Search Trees

Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

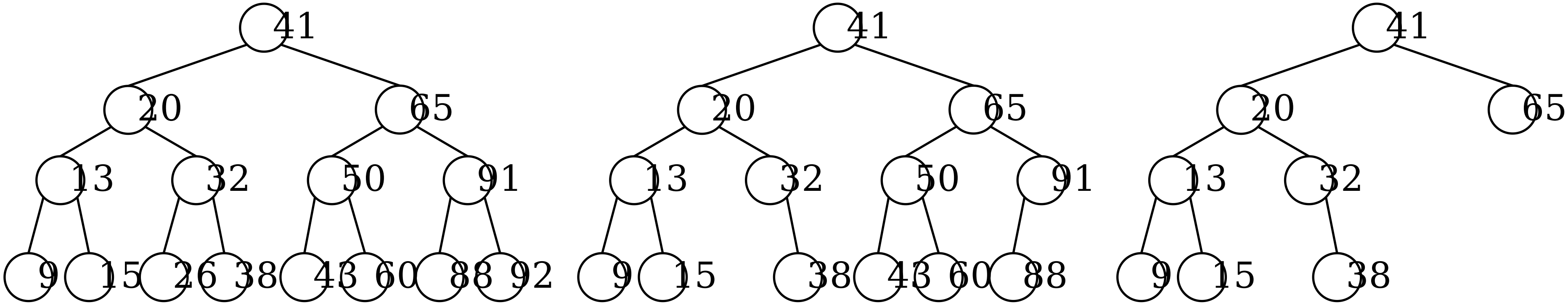
Efficient implementation of **OSet**

	Search(S,k)	**Insert(S,x)**	**Remove(S,x)**
BinarySearchTree	$O(h)$ worst-case	$O(h)$ worst-case	$O(h)$ worst-case
Treap	$O(\log n)$ in expectation	$O(\log n)$ in expectation	$O(\log n)$ in expectation

A data structure supporting OSet operations in $O(\log n)$ time, even in worst-case?

“Balanced” BST

- What does it mean to be “balanced”?
  - Perfectly Balanced. (For each node, two subtrees have same height.)
  - Almost Perfectly Balanced.
  - Not Perfectly Balanced.
- An n-node BST is “balanced” if it has height $O(\log n)$.



Perfectly Balanced

Almost Perfectly Balanced

Not Perfectly Balanced

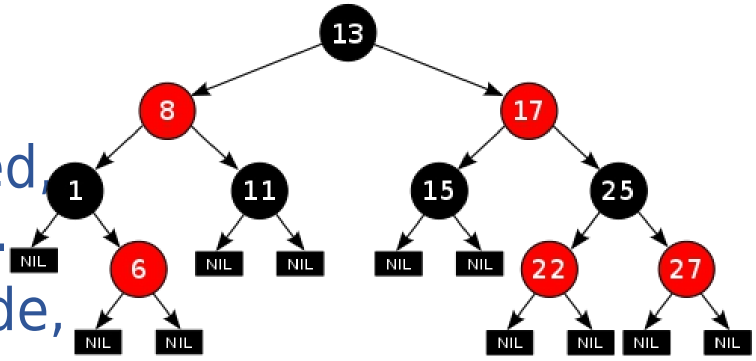
“Balanced” ~~Binary~~ Search Trees

- AVL tree (Adelson-Velsii & Landis, 1962)
- B-tree (Bayer & McCreight, 1970)
- Red-black tree (Bayer, 1972)
- Splay tree (Sleator & Tarjan, 1985)
- Treap (Seidel & Aragon, 1996)
- Skip list (Pugh, 1989)
- and so on …

Red-black Tree (RB-Tree)

A **Red-black Tree** (**RB-Tree**) is a BST in which each node has a *color*, and satisfies the following properties:

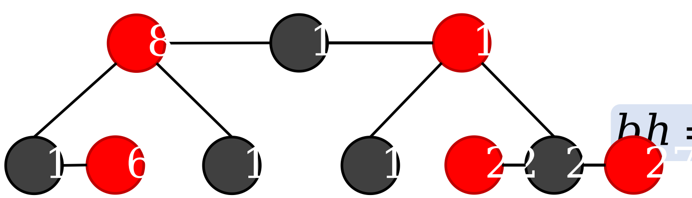
- Every node is either red or black.
- The root is black.
- Every leaf is black.
- **[no-red-edge]** If a node is red, then both its children are black.
- **[black-height]** For every node, all paths from the node to its descendant leaves contain same number of black nodes.

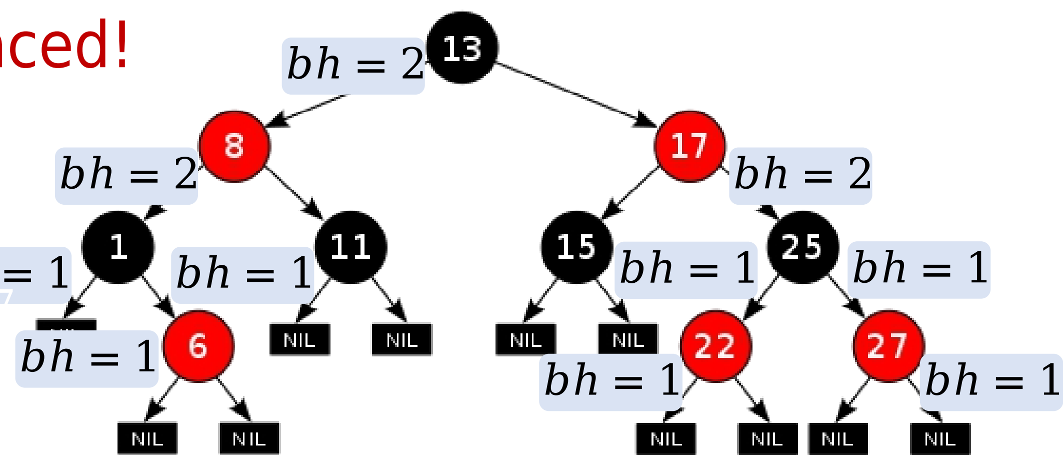


Black Height

- Call the number of black nodes on any simple path from, but not including, a node x down to a leaf the **black-height** of the node, denoted by $\boldsymbol{bh(x)}$.
- Due to black-height property, from the black-height perspective, RB-Trees are "perfectly balanced".
- Due to no-red-edge property, actual height of a RB-Tree does not deviate a lot from its black-height.

RB-Trees are well balanced!





Height of RB-Trees

- **Claim:** In a RB-Tree, the subtree rooted at x contains at least $2^{bh(x)} - 1$ internal nodes.
- **Proof** (via induction on height of x):
- **[Basis]** If x is a leaf, $bh(x) = 0$ and the claim holds.
- **[Hypothesis]** The claim holds for all nodes with height at most $h - 1$.
- **[Inductive Step]** Consider a node x with height $h \geq 1$. It must have two children. (Why?) So the number of internal nodes rooted at x is:
  $\geq 1 + (2^{bh(x.left)} - 1) + (2^{bh(x.right)} - 1)$
  $\geq 1 + (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1)$
  $= 2^{bh(x)} - 1$

Height of RB-Trees

- **Claim:** In a RB-Tree, the subtree rooted at x contains at least $2^{bh(x)} - 1$ internal nodes.
- Due to no-red-edge: $h = height(root) \leq 2 \cdot bh(root)$.
- $n \geq 2^{bh(root)} - 1 \geq 2^{h/2} - 1$, implying $h \leq 2 \cdot \lg(n+1)$.
- **Theorem:** The height of an n-node RB-Tree is $O(\log n)$.

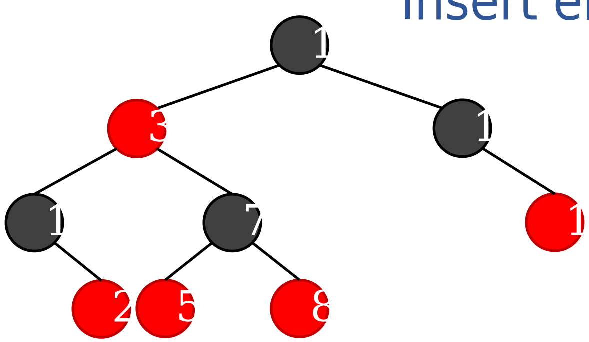
- RB-Trees support Search, Min, Max, Predecessor, Successor operations in worst-case $O(\log n)$ time!
- What about Insert and Remove?!

Insert node z into an RB-Tree

- **Step 1:** Color z as red and insert as if the RB-tree were a BST.
- **Step 2:** Fix any violated properties.
  - No fix is needed if z has a black parent after insertion.

Maintain black-height,
fix no-red-edge if necessary.

Example:
Insert element with key 2.



RB-Tree Properties:
- Each node is red or black.
- Root is black.
- Leaves are black.
- No-red-edge property.
- Black-height property.

Insert node z into an RB-Tree

- **Step 1:** Color z as red and insert as if the RB-tree were a BST.
- **Step 2:** Fix any violated properties.
  - **Case 0:** z becomes the root of the RB-Tree.
  - **Fix:** simply recolor z to be black.

RB-Tree Properties:

- Each node is red or black.
- Root is black (**easy fix**).
- Leaves are black.
- No-red-edge property (**fix**).
- Black-height property (**mainta**

Insert node z into an RB-Tree

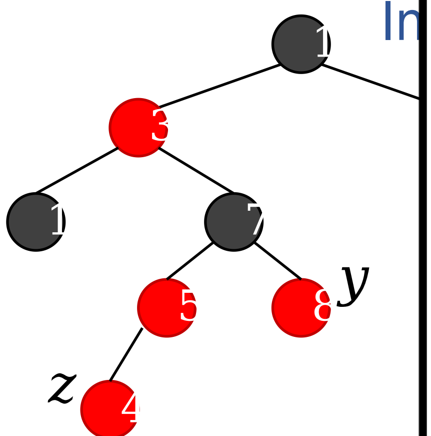
- **Step 1:** Color z as red and insert as if the RB-tree were a BST.
- **Step 2:** Fix any violated properties.
  - **Case 1:** z's … as red uncle y. “…

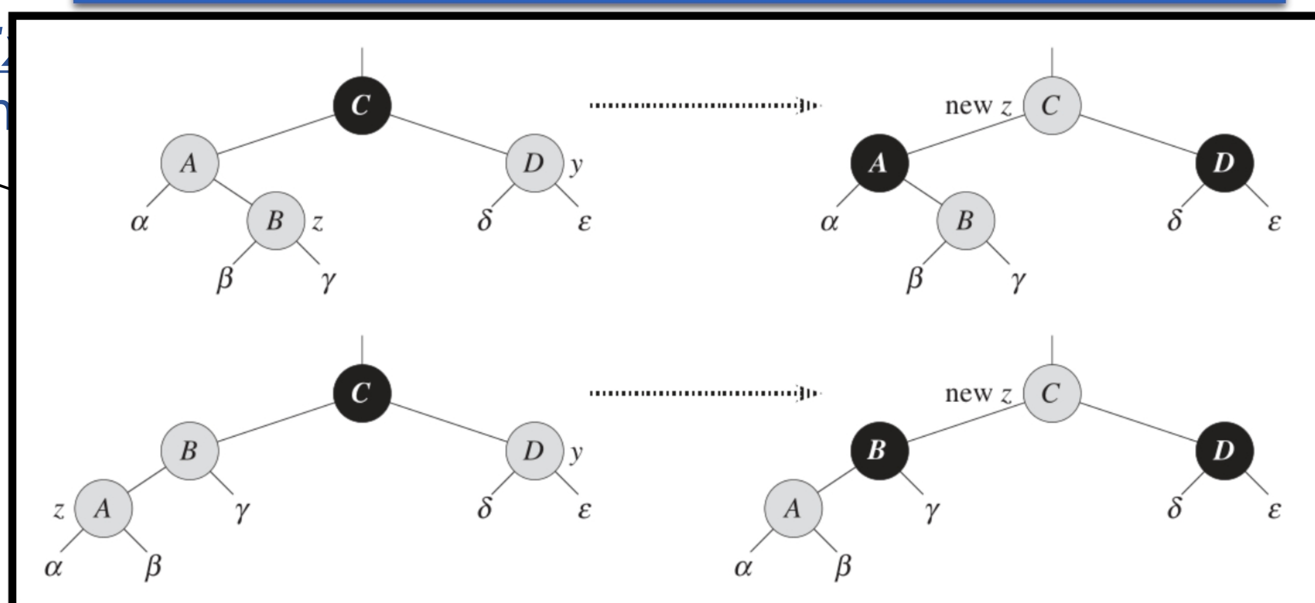
Black … at C … exist

Black-height property maintained, and we “push-up” violation of no-red-edge property.

Ex…

In…

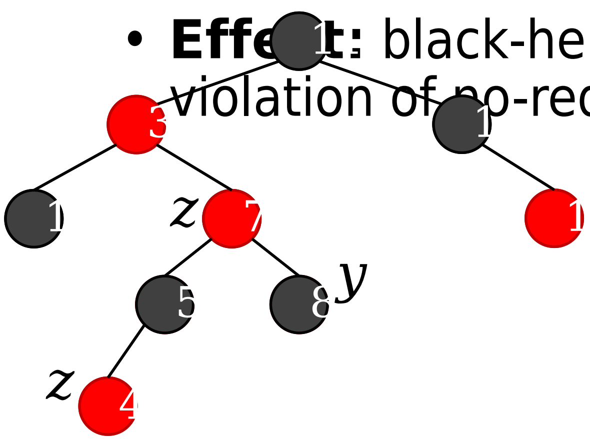




…ta…

Insert node z into an RB-Tree

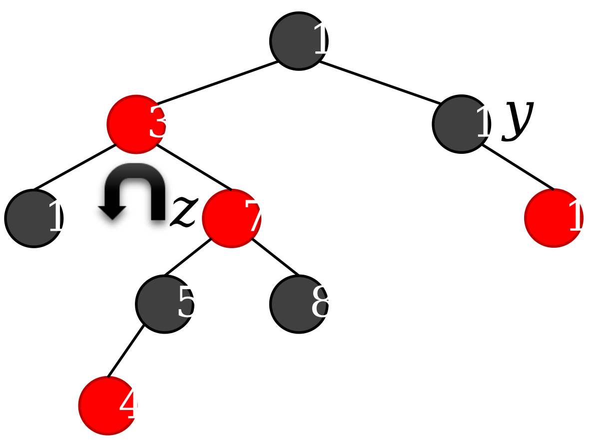
- **Step 1:** Color z as red and insert as if the RB-tree were a BST.
- **Step 2:** Fix any violated properties.
  - **Case 1:** z's parent is red (so z has black grandparent), and has red uncle y.
  - **Fix:** recolor z's parent and uncle to black, recolor z's grandparent to red.
  - **Effect:** black-height property maintained, and we "push-up" violation of no-red-edge property.

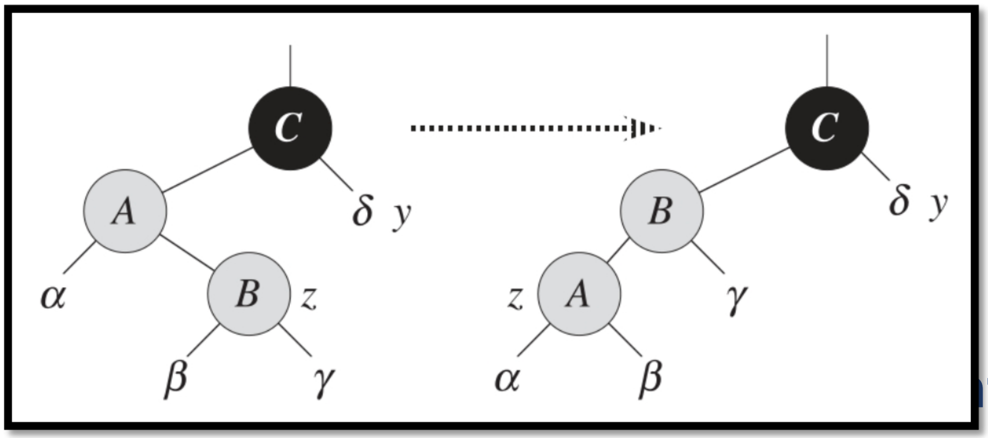


RB-Tree Properties:
- Each node is red or black.
- Root is black (**easy fix**).
- Leaves are black.
- No-red-edge property (**fix**).
- Black-height property (**mainta**

Insert node z into an RB-Tree

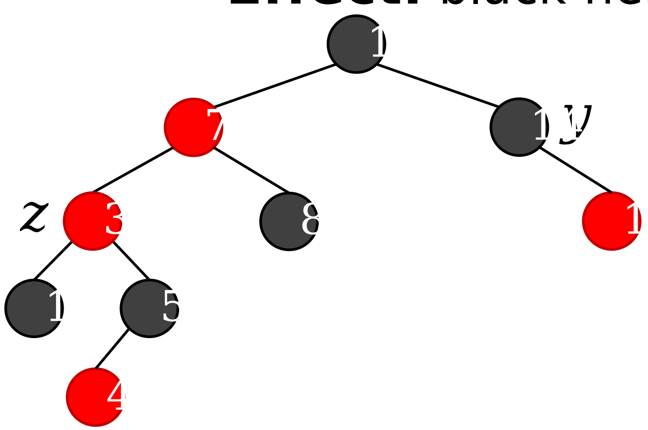
- **Step 1:** Color z as red and insert as if the RB-tree were a BST.
- **Step 2:** Fix any violated properties.
  - **Case 2:** z's parent is red, has black uncle y. z is right child of its parent.
  - **Fix:** "left-rotate" at z's parent.

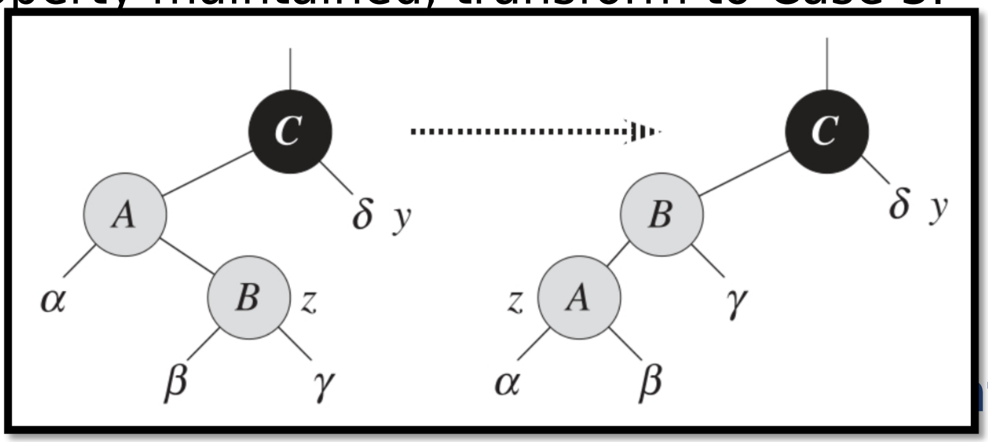




Insert node z into an RB-Tree

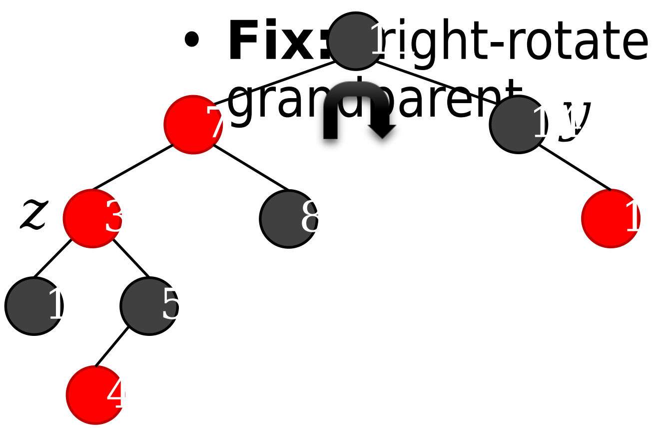
- **Step 1:** Color z as red and insert as if the RB-tree were a BST.
- **Step 2:** Fix any violated properties.
  - **Case 2:** z's parent is red, has black uncle y. z is right child of its parent.
  - **Fix:** "left-rotate" at z's parent.
  - **Effect:** black-height property maintained, transform to Case 3.

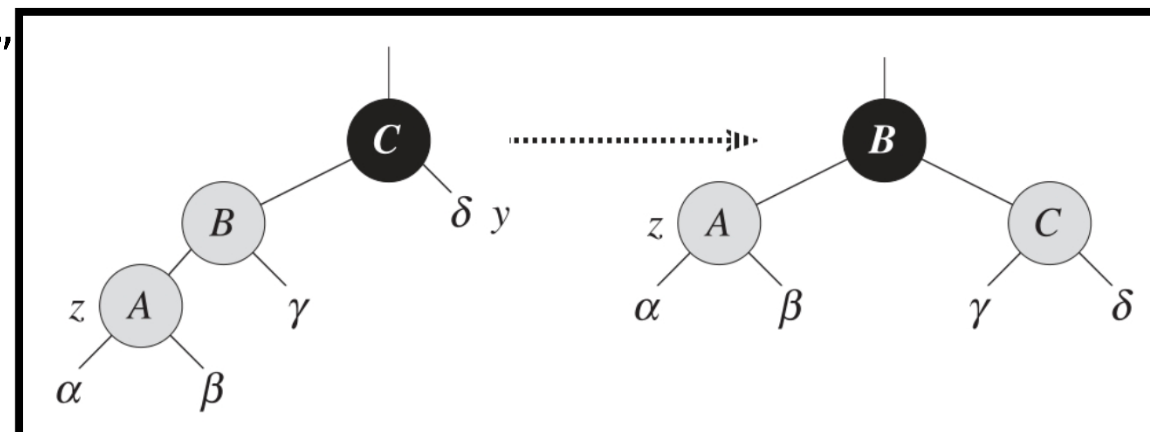




Insert node z into an RB-Tree

- **Step 1:** Color z as red and insert as if the RB-tree were a BST.
- **Step 2:** Fix any violated properties.
  - **Case 2:** z's parent is red, has black uncle y. z is right child of its parent.
  - **Case 3:** z's parent is red, has black uncle y. z is left child of its parent.
  - **Fix:** “right-rotate” grandparent

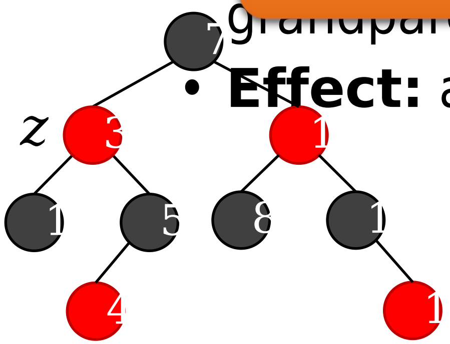


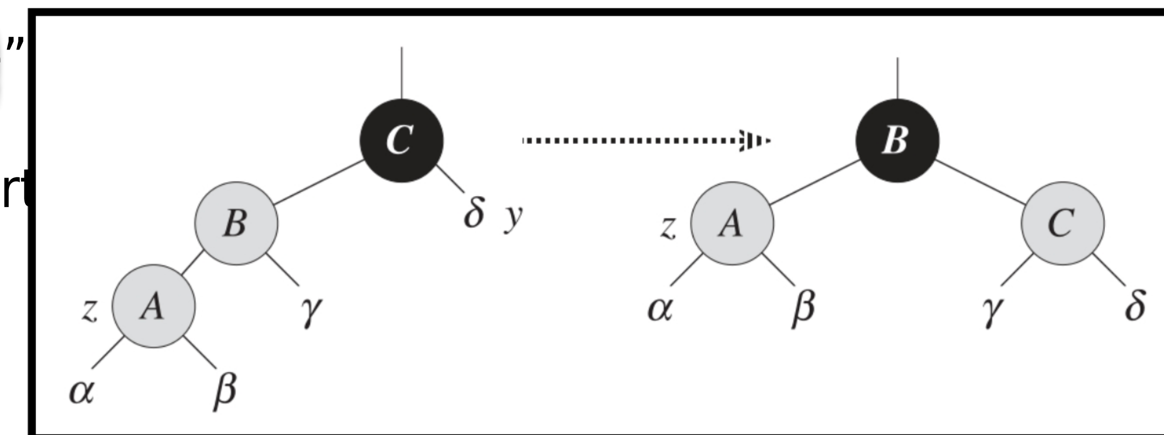


Insert node z into an RB-Tree

- **Step 1:** Color z as red and insert as if the RB-tree were a BST.
- **Step 2:** Fix any violated properties.
  - **Case 2:** z's parent is red, has black uncle y. z is right child of its parent.
  - **Case 3:** z's parent is red, has black uncle y. z is left child of its parent.
  - ” grandparent.
  - **Effect:** all propert

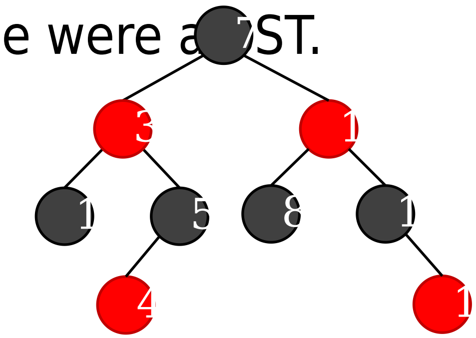
We are done!

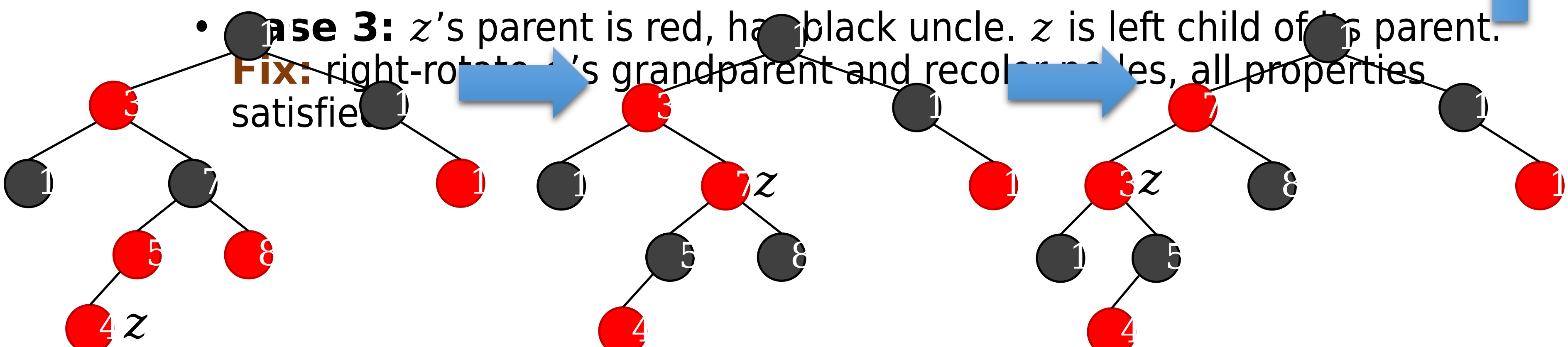




Insert node z into an RB-Tree

- **Step 1:** Color z as red and insert as if the RB-tree were a BST.
- **Step 2:** Fix any violated properties.
  - **No-Fix-Needed:** z has a black parent.
  - **Case 0:** z becomes the root.
    Fix: recolor z to be black.
  - **Case 1:** z's parent is red, has red uncle.
    Fix: recoloring to push-up "no-red-edge" violation (maintain "black-height").
  - **Case 2:** z's parent is red, has black uncle. z is right child of its parent.
    Fix: left-rotate z's parent to transform to Case 3 (maintain "black-height").
  - **Case 3:** z's parent is red, has black uncle. z is left child of its parent.
    Fix: right-rotate z's grandparent and recolor nodes, all properties satisfied.





Insert node z into an RB-Tree

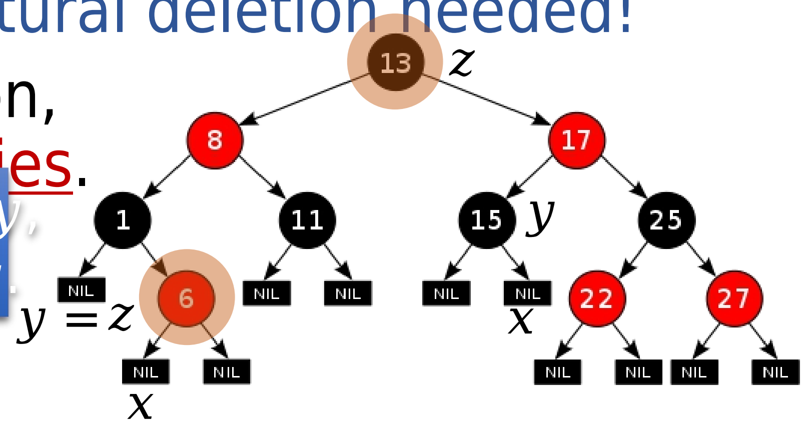
- **Step 1:** Color z as red and insert as if the RB-tree were a BST.
- **Step 2:** Fix any violated properties.
  - **No-Fix-Needed:** z has a black parent.
  - **Case 0:** z becomes the root.
    Fix: recolor z to be black.
  - **Case 1:** z's parent is red, has red uncle.
    Fix: recoloring to push-up "no-red-edge" violation (maintain "black-height").
  - **Case 2:** z's parent is red, has black uncle. z is right child of its parent.
    Fix: left-rotate z's parent to transform to Case 3 (maintain "black-height").
  - **Case 3:** z's parent is red, has black uncle. z is left child of its parent.
    Fix: right-rotate z's grandparent and recolor nodes, all properties satisfied.

- Time Complexity of **Insert** operation?
  - $O(h) = O(\log n)$ time. (Case 1 appears at most $O(h)$ times.)
  - $O(1)$ rotations. (**Insert** has limited impact on tree shape.)

Remove node z from an RB-Tree

- If z's right child is an external node, then z is the node to be deleted *structurally*: subtree rooted at $z.left$ will replace z.
- If z's right child is an internal node, then let y be the min node in subtree rooted at $z.right$. Overwrite z's info with y's info, and y is the node to be deleted *structurally*: subtree rooted at $y.right$ will replace y.
- Either way, only **one** structural deletion needed!
- Apply the structural deletion, and repair violated properties.

Call the node to be deleted structurally y, and let x be the node that will replace y.

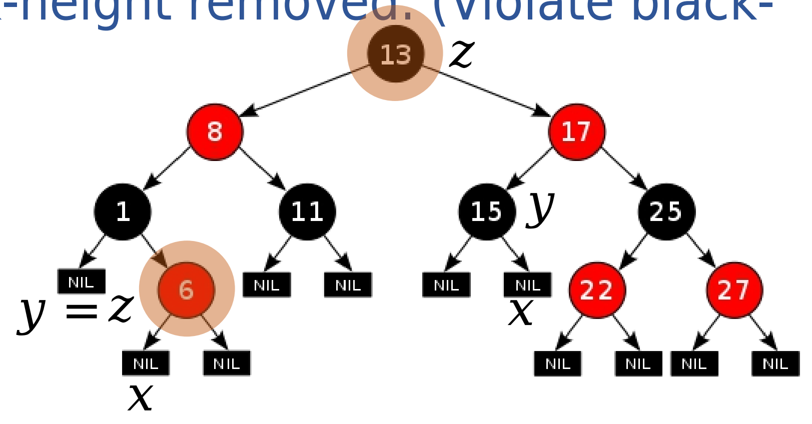


Remove node z from an RB-Tree

- **Step 1:** Identify the structural deletion.
- **Step 2:** Apply the structural deletion. (Maintain BST property.)
- **Step 3:** Repair violated RB-tree properties. (Maintain BST property.)
  - If y is a **red** node: no violations.
  - If y is a **black** node and x is a **red** node: recolor x to black and done.
  - If y is a **black** node and x is a **black** node:
    - y's contribution to black-height removed. (Violate black-height.)

RB-Tree Properties:

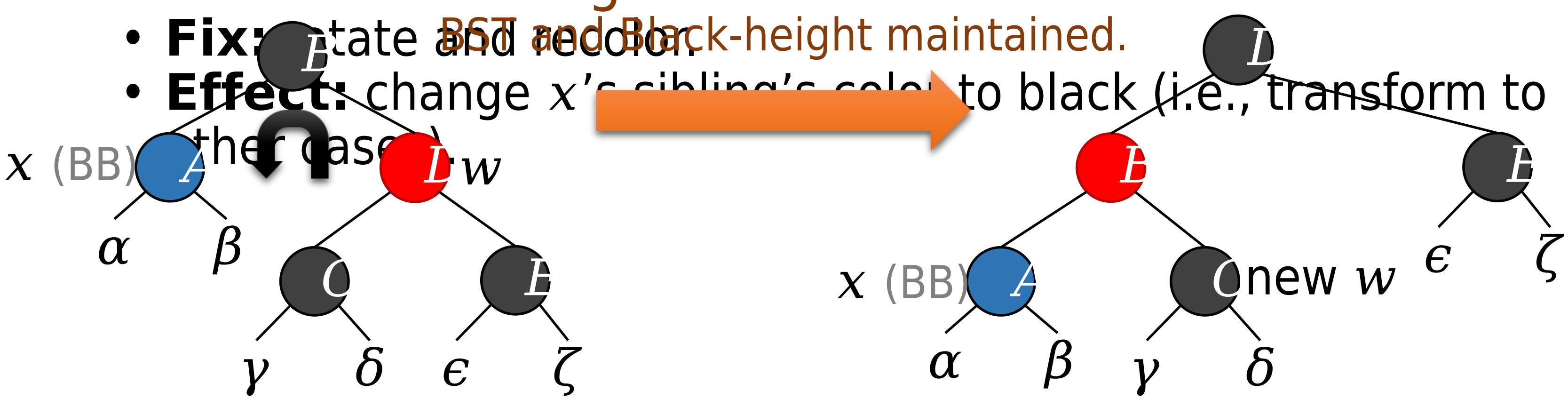
- Each node is red or black.
- Root is black.
- Leaves are black.
- No-red-edge property.
- Black-height property.



Remove node z from an RB-Tree

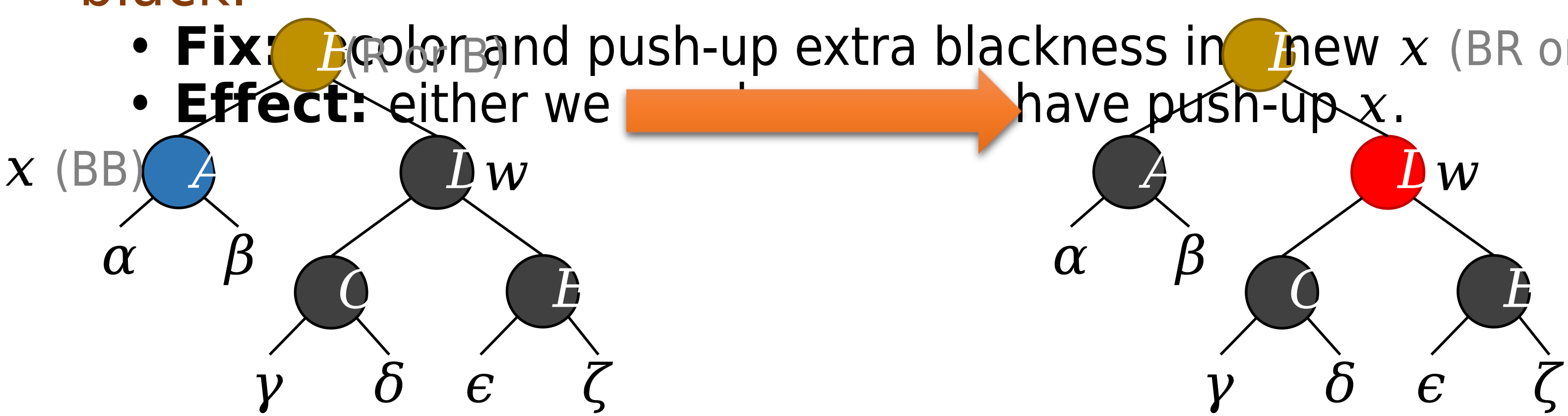
- **Step 1&2:** Find & apply structural deletion. (Maintain BST property.)
  - Let y be the structurally removed node, and x takes its place.
- **Step 3:** Repair violated RB-tree properties. (Maintain BST property.)
  - Assume *black* x is left child of its parent *after* taking *black* y's place.
  - Focus on fixing black-height property.
- **Case 1:** x's sibling w is red.
  - **Fix:** Rotate and recolor.
  - **Effect:** change x's sibling's color to black (i.e., transform to other cases).

BST and Black-height maintained.



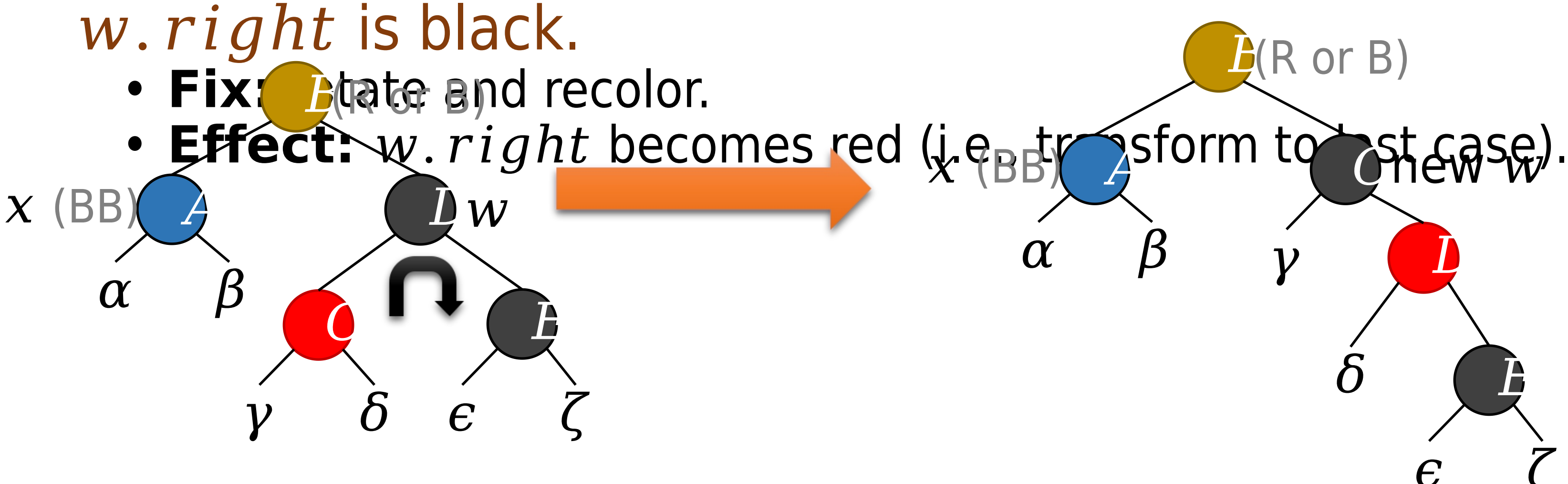
Remove node z from an RB-Tree

- **Step 1&2:** Find & apply structural deletion. (Maintain BST property.)
  - Let y be the structurally removed node, and x takes its place.
- **Step 3:** Repair violated RB-tree properties. (Maintain BST property.)
  - Assume x is left child of its parent.
  - Focus on fixing black-height property.
- **Case 2:** x's sibling w is black, and both w's children are black.
  - **Fix:** color and push-up extra blackness in new x (BR or BB)
  - **Effect:** either we have push-up x.



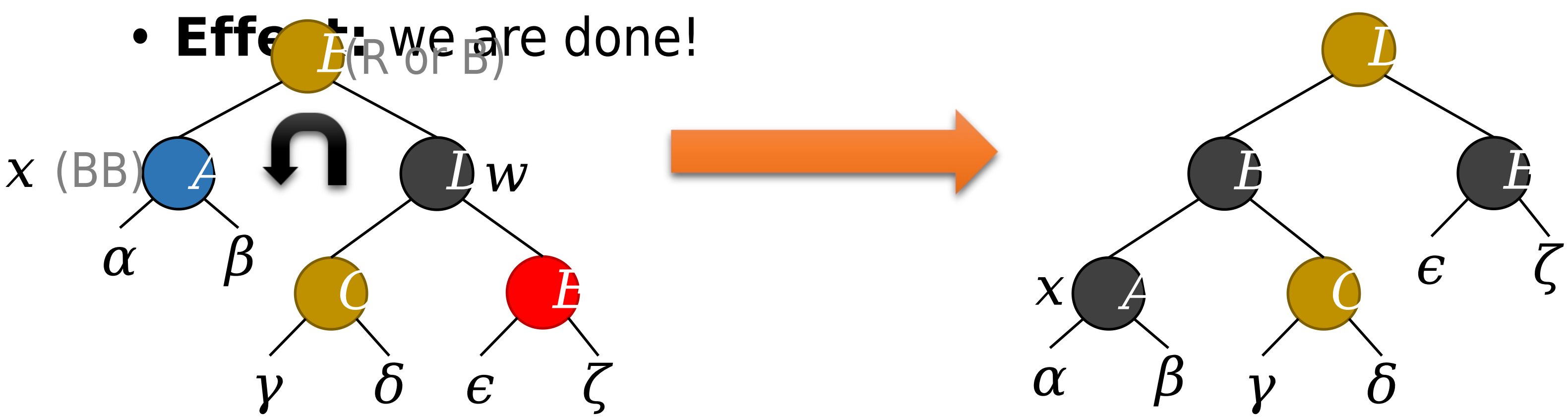
Remove node z from an RB-Tree

- **Step 1&2:** Find & apply structural deletion. (Maintain BST property.)
  - Let y be the structurally removed node, and x takes its place.
- **Step 3:** Repair violated RB-tree properties. (Maintain BST property.)
  - Assume x is left child of its parent.
  - Focus on fixing black-height property.
- **Case 3:** x's sibling w is black, $w.left$ is red and $w.right$ is black.
  - **Fix:** Rotate and recolor.
  - **Effect:** $w.right$ becomes red (i.e., transform to last case).



Remove node z from an RB-Tree

- **Step 1&2:** Find & apply structural deletion. (Maintain BST property.)
  - Let y be the structurally removed node, and x takes its place.
- **Step 3:** Repair violated RB-tree properties. (Maintain BST property.)
  - Assume x is left child of its parent.
  - Focus on fixing black-height property.
- **Case 4:** x's sibling w is black, $w.right$ is red.
  - **Fix:** rotate and recolor.
  - **Effect:** we are done!



Remove node z from an RB-Tree

- **Step 1&2:** Find & apply structural del $O(h) = O(\log n)$ T property.)
  - Let y be the structurally removed node, ar $O(h) = O(\log n)$
- **Step 3:** Repair violated RB-tree properties. (Maintain BST property.)
  - If x is not double-black: then done or easy fix.
  - If x is double-black:
    - **Case 1:** rotate and recolor; transform to other cases.
    - **Case 2:** recolor; done or push-up violations.
    - **Case 3:** rotate and recolor; transform to Case 4.
    - **Case 4:** rotate and recolor; then done.
- Time complexity of **Remove** operation?
  - $O(h) = O(\log n)$ time.
    (For each case, in $O(1)$ time, either done or push-up violation.)
  - $O(1)$ rotations. [**Remove** has limited impact on tree shape.]
    (Entering Case 2 from Case 1 will finish the fixing process.)

Efficient implementation of **OSet**

	Search(S,k)	**Insert(S,x)**	**Remove(S,x)**
BinarySearchTree	$O(h)$ worst-case	$O(h)$ worst-case	$O(h)$ worst-case
Treap	$O(\log n)$ in expectation	$O(\log n)$ in expectation	$O(\log n)$ in expectation
RB-Tree	$O(\log n)$ worst-case	$O(\log n)$ worst-case	$O(\log n)$ worst-case

Efficiency versus Simplicity

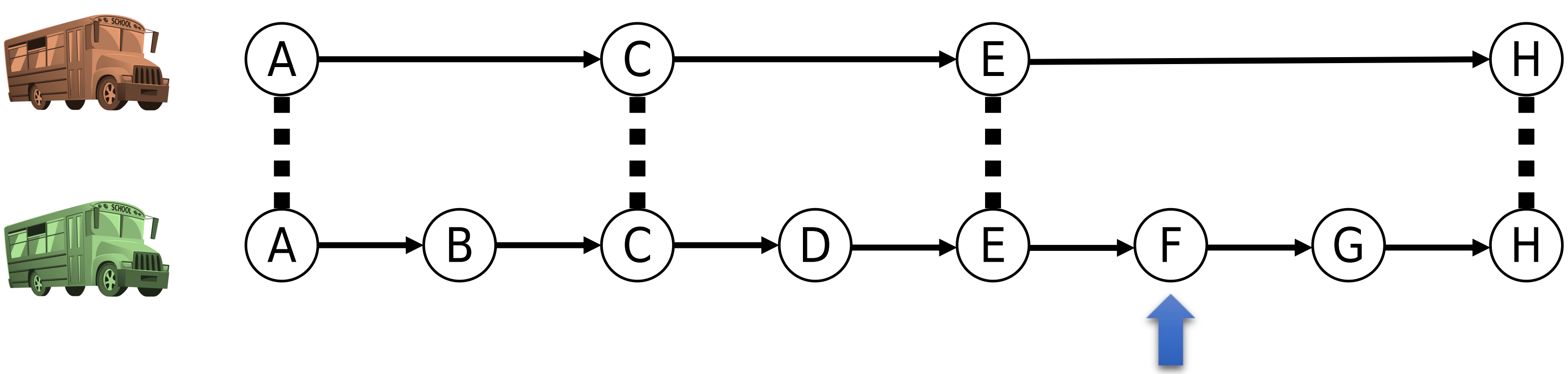
“Balanced” ~~Binary~~ Search Trees

- AVL tree (Adelson-Velsii & Landis, 1962)
- B-tree (Bayer & McCreight, 1970)
- Red-black tree (Bayer, 1972)
- Splay tree (Sleator & Tarjan, 1985)
- Treap (Seidel & Aragon, 1996)
- Skip list (Pugh, 1989)
- and so on …

SkipList

	Search(S,k)	Insert(S,x)	Remove(S,x)
SortedLinkedList	$O(n)$	$O(n)$	$O(1)$

Q: Why sorted linked-list is slow?
A: To reach an element, you have to move from current position to destination **one element at a time**.



SkipList

	Search(S,k)	Insert(S,x)	Remove(S,x)
SortedLinkedList	$O(n)$	$O(n)$	$O(1)$

Q: Why sorted linked-list is slow?
A: To reach an element, you have to move from current position to destination **one element at a time**.

Let's build an "expressway"!

Search cost is reduced by half!

Why stop at one layer of "expressway"?!

Example: search for 8.

SkipList

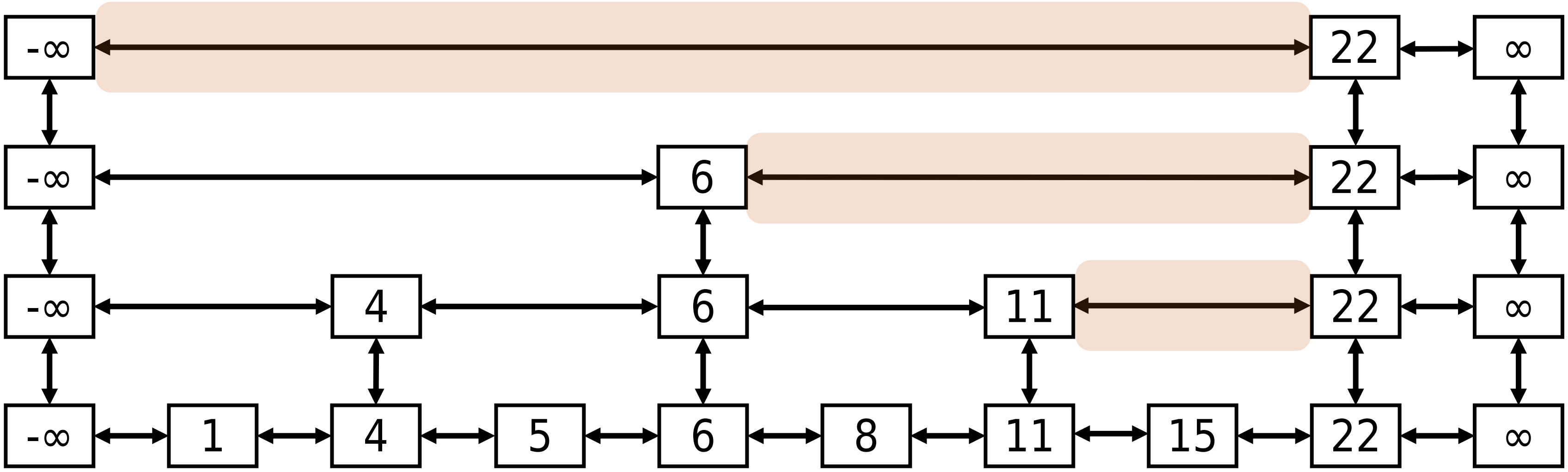
Search can be done in $O(\log n)$ time

Space complexity $\approx 2n$

Build multiple “expressways”:
Reduce number of elements by half at each level.

This is just binary search: reduce search range by half at each level.
This is very efficient: spend $O(1)$ time at each level, and $O(\log n)$ levels

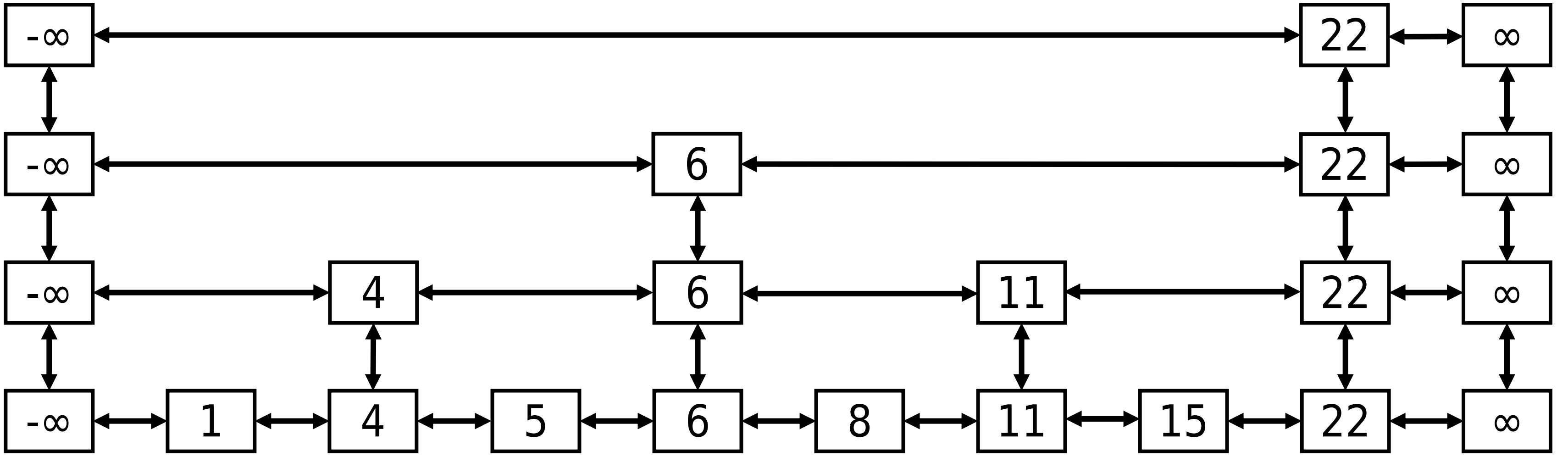
Example One: search for 15.*Example Two:* search for 14.



SkipList

	Search(S,k)	Insert(S,x)	Remove(S,x)
SortedLinkedList	$O(n)$	$O(n)$	$O(1)$
Static-SkipList	$O(\log n)$	???	???

Efficient **Search** with limited space overhead.
But how to implement **Insert** and **Remove**?

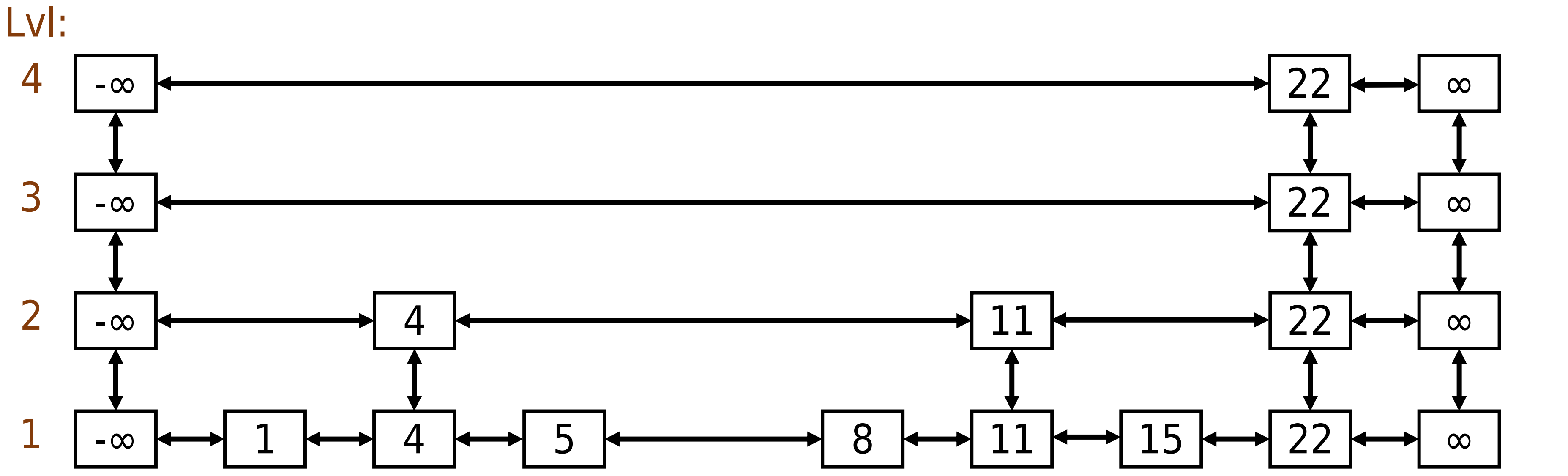


The real **SkipList**

<u>Insert(L,x):</u>

level = 1, done = false
while (!done)
  Insert x into level k list.
  Flip a fair coin:
    With probability 1/2: done = true
    With probability 1/2: k = k+1

<u>Example:</u> insert 7.

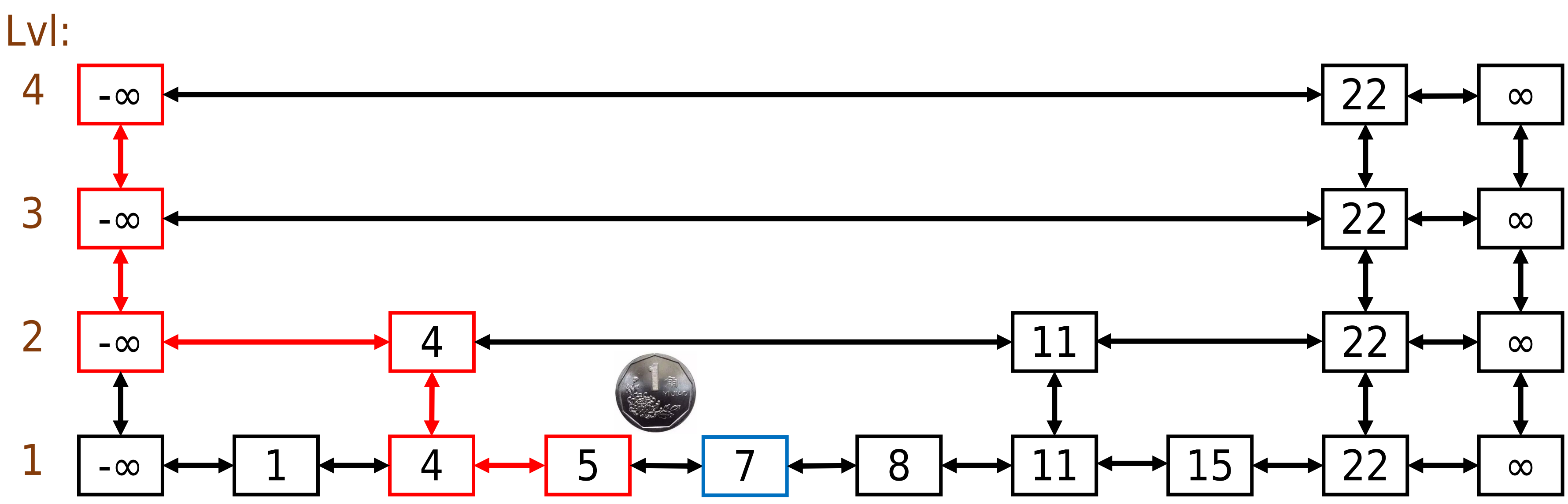


The real **SkipList**

<u>Insert(L,x):</u>

```
level = 1, done = false
while (!done)
  Insert x into level k list.
  Flip a fair coin:
    With probability 1/2: done = true
    With probability 1/2: k = k+1
```

<u>Example:</u> insert 7.

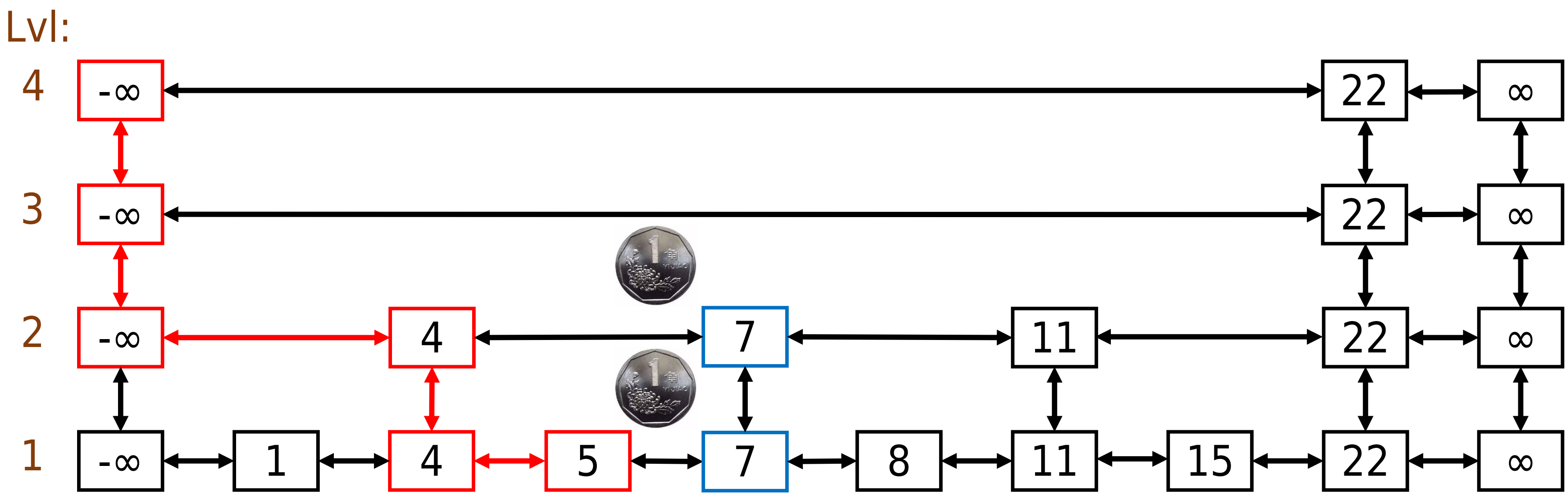


The real **SkipList**

<u>Insert(L,x):</u>

level = 1, done = false
while (!done)
  Insert x into level k list.
  Flip a fair coin:
    With probability 1/2: done = true
    With probability 1/2: k = k+1

<u>Example:</u> insert 7.



The real **SkipList**

Insert(L,x):

level = 1, done = false
while (!done)
  Insert x into level k list.
  Flip a fair coin:
    With probability 1/2: done = true
    With probability 1/2: k = k+1

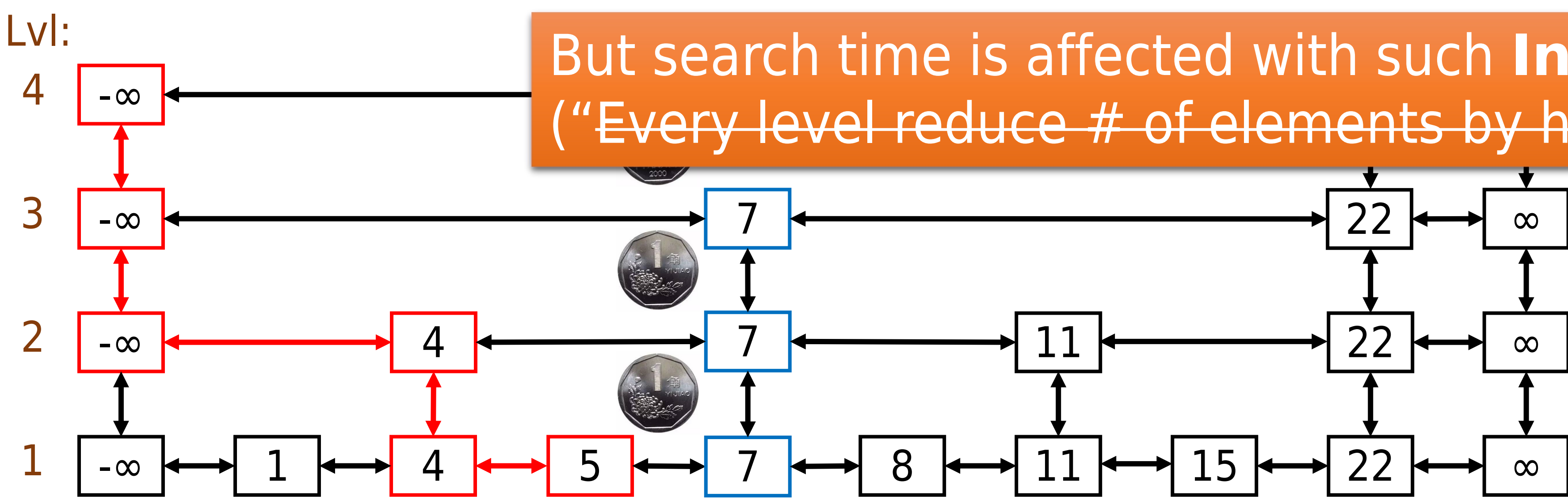
Time complexity of **Insert**:

- $O(1)$ in expectation.
- $O(\log n)$ with high probability.
  (with prob. $\geq 1 - 1/n^{\Theta(1)}$)

Max level of n-element **SkipLis**
is $O(\log n)$ with high probability

Example: insert 7

But search time is affected with such **Inser**
("~~Every level reduce # of elements by half~~"



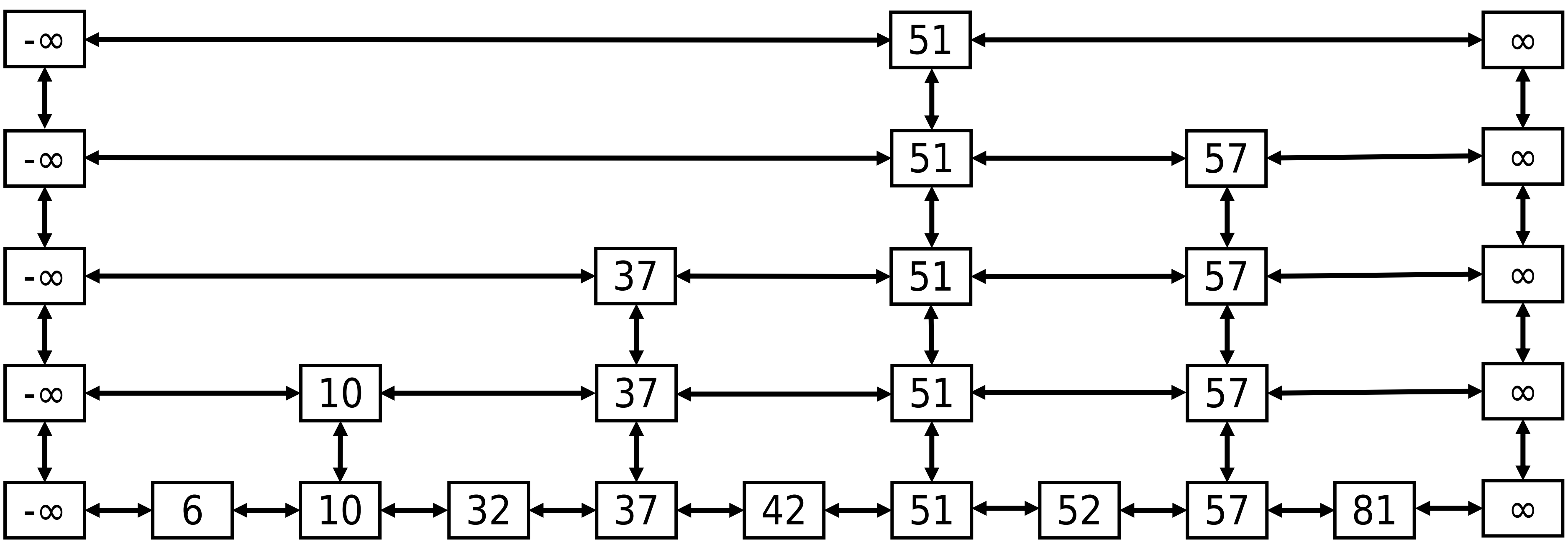
The real **SkipList**

Insert(L,x):

```
level = 1, done = false
while (!done)
  Insert x into level k list.
  Flip a fair coin:
    With probability 1/2: done = true
    With probability 1/2: k = k+1
```

Max level of n-element **SkipLis**
is $O(\log n)$ with high probability

Example: search 81.

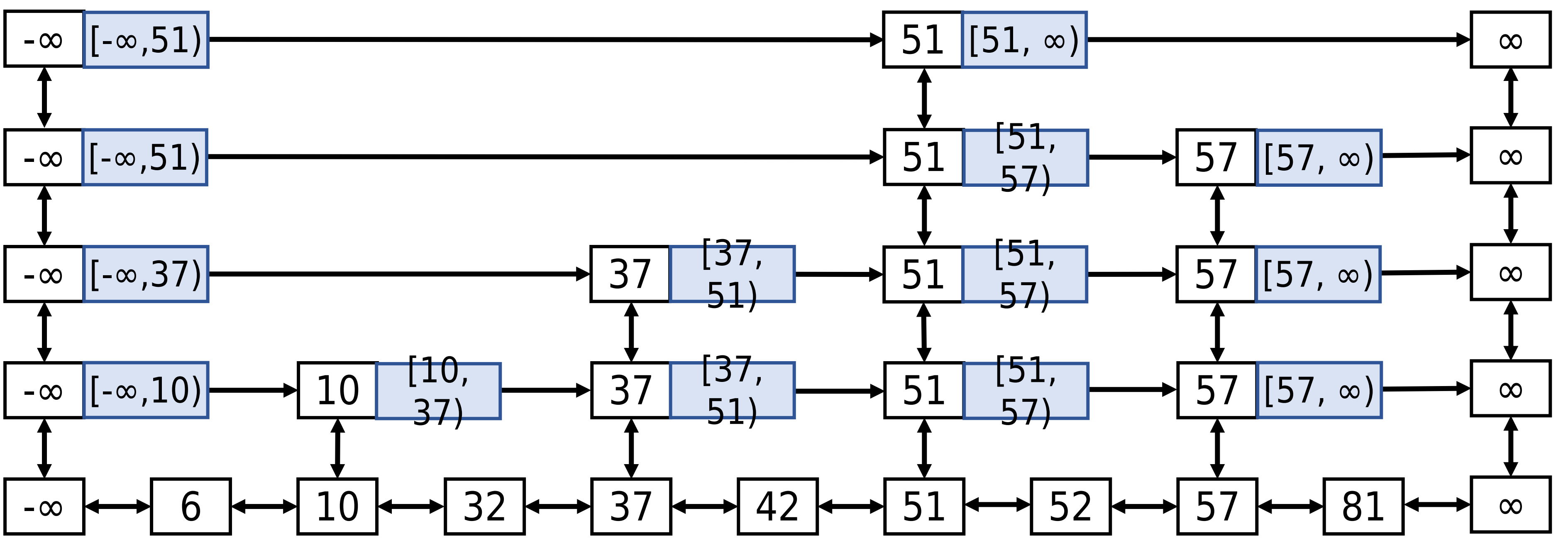


The real **SkipList**

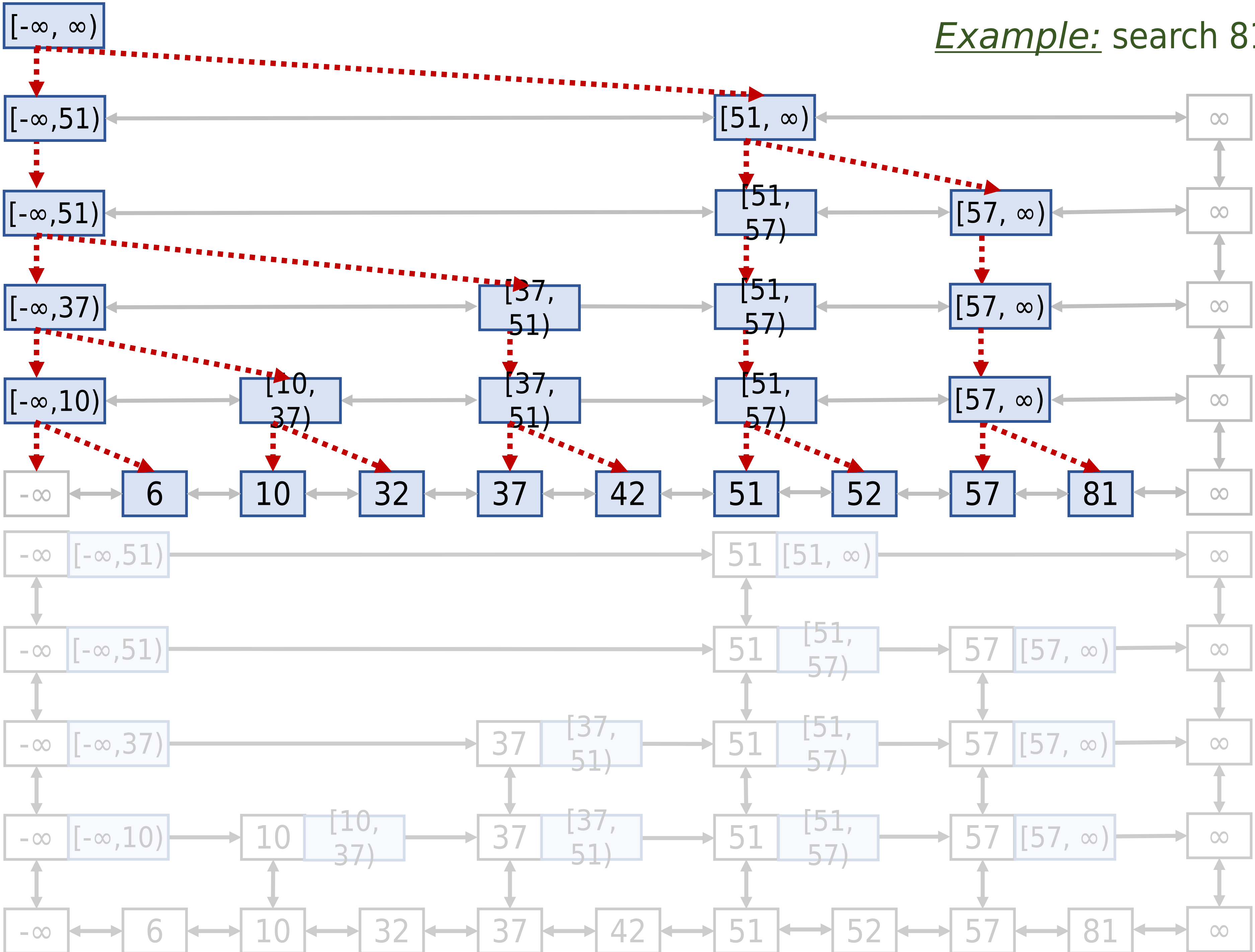
<u>Insert(L,x):</u>

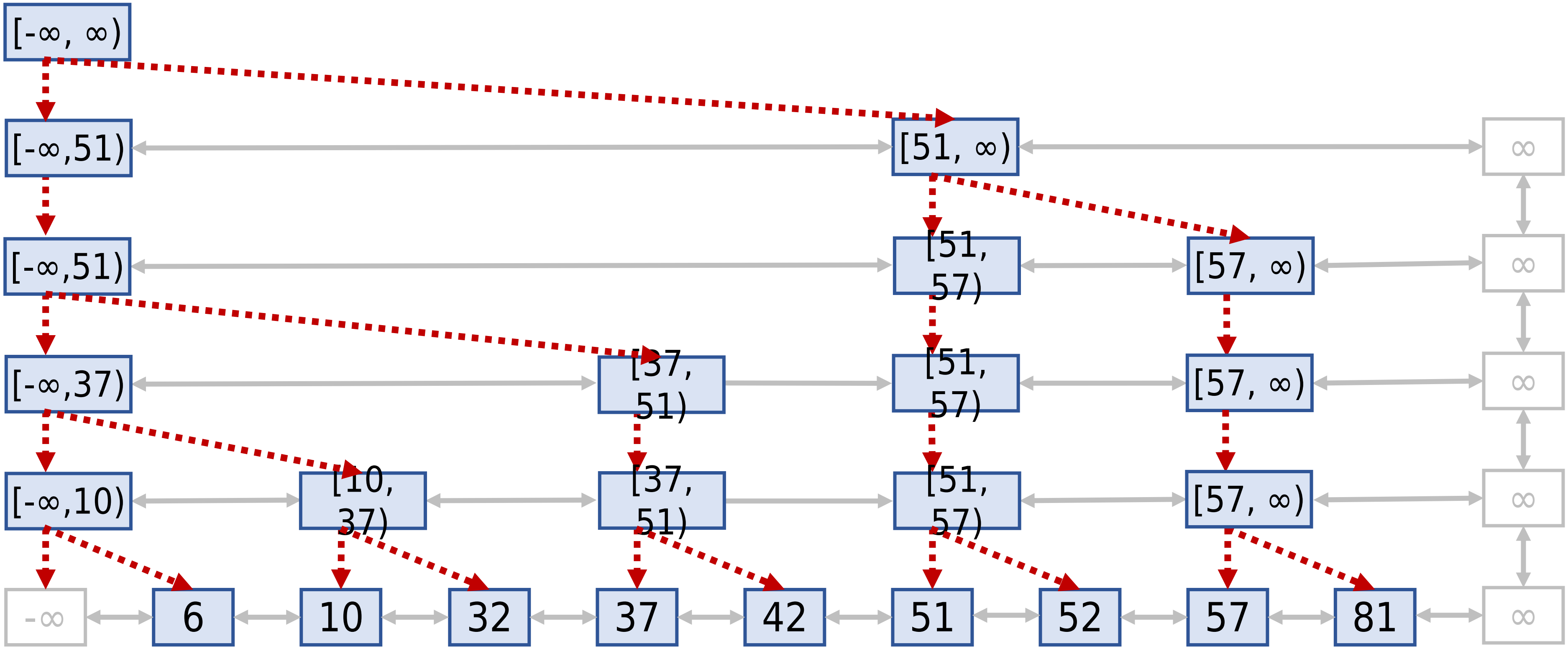
```
level = 1, done = false
while (!done)
  Insert x into level k list.
  Flip a fair coin:
    With probability 1/2: done = true
    With probability 1/2: k = k+1
```

Max level of n-element **SkipLis**
is $O(\log n)$ with high probability



Example: search 81





Each node in the “search tree” has $O(1)$ children, in expectation

Max level of n-element **SkipList** is $O(\log n)$ with high probability

Let r.v. L be the max level of a n-node SkipList, let C denote the cost for sear

$$\mathbb{P}[L \geq l] \leq n/2^{l-1}$$

Some large constant.

$$\mathbb{E}[C] = \mathbb{P}[L \leq \alpha \lg n] \cdot \mathbb{E}[C \mid L \leq \alpha \lg n] +$$

$$\sum_{l=(\alpha \lg n)+1}^{\infty} \mathbb{P}[L = l] \cdot \mathbb{E}[C \mid L = l]$$

Each node in the "search tree" has $O(1)$ children, in expectation.

Max level of n-element **SkipList** is $O(\log n)$ with high probability.

Let r.v. L be the max level of the skip list, let C denote the cost for search.

$$\mathbb{P}[L \geq l] \leq n/2^{l-1}$$

$$\mathbb{E}[C] = \mathbb{P}[L \leq \alpha \lg n] \cdot \mathbb{E}[C \mid L \leq \alpha \lg n] +$$

$$= \sum_{l=(\alpha \lg n)+1}^{\infty} \mathbb{P}[L = l] \cdot \mathbb{E}[C \mid L = l]$$

$$\leq 1 \cdot O(\lg n) = O(\lg n)$$?

$$\leq \sum_{l=(\alpha \lg n)+1}^{\infty} \mathbb{P}[L \geq l] \cdot \mathbb{E}[C \mid L = l]$$

$$\leq \sum_{l=(\alpha \lg n)+1}^{\infty} (n/2^{l-1}) \cdot (l + n)$$

$$= O(1)$$

$$\mathbb{E}[C] = O(\log n)$$

That is, search can be done in $O(\log n)$ time in expectation.

Efficient implementation of **OSet**

	Search(S,k)	**Insert(S,x)**	**Remove(S,x)**
BinarySearchTree	$O(h)$ worst-case	$O(h)$ worst-case	$O(h)$ worst-case
Treap	$O(\log n)$ in expectation	$O(\log n)$ in expectation	$O(\log n)$ in expectation
RB-Tree	$O(\log n)$ worst-case	$O(\log n)$ worst-case	$O(\log n)$ worst-case
SkipList	$O(\log n)$ in	$O(\log n)$ in	$O(\log n)$ in ectation

Performance versus Simplicity

Reading

- [CLRS] Ch.13
- [Morin] Ch.4