University of North Texas

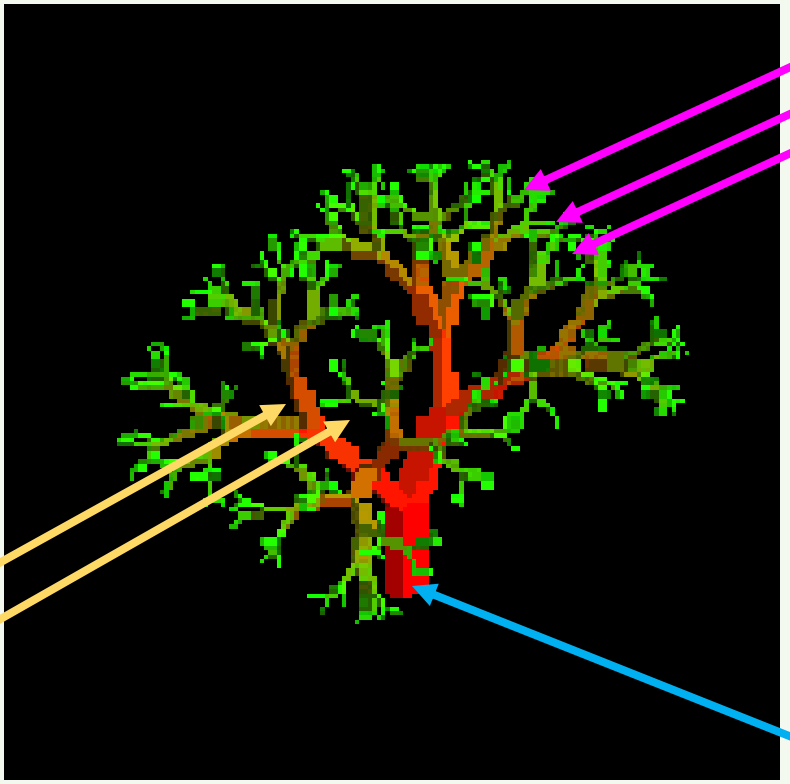
CSCE 2110
Foundations of Data Structures

Tree

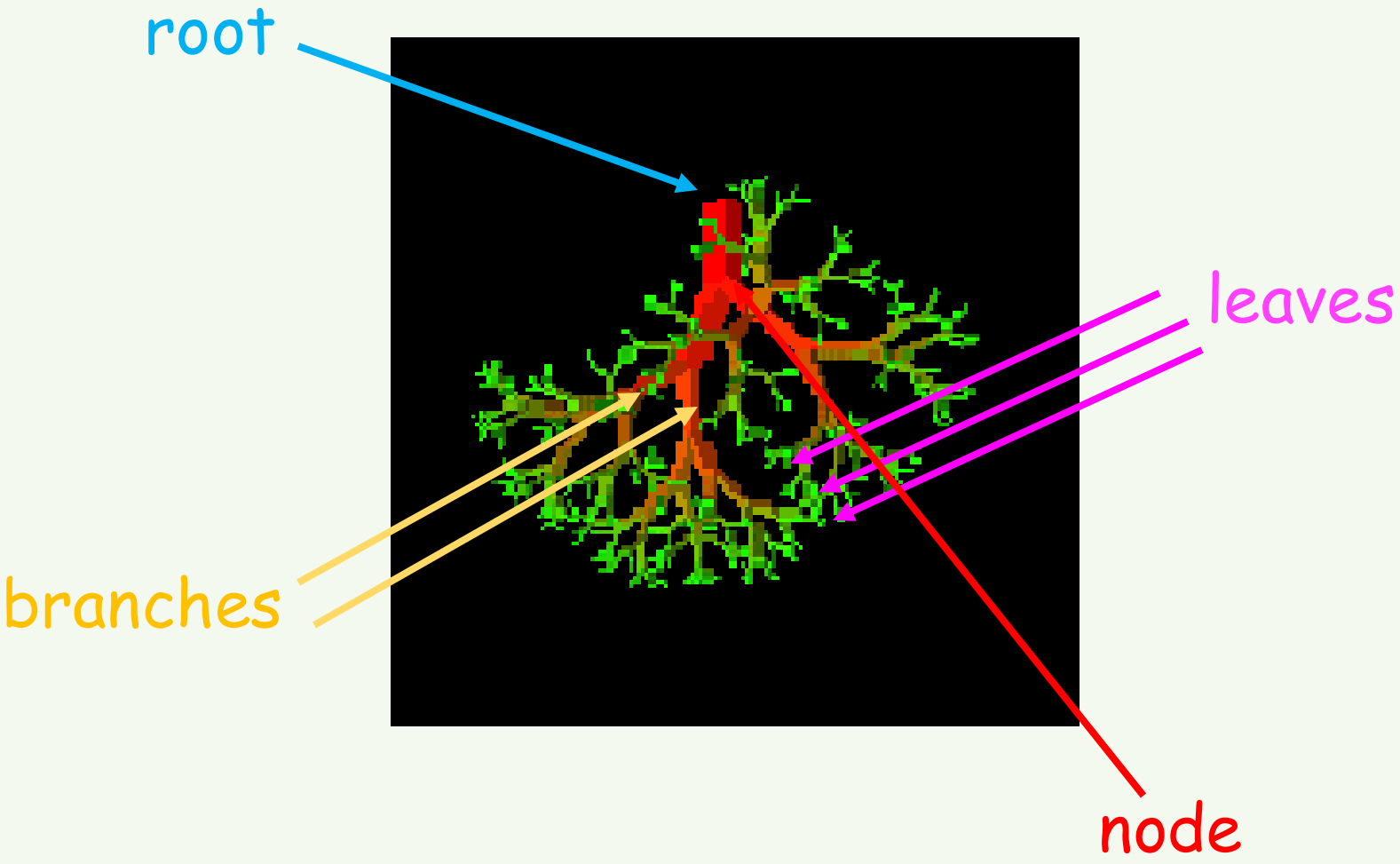
Contents

- Tree
  - Tree Traversal
- Binary Trees
- Binary Search Tree (BST)
- Balanced BST
  - AVL Tree
- Splay Tree

Nature View of a Tree

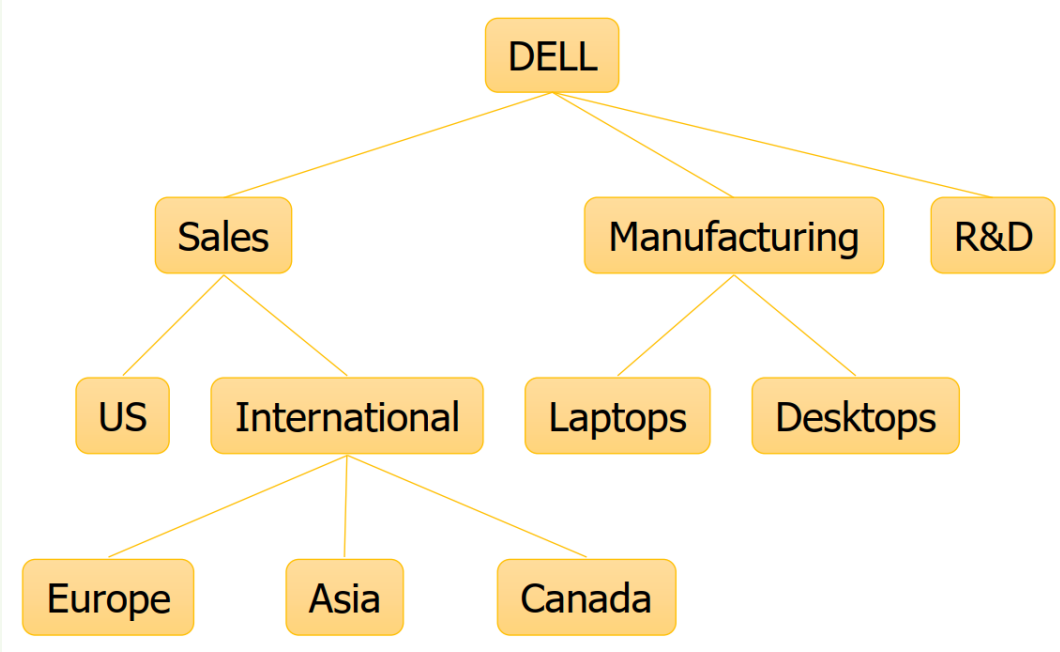


Computer Scientist's View



What is a Tree

- A tree is a finite nonempty set of elements.
- It is an abstract model of a hierarchical structure.
- consists of nodes with a parent-child relation.
- Applications:
  - Organization charts
  - File systems



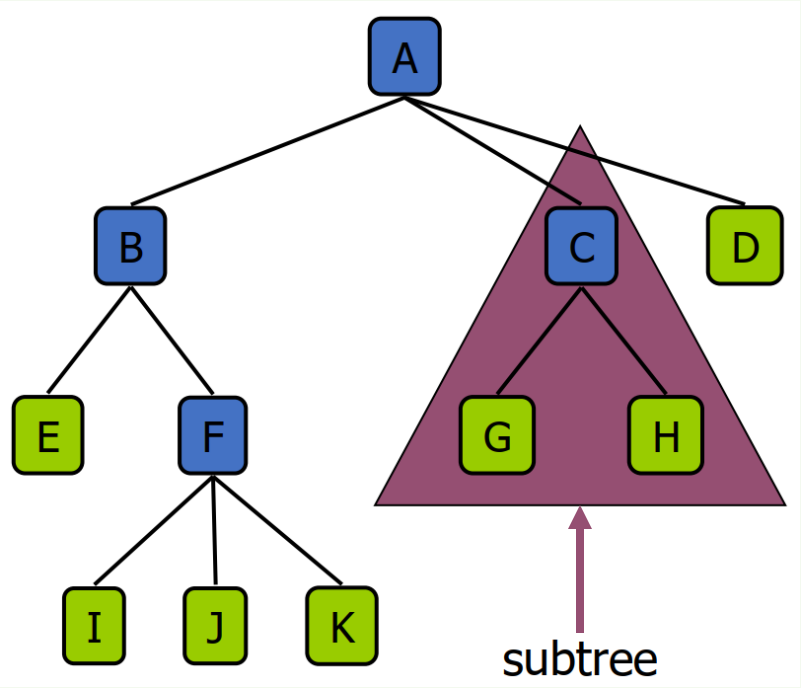
Definition and Tree Trivia

Recursive Definition of a Tree:

- A tree is a set of nodes that is
  - (a) an empty set of nodes, or
  - (b) has one node called the root from which zero or more trees (subtrees) descend.
- A tree with N nodes always has____edges

Tree Terminology

- Root: node without parent (A)
- Siblings: nodes share the same parent
- Internal node: node with at least one child (A, B, C, F)
- External node (leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand- grandparent, etc.
- Descendant of a node: child, grandchild, grand-grandchild, etc.
- Depth of a node: number of ancestors
- Height of a node: number of edges on the longest path from the node to a leaf
- Height/Depth of a tree: maximum depth of any node
- Degree of a node: the number of its children
- Degree of a tree: the maximum degree of its node.
- Subtree: tree consisting of a node and its descendants



YMTT (Yet More Tree Terminology)

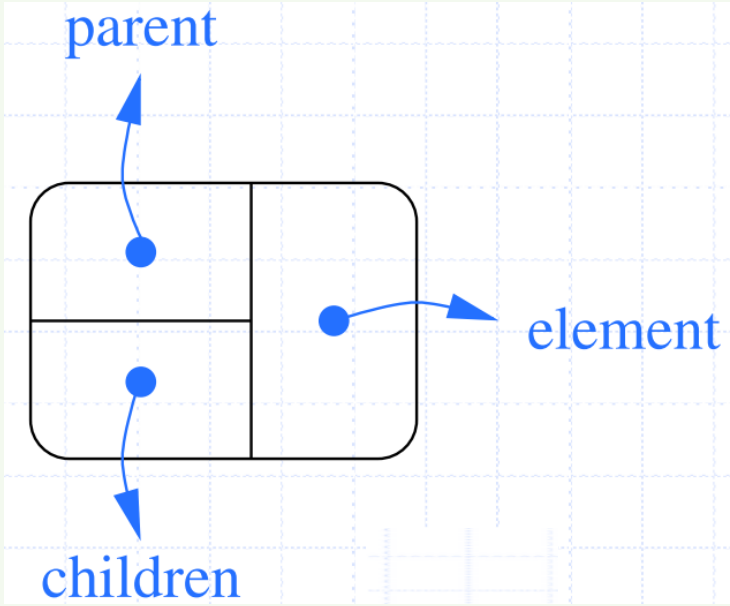
- Binary tree: each node has at most two children
- n-ary trere: each child has at most n children
- Complete tree: Each row of the tree is filled in left to right before the next one is started
- How deep can a complete binary tree (with n nodes) be?

Tree ADT

- We use positions to abstract nodes
- Query methods:
  - integer size()
  - boolean isEmpty()
  - objectIterator elements()
  - positionIterator positions()
  - position root()
  - position parent(p)
  - positionIterator children(p)
- Update methods:
  - insert(p)
  - delete(p)
  - swapElements(p, q)
  - object replaceElement(p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

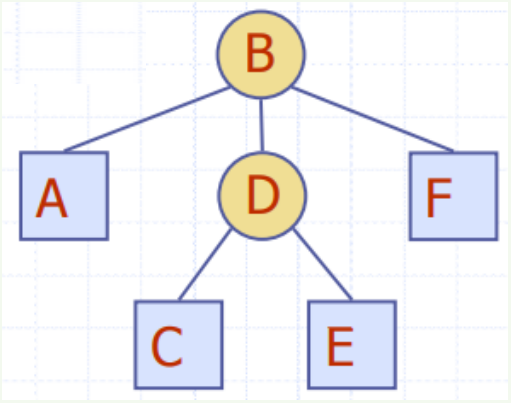
A Tree Representation

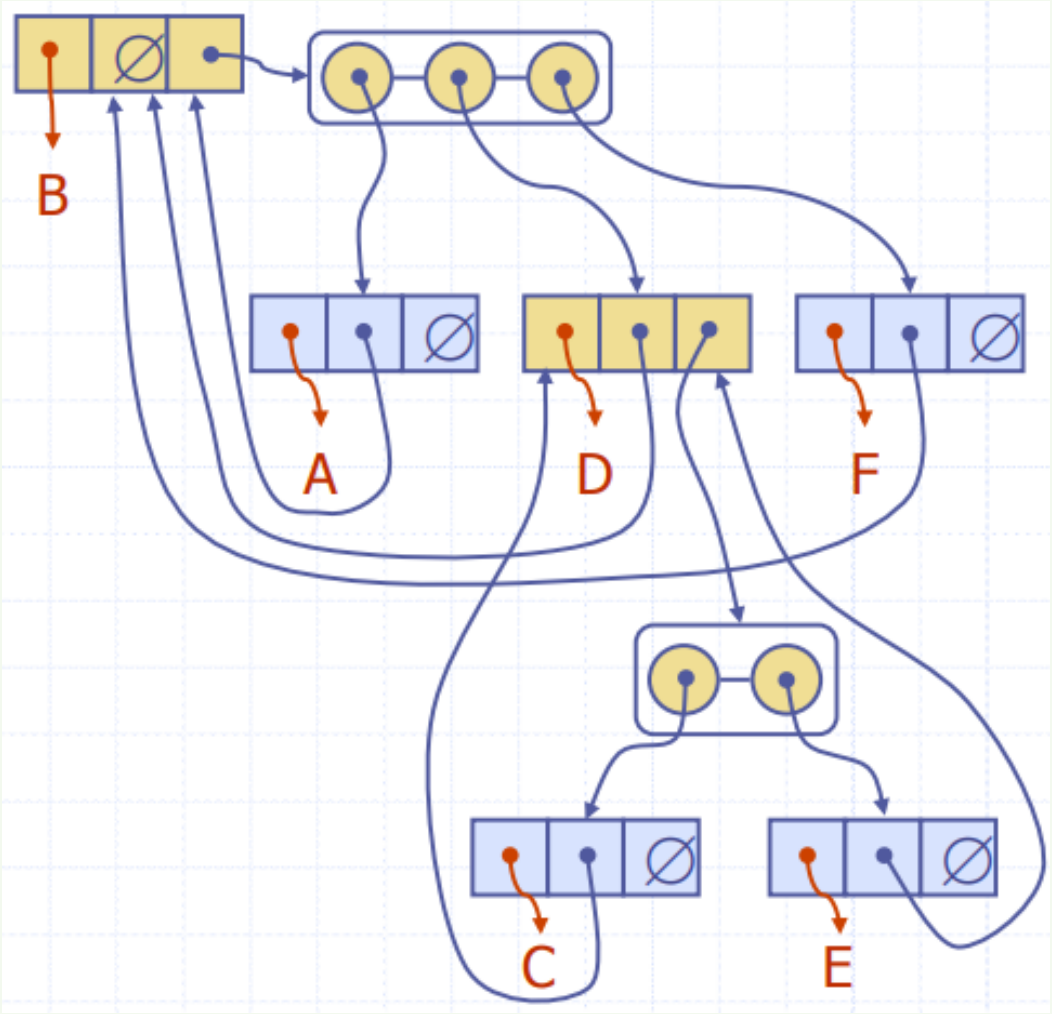
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes




```
// A node of N-ary tree
struct node {
    char element;
    node * parent;
    node * child[N];
};
```

A Tree Representation



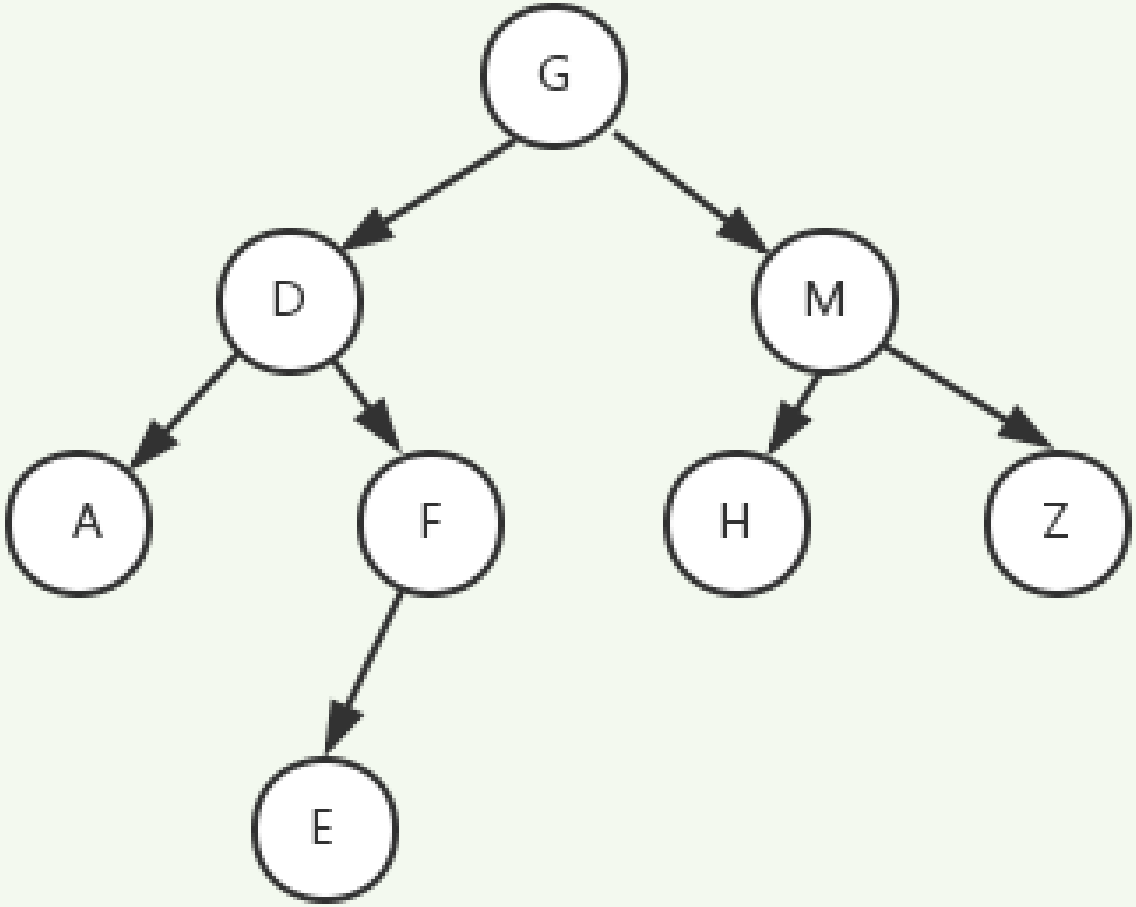


Tree Traversals

- Walking through a tree is called a traversal

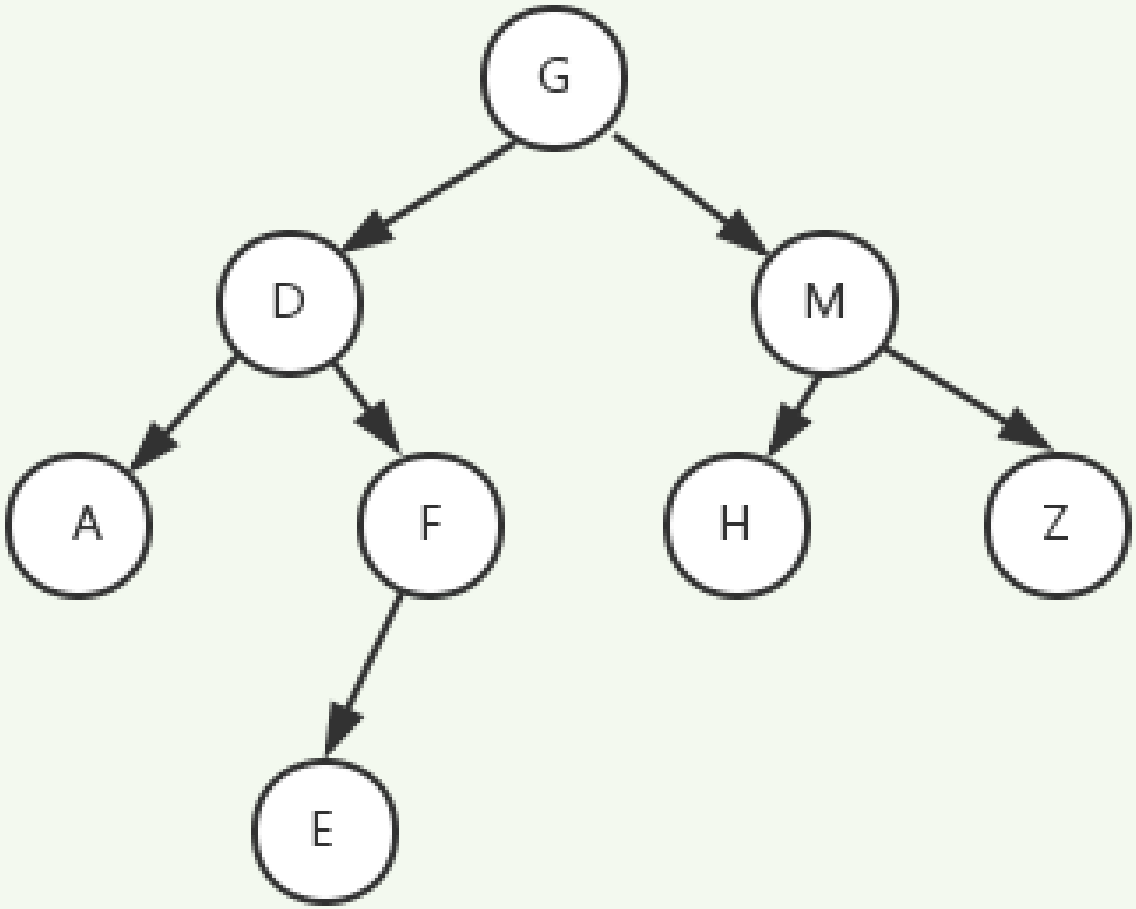
- Common kinds of traversal
  - Pre-order: node, then children
  - Post-order: children, then node
  - In-order: left, then node, then right (specific to binary trees)
  - Level-order: nodes at depth d, nodes at depth d+1, …

Tree Traversals



Tree Traversals

- Pre-order:
  G D A F E M H Z
- Post-order:
  A E F D H Z M G
- In-order:
  A D E F G H M Z



Tree Traversals

- Pre-order:
  A B D E G C F
- In-order:
  D B G E A C F
- Post-order:
  ?

Pre-Order Traversal

- Perform computation at the node, then recursively perform computation on each child

```
preorder(node * n) {
      node * c;
      if (n != NULL) {

            // DO SOMETHING;
            c = n->first_child();

            while (c != NULL) {
                  preorder(c);
                  c = c->next_sibling();
            }
      }
}
```

Pre-Order Applications

- Use when computation at node depends upon values calculated higher in the tree (closer to root)

- Example: computing depth
  - The depth of a node is the number of edges from the node to the tree's root node
  - depth(node) = 1 + depth(parent of node)

Pre-Order Example: Computing Depth of All Nodes

- Add a field `depth` to all nodes
- Call Depth(root,0) to set `depth` field

```
Depth(node * n, int d) {
  node * c;

  if (n != NULL) {
     n->depth = d;
     c = n->first_child();

     while (c != NULL) {
           Depth(c, d+1);
           c = c->next_sibling();
     }
  }
}
```

Post-Order Traversal

- Recursively perform computation on each child, then perform computation at node

```
postorder(node * n) {
  node * c;
  if (n != NULL) {
      c = n->first_child();
      while (c != NULL) {
            postorder(c);
            c = c->next_sibling();
      }
  // DO SOMETHING;
}
```

Post-Order Applications

- Use when computation at node depends on values calculated lower in tree (closer to leaves)

- Example: computing height
  - The height of a node is the number of edges on the longest path from the node to a leaf.
  - A leaf node will have a height of 0.
  - height(node) = 1 + MAX(height($child_1$), … height($child_k$))

- Example: size of tree rooted at node
  - size(node) = 1 + size($child_1$) + … + size($child_k$)

Post-Order Example: Computing Size of Tree

- Call Size(root) to compute number of nodes in tree

```
int Size(node * n) {
  node * c;
  if (n == NULL) return 0;
  else {
      int m = 1;
      c = n->first_child();

      while (c != NULL) {
             m += Size(c);
             c = c->next_sibling();
      }
   }
   return m;
}
```

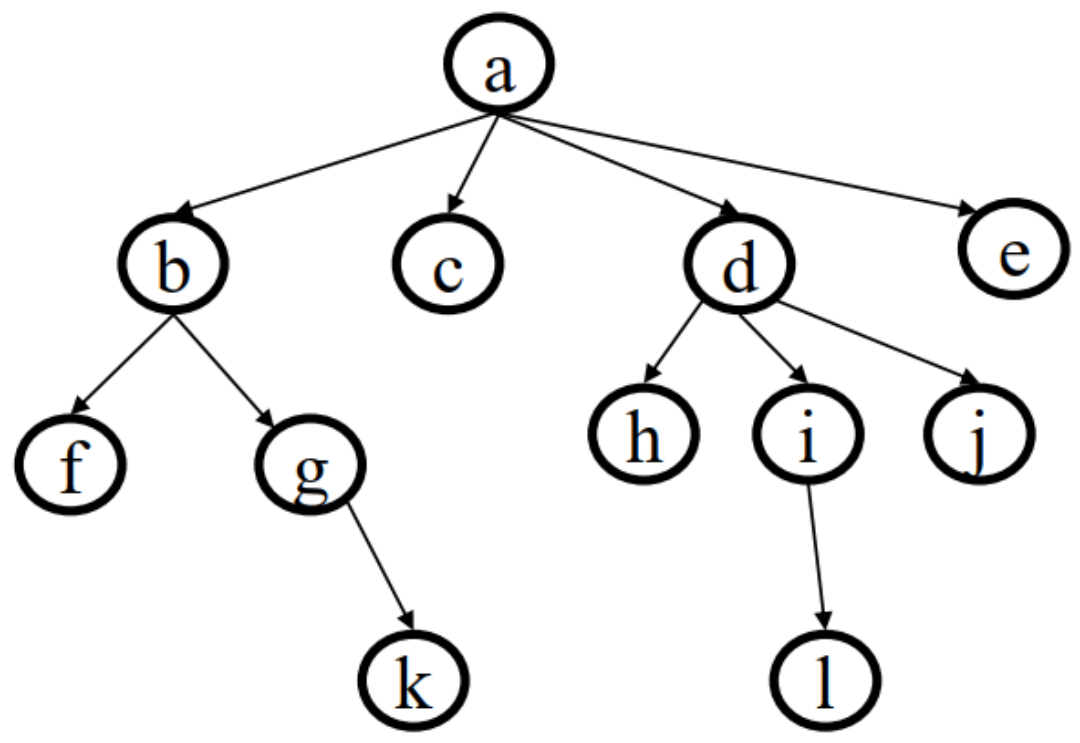
Depth-First Search

- Pre-Order and Post-Order traversals are examples of depth-first search:
  - Nodes are visited deeply on left-most branches before any nodes are visited on right-most branches
  - NOTE: visiting right deeply before left would still be depth-first - crucial idea is "go deep first"

- In DFS the nodes "being worked on" are kept on a stack (where?)

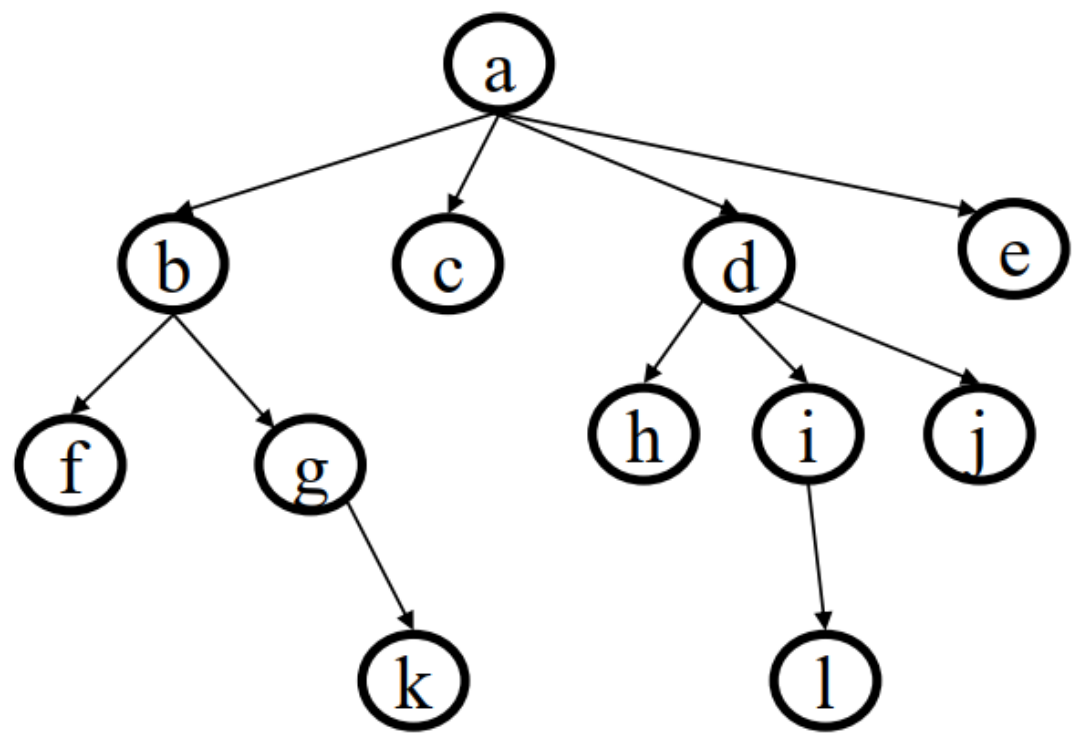
Level-Order (Breadth-First) Traversal

- Consider task of traversing tree level-by-level from top to bottom (alphabetic order, in example below)



Level-Order (Breadth-First) Traversal

- Consider task of traversing tree level-by-level from top to bottom (alphabetic order, in example below)



- Which data structure can best keep track of nodes?

Level-Order (Breadth-First) Algorithm

- Put root in a Queue
- Repeat until Queue is empty:
  - Dequeue a node
  - Process it
  - Add its children to queue

Level-Order Example: Printing the Tree

- Call Print(root) to print tree contents

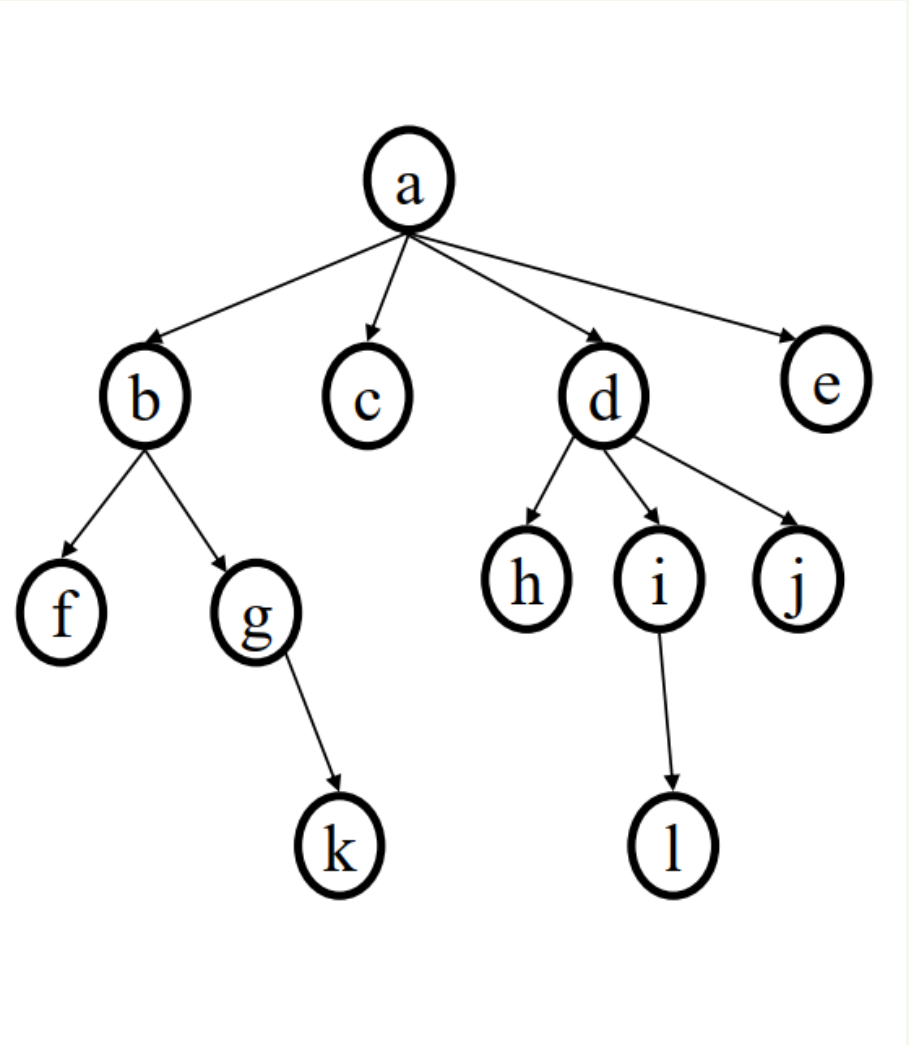
```
print(node * root) {
  node * n, c;  queue Q;

  Q.enqueue(root);
  while (! Q.empty()) {
      n = Q.dequeue();
      print n->data;

      c = n->first_child();
      while (c != NULL) {
             Q.enqueue(c);
             c = c->next_sibling();
      }
  }
}
```

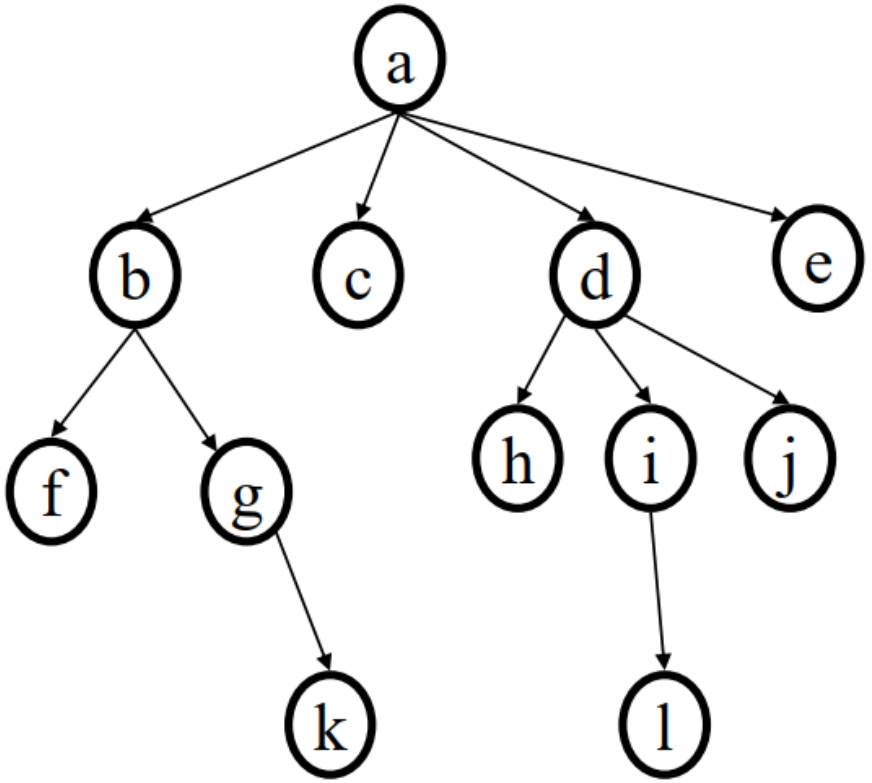
Example: Level-Order Queue

- Put root in a Queue
- Repeat until Queue is empty:
  - Dequeue a node
  - Process it
  - Add its children to queue



Example: Level-Order Queue

Process	Enqueue	Q
	a	**a**
a	b,c,d,e	**bcde**
b	f,g	**cdefg**
c		**defg**
d	h,i,j	**efghij**
e		**fghij**
f		**ghij**
g	k	**hijk**
h		**ijk**
i	l	**jkl**
j		**kl**
k		**l**
l		



Applications of Breadth-First Search

- Find shortest path from root to a given node N
  - if node N is at depth k, BFS will never visit a node at depth > k
  - important for really deep trees
- Generalizes to finding shortest paths in graphs
- Spidering the world wide web
  - From a root URL, fetch pages that are farther and farther away

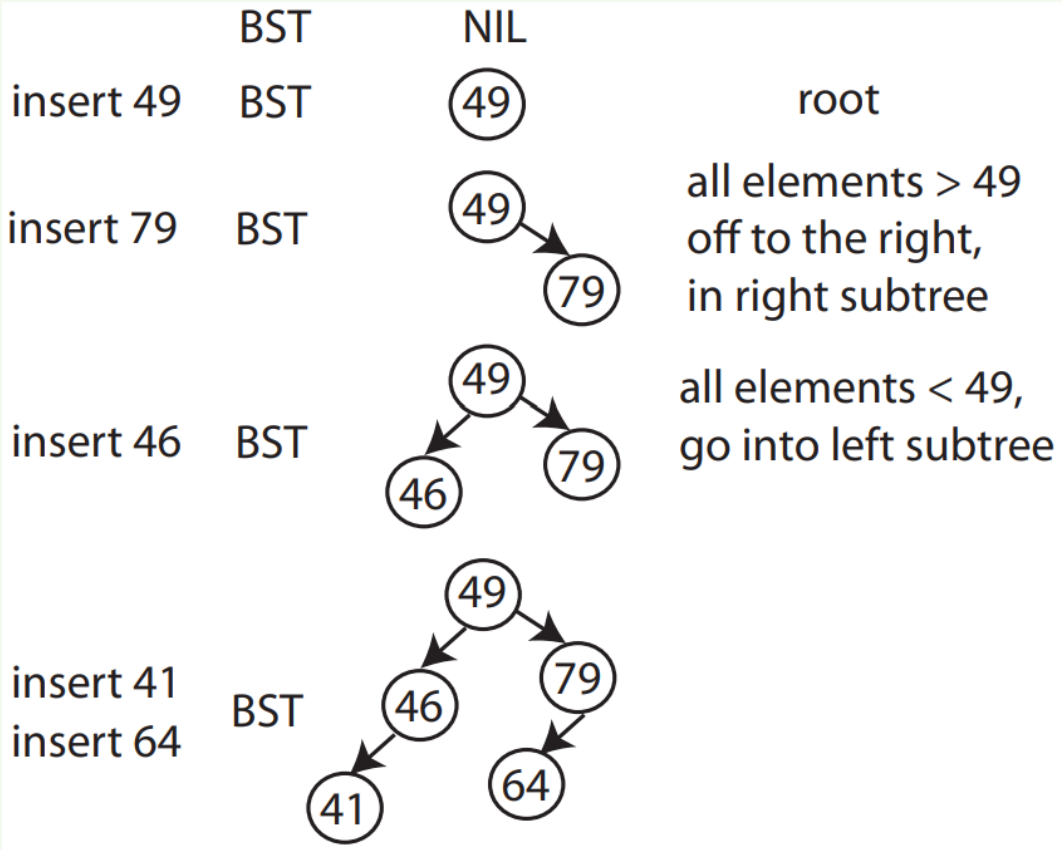
Binary Search Tree

- Binary Search Trees (BST) are a type of Binary Trees with a special organization of data.
  - Every element has a unique key.
  - The keys in the nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree (BST property)
  - The left and right subtrees are also binary search trees.

Binary Search Trees (BST)

- **Insertion:** `insert(val)`

```
typedef struct BinaryNode {
    int key;
    BinaryNode *right;
    BinaryNode *left;
}ptnode;
```



Insertion into a BST

```
void insert (ptnode * & node, int key){
   if (!node){
      node = (ptnode) malloc(sizeof(ptnode));
      node->key = key;
      node->left = node->right = NULL;
}  else if (key < node->key)
{
      insert(node->left, key);
}  else if (key > node->key)
{
      insert(node->right, key);
}
}
```

Finding a value in the BST if it exists: find(val)

- Follow left and right pointers until you find it or hit NULL.

Search in BST – Pseudocode

```
if the tree is empty
    return NULL
else if the item in the node equals the target
    return the node value
else if the item in the node is greater than the target
    return the result of searching the left subtree
else if the item in the node is smaller than the target
    return the result of searching the right subtree
```

Search in a BST

```
Ptnode search(ptnode * root,  int key){
  /* return a pointer to the node that
     contains key. If there is no such
     node, return NULL */

  if (!root) return NULL;

  if (key == root->key) return root;

  if (key < root->key)
       return search(root->left, key);

  return search(root->right, key);
}
```

Finding the minimum element in a BST: findmin()

- How?

Finding the minimum element in a BST: findmin()

- Key is to just go left till you cannot go left anymore.

BST Operations: Removal

- uses a binary search to locate the target item:
  - starting at the root it probes down the tree till it finds the target or reaches NULL
- removal of a node must not leave a 'gap' in the tree,

Removal in BST - Pseudocode

if the tree is empty return false

Attempt to locate the node containing the target using the binary search

- if the target is not found return false
- else the target is found, so remove its node:
  - Case 1: if the node has 2 empty subtrees
    - replace the link in the parent with null
  - Case 2: if the node has a left and a right subtree
    - replace the node's value with the max value in the left subtree
    - delete the max node in the left subtree

Removal in BST - Pseudocode

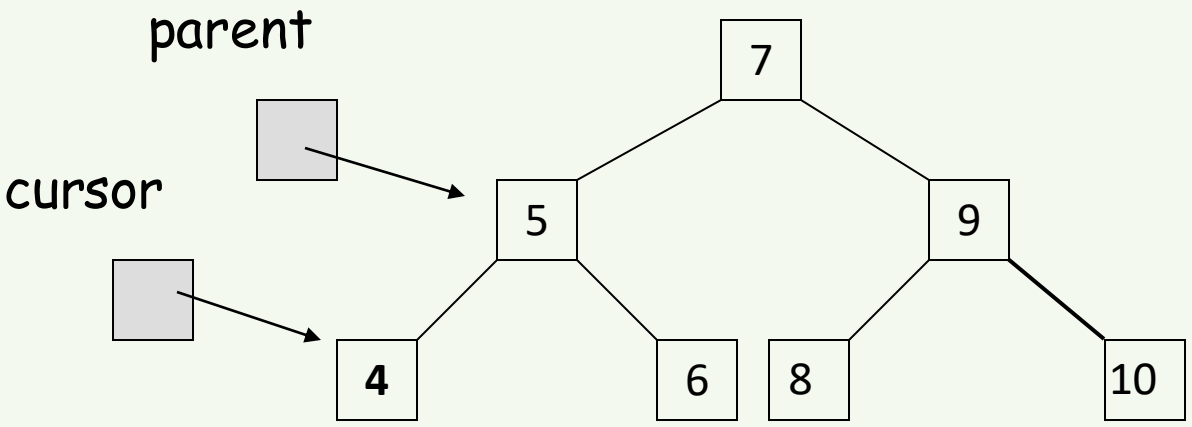
if the tree is empty return false

Attempt to locate the node containing the target using the binary search

- if the target is not found return false
- else the target is found, so remove its node:
  - Case 3: if the node has no left child
    - — link the parent of the node to the right (non-empty) subtree
  - Case 4: if the node has no right child
    - — link the parent of the target to the left (non-empty) subtree

Removal in BST: Example

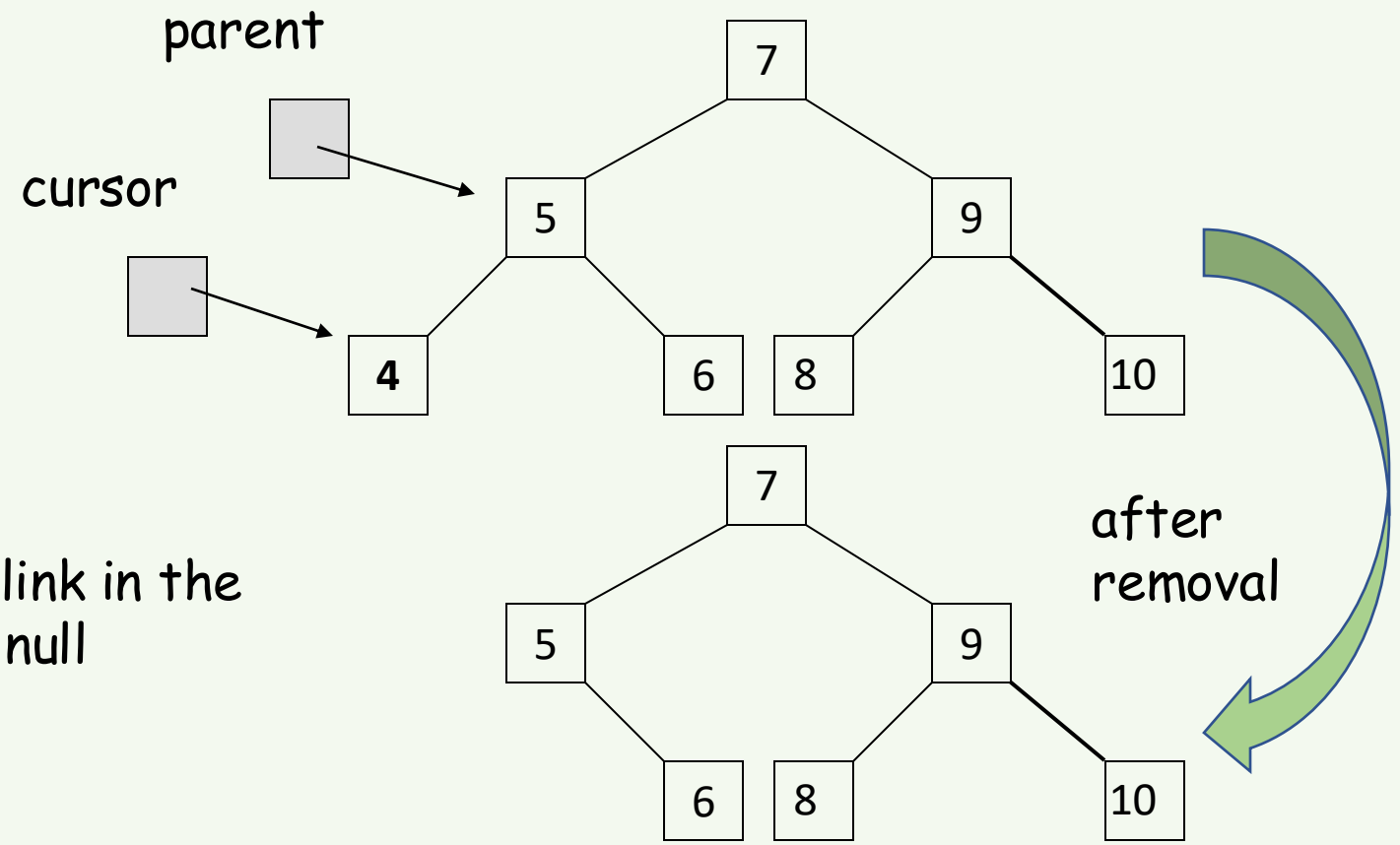
Case 1: removing a node with 2 EMPTY SUBTREES



Removing 4

Removal in BST: Example

Case 1: removing a node with 2 EMPTY SUBTREES



Removing 4
replace the link in the parent with null

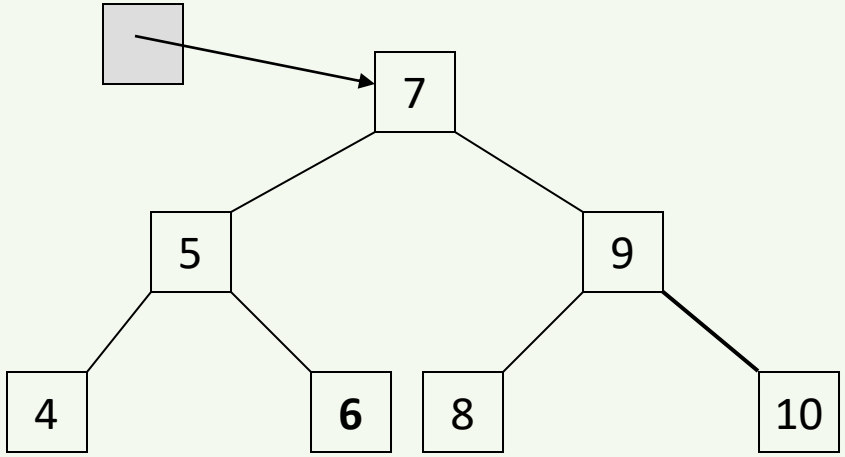
after removal

Removal in BST: Example

Case 2: removing a node with 2 SUBTREES

Removing 7

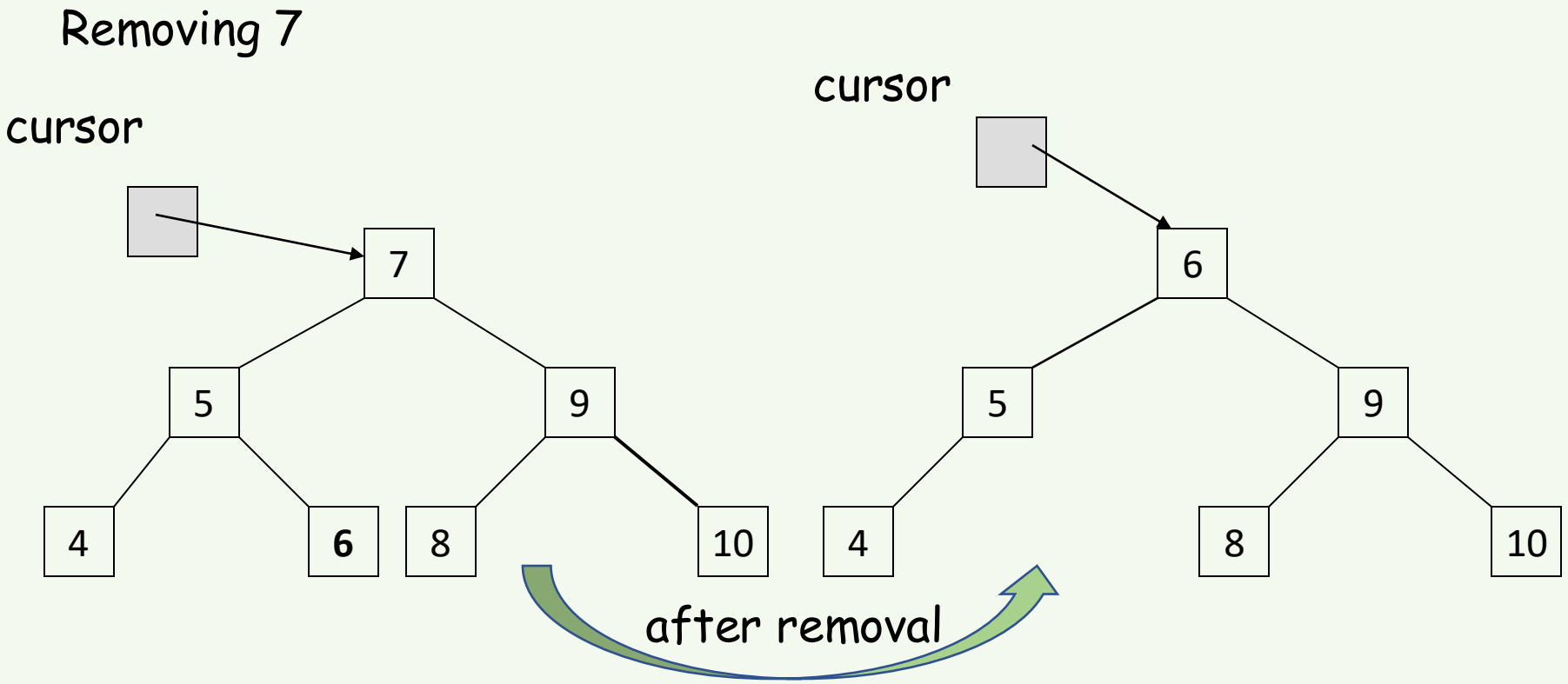
cursor



Removal in BST: Example

Case 2: removing a node with 2 SUBTREES

- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree

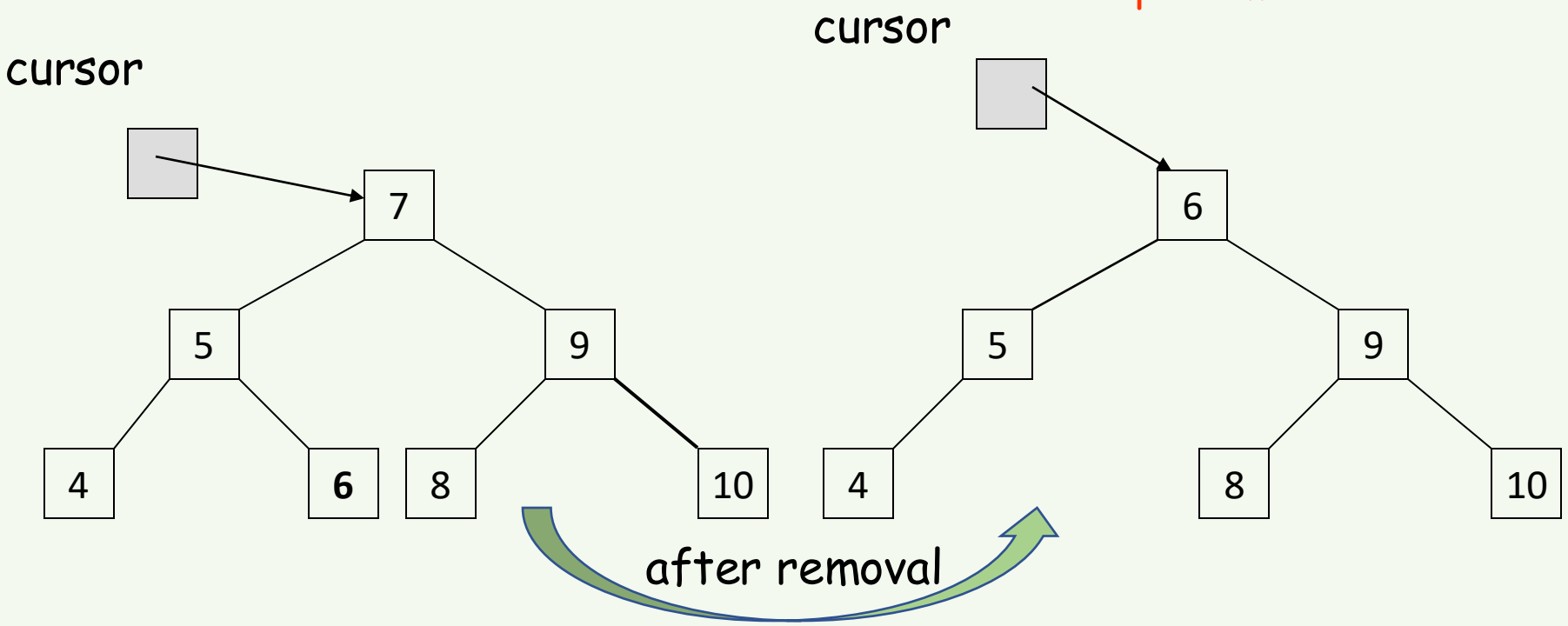


Removal in BST: Example

Case 2: removing a node with 2 SUBTREES

- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree

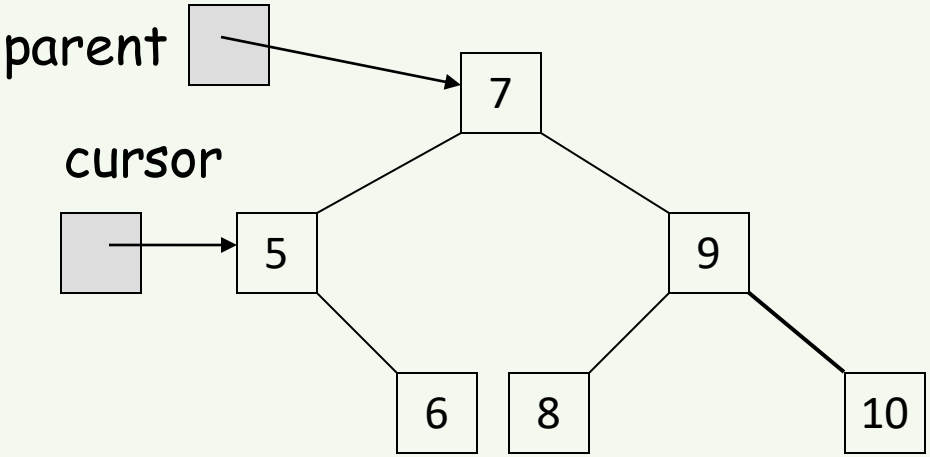
What element else can be used as replacement?



Removal in BST: Example

Case 3: removing a node with 1 EMPTY SUBTREE

Removing 5

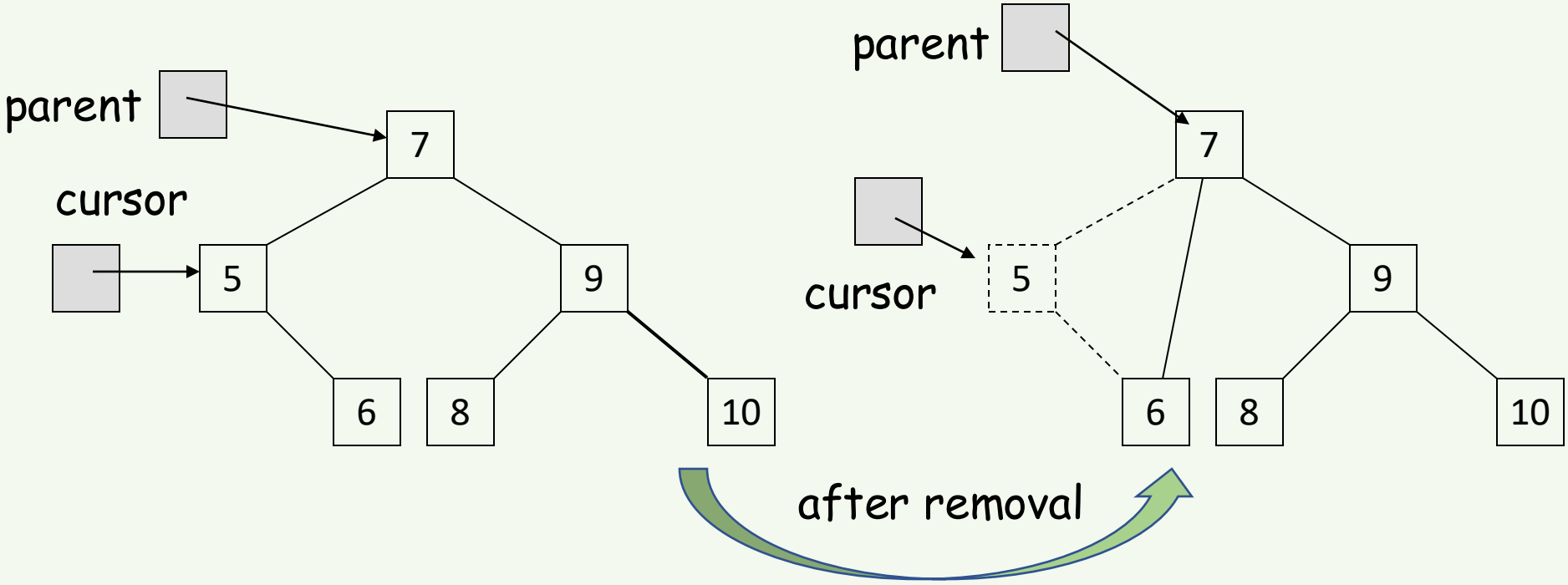


Removal in BST: Example

Case 3: removing a node with 1 EMPTY SUBTREE

- the node has no left child: link the parent of the node to the right (non-empty) subtree

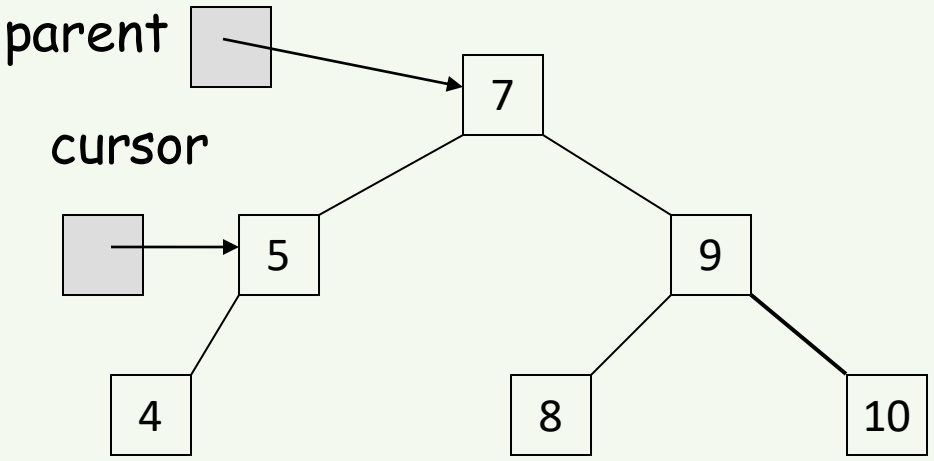
Removing 5



Removal in BST: Example

Case 4: removing a node with 1 EMPTY SUBTREE

Removing 5

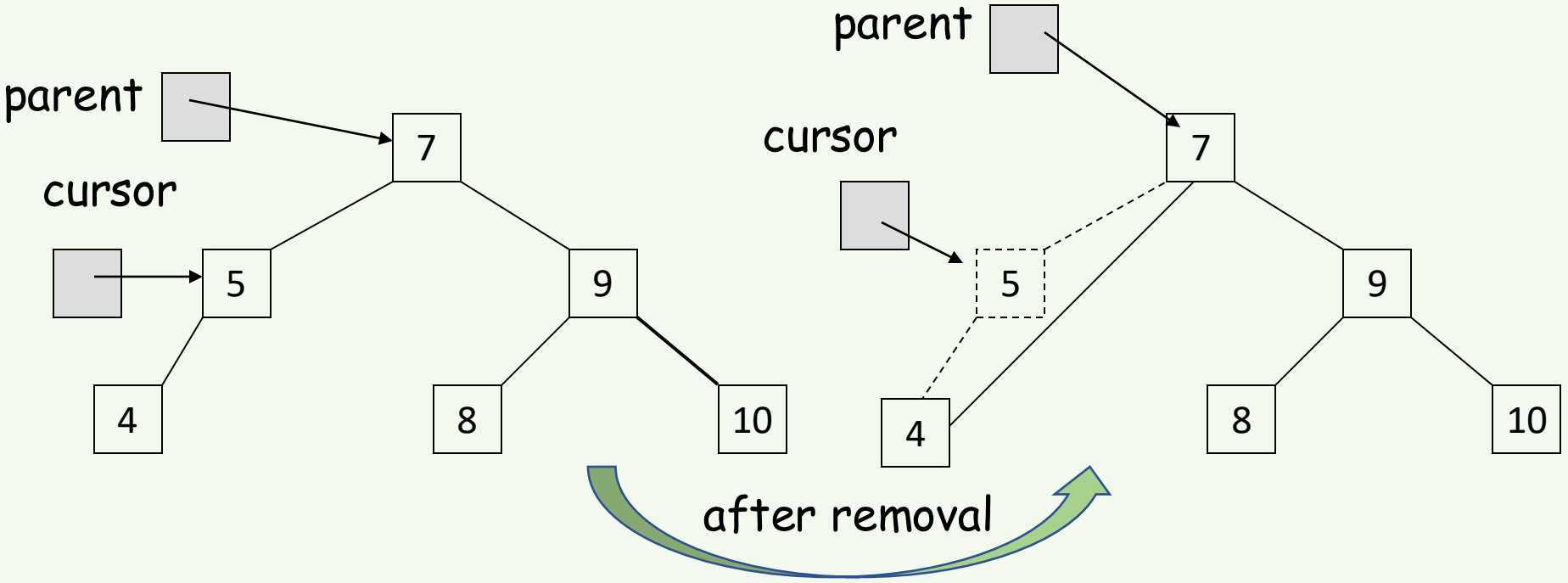


Removal in BST: Example

Case 4: removing a node with 1 EMPTY SUBTREE

the node has no right child:
link the parent of the node to the left (non-empty) subtree

Removing 5



Analysis of BST Operations

- The complexity of operations **`search`**, **`insert`** and **`remove`** in BST is $O(h)$, where h is the height.

- $O(\log n)$ when the tree is balanced, i.e., $h = O(\log n)$. The updating operations may cause the tree to become unbalanced.

- The tree can degenerate to a linear shape and the operations will become $O(n)$

Worst Case

```
BST tree = new BST();
for (int i = 1; i <= 8; i++)
   tree.insert (i);
```

```
>>>> Items in worst order:
        8
       7
      6
     5
    4
   3
  2
 1
```

Output

Balanced BSTs

Prevent the degeneration of the BST :

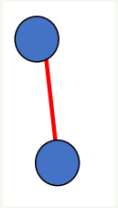
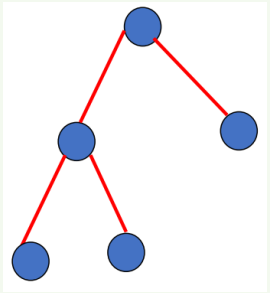
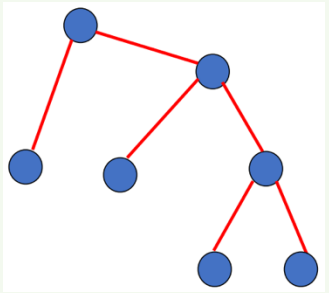
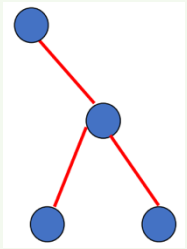
- A BST can be set up to maintain balance during updating operations (insertions and removals)
- To achieve a worst-case runtime of $O(\log n)$ for searching, inserting and deleting, h = $O(\log n)$

- Two types we'll look at :
  - AVL trees (named after inventors Adelson-Velsky and Landis)
  - splay trees
- There are many other types of balanced BSTs: 2-4 trees, Red-Black trees, B-trees

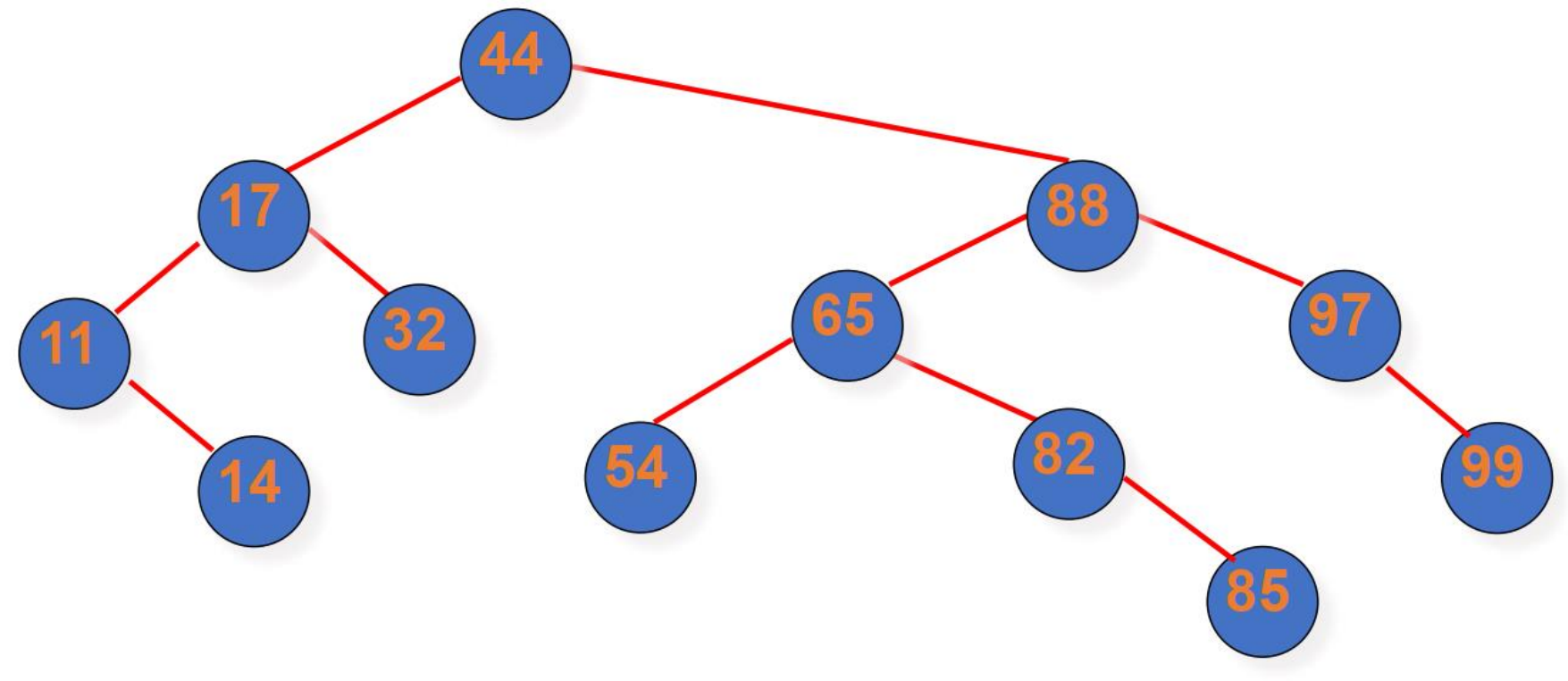
AVL Trees

- Invented in 1962 by Russian mathematicians Adelson-Velsky and Landis
- An AVL tree is a binary search tree such that (AVL property):
  - The height of the left and right sub-trees of the root differ by at most 1
  - The left and right sub-trees are AVL trees
  - treat nil tree as height -1

Which of these are AVL trees, assuming that they are BSTs?

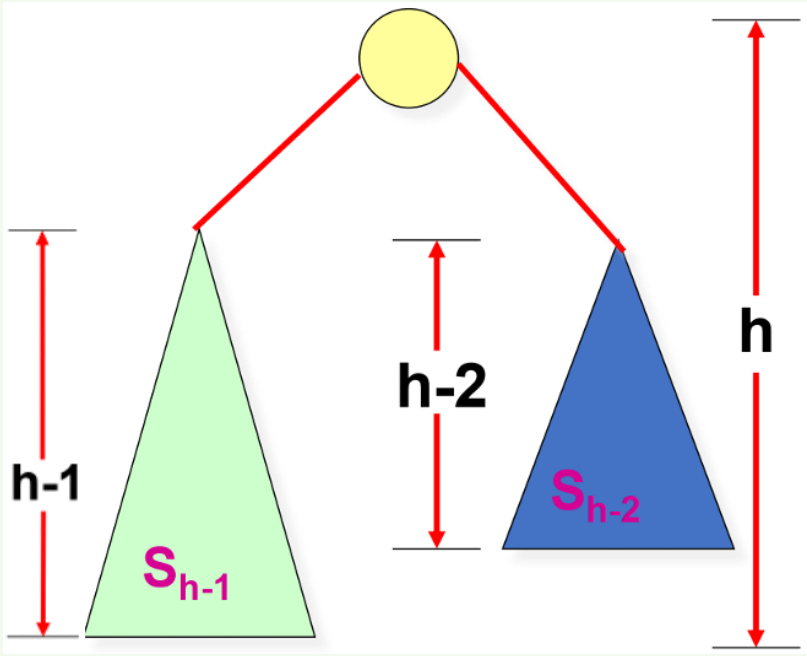
Valid AVL Tree



Note: it is not a requirement that all leaves be on the same or adjacent level

AVL Trees: Balanced

- Aim to get a tight upper bound for h
- Worst when the height of the left and right sub-trees of every node differs by 1
- let N_h = ($min.$) # nodes in height-h AVL tree



- $N_h = N_{h-1} + N_{h-2} + 1$
- $> 2N_{h-2}$
- $N_h > 2^{h/2}$
- $h < 2 \log N_h$

AVL Trees: Balanced

Based on $N_h = N_{h-1} + N_{h-2} + 1$, we can have N_{h-1} as follows by replacing h with h-1

$N_{h\text{-}1} = N_{h-2} + N_{h-3} + 1$

We can plug $N_{h\text{-}1}$ into N_h, and obtain the following

$N_h = N_{h-1} + N_{h-2} + 1 = (N_{h-2} + N_{h-3} + 1) + N_{h-2} + 1 = 2N_{h-2} + N_{h-3} + 2$

So, we have $N_h > 2N_{h-2}$. We can replace h with h-2, obtaining $N_{h\text{-}2} > 2N_{h-4}$

We can plug the above into $N_h > 2N_{h-2}$ to obtain the following

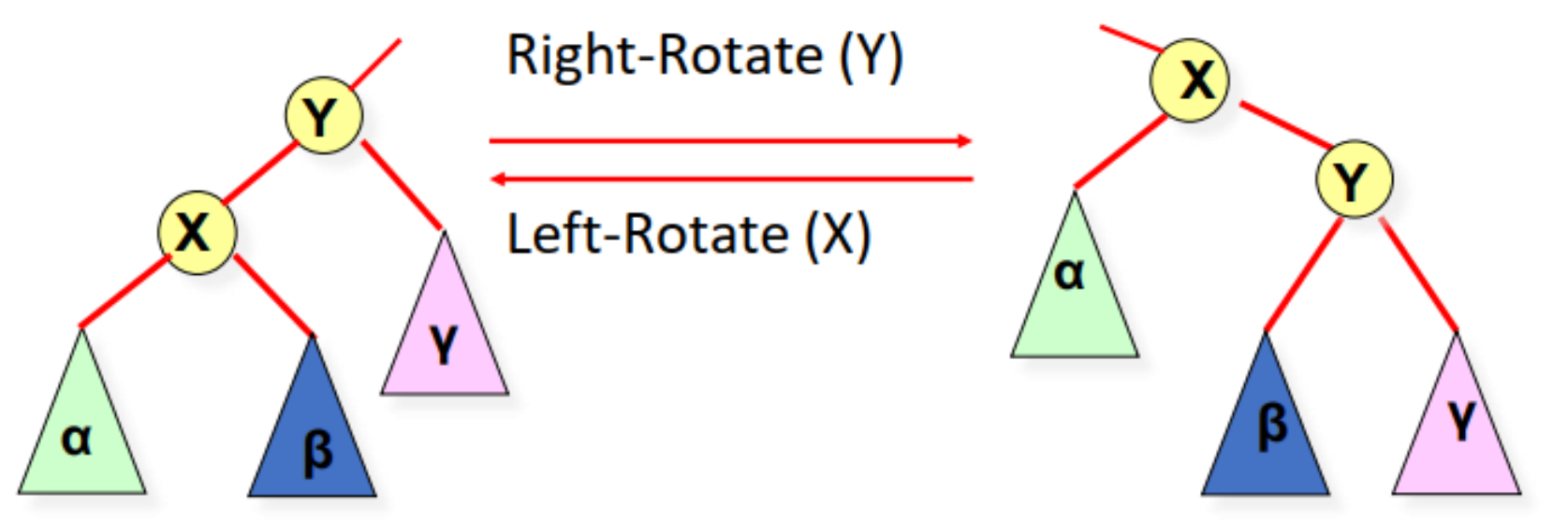
$N_h > 2N_{h-2} > 2{\times}2N_{h-4} \Rightarrow N_h > 2^2N_{h-2\times2} \Rightarrow N_h > 2^kN_{h-2k}$

When h-2k = 0, i.e., k = h/2, we have

$N_h > 2^{h/2}N_0 \Rightarrow N_h > 2^{h/2} \Rightarrow \log(N_h) > h/2 \Rightarrow 2\log(N_h) > h$

With $h < 2\log(N_h)$, we can easily get $h=O(\log(N_h))$ using the definition of big O.

Rotations



Rotations maintain the ordering property of BSTs.
A rotation is an O(1) operation

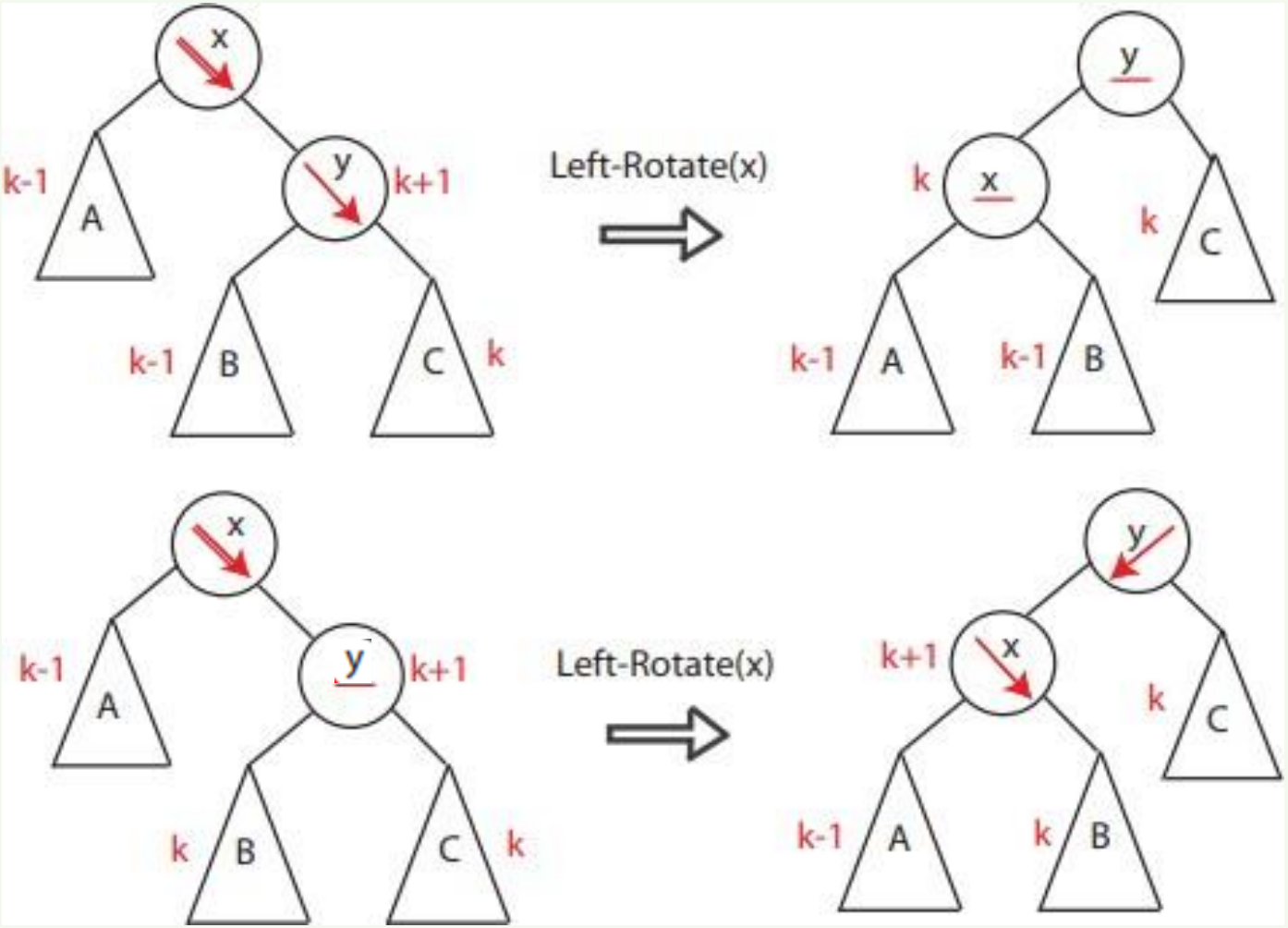
AVL Insert

(1) insert as in simple BST

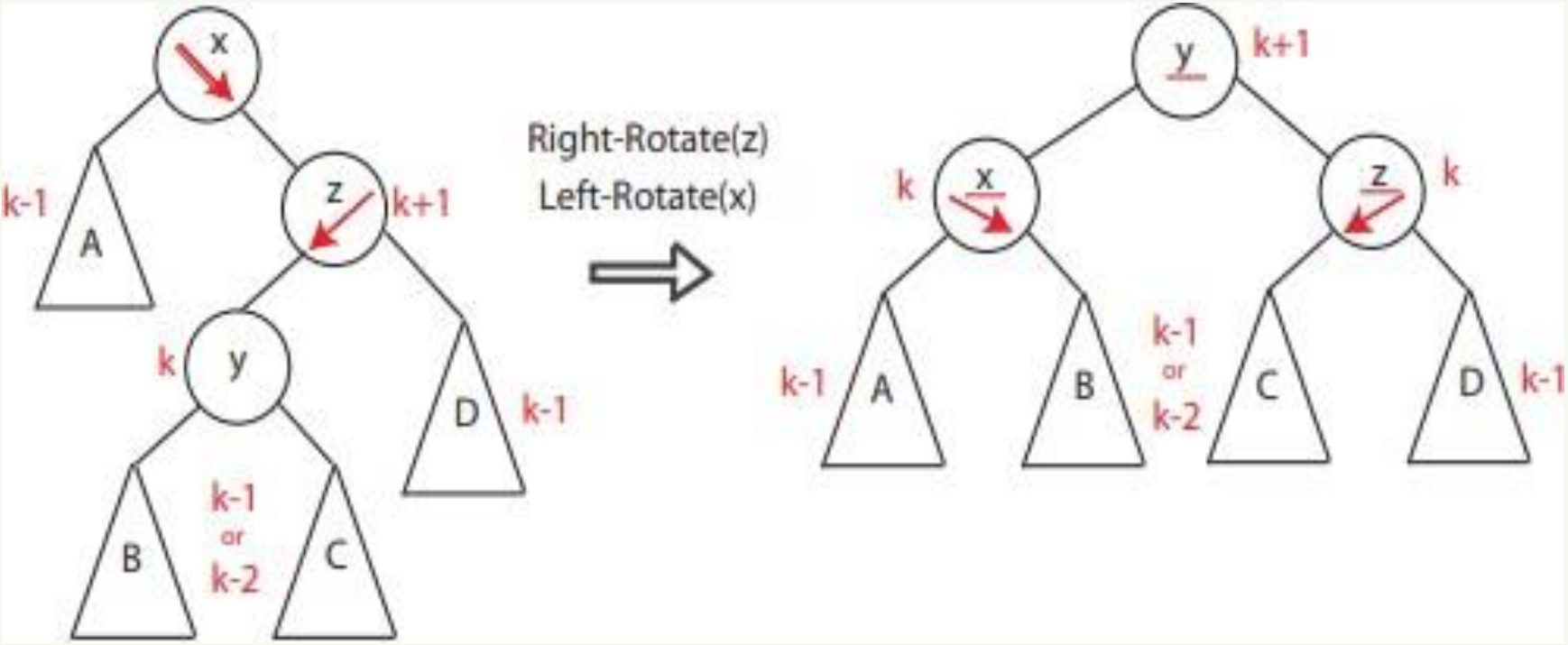
(2) work your way up tree, restoring AVL property (and updating heights as you go).

- suppose x is the lowest node violating AVL
- assume x is right-heavy (left case symmetric)
- two cases:
  - Case 1: x's right child is right-heavy or balanced → single rotation
  - Case 2: else → double rotations
- then continue up to x's parent, grandparent, greatgrandparent . . .

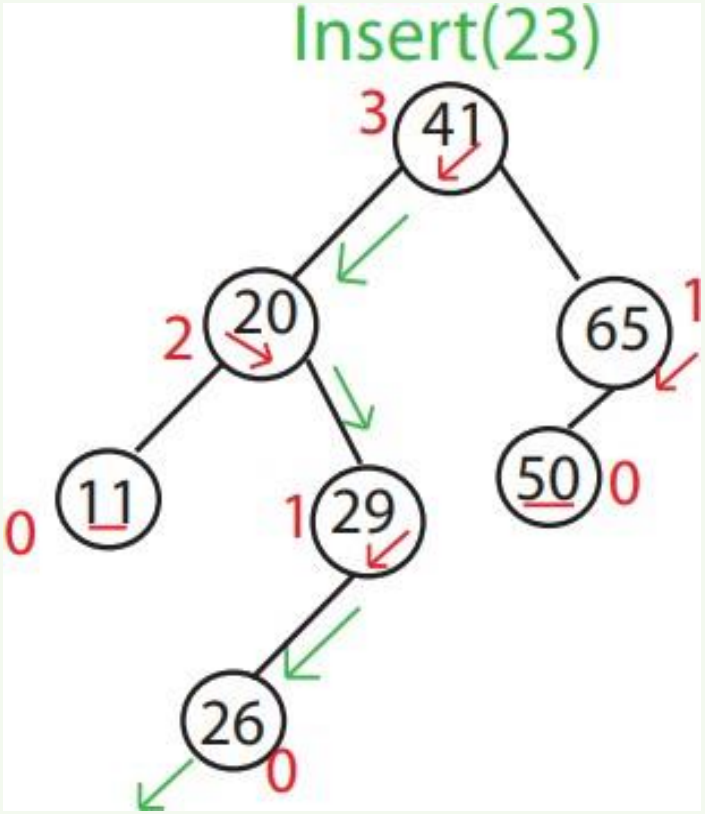
Case 1

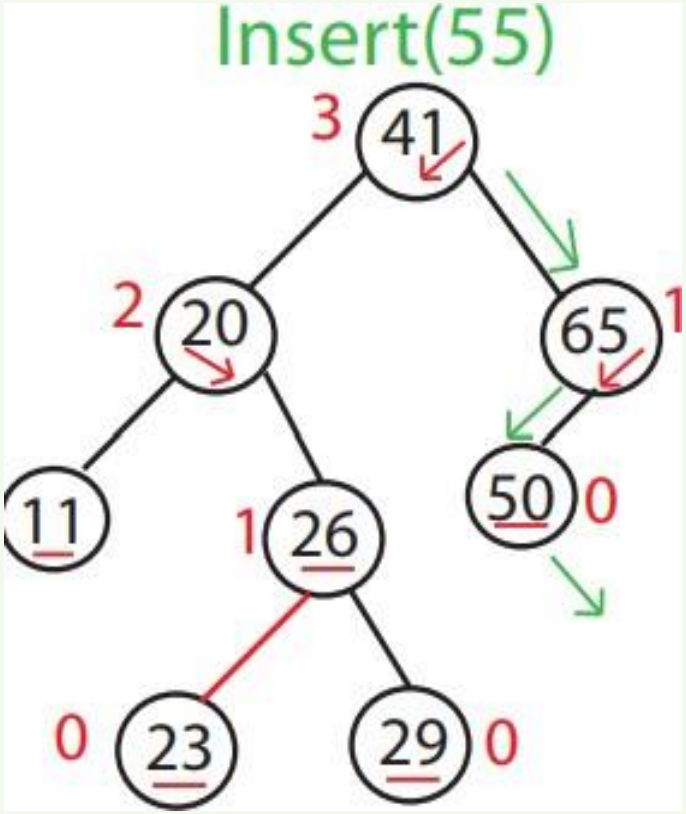

x
y
k-1
A
k+1
k-1
B
C
k
Left-Rotate(x)
y
x
k
k
C
k-1
A
k-1
B
x
y
k-1
A
k+1
k+1
k
B
C
k
Left-Rotate(x)
y
x
k+1
k
C
k-1
A
k
B

Case 2


x
k-1
A
z
k+1
k
y
D
k-1
B
k-1
or
k-2
C
Right-Rotate(z)
Left-Rotate(x)
y
k+1
k
x
z
k
k-1
A
B
k-1
or
k-2
C
D
k-1

Example


Insert(23)
3
41
2
20
65
1
0
11
1
29
50
0
26
0


Insert(55)
3
41
2
20
65
1
11
1
26
50
0
0
23
29
0

Balancing AVL tree

```
void balance( AvlNode * & t ) {
    if( t == NIL ) return;
    if( height( t->left ) - height( t->right ) > 1)
        if( height( t->left->left ) >= height( t->left->right ) )
            rotateWithLeftChild( t );
        else
            doubleWithLeftChild( t );
    else if( height( t->right ) - height( t->left ) > 1)
        if( height( t->right->right ) >= height( t->right->left ) )
            rotateWithRightChild( t );
        else
            doubleWithRightChild( t );
        t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

Balancing AVL tree

```
void rotateWithLeftChild( AvlNode * & k2 ) {
    AvlNode *k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
    k1->height = max( height( k1->left ), k2->height ) + 1;
    k2 = k1;
}
```

Comments

- In general, process may need several rotations before done with an Insert. $O(\log n)$ rotations may be required
- Deletion is similar — harder but possible.

- Running time for inserting each item into AVL tree?

Advantages/Disadvantage of AVL Trees

- Advantages
  - $O(\log n)$ worst-case searches, insertions and deletions

- Disadvantages
  - Complicated Implementation
    - Must keep balancing info in each node
    - To find node to balance, must go back up in the tree: easy if pointer to parent, otherwise difficult
    - Deletion complicated by numerous potential rotations