Jan 2021 | © Guven

Review of Data Structures

This notebook demonstrates,

1. Data structures of array, stack, linked list, tree and hash
2. Testing them with properly filled data
3. Measuring time complexity empirically
4. List of `search` functions
5. RB Trees framework

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from random import randint
from time import perf_counter
```

In [2]:
```
# Array
def insert_A(_array, _val):  # inserts value _val at the end - measur
ing list appends
    if _array is None:
        _array = []
    return _array+[_val]

# For measuring insertion cost
def new_A_val(_n):
    return (randint(0,_n),)
```

In [3]:

```python
# Stack
INITIAL_STACK_SIZE = 100

class Stack:
    stack_size = INITIAL_STACK_SIZE

    def __init__(self):
        self.stack = stack_size*[None]
        self.tos = 0

    def __len__(self):
        return Stack.tos

    def push(self, _x):
        # Handle stack overflow
        if Stack.tos < Stack.stack_size:
            Stack.tos += 1
            Stack.stack[Stack.tos-1] = _x

    def pop(self):
        # Handle stack underflow
        if Stack.tos > 0:
            Stack.tos -= 1
            return Stack.stack[Stack.tos+1]
        return None

    def empty(_stack):
        return True if Stack.tos == 0 else False

# For measuring insertion cost
def insert_S(_stack, _val):
    _stack.push(_stack, _val)  # Or should we find a position k and i
nsert?
```

In [4]:
```python
# Linked-list
class LLNode:
    def __init__(self, x):
        self.x=x; self.next=None

    def __len__(self):  # Counting every item
        n, node = 1, self
        while node.next is not None:
            n += 1
            node = node.next
        return n

def insert_LL(_root, _newnode):
    if _root is None:  # creation of the root node
        _root = _newnode
    else:
        node = _root
        while node.next is not None:
            node = node.next
        node.next = _newnode
    return _root

# For measuring insertion cost
def new_LL_node(_n):
    return (LLNode(randint(0,_n)),)
```

In [5]:
```python
# Tree
class TNode:
    n_nodes=0
    def __init__(self, x):
        self.x=x; self.left=None; self.right=None

    def __len__(self):  # Keeping the count, otherwise have to traver
se the tree
        return TNode.n_nodes

def insert_BST(_root, _newnode):
    if _root is None:
        TNode.n_nodes = 0
    y = None; node = _root
    while node is not None:
        y = node
        if node.x < _newnode.x:
            node = node.right
        else:
            node = node.left

    if y is None:
        _root = _newnode
    elif y.x < _newnode.x:
        y.right = _newnode
    else:
        y.left = _newnode

    TNode.n_nodes += 1
    return _root

# For measuring insertion cost
def new_BST_node(_n):
    return (TNode(randint(0,_n)),)
```

In [6]:
```python
# Hash
def insert_H(_hash, k, v):
    if _hash is None:
        _hash = {}
    _hash[k] = v
    return _hash

# For measuring insertion cost
def new_H_val(_n):  # So there will be collisions
    return (randint(0,_n/10), randint(0,_n))
```

Empirical Time Complexity

Let's measure how long an `insert` takes empirically.
The following function takes an insert function and a node/value to insert.
The node/value is returned as an `Iterable` so its value can be 1 or more depending on the situation. For example a tree node would be a **value** only but a hash would require **key and value**. Inner loop runs 10 times to reduce the variation of the measurement for statistical purposes.

We will use the `time.perf_counter` to measure time accurately. Also we will use a log scale plot to show very small variations at a zoomed/focused level.

Below, the generation of random input variable is also included in the time measurement.
Unfortunately it is hard to get rid of it as it is a passed function to `node_f`.

In [7]:
```
def measure_cost(n_runs, insert_f, node_f):
    ds = None
    t = []
    for n in n_runs:
        runs = []
        for j in range(10):  # reduce the variation of the measuremen
t
            ds = None  # starting from an empty data structure
            st = perf_counter()
            for i in range(n):
                ds = insert_f(ds, *node_f(n))
            runs += [perf_counter()-st]

        t += [np.mean(runs)]

    print('clock: ', ' '.join(['{:g}'.format(v) for v in t]))
    # ds dataset can be used for search
    return t, ds
```

In [8]:
```
N_RUNS = [10,100,500,700,1000,2000,3000,5000,7000,9000,10000]

t1, ds1 = measure_cost(N_RUNS, insert_A, new_A_val)
t2, ds2 = measure_cost(N_RUNS, insert_LL, new_LL_node)
t3, ds3 = measure_cost(N_RUNS, insert_BST, new_BST_node)
t4, ds4 = measure_cost(N_RUNS, insert_H, new_H_val)
```

```
clock:  1.335e-05 0.00018959 0.00117511 0.00219212 0.0026183 0.007701
24 0.0150975 0.0327746 0.0752413 0.126521 0.152472
clock:  1.812e-05 0.00043124 0.00859135 0.0233096 0.0391681 0.129291
0.280848 0.725559 1.26333 2.1047 2.5564
clock:  2.205e-05 0.00023492 0.00135512 0.00185644 0.0027204 0.007682
22 0.00924789 0.021562 0.032392 0.0395772 0.0484651
clock:  2.206e-05 0.00020499 0.00110383 0.00153321 0.00206598 0.00481
355 0.00742348 0.0117127 0.0171652 0.0197633 0.0212367
```

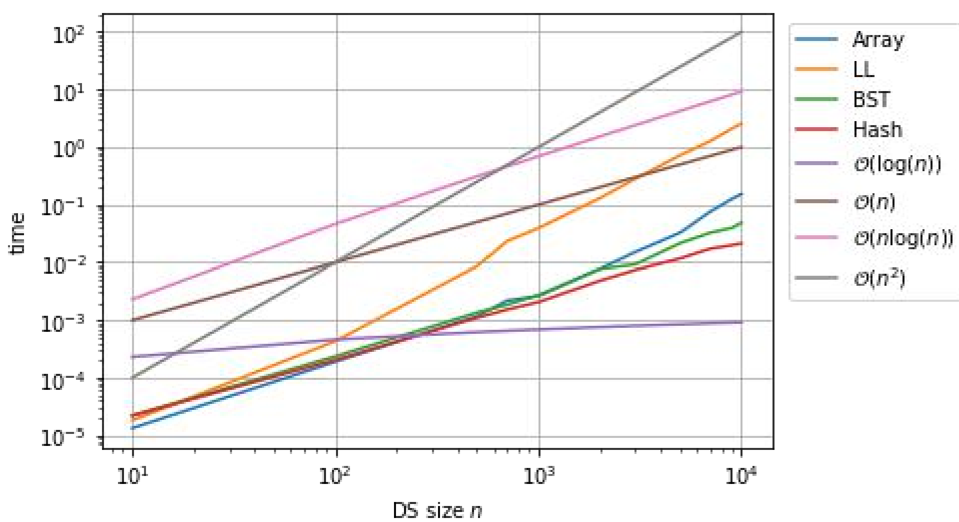
In [9]:
```
# Plot
df = pd.DataFrame({'DS size $n$':  N_RUNS,
                   'Array':        t1,
                   'LL':           t2,
                   'BST':          t3,
                   'Hash':         t4,
                   '$\mathcal{O}(\log(n))$': [1e-4*np.log(n) for n in
N_RUNS],
                   '$\mathcal{O}(n)$':       [1e-4*n for n in N_RUNS
],
                   '$\mathcal{O}(n\log(n))$':[1e-4*n*np.log(n) for n
in N_RUNS],
                   '$\mathcal{O}(n^2)$':     [1e-6*n**2 for n in N_RU
NS]
                  })
df.set_index('DS size $n$', drop=True, inplace=True)

fig = df.plot().get_figure()

plt.legend(bbox_to_anchor=(1.01, 1.0))

plt.ylabel('time')
plt.grid()
plt.xscale('log')
plt.yscale('log')

fig.savefig('data_structures_01.png')
```



Let's sanity check by examining the filled data structures - `ds1` , `ds2` , `ds3` , `ds4` are filled over and over again inside the `measure` function.

In [10]:
```
print(ds1[100], ds1[1000])
print(ds2.x, ds2.next.next.x)
print(ds3.x, ds3.right.right.right.x, ds3.right.right.left.x)
print(ds4[10], ds4[30])  # note that (k,v) randomly generated and has
h might not exist
```

```
1460 2461
4578 3347
8511 9125 9103
1403 2336
```

Search Function

In [11]:

```
# Search a vector/array for a value - O(n)
def search_A(_array, k):
    for i, x in enumerate(_array):
        if x == k:
            return i

    return None

# Search a linked list for a value - O(n)
def search_LL(_node, k):
    while _node != None:
        if _node.x == k:
            return _node
        _node = _node.next

    return None

# Search a BST recursively for a value - O(logn)
def search_BST_r(_node, k):
    if _node == None:
        return None
    elif _node.x == k:
        return _node

    if _node.x < k:
        return search_BST_r(_node.right, k)
    else:
        return search_BST_r(_node.left, k)

# Search a BST iteratively for a value - O(logn)
def search_BST_it(_node, k):
    while _node is not None and _node.x != k:
        if _node.x < k:
            _node = _node.right
        else:
            _node = _node.left

    return _node if _node is not None else None

# Search a hash/dictionary for a value - O(1)
def search_H(_hash, k):
    if k in _hash:
        return _hash[k]
    return None
```

Exercise: Write the code to search some values from the filled data structures and measure their complexity. Note that `n` has to be the size of the data structure so that you have to fill a new one for each `n` size measurement.

RB Trees

In [12]:
```
# used for RB tree node
RED, BLACK = 'R', 'B'

# Tnil necessary since code has reference assignments like y.right.p
class Tn:
    def __init__(self):
        self.p=None; self.color=BLACK

Tnil = Tn()

# All references are assigned to Tnil
class RBNode:
    def __init__(self, value):
        self.value=value; self.left=Tnil; self.right=Tnil; self.p=Non
e; self.color=None; self.height=None

def rotate_left(_root, x):
    y = x.right
    x.right = y.left  # turn y.left subT into x.right subT
    if y.left is not Tnil:
        y.left.p = x
    y.p = x.p  # link x’s parent to y
    if x.p is Tnil:
        _root = y
    elif x == x.p.left:
        x.p.left = y
    else:
        x.p.right = y
    y.left = x  # put x on y's left
    x.p = y
    return _root

def rotate_right(_root, x):
    y = x.left
    x.left = y.right  # turn y.right subT into x.left subT
    if y.right is not Tnil:
        y.right.p = x
    y.p = x.p  # link x’s parent to y
    if x.p is Tnil:
        _root = y
    elif x == x.p.right:
        x.p.right = y
    else:
        x.p.left = y
    y.right = x  # put x on y’s right
    x.p = y
    return _root
```

In [13]:
```
# Insert a node to the tree
def insert_RB(_root, z):  # insert node z with default color red
    # check if root inserted
    if _root is None:
        _root = z; z.color=BLACK; z.p=Tnil
        return _root
    # insert node
    y = Tnil; x = _root
    while x is not Tnil:
        y = x
        if z.value < x.value:
            x = x.left
        else:
            x = x.right
    #
    z.p = y
    if y == Tnil:
        _root = z
    elif z.value < y.value:
        y.left = z
    else:
        y.right = z
    z.color = RED
    # fixup
    while z.p.color == RED:
        if z.p == z.p.p.left:  # z parent is left child
            y = z.p.p.right
            if y.color == RED:  # case 1
                z.p.color = BLACK
                y.color = BLACK
                z.p.p.color = RED
                z = z.p.p
            else:
                if z == z.p.right:  # case 2
                    z = z.p
                    _root = rotate_left(_root, z)
                # case 3
                z.p.color = BLACK
                z.p.p.color = RED
                _root = rotate_right(_root, z.p.p)
        else:  # z parent is right child
            y = z.p.p.left
            if y.color == RED:  # case 1
                z.p.color = BLACK
                y.color = BLACK
                z.p.p.color = RED
                z = z.p.p
            else:
                if z == z.p.left:  # case 2
                    z = z.p
                    _root = rotate_right(_root, z)
                # case 3
                z.p.color = BLACK
                z.p.p.color = RED
                _root = rotate_left(_root, z.p.p)
    #
```

```python
        _root.color = BLACK  # red reached to root
    return _root
```

In [14]:
```python
# RB search iteratively, complexity O(log n)
def search_RB(_node, _val):
    while _node != Tnil and _node.value != _val:
        if _node.value < _val:
            _node = _node.right
        else:
            _node = _node.left
    #
    if _node != Tnil:
        return _node
    # return Tnil when value not found
    return Tnil
```

In [15]:
```python
import numpy as np

# Insert values and check if we can find them in the tree
def test_RB(vals):
    # Fill values
    rootrb = None
    for val in vals:
        rootrb = insert_RB(rootrb, RBNode(val))
    # Check values
    for val in vals:
        ret = search_RB(rootrb, val)
        if ret == Tnil:
            print(f'Test failed, value={val} not found in the tree.')
    #
    print(f'Test passed.')
    return rootrb



# Test
n = 10000

print('Random values (with replacement):')
vals = np.random.randint(0, n//2, n)

root1 = test_RB(vals)
```

```
Random values (with replacement):
Test passed.
```

Exercises

Write the code to measure the height and size of the tree (i.e. number of elements it has) and add these to the test while running the test for `[10-10000000]` number of randomly generated elements.