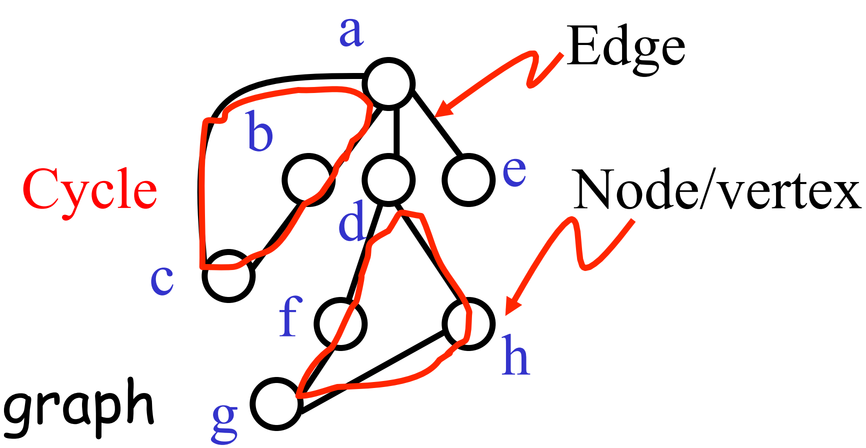
Outcome (2): Another data structure - tree

Tree

Do you remember what's a graph G(V, E) ?



1) Directed / Undirected graph

2) A path from v_i to v_j: a sequence $\langle v_i, v_{i+1}, \ldots v_{j-1}, v_j \rangle$ such that $(v_k, v_{k+1}) \in E$ (set of edges) for $i \leq k \leq j-1$

e.g. a d a b c a // length = 5

Length of a path = no. of edges in the sequence

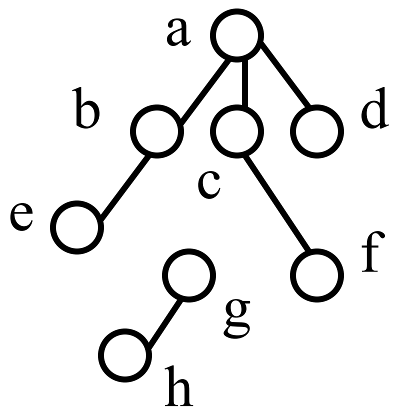
Remark: There is always a 0-length path from v to v.

A path is simple if all vertices in the path are distinct.

A path $\langle v_i, v_{i+1}, \ldots v_{j-1}, v_j \rangle$ forms a cycle if $v_i = v_j$

e.g. d f g h d // simple cycle

Not Connected

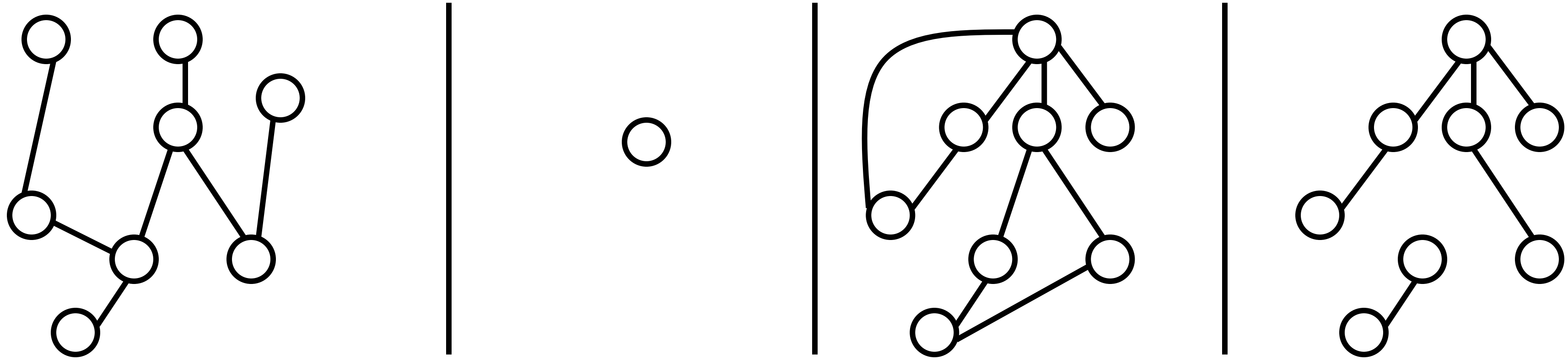


Vertices v_i and v_j are connected if there is a path from v_i to v_j.

e.g. a and f are connected; but a and g are not connected

An undirected graph is connected if every pair of vertices is connected.

A tree is a connected undirected graph with no cycles

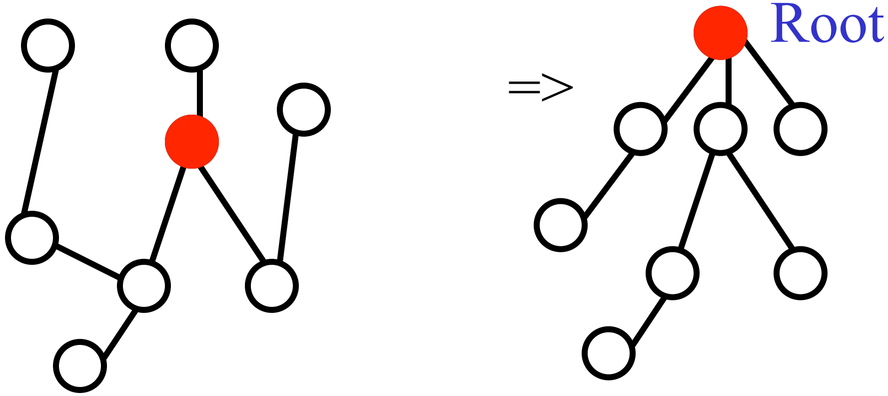


Remark: A tree with no nodes is called a null tree

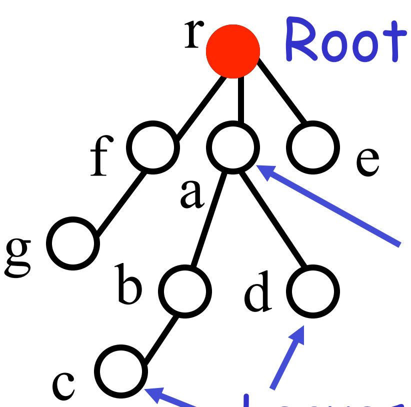
Guess: an undirected graph with no cycles, but not connected, what is it called?

Forest

Rooted (c.f. unrooted) Tree: One node is designed as root



Remark: Sometimes, we also will work on unrooted trees.



{ b is a parent of c
{ c is a child of b

Note: root has no parent

Internal nodes: with children

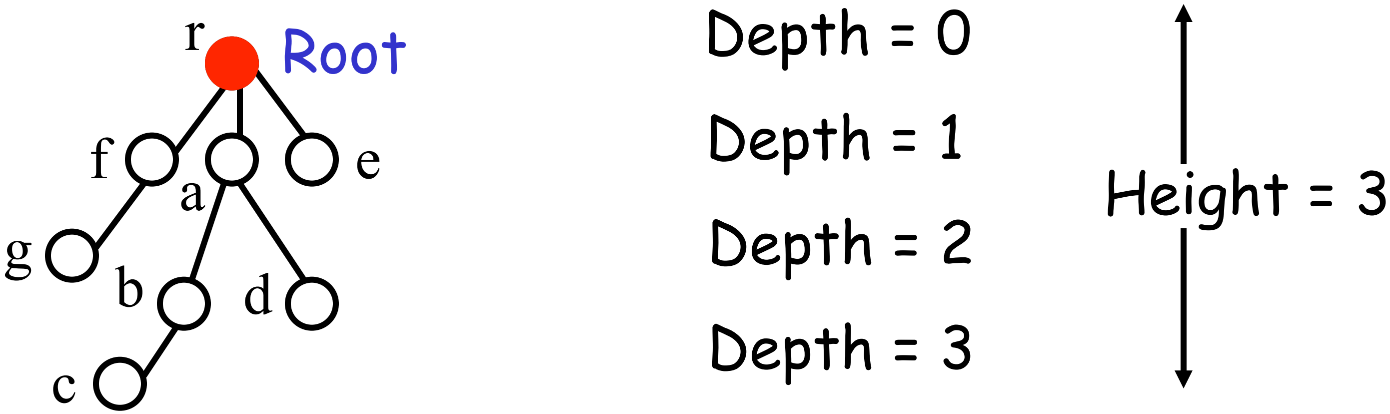
Leaves: without children

v_i is a parent of v_j ($i \neq j$) if $(v_i, v_j) \in E$ and v_i appears in the path from root to v_j
v_i is an ancestor of v_j ($i \neq j$) if v_i appears in the path from root to v_j

Siblings: nodes with same parent e.g. b and d

{ a (b) is an ancestor of c
{ c is a descendant of a (b)

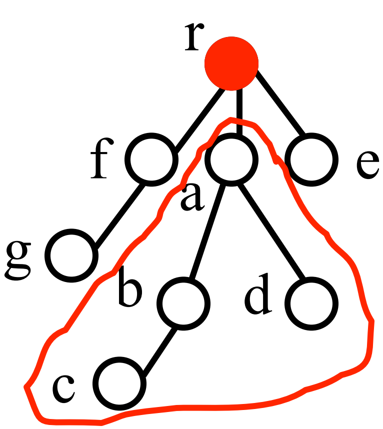
Remark: MIT book treats x is both an ancestor and descendant of x.



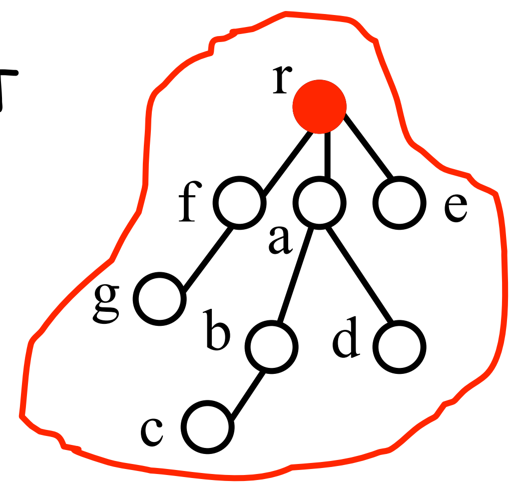
Degree of a node v (in a rooted tree): # of children v has
[c.f. Degree of a node in a graph / unrooted tree]

Depth of a node v: length of a path from root to v

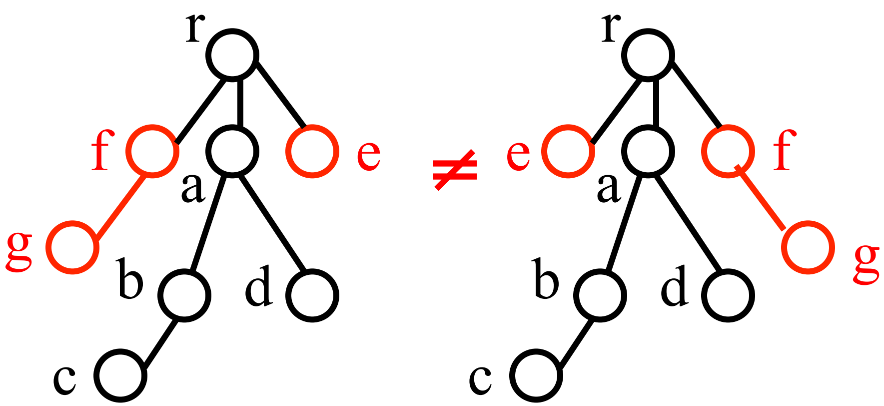
Height of a tree: max{depth of a node in the tree}



A subtree of a rooted tree T is a node v of T and all v's descendents and all edges connecting these nodes

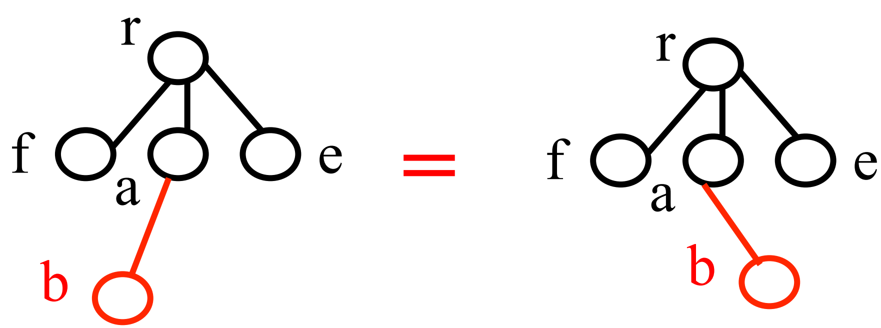


An ordered tree is a rooted tree where the children of each node are ordered (usually the drawing implies the order: from left to right)

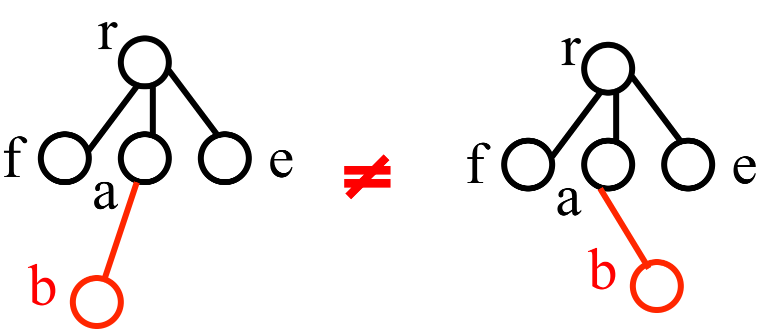


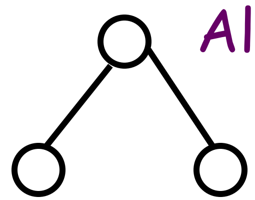
Then, you should know what an unordered tree is.

Considering ordered trees,



Sometimes, we want to treat them as different trees (positional trees).





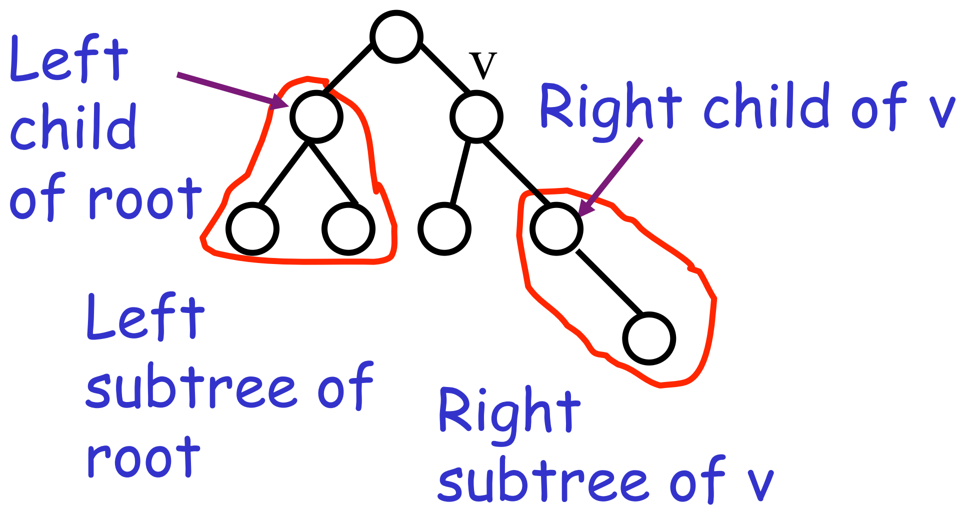
A rooted tree is called m-ary tree if every internal node has no more than m children.

Remark: Usually we will treat an m-ary tree as a positional tree.

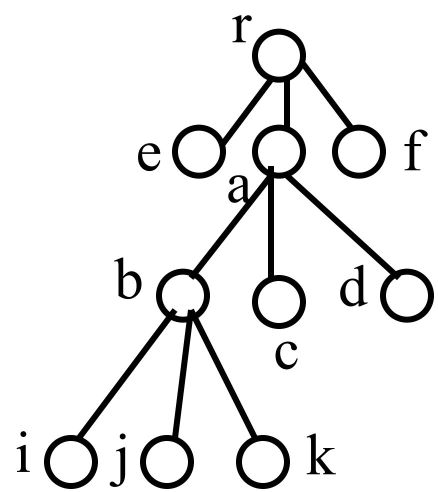
If m = 2, then the tree is called a binary tree.
[Reminder: treat it as a positional tree.]

Note:Usually omitted (clear from the context)

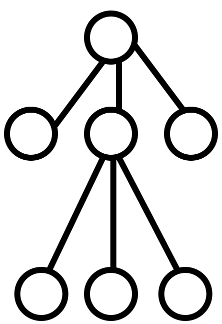
Positional binary tree:



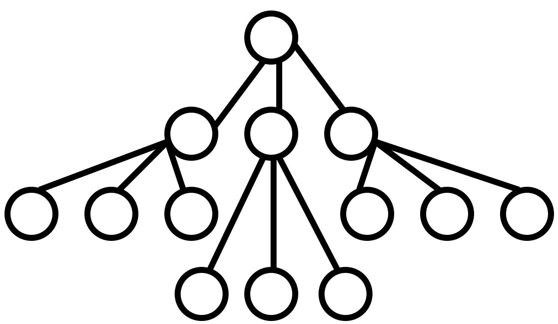
Full m-ary tree: every internal node has exactly m children



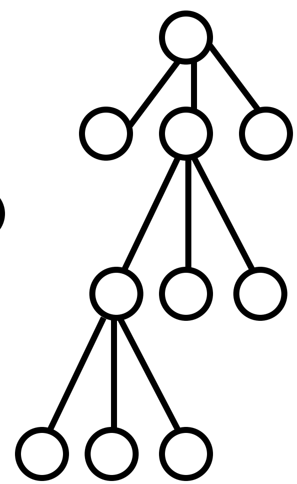
Complete m-ary tree: all leaves are of the same depth and all internal nodes with degree m



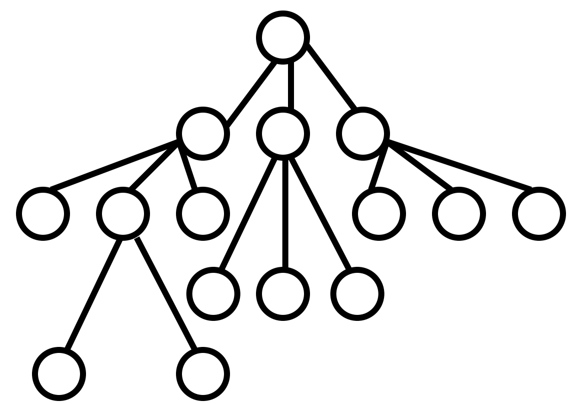
Full



Complete

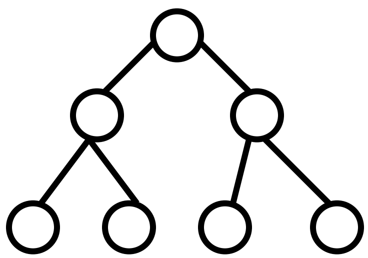


Full



Not full nor complete

Q: If a complete binary tree has n leaves,
what's the total number of nodes in the tree?



$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

Ans: 2n-1

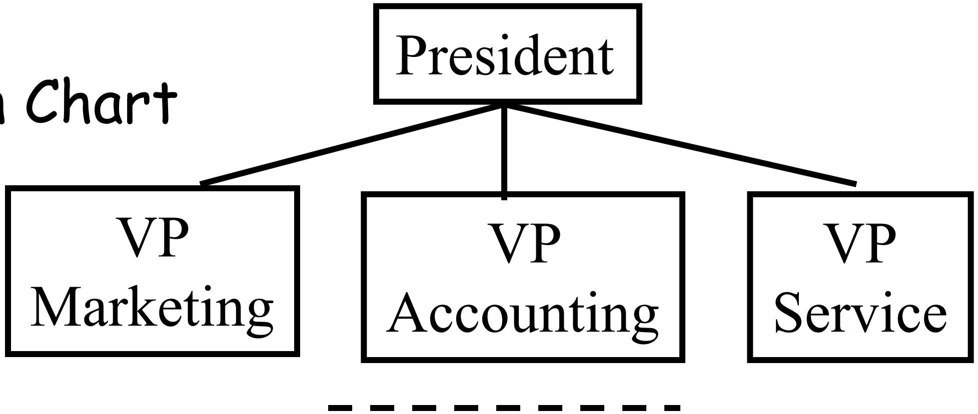
Remark: # of internal nodes = n – 1; # of leaves is about 1/2 of total no. of nodes in the tree.

Q: What is the number of nodes in a complete m-ary tree with height h?

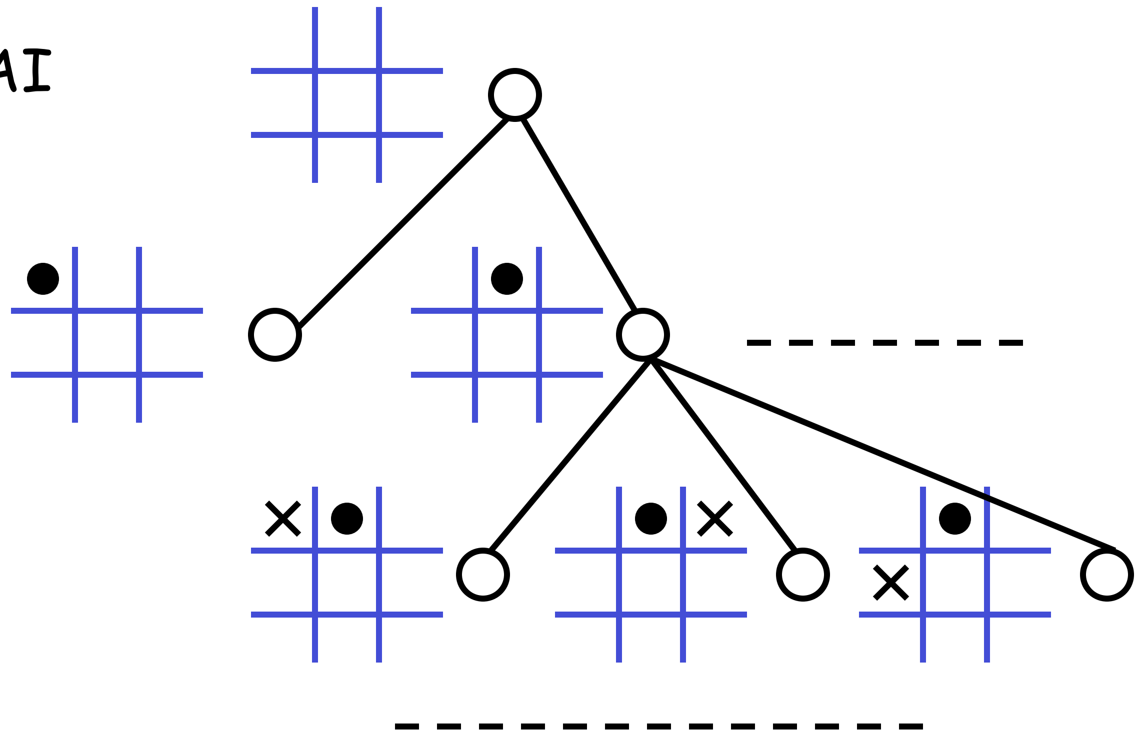
$$1 + m + m^2 + m^3 + ... + m^h = (m^{h+1} - 1)/(m - 1)$$

Some applications of trees

Organization Chart

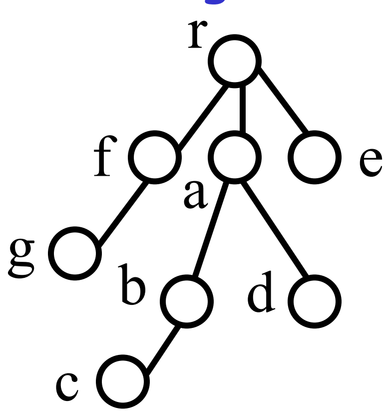


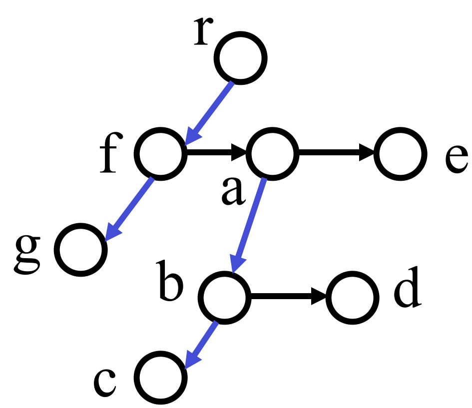
Game Tree in AI



Representation

Rooted tree with "unbounded" branching





The left-child right-sibling representation

```
struct node {
    element e;
    node *left-child;
    node *right-sibling;
}
```

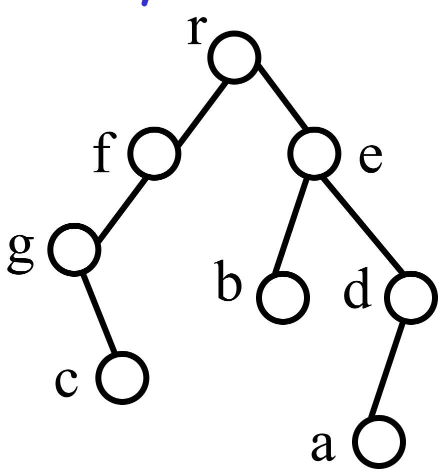
Note: sometimes, it helps to have a parent pointer

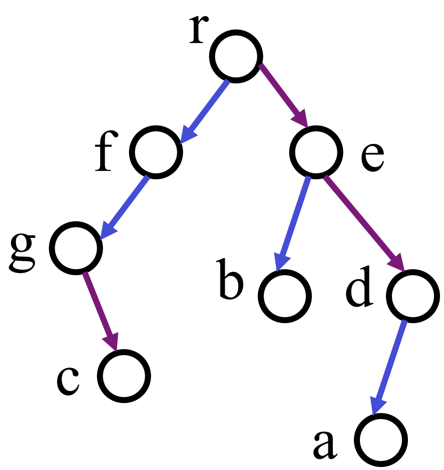
If we know that the degree of nodes $\leq$ k,

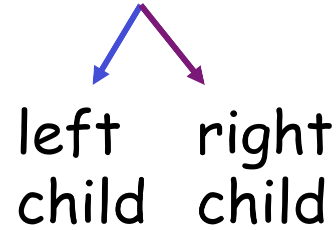
```
struct node {
    element e;
    node *child1;
    node *child2;
    .....
    node *childk;
}
```

Waste of storage if many nodes do not have k children

Binary tree








```
struct node {
    element e;
    node *left-child;
    node *right-child;
}
```

Pointer implementation

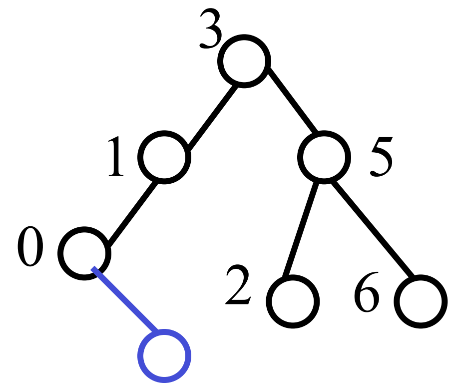
Do you know that we can use array to simulate pointer implementation?

Binary tree

Array implementation

	left	right
0	-1	-1 → 7
1	0	-1
2	-1	-1
3	1	5
4	-1	-1
5	2	6
6	-1	-1
7	-1	-1
	⋮	⋮

Root = 3



Note: need another array to store the elements; link up free nodes etc.

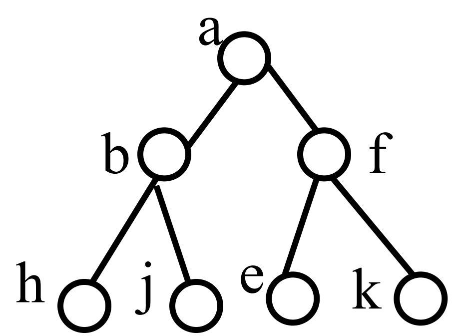
Both schemes use O(n) storage where n: # of nodes

How about a complete binary tree?

Use a single array!

0	1	2	3	4	5	6	7	8	9	10	11
a	b	f	h	j	e	k					

↑ Root

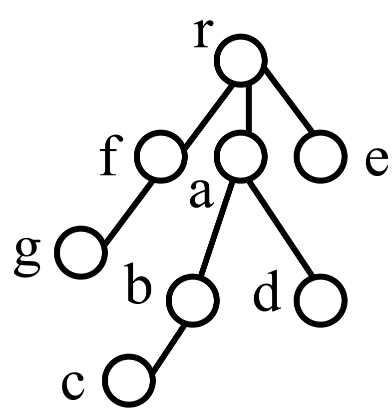


left child of T[i] = T[2i+1]
right child of T[i] = T[2i+2]

Just like in Graph, we want to visit nodes in a systematic way.

1 Preorder traversal

Staring from root, visit the node, then repeat the same procedure in each of its subtrees one by one following the order of the subtrees



Note: both BFS, DFS can be applied to trees

Preorder:
r f g a b c d e

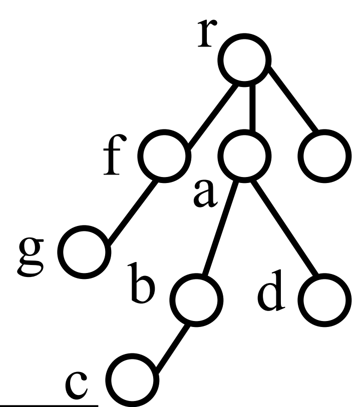
Preorder(T) {
  if (T = null) return;
  visit(root(T));
  /* Let T_1, T_2,... T_k be subtrees of root(T) from left to right */
  for i = 1 to k do
    Preorder(T_i);
}

You can image that visit( ) is to do some processing on the node

Can you guess what are postorder and inorder traversal?

2 Postorder traversal

```
Postorder(T) {
   if (T = null) return;
   /* Let T_1, T_2,... T_k be subtrees of root(T)
      from left to right */
   for i = 1 to k do
        Postorder(T_i);
   visit(root(T));
}
```



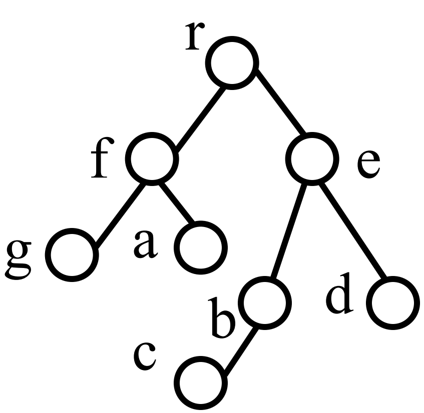
Postorder:
g f c b d a e r

3 Inorder traversal

```
Inorder(T) {  // for binary tree only
   if (T = null) return;
   /* Let T_l, T_r be left and right subtrees of
      root(T) respectively */
    Inorder(T_l);
    visit(root(T));
    Inorder(T_r);
}
```

Remark: Some extend the inorder traversal for general degree.

```
        Inorder(T_1);
        visit(root(T));
        for i = 2 to k do
           Inorder(T_i);
```

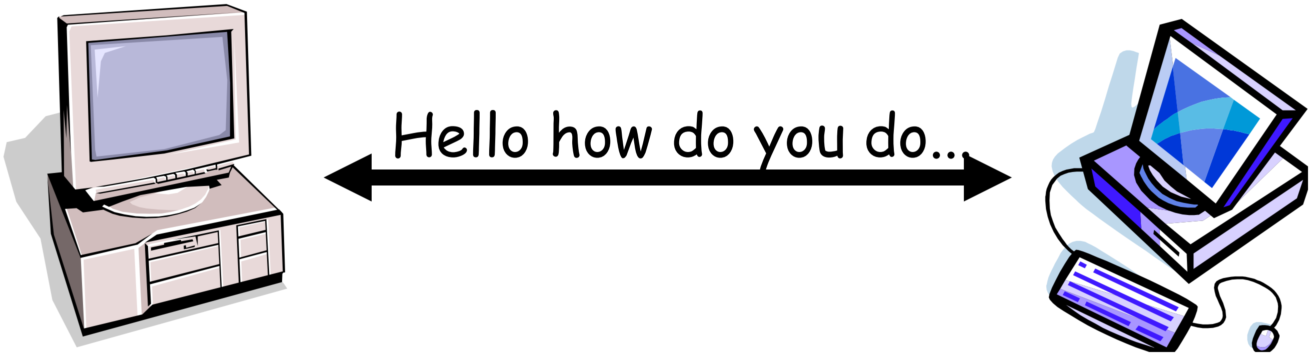


Inorder:
g f a r c b e d

Time complexity?
Ans: All are $O(n)$

Application: A Data Compression Problem – Huffman code

Motivation



You can use ASCII code for each letter (8 bits each): Total = 120 bits

Note: some letters appear more often than others, can we use fewer bits for more frequent letters?

Letter	Frequency	Code
d	2	00
e	1	000
h	2	01
l	2	10
o	5	0
u	1	001
w	1	010
y	1	011

Total: 29 bits

codewords

Idea seems ok, but one problem in decoding the message.

e.g. 00101
ohh or uh?

How about we make sure that the code for a character will not be the beginning (prefix) of another code?

Prefix Code

Letter	Frequency	Code
d	2	00
e	1	000
h	2	01
l	2	10
o	5	0
u	1	001
w	1	010
y	1	011

Not a prefix code

Advantages of prefix code: simplify the decoding; avoid ambiguity

Letter	Code
a	0
b	101
c	100
d	111
e	1101
f	1100

If message is: 1101101101100

After decoding: e b b c (no ambiguity)

To conclude: the idea of using fewer bits to represent more frequent symbol with the additional of the prefix property should work

How to design a prefix code such that fewer bits are used for more frequent letters? (Huffman Code: optimal prefix code)

Optimal

$$\frac{\sum \ell_i f_i}{\sum f_i} \quad \text{is a minimum}$$

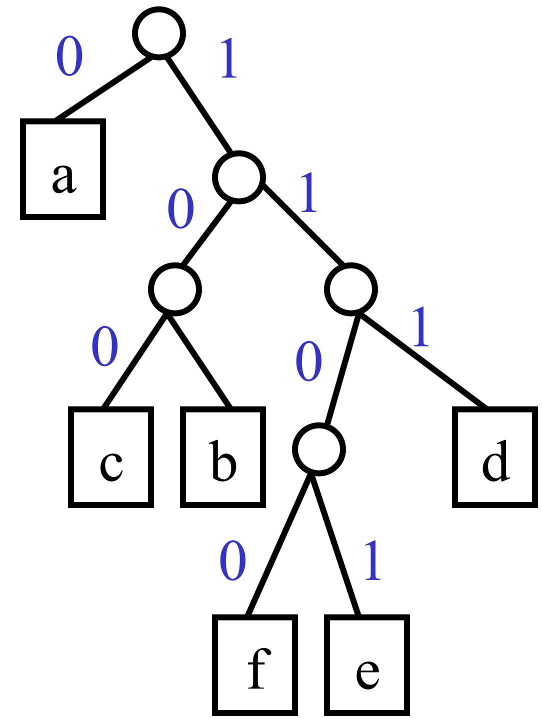
where l_i is length of codeword;
f_i is the corresponding frequency

Example
01, 000, 001, 100, 101
Not optimal no matter what f_i's are

A better one:
01, 10, 11, 000, 001

Letter	Code
a	0
b	101
c	100
d	111
e	1101
f	1100

This encoding can be represented by a (positional) binary tree!



Observation:
Less frequent letters at the lower part (with greater depths) of the tree

Constructing a Huffman Code

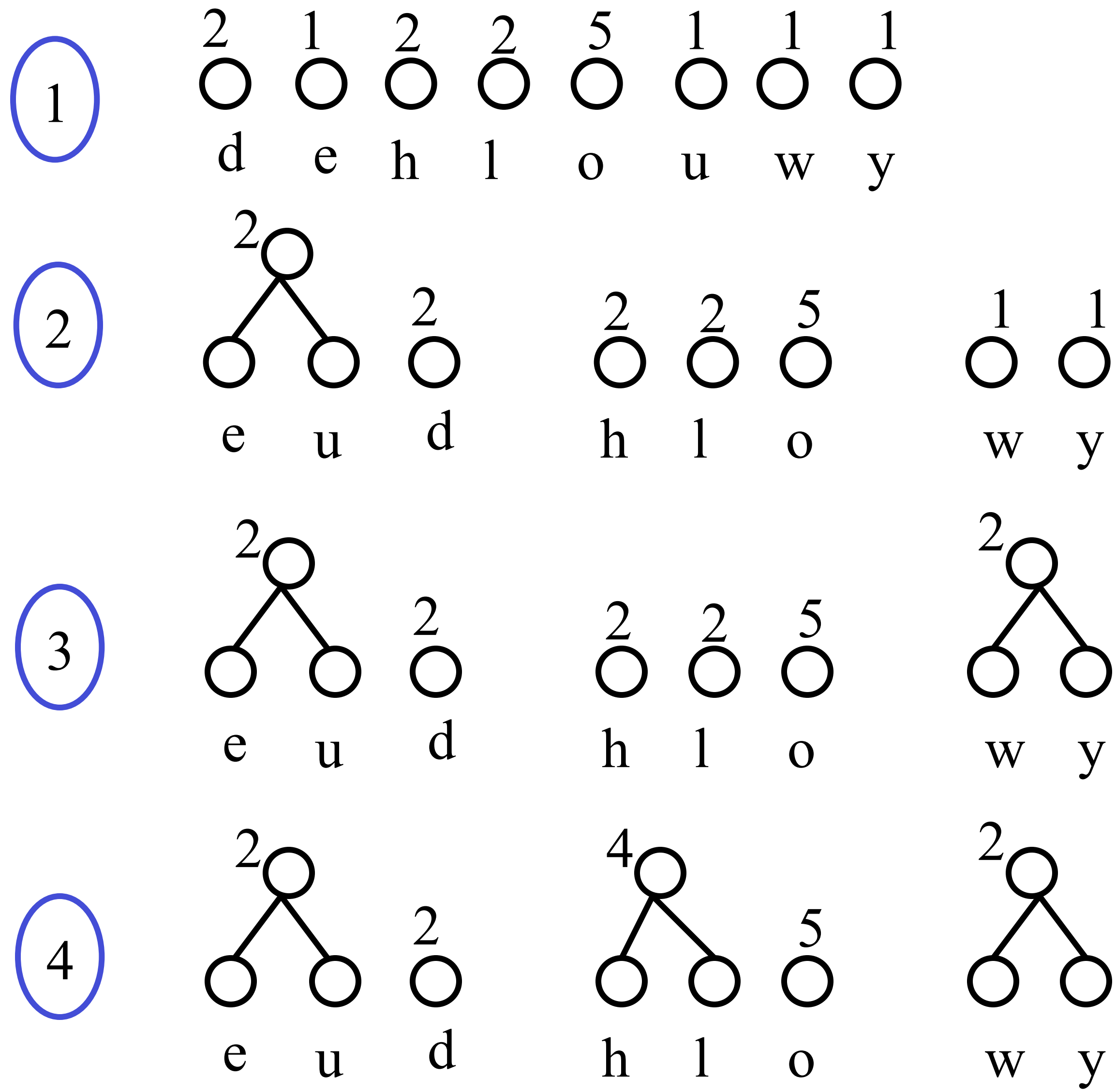
Build a binary tree using a bottom-up approach

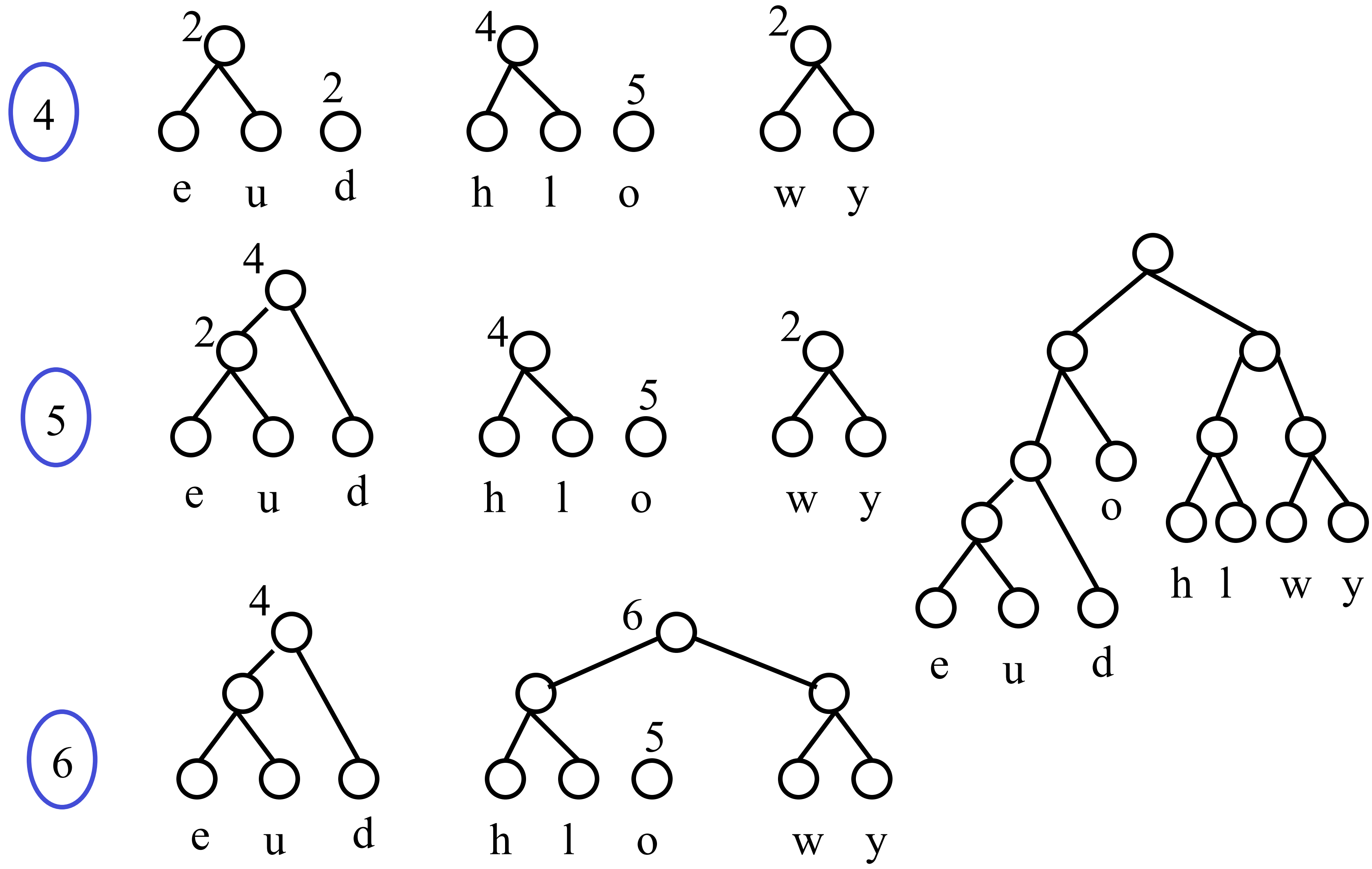
- Create n trees, each having one node representing one letter (symbol)

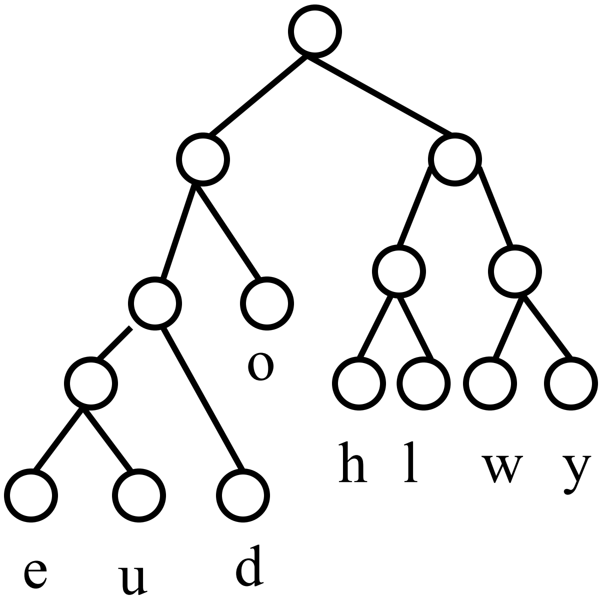
- Let the weight of a tree be total frequencies of all symbols represented by its leaves

- Repeat the following until we got only one tree.
    1. Pick two trees T1, T2 with the smallest weights
    2. Create a new tree T with T1 and T2 as the left and right subtrees respectively
    3. Set weight of T = weight of T1 + weight of T2

Letter	Frequency
d	2
e	1
h	2
l	2
o	5
u	1
w	1
y	1




4
2
2
e u d
4
5
h l o
2
w y
5
4
2
e u d
4
5
h l o
2
w y
o
h l w y
e u d
6
4
e u d
6
5
h l o
w y



d: 001
e: 0000
h: 100
l: 110
o: 01
u: 0001
w: 110
y: 111

Letter	Frequency
d	2
e	1
h	2
l	2
o	5
u	1
w	1
y	1

$$\frac{\sum l_i f_i}{\sum f_i} = \frac{2x5 + (3+3+3)2 + (3+3+4+4)1}{2+1+2+2+5+1+1+1}$$

$= 42/15$

Another application: Binary search tree (will be discussed later)