

Project 3: Doubly-Sorted List (250 pts)

Due at 11:59pm

Monday, Mar 23

*You are recommended to finish the project by **Friday, Mar 13** for both a happy spring break and a good preparation for Exam 2 on March 26.*

1. List Structure

In this project, you are asked to implement a sorted linked list structure to help a fruit store update its inventory. To simplify this task, we make the following assumptions:

- a) The store sells fruits **by unit** (e.g., an apple, a bunch of grapes, etc.) not by weight.
- b) Every type of fruit, regardless of its quantity on stock, is displayed in **exactly one** storage bin.
- c) Storage bins are numbered consecutively starting at 1, and as many bins as needed are available.
- d) The names of fruits from the input or passed as arguments to method calls will **always** be from the list given in Appendix A. (These names are in lower case English letters.)

A doubly-sorted list (DSL) consists of two doubly-linked lists (DLLs) that share the same set of nodes, and are **sorted** respectively by two different data fields. In the DSL, exactly one node N is created for every type of fruit currently on stock. As shown in the top row of Fig. 1, such a node has three data fields: fruit, quantity, and bin, which respectively store the name of the fruit (as a String object), its quantity in units, and the number of its storage bin. The figure also displays two nodes with actual data values. They record 50 apples stored in bin 5, and 100 bunches of grapes in bin 8.

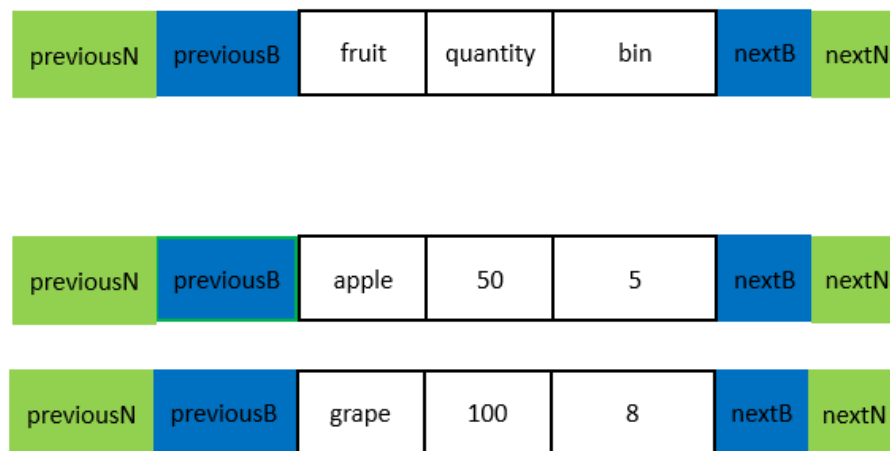


Fig. 1. A node has three data fields and two pairs of links. The bottom two nodes have their data fields filled with values.

In addition to the three data fields, the node N also has two pairs of links: $nextN$ and $previousN$, and $nextB$ and $previousB$. The first pair locates the two nodes created for the fruits whose names immediately precede and succeed $N.fruit$, the name of the fruit represented by N , in the alphabetical order. All the $nextN$ and $previousN$ links thus form a doubly-linked list (DLL) that orders the nodes by fruit name. The list, referred to as the **N-list**, is accessed via a variable $headN$ which references the node for the fruit first in the alphabetical order. Fig. 2 shows an N-list for four different types of fruits. Note that i) the N-list has no separate header node as some DLLs do, and ii) the last node (“pear”) has its $nextN$ link reference the first node (“apple”).

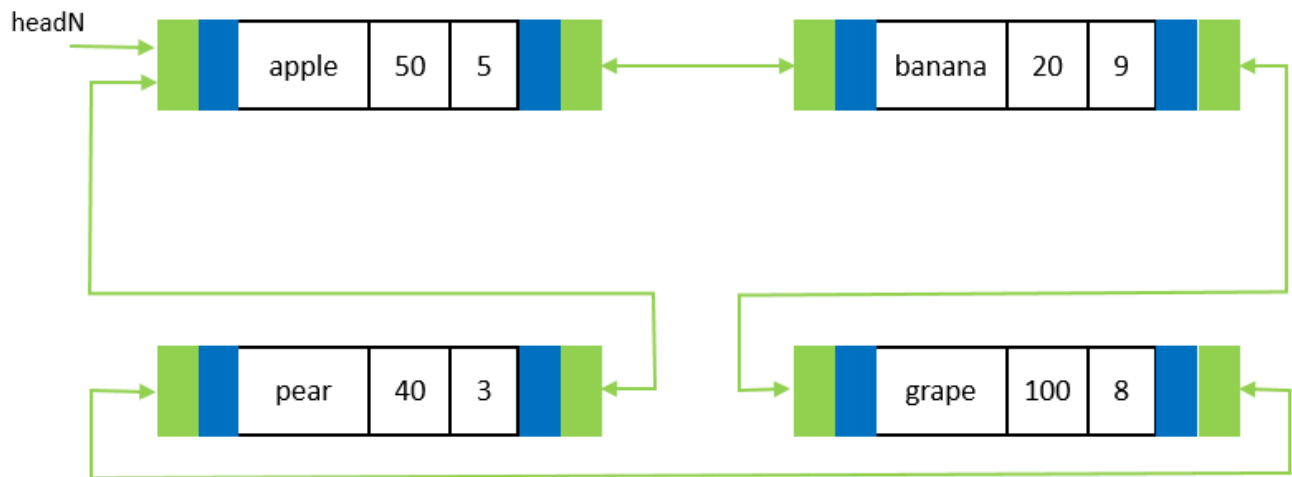


Fig. 2. N-list sorted by fruit name. Every pair of $nextN$ and $previousN$ links is represented by bi-directional arrows.

At the node N , the other two links $nextB$ and $previousB$ respectively reference the nodes for the fruits stored in the previous and next **non-empty** bins in the numerical order. For instance, suppose N references the “apple” node in Fig. 2. As the node shows, all apples are displayed in bin 5 at the fruit store. The last non-empty bin is bin 3, which stores pears, so the link $N.previousB$ references the “pear” node. Meanwhile, the next non-empty bin is bin 8. Therefore, $N.nextB$ references the “grape” node. All the $nextB$ and $previousB$ links induce a second DLL sorted by bin number. It is called the **B-list**. Fig. 3 displays the B-list for the same four nodes from Fig. 2.

To distinguish between the N- and B-lists, all the $nextN$ and $previousN$ links will be colored green as in Fig. 2, while all the $nextB$ and $previousB$ links will be colored blue as in Fig. 3. The DSL is now displayed compactly in Fig. 4 by simply merging the two DLLs from Figs. 2 and 3.

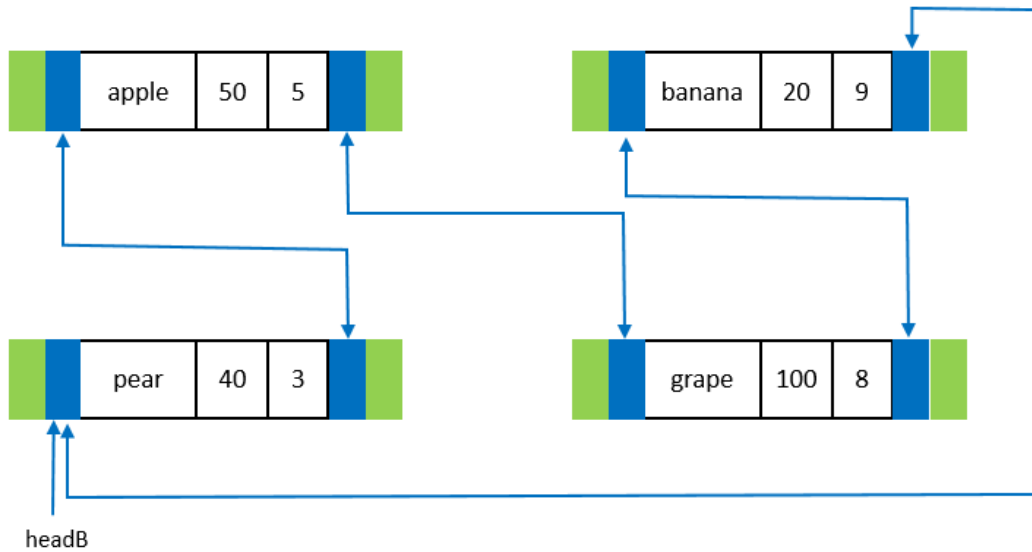


Fig. 3. B-list linking the same four nodes from Fig. 2.

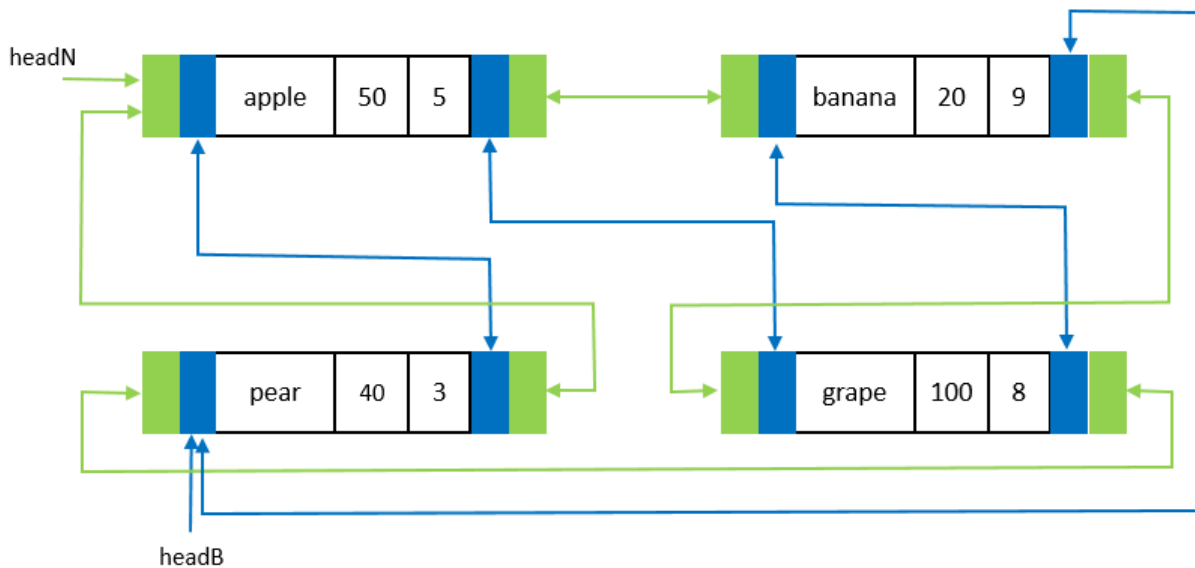


Fig. 4. Doubly-sorted list merging the N- and B-lists from Figs. 2 and 3.

When all fruits have been sold out or disposed, the DSL has empty N- and B-lists. The following conjunction is true:

`headN == null && headB == null`

Note that headN and headB must either be null together or reference some nodes (which could possibly be the same node).

2. Constructors

There are three constructors:

```
public DoublySortedList()  
public DoublySortedList(String inventoryFile) throws FileNotFoundException  
public DoublySortedList(int size, Node headN, Node headB)
```

The first one is a default constructor that initializes an empty DSL. Fruits will be added to the list later using addition methods introduced later. The second constructor builds a DSL over an inventory file, as the one below:

pear	40	3
apple	50	5
banana	20	9
grape	100	8

An input inventory file is assumed to always meet the following format requirements such that there is **no need** to check for correctness of formatting:

- e) *Fruits are listed on consecutive lines.*
- f) *Each line displays the name of a different fruit, its quantity, and the number of its storage bin, with at least one blank or tab in between.*

As in the earlier example, the fruits on an inventory list may not be in the alphabetical order. In the first stage of construction, you are asked to scan the list and create a node over every scanned line. The node's two pairs of links (nextN, previousN) and (nextB, previousB) are initialized to represent the order of the scan. The links headN and headB both reference the first created node. The scan creates two DCLs, neither of which is sorted at the moment.

In the second stage, an **insertion sort** is carried out on each DCL by the following method:

```
public void insertionSort(boolean NList, Comparator<Node> comp)
```

Two comparator classes, NameComparator and BinComparator, are provided for the insertion sorts by name and by bin number to construct the N- and the B-lists, respectively. Please note that sorting here is performed on the nodes. You **cannot** first put the fruit names or their bins in an array and then sort the array instead.

The third constructor has been implemented. It will be used for splitting a DSL into two (to be described in Section 6) and by the TA for testing.

3. Stocking Fruits

The list is updated whenever new fruits are stocked. Every new fruit type leads to the creation of a node. For compactness of display, all fruits of this type should be placed into the first empty bin by number.

3.1 Insertion

Addition of a fruit in some specified quantity is carried out by the following method:

```
public void add(String fruit, int n) throws IllegalArgumentException
```

It starts out with a sequential search in the N-list for fruit. Two possible situations may arise:

- 1) The search finds a node N housing fruit, simply increase the N .quantity field by n . This is illustrated in Fig. 5 below.

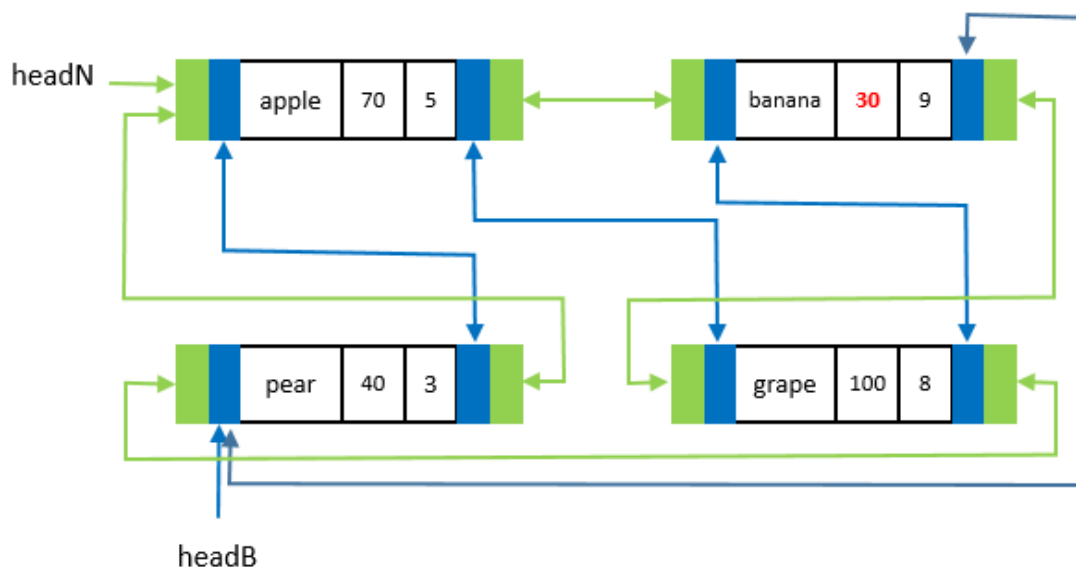
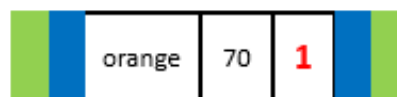


Fig. 5. Execution of `add("banana", 10)` on the DSL in Fig. 4. The changed value of the quantity field in the "banana" node is colored red.

- 2) If no node for fruit is found after the search, the fruit is new on the stock. In this case, a new node is created to store the fruit in the first available bin. To illustrate, suppose that `add("orange", 70)` is executed on the DSL in Fig. 4. A search of the B-list finds the first empty bin to be bin 1. Create the following node (where the bin number is shown in red).



Next, add the node to the N-list and the B-list by calling two helper methods: `insertN()` and `insertB()`. The first method determines the "orange" node to be between the

“grape” and “pear” nodes, while the second method determines it to be the first node of the B-list. In Fig. 6, all the links (including headB) to be removed are marked with red crosses. Fig. 7 displays the DSL after addition of the new node (which is now referenced by headB).

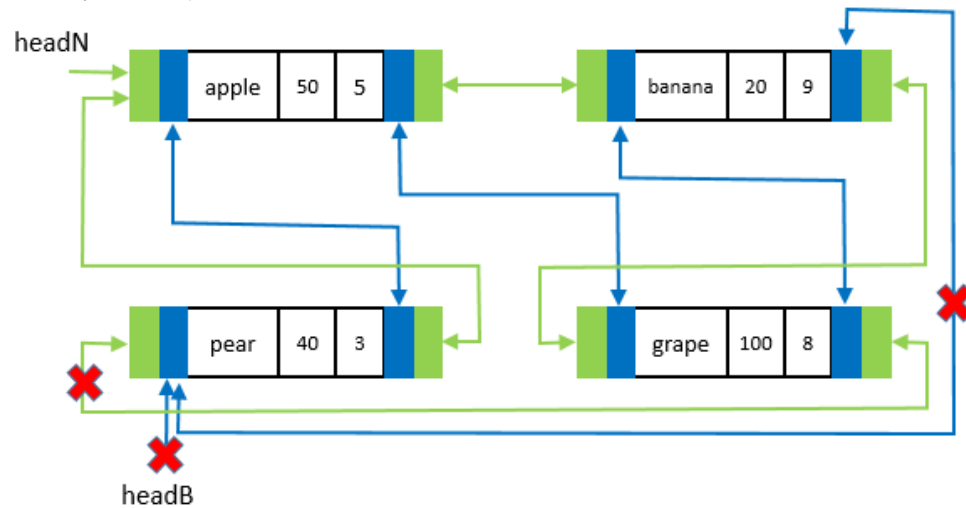


Fig. 6. Links marked by crosses are to be deleted during a call `add(“orange”, 70)` on the DSL in Fig. 4.

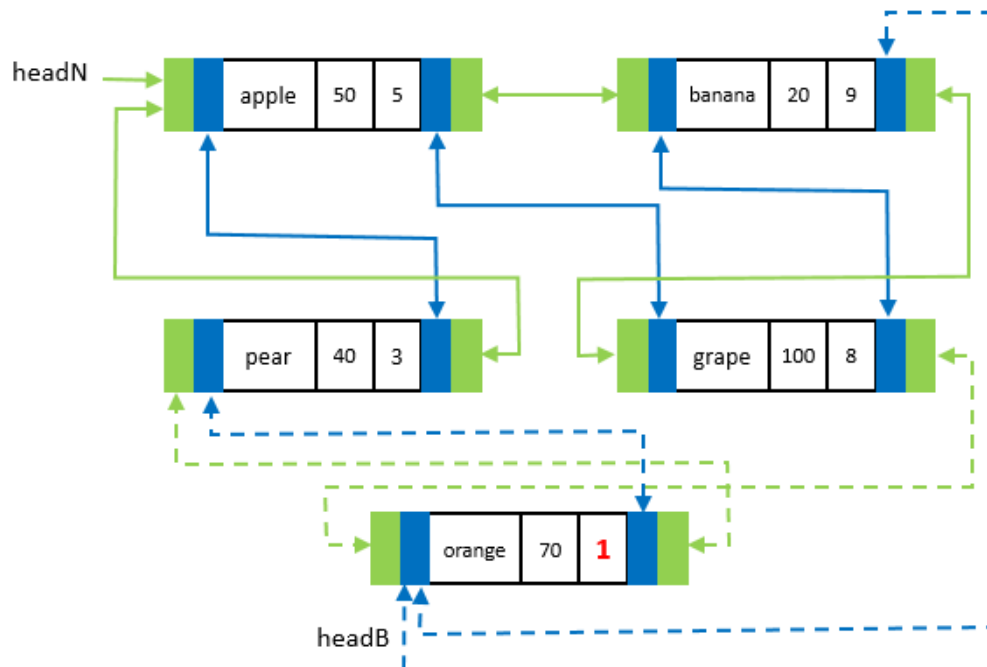


Fig. 7. DSL after the node insertion. Dashed blue arrows represent newly added links.

You are asked to implement two private methods below to carry out actual insertions of a node node into the N- and B-lists, respectively.

```
private void insertN(Node node, Node prev, Node next)
private void insertB(Node node, Node prev, Node next)
```

Each method inserts node between two nodes prev and next via updating either some previousN and nextN links or some previousB and nextB links.

Remember to handle two non-exclusive **special situations** below:

- 1) The N- and B-lists were empty before the insertion.
- 2) The newly inserted node becomes the first node of the N-list or the B-list.

3.2 Restocking

Regularly the store is restocked with fruits of multiple types. This is implemented by the following method:

```
public void restock(String fruitFile) throws FileNotFoundException
```

The parameter fruitFile is the name of a file that lists fruits line by line, where each line starts with the fruit name, follows with one or more blanks or \t, and ends with its quantity. Below is an example file.

```
pear      40
apple     50
grape     100
banana    20
```

Assumption e) on the inventory file also applies here, along with the following assumption:

g) Each line displays the name of a different fruit and its quantity.

The lines in the fruitFile are **not** sorted. For efficiency, you are asked to sort the fruits by name using the version of quicksort introduced in class. Within the restock method, declare two arrays fruit[] and bin[] to store the fruits and their bins, and implements the following two private methods:

```
public void quickSort(String fruit[], Integer quant[], int size)
private void partition(String fruit[], Integer quant[], int first, int last)
```

After the sorting, every pair of entries in the arrays fruit[] and quant[] at the **same index** must store information about the **same fruit**.

Traverse the array fruit[], the N-list, and the B-list simultaneously. Suppose that the fruit type fruit[i] is being scanned, where i is the index, and the node being visited is N such that N.previous.fruit < fruit[i] and N.fruit >= fruit[i] according to the alphabetical order. Two cases may arise:

- 1) fruit[i] and N.fruit are equal. Simply increase N.quantity by quant[i].
- 2) N.fruit > fruit[i]. Namely, fruit[i] is of a new type. There are two situations below.

- a. $N == \text{headB} \ \&\& \ \text{headB}.\text{bin} > 1$. Create a new node M for $\text{fruit}[i]$ with its bin field set to 1.
- b. Otherwise, if $N == \text{headB} \ \&\& \ \text{headB}.\text{bin} == 1$, set S to headB . If N is not headB , S has already been set to the last node stop . The bins numbered up to $S.\text{bin}$ are all occupied. Continue traversing the B-list starting from S . The traversal will stop at some node B such that the first empty bin is numbered $B.\text{bin} + 1$. Create a new node M for the fruit, setting $M.\text{bin}$ to $B.\text{bin} + 1$.

Add the node M to the N-list before the node N , and to the B-list as headB in the case 2a) or after the node B in the case 2b) by calling the private methods `insertN()` and `insertB()`, respectively. Set the node stop S to M .

The restocking algorithm described above runs in linear time.

4. Selling Fruits

Selling of fruits results in updates of their corresponding nodes. When fruits of one type are sold out, the node representing the fruit type must be deleted.

4.1 Removal

There are two removal methods, by fruit name and by bin number:

```
public void remove(String fruit)
public void remove(int bin) throws IllegalArgumentException
```

A removal starts by searching either in the N-list with the name `fruit` or in the B-list with the number `bin`. If no matching node is found, simply return. Otherwise, suppose the matching node is N . Call the following private method on N to remove it.

```
private void remove(Node node)
```

This method deletes the node from both the N-list and the B-list. It also does the following:

- 1) Update the heads (`headN` and `headB`) of either or both DLLs when necessary.
- 2) Set `headN` and `headB` to `null` if the two DLLs become empty after the deletion.

An example of removal by fruit name is shown in Fig. 8. Another example, of removal by storage bin, is shown in Fig. 9.

4.2 Sale

The shop does single sale and bulk sale, which are implemented by the two methods below:


```

public void sell(String fruit, int n) throws IllegalArgumentException
public void bulkSell(String fruitFile)

```

The method `sell()` first searches the N-list to see if the fruit is on stock. Nothing needs to be done if the answer is no. Otherwise, a node N is located for the fruit. The following is performed:

- 1) $n \geq N.\text{quantity}$. Call the internal method `remove(N)` to delete the node.
- 2) $n < N.\text{quantity}$. Decrease $N.\text{quantity}$ by n .

Fig. 10 shows the DSL after calling `sell("banana", 5)` on the list in Fig. 4.

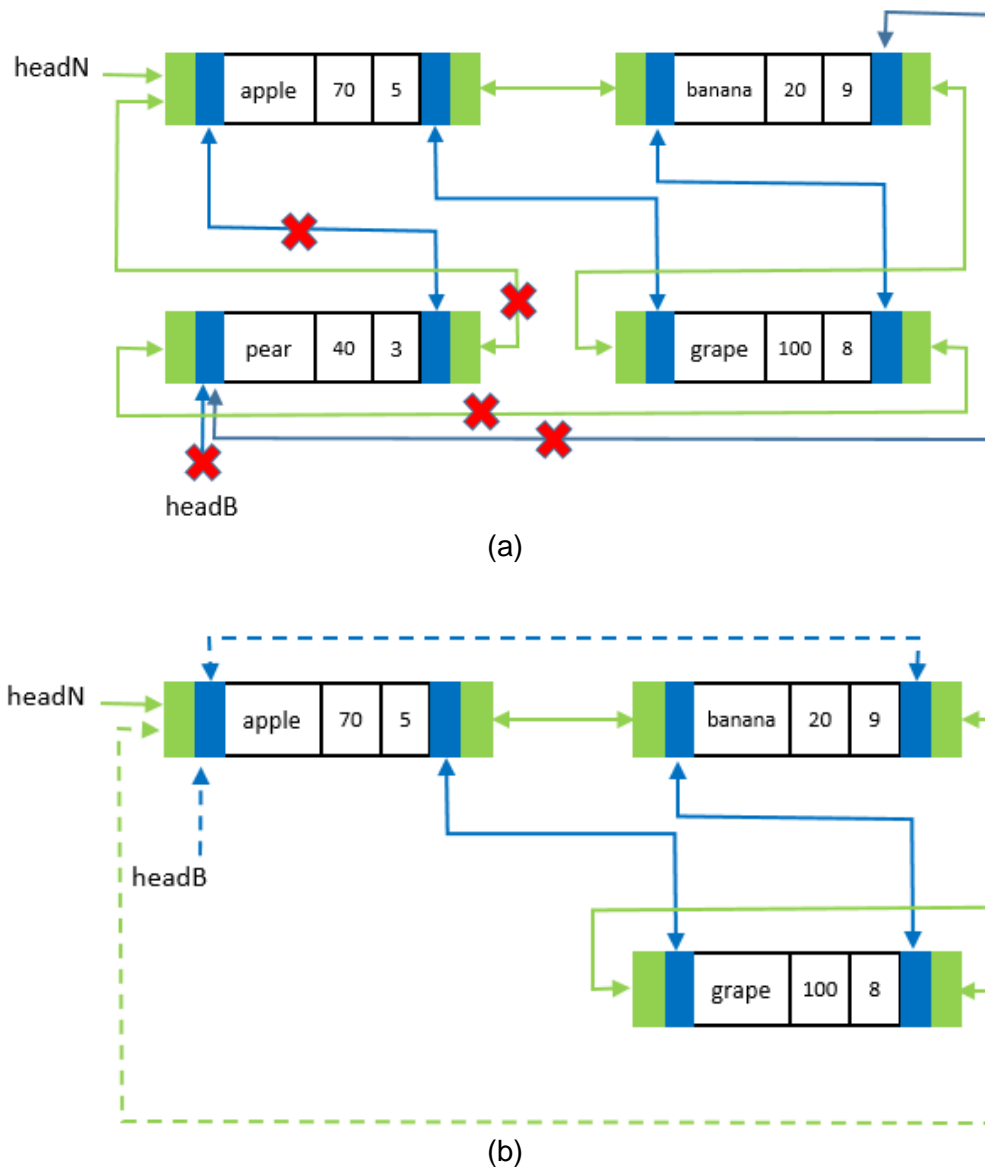
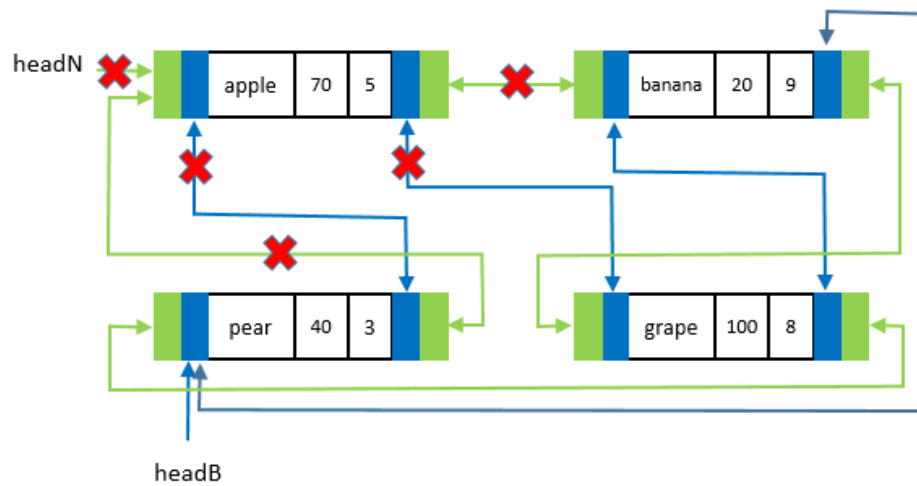
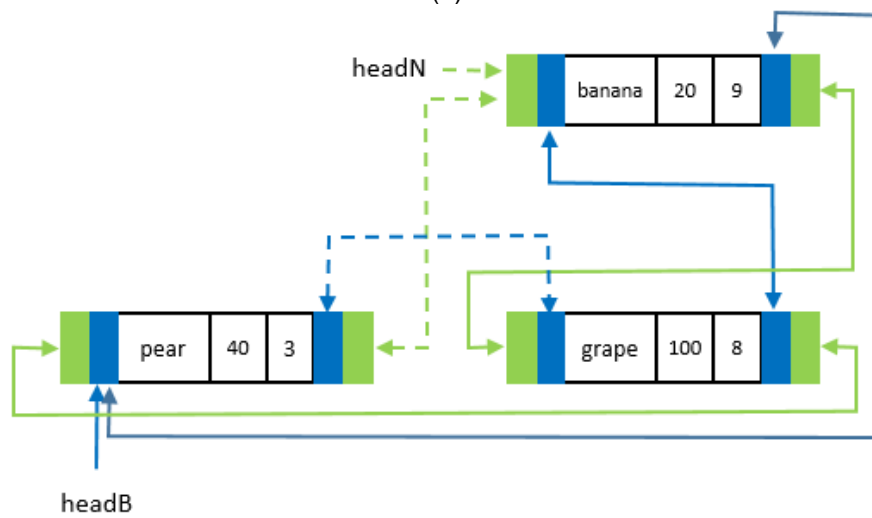


Fig. 8. Calling `remove("pear")`: (a) before and (b) after. Red crosses in (a) mark the links to be deleted, and dashed arrows in (b) mark the added ones. The reference `headB` is reset to the "apple" node.



(a)



(b)

Fig. 9. Calling remove(5).

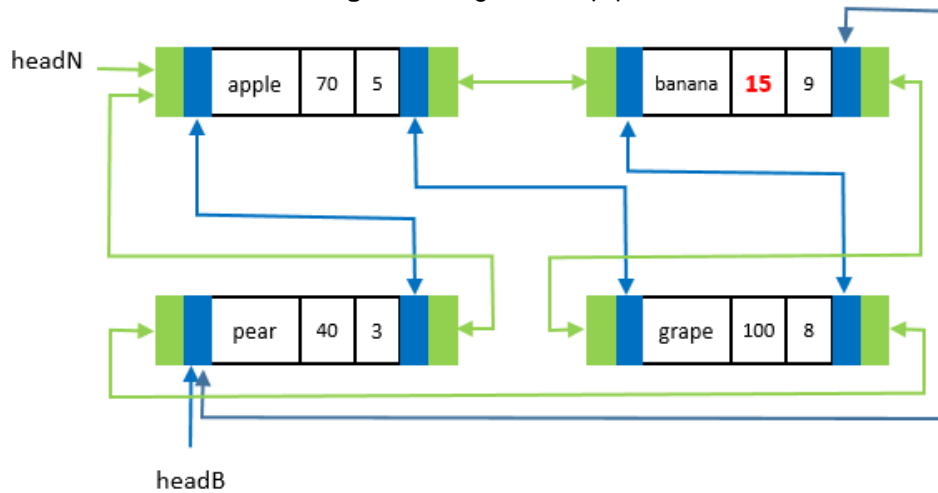


Fig. 10. DSL in Fig. 4 updated after sale of five bananas. The red number in the “banana” node shows the quantity after the sale.

While no check is needed for the correctness of the parameter `fruit`, an exception needs to be thrown if the parameter `n` is less than zero.

The method `bulkSell()` accepts a file with fruit names and quantities in exactly the same format accepted by the method `restock()` (see Section 3.2). Assumptions e) from Section 2 and g) from Section 3.2 both hold, so there is no need for file format checking. This method `bulkSell()` processes an order of multiple types of fruits in specified quantities. It executes the following steps:

- 1) Sort the purchased fruits and their quantities by name using the private method `quickSort()`, in a similar way as described for sorting in implementing `restock()`.
- 2) Simultaneously traverse the `N`-list and scan the sorted list of the fruits to be purchased. If the next fruit on the list is not on stock, just ignore its order. Otherwise, a node N with the next purchased fruit will be encountered. Let m be the ordered quantity of this fruit. Do the following.
 - a. If $m < 0$ throw an exception.
 - b. If $m == 0$, ignore.
 - c. If $0 < m < N.quantity$, decrease $N.quantity$ by m .
 - d. If $m \geq N.quantity$, call `remove(N)` to delete the node.

Suppose that there are n nodes in the DSL and k fruits on the purchase list `fruitFile`. The running time of your implementation of `bulkSell()` should be $O(n + k \log k)$.

Still consider the DSL in Fig. 4. Suppose the `fruitFile` has the following contents:

```
grape 200
pear 20
apple 10
jackfruit 5
```

After the sorting in Step 1, the arrays inside `bulkSell()`, say, `fruit[]` and `quantity[]`, to store the names and quantities of the ordered fruits, will respectively have the contents

```
{"apple", "grape", "jackfruit", "pear"}
{10, 200, 5, 20}
```

Step 2 is then carried out via simultaneous traversals of the `N`-list and the array `fruit[]`. The fruit "jackfruit" is not on the list. The node "grape" gets deleted, as shown in Fig. 11.

5. Inquiry and Storage Management

Queries for fruit availability can be performed. Every once in a while, the fruit store checks its inventory. Periodically, it compacts the space by re-arranging the bins with fruits next to each other. Occasionally, it needs to clear all the fruits on stock (due to deterioration of unsold fruits, for instance).

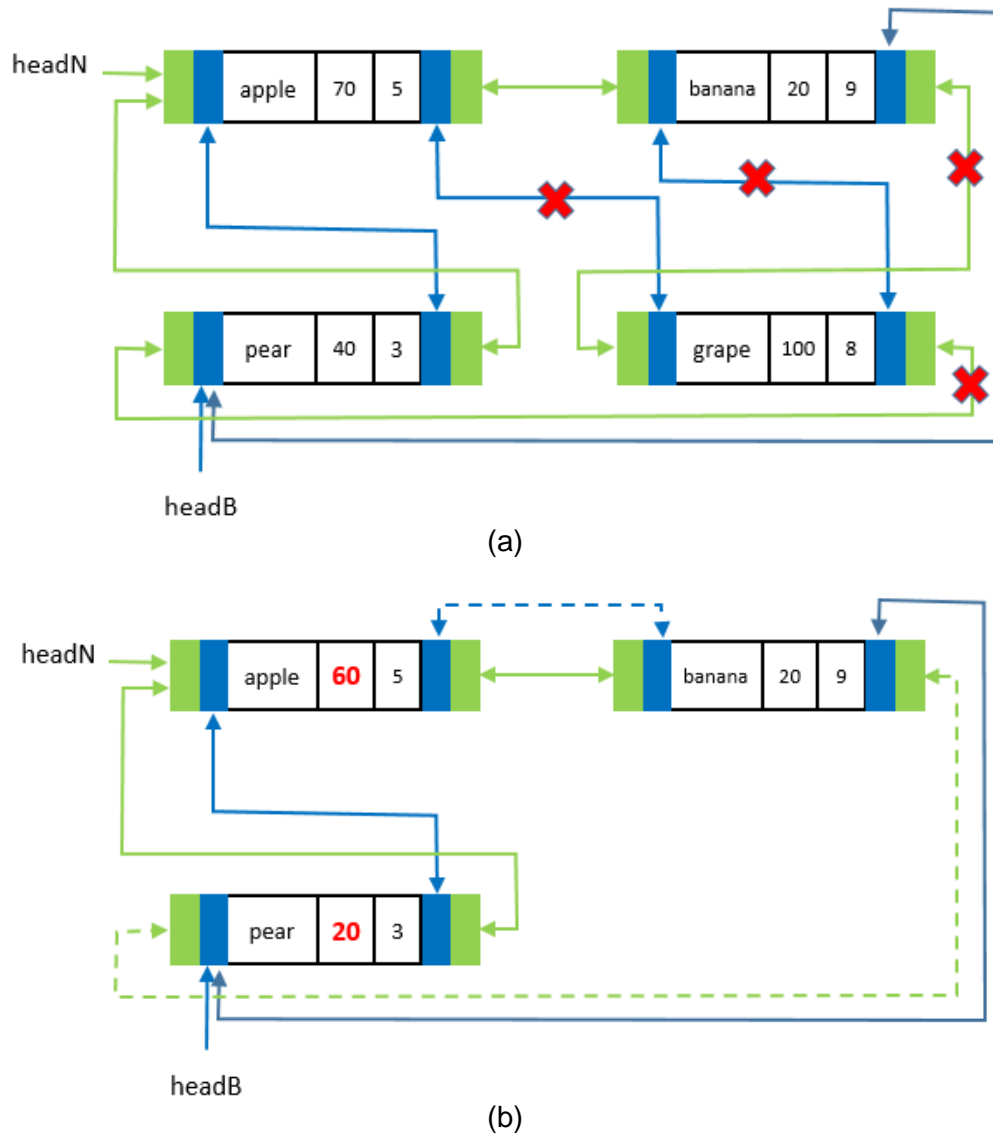


Fig. 11. The DSL from Fig. 4 (a) before and (b) after calling `bulkSell()` with the shown `fruitFile`. Entries colored red in (b) have been modified.

5.1 Inquiry and Output Methods

This category has three methods:

```
public int inquire(String fruit)
public String printInventoryN()
public String printInventoryB()
```

The first method checks the quantity of a fruit on stock. The second method outputs a string that is printed out to be an inventory listing fruits in the alphabetical order. Here is sample output of the method from the DSL in Fig. 4.

fruit	quantity	bin
apple	50	5
banana	20	9
grape	100	8
pear	40	3

The following formatting rules apply to the output.

- 1) Entries within each column must be left-aligned.
- 2) The first characters in two adjacent columns are 16 character spaces apart.

You may use the length of a fruit name (which is at most 11 according to the fruit list in Appendix A). You can make the following assumption:

h) The values in the quantity and bin fields will not exceed 3 digits.

The method `printInventoryB()` outputs a string that is printed out to be an inventory sorted by the bin number.

bin	fruit	quantity
3	pear	40
5	apple	70
8	grape	100
9	banana	15

The two output formatting rules for `printInventoryN()` apply to `printInventoryB()`.

You also need to override the method `toString()`, which converts a DSL object into a string that would print out in exactly the same format required for `printInventoryN()`.

5.2 Storage Methods

There are two methods to deal with storage bins:

```
public void compactStorage()
public void clearStorage()
```

The first method rearranges the bins containing fruits to be consecutive, starting at 1, while maintaining their original storage order. To implement this method, you need only traverse the B-list. See Fig. 12 for an example.

The method `clearStorage()` clears up the N-list and the B-list by setting `headN` and `headB` both to `null`.

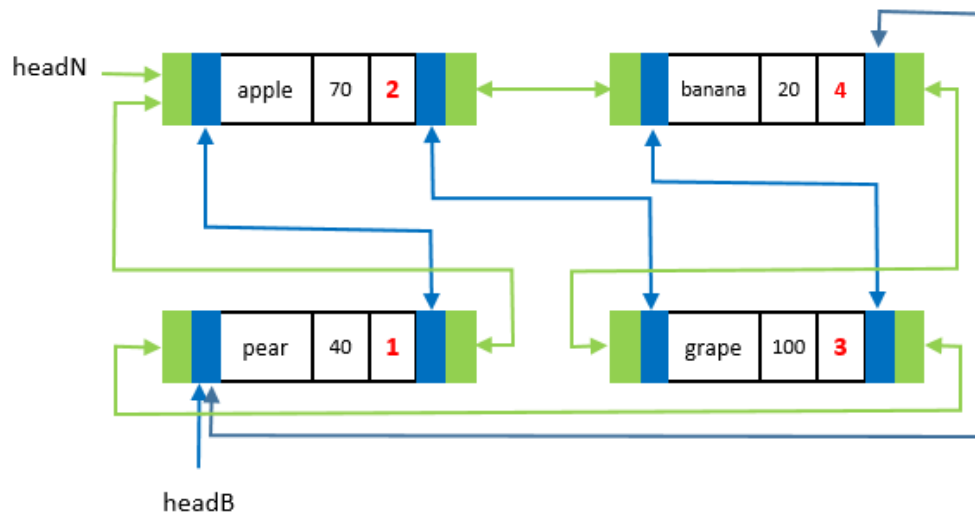


Fig. 12. The DSL in Fig. 4 after storage compaction. Fruits are now stored in bins 1, 2, 3, 4.

6. List Splitting

With an extra display room acquired, the store wants to split the original DSL of size n into two: DSL1 and DSL2 such that DSL1 stores the first $\lfloor n/2 \rfloor$ (i.e., $n/2$ in Java) fruits, while DSL2 stores the last $\lfloor n/2 \rfloor$ (i.e., $n - n/2$ in Java) fruits. We suppose that the fruits are ranked $0, 1, \dots, n - 1$ by name. This operation is performed by the method

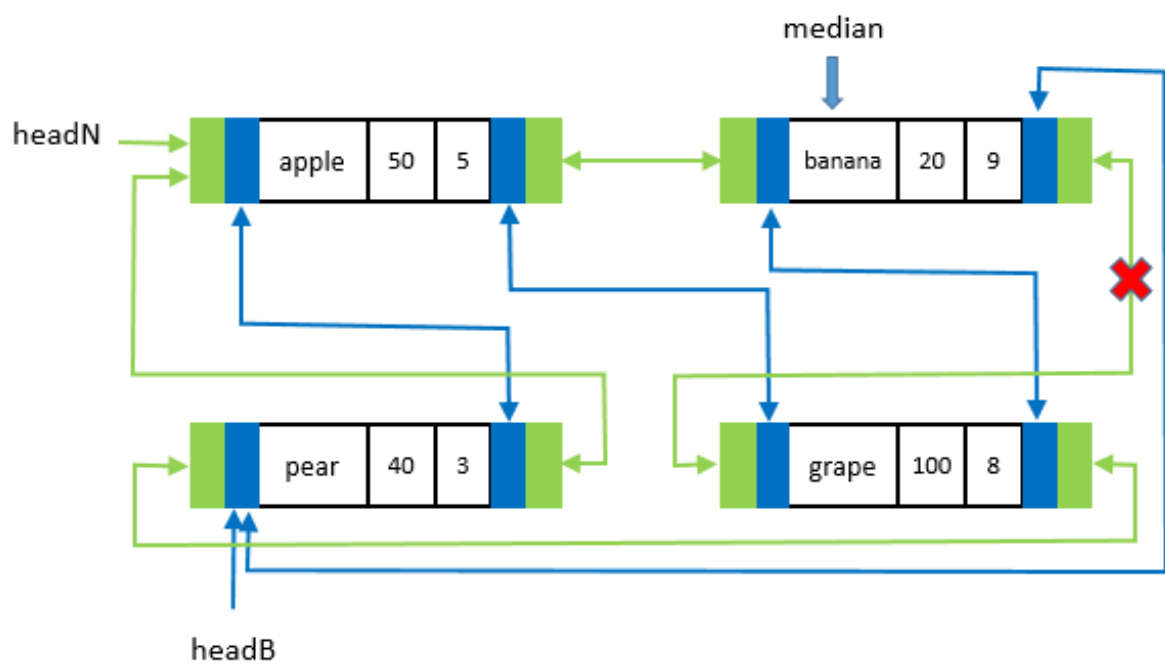
```
public Pair<DoublySortedList> split()
```

which returns an object o of the provided `Pair` class such that $o.first$ references DSL1 and $o.second$ references DSL2.

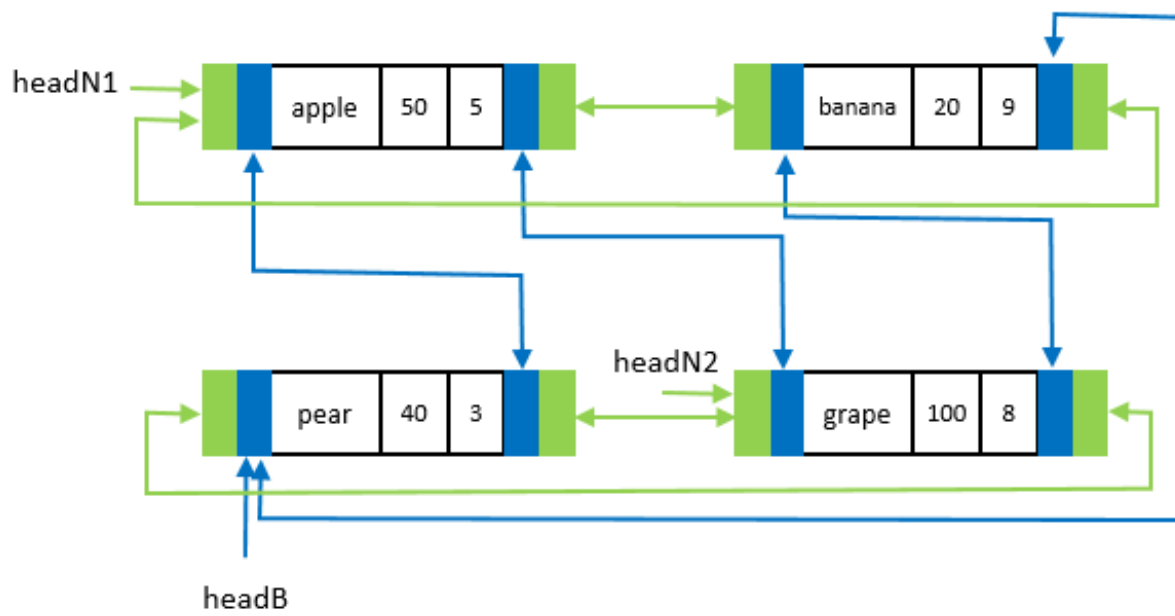
The algorithm for splitting carries out the following steps.

- 1) Split the N -list into DST1 and DST2 at the median node M by fruit name. The node is found via traversal of the N -list. The two resulting lists are sorted by fruit name but not by storage. Each needs to have a B-list reconstructed as a doubly-linked list.
- 2) Traverse the B-list. For every node N encountered, do the following:
 - a. If $N.fruit \leq M.fruit$, add the node N to the B-list of DST1.
 - b. If $N.fruit > M.fruit$, add the node N to the B-list of DST2.

Fig. 14(a)-(c) illustrate the execution of the algorithm on the DSL in Fig. 4.



(a)



(b)

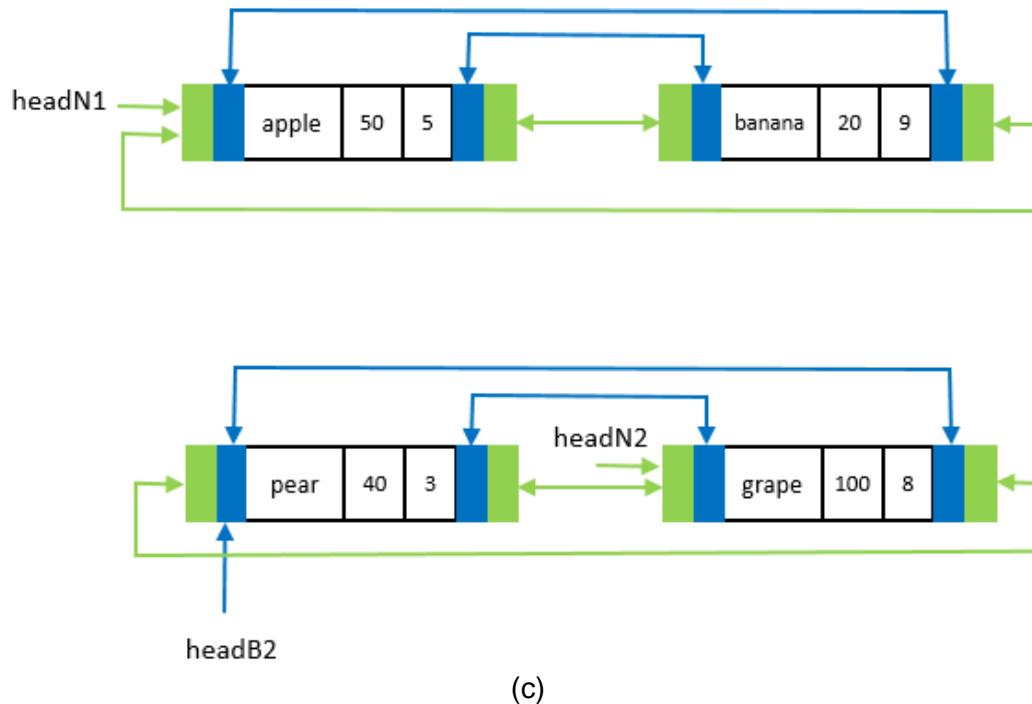


Fig. 13. Splitting a DSL into two. (a) Find the median node “banana” in the alphabetical order. (b) Split the original N-list into two. (c) Build two B-lists over the original B-list, using the median fruit name for partitioning.

7. Submission

Write your classes in the `edu.iastate.cs228.hw3` package. **Turn in the zip file not your class files.** Please follow the guideline posted under Documents & Links on Blackboard Learn.

You are not required to submit any JUnit test cases. Nevertheless, you are encouraged to write JUnit tests for your code. Since these tests will not be submitted, feel free to share them with other students.

Include the Javadoc tag `@author` in every class source file you have made changes to. Your zip file should be named `Firstname_Lastname_HW3.zip`.

A. Fruit Names

Below is a list of 36 common fruits relevant to the project:

- apple
- apricot
- avocado

- banana
- blackberry
- blueberry
- cantaloupe
- cherry
- coconut
- cranberry
- date
- durian
- fig
- grapefruit
- grape
- jackfruit
- kiwi
- kumquat
- lemon
- lime
- lychee
- mango
- mangosteen
- nectarine
- orange
- papaya
- peach
- pear
- pineapple
- plum
- pomelo
- prune
- raspberry
- strawberry
- tangerine
- watermelon