Trees: Data Structure #4

Data Structure Comparisons: Efficiency

Operation / Data Structure	Add	Remove	Get	Report: GetNext GetPrev
Sparse Array	**Instantaneous.** Assign to index slot	**Instantaneous.** Remove from index slot.	**Instantaneous.** Assign to index slot.	**Slow:** Walk thru all index spaces
Orderable Array	**Slow.** Insert → pushback	**Slow:** Remove → move down	**Fast.** Binary search to item	**Fast.** Binary Search to current item
Linked List	**Slow:** "walk" ½ entries	**Slow:** "walk" ½ entries	**Slow:** "walk" ½ entries	**Instantaneous** Move to adjacent entry
Hash Table	**Very Fast.** Hash key, assign to Bucket. Append	**Very Fast.** Hash Key, find Bucket, Delete	**Very Fast.** Hash Key, find Bucket. Get.	**Unsupported.** No "ordering"

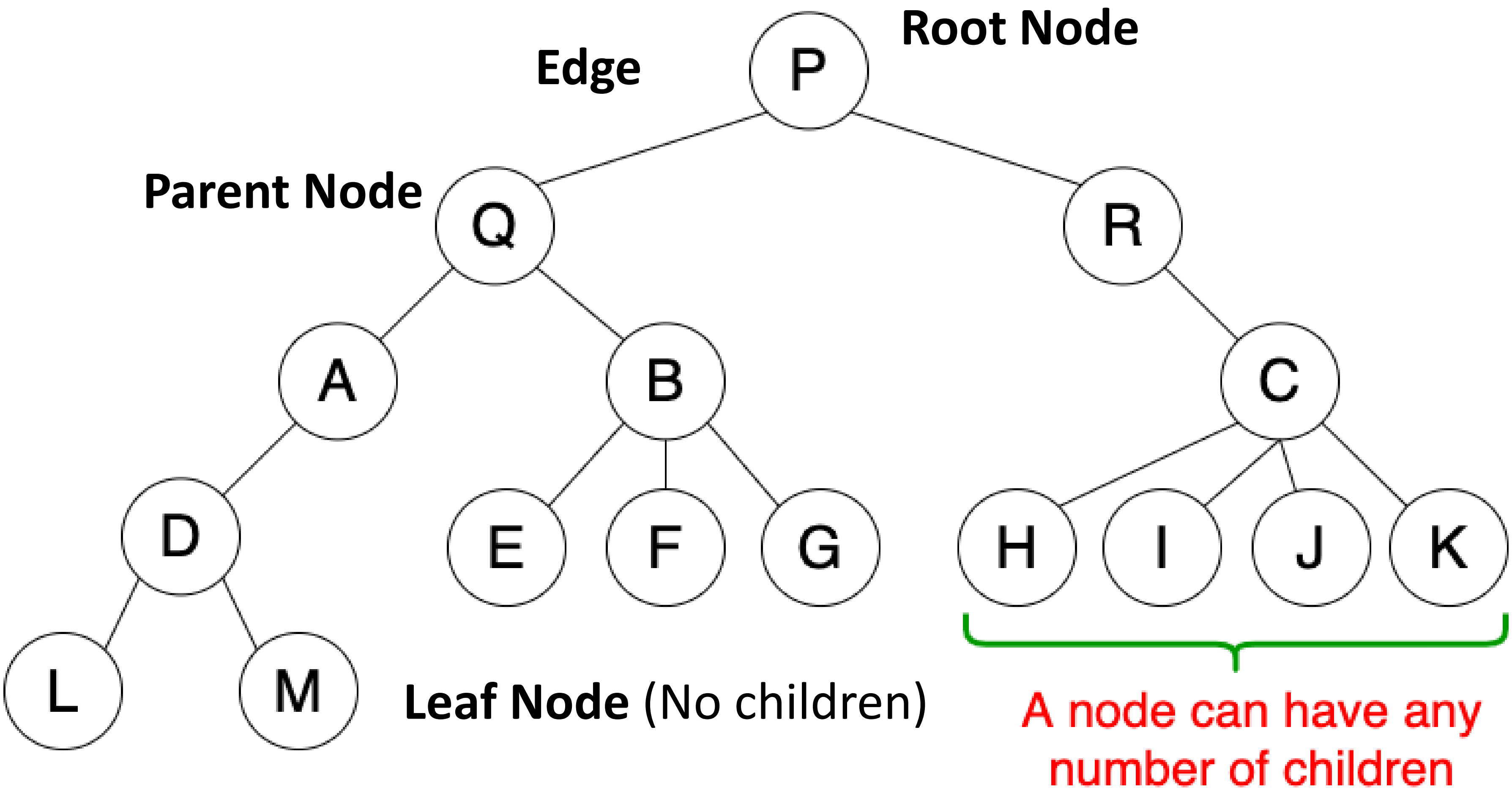
Data Abstraction Comparisons: Operations

Data Structures (Internal Classes)	Data Abstraction	External Interface
Linked List + Node Structure + Iterator or Ordered Array	Deque	Linked List Iterator (next/previous)
Ordered Array or Deque	LIFO FIFO	Stack (push / pop) Queue (push / pop)
Hash Table Ordered Array Buckets	Associative Array	Store element with key Get element by key Remove element by key

Data Abstraction Comparisons: API

Data Abstraction	Underlying Data Structure	Add / Delete	Ordering
Deque	Ordered Array or LinkedList	Insert / Remove	By time of insert (front and back) getNext/getPrev
Stack and Queue	Ordered Array or Deque	Push / Pop	By time of insert
Associative Array	Sparse Array Hash Table	Insert / Remove	N/A

Data Abstraction: General Data Tree: Unordered



Node Depth: # edges until Root Node
<Nodes A, B, C all at Depth Level 2>
Tree Height: Largest depth of any node

What is something this data structure might most effectively represent?

A Directory / Folder Hierarchy
Posix: Directory Walking & Recursion

Hierarchy cpp file Printer – prints location of all C++ code files in Directory Hierarchy

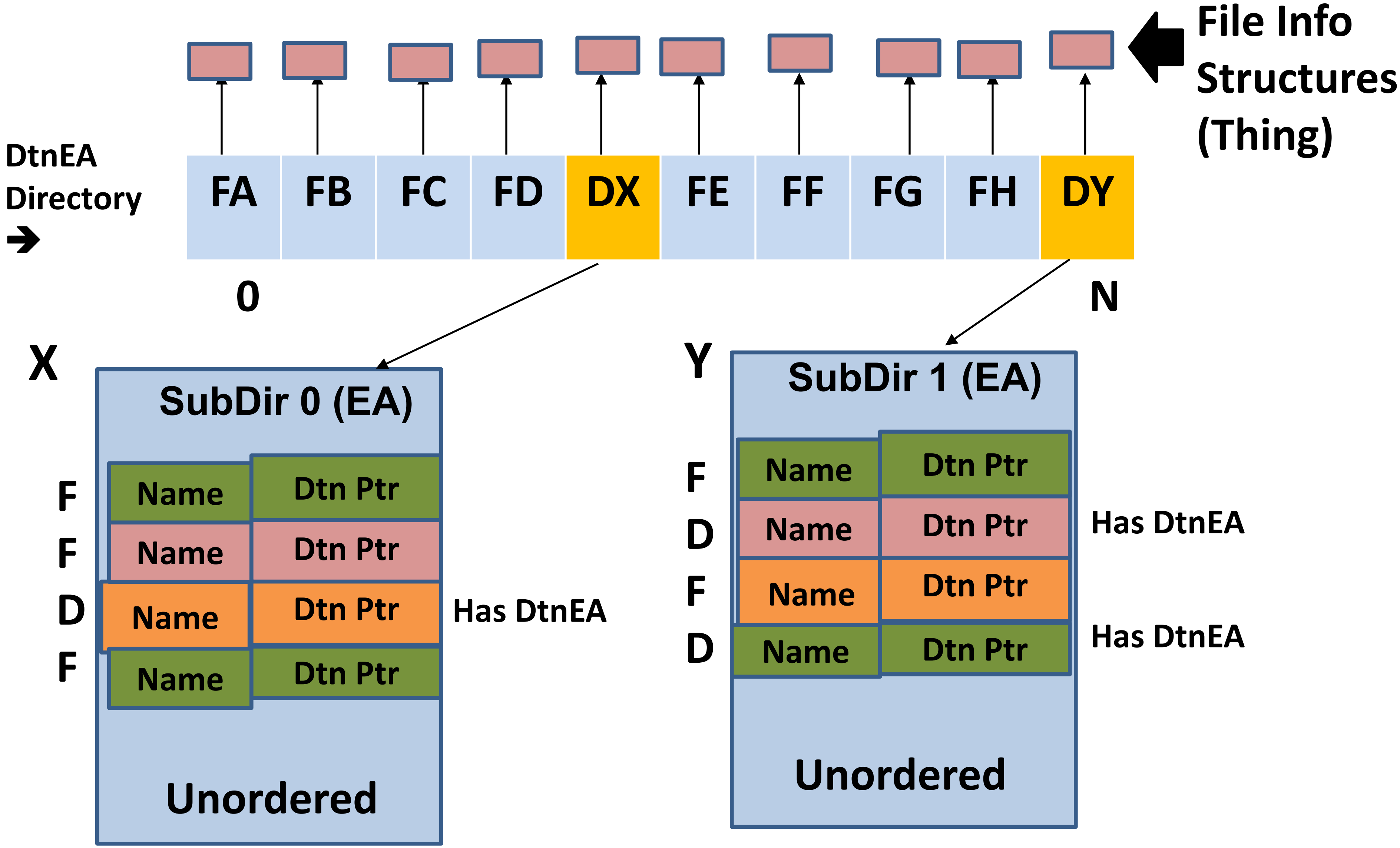
```
void fhp (DirectoryPath)
{   // Pseudocode (obviously)
   Open provided Directory  // opendir (returns an array of entries)
   While (more entries)   //  Ignore everything not a cpp file or subdirectory
       Read next entry    // readdir (returns an entry)
       if   (entryType == “cpp” file) Print (“DirectoryPath/entryname”)
       if   (entryType == subdirectory)   // Recursively invoke self
            fhp (“DirectoryPath/entryName”) // Print all cpp files in subdirectory hierarchy
   }
       // All cpp files in or under the provided Directory have been printed
};
```

Directory Tree "Node"

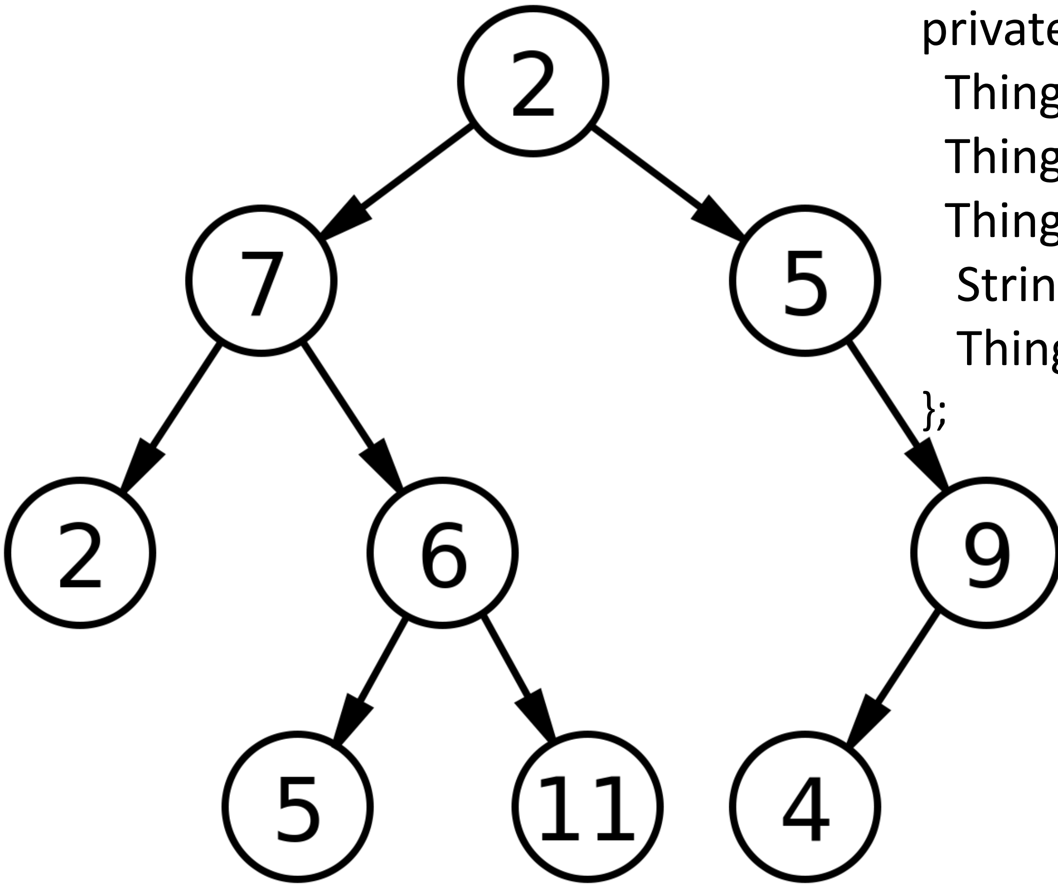
```
struct  Dtn {
    Dtn * parent;  // Owning Directory Ptr (or NULL if Root Directory)
    string name;  // ID in "parent" (or empty if Root)

    FileInfo *finfo; //  Struct with logical block assignments, RW owners …
                        // (i.e.  Ptr to the "Thing")

    DtnEA *contents;
         // If Directory:
              //Ptr to (non-ordered) EA of Dtn's
                   // Subdirectory Dtn's
                   // File Dtn's
         // If File:
                   // NULL
};
```

➔ **All Files and empty Directories are Leaf Nodes**

Directory Tree Hierarchy



Binary Trees: Nodes have 0,1,2 Children




```
class ThingNode {
private:
  ThingNode *par; // Ptr to Parent or 0 (root)
  ThingNode*lc;  // Ptr to Left Child or 0
  ThingNode *rc; // Ptr to Right Child or 0
   String id;  // Comparator (the #'s here)
   Thing *thing; // Ptr to Thing represented
};
```

Up to 2 Successors (and for all but the Root Node) a Parent

Unsorted Tree: How can you find the node with Identity you wanted?

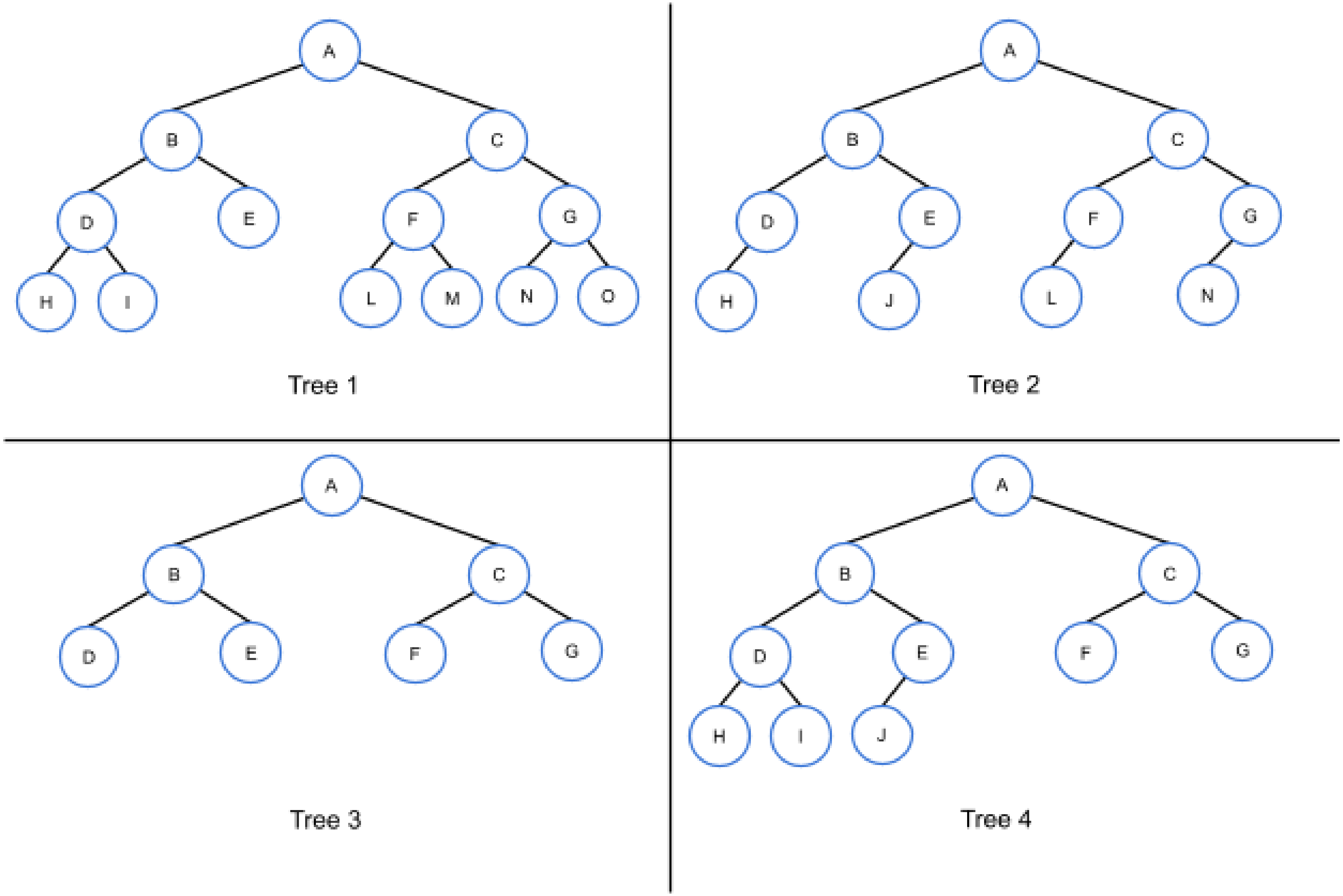
Ex: What path do you go down searching for the Node with ID 4??

Binary Tree Terminology

- **Node Depth:** # edges until Root Node
- **Tree Height:** Largest depth of any node
- **Root Node**: Topmost Node, if none, Tree is empty
- **Leaf Node**: Possessing neither a right or left "child"

- **Full Tree:** Every Node has 0 or 2 children
- **Complete Tree:** All levels except last contain all possible children, All Leaf Nodes must be on LHC
- **Perfect Tree:** Each non-Leaf node has 2 children, all Leaf nodes at same depth

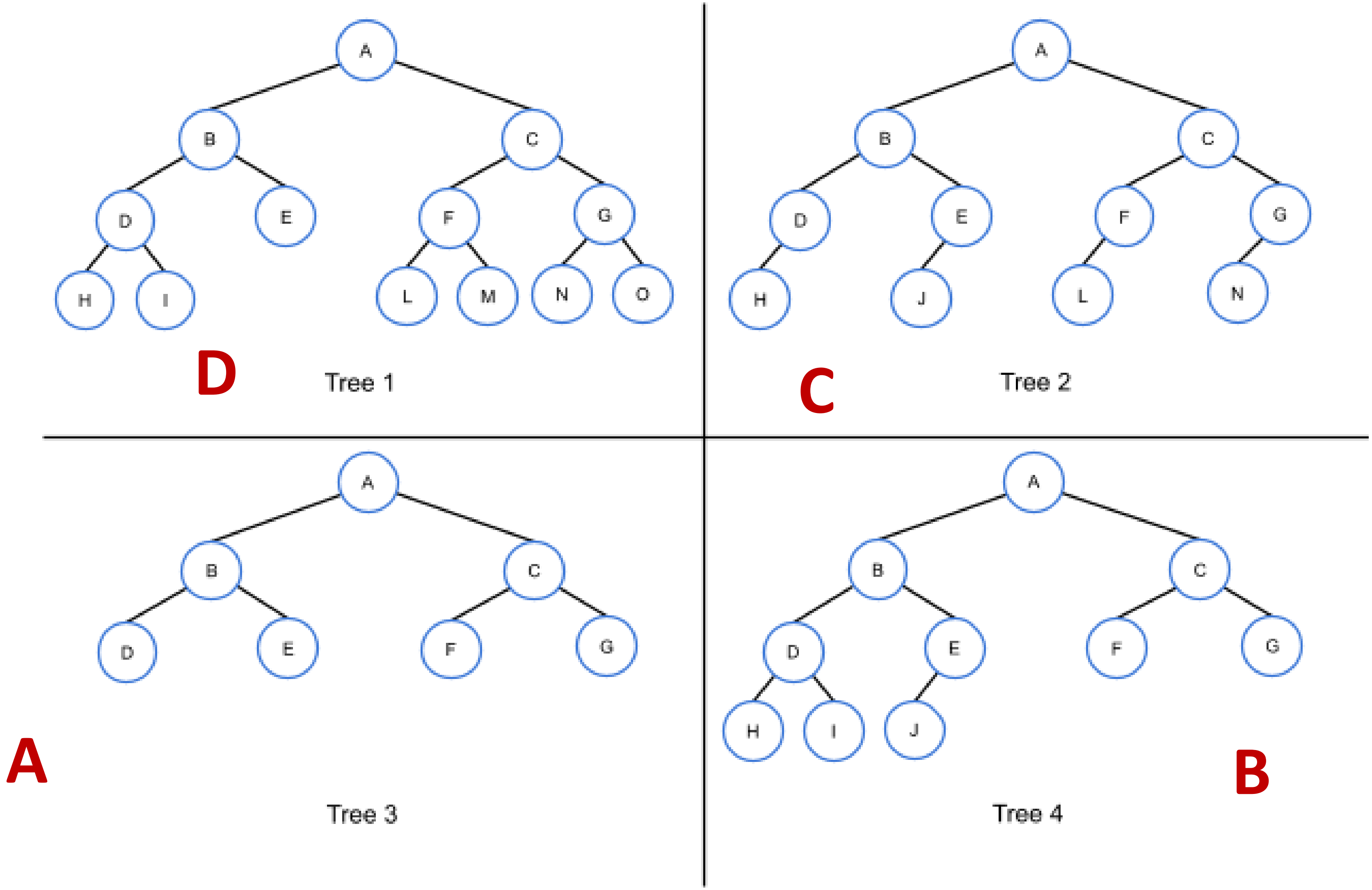
Tree Example



A. Full Complete Perfect / B. Not Full, Not Complete Not Perfect

C. Not Full Complete Not Perfect / D. Full Not Complete Not Perfect

Tree Example



A. Full Complete Perfect / B. Not Full, Not Complete Not Perfect
C. Not Full Complete Not Perfect / D. Full Not Complete Not Perfect

Binary Search Tree (BST): (Ordered)

Placing a new Node:

If the current node has a Left-Handed Child (LHC)

The ID of the LHC will be <= the ID of the current Node

If the current node has a Right-Handed Child (RHC)

The ID of the RHC will be > the ID of the current Node

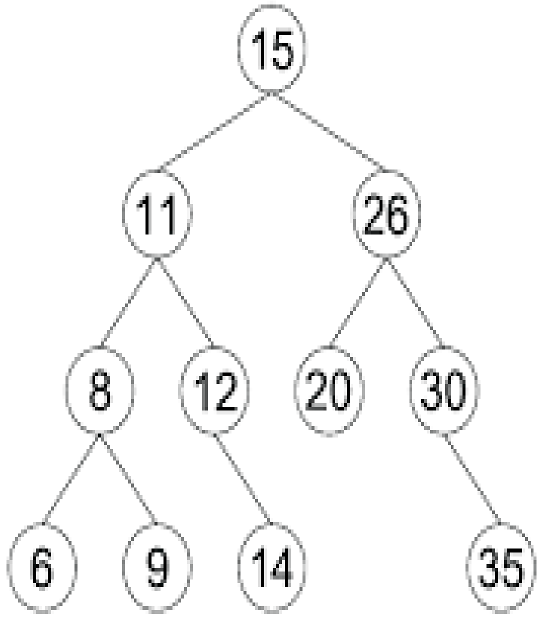
Implies:

No node under LHC will be > Parent

No node under RHC will be < Parent

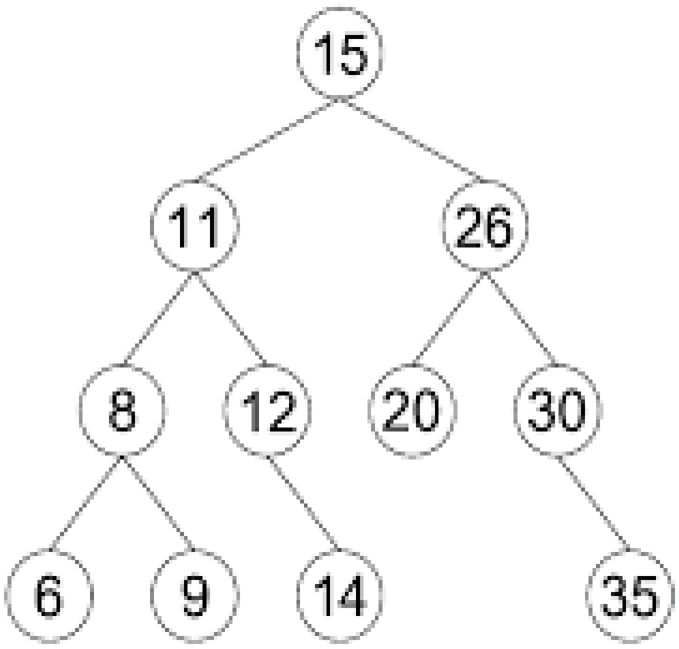
Q. Where could Node ID=9 have been placed?
(RHC of 8, LHC of 12, LHC of 20, LHC of 30)

Q. What path do you go down searching for Node ID=9??



Equivalent Binary Search Trees:

“Seeded by ID’s in order:
15,26,30,11,20,12,35,14,8,9,6
Max Node Depth = 3



Many different Binary Search Trees can be “seeded” by supplying identical elements in different order:

26,35,12,15,14,9,11,30,6,20,8 **Max Node Depth = ??**

(But things could be a lot worse …)

EXAMPLE OF A POORLY BALANCED TREE

A <u>VERY</u> different looking Binary Search Tree will result if the IDs of the elements being seeded are <u>ordered</u>.

Ex: 10,20,30,40,50,60,70

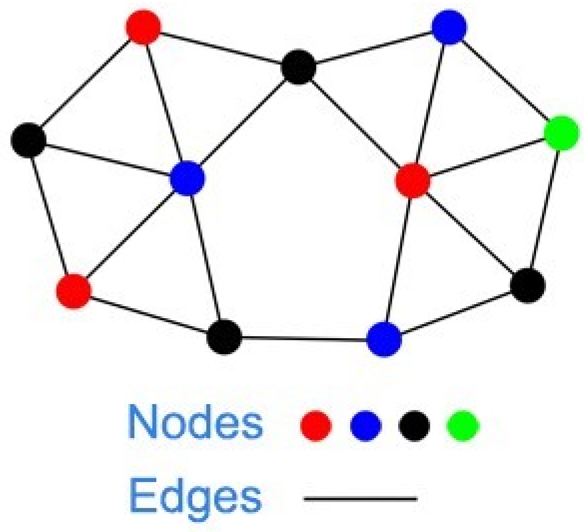
Max Node Depth = ??

The BST becomes a "Unichild" Linked List!!!

➔ **"Randomize" the insertion order or hash the Element IDs before insertion**

A brief review of “Nodes”

- **Accessible through index // Externally Ordered**
  - Sparse Array, Orderable Array
  - Pointer to object is enough
- Hash Table (no nodes)
- Linked List



Orderable Array

Orderable Array

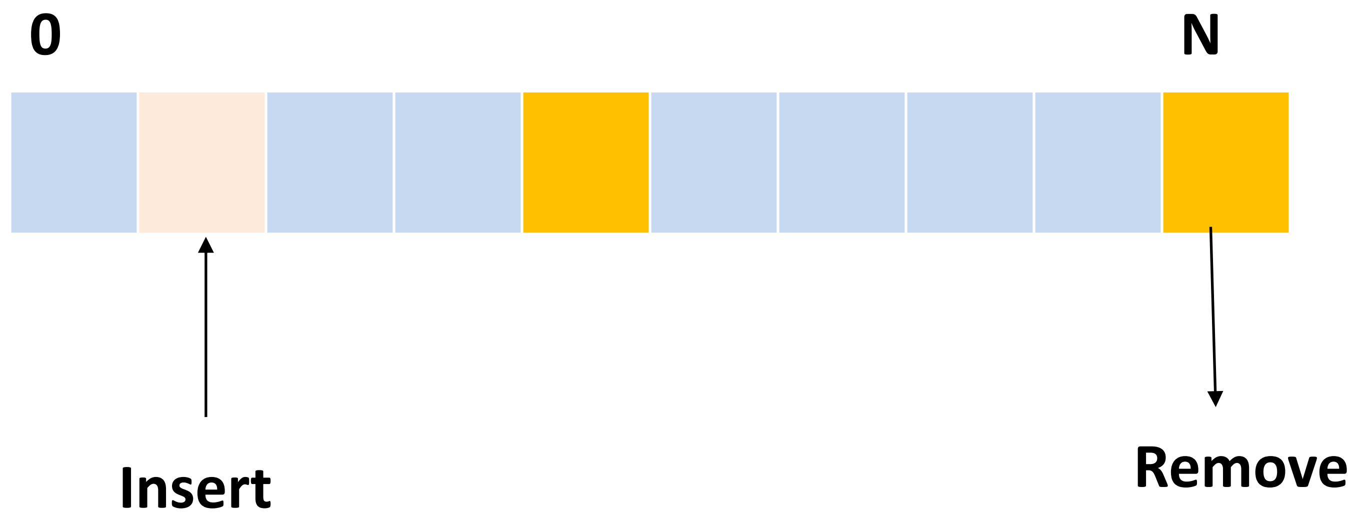
Ordering maintained by client (externally maps key → Location Index)

➔Object Key unknown to collection class

➔ "Neighbors" do not have to be linked to

➔ **No Connecting Nodes!!**

Sparse Array (empty spaces okay)



Sparse Array

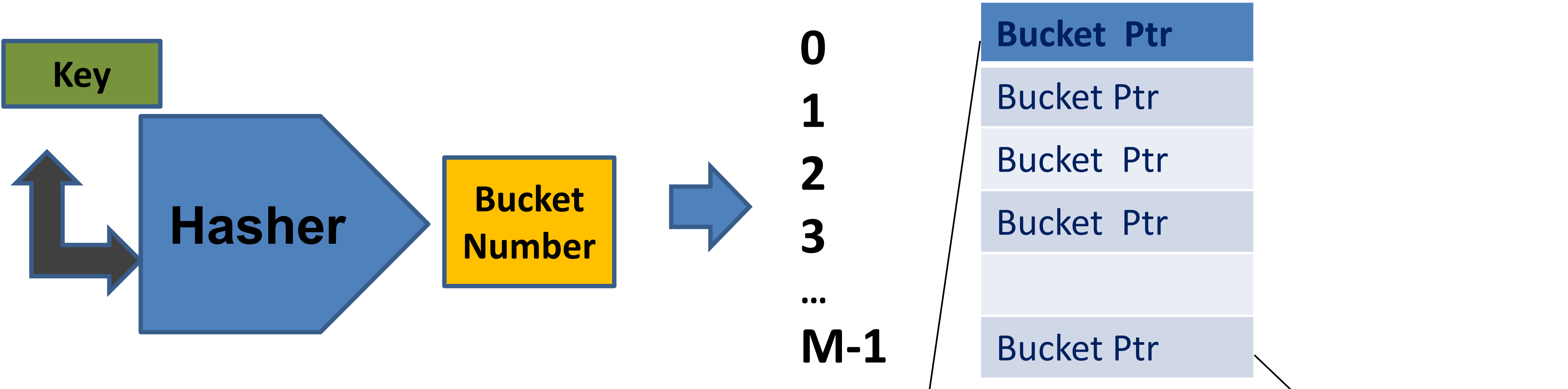
Ordering determined by client (uses key directly as Location Index)

➔Object Key unknown to collection class

➔ "Neighbors" do not have to be linked to

➔ **No Connecting Nodes!!**

Hash Table



Key hashed ➔ Bucket index
Collection class “knows” Key

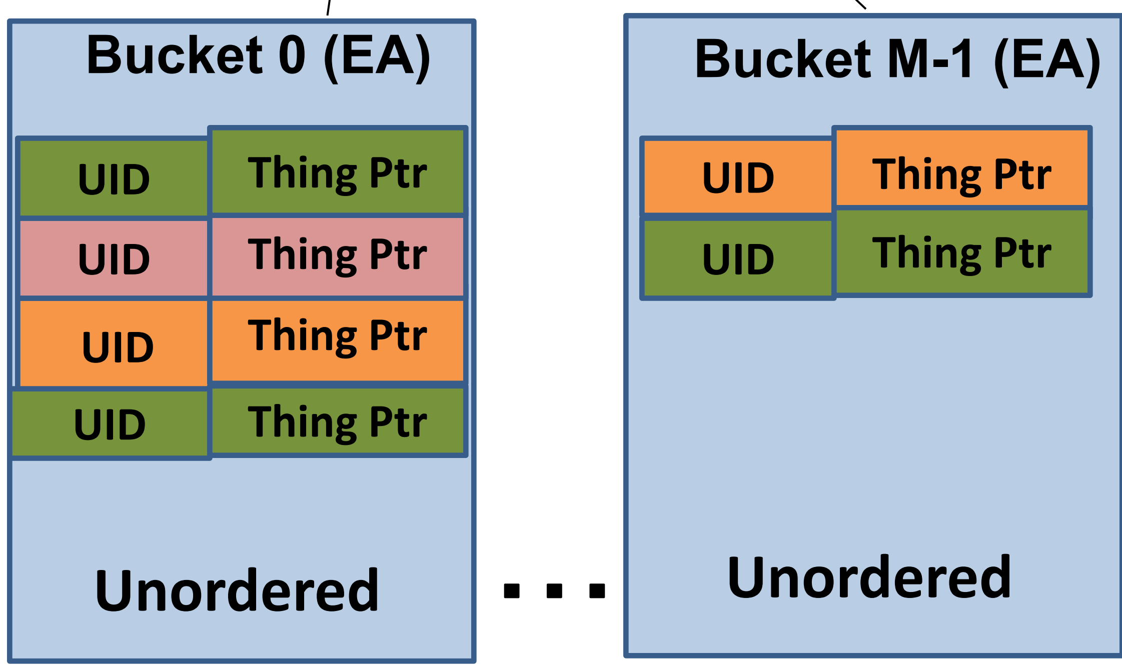
But:

No ordering possible

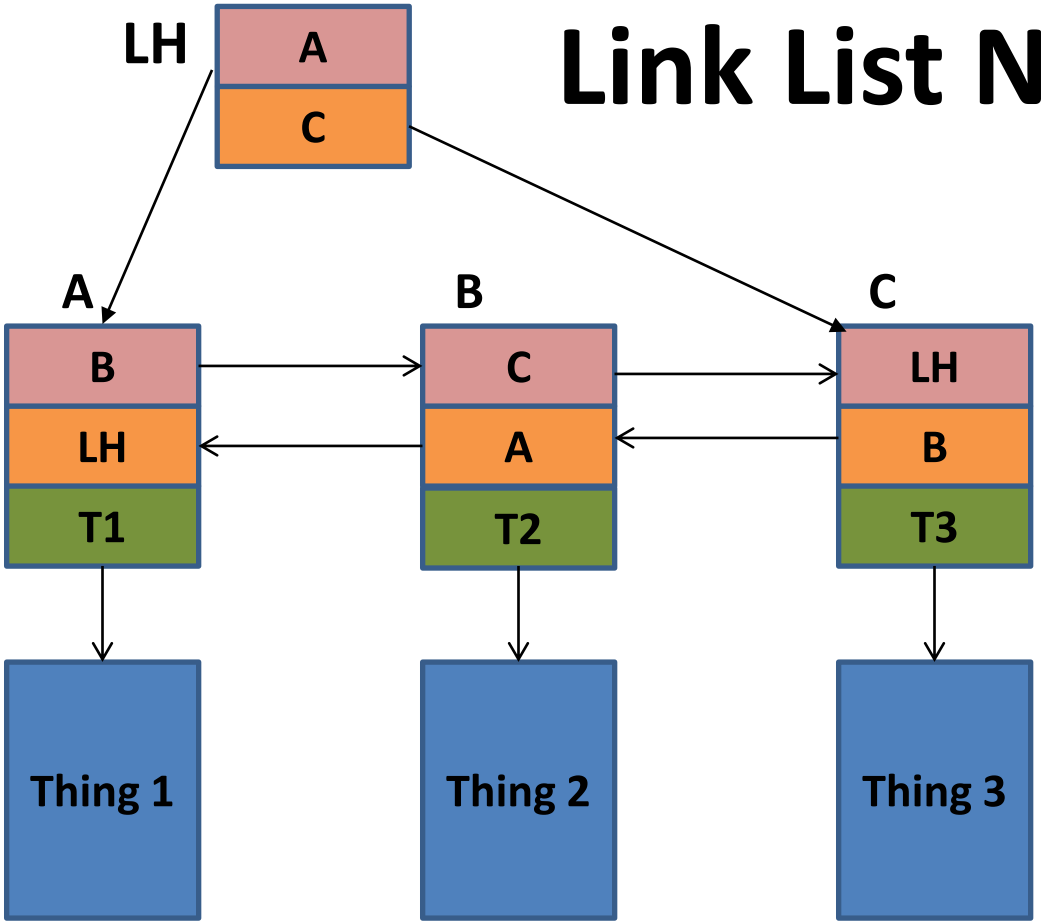
➔ No next/previous Links

➔ Collect ID/Obj Ptr Structs

➔ No Connecting Nodes!!



Link List Nodes



Class ThingNode
{ // Internal List class to support
// a Thing object in a Linked List
private:
ThingNode *fp;
ThingNode *bp;
Thing * tp;

➔ Thing collected in a Linked List but doesn't know it!

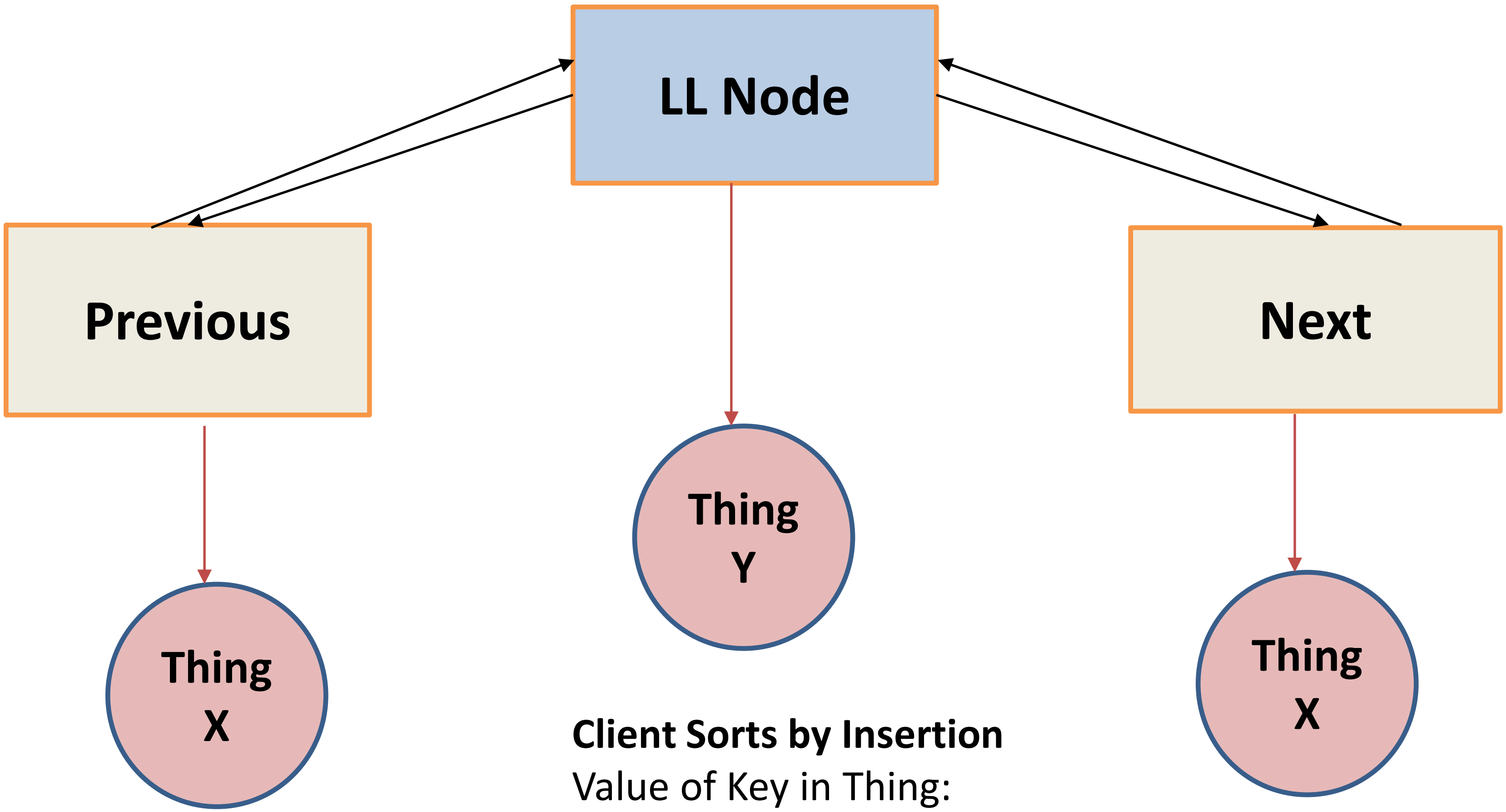
Positional Ordering determined by Client

➔ Object Key unknown to collection class

➔ "Neighbors" **HAVE** to be linked to

➔ **Connecting Nodes required**

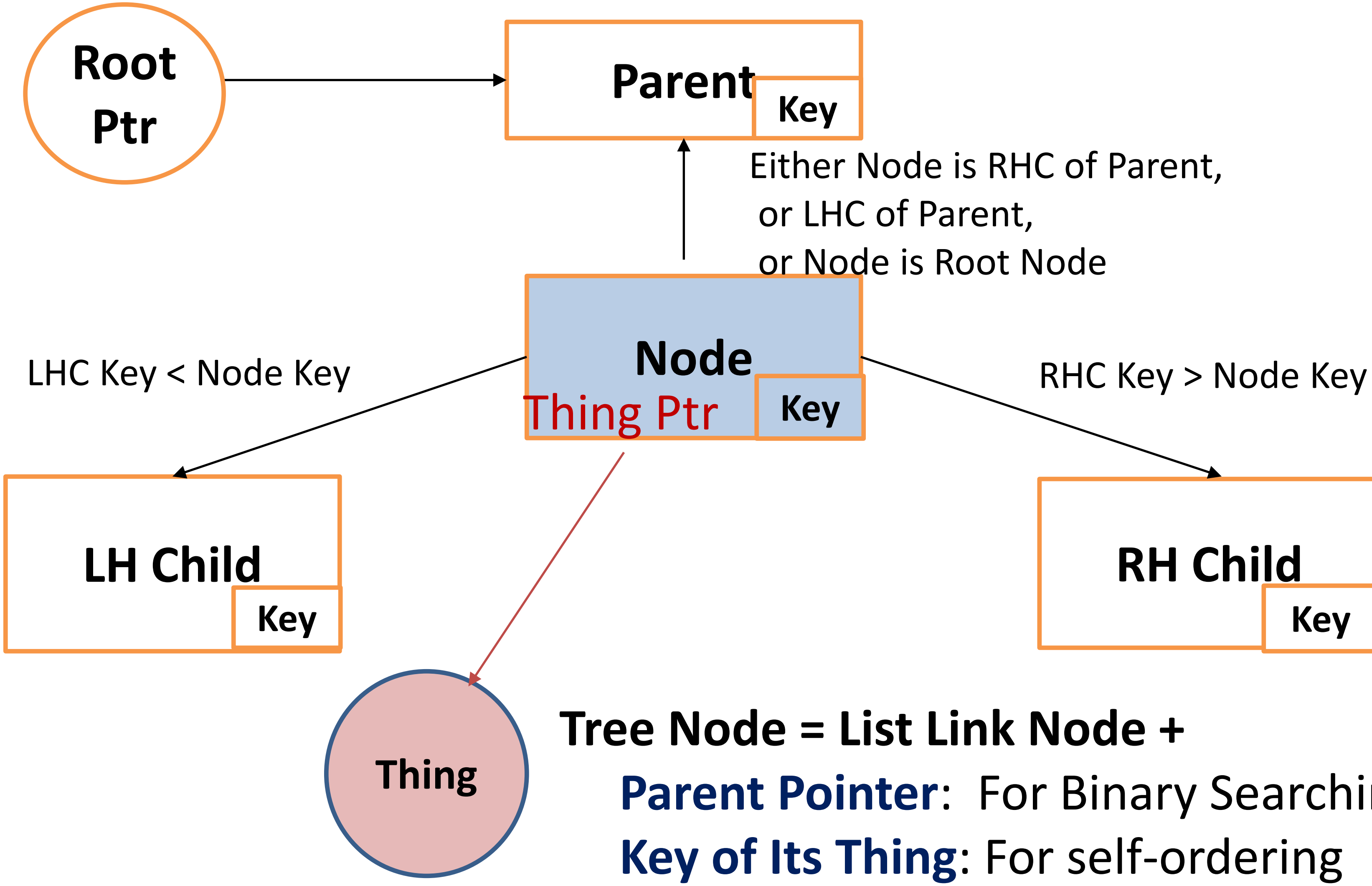
Link List Node Structure



Client Sorts by Insertion
Value of Key in Thing:
X <Y < Z

➔ **If you have ANY Node, you can walk the entire List**

BST Tree Node Structure



Tree Node = List Link Node +

- **Parent Pointer**: For Binary Searching
- **Key of Its Thing**: For self-ordering

➔ **<u>TREE</u>** can sort on Insert!!

➔ **If you have ANY Node, you can walk the entire Tree!!**

Node Interface

```
class Node { // Node created by BSL, interface used by BST, other Nodes
    private:
        Node *parent;  Node *lhc;  Node *rhc;  // Some could be  Null
        String key; Thing *t;  //Key and Pointer to Thing carried by this node
        BST *myTree;  // Could also be static (class level) variable
   public:
          Node(Node* parent, String key, Thing *tp,  BST *bst);
// No Children at Node Creation (BST Needed to change Tree Root Ptr)
          Node (Node&); ~Node();
          String getKey(); Thing* getThing();
          Node* getRHC(), getLHC(), getParent();
          void setRHC(Node*), setLHC(Node*),  setParent(Node*);
};
```

➔ **What is the problem with this Node Interface?**

What is the problem with Node Interface?

```
class Node { // Node created by BST, interface used by BST, other Nodes
   private:
      Node *parent;  Node *lhc;  Node *rhc;  // Some could be  Null
      String key; Thing *t;  //Key and Pointer to Thing carried by this node
      BST *myTree;  // Could also be static (class level) variable
  public:
        Node(Node* parent, String key, Thing *tp,  BST *bst);
// No Children at Node Creation (BST Needed to change Tree Root Ptr)
        Node (Node&); ~Node();
        String getKey(); Thing* getThing();
        Node* getRHC(), getLHC(), getParent();
        void setRHC(Node*), setLHC(Node*),  setParent(Node*);
};
```

➔ **It's all "sets" and "gets"!! So …**

Node is a Private "Linking" Structure which is manipulated by its BST!

```
struct ThingNode { // Node created by BST, data used by BST
{
        Node* parent; // Ptr to parent node (0 if Root)
        Node*lhc;      // Ptr to Left Hand Child
        Node *rhc;     // Ptr to Right Hand Child
        String key;    // Object UID.  Determines ordering
        Thing*Thing; // Pointer to the Thing with that key
} ;
```

➔ No "per-node" local or single global BST ptrs needed because the self-contained BST does all the work

BST Interface: Non-Iterator

// Note: BST does NOT remember Client's current position in Tree.

```
Node* toRoot;      // Pointer to Root Node.  Null if Tree empty
Node* getRoot ():            // Get Root Node or Null (empty Tree)
Node* getNode (Key):      // Get Node with that Key
                 //  Move currNode ptr.
Node *getNext (Node *); // Get Next (higher) Node or NULL (at end)
Node*getPrior (Node *); // Get Prior (lower) Node or NULL (at start)
              // Start at Root Node
Node* insert (Node*); // Utilizes Node Keys
Node* remove (Key); // Remove Node with that key. Return Next or NULL
Node* remove (Node *);  // Remove Node pointed to.  Return Next
             // Initially start at Supplied Node
 void printSelf (Node*); // Print key and Thing data
 void printAll (Node*); // Print key and Thing data + all nodes below
 unsigned int countNodes (Node *); // Return total Nodes below this
 unsigned int getHeight (Node*); // Return Max Height of this node
```

BST Iterator: External

// BST API : "Item" Struct is <Key / Object Ptr>

Iterator* BST::begin (); // Get Iterator (Set to Root Node)

BST*bst; Node*; currNode; // Tree & Current position in Tree (hidden)

// Iterator API: Leaves Current Position of Iterator unchanged

Item* getRoot(); // Get Root Item (i.e. "Get First")

Item* getThis(); // Get current Item

Item* getNext (); // Get Next (higher) Item or NULL (at end)

Item* getPrior (); // Get Prior (lower) Item or NULL (at start)

Item* remove (); // Remove & return current Item, advance position

// Initially start at Tree Root. Changes Current Position of Iterator

Item* getItem (Key): // Gets Item with that Key (Position at item)

Node* insert (Item* add); // Add item to Tree (Position at item)

Node* remove (Key); // Remove Item from Tree (Position at next item)

// Start at Current Position of Iterator. Does not change it

void printSelf (); // Print Item data

void printAll (); // Print Item data + all Items below

unsigned int countNodes (); // Return total # Items below this in Tree

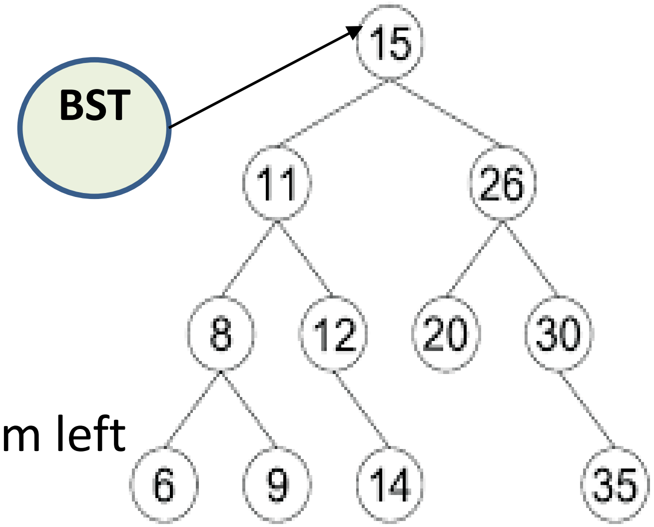
unsigned int getHeight (); // Return Max Height of current Item

Binary Search Tree Operations: Details

(Most are algorithmic / Some are Recursive)

- Count Nodes
- Get Height (longest path to bottom) of Node
- Ordered printout of all Tree Entry Values
- Get next Node
- **Get Previous Node**
- Search for Node
- Insert Leaf Node
- Remove Node

Count Nodes



Count all Nodes below this point in Tree

```
int BST:: countNodes (Node*sNode) {
   int count = 0;
{ // If LHC, recursively call self for LHC. Gets to bottom left
  // Return implies entire LH Hierarchy counted
  // If RHC, recursively call self for RHC. Gets to bottom right
  // Return implies entire RH Hierarchy counted
 // Add 1 to RH count + LH count, and return

  if (sNode→lhc)
      sum+= countNodes (sNode→lhc);
  if (sNode→rhc)
      sum+= countNodes (snode→rhc);
   return (sum+1);
};
```

Called by Iterator::countNodes();

BST::countNodes(Node*) is a simple recursive tree-walking routine.

There will be more!

Get Height

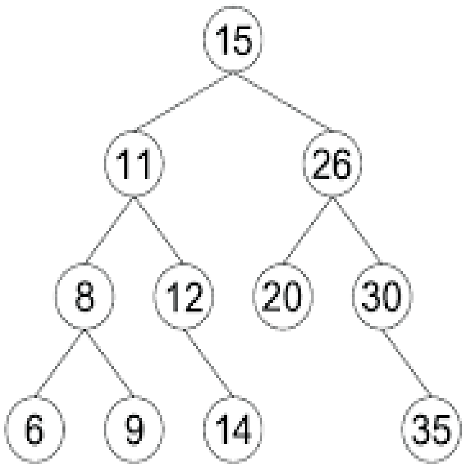
Strategy:

If LHC, invoke self there. Save value +1

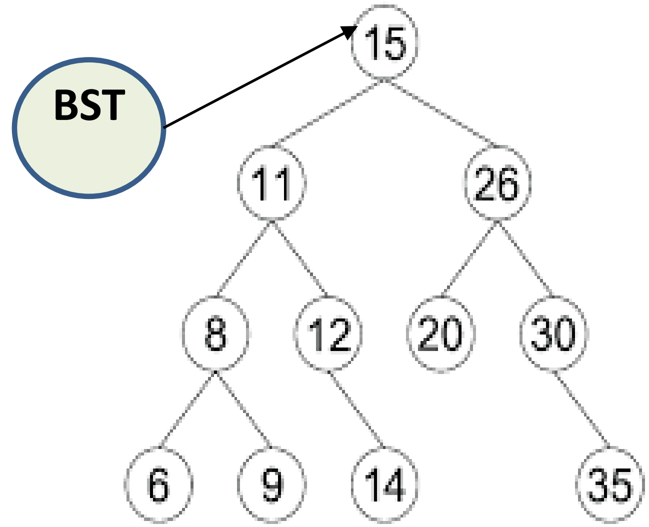
If RHC, invoke self there. Save value +1.

Return height

```
int BST::getHeight( Node* top) { // Return Longest path from Root
      //Increment larger of left / right paths.  Report 0 if Leaf
  int lhgt =rhgt = report = 0;
  if (lhc)
        lhgt =getHeight (lhc) +1;
  if (rhc)
        rhgt = getHeight (rhc) +1;
       // Report back 1 higher than L/R top height
   report = ((lhgt > rhgt) ? lght : rhgt):
   return (report);
}
```



Ordered Print



Print all Key Values below this point

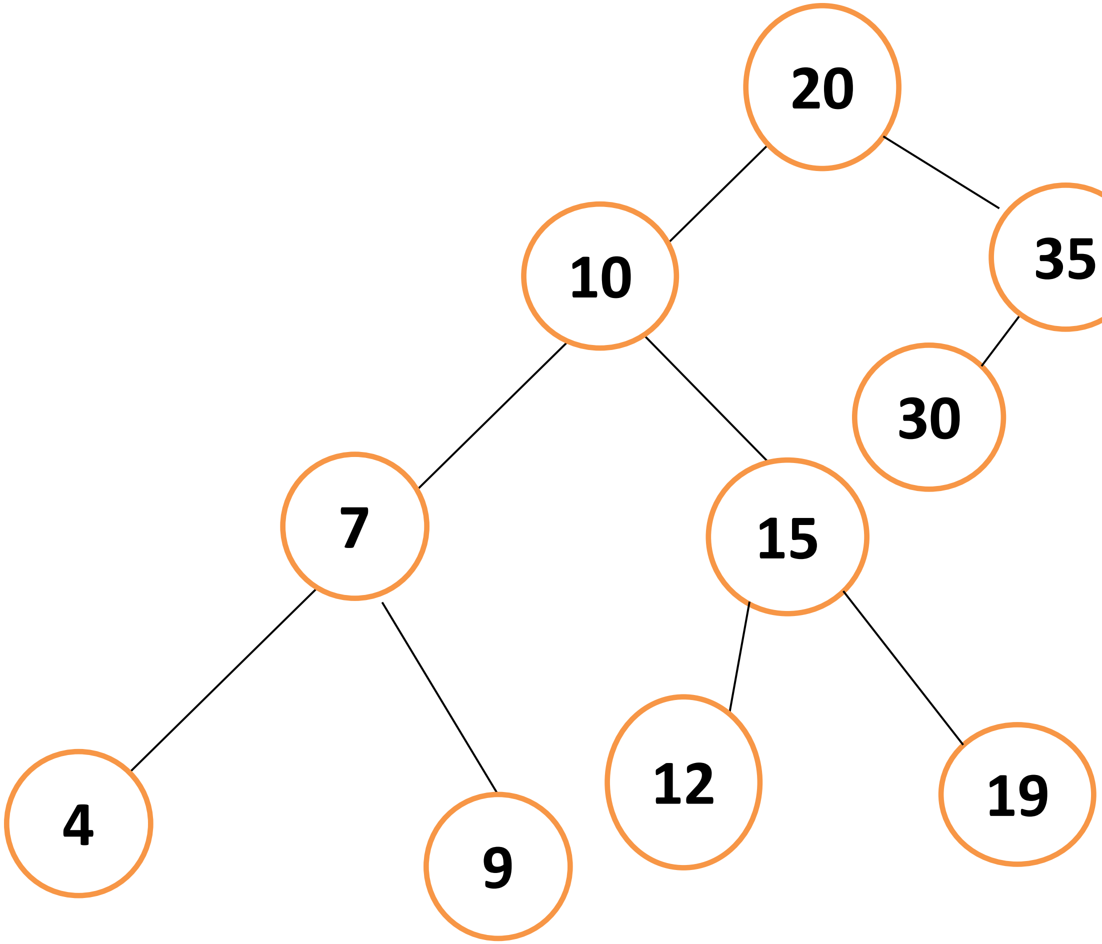
Strategy:

1. If LHC, recursively call this routine for it.
   <Entire Bottom Left printed>
2. Print own Key … you are next
3. If RHC, recursively call this routine for it.
   <Entire Bottom Right printed>

Method:

```
void BST::printSelf (Node*sNode)
{
  if (sNode→lhc) printSelf (sNode→lhc);
          cout <  sNode→key <  endl;
  if (sNode→rhc) printSelf (snode→rhc);
};
```

Get Next Node

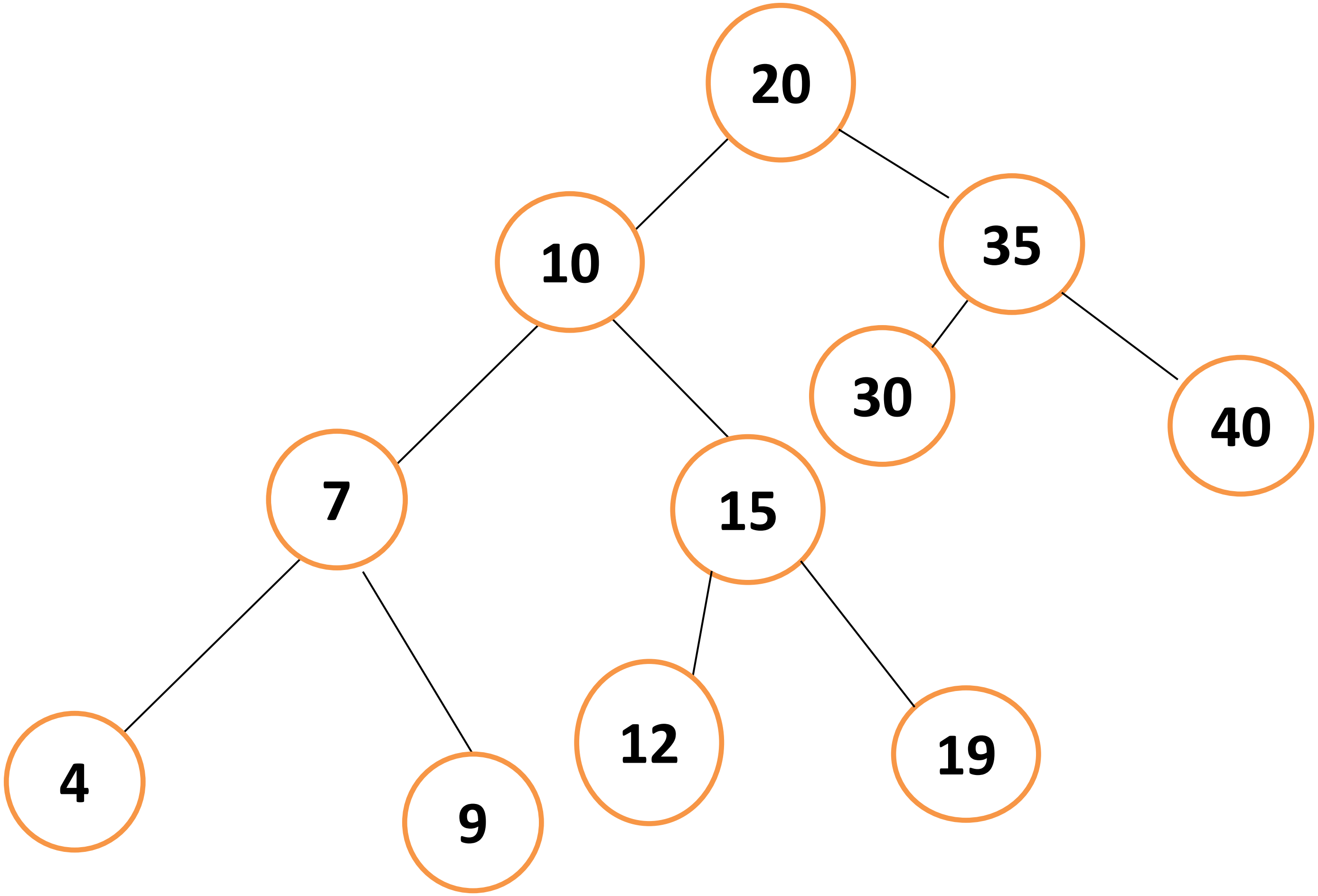


Finding "Successor" node (if exists):

Given a Node on the tree:
How do we find the "next node"?

Starting at:
Node 10?
Node 19?

Get Next Strategy

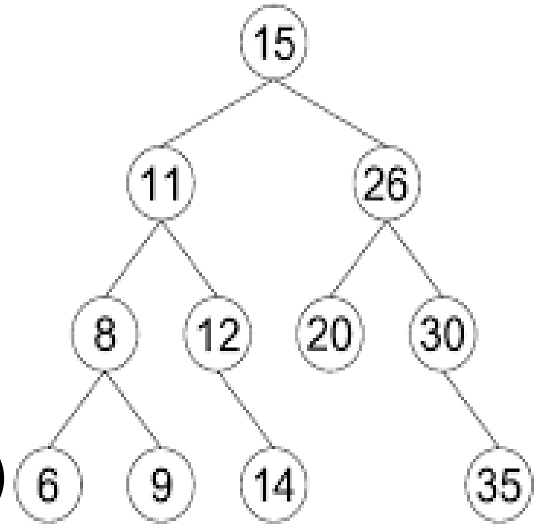


Next Node Starting at:

(10)? **If RHC exists, Go RHC, walk down all LHC's to Leaf** (12)

(19)? **Else, Walk-up Parents till Parent X LHC of GParent Y. Use Y** (20)

Get Next Node Code

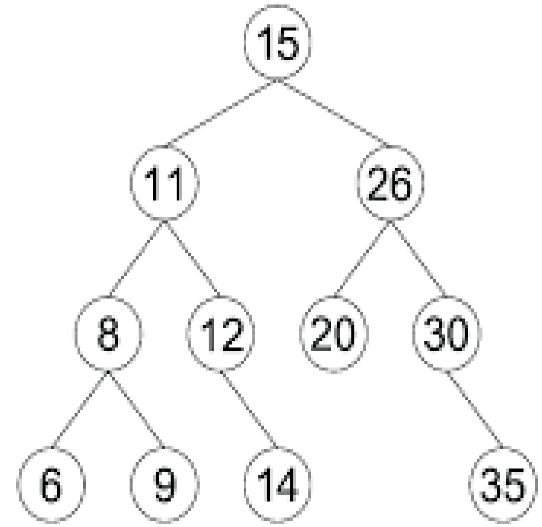


Finding "Successor" node (if exists):

(Non-recursive)

```
Node*BST:: getNext (Node* from) { // Returns next node
    Node *current = NULL;
    If (from->RHC) {   // If RHC, find leftmost leaf (15→20)
        currNode = RHC
        while (currNode→LHC) { currNode = LHC ; }
        return (currNode }
  //No RHC.  Go up tree until parent found where child was LHC. Return Parent
    while (currNode != toRoot) {   // Ensure not root
        if (currNode  == Parent→RHC) {  currNode = Parent };  (14→15)
      };
// If currNode is Root node, this node was RHC to everything INCLUDING Root (35)
// So when the walk is over, the return is Root Parent (Null)
      return (currNode→Parent);
}
```

Homework: Get Previous



Deliverables:

1. List of "Treewalking Rules
2. Code for:

   Node*BST:: getprevious (Node* from)

Finding the "Preceding" node (if exists):

Node* getPrevious (Node* from) { // Returns previous node or Null (if lowest value)
// Null should be returned only for Node 6 in example

Node 15 → 14
Node 20 → 15

Search For

Strategy (Find node == Supplied Key)

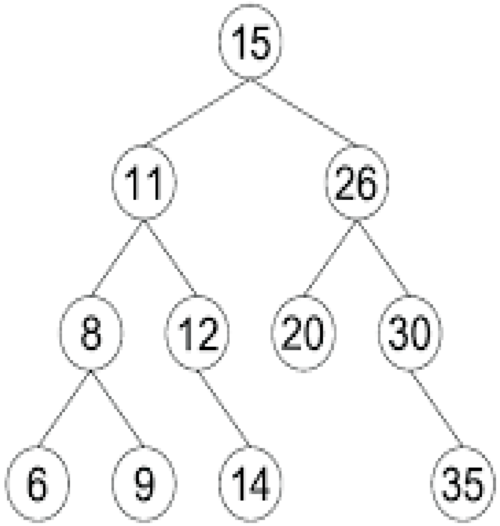
<On initial call, Node Ptr = Root>

How do we find:

Node 9?

Node 20?

Node 28?



Assumes prior check made for empty Tree (in which case, failure is immediate)

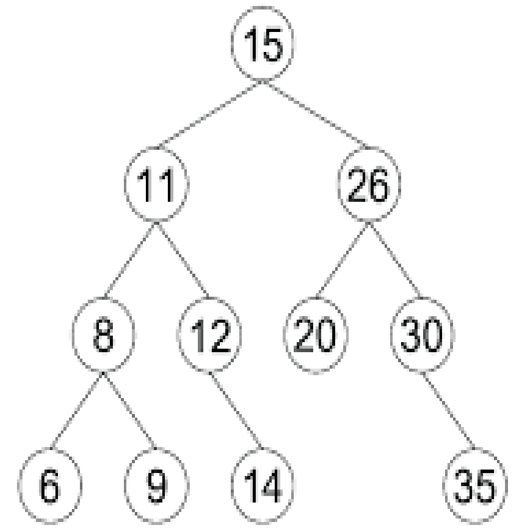
Otherwise. search is invoked with startN = Root

Search For: Strategy

Strategy (Find node == Supplied Key)

<On initial call, Node Ptr = Root>

If key matches this node's key
Set success
Else if LHC and key < LHC key
Recursive call at LHC
Else if RHC and key > RHC Key
Recursive call at RHC
Else (A non-matching key, but can't proceed)
Set failure

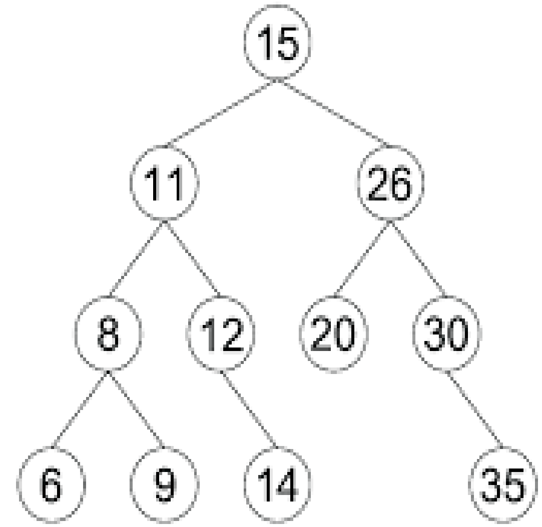


Assumes prior check made for empty Tree (in which case, failure is immediate)

Otherwise. search is invoked with startN = Root

Elements looked at = BST Height +1
(log 2 of N)

Search For: Code

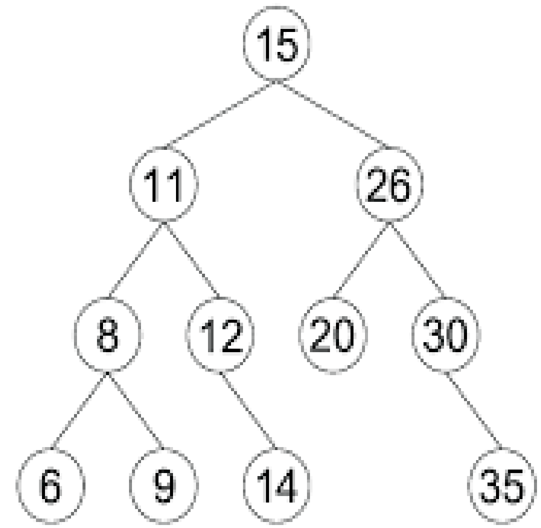


```
Node *search (Node *startN, int desKey)
{  // On initial call, Node Ptr = Root.
  // Object Key is ASSUMED integer
        Node *rval = NULL;  // Assume failure
        Node *currN  = startN;
         if (currN→key == desKey) {
              rval = currN;  }   // Node found. Return Ptr
         else if ((currN→LHC) && (desKey < currN→key))
         { // Recursive call to LHC
                   rval = search (currN->LHC, desKey);}
         else if currN→RHC) && (desKey > currN→key))
          { //  Recursive call to RHC
                   rval = search (currN->RHC, desKey); }
           else
           { // A non-matching Node and we cannot proceed
              // rval already set to failure
           } return (rval);
}
```

Assumes prior check made for empty Tree (in which case, failure is immediate)

Otherwise. search is invoked with startN = Root

Insert Leaf Node



Strategy (Make each new Node a Leaf Node)

<No 2 Nodes with same Key>
<May "unbalance" the Tree>

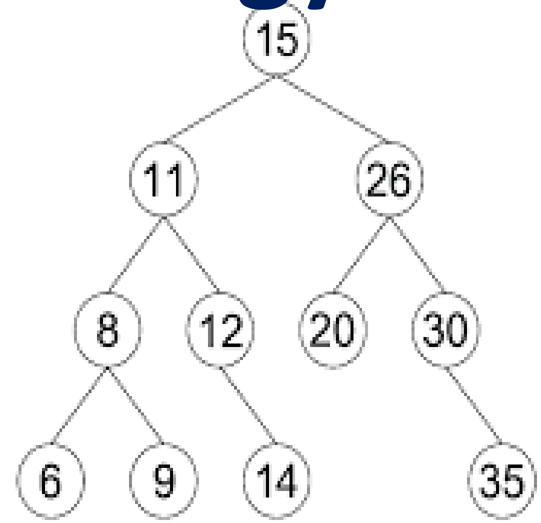
How do we insert:
Node 13?
Node 22?
Node 9?

Assumes prior check made for empty Tree (in which case, newN is made Root)

Otherwise insert is invoked with startN = Root

Do for Start 15 Insert 10.
<Where is 10>

Insert Leaf Node: Strategy



Strategy (Make each new Node a Leaf Node)

<No 2 Nodes with same Key>

<May “unbalance” the Tree>

<Call invoked on “top” node>

```
if LHC and Given Key < Top Node’s Key
        Recursive call at LHC
else if RHC and Given Key > Top Node’s Key
        Recursive call at RHC
else // Can’t descend further
        If Given Key < Supplied Node’s Key
              Insert Given Node as LHC of Top Node
        if Given Key > Supplied Node’s Key
              Insert Given Node as RHC of Top Node
        else // Given Key == Top Node’s Key
              Return error (no duplicate keys allowed)
```

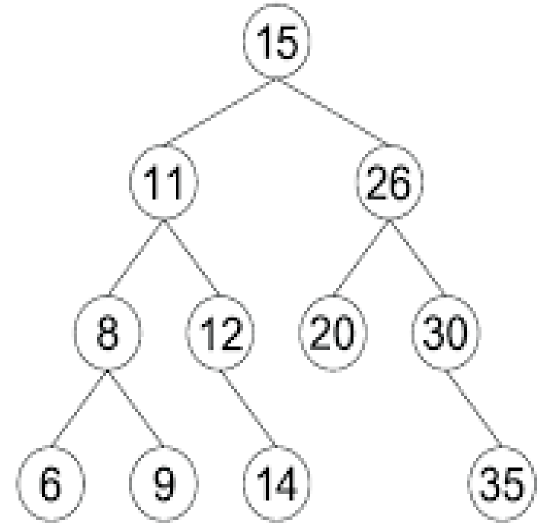
Assumes prior check made for empty Tree (in which case, newN is made Root)

Otherwise insert is invoked with startN = Root

Do for Start 15 Insert 10.

Insert Leaf Node: Code

```
Node *insert (Node *startN, int desKey  Node *newN)
{  // Fails (returns Null) if deskey node already exists
     Node *currN  = startN;
      if ((currN→LHC) && (desKey < currN→key))
      { // Recursive call to LHC
            rval = insert (currN->LHC, desKey,  newN); }
      else if currN→RHC) && (desKey < currN→key))
       { //  Recursive call to RHC
            rval = insert (currN->RHC, desKey, newN); }
      else { // Can't descend further.
              // Insert as leaf child unless key match
          newN->parent = currN;
          rval = newN;  // Assume success
          if (desKey < currN→key)
            { currN→LHC =newN; newN→parent = currN; }
          else if (desKey > currN→key)
           { currN→RHC = newN; newN→parent = currN); }
          else // Keys MATCH!!
              rval = NULL }}
    return (rval);   };
```

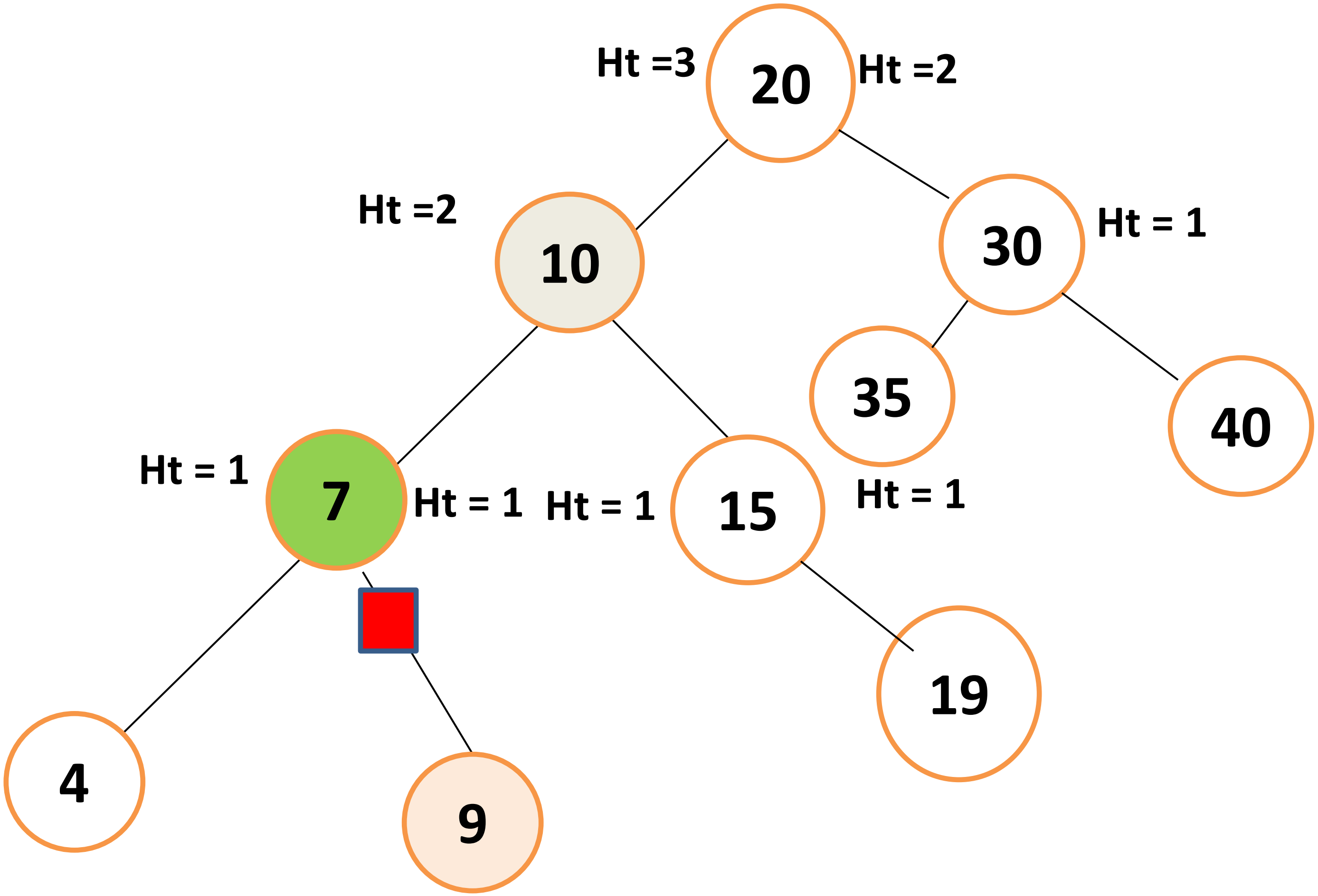


Assumes prior check made for empty Tree (in which case, newN is made Root)

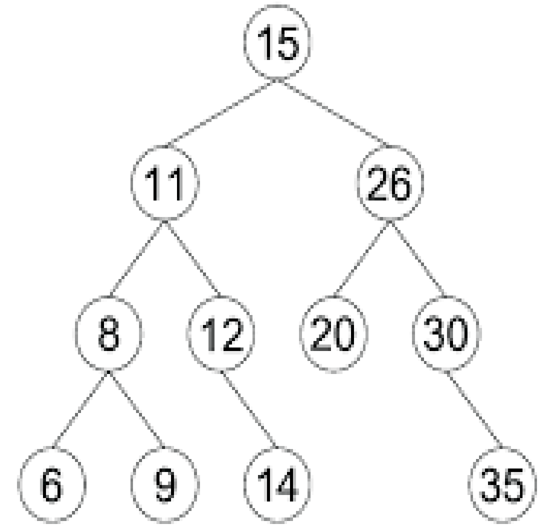
Otherwise insert is invoked with startN = Root

Do for Start 15 Insert 10.

Deletion of Leaf (9)

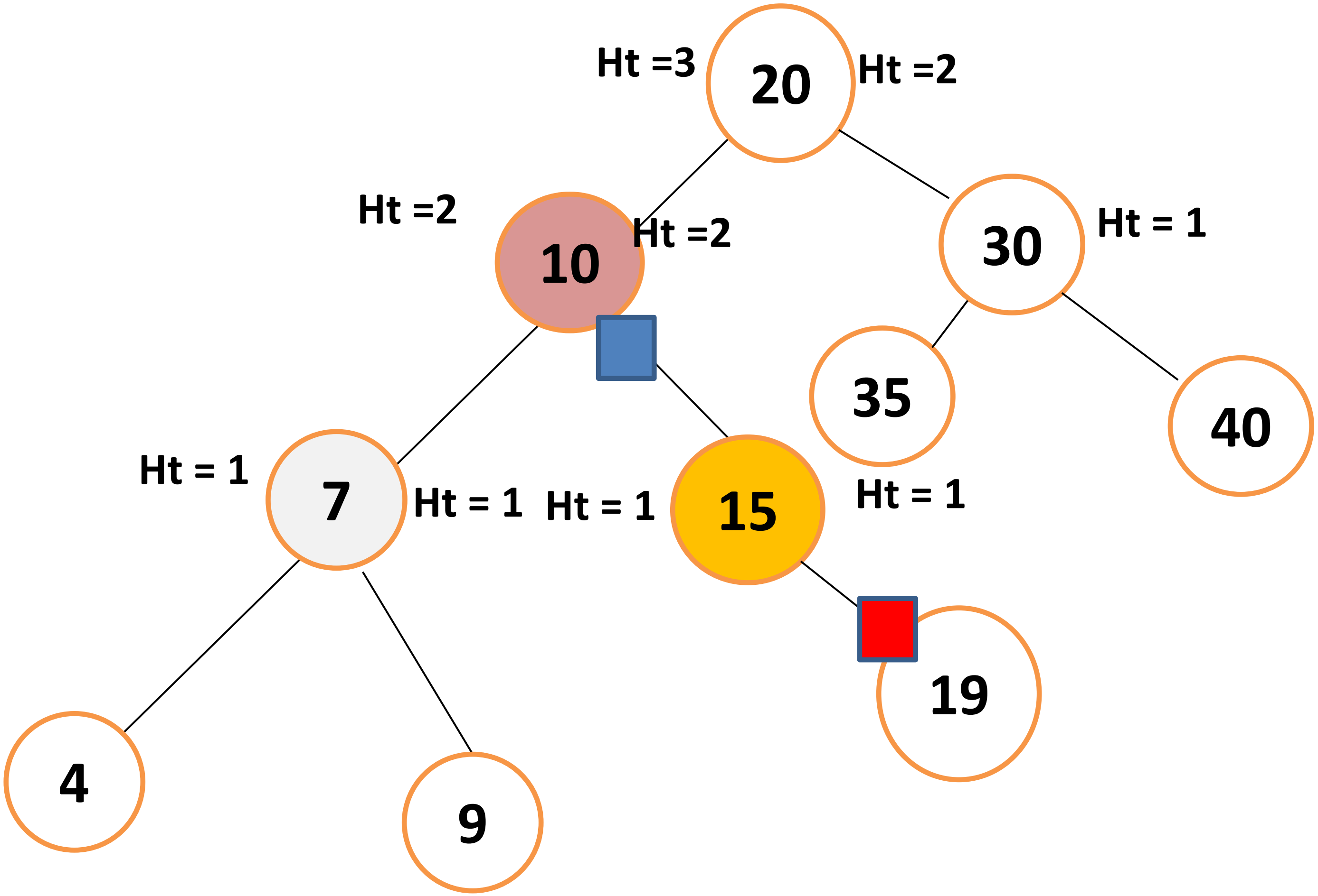


Remove Leaf

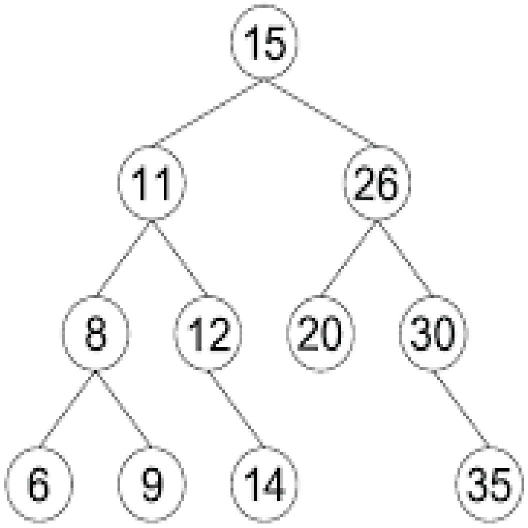



```
Node *remove (Node *tnode)  { // Check children
   Node* rc = tnode→rhc, lc = tnode→lhc,  repl = NULL;
   Node *par = tnode→parent, *w1 = NULL, *w2 = NULL;

// 4 cases need to be dealt with.  We are removing
     // A leaf node
    // A node with only a RHC
    // A node with only a LHC
    // A node with both LHC and RHC (nasty)

   if ((!rc) && (!lc)) // Neither child of node being removed exists: → Leaf
   {  // Leaf Node (ex: 9)
      if (!par) // Root node being deleted is only Node!
          toRoot = NULL;  // BST is empty
    // Clear out Parent pointer to this node
     else if (par→lhc == tnode)
          par→lhc = 0;
     else
          par→rhc =0;
   }
```

Deletion with Only RHC (15)

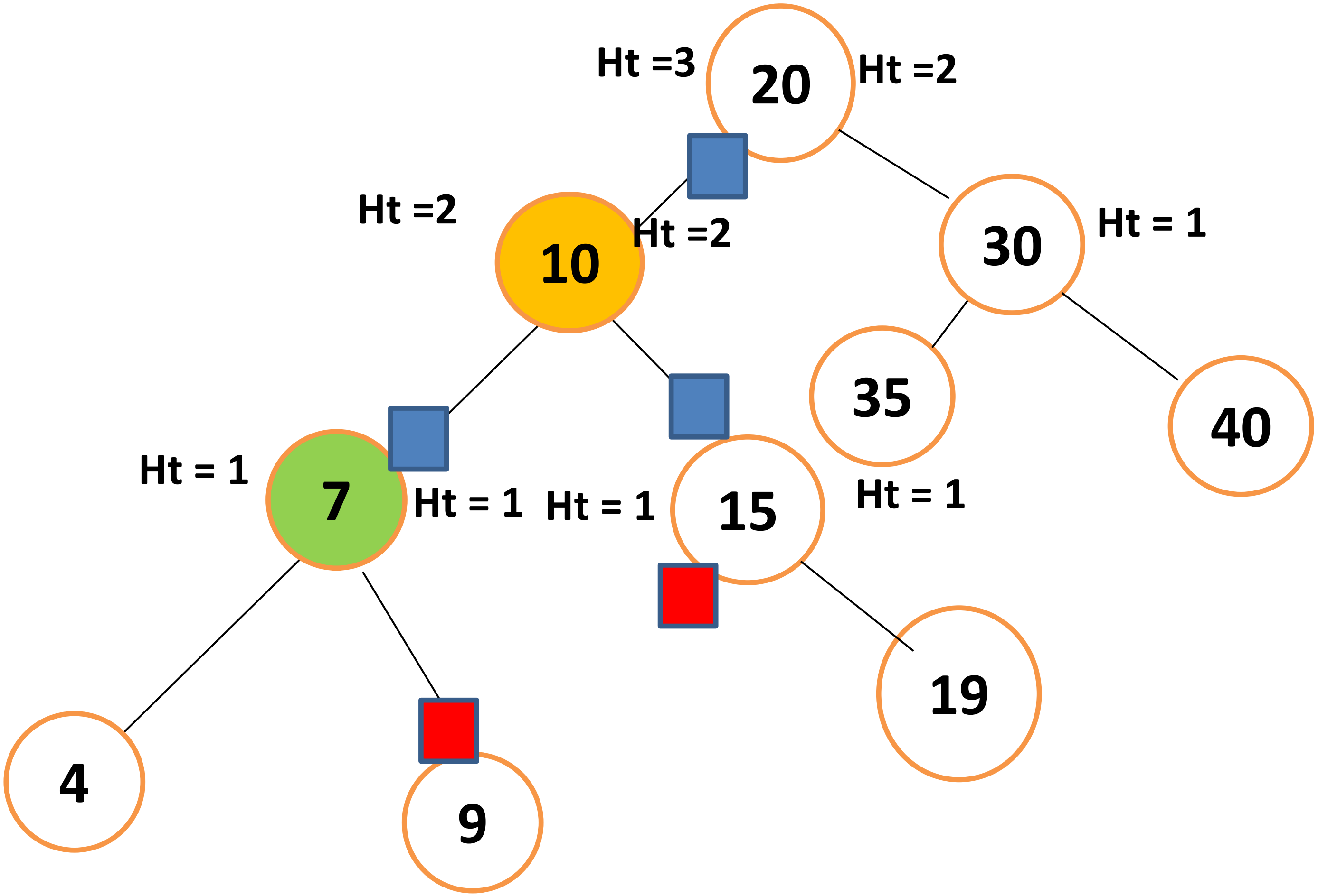


Remove Node with only 1 Child: Strategy




```
else if ((rc) && (!lc)) { // Only RHC  (30)
   if (!par)  { //Root is lowest Node in Tree
        toRoot = rc; rc→parent = NULL;}
   else { // Advance RHC
         rc→parent = par;  // Set 35 Par = 26 and
         if (par→rhc == tnode)  par→rhc = rc; // 26 RC = 35
         else
            par→lhc = rc;
    }
 }
else if ((!rc) && (lc))  {   // Only LHC
    if (!par)  { //Root is highest Node in Tree
        toRoot = lc; lc→parent = NULL;}
   else { // Advance LHC
         lc→parent = par;  // Replace appropriate Child Ptr
         if (par→rhc == tnode)  par→rhc = lc;
         else par→lhc = lc; }}
```

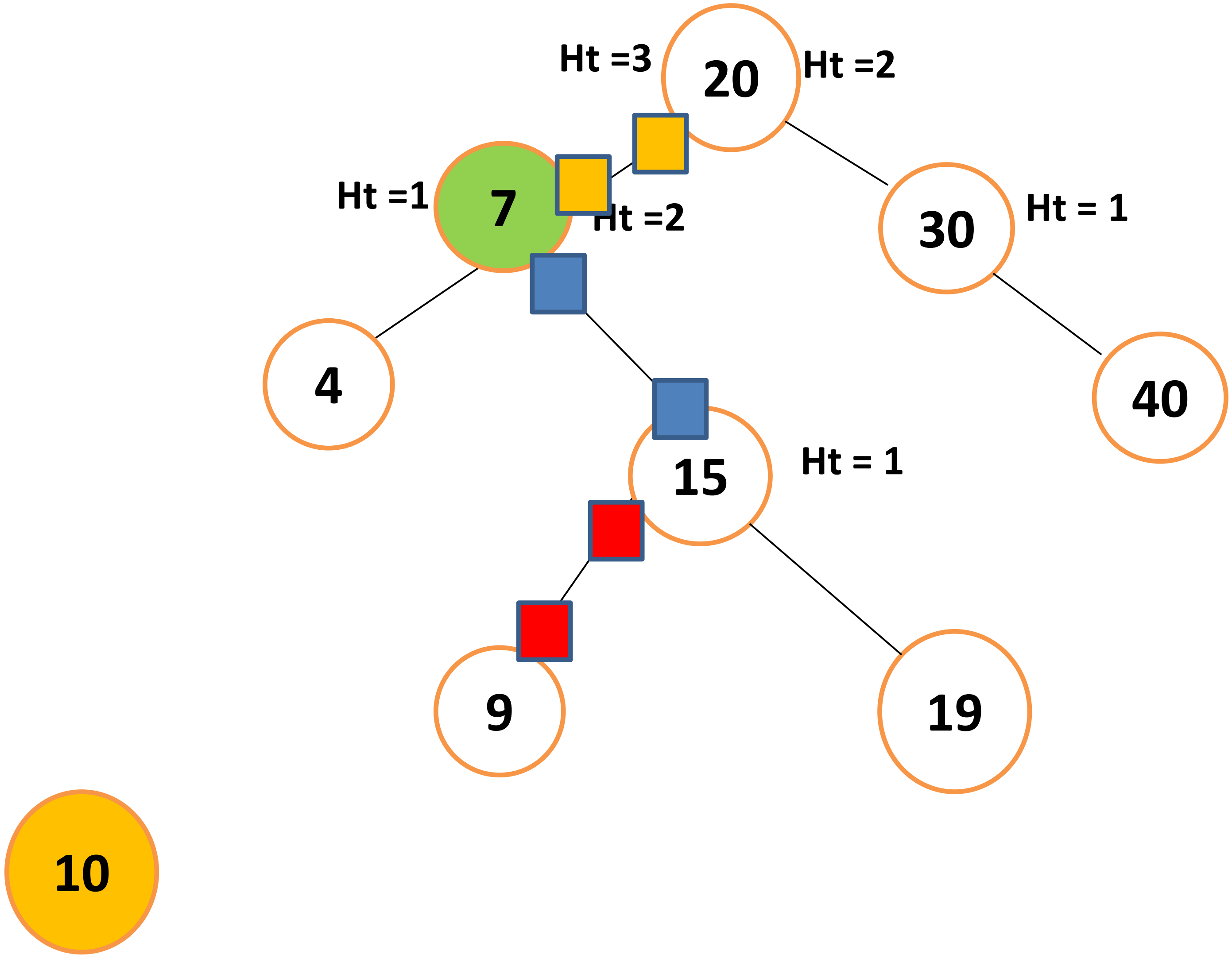
Deletion of Node with LHC & RHC (10)



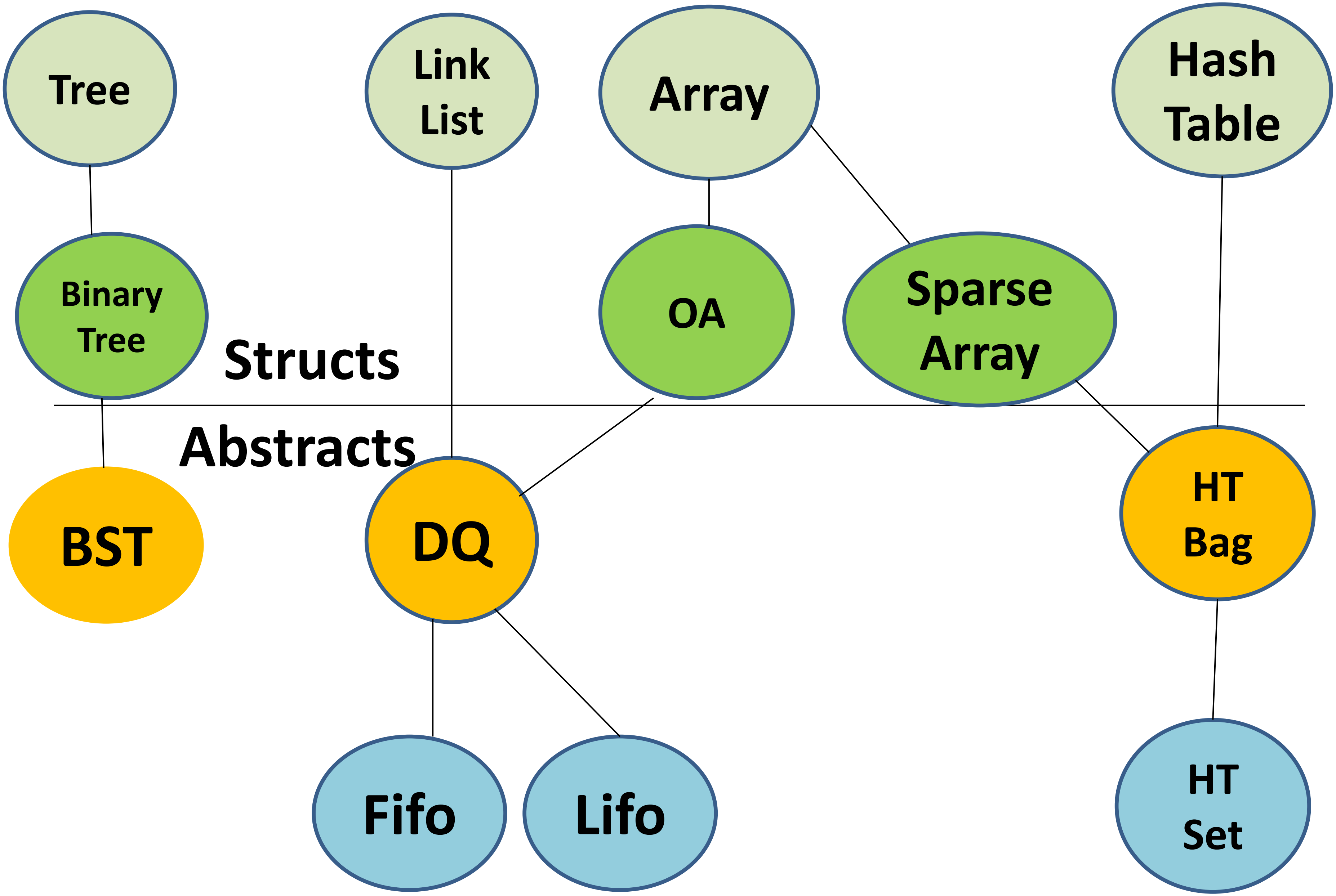
Remove Node with both children

```
else { // Deleted Node has both RHC and LHC (Node 10)
// tnode = 10, rc = 15, lc = 7, par = 20
// Use highest node under LHC as replacement (7)
       repl = lc;  w1 = repl→rhc;  //  Original repl RC reattached later
        tnode→rc→parent = repl; repl→rc = tnode→rc; // repl RC = tnode RC
       if (!par) {toRoot = repl; , repl->parent= NULL;} //New Root
      else { // repl parent = tnode parent
               repl→parent = par; (20)
               if (par→lhc== tnode) par→lhc = repl; // (20 LC = 7)
               else  par→rhc = repl;  }
//   Set RH hierarchy of deleted node as RH of replacement
        repl→rhc = tnode→rhc;
        if (repl→rhc == NULL) repl→rch = w1; // Deleted node had no RC. Reattach orig RC
        else if (w1) { //  Repl had an original RC, hook as LC to lowest LC of first RC
             w2 = repl→rhc;
             while (w2→lhc) { w2 = w2→lhc; } // W2 > repl original RHC , has no LC
             w2→lhc = w1;
             w1→parent = w2;
        }
        return (tnode); };
```

Post Deletion of Node with LHC & RHC



Relationship of Collections



Data Structure Comparisons: Retrievals

(assume N elements in collection)

Data Structure Collection Type	**Retrieval Given "Key"**	**Ordered Retrievals** getNext(),getPrev()	**Maintaining Order**
Ordered Array	**Excellent.** Binary Search for Element **O(Log N)**	**Excellent O(1)**	**Fair.** Ins / Rem ops require bulk moves **O(N/2)**
Linked List (Dequeue)	**Fair.** Search through half of elements **O(N/2)**	**Excellent O(1)**	**Fair.** Search through half of elements **O(N/2)**
Associative Array (Hash Table Set)	**Excellent.** Depends on Entries / Buckets. **O(1)**	**Not Possible**	**N/A**
BST (Ordered, unbalanced)	**Good to Excellent** Depends on balancing. Ranges from **O(Log N)** to **O(N)**	**Excellent (**Walk Tree nodes) **O(1)**	Excellent ignoring Tree Balancing. **O(Log N)**

Data Abstraction Comparisons: Updates

(assume N elements in collection)

Underlying Structure	Data Abstraction	Search	Add / Delete (after search)	Ordering
Array	**Ordered Array**	Binary Search **O(Log N)**	Involves push/pull of multiple elements **O(N/2)**	User determined
Linked List or Ordered Array	**DeQueue**	Run through ½ elements **O(N/2)**	Change a few pointers **O(1)**	User Determined
Hash Table	**Associative Array**	Instantaneous **O(1)**	Instantaneous by supplied Key **O(1)**	N/A
Tree	**Binary Search Tree**	Depends upon balance) **O(Log N) – O(N)**	Add as Leaf Delete (relink) **O(1)**	By Key
Tree	**AVL Tree**	Binary Search	Requires rotations	By Key

Questions?

Review: Data Collection Structures

- **Orderable Array**
  - Extending initial size via reallocate is **slow (2N+malloc)**
  - Access to element via index is **instantaneous**
  - Find element by key is **Log2 N** (using Binary Search)
  - Insert & Remove are **slow (2N = N)**
- **Dequeue**
  - Extending initial size **instantaneous** (independent of N)
  - **Find** element by key **slow** (walk chain **N/2 = N**)
  - **Insert / Remove** by Key is slow (precede with **Find**)
  - **next / previous** is **instantaneous**
- **Hash Table**
  - Extending initial size **instantaneous** (independent of N)
  - Find element by key **instantaneous** (hash time)
  - **No ordering possible**