# Algorithms – I (CS29003/203)

Autumn 2022, IIT Kharagpur

## Trees

# Resources

- Apart from the book
- UC Davis ECS 36C Course by Prof. Joël Porquet-Lupine

# Tree as a Data Structure

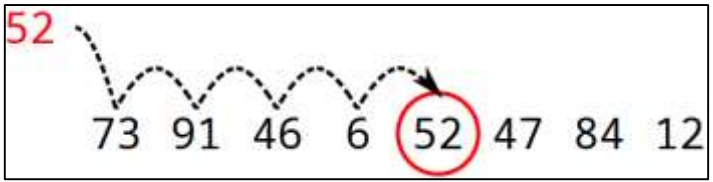
- Since Data is an integral part on which algorithms operate, particular ways of organizing data play a critical role in design and analysis of algorithms
- A **data structure** can be defined as a particular scheme of organizing related data items
- Two most important elementary data structures are **arrays** and **linked lists**
- Quick review:
  - Array: A sequence of items of same data type stored contiguously in memory and accessible by specifying an index
  - Linked list: A sequence of elements (nodes) containing some data as well as links to other nodes. Accessible by starting with the first node and going through the links until the node in question is reached

# Searching Data

- Searching data is one of the ubiquitous operations in programing
- Similar related operations are finding maximum or minimum etc.
- Now, the question is how to make searching as efficient as possible
- A related question: What are the best *search data structures*?

# Searching Data

- Linear search: Definition
  - Sequentially check each item of list until match is found
  - Also called sequential search




- Characteristics
  - Very simple to implement
  - Only practical when list is short or for single-search scenario

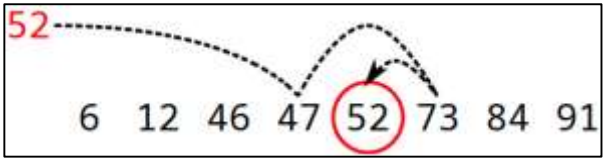
- Related data structures and complexity

| Data structure | Search | Insert | Delete | Find min/max |
|---|---|---|---|---|
| Unsorted array | $O(N)$ | $O(1)$ | (search time +) $O(1)$ | $O(N)$ |
| Unsorted linked list | $O(N)$ | $O(1)$ | (search time +) $O(1)$ | $O(N)$ |
| Sorted linked list | $O(N)$ | $O(N)$ | (search time +) $O(1)$ | $O(1)$ |

*Source: UC Davis, ECS 36C course, Spring 2020*

# Searching Data

- Sorted array and binary search: Definition
  - Compare with middle item of sorted array for match; if not a match, divide search interval in half and repeat




- Characteristics
  - Requires to have a sorted array
    - Either by construction, when inserting/deleting items
    - Or by sorting it before the search (at least $O(N \log N)$)

- Related data structures and complexity

| Data structure | Search | Insert | Delete | Find min/max |
|---|---|---|---|---|
| Sorted array | $O(\lg N)$ | $O(N)$ | $O(N)$ | $O(1)$ |

*Source: UC Davis, ECS 36C course, Spring 2020*

# Searching Data

- Summary

| Data structure | Search | Insert | Delete | Find min/max |
|---|---|---|---|---|
| Unsorted array | $O(N)$ | $O(1)$ | (search time +) $O(1)$ | $O(N)$ |
| Unsorted linked list | $O(N)$ | $O(1)$ | (search time +) $O(1)$ | $O(N)$ |
| Sorted linked list | $O(N)$ | $O(N)$ | (search time +) $O(1)$ | $O(1)$ |
| Sorted array | $O(\lg N)$ | $O(N)$ | $O(N)$ | $O(1)$ |

- Problem
  - For large sets of items, the linear complexity of linked lists and arrays for some operations is prohibitive
  - Need for a new search data structure for which the average time of most operations is $O(\log n)$

| Data structure | Search | Insert | Delete | Find min/max |
|---|---|---|---|---|
| Binary search tree | $O(\lg N)$ | $O(\lg N)$ | $O(\lg N)$ | $O(\lg N)$ |

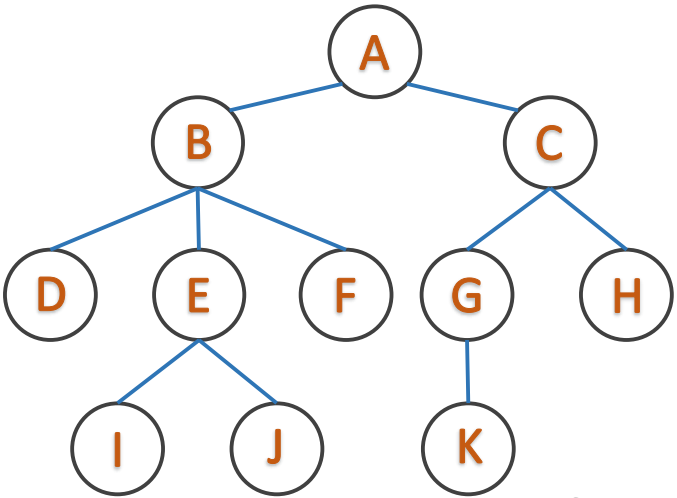
*Source: UC Davis, ECS 36C course, Spring 2020*

# Trees

## Recursive Definition

A tree can be defined recursively as:

- A collection of **nodes** starting at a **root** node
- Where each node is connected to zero or more **child** nodes via **edges**
- With the constraints that no reference is duplicated, and no node points back to the root

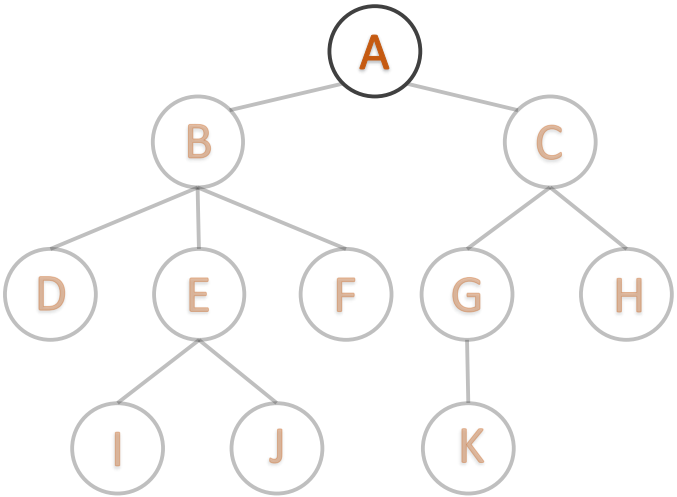



*Source: UC Davis, ECS 36C course, Spring 2020*

# Trees

## Some Terminologies

Root: The root node is the tree's top node



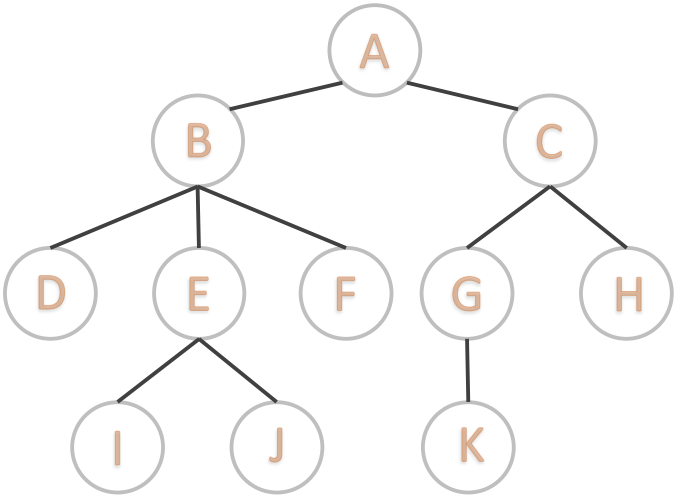

- There can be only one!

*Source: UC Davis, ECS 36C course, Spring 2020*

# Trees

## Some Terminologies

Edge: An edge is a connecting link between two nodes



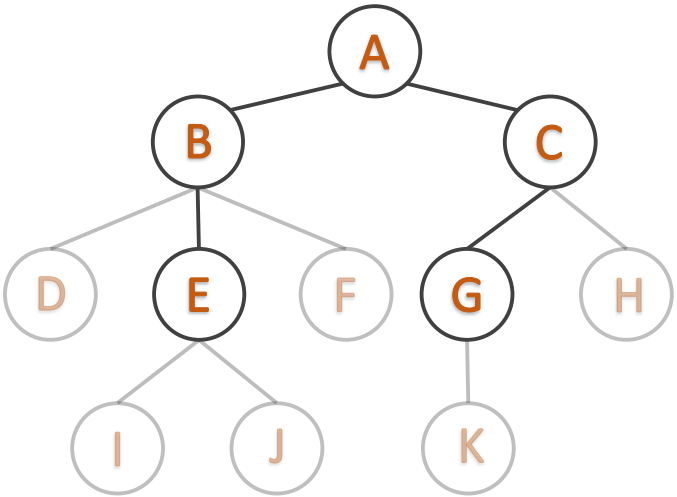

- A tree composed of N nodes has N - 1 edges

*Source: UC Davis, ECS 36C course, Spring 2020*

# Trees

## Some Terminologies

Parent: A parent node is the direct predecessor of another node

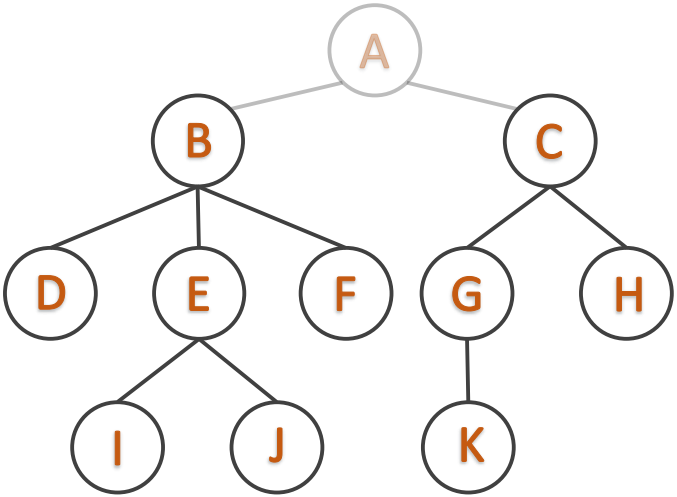



*Source: UC Davis, ECS 36C course, Spring 2020*

# Trees

## Some Terminologies

Child: A child is a node which is the direct descendant of another node



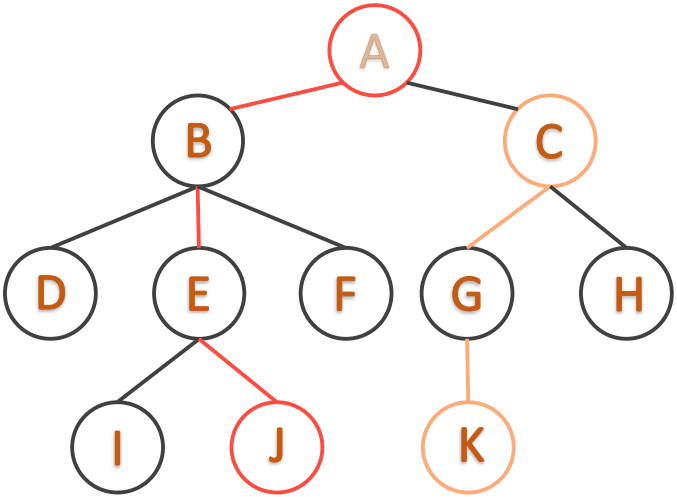

- In a tree, all the nodes except the root are child nodes

*Source: UC Davis, ECS 36C course, Spring 2020*

# Trees

## Some Terminologies

Descendent/Ancestor: Nodes connected by more than one edge



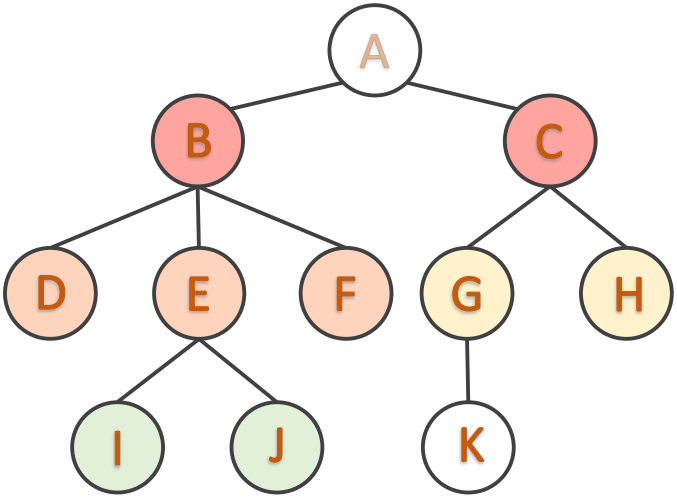

- A descendant node is also known as subchild

*Source: UC Davis, ECS 36C course, Spring 2020*

# Trees

## Some Terminologies

Sibling: Siblings are all the nodes descending from the same parent

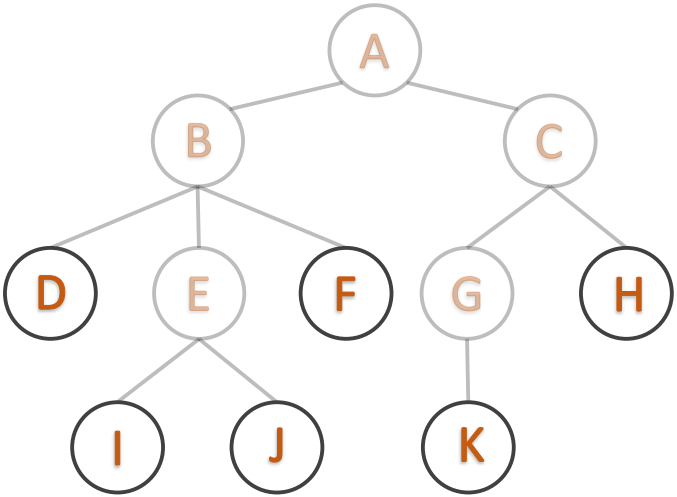



*Source: UC Davis, ECS 36C course, Spring 2020*

# Trees

## Some Terminologies

Leaf: A leaf node is a node which has no child



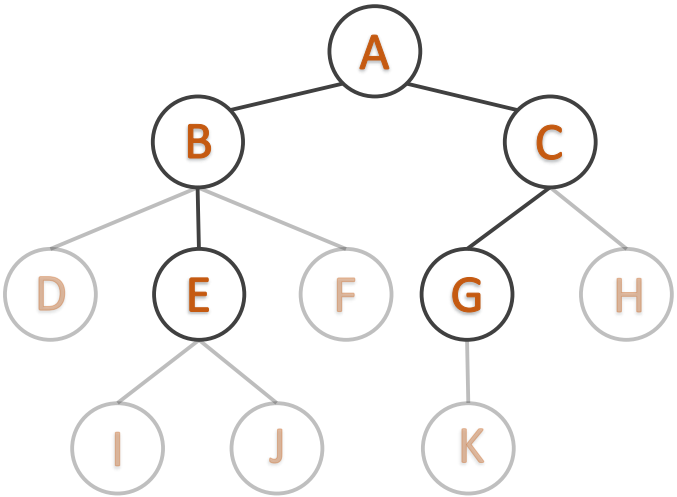

- Also called external, outer or terminal node

*Source: UC Davis, ECS 36C course, Spring 2020*

# Trees

## Some Terminologies

Internal Node: An internal node is a node which has at least one child



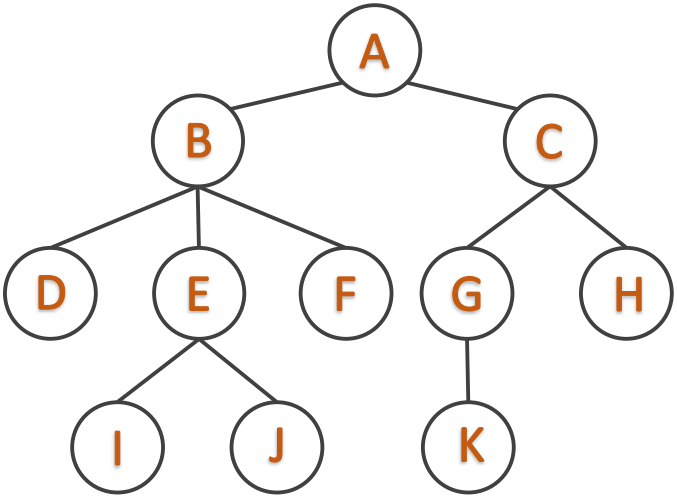

- Opposite of leaf node and also called inner or branch node

*Source: UC Davis, ECS 36C course, Spring 2020*

# Trees

## Some Terminologies

Degree: The degree of a node is the number of its children



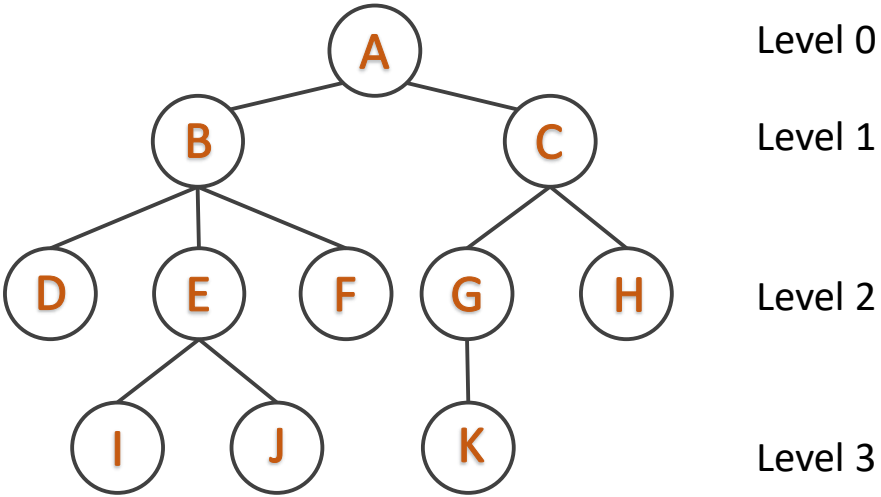

- A leaf node has always degree 0
- The degree of a tree is the maximum degree of any of its nodes

*Source: UC Davis, ECS 36C course, Spring 2020*

# Trees

## Some Terminologies

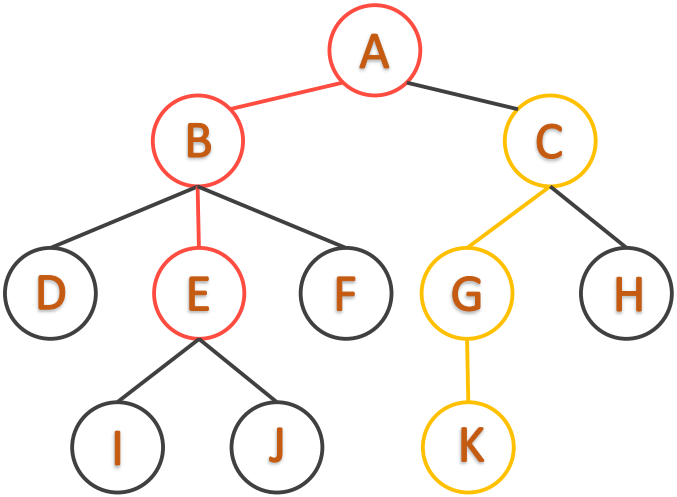
Level: The level of a node is one greater than the level of its parent with level of the root node being 0



# Trees

## Some Terminologies

Path: A path is a sequence of nodes and edges connecting a node with a descendant



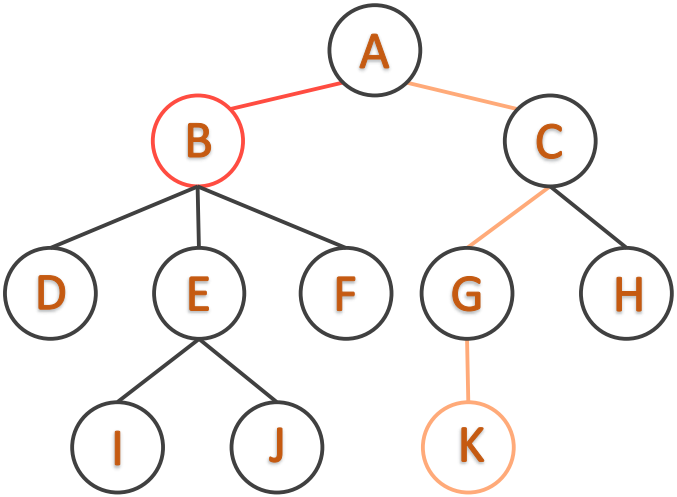

- The length of the path is the number of edges composing it

*Source: UC Davis, ECS 36C course, Spring 2020*

# Trees

## Some Terminologies

Depth: The depth of a node is the path length from the root to this node



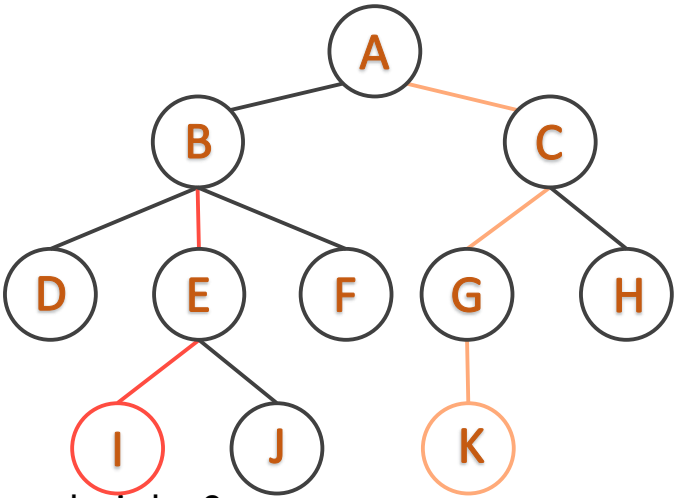

- The depth of root is 0
- The depth of a tree is equal to the depth of the deepest leaf

*Source: UC Davis, ECS 36C course, Spring 2020*

# Trees

## Some Terminologies

Height: The height of a node is the length of the longest path from a leaf to this node



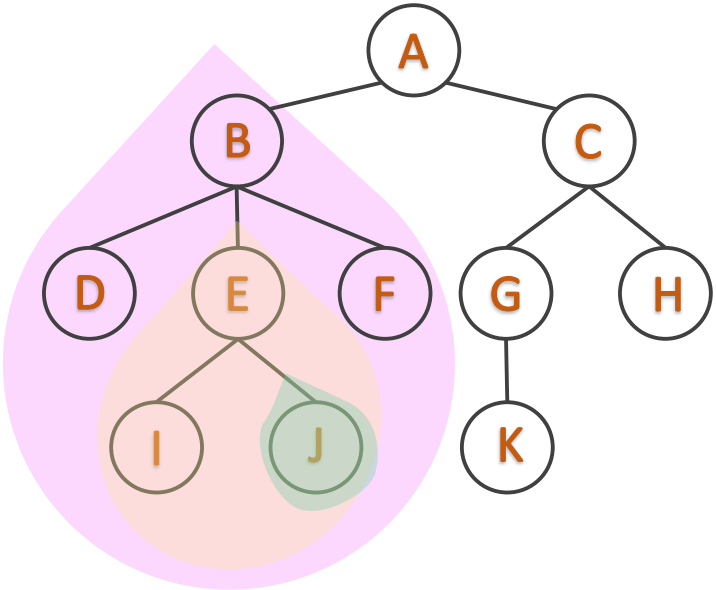

- All leaves are at height 0
- The height of a tree is equal to the height of the root from the deepest leaf (which is always equal to the depth of the tree)

*Source: UC Davis, ECS 36C course, Spring 2020*

# Trees

## Some Terminologies

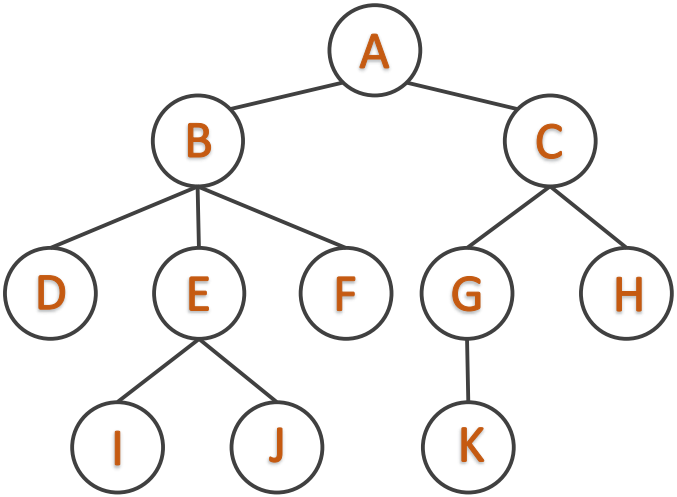
Subtree: A subtree consists of a child node and all of its descendants




*Source: UC Davis, ECS 36C course, Spring 2020*

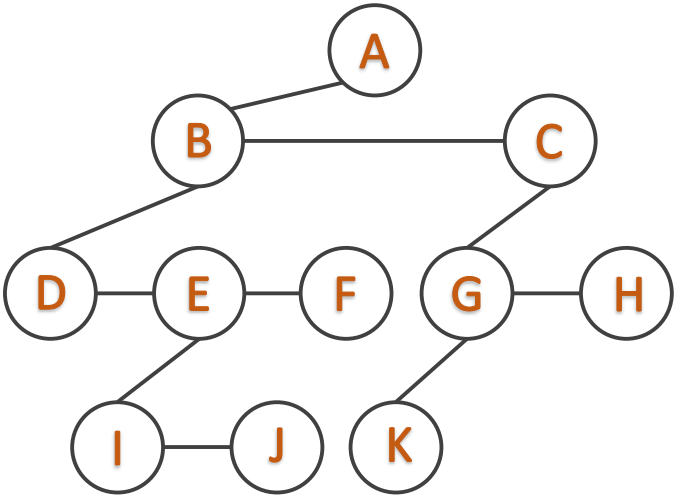
# Typical Implementations

- Each node is a struct containing data and 'n' pointers to 'n' children
- However, node degrees are not necessarily known in advance for the construction



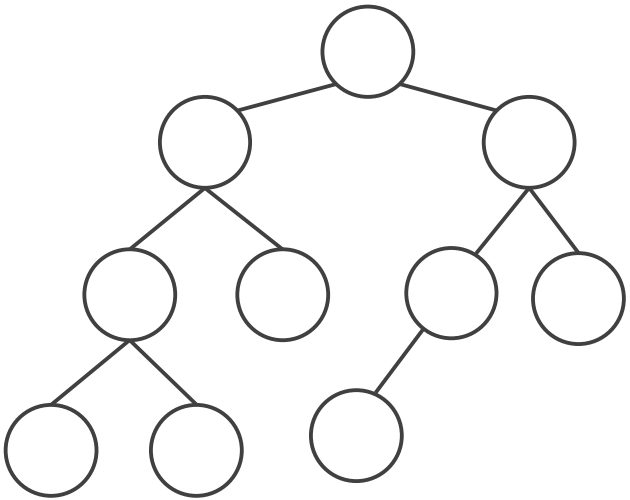
# Typical Implementations

- Each node is a linked list with data and two pointers
- First pointer links to the first child and second pointer to the next sibling



# Binary Trees

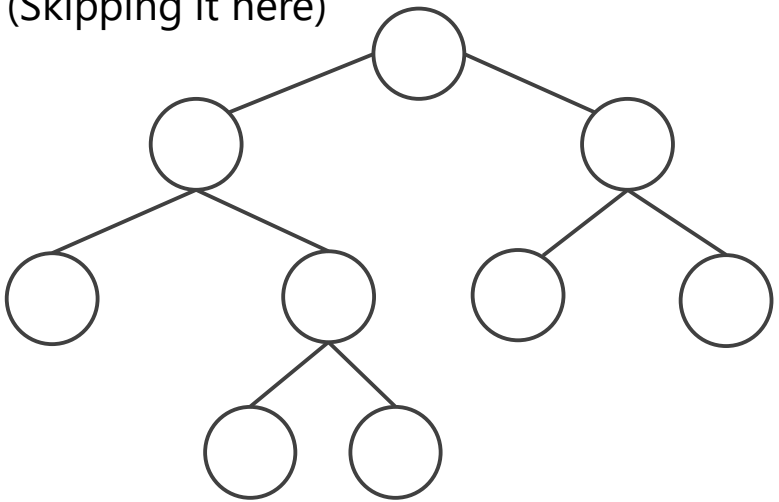
- A binary tree is a tree in which each node has **at most** two children
- Usually referred to as left child and right child



*Source: UC Davis, ECS 36C course, Spring 2020*

# Full Binary Tree

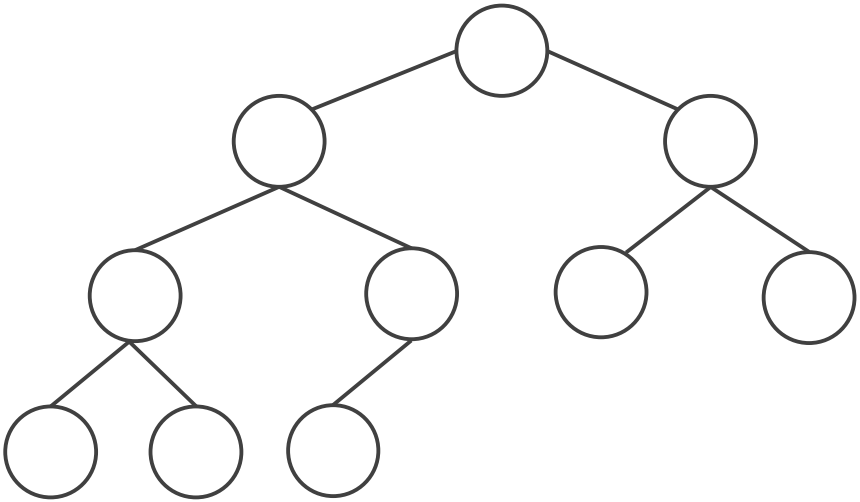
- In a full binary tree, every node has either 0 or 2 children
- The number of leaves in a non-empty full binary tree is one more than the number of internal nodes. Easy to see. Books have proof by induction. (Skipping it here)



*Source: UC Davis, ECS 36C course, Spring 2020*

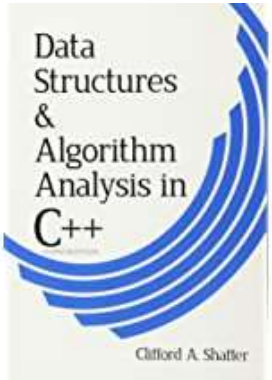
# Complete Binary Tree

- In a complete binary tree, every level, except possibly the last one, is completely filled and all nodes in the last level are as far left as possible



*Source: UC Davis, ECS 36C course, Spring 2020*

# Complete vs Full Binary Tree: Memory Aid

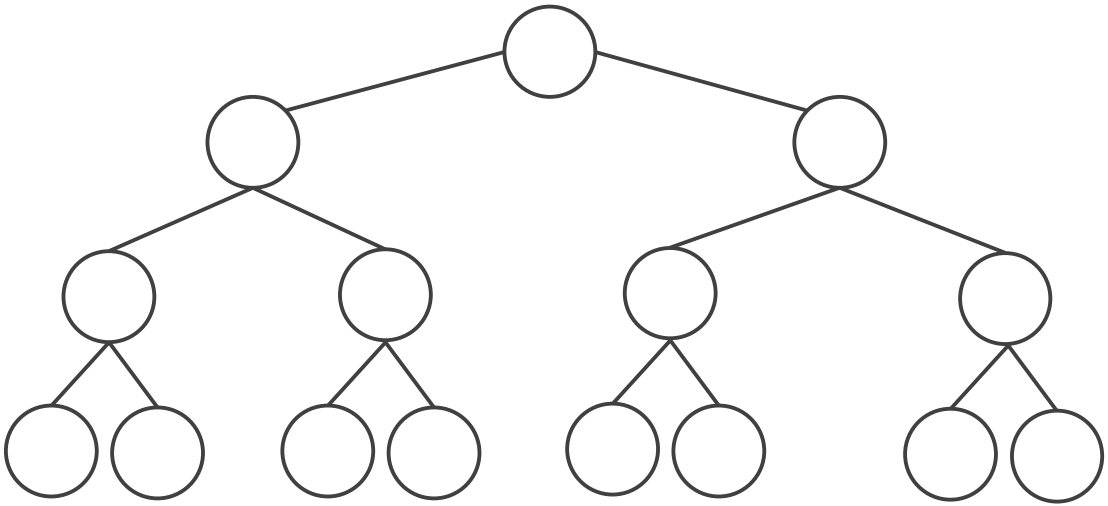



[1] While these definitions for full and complete binary tree are the ones most commonly used, they are not universal. Because the common meaning of the words "full" and "complete" are quite similar, there is little that you can do to distinguish between them other than to memorize the definitions. Here is a memory aid that you might find useful: "Complete" is a wider word than "full," and complete binary trees tend to be wider than full binary trees because each level of a complete binary tree is as wide as possible.

*Image Source: amazon.com*

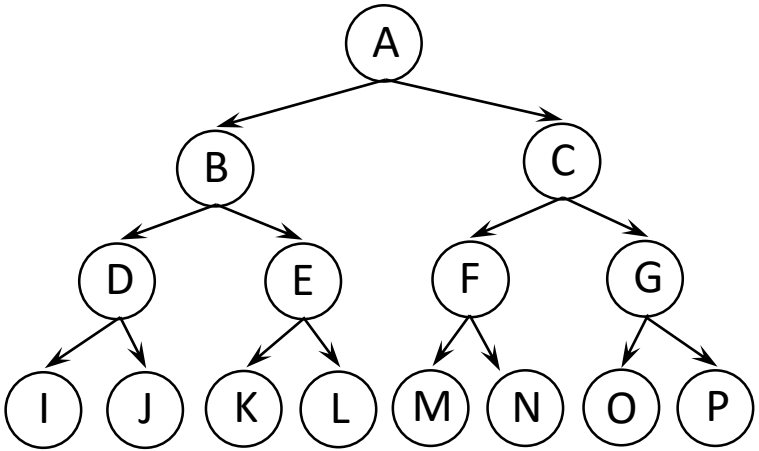
# Perfect Binary Tree

- In a perfect binary tree, all internal nodes have two children and all leaf nodes have the same depth



*Source: UC Davis, ECS 36C course, Spring 2020*

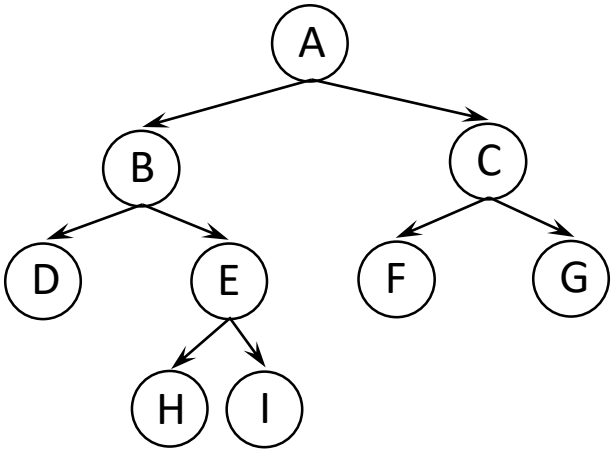
# Binary Trees




| | Max # of Nodes |
|---|---|
| Level 0 | $1 = 2^0$ |
| Level 1 | $2 = 2^1$ |
| Level 2 | $4 = 2^2$ |
| Level 3 | $8 = 2^3$ |

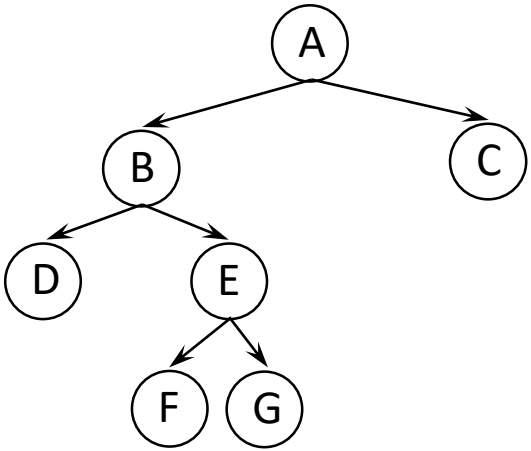
- Maximum number of nodes at level $i = 2^i$
- Maximum number of nodes of tree with height $H$ is $2^0 + 2^1 + \cdots + 2^H = 2^{H+1} - 1$
- Minimum height of a binary tree with total nodes $n$ is $\log \frac{n+1}{2}$
- Minimum number of nodes of tree with height $H$ is $H + 1$
- Maximum height of a binary tree with total nodes $n$ is $n - 1$

# Binary Trees




| | Max Nodes | Min Nodes |
|---|---|---|
| Binary | $2^{H+1} - 1$ | $H + 1$ |
| Full | $2^{H+1} - 1$ | |
| Complete | | |

# Binary Trees




For $H = 0$, Min nodes = 1
For $H = 1$, Min nodes = 3
For $H = 2$, Min nodes = 5
For $H = 3$, Min nodes = 7

| | Max Nodes | Min Nodes |
|---|---|---|
| Binary | $2^{H+1} - 1$ | $H + 1$ |
| Full | $2^{H+1} - 1$ | $2H + 1$ |
| Complete | $2^{H+1} - 1$ | $2^H$ |

| | Min Height | Max Height |
|---|---|---|
| Binary | $\log(n+1)/2$ | $n - 1$ |
| Full | $\log(n+1)/2$ | $(n-1)/2$ |
| Complete | $\log(n+1)/2$ | $\log n$ |

# Implementing Binary Tree

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

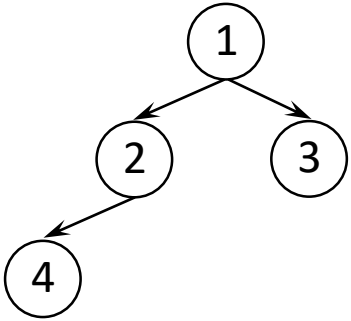
- Create a node with data and two NULL pointers in left and right

```
struct node* newNode(int data)
{
  // Allocate memory for new node
  struct node* node = (struct node*)malloc(sizeof(struct node));

  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return node;
}
```

# Implementing Binary Tree

- Create a simple binary tree

```
int main()
{
  // create root with data=1
  struct node* root = newNode(1)
  // left and right child of the root
  root->left = newNode(2);
  root->right = newNode(3);
  // left-left child of the root
  root->left->left = newNode(4);
  return 0;
}
```



# Binary Tree Traversal

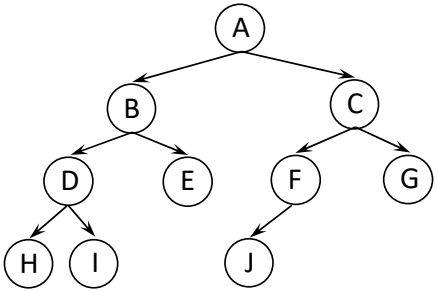
- Often we wish to process a binary tree by "visiting" each of its nodes, each time performing a specific action (such as printing, updating, searching the contents of the node)
- Any process for visiting all of the nodes in some order is called a **traversal**
- Unlike linear data structures (e.g. arrays, linked lists, etc), trees do not inherently have a linear order in which they can be traversed
- There is no "natural" end or start so to say of a tree.

- **General recursive pattern:**
- Different order of three basic operations from a node N
  - (N) Process N itself
  - (L) Recurse on N's left subtree
  - (R) Recurse on N's right subtree

# Binary Tree Traversal

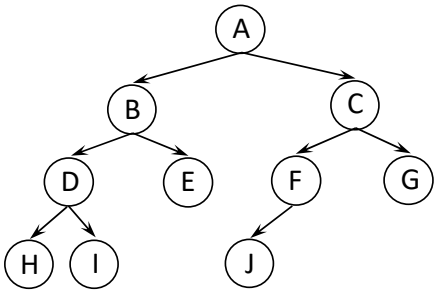
- From a node N
  - (N) Process N itself
  - (L) Recurse on N's left subtree
  - (R) Recurse on N's right subtree

Pre-order (NLR)



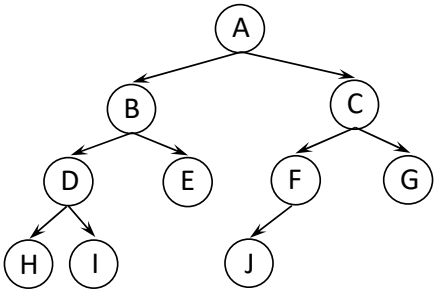

A-B-D-H-I-E-C-F-J-G

In-order (LNR)




H-D-I-B-E-A-J-F-C-G

Post-order (LRN)




H-I-D-E-B-J-F-G-C-A

# Implementing Binary Tree Traversals

Pre-order (NLR)

```
void printPreorder(struct node* node)
{
  if (node==NULL)
    return;

  // Print (N)
  printf("%d ", node->data);
  // Recurse on left subtree (L)
  printPreorder(node->left);
  // Recurse on right subtree (R)
  printPreorder(node->right);
};
```

# Implementing Binary Tree Traversals

In-order (LNR)

```
void printInorder(struct node* node)
{
  if (node==NULL)
    return;

  // Recurse on left subtree (L)
  printInorder(node->left);
  // Print (N)
  printf("%d ", node->data);
  // Recurse on right subtree (R)
  printInorder(node->right);
};
```

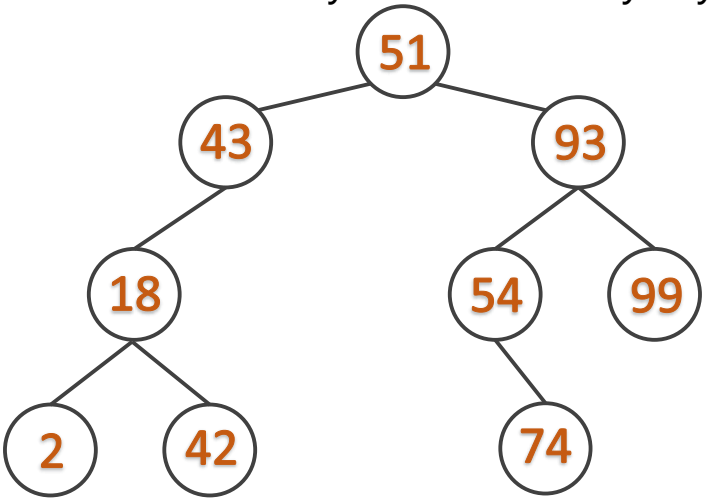
# Implementing Binary Tree Traversals

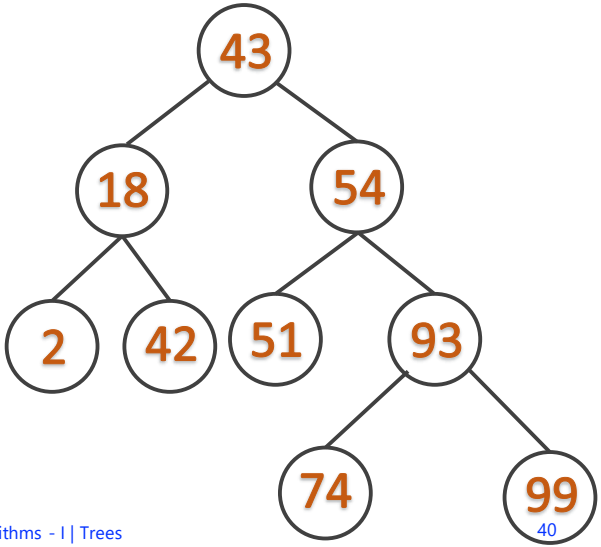
Post-order (LRN)

```
void printPostorder(struct node* node)
{
  if (node==NULL)
    return;

  // Recurse on left subtree (L)
  printPostorder(node->left);
  // Recurse on right subtree (R)
  printPostorder(node->right);
  // Print (N)
  printf("%d ", node->data);
};
```

# Binary Search Tree

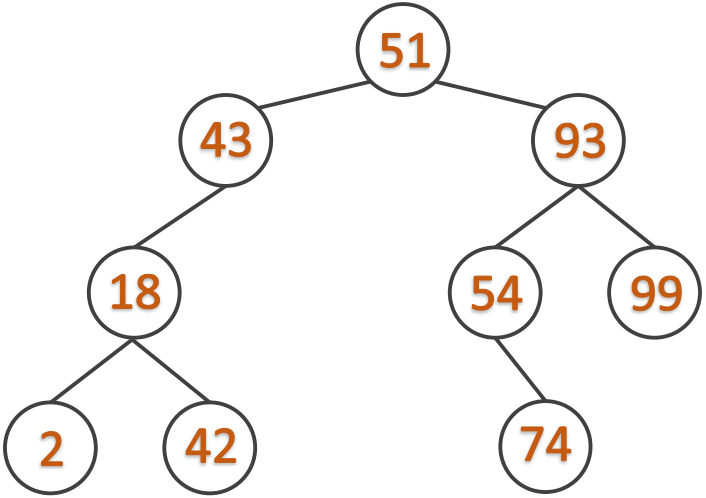
- Definition: A binary search tree is a special kind of binary tree in which
- Each node contains one comparable key
  - (and optionally some associated data/value)
- Each node's key is greater than any key stored in its left subtree
- Each node's key is less than any key stored in its right subtree



# Searching in Binary Search Tree

- Start at the root
- If query is equal to the root, return the position
- If query is less than the root, go to the left subtree and continue the search
- If query is more than the root, go to the right subtree and continue the search



# Implementing Search in Binary Search Tree

```
struct node* search(struct node* root, int query)
{
  // Base case: root is null or key is present at root
  if (root == NULL || root->key == query)
    return root;

  // If query is less than root's key, recurse on left subtree, else
  recurse on right subtree (R)
  if (query < root->key)
    return search(root->left, query);
  else
    return search(root->right, query);
}
```

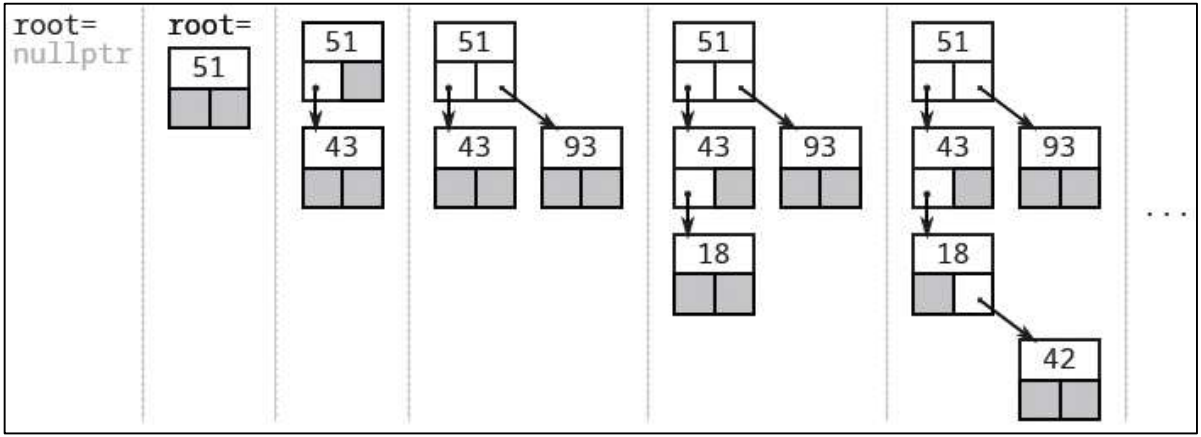
- What will be the running time? (Assume you know the height (h) of the tree)
- $O(h)$
- Worst case running time if you know there are total $n$ number of nodes?
- $O(n)$

# Inserting in Binary Search Tree

```
// function to insert a new node with given key in BST
struct node* insert(struct node* node, int key){
  // If we reach a NULL pointer, this is the place to insert
  if (node == NULL)
    return newNode(key);
  // Recur down the tree
  if (key < node->key)
    node->left = insert(node->left, key);
  else if (key > node->key)
    node->right = insert(node->right, key);
  else
    printf("Key %d is already present!\n", key);
  // Return the (unchanged) node pointer
  return node;
}
```

- What will be the running time? (Assume you know the height (h) of the tree)
- $O(h)$
- Worst case running time if you know there are total $n$ number of keys?
- $O(n)$

# Inserting in Binary Search Tree

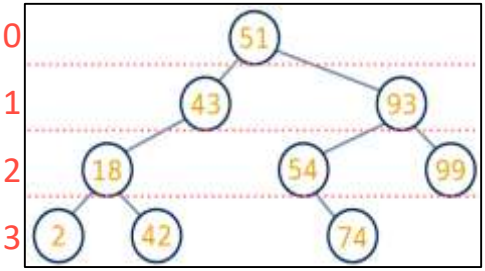



Source: UC Davis, ECS 36C course, Spring 2020

# Printing Binary Search Tree

- Basically inorder traversal – printing level of the node also

```
void printTree (struct node* node, int level)
{
  if (node==NULL)
    return;

  printTree(node->left, level+1);
  // Print node key and level
  printf("%d [%d] ", node->key, level);
  printTree(node->right, level+1);
}
```




2 [3] 18 [2] 42 [3] 43 [1] 51 [0] 54 [2] 74 [3] 93 [1] 99 [2]

# Finding Max/Min Element of Binary Search Tree

- Go as far right as possible

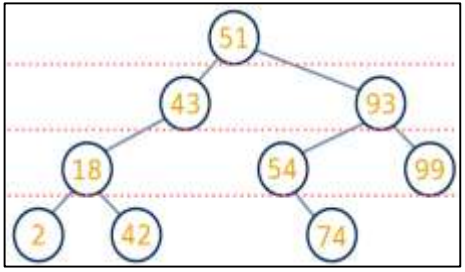
```
int getMax(struct node* node)
{
  while(node->right != NULL)
    node = node->right;
  return node->key;
}
```

- 99
- Finding max is $O(h)$

- Go as far left as possible

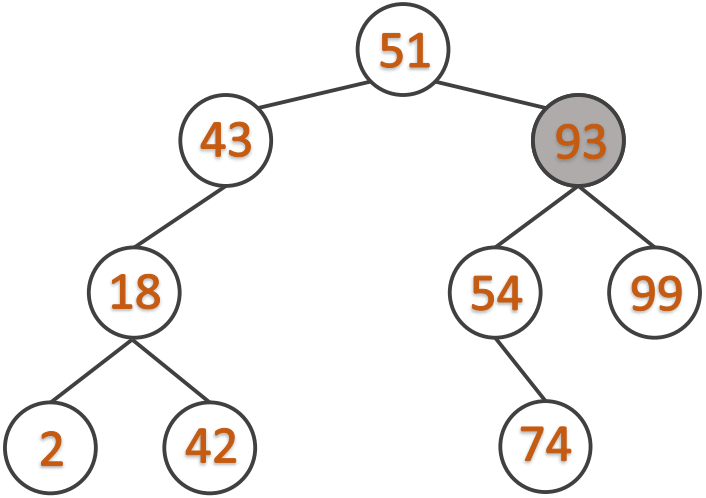
```
struct node* getMin(struct node* node)
{
  if(node->left != NULL)
    return getMin(node->left);
  else
    return node;
}
```

- 2
- Finding max is $O(h)$



# Removing Node from Binary Search Tree

- Lazy node deletion: Don't actually delete the node
- Keep the key and mark the node as "disabled"
- Upon later insertion of the same key, node can easily be re-enabled
- Somewhat viable approach only if number of deletions is very small (e.g., log files)

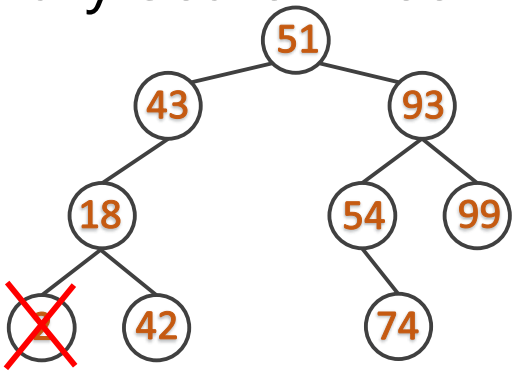



- Example after removing 93

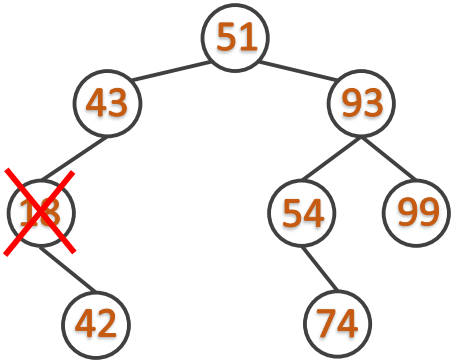
*Source: UC Davis, ECS 36C course, Spring 2020*

# Removing Node from Binary Search Tree

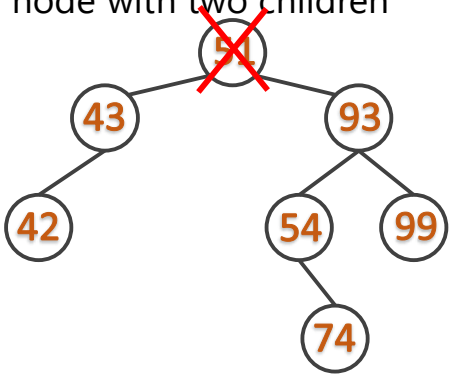
- Actual node deletion:
- 3 Scenarios
- Leaf node i.e., node with no child




- Internal node with one child




- Internal node with two children

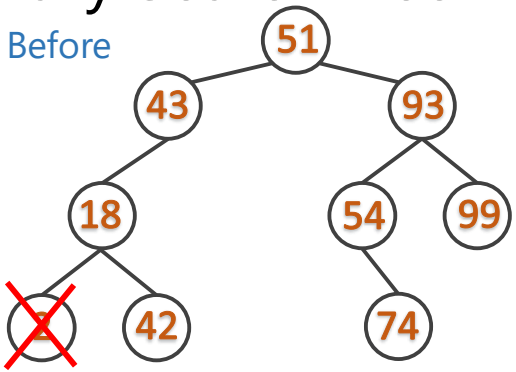



*Source: UC Davis, ECS 36C course, Spring 2020*

# Removing Node from Binary Search Tree

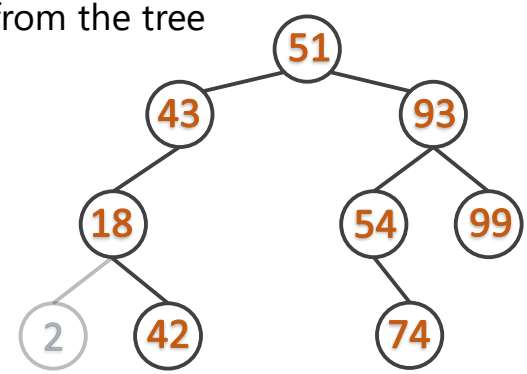
Leaf node i.e., node with no child

Before

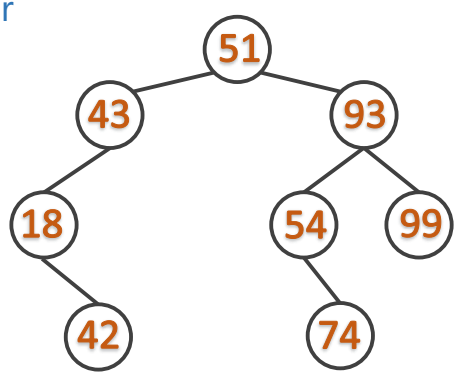



Algorithm

Simply remove node to be deleted from the tree




After

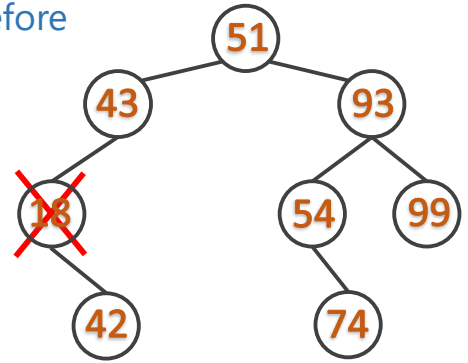



*Source: UC Davis, ECS 36C course, Spring 2020*

# Removing Node from Binary Search Tree

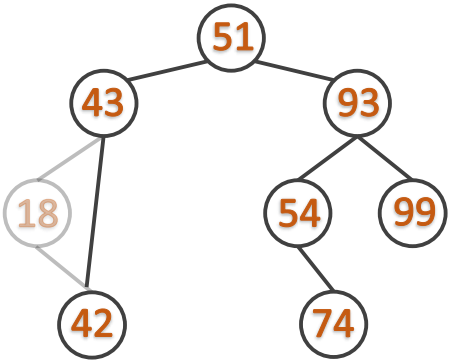
Internal node with one child

Before

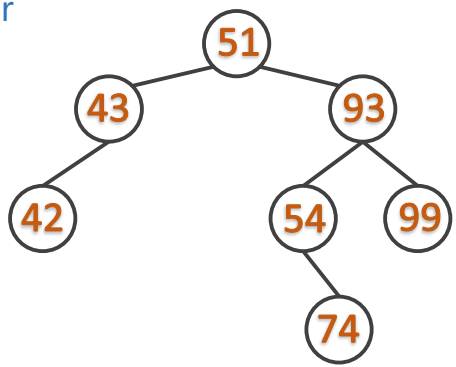



Algorithm

Replace node with its child




After




*Source: UC Davis, ECS 36C course, Spring 2020*

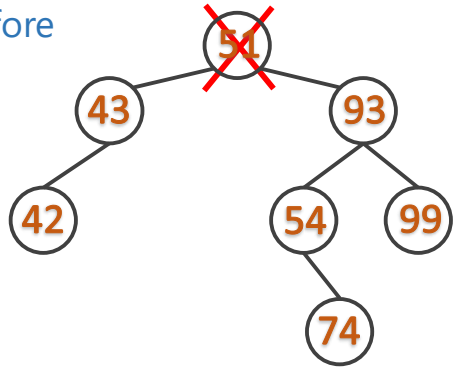
# Removing Node from Binary Search Tree

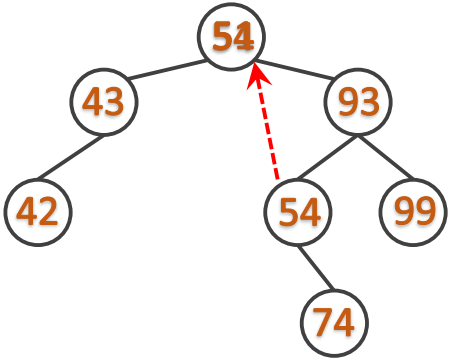
Internal node with two children

Algorithm

- Let $D$ be the node to be deleted
- Find min node in $D$'s right subtree
  - i.e., $D$'s successor key value
- Copy min node's key into $D$
- Recursively delete min node
- We can also proceed with $D$'s predecessor key value

Before



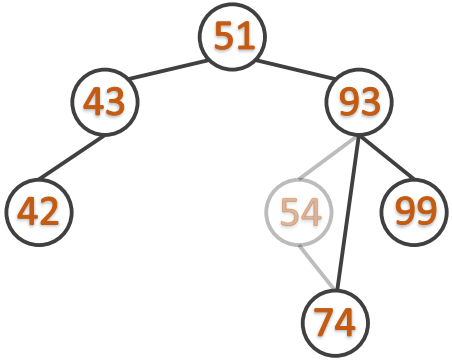

*Source: UC Davis, ECS 36C course, Spring 2020*

# Removing Node from Binary Search Tree

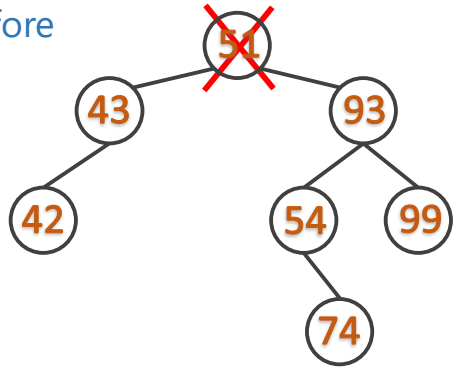
Internal node with two children

Algorithm

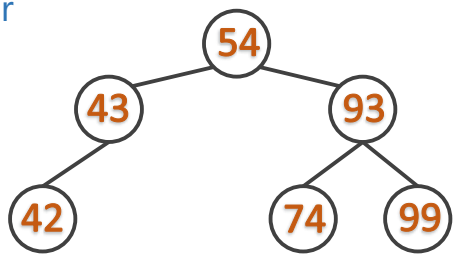
- Let $D$ be the node to be deleted
- Find min node in $D$'s right subtree
  - i.e., $D$'s successor key value
- Copy min node's key into $D$
- Recursively delete min node
- We can also proceed with $D$'s predecessor key value




Before




After

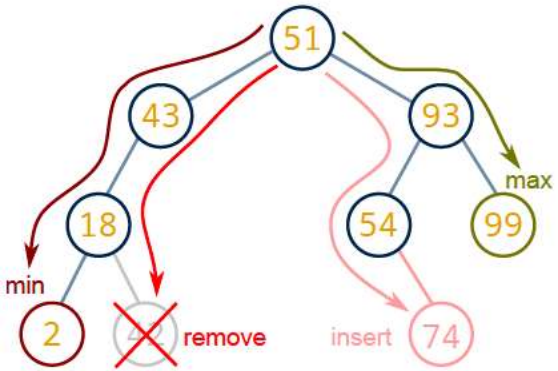



*Source: UC Davis, ECS 36C course, Spring 2020*

# Removing Node from Binary Search Tree

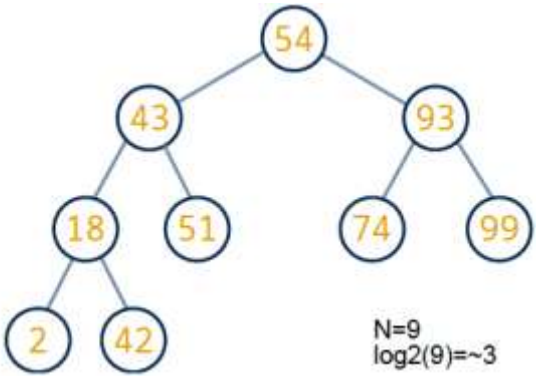
```
struct node* delete(struct node* node, int key){
    // key not found
    if (node == NULL){return NULL;}
    // otherwise, recur down the tree, if smaller then go left, otherwise right
    if (key < node->key)
        node->left = delete(node->left, key);
    else if (key > node->key)
        node->right = delete(node->right, key);
    // if key is same as root->key, then this is the node to be deleted
    else {
        // if leaf node, remove it
        if((node->left == NULL) && (node->right == NULL)){free(node); return NULL;}
        // if node with left child only
        else if((node->left != NULL) && (node->right == NULL)){struct node* temp = node->left; free(node); return temp;}
        // if node with right child only
        else if((node->left == NULL) && (node->right != NULL)){struct node* temp = node->right; free(node); return temp;}
        // if two children: replace with min node in right subtree
        if((node->left != NULL) && (node->right != NULL)){
            struct node* temp = getMin(node->right);
            node->key = temp->key;
            // delete the min node in the right subtree
            node->right = delete(node->right, temp->key);
        }
    }
    return node;
}
```

# Binary Search Tree – Complexity Analysis




- Running time of all operations is $O(d)$
  - Apart from traversal, which is necessarily $O(n)$
- $d$ is the depth of the tree and $n$ is the number of nodes

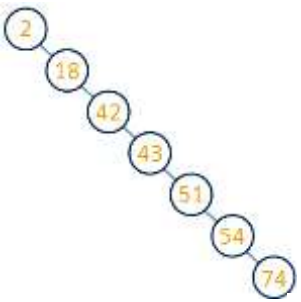
- But, tree shape depends on order of insertion
- At best, if the tree is complete, its height is $O(\log n)$
- On average, if keys are inserted in random order, the height is still proportional to $O(\log n)$




*Source: UC Davis, ECS 36C course, Spring 2020*

# Binary Search Tree – Complexity Analysis

- Deletions (with the presented approach) tend to displace nodes from the right subtrees to the left subtrees
- Trees become unbalanced
- Some solutions
  - Avoid/forbid deletions (Lazy deletions)
  - Alternately picking min node in right tree and max node in left tree when deleting nodes with two children (but improvement is not mathematically proven)
- Worst case scenario: Insertions of ordered values. Complexity of all operations become $O(n)$




*Source: UC Davis, ECS 36C course, Spring 2020*

# Binary Search Tree – Complexity Analysis

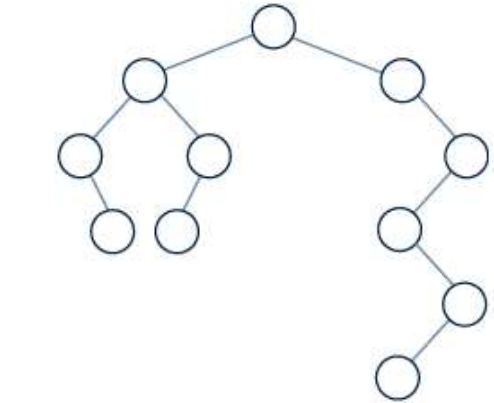
- Solutions:
- Make BST always balanced
  - Each node has about the number of descendants in its left subtree as in its right subtree
  - Guarantee logarithmic performance for all operations

- Such binary trees are called self-balancing
  - e.g., AVL trees, Red-Black trees etc.

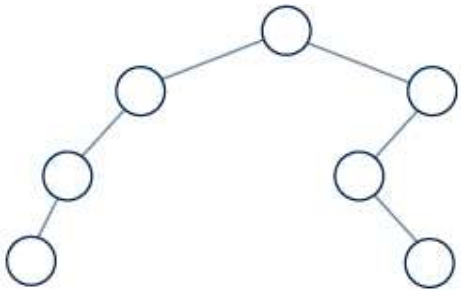
*Source: UC Davis, ECS 36C course, Spring 2020*

# Some Ideas to Balance Binary Search Tree

- Same population from root
- Maintain same number of **nodes** on each side of the **root**
- But …



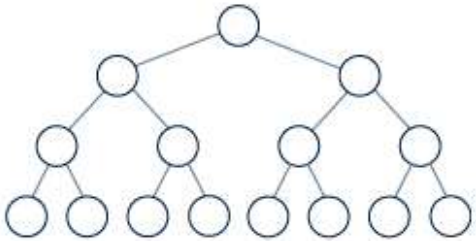
- Same height from root
- Maintain same **height** on each side of **root**
- But …



*Source: UC Davis, ECS 36C course, Spring 2020*

# Some Ideas to Balance Binary Search Tree

- Same height for each node
- For each **node**, maintain same **height** in left and right subtrees
- But …

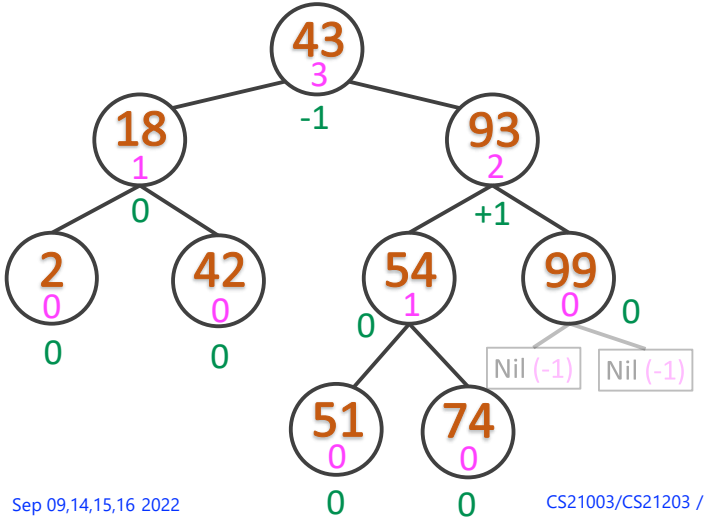


- Requirements:
- Must find a good balance condition
- Easy to maintain
- Depth of BST stays $O(\log n)$
- Not too rigid

*Source: UC Davis, ECS 36C course, Spring 2020*

# AVL Trees

- Published in 1962, by **A**delson-**V**elskii and **L**andis
  - One of the first self-balancing BST data structure to be invented
- AVL trees are height-balanced binary search trees
- For each node, the heights of left and right subtrees can differ by **at most 1**
- Such difference value is called the node's balance factor

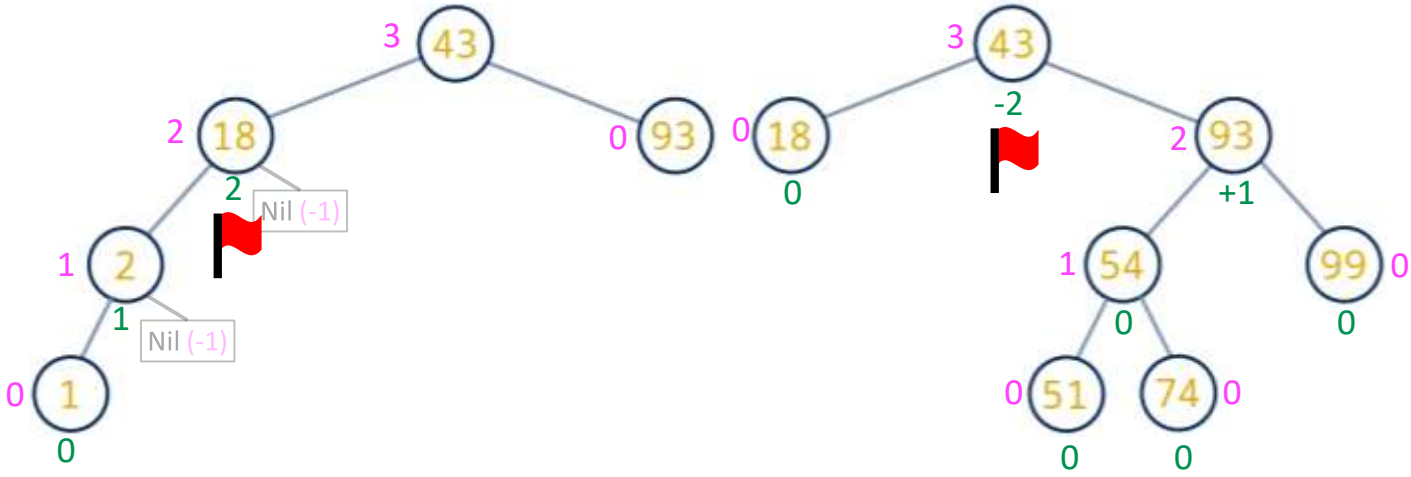



Height
Balance factor

*Source: UC Davis, ECS 36C course, Spring 2020*

# AVL Trees

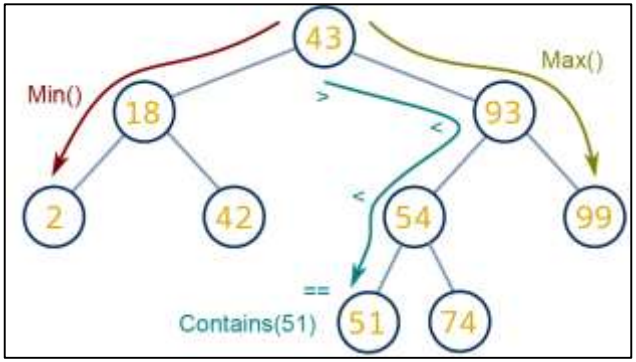
- Check for AVL/Non-AVL Trees
- For each node
  - Determine its height
  - Determine its balance factor
  - If balance factor is not -1, 0 or 1, then tree is not AVL



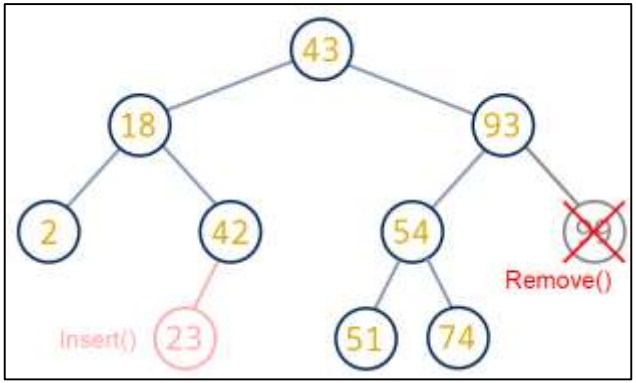

Source: UC Davis, ECS 36C course, Spring 2020

# Operations on AVL Trees

- Query operations stay unchanged
  - Searching, finding max/min are exactly the same
- But worst case complexity is now, $O(\log n)$
  - Because AVL trees are guaranteed to be height balanced



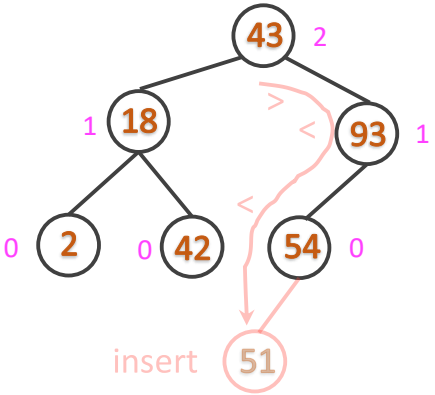

- Operations that could alter the balance of the tree must be slightly modified
  - Insert(), Remove()/Delete()
- Rebalancing after insertion or removal if balance condition is broken



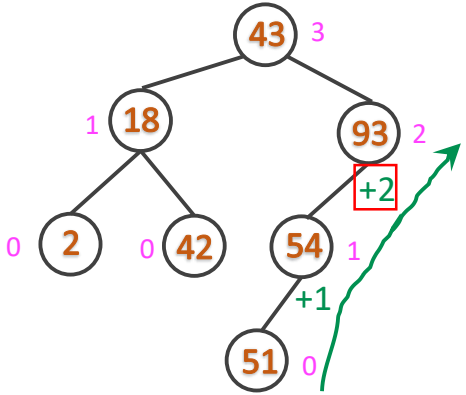

*Source: UC Davis, ECS 36C course, Spring 2020*

# AVL Trees – Simple Rebalancing

Insertion – first step is same regular BST insertion (recursive)



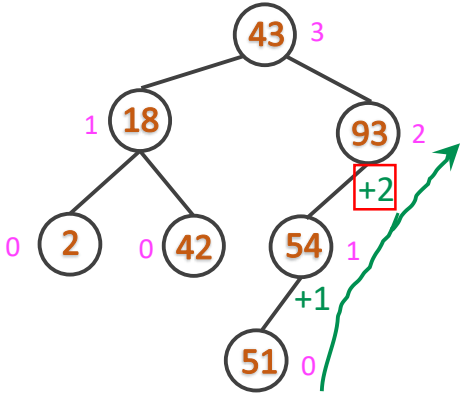

**Retracing** – check every ancestor's balance factor



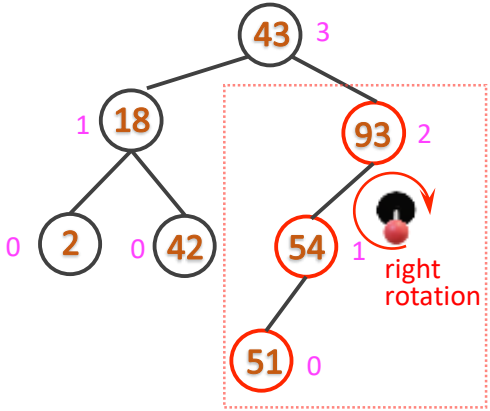

*Source: UC Davis, ECS 36C course, Spring 2020*

# AVL Trees – Simple Rebalancing

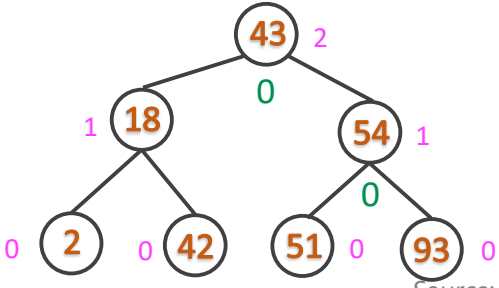
**Retracing** – check every ancestor's balance factor




Invert unbalanced parent with "heavy" child through rotation

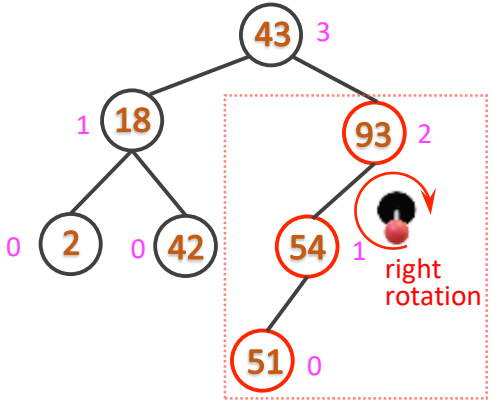


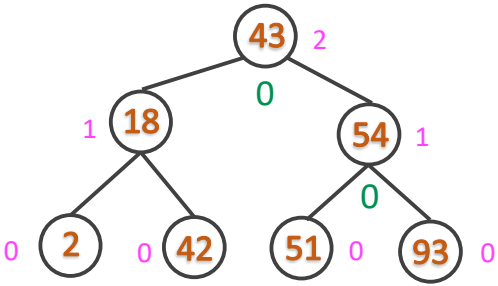

Tree is being back to AVL



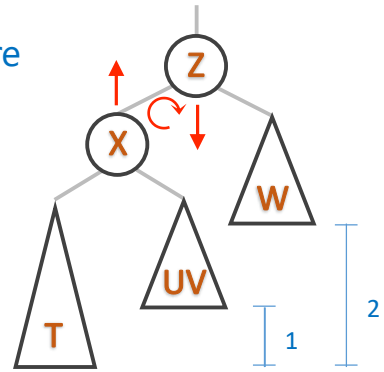

*Source: UC Davis, ECS 36C course, Spring 2020*

# AVL Trees – Rebalancing – A more General Case

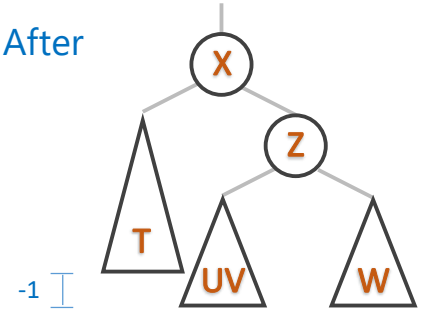



Before



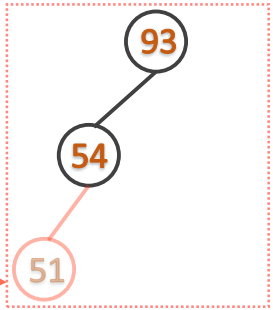

After



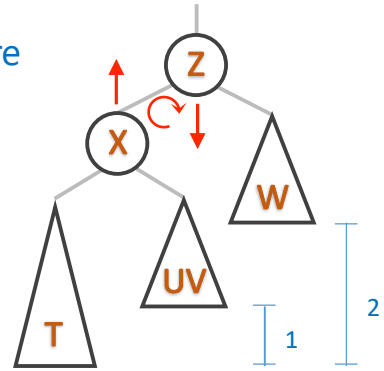

*Source: UC Davis, ECS 36C course, Spring 2020*

# AVL Trees – Rebalancing – (RR)

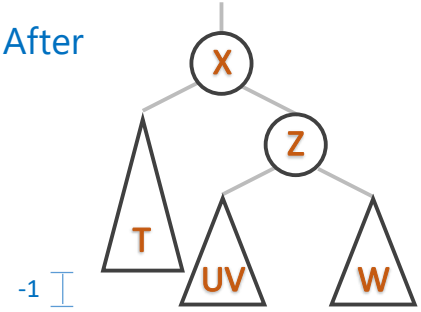
- Single Rotation Right (RR)
- Unbalance caused by either
  - Height decrease in subtree W (removal)
  - Height increase in subtree T (insertion)
- Need a right rotation to rebalance left-heavy tree
  - $O(1)$ operation




Before



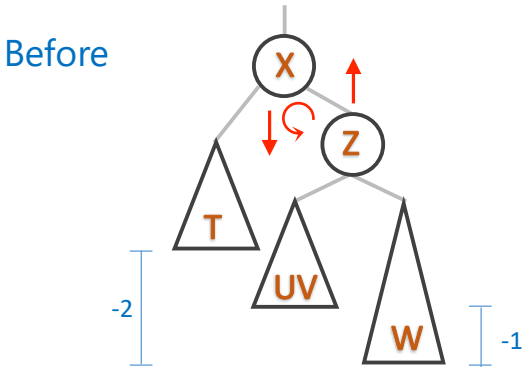

After

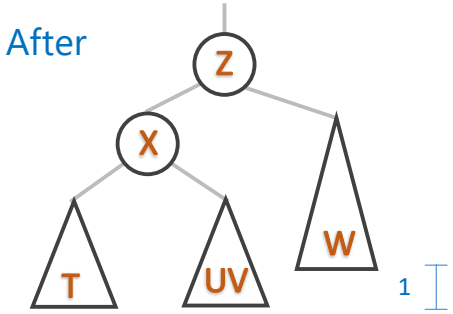



*Source: UC Davis, ECS 36C course, Spring 2020*

# AVL Trees – Rebalancing – (LL)

- Single Rotation Left (LL)
- Unbalance caused by either
  - Height decrease in subtree T (removal)
  - Height increase in subtree W (insertion)
- Need a left rotation to rebalance right-heavy tree
  - $O(1)$ operation

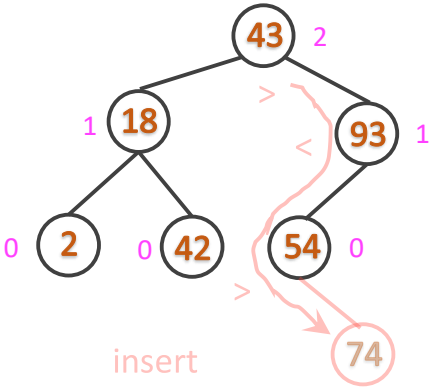


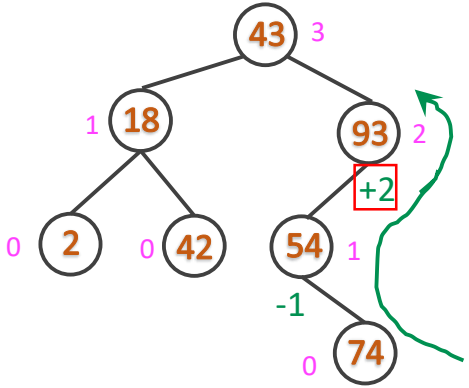

*Source: UC Davis, ECS 36C course, Spring 2020*

# AVL Trees – Complex Rebalancing

Insertion – first step is same regular BST insertion (recursive)



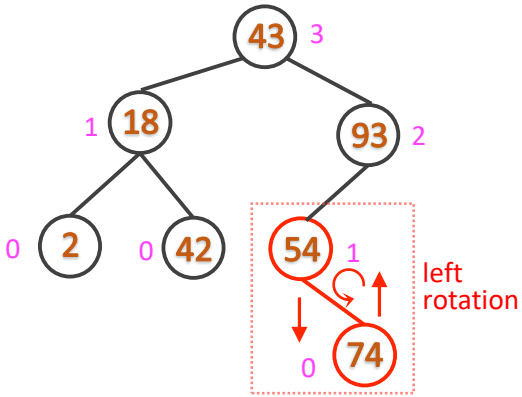

**Retracing** – child's balance factor has opposite sign



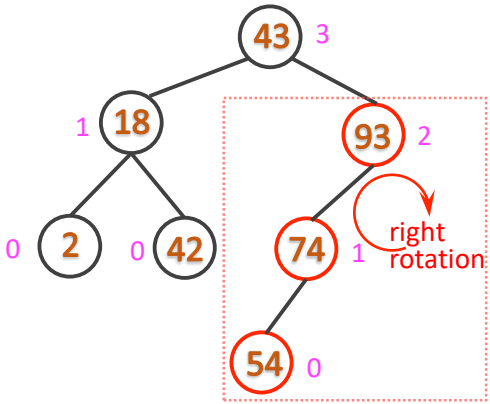

*Source: UC Davis, ECS 36C course, Spring 2020*

# AVL Trees – Complex Rebalancing

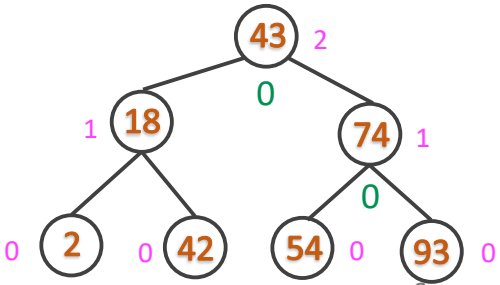
- **Retracing** – Need to first rotate child with heavy grandchild




- Only then rotate unbalanced parent with heavy child




Tree is being back to AVL

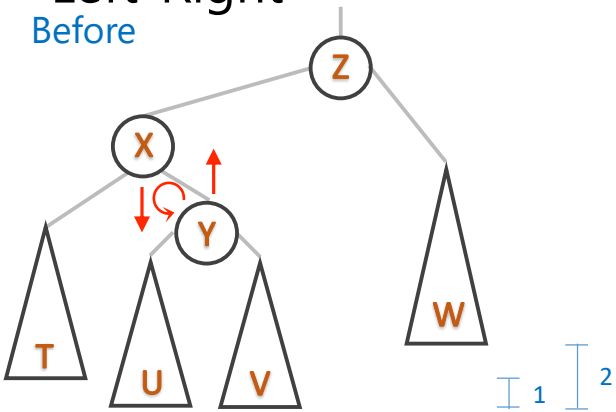



*Source: UC Davis, ECS 36C course, Spring 2020*

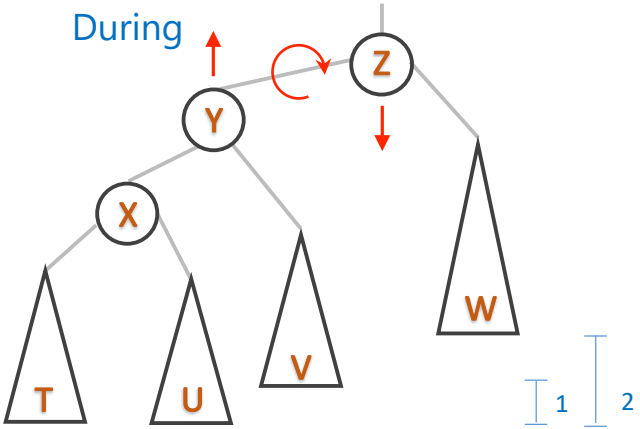
# Double Rebalancing – Left-Right

- Double Rotation: Left-right (LR)
- Unbalance caused by either
  - Insertion of node Y
  - Height decrease in subtree W (removal)
  - Height increase in subtrees U or V (insertion)
- Need a left rotation to rebalance inner right-heavy subtree
- Followed by a right rotation to rebalance left-heavy tree

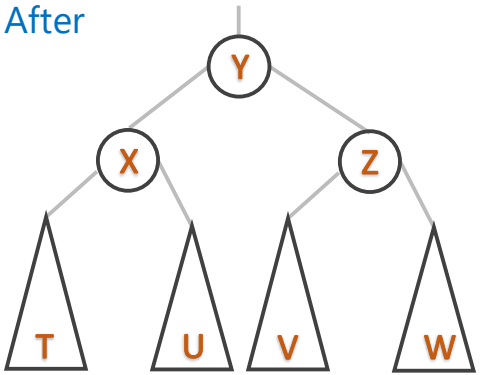
Before




During




After

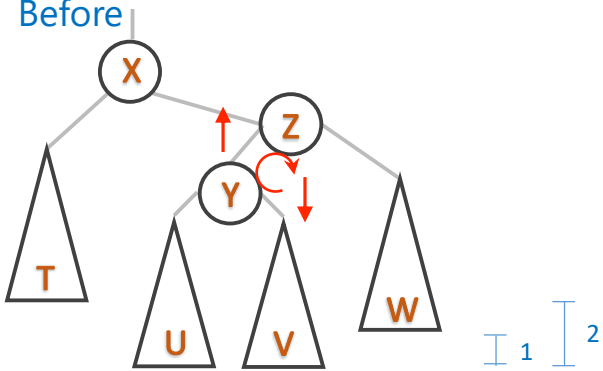



*Source: UC Davis, ECS 36C course, Spring 2020*

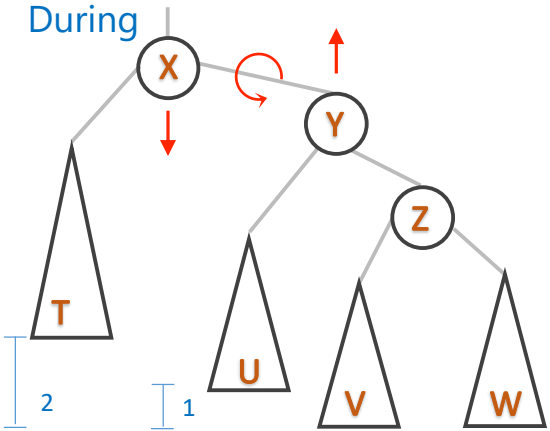
# Double Rebalancing – Right-Left

- Double Rotation: Right-left (RL)
- Unbalance caused by either
  - Insertion of node Y
  - Height decrease in subtree T (removal)
  - Height increase in subtrees U or V (insertion)
- Need a right rotation to rebalance inner left-heavy subtree
- Followed by a left rotation to rebalance right-heavy tree

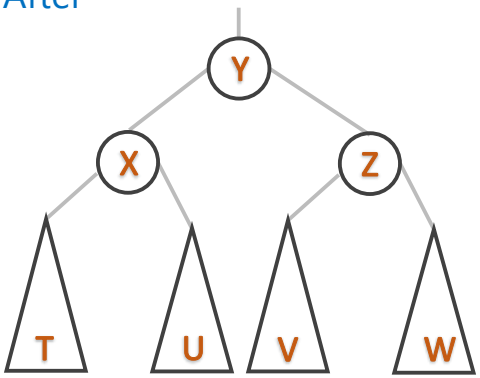
Before




During




After




*Source: UC Davis, ECS 36C course, Spring 2020*

# AVL Trees – Implementation

- Based on BST and unchanged for all query methods
  - Contains(), Min(), Max()
- Node structure increased to contain the height information
- A few extra internal methods for self-balancing purposes

```
struct node
{
  int key;
  int height;
  struct node* left;
  struct node* right;
};
```

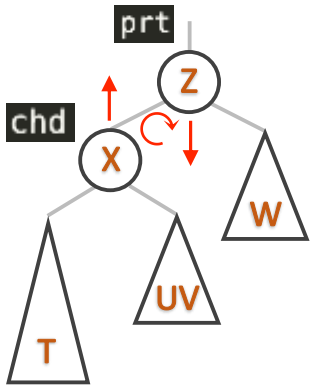
```
// Helper function to get the height of a node
int getHeight(struct node* node){
  // NIL nodes
  if (node == NULL)
    return -1;
  // Regular nodes
  return node->height;
}
```

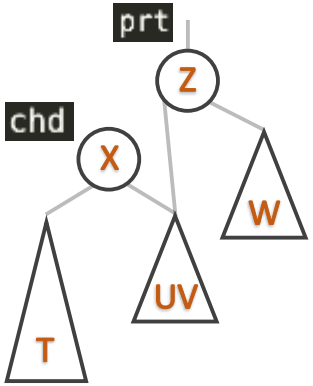
```
// Helper methods for self-balancing
int getHeight(struct node* node)
struct node* rightRotate(struct node* prt)
struct node*  leftRotate(struct node* prt)
struct node* retrace(struct node* node)
```

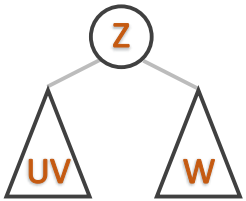
*Source: UC Davis, ECS 36C course, Spring 2020*

# Implementation – Right Rotation

```
// Helper function for right rotate
struct node* rightRotate(struct node* prt){
  struct node* chd = prt->left;
  prt->left = chd->right;
  prt->height = 1 + max(getHeight(prt->left),getHeight(prt->right));
  chd->right = prt;
  chd->height = 1 + max(getHeight(chd->left),getHeight(chd->right));
  prt = chd;
  return prt;
}
```



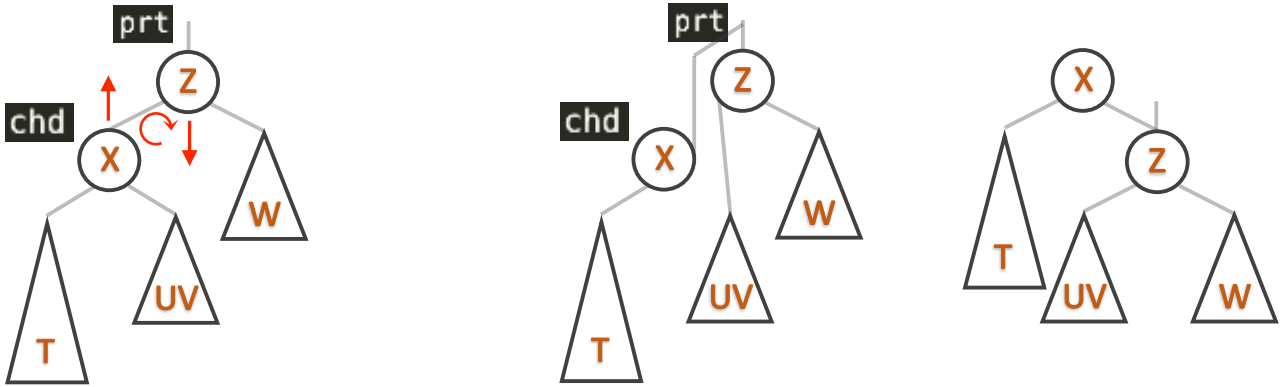




*Source: UC Davis, ECS 36C course, Spring 2020*

# Implementation – Right Rotation

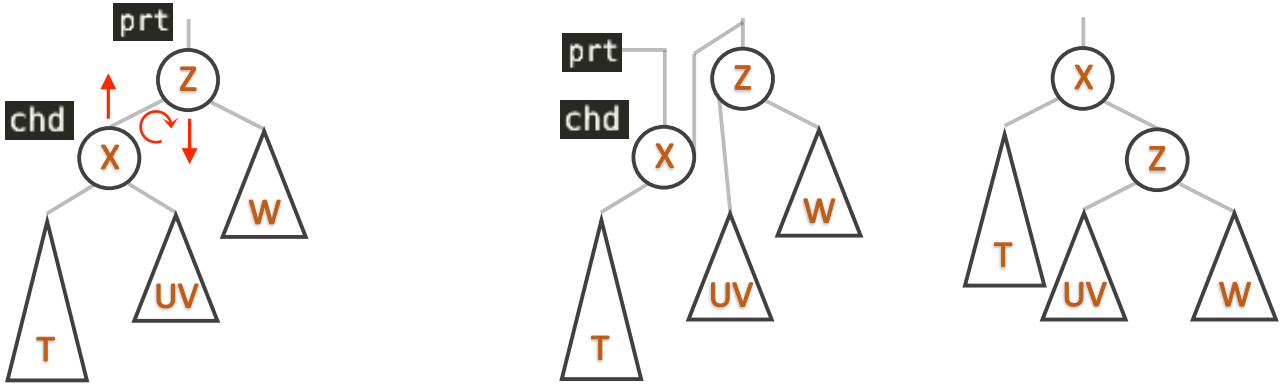
```
// Helper function for right rotate
struct node* rightRotate(struct node* prt){
  struct node* chd = prt->left;
  prt->left = chd->right;
  prt->height = 1 + max(getHeight(prt->left),getHeight(prt->right));
  chd->right = prt;
  chd->height = 1 + max(getHeight(chd->left),getHeight(chd->right));
  prt = chd;
  return prt;
}
```




*Source: UC Davis, ECS 36C course, Spring 2020*

# Implementation – Right Rotation

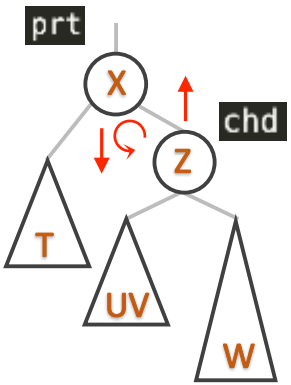
```
// Helper function for right rotate
struct node* rightRotate(struct node* prt){
  struct node* chd = prt->left;
  prt->left = chd->right;
  prt->height = 1 + max(getHeight(prt->left),getHeight(prt->right));
  chd->right = prt;
  chd->height = 1 + max(getHeight(chd->left),getHeight(chd->right));
  prt = chd;
  return prt;
}
```

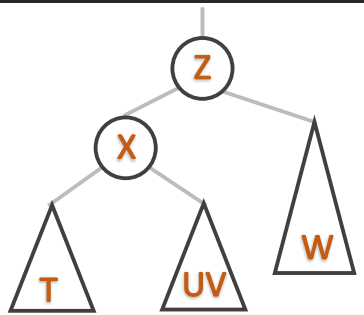



*Source: UC Davis, ECS 36C course, Spring 2020*

# Implementation – Left Rotation

```
// Helper function for left rotate
struct node* leftRotate(struct node* prt){
  struct node* chd = prt->right;
  prt->right = chd->left;
  prt->height = 1 + max(getHeight(prt->left),getHeight(prt->right));
  chd->left = prt;
  chd->height = 1 + max(getHeight(chd->left),getHeight(chd->right));
  prt = chd;
  return prt;
}
```




*Source: UC Davis, ECS 36C course, Spring 2020*

# Implementation – Insert/Remove

- Almost same body as BST
- Additional **retracing step** on the way back from the recursion

```
// function to insert a new node with given key in AVL
struct node* insert(struct node* node, int key){
  // If we reach a NULL pointer, this is the place to insert
  if (node == NULL)
    return retrace(newNode(key));
  // otherwise, recur down the tree
  if (key < node->key)
    node->left = insert(node->left, key);
  else if (key > node->key)
    node->right = insert(node->right, key);
  else
    printf("Key %d is already present!\n", key);
  // Retrace
  node = retrace(node);
  // Return the (unchanged) node pointer
  return node;
}
```

*Source: UC Davis, ECS 36C course, Spring 2020*

# Implementation – Insert/Remove

- Almost same body as BST
- Additional **retracing step** on the way back from the recursion

```
struct node* delete(struct node* node, int key){
    // key not found
    if (node == NULL){return NULL;}
    // otherwise, recur down the tree, if smaller then go left, otherwise right
    if (key < node->key)
        node->left = delete(node->left, key);
    else if (key > node->key)
        node->right = delete(node->right, key);
    // if key is same as root->key, then this is the node to be deleted
    else {
        // if leaf node, remove it
        if((node->left == NULL) && (node->right == NULL)){free(node); return NULL;}
        // if node with left child only
        else if((node->left != NULL) && (node->right == NULL)){struct node* temp = node->left; free(node); return temp;}
        // if node with right child only
        else if((node->left == NULL) && (node->right != NULL)){struct node* temp = node->right; free(node); return temp;}
        // if two children: replace with min node in right subtree
        if((node->left != NULL) && (node->right != NULL)){
            struct node* temp = getMin(node->right);
            node->key = temp->key;
            // delete the min node in the right subtree
            node->right = delete(node->right, temp->key);
        }
    }
    // Retrace
    node = retrace(node);
    return node;
}
```

*Source: UC Davis, ECS 36C course, Spring 2020*

# Implementation – Retracing

```
// The retrace function
struct node* retrace(struct node* node){
  // If NIL node no need to anything
  if(node == NULL)
    return node;
  // If tree is left-heavy, needs rebalancing
  if((getHeight(node->left) - getHeight(node->right)) > 1){
    // If left subtree is right-heavy, needs double rotation
    if(getHeight(node->left->left) < getHeight(node->left->right))
      node->left = leftRotate(node->left);
    node = rightRotate(node);
  }
  // If tree is right-heavy, needs rebalancing
  else if((getHeight(node->right) - getHeight(node->left)) > 1){
    // If right subtree is left-heavy, needs double rotation
    if(getHeight(node->right->left) > getHeight(node->right->right))
      node->right = rightRotate(node->right);
    node = leftRotate(node);
  }
  // Adjust node's height
  node->height = 1 + max(getHeight(node->left),getHeight(node->right));
  return node;
}
```

*Source: UC Davis, ECS 36C course, Spring 2020*

# Demo Time

- Insertion of ordered values

```
struct node* root = NULL;
int a[] = {2, 18, 42, 43, 51, 54, 74, 93, 99};
for(int i=0;i<9;i++){
  root = insert(root, a[i]);
  printTree(root, 0);
  printf("\n");
}
```

```
(base) dasabir@ABIRs-MacBook-Pro 06_Trees % ./a.out
2 [0]
2 [0] 18 [1]
2 [1] 18 [0] 42 [1]
2 [1] 18 [0] 42 [1] 43 [2]
2 [1] 18 [0] 42 [2] 43 [1] 51 [2]
2 [2] 18 [1] 42 [2] 43 [0] 51 [1] 54 [2]
2 [2] 18 [1] 42 [2] 43 [0] 51 [2] 54 [1] 74 [2]
2 [2] 18 [1] 42 [2] 43 [0] 51 [2] 54 [1] 74 [2] 93 [3]
2 [2] 18 [1] 42 [2] 43 [0] 51 [2] 54 [1] 74 [3] 93 [2] 99 [3]
```

*Source: UC Davis, ECS 36C course, Spring 2020*

# Demo Time

- Deletion
- Repetitive insertion and deletion for 100 times

*Source: UC Davis, ECS 36C course, Spring 2020*

# AVL Trees - Conclusion

- Average and worst case running times

| Search | Insert | Delete | Find min/max |
|---|---|---|---|
| $O(\log N)$ | $O(\log N)$ | $O(\log N)$ | $O(\log N)$ |

- Very efficient for lookups because well-balanced
  - **At most**, height of $\approx 1.44 \log n$
- Rigid balancing which can potentially slow down insertions and removals

*Source: UC Davis, ECS 36C course, Spring 2020*

# Thank You