Árboles de búsqueda avanzados

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

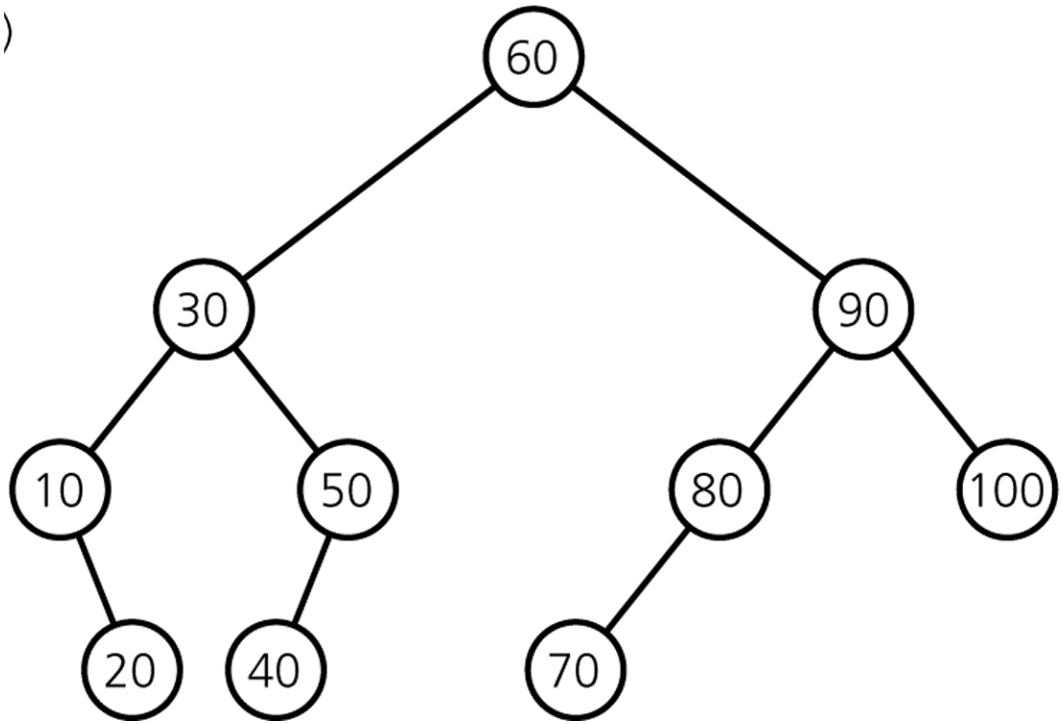
Septiembre 2015

Bibliografía

- F. M. Carrano y T. Henry. *Data Abstraction & Problem Solving with C++: Walls and Mirrors*. Sixth edition. Pearson, 2013.
  Capítulo 19
- R. Sedgewick y K. Wayne. *Algorithms*. Fourth Edition. Addison-Wesley, 2011.
  Sección 3.3
- M. A. Weiss. *Data Structures and Algorithm Analysis in Java*. Third edition. Addison-Wesley, 2012.
  Capítulo 4

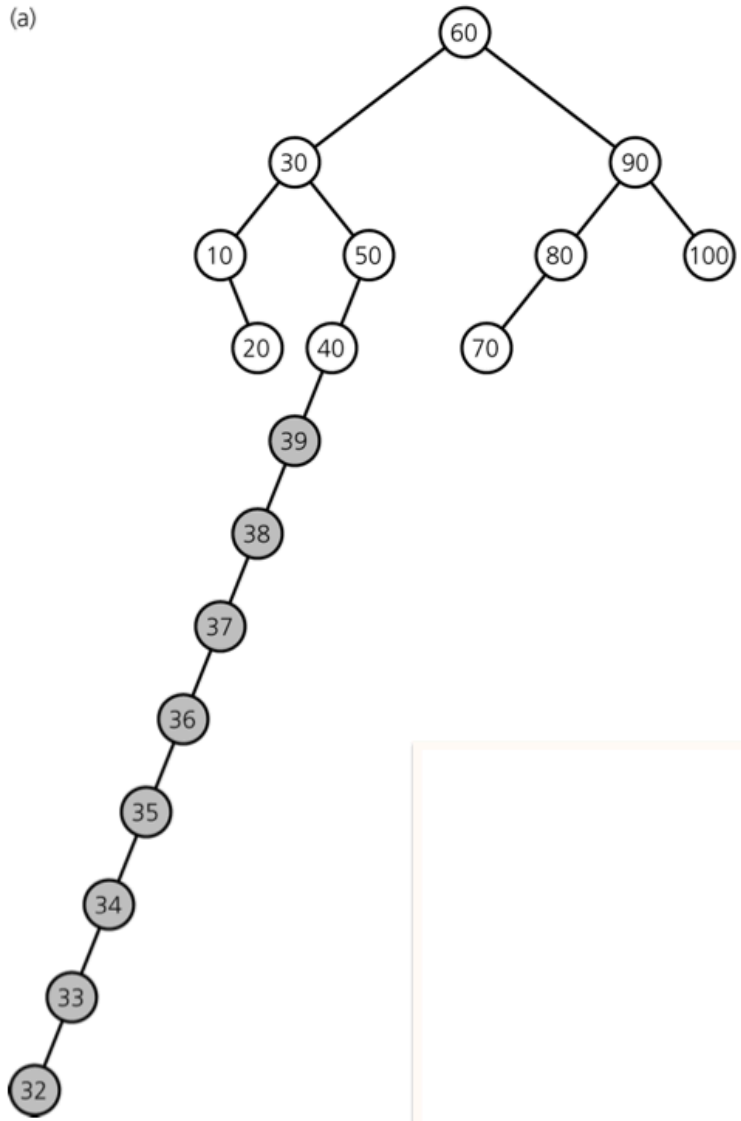
Árboles binarios de búsqueda

Ejemplo



Árboles binarios de búsqueda

La profundidad del árbol puede degenerar.



Análisis del caso medio

- El coste de las operaciones de búsqueda, inserción y borrado está en $O(d)$, siendo d la profundidad del árbol. En el caso peor, $O(N)$.
- Si todas las posibles ordenaciones de la entrada son posibles, la profundidad media sobre todos los nodos está en $O(\log N)$.

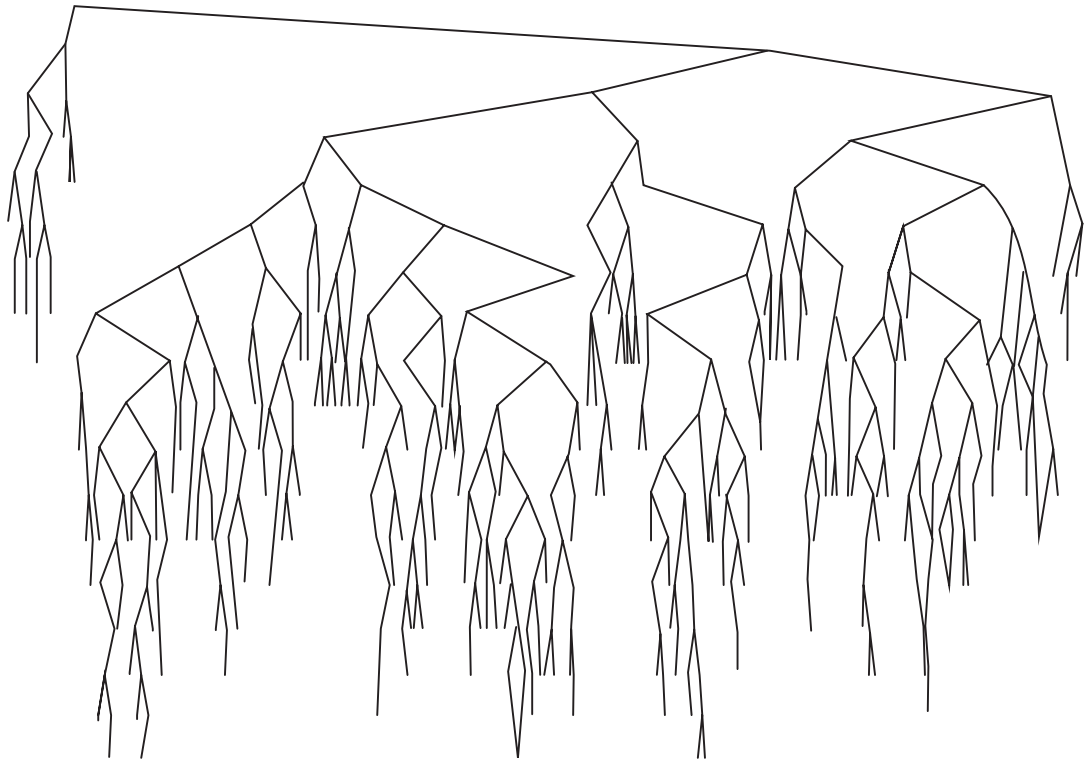


ABB generado aleatoriamente (500 hojas, profundidad media 9.98)

Análisis del caso medio

- Si hay también borrados, no está tan claro que todos los ABBs sean igual de probables.
- De hecho, la estrategia típica de borrado sustituye el nodo borrado por el menor elemento en su hijo derecho.
- El efecto exacto de esta estrategia aún se desconoce.

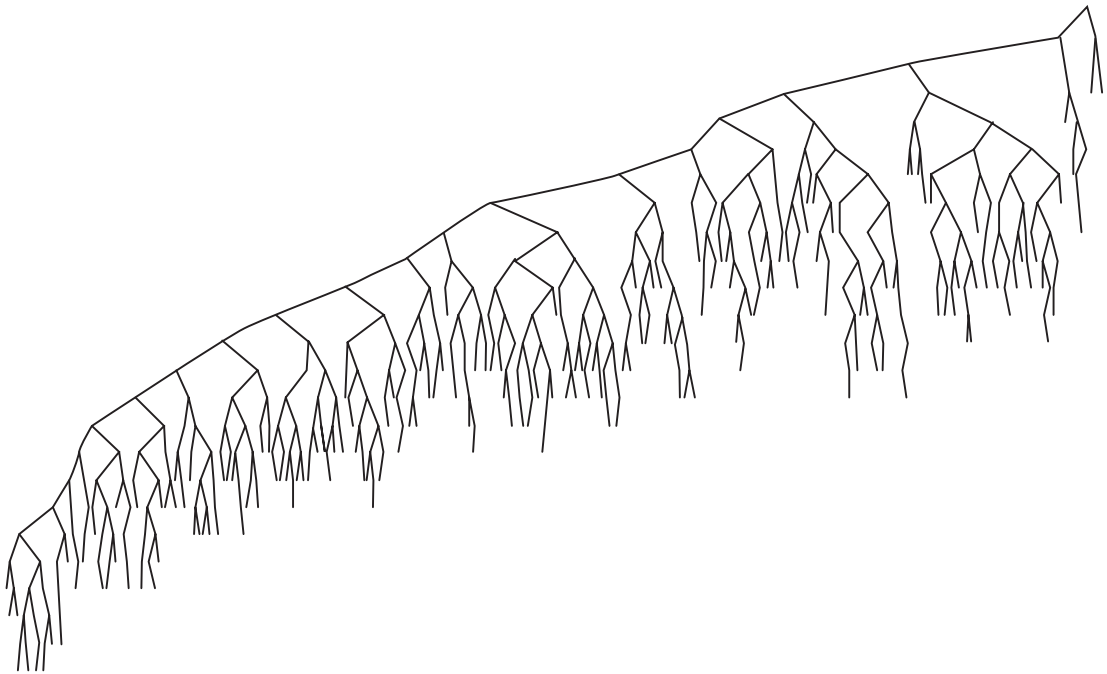
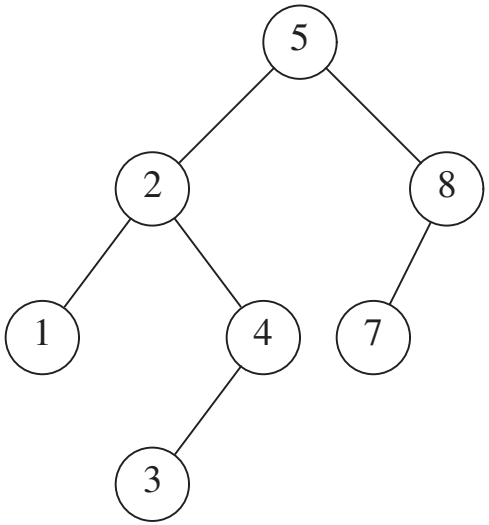
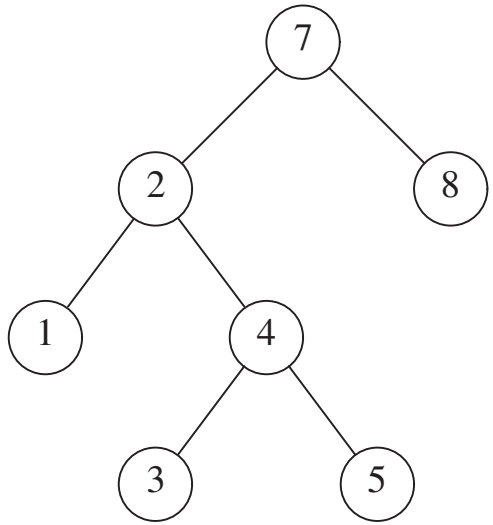


ABB después de $\Theta(N^2)$ pares inserción/borrado

Árboles AVL

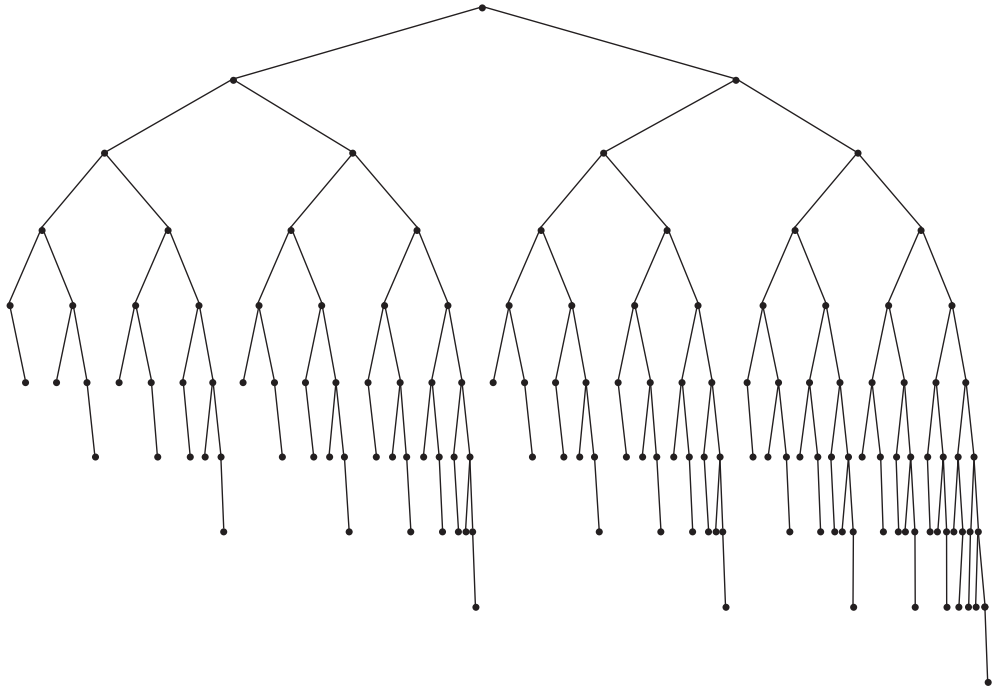
- Los árboles AVL (Adelson-Velskii y Landis, 1962) son ABBs con una condición de equilibrio: todos los nodos del árbol cumplen que la diferencia de alturas de sus dos hijos es como mucho 1.





Árboles AVL

- La altura de un árbol AVL es como mucho $1.44 \log(N+2) - 1.328$.
- Árbol AVL más pequeño de altura 9



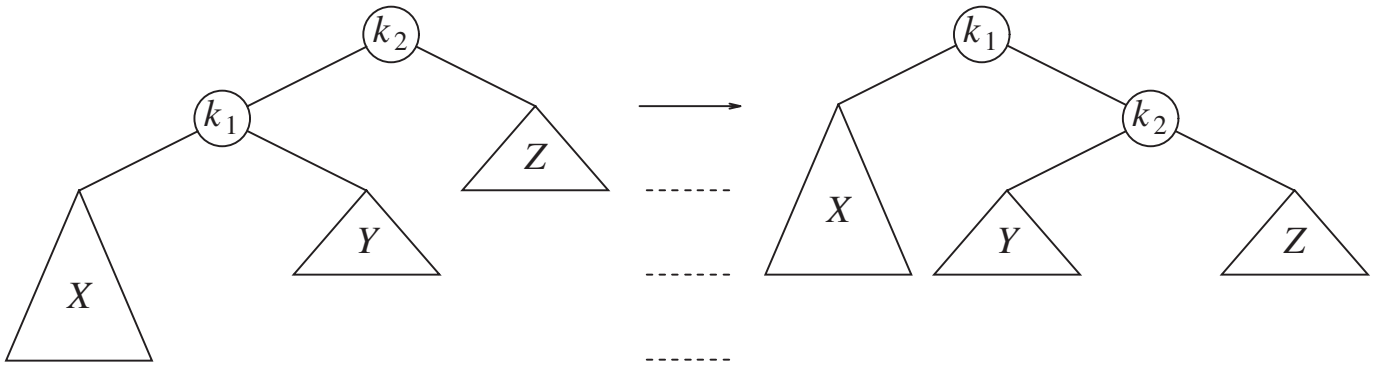
- El mínimo número de nodos, $S(h)$, en un árbol AVL de altura h viene dado por $S(h) = S(h-1) + S(h-2) + 1$, con $S(0) = 1$ y $S(1) = 2$. La función $S(h)$ está relacionada con los números de Fibonacci, de donde se saca la cota anterior.

Árboles AVL

- En un árbol AVL todas las operaciones pueden hacerse en $O(\log N)$.
- La inserción de un nodo puede hacer que deje de cumplirse la condición de equilibrio.
- Si ese es el caso, el árbol debe reestructurarse mediante rotaciones.
- Después de una inserción, solo los nodos en el camino desde el nodo insertado a la raíz pueden haberse desequilibrado. Sea α el primer nodo desequilibrado en ese camino. La diferencia de alturas entre sus hijos es 2, con cuatro casos posibles:
  1. Inserción en el subárbol izquierdo del hijo izquierdo de α.
  2. Inserción en el subárbol derecho del hijo izquierdo de α.
  3. Inserción en el subárbol izquierdo del hijo derecho de α.
  4. Inserción en el subárbol derecho del hijo derecho de α.

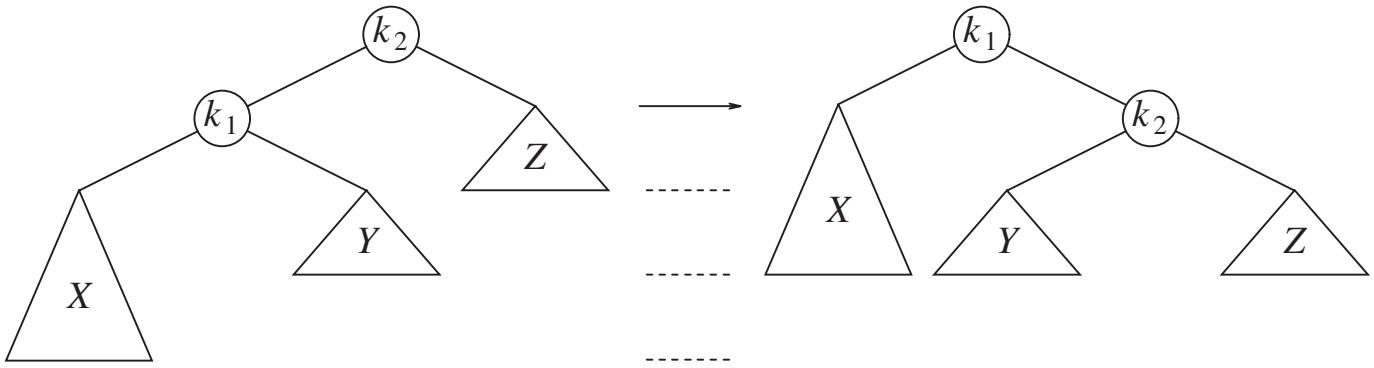
AVL: Rotación simple a la derecha

- El caso 1 puede equilibrarse con la siguiente rotación:

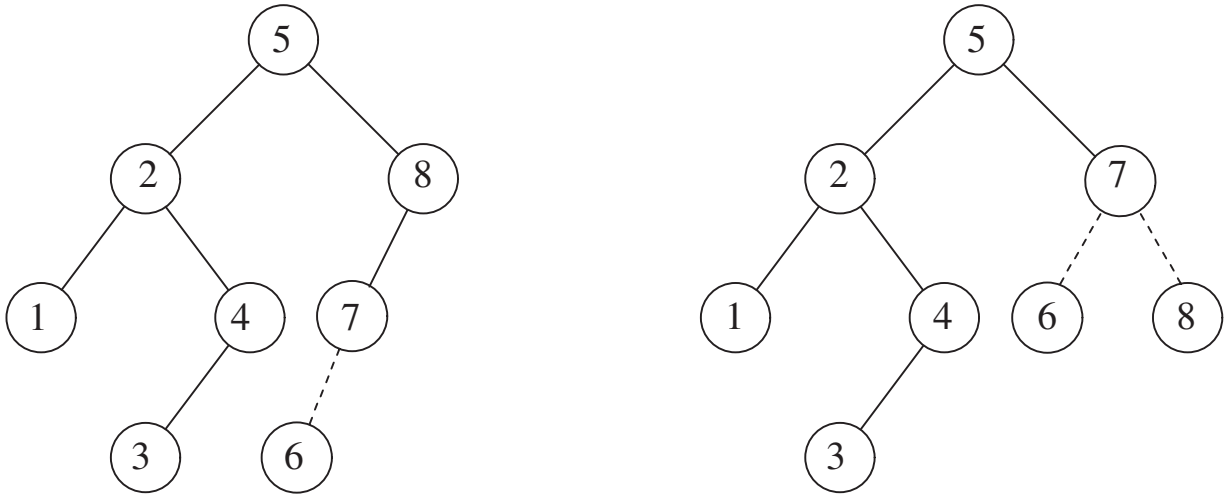


AVL: Rotación simple a la derecha

- El caso 1 puede equilibrarse con la siguiente rotación:

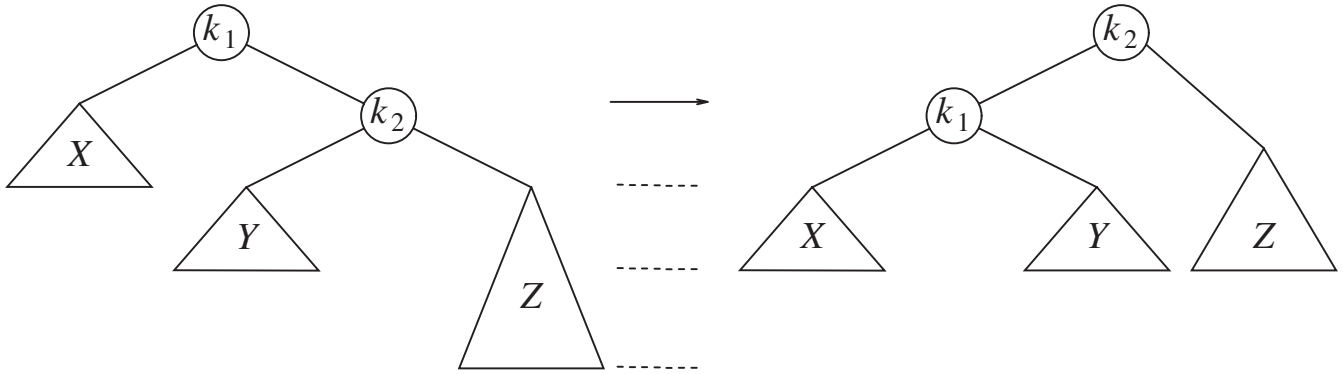


- Por ejemplo,



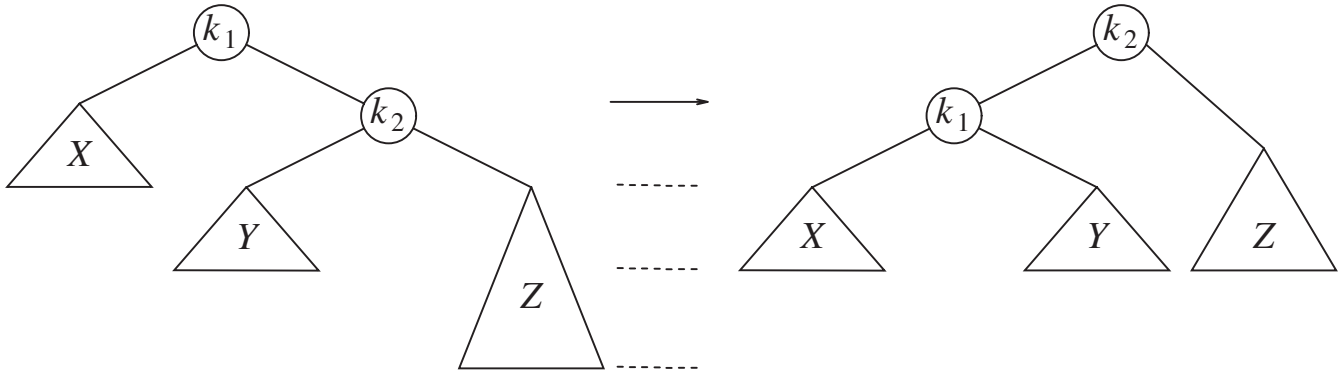
AVL: Rotación simple a la izquierda

- Necesaria en el caso 4:



AVL: Rotación simple a la izquierda

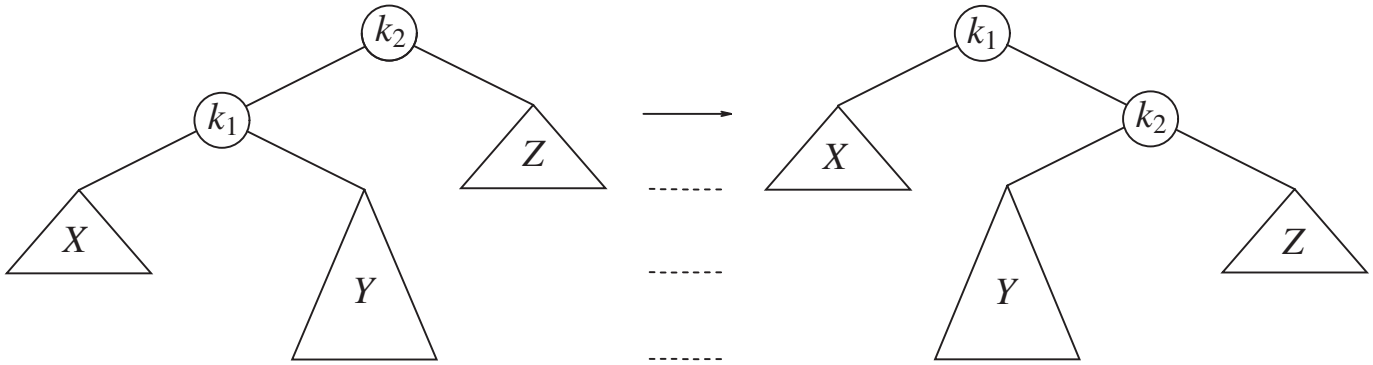
- Necesaria en el caso 4:



- Ejercicio: Insertar en un árbol AVL vacío los elementos 3, 2, 1 y del 4 al 7.

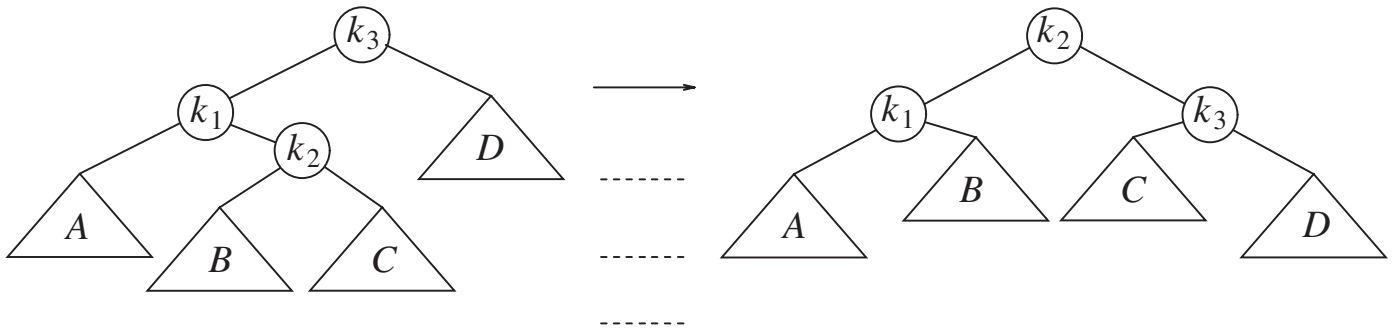
AVL: Rotación doble

- La rotación simple no funciona en los casos 2 y 3:



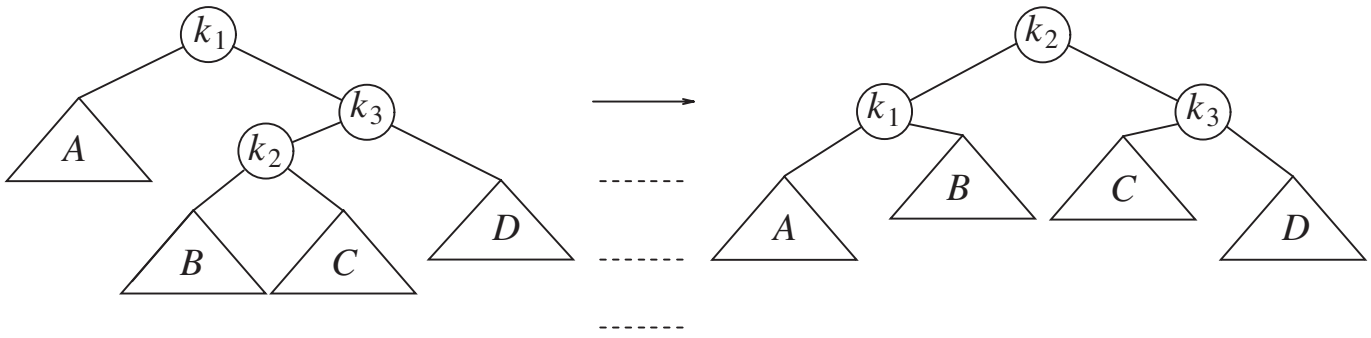
AVL: Rotación doble izquierda-derecha

- Hace falta una rotación doble. El hecho de que se haya insertado un nodo en el subárbol Y garantiza que no es vacío.



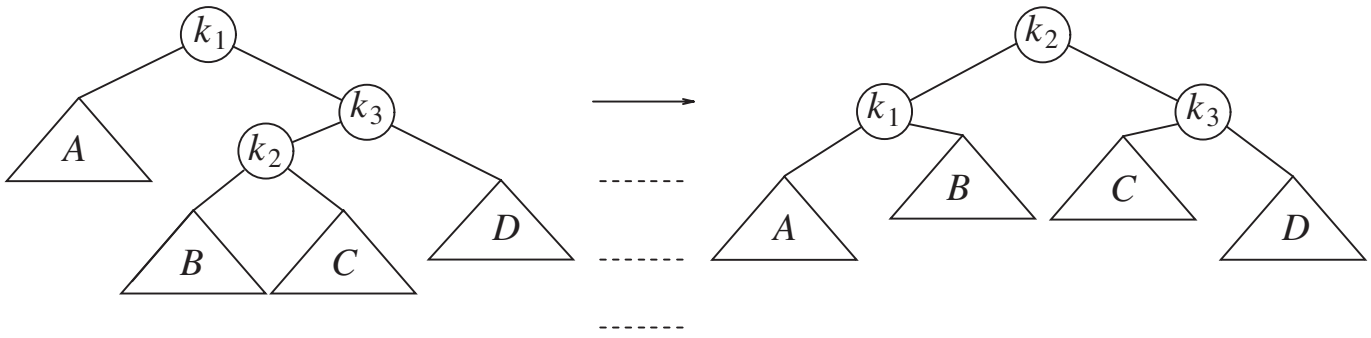
AVL: Rotación doble derecha-izquierda

- Para resolver el caso 3:



AVL: Rotación doble derecha-izquierda

- Para resolver el caso 3:



- Ejercicio: Insertar en el árbol AVL anterior del 10 al 16 en orden inverso y luego el 8 y el 9.

Implementación de los árboles AVL

```
template <typename Clave, typename Valor, typename Comparador = std::less<Clave>>
class TreeMap {
public:
   struct ClaveValor {
       const Clave clave;
       Valor valor;
   };
protected:
   /**
    Clase nodo que almacena internamente la pareja (clave, valor),
    los punteros al hijo izquierdo y al hijo derecho, y la altura.
    */
   using Link = TreeNode *;
   class TreeNode {
   public:
      ClaveValor cv;
      Link iz;
      Link dr;
      int altura;
      TreeNode(ClaveValor e, Link i = nullptr, Link d = nullptr, int alt = 1)
      : cv(e), iz(i), dr(d), altura(alt) {}
   };
```

Implementación de los árboles AVL

```
protected:

    /** Puntero a la raíz de la estructura jerárquica de nodos. */
    Link raiz;

    /** Objeto función que compara elementos. */
    Comparador menor;


public:

    TreeMap(Comparador c = Comparador()) : raiz(nullptr), menor(c) {};
```

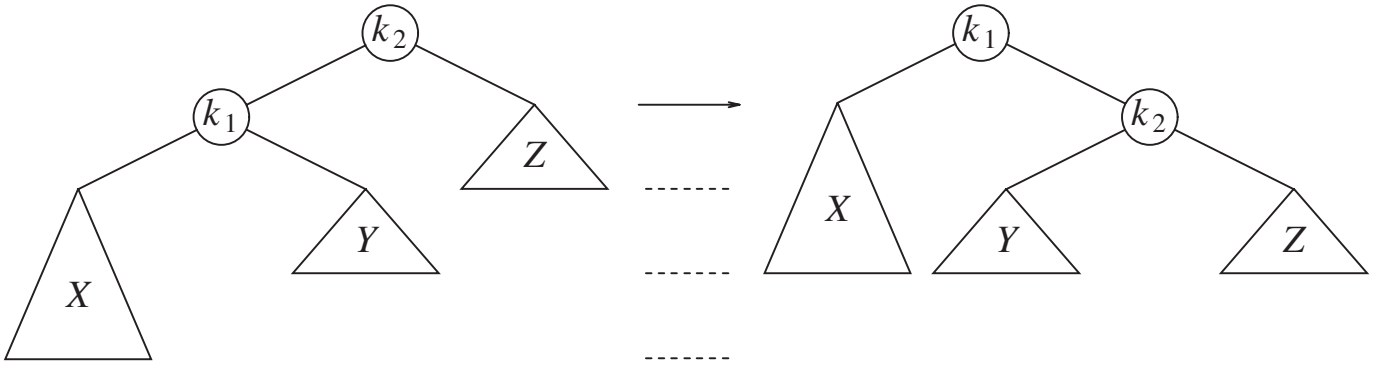
Implementación de los árboles AVL

```
public:
   Valor const& at(Clave const& clave) const {
      Link p = buscar(clave, raiz);
      if (p == nullptr)
          throw std::out_of_range("La clave no se puede consultar");
      return p->cv.valor;
   }

protected:
   Link buscar(Clave const& clave, Link a) const {
      if (a == nullptr)
          return nullptr;
      else if (menor(clave, a->cv.clave))
          return buscar(clave, a->iz);
      else if (menor(a->cv.clave, clave))
          return buscar(clave, a->dr);
      else // clave == a->cv.clave
          return a;
   }
```

Implementación de los árboles AVL

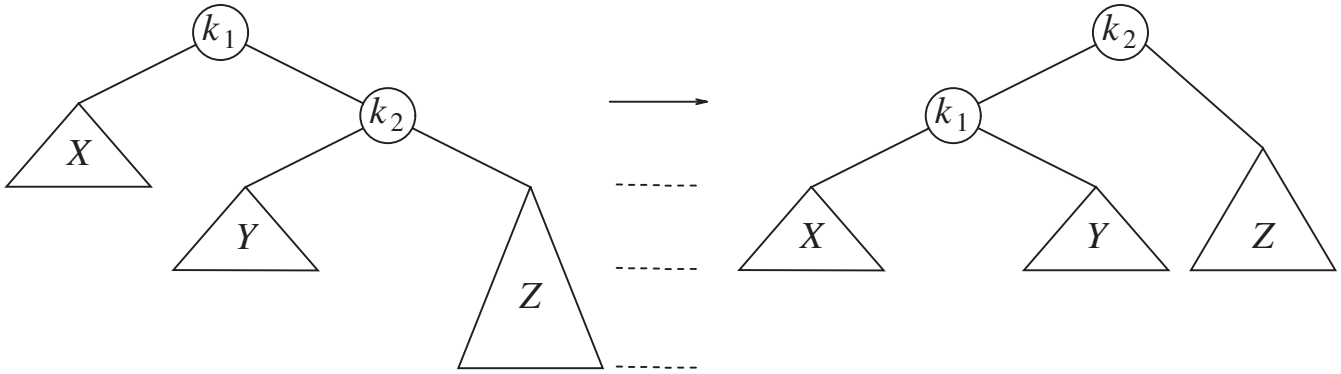
```
public:
   void insert(Clave const& clave, Valor const& valor) {
      insertar({clave, valor}, raiz);
   }

protected:
   void insertar(ClaveValor const& cv, Link & a) {
      if (a == nullptr) {
          a = new TreeNode(cv);
      } else if (menor(cv.clave, a->cv.clave)) {
          insertar(cv, a->iz);
          reequilibraDer(a);
      } else if (menor(a->cv.clave, cv.clave)) {
          insertar(cv, a->dr);
          reequilibraIzq(a);
      } else { // la clave ya está, se actualiza el valor asociado
          a->cv.valor = cv.valor;
      }
   }
```

Implementación de los árboles AVL



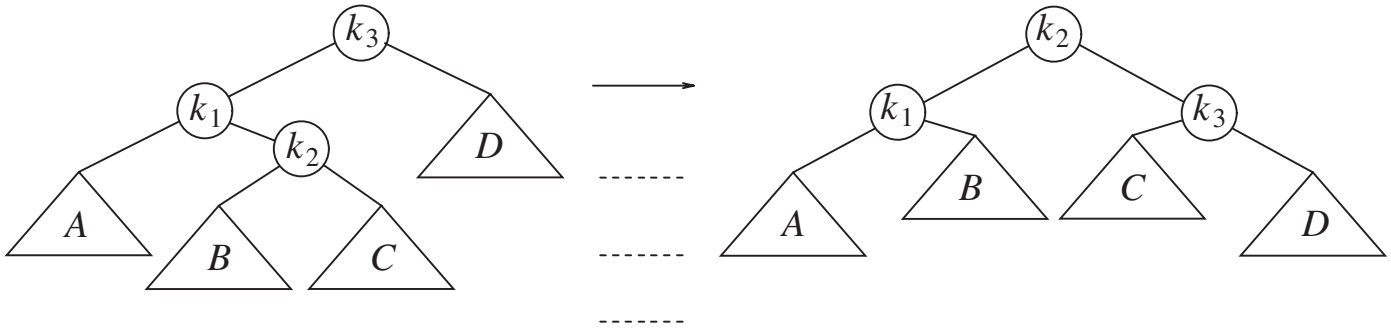

```
static void rotaDer(Link & k2){
   Link k1 = k2->iz;
   k2->iz = k1->dr;
   k1->dr = k2;
   k2->altura = std::max(altura(k2->iz),altura(k2->dr))+1;
   k1->altura = std::max(altura(k1->iz),altura(k1->dr))+1;
   k2 = k1;
}

static int altura(Link a){
   if (a == nullptr) return 0;
   else return a->altura;
}
```

Implementación de los árboles AVL



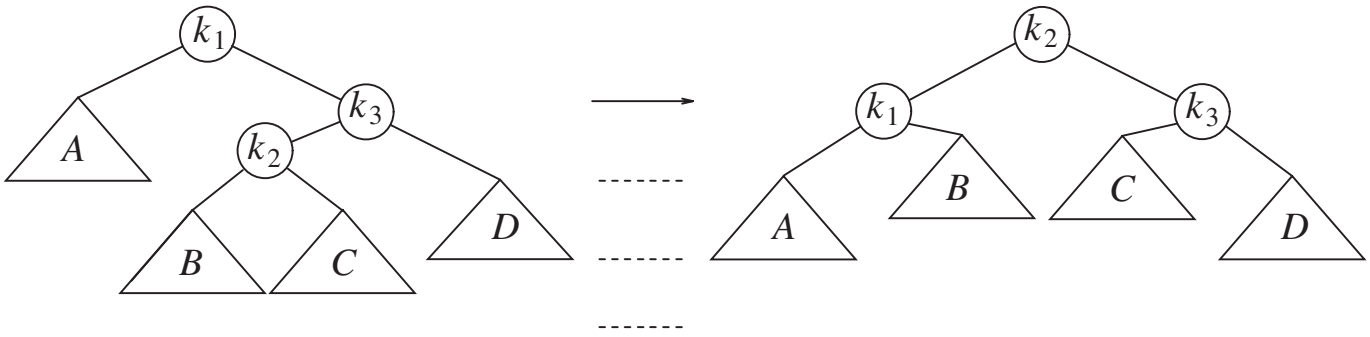

```
static void rotaIzq(Link & k1){
   Link k2 = k1->dr;
   k1->dr = k2->iz;
   k2->iz = k1;
   k1->altura = std::max(altura(k1->iz),altura(k1->dr))+1;
   k2->altura = std::max(altura(k2->iz),altura(k2->dr))+1;
   k1=k2;
}
```

Implementación de los árboles AVL




```
static void rotaIzqDer(Link & k3){
   rotaIzq(k3->iz);
   rotaDer(k3);
}
```

Implementación de los árboles AVL




```
static void rotaDerIzq(Link & k1){
   rotaDer(k1->dr);
   rotaIzq(k1);
}
```

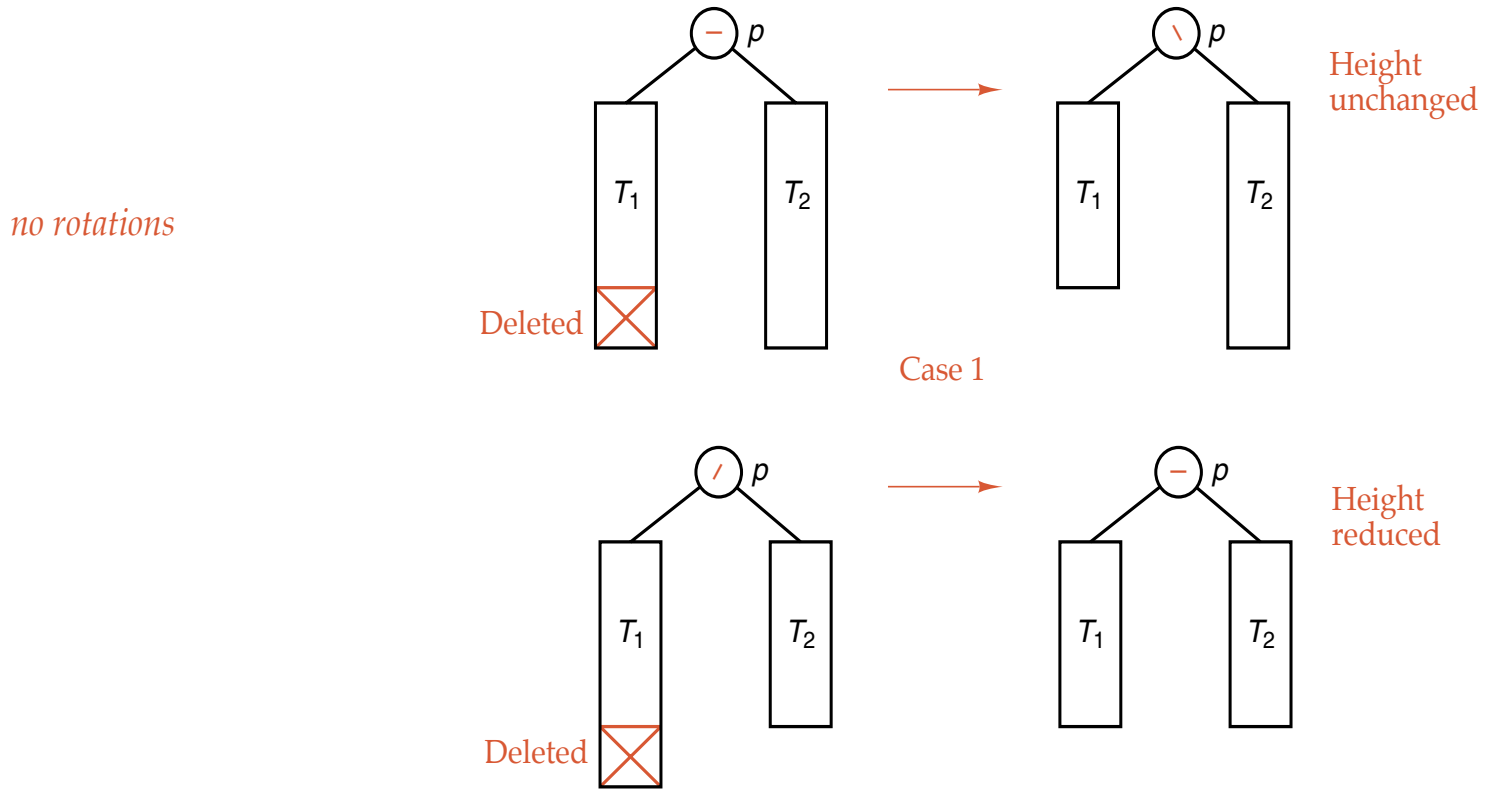
Implementación de los árboles AVL

```
static void reequilibraIzq(Link & a){
   if (altura(a->dr)-altura(a->iz) > 1) {
      if (altura(a->dr->iz) > altura(a->dr->dr))
         rotaDerIzq(a);
      else rotaIzq(a);
   }
   else a->altura = std::max(altura(a->iz),altura(a->dr))+1;
}
```

```
static void reequilibraDer(Link & a){
   if (altura(a->iz)-altura(a->dr) > 1) {
      if (altura(a->iz->dr) > altura(a->iz->iz))
         rotaIzqDer(a);
      else rotaDer(a);
   }
   else a->altura = std::max(altura(a->iz),altura(a->dr))+1;
}
```

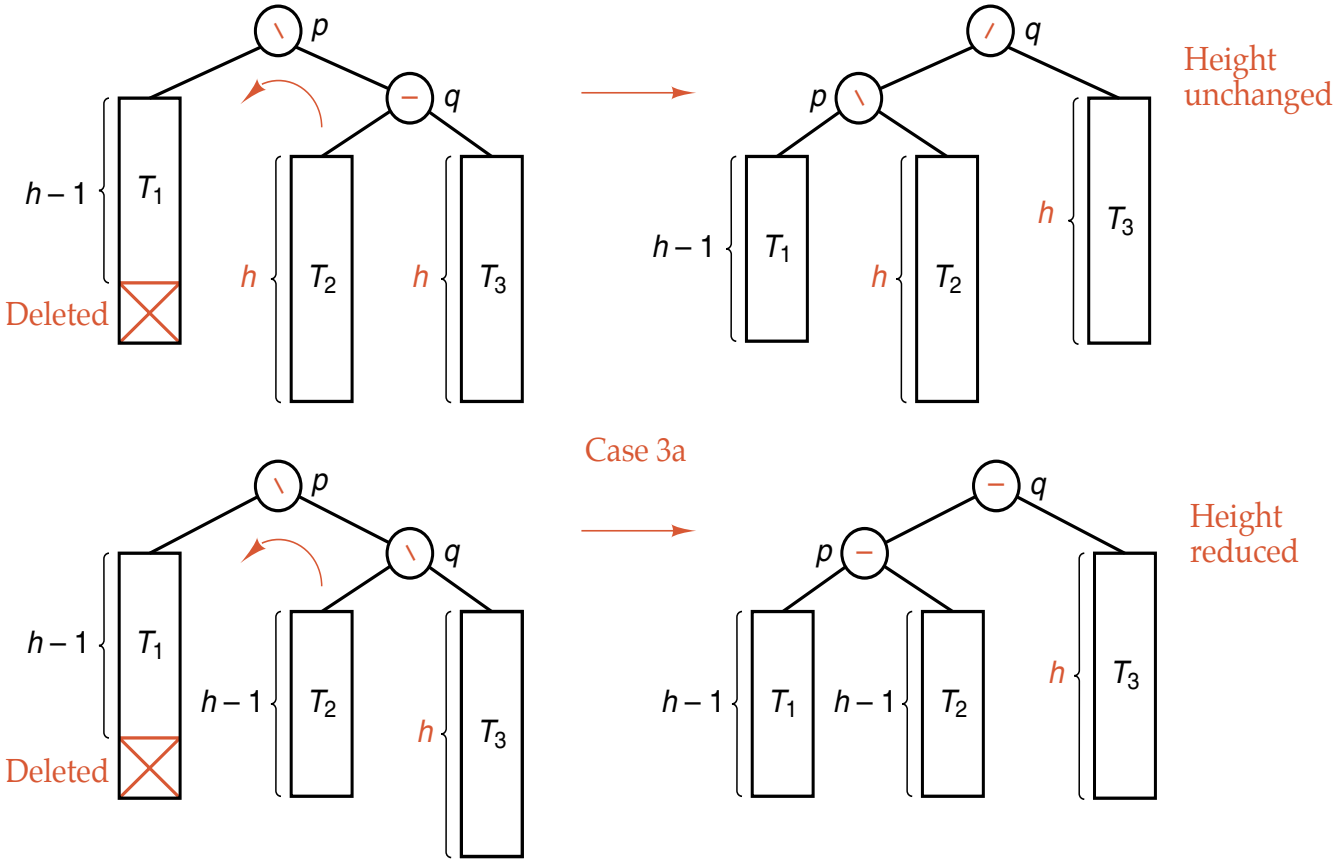
Borrado en árboles AVL

- Sigue los mismos pasos que el borrado en un ABB. Se reduce al caso de que el nodo a eliminar tiene solamente un hijo, por lo que se puede sustituir por él. La altura decrece por lo que pueden hacer falta rotaciones si algún subárbol se desequilibra.

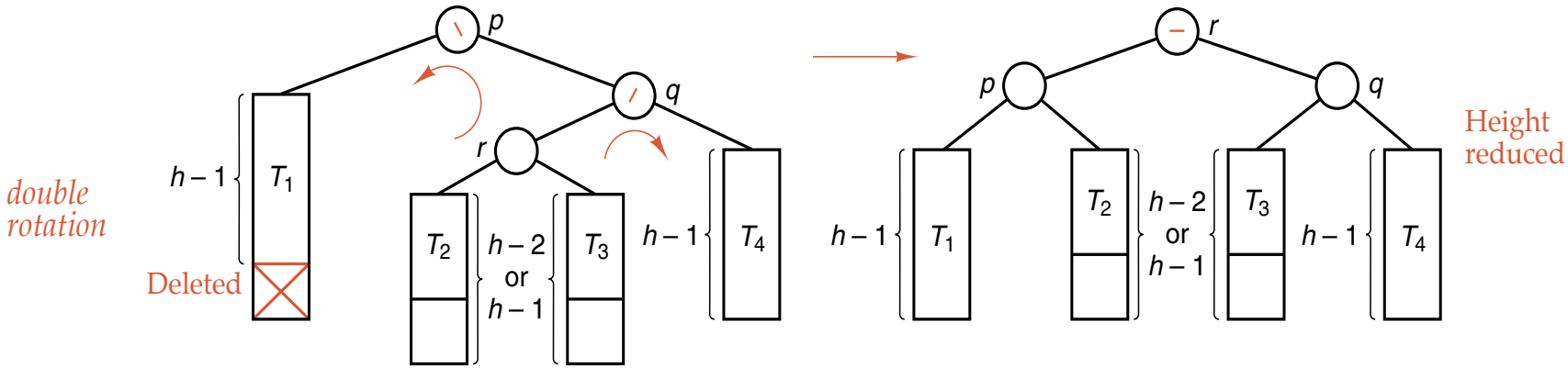


Borrado en árboles AVL

single left rotations



Borrado en árboles AVL



2-3 tree

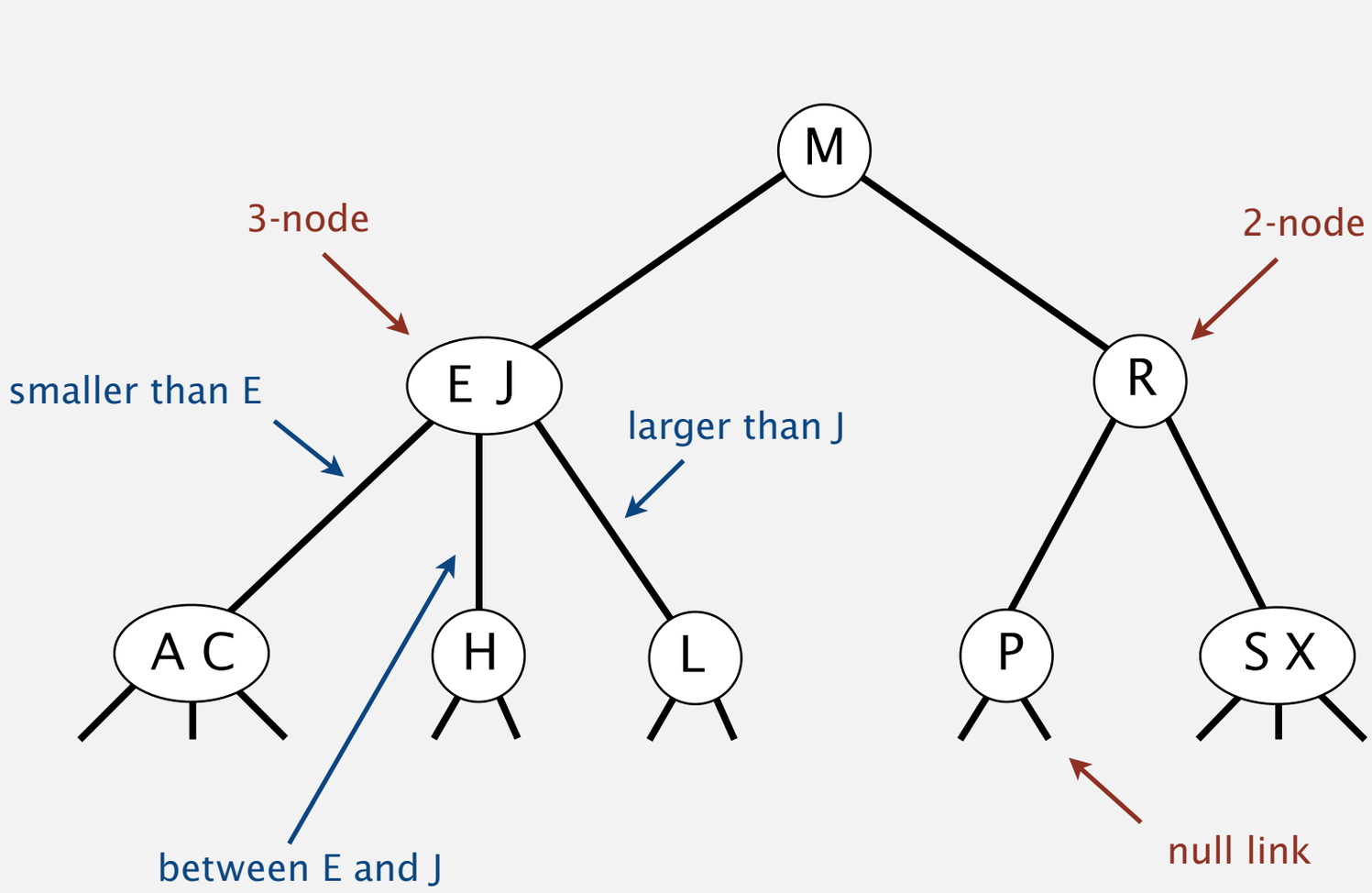
Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.

Symmetric order. Inorder traversal yields keys in ascending order.

Perfect balance. Every path from root to null link has same length.

how to maintain?

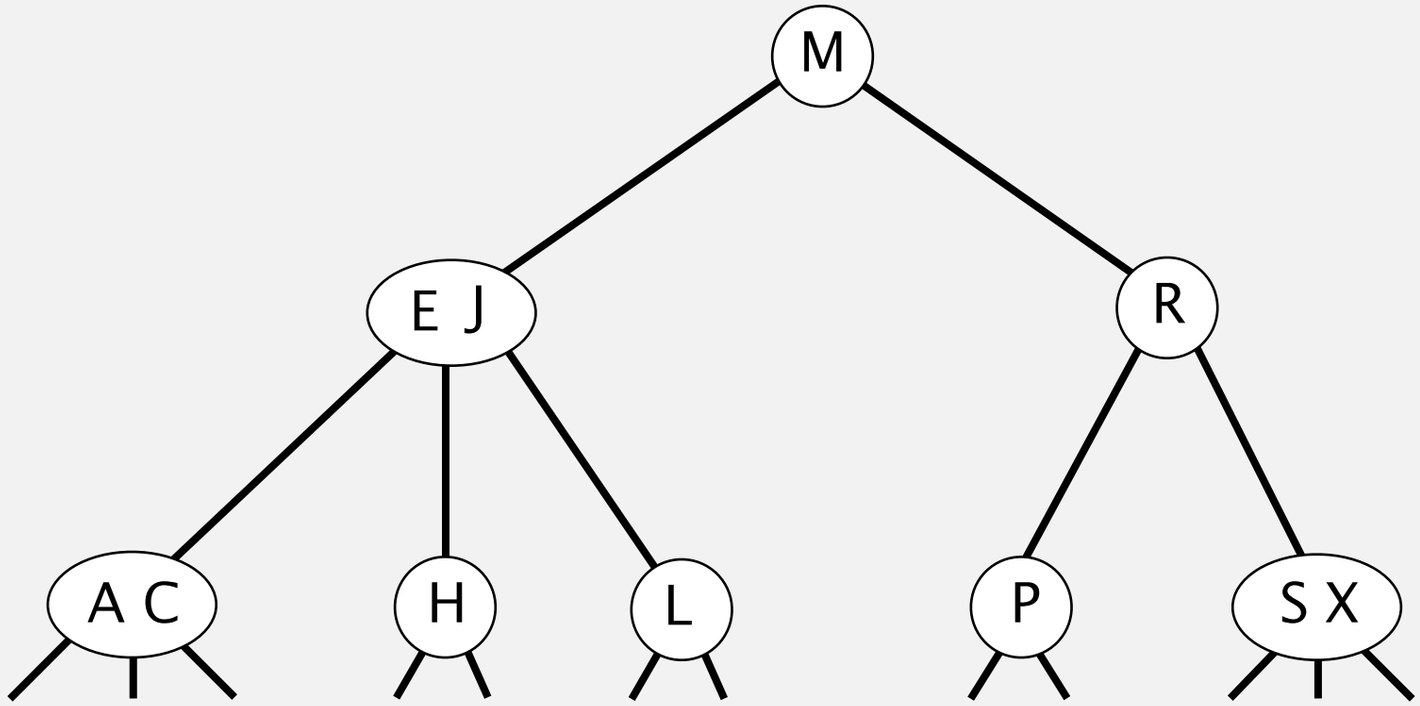


2-3 tree demo

Search.

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

search for H

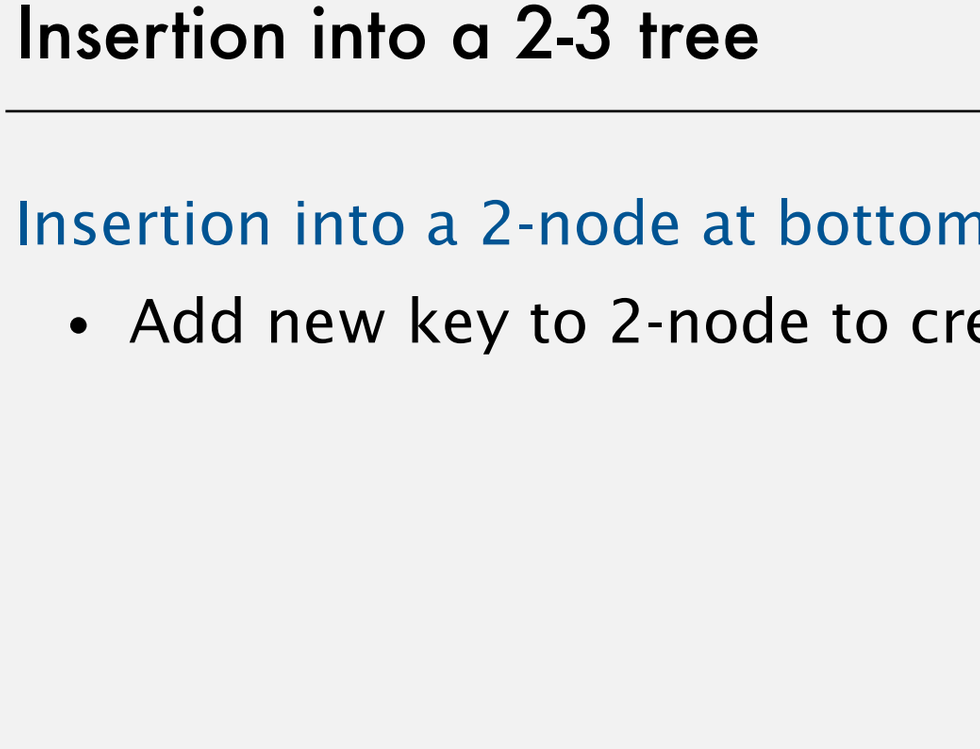


Insertion into a 2-3 tree

Insertion into a 2-node at bottom.

- Add new key to 2-node to create a 3-node.

insert G

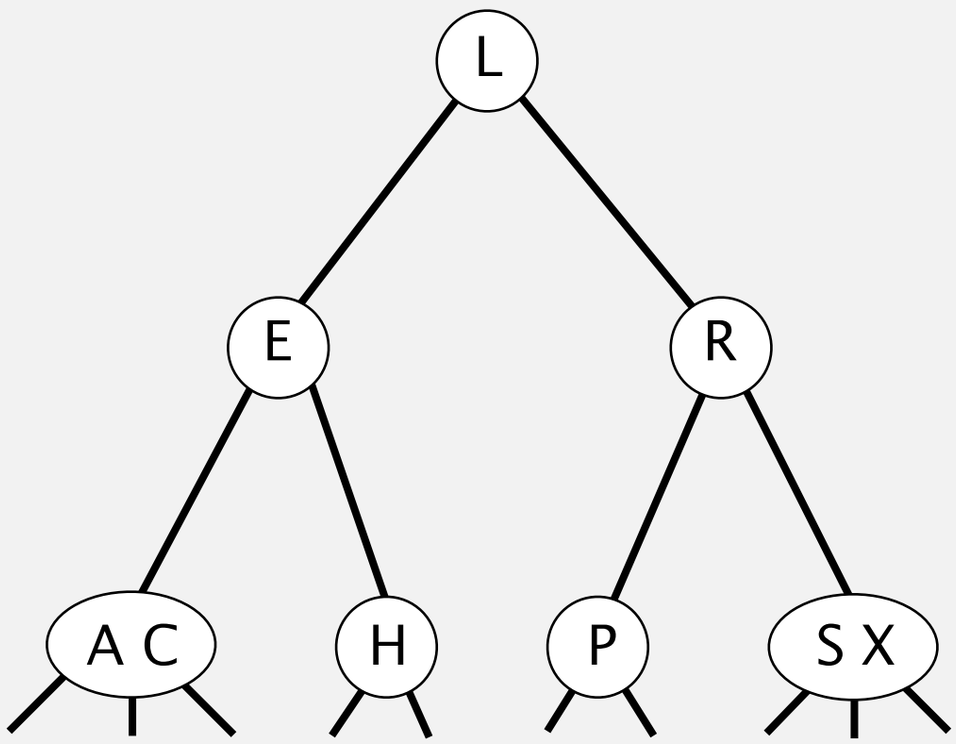


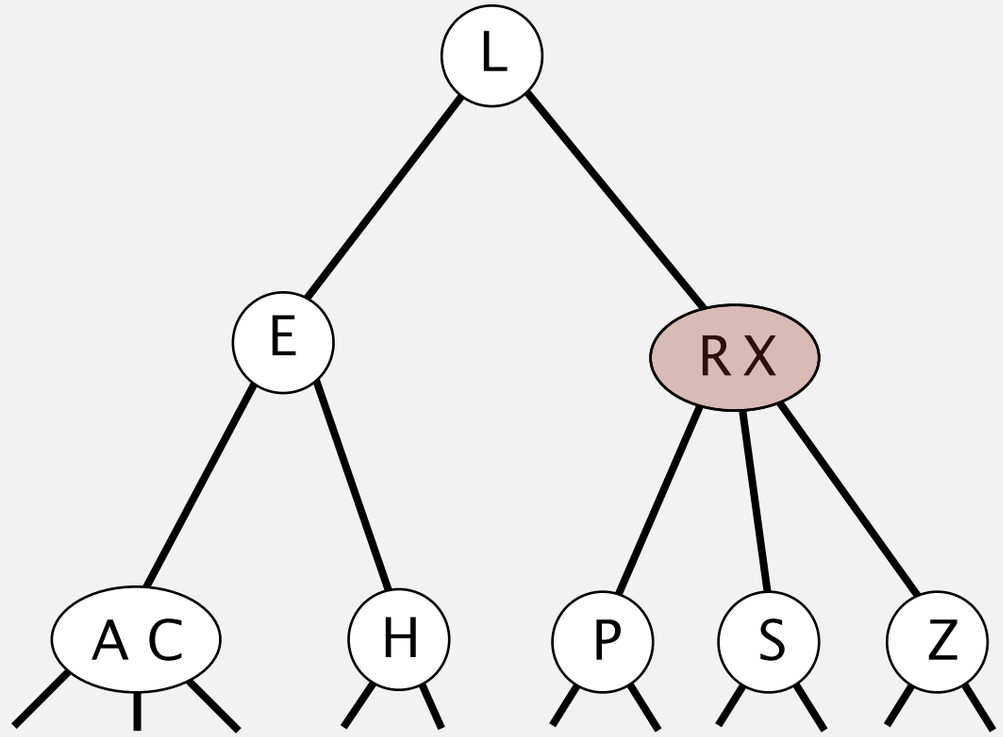
Insertion into a 2-3 tree

Insertion into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

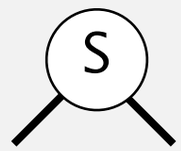
insert Z





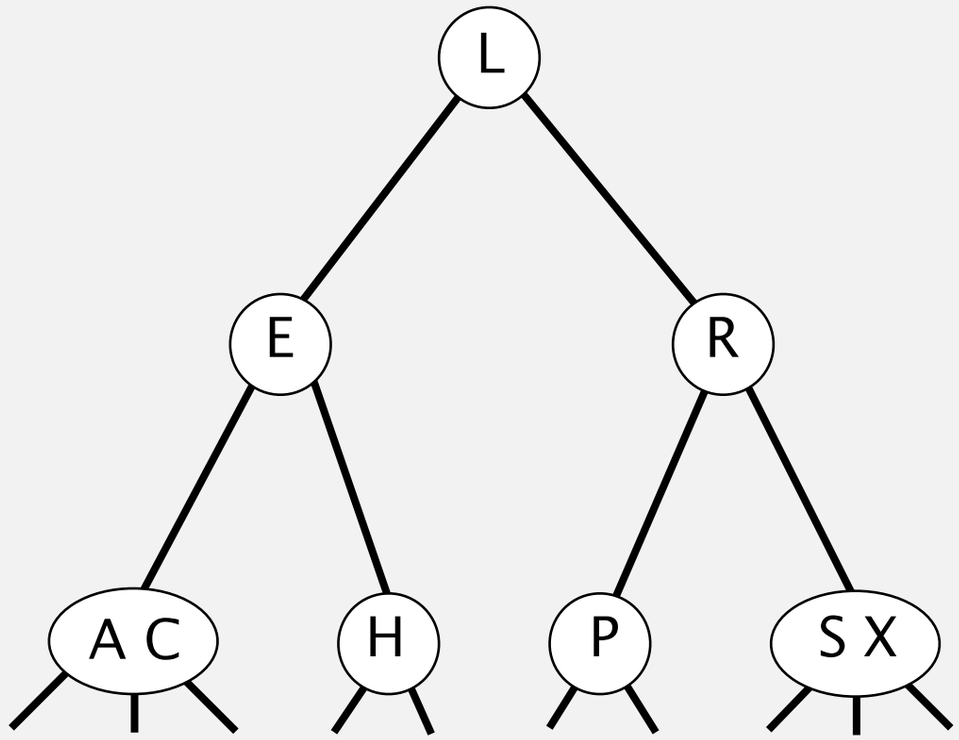
2-3 tree construction demo

insert S



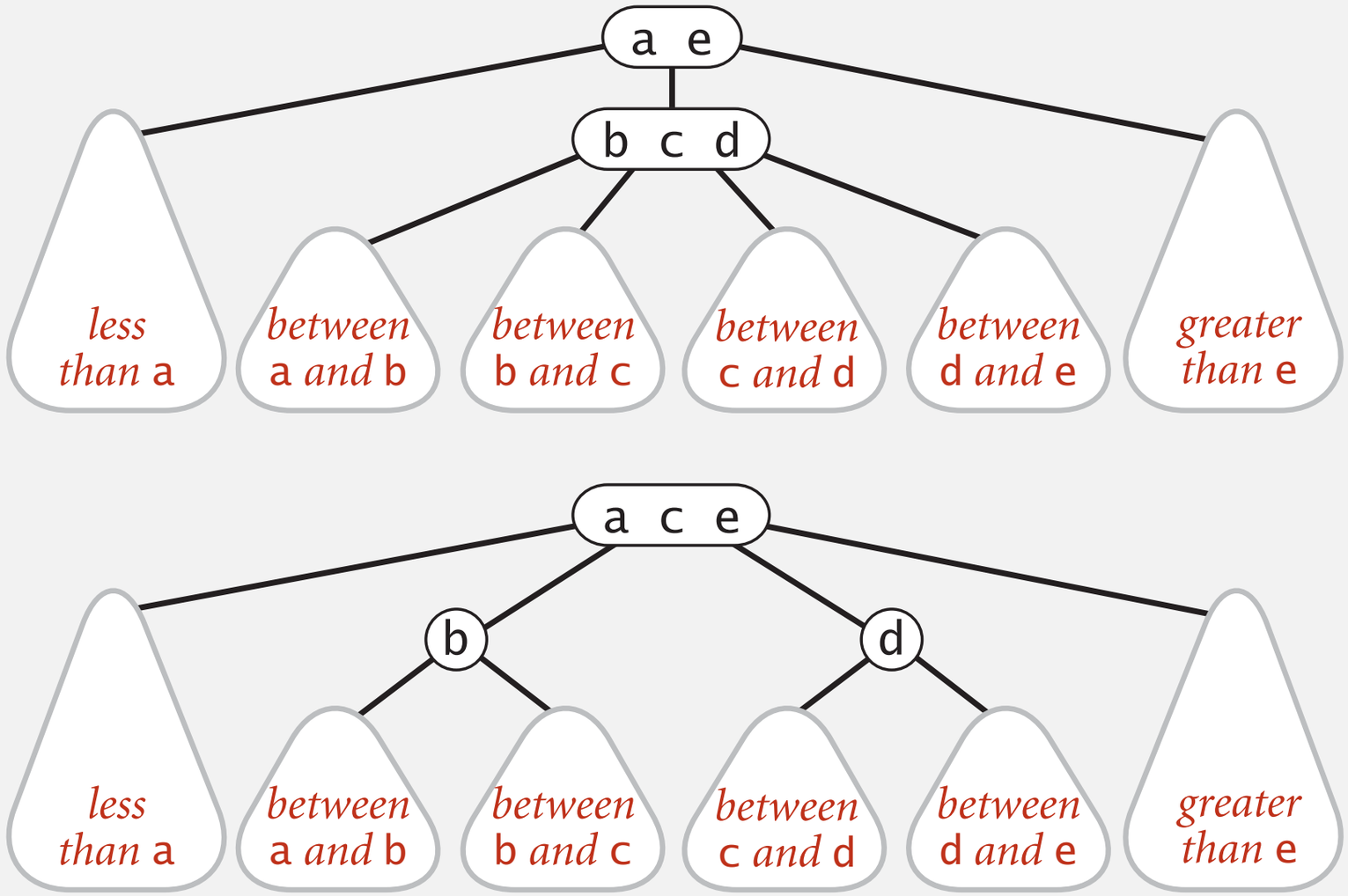
2-3 tree construction demo

2-3 tree



Local transformations in a 2-3 tree

Splitting a 4-node is a local transformation: constant number of operations.



Global properties in a 2-3 tree

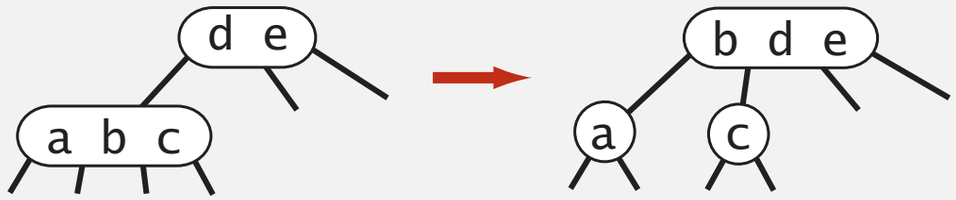
Invariants. Maintains symmetric order and perfect balance.

Pf. Each transformation maintains symmetric order and perfect balance.

root

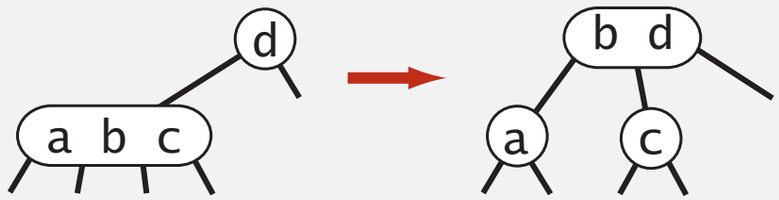
parent is a 3-node

left

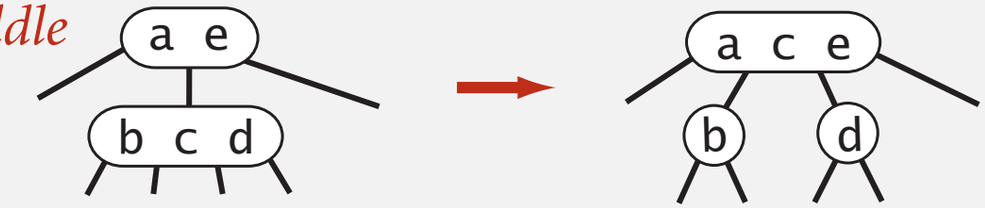


parent is a 2-node

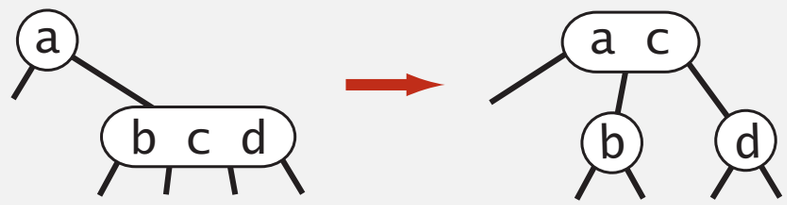
left



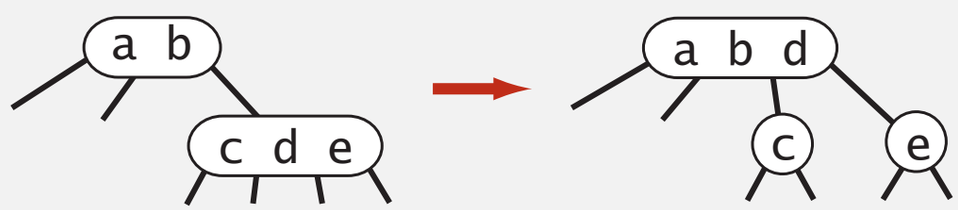
middle



right

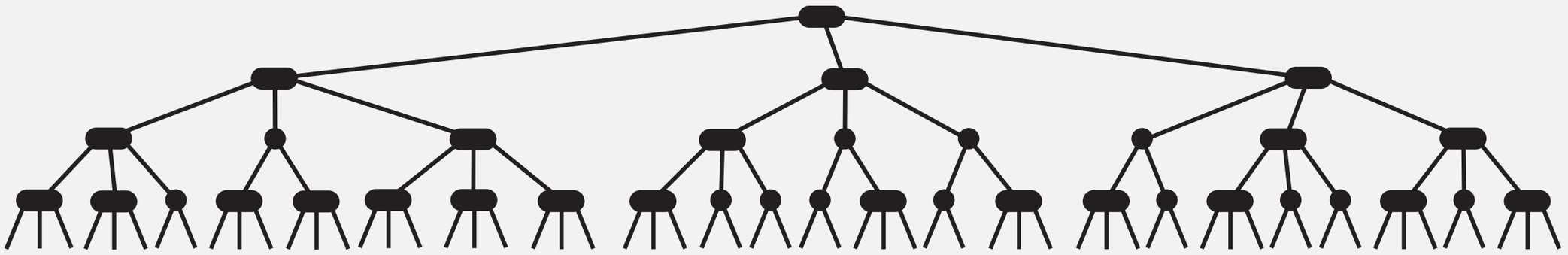


right



2-3 tree: performance

Perfect balance. Every path from root to null link has same length.

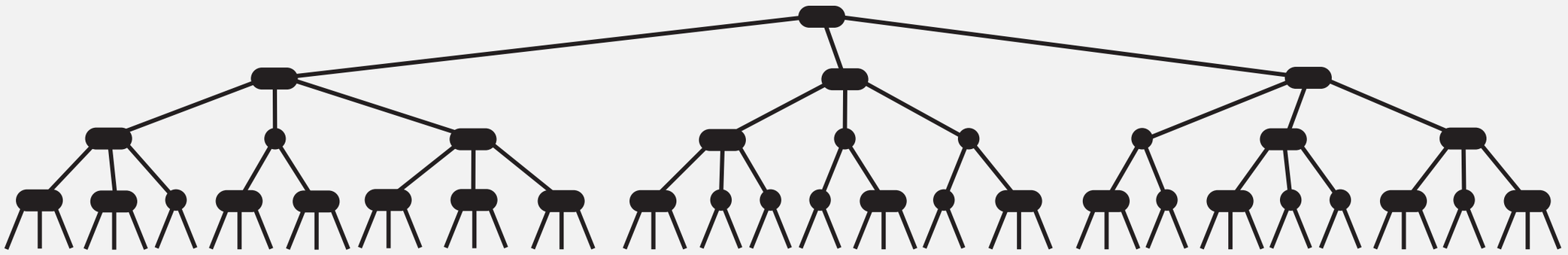


Tree height.

- Worst case:
- Best case:

2-3 tree: performance

Perfect balance. Every path from root to null link has same length.



Tree height.

- Worst case: $\lg N$. [all 2-nodes]
- Best case: $\log_3 N \approx .631 \lg N$. [all 3-nodes]
- Between 12 and 20 for a million nodes.
- Between 18 and 30 for a billion nodes.

Bottom line. Guaranteed logarithmic performance for search and insert.

2-3 tree: implementation?

Direct implementation is complicated, because:

- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

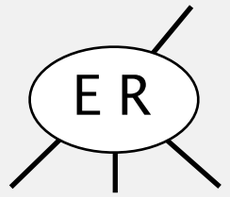
fantasy code

```
public void put(Key key, Value val)
{
   Node x = root;
   while (x.getTheCorrectChild(key) != null)
   {
      x = x.getTheCorrectChildKey();
      if (x.is4Node()) x.split();
   }
   if      (x.is2Node()) x.make3Node(key, val);
   else if (x.is3Node()) x.make4Node(key, val);
}
```

Bottom line. Could do it, but there's a better way.

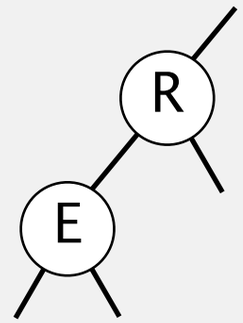
How to implement 2-3 trees with binary trees?

Challenge. How to represent a 3 node?



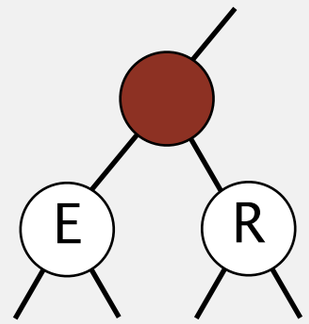
Approach 1. Regular BST.

- No way to tell a 3-node from a 2-node.
- Cannot map from BST back to 2-3 tree.



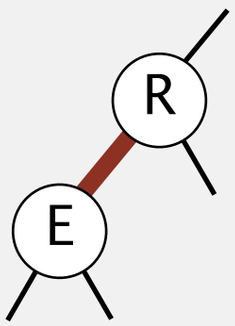
Approach 2. Regular BST with red "glue" nodes.

- Wastes space, wasted link.
- Code probably messy.



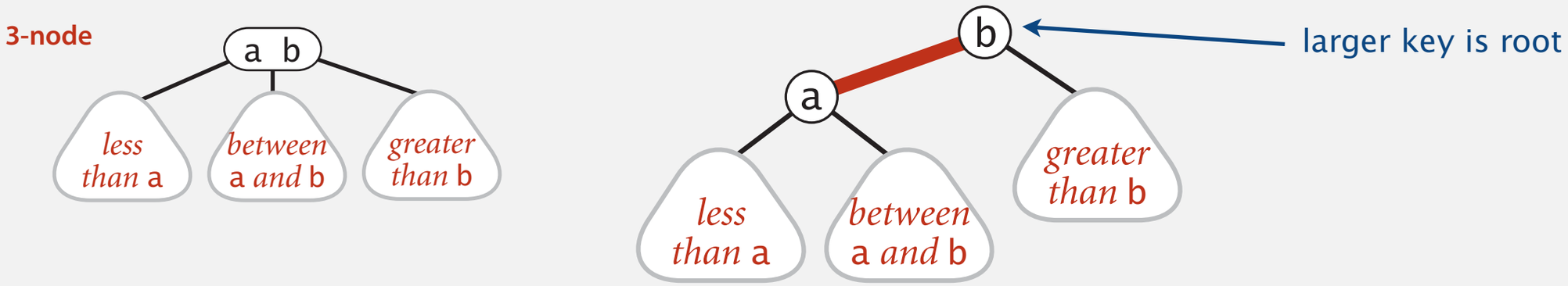
Approach 3. Regular BST with red "glue" links.

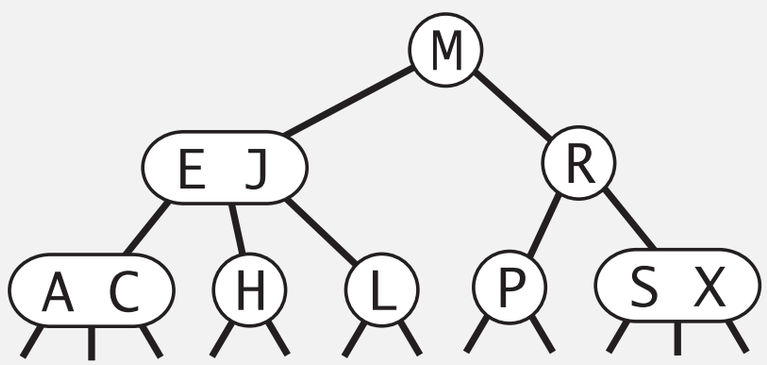
- Widely used in practice.
- Arbitrary restriction: red links lean left.



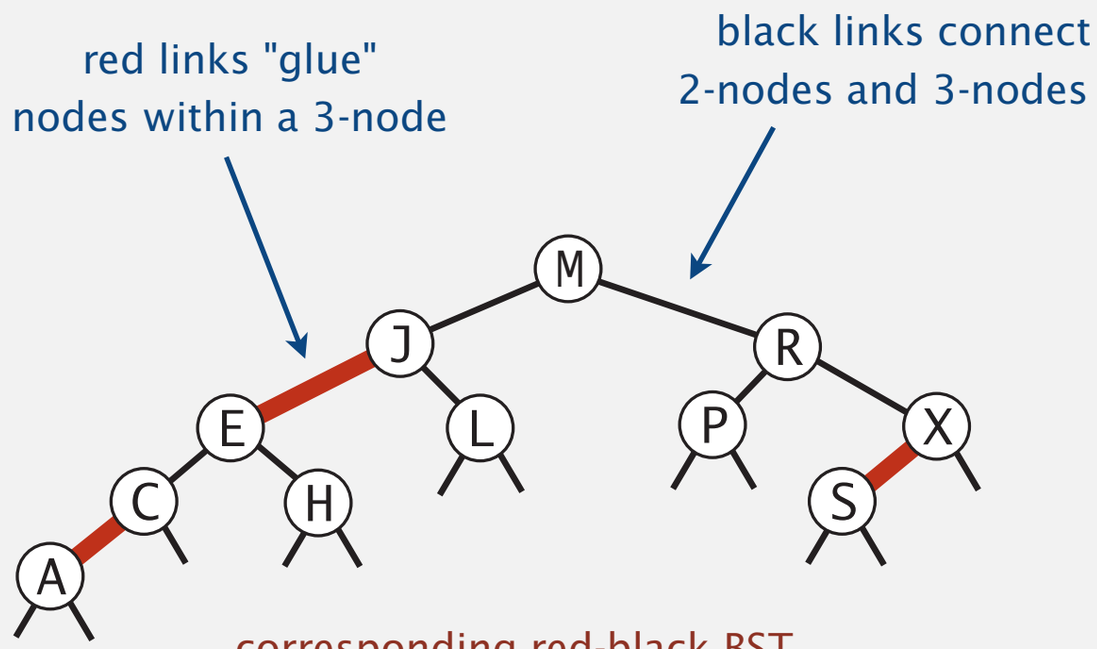
Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

1. Represent 2–3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3–nodes.





2-3 tree

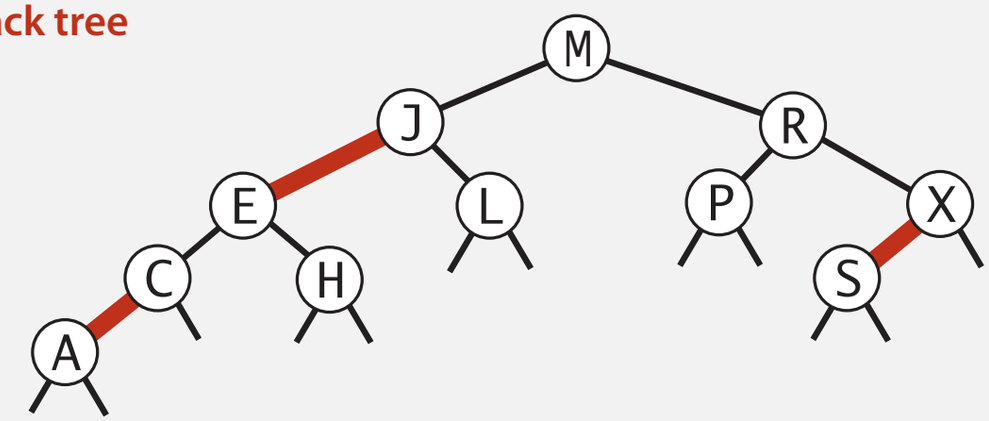


corresponding red-black BST

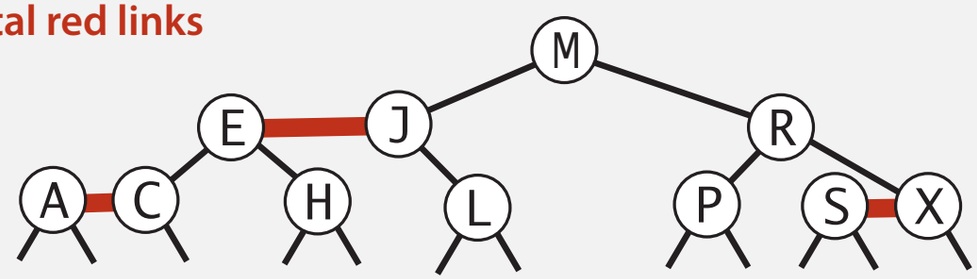
Left-leaning red-black BSTs: 1-1 correspondence with 2-3 trees

Key property. 1–1 correspondence between 2–3 and LLRB.

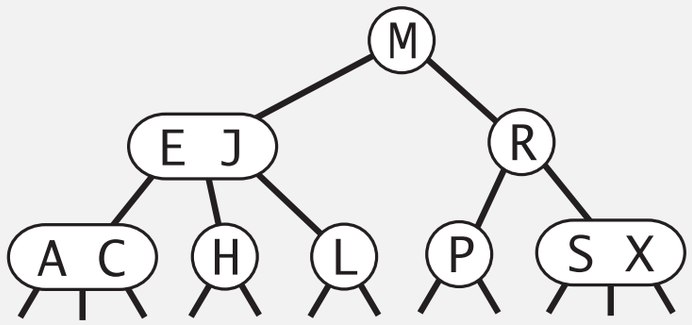
red–black tree



horizontal red links



2-3 tree

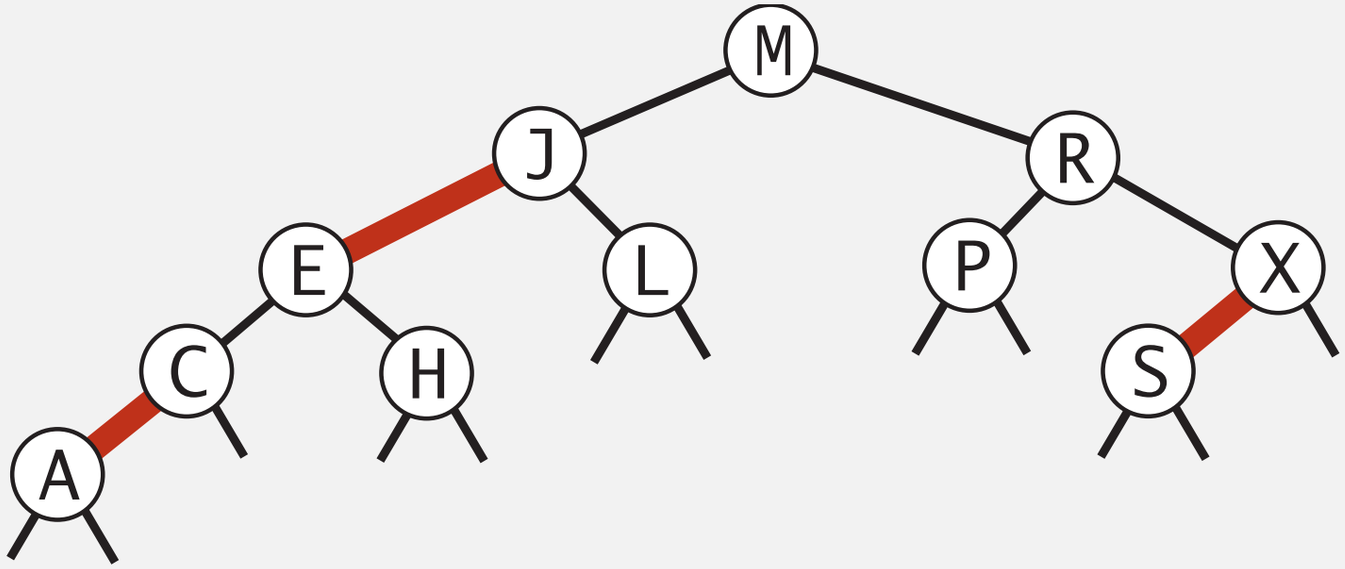


An equivalent definition

A BST such that:

- No node has two red links connected to it.
- Every path from root to null link has the same number of black links.
- Red links lean left.

"perfect black balance"

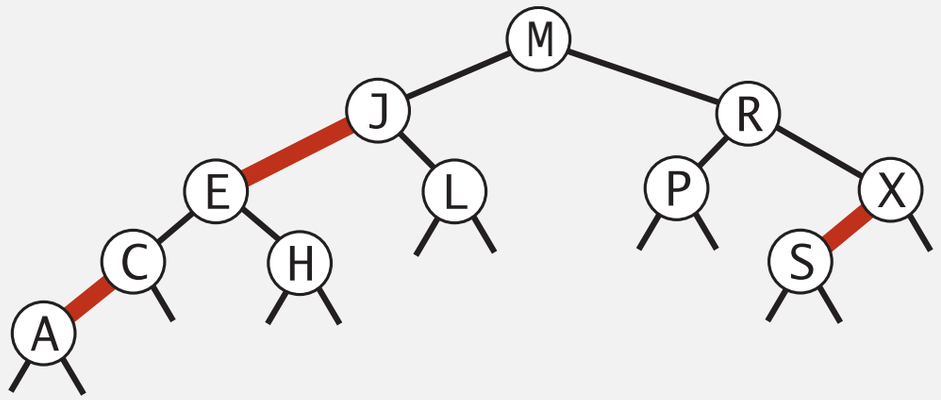


Search implementation for red-black BSTs

Observation. Search is the same as for elementary BST (ignore color).

but runs faster
because of better balance

```
public Val get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if      (cmp  < 0) x = x.left;
      else if (cmp  > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```



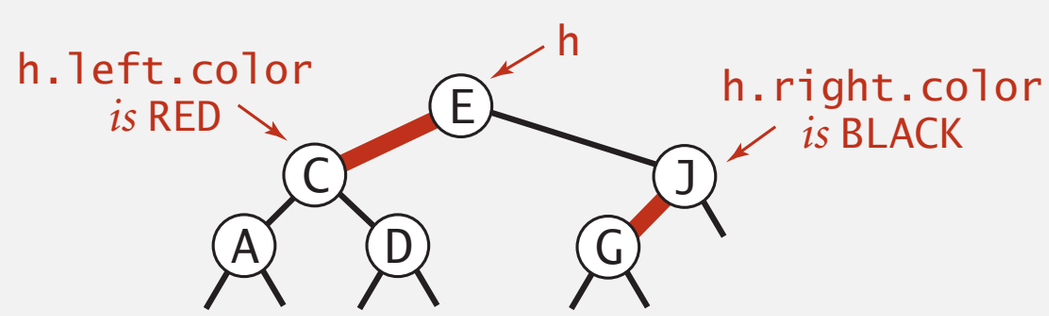
Remark. Most other ops (e.g., floor, iteration, selection) are also identical.

Red-black BST representation

Each node is pointed to by precisely one link (from its parent) ⇒
can encode color of links in nodes.

```
private static final boolean RED   = true;
private static final boolean BLACK = false;

private class Node
{
   Key key;
   Value val;
   Node left, right;
   boolean color;   // color of parent link
}

private boolean isRed(Node x)
{
   if (x == null) return false;
   return x.color == RED;
}
```

null links are black



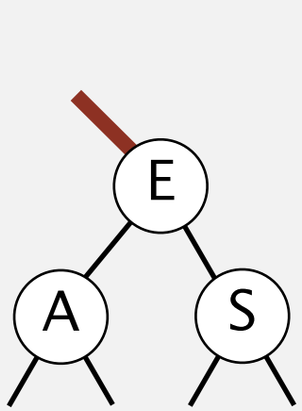
Insertion in a LLRB tree: overview

Basic strategy. Maintain 1-1 correspondence with 2-3 trees.

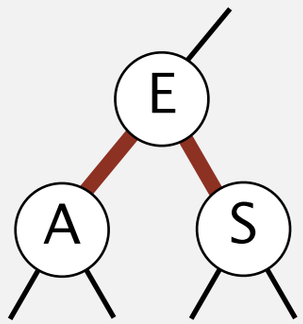
During internal operations, maintain:

- Symmetric order.
- Perfect black balance.

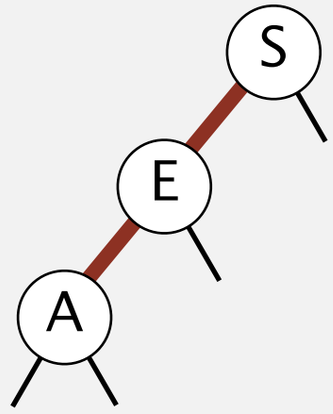
  [but not necessarily color invariants]



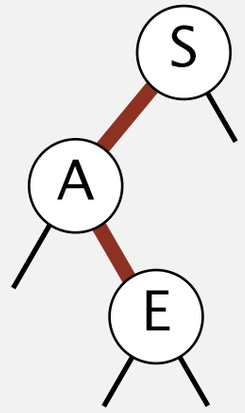
right-leaning red link



two red children (a temporary 4-node)



left-left red (a temporary 4-node)

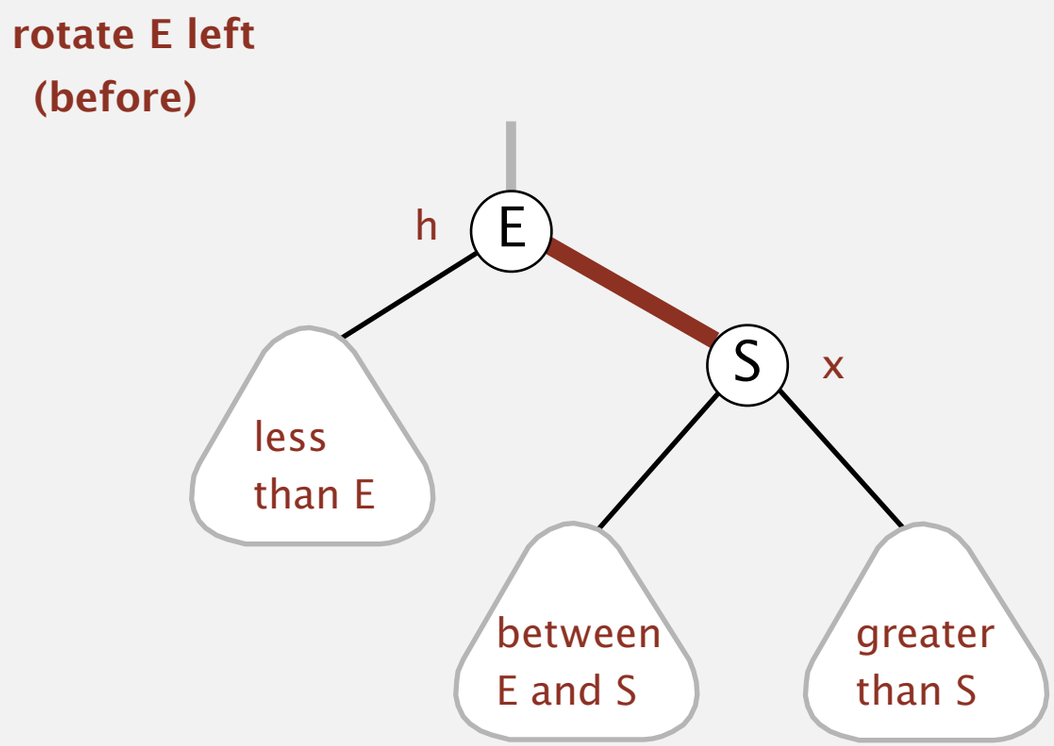


left-right red (a temporary 4-node)

How? Apply elementary red-black BST operations: rotation and color flip.

Elementary red-black BST operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

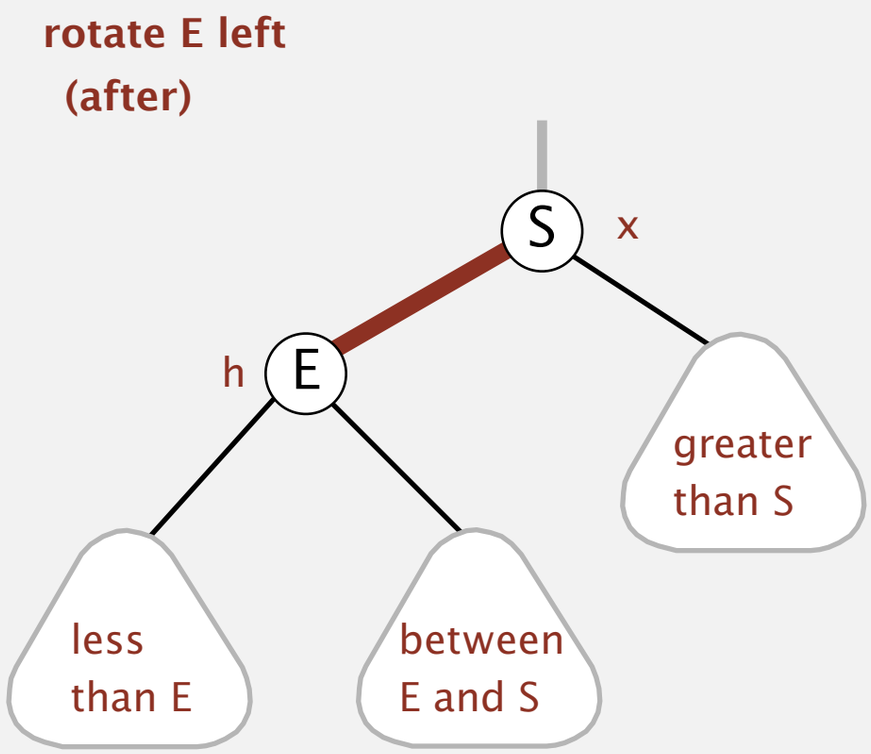



```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

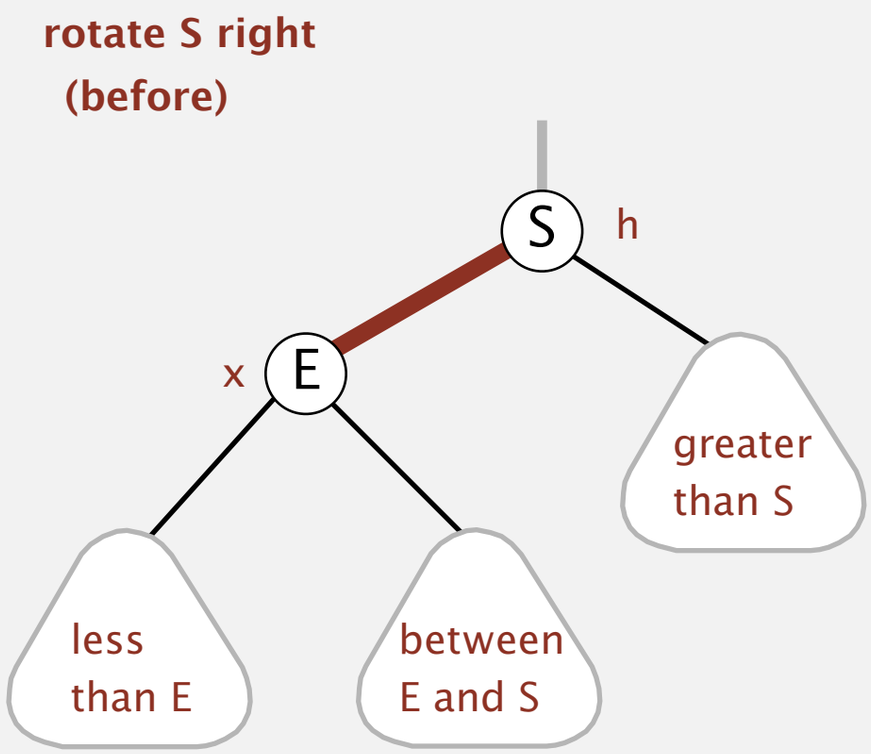



```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.

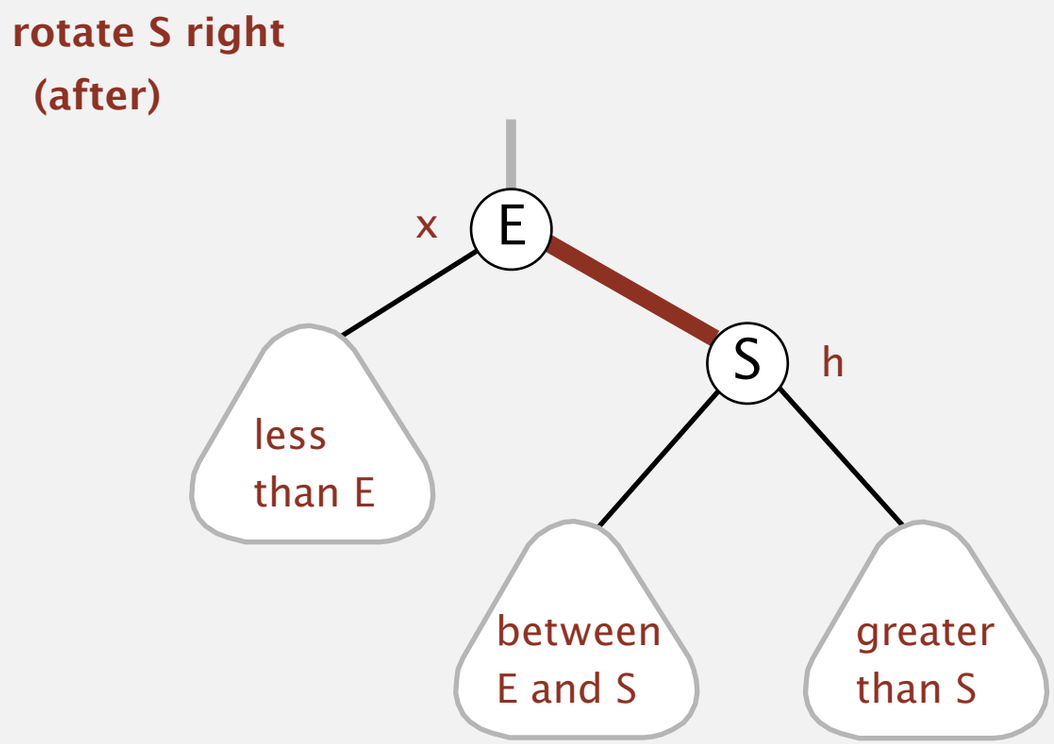



```
private Node rotateRight(Node h)
{
   assert isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.

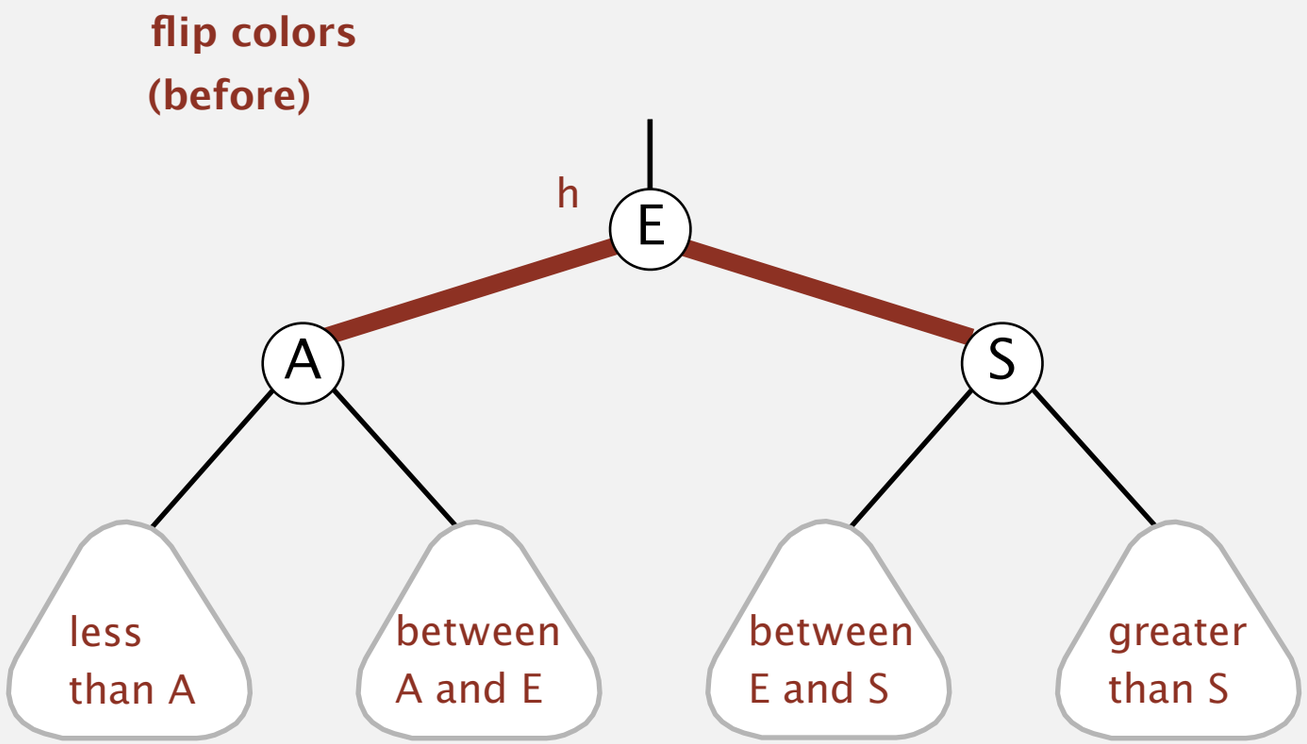



```
private Node rotateRight(Node h)
{
   assert isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Color flip. Recolor to split a (temporary) 4-node.

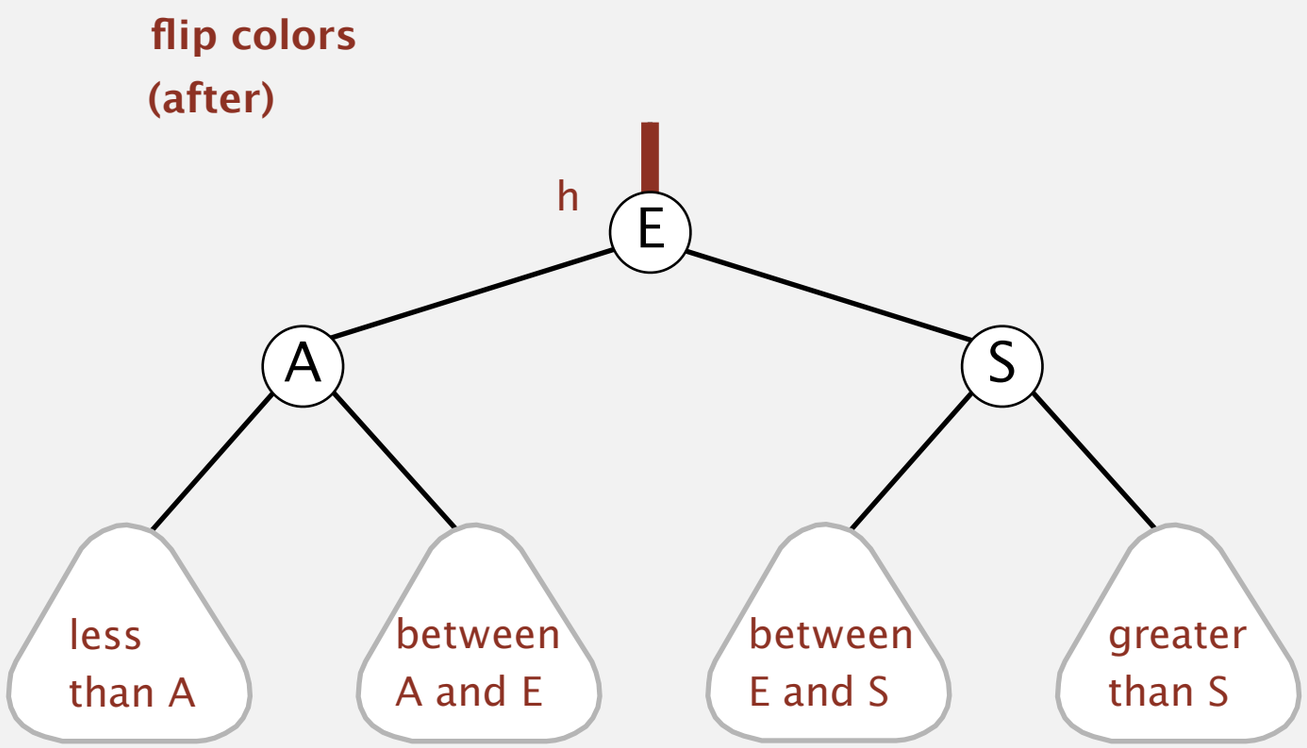



```
private void flipColors(Node h)
{
   assert !isRed(h);
   assert isRed(h.left);
   assert isRed(h.right);
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Color flip. Recolor to split a (temporary) 4-node.

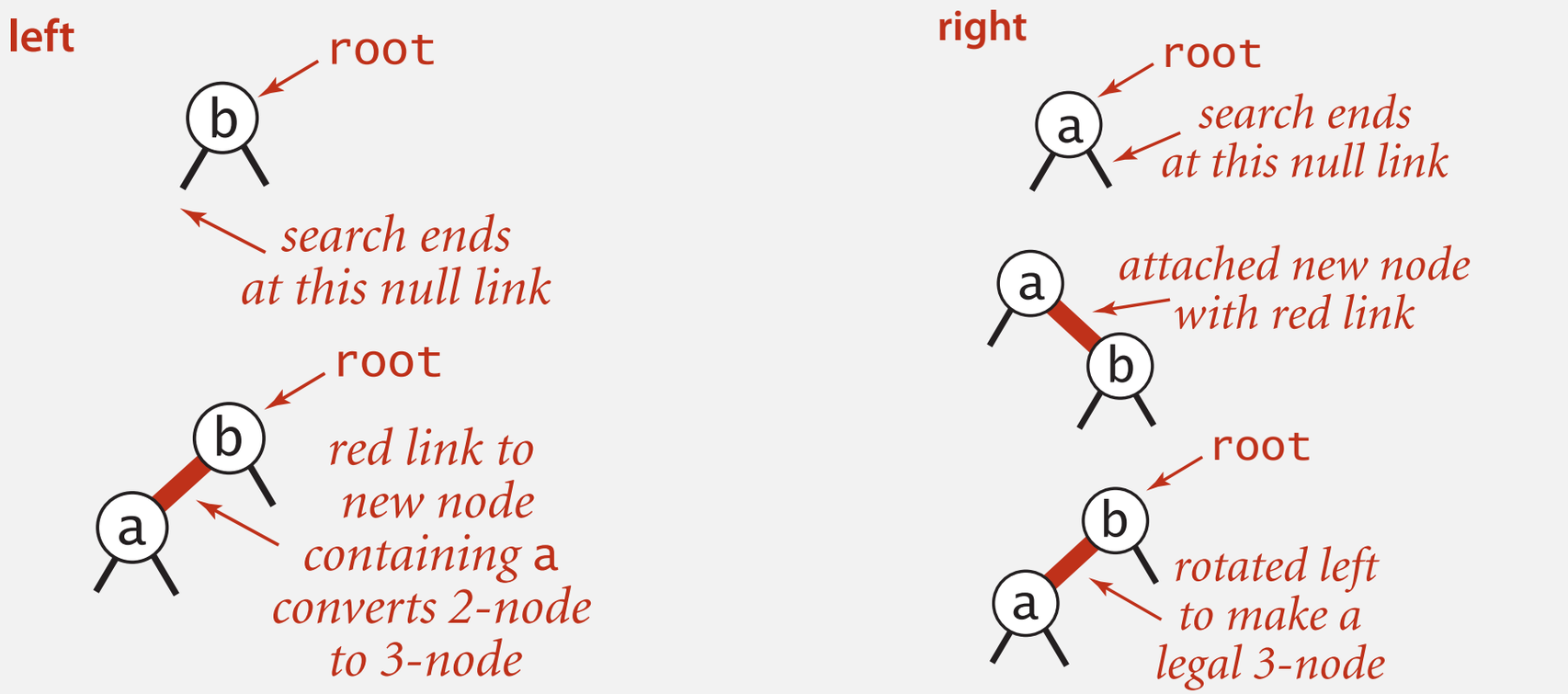



```
private void flipColors(Node h)
{
   assert !isRed(h);
   assert isRed(h.left);
   assert isRed(h.right);
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Insertion in a LLRB tree

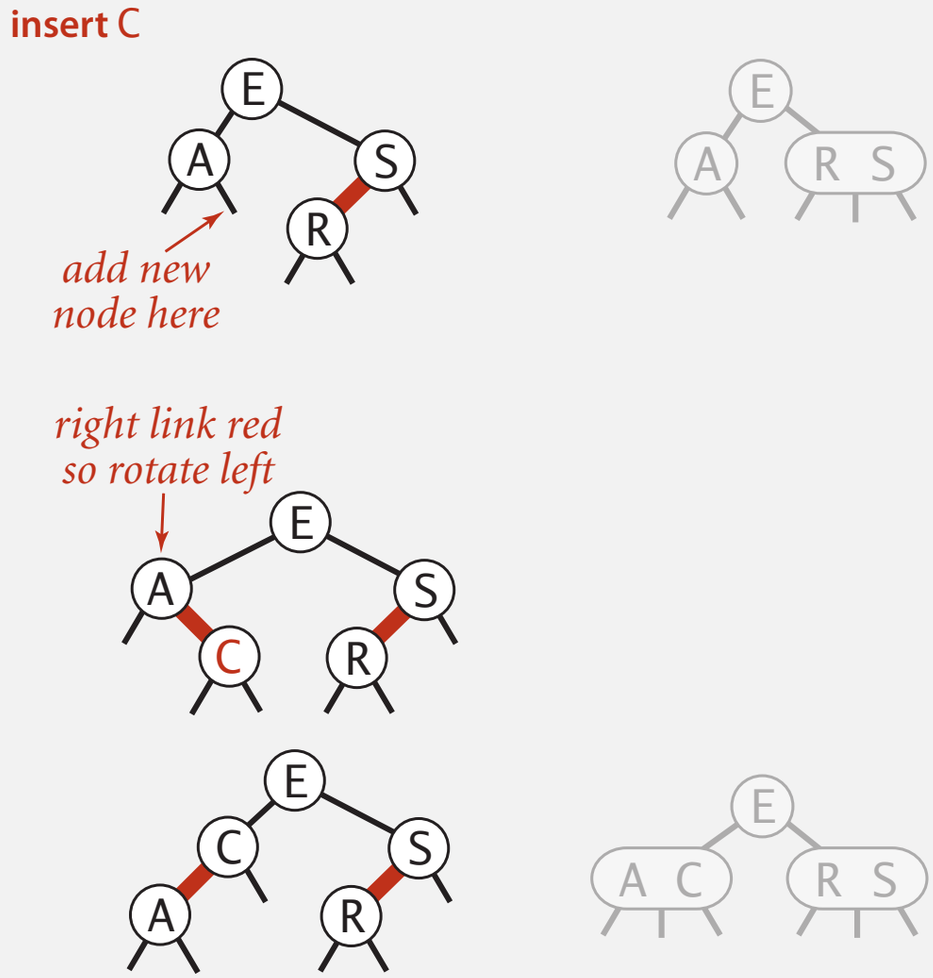
Warmup 1. Insert into a tree with exactly 1 node.



Insertion in a LLRB tree

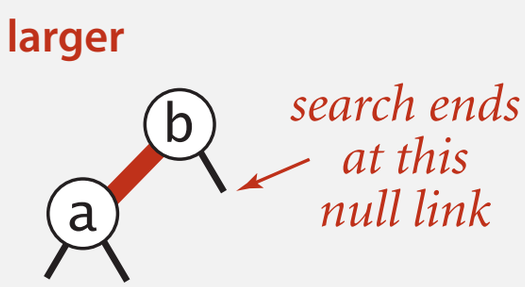
Case 1. Insert into a 2-node at the bottom.

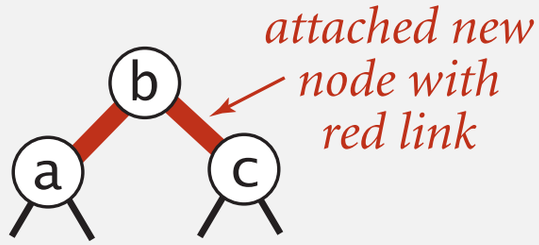
- Do standard BST insert; color new link red. ← to maintain symmetric order and perfect black balance
- If new red link is a right link, rotate left. ← to fix color invariants

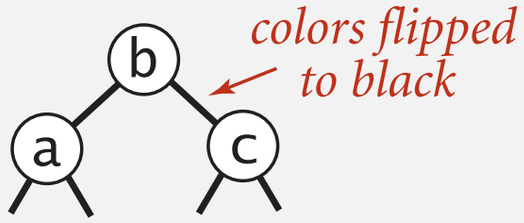


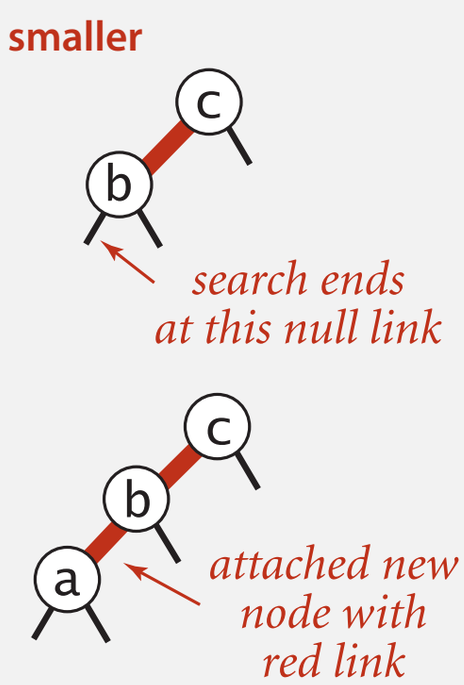
Insertion in a LLRB tree

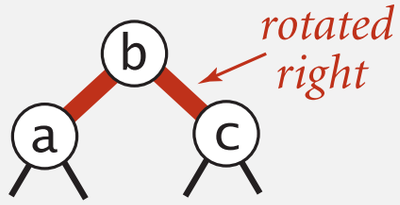
Warmup 2. Insert into a tree with exactly 2 nodes.

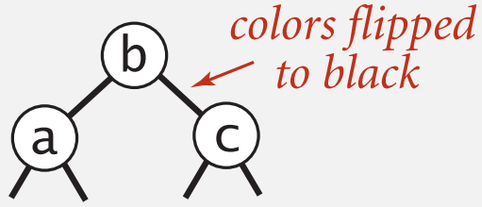


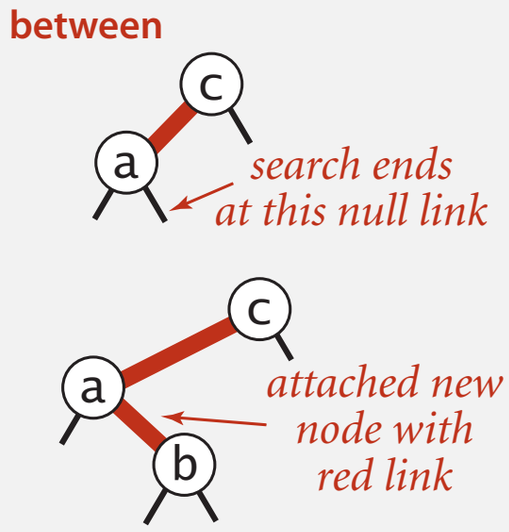


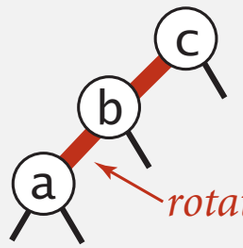


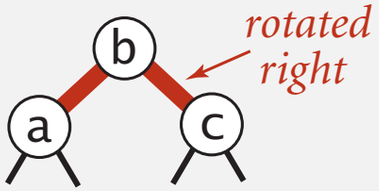


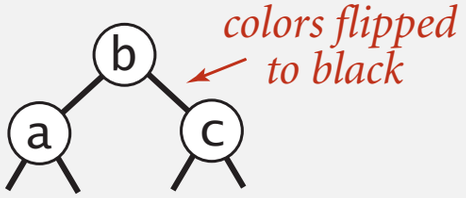








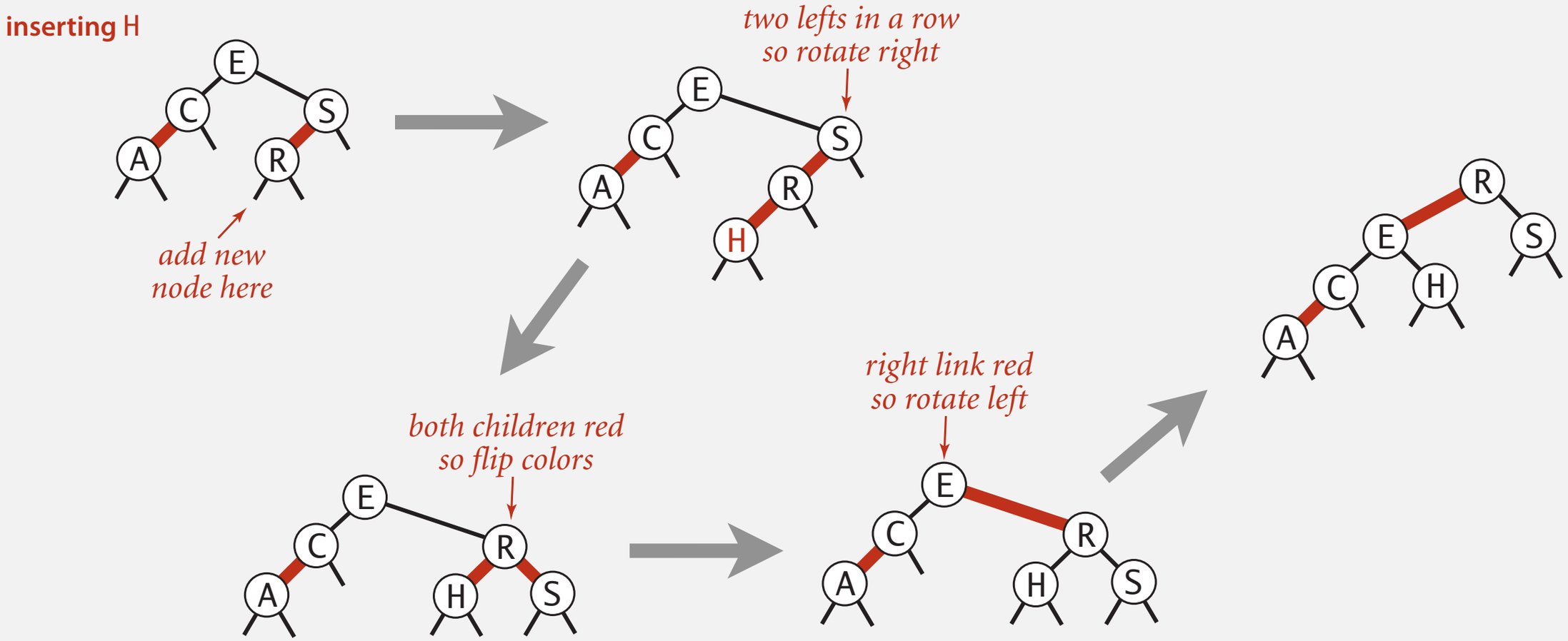




Insertion in a LLRB tree

Case 2. Insert into a 3-node at the bottom.

- Do standard BST insert; color new link red. ← to maintain symmetric order and perfect black balance
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed). ← to fix color invariants



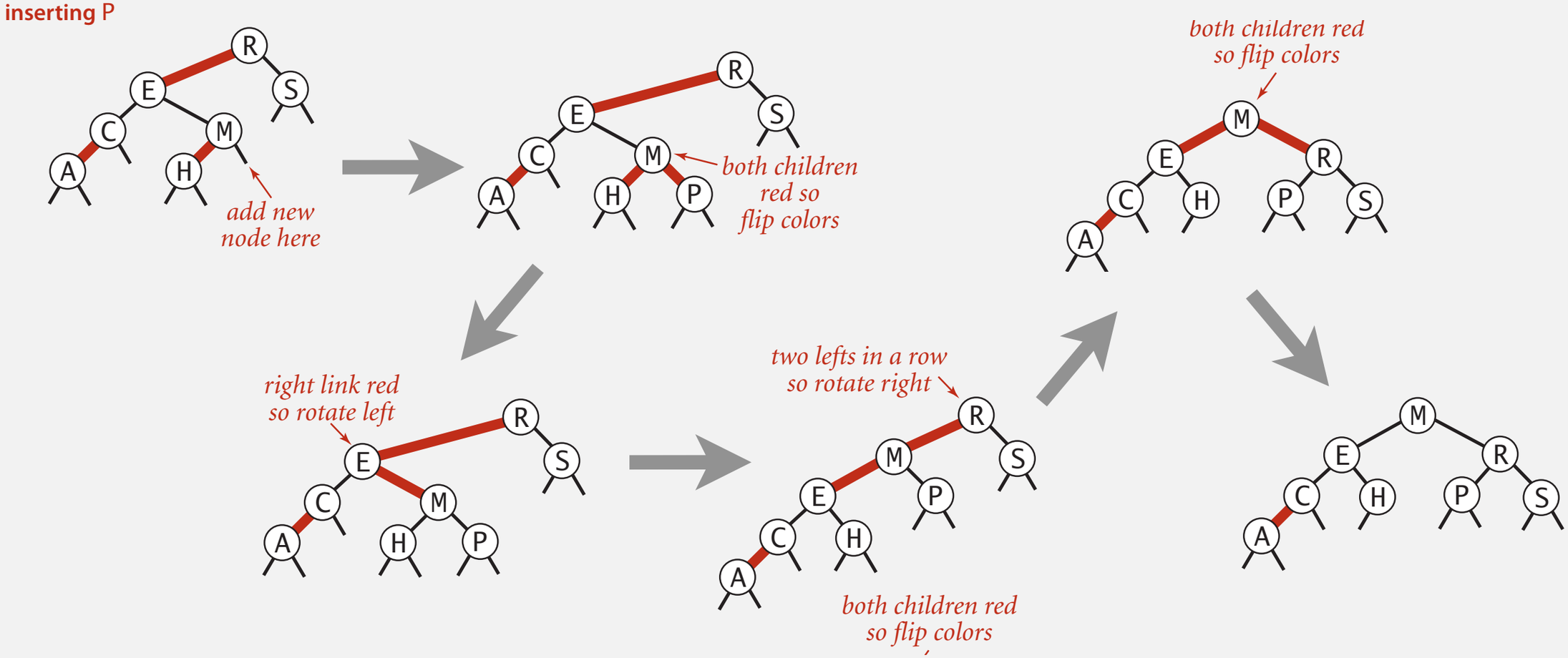
Insertion in a LLRB tree: passing red links up the tree

Case 2. Insert into a 3-node at the bottom.

- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).
- Repeat case 1 or case 2 up the tree (if needed).

to maintain symmetric order and perfect black balance

to fix color invariants

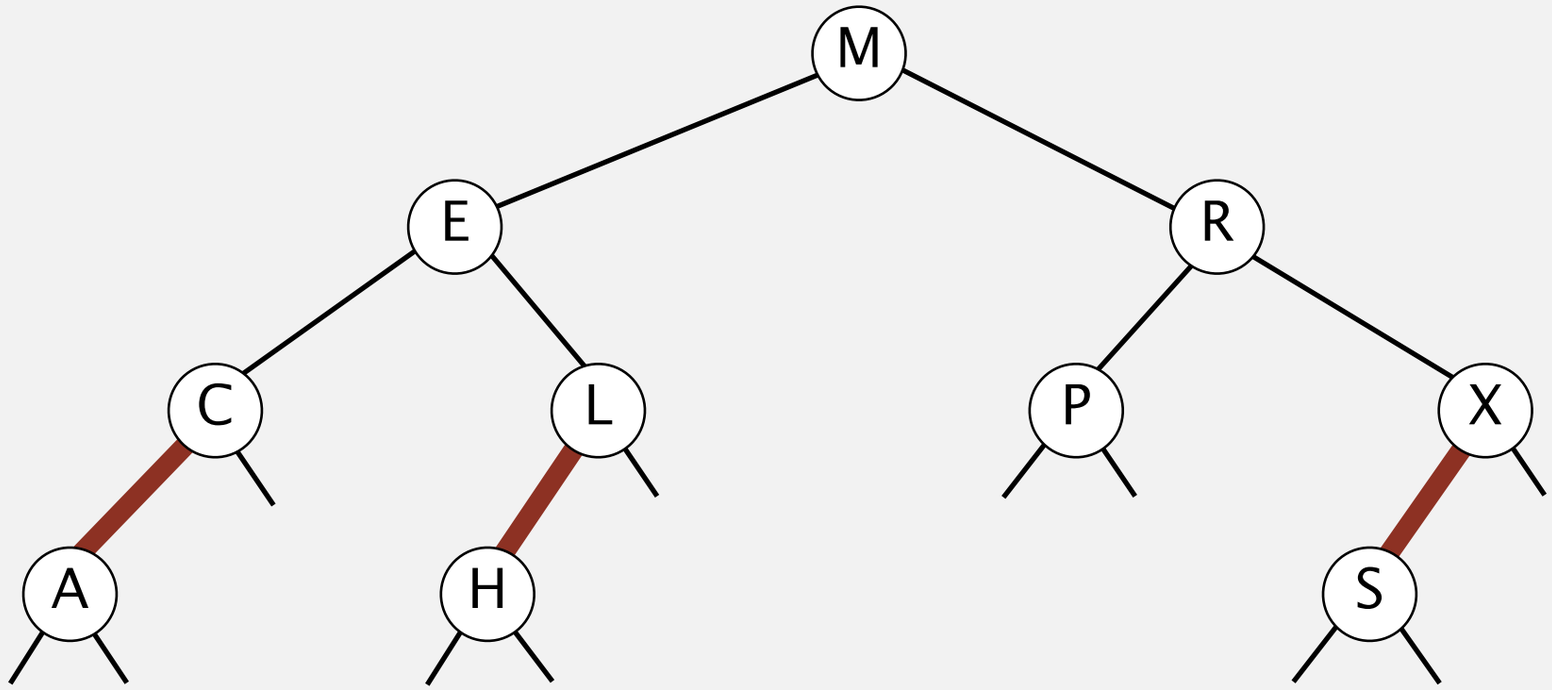


Red-black BST construction demo

insert S

Red-black BST construction demo

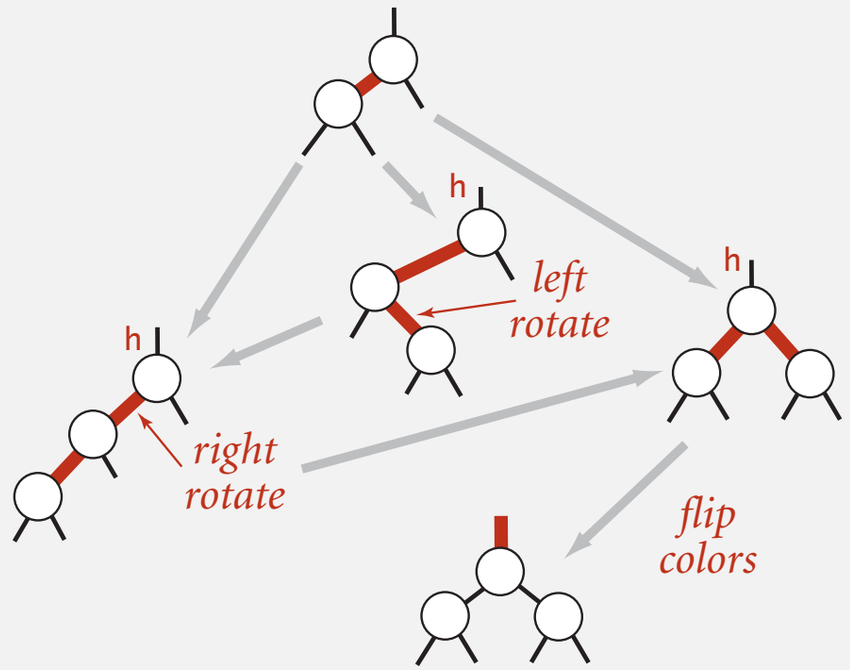
red-black BST



Insertion in a LLRB tree: Java implementation

Same code for all cases.

- Right child red, left child black: rotate left.
- Left child, left-left grandchild red: rotate right.
- Both children red: flip colors.




```
private Node put(Node h, Key key, Value val)
{
   if (h == null) return new Node(key, val, RED);
   int cmp = key.compareTo(h.key);
   if      (cmp  < 0) h.left  = put(h.left,  key, val);
   else if (cmp  > 0) h.right = put(h.right, key, val);
   else if (cmp == 0) h.val = val;

   if (isRed(h.right) && !isRed(h.left))     h = rotateLeft(h);
   if (isRed(h.left)  && isRed(h.left.left)) h = rotateRight(h);
   if (isRed(h.left)  && isRed(h.right))     flipColors(h);

   return h;
}
```

insert at bottom (and color it red)

lean left

balance 4-node

split 4-node

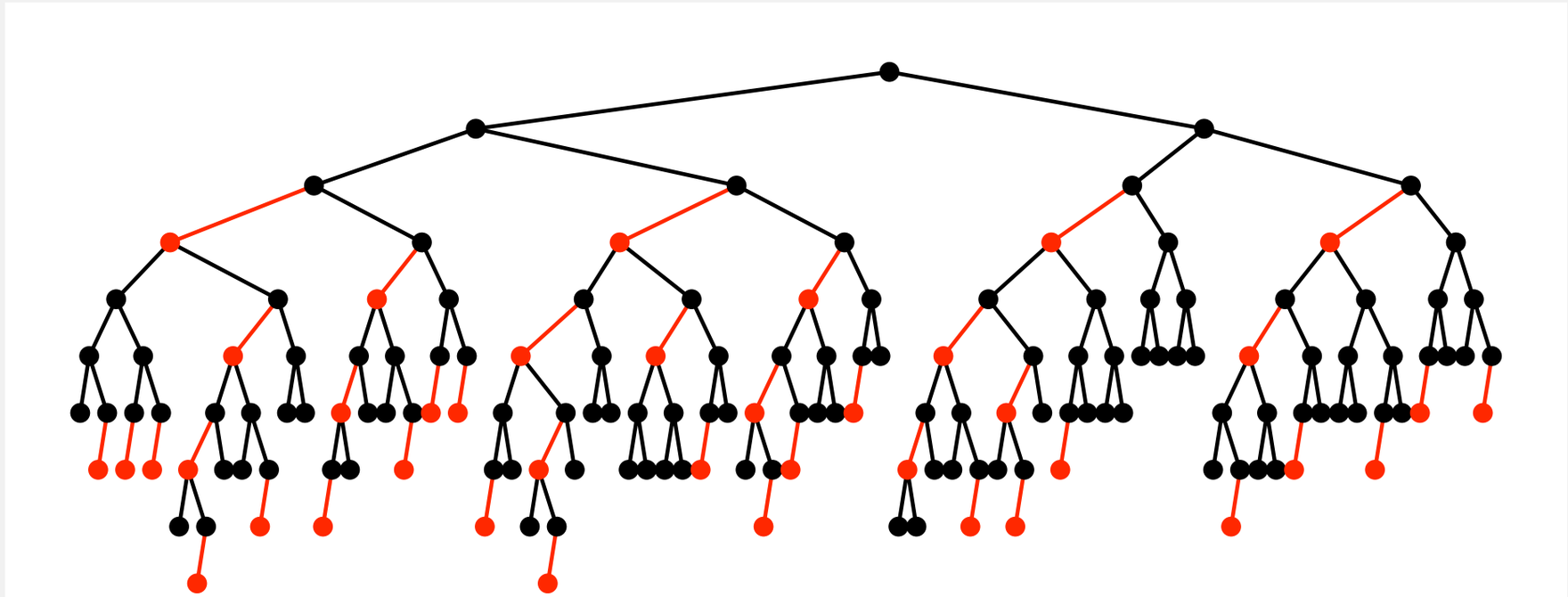
only a few extra lines of code provides near-perfect balance

Balance in LLRB trees

Proposition. Height of tree is $\leq 2 \lg N$ in the worst case.

Pf.

- Black height = height of corresponding 2-3 tree $\leq \lg N$.
- Never two red links in-a-row.



Property. Height of tree is ~ $1.0 \lg N$ in typical applications.

ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N	N	N		`equals()`
binary search (ordered array)	$\log N$	N	N	$\log N$	N	N	✔	`compareTo()`
BST	N	N	N	$\log N$	$\log N$	$\sqrt{N}$	✔	`compareTo()`
2-3 tree	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	✔	`compareTo()`
red-black BST	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	✔	`compareTo()`

hidden constant c is small
(at most 2 lg N compares)