Binary Trees

in Python

Data and File Structures Laboratory

http://www.isical.ac.in/~dfslab/2020/index.html

Binary trees

(Recursive) Definition

A *binary tree* over a domain D is either:

- the empty set (called an *empty binary tree*); or
- a 3-tuple $\langle S_1, S_2, S_3 \rangle$ where
  - $S_1 \in D$, (called the *root*) and
  - S_2 and S_3 are *binary trees* over D (called the *left* and *right* subtree resp.)

Binary trees

(Recursive) Definition

A *binary tree* over a domain D is either:

- the empty set (called an *empty binary tree*); or
- a 3-tuple $\langle S_1, S_2, S_3 \rangle$ where
  - $S_1 \in D$, (called the *root*) and
  - S_2 and S_3 are *binary trees* over D (called the *left* and *right* subtree resp.)

Properties

- *Height* (h): length of longest path from the root to a leaf
  - our convention: tree containing only a single node has $h = 0$
- *Number of nodes:* n
- *Number of leaves:* l

Binary tree traversals

- *Preorder*
- *Inorder*
- *Postorder*

Binary tree implementation

Conventional implementation

- For each node, need to store *data*, *left subtree*, *right subtree*, *parent* (optional)
- Possible built-in types that could be used

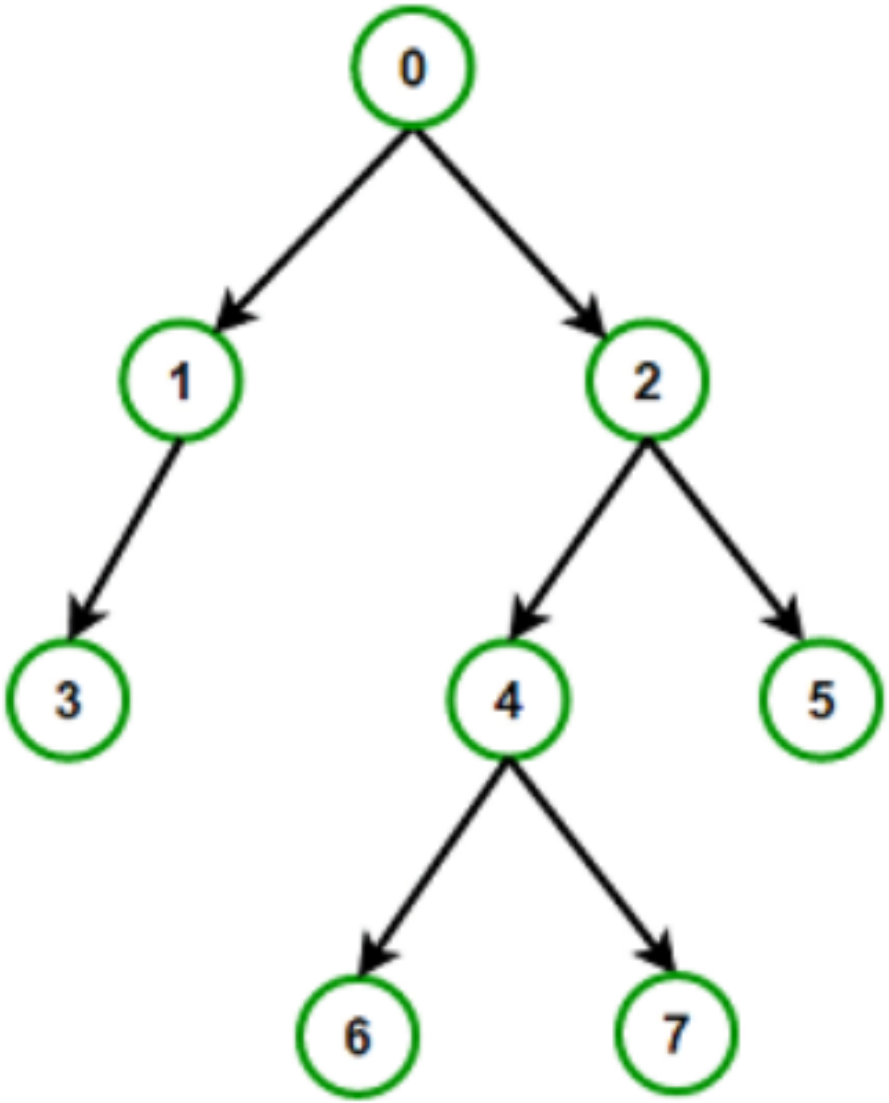
	tuple	**list**	**dict**
Indexing	positional	positional	named
Mutable	NO	Yes	Yes

(also see: `collections.namedtuple`)

- Use dict for now, better method later

Binary tree implementation

Example



Binary tree implementation – attempt I

```
root = {'data': 0,
        'left': None,
        'right': None,
        'parent': None}

root['left'] = {'data': 1, 'left': None, 'right': None, 'parent': root}
root['right'] = {'data': 2, 'left': None, 'right': None, 'parent': root}

l = root['left']
l['left'] = {'data': 3, 'left': None, 'right': None, 'parent': l}

r = root['right']
r['left'] = {'data': 4, 'left': None, 'right': None, 'parent': r}
r['right'] = {'data': 5, 'left': None, 'right': None, 'parent': r}

x = r['left']
x['left'] = {'data': 6, 'left': None, 'right': None, 'parent': x}
x['right'] = {'data': 7, 'left': None, 'right': None, 'parent': x}
```

Basic functions

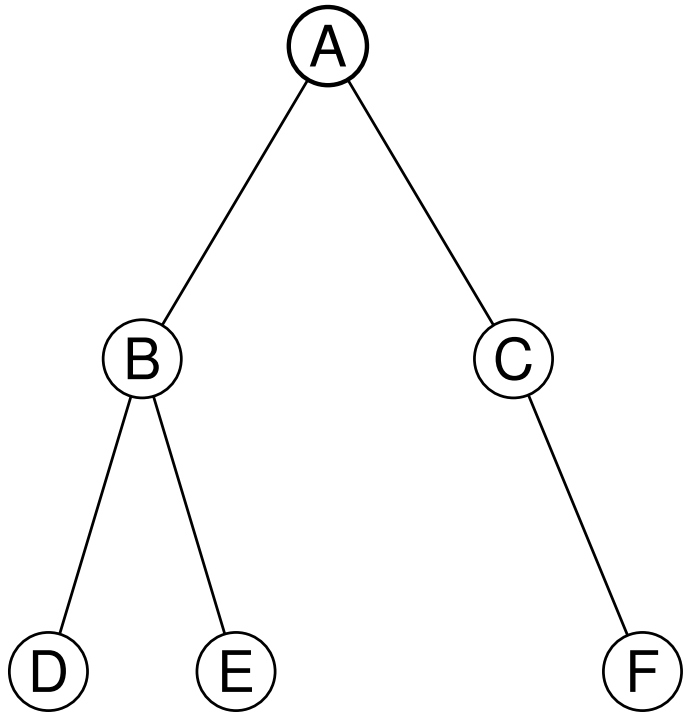
```
def preorder(T) :
    if T == None: return
    print(T['data'], end=" ")
    preorder(T['left'])
    preorder(T['right'])

def inorder(T) :
    if T == None: return
    inorder(T['left'])
    print(T['data'], end=" ")
    inorder(T['right'])
```

Binary tree implementation – attempt II

Alternative (tabular) implementation:

root = 0

	DATA	left	right
0	A	1	2
1	B	3	4
2	C	-1	5
3	D	-1	-1
4	E	-1	-1
5	F	-1	-1
free ⟶ 6	—	7	-1
⋮	⋮		
n-1	—	-1	-1



Binary tree implementation – attempt II

- All nodes of the tree are stored in a list.
- Each node is a dictionary with 3 or 4 keys: data, left, right, parent (optional).
- For a node, 'left' and 'right' store the indices of the left and right child of that node.
- If a node has no left / right child, the corresponding field is set to -1.

Binary tree implementation – attempt II

Alternative (tabular) implementation:

Initially:

root = -1

	DATA	left	right
free ⟶ 0	—	1	-1
1	—	2	-1
2	—	3	-1
3	—	4	-1
⋮		⋮	
n-1	—	-1	-1

Binary tree implementation II – `print_tree`

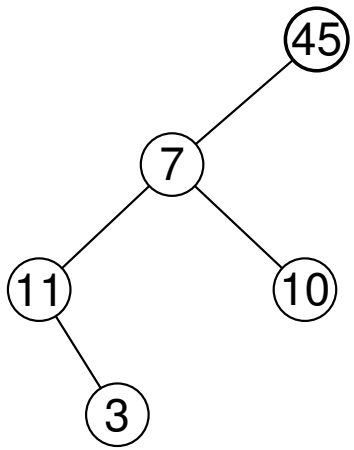
```
def print_tree (tree, f) :
    N = len(tree)
    print(N, file=f)
    for i in range(N) :
        print(tree[i]['data'], tree[i]['left'], tree[i]['right'],
    file=f)
    return
```

Binary tree implementation II – `print_tree`

```
def print_tree (tree, f) :
    N = len(tree)
    print(N, file=f)
    for i in range(N) :
        print(tree[i]['data'], tree[i]['left'], tree[i]['right'],
    file=f)
    return
```

Example output

```
5
45 1 -1
7 2 4
11 -1 3
3 -1 -1
10 -1 -1
```



Binary tree implementation II – `read_tree`

```
def read_tree (filename) :
    with open(filename) as f : # with => don't need a f.close()
        for line in f:
            fields = list(map(int, line.split()))
            if len(fields) == 1 : # this is the first line
                N = int(fields[0])
                tree = [ {'data': None, 'left': None, 'right': None,
    'parent': None}
                         for i in range(N) ]
                index = 0
            else :
                tree[index]['data'] = fields[0]
                tree[index]['left'] = fields[1]
                tree[index]['right'] = fields[2]
                index += 1
    return tree
```

Binary tree implementation II – `read_tree`

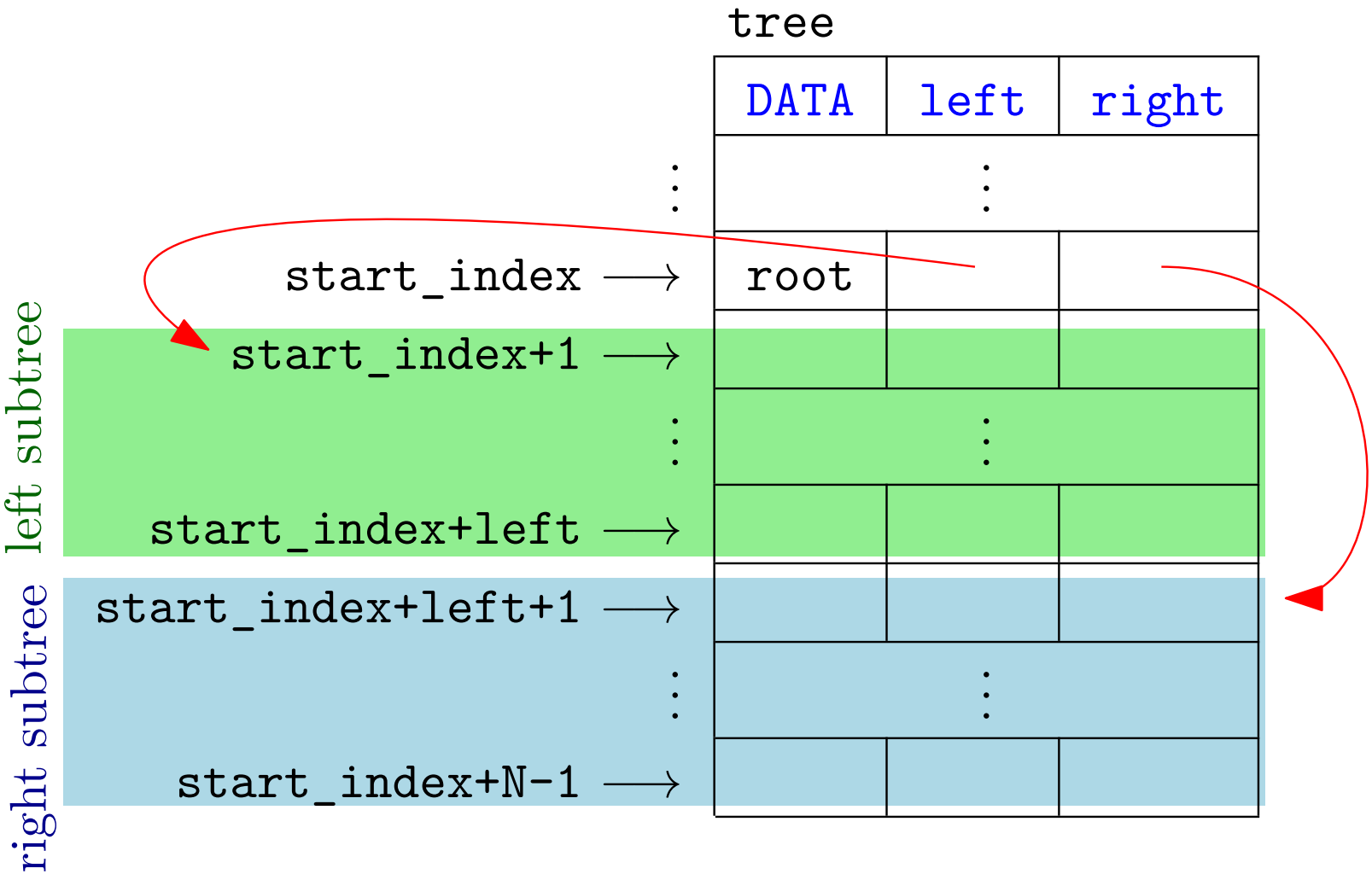
```
def read_tree (filename) :
    with open(filename) as f : # with => don't need a f.close()
        for line in f:
            fields = list(map(int, line.split()))
            if len(fields) == 1 : # this is the first line
                N = int(fields[0])
                tree = [ {'data': None, 'left': None, 'right': None,
    'parent': None}
                                for i in range(N) ]
                index = 0
            else :
                tree[index]['data'] = fields[0]
                tree[index]['left'] = fields[1]
                tree[index]['right'] = fields[2]
                index += 1
    return tree
```

Binary tree implementation II – `read_tree`

```
>>> line = "45 1 -1"
>>> line.split()
['45', '1', '-1']
>>> map(int, line.split())
<map object at 0x7ff51bdaf908>
>>> list(map(int, line.split()))
[45, 1, -1]
>>>
```

- `map` returns an *iterator object*
- iterators generate elements 'on demand'

Binary tree implementation II – `generate_random_tree`



Binary tree implementation II – `generate_random_tree`

```
def generate_random_tree(tree, start_index, num_nodes) :
    left = randint(0,num_nodes-1)
    right = num_nodes - left - 1
    tree[start_index]['data'] = randint(1,10*num_nodes)
    if left > 0 :
        tree[start_index]['left'] = start_index + 1
        generate_random_tree(tree, start_index + 1, left)
    else :
        tree[start_index]['left'] = -1
    if right > 0 :
        tree[start_index]['right'] = start_index + left + 1
        generate_random_tree(tree, start_index + left + 1, right)
    else :
        tree[start_index]['right'] = -1
    return
```

Digression: saving / loading data using **pickle**

- `pickle.dump(obj, file, protocol=None, *, fix_imports=True)`
  - write 'pickled' representation of `obj` to `file`
  - `file` must correspond to a file opened in binary mode
- `pickle.load(file, *, fix_imports=True, encoding="ASCII", errors="strict")`
  - read 'pickled' object representation from `file` and return the reconstituted object hierarchy specified therein
  - `file` must correspond to a file opened in binary mode

Problems – I

1. Given a binary tree with integer-valued nodes, and a *target* value, determine whether there exists a root-to-leaf path in the tree such that the sum of all node values along that path equals the target.
   Modify your program to consider *all* paths, not just root-to-leaf paths.
2. Given a binary tree stored in an array (as in the Alternative Implementation), and the indices of two nodes in the tree, find the index of the node that is the lowest common ancestor of the given nodes.
3. Write a program to print a binary tree, rotated anti-clockwise by 90° on the screen. For example, for the tree on slide 8, your output should like something like:

Problems – II

```
          F
     C
A
          E
     B
          D
```

Now, try to write a program that will print the same tree in the following format:

```
    ----A----
    |       |
  --B--     C--
  |   |       |
  D   E       F
```