Red Black Trees Q&A

Instructors: Subrahmanyam Kalyanasundaram
Karteek Sreenivasaiah

14 September 2020

Abstract Data Type

Set

maintains a set of elements from the universe

A set has the following functions:

- Insert(x) – Insert x into the set.
- Search(x) – Return True if x is an element of the set.
- Succ(x) – Returns the smallest value larger than x.
- Pred(x) – Returns the largest value smaller than x.
- GetMax() – Returns the largest value in the set.
- GetMin() – Returns the smallest value in the set.
- IsEmpty() – Returns True if and only if the set is empty.
- Delete(x) – Remove x from the set.

Data Structures for Set

Many choices of data structure to implement set:

- Array
- Sorted Array
- Heap
- Binary Search Tree
- Balanced Binary Search Trees

Data Structures for Set

Many choices of data structure to implement set:

- Array
- Sorted Array
- Heap
- Binary Search Tree
- Balanced Binary Search Trees

We now study a balanced Binary Search Tree called Red-Black Trees.

Red-Black Trees

RBTs have the following properties:

1. All nodes are colored either Red or Black.
2. The root node and the leaf nodes (NIL) are black.
3. Both children of a red node are black.
   No double red.
4. For any node x, all paths from x to the descendant leaves have the same number of black nodes. = Black height(x)

Red-Black Trees

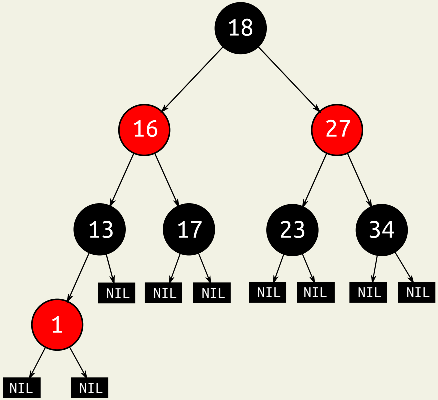
RBTs have the following properties:

1. All nodes are colored either Red or Black.
2. The root node and the leaf nodes (NIL) are black.
3. Both children of a red node are black.
   No double red.
4. For any node x, all paths from x to the descendant leaves have the same number of black nodes. = Black height(x)

Black height of a red black tree is the black height of its root.

Example

1. All nodes are colored either Red or Black.
2. The root node and the leaf nodes (NIL) are black.
3. Both children of a red node are black.
   No double red.
4. For any node x, all paths from x to the descendant leaves have the same number of black nodes. = Black height(x)



A Red-Black Tree supports all procedures of a BST:

- INSERT(*val*) – Inserts *val* into the RBT rooted at *node*.
- SEARCH(*val*) – Returns True of *val* exists in the BST rooted at *node*. False otherwise.
- SUCC(*val*) – Returns the smallest element greater than *val* in the RBT.
- PRED(*val*) – Returns the largest element lesser than *val* in the RBT.
- DELETE(*val*) – Deletes *val* from the RBT.

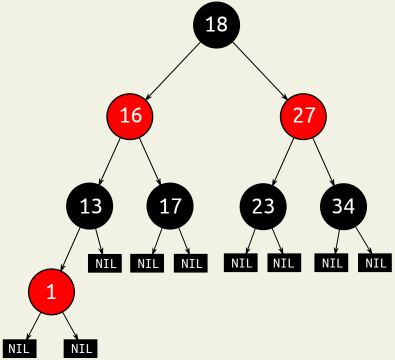
A Red-Black Tree supports all procedures of a BST:

- INSERT(*val*) – Inserts *val* into the RBT rooted at *node*.
- SEARCH(*val*) – Returns True of *val* exists in the BST rooted at *node*. False otherwise.
- SUCC(*val*) – Returns the smallest element greater than *val* in the RBT.
- PRED(*val*) – Returns the largest element lesser than *val* in the RBT.
- DELETE(*val*) – Deletes *val* from the RBT.

The procedures in green are implemented exactly like in a BST.

Black-Height

The black-height of a node X is the number of black colored nodes encountered on a path starting from X to any leaf (excluding X itself).



The black-height of the node with value 13 is 1.
The black-height of the root node is 2.
The black height of an red-black tree is the black height of its root.

Observations

Claim

A red-black tree with black-height β has height at most 2β.

Claim

A red-black tree with black-height β has height at most 2β.

Proof sketch:

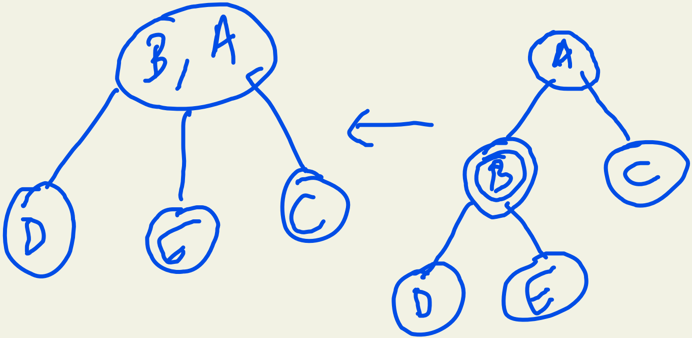
- Try to construct the longest possible path with at most β many black nodes.
- Property 4 will force you to color every alternate node black.

Observations

Theorem

If a red-black tree with n *internal* nodes and black height β, then

$$2^{\beta} \leq n + 1 \leq 4^{\beta}.$$



Observations

Theorem

If a red-black tree with *n internal* nodes and black height β, then

$$2^{\beta} \leq n + 1 \leq 4^{\beta}.$$

Proof sketch

- Merge each red node with its parent.
- Now each node has 1, 2 or 3 values with 2, 3 or 4 children.
  This is a 2-3-4 tree!

Observations

Theorem

If a red-black tree with *n internal* nodes and black height β, then

$$2^{\beta} \leq n + 1 \leq 4^{\beta}.$$

Proof sketch

- Merge each red node with its parent.
- Now each node has 1, 2 or 3 values with 2, 3 or 4 children.
  This is a 2-3-4 tree!
- The above 2-3-4 tree has height β.
- Thus $2^{\beta} - 1 \leq n \leq 4^{\beta} - 1$.

Observations

Theorem

A red-black tree with n *internal* nodes has height at most $2\log(n+1)$.

Observations

Theorem

A red-black tree with n *internal* nodes has height at most $2\log(n+1)$.

Proof

- We have seen that $2^{\beta} \leq n+1 \leq 4^{\beta}$.
- That is, $1/2\log(n+1) \leq \beta \leq \log(n+1)$.

Observations

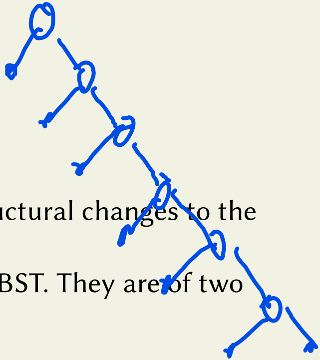
Theorem

A red-black tree with n *internal* nodes has height at most $2\log(n+1)$.

Proof

- We have seen that $2^\beta \leq n+1 \leq 4^\beta$.
- That is, $1/2\log(n+1) \leq \beta \leq \log(n+1)$.
- Use previous claim that height is at most twice the black-height to conclude the Theorem.

Balancing a BST



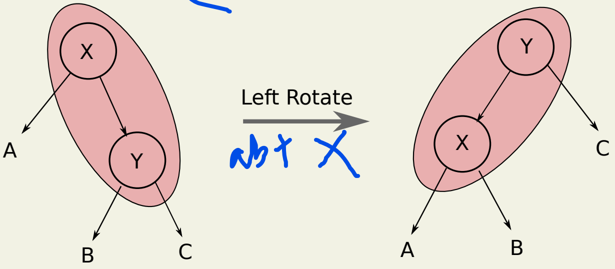
Balancing a BST is done by making structural changes to the underlying tree.

Rotations are operations on nodes of a BST. They are of two variants:

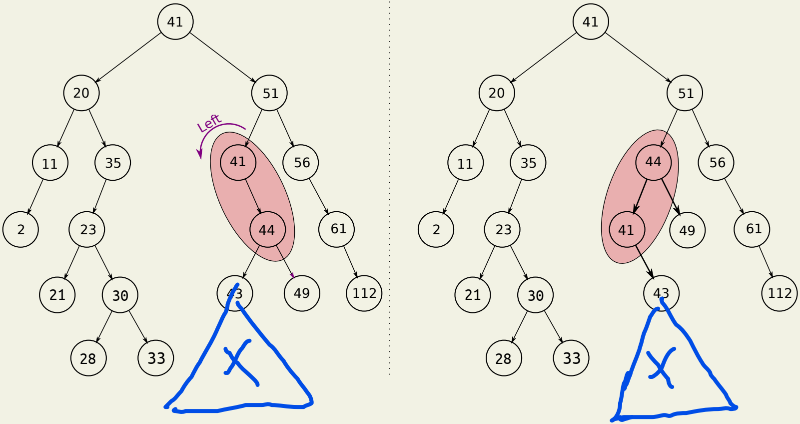
1. Left Rotate.
2. Right Rotate.

Left Rotate

Right rotate abt Y



Example


Left
41
20
51
11
35
41
56
2
23
44
61
21
30
43
49
112
28
33
X
41
20
51
11
35
44
56
2
23
41
49
61
21
30
43
112
28
33
X

INSERT procedure

INSERT(x) – Insert value x into the red-back tree.
High level strategy:

- Create a node X with value x and color red.
- Insert node X just like inserting into a Binary Search Tree.
- Call procedure FIXINSERT at node X.

FixInsert procedure

Which properties might be broken when we insert a new red node?

1. All nodes are colored either Red or Black.
2. The root node is black.
3. The leaf nodes (NIL) are black.
4. Both children of a red node are black.
5. For any node, all paths from the node to the descendant leaves have the same number of black nodes.

FixInsert procedure

Only properties 2 and 4 could be broken after inserting a red node:

1. All nodes are colored either Red or Black.
2. The root node is black.
3. The leaf nodes (NIL) are black.
4. Both children of a red node are black.
5. For any node, all paths from the node to the descendant leaves have the same number of black nodes.

FixInsert procedure

Only properties 2 and 4 could be broken after inserting a red node:

1. All nodes are colored either Red or Black.
2. The root node is black.
3. The leaf nodes (NIL) are black.
4. Both children of a red node are black.
5. For any node, all paths from the node to the descendant leaves have the same number of black nodes.

FixInsert - Fixing property 2

Property 2

The root node is black.

Some invariants when FixInsert is called on a node Z:

- Z is colored Red.
- If Property 2 is violated, then node Z itself is the root.

FixInsert - Fixing property 2

Property 2

The root node is black.

Some invariants when FixInsert is called on a node Z:

- Z is colored Red.
- If Property 2 is violated, then node Z itself is the root.

Resolution: Simply color Z black.

FixInsert - Fixing Property 4

Property 4

A red node has black children.

FixInsert - Fixing Property 4

Property 4

A red node has black children.

Some invariants when FixInsert is called on a node Z:

- Z is colored Red.
- If Property 4 is violated, it is violated only by the node Z and its parent.

FixInsert - Fixing Property 4

Property 4

A red node has black children.

Some invariants when FixInsert is called on a node Z:

- Z is colored Red.
- If Property 4 is violated, it is violated only by the node Z and its parent.

There are three cases when FixInsert is called on a node Z.

Cases

- Case 1: Uncle of Z is Red.

Cases

- Case 1: Uncle of Z is Red.
- Case 2: Uncle of Z is black and Z is a right child of a left child.

Cases

- Case 1: Uncle of Z is Red.
- Case 2: Uncle of Z is black and Z is a right child of a left child.
- Case 3: Uncle of Z is black and Z is a left child of a left child.

Cases

- Case 1: Uncle of Z is Red.
- Case 2: Uncle of Z is black and Z is a right child of a left child.
- Case 3: Uncle of Z is black and Z is a left child of a left child.
- Other cases follow by symmetry.

Case 1

Case 1

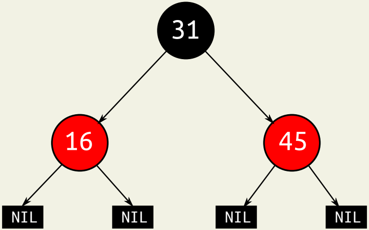
Uncle of Z is Red.

Resolution:

- Recolor parent, uncle and grandparent.
- Call FixInsert(grandparent)

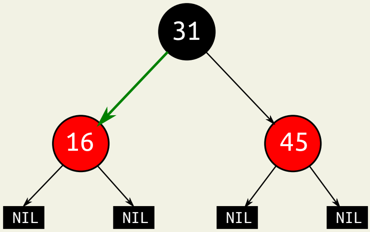
Example 1

Insert 9 to the following RBT:



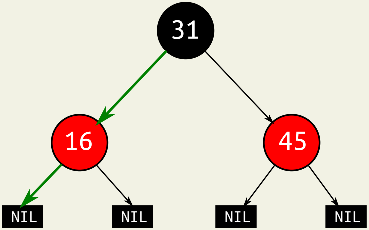
Example 1

Find the position where 9 should be inserted



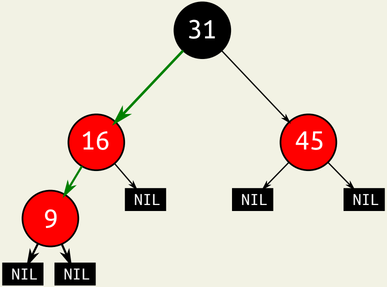
Example 1

Find the position where 9 should be inserted



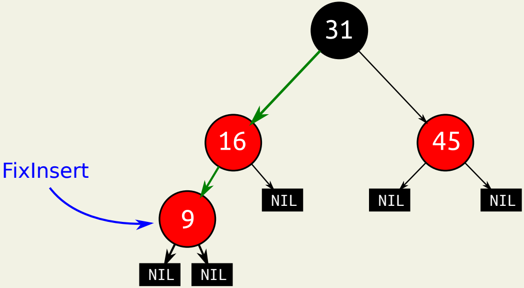
Example 1

Insert 9 as a new node with color red



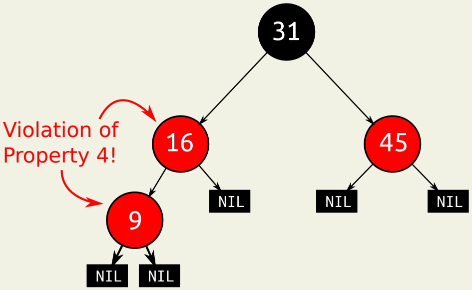
Example 1

Call FixInsert at the inserted location.



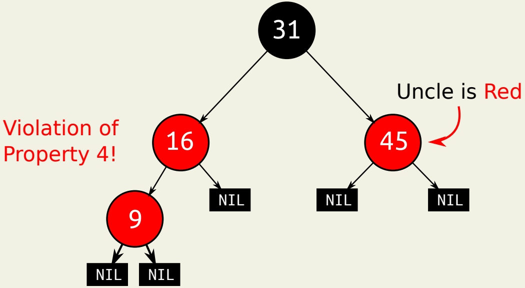
Example 1

Property 4 is violated. Check color of the uncle to determine case.



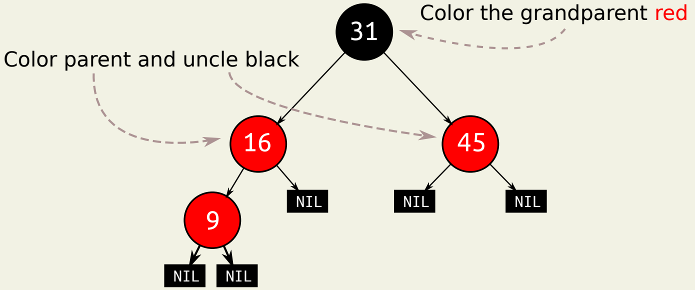
Example 1

We are in Case 1.



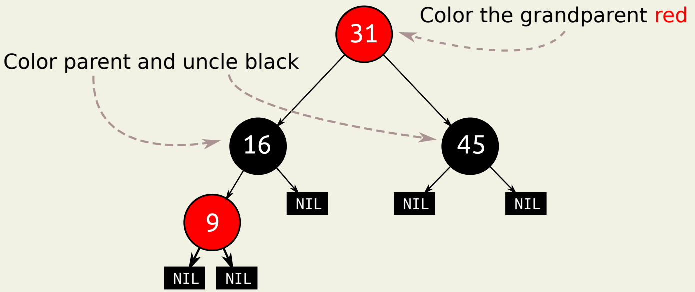
Example 1

Case 1 is resolved by recoloring.



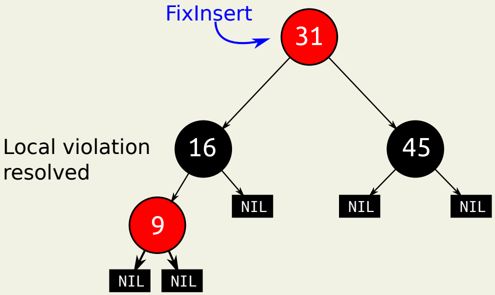
Example 1

Case 1 is resolved by recoloring.



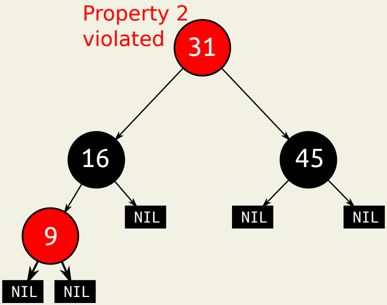
Example 1

Now call FixInsert on the grandparent.



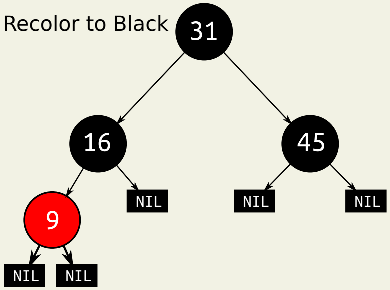
Example 1

Root node is not black.



Example 1

Simply recolor root to black.



Case 3

Case 3

Uncle of Z is Black and Z is left child of a left child.

Resolution:

- Recolor parent and grandparent.
- Rotate right at grandparent.

Case 3

Case 3

Uncle of Z is Black and Z is left child of a left child.

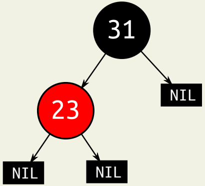
Resolution:

- Recolor parent and grandparent.
- Rotate right at grandparent.

Symmetric case: Z is right child of a right child.

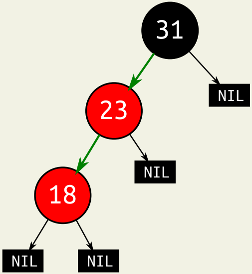
Example 2

Want to insert 18 into this RBT.



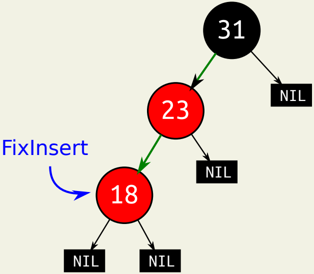
Example 2

Insert 18 as a red node according to BST property.



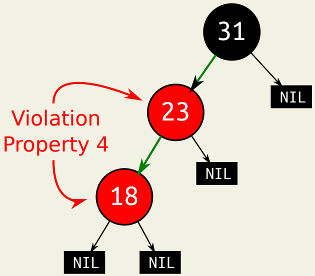
Example 2

Call FixInsert on the new node.



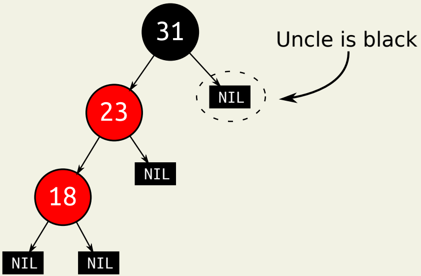
Example 2

FixInsert has to fix property 4.



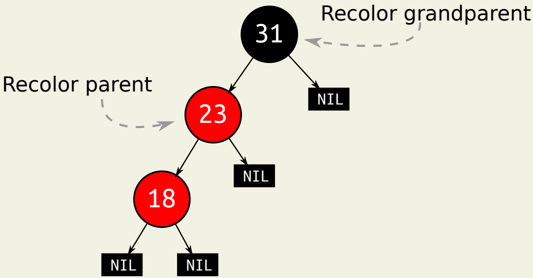
Example 2

Determine the case by looking at uncle.



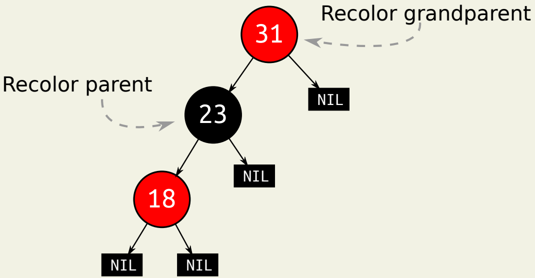
Example 2

Node 18 and its parent 23 are both left children. This is case 3.



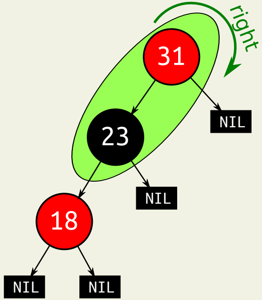
Example 2

Recolor parent to black and grandparent to red.



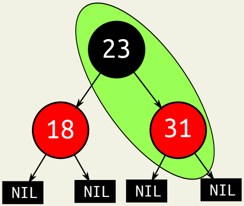
Example 2

Rotate right at grandparent.



Example 2

The resulting tree has no violations.



Case 2

Case 2

Uncle of Z is Black and Z is right child of a left child.

Resolution:

- Assign parent to Z.
- Rotate left at Z.
- Call FixInsert at Z.

The above procedure results in case 3.

Case 2

Uncle of Z is Black and Z is right child of a left child.

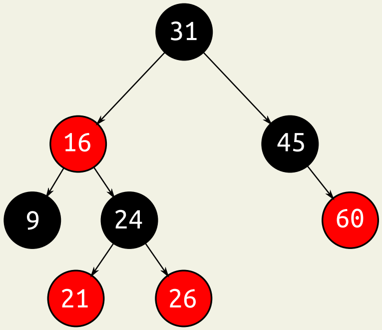
Resolution:

- Assign parent to Z.
- Rotate left at Z.
- Call FixInsert at Z.

The above procedure results in case 3.
Symmetric case: Z is left child of a right child.

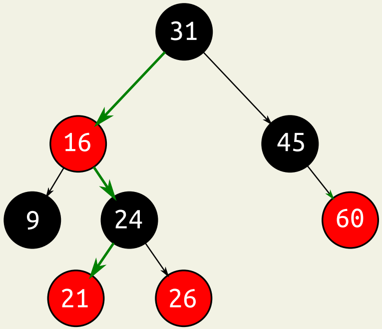
Example 3

Want to insert 18 into the following:



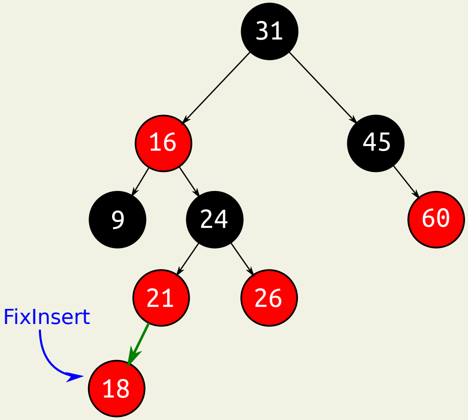
Example 3

Find the position for 18:



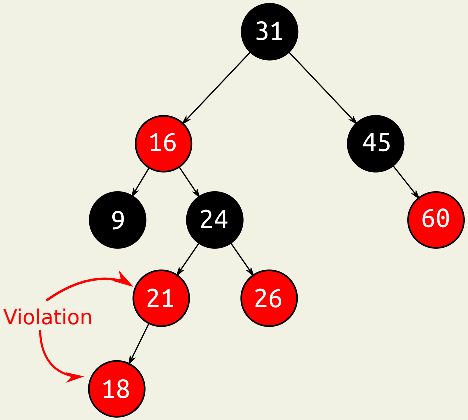
Example 3

Insert a new node with value 18 and color red. Call FixInsert at the inserted location.



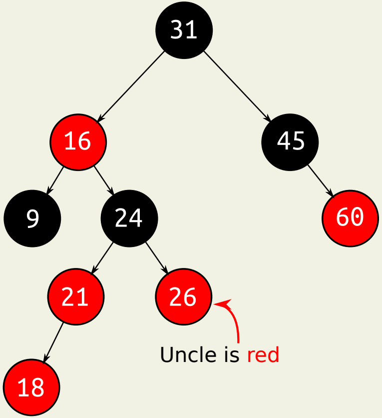
Example 3

Property 4 is violated. Check color of uncle to determine the case.



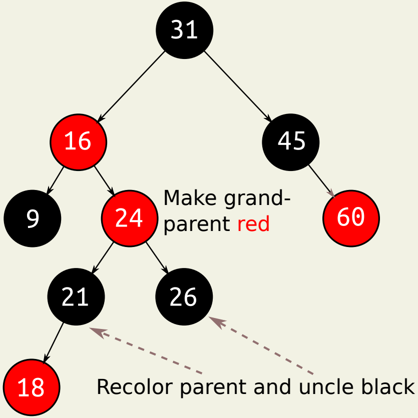
Example 3

Uncle is red. So we are in case 1.



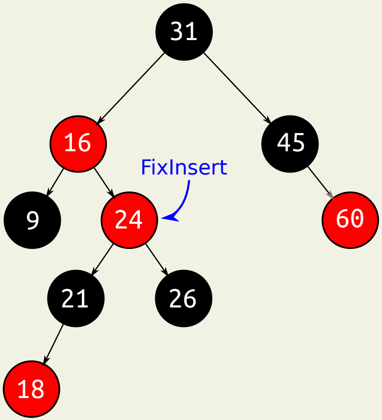
Example 3

Recolor as done earlier.



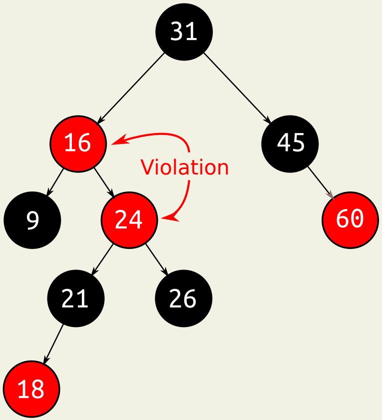
Example 3

Call FixInsert on grandparent.



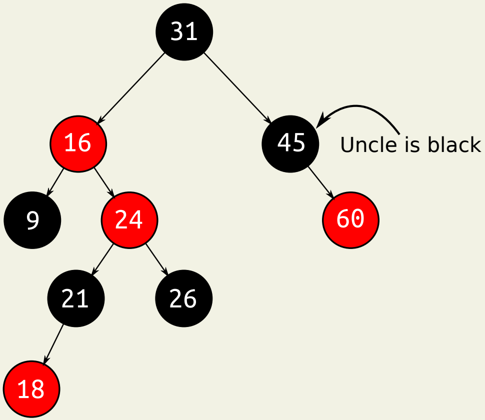
Example 3

Property 4 does not hold. Check color of Uncle to determine case.



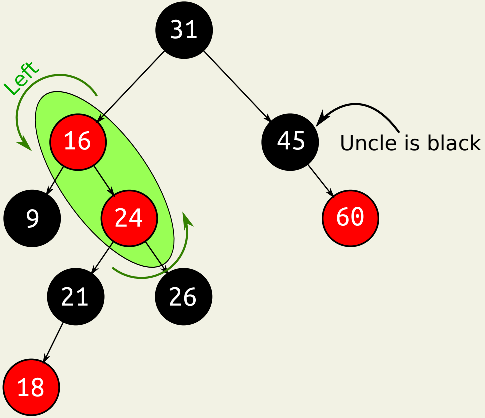
Example 3

Uncle is black and 24 is right child of a left child. So we are in case 2.



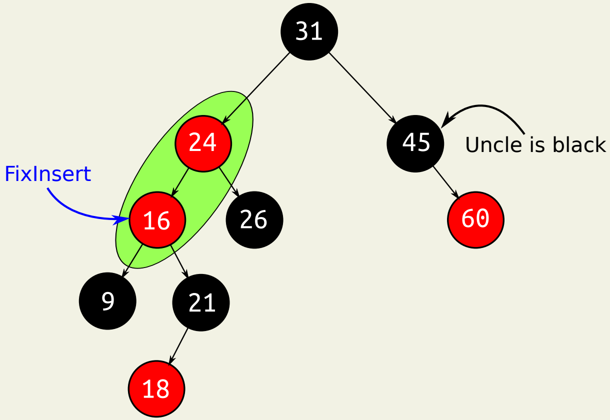
Example 3

Set Z as node with 16. Rotate left at Z and call FixInsert on Z.



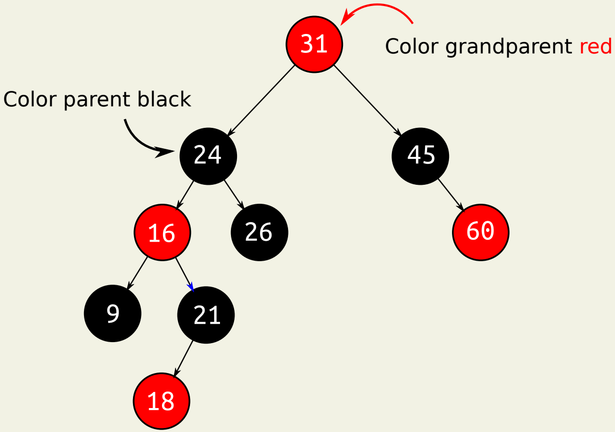
Example 3

Now, Uncle is black and 16 is left child of a left child. This is case 3.



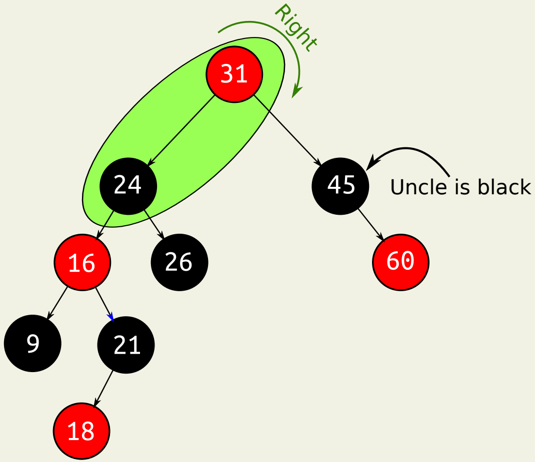
Example 3

Recolor granparent red, parent black.



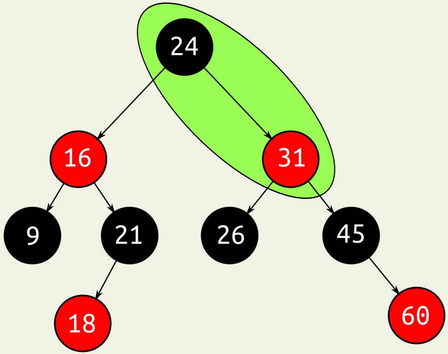
Example 3

Right rotate at grandparent.



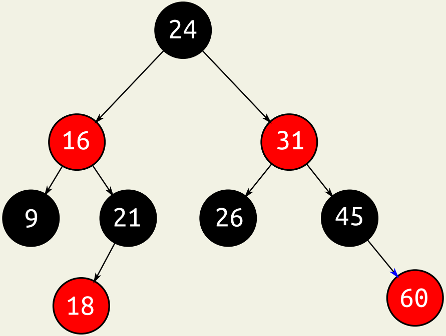
Example 3

Right rotate at grandparent.



Example 3

Done!

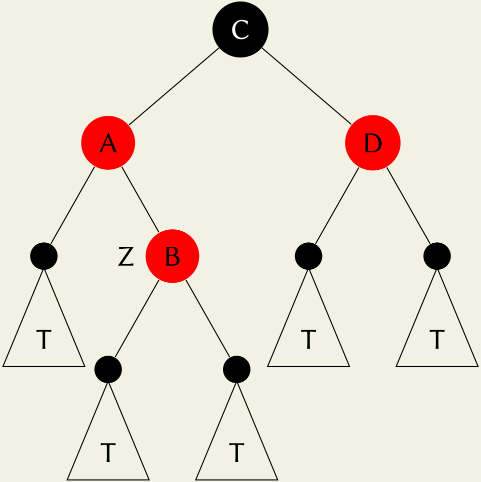


FixInsert pseudocode

Algorithm 1 FixInsert called on node Z

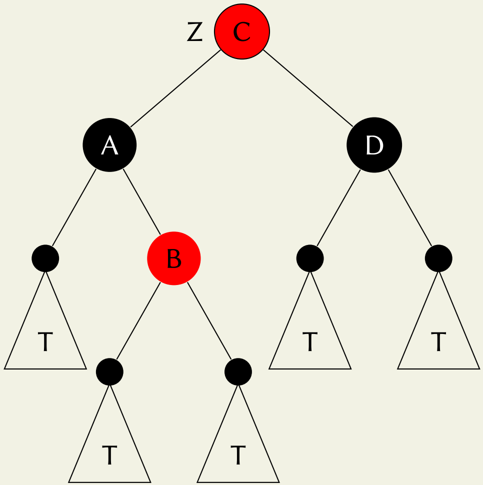
```
while color(parent(Z)) = red do
    U ← Uncle(Z)
    if parent(Z) is the left child of the grandparent then
        if color(U) = red then
            Recolor parent, uncle and grandparent.
            Z ← grandparent(Z).
        else
            if Z is the right child then
                Z ← parent(Z); Left rotate at (Z)
            end if
            Recolor parent and grandparent.
            Right rotate at grandparent(Z).
        end if
    end if
end while
color(root)← black.
```

Case 1



T = subtree of black height k
B could be on either side

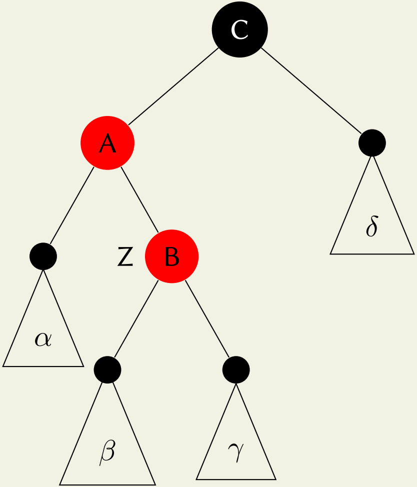
Case 1



T = subtree of black height k

B could be on either side

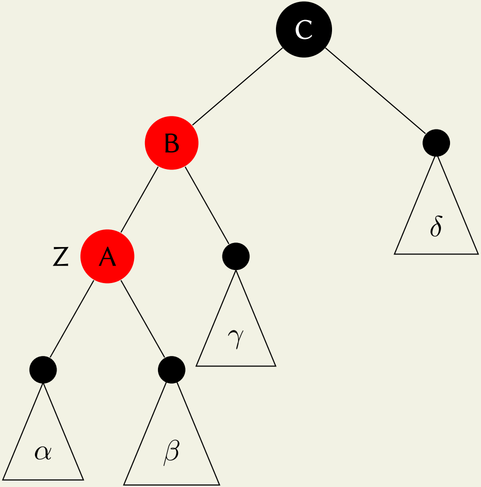
Case 2



$\alpha, \beta, \gamma, \delta =$ subtrees of black height k

We do: Left Rotate at A to get to Case 3

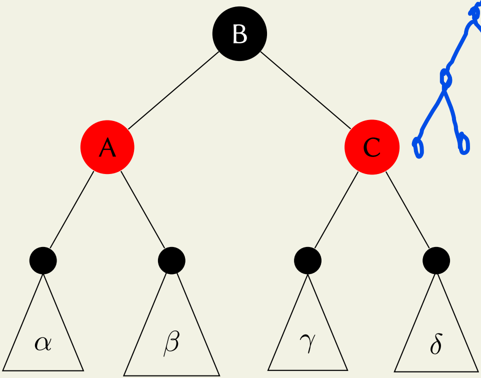
Case 2 –> Case 3

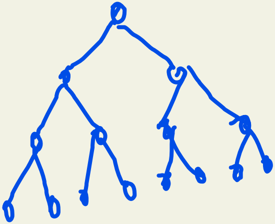


$\alpha, \beta, \gamma, \delta =$ subtrees of black height k

We do right rotate at C

Case 3





$\alpha, \beta, \gamma, \delta =$ subtrees of black height k

Solved!

Summary of Insert

- Case 1: Only recoloring. Pushes up the violation.
- Case 2: Only one rotation. Leads to Case 3.
- Case 3: Only one rotation. No more violations!

Summary of Insert

- Case 1: Only recoloring. Pushes up the violation.
- Case 2: Only one rotation. Leads to Case 3.
- Case 3: Only one rotation. No more violations!

- We have $\leq O(\log n)$ recolorings and ≤ 2 rotations
- Total time: $O(\log n)$

How did anyone come up with such an idea?

- 1962: First self-balancing tree invented by Adelson-Velsky and Landis: AVL Trees
- 1972: Bayer invents “symmetric binary B-trees” which became known as 2-3-4 Trees
- 1978: Guibas and Sedgewick studied 2-3-4 trees and introduced the analogous red-black notion
- Improved over the years

Delete procedure

The procedure to delete a node M at a high level:

- Case 1: M has two non-leaf children.
  - Replace (data of) M with the successor.
  - Splice out (delete) successor. This makes it case 2.

Delete procedure

The procedure to delete a node M at a high level:

- Case 1: M has two non-leaf children.
  - Replace (data of) M with the successor.
  - Splice out (delete) successor. This makes it case 2.
- Case 2: M has at most one non-leaf child. Call this C.
  - Trivial case: M is red.
  - Minor case: M is black but C is red.
  - Major case: M and C are both black.

Note: If M has both leaf children (NIL), then C is any one of the NIL nodes.

M has at most one non-leaf child C

Trivial Case

M is red:

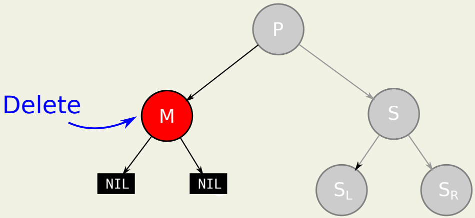
Resolution:

- Then simply replace M with C.

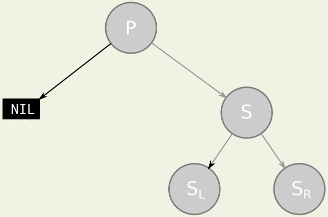
Note:

- M could not have been root.
- For all paths, the black height is not affected.

Trivial Case



Trivial Case



M has at most one non-leaf child C

Minor Case

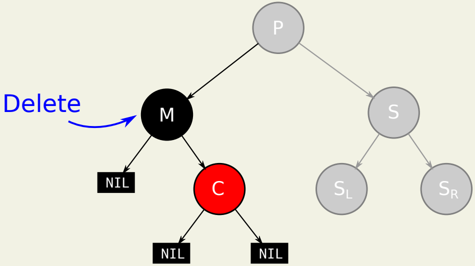
M is black and C is red.

Resolution:

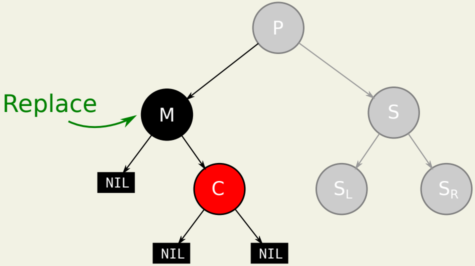
- Replace (splice) M with C.
- Place a “black token” on C.
- Safely discard black token by coloring C black.

The token indicates that the node contributes an extra black to the black count.

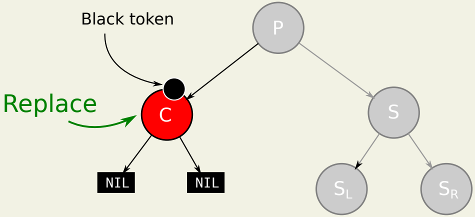
Minor Case



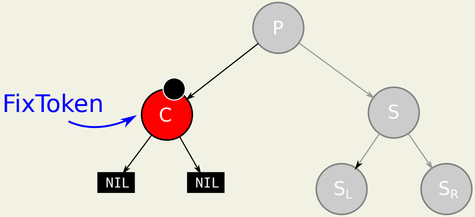
Minor Case



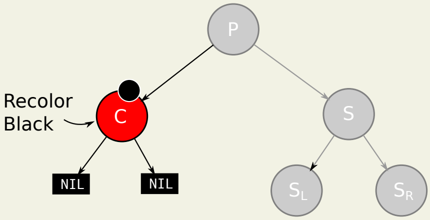
Minor Case



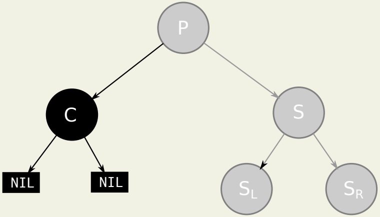
Minor Case



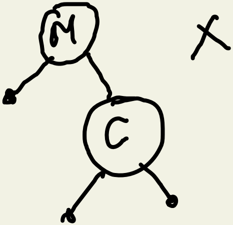
Minor Case



Minor Case



M has at most one non-leaf child C



Major Case

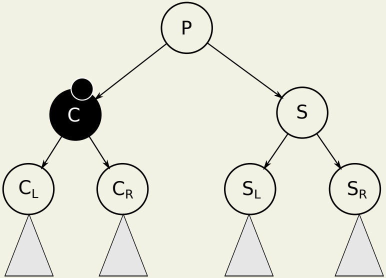
M is black and C is black.

Resolution:

- Replace (splice) M with C.
- Place a black token on C.
- Four cases to safely remove the token placed on C.

Major case - notation

Before replacement of M by C



Removing the token from C

Four cases based on the sibling S of C:

1. S is red.
2. S is black and has both children colored black.
3. S is black and has left child red and right child black.
4. S is black and has right child red.

Case 1

Sibling S of C is red.

Since S is red:

- It must have both black children.
- The parent P of S must be black.

Removing the token from C

Case 1

Sibling S of C is red.

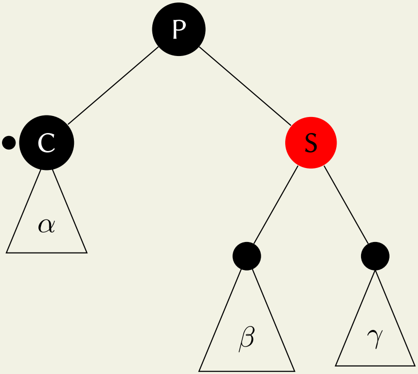
Since S is red:

- It must have both black children.
- The parent P of S must be black.

Resolution:

- Recolor S black and its parent red.
- Rotate at parent. (left rotate if C was left child)
- This is now one of cases 2, 3 or 4.

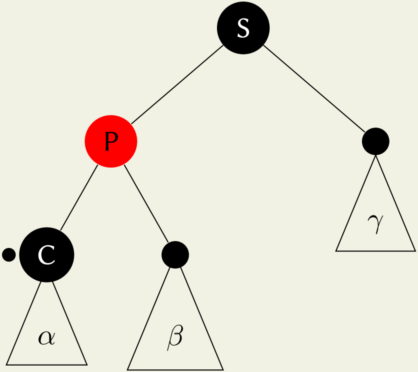
Case 1



$\alpha, \beta, \gamma =$ subtrees of black height k (including the black token)

Left Rotate about P and recolor

Case 1



$\alpha, \beta, \gamma, \delta =$ subtrees of black height *k* (including the black token)

Now in Case 2, 3 or 4

Case 2

Sibling S is black and has both black children.

Removing the token from C

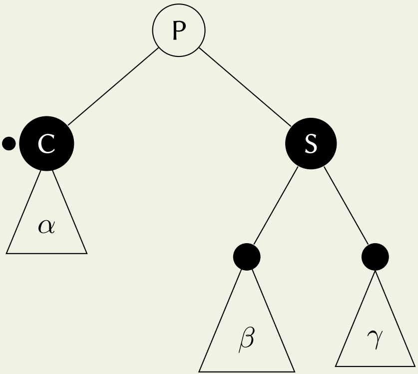
Case 2

Sibling S is black and has both black children.

Resolution:

- Remove a black from both C and S.
- Paste token on the parent P.

Case 2

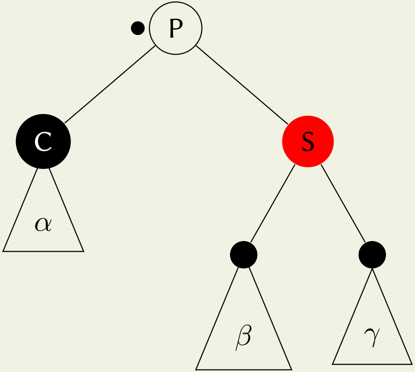


α = subtree of black height $k + 1$ (including the black token)

β, γ = subtrees of black height k

Shift the token to P

Case 2



P can absorb the token if red, else fixup continues

If we came from Case 1, then P is red.

Case 3

Sibling S is black.

Left child S_L is red.

S_R is black.

Case 3

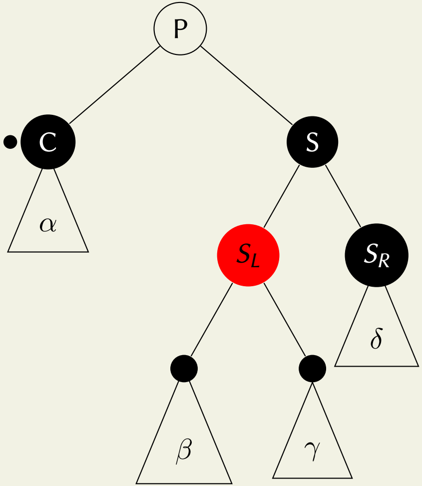
Sibling S is black.
Left child S_L is red.
S_R is black.

Resolution:

- Swap colors of S_L and S.
- Rotate right at S.

This gives us case 4.

Case 3

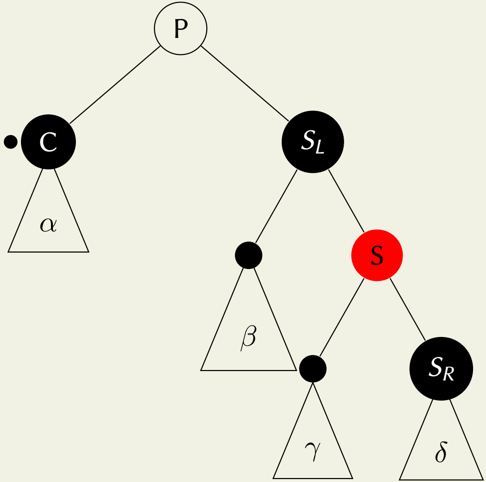


$\alpha =$ subtree of black height $k + 1$ (including the black token)

$\beta, \gamma, \delta =$ subtrees of black height k

Rotate and Recolor

Case 3



Now we are in Case 4

Case 4

Sibling S is black.

Right child S_R is red.

Removing the token from C

Case 4

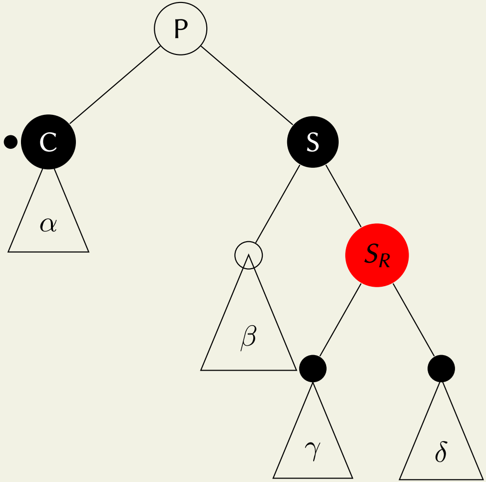
Sibling S is black.

Right child S_R is red.

Resolution:

- Color S_R black.
- S inherits the color of parent P.
- Color P black.
- Rotate left at P.
- Token is removed.

Case 4

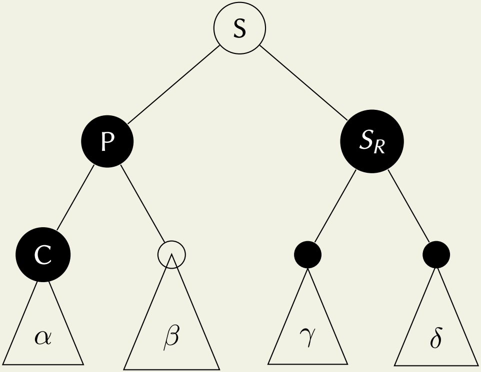


α = subtree of black height $k+1$ (including the black token)

β, γ, δ = subtrees of black height k

Rotate and Recolor

Case 4



$\alpha, \beta, \gamma, \delta =$ subtrees of black height k

Solved!

Case Progression

- Case 1 $\rightarrow$ Case 2 $\rightarrow$ end
- Case 1 $\rightarrow$ Case 3 $\rightarrow$ Case 4 $\rightarrow$ end
- Case 1 $\rightarrow$ Case 4 $\rightarrow$ end
- Case 2 $\rightarrow$ Loop in Case 2 $\rightarrow$ Case 1, 3 or 4
- Case 3 $\rightarrow$ Case 4 $\rightarrow$ end
- Case 4 $\rightarrow$ end