Randomized Data Structures

Amalia Duch and Conrado Martínez

Univ. Politècnica de Catalunya, Spain

LATIN 2022
November 7–11, 2022
Guanajuato, Mexico

Introduction


R. Karp


N. C. Metropolis


M. O. Rabin

The usefulnees of randomization in the design of algorithms has been known for a long time:

- Metropolis' algorithms
- Rabin's primality test
- Rabin-Karp's string search

Introduction

- Hashing is another early success of randomization for the design of data structures.
- For example, selecting the hash function from a universal class (Carter and Wegman, 1977) guarantees expected performance
- Worst-case analysis of hashing is trivial and useless in practice, we need to carry out a detailed probabilistic analysis of the performance
- The probabilistic analysis of various hash tables assumes that the probability that $\text{HASH}(x) = j$ is $1/M$ for all possible keys x and all possible hash values $j \in [0..M-1]$, where M is the number of memory slots in the hash table

Introduction

Randomization yields algorithms:

- Simple and elegant
- Practical
- With guaranteed expected performance
- Without assumptions on the probabilistic distribution of the input

Introduction

- The usual worst-case analysis is not useful for randomized algorithms
- The probabilistic model to use in the analysis is under control; it is not a working hypothesis, but built-in

Introduction

Two types of algorithms:

Las Vegas: Answers are always correct, only probabilistic guarantees on their performance (e.g., running time)

Montecarlo: Answers might be wrong with probability $\leq \epsilon < 1/2$; using amplification we can make the probability as small as needed

- One-sided error: there are only false positives or only false negatives
- Two-sided error: false positives and false negatives are possible

Introduction

- Randomization for the design of data structures renders usually “Las Vegas” algorithms to search and/or update the data structures, e.g., skip lists, randomized binary search trees, universal hashing
- But there are also “Montecarlo” data structures, e.g., Bloom filters, which might give wrong answers (with small probability)

Introduction

In this course:

- Skip lists
- Randomized binary search trees
- Randomized multidimensional data structures
- Bloom filters
- Universal hashing (if time permits)

Skip lists

W. Pugh

- Skip lists were invented by William Pugh (C. ACM, 1990) as a simple alternative to balanced trees
- The algorithms to search, insert, delete, etc. are very simple to understand and to implement, and they have very good expected performance—independent of any assumption on the input

Skip lists

W. Pugh

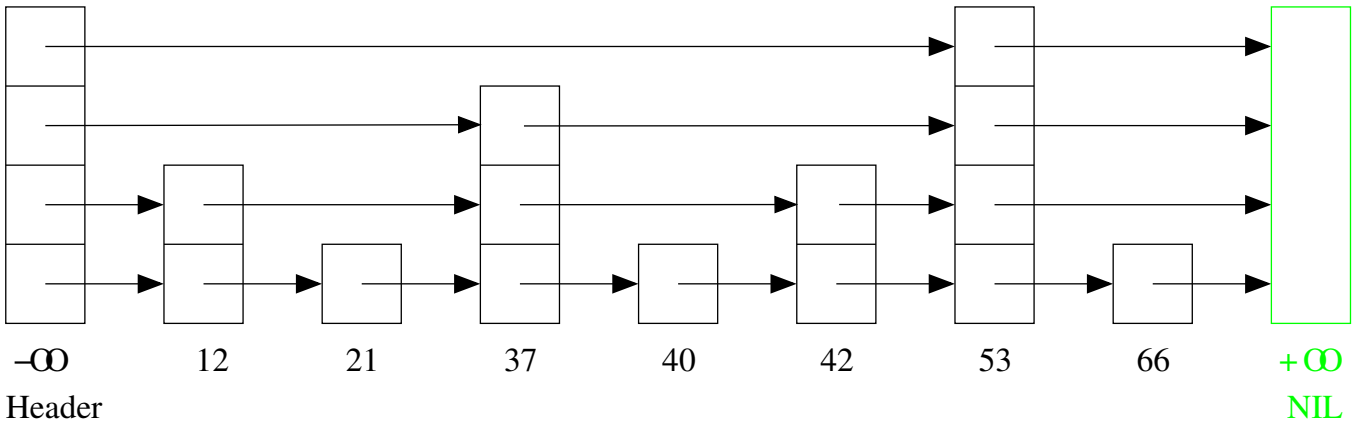
- Skip lists were invented by William Pugh (C. ACM, 1990) as a simple alternative to balanced trees
- The algorithms to search, insert, delete, etc. are very simple to understand and to implement, and they have very good expected performance—independent of any assumption on the input

Skip lists

A skip list S for a set X consists of:

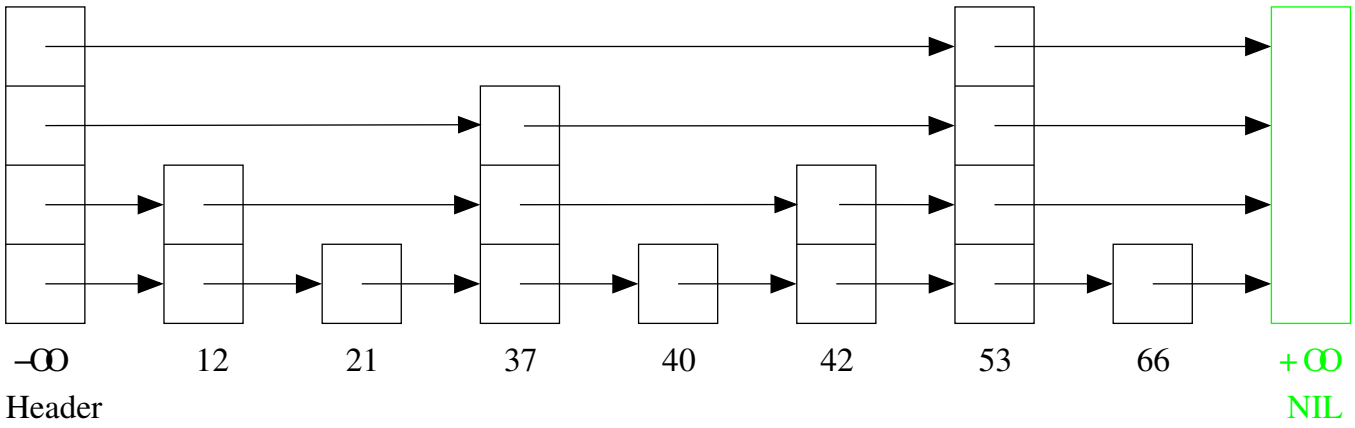
1. A sorted linked list L_1, called level 1, contains all elements of X
2. A collection of non-empty sorted lists L_2, L_3, ..., called level 2, level 3, ... such that for all $i \geq 1$, if an element x belongs to L_i then x belongs to L_{i+1} with probability q, for some $0 < q < 1$, $p := 1 - q$

Skip lists



To implement this, we store the items of X in a collection of nodes each holding an item and a variable-size array of pointers to the item's successor at each level; an additional dummy node gives access to the first item of each level

Skip lists



To implement this, we store the items of X in a collection of nodes each holding an item and a variable-size array of pointers to the item's successor at each level; an additional dummy node gives access to the first item of each level

Skip lists

- The level or height of a node x, height(x), is the number of lists it belongs to.
- It is given by a geometric r.v. of parameter p:

$$\Pr\{\text{height}(x) = k\} = pq^{k-1}, \qquad q = 1 - p$$

- The height of the skip list S is the number of non-empty lists,

$$\text{height}(S) = \max_{x \in S}\{\text{height}(x)\}$$

Skip lists

- The level or height of a node x, height(x), is the number of lists it belongs to.
- It is given by a geometric r.v. of parameter p:

$$\Pr\{\text{height}(x) = k\} = pq^{k-1}, \qquad q = 1 - p$$

- The height of the skip list S is the number of non-empty lists,

$$\text{height}(S) = \max_{x \in S}\{\text{height}(x)\}$$

Skip lists

- The level or height of a node x, height(x), is the number of lists it belongs to.
- It is given by a geometric r.v. of parameter p:

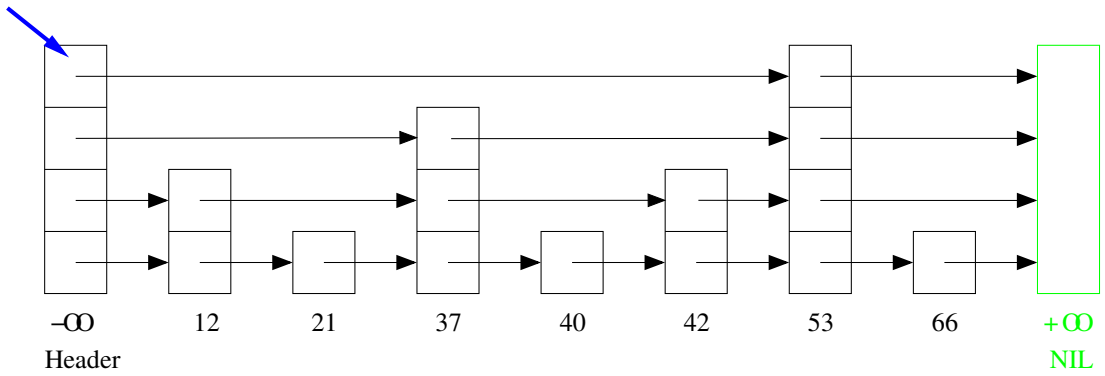
$$\Pr\{\text{height}(x) = k\} = pq^{k-1}, \qquad q = 1 - p$$

- The height of the skip list S is the number of non-empty lists,

$$\text{height}(S) = \max_{x \in S}\{\text{height}(x)\}$$

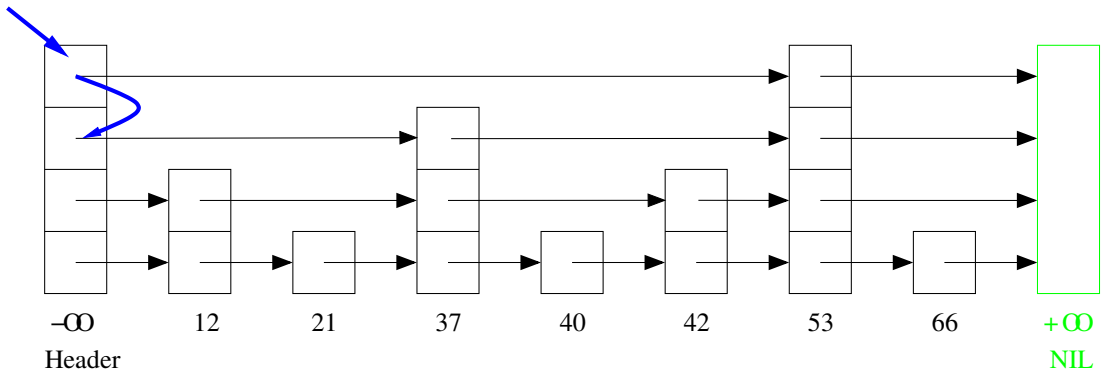
Searching in a skip list

Searching for an item x, $42 < x \leq 53$



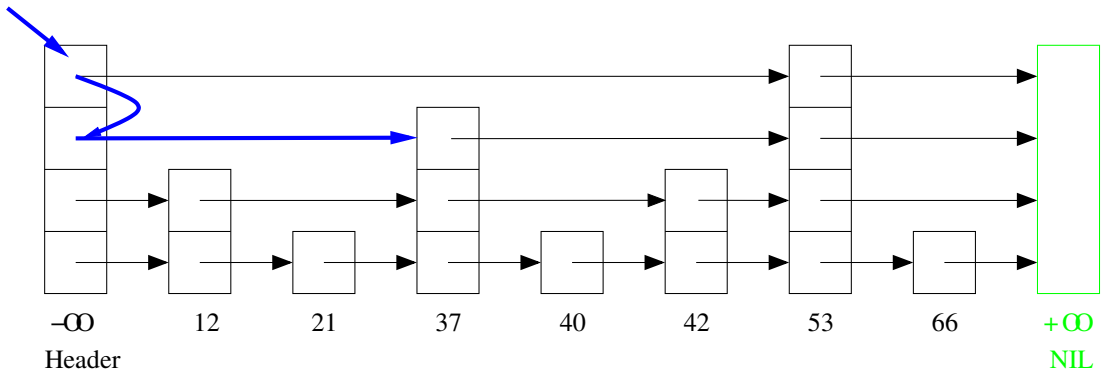
Searching in a skip list

Searching for an item x, $42 < x \leq 53$



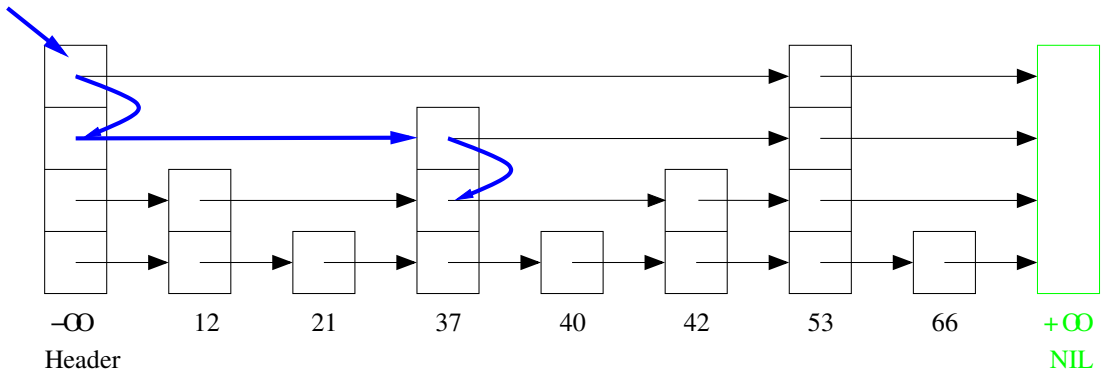
Searching in a skip list

Searching for an item x, $42 < x \leq 53$



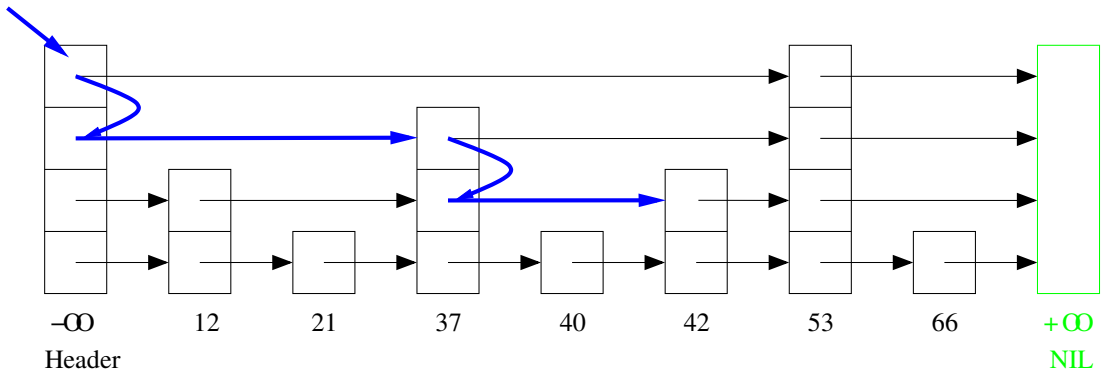
Searching in a skip list

Searching for an item x, $42 < x \leq 53$



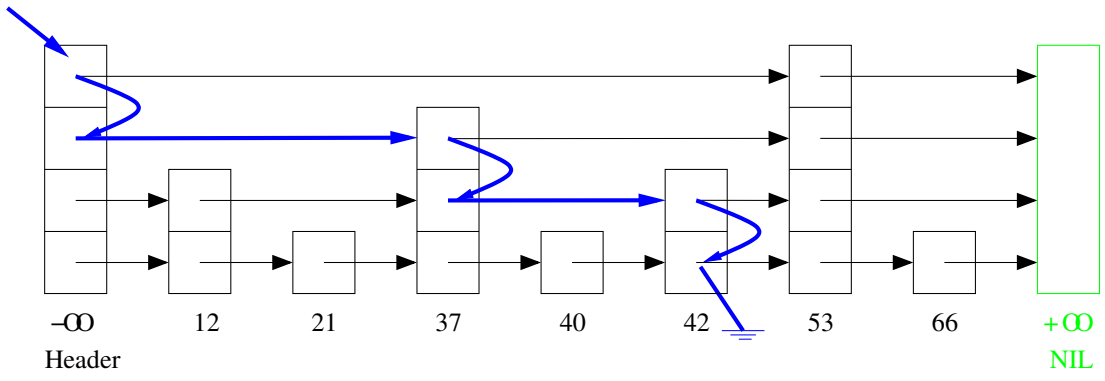
Searching in a skip list

Searching for an item x, $42 < x \leq 53$



Searching in a skip list

Searching for an item x, $42 < x \leq 53$

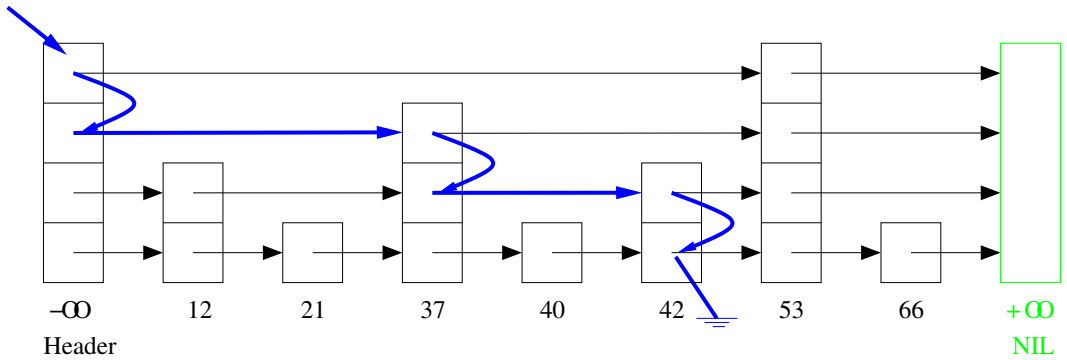


Implementing skip lists

```
▷ Returns pointer to item with key k or null
▷ if not such item exists in the skip list S
procedure SEARCH(k, S)
    p := S.header
    ℓ := S.height
    while ℓ > 0 do
        if p → next[ℓ] = null ∨ k ≤ p → next[ℓ] → key then
            ℓ := ℓ − 1
        else
            p := p → next[ℓ]
    if p → next[1] = null ∨ k ≠ p → next[1] → key then
        ▷ k is not present
        return null
    else  ▷ k is present, return pointer to the node
        return p → next[1]
```

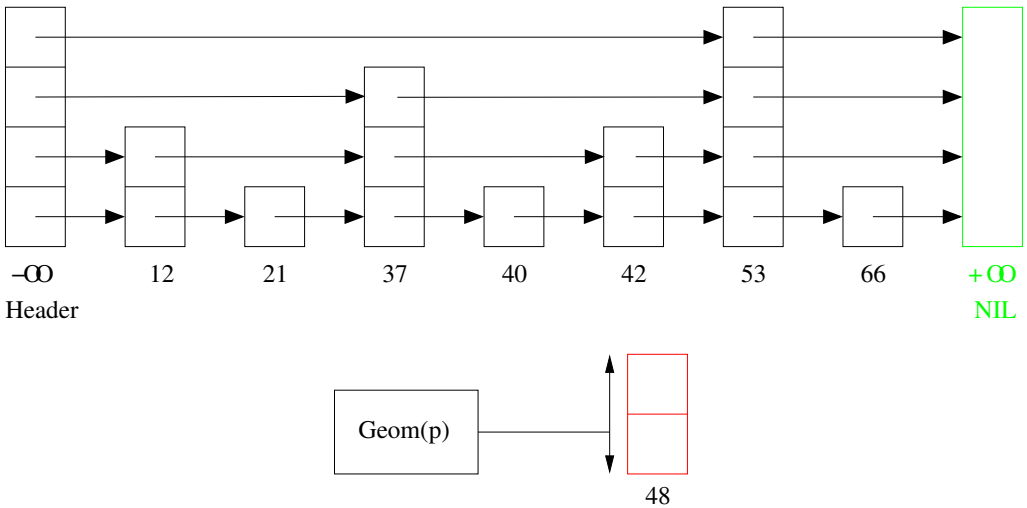
Insertion in a skip list

Inserting an item $x = 48$



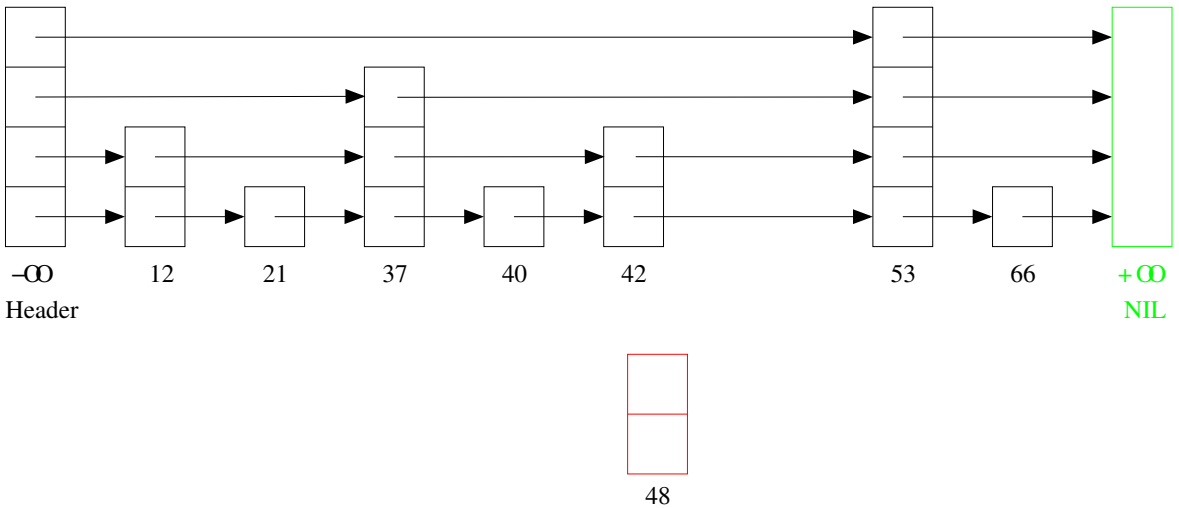
Insertion in a skip list

Inserting an item $x = 48$



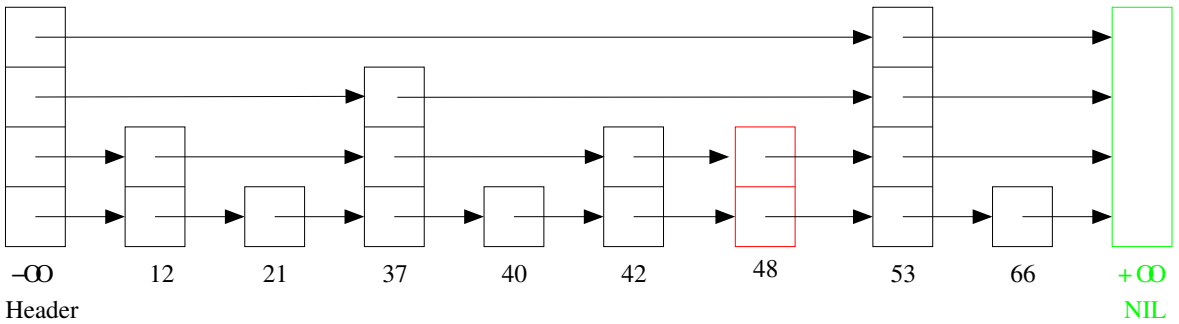
Insertion in a skip list

Inserting an item $x = 48$



Insertion in a skip list

Inserting an item $x = 48$



Implementing skip lists

To insert a new item we go through four phases:

1) Search the given key. The search loop is slightly different from before, since we need to keep track of the last node seen at each level before descending from that level to the one immediately below.
2) If the given key is already present we only update the associated value and finish.

Implementing skip lists

```
▷ Inserts new item ⟨k, v⟩ or
▷ updates value if key k is present in the skip list S
procedure INSERT(k, v, S)
    p := S.header; ℓ := S.height
    create array pred of pointers of size S.height
    for i := 1 to S.height do pred[i] := S.header
    while ℓ > 0 do
        if p → next[ℓ] = null ∨ k ≤ p → next[ℓ] → key then
            ▷ p should be the predecessor of the new item
            ▷ at level ℓ
            pred[ℓ] := p; ℓ := ℓ − 1
        else
            p := p → next[ℓ]
    ...
```

Implementing skip lists

```
procedure INSERT(k, v, S)
    ...
    while ... do
        ▷ loop to locate whether k is present or not
        ▷ and to determine predecessors at each level
    if p → next[1] = null ∨ k ≠ p → next[1] → key then
        ▷ k is not present
        ▷ Insert new item, see next slide
    else
        ▷ k is present, update its value
        p → next[1] → value := v
```

Implementing skip lists

3) When k is not present, create a new node with key k and value v, and assign a random level r to the new node, using geometric distribution
4) Link the new node in the first r lists, adding empty lists if r is larger than the maximum level of the skip list

Implementing skip lists

▷ Insert new item
▷ RNG() generates a random number $U(0,1)$

$h := 1$;
while RNG() $> p$ **do** $h := h + 1$
$nn :=$ **new** NODE(k, v, h)
if $h > S$.height **then**
    Resize S.header and *pred* with $h - S$.height
    new pointers, all set to **null** and S.header, resp.
    S.height $:= h$
for $i := 1$ **to** h **do**
    $nn \rightarrow$ next$[i] :=$ *pred*$[i] \rightarrow$ next$[i]$
    pred$[i] \rightarrow$ next$[i] := nn$

Other Operations

- Deletions are also very easy to implement
- Ordered raversal of the keys is trivially implemented
- Skip lists can also support many other operations, e.g., merging, search and deletion by rank, finger search, . . .
- They can also support concurrency and massive parallelism without too much effort

Other Operations

- Deletions are also very easy to implement
- Ordered raversal of the keys is trivially implemented
- Skip lists can also support many other operations, e.g., merging, search and deletion by rank, finger search, . . .
- They can also support concurrency and massive parallelism without too much effort

Other Operations

- Deletions are also very easy to implement
- Ordered raversal of the keys is trivially implemented
- Skip lists can also support many other operations, e.g., merging, search and deletion by rank, finger search, . . .
- They can also support concurrency and massive parallelism without too much effort

Other Operations

- Deletions are also very easy to implement
- Ordered raversal of the keys is trivially implemented
- Skip lists can also support many other operations, e.g., merging, search and deletion by rank, finger search, ...
- They can also support concurrency and massive parallelism without too much effort

Performance of skip lists

A preliminary rough analysis considers the search path backwards. Imagine we are at some node x and level i:

- The height of x is $> i$ and we come from level $i + 1$ since the sought key k is smaller than the key of the successor of x at level $i + 1$
- The height of x is i and we come from x's predecessor at level i since k is larger or equal to the key at x

Performance of skip lists

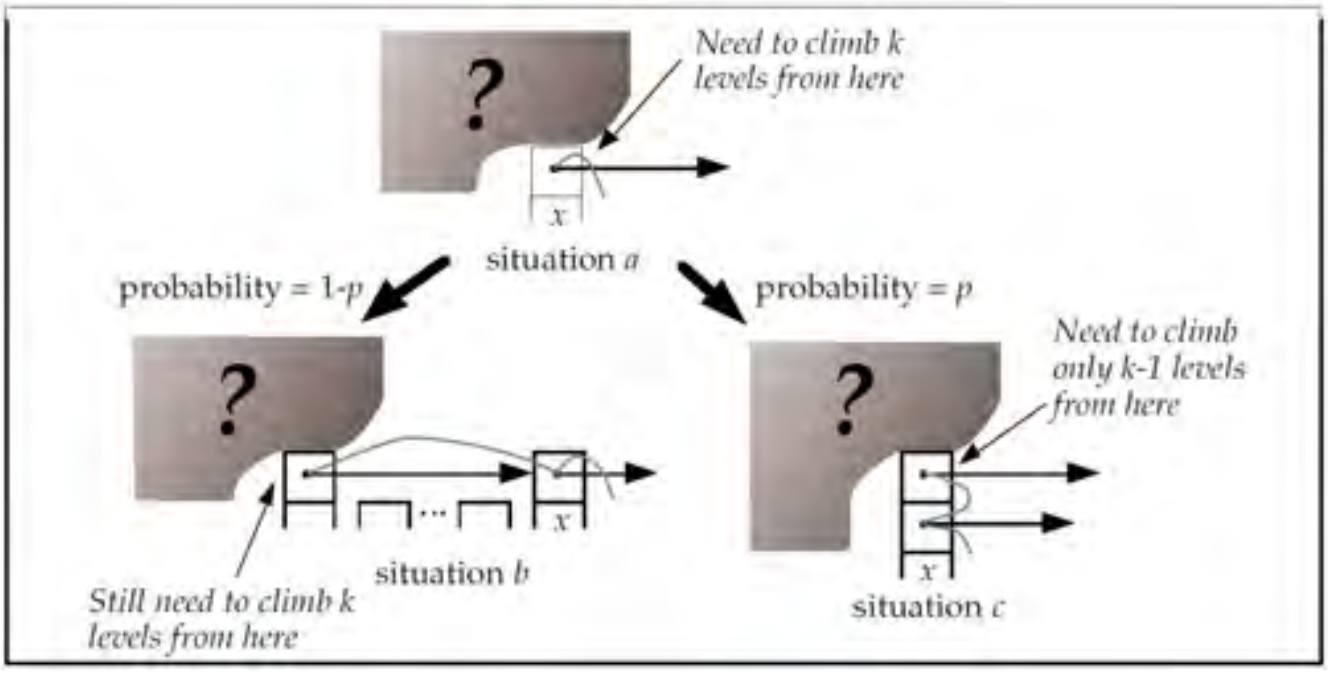


Figure from W. Pugh's *Skip Lists: A Probabilistic Alternative to Balanced Trees* (C. ACM, 1990)—the meaning of p is the opposite of what we have used!

Performance of skip lists

The expected number $C(k)$ of steps to "climb" k levels in an infinite list

$$\begin{aligned} C(k) &= p(1 + C(k)) + (1 - p)(1 + C(k-1)) \\ &= 1 + pC(k) + qC(k-1) = \frac{1}{q}(1 + qC(k-1)) \\ &= \frac{1}{q} + C(k-1) = k/q \end{aligned}$$

since $C(0) = 0$.

Performance of skip lists

The analysis above is pessimistic since the list is not infinite and we might "bump" into the header. Then all remaining backward steps to climb up to a level k are vertical—no more horizontal steps. Thus the expected number of steps to climb up to level L_n is

$$\leq (L_n - 1)/q$$

Performance of skip lists

- L_n = the largest level L for which

$$\mathbb{E}[\#\text{ of nodes with height} \geq L] \leq 1/q$$

- Probability that a node has height $\geq k$ is

$$\Pr\{\text{height}(x) \geq k\} = \sum_{i \geq k} pq^{i-1} = pq^{k-1} \sum_{i \geq 0} q^i = q^{k-1}$$

- Number of nodes with height $\geq k$ is a binomial r.v. with parameters n and q^{k-1}, hence

$$\mathbb{E}[\#\text{ of nodes with height} \geq k] = nq^{k-1}$$

- Then

$$nq^{L_n - 1} = 1/q \implies L_n = \log_q(1/n) = \log_{1/q} n$$

Performance of skip lists

- L_n = the largest level L for which

$$\mathbb{E}[\#\text{ of nodes with height} \geq L] \leq 1/q$$

- Probability that a node has height $\geq k$ is

$$\Pr\{\text{height}(x) \geq k\} = \sum_{i \geq k} pq^{i-1} = pq^{k-1} \sum_{i \geq 0} q^i = q^{k-1}$$

- Number of nodes with height $\geq k$ is a binomial r.v. with parameters n and q^{k-1}, hence

$$\mathbb{E}[\#\text{ of nodes with height} \geq k] = nq^{k-1}$$

- Then

$$nq^{L_n - 1} = 1/q \implies L_n = \log_q(1/n) = \log_{1/q} n$$

Performance of skip lists

- L_n = the largest level L for which

$$\mathbb{E}[\#\text{ of nodes with height} \geq L] \leq 1/q$$

- Probability that a node has height $\geq k$ is

$$\Pr\{\text{height}(x) \geq k\} = \sum_{i \geq k} pq^{i-1} = pq^{k-1} \sum_{i \geq 0} q^i = q^{k-1}$$

- Number of nodes with height $\geq k$ is a binomial r.v. with parameters n and q^{k-1}, hence

$$\mathbb{E}[\#\text{ of nodes with height} \geq k] = nq^{k-1}$$

- Then

$$nq^{L_n - 1} = 1/q \implies L_n = \log_q(1/n) = \log_{1/q} n$$

Performance of skip lists

- L_n = the largest level L for which

$$\mathbb{E}[\#\text{ of nodes with height} \geq L] \leq 1/q$$

- Probability that a node has height $\geq k$ is

$$\Pr\{\text{height}(x) \geq k\} = \sum_{i \geq k} pq^{i-1} = pq^{k-1} \sum_{i \geq 0} q^i = q^{k-1}$$

- Number of nodes with height $\geq k$ is a binomial r.v. with parameters n and q^{k-1}, hence

$$\mathbb{E}[\#\text{ of nodes with height} \geq k] = nq^{k-1}$$

- Then

$$nq^{L_n - 1} = 1/q \implies L_n = \log_q(1/n) = \log_{1/q} n$$

Performance of skip lists

Then the steps remaining to reach H_n (=the height of a random skip list of size n) can analyzed this way:

- we need not more horizontal steps than nodes with height $\geq L_n$, the expected number is $\leq 1/q$, by definition
- the probability that $H_n > k$ is

$$1 - \left(1 - q^k\right)^n \leq nq^k$$

- the expected value of the height H_n can be bounded as

$$\begin{aligned}
\mathbb{E}[H_n] &= \sum_{k \geq 0} \mathbb{P}[H_n > k] = \sum_{0 \leq k < L_n} \mathbb{P}[H_n > k] + \sum_{k \geq L_n} \mathbb{P}[H_n > k] \\
&\leq L_n + \sum_{k \geq 0} \mathbb{P}[H_n > L_n + k] = L_n + nq^{L_n} \sum_{k \geq 0} q^k \\
&= L_n + 1/p
\end{aligned}$$

thus the expected additional vertical steps need to reach H_n from L_n is $\leq 1/p$

Performance of skip lists

Summing up, the expected path length of a search is

$$\leq (L_n - 1)/q + 1/q + 1/p = \frac{1}{q} \log_{1/q} n + 1/p$$

On the other hand, the average number of pointers per node is $1/p$ so there is a trade-off between space and time:

- $p \to 0, q \to 1 \implies$ very tall "nodes", short horizontal cost
- $p \to 1, q \to 0 \implies$ flat skip lists
- Pugh suggested $p = 3/4$ as a good practical choice; the optimal choice minimizes factor $(q \ln(1/q))^{-1} \implies$ $q = e^{-1} = 0.36\ldots, p = 1 - e^{-1} \approx 0.632\ldots$

Analysis of the height

W. Szpankowski

V. Rego

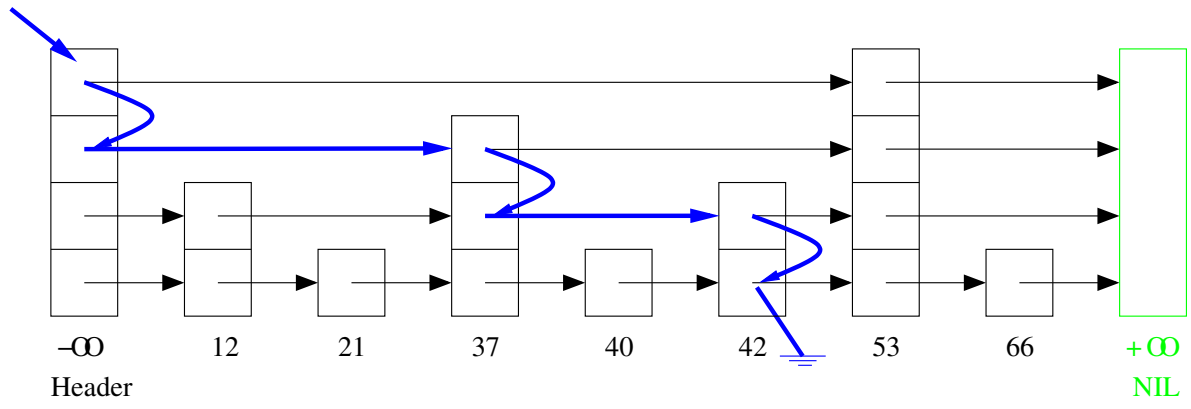
Theorem (Szpankowski and Rego,1990)

$$\mathbb{E}[H_n] = \log_{1/q} n + \frac{\gamma}{\ln(1/q)} - \frac{1}{2} + \chi(\log_{1/q} n) + \mathcal{O}(1/n)$$

where $\gamma = 0.577\ldots$ is Euler's constant and $\chi(t)$ a fluctuation of period 1, mean 0 and small amplitude.

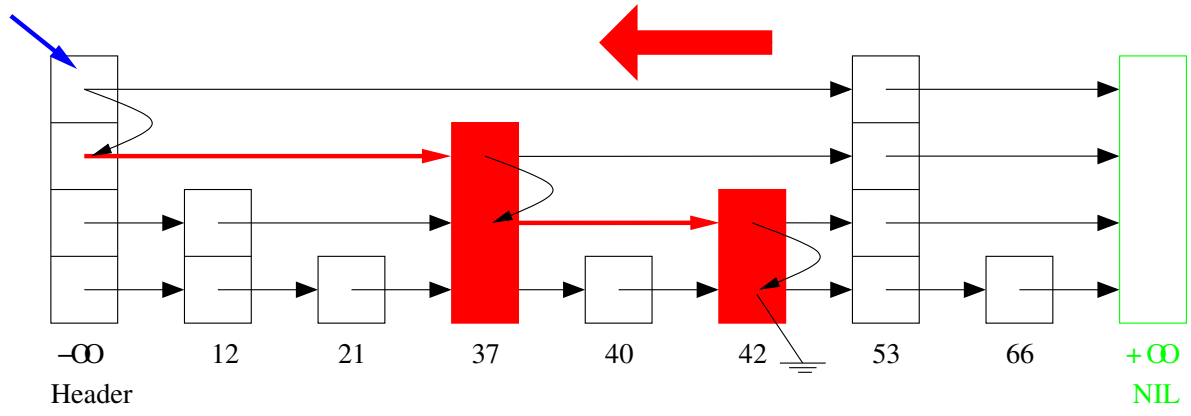
Analysis of the forward cost

The number of forward steps $F_{n,k}$ is the number of weak left-to-right maxima in $a_k, a_{k-1}, \ldots, a_1$, with $a_i = \text{height}(x_i)$



Analysis of the forward cost

The number of forward steps $F_{n,k}$ is the number of weak left-to-right maxima in $a_k, a_{k-1}, \ldots, a_1$, with $a_i = \text{height}(x_i)$



Analysis of the forward cost

- Total unsuccessful search cost

$$C_n = \sum_{0 \le k \le n} C_{n,k} = nH_n + F_n$$

- Total forward cost

$$F_n = \sum_{0 \le k \le n} F_{n,k}$$

Analysis of the forward cost

- Total unsuccessful search cost

$$C_n = \sum_{0 \le k \le n} C_{n,k} = nH_n + F_n$$

- Total forward cost

$$F_n = \sum_{0 \le k \le n} F_{n,k}$$

Analysis of the forward cost

P. Kirschenhofer

H. Prodinger

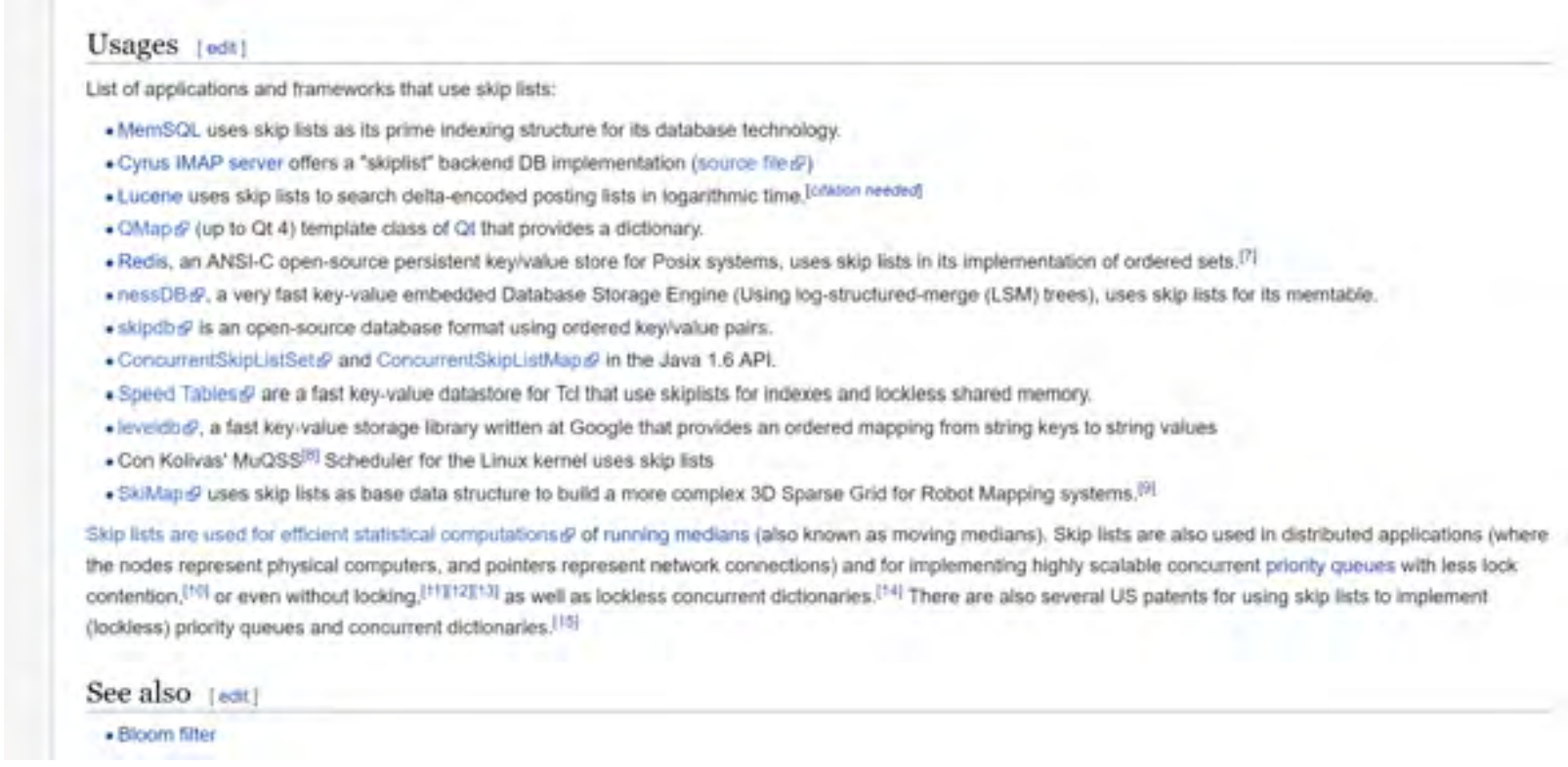
Theorem (Kirschehofer, Prodinger, 1994)

The expected total forward cost in a random skip list of size n is

$$\mathbb{E}[F_n] = \left(\frac{1}{q} - 1\right) \cdot n \cdot \left(\log_{1/q} n + \frac{\gamma - 1}{\ln(1/q)} - \frac{1}{2} + \frac{1}{\ln(1/q)} \chi(\log_{1/q} n)\right) + \mathcal{O}(\log n),$$

where $\gamma = 0.577\ldots$ is Euler's constant and χ a periodic fluctuation of period 1, mean 0 and small amplitude.

Skip Lists in Real Life



Usages [edit]

List of applications and frameworks that use skip lists:

- MemSQL uses skip lists as its prime indexing structure for its database technology.
- Cyrus IMAP server offers a "skiplist" backend DB implementation (source file)
- Lucene uses skip lists to search delta-encoded posting lists in logarithmic time.[citation needed]
- QMap (up to Qt 4) template class of Qt that provides a dictionary.
- Redis, an ANSI-C open-source persistent key/value store for Posix systems, uses skip lists in its implementation of ordered sets.[7]
- nessDB, a very fast key-value embedded Database Storage Engine (Using log-structured-merge (LSM) trees), uses skip lists for its memtable.
- skipdb is an open-source database format using ordered key/value pairs.
- ConcurrentSkipListSet and ConcurrentSkipListMap in the Java 1.6 API.
- Speed Tables are a fast key-value datastore for Tcl that use skiplists for indexes and lockless shared memory.
- leveldb, a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values
- Con Kolivas' MuQSS[8] Scheduler for the Linux kernel uses skip lists
- SkiMap uses skip lists as base data structure to build a more complex 3D Sparse Grid for Robot Mapping systems.[9]

Skip lists are used for efficient statistical computations of running medians (also known as moving medians). Skip lists are also used in distributed applications (where the nodes represent physical computers, and pointers represent network connections) and for implementing highly scalable concurrent priority queues with less lock contention,[10] or even without locking,[11][12][13] as well as lockless concurrent dictionaries.[14] There are also several US patents for using skip lists to implement (lockless) priority queues and concurrent dictionaries.[15]

See also [edit]

- Bloom filter

Source: Wikipedia

To learn more

[1] L. Devroye.
A limit theory for random skip lists.
The Annals of Applied Probability, 2(3):597–609, 1992.

[2] P. Kirschenhofer and H. Prodinger.
The path length of random skip lists.
Acta Informatica, 31(8):775–792, 1994.

[3] P. Kirschenhofer, C. Martínez and H. Prodinger.
Analysis of an Optimized Search Algorithm for Skip Lists.
Theoretical Computer Science, 144:199–220, 1995.

To learn more (2)

[4] T. Papadakis, J. I. Munro, and P. V. Poblete.
Average search and update costs in skip lists.
BIT, 32:316–332, 1992.

[5] H. Prodinger.
Combinatorics of geometrically distributed random variables: Left-to-right maxima.
Discrete Mathematics, 153:253–270, 1996.

[6] W. Pugh.
Skip lists: a probabilistic alternative to balanced trees.
Comm. ACM, 33(6):668–676, 1990.

[7] W. Pugh.
A Skip List Cookbook.
Technical Report UMIACS–TR–89–72.1. U. Maryland, College Park, 1989.

What are binary search trees? –quick remind

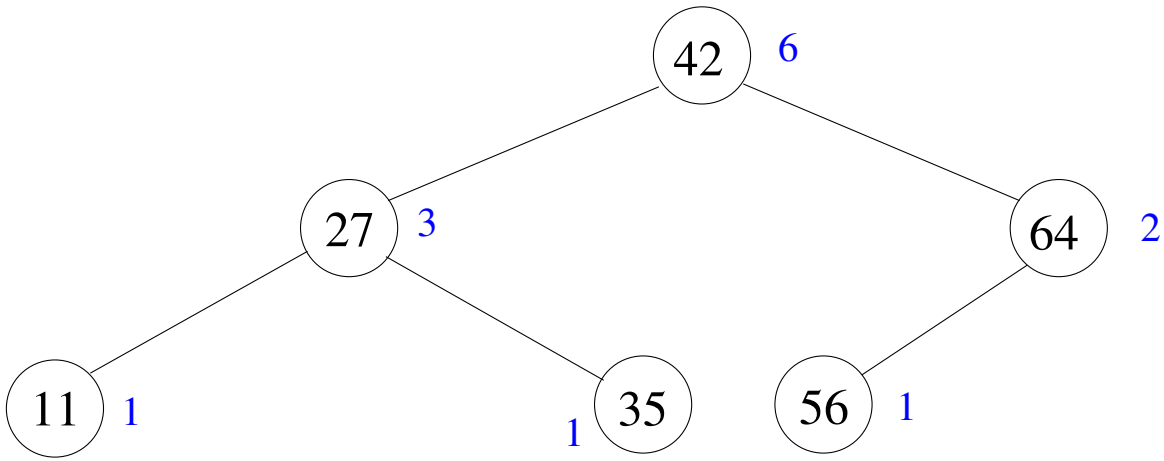
Definition

A binary search tree (or BST) T of size $n \geq 0$ is a binary tree that stores a set of n (distinct) keys, such that

- it is empty when $n = 0$, or
- its root stores a key x, and the remaining $n - 1$ keys are stored in the left and right subtrees of T, say L and R respectively, in such a way that both L and R are binary search trees and, for any key $u \in L$, it holds that $u < x$, and for any key $v \in R$, it holds that $x < v$.

BST: example

BST of size 6 built form keys: 42, 27, 64, 11, 35 and 56.



Random BSTs

- In a random binary search tree (built by a random permutation) any of its n elements is the root with probability $1/n$
- Idea: To obtain random BST –independently of any assumption on the distribution of the input– insert a new item in a tree of size n as follows:
  - insert it at the root with probability $1/(n+1)$,
  - otherwise proceed recursively

Random BSTs

- In a random binary search tree (built by a random permutation) any of its n elements is the root with probability $1/n$
- Idea: To obtain random BST –independently of any assumption on the distribution of the input– insert a new item in a tree of size n as follows:
  - insert it at the root with probability $1/(n+1)$,
  - otherwise proceed recursively

Random BSTs

- In a random binary search tree (built by a random permutation) any of its n elements is the root with probability $1/n$
- Idea: To obtain random BST –independently of any assumption on the distribution of the input– insert a new item in a tree of size n as follows:
  - insert it at the root with probability $1/(n+1)$,
  - otherwise proceed recursively

Random BSTs

- In a random binary search tree (built by a random permutation) any of its n elements is the root with probability $1/n$
- Idea: To obtain random BST –independently of any assumption on the distribution of the input– insert a new item in a tree of size n as follows:
  - insert it at the root with probability $1/(n+1)$,
  - otherwise proceed recursively

Randomized binary search trees

C. Aragon

R. Seidel

Two incarnations

- Randomized treaps (tree+heap) invented by Aragon and Seidel (FOCS 1989, Algorithmica 1996) use random priorities and bottom-up balancing
- Randomized binary search trees (RBSTs) invented by Martínez and Roura (ESA 1996, JACM 1998) use subtree sizes and top-down balancing

Randomized binary search trees


C. Aragon


R. Seidel

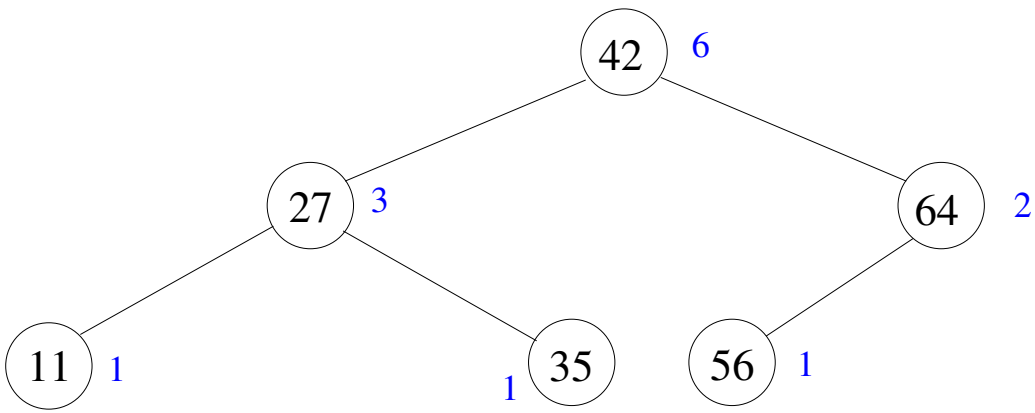

S. Roura

Two incarnations

- Randomized treaps (tree+heap) invented by Aragon and Seidel (FOCS 1989, Algorithmica 1996) use random priorities and bottom-up balancing
- Randomized binary search trees (RBSTs) invented by Martínez and Roura (ESA 1996, JACM 1998) use subtree sizes and top-down balancing

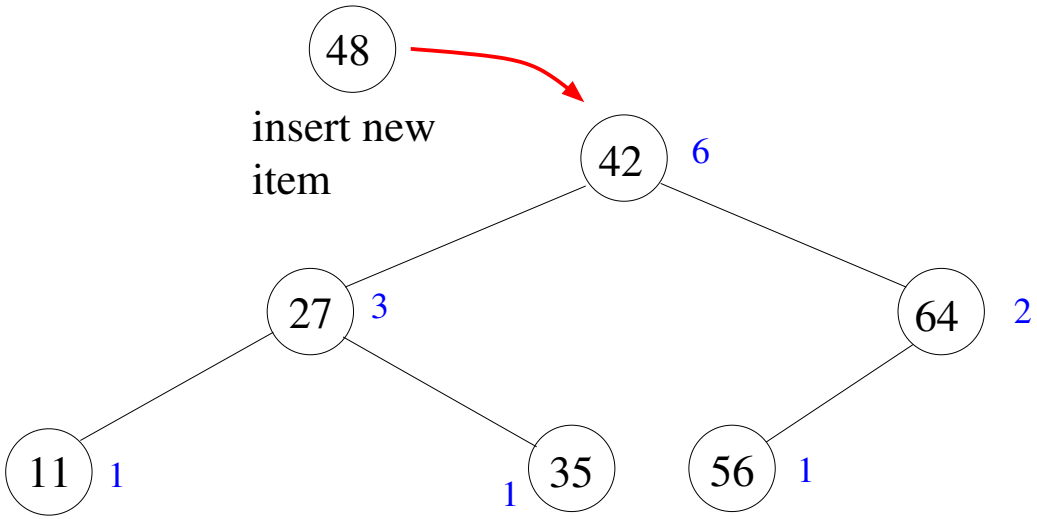
Insertion in a RBST

Inserting an item $x = 48$



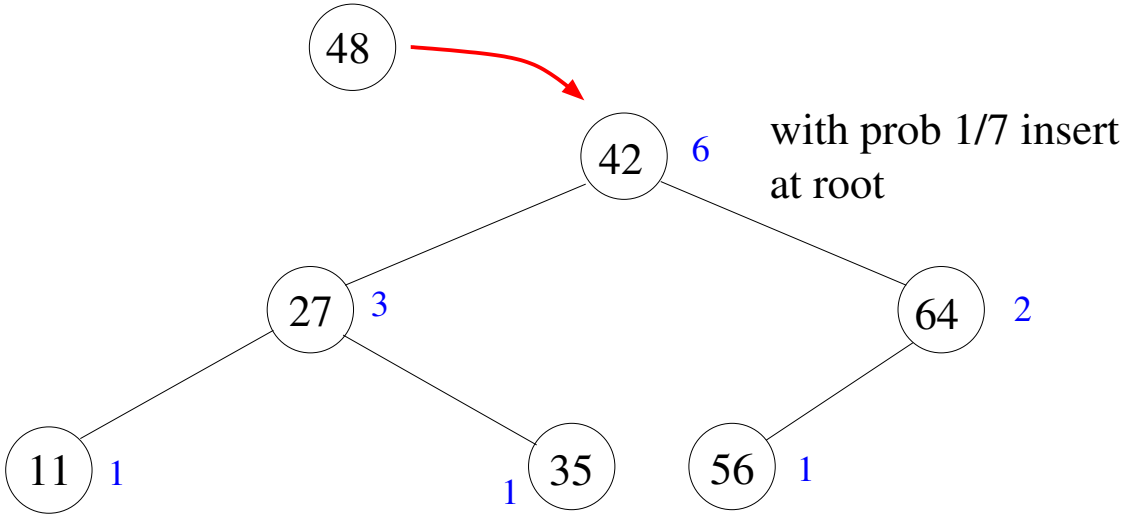
Insertion in a RBST

Inserting an item $x = 48$



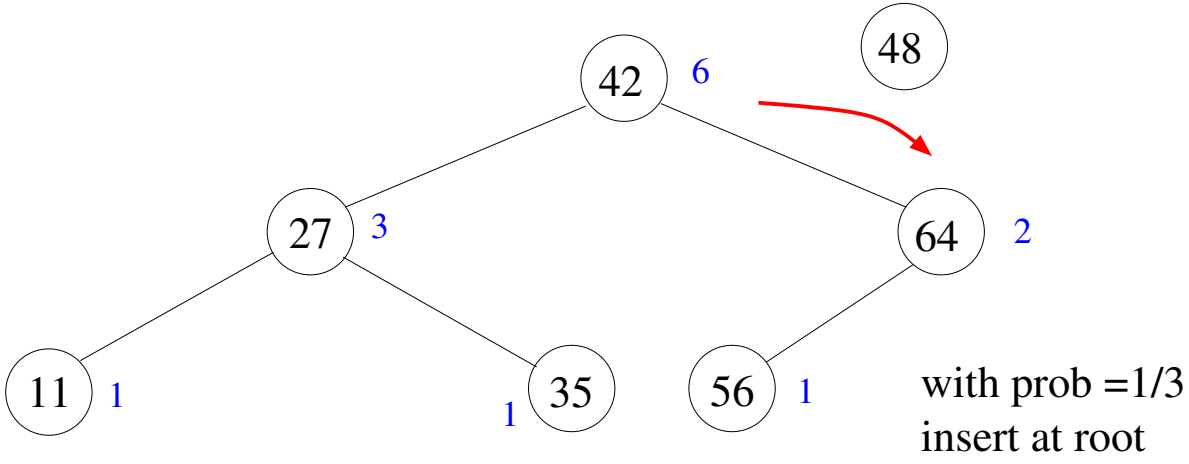
Insertion in a RBST

Inserting an item $x = 48$



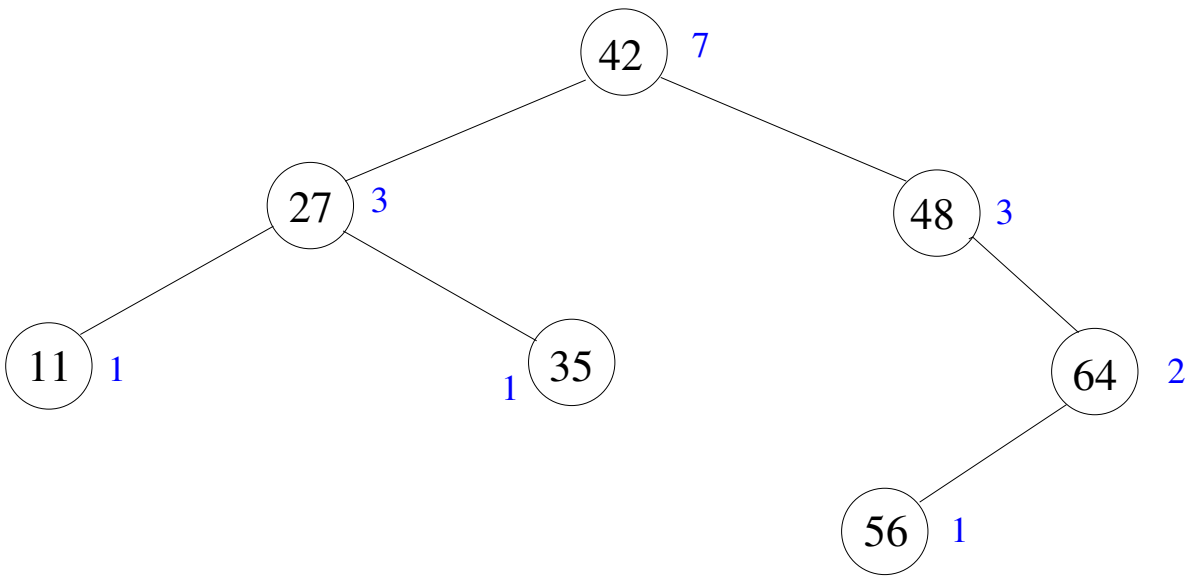
Insertion in a RBST

Inserting an item $x = 48$



Insertion in a RBST

Inserting an item $x = 48$



Insertion in a RBST

```
procedure INSERT(T, k, v)
    n := T → size   ▷ n = 0 if T = □
    if UNIFORM(0, n) = 0 then
        ▷ this will always succeed if T = □
        return INSERT-AT-ROOT(T, k, v)
    if k < T → key then
        T → left := INSERT(T → left, k, v)
    else
        T → right := INSERT(T → right, k, v)
    Update T → size
    return T
```

Insertion in a RBST

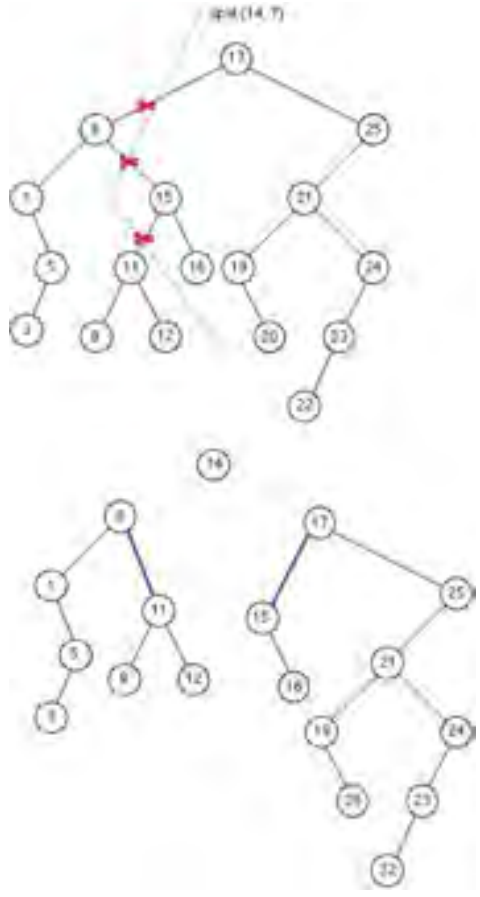
- To insert a new item x at the root of T, we use the algorithm SPLIT that returns two RBSTs T^- and T^+ with element smaller and larger than x, resp.

$$\begin{aligned} \langle T^-, T^+ \rangle &= \text{SPLIT}(T, x) \\ T^- &= \text{BST for } \{y \in T \mid y < x\} \\ T^+ &= \text{BST for } \{y \in T \mid x < y\} \end{aligned}$$

- SPLIT is like partition in Quicksort
- Insertion at root was invented by Stephenson in 1976

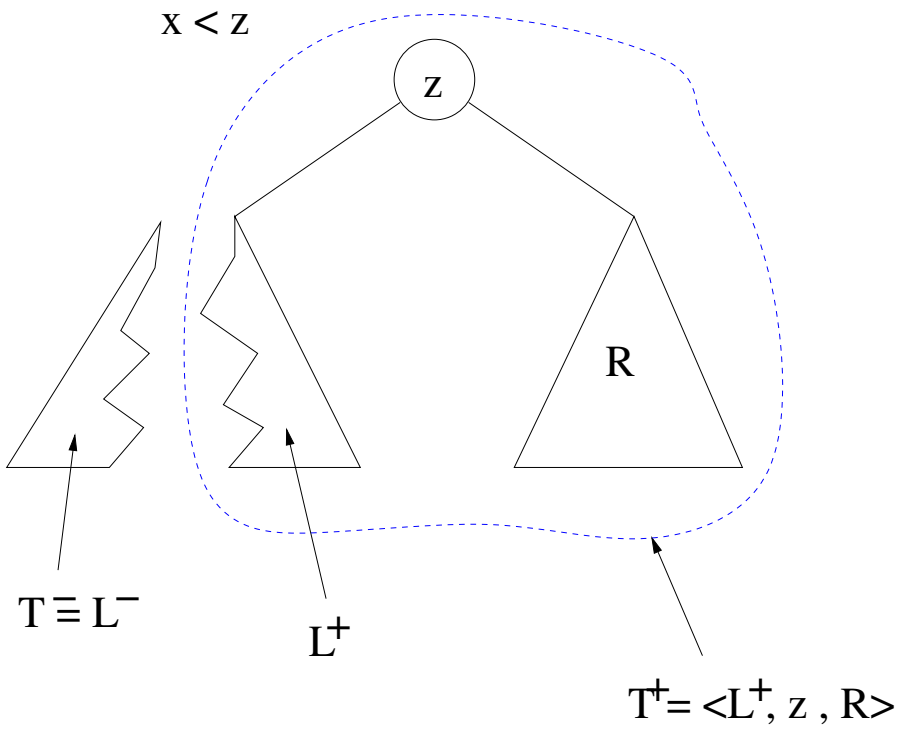
Splitting a RBST

To split a RBST T around x, we need just to follow the path from the root of T to the leaf where x falls



Splitting a RBST

To split a RBST T around x, we need just to follow the path from the root of T to the leaf where x falls



Splitting a RBST & Insertion at Root

```
▷ Pre: k is not present in T
procedure SPLIT(T, k, T⁻, T⁺)
    if T = null then
        T⁻ := null; T⁺ := null; return
    if k < T → key then
        SPLIT(T → left, k, L⁻, L⁺)
        T → left := L⁺
        Update T → size
        T⁻ := L⁻
        T⁺ := T
    else
        ▷ "Symmetric" code for k > T → key
```

Splitting a RBST

Lemma

Let T^- and T^+ be the BSTs produced by SPLIT(T, x)*. If T is a random BST containing the set of keys K, then T^- and T^+ are independent random BSTs containing the sets of keys $K^- = \{y \in T \mid y < x\}$ and $K^+ = \{y \in T \mid y > x\}$, respectively.*

Insertion in RBSTs

Theorem

If T is a random BST that contains the set of keys K and x is any key not in K, then INSERT(T, x) *produces a random BST containing the set of keys $K \cup \{x\}$.*

The Cost of Insertions

- The cost of the insertion at root (measured # of visited nodes) is exactly the same as the cost of the standard insertion
- For a random(ized) BST the cost of insertion is the depth of a random leaf in a random binary searh tree:

$$\mathbb{E}[I_n] = 2 \ln n + \mathcal{O}(1)$$

The Cost of Insertions

- The recurrence of $\mathbb{E}[I_n]$:

$$\mathbb{E}[I_n] = 1 + \frac{1}{n} \sum_{1 \leq j \leq n} \frac{j}{n+1} \mathbb{E}\left[I_{j-1}\right] + \frac{n-j+1}{n+1} \mathbb{E}\left[I_{n-j}\right]$$

- To solve this recurrence the Continuous Master Theorem (Roura, 20021) [*stay tuned!*] comes handy
- We need to produce $\mathcal{O}(\log n)$ random numbers on average to insert an item

The Cost of Insertions

- The recurrence of $\mathbb{E}[I_n]$:

$$\mathbb{E}[I_n] = 1 + \frac{1}{n} \sum_{1 \leq j \leq n} \frac{j}{n+1} \mathbb{E}\left[I_{j-1}\right] + \frac{n-j+1}{n+1} \mathbb{E}\left[I_{n-j}\right]$$

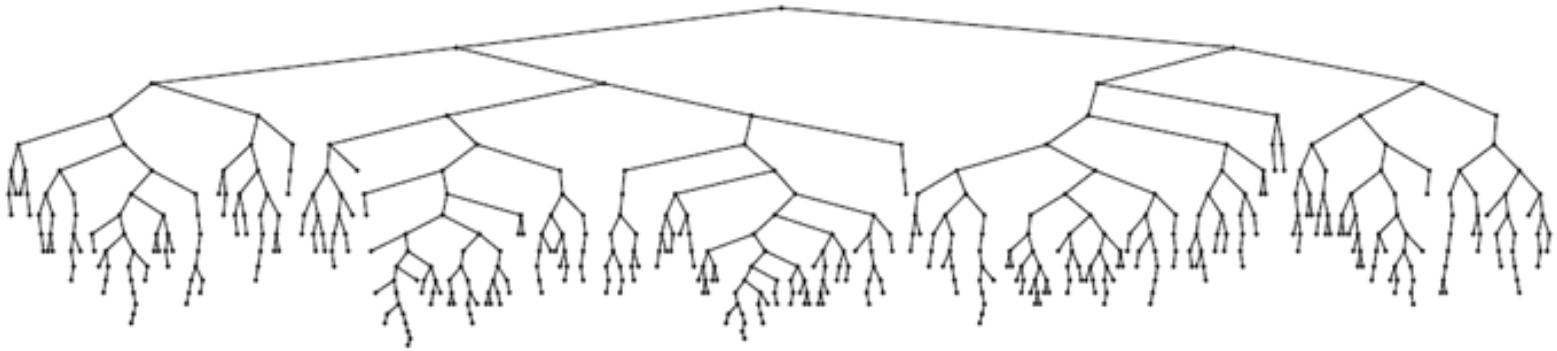
- To solve this recurrence the Continuous Master Theorem (Roura, 20021) [*stay tuned!*] comes handy
- We need to produce $\mathcal{O}(\log n)$ random numbers on average to insert an item

The Cost of Insertions

- The recurrence of $\mathbb{E}[I_n]$:

$$\mathbb{E}[I_n] = 1 + \frac{1}{n} \sum_{1 \leq j \leq n} \frac{j}{n+1} \mathbb{E}\left[I_{j-1}\right] + \frac{n-j+1}{n+1} \mathbb{E}\left[I_{n-j}\right]$$

- To solve this recurrence the Continuous Master Theorem (Roura, 20021) [*stay tuned!*] comes handy
- We need to produce $\mathcal{O}(\log n)$ random numbers on average to insert an item



RBST resulting from the insertion of 500 keys in ascending order

Source: R. Sedgewick, *Algorithms in C* (3rd edition), 1997

Deletions in RBSTs

- The fundamental problem is how to remove the root node of a BST, in particular, when both subtrees are not empty
- The original deletion algorithm by Hibbard was assumed to preserve randomness
- In 1975, G. Knott discovered that Hibbard's deletion preserves randomness of shape, but an insertion following a deletion would destroy randomness (Knott's paradox)

Deletions in RBSTs


J. Culberson


J.L. Eppinger


D.E. Knuth

- Several theoretical and experimental work aimed at understanding what was the effect of deletions, e.g.,
  - Jonassen & Knuth's *An Algorithm whose Analysis Isn't* (JCSS, 1978)
  - Knuth's *Deletions that Preserve Randomness* (IEEE Trans. Soft. Eng., 1977)
  - Eppinger's experiments (CACM, 1983)
  - Culberson's paper on deletions of the left spine (STOC, 1985)
- These studies showed that deletions degraded performance in the long run

Deletions in RBSTs

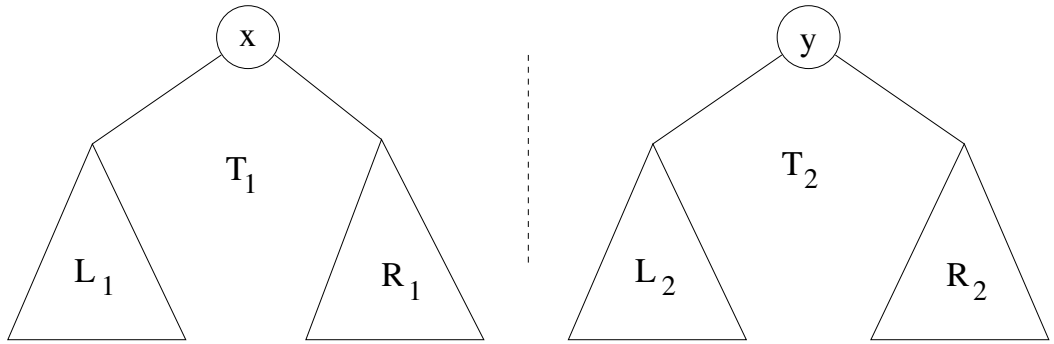
```
procedure DELETE(T, k)
    if T = □ then
        return T
    if k = T → key then
        return DELETE-ROOT(T)
    if x < T → key then
        T → left := DELETE(T → left, k)
    else
        T → right := DELETE(T → right, k)
    Update T → size
    return T
```

Deletions in RBSTs

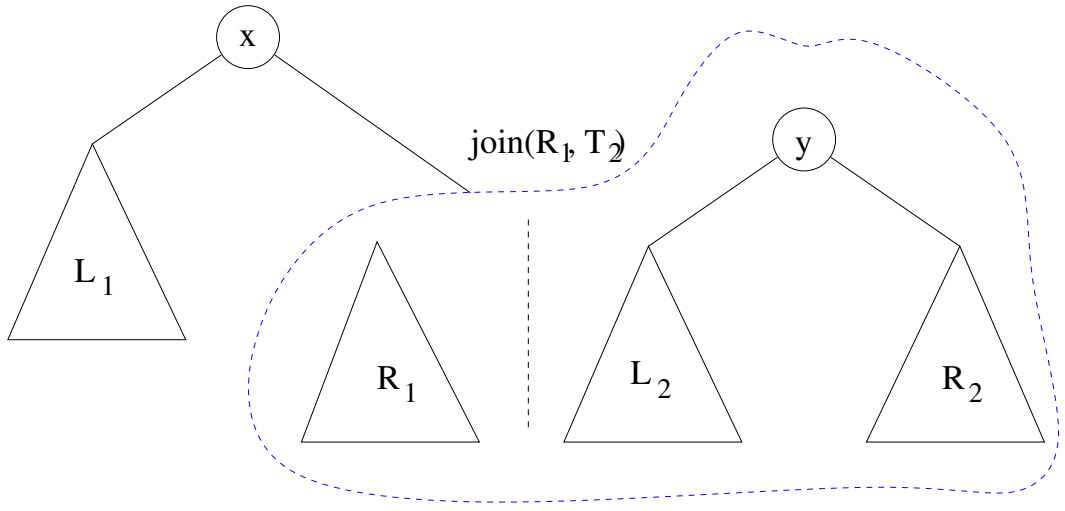
We delete the root using a procedure $\textsc{Join}(T_1, T_2)$. Given two BSTs such that for all $x \in T_1$ and all $y \in T_2$, $x \leq y$, it returns a new BST that contains all the keys in T_1 and T_2.

$$\textsc{Join}(\square, \square) = \square$$
$$\textsc{Join}(T, \square) = \textsc{Join}(\square, T) = T$$
$$\textsc{Join}(T_1, T_2) = ?, \qquad T_1 \neq \square, T_2 \neq \square$$

Joining two BSTs



Joining two BSTs



Joining two BSTs

- If we systematically choose the root of T_1 as the root of JOIN(T_1, T_2), or the other way around, we will introduce an undesirable bias
- Suppose both T_1 and T_2 are random. Let m and n denote their sizes. Then x is the root of T_1 with probability $1/m$ and y is the root of T_2 with probability $1/n$
- Choose x as the common root with probability $m/(m+n)$, choose y with probability $n/(m+n)$

$$\frac{1}{m} \times \frac{m}{m+n} = \frac{1}{m+n}$$
$$\frac{1}{n} \times \frac{n}{m+n} = \frac{1}{m+n}$$

Joining two RBSTs

Lemma

Let L and R be two independent random BSTs, such that the keys in L are strictly smaller than the keys in R. Let K_L and K_R denote the sets of keys in L and R, respectively. Then $T = \textsc{Join}(L, R)$ is a random BST that contains the set of keys $K = K_L \cup K_R$.

Joining two RBSTs

- The recursion for JOIN(T_1, T_2) traverses the rightmost branch (right spine) of T_1 and the leftmost branch (left spine) of T_2
- The trees to be joined are the left and right subtrees L and R of the ith item in a RBST of size n; then

$$\text{length of left spine of } L = \text{path length to } i\text{th leaf}$$

$$\text{length of right spine of } R = \text{path length to } (i+1)\text{th leaf}$$

- The cost of the joining phase is the sum of the path lengths to the leaves minus twice the depth of the ith item; the expected cost follows from well-known results

$$\left(2 - \frac{1}{i} - \frac{1}{n+1-i}\right) = \mathcal{O}(1)$$

Deletions in RBSTs

Theorem

If T is a random BST that contains the set of keys K, then DELETE(T, x) *produces a random BST containing the set of keys $K \setminus \{x\}$.*

Deletions in RBSTs

Theorem

If T is a random BST that contains the set of keys K, then DELETE(T, x) *produces a random BST containing the set of keys $K \setminus \{x\}$.*

Corollary

The result of any arbitary sequence of insertions and deletions, starting from an initially empty tree is always a random BST.

Additional remarks

- Arbitrary insertions and deletions yield always random BSTs
- A deletion algorithm for BSTs that preserved randomness was a long standing open problem (10-15 yr)
- Properties of random BSTs have been investigated in depth and for a long time
- Treaps only need to generate a single random number per node (with $\mathcal{O}(\log n)$ bits)
- RBSTs need $\mathcal{O}(\log n)$ calls to the random generator per insertion, and $\mathcal{O}(1)$ calls per deletion (on average)

Additional remarks

- Storing subtree sizes for balancing is more useful: they can be used to implement search and deletion by rank, e.g., find the ith smallest element in the tree
- Other operations, e.g., union and intersection are also efficiently supported by RBSTs
- Similar ideas have been used to randomize other search trees, namely, K-dimensional binary search trees (Duch and Martínez, 1998) and quadtrees (Duch, 1999) (stay tuned!)

To learn more

[1] C. Martínez and S. Roura.
Randomized binary search trees.
J. Assoc. Comput. Mach., 45(2):288–323, 1998.

[2] R. Seidel and C. Aragon.
Randomized search trees.
Algorithmica, 16:464–497, 1996.

To learn more (2)

[3] J. L. Eppinger.
An empirical study of insertion and deletion in binary search trees.
Comm. of the ACM, 26(9):663—669, 1983.

[4] W. Panny.
Deletions in random binary search trees: A story of errors.
J. Statistical Planning and Inference, 140(8):2335–2345, 2010.

[5] H. M. Mahmoud.
Evolution of Random Search Trees.
Wiley Interscience, 1992.

Why Multidimensional?

Nowadays data:

- Points, lines,
- rivers, maps, cities, roads,
- hyperplanes, cubes, hypercubes,
- mp3, mp4 and mp5 files,
- jpeg files, pixels,
- . . . ,

Used in applications such as:

- database design, geographic information systems (GIS),
- computer graphics, computer vision, computational geometry, image processing,
- pattern recognition,
- very large scale integration (VLSI) design,
- . . .

This course. . .

- Data: File of K-dimensional points, K-tuples of the form:

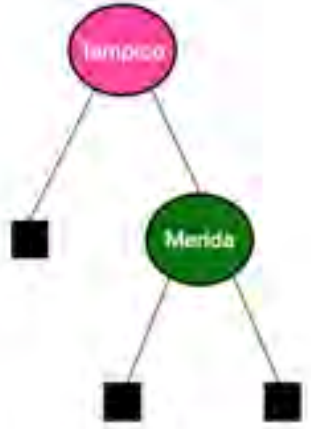
$$x = (x_0, x_1, \ldots, x_{K-1})$$

- Retrieval: associative queries that involve more than one of the K dimensions
- Data structures: two generalisations of RBSTs
  - Randomized K-d trees and
  - Randomized quad trees

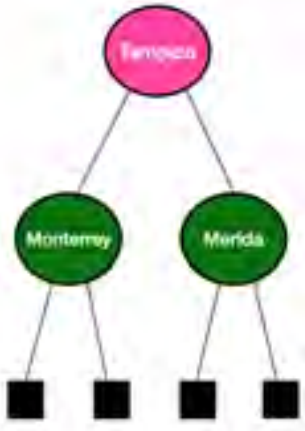
Standard K-d trees

Standard K-d trees

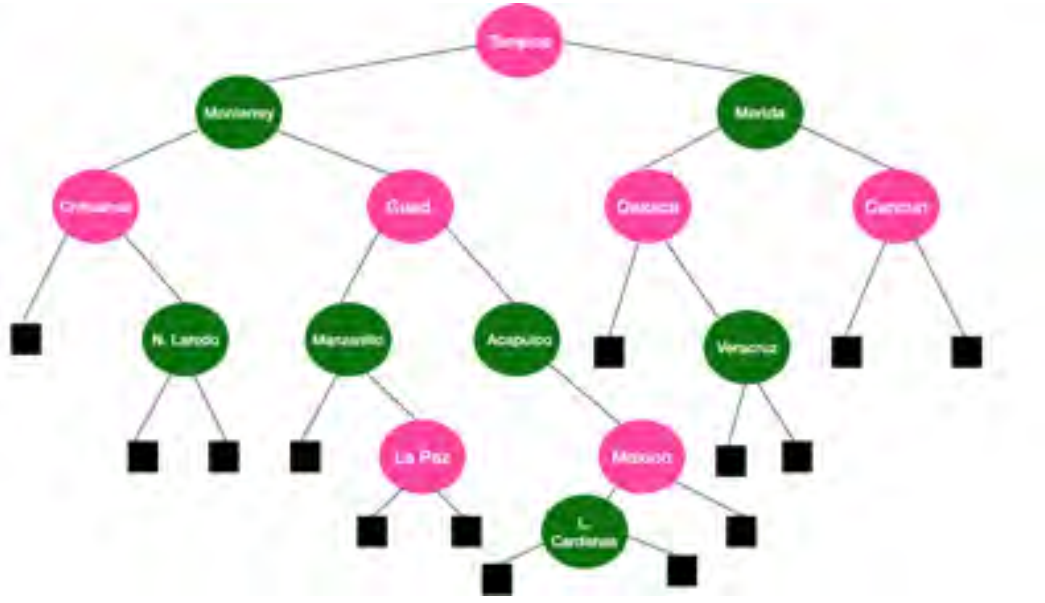
Standard *K*-d trees



Standard K-d trees



Standard *K*-d trees



Standard K-d Trees

Definition (Bentley75)

A standard K-d tree T of size $n \geq 0$ is a binary tree that stores a set of n K-dimensional points, such that

- it is empty when $n = 0$, or
- its root stores a key x and a discriminant $j =$ level of the root $\mod K$, $0 \leq j < K$, and the remaining $n - 1$ records are stored in the left and right subtrees of T, say L and R, in such a way that both L and R are K-d trees; furthermore, for any key $u \in L$, it holds that $u_j < x_j$, and for any key $v \in R$, it holds that $x_j < v_j$.

2-d Quad Trees

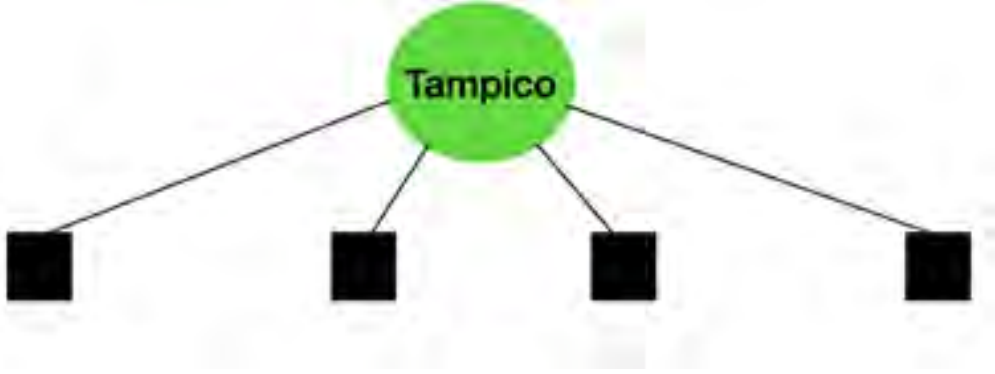

Definition (Bentley & Finkel, 1974)

A quad tree for a file of 2-dimensional records, is a quaternary tree in which:

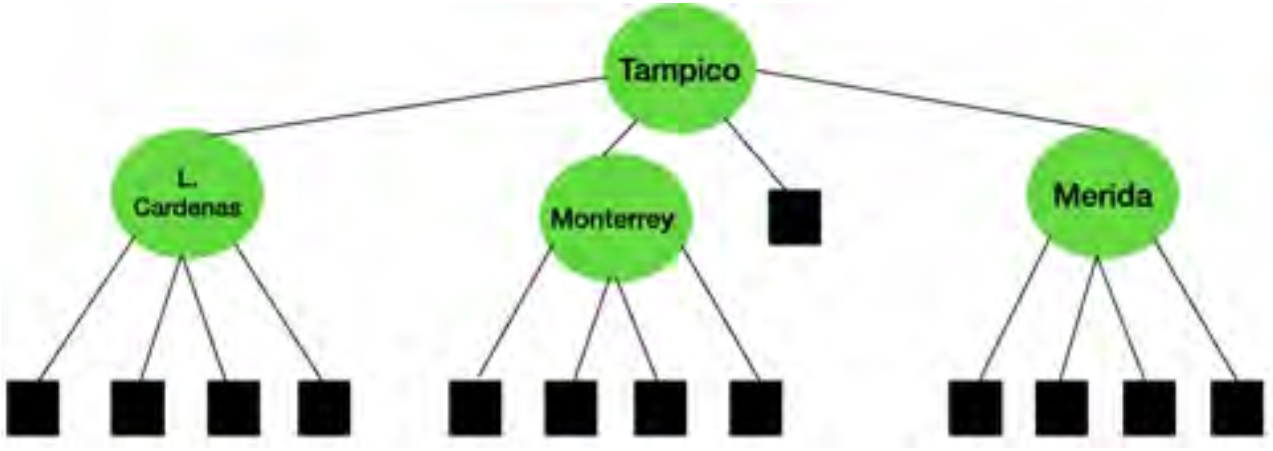
1. Each node contains a 2-dimensional key and has associated four subtrees corresponding to the quadrants *NW*, *NE*, *SE* and *SW*.
2. For every node with key x the following invariant is true: any record in the *NW* subtree with key y satisfies $y_1 < x_1$ and $y_2 \geq x_2$; any record in the *NE* subtree with key y satisfies $y_1 \geq x_1$ and $y_2 \geq x_2$; any record in the *SE* subtree with key y satisfies $y_1 \geq x_1$ and $y_2 < x_2$; and, any record in the *SW* subtree with key y satisfies $y_1 < x_1$ and $y_2 < x_2$.

2-d Quad Trees

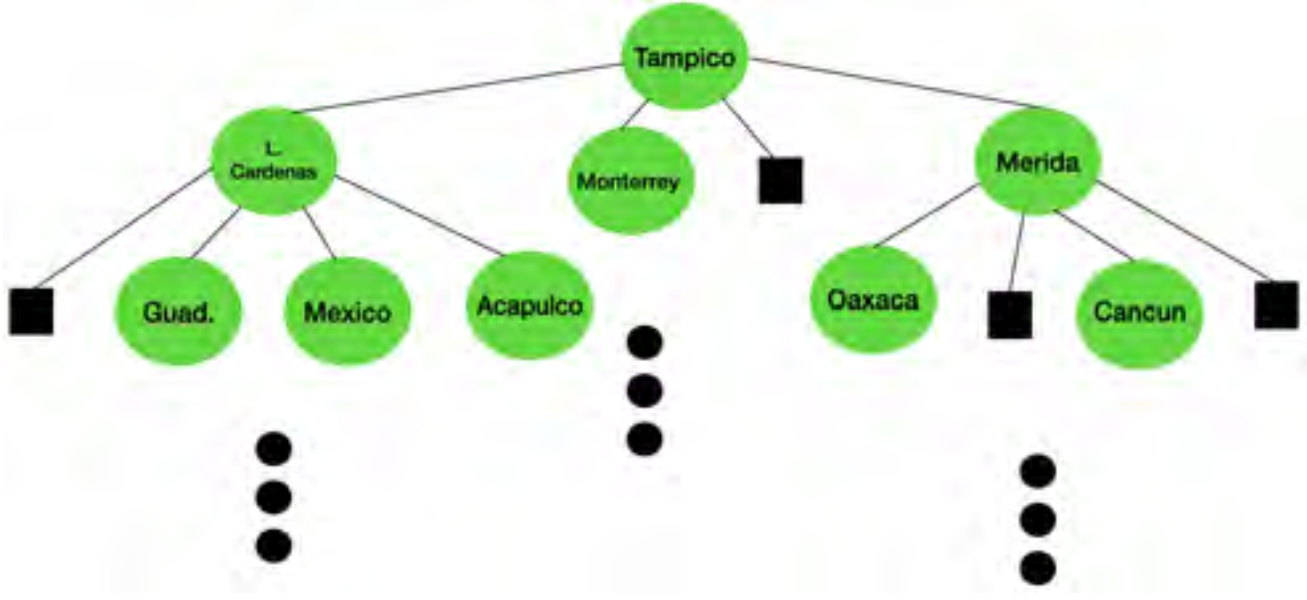
2-d Quad Trees



2-d Quad Trees



2-d Quad Trees



Quad Trees, $K \geq 2$

Definition (Bentley & Finkel, 1974)

A quad tree T of size $n \geq 0$ stores a set of n K-dimensional records. The quad tree T is a 2^K-ary tree such that

- either it is empty and $n = 0$, or
- its root stores a record with key x and has 2^K subtrees, each one associated to a K-bitstring $w = w_0 w_1 \ldots w_{K-1} \in \{0,1\}^K$, and the remaining $n-1$ records are stored in one of these subtrees, let's say T_w, in such a way that $\forall w \in \{0,1\}^K$: T_w is a quad tree, and for any key $y \in T_w$, it holds that $y_j < x_j$ if $w_j = 0$ and $y_j > x_j$ otherwise, $0 \leq j < K$.

Randomized K-d trees and Quad trees

- Goal Dynamic tree that supports *all* operations with good expected performance (less than linear) and using $\Theta(nK)$ memory space.
- Problems
  - The trees can be very unbalanced.
  - The rule to assign discriminants in K-d trees complicates updates.
  - Deletion of nodes into two-dimensional quad trees is complicated.
    - Finkel and Bentley (1974) suggested that all nodes of the tree rooted at the deleted node must be reinserted, but this is usually expensive.
    - A more efficient process developed by Sammet (1980) allows to reduce the number of nodes to be reinserted, although it is still an expensive and not straightforward process.
- Idea: insertions and deletions similar to RBSTs.

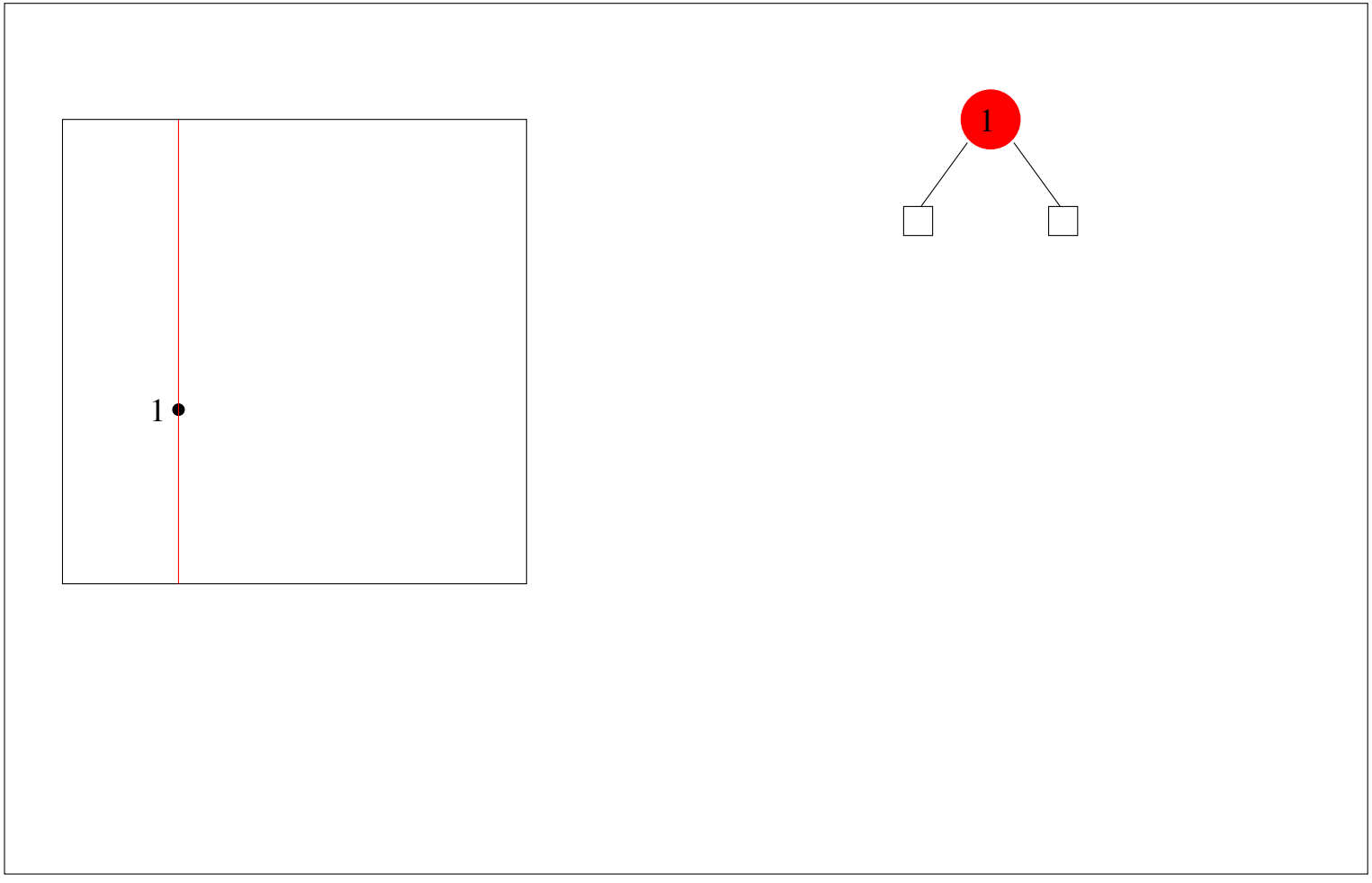
Relaxed K-d trees: first level of randomization

A relaxed K-d tree (Duch, Estivill-Castro, Martínez, 1998) for a set of K-dimensional keys is a binary tree in which:

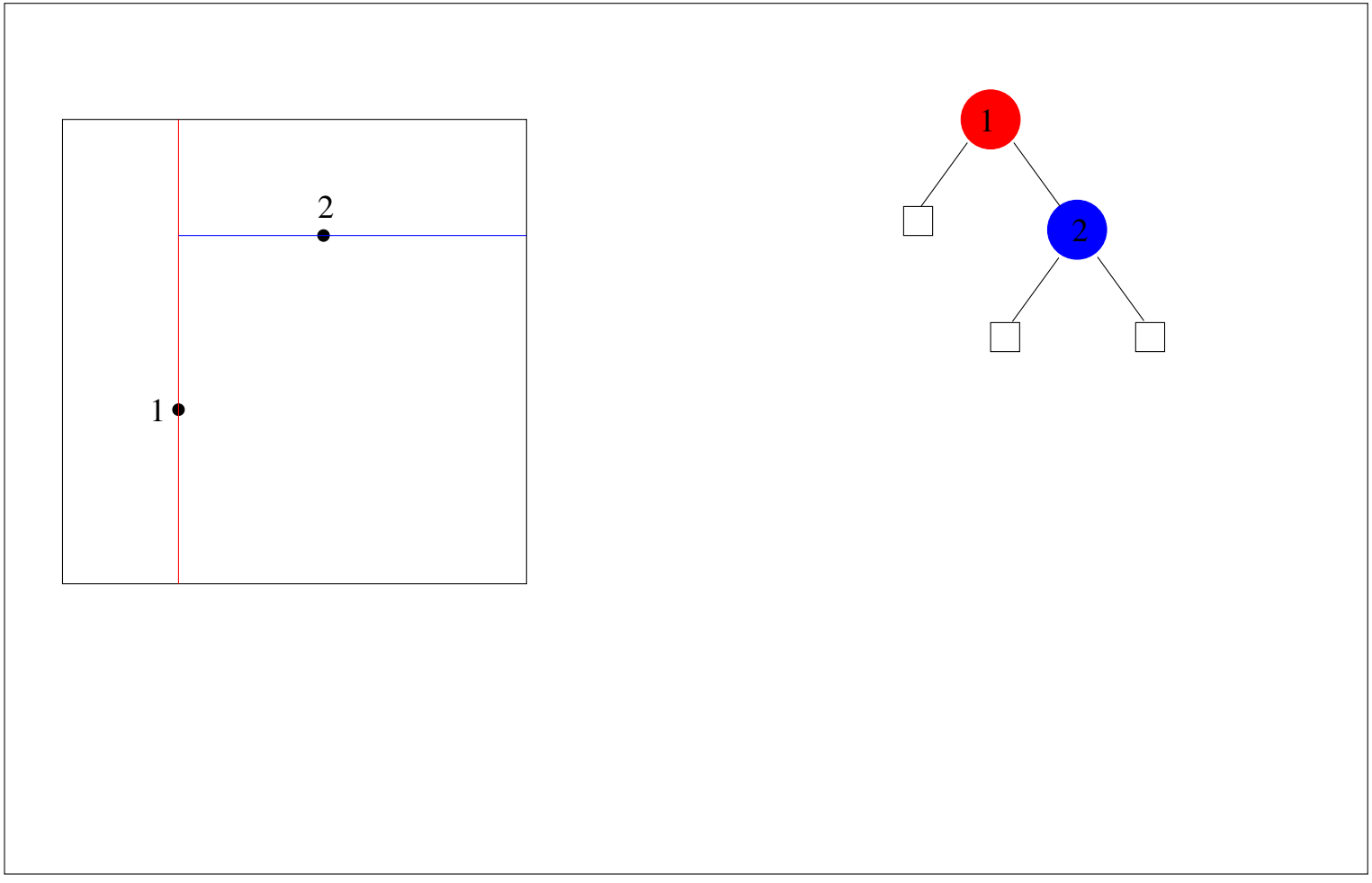
1. Each node contains a K-dimensional record and has associated an arbitrary discriminant $j \in \{0, 1, \ldots, K-1\}$.
2. For every node with key x and discriminant j, the following invariant is true: any record in the right subtree with key y satisfies $y_j < x_j$ and any record in the left subtree with key y satisfies $y_j \geq x_j$.

Relaxed K-d trees: first level of randomization

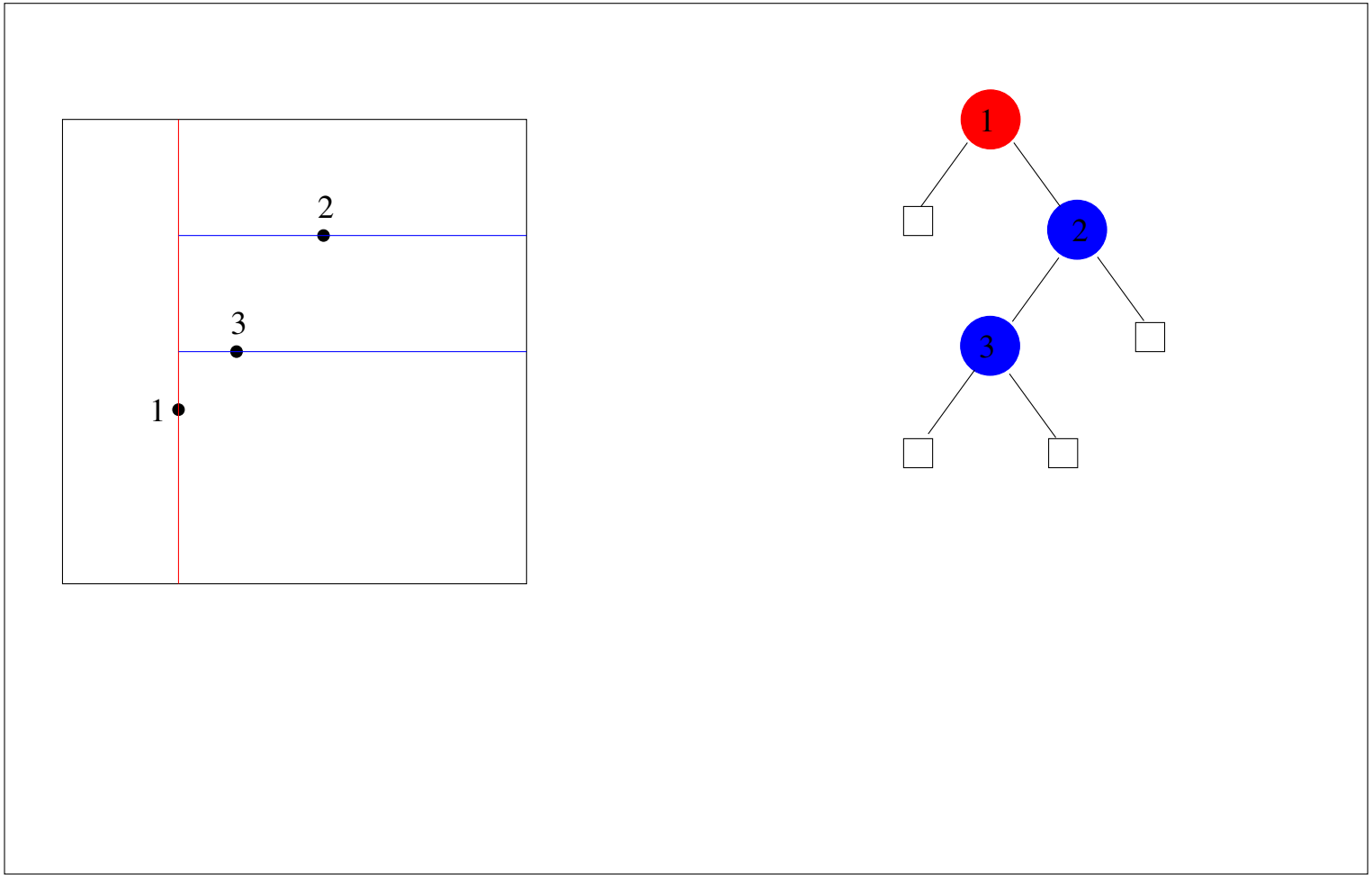
Relaxed K-d trees: first level of randomization



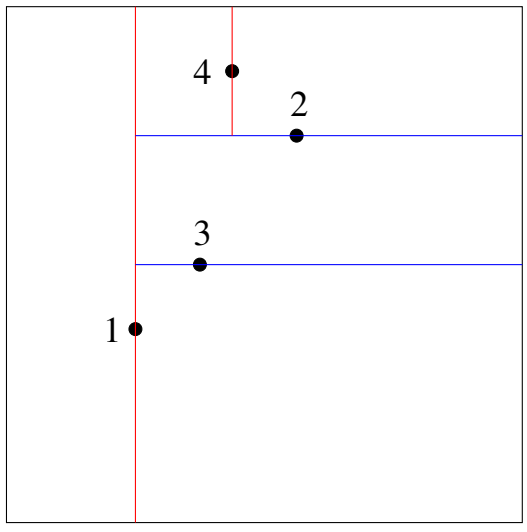
Relaxed K-d trees: first level of randomization

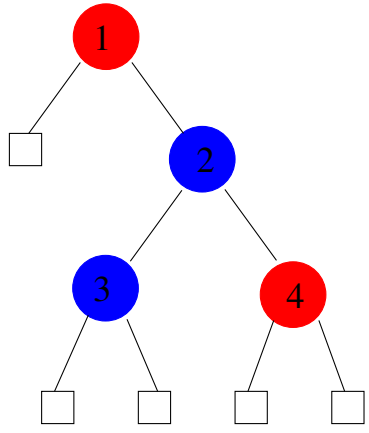


Relaxed *K*-d trees: first level of randomization

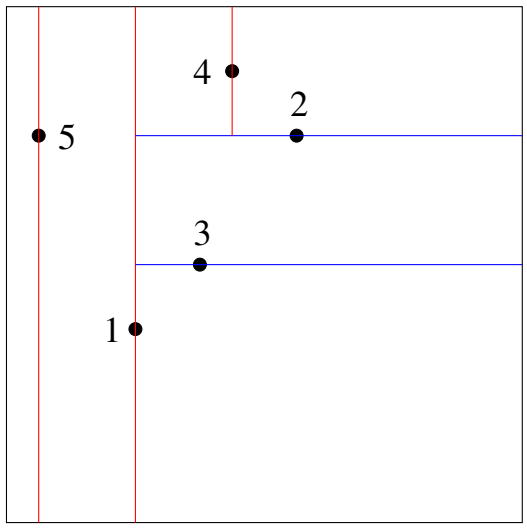


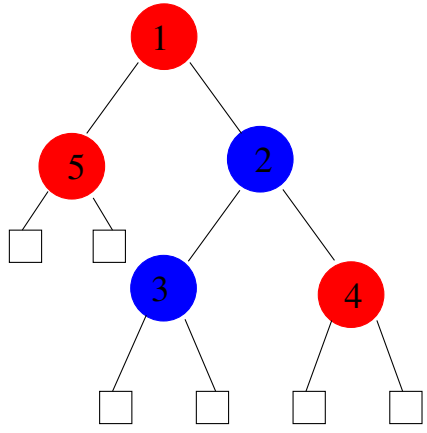
Relaxed K-d trees: first level of randomization





Relaxed K-d trees: first level of randomization

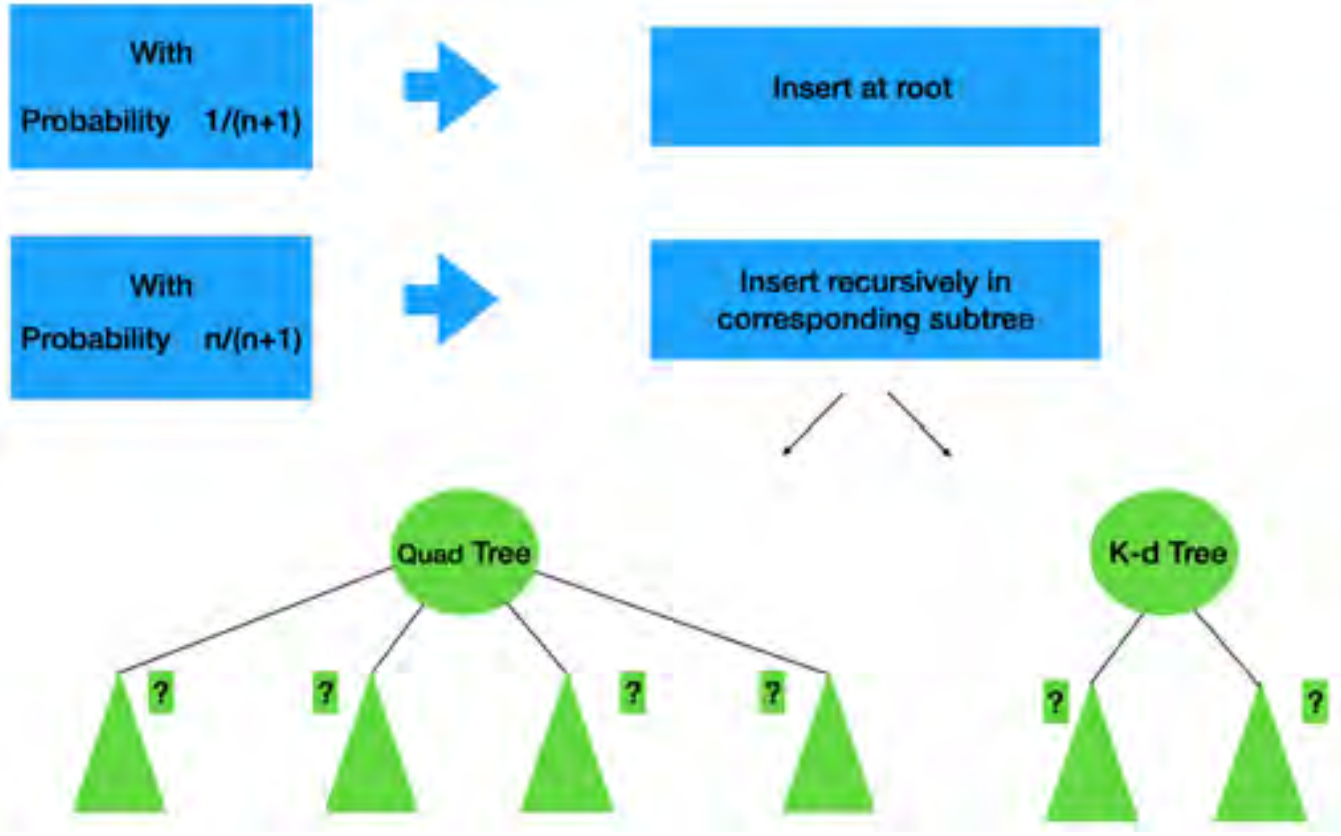




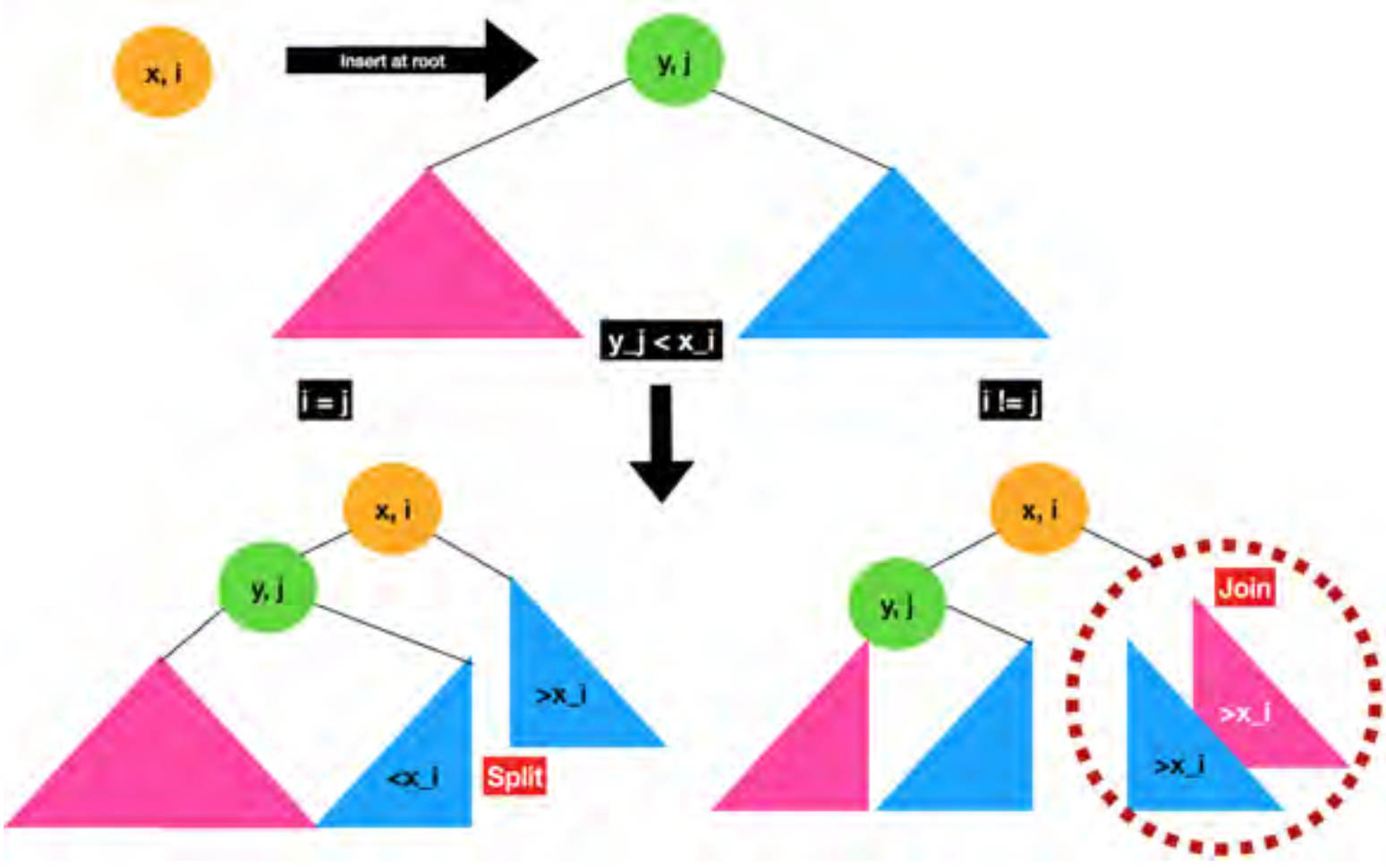
Insertion

INPUT: Random Relaxed K-d tree (or random quad tree) of size n keeping the set of K-dimensional keys S, K-dimensional point x.

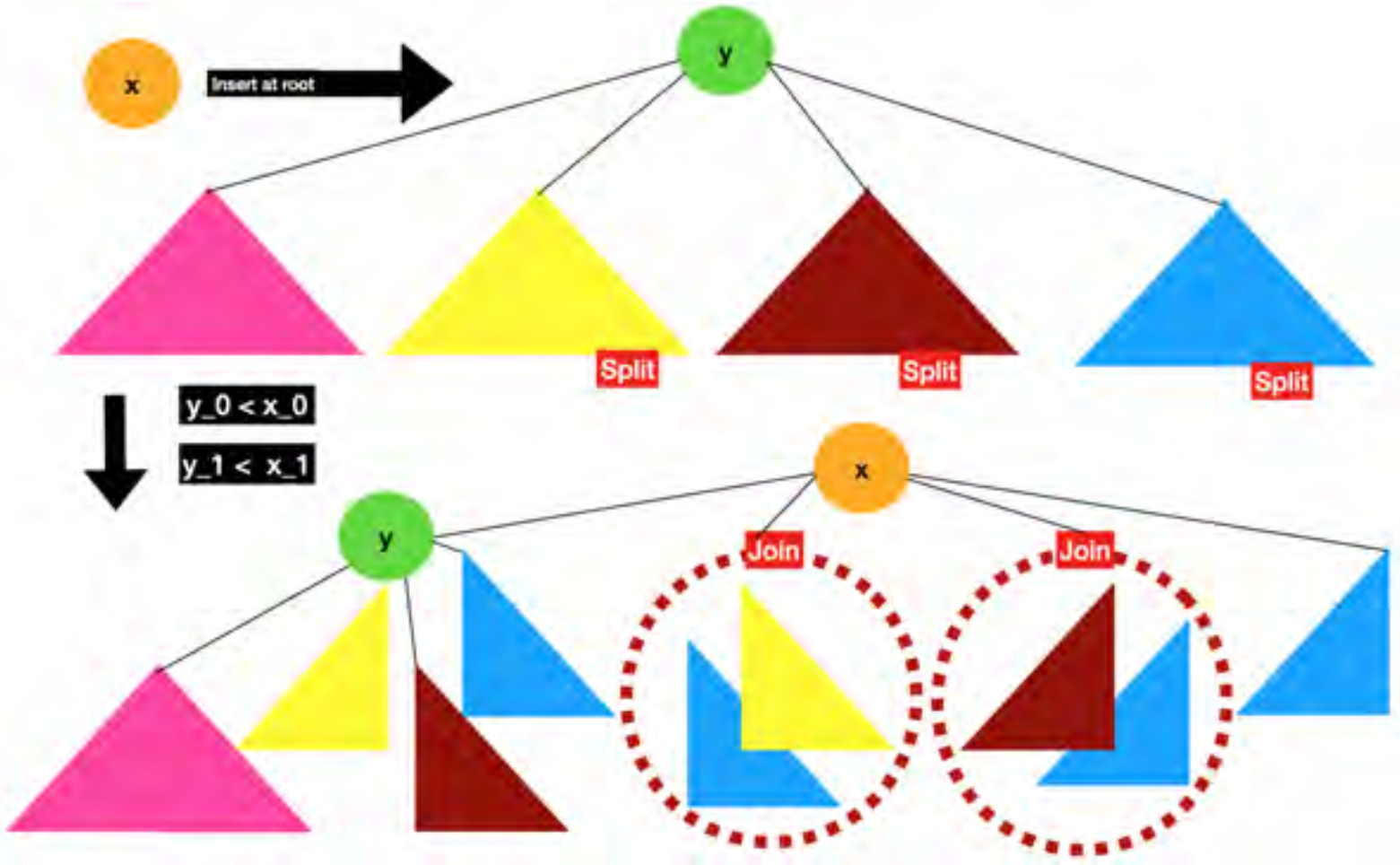
OUTPUT: Random Relaxed K-d tree (or random quad tree) of size $n+1$ keeping the set of K-dimensional keys $S \cup \{x\}$.



Insertion at the root: Randomized relaxed *K*-d trees



Insertion at the root: Randomized quad trees



Updates in Randomized relaxed K-d trees and quad trees

Theorem

If T is a randomized relaxed K-d tree (or a randomized quad tree) that contains the set of keys S and x is any key not in S, then INSERT(T, x) *produces a randomized relaxed K-d tree (or a randomized quad tree respectively) containing the set of keys $S \cup \{x\}$.*

Theorem

If T is a randomized relaxed K-d tree (or a randomized quad tree) that contains the set of keys S and x is any in T, then DELETE(T, x) *produces a randomized relaxed K-d tree (or a randomized quad tree respectively) containing the set of keys $S - \{x\}$.*

The Cost of Updates

- K-d trees:
  - Case $K = 2$: The cost of the updates (measured # of visited nodes) is equivalent to the expected height of the tree which is the same as for BSTs (Duch & Martínez, 2009).
  - Case $K > 2$: Unfortunately the cost of the updates is no longer logarithmic (Duch & Martínez, 2009).
- Quad trees:
  - Not analyzed. Conjecture: similar to K-d trees.
  - For a random(ized) quad tree of size n, the expected height H_n is asymptotically $(c/K) \log n$, where $c = 4.31107\ldots$ (Devroye, 1987). It has been shown independently by Devroye and Laforest (1990) and Flajolet et al.(1993) that the expected cost of a random search in a random K-dimensional quad tree of size $n - 1$ is $(2/K) \log n$.

Randomization: what for? Associative retrieval

Multidimensional data structures must support:

- Usual insertions, deletions, (exact) queries
- Associative queries such as:

  Partial Match Queries: Find the data points that match some specified coordinates of a given query point q.

  Orthogonal Range Queries: Find the data points that fall within a given hyper rectangle Q (specified by K ranges).

  Nearest Neighbor Queries: Find the closest data point to some given query point q (under a predefined distance).

Associative Queries

Random Partial Match Queries

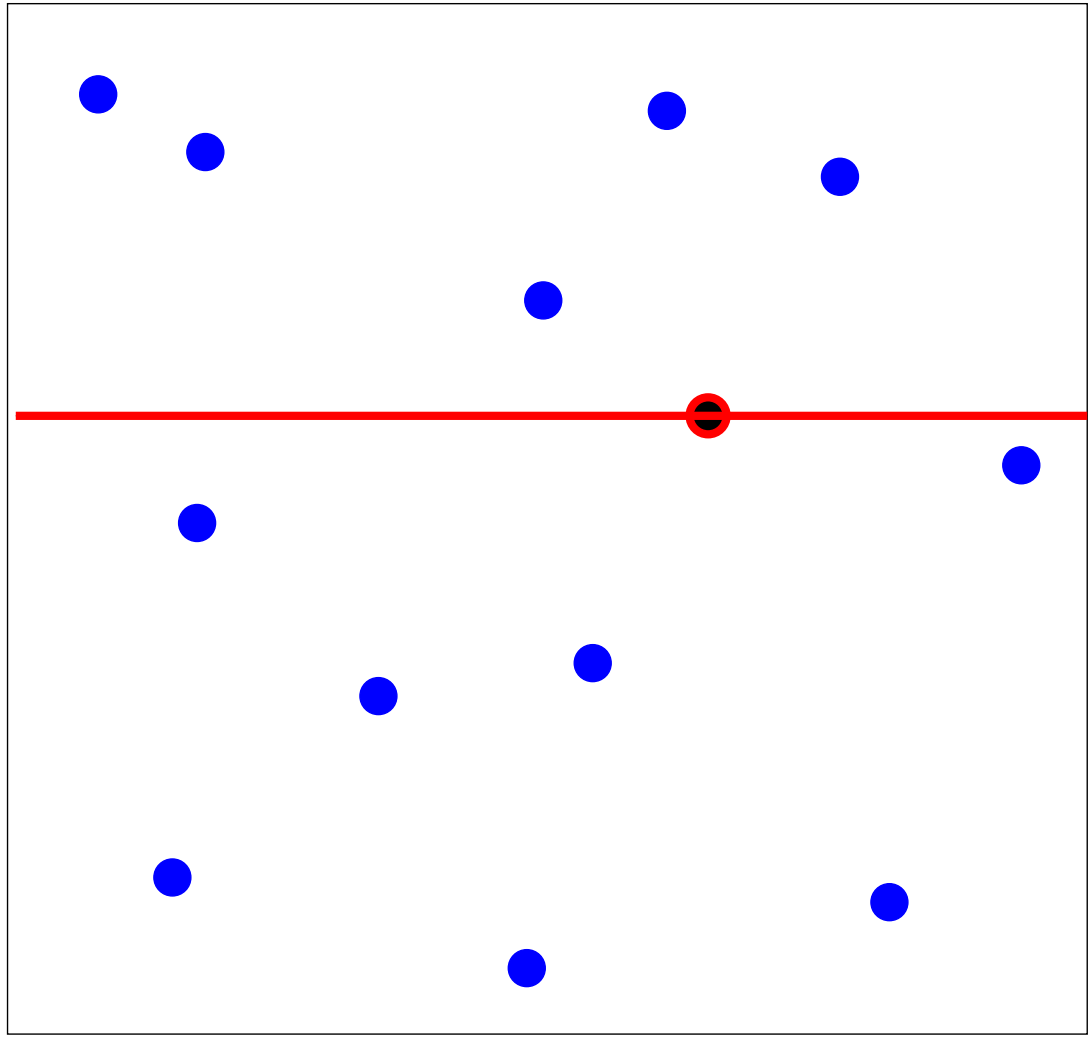
Definition

Given a file F of n K-dimensional records and a query $q = (q_0, q_1, \ldots, q_{K-1})$ where each q_i is either a value in D_i (it is specified) or $*$ (it is unspecified), a *partial match query* returns the subset of records x in F whose attributes coincide with the specified attributes of q. This is,

$$\{x \in F \mid q_i = * \text{ or } q_i = x_i, \forall i \in \{0, \ldots, K-1\}\}.$$

Example of Partial Match Queries

Query: $q = (*, q_2)$ or $q = (q_1, q_2)$ with specification pattern: 01



Partial Match Algorithm in Relaxed K-d Trees

Partial match search in relaxed K-d trees works as follows:

- At each node of the tree we verify if it satisfies the query and we examine its discriminant.
- If the discriminant is specified in the query then the algorithm recursively follows in the appropriate subtree depending on the result of the comparison between the key and the query.
- Otherwise the algorithm recursively follows the two subtrees of the node.

Random Partial Match Algoritm

```
procedure RANDOM_PM(T, u, q)
    ▷ T: tree, u: specification pattern, q: query
    if T ≠ □ then  ▷ nothing to do if T were empty
        i = T → discr
        if MATCH(T → key, u, q) then
            REPORT(T → key)
        if u[i] = 1 then  ▷ Coordinate i specified
            if q < T → key then
                RANDOM_PM(T → left, u, q)
            else
                RANDOM_PM(T → right, u, q)
        else  ▷ Coordinate i not specified
            RANDOM_PM(T → left, u, q)
            RANDOM_PM(T → right, u, q)
```

The Recurrence of Partial Match Searches

Following the random model at each node:

- With probability $\frac{s}{K}$ the discriminant will be specified in the query and the algorithm will follow one of the subtrees.
- With probability $\frac{K-s}{K}$ the algorithm will follow the two subtrees.
- Hence, the cost $M(T)$ of a Partial Match Search in a relaxed K-d tree T of size n with left subtree L of size ℓ and right subtree R is:

$$M(T \mid |L| = \ell) = 1 + \frac{s}{K}\left(\frac{\ell+1}{n+1}M(L) + \frac{n-\ell}{n+1}M(R)\right) + \frac{K-s}{K}(M(L) + M(R)).$$

The Expected Cost of Partial Match

Theorem (Duch et al., 1998)

The expected cost M_n (measured as the number of comparisons) of a PM query with s out of K coordinates specified in a random(ized) relaxed K-d tree of size n is

$$M_n = \beta n^{\alpha} + \mathcal{O}(1),$$

where

$$\begin{aligned} \alpha &= \alpha(s/K) = 1 - \frac{s}{K} + \phi(s/K) \\ \beta &= \beta(s/K) = \frac{\Gamma(2\alpha+1)}{(1-s/K)(\alpha+1)\Gamma^3(\alpha+1)} \end{aligned}$$

with
$\phi(x) = \sqrt{9-8x}/2 + x - 3/2$ *and* $\Gamma(x)$ *the Euler's Gamma function.*

Solving the Recurrence of Partial Match Searches

In order to get the cost of partial match searches we follow the next steps:

- Take averages for all possible values of ℓ in the cost equation.
- Simplify by taking symmetries in the resulting recurrence.
- Translate the recurrence into a hypergeometric differential equation on the corresponding generating function.
- Solve the differential equation and obtain the generating function of the average cost of partial match.
- Use transfer lemmas to extract the coefficients of the average cost of partial match.

Comparison with standard K-d trees

- Randomized relaxed K-d trees are fully dynamic.
- The expected performance of randomized relaxed K-d trees does not depend on any supposition regarding the distribution of the input.
- The α coefficient for standard K-d trees is slightly smaller, but the analysis is more complicated since it involves the solution of a system of differential equations, one for each level of the tree and depending on the query pattern (Flajolet & Puech, 1986).
- The β coefficient for standard K-d trees is dependent on the specification pattern (Flajolet & Puech, 1986; Chern & Hwang, 2006).

To learn more

[1] J. L. Bentley.
Multidimensional binary search trees used for associative retrieval.
Communications of the ACM, 18(9):509–517, 1975.

[2] J. L. Bentley and R. A. Finkel.
Quad trees: A data structure for retrieval on composite keys.
Acta Informatica, 4:1–9, 1974.

[3] H. H. Chern and H. K. Hwang.
Partial match queries in random k-d trees.
SIAM J. on Computing, 35(6):1440–1466, 2006.

[4] H. H. Chern and H. K. Hwang.
Partial match queries in random quad trees.
SIAM Journal on Computing, 32(4):904–915, 2003.

To learn more (2)

[5] L. Devroye.
Branching processes in the analysis of the height of trees.
Acta Informatica, 24:277–298, 1987.

[6] L. Devroye and L. Laforest.
An analysis of random d-dimensional quadtrees.
SIAM Journal on Computing, 19(5):821–832, 1990.

[7] A. Duch.
Randomized insertion and deletion in point quad trees.
In *Int. Symposium on Algorithms and Computation (ISAAC)*, LNCS. Springer–Verlag, 2004.

[8] A. Duch, V. Estivill-Castro, and C. Martínez.
Randomized K-dimensional binary search trees.
In K.-Y. Chwa and O. H. Ibarra, editors, *Int. Symposium on Algorithms and Computation (ISAAC'98)*, volume 1533 of *LNCS*, pages 199–208. Springer-Verlag, 1998.

To learn more (3)

[9] A. Duch and C. Martínez.
On the average performance of orthogonal range search in multidimensional data structures.
Journal of Algorithms, 44(1):226–245, 2002.

[10] A. Duch and C. Martínez.
Updating relaxed k-d trees.
ACM Transactions on Algorithms (TALG), 6(1):1–24, 2009.

[11] Ph. Flajolet, G. Gonnet, C. Puech, and J. M. Robson.
Analytic variations on quad trees.
Algorithmica, 10:473–500, 1993.

[12] Ph. Flajolet and C. Puech.
Partial match retrieval of multidimensional data.
Journal of the ACM, 33(2):371–407, 1986.

To learn more (4)

[13] C. Martínez, A. Panholzer, and H. Prodinger.
Partial match queries in relaxed multidimensional search trees.
Algorithmica, 29(1–2):181–204, 2001.

[14] R. Neininger.
Asymptotic distributions for partial match queries in K-d trees.
Random Structures and Algorithms, 17(3–4):403–4027, 2000.

[15] R. L. Rivest.
Partial-match retrieval algorithms.
SIAM Journal on Computing, 5(1):19–50, 1976.

[16] H. Samet.
Deletion in two-dimensional quad-trees.
Communications of the ACM, 23(12):703–710, 1980.

Bloom filters

A Bloom Filter is a probabilistic data structure representing a set of items; it supports:

- Addition of items: $F := F \cup \{x\}$
- Fast lookup: $x \in F$?

Bloom filters do require very little memory and are specially well suited for unsuccessful search (when $x \notin F$)

Bloom filters

- The price to pay for the reduced memory consumption and very fast lookup is the non-null probability of false positives.
- If $x \in F$ then a lookup in the filter will always return true; but if $x \notin F$ then there is some probability that we get a positive answer from the filter.
- In other words, if the filter says $x \notin F$ we are sure that's the case, but if the filter says $x \in F$ there is some probability that this is an error.

Bloom filters

Bloom filters are the most basic example of the so-called Approximate Membership Query Filters (AMQ filters) and support the following operations:

1. $F := \textsc{CreateBF}(N_{\max}, fp)$: creates an empty Bloom filter F that might store up to $N_{\max}$ items, and sets an upper bound fp on the *false positive rate* allowed
2. $F.\textsc{Insert}(x)$: add item x to filter F
3. $F.\textsc{Lookup}(x)$: returns whether x belongs to the filter F or not
   - if the answer is **true**, it might be wrong with probability $\leq fp$
   - if the answer is **false**, then $x \notin F$ for sure

Implementing Bloom filters

To represent a Bloom filter for a subset of items drawn from the domain $\mathcal{U}$ we will use:

1. A bitvector A of size M
2. A set of k pairwise independent hash functions $\{h_1, \ldots, h_k\}$, each $h_i : \mathcal{U} \to \{0, \ldots, M-1\}$

The values of M and k are carefully chosen as a function of $N_{\max}$ and fp

Implementing Bloom filters

procedure CREATEBF(N_{max}, fp)
  $M := \ldots$; $k := \ldots$
  A : bitvector[0..$M-1$]
  for $i := 0$ **to** $M-1$ **do** $A[i] := 0$
  for $j := 1$ **to** k **do** $h_i :=$ a random hash function

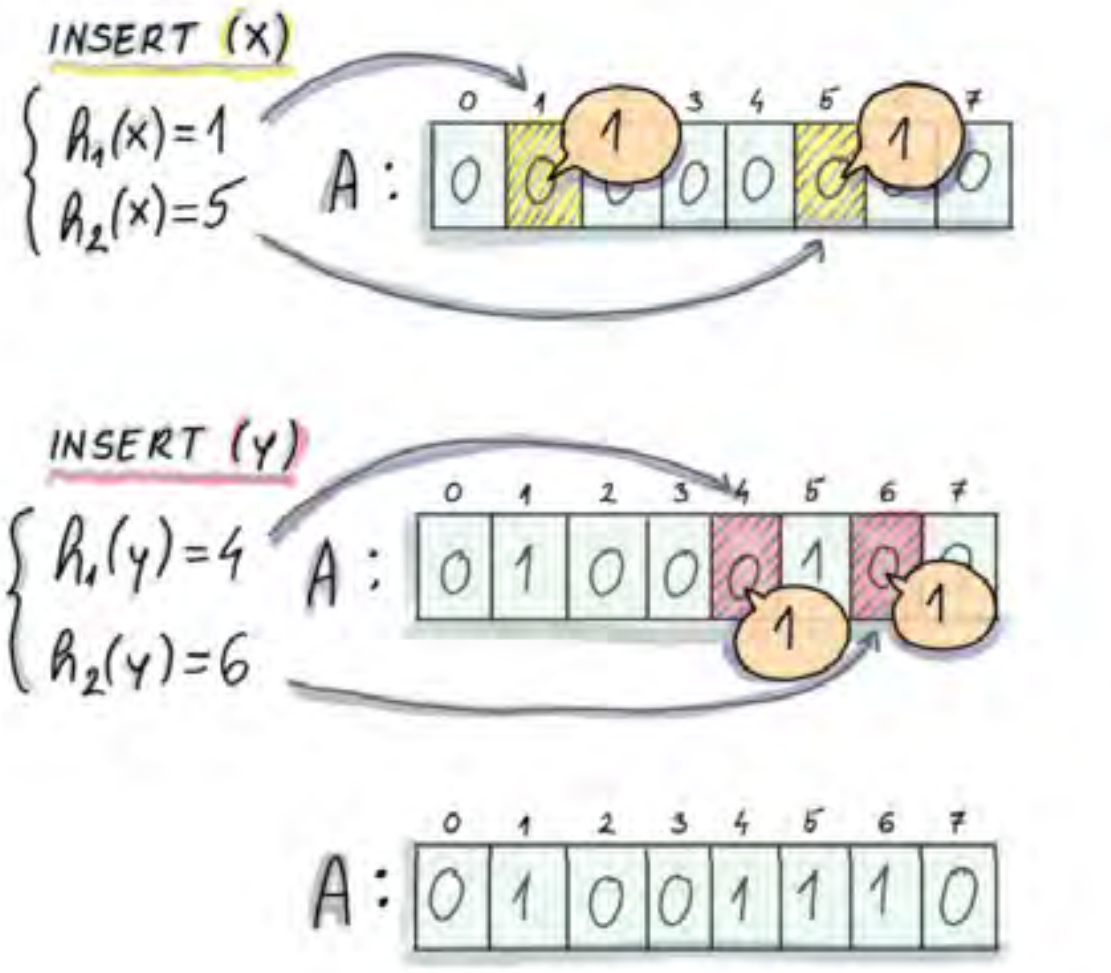
The k independent hash functions can be choosen from a universal class of hash functions (later in this course)

Insertion & lookup

```
procedure INSERT(x)
    for j := 1 to k do
        A[h_j(x)] := 1
```

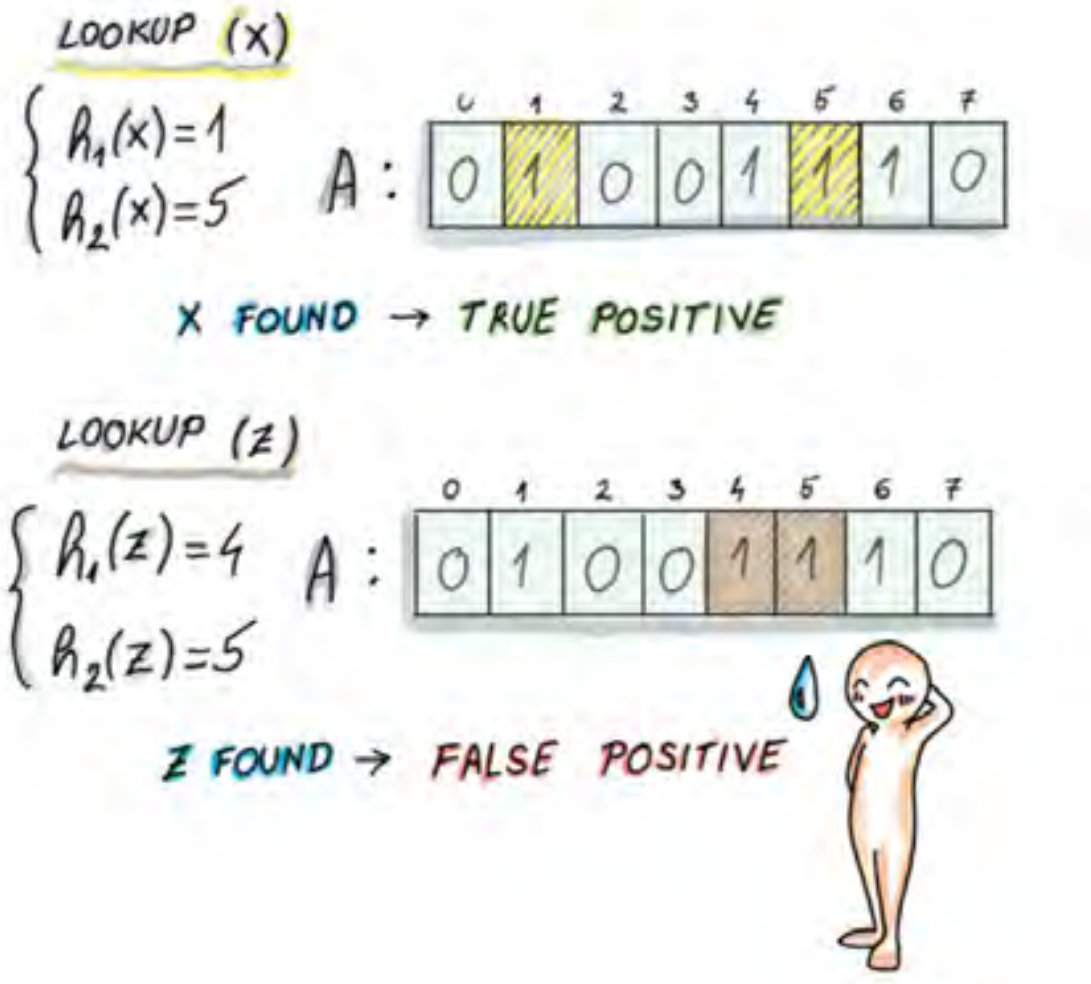
```
procedure LOOKUP(x)
    for j := 1 to k do
        if A[h_j(x)] = 0 then
            return false
    return true
```

Insertion & lookup



Source: D. Medjedovic & E. Tahirovic, *Algorithms and Data Structures for Massive Datasets*, 2022

Insertion & Lookup



Source: D. Medjedovic & E. Tahirovic, *Algorithms and Data Structures for Massive Datasets*, 2022

Analysis of Bloom filters

- Probability that the j-th bit is not updated when inserting x

$$\prod_{i=1}^{k} \mathbb{P}[h_i(x) \neq j] = \left(1 - \frac{1}{M}\right)^k$$

- Probability that the j-th bit is not updated after n insertions

$$\prod_{\ell=1}^{n} \mathbb{P}[A[j] \text{ is not updated in } \ell\text{-th insertion}] =$$

$$\left(\left(1 - \frac{1}{M}\right)^k\right)^n = \left(1 - \frac{1}{M}\right)^{k \cdot n}$$

Analysis of Bloom filters

- Probability that $A[j] = 1$ after n insertions

$$1 - \left(1 - \frac{1}{M}\right)^{k \cdot n}$$

- Probability that k checked bits are set to 1 $\approx$ probability of a false positive

$$\left(1 - \left(1 - \frac{1}{M}\right)^{k \cdot n}\right)^{k} \approx \left(1 - e^{-kn/M}\right)^{k}$$

if $n = \alpha M$, for some $\alpha > 0$

$$\left(1 - \frac{a}{x}\right)^{bx} \to e^{-ba}, \quad x \to \infty$$

Analysis of Bloom filters

- The derivation above is the so-called classic model for Bloom filters—but it is not the formula that Bloom himself derived in his paper!
- The approximation fails for small filters; correct formulas have been derived by Bose et al. (2008) and Christensen et al. (2010)
- For the rest of the presentation we will take

$$\mathbb{P}[x \text{ is a false positive}] = \mathbb{P}[x \notin F \wedge F.\texttt{contains}(x) = \textbf{true}]$$
$$\approx \left(1 - e^{-kn/M}\right)^k,$$

where x is drawn at random. Be careful! The formula does not give the probability that the filter reports x as a positive, conditioned to x being negative!

Optimal parameters for Bloom filters

- Fix n and M. The optimal value k^* minimizes the probability of false positive, thus

$$\frac{d}{dk}\left[\left(1-e^{-kn/M}\right)^k\right]_{k=k^*}=0$$

  which gives

$$k^* \approx \frac{M}{n}\ln 2 \approx 0.69\frac{M}{n}$$

- Call p the probability of a false positive. This probability is a function of k, $p = p(k)$; for the optimal choice k^* we have

$$p(k^*) \approx \left(1-e^{-\ln 2}\right)^{\frac{M}{n}\ln 2} = \left(\frac{1}{2}\right)^{\ln 2\frac{M}{n}} \approx 0.6185^{\frac{M}{n}}$$

Optimal parameters for Bloom filters

- Suppose that you want the probability of false positive $p^* = p(k^*)$ to remain below some bound P

$$p^* \leq P \implies \ln p^* = -\frac{M}{n}(\ln 2)^2 \leq \ln P$$

$$\frac{M}{n}(\ln 2)^2 \geq -\ln P = \ln(1/P)$$

$$\frac{M}{n} \geq \frac{1}{\ln 2}\log_2(1/P) \approx 1.44 \log_2(1/P)$$

$$M \geq 1.44 \cdot n \cdot \log_2(1/P)$$

Optimal parameters for Bloom filters

procedure CREATEBF(N_{max}, fp)
$M := 1.44 \cdot N_{max} \cdot \log_2(1/fp)$;
$k := \log_2(1/fp)$
...

Optimal parameters for Bloom filters

- If we want a Bloom filter for a database that will store about $n \approx 10^8$ elements and a false positive rate $\leq 5\%$, we need a bitvector of size $M \geq 624 \cdot 10^6$ bits (that's around 74MB of memory).
- Despite this amount of memory is big, it is only a small fraction of the size of the database itself: even if we store only keys of 32 bytes each, the database occupies more than 3GB.
- The optimal number k^* of hash functions for the example above is 4.32 ($\Longrightarrow$ use 4 or 5 hash functions for optimal performance)

To learn more

[1] B.H. Bloom.
Space/Time Trade-offs in Hash Coding with Allowable Errors.
Communications of the ACM 13 (7): 422–426, 1970.

[2] A. Broder and M. Mitzenmacher.
Network Applications of Bloom Filters: A Survey
Internet Mathematics 1 (4):485–509, 2003.

To learn more (2)

[3] P. Bose, H. Guo, E.Kranakis et al.
On the False-Positive Rate of Bloom Filters
Information Processing Letters 108 (4):210–213, 2004.

[4] K. Christensen, A. Roginsky and M. Jimeneo.
A New Analysis of the False-Positive Rate of a Bloom Filter
Information Processing Letters 110 (21):944–949, 2010.

Universal hashing

M. N. Wegman

Definition

A class

$$\mathcal{H} = \{h \mid h : \mathcal{U} \to [0..M-1]\}$$

of hash functions is universal iff, for all $x, y \in \mathcal{U}$ *with* $x \neq y$ *we have*

$$\mathbb{P}[h(x) = h(y)] \leq \frac{1}{M},$$

where h *is a hash function randomly drawn from* $\mathcal{H}$

Universal hashing

A stronger property is pairwise independence (a.k.a. strong universality). A class is strongly universal iff, for all $x, y \in \mathcal{U}$ with $x \neq y$ and any two values $i, j \in [0..M-1]$

$$\mathbb{P}[h(x) = i \wedge h(y) = j] = \frac{1}{M^2}$$

Strong universality implies universality; moreover

$$\mathbb{P}[h(x) = i] = \frac{1}{M}$$

for any x and i.

Universal hashing

Let $\mathcal{H}$ be a universal class and $h \in \mathcal{H}$ drawn at random. For any fixed set of n keys $S \subseteq \mathcal{U}$ we have the following properties:

1. For any $x \in S$, the expected number of elements in S that hash to $h(x)$ is n/M.
2. The expected number of collisions is $O(n^2/M)$. If $M = \Theta(n)$ then the expected number of collisions is $O(n)$.

Universal hashing

The big questions are:

- Are there universal classes? Strongly universal classes?
- If so, how complicated are its members? How much effort does it take to compute and represent the functions in the class?

Universal hashing

In 1977 Carter and Wegman introduced the concept of universal class of hash functions and gave the first construction. In what follows we put the universe $\mathcal{U}$ into one-to-one correspondence with $[0..U-1]$ ($U = |\mathcal{U}|$).

Theorem

Let $U = |\mathcal{U}|$ and let p be a prime number $\geq U$. The class

$$\mathcal{H} = \{h_{a,b} : \mathcal{U} \to [0..M-1] \mid 0 < a < p, 0 \leq b < p\}$$

is (strongly) universal, with

$$h_{a,b}(x) = ((ax + b) \mod p) \mod M$$

Universal hashing

The ingredients we need are thus a BIG prime p; picking a hash function at random from $\mathcal{H}$ amounts to choosing two integers a and b at random.
Let $r = \lceil \log_2(U+1) \rceil$. The prime number p and the numbers a and b will need roughly r bits each. For instance, if our universe are ASCII strings of length at most 30, $U \approx 256^{30}$ and $r \approx 240$ bits; these are huge numbers and a fast primality test is a must-have for a practical scheme.

Universal hashing

Suppose that $h_{a,b}$ has been picked at random and let x and y be two distinct keys that collide

$$h_{a,b}(x) = h_{a,b}(y)$$

Therefore

$$ax + b \equiv ay + b + \lambda \cdot M \pmod{p}$$

for some integer $\lambda \geq 0$, $\lambda \leq p/M$.

Universal hashing

Since $x \neq y$, $x - y \neq 0$, hence $x - y$ has an inverse multiplicative in the ring $\mathbb{Z}_p$, denote it $(x - y)^{-1}$.
Hence

$$\begin{aligned} ax &\equiv ay + \lambda \cdot M \pmod{p} \\ a(x - y) &\equiv \lambda \cdot M \pmod{p} \\ a &\equiv (x - y)^{-1} \cdot \lambda \cdot M \pmod{p} \end{aligned}$$

Universal hashing

There are $p - 1$ possible choices for a and $\lfloor p/M \rfloor$ possible values for λ; hence the probability of collision is

$$\leq \frac{\lfloor p/M \rfloor}{p-1} \approx \frac{1}{M}$$

for sufficiently large p.

Universal hashing

Notice that b plays no rôle in the universality of the family. We might have choosen $b = 0$ or any other convenient fixed value. However, picking b at random makes the class strongly universal.

To learn more

[1] L. Carter and M.N. Wegman.
Universal Classes of Hash Functions.
Journal of Computer and System Sciences, 18 (2): 143–154, 1979.

[2] O. Kaser and D. Lemire.
Strongly universal string hashing is fast.
Computer Journal (published on-line in 2013)

General References

[1] Ph. Flajolet and R. Sedgewick.
Analytic Combinatorics.
Cambridge University Press, 2008.

[2] D. E. Knuth.
The Art of Computer Programming: Sorting and Searching, volume 3.
Addison-Wesley, 2nd edition, 1998.

[3] C. Pandu Rangan.
Randomized Data Structures, in *Handbook of Data Structures and Applications*.
D.P. Mehta and S. Sahni, editors.
Chapman & Hall, CRC, 2005.

General References (2)

[4] P. Raghavan and R. Motwani.
Randomized Algorithms.
Cambridge University Press, 1995.

[5] M. Mitzenmacher and E. Upfal.
Probability and computing: Randomized algorithms and probabilistic analysis.
Cambridge University Press, 2005.

General References (3)

[6] R. Sedgewick.
Algorithms in C.
Addison-Wesley, 3rd edition, 1997.

[7] R. Sedgewick and K. Wayne.
Algorithms.
Addison-Wesley, 4th edition, 2011.

[8] D. Medjedovic and E. Tahirovic.
Algorithms and Data Structures for Massive Datasets
Manning, 2022.

THANK YOU FOR YOUR PARTICIPATION!