# 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) ≠ nil).

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

## Hash Tables (CLRS 11.1-11.4)

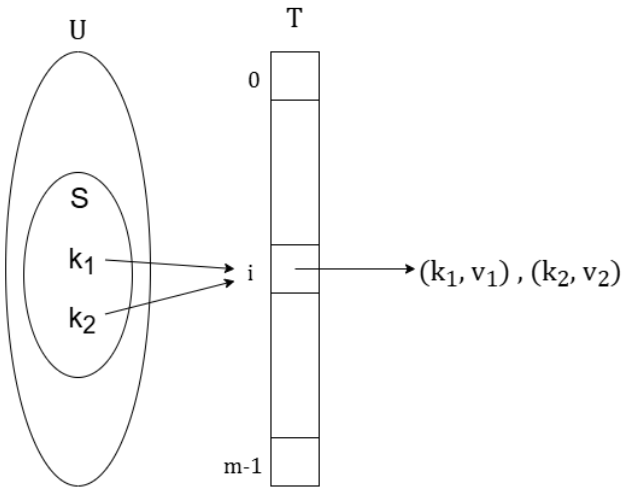
If the keys $k$ happen to be integers in the range 0 to $N - 1$, we can allocate an array of length $N$, then store the value $v$ in index $k$. This arrangement is called a *direct-address table*. Accessing a pair $(k, v)$ then has constant cost. The problem with direct addressing is obvious: if $N$ is too large, allocating an array of length $N$ may be impractical, or even impossible. Think of an application in which keys are credit card account numbers, and values are account balances. Such applications often have account numbers consisting of 16 or more decimal digits, which makes the universe $U$ of *all possible keys* very large indeed. Furthermore, the set $S$ of *keys in use* may be so small relative to $U$, that most of the space allocated for the array is wasted. In the case of credit card numbers with 16 decimal digits, the universe of possible keys has on the order of one quadrillion numbers. That's a very long array, and even if everyone on the planet has an account, over 99.999% of the space is unused. For this reason, direct-address tables are rarely used to build a Dictionary.

A *hash table* is an array $T$ that is used to implement a Dictionary, but which requires much less storage than a direct address table. Specifically, the storage requirements for a hash table can be reduced to $\Theta(|S|)$, while maintaining the benefit that the dictionary operations run in (average case) constant time $\Theta(1)$. To do this we use a *hash function* $h()$ to compute the index, or *slot*, $h(k)$ where a given pair $(k, v)$ will be stored. A suitable hash function must map the universe $U$ of possible keys to the set $\{0, 1, \ldots, m - 1\}$ of array indices, where $m = \text{length}(T)$, and thus $h: U \to \{0, 1, \ldots, m - 1\}$. The point of the hash function is to reduce the range of array indices that must be handled. Instead of $|U|$ indices, we need handle only $m$ indices, and storage requirements are thereby reduced.

There is of course one problem: two keys may hash to the same slot, i.e. $h(k_1) = h(k_2)$. We call this a *collision*. Fortunately there are effective techniques for resolving the conflicts created by collisions. It would be ideal to avoid collisions altogether. We might try to achieve this goal through our choice of hash function. By making $h(k)$ appear to be random, we can avoid collisions by making them improbable. The very term "hash", which evokes images of random mixing and chopping, captures the spirit of this approach. Of course, a hash function must be deterministic in the sense that a given input $k$ always produces the same output $h(k)$. Since $|U| > m$, and in general $|U|$ is *much larger* than $m$, there must be at least two keys that have the same hash value, and therefore completely avoiding collisions is impossible. Thus, while a well-designed random looking hash function can minimize the number of collisions, we still need a method for resolving collisions that do occur. Two such methods will be discussed here.

**Chaining**

The most common, and perhaps simplest method for collision resolution is called *chaining*. If two keys $k_1$ and $k_2$ hash to the same slot, so $h(k_1) = h(k_2) = i$, then both keys, and their associated values $v_1$ and $v_2$, are placed in a linked list headed by array element $T[i]$.




Therefore $T$ is an array of linked lists. More precisely, $T[i]$ is a reference to the head of a linked list storing all pairs that hash to slot $i$. If there are no such pairs, then list $T[i]$ is empty. The basic Dictionary operations are easy to implement in this scenario. To insert a pair $(k, v)$ into the Dictionary, create a new node storing the pair, then insert that node at the head of the linked list $T[h(k)]$. To lookup key $k$, do a linear search of the list $T[h(k)]$, and return the corresponding value $v$ if found, or a special value (say nil) if not found. To delete the pair with key $k$, delete the corresponding node from the list $T[h(k)]$.

We define the *load factor* $\alpha$ of $T$ to be $\alpha = n/m$, where $n = |S| =$ #keys in use, and $m = \text{length}(T)$. In other words, $\alpha$ is the average list length. At this point the only question is what to choose as our hash function $h(k)$. A good hash function should satisfy (at least approximately) the property of *simple uniform hashing*: each key is equally likely to hash to any of the $m$ slots of $T$, independently of the slot to which any other key has hashed. If such a function can be engineered, then the lengths of the linked lists will have expected value $\alpha$, and the three ADT operations described above will all run in (average case) $\Theta(1)$ time.

Let $N$ be a large positive integer. If keys are integers uniformily distributed over the set $\{0, \ldots, N-1\}$, it can be shown that the function $h(k) = k \bmod m$ (the remainder of $k$ upon division by $m$) satisfies simple uniform hashing. It is more common however for keys to be strings, or data objects of some other kind. Let us assume that $k$ is a string. We can then construct $h(k)$ in two steps. First convert $k$ into an integer

in the range 0 to $N-1$, which we refer to as the string's *hash code*, denoted by $\text{hash}(k)$. We try to do this in such a way as to make each integer $\{0, \dots, N-1\}$ equally likely. Second, we take the remainder of $\text{hash}(k)$ upon division by $m = \text{length}(T)$. The resulting hash function is $h(k) = \text{hash}(k) \bmod m$.

How then should we construct the hash code for the string $k$? This is really a programming question which depends on how strings are encoded and what int types are available. In the C language, a hash code will be an `unsigned long` (i.e. `uint64_t` in the `stdint.h` library). A string $k$ in C is an array of type `char`, each of which is an 8-bit `unsigned int`. Thus $k$ can be though of as a bit sequence of length $8 \cdot \text{length}(k)$. The hash code for $k$ should depend on every one of these bits. See /Examples/HashCode/HashTest.c on the class webpage for one way to do this.

**Open Addressing**

This method is an alternative to chaining in which all keys are stored directly in the table $T$. Therefore the load factor must satisfy

$$\alpha = \frac{n}{m} = \frac{\#\text{ keys in use}}{\#\text{ slots in table}} \leq 1$$

Necessarily $n \leq m$, and hence the number of pairs in the Dictionary can never exceed the length of $T$. To insert a pair $(k, v)$ into the dictionary, we first compute a sequence of slots into which we try to perform the insertion. If a given slot is occupied, we try the next slot in the sequence. Each try is called a *probe* and the sequence is called the *probe sequence* for the key $k$. Obviously, a probe sequence must probe every slot in the table. Thus the hash function is of the form

$$h: U \times \{0, \dots, m-1\} \to \{0, \dots, m-1\}$$

and the probe sequence for $k$ is

$$h(k,0), h(k,1), h(k,2), \dots\dots, h(k, m-1)$$

which necessarily is a permutation of the table slots $\{0, \dots, m-1\}$.

Several techniques are used to construct hash function for open addressing. One of the best is called *double hashing*. (Other techniques are discussed in the text CLRS.) In double hashing, $h$ takes the form

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

where $0 \leq i \leq m-1$ and $h_1, h_2: U \to \{0, \dots, m-1\}$ are auxilary hash functions. In this case the probe sequence becomes

$$\begin{array}{ll} h_1(k) & \bmod m \\ h_1(k) + h_2(k) & \bmod m \\ h_1(k) + 2h_2(k) & \bmod m \\ h_1(k) + 3h_2(k) & \bmod m \\ \vdots & \\ \vdots & \\ h_1(k) + (m-1)h_2(k) & \bmod m \end{array}$$

Note that for all $k \in U$, $h_2(k)$ must be non-zero, and relatively prime to $m$, otherwise the probe sequence will not include all slots in the table, as is required.

**Example 1** Let $k$ be an integer, $m = 11$, $h_1(k) = k \bmod 11$ and $h_2(k) = 1 + k \bmod 10$. Then

$$h(k,i) = \big(k \bmod 11 \; + \; i(1 + k \bmod 10)\big) \bmod 11$$

Since the remainder of $k$ upon division by 10 lies in the set $\{0, \ldots, 9\}$, we have $1 \leq 1 + k \bmod 10 \leq 10$, whence $h_2(k)$ is non-zero and relatively prime to 11, as desired. Calculating the probe sequences for $k = 24$ and $k = 25$ below gives

$$k = 24: \quad 2, 7, 1, 6, 0, 5, 10, 4, 9, 3, 8$$
$$k = 25: \quad 3, 9, 4, 10, 5, 0, 6, 1, 7, 2, 8$$

This example generalizes to $h(k,i) = \Big(k \bmod p + i\big(1 + k \bmod (p-1)\big)\Big) \bmod p$, where $m = p$ is any prime.

**Example 2** Another hash function that works well in case $k$ an integer, is to let $m$ be a power of 2, $h_1(k) = k \bmod m$, and $h_2(k) = 2\big(k \bmod (m/2)\big) + 1$, so that

$$h(k,i) = \left(k \bmod m + i\left(2\left(k \bmod \frac{m}{2}\right) + 1\right)\right) \bmod m$$

Observe that necessarily $h_2(k)$ is non-zero and relatively prime to $m$, since $h_2(k)$ is odd and $m$ is even. Thus $h(k,i)$ is a valid hash function for open addressing. We specialize this example to the case where $m = 8 = 2^3$, and again compute the probe sequences for $k = 24$ and $k = 25$, respectively. We get

$$h(k,i) = \big(k \bmod 8 + i(2(k \bmod 4) + 1)\big) \bmod 8$$

and

$$k = 24: \quad 0, 1, 2, 3, 4, 5, 6, 7$$
$$k = 25: \quad 1, 4, 7, 2, 5, 0, 3, 6$$

Let us see what happens when we insert the keys: 24, 25, 33, 20, 27, 41, 13, 52 (in order) into an initially empty hash table of length 8, using open addressing and the above hash function.

| key | probe sequence | slot in table |
|---|---|---|
| 24 | **0**, 1, 2, 3, 4, 5, 6, 7 | 0 |
| 25 | **1**, 4, 7, 2, 5, 0, 3, 6 | 1 |
| 33 | ~~1~~, **4**, 7, 2, 5, 0, 3, 6 | 4 |
| 20 | ~~4~~, **5**, 6, 7, 0, 1, 2, 3 | 5 |
| 27 | **3**, 2, 1, 0, 7, 6, 5, 4 | 3 |
| 41 | ~~1, 4~~, **7**, 2, 5, 0, 3, 6 | 7 |
| 13 | ~~5, 0, 3~~, **6**, 1, 4, 7, 2 | 6 |
| 52 | ~~4, 5, 6, 7, 0, 1~~, **2**, 3 | 2 |

Ignoring the associated values and showing keys only, the resulting state of the hash table is

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $T$ | 24 | 25 | 52 | 27 | 33 | 20 | 13 | 41 |

Deletions cause some complexity in an open address table. The problem is that an empty slot should stop the insertion process, but should not stop a search, since a deletion could create a "hole" in the probe sequence. The problem is solved by defining two special values which we call EMPTY and DELETED, and initialize all table slots to EMPTY. Insertion() halts when it reaches either value, while Lookup() halts for EMPTY but not for DELETED.

The following pseudocode for Lookup() returns the index where key $k$ is found, or returns EMPTY if $k$ is not found.

```
Lookup(T,k)
1. for i=0 to length(T)-1
2.    j = h(k,i)
3.    if T[j]==k
4.       return j
5.    else if T[j]==EMPTY
6.       return EMPTY
7. return EMPTY
```

Function Insert() inserts key $k$ into the first empty or deleted slot, with the precondition that $k$ is not already in the table (not tested in this pseudocode). Inserting into a full table is considered an error condition.

```
Insert(T,k)   pre: Lookup(T,k)==EMPTY
1. for i=0 to length(T)-1
2.    j = h(k,i)
3.    if T[j]==EMPTY or T[j]==DELETED
4.       T[j] = k
5. error('table full')
```

Function Delete() first finds the slot containing key $k$ by calling Lookup(), then replaces it with DELETED. It has the precondition that $k$ is in the table (tested in this pseudocode).

```
Delete(T,k)   pre: Lookup(T,k)!=EMPTY
1. j = Lookup(T,k)
2. if j==EMPTY
3.    error('deleting non-existent key')
4. T[j] = DELETED
```

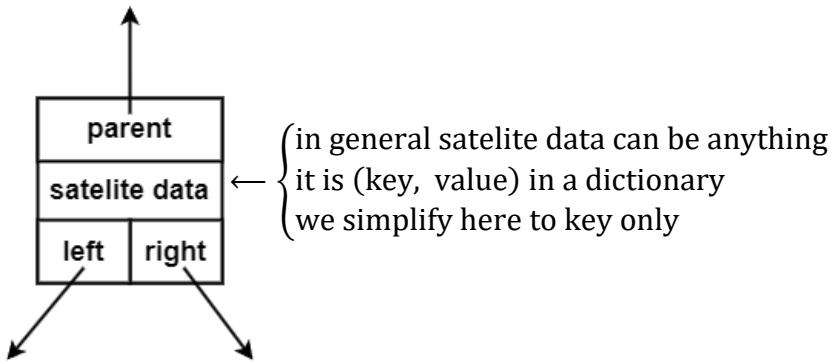
One issue remains with an open address table. Unlike with chaining, the table can fill up. In fact we have the constraint that $n = \#\text{keys in use} \leq \text{length}(T) = m$, so the load factor $\alpha$ satisfies $\alpha = n/m \leq 1$. The only way around this is to periodically resize the array. How to do this is another programming question, and depends on the implementation language. See C/FileIO, C/Queue/Array and C/Stack/Array in the directory /Examples on the class webpage for illustrations on expanding arrays in C.

It can be shown (CLRS Theorem 11.6) that the expected number of probes performed by Lookup() in an open address table is bounded above by $1/(1-\alpha)$. This motivates us to try to keep the load factor $\alpha$

bounded away from 1 by expanding the table well before it becomes full. This was illustrated in the previous example where the last few insertions needed to step further and further into the probe sequence to find an empty slot. A commonly accepted bound for $\alpha$ is $2/3$. The above upper bound is then 3, which effectively makes all the Dictionary operations run in (average case) constant time $\Theta(1)$.

## **Binary Search Trees** (CLRS 12.1-12.3)

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



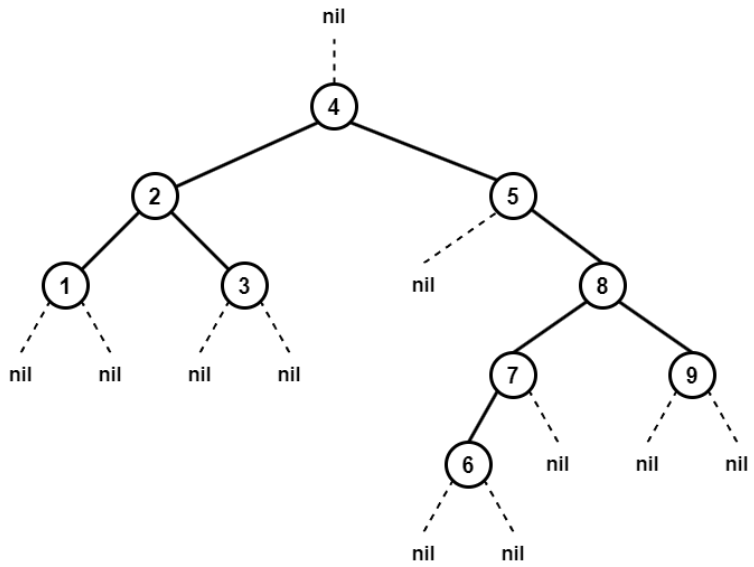

In what follows, dictionary values will play no role, so we will simplify our discussion by assuming that the *satellite data* consists of a key only. Binary search trees have other uses besides building a dictionary. For instance, a BST forms the basis of a simple and efficient sorting algorithm. In 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:**

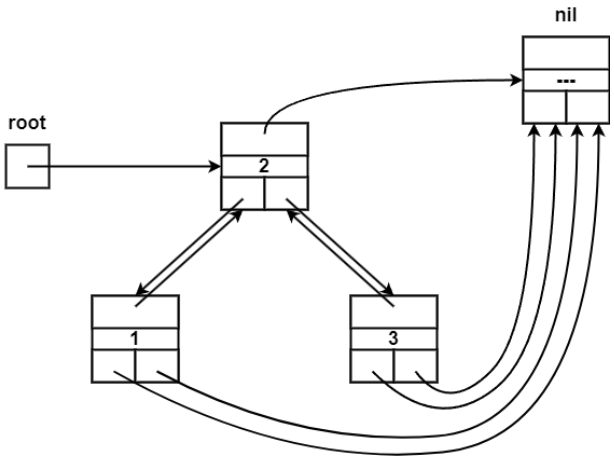
(1) If $y$ is in the left subtree of $x$, then $y$.key $\leq x$.key.
(2) If $y$ is in the right subtree of $x$, then $x$.key $\leq y$.key.

**Example 3**

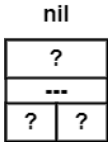


The value *nil* denotes the absence of a child, or parent. A node with no children is called a *leaf* (keys 1, 3, 6, 9 above), and an *internal node* is a non-leaf (keys 4, 2, 5, 8, 7). The root (key 4) necessarily has no parent. When implementing a BST, we can represent `nil` as `NULL` (in C), or as `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.

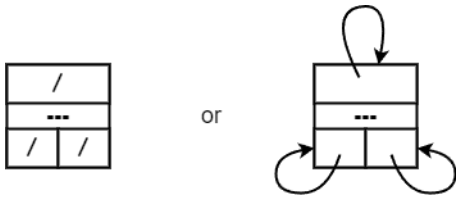
**Example 4**




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.

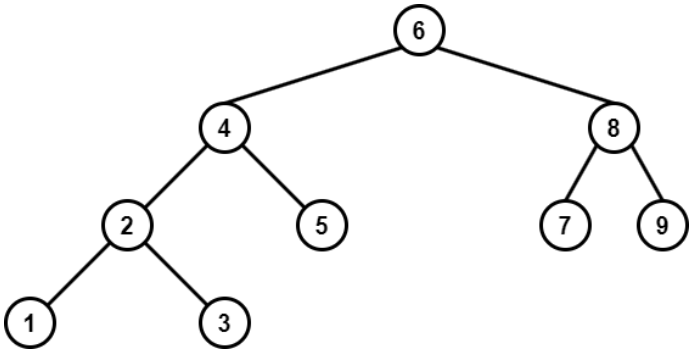
The *pre-order tree walk* processes a node before both recursive calls. Thus we swap lines 2 and 3, then change the name of the function.

```
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 5**




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)
```

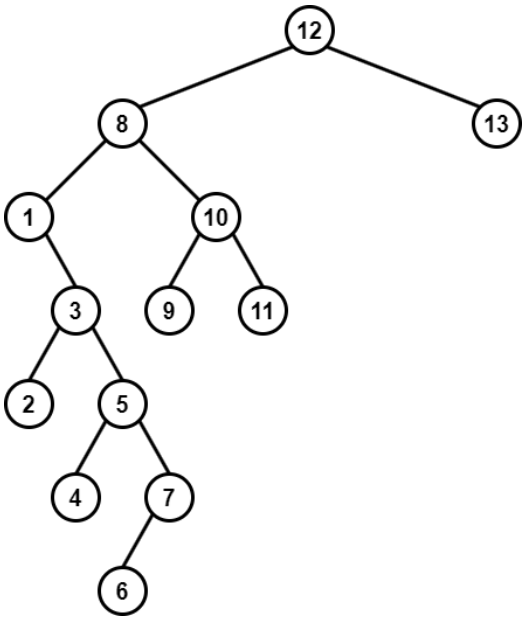
The algorithms `TreeMinimum` (respectively `TreeMaximum`) search the (non-empty) subtree rooted at $x$ for the node with the smallest (respectively largest) key. These algorithms are themselves used as subroutines for other queries.

```
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 6**




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
```

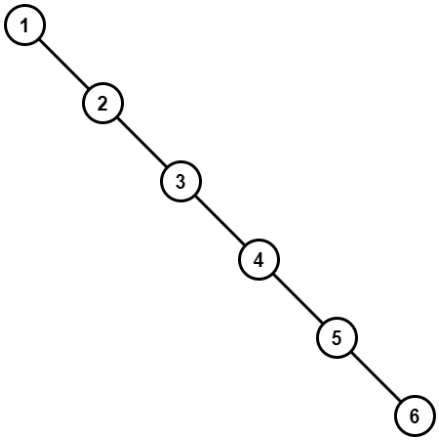
**Exercise**

- 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.

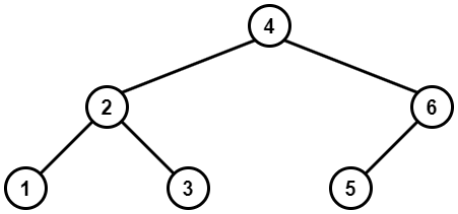
We define the *depth* of a node $x$ to be the distance (number of edges) from $x$ to the root. The *height* of a tree $T$, denoted $\text{height}(T)$, is the depth of its deepest leaf, and the height of a node $x$, denoted $\text{height}(x)$,

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



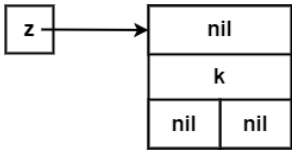

which has $\text{height}(T) = 5$. Any other tree in which every node has just one child would be equally bad. The *best* possible tree on the same set of keys would be




in which $\text{height}(T) = 2$. The following fact is not difficult to prove: if a binary tree $T$ contains $n$ nodes, then $\lfloor \log_2(n) \rfloor \leq \text{height}(T) \leq n - 1$. 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 $T$.

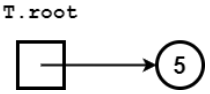
```
TreeInsert(T, z)
1.  y = NIL
2.  x = T.root
3.  while x != NIL
4.     y = x
5.     if z.key < x.key
6.        x = x.left
7.     else
8.        x = x.right
9.  z.parent = y
10. if y == NIL            // T was empty
11.    T.root = z
12. else if z.key < y.key
13.    y.left = z
14. else
15.    y.right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

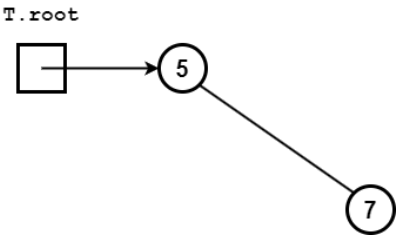
**Example 7**
We insert the keys: **5, 7, 3, 1, 2, 6, 4, 9, 8** (in order) into an initially empty BST.

Empty state:

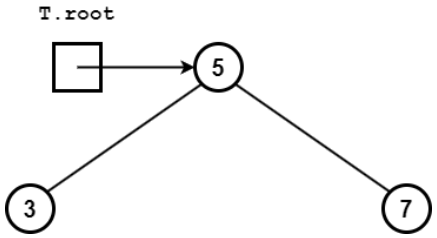

Insert 5:



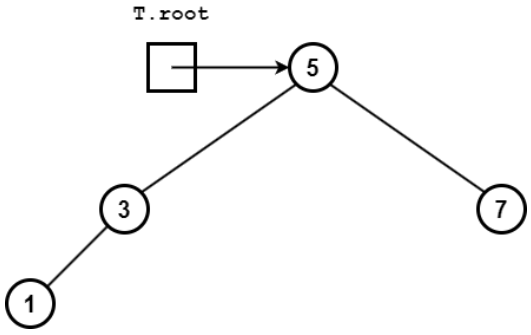
Insert 7:



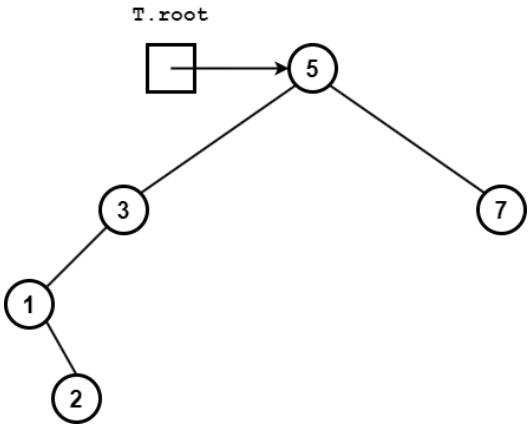

Insert 3:



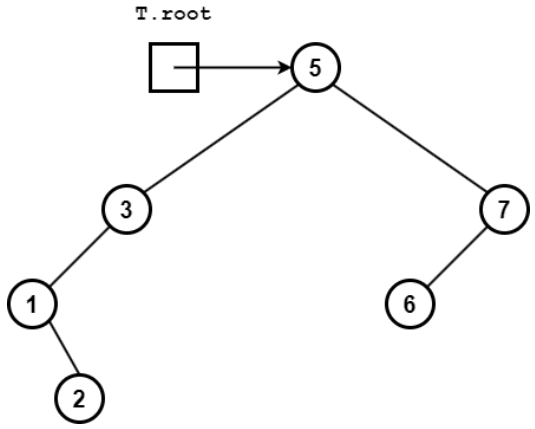

Insert 1:



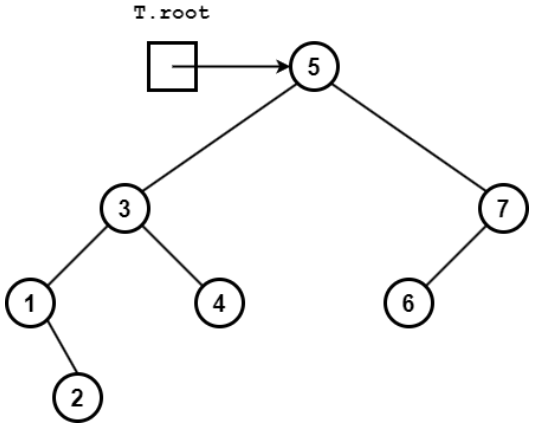

Insert 2:



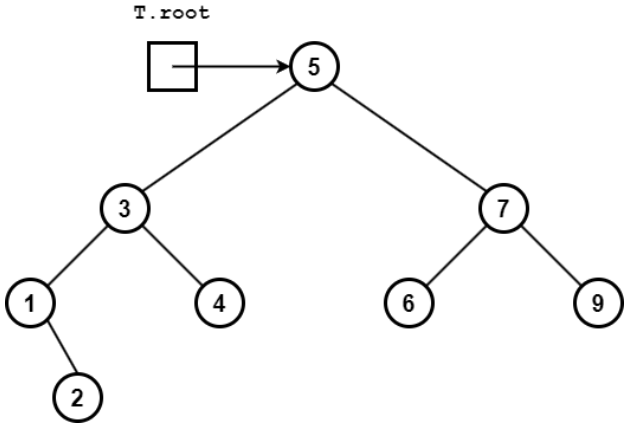
Insert 6:



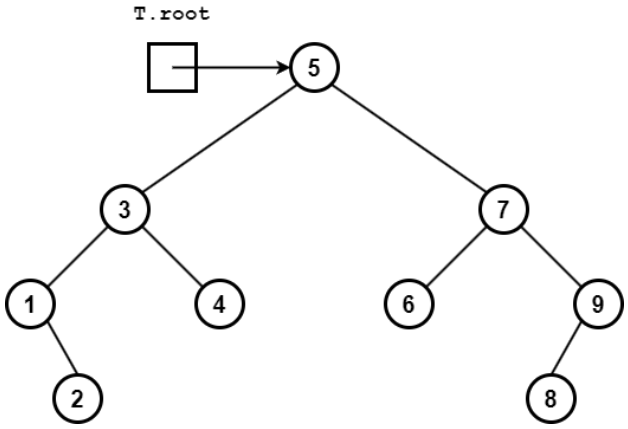

Insert 4:




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