STEVENS
INSTITUTE OF TECHNOLOGY
1870

CS590/CPE590

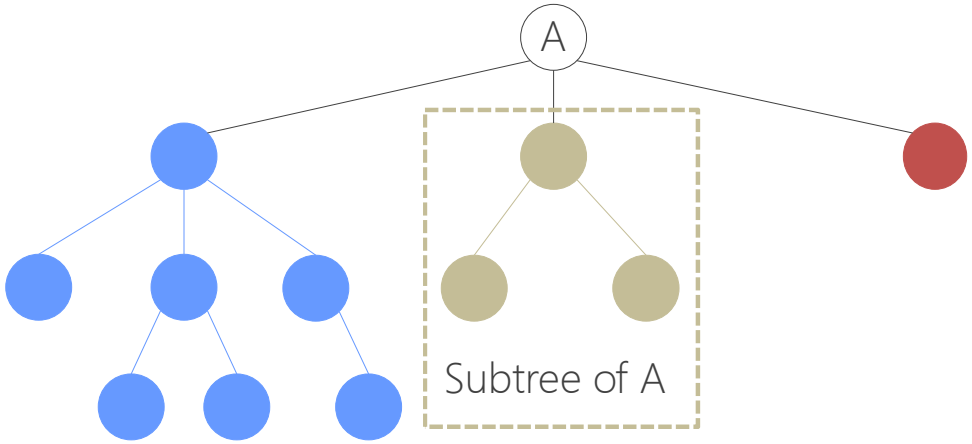
Tree | BST | 2-3 Tree

Kazi Lutful Kabir

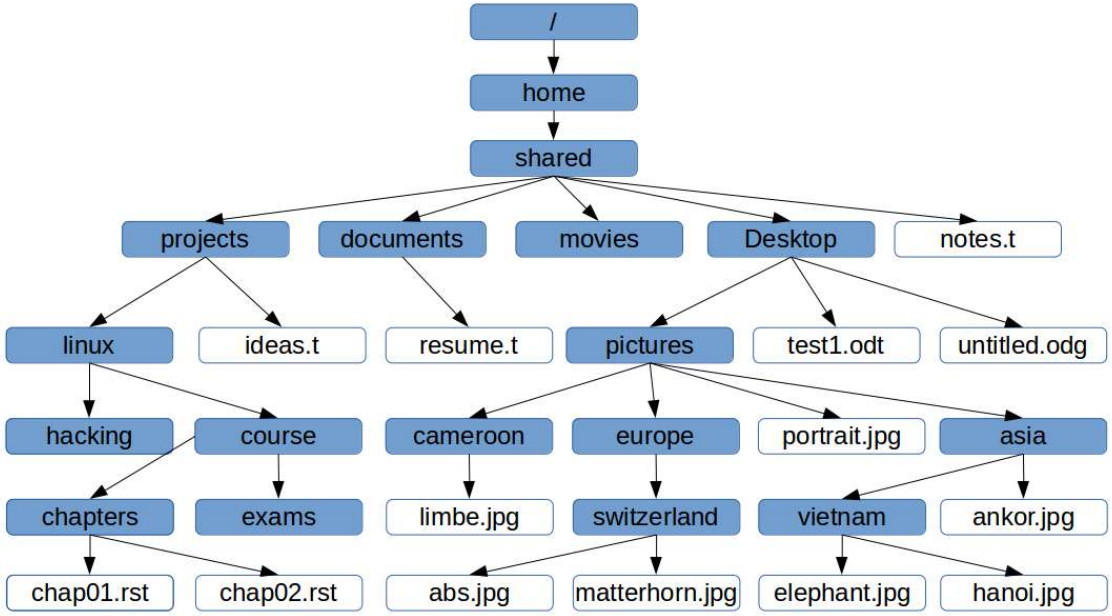
Spring 2023

Trees

- A tree is a collection of nodes
  - The collection can be empty
  - (recursive definition) If not empty, a tree consists of a distinguished node r (the root), and zero or more nonempty subtrees T_1, T_2, ..., T_k, each of whose roots are connected by an edge from root
- Special case of a connected acyclic graph
- An undirected graph having a unique path for every pair of distinct vertices

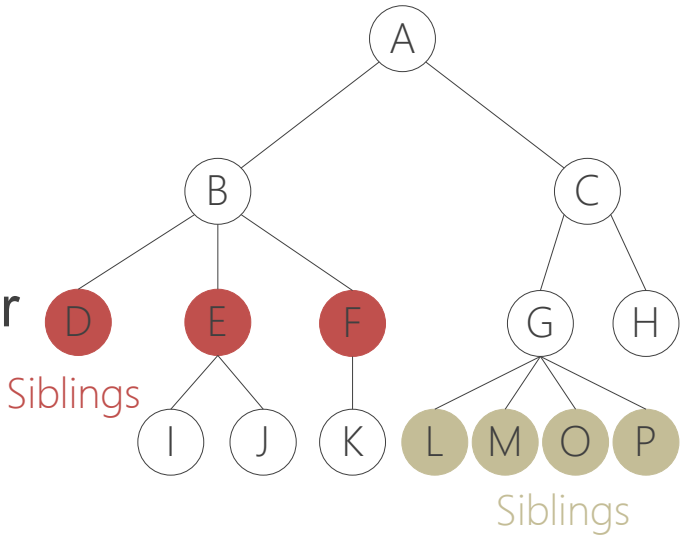


Example: UNIX Directory



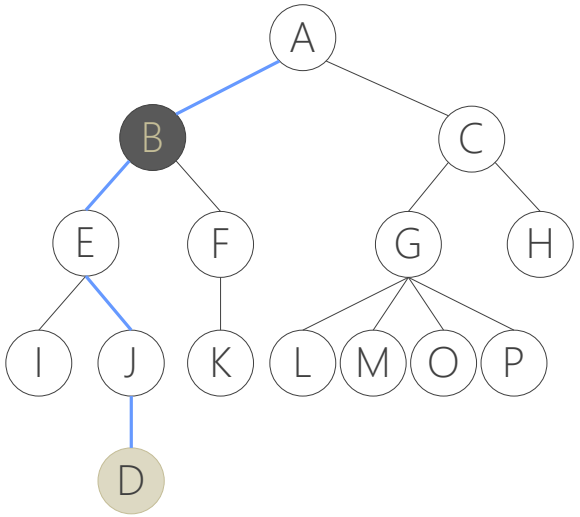
Some Terminologies

- Root and Leaf
- Child and Parent
  - Every node except the root has one parent
  - A node can have zero or more children
  - A leaf node has no children
  - A node which is not a leaf is an internal node
- Sibling
  - nodes with same parent



More Terminologies

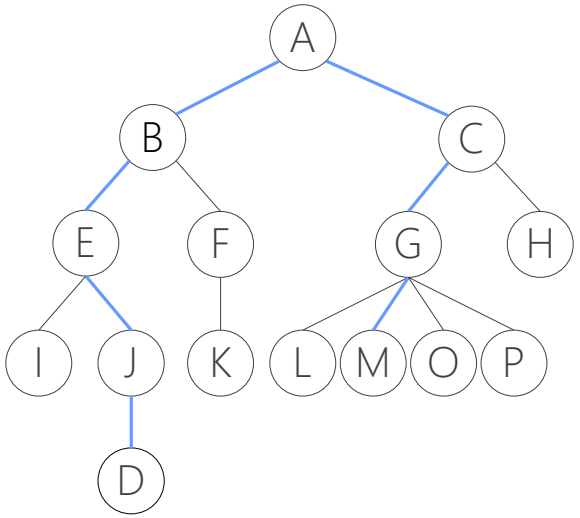
- Path
  - a sequence of edges
- Length of a path
- Depth of a node
  - length of the unique path to the root
- Height of a node
  - length of the longest path to a leaf
- Height of a tree
  - the height of the root
  - the depth of the deepest leaf
    - a lone root has a height of zero
    - an empty tree has a height of -1
- Ancestor and descendant
  - If there is a path from n1 to n2
  - n1 is an ancestor of n2, n2 is a descendant of n1



Length of the blue path = 4
Depth(B) = 1
Height(B) = 3
B is D's Ancestor
D is B's Descendant

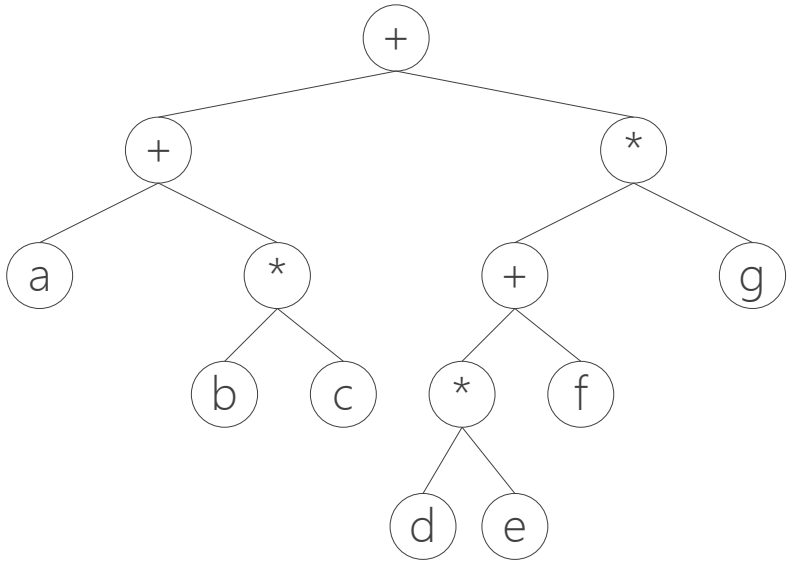
More Terminologies

- Size of a tree
  - The number of nodes in the tree
- Maximum width of a tree
  - the maximum number of nodes having the same depth
- Diameter of a tree
  - The length of the longest path between any two nodes (not taking the direction of edges into account)
    - May or may not go through the root



Size = 15
Width = 7
Diameter (blue path) = 7

Example: Expression Trees

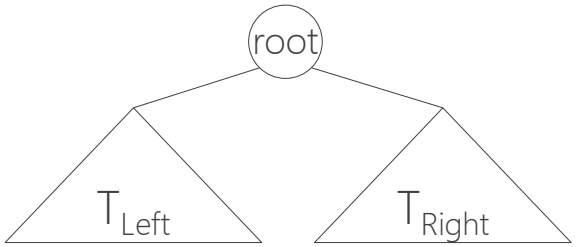


Expression tree for: (a + b*c) + (d*e + f) * g

- Leaves are operands (constants or variables)
- The internal nodes contain operators
- Will not be a binary tree if some operators are not binary

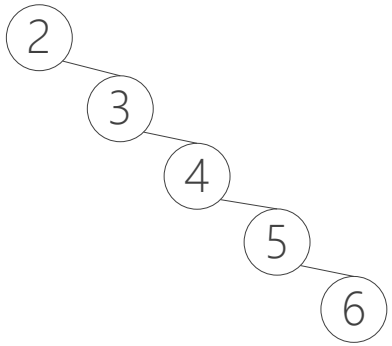
Binary Trees

- A tree in which no node can have more than two children



Generic binary tree

- The depth of an "average" binary tree is considerably smaller than N, even though in the worst case, the depth can be as large as N – 1.



Worst-case binary tree

Binary Tree Traversal

Three strategies for tree nodes enumeration

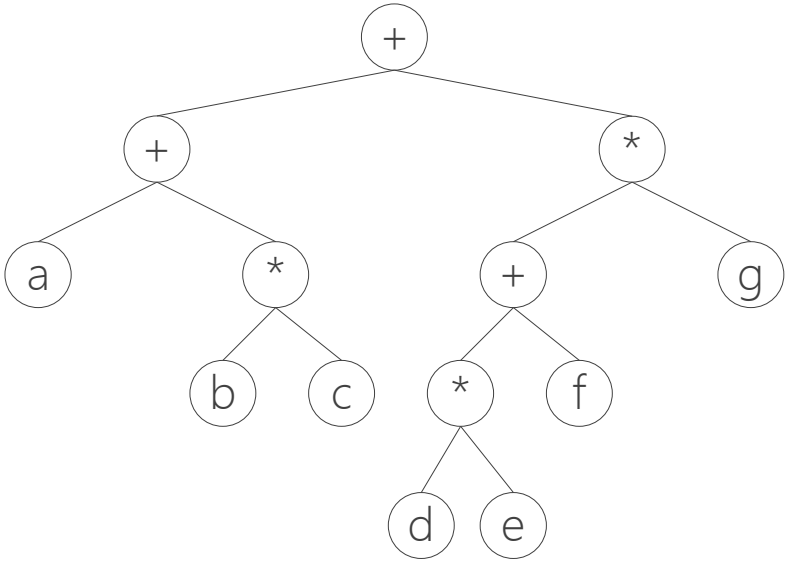
- Pre-order traversal
  - Recursive algorithm
  - First visit the root, then the left subtree, then the right
- In-order traversal
  - Recursive algorithm
  - First visit the left subtree, then the root, then the right subtree
- Post-order traversal
  - Recursive algorithm
  - First visit the left subtree, then the right, then the root

Running time: Θ(number of nodes)

Pre-order Traversal

- node, left, right
- prefix expression

  ++a*bc*+*defg



Expression tree for: (a + b*c) + (d*e + f) * g

Post-order Traversal

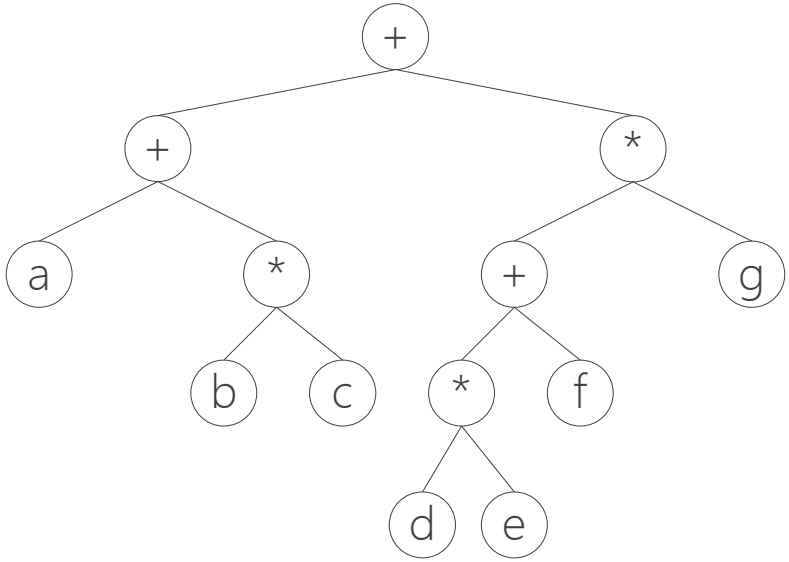
- left, right, node
- postfix expression

abc*+de*f+g*+

In-order Traversal

- left, node, right
- infix expression

a+b*c+d*e+f*g



Expression tree for: (a + b*c) + (d*e + f) * g

Pseudo Code for Pre-order, In-order and Post-order

```
PREORDER(root)
1.  IF root = Null
2.      return
3.  PRINT(root)
4.  PREORDER(LEFT(root))
5.  PREORDER(RIGHT(root))
```

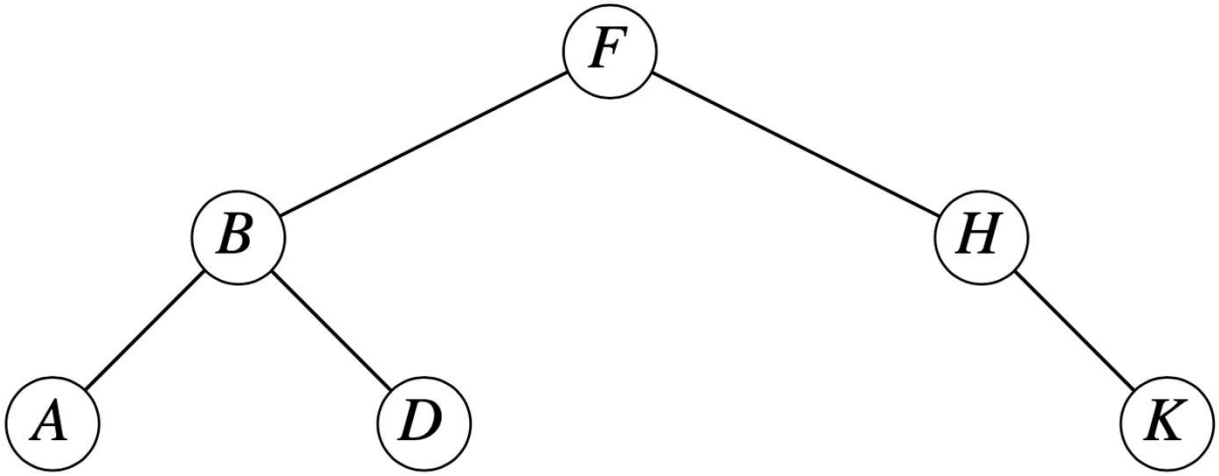
```
INORDER(root)
1.  IF root = Null
2.      return
3.  INORDER(LEFT(root))
4.  PRINT(root)
5.  INORDER(RIGHT(root))
```

```
POSTORDER(root)
1.  IF root = Null
2.      return
3.  POSTORDER(LEFT(root))
4.  POSTORDER(RIGHT(root))
5.  PRINT(root)
```

Binary Search Trees

- Binary search trees are an important data structure for dynamic sets.
- Accomplish many dynamic set operations in $O(h)$ time, where h = height of tree.
- We represent a **binary tree** by a linked data structure in which each node is an object.
- $T.root$ points to the root of tree T .
- Each node contains the fields
  - *key* (data/value).
  - *left*: points to left child.
  - *right*: points to right child.
  - p: points to parent. $\Rightarrow T.root.p$ = NIL.

- Stored keys must satisfy the ***binary-search-tree property***.
  - If y is in left subtree of x, then $y.key \leq x.key$.
  - If y is in right subtree of x, then $y.key \geq x.key$.



The ***binary-search-tree property*** allows us to print keys in a binary search tree in order, recursively, using an algorithm called an ***inorder-tree-walk***.
Elements are printed in monotonically increasing order.

```
INORDER-TREE-WALK(x)
if x ≠ NIL
  then INORDER-TREE-WALK(x.left)
       print x.key
       INORDER-TREE-WALK(x.right)
```

How INORDER-TREE-WALK works:

1. Check to make sure that x is not NIL.
2. Recursively, print the keys of the nodes in x's left subtree.
3. Print x's key.
4. Recursively, print the keys of the nodes in x's right subtree.

Correctness: Follows by induction directly from the binary-search-tree property.

Time: Intuitively, the walk takes $\Theta(n)$ time for a tree with n nodes, because we visit and print each node once.

Querying a binary search tree

- Need to search for a key stored in BST.
- BST can support queries such as MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR.

Searching

```
TREE-SEARCH(x, k)
if x = NIL or k = x.key
   then return x
if k < x.key
   then return TREE-SEARCH(x.left, k)
   else return TREE-SEARCH(x.right,k)
```

- Initial call is TREE-SEARCH(*T.root*, *k*).
- ***Time:*** The algorithm recurses, visiting nodes on a downward path from the root. Thus, running time is $O(h)$, where h is the height of the tree.

```
ITERATIVE-TREE-SEARCH(x, k)
while x = NIL and k = x.key
   if k < x.key
      then x = x.left
   else x = x.right
return x
```

- Unrolling the recursion into a while loop.

Minimum and maximum

The binary-search-tree property guarantees that

- the minimum key of a binary search tree is located at the leftmost node, and
- the maximum key of a binary search tree is located at the rightmost node.

Traverse the appropriate pointers (*left* or *right*) until NIL is reached.

```
TREE-MINIMUM(x)
while x.left ≠ NIL do
        x ← x.left
return x
```

```
TREE-MAXIMUM(x)
while x.right ≠ NIL do
        x ← x.right
return x
```

Time: Both procedures visit nodes that form a downward path from the root to a leaf. Both procedures run in $O(h)$ time, where h is the height of the tree.

Successor and predecessor

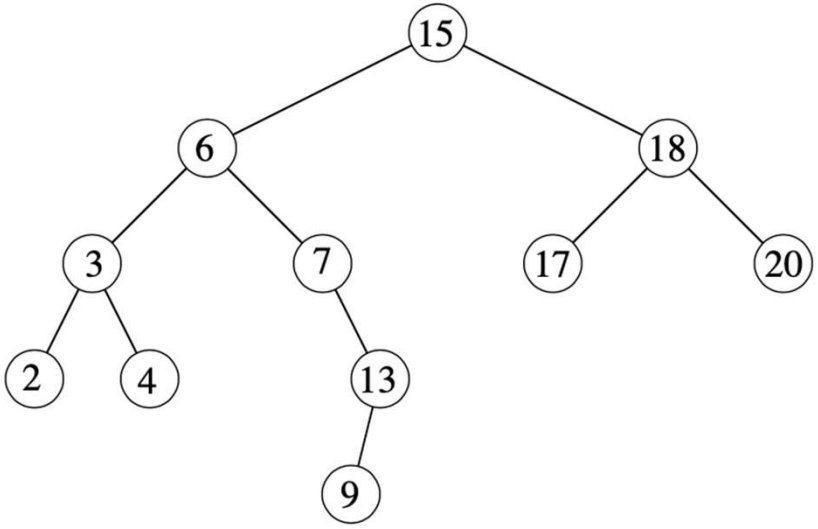
Assuming that all keys are distinct,

- the successor of a node x is the node y such that $y.key$ is the smallest $key > x.key$.
- the predecessor of a node x is the node y such that $y.key$ is the largest $key < x.key$.
- We can find x's successor based entirely on the tree structure.
- No key comparisons are necessary.
- If x has the largest key in the binary search tree, then we say that x's successor is NIL.
- If x has the smallest key in the binary search tree, then we say that x's predecessor is NIL.
- There are two cases:
  1. If node x has a non-empty right subtree, then x's successor is the minimum in x's right subtree.
  2. If node x has an empty right subtree, notice that:
     - As long as we move to the left up the tree (move up through right children), we're visiting smaller keys.
     - x's successor y is the node that x is the predecessor of (x is the maximum in y's left subtree).

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL then
   return TREE-MINIMUM(x.right)
y ← x.p
while y ≠ NIL and x = y.right do
   x ← y
   y ← y.p
return y
```

```
TREE-PREDECESSOR(x)
if x.left ≠ NIL then
   return TREE-MAXIMUM(x.left)
y ← x.p
while y ≠ NIL and x = y.left do
   x ← y
   y ← y.p
return y
```

Time: For both the TREE-SUCCESSOR and TREE-PREDECESSOR procedures, in both cases, we visit nodes on a path down the tree or up the tree. Thus, running time is $O(h)$, where h is the height of the tree.



Find the successor of the node with key value 15.
(Answer: Key value 17)
Find the successor of the node with key value 6.
(Answer: Key value 7)
Find the successor of the node with key value 4.
(Answer: Key value 6)
Find the predecessor of the node with key value 6.
(Answer: Key value 4)

Insertion and deletion

Insertion and deletion allows the dynamic set represented by a binary search tree to change.
The binary-search-tree property must hold after the change.

Insertion is more straightforward than deletion.

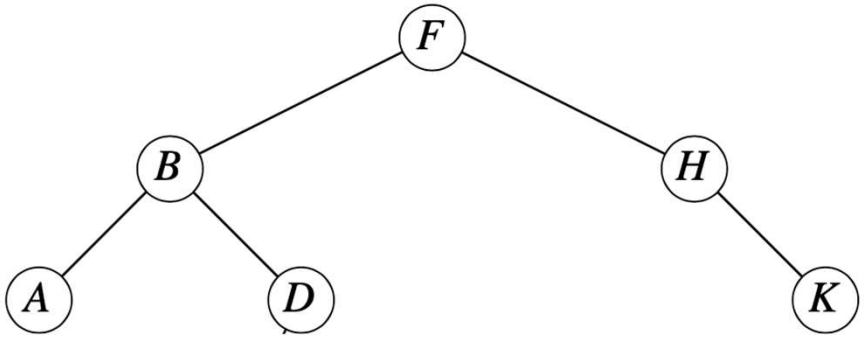
Insertion

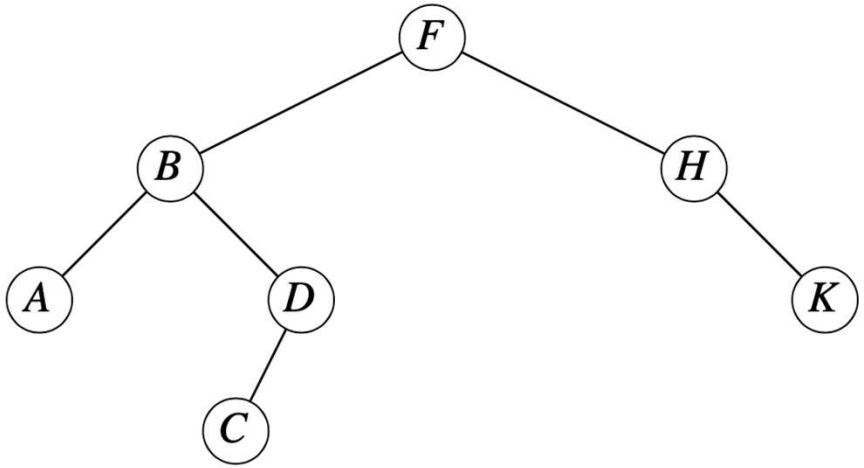
TREE-INSERT(T, z)

```
(1)  y ← NIL
(2)  x ← T.root
(3)  while x ≠ NIL do                  //while loop causes two
(4)     y ← x                          //pointers to move down
(5)     if z.key < x.key then
(6)        x ← x.left
(7)     else x ← x.right
(8)  z.p ← y
(9)  if y = NIL then
(10)    T.root ← z                     //Tree T was empty
(11)    else if z.key < y.key  then
(12)       y.left ← z
(13)    else y.right ← z
```

- To insert value v into the binary search tree, the procedure is given node z, with $z.key=v$, $z.left=$NIL, and $z.right=$NIL.
- Beginning at root of the tree, trace a downward path, maintaining two pointers.
  - Pointer x: traces downward path.
  - Pointer y: "trailing pointer" to keep track of parent of x.
- Traverse the tree downward by comparing the value of node at x with v, and move to the left or right child accordingly.
- When x is NIL, it is at the correct position for node z.
- Compare z's value with y's value, and insert z at either y's *left* or *right*, appropriately.
- ***Time:*** Same as TREE-SEARCH. On a tree of height h, procedure takes $O(h)$ time.

TREE-INSERT (T, C)

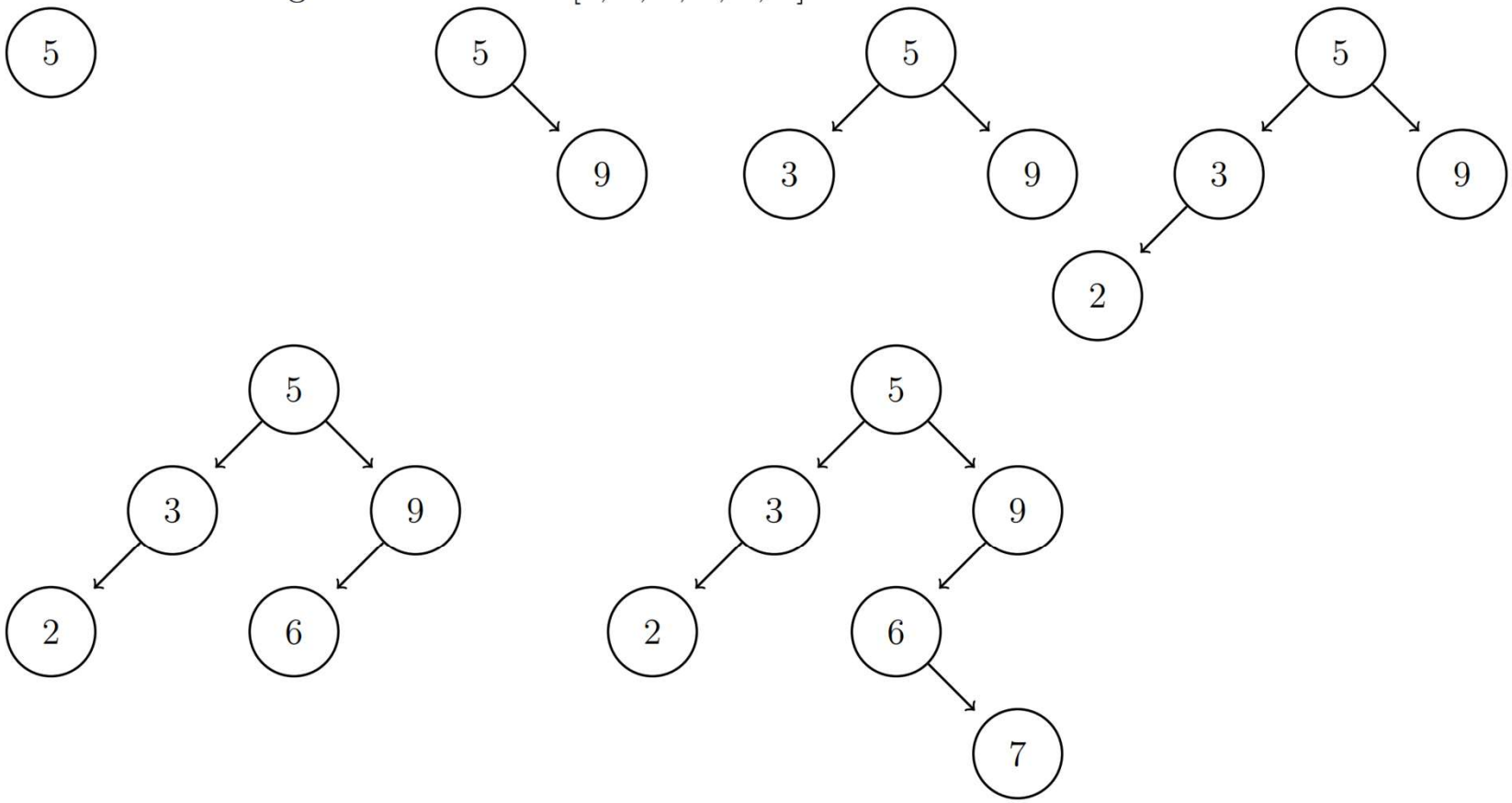




The node with value ‘C’ is going to be the new left child of the node with value ‘D’

Constructing a BST

Insert the following values in order: [5, 9, 3, 2, 6, 7]



TREE-INSERT can be used with INORDER-TREE-WALK to sort a given set of numbers.

TREE-SORT(A)
let T be an empty binary search tree
for $i \leftarrow 1$ **to** n
  do TREE-INSERT(T, $A[i]$)
INORDER-TREE-WALK($root[T]$)

Worst case: $O(n^2)$ occurs when a linear chain of nodes results from the repeated TREE-INSERT operations.
Best case: $\Omega(n \lg n)$ occurs when a binary tree of height of **lg *n*** results from the repeated TREE-INSERT operations.

Deletion

TREE-DELETE is broken into three cases.

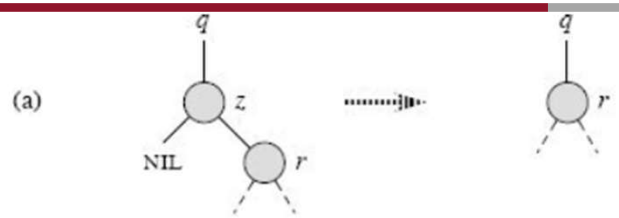
Case 1: z has no children. Delete z by making the parent of z point to NIL, instead of to z.

Case 2: z has one child. Delete z by making the parent of z point to z's child, instead of to z.

Case 3: z has two children.

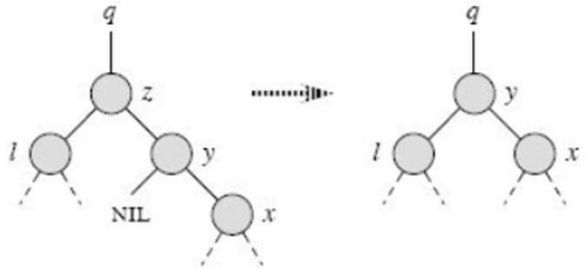
- z's successor y has either no children or one child. (y is the minimum node-with no left child-in z's right subtree.)
- Delete y from the tree (via Case 1 or 2).
- Replace z's key and data with y's.

if z has no left child, replace by its right child. The right child may or may not be NIL (If NIL -> no child)

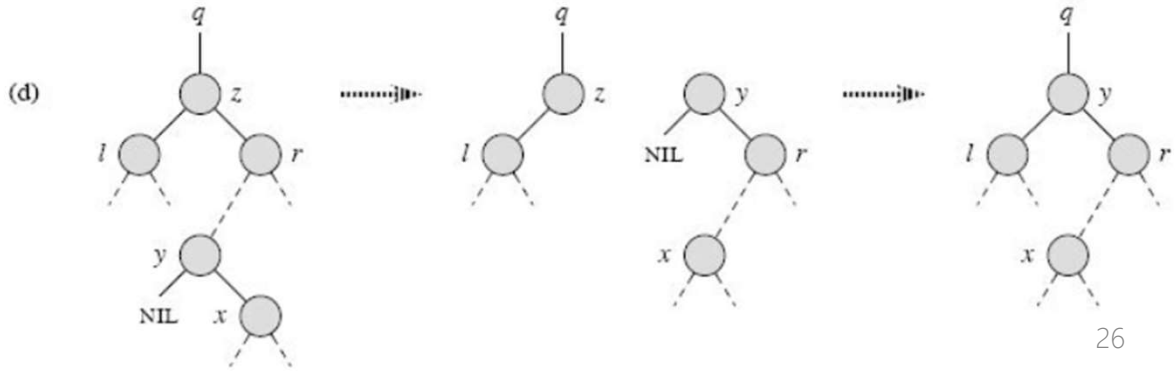


if z has just one child, the left child, then replace z by its left child.

if y is the right child of z, then replace z by y and leave the right child of y alone.

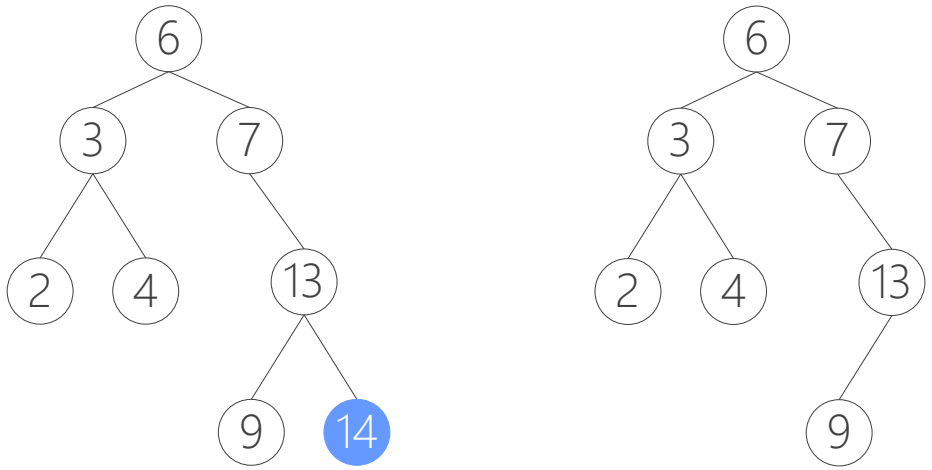


Otherwise, y lie with in the right subtree of z, but it is not the root of this subtree. We replace y by its own right child. Then we replace z by y.



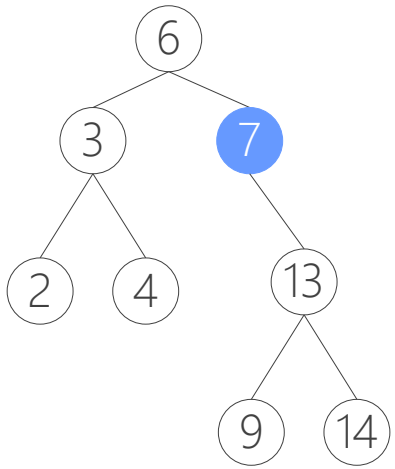
Three Delete Cases

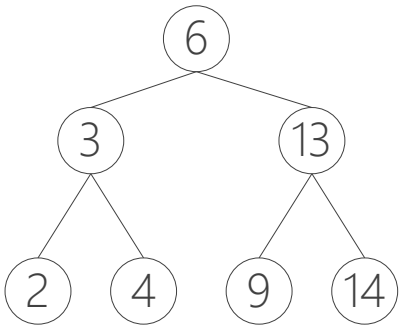
- Case 1: the node is a leaf
  - Delete it immediately



Three Delete Cases

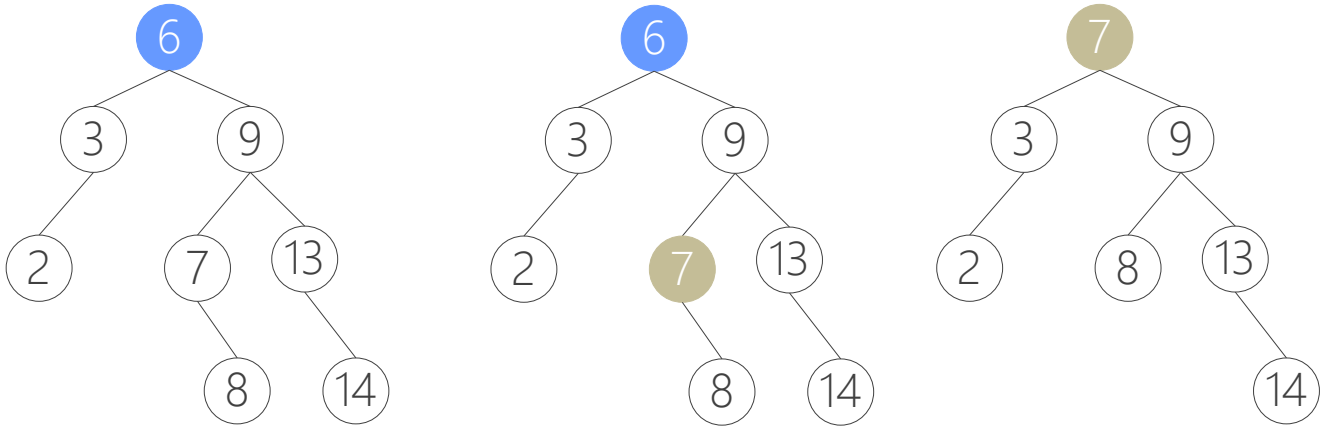
- Case 2: the node has one child
  - Adjust a pointer from the parent to bypass that node

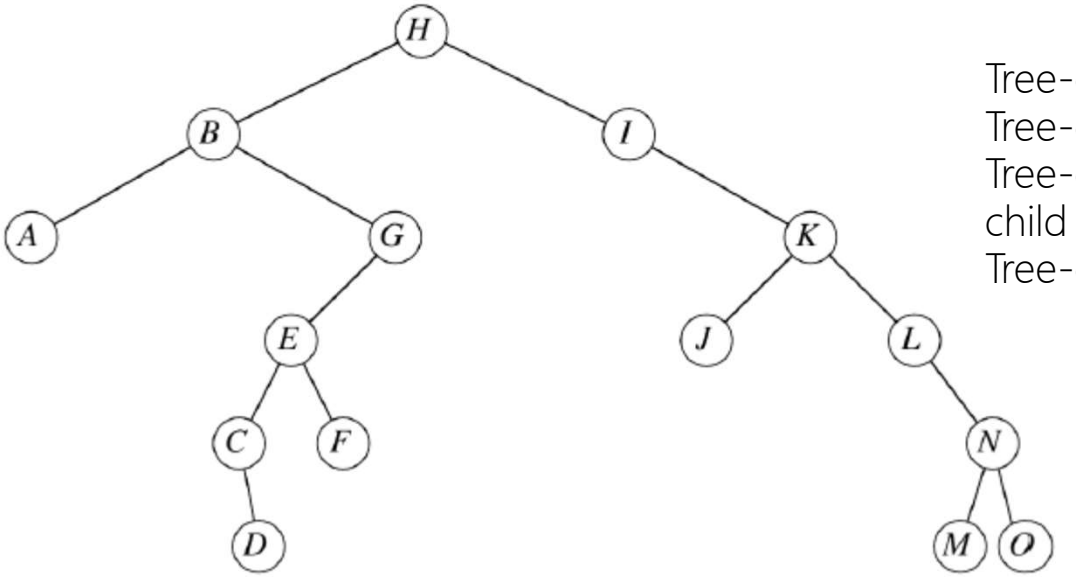




Three Delete Cases

- Case 3: the node has two children
  - Replace that node with the minimum node in the right subtree
    - Or replace it with the maximum node in the left subtree
  - This invokes removal of the minimum node
    - It's case 1 or 2.
- Time complexity = O(tree height)





Tree-delete(T,I): no left child
Tree-delete(T,G): has left, but no right child
Tree-delete(T,K): two children, successor is right child
Tree-delete(T,B): successor is not the right child.

Time: $O(h)$, on a tree of height h.

Minimizing running time

We've been analyzing running time in terms of h (the height of the binary search tree), instead of n (the number of nodes in the tree).

- Problem: Worst case for binary search tree is $\Theta(n)$ - no better than linked list.
- Solution: Guarantee small height (balanced tree) with $h = O(\lg n)$.
- Method: We restructure the tree. Querying works as before (no adjustments necessary). Insertion or deletion (changing the structure) may require some extra work.

BST Drawback

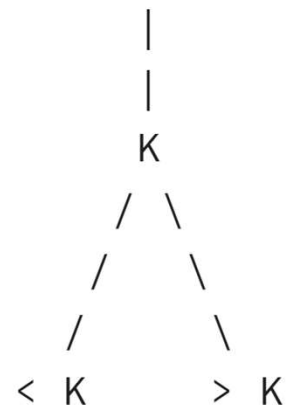
- Binary Search Trees (BST) are effective as they allow finding a key in the BST in O(tree height).

- Unfortunately, the height of a BST can be O(n) where n is the number of nodes in the BST.
  -- For example, when inserting n numbers in increasing order, the BST will be completely unbalanced and have a height of n - 1.

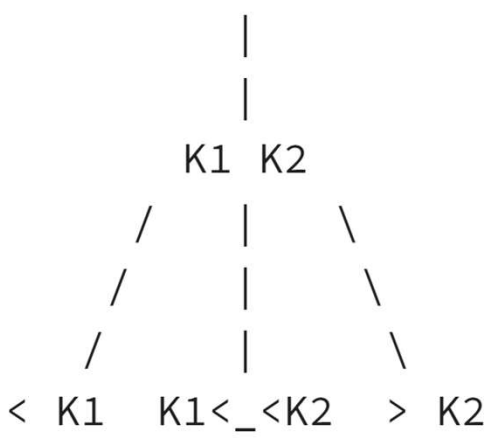
Balanced Tree: 2-3 Trees

A 2-3 tree is a tree that can have nodes of two kinds: 2-nodes and 3-nodes.

A 2-node contains a single key K and has two children, just like in a BST:



A 3-node contains two ordered keys K1 and K2 (K1 < K2) and has three children:



All the leaves must be on the same level. Therefore, a 2-3 tree is always perfectly height-balanced: the length of a path from the root to a leaf is the same for every leaf.

Properties

1. each node has either one value or two value
2. a node with one value is either a leaf node or has exactly two children (non-null). Values in left subtree < value in node < values in right subtree
3. a node with two values is either a leaf node or has exactly three children (non-null). Values in left subtree < first value in node < values in middle subtree < second value in node < value in right subtree.
4. all leaf nodes are at the same level of the tree

Lookup/Search

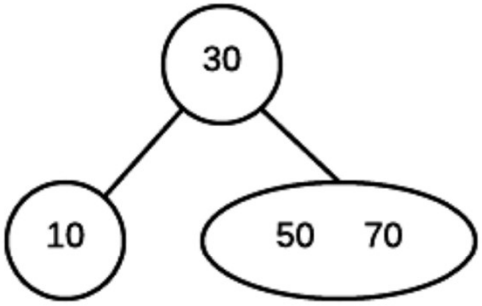
- To determine whether key value **k** is in a 2-3 tree T. The lookup operation for a 2-3 tree is very similar to the lookup operation for a binary-search tree.
- There are 2 base cases:
  -- T is empty: return false
  -- T is a leaf node: return true iff the key value in T is k
- And there are 3 recursive cases:
  -- k <= T.leftMax: look up k in T's left subtree
  -- T.leftMax < k <= T.middleMax: look up k in T's middle subtree
  -- T.middleMax < k: look up k in T's right subtree
- Time complexity for lookup is proportional to the height of the tree. The height of the tree is O(log N) for N = the number of **nodes** in the tree.

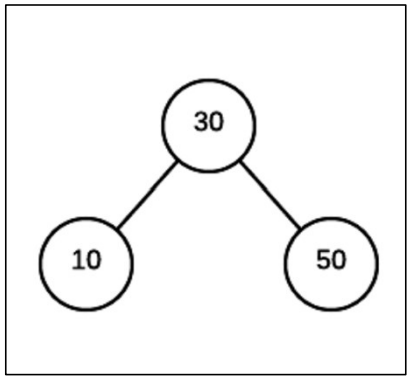
Insertion

The insertion algorithm into a two-three tree is quite different from the insertion algorithm into a binary search tree. In a two-three tree, the algorithm will be as follows:

1. If the tree is empty, create a node and put value into the node
2. Otherwise find the leaf node where the value belongs.
3. If the leaf node has only one value, put the new value into the node
4. If the leaf node has more than two values, split the node and promote the median of the three values to parent.
5. If the parent then has three values, continue to split and promote, forming a new root node if necessary

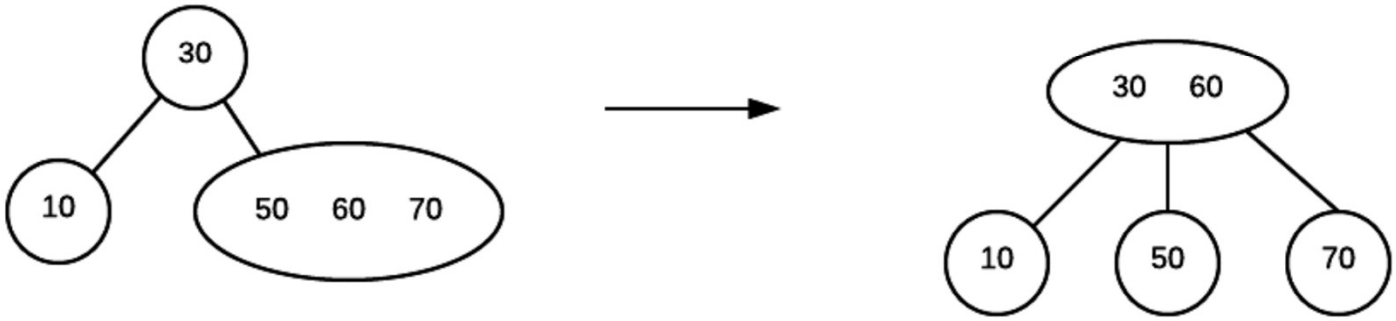
Example





Insert 70

Insert 60



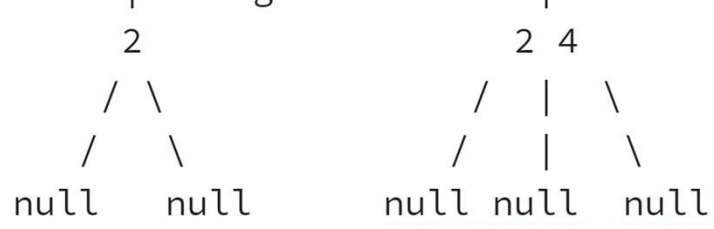
Insertion: Example

We always insert a new key K in a leaf, except for the empty tree.

The appropriate leaf is found by performing a search for K:

- If the leaf is a 2-node, we replace it with a 3-node and insert K there as either the first or the second key, depending on whether K is smaller or larger than the node's old key.

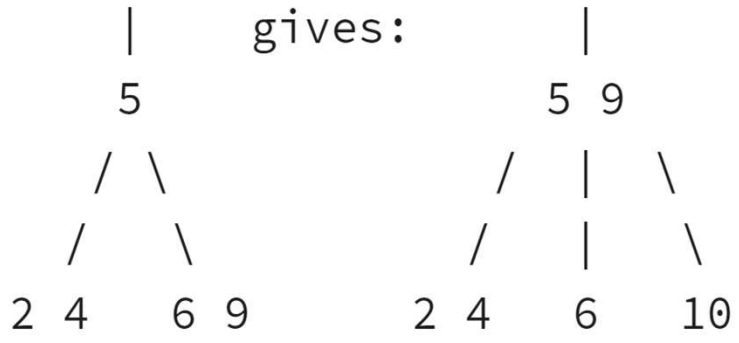
  Example: inserting 4 into



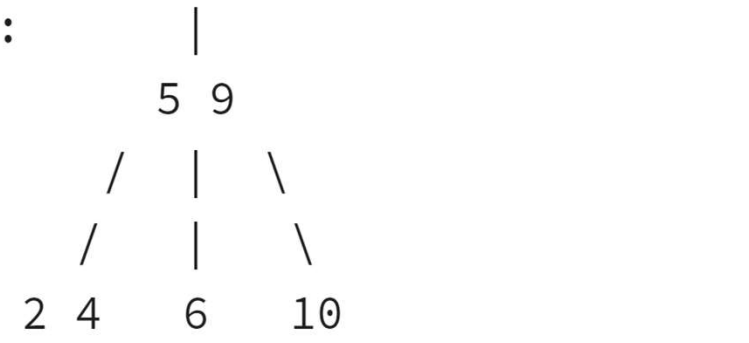
  gives:

- If the leaf is a 3-node, we split the leaf in two: the smallest of the three keys (two old ones and the new key) is put in the first leaf, the largest key is put in the second leaf, and the middle key is promoted to the old leaf's parent.

Example: inserting 10 into



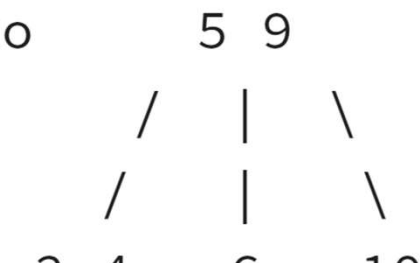
gives:



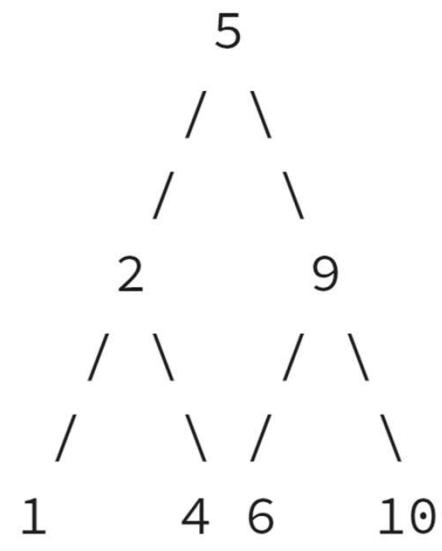
- Note that the promotion of a middle key to its parent can cause the parent's overflow (if it was already a 3-node) and hence can lead to more node splits along the chain of the leaf's ancestors.

- If a node to be split happens to be the tree's root, a new root is created to accept the middle key. The height of the tree then increases by one.

Example: inserting 1 into



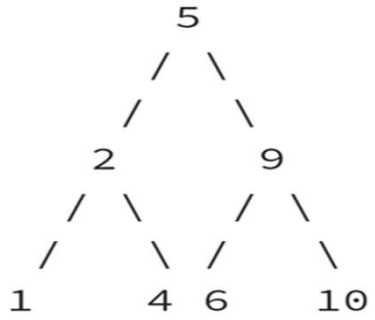
gives:



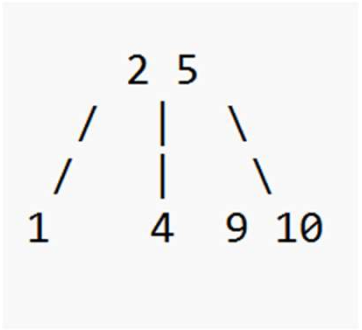
Deletion works in a similar manner, merging nodes as necessary (instead of splitting them).

Insertion (and deletion, and search) takes O(tree height) = O(lg(n)).

Deletion



Delete 6



Note that, because of the presence of 3-nodes, the height of a 2-3 tree will be on average smaller than the height of a BST containing the same values, so searching for values will be faster.
Essentially having more keys into the nodes makes the tree flatter, which speeds up search.

2-3 Trees can be generalized to contain many keys per node, giving a very flat tree with fast search. The result is B-Trees, which are often used for data storage in relational database systems.

Insertion and deletion in a 2-3 Tree can be nicely visualized here (a 2-3 Tree is the same as a B-Tree of degree 3):

https://www.cs.usfca.edu/~galles/visualization/BTree.html

Contents of this presentation are partially adapted from
My CS385 (Fall2022)
and from
Prof. In Suk Jang CS590 (Summer 2021 Lecture-6)
and are also based on
Book Chapter- 12, Introduction to Algorithms by *Cormen, Leiserson, Rivest, & Stein*

STEVENS
INSTITUTE OF TECHNOLOGY
1870

THANK **YOU**

Stevens Institute of Technology