

```
1 # Binary trees: each node has no more than two child nodes
2
3 # BST: ordering property
4 # -> left is less than node
5 # -> right is greater than node
6
7 # Balanced vs Unbalanced
8 # Traversal:
9 # 1. pre-order:
10 # -> root, left, right
11 # 2. in-order:
12 # -> left, root, right
13 # 3. post-order:
14 # -> left, right, root
15
16 # BST IMPLEMENTATION
17
18 class Node:
19     def __init__(self, val):
20         self.l = None
21         self.r = None
22         self.v = val
23
24 class Tree:
25     def __init__(self):
26         self.root = None
27
28     def getRoot(self):
29         return self.root
30
31     def add(self, val):
32         if(self.root == None):
33             self.root = Node(val)
34         else:
35             self._add(val, self.root)
36
37     def _add(self, val, node):
38         if(val < node.v):
39             if(node.l != None):
40                 self._add(val, node.l)
41             else:
42                 node.l = Node(val)
43         else:
44             if(node.r != None):
45                 self._add(val, node.r)
46             else:
47                 node.r = Node(val)
48
49     def find(self, val):
50         if(self.root != None):
51             return self._find(val, self.root)
52         else:
53             return None
54
55
56
57
58
59
60
61
62
63
64
65
```

```
66 # HackerRank: CTCI
67 # Tree: is this a BST
68
69 def check_binary_search_tree_(root):
70     arr = []
71     count = 0
72     arr = inorderTraversal(root, arr)
73     if ((sorted(arr)) == arr) and (len(set(arr)) == len(arr)):
74         return True
75     else:
76         return False
77
78 def inorderTraversal(root, arr):
79     if root != None:
80         inorderTraversal(root.left, arr)
81         arr.append(root.data)
82         inorderTraversal(root.right, arr)
83     return arr
84
85
86 # Depth First Search
87
88 def dfs(graph, start):
89     visited, stack = set(), [start]
90     while stack:
91         vertex = stack.pop()
92         if vertex not in visited:
93             visited.add(vertex)
94             stack.extend(graph[vertex] - visited)
95     return visited
96
97
98 def bfs(graph, start):
99     visited, queue = set(), [start]
100     while queue:
101         vertex = queue.pop(0)
102         if vertex not in visited:
103             visited.add(vertex)
104             queue.extend(graph[vertex] - visited)
105     return visited
106
107 bfs(graph, 'A') # {'B', 'C', 'A', 'F', 'D', 'E'}
108
109
110 # Generator example
111
112 def square(nums):
113     for i in nums:
114         yield (i * i)
115
116 my_nums = square([1,4,2,5])
117
118 # OR: using comprehension
119
120 # comprehension
121 nums_comprehended = [x*x for x in [1,2,3,4]]
122 # becomes: generator
123 nums_generator = (x*x for x in [1,2,3,4])
124
125 print my_nums
126 print nums_comprehended
127 print nums_generator
128
129 print list(nums_generator)
130
```

```
131
132 # Iterators per Data Type
133
134 # List
135 lst = [1, 2, 3, 4]:
136
137 for i in lst:
138     pass
139
140 # String
141 string = "python"
142
143 for c in string:
144     pass
145
146 # Tuples
147 tup = (1,2,3,4,5,6,7,8,9,10)
148
149 for i in tup:
150     pass
151
152 # Dict
153 dictionary = {'name': 'Helen', 'age': '21', 'job': 'boss'}
154
155 for key, val in dictionary.iteritems():
156
157 for k in dictionary:
158     pass
159
160 # Set
161 my_set = {10,20,30,40,50,20}
162 for i in my_set:
163     pass
164
165 # File
166 for line in open("a.txt"):
167     pass
168
169 # Sorting
170
171 # http://danishmujeeb.com/blog/2014/01/basic-sorting-algorithms-implemented-in-python/
172
173 # 1. Bubble sort
174
175 # It's basic idea is to bubble up the largest(or smallest), then the 2nd largest
176 # and the the 3rd and so on to the end of the list. Each bubble up takes a full
177 # sweep through the list.
178
179 def bubble_sort(items):
180     for i in range(len(items)):
181         for j in range(len(items)-1-i):
182             if items[j] > items[j+1]:
183                 items[j], items[j+1] = items[j+1], items[j]
184
185
186
187
188
189
190
191
192
193
194
195
```

```
196 # 2. Insertion Sort
197
198 # Insertion sort works by taking elements from the unsorted list and inserting them
199 # at the right place in a new sorted list. The sorted list is empty in the beginning.
200 # Since the total number of elements in the new and old list stays the same, we can
201 # use the same list to represent the sorted and the unsorted sections.
202
203 def insertion_sort(items):
204     for i in range(1, len(items)):
205         j = i
206         while j > 0 and items[j] < items[j-1]:
207             items[j], items[j-1] = items[j-1], items[j]
208             j -= 1
209
210 # 3. Merge Sort
211
212 # Merge sort works by subdividing the the list into two sub-lists, sorting them using
213 # Merge sort and then merging them back up. As the recursive call is made to subdivide
214 # each list into a sublist, they will eventually reach the size of 1, which is
215 # technically a sorted list.
216
217 def merge_sort(items):
218     """ Implementation of mergesort """
219     if len(items) > 1:
220
221         mid = len(items) / 2          # Determine the midpoint and split
222         left = items[:mid]
223         right = items[mid:]
224
225         merge_sort(left)              # Sort left list in-place
226         merge_sort(right)             # Sort right list in-place
227
228         l, r = 0, 0
229         for i in range(len(items)):   # Merging the left and right list
230
231             lval = left[l] if l < len(left) else None
232             rval = right[r] if r < len(right) else None
233
234             if (lval and rval and lval < rval) or rval is None:
235                 items[i] = lval
236                 l += 1
237             elif (lval and rval and lval >= rval) or lval is None:
238                 items[i] = rval
239                 r += 1
240             else:
241                 raise Exception('Could not merge, sub arrays \
242 sizes do not match the main array')
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
```

```
261 # 4. Quick Sort
262
263 # Quick sort works by first selecting a pivot element from the list. It then creates
264 # two lists, one containing elements less than the pivot and the other containing
265 # elements higher than the pivot. It then sorts the two lists and join them with the
266 # pivot in between. Just like the Merge sort, when the lists are subdivided to lists
267 # of size 1, they are considered as already sorted
268
269 def quick_sort(items):
270     """ Implementation of quick sort """
271     if len(items) > 1:
272         pivot_index = len(items) / 2
273         smaller_items = []
274         larger_items = []
275
276         for i, val in enumerate(items):
277             if i != pivot_index:
278                 if val < items[pivot_index]:
279                     smaller_items.append(val)
280                 else:
281                     larger_items.append(val)
282
283         quick_sort(smaller_items)
284         quick_sort(larger_items)
285         items[:] = smaller_items + [items[pivot_index]] + larger_items
286
287 # 5. Heap Sort
288
289 # This implementation uses the built in heap data structures in Python. To truly
290 # understand heapsort, one must implement the heapify() function themselves. This
291 # is certainly one obvious area of improvement in this implementation.
292
293 import heapq
294
295 def heap_sort(items):
296     """ Implementation of heap sort """
297     heapq.heapify(items)
298     items[:] = [heapq.heappop(items) for i in range(len(items))]
299
300 # Stack implementation
301
302 class Stack:
303     def __init__(self):
304         self.items = []
305
306     def isEmpty(self):
307         return self.items == []
308
309     def push(self, item):
310         self.items.append(item)
311
312     def pop(self):
313         return self.items.pop()
314
315     def peek(self):
316         return self.items[len(self.items)-1]
317
318     def size(self):
319         return len(self.items)
320
321
322
323
324
325
```

```
326
327 # Queue Implementation
328
329 class Queue:
330     def __init__(self):
331         self.items = []
332
333     def isEmpty(self):
334         return self.items == []
335
336     def enqueue(self, item):
337         self.items.insert(0,item)
338
339     def dequeue(self):
340         return self.items.pop()
341
342     def size(self):
343         return len(self.items)
344 # Tree Traversals
345
346 # 1. pre-order
347
348 # In a preorder traversal, we visit the root node first,
349 # then recursively do a preorder traversal of the left
350 # subtree, followed by a recursive preorder traversal of
351 # the right subtree.
352
353 def preorder(tree):
354     if tree:
355         print(tree.getRootVal())
356         preorder(tree.getLeftChild())
357         preorder(tree.getRightChild())
358
359 # 2. Post-order
360
361 # In a postorder traversal, we recursively do a postorder
362 # traversal of the left subtree and the right subtree
363 # followed by a visit to the root node.
364
365 def postorder(tree):
366     if tree != None:
367         postorder(tree.getLeftChild())
368         postorder(tree.getRightChild())
369         print(tree.getRootVal())
370
371 # 3. In-order
372
373 # In an inorder traversal, we recursively do an inorder
374 # traversal on the left subtree, visit the root node, and
375 # finally do a recursive inorder traversal of the right
376 # subtree.
377
378 def inorder(tree):
379     if tree != None:
380         inorder(tree.getLeftChild())
381         print(tree.getRootVal())
382         inorder(tree.getRightChild())
383
```