Introduction to Algorithms: Lecture 4b

Xue Chen
xuechen1989@ustc.edu.cn
2025 spring in

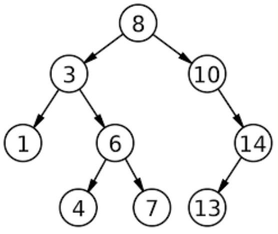
Outline

Introduction

Max-Heap does not support operations like FIND k-th largest in a set.

Why BST?

It supports almost all dynamic-set operations (including FIND) except MERGE — the most complicated data structure to implement in this course



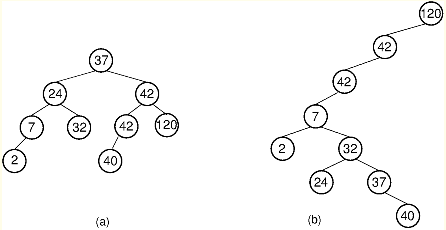
Example of BST

Overview

BST is a big class of data structures with many variations.

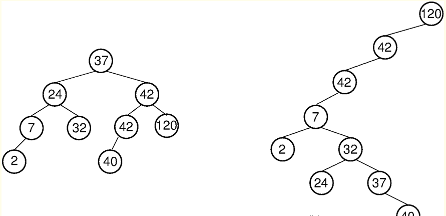
Basic Properties of BST

1. A binary tree
2. For each node v,
   (1) $v.key \geqslant u.key$ for any node u in the left sub-tree of v;
   (2) $v.key \leqslant u.key$ for any node u in the right sub-tree of v.



Example: BSTs

More about BST



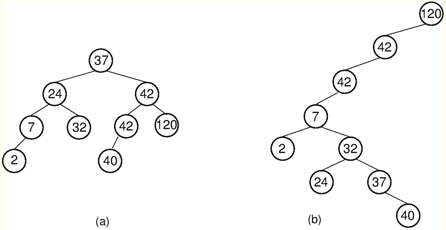
(a) (b)

Different from a heap:

(1) Far from a complete binary tree and the height could be $\Omega(\log n)$;

(2) Provides a total order among all nodes.

More about BST



Different from a heap:
(1) Far from a complete binary tree and the height could be $\Omega(\log n)$;
(2) Provides a total order among all nodes.

Plan

1. TREE-WALK and SEARCH operations on a given BST
2. Basic operations and its height
3. Red-Black tree guarantees the height is $O(\log n)$
4. Next time: FIND and COUNT operations

Tree Walk

From the BST property, output all the keys in sorted order:

Tree Walk

From the BST property, output all the keys in sorted order:

---

procedure INORDER-TREE-WALK(x)
  if $x \neq$ NIL **then**
    INORDER-TREE-WALK($x.left$)
    Print $x.key$
    INORDER-TREE-WALK($x.right$)

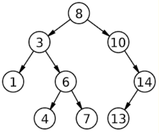
Notations: For a node x, $x.p :=$its parent node, $x.left :=$left child, $x.right :=$right child.

Tree Walk

From the BST property, output all the keys in sorted order:

procedure INORDER-TREE-WALK(x)
  if $x \neq$ NIL **then**
    INORDER-TREE-WALK($x.left$)
    Print $x.key$
    INORDER-TREE-WALK($x.right$)

Notations: For a node x, $x.p$:=its parent node, $x.left$:=left child, $x.right$:=right child.



INORDER-TREE-WALK(3), 8, INORDER-TREE-WALK(10)

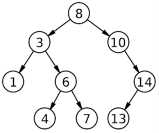
Tree Walk

From the BST property, output all the keys in sorted order:

---

procedure INORDER-TREE-WALK(x)
  if $x \neq$ NIL **then**
    INORDER-TREE-WALK($x.left$)
    Print $x.key$
    INORDER-TREE-WALK($x.right$)

Notations: For a node x, $x.p :=$its parent node, $x.left :=$left child, $x.right :=$right child.



INORDER-TREE-WALK(3), 8, INORDER-TREE-WALK(10)
$\Rightarrow$ INORDER-TREE-WALK(1), 3, INORDER-TREE-WALK(6), 8, 10, INORDER-TREE-WALK(14)

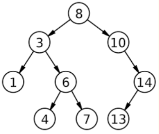
Tree Walk

From the BST property, output all the keys in sorted order:

---

procedure INORDER-TREE-WALK(x)
  if $x \neq$ NIL **then**
    INORDER-TREE-WALK($x.left$)
    Print $x.key$
    INORDER-TREE-WALK($x.right$)

Notations: For a node x, $x.p :=$its parent node, $x.left :=$left child, $x.right :=$right child.



INORDER-TREE-WALK(3), 8, INORDER-TREE-WALK(10)
$\Rightarrow$ INORDER-TREE-WALK(1), 3, INORDER-TREE-WALK(6), 8, 10, INORDER-TREE-WALK(14)
$\Rightarrow$ 1, 3, 4, 6, 7, 8, 10, 13, 14

Analysis

Running Time

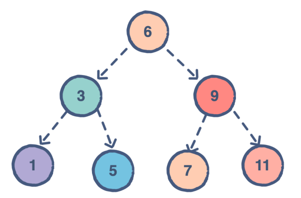
The running time is $O(n)$.

Analysis

Running Time

The running time is $O(n)$.

Preorder tree walk: Prints the root before the left subtree call.
Postorder tree walk: Prints the root after the right subtree call.



An example of a binary search tree

Preorder: 6 3 1 5 9 7 11
Postorder: 1 5 3 7 11 9 6

Operations

BST supports many operations in time $O(h)$ where h is its height.

TREE-SEARCH(k)

Find a node with key value k.

Operations

BST supports many operations in time $O(h)$ where h is its height.

TREE-SEARCH(k)

Find a node with key value k.

---

procedure TREE-SEARCH(k)
    $x = root$
    while $x \neq$ NIL and $x.key \neq k$ **do**
        if $k < x.key$ **then**
            $x = x.left$
        else
            $x = x.right$
    Return x

Operations

BST supports many operations in time $O(h)$ where h is its height.

TREE-SEARCH(k)

Find a node with key value k.

---

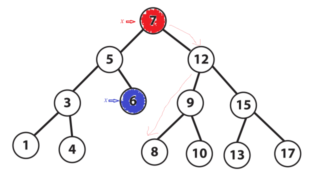
procedure TREE-SEARCH(k)
    $x = root$
    while $x \neq$ NIL and $x.key \neq k$ **do**
        if $k < x.key$ **then**
            $x = x.left$
        else
            $x = x.right$
    Return x

Question: How to implement TREE-MINIMUM and TREE-MAXIMUM?

TREE-SUCCESSOR

Find the element whose key value is next to $x.key$.

Two cases depend on whether $x.right$ is empty or not.



procedure TREE-SUCCESSOR(x)
  if $x.right \neq$ NIL **then**
    Return TREE-MINIMIUM($x.right$)
  else
    $y = x.p$
    while $y \neq$ NIL and $x = y.right$ **do**
      $x = y$ and $y = y.p$

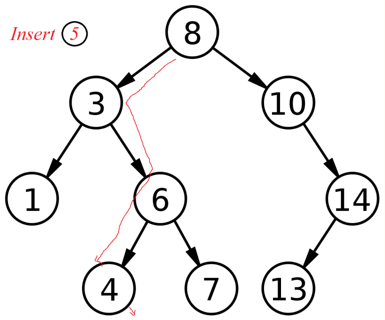
Theorem

Given a BST of height h, we can implement TREE-SEARCH, TREE-MINIMUM, TREE-MAXIMUM, TREE-SUCCESSOR and TREE-PREDECESSOR in time $O(h)$.

INSERTION

Insert a element with key k into BST:

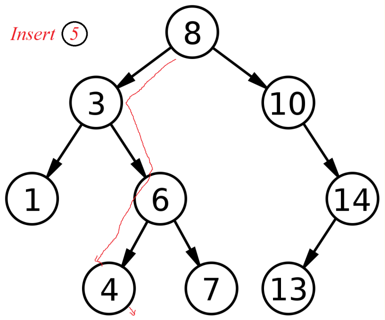
Find position like TREE-SEARCH



INSERTION

Insert a element with key k into BST:

Find position like TREE-SEARCH

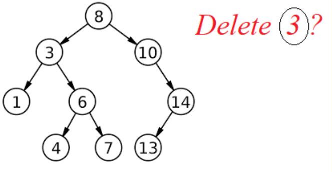



```
TREE-INSERT(T, z)
  y = NIL
  x = T.root
  while x ≠ NIL
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == NIL
      T.root = z          // tree T was empty
  elseif z.key < y.key
      y.left = z
  else y.right = z
```

y denote its "potential" parent node

TREE-DELETE

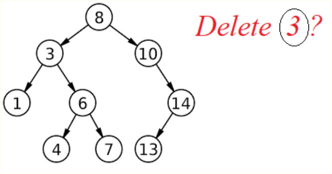
Only consider how to delete a given node z say $key = 3$:



Three cases:

TREE-DELETE

Only consider how to delete a given node z say $key = 3$:

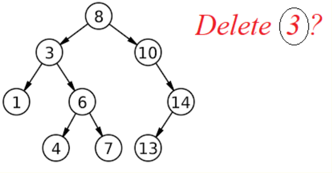


Three cases:

1. z has no children: Remove z directly.

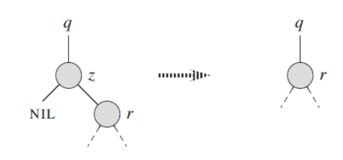
TREE-DELETE

Only consider how to delete a given node z say $key = 3$:



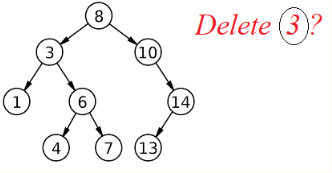
Three cases:

1. z has no children: Remove z directly.
2. z has just one child: Elevate that child to take z's position.



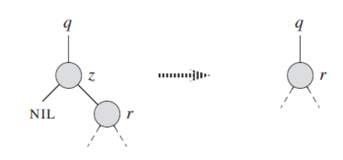
TREE-DELETE

Only consider how to delete a given node z say $key = 3$:



Three cases:

1. z has no children: Remove z directly.
2. z has just one child: Elevate that child to take z's position.



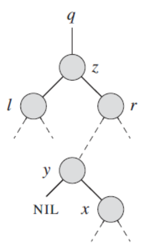
3. z has two children: The most complicated case — basic idea is to replace z by its successor y.

TREE-DELETE

The most complicated case — z has two children and its successor is y; and our plan is to replace z by y such that no change in z's left subtree

Key Observation

$y.left$ must be NIL.



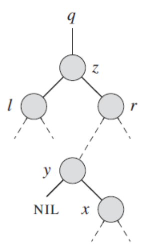
TREE-DELETE

The most complicated case — z has two children and its successor is y; and our plan is to replace z by y such that no change in z's left subtree

Key Observation

$y.left$ must be NIL.

1. Step 1: Exchange y with z's right child r. Notice $y.left = $ NIL in this process



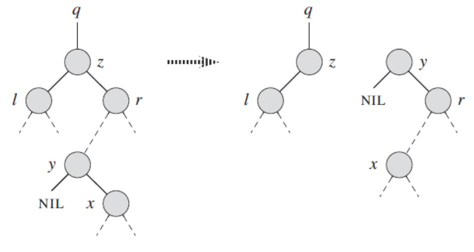
TREE-DELETE

The most complicated case — z has two children and its successor is y; and our plan is to replace z by y such that no change in z's left subtree

Key Observation

$y.left$ must be NIL.

1. Step 1: Exchange y with z's right child r. Notice $y.left = \text{NIL}$ in this process



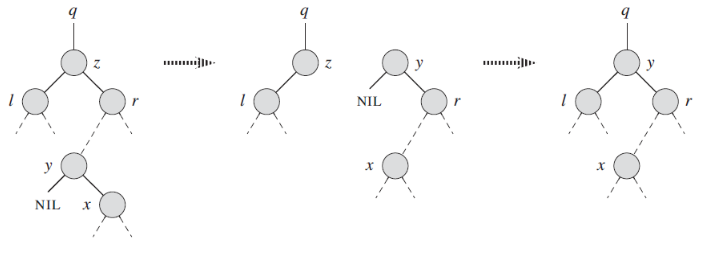
TREE-DELETE

The most complicated case — z has two children and its successor is y; and our plan is to replace z by y such that no change in z's left subtree

Key Observation

$y.left$ must be NIL.

1. Step 1: Exchange y with z's right child r. Notice $y.left = \text{NIL}$ in this process
2. Step 2: Replace z by y.



Height

Control the height is not easy especially after lots of TREE-DELETE operations.

Height

Control the height is not easy especially after lots of TREE-DELETE operations.

Theorem 12.4 in CLRS

The expected height h of a randomly built BST (by inserting n elements in a random order) on n distinct keys is $O(\log n)$.

Summary

Compare to heaps:

1. The big-O constant of heaps is small
2. BST supports more operations in time $O(h)$

Summary

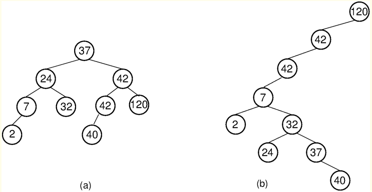
Compare to heaps:

1. The big-O constant of heaps is small
2. BST supports more operations in time $O(h)$
3. BST does not support MERGE or UNION unlike binomial/Fibonacci heaps
4. The height of BST could be fairly large

Summary

Compare to heaps:

1. The big-O constant of heaps is small
2. BST supports more operations in time $O(h)$
3. BST does not support MERGE or UNION unlike binomial/Fibonacci heaps
4. The height of BST could be fairly large



Example: The height could be large

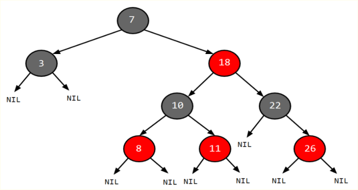
Extensions

Red-Black tree, AVL tree, Splay tree guarantees the height (or amortized height) is $O(\log n)$.

Outline

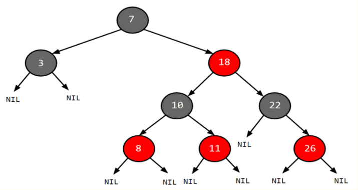
Introduction

Red-black trees: One extension guarantee the height is always $O(\log n)$.



Introduction

Red-black trees: One extension guarantee the height is always $O(\log n)$.

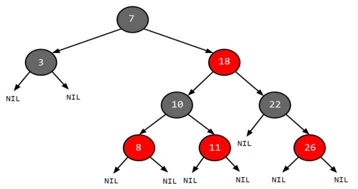


Basic Properties

1. Every node is either red or black.
2. The root is black.

Introduction

Red-black trees: One extension guarantee the height is always $O(\log n)$.



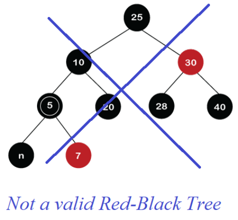
Basic Properties

1. Every node is either red or black.
2. The root is black.
3. Each leaf NIL is black.
4. The two children of a red node must be black.

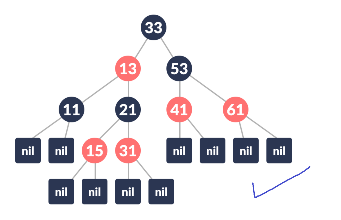
Overview

Most important property

All simple paths from any node to its descendant leaves have the same number of black nodes. (*)



Bad Red-Black Tree

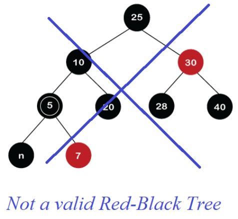


Good Red-Black Tree

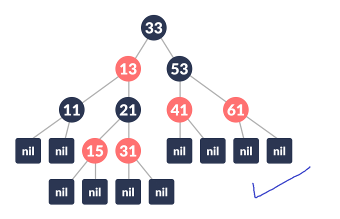
Overview

Most important property

All simple paths from any node to its descendant leaves have the same number of black nodes. (*)

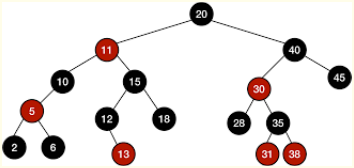


Bad Red-Black Tree



Good Red-Black Tree

Question: Is this good or bad?



Bound the Height

Lemma 13.1 in CLRS

Given Property (*) and Property (4) — the children of red nodes are black, a red-black tree with n nodes has height $\leqslant 2\log_2(n+1)$.

Bound the Height

Lemma 13.1 in CLRS

Given Property (*) and Property (4) — the children of red nodes are black, a red-black tree with n nodes has height $\leqslant 2\log_2(n+1)$.

Notation

$bh(x)$: The black height of a node x — well defined because of Property (*)

OBS: The subtree of node x has at least $2^{bh(x)} - 1$ internal nodes.

Bound the Height

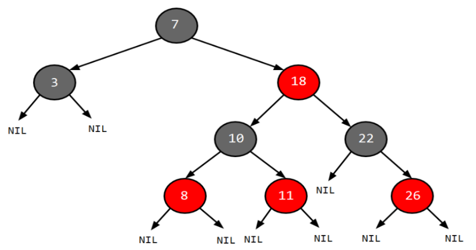
Lemma 13.1 in CLRS

Given Property (*) and Property (4) — the children of red nodes are black, a red-black tree with n nodes has height $\leqslant 2\log_2(n+1)$.

Notation

$bh(x)$: The black height of a node x — well defined because of Property (*)

OBS: The subtree of node x has at least $2^{bh(x)} - 1$ internal nodes.



Finish the proof: The height of x is at most $2 \cdot bh(x)$ by property (4).

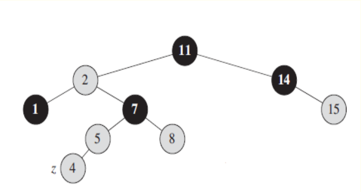
Maintenance

1. Same for SEARCH, SUCCESSOR, TREE-WALKS.
2. How to maintain those properties during INSERT and DELETE?
   (4) the children of red node are black
   (*) All paths have the same number of black nodes.

Maintenance

1. Same for SEARCH, SUCCESSOR, TREE-WALKS.
2. How to maintain those properties during INSERT and DELETE?
   (4) the children of red node are black
   (*) All paths have the same number of black nodes.

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```



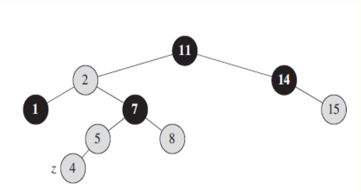
Example: INSERT z with key 4

Warm-up: First try BST-INSERT

Maintenance

1. Same for SEARCH, SUCCESSOR, TREE-WALKS.
2. How to maintain those properties during INSERT and DELETE?
   (4) the children of red node are black
   (*) All paths have the same number of black nodes.

```
TREE-INSERT(T, z)
    y = NIL
    x = T.root
    while x ≠ NIL
        y = x
        if z.key < x.key
            x = x.left
        else x = x.right
    z.p = y
    if y == NIL
        T.root = z        // tree T was empty
    elseif z.key < y.key
        y.left = z
    else y.right = z
```



Example: INSERT z with key 4

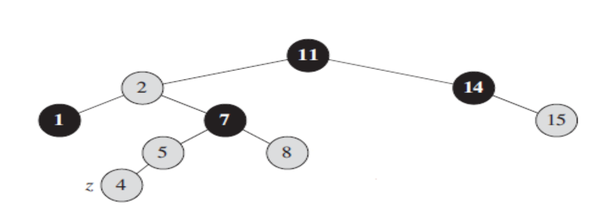
Warm-up: First try BST-INSERT

Set-up z

$z.left = z.right =$ NIL, what about $z.color$?
Next: basic idea of INSERTION but leave the details to Experiment 2

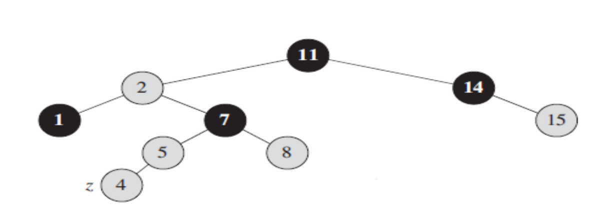
Basic idea of FIXUP

1. Set $z.color = $ BLACK will break Property (*) for all other paths — this would need more in FIXUP
2. Hope setting $z.color = $ RED will simplify FIXUP



Basic idea of FIXUP

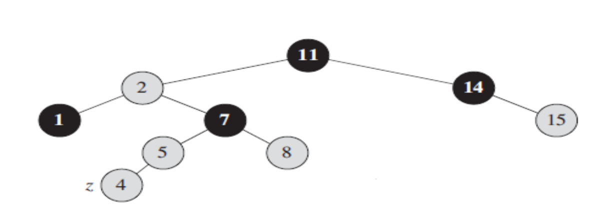
1. Set $z.color$ = BLACK will break Property (*) for all other paths — this would need more in FIXUP
2. Hope setting $z.color$ = RED will simplify FIXUP




```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

Basic idea of FIXUP

1. Set $z.color = $ BLACK will break Property (*) for all other paths — this would need more in FIXUP
2. Hope setting $z.color = $ RED will simplify FIXUP



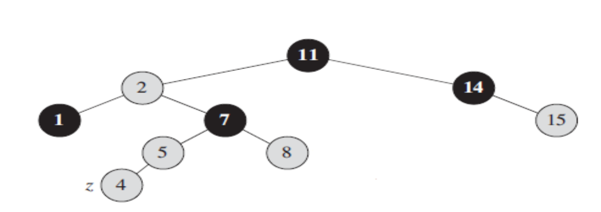

```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
      y = x
      if z.key < x.key
            x = x.left
      else x = x.right
z.p = y
if y == T.nil
      T.root = z
elseif z.key < y.key
      y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

Question

What if $z.p.color = $ BLACK?

Basic idea of FIXUP

1. Set $z.color$ = BLACK will break Property (*) for all other paths — this would need more in FIXUP
2. Hope setting $z.color$ = RED will simplify FIXUP



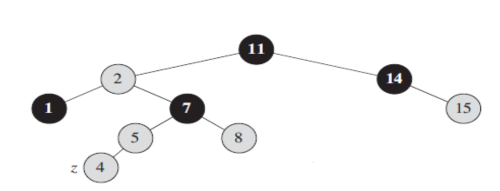

```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

Question

What if $z.p.color$ = BLACK? Done ☺
Only consider the case $z.p.color$ = RED.

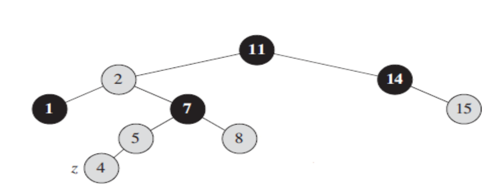
Discussion about FIXUP

Both *z.color* and *z.p.color* are RED — violate Property (4) about children of red nodes



Discussion about FIXUP

Both $z.color$ and $z.p.color$ are RED — violate Property (4) about children of red nodes

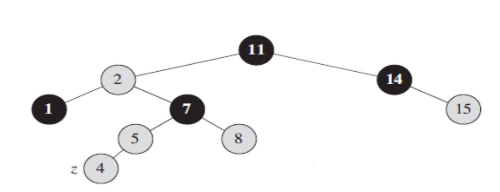


Solution

Must reset $z.p.color = $ BLACK — otherwise sticking on $z.p.color = $ RED returns to the point of setting $z.color = $ BLACK.

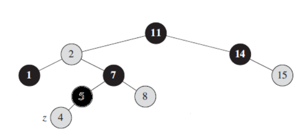
Discussion about FIXUP

Both $z.color$ and $z.p.color$ are RED — violate Property (4) about children of red nodes



Solution

Must reset $z.p.color = $ BLACK — otherwise sticking on $z.p.color = $ RED returns to the point of setting $z.color = $ BLACK.

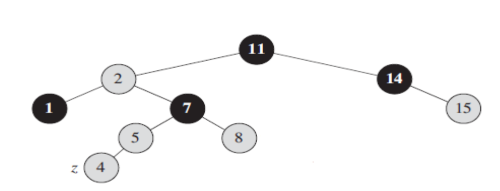


What if reset z.p.color=Black

$z.p.p$ must be BLACK

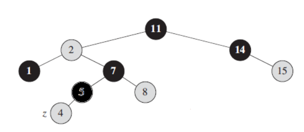
Discussion about FIXUP

Both $z.color$ and $z.p.color$ are RED — violate Property (4) about children of red nodes



Solution

Must reset $z.p.color = $ BLACK — otherwise sticking on $z.p.color = $ RED returns to the point of setting $z.color = $ BLACK.

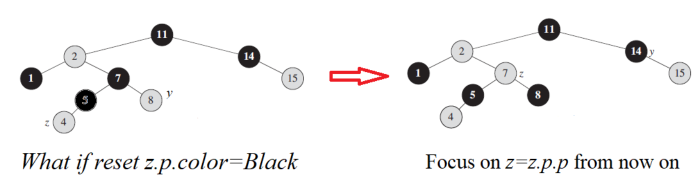


What if reset z.p.color=Black

$z.p.p$ must be BLACK because $z.p.color$ was RED before resetting
$\Rightarrow$ Question: Are we done or not?

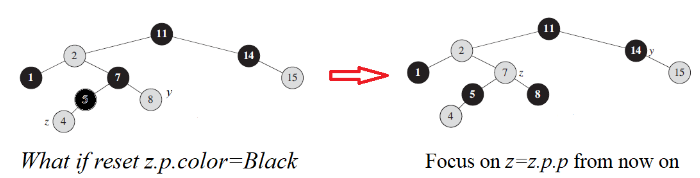
More about FIXUP

This violates Property (*) — to fix it, change the colors of $z.p.p$ and its children

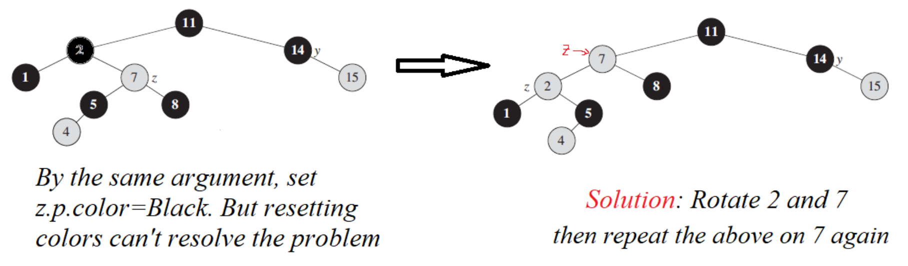


More about FIXUP

This violates Property (*) — to fix it, change the colors of $z.p.p$ and its children



2 and 7 are not valid — repeat the above argument on 7



Rotations

During INSERT(z) with $z.color = red$

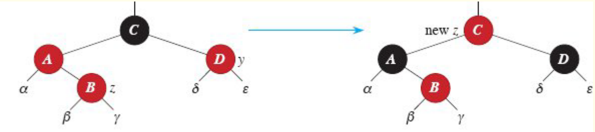
Violation comes from $z.p.color = red$ (and $z.p.p.color = black$)

Rotations

During INSERT(z) with $z.color = red$

Violation comes from $z.p.color = red$ (and $z.p.p.color = black$)

Easy Case: the other child of $z.p.p$, y, is red

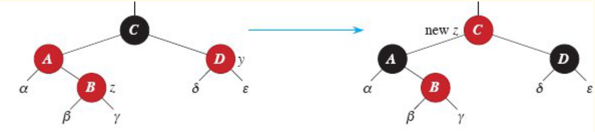


Rotations

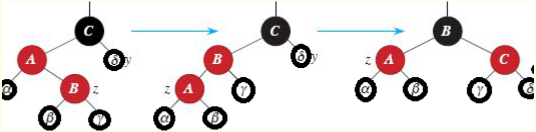
During INSERT(z) with $z.color = red$

Violation comes from $z.p.color = red$ (and $z.p.p.color = black$)

Easy Case: the other child of $z.p.p$, y, is red



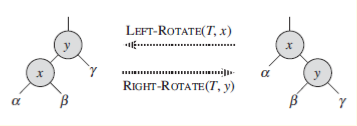
Involved case: the other child of $z.p.p$, y, is black — rotate them and recolor $z.p$ and $z.p.p$.



Example: Right rotation of B

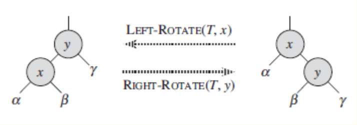
Formally

Left rotation of a left-child x: replace the position of $x.parent$ by x and reset $x.parent$ as x's right child.



Formally

Left rotation of a left-child x: replace the position of $x.parent$ by x and reset $x.parent$ as x's right child.

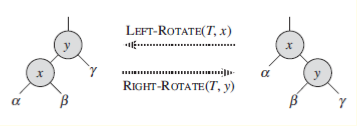


LEFT-ROTATE(T, x)

```
y = x.right                      // set y
x.right = y.left                 // turn y's left subtree into x's
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                        // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                       // put x on y's left
x.p = y
```

Formally

Left rotation of a left-child *x*: replace the position of *x.parent* by *x* and reset *x.parent* as *x*'s right child.

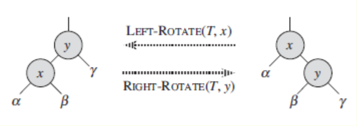



```
LEFT-ROTATE(T, x)
  y = x.right              // set y
  x.right = y.left         // turn y's left subtree into x's
  if y.left ≠ T.nil
      y.left.p = x
  y.p = x.p                // link x's parent to y
  if x.p == T.nil
      T.root = y
  elseif x == x.p.left
      x.p.left = y
  else x.p.right = y
  y.left = x               // put x on y's left
  x.p = y
```

Similarly, right rotation changes the position of a right-child.

Formally

Left rotation of a left-child x: replace the position of $x.parent$ by x and reset $x.parent$ as x's right child.




```
LEFT-ROTATE(T, x)
y = x.right                  // set y
x.right = y.left             // turn y's left subtree into x's
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                    // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                   // put x on y's left
x.p = y
```

Similarly, right rotation changes the position of a right-child.

For the full discussion, see Chapter 13.2 and 13.3 in [CLRS].

Outline

B-tree

An extension of BST: basically each node could have more children and more keys.

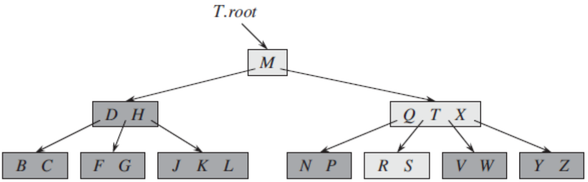


Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $x.n$ keys has $x.n+1$ children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R.

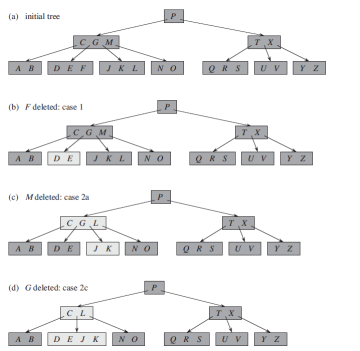
Motivation: Modify BST such that it fits better with memory hierarchy of cache.

Formally

1. If all nodes have degree in $[t, 2t]$ (equivalently, $[t-1, 2t-1]$ keys in one node), height $\leqslant \log_t \frac{n+1}{2}$
2. SEARCH: Similar to BST.

Formally

1. If all nodes have degree in $[t, 2t]$ (equivalently, $[t-1, 2t-1]$ keys in one node), height $\leqslant \log_t \frac{n+1}{2}$
2. SEARCH: Similar to BST.
3. INSERT: When there are $2t-1$ keys, split it into two and insert one more into its parent.
4. DELETE: More complicated — see Chapter 18 in CLRS!



Outline

Introduction

Some practical applications need to augment textbook data structures by introducing more parameters or combining two data structures.

Example

Given a BST (or a red-black tree),

1. find the ith largest element;
2. find the number of elements in $[key_1, key_2]$.

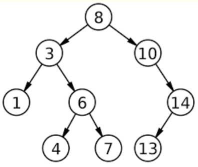
Introduction

Some practical applications need to augment textbook data structures by introducing more parameters or combining two data structures.

Example

Given a BST (or a red-black tree),

1. find the ith largest element;
2. find the number of elements in $[key_1, key_2]$.



Find the 3rd largest element? How many elements in $[5, 9]$?

Augmenting

Basic Idea

For each node x, maintain attribute $x.size$ such that

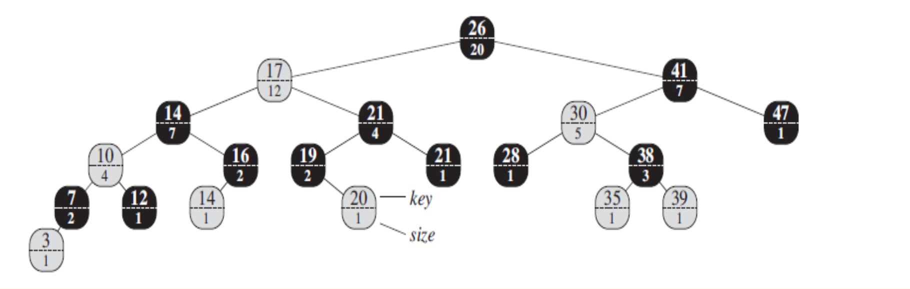
$$x.size = x.left.size + 1 + x.right.size$$

Augmenting

Basic Idea

For each node x, maintain attribute $x.size$ such that

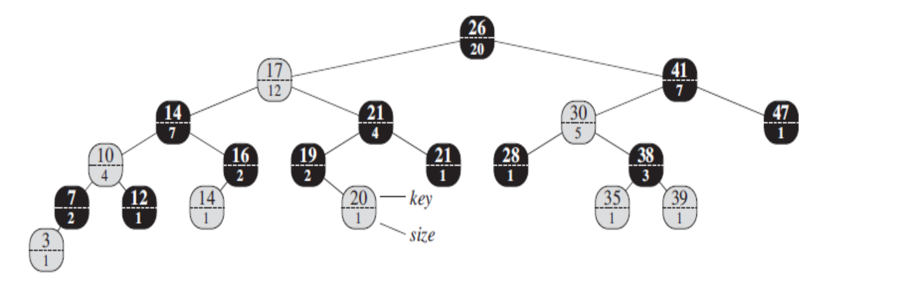
$$x.size = x.left.size + 1 + x.right.size$$



Introduce $x.size$ to each node in a red-black tree

OBS

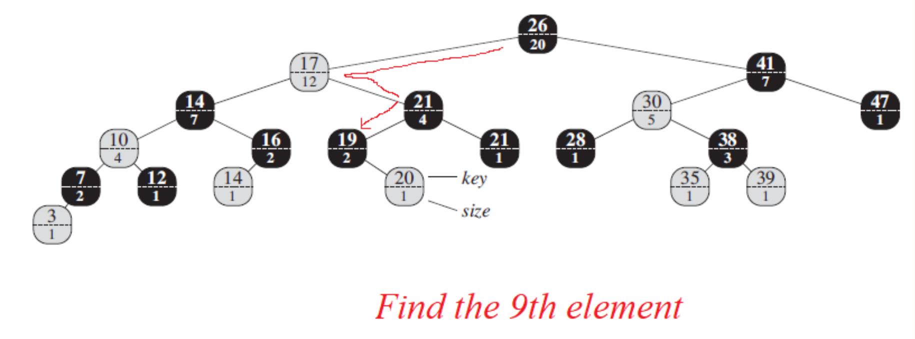
Since a BST is ordered, the rank of x in its subtree is $x.left.size + 1$.



SELECT

Task

Given an number $i \leqslant n$, return the node with the ith smallest element in the tree.



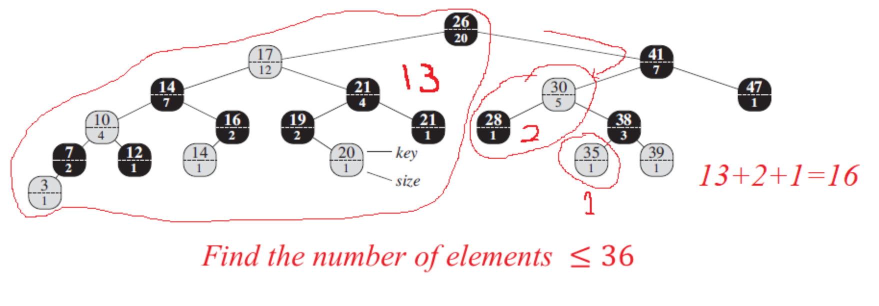
SELECT

procedure SELECT(x, i)
  $r = x.left.size + 1$
  if $i = r$ **then**
    Return x
  else if $i < r$ **then**
    Return SELECT($x.left$, i)
  else
    Return SELECT($x.right$, $i - r$)
procedure MAIN(i)
  Return SELECT($root$, i)

COUNT

Task

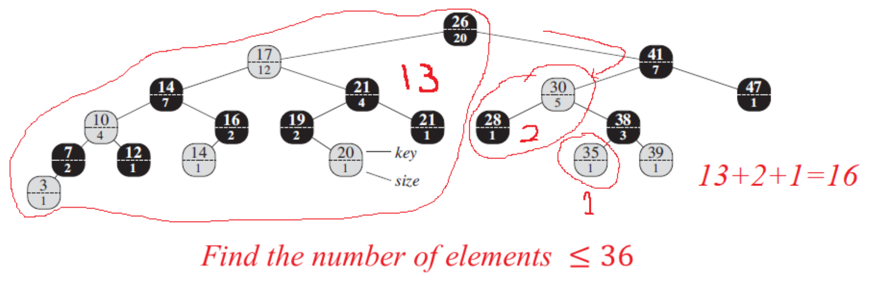
Given a key value k, return the number of nodes in the tree with a key value at most k.



COUNT

Task

Given a key value k, return the number of nodes in the tree with a key value at most k.



OBS:

1. If $k \geqslant x.key$, every node in subtree $x.left$ has a value $\leqslant k$.
2. Otherwise, every node in subtree $x.right$ has a value $> k$.

COUNT

procedure COUNT(x, k)
  if $k \leqslant x.key$ **then**
    Return $x.left.size + 1 + \text{COUNT}(x.right, k)$
  else
    Return COUNT($x.left, k$)
procedure MAIN(k)
  Return $\text{COUNT}(root, k)$

COUNT

procedure COUNT(x, k)
  if $k \leqslant x.key$ **then**
    Return $x.left.size + 1 + \text{COUNT}(x.right, k)$
  else
    Return COUNT($x.left, k$)
procedure MAIN(k)
  Return $\text{COUNT}(root, k)$

Extensions

To count the number of elements in $[k_1, k_2]$, apply it twice: $\text{COUNT}(k_2) - \text{COUNT}(k_1 - \epsilon)$.

More Augmenting

Question

Suppose there are n students with two keys: score and ID.
Design a data structure to support DELETE, INSERT, COUNT, QUERYSCORE(ID).

More Augmenting

Question

Suppose there are n students with two keys: score and ID.
Design a data structure to support DELETE, INSERT, COUNT, QUERYSCORE(ID).

Consider QUERYSCORE(ID): Hash supports it in time $O(1)$ while a red-black tree supports it in time $O(\log n)$.

More Augmenting

Question

Suppose there are n students with two keys: score and ID.
Design a data structure to support DELETE, INSERT, COUNT, QUERYSCORE(ID).

Consider QUERYSCORE(ID): Hash supports it in time $O(1)$ while a red-black tree supports it in time $O(\log n)$.

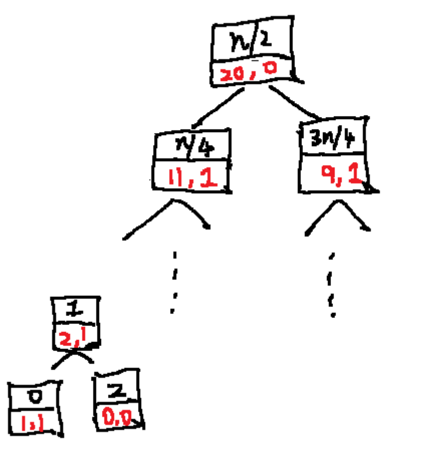
Augmenting

Combine them to inherit the advantages of both sides:

1. QUERYSCORE(ID) in time $O(1)$.
2. DELETE, INSERT, COUNT in time $O(\log n)$

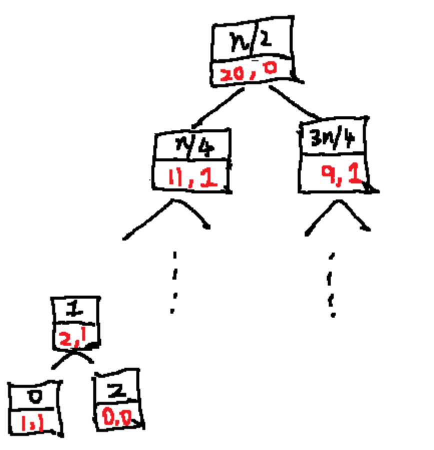
Static BST

If we know the range of key values are small, say $\{0, 1, \ldots, n\}$, one could consider its static version.



Static BST

If we know the range of key values are small, say $\{0, 1, \ldots, n\}$, one could consider its static version.



Similar to RADIXSORT, it is more efficient for a fixed range.

Summary

Operation	Hash	Heap(Fibonacci& Binomial)	Red-Black Tree
SEARCH	$O(1)$	$O(n)$	$O(\log n)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$
SELECT	$O(n)$	$O(n)$	$O(\log n)$
COUNT	$O(n)$	$O(n)$	$O(\log n)$
SUCCESSOR	$O(n)$	$O(n)$	$O(\log n)$
MIN	$O(n)$	$O(1)$	$O(\log n$
UNION	$O(n)$	$O(1)$	$O(n)$

Discussion

Like heap and BST, many data structures are based on binary trees.

1. Binary tree provides the most fundamental structure to reduce the time to $O(\log n)$
2. Choose/modify/combine proper data structures depends on environment, requirements, ...
3. Discuss data structures for disjoint sets (Chapter 21 in graph algorithms)

Questions?