Searching and Trees

- COMP1927 Computing 16x1
- Sedgewick Chapters 5, 12

SEARCHING (CONT)

Searching is a very important/frequent operation. Several approaches have been developed:

- *O(n)* ... linear scan (search technique of last resort)
- *O(logn)* ... binary search, **search trees** (trees also have other uses)
- *O(1)* ... **hash tables** (only *O(1)* under optimal conditions)

SEARCHING (CONT)

Linear structures: arrays, linked lists

Arrays = random access.

Lists = sequential access.

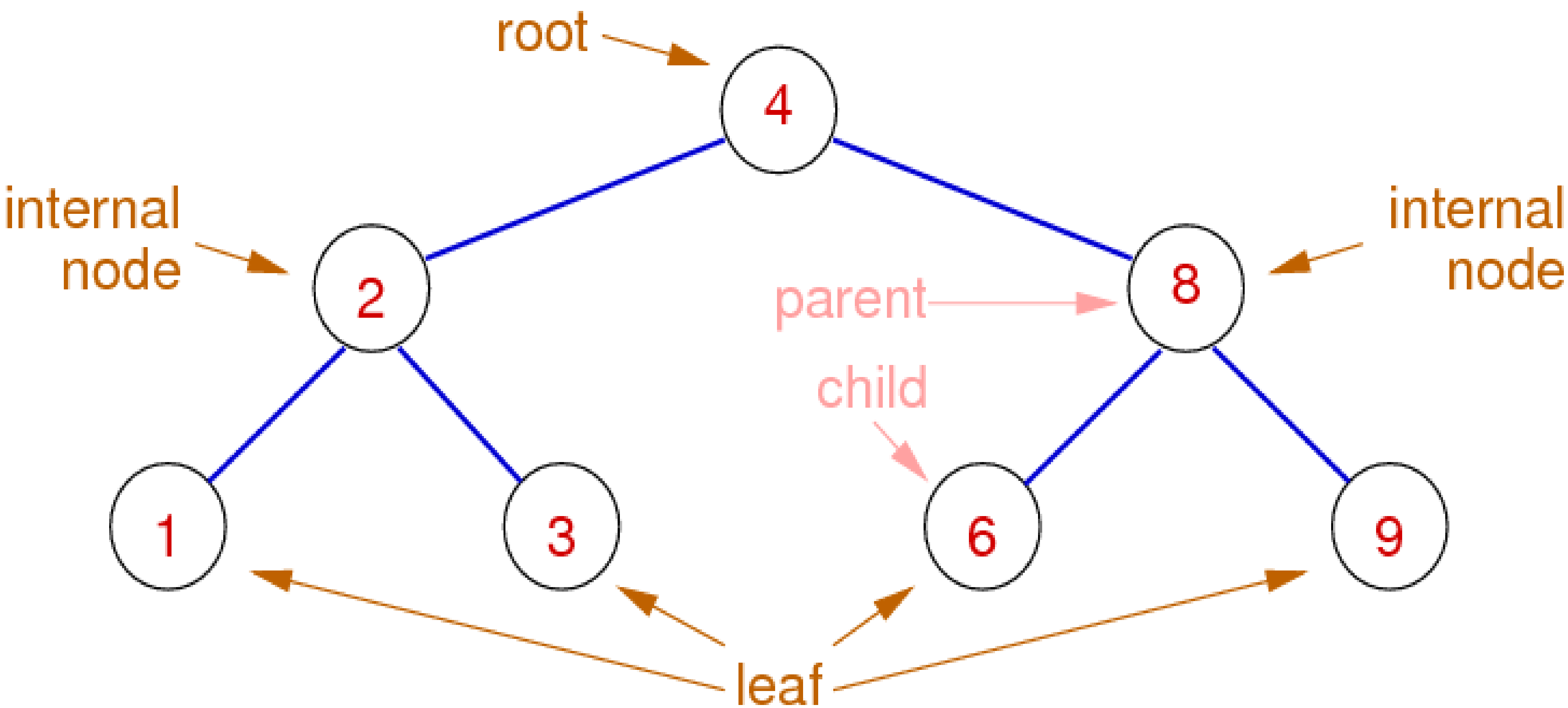
	Array	**List**
Unsorted	$O(n)$ *(linear scan)*	$O(n)$ *(linear scan)*
Sorted	$O(\log\ n)$ *(binary search)*	$O(n)$ *(linear scan)*

SEARCHING

- Storing and searching sorted data:
- Dilemma: Inserting into a sorted sequence
  - Finding the insertion point on an array – O(log n) but then we have to move everything along to create room for the new item
  - Finding insertion point on a linked list O(n) but then we can add the item in constant time.
- Can we get the best of both worlds?

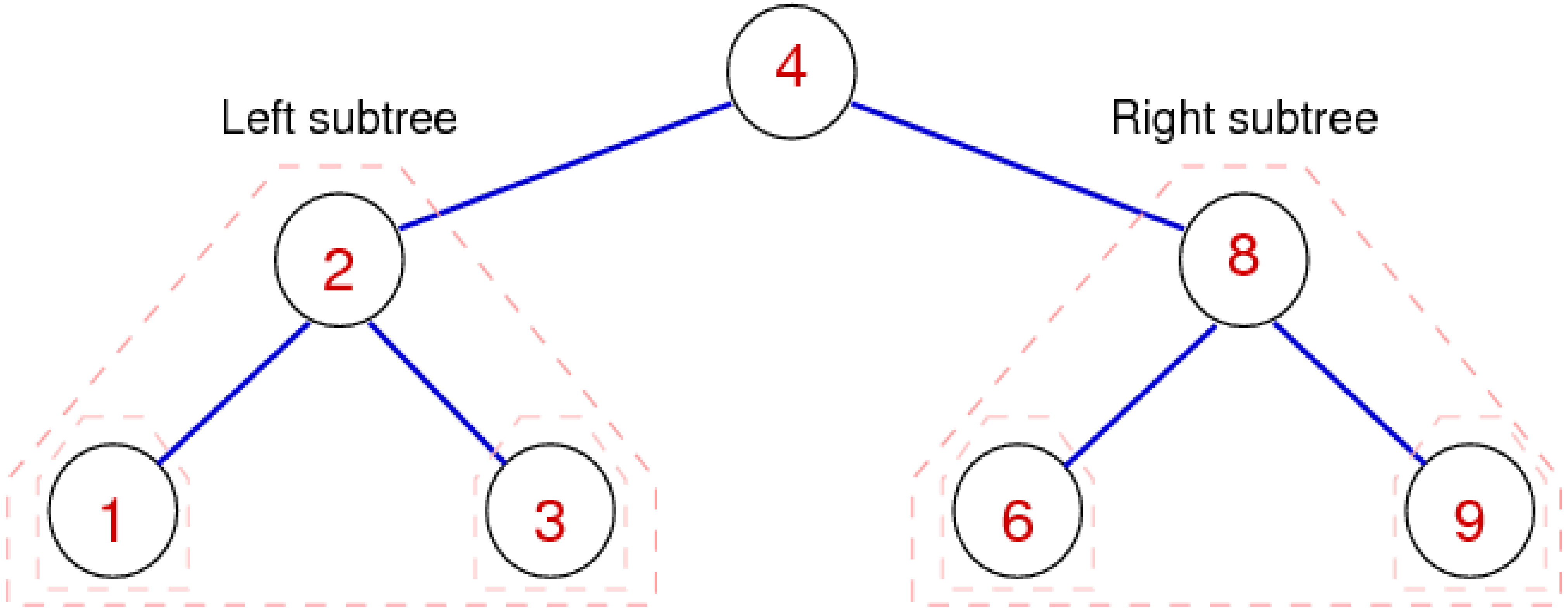
Tree terminology

- Trees are branched data structures consisting of *nodes (vertices)* and *links (edges)*, with no cycles
- each node contains a data value
- each node has links to $\leq k$ other nodes (k=2 below)



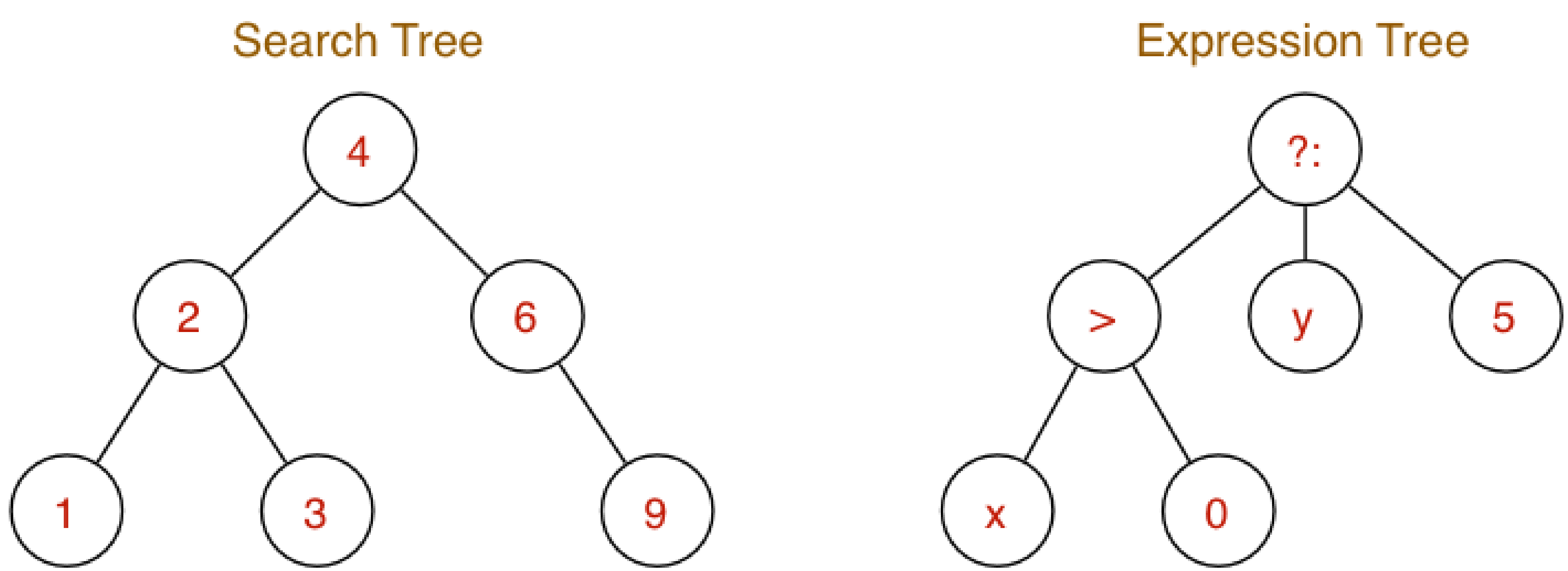
TREES AND SUBTREES

- Trees can be viewed as a set of nested structures: each node has k possibly empty subtrees



USES OF TREES

- Trees are used in many contexts, e.g. representing hierarchical data structures (e.g. expressions)
- efficient searching (e.g. sets, symbol tables, ...)



Special properties of some trees

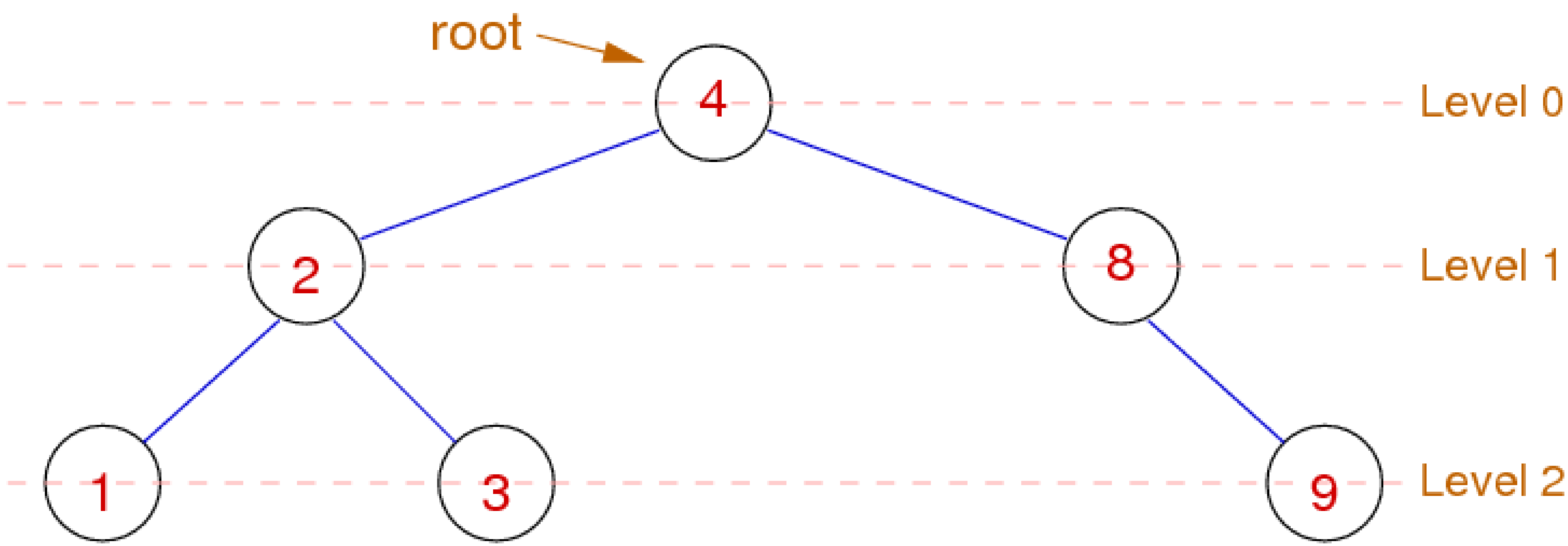
- M-ary tree: each internal node has exactly M children
- Ordered tree: order of the children at every node is specified through constraints on the data/keys in the nodes
- Balanced tree: a tree with properties that
  - #nodes in left subtree = #nodes in right subtree
  - this property applies over all nodes in the tree

BINARY TREES

- For much of this course, we focus on *binary trees* (*k=2*)
- A *binary tree* (simplest type of M-ary tree) is an ordered tree which can be defined recursively, as being either:
  - empty (contains no nodes)
  - consisting of a node, with two sub-trees
    - each node contains a value
    - the left and right sub-trees are *binary trees*

…Tree Terminolgy

- Level of a node in a tree (or depth) is one higher than the level of its parent
  - Depth of the root is 0
- We call the length of the longest path from the root to a node the height of a tree



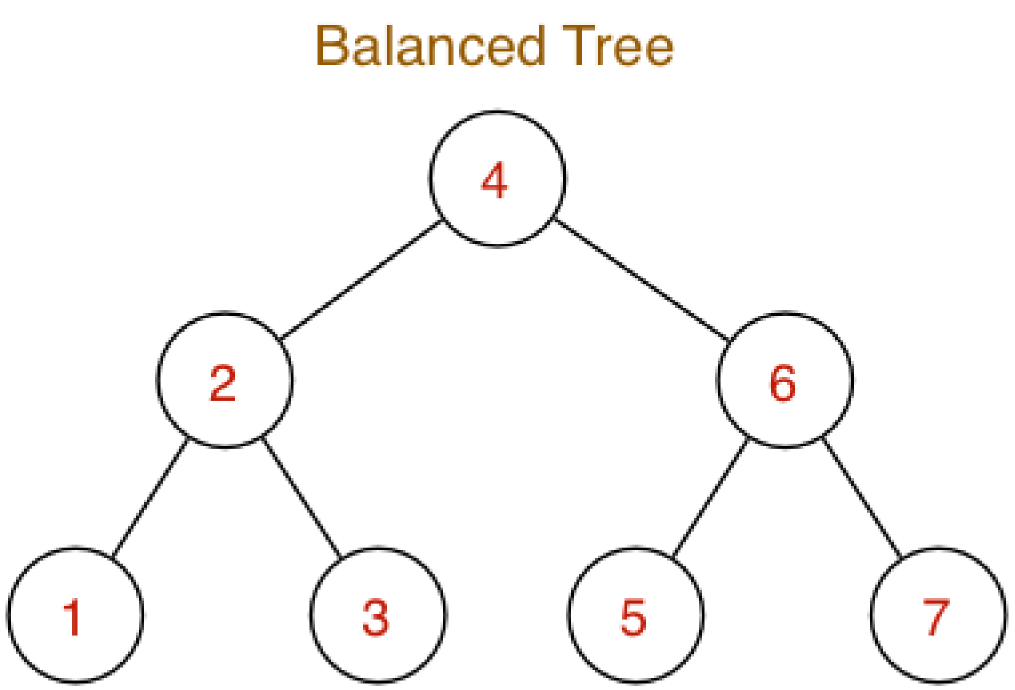
Binary Trees: Properties

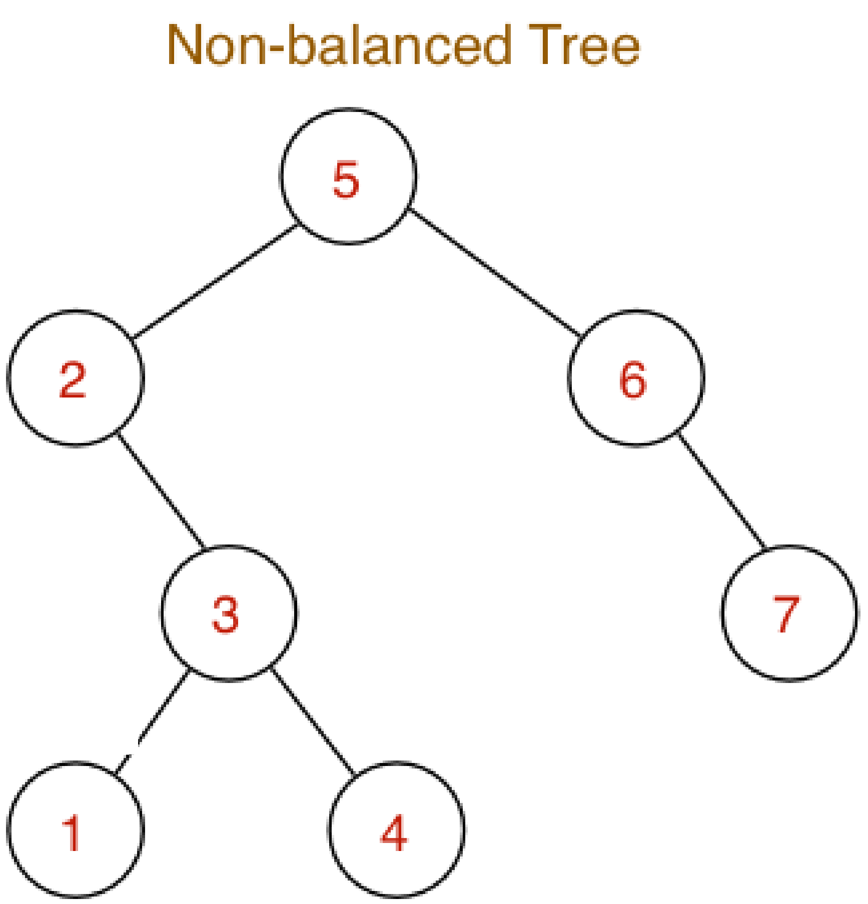
- A binary tree with n nodes has a height of
  - at most
    - n-1 (if degenerate) (an unbalanced tree, where for each parent node, there is only one child node)
  - at least
    - floor($\log_2$(n)) (if balanced)

These properties are important to estimate the runtime complexity of tree algorithms!

BINARY SEARCH TREE (BST)

- A BST is a binary tree that has:
  - all values in left sub-tree being less than root
  - all values in right sub-tree are greater than root
  - this property applies over all nodes in the tree
- Shape of tree is determined by the order of insertion





Exercise: Insertion into BSTs

- For each of the sequences below start from an initially empty binary search tree
  - show the tree resulting from inserting the values in the order given
  - What is the height of each tree?
- (a) 4 2 6 5 1 7 3
- (b) 5 3 6 2 4 7 1
- (c) 1 2 3 4 5 6 7

Binary Trees in C

A binary tree is a generalization of a linked list:

- nodes are a structure with two links to nodes
- empty trees are NULL links

```
typedef struct treenode *Treelink;

struct treenode {
   int data;
   Treelink left, right;
}
```

SEARCHING IN BSTS

- Recursive version

```
// Returns non-zero if item is found,
// zero otherwise
int search(TreeLink  n, Item i){
    int result;
    if(n == NULL){
      result =  0;
    }else if(i < n->data){
       result = search(n->left,i);
    }else if(i > n->data)
       result = search(n->right,i);
    }else{ // you found the item
      result = 1;
    }
    return result;
}
```

* Exercise: Try writing an iterative version

Insertion into a BST

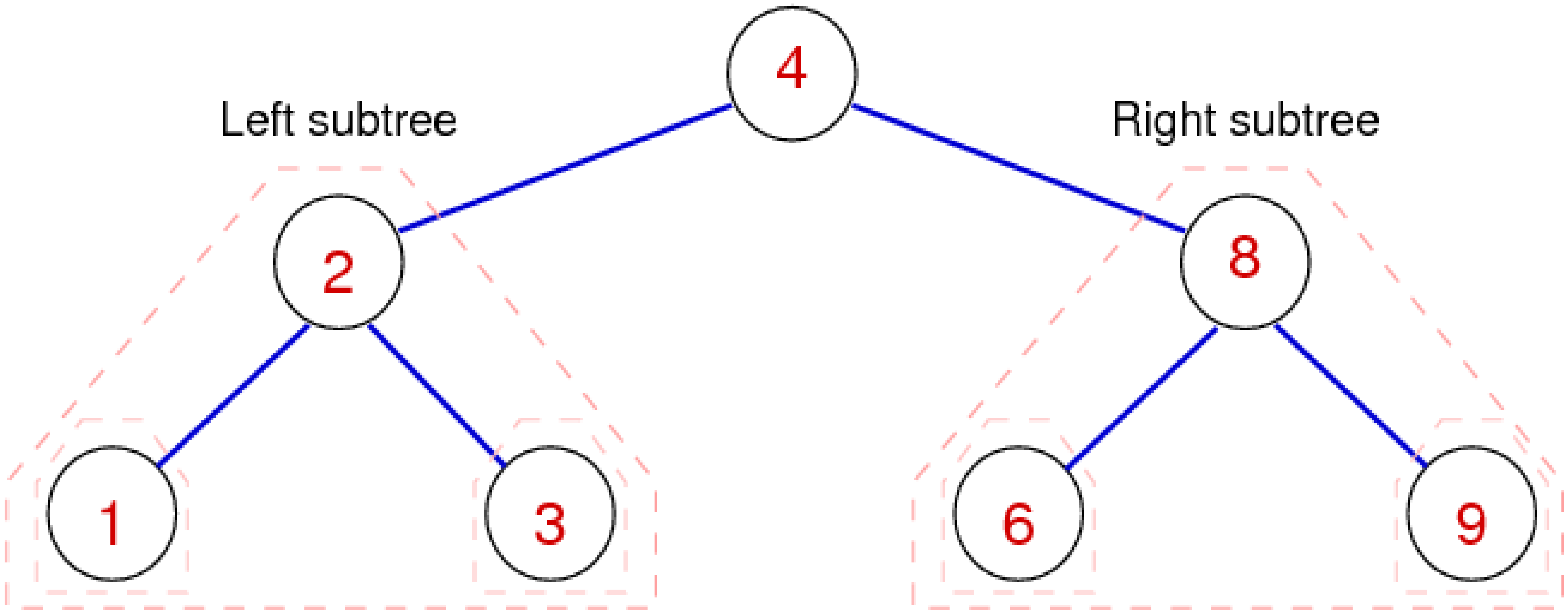
- Cases for inserting value V into tree T:
  - T is empty, make new node with V as root of new tree
  - root node contains V, tree unchanged (no dups)
  - V < value in root, insert into left subtree (recursive)
  - V > value in root, insert into right subtree (recursive)
- Non-recursive insertion of V into tree T:
  - search to location where V belongs, keeping parent
  - make new node and attach to parent
  - whether to attach L or R depends on last move

Binary Trees: Traversal

- For trees, several well-defined visiting orders exist:
  - Depth first traversals
    - preorder (NLR) ... visit root, then left subtree, then right subtree
    - inorder (LNR) ... visit left subtree, then root, then right subtree
    - postorder (LRN) ... visit left subtree, then right subtree, then root
  - Breadth-first traversal or level-order ... visit root, then all its children, then all their children

Example of traversals on a Binary tree

- Pre-Order: 4 2 1 3 8 6 9
- In-Order: 1 2 3 4 6 8 9
- Post-Order 1 3 2 6 9 8 4
- Level-Order: 4 2 8 1 3 6 8



Deletion from BSTs

- Insertion into a binary search tree is easy:
  - find location in tree where node to be added
  - create node and link to parent
- Deletion from a binary search tree is harder:
  - find the node to be deleted and its parent
  - unlink node from parent and delete
  - replace node in tree by ... ???

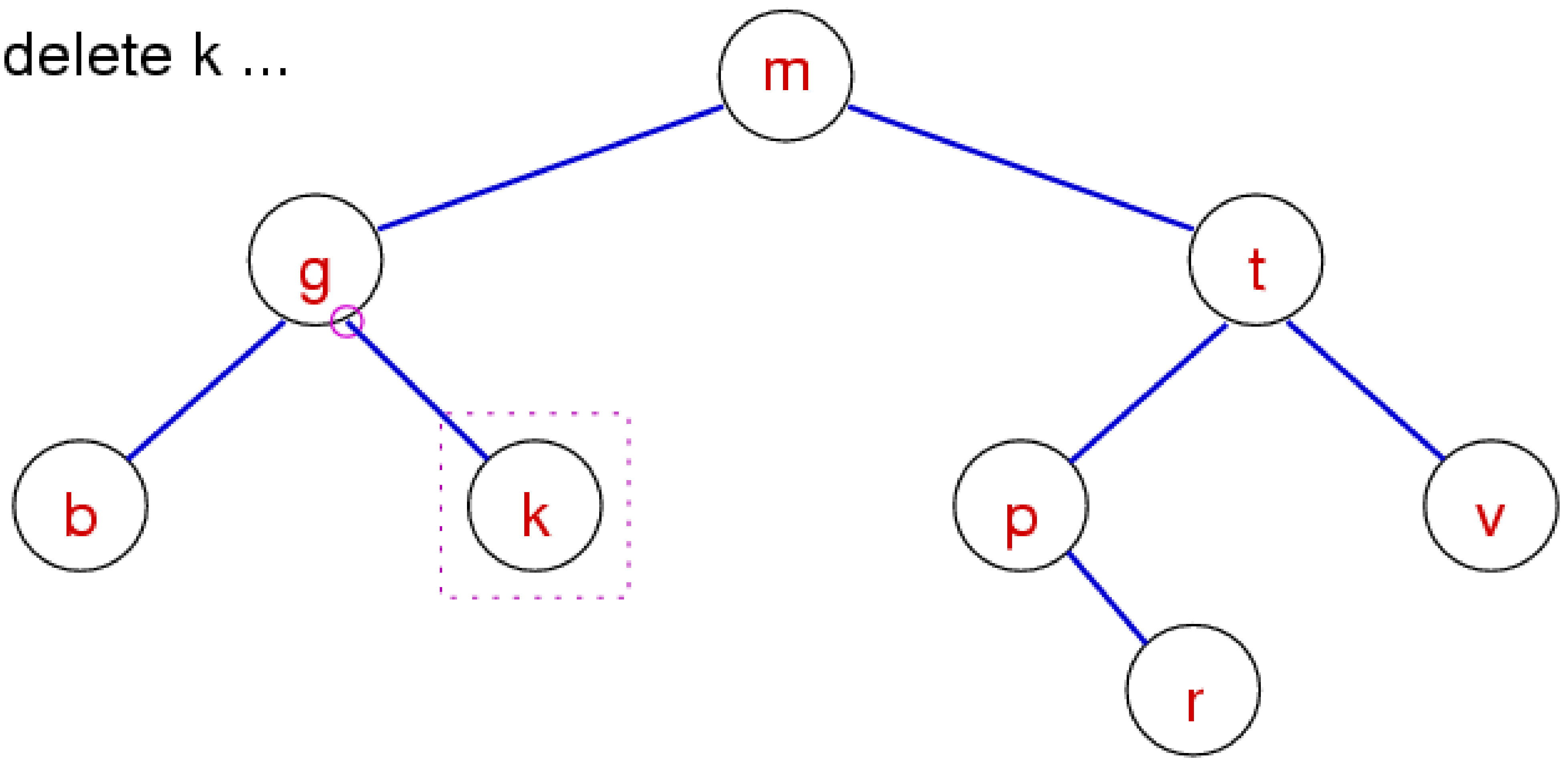
Deletion from BSTs…

- Easy option ... don't delete; just mark node as deleted
  - future searches simply ignore marked nodes
- If we want to delete, three cases to consider ...
  - zero subtrees ... unlink node from parent
  - one subtree ... replace node by child
  - two subtrees ... two children; one link in parent

Deletion from BSTs

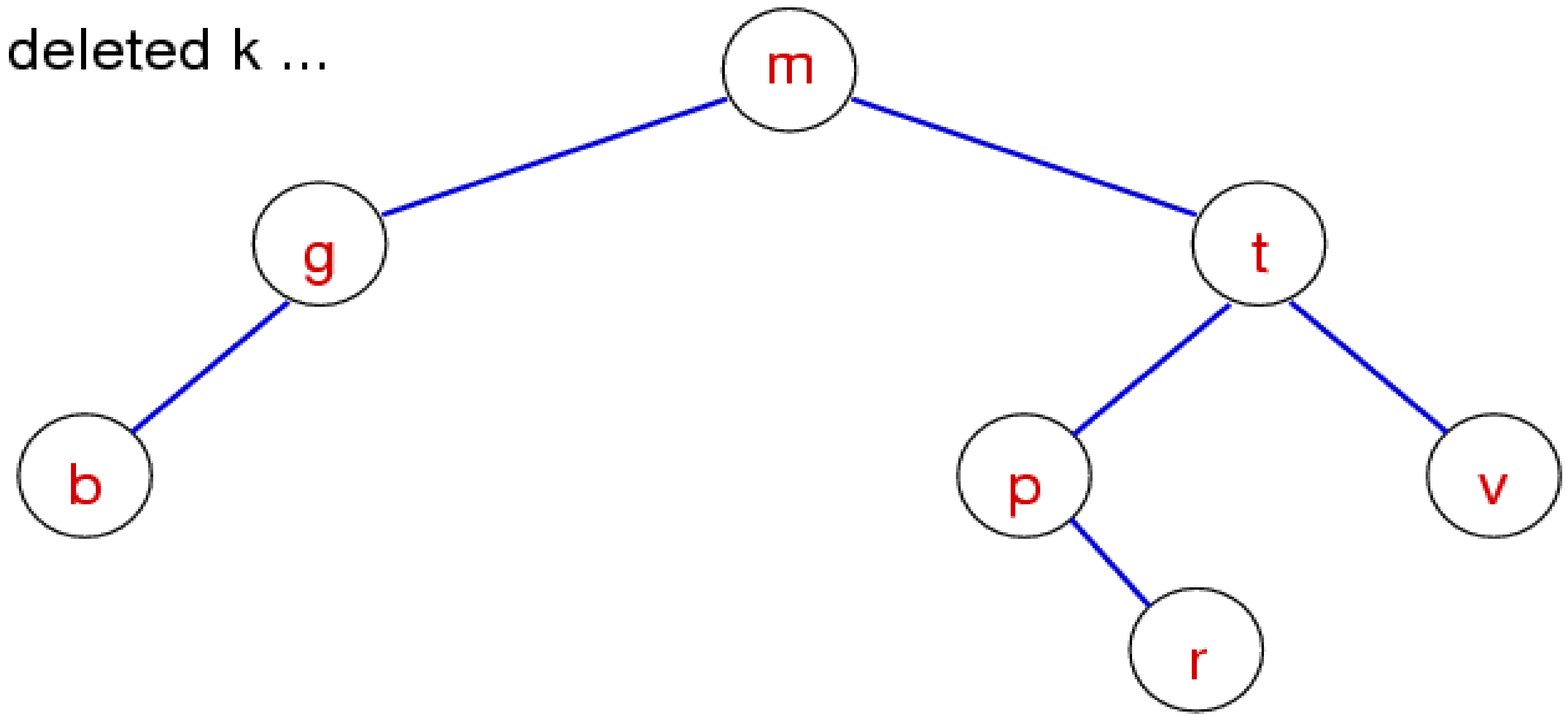
- Case 1: value to be deleted is a leaf (zero subtrees)

delete k ...



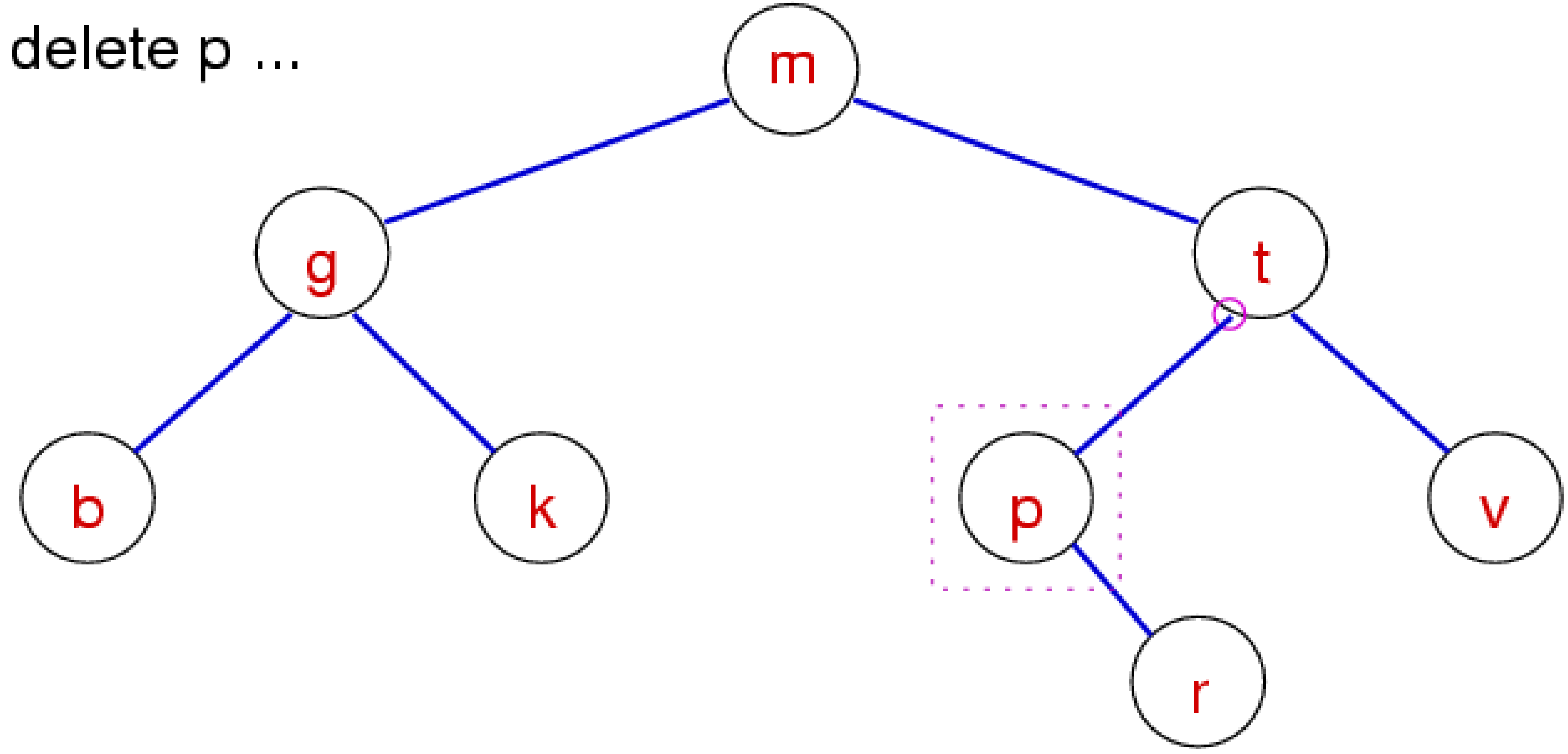
Deletion from BSTs

- Case 1: value to be deleted is a leaf (zero subtrees)



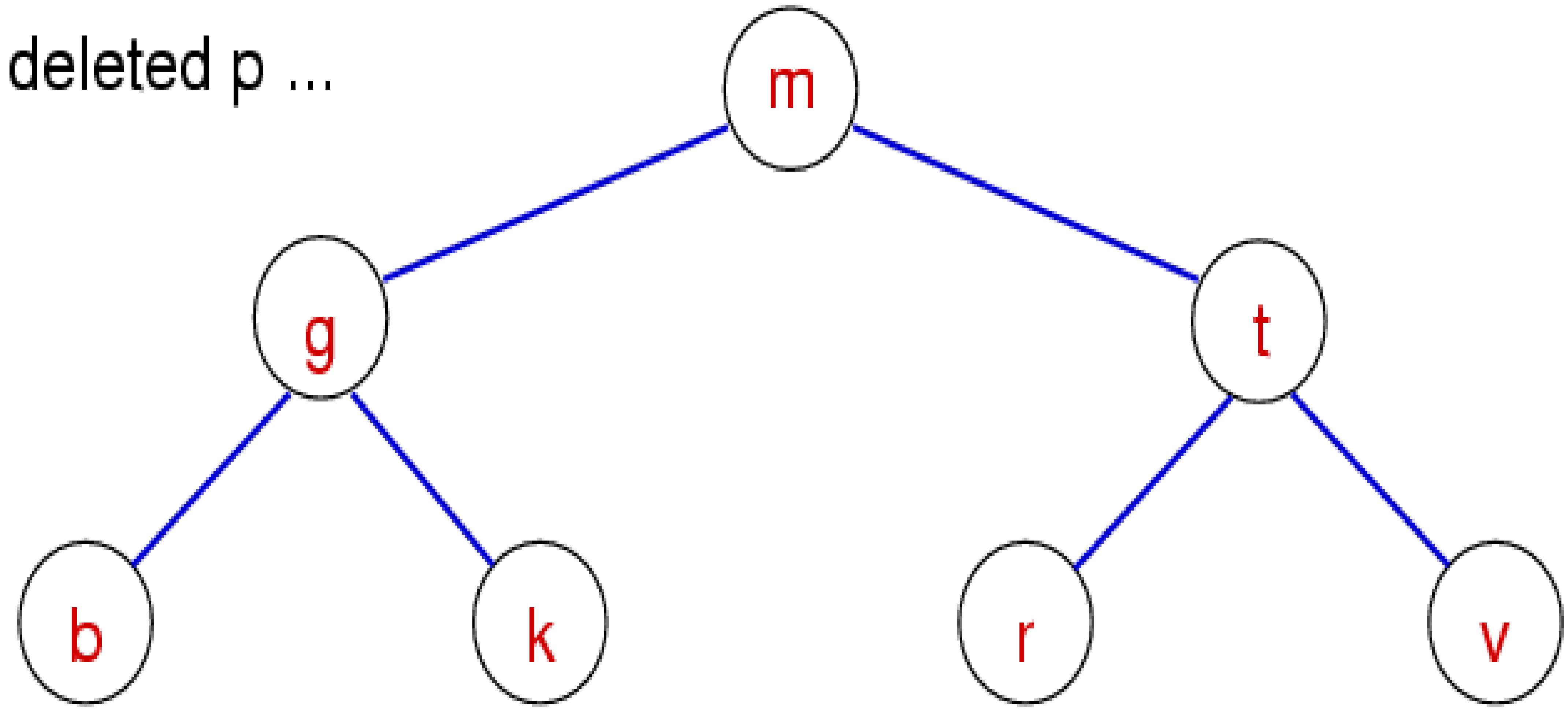
Deletion from BSTs

- Case 2: value to be deleted has one subtree



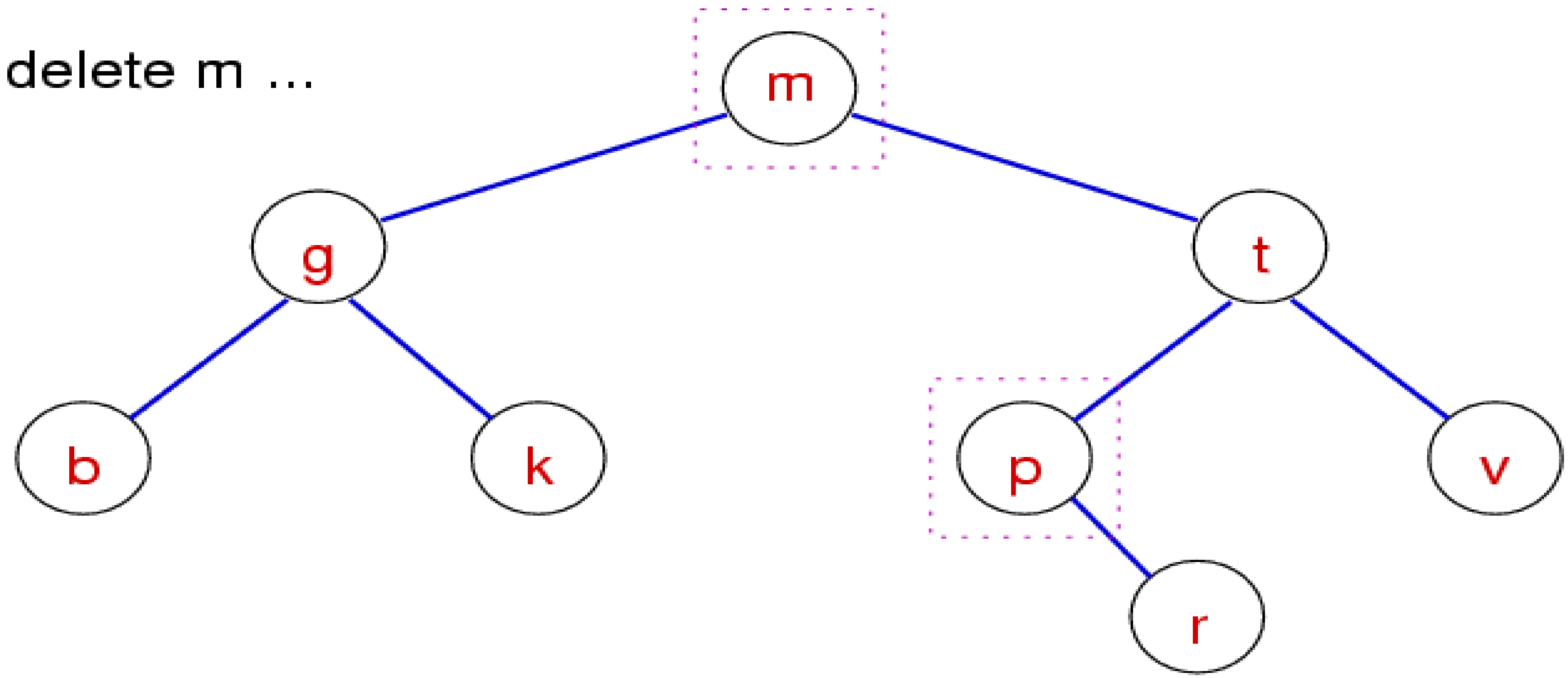
Deletion from BSTs

- Case 2: value to be deleted has one subtree



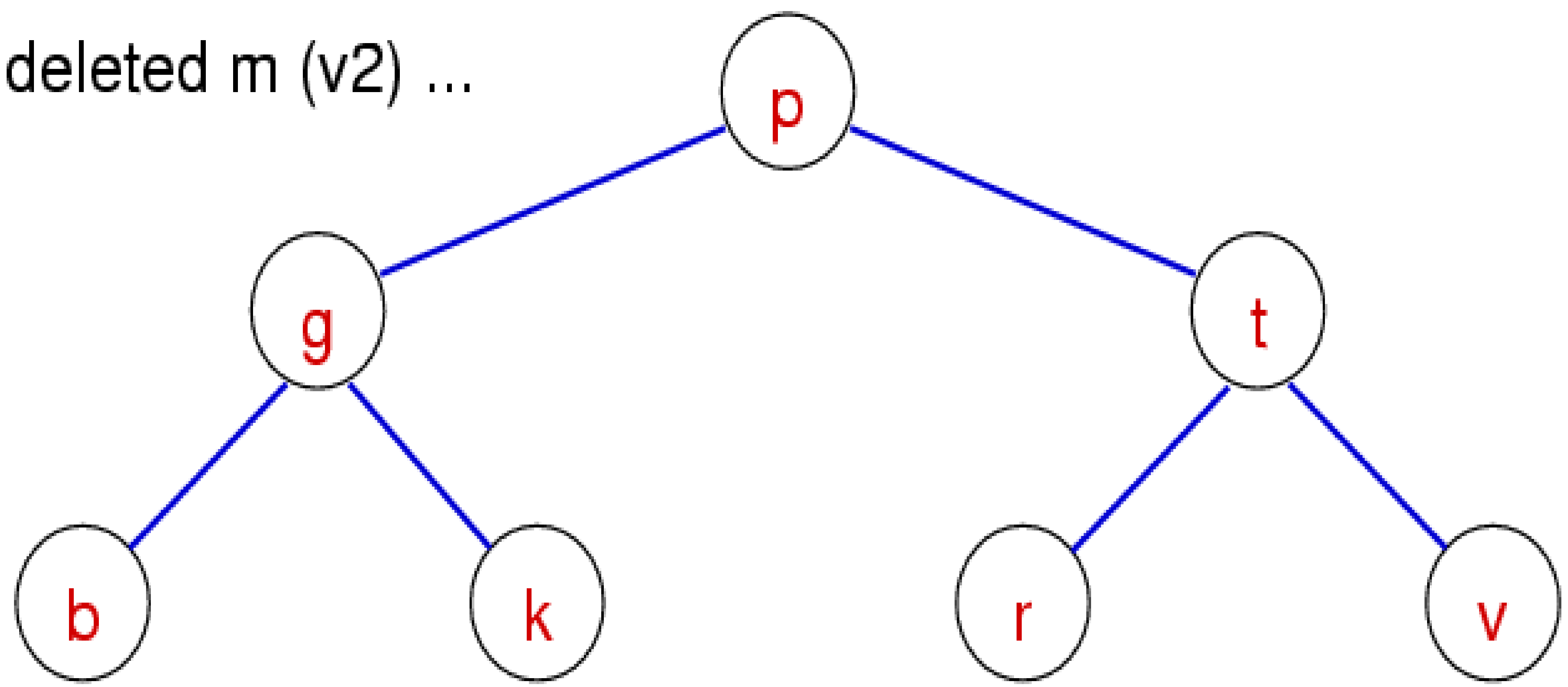
DELETION FROM BSTS

- Case 3a: value to be deleted has two subtrees
- Replace deleted node by its immediate successor
  - The smallest (leftmost) node in the right subtree



Deletion from BSTs

- Case 3a: value to be deleted has two subtrees



Binary Search Tree Properties

- Cost for searching/deleting:
  - Worst case: key is not in BST – search the height of the tree
    - Balanced trees – $O(\log_2 n)$
    - Degenerate trees – $O(n)$
- Cost for insertion:
  - Always traverse the height of the tree
    - Balanced trees – $O(\log_2 n)$
    - Degenerate trees – $O(n)$