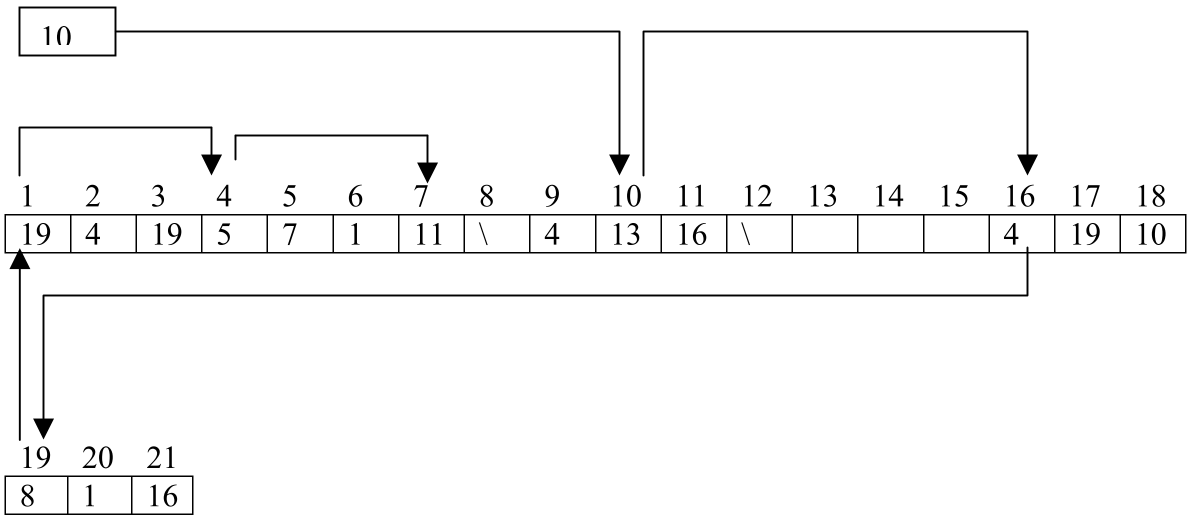
1. 10.3-1

1

Start

	1	**2**	**3**	**4**	**5**	**6**
NEXT	2	3	4	5	6	\
KEY	13	4	8	19	5	11
PREV	\	1	2	3	4	5



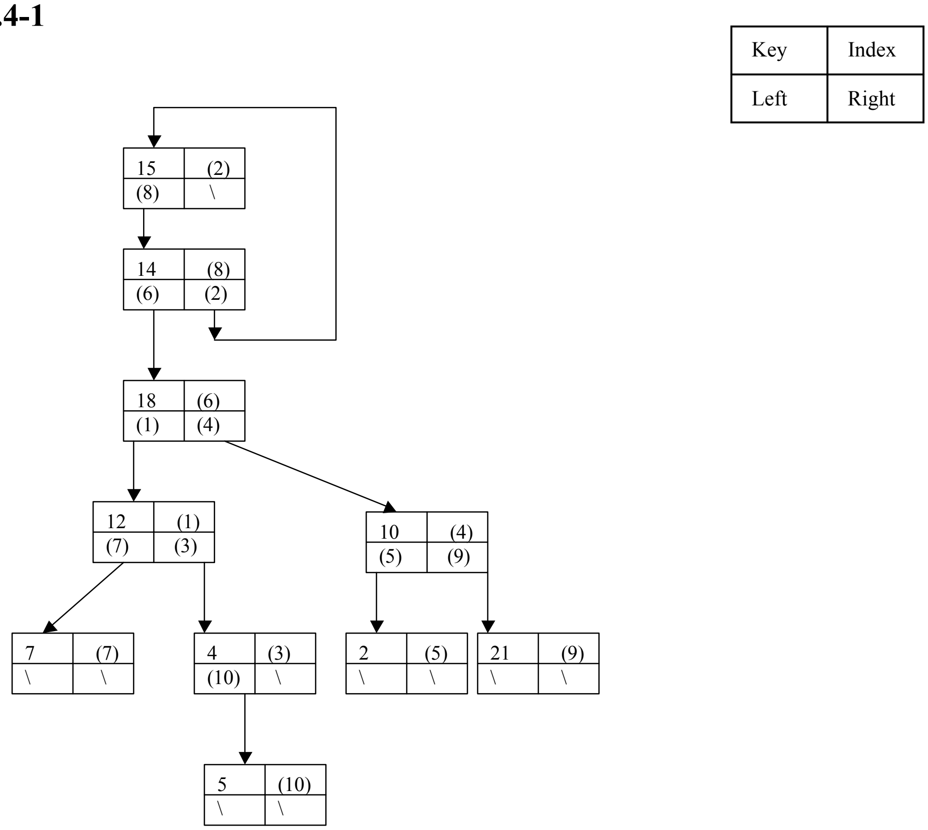
2. 10.3-2

Each list element is an object that occupies a contiguous sub array of length 2 within the array. The two fields are *key*, *next* corresponds to offsets 0 and 1 respectively. A pointer to an object is an index of the first element of the object. We keep the free objects in the same array, which we call the *free list*. The free list uses the *next*, which store the next pointers within the list. The head of the free list is held in the global variable *free*.

```
Allocate-Object()
{
         if(free = NIL)
                 then error " Out of Space"
        else
                 x=free
                 free = next[A[free ]]

        return x
}
```

```
Delete-Object(x)
{
        next[A[x]] = free
        free=x
}
```

3. 10.4-1



4. 10.4-4

```
print-tree(root)
{
      if (root ≠ NIL)
      {
            print(root.key)

            print-tree(root.left-child)
            print-tree(root.right-sibiling)
      }
}
```

5. 10-1

Comparison among lists

	Unsorted single linked list	Sorted Single linked list	Unsorted Doubly linked list	Sorted Doubly linked list
SEARCH (L, K)	Θ(n)	Θ(n)	Θ(n)	Θ(n)
INSERT (L, X)	Θ(1)	Θ(n)	Θ(1)	Θ(n)
DELETE (L, X)	Θ(n)	Θ(n)	Θ(1)	Θ(1)
SUCCESSOR (L, X)	Θ(n)	Θ(1)	Θ(n)	Θ(1)
PREDECESSOR (L, X)	Θ(n)	Θ(n)	Θ(n)	Θ(1)
MINIMUM (L)	Θ(n)	Θ(1)	Θ(n)	Θ(1)
MAXIMUM (L)	Θ(n)	Θ(n)	Θ(n)	Θ(n)

6. 12.2-1

c and e

c Because 912 cannot be encountered when a left path is taken from 911
e Because 299 cannot be encountered after taking a right path from 347

7. 12.3-4

Tree-Delete handles the deletion of a node z with two children by redirecting the pointers from p[z], left[z] and right [z]to point to z's successor. This replaces the copying of the data from the successor.

```
Tree Delete(T, z)
{
      if left[z] = NIL or right[z] =NIL
            then y ← z
            else y ← TREE-SUCCESSOR(z)

      if left[y] ≠ NIL
            then x ← left[y]
            else x ← right[y]

      if x ≠ NIL
            then p[x] ← p[y]

      if p[y] = NIL
            then root[T] ← x
            else if y = left[p[y]]
                  then  left[p[y]] ← x
                  else right[p[y] ← x
```

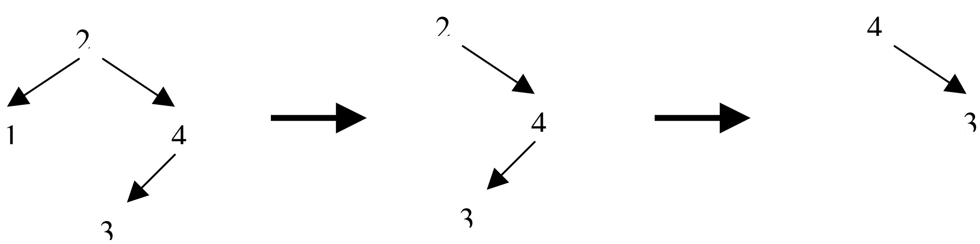
```
    if y ≠ z
          left[y] ← left[z]
          right[y] ← right[z]
          p[left[z]] ← y
          p[right[z] ← y

    if z = left[p[z]]
          left[p[z]] ← y

    else
          right[p[z]] ← y

}
```

8. 12.3-5

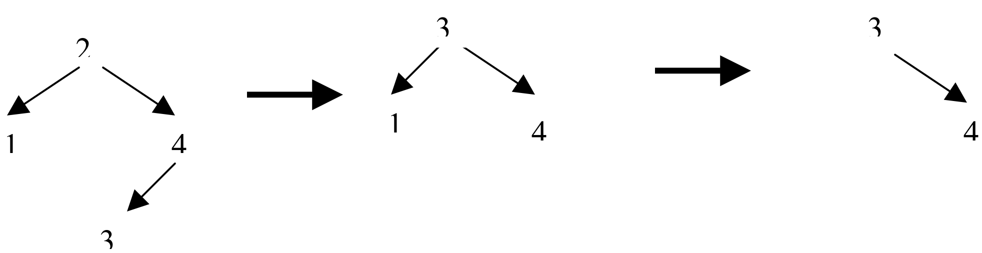
False

Below is a counter example

Deleting 1 then 2

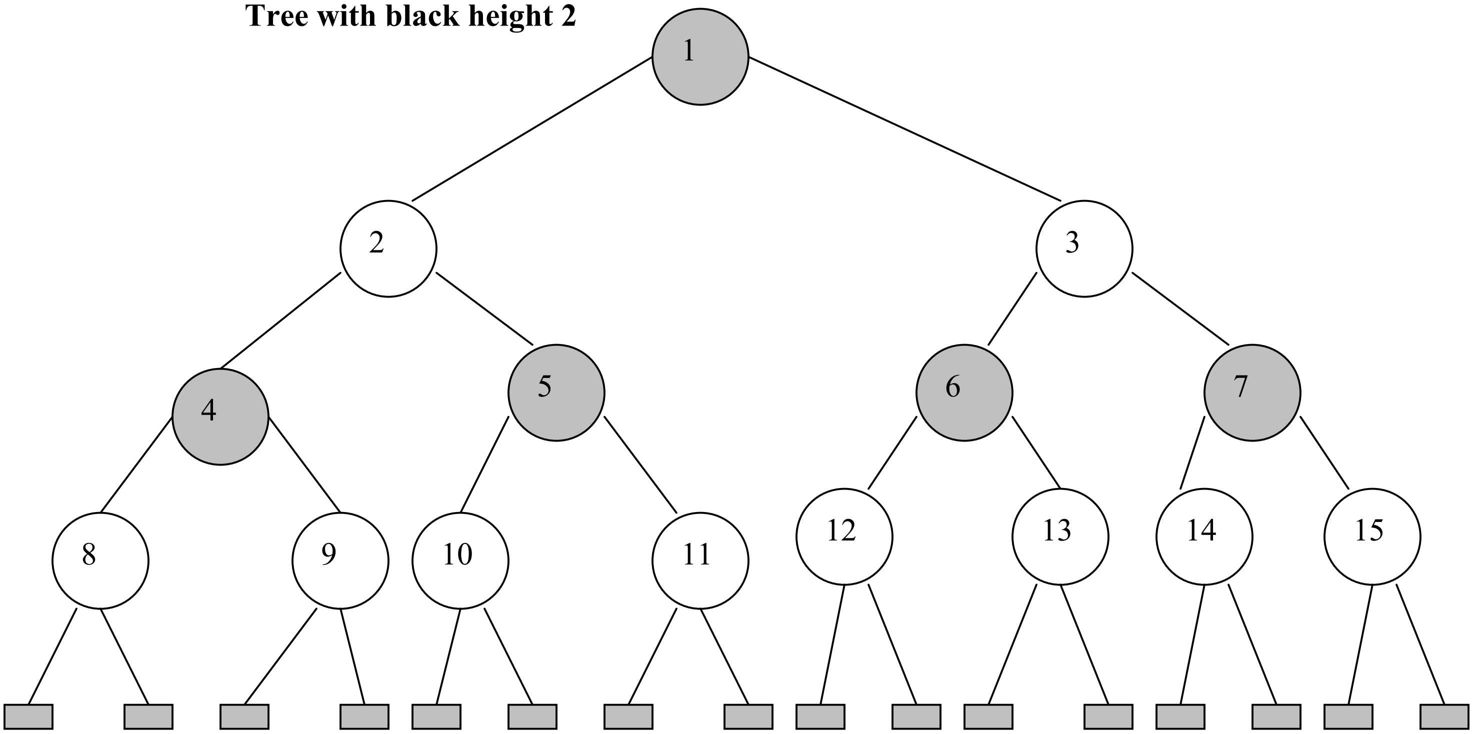


Deleting 2 then 1

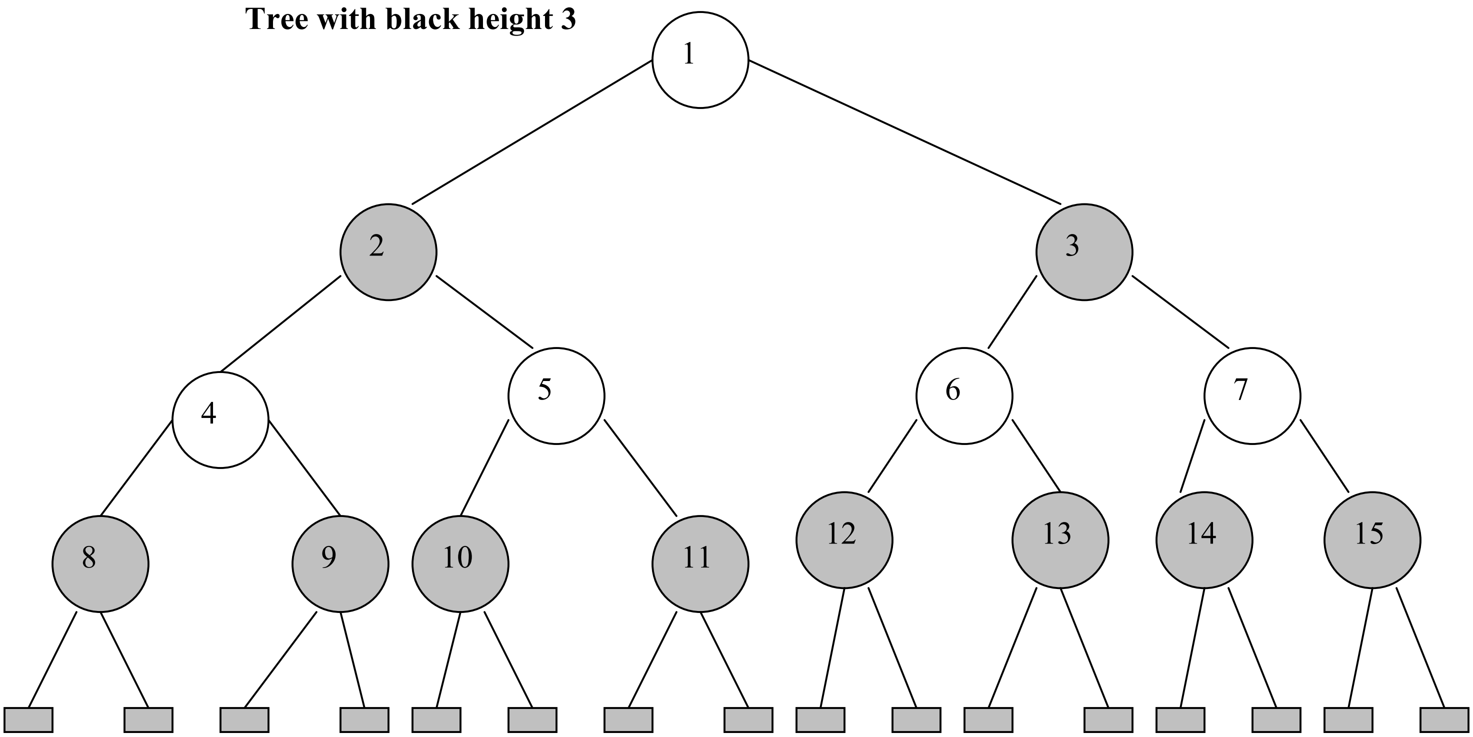


9. 13.1-1

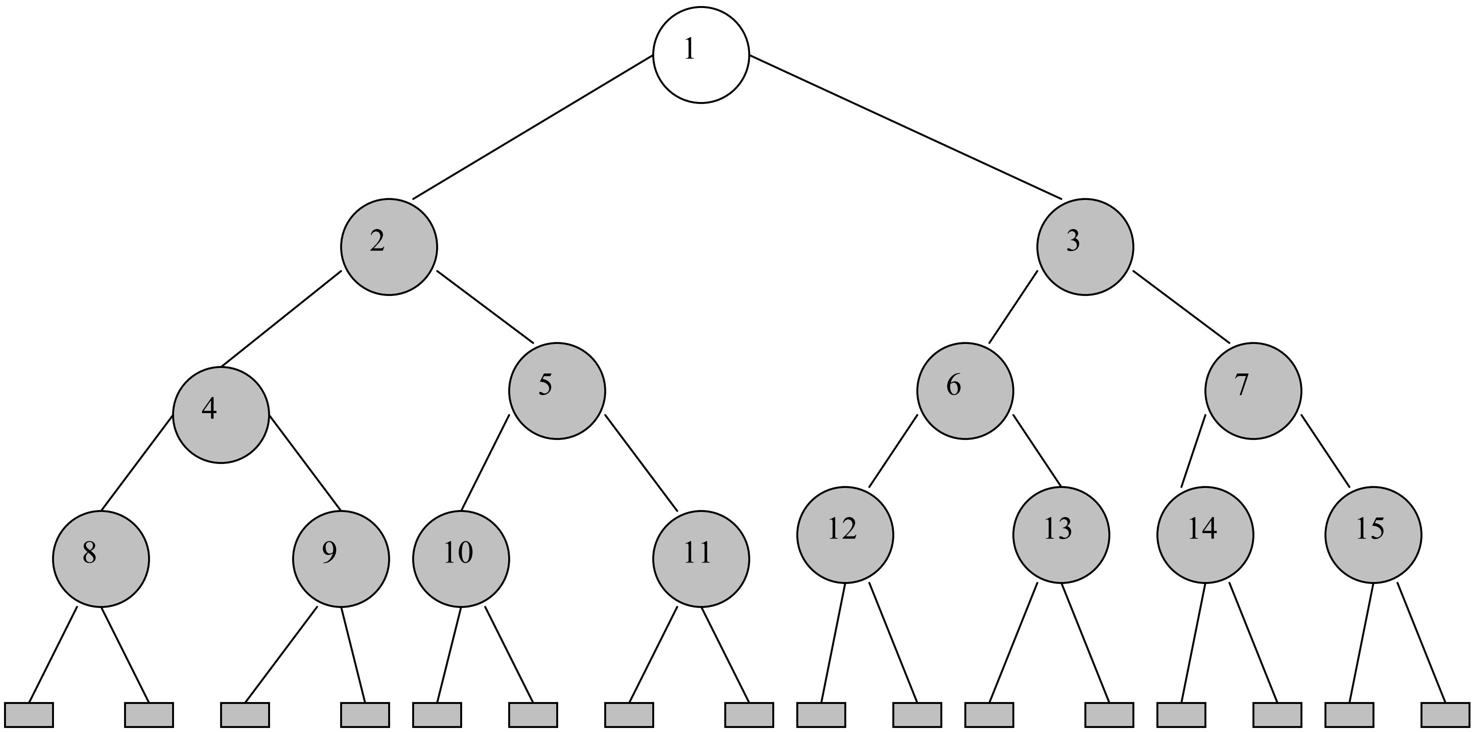
Tree with black height 2



Tree with black height 3



Tree with black height 4



10. 13.1-5

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest path from node x to a descendant leaf.

The shortest simple path from any node x will be the black height of the tree with x as root(i.e., bh(x)). There could be many branches in the tree; each branch is a combination of red and black nodes. The longest simple path in any tree will be that path which has the total number of nodes = (Property 4) bh(x) + max possible number of red nodes. The maximum possible number of red nodes will be equal to the bh(x), as to satisfy the red-black property., for each red node, its children has to be clack (no two consecutive red nodes in a path). Hence the max height of the tree could be 2 * bh(x), twice the shortest simple path.

11. 13.2-3

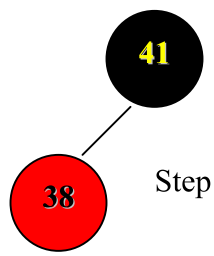
Let a, b, and c be arbitrary nodes in subtrees α, β, and γ, respectively, in the left tree of Figure 13.2. How do the depths of a, b, and c change when a left rotation is performed on node x in the figure?

The depth of a increases by +1
The depth of b remains the same
The depth of c changes by –1

12. 13.3-2

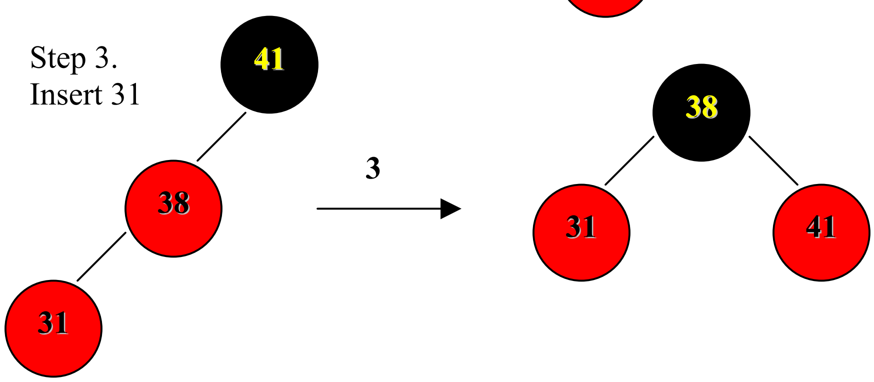
Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, and 8 into an initially empty red-black tree.

Step 1. Insert 41

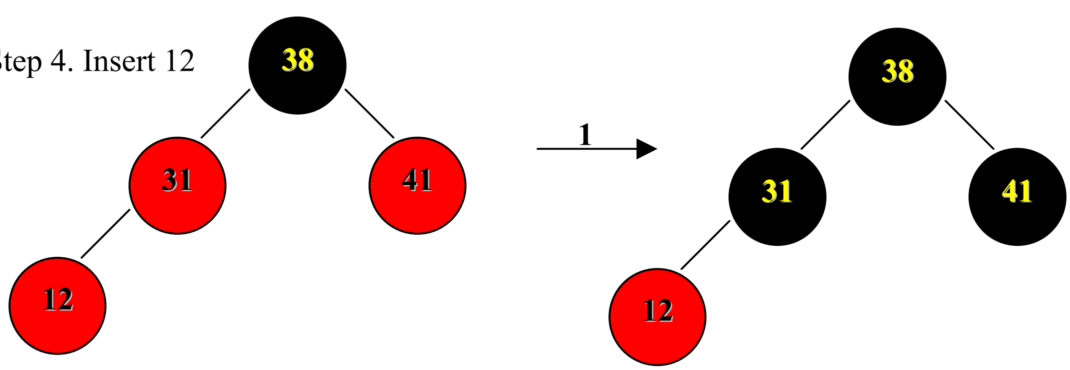


Step 2. Insert 38

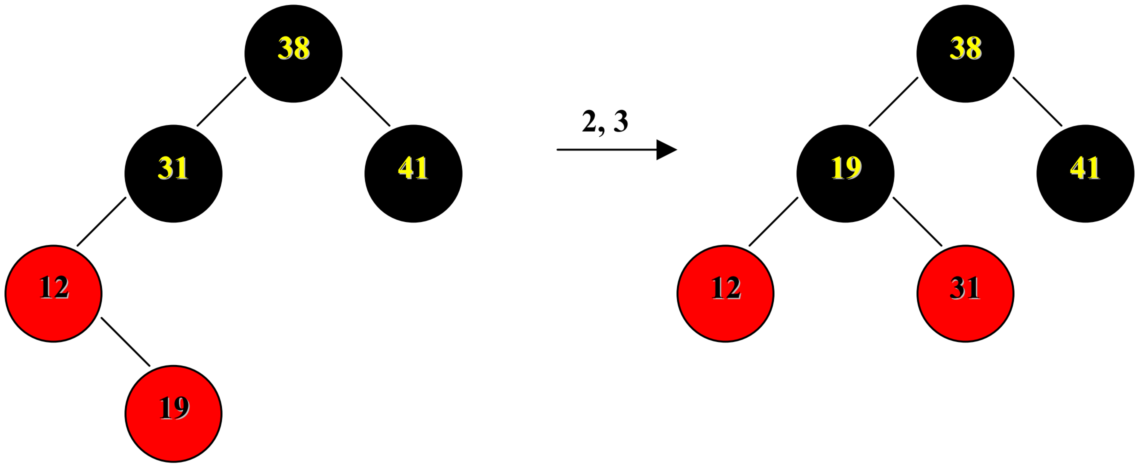
Step 3. Insert 31



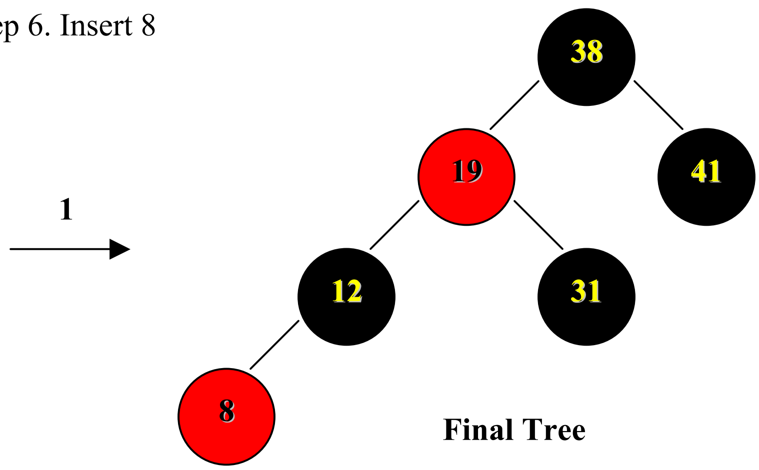
Step 4. Insert 12



Step 5. Insert 19



Step 6. Insert 8

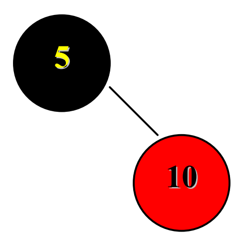


Final Tree

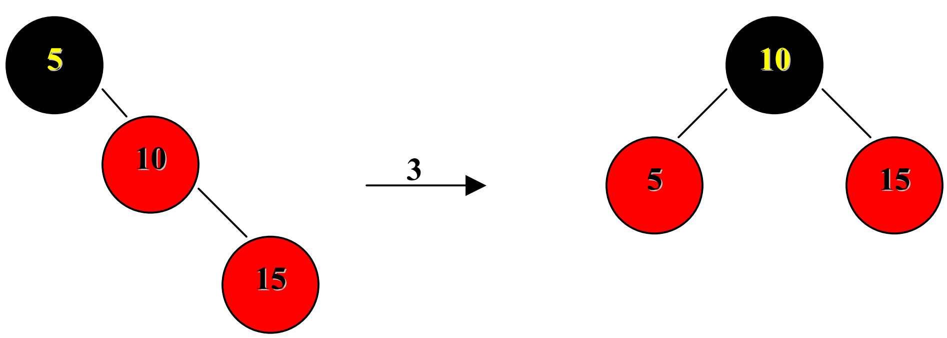
13. Show the red-black trees that result after successively inserting the keys 5, 10, 15, 25, 20, and 30 into an initially empty red-black tree.

Step 1. Insert 5

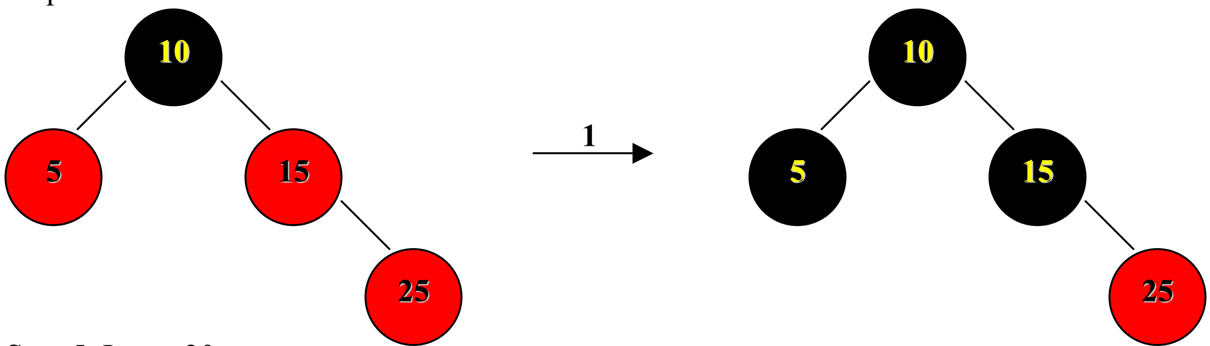
Step 2. Insert 10



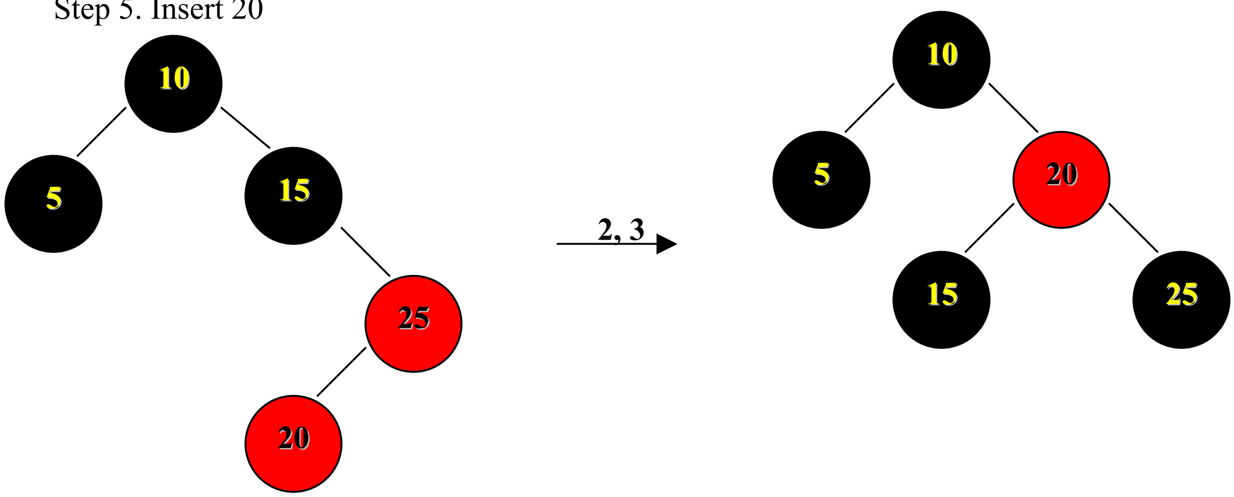
Step 3. Insert 15



Step 4. Insert 25

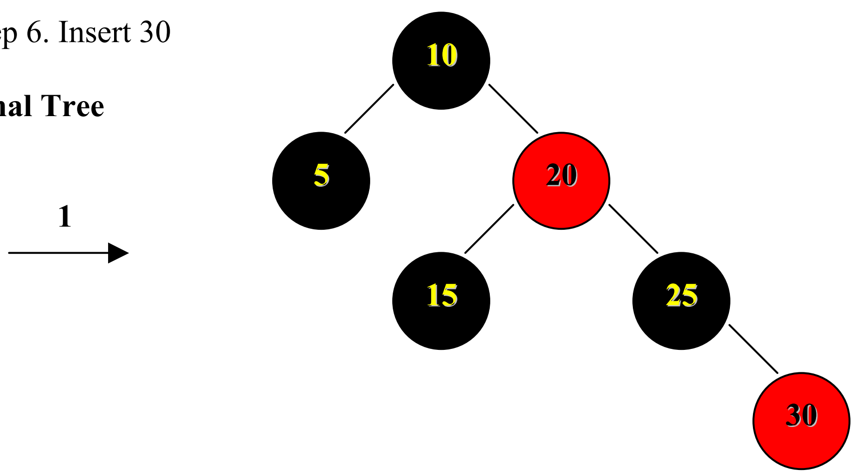


Step 5. Insert 20



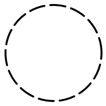
Step 6. Insert 30

Final Tree



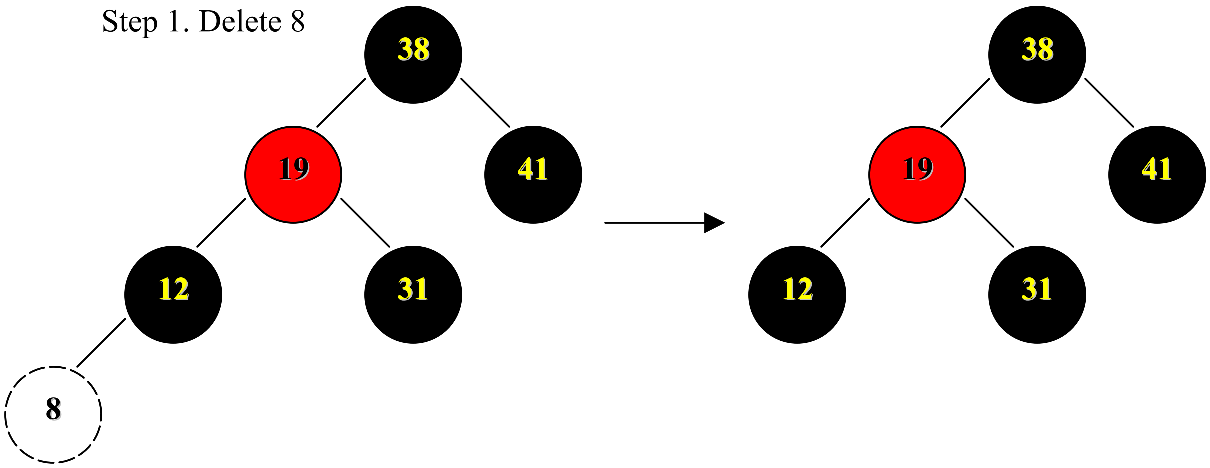
14. 13.4-3

In exercise 13.3-2 (problem 12), you found the red-black tree that results from successively inserting the keys 41, 38, 31, 12, 19, and 8 into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order 8, 12, 19, 31, 38, and 41.

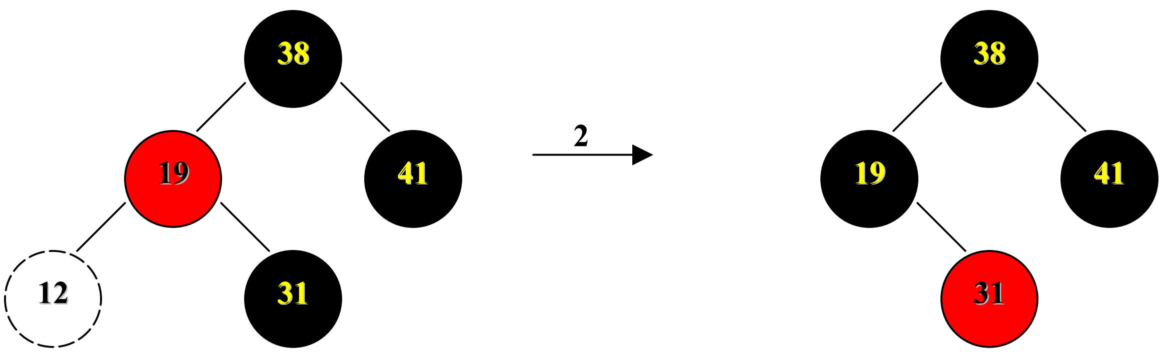


= Node to be deleted

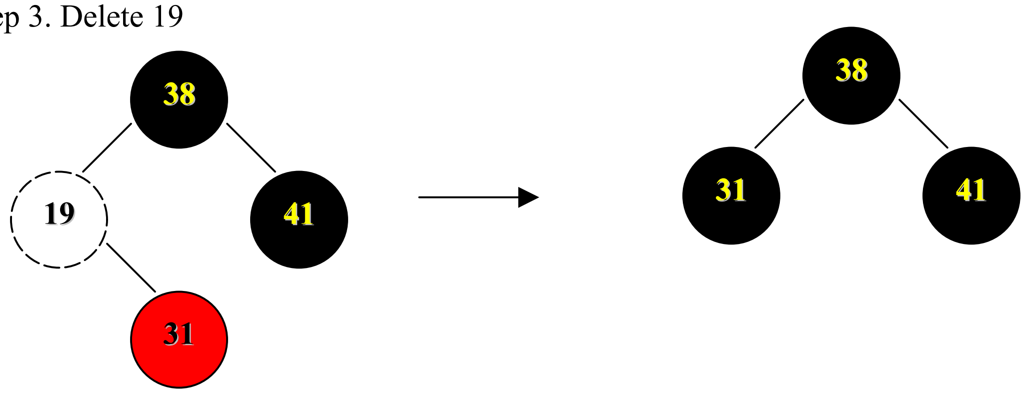
Step 1. Delete 8



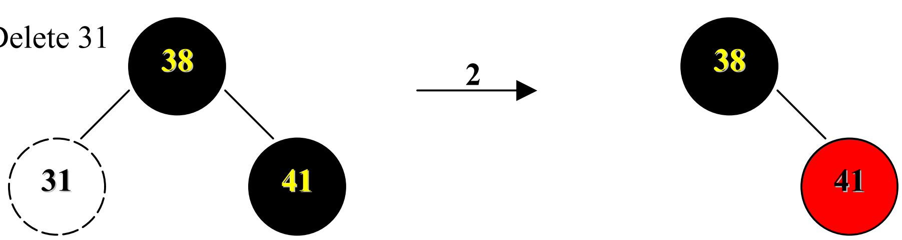
Step 2. Delete 12



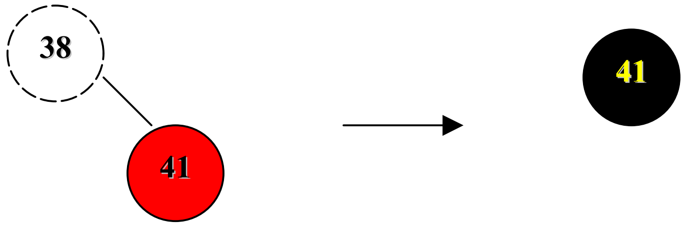
Step 3. Delete 19



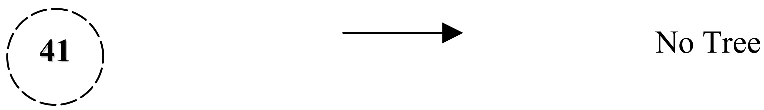
Step 4. Delete 31



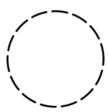
Step 5. Delete 38



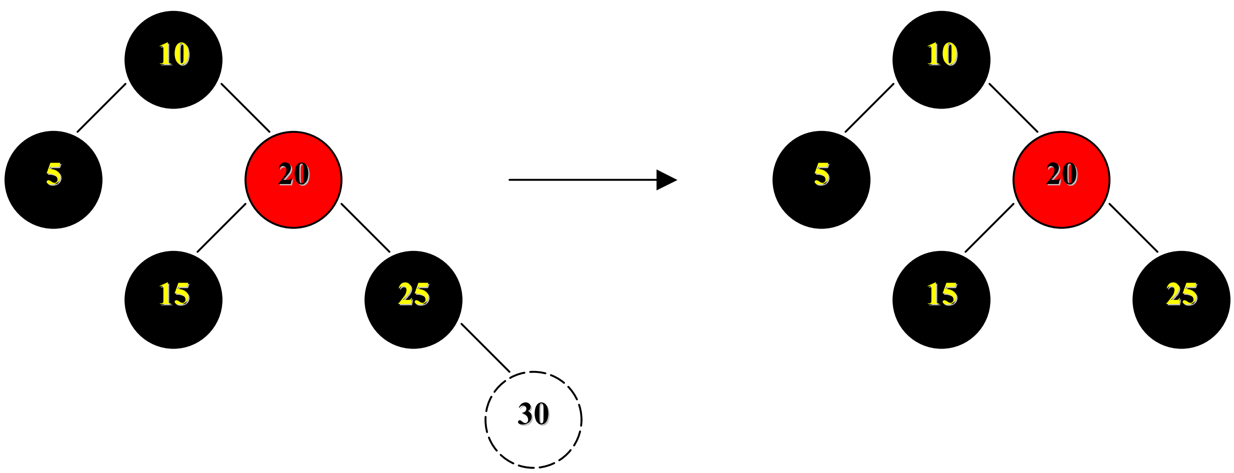
Step 6. Delete 41



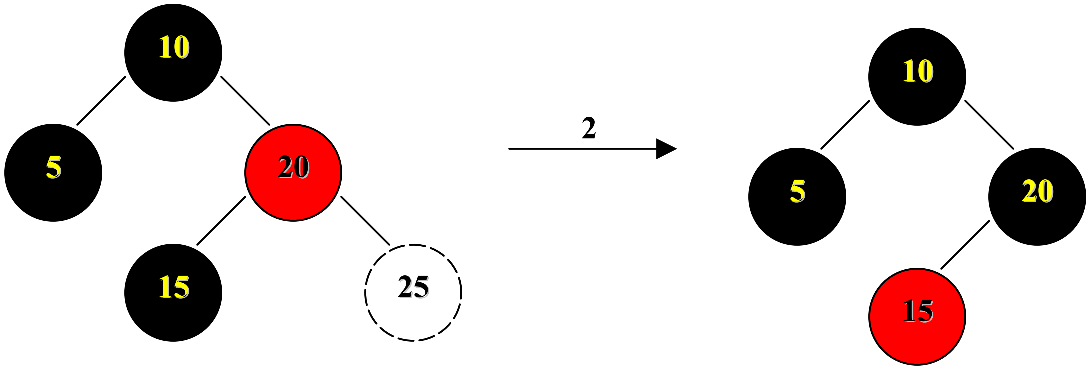
15. Show the red-black tree that results from successively deleting the keys 30, 25, 20, 15, 10, and 5 from the final tree in problem 13.



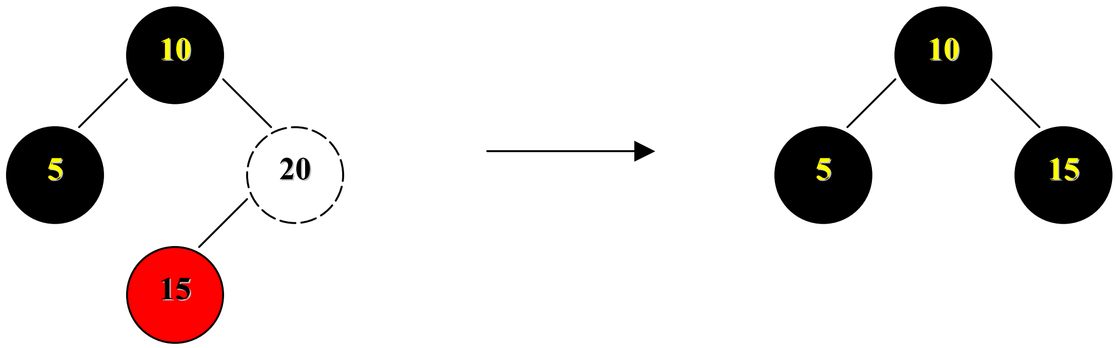
Step 1. Delete 30



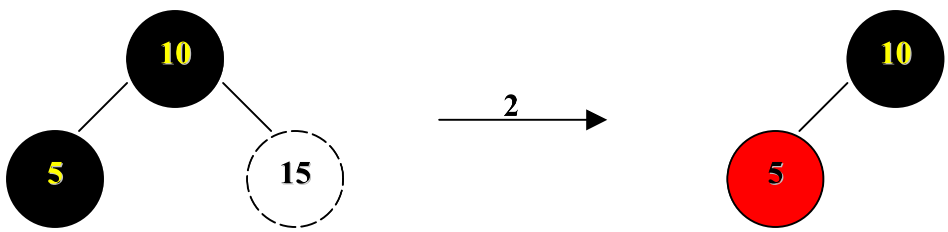
Step 2. Delete 25



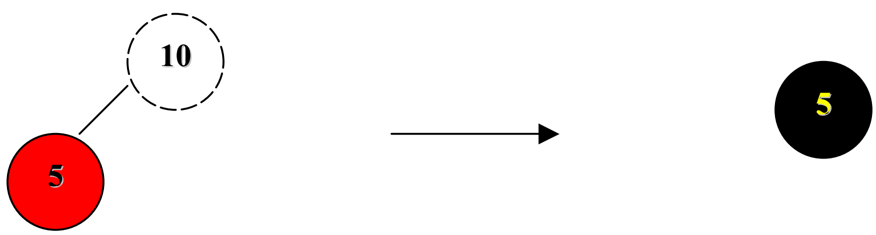
Step 3. Delete 20



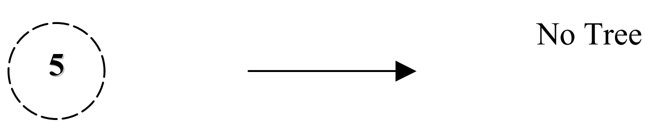
Step 4. Delete 15



Step 5. Delete 10



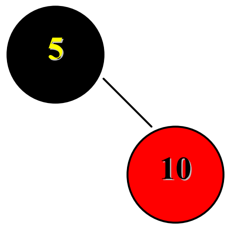
Step 6. Delete 5



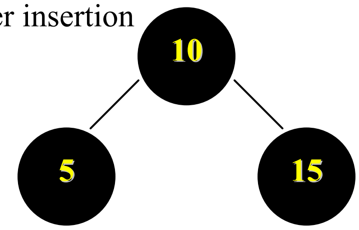
16. 13.4-7

Suppose that a node x is inserted into a red-black tree with RB-INSERT an then immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your answer.

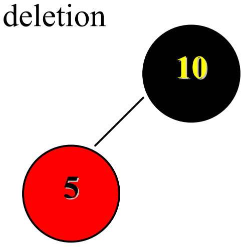
No. The tree after insertion and a deletion of the same node may or may not be different. Let us insert and delete node 15 into the following tree:



After insertion

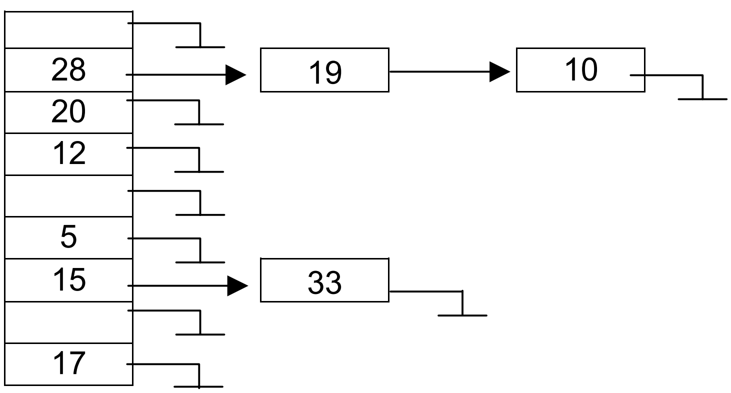


After deletion



17. 11.2-2

Demonstrate the insertion of the keys 5, 28, 19, 15, 20, 33, 12, 17, and 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be h(k) = k mod 9.



18. 11.3-1

Suppose we wish to search a linked list of length n, where each element contains a key k along with a hash value h(k). Each key is a long character string. How might we take advantage of the hash values when searching the list for an element with a given key?

Each key is a long character thus to compare keys, at every node we need to perform a string comparison operation which is very time consuming. Instead we generate a hash value for the key (i.e., generate a numeric value for each string) we are searching for and comparing hash values h(k) along the length of the list, which turns out to be numeric values and the comparison is faster.

19. 11.4-1

Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, and 59 into a hash table of length m=11 using open addressing with the primary hash function h'(k) = k mod m. Illustrate the result of inserting these keys using linear probing, using quadratic probing with $c_1 = 1$and $c_2 = 3$, and using double hashing with $h_2(k) = 1 + (k \bmod (m-1))$.

Using Linear probing the final state of the hash table would be:

0	22
1	88
2	
3	
4	4
5	15
6	28
7	17
8	59
9	31
10	10

Using Quadratic probing, with $c_1 = 1$, $c_2=3$) the final state of the hash table would be $h(k,i) = (h'(k) + c_1 * i + c_2 * i^2) \bmod m$ where m=11 and h'(k) = k mod m.

0	22
1	88
2	
3	17
4	4
5	
6	28
7	59
8	15
9	31
10	10

Using double hashing the final state of the hash table would be:

0	22
1	
2	59
3	17
4	4
5	15
6	28
7	88
8	
9	31
10	10

20. 11.4-4

Consider an open-address hash table with uniform hashing. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is ¾ and when it is 7/8.

Theorem 11.6. Given an open address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

$\alpha = ¾$, then the upper bound on the number of probes = 1 / (1 - ¾) = **4 probes**

$\alpha = 7/8$, then the upper bound on the number of probes = 1 / (1-7/8) = **8 probes**

Theorem 11.8. Given an open address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in a successful search is at most $(1/\alpha) \ln (1/(1-\alpha))$, assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

$\alpha = ¾$. (1/ ¾) ln (1/ (1 – ¾)) = **1.85 probes**

$\alpha = 7/8$. (1/ .875) ln (1/ (1 – .875)) = **2.38 probes**