Lecture 4

Integer sets

Today’s plan

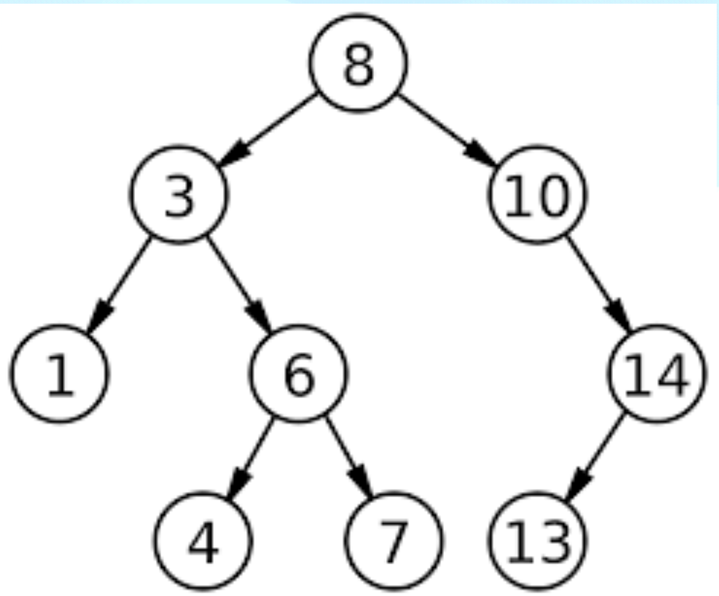
- Binary search trees
- Lower bound for sorting
- Predecessor problem

Binary search trees

Kayama Matazo

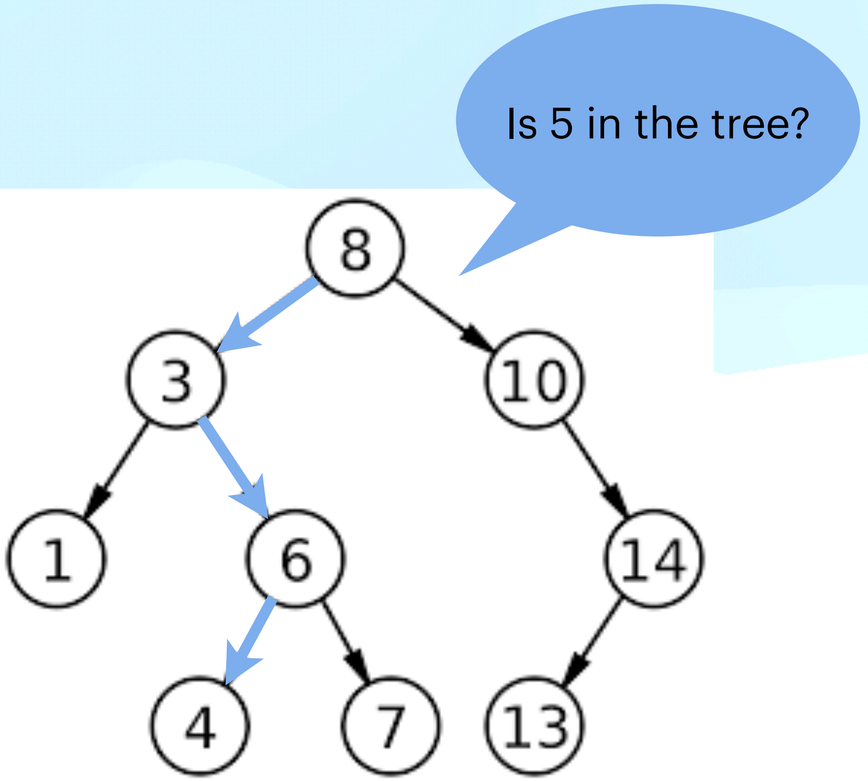
Binary search trees

- **Binary tree:** every node has at most two children
- For every node storing element ℓ, the subtree rooted at the left child (if it exists) contains elements $\leq \ell$, and the subtree rooted at the right child elements $> \ell$
- Access of a given element: $O(h)$ time, where h is the height of the tree.

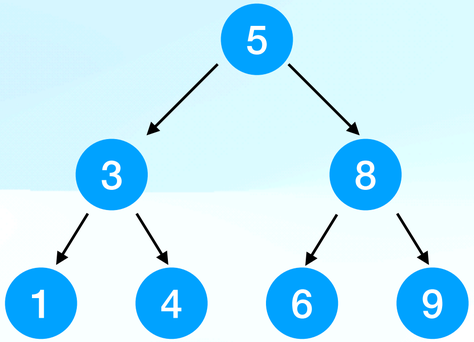


Binary search trees

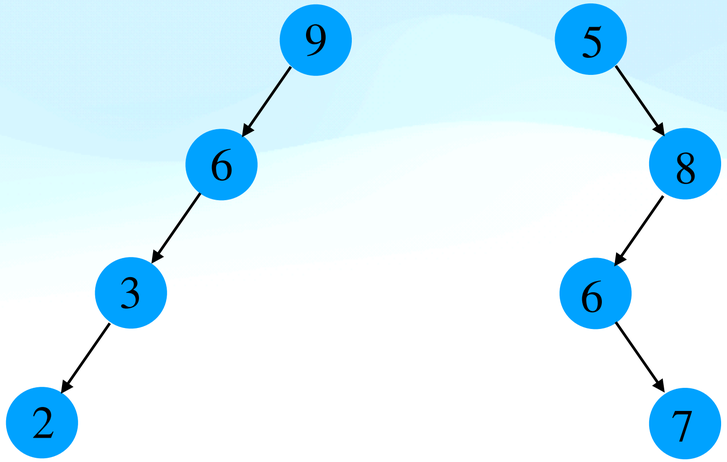
- **Binary tree:** every node has at most two children
- For every node storing element ℓ, the subtree rooted at the left child (if it exists) contains elements $\leq \ell$, and the subtree rooted at the right child elements $> \ell$
- Access of a given element: $O(h)$ time, where h is the height of the tree.



Binary search trees



7 elements, height = 3



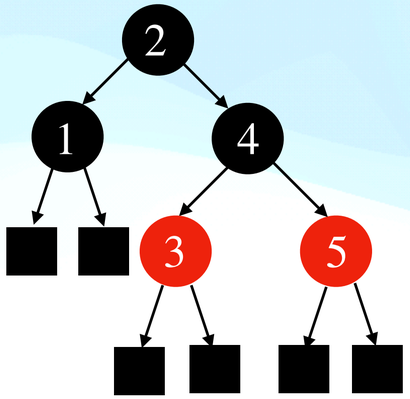
4 elements, height = 4

Implementations

- AA tree
- AVL tree
- **Red-black tree**
- Scapegoat tree
- Splay tree
- **Treap**
- Weight-balanced tree
- Tango trees

… et cetera

Red-black tree



A red-black tree is a binary tree that satisfies the following red-black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both of its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Red-black tree

Lemma. The height of a red-black tree with n nodes is at most $2\log(n+1)$.

Let $bh(x)$ be the number of black nodes in a path from x to a leaf.

By induction on $bh(x)$: the subtree rooted at x contains at least $2^{bh(x)} - 1$ nodes.

Hence, if h is the black height of the tree, $n \geq 2^h - 1$ and $h \leq \log(n+1)$.

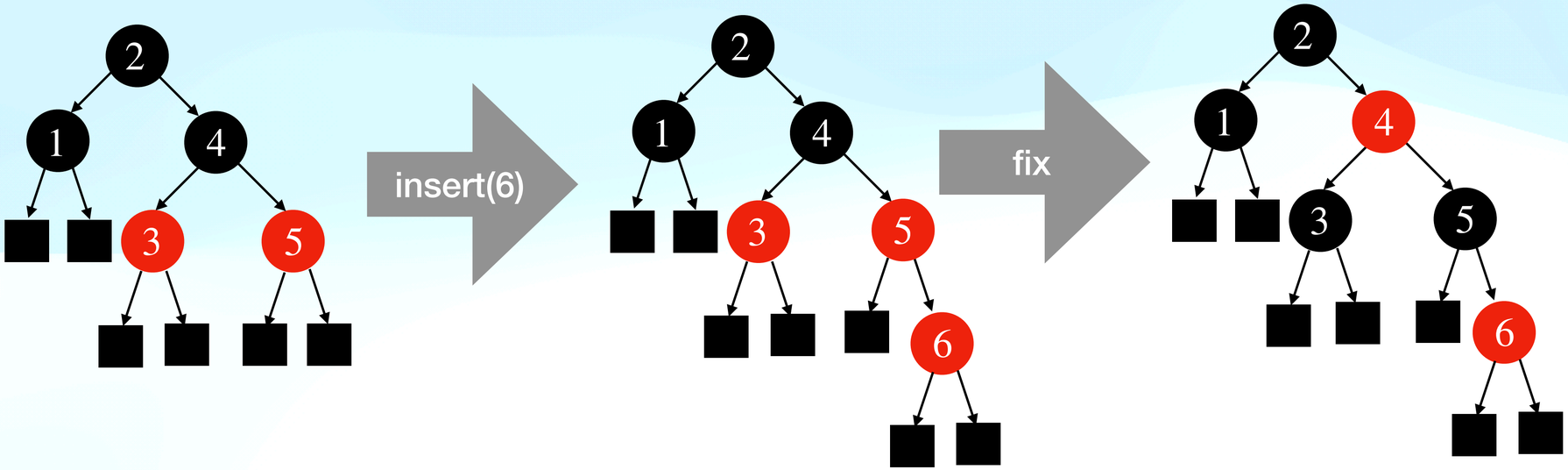
Since in any root-to-leaf path the number of nodes is at most twice the number of black nodes, the lemma follows.

Insertion

We can insert a node into an n-node red-black tree in $O(\log n)$ time.

- To do so, we insert the node into the tree T as if it were an ordinary binary search tree.
- Then we color it red.
- To guarantee that the red-black properties are preserved, we perform a number of rotations and node recolouring.

Insertion

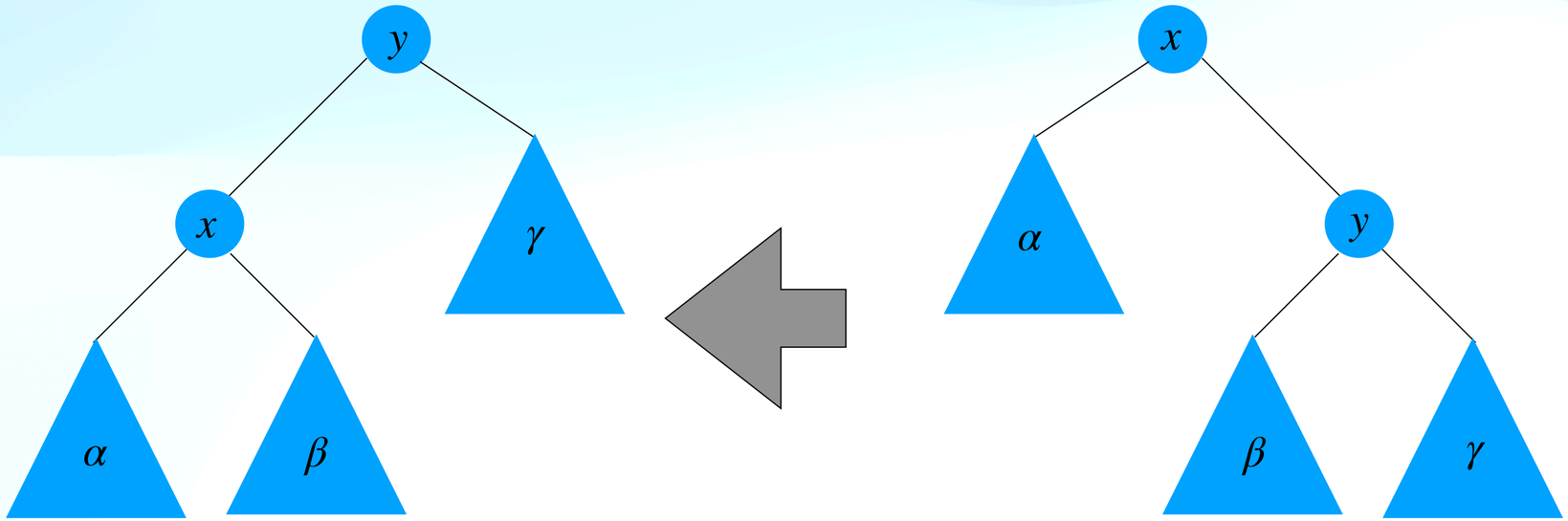


Insertion

```
RB-INSERT(T, z)
1.  y ← T.nil
2.  x ← T.root
3.  while x ≠ T.nil    //search for the placement of z
4.      y = x
5.      if z.key < x.key
6.          then x ← x.left
7.      else x ← x.right
8.  z.p = y       //y becomes the parent of z
9.  if y = T.nil
10.     then T.root ← z
11. else if z.key < y.key
12.     then y.left ← z
13. else
14.     y.right ← z
14. z.left ← T.nil    //both children of z are black leaves
15. z.right ← T.nil    //both children of z are black leaves
16. z.color← RED
17. RB-INSERT-FIXUP(T, z)
```

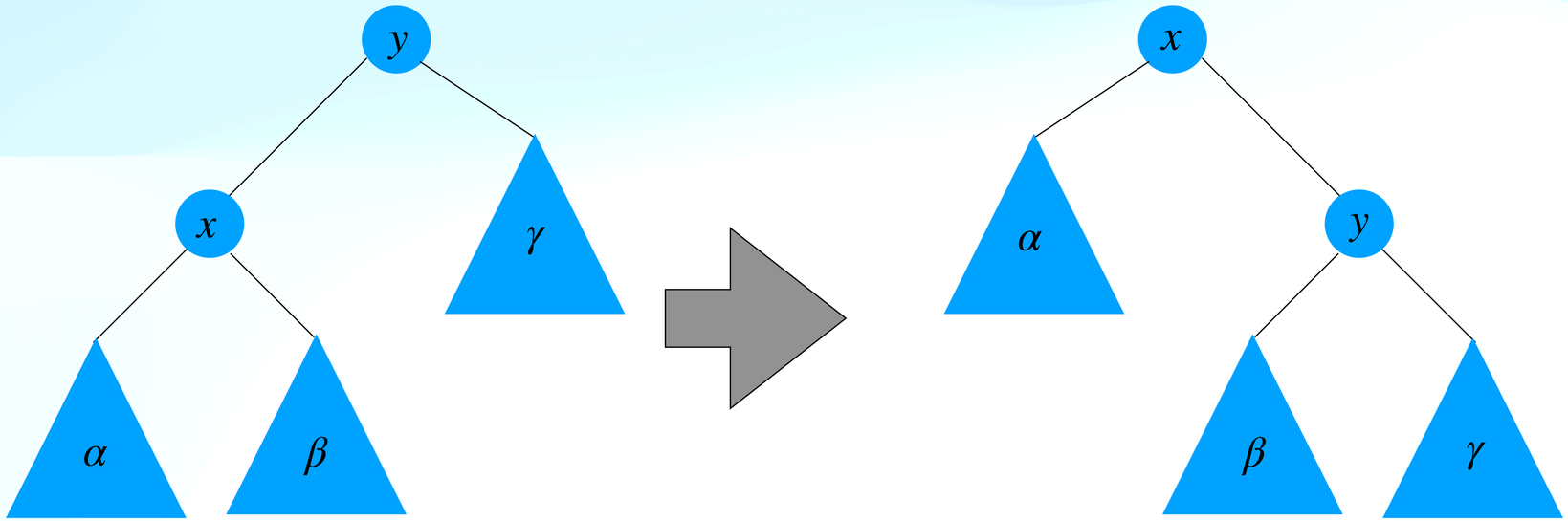
Left rotation

When we do a left rotation on a node x, we assume that its right child y is not null; x may be any node in the tree.



Right rotation

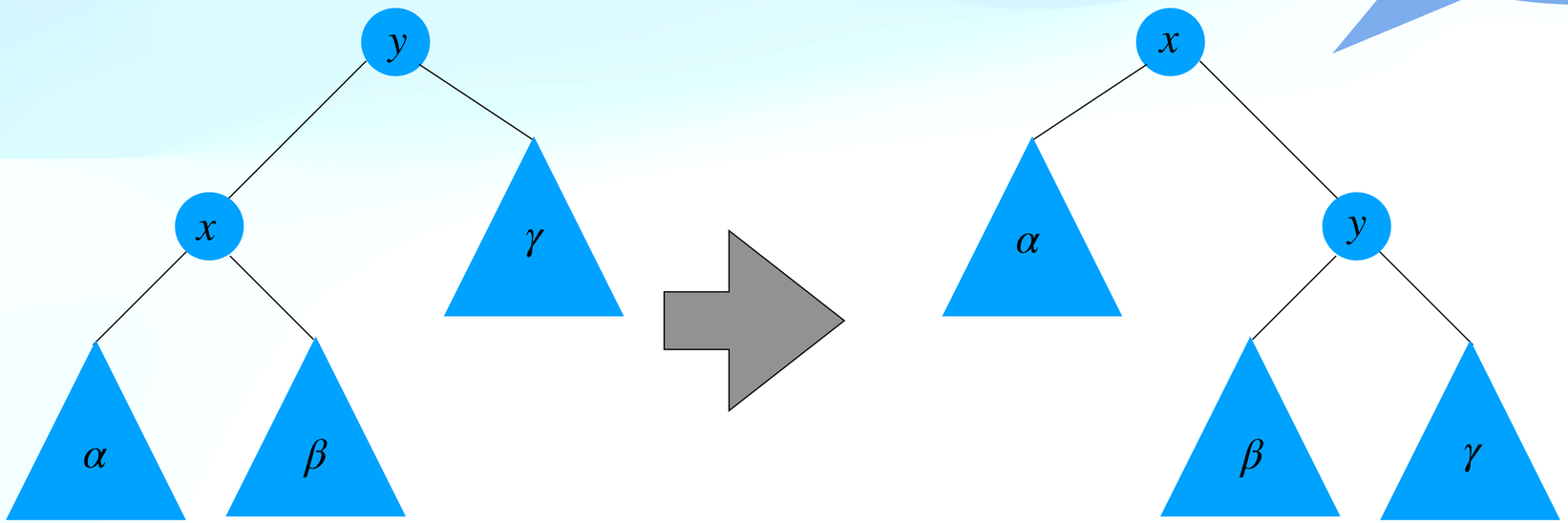
Right rotation is essentially the reverse operation.



Right rotation

Right rotation is essentially the reverse operation.

Important:
rotations do not break
the BST properties!



Restoring red-black properties

Let z be the node that violates the red-black properties.

Rule 1 is never violated. Rule 3 can be fixed in $O(1)$ time. Rule 5 is not violated as we color the new node red. Rule 4 can be violated in one of three cases:

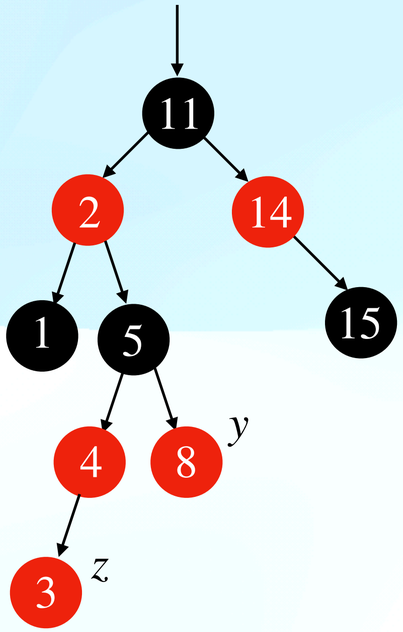
Case 1. z's uncle y is red

Case 2. z's uncle y is black and z is a right child

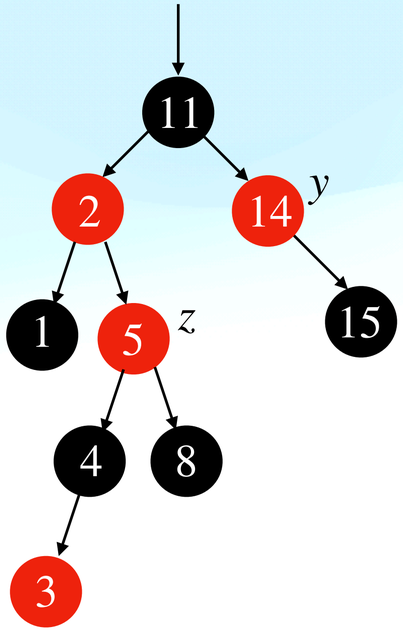
Case 3. z's uncle y is black and z is a left child

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both of its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Restoring red-black properties



color z.p BLACK,
color y BLACK,
color z.p.p RED

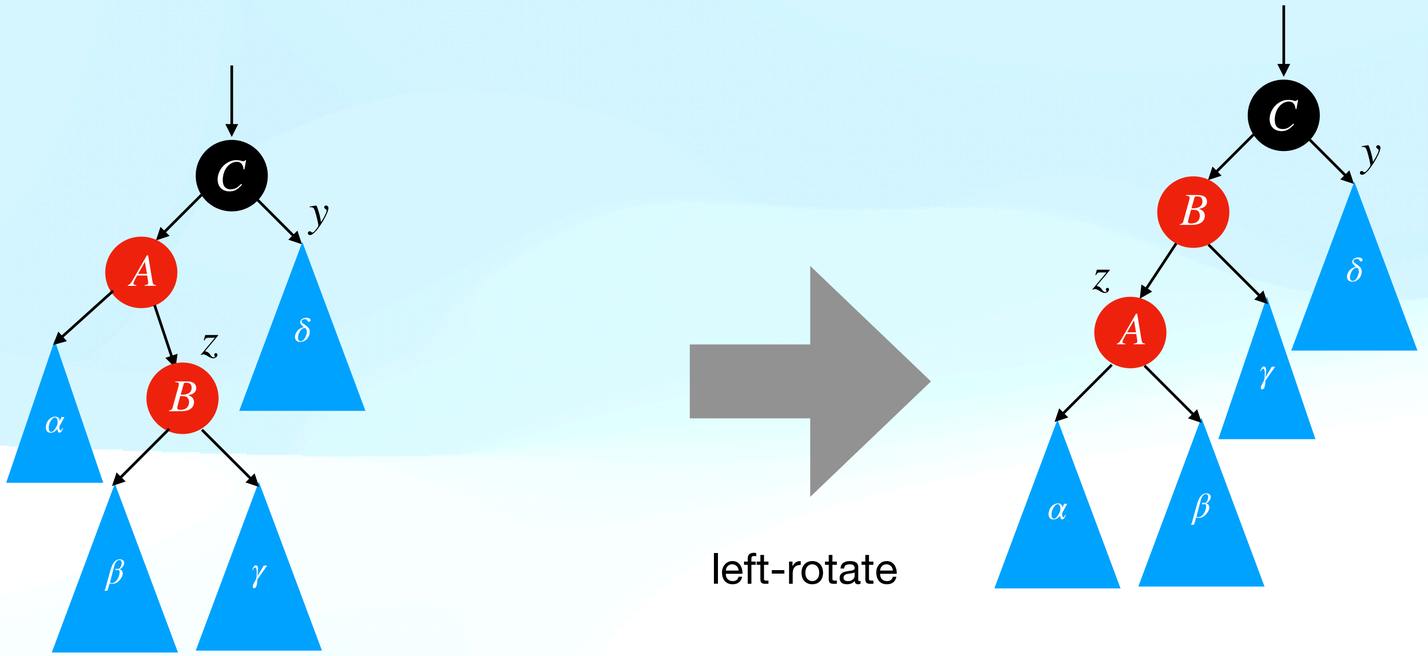


1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both of its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Case 1. z's uncle y is red

we moved z, the "bad" node, up

Restoring red-black properties

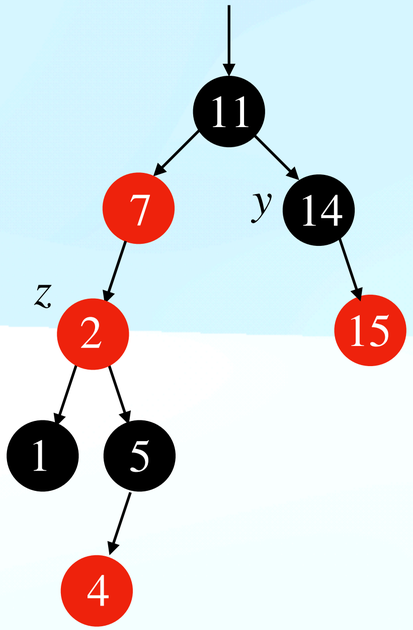


1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both of its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

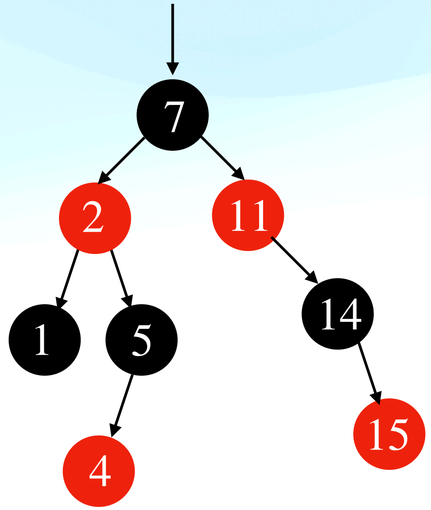
Case 2. z's uncle y is black and z is a right child

Case 3. z's uncle y is black and z is a left child

Restoring red-black properties



color z.p BLACK,
color z.p.p RED,
right-rotate



1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both of its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

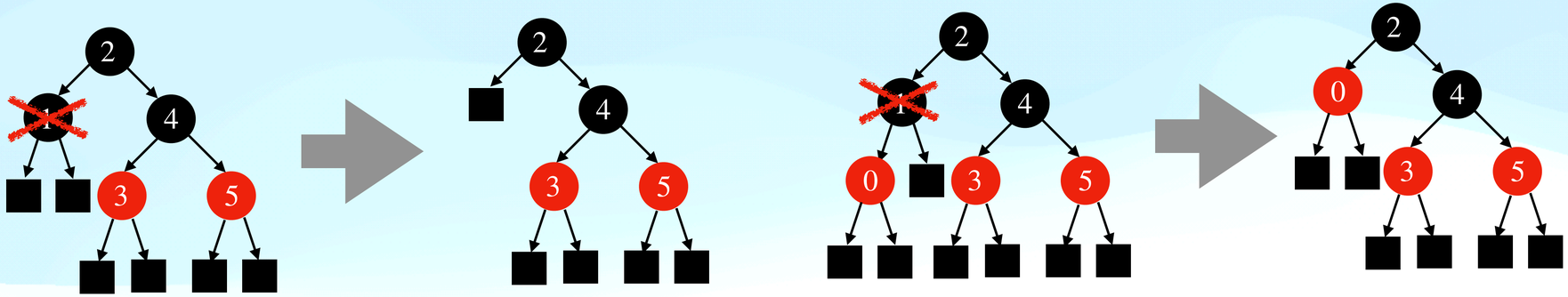
Case 3. z's uncle y is black
and z is a left child

We're good!

RB-INSERT-FIXUP

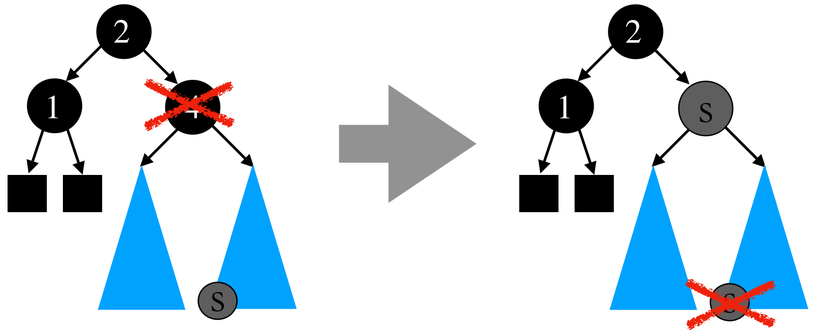
```
RB-INSERT-FIXUP(T, z)
1.  while z.p.color == RED
2.      if z.p == z.p.p.left
3.          then y ← z.p.p.right //y is the uncle of z
4.          if y.color == RED
5.              then z.p.color ← BLACK //Case 1
6.              y.color ← BLACK //Case 1
7.              z.p.p.color ← RED //Case 1
8.              z ← z.p.p //Case 1, moved z up
9.          else if z == z.p.right
10.             then z ← z.p //Case 2
11.             LEFT-ROTATE(T, z) //Case 2, reduced to Case 3
12.         z.p.color← BLACK //Case 3
13.         z.p.p.color← RED //Case 3
14.         RIGHT-ROTATE(T, z.p.p) //Case 3
15.     else same as then clause with "right" and "left" exchanged
16. T.root.color← BLACK
```

Deletion



Case 1: node has no children

Case 2: node has one child



Case 3: node has two children
S - successor of the node to delete

Deletion

First, we need to design the **RB-TRANSPLANT** procedure that puts a node v in place of a node u.

RB-TRANSPLANT(T, u, v)

```
1. if u.p == T.nil
2.       T.root = v
3. else if u == u.p.left
4.     u.p.left = v
5. else
6.     u.p.right = v
7. v.p = u.p
```

Deletion

RB-DELETE (T, z)

```
1.  if z.left == T.nil or z.right == T.nil then y = z else y = TREE-
MINIMUM(z.right)
2.  if y.left != T.nil then x = y.left else x = y.right //x is a non-nil child of y
3.  x.p = y.p //move x into y's place deleting y
4.  if y.p = T.nil then
5.      T.root = x
6.  else
7.      if y = y.p.left then
8.          y.p.left = x
9.      else
10.         y.p.right = x
11. if y != z then z.key = y.key //Case 3, move the successor to z
12. if y.color == BLACK then RB-DELETE-FIXUP(T, x)
13. return y
```

Deletion

- If y.color = **red**, no red-black properties are violated.
- If y.color = **black**, we can violate three properties:
  1. **Property 2:** The root is black (when y is the root and we move y's child x, possibly red, in its place)
  2. **Property 4:** If a node is red, then both its children are black
  3. **Property 5:** For each node, all simple paths from the node to descendant leaves contain the same number of black nodes (every root-to-leaf path containing node y now contains one less black node)

RB-DELETE-FIXUP

We now need to show how to restore the red-black properties. Let x be the node that violates the red-black properties. We will consider four cases:

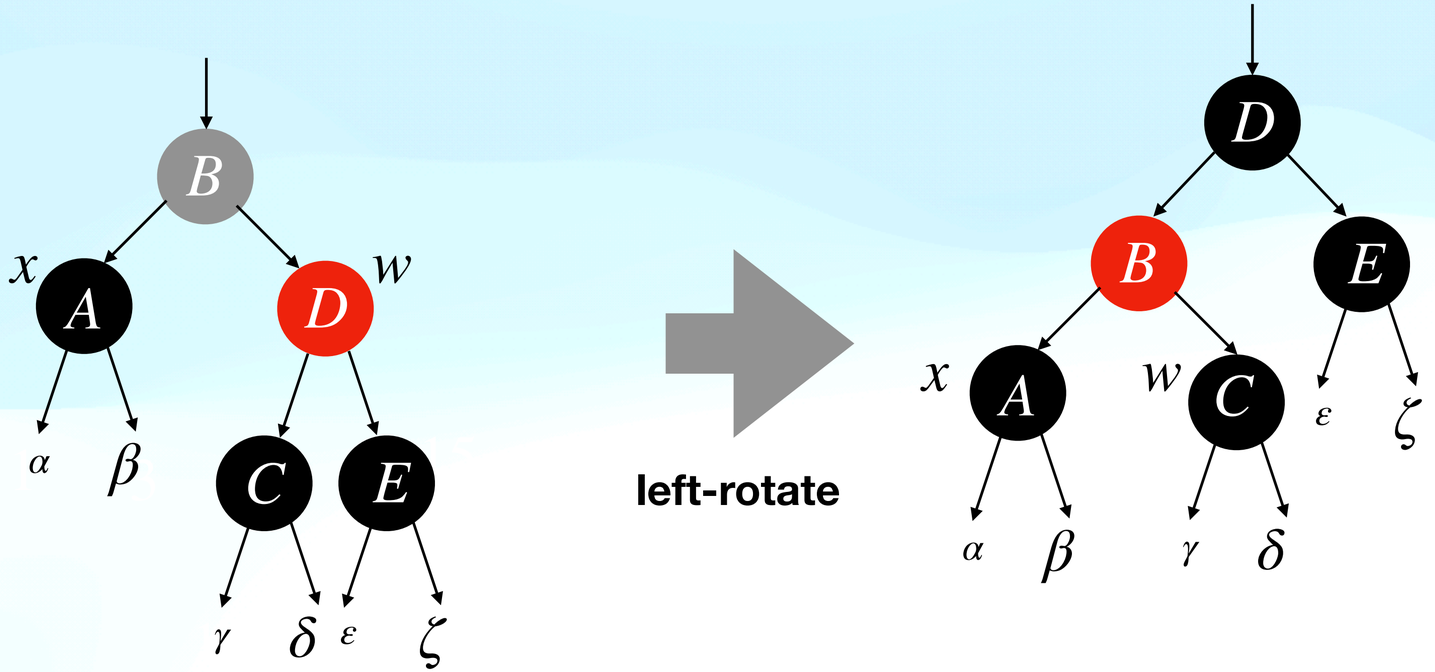
Case 1: x's sibling w is red

Case 2: x's sibling w is black, and both children of w are black

Case 3: x's sibling w is black, w's left child is red, and w's right child is black

Case 4: x's sibling w is black, and w's right child is red

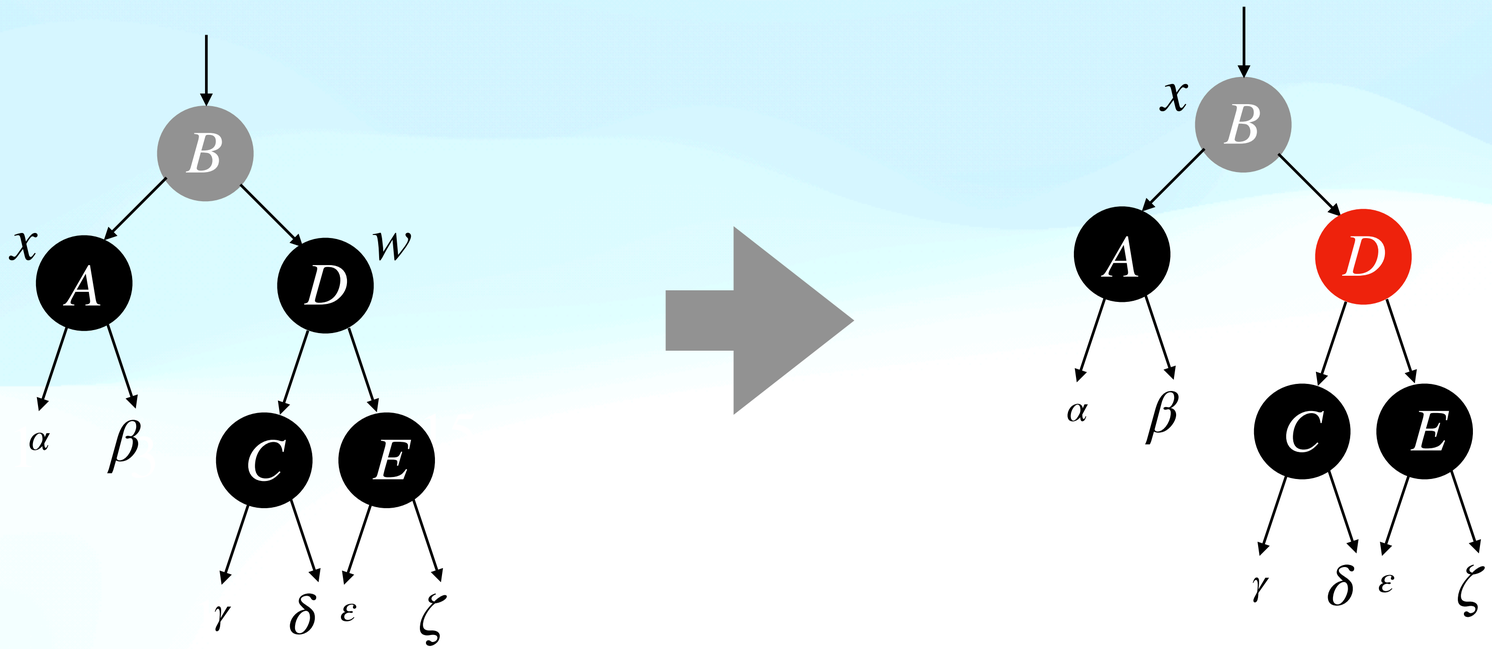
RB-DELETE-FIXUP



Case 1: x's sibling w is red

Reduces to **Cases 2-4**
depth x = depth x + 1

RB-DELETE-FIXUP

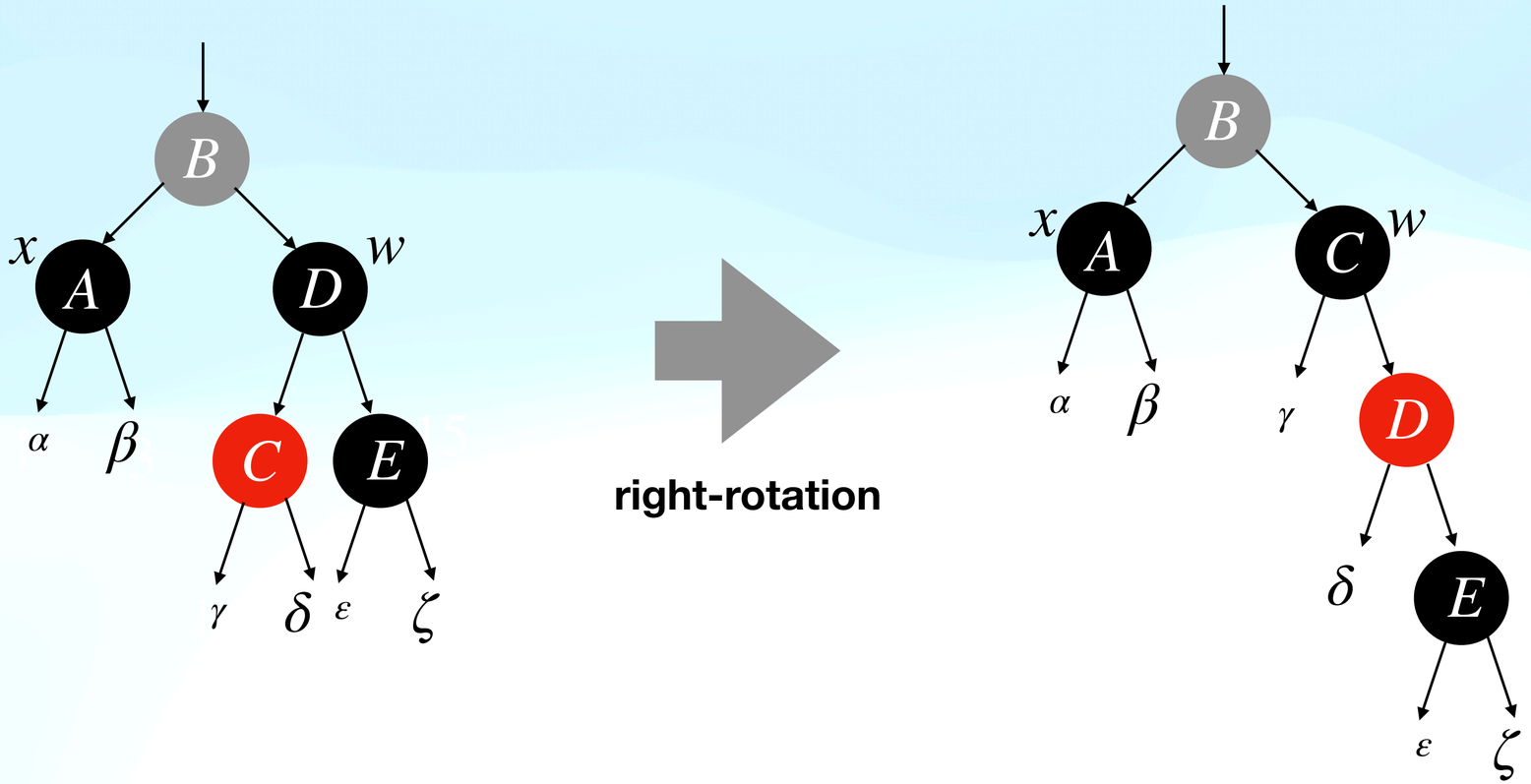


Case 2: x's sibling w is black, and both children of w are black

we can only violate Property 5 here, and colouring D red moves x to B, **depth x = depth x - 1**

NB: if we came to Case 2 from Case 1, we cannot go back to Case 1!

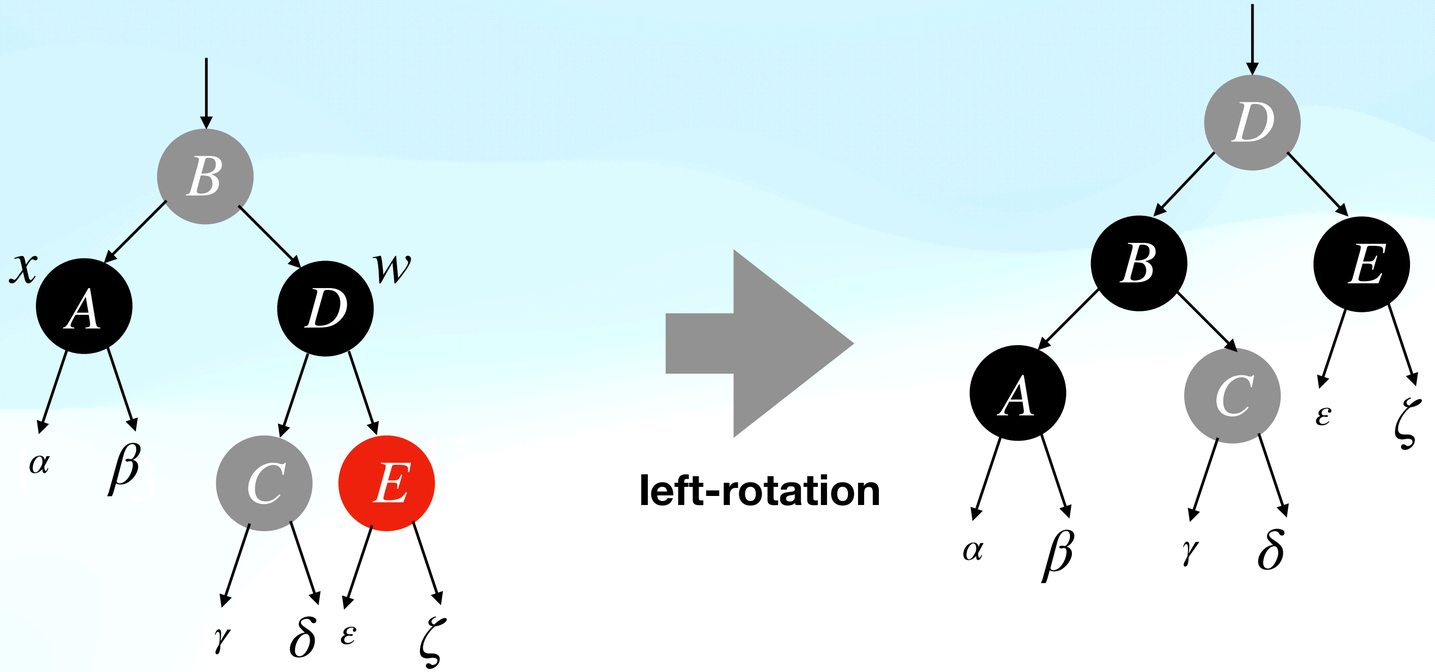
RB-DELETE-FIXUP



Case 3: x's sibling w is black, and
w's left child is red, and w's right child is black

Reduces to Case 4
depth x = depth x

RB-DELETE-FIXUP



Case 4: x's sibling w is black,
and w's right child is red

$$x = T.root$$

we are done!

RB-DELETE-FIXUP

```
RB-DELETE-FIXUP(T, x)
1.   while x !=T.root and x.color == BLACK
2.       if x == x.p.left
3.           w = x.p.right
4.           if w.color == RED
5.               w.color = BLACK //Case 1
6.               x.p.color = RED //Case 1
7.               LEFT-ROTATE(T, x.p) //Case 1
8.               w = x.p.right //Case 1
9.           if w.left.color == BLACK and w.right.color == BLACK
10.              w.color = RED //Case 2
11.              x = x.p //Case 2
```

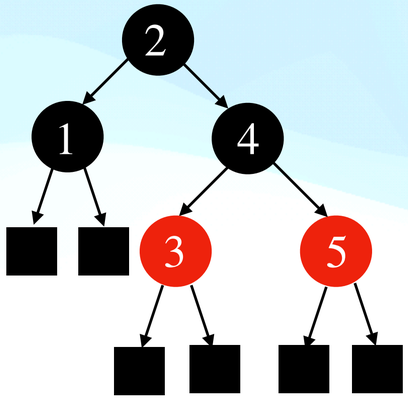
$$T = O(\log n)$$

RB-DELETE-FIXUP

```
12.       else if w.right.color == BLACK
13.             w.left.color = BLACK //Case 3
14.             w.color = RED //Case 3
15.             RIGHT-ROTATE(T, w) //Case 3
16.             w = x.p.right //Case 3
17.           w.color = x.p.color //Case 4
18.           x.p.color = BLACK //Case 4
19.           w.right.color = BLACK //Case 4
20.           LEFT-ROTATE(T, x.p) //Case 4
21.           x = T.root //Case 4
22. else same as then clause with “right” and “left” exchanged
23. x.color = BLACK
```

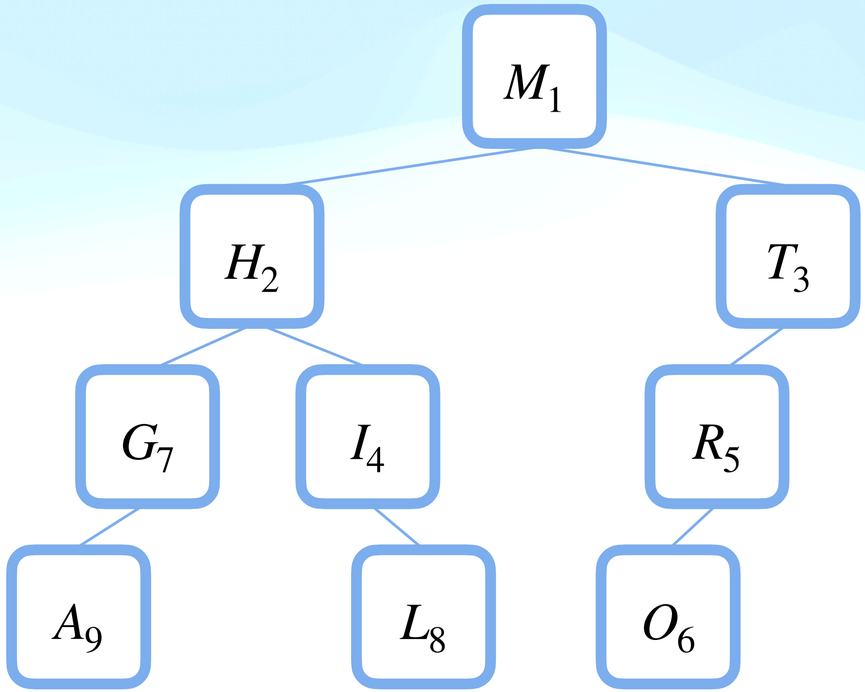
Red-black tree

- Uses $O(n)$ space
- Insertion time: $O(\log n)$ in the worst case
- Deletion time: $O(\log n)$ in the worst case
  (more tedious than deletion, but can be done, you can look it up in the full version of the slides)



Treaps

- Every node has a key and a priority;
- A treap is a binary search tree for the keys;
- And a min-heap for the priorities (i.e. the priority of a node is always *smaller* than the priorities of its children).

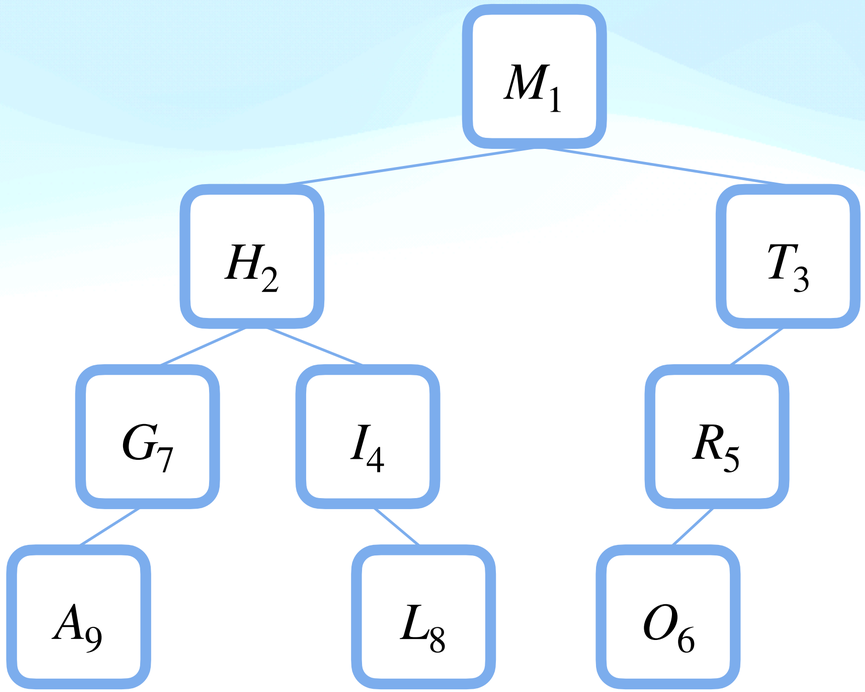


Treaps

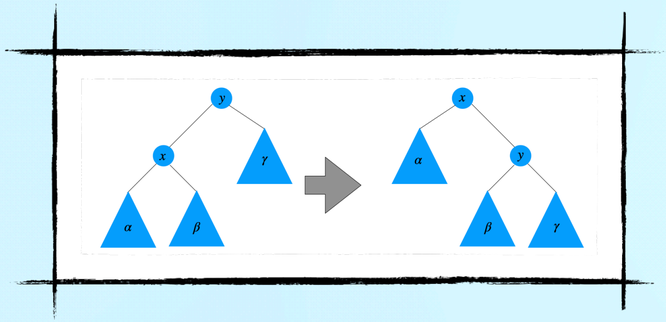
Search: the usual algorithm

Time for a successful search = O(depth of the node)

Time for an unsuccessful search = $O(\max\{\text{depth}(v^-), \text{depth}(v^+)\})$, where key($v^-$) (key($v^+$)) is the predecessor (successor) of k.



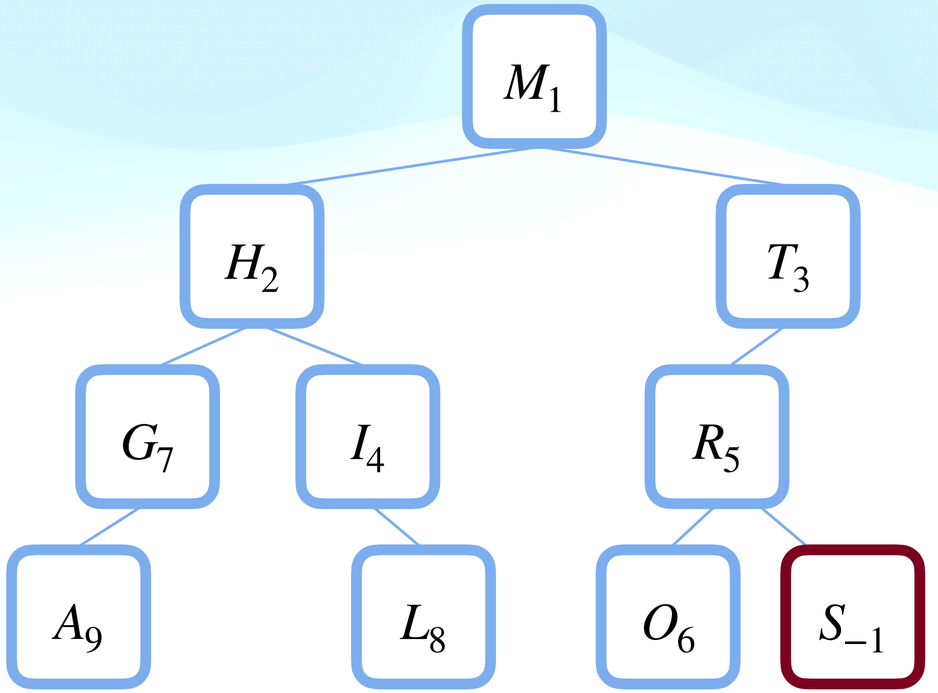
Treaps



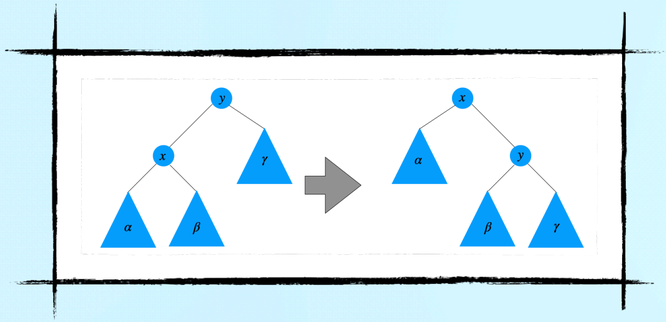
Insertion: standard BST insertion algorithm + rotations to fix the heap properties.

Namely, if the priority of a node z is smaller than the priority of parent(z), rotate around the edge (z, parent(z)).

As a result, the depth of the node where the heap rule is violated decreases.



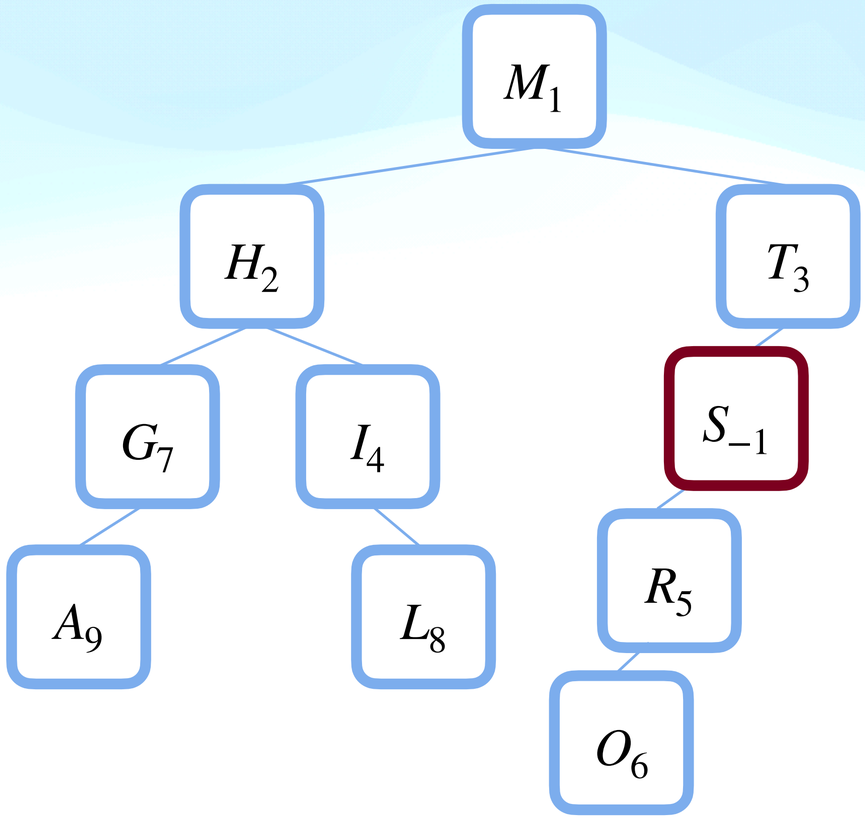
Treaps



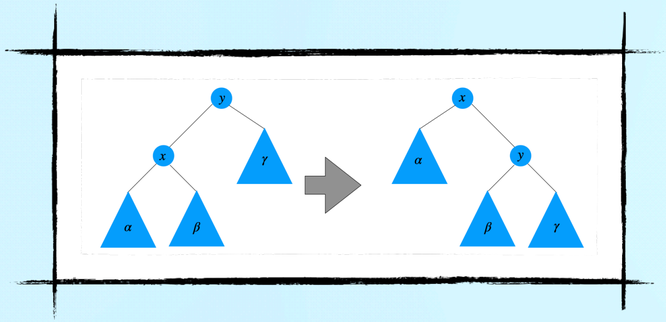
Insertion: standard BST insertion algorithm + rotations to fix the heap properties.

Namely, if the priority of a node z is smaller than the priority of parent(z), rotate around the edge (z, parent(z)).

As a result, the depth of the node where the heap rule is violated decreases.



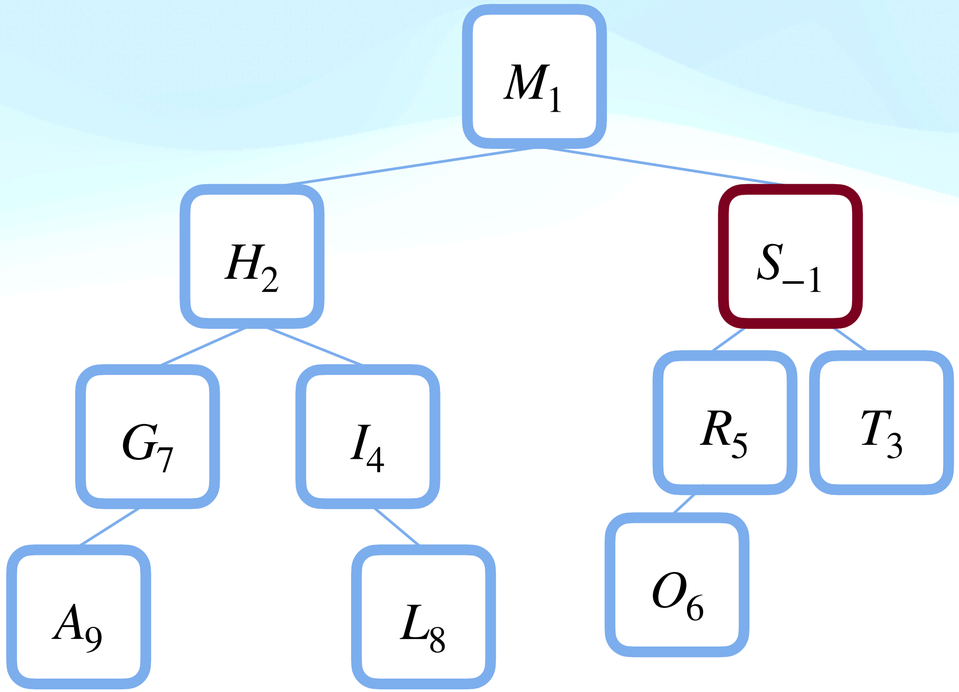
Treaps



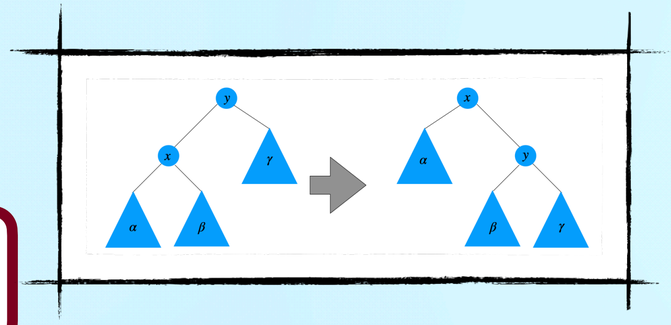
Insertion: standard BST insertion algorithm + rotations to fix the heap properties.

Namely, if the priority of a node z is smaller than the priority of parent(z), rotate around the edge (z, parent(z)).

As a result, the depth of the node where the heap rule is violated decreases.



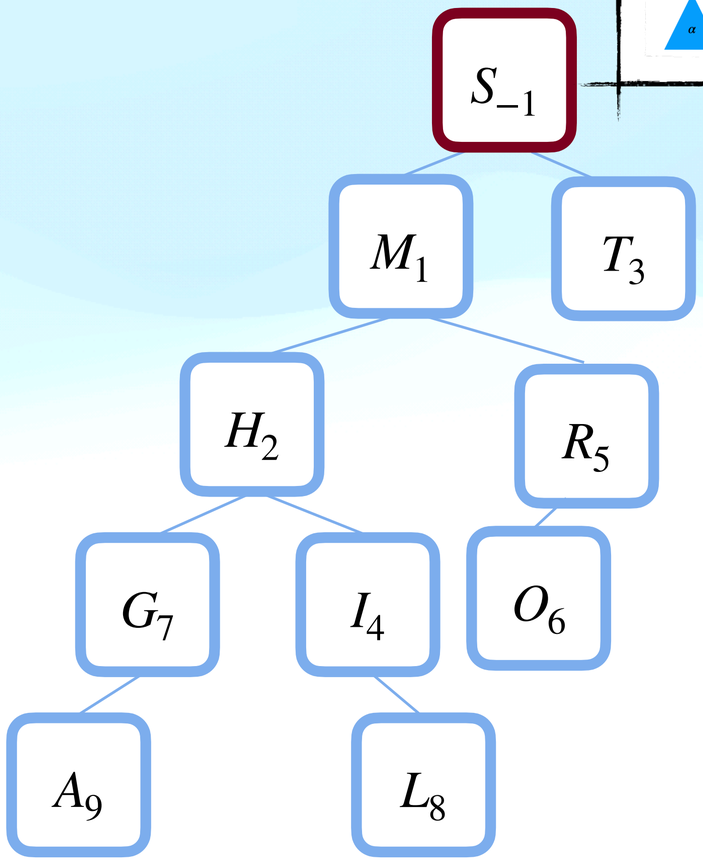
Treaps



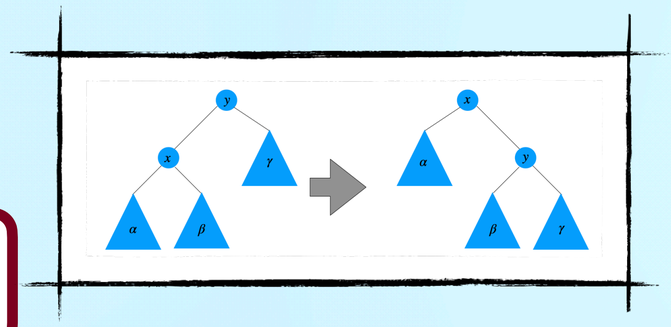
Insertion: standard BST insertion algorithm + rotations to fix the heap properties.

Namely, if the priority of a node z is smaller than the priority of parent(z), rotate around the edge (z, parent(z)).

As a result, the depth of the node where the heap rule is violated decreases.



Treaps

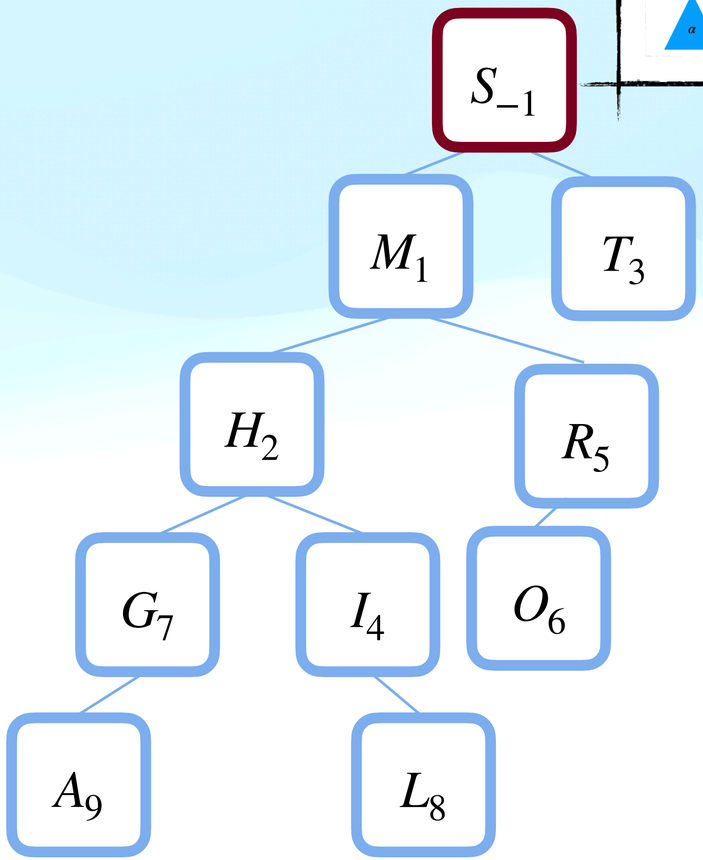


Deletion: run the insertion algorithm backwards

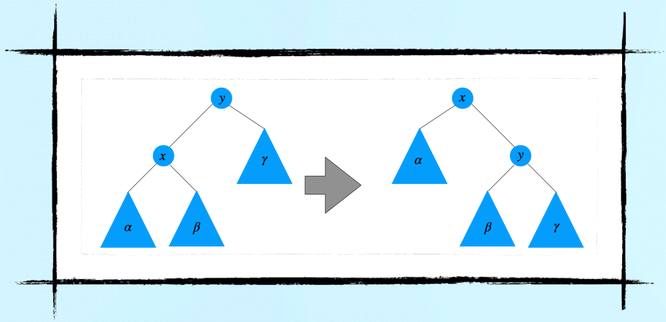
Let light(z) be the child of z with smaller priority. As long as z (the node to delete) is not a leaf, rotate around (z, light(z)).

The choice of the edge preserves the heap property everywhere except at z.

When z becomes a leaf, chop if off.



Treaps

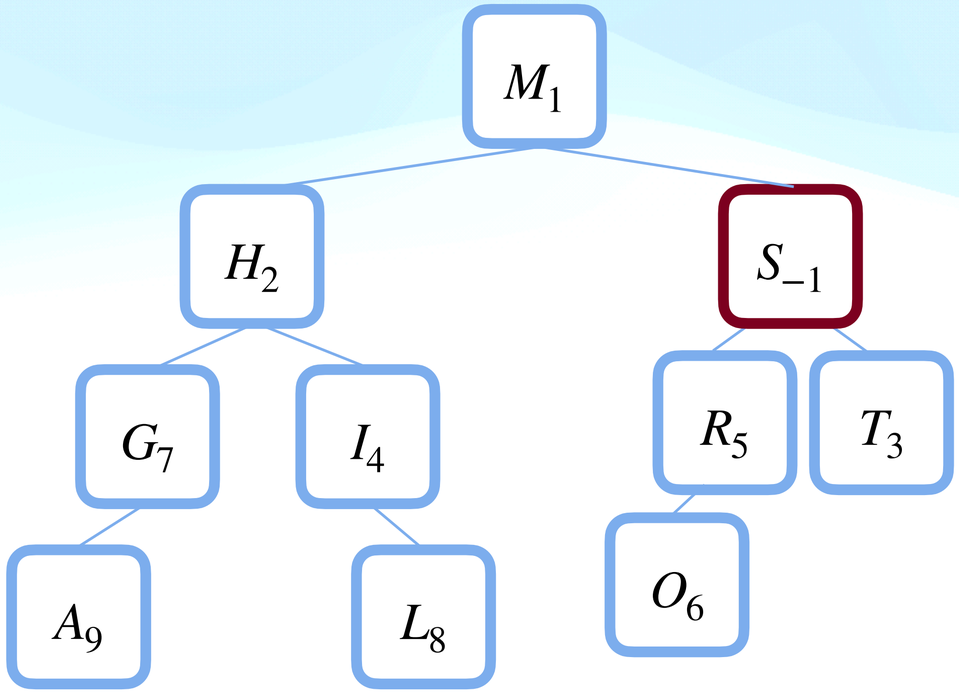


Deletion: run the insertion algorithm backwards

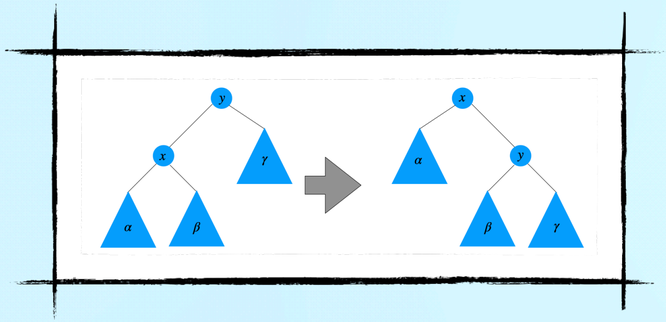
Let light(z) be the child of z with smaller priority. As long as z (the node to delete) is not a leaf, rotate around (z, light(z)).

The choice of the edge preserves the heap property everywhere except at z.

When z becomes a leaf, chop if off.



Treaps

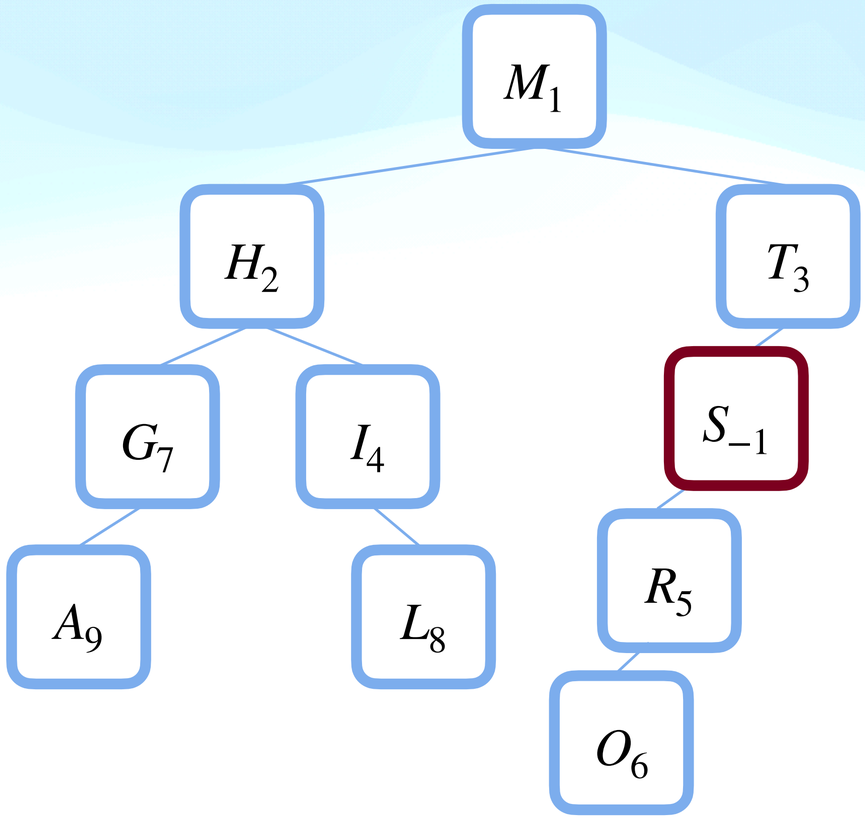


Deletion: run the insertion algorithm “backwards”

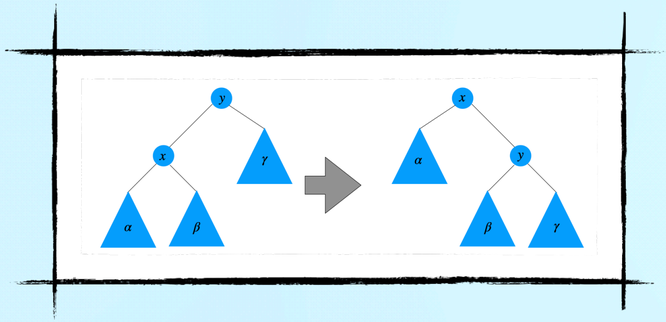
Let $\mathsf{light}(z)$ be the child of z with smaller priority. As long as z (the node to delete) is not a leaf, rotate around $(z, \mathsf{light}(z))$.

The choice of the edge preserves the heap property everywhere except at z.

When z becomes a leaf, chop if off.



Treaps

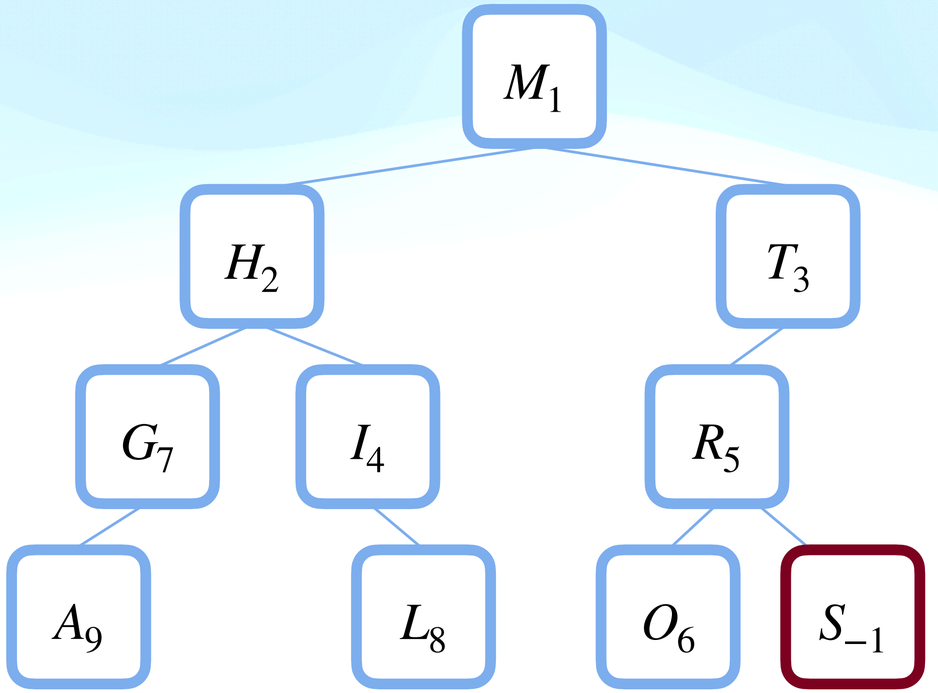


Deletion: run the insertion algorithm backwards

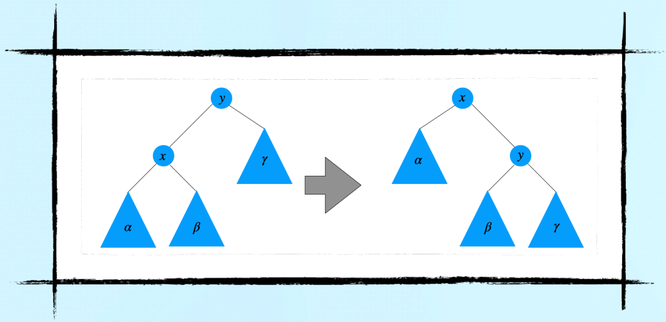
Let $\mathsf{light}(z)$ be the child of z with smaller priority. As long as z (the node to delete) is not a leaf, rotate around $(z, \mathsf{light}(z))$.

The choice of the edge preserves the heap property everywhere except at z.

When z becomes a leaf, chop if off.



Treaps

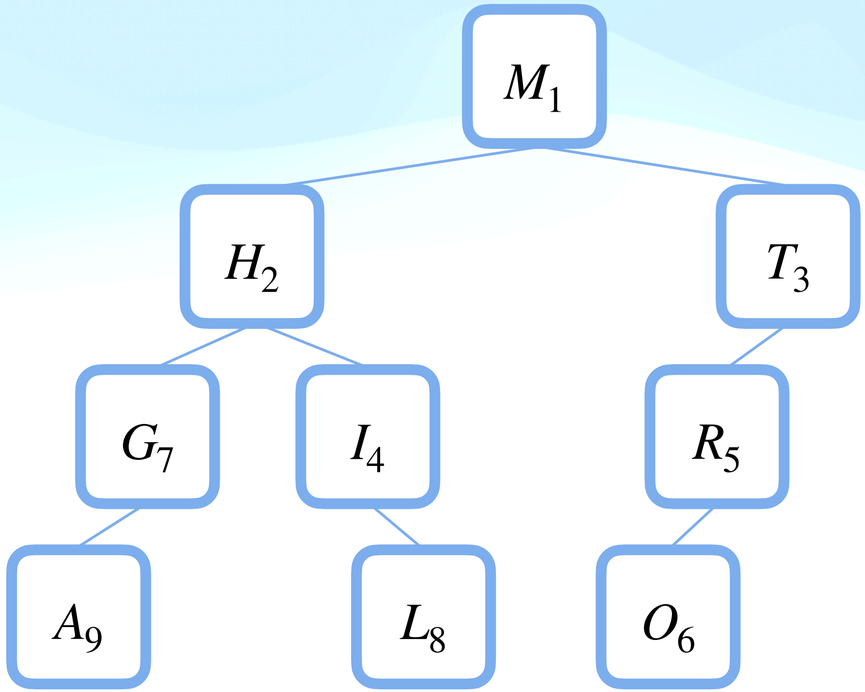


Deletion: run the insertion algorithm backwards

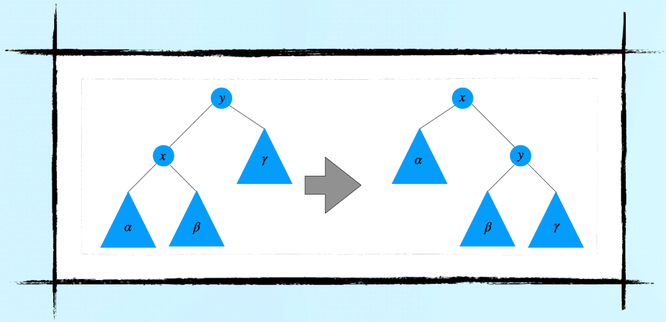
Let light(z) be the child of z with smaller priority. As long as z (the node to delete) is not a leaf, rotate around (z, light(z)).

The choice of the edge preserves the heap property everywhere except at z.

When z becomes a leaf, chop if off.

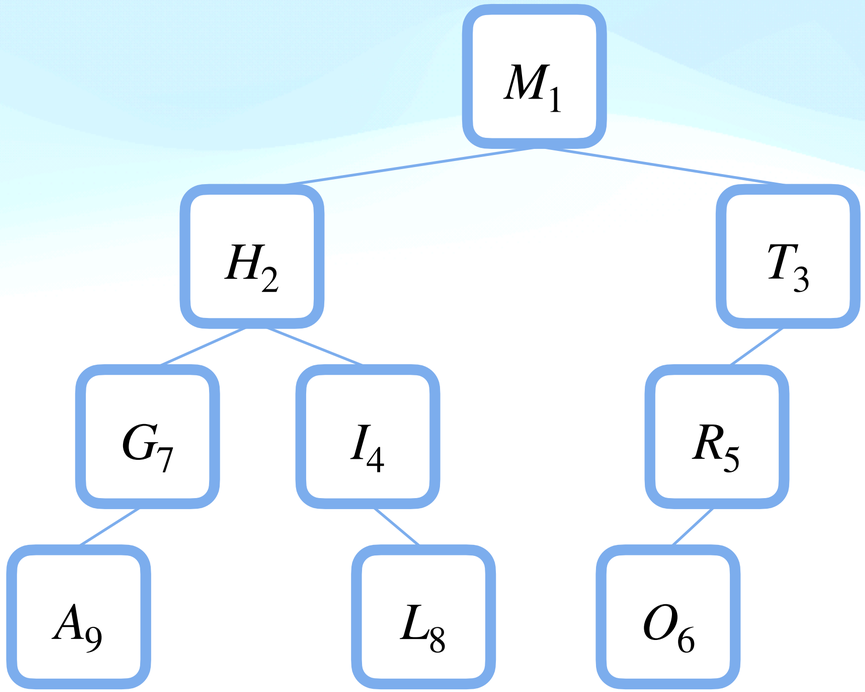


Treaps



Split: if we want to split the treap along a pivot π, we can insert a new node z with key π and priority $-\infty$.

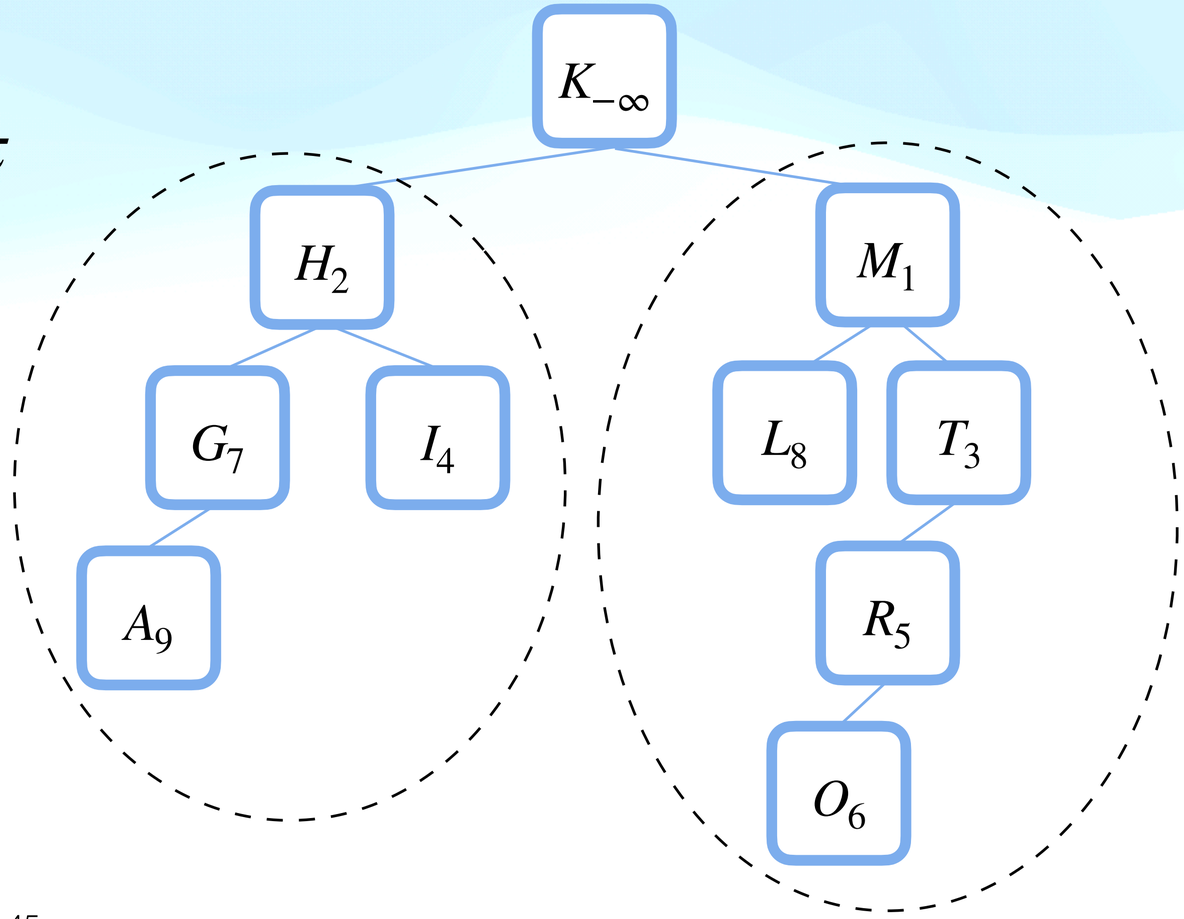
After the insertion, the new node is the root, all keys smaller than π belong to the left subtree of the root and all keys larger than π belong to the right subtree of the root



Treaps

Split: if we want to split the treap along a pivot π, we can insert a new node z with key π and priority $-\infty$.

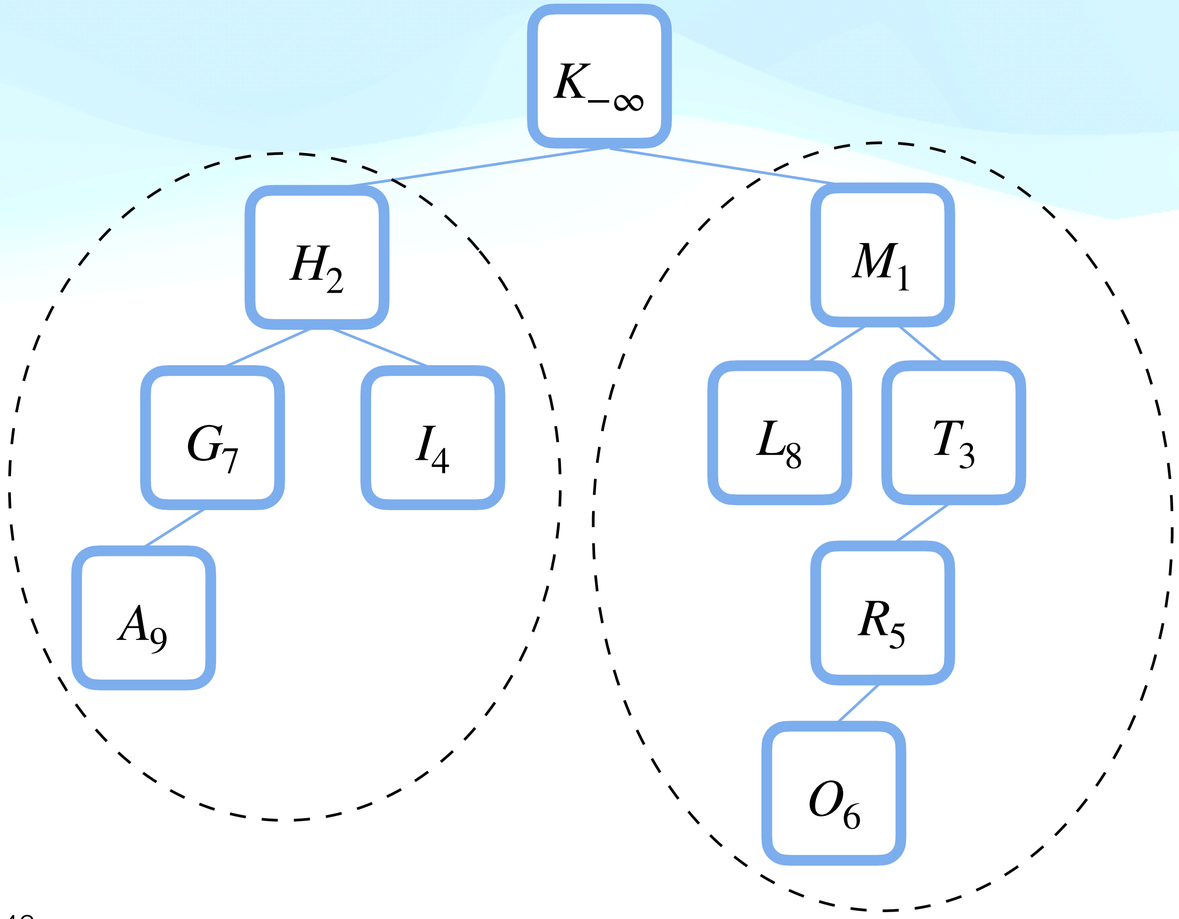
After the insertion, the new node is the root, all keys smaller than π belong to the left subtree of the root and all keys larger than π belong to the right subtree of the root



Treaps

Merge: Suppose we want to merge two treaps $T_<$ and $T_>$, where every key in $T_<$ is smaller than every key in $T_>$.

We can create a dummy root with priority $-\infty$, hang both $T_<$ and $T_>$ from the root, and rotate the root down to a leaf, where we can cut it off.



Treaps

- **Time for a successful search:** $O(\text{depth}(v))$, where $\text{key}(v) = k$.
- **Time for an unsuccessful search:** $O(\max\{\text{depth}(v^-), \text{depth}(v^+)\})$, where $\text{key}(v^-)$ ($\text{key}(v^+)$) is the predecessor (successor) of k.
- **Insertion time:** $O(\max\{\text{depth}(v^-), \text{depth}(v^+)\})$
- **Deletion time:** $O(\text{tree depth})$
- **Split/merge time:** same as insertion / deletion

Random priorities

priorities = independently and uniformly distributed continuous random variables

$x_1, x_2, \ldots, x_n$ — nodes of the tree in the increasing order of the keys

$i \uparrow k$ — x_i is a proper ancestor of x_k, $\mathsf{depth}(x_k) = \sum_{i=1}^{i=n} [i \uparrow k]$

$$\mathbb{E}[\mathsf{depth}(x_k)] = \sum_{i=1}^{i=n} \mathbb{E}[[i \uparrow k]] = \sum_{i=1}^{i=n} P[i \uparrow k]$$

Random priorities

$$X(i,k) = \begin{cases} x_i, x_{i+1}, \ldots, x_k & \text{if } i < k; \\ x_k, x_{k+1}, \ldots, x_i & \text{otherwise .} \end{cases}$$

Lemma. For all $i \neq k$, we have $i \uparrow k$ iff x_i has the smallest priority in $X(i,k)$.

1. If x_i is the root, then x_i has the smallest priority and $i \uparrow k$
2. If x_k is the root, then x_i is not an ancestor of x_k and x_i does not have the smallest priority (x_k does)
3. If x_j is the root, $i < j < k$, then x_i is not an ancestor of x_k and x_i does not have the smallest priority (x_j does)
4. If x_j is the root, and either $i < k < j$ or $i < j < k$, then x_i and x_j are in the same subtree and the claim follows by induction

Random priorities

As a corollary,

$$P[i \uparrow k] = \begin{cases} \dfrac{1}{k-i+1} & \text{if } i < k; \\ 0 & \text{if } i = k; \\ \dfrac{1}{i-k+1} & \text{if } i > k\,. \end{cases}$$

$$\mathbb{E}[\text{depth}(x_k)] = \sum_{i=1}^{i=n} P[i \uparrow k] = \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} = H_k - 1 + H_{n-k+1} - 1 < 2\ln n - 2$$

Hence, **all treap operations take $O(\log n)$ time in expectation**

Quicksort

In practice, the fastest sorting algorithm is Quicksort, which uses partitioning as its main idea.

Example: pivot about 10

Before: 17 12 6 19 23 8 5 10

After: 6 8 5 10 23 19 12 17

$O(n \log_2 n)$ **expected time!**

Choose a pivot as a random element of the array, and place all the elements less than the pivot in the left part of the array, and all elements greater than the pivot in the right part of the array. The pivot fits in the slot inbetween.

Note that the pivot element ends up in the correct place in the total order!

Then sort the two halves of the array recursively.

Quicksort

Another view: insert numbers into a binary search tree in random order (= with random priorities) and output the inorder sequence of keys in the tree

Why is this a quicksort? The first “pivot” is the root of the tree.

In the recursive formulation, we compare the pivot with every other element of the array to partition it into two subarrays.

In the binary tree formulation, we place all smaller number into the left subtree and all numbers larger than the pivot into the right subtree.

Either way, we compare the pivot with every other number. The rest is due to recursive structure of both approaches.

Open question: Dynamic optimality

- In practice, we access some elements more often than others. Can we use this to make the search faster?
- Standard operations on BSTs: move the pointer to a child / the parent of the pointer, perform a rotation on the pointer and its parent
- For a fixed access sequence X, let $OPT(X)$ be the number of unit-cost operations made by the fastest BST for X
- A BST is c-competitive if it executes all sufficiently long sequences X in time at most $c \cdot OPT(X)$.
- Tango trees are $O(\log \log n)$-competitive as shown by Demain et al. in 2004 (paper is available on Moodle). Splay trees are conjectured to be $O(1)$-competitive, but **we do not know if this is true**.

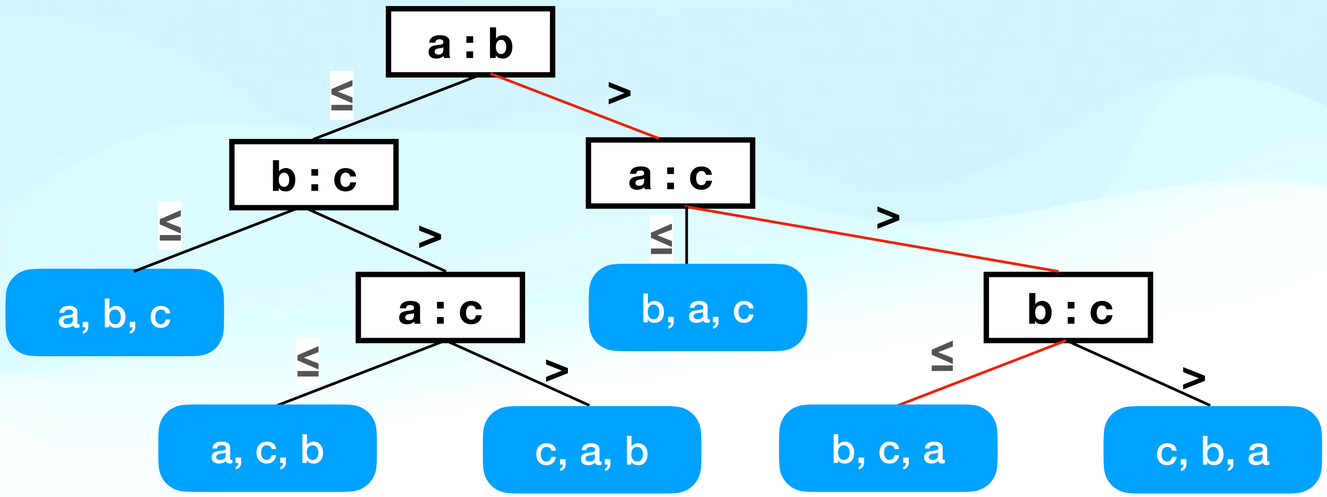
Lower bound for sorting

"Computational origami" by Erik Demaine and Martin Demaine (MoMA permanent collection)

Can we sort in o(n log n)?

- Spoiler: if only comparisons are allowed, then NO.
- Any comparison-based sorting program can be thought of as defining a decision tree of possible executions.
- Running the same program twice on the same permutation causes it to do exactly the same thing, but running it on different permutations of the same data causes a different sequence of comparisons to be made on each.

Decision tree



Example: a = 4, b = 2, c = 3

Claim: the minimum height of a decision tree is the worst-case complexity of comparison-based sorting.

Lower bound

Lemma: The height of any decision tree is $\Omega(n \log n)$.

Proof: Since any two different permutations of n elements require a different sequence of steps to sort, there must be at least n! different paths from the root to leaves in the decision tree.

Thus there must be at least n! different leaves in this binary tree.

Since a binary tree of height h has at most 2^h leaves, we know that $n! \leq 2^h$, or $h \geq \log(n!)$.

Finally, $\log(n!) = \Omega(n \log n)$.

Lower bound

Claim: The minimum height of a decision tree is the worst-case complexity of comparison-based sorting.

Lemma: The height of any decision tree is Ω(n log n).

Theorem: Any comparison-based sorting algorithm must use Ω(n log n) time.

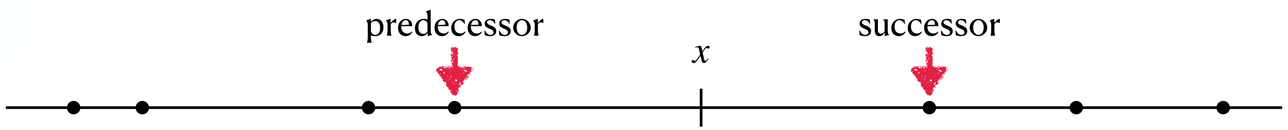
Predecessor problem

Source gallica.bnf.fr / Bibliothèque nationale de France

Predecessor problem

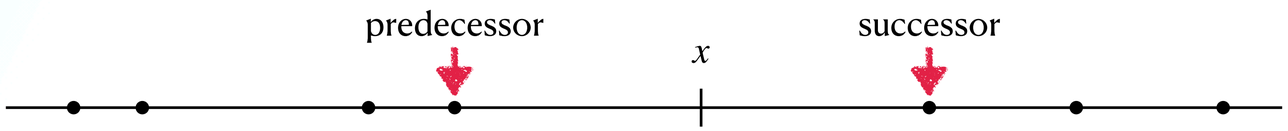
Maintain a set S of integers from a universe $U = [1,u]$ subject to:

- insertions
- deletions
- predecessor queries: given x, find the largest $y \in S$ s.t. $y \leq x$
- successor queries: given x, find the smallest $y \in S$ s.t. $y \geq x$



Predecessor problem

- Harder than dictionaries / hashing
- Binary search trees: $O(n)$ space and $\Theta(\log n)$ time (optimal in the comparison model)
- van Emde Boas trees: $O(u)$ space and $O(\log \log u)$ time, where u is the size of the universe

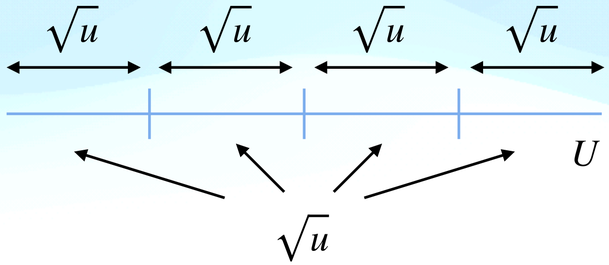


van Emde Boas trees

(reinterpreted by Bender and Farach-Colton)

If the time per operation satisfies

$$T(u) = T(\sqrt{u}) + O(1),$$

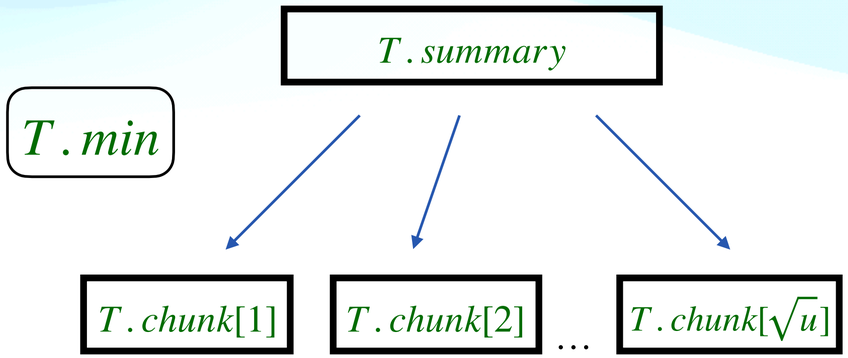


we're good: $T(u) = O(\log \log u)$ (substitution method)

Idea: split the universe U into $\sqrt{u}$ chunks of size $\sqrt{u}$ each, recurse for the chunks.

Recursive van Emde Boas tree T of size u

- $T.summary$ - vEB of size $\sqrt{u}$ containing all i such that the i-th chunk is not empty
- For each $1 \leq i \leq \sqrt{u}$, $T.chunk[i]$ - vEB of size $\sqrt{u}$ containing $x \% \sqrt{u}$ for each x in the i-th chunk
- $T.\min$ - the smallest element in T, **not** stored recursively
- $T.\max$ - the largest element in T



Hierarchical coordinates

Represent each integer $x = \langle c, i \rangle$, where

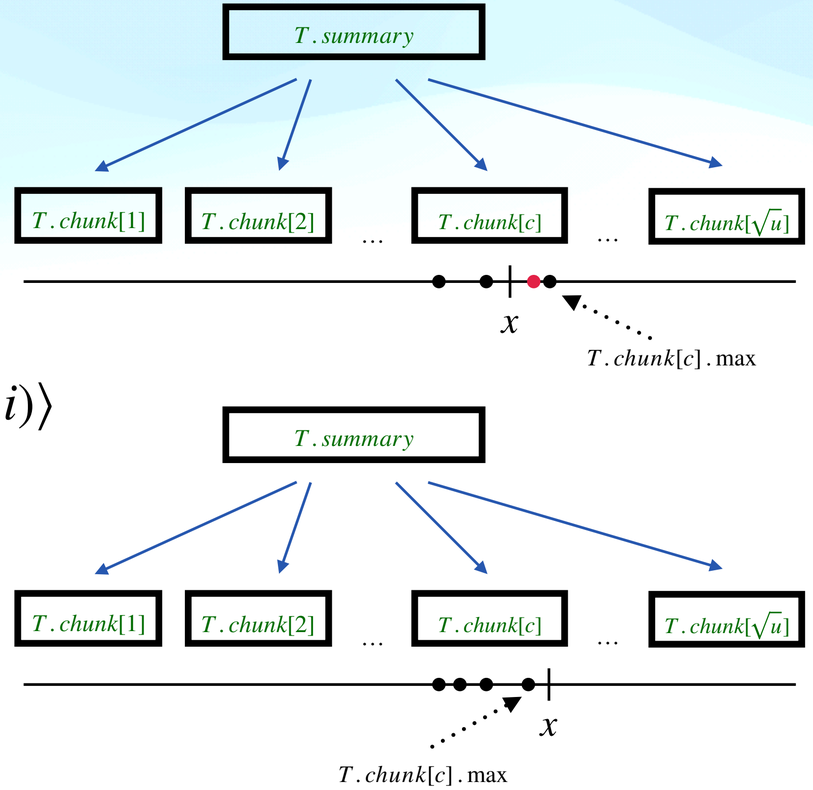
- c is the chunk containing x (that is, $c = x \bmod \sqrt{u}$)
- i is the position of x in the chunk (that is, $i = x \,\%\, \sqrt{u}$)

Successor

$T(u) = T(\sqrt{u}) + O(1)$

Successor($T, x = \langle c, i \rangle$)

1. **if** $x < T.\min$ **then return** $T.\min$
2. **if** $i < T.chunk[c].\max$:
3. **return** $\langle c, \textbf{Successor}(T.chunk[c], i) \rangle$
4. **else:**
5. $c' = \textbf{Successor}(T.summary, c)$
6. return $\langle c', T.chunk[c'].\min \rangle$

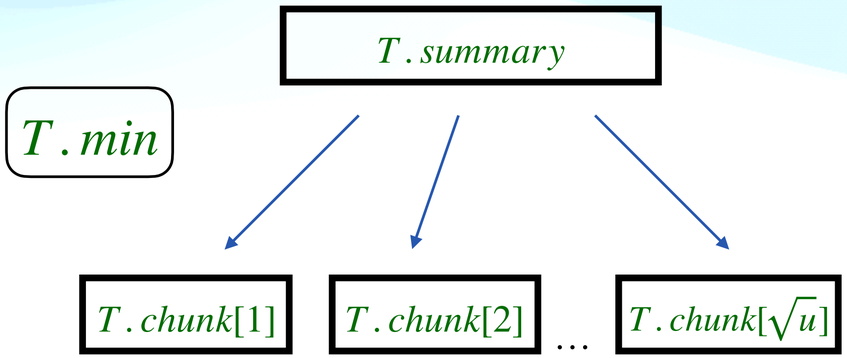


Insertion

$T(u) = T(\sqrt{u}) + O(1)$

Insert$(T, x = \langle c, i \rangle)$

1. **if** $T.\min = None$ **then**
2. $\quad T.\min = T.\max = x$
3. $\quad$ **return**
4. **if** $x < T.\min$ **then** swap$(x, T.\min)$
5. **if** $x > T.\max$ **then** $T.\max = x$
6. **if** $T.chunk[c].\min = None$ **then**
7. $\quad$ **Insert**$(T.summary, c)$ (next call is constant time: min not stored recursively)
8. **Insert**$(T.chunk[c], i)$

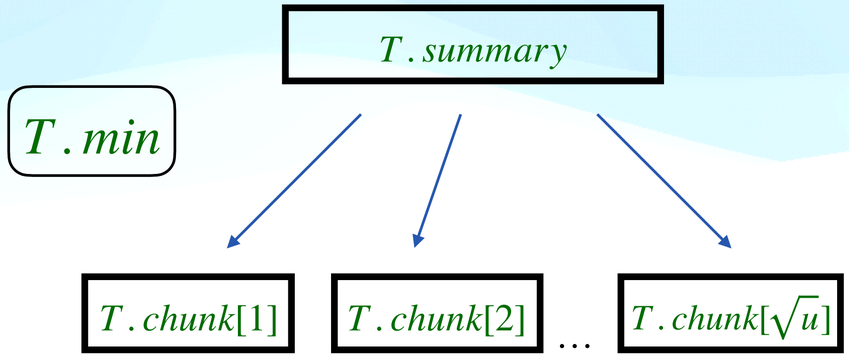


Deletion

$T(u) = T(\sqrt{u}) + O(1)$

Delete$(T, x = \langle c, i \rangle)$

1. **if** $x = T.\min$ **then**
2. $\quad c = T.summary.\min$
3. $\quad$ **if** $c = None$ **then** $T.\min = None$**; return**
4. $\quad x = T.\min = \langle c, i = T.chunk[c].\min \rangle$ (unstore new min)
5. **Delete**$(T.chunk[c], i)$
5. **if** $T.chunk[c].\min = None$ **then** (empty now)
6. $\quad$ **Delete**$(T.summary, c)$ (previous call takes constant time)
7. **if** $T.summary.\min = None$ **then** $T.\max = T.\min$
8. **else:**
9. $\quad c' = T.summary.\max$
10. $\quad T.max = \langle c', T.chunk[c'].\max \rangle$



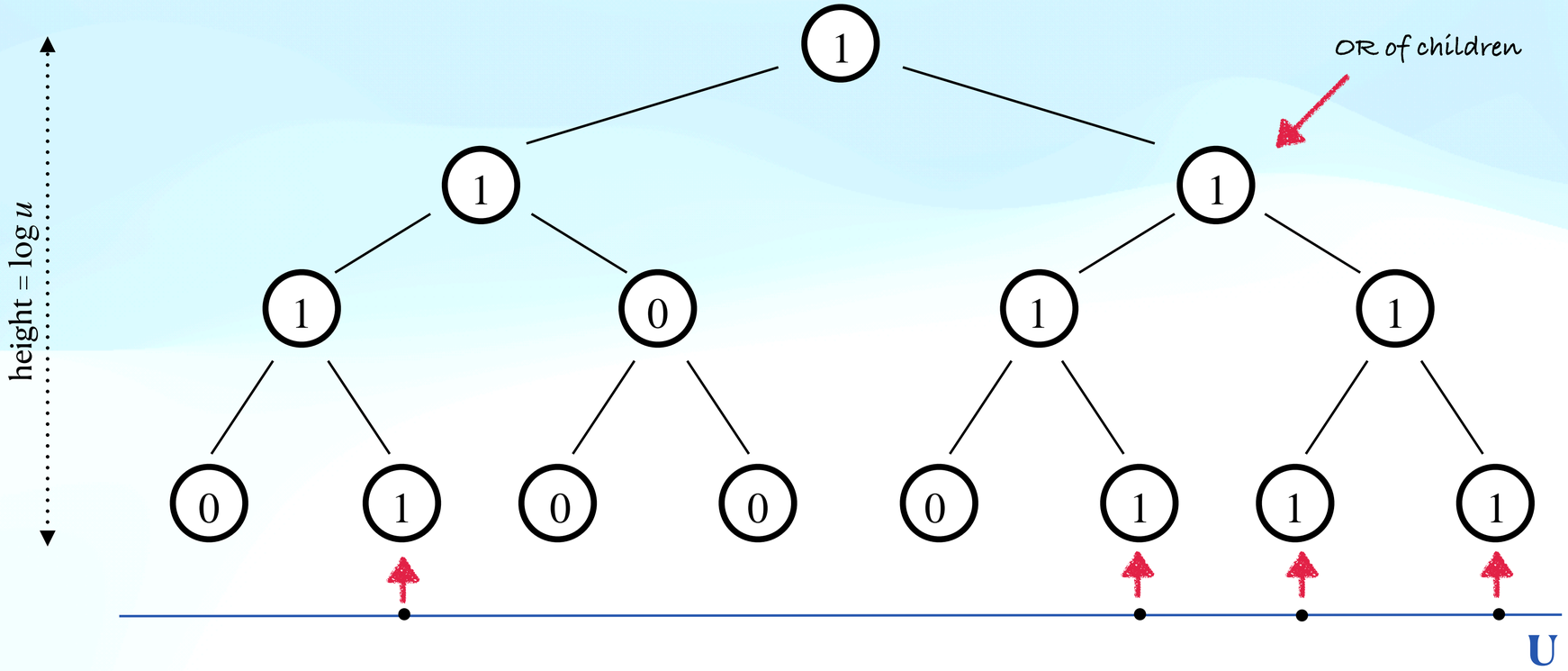
van Emde Boas trees

(reinterpreted by Bender and Farach-Colton)

- Time of successor / predecessor search, insertion, deletion satisfies $T(u) = T(\sqrt{u}) + O(1)$, and therefore $T(u) = O(\log \log u)$
- Space satisfies $S(u) = (\sqrt{u} + 1) \cdot S(\sqrt{u}) + O(1)$, and therefore $S(u) = O(u)$

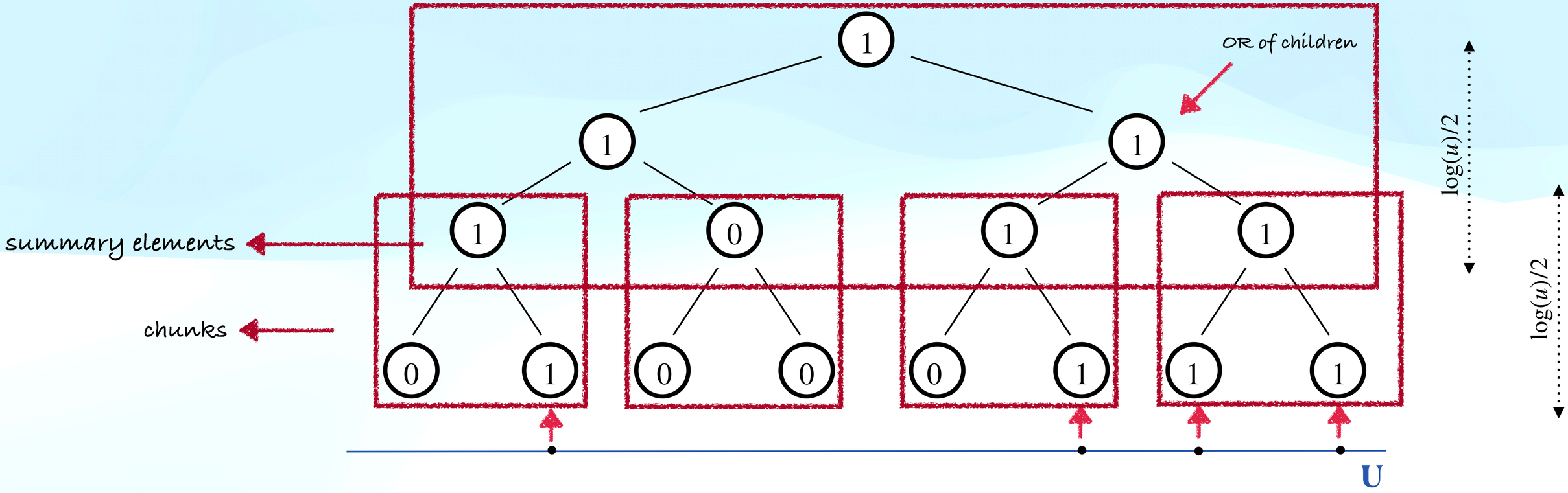
van Emde Boas trees

(van Emde Boas 1975)



van Emde Boas trees

(van Emde Boas 1975)

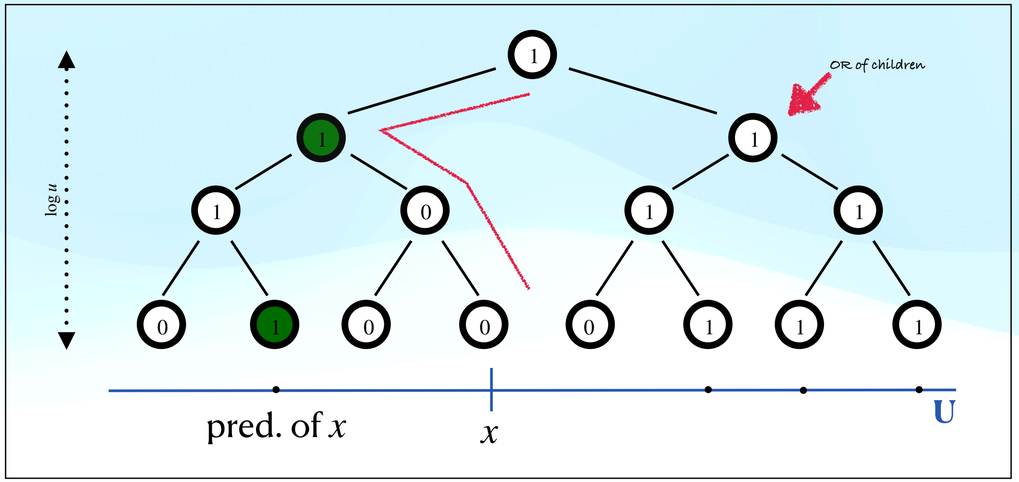


van Emde Boas trees

(van Emde Boas 1975)

Successor(x):

Path from the root to x is monotone. Binary search the path to find the lowest 1. It corresponds either to the predecessor, or to the successor of x.



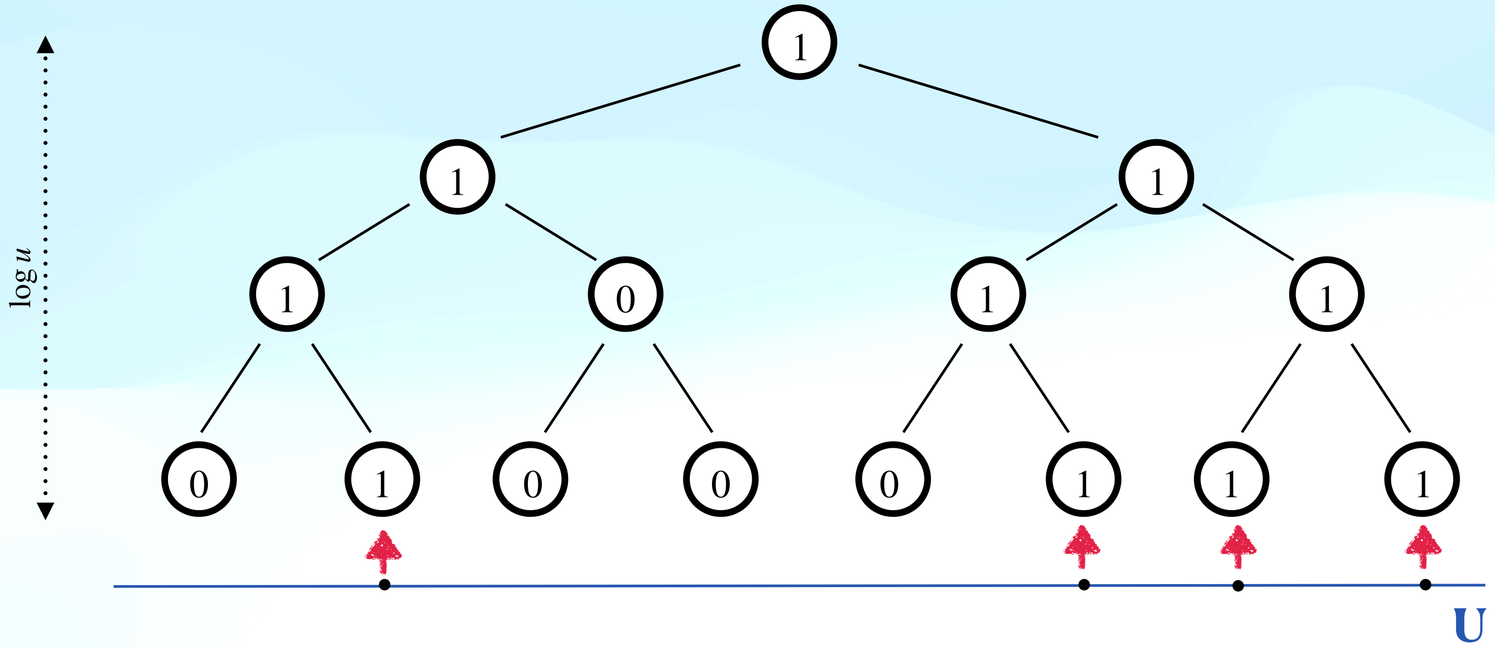
Assume that we store:

- all nodes in an array of size $O(u)$ to allow efficient binary search;
- a pointer from each node of the tree to the maximum and minimum elements in its subtree;
- all the elements as a doubly-linked list.

We can then find the successor and the predecessor of x in $O(\log \log u)$ time.

van Emde Boas trees

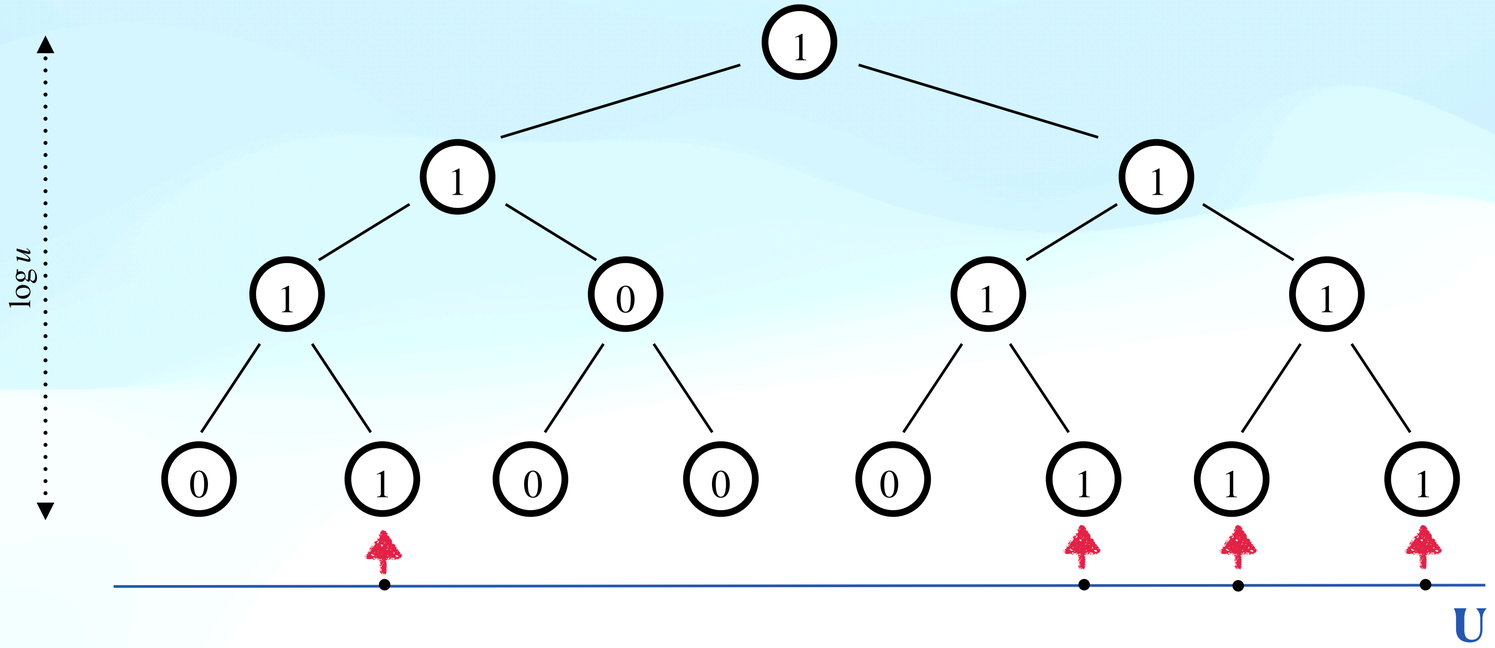
(van Emde Boas 1975)



- Updates require $O(\log u)$ time: we must update the element-to-root path.
- Space = $\Theta(u)$

van Emde Boas trees

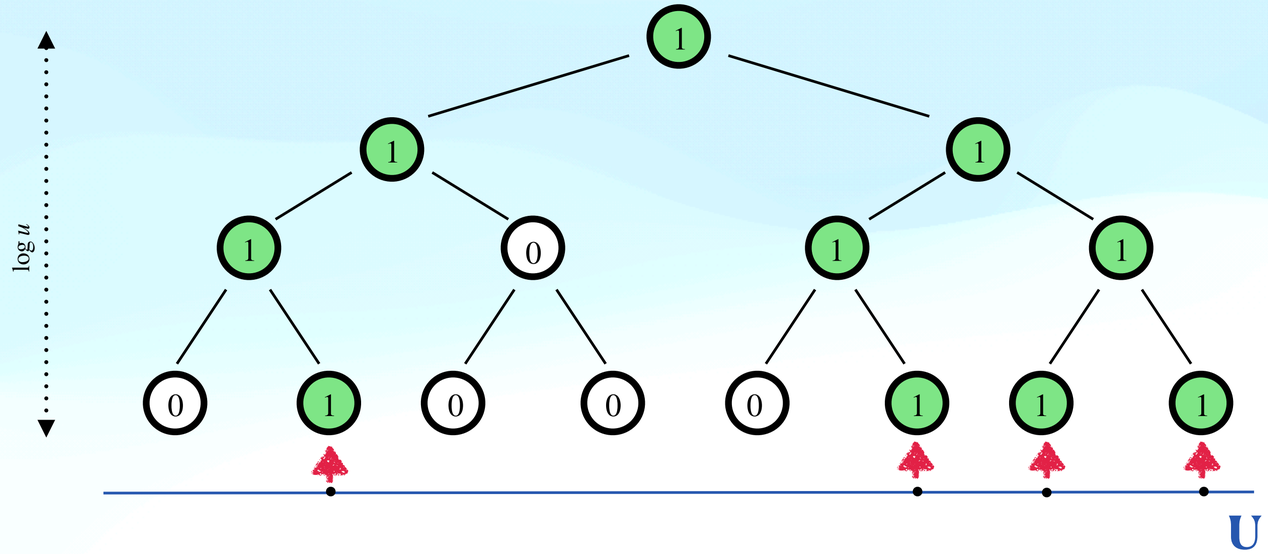
(van Emde Boas 1975)



- Updates require $O(\log u)$ time: we must update the element-to-root path.
- Space = $\Theta(u)$

x-fast trees

(Williard 1983)



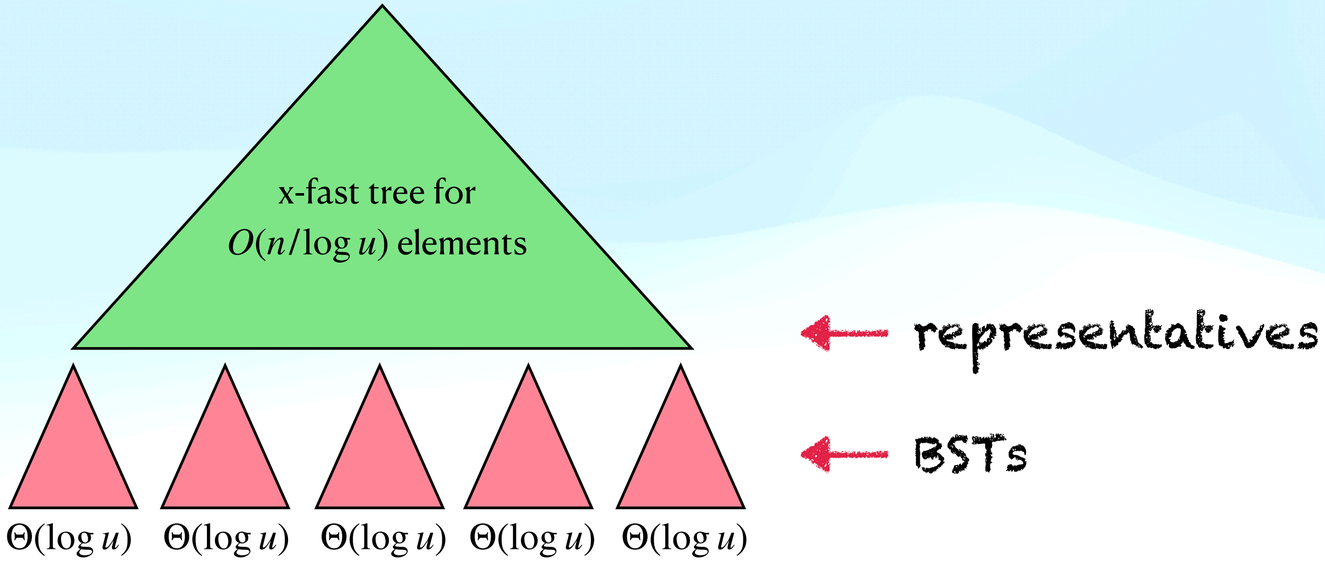
Store every root-to-green node path viewed in binary (left = 0, right = 1) via Cuckoo hashing

Predecessor queries in $O(\log \log u)$ time, updates in $O(\log u)$ expected amortised time

Space $O(n \log u)$ ❤️

y-fast trees

(Williard 1983)



Maintain elements in groups of size in $[\log(u)/4, 2\log(u)]$, for each group build a BST

Predecessor queries in $O(\log\log u)$ time, updates in $O(\log\log u)$ expected amortised time (insertion into the x-fast trie happens only once per $\Theta(\log u)$ new elements!) ❤️

Space $O(n)$ ❤️

y-fast trees

(Williard 1983)

How to maintain the elements of groups of size in $[\log(u)/2, 2\log u]$?

If there are fewer than $\log(u)/2$ elements, store them in a single BST (no x-fast tree)

Otherwise, suppose that we add/delete an element. If a group becomes too large, split it in two.

If a group becomes too small, merge it with its neighbour, then split if needed.

Summary

Today we saw:

- Binary search trees
- Lower bound for sorting
- Predecessor problem

Next time:

- Pattern matching
- Tries and dictionary look-up
- Aho-Corasick algorithm
- Suffix trees