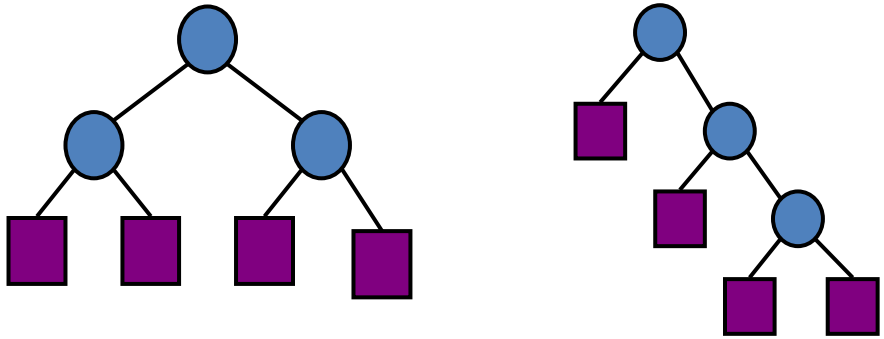
Balanced Binary Search Trees

Quick Recap of Simple BSTs☆

- **Notation**: **n**: number of nodes, **e**: number of external nodes, **i**: number of internal nodes, **h**: height.
- **Properties:**
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$



A **full binary tree** (sometimes proper **binary tree** or 2-**tree or a strict tree**) is a **tree** in which every node other than the leaves has two children.
A **complete binary tree** is a **binary tree** in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

- **Traversals**:
  - Preorder: visit, left, right
  - Inorder: left, visit, right
  - Postorder: left, right, visit

They are all O(n) operations

- The **depth** of a node v is the number of ancestors of v, excluding v itself. Note that this definition implies that the depth of the root of T is 0. The depth of a node v can also be recursively defined as follows: (1) If v is the root, then the depth of v is 0; (2) otherwise, the depth of v is one plus the depth of the parent of v. The **height** of a node v is (1) 0, if v is an external node and (2) is one plus the maximum height of a child of v.
- **Creation, Insertion, deletion, Look-up, Find_max, Find_min, and successor/predecessor** (with respect to a specific traversal); they are all O(h) operations; in the worst case O(n).
- **Applications**: Build trees for arithmetic expressions, print/evaluate expression trees, Range Queries, Trimming a BST

Binary Search Tree (BST)[*]

- A binary search tree is organized as a binary tree with some special property. We represent such a tree by a linked data structure in which each node is a *struct*. In addition to a **key** (for simplicity we assume it's an integer; it can be anything just like in any kind of linked lists), each node contains two pointers **left** and **right**; usually no information is explicitly stored about the parent of a node; if left (right) child does not exist, the corresponding link contains a NULL.

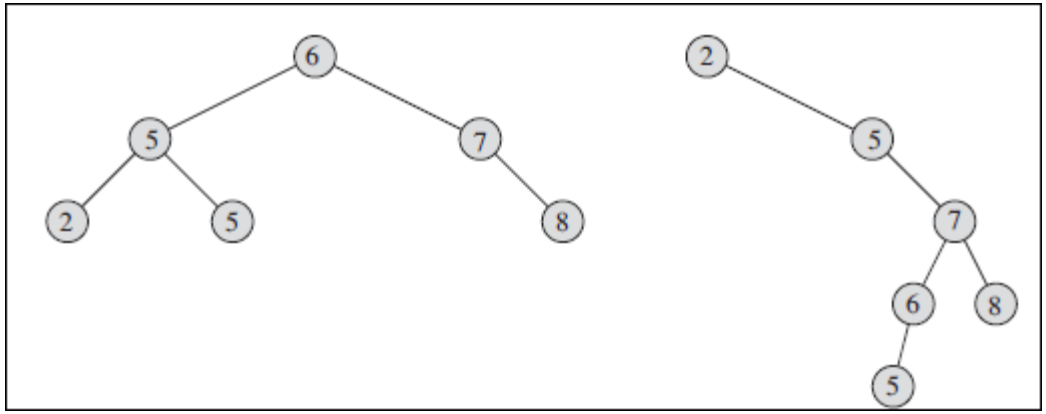
  typedef struct node{struct node *left; int key; struct node *right;} node;

- The keys in a binary search tree are always stored in such a way as to satisfy the binary-search-tree property: "Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key ≤ x.key. If y is a node in the right subtree of x, then y.key ≥ x.key."

- We can have many different BSTs with the same set of nodes [computing how many is outside our scope in this class]. Any given BST can be specified by the address (pointer) of the root; an empty tree is represented by the NUILL pointer (Create an empty BST using t_node *Root =NULL;)

Observe:

- Each node has a unique path to the root of the tree.
- The maximum of the length of all such paths is called the height of the tree.
- # nodes = # edges +1 (it is true for any tree, binary or not); **why?**
- In the worst case, height of the tree can be n – 1; example??



Complete Binary Tree, Balanced BST, Weight and Height Balanced BSTs – Later if time permits.

[*]Refreshing from 2120

Operations on BSTs[*]

- **Set / Dictionary**: maintain dynamic collection of elements
  - Insert (k) : insert key k
  - Remove (k) : remove (delete) key k
  - Find (k) : is key k present?
  - Traversals [will soon define]
- **Map:** maintain collection of (key, value) pairs
  - Insert (k, v) : insert key k with associated value v
  - Remove (k) : remove record with key k
  - Find (k) : return value associated with key k (or report that k isn't in the structure)
- **How to store a BST?**
  - We can maintain a BST by using a sorted/unsorted array, or linked (singly/doubly) list, a hash table [Later] and others – all depends on what we want to do in the applications.
  - Usually, we use a struct like this: **typedef Struct node{struct node *left; int key; struct node *right;} node;** Remember, we can have many other info in place of key alone; say in stead of value [see above] we can have a pointer pointing to a record of a file. Advantage: we can change the file structure without disturbing the BST.
  - To create an empty tree, we use node *root = NULL [Just like a linked list, the pointer variable root is a gateway to the entire tree.
  - Sometimes, we can add another component to the structure like struct node *parent to keep the information of the parent of a node in a BST.

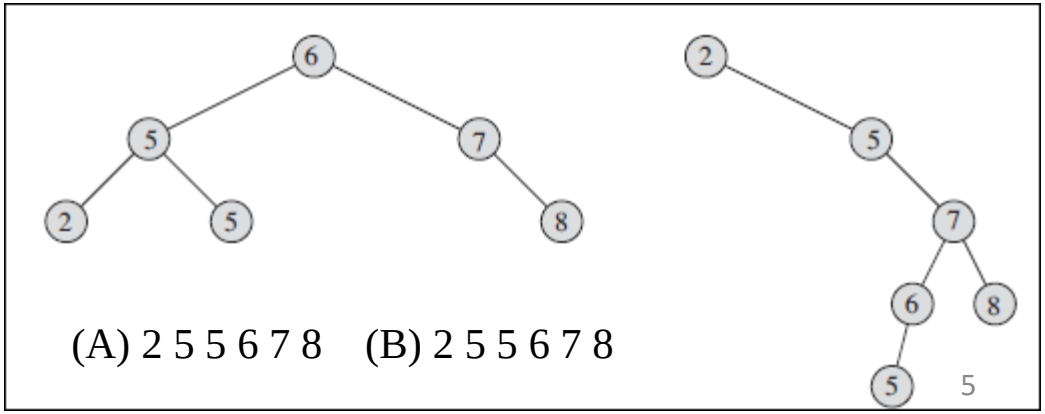
Use different names if you want to design .h file of all structures you use.

[*]Refreshing from 2120

Search, Insert, Traversals[*]

- **Search** a BST T (Tree root pointer) for a given key (Find the tree node containing key k) [will return the pointer to the desired node]: Execution time is O(n) in the worst case Observe
  - If the tree is empty or the key is not found, the function returns NULL.
  - If the tree is not empty, then compare the search key k with the key of the node; if equal, search is successful; otherwise go down either the left link or right link depending on if k is less or $\geq$. One can easily write a recursive routine [or iterative routine in C or C++]
- **Insert** a node with key k in T: Search for k; if found exit with an error; otherwise when search returns a NULL, create (malloc) a new node, adjust the fields. One can easily write a recursive routine [or iterative routine in C or C++]
- Tree **Traversals**: Enumeration [visiting (printing) once] of the contents of the nodes of a BST in some order. (1) **Inorder**: left, root, right; (2) **Preorder**: root, left, right; (3) **Postorder**: left, right, root;
- Pseudocode (recursive) for Inorder traversal: An inorder traversal prints the elements in sorted order in O(n) steps. Let us see examples.

```
Inorder(T):
if T == NULL, then return
Inorder(T->left)
print T->key
Inorder(T->right)
```



(A) 2 5 5 6 7 8 (B) 2 5 5 6 7 8

[*]Refreshing from 2120

Search, Insert, Max, Min, Traversals (in C)[*]

```
node *search (node **lp, int key){
  node *current = *lp;
  while (current != NULL && current->info != key)
   if (key < current->info) current = current->left;
   else current = current->right;
 return (current); }// One can check if the key exists.
```

Note: One can maintain a stack of pointers to keep track of the nodes while searching. Will be useful later.

```
void insert (node **lp, int key){
   node *current, *prev, *new;
   current = *lp; prev = NULL;
    while (current != NULL) { prev = current;
    if (key < current->info) current = current->left;
    else current = current->right; }
     new = (node *)malloc(sizeof(node));
        new->info = key;
        new->left = new->right =NULL;
        if (prev == NULL) *lp = new; else {
        if (key < prev->info) prev->left = new;
        else prev->right = new; } }
```

[*]Refreshing from 2120

```
int min (node **lp){
node *x =*lp;
while (x->left != NULL)
   x = x->left;
return (x->info);
} //Assuming the node is at least one node.
```

Note: int max is obtained by replacing left with right.

```
void inorder_recursive (node **lp){

node *x = *lp;
if (x != NULL)
{
inorder_recursive (&x->left);
printf ("%d  ", x->info);
inorder_recursive (&x->right);
} }
```

Note: preorder and post order traversals are similar.

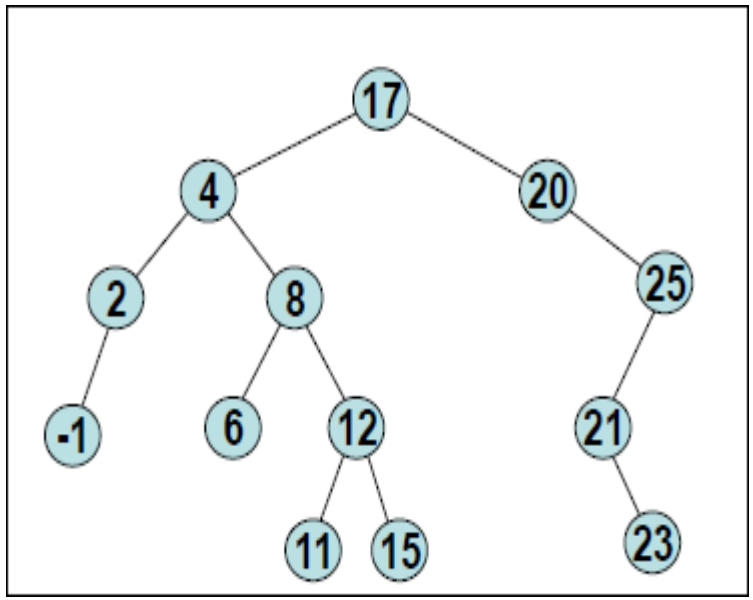
Q: How to write non recursive traversals? [Use stacks to remember the path!]

Predecessor & Successor[*]

- Predecessor (Successor) of any node in a given binary search tree is usually defined in terms of the unique inorder traversal of the tree; the node may be specified as a key or the address of the node (depending on application requirement) – accordingly, the function (method) has to be adjusted – the principle remains the same.

We'll study some scenarios first to understand the logic.

- inorder traversal: -1 2 4 6 8 11 12 15 20 21 23 25
- Say, key is 4 ⇒ its right link is not null ⇒ 4's successor must be the minimum of the node 8, which is 6.
- Say key is 11 ⇒ right link is null ⇒ its successor must be somewhere up in the tree, go to the predecessor, say node p ⇒ two cases:
- Case 1: key is NOT on its predecessor's right link ⇒ the key of p is key's successor (if p is not null) [i.e., 12 is successor of 11]
- Case 2: if p is not NULL and key is on its predecessor's right link, go up the chain and repeat.
- Be careful to handle the maximum key in the BST, that does not have a successor



Writing a function (method) for successor is relatively straightforward – remember to maintain a stack of pointers when searching for the tree [necessary to move up]; actually, stack is not needed; one can do it without a stack.

Note: Predecessor is similar to successor – left links will replace the right links.

[*]Refreshing from 2120

Successor of a node[*]

```
int successor (node **lp, int key){

node *root = *lp, *succ = NULL, *keynode = search (&root, key);
// if right subtree is not null
if (keynode->right != NULL) return (min (&keynode->right));
//printf ("HERE min is %d\n", min (&keynode->right));

// Start from root and search down the tree

while (root != NULL)
  {
  if (keynode->info < root->info)
    { succ = root; root = root->left;}
  else
     if (keynode->info > root->info) root = root->right;
  else  break;

  }
if (succ == NULL) return -100;
else return succ->info;
}
```

Note: if one needs to have a function that returns the address of the successor of the info of the successor node, one needs to redefine the function as
node *successor_node ((node **lp, int key);

Observe the same code will work up until the last if statement; the last if statement needs be replaced by return succ !!

One can adjust the code in a relatively straightforward way to get a function for the inorder predecessor; try that.

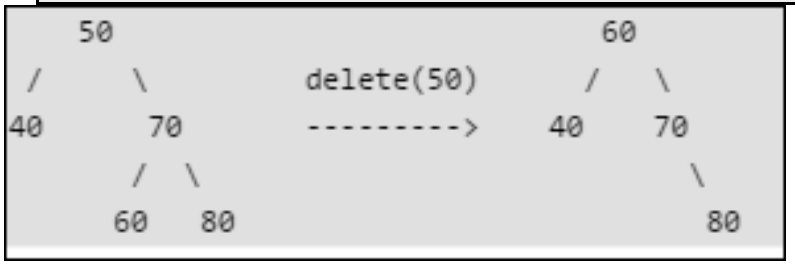
[*]Refreshing from 2120

Delete*

- Removing a node from a BST is a bit more tricky, since we do want to make sure that the BST remains a BST after the deletion. If the node has one child then the child is spliced to the parent of the node. If the node has two children then its successor has no left child; copy the successor into the node and delete the successor instead. TREE-DELETE (T, z) removes the node pointed to by z from the tree T. There are 3 possibilities:
  - **The node to be deleted is a leaf node**:
    Easiest: Just delete it; that's all [Be careful when deleting the root]
  - **Node to be deleted has only one child (left or right):**
    Connect the only subtree of the node to the parent of the node to be deleted and delete.
  - **Node to be deleted has two children**:
    Tricky: Find inorder successor of the node. Copy contents of the inorder successor to the node (to be deleted) and delete the inorder successor.







- Note that in the third case inorder successor of the node, say z, (to be deleted) is the minimum of the node z->right.

*Refreshing from 2120

C Code*

```
void delete (node **lp, int key){
node *root = *lp;
//*succ = NULL, *keynode = search (&root, key);
// base case: the tree is empty.
if (root == NULL) return;
// if key is less than root, key is on root's left subtree
  if (key < root->info) delete (&root->left, key);
// if key is greater than root, key is on root's right subtree
  else if (key > root->info) delete (&root->right, key);
// if key is equal to root, root is to be deleted.
  else
  {//node with one or no child
  if (root->left == NULL) {*lp = root->right; free
(root);return;}
  else if (root->right == NULL) {*lp = root->left; free
  (root);return;}

// node with two children; get inorder successor
node *temp =  minnode (&root->right);
//copy the inorder successor's content to this node
  root->info = temp->info; *lp = root;
  delete(&root->right, temp->info);
  } }
```

We have used a helper function (a variation of the min function we have seen before.

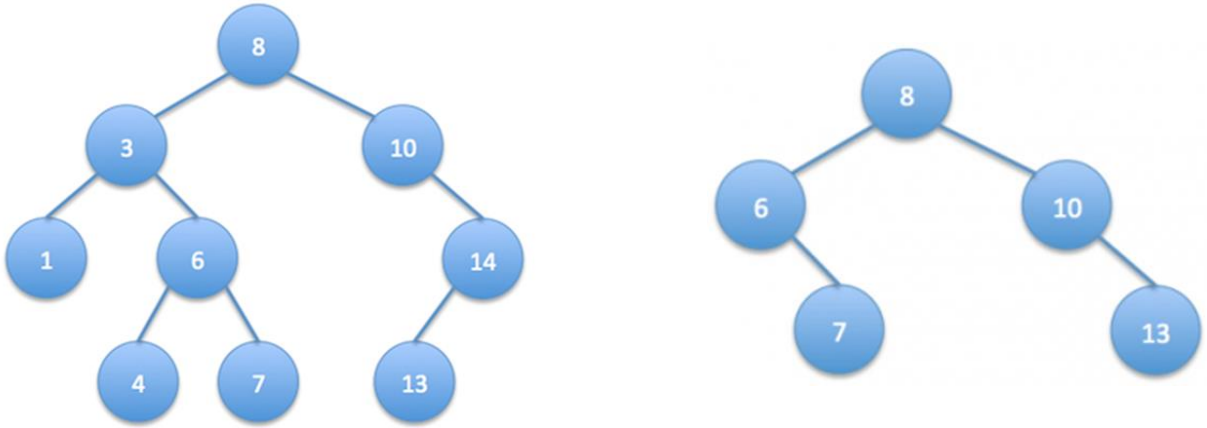
```
node *minnode (node **lp){
node *x =*lp;
while (x->left != NULL)
   x = x->left;
return x;
}
```

Note: It is not very difficult; try to understand the logic as explained by the example and the comments embedded in the code. And yes, it is a recursive routine; the iterative version would be a bit more complicated. If you understand the logic, you should be able to do the deletions by hand on small examples. Practice with several arbitrary BSTs.

*Refreshing from 2120

Warm-up: Trim a BST

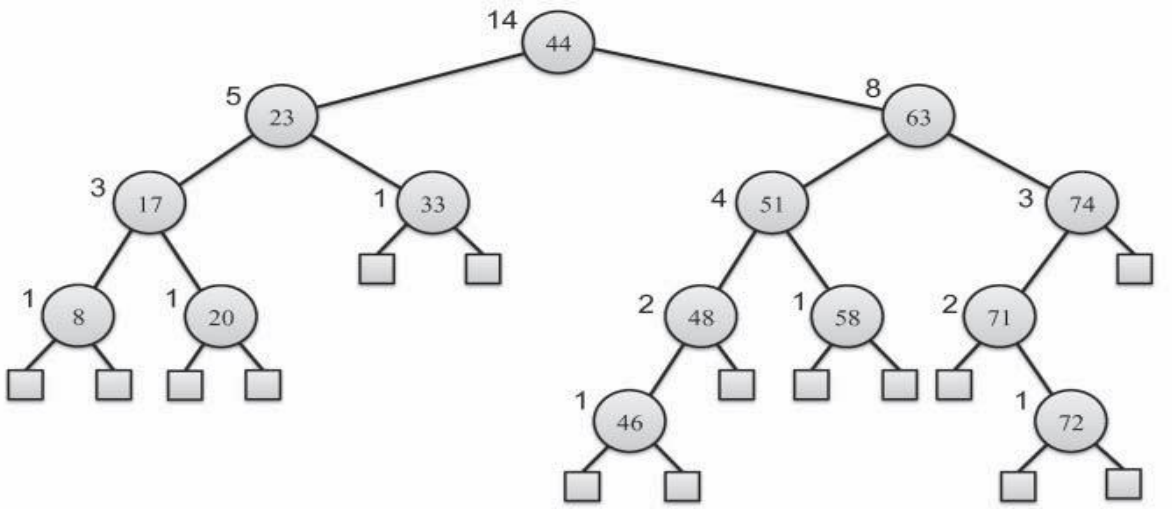
- **Problem Statement:** Given the root of a binary search tree and 2 numbers min and max, trim the tree such that all the numbers in the new tree are between min and max (inclusive). The resulting tree should still be a valid binary search tree. Given the BST on the left and min = 5, max =13, the resulting BST is shown on the right.



- **The Approach**: We should remove all the nodes whose value is not between min and max. We can do this by performing a post-order traversal of the tree. We first process the left children, then right children, and finally the node itself. So we form the new tree bottom up, starting from the leaves towards the root. As a result, while processing the node itself, both its left and right subtrees are correctly trimmed binary search trees (may be NULL as well).
- **Note:** The complexity of this algorithm is O(N), where N is the number of nodes in the tree, because we basically perform a post-order traversal of the tree, visiting each and every node once.
- Write the code and experiment – it will refresh your memory about simple BSTs

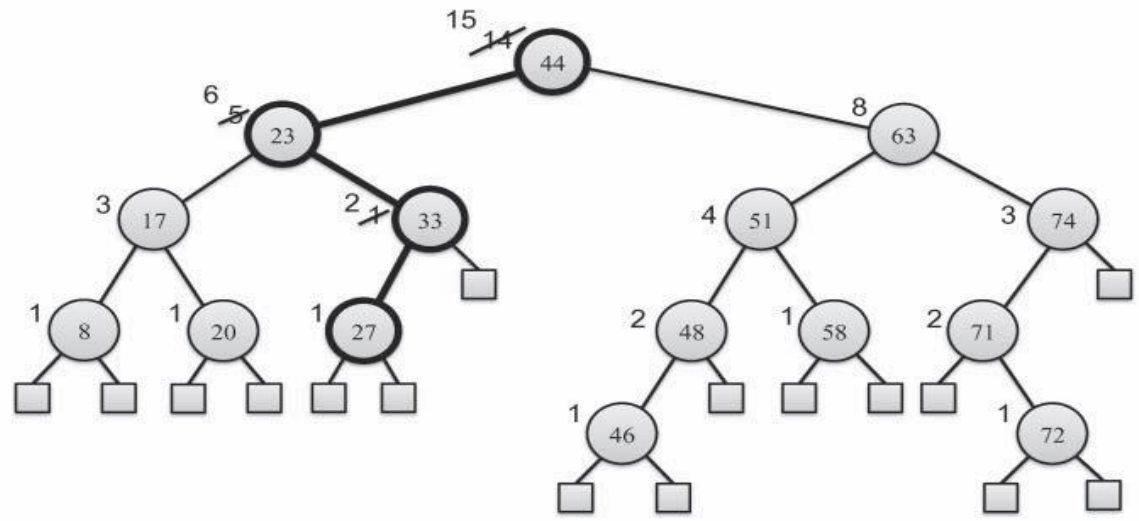
Index-Based Searching in a Simple BST

- We know that a sorted array allows us to quickly identify the i^{th} smallest item in the array simply by indexing the cell A[i]. While updating a BST is much more efficient, we lose the said ability. We can regain that easily at the expense of an extra field in the BST node structure:
- typedef struct node {struct node *left; int key; int size; struct node *right;} node; where size is the number of nodes rooted at the node.



- This can be easily done during creation, updating (insertion, deletion etc.) without any extra cost in asymptotic complexity.
- This will allow to execute index-based search in O(h) time. Design the algorithm, write the code, analyze the complexity and experiment.

Index-Based Searching in a Simple BST



Insert a node with key 27 in the tree

Algorithm TreeSelect(i, v, T):

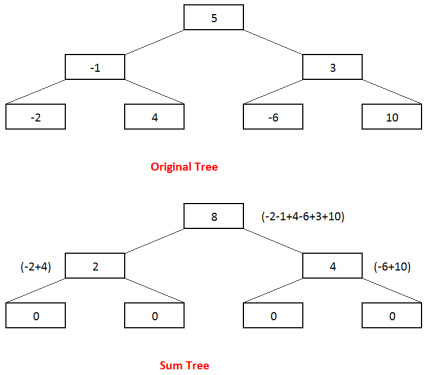
Input: Search index i and a node v of a binary search tree T

Output: The item with i^{th} smallest key stored in the subtree of T rooted at v

```
Let w ← T.leftChild(v)
if i ≤ size then
        return TreeSelect(i, w, T)
else if i = size + 1 then
        return (key(v), element(v))
else
return TreeSelect(i − size − 1, T.rightChild(v), T)
```

Practice Problems

1. Convert binary tree to its Sum tree: Sum tree of a binary tree, is a binary tree where each node in the converted tree will contain the sum of the left and right sub trees in the original tree.
2. We note that binary tree leaf nodes have lots of nulls – we want to utilize those spaces to expedite traversals, e.g., doing traversals without using recursion or stacks.

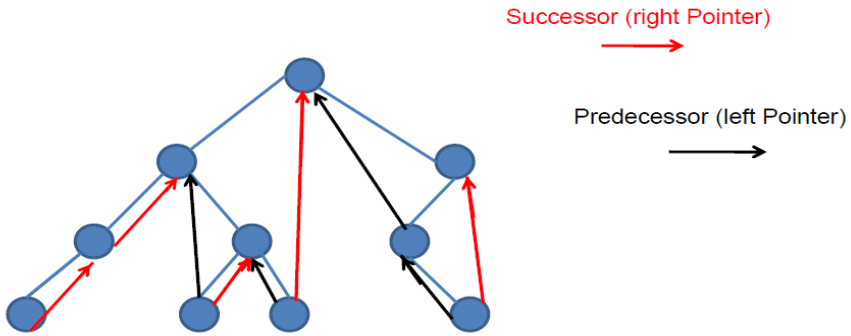


Threaded Binary trees: We have the pointers reference the next node in an inorder traversal; called threads; we need to know if a pointer is an actual link or a thread, so we keep a Boolean flag for each pointer.

Single Threaded: each node is threaded towards either the in-order predecessor or successor (left **or** right) means all right null pointers will point to inorder successor **OR** all left null pointers will point to inorder predecessor.

Double Threaded**:** each node is threaded towards both the in-order predecessor and successor (left **and** right) means all right null pointers will point to inorder successor **AND** all left null pointers will point to inorder predecessor.

Write codes for traversals without using recursion or explicit stack(s), in a threaded tree. Also, given a tree, write code to make it single or double threaded.



Traversal by Link Reversal (Non-Recursive and Stack less)

- In a link-reversal traversal, as we descend into a binary tree, we replace normally downward pointing pointers in the link fields of nodes with upward pointers, which point to immediate ancestors. On the way up again, we restore pointer fields to their original condition containing downward pointers. Thus, as the algorithm traces out a path down into the lower parts of the tree, it leaves a "reverse" path to use to climb back out again.
- Assume that each node x contains an additional flag field such that initially flag(x) = 0.

```
typedef struct node (struct node *left, int info, bool flag, struct node *right);
```

- The flag field is set to 1 when the right field is set to point upwards to its parent. It is reset to 0 when the right field is reset to point downward to its original right child node.
- The flag field is set to 1 when the right link field is set to point upwards to its parent. It is reset to 0 when the right link field is reset to point downward to its original right child node.
- We will write a composite traversal algorithm pseudocode (Non-Recursive and Stack less) using this slightly modified node structure.

Composite Traversal Pseudocode

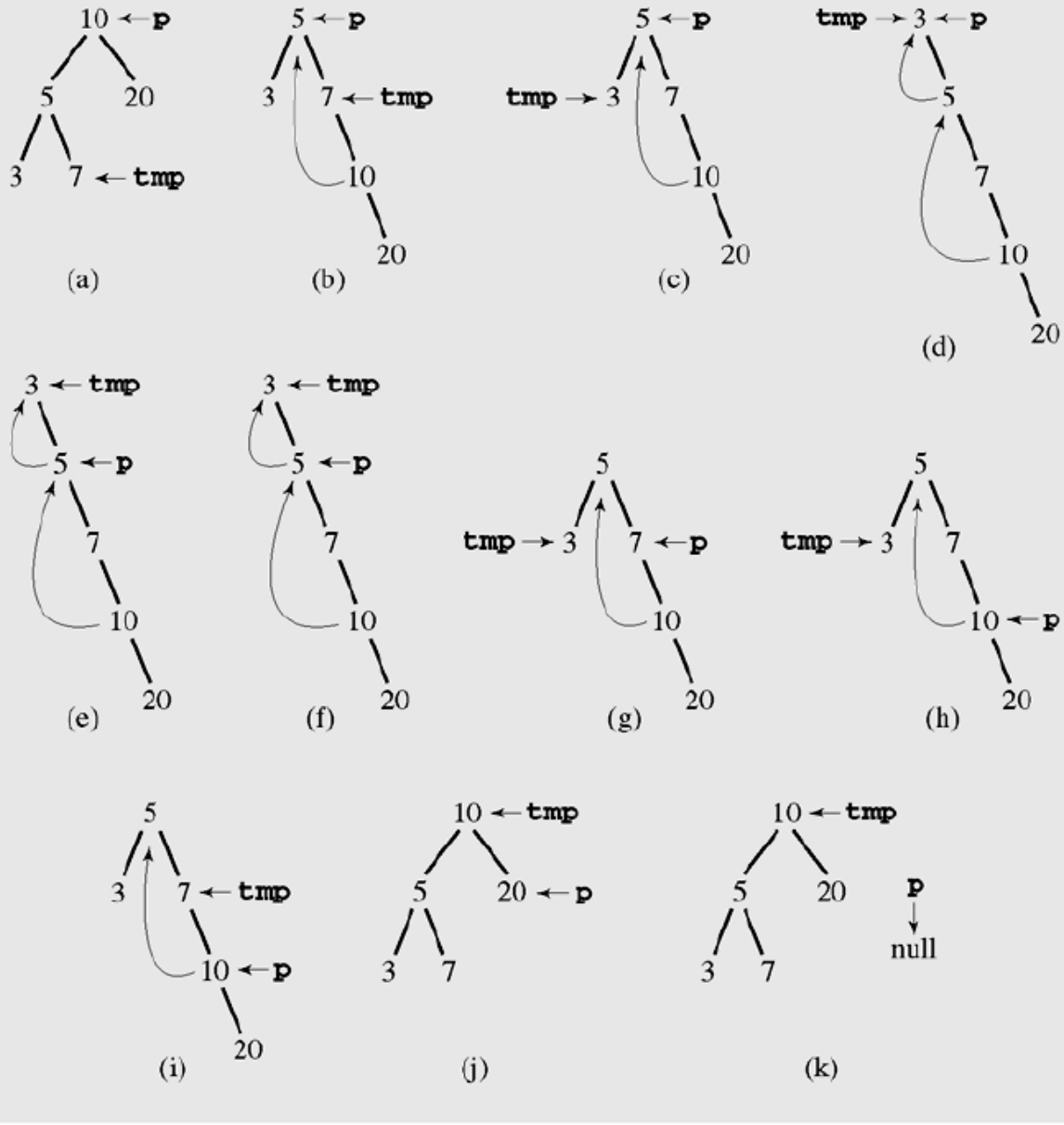
1. **[Initialize.]** Set PRES to point to the root of the tree, and set PREV = Null
2. **[Preorder visit.]** Visit node PRES if traversing in preorder.
3. **[Descend left.]** Set NEXT= left(PRES). If NEXT ≠ Null, then set left(PRES) = PREY, PREV = PRES, PRES =NEXT, and go to step 2.
4. ***[Symmetric order visit.]*** Visit node PRES if traversing in symmetric order.
5. **[Descend right.]** Set NEXT= right(PRES). If NEXT ≠ Null, then set TAG(PRES) = 1, right(PRES) = PREV, PREV = PRES, PRES = NEXT, and go to step 2.
6. **[Postorder visit.]** Visit node PRES if traversing in postorder.
7. **[Go up.]** If PREY= A, then the algorithm terminates. Otherwise, if TAG(PREV) = 0, then set NEXT = left(PREV), left(PREV) = PRES, PRES = PREV, PREV = NEXT, and go to step 4. Otherwise, set NEXT = right (PREV), TAG(PREV) = 0, right(PREV) = PRES, PRES = PREV, PREY = NEXT, and go to step 6.

Non-Recursive Inorder minus the extra memory

```
void inorder(node **lp){

node *parent =*lp, *tmp;
while (parent != NULL) // The outer loop
  {if (parent->left ==NULL)
     {printf("%d  ",parent->info); parent = parent->right;}
/* Find the inorder predecessor of parent */
  else
    {tmp = parent->left;
     while(tmp->right != NULL && tmp->right != parent)
          tmp = tmp->right;
/* Make parent as right child of its inorder predecessor */
     if (tmp->right == NULL){tmp->right=parent;
                parent = parent->left;}
/* Revert the changes made in if part to restore the original
       tree i.e., fix the right child of predecessor */
     else
     {printf("%d  ", parent->info); tmp->right=NULL;
                parent=parent->right; }      }     }   }
```

An Example Execution



Balanced BSTs

- Real-time systems are computational platforms that have real-time constraints, where computations must complete in a given amount of time, e.g., antilock braking systems on cars, video and audio processing systems, operating systems kernels, and web applications.
- Most frequently used operations on stored data are random insertions, deletions, look-up, neighborhood search based on keys of the searched data records – BSTs perform those operations more efficiently than linear arrays (sorted on keys).
- The problem is that without some way of limiting the height h of T, in the worst-case h = O(n); the worst-case running time for performing searches and updates in T can be linear in the number of items it stores. Indeed, this worst-case behavior occurs if we insert and delete keys in T in a somewhat sorted order, which is likely for a database. This results in poor performance for searches and updates.
- One needs a way of restructuring a binary search tree while it is being used so that it can guarantee logarithmic-time performance for searches and updates. These restructuring methods [to be incorporated in update operations, e.g., insert and delete] result in a class of data structures known as **balanced binary search trees**.
- There are different ways a BST can be balanced (to maintain a logarithmic height) – we will study some of those techniques.

Ranks and Rotations

- The primary way to achieve logarithmic running times for search and update operations in a binary search tree, T, is to perform restructuring actions on T based on specific rules that maintain some notion of "balance" between sibling subtrees in T. Intuitively, the reason balance is so important is that when a binary search tree T is balanced, the size of the left and right subtrees of any node is "roughly" equal – that's the goal.
- We need a metric for "balance" – there are many (same in principle, but different in implementation) – we will assume the metric is (rank) height (h) of a tree (or a subtree) – we will augment the node structure as follows:

  typedef struct node {struct node *left; int key; **int h**; struct node *right;} node;
- We will say a tree (or a subtree) rooted at a node is **balanced iff the difference in height of its left and right subtrees is ≤ 1** (will show later that it will make the height of the tree always $O(\log_2 n)$. [There are other ways to define rank of a node; all of them are related to height in some ways]. Such a tree is called an **AVL tree** [other examples are **red-black trees**, **weak AVL trees, B-trees, B$^+$ trees,** etc.]
- Balance in such a tree, T, is enforced by maintaining certain rules on the relative ranks of children and sibling nodes in T. One restructuring operation, which is used in all balanced binary search trees called a **rotation**; there are 4 types of rotations.
- We will use a unified restructuring operation, called **trinode restructuring**, which combines the four types of rotations into one action.

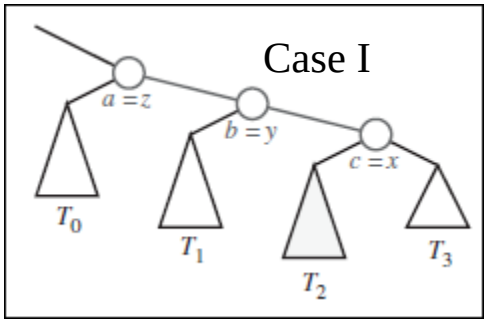
How to know a BST is an AVL tree

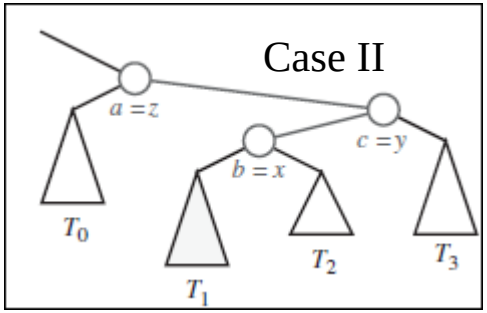
- A null BST is also an AVL tree; a non null BST is an AVL tree iff the difference between the heights of its left and right subtrees ≤ 1. Easy to write a recursive routine; O(n) execution time (one needs to visit each node exactly twice starting from the root), much like any of the traversals; also, easy to write non recursive version as well.

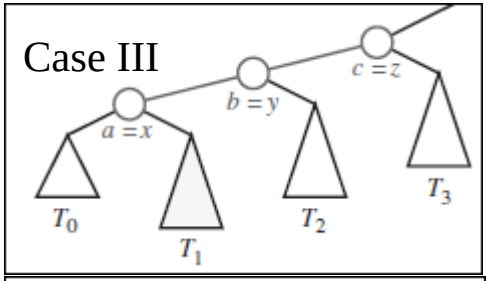
```
bool isBalanced (node **lp){node *root = *lp;
if (root == NULL) return 1;
lh = ht(root->left); rh = ht(root->right);
if  (abs(lh – rh) <= 1 && isBalanced(root->left)
    && isBalanced(root->right)) return 1;
return 0;
//NEED a helper function to compute the height of a tree; easy.
```

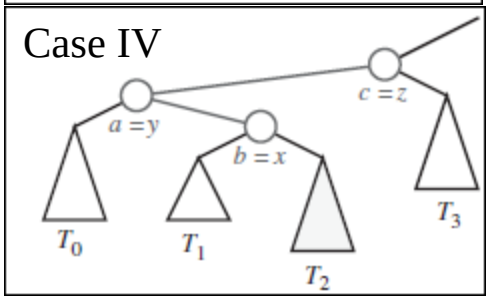
- An empty tree has a height = 0; otherwise, height = 1 + max of left height and right height.

Implementation of Rotations





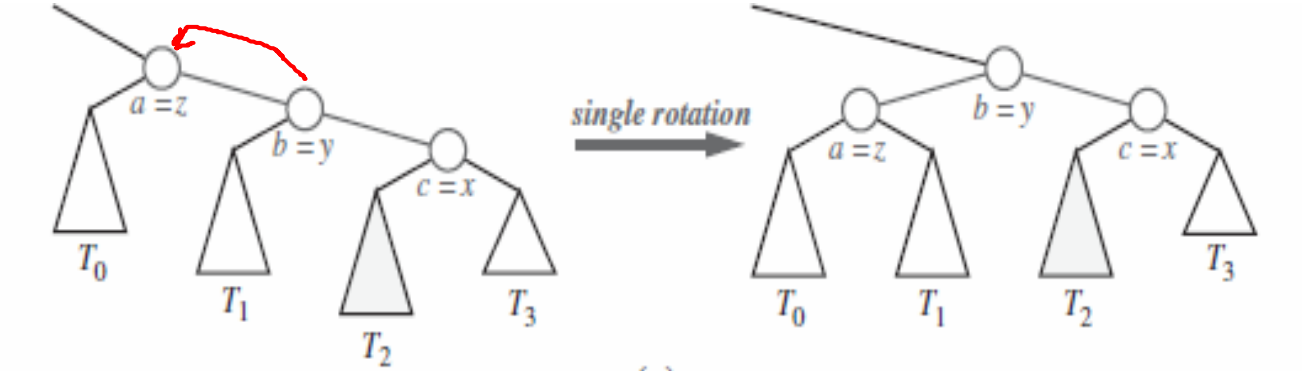




Consider a **node x** of a binary search tree T that has both a **parent y** and a **grandparent z** [x, y, z are node pointers]. Why do we need to consider this? [This is a necessary condition for a tree to be unbalanced] There are only 4 possibilities; why? Consider node z ⇒ y can be either left or right child; for each choice of y, there are two choices of x (T is a BST).

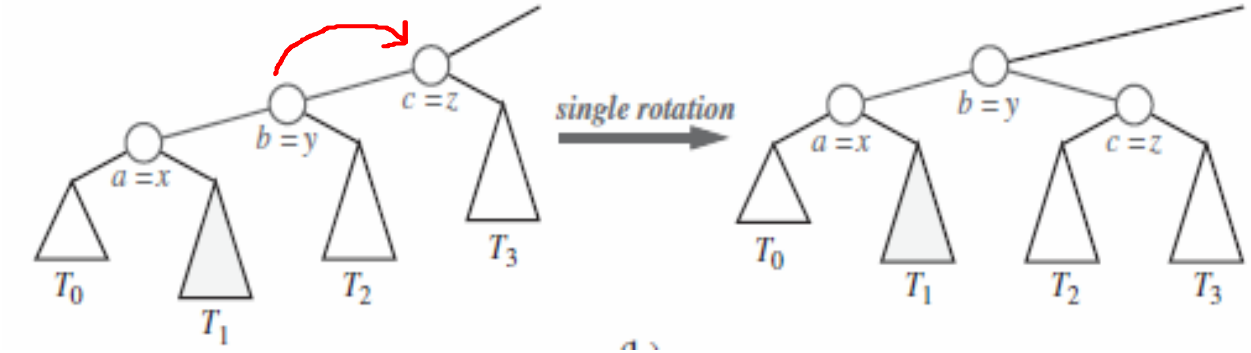
- An insertion/deletion in an AVL tree may destroy the height-balance property at some node(s) ; we need to rebalance the tree with minimum effort to make it AVL again – this is called **restructuring** the tree.
- The trinode restructuring operation involves a node, x, which has a parent, y, and a grandparent, z.
- At a high level, a trinode restructure temporarily renames the nodes x, y, and z as a, b, and c, so that a precedes b and b precedes c in an inorder traversal of T. There are four possible ways of mapping x, y, and z to a, b, and c, as shown.
- The trinode restructure then replaces z with the node called b, makes the children of this node be a and c, and makes the children of a and c be the four previous children of x, y, and z (other than x and y) while maintaining the inorder relationships of all the nodes in T.
- We need 4 different kinds of rotation in 4 different situations; the relabeling is just a mechanism to handle necessary modifications by a single function.

Schematic illustration of a trinode restructure operation



Also known as right-right case or Left-rotate(z)

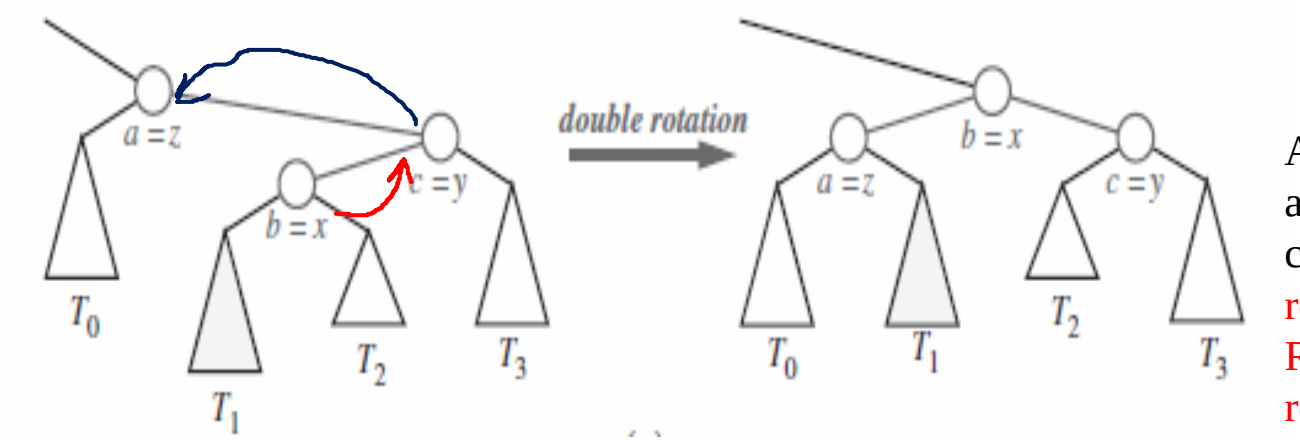
Case I: z precedes y, and y precedes x in **inorder** traversal



Also known as left-left-left case or Right-rotate(z)

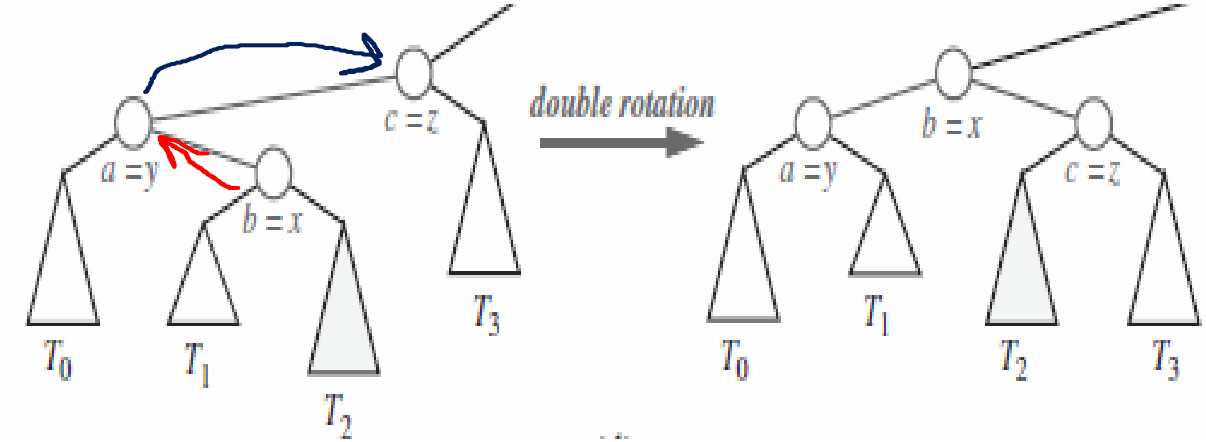
Case II: x precedes y and y precedes z in **inorder** traversal

Schematic illustration of a trinode restructure operation



Also known as left-right case or left-rotate(z), Right-rotate(z)

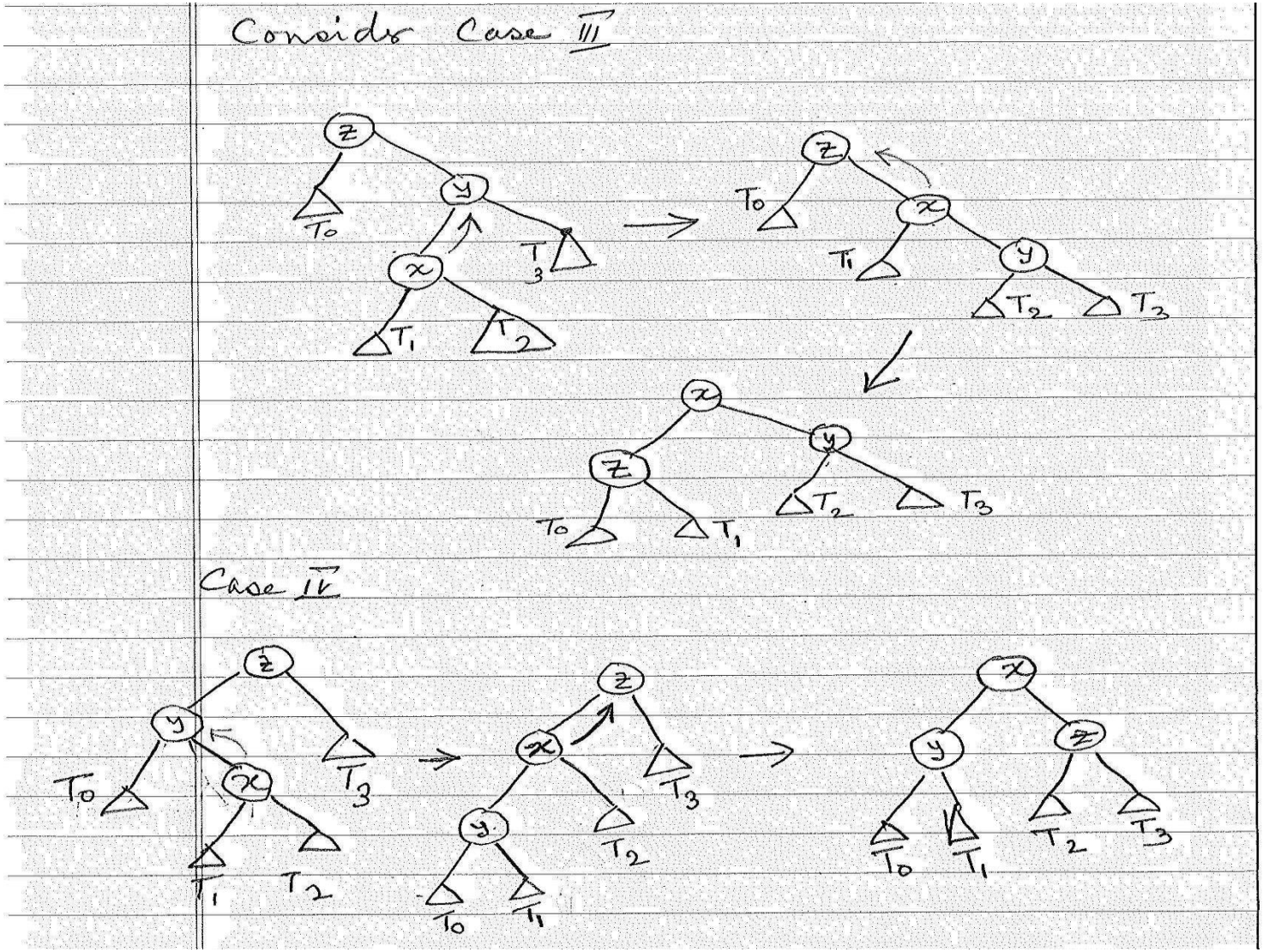
Case III: z precedes y and x precedes y in inorder traversal



Also known as right-left case or Right-rotate(z), left-rotate(z)

Case IV: y precedes x and y precedes z in inorder traversal

What is Double Rotation

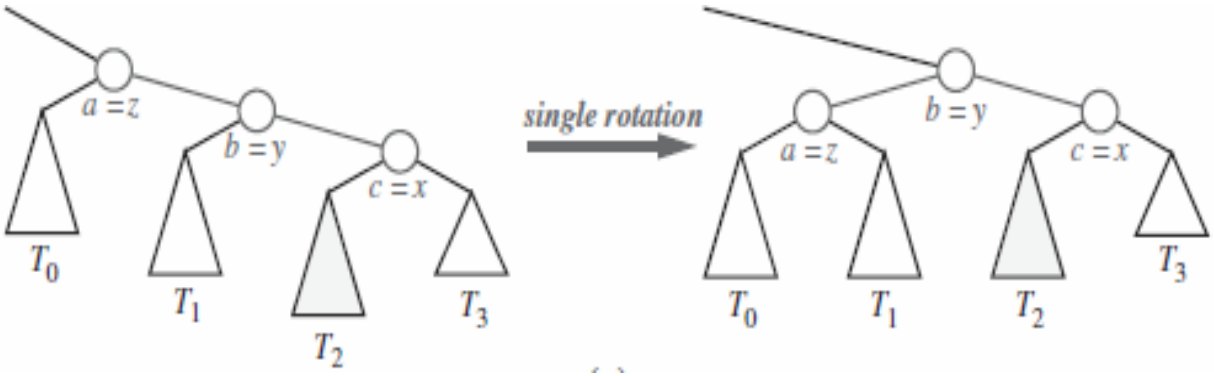


Algorithm restructure(x)

Input: A node x of a binary search tree T that has both a parent y and a grandparent z [x, y, z are node pointers]; node x is the starting point (will be clear soon)

Output: Tree T after a trinode restructuring (which corresponds to a single or double rotation) involving nodes x, y, and z

1. Let (a, b, c) be a left-to-right (inorder) listing of the nodes x, y, and z, and let (T_0, T_1, T_2, T_3) be a left-to-right (inorder) listing of the four subtrees of x, y, and z that are **not** rooted at x, y, or z. Note: a, b, c are **temporary label**s of the nodes x, y, and z, to facilitate restructuring for all possible cases.
2. Replace the subtree rooted at z with a new subtree rooted at b.
3. Let a be the left child of b and let T_0 and T_1 be the left and right subtrees of a, respectively.
4. Let c be the right child of b and let T_2 and T_3 be the left and right subtrees of c, respectively.
5. Recalculate the heights of a, b, and c, (or a "stand-in" function for height), from the corresponding values stored at their children, and return b.



In this case, z precedes y and y precedes x, i.e., a=z, b=y, and c= x in inorder traversal.

Note: To execute restructure(x), one needs to know y and z. Do iterative search & maintain a stack to go up.

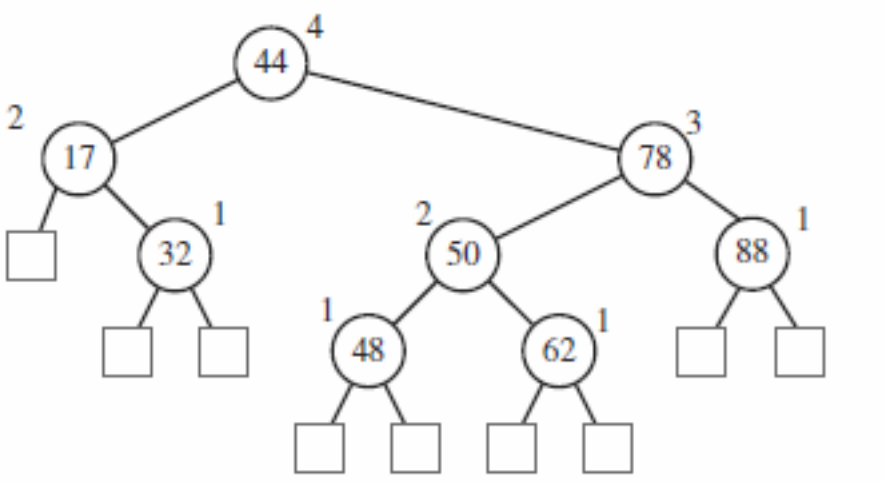
AVL Tree

- The **AVL** tree, named after its inventors, Adel'son-Vel'skii and Landis, is the oldest known and most usable balanced search tree. We define the rank, r(v), of a node, v, in a binary tree, T, simply to be the height of v in T. The rank-balancing rule for AVL trees is then defined as follows:

  Height-balance Property: For every internal node, *v*, in *T*, the heights of the children

  of *v* may differ by *at most* 1. That is, if a node, *v*, in *T* has children, *x* and *y*, then $|r(x) - r(y)| \leq 1$.

- An immediate consequence of the height-balance property is that any subtree of an AVL tree is itself an AVL tree.



Insert 5; is the new tree AVL?
Insert 54; Insert 5; is the new tree AVL?

Question: If the answer is negative, what can we do to make it AVL (of course with minimal changes, keeping insertion still $O(\log_2 n)$?

The height of an AVL tree, T, of n items is O(log n)

- Consider an AVL tree of height h; let the minimal number of internal nodes be n_h. Obviously, $n_1 = 1$ and $n_2 = 2$.
- For an AVL tree, T, with the minimum number n_h of nodes for height, h, note that the root will have as its children's subtrees an AVL tree with the minimum number of nodes for height h − 1 and an AVL tree with the minimum number of nodes for height h − 2. Thus,

$$n_h = 1 + n_{h-1} + n_{h-2}$$

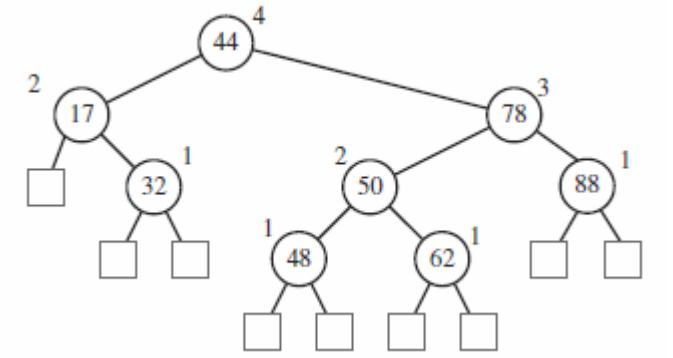
- Note that the function n_h is strictly increasing with h, i.e., $n_{h-1} > n_{h-2}$ for $h \geq 3$ and also $n_h > 2n_{h-2}$, i.e., n_h at least doubles each time h increases by 2. More formally,

$$n_h > 2^{h/2-1} \text{ or } \log n_h > \frac{h}{2} - 1 \text{ or } h < 2\log n_h + 2$$

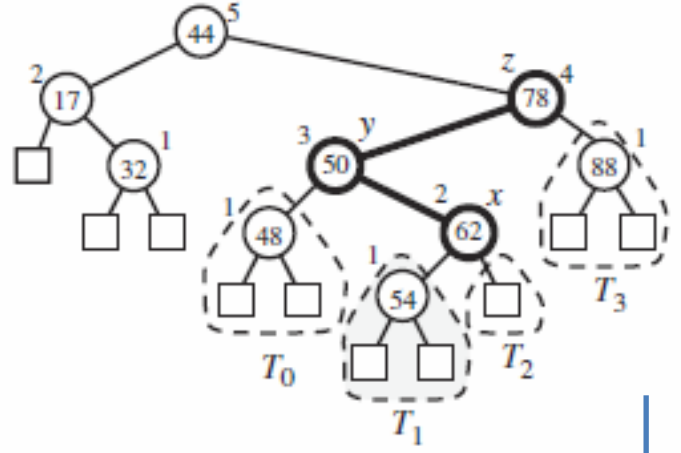
- Thus, an AVL tree storing *n* keys has height at most 2 log *n* + 2, i.e., height of an AVL tree, T, of n items is O(log n).
- Actually, this bound is somewhat overestimate. One can show that the height of an AVL tree storing *n* items is at most 1.441 log (*n* + 1) – asymptotically the same but is a tighter estimate.
- One can immediately observe that searching in an AVL tree is O(log n). The important issue remaining is to show how to maintain the height-balance property of an AVL tree after an insertion or deletion.

Insertion

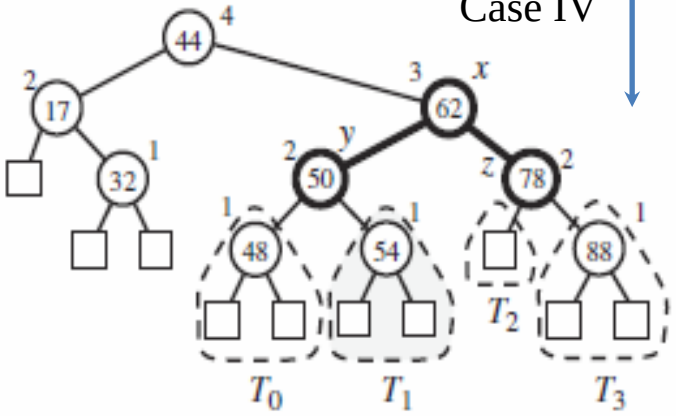
The initial phase of insertion in an AVL tree is exactly the same as in a simple BST [Recall that the operation always inserts the new item at a node w in T that was previously an external node, and it makes w become an internal node]. In an AVL tree, this action may violate the height-balance property, since some nodes increase their heights by one. In particular, node w (the node inserted), and possibly some of its ancestors, increase their heights by one. Recalculate height of the nodes along the path from w to the root and find the first nonconforming node.



Insert 54 **(w)**



Case IV



Note: (a) To check if T became unbalanced, we retrace the path up (update height of the nodes along the way) from the inserted node 54 (w) to the root to find the first unbalanced node 78 (z) ; (b) find the child (y) of z with higher height. (and note that y must be an ancestor of w; **why?**) (c) let x be the child of y with higher height (and if there is a tie, choose x to be an ancestor of w). Note that node x could be equal to w and x is a grandchild of z. Since z became unbalanced because of an insertion in the subtree rooted at its child y, the height of y is 2 greater than its sibling. We rebalance the subtree rooted at z by calling restructure(x).

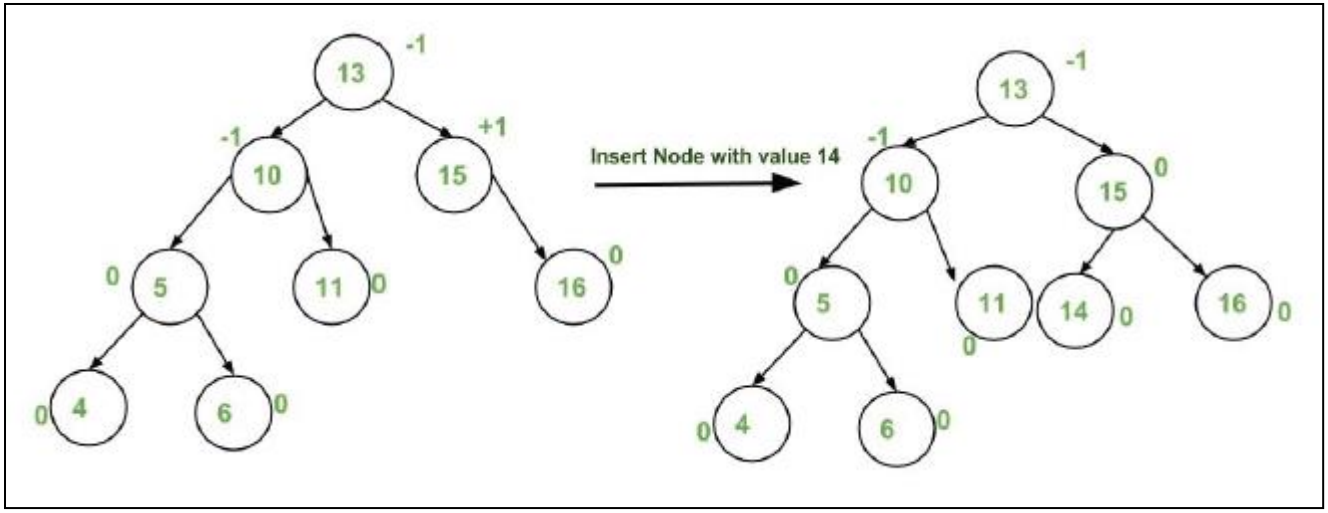
Insertion

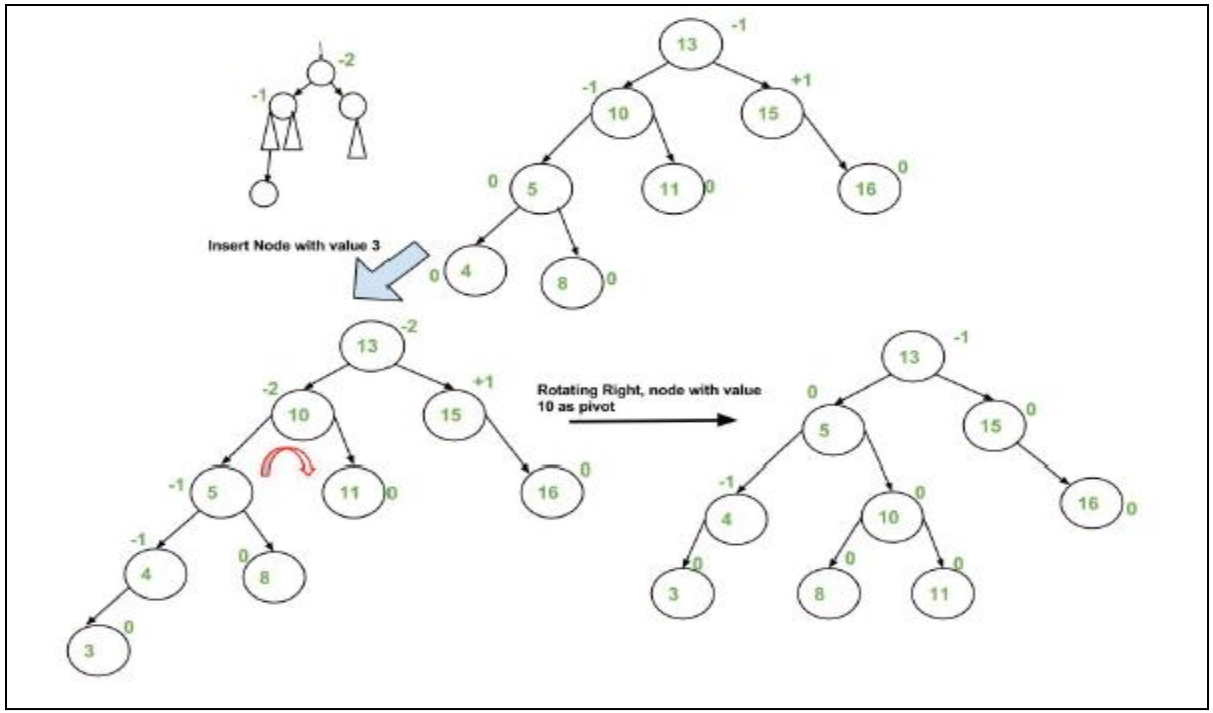
- A single insertion in an AVL tree does not always need restructuring; one needs to re-compute the heights and satisfaction of the height-balance property starting at the newly inserted node and moving up the path to the root to find the first unbalanced node (if any); other nodes in the tree will remain unaffected. We restore the height-balance property locally at the nodes x, y, and z.
- In addition, since after performing the new item insertion the subtree rooted at b replaces the one formerly rooted at z, which was taller by one unit, all the ancestors of z that were formerly unbalanced become balanced.
- Thus, this one restructuring also restores the height balance property globally. That is, one rotation (single or double) is sufficient to restore the height-balance in an AVL tree after an insertion. Of course, we may have to update the height values (ranks) of O(log n) nodes after an insertion, but the number of structural changes after an insertion in an AVL tree is O(1).

Algorithm insertAVL(k, T):
Input: A key k and an AVL tree, T pointer
Output: An updated T to now contain the item k
v ← IterativeTreeSearch(k, T)
if v is not an external node then
  return "An item with key k is already in T";
Expand v into an internal node with two external-node children;
v.key ← k, v.h ← 1, rebalanceAVL(v, T)

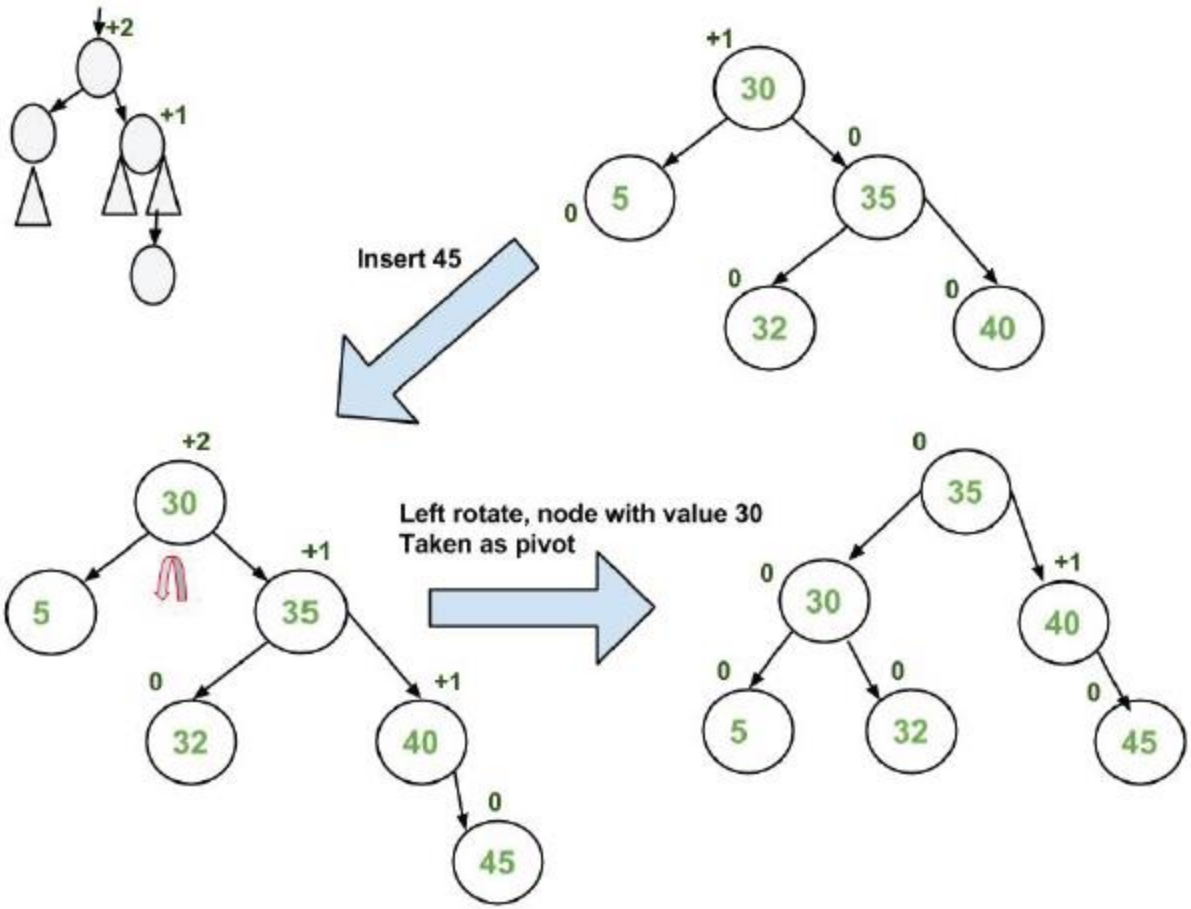
Note: T denotes a tree pointer, k is an integer key to be inserted, v is a tree node pointer. IterativeTreeSearch is a search in a simple BST that returns a node pointer; rebalanceAVL(v, T) is a non-typed function that rebalances the tree T using node x as the pivot; will define soon.

More Insertion Examples

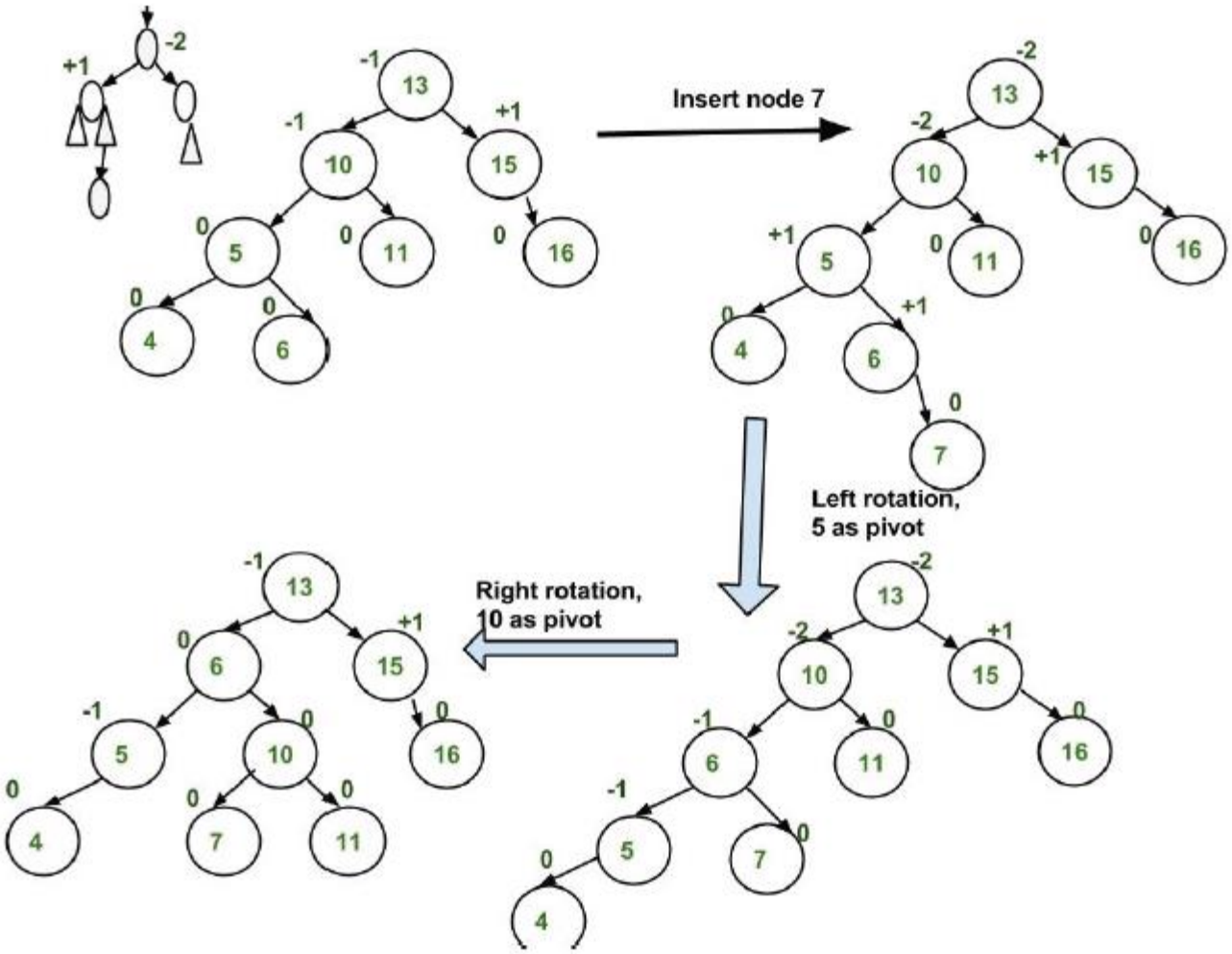




More Insertion Examples

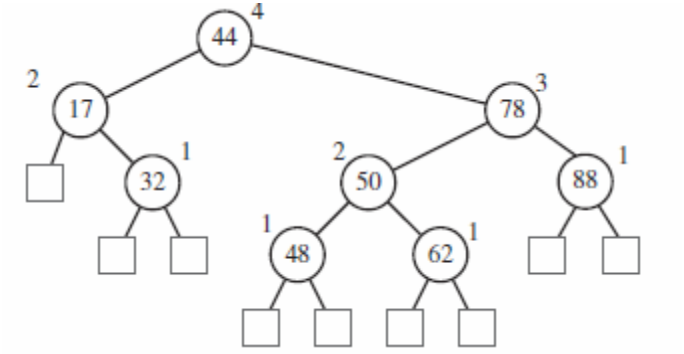


More Insertion Examples

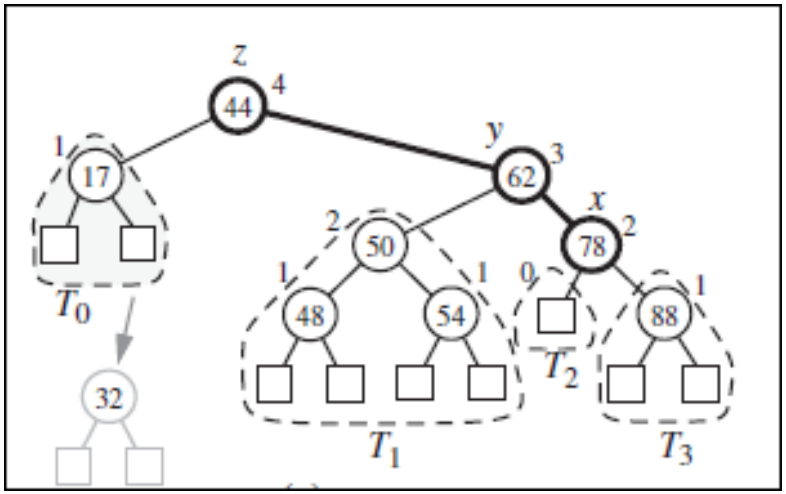


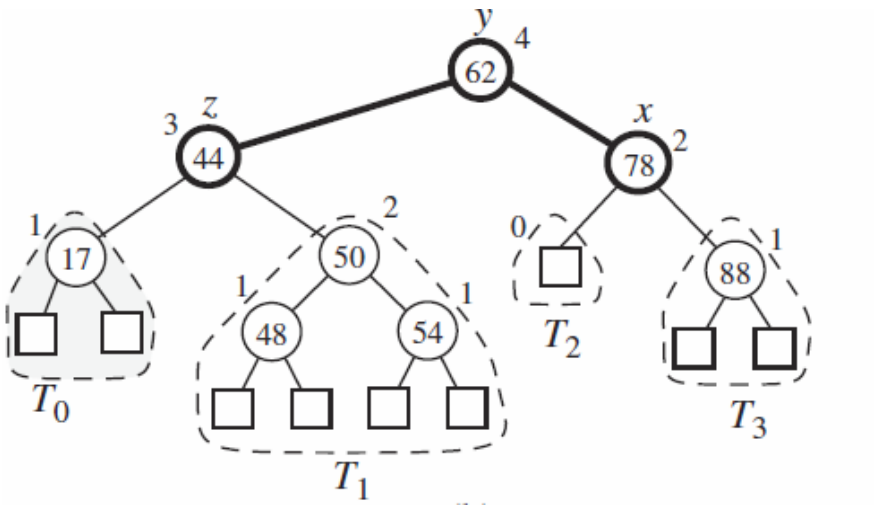
Deletion in an AVL tree

- After removing an internal node and elevating one of its children into its place, there may be an unbalanced node in T on the path from the parent w of the previously removed node to the root of T. Remember how deletions are done in a simple BST.



We want to delete the node 32 (node w); re compute height of the node 17; it is balanced; go up to the root to find the first unbalanced node; in this case, that is the root node 44. We will use trinode restructuring again. We need to identify nodes x, y and z as we did earlier. Let's recap deletion in a simple BST first.





Recap: Delete in a simple BST

- Removing a node from a BST is a bit trickier, since we do want to make sure that the BST remains a BST after the deletion. There are 3 possibilities:
  - **The node to be deleted is a leaf node**:
    Easiest: Just delete it; that's all [Be careful when deleting the root which is a leaf – one node tree]



  - **Node to be deleted has only one child (left or right):**
    Connect the only subtree of the node to the parent of the node to be deleted and delete.



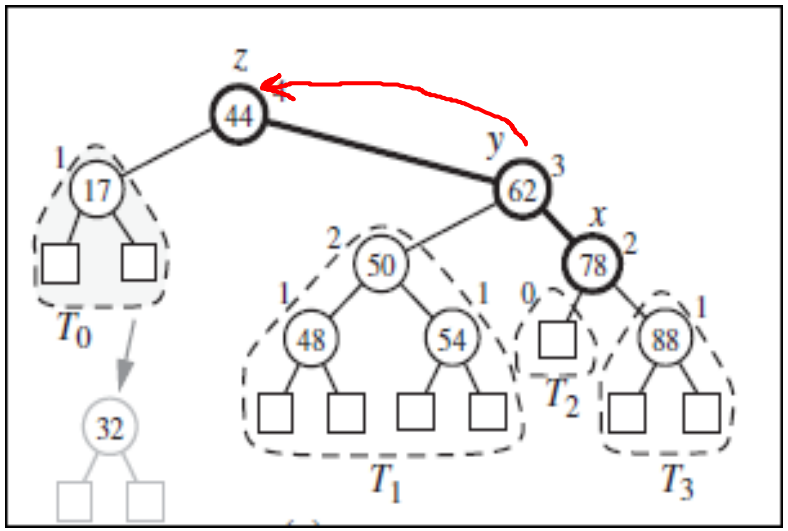
  - **Node to be deleted has two children**:
    Tricky: Find inorder successor of the node. Copy contents of the inorder successor to the node (to be deleted) and delete the inorder successor.

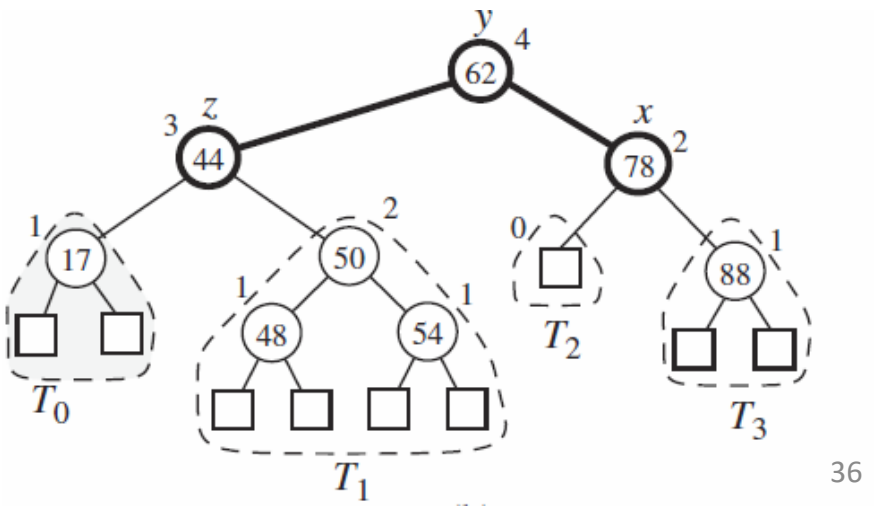


- The highlighted node positions show where deletion is made. Note that in the third case inorder successor of the node, say z, (to be deleted) is the minimum of the node z->right. This can also be done by using inorder predecessor ?
- Question: Do the three cases exhaust all possibilities?

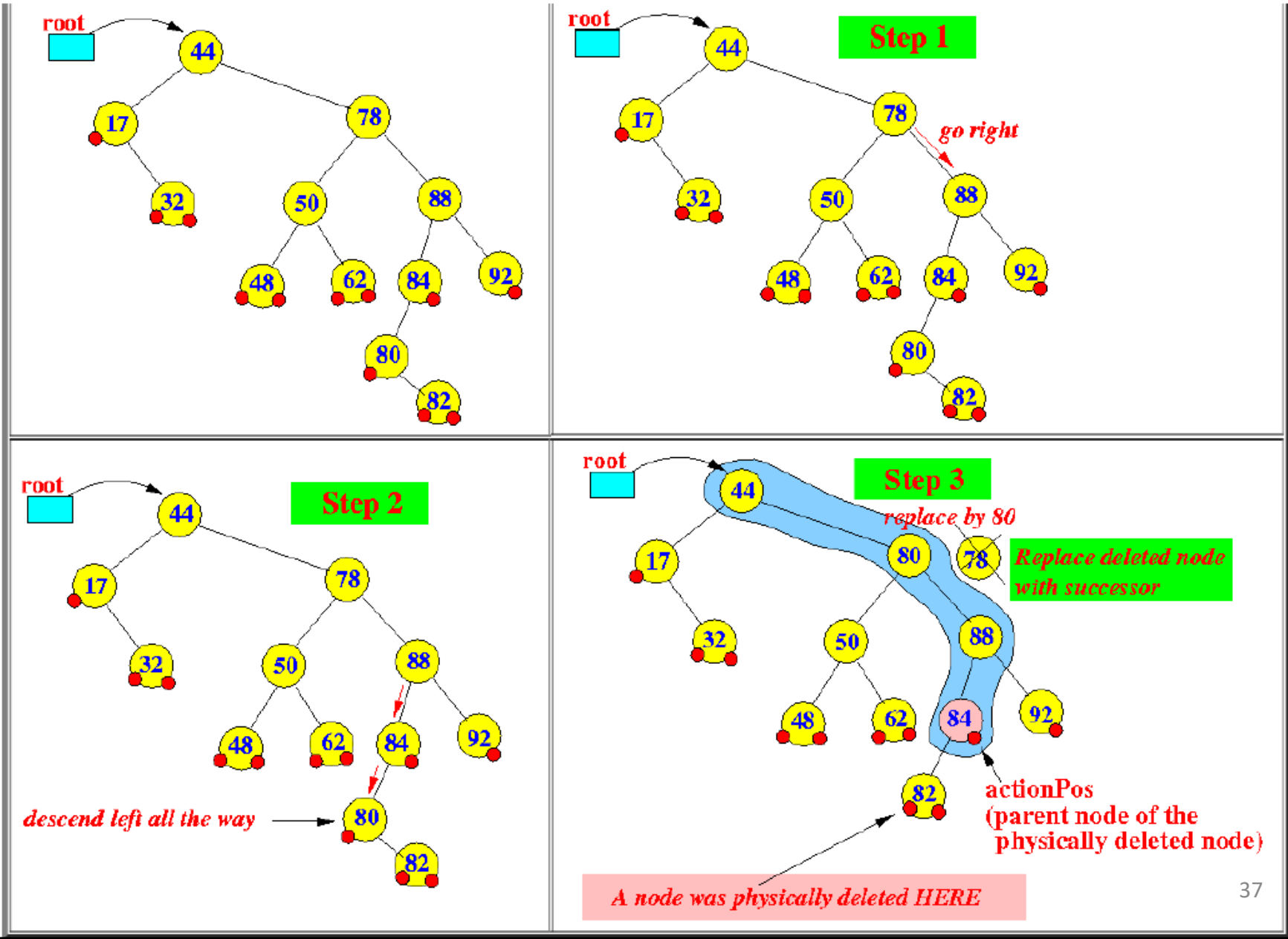
Deletion in AVL tree

- Let z be the first unbalanced node encountered going up from w toward the root of T. Also, let y be the child of z with larger height (note that node y is the child of z that is not an ancestor of w). Finally, let x be the child of y defined as follows:
  - **if** one of the children of y is taller than the other, let x be the taller child of y;
  - **else** (both children of y have the same height), let x be the child of y on the same side as y (that is, if y is a left child, let x be the left child of y, else let x be the right child of y).
- Look at the example. k=32; w (the node to be deleted) has 2 external node child; go up to find the first unbalanced node z (64, height is 4); the taller child of z is 62 (y) [note that node y has to be the child of z that is not an ancestor of w]; both children of y has same height (2), so we choose 78 (the right child of y since y is the right child of z) to be node x. Remember that a, b, c are temporary relabeling of nodes x, y, z in inorder fashion; a=z, b=y, c=x; Case I]. Perform restructure(x) operation which restores the height-balance property locally, at the subtree that was formerly rooted at z and is now rooted at the node we temporarily called b.





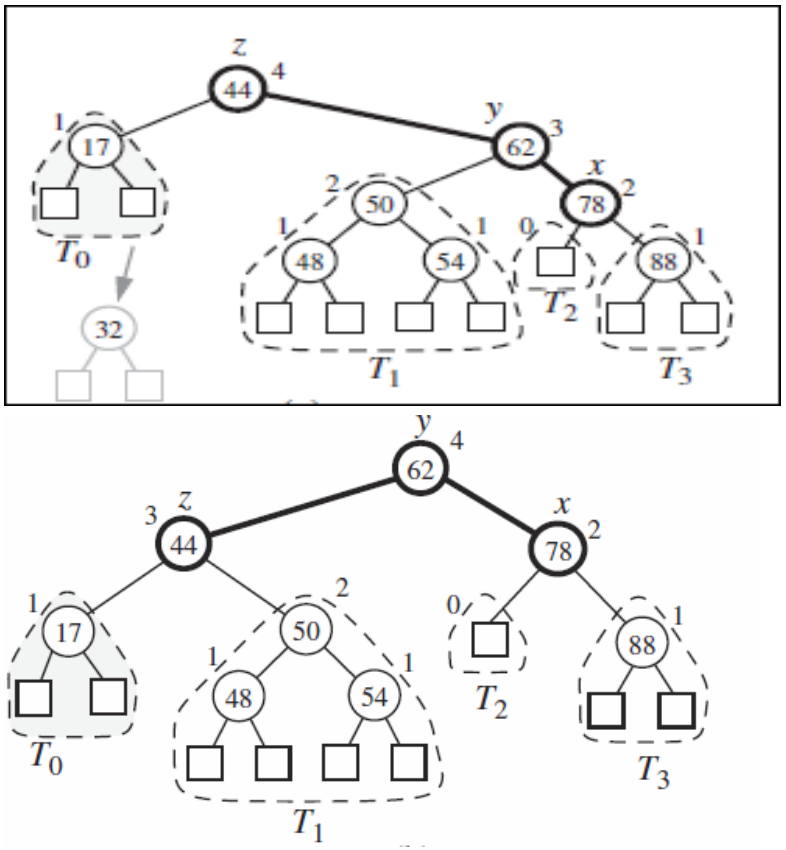
More Examples of Deletion in AVL Trees



More Deletion Examples

Pseudocode of Deletion

Algorithm removeAVL(k, T):
Input: A key, k, and an AVL tree, T
Output: An update of T without the key k
v ← IterativeTreeSearch(k, T)
if v is an external node **then**
    return "no item with key k in T"
if v has no external-node child **then**
    Let u be the node in T with key nearest to k
 // u is inorder successor/predecessor of v
    Move u's key-value pair to v
    v ← u
Let w be v's smallest-height child
Remove w and v from T, replacing v with w's sibling, z
rebalanceAVL(z, T)



OK, consider the example. k=32; w (the node to be deleted) has 2 external node child; go up to find the first unbalanced node z (44, height is 4); the taller child of z is 62 (y) [note that node y has to be the child of z that is not an ancestor of w; **why?**]; both children of y has same height (2), so we choose 78 (the right child of y since y is the right child of z) to be node x [if one of the children of y is taller than the other, let x be the taller child of y]. Observe the relabeling: a=z, b=y, c=x. Apply restructure [Case I].

Pseudocode of Rebalance

Algorithm rebalanceAVL(v, T):
Input: A node, v, where an imbalance may have occurred in an AVL tree, T
Output: An update of T to now be balanced

```
v.height ← 1+max{v.leftChild().height, v.rightChild().height}
while v is not the root of T do
    v ← v.parent()
    if |v.leftChild().height − v.rightChild().height| > 1 then
      Let y be the tallest child of v and let x be the tallest child of y
      v ← restructure(x) // trinode restructure operation
    v.height ← 1+max{v.leftChild().height, v.rightChild().height}
```

Note:

1. This version of the rebalance method does not include the heuristic of stopping as soon as balance is restored, and instead always performs any needed rebalancing operations all the way to the root.
2. The restructure operation is O(1); thus, rebalancing after a single insertion is O(1)
3. An AVL tree for n key-element items uses O(n) space and executes the operations find, insert and remove to each take O(log n) time.

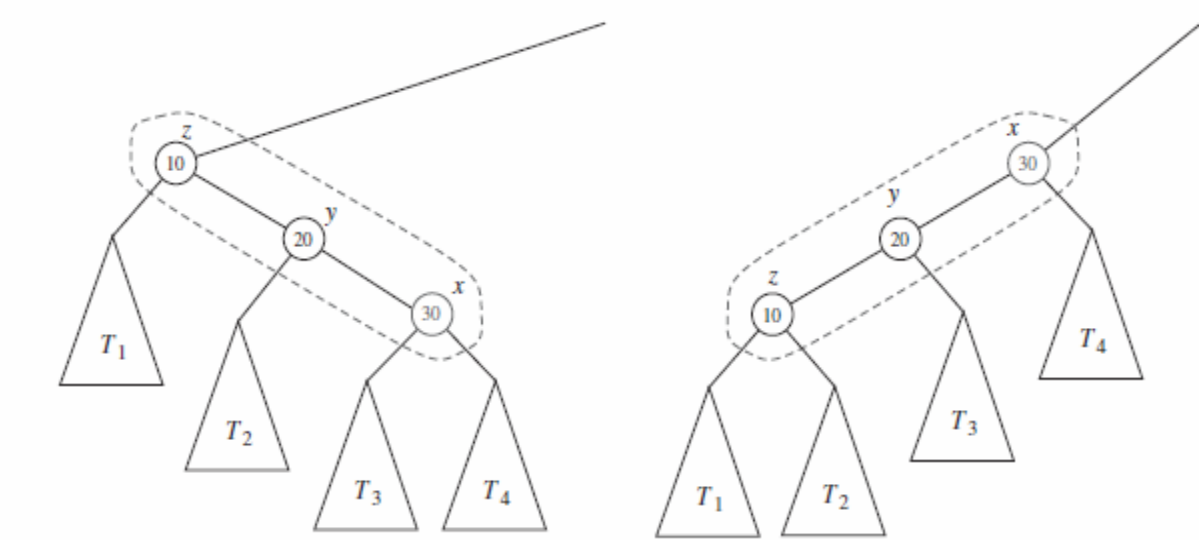
Operation	Time	Structural Changes
find	$O(\log n)$	none
insert	$O(\log n)$	$O(1)$
remove	$O(\log n)$	$O(\log n)$

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.
- The structure of **Splay Tree** is conceptually different from that of AVL or balanced binary search trees. It applies a certain **move-to-root** operation, called **splaying** after every access, in order to keep the search tree balanced in an amortized sense. The splaying operation is performed at the bottommost node x reached during an insertion, deletion, or even a search.
  - The structure of a splay tree is simply a binary search tree T; there are no additional height, balance, or color labels that we associate with the nodes of this tree.
  - Each operation, search, insert, delete, has an amortized cost of $O(\log_2 n)$.
  - In the worst case, an individual operation may be as bad as O(n) and the height of the tree can also be O(n) in the worst case.

Splaying

- Given an internal node x of a binary search tree T, we splay x by moving x to the root of T through a sequence of restructurings. The specific operation we perform to move x up depends upon the relative positions of x, its parent y, and (if it exists) x's grandparent z. There are three cases that we consider.
  - **zig-zig**: The node x and its parent y are both left or right children. We replace z by x, making y a child of x and z a child of y, while maintaining the inorder relationships of the nodes in T.



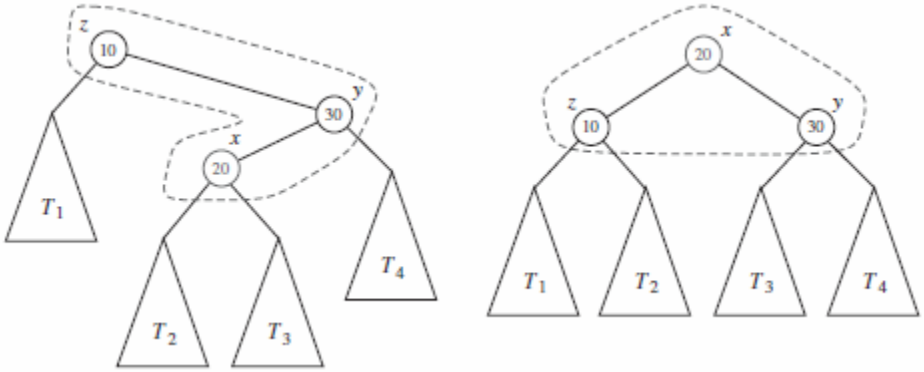
before after

Note: There is another *symmetric configuration* where x and y are left children. Remember what we did for AVL trees.

Goodrich+Tamasia Book

Splaying

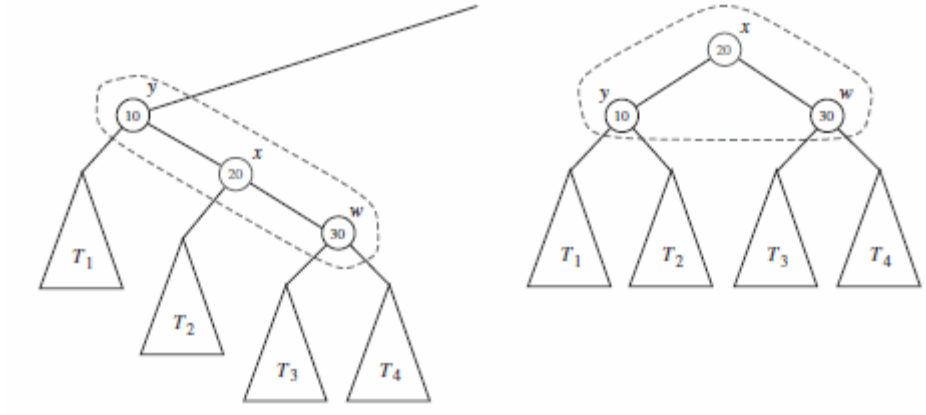
- **zig-zag**: One of x and y is a left child and the other is a right child. In this case, we replace z by x and make x have as its children the nodes y and z, while maintaining the inorder relationships of the nodes in T.



before after

Note: There is another *symmetric configuration* where x is a right child and y is a left child. Remember what we did for AVL trees.

- **zig:** x does <u>not</u> have a grandparent (or we are not considering x's grandparent for some reason). In this case, we rotate x over y, making x's children be the node y and one of x's former children w, so as to maintain the relative inorder relationships of the nodes in T.



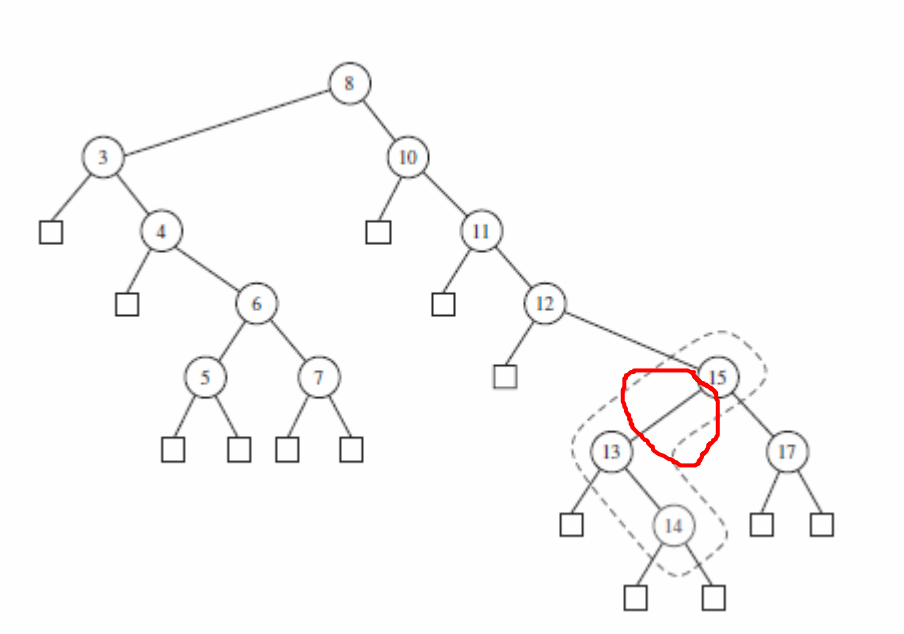
before after

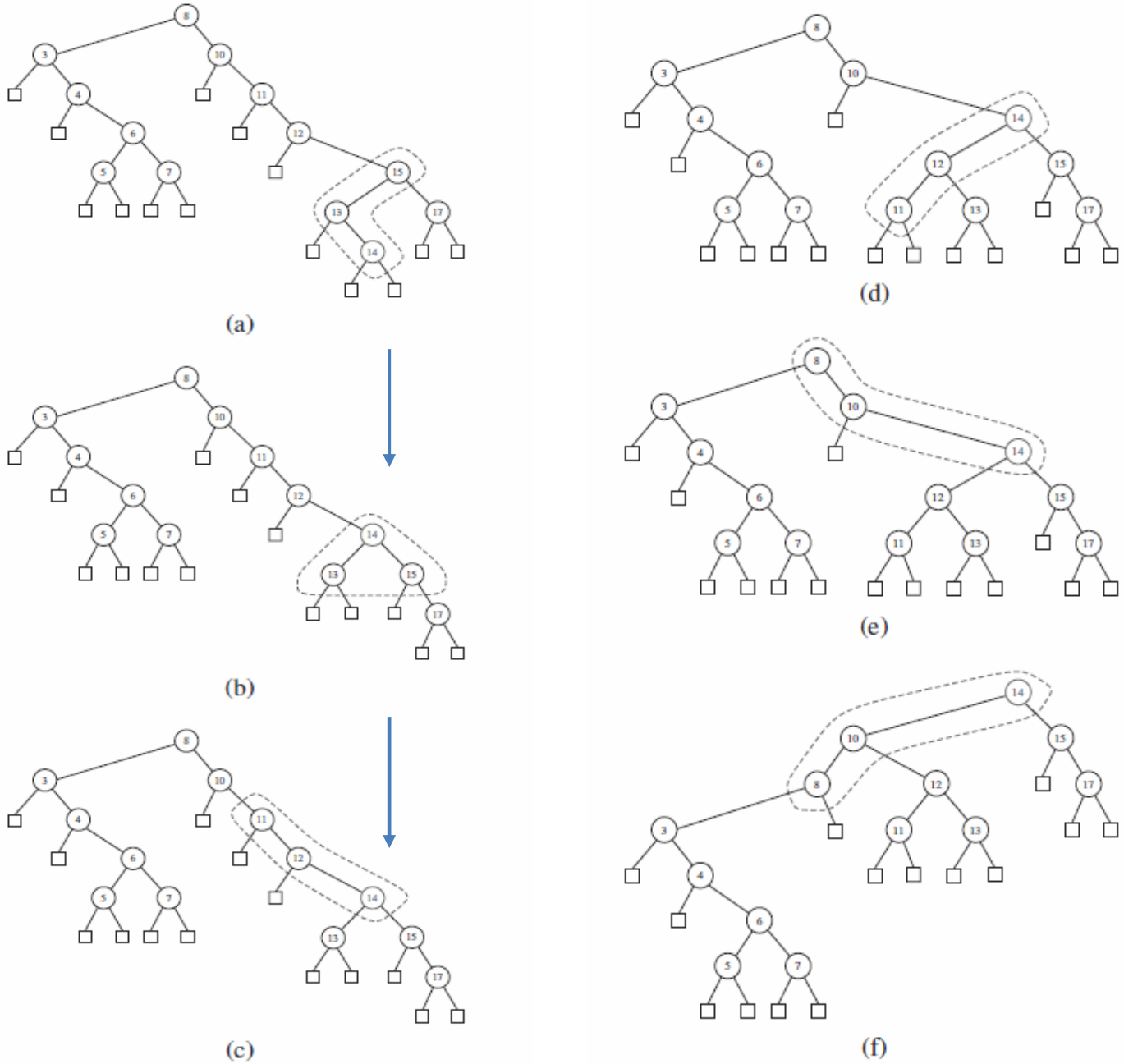
Note: There is another *symmetric configuration* where x and w are left children. Remember what we did for AVL trees.

Note: Both zig-zig and zig-zag operations move a node x up 2 levels. The zig operation moves a node x up 1 level.

Guidelines for Splaying

- We perform a zig-zig or a zig-zag when x has a grandparent, and we perform a zig when x has a parent but not a grandparent.
- A splaying step consists of repeating these restructurings at x until x becomes the root of T. [**Caution**: this is not the same as a sequence of simple rotations that brings x to the root.]
- We will splay the node storing 14; what do we do first? zig-zag; why?



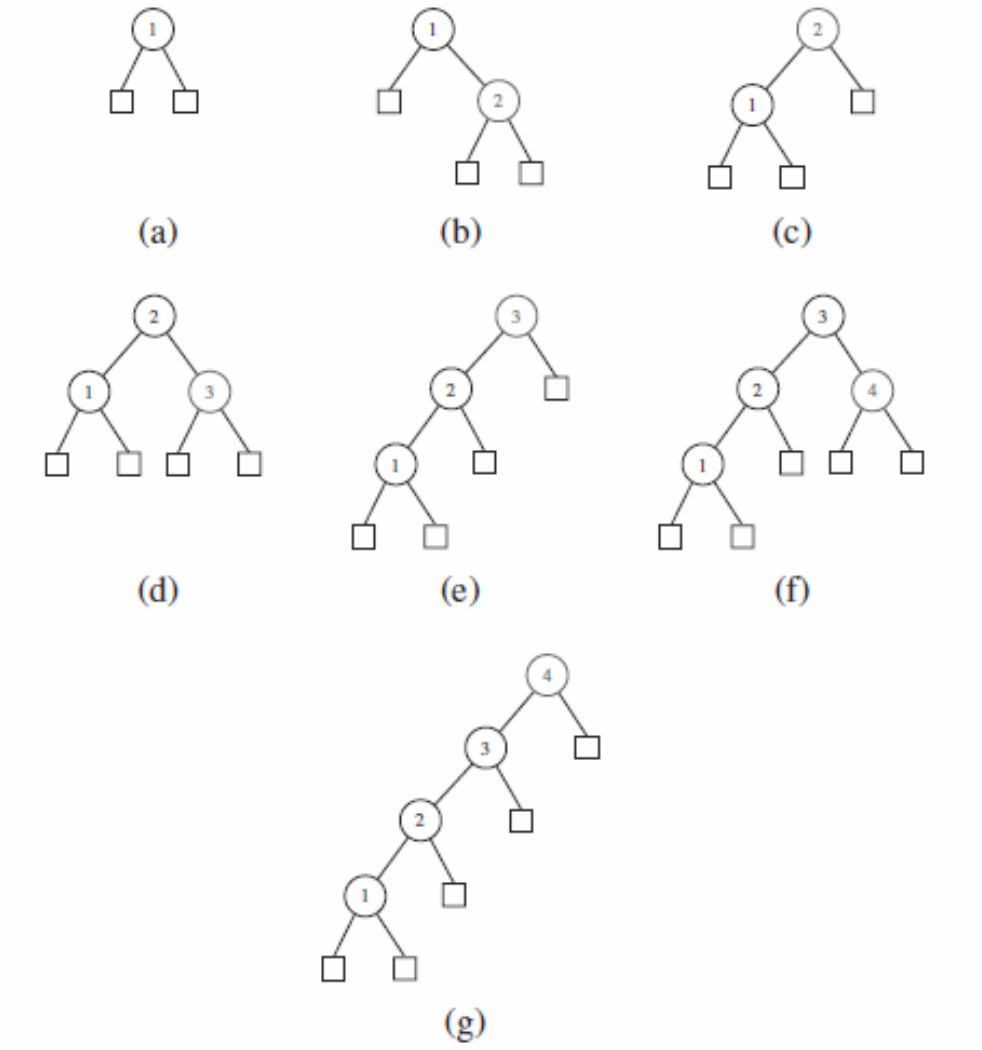


(a) splaying the node storing 14 starts with a zig-zag; (b) after the zig-zag; (c) the next step is a zig-zig; (d) after the zig-zig; (e) the next step is again a zig-zig; (f) after the zig-zig.

Observations

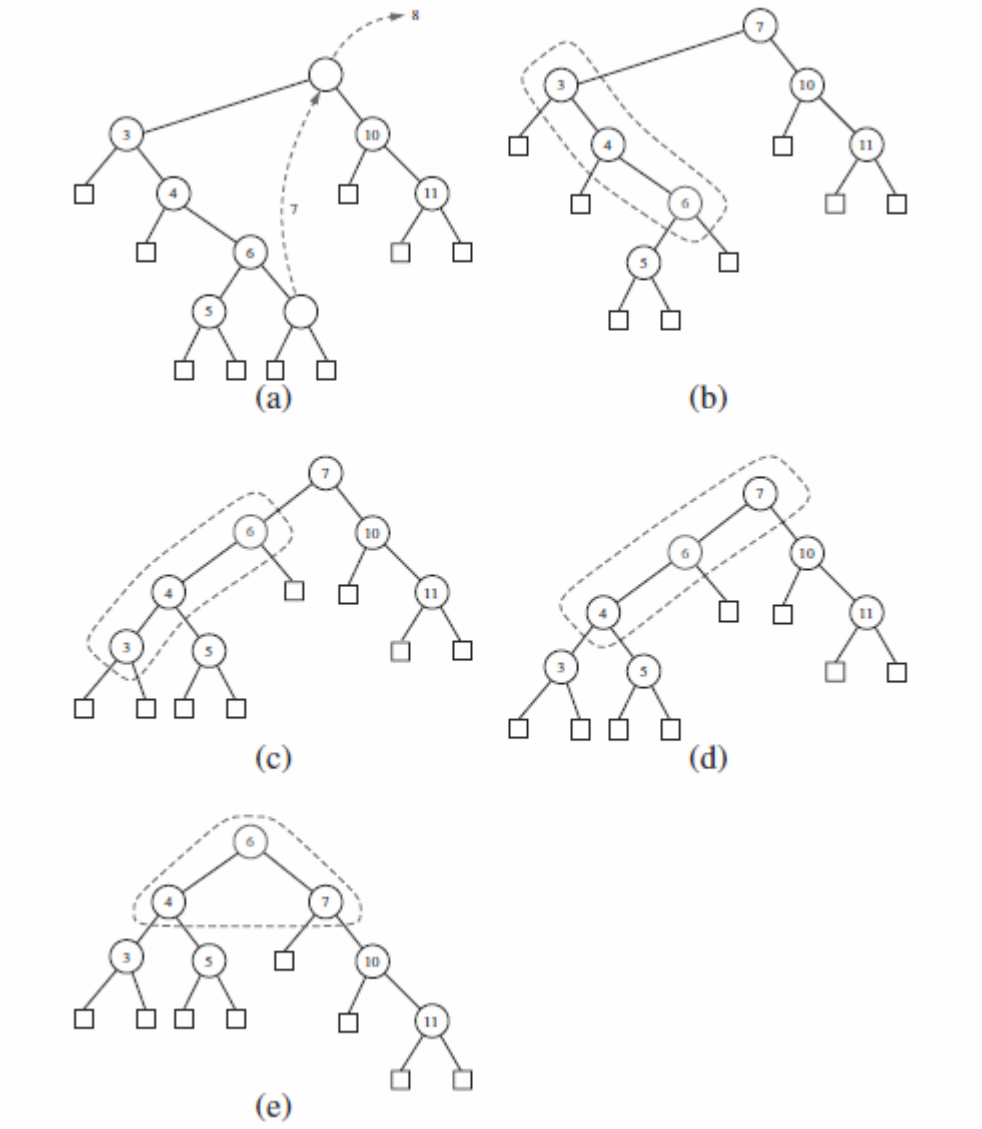
- After a zig-zig or zig-zag, the depth of x decreases by two, and after a zig the depth of x decreases by one. Thus, if x has depth d, splaying x consists of a sequence of $\lfloor d/2 \rfloor$ zig-zigs and/or zig-zags, plus one final zig if d is odd.
  - Since a single zig-zig, zig-zag, or zig affects a constant number of nodes, it can be done in O(1) time. Thus, splaying a node x in a binary search tree T takes time O(d), where d is the depth of x in T.
  - In other words, the time for performing a splaying step for a node x is asymptotically the same as the time needed just to reach that node in a top-down search from the root of T.
- **When to splay:**
  - When searching for key k, if k is found at a node x, we splay x, else we splay the parent of the external node at which the search terminates unsuccessfully.
  - When inserting key k, we splay the newly created internal node where k gets inserted.
  - When deleting a key k, we splay the parent of the node w that gets removed, that is, w is either the node storing k or one of its descendants. (Recall the deletion algorithm for binary search trees)
- In the **worst case**, the overall running time of a search, insertion, or deletion in a splay tree of height h is O(h), since the node we splay might be the deepest node in the tree. Moreover, it is possible for h to be $\Omega(n)$. Thus, from a worst-case point of view, a splay tree is not an attractive data structure.

Insertion Example



(a) 1 is inserted; (b) 2 is inserted; (c) after splaying; (d) after inserting 3; (e) after splaying; (f) after inserting 4; (g) after splaying.

Deletion Example



(a) the deletion of 8 from node r is performed by moving to r the key of the right-most internal node v, in the left subtree of r, deleting v, and splaying the parent u of v; (b) splaying u starts with a zig-zig; (c) after the zig-zig; (d) the next step is a zig; (e) after the zig.

Amortized Cost

- In an arbitrary sequence of intermixed searches, insertions, and deletions, each operation in a splay tree takes on average logarithmic time. We note that the time for performing a search, insertion, or deletion is proportional to the time for the associated splaying; hence, in our analysis, we consider only the splaying time.
- Let T be a splay tree with n keys, and let v be a node of T. We define the **size n(v) of v** as the number of nodes in the subtree rooted at v. Note that the size of an internal node is one more than the sum of the sizes of its two children. We define **the rank r(v) as r(v) = $\log_2$n(v).** [if e is a leaf node, n(e) = 1 and r(e) =0]
- Clearly, the root of T has the maximum size 2n + 1 and the maximum rank, log (2n + 1), while each external node has size 1 and rank 0.
- We use a method similar to the potential function. Say, we pay for the work in splaying a node x in T, and we assume that **one cyber-dollar pays for a zig**, while **two cyber-dollars pay for a zig-zig or a zig-zag.** Hence, the cost of splaying a node at depth *d* is *d* cyber-dollars.
- We keep a virtual (imaginary, not actually stored in the data structure) account, storing cyber-dollars, at each internal node of *T*. We make a payment for each operation.
- We have 3 cases to consider:
  - If the payment is equal to the splaying work, then we use it all to pay for splaying.
  - If the payment is greater than the splaying work, we deposit the excess in the accounts of several nodes.
  - If the payment is less than the splaying work, we make withdrawals from the accounts of several nodes to cover the deficiency.

Amortized Cost (contd.)

- We pay O(log n) cyber-dollars per operation to keep the system working, that is, to ensure that each node keeps a nonnegative account balance. We use a scheme in which transfers are made between the accounts of the nodes to ensure that there will always be enough cyber-dollars to withdraw for paying for splaying work when needed. We also maintain the following invariant:

  Before and after a splaying, each node v of T has r(v) cyber-dollars

- Note that the we do not require us to endow an empty tree with any cyber-dollars (really?)
- **Notations:**
  - **(1)** Let **r(T) be the sum of the ranks of nodes of T**. To preserve the invariant after a splaying, we must make a payment equal to the splaying work plus the total change in r(T). We refer to a single zig, zig-zig, or zig-zag operation in a splaying as a **splaying sub step**.
  - **(2)** we denote the rank of a node v of T before and after a splaying sub step with $r(v)$ and $q(v)$ [we will need to compute the change]
  - **(3)** Let **δ** be the variation of $r(T)$ caused by a single splaying sub step (a zig, zig-zig, or zig-zag) for a node x in a splay tree T. Also, let **Δ** be the total variation of r(T) caused by splaying a node x at depth d.
- We need to use the formula **If $a > 0$, $b > 0$, and $c > a + b$, then $\log a + \log b \leq 2 \log c - 2$.** (Can you show why it is true? Try!)
- What is the upper bound of **δ?**

Amortized Cost (contd.)

- **Claim:** Let **δ** be the variation of $r(T)$ caused by a single splaying sub step (a zig, zig-zig, or zig-zag) for a node x in a splay tree T, at depth d. Then
  - $\delta \leq 3(q(x) - r(x)) - 2$ if the substep is a zig-zig or zig-zag.
  - $\delta \leq 3(r(x) - r(x))$ if the substep is a zig.
- We consider each type of splaying substep. Recall that the size of each node is one more than the size of its two children, note that only the ranks of x, y, and z change in a zig-zig operation, where y is the parent of x and z is the parent of y.

zig-zig: Note $q(x) = r(z)$, $q(y) \leq q(x)$, $r(y) \geq r(x)$ [Check why!]

Then, $\delta = q(x) + q(y) + q(z) - r(x) - r(y) - r(z)$

$= q(x) + q(z) - r(x) - r(y)$

$\leq q(x) + q(z) - 2r(x)$

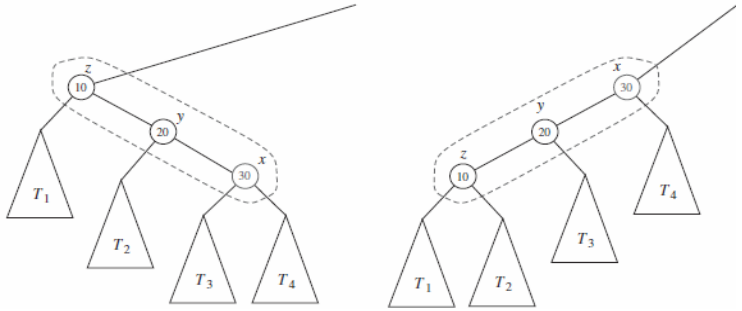
Also, $n(x) + n'(z) \leq n'(x) \Rightarrow r(x) + q(z) \leq 2q(x) - 2$

$\Rightarrow q(z) \leq 2q(x) - r(x) - 2.$

Combining this with the previous inequality we get,

$\delta \leq q(x) + (2q(x) - r(x) - 2) - 2r(x)$

$\leq 3(q(x) - r(x)) - 2$



before after

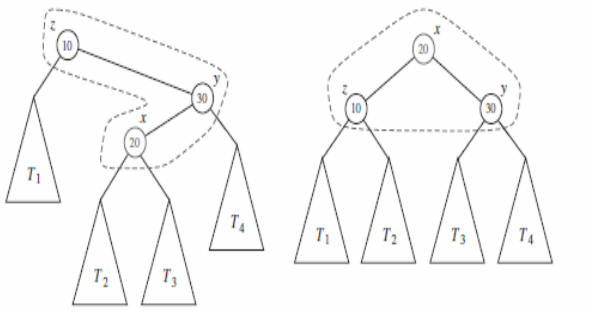
Amortized Cost (contd.)

zig-zag: By the definition of size and rank, only the ranks of x, y, and z change, where y denotes the parent of x and z denotes the parent of y. Also, q(x) = r(z) and r(x) ≤ r(y). Thus,

$$\begin{aligned}\delta &= q(x) + q(y) + q(z) - r(x) - r(y) - r(z)\\ &\le q(x) + q(z) - r(x) - r(y)\\ &\le q(x) + q(z) - 2r(x)\end{aligned}$$

Again, $n'(y) + n'(z) \le n'(x)$ or $q(y) + q(z) \le 2q(x) - 2$.
So, we get for δ

$\delta \le 2q(x) - 2 - 2 - 2r(x) \le 3(q(x) - r(x)) - 2$.



zig: In this case, only the ranks of x and y change, where y denotes the parent of x. Also, $q(y) \le r(y)$ and $q(x) \ge r(x)$. Thus,

$$\begin{aligned}\delta &= q(y) + q(x) - r(y) - r(x)\\ &\le q(x) - r(x)\\ &\le 3(q(x) - r(x)).\end{aligned}$$

