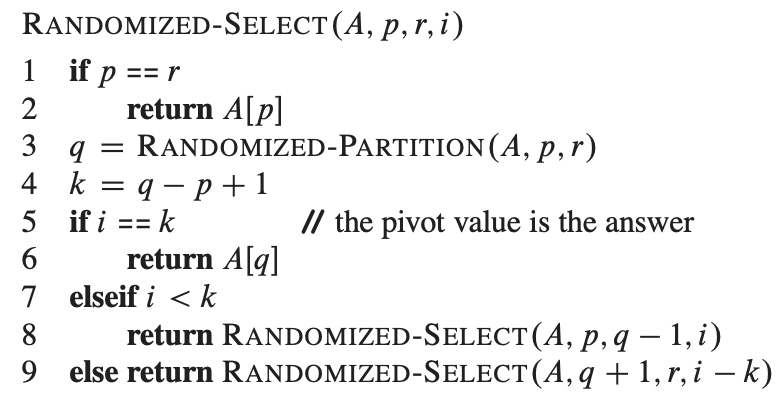
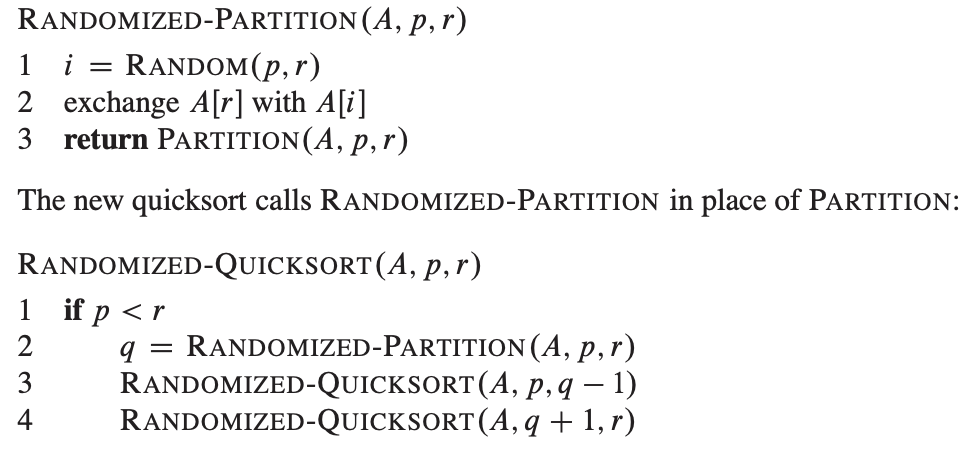
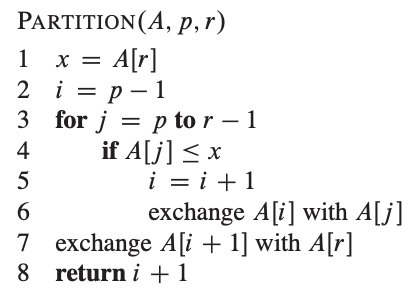
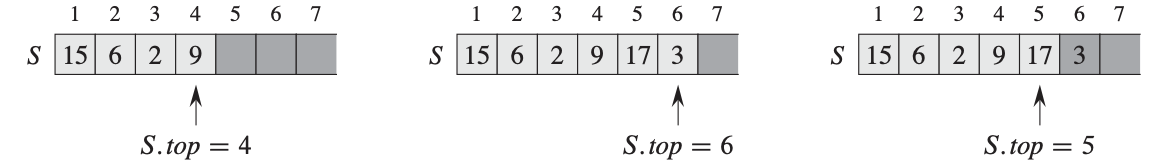
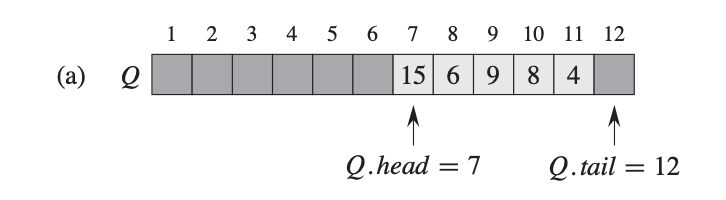
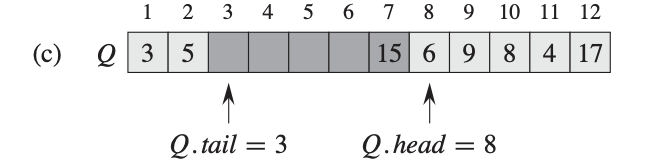
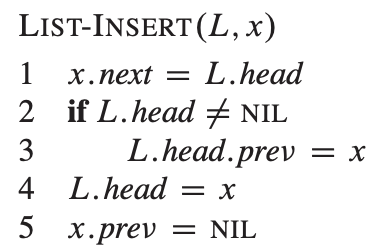
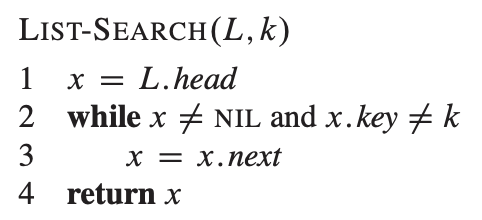
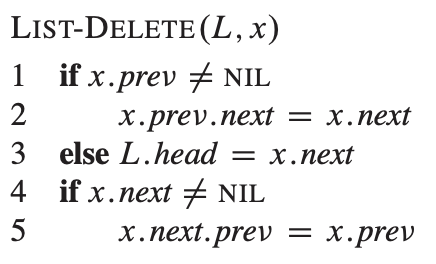
**CH 9. Medians and Order Statistics**

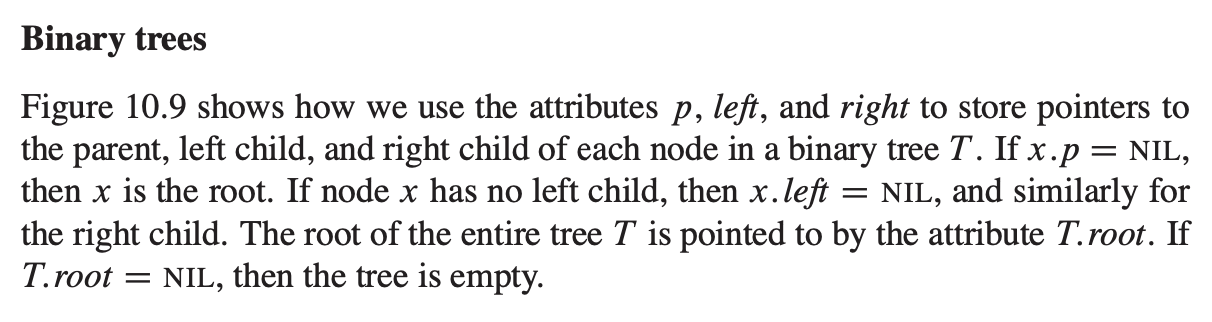
* The *i*’th ***order statistic*** of a set of n elements is the *i*’th smallest element. For example, the ***minimum*** of a set of elements is the first order statistic (*i* = 1), and the ***maximum*** is the nth order statistic (*i* = n). A ***median***, informally, is the “halfway point” of the set
* ****The following code for RANDOMIZED-SELECT returns the ith smallest element of the array A[p .. r].

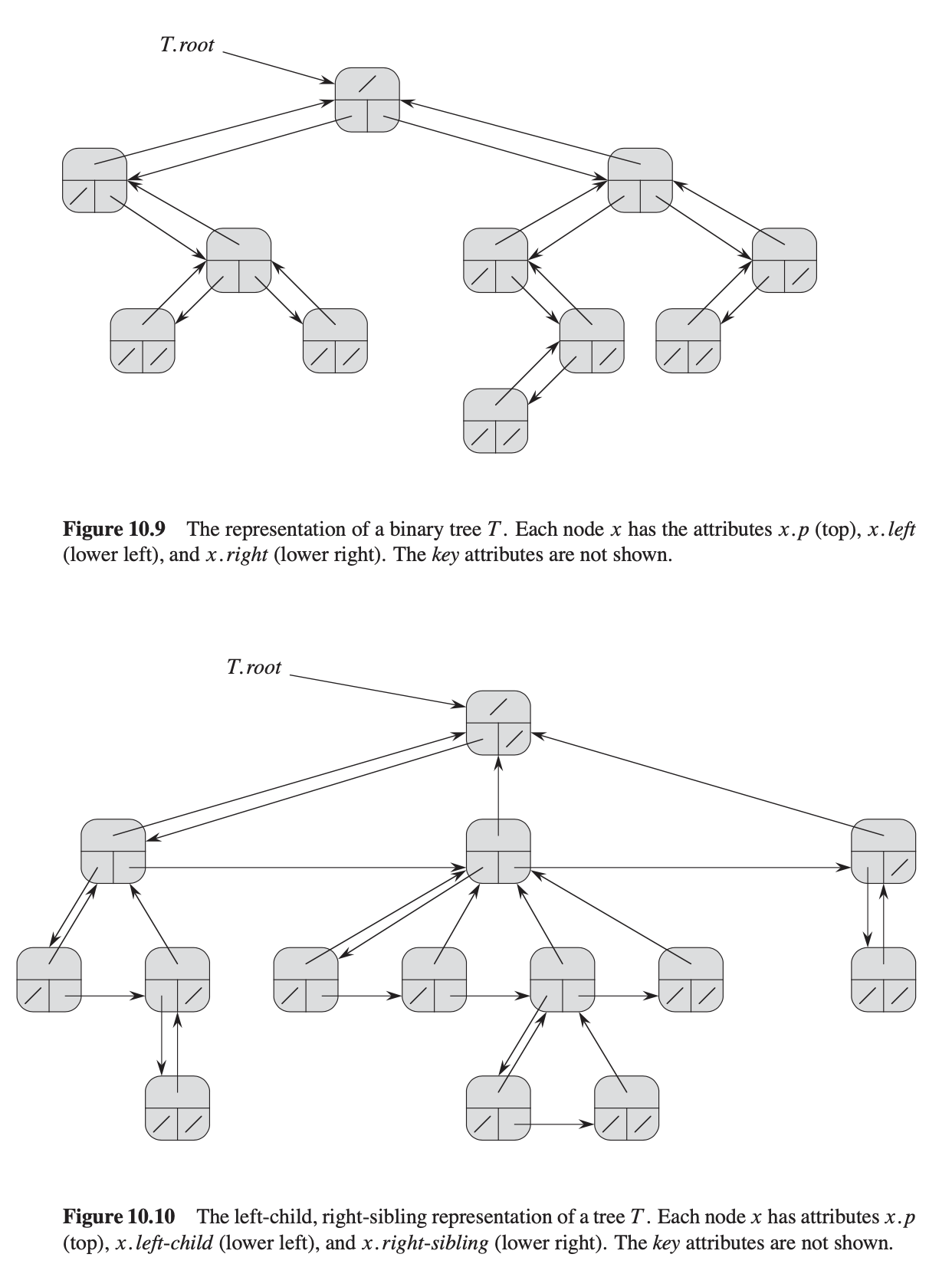
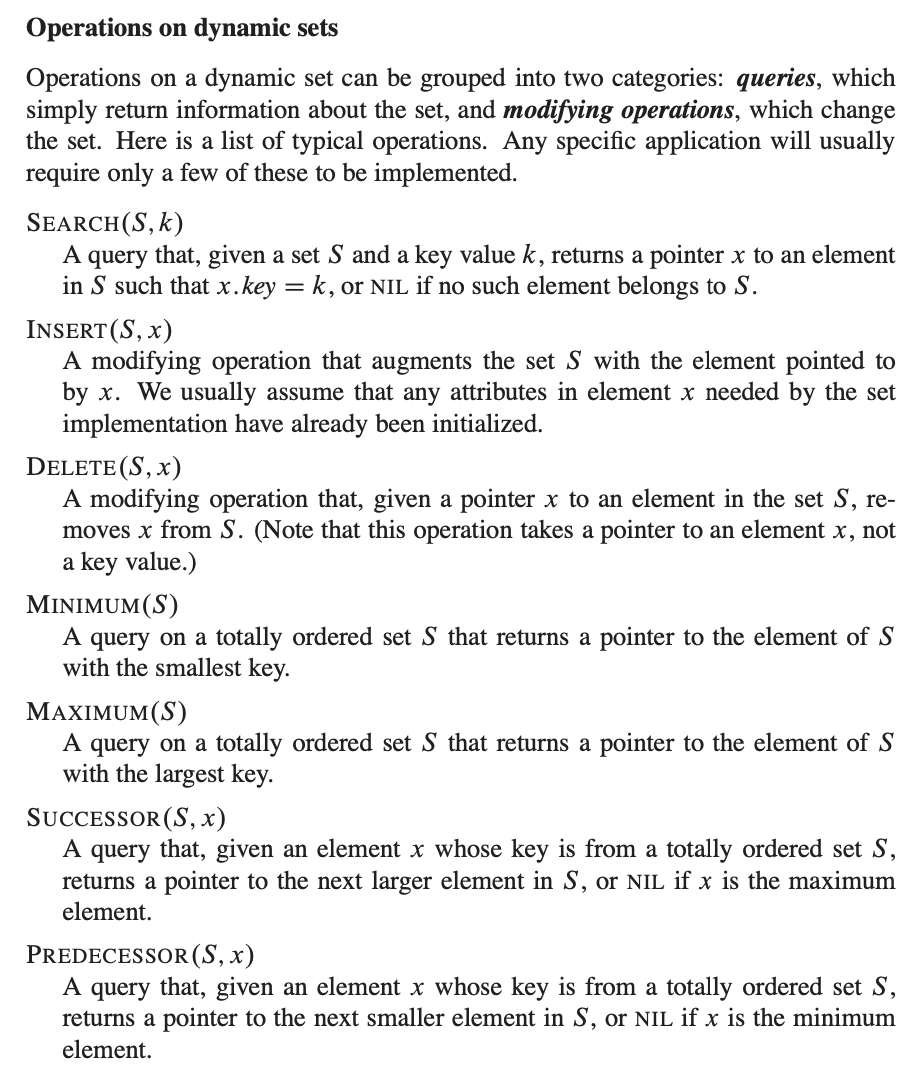
**CH 10. Elementary Data Structures**

* In a ***stack***, the element deleted from the set is the one most recently inserted: the stack implements a ***last-in, first-out***, or ***LIFO***, policy
* in a ***queue***, the element deleted is always the one that has been in the set for the longest time: the queue implements a ***first-in, first-out***, or ***FIFO***, policy
* in stacks, Insert is called PUSH, and delete is called POP b/c these are allusions to physical stacks



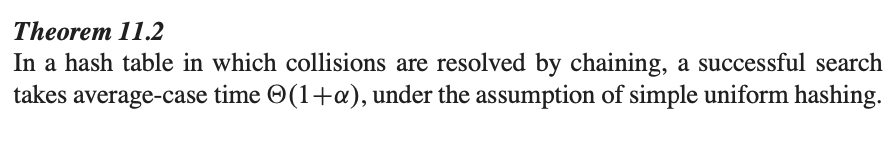
* When S.*top* = 0, the stack contains no elements and is ***empty***.
* In queues, insert is called ENQUEUE, and delete is called DEQUEUE.
* A linked list is a data structure in which the objects are arranged in a linear order.
* each element of a ***doubly linked list*** L is an object with an attribute *key* and two other pointer attributes: *next* and *prev*.
* If a list is ***singly linked***, we omit the *prev* pointer in each element
* If a list is ***sorted***, the linear order of the list corresponds to the linear order of keys stored in elements of the list; the minimum element is then the head of the list, and the maximum element is the tail
* In a ***circular list***, the *prev* pointer of the head of the list points to the tail, and the *next* pointer of the tail of the list points to the head.
* Searching a linked list + inserting to a list: 
* Deleting from a linked list: 

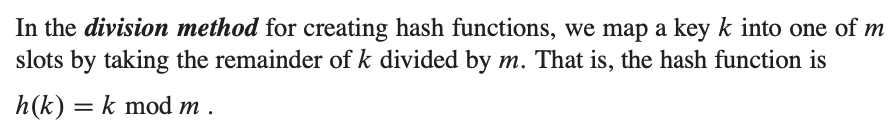


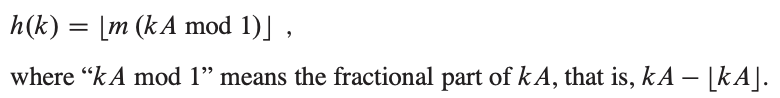


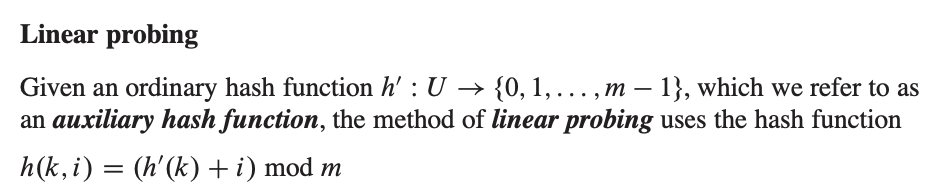
**CH 11. Hash Tables**

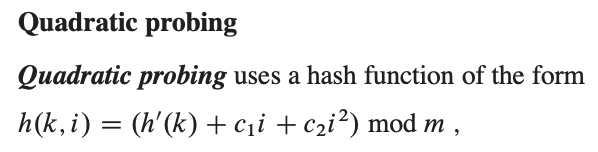
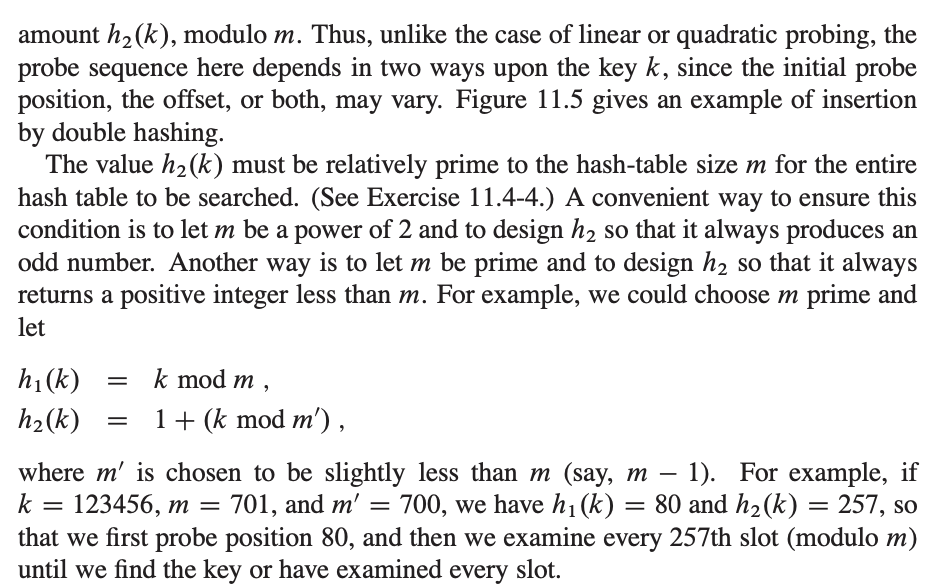
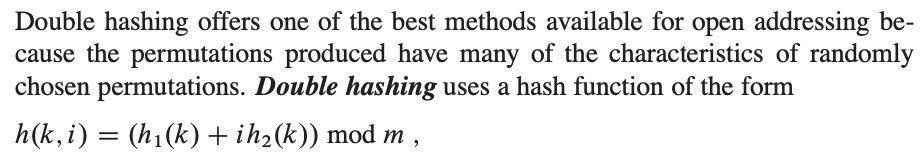
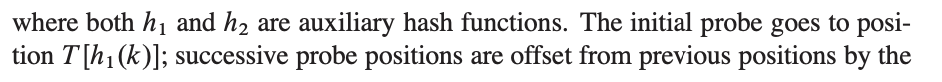
* “perfect hashing” can support searches in O(1) *worst- case* time, when the set of keys being stored is static (that is, when the set of keys never changes once stored).
* There is one hitch: two keys may hash to the same slot. We call this situation a ***collision***.
* ***Chaining:*** place all the elements that hash to the same slot into the same linked list
* Given a hash table T with m slots that stores n elements, we define the ***load factor*** ̨ for T as n=m, that is, the average number of elements stored in a chain.

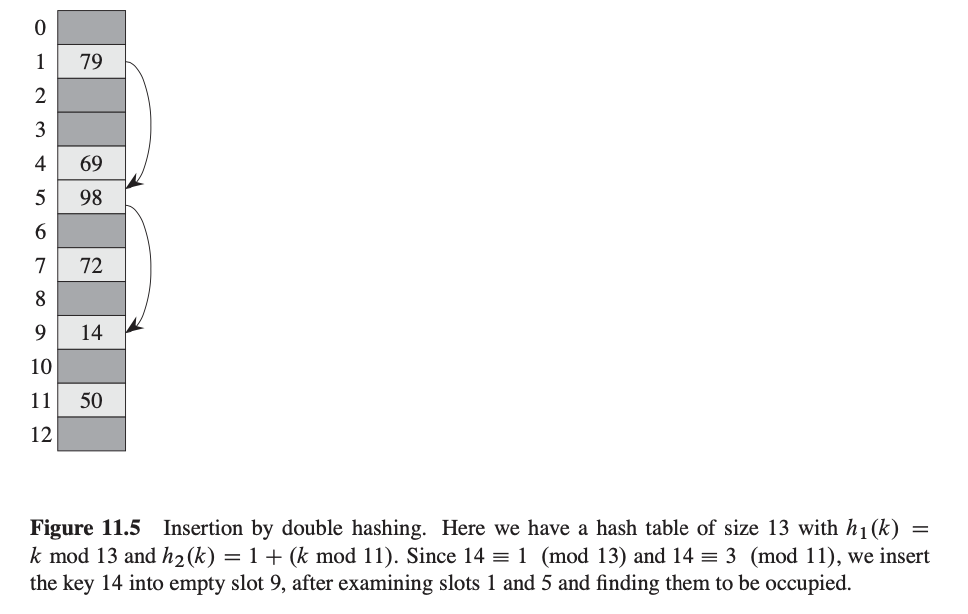
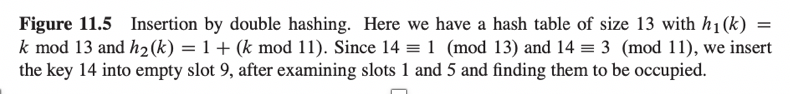




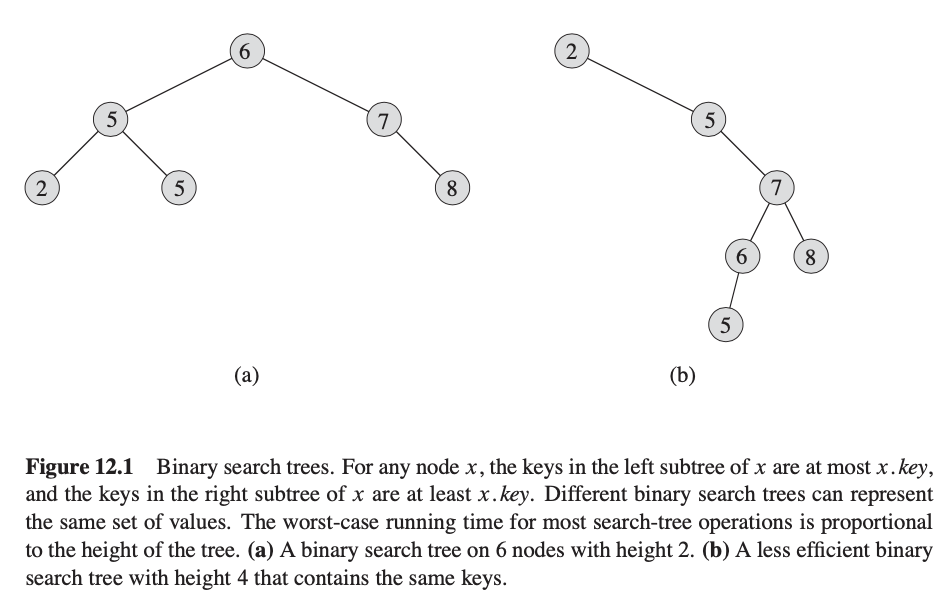
* In division method, avoid powers of 2
* Multiplication ^ method allows *m* to be a power of 2
* ***universal hashing***, can yield provably good performance on average, no matter which keys the adversary chooses.
* In open addressing, the load factor cannot exceed 1

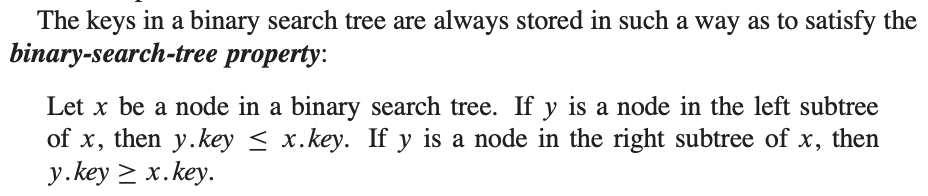
suffers from primary clustering

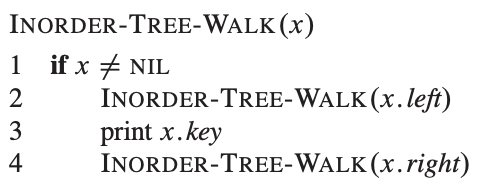
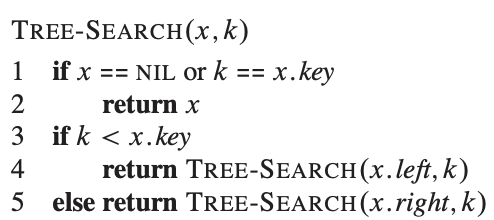
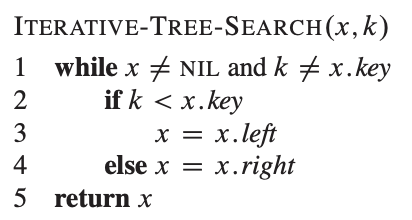


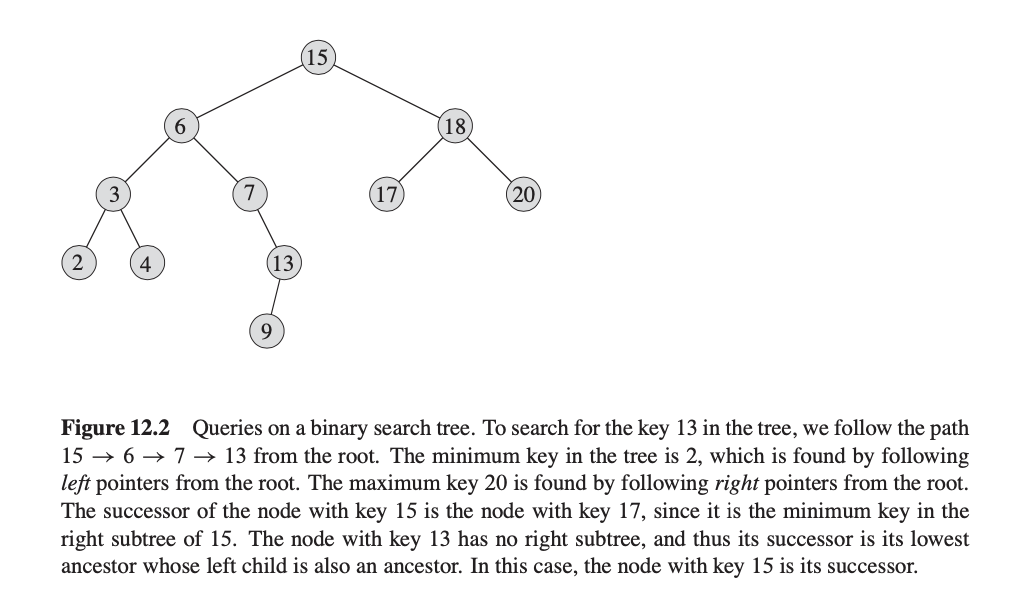


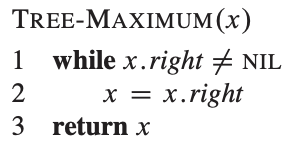
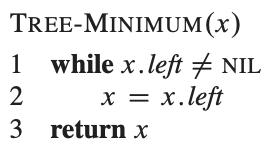
**CH 12. Binary Search Trees**

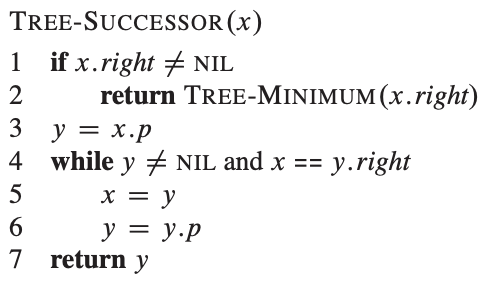
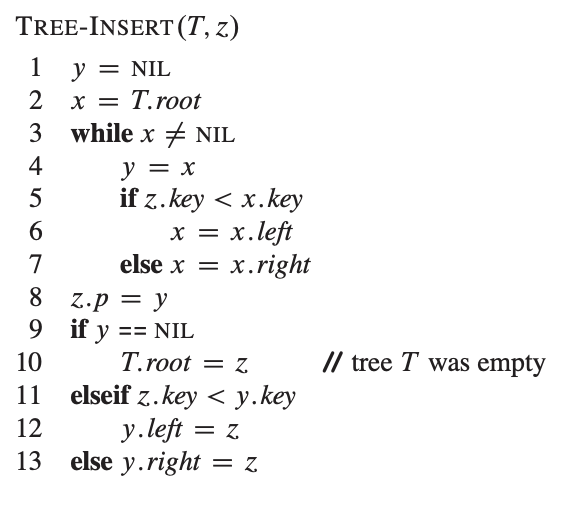
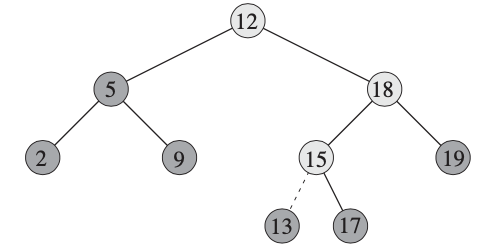


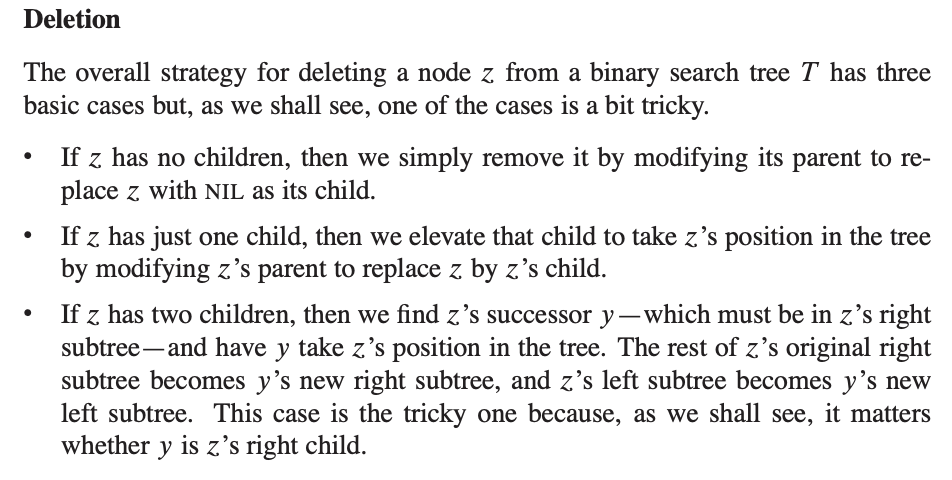
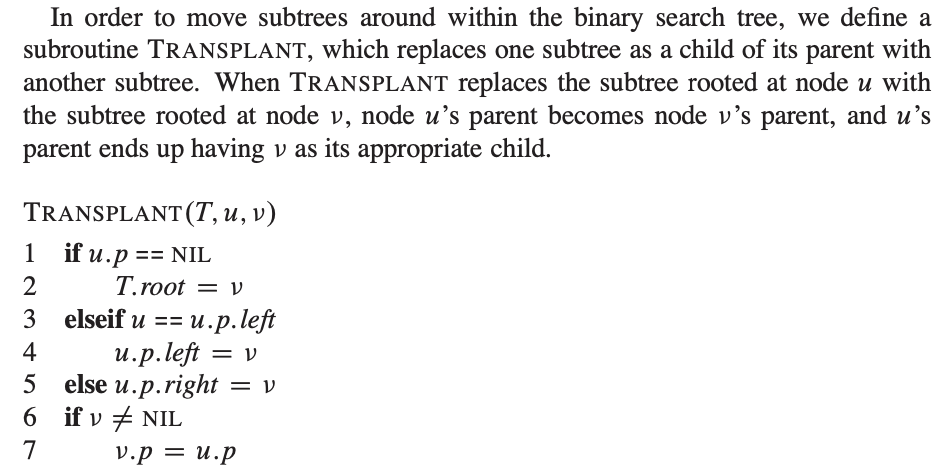
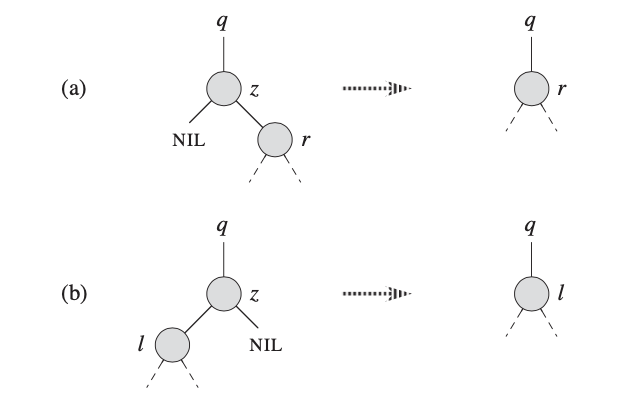
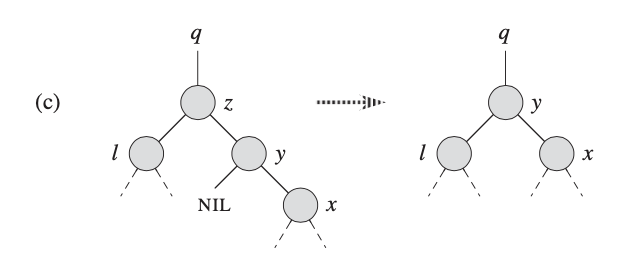
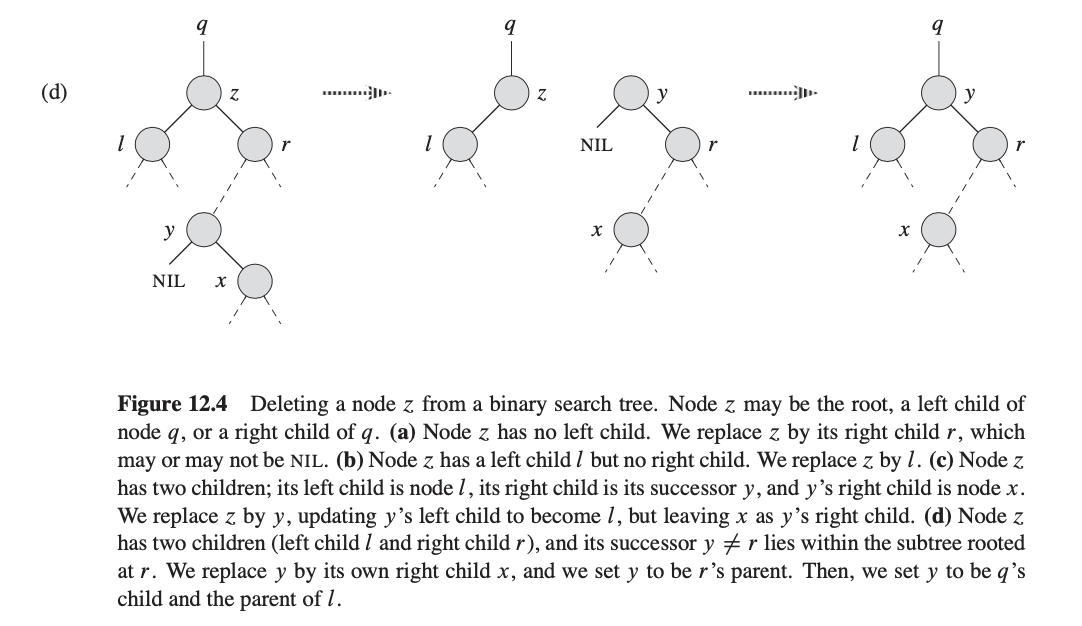


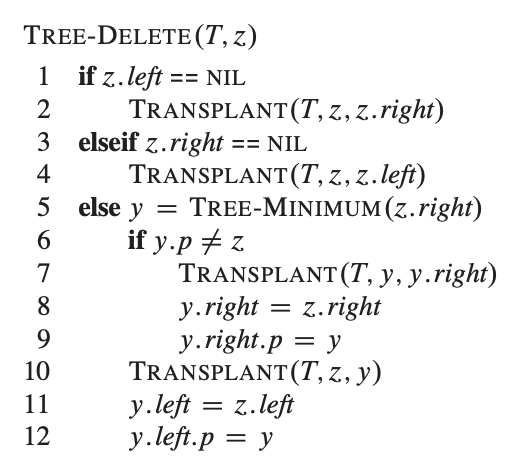
* The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an ***inorder tree walk O(n)***. This algorithm is so named because it prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree. (Similarly, a ***preorder tree walk*** prints the root before the values in either subtree, and a ***postorder tree walk*** prints the root after the values in its subtrees.)
* Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.



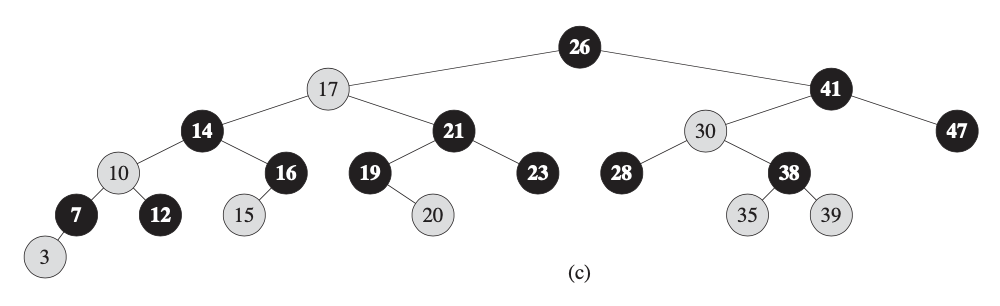
q2

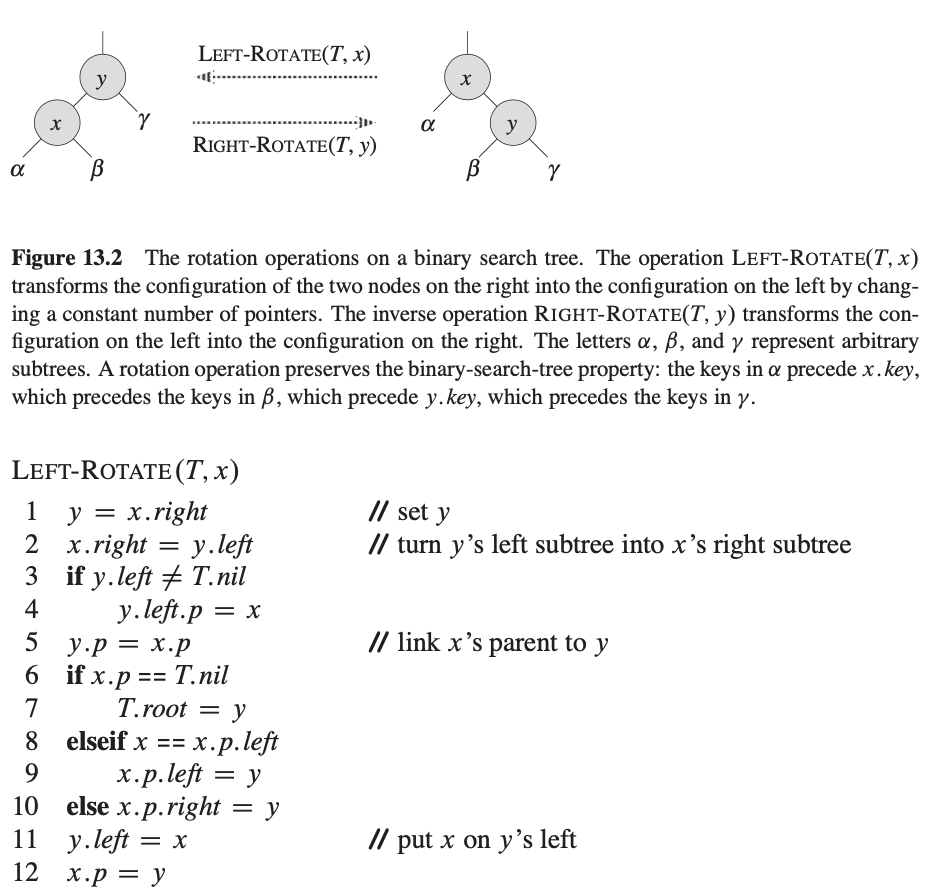
* We can always find an element in a binary search tree whose key is a minimum by following *left* child pointers from the root until we encounter a NIL **O(h)**
* If all keys are distinct, the successor of a node x is the node with the smallest key greater than x.*key*
* To insert a new value  *v* into a binary search tree T , we use the procedure TREE- INSERT **O(h)**. The procedure takes a node *z* for which *z.key* = *v*, *z.left* D NIL, and *z*.*right* = NIL. It modifies T and some of the attributes of ́ in such a way that it inserts ́ into an appropriate position in the tree.

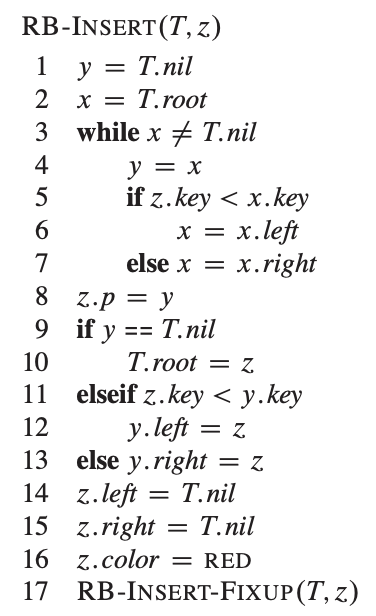


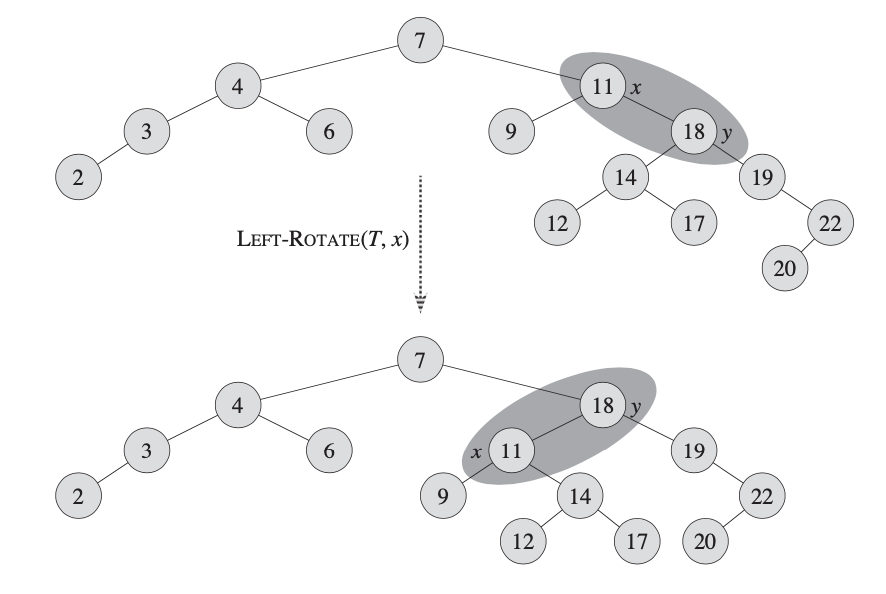


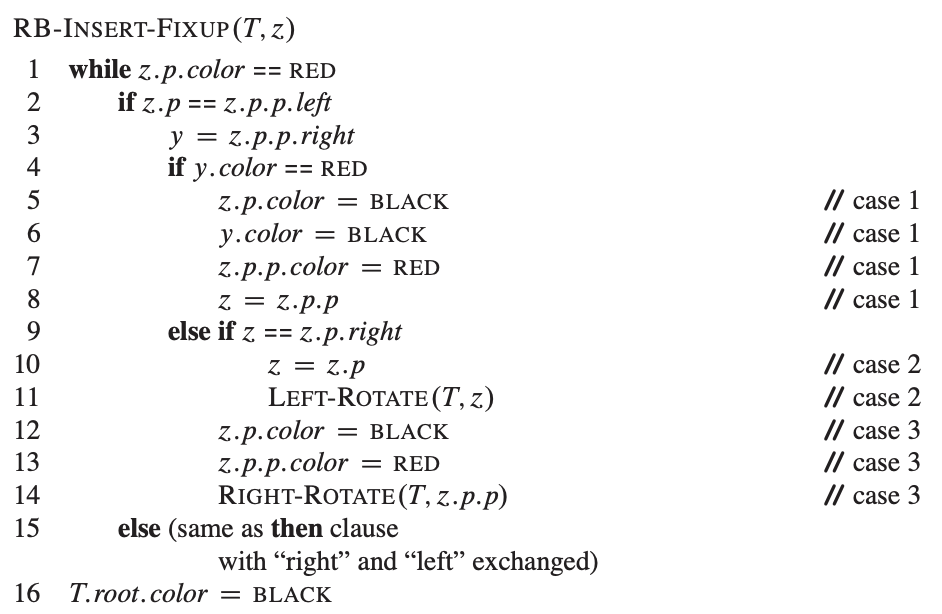
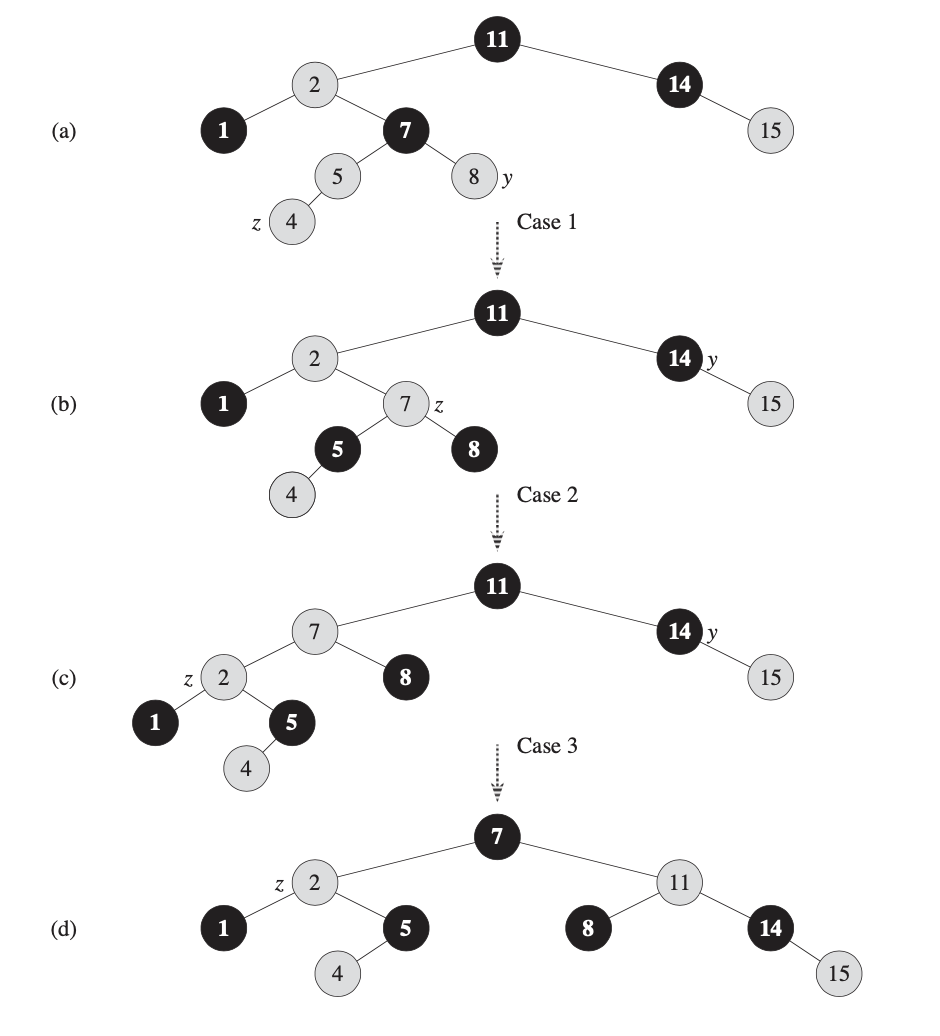
**CH 13. Red-Black Trees**

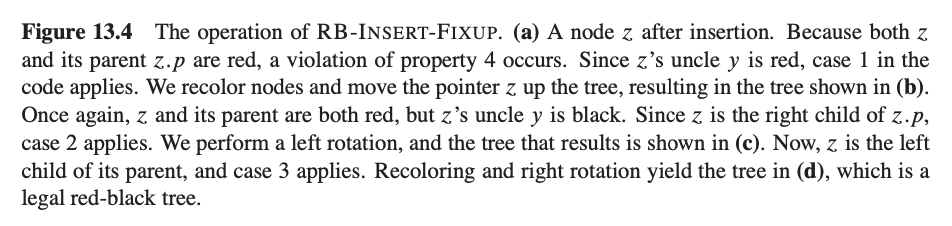
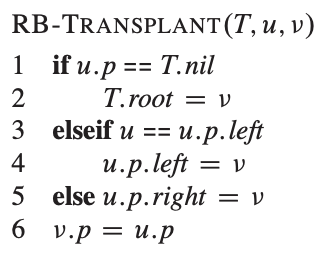
* A ***red-black tree*** is a binary search tree with one extra bit of storage per node: its ***color***, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately ***balanced***
* Each node of the tree now contains the attributes *color*, *key*, *left*, *right*, and *p*
* We shall regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.
* A red-black tree is a binary tree that satisfies the following ***red-black properties***:
  + Every node is either red or black.
  + The root is black.
  + Every leaf (NIL) is black.
  + If a node is red, then both its children are black.
  + For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.
* We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the ***black-height*** of the node, denoted bh(x)
* 

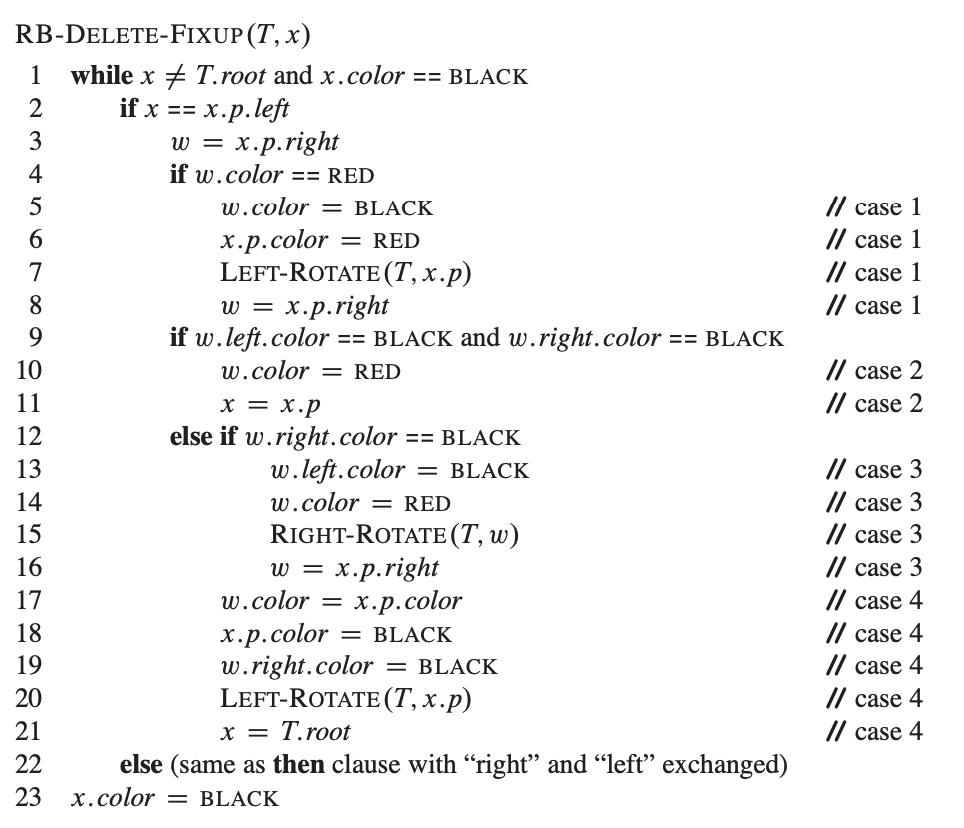


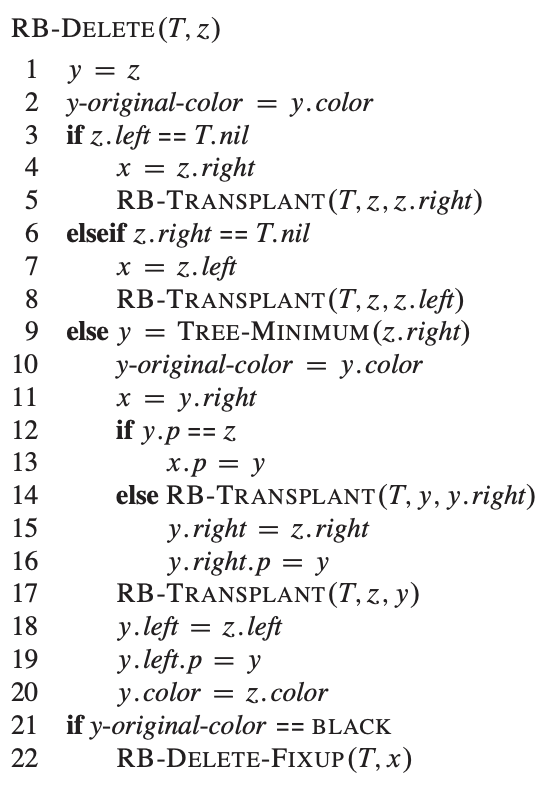


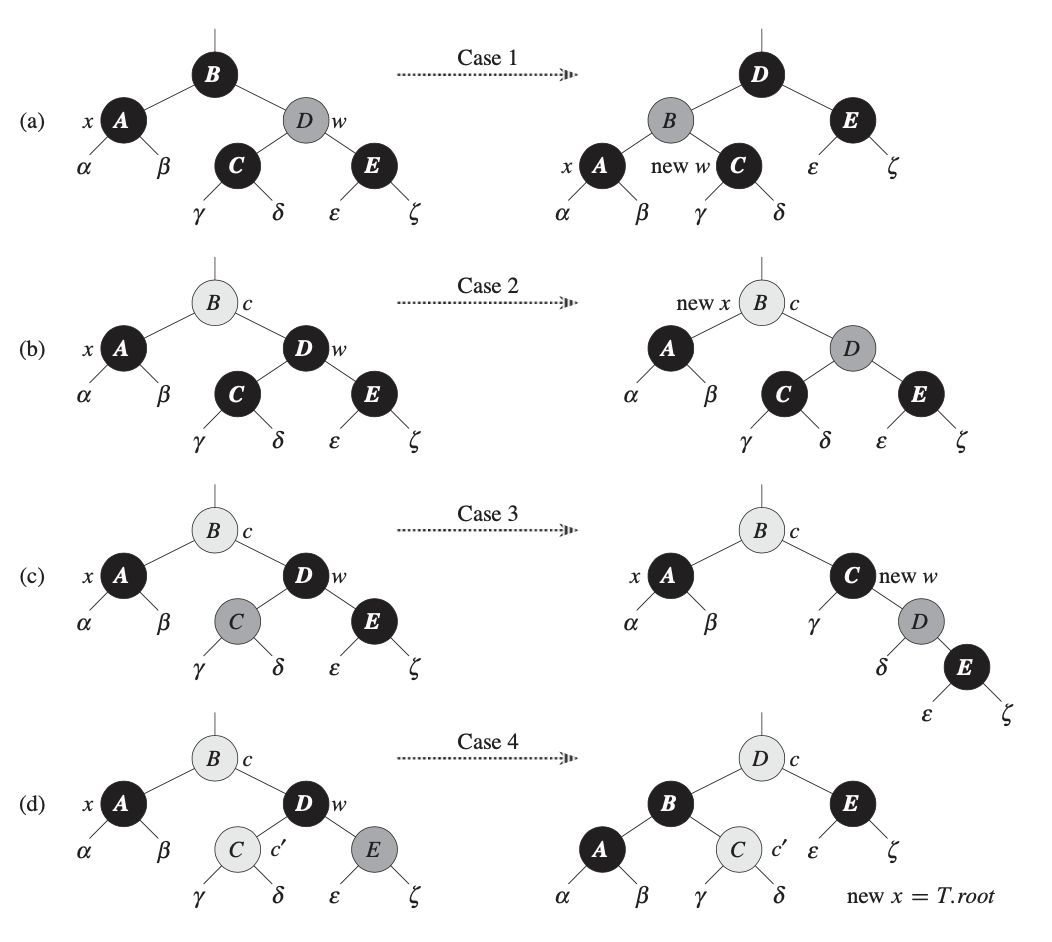


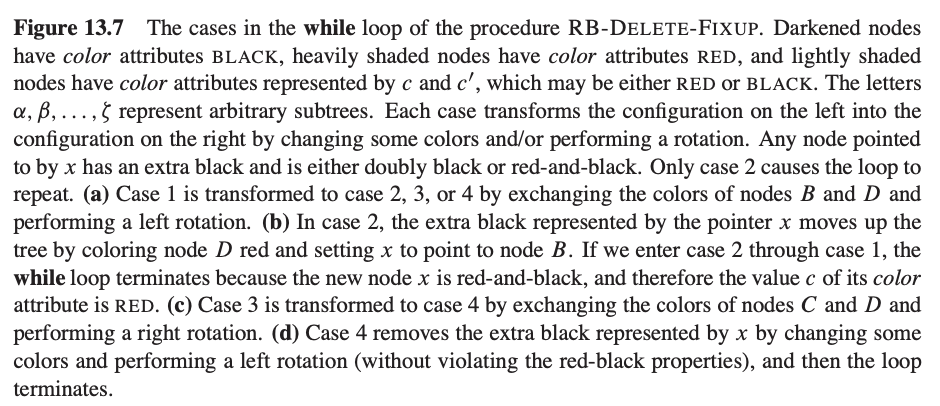




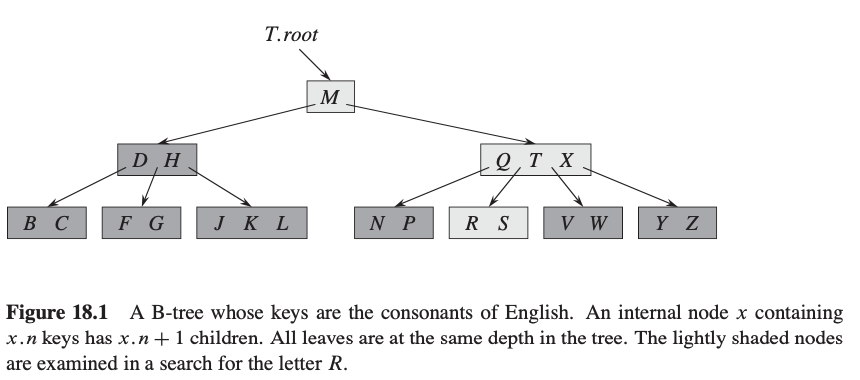




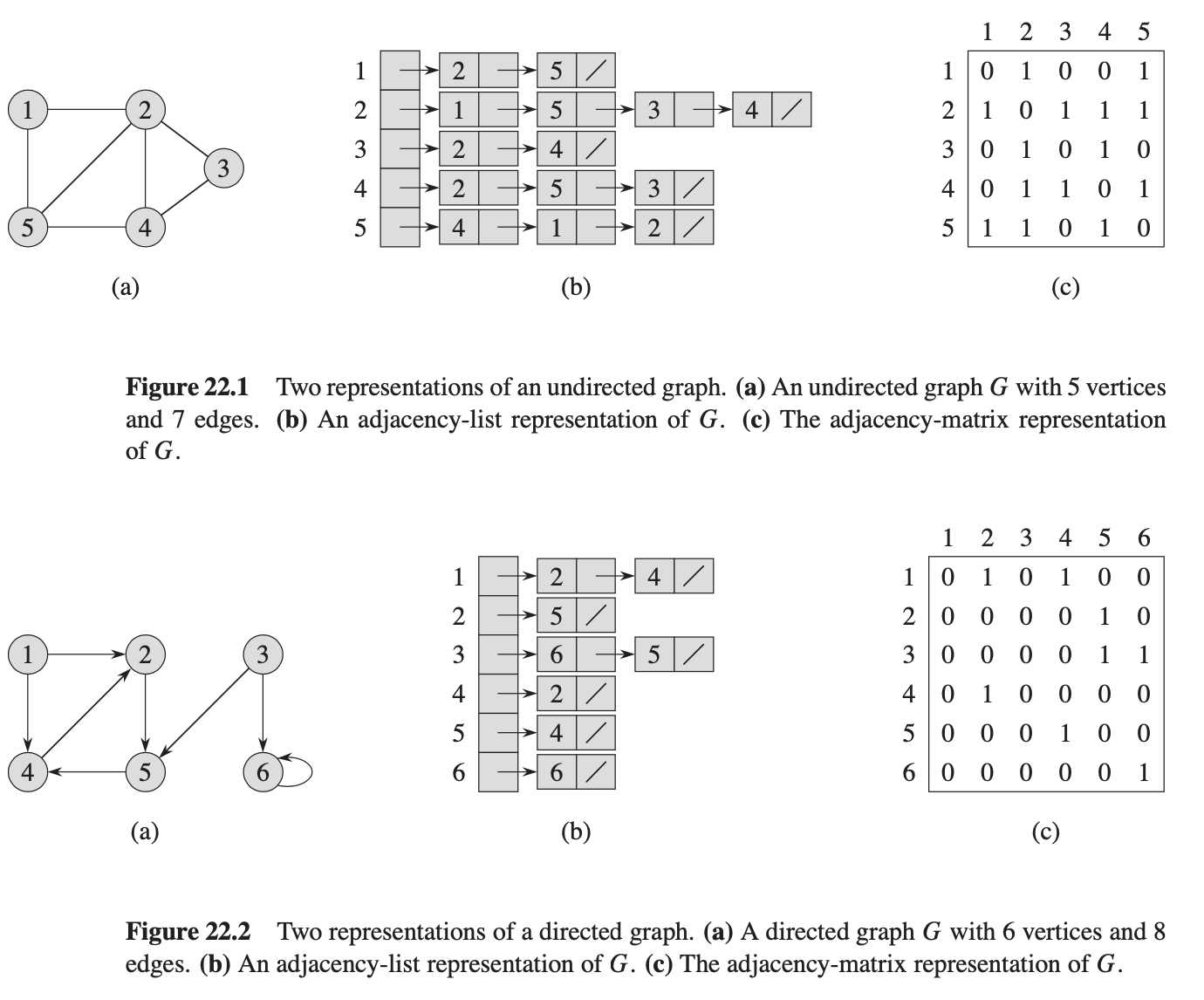
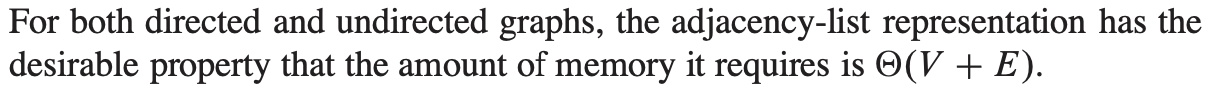
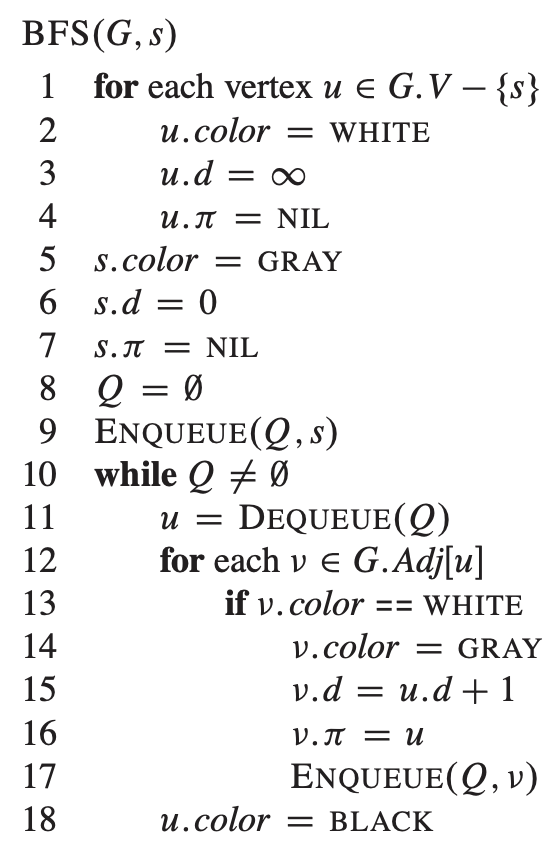
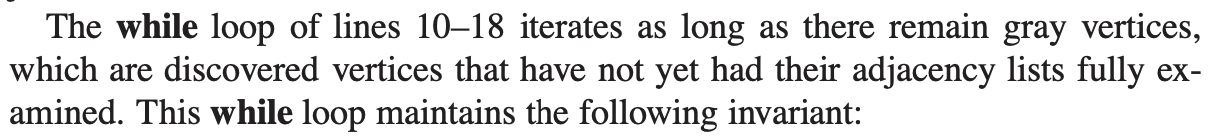


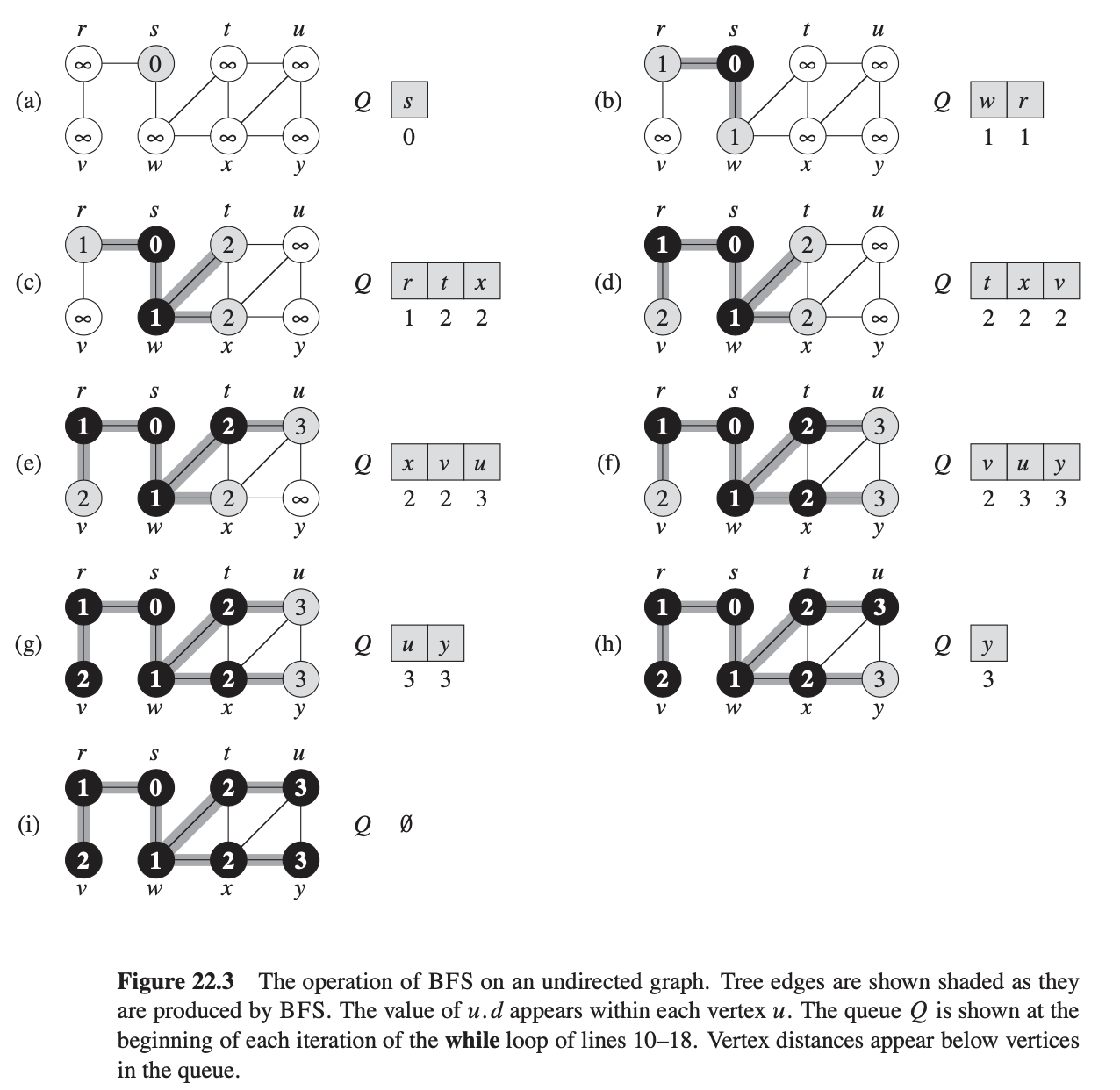


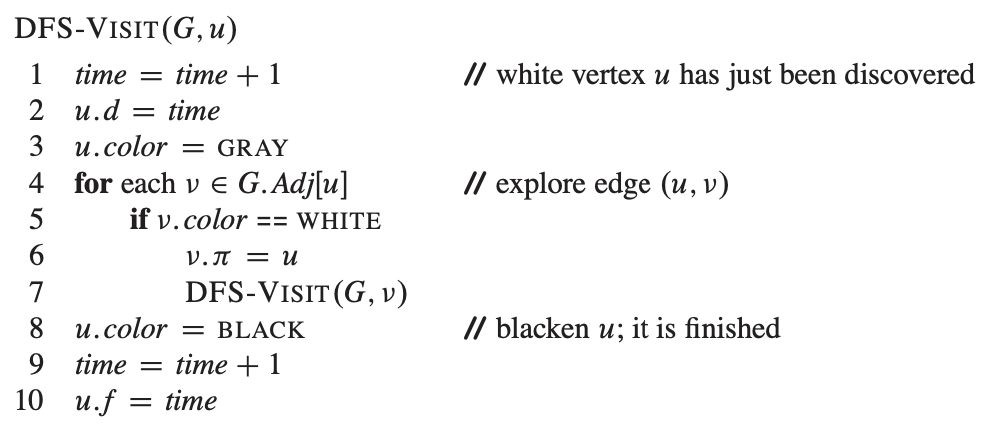
**CH 18. B-Trees**

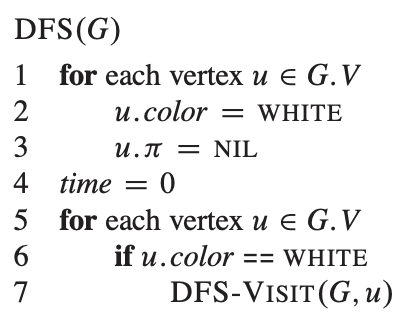


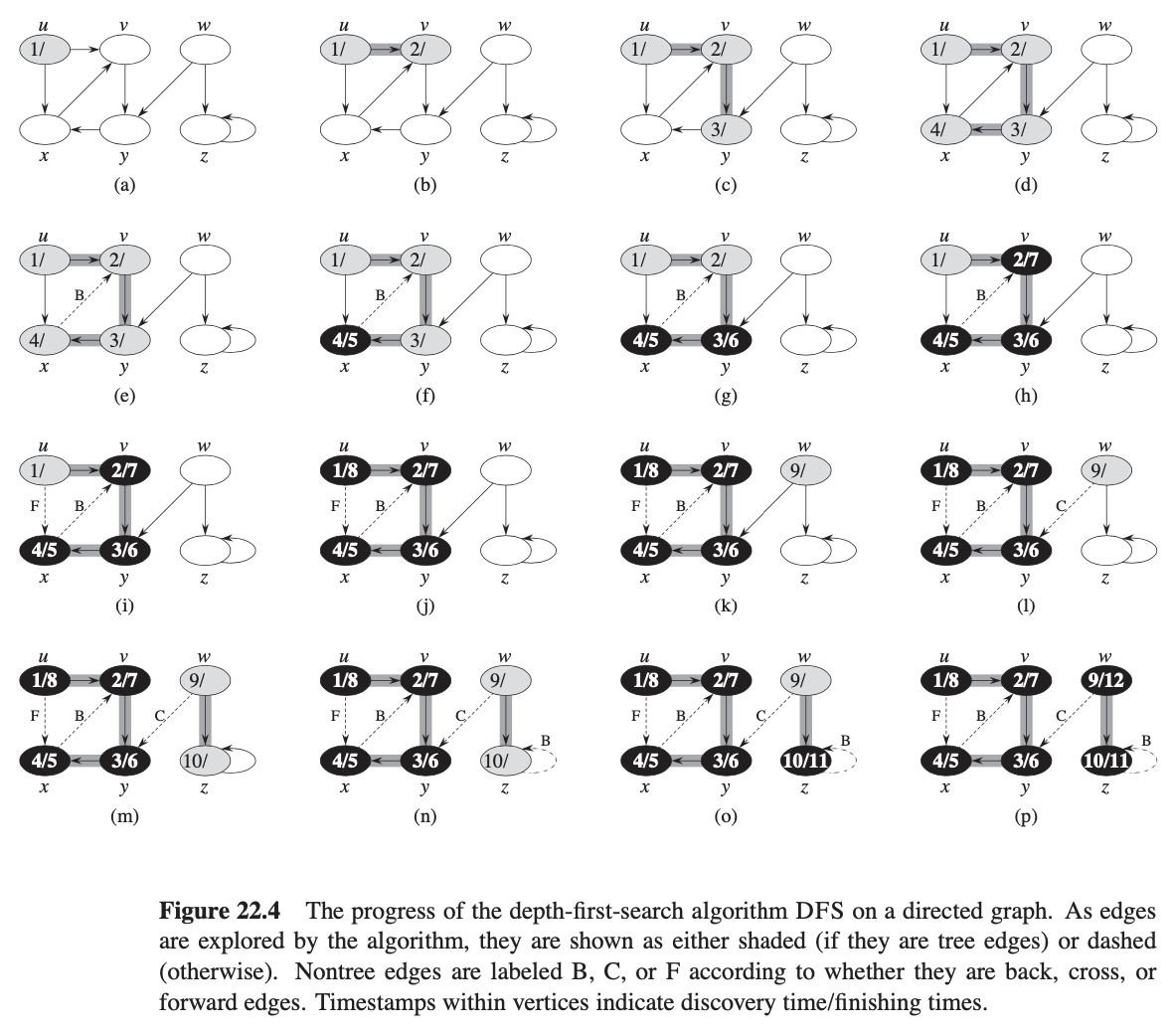
**CH 22. Elementary Graph Functions**

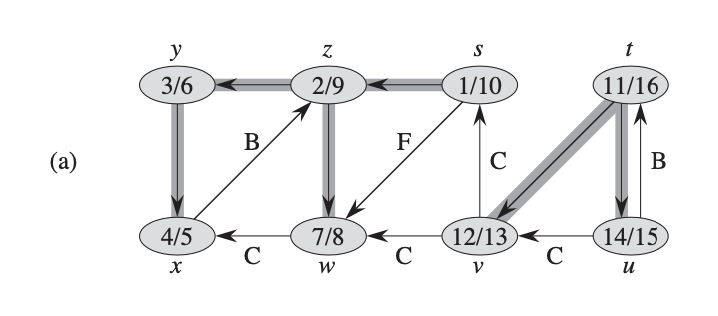
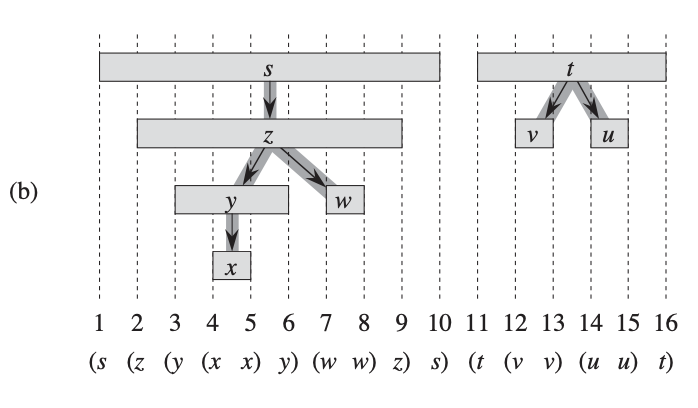
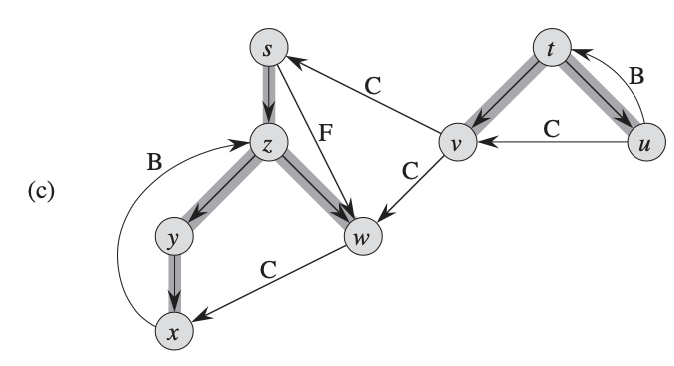
* G = (V,E) used to represent a graph. A sparse graph has E be much less then abs(V^2) and is represented by an adjacency list. Dense graphs represented by adjacency matrix
* Adjacency matrix can quickly tell is there is an edge connecting two given vertices but is very cost effective
* If G is a directed graph, the sum of the lengths of all the adjacency lists is abs(E), undirected has sum as 2abs(E)
* 
* Adjacency matrix require O(V^2) memory
* Given a G = (V,E), ***Breadth-first search*** selects a distinguished ***source*** vertex s, breadth-first search systematically explores the edges of G to “discover” every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a “breadth-first tree” with root s that contains all reachable vertices. For any vertex *v* reachable from s, the simple path in the breadth-first tree from s to *v* corresponds to a “shortest path” from s to *v* in G, that is, a path containing the smallest number of edges.
* The algorithm discovers all vertices at distance *k* from *s* before discovering any vertices at distance *k* + 1.
* To keep track of progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is ***discovered*** the first time it is encountered during the search, at which time it becomes nonwhite.
* *Pi* represents parent, *d* represents the property, *s* is the source vertex. *S* is painted grey to start because it is discovered as the procedure begins.
* At the test in line 10, the queue Q consists of the set of gray vertices. Line 11 determines the gray vertex *u* at the head of the queue Q and removes it from Q. The **for** loop of lines 12–17 considers each vertex *v* in the adjacency list of *u*. If *v* is white, then it has not yet been discovered, and the procedure discovers it by executing lines 14–17. The procedure paints vertex *v* gray, sets its distance *v*.*d* to u.*d*+1, records *u* as its parent *v*.*pi*, and places it at the tail of the queue Q . **RUN TIME : *O*(*V* + *E*)**

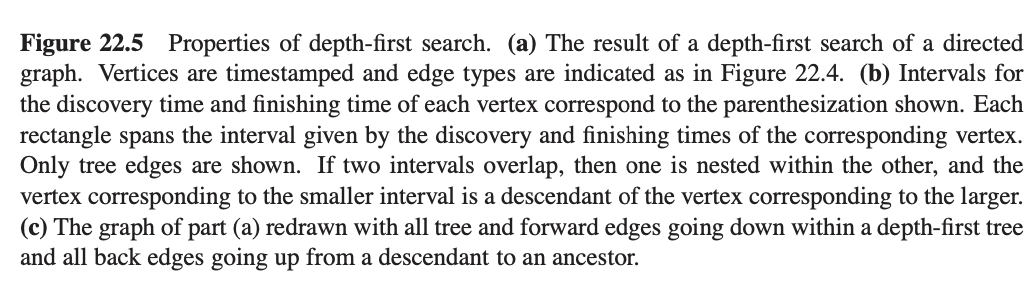
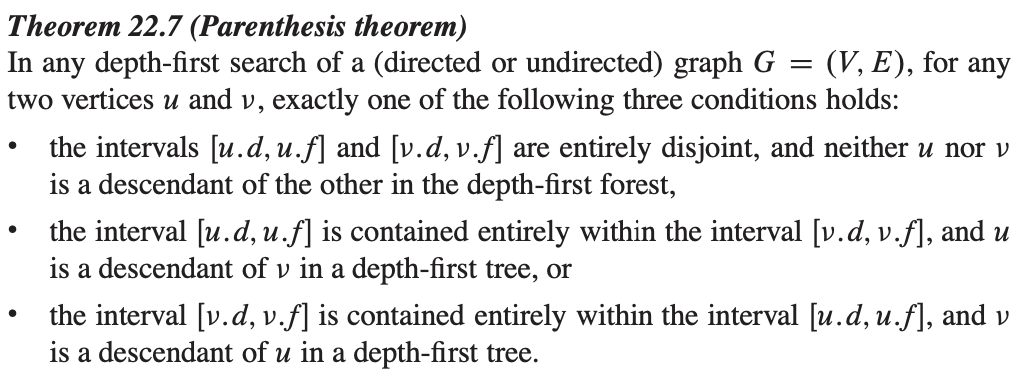


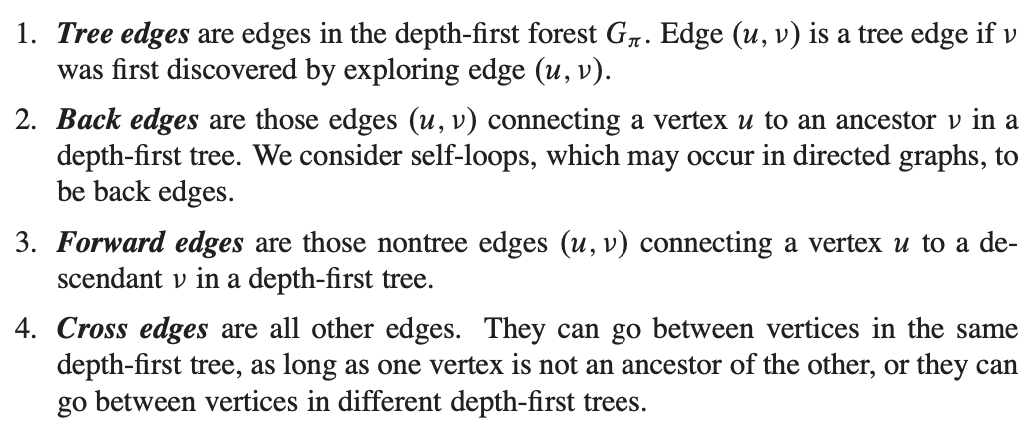
**\*\* 24.3 -** Dijkstra’s single-source shortest-paths algorithm \*\* \*\*for H.W. 7 Q3\*\*

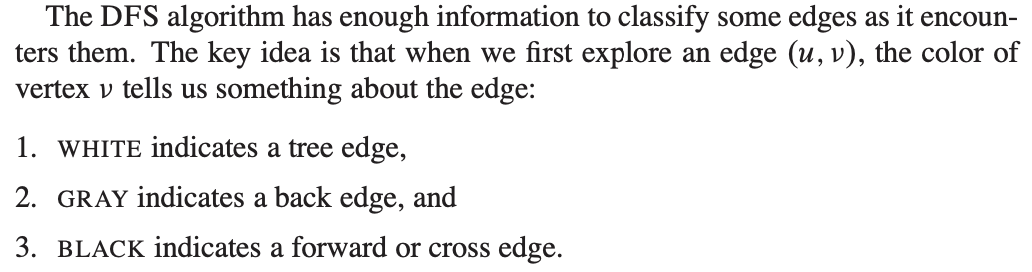


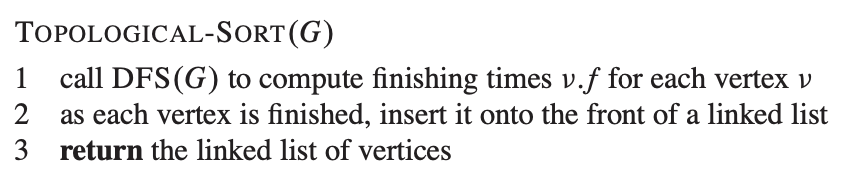
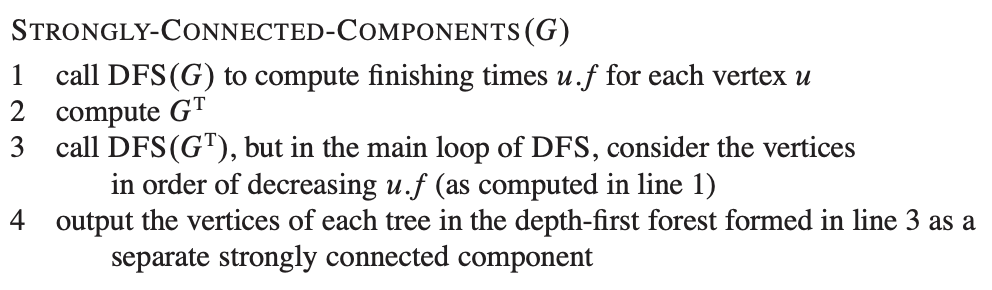
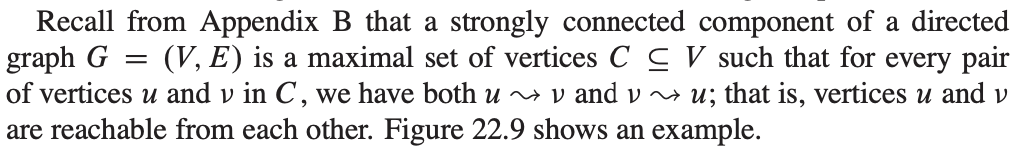
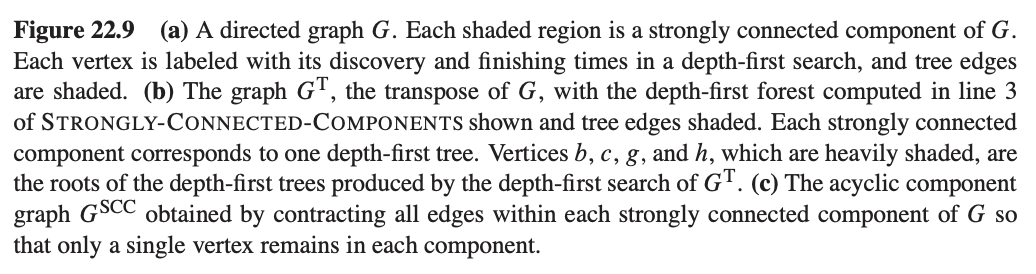


* **RUN TIME: *O(V+E)***

****

****



* A ***topological sort*** of a dag (acyclic graph) G D .V; E/ is a linear ordering of all its vertices such that if G contains an edge (u, *v*), then *u* appears before *v* in the ordering. (If the graph contains a cycle, then no linear ordering is possible.) **RUNTIME: O(V + E)**

