Name: Pheakdey Luk
ID: 986591

Assignment 10

C-4.16 Given a sequence S of *n* comparable elements, describe an efficient method for determining whether there are two equal elements in S. What is the running time of your method?

Answer:

```
Algorithm isExistTwoEqualElement(S)
     Input: Sequence S with n elements
     Output: true or false if two
equal elements exis in the
sequence,otherwise false
     D<-new Dictionary(HashTable)
     for each x of S do
          cnt <-D.findElement(x)
          if cnt != NO_SUCH_KEY then
               D.insertItem(x,0)
          else
               return true
     return false
```

L1: O(1)
L2:O(n)
L3:O(n)
L4:O(n)
L5:O(n)
L6:O(1)
L7:O(1)
Total running time = O(n)

C-4.18 Modify Algorithm inPlaceQuickSort (Algorithm 4.17) to handle the general case efficiently when the input sequence, S, may have duplicate keys.

Answer:

```
Algorithm inPlacePartition(S, lo, hi)
     Input: Sequence S and ranks lo and hi, 0 <= lo,hi < S.size()
     Output: Skip duplicate keys in the next partition

     p <-- a random integer between lo and hi
     S.swapElements(S.atRank( lo ), S.atRank( p ))
     pivot <-- S.elemAtRank(lo)
     j <-- lo + 1
     k <-- hi
     while  j < k  do
          while k >= j ^ S.elemAtRank(k) > pivot do
               k <-- k  1
          while j <= k ^ S.elemAtRank(j) < pivot do
               j <--  j + 1
          if  j < k  then
               S.swapElements(S.atRank( j ), S.atRank( k ))
     S.swapElements(S.atRank( lo ), S.atRank( k )) {move pivot to sorted rank}

     return k
```

C-4-19 Let S be a sequence of n elements on which a total order relation is defined. An ***inversion*** in S is a pair of elements x and y such that x appears before y in S but x > y. Describe an algorithm running in O(n log n) time for determining the number of inversions in S. **Hint:** try to modify the merge-sort algorithm to solve this problem.

Answer:

```
Algorithm countInversion(S, C)
    Input : sequence S with total order
n elements, comparator C
    Output: number of Inversion

    if S.size() > 1 then
        (S1, S2)<-partition(S, n/2)
        countInversion(S1, C)
        countInversion(S2, C)
        (S,cnt) <-merge(S1, S2, C)
        count <- count + cnt
    return count
```

```
Algorithm merge(A, B, C)
    Input: sequences A and B with n/2
elements each, comparator C
    Output: count of number of
inversion
    count<-0
    S <- empty sequence
    while !A.isEmpty() ^ !B.isEmpty()
do
        if C.isLessThan(
B.first().element(),
A.first().element() ) then

S.insertLast(B.remove(B.first()))
            count <- count + 1
        else

S.insertLast(A.remove(A.first()))
        while !A.isEmpty() do

S.insertLast(A.remove(A.first()))
        while !B.isEmpty() do

S.insertLast(B.remove(B.first()))
    return count,S
```

C-4.25 Bob has a set A of *n* nuts and a set B of *n* bolts, such that each nut in A has a unique matching bolt in B. Unfortunately, the nuts in A all look the same, and the bolts in B all look the same as well. The only kind of comparison that Bob can make is to take a nut-bolt pair (*a,b*), such that *a* is from A and *b* is from B, and test it to see if the threads are larger, smaller or a perfect match with the threads of *b*. Describe an efficient algorithm for Bob to match up all of his nuts and bolts. What is the running time of this algorithm, in terms of nut-bolt tests that Bob must make?

Answer:

```
Algorithm nutsBoltsMatchup(A, B)
    Input : Sequence A of nuts, sequence B of bolts
    Output : Matched set of nuts and bolts

    T <- insertIntoRedBlackTree(B)
    PQ <- new Priority Queue Array
    for each x of A do
        PQ.insert(x, PQ.remove(x))

    return PQ
```

Design a pseudo code algorithm **createBST(S)** that takes a sorted Sequence S of numbers and creates a balanced binary search tree with height O(log n). Hint: start with an empty tree T and insert the nodes using operation **expandExternal(v)** where **v** is an external node. Another hint: in the new tree T, a search for a key will reflect a binary search in a sorted Sequence or Array (drawing the picture from an example should help).

What is the time complexity of your algorithm?

Given a Tree ***T***, write a pseudo code algorithm **findDeepestNodes(T)**, that returns a Sequence of pairs **(*v*, *d*)** where ***v*** is an internal node of tree ***T*** and ***d*** is the depth of ***v*** in ***T***. The function must return all internal nodes that are at the maximum depth. What is the time complexity of your algorithm?