Graphs and Trees

General, DFS, BFS, BST, Heaps, AVL, 2-3 and BTrees

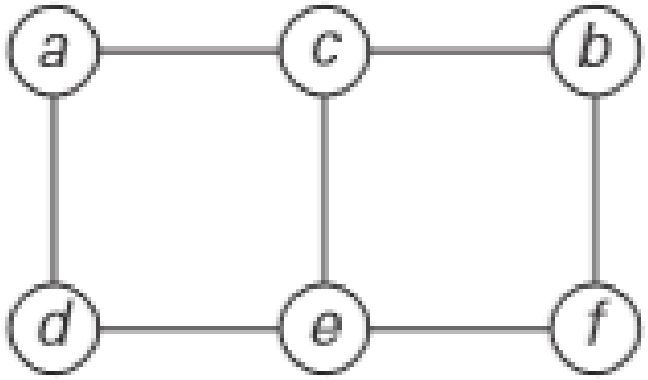
Graphs general

- Informally, a ***graph*** can be thought of as a collection of points called vertices, some of which are connected by line segments called edges.
- Graphs can be used for modeling a wide variety of applications, including transportation, communication, social and economic networks, project scheduling, and games.
- Basic graph algorithms include graph-traversal algorithms, shortest-path algorithms, and topological sorting for graphs with directed edges

Graph Representations

1. Adjacency matrix: of a graph with n vertices is an n×n Boolean matrix with one row and one column for each of the graph's vertices, in which the element in the ith row and the jth column is equal to 1 if there is an edge from the ith vertex to the jth vertex, and equal to 0 if there is no such edge
2. Adjacency lists: is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex (i.e., all the vertices connected to it by an edge). Usually, such lists start with a header identifying a vertex for which the list is compiled

Example



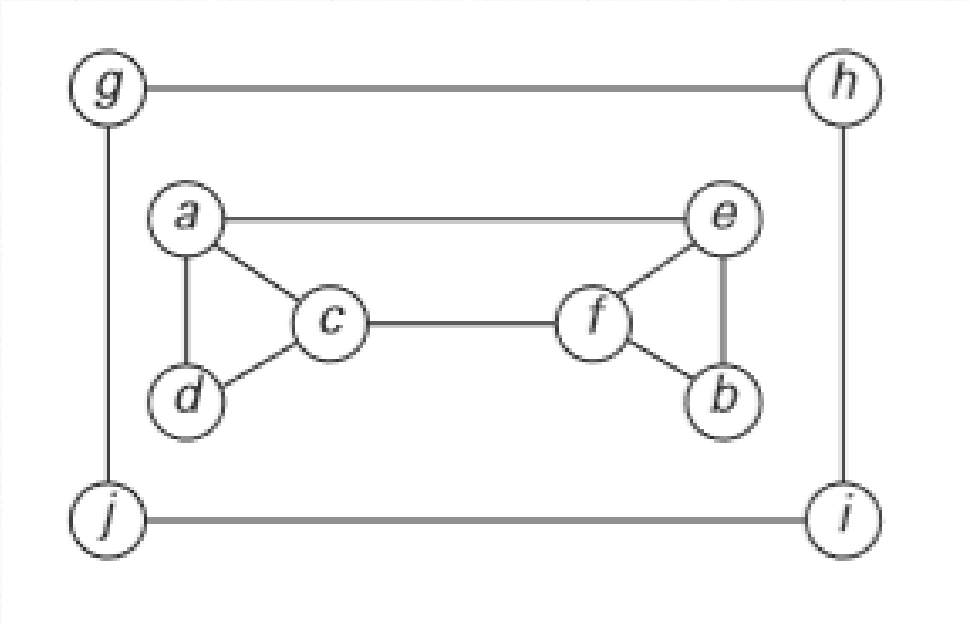
	a	b	c	d	e	f
a	0	0	1	1	0	0
b	0	0	1	0	0	1
c	1	1	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	1	0	1
f	0	1	0	0	1	0

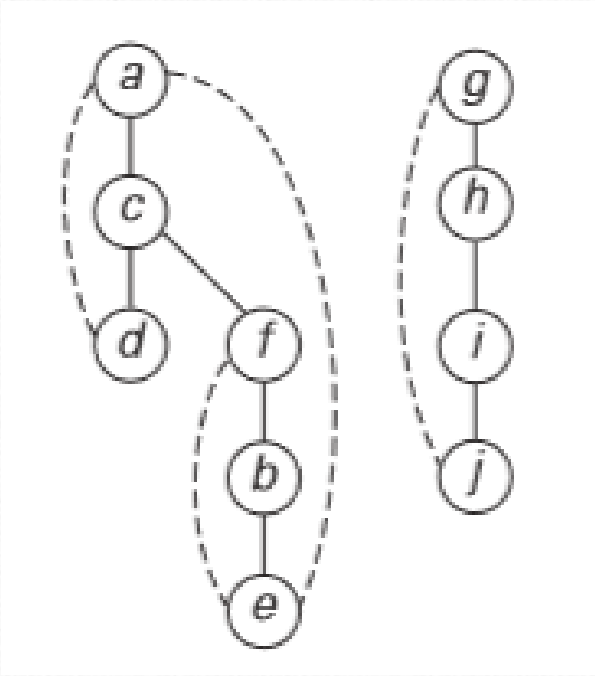
a	→	c	→	d		
b	→	c	→	f		
c	→	a	→	b	→	e
d	→	a	→	e		
e	→	c	→	d	→	f
f	→	b	→	e		

Depth-First Search

- starts a graph's traversal at an arbitrary vertex by marking it as visited
- on each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. (If there are several such vertices, a tie can be resolved arbitrarily. As a practical matter, which of the adjacent unvisited candidates is chosen is dictated by the data structure representing the graph.)
- Continues until a dead end—a vertex with no adjacent unvisited vertices— is encountered.
- at a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there.
- The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end. By then, all the vertices in the same connected component as the starting vertex have been visited. If unvisited vertices still remain, the depth-first search must be restarted at any one of them.

Example





Algorithm

ALGORITHM *DFS*(G)

```
//Implements a depth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//        in the order they are first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)
```

```
dfs(v)
//visits recursively all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are encountered
//via global variable count
count ← count + 1;   mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)
```

Running time

- adjacency matrix representation, the traversal time is in V^2
- and for the adjacency list representation, $(|V|+|E|)$ where $|V|$ and $|E|$ are the number of the graph's vertices and edges, respectively

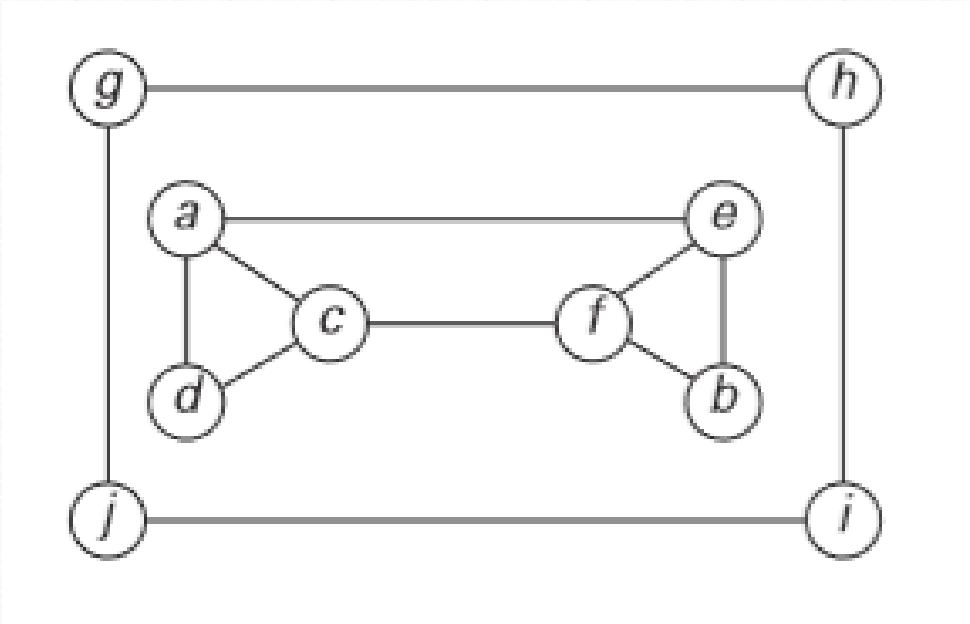
Use of DFS

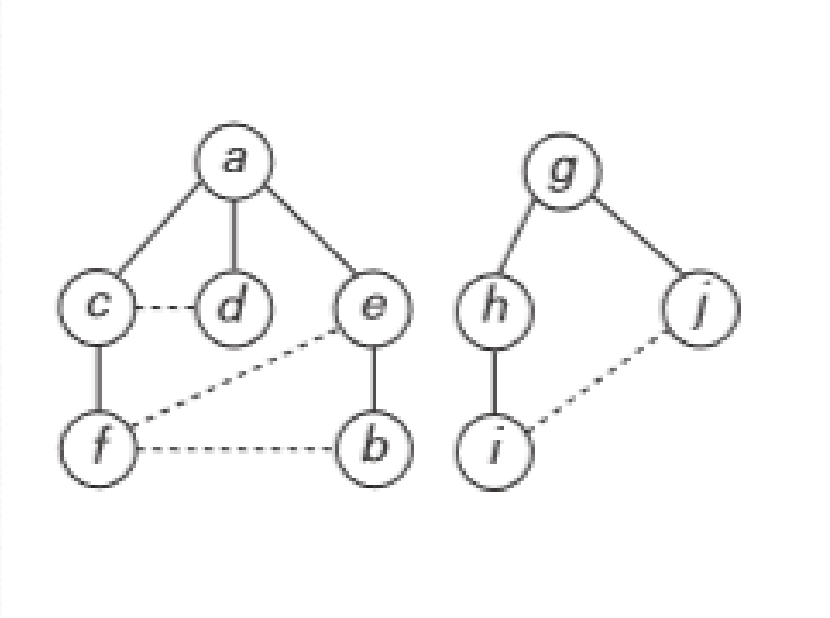
- Important elementary applications of DFS include checking connectivity and checking acyclicity of a graph.
- Since dfs halts after visiting all the vertices connected by a path to the starting vertex, checking a graph's connectivity can be done as follows:
  - Start a DFS traversal at an arbitrary vertex and check, after the algorithm halts, whether all the vertices of the graph will have been visited.
  - If they have, the graph is connected; otherwise, it is not connected. More generally, we can use DFS for identifying connected components of a graph
- As for checking for acycle presence in a graph. If the latter does not have back edges, the graph is clearly acyclic
- finding articulation points of a graph (A vertex of a connected graph is said to be its articulation point if its removal with all edges incident to it breaks the graph into disjoint pieces.)

Breadth-First Search

- visiting first all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it, and so on, until all the vertices in the same connected component as the starting vertex are visited.
- If there still remain unvisited vertices, the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.
- It is convenient to use a queue to trace the operation of breadth-first search.
- The queue is initialized with the traversal's starting vertex, which is marked as visited. On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the front vertex, marks them as visited, and adds them to the queue; after that, the front vertex is removed from the queue.

Example





Algorithm

ALGORITHM *BFS*(G)

//Implements a breadth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph G with its vertices marked with consecutive integers
// in the order they are visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count $\leftarrow 0$
for each vertex v in V **do**
  if v is marked with 0
    bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are visited
//via global variable *count*
count $\leftarrow$ *count* $+ 1$; mark v with *count* and initialize a queue with v
while the queue is not empty **do**
  for each vertex w in V adjacent to the front vertex **do**
    if w is marked with 0
      count $\leftarrow$ *count* $+ 1$; mark w with *count*
      add w to the queue
  remove the front vertex from the queue

Running time

- Breadth-first search has the same efficiency as depth-first search
- adjacency matrix representation, the traversal time is in V^2
- and for the adjacency list representation, $(|V|+|E|)$ where $|V|$ and $|E|$ are the number of the graph's vertices and edges, respectively

Min Edge Path

- BFS can be used for finding a path with the fewest number of edges between two given vertices.
- To do this, we start a BFS traversal at one of the two vertices and stop it as soon as the other vertex is reached. The simple path from the root of the BFS tree to the second vertex is the path sought.

DFS vs BFS

	DFS	BFS
Data structure	a stack	a queue
Number of vertex orderings	two orderings	one ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacency matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacency lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

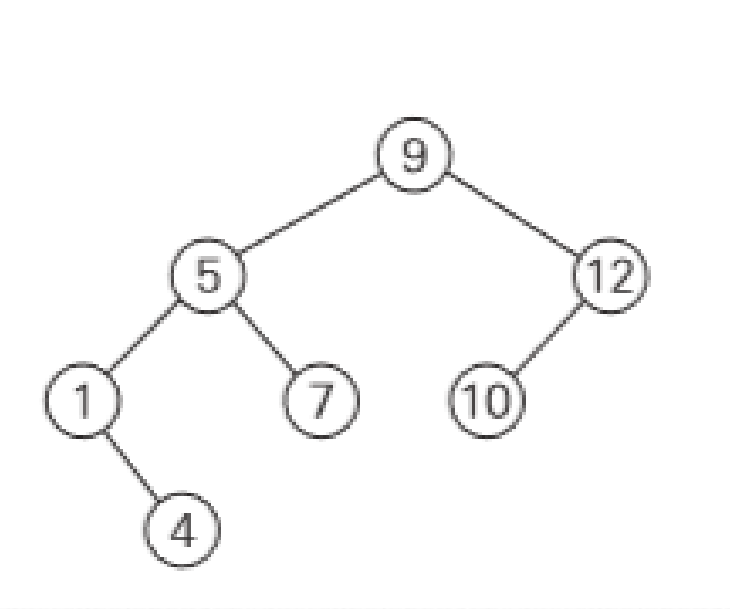
Trees

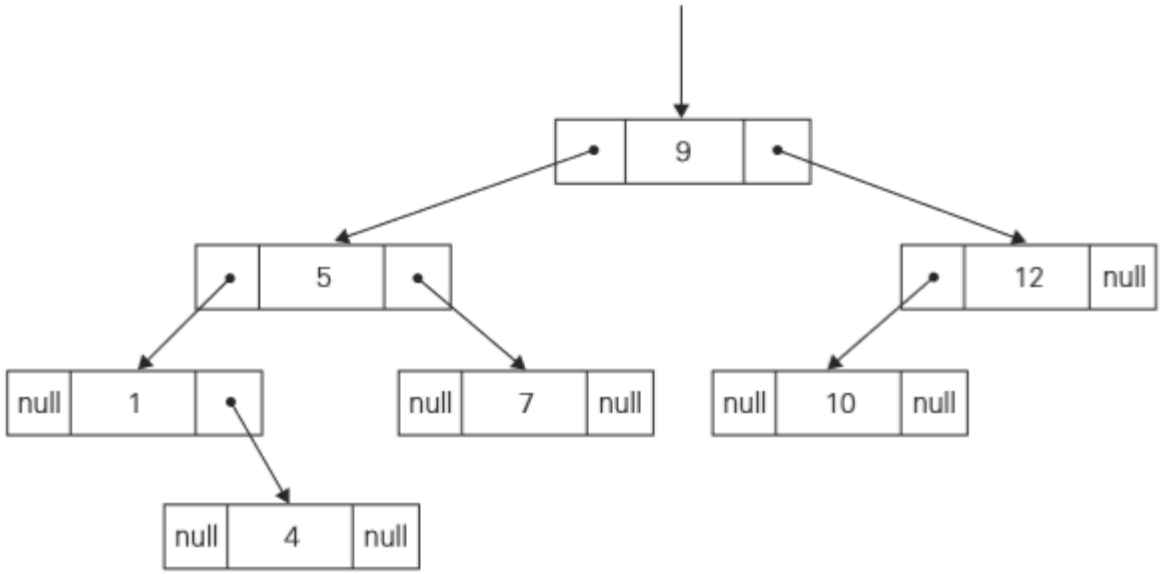
- A tree (more accurately, a free tree) is a connected acyclic graph.
- The number of edges in a tree is always one less than the number of its vertices: $|E|=|V|-1$.
- every two vertices in a tree, there always exists exactly one simple path from one of these vertices to the other. -> select an arbitrary vertex in a free tree and consider it as the root of the so-called rooted tree.
- A rooted tree is usually depicted by placing its root on the top (level 0 of the tree), the vertices adjacent to the root below it (level 1), the vertices two edges apart from the root still below (level 2), and so on.
- The depth of a vertex v is the length of the simple path from the root to v.
- The height of a tree is the length of the longest simple path from the root to a leaf.

Ordered Trees

- An ordered tree is a rooted tree in which all the children of each vertex are ordered. It is convenient to assume that in a tree's diagram, all the children are ordered left to right
- A binary tree can be defined as an ordered tree in which every vertex has no more than two children and each child is designated as either a left child or a right child of its parent;
- The binary tree with its root at the left (right) child of a vertex in a binary tree is called the left (right) subtree of that vertex. Since left and right subtrees are binary trees as well, a binary tree can also be defined recursively. This makes it possible to solve many problems involving binary trees by recursive algorithms.
- Binary search tree: a number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree

Example (9, 5, 12, 7, 10, 1, 4)





Height of a BST

- It can be computed as the maximum of the heights of the root's left and right subtrees plus 1
- Also note that it is convenient to define the height of the empty tree as −1.

ALGORITHM *Height*(T)

//Computes recursively the height of a binary tree
//Input: A binary tree T
//Output: The height of T
if $T = \varnothing$ **return** -1
else return $\max\{Height(T_{left}), Height(T_{right})\} + 1$

recurrence relation

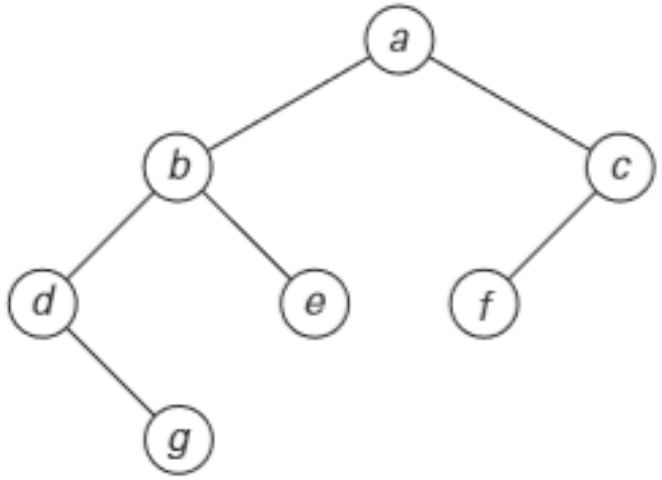
$$A(n(T)) = A(n(T_{left})) + A(n(T_{right})) + 1 \quad \text{for } n(T) > 0,$$

$$A(0) = 0.$$

Algorithms

- The most important divide-and-conquer algorithms for binary trees are the three classic traversals:

1. Preorder: the root is visited before the left and right subtrees are visited (in that order)
2. Inorder: the root is visited after visiting its left subtree but before visiting the right subtree
3. Postorder: the root is visited after visiting the left and right subtrees



preorder: *a, b, d, g, e, c, f*
inorder: *d, g, b, e, a, f, c*
postorder: *g, d, e, b, f, c, a*

How to find

- MAX
- MIN
- Spec. Element k

- Big O?

Heaps and Heapsort

- Partially ordered data structure that is especially suitable for implementing priority queues
- Recall that a priority queue is a multiset of items with an orderable characteristic called an item's priority, with the following operations:

1. finding an item with the highest (i.e., largest) priority
2. deleting an item with the highest priority
3. adding a new item to the multiset

Heap

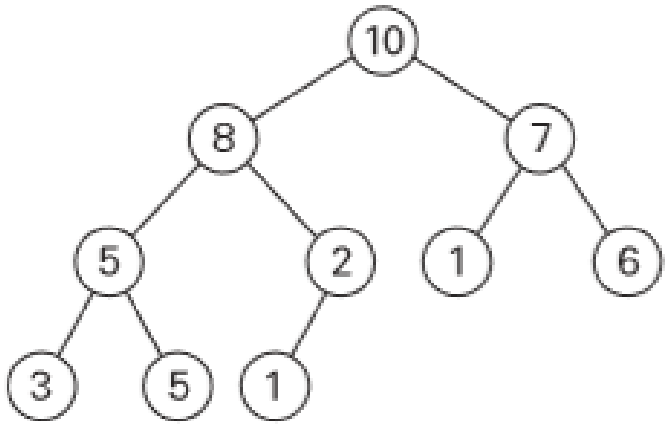
- A heap can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:

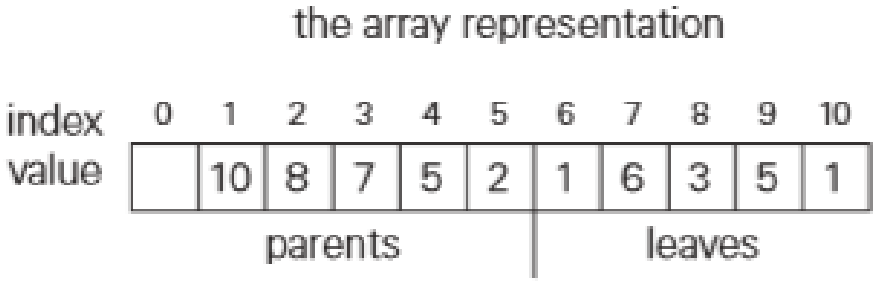
1. The shape property—the binary tree is essentially complete (or simply complete),i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
2. The parental dominance or heap property—the key in each node is greater than or equal to the keys in its children. (This condition is considered automatically satisfied for all leaves.)

Important properties

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$.
2. The root of a heap always contains its largest element.
3. A node of a heap considered with all its descendants is also a heap.
4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array, leaving $H[0]$ either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,
   a. the parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions;
   b. the children of a key in the array's parental position i $(1 \le i \le \lfloor n/2 \rfloor)$ will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position i $(2 \le i \le n)$ will be in position $\lfloor i/2 \rfloor$.

Example

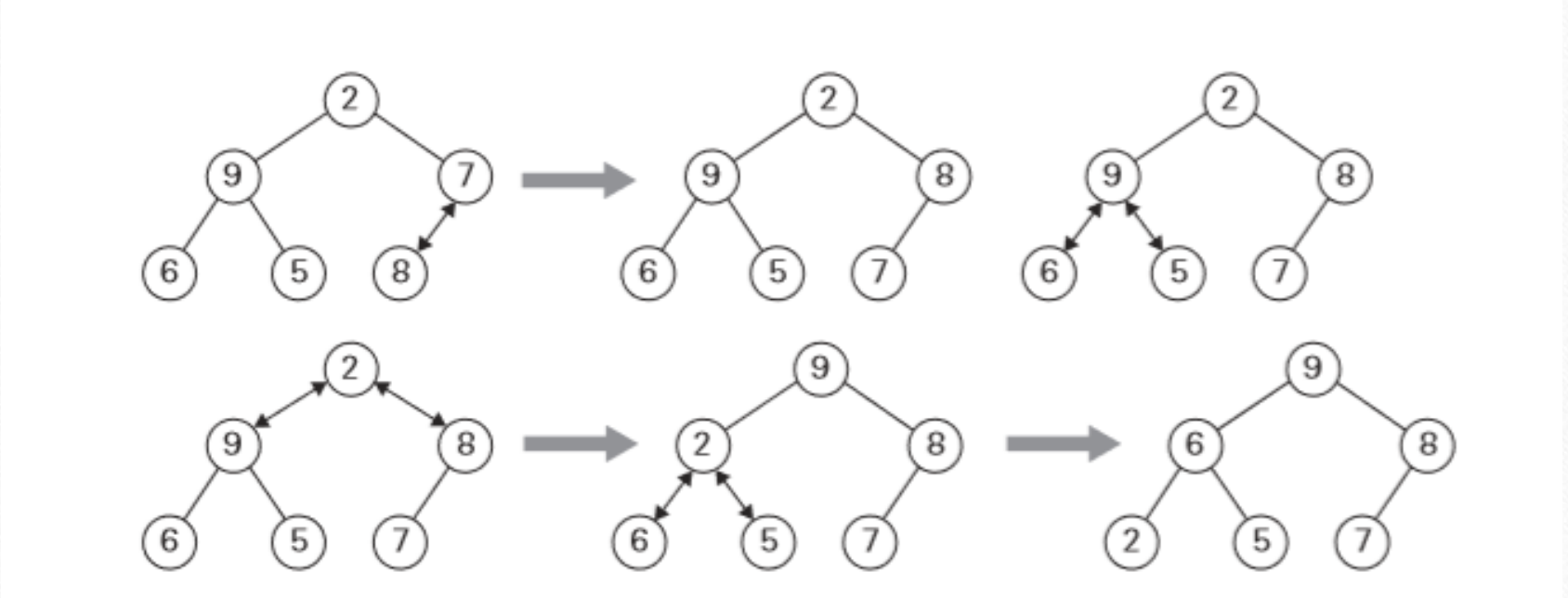

10
8
7
5
2
1
6
3
5
1


the array representation
index 0 1 2 3 4 5 6 7 8 9 10
value 10 8 7 5 2 1 6 3 5 1
parents
leaves

Create a Heap

- bottom-up heap construction algorithm
- initializes the essentially complete binary tree with n nodes by placing keys in the order given and then "heapifies" the tree as follows:
  - Starting with the last parental node, the algorithm checks whether the parental dominance holds for the key in this node.
  - If it does not, the algorithm exchanges the node's key K with the larger key of its children and checks whether the parental dominance holds for K in its new position.
  - This process continues until the parental dominance for K is satisfied.
  - After completing the "heapification" of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor. The algorithm stops after this is done for the root of the tree.

Example - Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8.

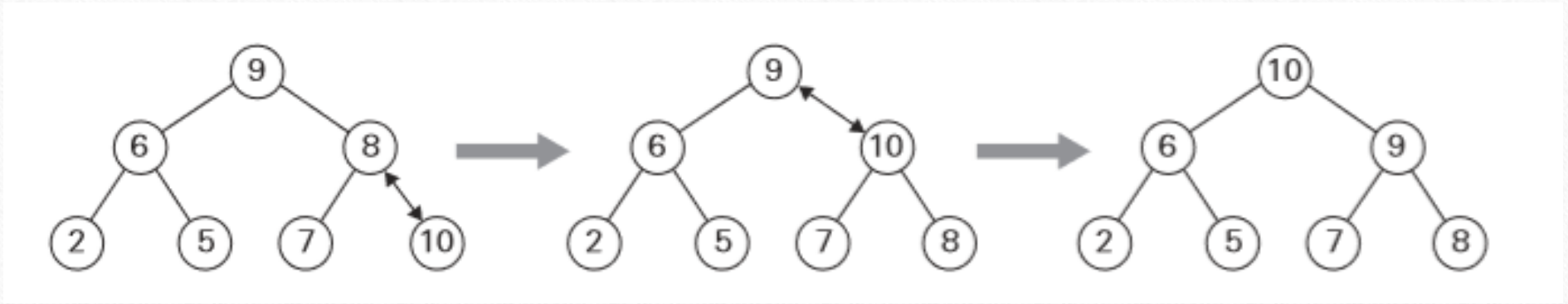


Algorithm

ALGORITHM *HeapBottomUp*$(H[1..n])$

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n    //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];    k ← j
    H[k] ← v
```

Add - 10



Deletion

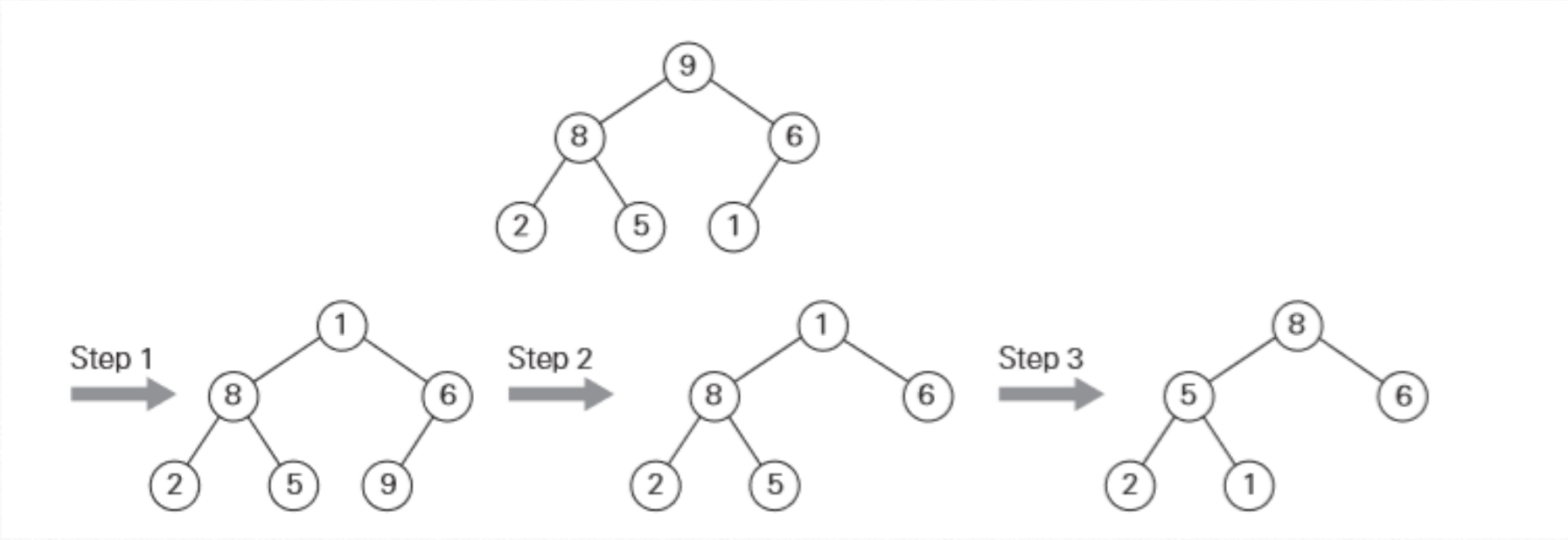
Maximum Key Deletion from a heap

Step 1 Exchange the root's key with the last key K of the heap.

Step 2 Decrease the heap's size by 1.

Step 3 "Heapify" the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for K: if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

Delete – root element



Heapsort

Stage 1 (heap construction)	Stage 2 (maximum deletions)
2 9 **7** 6 5 8	**9** 6 8 2 5 7
2 **9** 8 6 5 7	7 6 8 2 5 | **9**
2 9 8 6 5 7	**8** 6 7 2 5
9 **2** 8 6 5 7	5 6 7 2 | **8**
9 6 8 2 5 7	**7** 6 5 2
	2 6 5 | **7**
	6 2 5
	5 2 | **6**
	5 2
	2 | **5**
	2

Heapsort

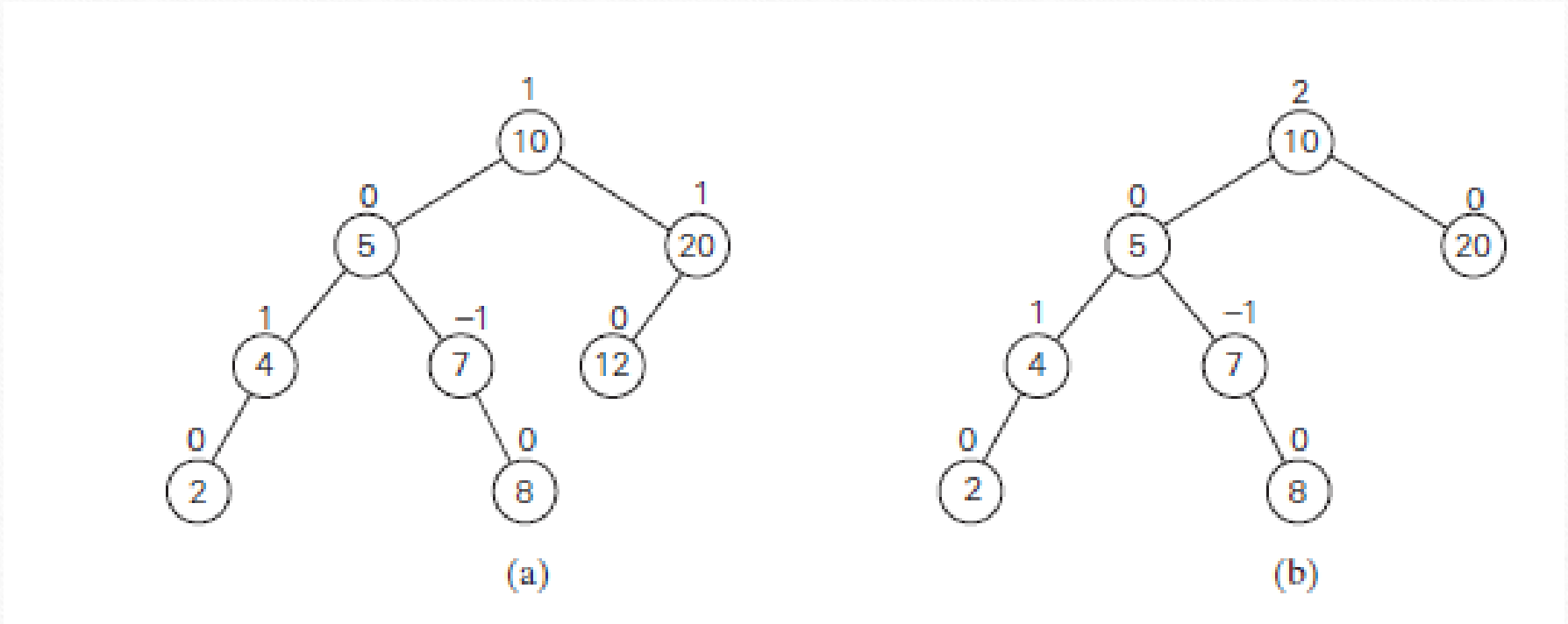
$$C(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2 \sum_{i=1}^{n-1} \log_2 i$$

$$\le 2 \sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1) \log_2(n-1) \le 2n \log_2 n.$$

AVL Trees

DEFINITION An *AVL tree* is a binary search tree in which the *balance factor* of every node, which is defined as the difference between the heights of the node's left and right subtrees, is either 0 or +1 or −1. (The height of the empty tree is defined as −1. Of course, the balance factor can also be computed as the difference between the numbers of levels rather than the height difference of the node's left and right subtrees.)

Example

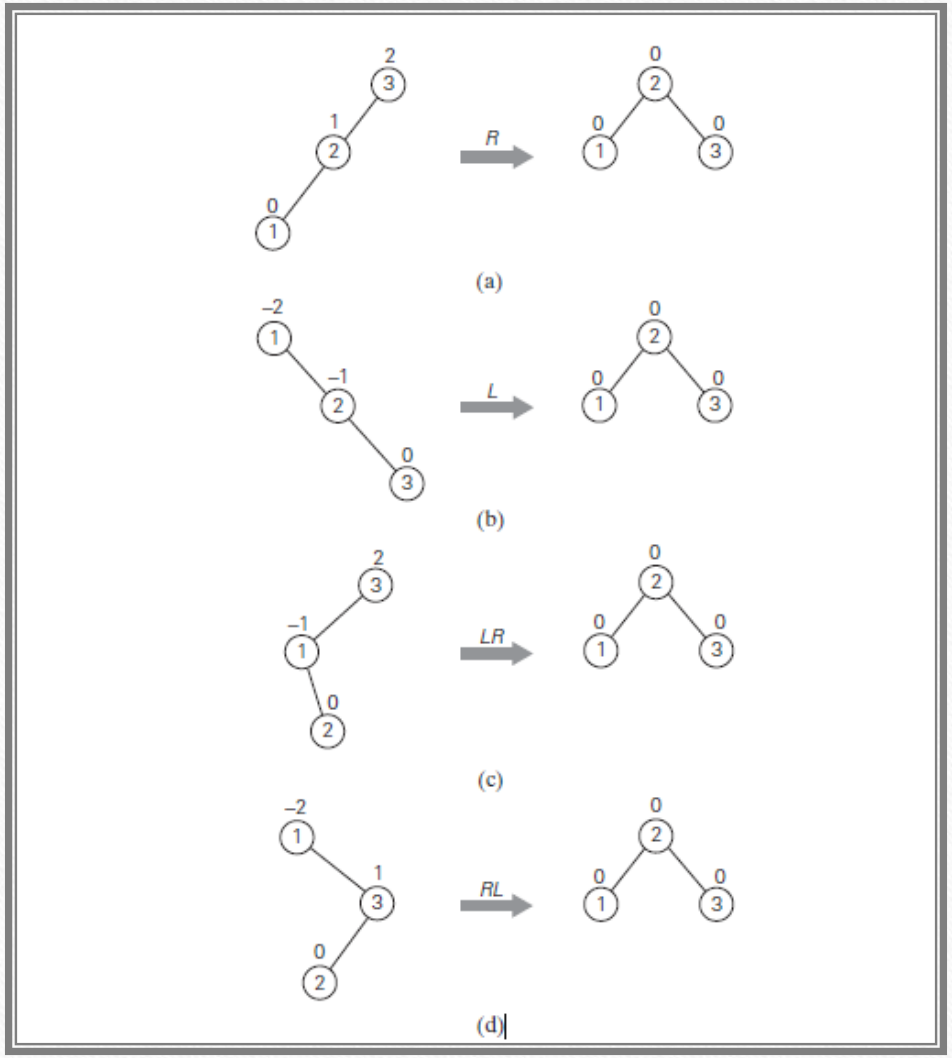


Insert of a new node

- If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by a rotation.
- A rotation in an AVL tree is a local transformation of its subtree rooted at a node whose balance has become either +2 or −2. If there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf.
- There are only four types of rotations; in fact, two of them are mirror images of the other two.

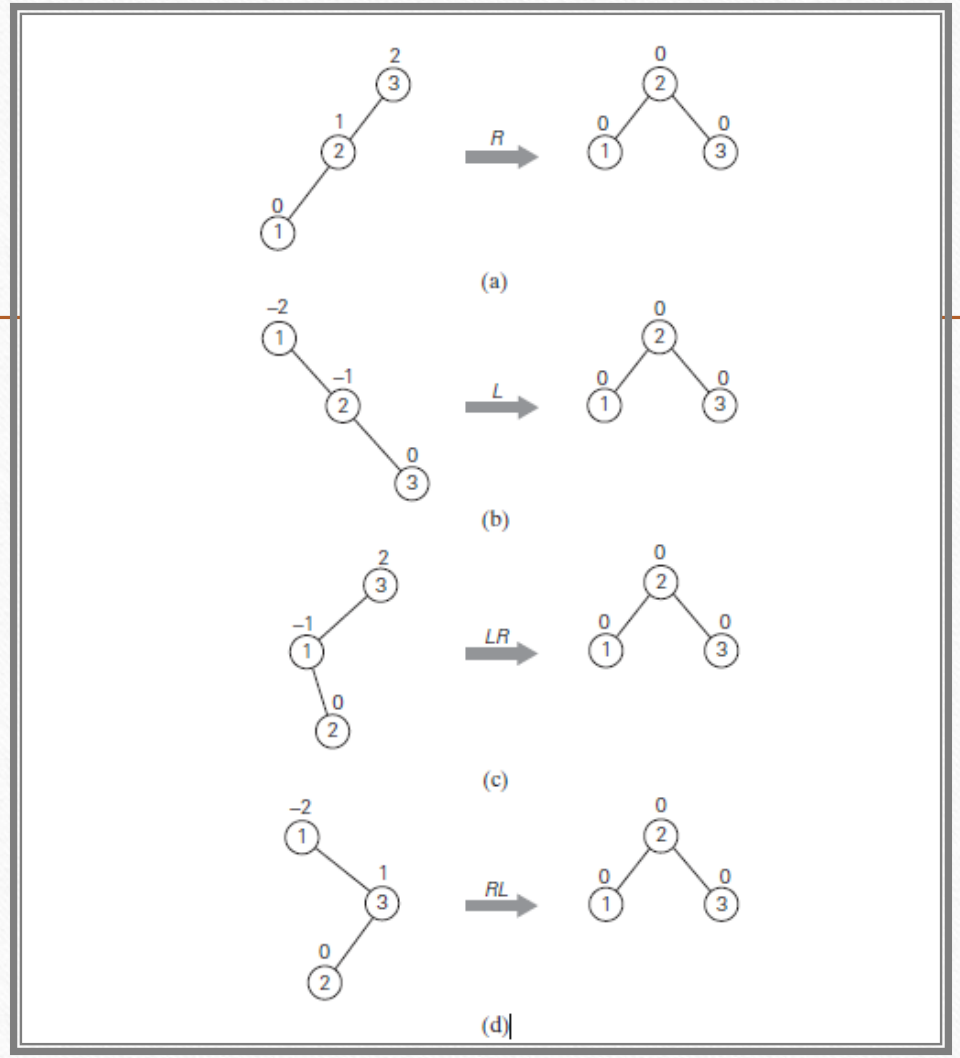
Rotations

- The first rotation type is called the single right rotation, or R-rotation. (Imagine rotating the edge connecting the root and its left child in the binary tree to the right.) Note that this rotation is performed after a new key is inserted into the left subtree of the left child of a tree whose root had the balance of +1 before the insertion.
- The symmetric single left rotation, or L-rotation, is the mirror image of the single R-rotation. It is performed after a new key is inserted into the right subtree of the right child of a tree whose root had the balance of −1 before the insertion.

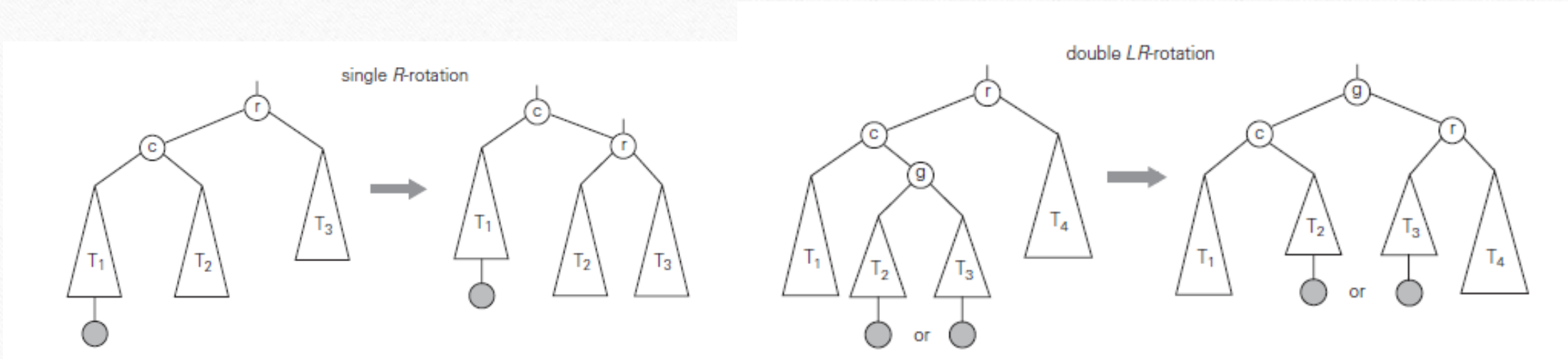


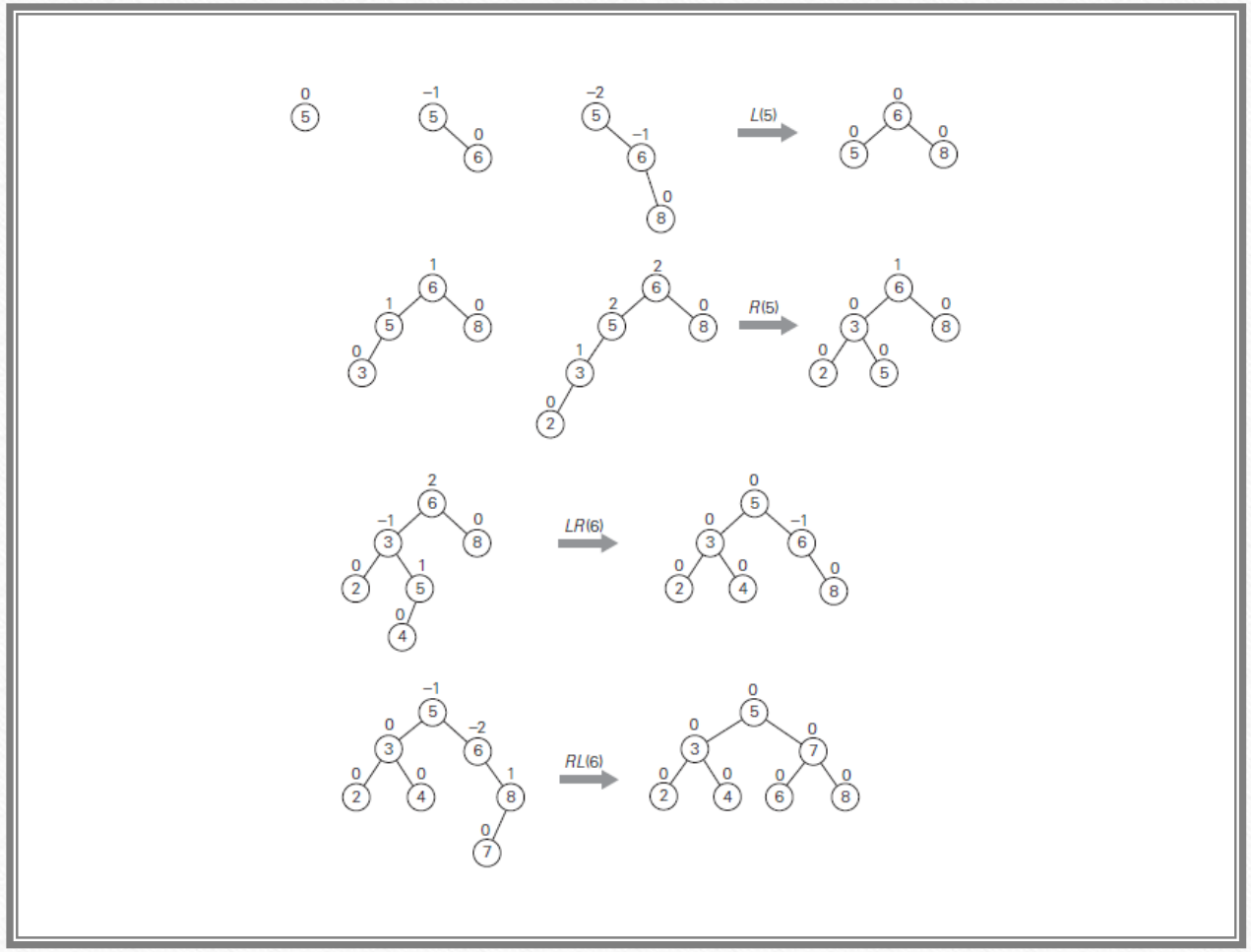
Rotations

- The second rotation type is called the ***double left-right rotation*** (***LRrotation***). It is, in fact, a combination of two rotations: we perform the *L*-rotation of the left subtree of root *r* followed by the *R*-rotation of the new tree rooted at *r*. It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.
- The ***double right-left rotation*** (***RL-rotation***) is the mirror image of the double *LR*-rotation



Single R and double LF rotation



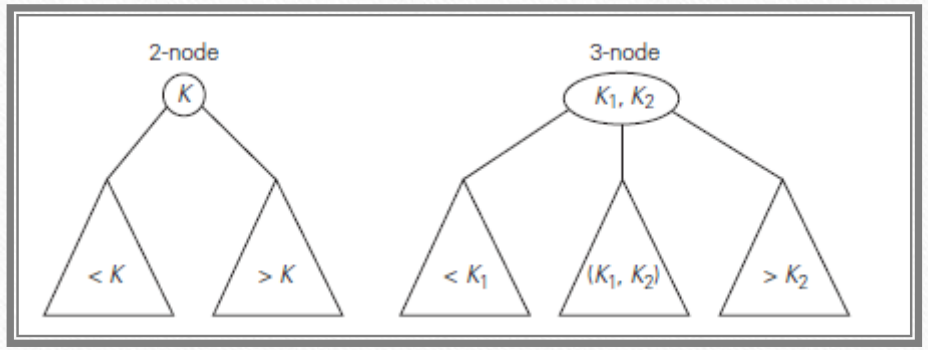


Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7

How efficient are AVL trees?

- Please read in your textbook ☺

2-3 Trees

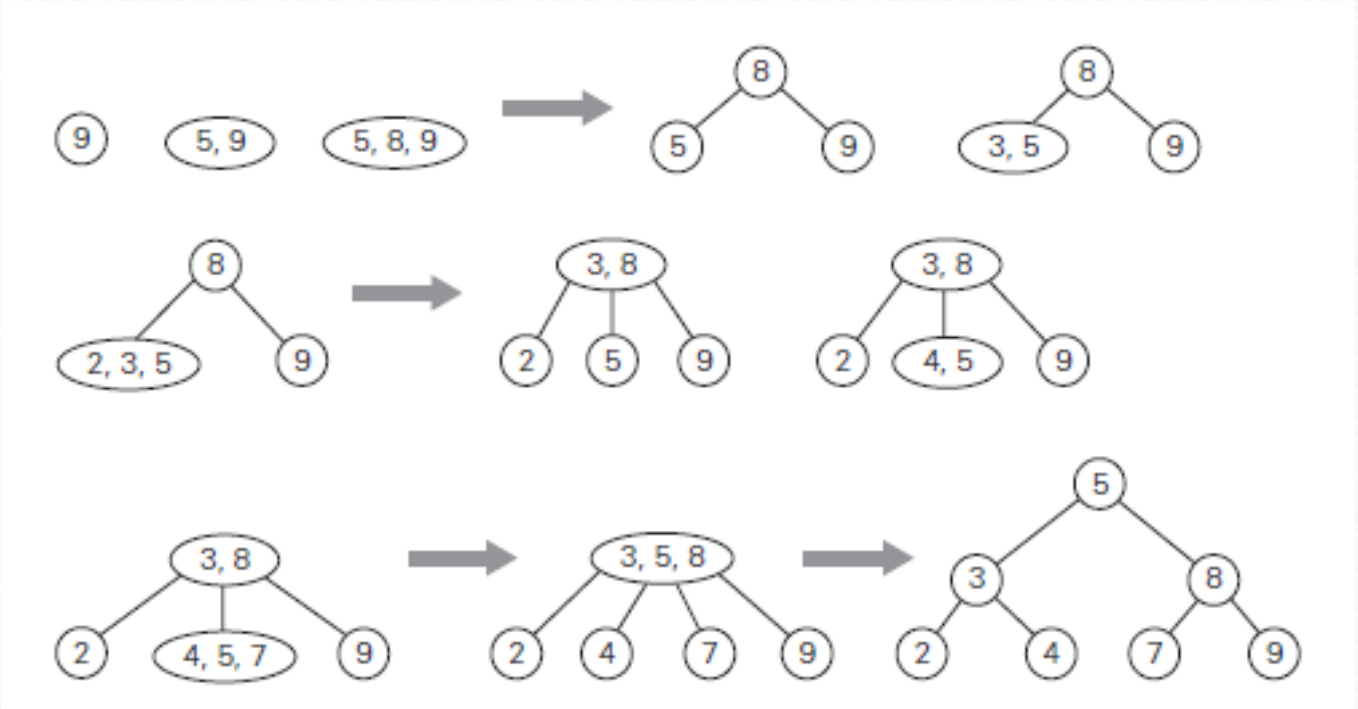


- A 2-3 tree is a tree that can have nodes of two kinds:
  - 2-nodes: contains a single key K and has two children the left child serves as the root of a subtree whose keys are less than K, and the right child serves as the root of a subtree whose keys are greater than K.
  - 3-nodes: contains two ordered keys K1 and K2 (K1<K2) and has three children. The leftmost child serves as the root of a subtree with keys less than K1, the middle child serves as the root of a subtree with keys between K1 and K2, and the rightmost child serves as the root of a subtree with keys greater than K2
- The last requirement of the 2-3 tree is that all its leaves must be on the same level. In other words, a 2-3 tree is always perfectly height-balanced: the length of a path from the root to a leaf is the same for every leaf. It is this property that we "buy" by allowing more than one key in the same node of a search tree.

Insertion

- insert a new key K in a leaf, except for the empty tree.
- The appropriate leaf is found by performing a search for K.
  - If the leaf in question is a 2-node, we insert K there as either the first or the second key, depending on whether K is smaller or larger than the node's old key.
  - If the leaf is a 3-node, we split the leaf in two: the smallest of the three keys (two old ones and the new key) is put in the first leaf, the largest key is put in the second leaf, and the middle key is promoted to the old leaf's parent. (If the leaf happens to be the tree's root, a new root is created to accept the middle key.)

Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.



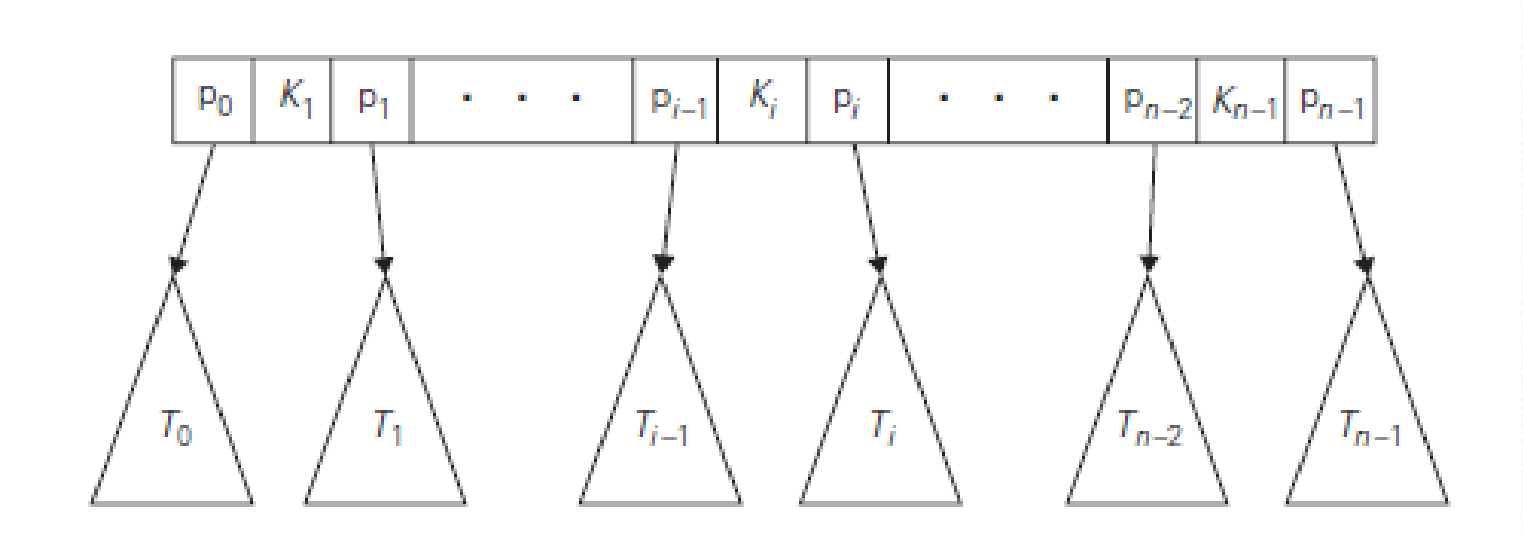
B-Trees

- extends the idea of the 2-3 tree by permitting more than a single key in the same node of a search tree
- all data records (or record keys) are stored at the leaves, in increasing order of the keys. The parental nodes are used for indexing.
- Specifically, each parental node contains n − 1 ordered keys K1 < . . .<Kn−1 assumed, for the sake of simplicity, to be distinct.
- The keys are interposed with n pointers to the node's children so that all the keys in subtree T0 are smaller than K1, all the keys in subtree T1 are greater than or equal to K1 and smaller than K2 with K1 being equal to the smallest key in T1, and so on, through the last subtree Tn−1 whose keys are greater than or equal to Kn−1 with Kn−1 being equal to the smallest key in Tn−1

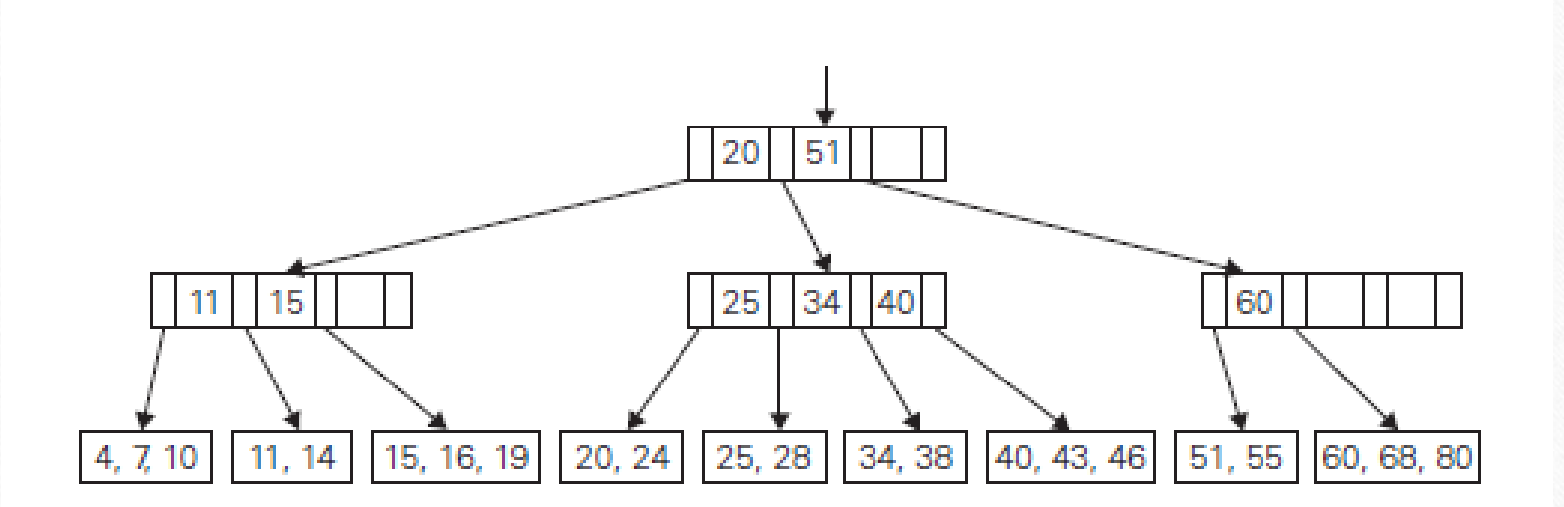
B-Trees

- In addition, a B-tree of order $m \geq 2$ must satisfy the following structural properties:
  - The root is either a leaf or has between 2 and m children.
  - Each node, except for the root and the leaves, has between $m/2$ and m children (and hence between $m/2 - 1$ and $m - 1$ keys).
  - The tree is (perfectly) balanced, i.e., all its leaves are at the same level.

B-Trees



B-Tree Example



Animations

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

Thanks