ARRAYS AND LINKED LISTS

DATA STRUCTURES IN JAVA

Sercan Külcü | Data Structures In Java | 10.05.2023

Contents

Arrays

An array is a collection of elements of the same data type stored in contiguous memory locations. It can be thought of as a group of variables with the same name and data type that are accessed using a common index. The index is used to identify the location of a specific element in the array.

Arrays in Java are implemented as objects, which means that they have methods that can be used to access and manipulate their elements. Arrays in Java are also fixed in size, which means that you must specify the size of the array when it is created.

Arrays have many advantages, including fast access times, easy implementation, and efficient memory usage. They are often used to store collections of data that are of the same type, such as a list of numbers or a set of strings.

To create an array in Java, you must first declare the array variable and specify the data type of the elements in the array. For example, to create an array of integers, you would use the following code:

```
int[] myArray;
```

This creates a variable called myArray that can hold an array of integers. To create the actual array, you must use the new keyword and specify the size of the array. For example:

```
myArray = new int[10];
```

This creates an array of integers with a size of 10. You can also initialize the elements of the array when you create it, like this:

```
int[] myArray = {1, 2, 3, 4, 5};
```

Once you have created an array, you can access its elements using the index. The index of the first element in an array is 0, and the index of the last element is one less than the size of the array. For example, to

access the first element in an array called myArray, you would use the following code:

```
int firstElement = myArray[0];
```

Arrays can be used in a variety of applications, including sorting algorithms, searching algorithms, and data compression. They are also commonly used in graphical user interfaces (GUIs) to store and display data.

Multidimensional Arrays

A multidimensional array is an array that can store multiple arrays. For example, a 2D array can store two arrays, a 3D array can store three arrays, and so on.

Multidimensional arrays are useful for storing data that has a natural multidimensional structure. For example, a 2D array can be used to store a matrix, a 3D array can be used to store a volume, and so on.

In Java, multidimensional arrays are declared using the following syntax:

```
type arrayName[][];
```

where type is the data type of the elements in the array, and arrayName is the name of the array.

For example, the following declaration creates a 2D array of integers:

```
int array[][] = new int[3][4];
```

This array can store 12 integers, arranged in a 3x4 grid.

Multidimensional arrays can be indexed using multiple subscripts. For example, the following statement assigns value 10 to the element at row 2, column 3 of the array:

```
array[2][3] = 10;
```

Multidimensional arrays can be used to store a variety of data, including numbers, strings, and objects.

Here are some examples of how multidimensional arrays can be used:

- A 2D array can be used to store a matrix.
- A 3D array can be used to store volume.
- A multidimensional array can be used to store a list of lists.
- A multidimensional array can be used to store a collection of objects.

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

Singly Linked Lists

A singly linked list is a data structure that consists of a series of nodes, where each node contains a data element and a reference (or pointer) to the next node in the list. The first node in the list is called the head, and the last node is called the tail. Singly linked lists are often used to represent linear sequences of data.

One of the advantages of singly linked lists is that they can be dynamically resized, which means that nodes can be added or removed from the list at any time. This makes them very flexible and useful in many different types of applications.

To implement a singly linked list in Java, you first need to create a Node class that will hold the data and the reference to the next node. Here is an example of a simple Node class:

```
class Node {
    int data;
    Node next;
}
```

Once you have defined the Node class, you can create a singly linked list by creating a head node and linking it to the next node, and so on, until you reach the end of the list.

To add a new node to the list, you create a new Node object and set its data and next values, and then link it to the previous node by setting its next reference to the new node. To remove a node from the list, you simply set the next reference of the previous node to the next node in the list, effectively removing the current node from the list.

One of the key operations on a singly linked list is traversing the list, which means visiting each node in the list in order. This can be done using a loop that starts at the head node and iteratively follows the next references until the end of the list is reached.

Singly linked lists can be used in a variety of applications, such as implementing stacks, queues, and hash tables. They are also useful for representing sparse matrices and graphs.

Circularly Linked Lists

Circularly linked lists, which are similar to singly linked lists but with one key difference - the last node in the list is linked back to the first node, creating a circular structure.

A circularly linked list is implemented using a Node class that is like the one used for singly linked lists, but with an additional reference to the

first node in the list. Here is an example of a simple Node class for a circularly linked list:

```
class Node {
    int data;
    Node next;
}
```

To create a circularly linked list, you first create a head node and link it to the next node in the list. Then, you continue linking each subsequent node to the next node until you reach the end of the list, at which point you link the last node back to the head node, creating a circular structure.

Circularly linked lists can be used in much the same way as singly linked lists, but with a few additional operations that take advantage of the circular structure. For example, you can traverse the list by starting at the head node and following the next references until you return to the head node, at which point you know you have visited every node in the list.

Another operation that is specific to circularly linked lists is rotating the list, which means moving the head node to the next node in the list and updating the last node's reference to point to the new head node. This operation can be useful in certain applications, such as implementing a round-robin scheduling algorithm.

One potential downside of circularly linked lists is that they can be more difficult to implement and manipulate than singly linked lists. However, they can also provide some advantages in terms of efficiency and convenience, particularly in situations where the circular structure is a natural representation of the data being stored.

Doubly Linked Lists

As the name suggests, a doubly linked list is similar to a singly linked list, but with one important difference: each node in a doubly linked list has references to both the next and previous nodes in the list. This allows for more flexibility and versatility when manipulating the list.

To implement a doubly linked list in Java, we first need to define a Node class that has three instance variables: a data field to store the value of the node, and two reference fields to point to the previous and next nodes in the list, respectively. Here is an example of what this class might look like:

```
class Node {

    int data;

    Node prev;

    Node next;

}
```

Once we have our Node class defined, we can create a doubly linked list by initializing the head and tail nodes and linking them together. From there, we can add and remove nodes from the list by manipulating the prev and next references.

One of the key advantages of a doubly linked list is that it allows for efficient traversal of the list in both directions. For example, to traverse a singly linked list in reverse, you would need to start at the head node and follow the next references until you reach the end of the list, then work your way back through the list using a stack or other data structure. With a doubly linked list, you can simply start at the tail node and follow the prev references back to the head node.

Another advantage of doubly linked lists is that they allow for efficient insertion and deletion of nodes in the middle of the list. With a singly

linked list, deleting a node requires updating the next reference of the previous node to point to the next node after the deleted node. With a doubly linked list, however, you can simply update the prev and next references of the neighboring nodes to remove the deleted node from the list.

While doubly linked lists can provide some advantages over singly linked lists, they also come with some additional complexity and overhead. For example, each node in a doubly linked list requires twice as many references as a node in a singly linked list, which can lead to higher memory usage and slower performance in some situations.

Integers with no size restriction

To represent integers with no size restriction, we can use a linked list or array-based representation for arbitrary-precision integers. This type of representation is commonly referred to as a Big Integer or Arbitrary-Precision Integer. The main idea is to store the integer's digits as individual elements of a data structure and perform operations digit by digit, much like manual calculations in mathematics.

Structure:

Use an array or linked list to store digits of the number in base-10 or base-2. Each element of the array or node in the linked list represents a single digit or a group of digits. The number is stored in reverse order to make addition and multiplication operations easier (e.g., least significant digit first). Use an additional flag (boolean isNegative) to store the sign of the number.

By representing integers as arrays or linked lists of digits, we can store arbitrarily large numbers, limited only by the available memory. Operations like addition, multiplication, and exponentiation are performed digit by digit, using manual computation techniques. This approach ensures no practical limit on integer size.

Example:

The number 12345 can be represented as an array [5, 4, 3, 2, 1] (least significant digit first).

The number -987654321 can be represented as an array [1, 2, 3, 4, 5, 6, 7, 8, 9] with isNegative = true.

1. Addition

Addition is performed digit by digit, accounting for carry, like manual addition. Traverse both numbers from the least significant digit (start of the array/list). Add corresponding digits and include any carry from the previous step. If one number is shorter, assume missing digits are 0. Append the carry to the result if it remains after processing all digits.

Example: Adding 123 ([3, 2, 1]) and 789 ([9, 8, 7]):

Add digits: 3 + 9 = 12, store 2, carry 1.

Add digits: 2 + 8 + 1 = 11, store 1, carry 1.

Add digits: 1 + 7 + 1 = 9, store 9.

Result: [2, 1, 9] → 912.

2. Multiplication

Multiply each digit of one number by every digit of the other number, keeping track of carry, and shift the results appropriately. Multiply each digit of the first number by each digit of the second number. Shift the partial results by their positional value (e.g., multiplying by 10, 100, etc.). Add all partial results using the addition algorithm.

Example: Multiplying 12 ([2, 1]) by 34 ([4, 3]):

Multiply 2 × 4 = 8, 2 × 3 = 6 → Partial result [8, 6].

Multiply 1 × 4 = 4, 1 × 3 = 3 → Partial result [4, 3] shifted one place → [0, 4, 3].

Add partial results: [8, 6] + [0, 4, 3] → [8, 0, 4] → 408.

3. Exponentiation

Exponentiation is achieved using repeated multiplication or optimized techniques like Exponentiation by Squaring. Represent the exponentiation as repeated multiplications: $a^b = a \times a \times \ldots \times a$ (b times).

Use the multiplication algorithm to compute each step.

Optimize using Exponentiation by Squaring for efficiency:

$a^b=(a^{b/2})^2$ if b is even.

$a^b= a \times a^{b-1}$ if b is odd.

Example: Computing 3^4

Step 1: $3 \times 3=9$.

Step 2: $9 \times 9=81$.

Implementation Notes

Use arrays for faster random access if performance is critical.

Use linked lists for dynamic memory allocation and to handle very large numbers.

Addition and multiplication must check the signs of the numbers and adjust the operation (e.g., subtraction for opposite signs).

Most programming languages provide libraries for arbitrary-precision integers (e.g., java.math.BigInteger in Java, int in Python, BigInt in JavaScript).

Abstract Data Type (ADT) for Sparse Arrays

A two-dimensional array is a collection of integers organized in rows and columns, supporting operations for accessing and modifying elements, as well as other utility functions.

Basic Operations

Here are the operations defined for a two-dimensional array:

Create: Initialize a two-dimensional array of size m×n, optionally with a default value (e.g., 0). Example: create(m, n, defaultValue).

Set Value: Set the value of the element at position (i, j). Example: set(i, j, value).

Get Value: Retrieve the value of the element at position (i, j). Example: get(i, j).

Get Dimensions: Retrieve the number of rows and columns in the array. Example: getDimensions() → (rows, columns).

Reset: Reset all elements in the array to a specific value (e.g., 0). Example: reset(value).

Iterate: Iterate over all elements or only non-zero elements. Example: iterate() or iterateNonZero().

Count Non-Zero Elements: Return the number of non-zero elements in the array. Example: countNonZero().

Transpose: Swap rows and columns of the array. Example: transpose().

Get Row/Column: Retrieve all elements from a specific row or column. Example: getRow(i) or getColumn(j).

Efficient Implementations for Sparse Arrays

Given the scenario of a 1000×1000 array where fewer than 10,000 elements are non-zero, a standard implementation would require memory for 1,000,000 elements, most of which would be unnecessary.

Coordinate List (COO) Representation

This method stores only the non-zero elements along with their row and column indices.

Use a list of tuples (row, column, value) for each non-zero element.

Example: For the following array:

0 0 5

0 0 0

3 0 0

The COO representation is: [(0, 2, 5), (2, 0, 3)].

Operations:

Get Value: Search the list for the (row, column) pair.

Time Complexity: O(k), where k is the number of non-zero elements.

Set Value: Update the existing tuple if it exists or add a new tuple for non-zero values.

Time Complexity: O(k).

Space Complexity: O(k), where k is the number of non-zero elements.

Advantages: Space-efficient for sparse arrays. Simple to implement.

Disadvantages: Slower access compared to direct indexing in a dense array.