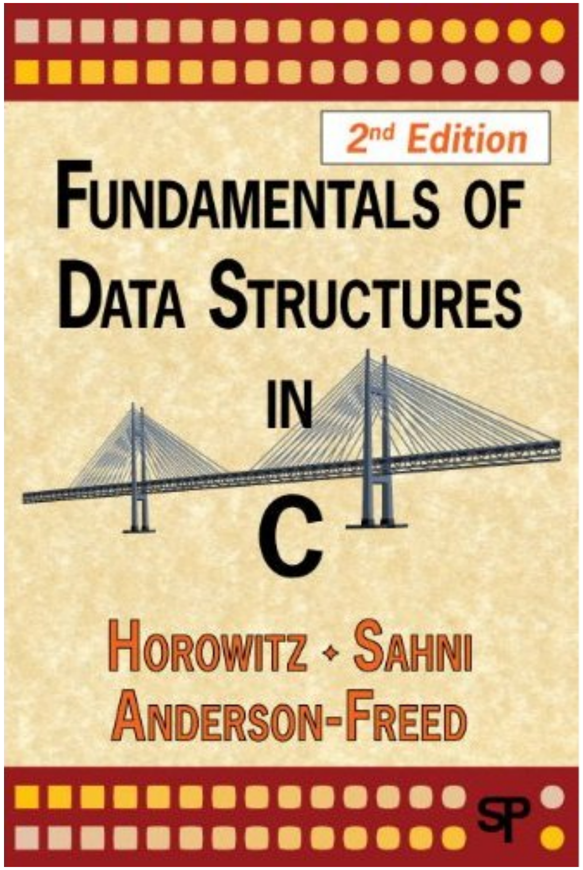
TREE 1

Michael Tsai

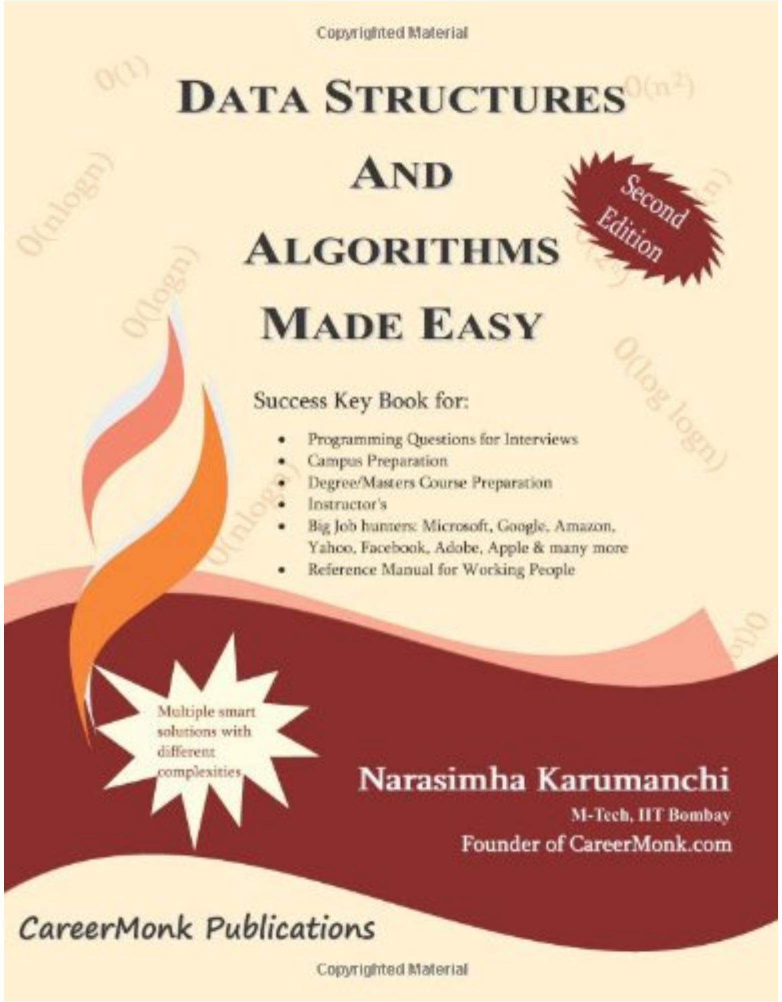
2017/04/11

Reference



- Fundamentals of Data Structures in C, 2nd Edition, 2008
- Chapter 5
- Horowitz, Sahni, and Anderson-Freed

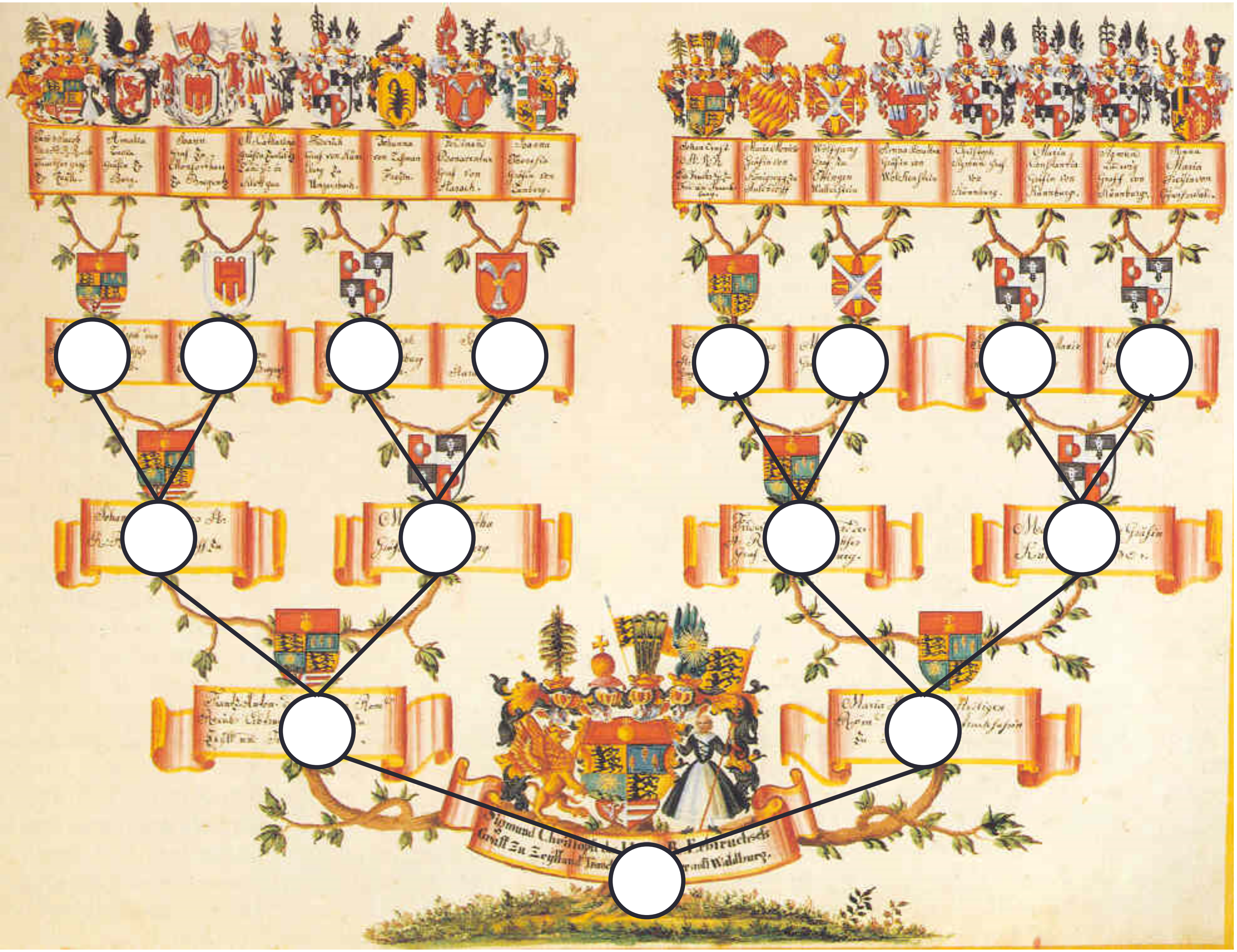
Reference



- Data Structures and Algorithms Made Easy, Second Edition, 2011, CareerMonk Publications, by Karumanchi
- Chapter 6.11

樹

The family tree of Sigmund Christoph von Waldburg-Zeil-Trauchburg

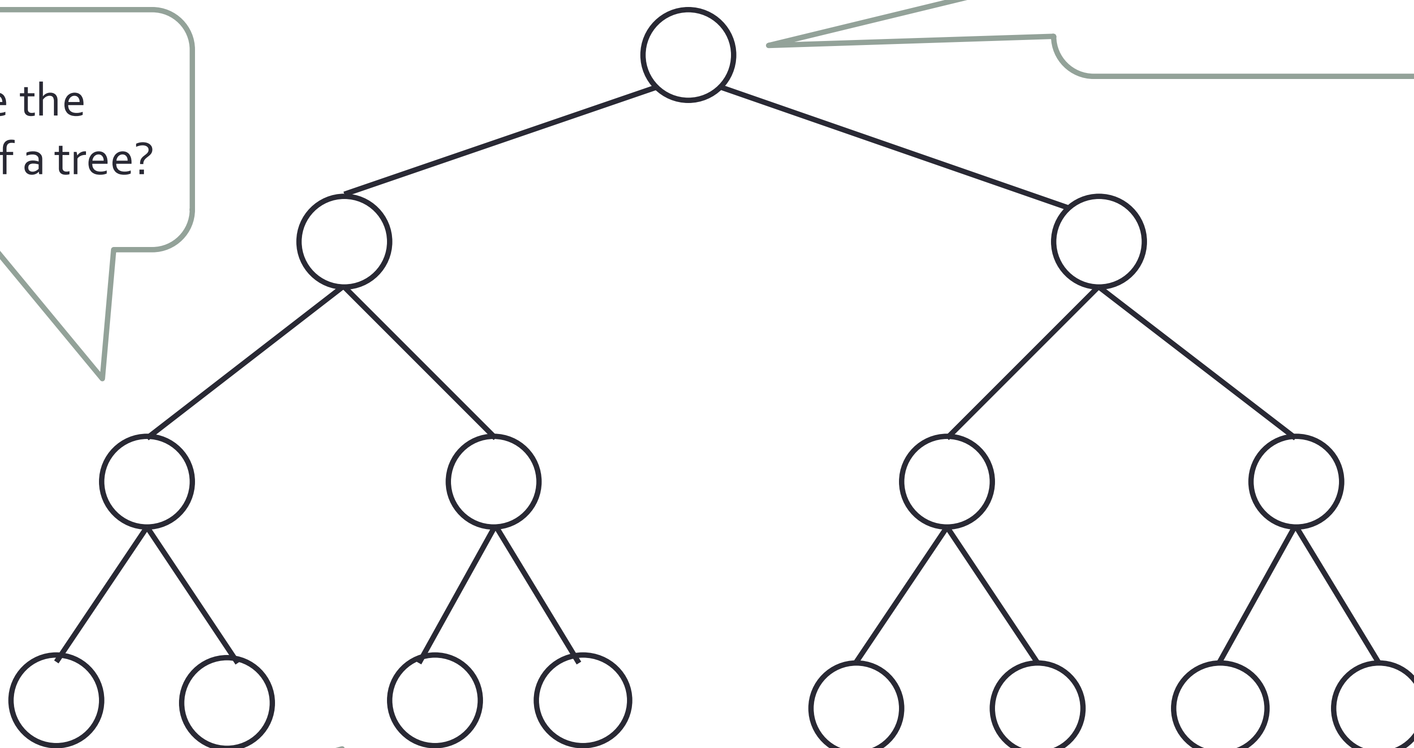


http://www.ahneninfo.com/de/ahnentafel.htm

In the CS world, we usually draw the tree upside-down.

Root

What are the properties of a tree?



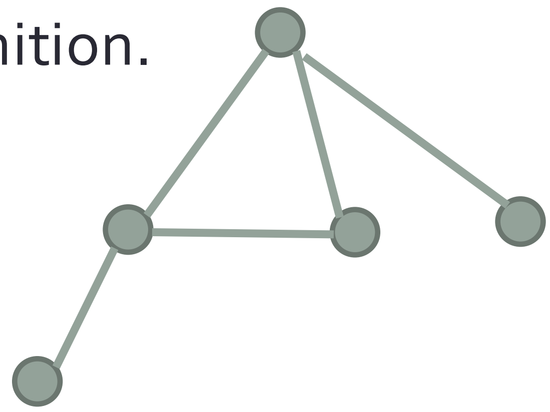
A nonlinear, hierarchical way to represent data.

Leaves

Definition

- Definition: A **tree** is a finite set of **one or more nodes** such that
- (1) There is a specially designated node called the **root**.
- (2) The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, T_2, \ldots, T_n$, where each of these sets is a tree.
- (3) $T_1, T_2, \ldots, T_n$ are called the **subtrees** of the root.

- Note that the above is a recursive definition.
- A node with no subtree, is it a tree?
- Is "No node" (null) a tree?
- Is the "graph" on the right a tree?



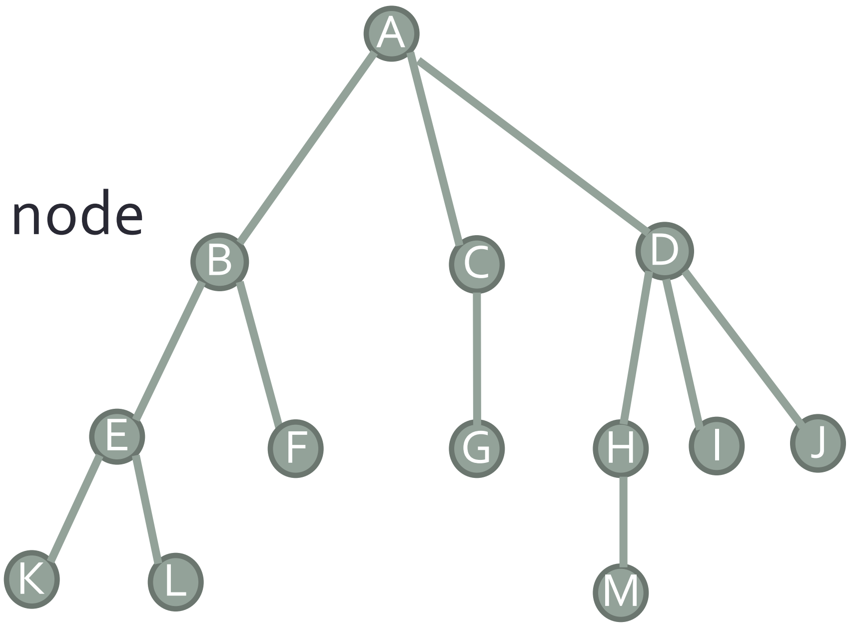
Tree Dictionary

- **Root**
- **Node/Edge (branch)**
- **Degree (of a node):**

    The number of subtrees of a node
- **Leaf/Terminal node:**

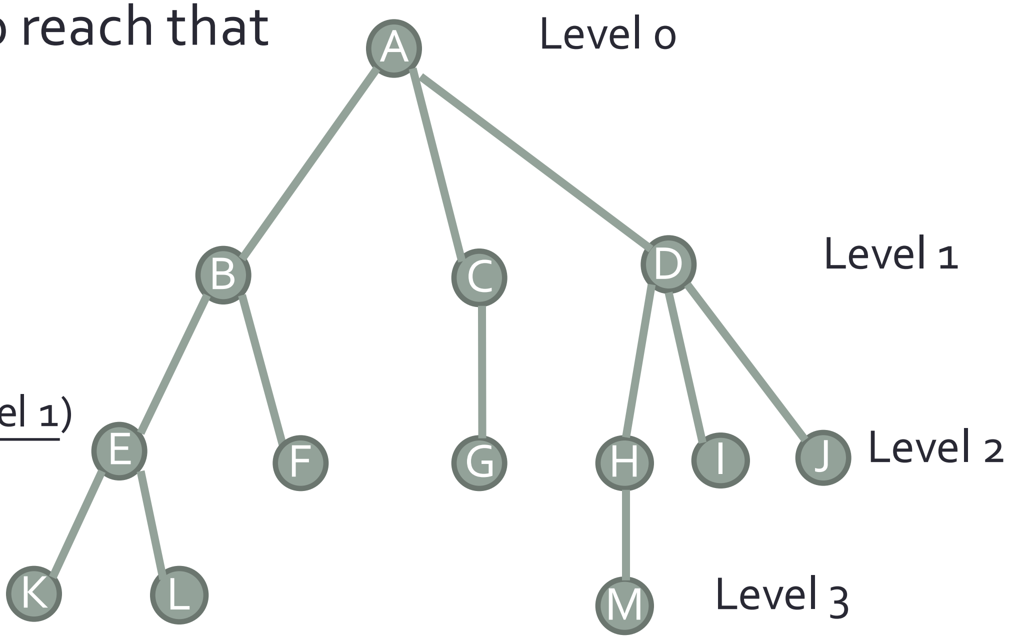
    its degree=0
- **Parent/Children**
- **Siblings**

    they have the same parent.
- **Ancestors/Descendants**



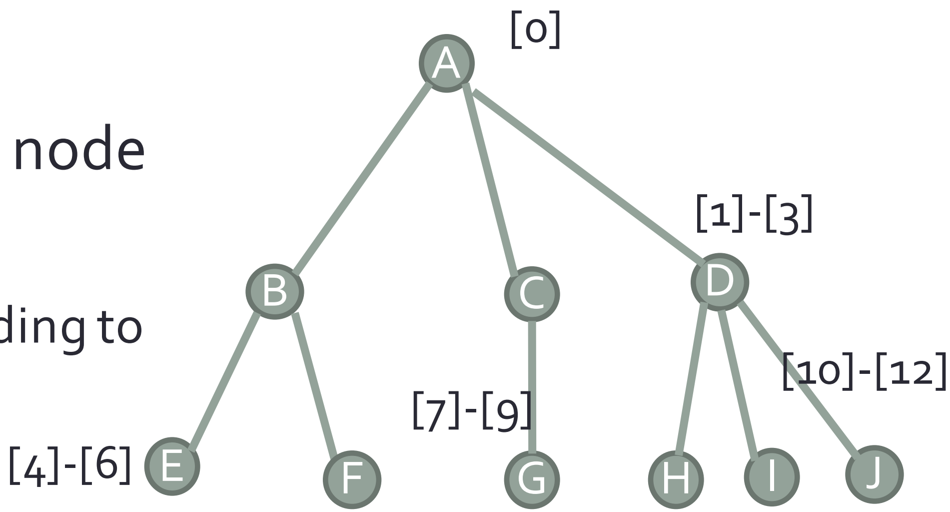
Tree Dictionary

- **Level/depth (of a node):**
  The number of branch to reach that node from the root node.
  (i.e., root is at level 0)
- **Height (of a tree):**
  The number of levels in a tree
  (Note that some definitions start from level 1)
- **Size (of a tree):**
  The number of nodes in a tree
- **Weight (of a tree):**
  The number of leaves in a tree
- **Degree (of a tree):**
  The maximum degree of any node in a tree



Representing a tree with array

- What to store?
- The data associated with each node
- Array method
  - Store the data sequentially according to the level of the node



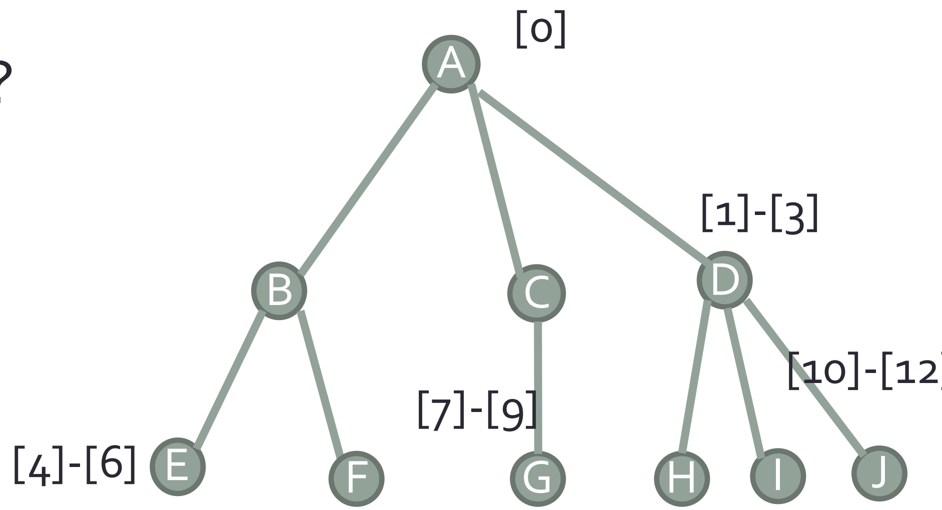
Max degree of the tree=3

- In the array:
- How to find the parent of a node?
- How to find the children of a node?

0	1	2	3	4	5	6	7	8	9	10	11	12
A	B	C	D	E	F		G			H	I	J

Representing a tree with array

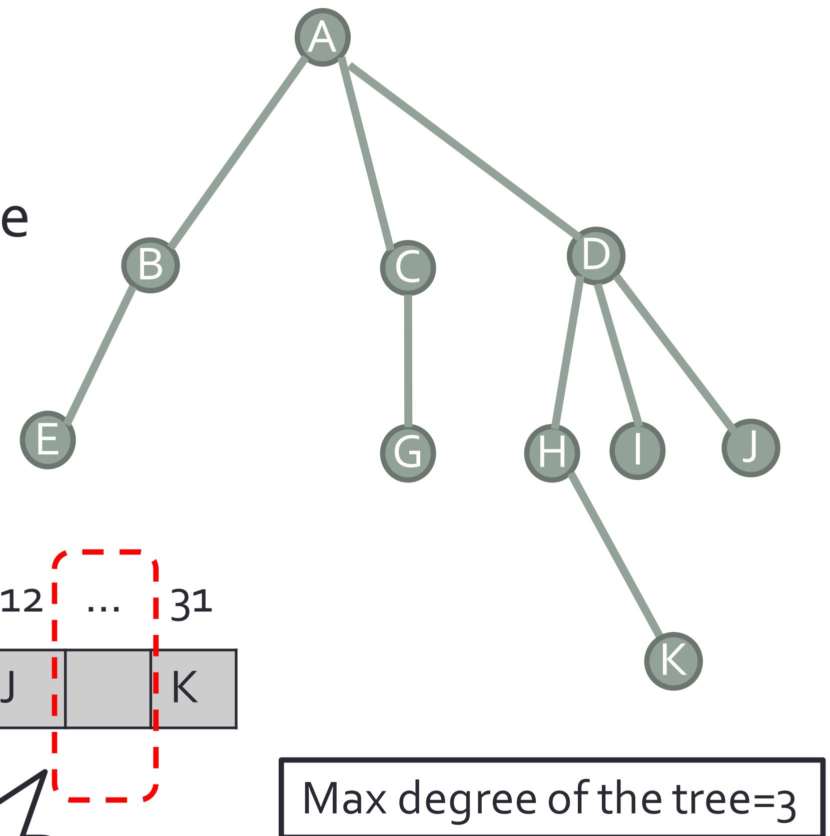
- Assume degree (of the tree)=d (The example on the right uses d=3)
- How to find the parent of a node?
- Observation:
  For node with index i, its parent's index is $\lfloor (i-1)/d \rfloor$
- How to find the children of a node?
- Observation:
  For node with index i, its children's indices range from ???

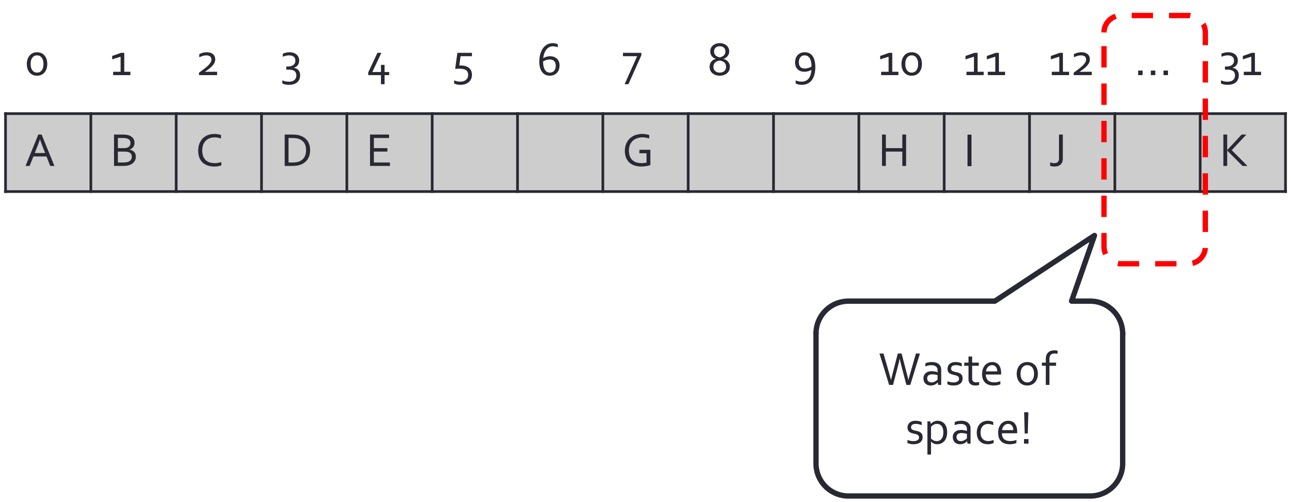


Representing a tree with array

- Downside?

- If there are many nodes with degree less than d,
- then we waste a lot of space in the array.

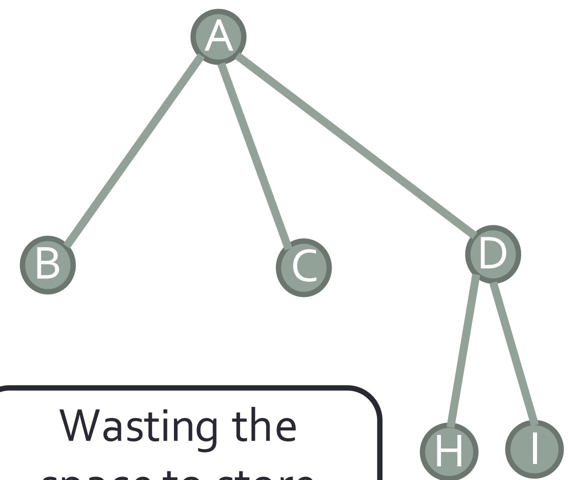


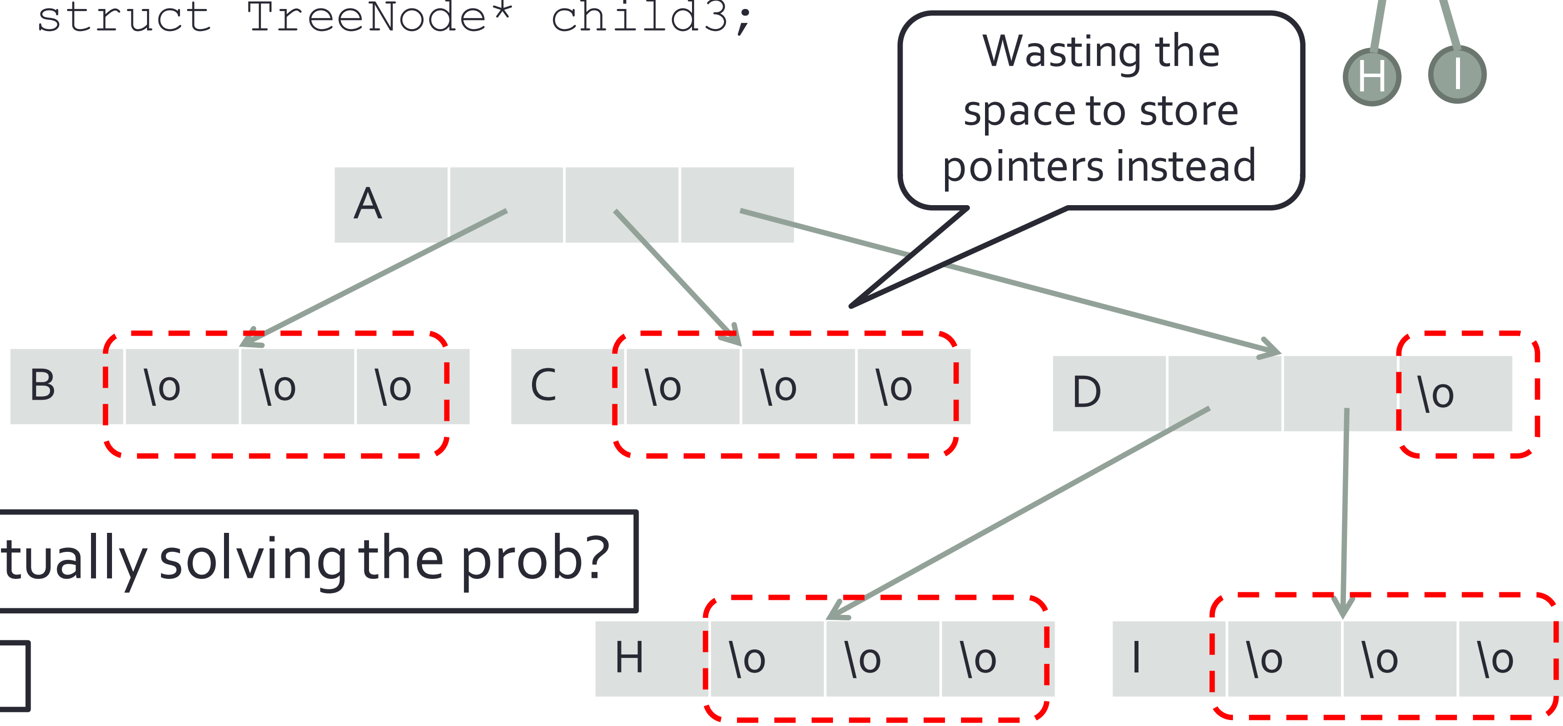


Representing a tree with linked structure

- Assume degree =3

```
struct TreeNode{
        char data;
        struct TreeNode* child1;
        struct TreeNode* child2;
        struct TreeNode* child3;
};
```



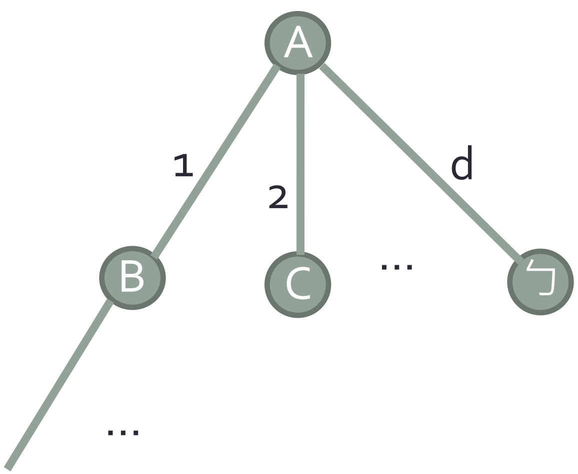


Not actually solving the prob?

\o == NULL

Representing a tree with linked structure

Data	child 1	child 2	child 3	…	child k



- Assume degree of tree = d, size of the tree = n
- How many pointers are null?
- Total number of pointers: nd
- Number of branch? n-1.
- $nd - (n-1) = n(d-1) + 1$

Better if smaller!

Number of wasted pointers

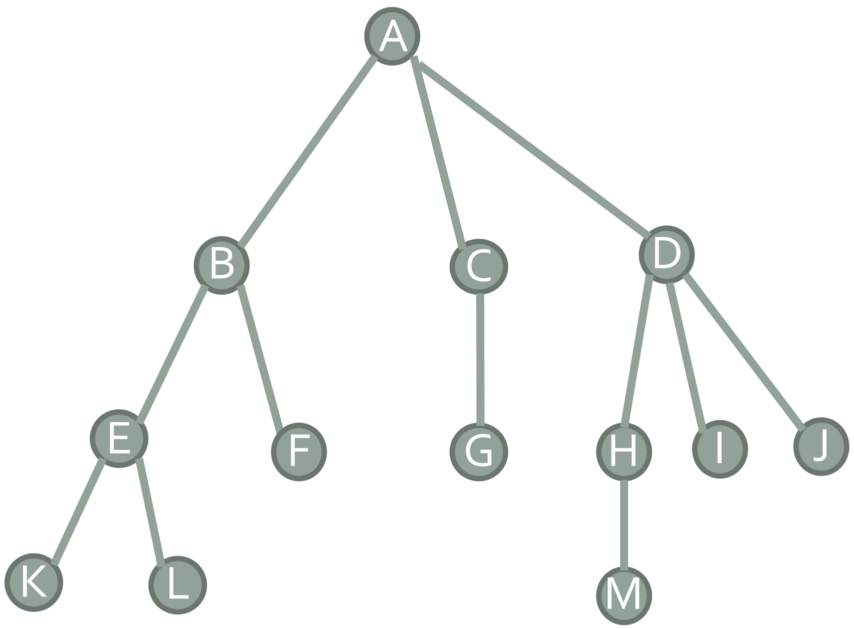
左小孩-右兄弟姊妹 表示法

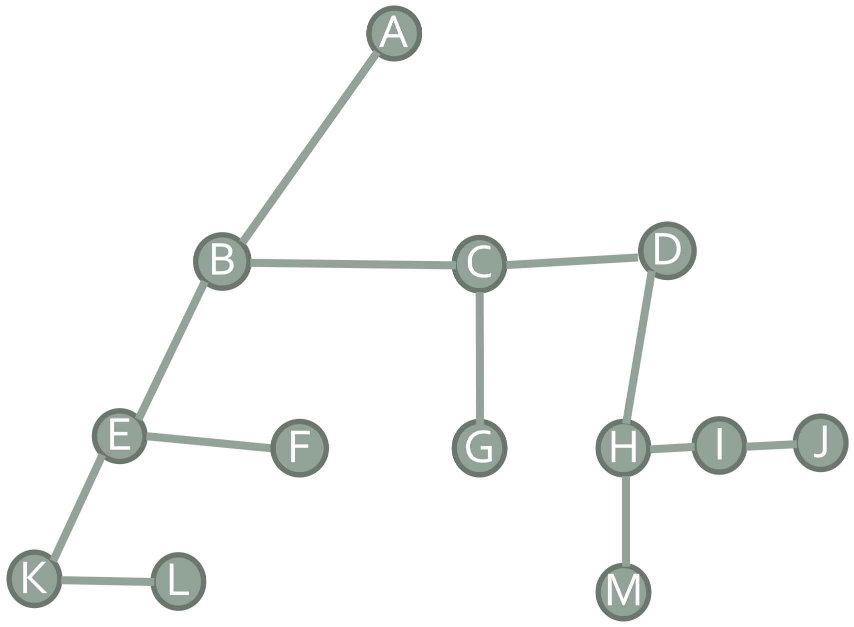
- Left child-right sibling representation

Data	left child	Right sibling

- Inspired by observations:
- 1. Each node has a leftmost child (是廢話)
- 2. Each node has only a immediately-right sibling (也是廢話)

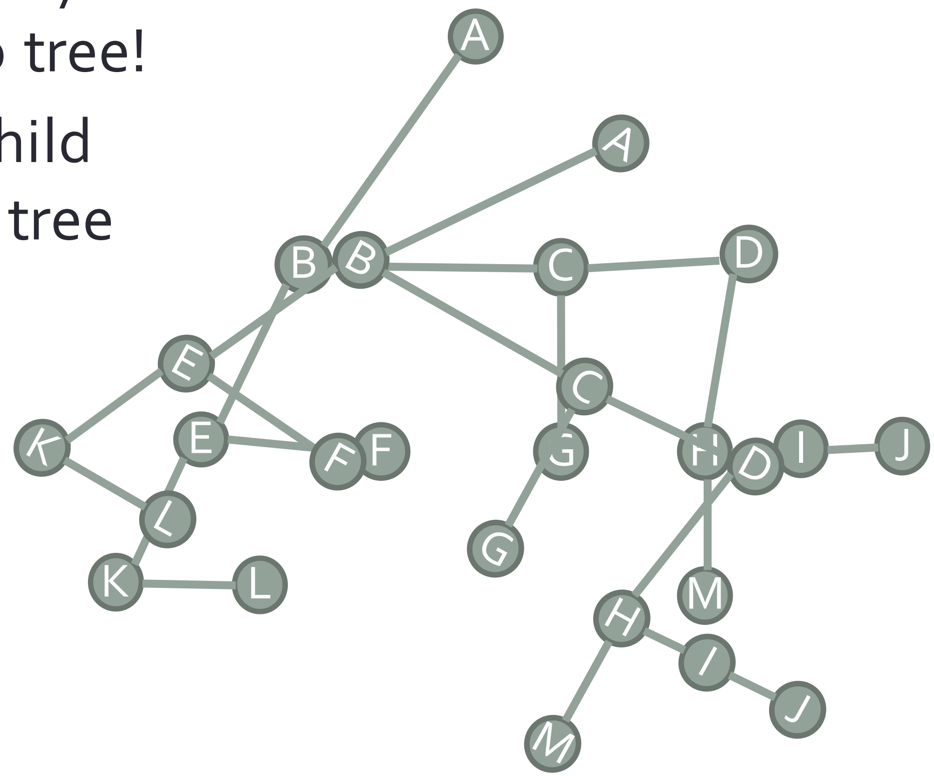
Converting to LCRS tree





LCRS tree

- It's always a **degree-two** tree!
- Converting to from an arbitrary-degree tree to a **degree-two** tree!
- Root does not have a right child (because root in the original tree does not have a sibling)



Binary Tree

- Definition: A **binary tree** is a finite set of nodes that is either **empty** or **consists of a root and two disjoint binary trees** called the left subtree and the right subtree.

- According to this definition:
- Note: “null” (no node) is a valid tree.
- Note: the order of the children (left or right) is meaningful.

一些證明

在level i的node數目最多為$2^i, i \geq 0$
(root在level o)

- 證明: 用歸納法

- i=o時, 為root那一層, 所以只有一個node, 也就是最多有$2^0 = 1$個node. (成立)
- 假設i=k-1的時候成立→level k-1最多有2^{k-1}個node
- 那麼i=k的時候, 最多有幾個node?
- 因為是binary tree, 所以每個node最多有兩個children
- 因此最多有$2^{k-1+1} = 2^k$node (得證)

兩些證明(誤)

一棵height為k的binary tree, 最多有2^k-1個node, $k \geq 1.$

- 證明:
- 利用前一頁的結果
- 則總共node數目最多為
- $\sum_0^{k-1} 2^i = \frac{2^k-1}{2-1} = 2^k - 1.$ 喔耶.

三些證明(誤)

對於任何不是空的binary tree, 假設n_0為leaf node數目, n_2為degree 2的node數目, 則$n_0 = n_2 + 1$.

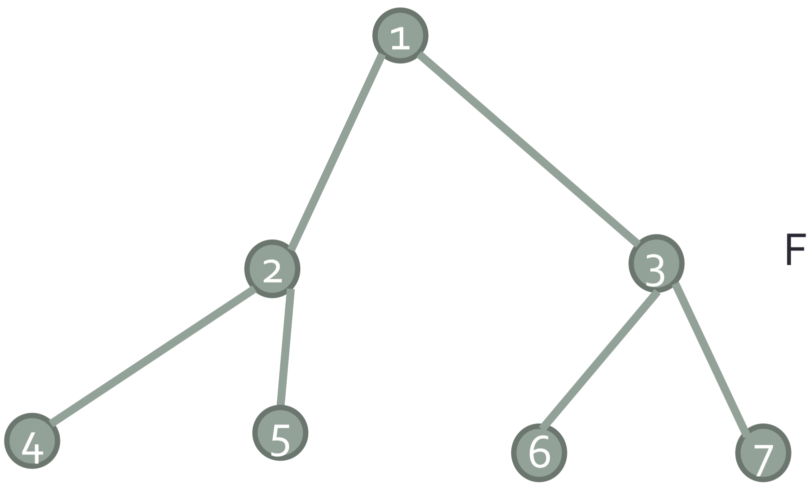
- 證明:
- 假設n為所有node數目, n_1為degree 1的node數目,
- 則$n = n_0 + n_1 + n_2$.　(1)

- 假設B為branch的數目, 則$B = n_1 + 2n_2$.　(2)
- 而且$n = B + 1$ (3). (只有root沒有往上連到parent的branch, 其他的node正好每個人一個)
- (2)代入(3)得 $n = n_1 + 2n_2 + 1$　(4)
- (4)減(1) 得 $n_0 = n_2 + 1$. 喔耶.

Full binary tree

- Definition: a **full binary tree** of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 1$.

- In other words, the maximum size tree of depth k (full)
- All nodes except the leaves have two children

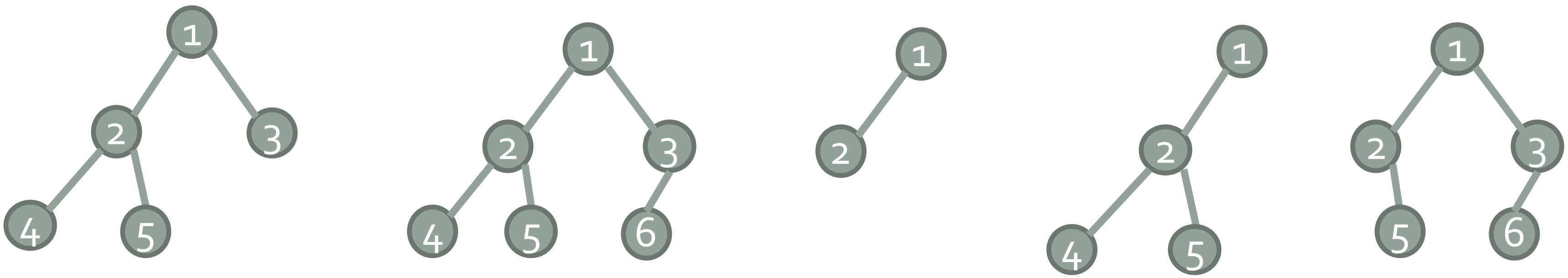


Full binary tree of depth 3

Complete binary tree

- Definition: A binary tree with n nodes and depth k is complete iff its nodes correspond to **the nodes numbered from 1 to n in the full binary tree of depth k.**

- All leaves at level k-1 and k-2 (the last two levels) have no "missing ones"



Yes Yes Yes No No

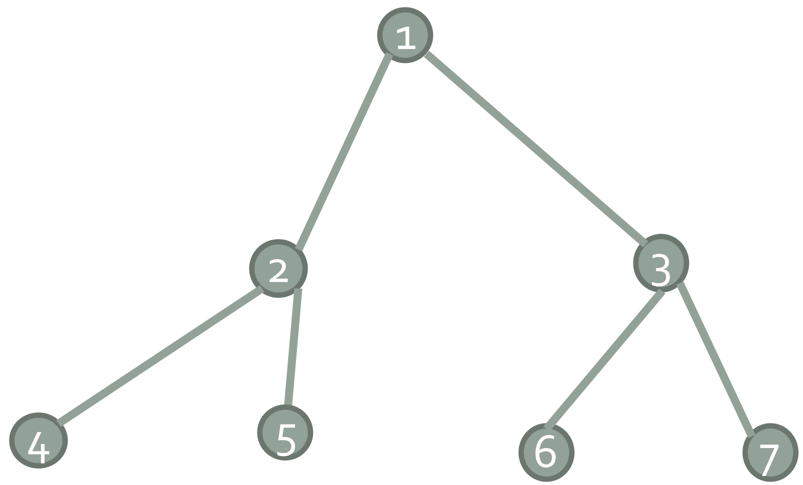
Height of a complete binary tree

- Exercise: if a complete binary tree has n node, what's the height of the tree?

- Hint: a full binary tree of height k has $2^{k} - 1$ nodes
- Hint: what is the minimum size of such a tree in terms of k?
- Hint: use a ceiling or floor function

Binary Tree Traversal

- How do we traverse each node in the given binary tree?

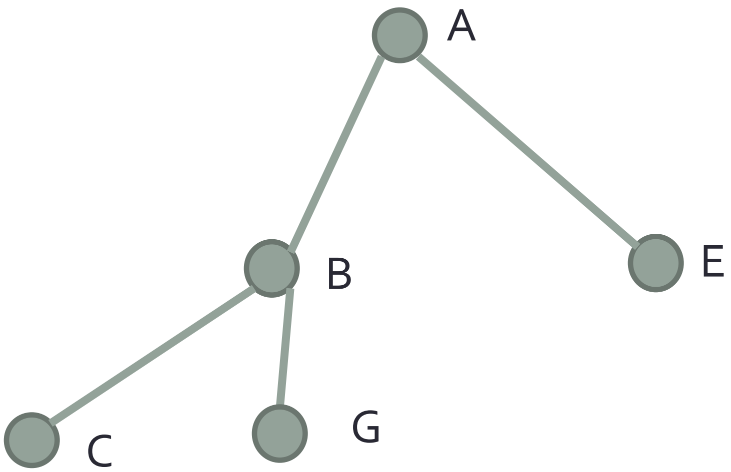


- When we reach a certain node, there are three possible actions:

1. Go to left child (use **L** to represent left branch)
2. Go to right child (use **R** to represent right branch)
3. Process the data of this node (use **V** to represent visiting this node)

Binary Tree Traversal

- Usually L takes place before R.
- Then we have 3 possibilities:
- VLR: preorder
- LVR: inorder
- LRV: postorder

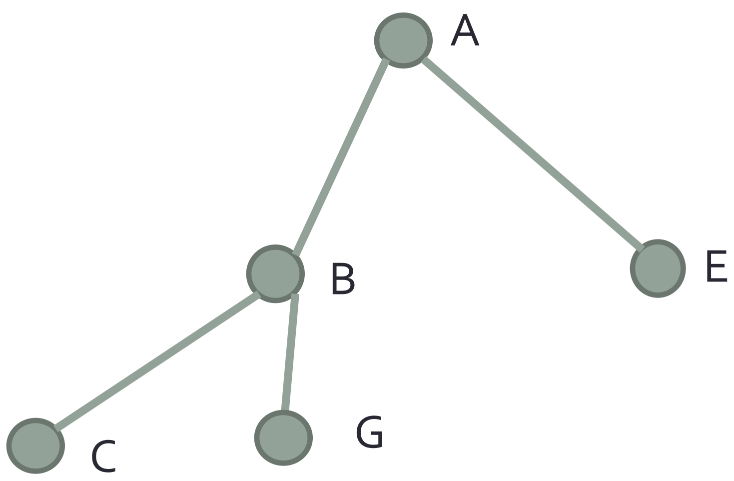


- Preorder: ABCGE
- Please write down the results of inorder & postorder traversals.

Recursive traversal

- Use recursive implementation for tree traversal:

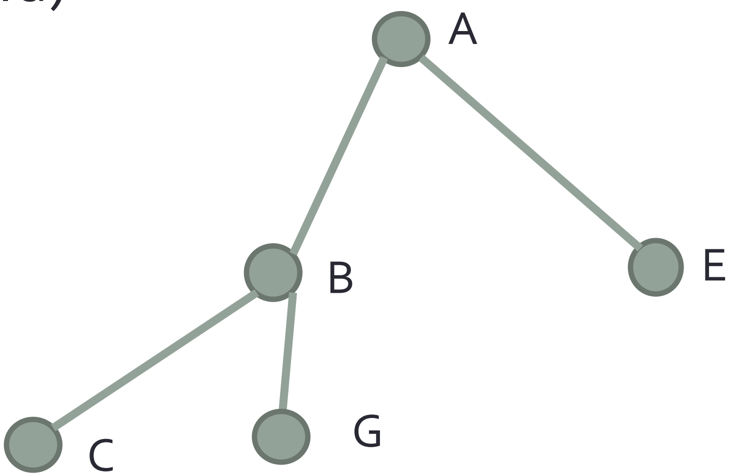
```
void inorder(treePointer ptr) {
        inorder(ptr->leftChild);
        visit(ptr);
        inorder(ptr->rightChild);
}
```



How about non-recursive? (iterative)

- Use a stack to help:

```
for(;;) {
        for(;node;node=node->leftChild)
                push(node);
        node=pop();
        if (!node) break;
        printf("%s", node->data);
        node=node->rightChild;
}
```



Arithmetic Expression

- Example: 1+2*3-5/(4+5)/5

- We have:
- Operand – 1, 2, 3, 5, 4, 5, etc.
- Operator – + * - /
- Parenthesis– (,)

- Feature 1: left-to-right associativity
- Feather 2: infix: operator is between two operands
- Association order is according to the priority of the operator
- Example: multiplication's priority is larger than addition

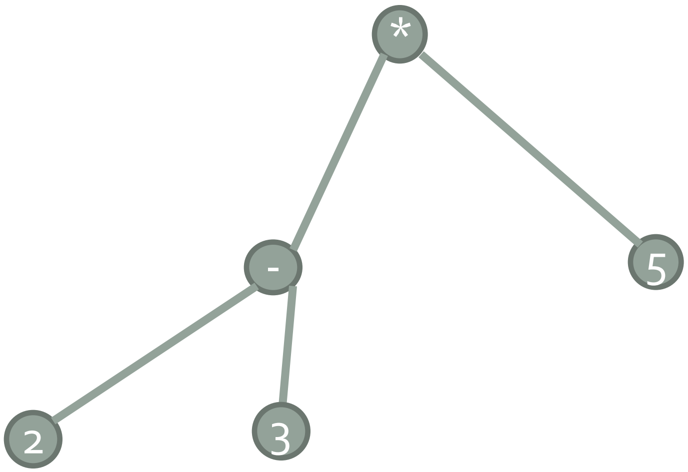
Alternative expressions

- Postfix: put the operator after the two operands
- Example
- 2+3*4 → 2 3 4 * +
- a*b+5 → ?
- (1+2)*7 → ?
- a*b/c → ?
- (a/(b-c+d))*(e-a)*c → ?
- a/b-c+d* → ?
- e-a*c → ?

Binary tree with arithmetic expression

- Every arithmetic expression can be converted to an expression tree

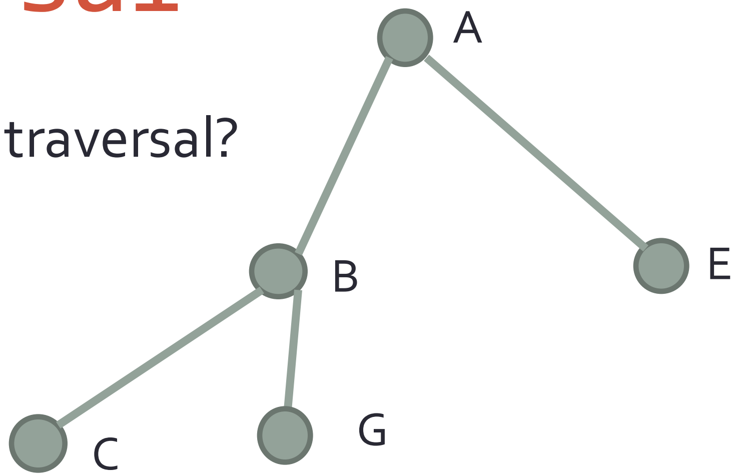
- Preorder → prefix
- Inorder → infix
- Postorder → postfix



- Exercise:
- Draw the arithmetic expression tree of
  (a/(b-c+d))*(e-a)*c

Level-order traversal

- What if we use a queue to help with the traversal?




```
add(ptr);
for(;;) {
        ptr=delete();
        if (ptr) {
                printf("%s", ptr->data);
                if (ptr->leftChild)
                        add(ptr->leftChild);
                if (ptr->rightChild)
                        add(ptr->rightChild);
        } else break;
}
```

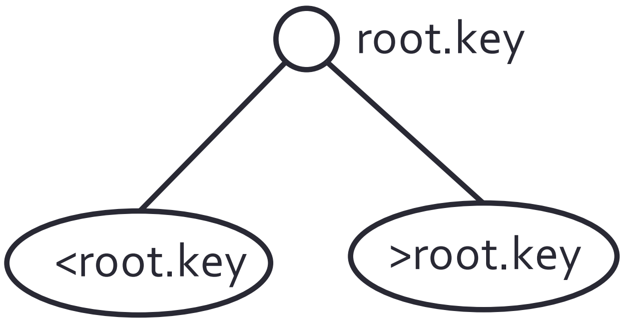
Binary search tree

- Problem: looking for the grade of a particular student in the university database.

- Assumptions:
- We know the student ID (key)
- Use the key to find the location where the data is stored
- Frequent addition of new students
- Frequent removal of students who dropped out

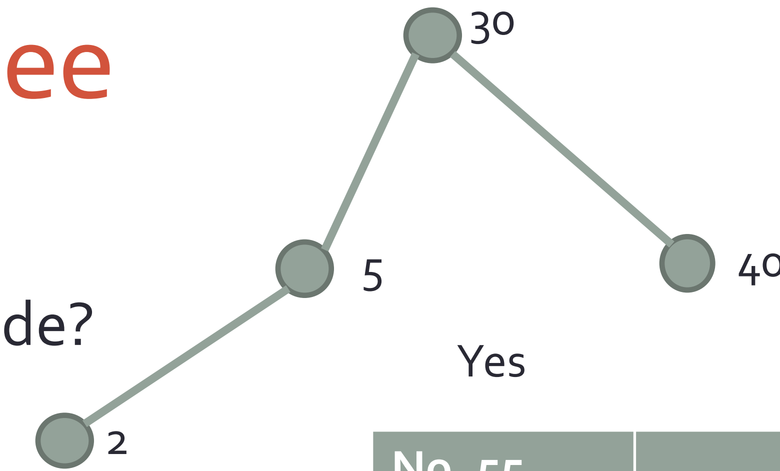
Binary search tree

- Definition: A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:
- 1. The root has a key.
- 2. The keys (if any) in the **left** subtree are **smaller** than the key in the root
- 3. The keys (if any) in the **right** subtree are **larger** than the key in the root
- 4. The left and right subtrees are **also binary search trees**
- (Hidden) All keys are distinct.
- Note that the definition is recursive.



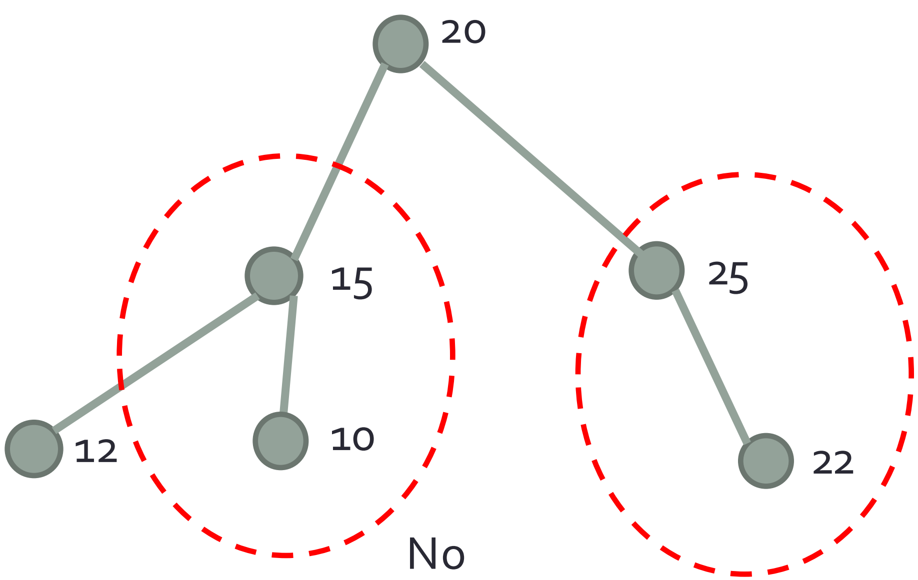
Binary search tree

- Are these binary search trees?
- What's next when we find the node?

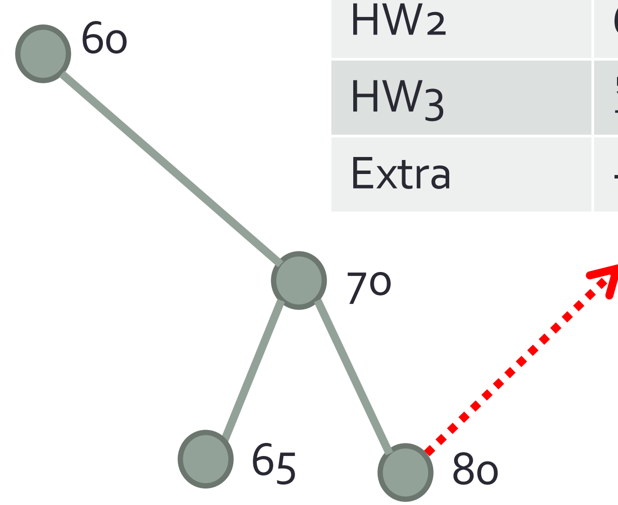


Yes

No. 55	
HW1	65
HW2	65
HW3	空
Extra	-20000



No

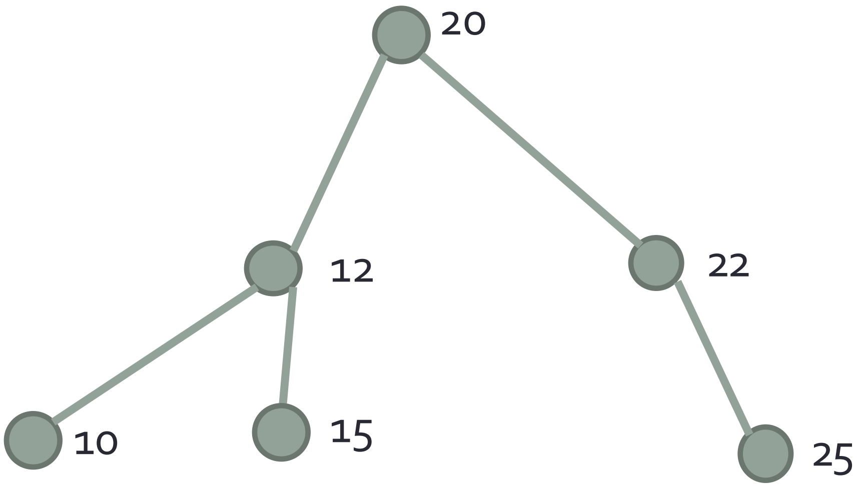


Yes

BST struct definition

```
struct BinarySearchTreeNode {
      int data;
      struct BinarySearchTreeNode *left;
      struct BinarySearchTreeNode *right;
};
```

Search

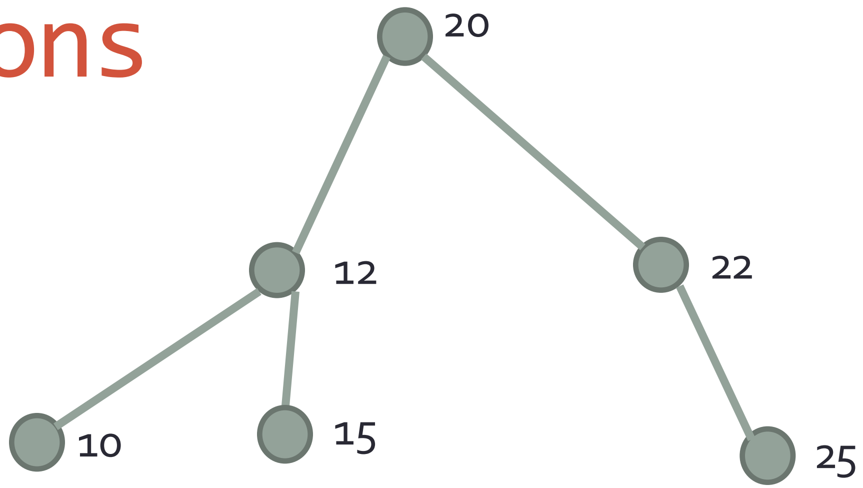



```
struct TreeNode* find(struct TreeNode* root, int data)
{
  if (root==NULL) return NULL;
  if (data==root->data) return root;
  if (data<root->data) return find(root->left,data);
  return find(root->right, data);
}
```

- Time complexity = O(??)
- A: O(h), h: height of the tree.
- Worst case: $O(n)$ Average case: $O(\log_2 n)$

Binary Search Tree Algorithm usually is like:
(1) If the key matches the key of this node, then we process and return.
(2) If key is larger or smaller, then use a recursive call to process left or right branch.

Other operations

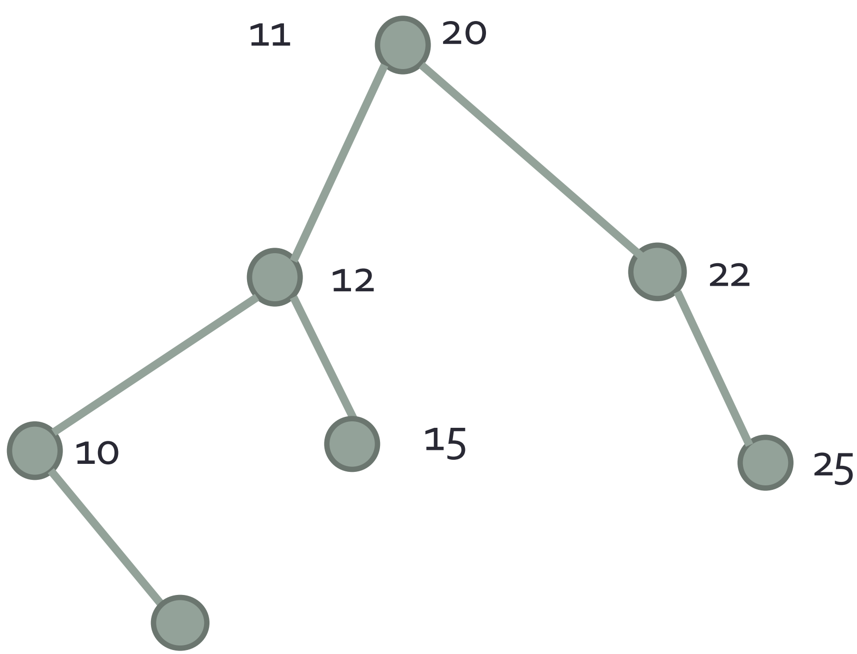


- Q: How to find the largest or smallest key in the BST?
- A: Keep going right(right), until reaching NULL (a leaf).

- Q: How to list all keys in a binary search tree in ascending order?
- A: Perform inorder traversal of the BST.

How to insert a new node?

- Search if there exists the same key in the BST
  (Recall the rule: each key in the BST is unique)
- If not found, insert at the last location (where we cannot find the key)
- Insert: 11



```
struct BinarySearchTreeNode *Insert(struct BinarySearchTreeNode *root, int data) {
        if (root==NULL) {
                root=(struct BinarySearchTreeNode*)malloc(sizeof(struct BinarySearchTreeNode));
                if (root==NULL) {
                        printf("Error\n");
                        exit(-1);
                }
                root->data=data;
                root->left=NULL;
                root->right=NULL;
        }else{
                if (data<root->data)
                        root->left=Insert(root->left,data);
                else if (data>root->data)
                        root->right=Insert(root->right,data);
        }
        return root;
}
```

Finding NULL means we have reached a leaf and the key is not found. → insert at this location.

If larger or smaller, use a recursive call to process.

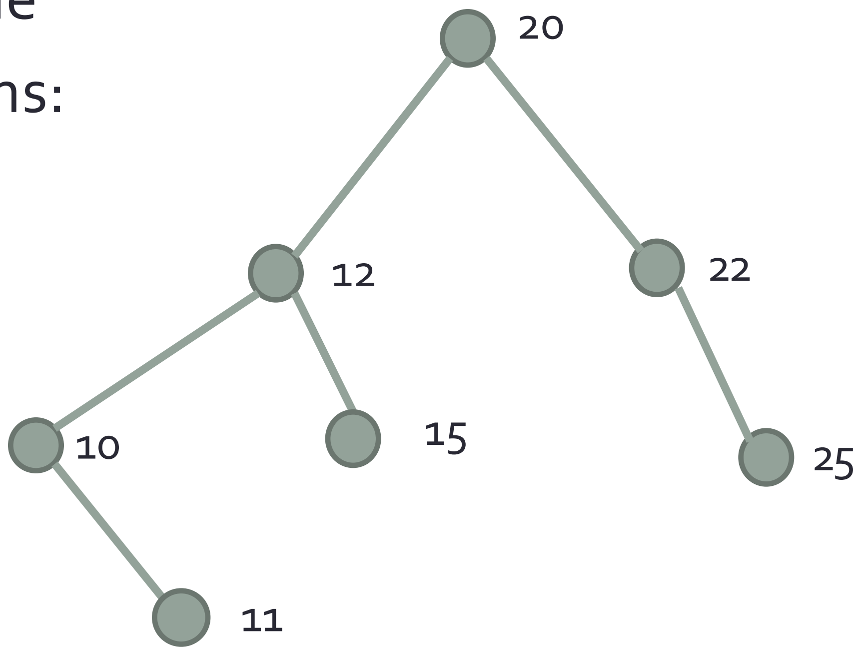
Return to prev. level

How to delete a node?

- First find the location of the node
- Then, according to the conditions:

- The node with the key has no branch (degree=0)

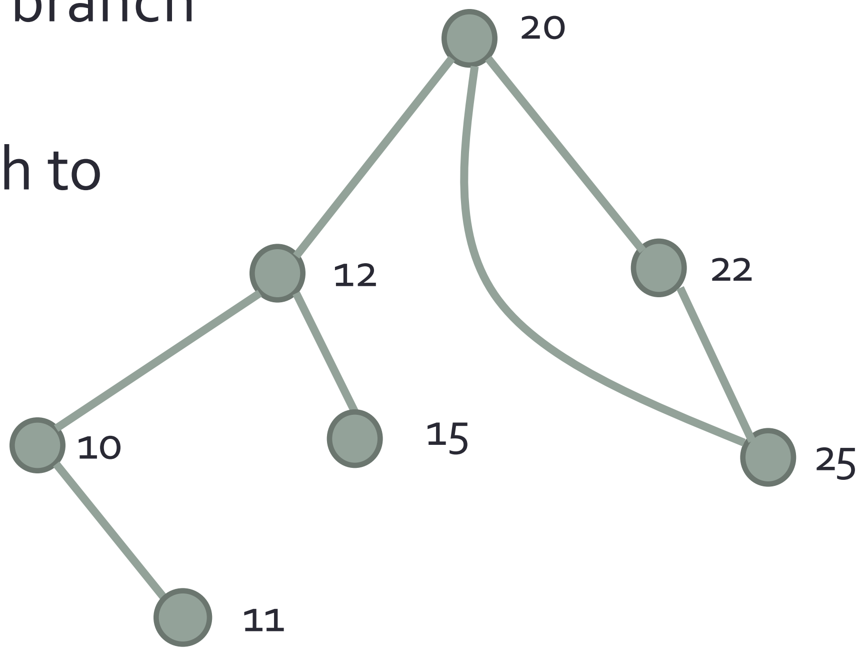
- Remove the node and then done!



How to delete a node?

- If the node with the key has one branch (degree=1)
- Then get its only child and attach to the parent

- Example: remove 25



- Q: How to remember the pointer?
- (return to the previous level to process, similar to slide #37)
- (Karumanchi p.152)

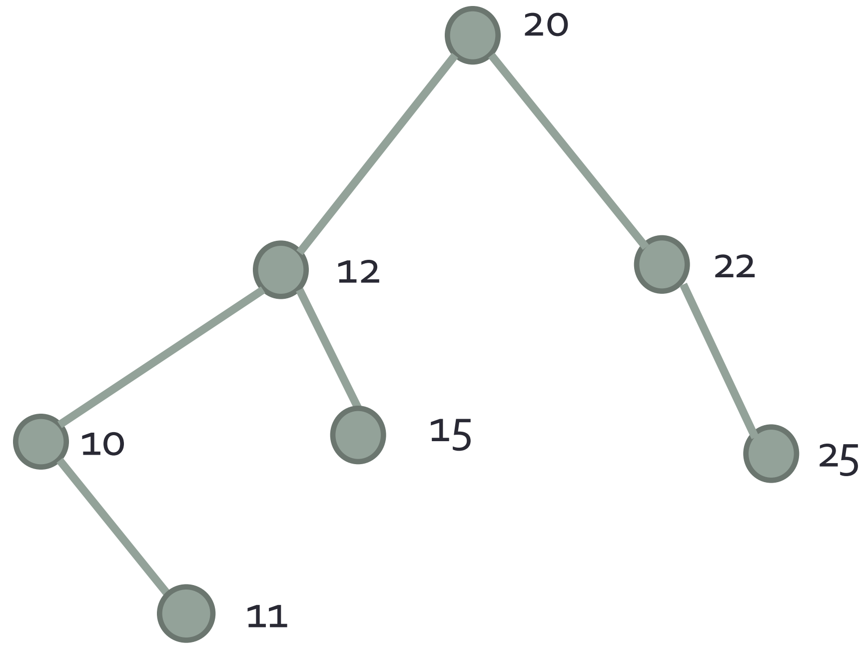
How to delete a node?

- What if both branches of the node with the key exist (degree=2)?

- Example: remove 12
- Find the largest node of the left branch (or the smallest of the right branch)

- Remove that node and move it to where the node with the key was.

- Q: What if that node still has child node(s)?



A: there would be only one child.

```
struct TreeNode *delete(struct TreeNode *root, int data) {
        TreeNode * temp;
        if (root==NULL) {
                printf("error\n");
                return NULL;
        } else if (data < root->data)
                root->left=delete(root->left, data);
        else if (data > root->data)
                root->right=delete(root->right, data);
        else { // data == root->data
                if (root->left && root->right) { //two children
                        temp=findmax(root->left);
                        root->data=temp->data;
                        root->left=delete(root->left,root->data);
                }else{ // one child or no child
                        temp=root;
                        if (root->left==NULL)
                                root=root->right;
                        if (root->right==NULL)
                                root=root->left;
                        free(temp);
                }
        return root;    }
```

If larger or smaller, use a recursive call to process.

If we have found the key, process it here.

Return to the prev. level. If we delete the current node, we can use this to connect the parent node to a child node.