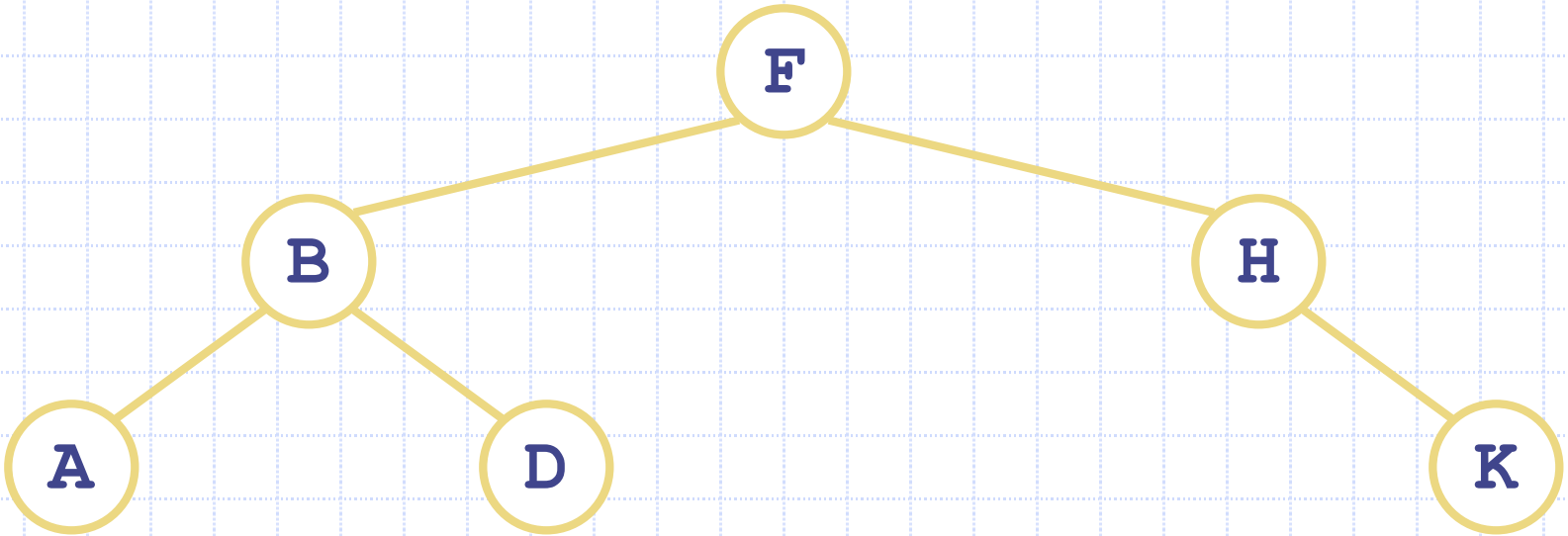
Lesson 9
Binary Search Trees
: *Solving Problems by Engaging the Field of Solutions*

Binary search trees make it possible to store data in memory while preserving a specified order in a way that cannot be achieved as efficiently using any kind of a list. However, their worst case performance, which can potentially occur more often than desirable, reduces their efficiency to that of a linked list. One example of reducing or eliminating this worst case performance is through the introduction of a *balance condition* to ensure the tree never becomes skewed (for example, AVL trees and Red-Black trees).

Maharishi's Science of Consciousness: Balance is the expression of the *invincible* quality of pure creative intelligence, which preserves the integrity of unboundedness even as it is expressed within boundaries.

Review: Binary Search Trees

- BST property:
  key[leftSubtree(x)] ≤ key[x] ≤ key[rightSubtree(x)]
- Example:



Inorder Tree Walk

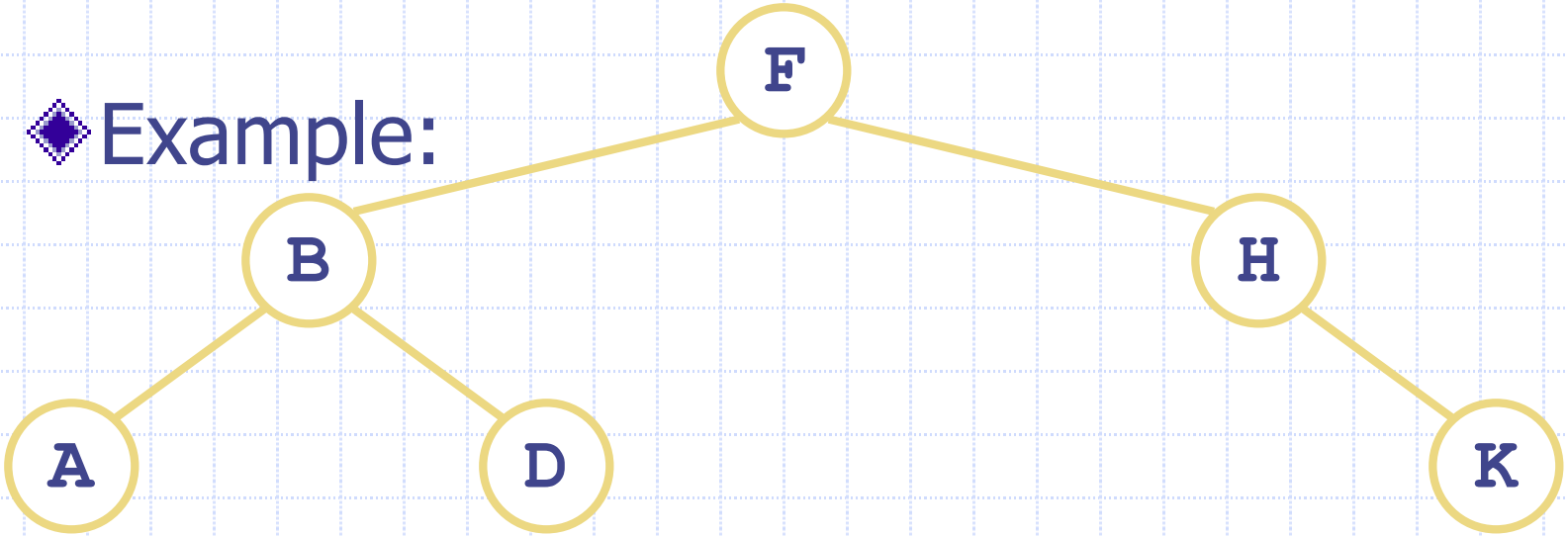
- *What does the following code do?*

```
TreeWalk(x)
        TreeWalk(left[x]);
        print(x);// root
        TreeWalk(right[x]);
```

- A: prints elements in sorted (increasing) order (left, root, right)
- This is called an *inorder tree walk*
  - *Preorder tree walk*: print root, then left, then right
  - *Postorder tree walk*: print left, then right, then root

Inorder Tree Walk

- Example:



- *How long will a tree walk take?* $O(n)$
- *Prove that inorder walk prints in monotonically increasing order*

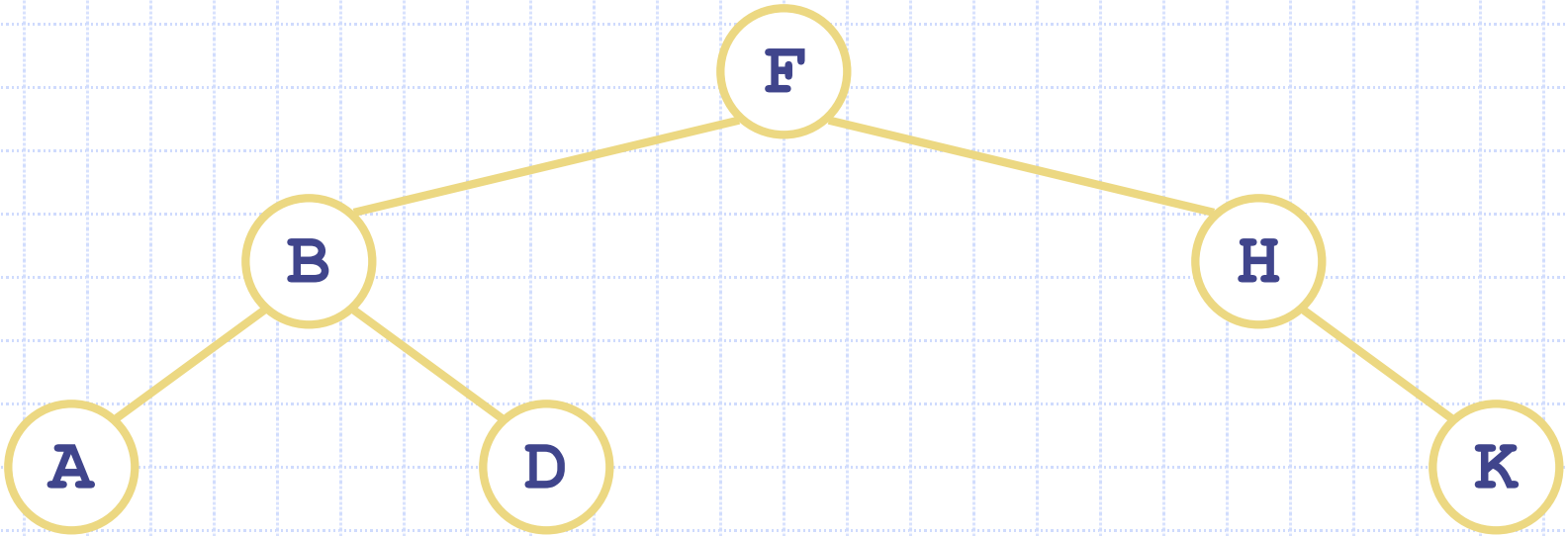
Operations on BSTs: Search

- Given a key and a pointer to a node, returns an element with that key or NULL:

```
TreeSearch(x, k)
    if (x = NULL  or  k = key[x])
        return x;
    if (k < key[x])
        return TreeSearch(left[x], k);
    else
        return TreeSearch(right[x], k);
```

BST Search: Example

- Search for *D* and *C*:



Operations on BSTs: Search

- Here's another function that does the same:

```
TreeSearch(x, k)
    while (x != NULL  and  k != key[x])
        if (k < key[x])
            x = left[x];
        else
            x = right[x];
    return x;
```

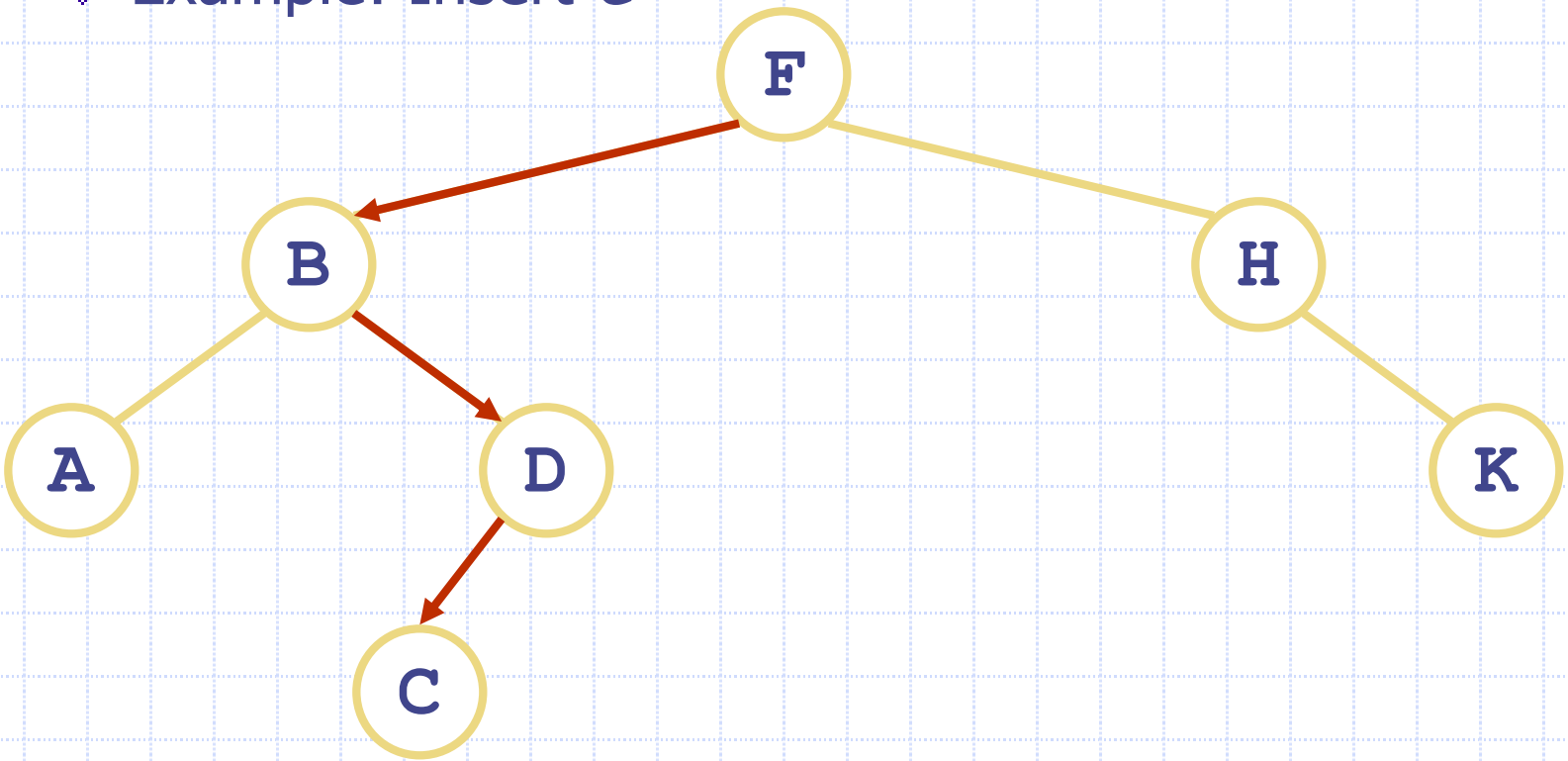
- *Which of these two functions is more efficient?* → while loop

Operations of BSTs: Insert

- Adds an element x to the tree so that the binary search tree property continues to hold
- The basic algorithm
  - Like the search procedure above
  - Insert x in place of NULL
  - Use a "trailing pointer" to keep track of where you came from (like inserting into singly linked list)

BST Insert: Example

- Example: Insert *C*



BST Search/Insert: Running Time

- *What is the running time of TreeSearch() or TreeInsert()?*
- A: O(h), where h = height of tree
- *What is the height of a binary search tree?*
- A: worst case: h = O(n) when tree is just a linear string of left or right children
  - We'll keep all analysis in terms of h for now
  - Later we'll see how to maintain h = O(lg n)

Sorting With Binary Search Trees

- Informal code for sorting array A of length *n*:

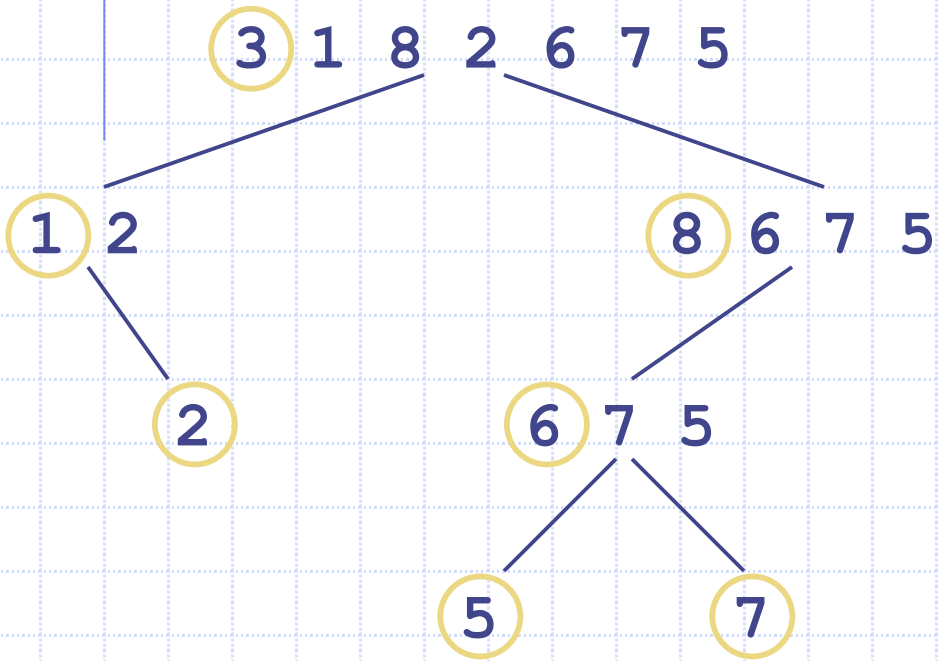
```
BSTSort(A)
        for i=1 to n
                TreeInsert(A[i]);
        InorderTreeWalk(root);
```

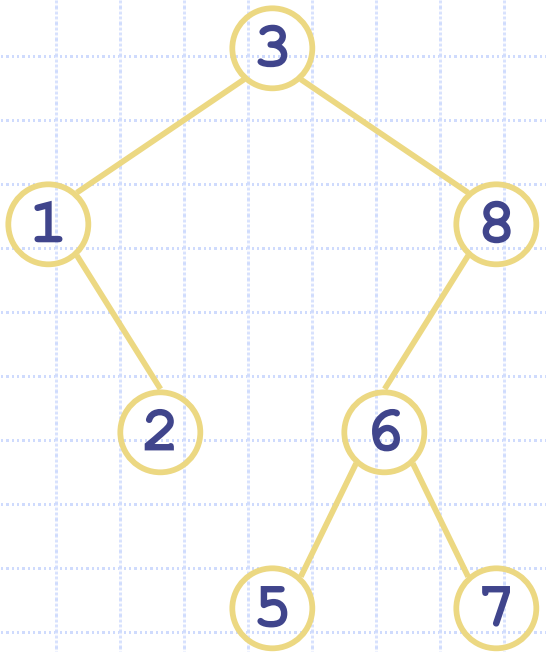
- *Argue that this is $\Omega(n \lg n)$*
- *What will be the running time in the*
  - *Worst case?*
  - *Average case? (hint: remind you of anything?)*

Sorting With BSTs

```
for i=1 to n
    TreeInsert(A[i]);
InorderTreeWalk(root);
```

- Average case analysis
  - It's a form of quicksort!





Sorting with BSTs

- Same partitions are done as with quicksort, but in a different order
  - In previous example
    - Everything was compared to 3 once
    - Then those items < 3 were compared to 1 once
    - Etc.
  - Same comparisons as quicksort, different order!
    - Example: consider inserting 5

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: O(n lg n)
- *Which do you think is better, quicksort or BSTsort? Why?*

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: O(n lg n)
- *Which do you think is better, quicksort or BSTSort? Why?*
- A: quicksort
  - Better constants
  - Sorts in place
  - Doesn't need to build data structure

More BST Operations

- BSTs are good for more than sorting. For example, can implement a priority queue
- *What operations must a priority queue have?*
  - Insert
  - Minimum
  - Extract-Min

BST Operations: Successor

- For deletion, we will need a Successor() operation
- Successor example – next slide
- *What is the successor of node 3? Node 15? Node 13?*
- *What are the general rules for finding the successor of node x? (hint: two cases)*

Successors - Example

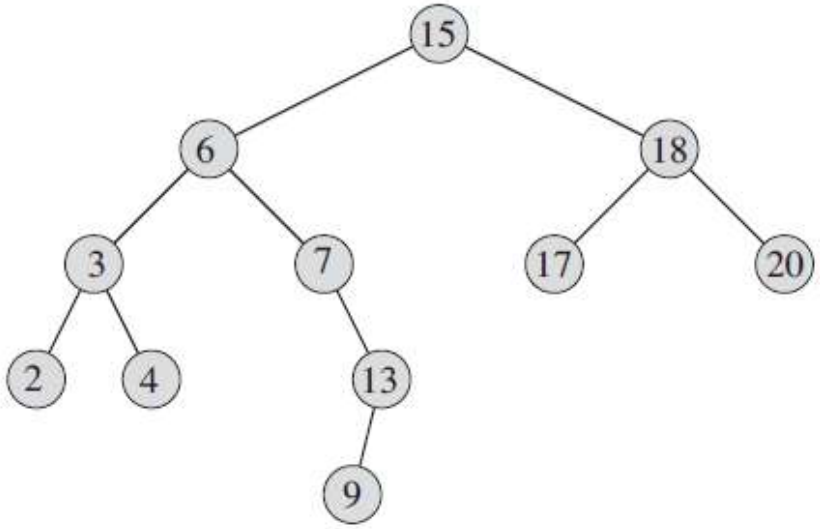


Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$ from the root. The minimum key in the tree is 2, which is found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

BST Operations: Successor

- Two cases:
  - x has a right subtree: successor is minimum node in right subtree
  - x has no right subtree: successor is first ancestor of x whose left child is also ancestor of x
    - Intuition: As long as you move to the left up the tree, you're visiting smaller nodes.
- Predecessor: similar algorithm

BST Operations: Successor

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2       return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5       x = y
6       y = y.p
7  return y
```

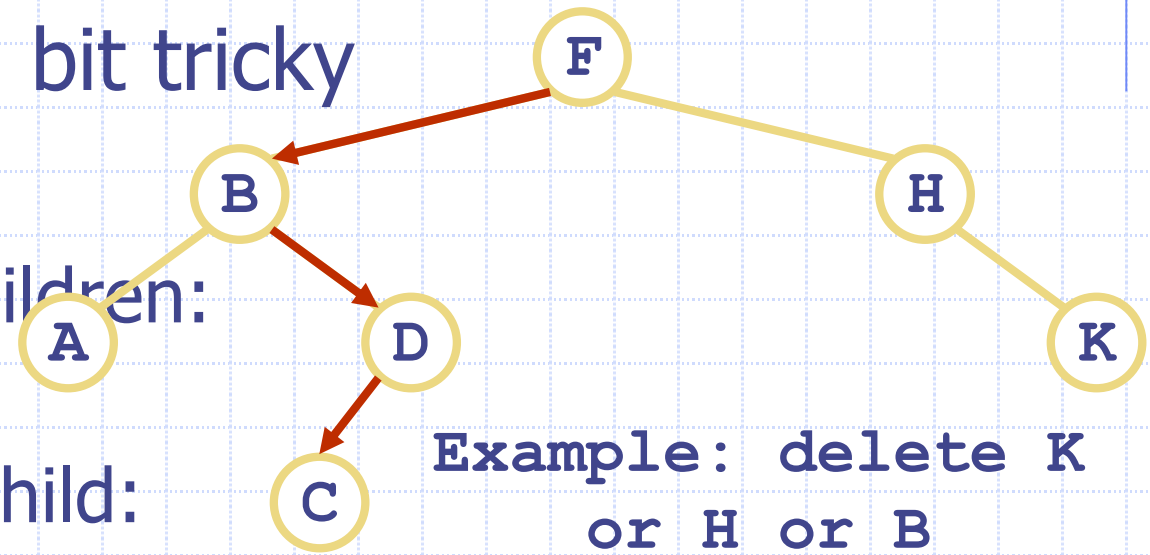
BST Operations: Predecessor

Predecessor:

- Predecessor is the left subtree's right-most child or
- The node with largest key that belongs to the tree and that is strictly less than x's key.
  So, In the previous Fig (3.2) ->
- predecessor of 17 is 15 , predecessor of 6 is 4 and predecessor of 13 is 9.

BST Operations: Delete

- Deletion is a bit tricky
- 3 cases:
  - x has no children:
    - Remove x
  - x has one child:
    - Splice out x
  - x has two children:
    - Swap x with successor y which has at most ONE CHILD
    - Perform case 1 or 2 to delete it



Example: delete K or H or B

BST Operations: Delete

- *Why will case 2 always go to case 0 or case 1?*
- A: because when x has 2 children, its successor is the minimum in its right subtree
- *Could we swap x with predecessor instead of successor?*
- A: yes. *Would it be a good idea?*
- A: might be good to alternate

Main Point – BST

- A Binary Search Tree can be used to maintain data in sorted order more efficiently than is possible using any kind of list. Average case running time for insertions and searchers is O(log n).
- In a Binary Search Tree that does not incorporate procedures to maintain balance, insertions, deletions and searches all have a worst-case running time of $\Omega(n)$. By incorporating balance conditions, the worst case can be improved to O(log n).

Main Point – BST

- *Transcendental Consciousness* is the field of perfect balance. All differences have Transcendental Consciousness as their common source.
- *Impulses Within The Transcendental Field.* The sequential unfoldment that occurs within pure consciousness and that lies at the basis of creation proceeds in such a way that each new expression remains fully connected to its source. In this way, the balance between the competing emerging forces is maintained.
- *Wholeness Moving Within Itself.* In Unity Consciousness, balance between inner and outer has reached such a state of completion that the two are recognized as alternative viewpoints of a single unified wholeness.