CSE 101
Introduction to Data Structures and Algorithms
The Dictionary ADT and its Implementations

Recall that an ADT consists of a set of *states* together with a set of *operations*. A *dictionary* has as its state a set of ordered pairs, whose members are called *key* and *value*, respectively.

pair: (key, value)

state: $\{ (k_1, v_1), (k_2, v_2), (k_3, v_3), \ldots, (k_n, v_n) \}$

It is required that keys belong to a linearly ordered set, like numbers or strings. We also require that all keys in the dictionary be distinct, i.e. no two pairs have the same key. Values on the other hand can be any data type whatever, and may be repeated. We think of a key as something like an account number, that uniquely identifies the pair.

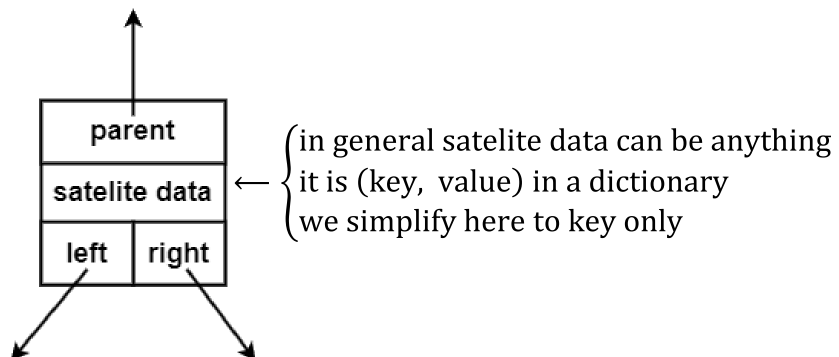
The main dictionary operations are

- Lookup(key): Return the unique value associated with *key*, or *nil* if *key* does not exist.
- Insert(key, value): Add a new key-value pair to the dictionary. (Pre: Lookup(key) = nil).
- Delete(key): Remove the pair with key *key* from the dictionary. (Pre: Lookup(key) $\neq$ nil).

There may also be a number of peripheral dictionary operations, depending on the particular implementation used.

Binary Search Trees

A *Binary Search Tree* (BST) is a data structure commonly used to implement a dictionary. It is a linked data structure built from node objects as pictured below.



In what follows, dictionary values will play no role, so we will simplify our discussion by assuming that the *satellite data* consists of a key only. Binary search trees have other uses besides building a dictionary. For instance, a BST forms the basis of a simple and efficient sorting algorithm. In these other uses, it is not necessary that keys be distinct, so we will suspend that requirement in most of what follows.

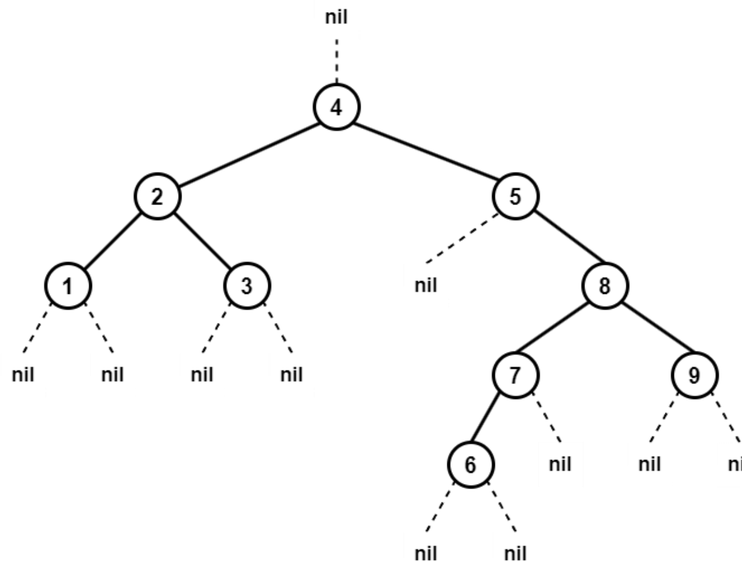
A binary search tree must satisfy the *Binary Search Tree Properties*. Let x and y denote nodes in a BST. We say that y is in the *left* (respectively *right*) *subtree* of x if and only if y is a descendant of the left (respectively right) child of x.

BST Properties:

(1) If y is in the left subtree of x, then y.key $\leq x$.key.

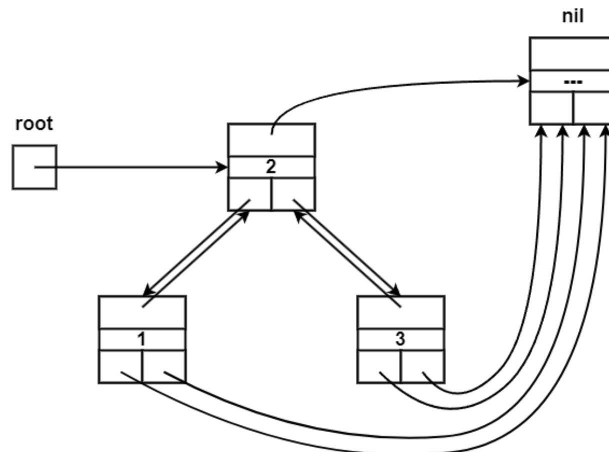
(2) If y is in the right subtree of x, then x.key $\leq y$.key.

Example 1

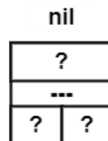


The value *nil* denotes the absence of a child, or parent. A node with no children is called a *leaf* (keys 1, 3, 6, 9 above), and an *internal node* is a non-leaf (keys 4, 2, 5, 8, 7). The root (key 4) necessarily has no parent. When implementing a BST, we can represent `nil` as `NULL` (in C), or as `nullptr` (in C++). However, a better option would be to create a node object called `nil`, and have the left, right or parent fields point to this dummy node, as necessary. The BST data structure must also maintain a pointer to the root node itself, much like a pointer to the head of a linked list.

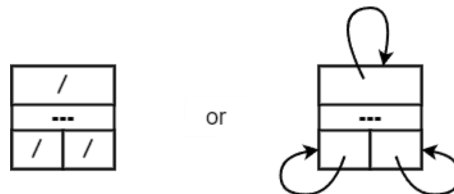
Example 2



What should we put in the left, right and parent fields of this `nil` node object?



There are two sensible choices for these values. We could place `NULL` or `nullptr`, denoted by `/` below, or we could place a pointer back to the `nil` object itself.



The value `nil` will be used in BST algorithms to terminate certain loops and recursions, so the question becomes, if something goes wrong, do we want a segmentation fault or do we want an infinite loop?

Tree Traversals

Amongst the algorithms that operate on a BST are the *tree traversals*, each of which process every node in the tree. The *in-order tree walk* processes the keys in sorted order.

```
InOrderTreeWalk(x)
1. if x != NIL
2.    InOrderTreeWalk(x.left)
3.    print(x.key)
4.    InOrderTreeWalk(x.right)
```

Output for Example 1: **1, 2, 3, 4, 5, 6, 7, 8, 9**

Thus, if we can build a BST, then we have essentially sorted its keys. The print operation on line 3 can be replaced with any operation on the satellite data stored in a node. To sort an array, we build a BST whose keys are the array elements, then do an in-order tree walk with the print operation replaced by a write into the original array.

The *pre-order tree walk* processes a node before both recursive calls. Thus we swap lines 2 and 3, then change the name of the function.

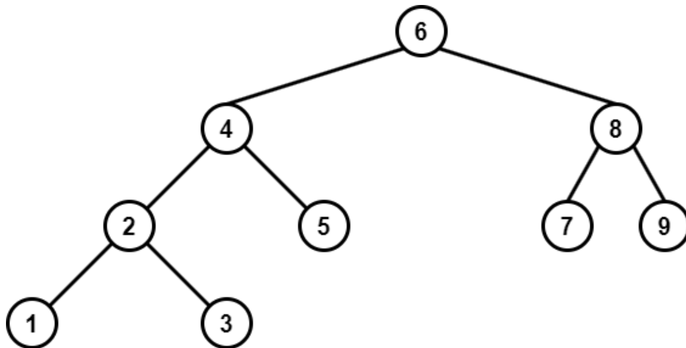
```
PreOrderTreeWalk(x)
1. if x != NIL
2.    print(x.key)
3.    PreOrderTreeWalk(x.left)
4.    PreOrderTreeWalk(x.right)
```

Output for Example 1: **4, 2, 1, 3, 5, 8, 7, 6, 9**

It is possible to recover the tree structure of a BST from the output of a pre-order tree walk. To do this, extract the first element in the output list (4 here) as the root, then split the remaining elements into two

sublists consisting of those keys less that the root (2, 1, 3), which constitute the left subtree, and those greater than the root (5, 8, 7, 6, 9), forming the right subtree. Next, recur on the two sublists, and halt the recursion if a sublist becomes empty. This procedure assures us that the pre-order tree walk uniquely identifies the structure of a BST. Consequently, if two BSTs have the same output for pre-order tree walk, then they have identical structures as trees. Notice that the same cannot be said for the in-order tree walk. In the following example, we get the same output as Example 1 from an in-order tree walk, but different ouput for a pre-order tree walk.

Example 3



Pre-order tree walk: **6, 4, 2, 1, 3, 5, 8, 7, 9**

The *post-order tree walk* processes a node after both recursive calls. We obtain this algorithm by swapping lines 3 and 4 from in-order tree walk, and changing the name of the function.

```
PostOrderTreeWalk(x)
1. if x != NIL
2.    PostOrderTreeWalk(x.left)
3.    PostOrderTreeWalk(x.right)
4.    print(x.key)
```

Output for Example 1: **1, 3, 2, 6, 7, 9, 8, 5, 4**
Output for Example 3: **1, 3, 2, 5, 4, 7, 9, 8, 6**

It is also possible to recover the tree structure of a BST from the output of a post-order tree walk. This is accomplished recursively, as in the pre-order case, but now using the fact that the root is the last element in the output list. Another use for the post-order tree walk is to free the heap memory used by a BST in languages like C and C++. To do this we replace the print statement on line 4, with a deallocation of the current node. This avoids memory leaks since we deallocate a node only after we have deallocated both its left and right subtrees.

There are 3 other tree traversals obtained by swapping the two recursive calls in the in-order, pre-order and post-order tree walks. There are no standard names for these algorithms, but we could call them the *reverse-order*, *reverse-pre-order* and *reverse-post-order* tree walks, respectively. We leave it as an exercise to write their pseudo-code.

All tree traversals have a runtime of $\Theta(n)$, where n denotes the number of nodes in the tree, since every node is visited exactly once.

Queries

The BST can be an efficient data structure for searching. The `TreeSearch` algorithm below searches the subtree rooted at x for a node containing the key k. It recurs down a line of ancestry starting at x, and returns (a pointer to) the first node encountered bearing the key k, or it returns `nil` if no such key is found.

```
TreeSearch(x, k)
1. if x == NIL or k == x.key
2.    return x
3. else if k < x.key
4.    return TreeSearch(x.left, k)
5. else // k > x.key
6.    return TreeSearch(x.right, k)
```

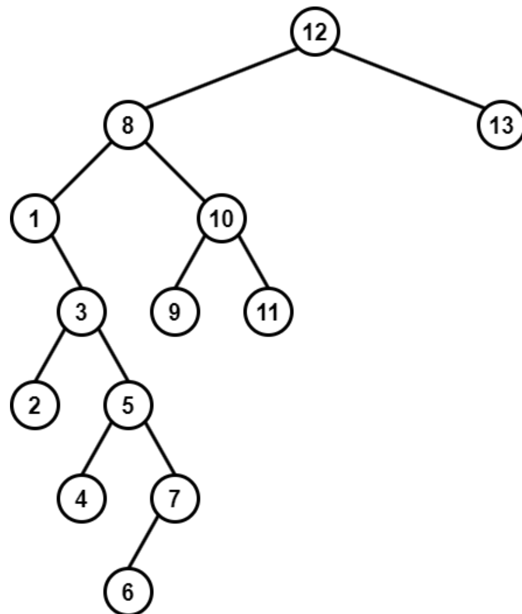
The algorithms `TreeMinimum` (respectively `TreeMaximum`) search the (non-empty) subtree rooted at x for the node with the smallest (respectively largest) key. These algorithms are themselves used as subroutines for other queries.

```
TreeMinimum(x) Pre: x != NIL
1. while x.left != NIL
2.    x = x.left
3. return x

TreeMaximum(x) Pre: x != NIL
1. while x.right != NIL
2.    x = x.right
3. return x
```

We define the successor of a node x to be the next node after x to be processed in an in-order tree walk.

Example 4



In this example, the in-order tree walk is: **1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13**. When searching for successor of a node x, we have two cases to consider.

Case 1: x has a right child.
In this case, the successor of x is the leftmost descendant of that right child, i.e. the leftmost node in $x's$ right subtree. For instance, in the above example, the successor of 1 is 2, and the successor of 3 is 4.

Case 2: x has no right child.
In this case, the successor of x is the lowest ancestor of x whose left child is also an ancestor of x (keeping in mind that x is considered it's own ancestor). In other words, we climb up a line of ancestry from x until we make the first right turn. In the above example, the sucessor of 7 is 8, and the successor of 11 is 12.

These cases form the basis of the algorithm `TreeSuccessor`. Note that if a node has no successor (like 13 in the example), then the algorithm returns `nil`.

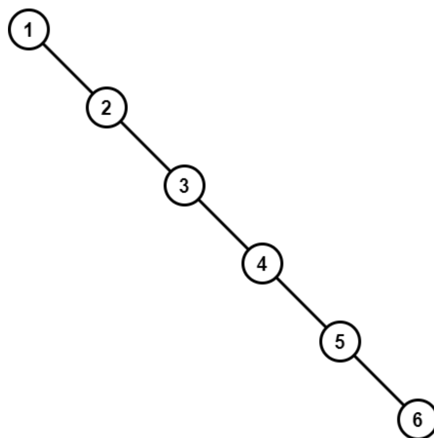
```
TreeSuccessor(x)
1. if x.right != NIL                        // case 1
2.    return TreeMinimum(x.right)
3. y = x.parent                             // case 2
4. while y != NIL and x == y.right
5.    x = y
6.    y = y.parent
7. return y
```

Exercise

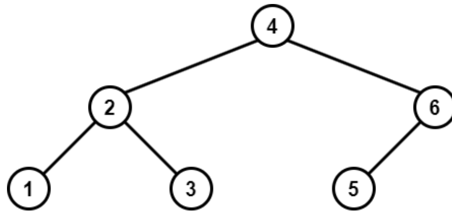
- Define the *predecessor* of a node x.
- Write pseudo-code for an algorithm called `TreePredecessor` that given a node x, returns the predecessor of x, or returns `nil` if it has no predecessor.

We define the *depth* of a node x to be the distance (number of edges) from x to the root. The *height* of a tree T, denoted $\text{height}(T)$, is the depth of its deepest leaf, and the height of a node x, denoted $\text{height}(x)$, is the height of the subtree rooted at x. The maximum length of any line of ancestry in a tree T is therefore $\text{height}(T)$. We see that the worst case runtime of each of the 5 query algorithms (`TreeSearch`, `TreeMin`, `TreeMax`, `TreeSuccessor`, `TreePredecessor`) is thus $\Theta(\text{height}(T))$, since each query either iterates, or recurs, up or down a line of ancestry.

Given the 6 keys $\{1, 2, 3, 4, 5, 6\}$, the *worst* possible BST for queries would be



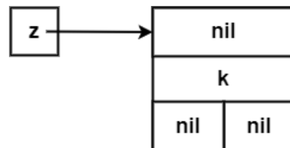
which has $\text{height}(T) = 5$. Any other tree in which every node has just one child would be equally bad. The *best* possible tree on the same set of keys would be



in which $\text{height}(T) = 2$. The following fact is not difficult to prove: if a binary tree T contains n nodes, then $\lfloor \log_2(n) \rfloor \le \text{height}(T) \le n - 1$. For the sake of the runtime of query algorithms, we are motivated to build our BSTs in such a way that the height is as small as possible. Any binary tree for which $\text{height}(T) = \Theta(\log(n))$ is said to be a *balanced tree*. We will discuss methods by which our BSTs can be arranged to be balanced. At the moment however, we don't even know how to build a BST, balanced or not. This is remedied by the next group of algorithms.

Insertion and Deletion

To insert a new node z into an existing BST T, while maintaining the BST properties, we first initialize the fields of z, then determine where to place z in T.



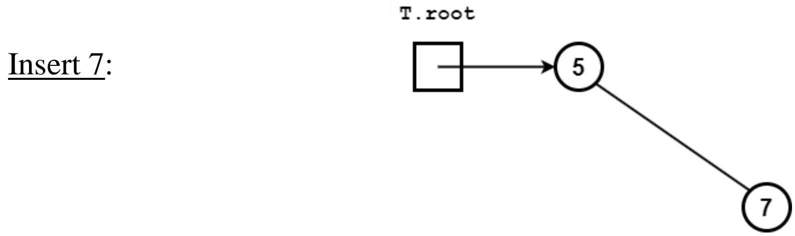
- Set $z.\text{key} = k$
- Set $z.\text{left} = z.\text{right} = z.\text{parent} = \text{nil}$
- If T is non-empty, then find an existing node to adopt z as left or right child, as appropriate. If T is empty, then let z be the root of T.

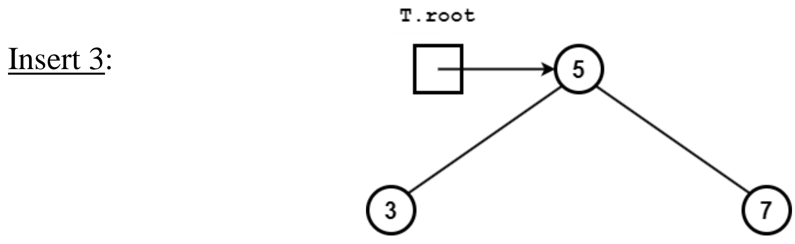
To accomplish the last step, we do a search for the key k within T and find the `nil` child position where key k would reside, were it in T.

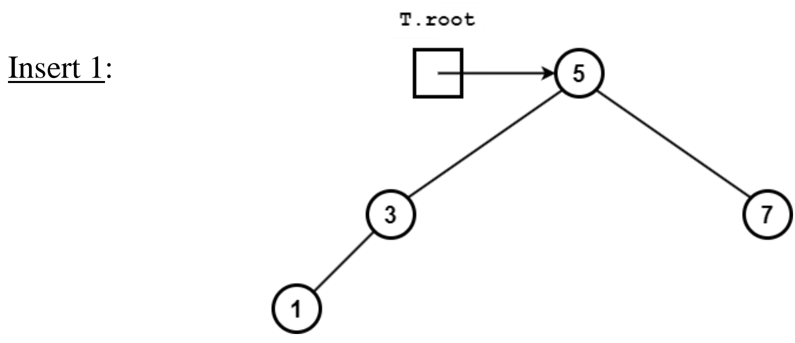
```
TreeInsert(T, z)
1.  y = NIL
2.  x = T.root
3.  while x != NIL
4.     y = x
5.     if z.key < x.key
6.        x = x.left
7.     else
8.        x = x.right
9.  z.parent = y
10. if y == NIL              // T was empty
11.    T.root = z
12. else if z.key < y.key
13.    y.left = z
14. else
15.    y.right = z
```

Observe that if z.key is already in T, say at node x, then the new node z will end up in $x's$ right subtree. The sorting algorithm we alluded to earlier is obtained by steping through an array, inserting its elements (as keys) into an initially empty BST as we go, then doing an in-order tree walk on the resulting BST, writing keys back into the array. The observation above shows that repeated keys in the input array, will appear in the same order in the output array. Such a sorting algorithm is said to be *stable*. We leave it as an exercise to write pseudo-code for this algorithm.

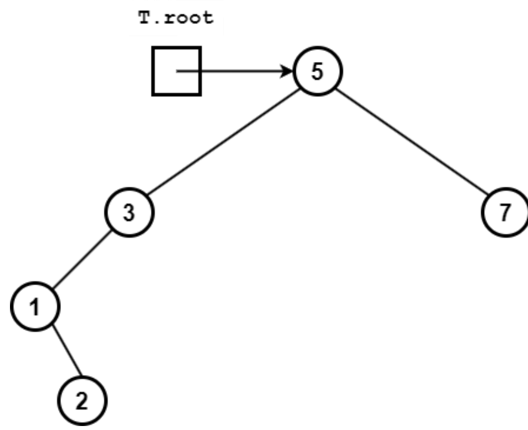
Example 5
We insert the keys: **5, 7, 3, 1, 2, 6, 4, 9, 8** (in order) into an initially empty BST.



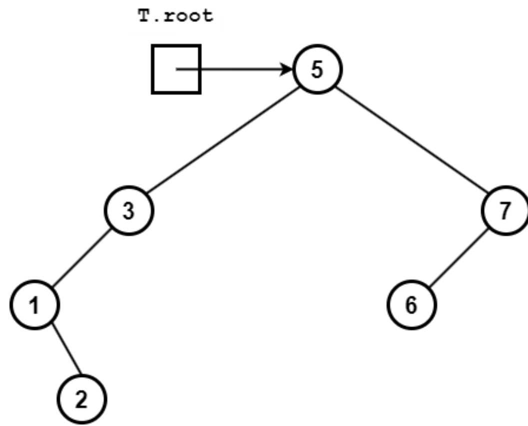




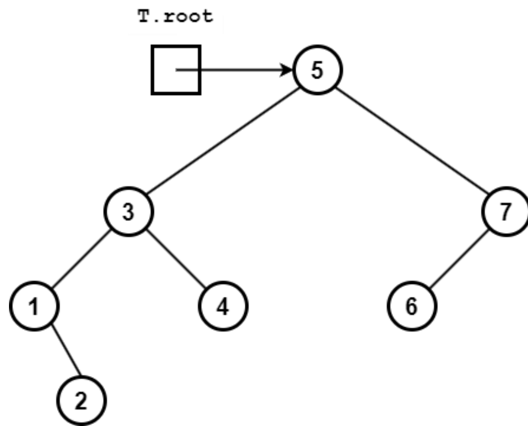
Insert 2:



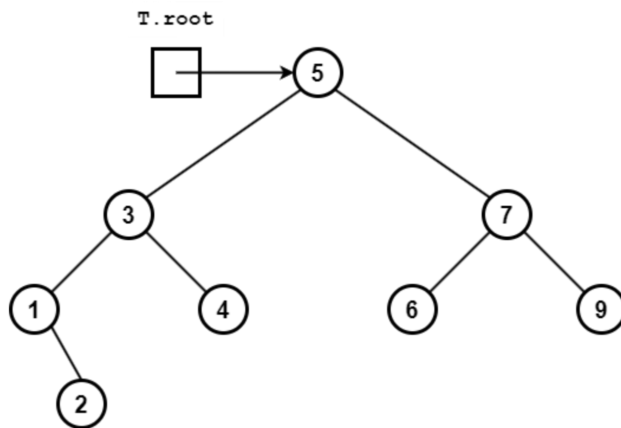
Insert 6:



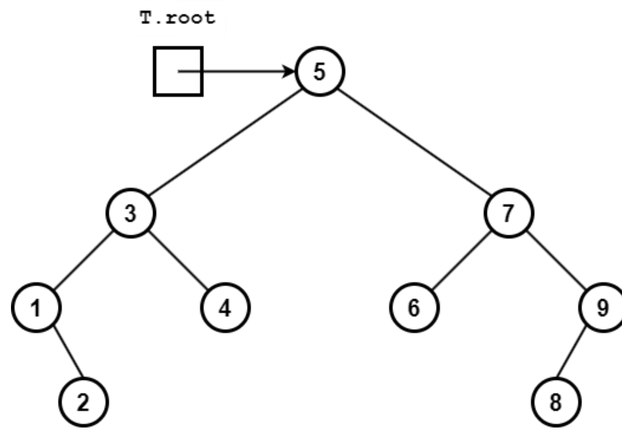
Insert 4:



Insert 9:



Insert 8:



The worst case runtime of `TreeInsert` $\Theta(\text{height}(T))$, since like `TreeSearch`, it walks down a single line of ansestry. Note also that many different insertion orders on the same set of keys will produce the same tree structure. In particular, the pre-order tree walk (**5, 3, 1, 2, 4, 7, 6, 9, 8**) can be used to create a copy of the above tree.

More to come