Implementing tree traversals

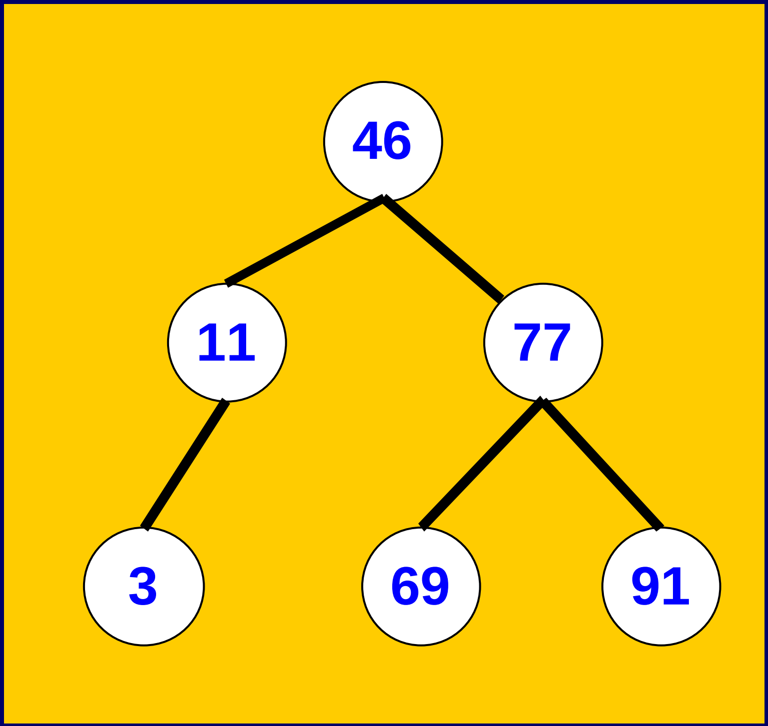
- Naturally recursive functions
  - Order of recursive calls determines traversal order
    - Remember recursive ruler tick-mark drawing?
- e.g., function to “visit” nodes in-order:

```
void inOrderTraverse(TreeNode *n) {
    if (n != NULL) {
        inOrderTraverse(n->left); /* A */
        visit(n); /* B */
        inOrderTraverse(n->right); /* C */
    }
}
```

- Pre-order: B A C; Post-order: A C B

Binary search trees – BSTs

- Order rule for BSTs – for a tree node, n:
  - Info in left subtree of n is less than info in n
  - Info in right subtree of n is greater than info in n
- Tree may not contain any duplicate info
- No rule for tree shape (except must be binary)



Searching a BST iteratively

- e.g., return pointer to node with “key” info:

```
TreeNodePointer n = tree; /* aim at root */
while (n != NULL && n->info != key)
    if (key < n->info) /* search left subtree */
        n = n->left;
    else /* search right subtree */
        n = n->right;
return n; /* either NULL, or node with key info */
```

- Each iteration eliminates half of remaining nodes
  - So logarithmic complexity class
  - Similar result applies to many binary tree functions

Searching a BST recursively

- Must have access to nodes

```
TreeNodePointer findNode(DataType key,
                TreeNodePointer n){ …
  if (n is NULL || n->info equals key)
      return n;    /* works for both base cases */
  else if (key is less than n->info)
      return findNode(key, n->left);
  else return findNode(key, n->right);
  }
```

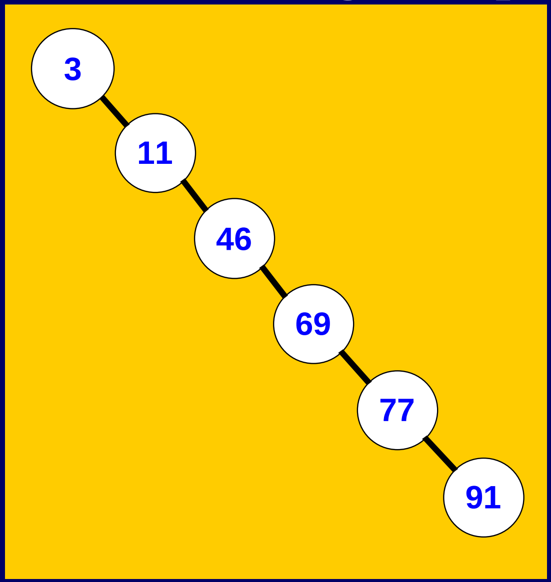
BST search efficiency

- Q: what determines the *average* time to find a value in a tree containing n nodes?
- A: the average path length from root to nodes
  – How long is that?
  – *If full tree*, then 1 node at depth 0, 2 nodes at depth 1, 4 nodes at depth 2, 8 nodes at depth 3, …, to log *n* depths

$$average = \frac{1}{n} \cdot \sum_{i=0}^{\log n} 2^i \cdot i \approx \log n$$

- But …
  – … tree must be balanced!
  – Or complexity can reach O(n)

→



Insert to a BST

- Same general strategy as find operation:

```
if (info < current node) insert to left;
else if (info > current node) insert to right;
else – duplicate info – abort insert;
```

- Use either iterative or recursive approach
- 2 potential base cases for recursive version
  - Already in tree – so return false; do not insert again
  - An empty tree where it should go – so set parent link

Insertion order affects the tree?

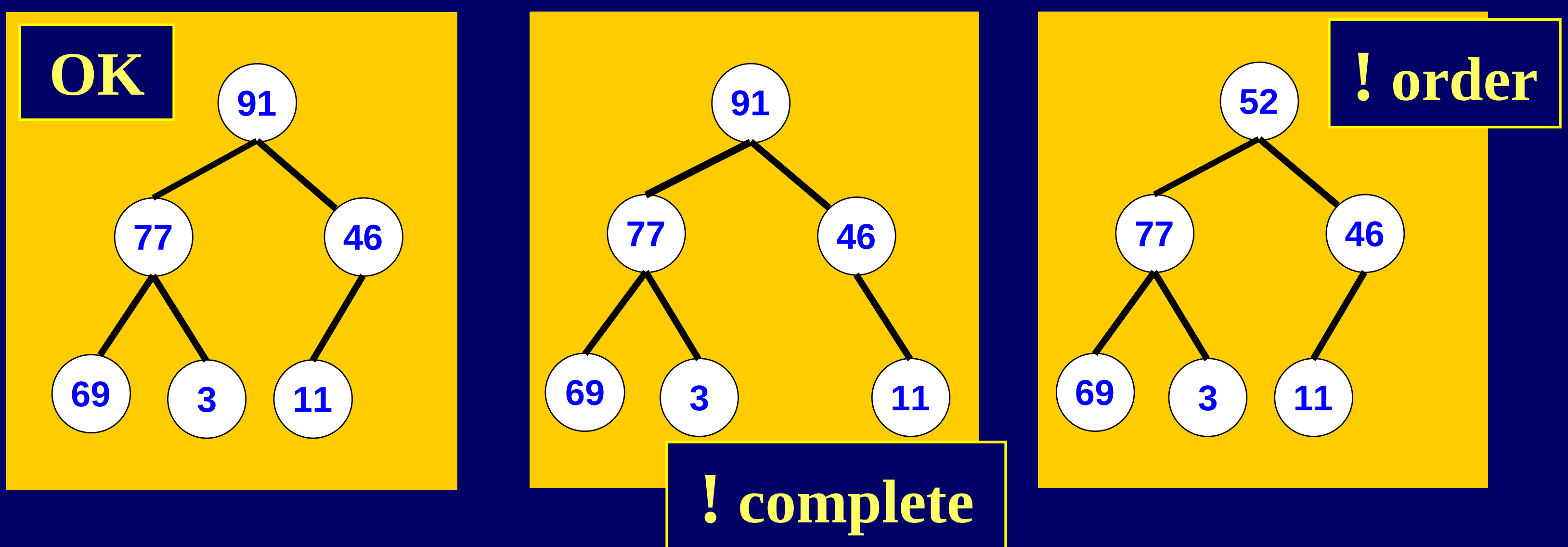
- Try inserting these values *in this order*:
  `6, 4, 9, 3, 11, 7`
- Now insert same values in this order:
  `3, 4, 6, 7, 9, 11`
- Moral: sorted order is bad, random is good.
- Alternative is to set up self-balancing trees (see AVL trees in text)

Deleting a node (outline)

- All depends on how many children the node has
- No children: no problem – just delete it (by setting appropriate parent link to NULL)
- One child: still easy – just move that child "up" the tree (set parent link to that child)
- Two children: more difficult
  - Basic strategy: replace node's *info* with (either) largest value in its left subtree (or smallest in right subtree) – can lead to 1 more delete

Heaps – another type of tree

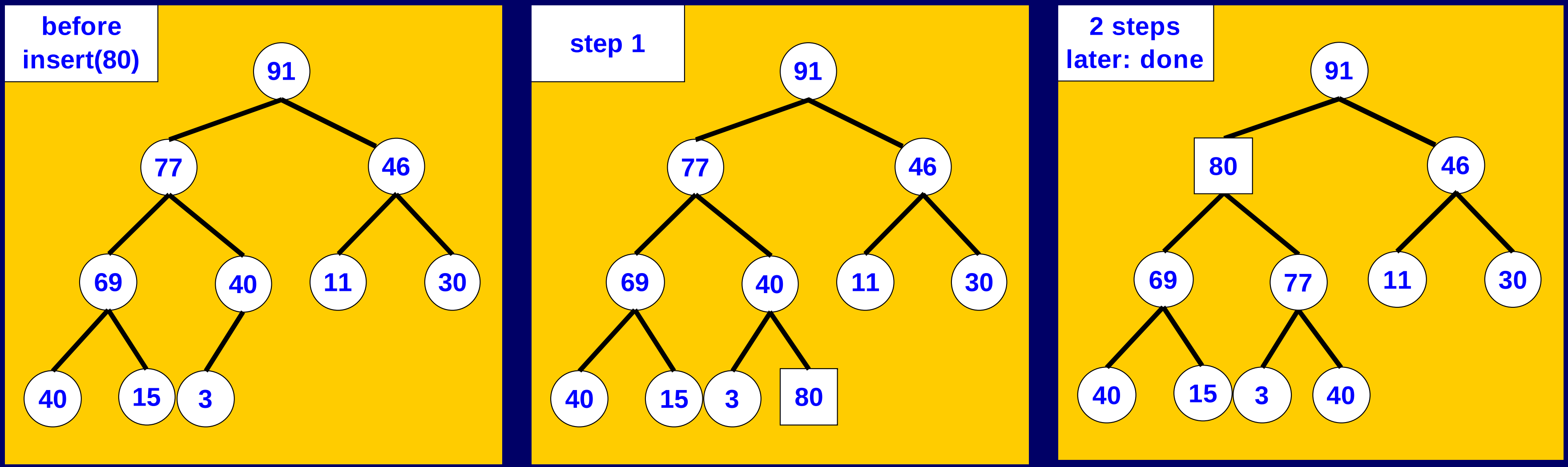
- Complete binary trees, whose items must be comparable and stored in heap order
  - Heap order – a node's information is never less than the information of one of its children



Inserting an item in a heap

- `insertHeap` algorithm keeps complete / in order:

```
put item in first available slot; /*keep complete*/
while (new info > parent info)
    swap info with parent;        /* "reheapify" */
```



Implementing a heap

- Convenient to implement as an array
  - Root: [1]; root children: [2,3]; their children: [4:7] …
  - Works because of binary completeness requirement – tree is full at all depths except leaves
- e.g., insertHeap algorithm
  - Step 1: `put item at end of array;`
    - `O(1)` complexity, unless array is filled up
  - Step 2 until done: `reheapify by array indexing;`
    - Have parent of `array[i]` at `array[i/2]`, ∀ `i>1`
    - `O(log n)` complexity to reheapify this way
- So complexity of `insertHeap` is `O(log n)` overall