1 + 2^2 + 3^2 + ... + n^2

n(n+1)(2n+1)/6

Algorithm countBooksByPublisher(B, P)

1 Pub <- new Dictionary(Hashtable)

30 for each p in P do

30 Pub.insertItem(p, 0)

n for each (title, author, call number, publisher) in B.elements() do

n cnt <- Pub.findElement(publisher) //return value/count of the key publisher, not position

n if q <> NO\_SUCH\_KEY then

n Pub.replaceElement(publisher, cnt+1)

1 return Pub

Total running time = 4n + 2\*30 + 2 = O(n)

Algorithm unPaidBill(B, P)

if B.size() = 0 \/ P.size() then

return null

CCard <- new Dictionary(BST)

for each b in B.elements() do

cc <- new CreditCard(num, name, address, amountDue, amountPaid)

cc.num <- b.creditCardNumber

cc.name <- b.name

cc.address <- b.address

cc.amountDue <- b.amountDue

cc.amountPaid <- 0

CCard.insertItem(cc.num, cc)

for each p in P.elements() do

b = CCard.findElement(p.creditNumber)

if b <> NO\_SUCH\_KEY then

if p.amountPaid >= b.amountDue then

CCard.removeElement(b.num)

else

b.amountPaid <- b.amountPaid + p.amountPaid

CCard.replaceElement(b)

return CCard

Algorithm calHeightAndBalanceFactor(T)

if T.isEmpty() = true then

return

\_calH\_BL(T, T.root())

\_calcH\_BL(T, v)

if T.isExternal(v) then

return 0

hl <- \_calcH\_BL(T, T.leftChild(v))

hr <- \_calcH\_BL(T, T.rightChild(v))

bl <- hl - hr

h <- 1 + max(hl, hr)

T.setHeight(v, h)

T.setBalFactor(v, bl)

return h

//

Algorithm findAllInRange(D, k1, k2)

Iterator iter <- new Iterator

if D.isEmpty() = true then

return iter

Dr <- new Dictionary(BST)

Iterator dIter <- D.keys()

while dIter.hasNext() do

p <- dIter.nextObject()

if p.key() > k1 /\ p.key() < k2 then

Dr.insertItem(p.key(), p.element())

else

if p.key() >= k2 then

break

rIter <- Dr.keys()

return iter

Algorithm isPermutation(A, B)

D <- new Dictionary(HT)

for each a in A.elements() do

D.insertElement(a, a)

for each b in B.elements() do

p <- D.findElement(b)

if p = NO\_SUCH\_KEY then

return false

else

D.removeElement(b)

return true

//Need to recheck

Algorithm isPermutation(A, B)

if A.size() != B.size() then

return false

H1 <- \_buildHeap(A)

H2 <- \_buildHeap(B)

n <- A.size()

for i<-0 to n-1 do

p1 <- H1.removeMin()

p2 <- H2.removeMin()

if p1.element() != p2.element() then

return false

return true

//Need to recheck

Agorithm isSameElements(A, B)

if A.size() <> B.size() then

return false

\_inPlaceQuickSort(A, 0, A.size())

\_inPlaceQuickSort(B, 0, B.size())

for i<-0 to A.size() do

p <- A.elementAtRank(i)

q <- B.elementAtRank(i)

if p.element() <> q.element() then

return false

return true

\_inPlaceQuickSort(S, l, h)

if l < h then

k <- \_inPlacePartition(S, l, h)

\_inPlaceQuickSort(S, l, k-1)

\_inPlaceQuickSort(S, k+1, h)

\_inPlacePartition(S, l, h)

p <- randomInt(l, h)

S.swapElement(S.atRank(l), S.atRank(p))

j <- l+1

k <- h

while j < k do

while k > j /\ S.elementAtRank(k) >= S.elementAtRank(l) do

k <- k - 1

while j < k /\ S.elementAtRank(j) <= S.elementAtRank(l) do

j <- j + 1

if j < k then

S.swapElement(S.elementAtRank(j), S.elementAtRank(k))

S.swapElement(S.elementAtRank(k), S.elementAtRank(l)

return k

//Need to recheck

Algorithm removeDuplicateAndUnion(A, B)

Input:sequences A and B with n elements each

Output:sorted sequence of A U B

S <- empty sequence

while !A.isEmpty() /\ !B.isEmpty() do

if B.first().element() < A.first().element() then

S.insertLast(B.remove(B.first()))

else if B.first().element() > A.first().element() then

S.insertLast(A.remove(A.first()))

else

S.insertLast(A.remove(A.first()))

B.remove(B.first())

return S

Algorithm findPair(A, B, x)

Input: n-element sequence A and B include n integers

Ouput: true if existing a pair a & b so that a + b = x

B <- new Dictionary(HT)

for each n in B do

B.insertItem(n, n)

for each a in A.elements() do

b <- B.findElement(x - a)

if b <> NO\_SUCH\_KEY then

return true

}

return false

//Using Dictionary

Algorithm findWinner(S, C)

B <- new Dictionary(BST)

cnt <- 0

for each id in C do

B.insertItem(id, cnt)

maxVote <- 0

winnerID <- 0

v <- 0

for i<-0 to S.size()-1 do

v <- S.elementAtRank(i) //return candidate ID at the sequence i

p <- B.findElement(v)

if p <> NO\_SUCH\_KEY then

cnt <- B.elem(p) + 1

B.insertElement(B.key(p), cnt)

if cnt > max then

max <- cnt

winnerID <- B.key(p)

return winnerID

//Using Dictionary and PriorityQueue

Algorithm findWinner(S)

if S.size() = 0 then

return 0

D <- new Dictionary(HT)

for each id in S.elements() do

p <- D.findElement(id)

if p = NO\_SUCH\_KEY then

D.insertItem(id, 1)

else

cnt <- D.elem(p)

D.replaceElement(id, cnt+1)

P <- new MaxPriorityQueue()

for each (id, cnt) in D.items() do

P.insertItem(cnt, id)

maxVote <- P.maxKey()

winnerID <- P.maxElement()

return winnerID

Algorithm isAVLTree(T)

Input: T is BST

Output: true of the tree T is AVL tree

if T.isEmpty() = true then

return false

\_isAVLSubTree(T, T.root())

\_isAVLSubTree(T, v)

l <- T.leftChild(v)

r <- T.rightChild(v)

if T.isExternal(l) = true /\ T.isExternal() = true then

return true

if T.isExternal(l) then

return \_isNotParent(r)

else

if T.isExternal(r) then

return \_isNotParent(l)

return \_isAVLSubTree(l) /\ \_isAVLSubTree(r)

\_isNotParent(T, v)

l <- T.leftChild()

r <- T.rightChild()

return (T.isExternal(l) /\ T.isExternal(r))

\_heightTree(T, v)

if T.isExternal(v) then

return 0

l <- T.leftChild(v)

r <- T.rightChild(v)

hl <- heightTree(T, l)

hr <- heightTree(T, r)

return 1 + max(hl, hr)