

Problem Set 7

Due: Friday, November 15

Exam 2 Notice:

In class on Friday, November 15, the second take-home exam will be handed out. It will be due at 11:59pm on Monday, November 25. **This is a hard deadline. No extensions will be given after this time.** The exam will cover the material in lecture through Lecture 19 (Tue. Nov. 12) and the material in problem sets through PS7, including immutable and mutable lists, binary trees, and binary search trees; doubly-linked lists; “hand-wavy” performance comparisons of data structures; and use and implementation of standard data structures, including stacks, queues, priority queues, sets, bags, sequences, and tables. Because you should focus on the exam, it is strongly recommended that you submit PS7 on time (11:59pm on Friday November 15).

Overview: In this problem set, you will get experience with using and implementing sets and bags.

Download: To begin this assignment, you should download a copy of the directory `~cs230/download/ps7`.

Submission:

- For Problem 1, your hardcopy submission should be your final version of `Frequency.java`.
- For Problem 2, your hardcopy submission should include:
 - your final version of `BagDLLSortedFrontBack.java`;
 - your testing transcripts showing that this work as expected;
 - your list of the running times requested in part (b).
- For Problem 3, your hardcopy submission should include:
 - your final version of `BagMBSTEntries.java`;
 - your testing transcripts showing that this work as expected;
 - your list of the running times requested in part (b).

Your softcopy submission for this problem should be your entire `ps7` directory.

Remember to include a signed cover sheet (found at the end of this problem set description) at the beginning of your hardcopy submission.

Problem 1 [30]: Frequency Revisited

In this problem, we revisit the problem of determining the frequency with which words appear in a file, which we first considered in Problem 3 of PS2. This time around, the problem is simplified by our ability to use powerful abstract data types like sets, bags, and priority queues.

You should begin this problem by creating from scratch a new public class named `Frequency` that will be used as a repository for many of the static methods you define in this problem.

a. [10]: `byAlpha`

In your `Frequency` class, implement the following method:

```
public static void byAlpha (String filename);
```

Prints the lower case version of each distinct word appearing in the file named `filename` one per line, along with the frequency with which it appears in the file. The words should be listed in alphabetical order.

In your implementation, you should use the standard abstract data types we have been studying (e.g., sets, bags, priority queues) to simplify your implementation.

You should also implement a `main` method in `Frequency` that allows `byAlpha` to be tested on a file named `filename` by invoking `java Frequency byAlpha filename`.

For example, Fig. 1 shows the result of executing `byAlpha` on the file `initial.txt`, which contains an initial segment of Dr. Seuss's timeless classic *Green Eggs and Ham*, and Fig. 2 shows the result on the entire poem, which is in the file `green.txt`.

Notes:

- Don't forget to import `java.util.Enumeration`.
- Use the `FileWords` class to extract words from a file.
- Be sure to convert all words to lower case. Which `String` method should you use for this?
- Think carefully about where you need to use *ordered* sets and bags, and where *unordered* ones will do.
- For this problem, the following are the "default" implementations of sets, bags, and priority queues that you should use:

```
BagVectorSorted  
MinPQVector  
MaxPQVector  
OrderedBagVectorSorted  
OrderedSetVectorSorted  
SetVectorSorted
```

Note that you do not have to use all of these, only some of them. Indeed, some solutions only need to use one such structure.

- In general, if you first make an enumeration of a collection and then delete elements from the collection, the effect of the deletions on the enumeration are unpredictable. For instance, consider the following:

```
OrderedSet oset = new OrderedSetVectorSorted();  
oset.insert("a");
```

```
$ java Frequency byAlpha initial.txt % $
Frequency of words in initial.txt, ordered alphabetically:
am:3
and:2
do:4
eggs:2
green:2
ham:2
i:6
like:4
not:3
sam:3
sam-i-am:4
that:3
them:1
you:1
```

Figure 1: The result of the invocation `java Frequency byAlpha initial.txt`

```
oset.insert("b");
Enumeration e = oset.elements();
oset.delete("a");
EnumTest.test(e);
```

When the elements of `e` are enumerated by `EnumTest`, will "a" be the first elements not? This question cannot be answered reliably given our specification of collections. One correct implementation of `OrderedSetVectorSorted()` might enumerate both "a" and "b", while another correct implementation might enumerate only "b". The lesson is that you should never depend on the behavior of enumerations for a collection if you delete from the collection after making the enumeration.

In the above example, to guarantee that both "a" and "b" are enumerated, you could make an enumeration of a *clone* of `oset`, where no other operations are performed on the clone.

```
$ java Frequency byAlpha green.txt % $
Frequency of words in green.txt, ordered alphabetically:
a:59
am:3
and:25
anywhere:8
are:2
be:4
boat:3
box:7
car:7
could:14
dark:7
do:37
eat:25
eggs:11
fox:7
goat:4
good:2
green:11
ham:11
here:11
house:8
i:72
if:1
in:40
let:4
like:44
may:4
me:4
mouse:8
not:84
on:7
or:8
rain:4
sam:6
sam-i-am:13
say:5
see:4
so:5
thank:2
that:3
the:11
them:61
there:9
they:2
train:9
tree:6
try:4
will:21
with:19
would:26
you:34
```

Figure 2: The result of the invocation `java Frequency byAlpha green.txt`

b. [20]: byFreq

In your `Frequency` class, implement the following method:

```
public static void byFreq (String filename);
```

Prints the lower case version of each distinct word appearing in the file named `filename` one per line, along with the frequency with which it appears in the file. The words should be listed in order from high frequency words to low frequency words; words with the same frequency should be listed in alphabetical order.

In your implementation, you should use the standard abstract data types we have been studying (e.g., sets, bags, priority queues) to simplify your implementation.

You should also extend the `main` method in `Frequency` to allow `byFreq` to be tested on a file named `filename` by invoking `java Frequency byFreq filename`.

For example, Fig. 3 shows the result of executing `byFreq` on `initial.txt` and Fig. 4 shows the result on the file `green.txt`.

Notes:

- All the notes from part (a) hold here as well.
- You have been provided with the following `WordEntry` class, which you may find useful:

```
public class WordEntry {  
  
    public String word;  
    public int freq;  
  
    public WordEntry (String w, int f) {  
        word = w;  
        freq = f;  
    }  
  
}
```

If you use this class, you will probably also want to define a comparator class that compares instances of `WordEntry`. The details of such a class are left up to you.

```
$ java Frequency byFreq initial.txt % $  
Frequency of words in initial.txt, ordered by frequency:  
i: 6  
do: 4  
like: 4  
sam-i-am: 4  
am: 3  
not: 3  
sam: 3  
that: 3  
and: 2  
eggs: 2  
green: 2  
ham: 2  
them: 1  
you: 1
```

Figure 3: The result of the invocation `java Frequency byFreq initial.txt`

```
$ java Frequency byFreq green.txt % $
Frequency of words in green.txt, ordered by frequency:
not: 84
i: 72
them: 61
a: 59
like: 44
in: 40
do: 37
you: 34
would: 26
and: 25
eat: 25
will: 21
with: 19
could: 14
sam-i-am: 13
eggs: 11
green: 11
ham: 11
here: 11
the: 11
there: 9
train: 9
anywhere: 8
house: 8
mouse: 8
or: 8
box: 7
car: 7
dark: 7
fox: 7
on: 7
sam: 6
tree: 6
say: 5
so: 5
be: 4
goat: 4
let: 4
may: 4
me: 4
rain: 4
see: 4
try: 4
am: 3
boat: 3
that: 3
are: 2
good: 2
thank: 2
they: 2
if: 1
```

Figure 4: The result of the invocation `java Frequency byFreq green.txt`

Problem 2 [30]: Doubly-linked List Implementation of Bags

In this problem, you are to implement a class `BagDLLSortedFrontBack` that represents bags as doubly-linked lists of sorted elements. It turns out to be helpful to maintain pointers to both the first and last nodes of the doubly-linked list, much as in the front/back implementation of a queue as a mutable list. To improve the running time of `size()` and `count()`, the values to be returned by these methods should be cached in instance variables.

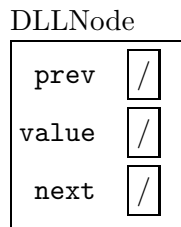
So instances of `BagDLLSortedFrontBack` should have the following instance variables:

- `comp`: a comparator for determining the order of elements.
- `front`: the first node of a doubly-linked list of elements sorted from low to high by `comp`.
- `back`: the last node of the doubly-linked list of elements.
- `size`: the number of elements currently in the bag (includes duplicates).
- `count`: the number of *distinct* elements currently in the bag (does not include duplicates).

Doubly-linked lists are composed out of instances of the following `DLLNode` class:

```
public class DLLNode {  
  
    public DLLNode prev, next;  
    public Object value;  
  
}
```

Note that this class does not have an explicit constructor method. But it still has a default constructor method. Invoking `new DLLNode()` creates a instance in which all three instance variables are null (indicated by a box filled with a “/”).



For example, Fig. 5 shows the `BagDLLSortedFrontBack` representation of a bag that has two As, three Bs and one C.

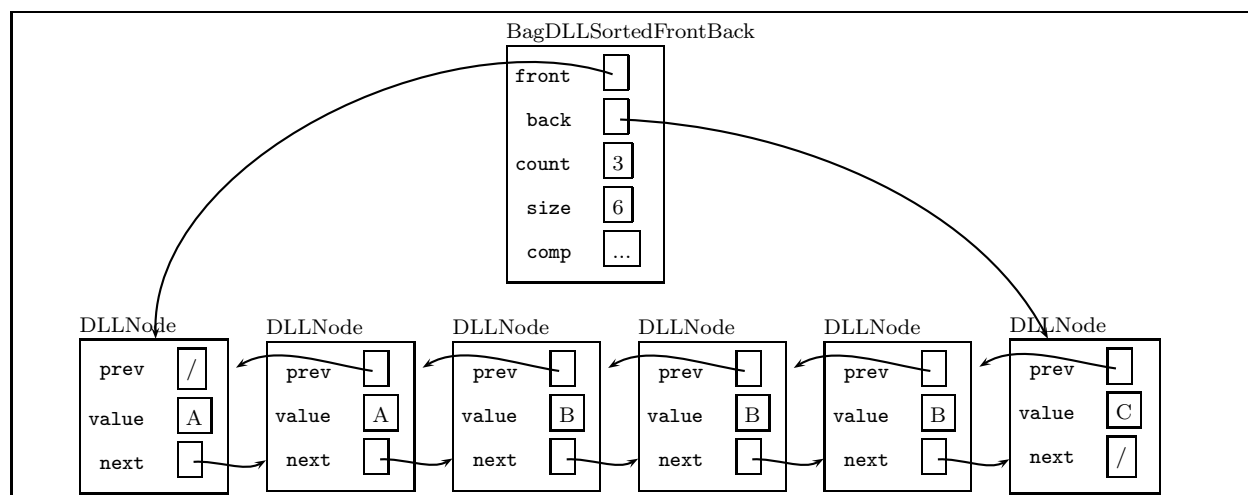


Figure 5: An example of a BagDLLSortedFrontBack.

a. [25]: Implementation

Flesh out the bag implementation in `BagDLLSortedFrontBack.java` using the representation described above. Test your implementation by executing `java BagDLLSortedFrontBack`, and turn in the transcript of this test for your final version of the code as part of your hardcopy submission.

b. [5]: Running Times

Give the worst-case running time of the following operations on a bag with n elements for your implementation of `BagDLLSortedFrontBack`. Use “Theta notation”, e.g. $\Theta(1)$, $\Theta(\log(n))$, $\Theta(n)$, $\Theta(n \cdot \log(n))$, $\Theta(n^2)$.

```

public boolean isMember (Object x);
public void insert(Object x);
public Object choose();
public Object deleteOne();
public boolean delete (Object x);
public boolean deleteAll (Object elt);
public void clear();
public ObjectList toList();
public Object clone ();
public int size();
public int count ();
public int occurrences (Object elt);

```

Notes:

- Since the instance variables of `DLLNode` are public, you can access them directly via the usual “dot” syntax. E.g., if `node` is a instance of `DLLNode`, use `node.value` to extract the value, `node.prev` to extract the “previous pointer”, and `node.next` to extract the “next pointer”. To test if a pointer is the null pointer, use `== null`; e.g., `node.left == null`.
- `ObjectList` operations are accessible via the `OL.` prefix.

- You are encouraged to define private auxiliary methods that will help you implement the required methods. One particularly useful auxiliary instance method is the following:

```
private DLLNode find (Object x, DLLNode L);
```

Assume that L is a sorted doubly-linked list. If x appears at or to the right of node L, returns the first node encountered in a left-to-right linear search of the list whose head is x. If x does not appear at or to the right of node L, returns the first node encountered in a left-to-right linear search of the list whose head is greater than x (according to the bag comparator), if it exists. If there is no node at or to the right of L whose head is greater than x, returns null.

The above method is a useful utility for all the other methods that require searching through the doubly-linked list. It is a good idea to abstract the search process into a single find method rather than writing specialized versions of it several times in different methods.

- The empty bag must be represented specially as an instance of BagDLLSortedFrontBack whose front and back instance variables are both null. (Compare this to the implementation of QueueFrontBack.)

Problem 3 [40]: Mutable BST of Bag Entries Implementation of Bags

In this problem, you are to implement a class BagMBSTBagEntries that represents bags as a mutable binary search tree of entries pairing elements with their number of occurrences. Each entry should be an instance of the following BagEntry class:

```
public class BagEntry {

    public Object elt;
    public int num;

    public BagEntry (Object elt, int num) {
        this.elt = elt;
        this.num = num;
    }

}
```

To improve the running time of size() and count(), the values to be returned by these methods should be cached in instance variables.

So instances of BagMBSTEntries should have the following instance variables:

- **comp**: a comparator for determining the order of elements.
- **entries**: a mutable binary search tree whose elements are instances of BagEntry.
- **size**: the number of elements currently in the bag (includes duplicates).
- **count**: the number of *distinct* elements currently in the bag (does not include duplicates).

For example, Fig. 6 shows one possible representation of an instance of BagMBSTEntries that contains two As, three Bs and one C. (The shape of the tree depends on the order in which the elements are inserted.)

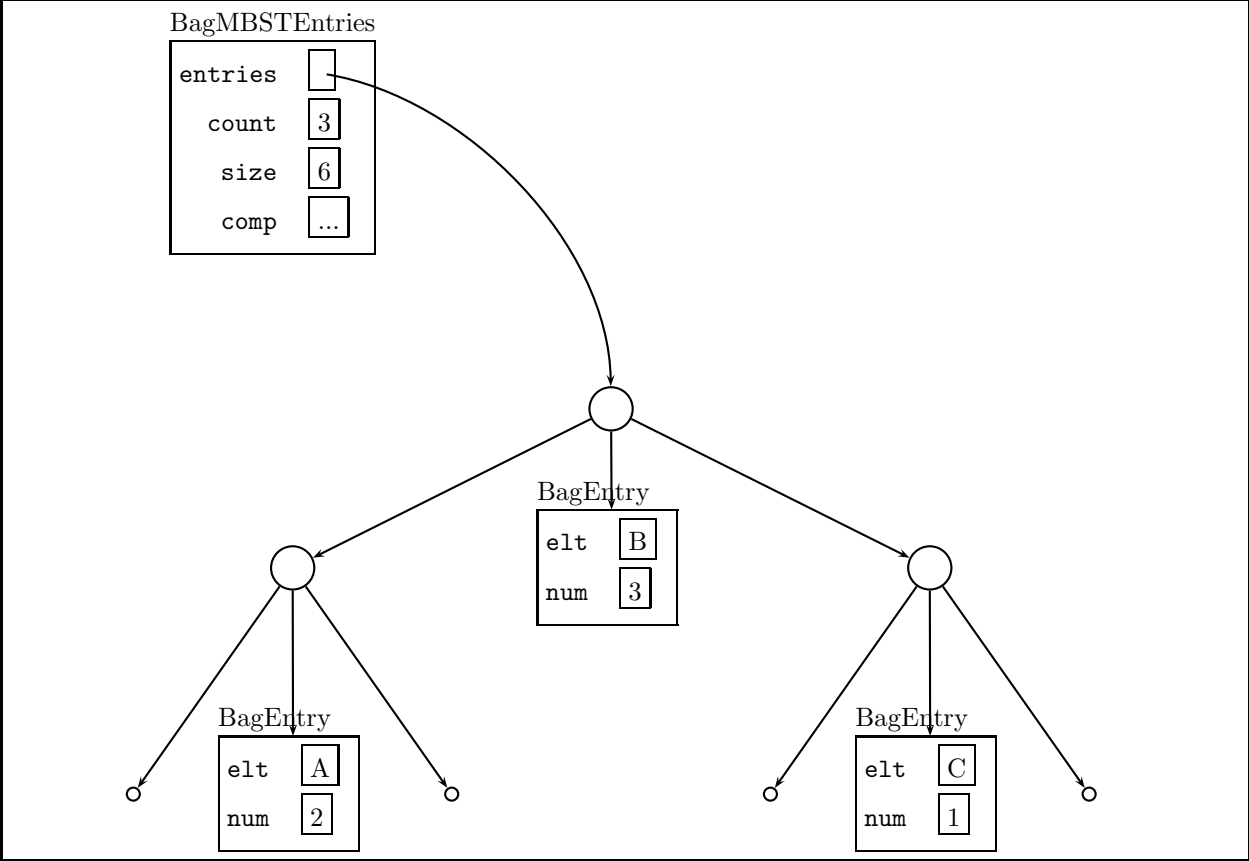


Figure 6: An example of a BagMBSTEntries instance.

a. [30]: Implementation

Flesh out the bag implementation in `BagMBSTEntries.java` using the representation described above. Test your implementation by executing `java BagMBSTEntries`, and turn in the transcript of this test for your final version of the code as part of your hardcopy submission.

b. [10]: Running Times

Give the (1) worst-case and (2) best-case running times of the following operations on a bag with n elements for your implementation of `BagMBSTEntries`. Use “Theta notation”, e.g. $\Theta(1)$, $\Theta(\log(n))$, $\Theta(n)$, $\Theta(n \cdot \log(n))$, $\Theta(n^2)$.

```
public boolean isMember (Object x);
public void insert(Object x);
public Object choose();
public Object deleteOne();
public boolean delete (Object x);
public boolean deleteAll (Object elt);
public void clear();
public ObjectList toList();
public Object clone ();
public int size();
public int count ();
public int occurrences (Object elt);
```

Notes:

- Mutable object trees are instances of the class `ObjectMTree` that are manipulated with the following class methods:

```
public static ObjectMTree leaf();
public static boolean isLeaf(ObjectMTree t);
public static ObjectMTree node(Object v, ObjectMTree l, ObjectMTree r);
public static Object value (ObjectMTree t);
public static ObjectMTree left (ObjectMTree t);
public static ObjectMTree right (ObjectMTree t);
public static void setValue (ObjectMTree t, Object newValue);
public static void setLeft (ObjectMTree t, ObjectMTree newLeft);
public static void setRight (ObjectMTree t, ObjectMTree newRight);
```

The `BagMBSTEntries` class has been configured so that the above operations are accessible via the `OMT.` prefix.

- `ObjectList` operations are accessible via the `OL.` prefix.
- You may want to define private auxiliary methods that will help you implement other methods. In particular, you may want to defined “generic” operations for inserting, deleting, searching for, etc. elements in a mutable binary search tree (but you don’t have to).
- Depending on how your binary search operations are defined, you might directly use `comp`, or you might need to “lift” `comp` via the `BagEntryComparator` class discussed in class.

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS230 Problem Set 7

Due Friday, November 15

Name:

Date & Time Submitted:

Collaborators (*anyone you worked with on the problem set*):

By signing below, I attest that I have followed the collaboration policy as specified in the Course Information handout.

Signature:

*In the **Time** column, please estimate the time you spend on the parts of this problem set. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the **Score** column when grading your problem set.*

Part	Time	Score
General Reading		
Problem 1 [30]		
Problem 2 [30]		
Problem 3 [40]		
Total		