**Logbook**

**Bradley Pratt - Computer Games Programming U1664020314**

**Algorithms Processes and Data**

**Week 1-2 :**

**package** intArrays;

**import** java.util.Arrays;

**public** **class** CleverRandomListing **extends** RandomListing {

**public** CleverRandomListing (**int** size) {

**super**(size);

}

/\*\*

\* The purpose of this method is to rebuild an array in a completely random order

\* Passes an array in from the SortedListing class.

\*/

**protected** **void** randomise() {

**for** (**int** index = 0; index < getArray().length; index++) {

**int** randomArray = getRandomIndex(); // Uses the getRandomIndex method to randomise the array index

**int** newInt = getArray()[randomArray];

getArray()[randomArray] = getArray()[index]; // Changes the grabbed array to randomise its index

getArray()[index] = newInt; // Builds the array using its new index

}

}

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

RandomListing count = **new** CleverRandomListing(50); // create a new list, as long as the specified length.

System.***out***.println(Arrays.*toString*(count.getArray())); // prints the array to the console

}

}

The tests for this class showed that is more efficient than the standard sorting class, with a testMillionSize taking 96423 milliseconds. In SimpleRandomTesting testMillionSize took 261148 milliseconds in my last test; proving the above shown method is more efficient.

**Week 3-4**

/\*\*

\* Swaps the specified elements within the array

\* **@param** array the array which is passed into the method

\* **@param** index1 the index which needs to be swapped with index2

\* **@param** index2

\*/

**public** **static** <T> **void** swap(T[] array, **int** index1, **int** index2) {

T objectOne = array[index1];

T objectTwo = array[index2];

array[index1] = objectTwo; //Uses the defined first position and places "objectTwo" there

array[index2] = objectOne; //Uses the defined second position and places "objectOne" there

}

/\*\*

\* The purpose of max is to find the largest element in between index1 and index2.

\* **@param** array is the array that is passed in

\* **@param** index1 is the first index, which elements before it may be ignored

\* **@param** index2 is the second index location, which elements after it may be ignored

\* **@return** returns the largest element

\*/

**public** **static** <T> String max(String[] array, **int** index1, **int** index2) {

**int** index = 0;

**int** elementLength = array[0].length();

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

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

**if** (i >= index1 && i <= index2) {

**if** (array[i].length() > elementLength) {

index = i;

elementLength = array[i].length();

}

}

}

**return** array[index];

}

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

String[] names = {"Hugh", "Andrew", "Ebrahim","Diane","Paula", "Simon"};

System.***out***.println(*max*(names,1,4));

}

}

The tests class for the swap method shows that the elements are successfully swapped, using the array 1,2,3,4,5 and adding index1 as 1, and index2 as 2 showed that the array became 1,3,2,4,5 as expected.

**Week 5**

(10,000) = 180

(20,000) = 890

(30,000) = 2150

(40,000) = 4000

(50,000) = 6500

I ran the test three times and above are the results I got, the formulas show the average result found between the three tests. A function couldn’t be found since there doesn’t seem to be a running trend that would allow you to predict the next result with reasonably accuracy.

/\*\*

\* Method for the SelectionSort

\*/

**public** **void** sort(T[] array) {

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

**int** minIndex = i;

**for** (**int** j = i + 1; j < array.length; j++) {

**if** (array[j].compareTo(array[minIndex]) < 0) {

minIndex = j;

}

}

**if** (minIndex != i) {

T temp = array[i];

array[i] = array[minIndex];

array[minIndex] = temp;

}

}

}

The SelectionSort algorithm builds the array first and sorts it as it is built. The algorithm checks the value of each element of the array as it is inputted, if the value of the element at the position of j is greater than the element at the position of minIndex, then j is checked if it is not equal to i then i is replaced with j.

/\*\*

\* Method for the quicksort

\*/

**private** **void** sort(T[] array,**int** from,**int** to) {

**if** (from < to) {

**int** pivotIndex = from;

**int** highIndex = to;

**int** lowIndex = pivotIndex;

T pivot = array[(highIndex + lowIndex) / 2];

**do** { //Runs a do-while loop so that the method is ran whilst the conditions are true

**while** (array[lowIndex].compareTo(pivot) < 0) lowIndex++; //Increases the lowIndex amount by the amount of elements before the pivot

**while** (pivot.compareTo(array[highIndex]) < 0) highIndex--; //Reduces the highIndex amount by the amount of elements above it, meaning elements above the pivot are ignored

**if** (lowIndex <= highIndex) { //Checks the size of the element to see if it can be swapped

T temp = array[lowIndex]; //Gets the lowIndex and places it in the generic temp

array[lowIndex] = array[highIndex]; //Moves the smaller element to the higher element

array[highIndex] = temp; //Changes the the value of highIndex to the temp

lowIndex++;

highIndex--;

}

} **while** (lowIndex <= highIndex); //Runs the do while this is true

sort(array, from, highIndex);//Reruns the do-while loop with the new pivot

sort(array, lowIndex, to);

}

}

|  |  |  |
| --- | --- | --- |
| 10000 | 64.472 | 65.462 |
| 20000 | 278.786 | 284.874 |
| 30000 | 690.19 | 678.444 |
| 40000 | 1281.19 | 1275.674 |
| 50000 | 2079.638 | 2026.574 |
| 60000 | 3033.605 | 3042.342 |
| 70000 | 4222.341 | 4223.89 |
| 80000 | 5605.172 | 5559.824 |

|  |  |  |
| --- | --- | --- |
| 1000000 | 204.888 | 204.813 |
| 2000000 | 479.814 | 455.249 |
| 3000000 | 734.502 | 735.89 |
| 4000000 | 1023.676 | 1062.652 |
| 5000000 | 1316.623 | 1347.981 |
| 6000000 | 1633.81 | 1659.239 |
| 7000000 | 2005.972 | 2142.408 |
| 8000000 | 2335.258 | 2526.845 |
| 9000000 | 2679.885 | 2798.874 |
| 10000000 | 2988.025 | 3154.712 |
| 20000000 | 6648.698 | 7149.259 |

I ran two graphs, the first being the results selection sort and the second graph being the results for the quick sort. The selection sort doesn’t seem to maintain a trend per each 10000. The quick sort shows an increase of 300 per 1000000 and overall looks a great deal more efficient than the selection sort algorithm.

**Week 6**

**Week 7**

**package** binaryTree;

**public** **class** BinaryTree<T **extends** Comparable<? **super** T>> **implements** BTree<T> {

TreeNode<T> root;

//BTree<T> left, right;

@Override

**public** **void** insert(T value) {

**if** (root == **null**) {

root = **new** TreeNode<T>(value);

} **else** **if** (value.compareTo(value()) < 0) {

root.left().insert(value);

} **else** {

root.right().insert(value);

}

}

@Override

**public** T value() {

**return** root.value;

}

@Override

**public** BTree<T> left() {

**return** root.left;

}

@Override

**public** BTree<T> right() {

**return** root.right;

}

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

BinaryTree<Integer> tree = **new** BinaryTree<>();

tree.insert(1);

tree.insert(0);

tree.insert(2);

Integer leftValue = tree.left().value();

Integer rightValue = tree.right().value();

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

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

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

}

}

**Week 8**

The hashtableWrapper in week 8 uses modular arithmetic to sort the positions of objects within the hashtable. The way it works out where to place these objects is to use the following equation of “object hash” % “length of hashtable”. This is an efficient way to store the objects of the hash table since conflicts are unlikely since the hash is unique for each object.

The size of the hashtable also increases once it has reached the threshold of 0.75 and seems to use a formula of arrayLength x 2n. N being the amount of times the array has increased in length.

**Week 10**

**public** **class** DepthFirstTraversal <T> **extends** AdjacencyGraph <T> **implements** Traversal <T> {

**private** List<T> traversal = **new** ArrayList<T>();

**private** List<T> visited = **new** ArrayList<T>();

@Override

**public** List<T> traverse() **throws** GraphError {

**while**(visited.size() < getNodes().size()) { //Makes sure that the visited array is smaller than the amount of nodes

**for** (T node: getNodes()) { //Goes through the nodes one by one to build the array

getUnvisitedNode(node); //Checks the node to see if it has been visited before

traverse(node); //Beings the depth first traversal with the new node

}

}

**return** traversal; // Returns the array once its completed

}

**void** traverse(T node) **throws** GraphError {

**for** (T neighbour: getNeighbours(node)) { //Goes through each neighbour of the node

node = neighbour; //Changes the node to the neighbour

**if**(node != **null** && !visited.contains(node)) // If the node hasn't been visited(to prevent going to the same neighbour over and over)

{

traversal.add(node); //Adds the node to the array

visited.add(node); //Adds the node to the visited array so it prevents infinite loops

traverse(node); //Traverses again

}

}

}

T getUnvisitedNode(T node) **throws** GraphError {

visited.add(node); //Adds the node to the visited array to prevent infinite loops

**return** node; //Returns the node to the traverse

}

**Week 11**

**public** **class** RefCountTopologicalSort<T> **extends** AdjacencyGraph<T> **implements** TopologicalSort<T> {

**private** HashMap<T,Integer> refCountTable = **new** HashMap<T,Integer>();

**private** Stack<T> sort = **new** Stack<T>();

@Override

**public** List<T> getSort() **throws** GraphError {

setUpRefCounts();

sort();

**return** sort;

}

**private** **void** setUpRefCounts() **throws** GraphError {

initialiseRefCounts();

countReferences();

}

**private** **void** countReferences() **throws** GraphError {

**for** (T node: getNodes()) { //Get all the nodes and go through them one by one

**for** (T neighbour: getNeighbours(node)) { //Get the neighbours of the selected node

**int** currentCount = refCountTable.get(neighbour); //Get the number of children of the neighbour

refCountTable.put(neighbour, ++currentCount); //Increment the current count in the refCountTable for the amount of neighbours

}

}

}

**private** **void** initialiseRefCounts() {

**for** (T node: getNodes()) { // Gather the nodes

refCountTable.put(node, 0); // Declare all nodes in the table as a count of 0

}

}

**private** **void** sort() **throws** GraphError {

T node; // Declare the node as a variable

**while** ((node = nextReferenceZeroNode()) != **null**) { // Check the nodes next reference and continue the loop whilst it isn't null

**for** (T neighbour: getNeighbours(node)) { // Check the neighbours of the node

Integer count = refCountTable.get(neighbour); // Get the amount of neighbours from the refCountTable

**if** (count != **null**) {

refCountTable.put(neighbour, count-1); // Reduce the count from the refCountTable by 1 for each neighbour

}

refCountTable.put(node, count-1);

}

refCountTable.remove(node); // Remove the node from the refCountTable

sort.add(node); // Add the node to the sort

}

}

**private** T nextReferenceZeroNode(){

**for** (Entry<T, Integer> entry : refCountTable.entrySet()) { // Get each T and Integer from the entrySet

**if**(entry.getValue() == 0){

**return** (T) entry.getKey(); // Returns the getKey to the sort

}

}

**return** **null**;

}