DATA STRUCTURES

When comparing data structures for different operations such as insertion, deletion, updating values, retrieval, time complexity, and space efficiency, the following data structures from the `java.util` package are commonly used and suitable for different scenarios:

1. ArrayList:

A number of objects in a row

Description automatically generated

- It provides dynamic arrays that can grow or shrink dynamically.

- Insertion, deletion, and updating values: The time complexity is O(1) for appending elements to the end and updating values by index. However, inserting or removing elements in the middle of the list requires shifting elements, resulting in a time complexity of O(n).

- Retrieval: The time complexity for retrieving elements by index is O(1).

- Space-wise: It consumes contiguous memory, which may result in wasted space if the size changes frequently.

1. LinkedList:

A diagram of data flow

Description automatically generated

- It provides a doubly-linked list where each element has references to the previous and next elements.

- Insertion, deletion, and updating values: Insertion and deletion at both ends (head or tail) have a time complexity of O(1). However, inserting or removing elements in the middle requires traversing the list, resulting in a time complexity of O(n).

- Retrieval: The time complexity for retrieving elements by index is O(n) since it requires traversing the list from the head or tail.

- Space-wise: It requires additional memory for maintaining the links between elements.

1. HashMap:

A diagram of a key value pair

Description automatically generated

- It provides a hash table implementation where elements are stored based on key-value pairs.

- Insertion, deletion, and updating values: The time complexity is generally O(1) for these operations. However, in worst-case scenarios where there are many collisions, the time complexity may become O(n).

- Retrieval: The time complexity for retrieving values by key is typically O(1), as it uses the hash function to locate the element quickly.

- Space-wise: It requires additional memory for storing the keys and values, and the space usage grows with the number of elements stored.

1. TreeSet:

A diagram of a diagram

Description automatically generated

- It provides a red-black tree implementation that stores elements in a sorted order.

- Insertion, deletion, and updating values: These operations have a time complexity of O(log n) as the tree is self-balancing.

- Retrieval: The time complexity for retrieving elements is O(log n) since it performs binary search on the tree.

- Space-wise: It requires additional memory for storing the tree structure and the elements.

For choosing the appropriate data structure, consider the following guidelines:

- If you need frequent random access and modification by index, ArrayList is a good choice.

- If you frequently insert or delete elements at the beginning or end of the collection, LinkedList performs well.

- If you require key-value mappings and fast retrieval by key, HashMap is suitable.

- If you need sorted elements and efficient retrieval, TreeSet provides ordered storage.

It's important to note that the choice of data structure depends on the specific requirements of your application and the characteristics of your data. It's recommended to benchmark and analyze the performance for your specific use case to make an informed decision.

import java.util.ArrayList;

import java.util.HashMap;

import java.util.LinkedList;

import java.util.TreeSet;

public class DataStructureTest {

public static void main(String[] args) {

// Test case for 1-100 elements

int[] elements = generateElements(1, 100);

// ArrayList

long arrayListStartTime = System.nanoTime();

ArrayList<Integer> arrayList = new ArrayList<>();

for (int element : elements) {

arrayList.add(element);

}

long arrayListEndTime = System.nanoTime();

System.out.println("ArrayList (1-100): " + (arrayListEndTime - arrayListStartTime) + " ns");

// LinkedList

long linkedListStartTime = System.nanoTime();

LinkedList<Integer> linkedList = new LinkedList<>();

for (int element : elements) {

linkedList.add(element);

}

long linkedListEndTime = System.nanoTime();

System.out.println("LinkedList (1-100): " + (linkedListEndTime - linkedListStartTime) + " ns");

// HashMap

long hashMapStartTime = System.nanoTime();

HashMap<Integer, Integer> hashMap = new HashMap<>();

for (int element : elements) {

hashMap.put(element, element);

}

long hashMapEndTime = System.nanoTime();

System.out.println("HashMap (1-100): " + (hashMapEndTime - hashMapStartTime) + " ns");

// TreeSet

long treeSetStartTime = System.nanoTime();

TreeSet<Integer> treeSet = new TreeSet<>();

for (int element : elements) {

treeSet.add(element);

}

long treeSetEndTime = System.nanoTime();

System.out.println("TreeSet (1-100): " + (treeSetEndTime - treeSetStartTime) + " ns");

System.out.println();

// Example cases for larger sizes (1000, 10000, 100000000)

int[] exampleSizes = {1000, 10000, 100000000};

for (int size : exampleSizes) {

elements = generateElements(1, size);

// ArrayList

arrayListStartTime = System.nanoTime();

arrayList = new ArrayList<>();

for (int element : elements) {

arrayList.add(element);

}

arrayListEndTime = System.nanoTime();

System.out.println("ArrayList (" + size + " elements): " + (arrayListEndTime - arrayListStartTime) + " ns");

// LinkedList

linkedListStartTime = System.nanoTime();

linkedList = new LinkedList<>();

for (int element : elements) {

linkedList.add(element);

}

linkedListEndTime = System.nanoTime();

System.out.println("LinkedList (" + size + " elements): " + (linkedListEndTime - linkedListStartTime) + " ns");

// HashMap

hashMapStartTime = System.nanoTime();

hashMap = new HashMap<>();

for (int element : elements) {

hashMap.put(element, element);

}

hashMapEndTime = System.nanoTime();

System.out.println("HashMap (" + size + " elements): " + (hashMapEndTime - hashMapStartTime) + " ns");

// TreeSet

treeSetStartTime = System.nanoTime();

treeSet = new TreeSet<>();

for (int element : elements) {

treeSet.add(element);

}

treeSetEndTime = System.nanoTime();

System.out.println("TreeSet (" + size + " elements): " + (treeSetEndTime - treeSetStartTime) + " ns");

System.out.println();

}

}

private static int[] generateElements

Certainly! Here are the modified test cases to put elements from 1 to 100 and measure the timestamp for each data structure. Additionally, I have included example cases for larger sizes (1000, 10000, 100000000):

A screenshot of a computer

Description automatically generated

1. `elements` (int[]): This variable stores an array of integers that will be used as the elements for testing different data structures. It is generated using the `generateElements` method.

2. `arrayList` (ArrayList<Integer>): This variable represents an ArrayList data structure. It is used to store integers in a dynamic array-like structure. In the code, elements are added to this ArrayList using a loop.

3. `linkedList` (LinkedList<Integer>): This variable represents a LinkedList data structure. It is used to store integers in a linked list structure where each element has a reference to the next element. Similar to `arrayList`, elements are added to this LinkedList using a loop.

4. `hashMap` (HashMap<Integer, Integer>): This variable represents a HashMap data structure. It is used to store key-value pairs, where the keys and values are both integers. In the code, elements are added to this HashMap as key-value pairs, with the element itself being used as both the key and the value.

5. `treeSet` (TreeSet<Integer>): This variable represents a TreeSet data structure. It is used to store a sorted set of integers. In the code, elements are added to this TreeSet, which automatically maintains the sorting order.

6. `arrayListStartTime`, `arrayListEndTime`, `linkedListStartTime`, `linkedListEndTime`, `hashMapStartTime`, `hashMapEndTime`, `treeSetStartTime`, `treeSetEndTime` (long): These variables store the start and end times (in nanoseconds) for measuring the execution time of adding elements to the respective data structures. The difference between `endTime` and `startTime` for each data structure provides the elapsed time for the operation.

7. `exampleSizes` (int[]): This variable stores an array of example sizes (1000, 10000, and 100000000) that represent the number of elements to be tested for each data structure.

The `main` method executes the code logic. It performs the following steps:

1. Generates an array of elements from 1 to 100 using the `generateElements` method.

2. Measures the execution time of adding elements to the ArrayList, LinkedList, HashMap, and TreeSet data structures and prints the results.

3. Repeats the above step for larger example sizes (1000, 10000, and 100000000) by generating a new array of elements for each size.

Overall, this code measures and compares the execution times of adding elements to different data structures (ArrayList, LinkedList, HashMap, and TreeSet) for various example sizes.

A screenshot of a computer program

Description automatically generated

DATA STTRUCTURES WITH OPERATIONS

**import** java.util.ArrayList;

**import** java.util.HashMap;

**import** java.util.LinkedList;

**import** java.util.TreeSet;

**public** **class** Operations {

**public** **static** **void** main(String[] args) {

**int**[] testCases = { 1, 100, 10000, 100000 };

**for** (**int** testCase : testCases) {

System.***out***.println("Test Case: " + testCase);

System.***out***.println("-----------------------------");

*testOperations*(**new** ArrayList<>(), testCase);

*testOperations*(**new** LinkedList<>(), testCase);

*testOperations*(**new** HashMap<>(), testCase);

*testOperations*(**new** TreeSet<>(), testCase);

System.***out***.println();

}

}

**private** **static** **void** testOperations(Object dataStructure, **int** testCase) {

**long** startTime, endTime;

// Generate elements based on the testCase

**int**[] elements = *generateElements*(1, testCase);

// Insertion

startTime = System.*nanoTime*();

// Add elements to the data structure

// ...

endTime = System.*nanoTime*();

System.***out***.println("Insertion Time: " + (endTime - startTime) + " ns");

// Retrieval

startTime = System.*nanoTime*();

// Retrieve elements from the data structure

// ...

endTime = System.*nanoTime*();

System.***out***.println("Retrieval Time: " + (endTime - startTime) + " ns");

// Deletion

startTime = System.*nanoTime*();

// Delete elements from the data structure

// ...

endTime = System.*nanoTime*();

System.***out***.println("Deletion Time: " + (endTime - startTime) + " ns");

// Memory Usage

**long** memoryUsage = Runtime.*getRuntime*().totalMemory() - Runtime.*getRuntime*().freeMemory();

System.***out***.println("Memory Usage: " + memoryUsage + " bytes");

}

**private** **static** **int**[] generateElements(**int** start, **int** end) {

**int**[] elements = **new** **int**[end - start + 1];

**for** (**int** i = 0; i < elements.length; i++) {

elements[i] = start + i;

}

**return** elements;

}

}

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

1. Added testOperations methods for each data structure to perform retrieval, deletion, and memory usage operations.
2. Each testOperations method takes the data structure (list, map, set) and the elements array as parameters.
3. Inside each testOperations method, the following operations are performed: Insertion: Elements are added to the data structure.
4. Retrieval: Elements are retrieved and an operation (multiplying by 2) is performed with each retrieved element.
5. Deletion: Elements are removed from the data structure. Memory Usage: The memory usage of the data structure is calculated using the Runtime class.
6. The main method now calls the testOperations method for each data structure with the respective data structure instance (arrayList, linkedList, hashMap, treeSet) and the generated elements array.

Make a table of rows are data structures and the columns are the operations and to fill up the lapse time, for linked list and hash mash how much time for a million. One column has get, set, remove, insert operations. Many nodes.

Code:

**package** Simpleproject;

**import** java.util.\*;

**public** **class** PerformanceTable {

**private** **static** **final** **int** ***NUM\_ELEMENTS*** = 1\_000\_000;

**private** **static** **final** **int** ***NUM\_ITERATIONS*** = 10; // Number of iterations for accurate measurement

**public** **static** **void** main(String[] args) {

List<Integer> linkedList = **new** LinkedList<>();

Map<Integer, Integer> hashMap = **new** HashMap<>();

// Fill the linked list and hash map with elements

**for** (**int** i = 0; i < ***NUM\_ELEMENTS***; i++) {

linkedList.add(i);

hashMap.put(i, i);

}

System.***out***.println("Operation\tLinkedList (ms)\tHashMap (ms)");

// Perform get operation

**long** linkedListGetTime = *measureTime*(() -> linkedList.get(***NUM\_ELEMENTS*** / 2));

**long** hashMapGetTime = *measureTime*(() -> hashMap.get(***NUM\_ELEMENTS*** / 2));

System.***out***.println("Get\t\t" + linkedListGetTime + "\t\t" + hashMapGetTime);

// Perform set operation

**long** linkedListSetTime = *measureTime*(() -> linkedList.set(***NUM\_ELEMENTS*** / 2, -1));

**long** hashMapSetTime = *measureTime*(() -> hashMap.put(***NUM\_ELEMENTS*** / 2, -1));

System.***out***.println("Set\t\t" + linkedListSetTime + "\t\t" + hashMapSetTime);

// Perform remove operation

**long** linkedListRemoveTime = *measureTime*(() -> linkedList.remove(***NUM\_ELEMENTS*** / 2));

**long** hashMapRemoveTime = *measureTime*(() -> hashMap.remove(***NUM\_ELEMENTS*** / 2));

System.***out***.println("Remove\t\t" + linkedListRemoveTime + "\t\t" + hashMapRemoveTime);

// Perform insert operation

**long** linkedListInsertTime = *measureTime*(() -> linkedList.add(***NUM\_ELEMENTS*** / 2, -1));

**long** hashMapInsertTime = *measureTime*(() -> hashMap.put(***NUM\_ELEMENTS*** / 2, -1));

System.***out***.println("Insert\t\t" + linkedListInsertTime + "\t\t" + hashMapInsertTime);

}

**private** **static** **long** measureTime(Runnable operation) {

**long** totalTime = 0;

**for** (**int** i = 0; i < ***NUM\_ITERATIONS***; i++) {

**long** startTime = System.*currentTimeMillis*();

operation.run();

**long** endTime = System.*currentTimeMillis*();

totalTime += endTime - startTime;

}

**return** totalTime / ***NUM\_ITERATIONS***; // Average time over multiple iterations

}

}

Output:

A screenshot of a computer

Description automatically generated

How to increase the heap space in java.

To increase the heap space in Java, you need to adjust the Java Virtual Machine (JVM) parameters using the "-Xmx" flag. The "-Xmx" flag is used to set the maximum heap size, which allows Java applications to use more memory for objects and data.

**Eclipse**: In Eclipse, you can specify the heap size by going to "Run" > "Run Configurations" > "Arguments" tab. Then, in the "VM arguments" field, add "-Xmx" followed by the desired heap size. For example, "-Xmx1G" for 1GB heap space.