095946- ADVANCED ALGORITHMS AND PARALLEL PROGRAMMING

Fabrizio Ferrandi

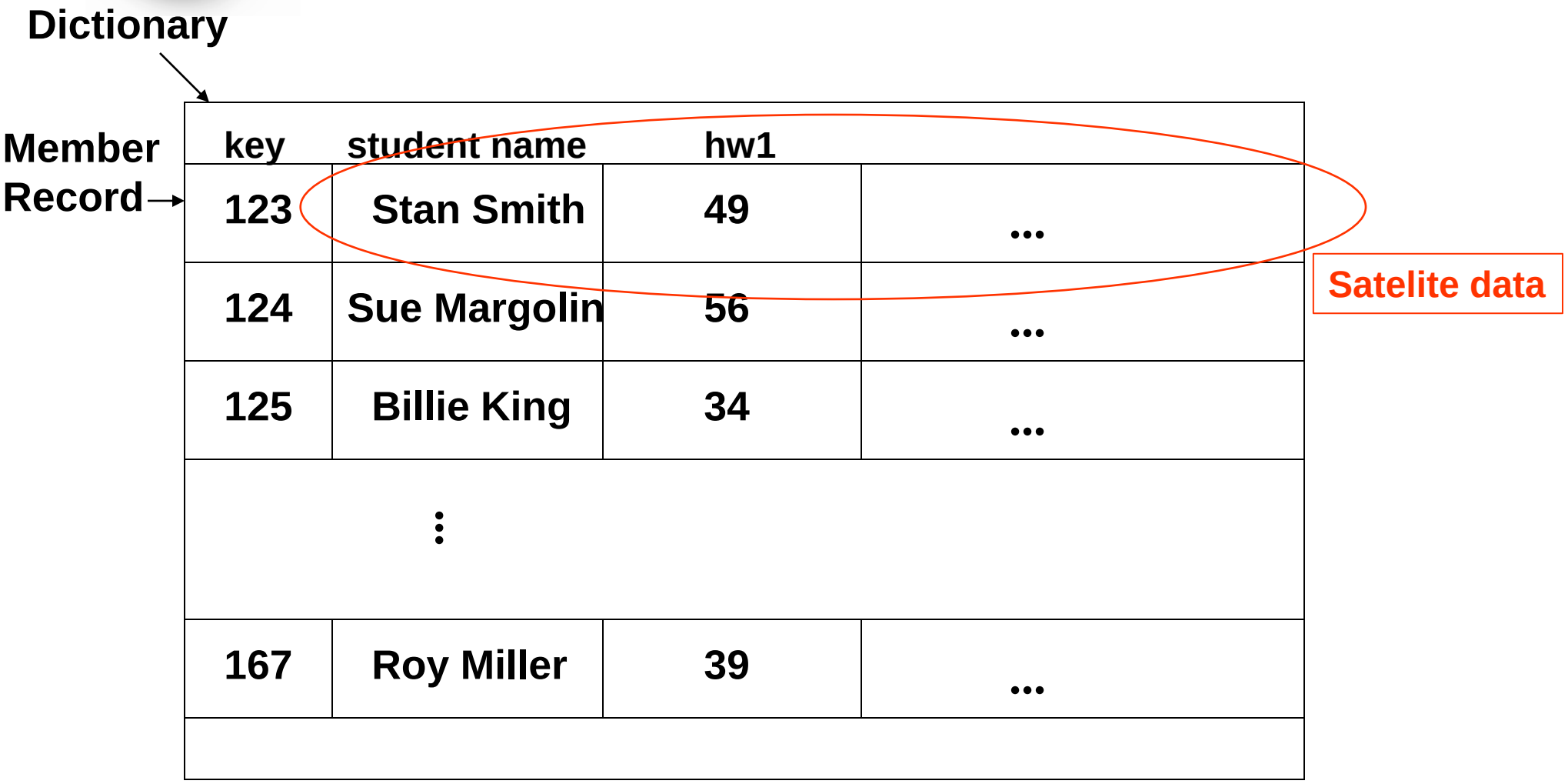
a.a. 2021-2022

- Randomized data structures
  - Random Treaps
  - Skip Lists

Dictionaries

- A dictionary is a collection of elements each of which has a unique search key
  - Uniqueness criteria may be relaxed (multiset)
  - (i.e. do not force uniqueness)
- Keep track of current members, with periodic insertions and deletions into the set
- Examples
  - Membership in a club, course records
  - Symbol table (contains duplicates)
  - Language dictionary (WordSmith, Webster, WordNet)
- Similar to database

Course Records



The dictionary problem

- **Given:** Universe $(U,<)$ of keys with a total order

- **Goal:** Maintain set $S \subseteq U$ under the following operations
- Search(x,S): Is $x \in S$?
- Insert(x,S): Insert x into S if not already in S.
- Delete(x,S): Delete x from S.

Extended set of operations

- Minimum(*S*): Return smallest key
- Maximum(*S*): Return largest key
- List(*S*): Output elements of *S* in increasing order by key
- Union(*S1*,*S2*): Merge S_1 and S_2

  Condition: $\forall\ x_1 \in S_1\ ,\ x_2 \in S_2 : x_1 < x_2$
- Split(S,x,S_1,S_2): Split *S* into S_1 and S_2.

  $\forall\ x_1 \in S_1\ ,\ x_2 \in S_2 : x_1 \leq x$ and $x < x_2$

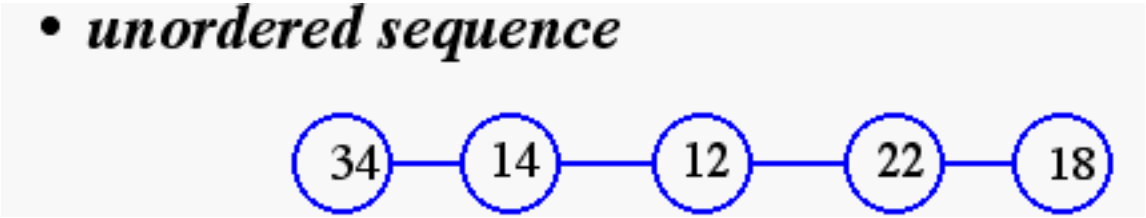
How to Implement a Dictionary?

- Sequences / Arrays
  - ordered
  - unordered
- Binary Search Trees
- AVL tree
- Splay tree

- Random Treaps
- Skip lists

Recall Arrays …

- Unordered array



- searching and removing takes O(n)
- inserting takes O(1) time
- applications to log files (frequent insertions, rare searches and removals)

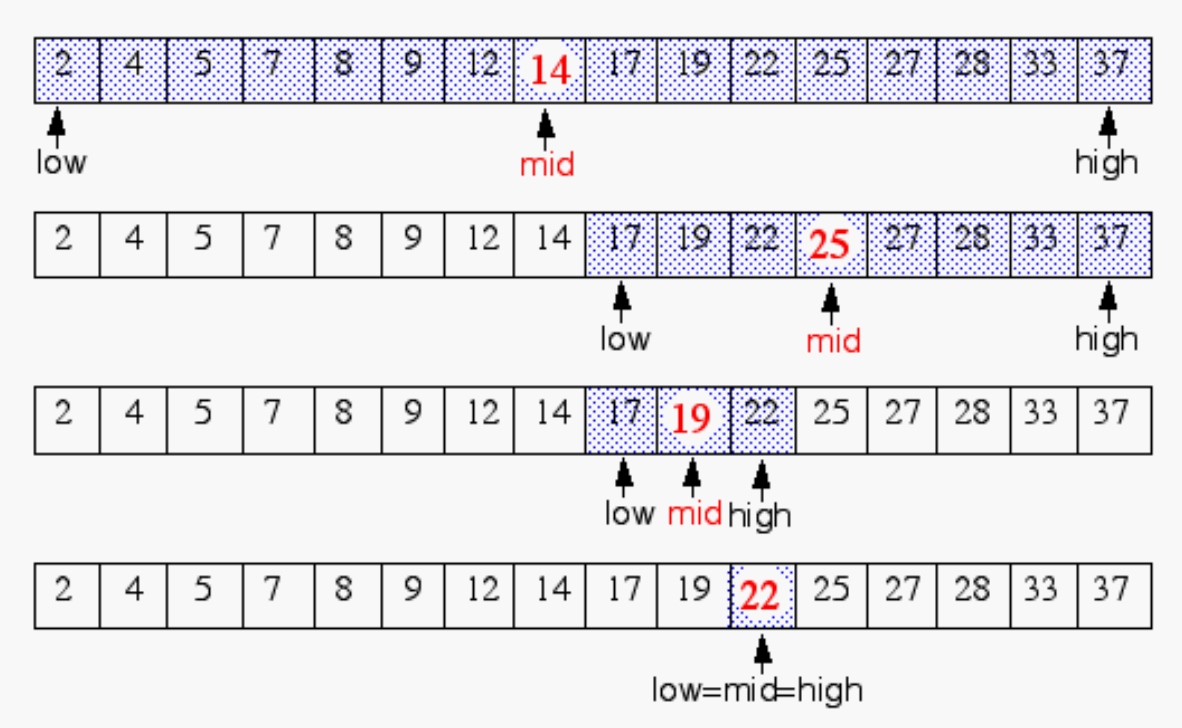
More Arrays

- Ordered array
- searching takes O(log n) time (binary search)
- inserting and removing takes O(n) time
- application to look-up tables (frequent searches, rare insertions and removals)

- Apply binary search

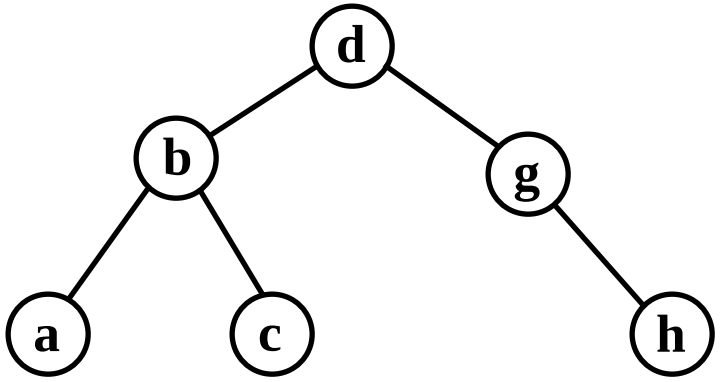
Binary Searches

- narrow down the search range in stages
- “high-low” game
- findElement(22)

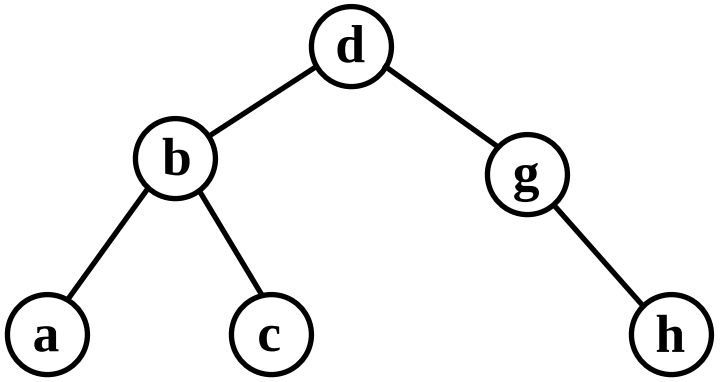


Recall Binary Search Trees…

- Implement a dictionary with a BST
  - A binary search tree is a binary tree T such that
  - each internal node stores an item (k, e) of a dictionary.
  - keys stored at nodes in the left subtree of v are less than or equal to k.
  - keys stored at nodes in the right subtree of v are greater than or equal to k.



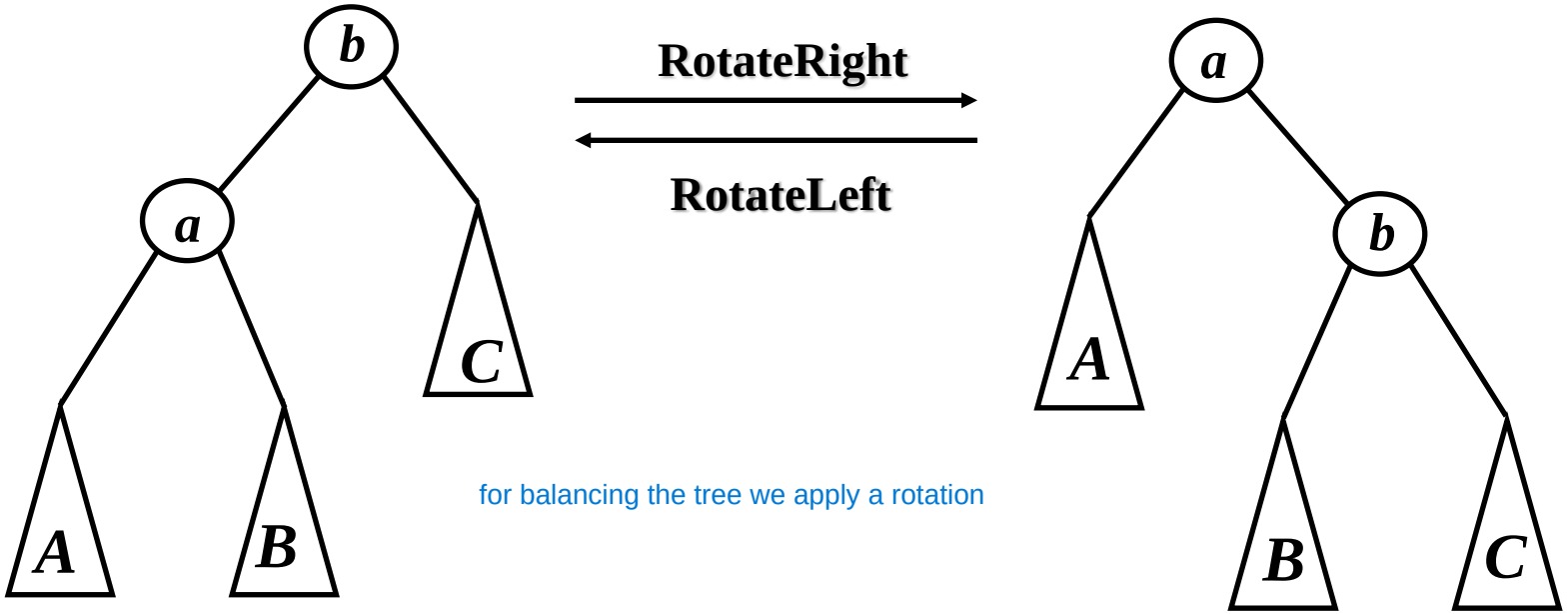
Binary search trees



- BST Drawback: Sequence of insertions may lead to a linear list a, b, c, d, e, f
- AVL tree
  - Better performance, difficult to implement
- Splay tree
  - Good overall performance -- *amortized*

AVL tree - self-balancing binary search tree

- Several strategies have been devised to handle BST drawbacks, usually involving balancing operations to ensure that the tree has height O(log *n)*
  - The most commonly used strategy is to perform *rotations* during the update operations



for balancing the tree we apply a rotation

Splay tree

- A different strategy, called *splaying*, is used in "self-adjusting" search trees to guarantee an *amortized* time bound of O(log *n);*
  - the splay operation moves a specified node to the root via a sequence of rotations
  - Amortization is the partitioning of the total cost of a sequence of operations among the individual operations in that sequence; thus, an amortized time bound can be viewed as the average cost of the operations in a sequence

Splay tree

- The idea behind self-adjusting trees is to use a particular implementation of the splay operation to move to the root a node accessed by a FIND operation.
- If a node is accessed often enough, it will remain close to the root and will not contribute much to the total running time; an infrequently accessed node cannot contribute much to the total running time in any case
  - guarantee only amortized logarithmic time per operation
  - relatively simple to implement
  - do not require explicit balance information to be stored at nodes
  - can be shown to be optimal with respect to arbitrary access frequencies

Splay tree drawbacks

- they restructure the entire tree not only during updates but also while performing simple search operations
  - significant slowdown in practice in caching and paging environments
- during any given operation splay trees may perform a logarithmic number of rotations
- inefficient in implementing higher dimensional search trees
  - secondary data structures associated with each node of these higher dimensional trees, and the secondary data structure at any node depends on the set of keys stored in the sub-tree rooted at that node. Since the entire secondary data structure has to be recomputed during each rotation, the cost of performing a single rotation could increase from a constant to some super-linear function of the sub-tree size
- we do not have the guarantee that every operation will run quickly
  - we obtain bounds only on the total cost of the operations

Random Treaps

- treaps achieve essentially the same time bounds
- they do not require any explicit balance information
- the expected number of rotations performed is small for each operation
- simple to implement

- Skip lists similar benefits but alternative randomized data structure

Approach for randomized search trees

- If *n* elements are inserted in random order into a binary search tree, the expected depth is 1.39 log *n*

- Goal:
  - the search tree has the structure that would result if elements were inserted in the order of their priorities (property 1)

- Idea:
  - Each element *x* is assigned a priority chosen uniformly at random

$$\text{prio}(x) \in R$$

Treaps: tree + heap

- **Definition:** A treap is a binary tree
- Each node contains one element x with key(x) $\in U$ and prio(x) $\in R$
- The following properties hold:
  - Search tree property
    - For each element x:
    - elements y in the left subtree of x satisfy: key(y) < key(x)
    - elements y in the right subtree of x satisfy : key(y) > key(x)
  - Heap property
    - For all elements x,y:
      - If y is a child of x, then prio(y) > prio(x).
      - All priorities are pairwise distinct.

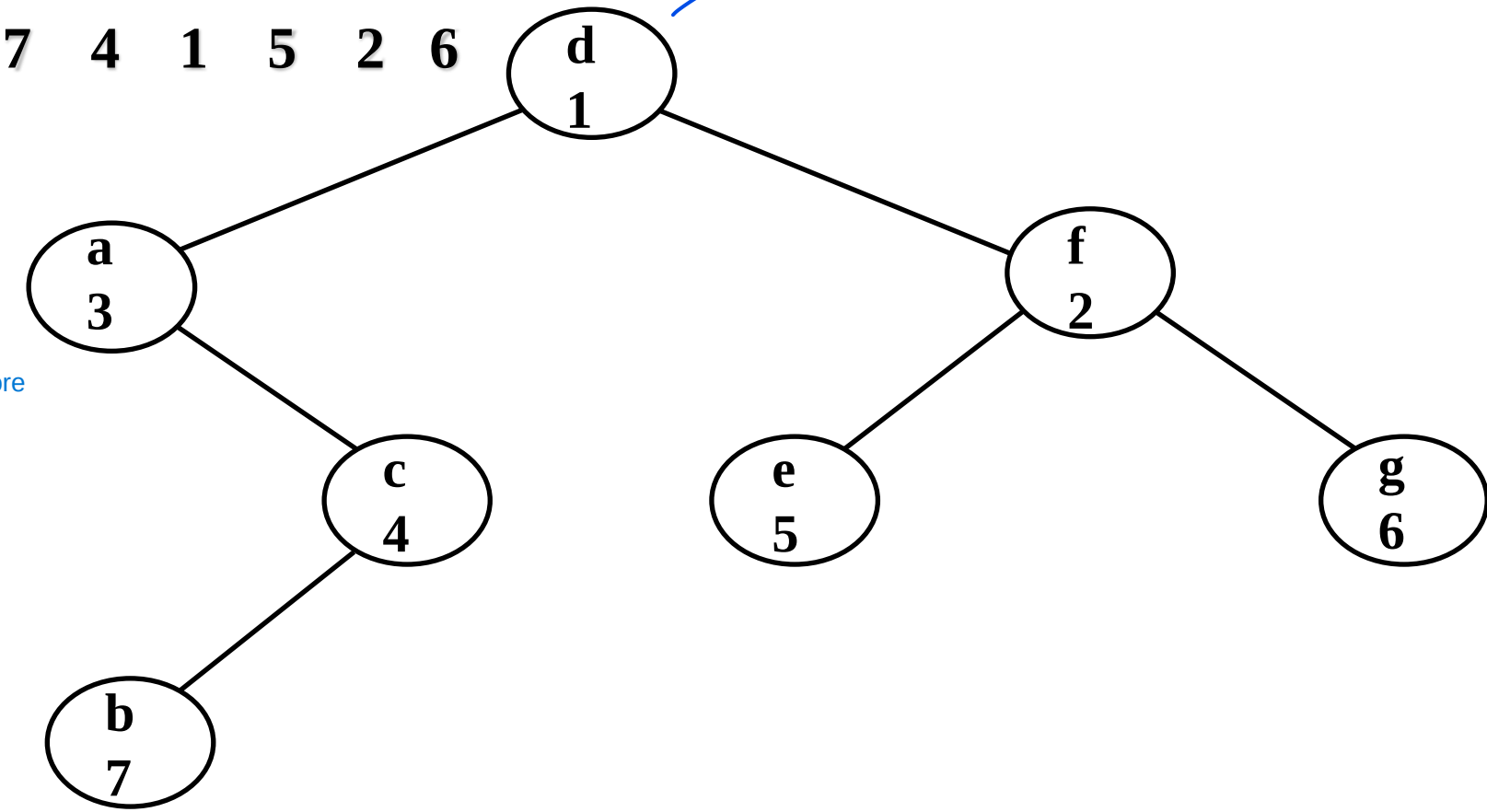
Example

key	a	b	c	d	e	f	g
Priority	3	7	4	1	5	2	6

with lower priority

nota che nel sottoalbero sx della root tutte le chiavi sono < di d.

e vale anche l'altra propr. che i nodi figli hanno priorità maggiore di quella del genitore



Treap uniqueness

- **Lemma:** For elements $x_1, \ldots, x_n$ with key(x_i) and prio(x_i), there exists a unique treap.

- **Proof by induction on n:**
- n=1: ok
- n>1: in that case we're focusing on the root and the children
  - The root has the smallest priority (k_1,p_1)
  - All the elements y with key(y) < k_1 *go on the left subtree* $\leq n-1$
  - All the elements y with key(y) > k_1 *go on the right subtree* $\leq n-1$

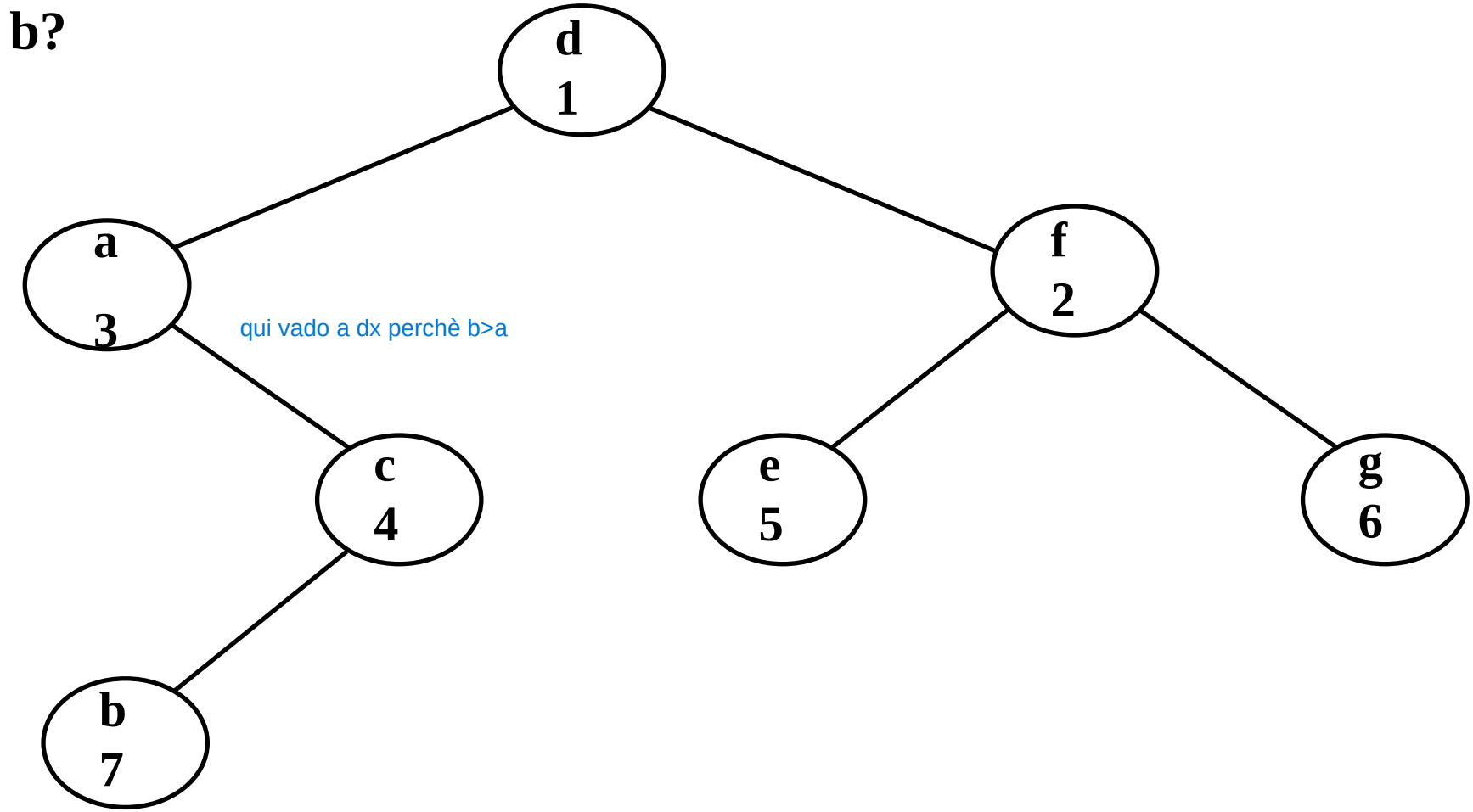
Treap structure

- the search tree has the structure that would result if elements were inserted in the order of their priorities

- **Proof by induction on n:**
- n=1: ok
- *n*>1:
  - The element with the smallest priority is in the root whatever it is the insertion order (k_1,p_1)
  - All y such that key(y) < k_1 *go in the left subtree*
  - All y such that key(y) > k_1 *go in the right subtree*

search propcedure. B is less than d so i go to the left and so on

Search for an element: key



Search for element with key *k*

```
1 v := root;
2 while v ≠ nil do
3    case key(v) = k : stop; "element found"
(successful search)
4       key(v) < k : v:= RightChild(v);
5       key(v) > k : v:= LeftChild(v);
6    endcase;
7 endwhile;
8 "element not found" (unsuccessful search)
```

Running time: O(# elements on the search path)

Analysis of the search path

Elements $x_1, \ldots, x_n$

- x_i has i-th smallest key
- $key(x_1) < key(x_2) < key(x_3) < \ldots < key(x_n)$

Let M be a subset of the elements.

$P_{min}(M)$ = element in M with lowest priority

Let's find out x_m

Lemma:

a) Let $i<m$. x_i is ancestor of x_m iff $P_{min}(\{x_i,\ldots,x_m\}) = x_i$

b) Let $m<i$. x_i is ancestor of x_m iff $P_{min}(\{x_m,\ldots,x_i\}) = x_i$

Analysis of the search path

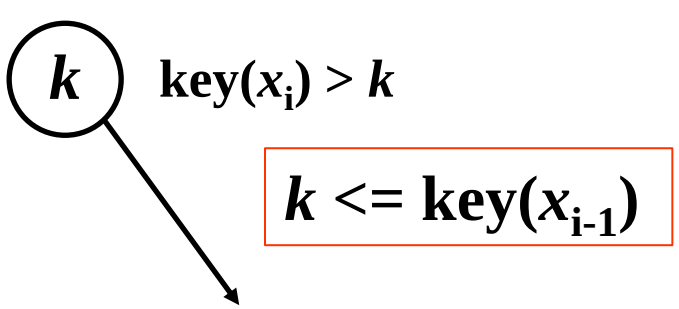
Proof: a) Use of property 1. Elements are inserted in order of increasing priorities.

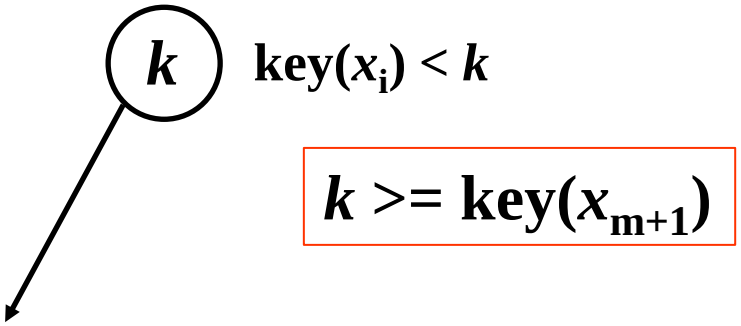
"⇐" $P_{min}(\{x_i,\dots,x_m\}) = x_i \quad \Rightarrow \quad x_i$ is inserted first among $\{x_i,\dots,x_m\}$.

When x_i is inserted, the tree contains only keys k with

$k < key(x_i)$ or $k > key(x_m)$

$$key(x_i) < \dots < key(x_m)$$





Analysis of the search path

Proof: a) (Let $i<m$. x_i is ancestor of x_m iff Pmin($\{x_i,\ldots,x_m\}$) = x_i)

"⇒" Let $x_j = P_{min}(\{x_i,\ldots,x_m\})$. Show: $x_i = x_j$

Suppose: $x_i \neq x_j$

Use property 1, considering the search path when x_j is inserted

As before any x_l traverse the same path as x_j

imply that x_j is ancestor of x_l

Case 1: $x_j = x_m$

Case 2: $x_j \neq x_m$

Part b) follows analogously.

Analysis of the 'Search' operation

Let T be a treap with elements $x_1, \ldots, x_n$ x_i has i-th smallest key

n-th Harmonic number:

$$H_n = \sum_{k=1}^{n} 1/k$$

$$H_n = \ln n + O(1)$$

Lemma:

1. Successful search: The expected number of nodes on the path to x_m is $H_m + H_{n-m+1} - 1$.

2. Unsuccessful search : Let m be the number of keys that are smaller than the search key k. The expected number of nodes on the search path is $H_m + H_{n-m}$.

Analysis of the 'Search' operation

Proof: Part 1

$$X_{m,i} = \begin{cases} 1 & x_i \text{ is ancestor of } x_m \\ 0 & \text{otherwise} \end{cases}$$

X_m = # nodes on the path from the root to x_m (incl. x_m)

in case i is before m

in case i is after m

$$X_m = 1 + \sum_{i<m} X_{m,i} + \sum_{i>m} X_{m,i}$$

$$E[X_m] = 1 + E\left[\sum_{i<m} X_{m,i}\right] + E\left[\sum_{i>m} X_{m,i}\right]$$

Analysis of the 'Search' operation

***i* < *m* :**

$$E[X_{m,i}] = \text{Prob}[x_i \text{ is ancestor of } x_m] = 1/(m-i+1)$$

All elements in $\{x_i, \ldots, x_m\}$ have the same probability of being the one with the smallest priority

Prob[$P_{min}(\{x_i,\ldots,x_m\}) = x_i$] = 1/(*m*-*i*+*1*)

***i* > *m* :**

$$E[X_{m,i}] = 1/(i-m+1)$$

Analysis of the ‘Search’ operation

$$E[X_m] = 1 + \sum_{i<m} \frac{1}{m-i+1} + \sum_{i>m} \frac{1}{i-m+1}$$

$$= 1 + \frac{1}{m} + \ldots + \frac{1}{2} + \frac{1}{2} + \ldots + \frac{1}{n-m+1}$$

$$= H_m + H_{n-m+1} - 1$$

so the searching is going to work with a complexity of logn (due to the fact we're considering binary tree)

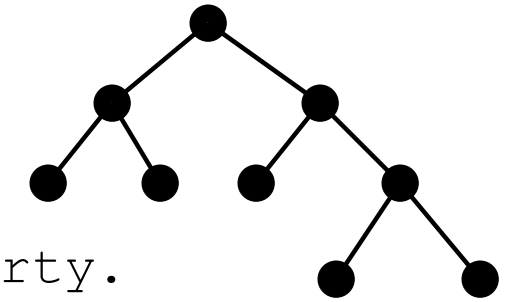
Part 2 follows analogously

Inserting a new element *x*

costant time

1. Choose prio(x).
2. Search for the position of *x* in the tree.

logn cost



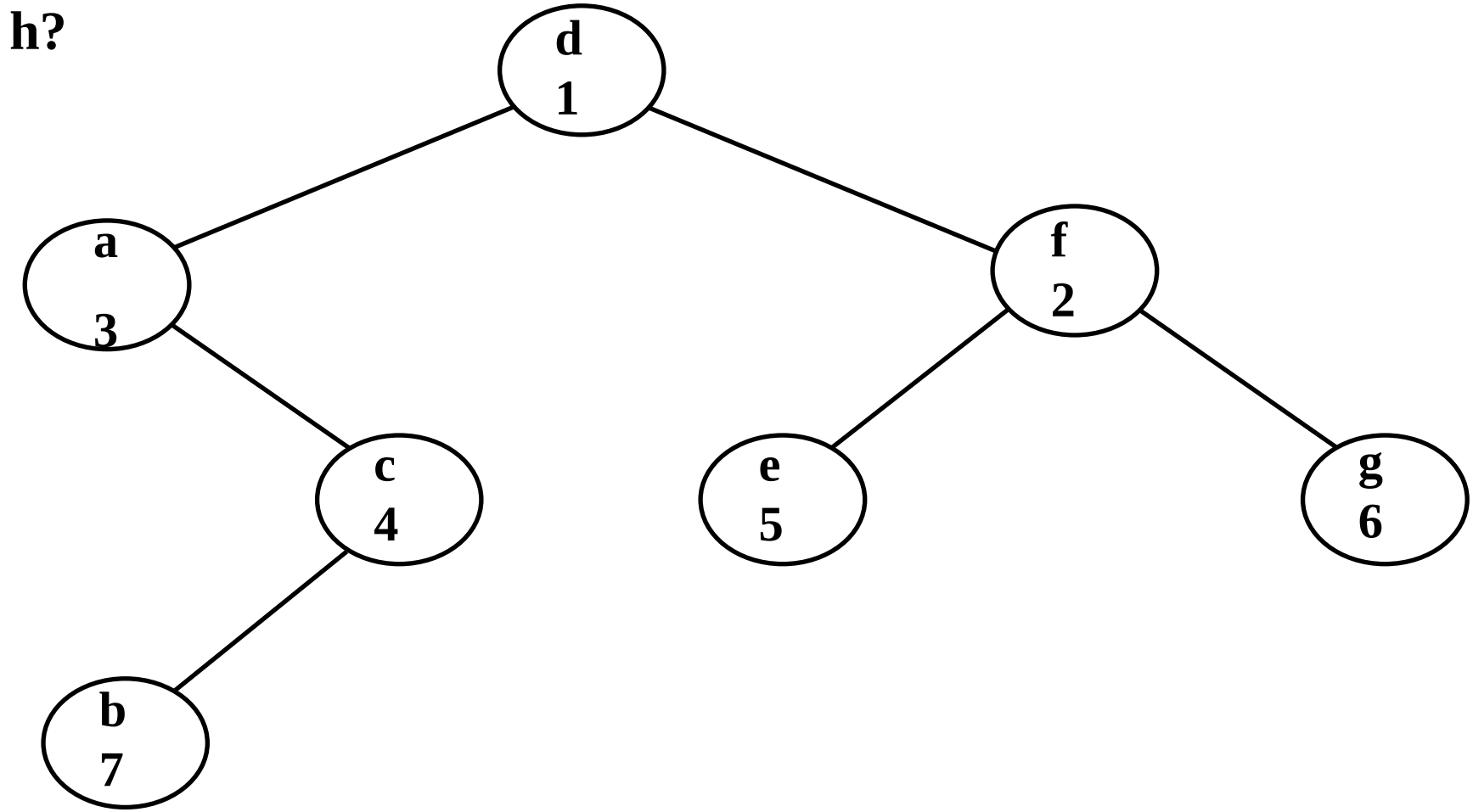
constant time

3. Insert *x* as a leaf.
4. Restore the heap property.

```
while prio(parent(x)) > prio(x) do
  if x is left child then RotateRight(parent(x))
  else RotateLeft(parent(x));
  endif
endwhile;
```

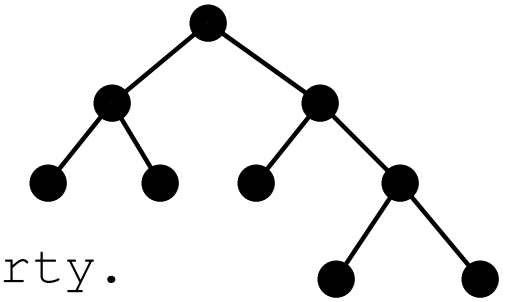
the rotation costs in the order of logn.

Insert an element



Inserting a new element *x*

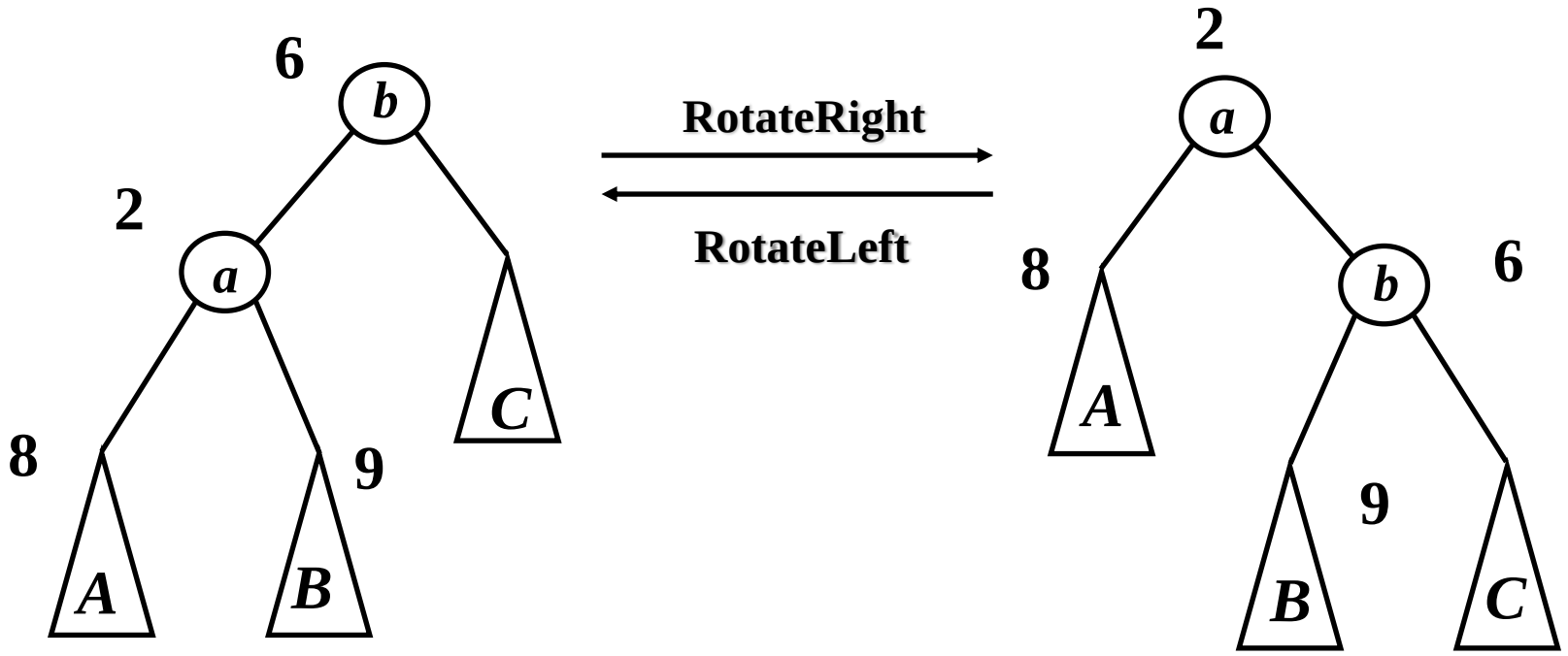
1. Choose prio(x).
2. Search for the position of *x* in the tree.



3. Insert *x* as a leaf.
4. Restore the heap property.

```
while prio(parent(x)) > prio(x) do
  if x is left child then RotateRight(parent(x))
  else RotateLeft(parent(x));
  endif
endwhile;
```

Rotations



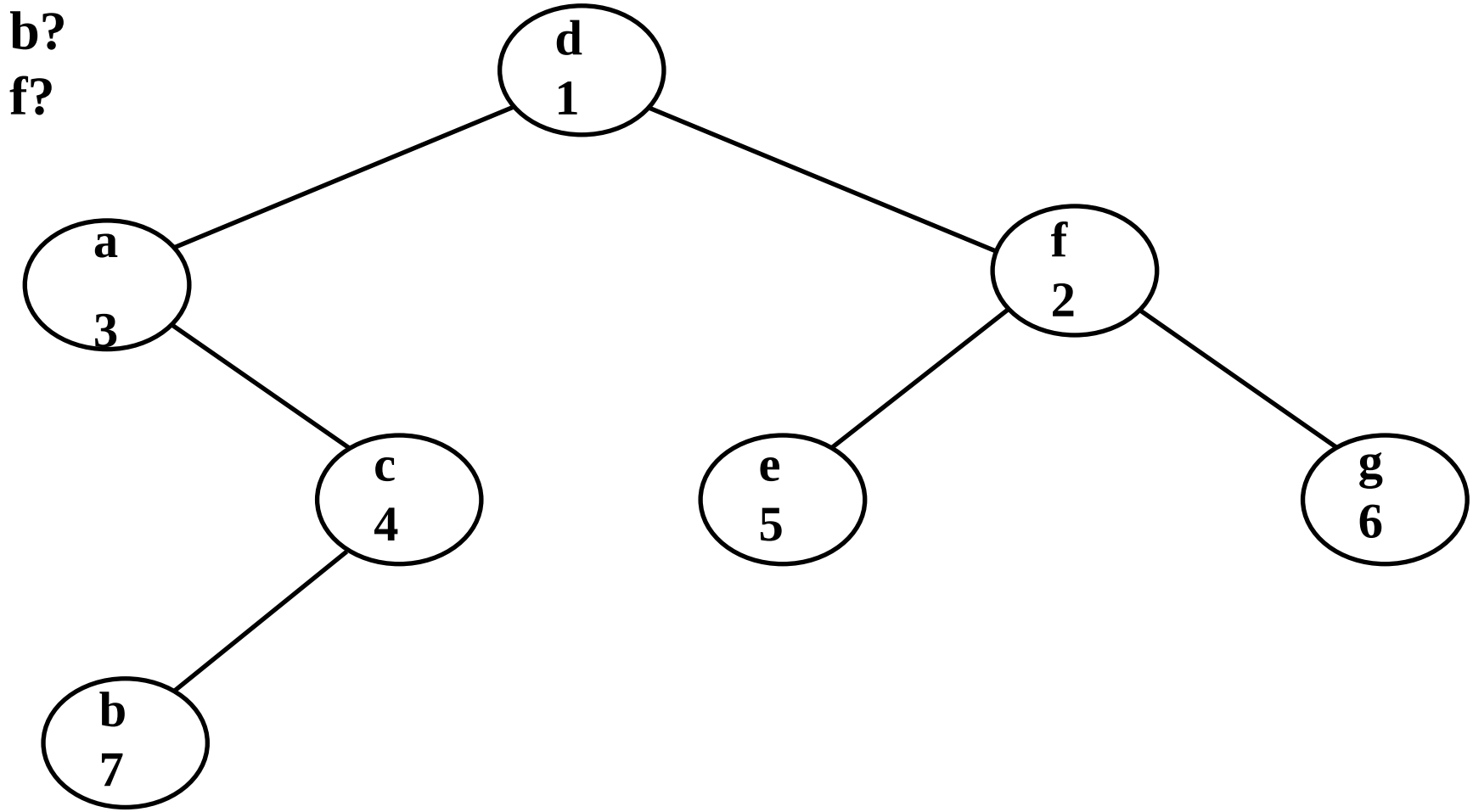
- The rotations maintain the search tree property and restore the heap property.

Deleting an element *x*

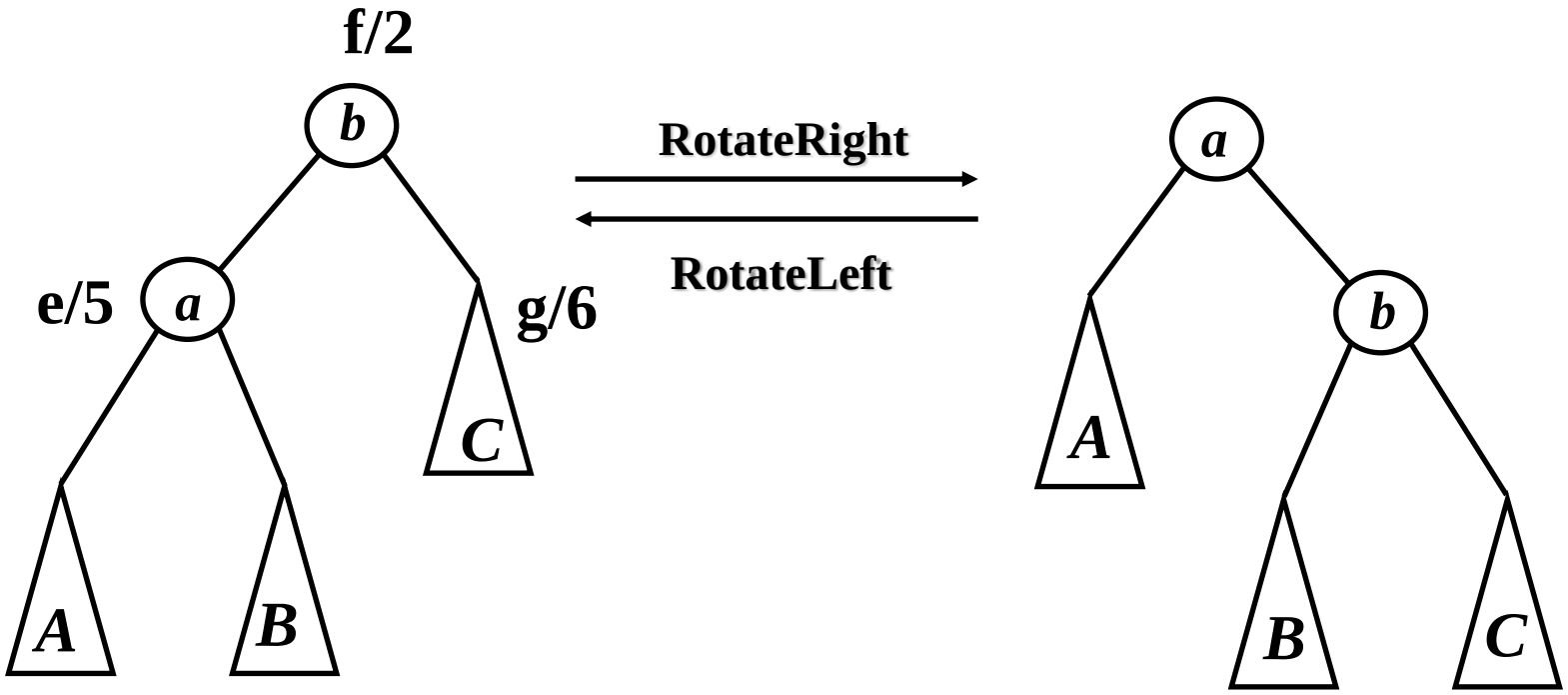
```
1.Find x in the tree.
2. while x is not a leaf do
       u := child with smaller priority;
       if u is left child then RotateRight(x))
       else RotateLeft(x);
       endif;
   endwhile;
3. Delete x;
```

if the element is a leaf, we can remove directly teh node in a costant time. otherwise we have to do a specific number of rotations before deleting it

Delete an element



Rotations



Analysis of ‘Insert’ and ‘Delete’ operations

Lemma: The expected running time of insert and delete operations is O(log *n*). The expected number of rotations is 2.

Proof: Analysis of insert (delete is the inverse operation)

rotations = depth of *x* after being inserted as a leaf (1)

- depth of *x* after the rotations (2)

Let $x = x_m$.

(2) Expected depth is $H_m + H_{n-m+1} - 1$.

(1) Expected depth is $H_{m-1} + H_{n-m} + 1$.

The tree contains *n-1* elements, *m-1* of them being smaller.

rotations = $H_{m-1} + H_{n-m} + 1 - (H_m + H_{n-m+1} - 1) < 2$

Extended set of operations

n = number of elements in treap T.

- Minimum(T): Return the smallest key. O(log n)
- Maximum(T): Return the largest key. O(log n)
- List(T): Output elements of S in increasing order. O(n)
- Union(T_1,T_2): Merge T_1 and T_2.

  Condition: $\forall\ x_1 \in T_1\ ,\ x_2 \in T_2$: key($x_1$) < key($x_2$)
- Split(T,k,T_1,T_2): Split T into T_1 and T_2.

  $\forall\ x_1 \in T_1\ ,\ x_2 \in T_2$: key($x_1$) ≤ k and k < key(x_2)

The ‘Split’ operation

Split(T,k,T_1,T_2): Split T into T_1 and T_2.

$\forall\ x_1 \in T_1\ ,\ x_2 \in T_2$: key($x_1$) ≤ k and k < key(x_2)

W.l.o.g. key k is not in T.

Otherwise delete the element with key k and re-insert it into T_1 after the split operation.

1. Generate a new element x with key(x)=k and prio(x)=-∞. (so the root -> minimum priority)
2. Insert x into T.
3. Delete the new root. The left subtree is T_1, the right subtree is T_2.

The ‘Union’ operation

Union(T_1,T_2): Merge T_1 and T_2.

Condition: $\forall\ x_1 \in T_1\ ,\ x_2 \in T_2$: key($x_1$) < key($x_2$)

1. Determine key k with key(x_1) < k < key(x_2)
   for all $x_1 \in T_1$and $x_2 \in T_2$.
2. Generate element x with key(x)=k and prio(x) = -∞.
3. Generate treap T with root x, left subtree T_1 and right subtree T_2.
4. Delete x from T.

Analysis

- **Lemma:** The expected running time of the operations Union and Split is O(log *n*).

Now there is a probelm. -> when we're generating with a randomic algorithm priority we have to consider that the child of the parent must have a priority < than the parent's priority. -> we have to manage the randomic algorithm taking care also of this problem

Implementation

- Priorities from [0,1)

- Priorities are used only when two elements are compared to find out which of them has the higher priority. stiamo considerando le priorità rappresentabili con un certo numero di bits

- In case of equality, extend both priorities by bits chosen uniformly at random until two corresponding bits differ.

$p_1 = 0{,}010111001$
$p_2 = 0{,}010111001$

$p_1 = 0{,}010111001011$
$p_2 = 0{,}010111001010$

genero i primi bit rossi, se sono uguale ne ripeto altri (sempre randomicamente) fino ad ottenere due priorità diverse

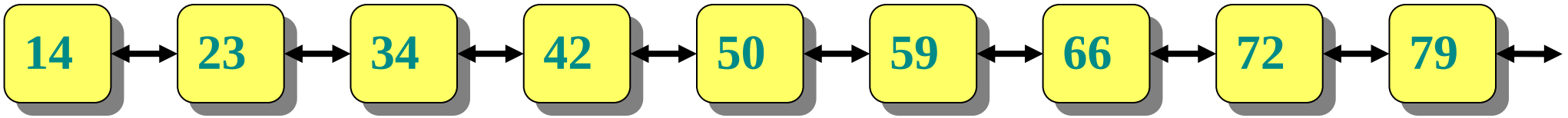
Skip lists

- Simple randomized dynamic search structure
  - Invented by William Pugh in 1989
  - Easy to implement
- Maintains a dynamic set of n elements in $O(\lg n)$ time per operation in expectation and *with high probability*
  - Strong guarantee on tail of distribution of $T(n)$
  - $O(\lg n)$ "almost always"

One linked list

Start from simplest data structure:
(sorted) linked list

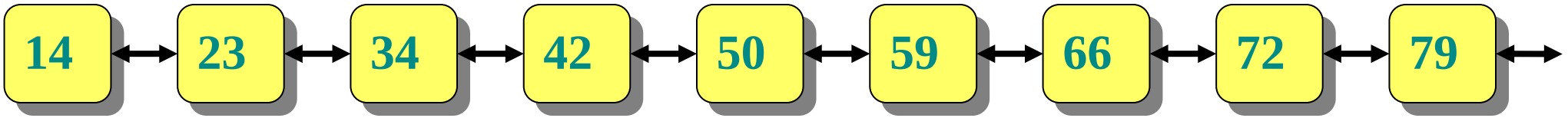
- Searches take $\Theta(n)$ time in worst case
- How can we speed up searches?



Two linked lists

Suppose we had *two* sorted linked lists
(on subsets of the elements)

- Each element can appear in one or both lists
- How can we speed up searches?

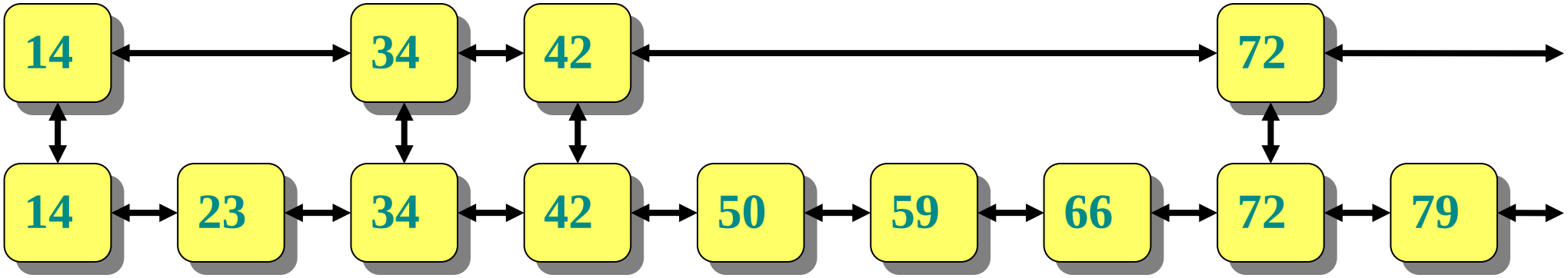


Two linked lists as a subway

IDEA: Express and local subway lines (à la New York City 7th Avenue Line)

- Express line connects a few of the stations
- Local line connects all stations
- Links between lines at common stations

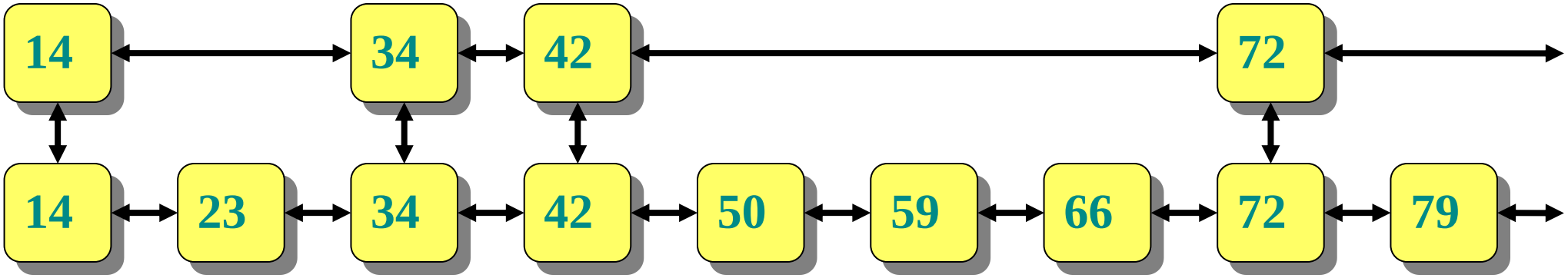
sta prendendo a caso logn elementi dalla lista di sotto e li sta mettendo nella lista di sopra che in qualche modo è collegata alla seconda. Così se sono fortunato posso trovare l'elemento che sto cercando subito pagando solo logn (che è minore di n), se invece non la trovo li, posso cercarla sotto visto che ho dei collegamenti ma comunque spendo meno di n



Searching in two linked lists

SEARCH(x):

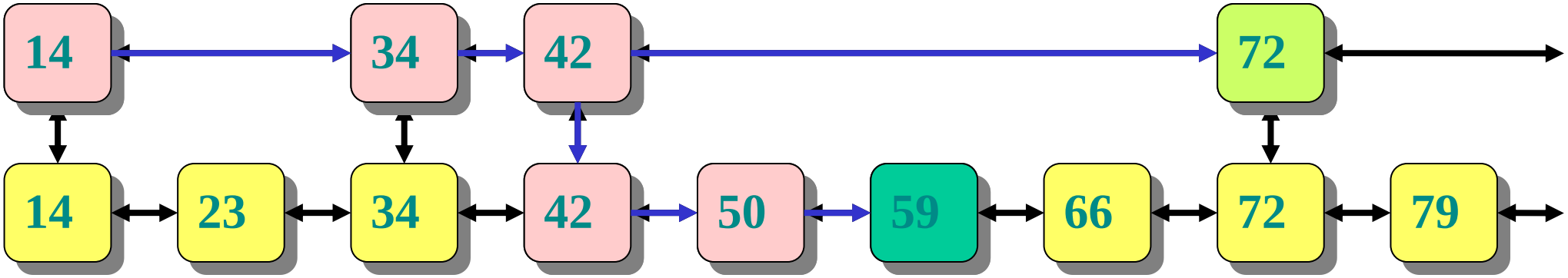
- Walk right in top linked list (L_1) until going right would go too far
- Walk down to bottom linked list (L_2)
- Walk right in L_2 until element found (or not)



Searching in two linked lists

EXAMPLE: SEARCH(59)

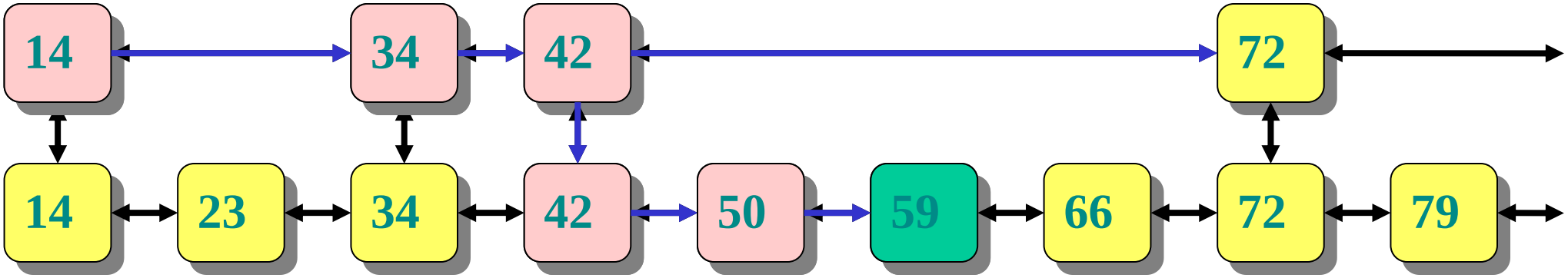
Too far:
59 < 72



Design of two linked lists

QUESTION: Which nodes should be in L_1?

- In a subway, the "popular stations"
- Here we care about *worst-case performance*
- **Best approach:** Evenly space the nodes in L_1
- But *how many nodes* should be in L_1?

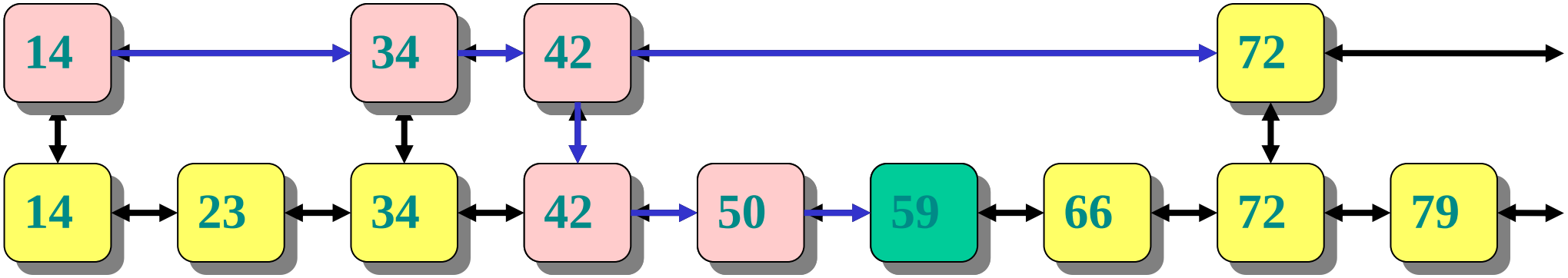


Analysis of two linked lists

ANALYSIS:

- Search cost is roughly $|L_1| + \frac{|L_2|}{|L_1|}$
- Minimized (up to constant factors) when terms are equal

$$|L_1|^2 = |L_2| = n \Rightarrow |L_1| = \sqrt{n}$$

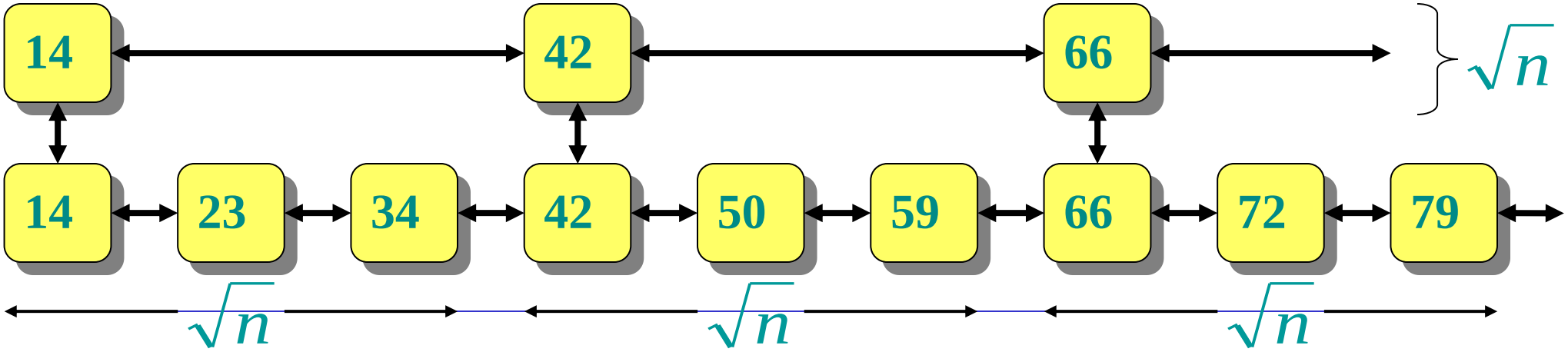


Analysis of two linked lists

ANALYSIS:

- $|L_1| = \sqrt{n}$ $\quad$ $|L_2| = n$
- Search cost is roughly

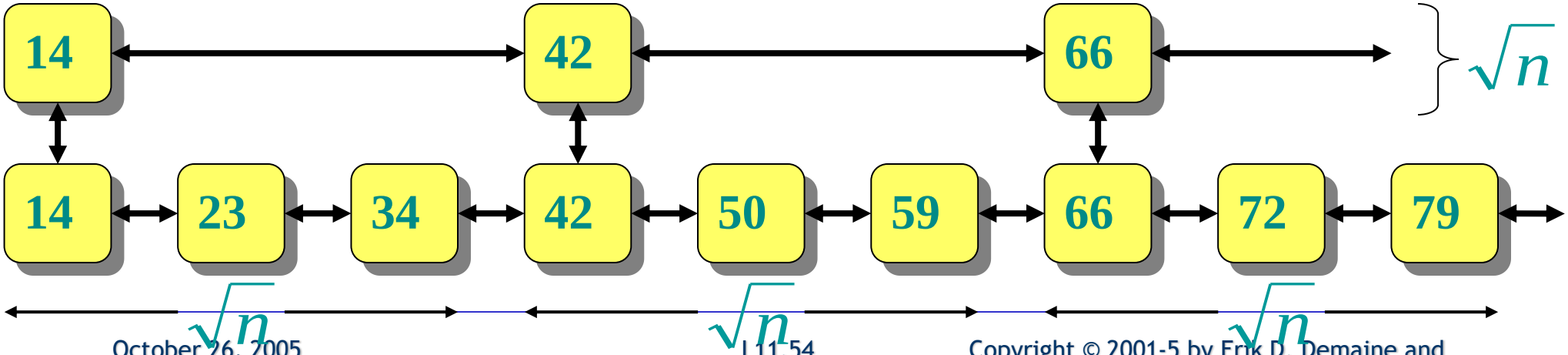
$$|L_1| + \frac{|L_2|}{|L_1|} = \sqrt{n} + \frac{n}{\sqrt{n}} = 2\sqrt{n}$$



More linked lists

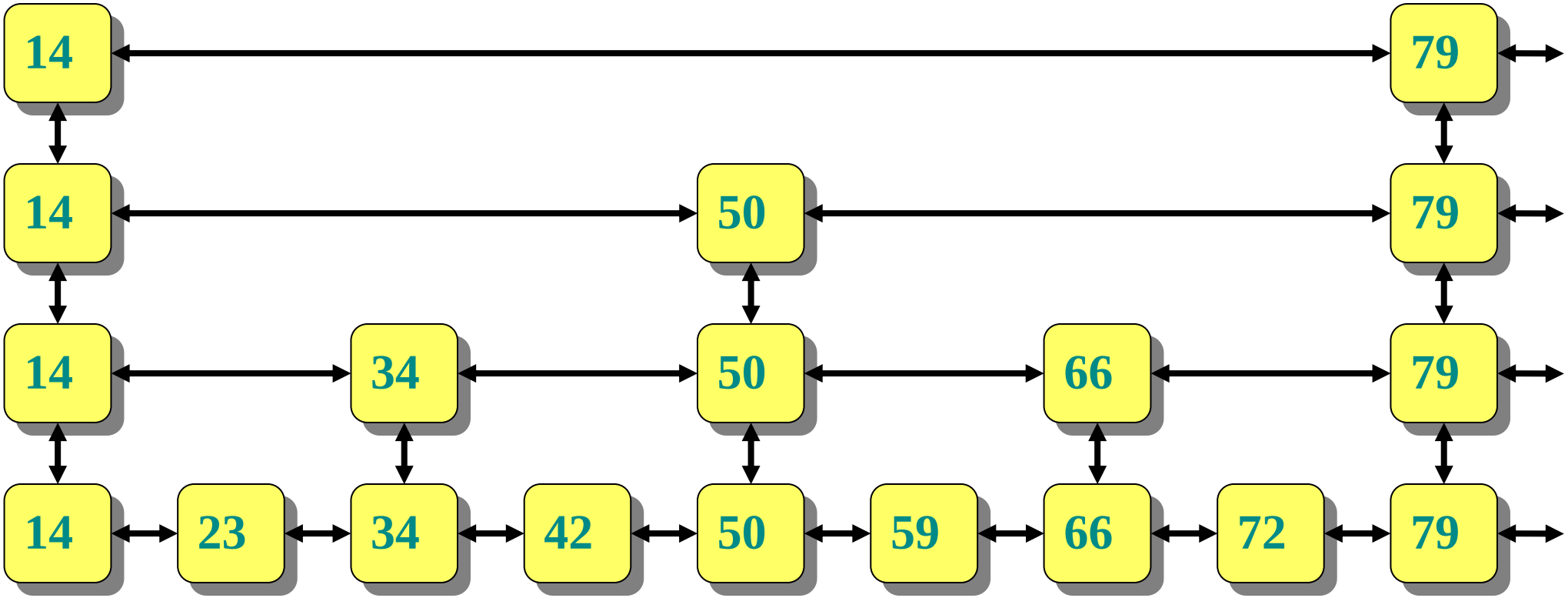
What if we had more sorted linked lists?

- 2 sorted lists $\Rightarrow 2 \cdot \sqrt{n}$
- 3 sorted lists $\Rightarrow 3 \cdot \sqrt[3]{n}$
- k sorted lists $\Rightarrow k \cdot \sqrt[k]{n}$
- $\lg n$ sorted lists $\Rightarrow \lg n \cdot \sqrt[\lg n]{n} = 2 \lg n$



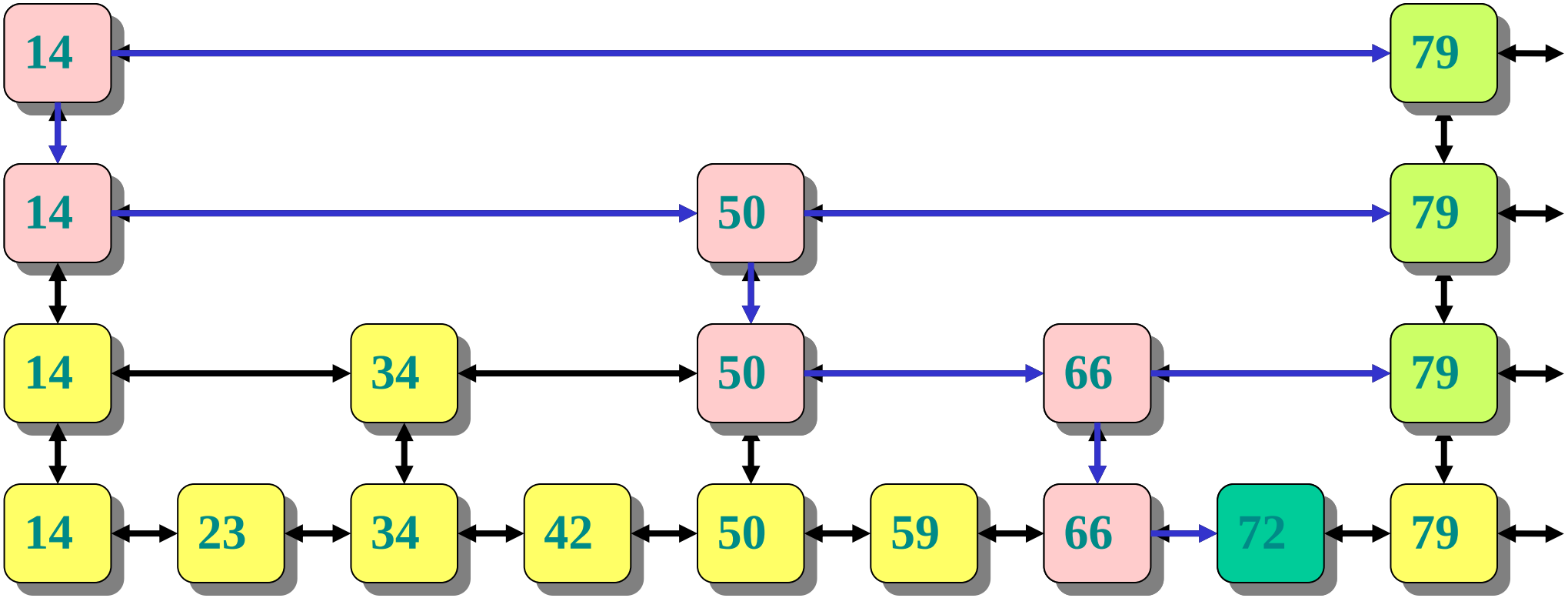
lg *n* linked lists

lg *n* sorted linked lists are like a binary tree



Searching in lg *n* linked lists

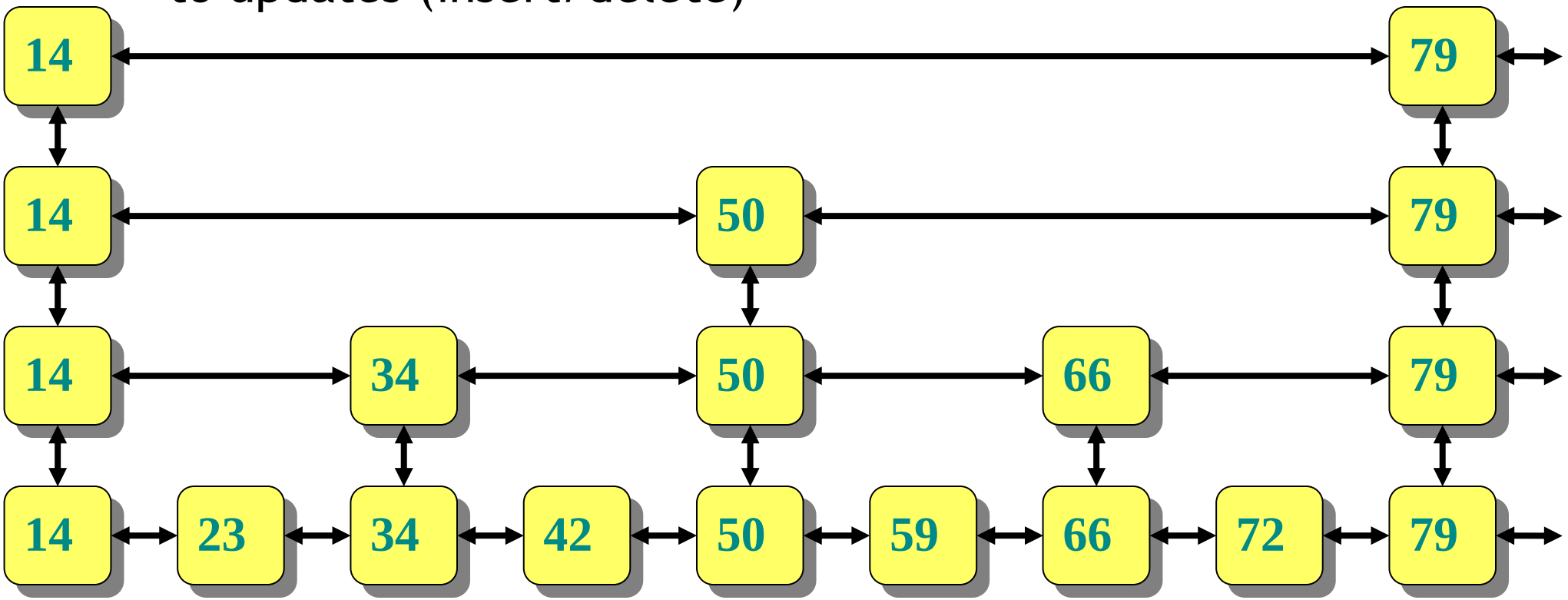
EXAMPLE: SEARCH(72)



Skip lists

Ideal skip list is this lg *n* linked list structure

Skip list data structure maintains roughly this structure subject to updates (insert/delete)



INSERT(x)

To insert an element x into a skip list:

- SEARCH(x) to see where x fits in bottom list
- Always insert into bottom list

INVARIANT: Bottom list contains all elements

- Insert into some of the lists above…

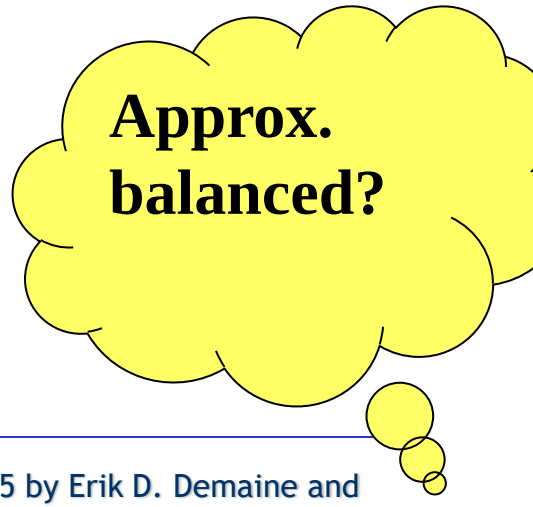
QUESTION: To which other lists should we add x?

INSERT(x)

QUESTION: To which other lists should we add x?

IDEA: Flip a (fair) coin; if HEADS,
promote x to next level up and flip again

- Probability of promotion to next level = 1/2
- On average:
  - 1/2 of the elements promoted 0 levels
  - 1/4 of the elements promoted 1 level
  - 1/8 of the elements promoted 2 levels
  - etc.

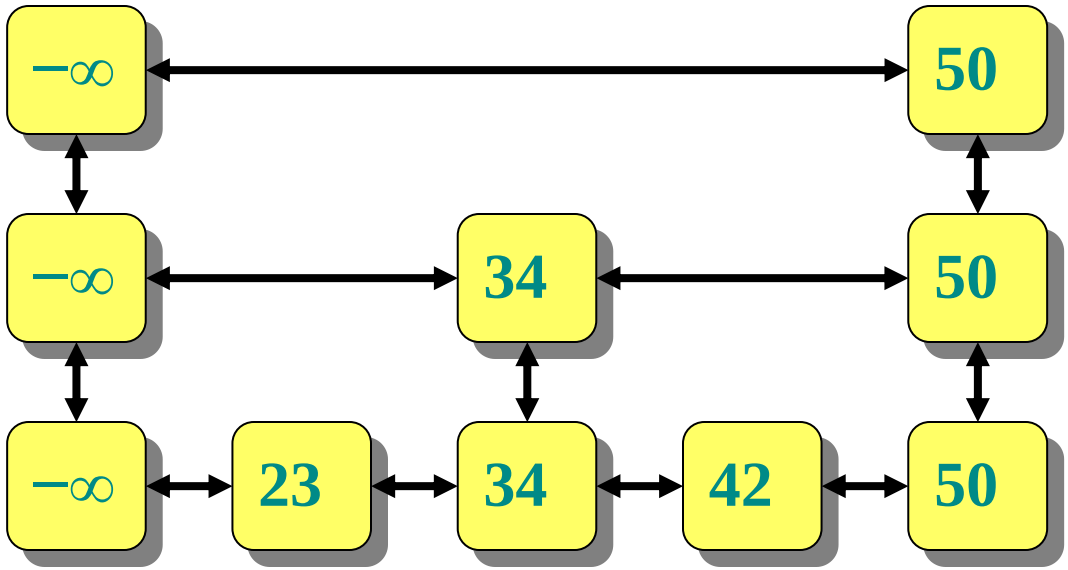


Example of skip list

EXERCISE: Try building a skip list from scratch by repeated insertion using a real coin

Small change:

- Add special $-\infty$ value to *every* list $\Rightarrow$ can search with the same algorithm



Skip lists

A ***skip list*** is the result of insertions (and deletions) from an initially empty structure (containing just $-\infty$)

- INSERT(x) uses random coin flips to decide promotion level
- DELETE(x) removes x from all lists containing it

Skip lists

A ***skip list*** is the result of insertions (and deletions) from an initially empty structure (containing just $-\infty$)

- INSERT(x) uses random coin flips to decide promotion level
- DELETE(x) removes x from all lists containing it

How good are skip lists? (speed/balance)

- **INTUITIVELY:** Pretty good on average
- **CLAIM:** Really, really good, almost always

With-high-probability theorem

THEOREM: *With high probability*, every search in an n-element skip list costs $O(\lg n)$

With-high-probability theorem

THEOREM: *With high probability*, every search in a skip list costs $O(\lg n)$

- **INFORMALLY:** Event E occurs ***with high probability*** (***w.h.p.***) if, for any $\alpha \geq 1$, there is an appropriate choice of constants for which
  E occurs with probability at least $1 - O(1/n^{\alpha})$
  - In fact, constant in $O(\lg n)$ depends on α
- **FORMALLY:** Parameterized event E_{α} occurs ***with high probability*** if, for any $\alpha \geq 1$, there is an appropriate choice of constants for which
  E_a occurs with probability at least $1 - c_{\alpha}/n^{\alpha}$

With-high-probability theorem

THEOREM: With high probability, every search in a skip list costs $O(\lg n)$

- **INFORMALLY:** Event E occurs ***with high probability*** (***w.h.p.***) if, for any $\alpha \geq 1$, there is an appropriate choice of constants for which
  E occurs with probability at least $1 - O(1/n^{\alpha})$
- **IDEA:** Can make ***error probability*** $O(1/n^{\alpha})$ very small by setting α large, e.g., 100
- Almost certainly, bound remains true for entire execution of polynomial-time algorithm

Boole's inequality / union bound

Recall:

BOOLE'S INEQUALITY / UNION BOUND:

For any random events $E_1, E_2, \ldots, E_k$,

$$\Pr\{E_1 \cup E_2 \cup \ldots \cup E_k\} \le \Pr\{E_1\} + \Pr\{E_2\} + \ldots + \Pr\{E_k\}$$

Application to with-high-probability events:

If $k = n^{O(1)}$, and each E_i occurs with high probability, then so does $E_1 \cap E_2 \cap \ldots \cap E_k$

Analysis Warmup

LEMMA: With high probability, n-element skip list has $O(\lg n)$ levels

PROOF:

- Error probability for having at most $c \lg n$ levels
  - $= \Pr\{\text{more than } c \lg n \text{ levels}\}$
  - $\leq n \cdot \Pr\{\text{element } x \text{ promoted at least } c \lg n \text{ times}\}$
    (by Boole's Inequality)
  - $= n \cdot (1/2^{c \lg n})$
  - $= n \cdot (1/n^c)$
  - $= 1/n^{c-1}$

Analysis Warmup

LEMMA: With high probability, n-element skip list has $O(\lg n)$ levels

PROOF:

- Error probability for having at most $c \lg n$ levels $\leq 1/n^{c-1}$
- This probability is ***polynomially small***, i.e., at most n^{α} for $\alpha = c - 1$.
- We can make α arbitrarily large by choosing the constant c in the $O(\lg n)$ bound accordingly.

Proof of theorem

THEOREM: With high probability, every search in an n-element skip list costs $O(\lg n)$

COOL IDEA: Analyze search backwards—leaf to root

- Search starts [ends] at leaf (node in bottom level)
- At each node visited:
  - If node wasn't promoted higher (got TAILS here), then we go [came from] left
  - If node was promoted higher (got HEADS here), then we go [came from] up
- Search stops [starts] at the root (or $-\infty$)

Proof of theorem

THEOREM: With high probability, every search in an n-element skip list costs $O(\lg n)$

COOL IDEA: Analyze search backwards—leaf to root

PROOF:

- Search makes "up" and "left" moves until it reaches the root (or $-\infty$)
- Number of "up" moves $<$ number of levels $\leq c \lg n$ w.h.p. *(Lemma)*
- $\Rightarrow$ w.h.p., number of moves is at most the number of times we need to flip a coin to get $c \lg n$ HEADs

Coin flipping analysis

CLAIM: Number of coin flips until $c \lg n$ HEADs
$= \Theta(\lg n)$ with high probability

PROOF:

Obviously $\Omega(\lg n)$: at least $c \lg n$

Prove $O(\lg n)$ "by example":

- Say we make $10\, c \lg n$ flips
- When are there at least $c \lg n$ HEADs?

(Later generalize to arbitrary values of 10)

Coin flipping analysis

CLAIM: Number of coin flips until $c \lg n$ HEADs
$= \Theta(\lg n)$ with high probability

PROOF:

- Pr{exactly $c \lg n$ HEADs} =

$$\underbrace{\binom{10c\lg n}{c\lg n}}_{\textbf{orders}} \cdot \underbrace{\left(\frac{1}{2}\right)^{c\lg n}}_{\textbf{HEADS}} \cdot \underbrace{\left(\frac{1}{2}\right)^{9c\lg n}}_{\textbf{TAILS}}$$

- Pr{at most $c \lg n$ HEADs} $\le$

$$\underbrace{\binom{10c\lg n}{c\lg n}}_{\textbf{overestimate on orders}} \cdot \underbrace{\left(\frac{1}{2}\right)^{9c\lg n}}_{\textbf{TAILS}}$$

Coin flipping analysis (cont'd)

- Recall bounds on $\binom{y}{x}$:

$$\left(\frac{y}{x}\right)^x \le \binom{y}{x} \le \left(e\frac{y}{x}\right)^x$$

- Pr{at most $c \lg n$ HEADs}

$$\le \binom{10c\lg n}{c\lg n}\cdot\left(\frac{1}{2}\right)^{9c\lg n}$$

$$\le \left(e\frac{10c\lg n}{c\lg n}\right)^{c\lg n}\cdot\left(\frac{1}{2}\right)^{9c\lg n}$$

$$= (10e)^{c\lg n}\,2^{-9c\lg n}$$

$$= 2^{\lg(10e)\cdot c\lg n}\,2^{-9c\lg n}$$

$$= 2^{[\lg(10e)-9]\cdot c\lg n}$$

$$= 1/n^{\alpha} \quad \textbf{for} \quad \alpha = [9-\lg(10e)]\cdot c$$

Coin flipping analysis (cont'd)

- Pr{at most $c \lg n$ HEADs} $\le 1/n^{\alpha}$ for $\alpha = [9 - \lg(10e)]c$
- **KEY PROPERTY:** $\alpha \to \infty$ as $10 \to \infty$, for any c
- So set 10, i.e., constant in $O(\lg n)$ bound, large enough to meet desired α

This completes the proof of the coin-flipping claim and the proof of the theorem.

Acknowledge

- Based on Randomized Algorithms, by R. Motwani and P. Raghavan.
- Based on Lectures of Prof. Dr. Th. Ottmann and of Prof. Dr. Susanne Albers: http://electures.informatik.uni-freiburg.de/portal/web/guest
- Based on Lectures of Prof. Rada Mihalcea Univ. of North Texas