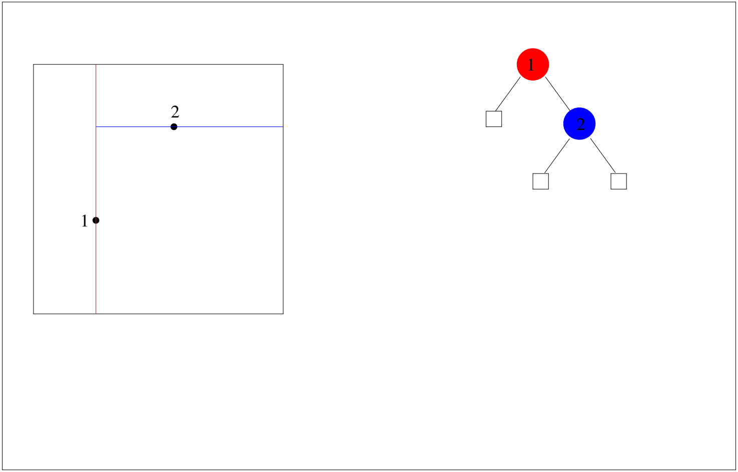
On the average cost of insertions on random relaxed K-d trees

Amalia Duch[1] Conrado Martínez[1]

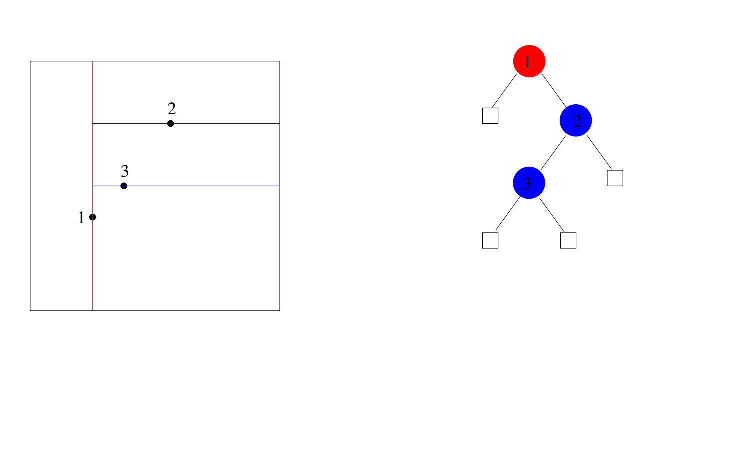
[1]Univ. Politècnica de Catalunya, Spain

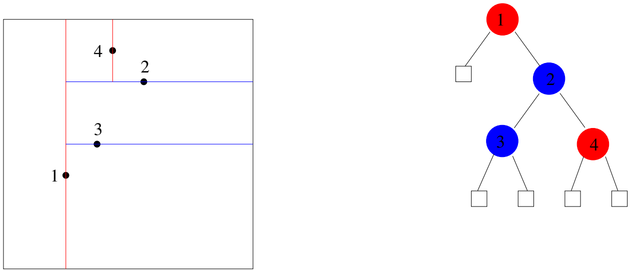
- A relaxed K-d tree is a variant of K-d trees (Bentley, 1975), where each node stores a random discriminant i, $0 \leq i < K$
- They were introduced by Duch, Estivill-castro and Martínez (1998) and subsequently analyzed by Martínez, Panholzer and Prodinger (2001), by Duch and Martínez (2002a, 2002b), and by Broutin, Dalal, Devroye and McLeish (2006)

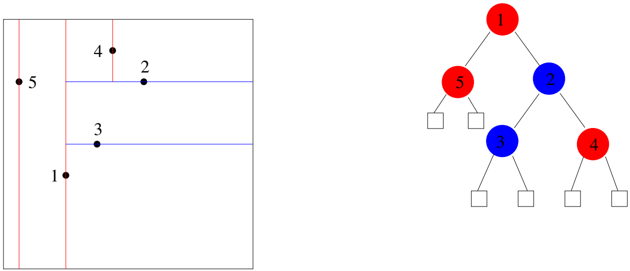

1
1



2
1
1
2


2
3
1
1
2
3


4
2
3
1
1
2
3
4


4
2
5
3
1
1
5
2
3
4

- Relaxation allows insertions at arbitrary positions
- Subtree sizes can be used to guarantee randomness under arbitrary insertions or deletions, hence we can provide guarantees on expected performance
- The average performance of associative queries (e.g., partial match, orthogonal range search, nearest neighbors) is slightly worse than standard K-d trees

```
struct node {
        Elem key;
        int discr, size;
        node* left, * right;
};
typedef node* rkdt;
```

Insertion in relaxed K-d trees

```
rkdt insert(rkdt t, const Elem& x) {
     int n = size(t);
     int u = random(0,n); // returns a random int in [0..n]
     if (u == n)
        return insert_at_root(t, x);
     else { // t cannot be empty
        int i = t -> discr;
        if (x[i] < t -> key[i])
           t -> left = insert(t -> left, x);
        else
           t -> right = insert(t -> right, x);
        return t;
     }
}
```

Insertion at root

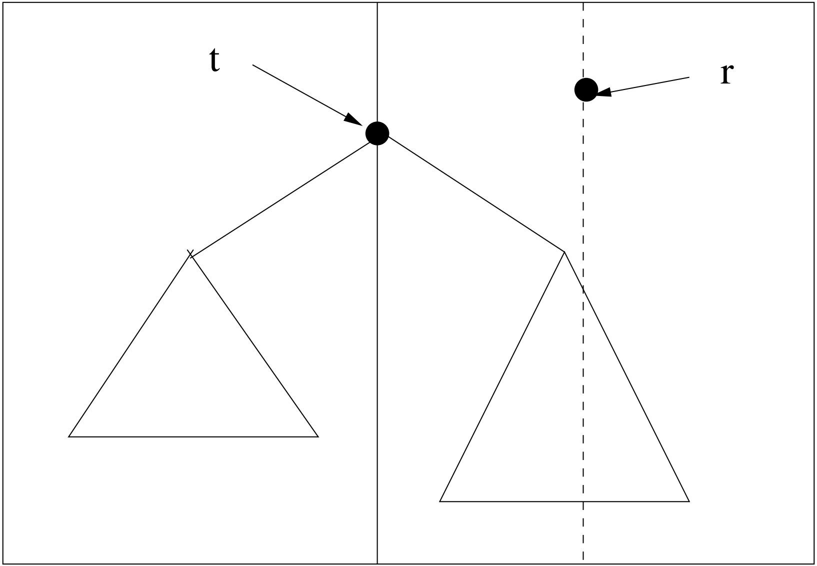
```
rkdt insert_at_root(rkdt t, const Elem& x) {
     rkdt r = new node;
     r -> info = x;
     r -> discr = random(0, K-1);
     pair<rkdt, rkdt> p = split(t, r);
     r -> left = p.first;
     r -> right = p.second;
     return r;
}
```

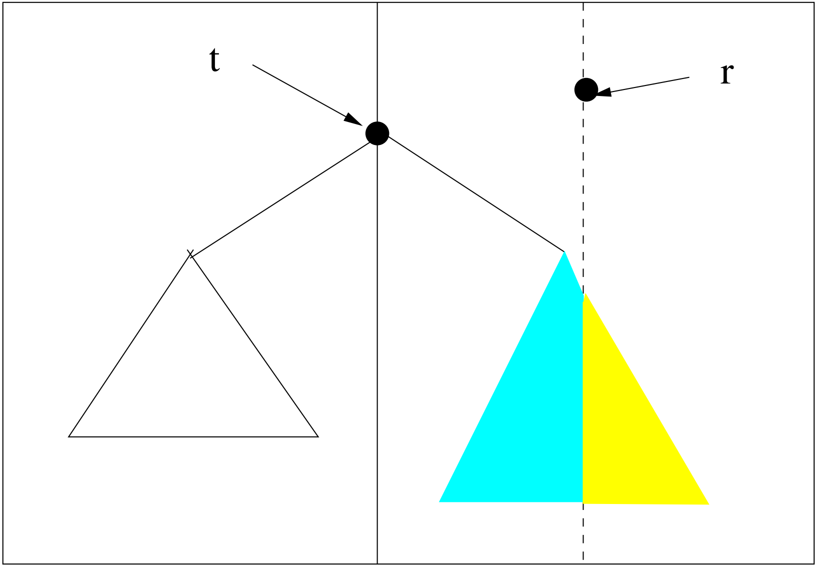
Split

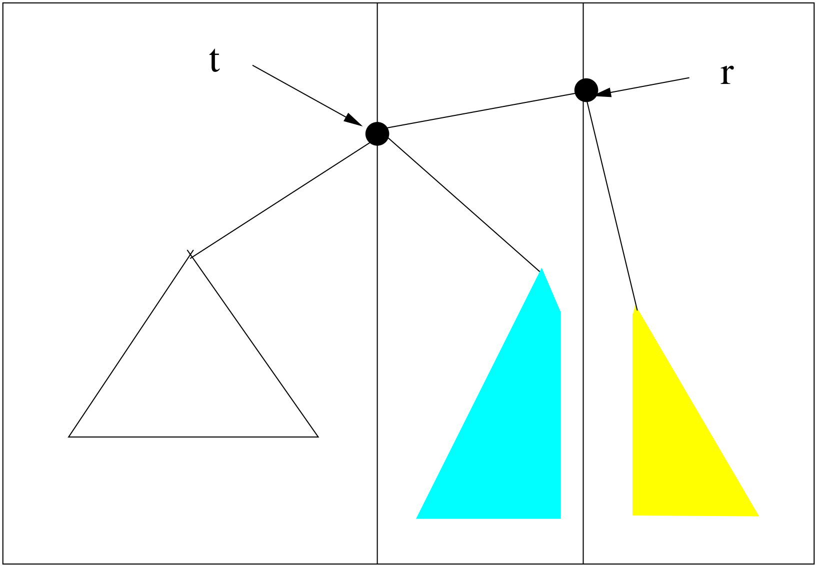
```
pair<rkdt, rkdt> split(rkdt t, rkdt r) {
     if (t == NULL) return make_pair(NULL, NULL);
     int i = r -> discr; int j = t -> discr;
     if (i == j) {
          // Case I
          ...
     } else {
          // Case II
          ...
     }
}
```

Split: Case I

```
if (i == j) {
     if (r -> key[i] < t -> key[i]) {
      pair<rkdt,rkdt> p = split(t -> left, r);
      t -> left = p.second;
      return make_pair(p.first, t);
   } else {
      pair<rkdt, rkdt> p = split(t -> right, r);
      t -> right = p.first;
      return make_pair(t, p.second);
   }
} else { // i != j
      ...
}
```

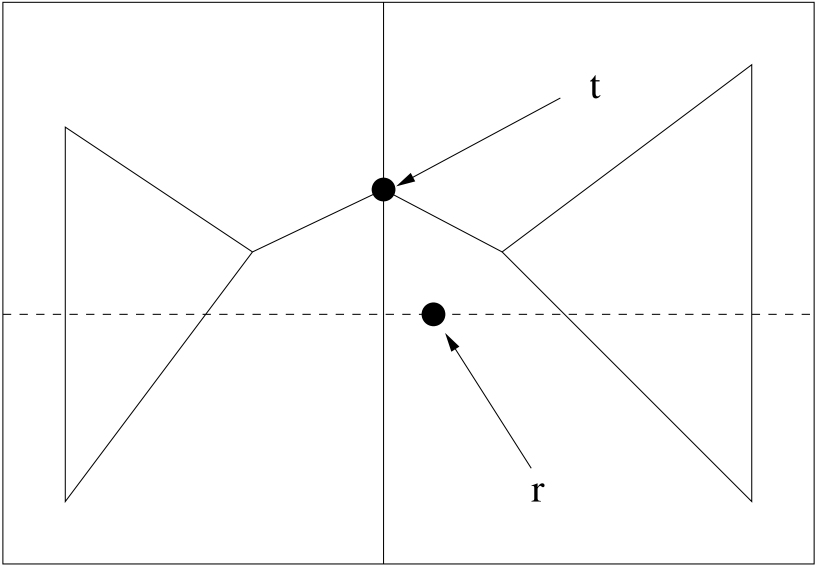

t
r

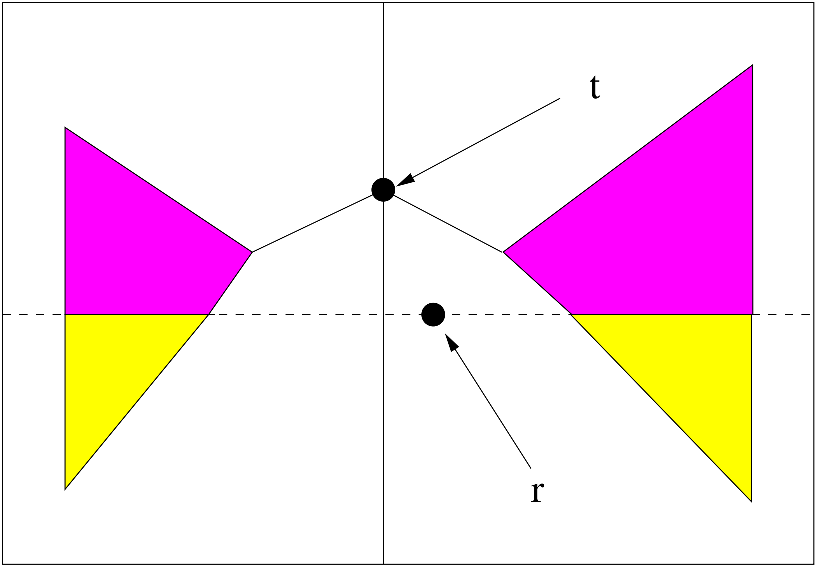

t
r

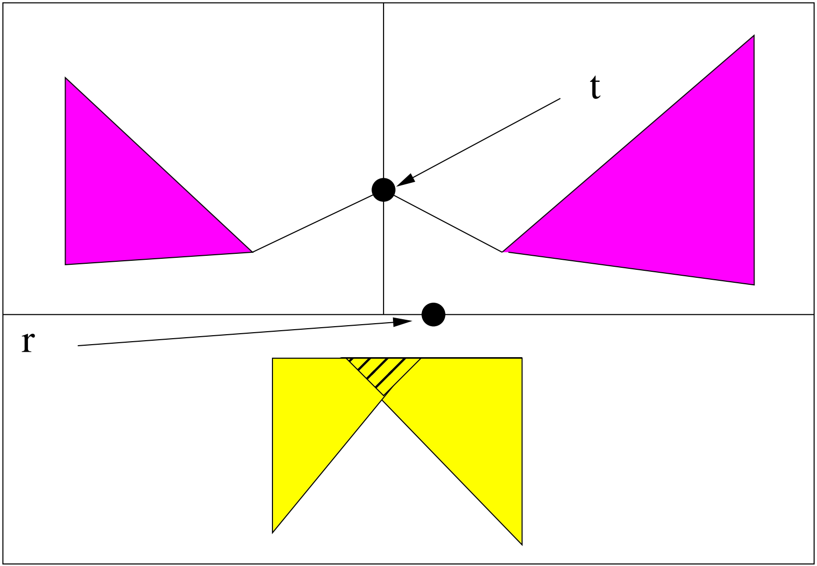

t
r

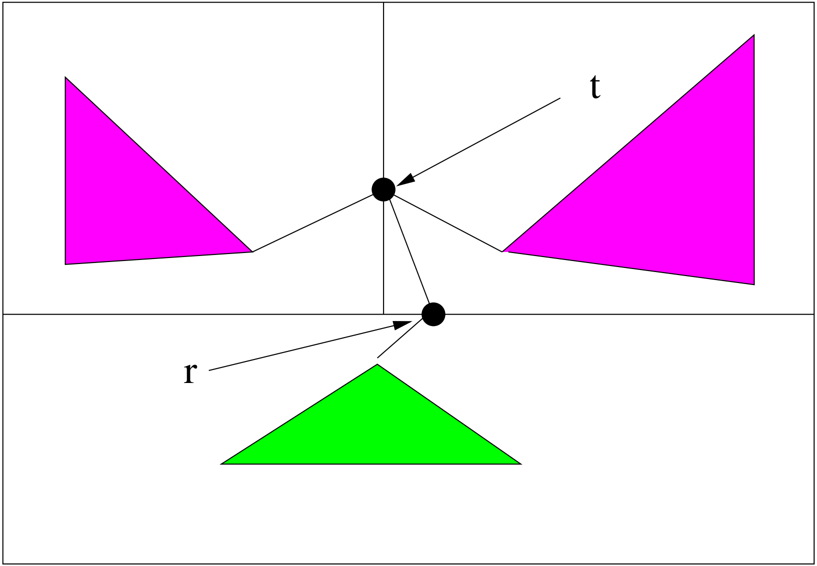
Split: Case II

```
if (i == j) {
    ...
} else { // i != j
    pair<rkdt, rkdt> L = split(t -> left, r);
    pair<rkdt, rkdt> R = split(t -> right, r);
    if (r -> key[i] < t -> key[i]) {
      t -> left = L.second;
      t -> right = R.second;
      return make_pair(join(L.first, R.first, j), t);
    } else {
      t -> left= L.first;
      t -> right = R.first;
      return make_pair(t, join(L.second, R.second, j));
    }
}
```


t
r


t
r


t
r


t
r

Deletion in relaxed K-d trees

```
rkdt delete(rkdt t, const Elem& x) {
     if (t == NULL) return NULL;
     int i = t -> discr;
     if (t -> key == x)
        return join(t -> left, t -> right, i);
     if (x -> key[i] < t -> key[i]) {
        t -> left = delete(t -> left, x);
     else
        t -> right = delete(t -> right, x);
     return t;
}
```

Joining two trees

```
rkdt join(rkdt L, rkdt R, int i) {
  if (L == NULL) return R;
  if (R == NULL) return L;

  // L != NULL and R != NULL
  int m = size(L); int n = size(R);
  int u = random(0, m+n-1);
  if (u < m) // with probability m / (m + n)
             // the joint root is that of L
     ...
  else       // with probability n / (m + n)
             // the joint root is that of R
}
```

- s_n = avg. number of visited nodes in a split
- m_n = avg. number of visited nodes in a join
- $$s_n = 1 + \frac{2}{nK} \sum_{0 \le j < n} \frac{j+1}{n+1} s_j + \frac{2(K-1)}{nK} \sum_{0 \le j < n} s_j + \frac{K-1}{K} \sum_{0 \le j < n} \pi_{n,j} m_j,$$

  where $\pi_{n,j}$ is probability of joining two trees with total size j.

- s_n = avg. number of visited nodes in a split
- m_n = avg. number of visited nodes in a join
-

$$s_n = 1 + \frac{2}{nK} \sum_{0 \le j < n} \frac{j+1}{n+1} s_j + \frac{2(K-1)}{nK} \sum_{0 \le j < n} s_j + \frac{K-1}{K} \sum_{0 \le j < n} \pi_{n,j} m_j,$$

where $\pi_{n,j}$ is probability of joining two trees with total size j.

- s_n = avg. number of visited nodes in a split
- m_n = avg. number of visited nodes in a join
- $$s_n = 1 + \frac{2}{nK} \sum_{0 \le j < n} \frac{j+1}{n+1} s_j + \frac{2(K-1)}{nK} \sum_{0 \le j < n} s_j + \frac{K-1}{K} \sum_{0 \le j < n} \pi_{n,j} m_j,$$

  where $\pi_{n,j}$ is probability of joining two trees with total size j.

- The recurrence for s_n is

$$s_n = 1 + \frac{2}{nK} \sum_{0\le j<n} \frac{j+1}{n+1} s_j + \frac{2(K-1)}{nK} \sum_{0\le j<n} s_j + \frac{2(K-1)}{nK} \sum_{0\le j<n} \frac{n-j}{n+1} m_j,$$

  with $s_0 = 0$.
- The recurrence for m_n has exactly the same shape with the rôles of s_n and m_n interchanged; it easily follows that $s_n = m_n$.

- Define

$$S(z) = \sum_{n \geq 0} s_n z^n$$

- The recurrence for s_n translates to

$$z \frac{d^2 S}{dz^2} + 2 \frac{1-2z}{1-z} \frac{dS}{dz} - 2 \left(\frac{3K-2}{K} - z \right) \frac{S(z)}{(1-z)^2} = \frac{2}{(1-z)^3},$$

with initial conditions $S(0) = 0$ and $S'(0) = 1$.

- The homogeneous second order linear ODE is of hypergeometric type.
- An easy particular solution of the ODE is

$$-\frac{1}{2}\left(\frac{K}{K-1}\right)\frac{1}{1-z}$$

Theorem

The generating function $S(z)$ of the expected cost of split is, for any $K \geq 2$,

$$S(z) = \frac{1}{2} \frac{1}{1 - \frac{1}{K}} \left[(1-z)^{-\alpha} \cdot {}_2F_1\left(\left. \begin{matrix} 1-\alpha, 2-\alpha \\ 2 \end{matrix} \right| z \right) - \frac{1}{1-z} \right],$$

where $\alpha = \alpha(K) = \frac{1}{2}\left(1 + \sqrt{17 - \frac{16}{K}}\right)$.

Theorem

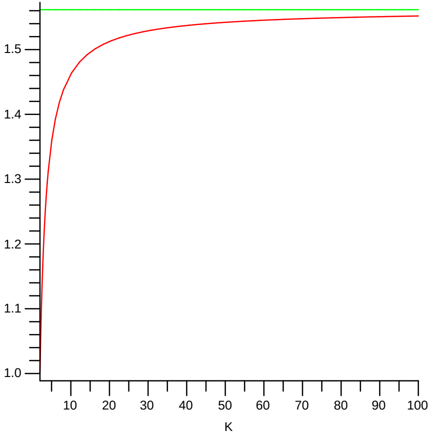
The expected cost s_n of splitting a relaxed K-d tree of size n is

$$s_n = \beta\, n^{\phi(K)} + o(n),$$

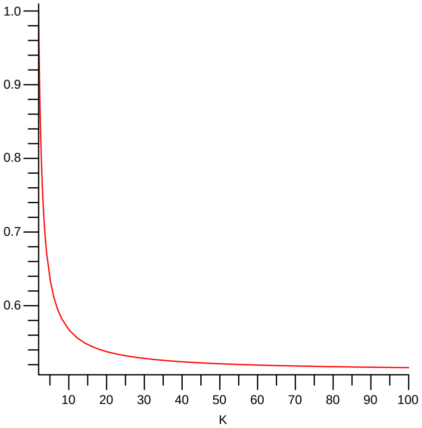
with

$$\beta = \frac{1}{2}\frac{1}{1-\frac{1}{K}}\frac{\Gamma(2\alpha-1)}{\alpha\Gamma^3(\alpha)},$$

$$\phi = \alpha - 1 = \frac{1}{2}\left(\sqrt{17-\frac{16}{K}}-1\right).$$



Plot of $\phi(K)$



Plot of $\beta(K)$

- The recurrence for the expected cost of an insertion is

$$
\begin{aligned}
\mathbb{E}[I_n] &= \frac{s_n}{n+1} \\
&\quad + \left(1 - \frac{1}{n+1}\right)\left(1 + \frac{2}{n}\sum_{0\le j<n}\frac{j+1}{n+1}\,\mathbb{E}[I_j]\right) \\
&= \frac{s_n}{n+1} + 1 + \mathcal{O}\left(\frac{1}{n}\right) + \frac{2}{n+1}\sum_{0\le j<n}\frac{j+1}{n+1}\,\mathbb{E}[I_j]\,.
\end{aligned}
$$

- The expected cost of deletions satisfies a similar recurrence; it is asymptotically equivalent to the average cost of insertions

- For $K = 2$,

$$\mathbb{E}[I_n] = 4 \ln n + \mathcal{O}(1),$$

- For $K > 2$,

$$\mathbb{E}[I_n] = \frac{\beta}{1 - \frac{2}{\phi+1}} n^{\phi-1} + 2 \ln n + \mathcal{O}(1)$$
$$= \beta \frac{\phi+1}{\phi-1} n^{\phi-1} + 2 \ln n + \mathcal{O}(1).$$

where β and ϕ are as in previous theorem.

Recently, we have succeeded in showing that the copy-based insertion at root and root deletion algorithms by Broutin, Dalal, Devroye and McLeish (2006) have sublinear complexity for any K. We find explicit closed forms for the factor and exponent in the leading term. This leads to insertions and deletions in expected logarithmic time, as the "reconstruction" phase has expected constant time.