$ diameter. We create a box node for C, store the center of the box and a random representative point from C as metadata, and recurse on $P - C$. See Figure 2 for a visual example.

Lemma 4. *Indyk's data structure has query time $O(d \log n)$.*

The query procedure is given in Algorithm 1. On separator nodes, we recurse on either subproblem P_1 or P_2 by comparing $y_i < t$ or $y_i > t$. On box nodes, we check if the ℓ_∞ distance between y and the box is ≤ 1. If this is true, we can return the representative point in C, as all points in C are within $O(\log_{1+\rho} \log d) + 1 = O(\log_{1+\rho} \log d)$ of y under ℓ_∞ distance. If y is not within 1 of C under ℓ_∞, we continue on $P - C$. There are two cases:

- The number of box nodes visited is at most $O(\log n)$ as the size of the point set is divided in half at each iteration. Calculating the distance between the box and the query point takes $O(d)$ time. Thus, the time complexity for recursing on the box nodes is at most $O(d \log n)$.
- The number of separator nodes visisted is at most $O(d \log n)$, since each child contains at most $(1 - \frac{1}{4d})$ fraction of parent points. The time spent in each separator is $O(1)$ (single coordinate comparison), thus the total cost is $O(d \log n)$.

Lemma 5. *Indyk's data structure uses space $O(dn^{1+\rho})$.*

To bound the number of nodes in the data structure, we add a second child to each of the box nodes consisting of a balanced binary tree with leaves equal to the number of points in C. Let this modified tree be T', and note that it bounds the size of the original data structure but still has depth at most $O(d \log n + \log n) = O(d \log n)$. Let $N(n)$ be the maximum number of leaf nodes in T'_P where $|P| = n$. Indyk shows $N(n) \leq n^{1+L}$, where $L = L(m, r) \leq \rho$. Assume by induction we have $N(k) \leq k^{1+L}$ for $k < n$. For box nodes the proof is trivial as no points are replicated, so we only consider separators:

$$N(n) \leq N((l + m)n) + N((m + r)n)$$

By induction

$$\begin{aligned}
N(n) &\le ((r+m)n)^{1+L} + ((m+l)n)^{1+L} \\
&\le n^{1+L}((r+m)(r+m)^{L} + (1-r)) && l+m+r=1 \\
&= n^{1+L}((r+m)(r+m)^{\log_{1/(r+m)} \frac{r+m}{r}} + (1-r)) && \log_{1/(r+m)} \frac{r+m}{r} = L \\
&= n^{1+L}((r+m)\frac{r}{r+m} + (1-r)) \\
&= n^{1+L}
\end{aligned}$$

Lemma 6. *Indyk's data structure has preprocessing $O(d^2 n^{1+\rho} \log n)$.*

Again consider the modified tree T'. Deciding whether a set of $|P|$ points should be a separator or a box node takes time $O(|P|d)$. The size of $|P|$ can be bounded by the number of leaf nodes under T'_P. Via a charging argument, each leaf can be "charged" for at most $O(d \log n)$ internal nodes over the entire tree T'. Thus, since the total number of nodes is $O(n^{1+\rho})$, then the total preprocessing time is $O(d^2 n^{1+\rho} \log n)$.

Theorem 7 (Indyk's ℓ_∞ Data Structure)**.** *For any $\rho > 0$, there is a data structure for $O(\log_{1+\rho} \log d)$-ANN search using $O(dn^{1+\rho})$ space and supporting $O(d \log n)$ query time and $O(d^2 n^{1+\rho} \log n)$ preprocessing time.*

Algorithm 1 Query for Indyk's ℓ_∞ Data Structure

```
1:  function QUERY(y, T)
2:      if T.P = ∅ then
3:          return ∅
4:      end if
5:      if T.root is SEPARATOR then
6:          if y_i < T.t then                    ▷ i is the separator index, t is the value
7:              return QUERY(y, T.P_1)
8:          else
9:              return QUERY(y, T.P_2)
10:         end if
11:     end if
12:     if T.root is BOX then
13:         if ‖y − T.BOX‖_∞ ≤ 1 then            ▷ O(d) (use center of box)
14:             return T.x                       ▷ return some x ∈ C
15:         else
16:             return QUERY(y, T.P \ C)         ▷ recurse on points not in box
17:         end if
18:     end if
19: end function
```

3 Dynamization

3.1 Insertions via Bentley-Saxe

Because the c-ANN problem under ℓ_∞ is decomposable, we note that it is possible to dynamize Indyk's data structure in a black-box manner. One way to do so for insertions is via the Bentley-Saxe method [3]. The Bentley-Saxe method partitions the set of points into blocks $B_0 \cdots B_{\log n - 1}$ of exponentially increasing size (each block B_i has capacity 2^i). To insert some point x, we do the following:

- Locate the smallest $B_i = \emptyset$

- Insert $B_0 \cdots B_{i-1}$ and x into B_i and preprocess
- Set $B_0 \cdots B_{i-1} = \emptyset$

Let $I(n)$ be the insertion time, $Q(n)$ be the query time, and $P(n)$ be the preprocessing time. The Bentley-Saxe method yields the following theorem:

Theorem 8 (Bentley-Saxe)**.** *Given a (static) data structure S for decomposable searching where $Q_S(n) = o(n^{1+\epsilon})$ and $P_S(n) = \Omega(n^{1+\epsilon})$, there exists a data structure S' such that:*

- $Q_{S'}(n) = O(\log n) Q_S(n)$
- $I_{S'}(n) = O(P_S(n)/n)$

By applying Bentley-Saxe to Indyk's data structure we note that we can dynamize the ℓ_∞ data structure for insertions with $O(\log n)$ query time overhead.

Observation 9. *There is a data structure dynamized under insertions for $O(\log_{1+\rho} \log d)$-ANN under ℓ_∞ using space $O(dn^{1+\rho})$ with query time $O(d \log^2 n)$ and amortized insertion time $O(d^2 n^\rho \log n)$.*

3.2 Naive Deletions via Overmars

Overmars's dynamization technique achieves a better worst-case bound on insertion time and can support deletions in the dynamic case, as long as the static data structure supports *weak deletions* in $D(n)$ time. A data structure supports weak deletions if it can either remove or mark points as deleted without changing the run time of update and query operations [4].

Overmars maintains bags $B_0, B_1, \cdots$ where each bag B_i stores three blocks S_i^1, S_i^2, S_i^3 of size 2^i that are in use. In addition, there is another block S_i^c that is under construction. The idea is that as soon as two S_i blocks are full in B_i we start building S_{i+1}^c with these $2^i + 2^i = 2^{i+1}$ elements. The work is spread out over 2^{i+1} insertions, each time doing $P_S(2^{i+1})/2^{i+1}$ units of construction work. A third S_i may arrive into B_i, but by the time the fourth is ready, S_{i+1}^c will be finished and sent to B_{i+1}. In this way, Overmars achieves the same bounds as Bentley-Saxe, but achieves $O(P_S(n)/n)$ insertion in the worst-case as rebuilding is spread out over a long period of inserts.

To enable deletions, Overmars adds a dictionary to his data structure which allows us to locate the blocks S_i for each point. When we want to delete a point, we locate the block for the point using the dictionary and delete the point. This takes at most $D(n)$ time. Deleting points from blocks under construction requires slightly more nuance, but the main idea is that we can buffer deletions into an array BUF_i so that they can be applied to the block under construction S_i^c once construction is finished.

Theorem 10 (Overmars)**.** *Given a (static) data structure S supporting weak deletions for decomposable searching where $Q_S(n) = o(n^{1+\epsilon})$ and $P_S(n) = \Omega(n^{1+\epsilon})$, there exists a data structure S' such that:*

- $Q_{S'}(n) = O(\log n) Q_S(n)$
- $I_{S'}(n) = O(P_S(n)/n)$
- $D_{S'}(n) = O(\log n + D_S(n) + P_S(n)/n)$

Dynamizing Indyk's data structure for insertions and deletions is non-trivial. This is because points in Indyk's data structure can be replicated in many different places in the recursive tree. To prevent a deleted point x from being returned in the future, we must delete x from potentially many box nodes. Nonetheless, to enable deletions in this case, we can have each box node keep a hash table of all the points contained in its box set C at no extra space cost. This can be done because the space cost of Indyk's data structure is calculated assuming the box nodes use space linear in the size of C (via the modified tree T'). Thus, since hash tables also use linear space, we can have each box node keep a hash table of the points in C while still using $O(dn^{1+\rho})$ space.

Our deletion strategy is to perform a find in the recursive tree for x and delete x from the box node's with hash tables containing x. The number of nodes we visit in the find procedure is equal to the number of nodes containing x in their point set. In the worst case, x can be replicated $O(n^{1+\rho})$ many times. However, while we may have to update $O(n^{1+\rho})$ box nodes, the cost of the weak deletion is $O(1)$ in each box node. Finally, we can dynamize for insertions using Overmars, which yields the following observation.

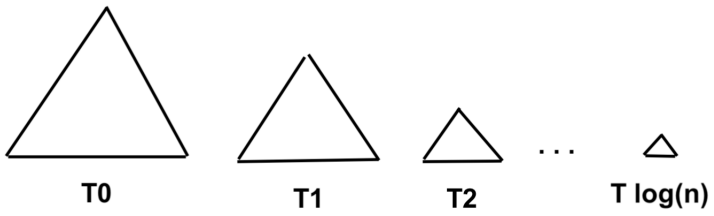

Figure 3: Indyk dynamizes his static data structure by creating $\log n$ copies of the static data structure via a recursive procedure. All the points in a given T_i with multiplicity greater than 2x the average are removed to create T_{i+1}.

Observation 11. *There is a fully dynamic data structure for $O(\log_{1+\rho} \log d)$-ANN under ℓ_∞ using space $O(dn^{1+\rho})$ with $O(d \log^2 n)$ query time, $O(d^2 n^\rho \log n)$ insertion time, and $O(n^{1+\rho})$ deletion time.*

3.3 Indyk's Dynamization

Indyk describes a different approach for dynamizing his data structure which results in improved deletion time at the cost of an extra $\log n$ factor in query time [5]. To key idea is to bound the number of times any point in the tree is replicated. To do so, Indyk defines the *multiplicity* $m(x)$ of a point x as the number of nodes in the tree containing x in their point set. The average multiplicity for any point is $\frac{n^{1+\rho}}{n} = n^\rho$.

Indyk's dynamization algorithm first builds the tree T_0 and computes the multiplicity of each point. We then remove all the bad points with multiplicity more than $2n^\rho$ (twice the average). By Markov, only $\frac{1}{2}$ fraction of the points have multiplicity greater than $2n^\rho$. Thus, we remove at most half the points from T_0 and then build a new tree T_1 with at most $\frac{n}{2}$ points. After building T_1, we remove all the points with multiplicity more than $2(\frac{n}{2})^\rho$ in T_1 to build T_2 (at most $\frac{n}{4}$ points remain). We continue recursively until we have trees $T_1 \cdots T_{\log n}$. Each tree T_i has at most $\frac{n}{2^i}$ points, and the multiplicity of any point $x \in T_i$ is at most

$$m(x) \leq 2(\frac{n}{2^i})^\rho \leq O(n^\rho)$$

Indyk shows the improved construction procedure takes time at most $O(P(n)+dn)$. Search is just a composition of a search in $\log n$ trees $T_1 \cdots T_{\log n}$, which takes at most $O(d \log^2 n)$ time. To delete a point x, we find the tree T_i containing x, then perform a find within T_i to delete x from at most $O(n^\rho)$ nodes. Indyk claims the total deletion time is $O(dn^\rho)$.

As an aside, we note that this deletion procedure does not make use of $O(1)$ deletions within each box node. Indyk bounds deletion time as an $O(d)$ operation in $O(n^\rho)$ nodes. However, by implementing the box nodes with hash maps, we should be able to delete points within a box in $O(1)$ time. Thus, it may be possible to perform deletions in only $O(n^\rho)$ time.

Finally, after describing the deletion procedure, Indyk claims his dynamic data structure has the following properties when dynamizing for insertions via Overmars.

Claim 12 (Indyk's Dynamization)**.** *There is a fully dynamic data structure for $O(\log_{1+\rho} \log d)$-ANN under ℓ_∞ using space $O(dn^{1+\rho})$ supporting $O(d^2 \log^3 n)$ query time, $O(d^2 n^\rho \log n)$ insertion time, and $O(dn^\rho)$ deletion time*[1].

[1] I think this claim from Indyk's PhD thesis may have an error. First, I think deletion time should be at least $O(d^2 n^\rho \log n)$ after applying Overmars, as deletion time under Overmars always is at least the insertion time. Additionally, I think the query time may only be $O(d \log^3 n)$ as the static query time is $O(d \log n)$, then we only need one $O(\log n)$ factor to compose the queries and one $O(\log n)$ factor from Overmars. I have not been able to confirm that these are errors, as this claim is not cited or referenced anywhere else on the internet.

4 ℓ_∞ ANN for $d = o(\log n)$

4.1 Intuition

We propose several alternative constructions for c-ANN search under ℓ_∞ that provide strong time and space bounds for low-dimensions. The first proposal gives efficient space, insertion, and deletion procedures at the cost of exponential in d query time. The second proposal gives sublinear query time, but has exponential in d space, insertion, and deletion time. Both strategies heavily leverage decomposability of ℓ_∞ and rely on creating buckets of size $\approx c$ over a universe U of possible values per index. We first show how to implement these solutions using binary search trees (assume with rotations), then extend the idea to other data structures in a black-box manner. Finally, we also show how to reduce the exponential time complexity blow-up via randomization.

Since both proposals are exponential in d for some operations, these bounds are trivial if $d = \Omega(\log n)$. Thus, it is best to use these strategies if d is a large constant or $d = o(\log n)$. Additionally, as we assume points are drawn from U^d, we note that these strategies are generally more effective for large universes where $n \ll U$.

4.2 Linear Space Model

We exploit the decomposability of c-ANN search under ℓ_∞ by considering the data set coordinate by coordinate. We assume each point x exists in some U^d, such that U is the universe of possible values for each coordinates $i = 1 \cdots d$. The idea is to divide the universe into $\frac{U}{\Delta}$ non-overlapping buckets, where each bucket b has size Δ (chosen later). In particular, each bucket b_j covers a disjoint interval $[\Delta j, \Delta(j+1))$. Thus, for a given coordinate i, each point x maps to a bucket via $\lfloor \frac{x_i}{\Delta} \rfloor$.

Our static data structure is composed recursively over coordinates $i = 1 \cdots d$. Starting with coordinate 1, we create a binary search tree T_1 over the "present" buckets. A bucket b_j is present in T_1 if some point x in the data set satisfies:

$$x_1 \in [\Delta j, \Delta(j+1))$$

We assume a given bucket b_j is keyed by the identifier j in T_1. While it is possible that multiple data set points map to a given bucket, only one point is needed for b_j to be present in T_1. Additionally, we also maintain a linked list of the buckets in T_1 in sorted order, so that we can access adjacent buckets quickly.

Consider some bucket b_j which is a node in the binary search tree T_1. We have b_j store a new binary search tree $T_{2,j}$ on coordinate 2. $T_{2,j}$ is constructed recursively on all points that map to the bucket b_j, and is keyed on coordinate 2. For each leaf bucket present in $T_{2,j}$, we store another tree on coordinate 3 and so forth. We continue this recursive process for all $1 \cdots d$ coordinates. For any leaf at level d, the path from root to leaf goes through d binary trees and d different buckets. The points in the final leaf buckets we can store directly.

The important thing to note is that any points in the same leaf bucket on the ith tree must have been in the same leaf bucket in the previous $1 \cdots i-1$ trees. Thus, the bucket for the ith binary tree contains a set of points within Δ of each other on coordinates $1 \cdots i$. Thus, any points in the leaf bucket on the dth coordinate are within Δ of each other on coordinates $1 \cdots d$. Finally, this implies any two points x and y in the same bucket on the final coordinate satisfy:

$$\|x - y\|_\infty \leq \Delta$$

To perform an approximate nearest neighbor query for a point y, we find the nearest bucket to y_1 by searching for $\lfloor \frac{y_1}{\Delta} \rfloor$ in T_1. Let bucket b_j be the nearest bucket returned for y_1. We check buckets b_{j-1}, b_j and b_{j+1} to see if any bucket's interval is within 1 of y_1 (we only want to recurse on buckets that could have potential near neighbors). For each bucket b_j we can simply check:

$$y_i \in [j\Delta - 1, (j+1)\Delta + 1)$$

It is important to note that we need $\Delta \geq 2$, as that ensures y_1 can only be within 1 of at most two of the buckets. For whichever buckets satisfy this condition, we recurse in their trees on the next coordinate. We continue this process for coordinates $2 \cdots d$ and return any points present in the buckets on the dth level. Any returned point in this leaf bucket must be within $\Delta + 1$ of y for each coordinate $1 \cdots d$ (as any bucket we recuse into must be within 1 of y_i). Thus, if we set $\Delta = c - 1$, then any point returned is a c-approximate near neighbor to y under ℓ_∞.

Since no points are replicated, the space used is $O(nd)$. Performing a search operation in the first coordinate costs $O(\log n)$. At each level, we recurse in at most two buckets, and there are at most d levels. Thus, the query time is $O(2^d \log n)$. Insertions and deletions require us to insert in at most d binary search trees, for a total cost of $O(d \log n)$.

Theorem 13 (linear space dynamic ℓ_∞ c-ANN)**.** *There exists a linear space dynamic ℓ_∞ c-ANN data structure, for $c \geq 3$, with $O(2^d \log n)$ query time, and $O(d \log n)$ insertion and deletion time.*

Algorithm 2 Linear Space ℓ_∞ c-ANN Data Structure

```
1: function QUERY(y, i, T)
2:     j = FIND(T, ⌊y_i/Δ⌋)
3:     if y_i ∈ [(j − 1)Δ − 1, jΔ + 1) then                ▷ Check b_{j−1}
4:         if i = d then
5:             return b_{j−1}.rand()                      ▷ Return any point in the last bucket
6:         end if
7:         return QUERY(y, i + 1, b_{j−1}.T)
8:     end if
9:     if y_i ∈ [(jΔ − 1, (j + 1)Δ + 1) then               ▷ Check b_j
10:        if i = d then
11:            return b_j.rand()
12:        end if
13:        return QUERY(y, i + 1, b_{j−1}.T)
14:    end if
15:    if y_i ∈ [(j + 1)Δ − 1, (j + 2)Δ + 1) then          ▷ Check b_{j+1}
16:        if i = d then
17:            return b_{j+1}.rand()
18:        end if
19:        return QUERY(y, i + 1, b_{j+1}.T)
20:    end if
21: end function
```

4.3 Sublinear Query Time Model

Our second construction alters the linear space data structure using a strategy similar to Indyk's data structure. The main idea is that in order to improve query time, we need to ensure that we always recurse in one bucket in all $i = 1 \cdots d$ levels. To do so, we make the buckets overlapping, such that any data set points in the tree satisfying $|x_i - y_i| \leq 1$ are contained in the bucket for y_i. This is accomplished by modifying each bucket b_j to contain the points in the *overlapping* interval:

$$[j\Delta - 1, (j+1)\Delta + 1)$$

During the query procedure, on coordinate i, we simply find the nearest bucket b_j to y_i in the tree using $\lfloor \frac{y_i}{\Delta} \rfloor$ as before. We then recurse in b_j if $y_i \in [j\Delta, (j+1)\Delta)$. The result is that we only recurse in one bucket at each level, and the buckets we recurse into always retain any potential near neighbors of y. The trade-off is the increased space cost, as now that the buckets are overlapping, each point in the data set may be replicated in at most two buckets at any given level. After d levels of branching, the space cost

$$S(n) = (dn + 2dn + 4dn + \cdots 2^d dn)n = d2^d(n + \frac{n}{2} + \cdots) = O(dn2^d)$$

Thus, a single point from the database can be replicated up to $O(2^d)$ times. This can happen if points from the database are replicated into two buckets for each coordinate $i = 1 \cdots d$. Unfortunately, this also implies insertions and deletions will be more expensive, as database points may require inserting into or deleting from $O(2^d)$ different trees. The result is that both insertion and deletion algorithms will require $O(2^d \log n)$ time. The benefit is that query time is only $O(d \log n)$.

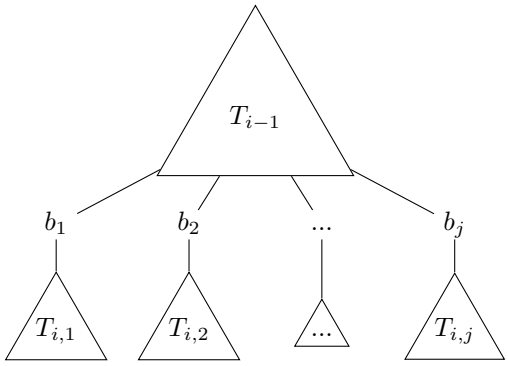

Figure 4: Structure of both linear space and sublinear query data structures for ℓ_∞ c-ANN. Assume the $i-1$th coordinate for point set P is stored in a data structure T_{i-1}. The leaves of T_{i-1} are buckets of size Δ, where bucket b_j maps to a data structure $T_{i,j}$ that is keyed on the ith coordinate for all $x \in b_j$.

Theorem 14 (sublinear query dynamic ℓ_∞ c-ANN)**.** *There exists a dynamic ℓ_∞ c-ANN data structure, for $c \geq 3$, using $O(dn2^d)$ space that maintains $O(d \log n)$ query time and $O(2^d \log n)$ insertion and deletion time.*

4.4 Alternative Models

While our example uses binary search trees, any linear space data structure that supports find, insertion, and deletion will suffice. Thus, we can substitute binary search trees for Van Emde Boas trees, fusion trees, and even hash tables. Additionally, note that we only need to maintain these data structures over the bucketed universe U of size $\frac{U}{\Delta} = \frac{U}{c-1}$ at each level. Substituting Van Emde Boas trees yields the following bounds:

Observation 15 (Van Emde Boas ℓ_∞ c-ANN)**.** *There exists a dynamic ℓ_∞ c-ANN data structure, for $c \geq 3$, using Van Emde Boas trees with the following properties*

- *$O(n)$ space, $O(2^d \log\log \frac{U}{c-1})$ query time, $O(d \log\log \frac{U}{c-1})$ insert and deletion time.*
- *$O(dn2^d)$ space, $O(d \log\log \frac{U}{c-1})$ query time, $O(2^d \log\log \frac{U}{c-1})$ insertion and deletion time.*

We can also substitute fusion trees in the case when the universe is large. In the static case, fusion trees can perform predecessor search in time $O(\frac{\log n}{\log\log U})$ while using linear space. Fusion trees can be dynamized using exponential trees, which yield $O(\log\log n + \frac{\log n}{\log\log U})$ insertion and deletion time [6].

Observation 16 (Fusion Trees ℓ_∞ c-ANN)**.** *There exists a dynamic ℓ_∞ c-ANN data structure, for $c \geq 3$, using fusion trees with the following properties*

- *$O(n)$ space, $O(2^d \frac{\log n}{\log\log U})$ query time, $O(d(\log\log n + \frac{\log n}{\log\log U}))$ insertion and deletion time.*
- *$O(dn2^d)$ space, $O(d\frac{\log n}{\log\log U})$ query time, $O(2^d(\log\log n + \frac{\log n}{\log\log U}))$ insertion and deletion time.*

We can also improve our average case time complexity by using hash tables instead of trees. The keys for our hash tables will be the $\frac{U}{c-1}$ bucket identifiers. In the linear space example, we can simply do two hash lookups to retrieve the at most two buckets within 1 of our query index y_i. We then recurse on each buckets hash table on the next coordinate using y_{i+1}. The procedure in the sublinear query time example is similar, and also achieves improved average time complexity bounds. The average cost of inserting, deleting, and finding is $O(1)$, meaning it is possible to get query and update time entirely in terms of d!

Observation 17 (Hash Tables ℓ_∞ c-ANN)**.** *There exists a dynamic ℓ_∞ c-ANN data structure, for $c \geq 3$, using hash tables with the following properties*

- *$O(dn)$ space, $O(2^d)$ average query time, $O(d)$ average insertion and deletion time.*
- *$O(dn2^d)$ space, $O(d)$ average query time, $O(2^d)$ average insertion and deletion time.*

4.5 Approximation Trade-Offs

Consider the case where we use the hash table data structure with sublinear query time (data set points are replicated). We show that it is possible to use randomization and the approximation factor c to improve space, insertion, and deletion time on expectation.

First, we note that constructing the recursive hash table data structure from scratch takes time

$$P(n) = O(n2^d)$$

We want to avoid the worst-case outcome of a point being replicated $O(2^d)$ times during construction. The key insight is that a data set point is only replicated into two buckets if the point is within 1 of an adjacent bucket. Intuitively, this should happen rarely if the bucket size $\Delta \gg 2$. To formalize this, we let $\alpha \in [0, \Delta)$ be the bucket offset. We define the interval for a bucket b_i as follows:

$$b_i = [j\Delta + \alpha, (j+1)\Delta + \alpha)$$

Effectively, the choice of α shifts all the buckets by a random offset. The idea is to choose α uniformly at random for each coordinate during construction. After randomly choosing the offset, we construct the data structure.

Claim 18. *The probability a data set point is replicated at any given level $\frac{2}{c-1}$ on expectation.*

Proof. Each bucket b_j has size $c-1$. If a point x is within 1 of an adjacent bucket, it is replicated. Note that x is either within 1 of b_{j-1} on the left or b_{j+1} on the right. Thus, within the bucket of size $c-1$, the bad region where x can be replicated has size 2. Thus, the probability x is replicated given each bucket is randomly shifted by α is $\frac{2}{c-1}$. □

Claim 19. *The expected number of points across the entire data structure is $O(n2^{\frac{d}{c-1}})$*

Proof. Let N_i be the number of points at level i. The the size of N_{i+1} is N_i plus the expected number of replicated points at level i:

$$N_{i+1} = N_i + \mathbb{E}[\#\text{ replicated points}]$$

Each point is replicated with probability $\frac{2}{c-1}$. Since there are N_i points at level i, the expected number of replicated points is $(\frac{2}{c-1})N_i$. Thus, on expectation:

$$N_{i+1} = N_i + (\frac{2}{c-1})N_i = N_i(1 + \frac{2}{c-1})$$

We know that $N_0 = n$ on the first level. On each subsequent level, the number of nodes increases by a factor of $(1 + \frac{2}{c-1})$. Thus, on level d, the expected number of nodes is

$$N_0(1 + \frac{2}{c-1})^d = n(1 + \frac{2}{c-1})^d = n2^{d\log(1+\frac{2}{c-1})} = O(n2^{\frac{d}{c-1}})$$

□

Thus, the expected space is $S(n) = O(dn2^{\frac{d}{c-1}})$ and expected preprocessing time is $P(n) = O(n2^{\frac{d}{c-1}})$. Our construction algorithm is to randomly choose α for each coordinate before inserting all the points in the data set. The total cost for construction is $O(n2^{\frac{d}{c-1}})$.

Theorem 20. *There exists a static ℓ_∞ c-ANN data structure that uses $O(dn2^{\frac{d}{c-1}})$ expected space and supports $O(d)$ query time and $O(n2^{\frac{d}{c-1}})$ expected construction time.*

For dynamization, we can use naive deletions and apply Overmars for insertions.

Theorem 21. *There exists a static ℓ_∞ c-ANN data structure that uses $O(dn2^{\frac{d}{c-1}})$ expected space and supports $O(d\log n)$ query time and $O(2^{\frac{d}{c-1}})$ insertion time and $O(2^{\frac{d}{c-1}} + \log n)$ deletion time.*

5 ℓ_∞ ANN for $c < 3$

5.1 Reduction to Superset Query Problem

ℓ_∞ c-ANN search appears to be a difficult problem for $c < 3$ approximations. In the linear space and sublinear query time dynamizations in the previous section, $c \geq 3$ is necessary as otherwise we may have to recurse in more than two buckets to find potential near neighbors. As c shrinks, the buckets become smaller and an individual point becomes within 1 of an increasing number of buckets. We note that $\forall c$, the worst-case replication factor becomes:

$$(1 + \lceil \frac{2}{c-1} \rceil)^d$$

Thus, as c approaches $1 + \epsilon$, the replication factor of our proposed data structures approaches $(\frac{1}{\epsilon})^d$, which is not good. Furthermore, Indyk's static data structure faces a similar problem in that its size also blows up as c decreases. The best approximation factor c for which his data structure can maintain slightly super-linear space is

$$4\lceil \log_{1+\rho} \log 4d \rceil + 1 \geq 5$$

By altering the data structure, Indyk can achieve a static 3-approximation at the cost of $n^{O(\log d)}$ space. The idea is to enhance the search procedure in the box nodes, which have diameter at least 4 and thus do not necessarily satisfy the approximation. If the query point y is in the center of the box node, this is fine as it is near all the other points. However if certain coordinates y_i are not near the center of the box, then we perform a separate search procedure for points in the box node with x_i near y_i. Indyk notes we can build d separate data structures for each coordinate to help with this search, without surpassing an $n^{O(\log d)}$ space bound. The query time remains $O(d \log n)$.

Unfortunately, this strategy stops working for approximations $c < 3$, as it is intrinsically difficult to maintain subexponential space and sublinear query time at this level of approximation. Interestingly, the hardness of ℓ_∞ c-ANN for $c < 3$ is not a coincidence. Indyk shows that the ℓ_∞ c-ANN problem in this case is actually at least as hard as the *supserset query problem*. The only solutions for this problem are known to either be linear in query time or exponential in space.

Definition 22 (Superset Query Problem)**.** *Given n sets $S_1 \cdots S_n$ such that $S_j \subset X = \{1 \cdots d\}$, devise a data structure which for any query $Q \subset X$, distinguishes:*

- *If $\exists S_j$ such that $S_j \subset Q$, return S_j*
- *Else, return $\emptyset$*

Note that each set S_j maps to a subset of coordinates $1 \cdots d$. We create a database point x from each S_j such that $x_i = 1$ if $i \in S_j$ and $x_i = \frac{1}{3}$ otherwise. For the query point y, we set $y_i = \frac{2}{3}$ if $i \in Q$ and $y_i = 0$ otherwise. It is easy to verify that

- $\exists S_j \subset Q \implies \|x - y\|_\infty = \frac{1}{3}$
- $\nexists S_j \subset Q \implies \|x - y\|_\infty = 1$

The result implies the following lemma.

Theorem 23. *For $c < 3$, the superset query problem is reducible to the c-ANN problem under ℓ_∞.*

5.2 Reduction to Orthogonal Range Search

We now show that ℓ_∞ c-ANN is easier than the orthogonal range search problem, as orthogonal range search (ORS) data structures can solve ℓ_∞ c-ANN search for arbitrary c. The main idea is that searching for points within a certain ℓ_∞ radius of y is reducible to an orthogonal range search over a cube centered at y with diameter $2c$. The orthogonal range search problem is defined below:

Definition 24 (Orthogonal Range Search)**.** *Given a set of n points in d-dimensional space, build a data structure such that for any query rectangle R, the data structure reports all points in R.*

To show that this problem is harder than ℓ_∞ c-ANN, we show that a data structure for d-dimensional ORS, such as a range tree, can be used to solve ℓ_∞ c-ANN. We begin by storing all of our data set points in a range tree T. On an approximate nearest neighbor query for vector y, we define a rectangle R where for a given coordinate i, the interval of our query rectangle is

$$R_i = [y_i - c, y_i + c]$$

We then perform an orthogonal range search for R on T. If the range search returns a point, we know that this point is a c-ANN to query point y since all its indices are within c of the indices of y. If no point is returned, then no point falls in the range interval, and thus no point is within c of y. Thus, the ORS problem is at least as hard as the c-ANN ℓ_∞ search problem. Additionally, we also note range trees can be combined with weight-balanced trees to be fully dynamic[2], which gives us the following result.

Theorem 25 (Range Trees)**.** *There exists a dynamic ℓ_∞ c-ANN data structure using range trees that uses $O(n \log^{d-1} n)$ space and supports $O(\log^{d-1} n)$ query time and $O(\log^d n)$ update time.*

Intuitively, ORS data structures must solve a much harder problem than c-ANN under ℓ_∞. Range trees must *always* return a point within c of y if one exists, must work with ranges of any size that differ per coordinate, and do not get to know this range in advance. In this way, the improved bounds of our ℓ_∞ c-ANN data structure are non-trivial, as our model is able to beat range trees in two different ways:

- We can avoid exponential in d blow-up for either query time or space and update time
- We have worst case $(1 + \lceil \frac{2}{c-1} \rceil)^d$ blow-up instead of $\log^{O(d)} n$

Note that the second case is an improvement when $c \geq 1 + \omega(\frac{1}{\log n})$.

At this point, the complexity gap between orthogonal range search and ℓ_∞ c-ANN is still unclear. While ℓ_∞ c-ANN is certainly easier for sufficient c, an interesting question is whether an ℓ_∞ c-ANN data structure with a strong enough approximation factor, such as $c = 1 + \epsilon$, can be used to solve certain orthogonal range search problems. We leave this as an open question for future work.

References

[1] Alexandr Andoni. *Nearest Neighbor Search - the Old, the New, and the Impossible.* PhD thesis, Massachusetts Institute of Technology, 2009.

[2] Piotr Indyk. On approximate nearest neighbors under l-infinity norm. *J. Comput. Syst. Sci.*, 63(4):627–638, December 2001.

[3] J. B. Saxe and J. L. Bentley. Transforming static data structures to dynamic structures. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pages 148–168, Oct 1979.

[4] Mark H. Overmars. *Design of Dynamic Data Structures.* Springer-Verlag, Berlin, Heidelberg, 1987.

[5] Piotr Indyk. *High-dimensional Computational Geometry.* PhD thesis, Stanford, CA, USA, 2001. AAI3000045.

[6] Arne Andersson, Mikkel Thorup, and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3), June 2007.

[2] https://pdfs.semanticscholar.org/841a/31780b7e8f4de224fac06181321ca2ea807e.pdf

6 Appendix

6.1 A

Lemma 26. *Let $m = \frac{|M|}{|P|}$, $r = \frac{|R|}{|P|}$, and $l = \frac{|L|}{|P|}$ such that $l + r + m = 1$. Define*

$$L(m, r) = \log_{1/(m+r)}(\frac{1}{m} - 1)$$

For any set P and $\rho > 0$, either

- *(1) We can choose t such that $L(m, r) \leq \rho$ and $l, r > \frac{1}{4d}$*
- *(2) There is a $C \subset P$ of size $\frac{|P|}{2}$ having diameter $\leq 4\lceil \log_{1+\rho} \log 4d \rceil$*

Proof. Intuitively, the first condition says there exists a good separator such that $L(m, r) \leq \rho$ and more than $\frac{1}{4d}$ points are not replicated in L and R. Indyk proves if this condition cannot be satisfied, then half the points can be contained in a box with diameter $O(\log_{1+\rho} \log d)$. Thus, assume that for every choice of $r > \frac{1}{4d}$, we have $L(m, r) \geq \rho$. For a choice of i, let $M_i = \{x \in P : x_i \in [2i, 2i + 2)\}$, $m_i = \frac{|M_i|}{|P|}$, and define

$$r_i = \sum_{j \geq i} m_i, l_i = \sum_{j \leq i} m_i$$

Claim 27. $r_{i+1} > \frac{1}{4d} \implies r_{i+1} < r_i^{1+\rho} \implies r_i \leq \frac{1}{2^{(1+\rho)^i}}$

Assume $r_{i+1} > \frac{1}{4d}$. From (1) not being satisfied, $L(m_i, r_{i+1}) > \rho$. Thus we have

$$\log_{\frac{1}{m_i + r_{i+1}}} \frac{m_i + r_{i+1}}{r_{i+1}} > \rho \implies \frac{m_i + r_{i+1}}{r_{i+1}} > \frac{1}{(m_i + r_{i+1})^\rho}$$

Using $r_{i+1} + m_i = r_i$, then:

$$\frac{r_i}{r_{i+1}} > \frac{1}{r_i^\rho} \implies r_i^{\rho+1} > r_{i+1}$$

WLOG, we can translate the origin such that $r_0 = \frac{1}{2}$, then we have:

$$(\frac{1}{2})^{(\rho+1)^i} > r_i$$

Set $i = O(\log_{1+\rho} \log 4d)$. Then $r_i \leq \frac{1}{4d}$ and $l_{-(i+1)} \leq \frac{1}{4d}$ by symmetry. Therefore, there are at least $1 - \frac{1}{4d} - \frac{1}{4d} = 1 - \frac{1}{2d}$ fraction of points x where $x_i \in [-i, i - 1]$. We remove all of these points from P. Repeating over all d coordinates, then we have at least $1 - \frac{1}{2d}d = \frac{1}{2}$ fraction of points remaining. Thus, we have $\frac{|P|}{2}$ points remaining where the diameter is:

$$|[-i, i - 1]| \leq O(i) = O(\log_{1+\rho} \log d)$$

□

6.2 B

	Approx	S(n)	Q(n)	I(n)	D(n)
A	$4\lceil\log_{1+\rho}\log d\rceil+1$	$O(dn^{1+\rho})$	$O(d\log n)$	$\emptyset$	$\emptyset$
B	$c=3$	$n^{O(\log d)}$	$O(d\log n)$	$\emptyset$	$\emptyset$
C	$4\lceil\log_{1+\rho}\log d\rceil+1$	$O(dn^{1+\rho})$	$O(d\log^2 n)$	$O(d^2n^\rho\log n)$	$\emptyset$
D	$4\lceil\log_{1+\rho}\log d\rceil+1$	$O(dn^{1+\rho})$	$O(d\log^2 n)$	$O(d^2n^\rho\log n)$	$O(n^{1+\rho})$
E	$4\lceil\log_{1+\rho}\log d\rceil+1$	$O(dn^{1+\rho})$	$O(d^2\log^3 n)$	$O(d^2n^\rho\log n)$	$O(dn^\rho)$
F	$c\geq 3$	$O(dn)$	$O(2^d\log n)$	$O(d\log n)$	$O(d\log n)$
G	$c\geq 3$	$O(dn2^d)$	$O(d\log n)$	$O(2^d\log n)$	$O(2^d\log n)$
H	$c\geq 3$	$O(dn)$	$O(2^d\log\log\frac{U}{c-1})$	$O(d\log\log\frac{U}{c-1})$	$O(d\log\log\frac{U}{c-1})$
I	$c\geq 3$	$O(dn2^d)$	$O(d\log\log\frac{U}{c-1})$	$O(2^d\log\log\frac{U}{c-1})$	$O(2^d\log\log\frac{U}{c-1})$
J	$c\geq 3$	$O(dn)$	$O(2^d\frac{\log n}{\log\log U})$	$O(d(\log\log n+\frac{\log n}{\log\log U}))$	$O(d(\log\log n+\frac{\log n}{\log\log U}))$
K	$c\geq 3$	$O(dn2^d)$	$O(d\frac{\log n}{\log\log U})$	$O(2^d(\log\log n+\frac{\log n}{\log\log U}))$	$O(2^d(\log\log n+\frac{\log n}{\log\log U}))$
L	$c\geq 3$	$O(dn)$*	$O(2^d)$*	$O(d)$*	$O(d)$*
M	$c\geq 3$	$O(dn2^d)$*	$O(d)$*	$O(2^d)$*	$O(2^d)$*
N	$\forall c$	$O(dn2^{\frac{d}{c-1}})$*	$O(d)$*	$\emptyset$	$\emptyset$
O	$\forall c$	$O(dn2^{\frac{d}{c-1}})$*	$O(d\log n)$*	$O(2^{\frac{d}{c-1}})$*	$O(2^{\frac{d}{c-1}}+\log n)$*
P	$\forall c$	$O(nd)$	$O((1+\lceil\frac{2}{c-1}\rceil)^d)$	$O(nd)$	$O(nd)$
Q	$\forall c$	$O(nd(1+\lceil\frac{2}{c-1}\rceil)^d)$	$O(d)$	$O((1+\lceil\frac{2}{c-1}\rceil)^d)$	$O((1+\lceil\frac{2}{c-1}\rceil)^d)$
R	$\forall c$	$O(n\log^{d-1}n)$	$O(\log^{d-1}n)$	$O(\log^d n)$	$O(\log^d n)$

Figure 5: Dynamization for ℓ_∞ c-ANN data structures

Legend:
A: Indyk static
B: Indyk static 3-approximation
C: Indyk + Overmars
D: This Paper (Naive Deletions)
E: Indyk's dynamization
F: This Paper (Linear Space BST)
G: This Paper (Sublinear Query BST)
H: This Paper (Linear Space van Emde Boas)
I: This Paper (Sublinear Query van Emde Boas)
J: This Paper (Linear Space Fusion Trees)
K: This Paper (Sublinear Query Fusion Trees)
L: This Paper (Linear Space Hash Tables)
M: This Paper (Sublinear Query Hash Tables)
N: This Paper (Static Randomized Sublinear Query Hash Tables)
O: This Paper (Dynamized Randomized Sublinear Query Hash Tables)
P: This Paper (Generalized Linear Space Hash Tables)
Q: This Paper (Generalized Sublinear Query Hash Tables)
R: Range Trees
* = on expectation