**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

**//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)

**//DFS**

**Algorithm DFS(G)**

Iterator verticesIter <- G.vertices()

while verticesIter.hasNext() do

v <- verticesIter.nextItem()

setLabel(v, UNEXPLORED)

Iterator edgesIter <- G.edges()

while edgesIter.hasNext() do

e <- edgesIter.nextItem()

setLabel(e, UNEXPLORED)

for each v in verticesIter do

if getLabel(v) = UNEXPLORED then //need to check this when all vertices are initialized to UNEXPLORED?

DFS(G, v)

**//Using recursion**

**\_DFS(G, v)**

if getLabel(v) = UNEXPLORED then

setLabel(v, VISITED)

Iterator edgesIter <- G.incidentEdges(v)

for each e in edgesIter do

if getLabel(e) = UNEXPLORED then

w <- G.apposite(v, e)

if getLabel(w) = UNEXPLORED then

setLabel(e, DISCOVERY) // draw a straight line

\_DFS(G, w)

else setLabel(e, BACK) //draw a dot line

else

//e is DISCOVERY or BACK, therefore no update more

**//Using template method**

**public abstract class DFS** {

protected \_DFS\_Template(G, v){

checkInput(G, v)

initial()

mark(v)

startVisit(v)

Iterator edgesIter <- queryIncidentEdges(v)

for each e in edgesIter do

if isMarked(e) = false then

discoveryEdge(v, e)

w <- G.opposite(v, e)

if isMarked(w) = false then

mark(e)

\_DFS\_Template(G, w)

else

backEdge(v, e)

finishVisit(v)

**//Using a stack instead of recursion**

**\_DFS(G, v)**

if G.degree(v) = 0 then

setLabel(v, VISITED)

else

S <- new Stack()

setLabel(v, VISITED)

S.push(v)

while not S.isEmpty() do

v <- S.pop() //S.top() //it should be popped when checked its incidentEdges and pushed them to the stack if UNEXPLORED yet

//setLabel(v, VISITED)

//bDiscovered <- true

for each e in G.incidentEdges(v) do

if getLabel(e) = UNEXPLORED then

setLabel(e, DISCOVERY)

w <- S.opposite(v, e)

if getLabel(w) = UNEXPLORED then

setLabel(w, VISITED)

S.push(w) // just pushed to the stack regardless of having incident edges or not

else

setLabel(e, BACK)

else

// do nothing

-------------------------------------------------------------------------

**//find simple path**

**\_DFS(G, v, z, S)**

S.push(v)

for each e in G.incidentEdges(v) do

-------------------------------------------------------------------------

**//BSF without recursion**

**Algorithm BFS(G)**

Iterator verticesIter <- G.vertices()

while verticesIter.hasNext() do

v <- verticesIter.nextItem()

setLabel(v, UNEXPLORED)

Iterator edgesIter <- G.edges()

while edgesIter.hasNext() do

e <- edgesIter.nextItem()

setLabel(e, UNEXPLORED)

Q <- new Queue()

v <- G.aVertex()

setLabel(v, VISITED)

Q.enqueue(v)

while not Q.isEmpty() do

v <- Q.dequeue()

for each e in G.incidentEdges(v) do

w <- G.opposite(v, e)

if getLabel(w) = UNEXPLORED then

setLabel(e, DISCOVERY)

setLabel(w, VISITED)

Q.enqueue(w)

else

setLabel(e, BACK)

-------------------------------------------------------------------------

**//BSF applying the template method**

**Algorithm BFS\_TemplateMethod(G, startV)**

initResult(G) //Template

Iterator verticesIter <- G.vertices()

while verticesIter.hasNext() do

vt <- verticesIter.nextItem()

setLabel(vt, UNEXPLORED)

preInitVertex(vt) //Template

Iterator edgesIter <- G.edges()

while edgesIter.hasNext() do

e <- edgesIter.nextItem()

setLabel(e, UNEXPLORED)

preInitEdge(e) //Template

Q <- new Queue()

//v <- G.aVertex()

setLabel(v, VISITED)

startVertexVisit(G, v) //Template

Q.enqueue(startV)

while not Q.isEmpty() do

v <- Q.dequeue()

for each e in G.incidentEdges(v) do

w <- G.opposite(v, e)

if getLabel(w) = UNEXPLORED then

preDiscoveryTraversal(G, v, e, w)  //Template

setLabel(e, DISCOVERY)

setLabel(w, VISITED)

Q.enqueue(w)

postDiscoveryTraversal(G, v, e, w)  //Template

else

setLabel(e, BACK)

backTraversal(G, v, e, w) //Template

finishVertexVisit(G, startV) //Template

-------------------------------------------------------------------------

**startVertexVisit(G, v)**

setLevel(v, 0)

**postDiscoveryTraversal(G, v, e, w)**

l <- v.getLevel()

setLevel(w, l)

-------------------------------------------------------------------------

**Algorithm findPath(G, u, v)**

S <- new Stack

minPath <- 0

BFS\_TemplateMethod(G, v)

return minPath

initResult(G)

minVertex <- v

minEdges <- 0

startVertexVisit(G, v)

setParent(v, 0)

setLevel(v, 0)

postDiscoveryTraversal(G, v, e, w)

setParent(w, e)

l <- getLevel(v) + 1

setLevel(w, l)

if w = u /\ l < minEdges then

minVertex <- w

minEdges = l

finishVertexVisit(G, v)

if minVertex = v then

return minPath

//Using the backtracking to find the path with minimum number of edges

S <- new Stack

z <- minVertex

while z <> v do

S.push(z)

e <- z.getParent()

S.push(e)

z <- G.opposite(z, e)

S.push(v)

minPath <- S.elements()

return minPath

-------------------------------------------------------------------------

findCycle(G)

cycle <- new Sequence

startV <- G.aVertex()

BFS\_TemplateMethod(G, startV)

return cycle

startVertexVisit(G, v)

setParent(v, 0)

backTraversal(G, v, e, w)

**// How to trace back the cycle???**

D1 <- new Dictionary(HT)

D1.insertItem(v, v.getParent())

D2 <- new Dictionary(HT)

D2.insertItem(w, w.getParent())

f <- false

while f = false do

v <- G.opposite(v, v.getParent())

w <- G.opposite(w, w.getParent())

if(v = w)

f <- true

else

p <- D2.findElement(v)

if p <> NO\_SUCH\_KEY then

makeCycle(cycle, D2, p, D1)

f <- true

else

D1.insertItem(v, v.getParent())

p <- D1.findElement(w)

if p <> NO\_SUCH\_KEY then

makeCycle(cycle, D1, p, D2)

f <- true

else

D2.insertItem(w, w.getParent())

makeCycle(cycle, B1, p, B2)

**//Output p items from D2**

Vs <- B1.keys()

Es <- B1.elements()

i <- 0

for each (k,f) in B1.items() do

cycle.insertLast(k)

cycle.insertLast(f)

i++

if i = p then //just get p items

break

//Output edge e between v & w

cycle.insertFirst(e)

//Output all items from D1

for each (k,f) in B2.items() do

cycle.insertFirst(k)

cycle.insertFirst(f)

-------------------------------------------------------------------------

**// Dijkstra algorithm**

**// Using new methods: set/getLabel(),** set/getDistance(), weight(), replaceKey(u: element of key, new key)

**Algorithm Dijkstra(G)**

**//1. Select a start vertex**

s <- G.aVertex()

H <- new Heap()

**//2. Set other vertices to the infinitive**

for each v in G.vertices() do

setLabel(v, UNEXPLORED) //new

if s = v then

H.insertItem(0, s)

else

setDistance(v, infinitive)

H.insertItem(infinitive, v)

**//3. Set distance for endpoints of the start vertex's edges by applying the relaxation for other vertices**

a. find the smallest distance

b. update the distance of the related endpoint if its is greater than

while not H.isEmpty() do

v <- H.removeMin()

setLabel(v, VISITED) //new

for each e in G.incidentEdges(v) do

u <- G.oppsite(v, e)

if getLabel(u) = UNEXPLORED then //new

d <- getDistance(v) + weight(e)

du <- getDistance(u)

if (d < du) then

setDistance(u, d)

H.replaceKey(u, d)

-------------------------------------------------------------------------

// Dijkstra algorithm (using Locator)

// Using new methods: insert(k, e), replaceKey(u: element of key, new key)

**Algorithm Dijkstra(G)**

**//1. Select a start vertex**

s <- G.aVertex()

H <- new Heap()

**//2. Set other vertices to the infinitive**

for each v in G.vertices() do

setLabel(v, UNEXPLORED) //new

if s = v then

setDistance(v, 0)

else setDistance(v, infinitive)

l <- H.insert(getDistance(v), v)

H.setLocator(v, l)

**//3. Set distance for endpoints of the start vertex's edges by applying the relaxation for other vertices**

a. find the smallest distance

b. update the distance of the related endpoint if its is greater than

while not H.isEmpty() do

v <- H.min()

setLabel(v, VISITED) //new

for each e in G.incidentEdges(v) do

u <- G.oppsite(v, e)

if getLabel(u) = UNEXPLORED then //new

d <- getDistance(v) + weight(e)

du <- getDistance(u)

if (d < du) then

setDistance(u, d)

H.replaceKey(getLocator(u), d)

-------------------------------------------------------------------------

//Applying the template method

// --> Just add 1 more method, setParent()

**Algorithm Dijkstra(G)**

**//1. Select a start vertex**

s <- G.aVertex()

H <- new Heap()

**//2. Set other vertices to the infinitive**

for each v in G.vertices() do

setLabel(v, UNEXPLORED) //new

if s = v then

setDistance(v, 0)

//Template method

setParent(v, 0)

else

setDistance(v, infinitive)

l <- H.insert(getDistance(v), v)

H.setLocator(v, l)

**//3. Set distance for endpoints of the start vertex's** **edges by applying the relaxation for other vertices**

a. find the smallest distance

b. update the distance of the related endpoint if its is greater than

while not H.isEmpty() do

v <- H.min()

setLabel(v, VISITED) //new

for each e in G.incidentEdges(v) do

u <- G.oppsite(v, e)

if getLabel(u) = UNEXPLORED then //new

d <- getDistance(v) + weight(e)

du <- getDistance(u)

if (d < du) then

setDistance(u, d)

//Template method

setParent(u, e)

H.replaceKey(getLocator(u), d)

-------------------------------------------------------------------------

**//The same Dijkstra's algorithm, just differ the way of setting weight to vertices**

**// and using the locator to check whether it should be updated distance or not**

**Algorithm Prim(G)**

s <- G.aVertex()

H <- new Heap()

**//2. Set other vertices to the infinitive**

for each v in G.vertices() do

setLabel(v, UNEXPLORED) //new

if s = v then

setDistance(v, 0)

else

setDistance(v, infinitive)

setParent(v, 0)

l <- H.insert(getDistance(v), v)

H.setLocator(v, l)

**//3. Set distance for endpoints of the start vertex's edges by applying the relaxation for other vertices**

a. find the smallest distance

b. update the distance of the related endpoint if its is greater than

while not H.isEmpty() do

v <- H.min()

setLocator(v, 0) //because v was removed from the heap

for each e in G.incidentEdges(v) do

u <- G.oppsite(v, e)

if getLocator(u) <> 0 then //if the vertex 'u' was visited (not in the heap), it's unnecessary to check more

d <- weight(e)

du <- getDistance(u)

if (d < du) then

setDistance(u, d)

setParent(u, e)

H.replaceKey(getLocator(u), d)

-------------------------------------------------------------------------