Roussos Ioannis

Fundamental Data-structuring Problem

- Collection $\{S_1, S_2, \ldots\}$ of sets of items.
- Each item i is an arbitrary record, indexed by a key k(i).
- Each k(i) is drawn from a totally ordered universe U and all keys are distinct.

Maintain the above set so as to support certain types of queries and operations

Fundamental Data-structuring Problem (cont)

Operations to be supported:

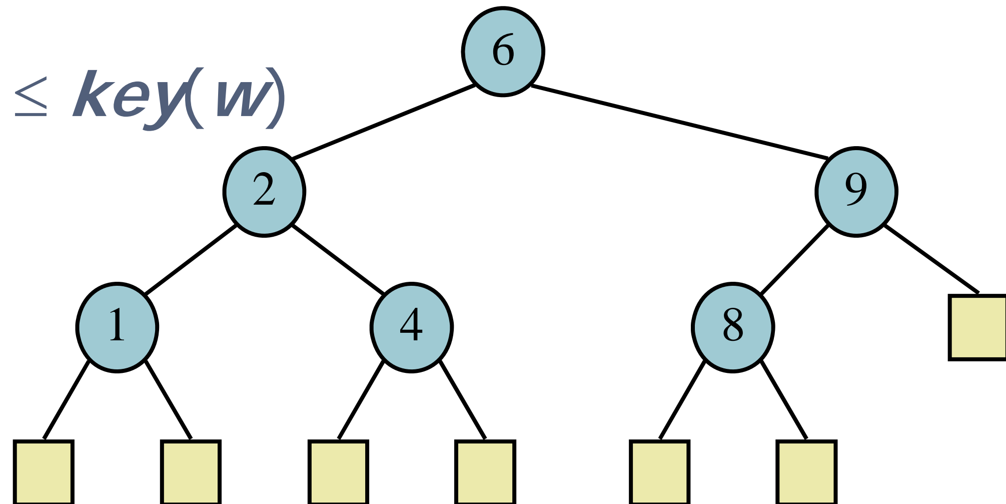
- MAKESET(S)
- INSERT(i,S)
- DELETE(k,S)
- FIND(k,S)
- JOIN(S_1,i,S_2): Replace the sets S_1 and S_2 by the new set $S=S_1 \cup \{i\} \cup S_2$, where:
  - For all items $j \in S_1$, $k(j) < i$
  - For all items $j \in S_2$, $k(j) > i$
- PASTE(S_1,S_2): Replace the sets S_1 and S_2 by the new set $S=S_1 \cup S_2$, where for all items $i \in S_1$, $j \in S_2$, $k(i)<k(j)$
- SPLIT(k,S)

Binary Search Tree

- A binary search tree (BST) is a binary tree storing key-element pairs at its internal nodes, satisfying the **BST property**:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. Then,

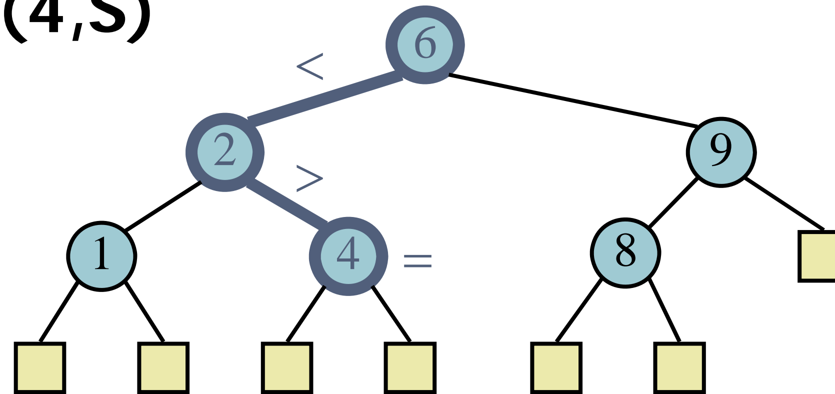
$$key(u) \leq key(v) \leq key(w)$$

- External nodes do not store items



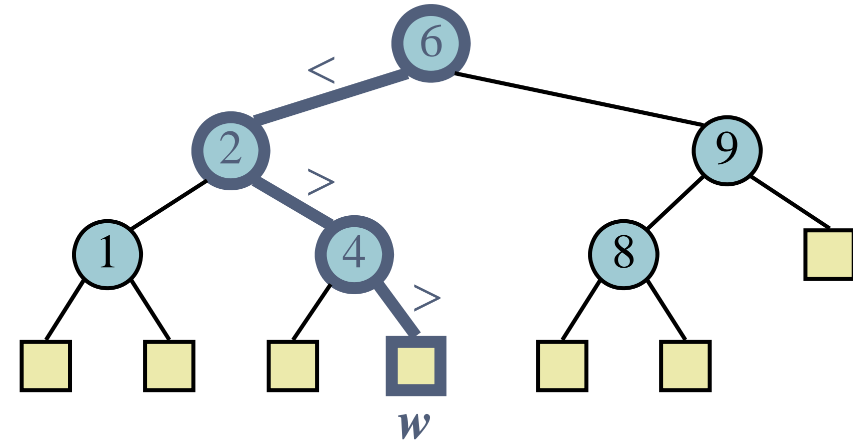
Binary Search Tree
Search

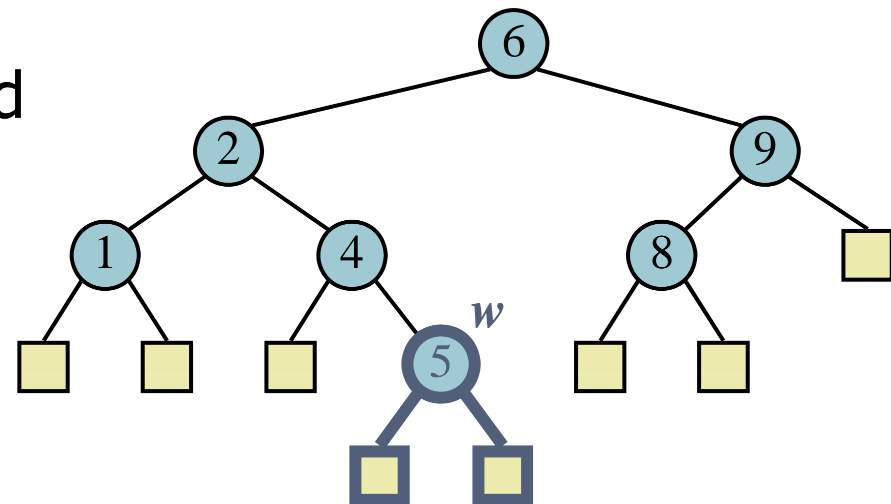
- Implements Dictionary operation **FIND(k,S)**
- To search for a key ***k***, we trace a downward path starting at the root
- The next node visited depends on the outcome of comparing ***k*** with the key of the current node
- Example: **FIND(4,S)**



Binary Search Tree *Insertion*

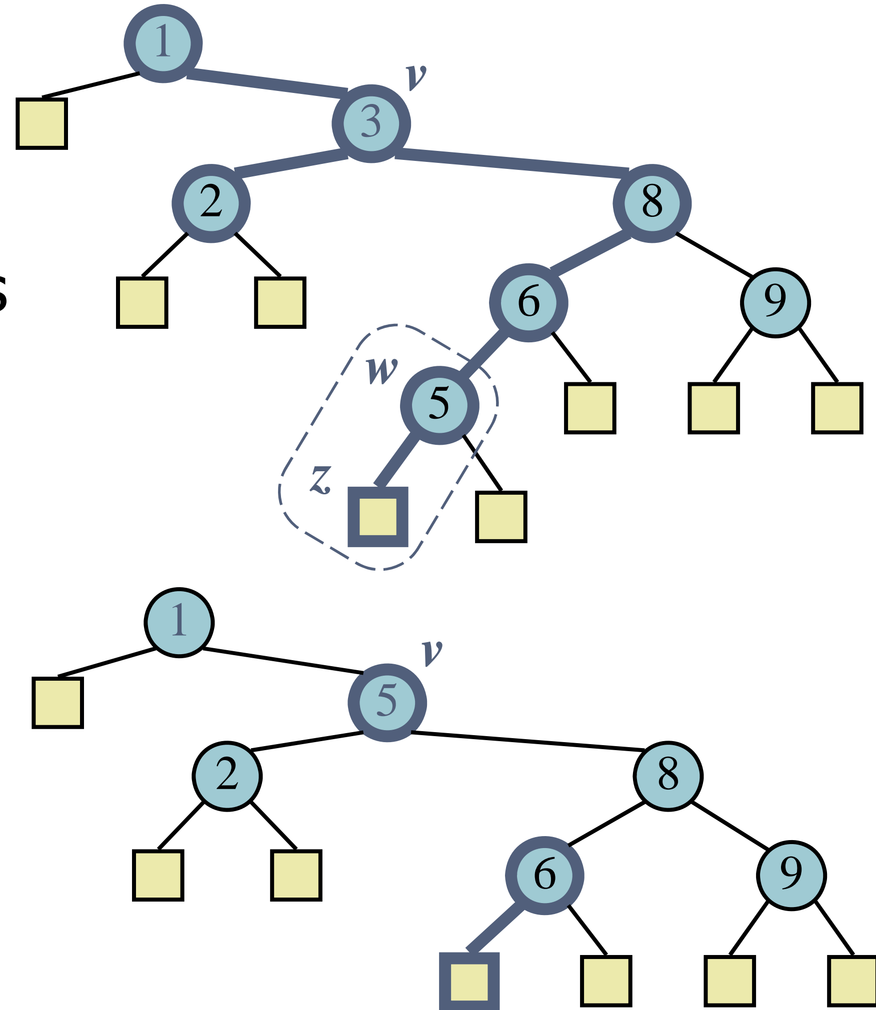
- To perform operation **INSERT(i,S)** we search for key k, i.e. **FIND(k,S)**
- Assume k is not already in the tree, and let let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: **INSERT(5,S)**



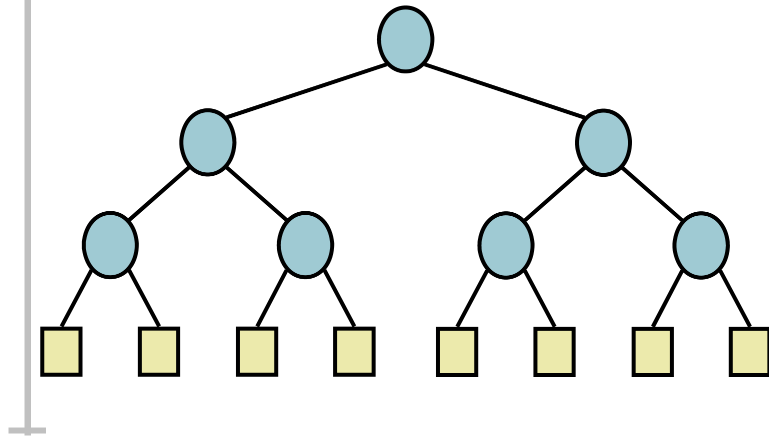


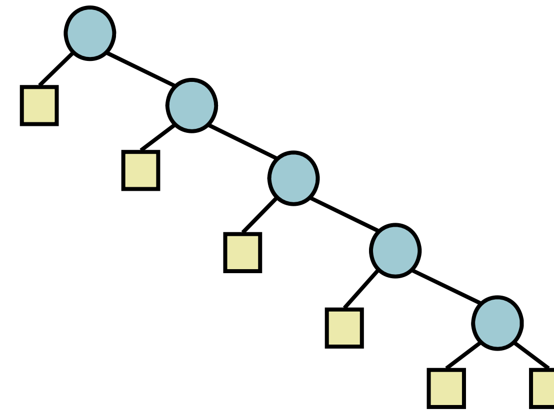
Binary Search Tree
Deletion

- To perform operation **DELETE(k,S)**, we search for key ***k***
- If the node v containing k has a leaf as one of its two children, replace v by its children that is internal node
- If neither of the children is a leaf, then let k' be the key value that is the predecessor of k. Replace v by node w, where k' is stored
- Example: **DELETE(3,S)**



Binary Search Tree *Performance*





Balanced Tree

Unbalanced Tree

Height = Θ(log n)

Each operation can be performed in time proportional to the height of the tree

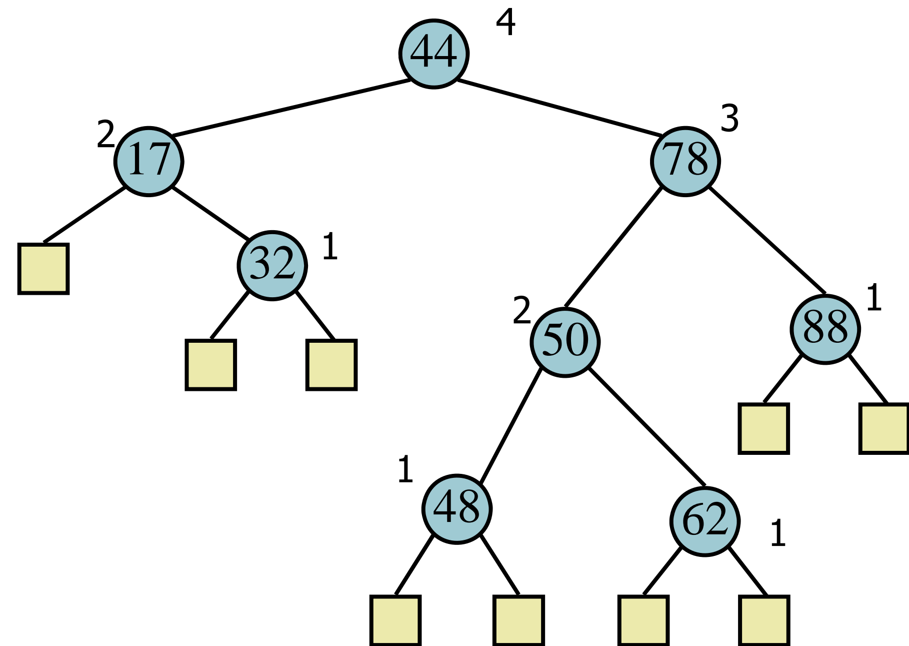
Keep our trees **balanced**

Balanced Trees

- **Balanced trees** maintain $O(\log n)$ height.
- Arbitrary binary trees have worst case $O(n)$ height.
- Approaches to balanced trees:
  - AVL trees
  - Splay trees (good *amortized* performance)
  - Weight balanced trees
  - Red-Black trees
  - B-trees
  - 2-3 trees
  - etc…

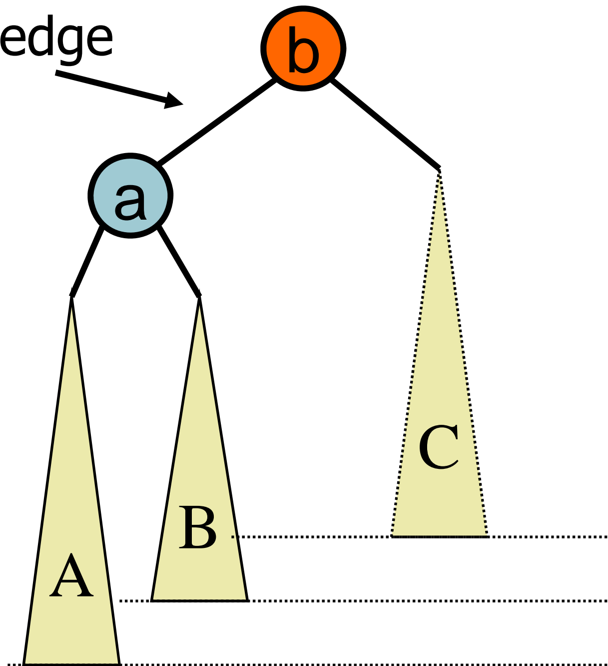
Balanced Trees Example AVL Trees

- An AVL tree is a binary tree with a **height-balance property**:
  for every internal node v, the heights of the children of v can differ by at most 1.
- AVL trees would guarantee $O(\log n)$ performance for all operations, because their height is $O(\log n)$

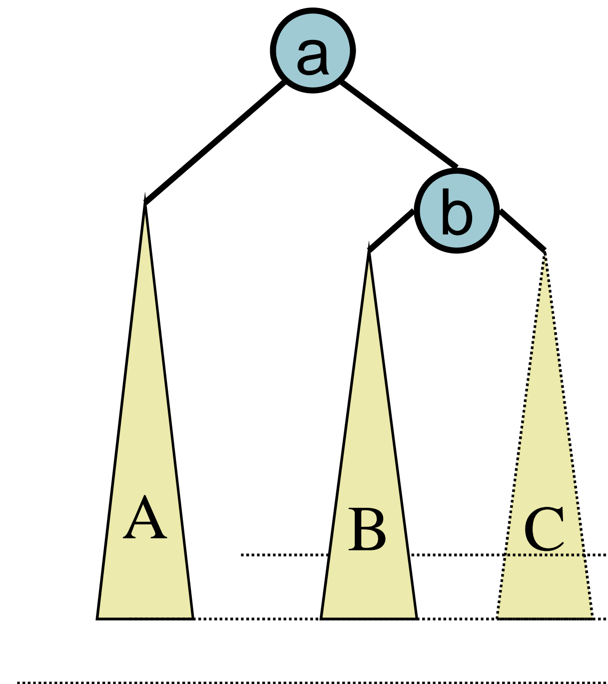


AVL Trees (single rotation)

- Before

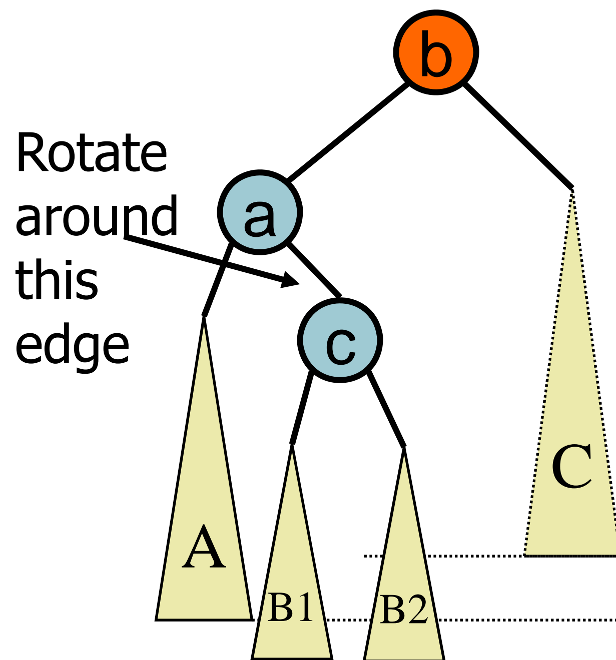


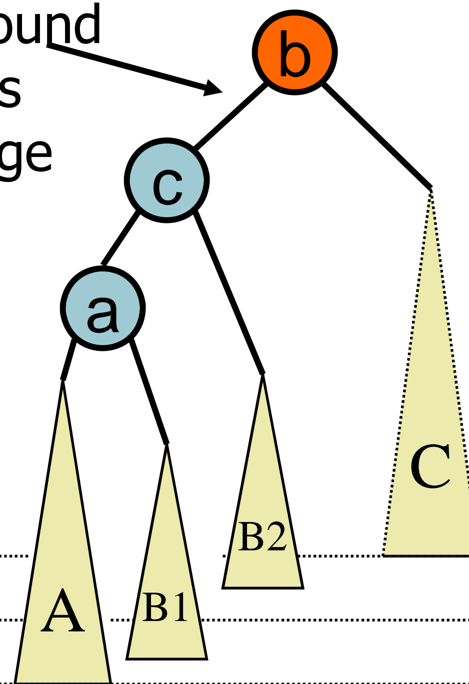
- After



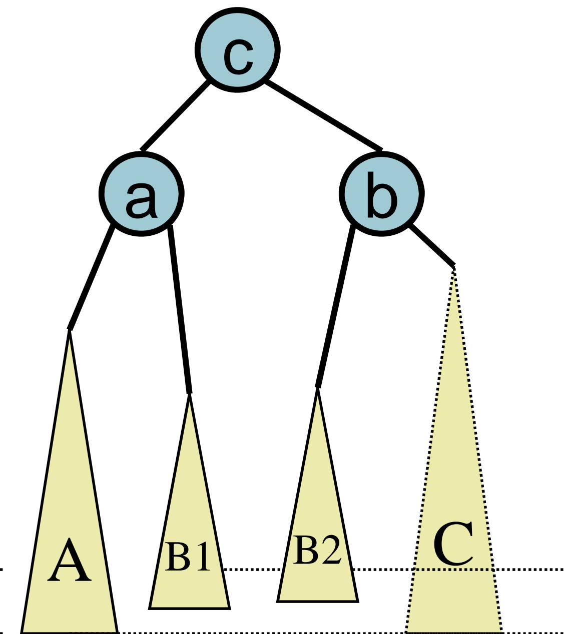
AVL Trees (double rotation)

- Before:





- After:



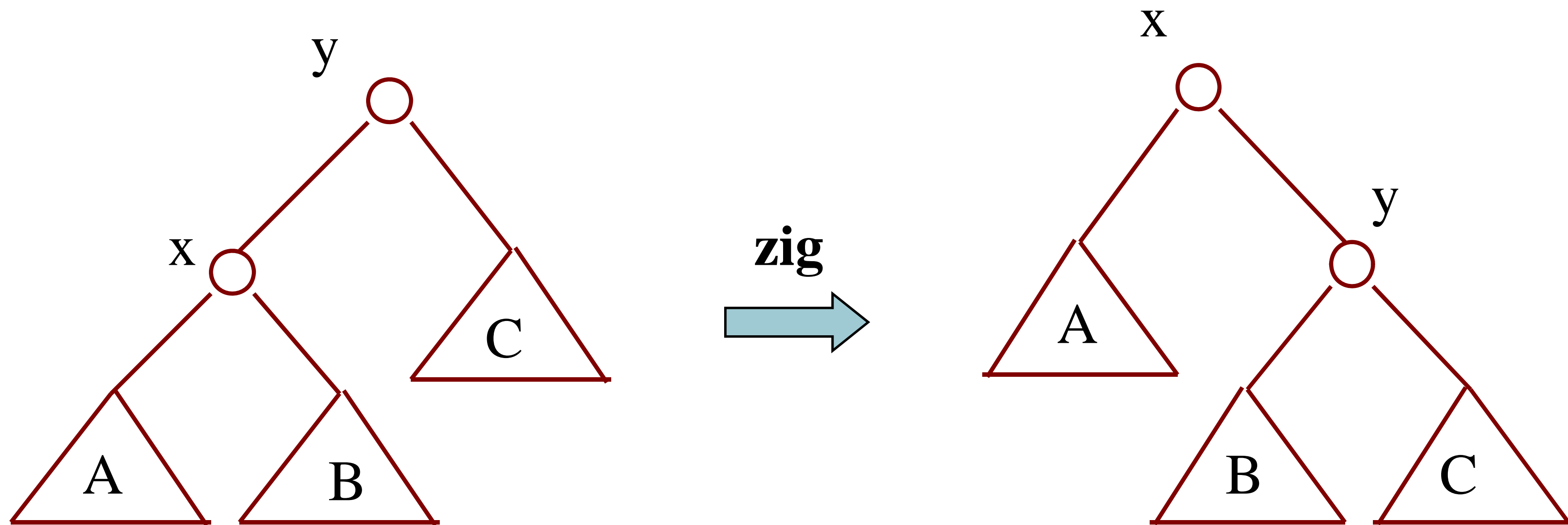
- Splay tree is a binary tree. Each operation follows a splay operation.
- The goal of a splay operation is to lift the final accessed node, says *x,* to the root.
- They have O(logn) **amortized** run time for
  - **Insert(), Search(),Delete()**
  - **Split()**
  - **Join()**

  } **These are expensive operations for AVLs**
- In **amortized analysis** we care for the cost of one operation if considered *in a sequence of n operations*

http://aleph0.clarku.edu/%7Eachou/cs160/examples/bst_animation/SplayTree-Example.html

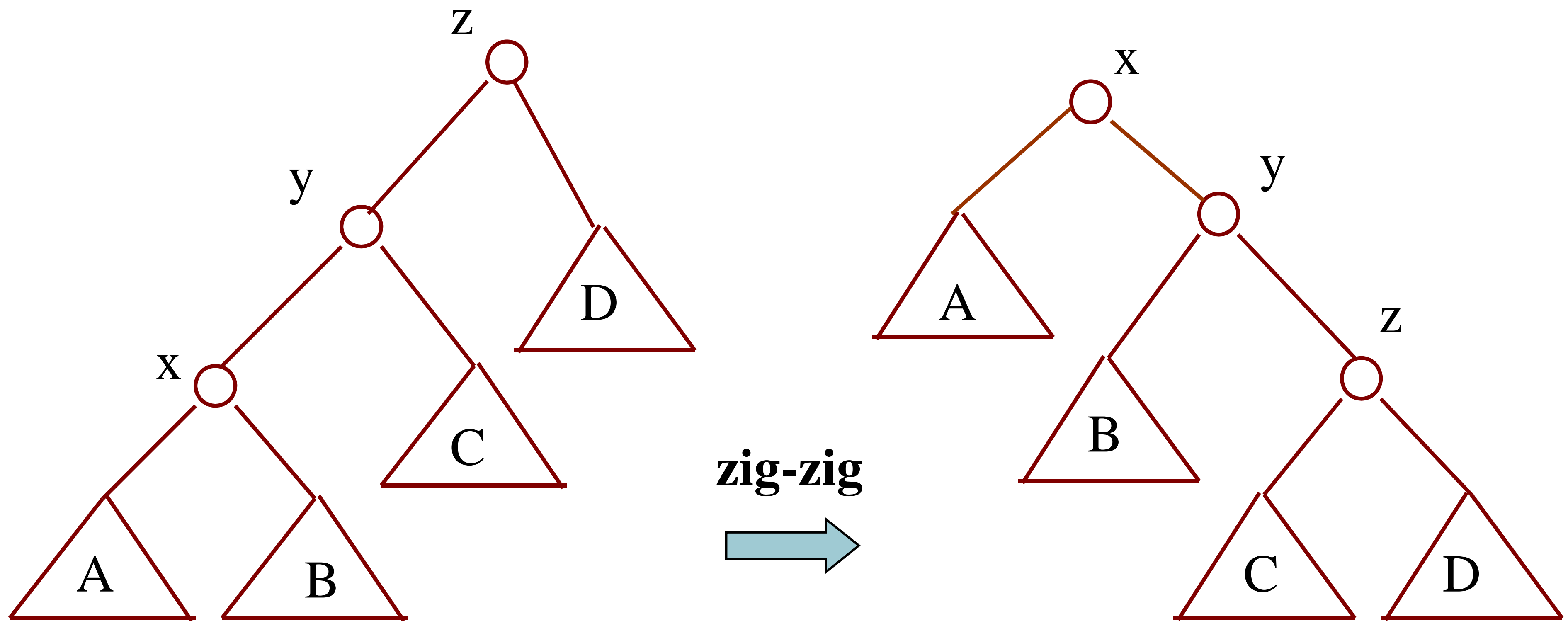
Splay Trees
Splay(x) operation

ZIG (1 rotation):



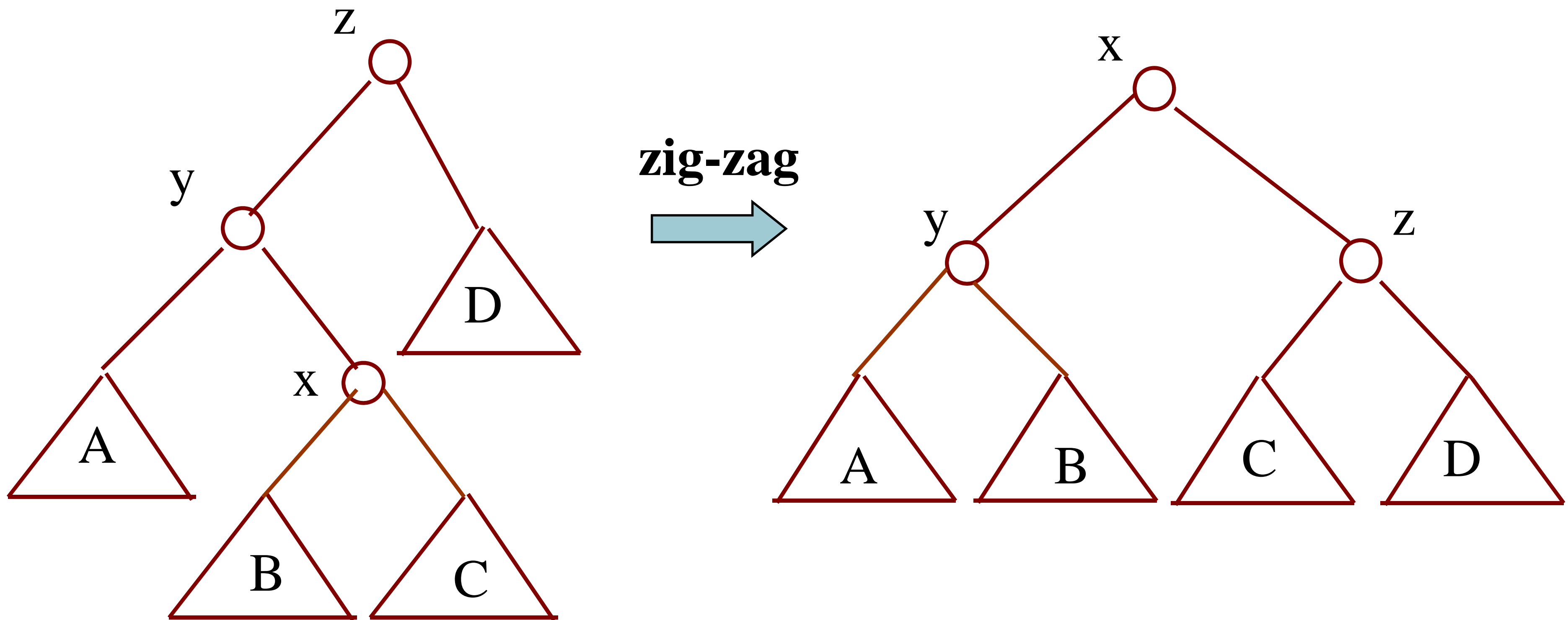
Splay Trees
Splay(x) operation

ZIG-ZIG (2 rotations):



Splay Trees
Splay(x) operation

ZIG-ZAG (2 rotations):



Splaying Examples

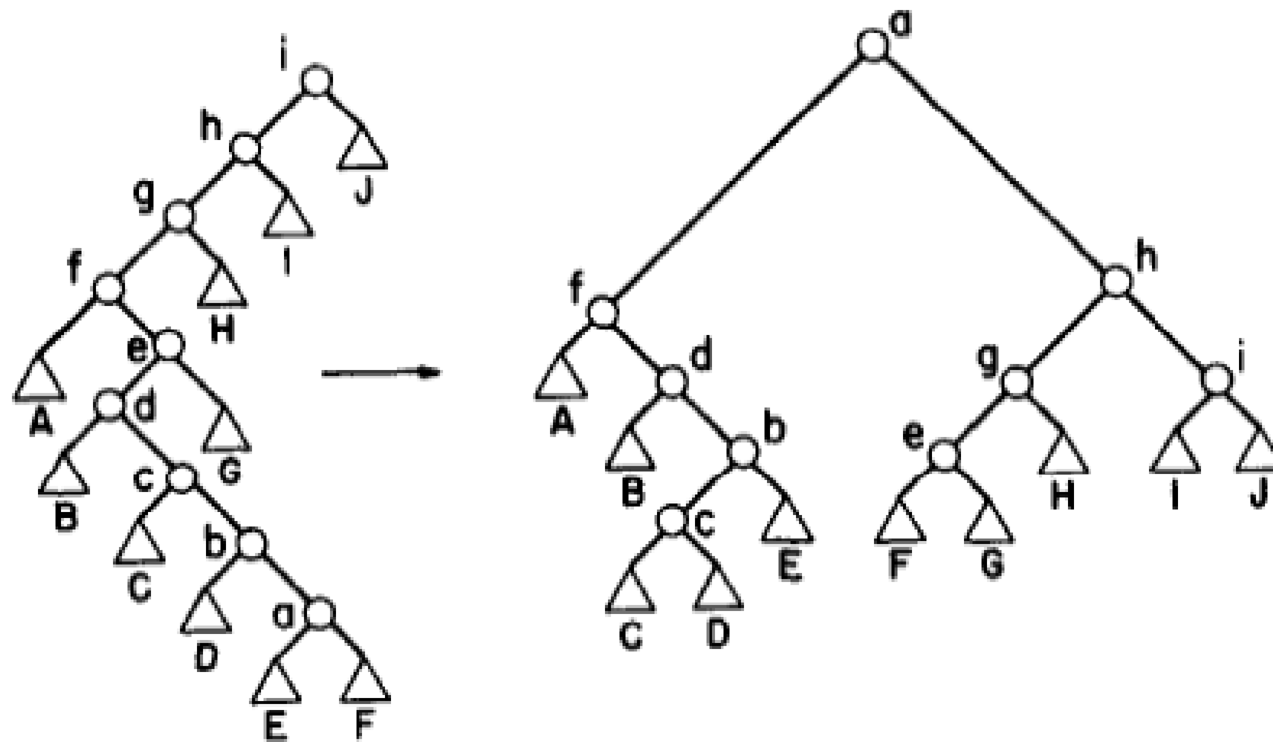


FIG. 4. Splaying at node *a*.

Splaying Examples (cont)

(a)

FIG. 5. Extreme cases of splaying. (a) All zig–zig steps. (b) All zig–zag steps.

Splaying Examples (cont)

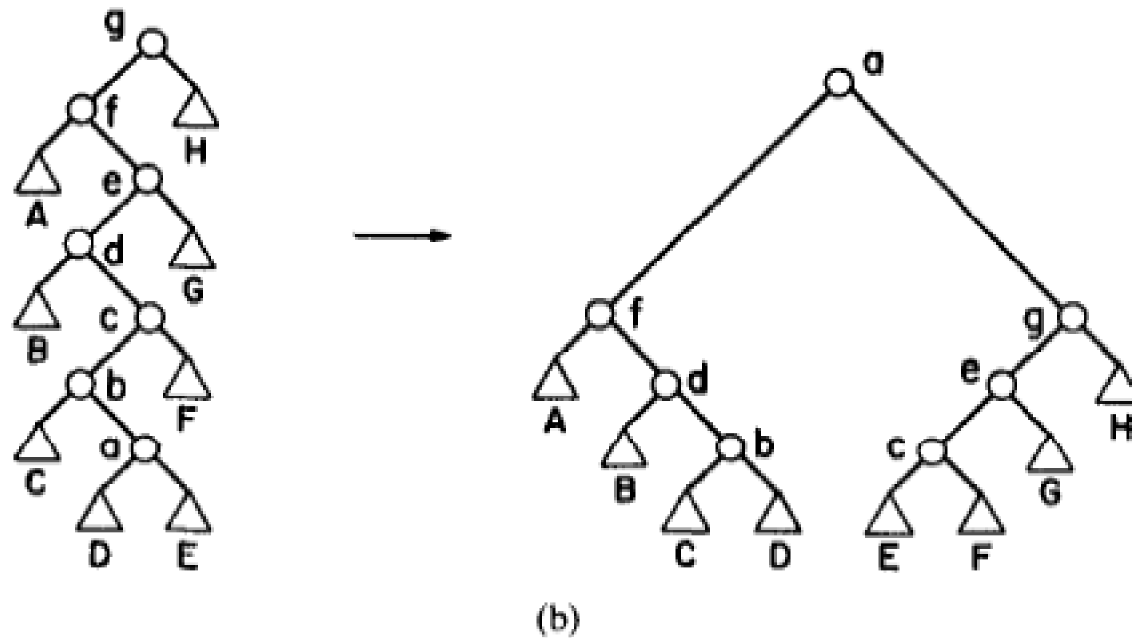


FIG. 5. Extreme cases of splaying. (a) All zig–zig steps. (b) All zig–zag steps.

Motivation For Randomized Data Structures

- Balanced trees are not as efficient as possible if the access pattern is **non uniform.**
- Balanced trees need **extra space** for storage of balance information.

Self-adjusting trees (Splay trees) overcome these problems but suffer from some drawbacks:

- They **restructure** the entire tree not only during updates, but also while performing simple **search** operations. (slow down in caching and paging environments).
- During any operation may perform a **linear number of rotations.**
- Amortized time bound → **NO** guarantee that every operation will run quickly. We have **bounds only on the total cost** of the operation

Motivation For Randomized Data Structures

- Given a Set of *n* items with distinct keys and insert them into a standard binary search tree using a *random* order, the expected height of the tree is O(log n)

  (Cormen, p 254-259)

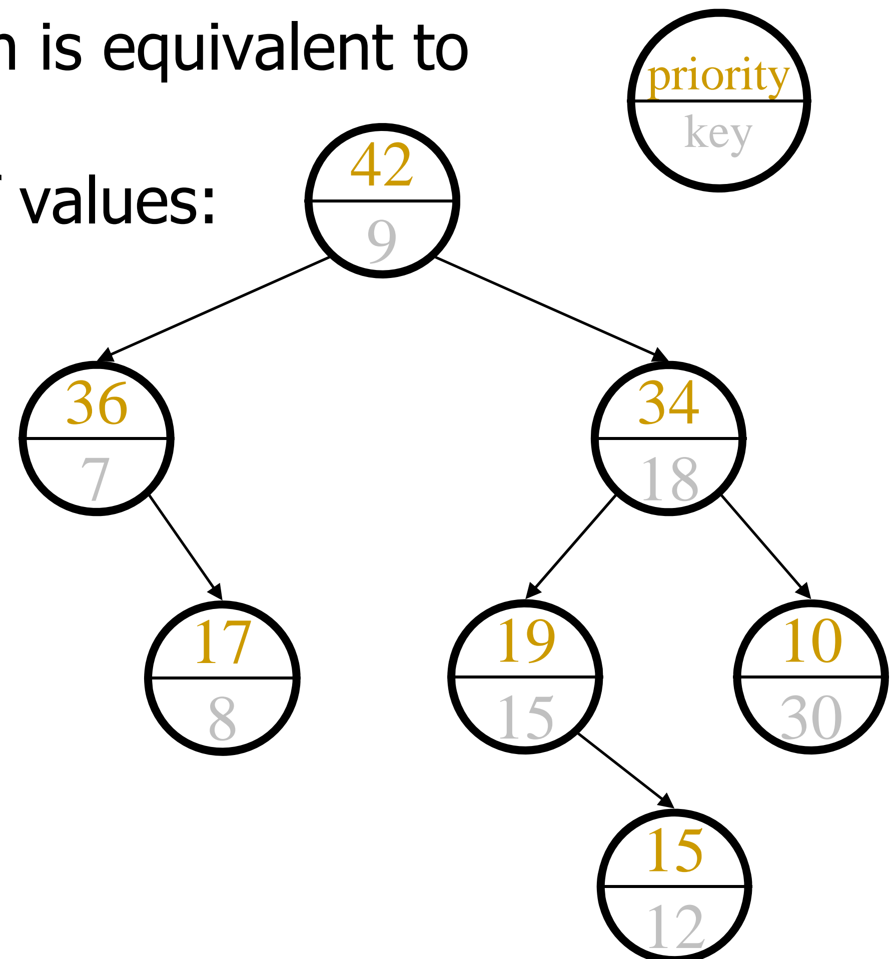
- The difficulty, however, is that in a dynamic setting we cannot really assume that the items come in a random order, and we cannot afford to wait until all items arrive before building the data structure.

→ Instead of randomizing the input (since we cannot!), consider randomizing the data structure

  - **Expected case** good behavior on any input
  - Balancing a data structure probabilistically is easier than explicitly maintaining the balance.

Random Treaps

- Treap is a data structure which is equivalent to binary search tree + heap.
- Each node u contains a pair of values:
  - A key k(u)
  - A priority p(u)
- Treap is a BST
  - binary tree property
  - search tree property
    (with respect to key values)
- Treap is also a heap
  - heap-order property
    (with respect to priorities)



- Properties:
  - Every node *x* has two associated values, *key*[*x*] and *priority*[*x*].
  - Key values and priorities are drawn from (possibly different) totally ordered universes and are distinct.
  - If node *y* is the left (respectively right) child of node *x*, then
    - $key[y] < key[x]$ (respectively $key[y] > key[x]$).
  - If node *y* is the child of node *x*, then
    - $priority[y] < priority[x]$.
  - The operations of standard heap and binary search tree hold.
    - Priority[] are used to perform heap operations.
    - Key[] are used to perform binary search tree operations.

http://aleph0.clarku.edu/%7Eachou/cs160/examples/bst_animation/Treap-Example.html

- ***Theorem 8.1***

  - Let $S = \{(k_1,p_1),\ldots, (k_n,p_n)\}$ be any set of key-priority pairs such that the keys and the priorities are distinct. Then, there exists a unique treap T(S) for it.

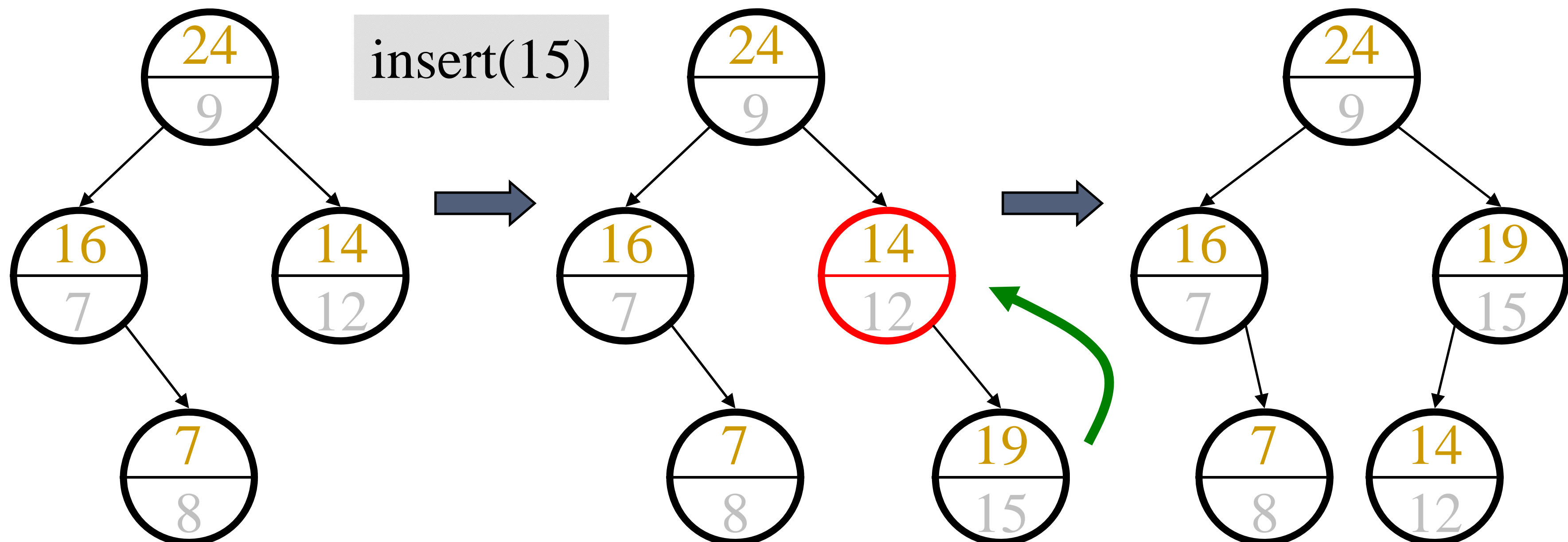
Proof: (by recursive construction)

- Theorem is true for n = 0 and n = 1
- Suppose that $n \geq 2$ and assume that (k_1,p_1) has the highest priority in S.
- Then a Treap for S can be constructed by putting item 1 at the root of T(S).
- A Treap for the items in S of key value **smaller** than k_1 can be constructed recursively and this is stored as the **left** sub-tree of item 1.
- Similarly, a Treap for the items in S of key value **larger** than k_1 is constructed recursively and this is stored as the **right** sub-tree of item 1.

□

Random Treaps
Insert

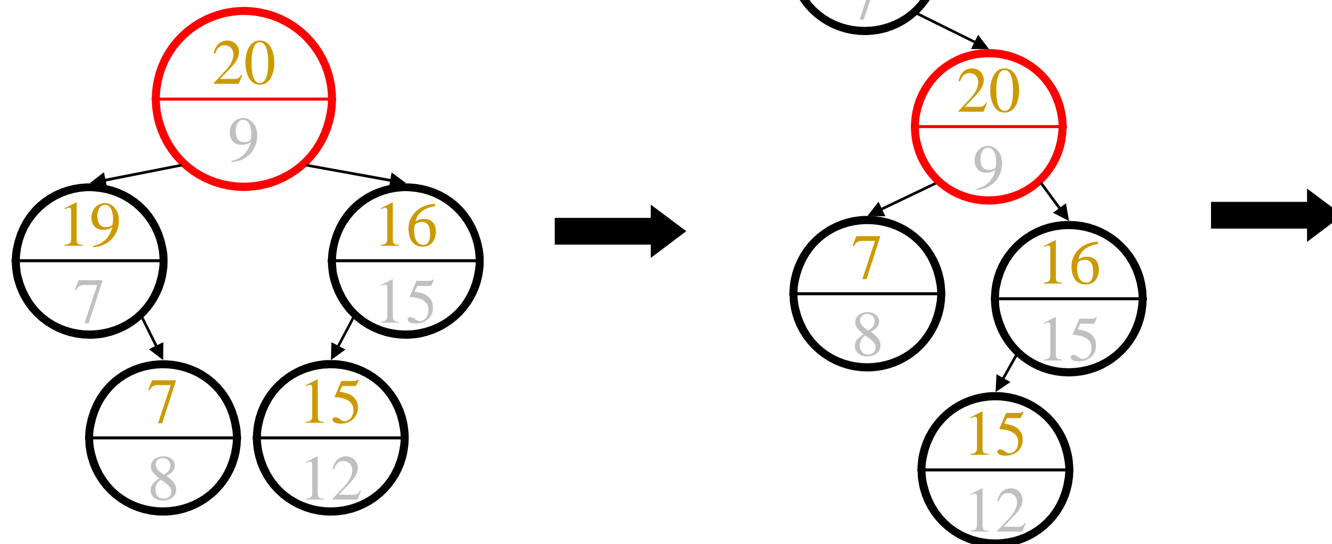
- Choose a random priority
- Insert as in normal BST
- Rotate up until heap order is restored



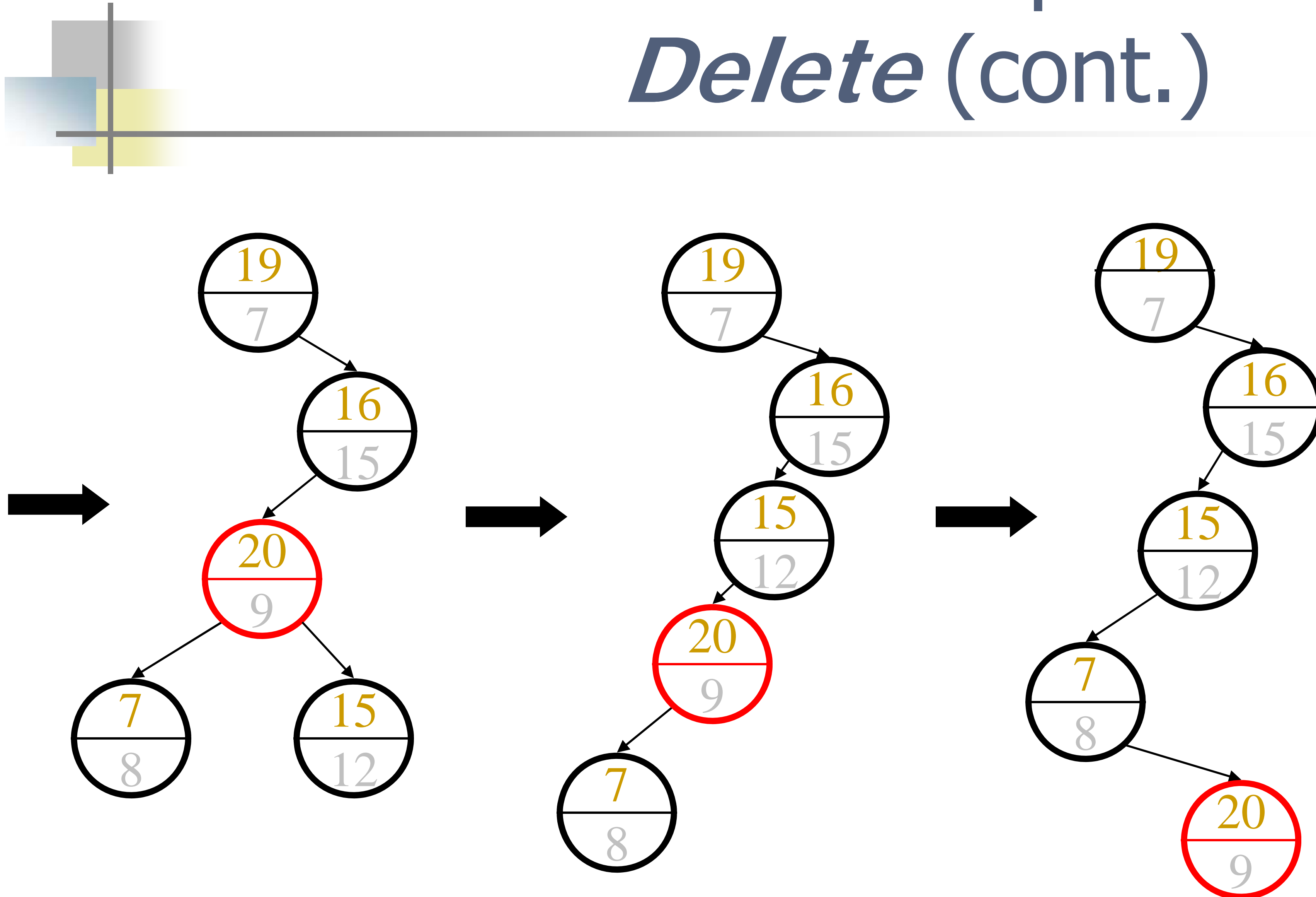
Random Treaps
Delete

- Reverse of an insertion
  - Find the key
  - Rotate the node containing the k downward until both children are leaves
  - Discard the node

delete(9)



Random Treaps
Delete (cont.)



- The shape of the tree underlying the treap is determined by the relative priorities of the key values.
- To solve the fundamental data structuring problem ➔ pick a good set of priorities for the items.
- Choose the priorities independently from some probability distribution D.
- Only restriction: D should ensure that with probability 1 the priorities are all distinct.
- In general, it suffices to use any continuous distribution, such as the uniform distribution U[0,1].

Need it later…

- Define the left spine of a tree as the path obtained by starting at the root and repeatedly moving to the left child until a leaf is reached.
- The right spine is defined simirarly

- The number of rotations during a DELETE operation on a node v is equal to the sum of the lengths of:
  - The left spine of the right sub-tree
  - The right spine of the left sub-tree

Random Treaps Mulmuley Games

The cast of characters is:

- A set $P = \{P_1,\ldots,P_p\}$ of players.
- A set $S = \{S_1,\ldots,S_s\}$ of stoppers.
- A set $T = \{T_1,\ldots,T_t\}$ of triggers.
- A set $B = \{B_1,\ldots,B_b\}$ of bystanders.

- The set $P \cup S$ is drawn from a totally ordered universe and for all i and j, $P_i < S_j$
- The sets are pairwise disjoint.

Mulmuley Games
Game A

- The game starts with the initial set of characters:

  $X = P \cup B.$
- Repeatedly sample from X *without* replacement, until the set X becomes empty.
- Each sample is a character chosen uniformly at random from the remaining pool in X
- Let random variable V denote the number of samples in which a player P_i is chosen such that P_i is larger than all previously chosen players.
- We define the value of the game:

  $A_p = \mathbf{E}[V]$

Mulmuley Games
Game A

$H_k \approx \ln k$

- ***Lemma 8.2***: For all $p \geq 0$, $A_p = H_p$

Proof:

- Assume that P is ordered as $P_1 > P_2 > \ldots > P_p$
- *Key observation*: value of the game is not influenced by the number of bystanders ➔ assume that b = 0
- We will use the following property of Harmonic numbers:

$$\sum_{k=1}^{n} H_k = (n+1)H_{n+1} - (n+1)$$

Mulmuley Games
Game A

Proof (cont):

- Conditional upon the first random sample being a particular Player P_i, the expect value of the game is:
  - $A_p = 1 + A_{i-1}$
- Since i is uniformly distributed over the set {1,…,p}:

$$A_p = \sum_{i=1}^{p} \frac{1 + A_{i-1}}{p} = 1 + \sum_{i=1}^{p} \frac{A_{i-1}}{p}$$

- Upon rearrangement, using the fact that $A_0 = 0$:

$$\sum_{i=1}^{p-1} A_i = pA_p - p$$

□

$$\sum_{k=1}^{n} H_k = (n+1)H_{n+1} - (n+1)$$

Mulmuley Games
Game C

- The game starts with the initial set of characters:

  $X = P \cup B \cup S.$
- The process is exactly the same as that in game A.
- Treat stoppers as players **but** when a stopper is chosen for the first time, the game **stops**.
- Since all players are smaller than all stoppers, we will always get a contribution of 1 to the game value from the first stopper.
- The value of the game is (where V is defined as in game A): $C_p^s = \mathbf{E}[V + 1] = 1 + \mathbf{E}[V]$

Mulmuley Games
Game C

- ***Lemma 8.3***: For all p,s≥0, $C_p^s = 1 + H_{s+p} - H_s$

Proof:

- As before, assume that:
  - P is ordered as $P_1 > P_2 > \ldots > P_p$
  - Number of bystanders is 0.
- If the first sample is a stopper, then the game value is 1. (with probability s/(s+p))
- If the first sample is a player P_i, then the game value is (with probability 1/(s+p) for each player):

$$C_p^s = 1 + C_{i-1}^s$$

Mulmuley Games

Game C

$$\sum_{k=1}^{n} H_k = (n+1)H_{n+1} - (n+1)$$

Proof (cont):

- Noting the given probabilities:

$$C_p^s = \left(\frac{s}{s+p} \times 1\right) + \left(\frac{1}{s+p} \times \sum_{i=1}^{p} \left(1 + C_{i-1}^s\right)\right)$$

- Upon rearrangement, using the fact that $C_0^s = 1$:

$$C_p^s = \frac{s+p+1}{s+p} + \frac{\sum_{i=1}^{p-1} C_i^s}{s+p} \Leftrightarrow$$

$$\sum_{i=1}^{p-1} C_i^s = (s+p) \cdot C_p^s - (s+p+1) \Rightarrow$$

$$C_p^s = 1 + H_{s+p} - H_s$$

□

- The game starts with the initial set of characters:

  $X = P \cup B \cup T$ (Game D).

  $X = P \cup B \cup S \cup T$ (Game E).

- The process is exactly the same as that in games A,C.
- The role of the triggers is that the counting process begins **only after** the first trigger has been chosen. i.e. a player or a stopper contributes to V only if it is sampled after a trigger and before any stopper (and of course it is larger than all previously chosen players).

Mulmuley Games
Games D & E

- ***Lemma 8.4***: For all p,t≥0

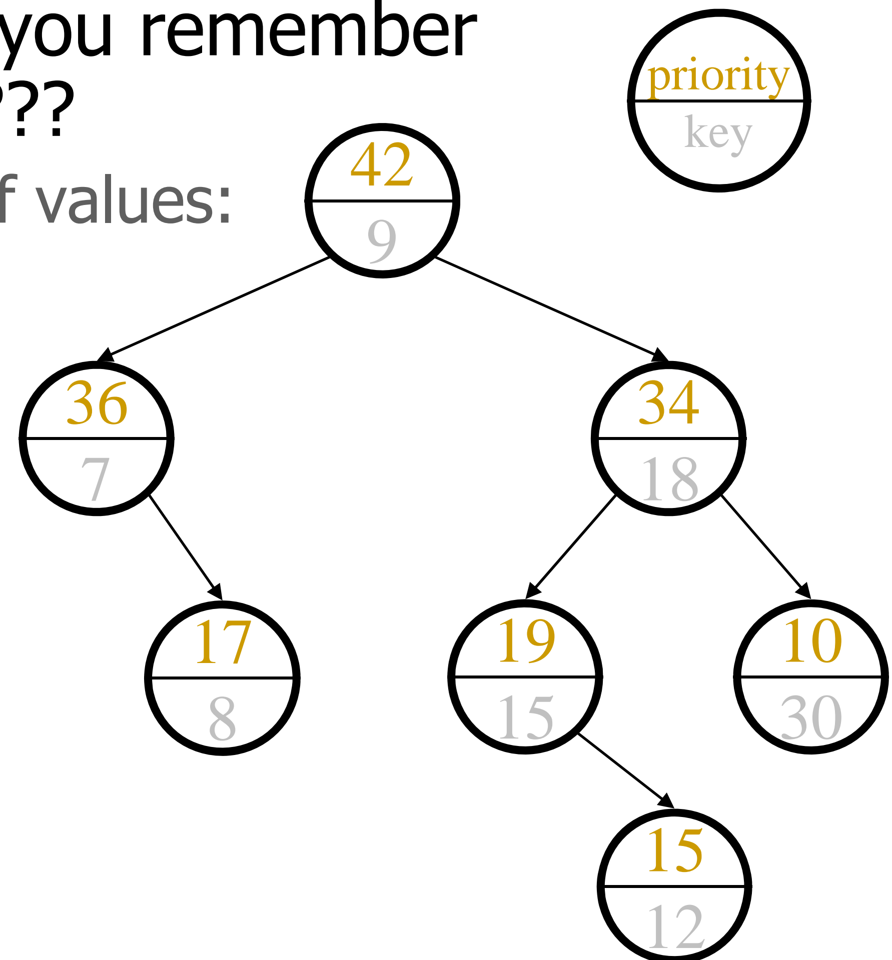
$$D_p^t = H_p + H_t - H_{p+t}$$

- ***Lemma 8.5***: For all p,s,t≥0

$$E_p^{s,t} = \frac{t}{s+t} + \left(H_{s+p} - H_s\right) - \left(H_{s+p+t} - H_{s+t}\right)$$

Analysis of Treaps

- After all these proofs, do you remember anything about Treaps?????
- Each node u contains a pair of values:
  - A key k(u)
  - A priority p(u)
- Treap is a BST
  - binary tree property
  - search tree property
    (with respect to key values)
- Treap is also a heap
  - heap-order property
    (with respect to priorities)



Analysis of Treaps

Important property of random Treaps

Memoryless property:

- Consider a Treap obtained by inserting the elements of a set S into an initially empty treap
- Since the random priorities of elements are chosen independently ➔ we can assume that the priorities are chosen **before** the insertion process initiated.
- Once the properties have been fixed, Theorem 8.1 implies that the treap T is **uniquely** determined.
- *The order in which the elements are inserted does not affect the structure of the tree.*
    - Thus, we can assume that the elements of S are *inserted* into T in the order of *decreasing priority*.

Analysis of Treaps

- ***Lemma 8.6***: Let T be a random treap for a set S of size n. For an element x ∈ S having rank k,

  $$\mathbf{E}[\text{depth}(x)] = H_k + H_{n-k+1} - 1$$

➢ Given lemma establishes that the expected depth of the element of rank k in S is

  $$O(\log k + \log(n - k + 1))$$

  Which is always *O(logn)*.

Analysis of Treaps

Proof: ($E[depth(x)] = H_k + H_{n-k+1} - 1$)

- Define the sets:
  - $S^- = \{y \in S \mid y <= x\}$
  - $S^+ = \{y \in S \mid y \geq x\}$
- Since x has rank k
  - $|S^-| = k$
  - $|S^+| = n-k+1$
- Denote by $Q_x \subseteq S$ the set of elements that are stored at nodes on the path from the root of T to the node containing x, i.e. the ancestors of x.
- Let Q_x^- denote $S^- \cap Q_x$
- Let Q_x^+ denote $S^+ \cap Q_x$

Proof (cont.): $(\mathbf{E}[depth(x)] = H_k + H_{n-k+1} - 1)$

- We will establish that:

  $E[|Q_x^-|] = H_k$

- By symmetry follows that:

  $E[|Q_x^+|] = H_{n-k+1}$

- Since $Q_x^- \cap Q_x^+ = \{x\}$, the expected path from the root to x is:

  $H_k + H_{n-k+1} - 1$

Analysis of Treaps

Proof (cont.): ($E[|Q_x^-|] = H_k$)

- By the memoryless assumption, any ancestor $y \in Q_x^-$ of the node x:
  - Has been inserted prior to x
  - $p_y > p_x$
- Since y < x, x lies in the right subtree of y.
- We claim that all elements z such that y < z < x lie in the right sub-tree of y.

➔ y is an ancestor of every node containing an element of value between y and x.

➔ *An element $y \in S^-$ is an ancestor of x, or a member of Q_x^- iff it was the largest element of S^- in the treap at the time of its insertion.*

Analysis of Treaps

Proof (cont.): $(E[|Q_x^-|] = H_k)$

- The order of insertion is determined by the order of the priorities, and the latter is uniformly distributed
- → the order of insertion can be viewed as being determined by uniform sampling without replacement from the pool S.
- So, the distribution of $|Q_x^-|$ is the same as that of the value of Mulmuley Game A when:
  - $P = S^-$
  - $B = S \setminus S^-$
- Since $|S^-| = k$
  - $E[|Q_x^-|] = H_k$

□

Analysis of Treaps

- ***Lemma 8.7***:
  Let T be a random treap for a set S of size n. For an element x ∈ S of rank k,

$$E[R_x] = 1 - \frac{1}{k}$$

And

$$E[L_x] = 1 - \frac{1}{n - k + 1}$$

➢ Given lemma helps us bound the expected number of rotations required during an update operation.

Proof: (*E[Rx]=1-1/k*)

- Prove only the $E[R_x]$ and the $E[L_x]$ follows by symmetry since the rank of x becomes n-k+1 if we invert the total order underlying key values.
- Demonstrate that the distribution of R_x is the same as that of the value of game D with the choice of characters:
  - $P = S^- \backslash \{x\}$ ➔ $|P| = k-1$
  - $T = \{x\}$ ➔ $|T| = 1$
  - $B = S^+ \backslash \{x\}$ ➔ $|B| = n-k$
- Lemma 8.4 implies that:

$$E[R_x] = D_{k-1}^{1} = H_{k-1} + H_1 - H_k = 1 - \frac{1}{k}$$

Analysis of Treaps

Proof (cont.):

- Memoryless property ➔ an element z < x lies on the right spine of the left sub-tree of x *if and only if*
  - z is inserted after x.
  - All elements between z and x are inserted after z.

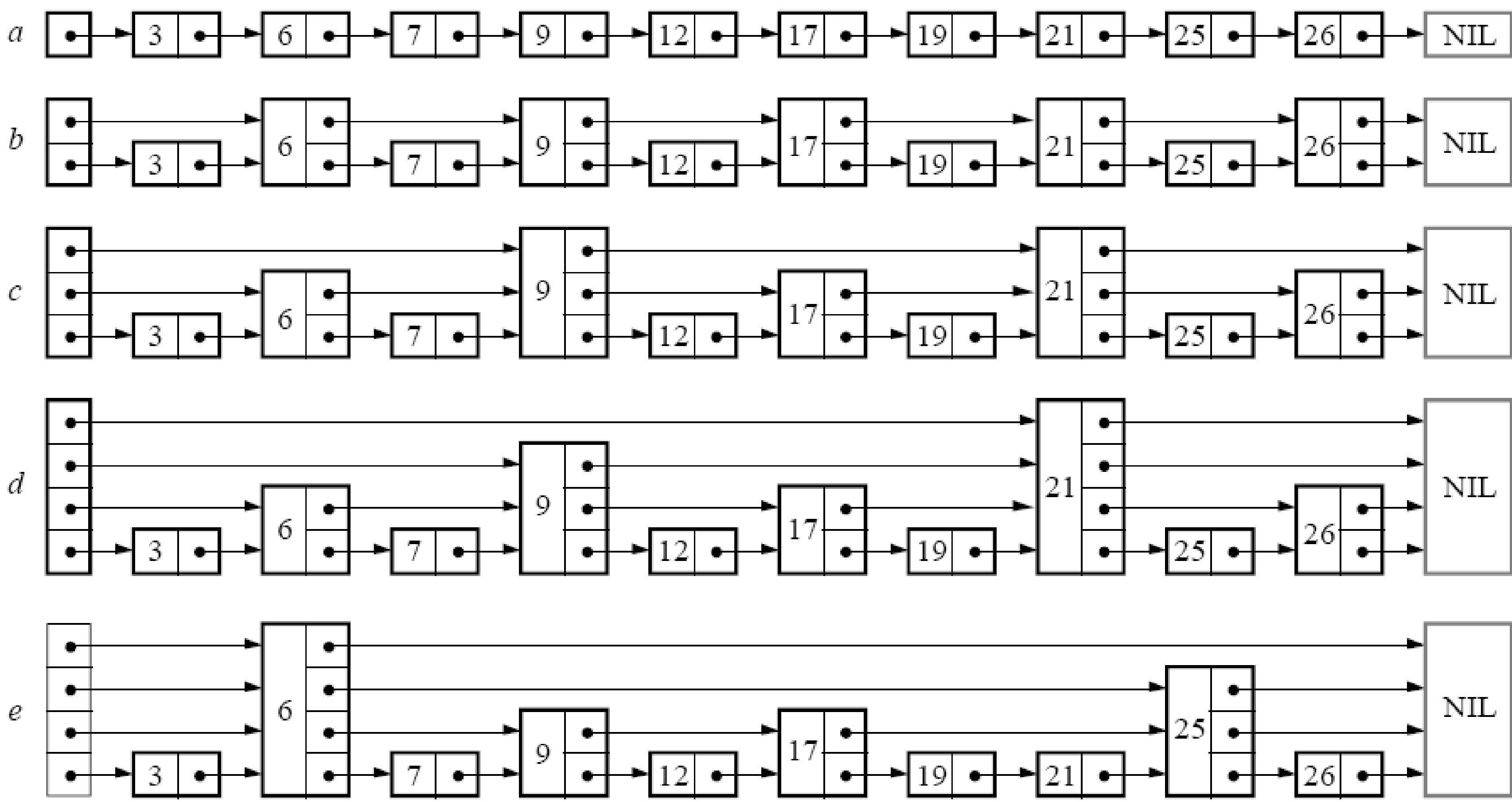
□

Analysis of Treaps

- ***Theorem 8.8*: Let T be a random treap for a set S of size n.**

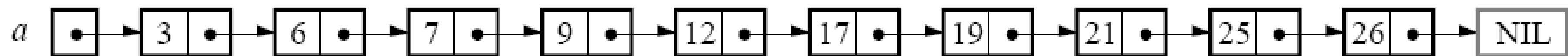
1. The *expected time* for a FIND,INSERT, or DELETE operation on T is **O(logn)**.
2. The *expected number of rotations* required during an INSERT or DELETE operation is at most **2**.
3. The *expected time* for a JOIN, PASTE or SPLIT operation involving sets S_1 and S_2 of sizes n and m, respectively, is **O(logn + logm)**

Skip Lists

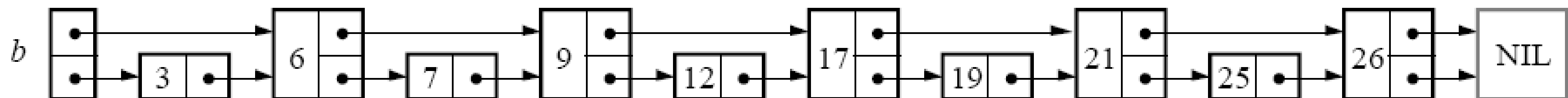


Skip Lists
Intuition

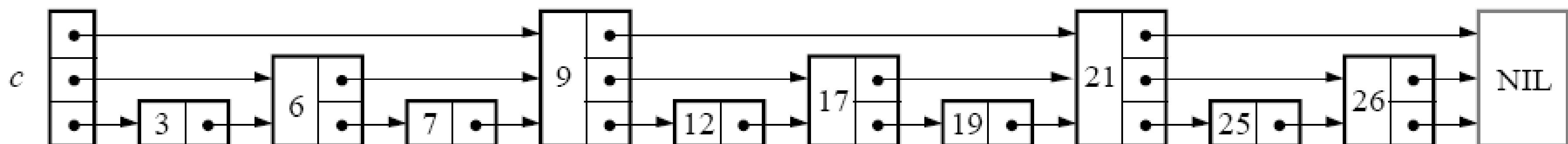
- We might need to examine every node of the list when searching a linked list. O(n)



- If the list is stored in sorted order and every other node of the list also has a pointer to the node two ahead it in the list, we have to examine no more than $\lceil n/2 \rceil + 1$ nodes

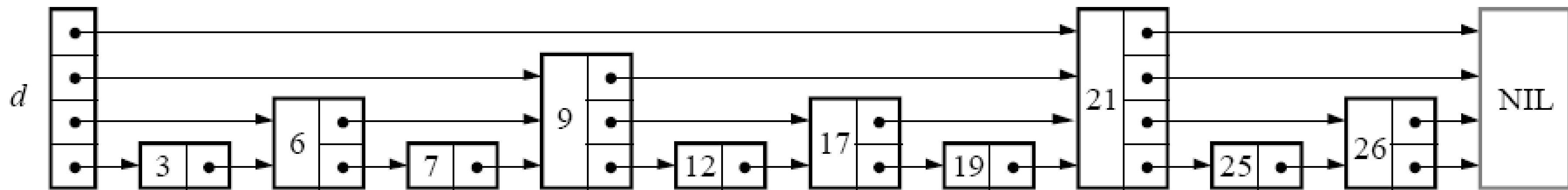


- Also giving every fourth node a pointer four ahead requires that no more than $\lceil n/4 \rceil + 2$ nodes be examined.



Skip Lists
Intuition

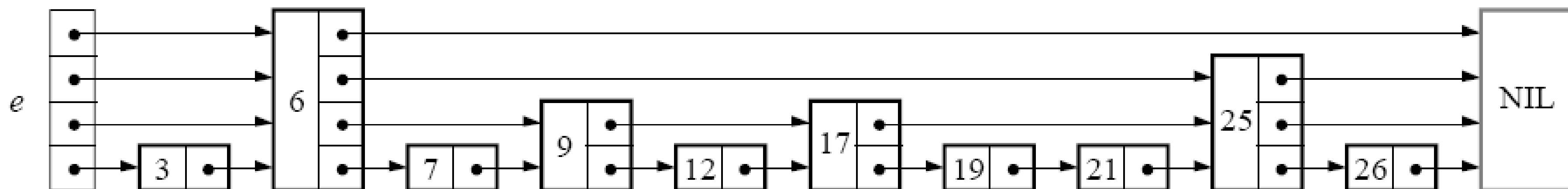
- If every $(2^i)^{th}$ node has a pointer 2^i nodes ahead, the number of nodes that must be examined can be reduced to $\lceil \log_2 n \rceil$ while only doubling the number of pointers.



- Skip Lists let you do a **binary search** in a list. In effect balancing is perfect.
- Maintaining such balance is very expensive for Insert/Delete operations ➔Reorganizing the list is O(N).

Skip Lists
Intuition

- If every $(2^i)^{th}$ node has a pointer 2^i nodes ahead, then levels of nodes are distributed in a simple pattern:
  - 50% are level 1, 25% are level 2, 12.5% are level 3 and so on.
- What would happen if the levels of nodes were chosen randomly, but in the same proportions? A node's i^{th} forward pointer, instead of pointing 2^{i-1} nodes ahead, points to the next node of level *i* or higher.



- Insertions or deletions would require only **local** modifications; the level of a node, chosen randomly when the node is inserted, need never change.

Random Skip Lists

- Consider a set $S = \{x_1 < x_2 < .. < x_n\}$ drawn from a totally ordered universe.
- A *leveling* with r levels of an ordered set S is a sequence of nested subsets (called *levels*)

  $$L_r \subseteq L_{r-1} \subseteq \ldots \subseteq L_2 \subseteq L_1$$

  such that $L_r = \emptyset$ and $L_1 = S$
- Given an ordered set S and a leveling for it, the level of any element $x \in S$ is defined as:

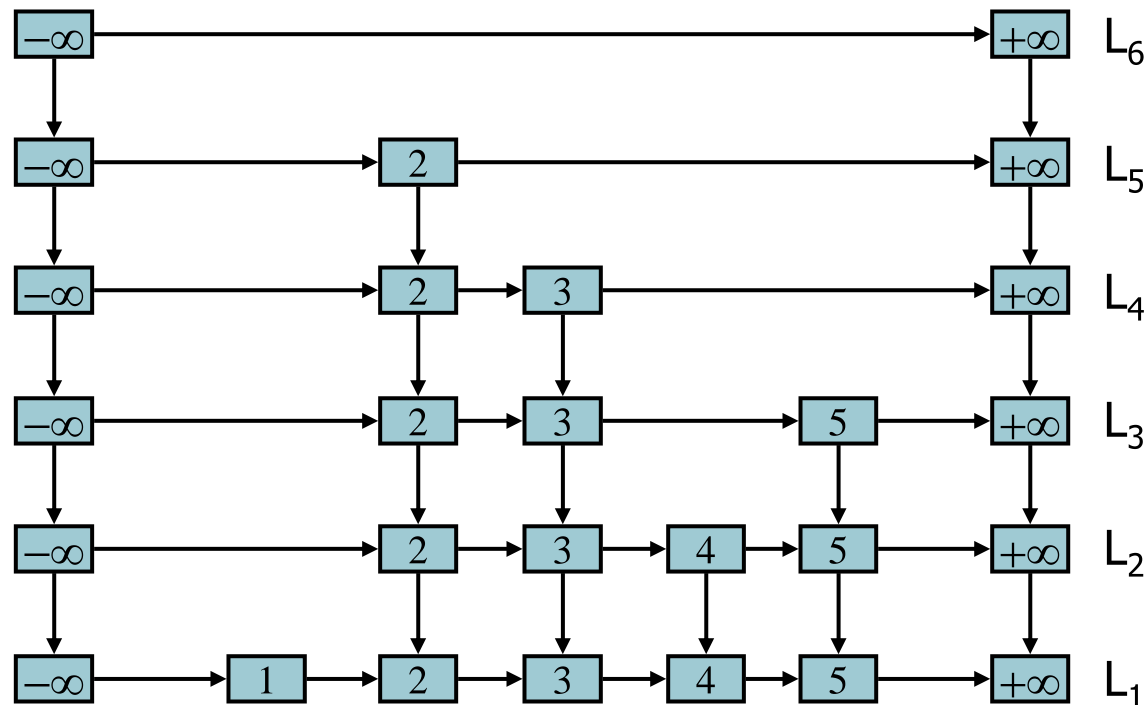
  $$l(x) = \max\{i \mid x \in L_i\}$$

Random Skip Lists

- Given any leveling of the set S, we can define an ordered list data structure:
  - Assume that two special elements $-\infty$ and $+\infty$ belong to each of the levels.
  - $-\infty$ is smaller than all elements in S and $+\infty$ is larger than all elements in S.
  - Both $-\infty$ and $+\infty$ are of level r.
  - The level L_1 is stored in a sorted linked list and each node x in this list has a pile of $l(x)-1$ nodes sitting above it.

Random Skip Lists

- Skip list for the set S={1,2,3,4,5} and for the leveling:
  - $L_6 = \emptyset$, $L_5 = \{2\}$, $L_4 = \{2,3\}$, $L_3 = \{2,3,5\}$, $L_2 = \{2,3,4,5\}$, $L_1=\{1,2,3,4,5\}$



Random Skip Lists

- An *interval* at level i is the set of elements of S spanned by a specific horizontal pointer at level i.
- The sequence of levels L_i can be viewed as successively coerser partitions of S into a collection of intervals:

$$L_1 = [-\infty,1] \cup [1,2] \cup [2,3] \cup [3,4] \cup [4,5] \cup [5,+\infty]$$

$$L_2 = [-\infty,2] \cup [2,3] \cup [3,4] \cup [4,5] \cup [5,+\infty]$$

$$L_3 = [-\infty,2] \cup [2,3] \cup [3,5] \cup [5,+\infty]$$

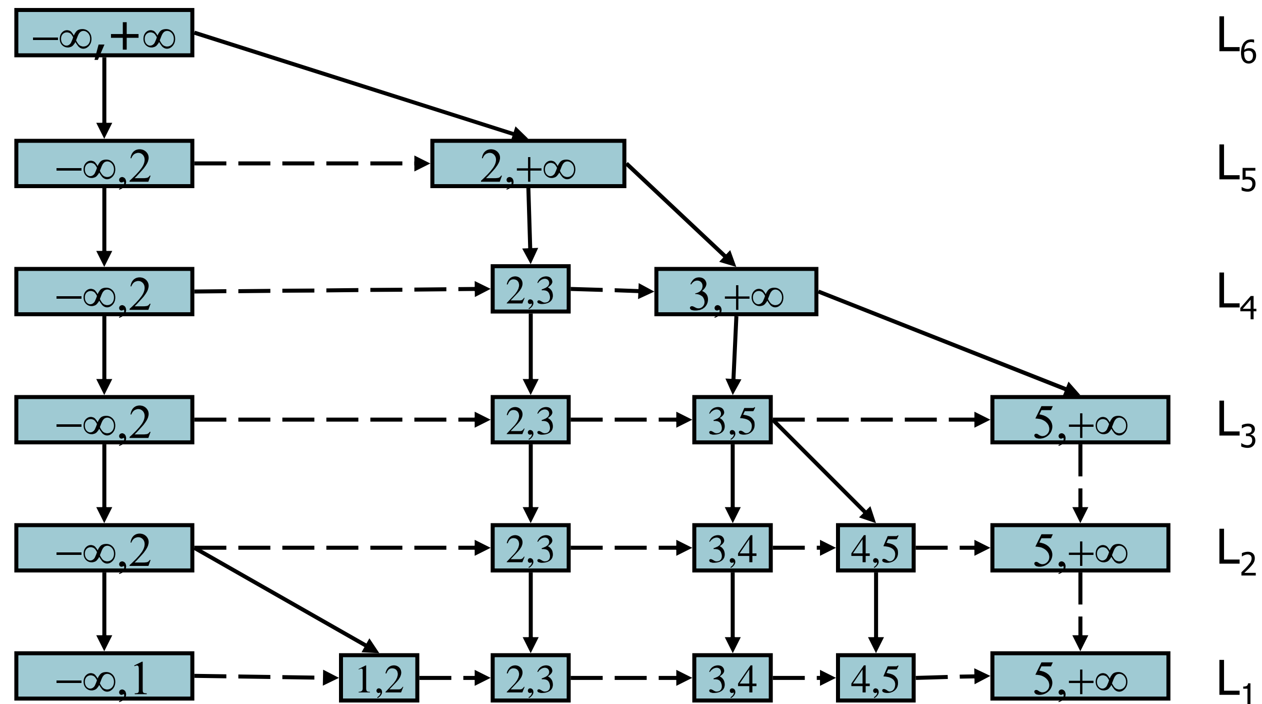
$$L_4 = [-\infty,2] \cup [2,3] \cup [3,+\infty]$$

$$L_5 = [-\infty,2] \cup [2,+\infty]$$

$$L_6 = [-\infty,+\infty]$$

Random Skip Lists

- The interval partition structure is more conveniently viewed as a tree
- If an interval J at level i+1 contains as a subset an interval I at level i, then node J is parent of node I in the tree.
- For an interval I at level i+1, c(I) denotes the number of children it has at level i.



Random Skip Lists

- Consider an element y, which is not necessarily a member of S.
- Define $I_j(y)$ as the interval at level j that contains y.
- We can now view the nested sequence of intervals $I_r(y) \subseteq I_{r-1}(y) \subseteq \ldots \subseteq I_1(y)$ containing y as a root-leaf path in the tree representation of the skip list.
- Choose a *random leveling* ➔ will show that there is a high probability that the search tree corresponding to a random skip list is balanced.

Random Skip Lists

A *random leveling* of the set S is defined as follows:

- Given the choice of level L_i, the level L_{i+1} is defined by independently choosing to retain each element $x \in L_i$ with probability ½.
- The process starts with $L_1=S$ and terminates when a newly constructed level is empty.
- Alternate view:
  - Let the levels $l(x)$ for $x \in S$ be independent random variables, each with the geometric distribution with parameter $p = ½$.
  - Let r be $\max_{x \in S}(l(x)) + 1$
  - Place x in each of the levels $L_1,\ldots,L_{l(x)}$.
  - Like random Treaps, a random level is chosen for every element of S upon its insertion and remains fixed until the element is deleted.

Random Skip Lists

Lemma 8.9:

The number of levels r in a random leveling of a set S of size n has expected value **E**[r]=O(logn). Moreover, r = O(logn) with high probability.

Proof:

- $r = \max_{x \in S}(l(x)) + 1$
- Levels l(x) are i.i.d. random variables distributed geometrically with parameter ½.

Random Skip Lists

Proof (cont.):

- We may thus view the levels of the members of S as independent geometrically distributed random variables $X_1,\ldots,X_n$:

$$\Pr[X_i > t] \le (1-p)^t \Rightarrow$$

$$\Pr[\max_i X_i > t] \le n(1-p)^t \overset{p=1/2}{=} \frac{n}{2^t}$$

- Using t=alogn and $r=\max_i X_i$, we obtain the desired result for any a>1:

$$\Pr[r > a \log n] \le \frac{1}{n^{a-1}}$$

□

Random Skip Lists

- Lemma 8.9 implies that the tree representing the skip list has height O(logn) with high probability.
- Unfortunately, since the tree is not binary, it does not immediately follow that the search time is similarly bounded.
- Cost of Find(y,S) operation is proportional to
  - number of levels.
  - The number of intervals (nodes) visited at each level
- For a level j, the number of nodes visited at this level does not exceed the number of children of the interval $I_{j+1}(y)$.
- The cost from the total number of children of the nodes on the search path is bounded by

$$O\left(\sum_{j=1}^{r}\left(1+c\left(I_j(y)\right)\right)\right)$$

Random Skip Lists

Lemma 8.10:

Let y be any element and consider the search path $I_r(y),\ldots,I_1(y)$ followed by FIND(y,S) in a random Skip List for the set of size n.

Then,

$$E\left(\sum_{j=1}^{r}\left(1+c\left(I_j(y)\right)\right)\right)=O(\log\ n)$$

Random Skip Lists

Proof :

- We will show that for any specific interval I in a random Skip List, **E**[c(I)] = O(1).

  Since Lemma 8.9 guarantees that r = O(logn) with high probability, this will yield the desired result.
- Let J be any interval at level i of the skip list. We will prove that the expected number of siblings of J is bounded by a constant ➔ the expected number of children is bounded by a constant.
  - It suffice to prove that the number of siblings of J to its right is bounded by a constant.

Random Skip Lists

Proof (cont.):

- Let the intervals to the right of J be
  - $J_1=[x_1,x_2]$, $J_2=[x_2,x_3]$,…, $J_k=[x_k,+\infty]$
- These intervals exist at level i iff each of the elements $x_1,x_2,\ldots,x_k$ belong to L_i.
- If J has s siblings to its right then:
  - $x_1,x_2,\ldots,x_s \notin L_{i+1}$
  - $x_{s+1} \in L_{i+1}$
- Since each element of L_i is independently chosen to be in L_{i+1} with probability ½, the number of right siblings of J is stochastically dominated by a random variable that is geometrically distributed with parameter ½.
- It follows that the expected number of right siblings of J is at most 2.

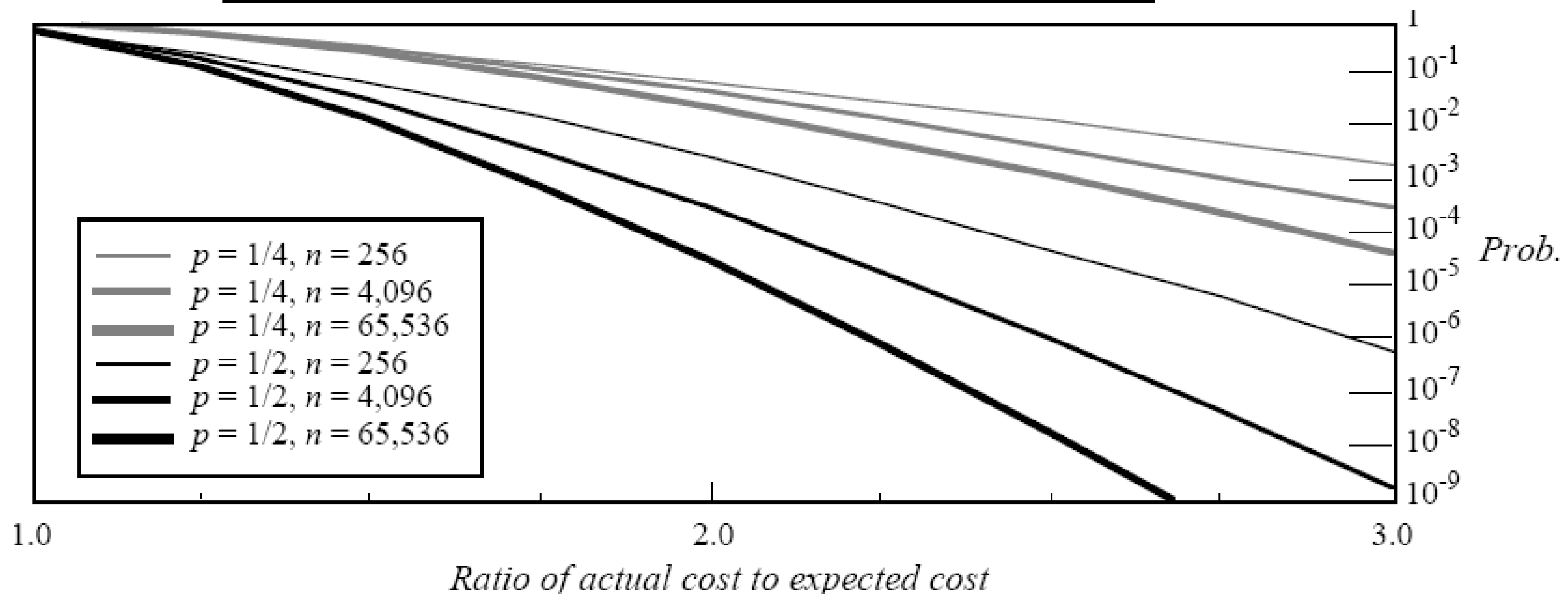
□

Theorem 8.11:

In a random Skip List for a set S of size n, the operations FIND,INSERT and DELETE can be performed in expected time O(logn).

Random Skip Lists

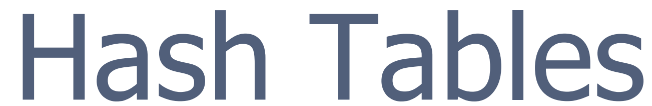
p	Normalized search times (*i.e.*, normalized $L(n)/p$)	Avg. # of pointers per node (*i.e.*, $1/(1-p)$)
$^1/_2$	1	2
$^1/_e$	0.94...	1.58...
$^1/_4$	1	1.33...
$^1/_8$	1.33...	1.14...
$^1/_{16}$	2	1.07...



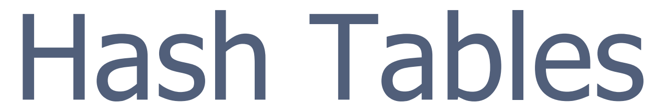
Random Skip Lists

Implementation	Search Time	Insertion Time	Deletion Time
Skip lists	0.051 msec (1.0)	0.065 msec (1.0)	0.059 msec (1.0)
non-recursive AVL trees	0.046 msec (0.91)	0.10 msec (1.55)	0.085 msec (1.46)
recursive 2–3 trees	0.054 msec (1.05)	0.21 msec (3.2)	0.21 msec (3.65)
Self–adjusting trees:			
top-down splaying	0.15 msec (3.0)	0.16 msec (2.5)	0.18 msec (3.1)
bottom-up splaying	0.49 msec (9.6)	0.51 msec (7.8)	0.53 msec (9.0)

Table 2 - Timings of implementations of different algorithms

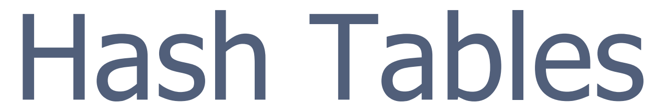

Hash Tables

- Lots of motivations for hashing.
- One is the Dictionary problem:
  - *static:*
    - set S of keys and associated with each key objects.
    - Want to store S in order to be able to do efficient lookups.
  - *dynamic:*
    - sequence of insert(key,object), find(key), delete(key) requests.



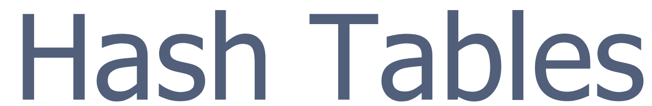
Hash Tables

- Data Structuring problem
  - All data structures discussed earlier require Ω(logn) time to process any search or update operation.
- These time bounds are *optimal*
  - for data structures based on pointers and search trees we are faced with a logarithmic lower bound
  - These time bounds are based on the fact that the only computation we can perform over the keys is to compare them and thereby determine their relationship in the underlying total order.

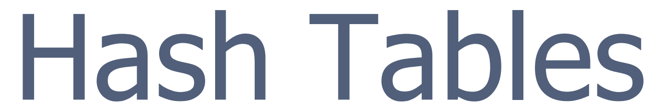


*Can we achieve **O(1)** search time?*

- Suppose:
  - The keys in S are chosen from a totally ordered universe M of size m
  - Without loss of generality, M = {0,1,…,m-1}
  - Keys are distinct
- The idea:
  - Create an array T[0..m-1] of size m in which
    - T[k] = 1 if $k \in S$
    - T[k] = NULL otherwise
  - This is called a *direct-address table*
    - Operations take O(1) time!
    - *So what's the problem?*

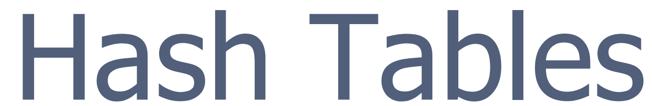


- Direct addressing works well when the range *m* of keys is relatively small.
- But what if the keys are 32-bit integers?
  - Problem 1: direct-address table will have 2^{32} entries, more than 4 billion.
  - Problem 2: even if memory is not an issue, the time to initialize the elements to NULL may be.
- We want to reduce the size of the table to value close to |S|, while maintaining the property that a search or update can be performed in O(1) time.



Hash Tables

- A ***hash table*** consists of:
  - A table T consisting of ***n cells*** indexed by N={0,1,...,n-1}
  - A ***hash function h()***, which is a mapping from M into N.
- $n < m$, otherwise use direct address table.
- The hash function is a fingerprint function for keys in M and specifies a location in the table for each element of M.
- Ideally: we want the hash function to map distinct keys in S to distinct locations in the table.
- A collision occurs when: two distinct keys x and y map in the same location, i.e. h(x) = h(y).
- ***Goal:*** maintain a small table, and use hash function h to map keys into this table. If h behaves randomly, shouldn't get too many collisions.

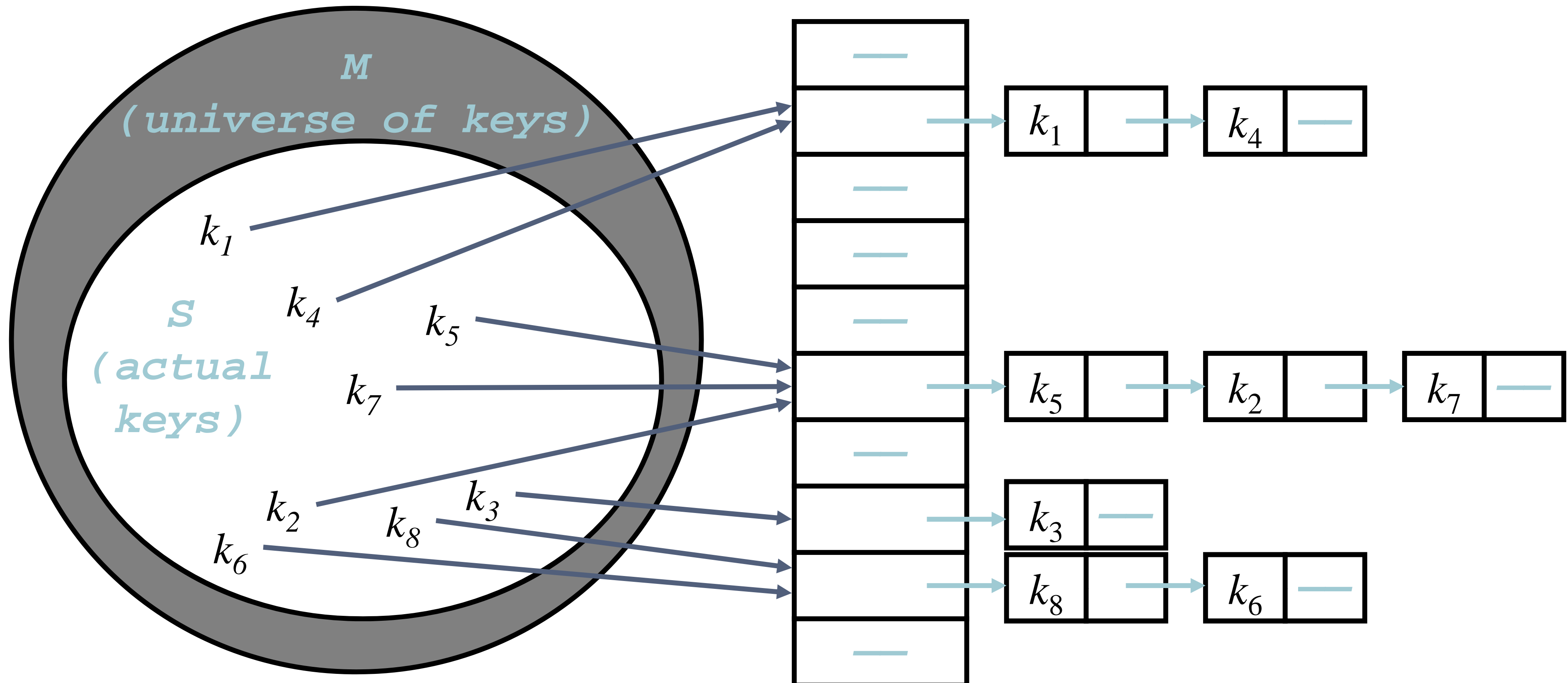


Hash Tables

- *How can we solve the problem of collisions?*
- 30 years of research…
- *Open addressing*
  - To insert: if slot is full, try another slot, and another, until an open slot is found (*probing*)
  - To search, follow same sequence of probes as would be used when inserting the element
- *Chaining*
  - Keep linked list of elements in slots
  - Upon collision, just add new element to list

Hash Tables
Chaining

- Chaining puts elements that collide in a linked list:



Hash Tables
Universal Hash Families

- *Idea*: choose a *family* of hash functions $H=\{h: M \to N\}$, where h is easily represented and evaluated.
  - Any one function $h \in H$ may not be perfect for very many choices of the set S

    BUT
  - We can ensure that for every set S of small cardinality, a large fraction of the hash functions in H are *near-perfect* for S (i.e. the number of collisions is small)
- Thus, for *any* particular set S, a random choice of $h \in H$ will give the desired performance.

Definition 8.5:

- Let M = {0,1,…,m-1} and N = {0,1,…,n-1}, with $m \geq n$
- A family H of functions from M into N is said to be **2-universal** if
  - for all $x,y \in M$ such that $x \neq y$, and
  - For h chosen uniformly at random from H

$$\Pr[\, h(x) = h(y)] \leq \frac{1}{n}$$

➢ A totally random mapping from M to N has a collision probability of exactly 1/n ➔ a random choice from a 2-universal family of hash functions gives a seemingly random function.

Hash Tables
Universal Hash Families

- The collection of all possible mappings from M to N is a 2-universal family.
- Why not use it?
  - Requires Ω(mlogn) random bits to pick one
  - Requires Ω(mlogn) bits to store the chosen function
- *Goal*: smaller 2-universal families that require a *small amount of space* and are *easy to evaluate*.
- This is possible because a randomly chosen $h \in H$ is required to behave like a random function only with respect to *pairs of elements* and not have complete independence like purely random function.

Hash Tables
Universal Hash Families

- For any x,y∈M and h∈H, define the following indicator function for a collision between x,y:

$$\delta(x, y, h) = \begin{cases} 1 & for \;\; h(x) = h(y) \;\; and \;\; x \neq y \\ 0 & otherwise \end{cases}$$

- For all X,Y $\subseteq$ M define the following extensions of δ:

$$\delta(x, y, H) = \sum_{h \in H} \delta(x, y, h)$$

$$\delta(x, Y, h) = \sum_{y \in Y} \delta(x, y, h)$$

$$\delta(X, Y, h) = \sum_{x \in X} \delta(x, Y, h)$$

$$\delta(x, Y, H) = \sum_{y \in Y} \delta(x, y, H)$$

$$\delta(X, Y, H) = \sum_{h \in H} \delta(X, Y, h)$$

Hash Tables
Universal Hash Families

- For a 2-universal family H and any $x \neq y$:
  - $\delta(x,y,H) <= |H|/n$.
- The following theorem shows that our definition of 2-universality is essentially the best possible, since a significantly smaller collision probability cannot be obtained for m>>n.
- *Theorem 8.12* : For any family H of functions from M to N, there exists $x,y \in M$ such that:

$$\delta(x, y, H) > \frac{|H|}{n} - \frac{|H|}{m}$$

Hash Tables
Universal Hash Families

- Why a 2-universal Hash family gives a good solution to the dynamic directory problem?
- ***Lemma 8.13:*** For all x $\in$ M, S $\subseteq$ M and *random* h$\in$H

$$E\left[\delta(x, S, h)\right] \leq \frac{|S|}{n}$$

- The time to perform an INSERT,DELETE or FIND for a key x is
  - *O(1)* to check location h(x) in the Hash Table
  - and if some other key is stored there, the time required to search the linked list maintained for that cell (O(list length))
- If the set of keys currently stored in the Hash Table is S$\subseteq$M, the length of the linked list is δ(x,S,h)

Proof :

$$E\left[\delta(x,S,h)\right] = \sum_{h\in H} \frac{\delta(x,S,h)}{|H|} =$$

$$= \frac{1}{|H|}\sum_{h\in H}\sum_{y\in S}\delta(x,y,h) = \frac{1}{|H|}\sum_{y\in S}\sum_{h\in H}\delta(x,y,h) =$$

$$= \frac{1}{|H|}\sum_{y\in S}\delta(x,y,H) \le \frac{1}{|H|}\sum_{y\in S}\frac{|H|}{n} = \frac{|S|}{n}$$

□

Hash Tables
Universal Hash Families

- Starting with an empty Hash Table.
- Consider a request sequence $R=R_1R_2\ldots R_r$ of update and search operations.
- Suppose that the sequence contains *s* inserts ➔ table will never contain more than s keys.
- ρ(h,R): total cost of processing these requests.
- *Theorem 8.14:* For any sequence R of length r with s INSERTs, and h chosen uniformly at random from a 2-universal family H:

$$E\left[\rho(h,R)\right] \le r\left(1+\frac{|S|}{n}\right)$$

Hash Tables
Universal Hash Families

- Construction of a Universal Hash Family
- Fix m and n, and choose a prime $p \geq m$
- We will work over the field $Z_p = \{0,1,\ldots,p-1\}$
- Let $g: Z_p \to N$ be the function given by
  - $g(x) = x \bmod n$
- For all $a,b \in Z_p$ define the linear function $f_{a,b}:Z_p \to Z_p$ and the hash function $h_{a,b}:Z_p \to N$ as follows:
  - $f_{a,b}(x) = (ax + b) \bmod p$
  - $h_{a,b}(x) = g(f_{a,b}(x))$
- Define the family of hash functions
  - $H = \{h_{a,b} \mid a,b \in Z_p \text{ with } a \neq 0\}$

Hash Tables
Universal Hash Families

- *Lemma 8.15*:

  For all $x,y \in Z_p$ such that $x \neq y$

  $\delta(x,y,H) = \delta(Z_p, Z_p, g)$

- ***Theorem 8.16:***

  The family $H = \{h_{a,b} \mid a,b \in Z_p$ with $a \neq 0\}$ is a 2-universal family

Hash Tables
Universal Hash Families

Proof:

- For each $z \in N$, let $A_z = \{z \in Z_p \mid g(x) = z\}$
- It is clear that: $|A_z| \leq \lceil p/n \rceil$
- In other words, for every $r \in Z_p$ there at most $\lceil p/n \rceil$ different choices of $s \in Z_p$ such that $g(s)=g(r)$.
- Since there are p different choices of $r \in Z_p$:

$$\delta(Z_p, Z_p, g) \leq p\left(\left\lceil \frac{p}{n} \right\rceil - 1\right) \leq \frac{p(p-1)}{n}$$

- Lemma 8.15 implies that for any distinct $x,y \in Z_p$ $\delta(x,y,H) \leq p(p-1)/n$
- Since the size of $|H|=p(p-1)$: $\delta(x, y, H) \leq \frac{|H|}{n}$ □

Hash Tables
for
Static Dictionaries

- The hashing scheme described
  - achieves bounded time for |S|=O(logn)
  - but requires unbounded time in the worst case (|S|>>n)
- Static dictionary problem:
  - A set S of size s is *fixed in advance*
  - We *only* need to support the FIND operation
- FIND() in O(1) *worst case*.

Hash Tables

for

Static Dictionaries

- Use a hash function that is perfect for S.
- Since a hash function cannot be perfect for every possible set S ➔ use a family of perfect hash functions.
- Definition 8.7:

  A family of hash functions $H = \{h : M \rightarrow N\}$ is said to be a *perfect hash family* if for each set $S \subseteq M$ of size $s<n$, there exists a hash function $h \in H$ that is perfect for S.
- To guarantee even the existence of a perfect hash family of size polynomial in m: $s = O(\log m)$

Hash Tables
for
Static Dictionaries

- Relax the property of perfection ➔ *double hashing* (allow a few collisions)
  - Keys that are hashed to a particular location of the primary table are handled by using a *new hash function* to map them into a *secondary* hash table associated with that location.
- The set of keys colliding at a specific location of the primary hash table is called a *bin*.
- Let $S \subset M$ and $h : M \rightarrow N$. For each table location $i \in \{0,\ldots,n-1\}$, define the bin

  $$B_i(h,S) = \{x \in S \mid h(x) = i\}$$

  The size of a bin is $b_i(h,S) = |B_i(h,S)|$

Hash Tables
for
Static Dictionaries

- A perfect hash function ensures that all bins are of size at most 1.

- Definition 8.9:

  A hash function h is *b-perfect* for S if $b_i(h,S) <= b$, for each i. A family of hash functions $H = \{h : M \rightarrow N\}$ is said to be a *b-perfect family* if for each $S \subseteq M$, of size s, there exists a hash function $h \in H$ that is b-perfect for S.

Hash Tables

for

Static Dictionaries

Scheme for double hashing:

- Use a (logm)-perfect hash function h to map S into the primary table T.
  - The description of h can be stored in one auxiliary cell
- Consider the bin B_i consisting of all keys from S mapped into a particular cell T[i].
  - In this cell we store the description of a secondary hash function h_i, which is used to map the elements of the bin B_i into a secondary table T_i associated with that location.
  - Since the size of B_i is is bounded by b, we know that we can find a hash function h_i that is perfect for B_i provided 2b is polynomially bounded in m.
  - Since b = logm, this condition holds and and so the double hashing scheme can be implemented with:
    - **O(1)** query time for any $m \geq n$