2022 – 2023 Fall Semester
CSE 2105 – Data Structures

Project Report

Çağan ATAN 210315025
Barış POZLU 210315067

Assignment 1:

We implemented a B-Tree to sort a given array of keys. A tree sort's time complexity for sorting granted that it is balanced is n * log(n) which performs a lot better compared to sorting it naively.

Because the whole point of implementing the B-Tree is to sort an array, we didn't implement the methods that we won't use. For example; remove, contains, size methods.

We have an internal Node class that has key and child arrays, a field to determine if it is a leaf node or not, and a field to keep track of the number of keys.

We don't need to keep track of the number of children explicitly because we know for sure that the number of children is equal to the number of keys + 1

We have the BTree class that has a field called T that represents the branching factor. While constructing the BTree the user determines its value. And a root Node

field to have a reference to it. We also have methods for insertion and split operations, but we won't get into much detail about them because the main focus is going to be on the sorting algorithm.

```
public void sortArray(int[] array)
```

This method takes in the array that we want to sort, loops through every element in it and inserts them to the B-Tree. After that it invokes the private sortArray method. The reason we have 2 sortArray methods one being public other one being private is the private one is a recursive method. And for that method to work, it needs 2 additional parameters. But we don't want the user the worry about them. Thus, we just get the array in this method and invoke the private method after also passing those necessary parameters.

```
private int sortArray(int[] array, int insertIndex,
Node node)
```

This is the method where the major work is done. Public method invokes this method where insertIndex is set to be 0 and node is set to be the root node.

Let's start with why we use these additional parameters, insertIndex is used to determine exactly in which index we want to insert the key. Because we start adding the values from lowest to highest, it starts from zero and we keep increasing it as we insert. Node parameter is used to determine exactly which node we are traversing now.

So, the method starts from the root node, we recursively go the leftmost child of the current node. And when we reach a leaf node, we insert them to the array in order. And when the leftmost child is done with, it inserts the leftmost value of the node before it that hasn't been inserted yet, then starts to traverse that node's next child. As we do this, we keep inserting the keys to the array. When every function call is done the array becomes sorted.

Assignment 2:

We implemented a hash table using a binary search tree in order to resolve collisions. The upside of using a binary search tree instead of a linked list is that the time complexity of binary search trees for searching is logarithmic whereas linked lists' time complexity for searching is linear, which will boost the performance noticeably. But choosing a binary search tree over a linked list doesn't come without a cost.

LinkedList's insertion is constant time granted that you insert at the head or tail which we would do if we used a linked list. But binary search tree's insertion is logarithmic.

So, at the end we lose some performance in insertion but gain some in searching compared to using a LinkedList.

We have an Entry class whose keys need to be comparable. The reason for that is in order to be able to use a binary search tree to resolve collisions, the entries in the binary search tree need to be comparable. And the way we compare the entries is by comparing their keys. This is crucial for our program to work.

We have the HashTable class that implements the Iterable interface. We did this to enable the users to use the foreach operation on our hashTable.

```
private int normalizeIndex(int hashCode)
```

The first critical method that we have is the normalizeIndex method. What this method does is it takes in a hash code which is generated by a built-in method in java, gets rid of the negative sign and takes the mod to ensure that the index is within the bounds.

```
private Entry<K, V> bucketSeekEntry(K key, int index)
```

This method is almost used everywhere in our program. It takes a key of type K and index of type int. The purpose of this method is to find the entry that has the given key and is in the given index in the table. And return it. If it does not exist, then it simply returns null. Given the index, it finds the corresponding binary search tree and checks if we have an entry that has the given key and returns it.

```
private V bucketInsertEntry(Entry<K, V> entry, int
index)
```

Given the index, it finds the corresponding binary search tree. If it is null, it creates one. Then if there exists an entry already with the given key, it simply updates the value of the key and returns the original value. Otherwise, it inserts the given entry to the binary search tree and returns null indicating there was not an entry that had the same key before. After inserting if we reach the threshold, it invokes the resizeTable method.

```
private V bucketRemoveEntry(K key, int index)
```
This method invokes the bucketSeekEntry method and if it returns null, we know that there isn't an entry with the given key. So, we also return null. Otherwise, we find the corresponding binary search tree using the given index and remove the entry with the given key. And return the value of the removed entry.

```
private void resizeTable()
```
We invoke this method when we reach the threshold. This method creates a new table with doubled capacity, and it recalculates the threshold. It basically traverses through every entry and copies them to the new table. While doing so, it also clears the old table to help the garbage collector.

```
public V get(K key)
```
if the given key is null, it simply returns null. Otherwise, it calculates the index using the normalizeIndex method. It also invokes bucketSeekEntry method. If it is null it also returns null beacuse there is no entry with the given key. Otherwise, it gets the entry and returns its value.

```
public V insert(K key, V value)
```
It simply returns null if any of the key or value is null. It invokes the normalizeIndex method to find the index using the key's hash code. Then it creates an entry object using the given key and the value and invokes the bucketInsertEntry method. Then returns its value.

```
public V remove(K key)
```
It returns null if the given key is null. Otherwise it calculates the index using the normalizeIndex method. Then it invokes bucketRemoveEntry method and returns its value.

Assignment 3:

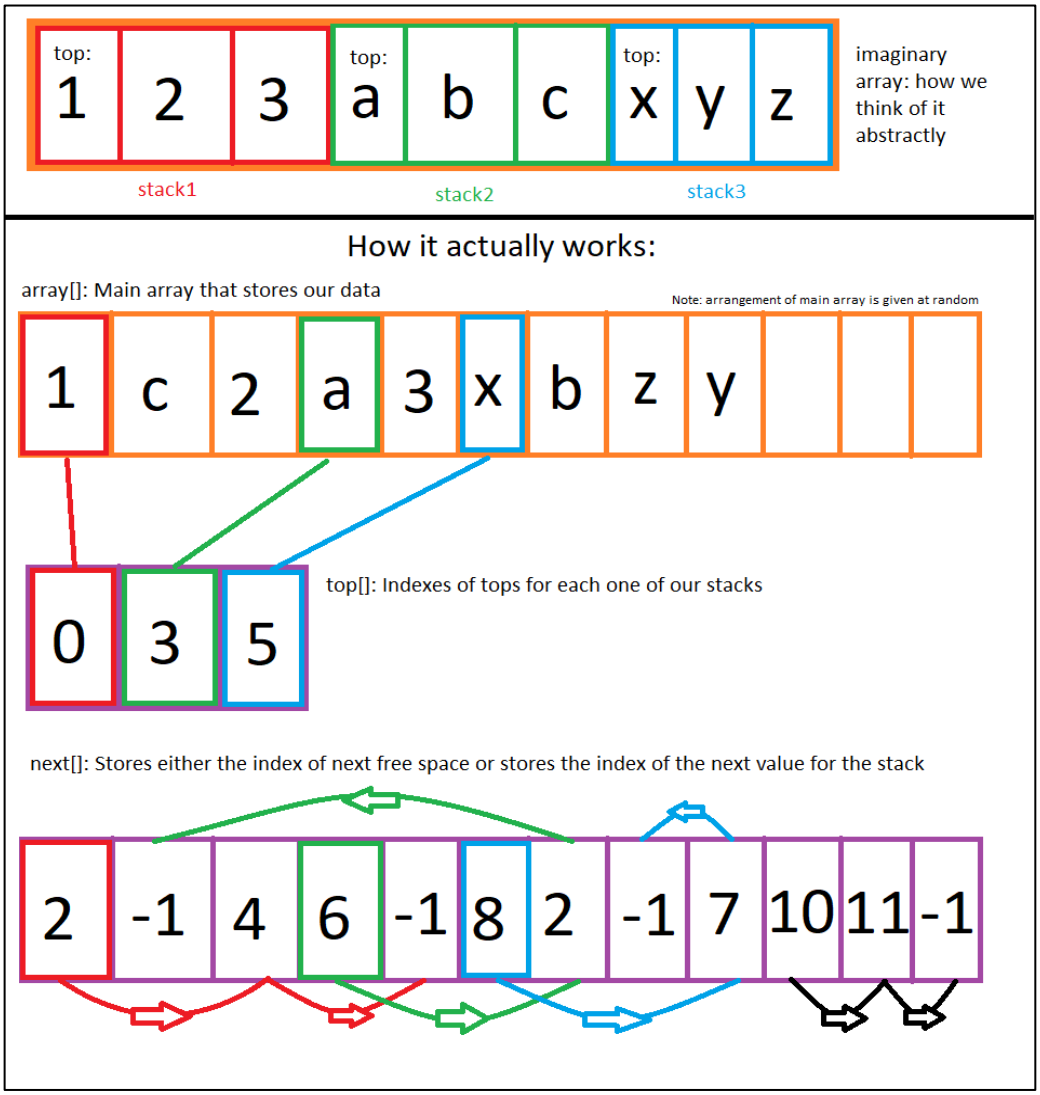
We can use 2 different implementations to store multiple stacks in an array:

1. We can partition the array into n pieces where n is the number of stacks. This will cause all the stacks to have same capacity. The issue with this implementation is you could have filled up a stack whilst there is already enough space in the array to push the element. This implementation makes sense if you have stacks of relatively **small object types** such as integers and Booleans.

2. We can implement 3 different arrays where first one is where we store the data, second one is where we store the indexes of the top values, third one where we either store the next free space or the next value in the stack. We use -1 to indicate its either the last value of the stack, or it is the last empty value in the array. We also have an integer where we store the index of the next free space in the array. This implementation makes sense if we have stacks of relatively **bigger object types**, such as strings and custom classes.

We have implemented the second one in this project.

Here is a visual to make it more clear about how it works:



We also have implemented this stack array such that it accepts generic elements, as this is usually the circumstance where it thrives the most.

```
private final T[] array
```

The array where the data is stored.

```
private final int[] top
```

The array where we store the indexes for top values in our array. It has the capacity of the amount of stacks.

```
private final int[] next
```

The array where we store either the index of the next free space or the next value in the stack. If it is the last free space in the array or the last value of the array, we store -1 in that index.

```
private int free
```

The index of the first free index in the array.

```
public StackArray(int numberOfStacks, int capacity,
Class<T> clazz)
```

This is the constructor of our class. We pass (in order) number of stacks, capacity of the main array, and the class of our data. We need to pass in class of our data to instantiate the array, as we are using generics.

```
private T[] createGenericArr(Class<T> clazz, int
capacity)
```

This is the method we use to generate the generic array. It returns an array with the specified class and capacity.

```
public void push(T value, int stackNumber)
```

This is the method we use to push a value to a specific stack. If the specified stack is full, it prints “Stack stackNumber is full” and returns. Else, moves the ‘free’ to the next free space, inserts the value in the previous free index, sets the index at the ‘top’ array and sets the index of previous top in the ‘next’ array.

```
public T pop(int stackNumber)
```

This is the method we use to pop a value to from a specific stack. If the specified stack is empty, it prints “Stack stackNumber is empty” and returns. Else, moves the ‘free’ to the space that top occupied, moves the ‘top’ to the next value at the stack and sets the ‘next’ to the next free value at the array.

Assignment 4:

A graph where all the nodes are connected without a cycle and has the least amount of weight is known as a minimum spanning tree (MST).

To convert a graph into an MST, we will use the **Kruskal's algorithm**, which is an algorithm that builds the MST by adding edges in ascending order of their weight, until all the vertices are connected. While adding these edges if we form a cycle, we do not include that edge. Because of the fact that we do not include the cycles, we can say that we will keep selecting until number of edges will be equal to (vertex-1). So, our algorithm will consist of 3 steps:

1. Sort the edges, select the edge with the least weight.
2. Keep selecting from ascending order and check that you do not make a cycle.
3. If all the vertices are connected, stop the algorithm. The selected graph will give you (one of) the MST.

We will be using **disjoint sets (union-find)** to store the graph, as that will make the process most efficient for Kruskal's algorithm. We can know if we make a cycle if we check the vertices we are connecting. We will build up from subsets while making our MST graph, so we can check if that subset contains those 2 vertices, if they do we have a cycle.

```
public class Graph<T>:
```

```
private static class Edge<E> implements
Comparable<Edge<E>>
```

We have this class to represent an edge in our graph structure. It has a weight, and 2 generic typed vertices. It compares the weight values with the compareTo method.

```
public void addEdge(T vertex1, T vertex2, int weight)
```

This method adds an edge to the graph.

```
public Graph<T> getMinimumSpanningTree()
```

This is the method that implements the Kruskal's algorithm. It will return an MST version of the graph.

```
public void printGraph()
```

This method will print each edge in the format of `edge1 <-> edge2: weight'.

```
public class DisjointSet<T>:
```

```
private final ArrayList<T> elements = new
ArrayList<>();
```

This is the list where we hold our elements.

```
private final ArrayList<Integer> parents = new
ArrayList<>();
```
This list holds the parent of each element, in place of its index.

```
private final ArrayList<Integer> ranks = new
ArrayList<>();
```
This list holds the rank of each element, in place of its index.

```
public void union(T element1, T element2)
```
This method will do the “union” functionality of the “union-find” algorithm. It also sets the rank while doing the union.

```
public T find(T element)
```
This method will do the “find” functionality of “union-find” algorithm. It also does path compression every time it’s called.