Sheet # 1

1. Consider the class Counter shown below. Create an interface for the class. Include comments that specify the methods of the class.

```
/** The interface Counter operates a non-negative integer. */

public interface Counter {

/**   resets the count to 0.    */
   public void reset();

/**   increases the count by 1.    */
   public void increase();

/**   decrease the count by 1,
postcondition: count >= 0 */
   public void decrease();

/**   gets current count value
@return  the non-negative count*/
   public int getCount ();

/**   gets the count as a string
@return  a String form of the count*/
   public String toString();
}
```

2. Consider the interface Circular and the class Circle, as given in Segment D.26.

a. Is the client or the method setRadius responsible for ensuring that the circle's radius is positive?

    The client.

b. Write a precondition and a postcondition for the method setRadius.

    Precondition: newRadius >0
    Postcondition: the radius is changed to newRadius.

c. Write comments for the method setRadius in a style suitable for javadoc.

```
/** sets the radius
@param newRadius     the positive radius to be set
Precondition: newRadius>0
Postcondition: the radius is changed to newRadius if newRadius>0,
otherwise throw Exception (“Radius can’t be negative.”)
*/
```

d. Revise the method setRadius and its precondition and postcondition to change the responsibility mentioned in your answer to Part a.

```
public void setRadius(double newRadius)
   {
   if (newRadius>0)
   radius=newRadius;
   else
   throw new Exception ("Radius can't be negative.");
   }
```

IT212: Data Structures, Sheet # 2

1: Write a method containsAll that takes two bags and returns true if bag1…

```
public boolean containsAll(BagInterface<T> Bag1, BagInterface<T> Bag2){
boolean result=true;
Object[] bag2Array = bag2.toArray();
for (int index=0;index<bag2Array.length;index++)
if(!Bag1.contains(bag2Array[index]))
result=false;
return result; }
```

2:

a. Write Java statements that remove and count all occurrences of "soup" …

```
int count = 0;
while (groceryBag.remove("soup"))
count++;
System.out.print("soup has occurred "+count+" times in groceryBag");
```

b. What effect does the operation groceryBag.toArray() have on groceryBag?

```
It will have no effect on groceryBag itself, it only returns an array of
grocesryBag contents.
```

c. Write some Java statements that create an array of the distinct strings that are in this bag…

```
BagInterface <String> temp = new Bag<String>(groceryBag.getCurrentSize());
String[] products = groceryBag.toArray();
for (int index=0 ; index<products.length ; index++ )
if (!temp.contains(products[index]))
temp.add(products[index]);

products = temp.toArray();
```

3. Write code that accomplishes the following tasks: Consider two bags that can hold strings…

```
int numberOfVowels = 0;
String letter;
while (!letters.isEmpty()) {
letter=letters.remove();

if (letter.equals("a") || letter.equals("e") || letter.equals("i") ||
letter.equals("u" || letter.equals("o"))
{  numberOfVowels++;
vowels.add(letter);  } //end if, checks if the letter is a vowel.
)//end while
System.out.println("the number of vowel in the bag= "numberOfVowels);
String[] vowel = {"a","e","i","u","o"};
for (String aVowel : vowel) //prints the vowels and how many times each
appeared once at a time.
System.out.println("The vowel "+aVowel+" appears
"+vowel.getFrequencyOf(aVowel)+" times");
```

IT212: Data Structures,   Sheet # 3

1. Suppose that a bag contains Comparable objects. Implement method getMin that returns the smallest object in a bag for the class LinkedBag

```
/** Returns the smallest Object in the bag.
       @return the minimum value if the bag is not empty, or null
otherwise */
   public T getMin(){
      if (isEmpty())
         return null;
      Node min=firstNode;
      Node current=firstNode;
      while (current!=null){
         if ((current.data).compareTo(min.data) < 0)
            min=current;
         current=current.next;
      }
      return min.data;
   }
```

for this method to work, both BagInterface and LinkedBag must extend comparable as such:

```
   1- public interface BagInterface<T extends Comparable<? super T>>
   2- public class LinkedBag<T extends Comparable<? super T>> implements
      BagInterface<T>
```

2. Define a method removeEvery that removes all occurrences of a given entry from a bag.

a. for the class ArrayBag

```
/** Removes all occurrences of a specified entry from
the bag, if possible.
      @param anEntry  the entry to be removed */
   public void removeEvery(T anEntry){
      int index = getIndexOf(anEntry);
      while (index!=-1)
      {
         removeEntry(index);
         index = getIndexOf(anEntry);
      }
   }
```

b. as a client method

```
public void removeEvery (T anEntry){

      int occurences = getFrequencyOf(anEntry);
      while (occurences!=0){
         remove(anEntry);
         occurences--;}
   }
```

3. The union of two collections consists of their contents combined into a new collection….

```
      public BagInterface union(BagInterface anotherBag){

            BagInterface<T> unionBag= new ArrayBag<T>(numberOfEntries +
      anotherBag.getCurrentSize());

            @SuppressWarnings("unchecked")
               ArrayBag<T> anotherBag2=(ArrayBag) anotherBag;

         //adding first bag elements:
            for (int index=0; index < numberOfEntries ; index++){
               unionBag.add( bag[index] );
            }
         //adding second bag elemens:
            for (int index=0; index < anotherBag2.numberOfEntries ; index++){
               unionBag.add( anotherBag2.bag[index] );
            }

            return unionBag;
         }
```

4. Consider the definition of LinkedBag's add method that …

5. What is displayed by the following statements in a client of the modified LinkedBag?

```
20 20 20 20
The method allows adding one and only one entry, which is the
last one added.
```

a. What methods, if any, in LinkedBag could be affected by the change to the method add when they execute? Why?

```
Since the method keeps adding the same last entry over and over
again, the bag (and the bag methods) won't be affected, but they
will perform on wrong entries.
```

5. Using Big Oh notation, indicate the time requirement of each of the following tasks in the worst case.

a. Display all the integers in an array of integers.

O(n)

b. Display all the integers in a chain of linked nodes.

O(n)

c. Display the nth integer in an array of integers.

O(1)

d. Compute the sum of the first n even integers in an array of integers.

O(n)

6. Consider the class LinkedBag. If the bag has n nodes, what is the time complexity of adding or removing from the end of the bag? Can this complexity be improved?

```
The program will have to go through nodes one by one (n times),
to add or remove the last entry, which has an order of n: O(n).
That is of course assuming there's no tail variable.
The efficiency can be improved by allocating a tail variable,
the adding and removing will then both have an order of 1: O(1).
```

IT212: Data Structures, Sheet # 4

1. What is the runtime complexity of the following code fragments in big-Oh notation as functions of m and/or n? Explain each answer.

a.

```
int sum1 = 0;                          Assignment,   O(1)
    for (int i = 1; i <= n; i*=2)      Cond.         O(1)   Loop: O(log n) (since i doubles each iteration.
        sum1 += i;                     Addition.     O(1)   All loop= O(log n)+O(1)+O(1)=O(log n)
    int sum2 = 0;                      Assignment.   O(1)
    for (int j = 1; j <= 3*n; j++)     Cond.         O(1)
        sum2 ++;                       Addition.     O(1)   Loop: O(3n) + O(1) = O(n).
                                                              All= O(log n) + O(n) = MAX[O(log n) , O(n)] = O(n)
```

b.

```
    for (int i = n; i>=1; i/=2 ) {     Cond.         O(1)
        System.out.println(i);         Printing.     O(1)   Loop: O(log n) (since i is halved each iteration.
    }
                                                             All= O(log n)
```

c.

```
    int sum = 0;                         Assignment,   O(1)
    for (int i = 1; i <= n; i++)         Cond.         O(1)   Loop: O(n)
        sum +=i;                         Addition.     O(1)   All loop= O(n)+O(1)+O(1)=O(n)
    for (int j = n; j >=1 ; j/=2)        Cond.   O(1)
        for (int k = 1; k <= m; k+=2)    Cond.   O(1)         outer loop: O(log n) (since j is halved each
            sum += (j+k);                Addition. O(1)       inner loop: O(n) + O(1) = O(n).
                                                                  Both loops: O(log n) x O(n) = O(n log n)
     All= O(n log n) + O(n) = MAX[O(n log n) , O(n)] = O(n log n)
```

d.

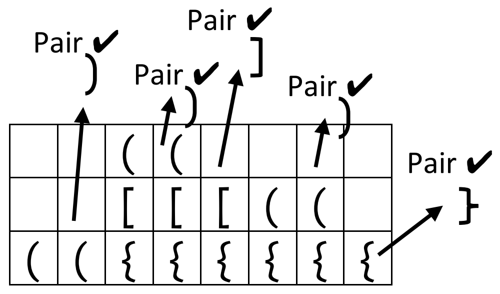
```
    int sum = 0;                              Assignment,   O(1)
    for (int i = 1; i <= 10; i+=2)            Cond.         O(1)   outer loop: O(n)
        for (int j = 1; j <= (n*n) + i; j++)  Cond.         O(1)   inner loop= O(n^2) + O(1) =O(n^2)
            sum += (i+j) ;                    Addition.     O(1)   All loop= O(n)+O(1)+O(1)=O(n)
                                                                   Both loops: O(n^2) x O(n) = O(n^2)
```

e.

```
    int i = n;                          Assignment,      O(1)
    while (i > 0) {                     Cond.            O(1)   while loop: O(log n) (since i is halved each iteration)
        for (int j = 0; j < m; j++)     Cond.            O(1)   outer for loop: O(n)
            for (int k=0; k<m; k++)     Cond.            O(1)   inner for loop: O(n)
              System.out.println("*");  printing.        O(1)   for loops= O(n) x O(n) = O(n^2)
        i = i / 2; }                    Mathematic op.   O(1)   while loop & its body: O(log n) x O(n^2)=

                                                                      All = O(n^2 log n)
```

2. Show the contents of the stack as you trace the algorithm **checkBalance**, given in segment 5.8, for the following expression: **a*(-b) +{[(c-d)/f]*(f-g)}** State whether it will return true or false.

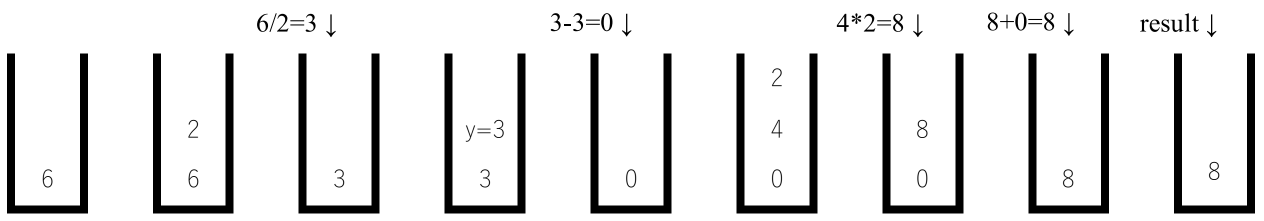


It will return true.

3. Using the algorithm **convertToPostfix**, given in segment 5.16, convert the following infix expression to postfix expression; trace and show the stack: **(a * b) + ((c / d) ^ e)**

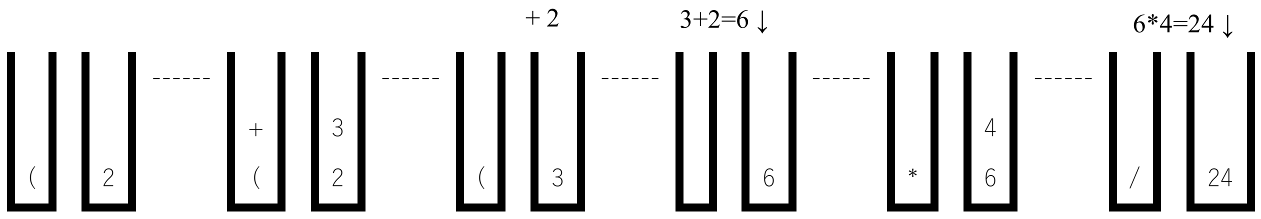
Next character	Postfix	stack
(		(
a	a	(
*	a	(*
b	ab	(*
)	ab*	empty
+	ab*	+
(	ab*	+(
(	ab*	+((
c	ab*c	+((
/	ab*c	+((/
d	ab*cd	+((/
)	ab*cd/	+
^	ab*cd/	+^
e	ab*cd/e	+^
)	ab*cd/e^+	empty

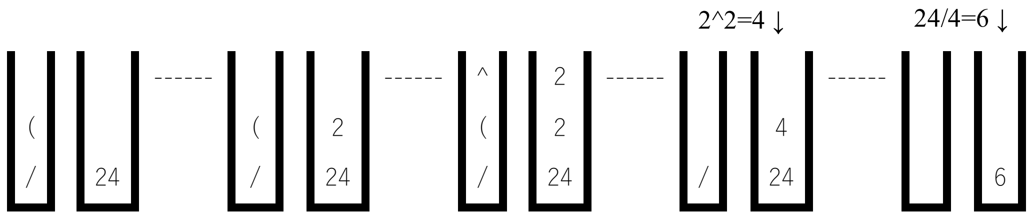
4. Using the algorithm **evaluatePostfix**, given in segment 5.18, evaluate the following postfix expression: 6 x / y - z x * + Assume that x = 2, y = 3, z = 4.



5. Show the contents of the two stacks as you trace the algorithm **evaluateInfix**, given in segment 5.21, for the following infix expression:

(a + b) * c / (a ^ a). Assume that a = 2, b = 3, c = 4.





IT212: Data Structures, Sheet # 5

1. Write method **popBottom()** that removes and returns the bottom of the stack if it has items.

a. Implement this method for each of the following classes:

i. LinkedStack.

```
public T popBottom(){
   T bottom = null;
   if (!isEmpty())
   {
      Node prev=topNode;
      Node last = topNode;
      while (last.next!=null){
         prev=last;
         last = last.next;
      }//end while
      bottom = last.data;
      if (prev==last)
         topNode = last = null;
      else
         prev.next = null;
   } // end if (not empty).
   return bottom;   }
```

ii. ArrayStack.

```
  public T popBottom(){
     T bottom = null;
     if (!isEmpty())
     {bottom = stack[0];
     //shifting
 for (int index=0;index < stack.length-1; index++)
           stack[index]=stack[index+1];
        topIndex--;}
     return bottom; }
```

b. What would be the time efficiency of the methods defined in part a for each of the implementations?

They're both O(n), the actual removing in an ArrayStack is O(1), however, the shifting needs to loop through the whole array so it's O(n).

2. After each statement executes, what is the content of the queue?

`QueueInterface<Integer> myQueue = `**new**` LinkedQueue<Integer>();`

`myQueue.enqueue(8);`

8									

`myQueue.enqueue(9);`

8	9								

`myQueue.enqueue(2);`

8	9	2							

`myQueue.enqueue(myQueue.getFront());`

8	9	2	8						

`myQueue.enqueue(myQueue.dequeue());`

9	2	8	8						

`myQueue.dequeue();`

2	8	8							

`myQueue.dequeue();`

8	8								

`myQueue.enqueue(4);`

8	8	4							

`myQueue.enqueue(myQueue.getFront());`

8	8	4	8						

3. After each statement executes, what is the content of the deque?

```
DequeInterface<String> myDeque = new LinkedDeque<String>();
```


```
myDeque.addToFront(1);
```

1									

```
myDeque.addToBack(4);
```

1	4								

```
myDeque.addToFront(2);
```

2	1	4							

```
myDeque.addToBack(myDeque.removeFront());
```

1	4	2							

```
myDeque.addToFront(1);
```

1	1	4	2						

```
myDeque.addToFront(myDeque.getFront());
```

1	1	1	4	2					

```
myDeque.addToFront(myDeque.getBack());
```

2	1	1	1	4	2				

```
myDeque.removeBack();
```

2	1	1	1	4	2				

```
myDeque.addToBack(5);
```

2	1	1	1	4	2	5			

4. After each statement executes, what is the content of the priority queue? (Assuming smaller numbers represent higher priorities)

```
PriorityQueueInterface<Integer> myPQ = new LinkedPriorityQueue<Integer>();
```


```
myPQ.add(2);
```

2									

```
myPQ.add(4);
```

2	4								

```
myPQ.add(3);
```

2	3	4							

```
myPQ.add(myPQ.remove());
```

2	3	4							

```
myPQ.add(myPQ.peek());
```

2	2	3	4						

```
myPQ.add(1);
```

1	2	2	3	4					

```
Integer i= myPQ.remove();
```

2	2	3	4						

```
myPQ.add(myPQ.peek());
```

2	2	2	3	4					

5. Write a client method reverse that reverses the order of the elements in a queue of strings using a Stack.

```
public void reverse(QueueInterface<String> queue){
       StackInterface<String> stack = new LinkedStack<String>();

       while(!queue.isEmpty()){
          stack.push(queue.dequeue());}

       while(!stack.isEmpty()){
          queue.enqueue(stack.pop());} }
```

IT212: Data Structures,   Sheet # 6

1. For the class ArrayQueue, implement a method length() which returns the number of elements in the queue as an int.

```
public int length(){
   if(!isEmpty())
   return backIndex-frontIndex+1;
   return 0;
   }
```

2. Write the header and implementation of a PrintReversed method for the class ArrayQueue that prints the element of the queue in reversed order from last to first.

```
public void PrintReversed(){
   int index = backIndex;
   while(index!= frontIndex-1){
   System.out.println(queue[index]);
   index = (index-1) % queue.length;}}
```

3. If myList is an empty list of strings, what does it contain after the following statements execute?

myList.add("beta");

1. beta

myList.add("gamma");

1. beta
2. gamma

myList.add(1, "alpha");

1. alpha
2. beta
3. gamma

myList.replace(3, "delta");

1. alpha
2. beta
3. delta

myList.add(3, "delta");

1. alpha
2. beta
3. delta
4. delta

myList.remove(2);

1. alpha
2. delta
3. delta

myList.add(3, "beta");

1. alpha
2. delta
3. beta
4. delta

myList.remove(2);

1. alpha
2. beta
3. delta

4. Write the specification and implementation of a client level method that returns the position of a given String in the list myList. Assume the list is not empty.

```
/* Gets the position of a string in a ListInterface,
@return the position, or -1 if string not found
@param   myList     the list to be searched for the string
@param   specific    the string the method will search for */

   public static int getPos(ListInterface<String> myList, String
specific){
      Object[] listArray = myList.toArray();
      int position = -1;
      for (int index = 0; index<listArray.length; index++){
         if(listArray[index].equals(specific)){
            position = index+1;
            break;
         }
      }
      return position;
   }
```

1. Add a constructor to each of the classes AList and LList that creates a list from a given array of objects.

    a. *Arraylist:*

```
 public AList(T[] entries)
{   numberOfEntries = 0;
    @SuppressWarnings("unchecked")
    T[] tempList = (T[])new Object[DEFAULT_INITIAL_CAPACITY];
    list = tempList;
    for (int index = 0 ; index < entries.length ; index++)
    add(entries[index]);}
```

    b. *LinkedList(with tail reference):*

```
public LList(T[] entries){
         clear();
         for (int index = 0 ; index < entries.length ; index++)
                add(entries[index]);}
```

2. Suppose that you want an operation for the ADT list that moves the first item in the list to the end of the list. The header of the method could be as follows: `public void moveToEnd()`

    a. *Write an implementation of this method for class AList:*

```
public void moveToEnd(){
T first=list[0];
for(int index=0;index<numberOfEntries-1;index++){
list[index]=list[index+1];}
list[numberOfEntries-1]=first;}
```

    b. *Write an implementation of this method for class LList:*

```
public void moveToEnd(){
Node lastNode = getNodeAt(numberOfEntries);
lastNode.next(firstNode);
firstNode=firstNode.next;}
```

    c. *Write an implementation of this method at the client level.*

```
public static void moveToEnd(ListInterface list){
Object first = list.remove(1);
list.add(first);}
```

    d. *Compare the time complexity required for all three methods above.*

a- *It's O(n) because entries need to be shifted.*
b- *It's O(n) as well because it calls getNodeAt which is O(n), but if there's a tail reference it will be O(1)*
c- *It's O(n) regardless of the implementation because removing an item given a specific position is O(n) in all implemenatations, in case it's an array shifting must be done, if it's a chain of nodes getting a refernce to that position requires n iterations.*

3. The mode of a list of values is the value having the greatest frequency.

   a. *Write a method to find the mode of a sorted list using only methods of the ADT sorted list:*

```
public int mode (){
   int mode=0, frequency=0;
   Object entry;
   Object[] entries = toArray();
   for(int index=0;index<numberOfEntries;index++){
      entry = entries[index];
      for (int inner = 0; inner<numberOfEntries ; inner++)
         if(entries[inner].equals(entry)) frequency++;
      if (mode<frequency)
         mode=frequency;
      frequency=0;}
   return mode;
}
```

   b. *What is the Big Oh of your method.*

   O(n2)

4. Consider the implementation of the sorted list that uses an instance of the ADT list. In particular, consider the method contains. One implementation of contains could invoke getPosition. Another implementation could simply invoke list.contains.

   a. *Implement method contains by invoking getPosition:*

```
public boolean contains(T anEntry){
      return getPosition(anEntry) > 0; }
```

   b. *Implement method contains by invoking list.contains.*

```
public boolean contains(T anEntry){
      return list.contains(anEntry); }
```

   c. *Compare the efficiencies of these two implementations*

   Both are O(n), however the first implementation may exit the method earlier; hence, saving more time.

5. Suppose that nameList is a sorted list of names. Using the operations of the ADT list and the ADT sorted list, create a list of these names in the reverse order.

```
String[] list= nameList.toArray();

ListInterface<String> myReversedList = new LList<String>();

for(int index=list.length-1 ; index>=0 ; index--)
myReversedList.add(list[index]);
```

1. Trace the method binarySearch, when searching for 23 in the following array of values: 1, 3, 5, 7, 8, 12, 20, 25, 30.

call	header binarySearch(first, last, entry)	mid	comparison	found =	return
1	(0,8,23)	8/2=4	23>8	binarySearch(5,8,23)	← false ↑
2	(5,8,23)	5+(8-5)/2=6	23>20	binarySearch(7,8,23)	← false ↑
3	(7,8,23)	7+(8-7)/2=7	23<25	binarySearch(7,6,23)	← false ↑
4	(7,6,23)	//	first > last	false	false ↑

2. Repeat the trace when searching for 7.

call	header binarySearch(first, last, entry)	mid	comparison	found =	return
1	(0,8,7)	8/2=4	7<8	binarySearch(0,3,7)	← true ↑
2	(0,3,67)	0+(3-0)/2=1	7>3	binarySearch(2,3,7)	← true ↑
3	(2,3,7)	2+(3-2)/2=2	7>5	binarySearch(3,3,7)	← true ↑
4	(3,3,7)	3	7 == 7	true	true ↑

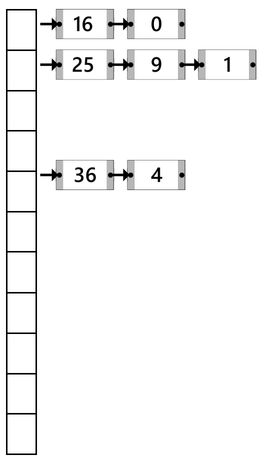
3. Given the following input that consists of key values to be inserted into the hash table, {0, 1, 4, 9, 16, 25, 36} and a hash function: h(x) = x % 8, Show the resulting:

   a. *Separate chaining hash table*
   h(0)=0 ; h(1)=1 ; h(4)=4 ; h(9)=1 ; h(16)=0 ; (h25)=1 ; h(36)=4

   b. *Open addressing hash table using linear probing.*
   h(0)=0 → index 0✔
   h(1)=1 → index 1✔
   h(4)=4 → index 4✔
   h(9)=1 → index 1 is occupied! Go to 1+1, empty? Index 2 ✔
   h(16)= 0→ index 0 is occupied! Go to 0+1, Occupied! 0+2, Occupied! 0+3, empty? Index 3 ✔
   h(25)=1→ index 1 is occupied! Go to 1+1, Occupied! 1+2, Occupied! 1+3, Occupied! 1+4, empty? Index 5 ✔
   h(36)=4 → index 4 is occupied! Go to 4+1, Occupied! 4+2, empty? Index 6 ✔

   c. *Open addressing hash table using double hashing with second function*
   $h_2(x) = 5 - (x \% 5)$.
   h(0)=0 → index 0✔
   h(1)=1 → index 1✔
   h(4)=4 → index 4✔
   h(9)=1 → index 1 is occupied! h2(9)=1, 1+1 empty? Index 2 ✔
   h(16)=0 → index 0 is occupied! h2(16)=4, 0+4, occupied! 4+4, empty? Index 8 ✔
   h(25)=1 → index 1 is occupied! h2(25)=5, 1+5, empty? Index 6 ✔
   h(36)=4 → index 4 is occupied! h2(36)=4, 4+4, occupied! 8+4%11 = 1, occupied! 1+4, empty? Index 5 ✔

A



B

0	O
1	O
9	O
16	O
4	O
25	O
36	O
	E
	E
	E
	E

C

0	O
1	O
9	O
	E
4	O
36	O
25	O
	E
16	O
	E
	E

4. Given the hash table you constructed in part (b) of the previous exercise, do the following in sequence:

a. *Explain the process of deleting the object with key value 16.*
h(16) = 0,
index 0 → O and !=16
index 1 → O and !=16
index 2 → O and !=16
index 3 → O and ==16 , remove 16 and change the status to Available

b. *Explain the process of searching for an object with key value 24 (after deleting 16)*
h(24) = 0
index 0 → O and !=24
index 1 → O and !=24
index 2 → O and !=24
index 3 → A and !=24
index 4 → O and !=24
index 5→ O and !=24
index 6 → O and !=24
index 7 → E , search stops

c. *Explain the process of adding an object with key value 17:*

h(17) = 1, search starts at index 1.

i. *When duplicates are allowed.*
ii. Searching stops at index (h(17)) 1.
iii. *When duplicates are not allowed.*
iv. Searching if there's another 17 stops at first empty location (7)

d. *Do we need to rehash the hash table? Explain*

Yes, I assumed the table size is 11, if it was bigger than that then we wouldn't need to rehash but it's not, so we compute the load factor:
λ = existing entries / size of the table, = 8 / 11 = 0.7, which is greater than 0.5

e. *If your answer in (d) is yes, show the result of rehashing.*
New size = 23. (I'm assuming we already deleted 16)
h(0)=0 → index 0✔
h(1)=1 → index 1✔
h(4)=4 → index 4✔
h(9)=1 → index 1 is occupied! Go to 1+1, empty? Index 2 ✔
h(25)=1→ index 1 is occupied! Go to 1+1, Occupied! 1+2, available! Index 3 ✔
h(36)=4 → index 4 is occupied! Go to 4+1, empty? Index 5 ✔

0	O
1	O
9	O
25	O
4	O
36	O
	E
	E
	E
	E
	E
	E
	E
	E
	E
	E
	E
	E
	E
	E
	E
	E

1. Write a client method **getLowInventoryItems** that returns a list of items with quantity less than 5 and remove those items from **inventoryList**.

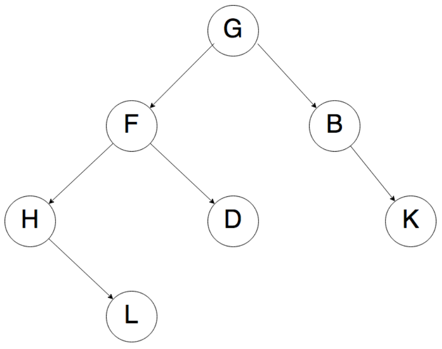
```
public ListInterface<Item> getLowInventoryItems (ListInterface inventoryList){

ListInterface<Item> lowInventoryList = new List<Item>();
Iterator<Item> inventoryIterator = inventoryList.getIterator();
Item temp;
temp = inventoryIterator.Next();

while (inventoryIterator.hasNext()) {
  if(temp.getQuantity()<5){
    lowInventoryList.add(temp);
    inventoryIterator.remove();}
  temp = inventoryIterator.Next();}

return lowInventoryList;}
```

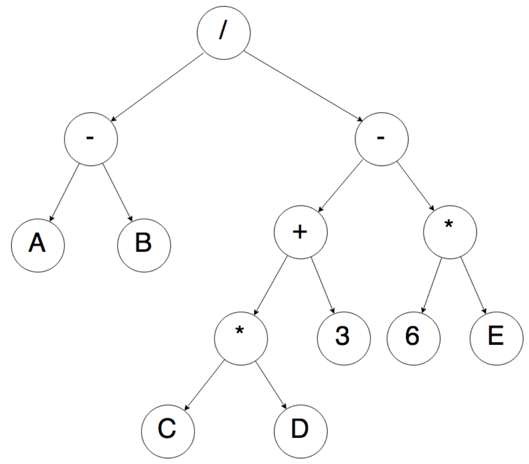
2. Draw the unique binary tree that has the following preorder and inorder traversals: Preorder: GFHLDBK - Inorder: HLFDGBK



3. Traverse the following binary tree using the three types of tree traversal

a- Preorder() ABIDGCEHJFK
b- Inorder() IBGDAEJHCKF
c- Postorder() IGDBJHEKFC

4. Draw an expression tree for (A-B)/((C*D+3) - (6*E))

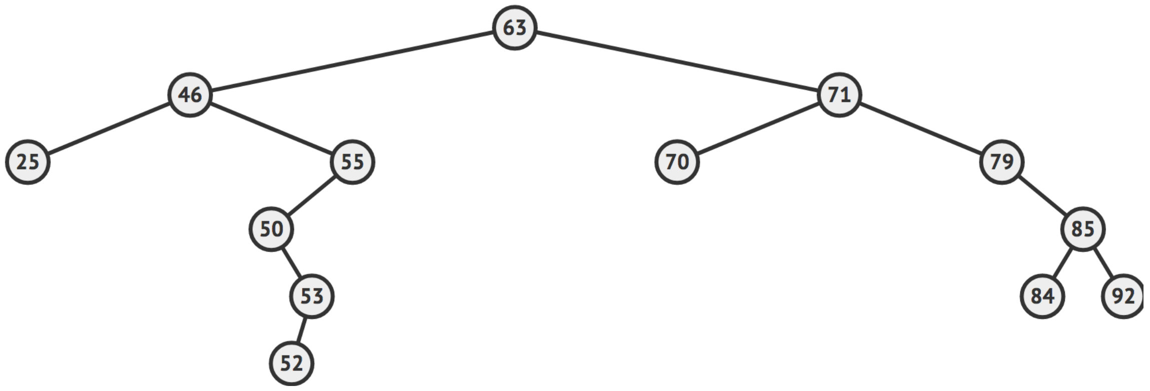


5. For the class BinaryTree, write the implementation of a method that counts the number of time s an object occurs in the tree without using an iterator.

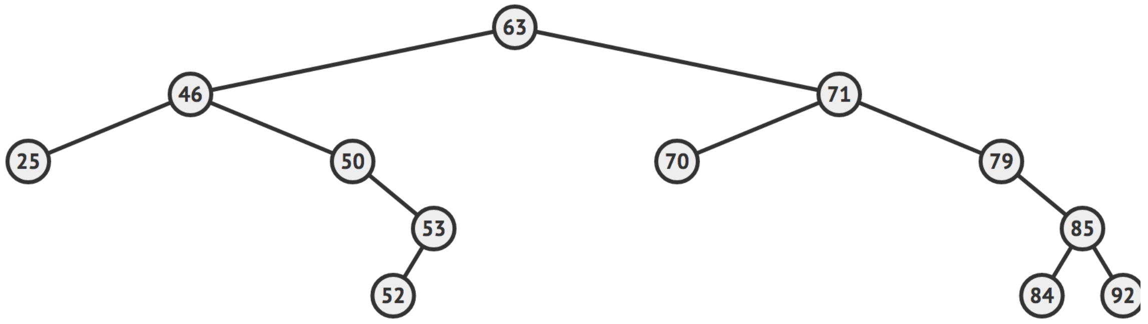
```
public int frequencyOf(BinaryNode rootX,Object obj)
{
  if (rootX==null)
      return 0;

  if (rootX.data.equals(obj))
      return (1 + frequencyOf(root.left, obj) + frequencyOf(root.right, obj));
  else return (frequencyOf(root.left, obj) + frequencyOf(root.right, obj));
}
```

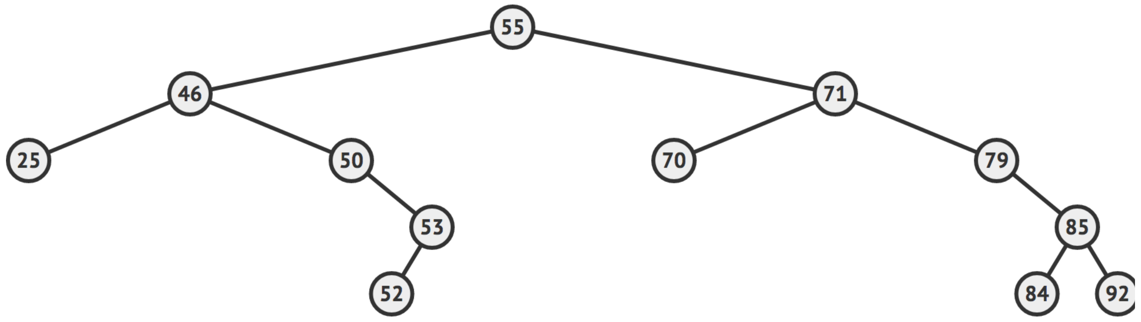
1. Show the results of adding the following search keys to an initially empty binary search tree: 63, 46, 71, 25, 55, 70, 50, 79, 85, 84, 53, 92, 52



- **a.** Is the resulting tree a complete binary tree? Is it full?
  It's neither.
- **b.** What is the worst case time complexity of searching this tree?
  Tree height, 6
- c. Show what this tree would look like after deleting 55

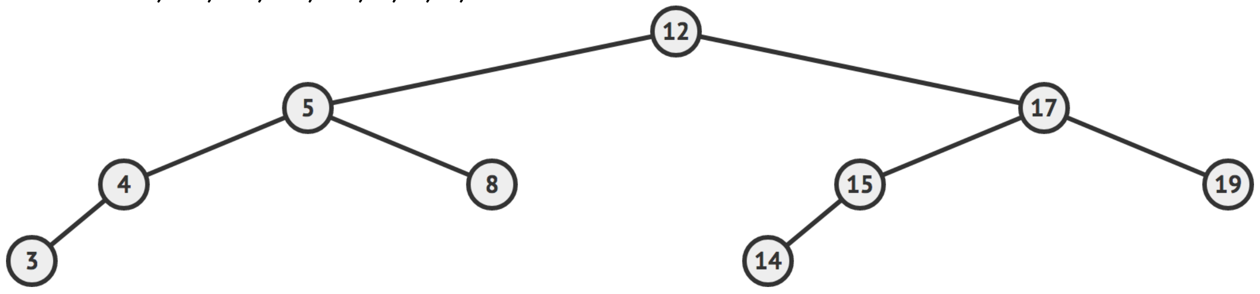


- d. Show what this tree would look like after deleting the root
  Note: Use the original tree every time you want to delete.



2. Give four different orderings of the search keys 9, 17, 5, 12, 4, 15, 8, 14, 19, 3 that would result in the least balanced tree:
    1- 3,4,5,8,9,12,14,15,17,19
    2- 19,17,15,14,12,9,8,5,4,3
    3- 3,4,5,8,9,19,12,14,15,17
    4- 17,15,14,12,19,9,8,5,4,3

3. What ordering of the search keys 9, 17, 5, 12, 4, 15, 8, 14, 19, 3 would result in the most balanced tree if they were added to an initially empty binary search tree?
    -12,17,15,19,14,5,4,3,8



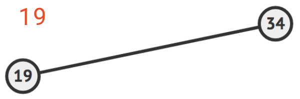
4. Write a client method that accepts as its argument a BinaryTree object and returns true if the argument tree is a binary search tree. Examine each node in the given tree only once (hint: use tree iterator). Extra question to be solved in the tutorial: Think about other solution without using iterator

```
public boolean isBinarySearchTree (BinaryTree t){
Iterator i = t.getInorderIterator();
BinaryNode n1 = i.next();
BinaryNode n2 = i.next();
while(i.hasNext() && n2!=null)
  if (n1.data.compareTo(n2.data)  <=0 ){
    n1 = n2;
    n2=i.next();}
  else return false;}
return true;
}
```

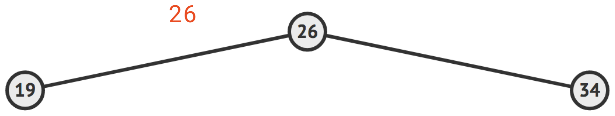
Section: 27807 Mon 10-12. *IT212: Data Structures, Sheet #11*

1. Show the AVL tree after each insertion. 34,19,26,35,39,29,17,11,24,10

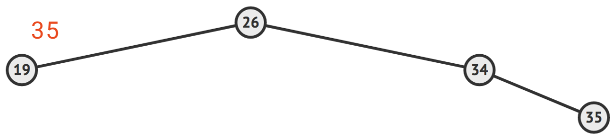
34, 19



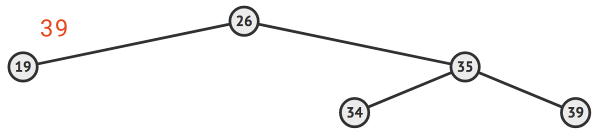
26



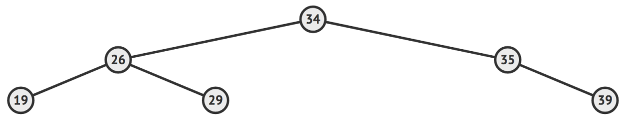
35



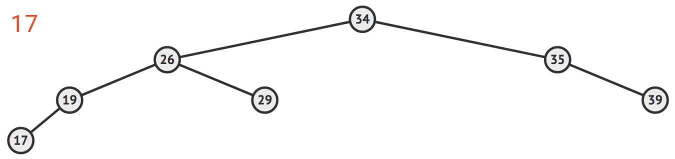
39



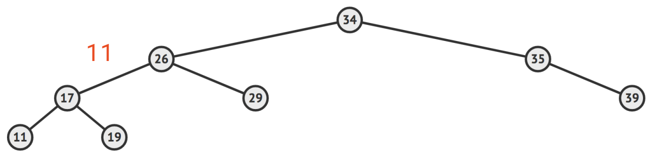
29



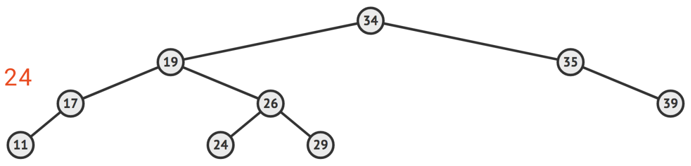
17



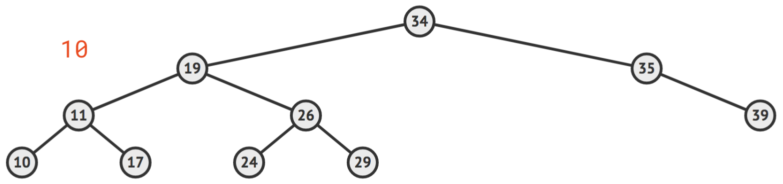
11



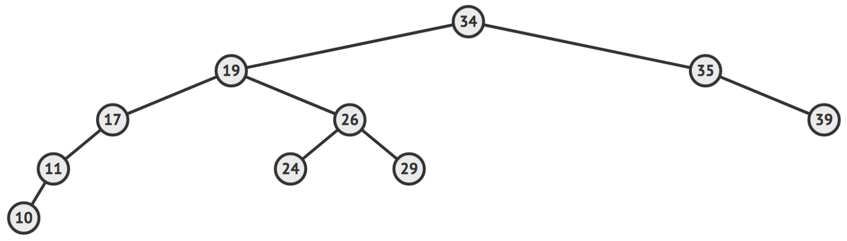
24



10

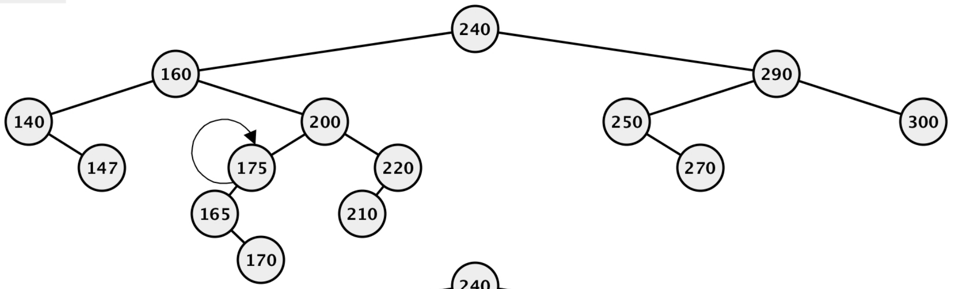


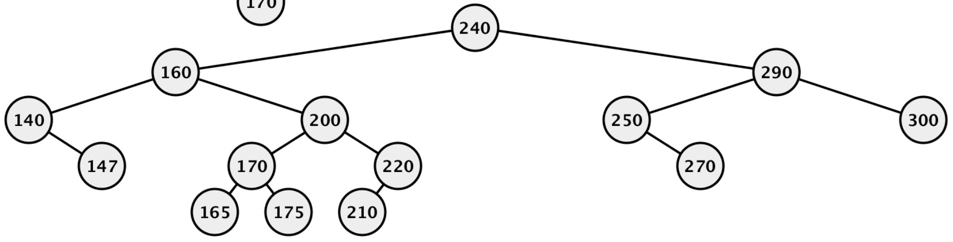
2. Using the previous keys in question 1. Draw binary search tree.



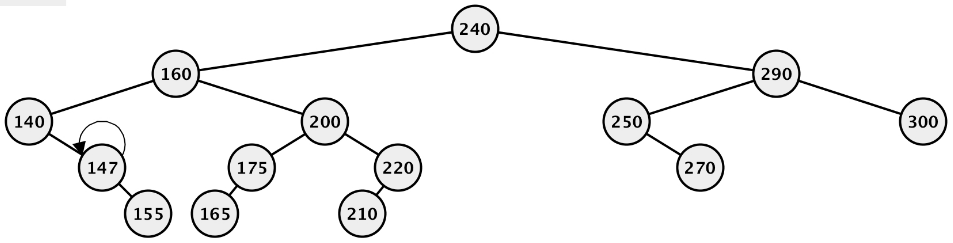
3. Consider the AVL.. Add the keys with respect to the original tree

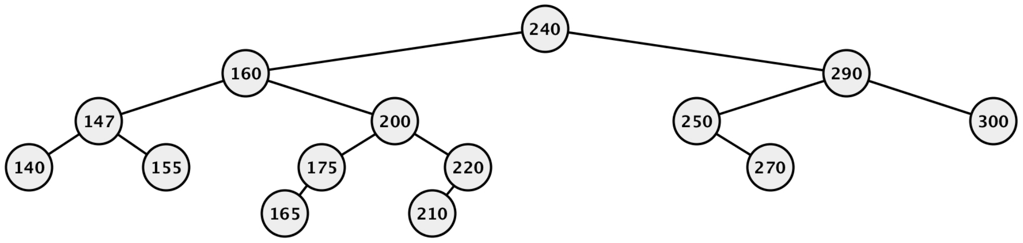
Add 170



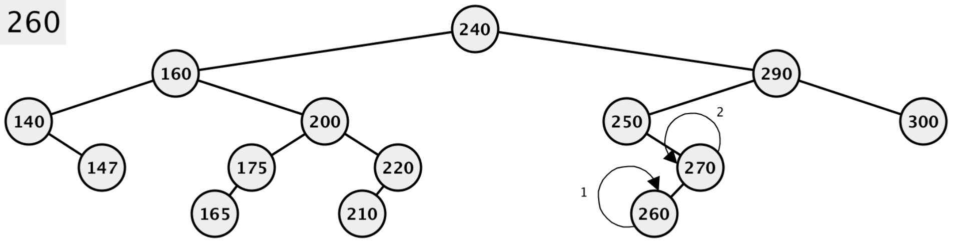


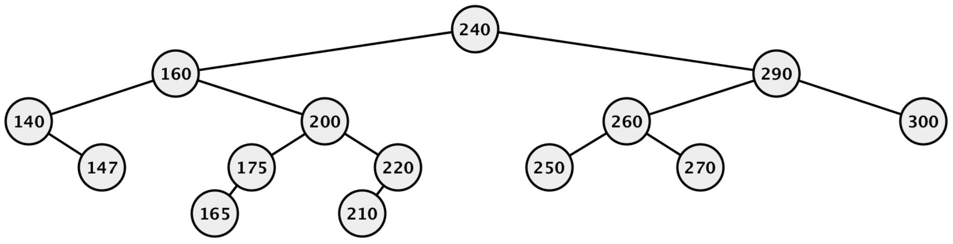
Add 155





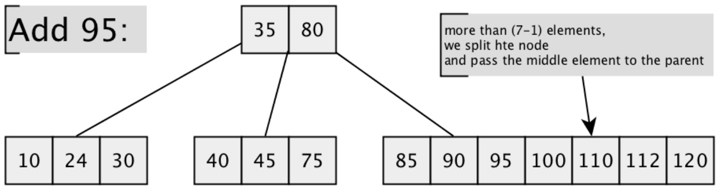
Add 260



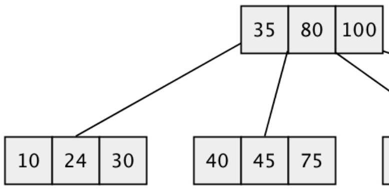


4. The following B - Tr... draw the resulting tree after each operation

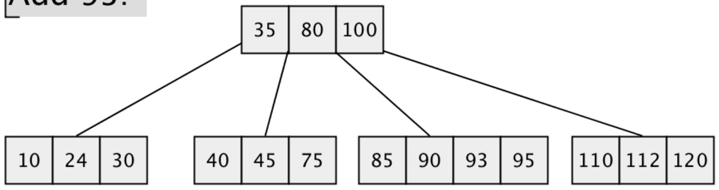
Add 95:



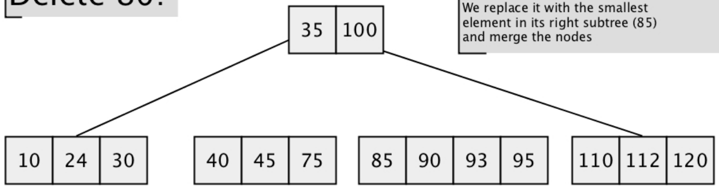
more than (7-1) elements,
we split hte node
and pass the middle element to the parent



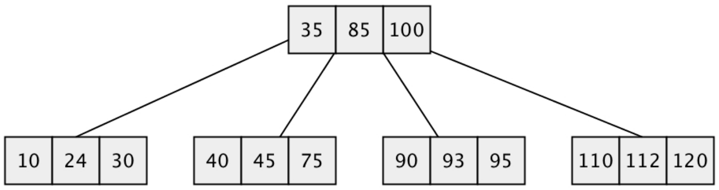
Add 93:



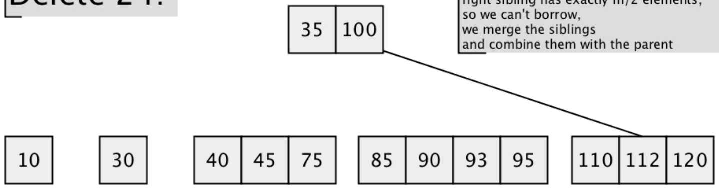
Delete 80:



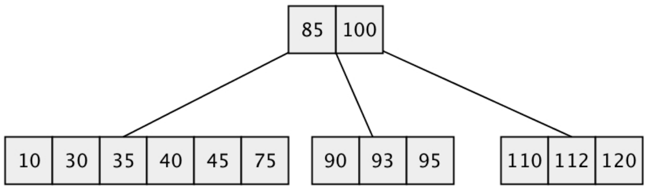
We replace it with the smallest
element in its right subtree (85)
and merge the nodes



Delete 24:

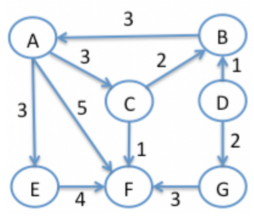


right sibling has exactly m/2 elements,
so we can't borrow,
we merge the siblings
and combine them with the parent

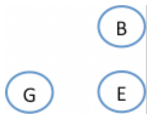


1. Consider the graph shown below:

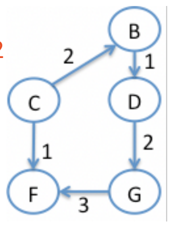
G



G1



G2



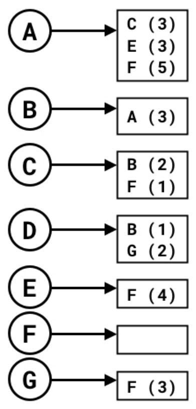
A: State whether each of the following statements is true or false:

1. C is adjacent to F. FALSE
2. The graph (G) is cyclic. TRUE
3. The graph (G) is connected. FALSE
4. The graph (G) is complete. FALSE
5. The graph G1 is a subgraph of G. TRUE
6. The graph G2 is a subgraph of G. FALSE (The edge B to D never existed in G, but D to B did and it's not in G2)

b. Find the adjacency matrix and the adjacency lists of the graph (G).

Matrix: **List:**

	A	B	C	D	E	F	G
A	0	0	3	0	3	5	0
B	3	0	0	0	0	0	0
C	0	2	0	0	0	1	0
D	0	1	0	0	0	0	2
E	0	0	0	0	0	4	0
F	0	0	0	0	0	0	0
G	0	0	0	0	0	0	3



c. Starting with node D, in what order does a depth first traversal visit the vertices?

DBACFEG

d. Starting with node C, in what order does a breadth first traversal visit the vertices?

CBFAE

e. Trace the traversal in the algorithm to find the cheapest path from vertex B to vertex F.

c: **DBACFEG, B→A→C→F costs 3+3+1 = 7**