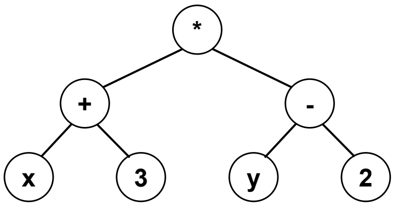
Context for Trees

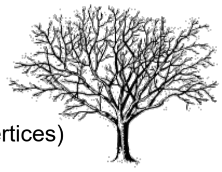
Tree: a mathematical abstraction that

- Captures common properties of data
- Critical to the design and analysis of algorithms



(x + 3) * (y - 2)

Trees



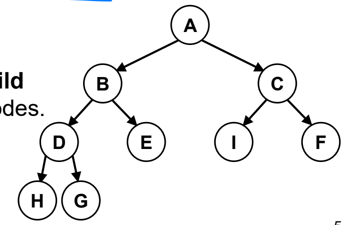
A **graph** consists of **nodes** (sometimes called vertices) connected together by **edges**.

Each node can contain some data.

A **tree** is:
(1) a connected graph (nodes + edges) w/o cycles.
(2) a graph where any 2 nodes are connected by a unique shortest path.
(the two definitions are equivalent)

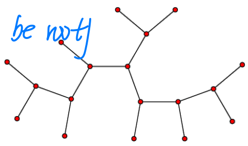
In a directed tree, we can identify **child** and **parent** relationships between nodes.

In a **binary tree**, a node has at most two children.



Types of Trees

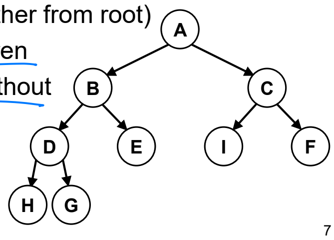
1. (Simple) tree (any node can be root)
   - Acyclic connected graph
   - Considered *undirected*
2. Rooted tree
   - A simple tree with a selected node (root)
   - All edges are *directed* away from root



- Any node could be selected as root

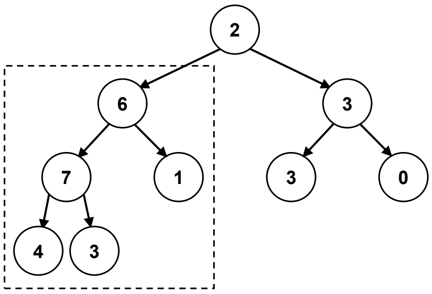
Tree Terminology

Root: the "topmost" node in the tree
Parent: Immediate predecessor of a node
Child: Immediate successor of a node
Siblings: Children of the same parent
Ancestor: parent of a parent (closer to root)
Descendent: child of a child (further from root)
Internal node: a node with children
External (Leaf) node: a node without children



Trees are Recursive Structures

Any subtree is just as much a "tree" as the original!

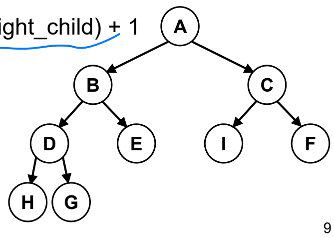


Tree Properties

Height:
height(empty) = 0
height(node) = max(height(left_child), height(right_child)) + 1

Size:
size(empty) = 0
size(node) = size(left_child) + size(right_child) + 1

Depth:
depth(empty) = 0
depth(node) = depth(parent) + 1



Binary Trees

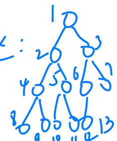
- **Ordered Tree**: linear ordering for the children of each node
- **Binary Tree**: ordered tree in which every node has at most two children
- Multiple binary tree implementations styles

Complete Binary Tree Property

Definition: ***complete binary tree***

- A binary tree with depth *d* where
  - Tree depths 1, 2, …, *d - 1* have the max number of nodes possible
  - All internal nodes are to the left of the external nodes at depth *d - 1*
  - That is, all leaves are at *d* - 1 or leftmost at depth *d*

ex :



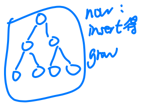
Binary Tree Implementation

Binary tree array implementation

- Root at index 1
- Left child of node *i* at 2 * *i*
- Right child of node *i* at (2 * *i*) + 1
- Some indices may be skipped
- Can be space prohibitive for sparse trees

Binary Tree Implementation

Complexity of array implementation



- Insert key (best-case) 当前大小 $2^{n+1}-1$ tail: $[2^n, 2^{n+1}-1)$ **$O(1)$**
- Insert key (worst-case) 当前大小 $2^{n+1}-1$ tail: 正好在 $2^{n+1}-1$ **$O(n)$**
- Remove key (worst-case) (还要动子节点) **$O(n)$**
- Parent $[\lfloor \frac{n}{2} \rfloor]$ **$O(1)$**
- Child $[2n+1]$, $[2n+2]$ **$O(1)$**
- Space (best-case) **$O(n)$**
- Space (worst-case) **$O(2^n)$**

Binary Tree Implementation

- Pointer-based binary tree implementation

```
template <class KEY>
struct Node {
  KEY key;
  Node *left = nullptr;
  Node *right = nullptr;
  Node(const KEY &k) : key{k} {}
}; // Node{}
```

- A node contains some information, and points to its left child node and right child node
- Efficient for moving *down* a tree from parent to child

Binary Tree Implementation

Complexity of pointer implementation

- Insert key (best-case) **$O(1)$**
- Insert key (worst-case) **$O(n)$**
- Remove key (worst-case) **$O(n)$**
- Parent **$\underline{O(n)}$**
- Child **$O(1)$**
- Space (best-case) **$O(n)$**
- Space (worst-case) **$\underline{O(n)}$**

Binary Tree++

Another way to do it (not common)

```
template <class KEY>
struct Node {
    KEY key;
    Node *parent, *left, *right;
}; // Node()
```

- If node is root, then `*parent` is `nullptr`
- If node is external, then `*left` and `*right` are `nullptr`

Translating General Trees into Binary Trees

T: General tree

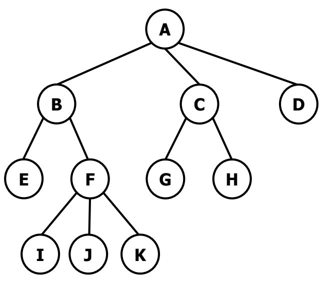
T': Binary tree

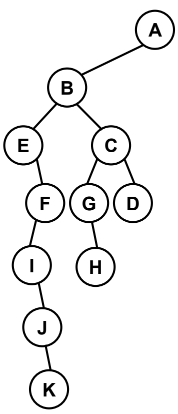
Intuition:

- Take set of siblings $\{v_1, v_2, \ldots, v_k\}$ in *T*, that are children of *v*
- v_1 becomes left child of *v* in *T'*
- $v_2 \ldots v_k$ become chain of right children of v_1, in *T'*
- Recurse from v_2

Left: new "generation"; Right: sibling

Tree Translation





Tree Traversal

Systematic method of processing every node in a tree

- Preorder
  1. Visit node
  2. Recursively visit left subtree
  3. Recursively visit right subtree
- Inorder
  1. Recursively visit left subtree
  2. Visit node
  3. Recursively visit right subtree

Tree Traversal

Systematic method of processing every node in a tree

- Postorder
  1. Recursively visit left subtree
  2. Recursively visit right subtree
  3. Visit node
- Level order
  – Visit nodes in order of increasing depth in tree

Recursive Implementations

```
template <class KEY>
struct Node {
  KEY key;
  Node *left = nullptr;
  Node *right = nullptr;
}; // Node{}
void preorder(Node *p) {
  if (!p) return;
  visit(p->key);
  preorder(p->left);
  preorder(p->right);
} // preorder()
```

```
void postorder(Node *p) {
  if (!p) return;
  postorder(p->left);
  postorder(p->right);
  visit(p->key);
} // postorder()
void inorder(Node *p) {
  if (!p) return;
  inorder(p->left);
  visit(p->key);
  inorder(p->right);
} // inorder()
```


Summary of Tree Traversals

Tree traversal: Systematically process all nodes in a tree

- Preorder
- Inorder
- Postorder

All are depth-first traversals

- Level order (breadth-first traversal)

Search

- Retrieval of a particular piece of information from large volumes of previously stored data
- Purpose is typically to access information within the item (not just the key)
- Recall that arrays, linked lists are worst-case ***O*(*n*)** for either searching or inserting
- Even a hash table has worst-case ***O*(*n*)**

Need a data structure with optimal efficiency for searching and inserting.

What if order is important?

Symbol Table: ADT

- **insert** a new item
- **search** for an item (items) with a given key
- **remove** an item with a specified key
- **sort** the symbol table
- **select** the item with the k^{th} largest key
- **join** two symbol tables

Also may want construct, test if empty, destroy, copy...

Binary Search Tree

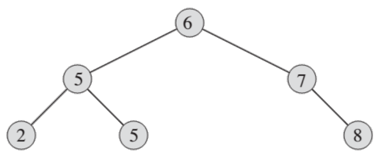
- The keys in a binary search tree satisfy the Binary Search Tree Property
  – The key of any node is:
    > keys of all nodes in its left subtree and
    ≤ keys of all nodes in its right subtree
- Essential property of BST is that `insert()` is as easy to implement as `search()`

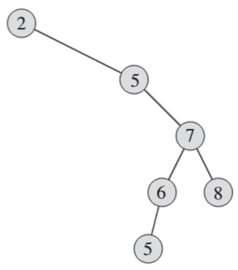
Binary Search Tree Property

The key of any node is:

> keys of all nodes in its left subtree and

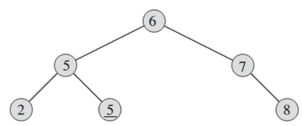
≤ keys of all nodes in its right subtree





Exercise

Write the output for inorder, preorder and post-order traversals of this BST



inorder: 2, 5, <u>5</u>, 6, 7, 8

preorder: 6, 5, 2, <u>5</u>, 7, 8

postorder: 2, <u>5</u>, 5, 8, 7, 6

```
void inorder(Node *x) {
  if (!x) return;
  inorder(x->left);
  print(x->key);
  inorder(x->right);
} // inorder()

void preorder(Node *x) {
  if (!x) return;
  print(x->key);
  preorder(x->left);
  preorder(x->right);
} // preorder()

void postorder(Node *x) {
  if (!x) return;
  postorder(x->left);
  postorder(x->right);
  print(x->key);
} // postorder()
```

Search

- How can we find a key in a binary search tree?

```
// return a pointer to node with key k if
// one exists; otherwise, return nullptr
Node *tree_search(Node *x, KEY k);
```

- BST Property - the key of any node is:
  - > keys of all nodes in its left subtree
  - ≤ keys of all nodes in its right subtree
- What are the average- and worst-case complexities?

Search

```
// return a pointer to node with key k if
// one exists; otherwise, return nullptr
Node *tree_search(Node *x, KEY k) {
  while (x != nullptr && k != x->key) {
    if (k < x->key)
      x = x->left;
    else
      x = x->right;
  } // while
  return x;
} // tree_search()
```

Iterative

Search

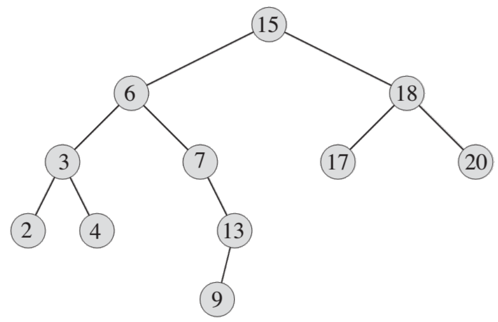
```
// return a pointer to node with key k if
// one exists; otherwise, return nullptr
Node *tree_search(Node *x, KEY k) {
  if (x == nullptr || x->key == k)
    return x;
  if (k < x->key)
    return tree_search(x->left, k);
  return tree_search(x->right, k);
} // tree_search()
```

Tail Recursive

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node

Search Example

```
tree_search(root, 9);
```



Search

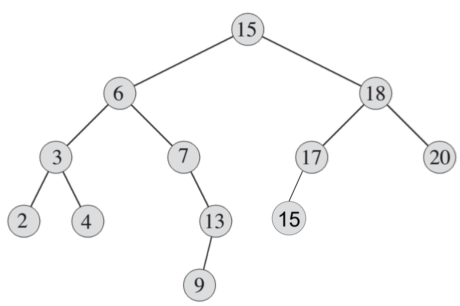
- Complexity is ***O*(*h*)**, where *h* is the (maximum) height of the tree
- Average-case complexity: ***O*(log *n*)**
  - Balanced tree
- Worst-case complexity: ***O*(*n*)**
  - "Stick" tree

Insert

- How do we insert a new key into the tree?
- Similar to search
- Start at the root, and trace a path downwards, looking for a null pointer to append the node

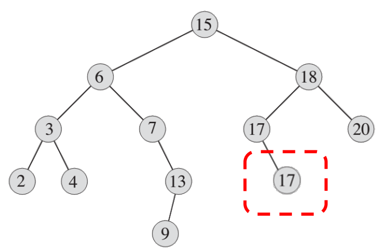
Insert Example

```
tree_insert(root, 15);
```



Insert with Duplicates

- For sets with no duplicates, use (<, >)
- For duplicates, we need a deterministic policy
  - Choose (<= , >) or (<, >=)… slides, STL use (<, >=)



Insert

```
void tree_insert(Node *&x, KEY k) {
  if (x == nullptr)
    x = new Node(k);
  else if (k < x->key)
    tree_insert(x->left, k);
  else
    tree_insert(x->right, k);
} // tree_insert()
```

- New node inserted at leaf
- Note the use of reference-to-pointer-to-Node
- Exercise: modify this code to set the parent pointer

Exercise

- Start with an empty tree
- Insert these keys, in this order:
  12, 5, 18, 2, 9, 15, 19, 17, 13
- Draw the tree
- Write a new order to insert the same keys which generates a worst-case tree
- How many worst-case trees are possible for *n* unique values?

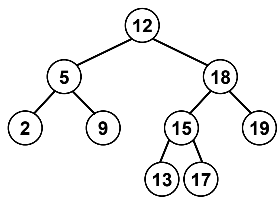
Complexity

- The complexity of insert (and many other tree functions) depends on the height of the tree
- Average-case (balanced): ***O*(log *n*)**
- Worst-case (unbalanced "stick"): ***O*(*n*)**
- Average-case:
  - Random data
  - Likely to be well-balanced

Exercise

- Write a function to find the Node with the smallest key
- What are the average- and worst-case complexities?

```
// Return a pointer to the Node with the min key
Node *tree_min(Node *x);
```

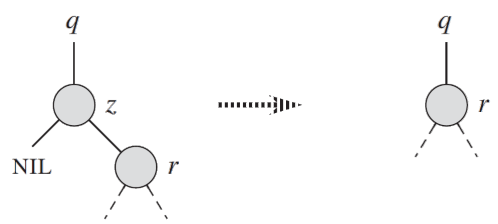


Remove

- What if we want to remove a node?
- To remove node *z*:
  1. *z* has no children (trivial)
  2. *z* has no left child
  3. *z* has no right child
  4. *z* has two children

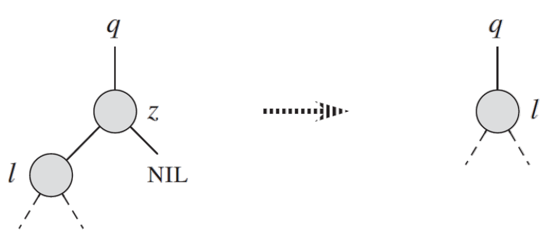
Remove: Easy Case #1

z has no left child: replace *z* by right child



Remove: Easy Case #2

z has no right child: replace *z* by left child

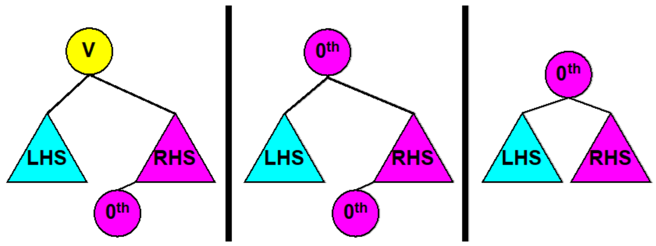


Remove: Hard Case

- *z* has left and right children
- Replace with a "combined" tree of both
- Key observation
  - All in LHS subtree < all in RHS subtree
  - Transplant smallest RHS node to root
    - Called the inorder successor
    - Must be some such node, since RHS is not empty
    - New root might have a right child, but no left child
  - Make new root's left child the LHS subtree

Remove: Hard Case

To Remove V:



Find smallest node in RHS

Replace V with smallest node's value

Remove V from RHS

Single-Function Remove 1/3

```
template <class T>
void BinaryTree<T>::remove(Node *&tree, const T &val) {
  Node *nodeToDelete = tree;
  Node *inorderSuccessor;

  // Recursively find the node containing the value to remove
  if (tree == nullptr)
    return;
  else if (val < tree->value)
    remove(tree->left, val);
  else if (tree->value < val)
    remove(tree->right, val);
  else {
```


Single-Function Remove 2/3

```
    // Check for simple cases where at least one subtree is empty
    if (tree->left == nullptr) {
      tree = tree->right;
      delete nodeToDelete;
    } // if
    else if (tree->right == nullptr) {
      tree = tree->left;
      delete nodeToDelete;
    } // else if
```


Single-Function Remove 3/3

```
    else {
      // Node to delete has both left and right subtrees
      inorderSuccessor = tree->right;

      while (inorderSuccessor->left != nullptr)
        inorderSuccessor = inorderSuccessor->left;

      // Replace value with the inorder successor's value
      nodeToDelete->value = inorderSuccessor->value;
      // Remove the inorder successor from right subtree
      remove(tree->right, inorderSuccessor->value);
    } // else
  } // else
} // BinaryTree::remove()
```


Summary: Binary Search Trees

- Each node points to two children (left, right), and possibly a parent
- All nodes are ordered: left < root ≤ right
- Modification of nodes
  - External is easy
  - Internal is more complicated
- In general, operations on BSTs are:
  - ***O*(log *n*)** average-case
  - ***O*(*n*)** worst-case

AVL Tree

- Self-balancing Binary Search Tree
- Named for <u>A</u>delson-<u>V</u>elsky, and <u>L</u>andis
- Start with a BST
- Add the *Height Balance Property*
  - For every internal node *v* of *T*, the heights of the children of *v* differ by at most *1*
  - Use *rotations* to correct imbalance
- Worst-case search/insert is now ***O*(log *n*)**

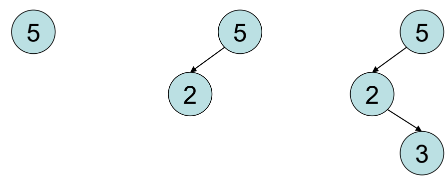
Tree Height

- Measured upward from leaf nodes; all of which have height equal to 1
- Independent from depth
- Recursive formula for a recursive data structure
  - height(empty) = 0;
  - height(node) = max(height(children)) + 1;

Is this an AVL tree?

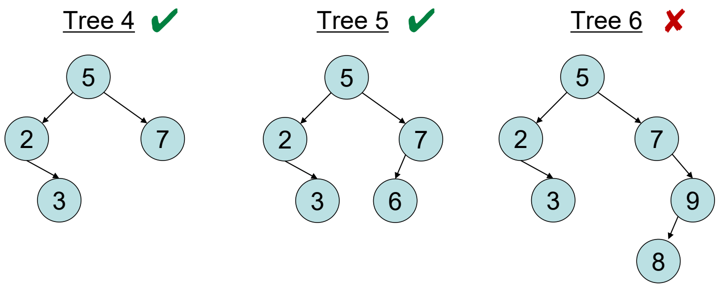
Height Balance Property: For every internal node *v* of *T*, the heights of the children of *v* differ by at most *1*.

<u>Tree 0</u> ✔ <u>Tree 1</u> ✔ <u>Tree 2</u> ✔ <u>Tree 3</u> ✘



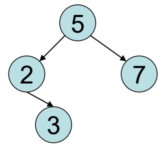
Is this an AVL tree?

Height Balance Property: For every internal node *v* of *T*, the heights of the children of *v* differ by at most *1*.



AVL Tree: Proof Setup

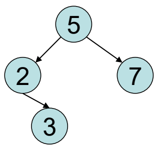
- *h*: height of tree
- *n(h)*: *minimum* number of nodes in AVL tree of height *h*
- *n(0) = 0* (Empty tree)
- *n(1) = 1* (Root-only tree)
- For *h* > 1, an AVL tree of height *h* contains:
  - Root Node
  - AVL Subtree of height *h - 1*
  - AVL Subtree of height *h* - 2



- Thus for h > 1, $n(h) = 1 + n(h - 1) + n(h - 2)$

Proof: Height Balance Property

- *h*: height of tree
- *n(h)*: *minimum* number of nodes in a tree of height *h*
- $n(h) = 1 + n(h - 1) + n(h - 2)$
- Knowing $n(h - 1) > n(h - 2)$ and $n(h) > 2n(h - 2)$, then by induction, $n(h) > 2^i n(h - 2i)$
- Closed form solution, $n(h) > 2^{h/2 - 1}$
- Taking logarithms: $h < 2 \log n(h) + 2$
- Thus the height of the tree is ***O*(log *n*)**



AVL Tree Algorithms

- Search is the same as a BST
- Sort (same as BST)
  - Insert nodes one at a time
    - What is worst-case complexity now?
  - Perform an inorder traversal
    - Still ***O*(*n*)** for this step

AVL Tree Insert

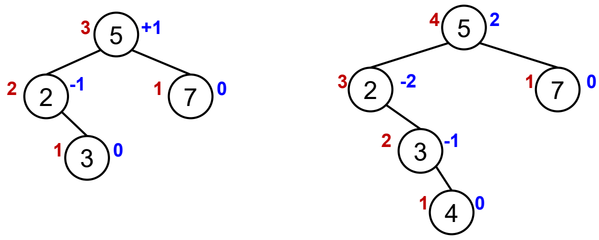
- The basic idea:
  1. Insert like a BST
  2. Rearrange tree to balance height
- Each node records its height
- Can compute a node's *balance factor*
  - `balance(n) = height(n->left) – height(n->right)`
- A node that is AVL-balanced:
  - `balance(n) = 0`: both subtrees equal
  - `balance(n) = +1`: left taller by one
  - `balance(n) = -1`: right taller by one
- `|balance(n)| > 1`: node is out of balance

Balance Factor Example

- What is the **height** for each node?

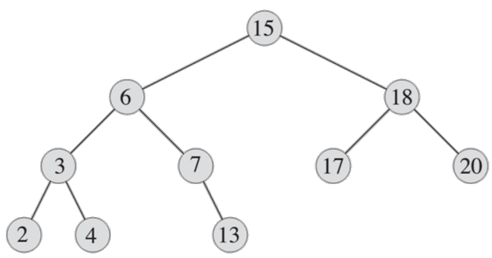
  `height(node) = max(height(children)) + 1;`
- What is the **balance factor**?

  `balance(n) = height(n->left) – height(n->right)`



Balance Factor Exercise

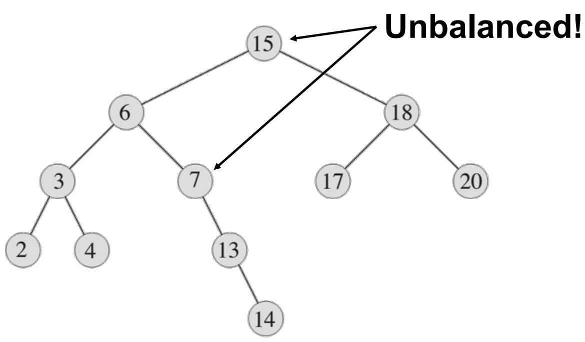
Label the balance factor on each node



`bal(n) = height(n->left) – height(n->right)`

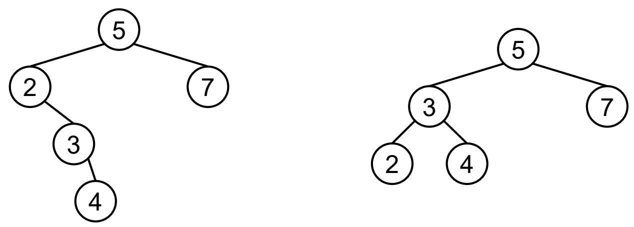
Insert (begins like BST)

`tree_insert(root, 14);`



Rotations

- We use *rotations* to rebalance the binary tree
  - Swap the roles of a parent and one of its children
  - BUT preserve the BST ordering property



- *Rotation is a local change involving only three pointers and two nodes*

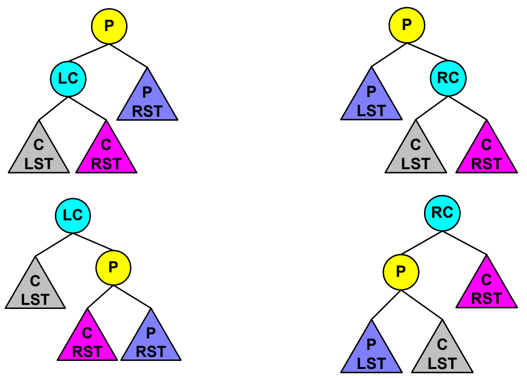
Rotations

- We use *rotations* to rebalance the binary tree
  - Interchange the role of a parent and one of its children in a tree…
  - Preserve the BST ordering among the keys in the nodes
- The second part is tricky
  - Right rotation: copy the right pointer of the left child to be the left pointer of the old parent
  - Left rotation: copy the left pointer of the right child to be the right pointer of the old parent

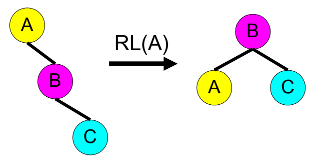
Rotations

Rotate Right: RR(P)

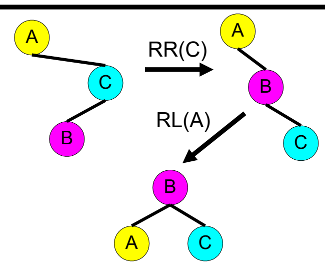
Rotate Left: RL(P)



Solution



- First tree is easy: rotate-left(A)
- single rotation



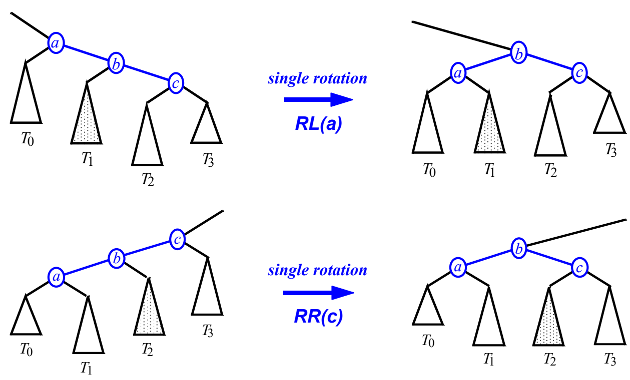
- Second is harder: rotate-right(C) rotate-left(A)
- double rotation

Insert

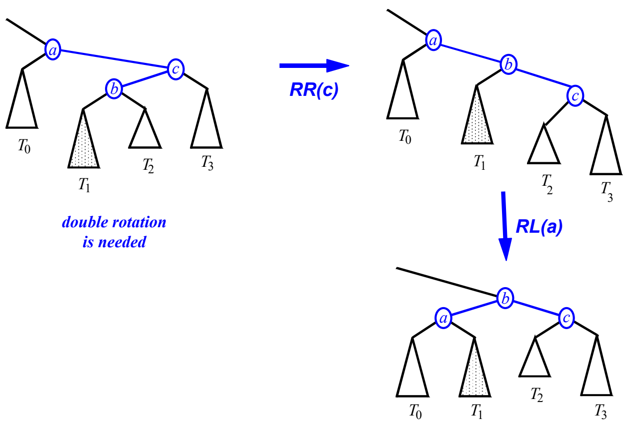
Four Cases

1. Single left rotation
   - RL(a)
2. Single right rotation
   - RR(a)
3. Double rotation right-left
   a. RR(c)
   b. RL(a)
4. Double rotation left-right
   a. RL(a)
   b. RR(c)

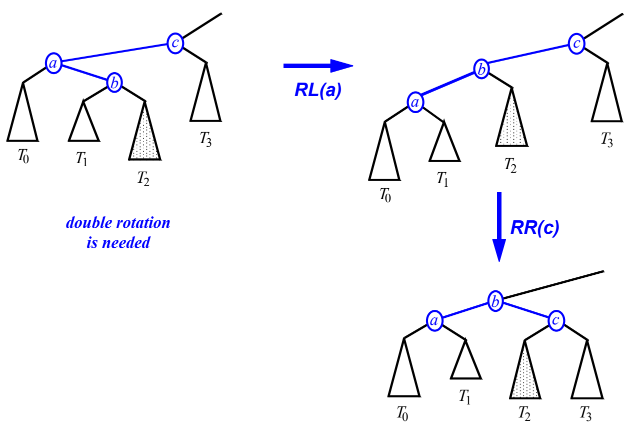
Single Rotations

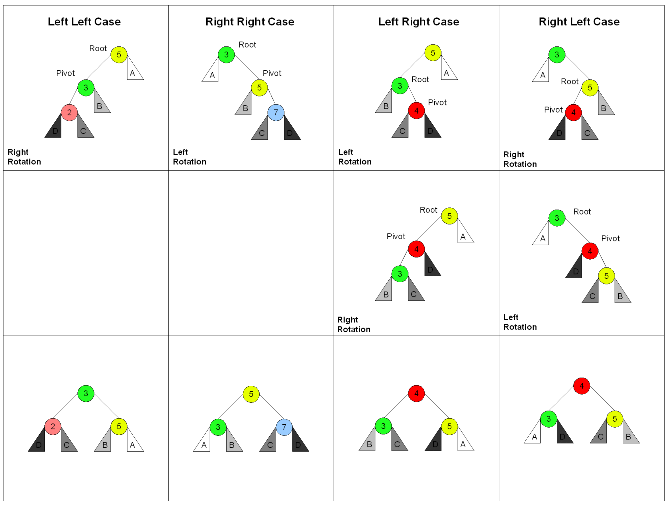


Double Rotations



Double Rotations





http://upload.wikimedia.org/wikipedia/commons/c/c4/Tree_Rebalancing.gif

Checking and Balancing

```
Algorithm checkAndBalance(Node *n)
  if balance(n) > +1
    if balance(n->left) < 0
      rotateL(n->left)
    rotateR(n)
  else if balance(n) < -1
    if balance(n->right) > 0
      rotateR(n->right)
    rotateL(n)
```

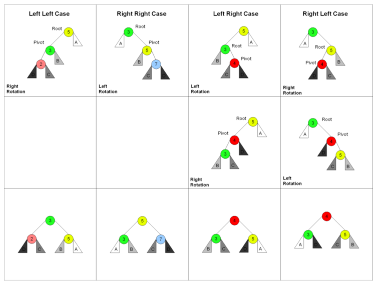
- Outermost if:
  Q: Is node out of balance?
  A: > +1: left too big
  < -1: right too big
- Inner ifs:
  Q: Do we need a double rotation?
  A: Only if signs disagree

Checking and Balancing

- As insert finishes, after every recursive call, update height of current node, then call `checkAndBalance()` on every node along the insertion path
- What is the time complexity of this?
- How many nodes need to be "fixed" after an insert?

Rotation Exercise

- Insert these keys into an AVL tree, rebalancing when necessary
- 3, 2, 1, 4, 5, 6, 7, 16, 15, 14



Rotation Exercise

- Insert 3… 2… 1…
- Unbalanced: RR(3)
- Insert 4… 5…
- Unbalanced: RL(3)
- Insert 6…
- Unbalanced: RL(2)
- Insert 7…
- Unbalanced: RL(5)
- Insert 16… 15…
- Unbalanced: RR(16), RL(7)
- Insert 14…
- Unbalanced: RR(15), RL(6)

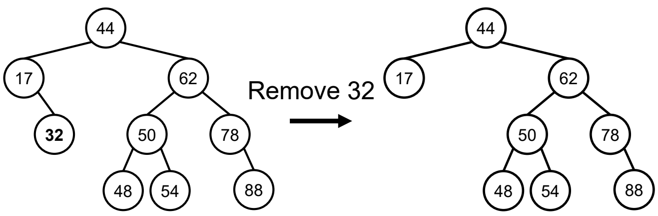
Animation screen capture from visualgo.net

AVL Tree Remove

1. Remove like a BST
   - Key observation: **All keys in LHS ≤ all in keys RHS**
   - Rearrange RHS so that its smallest node is its root
     - Must be some such node, since RHS is not empty
     - New RHS root has a right child, but no left child
   - Make the RHS root's left child the LHS root
2. Rearrange tree to balance height
   - Travel up the tree from the parent of removed node
   - At every unbalanced node encountered, rotate as needed
   - This restructuring may unbalance multiple ancestors, so continue check and rebalance **up to the root**

AVL Tree Remove

- Remove as in a binary search tree
- Rebalance if an imbalance has been created

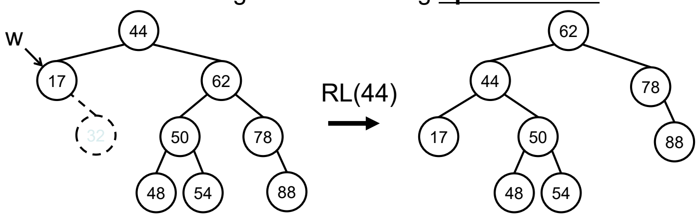


Before: balanced After: unbalanced

Rebalance Required!

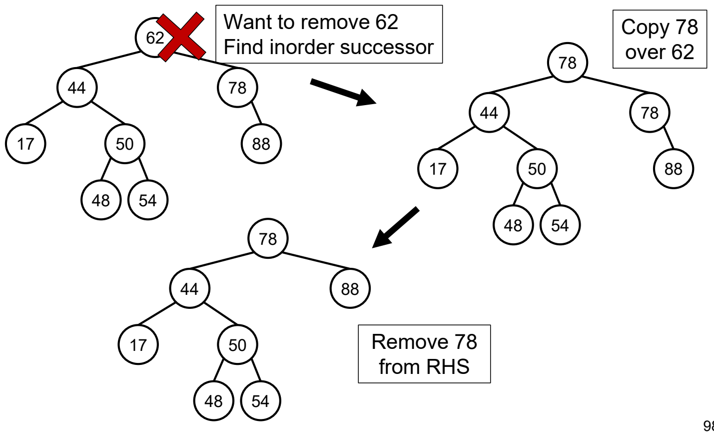
Rebalancing after a Remove

- Travel up the tree from *w*, the parent of the removed node
- At every unbalanced node encountered, rotate as needed
- This restructuring may unbalance multiple ancestors, so continue checking and rebalancing **up to the root**

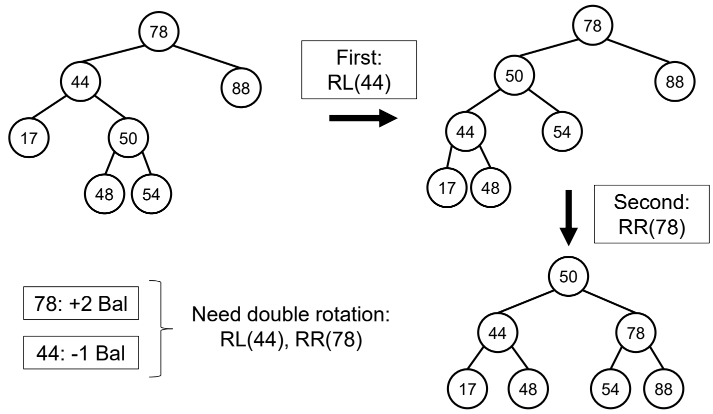


How many nodes need to be "fixed" after a remove?

Rebalancing after a Remove



Rebalancing after a Remove

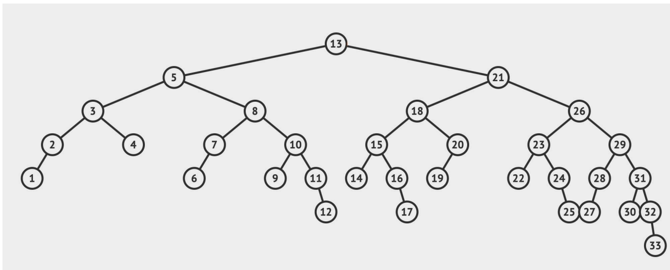


Rebalancing: 1 or More?

- A “fix” (single or double rotation) shortens a subtree that is too tall
- When inserting, the new node is the source of any imbalance, and a single fix will counteract it and repair the entire tree
- When removing, a deleted node can only create an imbalance by making subtrees shorter
- If a shortened subtree was already shorter than its sibling, a fix is needed at a higher level, so multiple fixes may be required

Rebalancing: 1 or More?

- Values 1-33 inserted into an AVL in a particular order
- Resulting tree is balanced
- Multiple fixes required when 4 is removed



Animation screen capture from visualgo.net

Useful Website

- https://visualgo.net/en/bst
- Close the help
- Click on “AVL TREE” near the top
- You can insert/remove several nodes at once, or one at a time, slow down or speed up the demo

Summary

- Binary Search Tree
  - Worst-case insert or search is ***O*(*n*)**
- AVL Tree
  - Worst-case insert or search is ***O*(log *n*)**
  - Must guarantee **height balance property**
- Operations
  - Search: ***O*(log *n*)** (same algorithm as BST, but faster)
  - Insert: ***O*(log *n*)** (Starts like BST, then may rebalance)
  - Remove: ***O*(log *n*)** (Starts like BST, then may rebalance)
  - Sort: ***O*(*n* log *n*)** to build the tree, ***O*(*n*)** to do inorder traversal
- Rotation:
  - ***O*(1)** cost for single rotation, ***O*(1)** cost for double rotation