1. 7.1-1

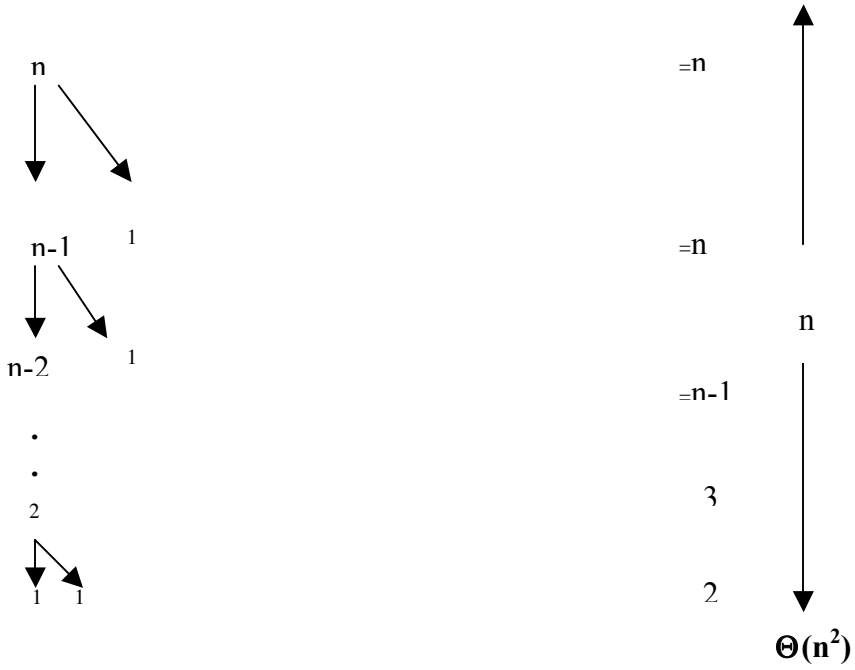
```
AB13   19    9    5   12    8    7    4   21    2    6   11
A 13  B19    9    5   12    8    7    4   21    2    6   11
A 13   19  B 9    5   12    8    7    4   21    2    6   11
   9 A 19   13  B 5   12    8    7    4   21    2    6   11
   9    5 A 13   19  B12    8    7    4   21    2    6   11
   9    5 A 13   19   12  B 8    7    4   21    2    6   11
   9    5    8 A 19   12   13  B 7    4   21    2    6   11
   9    5    8    7 A 12   13   19  B 4   21    2    6   11
   9    5    8    7    4 A 13   19   12  B21    2    6   11
   9    5    8    7    4 A 13   19   12   21  B 2    6   11
   9    5    8    7    4    2 A 19   12   21   13  B 6   11
   9    5    8    7    4    2    6 A 12   21   13   19  B11
   9    5    8    7    4    2    6  <11 > 21   13   19   12
```

2. & 3. 7.1-2, 7.2-2

Show that the running time of the quicksort is $\Theta\left(n^2\right)$ when all the elements in the array have the same value.

Solution: It is a special case of quicksort, when all the elements of the array have the same value. **For this special case the algorithm "partition" always returns the value of q as r.** Where r is the size of the partition (i.e. passed to the partition algorithm). So the array gets divided by (n-1) and 1 elements

Thus if we have an array of n elements, we can write the following binary tree



$T(n) = T(n-1) + \Theta(1)$

$= \sum_{k=1}^{n} \Theta(k)$

$= \Theta(\sum_{k=1}^{n})$

$T(n) = \Theta(n^2)$

PARTITION Modification

To return q = (p+r)/2 when all elements in the array has the same value

```
PARTITION (A,p,r)
 1. x = A[r]
 2. i = p-1
 3. flag = 0
 4. for j = p to r-1
 5.       do if A[j] ≤ x
 6.             then i = i +1
 7.                   exchange A[i]⇔ A[j]
 8.                   if A[j] ≠ x
 9.                      flag = 1
10. exchange A[i+1]⇔ A[j]
11. if  flag = 1
12.       return i+1
13. else
14.       return (p+r)/2
```

4. 8.2-1.

(a)

A

1	2	3	4	5	6	7	8	9	10	11
7	1	3	1	2	4	5	7	2	4	3

C

1	2	3	4	5	6	7
2	2	2	2	1	0	2

(b)

C

1	2	3	4	5	6	7
2	4	6	8	9	9	11

(c)

B

1	2	3	4	5	6	7	8	9	10	11
					3					

C

1	2	3	4	5	6	7
2	4	5	8	9	9	11

(d)

B

1	2	3	4	5	6	7	8	9	10	11
					3		4			

C

1	2	3	4	5	6	7
2	4	5	7	9	9	11

(e)

B

1	2	3	4	5	6	7	8	9	10	11
			2		3		4			

C

1	2	3	4	5	6	7
2	3	5	7	9	9	11

(f)

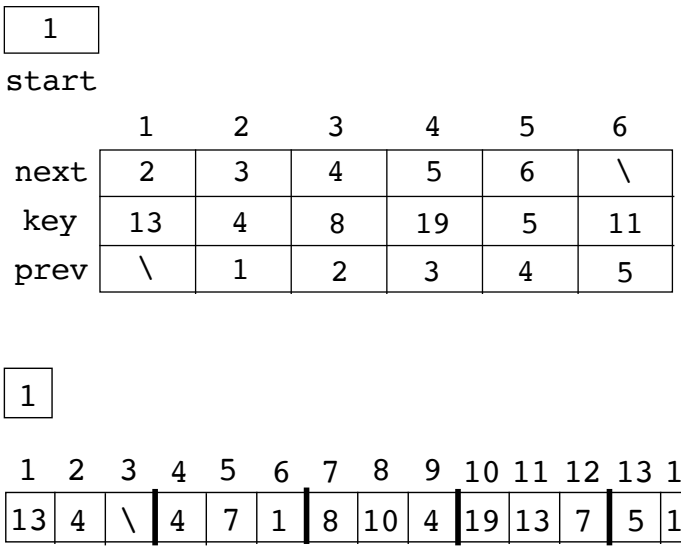
B

1	2	3	4	5	6	7	8	9	10	11
1	1	2	2	3	3	4	4	5	7	7

5. 8.3-1.

Initial	Step1	Step2	Step3
COW	SEA	BAR	BAR
DOG	TEA	EAR	BIG
SEA	MOB	TAB	BOX
RUG	TAB	TAR	COW
ROW	DOG	SEA	DIG
MOB	RUG	TEA	DOG
BOX	DIG	DIG	EAR
TAB	BIG	BIG	FOX
BAR	BAR	MOB	MOB
EAR	EAR	DOG	NOW
TAR	TAR	COW	ROW
DIG	COW	ROW	RUG
BIG	ROW	NOW	SEA
TEA	NOW	BOX	TAB
NOW	BOX	FOX	TAR
FOX	FOX	RUG	TEA
COLUMN AFFECTED	↑	↑	↑

6. 10.3-1

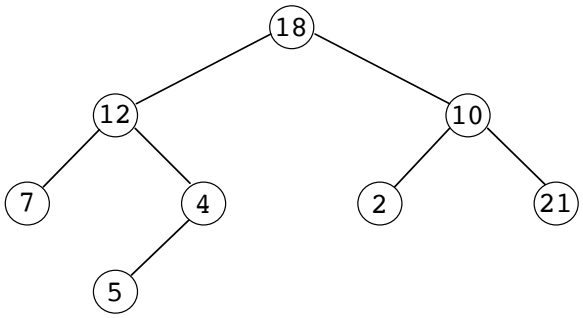


7. 10.3-2

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
}
```



8. 10.4-1

9. 10.4-4

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

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

10. 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(1)

11. 12.2-1

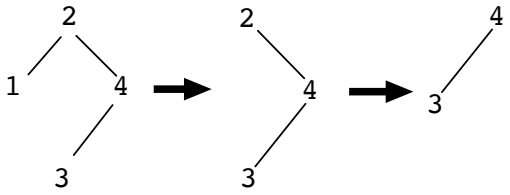
```
c and e

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
```

12. 12.3-4

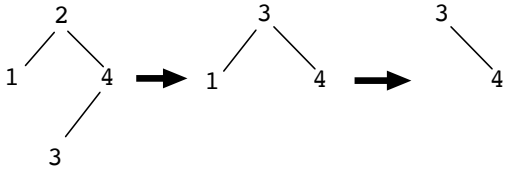
False

Below is a counter-example

Deleting 1 then 2

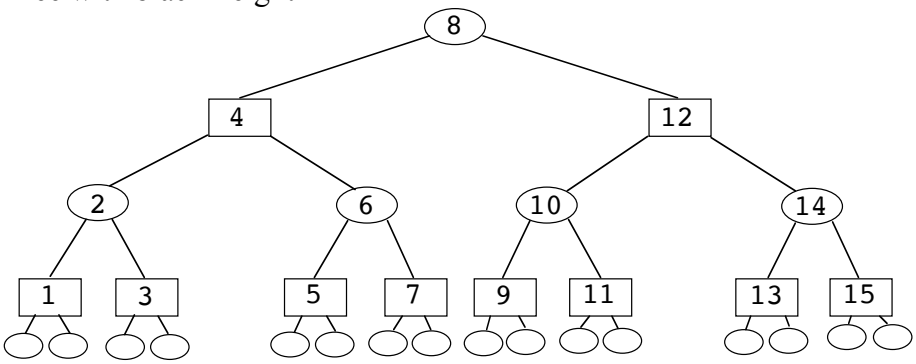


Deleting 2 then 1

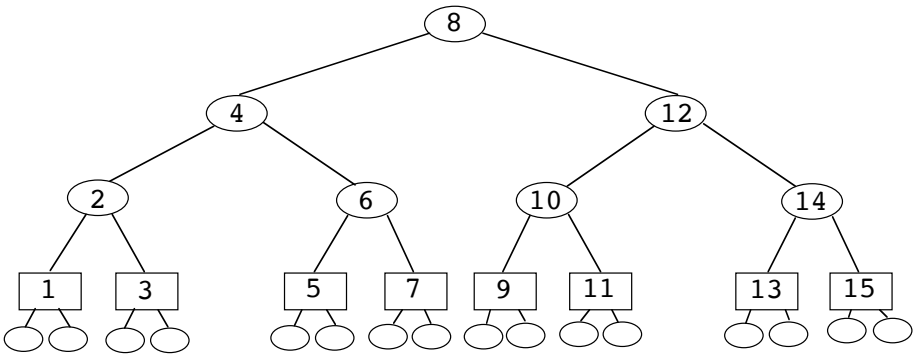


13. 13.1-1

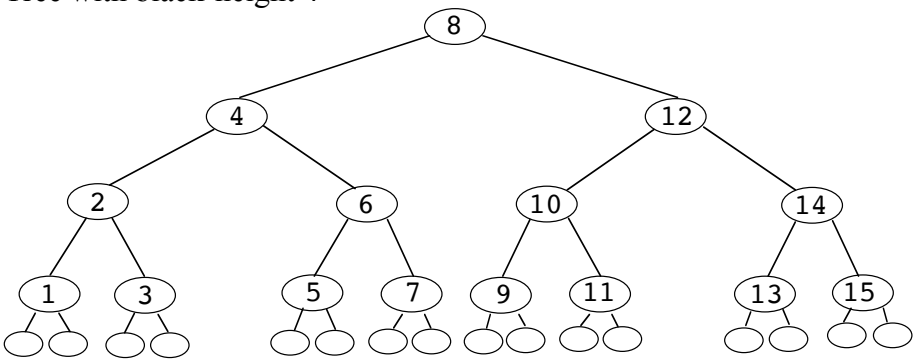
Tree with black-height 2



Tree with black-height 3



Tree with black-height 4



14. 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.

15. 13.2-3

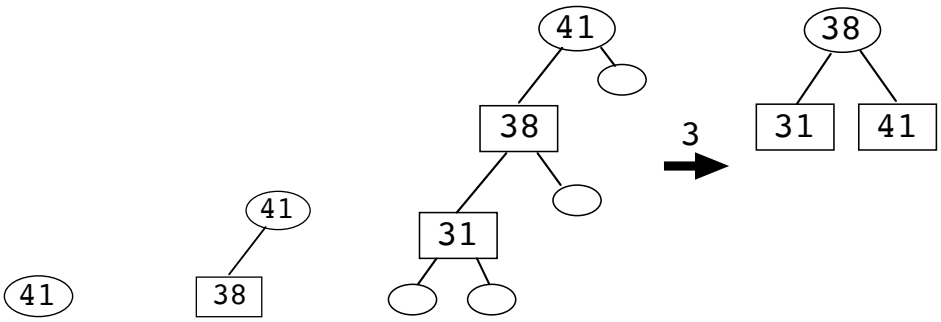
Let a, b, and c be arbitrary nodes in subtrees α, β, and γ, respectively, in the left tree of Figure 13.2. How do the depths of a, b, and c change when a left rotation is performed on node x in the figure?

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

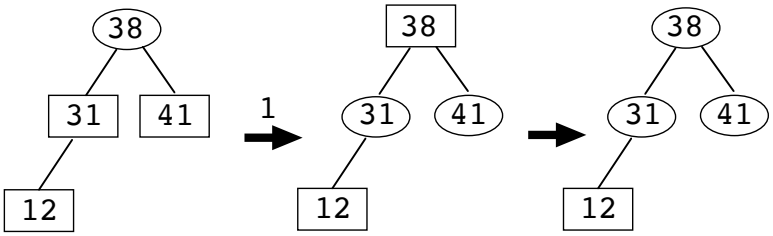
16. 13.3-2

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.

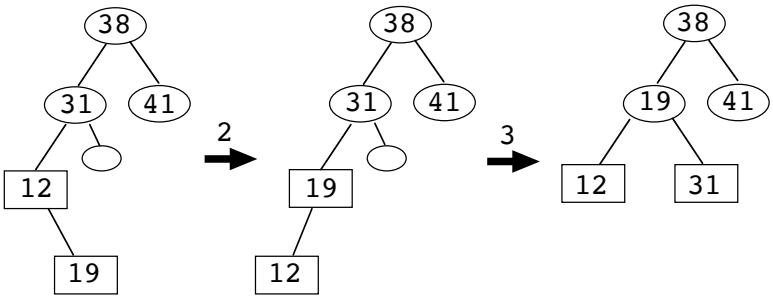
Insert 41 Insert 38 Insert 31



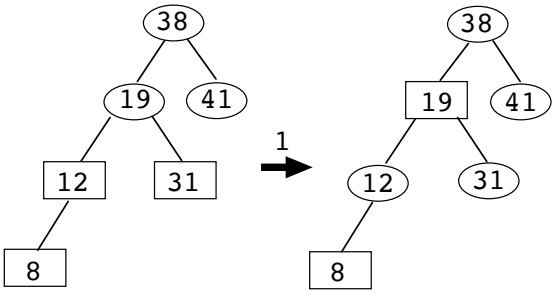
Insert 12



Insert 19



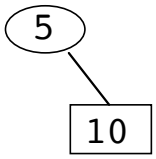
Insert 8



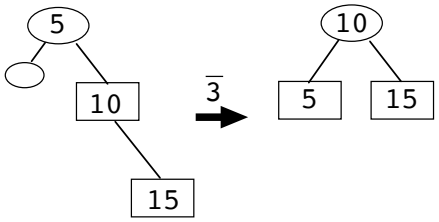
17. Show the red-black trees that result after successively inserting the keys 5, 10, 15, 25, 20, and 30 into an initially empty red-black tree.

Insert 5

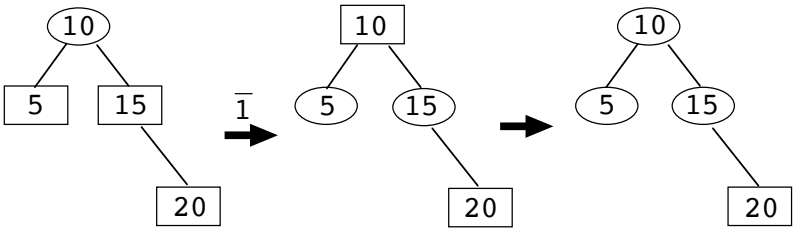
Insert 10



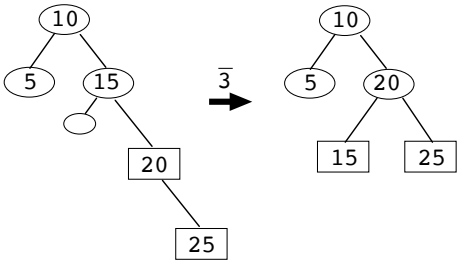
Insert 15



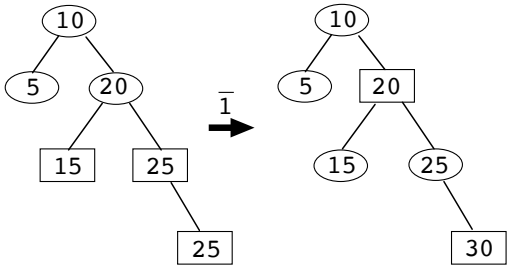
Insert 20



Insert 25

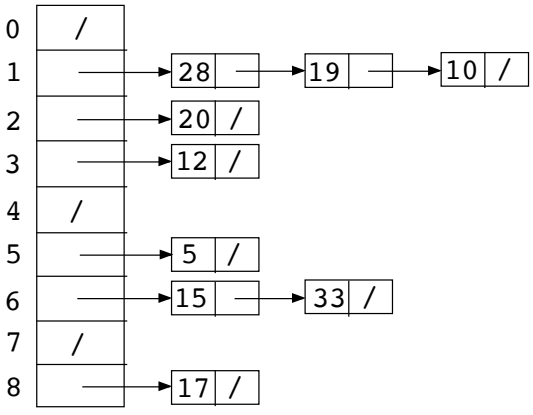


Insert 30



18. 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.



19. 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?

First compute the hash value for the given key. For each list element, perform the string comparison only after verifying that the hash value for the given key is the same as the one stored in the list element.

20. 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 = 1and c_2 = 3, and using double hashing with $h_2(k) = 1 + (k \bmod (m-1))$.

Linear probing

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

Quadratic Probing

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

Double Hashing

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

21. 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 $\frac{3}{4}$ and when it is $\frac{7}{8}$.

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

$\alpha = \frac{3}{4}$, then the upper bound on the number of probes = $\frac{1}{1-\frac{3}{4}}$ = **4 probes**

$\alpha = \frac{7}{8}$, then the upper bound on the number of probes = $\frac{1}{1-\frac{7}{8}}$ = **8 probes**

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

$\alpha = \frac{3}{4}$. $\frac{1}{\frac{3}{4}}\ln\frac{1}{1-\frac{3}{4}}$ = **1.85 probes**

$\alpha = \frac{7}{8}$. $\frac{1}{\frac{7}{8}}\ln\frac{1}{1-\frac{7}{8}}$ = **2.38 probes**