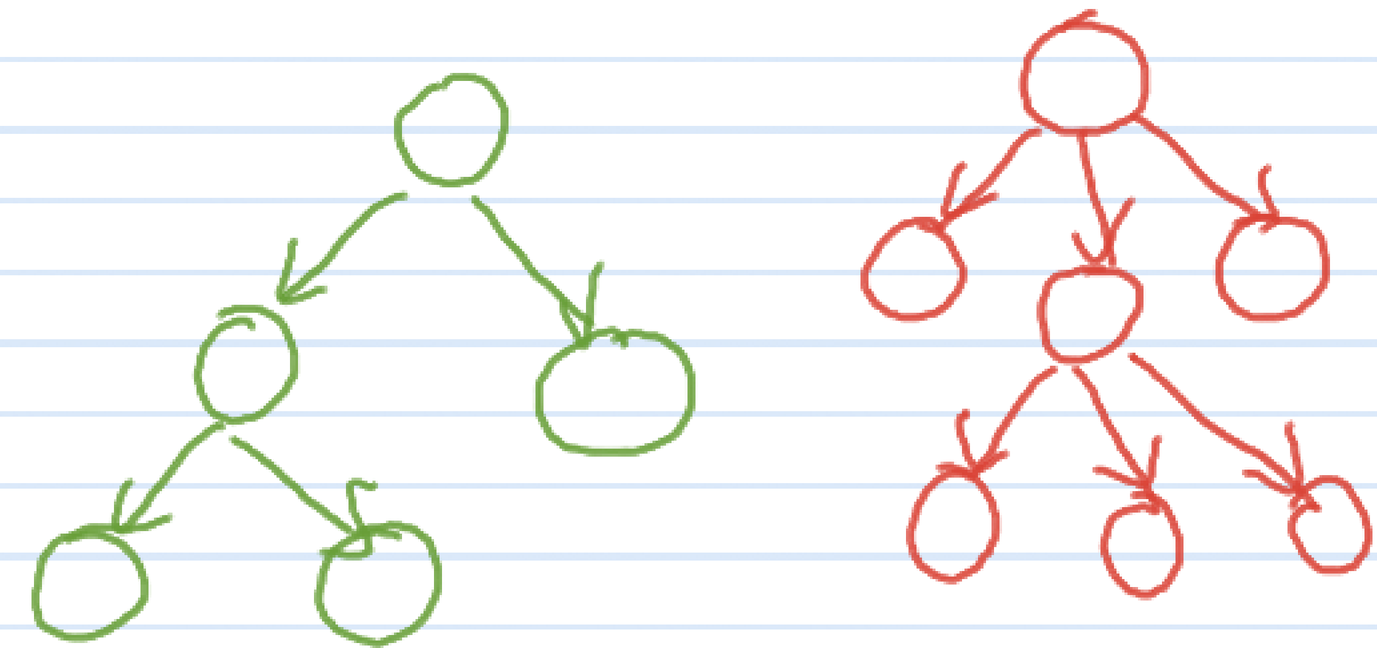
CS2700 PROGRAMMING AND DATA STRUCTURES

WEEK 7-8 TREES



Why trees?

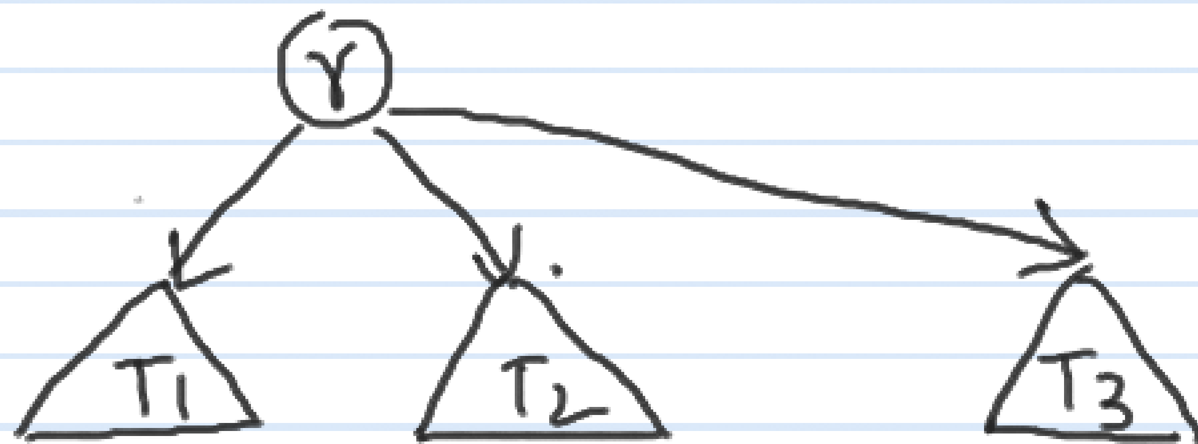
- Allow storage and access hierarchical data
- Efficient search of data

What is a tree?

- A TREE is a collection of nodes
  - collection can be empty (base case)

ELSE :

- A TREE consists of a distinguished node r and zero or more non empty subtrees $T_1, T_2 \ldots T_k$ connected by directed edges.



Every node except the root has exactly one parent

Another way to define a tree

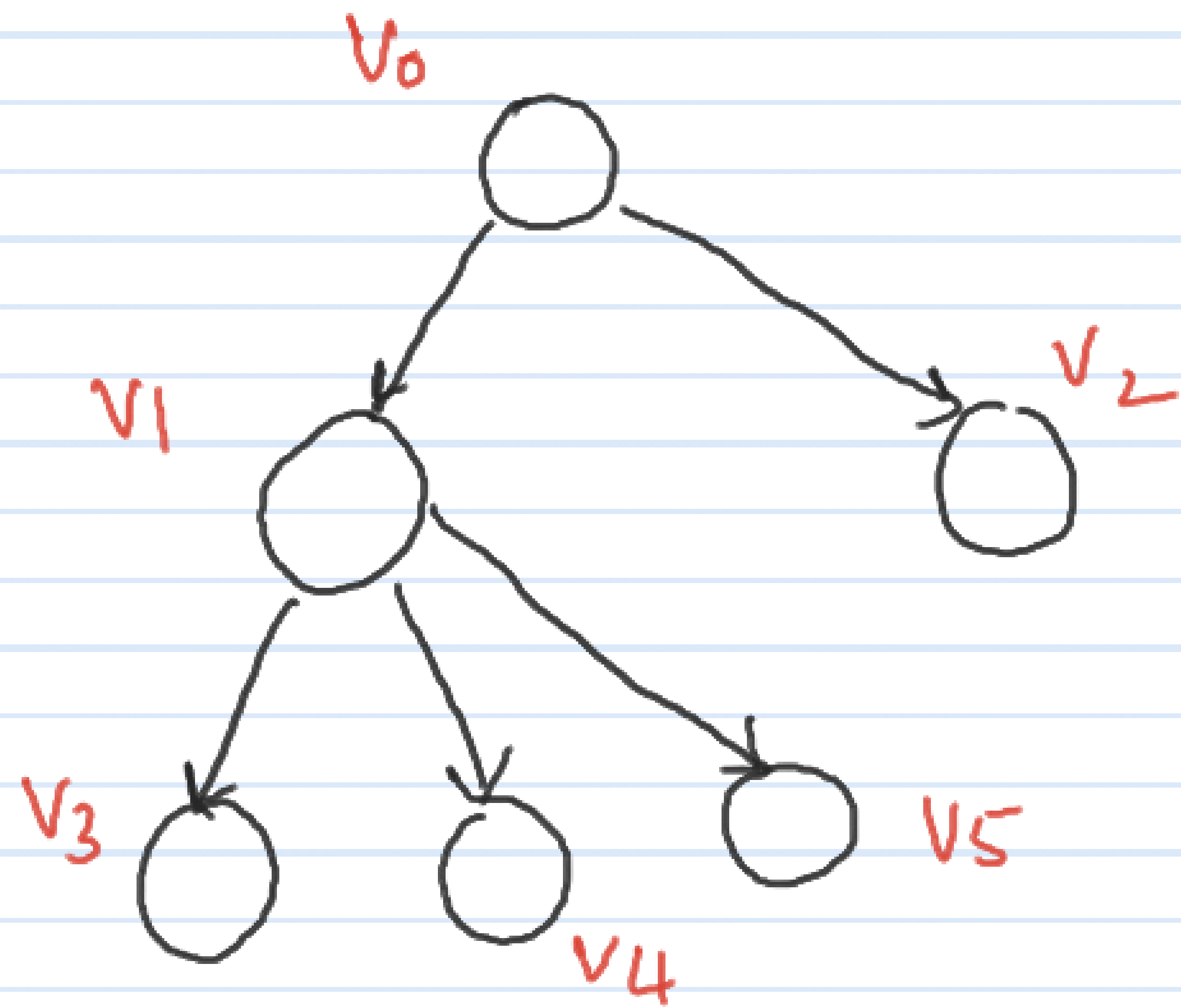
- Collection of nodes and directed edges s.t every node except root has exactly one parent

- Unique node w/o parent is called root

- Nodes without children are called leaves

- Rest of the nodes are internal nodes

SOME MORE DEFINTIONS



PARENT

SIBLING

GRAND PARENT

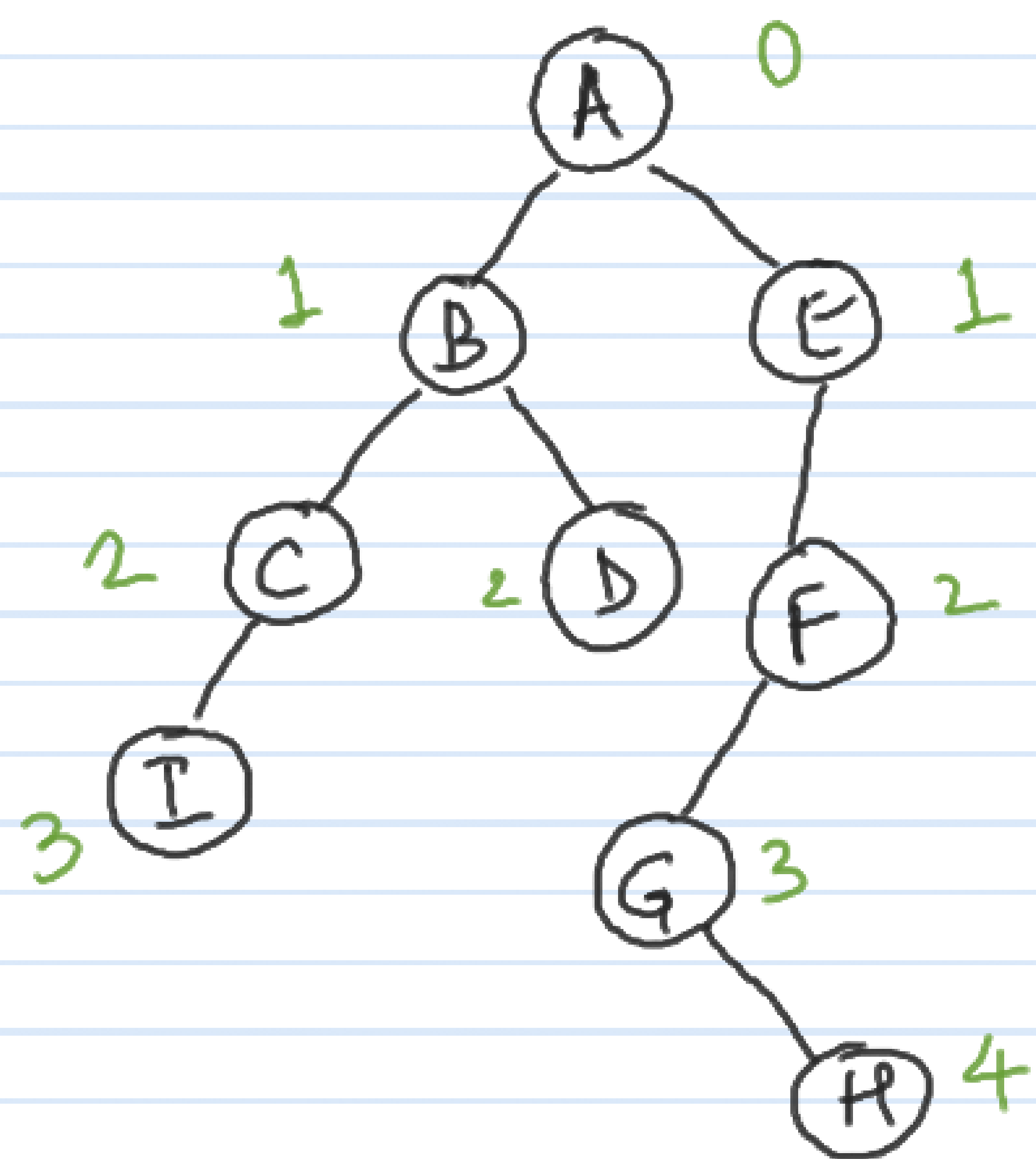
ANCESTOR

- PATH
- DEPTH

GOALS IN THIS PART

- TREE ADT
- APPLICATIONS WHERE TREES ARE USEFUL
- TREE TRAVERSALS
- SPECIAL TREES (BINARY SEARCH TREES)

BINARY TREES



N nodes (N = = 9) in example

Depth of tree

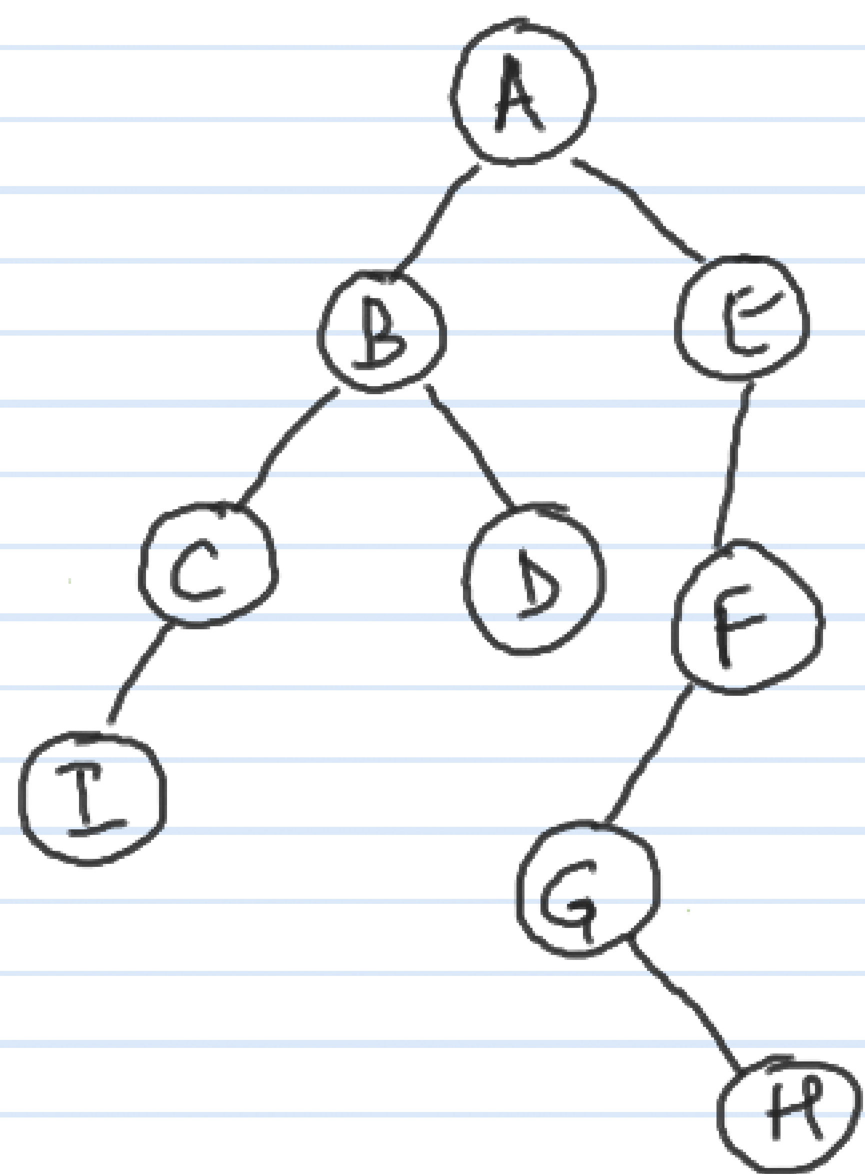
= depth of deepest leaf

Height of tree : height of root

Recall : Depth of node = = length of unique path from root to node

Height of a node : length of longest path from node to a leaf

BINARY TREES



N nodes ✓ (N = = 9) in example

max / min height : ?? ✓

max / min leaves : ?!

# of NULL pointers : n+1 :)

height and size defined recursively

height (r) = 1+ max(height (r→left)
height (r→right))

A B C I D E F G H

BINARY TREES : Pre order traversal

```
struct TreeNode {
        DataType data;
        struct TreeNode *left;
        struct TreeNode * right;
}
struct TreeNode * root;
```

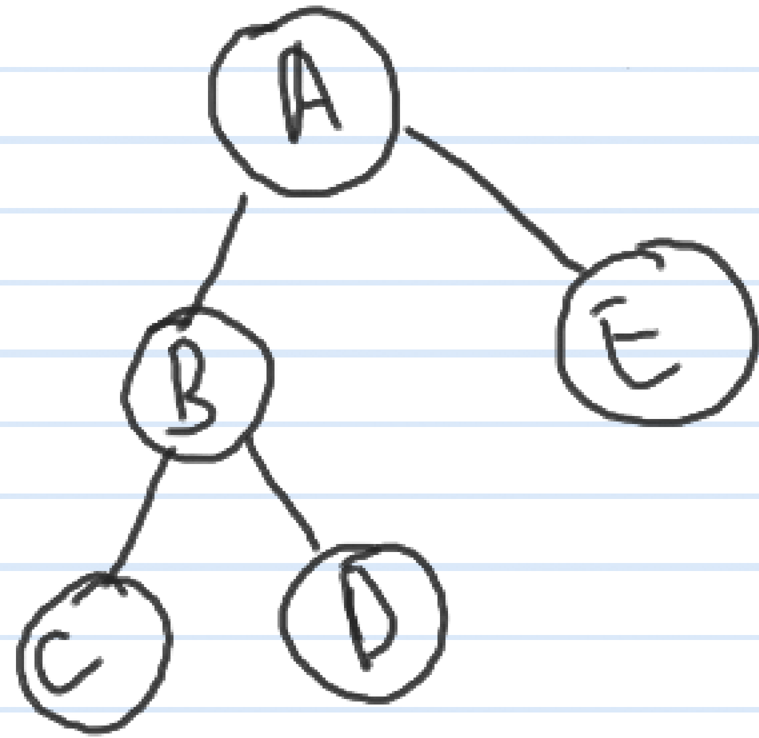
```
preorder (struct TreeNode r)
{
    if (r != NULL)
        print (r -> data);
        preorder (r -> left);
        preorder (r -> right);
}
```

BINARY TREES : PREORDER TRAVERSAL (w/o recursion)

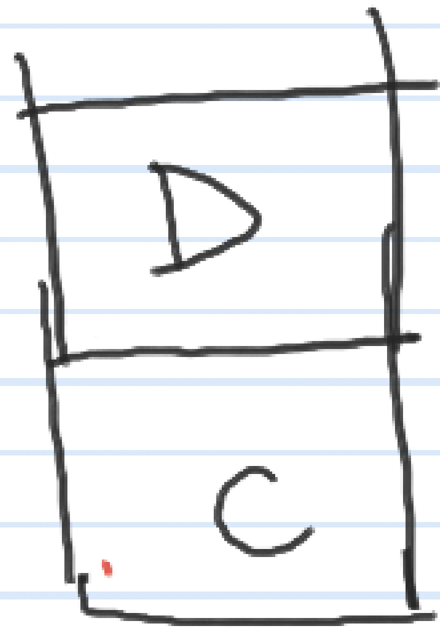
```
struct TreeNode {
        DataType data;
        struct TreeNode *left;
        struct TreeNode *right;
}

struct TreeNode *root;
```

```
preorder (struct TreeNode r)
{   stack< struct TreeNode > S
    if (r != NULL)
    S.push(r);
    while(S.empty() != true)
    {  temp = S.top(); S.pop();
       print(temp->data);
       S.push(temp->left);
       S.push(temp->right);
    }
}
```

BINARY TREES : PREORDER TRAVERSAL (w/o recursion)



A B C D E



A E B D C

```
preorder (struct TreeNode r)
{   stack< struct TreeNode > S
    if (r != NULL)
    S.push(r);
    while (s.empty() != true)
    {   temp = S.top(); S.pop();
        print (temp -> data);
        S. push(temp -> left);
        S. push (temp -> right);
} }
```

BINARY TREES : APPLICATIONS :

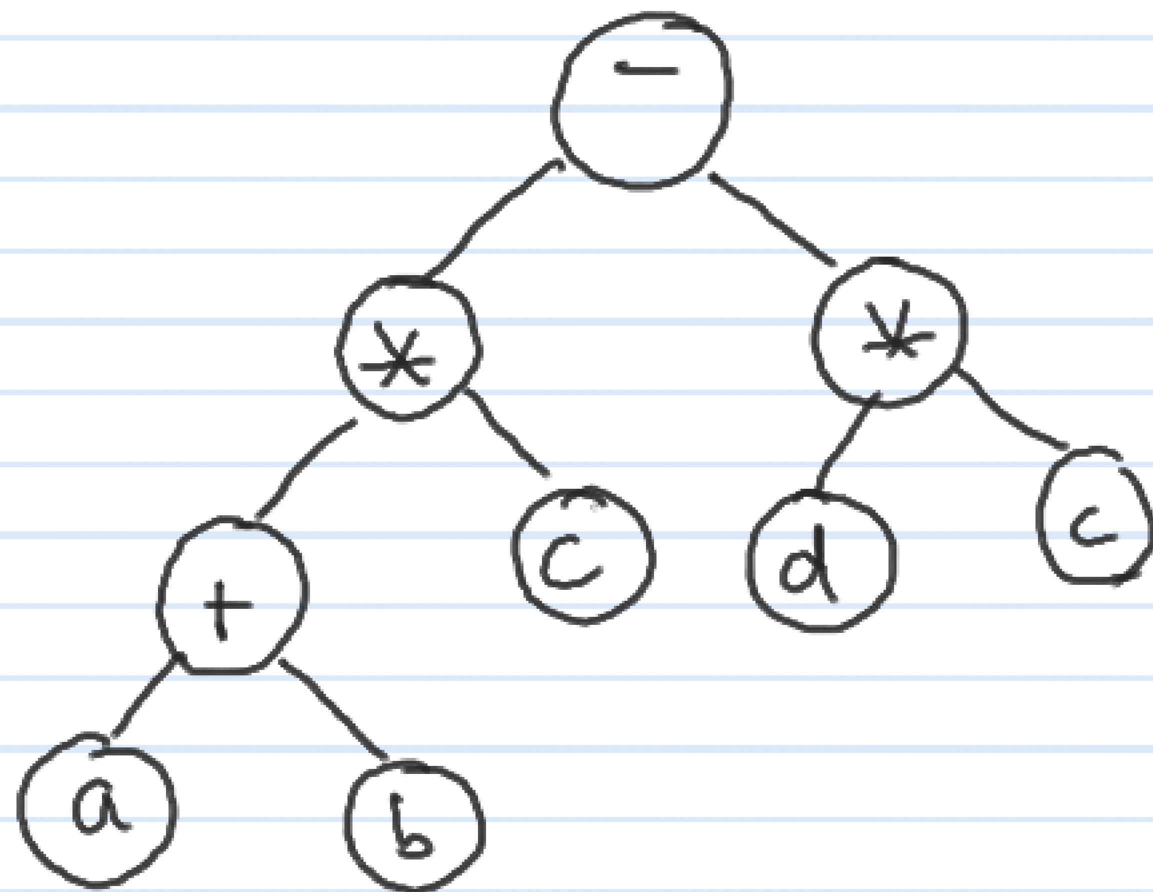
(1) EXPRESSION TREES

(2) CODING (not programming!)

EXPRESSION TREES :

A convenient way to represent expressions

$$((a+b) * c) - (d * c)$$

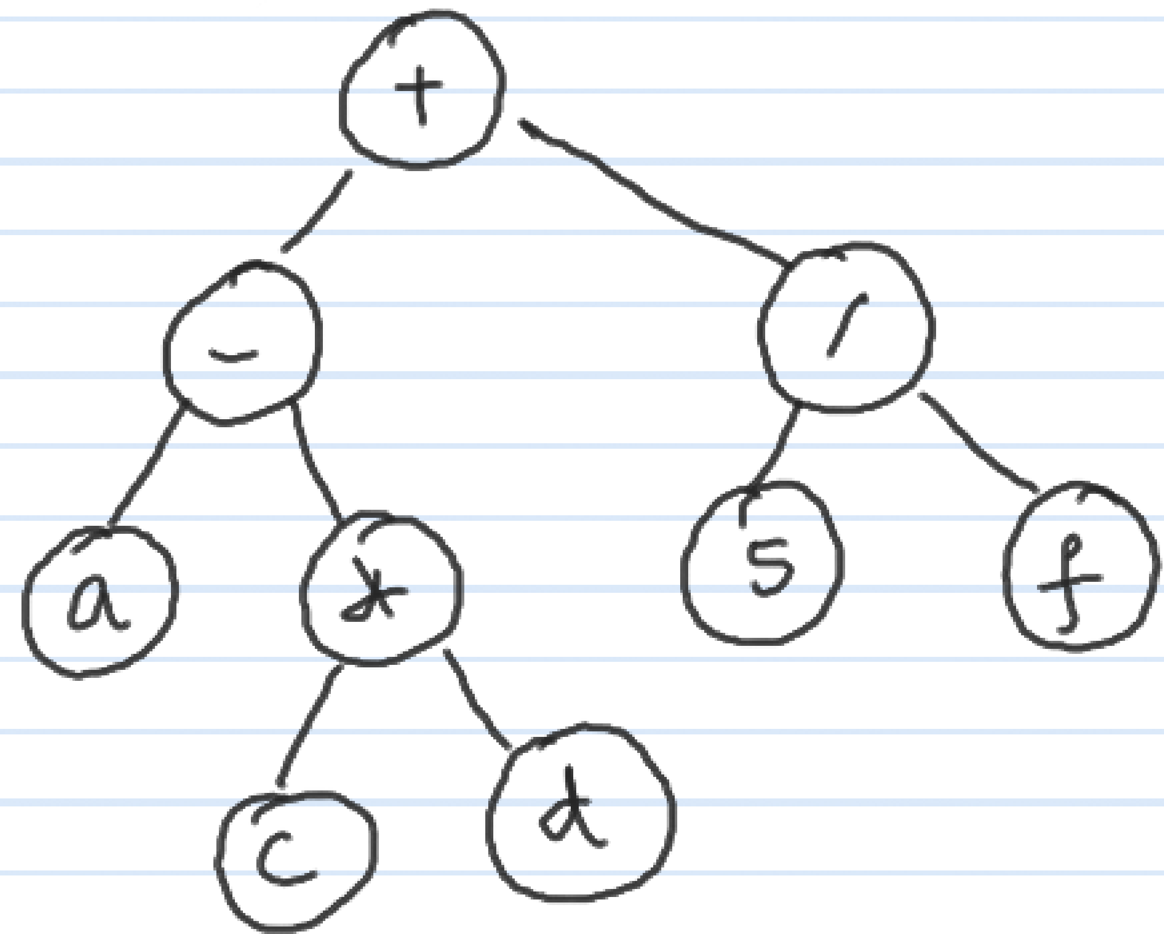


internal nodes :

leaves :

No parenthesis needed.

EXPRESSION TREES



POSTORDER TRAVERSAL OF TREE

LRN

$(a - (c * d)) + (5 / f)$

a c d * - 5 f / +

POSTFIX FORM → EXPRESSION TREE CONSTRUCTION.

CODING (not programming!)

Generate codes to transmit data

Like Linked lists trees are also recursive data structures

L
i
k
e
n

e
d
.
.

26 lower case
+
26 upper case
+
10 punctuations

46 symbols

6 bit code is sufficient

CODING (not programming!)

Generate codes to transmit data

Fixed length encoding (say 5 bits)

␣	00000
a	00001
b	00010
c	00011

A string of the form is interpreted

00001 010101 10001 00101 00000 10101

uniquely

CODING (not programming!)

Generate codes to transmit data

Fixed length encoding:

- simple and easy decoding
- n char, k length of code for each char

$n \times k$: size of total code

↳ can this be improved?

CODING (not programming!)

Generate codes to transmit data

Char	Freq
a	9
b	10
c	15
d	4
e	25
i	30

Reasonable to have shorter codes for frequently occurring characters.

---

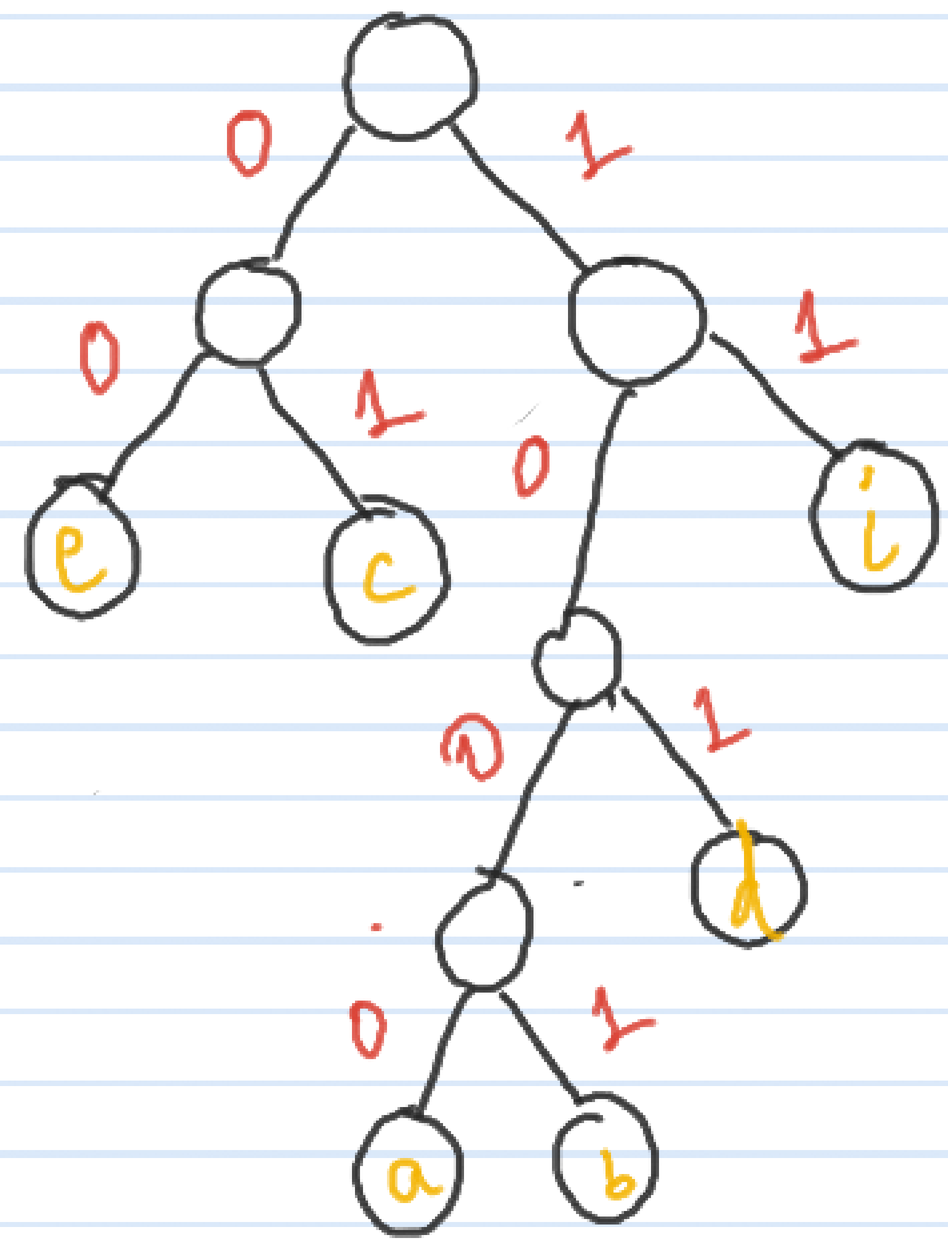
are there such codes?

how are they related to trees?

How to decode?

CODING (not programming!)

Generate codes to transmit data



a	1000
b	1001
c	01
d	101
e	00
i	11

1001 11 101

b i d

PREFIX CODES:

no word in code is prefix of any other word in the code

Fixed length codes: always satisfy this property

Variable length codes: may or may not

ex:

a : 00
b : 001
} not prefix

a : 01
b : 001
} ✓ prefix