CSN 102: DATA STRUCTURES

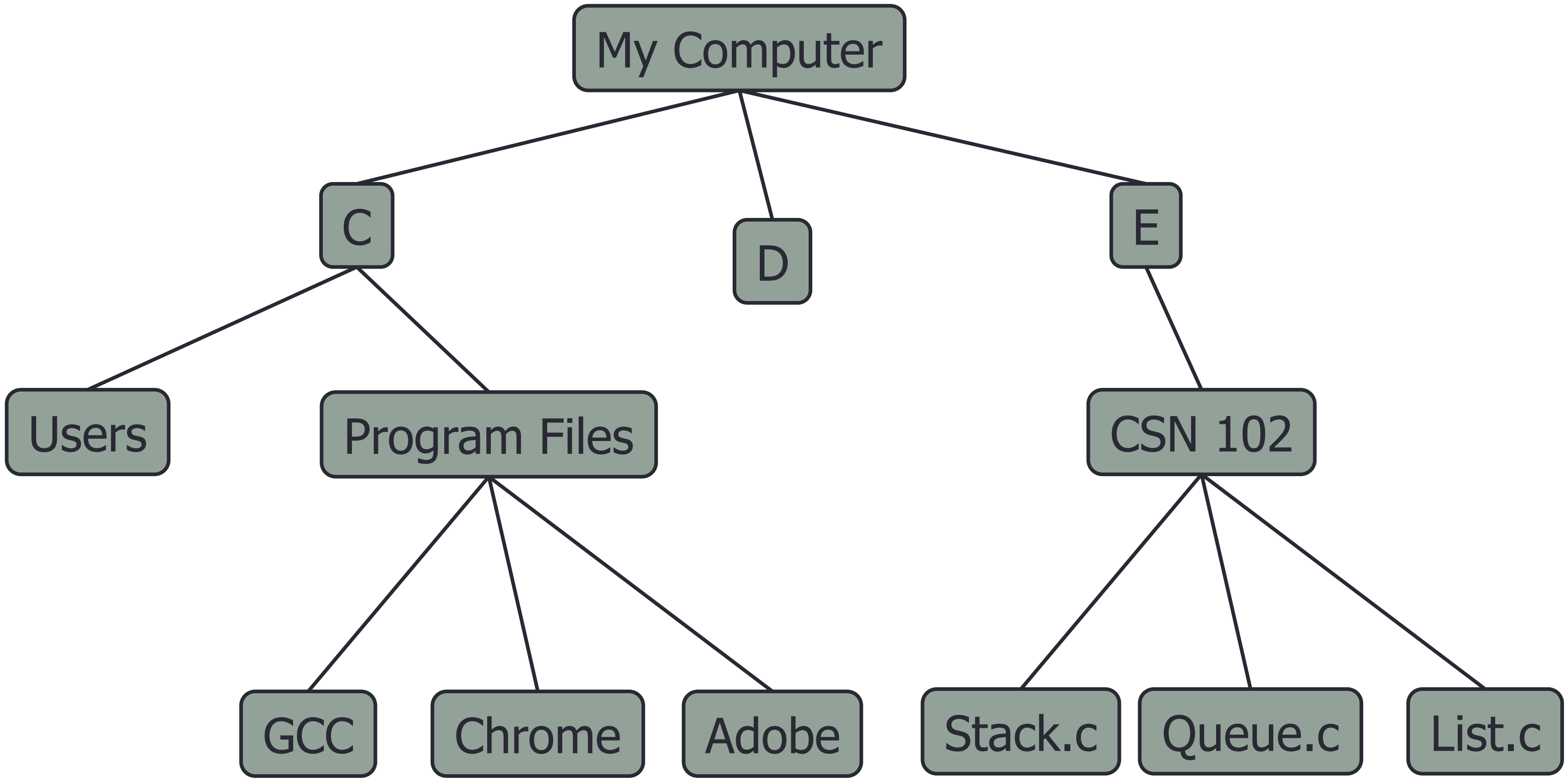
Tree: Terminology, Binary Tree, Binary Tree Traversal, Binary Search Tree, AVL tree, B Tree, Applications

What is a Tree?

- Similar to tree in general world
- Finite set of elements
- Non-linear data structure
- Used to represent hierarchical structures
- Eg. Syntax tree, Binary Search Tree, Animal Kingdom
- Application: Organization Charts, File system
- Tree can also be defined in itself as a node and list of child trees.

❖**In Computer Science, Trees grow down! ☺**

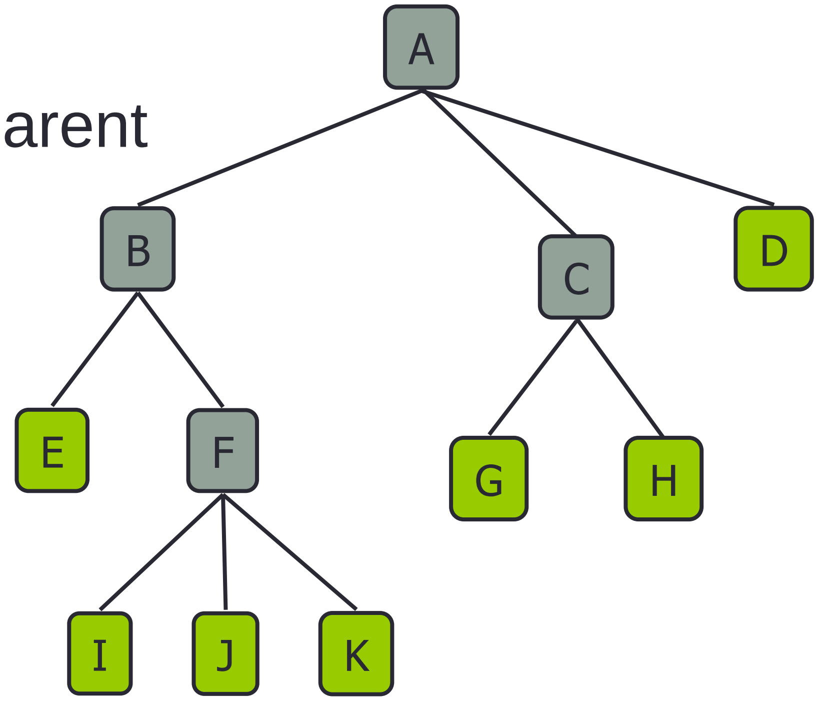
Examples


My Computer
C
D
E
Users
Program Files
CSN 102
GCC
Chrome
Adobe
Stack.c
Queue.c
List.c

Terminology

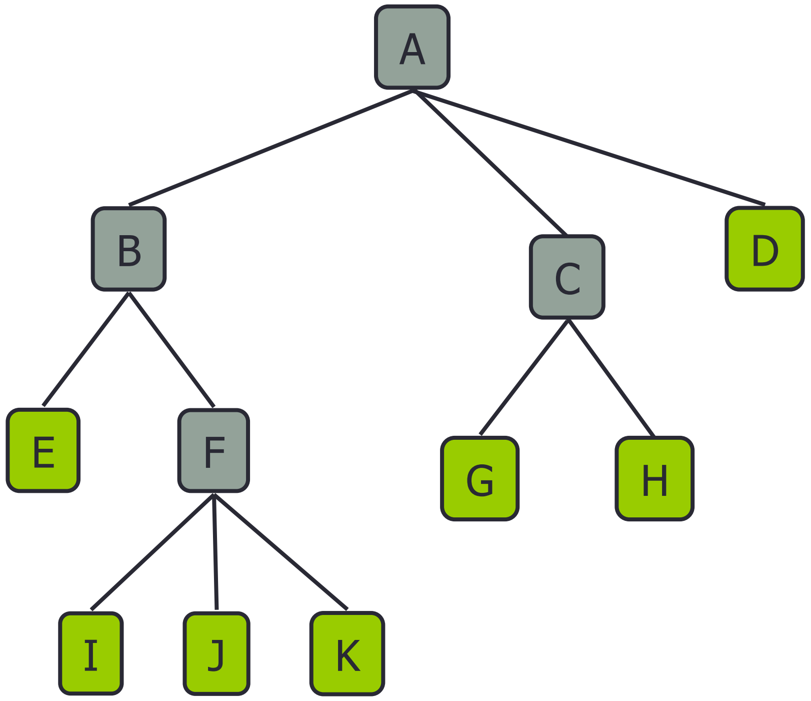
- Node: stores the key (all A, B, C… K)
- Child node: node directly connected below the parent node (B is child of A)
- Parent node: node directly connected above the child node (B is parent of E)
- Siblings: nodes sharing same parent

  (E and F are siblings)



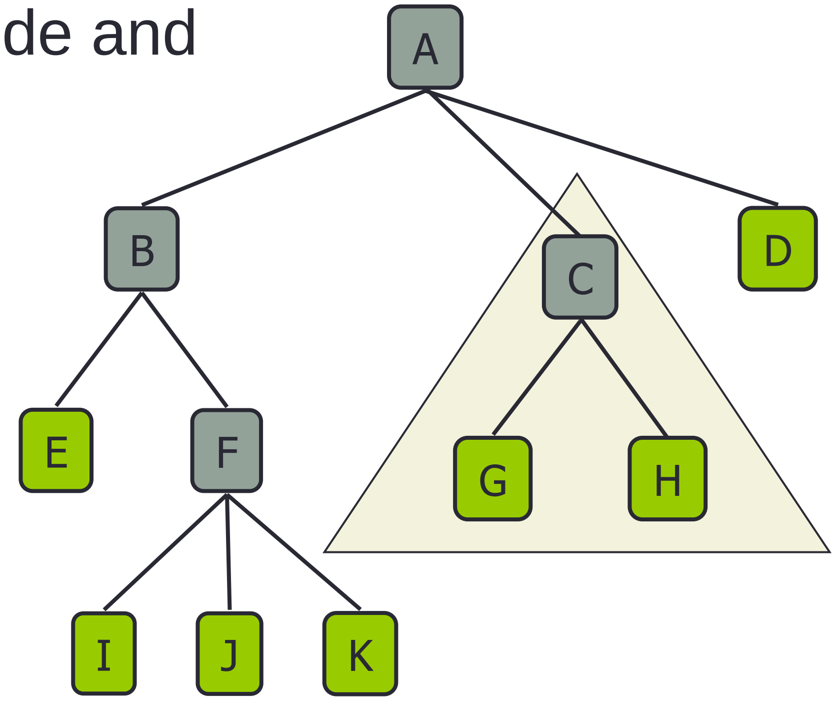
Terminology

- Root: top node in tree or node with no parent (A is root)
- Leaf: node with no child (E, I, J etc. are leaf node)
- Interior node: All non-leaf node(B, F, A etc.)



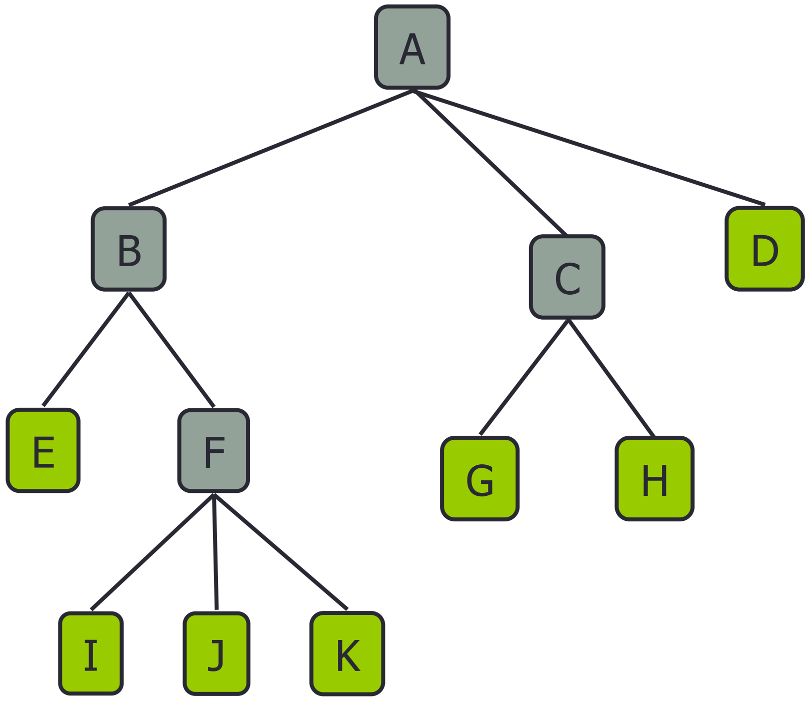
Terminology

- Ancestors: parent, parent of parent and so on. (F, B, A are ancestors of K)
- Descendants: child, child of child and so on. (E, F, I, J, K are descendants of B)
- Subtree: tree consisting of a node and its descendants



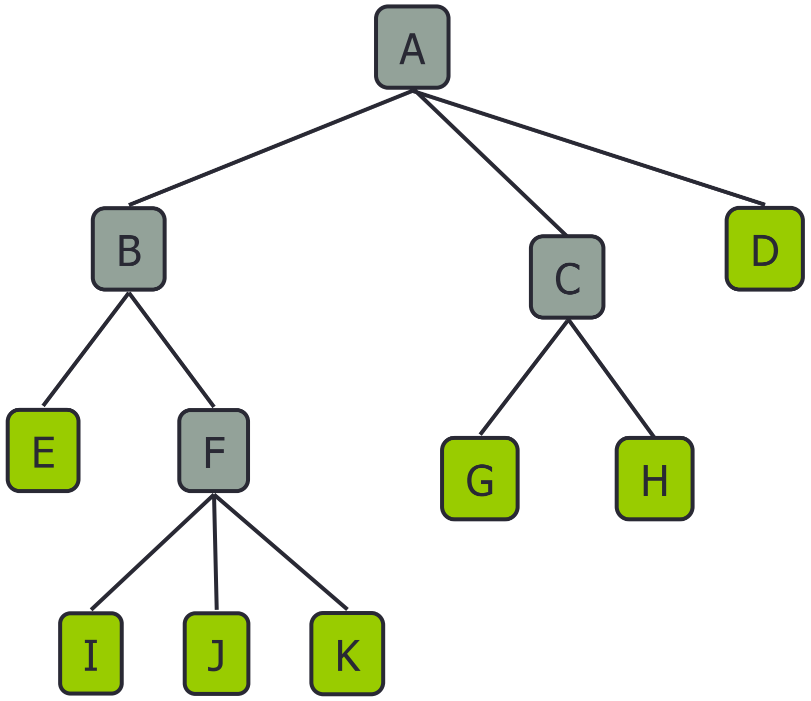
Terminology

- Degree of a node: number of its child (degree(A)=3, degree(B)=2)
- Degree of tree: maximum of degrees of all nodes
- Level: 1 + number of connection between node and root (level(F)=3)



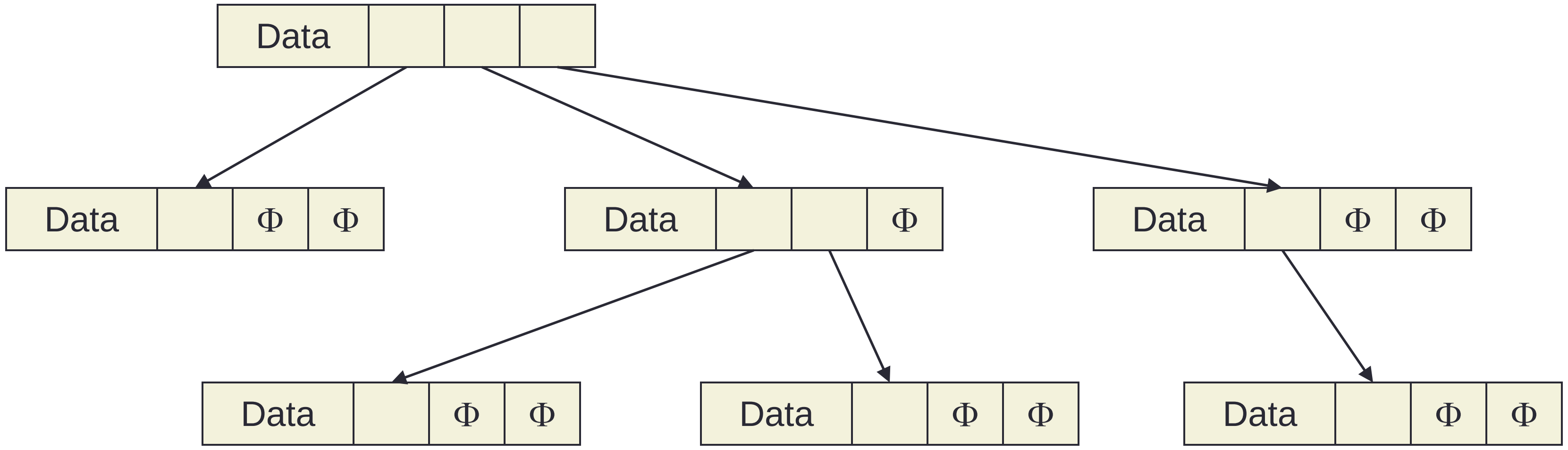
Terminology

- Height of node: number of edges between node and farthest leaf (Height(B)=2)
- Height of tree: height of root node (height of tree is 3)
- Depth of node: number of edges between root and node (depth of K is 3)



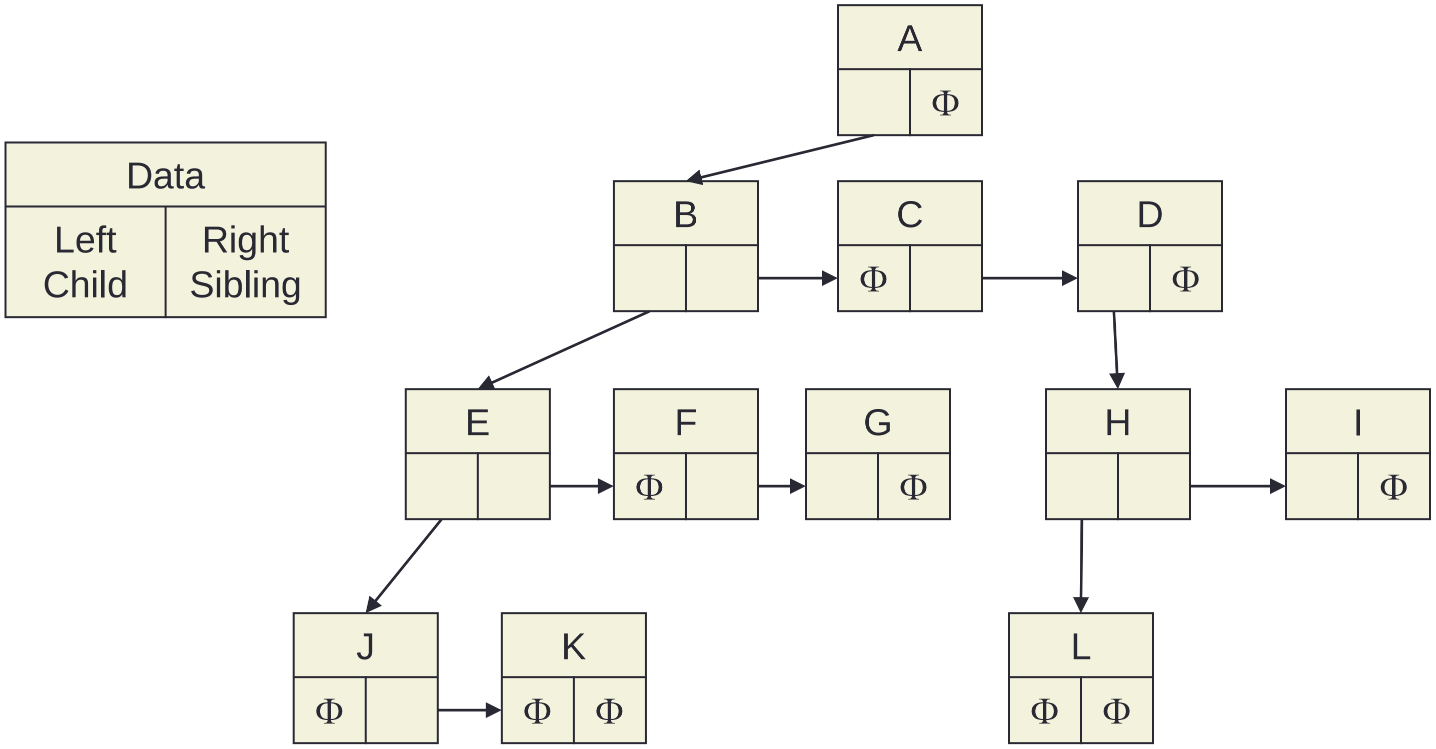
Tree Representation

- Every node contains:
  - Key/data
  - Children nodes
  - Parent node(optional)



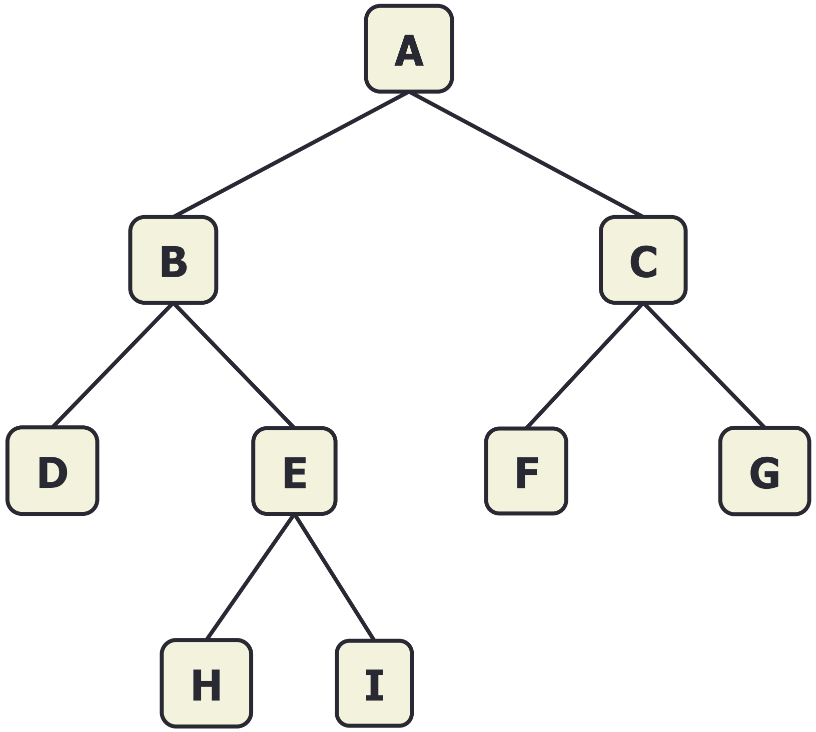
Left Child, Right Sibling Representation

- Every node contains
  - Key/data
  - Pointer to left child and right sibling



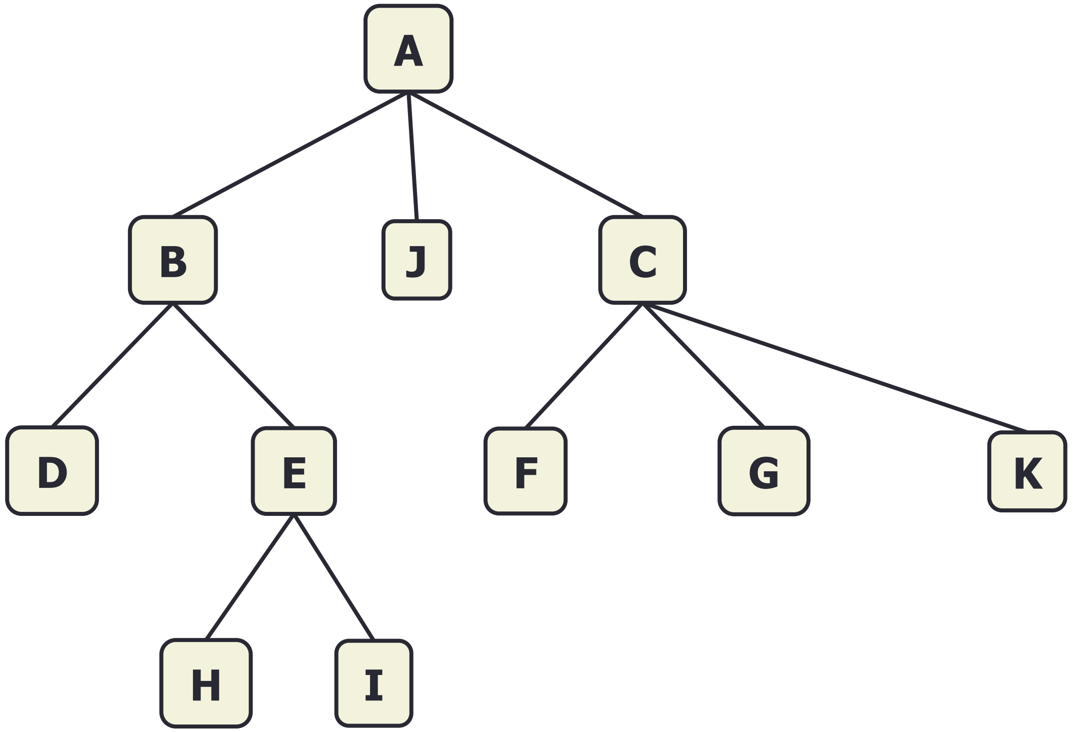
Binary Tree

- Each node has at most two children
- The children of a node are ordered pair (a left and a right child)
- Each node contains:
    - Key
    - Left
    - Right
    - Parent(optional)



K-ary Tree

- Each node has at most **K** children
- Binary tree is a special case with K=2
- Eg. 3-ary tree



Breadth First Traversal

- Traverse all the nodes at level ***i*** before progressing to level ***i+1*** starting from root node
- Add nodes in the queue as soon as their parent is visited.
- In each iteration, delete one element from queue and mark visited

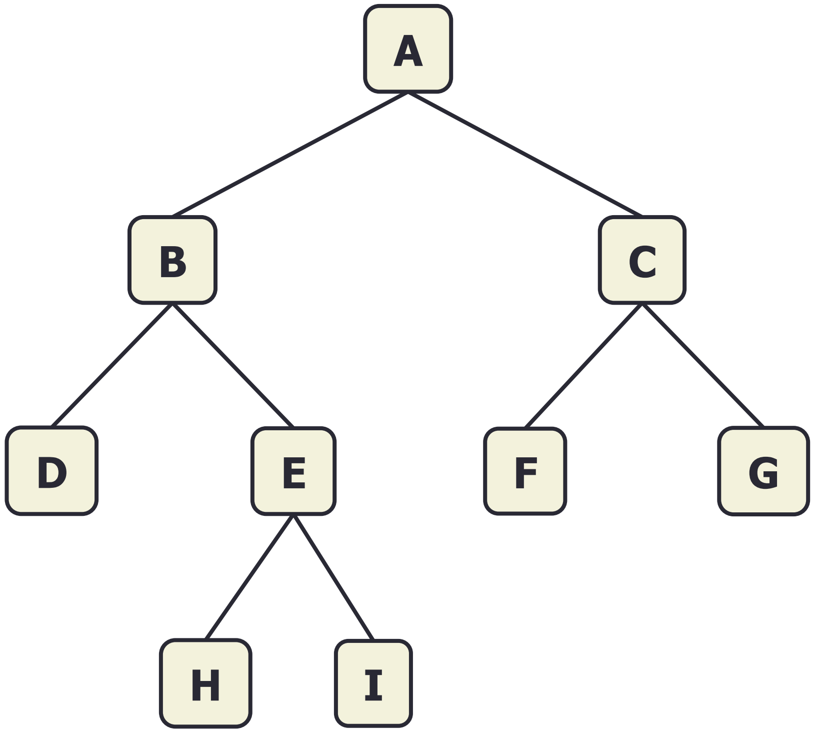
BFS Algorithm

```
BFS(Tree) {
        if (!isEmpty(Tree)) enqueue(Q, root);
        while (!isEmpty(Q)) {
                node = dequeue(Q);
                print(node->data);
                if (node->left != NULL) enqueue(Q,node->left);
                if (node->right != NULL) enqueue(Q,node->right);
        }
}
```

BFS Example

Output:

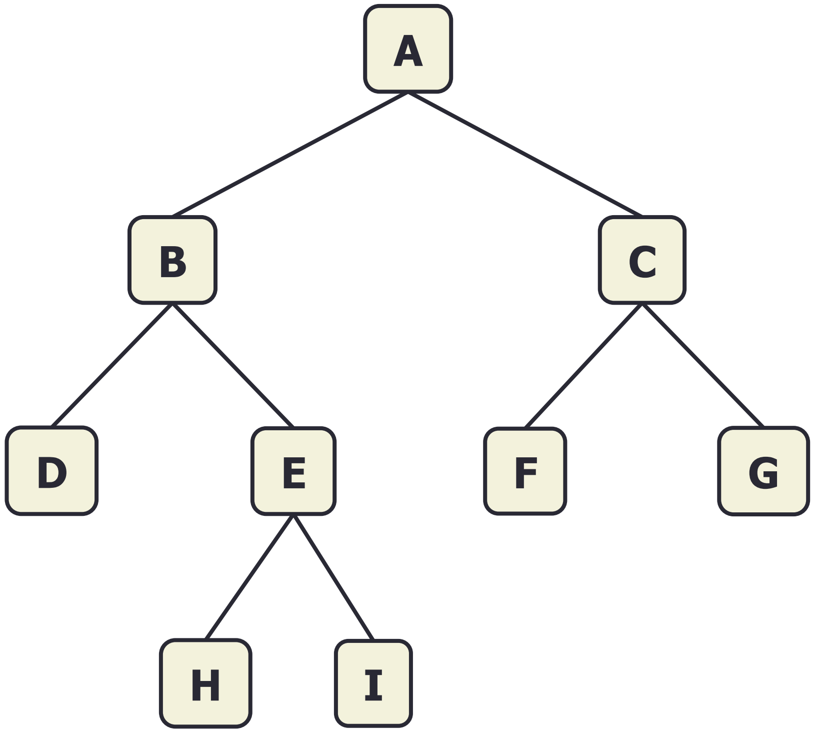
Queue(Q): A



BFS Example

Output: A

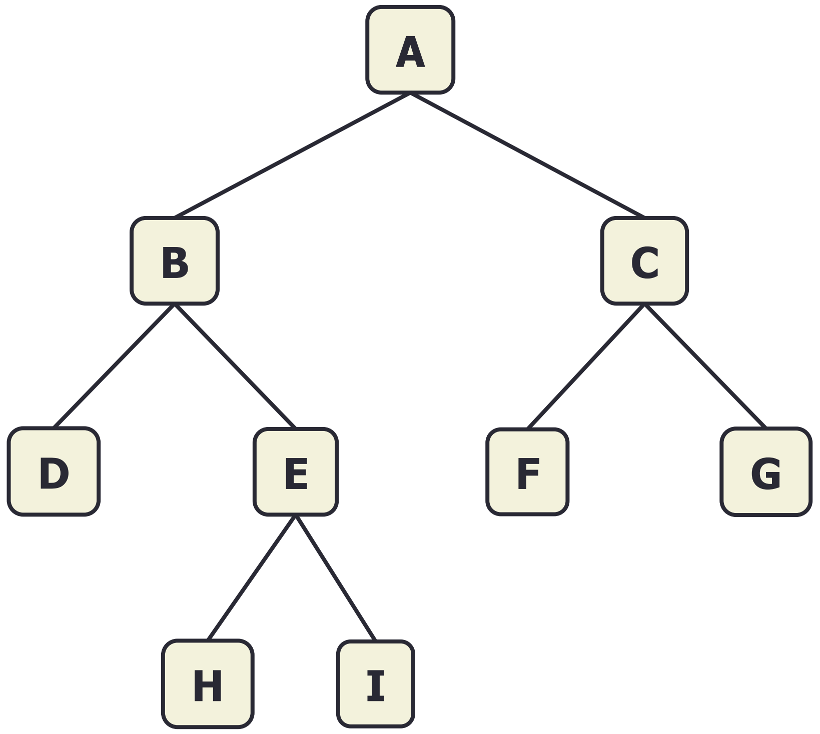
Queue(Q): B, C



BFS Example

Output: A B

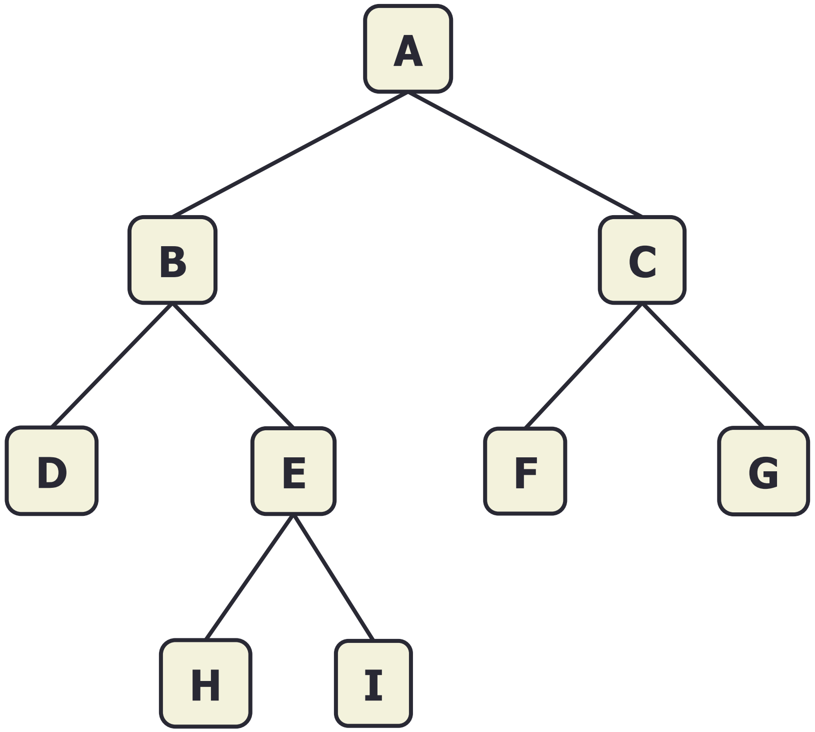
Queue(Q): C, D, E



BFS Example

Output: A B C

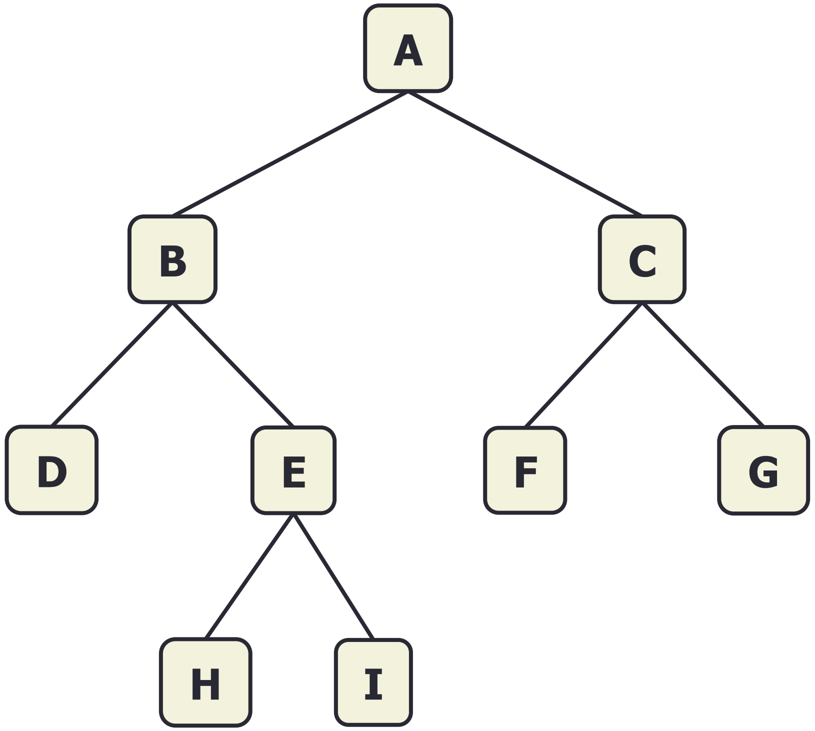
Queue(Q): D, E, F, G



BFS Example

Output: A B C D

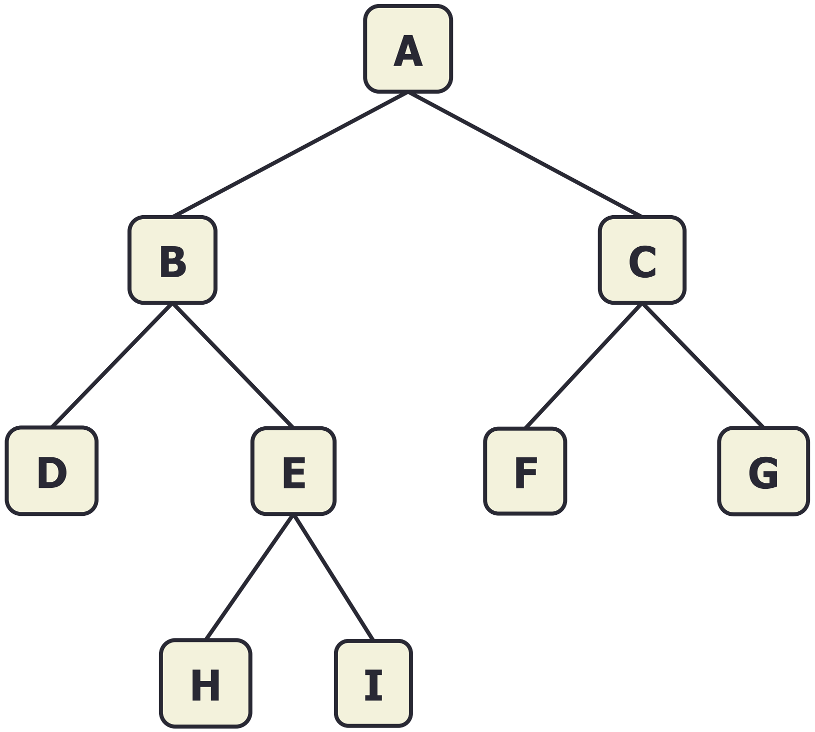
Queue(Q): E, F, G



BFS Example

Output: A B C D E

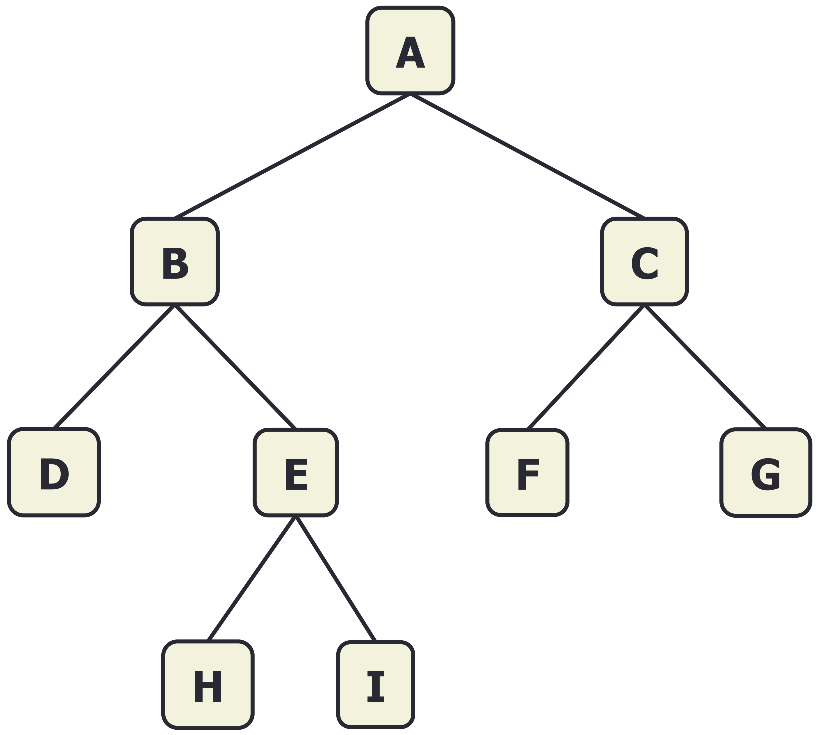
Queue(Q): F, G, H, I



BFS Example

Output: A B C D E F

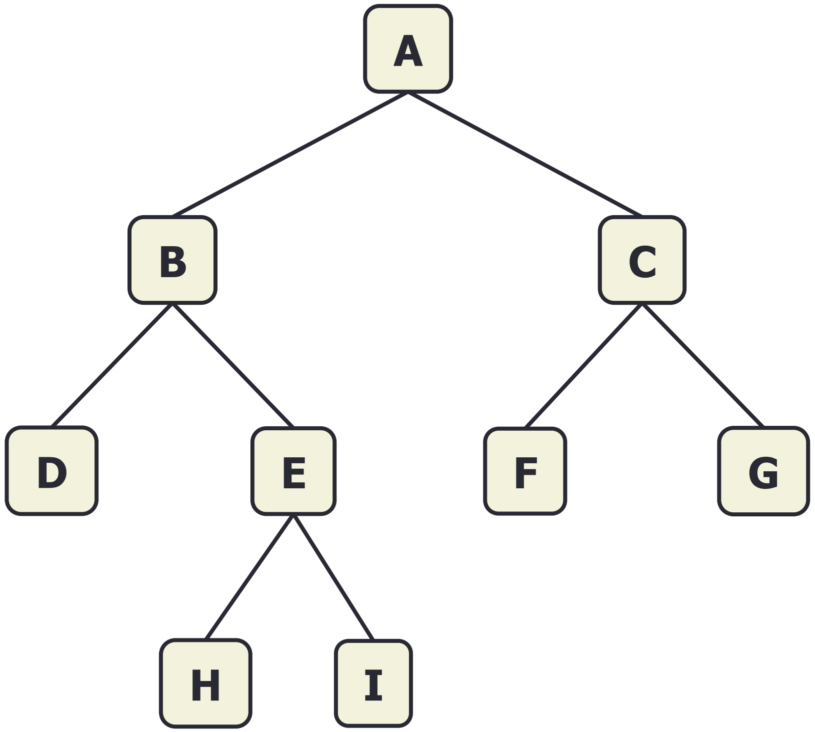
Queue(Q): G, H, I



BFS Example

Output: A B C D E F G

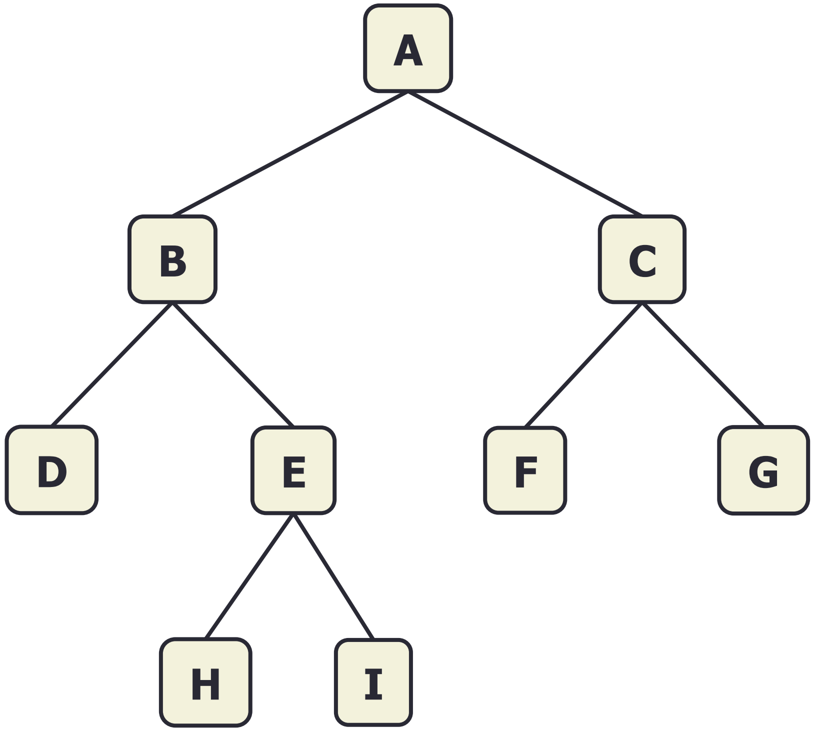
Queue(Q): H, I



BFS Example

Output: A B C D E F G H

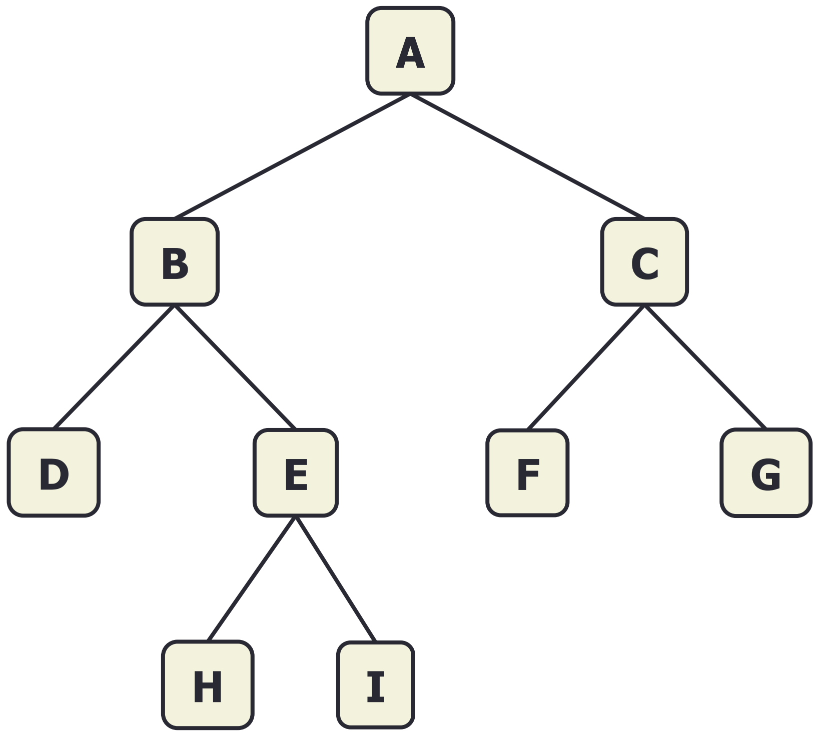
Queue(Q): I



BFS Example

Output: A B C D E F G H I

Queue(Q):



Depth First Search

- Travel All the nodes of one sub-tree of binary search before travelling other sub-tree
- DFS is recursively implemented on a tree to visit nodes
- **Note: Many of the tree algorithms are recursively implemented for the reason that tree itself is implemented recursively**
- DFS on binary tree can be implemented in 3 ways
  - PreOrder: Root-Left-Right
  - InOrder: Left-Root-Right
  - PostOrder: Left-Right-Root

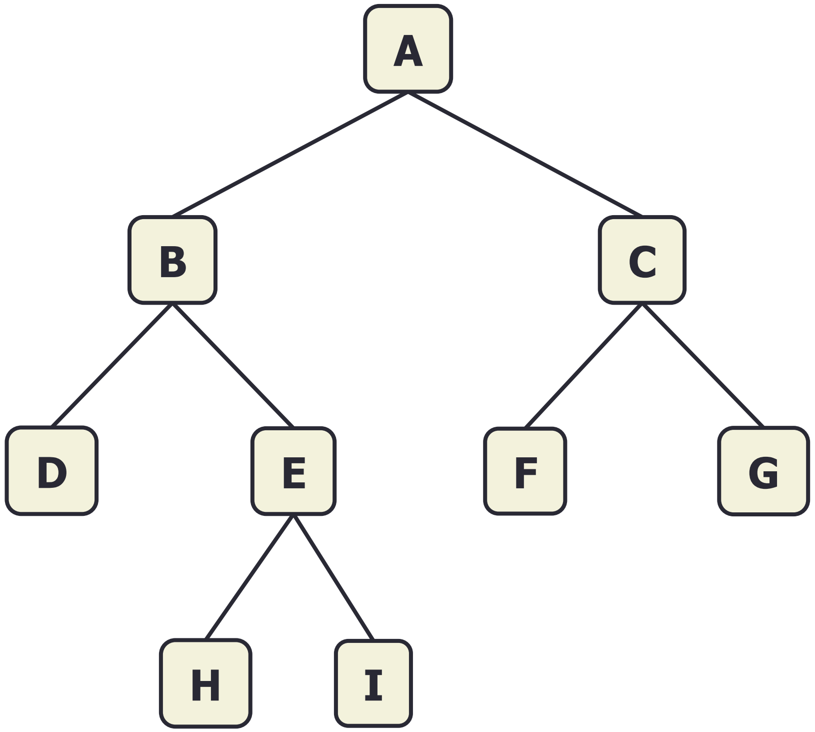
PreOrder Traversal

- Visit the root node first
- Visit left sub-tree in PreOrder
- Visit right sub-tree in PreOrder

```
PreOrderTraversal(Tree) {
        if (isEmpty(Tree)) return;
        else {
                print (tree->data);
                PreOrderTraversal(tree->left);
                PreOrderTraversal(tree->right);
        }
}
```

PreOrder Traversal: Example

Output: A B D E H I C F G



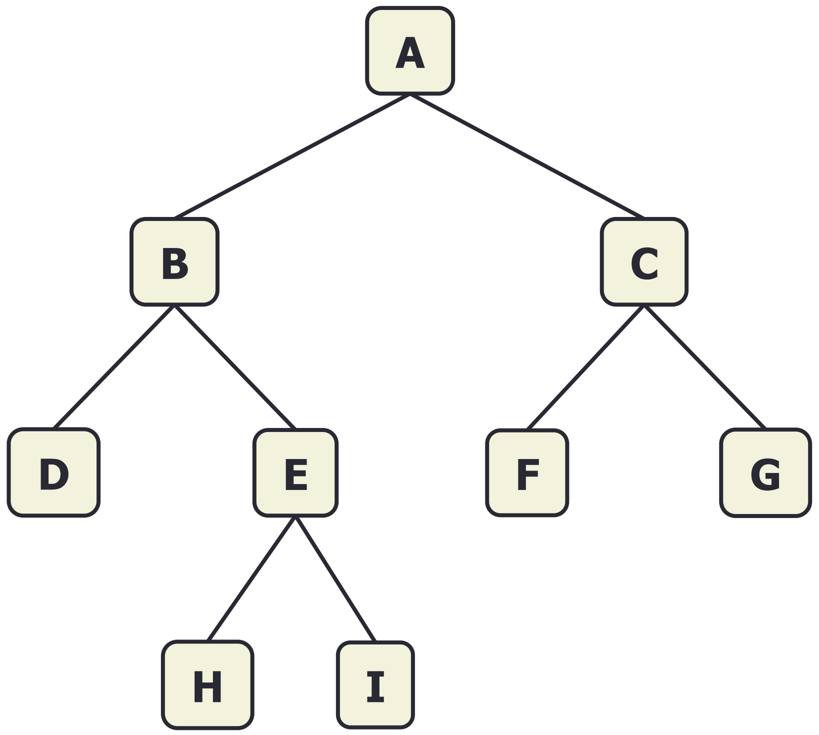
InOrder Traversal

- Visit the left sub-tree in InOrder
- Visit root node
- Visit right sub-tree in InOrder

```
InOrderTraversal(Tree) {
        if (isEmpty(Tree)) return;
        else {
                InOrderTraversal(tree->left);
                print (tree->data);
                InOrderTraversal(tree->right);
        }
}
```

InOrder Traversal: Example

Output: D B H E I A F C G



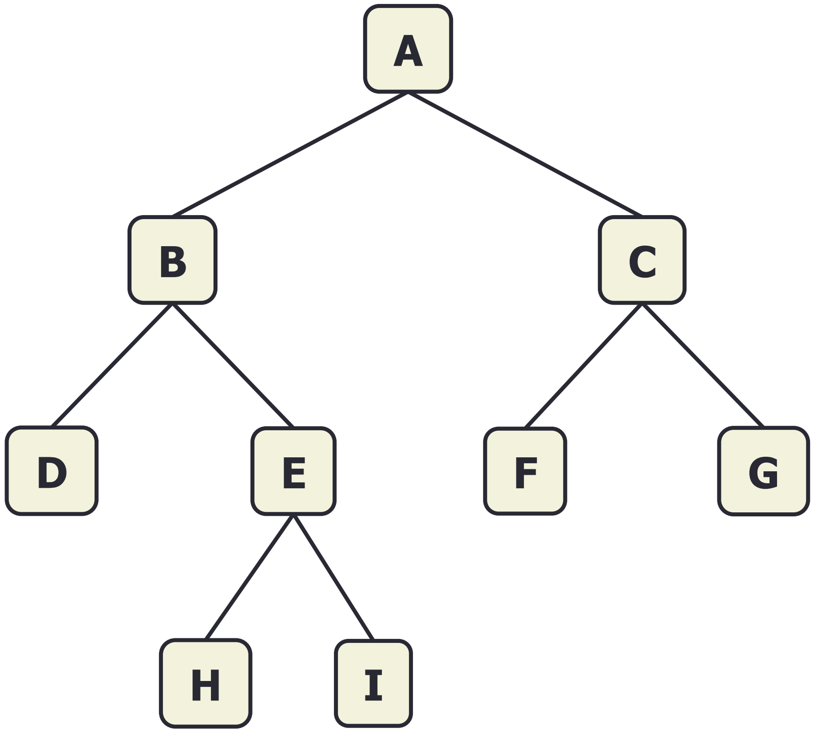
PostOrder Traversal

- Visit the left sub-tree in PostOrder
- Visit right sub-tree in PostOrder
- Visit root node

```
PostOrderTraversal(Tree) {
        if (isEmpty(Tree)) return;
        else {
                PostOrderTraversal(tree->left);
                PostOrderTraversal(tree->right);
                print (tree->data);
        }
}
```

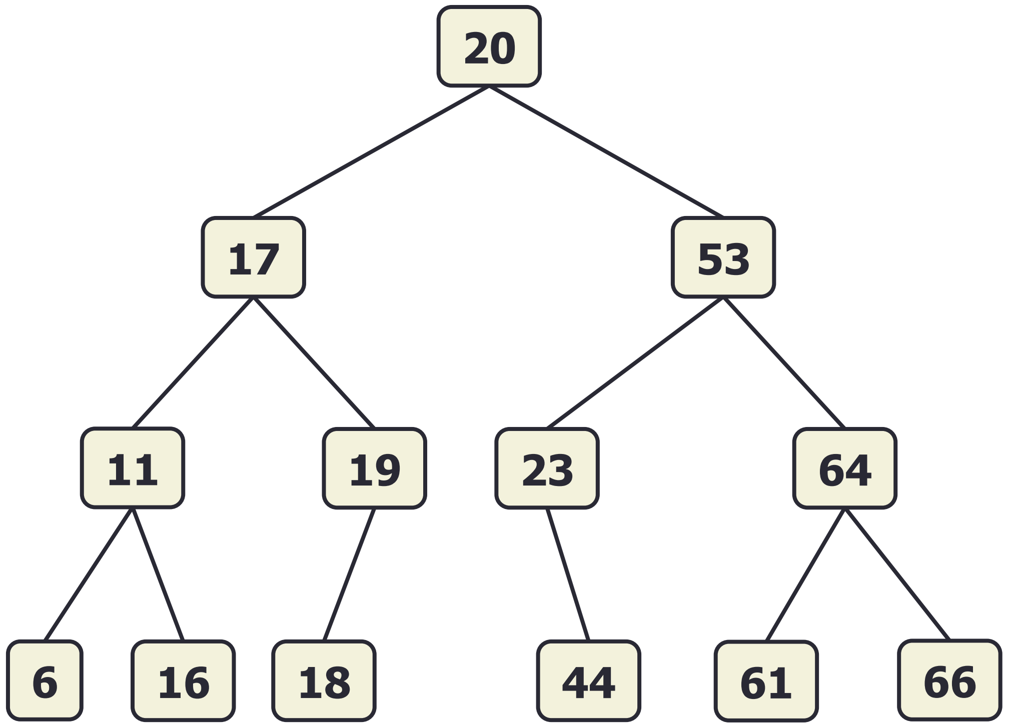
PostOrder Traversal: Example

Output: D H I E B F G C A



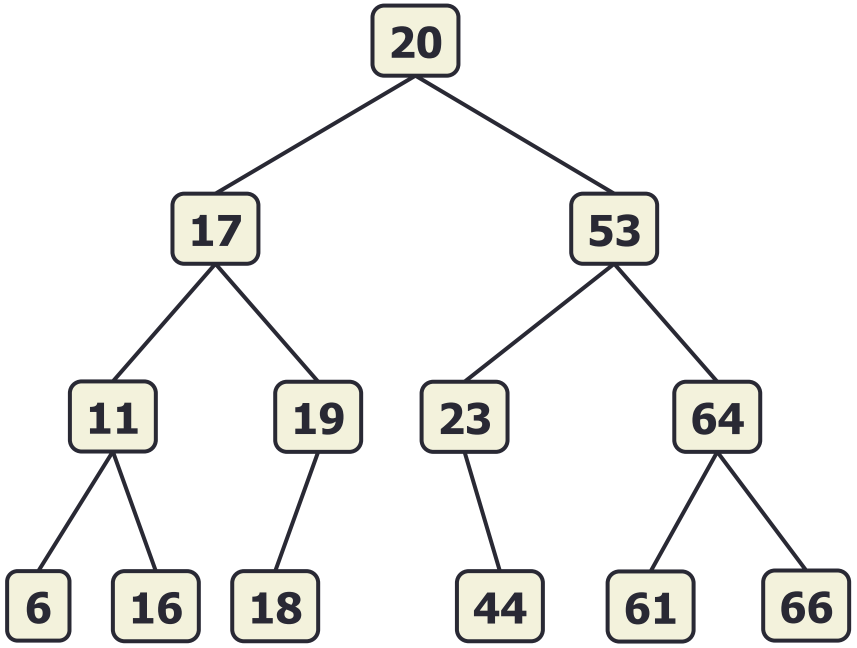
Exercise

- What will be the order of BFS, PreOrder, InOrder and PostOrder traversals on below tree?



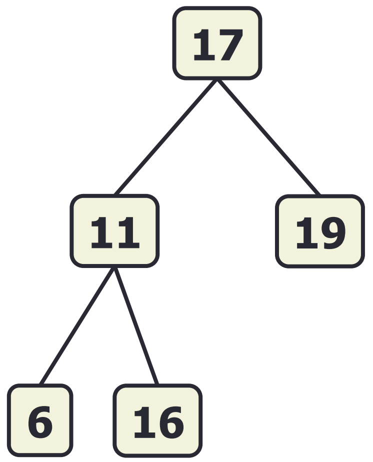
Exercise: Solution

BFS:	20 17 53 11 19 23 64 6 16 18 44 61 66
PreOrder:	20 17 11 6 16 19 18 53 23 44 64 61 66
InOrder:	6 11 16 17 18 19 20 23 44 53 61 64 66
PostOrder:	6 16 11 18 19 17 44 23 61 66 64 53 20

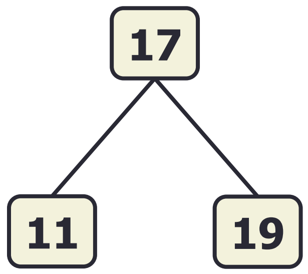


Full Binary Tree

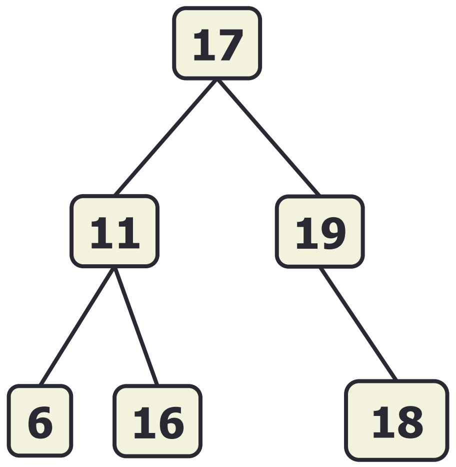
- Either a node has 2 children or no child in a binary tree



Full Binary Tree



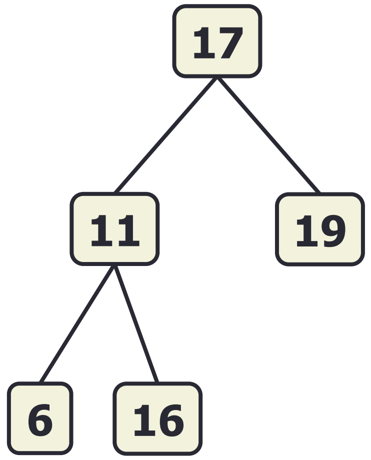
Full Binary Tree



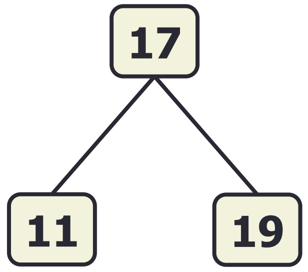
Not Full Binary Tree

Perfect Binary Tree

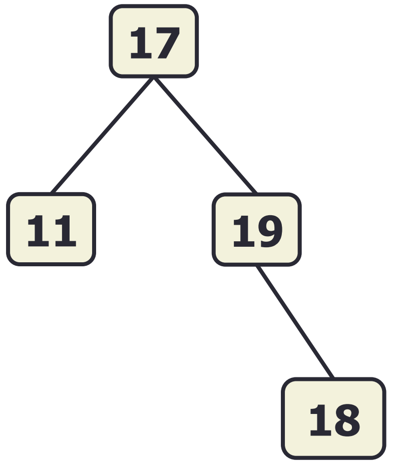
- All internal nodes has two children and all leaves are at same level/depth



full; not perfect



Full; perfect



Not full; not perfect

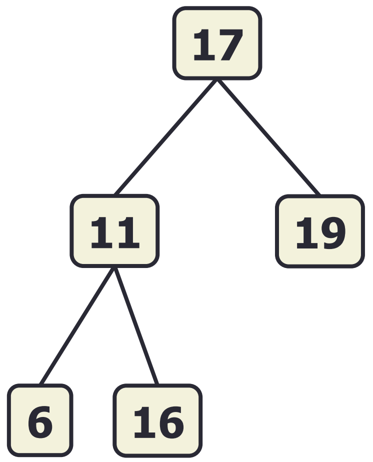
Perfect Binary Tree

- Level i has 2^i nodes
- If leaves are level h
  - Number of leaves is 2h
  - Number of internal node = $1+2+2^2+2^3+...+ 2^{h-1} = 2^h – 1$
    = number of leaves - 1
  - Total number of nodes = $2^h + 2^h – 1 = 2^{h+1} – 1$

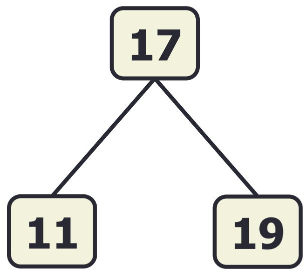
- If total number of nodes is n
  - Number of leaves = (n+1)/2
  - Height of tree = $\log_2$ (number of leaves) = $\log_2(n+1)/2$

Complete Binary Tree

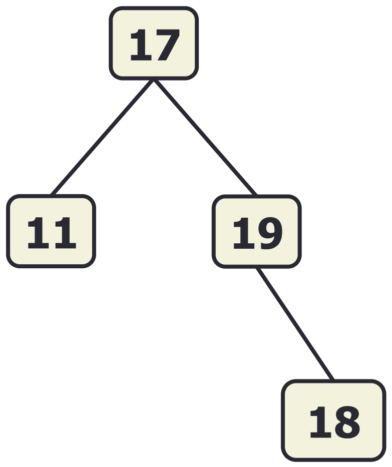
- All levels are completely filled except last. Also, all nodes in last level are as far left as possible



Full;
Not perfect;
Complete



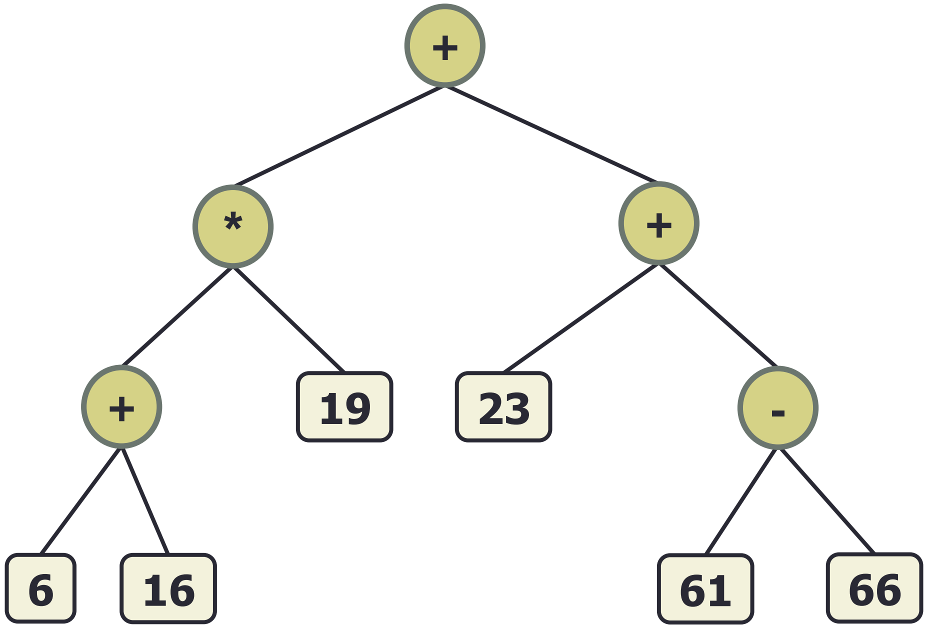
Full;
Perfect;
Complete



Not full;
Not perfect;
Not Complete

Example : Expression Tree

- An arithmetic expression can be represented as binary tree where internal nodes are operators and leaves are operands
- Eg. ((6+16) * 19) + (23 + (61-66))

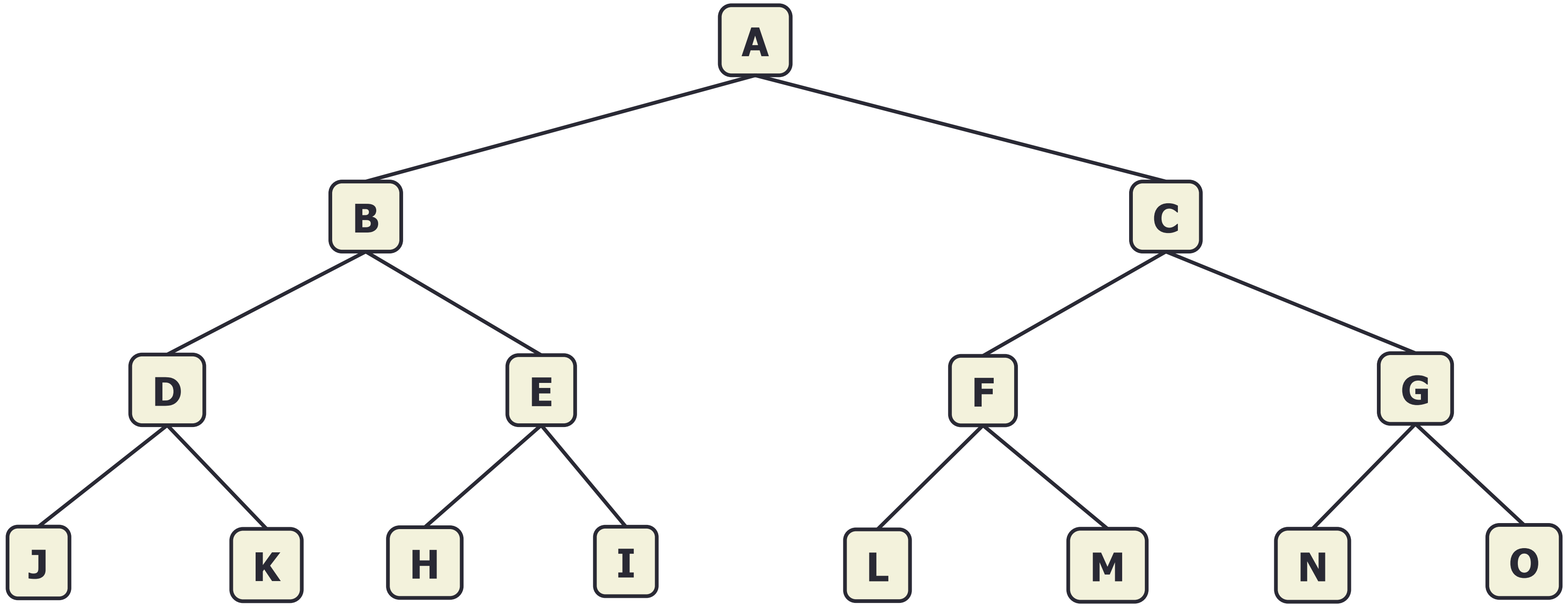


Complete Binary Tree

- Perfect tree is a special case of complete tree with last level completely filled.
- In some literature, Perfect binary tree is referred as Complete binary tree. In that case, Complete binary tree is referred Almost Complete binary tree.

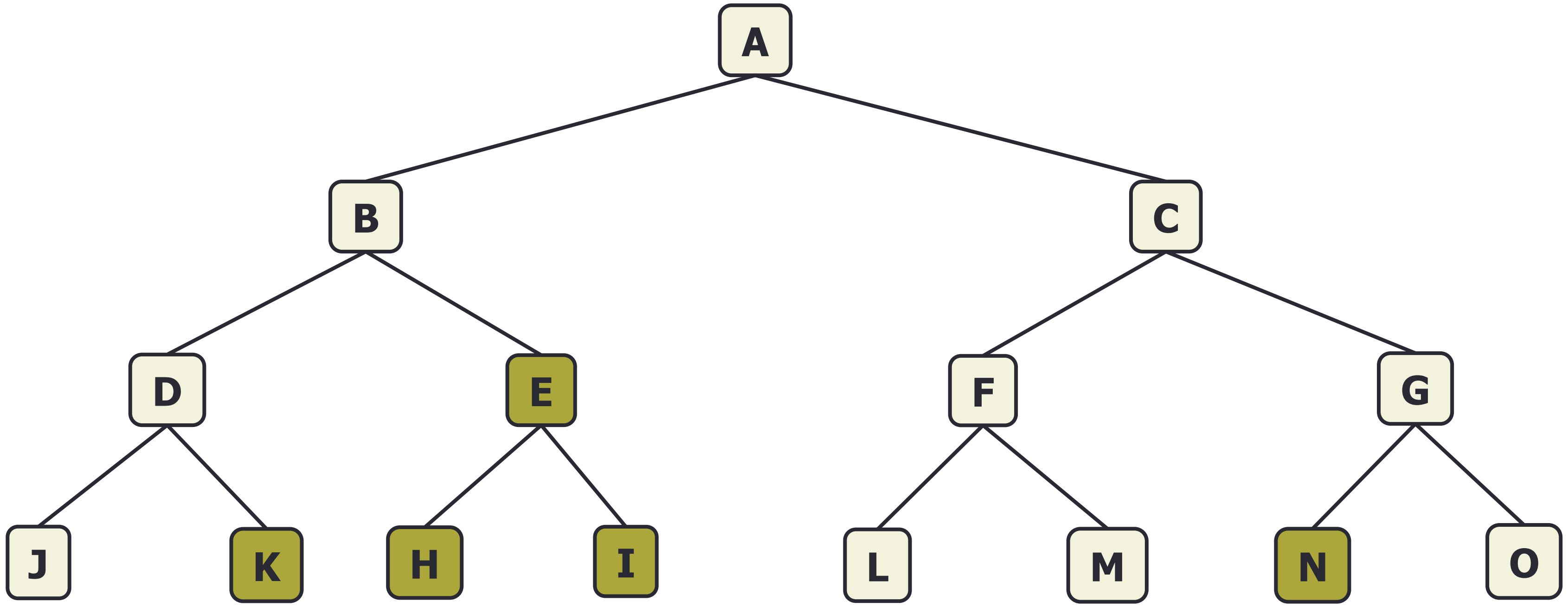
Binary Tree

- Any binary tree can be thought of as a tree obtained by pruning some nodes of a perfect binary tree



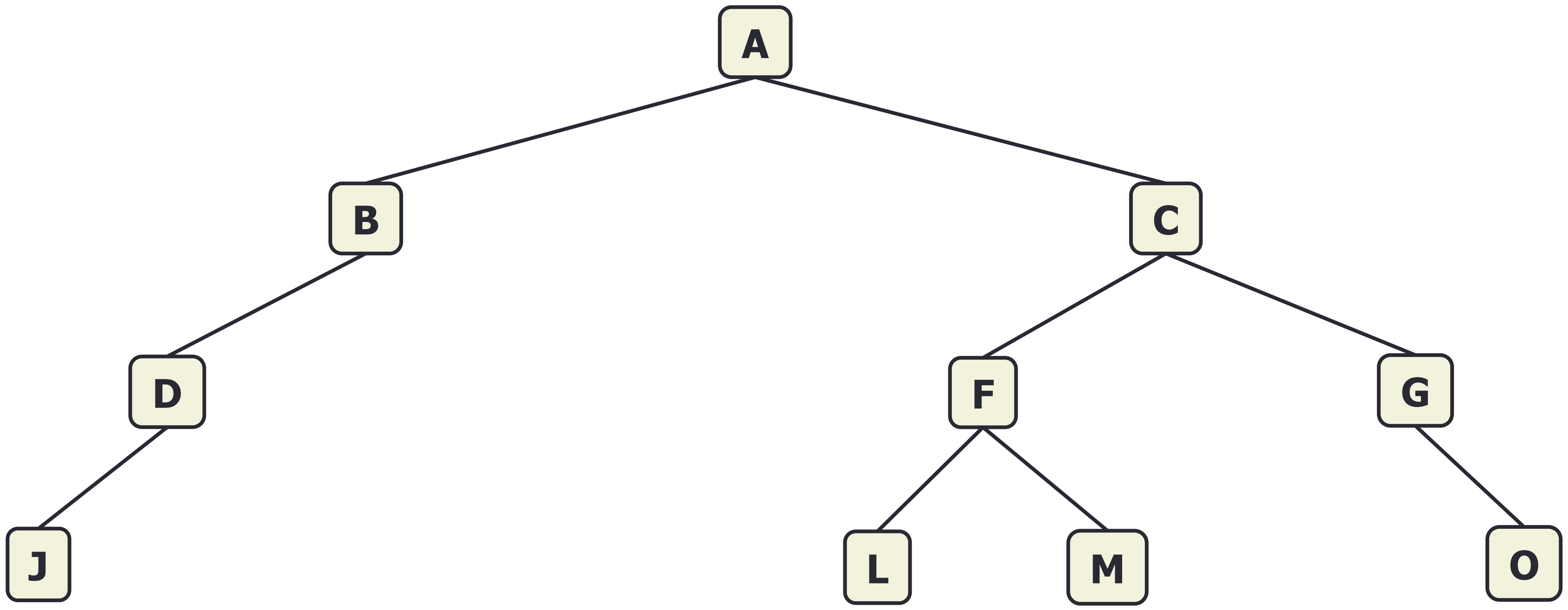
Binary Tree

- Any binary tree can be thought of as a tree obtained by pruning some nodes of a perfect binary tree



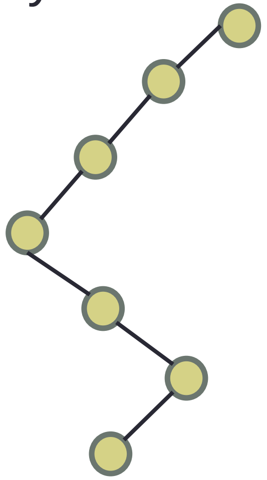
Binary Tree

- Any binary tree can be thought of as a tree obtained by pruning some nodes of a perfect binary tree



Height of a Binary Tree

- If a binary tree has n nodes and height h, then
  - Level i has at most 2^i nodes
  - $n <= 2^{h+1} – 1$
  - Hence, $h >= \log_2(n+1)/2$ i.e. minimum height of a tree with n nodes is $O(\log_2 n)$
  - Maximum height of a tree with n nodes is n-1 which is obtained when every non-leaf node has exact one child

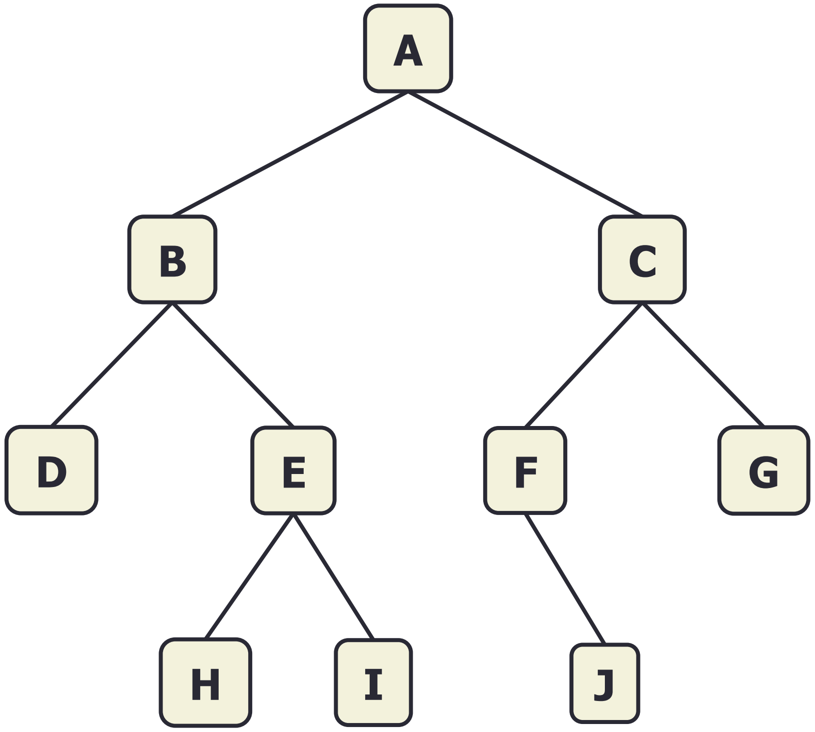


Linear Representation of Binary Tree

- Binary tree can also be represented using arrays
- Store root node at index ***0***
- Store left child of a parent node is at ***2*i+1*** where ***i*** is the index of parent node in array
- Store right child of a parent node is at ***2*i+2*** where ***i*** is the index of parent node in array
- Parent of a node at index ***i*** can be found at ***(i-1)/2*** except for root node

Example

A	B	C	D	E	F	G			H	I		J						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	…	26



Linear Representation of Binary Tree

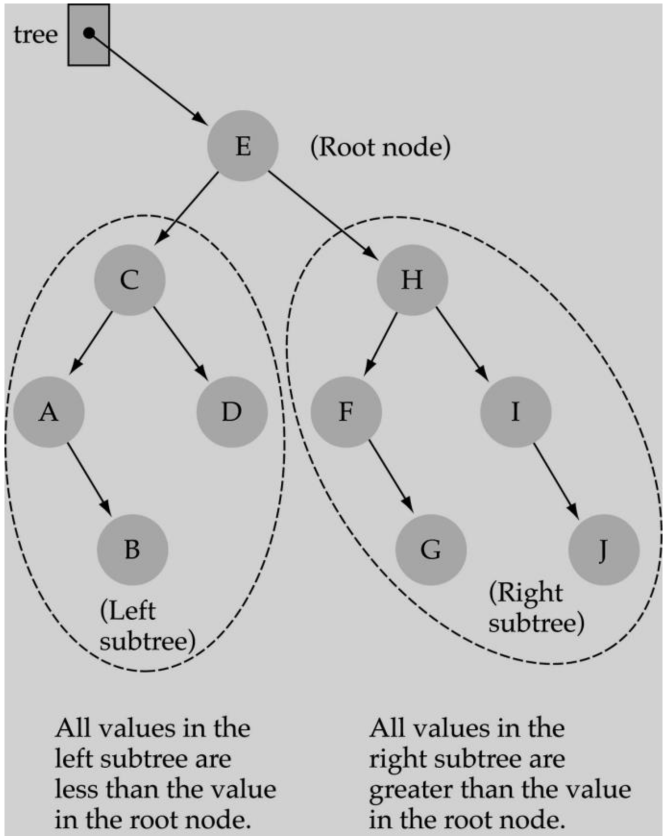
- If a node doesn’t have a left or/and right child, indices for left or/and right child are empty
- If index of a child is greater than size of array, child of that node does not exist

Searching a Binary Tree

- BFS or DFS
- Worst case time complexity = O(n) where n is the number of nodes in binary tree
- Can we improve searching?

Binary Search Tree

- Root is greater than left child and less than right child
- Equal keys can be kept in left or right sub-tree
- Same strategy should be followed for whole tree for repeated keys

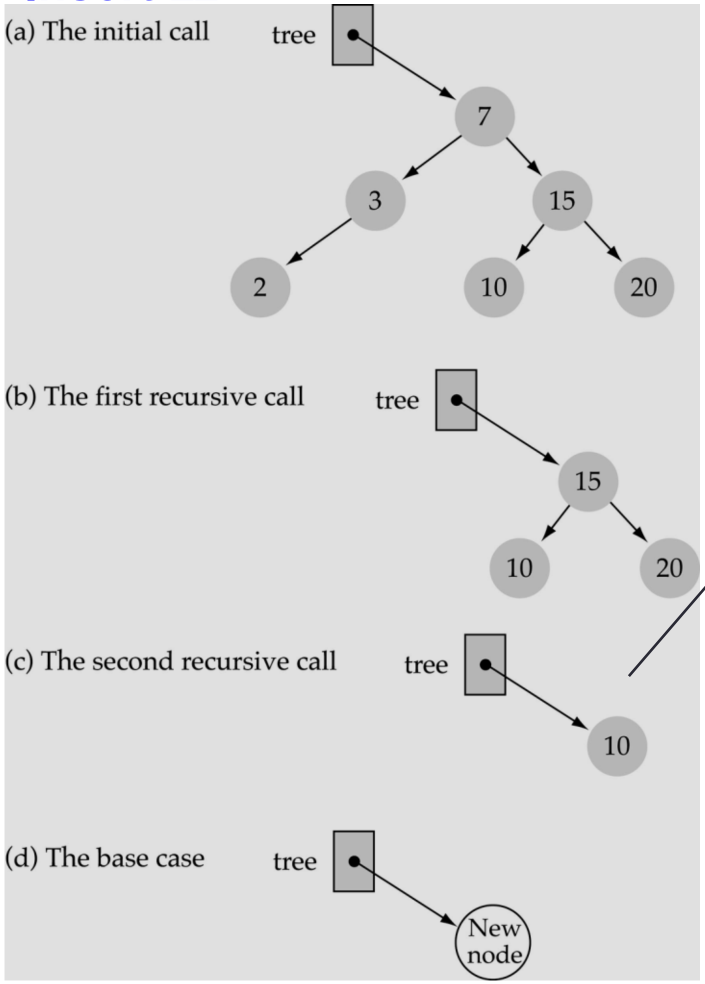

tree
E
(Root node)
C
H
A
D
F
I
B
G
J
(Left subtree)
(Right subtree)
All values in the left subtree are less than the value in the root node.
All values in the right subtree are greater than the value in the root node.

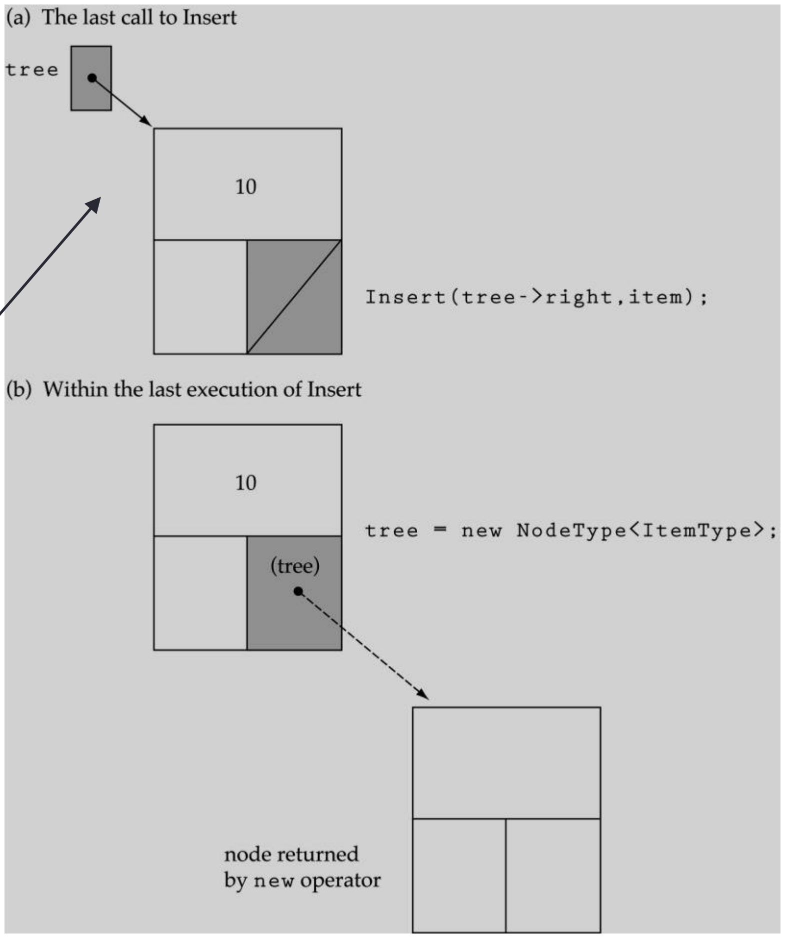
Insertion in BST

```
insertKey(tree, key) {
    if tree = NULL
        create new node (key, NULL, NULL);
    else tree->data < key
        tree->right = insertKey(tree->right, key);
    else
        tree->left = insertKey(tree->left, key);

}
```

Insertion in BST (cont.)

Insert 11




(a) The last call to Insert
tree
10
Insert(tree->right,item);
(b) Within the last execution of Insert
10
tree = new NodeType<ItemType>;
(tree)
node returned by new operator

Insertion in BST : Code

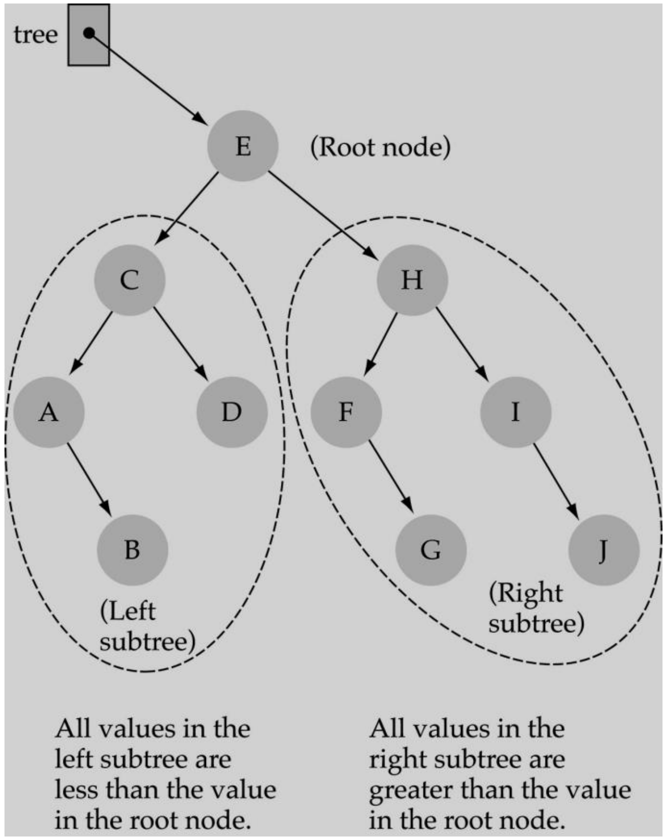
```
typedef struct Node {  //structure of a node
    int data;
    struct Node left;
    struct Node right;
} * Nptr;

Nptr newNode(int data, Nptr left, Nptr right) { //create a new node
    Nptr nNode = (Nptr) malloc(sizeof(struct Node));
    nNode->data = data;
    nNode->left = left;
    nNode->right =right;
    return nNode;
}
```

Insert in BST : Code (cont.)

```
Nptr insertKey(Nptr tree, int data) {
    if (tree == NULL) {
        return newNode(data, NULL, NULL);
    }
    else if (tree->data < data) {
        tree->right = insertKey(tree->right, data);
    }
    else {
        tree->left = insertKey(tree->left, data);
    }
    return tree;
}
```

Search in BST: Algorithm

```
searchBST(tree, key) {
    if tree = NULL {
        print("Not fount");
        return;
    }
    else if tree->data = key
        print("Key found");
    else if tree->data >key
        searchBST(tree->left, key);
    else
        searchBST(tree->right, key);
}
```


tree
E
(Root node)
C
H
A
D
F
I
B
G
J
(Left subtree)
(Right subtree)
All values in the left subtree are less than the value in the root node.
All values in the right subtree are greater than the value in the root node.

Search in BST : Code

```
Nptr searchBST(Nptr tree, int key) {  //Search key in tree
    if (tree == NULL) {
        printf("\nkey not found");  //print not found if key not present
        return NULL;
    }
    else if (tree->data == key) {
        printf("\n Key found");  //print found if key not present
        return tree;
    }
    else if (tree->data < key)
        return searchBST(tree->right, key);  //Search right sub-tree
    else
       return searchBST(tree->left, key);  //Search left sub-tree
}
```

Search in BST: Time Complexity

- Search in BST depends on the height of the tree
- In worst case,

  Height of a binary tree with n nodes = $O(n)$

  Thus, Time complexity of search = $O(n)$

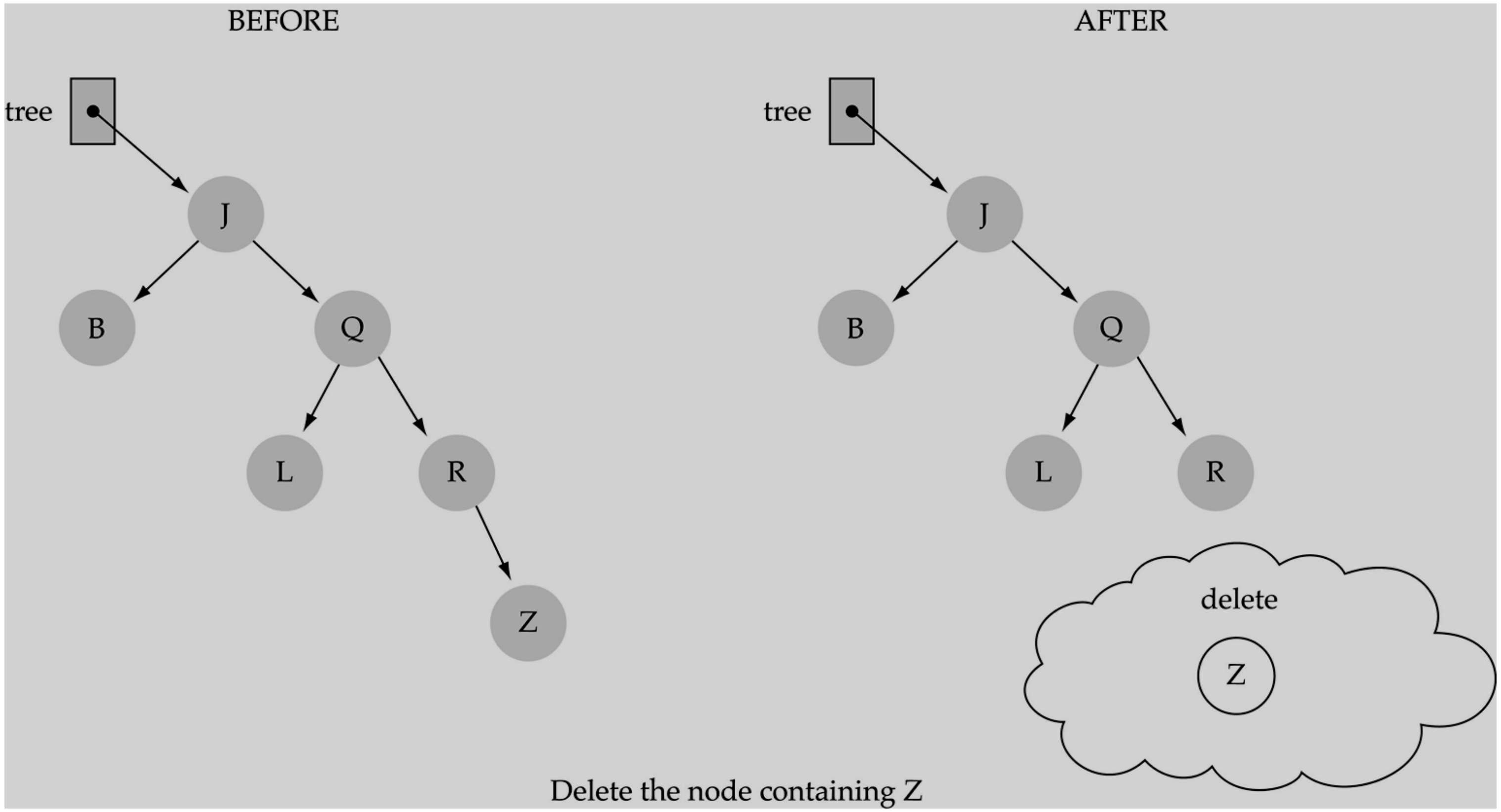
- In general,

  Height of a binary tree with n nodes = $O(\log_2 n)$

  Thus, time complexity of search = $O(\log_2 n)$

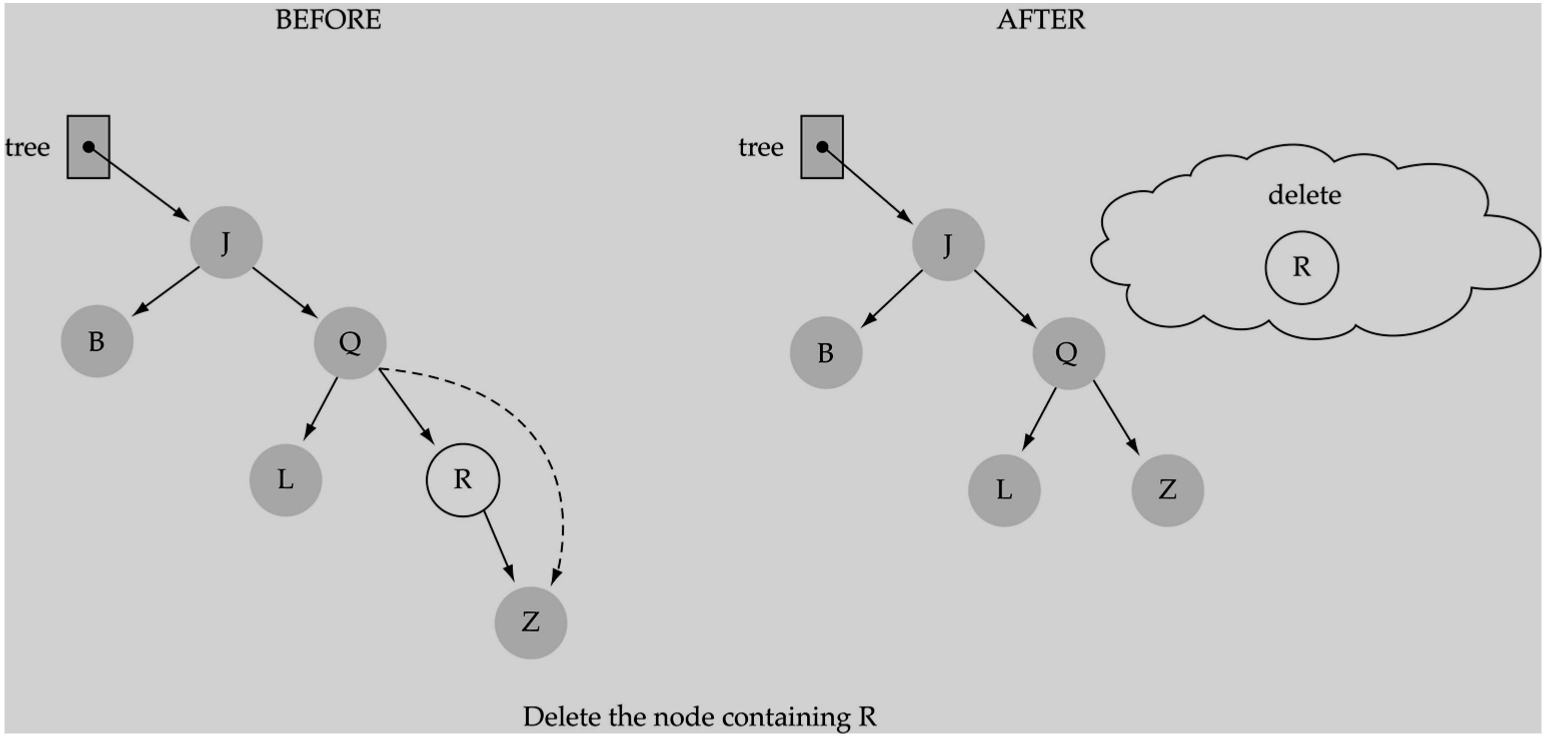
Deletion in BST: With no Children

- Simply delete the node



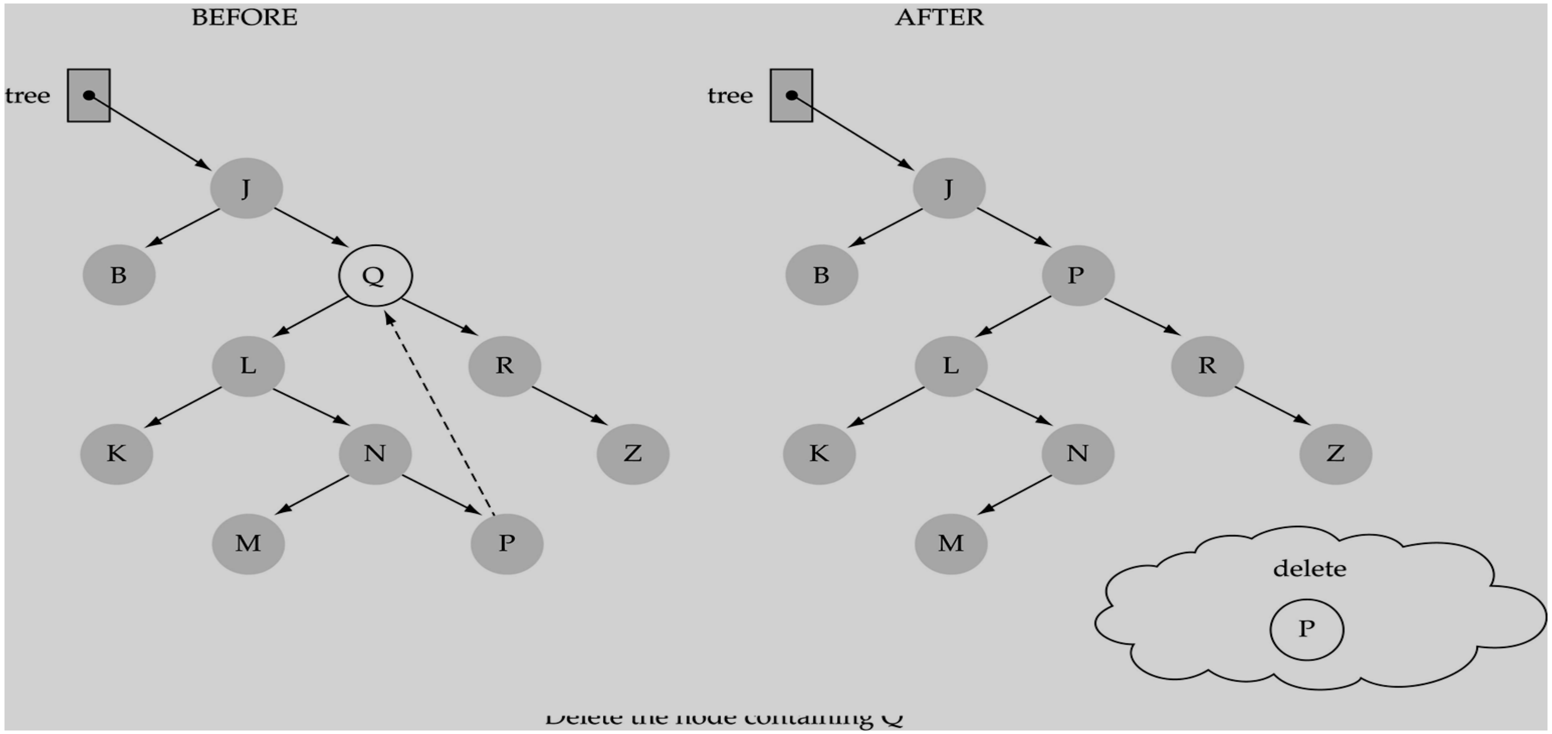
Deletion in BST: With One Child

- Make parent of node to be deleted to point its child node
- Set node to be deleted free (optional)



Deletion in BST : With Two Child

- Find predecessor (i.e. rightmost node in the left sub-tree) and replace data
- Delete predecessor node



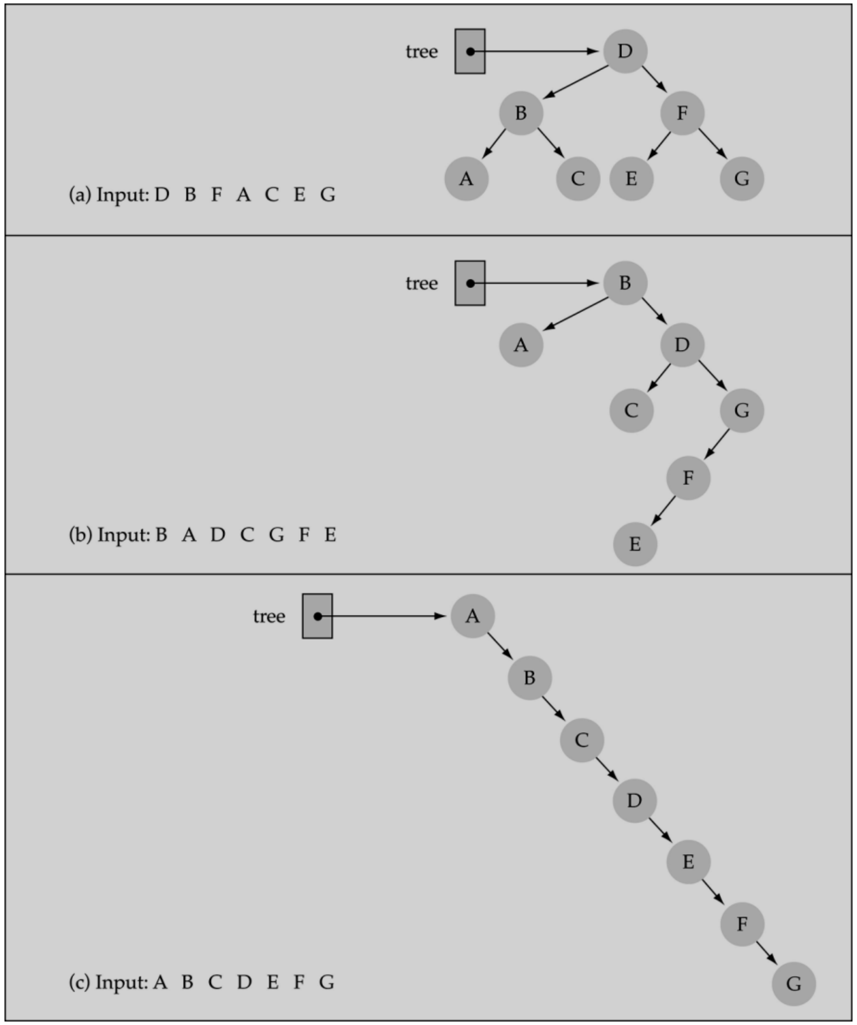
Delete the node containing Q

Order of Insertion in BST

- Does the order of inserting elements into tree matters?

Order of Insertion in BST

- Does the order of inserting elements into tree matters?
- Yes, certain orders might produce very unbalanced trees!
- Unbalanced trees are not desirable because search time increases!
- Solution are balanced BST like AVL tree.



Balanced BST

- Most of the applications (eg. Search, insert, delete, minimum, maximum etc.) are dependent on height of BST
- Order of insertion of element in BST could generate unbalanced BST
- Therefore, in worst case time complexity can be **O(n)**

- If we can ensure the height of BST is O(log n) always, we can guarantee other operations will also have time complexity of O(log n)
- Balanced BST have height O(log n) always
- Eg. AVL tree, Red-black tree, 2-3 tree etc

AVL Tree

- What is an AVL tree?
- How the insertion happens? Time complexity, code
- How the deletion happens? Time complexity, code

AVL Tree

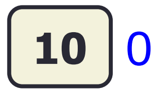
- A balanced BST named after scientist invented
- Maintains height close to minimum i.e. O(log n)
- After each insertion, check whether tree is balanced
- If not balanced, fix imbalance by rotation(s) to bring back to balance
- Balance factor of any node is difference between height of left and right sub-tree
- Tree is balanced if

-1 < balance factor of a node < 1

is true for every node in the AVL tree

AVL Tree : Examples

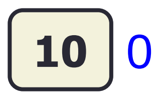
(i)



YES

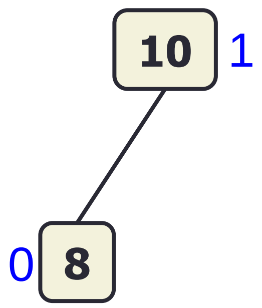
AVL Tree : Examples

(i)



YES

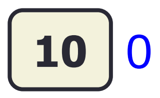
(ii)



YES

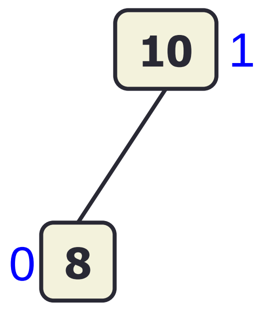
AVL Tree : Examples

(i)



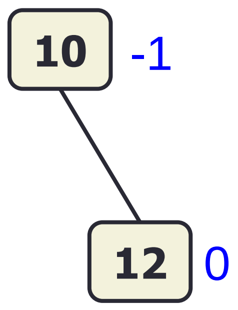
YES

(ii)



YES

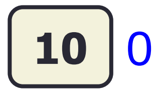
(iii)



YES

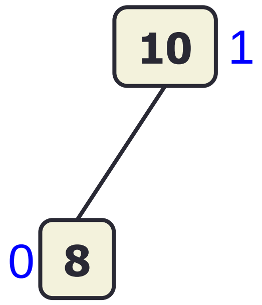
AVL Tree : Examples

(i)



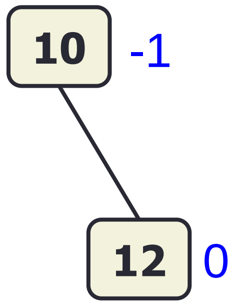
YES

(ii)



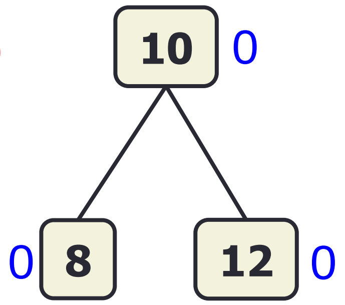
YES

(iii)



YES

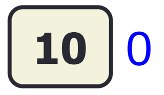
(iv)



YES

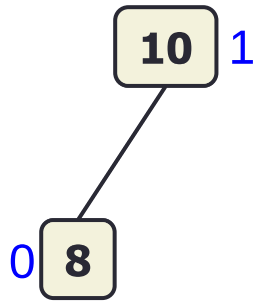
AVL Tree : Examples

(i)



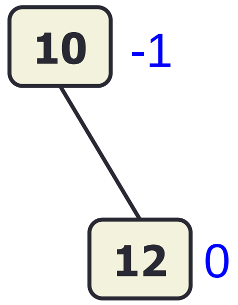
YES

(ii)



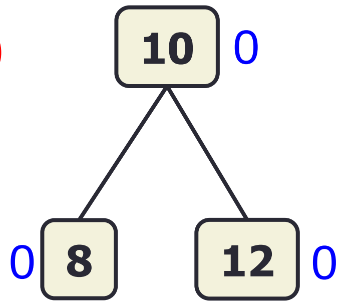
YES

(iii)



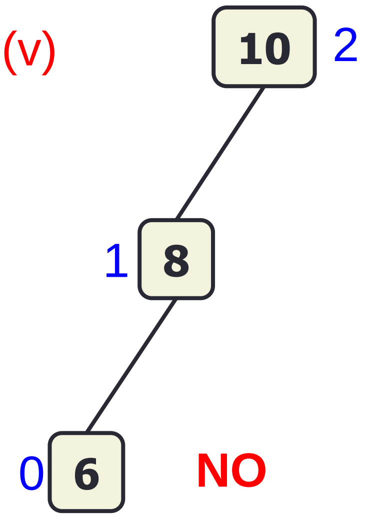
YES

(iv)



YES

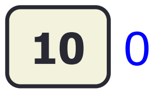
(v)



NO

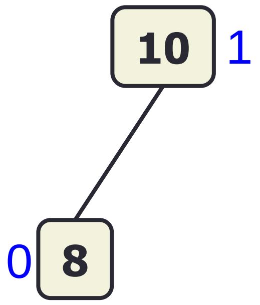
AVL Tree : Examples

(i)



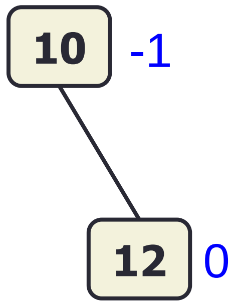
YES

(ii)



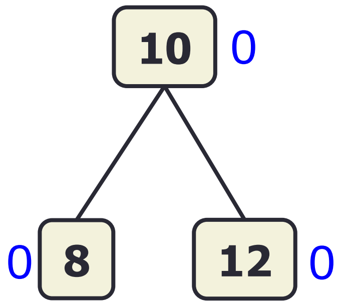
YES

(iii)



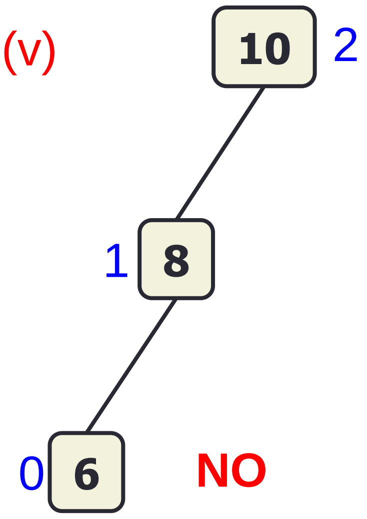
YES

(iv)



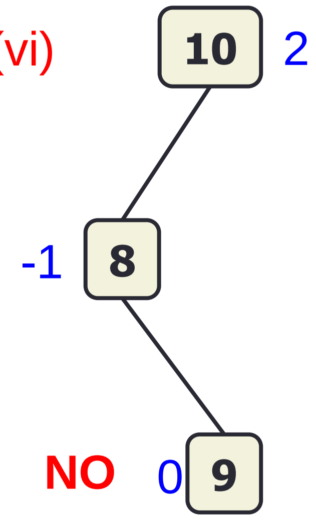
YES

(v)



NO

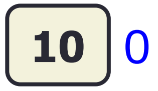
(vi)



NO

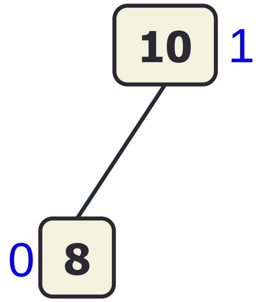
AVL Tree : Examples

(i)



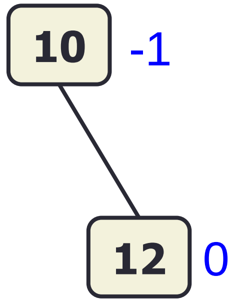
YES

(ii)



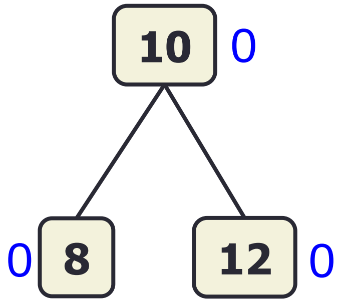
YES

(iii)



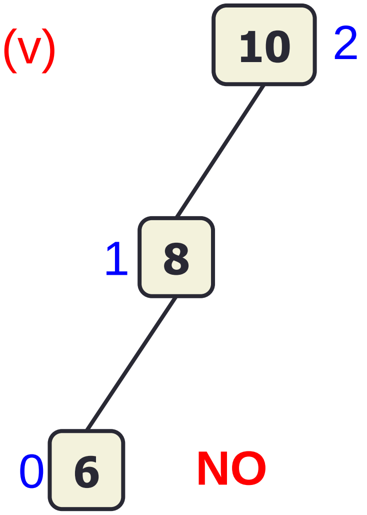
YES

(iv)



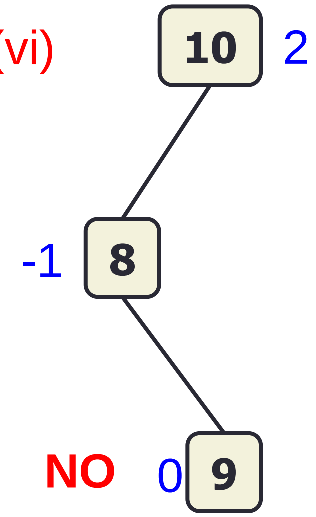
YES

(v)



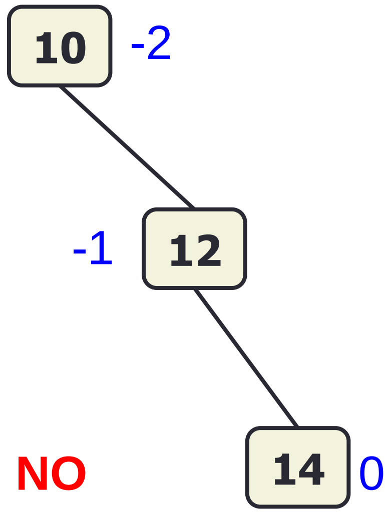
NO

(vi)



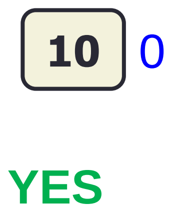
NO

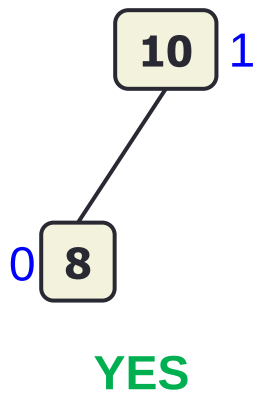
(viii)

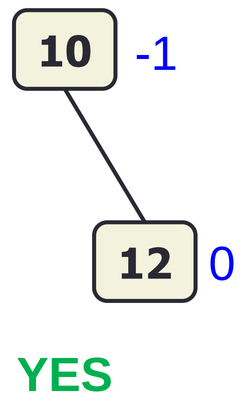


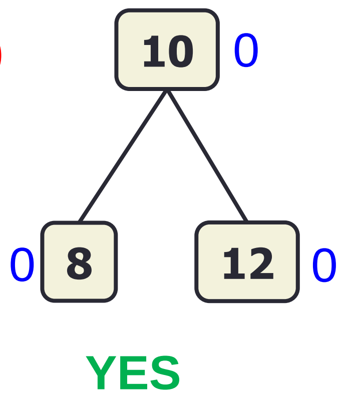
NO

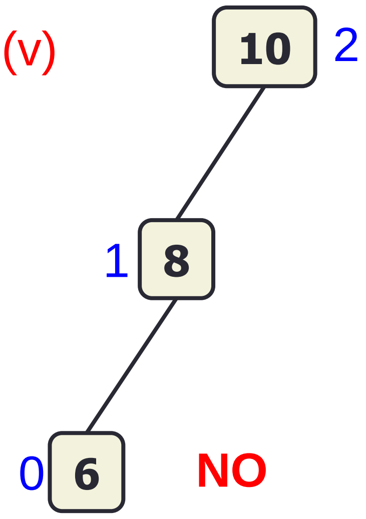
AVL Tree : Examples

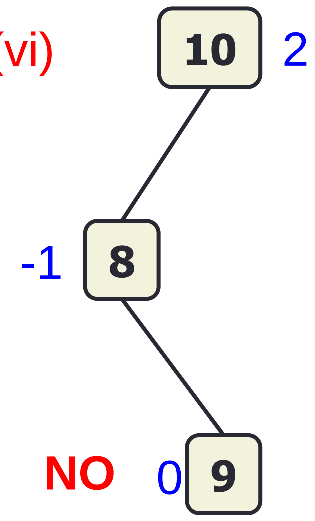
(i)


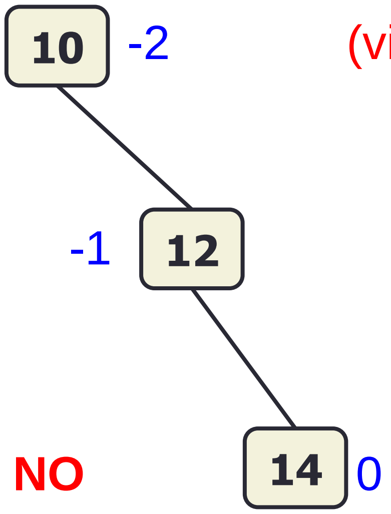
(ii)


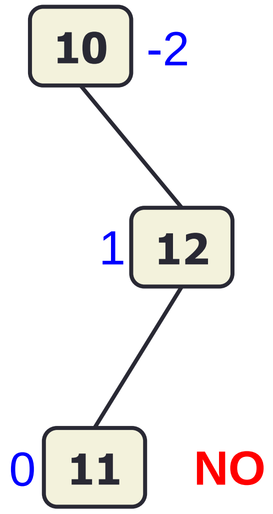
(iii)


(iv)


(v)


(vi)


(viii)


(vii)


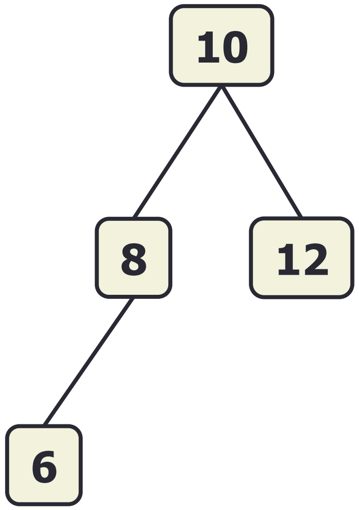
Insertion in AVL

Steps to insert in AVL:

1. Insert in BST (i.e. find proper location and create node)
2. Starting from new node inserted, check upwards for the first node which is imbalance
3. Re-balance tree by performing rotation(s) based on the type of node arrangement causing imbalance
    1. Left-Left case
    2. Right-Right case
    3. Left-Right case
    4. Right-Left case

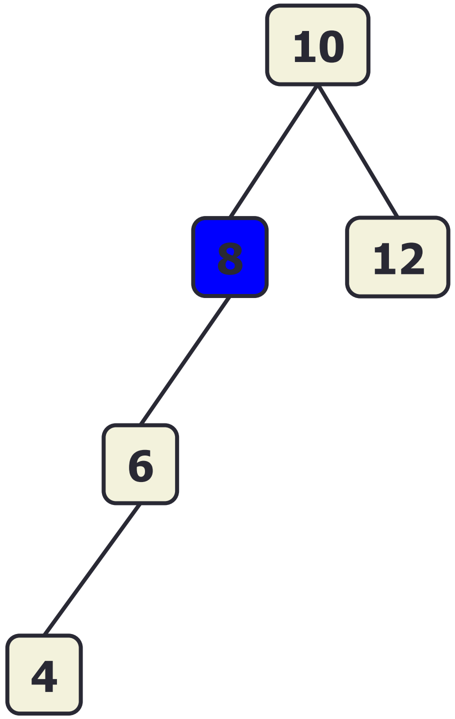
Left-Left Case

- Consider the AVL tree
- Insert **4** in the tree



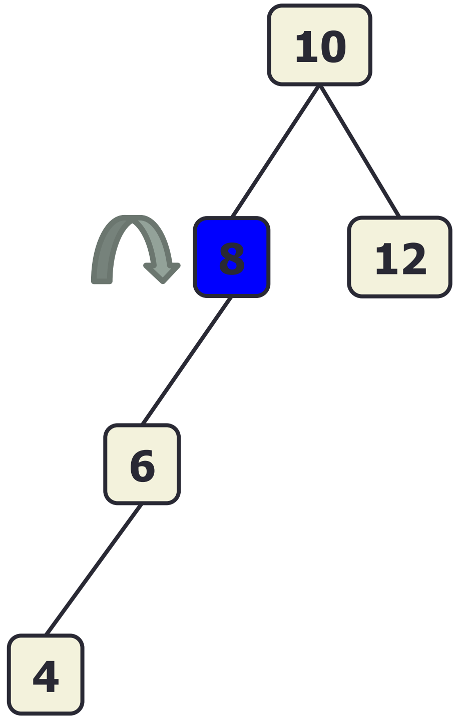
Left-Left Case

- Consider the AVL tree
- Insert **4** in the tree
- Travel upwards from new node to find first imbalance node i.e. **8**
- If new node is left-left to imbalance node, its left-left case



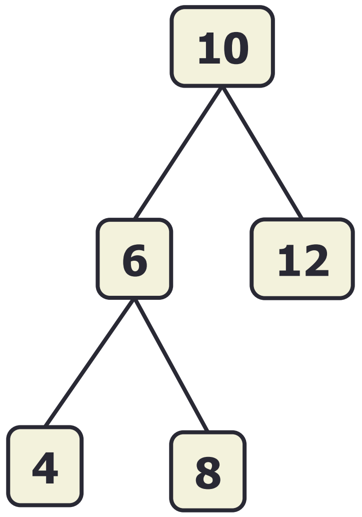
Left-Left Case

- Consider the AVL tree
- Insert **4** in the tree
- Travel upwards from new node to find first imbalance node i.e. **8**
- If new node is left-left to imbalance node, its left-left case
- Do a right rotate at **8** to balance tree



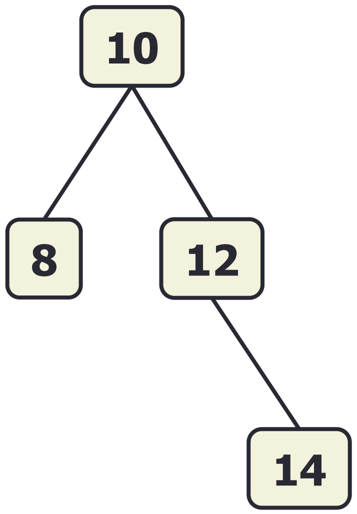
Left-Left Case

- Consider the AVL tree
- Insert **4** in the tree
- Travel upwards from new node to find first imbalance node i.e. **8**
- If new node is left-left to imbalance node, its left-left case
- Do a right rotate at **8** to balance tree
- Now the tree is again balanced



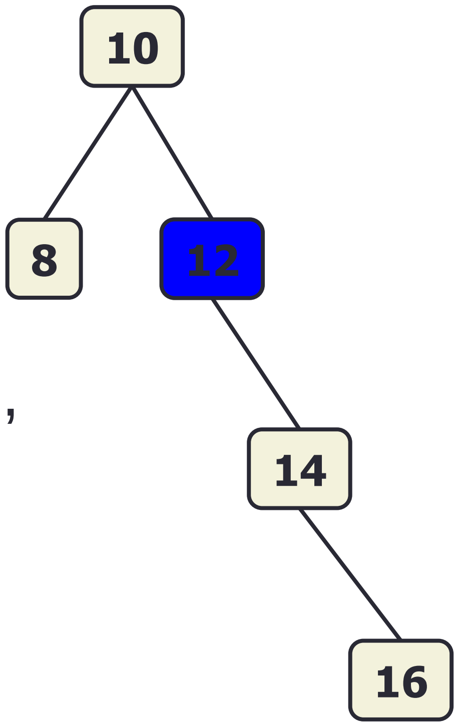
Right-Right Case

- Consider the AVL tree
- Insert **16** in the tree



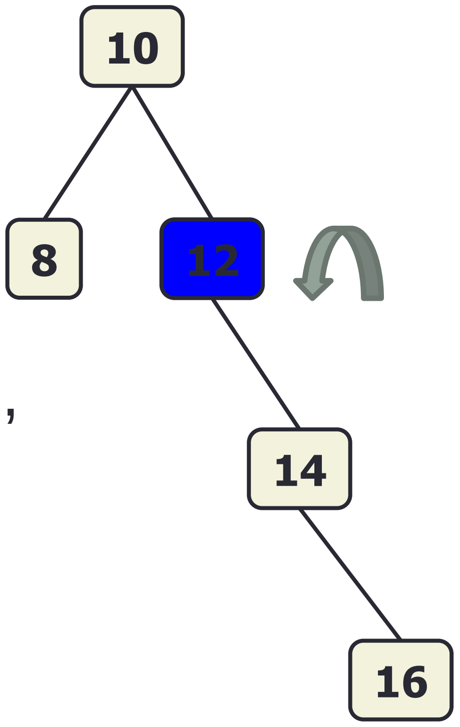
Right-Right Case

- Consider the AVL tree
- Insert **16** in the tree
- Travel upwards from new node to find first imbalance node i.e. **12**
- If new node is right-right to imbalance node, its right-right case



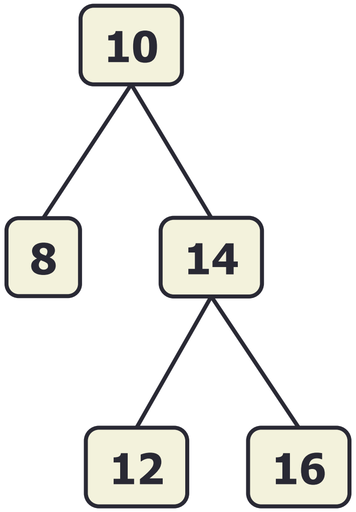
Right-Right Case

- Consider the AVL tree
- Insert **16** in the tree
- Travel upwards from new node to find first imbalance node i.e. **12**
- If new node is right-right to imbalance node, its right-right case
- Do a left rotate at **12** to balance tree



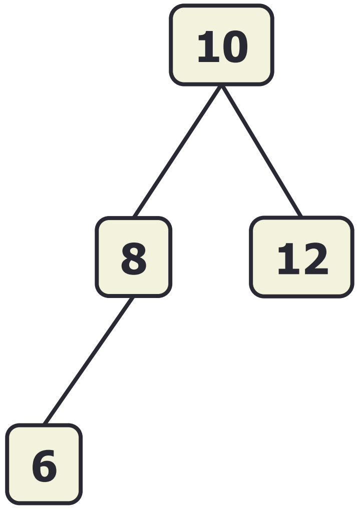
Right-Right Case

- Consider the AVL tree
- Insert **16** in the tree
- Travel upwards from new node to find first imbalance node i.e. **12**
- If new node is right-right to imbalance node, its right-right case
- Do a right rotate at **12** to balance tree
- Now the tree is again balanced



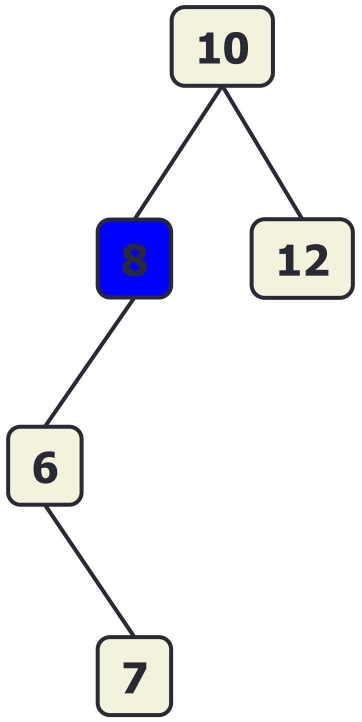
Left-Right Case

- Consider the AVL tree
- Insert **7** in the tree



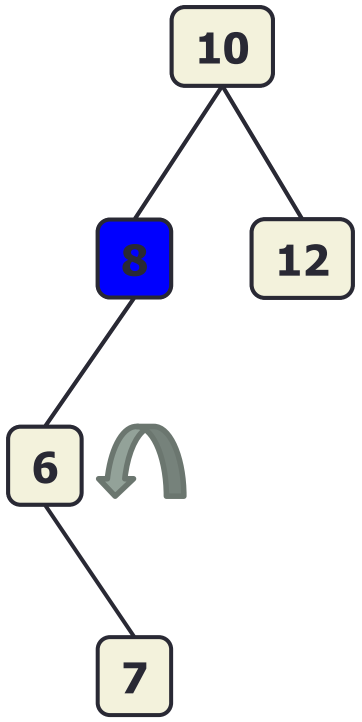
Left-Right Case

- Consider the AVL tree
- Insert **7** in the tree
- Travel upwards from new node to find first imbalance node i.e. **8**
- If new node is left-right to imbalance node, its left-right case



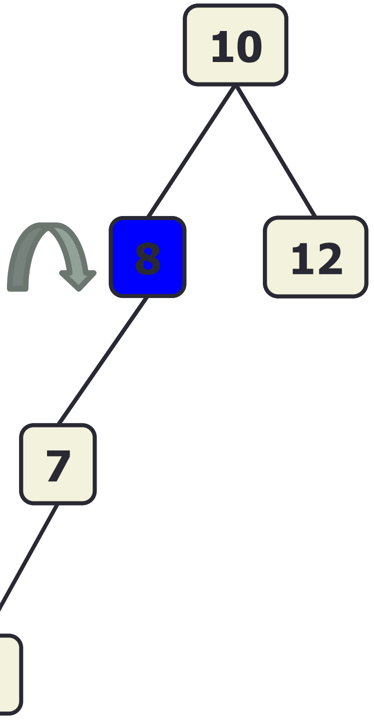
Left-Right Case

- Consider the AVL tree
- Insert **7** in the tree
- Travel upwards from new node to find first imbalance node i.e. **8**
- If new node is left-right to imbalance node, its left-right case
- Two steps to balance:
  - Do a left rotate at **8->left (i.e. 6)**



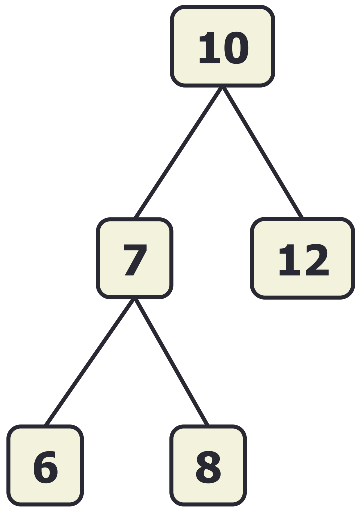
Left-Right Case

- Consider the AVL tree
- Insert **7** in the tree
- Travel upwards from new node to find first imbalance node i.e. **8**
- If new node is left-right to imbalance node, its left-right case
- Two steps to balance:
  - Do a left rotate at **8->left (i.e. 6)**
  - Do a right rotate at **8**



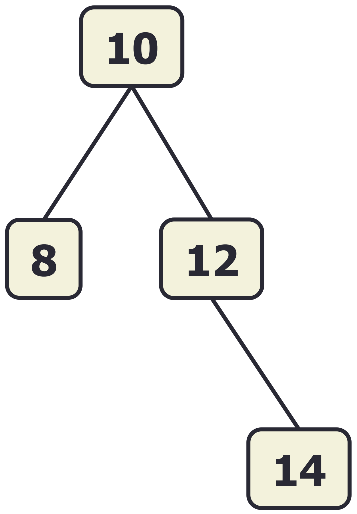
Left-Right Case

- Consider the AVL tree
- Insert **7** in the tree
- Travel upwards from new node to find first imbalance node i.e. **8**
- If new node is left-right to imbalance node, its left-right case
- Two steps to balance:
  - Do a left rotate at **8->left (i.e. 6)**
  - Do a right rotate at **8**



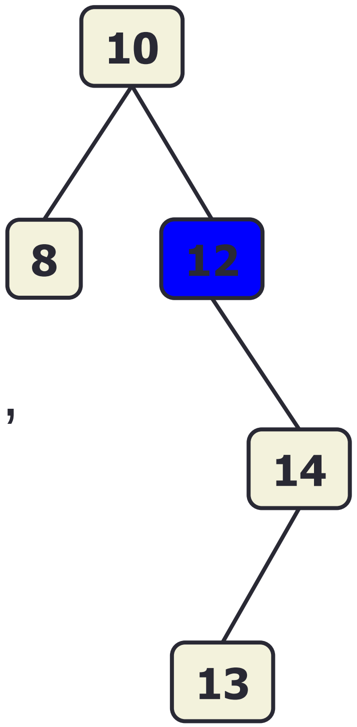
Right-Left Case

- Consider the AVL tree
- Insert **13** in the tree



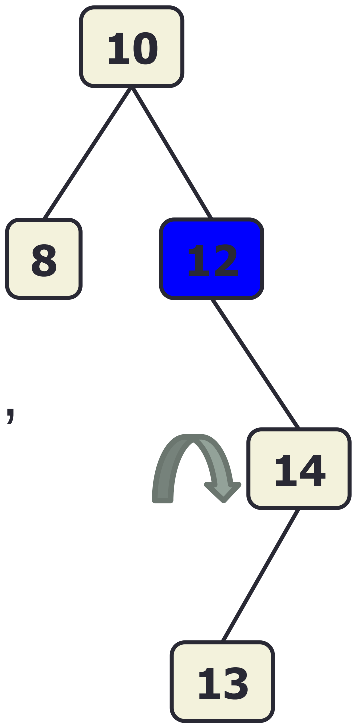
Right-Left Case

- Consider the AVL tree
- Insert **13** in the tree
- Travel upwards from new node to find first imbalance node i.e. **12**
- If new node is right-right to imbalance node, its right-right case



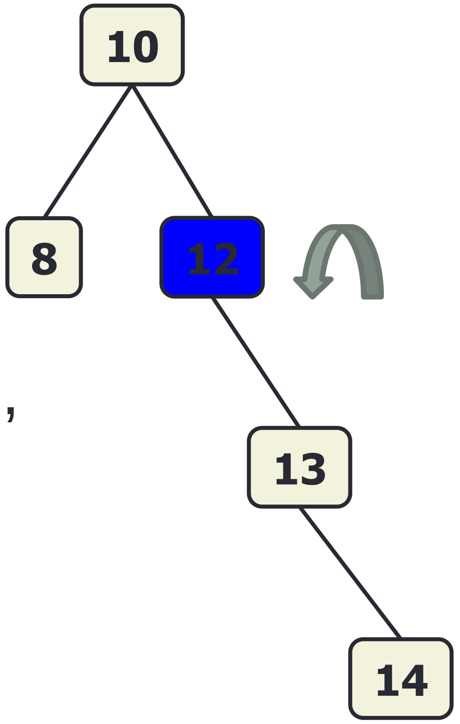
Right-Left Case

- Consider the AVL tree
- Insert **13** in the tree
- Travel upwards from new node to find first imbalance node i.e. **12**
- If new node is right-right to imbalance node, its right-right case
- Two steps to balance:
  - Do a rotate right at 12->right(i.e. 14)



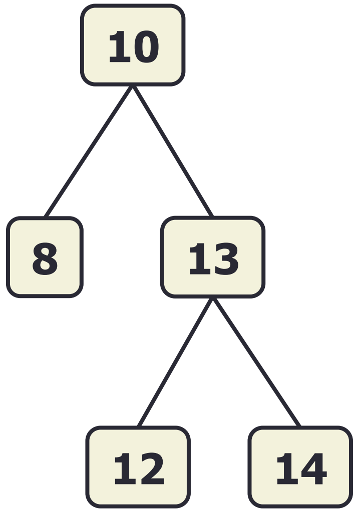
Right-Left Case

- Consider the AVL tree
- Insert **13** in the tree
- Travel upwards from new node to find first imbalance node i.e. **12**
- If new node is right-right to imbalance node, its right-right case
- Two steps to balance:
  - Do a rotate right at 12->right(i.e. 14)
  - Do a rotate left at 12



Right-Left Case

- Consider the AVL tree
- Insert **13** in the tree
- Travel upwards from new node to find first imbalance node i.e. **12**
- If new node is right-right to imbalance node, its right-right case
- Two steps to balance:
  - Do a rotate right at 12->right(i.e. 14)
  - Do a rotate left at 12



Computing Height

- Height of node: number of edges between node and farthest leaf
- We can calculate height of any node recursively
- height(x) = 1 + max(height(left), height(right))
- Height of leaf nodes are 0.

Compute Height : Implementation

```
int height(tree) {
    if (tree == NULL) {
        return -1;  //For NULL tree, eases computation
    }
    else {
        return (1 + max(heightBST(tree->left),
                                heightBST(tree->right)));
    }
}
```

Time Complexity = **O(n)**

Insertion in AVL : Implementation

- We alter the node structure to have one more field **ht**
- Updated node structure therefore is

```
typedef struct Node {
        int data;
        struct Node * left;
        struct Node * right;
        int ht;
} * Nptr;
```

- “ht” of leaf nodes are 0. Thus every new node has ht=0

Insertion in AVL : Implementation (cont.)

```
//Algorithm
Nptr insertAVL(Nptr tree, int data) {
    1. insert similar to BST
    2. This insertion might have caused change in height of ancestors of new node, thus update “height” of ancestors
    3. Check balance factor of every ancestor as we move up in the AVL Tree
    4. if balance factor is greater than 1, test for left-left or left-right case and fix accordingly
    5. if balance factor is less than -1, test for right-right or right-left case and fix accordingly
}
```

Insertion in AVL : Implementation (cont.)

```
int getHt(Nptr tree) {
    if (tree == NULL)
        return -1;
    else
        return (tree->ht);
}

int balanceFactor(Nptr tree) {
    if (tree == NULL)
        return 0;
    else
        return (getHt(tree->left) - getHt(tree->right));
}
```

Insertion in AVL : Implementation (cont.)

```
Nptr leftRotate(Nptr tree) {
    Nptr X = tree->right;
    Nptr Y = X->left;

    tree->right = Y;
    X->left = tree;

    tree->ht = 1 + max(getHt(tree->left), getHt(tree->right));
    X->ht = 1 + max(getHt(X->left),getHt(X->right));
    return X;
}
```

Insertion in AVL : Implementation (cont.)

```
Nptr rightRotate(Nptr tree) {
    Nptr X = tree->left;
    Nptr Y = X->right;

    tree->left = Y;
    X->right = tree;

    tree->ht = 1 + max(getHt(tree->left), getHt(tree->right));
    X->ht = 1 + max(getHt(X->left),getHt(X->right));
    return X;
}
```

Insertion in AVL : Implementation (cont.)

```
Nptr insertAVL(Nptr tree, int data) {
    //insertBST function, thus skipping writing
    //update height of current node
    tree->ht= 1 + max(getHt(tree->left), getHt(tree->right));
    int bal = balanceFactor(tree);

    if (bal > 1 && data < tree->left->data)  //left-left case
        return rightRotate(tree);

    else if (bal > 1 && data > tree->left->data) {  //left-right case
        tree->left = leftRotate(tree->left);
        return rightRotate(tree);
    }
```

Insertion in AVL : Implementation (cont.)

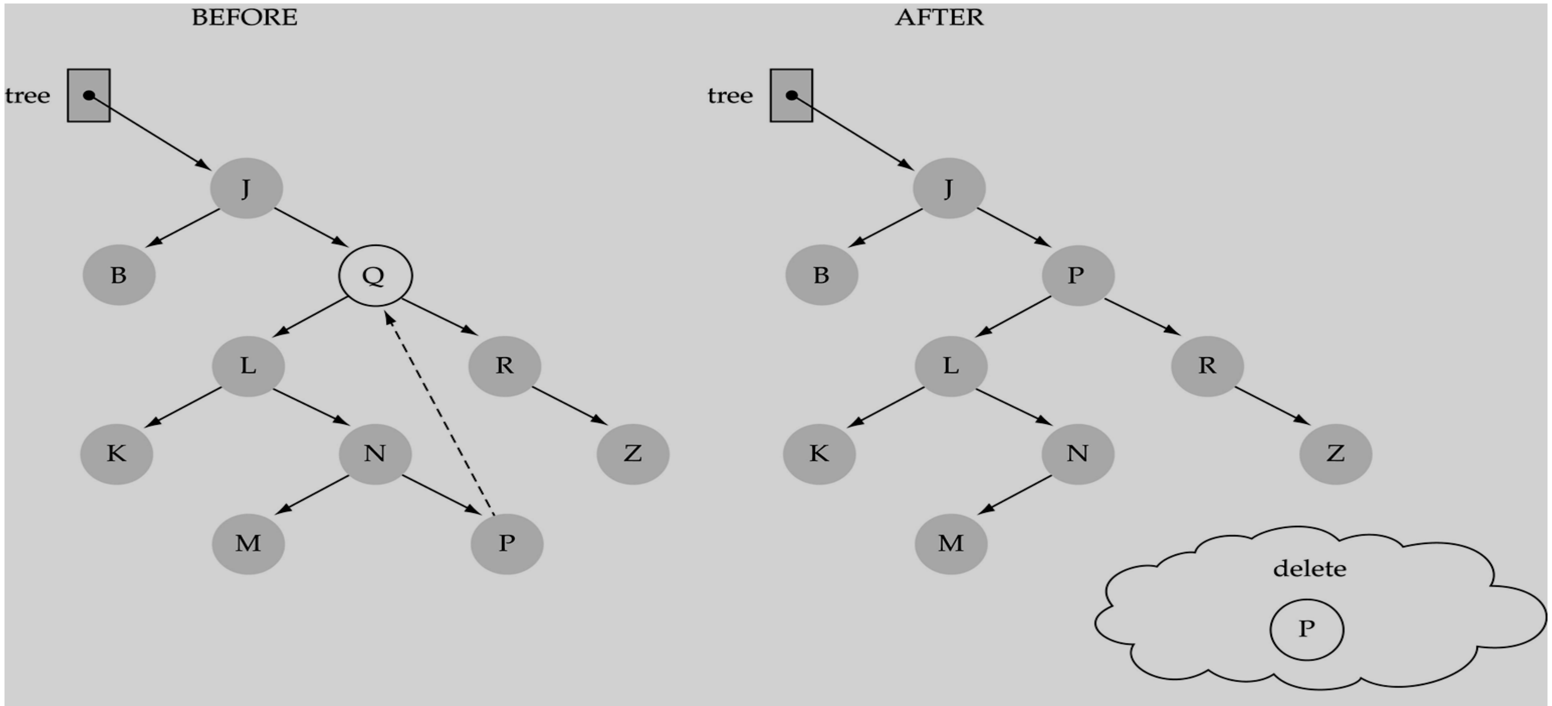
```
    else if (bal<-1 && data < tree->right->data) {  //right-left case
        tree->right = rightRotate(tree->right);
        return leftRotate(tree);
    }

    else if (bal < -1 && data > tree->right->data)  //right-right case
        return leftRotate(tree);
    else
        return tree;
}
```

Insertion in AVL : Analysis

- Insert the new key as a new leaf just as in ordinary binary search tree: O(logN)
- Then trace the path from the new leaf towards the root, for each node x encountered: O(logN)
    - Check height difference: O(1)
    - If satisfies AVL property, proceed to next node: O(1)
    - If not, perform single/double rotation: O(1)
- The insertion stops when
    - Single/double rotation is performed
    - Or, we’ve checked all nodes in the path
- Time complexity for insertion O(logN)

Deletion in AVL

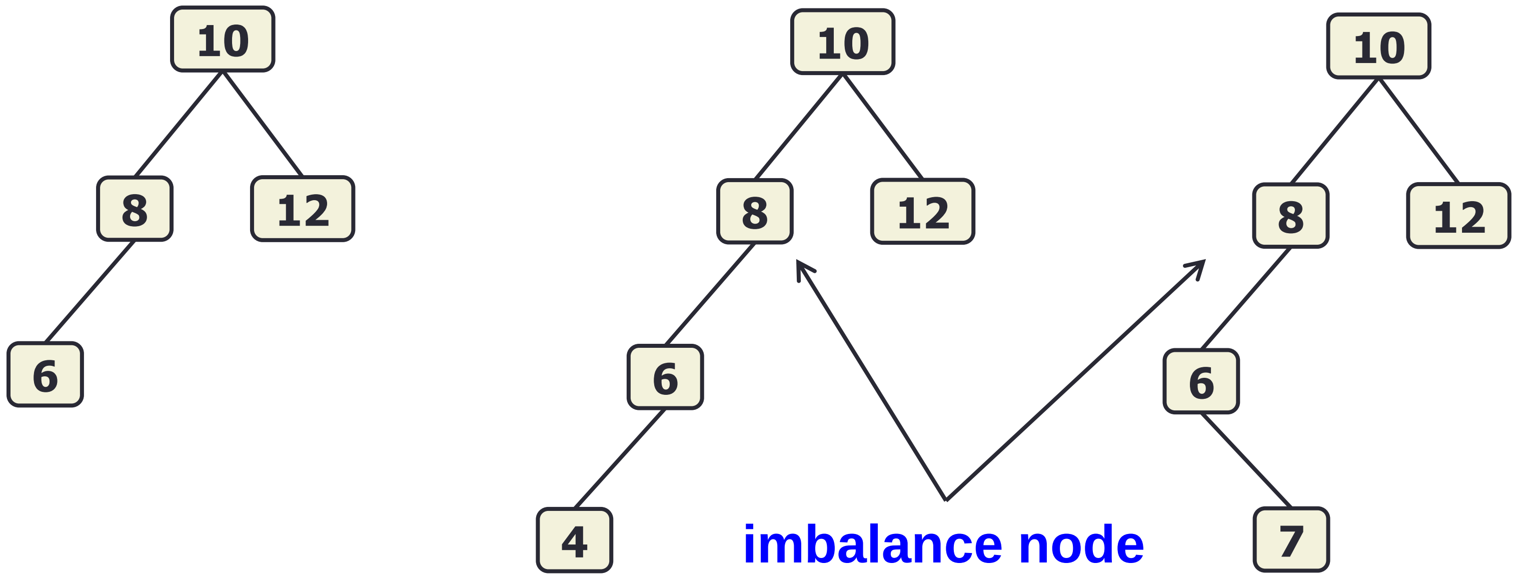
- Deletion in BST always result in deleting a leaf node or node with one child
- For instance, deleting Q results in deleting node p



Delete the node containing Q

Deletion in AVL (cont'd)

- In AVL, child of node with one child is a leaf node. Because if that is not true, it is no longer an AVL tree



- Therefore, either a leaf or parent of leaf node is deleted

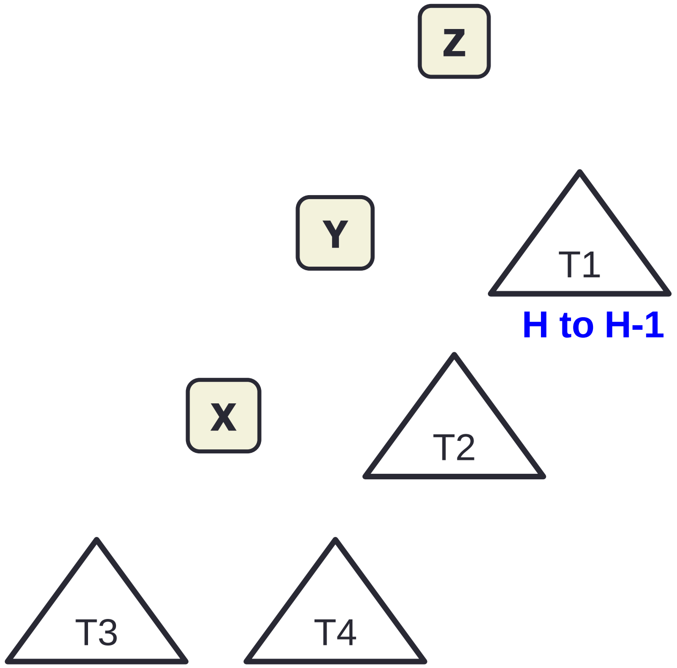
Deletion in AVL (cont'd)

- Let *w* is the node deleted.
- Traverse bottom up in AVL tree up to root

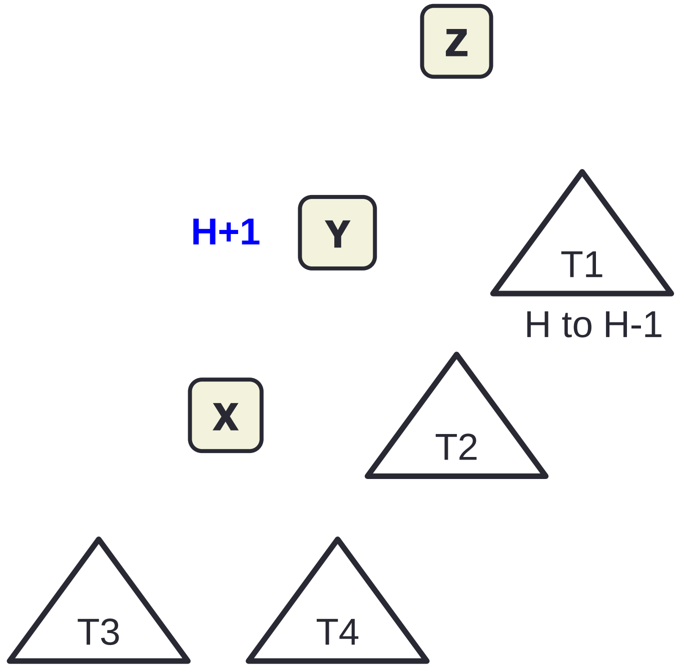
- Let *z* is the first imbalance node. *y* be the child of *z* with larger height and *x* be the child of *y* with larger height.
- Make rotation(s) based on the arrangement between *x*, *y* and *z*.

- Rotation(s) can disturb the balance of ancestors of *z*. Hence keep checking for imbalance node as we move upward in AVL tree

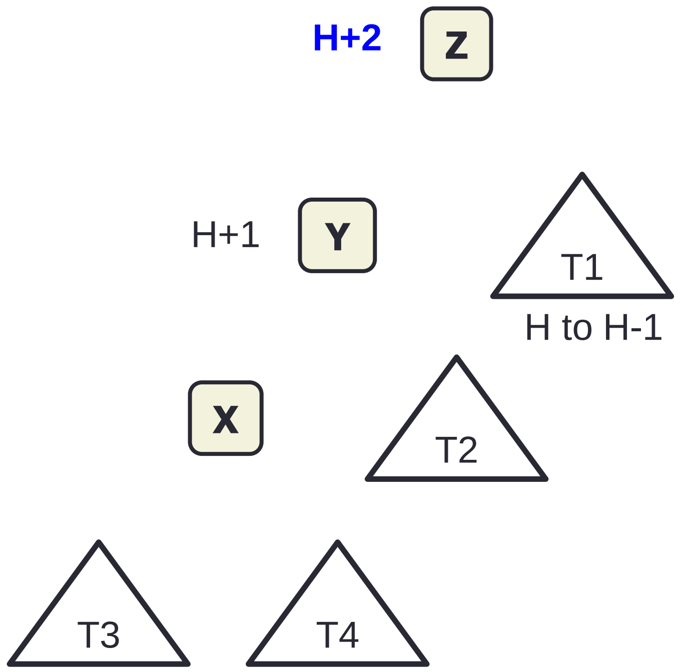
Deletion in AVL : Example(1)



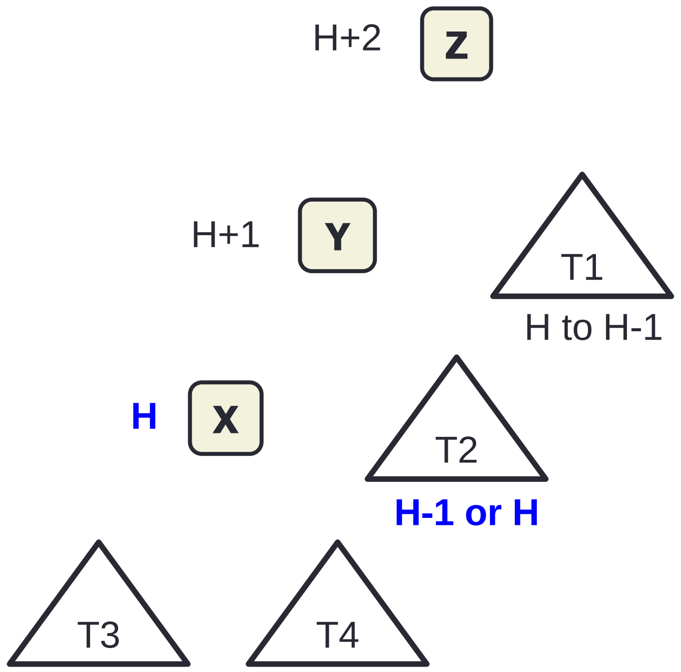
Deletion in AVL : Example(1)



Deletion in AVL : Example(1)

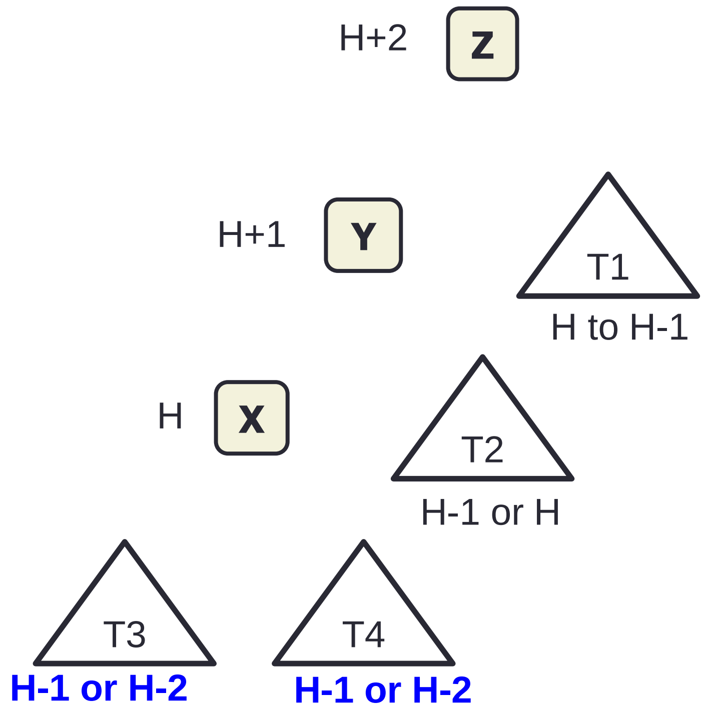


Deletion in AVL : Example(1)



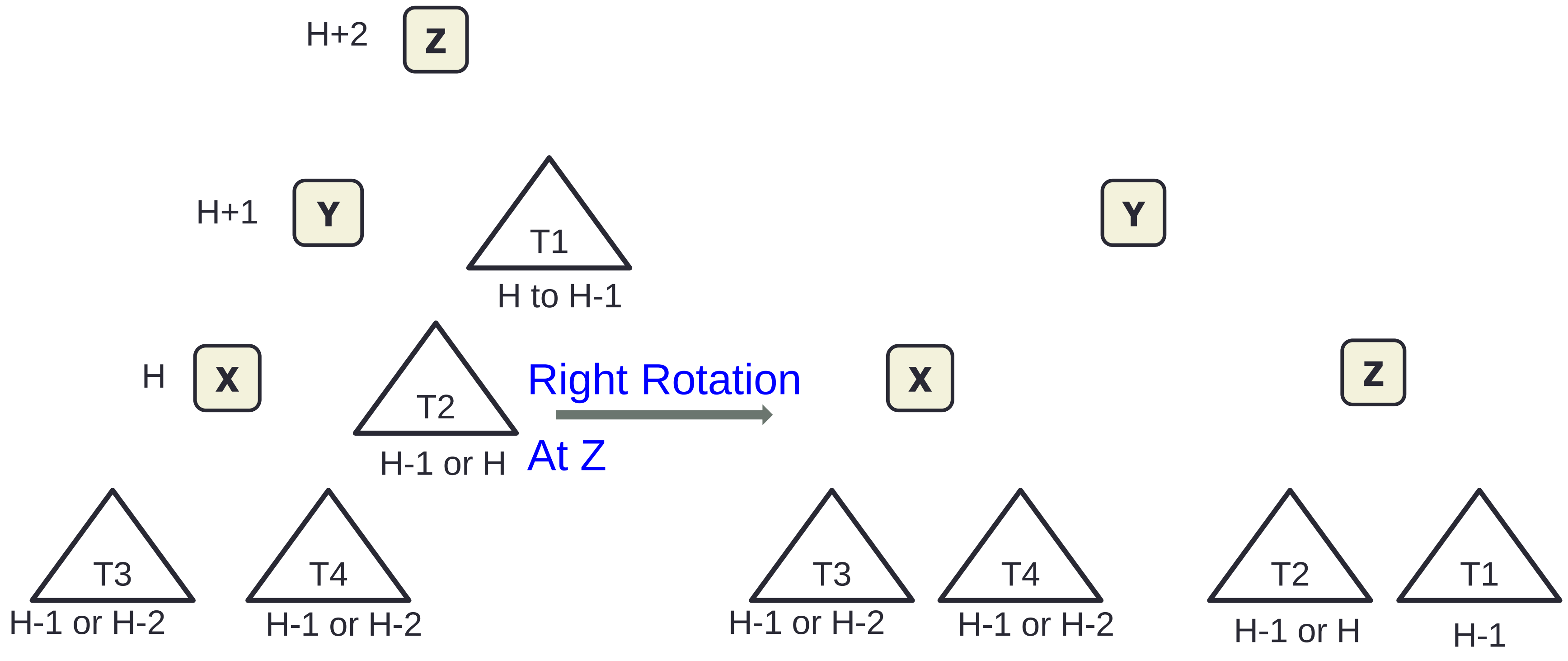
Deletion in AVL : Example(1)

- Since *x* is balanced ht(T3) and ht(T4) are H-1 or H-2 but both can not have H-2.



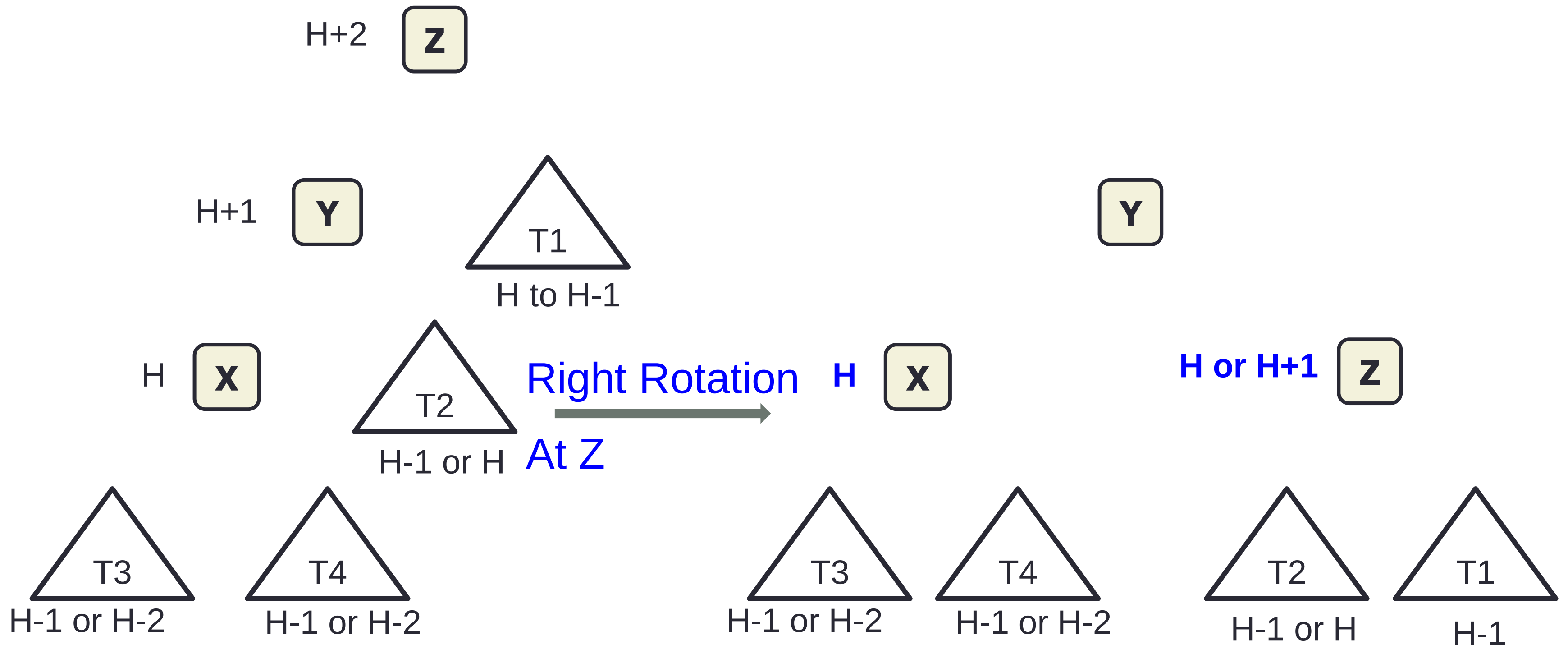
Deletion in AVL : Example(1) (cont'd)

- Since *x* is balanced ht(T3) and ht(T4) are H-1 or H-2 but both can not have H-2.



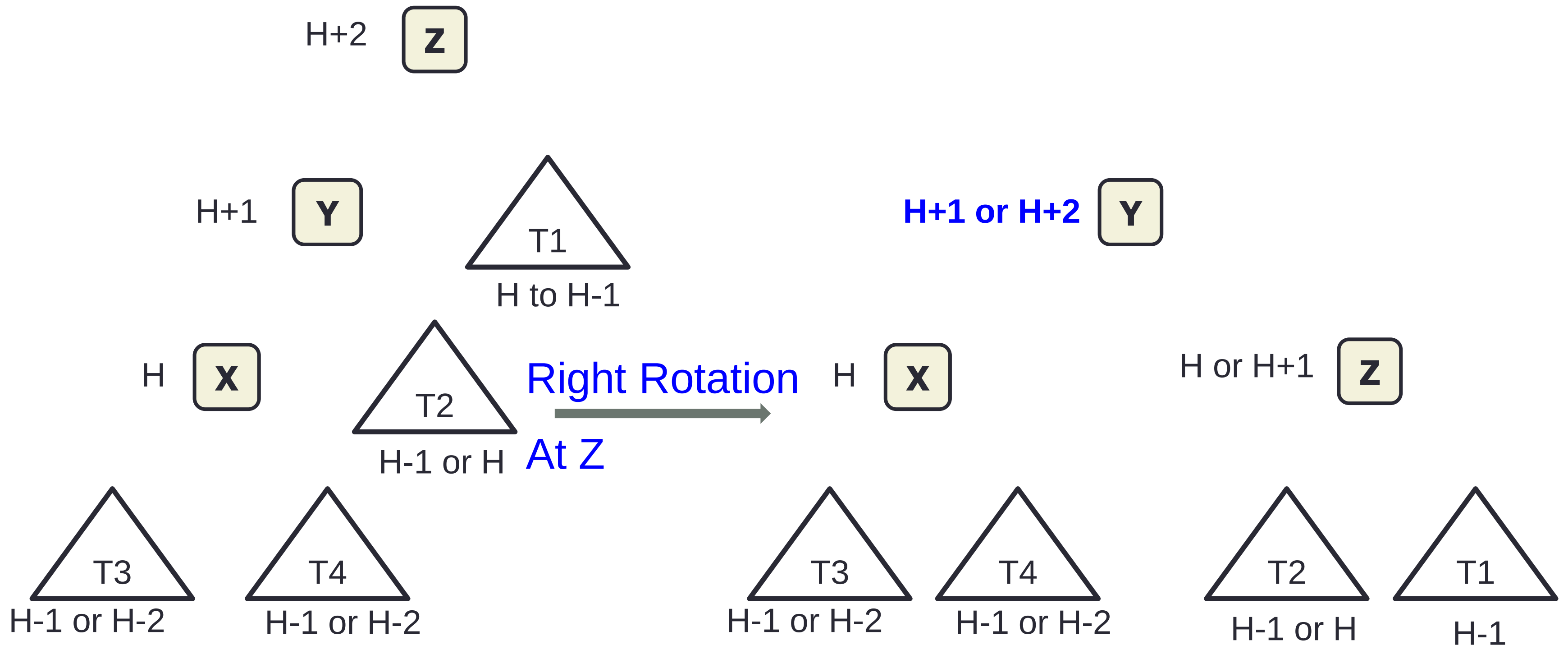
Deletion in AVL : Example(1) (cont'd)

- Since *x* is balanced ht(T3) and ht(T4) are H-1 or H-2 but both can not have H-2.



Deletion in AVL : Example(1) (cont'd)

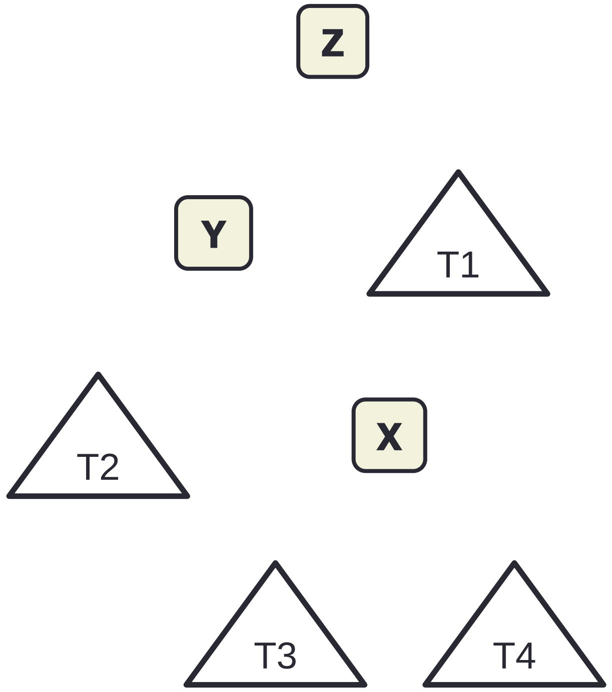
- Since *x* is balanced ht(T3) and ht(T4) are H-1 or H-2 but both can not have H-2.



Deletion in AVL : Example(1) (cont'd)

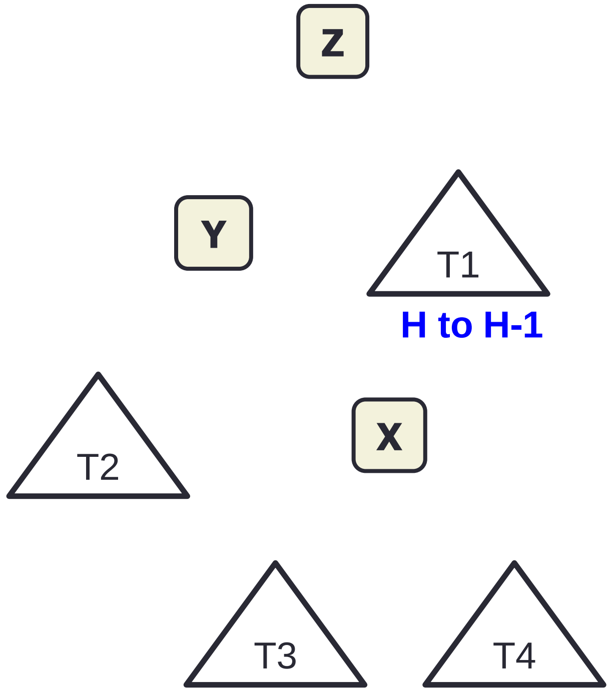
- Originally, the height of tree rooted at z was H+2.
- After rotation, sub-tree of height could be H+1 or H+2
- If height gets reduced, we will have to continue upwards to check imbalance

Deletion in AVL : Example(2)



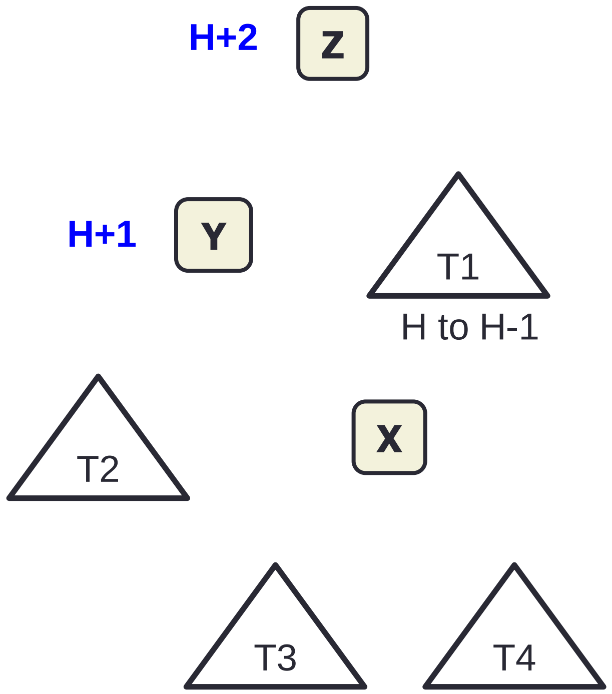
Deletion in AVL : Example(2)

- Let the deleted node *w* was part of T1



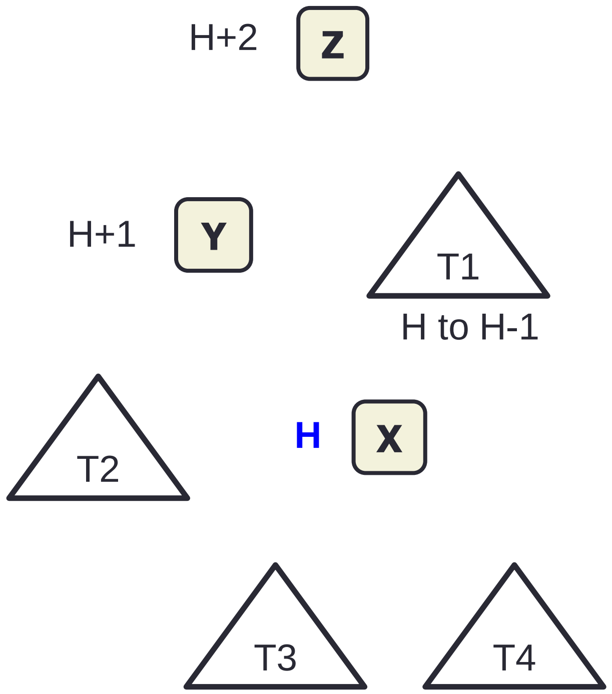
Deletion in AVL : Example(2)

- Let the deleted node *w* was part of T1



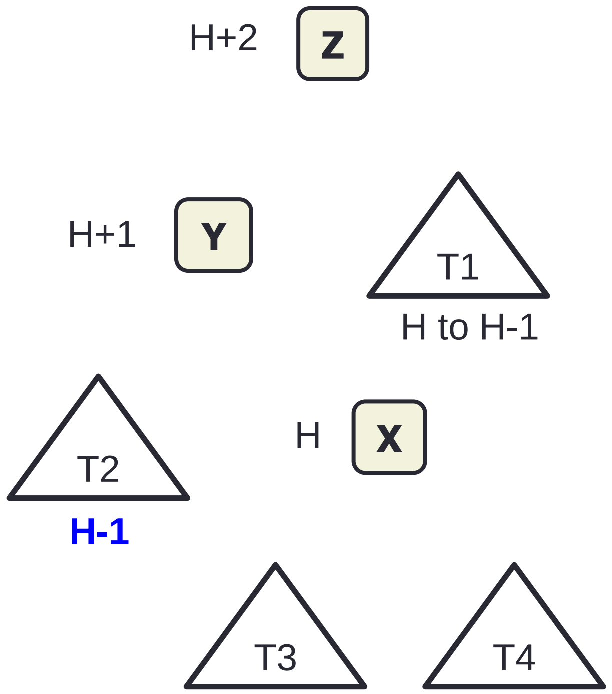
Deletion in AVL : Example(2)

- Let the deleted node *w* was part of T1



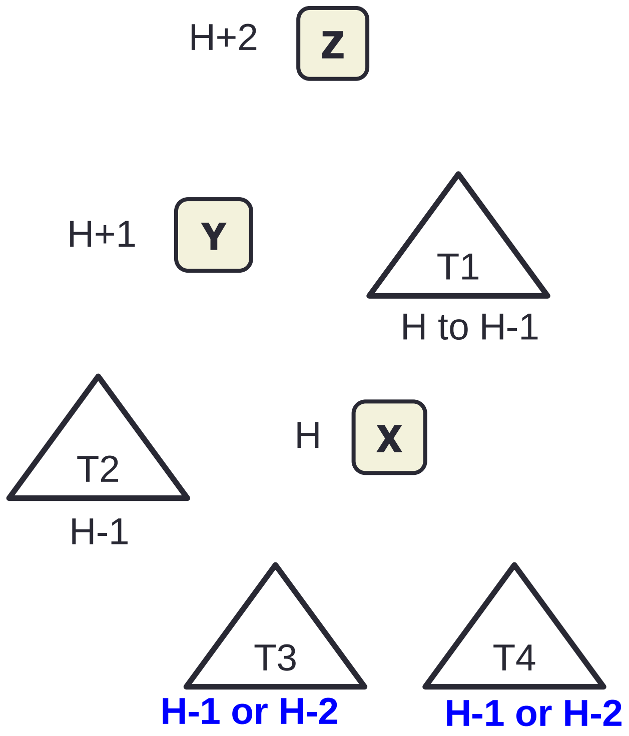
Deletion in AVL : Example(2)

- Let the deleted node *w* was part of T1

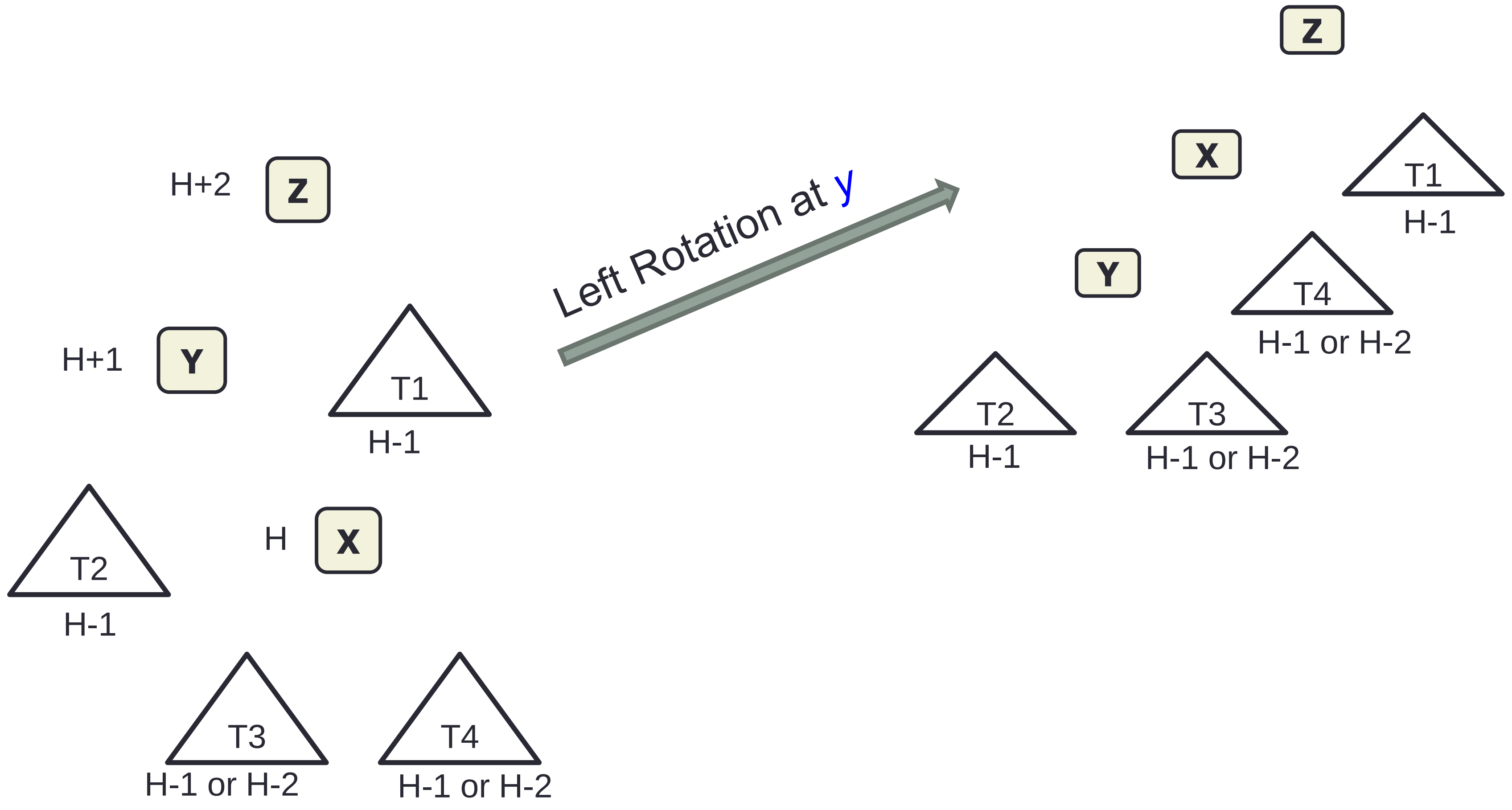


Deletion in AVL : Example(2)

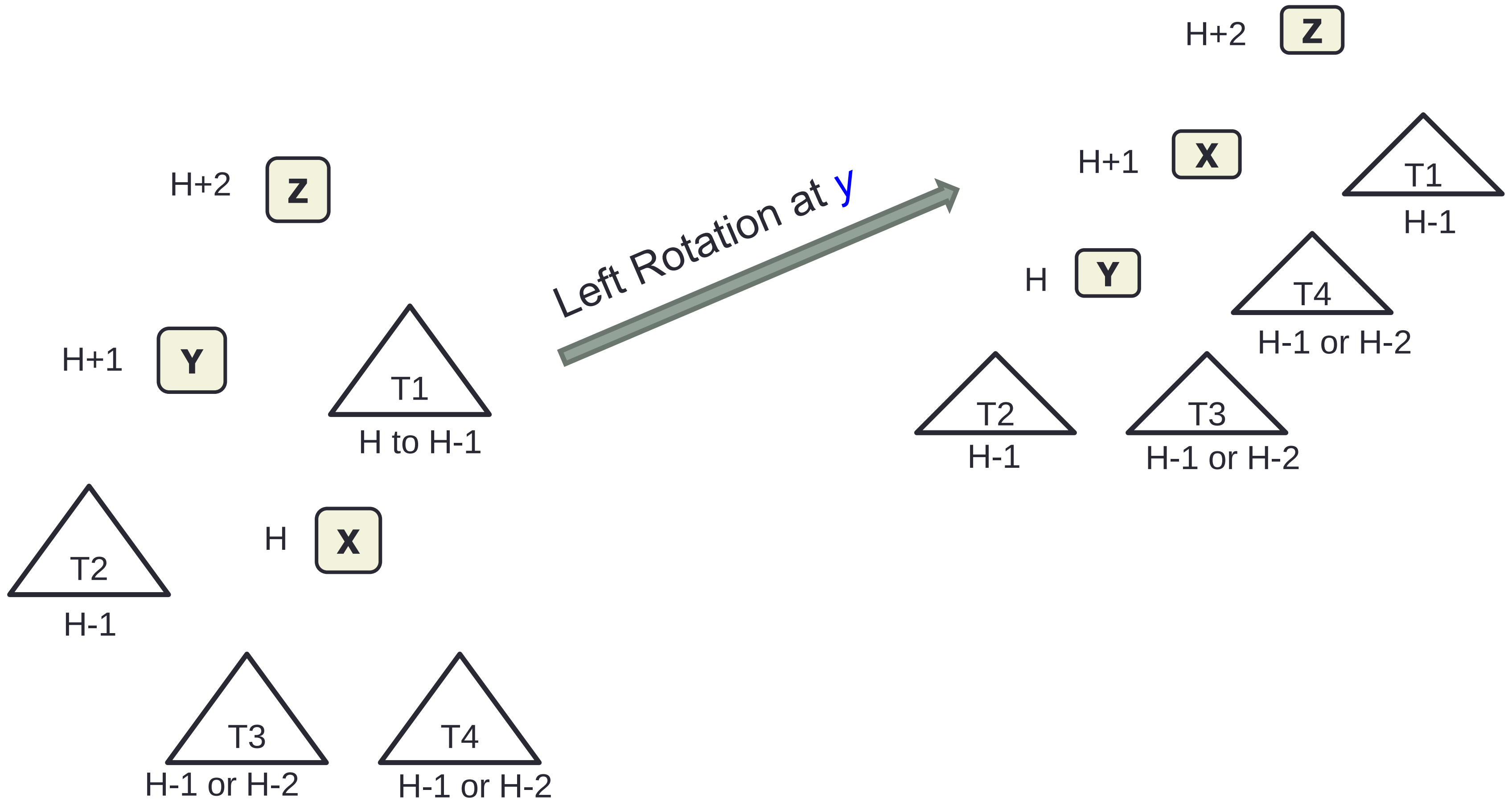
- Again, T3 and T4 both can’t have height H-2.
- If height of T2 would have been H, we would have picked root of T2 as *x*.



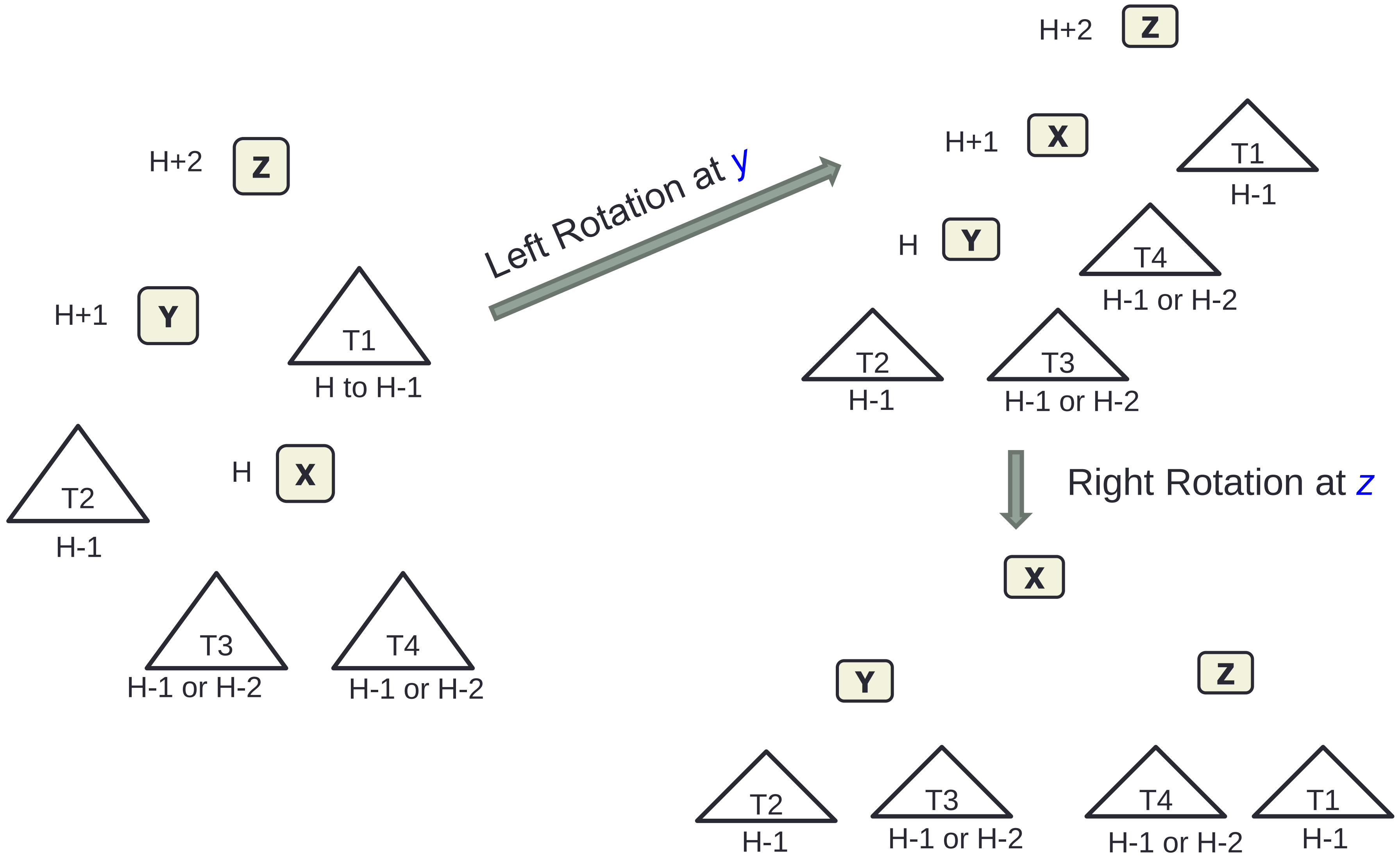
Deletion in AVL : Example(2) (cont'd)



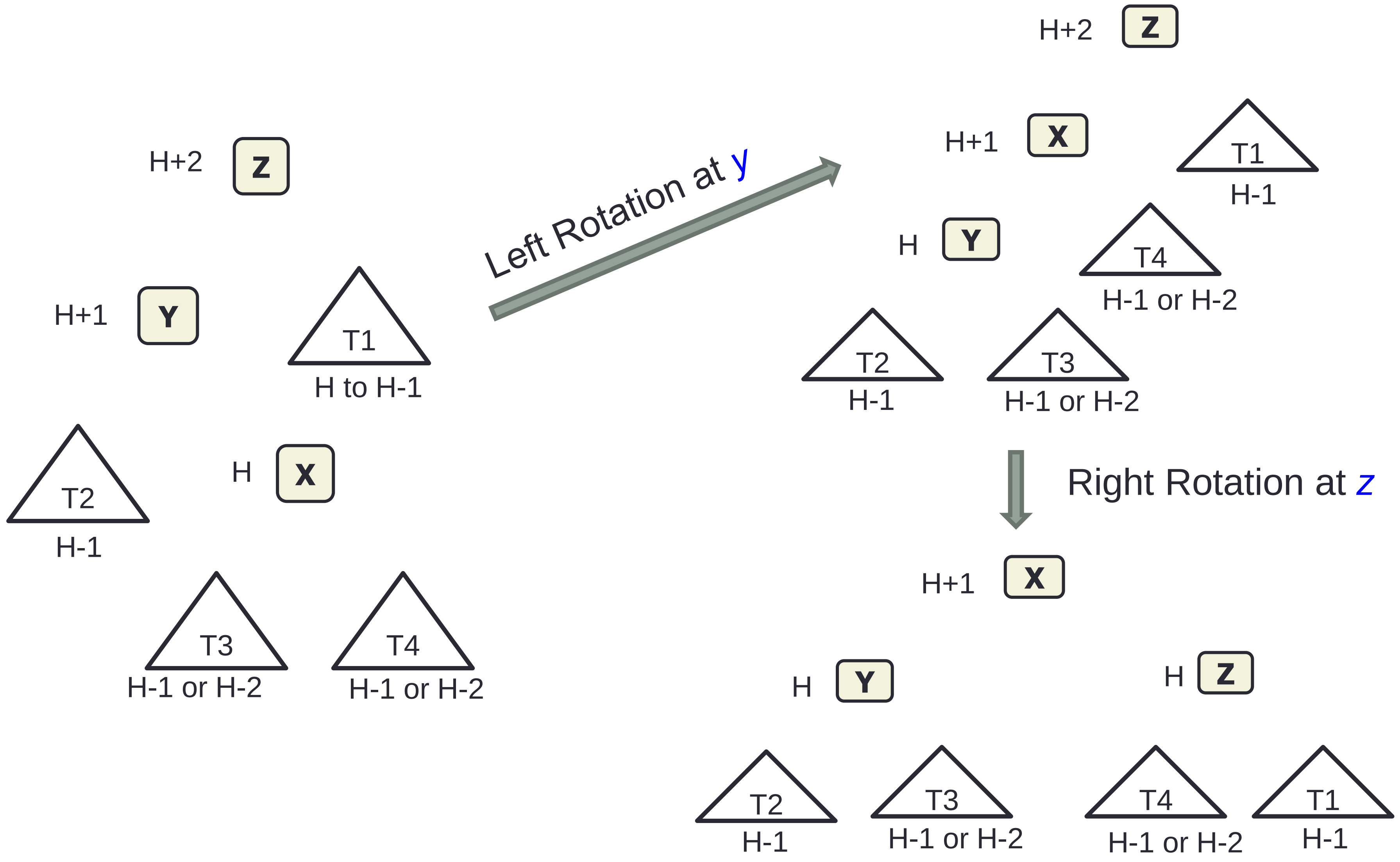
Deletion in AVL : Example(2) (cont’d)



Deletion in AVL : Example(2) (cont'd)



Deletion in AVL : Example(2) (cont'd)



Deletion in AVL : Example(2) (cont'd)

- Left-Right case strictly reduces the height of sub-tree originally rooted at z.
- Thus, check for imbalance as we move upward

Deletion in AVL : Algo

1. Delete as in an ordinary BST
2. Find first imbalance node *z* going up from deleted node
3. Find y and x respective to z. If both child of y has same height, pick child which would result in single rotation.
4. Make rotation(s) to balance the tree again and update heights
5. If, height of sub-tree rooted at z is reduced, start again from step 2 else finish.

Deletion in AVL : Analysis

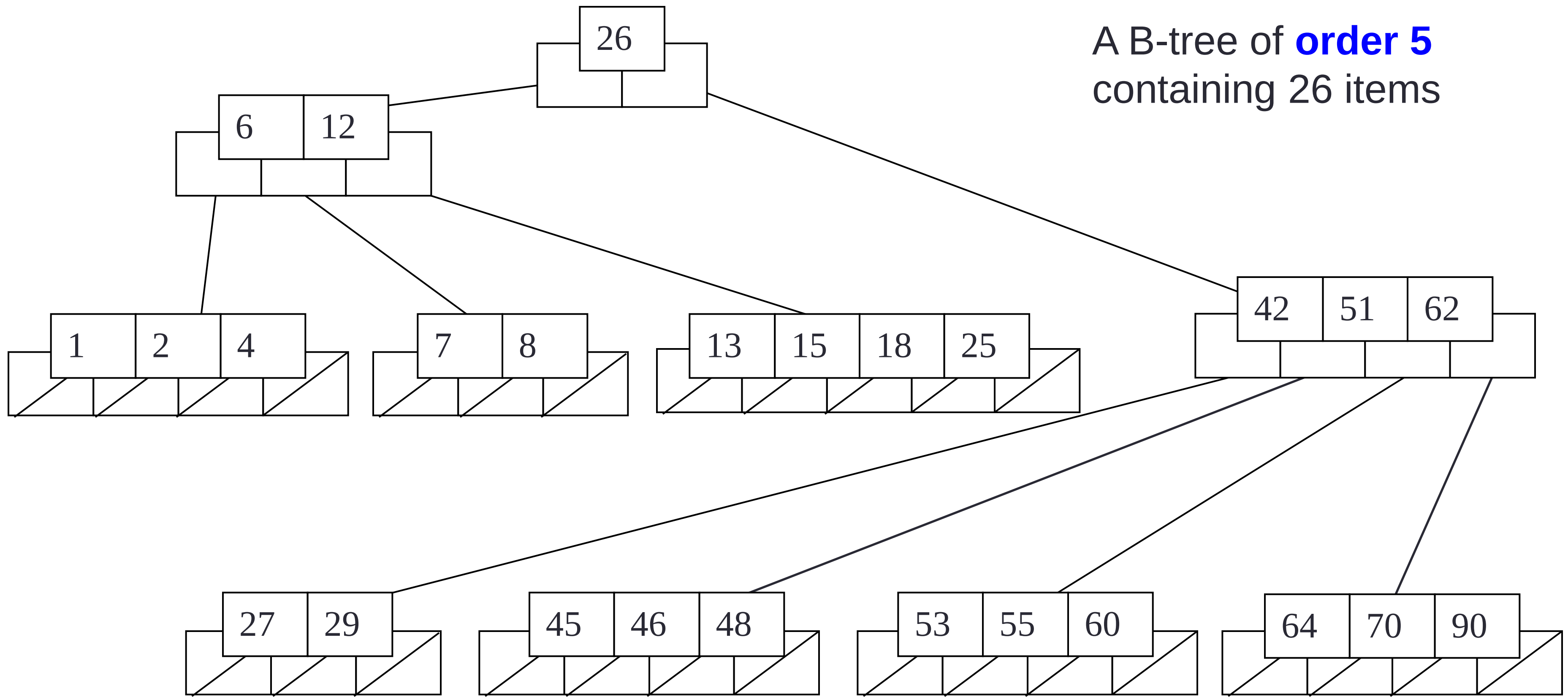
- Delete the new key as in ordinary BST: O(logN)
- Rotations requires constant time : O(1)
- In worst case, we might have to do the rotation O(logN) times
- So, total time required to rebalance AVL tree after deletion is O(logN)
- Time complexity for deletion O(logN)

Inefficiency of AVL Tree

- Large datasets cannot be stored in main memory, and hence stored on disk
- Assume that we use an AVL tree to store about 20 million records
- We end up with a **very** deep binary tree with lots of different disk accesses; $\log_2$ 20,000,000 is about 25
- We know we can't improve on the log *n* lower bound on search for a binary tree
- So to improve efficiency, solution is to use more branches and thus reduce the height of the tree!
    - As branching increases, depth decreases

B-tree

- B-tree of order m is an *m*-way search tree, where
  - Maximum number of children of a non-leaf node(except root) : m
  - Minimum number of children of a non-leaf node(except root): $\lceil m / 2 \rceil$
  - The root has minimum 2 and maximum *m* children
  - Number of keys in each non-leaf node is one less than the number of its children
  - Keys partition the keys in the children in the fashion of a search tree
  - All leaves are on the same level
  - A leaf node contains minimum $\lceil m / 2 \rceil$-1 and maximum $m – 1$ keys



A B-tree of **order 5** containing 26 items

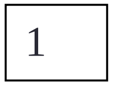
Note that all the leaves are at the same level

Constructing a B-tree

- Suppose we start with an empty B-tree and keys arrive in the following order: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45
- We want to construct a B-tree of order 5
- Remember, Key inside a node are always sorted to maintain search

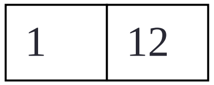
Constructing a B-tree (cont'd)

keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



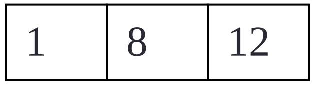
Constructing a B-tree (cont'd)

keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



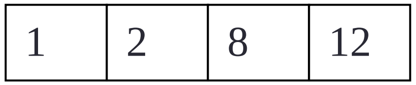
Constructing a B-tree (cont'd)

keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



Constructing a B-tree (cont'd)

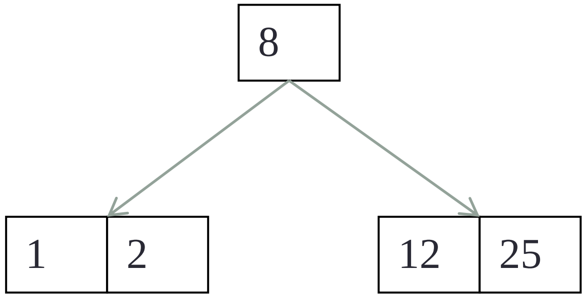
keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



Constructing a B-tree (cont'd)

keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45

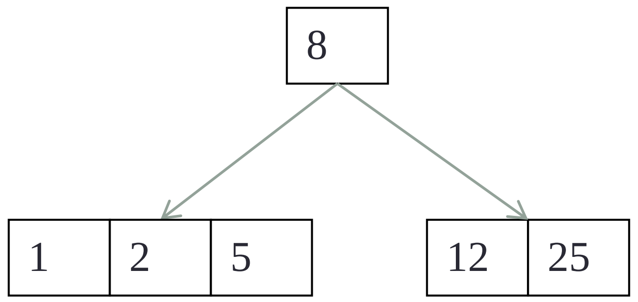
- Node can not have 5 keys, hence split the node and create new root with middle key 8



- Splitting the root node results in increasing the height by 1

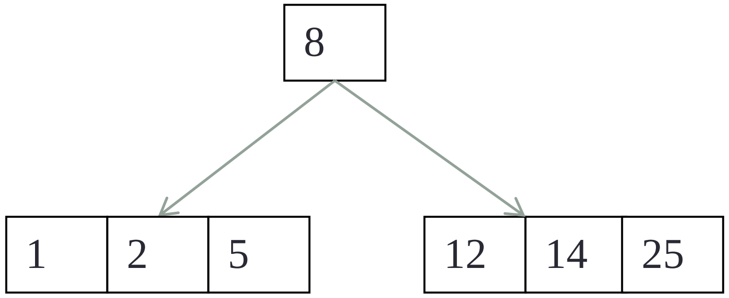
Constructing a B-tree (cont'd)

keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



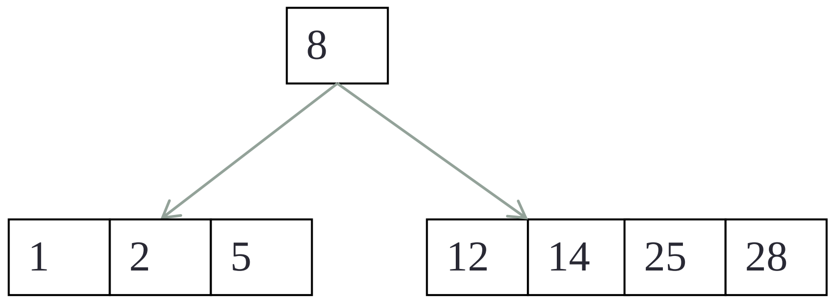
Constructing a B-tree (cont'd)

keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



Constructing a B-tree (cont'd)

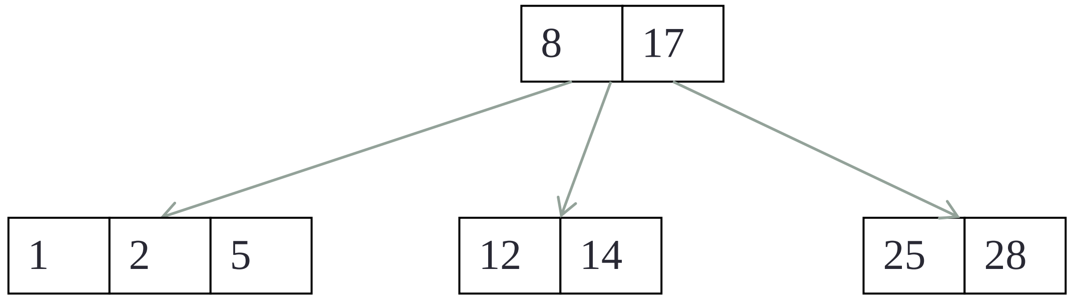
keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



Constructing a B-tree (cont'd)

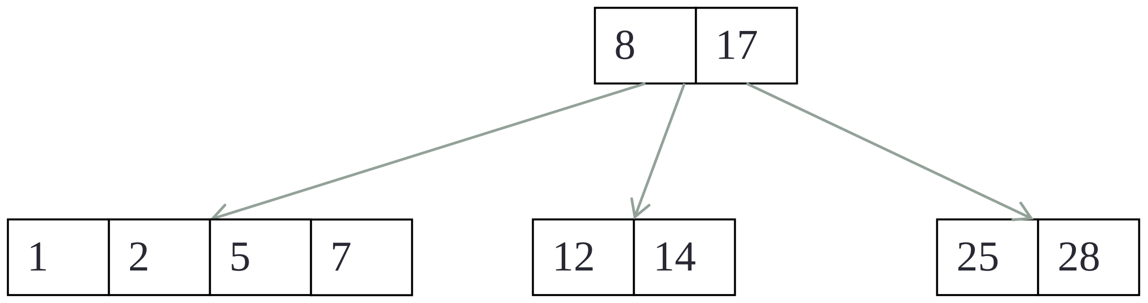
keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45

- Inserting 17 will result in node split. The middle key 17 is promoted to parent



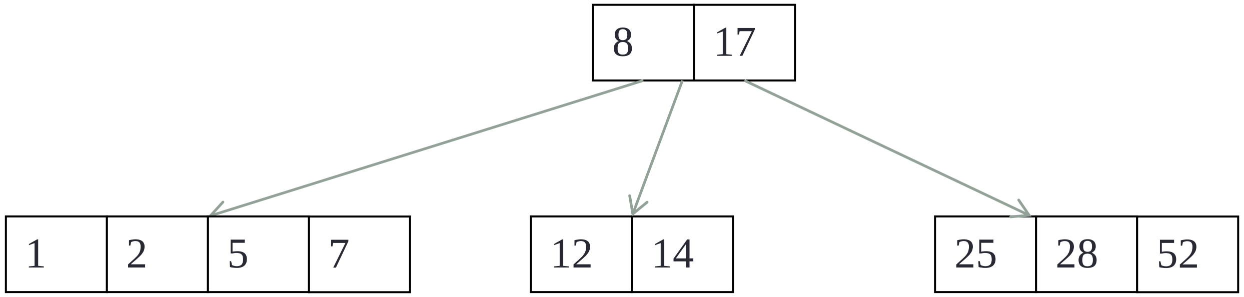
Constructing a B-tree (cont'd)

keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



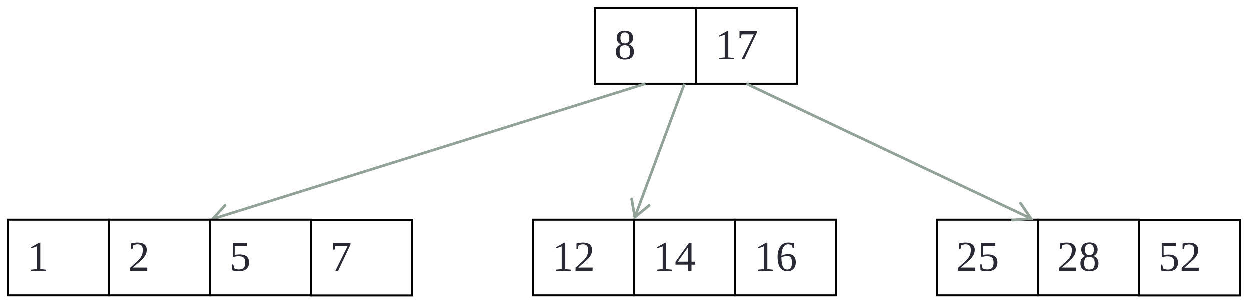
Constructing a B-tree (cont'd)

keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



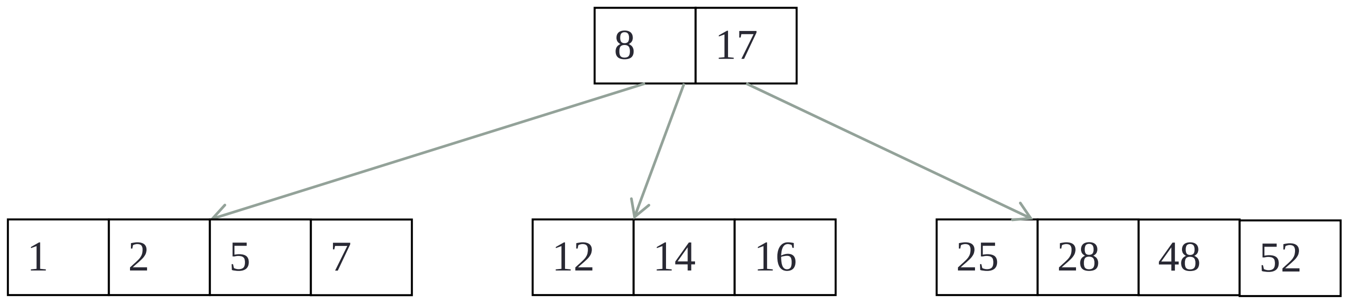
Constructing a B-tree (cont'd)

keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



Constructing a B-tree (cont'd)

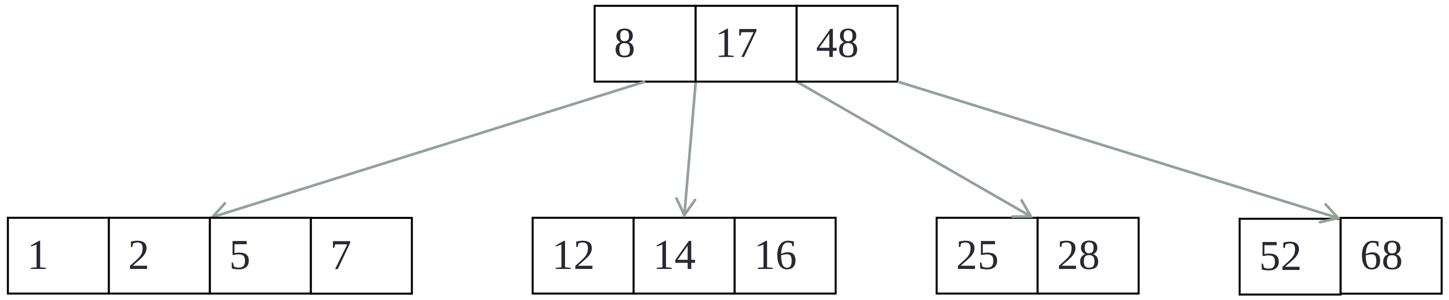
keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



Constructing a B-tree (cont'd)

keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45

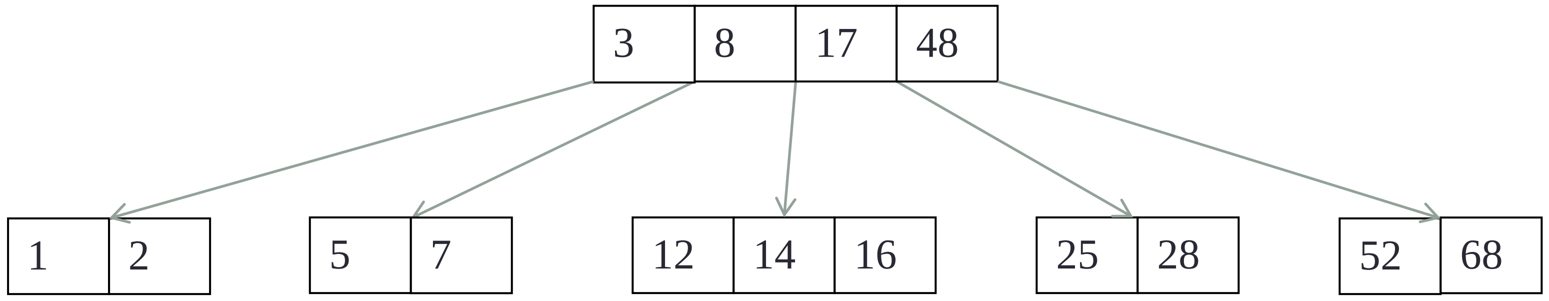
- Node will be split and middle key 48 will be promoted to parent



Constructing a B-tree (cont'd)

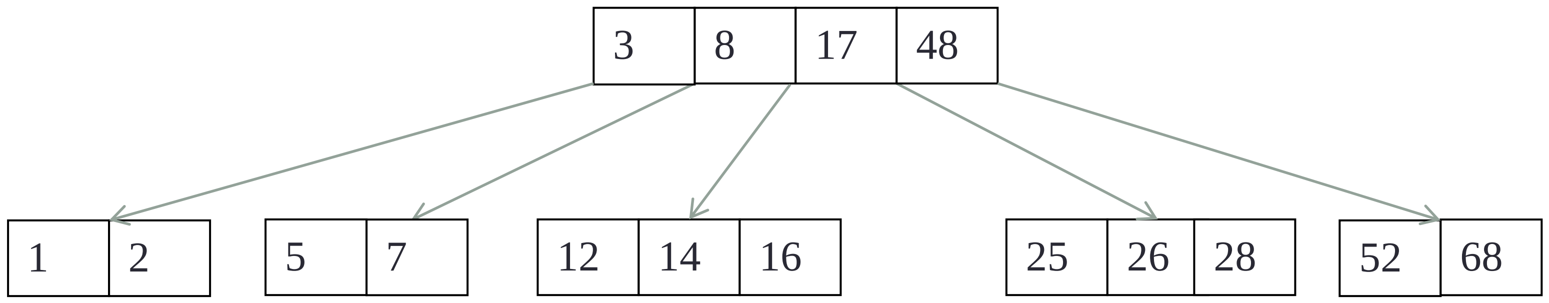
keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45

- Node will be split and middle key will be promoted to parent



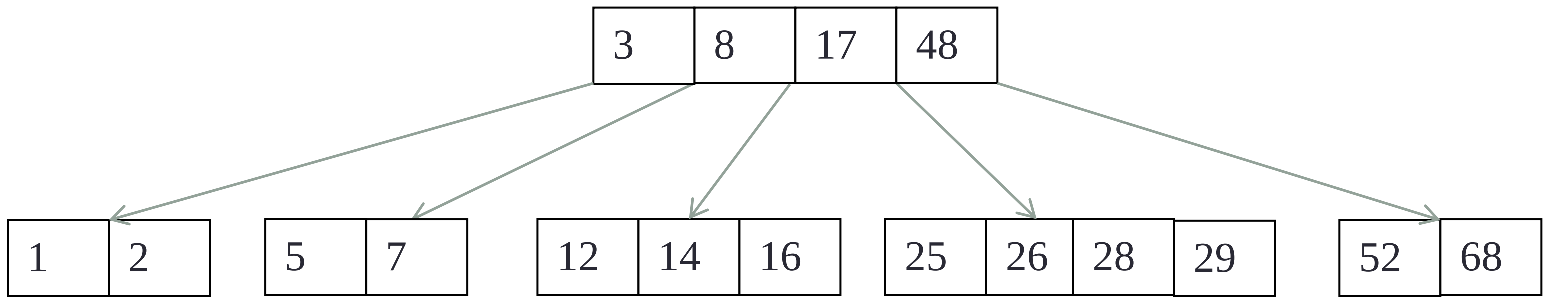
Constructing a B-tree (cont'd)

keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



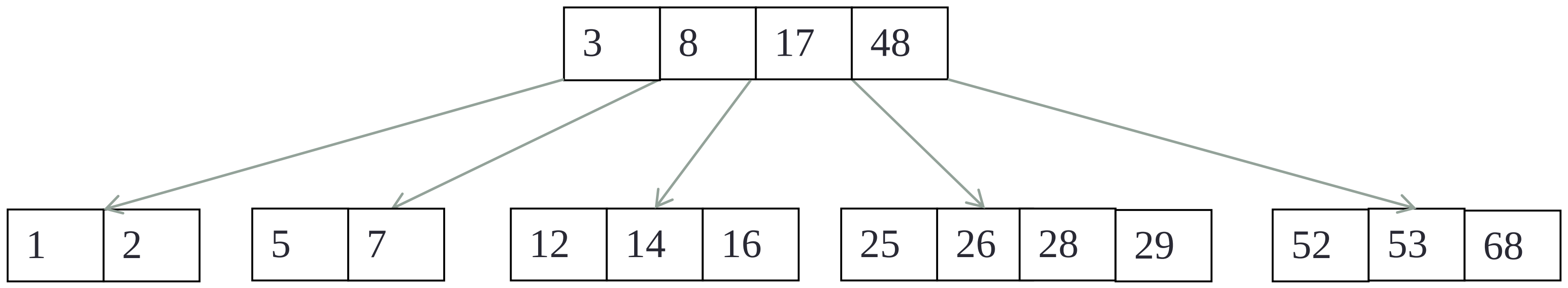
Constructing a B-tree (cont'd)

keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



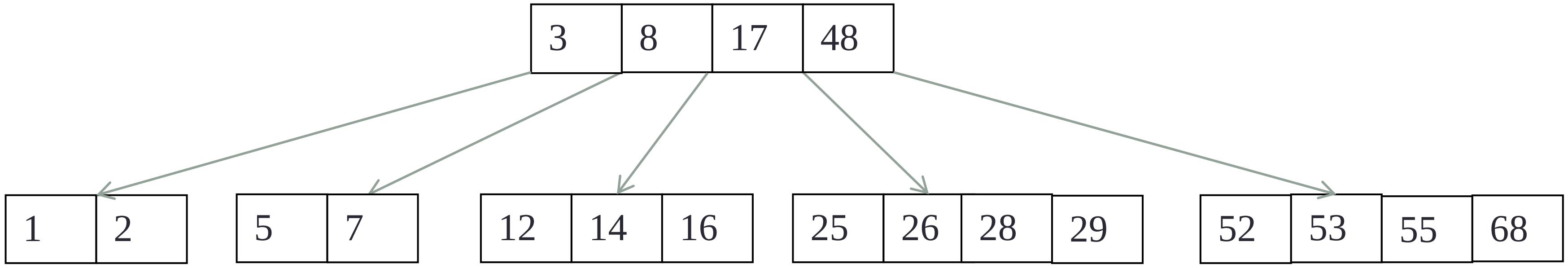
Constructing a B-tree (cont'd)

keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



Constructing a B-tree (cont'd)

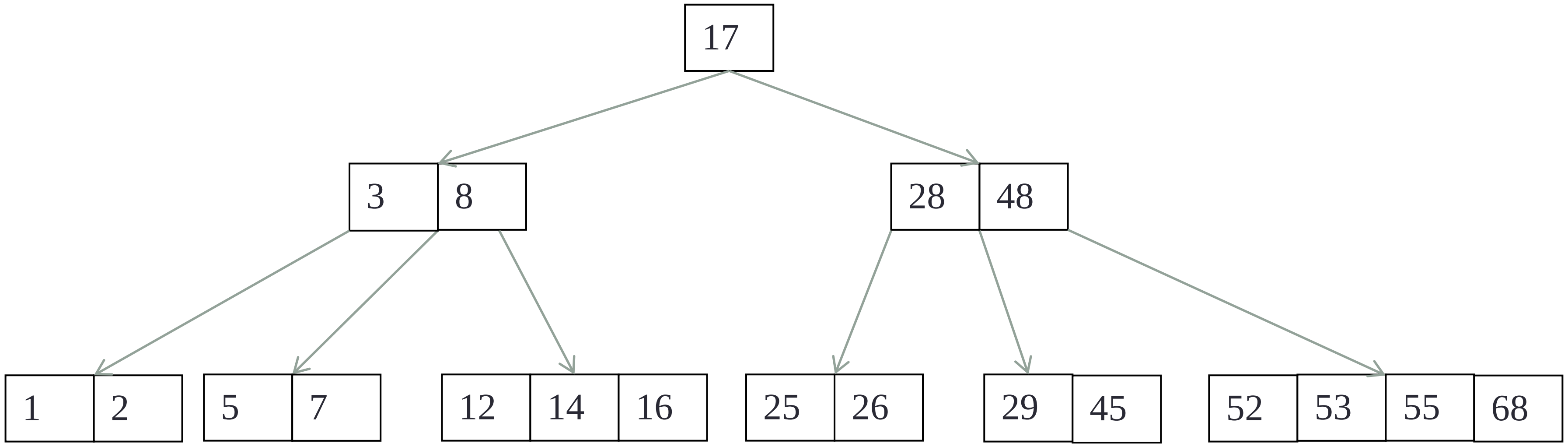
keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45



Constructing a B-tree (cont'd)

keys: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45

- As 45 can not be accommodated in leaf node, we will split the node and promote middle key 28 to parent
- Further parent node also needs to be split
- As the parent is also root node, will increase the height of B-tree



Insertion in B-tree: Algo

1. Attempt to insert the new key into a leaf
2. If this could result in too many keys in leaf node, split the leaf node and promote the key upward to parent node
3. Repeat step 2 all the way to the top
4. If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

Insertion in B-tree : Analysis

- Finding the correct location:
  - In worst case, we might require to compare input “key” with all key present in one node i.e. *m-1* comparison. But *m* is a constant, thus time required is O(1)
  - Total number of levels to be traversed is O(h) or O(log n)
- Checking the tree after insertion is B-tree
  - Split takes constant time as few pointers only needs to be updated. Hence, O(1)
  - Might have to do up to root. Thus, O(log n)

- Total time complexity for insertion: O(log n)

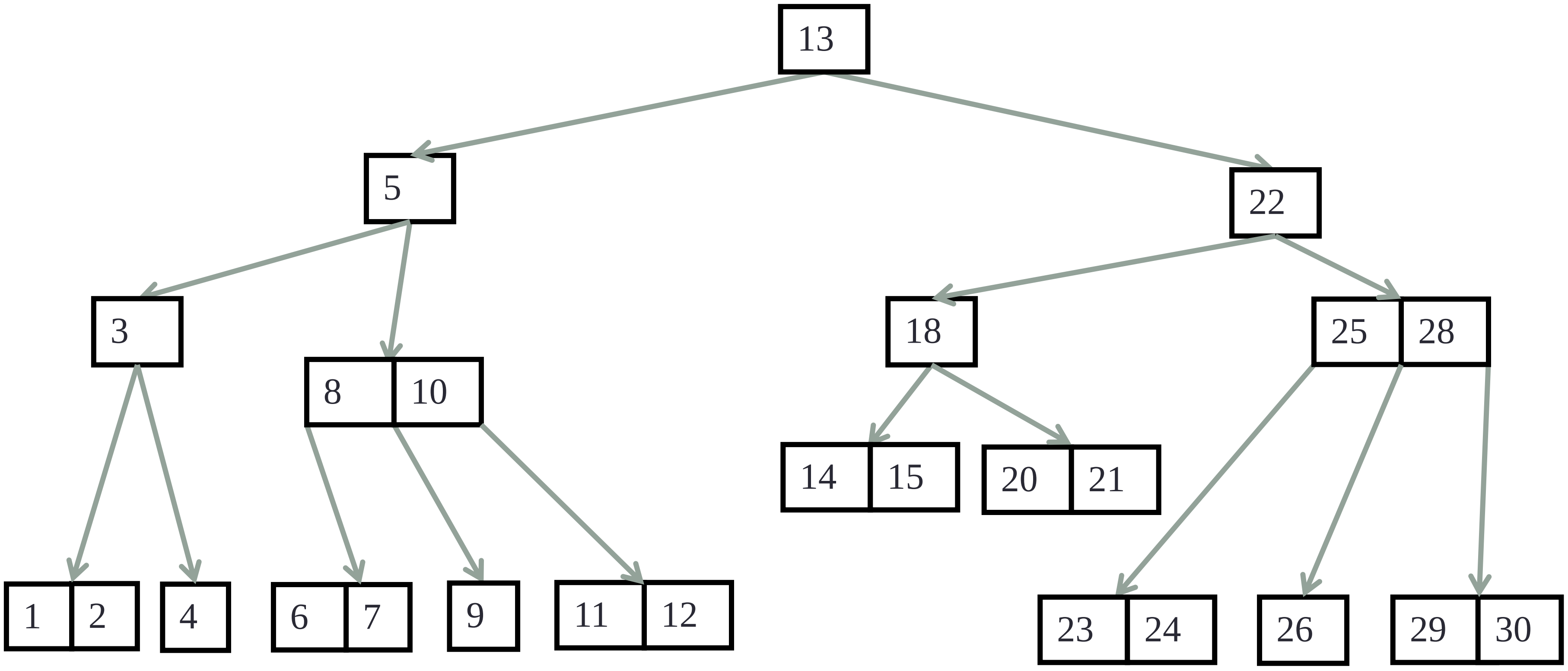
Exercise

Insert the following keys to a 5-way B-tree:

3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

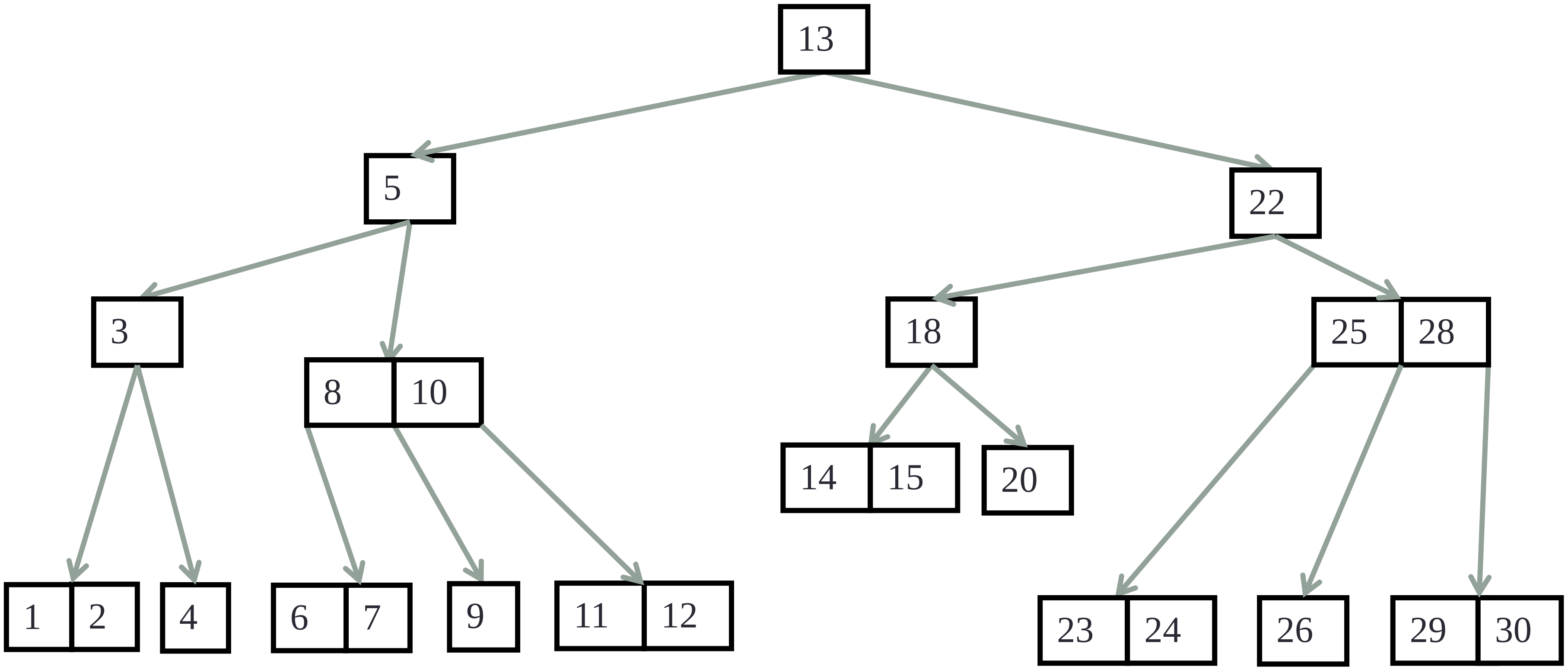
Deletion in B-tree

- Order is B-tree is 4. Max and min no. children are 4 and 2.
- Delete key in given order: 21, 25, 20, 23, 18



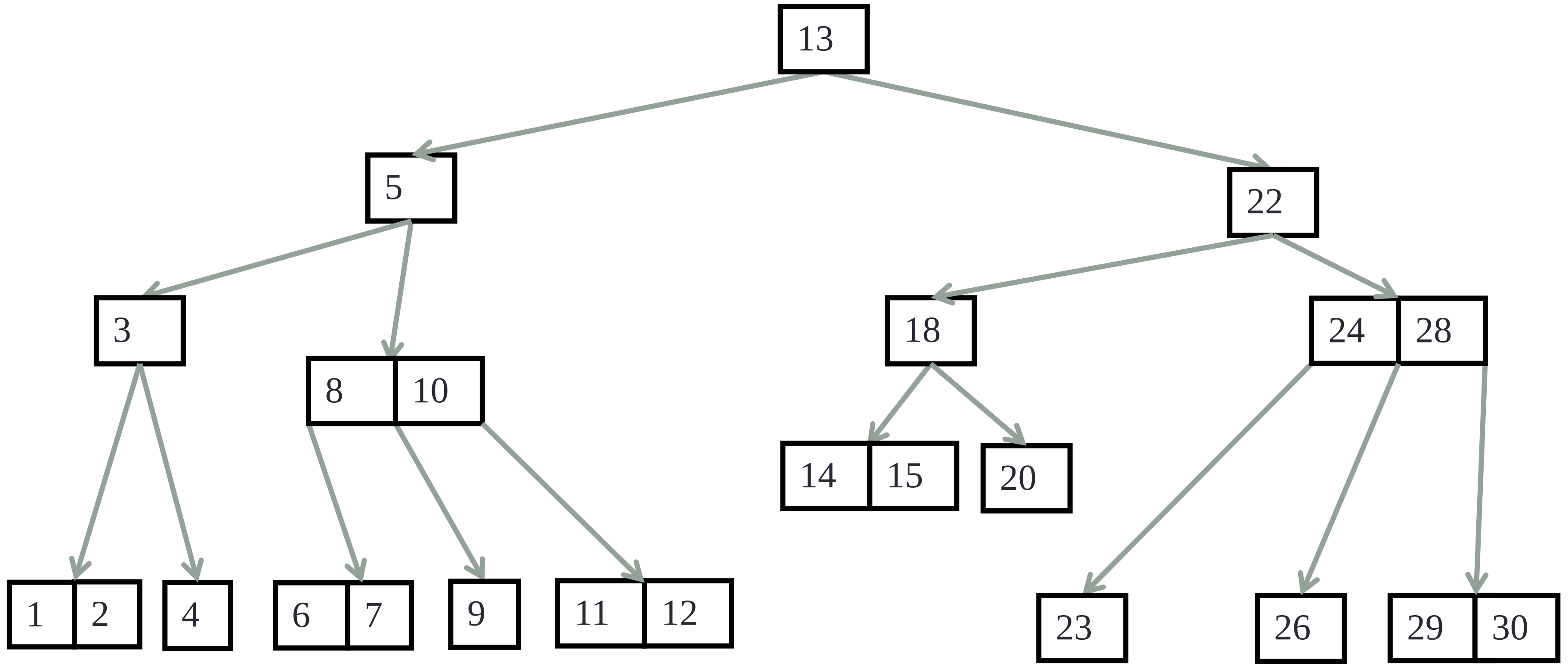
Deletion in B-tree(Cont'd)

- Delete key in given order: 21, 25, 20, 23, 18
- When deletion is done from leaf node, if node satisfies minimum key requirement, do nothing



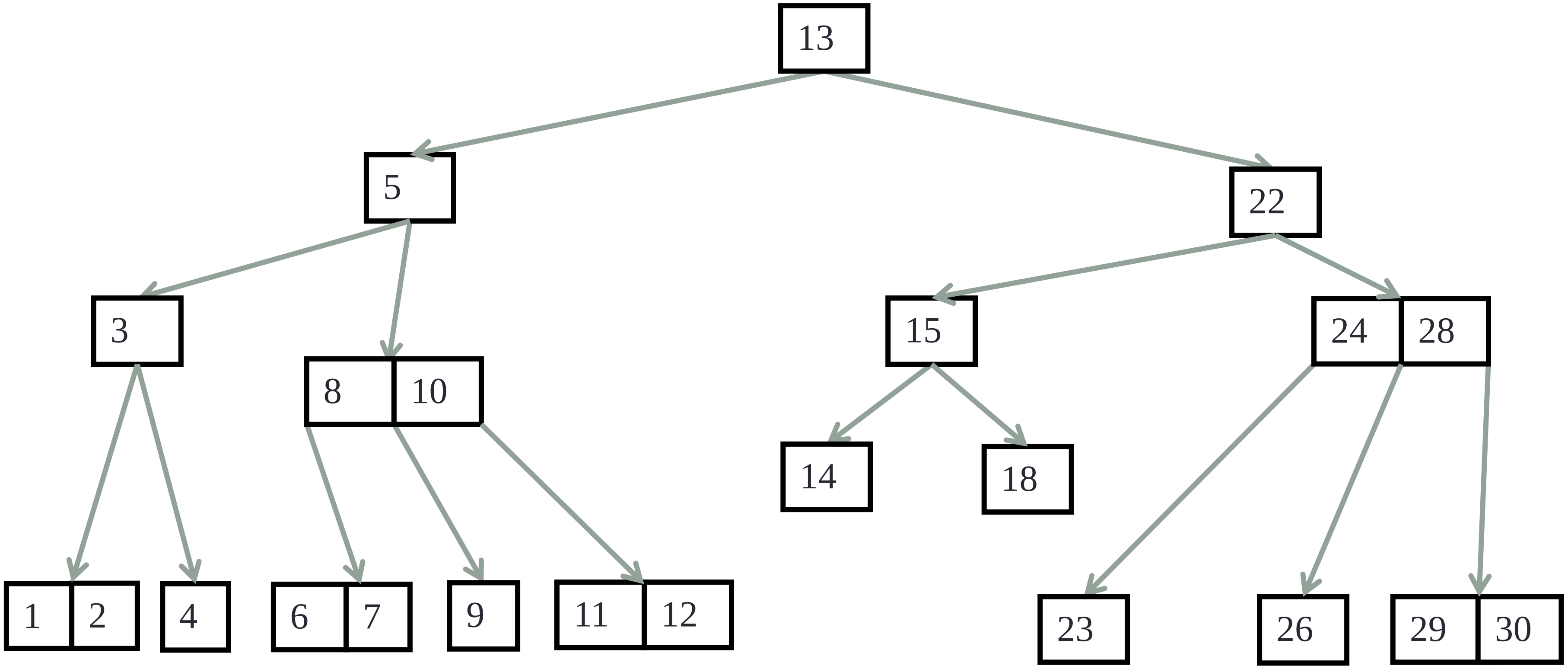
Deletion in B-tree(Cont'd)

- Delete key in given order: 21, 25, 20, 23, 18
- When deletion is done from internal node, swap it with predecessor and delete predecessor



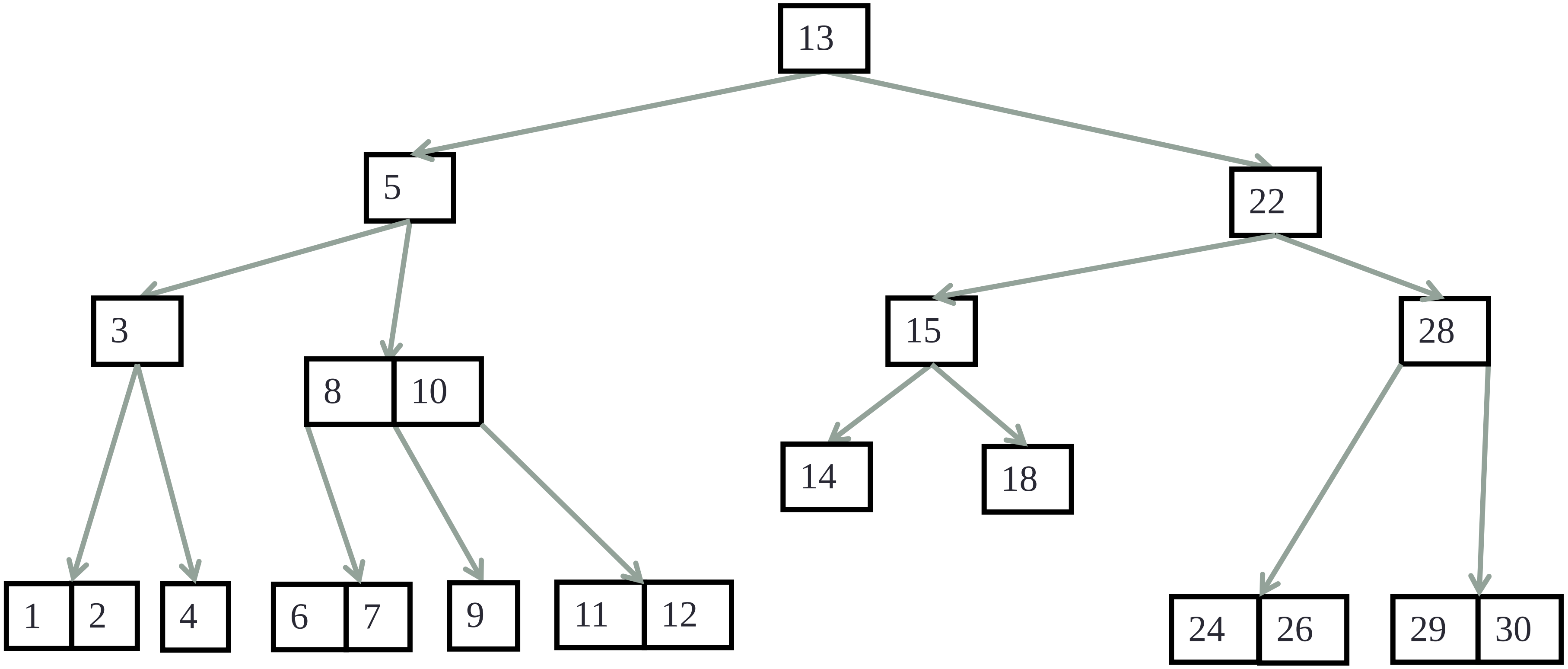
Deletion in B-tree(Cont’d)

- Delete key in given order: 21, 25, 20, 23, 18
- If deleting a key violates minimum key requirement, borrow key from adjacent sibling(s)



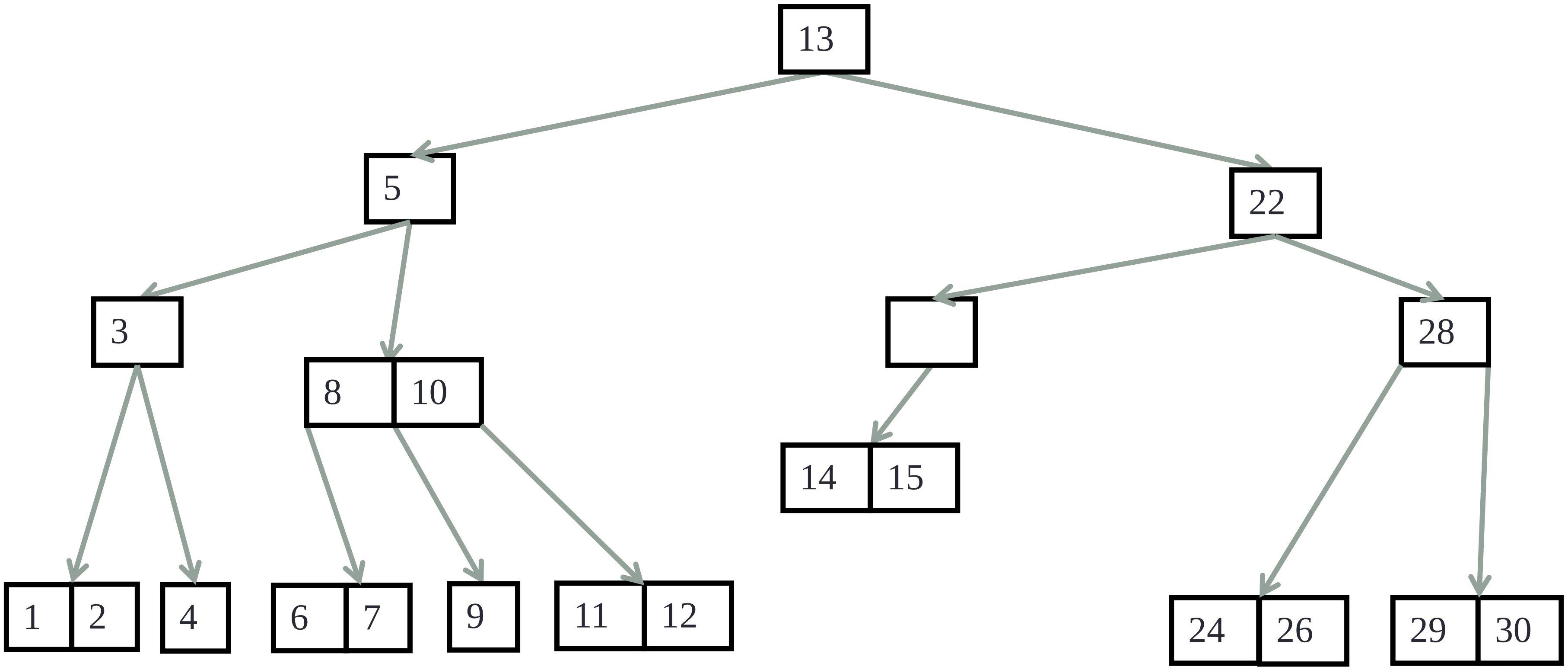
Deletion in B-tree(Cont'd)

- Delete key in given order: 21, 25, 20, 23, 18
- If adjacent sibling(s) can not lend key, merge siblings and bring down the key in parent node separating them



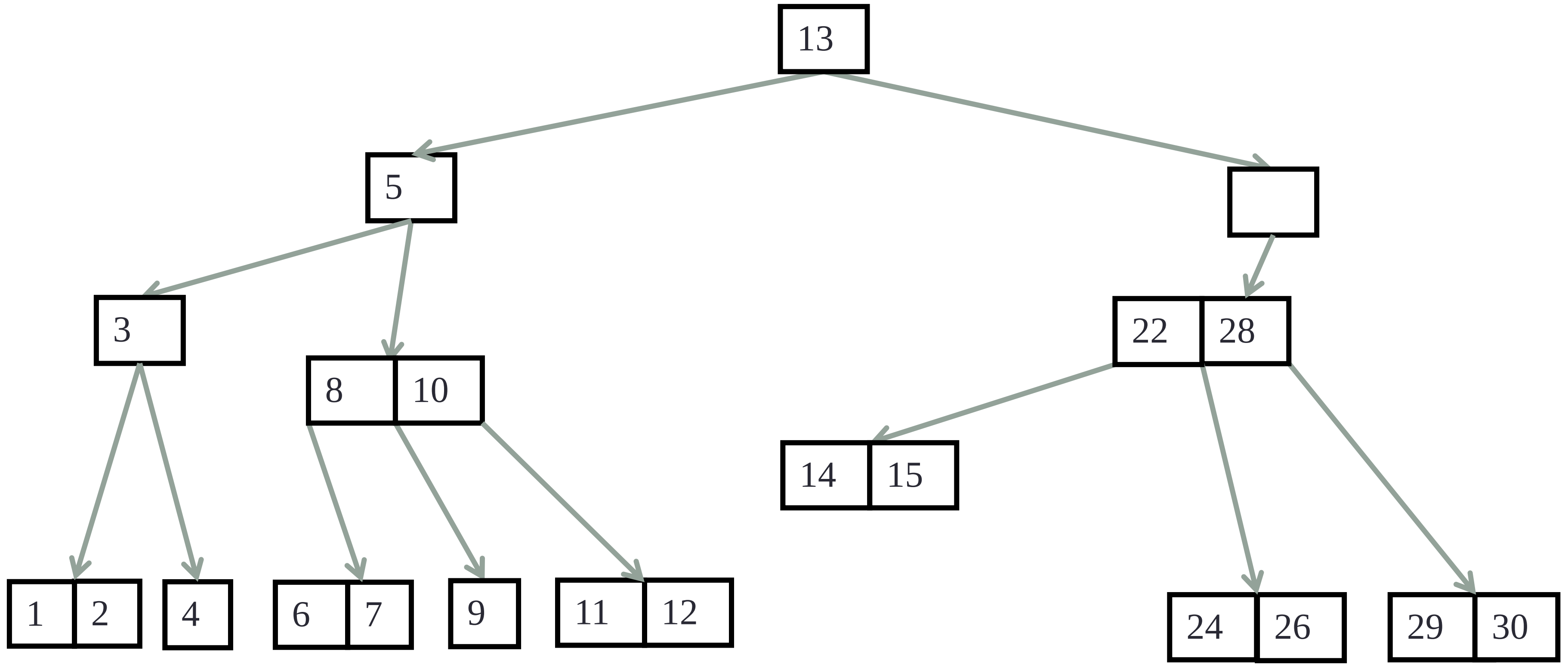
Deletion in B-tree(Cont'd)

- Delete key in given order: 21, 25, 20, 23, 18
- Moving down key from parent node might result in parent node violating minimum key requirement. Perform deletion algo at parent.



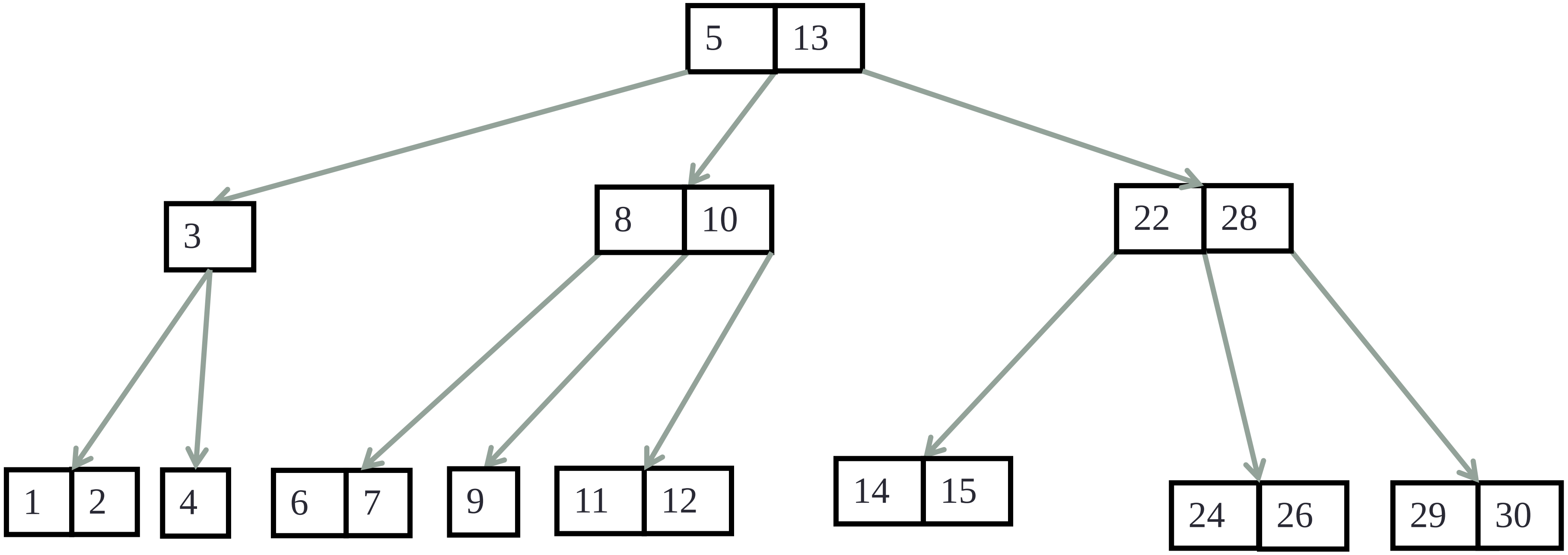
Deletion in B-tree(Cont'd)

- Delete key in given order: 21, 25, 20, 23, 18
- Moving down key from parent node might result in parent node violating minimum key requirement. Perform deletion algo at parent.



Deletion in B-tree(Cont'd)

- Delete key in given order: 21, 25, 20, 23, 18
- Moving down key from parent node might result in parent node violating minimum key requirement. Perform deletion algo at parent.



Deletion in B-tree : Algo

1. If deletion is from leaf node, delete key and check if minimum key requirement is satisfied or not. If not then,
   1. Try to borrow key from left sibling if left sibling has extra key
   2. Try to borrow key from right sibling if right sibling has extra key
   3. If not possible, merge left(right if left is null) sibling and key in parent node separating them
2. If deletion is to be done from internal node, replace with predecessor and delete predecessor key if possible. Else replace with successor and delete successor key. Then repeat step 1 for deletion from leaf
3. Keep repeating steps for ancestors of node where deletion happened.

Deletion in B-tree : Analysis

- Deleting the key from B-tree : O(log n)
- Borrowing key from adjacent sibling : O(1)
- Merging of siblings and bringing it down : O(1)
- In worst case, borrowing and merging might be done for all the nodes up to root. Therefore, might take O(log n)
- Thus, time complexity of deletion : O(log n)

Exercise

Given 5-way B-tree created by these data (last exercise):

3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

Add these further keys: 2, 6,12

Delete these keys: 4, 5, 7, 3, 14

B+ tree : Introduction

- A key is present in internal node as well as at leaf level
- When a node is split, middle key go to parent as well as a copy of that is also kept at right child (or left but follow one convention through out)
- Therefore, internal nodes only has key and pointers to children. Data is stored in leaves only
- B+ tree also allows sequential access at leaf level i.e. we can access leaves left to right as well
- Advantage:
  - Internal nodes can have more keys thus reducing the height
  - Sequential access at leaves help accessing data in ascending order

Priority Queue

- List of elements where each element also has priority associated
- Priority values not necessarily be unique
- Priority value are total ordered
- Total Order Relation, denoted by $\leq$
  - Reflexive: $K \leq K$
  - Anti-symmetric: if $K1 \leq K2$ and $K2 \leq K1$ then, $K1 = K2$
  - Transitive: if $K1 \leq K2$ and $K2 \leq K3$ then $K1 \leq K3$

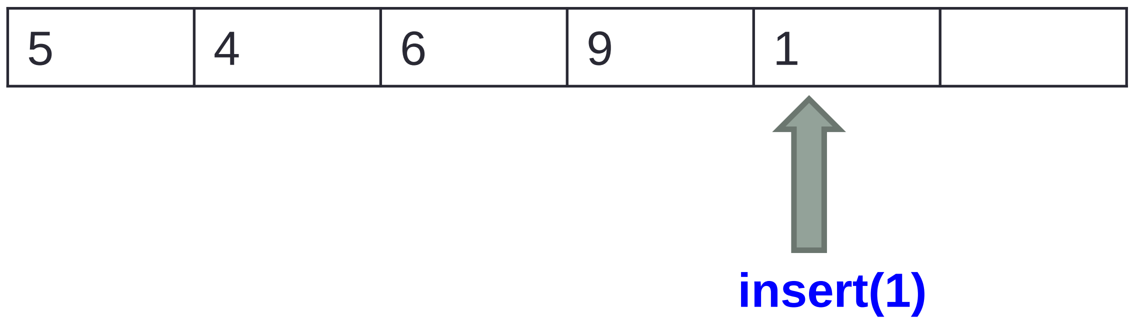
- Example: Queue of passengers at check-in counters. Allow passengers with shorter time to fly to check-in first.
- Application: job scheduling in OS etc.

Priority Queue : Operations

- Priority Queue should support operations **insert(), minimum()** and **delete-min()**.
- For our purpose, an element with lower priority value has higher priority. For instance if A and B have priority value 5 and 10 respectively then A will be deleted first from the priority queue.

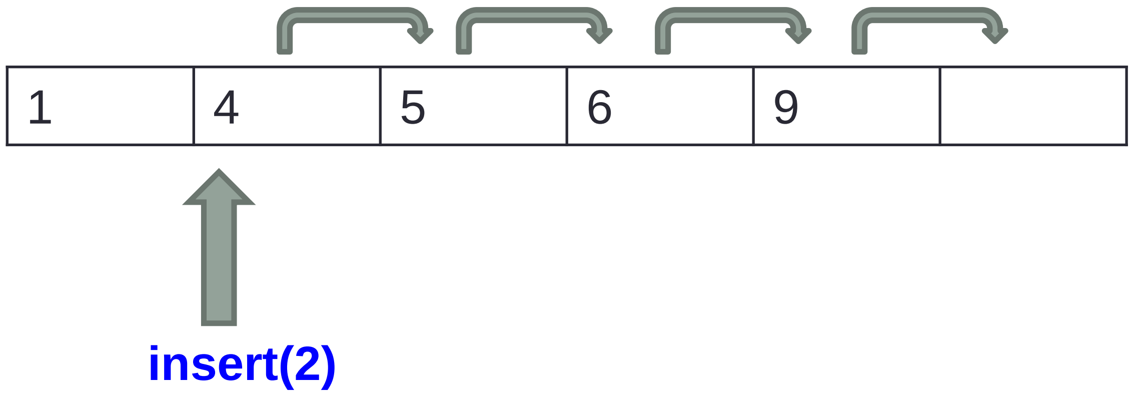
PQ with Unordered List

- Unordered list of priority values.
- Insertion happens at rear. Thus, O(1)
- To find minimum, search whole list i.e. O(n)
- To delete minimum, search the list first and then delete that element i.e. O(n)



PQ with Ordered List

- Minimum and delete minimum can be performed in O(1) time.
- However, insertion will take O(n) time



(Binary) Heap

- A **complete** binary tree where every node is greater than or equal to its respective parent.
- Because it is complete binary tree, all leaves at last level are filled from left most side.
- The minimum element is at root
- Binary heap is also called min-heap(minimum element at root) .
- Binary heap with maximum element at root is called max-heap.
- Efficiently represented using arrays.

Height of Heap

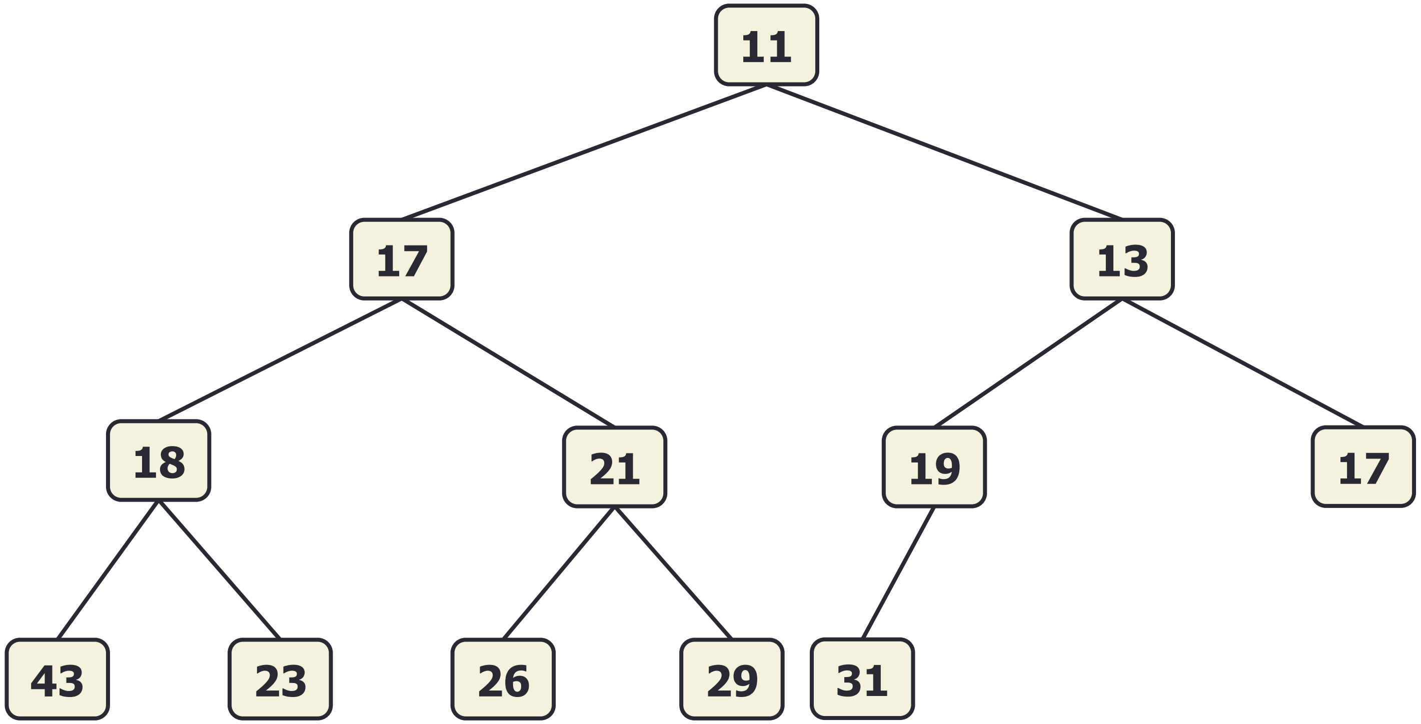
- Suppose heap with **n** nodes has height **h**.
- A perfect binary tree with height h has $2^{h+1} - 1$ nodes and height h-1 has $2^h - 1$ nodes.
- As heap is complete binary tree, therefore

$$2^h - 1 < n \le 2^{h+1} - 1$$

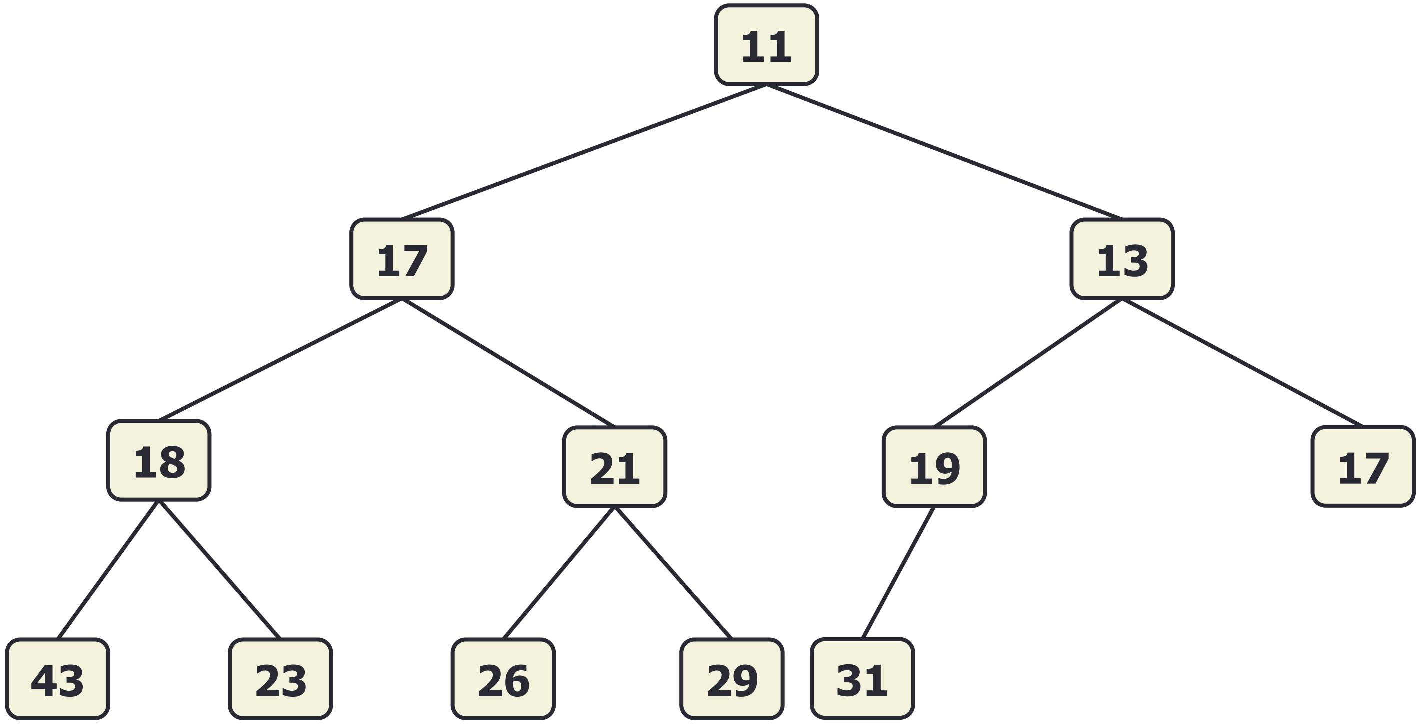
- $h = \text{floor}(\log_2 n)$

Example of Min-Heap

11	17	13	18	21	19	17	43	23	26	29	31
0	1	2	3	4	5	6	7	8	9	10	11

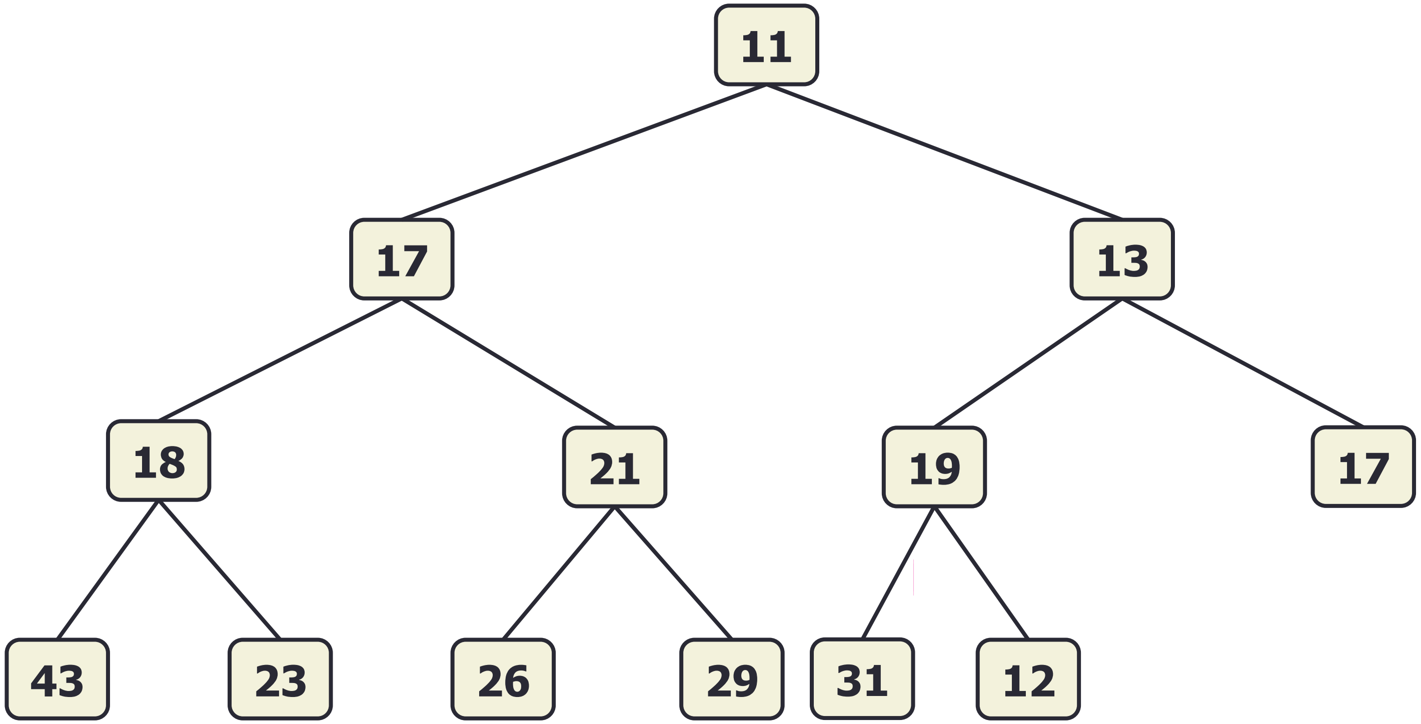


Insertion in Heap : Example



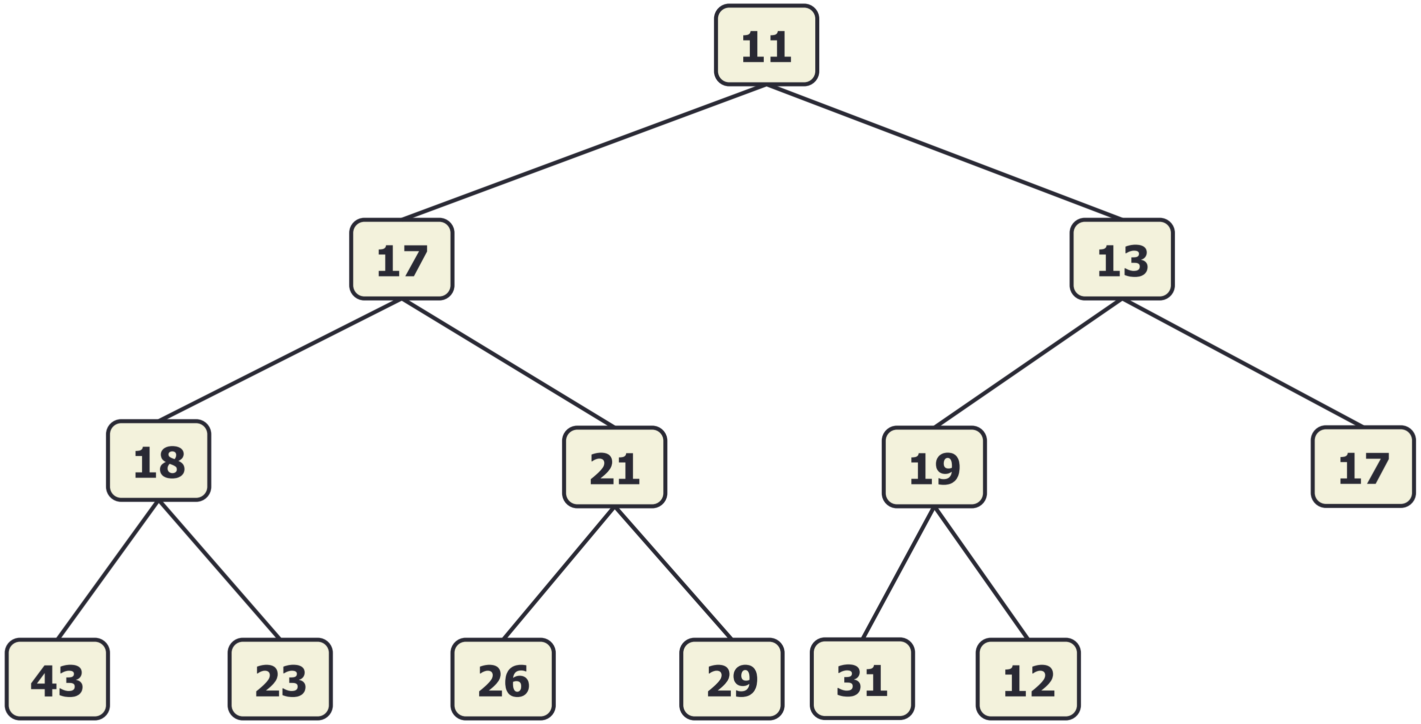
Insertion in Heap : Example

- Insert 12



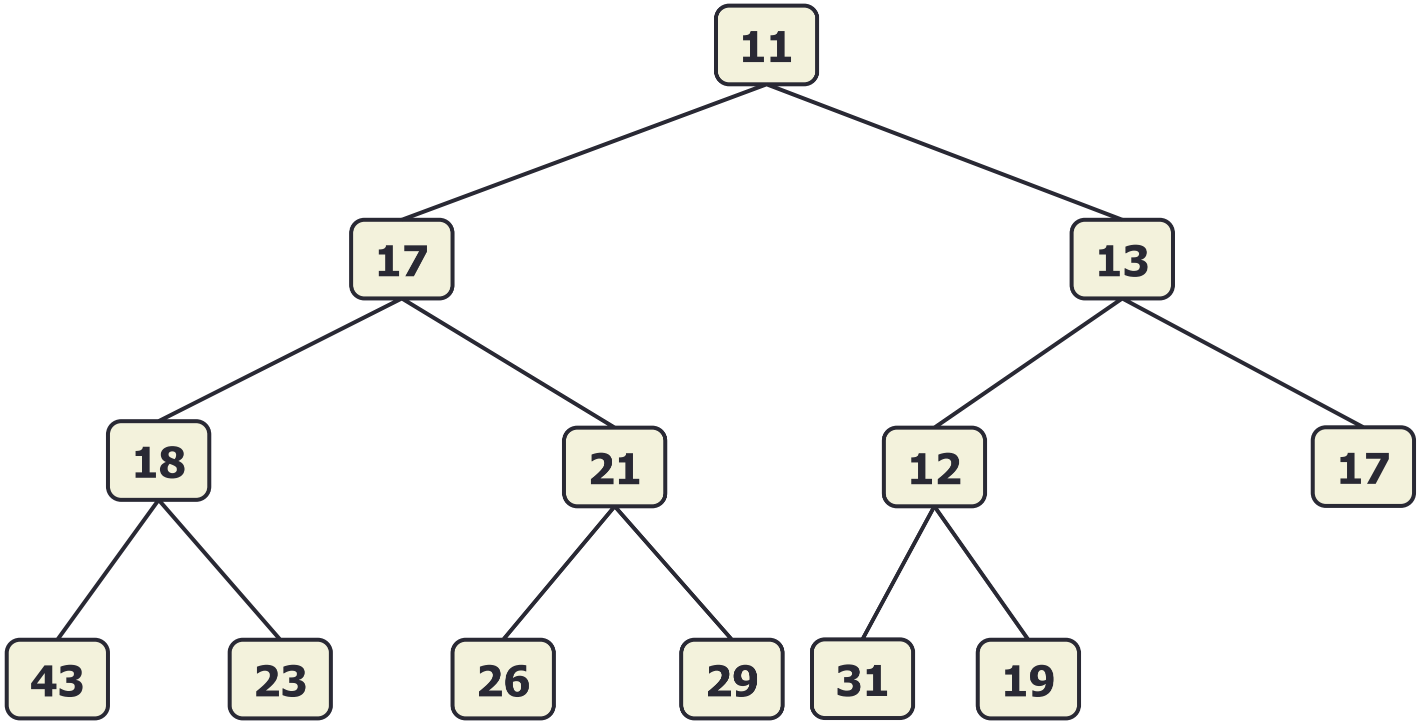
Insertion in Heap : Example

- Insert 12
- Compare 12 with its parent 19. As 19>12, swap values



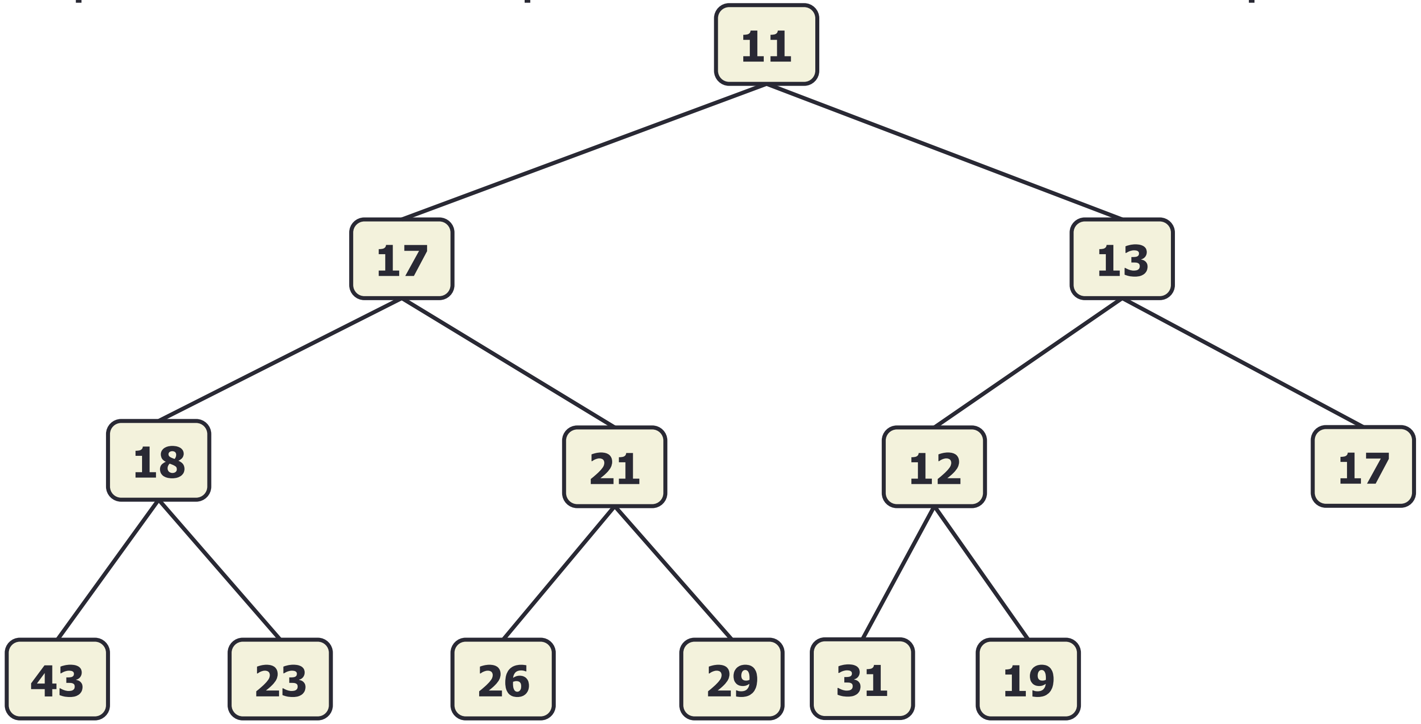
Insertion in Heap : Example

- Insert 12
- Compare 12 with its parent 19. As 19>12, swap values



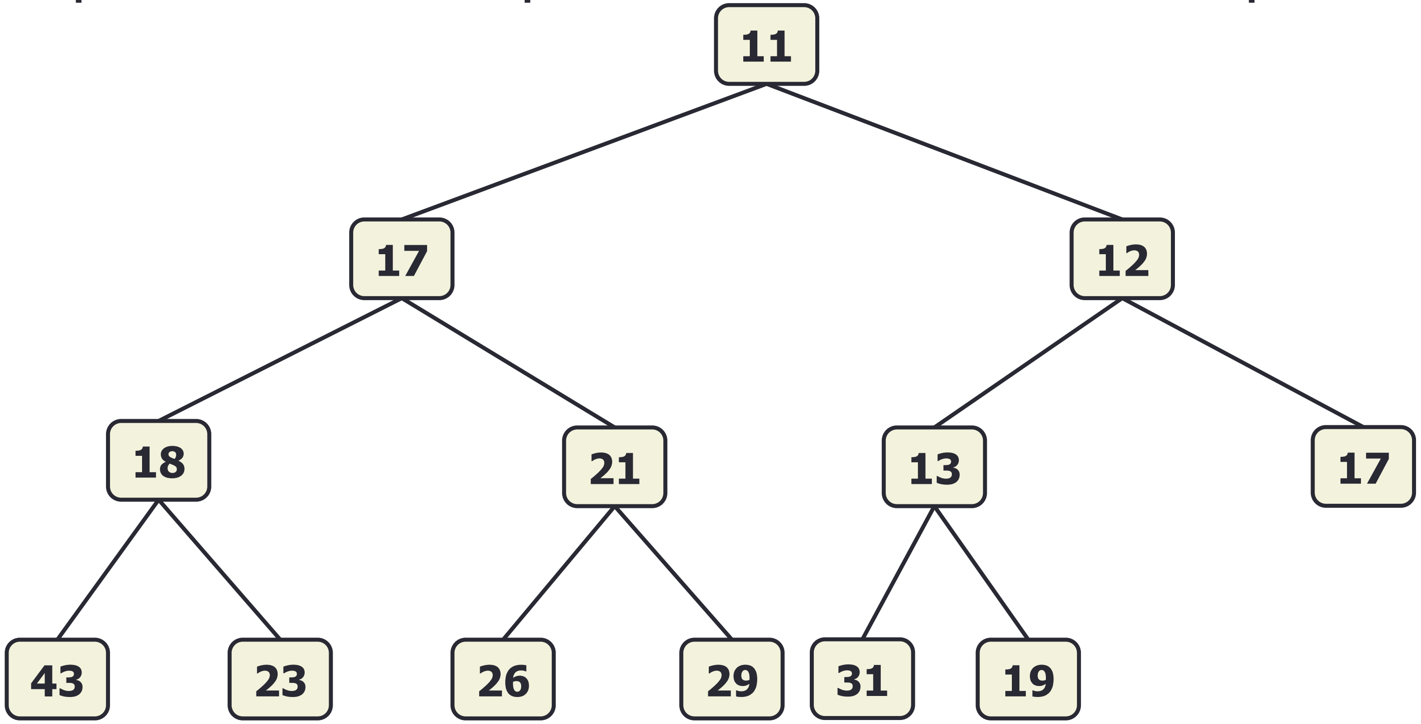
Insertion in Heap : Example

- Insert 12
- Compare 12 with its parent 19. As 19>12, swap values
- Compare 12 with its parent 13. As 13>12, swap values



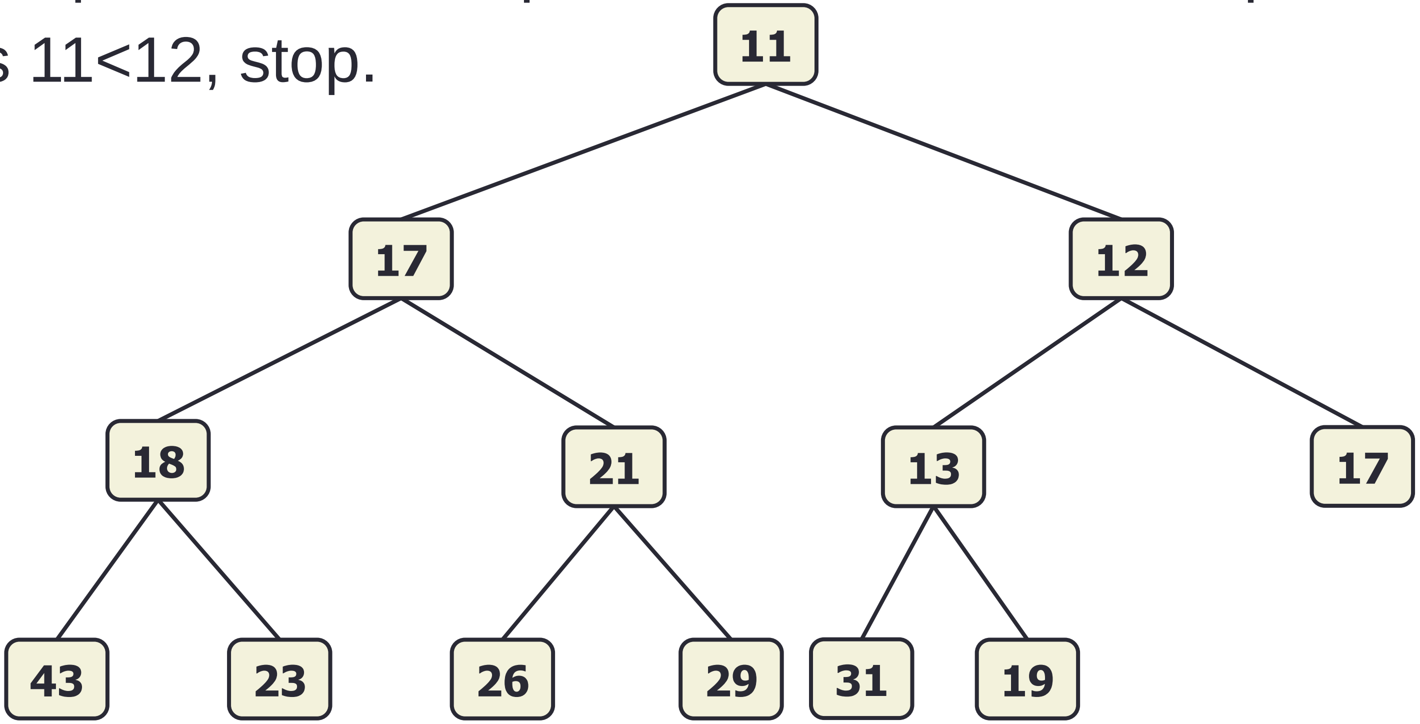
Insertion in Heap : Example

- Insert 12
- Compare 12 with its parent 19. As 19>12, swap values
- Compare 12 with its parent 13. As 13>12, swap values



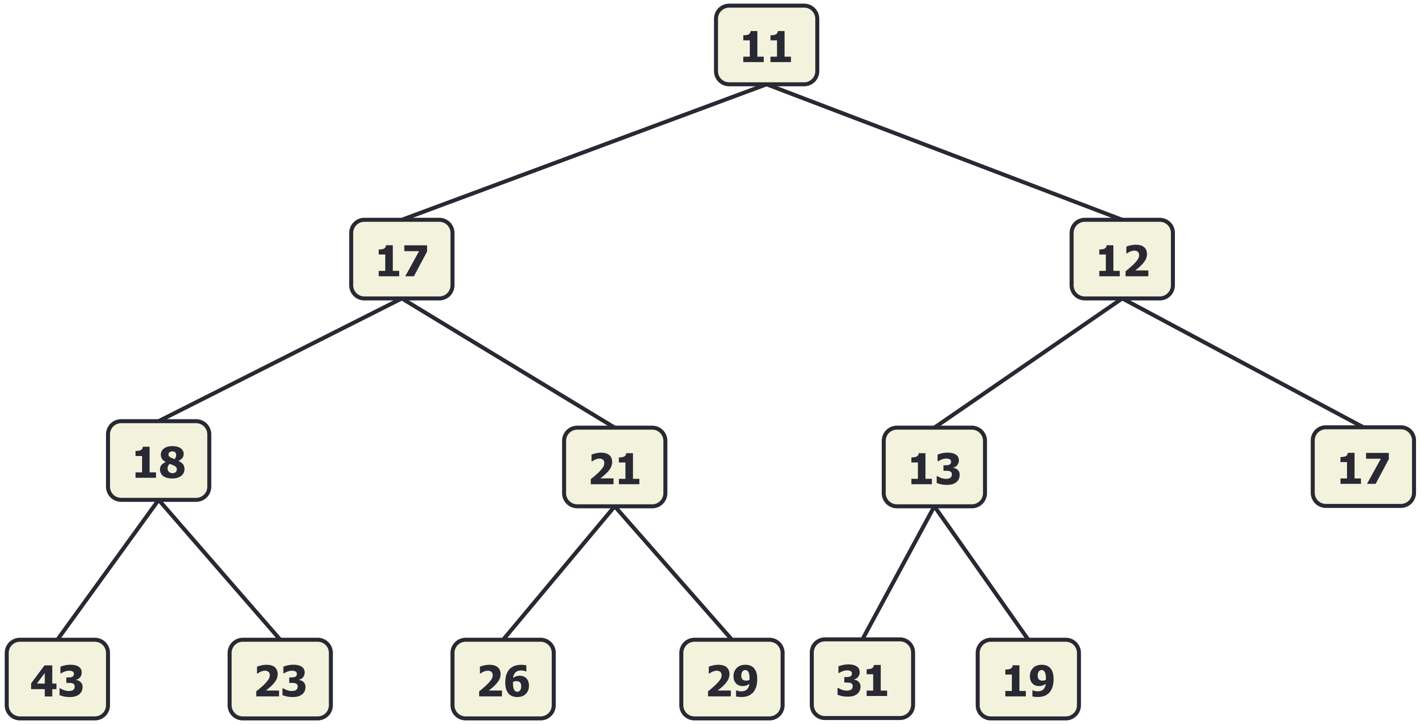
Insertion in Heap : Example

- Insert 12
- Compare 12 with its parent 19. As 19>12, swap values
- Compare 12 with its parent 13. As 13>12, swap values
- As 11<12, stop.



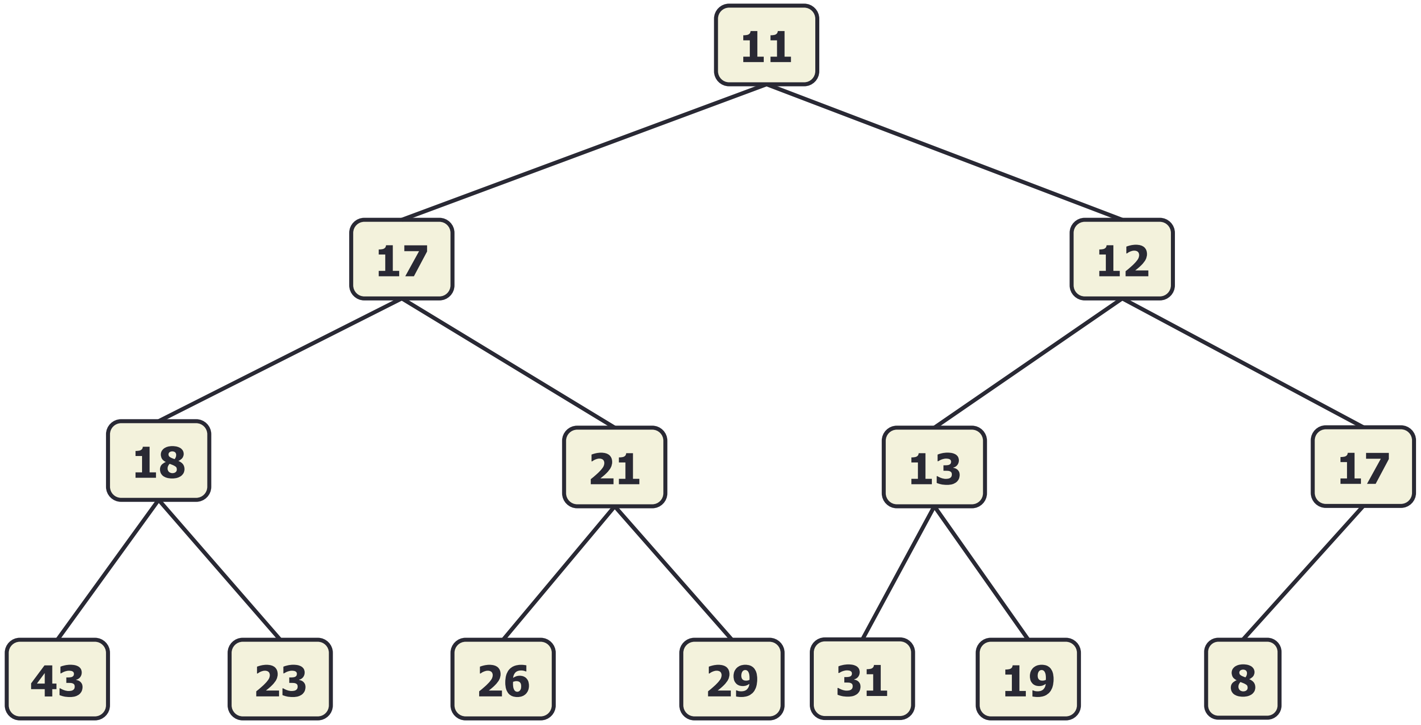
Insertion in Heap : Example

- insert 12
- insert 8



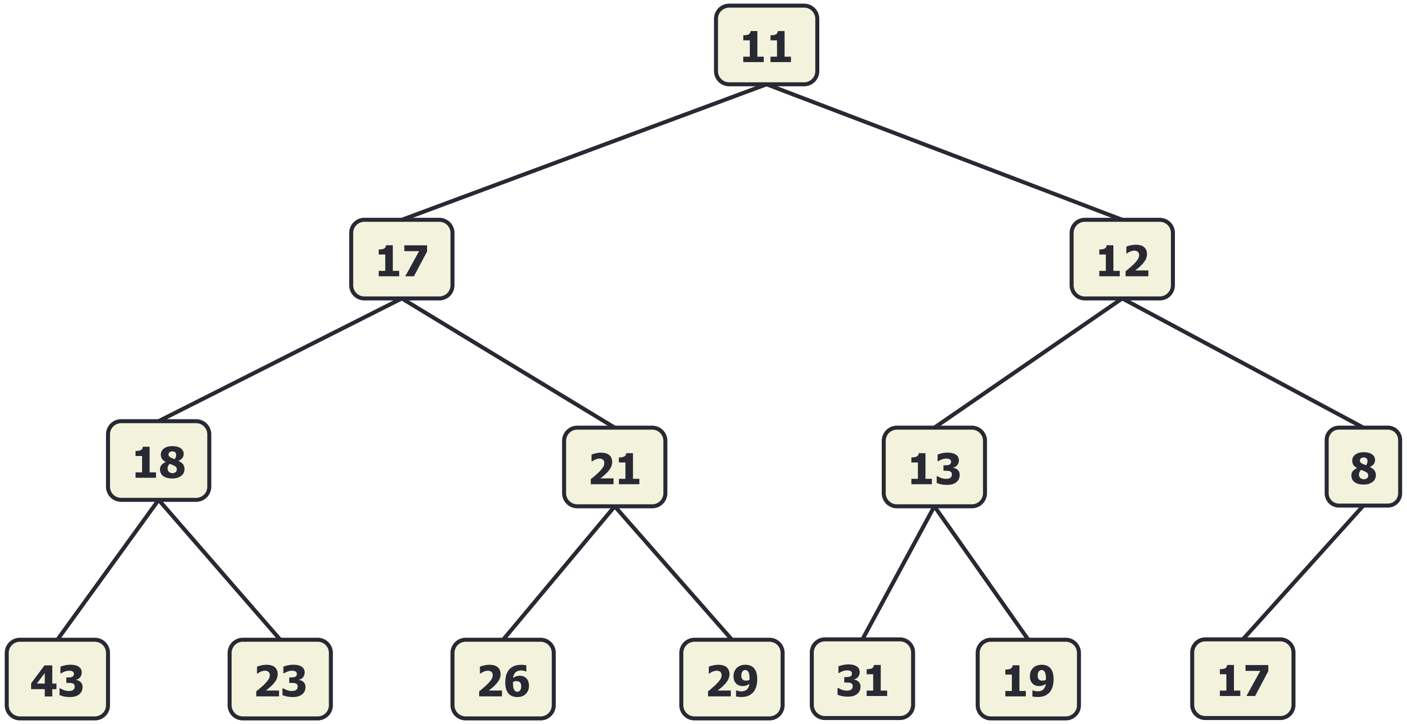
Insertion in Heap : Example

- insert 12
- insert 8



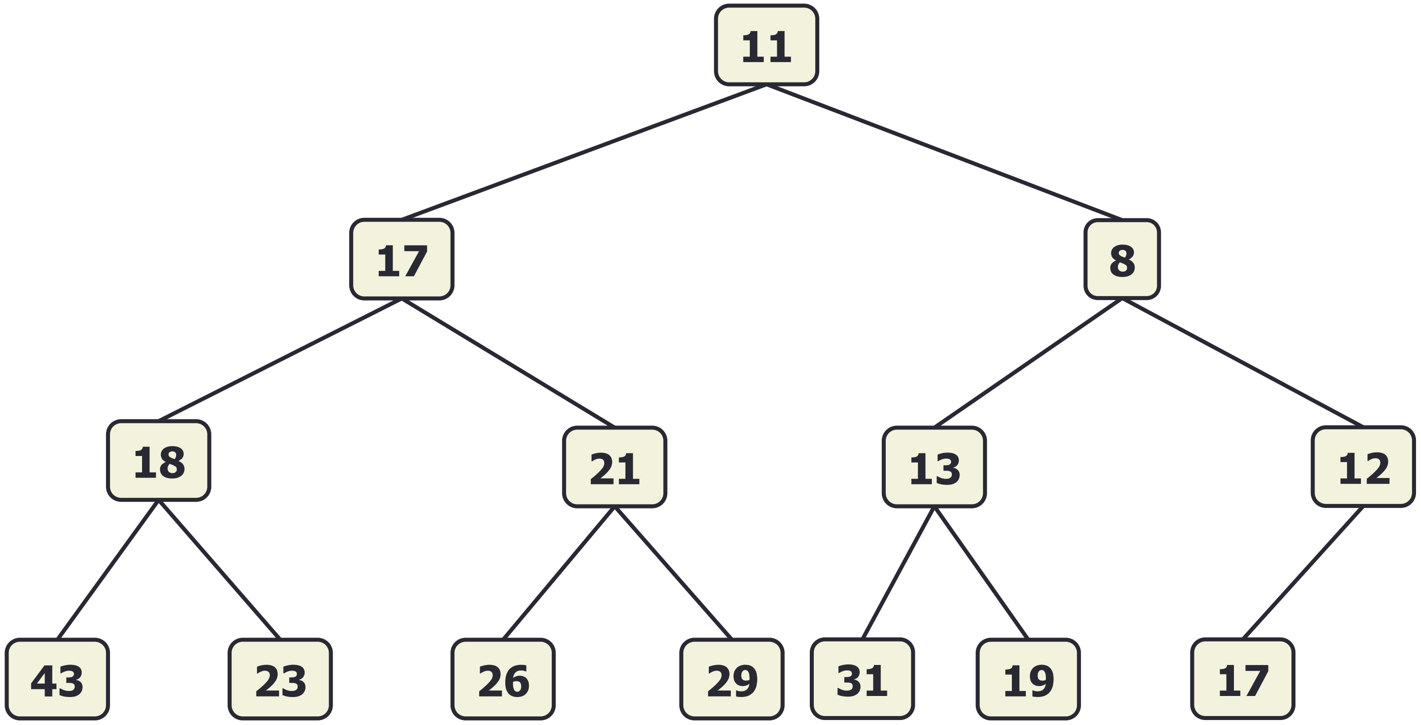
Insertion in Heap : Example

- insert 12
- insert 8



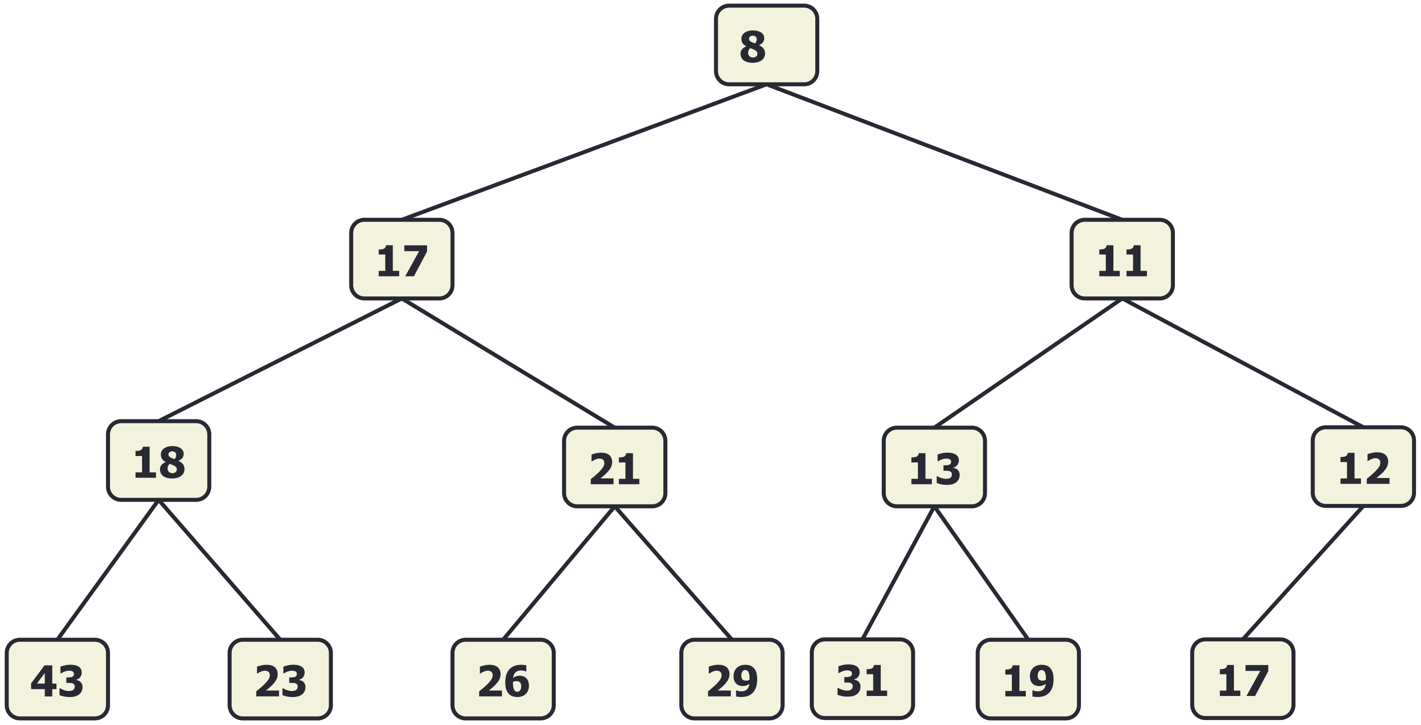
Insertion in Heap : Example

- insert 12
- insert 8



Insertion in Heap : Example

- insert 12
- insert 8



Insertion in Heap : Algo

1. Insert node at last level at leftmost available position.
2. Subsequently compare newly inserted node with its parent. If parent is greater than new node, swap two values.
3. Repeat step 2 for the parent upto root.

Insertion in Heap : Analysis

- Adding a node at right place: O(1)
- Swapping two values: O(1)
- Number of swap operation: $O(\log_2 n)$ (height of heap)
- Total time complexity = $O(\log_2 n)$

Deleting Minimum

- As minimum element sits at root, finding minimum will take constant O(1) time.
- To delete minimum, we must reorganize tree after deleting the root.
- To reorganize, we “**Heapify**” the heap.

Heapify

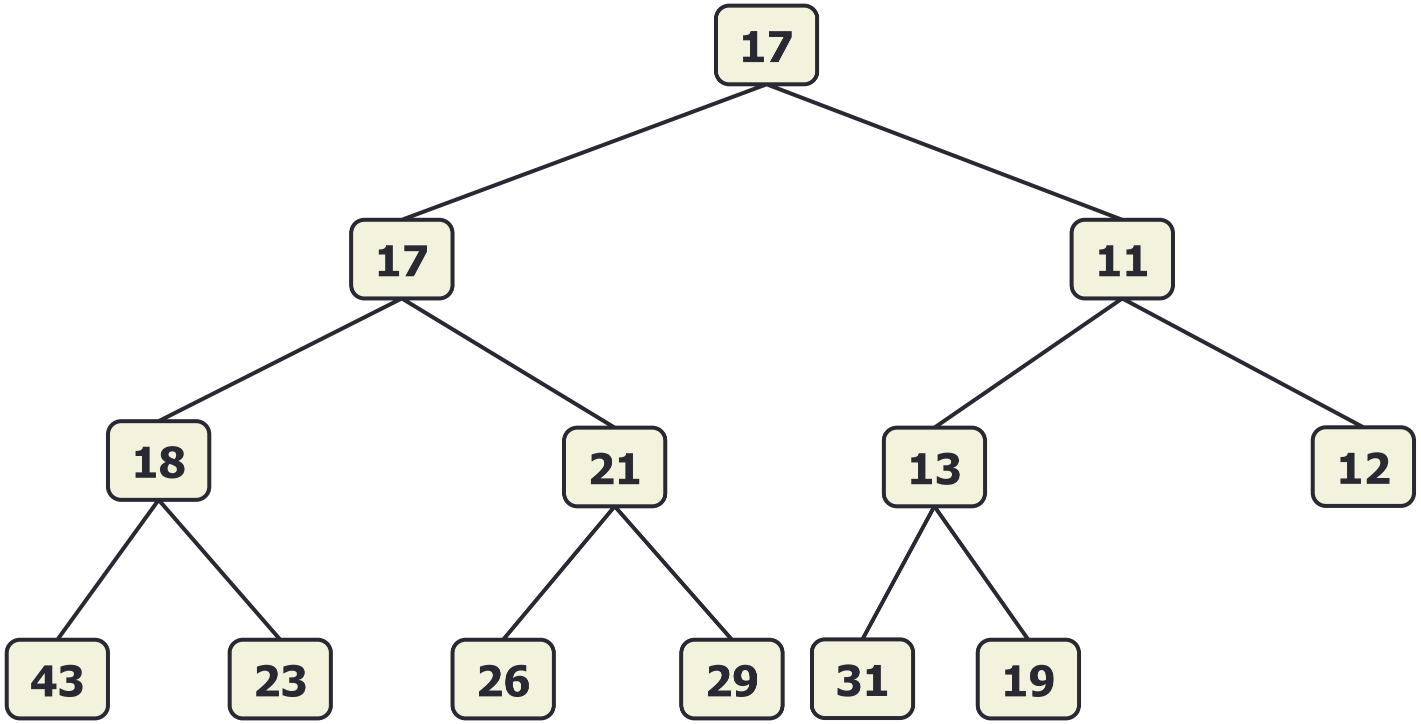
- Precondition to Heapify:
  - Let **i** be the index in array A
  - Left and right sub-tree of tree rooted at A[i] are heaps
  - A[i] might be violating the heap property i.e. A[i] might be greater than its children
- Method Heapify make binary tree rooted at index **i** a heap by moving A[i] down in the heap

Heapify : Algo

```
Heapify(A, i) {  //i is index where heapification to be done
    while(true) {
        if (A[i] > left(i) and A[i] > right(i)) {
            then swap A[i] with min(left(i), right(i))
             if swapped with left
                 then i= index of left
             else if swapped with right
                 then i= index of right
        }
        else
              break
    }
}
```

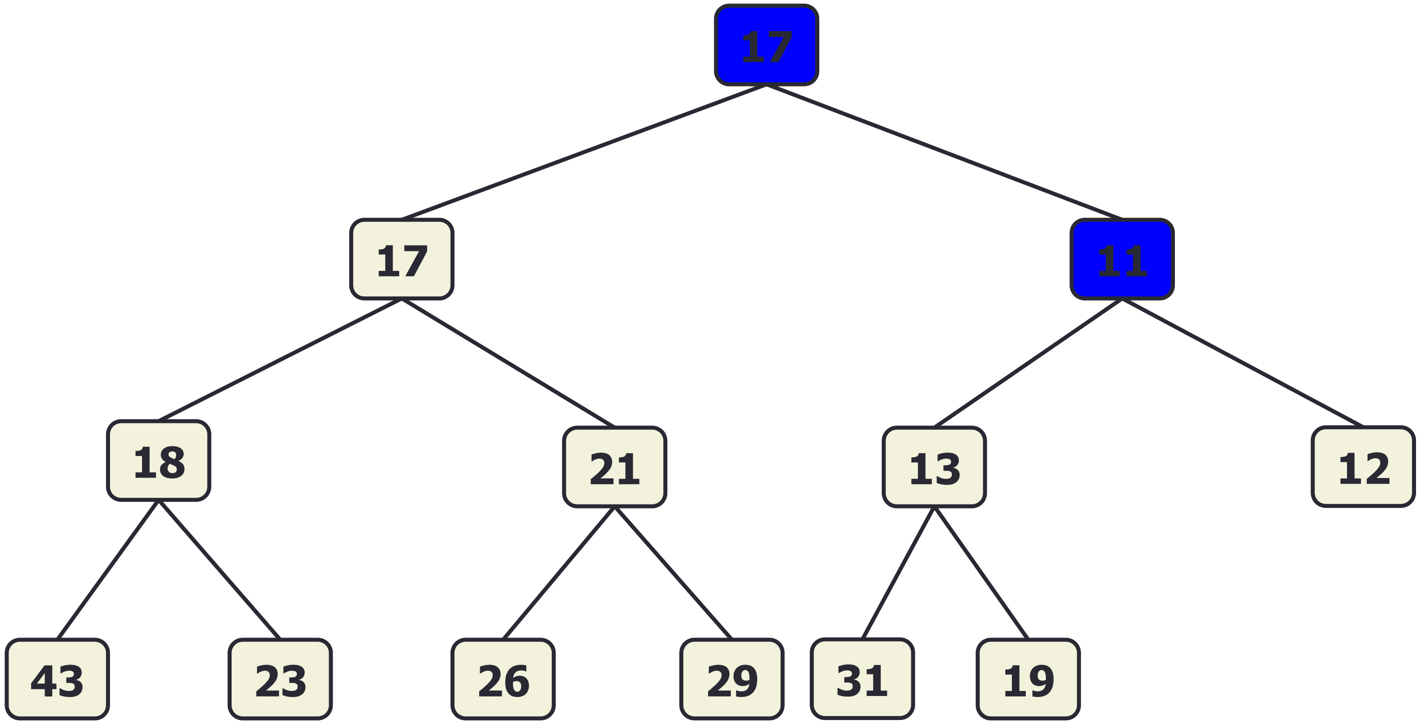
Heapify : Example

- Index i=0 is not heap. Left and right sub-tree of i are heap



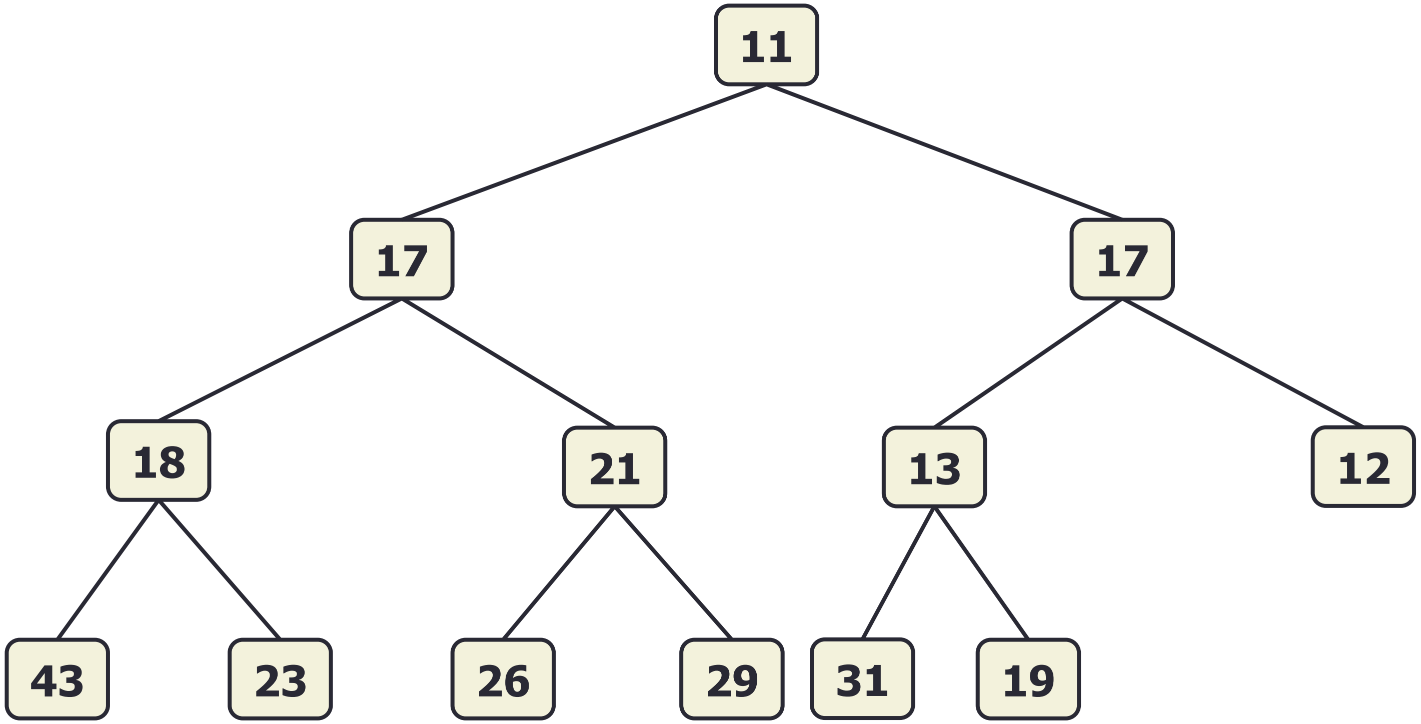
Heapify : Example

- Index i=0 is not heap. Left and right sub-tree of i are heap
- Swap min(17, 11) with A[0]



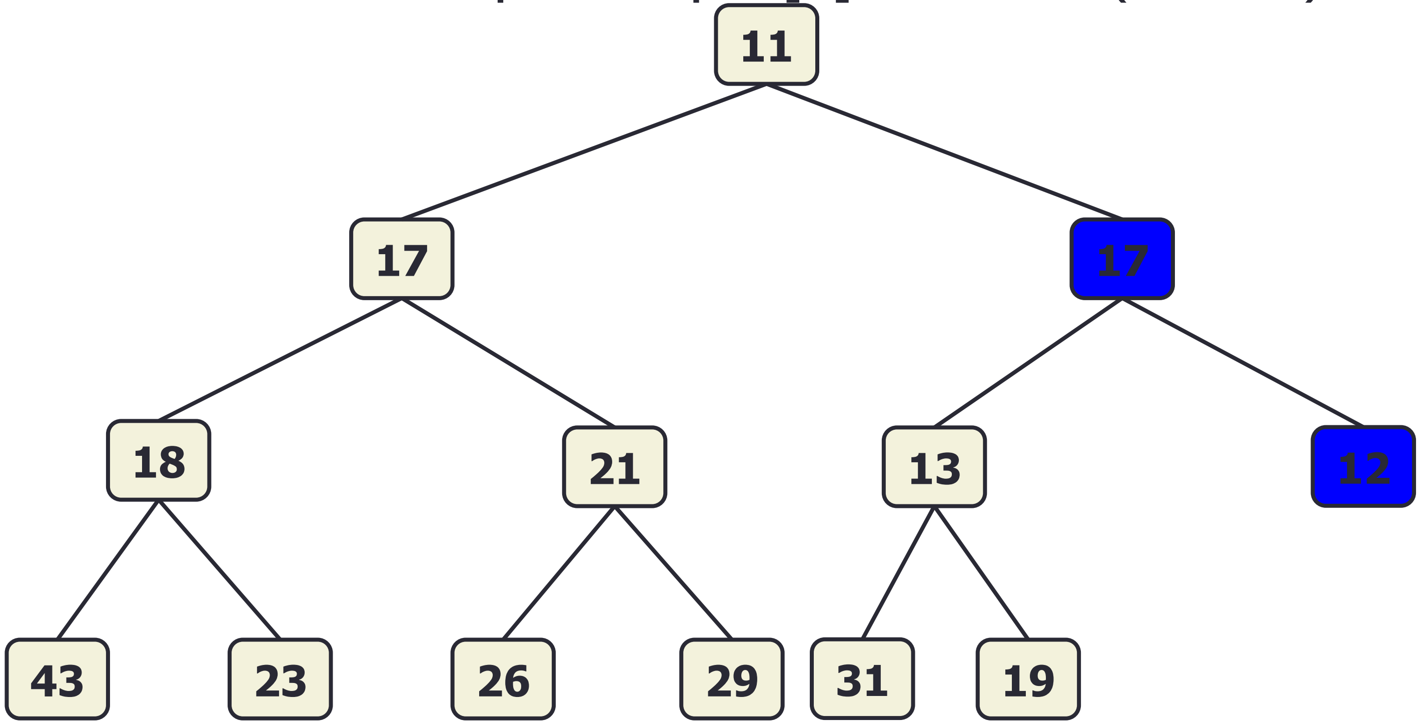
Heapify : Example

- Index i=0 is not heap. Left and right sub-tree of i are heap
- Swap A[0] with min(17, 11)



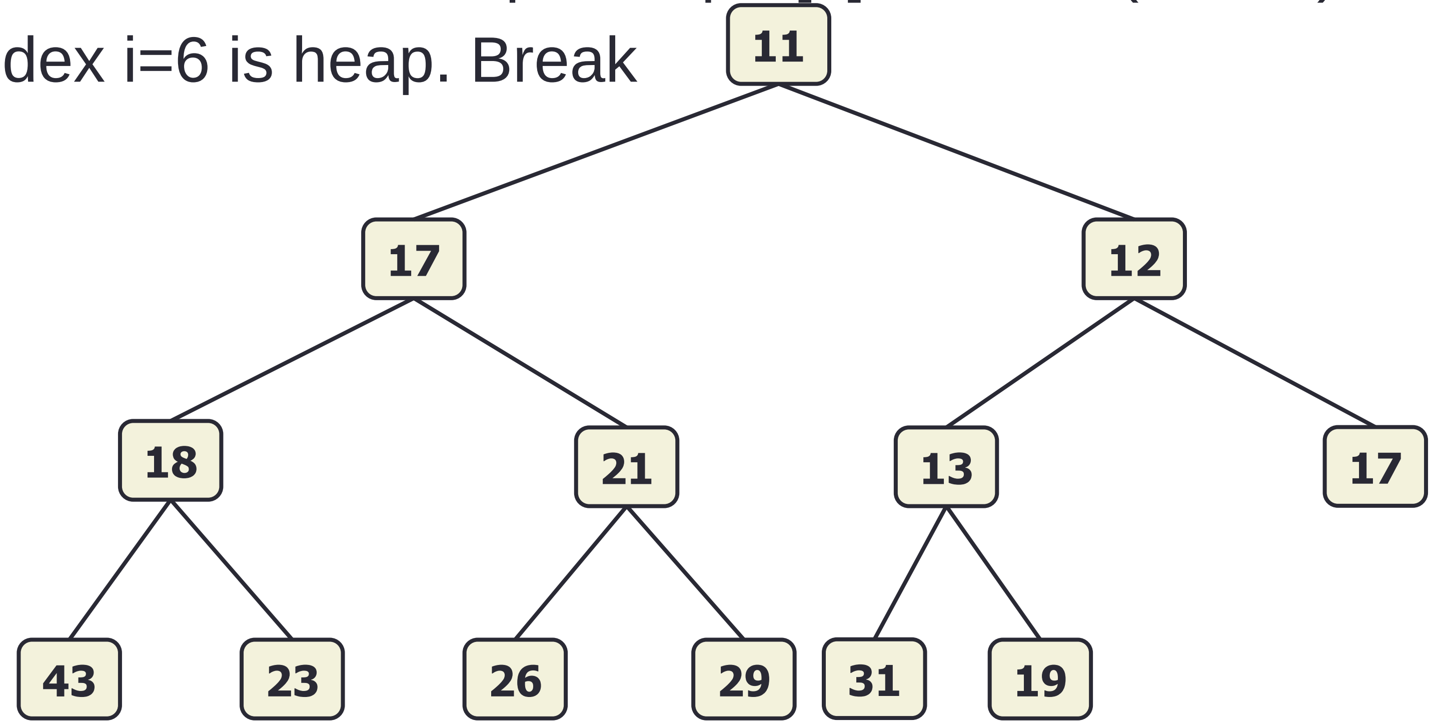
Heapify : Example

- Index i=0 is not heap. Left and right sub-tree of i are heap
- Swap A[0] with min(17, 11)
- Index i=2 is not heap. Swap A[2] with min(13, 12)



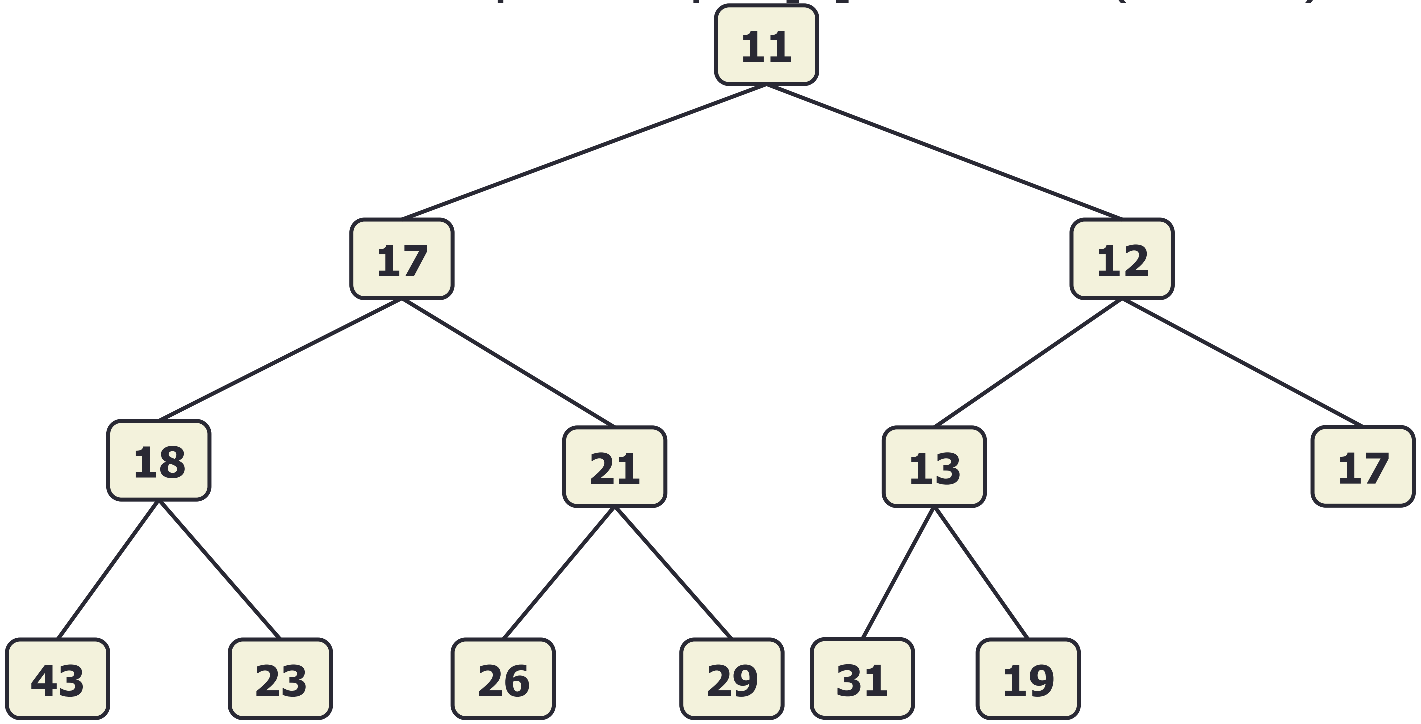
Heapify : Example

- Index i=0 is not heap. Left and right sub-tree of i are heap
- Swap A[0] with min(17, 11)
- Index i=2 is not heap. Swap A[2] with min(13, 12)
- Index i=6 is heap. Break



Heapify : Example

- Index i=0 is not heap. Left and right sub-tree of i are heap
- Swap A[0] with min(17, 11)
- Index i=2 is not heap. Swap A[2] with min(13, 12)



Delete Minimum : Algo

1. Delete the minimum node from root
2. Replace root with rightmost node at last level
3. Heapify root i.e. Heapify(a, 0)

Delete Minimum : Analysis

- Deleting the root and replacing with rightmost node at last level: O(1)
- Performing heapification once: O(1)
- In worst case, heapify will run equal to height of heap times. Thus, $O(\log_2 n)$ times
- Thus, total time complexity $O(\log_2 n)$

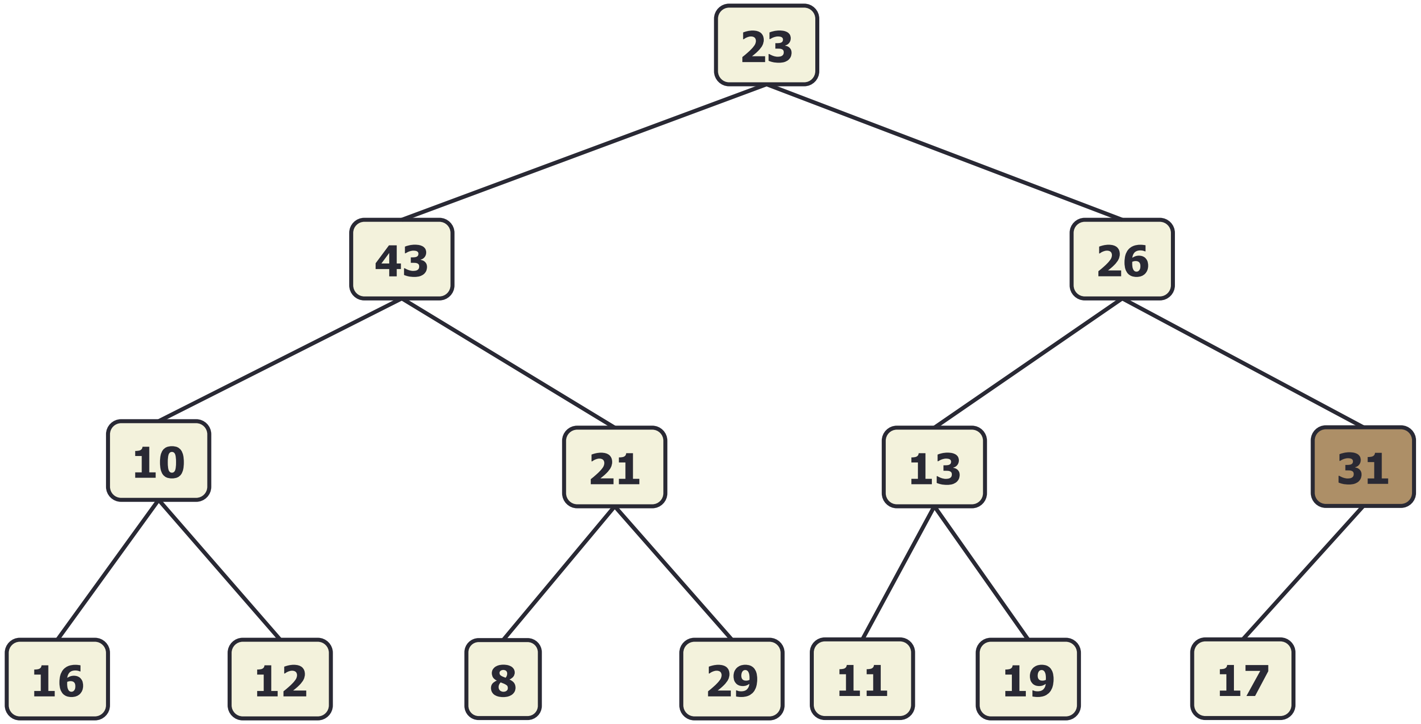
Build a Heap

- If we are given an array of n elements, we can build heap in O(n).

```
Build-Heap(A,n)
        for i=floor(n/2)-1 to 0
                run Heapify(A,i)
```

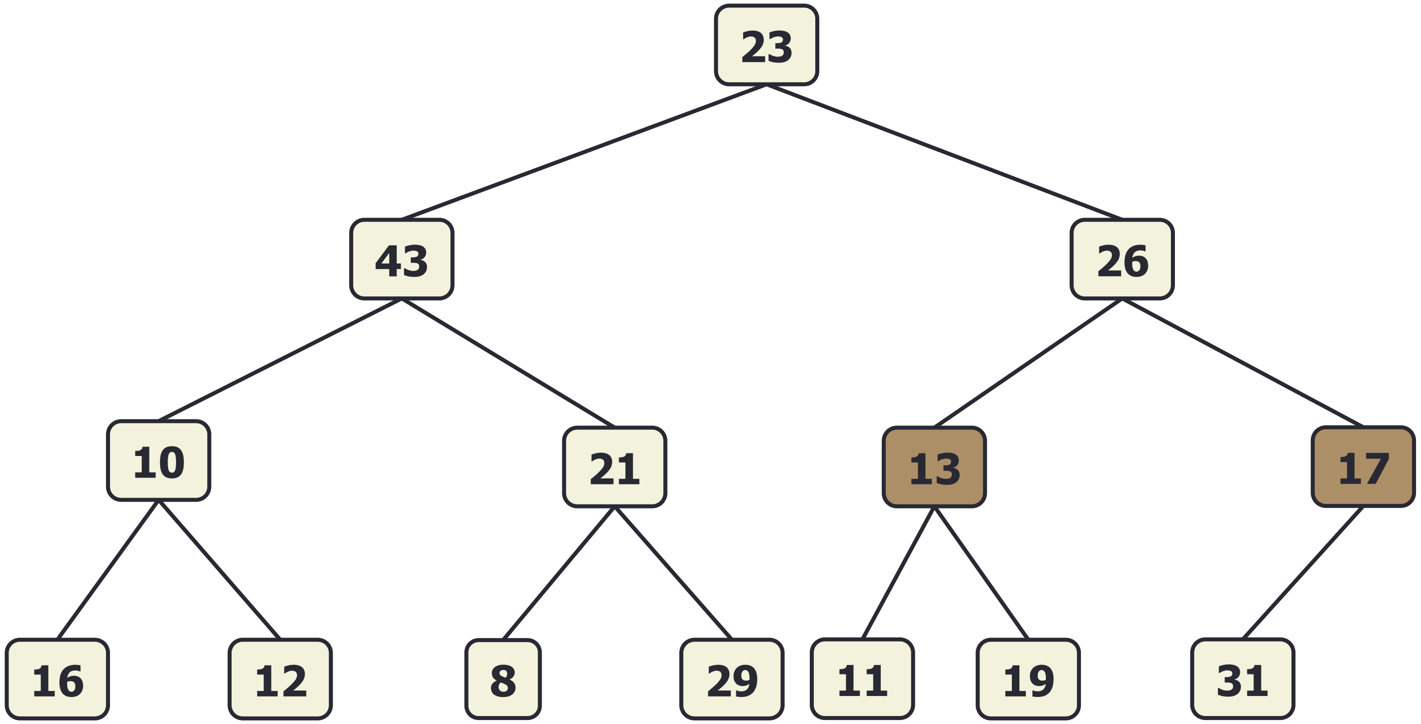
Build a Heap : Example

- Heapify (A,6)



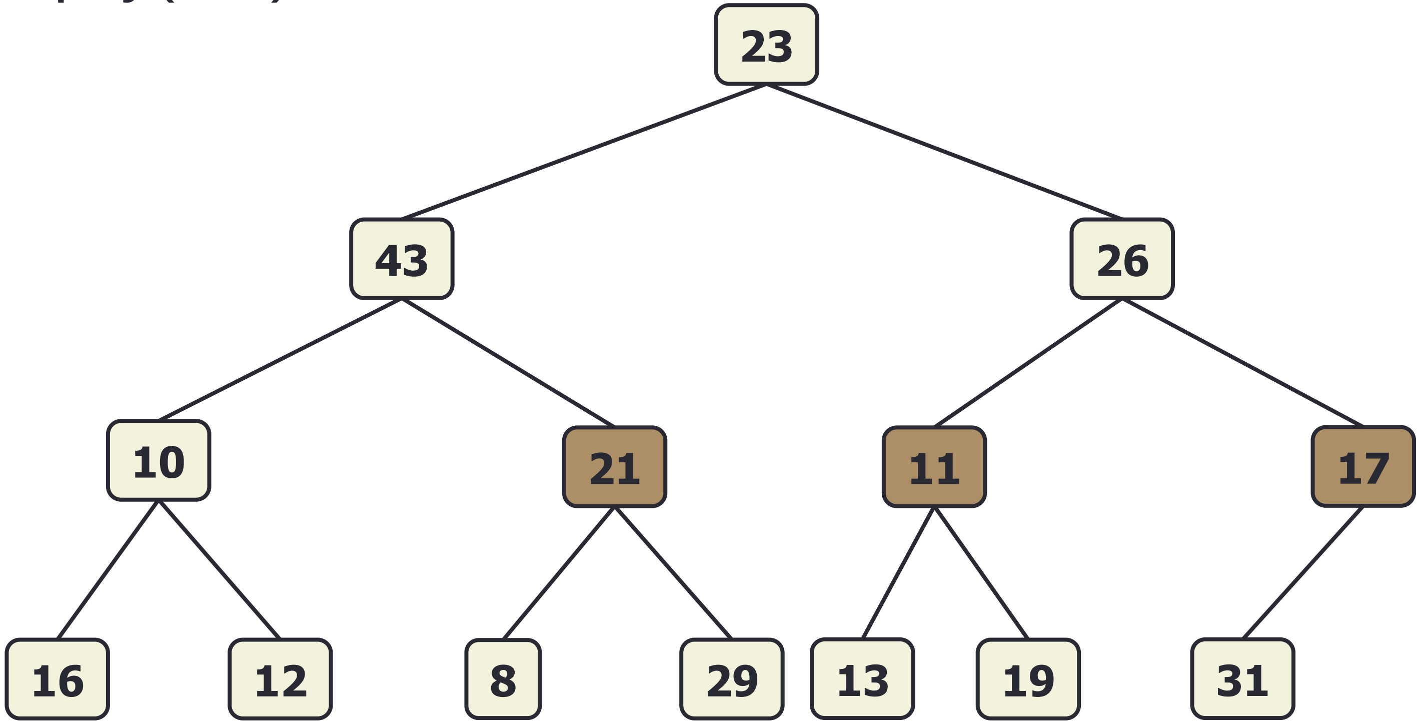
Build a Heap : Example

- Heapify(A,6)
- Heapify(A,5)



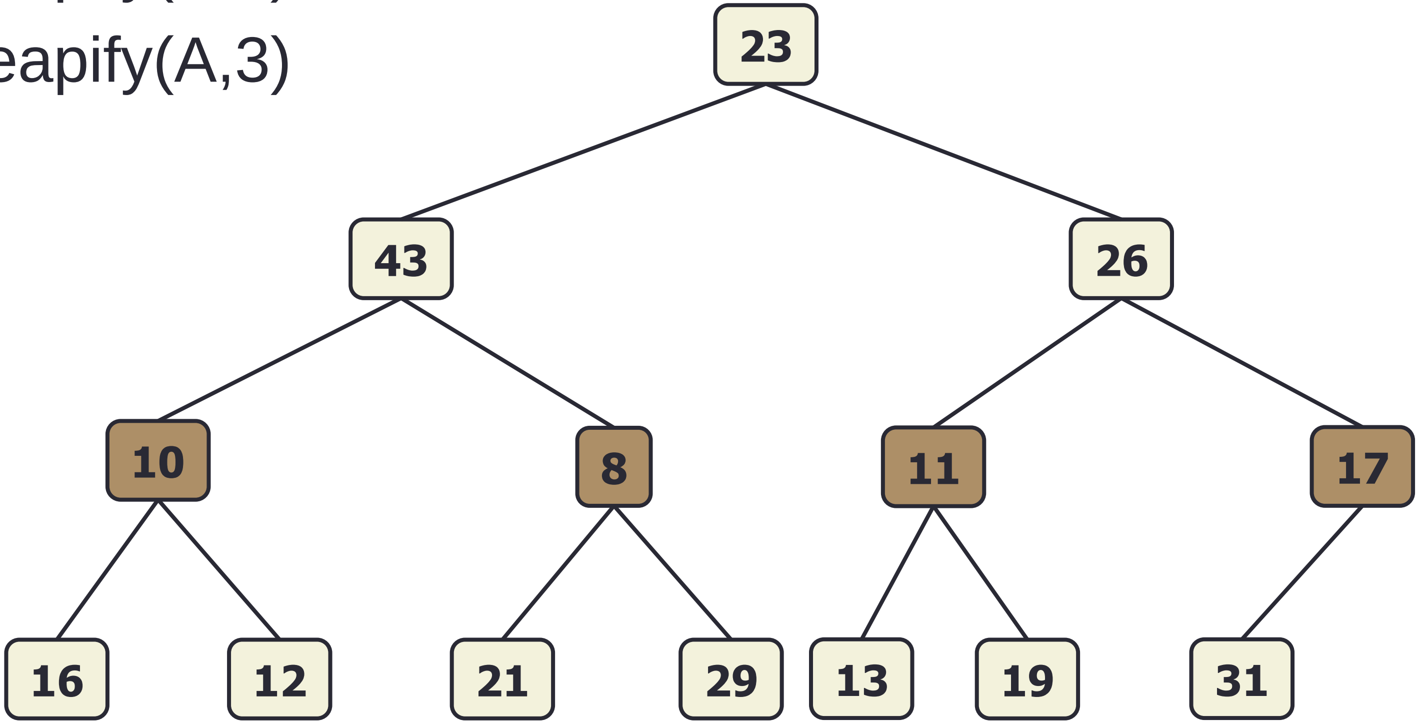
Build a Heap : Example

- Heapify(A,6)
- Heapify(A,5)
- Heapify(A,4)



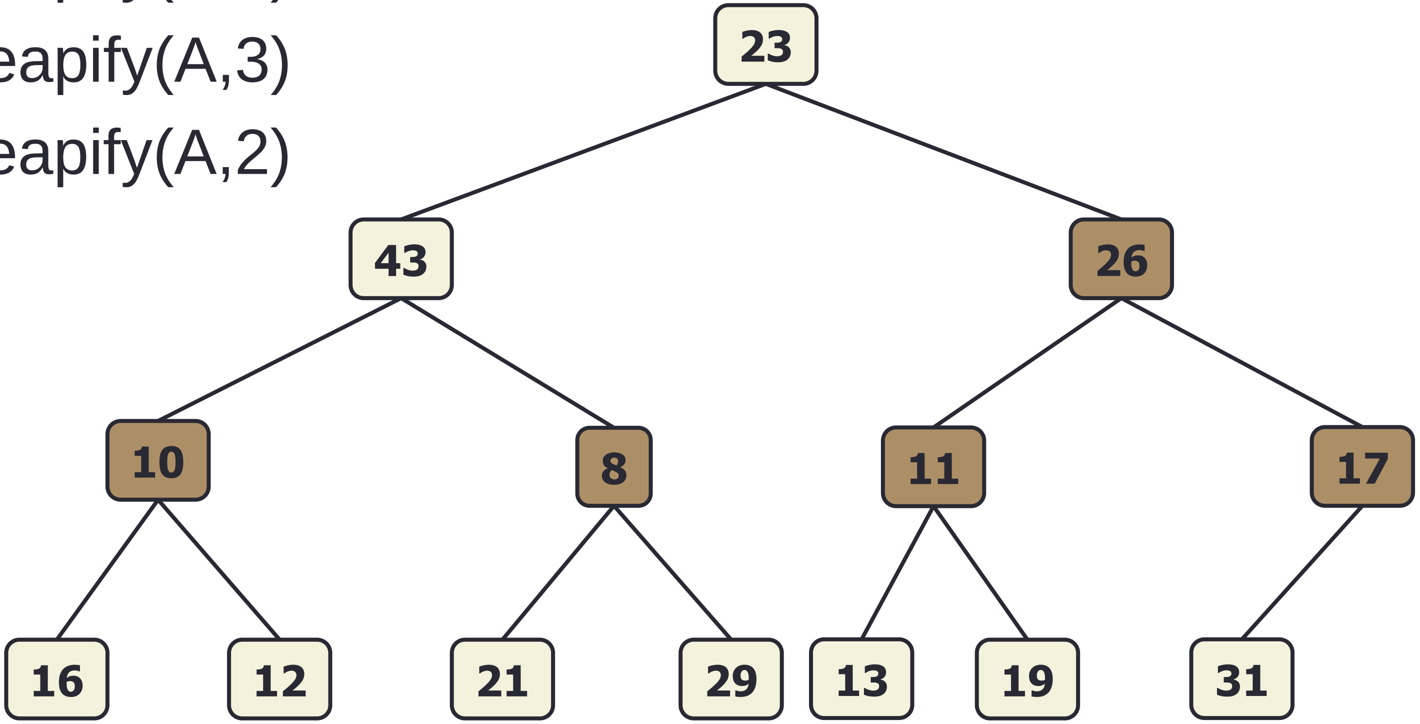
Build a Heap : Example

- Heapify(A,6)
- Heapify(A,5)
- Heapify(A,4)
- Heapify(A,3)



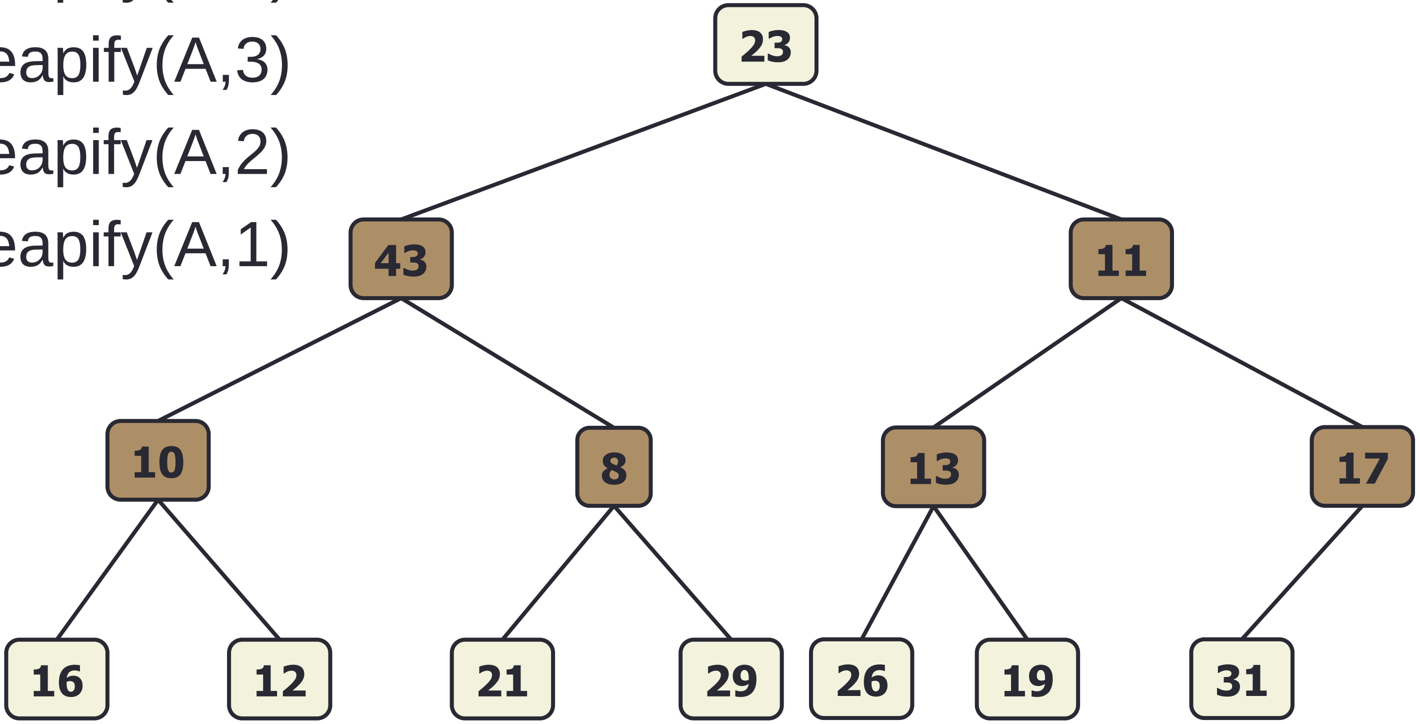
Build a Heap : Example

- Heapify(A,6)
- Heapify(A,5)
- Heapify(A,4)
- Heapify(A,3)
- Heapify(A,2)



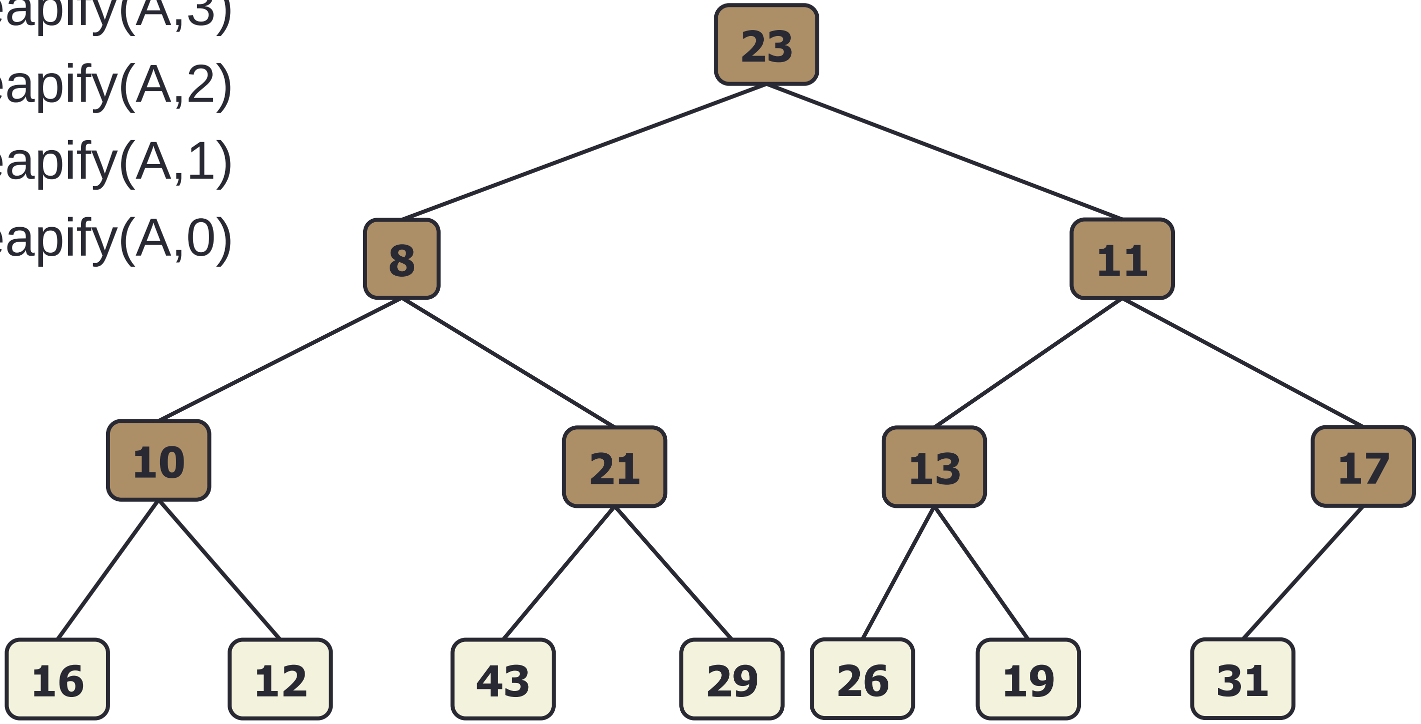
Build a Heap : Example

- Heapify(A,6)
- Heapify(A,5)
- Heapify(A,4)
- Heapify(A,3)
- Heapify(A,2)
- Heapify(A,1)



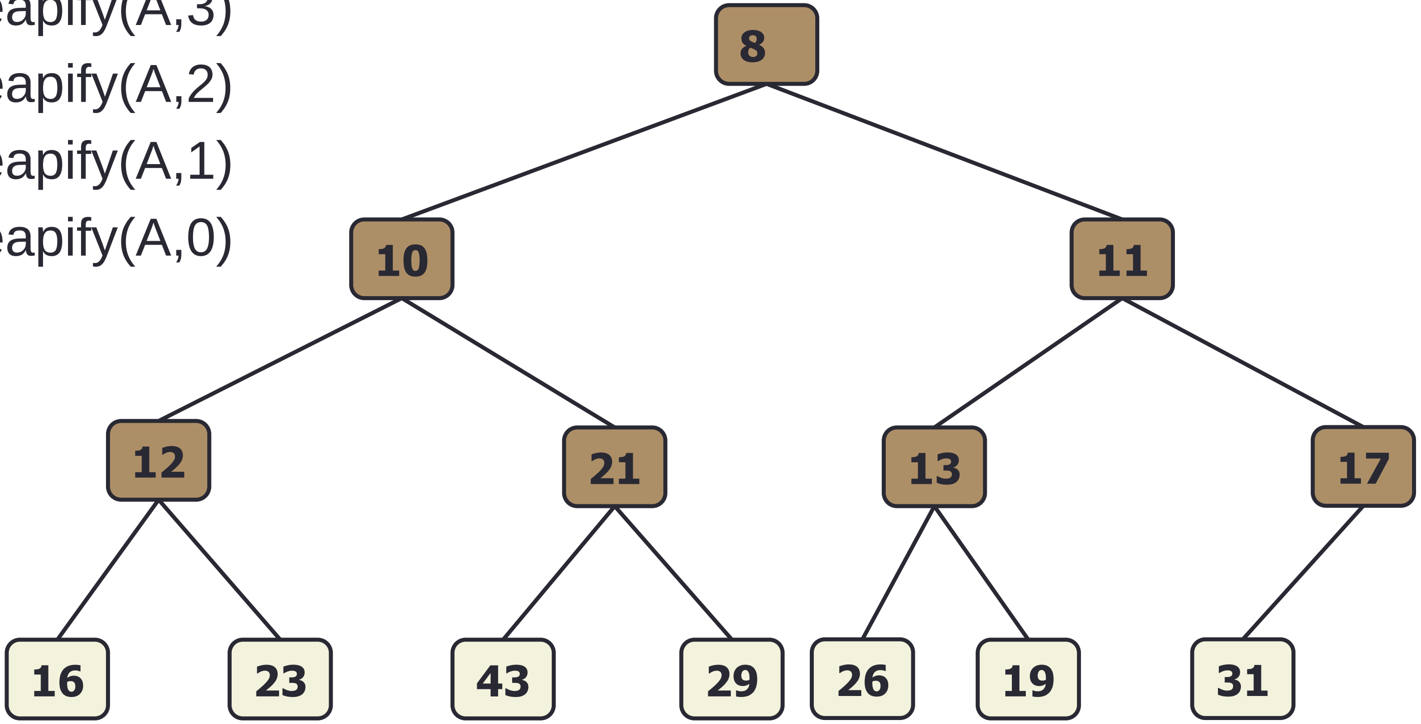
Build a Heap : Example

- Heapify(A,6)
- Heapify(A,5)
- Heapify(A,4)
- Heapify(A,3)
- Heapify(A,2)
- Heapify(A,1)
- Heapify(A,0)



Build a Heap : Example

- Heapify(A,6)
- Heapify(A,5)
- Heapify(A,4)
- Heapify(A,3)
- Heapify(A,2)
- Heapify(A,1)
- Heapify(A,0)



Build Heap : Analysis

- For (n+1)/2 nodes, heapify() requires at most 1 swap
- For (n+1)/4 nodes, heapify() requires at most 2 swap
- For (n+1)/8 nodes, heapify() requires at most 3 swap
- For (n+1)/2^i nodes, heapify() requires at most i swap

- Therefore, total swaps required is

$$T(n) = O\left(\frac{n+1}{2} + \frac{n+1}{2^2} * 2 + \frac{n+1}{2^3} * 3 + \cdots + 1.\log_2 n\right)$$

$$= O((n+1)\sum_{i=1}^{\log_2 n} \frac{i}{2^i} = O(n) \qquad (since \sum_{i=1}^{\log_2 n} \frac{i}{2^i} = 2)$$