William McAllister

DATA STRUCTURES AND ALGORITHMS USING JAVA

Trees

Chapter 7

Click to proceed

How To View This Presentation

This presentation is arranged in outline format. To view the slides in proper order

- For each slide
  - Read the entire slide's text
  - Then click the links (if any) in the text, in the order they appear in the text
  - Then click the or button at the bottom-right side of the slide
  - Do not use the space bar to advance

Overview

- Structures based on trees
  - Are accessible in the key field mode
  - Like linked list, normally do not require contiguous memory and are normally dynamic
  - Unlike linked lists, their average operation speed can be O(log2n)
- Binary trees are the most common trees, most often implemented as binary search trees
- An array-based representation of a binary tree is used for some applications
- Java’s TreeMap class is a form of a binary search tree
- All classic structures have their niche

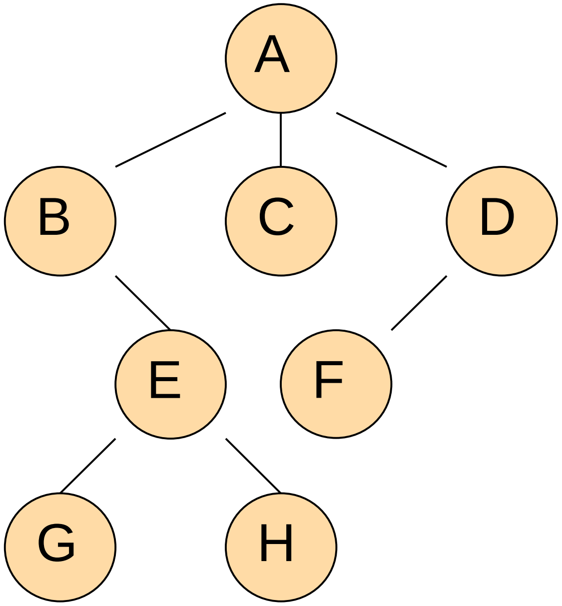
Trees

- All trees are depicted using a standard high-level graphic

- All trees share a common terminology

Graphical Depiction of a Tree

- Nodes are represented by circles
- Annotation inside the circle is normally the contents of the key field
- Lines connecting the circles (nodes) indicate downward traversal paths
  - Analogous to the arrows in linked list depictions

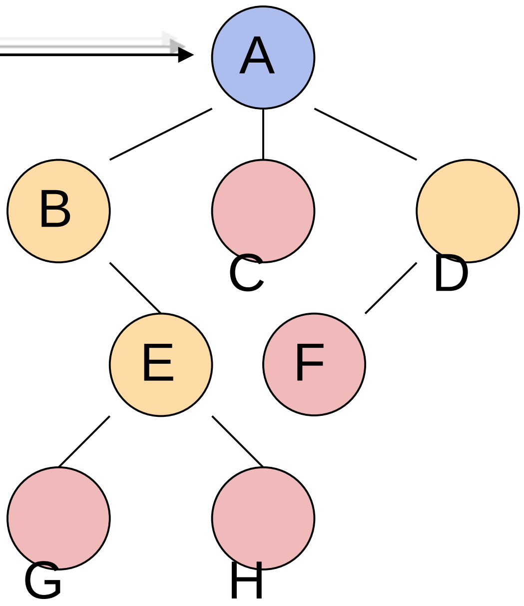


Terminology of Trees

- Some terms follow an arbor analogy
  - Directed (or general) tree, root node, leaf node
- Other terms follow a family analogy
  - Parent node, child node, grandparent node
- Other terms include
  - Outdegree of a node and tree
  - Levels of a tree
  - Visiting a node
  - Traversing a tree
- An understanding of these terms is an essential to the study of tree structures

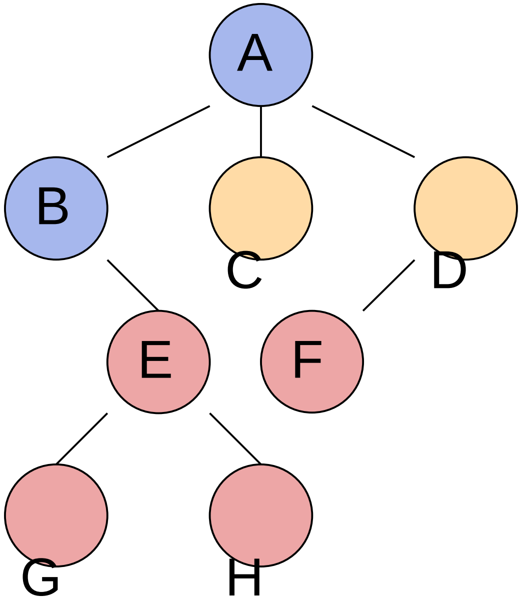
Arbor Terminology Analogies

- *Directed* (or general) *tree*
  - Has a unique first node, the ***root node***
  - Every other node has one, and only one, node before it
  - All nodes have 0, or 1, or 2, or 3, or….. nodes after them (G and H after E, none after H; …)
- A node with no node after it is a ***leaf*** node



Family Terminology Analogies

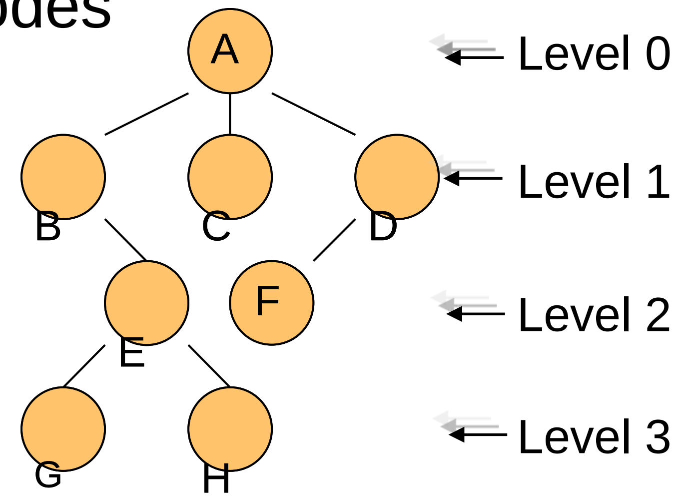
- A node's unique predecessor is its *parent* (B is E's parent)
- A *child* is a node that comes directly after a node (E is B's child, H is E' child)
- A *grandchild* node is a child of a child (H is B's grandchild)
- A *grandparent* node is a parent of a parent (B is H's grandparent
- Analogy is extended to *great-grandchildren* and *parents*…



More Terminology of Trees

- Outdegree of a
  - Node: number of children (E is 2)
  - Tree: Highest outdegree of all nodes (3 for this tree)
- Levels of a tree

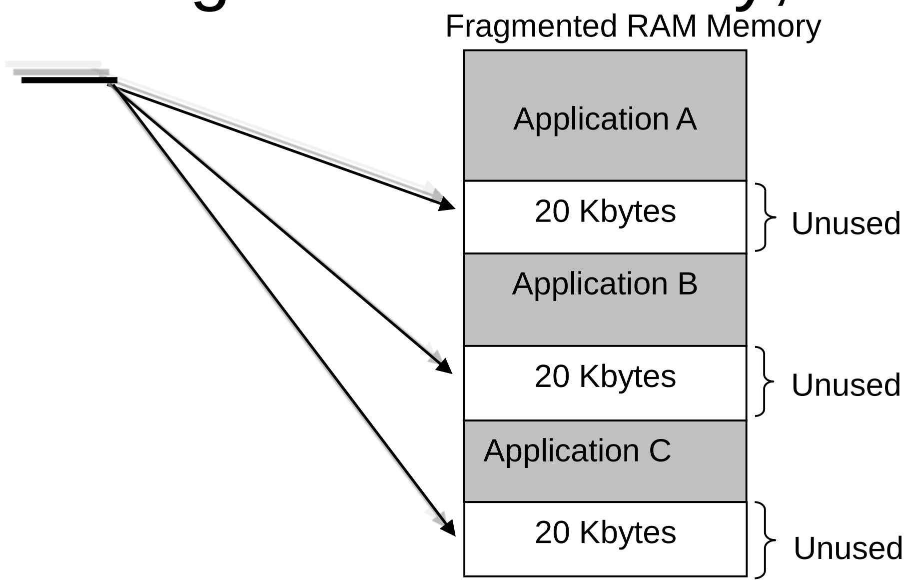
See right of tree



- Visit a node
  - Locating a node and performing an operation on it (e.g. output it)
- Traversing a tree
  - Visiting all nodes once, and only once

Noncontiguous Structures

- Do not require contiguous (sequential) memory
  - Client need not specify the maximum number of nodes to be stored in the structure
  - They can utilize fragmented memory, all 60 Kbytes



Dynamic Structures

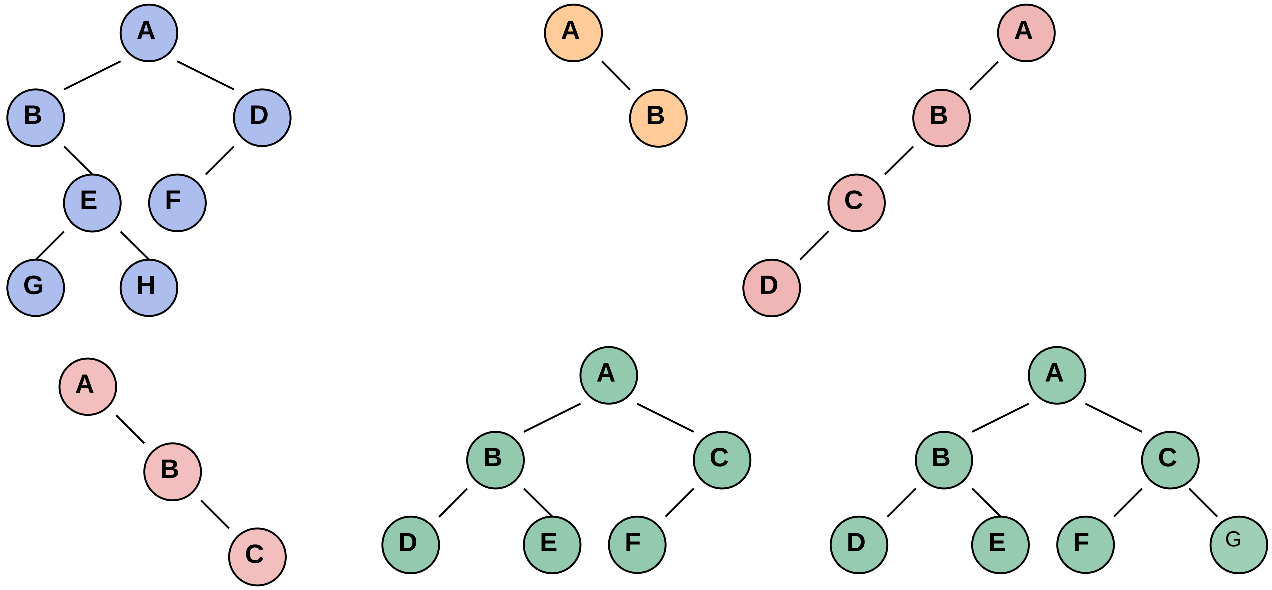
- Expand *and* contract at runtime during *every*
  - Insert operation (expansion)
  - Delete operation (contraction)
- Therefore the are memory frugal
  - Never assigned more memory than they require

Binary Trees

- A binary trees is, by definition, a subset of tree
- The terminology of binary trees extend the terminology of trees
  - The family and arbor analogies
- Binary trees have a set of mathematics associated with them
- There implementation can be array based or linked based

Definition of a Binary Tree

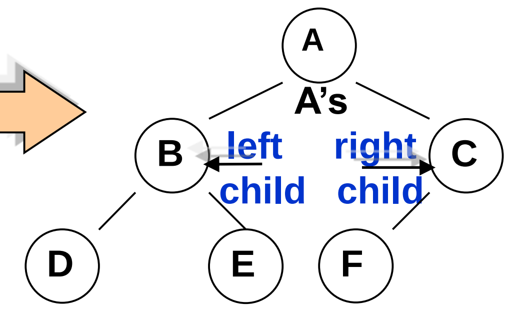
- A *binary tree* is a directed tree with a maximum outdegree of 2



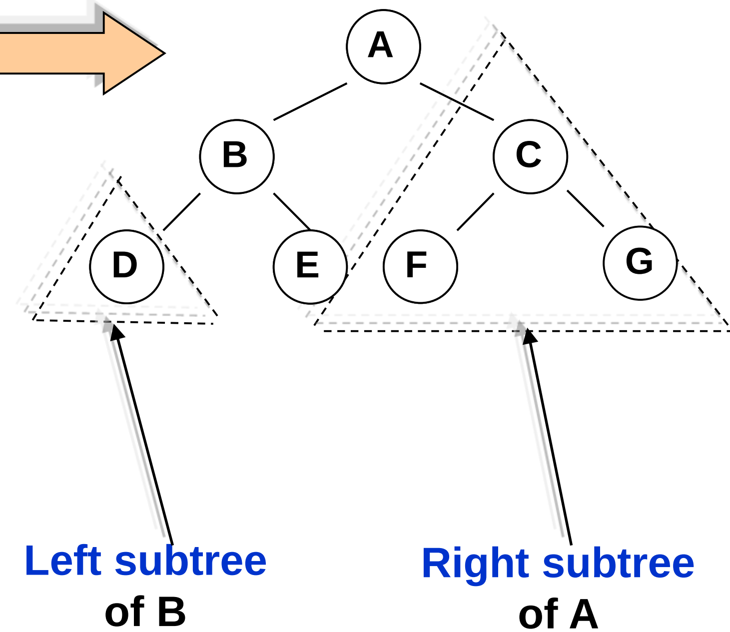
- Each parent can have at most two children

Extended Terminology of Binary Trees

- A *balanced tree* has all levels full except possibly the highest level



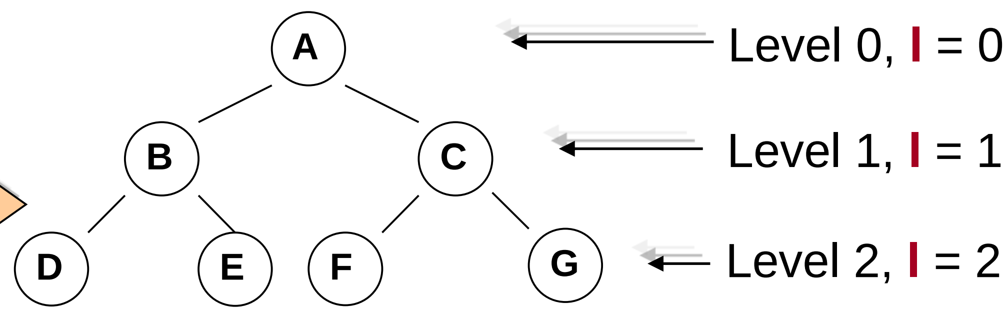
- A *complete tree* has all levels full
- A *complete left tree* is a balanced tree with all nodes on the highest level on the left (e.g., the upper tree)



Mathematics of Binary trees

- Maximum number of nodes at level $l = 2 ^ l$
- Maximum number of nodes in a tree with L levels $= 2 ^ L - 1$
- Number of levels in a *complete* tree with N nodes $= \log_2(N + 1)$
- Minimum number of levels in a tree with N nodes $= \text{ceiling}[\log_2(N + 1)]$ (a *balanced* tree)

Above can be verified using this complete tree



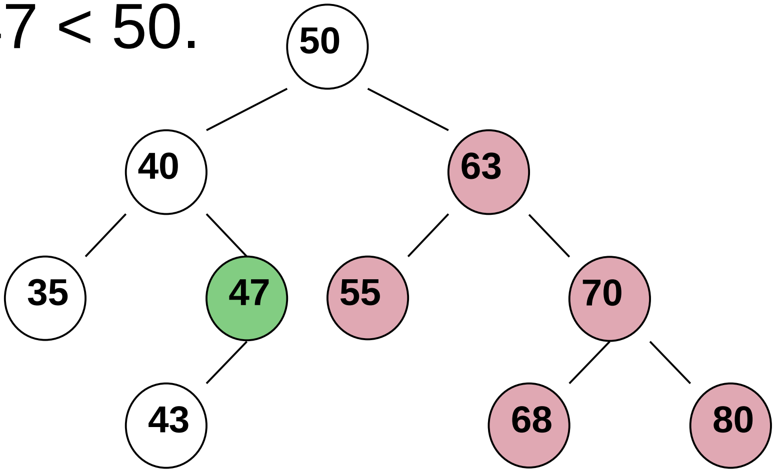
A 3 Level Tree, L = 3, N = 7

Binary Search Trees

- A binary search tree, by definition, is a binary tree arranged to reduce search time
- As with all binary trees, the meaning of the circle graphic at the implementation level depends on the implementation approach
- The operation algorithms of structures based on binary search trees are complicated
- However their performance is *usually* good, and *always* good for implementations that keep them balanced

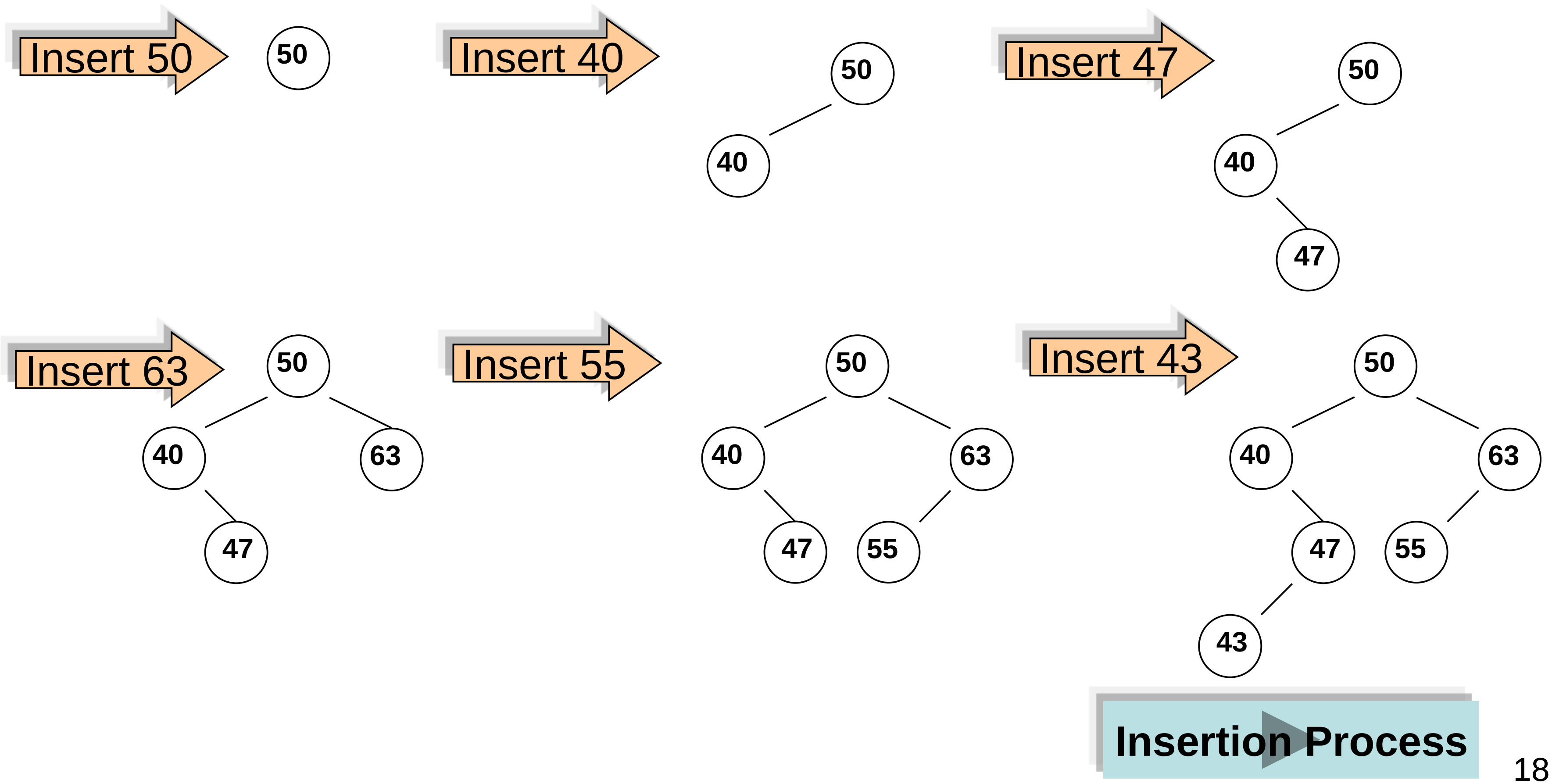
Definition of a Binary Search Tree

- A binary search tree is a binary tree in which every parent is
  - *Greater than* all nodes in its *left subtree*
  - *Less than* all the nodes in its *right subtree*
- The Insert algorithm imposes this ordering
- The ordering can greatly reduce search times
  - Looking for node 47, we can eliminate 50's entire right subtree because 47 < 50.



Progressive Buildup of a Binary Search Tree

- Node insertion order: 50, 40, 47, 63, 55, 43

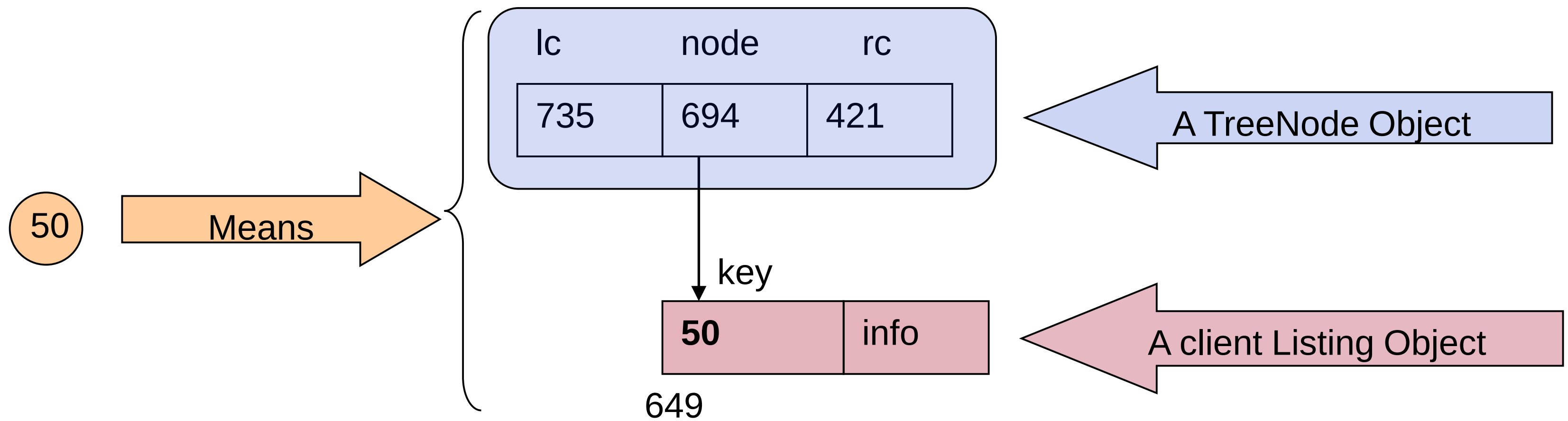


Binary Search Tree Insertion Process

1. The first node inserted becomes the root node.
2. For any subsequent node, consider the root node to be a root of a subtree, and start at the root of this subtree.
3. Compare the new node's key to the root node of the subtree.
   - 3.1 If the new node's key is *smaller,* then replace the subtree with the root's *left* subtree.
   - 3.2 Else, replace the subtree with the root's *right* subtree.
4. Repeat step 3 until the new subtree is empty.
5. Insert the node as the root of this empty subtree.

Linked Implementation Level Meaning of the Graphical Circle Symbol

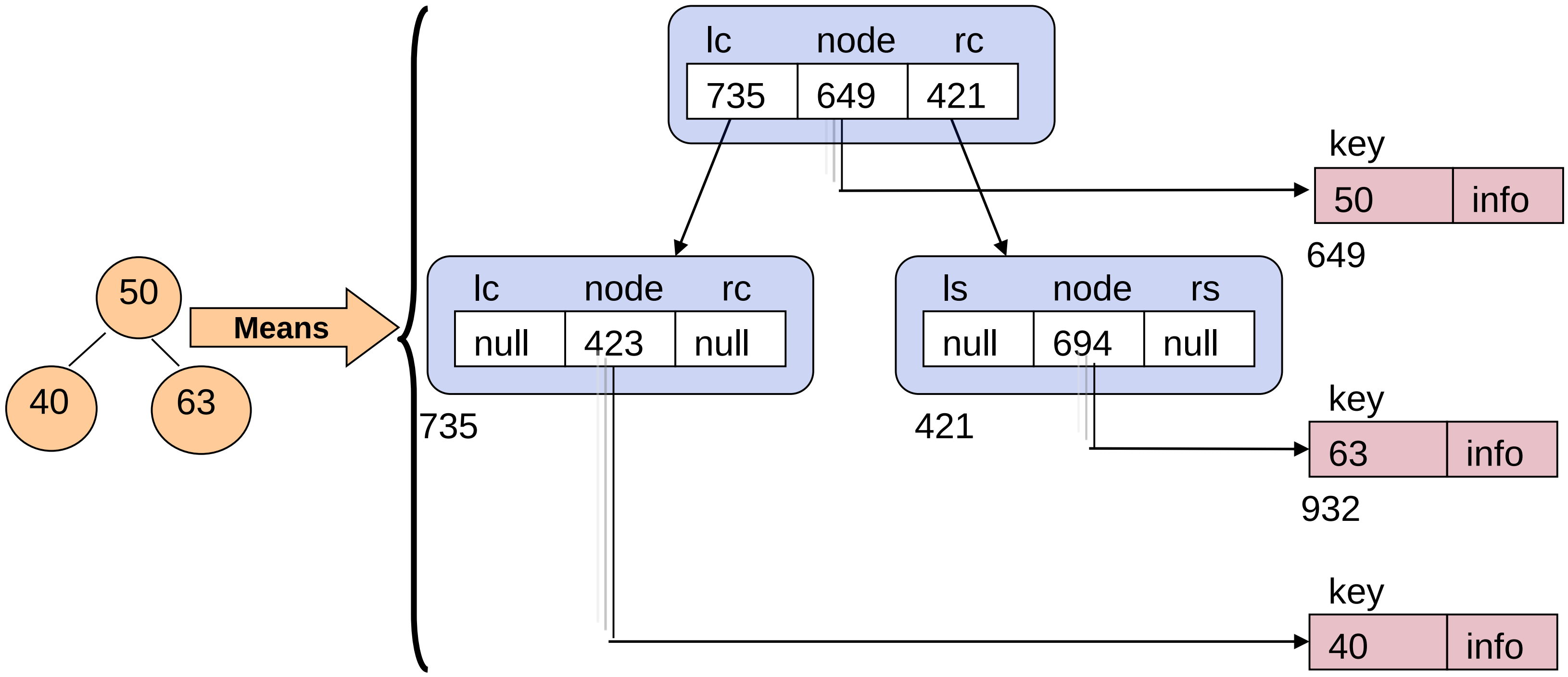
- The circle symbol represents
  - A three field TreeNode Object that references a client object (e.g., a Listing object)



- The fields lc and rc reference the nodes left and right children

Implementation Level Meaning of the Standard Depiction of a Binary tree

- A binary tree is really a tree of TreeNode Objects

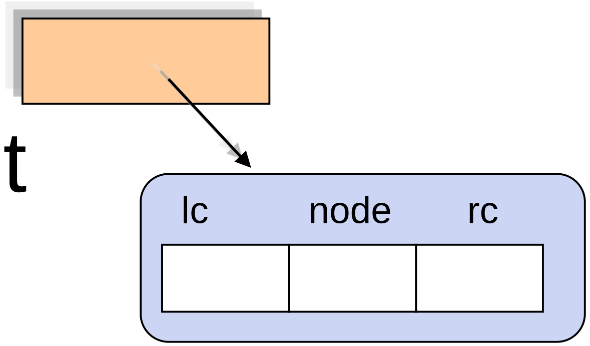


Operation Algorithms of a Binary Search Tree Linked Implementation

- The initialization is simple
- An algorithm to locate a node given its key is used by the
  - Fetch algorithm
  - Insert algorithm (with a little trickery)
  - Delete algorithm, which is very complicated and developed as three separate cases
  - Update algorithm, developed as a Delete followed an Insert operation
- Usually a traversal operation is added

Binary Search Tree Initialization Algorithm Linked Implementation

- This implementation uses a reference variable **`root`** to store address of the root Node object

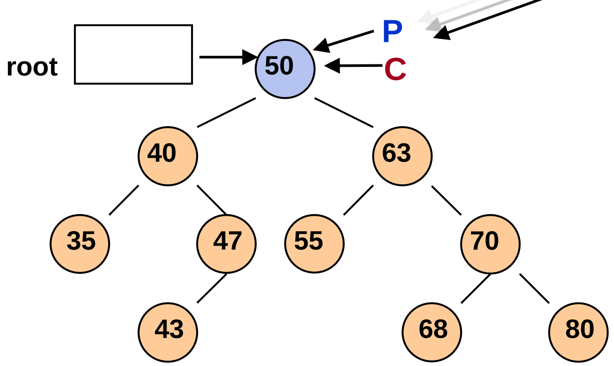


- The initialization algorithm sets **`root`** to **null** (empty tree)

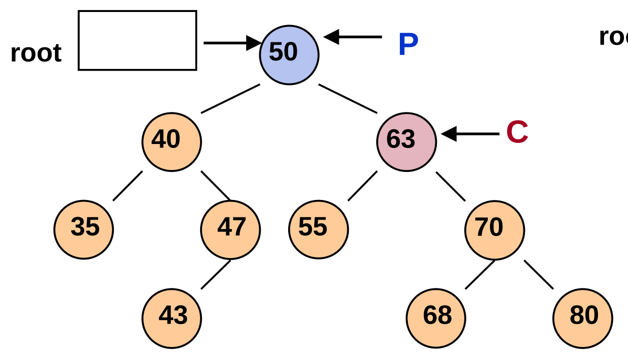
root = null;

Algorithm to Locate a Node Given Its Key (55)

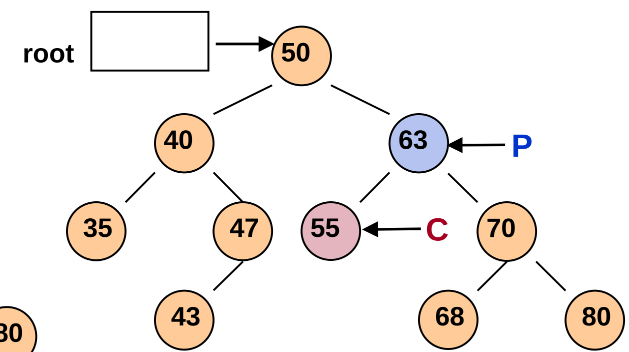
- The algorithm positions two reference variables P and C as shown below; P follows C down tree



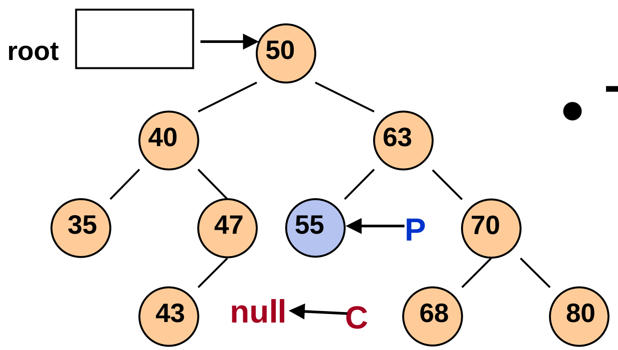
55 > 50 go right



55 < 63 go left



End when looking for 55;
P points to 55's parent



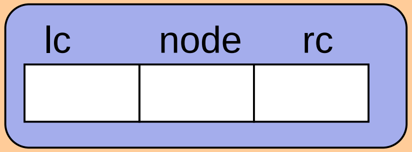
End when looking for 52

- The algorithm ends when the node is found *or* C == null

Pseudocode

Algorithm to Locate a Node Given Its Key -- findNode --

```
P = root;
C = root;
while(C != null)
{  if(targetKey == C.node.key) // node found
      return true;
   else  // continue searching
   {  P = C;
      if(targetKey < C.node.key) // move into left subtree
         C = C.lc;
      else  // move into right subtree
         C = C.rc;
   } // end else clause
} // end while
return false;
```

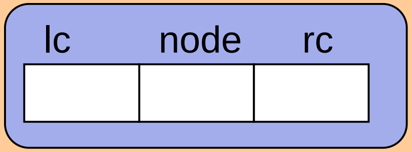


Recursive version

Recursive Version of findNode

boolean findNode(root, targetKey, P, C)

```
if(root == null)  // first base case
   return false;
C = root;
if(targetKey == C.node.key) // second base case
   return true;
P = C;
if(targetKey < C.node.key) // look in the left subtree
   root  = C.ls;
else  // look in the right subtree
   root  = C.rs
return findNode(root, targetKey, P, C); // reduced problem
                                        // and general solution
```



Binary Search Tree Fetch Pseudocode Linked Implementation

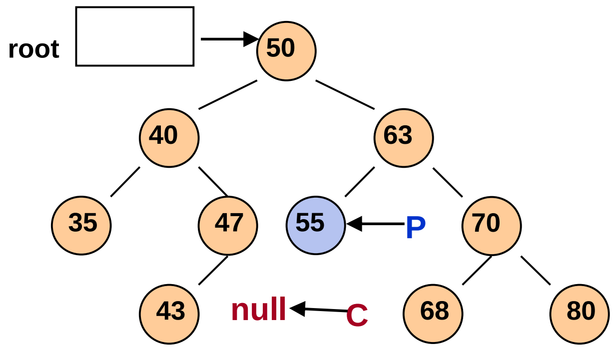
- Assuming findNode's pseudo signature is

  boolean findNode(root, targetKey, P, C)

1. found = findNode(root, targetKey, P, C);
2. **if**(found == **true**)
3. **return** C.node.deepCopy();
4. **else**
5. **return null**;

Trick Used by the Binary Search Tree Insert Algorithm

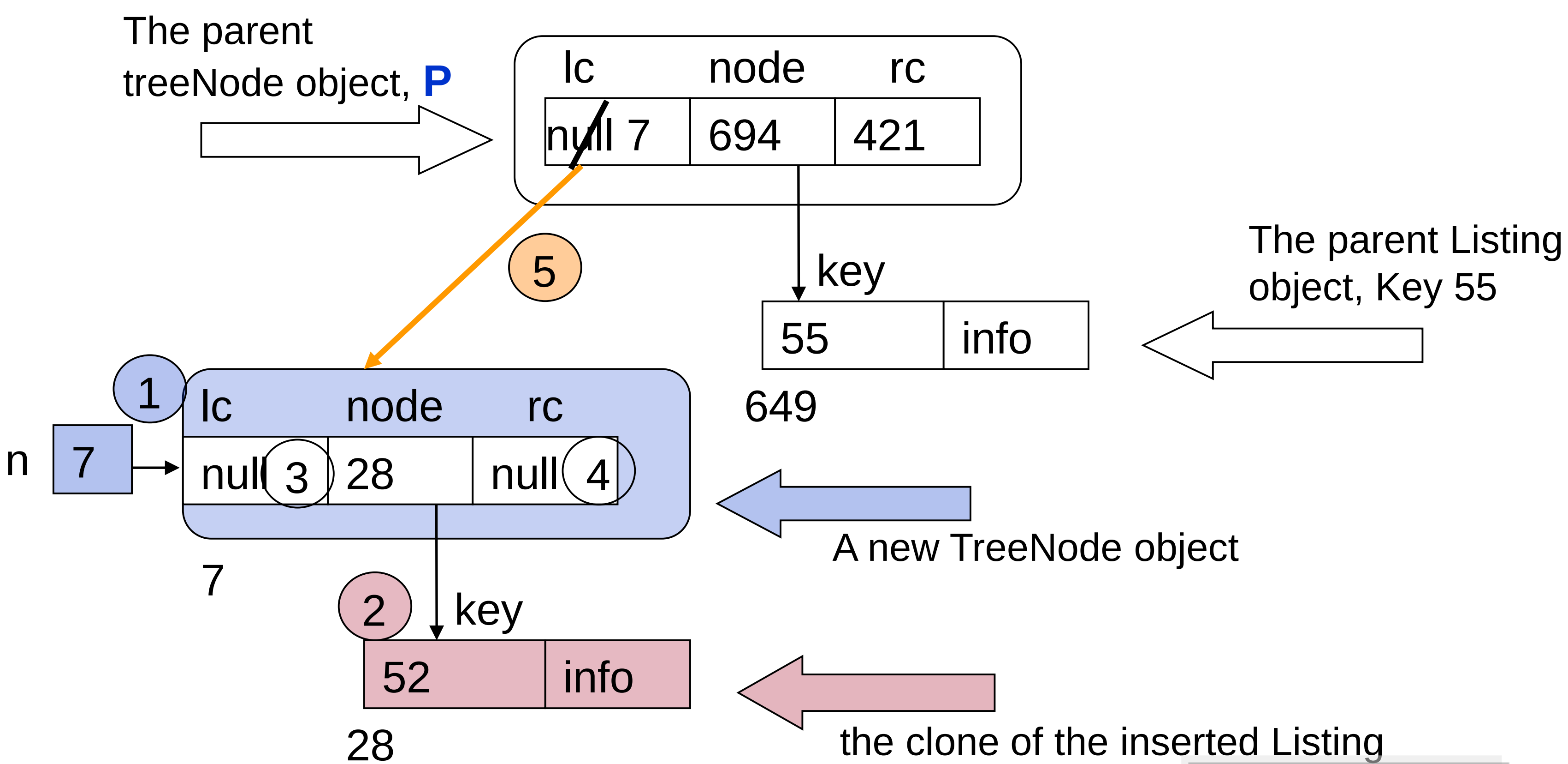
- A trick is used to locate the new node's parent
  - findNode is invoked and passed the node's key
  - Since the key is not yet in the structure, findNode ends with C == **null** as shown
  - P is then pointing a node that would have been be the parent of the given key, if it was in the structure



Algorithm

Graphical Representation of the Binary Search Tree Insert Algorithm

- Five steps: 1 2 3 4 5 shown below



Pseudocode

Binary Search Tree Insert Pseudocode Linked Implementation

```
1.      TreeNode n = new TreeNode();
2.      n.node = newListing.deepCopy(); // copy the node and make it a leaf node
3.      n.lc = null;
4.      n.rc = null;
5a.  if (root == null) // the tree is empty
5b.       root = n;
5c.  else // the tree is not empty
5d.  {   findNode(root, newListing.key,  P,  C); // find the new node's parent
5e.        if (newListing.key < P.node.key) // new node is parent's left child
5f.             P.lc = n;
5g.        else  // new node is parent's right child
5h.             P.rc = n;
5i.    }
```

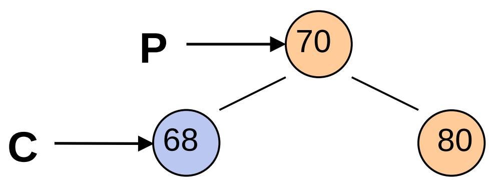
Binary Search Tree Delete Algorithm

- The algorithm is divided into three cases
- The node to be deleted has
  - Case 1: *no* Children (is a leaf)
  - Case 2: *one* child, or subtree
  - Case 3: *two* children, or subtrees
- Case 1 is the simplest, Case 3 is the most complex
- Each case uses findNode to locate the node to be deleted *and* its parent

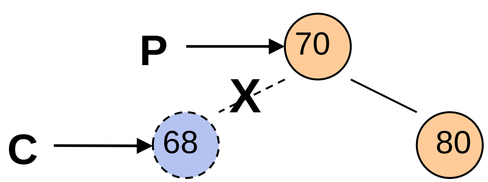
Graphical Representation of the Binary Search Tree Delete Algorithm, Case 1

The node to be deleted is a

left child

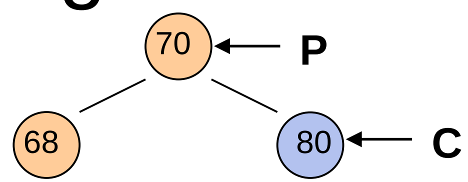


Before* the Deletion of Key *68

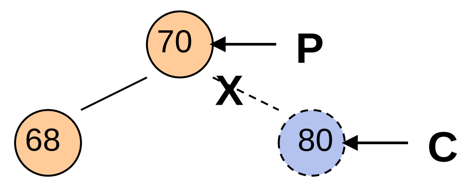


After* the Deletion of Key *68

right child



Before* the Deletion of Key *80



After* the Deletion of Key *80

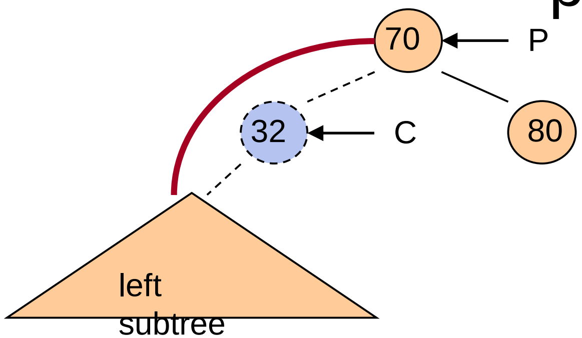
Pseudocode

The Binary Search Tree Delete Algorithm, Case 1: Deleted Node Has No Children

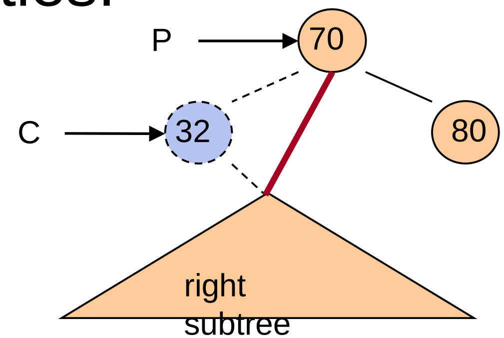
```
    found =  findNode(root, targetKey, P, C);
    if(found == false) // node not found
       return false;
    if (C.lc == null  &&  C.rc == null) // Case 1
    {  if (P.lc == C) // the deleted node is a left child
          P.lc = null;
       else               // the deleted node is a right child
          P.rc = null
       return true;
} // end of Case 1
```

Graphical Representation of the Binary Search Tree Delete Algorithm, Case 2

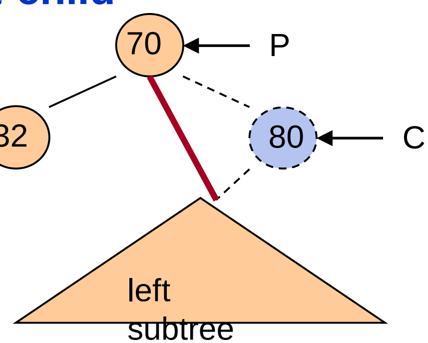
The node to be deleted has one child or subtree. Four possibilities:



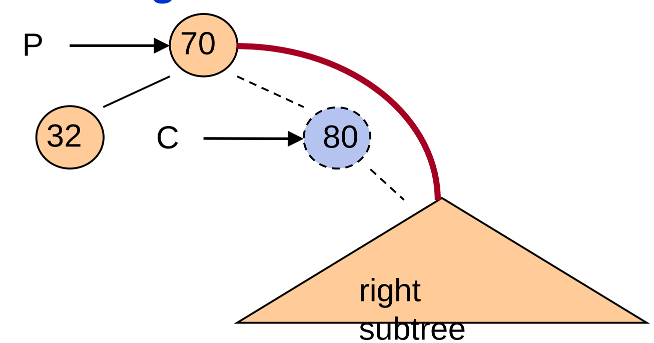
Deleted node is a *left* child and has a *left* child



Deleted node is a *left* child and has a *right* child



Deleted node is a *right* child and has a *left* child



Deleted node is a right child and has a *right* child

Pseudocode

The Binary Search Tree Delete Algorithm, Case 2: Deleted Node Has One Child

```
found =  findNode(root, targetKey, P, C);
if(found == false) return false; // node not found
if(C.lc != null && C.rc == null || C.rc != null && C.lc == null) // Case2
{  if(P.lc == C)  // deleted node is a left child,
     {  if(C.lc != null)  // and deleted node has a left child
            P.lc = C.lc;
        else
            P.lc = C.rc;
     } // end of deletion of a left child
   else  // deleted node is a right child
   {  if(C.lc != null) // and deleted node has a left child
           P.rc = C.lc;
      else
           P.rc = C.rc;
    } // end of deletion of a right child
    return true;
} // end of Case 2
```

Binary Search Tree Delete Algorithm Case 3

- The deleted node has two children or subtrees
- This case has *four possibilities*:

3a. C is a ***left*** child, and its left child ***has a*** right subtree

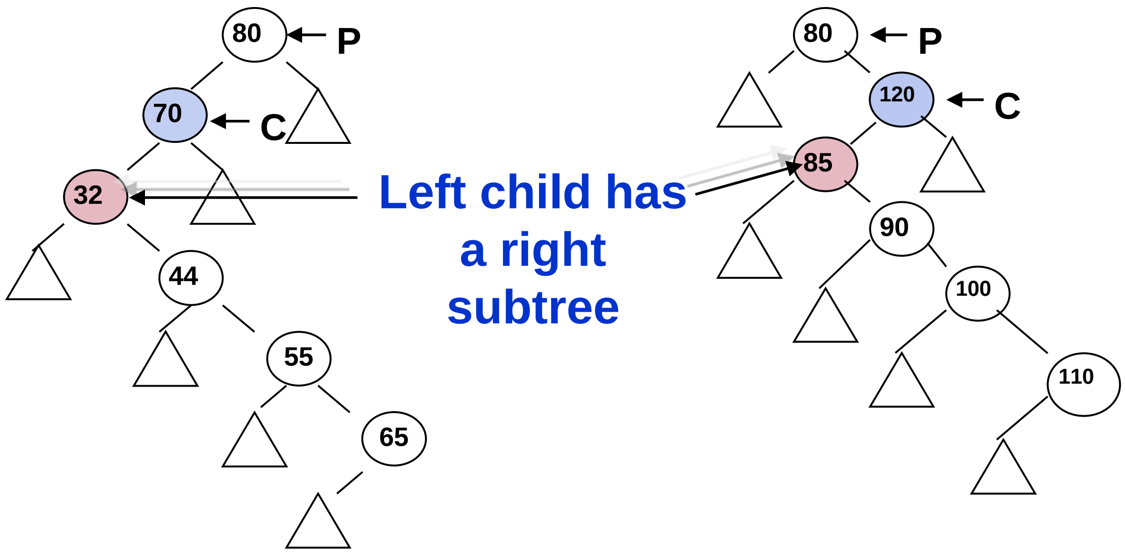
3b. C is a ***right*** child, and its left child ***has a*** right subtree

3c. C is a ***left*** child, and its left child ***has no*** right subtree

3d. C is a ***right*** child, and its left child ***has no*** right subtree

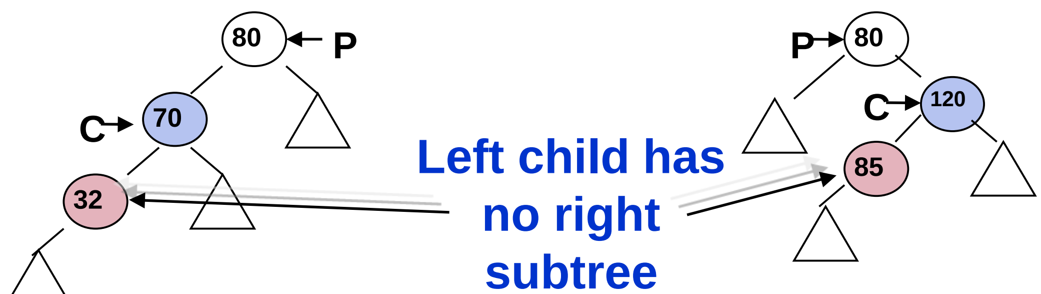
- The four possibilities are combined into one algorithm

The Four Possibilities for Case 3 of the Binary Search Tree Delete Algorithm



Case 3a, C is a Left Child

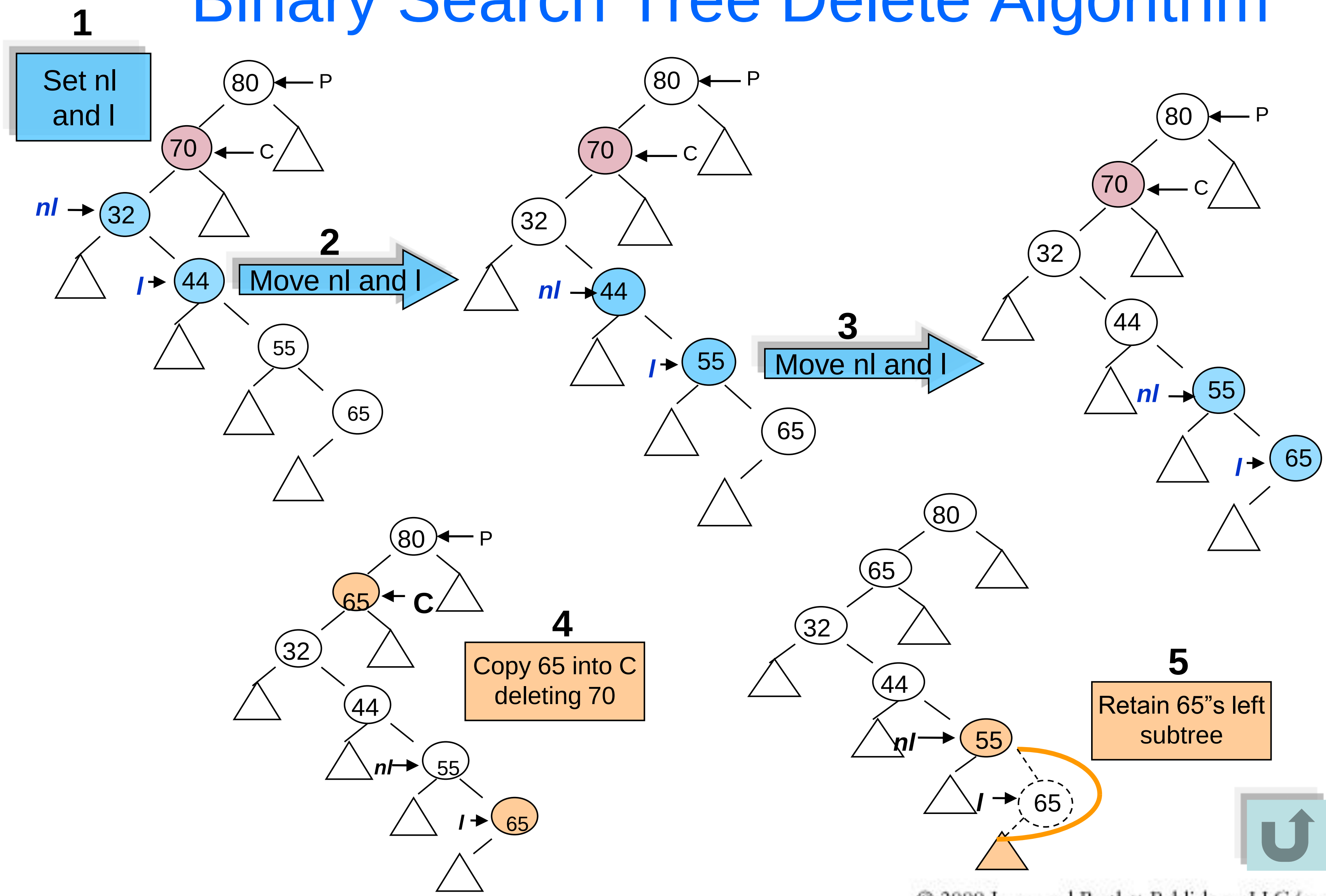
Case 3b, C is a Right Child



Case 3c, C is a Left Child

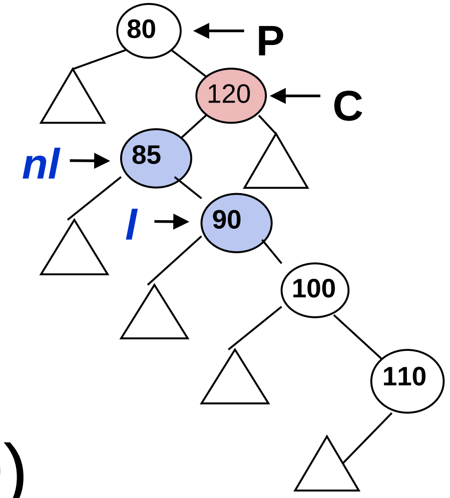
Case 3d, C is a Right Child

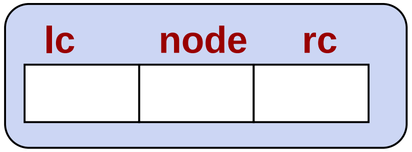
Graphical Representation of Case 3a of the Binary Search Tree Delete Algorithm



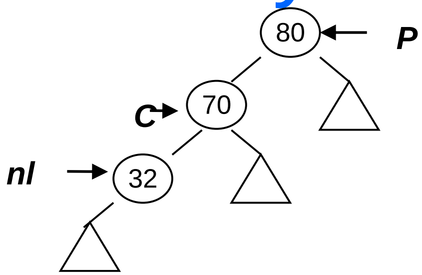
Case 3b of the Binary Search Tree Delete Algorithm

- This process is the same as Case 3a
  - Set **nl** to C's left child
  - Set **l** to **nl**'s right child
  - Repeatedly reset **nl** to **l**, and **l** to **l**'s right child, 'till **l** does not have a right child (**l** is at 110)
  - Set C's node field to **l**'s node field
  - Set **nl**'s rc field to **l**'s lc field

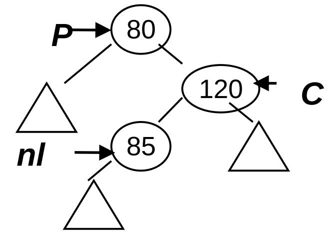


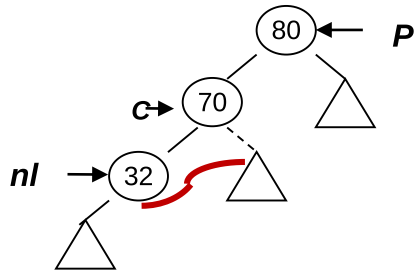


Graphical Representation of Cases 3c and 3d of the Binary Search Tree Delete Algorithm

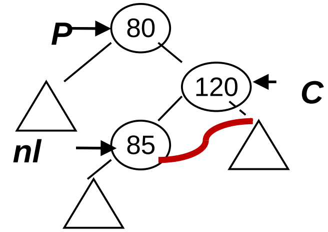


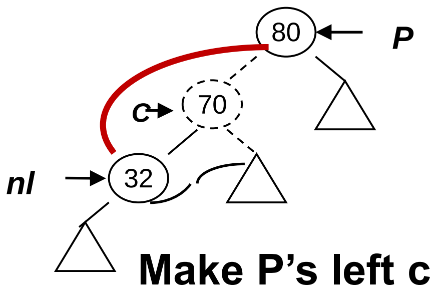
Position nl as shown





Make nl's right child C's right Child

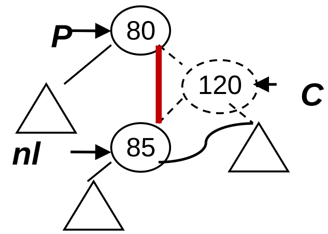




Make P's left child nl

Case 3, Part *C*

(The deleted node is a *left* child)



Make P's right child nl

Case 3, Part *D*

(The deleted node is a right child)

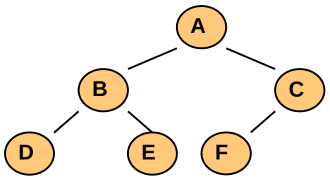
Binary Search Tree Delete Algorithm Case 3

```
found =  findNode(root, targetKey, P, C);
if(found == false) return false; // node not found
if(C.lc != null  &&  C.rc != null) // Case 3
{   nextLargest = C.lc;
    largest = nextLargest.rc;
    if(largest != null) // Cases 3a-b
    {  while(largest.rc != null)  // move nl and l
       {  nextLargest = largest;
          largest = largest.rc;
       } // end while loop, replacement node located
       C.node = largest.node; // "relocate" it
       nextLargest.rc = largest.lc; // save left subtree
    } // end of right subtree exists case
else  // Cases 3c-d
    {  nextLargest.rc = C.rc; // save right subtree
       if(P.lc == C)  // deleted node is a left child
          P.lc = nextLargest; // jump around  it
       else           // deleted node is a right child
          P.rc = nextLargest; // jump around  it
    } //end of no right subtree case
    return true;
} // end of Case 3
```

Traversing Binary Search Trees

- More complicated than traversing linear structures (arrays and singly linked lists)
- Common traversals fall into two groups
  - Breath first traversals (BFT)
    - Visit all nodes a level i (siblings), before proceeding to level i+1
  - Depth first traversals (DFT)
    - Visit all descendants of a node before visiting its siblings
    - Most popular DFTs have been assigned names
- Usually implemented recursively

Popular Depth First Traversals



- **NLR** traversal
  - **The root N ode would be visited first: A**
  - **All nodes in the root's L eft subtree would be visited next, in LNR order: B, D then E**
  - **All nodes in the root's R ight subtree would be visited next, in LNR order: C then F**

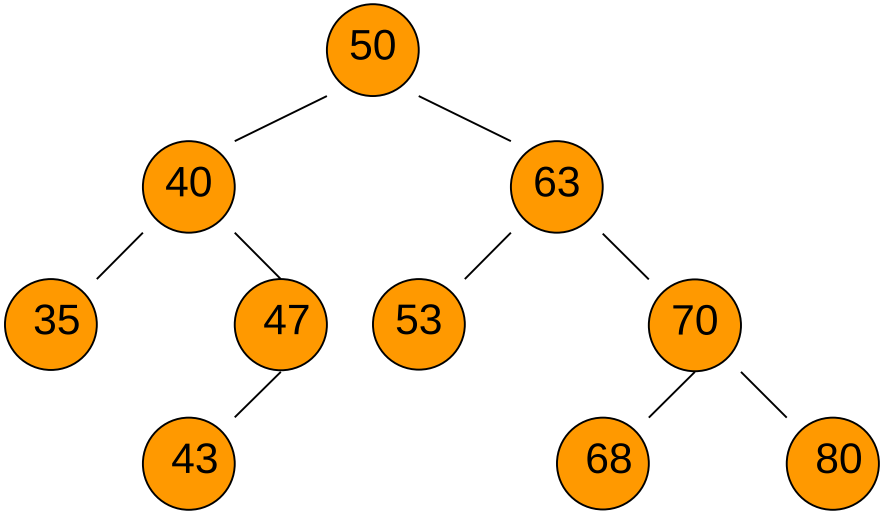
visit the root **N**ode, then traverse the **L**eft subtree, then traverse the **R**ight subtree.

- Other popular traversal orders:

- **NRL** visit the root **N**ode, then traverse the **R**ight subtree, then traverse the **L**eft subtree.
- **LNR** traverse the **L**eft subtree, then visit the root **N**ode, then traverse the **R**ight subtree.
- **RNL** traverse the **R**ight subtree, then traverse the root **N**ode, then visit the **L**eft subtree.
- **LRN** traverse the **L**eft subtree, then traverse the **R**ight subtree, then visit the root **N**ode.
- **RLN** traverse the **R**ight subtree, then traverse the **L**eft subtree, then visit the root **N**ode.

Examples

Examples of Depth First Traversals



LNR traversal: 35, 40, 43, 47, 50, 53, 63, 68, 70, 80

NLR traversal: 50, 40, 35, 47, 43, 63, 53, 70, 68, 80

RNL traversal: 80, 70, 68, 63, 53, 50, 47, 43, 40, 35

LNR and **RNL** traverse the nodes in ***sorted*** order

Recursive Implementation and Use of an LRN Output Traversal

Implementation

```
public void LNRoutputTraversal(TreeNode root)
{ if(root.lc != null)
     LNRoutputTraversal(root.lc);  // traverse the entire left subtree
  System.out.println(root.node);  // output the root node
  if(root.rc != null)
     LNRoutputTraversal(root.rc);  // traverse the entire right subtree
}
```

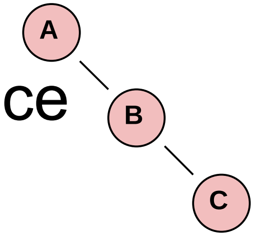
Use: Output all nodes in ascending order

```
public void showAll( )
{  if(root == null)  // check for an empty tree
      System.out.println("the structure is empty");
   else
      LNRoutputTraversal(root);
} // end of showAll method
```

Performance of a Binary Search Tree Speed of the Linked Implementation

- findNode
  - For a *balanced* binary tree, its loop executes at most $\log_2(n+1)$ times (once for each level of the tree), $O(\log_2(n+1))$
- Insert, Delete, and Fetch
  - Only loop in these algorithms is in the invocation of findNode, $O(\log_2(n+1))$
- Update
  - Combines a Delete and Insert Operation, $O(\log_2(n+1))$
- Average operation, $O(\log_2 n)$

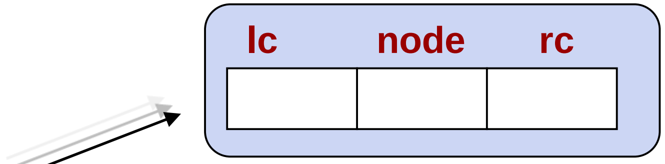
- When the tree is *not* balanced, worst case highly skewed, findNode's loop executes once for each node, $O(n)$
  - Average speed is then $O(n)$



Density

Density Of A Binary Search Tree Linked Implementation

- Density = information bytes / total bytes
  - Information bytes = n * w
    - n is the number of nodes, w is the bytes per node
  - Overhead = 4(1 + 3n) bytes (for ref. variables)
    - 4 bytes per variable
    - Root is 1 reference variable
    - n treeNodes with 3 references each

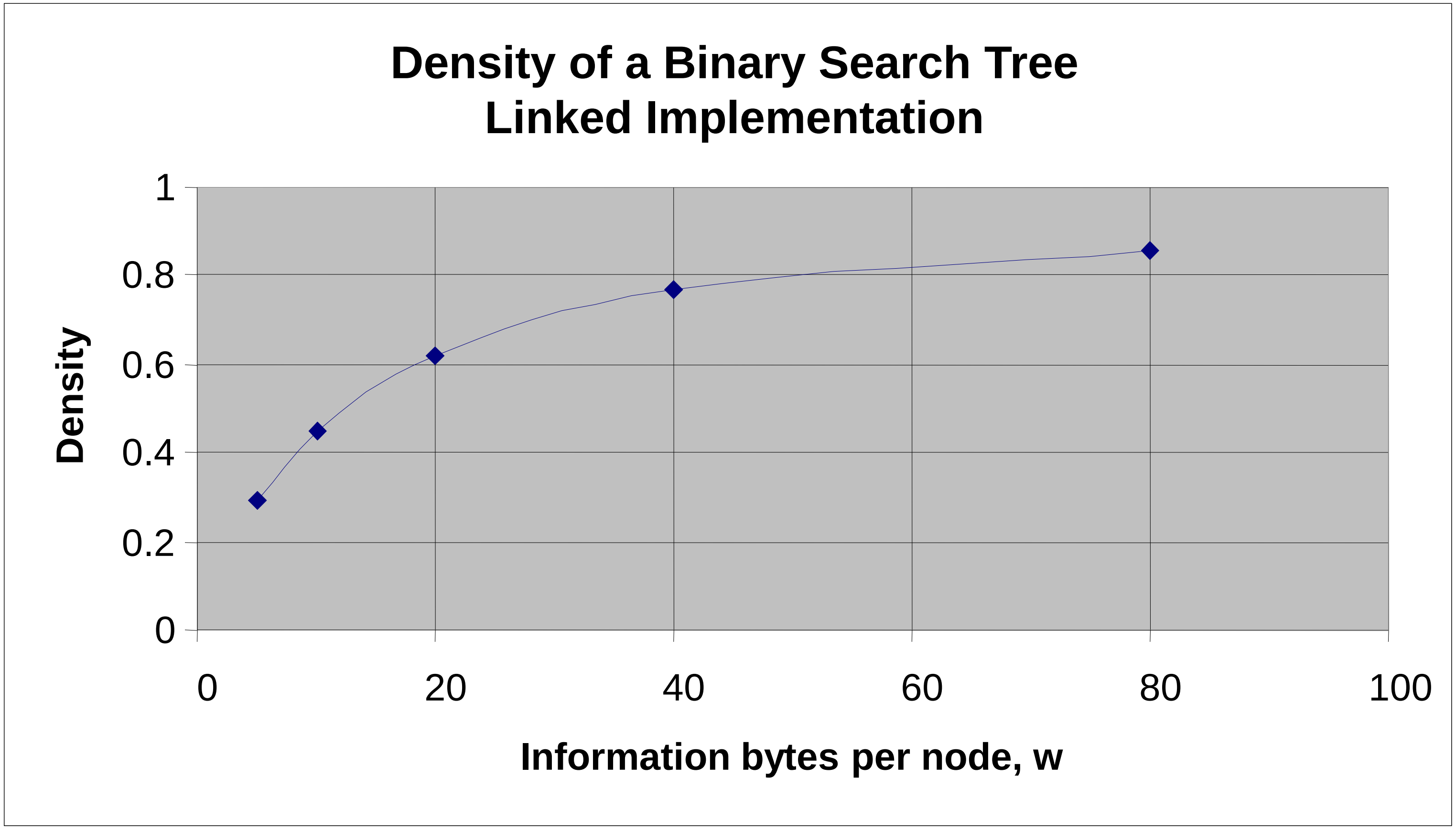


- Density = n * w / (n * w + 4(1 + 3n))

  $= 1 / (1 + 4 / (n*w) + 12 / w)$

  - As n gets large above approaches 1 / (1 + 12 / w)

Variation In Density Of A Linked Implementation of a Binary Search Tree



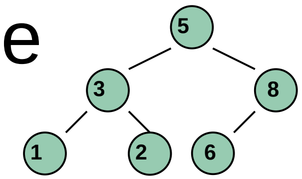
Performance Comparison

Data Structure	Operation Speed (in memory accesses)							Condition for Density > 0.8
	Insert	**Delete**	**Fetch**	**Update = Delete + Insert**	**Average[1]**	**Big-O Average**	**Average for n = 10^7**	
Unsorted-Optimized Array	3	$\leq$ n	$\leq$ n	$\leq$ n +3	(3n+6)/4 = 0.75n + 1.5	O(n)	0.75×10^7 + 1.5	w > 16
Stack and Queue	5	combined with fetch	4.5	not supported	9.5/2 = 5	O(1)	5	w > 16
Singly Linked List	6	1.5n	1.5n	1.5n +6	(4.5n+12)/4= 1.13n + 3	O(n)	1.13×10^7 + 3	w > 33
Direct Hashed (with pre-processing)	1 or (3)	2 or (4)	1 or (3)	3 or (7)	7/4 = 1.75 or (17/4 = 4.25)	O(1)	1.75 or (4.25)	w*l > 16
LQHashed	m+6	m + 10	m + 10	2m+16	(5m+42)/4	O(1)	1.25m + 11	w > 23
Balanced Search Tree	11 + 3 * log2(n+1)	9.3 + 3 * log2(n+1)	4 + 3 * log2(n+1)	20.3 + 6 log2(n+1)	11/2 + 4 * log2(n+1)	O(log2(n))	105	w > 48

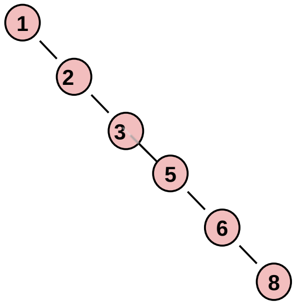
[1] Assumes all operations are equally probable and is therefore calculated as an arithmetic average of the four operation times.

Binary Search Tree Implementations that Keep the Tree Balanced

- When the tree is balanced the performance is O(log2n)



- When the tree is highly skewed the performance is O(n)



- Two implementations keep the tree (close to) balanced
  - AVL trees
  - Red-black trees
- Red-black trees have the best performance

AVL Trees

- Named after their inventors **A**delson-**V**elskii and **L**andis (1960's)
- The Insert and Delete algorithms are expanded to keep the height of the root's left and right subtree within 1 of each other
- The Fetch and Update algorithms are not altered

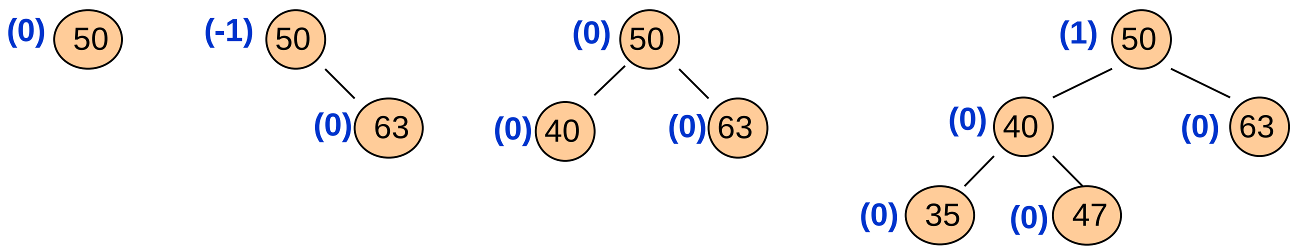
AVL's Expansion of the Fetch and Delete Algorithms

- Fetch and Delete operations
  - Adjust the balance factors of the tree's nodes to reflect the inserted or deleted node
  - Then they examine the tree to determine if a re-balancing is necessary
    - Start at the node's parent
    - Progress up the tree looking for a balance factor, bf, such that |bf| > 1 (not -1, 0, or +1)
  - If an |bf| > 1 is found, they perform rebalancing

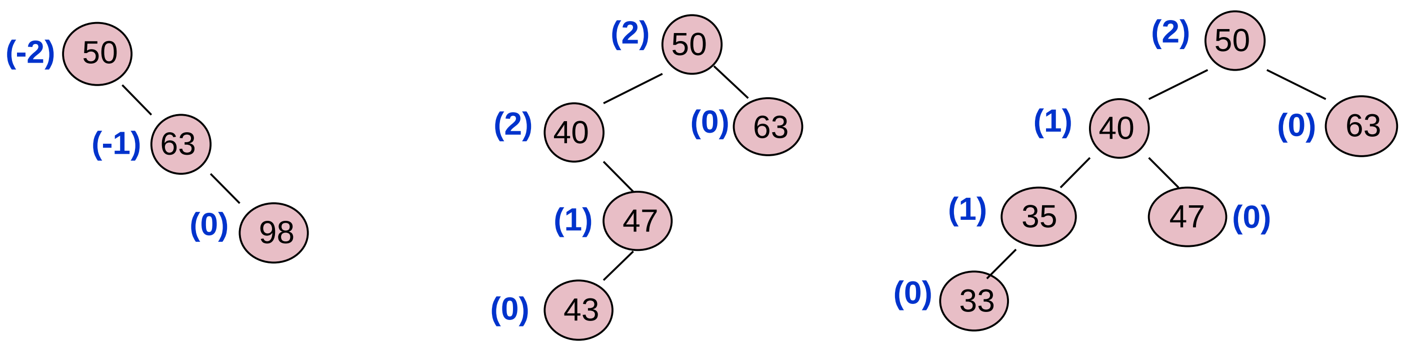
AVL Balance Factors

- Each node has a balance factor, bf

  (bf) = left subtree height – right subtree height



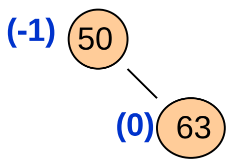
Balanced Binary Search Trees

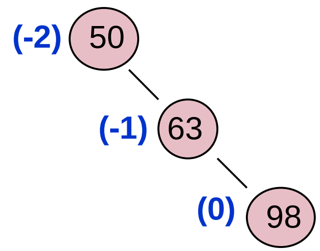


Imbalanced Binary Search Trees

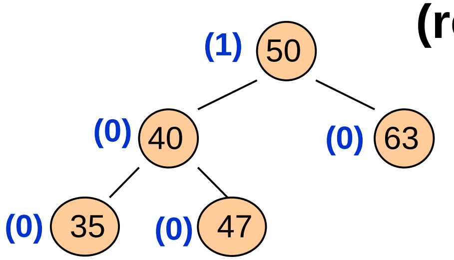
Changes Made to a Tree's Balance Factors During an Insert and Delete Operation

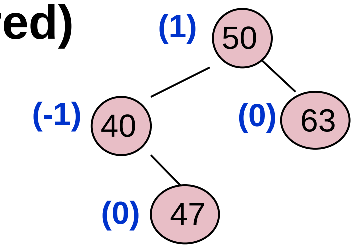
Insert of Node 98
(rebalance required)





Deletion of Node 35
(rebalance *not* required)





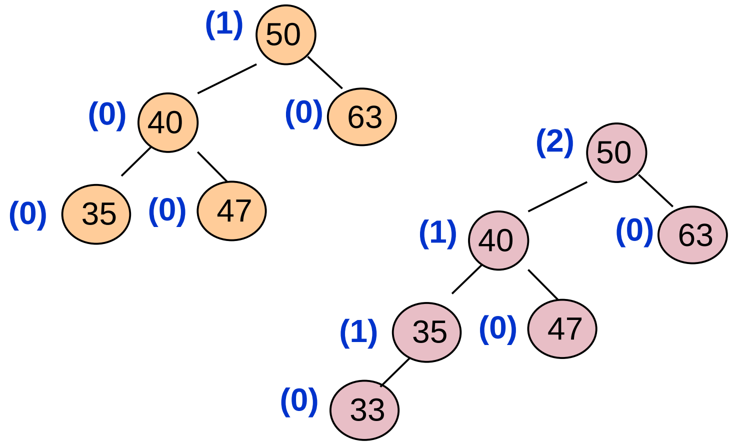
Rebalancing of AVL Trees

- Nodes are subjected to a series of “rotations” to rebalance the tree
- The rotations performed depend on the tree’s new configuration, four cases
  - The operation has produced a
    - Left high subtree in a left high tree (left of left)
    - Right high subtree in a right high tree (right of right)
    - Right high subtree in a left high tree (right of left)
    - Left high subtree in a right high tree (left of right)

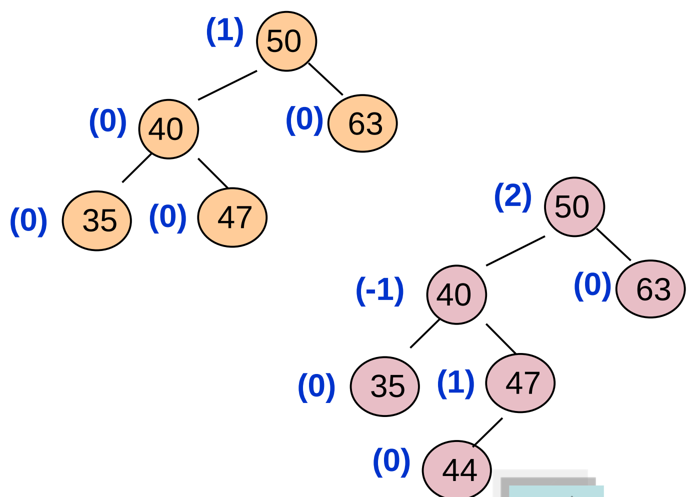
Examples

Examples of Trees Becoming Left of Left and Right of Right

- Insertion of 33 produces a
  - Left high subtree (rooted by 40)
  - In a left high tree (rooted by 50)
  - The tree is now ***left of left***
  - A right rotation is required

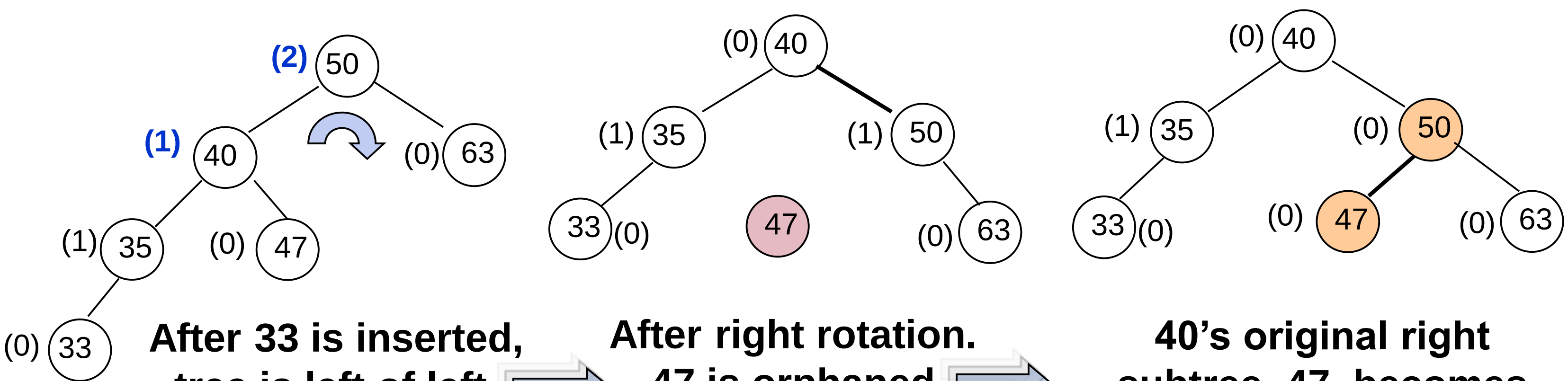


- Insertion of 44 produces a
  - Right high subtree (rooted by 40)
  - In a left high tree (rooted by 50)
  - The tree is now **right of left**
  - Two rotations are required



Rebalancing Rotations for Left of Left trees

- One rotation is required to rebalance the tree, *and* then an orphaned subtree has to be assigned a parent



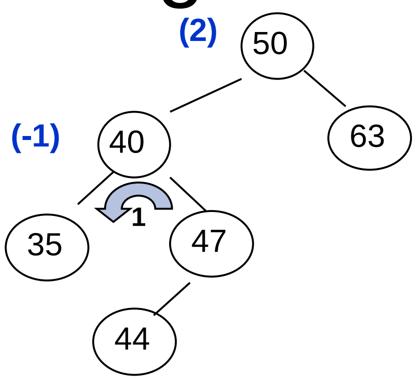
After 33 is inserted, tree is left of left. Prior to right rotation.

After right rotation. 47 is orphaned

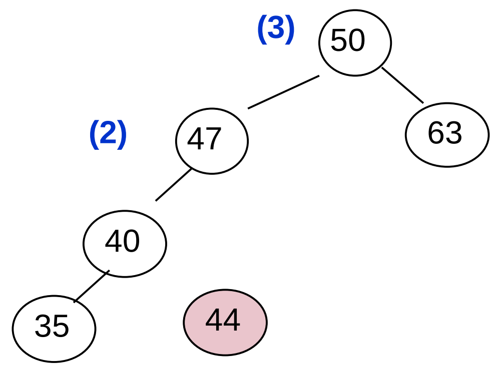
40's original right subtree, 47, becomes 50's left subtree

Rebalancing Rotations for Right of Left Trees

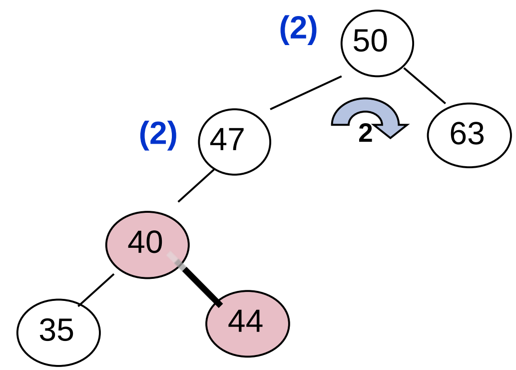
- Two rotation are required to rebalance the tree, *and* an orphaned subtree has to be assigned a parent



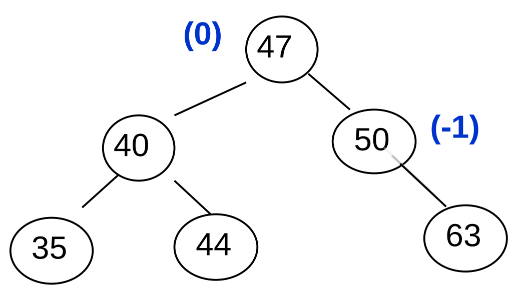
After 44 is inserted, tree is right of left. Prior to right rotation.



After left rotation. 44 is orphaned



After 47's original left subtree, 44, becomes 40's right subtree. Prior to right rotation.



After right rotation

Red-Black Trees

- Invented by Rudolf Bayer in 1972
- Shares many characteristics with AVL trees
  - Not perfectly balanced
  - Add balancing to Insert and Delete algorithms
  - Balances using rotations
- These trees comply to five conditions
- When conditions 3 or 4 are violated, rebalancing takes place

Five Conditions of a Red-Black Tree

1. **Every node** in the tree must be **red or black**.
2. The **root** of the tree is **always black**.
3. A **red node's children** must be **black**.
4. **Every path** from a node **to a null** link (a leaf's left or right **null** reference) must contain the **same number of black nodes**.
5. The **tree** must be **a binary search tree**.

Rebalancing a Red-Black tree

- Rotations are more complicated than AVL rotations because they involve color changes
- However the algorithm is faster than AVL because the color changes and rotations are made during the downward traversal to find
  - The node to be deleted
  - The insertion point
- AVL requires a second traversal up the tree to perform its balancing

Implementation Of A Singly Linked List

- Implemented as a class with
  - One private data member, `h`
  - An inner class, `Node`, defines the linked nodes
    - Class Node has two data members, `l` and `next`
- Linked List class methods
  - A constructor that implements the initialization pseudocode
  - `insert`, `delete`, `fetch` that implement the operation algorithm pseudocode (and `update`)
  - `showAll` to output all nodes
    - Traverses the list
    - Invokes the node definition class' `toString` method

Array-Based Representation Of Binary Trees

- An alternative to the linked (non-contiguous, dynamic) representation
  - Client node references are stored in an array
- Relies on a rule to locate a node's children
- Compare to the linked representation
  - Its Inset and Fetch algorithms are simpler
  - For a *limited set* of trees applications (heap sort,…) its
    - Speed is also $O(\log_2 n)$
    - Density better then the linked approach when the tree is balanced
  - However the Delete operation offers some problems

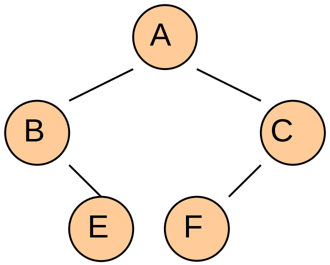
2i + 1, 2i + 2 Rule

- The *root node* reference is stored at index, i = 0
- The "rule" is used to locate a node' s children

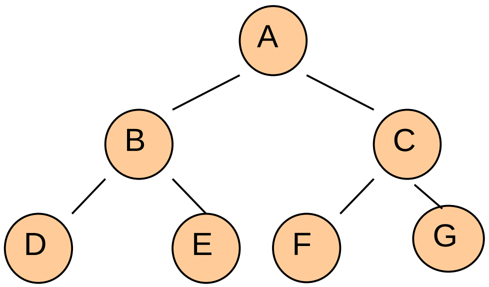
> For a node whose reference is stored at index i of the array
> - its *left* child's reference will be stored at index 2i + 1,
> - its *right* child's reference will be stored at index 2i + 2.

- Using the rule
  - Any binary tree can be stored in an array
  - Any array can be "viewed" as a tree

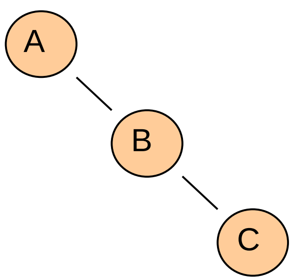
Examples of Binary Trees Stored in Arrays



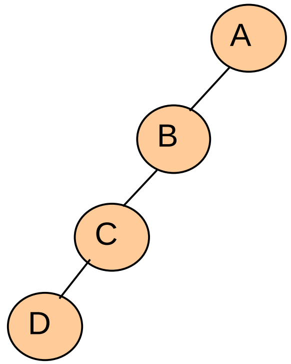
data[0]	A's address
data [1]	B's address
data [2]	C's address
data [3]	**null**
data [4]	E's address
data [5]	F's address
data [6]	**null**
data [7]	**null**
	:
data [n-1]	**null**



data [0]	A's address
data [1]	B's address
data [2]	C's address
data [3]	D's address
data [4]	E's address
data [5]	F's address
data [6]	G's address
data [7]	**null**
	:
data [n-1]	**null**

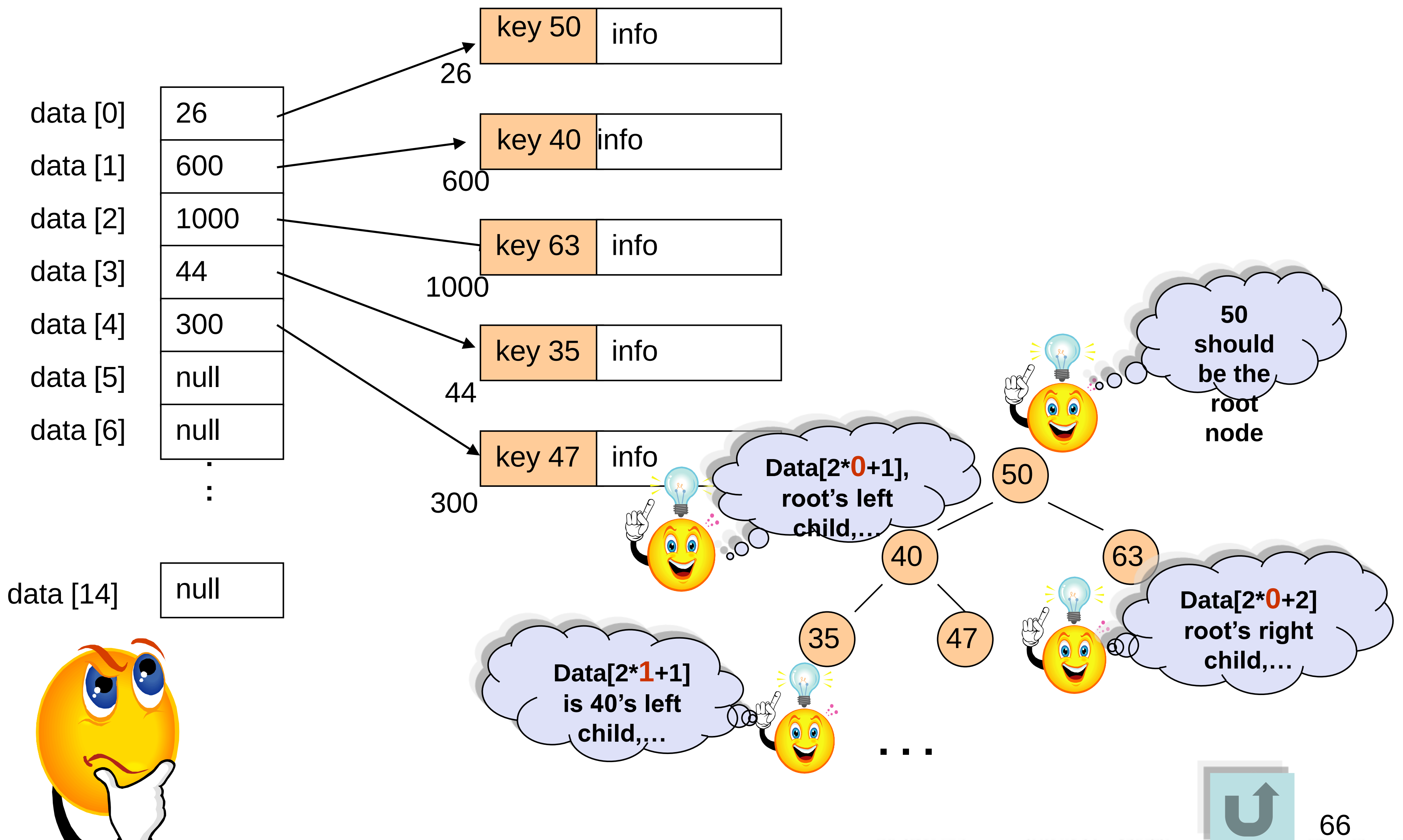


data [0]	A's address
data [1]	**null**
data [2]	B's address
data [3]	**null**
data [4]	**null**
data [5]	**null**
data [6]	C's address
data [7]	**null**
	:
node[n-1]	**null**



data [0]	A's address
data [1]	B's address
data [2]	**null**
data 3]	C's address
data [4]	**null**
data [5]	**null**
data [6]	**null**
data [7]	D's address
	:
data [n-1]	**null**

Viewing an Array as a Binary Tree



Array-Based Binary Search Tree Insert Algorithm

Same as the linked based algorithm except the 2i+1, 2i+2 rule is used to locate the insertion point

```
i = 0;
while(i < size && data[i] != null) // continue search for insertion point
{  if(data[i].key > newListing.key) // go into left subtree
        i = 2 * i + 1;
   else  // go into the right subtree
        i = 2 * i + 2;
} // end search
if (i >= size) // node position exceed the bounds of the array
   return false;
else // insert the node
{  data[i] = newListing.deepCopy();
   return true;
}
```

Fetch

Array-Based Binary Search Tree Fetch Algorithm

Same as the linked based algorithm except the 2i+1, 2i+2 rule is used to locate the insertion point

```
i = 0;
while(i < size && data[i] != null && data[i].key != targetKey) // search
{  if(data[i].key) > targetKey) // go into left subtree
       i = 2 * i + 1;
   else  // go into the right subtree
       i = 2 * i + 2;
} // end search
if (i >= size || data[i] == null) // node not found
    return null;
else // return the node
    return data[i].deepCopy( );
```

Speed of the Array-Based Binary Search Tree

- In both the Insert and Delete operations
  - The loop, to find the insertion point or the node to be fetched, performs 2 memory accesses. For a
    - **Balanced** tree executes an average of $\leq \log_2(n+1)$ times, $O(\log_2 n)$
    - **Highly skewed** tree executes an average of $\leq n$ times, $O(n)$
- Therefore, $O(\log_2 n) \leq$ speed $\leq O(n)$

Density of the Array-Based Binary Search Tree

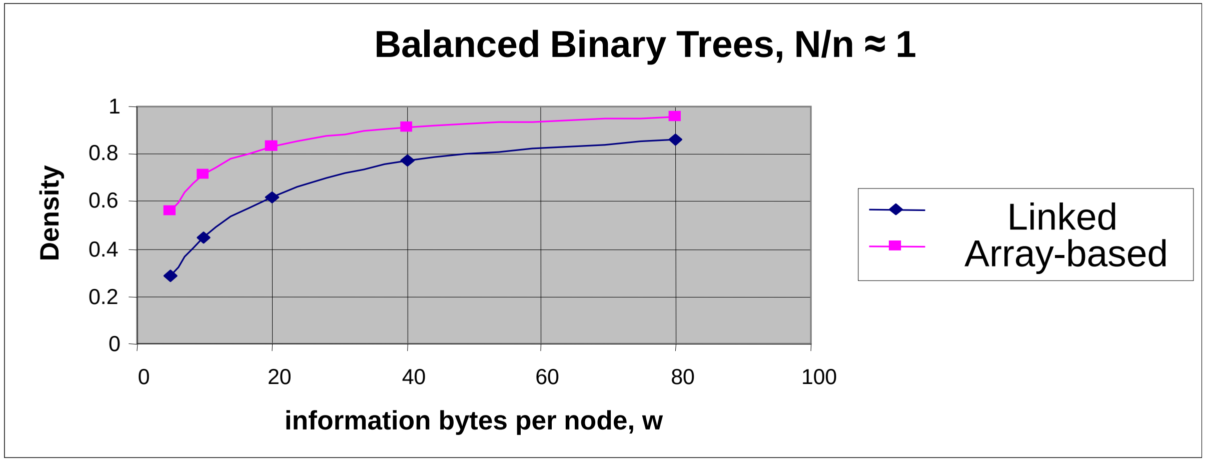
- Density = information bytes / total bytes
  - Information bytes = n * w
    - n is the number of nodes, w is the bytes per node
  - Overhead = 4N + 4; N being the size of the array
    - 4 bytes for each array element + 4 bytes for `size`
- Density = n * w / (n * w + 4N + 4)

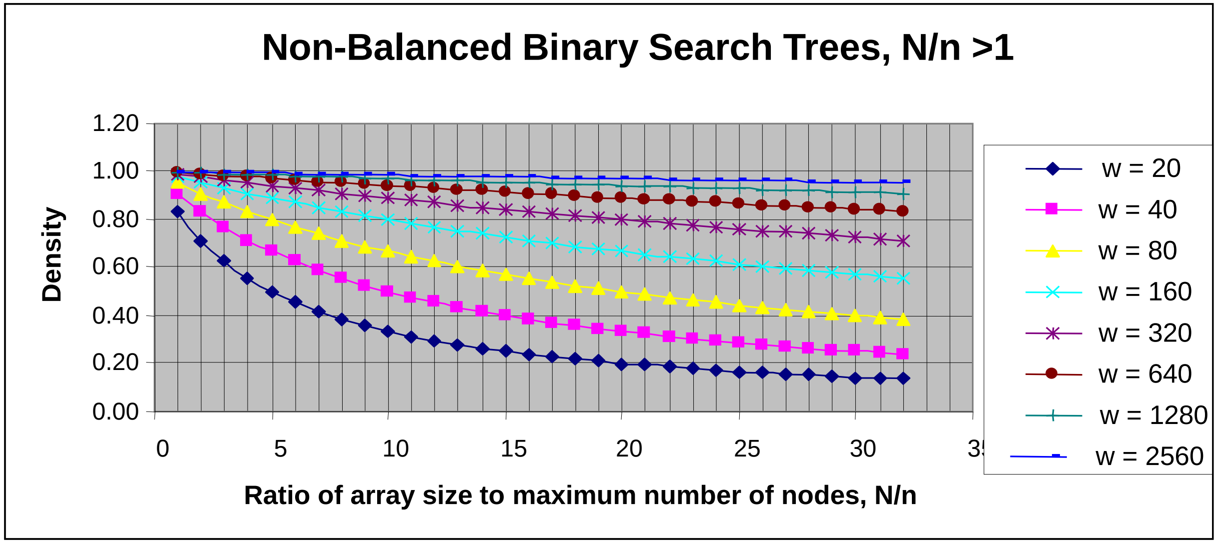
$$= 1 / (\, 1 + 4N / (n^* w) + \underbrace{4 / (n * w)}_{\approx\, 0 \text{ for large } n} \,)$$

$$\approx 1 / (1 + \underbrace{N / n}_{1 \text{ for a balanced tree } (n \approx N)} * 4 / w))$$

$$\approx 1 / (1 + (4 / w))$$

Variation in Density of the Array-Based Binary Tree Representation



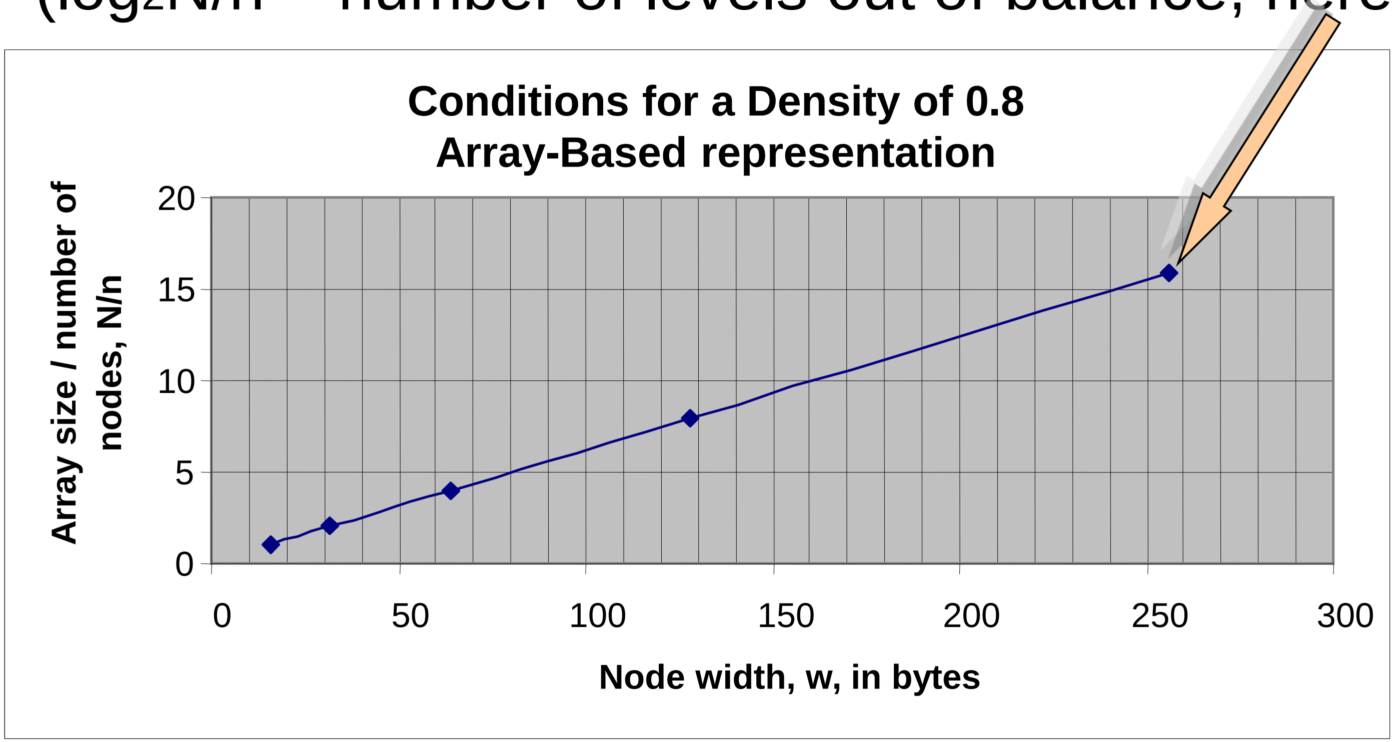


Condition for A Density of 0.8
Array-Based Binary Tree Representation

Density = d ≈ 1 / (1 + N / n * 4 / w))

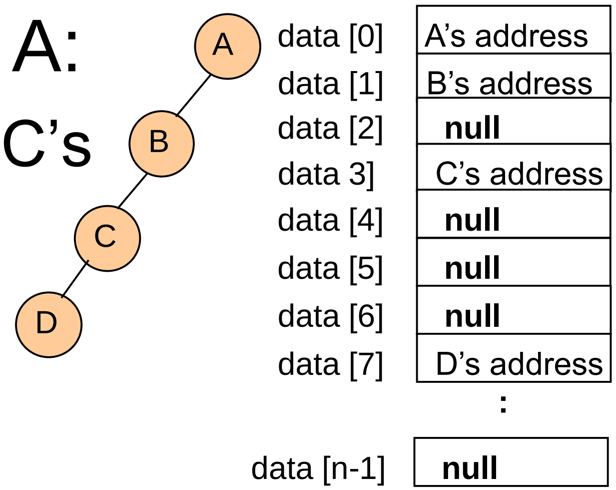
Substituting d = 0.8 and solving for N/n we obtain N/n = 0.625w

($\log_2 N/n$ = number of levels out of balance, here 4)



Array-Base Binary Tree
Delete Operation Problems

- Very slow except for a leaf node deletion
  - E. g., to delete the root node, A:
    - Copy B's address into data [0], C's into data[1], D's into data[3], …



- A remedy would be to mark a node deleted, and ignore it in the Delete and Fetch algorithm search
  - The down side of this is that there is no garbage collection

Java's TreeMap Class

- The API TreeMap class
  - Is in the package java.util
  - Is an implementation of Red-black tree
  - Is accessed in the key field mode
    - The key's class must implement the interface Comparable: **int** compareTo(KeyObjectType aKey)
      - Strings and numeric wrappers implement this interface
  - Is an unencapsulated generic structure
  - Its methods include Insert (set), Fetch (get), and Delete (remove)

Coding Example

Use of the TreeMap Class

```
public static void main(String[] args)
{  TreeMap <String, Listing> dataBase = new TreeMap<String, Listing>();
   Listing b, t;
   Listing bill = new Listing("Bill", "1st Avenue", "999 9999");
   Listing tom = new Listing("Tom", "2nd Avenue", "456 8978");
   Listing newBill = new Listing((“William”, “99th Street”, “123 4567”);
  // inserts
   dataBase.put("Bill",bill);
   dataBase.put("Tom",tom);
  // fetches
    b = dataBase.get("Bill");    t = dataBase.get("Tom");
    System.out.println(b, + “\n” t);
  // effectively an update of Bill's address
    dataBase.put("Bill", newBill);      b = dataBase.get("Bill");  // fetches
    System.out.println(b);
  // demonstration of the lack of encapsulation. Client can change node contents
    newBill.setAddress(“18 Park Avenue”);
    b = dataBase.get("”Bill”);
    System.out.println(b);
  // delete operation
   dataBase.remove("Bill");   b = dataBase.get("Bill");
   System.out.println(b)
}
```

Relative Merits of the Fundamental Structures

Data Structure	Operation Speed: Insert	Delete	Fetch	Update	Big-O Average [1]	Average for n = 10^7	Condition for Density > 0.8	Non-Contiguous memory	Inherently Dynamic
Unsorted-Optimized Array					O(n)	10^7	w > 16	No	
Stack and Queue		not allowed		not allowed	O(1)	5	w > 16	Yes	
Singly Linked List					O(n)	10^7	w > 33	Yes	Yes
Direct Hashed					O(1)	1.75 or (4.25)	w*l > 16	No	
LQHashed					O(1)	1.25m + 11	w > 23	No	
Balanced Search Tree					O(log2n)	105	w > 48	Yes	Yes
Array-based Search Tree					O(log2n)	93	w > 16	No	

best → worst

[1] Assumes all operations are equally probable and is therefore calculated as an arithmetic average of the four operation times.

The End

Return to, Overview, Trees, Binary Trees, Binary Search Trees, Array-based trees, Java's TreeMap class, Relative merits of classic structures

End Presentation