LISTS

DATA STRUCTURES IN JAVA

Sercan Külcü | Data Structures In Java | 10.05.2023

Contents

List

A list is a data structure that can store a collection of elements. Lists are often used to store data that is ordered, such as a list of names or a list of numbers.

In Java, lists are implemented using the List interface. The List interface defines a number of methods for adding, removing, and accessing elements in a list.

Some of the most common methods in the List interface include:

- add(): Adds an element to the end of the list.
- remove(): Removes an element from the list.
- get(): Gets the element at a specific index in the list.
- size(): Gets the number of elements in the list.

There are a number of different implementations of the List interface in Java. Some of the most common implementations include:

- ArrayList: An array-based implementation of the List interface.
- LinkedList: A linked list implementation of the List interface.
- Vector: A synchronized implementation of the List interface.

The ArrayList implementation is the most commonly used implementation of the List interface. ArrayList is a dynamic array, which means that it can grow and shrink as needed.

The LinkedList implementation is a more efficient implementation of the List interface for accessing elements that are not near the beginning or end of the list.

The Vector implementation is a synchronized implementation of the List interface. This means that all access to the list is synchronized, which can be useful for preventing race conditions in multithreaded applications.

Lists are a powerful tool that can be used to store and manipulate data. By understanding how lists work, you can write more efficient and effective code.

Here are some examples of how lists can be used:

- A list can be used to store a shopping list.
- A list can be used to store a list of tasks to be completed.
- A list can be used to store a list of contacts.
- A list can be used to store a list of files.

Lists are a versatile data structure that can be used to store a variety of data. By understanding how lists work, you can write more efficient and effective code.

ArrayList

An ArrayList is a data structure that can store a collection of elements. ArrayLists are often used to store data that is ordered, such as a list of names or a list of numbers.

In Java, ArrayLists are implemented using the ArrayList class. The ArrayList class defines a number of methods for adding, removing, and accessing elements in an ArrayList.

Some of the most common methods in the ArrayList class include:

- add(): Adds an element to the end of the ArrayList.
- remove(): Removes an element from the ArrayList.
- get(): Gets the element at a specific index in the ArrayList.
- size(): Gets the number of elements in the ArrayList.

ArrayLists are a powerful tool that can be used to store and manipulate data. By understanding how ArrayLists work, you can write more efficient and effective code.

Here are some examples of how ArrayLists can be used:

- An ArrayList can be used to store a shopping list.
- An ArrayList can be used to store a list of tasks to be completed.
- An ArrayList can be used to store a list of contacts.
- An ArrayList can be used to store a list of files.

ArrayLists are a versatile data structure that can be used to store a variety of data. By understanding how ArrayLists work, you can write more efficient and effective code.

Here are some additional details about ArrayLists:

- ArrayLists are implemented using dynamic arrays, which means that they can grow and shrink as needed.

- ArrayLists are synchronized, which means that all access to the list is synchronized. This can be useful for preventing race conditions in multithreaded applications.
- ArrayLists are not thread-safe, which means that multiple threads can access the list at the same time. This can lead to race conditions, which are errors that can occur when multiple threads are trying to access the same data at the same time.

Here are some tips for using ArrayLists:

- Use ArrayLists when you need to store a collection of elements that is ordered.
- Use ArrayLists when you need to be able to add and remove elements from the collection.
- Use ArrayLists when you need to be able to access elements in the collection by index.
- Use ArrayLists when you need to be able to iterate over the collection.
- Use a synchronized ArrayList when you need to prevent race conditions in multithreaded applications.
- Use a thread-safe ArrayList when you need to allow multiple threads to access the list at the same time.

ArrayLists are a powerful tool that can be used to store and manipulate data. By understanding how ArrayLists work and using them correctly, you can write more efficient and effective code.

Let's start by taking a look at an example implementation using an ArrayList:

```
import java.util.ArrayList;

import java.util.List;


public class MyList {
```

```
    public static void main(String[] args) {
        List<String> myList = new ArrayList<>();

        // Adding elements to the list
        myList.add("apple");
        myList.add("banana");
        myList.add("cherry");

        // Accessing elements of the list
        String firstElement = myList.get(0);
        String lastElement = myList.get(myList.size() - 1);

        // Removing an element from the list
        myList.remove(1);

        // Iterating over the elements of the list
        for (String element : myList) {
            System.out.println(element);
        }
    }
}
```

In this example, we're creating an ArrayList of strings and adding three elements to it using the add method. We're then accessing the first and

last elements of the list using the get method, which takes an index as a parameter. We're also removing the second element of the list using the remove method, which also takes an index as a parameter. Finally, we're iterating over the elements of the list using a for-each loop.

One advantage of using an ArrayList is that it provides constant-time random access to its elements, meaning that accessing an element at a specific index takes the same amount of time, regardless of the size of the list. However, adding or removing elements from the middle of an ArrayList can be expensive, as it requires shifting all the subsequent elements by one position.

LinkedList

A linked list is a data structure that can store a collection of elements. Linked lists are often used to store data that is not ordered, such as a list of tasks to be completed or a list of files.

In Java, linked lists are implemented using the LinkedList class. The LinkedList class defines a number of methods for adding, removing, and accessing elements in a linked list.

Some of the most common methods in the LinkedList class include:

- add(): Adds an element to the end of the linked list.
- remove(): Removes an element from the linked list.
- get(): Gets the element at a specific index in the linked list.
- size(): Gets the number of elements in the linked list.

Linked lists are a powerful tool that can be used to store and manipulate data. By understanding how linked lists work, you can write more efficient and effective code.

Here are some examples of how linked lists can be used:

- A linked list can be used to store a list of tasks to be completed.
- A linked list can be used to store a list of files.
- A linked list can be used to store a list of websites to visit.
- A linked list can be used to store a list of friends on social media.

Linked lists are a versatile data structure that can be used to store a variety of data. By understanding how linked lists work, you can write more efficient and effective code.

Here are some additional details about linked lists:

- Linked lists are implemented using nodes, which are data structures that contain data and a reference to the next node in the list.

- Linked lists are not synchronized, which means that multiple threads can access the list at the same time. This can lead to race conditions, which are errors that can occur when multiple threads are trying to access the same data at the same time.
- Linked lists are a good choice for storing data that is not ordered, such as a list of tasks to be completed or a list of files.
- Linked lists are a good choice for storing data that needs to be frequently added to or removed from the beginning or end of the list.

Here are some tips for using linked lists:

- Use linked lists when you need to store a collection of elements that is not ordered.
- Use linked lists when you need to be able to add and remove elements from the collection frequently.
- Use linked lists when you need to be able to iterate over the collection in reverse order.

Linked lists are a powerful tool that can be used to store and manipulate data. By understanding how linked lists work and using them correctly, you can write more efficient and effective code.

Now, let's take a look at an example implementation using a LinkedList:

```
import java.util.LinkedList;

import java.util.List;


public class MyList {

   public static void main(String[] args) {

      List<String> myList = new LinkedList<>();
```

```
        // Adding elements to the list
        myList.add("apple");
        myList.add("banana");
        myList.add("cherry");

        // Accessing elements of the list
        String firstElement = myList.get(0);
        String lastElement = myList.get(myList.size() - 1);

        // Removing an element from the list
        myList.remove(1);

        // Iterating over the elements of the list
        for (String element : myList) {
            System.out.println(element);
        }
    }
}
```

In this example, we're creating a LinkedList of strings and adding three elements to it using the add method. We're then accessing the first and last elements of the list using the get method, which takes an index as a parameter. We're also removing the second element of the list using the remove method, which also takes an index as a parameter. Finally, we're iterating over the elements of the list using a for-each loop.

One advantage of using a LinkedList is that adding or removing elements from the middle of the list is cheap, as it only requires updating the next and previous pointers of the affected nodes. However, accessing an element at a specific index takes linear time, as the list must be traversed from the beginning or the end until the desired element is found.

In general, the choice between an ArrayList and a LinkedList depends on the specific use case. If random access to the elements is important and the list is relatively small or the number of modifications is limited, an ArrayList may be a good choice. If adding or removing elements from the middle of the list is frequent or the list is large,

Vector

A vector is a data structure that can store a collection of elements. Vectors are often used to store data that is ordered, such as a list of names or a list of numbers.

In Java, vectors are implemented using the Vector class. The Vector class defines a number of methods for adding, removing, and accessing elements in a vector.

Some of the most common methods in the Vector class include:

- add(): Adds an element to the end of the vector.
- remove(): Removes an element from the vector.
- get(): Gets the element at a specific index in the vector.
- size(): Gets the number of elements in the vector.

Vectors are a powerful tool that can be used to store and manipulate data. By understanding how vectors work, you can write more efficient and effective code.

Here are some examples of how vectors can be used:

- A vector can be used to store a shopping list.
- A vector can be used to store a list of tasks to be completed.
- A vector can be used to store a list of contacts.
- A vector can be used to store a list of files.

Vectors are a versatile data structure that can be used to store a variety of data. By understanding how vectors work, you can write more efficient and effective code.

Here are some additional details about vectors:

- Vectors are implemented using dynamic arrays, which means that they can grow and shrink as needed.

- Vectors are synchronized, which means that all access to the vector is synchronized. This can be useful for preventing race conditions in multithreaded applications.
- Vectors are not thread-safe, which means that multiple threads can access the vector at the same time. This can lead to race conditions, which are errors that can occur when multiple threads are trying to access the same data at the same time.

Here are some tips for using vectors:

- Use vectors when you need to store a collection of elements that is ordered.
- Use vectors when you need to be able to add and remove elements from the collection.
- Use vectors when you need to be able to access elements in the collection by index.
- Use vectors when you need to be able to iterate over the collection.
- Use a synchronized vector when you need to prevent race conditions in multithreaded applications.
- Use a thread-safe vector when you need to allow multiple threads to access the vector at the same time.

Vectors are a powerful tool that can be used to store and manipulate data. By understanding how vectors work and using them correctly, you can write more efficient and effective code.

Positional Lists

In this chapter, we will introduce positional lists, which are an extension of the linked list data structure. A positional list allows us to maintain a sequence of elements where each element is associated with a position, and we can access and manipulate these elements using their positions.

A positional list is a collection of elements, each of which is associated with a position in the list. We can think of each position as a unique identifier for an element, much like an index in an array. The position of an element in a positional list is determined by its relative position to other elements in the list. The two most common operations on a positional list are:

- elementAtRank(rank): This operation returns the element at a specified rank, which is the position of the element in the list.
- rankOf(element): This operation returns the rank of a specified element, which is its position in the list.

Positional lists can be implemented using a doubly linked list, where each node in the list contains a reference to its predecessor and successor nodes. In addition, each node also contains a reference to the element it represents, and an integer value that represents the rank of the element in the list.

Positional lists are commonly used in situations where we need to maintain a sequence of elements that can be accessed and manipulated using their positions. For example, they can be used to implement the undo and redo operations in a text editor, where each change to the document is represented by an element in the positional list.

Positional lists can also be used in the implementation of other data structures. For example, they are used in the implementation of binary trees, where each node in the tree is associated with a position in the list.

We can implement a positional list in Java using the PositionalList interface and the Position interface. The PositionalList interface defines the operations that can be performed on a positional list, while the Position interface represents a position in the list.

Here is an example implementation of the PositionalList interface:

```
public interface PositionalList<E> {

   int size();

   boolean isEmpty();

   Position<E> first();

   Position<E> last();

   Position<E> before(Position<E> p) throws IllegalArgumentException;

   Position<E> after(Position<E> p) throws IllegalArgumentException;

   Position<E> addFirst(E e);

   Position<E> addLast(E e);

   Position<E> addBefore(Position<E> p, E e) throws IllegalArgumentException;

   Position<E> addAfter(Position<E> p, E e) throws IllegalArgumentException;

   E set(Position<E> p, E e) throws IllegalArgumentException;

   E remove(Position<E> p) throws IllegalArgumentException;

   Iterable<Position<E>> positions();

   Iterable<E> elements();

}
```

The Position interface can be implemented as follows:

```
public interface Position<E> {
    E getElement() throws IllegalStateException;
}
```

Positional lists are a powerful data structure that allows us to maintain a sequence of elements that can be accessed and manipulated using their positions. They can be implemented using a doubly linked list and are commonly used in the implementation of other data structures. In the next chapter, we will introduce trees, which are another important data structure in computer science.