Linked List

What is a Linked List?

A linked list is a common data structure that is made of a chain of nodes. Each node contains a value and a pointer to the next node in the chain.

The head pointer points to the first node, and the last element of the list points to null. When the list is empty, the head pointer points to null.

Linked lists can dynamically increase in size. It is easy to insert and delete from a linked list because unlike arrays, as we only need to change the pointers of the previous element and the next element to insert or delete an element.

Some important applications of Linked Lists include:

- Implementing HashMaps, File System and Adjacency Lists
- Dynamic memory allocation: use linked lists of free blocks
- Performing arithmetic operations on long integers
- Maintaining a directory of names

Types of linked lists

Since a linked list is a linear data structure, meaning that the elements are not stored at contiguous locations, it’s necessary to have different types of linked lists to access and modify our elements accordingly.

There are a three different types of linked lists that serve different purposes for organizing our code.

Let’s take a look.

Singly linked list (Uni-directional)

A singly linked list is unidirectional, meaning that it can be traversed in only one direction from head to the last node (tail). Some common operations for singly linked lists are:

Doubly linked list (Bi-directional)

Doubly linked lists (DLLs) are an extension of basic linked lists, but they contain a pointer to the next node as well as the previous node. This ensures that the list can be traversed in both directions. A DLL node has three fundamental members:

- The data
- Pointer to the next node
- Pointer to the previous node

Circular linked list

Circular linked lists function circularly: the first element points to the last element, and the last element points to the first. A single linked list and double linked list can be made into a circular linked list. The most important operations for a circular linked list are:

- insert − insert elements at the start of the list
- display − display the list
- delete − delete elements from the start of the list

Structure of a Singly Linked List

In Java, the linked list class is an ordered collection that contains many objects of the same type. Data in a Linked List is stored in a sequence of containers. The list holds a reference to the first container and each container has a link to the next one in the sequence.

Linked lists in Java implement the abstract list interface and inherit various constructors and methods from it. This sequential data structure can be used as a list, stack or queue.

As I briefly discussed before, a linked list is formed by nodes that are linked together like a chain. Each node holds data, along with a pointer to the next node in the list.

The following illustration shows the theory of a Singly Linked List.

To implement a linked list, we need the following two classes:

Class Node

The Node class stores data in a single node. It can store primitive data such as integers and string as well as complex objects having multiple attributes.

Along with data, it also stores a pointer to the next element in the list, which helps in linking the nodes together like a chain.

Here’s a typical definition of a Node class:

```
//Class node having Generic data-type <T>

public class Node<T> {

  public T data; //Data to store (could be int, string, Object etc)

  public Node nextNode; //Pointer to next node in list

}
```

Class Linked list

As mentioned above, the Singly Linked list is made up of nodes that are linked together like a chain. Now to access this chain, we would need a pointer that keeps track of the first element of the list.

As long as we have information about the first element, we can traverse the rest of the list without worrying about memorizing their storage locations.

The Singly Linked List contains a head node: a pointer pointing to the first element of the list. Whenever we want to traverse the list, we can do so by using this head node.

Below is a basic structure of the Singly Linked List’s class:

```
public class SinglyLinkedList<T> {

    //Node inner class for SLL

    public class Node{

        public T data; //Data to store (could be int, string, Object etc)

        public Node nextNode; //Pointer to next node in list



    }

  public Node headNode; //head node of the linked list

    public int size;      //size of the list
```

How to create and use a Linked List

Linked lists are fairly easy to use since they follow a linear structure. They are quite similar to arrays, but linked lists are not as static, since each element is its own object. Here is the declaration for Java Linked List class:

```
public class LinkedList<E>

extends AbstractSequentialList<E>

implements List<E>, Deque<E>, Cloneable, Serializable
```

Let’s see a more detailed example of that in code. Here is how we create a basic linked list in Java:

```
import java.util.LinkedList;

class Main {

  public static void main(String[] args) {

    LinkedList<String> names = new LinkedList<String>();
```

```
 }
}
```

The Linked List class is included in the Java.util package. In order to use the class, we need to import the package in our code. We have initialized an empty Linked List using the new keyword. A Linked List can hold any type of object, including null.

Adding Elements to a Linked List

In order to add an element to the list, we can use the .add() method. This method takes an element (passed as an argument) and appends it to the end of the list.

```
import java.util.LinkedList;

class Main {

  public static void main(String[] args) {

    LinkedList<String> names = new LinkedList<String>();

    names.add("Brian");

    names.add("June");

    System.out.println(names); // This will output [Brian, June]

  }

}
```

Run

If you want to add the new element to a specific location instead, you can do so by passing the index value as the first argument to the .add() method.

```
names.add(1, "Kathy");

System.out.println(names);

// Outputs [Brian, Kathy, June]
```

The above line of code inserts “Kathy” into the names list at the 1 position or index. Since the first element in the list has the index 0, “Kathy” will be inserted right after “Brian” and just before “June”. This will feel familiar if you’ve worked with Arrays in JavaScript. This behavior is possible because Linked Lists are indexed like JavaScript arrays.

There are also methods for explicitly adding elements to the end or start of the list.

```
names.addFirst("Luke");

names.addLast("Harry");

System.out.println(names);

// Outputs [Luke, Brian, Kathy, June, Harry]
```

The .addFirst() method adds the specified element at the start of the list. To append an element at the list’s end position, use the .addLast() method. In the code block above, “Luke” is inserted into the list and it becomes the first element in the list (it now has the index 0). The element “Harry” is inserted at the end, making it the last element on the list.

Keep the learning going.

Learn data structures for Java coding interviews without scrubbing through videos or documentation. Educative’s text-based courses are easy to skim and feature live coding environments, making learning quick and efficient.

Data Structures for Coding Interviews in Java

Removing Elements from a LinkedList

Similar to element addition, Linked List provides methods for removing elements in a list. These methods are similar in operation to the methods for adding elements to the list. The .remove() method removes the first occurrence of a specified element.

```
names.remove("Brian"); // This will remove the first occurrence of "Brian" in the LinkedList
```

This method is similar to the .add() method, as it allows the removal of an element at a specific index. Calling names.remove(2) will remove the element at index 2, which is “Brian” in this list. It is also possible to remove the first element and the last element in the list using the .removeFirst() and .removeLast() methods respectively.

```
names.removeFirst();

names.removeLast();
```

Updating Elements in a LinkedList

The Linked List class provides a method to change an element in a list. This method is called .set(), and it takes an index and the element which needs to be inserted, replacing the previous element at that position.

```
// names list is: [Kathy, June]

names.set(0, "Katherine");

// names list is: [Katherine, June]
```

Iterating Over a LinkedList

There are a couple of methods for iterating over the elements in a LinkedList. In the example below, we are using a for loop and the .get() method of a Linked List.

```
for (int i = 0; i < names.size(); i++) {

  System.out.println(names.get(i));

}
```

We could also use a foreach loop to iterate over a Linked List.

```
for (String str : names) {

  System.out.println(str);

}
```

What to consider before using a Linked List

A linked list acts as a dynamic array. This means we do not have to specify the size when creating it, its size automatically changes when we add and remove elements. The Linked List class is also implemented using the doubly linked list data structure.

This means that each element in the list holds a reference to elements before and after it. If an element is the last in the list, its next reference will return null.

This design makes the Linked List useful in cases where:

You only use the list by looping through it instead of accessing random elements

You frequently need to add and remove elements from the beginning or middle of the list

This design also makes the LinkedList comparably unfavorable to the ArrayList, which is usually the default List implementation in Java, in the following ways:

It uses more memory than ArrayList because of the storage used by its items' references, one for the previous item and one for the next item

Elements in a linked list must be read in order from the beginning (or end) as linked lists are inherently sequential access