Red Black Trees

Observations

- There are many variations of balanced binary trees. The prominent among them are Red Black trees, B-trees (again of many kinds), weak AVL trees etc.; they all share the same property that (1) height of the tree is $O(\log_2 n)$, where n is the number of nodes and (2) the worst-case time for each of the operations of search, insert, delete is $O(\log_2 n)$.
- Limitations of Balanced Search Trees: Balanced search trees require storing an extra piece of information per node. Their worst-case, average-case, and best-case performance are essentially identical. We do not win when easy inputs occur – would be nice if the second access to the same piece of data was cheaper than the first. The 90-10 rule – empirical studies suggest that in practice 90% of the accesses are to 10% of the data items; it would be nice to get easy wins for the 90% case.
- A **red-black** tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often called the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree.

Why Red-Black Trees?

- Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations. The height of a Red-Black tree is always O(log n) where n is the number of nodes in the tree.
- **Most of the BST operations** (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations. The height of a Red-Black tree is always O(log n) where n is the number of nodes in the tree.
- **Comparison with AVL Tree**: The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion.
  - If the application involves frequent insertions and deletions, then Red-Black trees should be preferred.
  - And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

Rules of Red Black Trees

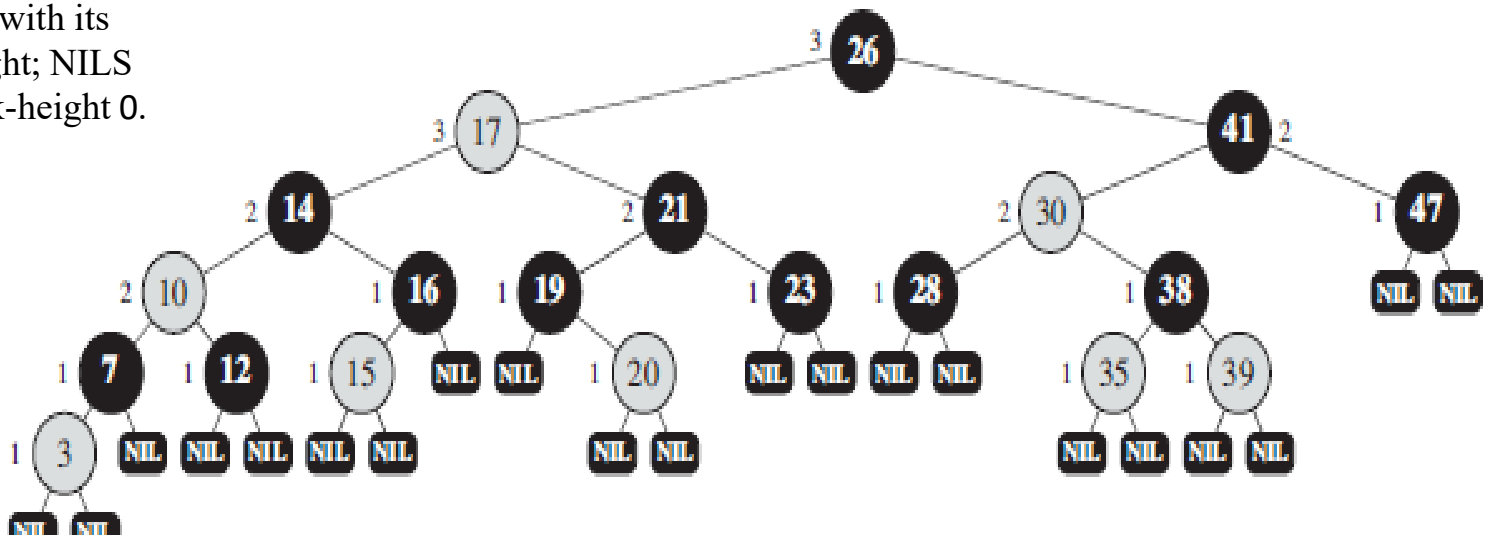
- A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK.
- By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice that of any other, so that the tree is approximately balanced.
- Each node of the tree contains the attributes **color, key, left, right, and p [parent].** If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree. A red-black tree satisfies the following **red-black properties**:
  - 1. Every node is either red or black.
  - 2. The root is black.
  - 3. Every leaf (NIL) is black.
  - 4. If a node is red, then both its children are black.
  - 5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

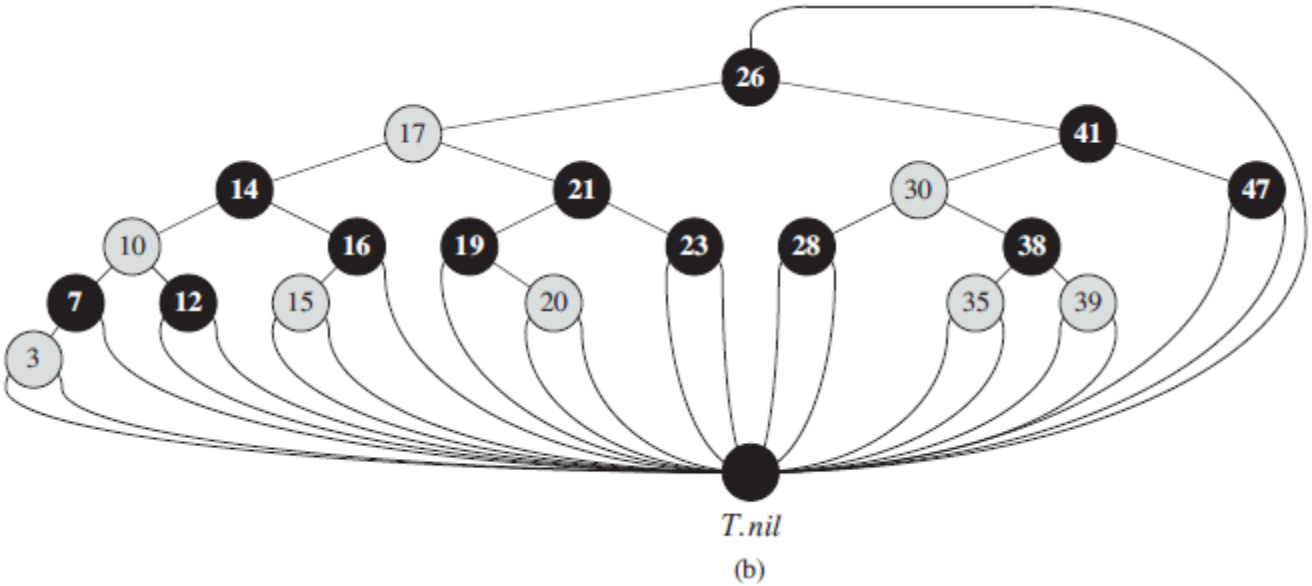
Convention of Drawing Pictures

- As a matter of convenience in dealing with boundary conditions in red-black tree code, we use a single **sentinel** to represent NIL. For a red-black tree T , the **sentinel T.nil** is an object with the same attributes as an ordinary node in the tree. Its color attribute is BLACK, and its other attributes—p, left, right, and key—can take on arbitrary values. As the second picture shows [see next page], all pointers to NIL are replaced by pointers to the sentinel T.nil.
- We use the sentinel so that we can treat a NIL child of a node x as an ordinary node whose parent is x. [Although we instead could add a distinct sentinel node for each NIL in the tree, so that the parent of each NIL is well defined, that approach would waste space.]
  - We use the one sentinel T.nil to represent all the NILs—all leaves and the root's parent.
  - The values of the attributes p, left, right, and key of the sentinel are immaterial, although we may set them during a procedure for our convenience.
  - We generally confine our interest to the internal nodes of a red-black tree, since they hold the key values. We omit the leaves when we draw red-black trees,

Example Red Black Tree

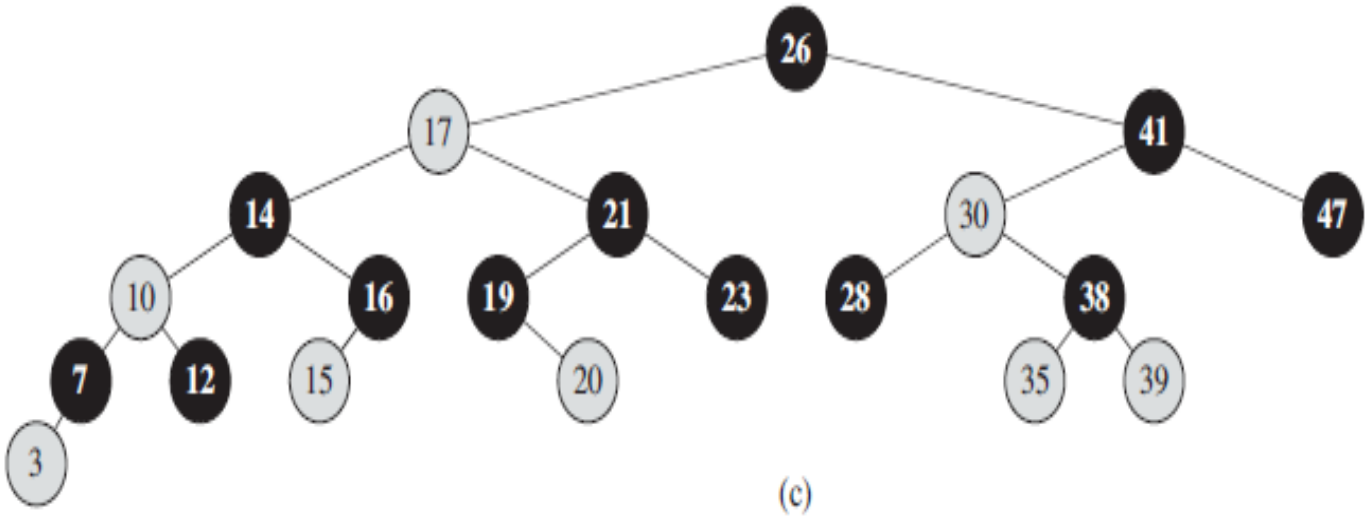
Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILS have black-height 0.





The same red-black tree but with each NIL replaced by the single sentinel T:nil, which is always black, and with black-heights omitted

Same Example Red Black Tree



The same red-black tree but with leaves and the root's parent omitted entirely.

Red Black Tree Properties

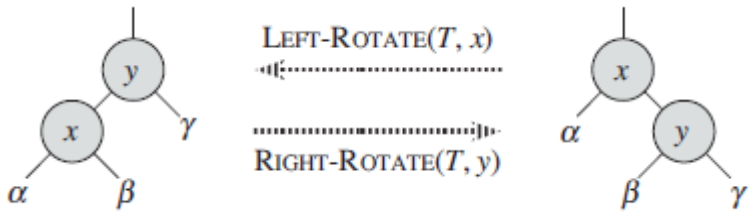
- **Theorem**: A red-black tree with n internal nodes has height at most 2 lg (n + 1)
- **Proof:** We need to show that the subtree rooted at any node x contains at least $2^{bh(x) - 1}$ internal nodes. We prove this claim by induction on the height of x.

  If the height of x is 0, then x must be a leaf (T.*nil*), and the subtree rooted at x indeed contains at least $2^{bh(x)} – 1 = 2^0 – 1 = 0$ internal nodes. For the inductive step, consider a node x that has positive height and is an internal node with two children. Each child has a black-height of either bh(x) or bh(x) – 1, depending on whether its color is red or black. Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)} – 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{bh(x)-1} – 1) + (2^{bh(x)-1} – 1) + 1 = (2^{bh(x)-1} – 1)$ internal nodes, which proves the claim.

To complete the proof of the theorem, let h be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least h/2; thus, n) ≥ $(2^{bh(x)-1} – 1)$. Simplifying, we get lg (n + 1) ≥ h/2 or h ≤ 2 lg(n+1).

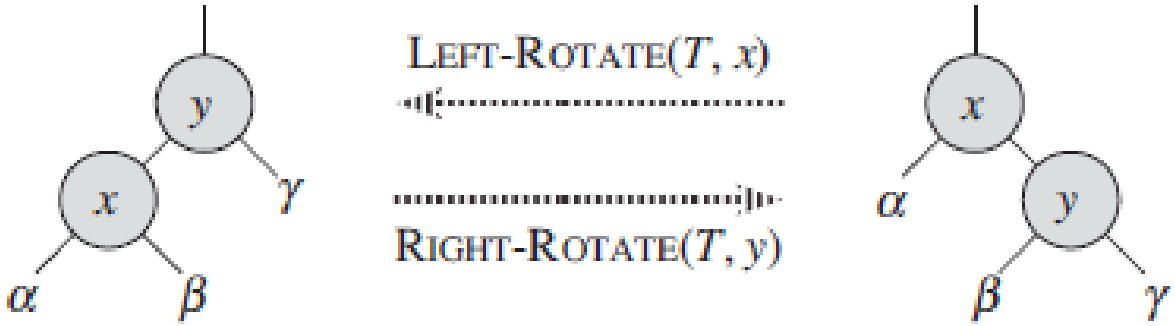
Operations on Red Black Trees

- Now, we can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in O(lg n) time on red-black trees, since each can run in O(h) time on a binary search tree of height h and any red-black tree on n nodes is a binary search tree with height O(lg n).
- The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red black tree with n keys, take O(lg n) time. Because they modify the tree, the result may violate the red-black properties.
- To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.
- We change the pointer structure through ***rotation***, which is a local operation in a search tree that preserves the binary-search-tree property. There are two kinds of rotations: left rotations and right rotations.
- When we do a left rotation on a node x, we assume that its right child y is not T.*nil*; x may be any node in the tree whose right child is not T.*nil*. The left rotation "pivots" around the link from x to y. It makes y the new root of the subtree, with x as y's left child and y's left child as x's right child.



The pseudocode for LEFT-ROTATE assumes that x.right ≠ T.nil and that the root's parent is T.nil.

Rotations Pictorially



The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers.

The inverse operation RIGHT-ROTATE.T; y/ transforms the configuration on the left into the configuration on the right. The letters α, β, γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede x.key, which precedes the keys in β, which precede y:key, which precedes the keys in γ.

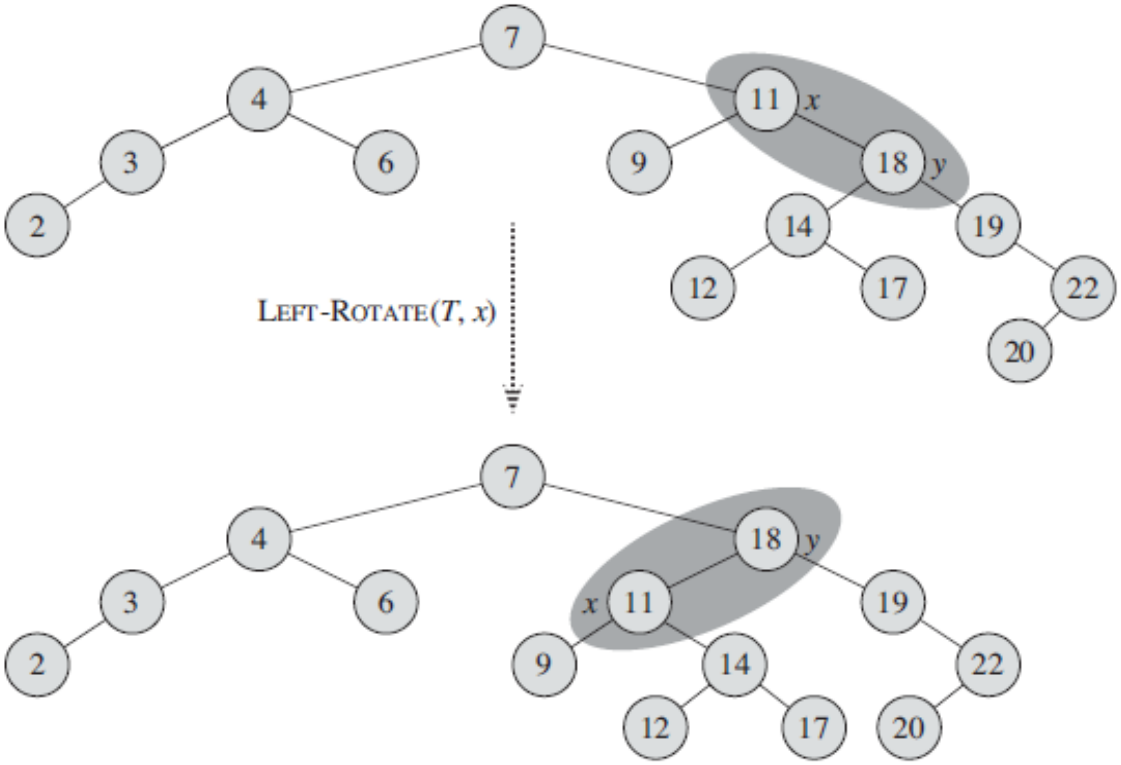
Note that in both trees, an in-order traversal yields **αxβyγ;** Why?

Left-Rotate

```
LEFT-ROTATE(T, x)
 1  y = x.right              // set y
 2  x.right = y.left         // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil
 4       y.left.p = x
 5  y.p = x.p                // link x's parent to y
 6  if x.p == T.nil
 7       T.root = y
 8  elseif x == x.p.left
 9       x.p.left = y
10  else x.p.right = y
11  y.left = x               // put x on y's left
12  x.p = y
```

The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers.

Example of Left Rotation



The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers.

Insertion

- We can insert a node into an n-node red-black tree in O(lg n) time.
- To do so, we use a slightly modified version of the TREE-INSERT (we did earlier) to insert node z into the tree T as if it were an ordinary binary search tree, and then we color z red.
- To guarantee that the red-black properties are preserved, we then call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations.
- The call RB-INSERT (T, z) inserts node z, whose key is assumed to have already been filled in, into the red-black tree T .

Insertion Pseudocode☆

```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == T.nil
10      T.root = z
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

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

The procedures TREE-INSERT and RB-INSERT differ in four ways. First, all instances of NIL in TREE-INSERT are replaced by T.nil. Second, we set z.left and z.right to T.nil in lines 14–15 of RB-INSERT, in order to maintain the proper tree structure. Third, we color ´ red in line 16. Fourth, because coloring z red may cause a violation of one of the red-black properties, we call RB-INSERT-FIXUP(T, z) in line 17 of RB-INSERT to restore the red-black properties.

Left Rotation

The left_rotate operation may be encoded:

```
left_rotate( Tree T, node x ) {
    node y;
    y = x->right;
    /* Turn y's left sub-tree into x's right sub-tree */
    x->right = y->left;
    if ( y->left != NULL )
        y->left->parent = x;
    /* y's new parent was x's parent */
    y->parent = x->parent;
    /* Set the parent to point to y instead of x */
    /* First see whether we're at the root */
    if ( x->parent == NULL ) T->root = y;
    else
        if ( x == (x->parent)->left )
            /* x was on the left of its parent */
            x->parent->left = y;
        else
            /* x must have been on the right */
            x->parent->right = y;
    /* Finally, put x on y's left */
    y->left = x;
    x->parent = y;     }
```

Node Structure:

```
struct t_red_black_node {
    enum { red, black } color;
    void *item;
    struct t_red_black_node *left,
                    *right,
                    *parent;
    }
```

Insertion

```
rb_insert( Tree T, node x ) {
   /* Insert in the tree in the usual way */
   tree_insert( T, x );
   /* Now restore the red-black property */
   x->color = red;
   while ( (x != T->root) && (x->parent->color == red) ) {
     if ( x->parent == x->parent->parent->left ) {
        /* If x's parent is a left, y is x's right 'uncle' */
        y = x->parent->parent->right;
        if ( y->color == red ) {
           /* case 1 - change the colours */
           x->parent->color = black;
           y->color = black;
           x->parent->parent->color = red;
           /* Move x up the tree */
           x = x->parent->parent;                }
        else { /* y is a black node */
           if ( x == x->parent->right ) {
              /* and x is to the right */
              /* case 2 - move x up and rotate */
              x = x->parent;
              left_rotate( T, x );               }
           /* case 3 */
              x->parent->color = black;
              x->parent->parent->color =
red;
              right_rotate( T, x->parent->
parent );
              }
           }
        else {
           /* repeat the "if" part with right
and left
              exchanged */
           }
        }
   /* color the root black */
   T->root->color = black;
   }
```

AA Trees, Treaps

Observations

- There are many variations of balanced binary trees. The prominent among them are Red Black trees, B-trees (again of many kinds), weak AVL trees etc.; they all share the same property that (1) height of the tree is $O(\log_2 n)$, where n is the number of nodes and (2) the worst-case time for each of the operations of search, insert, delete is $O(\log_2 n)$.
- Limitations of Balanced Search Trees: Balanced search trees require storing an extra piece of information per node. Their worst-case, average-case, and best-case performance are essentially identical. We do not win when easy inputs occur – would be nice if the second access to the same piece of data was cheaper than the first. The 90-10 rule – empirical studies suggest that in practice 90% of the accesses are to 10% of the data items; it would be nice to get easy wins for the 90% case.
- An **AA** tree is another kind of self-balancing binary search tree, a variation of Red Black trees, where there are fewer rotation cases so easier to code, especially deletions (eliminates about half of the rotation cases). [The implementation and number of rotation cases in Red-Black Trees is relatively complex]. AA-trees still have O(log n) searches in the worst-case, although they are slightly less efficient empirically.

AA-Tree Ordering Properties

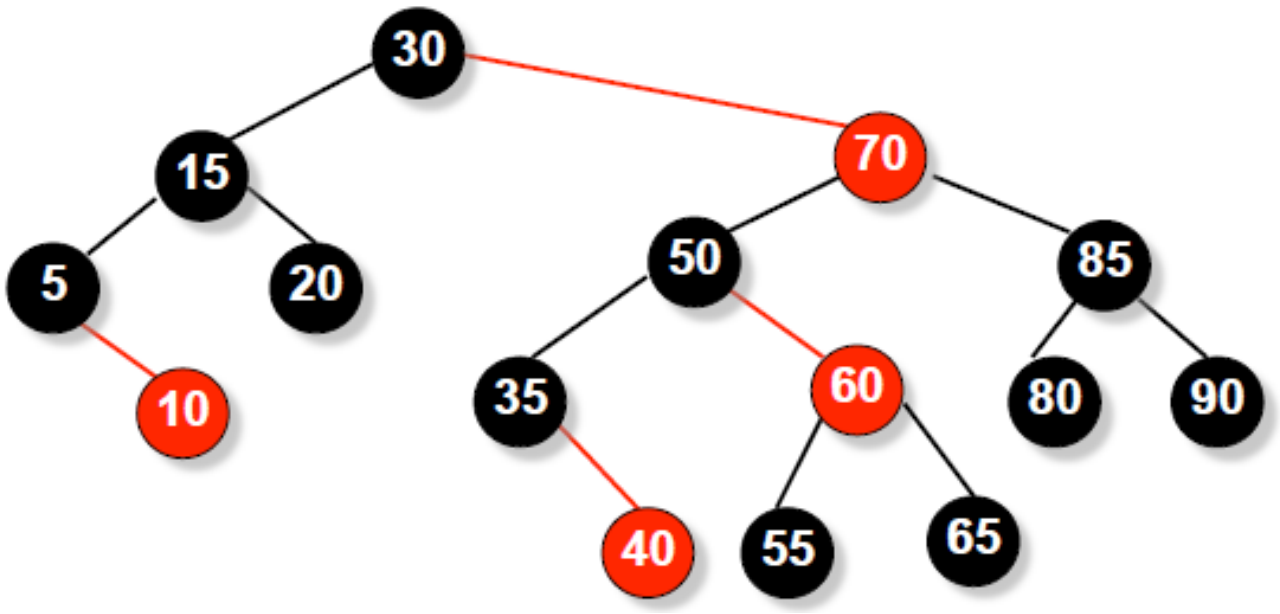
An AA-Tree is a binary search tree with all the ordering properties of a red-black tree:

1. Every node is colored either red or black!
2. The root is black
3. External nodes are black
4. If a node is red, its children must be black
5. All paths from any node to a descendent leaf must contain the same number of black nodes (black-height, not including the node itself)

   PLUS!

6. Left children may not be red.

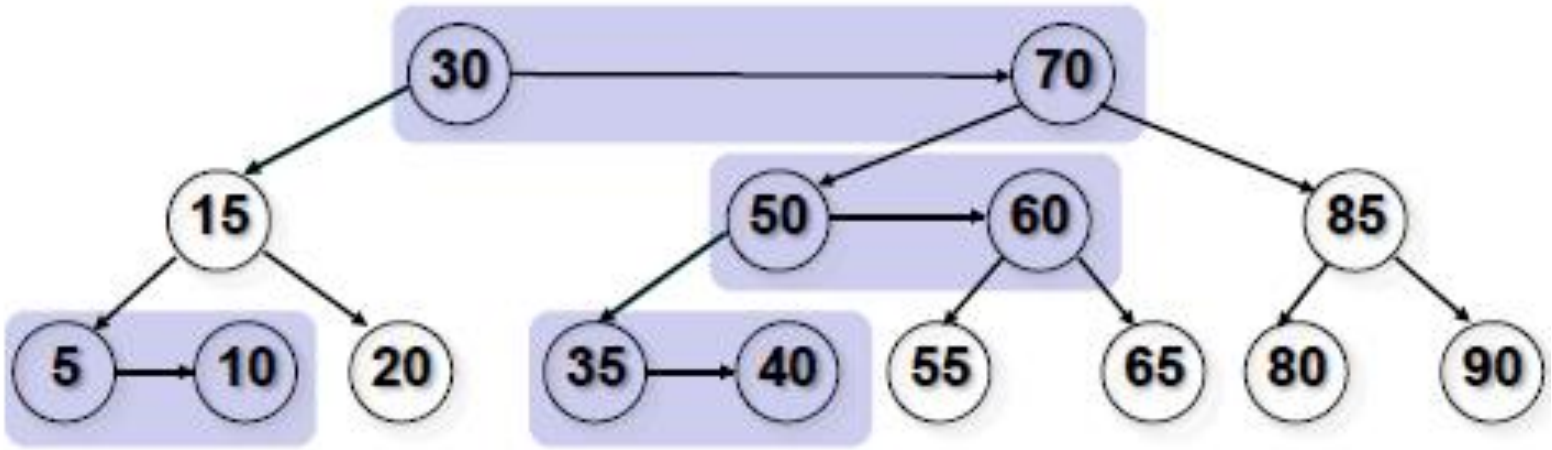
An AA-Tree Example



- No left red children!! Half of red-black tree rotation cases eliminated!!

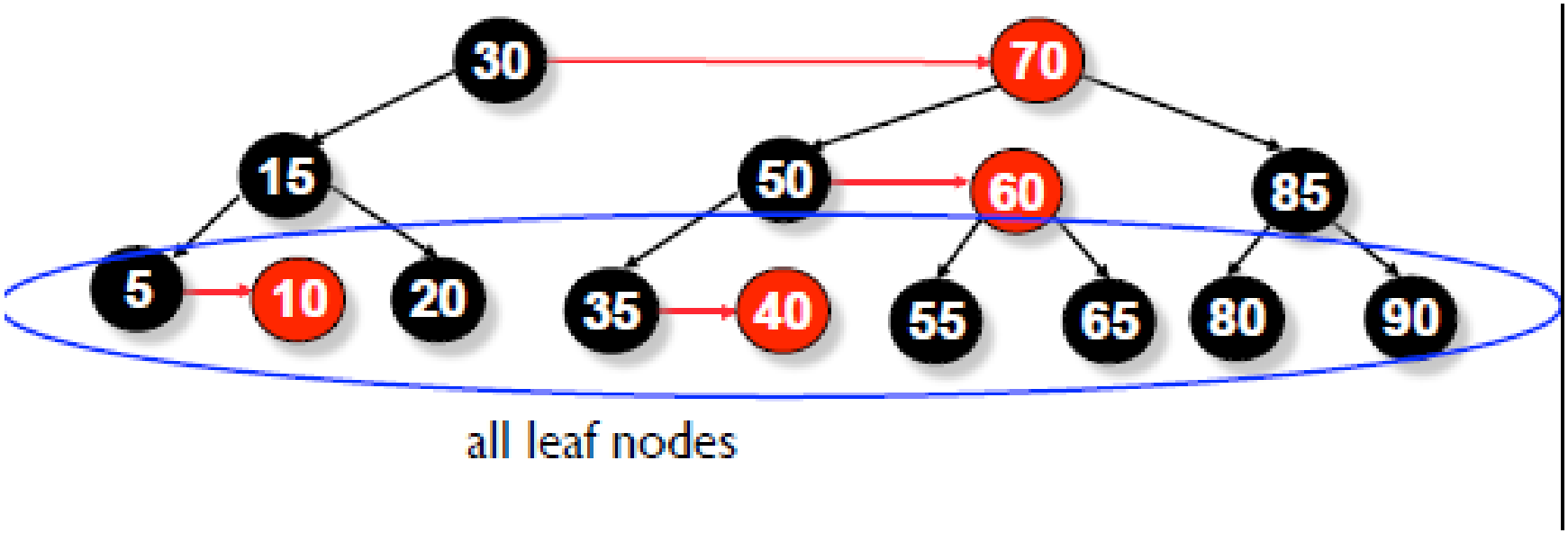
Representation of Balancing Info!

- The level of a node (instead of color) is used as balancing info!
- "red" nodes are simply nodes that located at the same level as their parents!
- For the tree on the previous slide:



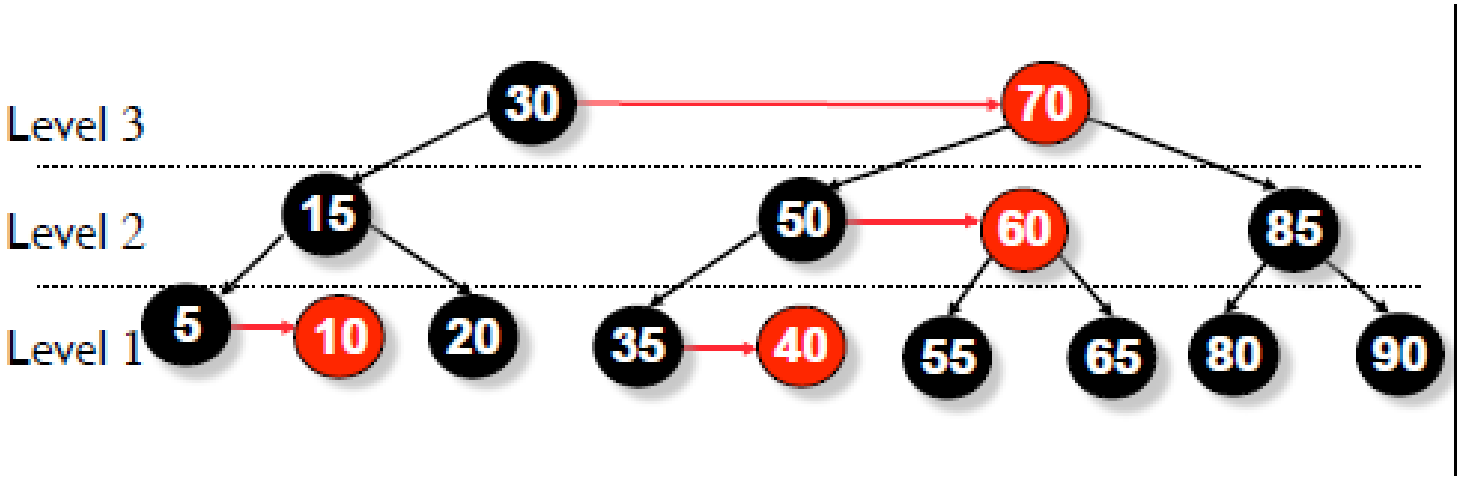
Redefinition of "Leaf"

- Both the terms leaf and level are redefined:
- A leaf in an AA-tree is a node with no black internal-node as children!



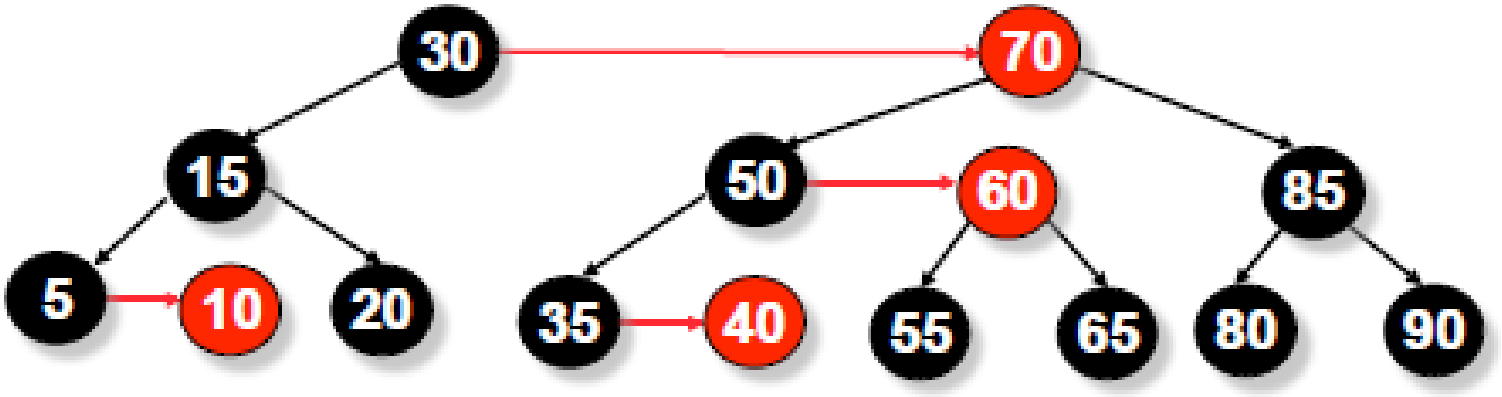
Redefinition of "Level"

- The level of a node in an AA-tree is:
  - leaf nodes are at level 1
  - red nodes are at the level of their parent
  - black nodes are at one less than the level of their parent as in red-black trees, a black node corresponds to a level change in the corresponding 2-3 tree!



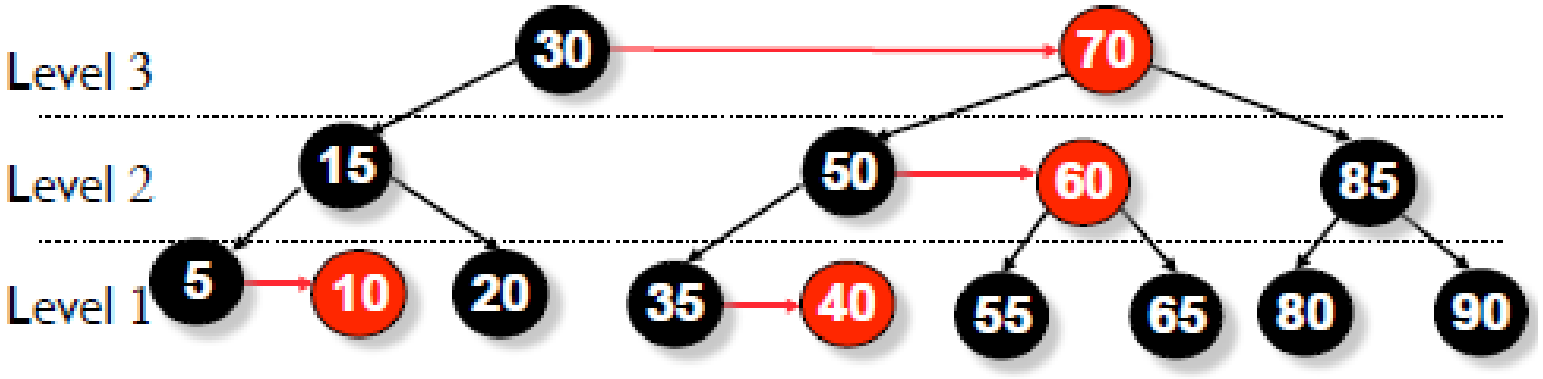
Implications of Ordering Properties

1. Horizontal links are right links! [because only right children may be red!]
2. There may not be double horizontal links! [because there cannot be double red nodes.



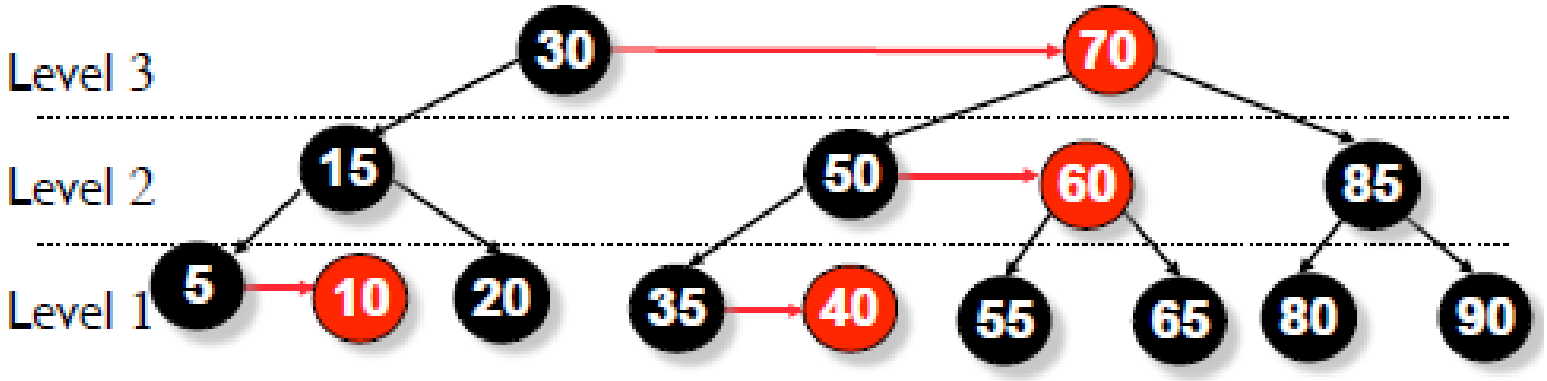
Implications of Ordering Properties

3. Nodes at level 2 or higher must have two children
4. If a node does not have a right horizontal link, its two children are at the same level



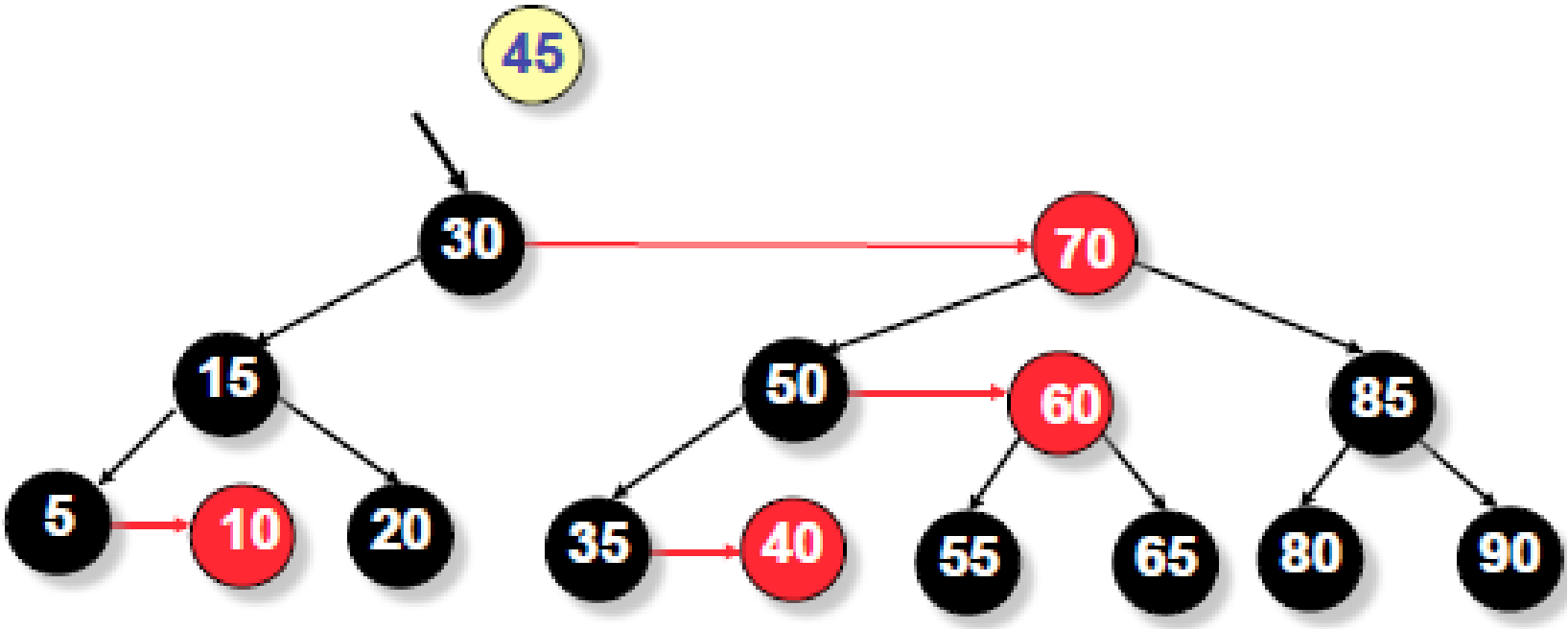
Implications of Ordering Properties

5. Any simple path from a black node to a leaf contains one black node on each level.



Example: Insert 45

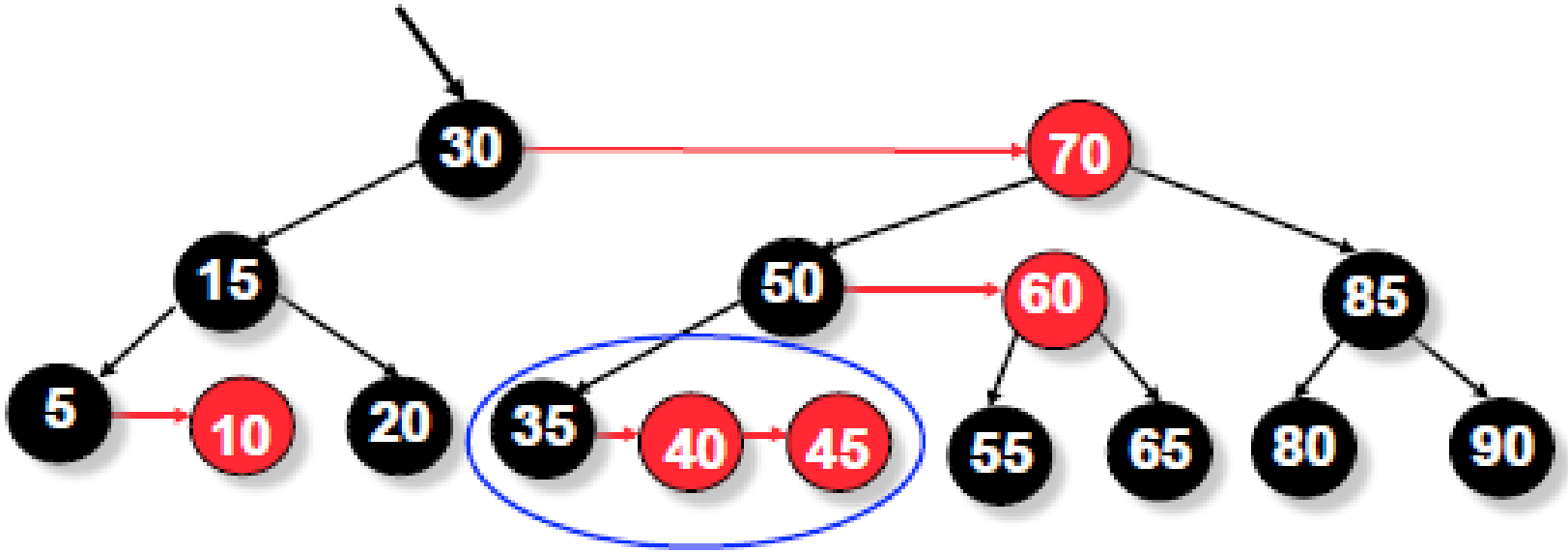
First, insert as for simple binary search tree
Newly inserted node is red



Example: Insert 45

After insert to right of 40:

Problem: double right horizontal links starting at 35, need to split

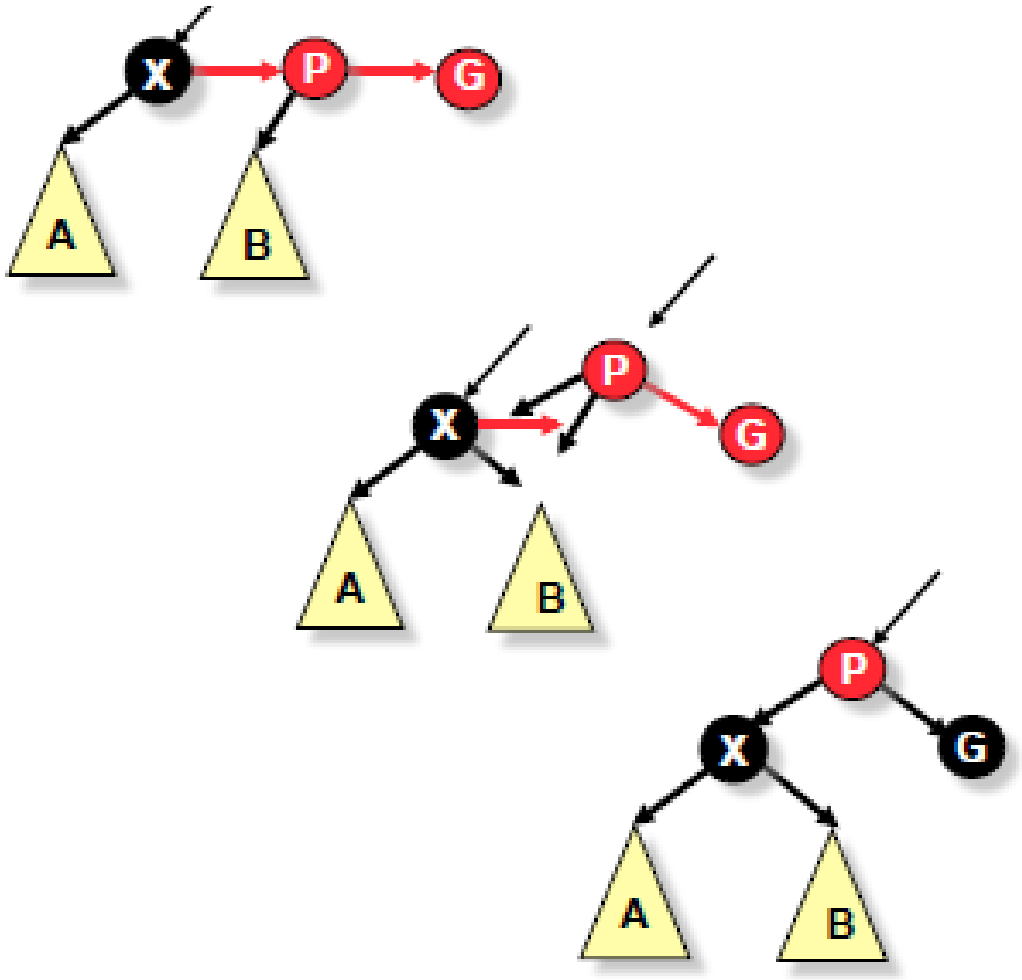


Split: Removing Double Reds

Problem: With G inserted, there are two reds in a row

Split is a simple left rotation between X and P

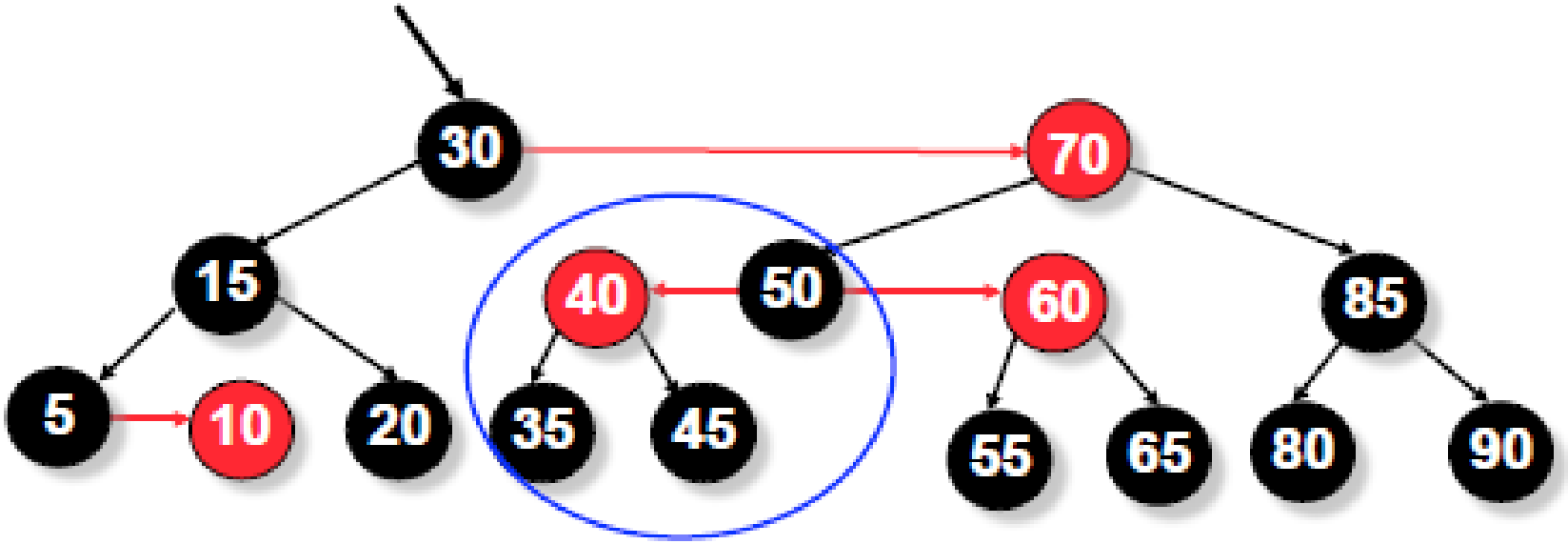
P's level increases in the AA-tree



Example: Insert 45

After split at 35:

Problem: left horizontal link at 50 is introduced, need to skew

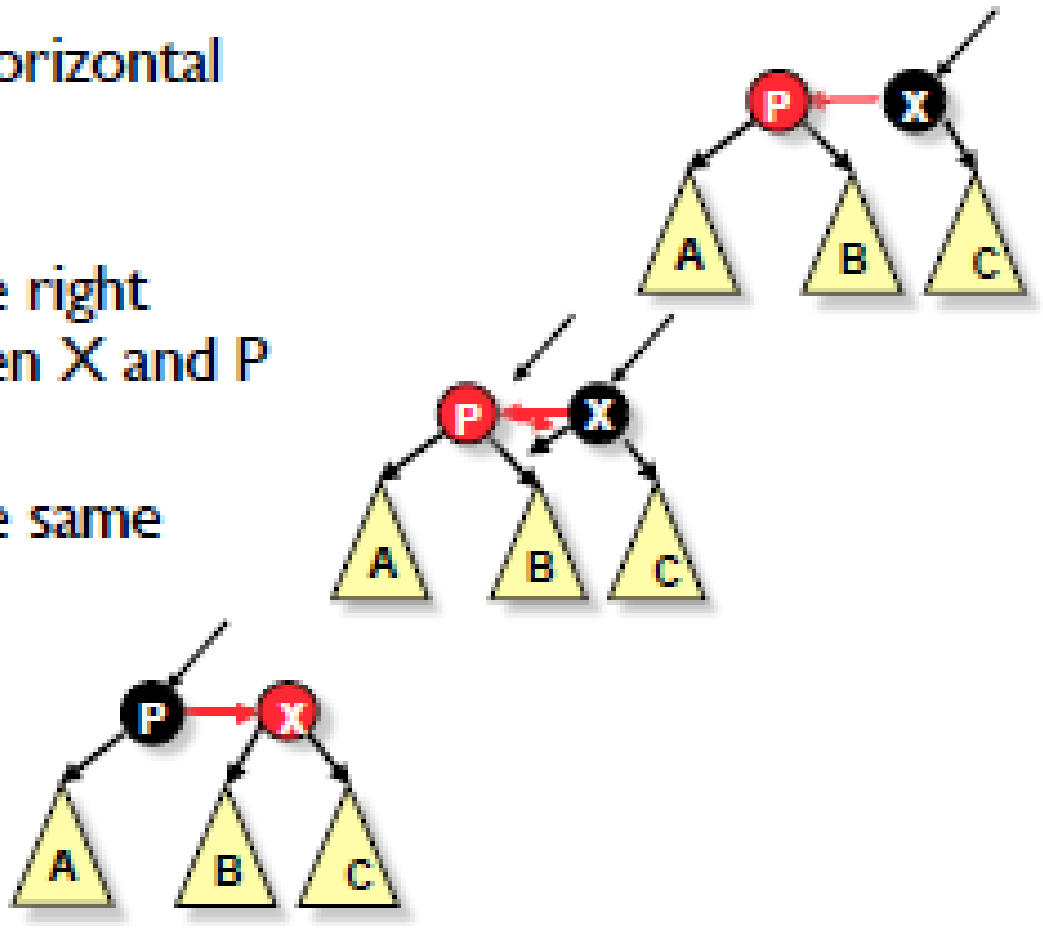


Skew: Removing Left Horizontal Link

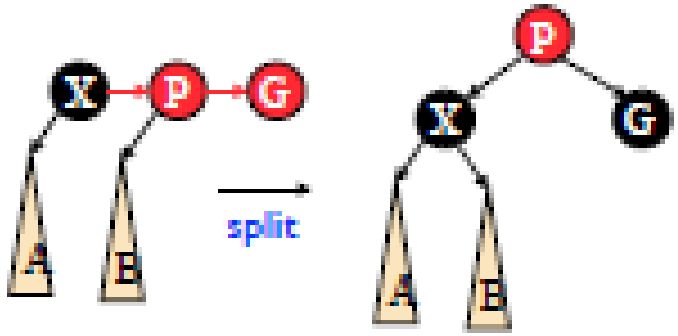
Problem: left horizontal link in AA-tree

Skew is a simple right rotation between X and P

P remains at the same level as X

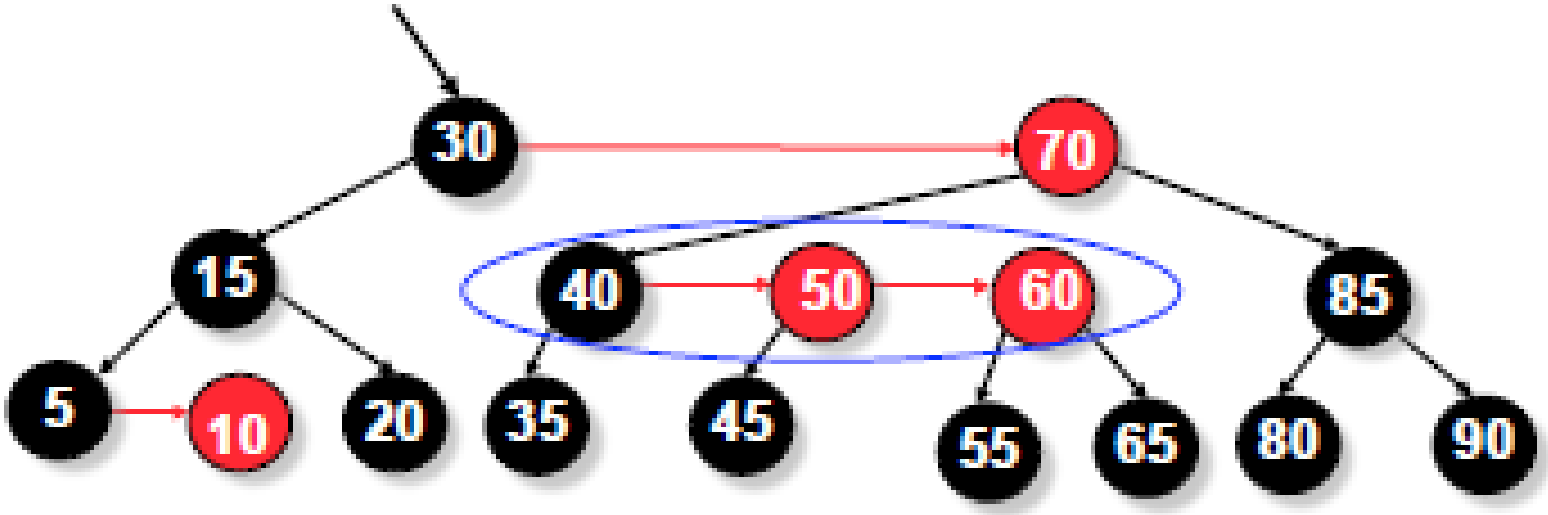


Example: Insert 45

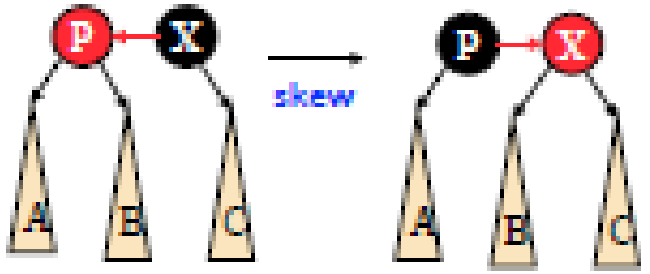


After skew at 50:

Problem: double right horizontal links starting at 40, need to split

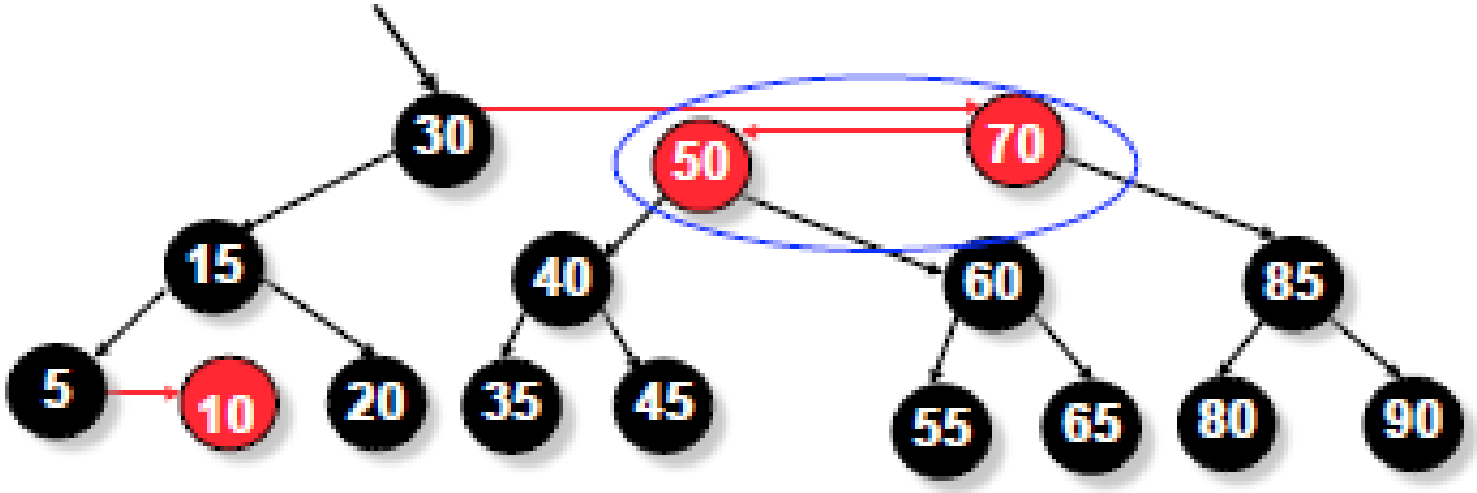


Example: Insert 45

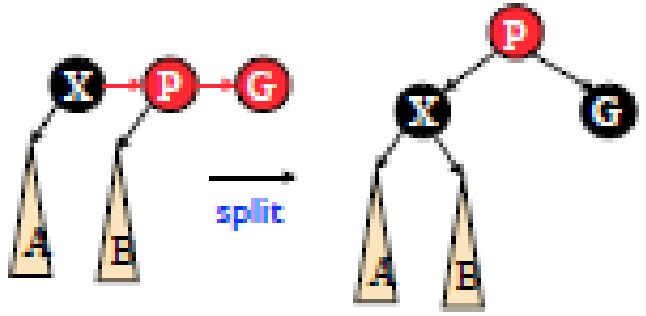


After split at 40:

Problem: left horizontal link at 70 introduced (50 is now on same level as 70), need to skew

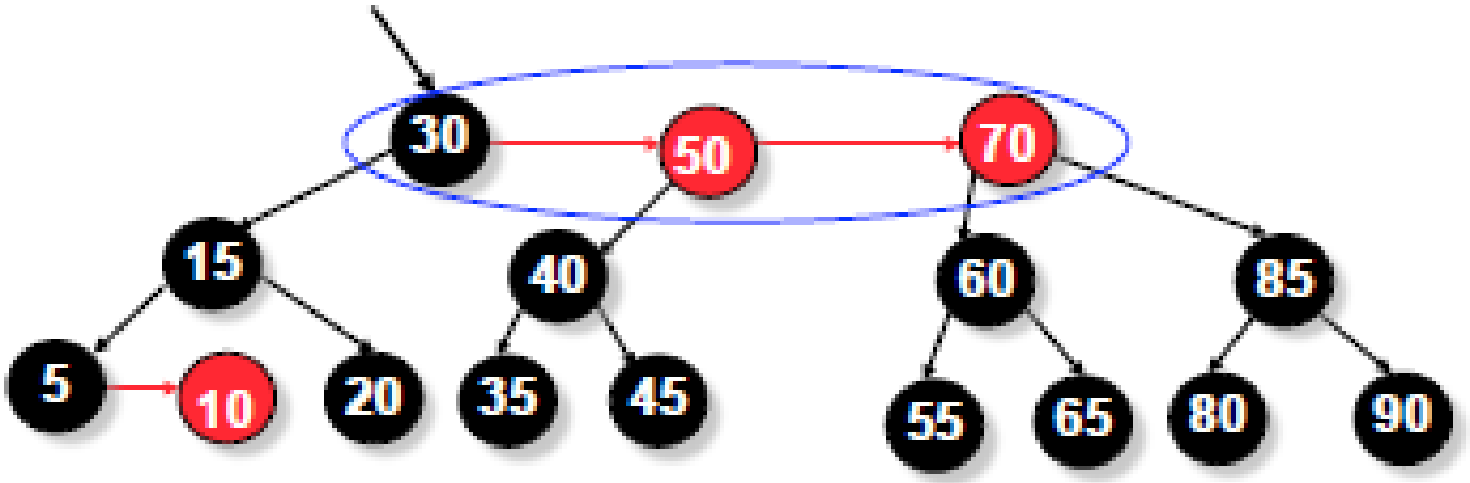


Example: Insert 45



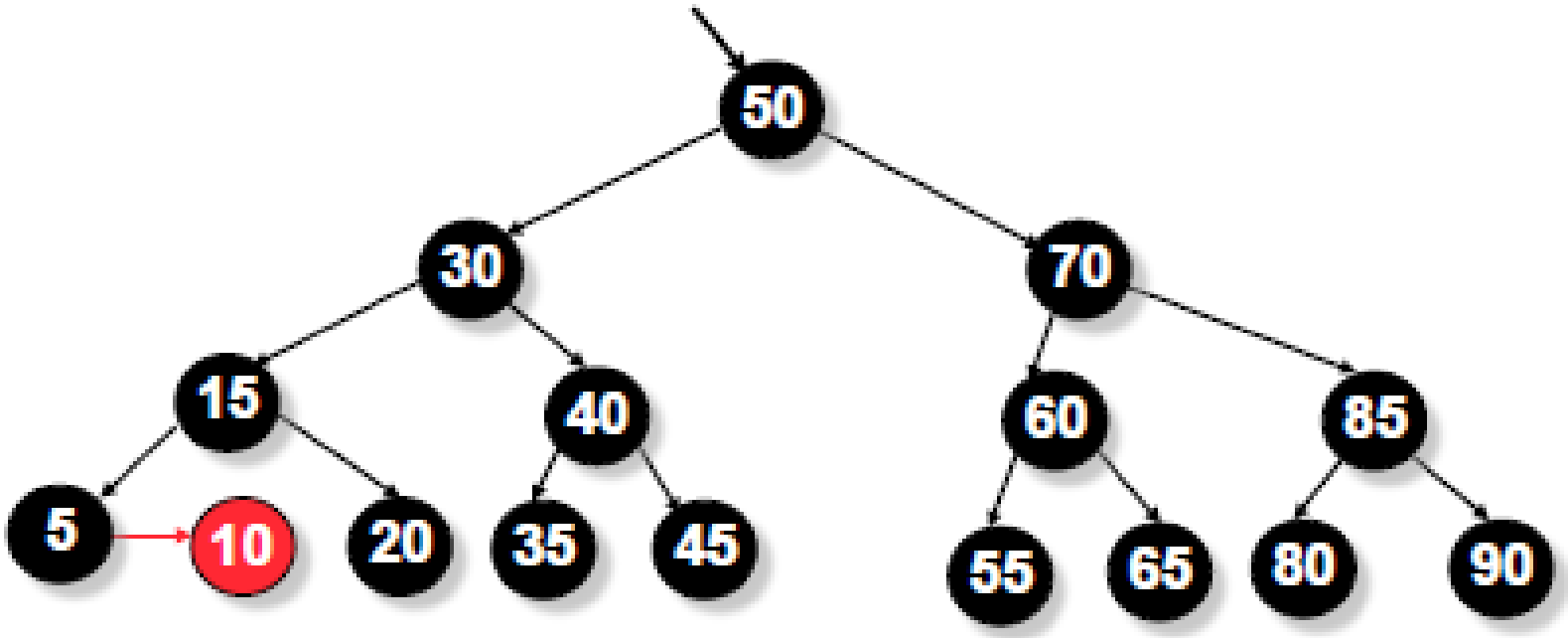
After skew at 70:

Problem: double right horizontal links starting at 30, need to split



Example: Insert 45

After split at 30:
Insertion is complete (finally!)



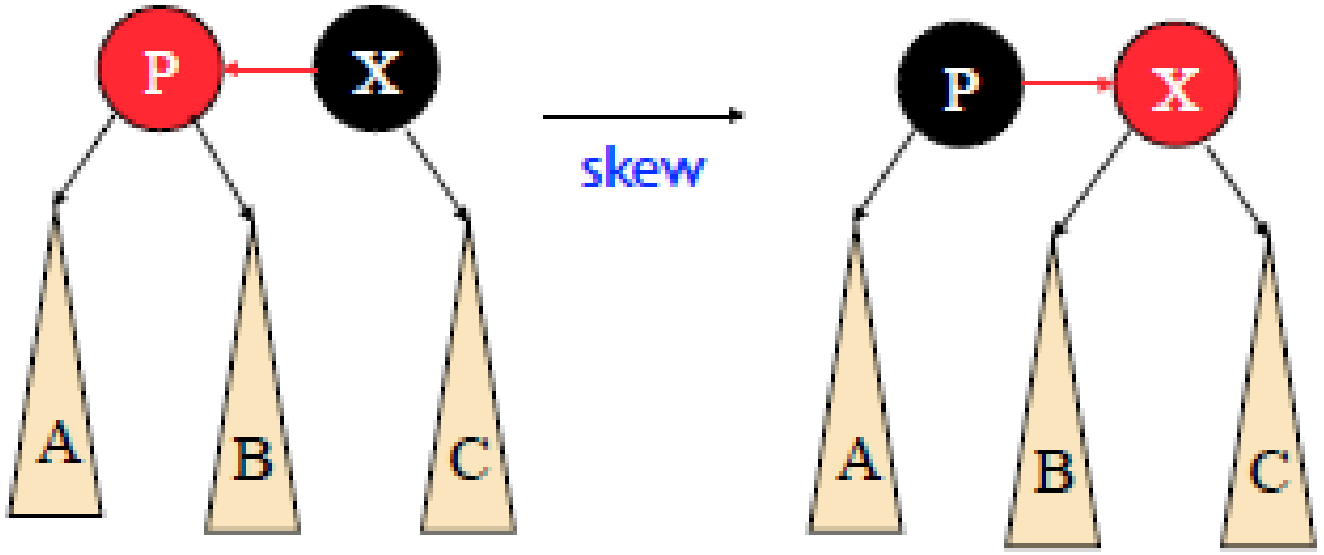
AATree::Insert()

AATree::Insert()

```
void AATree::
insert(Link &root, Node &add) {
   if (root == NULL)  // have found where to insert y
     root = add;
   else if (add→key < root→key) // <= if duplicate ok
     insert(root→left, add);
   else if (add→key > root→key)
     insert(root→right, add);
   // else handle duplicate if not ok

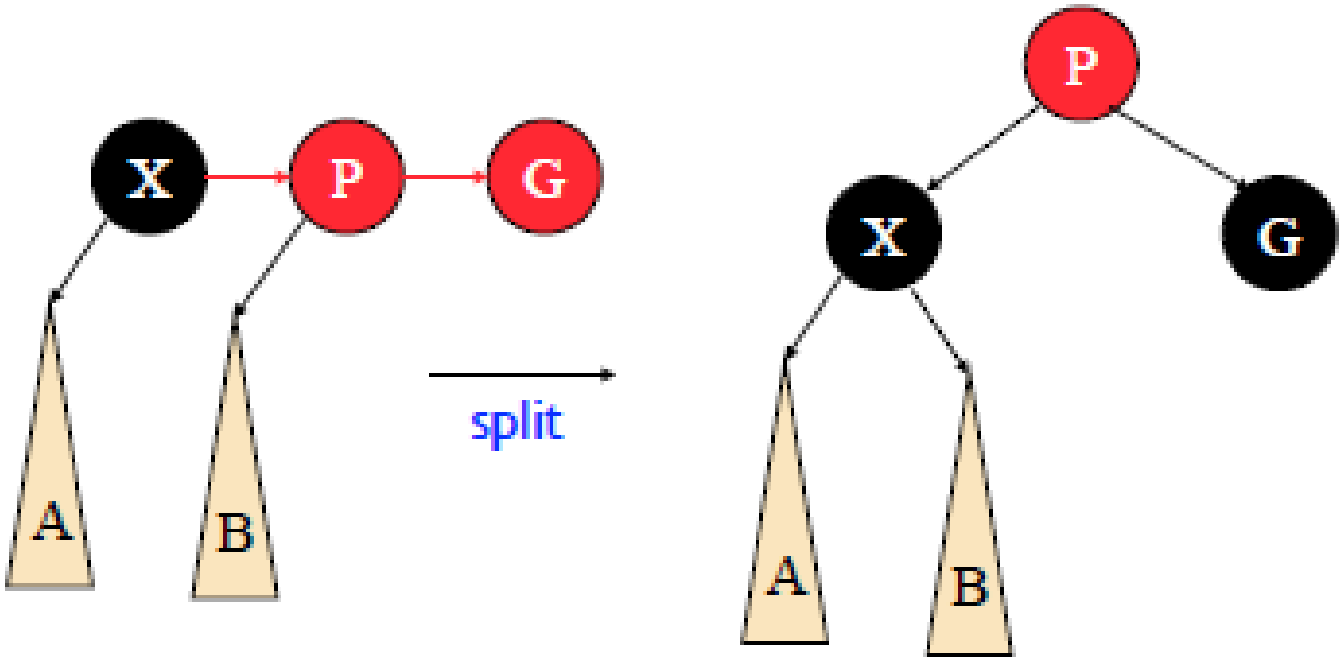
   skew(root);     // do skew and split at each level
   split(root);
}
```

Skew: Remove Left Horizontal Link




```
void AATree::skew(Link &root) { // root = X
   if (root→left→level == root→level)
     rotate_right(root);
}
```

Split: Remove Double Reds!

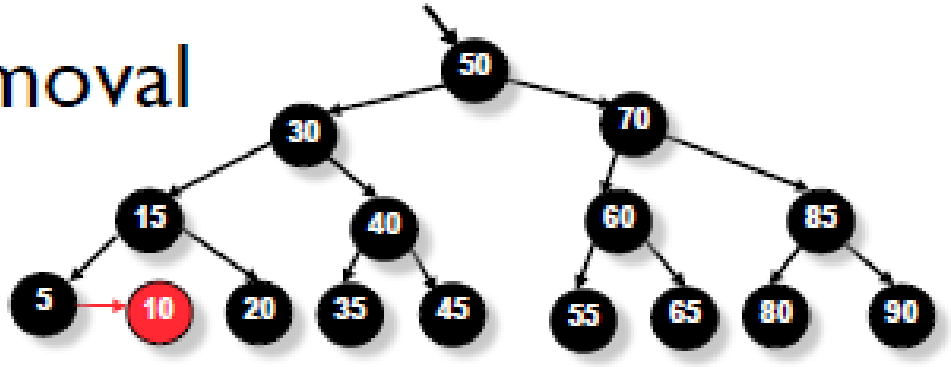



```
void AATree::split(Link &root) { // root = X
   if (root→right→right→level == root→level)
     rotate_left(root);
}
```

More on Skew and Split

- Skew may cause double reds
  - First, we apply skew, then we do split if necessary.
  - After a split, the middle node increases a level, which may create a problem for the original parent
    - parent may need to skew and split.

AA-Tree Removal



Rules:

1. if node to be deleted is a red leaf, e.g., 10, remove leaf, done
2. if it is parent to a single internal node, e.g., 5, it must be black; replace with its child (must be red) and recolor child black
3. if it has two internal-node children, swap node to be deleted with its in-order successor
   - if in-order successor is red (must be a leaf), remove leaf, done
   - if in-order successor is a single child parent, apply second rule

In both cases the resulting tree is a legit AA-tree
(we haven't changed the number of black nodes in paths)

3. if in-order successor is a black leaf, or if the node to be deleted itself is a black leaf, things get complicated . . .

Black Leaf Removal

Follow the path from the removed node to the root

At each node p with 2 internal-node children do:

- if either of p's children is two levels below p
  - decrease the level of p by one
  - if p's right child was a red node, decrease its level also
- skew(p); skew(p→right); skew(p→right→right);
- split(p); split(p→right);

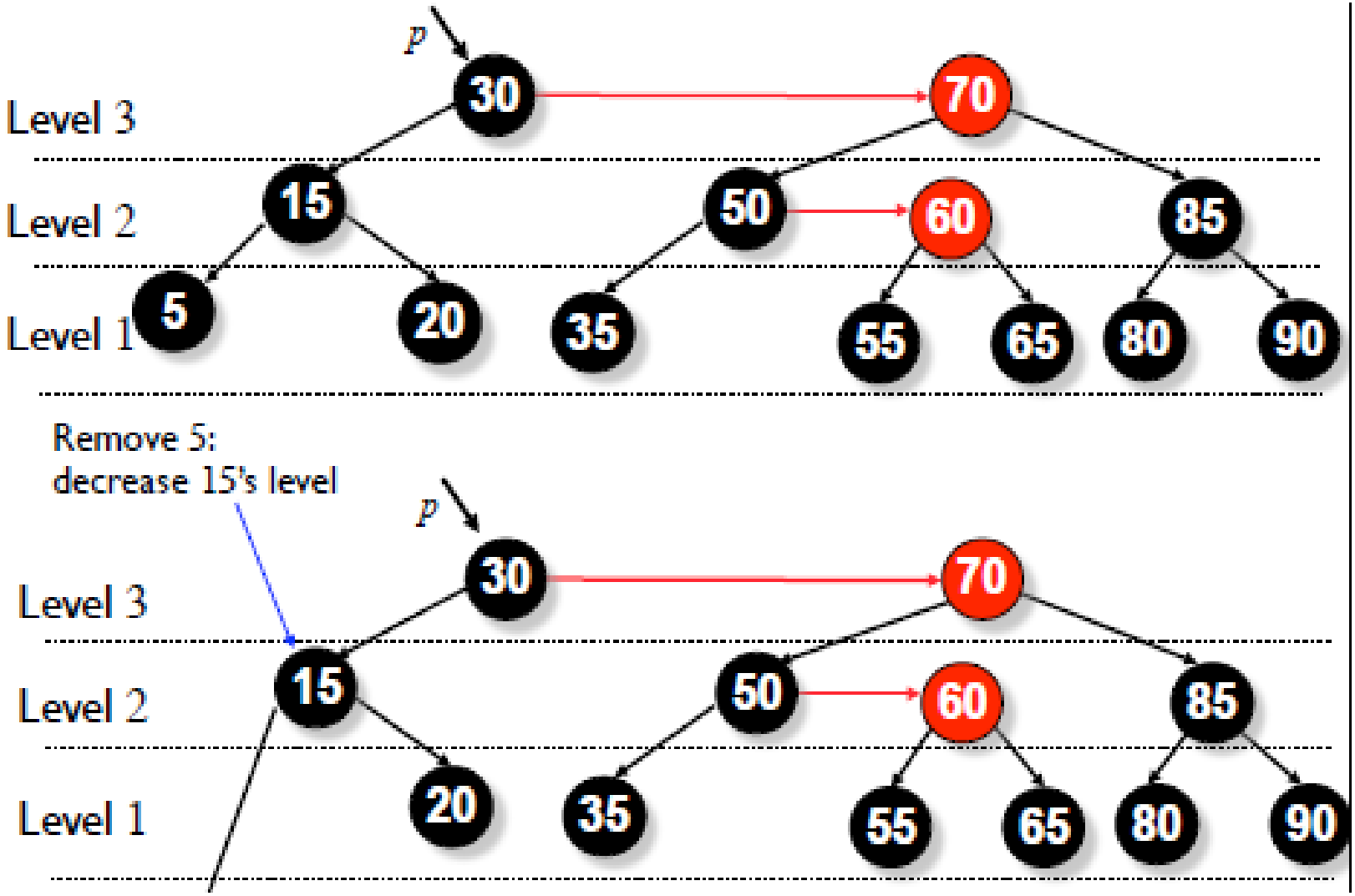
In the worst case, deleting one leaf node, e.g., 15, could cause six nodes to all be at one level, connected by horizontal right links

- but the worst case can be resolved by 3 calls to `skew()`, followed by 2 calls to `split()`!

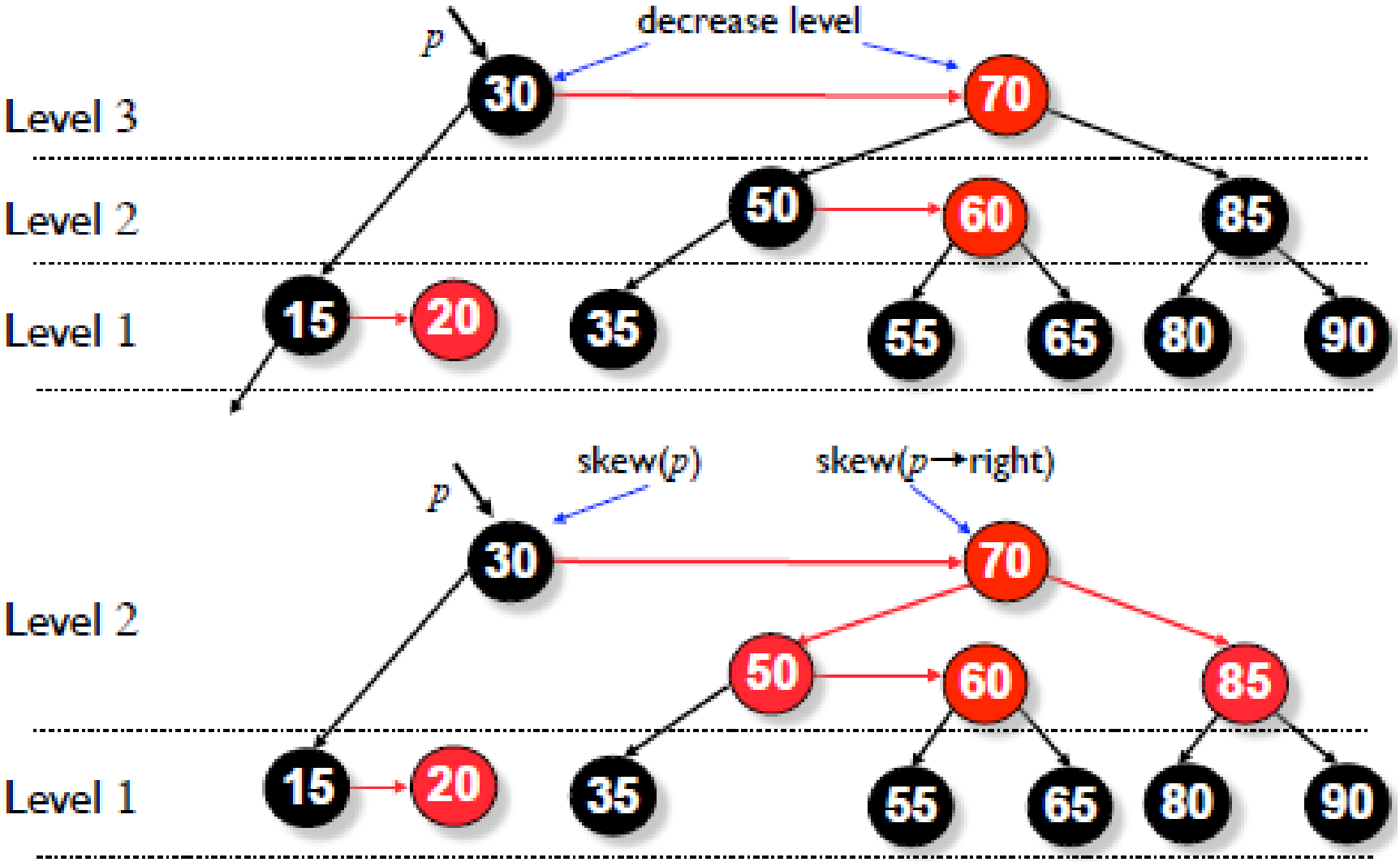


[Andersson,McCollam]

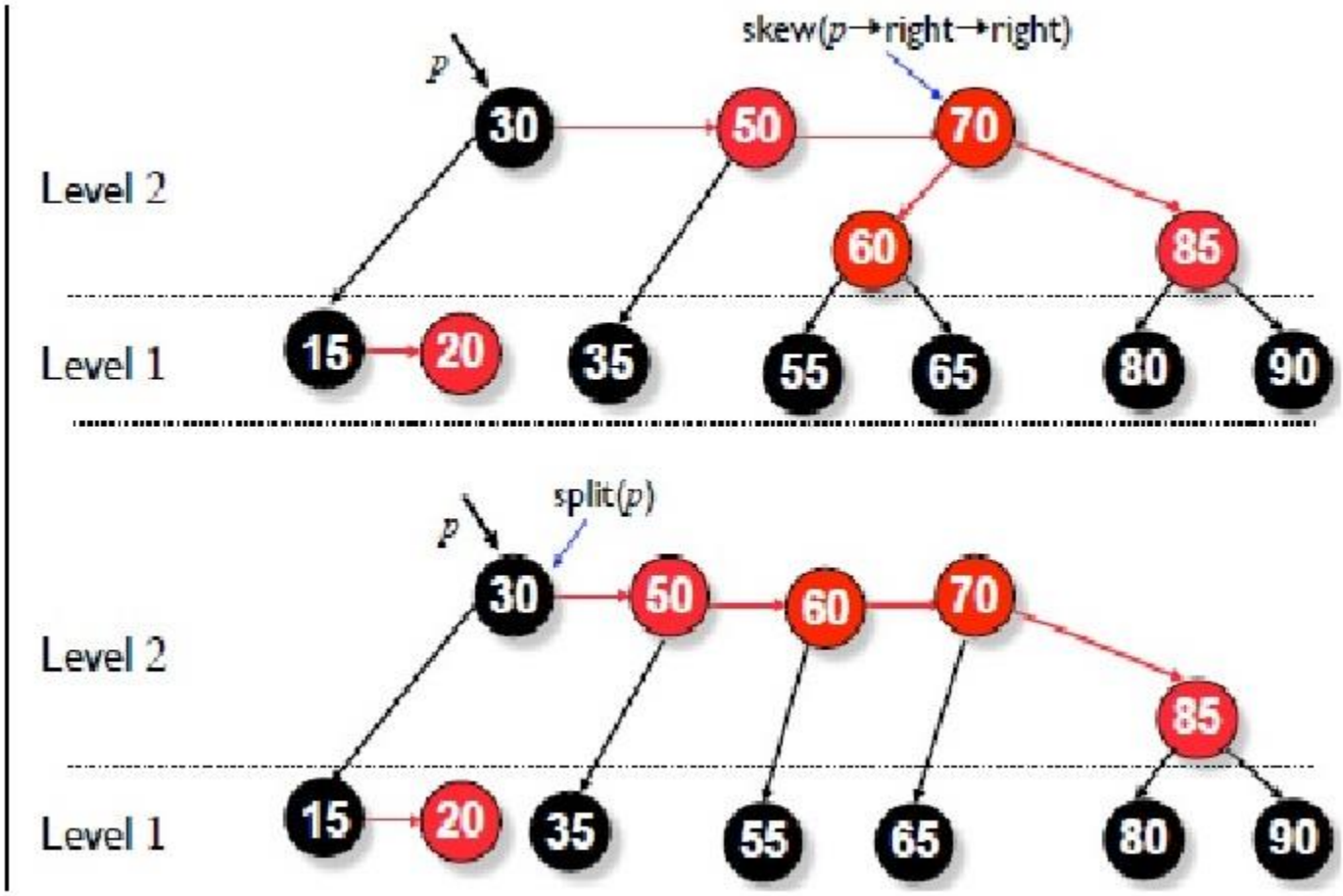
Black Leaf Removal



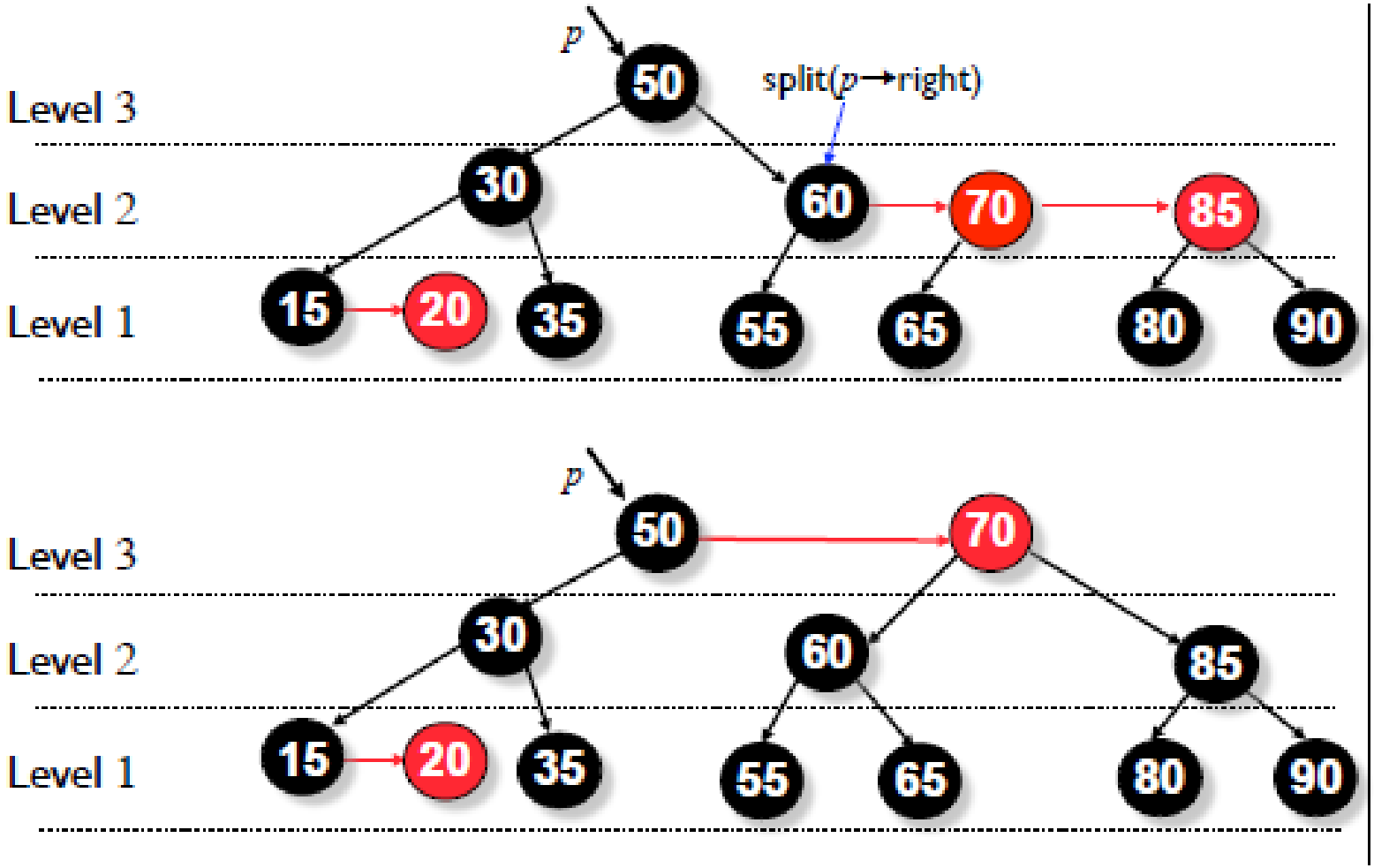
Black Leaf Removal



Black Leaf Removal



Black Leaf Removal



AA-Tree Implementation

```
procedure Skew (var t: Tree);
var temp: Tree;
begin
  if t↑.left↑.level = t↑.level then
  begin { rotate right }
    temp := t;
    t := t↑.left;
    temp↑.left := t↑.right;
    t↑.right := temp;
  end;
end;

procedure Split (var t: Tree);
var temp: Tree;
begin
  if t↑.right↑.right↑.level = t↑.level then
  begin { rotate left }
    temp := t;
    t := t↑.right;
    temp↑.right := t↑.left;
    t↑.left := temp;
    t↑.level := t↑.level + 1;
  end;
end;

procedure Insert (var x: data;
    var t: Tree; var ok: boolean);
begin
  if t = bottom then begin
    new (t);
    t↑.key := x;
    t↑.left := bottom;
    t↑.right := bottom;
    t↑.level := 1;
    ok := true;
  end else begin
    if x < t↑.key then
      Insert (x, t↑.left, ok)
    else if x > t↑.key then
      Insert (x, t↑.right, ok)
    else ok := false;
    Skew (t);
    Split (t);
  end;
end;
```

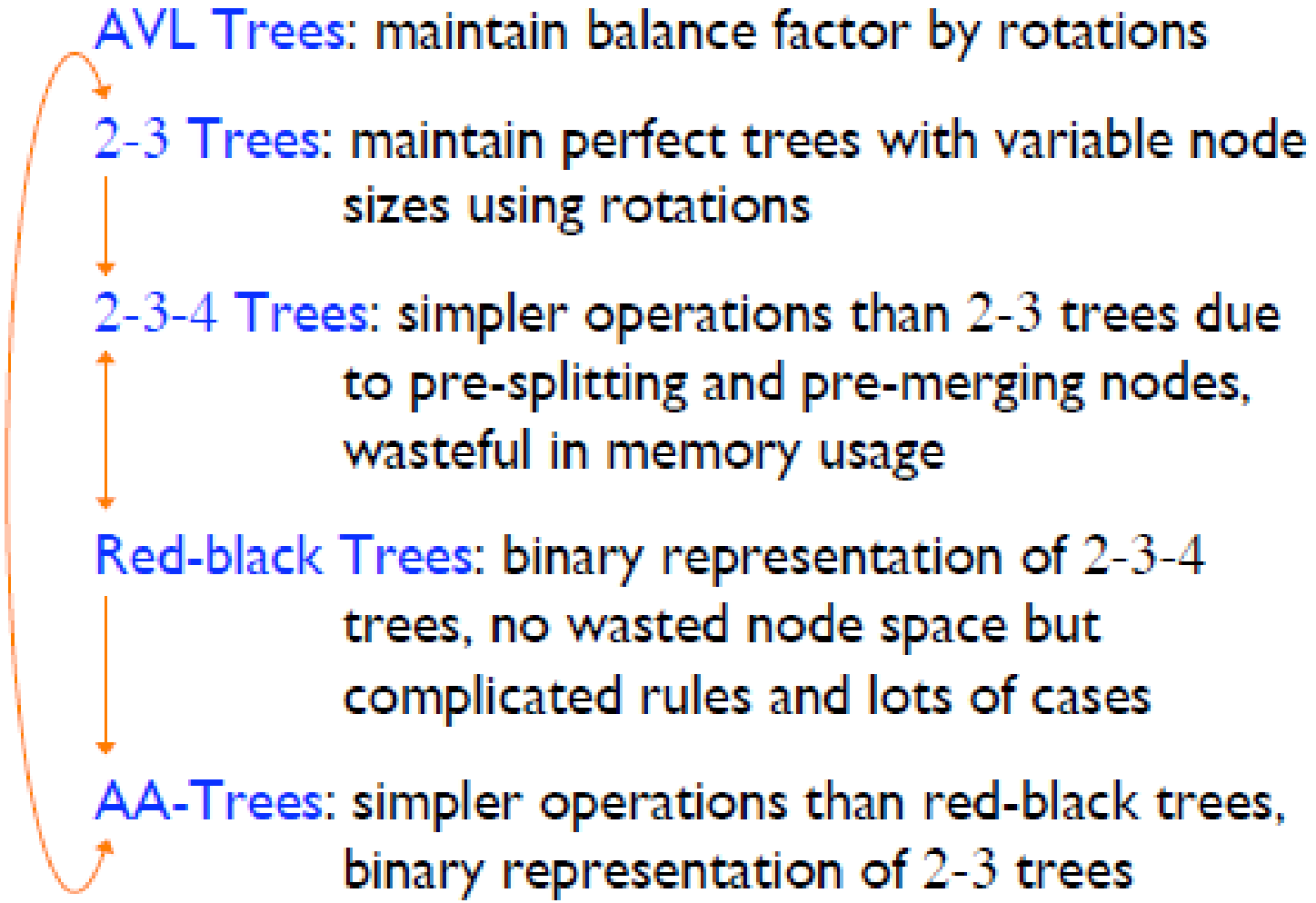
```
procedure Delete (var x: data;
    var t: Tree; var ok: boolean);
begin
  ok := false;
  if t <> bottom then begin

{ 1: Search down the tree and }
{ set pointers last and deleted. }
    last := t;
    if x < t↑.key then
      Delete (x, t↑.left, ok)
    else begin
      deleted := t;
      Delete (x, t↑.right, ok);
    end;

{ 2: At the bottom of the tree we }
{ remove the element (if it is present). }
    if (t = last) and (deleted <> bottom)
    and (x = deleted↑.key) then
    begin
        deleted↑.key := t↑.key;
        deleted := bottom;
        t := t↑.right;
        dispose (last);
        ok := true;
    end

{ 3: On the way back, we rebalance. }
    else if (t↑.left↑.level < t↑.level-1)
    or (t↑.right↑.level < t↑.level-1) then
    begin
      t↑.level := t↑.level -1;
      if t↑.right↑.level > t↑.level then
        t↑.right↑.level := t↑.level;
      Skew (t);
      Skew (t↑.right);
      Skew (t↑.right↑.right);
      Split (t);
      Split (t↑.right);
    end;
  end;
end;
```

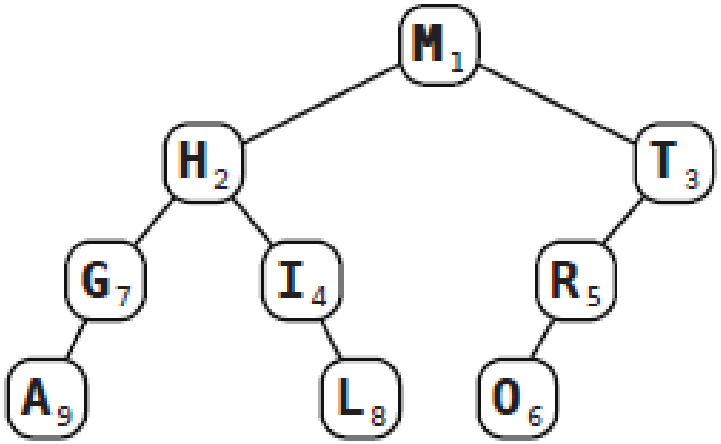
Balanced BST Summary



Randomized Binary Search Trees – Treaps

What is a Treap?

- We consider randomized alternative(s) to balanced binary search tree structures such as AVL trees, red-black trees, B-trees, or splay trees, which are arguably simpler than any of these deterministic structures.
- A **Treap is a binary tree** in which every node has both a **search key and a priority**, where the in-order sequence of search keys is sorted, and each node's priority is smaller than the priorities of its children. In other words, a treap is simultaneously a binary search tree for the search keys and a (min-)heap for the priorities. In our examples, we will use letters for the search keys and numbers for the priorities. **Note:** A treap is a BST with heap-ordered priorities (but it is not a heap as it is not required to be a complete binary tree).

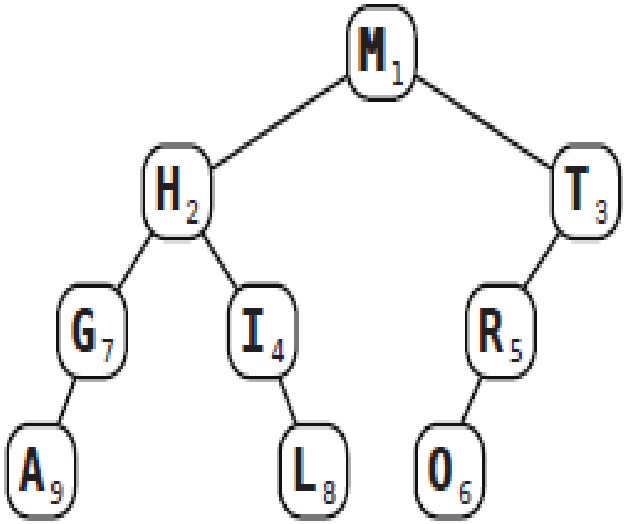


A treap. Letters are search keys; numbers are priorities.

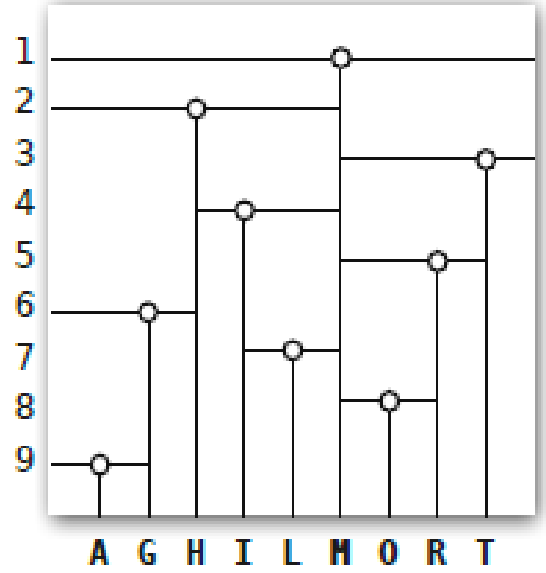
Alternative Representations & A Simplifying Assumption

- We assume from now on that all the keys and priorities are distinct.
- Under this assumption, we can easily prove by induction that the structure of a treap is completely determined by the search keys and priorities of its nodes. Since it's a heap, the node v with highest priority must be the root. Since it's also a binary search tree, any node u with key(u) < key(v) must be in the left subtree, and any node w with key(w) > key(v) must be in the right subtree. Finally, since the subtrees are treaps, by induction, their structures are completely determined. The base case is the trivial empty treap.
- Another way to describe the structure is that a treap is exactly the binary search tree that results by inserting the nodes one at a time into an initially empty tree, in order of increasing priority, using the standard textbook insertion algorithm. This characterization is also easy to prove by induction.
- A third description interprets the keys and priorities as the coordinates of a set of points in the plane. The root corresponds to a T whose joint lies on the topmost point. The T splits the plane into three parts. The top part is (by definition) empty; the left and right parts are split recursively [See the picture in the next slide]. This interpretation has some interesting applications in computational geometry [we will skip details of this].

Geometric Representation of a Treap



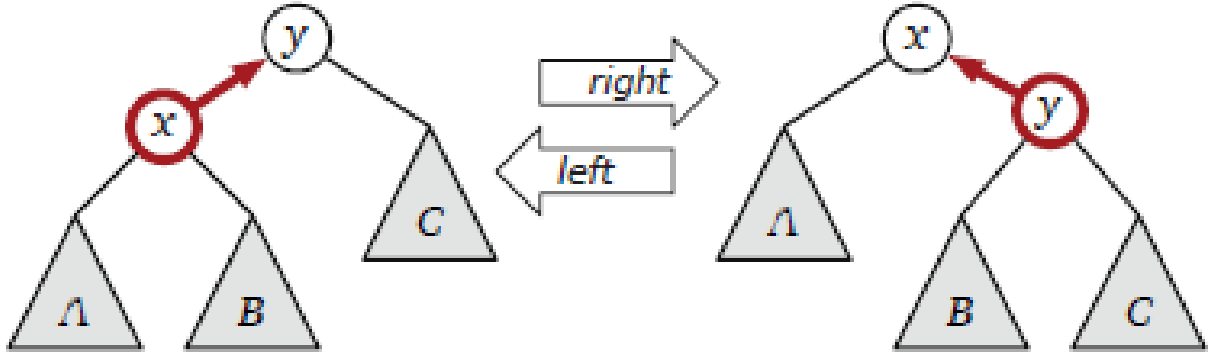
A treap. Letters are search keys; numbers are priorities.



A geometric interpretation of the same treap.

Treap Operations

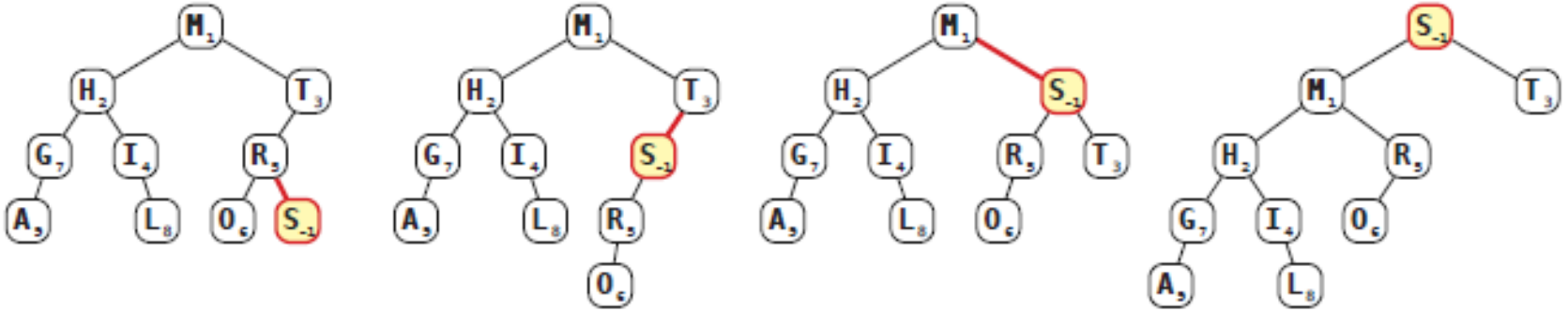
- The search algorithm is the usual one for binary search trees. The time for a successful search is proportional to the depth of the node. The time for an unsuccessful search is proportional to the depth of either its successor or its predecessor.
- To insert a new node z, we start by using the standard binary search tree insertion algorithm priorities may no longer form a heap. To fix the heap property, as long as z has smaller priority than its parent, perform a rotation at z, a local operation that decreases the depth of z by one and increases its parent's depth by one, while maintaining the search tree property. Rotations can be performed in constant time, since they only involve simple pointer manipulation.



A right rotation at x and a left rotation at y are inverses.

Treap Operations

- The overall time to insert z is proportional to the depth of z before the rotations—we must walk down the treap to insert z, and then walk back up the treap doing rotations. Another way to say this is that the time to insert z is roughly twice the time to perform an unsuccessful search for key(z).



Left to right: After inserting S with priority -1, rotate it up to fix the heap property.
Right to left: Before deleting S, rotate it down to make it a leaf.

To delete a node, we just run the insertion algorithm backward in time. Suppose we want to delete node z. As long as z is not a leaf, perform a rotation at the child of z with smaller priority. This moves z down a level and its smaller-priority child up a level. The choice of which child to rotate preserves the heap property everywhere except at z. When z becomes a leaf, chop it off.

Splitting and Joining a Node

- We sometimes want to **split a treap** T into two treaps $T_<$ and $T_>$ along some pivot key π, so that all the nodes in $T_<$ have keys less than π and all the nodes in $T_>$ have keys bigger then π. A simple way to do this is to insert a new node z with key(z) = π and priority(z) = $-\infty$. After the insertion, the new node is the root of the treap. If we delete the root, the left and right sub-treaps are exactly the trees we want. The time to split at is roughly twice the time to(unsuccessfully) search for π.
- Similarly, we may want to **join two treaps** $T_<$ and $T_>$, where every node in $T_<$ has a smaller search key than any node in $T_>$, into one super-treap. Merging is just splitting in reverse – create a dummy root whose left sub-treap is $T_<$ and whose right sub-treap is $T_>$, rotate the dummy node down to a leaf, and then cut it off.

Cost of Operations

- **Search**: A successful search for key k takes O(depth(v)) time, where v is the node with key(v) = k. For an unsuccessful search, let v^- be the inorder predecessor of k (the node whose key is just barely smaller than k), and let v^+ be the inorder successor of k (the node whose key is just barely larger than k). Since the last node examined by the binary search is either v^- or v^+, the time for an unsuccessful search is either O(depth(v^+)) or O(depth(v^-)).
- **Insert/Dele**te: Inserting a new node with key k takes either O(depth(v^+)) time or O(depth(v^-)) time, where v^+ and v^- are the predecessor and successor of the new node. Deletion is just insertion in reverse.
- **Split/Join**: Splitting a treap at pivot value k takes either O(depth(v+)) time or O(depth(v^-)) time, since it costs the same as inserting a new dummy root with search key k and priority $-\infty$. Merging is just splitting in reverse.
- **Note**:
  - In the worst case, the depth of an n-node treap is $\Theta(n)$, so each of these operations has a worst-case running time of $\Theta(n)$.
  - There are other variations like **Random Priorities** and **Skip Lists** [we omit]

Amortization and Splay Trees*

Simple Examples

*Mostly taken from CLR book (Chapter 17)

Amortization – Why and What?

- As an everyday example think of paying off a debt, e.g., mortgage, car loan etc., by smaller payments made over time.
- In an amortized analysis, we average the time required to perform a sequence of data-structure operations over all the operations performed.
- With amortized analysis, we can show that the average cost of an operation is small, if we average over an arbitrary sequence of operations, even though a single operation within the sequence might be expensive.
- Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the average (over an arbitrary sequence of operations) performance of each operation in the worst case.
- Word of Caution: In applications where cost of individual operations is required to have low and/or comparable costs, we may need an algorithm with a worse amortized cost but a better worst-case per operation bound.
- Another way of looking at performance of an algorithm: Amortized analysis exploits the fact that some "expensive" operations usually pay for future operations by limiting the number or cost of future such operations. Examples: splay trees, B-trees, disjoint set unions, Fibonacci heaps, security, database, distributed computing applications etc.
- "Amortized analysis is closely related to competitive analysis, which involves comparing the worst case performance of an online algorithm to the performance of an optimal offline algorithm on the same data. Amortization is useful because competitive analysis's performance bounds must hold regardless of the particular input."

Amortization – How?

- There are 3 principal approaches:
  - **Aggregate analysis**: we determine an upper bound T(n) on the total cost of a sequence of n operations. The average cost per operation is then T(n)/n. We take the average cost as the amortized cost of each operation, so that all operations have the same amortized cost.
  - **Accounting method**: [Banker's view] we determine an amortized cost of each operation. When there is more than one type of operation, each type of operation may have a different amortized cost. The accounting method *overcharges some operations* early in the sequence, storing the overcharge as "prepaid credit" on specific objects in the data structure. Later in the sequence, the credit pays for operations that are charged less than they actually cost.
  - **Potential method**: Similar to the accounting method in that we determine the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later. The potential method maintains the credit as the "potential energy" of *the data structure as a whole instead of associating the credit with individual objects* within the data structure.
- The charges assigned during an amortized analysis are for analysis purposes only. They need do not appear in the code.
- When we perform an amortized analysis, we often gain insight into a particular data structure, and this insight can help us optimize the design.

Aggregate Analysis

- In aggregate analysis, we show that for all n, a sequence of n operations takes worst-case time T(n) in total. In the worst case, the average cost, or amortized cost, per operation is therefore T(n)/n. **Note** that this amortized cost applies to each operation, even when there are several types of operations in the sequence; this is one difference with the accounting method and the potential method, that may assign different amortized costs to different types of operations.
- **Example 1**: Consider a fixed size (say n) stack S with 2 operations Push (S, x) and Pop (S, x). Assuming there are overflow and underflow checks for errors, each operation takes O(1) time; consider the cost of each is 1; then total cost of any arbitrary sequence of n operations $T(n) = \Theta(n) \Rightarrow$ amortized cost of each operation is O(1).
  - Now, add a multi-pop operation MP (S, k) that pops k elements sequentially; obviously, total cost of MP (S, k) operation is min (s, k) [this many Pops]. Worst case behavior of multi-pop operation MP (S, k) is then O(n). Does that mean that amortized cost of an arbitrary sequence of 3 operations is O(n)? NO.
  - Note that we can pop each object from the stack at most once for each time we have pushed it onto the stack. The number of times that POP can be called on a nonempty stack, including calls within multi-pop is at most the number of Push operations, which is at most n. For any value of n, any sequence of n Push, Pop and multiop operations takes a total of O(n) time. The average cost of an operation is O(n)/n = O(1).
  - Note that we did not use probabilistic reasoning.

Aggregate Analysis

- **Example 2:** [Increment a binary counter] Consider a k-bit counter A[0 .. k – 1] that is initially 0. An integer stored in A is $x = \sum_{0}^{k-1} A[i].2^i$
  - To add 1 (modulo 2^k) to the counter, we use **{**i=0; while (i<k && A[i]=1) {A[i]=0; i++} if i<k A[i]=1**}**

- A single execution of increment takes time O(k) in the worst case, when A contains all 1s. Thus, a sequence of n such operations on an initially zero counter takes time O(nk) in the worst case. Correct? Yes, but …
- Observe that not all bits flip each time increment is called. A[0] does flip each time increment is called. The next bit up, A[1], flips only every other time and so on.
- The total number of flips in the sequence of n increment operation is $\sum_{0}^{k-1} \left\lfloor n/2^i \right\rfloor < n \sum_{0}^{\infty} 1/2^i = 2n$
- The worst-case time for a sequence of n increment operations on an initially zero counter is therefore O(n). The average cost of each operation, and therefore the amortized cost per operation, O(n)/n = O(1).
- **Exercise**: Show that if a decrement operation is included, a sequence of n operations will cost Θ(nk) time in the worst case.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Accounting Method

- In the accounting method, we assign differing charges to different operations, with some operations charged more or less than they actually cost (c). We call the amount we charge an operation its amortized cost (ac). When an operation's amortized cost exceeds its actual cost, we assign the difference to specific objects in the data structure as credit. Credit can help pay for later operations whose amortized cost is less than their actual cost. Different operations may have different amortized costs – thus, this method differs from aggregate analysis, in which all operations have the same amortized cost.
- To show that in the worst case the average cost per operation is small by analyzing with amortized costs, we must ensure that the total amortized cost of a sequence of operations provides an upper bound on the total actual cost of the sequence. Moreover, as in aggregate analysis, this relationship must hold for all sequences of operations.
- If we denote the actual cost of the i^{th} operation by c_i and the amortized cost of the i^{th} operation by ac_i , we require $\sum_{i=1}^{n} ac_i \geq \sum_{i=1}^{n} c_i$ for all sequences of n operations.
- **Example 1**: Consider the stack example again. Recall that the actual costs of the operations were c(Push) = 1, c(Pop) = 1, c(nulti-pop) = min (s, k). Let us assign the amortized costs of those be 2, 0, 0 respectively. The rest is obvious.
- Note that the amortized cost of multi-pop is a constant (0), whereas the actual cost is variable. Here, all three amortized costs are constant. In general, the amortized costs of the operations under consideration may differ from each other, and they may even differ asymptotically.

Accounting Method

- **Example 2:** [Incrementing a binary counter] We again we analyze the Increment operation on a binary counter that starts at zero.
  - The running time of this operation is proportional to the number of bits flipped; we use this as our cost.
  - We charge an amortized cost of 2 dollars to set a bit to 1. When a bit is set, we use 1 (out of the 2 charged) to pay for the actual setting of the bit, and we place the other 1 on the bit as credit to be used later when we flip the bit back to 0. At any point in time, every 1 in the counter has a credit 1 on it, and thus we can charge nothing to reset a bit to 0; we just pay for the reset with the 1 already credited on the bit.
  - Now we determine the amortized cost. The cost of resetting the bits within the while loop is paid for by the credits on the bits that are reset. The Increment procedure sets at most one bit, and therefore the amortized cost of an Increment operation is at most 2. The number of 1s in the counter never becomes negative, and thus the amount of credit stays nonnegative at all times. Thus, for n Increment operations, the total amortized cost is O(n) which bounds the total actual cost.
- **Exercise:** Suppose we perform a sequence of stack operations on a stack whose size never exceeds k. After every k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack, is O(n) by assigning suitable amortized costs to the various stack operations.

Potential Method

- The potential method of amortized analysis represents the prepaid work as "potential energy," or just "potential," which can be released to pay for future operations. We associate the potential with the data structure as a whole rather than with specific objects within the data structure.
- We perform n operations, starting with an initial data structure D_0. For each i = 1, 2, … n, let c_i be the actual cost of the i^{th} operation and D_i be the data structure that results after applying the i^{th} operation to data structure D_{i-1}. A **potential function Φ** maps each D_i to a real number $\Phi(D_i)$. The **amortized cost** ac_i of the i^{th} operation is given by $ac_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$, i.e., amortized cost of each operation is therefore its actual cost plus the change in potential due to the operation. Thus, the **total amortized cost of n operations is $\sum ac_i = \sum c_i + \Phi(D_n) - \Phi(D_0)$.**
- If we can choose a suitable Φ such that $\Phi(D_n) \geq \Phi(D_0)$, then total amortized cost $\sum ac_i$ will be an upper bound on total actual cost $\sum c_i$, i.e., $\Phi(D_i) \geq \Phi(D_0)$ for all i, since n is arbitrary. Usually we choose $\Phi(D_0) = 0$ and then show that $\Phi(D_i) \geq 0$ for all i [This is by no means necessary].
- The amortized costs, as defined, depend on the choice of the potential function Φ. Different potential functions may yield different amortized costs yet still be upper bounds on the actual costs. We often find trade-offs that we can make in choosing a potential function; the best potential function to use depends on the desired time bounds.

Potential Method

- Example 1: Stack Operations with Pop, Push and multi-pop:
  - Define Φ to be the number of elements in the stack, $\Phi(D_0) = 0$ (stack is initially empty).
  - Potential after i operations, $\Phi(D_i) \geq 0 = \Phi(D_0)$ (the stack never has negative number of elements); total amortized cost of n operations with respect to Φ represents an upper bound on the actual cost.
  - If the i^{th} operation on a stack with x elements is a Push, $\Phi(D_i) - \Phi(D_{i-1}) = (x+1) - x = 1$. So, amortized cost $ac_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$.
  - If the i^{th} operation on a stack with x elements is a multipop (S, k), causing y = min(x,k) elements to be popped from the stack S, then actual cost is y and potential difference is $\Phi(D_i) - \Phi(D_{i-1}) = y - y = 0$. Similarly, the amortized cost of an ordinary Pop operation is 0.
  - The amortized cost of each of the three operations is O(1), and thus the total amortized cost of a sequence of n operations is O(n). Since $\Phi(D_i) \geq 0 = \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost. The worst-case cost of n operations is therefore O(n).

Potential Method

- **Example 2:** Incrementing a binary counter. We define the potential of the counter after the i^{th} Increment operation to be b_i, *the number of 1s in the counter after* the i^{th} operation.
  - Assume the i^{th} operation resets t_i bits and sets <u>at most</u> one bit to 1 (why?). The actual cost c_i is at most t_i+1.
    - If $b_i = 0$, then the i^{th} operation resets all k bits, and so $b_{i-1} = t_i = k$.
    - If $b_i > 0$, $b_i = b_{i-1} - t_i + 1$
  - In either case, $b_i \leq b_{i-1} - t_i + 1$ and so, the potential difference is $\Phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$, or amortized cost $\mathbf{ac_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2}$.
  - If counter starts at zero, then $\Phi(D_0) = 0$ and since $\Phi(D_i) \geq 0$ for all i, the total amortized cost of a sequence of n Increment operations is an upper bound on the total actual cost, and so the worst-case cost of n Increment operations is O(n).
  - **Brain Teaser**: Now, suppose the counter does not start at 0, but at some (arbitrary) positive value. Can we compute the amortized cost? Is it still O(n)? May be, under some condition?
- **Exercises:**
  1. Consider an ordinary binary min-heap data structure with n elements supporting the instructions INSERT and EXTRACT-MIN in O(log n) worst-case time. Give a potential function Φ such that the amortized cost of INSERT is O(log n) and the amortized cost of EXTRACT-MIN is O(1) and show that it works.
  2. What is the total cost of executing n of the stack operations PUSH, POP, and MULTIPOP, assuming that the stack begins with s_0 objects and finishes with s_n objects?

Solution of Brain Teaser

If the counter starts at zero, then $\Phi(D_0) = 0$. Since $\Phi(D_i) \geq 0$ for all i, the total amortized cost of a sequence of n INCREMENT operations is an upper bound on the total actual cost, and so the worst-case cost of n INCREMENT operations is $O(n)$.

The potential method gives us an easy way to analyze the counter even when it does not start at zero. The counter starts with b_0 1s, and after n INCREMENT operations it has b_n 1s, where $0 \leq b_0, b_n \leq k$. (Recall that k is the number of bits in the counter.) We can rewrite equation (17.3) as

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0) .$$

We have $\hat{c}_i \leq 2$ for all $1 \leq i \leq n$. Since $\Phi(D_0) = b_0$ and $\Phi(D_n) = b_n$, the total actual cost of n INCREMENT operations is

$$\begin{aligned} \sum_{i=1}^{n} c_i &\leq \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0 . \end{aligned}$$

Note in particular that since $b_0 \leq k$, as long as $k = O(n)$, the total actual cost is $O(n)$. In other words, if we execute at least $n = \Omega(k)$ INCREMENT operations, the total actual cost is $O(n)$, no matter what initial value the counter contains.

Dynamic Tables

- We allocate space for a table, only to find out later that it is not enough. We must then reallocate the table with a larger size and copy all objects stored in the original table over into the new, larger table. Similarly, if many objects have been deleted from the table, it may be worthwhile to reallocate the table with a smaller size. Using amortized analysis, we show that the amortized cost of insertion and deletion is only O(1), even though the actual cost of an operation is large when it triggers an expansion or a contraction.
- The details of data structures used (e.g., arrays, stack, heap, or hash table) is not necessary for this analysis – we dynamically expand the table when we attempt an insertion in an already full table (T) and we contract the table when the table becomes mostly empty.
- We need a measure when to trigger contraction. We use a similar concept used in hashing – the load factor of a table T, $\alpha(T)$, number of items stored in the table divided by the size (number of slots) of the table. We assign an empty table (one with no items) size 0, and we define its load factor to be 1. If the load factor of a dynamic table is bounded below by a constant, the unused space in the table is never more than a constant fraction of the total amount of space.
- Note that the problem is different from dynamic insert/delete in a linked list. We will start with table expansion only and then will add the deletion.

Dynamic Table Expansion

- Assume that storage for a table T is allocated as a contiguous array of slots. A table fills up when all slots have been used or, equivalently, when its load factor is 1; assuming integer elements, int *T = (int *) malloc (size_of_T * sizeof(int)); use integers Tsize and Tnum to denote the size of the table and the actual number of elements inserted in T. Note that only operation on the table is an insert (T, x).
- To insert an item into a full table, we expand the table by allocating a new table with more slots than the old table had; we allocate a new array for the larger table and then copy items from the old table into the new table.
- A common heuristic allocates a new table with twice as many slots as the old one. If the only table operations are insertions, then the **load factor** of the table is always at least 1/2, and thus the amount of wasted space never exceeds half the total space in the table.
- Initially, the table is empty, i.e., Tsize = 0 and Tnum = 0.

  Insert (T, x)

  if (Tsize == 0) {allocate a table of size 1; Tsize = 1; insert x; Tnum = 1};

  if (Tnum == Tsize) {allocate a table newT of size 2*Tsize; copy elements; free T; T = newT; Tsize = 2*Tsize; insert x; Tnum ++;}
- What is the cost of ith operation? If the table is not full, $c_i = 1$, else $c_i = i$ (1 + cost of copying earlier i – 1 elements).

$$c_i = \begin{cases} i & \text{if } i\text{-}1 \text{ is a power of } 2 \\ 1 & \end{cases}, \quad \text{or } \sum_{i=1}^{n} c_i = n + \sum_{j=0}^{\log n} 2^j < n + 2n = 3n$$

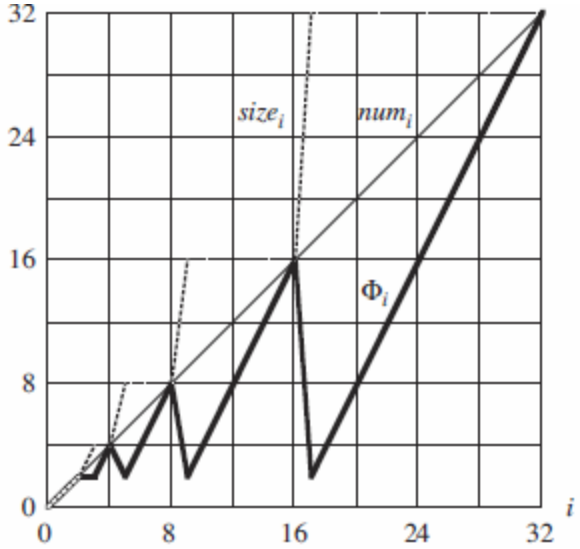
Dynamic Table Expansion

- Using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3. Intuitively, each item pays for 3 elementary insertions: (1) inserting itself into the current table, (2) moving itself when the table expands, and (3) moving another item that has already been moved once when the table expands.
- Consider the case when the table size is m immediately after expansion – the table has m/2 items (m is a power of 2) and has no credit at this point. We charge 3 dollars for each insertion. The elementary insertion that occurs immediately costs 1 dollar. We place another dollar as credit on the item inserted (for its eventual copy into a bigger table). We place the third dollar as credit on one of the m/2 items already in the table (for its eventual copy). The table will not fill again until we have inserted another m/2 – 1 items, and thus, by the time the table contains m items and is full, we will have placed a dollar on each item to pay to reinsert it during the expansion.
- We can use the potential method to analyze a sequence of n table-insertions. Consider a potential function Φ that is 0 immediately after an expansion but builds to the table size by the time the table is full, so that we can pay for the next expansion by the potential. Consider the function **$\Phi(T) = 2Tnum - Tsize$**. We observe
  - $\Phi(T) = 0$; Immediately after an expansion, we have Tnum = Tsize/2, so $\Phi(T) = 0$, as desired.
  - Since table is always at least half full, Tnum $\geq$ Tsize/2 $\Rightarrow \Phi(T)$ is always +ve.
- Thus, the sum of the amortized costs of n TABLE-INSERT operations gives an upper bound on the sum of the actual costs.
- We still need to compute the amortized cost of the i^{th} insert operation.

Dynamic Table Expansion

- Let num_i, $size_i$, Φ_i denote number of items in the table, size of the table and potential function respectively after the i-th operation. Initially $num_0 = size_0 = \Phi_0 = 0$. we need to consider two cases:
  - i^{th} operation does not trigger an expansion: then $size_i = size_{i-1}$ and the amortized cost of the operation is $ac_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) = 1 + (2num_i - size_i) - (2(num_i - 1) - size_i) = 3$
  - If the i^{th} operation does trigger an expansion, then we have $size_i = 2size_{i-1}$ and $size_{i-1} = num_{i-1} = num_i - 1$, which implies that $size_i = 2(num_i - 1)$. Thus, the amortized cost of the operation is $ac_i = c_i + \Phi_i - \Phi_{i-1} = 1 = num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) = num_i + (2num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) = num_i + 2 - (num_i - 1) = 3$

- The thin line shows num_i, the dashed line shows $size_i$, and the thick line shows Φ_i .
- Notice that immediately before an expansion, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table.
- Afterwards, the potential drops to 0, but it is immediately increased by 2 upon inserting the item that caused the expansion.



Dynamic Table Expansion & Contraction

- Simple deletion is easily done by Tnum --. We also want to contract the table when the load factor becomes too small; such contraction is somewhat analogous to expansion – when the number of items in the table drops too low, we allocate a new, smaller table and then copy the items from the old table into the new one. As before, we want
  - Load factor is bounded below by a positive constant
  - Amortized cost of a single operation is bounded above by a constant.
- To mirror the strategy, we used for table expansion, a possible strategy for contraction could be to halve the size when a deleting an item would cause the table to become less than half full, guaranteeing that the load factor of the table never drops below ½.
- This strategy looks simple but does not satisfy the second requirement. Consider a scenario: we perform n operations on an initially empty table where n is a power of 2. The first n/2 operations are insertions, with a total cost of $\Theta(n)$ and Tnum = Tsize = n/2. The next n/2 operations is given by the sequence, insert, delete, delete, insert, insert, delete, delete, insert, insert, ...
- The first insertion causes the table to expand to size n. The next two deletions cause the table to contract back to size n/2. Two further insertions cause another expansion, and so forth. The cost of each expansion and contraction is $\Theta(n)$ and there are $\Theta(n)$ of them. Thus, the total cost of the n operations is $\Theta(n^2)$, making the amortized cost of a single operation $\Theta(n)$.
- The downside of this strategy is obvious: after expanding the table, we do not delete enough items to pay for a contraction. Likewise, after contracting the table, we do not insert enough items to pay for an expansion.

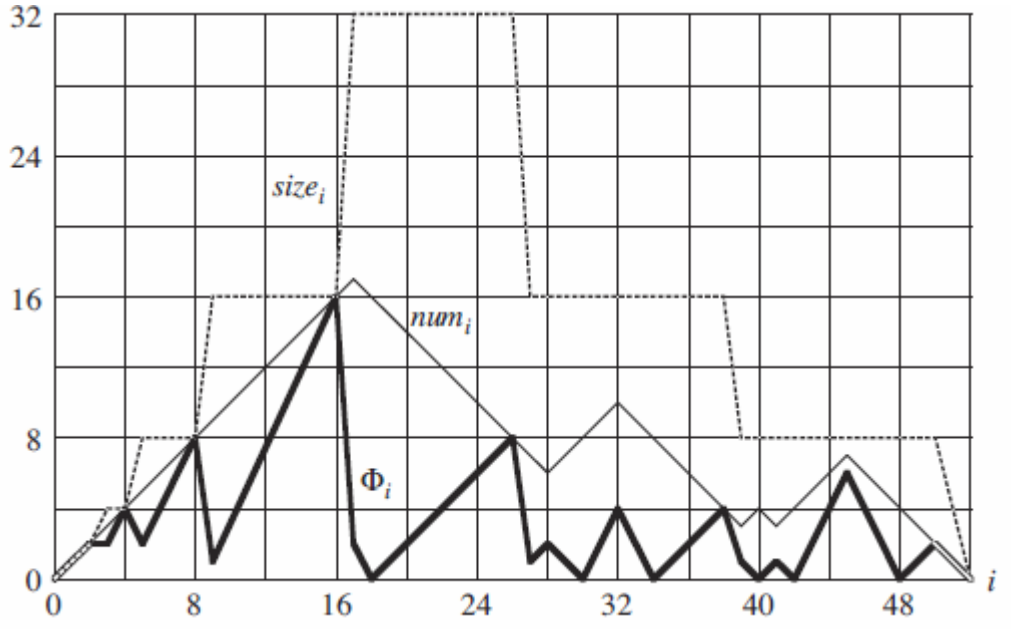
Dynamic Table Expansion & Contraction

- New strategy: We allow the load factor of the table to drop below ½. Specifically, we continue to double the table size upon inserting an item into a full table, but we halve the table size when deleting an item causes the table to become less than ¼ full, rather than ½ full as before. The load factor of the table is therefore bounded below by the constant ¼. The deletion algorithm is analogous to the insertion one.
- *Why does it solve the previous problem*? The problem was that no credit was available at the time of contraction, hence amortized cost was not constant. The new scheme solves the problem. The idea is at the beginning, after both expansion and contraction the potential ideally should be zero. Thus, we will need a potential function that has grown to Tnum by the time that the load factor has either increased to 1 or decreased to ¼. After either expanding or contracting the table, the load factor goes back to ½ and the table's potential reduces back to 0. Note: inventing an appropriate potential function for an application is the most important job.
- Remember α(T) = Tnum/Tsize; for an empty table Tnum = Tsize = 0 and α(T) = 1. Here is one possible potential function we use.

$$\Phi(T) = \begin{cases} 2Tnum - Tsize, & \text{if } \alpha(T) \geq 1/2 \\ Tsize/2 - Tnum, & \text{if } \alpha(T) < 1/2 \end{cases}$$

- **Observe** that the potential of an empty table is 0 and that the potential is never negative. Thus, the total amortized cost of a sequence of operations with respect to Φ provides an upper bound on the actual cost of the sequence.

Dynamic Table Expansion & Contraction



- When the load factor is ½, the potential is 0.
- When the load factor is 1, Tsize = Tnum, or Φ(T) is Tnum and potential is enough to pay for an expansion.
- When the load factor is ¼, Tsize = 4Tnum or Φ(T) = Tnum; thus potential can pay for contraction if the next operation is a deletion.

- The effect of a sequence of n Insert and delete operations on the number num_i of items in the table, the number $size_i$ of slots in the table, and the potential Φ_i each measured after i^{th} operation.
- The thin line shows num_i, the dashed line shows $size_i$, and the thick line shows Φ_i

$$\Phi_i = \begin{cases} 2 \cdot num_i - size_i & \text{if } \alpha_i \geq 1/2 , \\ size_i/2 - num_i & \text{if } \alpha_i < 1/2 , \end{cases}$$

Dynamic Table Expansion & Contraction

- Now, consider a sequence of n insert/delete operations. Initially, $num_0 = 0$, $size_0 = 0$, $\alpha_0 = 1$ and $\Phi_0 = 0$.
- Consider the case when i^{th} operation is an insert; $num_i = 1 + num_{i-1}$
  - If $\boldsymbol{\alpha_{i-1} \geq ½}$, the analysis is exactly as before; whether the table expands or not, the amortized cost ac_i of the operation is at most 3.
  - If $\boldsymbol{\alpha_{i-1} < ½}$, the table cannot expand as the result of i^{th} operation; the table expands only when $\alpha_{i-1} = 1$. There are 2 subcases.
    - $\boldsymbol{\alpha_i < ½}$: $ac_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) = 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) = 0$.
    - $\boldsymbol{\alpha_i \geq ½}$: $ac_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (2num_i - size_i) - (size_{i-1}/2 - num_{i-1}) = 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) = 3num_{i-1} - (3/2)size_{i-1} + 3 = 3\alpha_{i-1}size_{i-1} - (3/2)size_{i-1} + 3 < (3/2)size_{i-1} - (3/2)size_{i-1} + 3 = 3$.

  Thus, the amortized cost of an insert operation is at most 3.
- Consider the case when i^{th} operation is a delete; $num_i = num_{i-1} - 1$.
  - $\boldsymbol{\alpha_{i-1} < ½}$**:** There are two possibilities:
    - Table does not contract after i^{th} operation: $size_i = size_{i-1}$; $ac_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) = 2$.
    - Table does contract after i^{th} operation: since copying is involved, actual cost $c_i = num_i + 1$, $size_i = size_{i-1}/2$, $num_{i-1} = size_{i-1}/4$. : $ac_i = c_i + \Phi_i - \Phi_{i-1} = (num_i + 1) + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) = 1$.
  - $\boldsymbol{\alpha_{i-1} \geq ½}$**:** No table contraction; $size_i = size_{i-1}$; $ac_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (2numi - sizei) - (2num_{i-1} - size_{i-1}) = 3$. Thus, the amortized cost of a delete operation is at most 3.