```python
# Binary trees: each node has no more than two child nodes

# BST: ordering property
# -> left is less than node
# -> right is greater than node

# Balanced vs Unbalanced
# Traversal:
# 1. pre-order:
# -> root, left, right
# 2. in-order:
# -> left, root, right
# 3. post-order:
# -> left, right, root

# BST IMPLEMENTATION

class Node:
    def __init__(self, val):
        self.l = None
        self.r = None
        self.v = val

class Tree:
    def __init__(self):
        self.root = None

    def getRoot(self):
        return self.root

    def add(self, val):
        if(self.root == None):
            self.root = Node(val)
        else:
            self._add(val, self.root)

    def _add(self, val, node):
        if(val < node.v):
            if(node.l != None):
                self._add(val, node.l)
            else:
                node.l = Node(val)
        else:
            if(node.r != None):
                self._add(val, node.r)
            else:
                node.r = Node(val)

    def find(self, val):
        if(self.root != None):
            return self._find(val, self.root)
        else:
            return None
```

```python
# HackerRank: CTCI
# Tree: is this a BST

def check_binary_search_tree_(root):
    arr = []
    count = 0
    arr = inorderTraversal(root, arr)
    if ((sorted(arr)) == arr) and (len(set(arr)) == len(arr)):
        return True
    else:
        return False

def inorderTraversal(root, arr):
    if root != None:
        inorderTraversal(root.left, arr)
        arr.append(root.data)
        inorderTraversal(root.right, arr)
    return arr


# Depth First Search

def dfs(graph, start):
    visited, stack = set(), [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            stack.extend(graph[vertex] - visited)
    return visited


def bfs(graph, start):
    visited, queue = set(), [start]
    while queue:
        vertex = queue.pop(0)
        if vertex not in visited:
            visited.add(vertex)
            queue.extend(graph[vertex] - visited)
    return visited

bfs(graph, 'A') # {'B', 'C', 'A', 'F', 'D', 'E'}


# Generator example

def square(nums):
    for i in nums:
        yield (i * i)

my_nums = square([1,4,2,5])

# OR: using comprehension

# comprehension
nums_comprehended = [x*x for x in [1,2,3,4]]
# becomes: generator
nums_generator = (x*x for x in [1,2,3,4])

print my_nums
print nums_comprehended
print nums_generator

print list(nums_generator)

```

```python

# Iterators per Data Type

# List
lst = [1, 2, 3, 4]:

for i in lst:
    pass

# String
string = "python"

for c in string:
    pass

# Tuples
tup = (1,2,3,4,5,6,7,8,9,10)

for i in tup:
    pass

# Dict
dictionary = {'name': 'Helen', 'age': '21', 'job': 'boss'}

for key, val in dictionary.iteritems():

for k in dictionary:
    pass

# Set
my_set = {10,20,30,40,50,20}
for i in my_set:
    pass

# File
for line in open("a.txt"):
    pass

# Sorting

# http://danishmujeeb.com/blog/2014/01/basic-sorting-algorithms-implemented-in-python/

# 1. Bubble sort

# It’s basic idea is to bubble up the largest(or smallest), then the 2nd largest
# and the the 3rd and so on to the end of the list. Each bubble up takes a full
# sweep through the list.

def bubble_sort(items):
        for i in range(len(items)):
                for j in range(len(items)-1-i):
                        if items[j] &gt; items[j+1]:
                                items[j], items[j+1] = items[j+1], items[j]
```

```
# 2. Insertion Sort

# Insertion sort works by taking elements from the unsorted list and inserting them
# at the right place in a new sorted list. The sorted list is empty in the beginning.
# Since the total number of elements in the new and old list stays the same, we can
# use the same list to represent the sorted and the unsorted sections.

def insertion_sort(items):
        for i in range(1, len(items)):
                j = i
                while j &gt; 0 and items[j] &lt; items[j-1]:
                        items[j], items[j-1] = items[j-1], items[j]
                        j -= 1

# 3. Merge Sort

# Merge sort works by subdividing the the list into two sub-lists, sorting them using
# Merge sort and then merging them back up. As the recursive call is made to subdivide
# each list into a sublist, they will eventually reach the size of 1, which is
# technically a sorted list.

def merge_sort(items):
        """ Implementation of mergesort """
        if len(items) &gt; 1:

                mid = len(items) / 2        # Determine the midpoint and split
                left = items[0:mid]
                right = items[mid:]

                merge_sort(left)            # Sort left list in-place
                merge_sort(right)           # Sort right list in-place

                l, r = 0, 0
                for i in range(len(items)):     # Merging the left and right list

                        lval = left[l] if l &lt; len(left) else None
                        rval = right[r] if r &lt; len(right) else None

                        if (lval and rval and lval &lt; rval) or rval is None:
                                items[i] = lval
                                l += 1
                        elif (lval and rval and lval &gt;= rval) or lval is None:
                                items[i] = rval
                                r += 1
                        else:
                                raise Exception('Could not merge, sub arrays \
                                sizes do not match the main array')
```

```python
# 4. Quick Sort

# Quick sort works by first selecting a pivot element from the list. It then creates
# wo lists, one containing elements less than the pivot and the other containing
# elements higher than the pivot. It then sorts the two lists and join them with the
# ivot in between. Just like the Merge sort, when the lists are subdivided to lists
# of size 1, they are considered as already sorted

def quick_sort(items):
        """ Implementation of quick sort """
        if len(items) &gt; 1:
                pivot_index = len(items) / 2
                smaller_items = []
                larger_items = []

                for i, val in enumerate(items):
                        if i != pivot_index:
                                if val &lt; items[pivot_index]:
                                        smaller_items.append(val)
                                else:
                                        larger_items.append(val)

                quick_sort(smaller_items)
                quick_sort(larger_items)
                items[:] = smaller_items + [items[pivot_index]] + larger_items

# 5. Heap Sort

# This implementation uses the built in heap data structures in Python. To truly
# understand haepsort, one must implement the heapify() function themselves. This
# is certainly one obvious area of improvement in this implementation.

import heapq

def heap_sort(items):
        """ Implementation of heap sort """
        heapq.heapify(items)
        items[:] = [heapq.heappop(items) for i in range(len(items))]

# Stack implementation

class Stack:
     def __init__(self):
         self.items = []

     def isEmpty(self):
         return self.items == []

     def push(self, item):
         self.items.append(item)

     def pop(self):
         return self.items.pop()

     def peek(self):
         return self.items[len(self.items)-1]

     def size(self):
         return len(self.items)
```

```python

# Queue Implementation

class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
# Tree Traversals

# 1. pre-order

# In a preorder traversal, we visit the root node first,
# then recursively do a preorder traversal of the left
# subtree, followed by a recursive preorder traversal of
# the right subtree.

def preorder(tree):
    if tree:
        print(tree.getRootVal())
        preorder(tree.getLeftChild())
        preorder(tree.getRightChild())

 # 2. Post-order

# In a postorder traversal, we recursively do a postorder
# traversal of the left subtree and the right subtree
# followed by a visit to the root node.

 def postorder(tree):
    if tree != None:
        postorder(tree.getLeftChild())
        postorder(tree.getRightChild())
        print(tree.getRootVal())

# 3. In-order

# In an inorder traversal, we recursively do an inorder
# traversal on the left subtree, visit the root node, and
# finally do a recursive inorder traversal of the right
# subtree.

def inorder(tree):
  if tree != None:
      inorder(tree.getLeftChild())
      print(tree.getRootVal())
      inorder(tree.getRightChild())
```