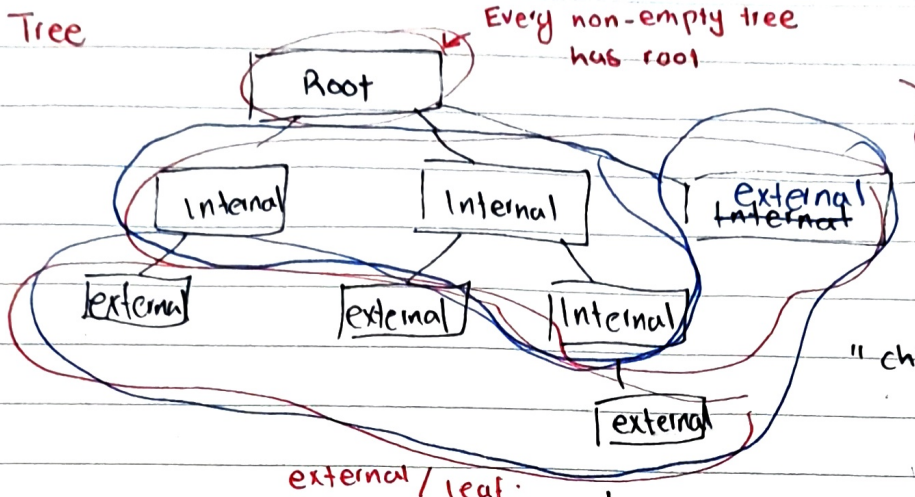
Week 3 - Lecture

Tree



external / leaf: nodes without a child

Ancestors: parent, grandparents, great-grandparent ...

descendants: child, grandchild, ...

~~Sibling~~

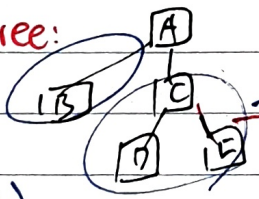
Siblings: two nodes with same parent

Depth: Number of ancestors (not including itself.
(depth of root = 0)

Level: set of nodes with same depth

Height of a tree: maximum depth

Subtree:



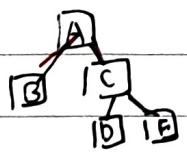
subtree rooted at B (by itself)

subtree rooted at C

edge: pair of nodes with P-C relation [u, v]

Path: consecutive edges (both up & down)

Lowest Common Ancestor



LCA of B and E is A

for z to be LCA of x, y

z must be ancestor of both x, y and no other descendant of z has such property.

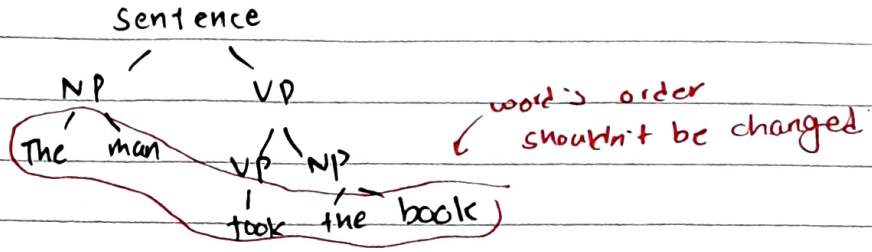
(A is not LCA of D, E)

Ordered Trees

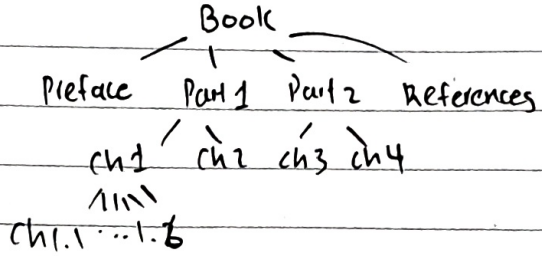
- Order of sibblings matter!?

ex) ~~folder~~ OS file system [files & folders under same folder are listed alphabetically]

ex) Phrase structure tree



ex) Document structure



Tree ADT

```
int size();
boolean isEmpty();
Iterator iterator();   ← more complicated to iterate through all nodes than in lists
Iterable positions();

Position root(); Position parent(p); Iterable children(p);
Integer numChildren();

Boolean isRoot(), isInternal(), isExternal()
```

↑ check if it has parent

↳ check if it has no children

Implementing...

Node has

- value associated
- ordered list/set of children
- parent (optional if we only need to go down the tree)

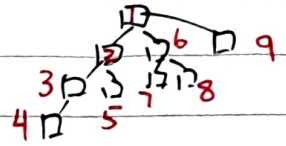
Traversals

Preorder traversal

```
def pre_order_trav(v):
    visit(v)
    for each child w of v:
        preorder_trav(w)
```

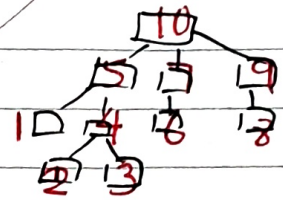
recursion

- print values
- aggregate values
- modify selectively values



Postorder traversal

```
def post_order(v):
    for each child w of v:
        post_order(w)
    visit(v)
```



Binary Trees

↳ each internal node has at most ~~one~~ two children.

tree is proper when each node has exactly 2 children

ex) Arithmetic expression tree

Decision trees

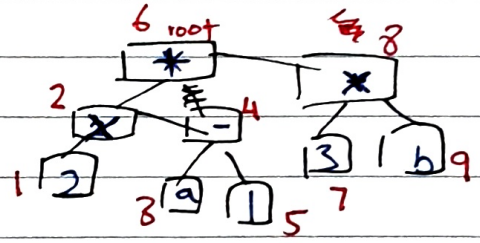
ADT

position leftchild(p); rightchild(p), sibling(p)

if both are null → node is a leaf

Inorder Traversal (apply only for binary trees)

```
def in_order(v):
    if v.left ≠ null:
        in_order(v.left)
    visit(v)
    if v.right ≠ null
        in_order(v.right)
```

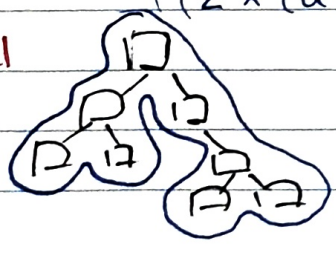


((2 x (a - 1)) + (3 x b))

((2 x (a-1)) + (3 x b))

Euler Tour Traversal

* walk around the tree

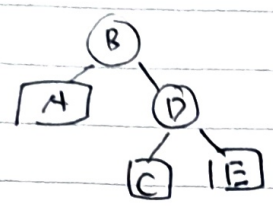


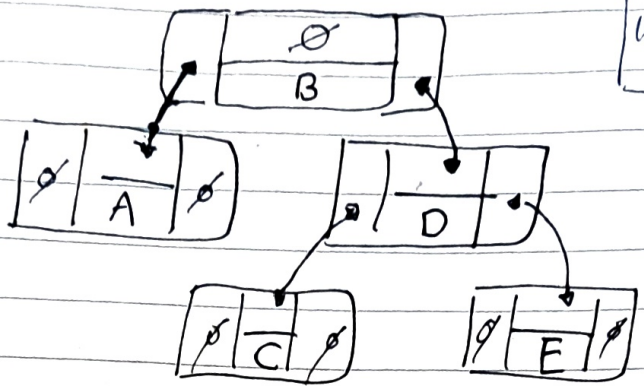
left: pre-order

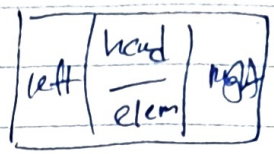
bottom: in-order

right: post-order

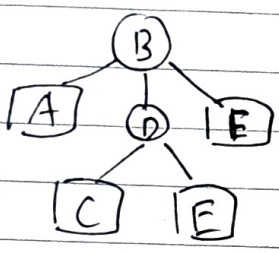
Linked Structure for binary trees

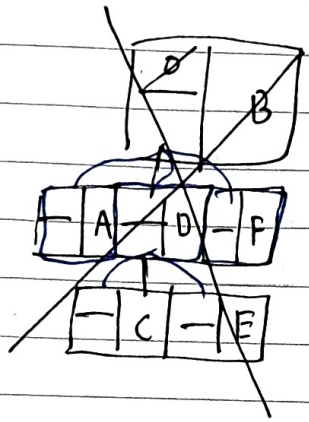


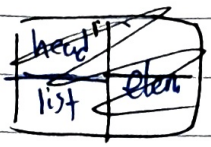


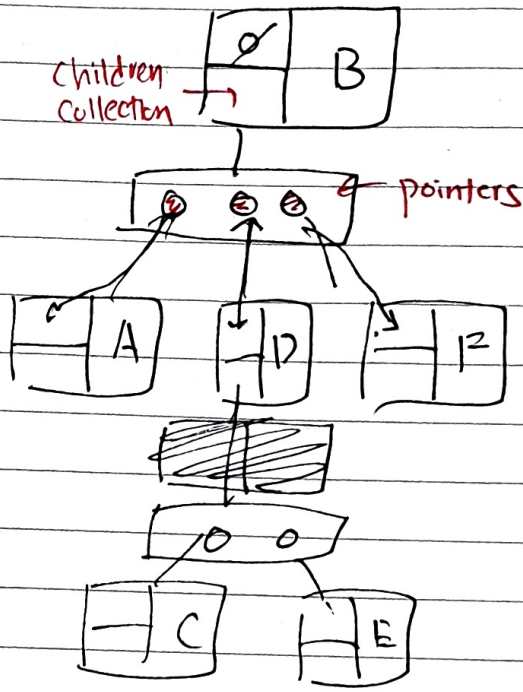


for general trees









Recursive codes on trees

```
def depth(v):
    if v.parent == null:
        return 0
    else
        return depth(v.parent) + 1

def height(v):
    if v.external():
        retun 0
    else
        ~~for child w of v:~~
        max = 0
        for child w of v:
            if( ~~height(w) > max~~
                max = h = height(w)
            if (w > max): max = h
        return max + 1
```

- O(height)
- x O(n) every nodes
- O(n)