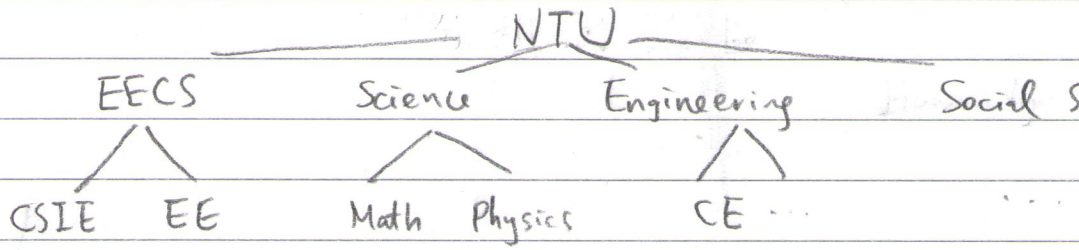
* Trees

Vector / array (indexed access) stack/queue (restricted access)

list (sequential access)

tree : hierarchical access



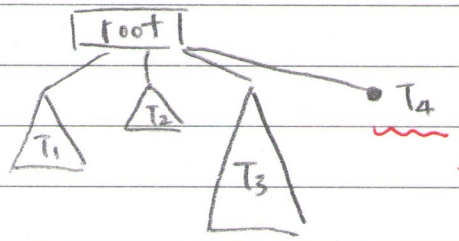
* similarly, directory/files in your filesystem

* formal definition

$T \equiv (root ; T_1, T_2, \cdots, T_n)$ recursive definition

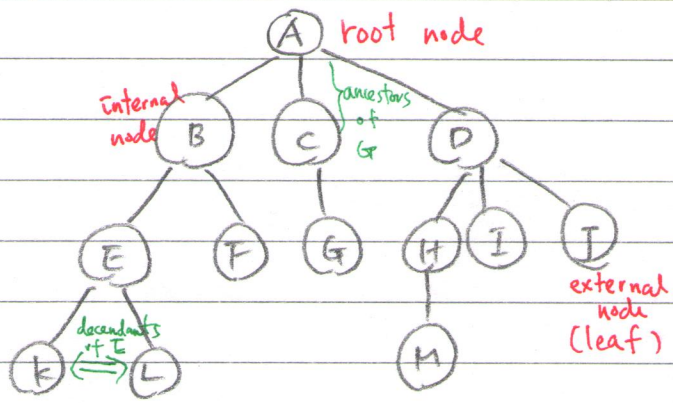
disjoint subtrees : no cross links

(ordered or unordered)



termination of recursion (no subtrees)

*



level (depth) 0

depth 1

depth 2

depth 3 = height of tree

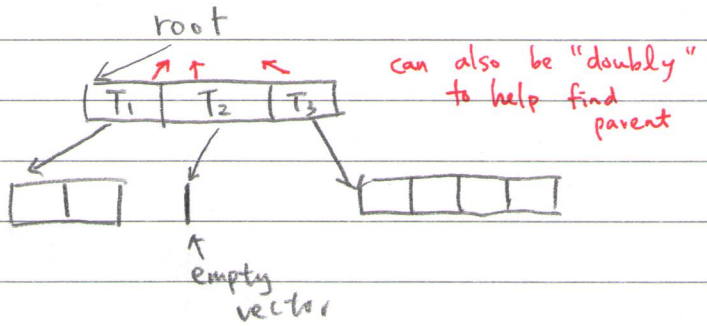
↑ parent ↓ child ⟷ sibling

degree of node : # child degree of tree: max degree of node

* representing trees

$T \equiv (root ; T_1, T_2, \cdots, T_n)$

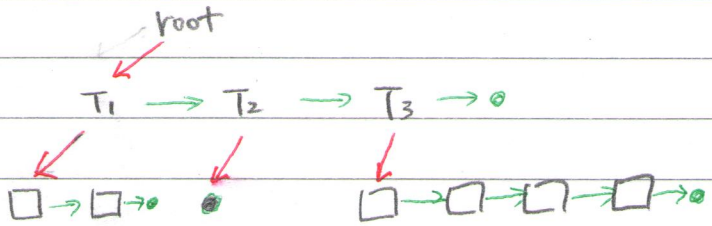
* use vector as container (space : O(n) in general)



isRoot(p) : O(1)

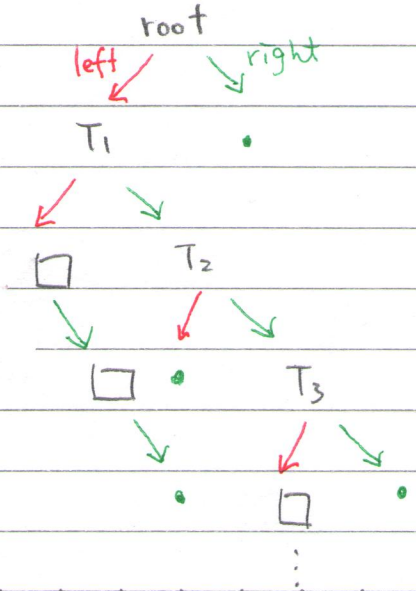
isLeaf(p) : O(1)

* use l-list as container



called left-child right-sibling

* rotate a little

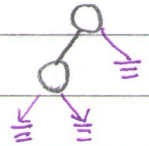


an equivalent representation of general tree

by

binary tree

deg = 2, ordered, allowing empty subtrees

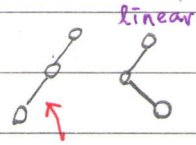


usually don't draw !

* how many nodes ?

$h=0$	min = 1	max = 1
$h=1$	min = 2	max = 3
$h=2$	min = 3	max = 7

min = $h+1$ (linear)



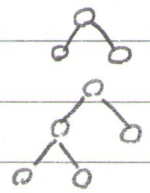
"skewed"

max = $2^{h+1}-1$ (exponential)

full binary tree

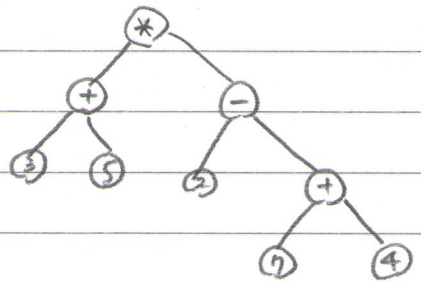
proper binary tree : two non-empty children per internal node

$h=0$	min = 1	max = 1
$h=1$	min = 3	max = 3
$h=2$	min = 5	max = 7



min = $2h+1$

e.g. proper binary tree for binary operations



(expression tree)

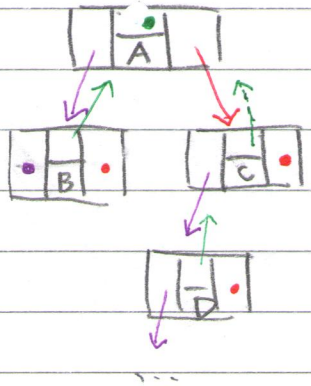
* $h+1 \leq n \leq 2^{h+1}-1$

$\Updownarrow$

$\log(n+1)-1 \leq h \leq \frac{n-1}{2}$

logarithmic (more efficient) — linear (like linked list)

* linked representation of binary tree

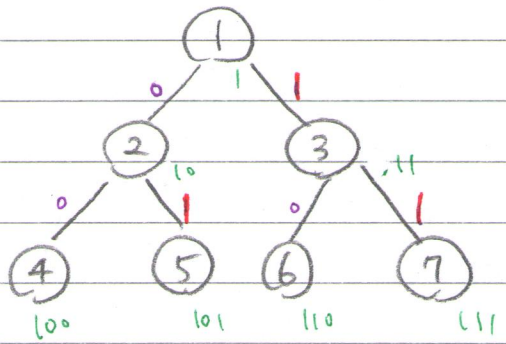


left

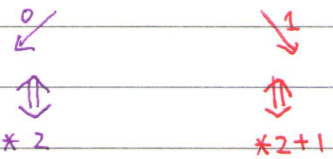
right

parent (optional)

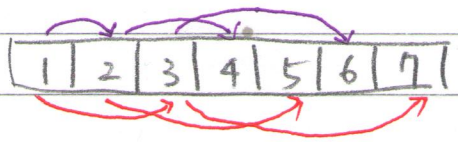
* full binary tree



node # = $(1 \cdot \text{path code})_2$



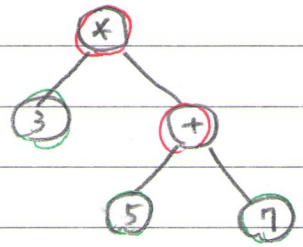
can "pack" the tree in a vector



- links "implicit" (no need to store)
- waste space if not full binary tree (useful if nearly full)

complete binary tree : nearly full (w/ nodes 1 ~ n exactly)

* expression tree revisited



3 * (5 + 7)

internal : operator

external : operands

sub-tree : ( )

print out infix notation

```
InfixPrint ( tpt ) {
• if (isLeaf(p))   print p→data; // operand
  else {
      print "(";
•     InfixPrint (p→left);
•     print p→data;   // operator
•     InfixPrint (p→right);
      print ")";
  }
}
```

• : inorder traversal of the tree (print ⇒ visit)

visit sequence : 3, *, 5, +, 7

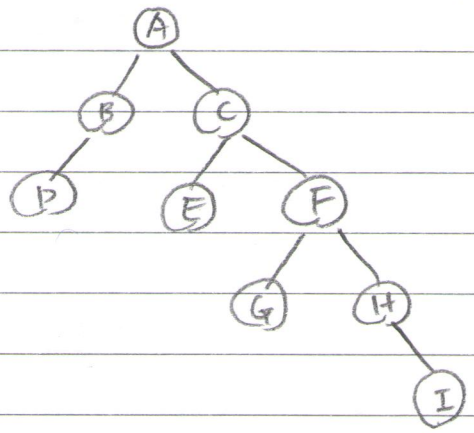
* postfix notation ⇒ postorder traversal

Postfix (p→left); Postfix (p→right); visit p→data;

prefix notation ⇒ prefix traversal

visit p→data; Prefix (p→left); Prefix (p→right);

*



in : D B A E C G F H I
post : D B E G I H F C A
pre : A B D C E F G H I

* why traversal : many bin. tree operations are similar to one of the traversal

postorder on exp. tree ⇒ evaluation
preorder on two bin. trees ⇒ equality test
inorder on T_L < root < T_R ⇒ ordered output