1 Introduction

In this assignment you will implement a data structure to store a collection of items of the same type; we call this structure a **doubling list**. A doubling list is a doubly-linked list with dummy head and tail nodes, where each successive node has the capacity to store twice as many elements as its predecessor. Figure 1 below shows an example of a doubling list; the logical indices of its elements are indicated below the nodes.

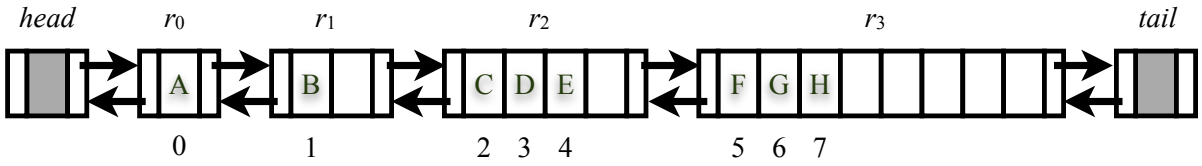

Figure 1: A doubling list of size 8 and capacity 15.

Each node stores its elements in an array of the element type; the jth node of the list has an array of length 2^j. **An array may have null entries, which are not considered part of the list - they are simply available slots**. The list elements within a node are stored consecutively, starting at position 0. The logical index of an element within the list is represented by two values: a **reference** r to the node containing the element, and the **offset** within that node's array. For example, in Figure 1 element E, whose logical index is 4, is in node r_2 at offset 2; element G is in node r_3, at offset 1. As Figure 2 shows, an empty list simply contains a head node and a tail node.

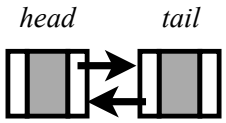

Figure 2: An empty list.

In general, a doubling list with k nodes (aside from the head and tail nodes) has the capacity to hold up to $1+2+\cdot+2^{k-1} = 2^k - 1$ elements. It might not actually store this many elements, as some of its nodes may not be full (see Figure 1). Indeed, some, but not all, arrays may be empty. When all the nodes are full and we need to add one more element, a new node is added, with double the capacity of the last node, giving us the space to insert the new element — the precise rules are given in Section 3.2).

- A doubling list with zero nodes contains zero elements (obviously).
- A doubling list with one node must contain exactly one element.
- For $k \geq 2$, a doubling list with k nodes must contain **at least** 2^{k-2} elements.

Thus, roughly speaking, a doubling list must be used to at least one quarter of its maximum capacity. This is satisfied by the list of Figure 1, where k is 4, and the number of elements is 8, which is bigger than or equal to $2^{4-2} = 4$. To guarantee that the occupancy constraints are satisfied, doubling lists must occasionally be compacted after an element is removed from it, if the number of empty slots becomes too large (see Section 3.3).

The peculiar occupancy constraints of doubling lists guarantee certain nice properties. For instance, they imply that an n-element list has $O(\log n)$ nodes and that the total time to perform n add() operations to the end of the list, starting from an initially empty list, is $O(n)$ – i.e., the average time

(or, more precisely, the amortized time) per addition is $O(1)$. One can show that deletion has certain nice properties as well.

The objectives of this project are:

- to give you experience at working with linked data structures,
- to allow you to become intimately familiar with the `List` and `ListIterator` interfaces, and
- to give you the opportunity to practice your debugging skills.

2 Task

You need to write the code for the generic `DoublingList<E>` class, which implements doubling lists for items of the same type E. `DoublingList<E>` must extend Java's abstract class `AbstractSequentialList<E>`. A `DoublingList` object should not allow `null` elements. Thus, your `add` methods (those within the iterator as well as those implemented without the iterator) should explicitly throw a `NullPointerException` if a client attempts to add a `null` element. Note that `AbstractSequentialList<E>` is a partial implementation of the `List<E>` interface. In `AbstractSequentialList` the `size()` and `listIterator()` methods are abstract, but all other methods have default implementations using the list iterator. This means that, in principle, all you need to produce a correct implementation of `DoublingList` is a correct implementation of the list iterator for the class. Nevertheless, you are required to override the following methods of `AbstractList` **without** using the iterator:

`boolean add(E item)`, `void add(int pos, E item)`, `E remove(int pos)`

The purpose of asking you to do this is to help you to get partial credit for implementing the expansion and compaction rules, even if your iterator is not completely correct.

3 Implementing `DoublingList`

Here we describe the implementation details of `DoublingList`. Section 3.1 explains the mechanism for accessing the element with a given index. Sections 3.2 and 3.3 give the rules for `add()` and `remove()`, respectively. Section 3.4 describes the require iterators. Finally, Section 3.5 discusses two methods that are useful for visualizing doubling lists and debugging code.

We assume that a `DoublingList` object has a `cap` field, which equals the maximum number of elements that the list can hold. Thus, for a doubling list with k nodes, `cap` $= 2^k - 1$.

3.1 Finding Indices

Your index-based methods `remove(int pos)`, `add(int pos, E item)` and `listIterator(int pos)` must not traverse every element in order to find the node and offset for a given index `pos`. Instead, you should be able to skip over nodes just by looking at the number of elements in the node. For example, for the list of Figure 1, to find the node and offset for logical index 3, we see one element in node 0, plus one in node 1, which makes 2, plus three in node 2, which is 5. Since 3 is greater than 2 and less than or equal to 5, index 3 must be in node 2.

You will find it helpful to represent a node and offset using a simple inner class similar to the one shown below. Then you can create a helper method something like the following. The time taken by

```
// returns the node and offset for the given logical index
NodeInfo find(int pos) {...}

private class NodeInfo
{
  public Node node;
    public int offset;
    public NodeInfo(Node node, int offset)
    {
      this.node = node;
      this.offset = offset;
    }
}
```

Figure 3: Representing node references and offsets.

`find()` should be linear in the number of nodes in the list; that is, it should be $O(\log n)$, where n is the number of elements (as given by `size()`). This requires that `add()` and `remove()` be implemented as described in the next sections.

3.2 The `add()` Rules

To implement `add()`, we rely on two basic operations, **leftward shift** and **rightward shift**, in both of which the input is a list L and a logical index i into L. We assume that r is a reference to the node in L containing element i and that node r has no empty slots. The operations shift elements of L left or right, starting at element i, leaving an empty slot at the array entry of r that initially contained element i. This slot will be used to insert the new ith element. The operations are as follows.

- **Leftward shift at** i. Let t be the rightmost predecessor of r that has empty slots (we assume that such a node exists). Then, we shift by one position to the left all elements with logical index less than or equal to i stored in the nodes between the successor of t and r inclusive. See Figure 6.
- **Rightward shift at** i. Let t be the leftmost successor of r that has empty slots (we assume that such a node exists). Then, we shift by one position to the right every element with logical index greater than or equal to i stored in the nodes between r and t inclusive. See Figure 7.

Note that both operations require moving elements across the boundaries between adjacent nodes.

Suppose that we need to add element X at logical index i to a doubling list L with k nodes. Remember that adding an element without specifying an index is the same as adding the element at index `i = L.size()`.

Case 1. `L.size() < cap`. That is, at least one of the nodes in L has available slots.

1. Find the node r containing the element with index i.
2. If `cap = 0`, then we must be inserting at logical index `i = 0`. Create a new node with an array of length one and put X at offset 0. See Figure 4.
3. If `cap > 0` and `i ≤ L.size() - 1`, there are three possibilities:
   (a) If node r is not full, shift the elements of r with logical index i or greater up by one and insert X in the vacated slot. See Figure 5.

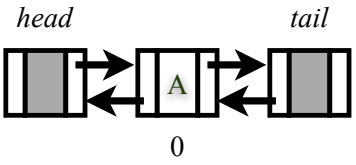

Figure 4: The result of inserting A into an empty list.

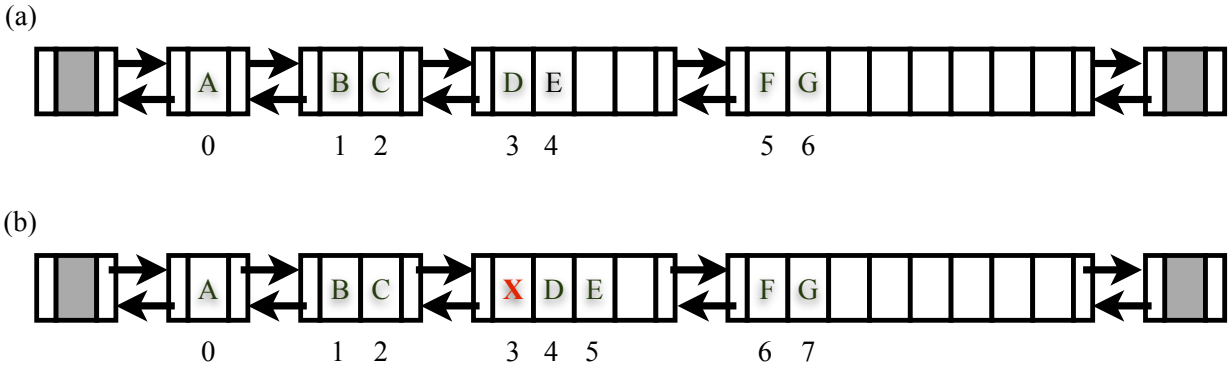

Figure 5: (a) Original list. (b) After inserting X at index 3.

(b) If node r is full and it has a predecessor with available space, do a leftward shift at logical index i and insert X in the vacated slot in r. See Figure 6.

(c) If node r and all its predecessors are full, then r must have a successor with available space (since we are assuming that `L.size() < cap`). Do a rightward shift at logical index i and insert X in the vacated slot in r. See Figure 7.

4. If `cap > 0` and `i = L.size()`, find the last non-empty node ℓ in the list. Observe that the properties of doubling lists imply that ℓ is either the last or next-to-last node in L – otherwise, we would have `L.size()` $\leq 2^{k-2} - 1$, which is not allowed.

   (a) If node ℓ is not full, put X in the first available slot of ℓ. See Figure 8(a–b).

   (b) If node ℓ is full, but one of ℓ's predecessors has space, do a leftward shift at i and put X in the newly-vacated slot in node ℓ. See Figure 8(c–d).

   (c) If node ℓ and all its predecessors are full, the successor of ℓ must be empty (since we are assuming that `L.size() < cap`). Put X in the first available slot in the successor of ℓ. See Figure 8(e).

Case 2. `L.size() = cap`. That is, all the arrays in the list are full. Proceed as follows.

1. Add a $(k+1)^{st}$ node to the end of L, with the capacity to hold 2^k elements.
2. If `i = L.size()`, insert X in the first position in that node's array. See Figure 9.
3. If `i < L.size()`, perform a a rightward shift at logical index i and put X in the vacated slot. (This is like Case 1.3(c) above.)

3.3 The `remove()` Rules

Suppose we need to remove the element at logical index i from a doubling list L with k nodes. Proceed as follows:

1. Find the node r and the offset j for element with logical index i.

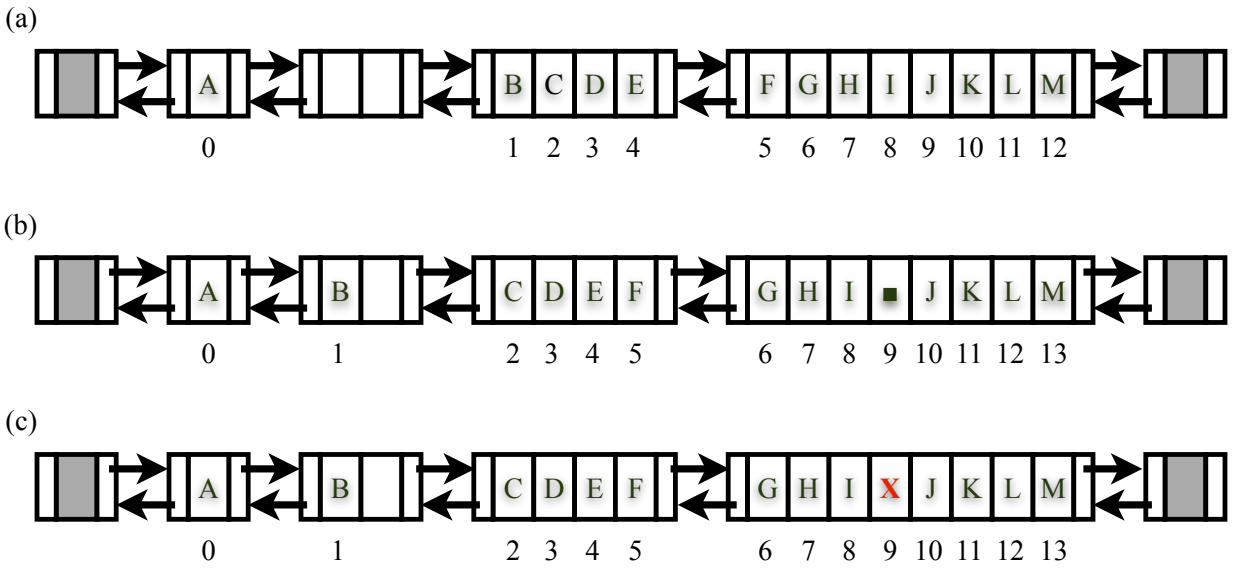

Figure 6: (a) Original list. (b) After a leftward shift at logical index 8. (c) After inserting X at index 9.

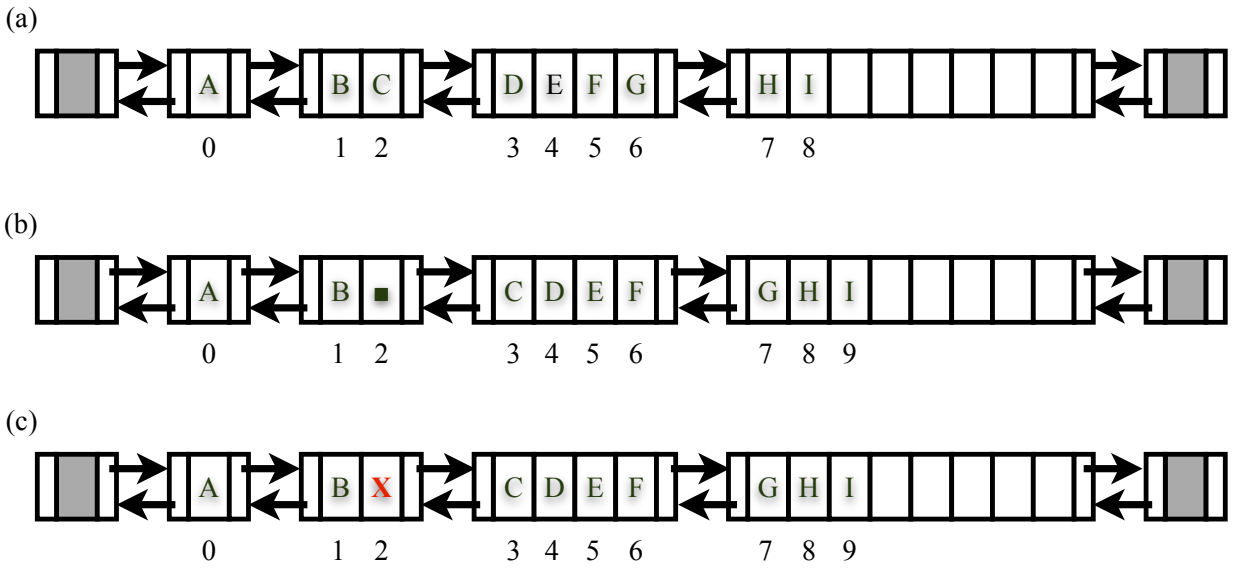

Figure 7: (a) Original list. (b) After a rightward shift at logical index 2. (c) After inserting X at index 2.

2. Save the contents of entry j of the array at r in a variable x and set the entry to null.
3. If the total number of elements in L drops to zero, replace L with an empty list and return x.
4. Shift all the elements in r with offset greater than j down by one. Make sure that the resulting empty slot in the array is set to null. See Figure 10.
5. If `L.size()` $\leq 2^{k-2} - 1$, replace L by another doubling list L' such that
   - L' has $k - 1$ nodes (numbered 0 through $k - 2$),
   - L' contains the same elements as L, in the same order, and
   - all elements of L' are stored in nodes 0 through $k - 3$. Thus, the last node is completely empty.

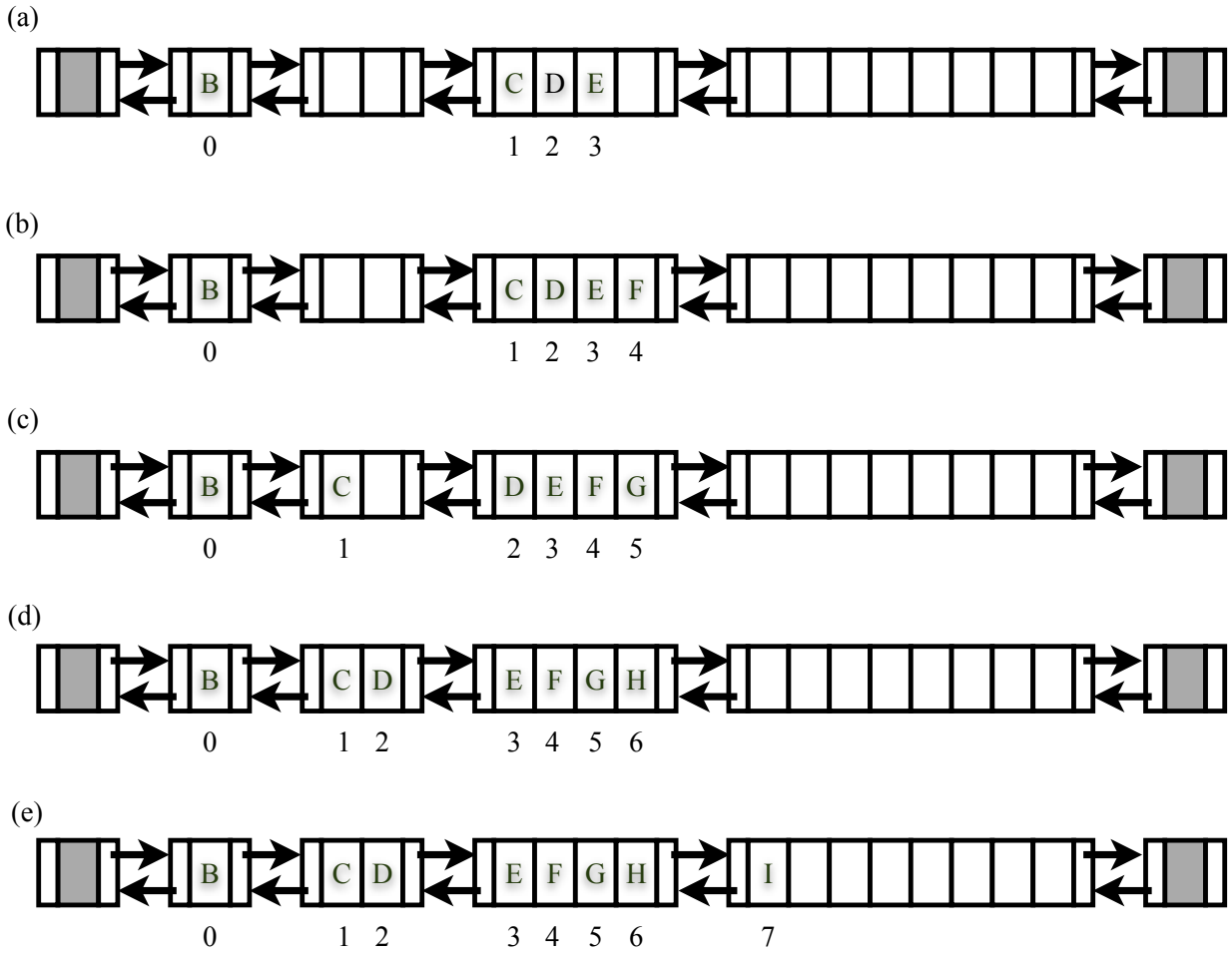

Figure 8: (a) Original list. (b) After inserting F at index 4. (c) After inserting G at index 5. (d) After inserting H at index 6. (e) After inserting I at index 7.

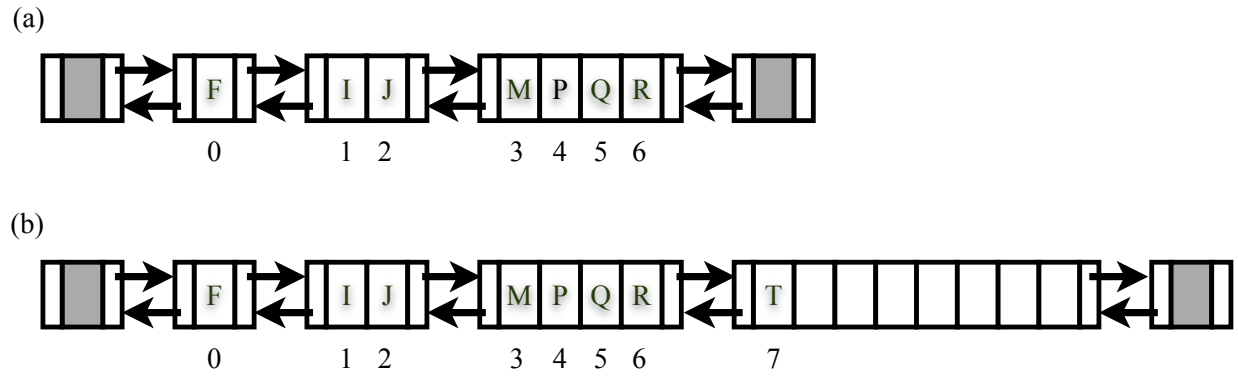

Figure 9: Insertion with list expansion. (a) Original list ($k = 3$), with no empty slots. (b) To insert item T, we add a new node of capacity $2^k = 8$, and put T at the first position of the new node's array.

See Figure 11. We refer to this step as **list compaction**. Notice that after compaction the last node contains no elements, while all the previous ones are used to full capacity.

6. Return x.

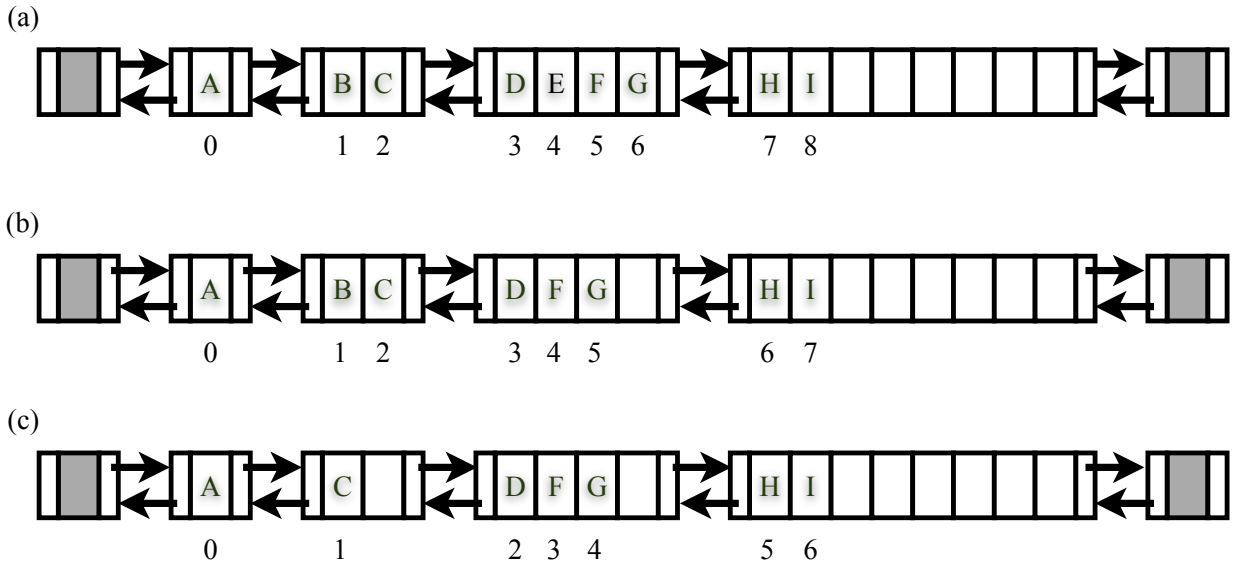

Figure 10: (a) Original list. (b) After deleting the element with index 4 (E). (c) After deleting the element with index 1 (B).

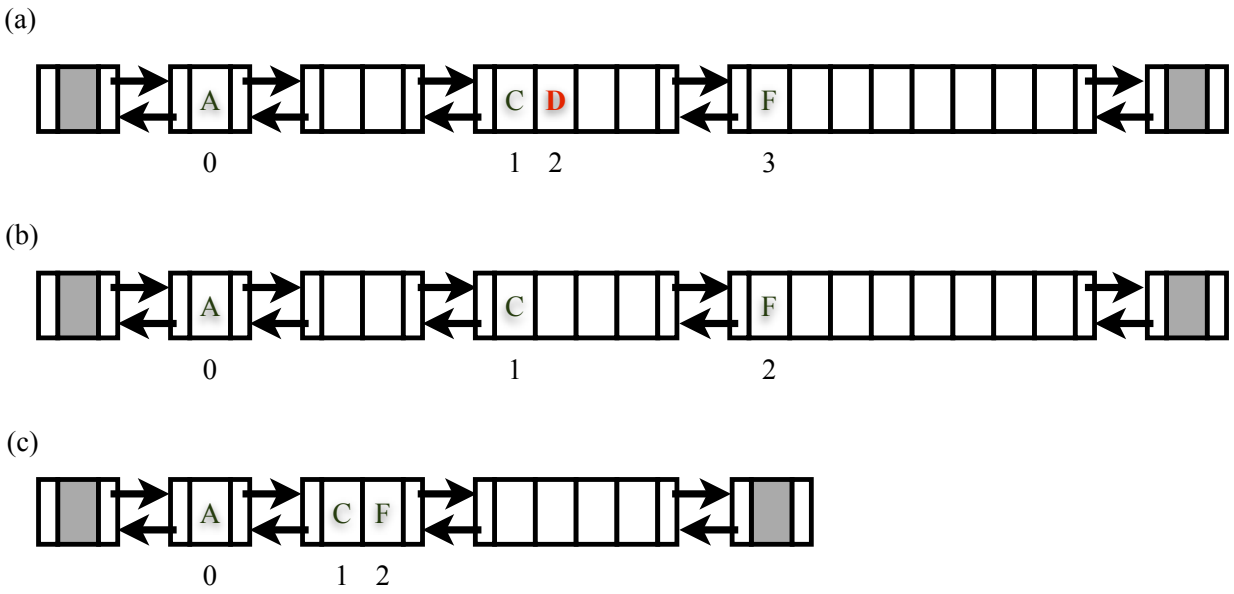

Figure 11: (a) Original list. (b) After removing D, the size drops to $2^{k-2} - 1$, so the list must be compacted. (c) After compaction.

3.4 Iterators

You must provide a complete implementation of an inner class `DoublingIterator` implementing `Iterator<E>` and a class `DoublingListIterator` implementing `ListIterator<E>`. `DoublingIterator` does not need to implement `remove()` (i.e., it can `throw UnsupportedOperationException`).

Optionally, if you are confident that your `DoublingListIterator` is correct, you can return an instance of `DoublingListIterator` from your `iterator()` method and forget about `DoublingIterator`. However, you are encouraged to keep them separate when you first start development, since the basic

one-directional Iterator, without a `remove()` operation, is relatively simple, while the `add()` and `remove()` operations for the full `ListIterator` are tricky.

3.5 The `toStringInternal` methods

To help you in debugging, you need to write `toStringInternal` methods that show the internal structure of the nodes. There are two versions. The first takes no arguments. For example, if the list of Figure 1 contained String objects A, B, C, D and E, an invocation of toStringInternal() would return the string

[(A),(B, –), (C, D, E, –), (F, G, H, –, –, –, –, –)]

where the elements are displayed by invoking their own `toString()` methods and empty cells in the array inside each node (which should always be null) are displayed as "-".

The second version of `toStringInternal` takes a `ListIterator` argument and will show the cursor position as a character | according to the `nextIndex()` method of the iterator. For example, if `iter` is a `ListIterator` for the list above and `iter.nextIndex() = 3`, the invocation `toStringInternal(iter)` would return the following string.

[(A),(B, –), (C, | D, E, –), (F, G, H, –, –, –, –, –)]

Note that, normally, methods such as `toStringInternal` would not be public, since they reveal implementation details. To simplify unit testing, you can make them public.

4 Suggestions for Getting Started

1. Begin by implementing `add(E item)`, which adds an element at the end of the list. You can use `toStringInternal()` to check your work.

2. Implement `hasNext()` and `next()` for `DoublingIterator` and implement the `iterator()` method. At this point the `List` methods `contains(Object)` and `toString()` should work.

3. Start `DoublingListIterator`.

   Implement `ListIterator` methods `nextIndex()`, `previousIndex()`, `hasPrevious()`, and `previous()`.

   Implement the `listIterator()` method.

   You should then be able to iterate forward and backward, and you can check your iterator position using `toStringInternal(iter)`.

   The `indexOf(Object obj)` method of `List` should work now.

4. Implement the `set()` method of `DoublingListIterator`.

   You will need to keep track of whether to act on the element before or after the cursor (and possibly throw an `IllegalStateException)`.

5. Implement a helper method for finding indices described in Section 3.1.

   Then you can easily implement `listIterator(int pos)`. After that, the `get(int pos)` method of `List` should work.

6. Implement `add(int pos, E item)`. Now you will need to carefully review the rules for adding elements from Section 3.2. To keep things organized, it might be useful to write helper methods to perform leftward and rightward shifts at a given index `pos`.

7. Implement the `remove(int pos)` method. You will need to carefully review the rules for removing elements from Section 3.3.

8. Implement the `add()` method of `listIterator`. The helper methods from (6) will be useful here. Be careful to update the logical cursor position to the element after the one that was added.

9. Implement the `remove()` method of `listIterator`. Remember that you must update the cursor position differently depending on whether you are removing ahead of the cursor or behind the cursor, and depending on whether deletion triggers list compaction.

10. After the unit tests, create a driver class to fully test all of the functionalities. For example, you can create a DoublingList of Strings, add multiple strings into the list, print them out, then add specific strings at given positions, print the list, then remove elements from the list and print out the list each time.

5 Submission

Inside your project folder, create a pure text file called README.txt. You can install Sublime Text software for writing this file. In this file,

1. write down any problems that you encountered and how you solved these problems;

2. list any functionalities that you were not able to successfully implement;

3. your experience of working on this problem, what did you learn?

Compress your project folder as a zip file named `Firstname_Lastname_P2.zip` containing all your source code for the project, including:

- your README.txt file,
- implementation of the required classes, and
- unit tests.

Note that your source code should have proper comments, including class header comments, method header comments, and inline comments if necessary. All the given examples in the figures should be tested.

For grading and testing purposes, you need to use exactly the same name for the classes and methods as described in this document. Failing to use the same name may affect your grade.

Grading:

- Program compiles: required for a grade
- Poorly organized code: upto -2 points

- Poor comments: upto −2 points
  Make sure that your projects are commented properly. This includes file headers, method headers, and appropriate inline comments.
- README file: 1 point
- Completeness of the classes' unit-testing: 2 points
- Correctness of your classes: 7 points