ArrayLists and Linked Lists

Sujit Chakrabarti

1 Why Collections?

There are situations in programming where the data you are interested in represents collection of smaller things. Consider the institute information management system. We create one object of the `Student` class corresponding to each student. In order to implement any functionality that requires the entire collection of students to be available at once, we need to keep these objects in a single place. One example is `printStudentList` method which prints all the students. Another example could be `search` method.

From what have learnt so far, we could use arrays for this purpose as shown in fig. 1.

```
public class WithArray {

  public static void main(String[] args) {
    String studentList[] = {"Sujit", "Siddharth", "Karanpreet"};
    printStudentList(studentList);
  }

  public static void printStudentList(String[] students) {
    for(String s : students) {
      System.out.println(s);
    }

  }
}

/*
  SUMMARY
  =======
  1. Collection of students can be represented as an array
*/
```

Figure 1: Representing List of Students using Java arrays

Arrays have a few advantages, namely they are simple to create and it's easy to access elements within an array. However, there's a big disadvantage. Once an array object is created, you can't add anything to it, nor can you

delete any element from it. And this is a serious problem for the Institute Information Management System we have been building. For example, if you wish to enrol a student, you would like to add this student into the array. But Java arrays don't allow you to do this: once you have created an array, you are allowed to modify its elements, but you can't expand or shrink it.

Can we have an array which allows us the functionality of arrays along with the flexibility to expand or shrink it as required. Yes, we can do this using Java lists.

In this section, we will learn about two types of lists: `ArrayList`, and `LinkedList`. Apart from learning how to use them, we also discuss how one can often be used in place of the other rather seamlessly. We also discuss the various trade-offs between the two types of lists.

In summary:

1. This module is an introduction to the use of data-structures as a fundamental mechanism for implementing robust and efficient programs.

2. We get to discuss – although briefly and indirectly – how such data-structures are typically implemented in an object-oriented language like Java.

3. Finally, this segment is a gentle primer to the much larger and very important topic of data-structures and algorithms, for which we have reserved an entire module of the course.

```
import java.util.*;

public class WithArrayList {

  public static void main(String[] args) {
    ArrayList studentList = new ArrayList();
    studentList.add(new Student("Sujit", 1));
    studentList.add(new Student("Siddharth", 2));
    studentList.add(new Student("Karanpreet", 3));
    // studentList.add(new String("Hari")); // this would lead to runtime type error
    WithArrayList.printStudentList(studentList);
  }

  public static void printStudentList(ArrayList students) {
    for(Object s : students) {
      Student st = (Student)s;
      System.out.println(st.getDetails());
    }
  }
}

class Student {
  private final String name;
  private final int rollNumber;

  public Student(String name, int rollNumber) {
    this.name = name;
    this.rollNumber = rollNumber;
  }

  public String getDetails() {
    return
      "name = " + this.name + '\n' +
      "roll number = " + this.rollNumber + '\n';
  }
}

/*
  SUMMARY
  =======
  1. ArrayList Simple way to representation a dataset which is a collection of elementary data it
      This is a type-unsafe approach of creating containers. We can add anything into the List wh
      result in a variety of runtime type errors.
  2. When printing, the ArrayList prints the elements in the same sequence as they had been inser
      In other words, ArrayList (and any other form of lists, including arrays) remember the sequ
      of insertion. This is an important property which can be safely exploited in the program de
      There are other containers, e.g. Sets and Maps, where the sequence of insertion is of no
      consequence. A program using such containers must not depend on them to remember the sequen
      insertion for its correctness.

  3. Using a non-generic ArrayList is type-unsafe.
*/
```

Figure 2: Representing List of Students using Java `ArrayList` (code/WithArrayList.java)

One possible way of storing the list of students is shown in fig. 2. And one of the advantages of this type is immediately evident here: Note that

we are adding elements to this list one after another. This is equivalent to being able to enrol students as and when we wish. As we will show shortly, it's equally easy to remove a student from the rolls.

2 Generics

One of the issues with the code in fig. 2 is about *type safety*. A type-safe program is one which doesn't or can't encounter a runtime type error. The `ArrayList` we have used can have objects of any class which is a sub-type of `Object`. As a result, we are forced to use type-casting in printStudentList method to obtain a `Student` from an `Object`. This is essential so that we can access the `getDetails` method of the `Student` class. Such type-casting, although unavoidable in certain rare circumstances, is an inherently unsafe thing to do as they may lead to runtime type errors.

To understand this, consider the line which has been commented out in the `main` method:

```
studentList.add(new String("Hari"));
```

If this code is uncommented, this will add a string to the `studentList` in which all other elements happen to be of `Student` type. The result of running this code will as follows:

```
name = Sujit
roll number = 1

name = Siddharth
roll number = 2

name = Karanpreet
roll number = 3

Exception in thread "main" java.lang.ClassCastException:
  java.lang.String cannot be cast to Student
at WithArrayList.printStudentList(WithArrayList.java:16)
at WithArrayList.main(WithArrayList.java:11)
```

This output is not surprising, and demonstrates a vulnerability of the given code which comes inherently with use of `ArrayList` of `Object`s. The solution to this problem lies in being able to make the type of the `ArrayList` more specific: to be able to say that it should an `ArrayList` that is allow to have elements of `Student` type and none else. Turns out, it's possible to do this, using Java generics. The modified code is shown in fig. 3.

Note the following:

1. The typecasting needed in `printStudentList` method of fig. 2 is no more needed.

```
import java.util.*;

public class WithArrayListGeneric {
  public static void main(String[] args) {
    ArrayList<Student> studentList = new ArrayList<Student>();
    studentList.add(new Student("Sujit", 1));
    studentList.add(new Student("Siddharth", 2));
    // studentList.add(new String("Hari")); // this would lead to compile error
    studentList.add(new Student("Karanpreet", 3));

    printStudentList(studentList);
  }

  public static void printStudentList(ArrayList<Student> students) {
    for(Student s : students) {
      System.out.println(s.getDetails());
    }
  }
}

class Student {
  private final String name;
  private final int rollNumber;

  public Student(String name, int rollNumber) {
    this.name = name;
    this.rollNumber = rollNumber;
  }

  public String getDetails() {
    return
      "name = " + this.name + '\n' +
      "roll number = " + this.rollNumber + '\n';
  }
}

/*
  SUMMARY
  =======
  1. ArrayList Simple way to representation a dataset which is a collection of elementary data it
  2. This is a type-safe approach of creating containers. Any attempt to add an element which is
       type-incompatible with the declared contained type will fail at the static type checking st
       resulting in a compile error.

*/
```

Figure 3: ArrayList using Java Generics (code/WithArrayListGeneric.java)

2. The commented line in `main` method, if uncommented, would lead to a compile error (instead of a runtime error as in fig. 2). This is definitely better, as Java's type-system is helping us detect a potential runtime type error at compile time.

Also note that Java arrays were already type safe as we have used them in our examples. By starting to use `ArrayList` in a non-generic way, we had lost out on that type-safety. However, by bringing in generics, we have got it back now. All container classes shipped with Java are implemented using generics. Using them is almost exactly the same as how you use `ArrayList` and `LinkedList`. We have briefly explained how to *use* these classes. How Java generics is used to implement these – and other similar – classes is a topic beyond the scope of this discussion.

3 Linked List

All that we have done in the last two pieces of code that we developed, can also be done pretty much exactly the same way with another kind of list, called the *linked list*. `LinkedList` class of `java.util` package gives us an implementation of this data-structure. The above sets of code have been developed using `LinkedList` in fig. 4 (type unsafe) and fig. 5 (type safe using generics).

4 `List` and Polymorphism

The reason why we have almost identical set of features in `LinkedList` class and `ArrayList` class is because these features are really those of the `List` class (again in `java.util` package) of which `LinkedList` class and `ArrayList` class are sub-classes. The internal implementation of `List` class of which `LinkedList` class are different which leads to different runtime performance characteristics for these two classes (more about this in sec. 6). However, both of them have nearly identical, governed by the interface of the `List` class. Let's see an example of how the `List` class, along with some clever use of polymorphism, can be used to implement some nice and reusable piece of code.

Fig. 6 shows a piece of code where we have used both `ArrayList` and `LinkedList` class. Both `studentList1` and `studentList2` are `List<String>` type. However, they are being initialised with an object of the class `ArrayList<String>` and `LinkedList<String>` respectively. Both these `List`s are passed in turn to the `printStudents` method which takes a parameter of the type `List<String>` type.

The output of running the code is as shown below:

```
import java.util.*;

public class WithLinkedList {

  public static void main(String[] args) {
    LinkedList studentList = new LinkedList();
    studentList.add("Sujit");
    studentList.add("Siddharth");
    studentList.add("Karanpreet");
    System.out.println(studentList);
  }
}

/*
  SUMMARY
  =======
  1. LinkedList Simple way to representation a dataset which is a collection of elementary data i
      This is a type-unsafe approach of creating containers. We can add anything into the List wh
      result in a variety of runtime type errors.
  2. When printing, the LinkedList prints the elements in the same sequence as they had been inse
      In other words, LinkedList (and any other form of lists, including arrays) associate each e
      with a integer index which indicates the position of the element in the list. This is an im
      property which can be safely exploited in the program design.
      There are other containers, e.g. Sets and Maps, where the position of an element is of no
      consequence. A program using such containers must not depend on them to associate any notio
      position to the elements for its correctness.
*/
```

Figure 4: `LinkedList` (`code/WithLinkedList.java`)

```
import java.util.*;

public class WithLinkedListGeneric {

  public static void main(String[] args) {
    LinkedList<String> studentList = new LinkedList<String>();
    studentList.add("Sujit");
    studentList.add("Siddharth");
    studentList.add("Karanpreet");
    System.out.println(studentList);
  }
}

/*
  SUMMARY
  =======
  1. LinkedList Simple way to representation a dataset which is a collection of elementary data i
      This is a type-safe approach of creating containers. Any attempt to add an element which is
      type-incompatible with the declared contained type will fail at the static type checking st
      resulting in a compile error.

*/
```

Figure 5: `LinkedList` using Java Generics (`code/WithLinkedListGeneric.java`)

```java
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Arrays;

public class ListPolymorphism {

  public static void main(String[] args) {
    List<String> studentList1 = new ArrayList<String>(Arrays.asList("Tricha", "Murali", "Sujit"));
    List<String> studentList2 = new LinkedList<String>(Arrays.asList("Siddharth", "Karanpreet"));

    printStudents(studentList1);
    printStudents(studentList2);
  }

  public static void printStudents(List<String> students) {
      System.out.println("Printing student list ...");
      for(int i = 0; i < students.size(); i++) {
        System.out.println("Student number " + i + " : " + students.get(i)); // correct way.
      }
  }
}

/*
  SUMMARY
  =======

  printStudents as a polymorphic function. It uses List (which is a super-class of ArrayList and
    instead of ArrayList or List, thus inter-operates smoothly with both types.
*/
```

Figure 6: `List` and Polymorphism (`code/ListPolymorphism.java`)

```
Printing student list ...
Student number 0 : Tricha
Student number 1 : Murali
Student number 2 : Sujit
Printing student list ...
Student number 0 : Siddharth
Student number 1 : Karanpreet
```

5 List Iterators

6 Choosing between the Various List Implementations

As we have seen through the various examples, what you can do with `ArrayList` is pretty much the same as what you can accomplish by using `LinkedList`. You may wonder, why then, makers of Java language have provided two different `List`s when their capabilities are so similar. The answer lies, not in functionality, but in performance.

Let's conduct a few experiments to understand this point.

In all these experiments, we create two lists `list1` and `list2`, each with a around a lakh integers. One is an `ArrayList` while `list2` is a `Linkedlist`. We then perform the same operation on both the lists, and estimate how much time it took to perform the operation in each case. Further, you will notice that we perform the operation a large number of times. That's because we wish to make the numbers significant enough to bring out the performance differences clearly. Further, doing the same thing a large number of times also flushes out any random error in the values.

6.1 Getting by Index

In this experiment (see fig. 7), we perform the `get` operation on the large element of the long lists. We get the following output:

```
array value = 49999
ArrayList took 291772 ns.
array value = 49999
Linked List took 610700 ns.
ArrayList faster by  318928 ns!
```

We observe that `ArrayList` is significantly faster. Let me tell you that this value is not precise, since it includes the time taken by many other thinks (e.g. iterating through the loop etc.). Also, there are external factors associated with the computer itself (e.g. multiprocessing, cache-misses and page faults etc.)which have an influence on the exact amount of time needed to perform

```
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;

public class LargeListGet {

  public static void main(String[] args) {
    // creating ArrayList
    List<Integer> list1 = new ArrayList<Integer>();
    for(int i = 0; i < 100000; i++) {
      list1.add(i);
    }

    // creating LinkedList
    List<Integer> list2 = new LinkedList<Integer>();
    for(int i = 0; i < 100000; i++) {
      list2.add(i);
    }

   // measuring ArrayList performance
   final long start1 = System.nanoTime();

    // action performed
    System.out.println("array value = " + list1.get(99999/2));

    final long end1 = System.nanoTime();
    final long time1 = end1 - start1;
    System.out.println("ArrayList took " + time1 + " ns.");


   // estimating LinkedList performance
    final long start2 = System.nanoTime();

    // action performed
    System.out.println("array value = " + list2.get(99999/2));

    final long end2 = System.nanoTime();
    final long time2 = end2 - start2;
    System.out.println("Linked List took " + time2 + " ns.");

    System.out.println("ArrayList faster by  " + (time2 - time1) + " ns!");
  }
}

/*
  SUMMARY
  =======

*/
```

Figure 7: getting elements from anywhere (`code/LargeListGet.java`)

the task. However, this still give a fair estimate about which of the two elements is faster. So, it can be inferred that:

> `ArrayList`s are faster than `LinkedList`s when it comes to reading the values of elements at arbitrary positions in the list.

6.2 Adding at the End

6.3 Adding Anywhere

On running the code in fig. 9, we get the following output:

```
ArrayList took 1610172712 ns.
LinkedList took 4834461 ns.
Linked Lists faster by a factor of 333!
```

We see that linked lists are faster, not by a smaller factor, several hundred times. If we try removing elements from somewhere in the beginning of the list, you would see similar results.

The inference from this experiment is:

> `LinkedList`s are significantly faster when elements are added/removed from arbitrary positions.

6.4 Explanation

If you are curious to know why the observations are they way they are in the experiments just shown, you would have to delve a bit into the internal details of how these `List`s – `ArrayList` and `LinkedList` – are built and how they work.

6.4.1 `ArrayList`

The `ArrayList` class uses a Java array internally. However, depending on how many elements are currently stored in the array, the capacity of the array keeps changing. Therefore, `ArrayList` works with two important attributes:

1. size. The number of elements currently stored in the array.
2. capacity. The actual size of the array.

Of course, at any point the capacity must always be greater than or equal to the size. Let's call the ratio *size/capacity* as the *loading factor* L of the `ArrayList`.

```
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;

public class LargeListAddEnd {

  public static void main(String[] args) {
    List<Integer> list1 = new ArrayList<Integer>();
    for(int i = 0; i < 100000; i++) {
      list1.add(i);
    }
    final long start1 = System.nanoTime();
    list1.add(1);

//    for(int i = 0; i < 100000; i++) {
//      list1.add(i);
//    }
    final long end1 = System.nanoTime();
    final long time1 = end1 - start1;
    System.out.println("ArrayList took " + time1 + " ns.");

    List<Integer> list2 = new LinkedList<Integer>();
    for(int i = 0; i < 100000; i++) {
      list2.add(i);
    }
    final long start2 = System.nanoTime();
    list2.add(1);
//    for(int i = 0; i < 100000; i++) {
//      list2.add(i);
//    }
    final long end2 = System.nanoTime();
    final long time2 = end2 - start2;
    System.out.println("LinkedList took " + time2 + " ns.");

    System.out.println("Arrays faster by " + (time2 - time1) + "!");
  }
 }

/*
  SUMMARY
  =======

*/
```

Figure 8: `add`ing elements at the tail end of the list (`code/LargeListAddEnd.java`)

```java
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;

public class LargeListAdd {

  public static void main(String[] args) {
    List<Integer> list1 = new ArrayList<Integer>();
    for(int i = 0; i < 100000; i++) {
      list1.add(i);
    }
    List<Integer> list2 = new LinkedList<Integer>();
    for(int i = 0; i < 100000; i++) {
      list2.add(i);
    }

    // estimating ArrayList performance
    final long start1 = System.nanoTime();
    for(int i = 0; i < 100000; i++) {
      list1.add(0, i);
    }
    final long end1 = System.nanoTime();
    final long time1 = end1 - start1;


    // estimating LinkedList performance
    final long start2 = System.nanoTime();
    for(int i = 0; i < 100000; i++) {
      list2.add(0, i);
    }
    final long end2 = System.nanoTime();
    final long time2 = end2 - start2;

    // printing result
    System.out.println("ArrayList took " + time1 + " ns.");
    System.out.println("LinkedList took " + time2 + " ns.");
    System.out.println("Linked Lists faster by a factor of " + time1/time2 + "!");
  }
 }

/*
  SUMMARY
  =======

*/
```

Figure 9: `add`ing anywhere (`code/LargeListAdd.java`)

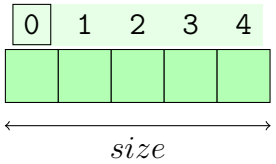

Figure 10: ArrayList: Schematic

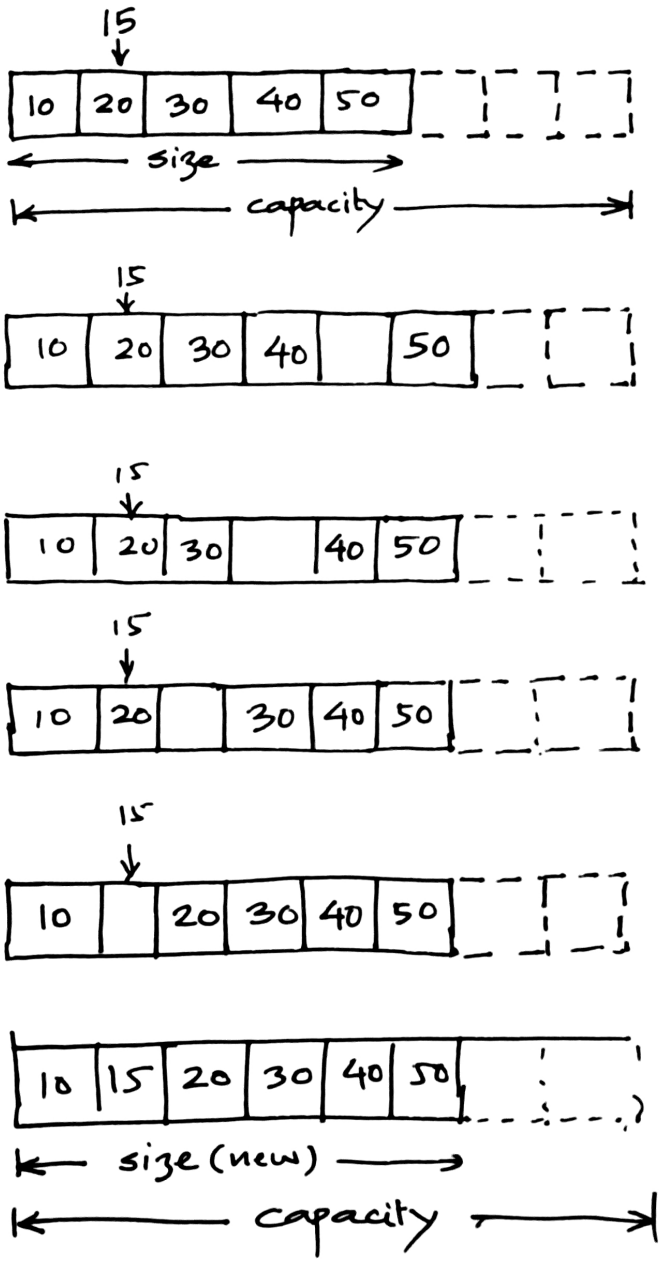

Figure 11: ArrayList: Adding and element at arbitrary location

To begin with, `ArrayList` starts off with an array A_1 as the internal store with some appropriate initial value of capacity. As elements get added, A_1 starts getting filled up. When it is close to full, indicating by $L > \tau_1$ where tau_1 is some threshold value, a new array A_2 is created with another larger capacity and all the elements in the original array are copied to the new array. This point on, the new array A_2 is used as the internal store; A_1 is discarded.

Likewise, whenever elements start getting removed, loading factor L starts dropping. When $L < \tau_2$, a new array A_3 is created with another smaller capacity and all the elements in the original array are copied to A_3. This point on, the new array A_3 is used as the internal store; A_2 is discarded.

Let's call the above events: copying A_1 to A_2, A_2 to A_3 and so on as *array copy*. `ArrayList` tries to keep the value of L reasonably close to 1, so that the amount of wasted space is minimised (remember that at any point $capacity - size$ cells of the array are unused). On the contrary, keeping it too close to 1 would result in more frequent array copies, hitting the runtime performance of the data structure. Further, it is typically a good idea to keep $\tau_1 \neq \tau_2$ to avoid creating *thrashing points*. If $\tau_1 = \tau_2$, then `add`s and `remove`s around these thrashing points would start getting prohibitively expensive.

Anyway, the above are some design considerations useful for implementing a data-structure like `ArrayList`.

How does `ArrayList` do when `add`s happen at the right end? Well, they are lightning fast: almost as fast as writing into a cell of an array. Similarly, for `remove`s done at the right end. Of course, if the `add` causes L to cross τ_1 (in case of `add`)or τ_2 (in case of `remove`), it will result in an array copy. That's expensive, but with well-chosen values of τ_1 and τ_2, that should be fairly rare.

But what happens when you `add` or `remove` from close to the left end of the `ArrayList`? Well, then the things aren't as rosy as before. Let's say, you have 5 elements in your array (indices 0, 1, 2, 3 and 4), and you are `add`ing an element at index 1. The cell at index 1 is occupied by another element. So, it must be shifted rightward (into index 2). But index 2 is also occupied by another element. So, that needs to shifted rightward to index 3. So, you get the drift, right? All the elements to the right of the index at which you wish to make the addition must be moved one place to their right. The process must begin from the right end. How many steps will the process take? In our example, 4 elements need to be shifted. In general, whenever you are `add`ing to an `ArrayList` close to its left end, or to any random position for that matter, the number of elements needed to be shifted this way would be something proportional to the *size* of the array. This is undoubtedly a fairly expensive thing to do! If your computation requirement involves lots of `add`s and `remove`s at arbitrary indices of the list, `ArrayList` would turn out to be a poor choice

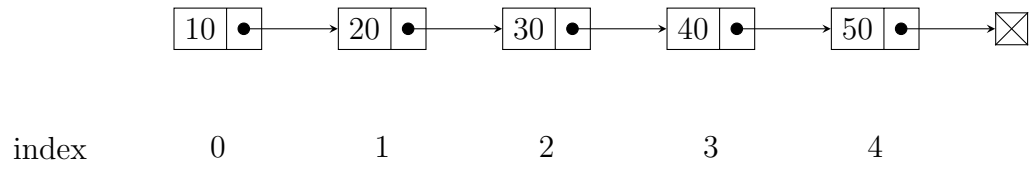

Figure 12: Linked List: Schematic representation

because of this.

6.4.2 `LinkedList`

`LinkedList` class is an implementation of the linked list data structure. This helps overcome the following drawbacks of an `ArrayList`:

1. Array copies are expensive even though rare. They do not affect the average execution time of additions into and removals of elements from the array, the worst case time gets severely affected due to that. This may sometimely be unacceptable in certain cases, e.g. real-time systems where worst case execution times are as important as average case.

2. Finally, additions into and removal from arbitrary positions in the list is very expensive, even in the average case, for `ArrayList`s. This becomes the primary reason why `LinkedList` is sometimes the list of choice.

A linked list arranged its elements into dynamically created two cell nodes. The first cell of each node called the *value* contains the value of the element; the second cell contains the reference to the next node, and hence is called the *next* cell. The rightmost node's *next* points to a *null* or *nil* address, indicating that there's nothing to the right of this.

How do we add to a linked list. Let's say that you again have a linked list with 5 elements, and you are adding at the index 1.

1. Firstly, a new node N is created. The *value* field of this node is the value you want to insert.

2. Its *next* field is made to point to the node at index 2 (note that the reference to this node is available in the *next* cell of the node N_1 at index 0).

3. Finally, the *next* field of the node in index 0, which has been pointing to the N_2 (index 1 so far, and index 2 here on), will now be redirected to point to N.

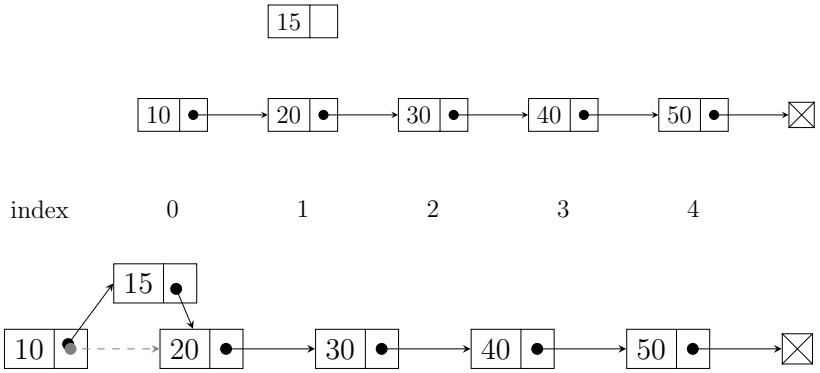

Figure 13: Linked List: Adding an element

Done in 3 simple steps, regardless of where in the list we are making the addition.

Similarly, if you are removing from an arbitrary position, it's even simpler. Let's say, you wish to remove from index 1. Let's say that the nodes on indices 0, 1 and 2 are N_0, N_1 and N_2 respectively. All you need to do, to remove N_0 is to redirect the *next* field of N_0 (which has been hitherto pointing to N_1) to point to N_2.

Due to the above property of constant time additions and removals, linked lists do very well in scenarios that involve repeated additions and removals at arbitrary index values in the list.

But all isn't hunky dory with `LikedList`s either, for then, why would anyone have needed `ArrayList`s. Consider doing a `get`. In `ArrayList`, it's just as far as accessing/reading an element from a Java array, irrespective of where in the list we are reading from. Unfortunately, with linked lists, it's not so easy. The only way to reach an element in a linked list is by starting with the first element, and hopping through the links one by one till we reach the desired index. Clearly, to `get` the element at index n will need n hops. And that's again expensive – very expensive – compared to `get`ting from an `ArrayList`. Hence, in scenarios which require repeated `get`s or reads from arbitrary locations of the list, linked lists fare poorly as compared to array lists.

6.5 Doubly Linked List

There's an interesting quirk about the linked list implementation that we have introduced in this section. Operations done close to its left end are

faster, while those done closer to its right end are slower [1]. For example, `l.get(0)` (`get` at the extreme left or start of the list) will take a lot less time than `l.get(l.size() - 1)` (`get` done at the extreme right or far end of the list). This behaviour is assymmetric. Another example of assymmetric behaviour is in the performance of the forward and backward iterator. The forward iterator (moving from the left to the right end, or from the beginning of the list to the end of it) will perform must faster than the backward iterator (moving from the right end to the left, or from the end to the beginning of the list). This assymmetric behaviour is often not desirable. To counter this effect, we have another variant of the linked list, called the *doubly linked list*, we has a symmetric behaviour. The operations would typically take a time proportional to how far they are from either end, which end – doesn't matter! In contrast, the more precious name for the linked list type we studied just now is *singly linked list*.

In a doubly linked list, there are 3 – instead of 2 – fields: the *value* field, the *next* field and additionally, the *prev* (standing for previous) field. Each node thus has a provision to not just point to the next node, but also the previous one. Traversal of the list may begin from any of the two ends: begin from the left end and proceed rightwards by following the *next* field of each node; or begin from the right end and proceed leftwards by hopping from node to node using the *prev* field.

An `add` at any position would be very similar to how it's done for a singly linked list, except for some book-keeping associated with the *prev* field. For example, let's consider the earlier example of a linked list with 5 elements. We wish to add an element at index 1. Let's also say that the node at index 0 is L_0, and that at index 1 is L_1. The addition will involve the following steps:

1. Create a node N.
2. Make the *next* field of N_0 point to N (instead of N_1). Make the *next* field of N point to N_1.
3. Make the *prev* field of N_1 point to N (instead of N_0). Make the *prev* field of N point to N_0.

Similarly, removal of an element will involve the exact opposite step to take N out of the picture.

[1] Please note that, for the data structure, in reality there's no left or right end. We are calling one of the ends as the left end (the first node of the list) and the other extreme as the right end simply because we have chosen to draw the linked list on paper that way.

So, there we go! We an addition of another field *prev* to our nodes we are able to have a doubly linked list, that shows a symmetric performance both ways.

When do you choose a doubly linked list? Whenever you need to traverse in randomly directions, it's better to go with a doubly linked list. DLLs come with a small cost in terms of space. But if you know a priori which way you would always want to traverse the list – as happens quite often – it mayn't be worth your while to add another field to each node as in DLL, which makes them a bit heavier than in SLL.

In Java, the specification of `LinkedList` class doesn't specify which type of linked list it is. So, in general, the distinction is non of our concern. But again, there are cases where these choices may have serious performance implications. In such cases, you would probably like to check out which of the two types of linked lists has your language vendor provided you with: SLL or DLL?

7 Summary

In this section, we studied two list data-structures, namely `ArrayList` and `LinkedList`. We saw that they are very similar in terms of features, but differ in performance for different types of operations. Therefore, the choice of which of the two implementations to use depends on which of the operations are more likely to be done on the list. For example:

1. Consider a message queue. Typically messages will be added to its end. Any message will be read at arbitrary positions. Clearly, `ArrayList` appears to be a better choice.

2. Consider a list of students which needs to be kept sorted by roll numbers. Such a list will involve many updates at arbitrary positions in the list. A `LinkedList` appears to be a better choice.

This discussion was with the intent of giving you a primer to how you pick and choose between two functionally equivalent implementation of a data-structure based on their performance. Mastering this skill has significant implications on how your real software systems will performance when deployed. For example, an appropriately chosen data-structure placed at the heart of your enterprise server may bump up the performance of your server by several factors of magnitude.

This topic is therefore so central to the theme of software engineering that we have an entire module in this programme devoted to this very topic on *Data Structures and Algorithms*.