3. A logarithmic algorithm has a time complexity of O(log n)

4. A quadratic algorithm has a time complexity of O(n^2)

5. A linear algorithm has a time complexity of O(n)

6. A binary tree that has 10,000 internal nodes will have a height between -15-and -(10000+1)-

7. A red-black tree that has 10,000 internal nodes will have a height between -15-and-30—

8. An AVL, tree that has 10,000 internal nodes will have a height between ---15-- and –30-

15. In a hash table implementation of the Dictionary ADT, the insertItem, findElement,

and removeElement operations run in –O(1)---, --O(1)---, and –-O(1)--- expected time

respectively.

16. In an unsorted, growable array implementation of the Dictionary ADT, the insertItem,

findElement, and removeEIement operations run in –-O(1)--, --O(n)--, --O(n)-- and time

respectively.

17. In a sorted, growable array implementation of the Dictionary ADT, the insertItem,

findElement, and removeElement operations run in --O(n)--, -- O(log n)--, and --O(n)--

time respectively

18. In a red-black tree implementation of the Dictionary ADT, the insertItem, findElement,

and removeElement operations run in -- O(log n)---, -- O(log n)--, and --O(log n)-- time

respectively

6. All implementation of an unordered dictionary are necessarily inefficient for finding

items since the entire dictionary might have to be scanned to find the key /FALSE

8. In a Red-Black tree, the restructuring and recoloring operations are sometimes necessary

when searching the tree. /FALSE

C-4.11 Suppose we are given **an n-element** **sequence S** such that each element in S  
represents a different vote in an election, where each vote is given as an integer  
representing the ID of the chosen candidate. Suppose we know who the candidates are  
and the number of candidates **running is k < n.** **Describe an O(n log k)-**time pseudo code  
algorithm for determining who wins the election.

**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

HT                                      BST

**Algorithm countVotes(L)**

1   D := new Dictionary                 1

n   for each id in L.elements() do      n

n       cnt := D.findValue(id)          nlog k

n       if null = cnt then              n

k           D.insertItem(id,1)          klog k

        else

n           D.insertItem(id, cnt + 1)   nlog

**Algorithm findWinner(L)**

    D :=countVotes(L)

    iter :=D.items()

    maxCount := 0

    while iter.hasNext() do

        (id, count) :=iter.nextObject()

        if count > maxCount then

            maxCount := count

            winners := []

            winners.push(id)

        else if count = maxCount then

            winners.push(id)

    return winners

C-4.10 Suppose we are given an **n-element** **sequence S** such that each element **in S**  
represents a different vote in an election, where each vote is given as an integer  
representing the **ID of the chosen candidate**. Without making any assumptions about  
who is running or even how many candidates there are, design an efficient algorithm to  
see who wins the election S represents, assuming the candidate with the most votes  
wins.  Handle the possibility of multiple winners and do this using a **Dictionary.**  

**Algorithm findElectionWinner(S)**

    Input: n-element sequence S where each element represents a different vote

    Output: ID of winning candidate

    mergeSort(S, C)

    winCandidateId <- S.first()

    maxVote <- 0

    prevId <- S.first()

    noOfVote <- 0

    while !S.isEmpty() do

        curId <- S.remove(S.first())

        if curId != prevId then

            if maxVote < noOfVote then

                   maxVote <- noOfVote

                   winCandidateId <- curId

     noOfVote <- 0

        else

                prevId <- curId

                noOfVote <- noOfVote + 1

     return winCandidateId

R-3.10 A certain Professor Amongus claims that a (2,4) tree storing a set of items will

always have the same structure, regardless of the order in which the items are inserted.

Show that Professor Amongus is wrong.

5, 8, 3, 9, 2, 7, 1, 4, 6 ---> 2,5,8; 1; 3,4; 6,7; 9;

\*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\*

**findelements(T, low, high)**

    R: = new List

    findHelper(T, low, high, T.root(), R)

    return R

findHelper(T, low, high,T.root(), R)

IF T.isExternal(p) then

    return R

findHelper(T, low, high, T.leftChild(p), R)

findHelper(T, low, high, T.rightChild(p), R)

if p.element() <= low /\ p.element() >= high

    R.insertLast(p.element())

\*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\*

**findHelper(T, p, low, high, L)**

    if T.isExternal(p) then                       //post-order

        return

    findHelper(T, T.leftChild(p), low, high, L)   //left

    findHelper(T, T.rightChild(p), low, high, L)  //right

    e := p.element()

    if low <= e /\ e <= high                      //parent

        L.insertLast(e)

findElements(T, low, high)

    L := new List

    findHelper(T, T.root(), low, high, L)

    return L

\*/

**class findElements extends EulerTour {**

  #List;

  #low;

  #high;

  constructor() {}

  findElements(T, low, high) {

    this.#List = List;

    this.#low = low;

    this.#high = high;

    eulerTour(T, T.root());

    return this.#List;

  }

  visitInOrder(T, p, result) {

    let e = p.element();

    if (this.#low <= e && e <= this.#high) {

      this.#List.insertLast(e);

    }

    result[1] = this.#List;

  }

  visitExternal(T, p, result) {

    result[1] = this.#List;

  }

}

**Design a pseudo-code algorithm**, **isPermutation(A,B),** that takes two Sequences A and B and determines whether or not they are permutations of each other, i.e., they contain same elements but possibly occurring in a different order. Hint: A and B may contain duplicates. Same problem as in previous homework, but this time use a dictionary to solve the problem.

**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

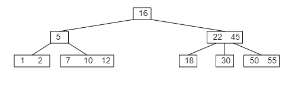
                                                D.removeElement(b)

                return true

R-3.11 Consider the following sequence of keys: (5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 1, 7, 55)

Consider the insertion of items with this set of keys, in the order given, into:

an initially empty (2,4) tree T’.



C-3.**10 Let D be** an ordered dictionary with n items implemented by means of an **AVLtree (or a Red-Black tree).**Show how to implement the following operation **on D in time O(log n + s),** where s is the size of the iterator returned:

**FindAllInRange(k1, k2):**

Return an iterator of all the elements in D with key k such that k1 < k < k2.

**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 calculateheightOfNodes(T)**

    heightHelper(T, T.root())

**Algorithm heightHelper(T,p)**

    if T.isExternal(p) then

        return 0

    leftChild := T.leftChild(p)

    if T.isExternal(leftChild) then

        Lh := 0

    else

        heightHelper(leftChild)

        Lh := getHeight(leftChild)

    rightChild :=T.rightChild(p)

    if T.isExternal(rightChild) then

        Rh := 0

    else

        heightHelper(rightChild)

        Rh := getHeight(rightChild)

    setHeight(p, max(Lh, Rh) + 1)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Algorithm isValidAVL(T)**

    calculateHeightOfNodes(T, T.root())

    return isValidHelper(T, T.root())

**Algorithm isValidHelper(T, p)**

    if T.isExternal() then

        return true

    leftChild := T.leftChild(p)

    if

    rightChild := T.rightChild(p)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Algorithm isValidAVL(T)**

    return isValidHelper(T, T.root())

**Algorithm isValidHelper(T,p)**

    if T.isExternal(p) then

        return 0

    leftChild := T.leftChild(p)

    if T.isExternal(leftChild) then

        Lh := 0

    else

        lValid := heightHelper(leftChild)

        Lh := getHeight(leftChild)

    rightChild :=T.rightChild(p)

    if T.isExternal(rightChild) then

        Rh := 0

    else

        rValid := heightHelper(rightChild)

        Rh := getHeight(rightChild)

    setHeight(p, max(Lh, Rh) + 1)

    valid := (ABS(Lh - Rh) <= 1)

    return lValid /\ rValid /\ valid

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Algorithm isValidAVL(T)**

    return isValidHelper(T, T.root())

**Algorithm isValidHelper(T,p)**

    if T.isExternal(p) then

        return (true,0)

    leftChild := T.leftChild(p)

    (lValid, Lh ):= heightHelper(leftChild)

    rightChild :=T.rightChild(p)

    (rValid, Rh) := heightHelper(rightChild)

    valid := (ABS(Lh - Rh) <= 1)

    return lValid /\ rValid /\ valid

Using the **DFS Template Method** algorithm given in the lecture notes, define a pseudo code algorithm, **connectedComponents(G),** that computes the connected components of a graph G.  Your method should return a sequence of vertices, 1 representative from each connected component.

**Algorithm initResult(G)**

                R<-new Sequence

**Algorithm preComponentVisit(G, v)**

                S<-new Sequence

**Algorithm postComponentVisit(G, v)**

                R.insertLast(S)

**Algorithm startVertexVisit(G, v)**

                S.insertLast(v)

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

                S. insertLast(e)

**Algorithm result(G)**

                return R

**Modify the breadth-first search algorithm so it can be used as a Template Method**

**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**

Algorithm BFS\_TemplateMethod(G, startV)

                initResult(G)

                Iterator verticesIter <- G.vertices()

                while verticesIter.hasNext() do

                                vt <- verticesIter.nextItem()

                                setLabel(vt, UNEXPLORED)

                                preInitVertex(vt)

                Iterator edgesIter <- G.edges()

                while edgesIter.hasNext() do

                                e <- edgesIter.nextItem()

                                setLabel(e, UNEXPLORED)

                                preInitEdge(e)

                Q <- new Queue()

                //v <- G.aVertex()

                setLabel(v, VISITED)

                startVertexVisit(G,v)

                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)

                    setLabel(e, DISCOVERY)

                     setLabel(w, VISITED)

                     Q.enqueue(w)

                     postDiscoveryTraversal(G, v, e, w)

                         else  setLabel(e, BACK) backTraversal(G, v, e, w)

                finishVertexVisit(G, startV) //Template

b. Give a pseudo code algorithm, **findPath(G, u, v) that** finds a path between u and v.  
You will need to override the appropriate methods so that given two vertices u and  
v of **graph G, your BFS finds a path in G between them**, or report that no such path  
exists.  Note that this path will be a path with the minimum number of edges.

**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

**Algorithm findCycle(G)**

    DFS(G)

**Algorithm preDiscoveryVisit(G, v, e, w)**

    setParent(w, e)

**Algorithm backEdgeVisit(G, v, e, w)**

    buildCycle(G, v, e, w)

**Algorithm buildPath(G, dest)**

    path := new List

    par := getParent(dest)

    while par !=null do

        path.insertFirst(dest)

        path.insertFirst(par)

        dest := G.opposite(par, dest)

        par := getParent(dest)

    path.insertFirst(dest)

    return path

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

    for all e in G.vertices() do

        setLabel(v, Unexplored)

        setParent(v, null)

    for all e in G.edges() do

        setLabel(e, Unexplored)

    DFScomponent(G,u)

    return buildPath(G, v)

**Algorithm preDiscoveryVisit(G, v, e, w)**

    setParent(w, e)