Youssof Mohammad Attallah

**CU1900127**

Applied Algorithms Coursework Portfolio

Table of Contents

[Introduction: 2](#_Toc103291100)

[Task 1: 2](#_Toc103291101)

[Task 2: 2](#_Toc103291102)

[Task 3: 4](#_Toc103291103)

[Task 4: 4](#_Toc103291104)

[Task 5: 5](#_Toc103291105)

[Task 6: 7](#_Toc103291106)

[Task 7: 8](#_Toc103291107)

[Task 8: 8](#_Toc103291108)

[Conclusion : 11](#_Toc103291109)

[References: 11](#_Toc103291110)

# Introduction:

Advanced algorithms are an important concept for any Computer Science student to understand, and this coursework will show the level of knowledge we have regarding the specified topic. Each task is accompanied by an introduction to the problem, then an explanation to the approach along side snippets of important code parts. However, the detailed code with the comments is available on the git hub repo (: <https://github.com/YoussofAttallah/AdvancedAlgorithmsCourseWord>) and after each task there is the name of the class the answer’s task resides in.

# Task 1:

In this task, we are required to take a binary tree as an input, and duplicate each node, then inserting it in the tree as a left child of the original node using a recursive method. The approach is to from the bottom of the tree instead of the top to avoid the duplicated nodes be duplicated again and the algorithm will then be stuck in an infinite loop.

Text

Description automatically generated

After taking the input from the user, we will call the function. It will mark the original node as old\_left, then call itself on the left child then right child of that node, after the previous function calls are completed it will set old\_left to be the left child of the original node, set the left child of the original node as a new node with the value of the original node, and then set the left child of the left child of the original node to the value of old\_left(which is the left child of the original node) This will result in the duplication of every node without going in an infinite loop.

The full code for this task is in Task1.java

# Task 2:

This task requires us to create a red-black tree from scratch. A red-black tree is a form of balanced trees, but the difference is that the red-black has an extra bit which is colour. The root of the tree is always a black node, a red node can only have black children, the number of black nodes from the root to the leaves is always the same, the height of the tree is only determined by the counting of the black nodes, and that the tree is always balanced since the right side of the tree has the same height as the left side. Compared to AVL trees, red-black trees are more efficient when it comes to insertion and deletion, but the AVL are more oriented for searching the tree. To implement the tree, we need to have insertion and deletion functions. Step 1, to insert, the inserted node must be a red node, if it’s the root then the colour is changed to black and inserted, if not then we check the parent. Step2, if the parent is black then we insert as a child, if the parent is red then we have step 3 with two scenarios; the first is that if the uncle is red, so we change the colour of the parent and uncle to black, the grandparent to red, then we go to the grandparent node and repeat steps 2 and 3. The second scenario for step 3 is if the uncle is black, we rotate the tree so it gets balanced. For the removal of a node, we have multiple scenarios. The first is if the node is a lead or has only one child, then it can be deleted, and its child replaces it. If not, then the inorder predecessor of that node is removed, and if that node was red nothing else happens, if its black with a red child then the red child takes the black node place and changes its colour. For this task we have a sample array to test.

Sample array after insertion:

Text

Description automatically generated

Tree after deleting 45:

Text

Description automatically generated

The full code for this task is in Task2.java

# Task 3:

In this task, we have will have the user input an unsorted array and then use the heapsort algorithm to output the sorted array. The algorithm has a time complexity of O (N log n), similar to Merge sort, but unlike the merge sort which requires more temporary memory, the heapsort does not require any additional space. It works by first constructing a binary tree, then turning a heap (either minimum heap where the root is the smallest number or a maximum heap where the root is the biggest number). After that the root of the tree is removed and inserted into the sorted list, the tree then gets heapified so the smallest/largest number is now the root of the tree. The process is repeated until the tree is empty and the result is a sorted list. A class HeapSortClass is created with attributes of array which has the numbers from the user and array\_length which is how many elements are in the array. Then the function heapsort is called, and what it does is that it loops array\_length/2 -1 times, which is equivalent to the number of nodes that have branches (all nodes except leafs) and heapifies them to find the smallest node and make it the root starting from the bottom to the top. After that is done, it takes the root element and excludes it, after that it heapifies the list again unit the heap is empty, and the list is then printed in order.

Resulting test:



The full code for this task is in Task3.java

# Task 4:

This task requires to take a few numbers and use the hash function to add them to the table using two methods, Linear Probing and Quadratic Probing. To start with the Quadratic Probing, it applies the hash function and if an insertion fails the first time, it adds 1\*1 to the value and then applies the hash function again, if the insertion fails again, it adds 2\*2 to the original value and applies the hash function again. This is repeated, changing the added value to 3\*3, 4\*4 and so on for every failed insertion until its inserted. In this question, we have a table size of 7, and the values (12,5,19,2,23), with a hashing function of k modulo table size. The first value to be added is 12, 12 mod 7 equals to 5, thus adding 12 to index 5 in the hash table. Next, we have 5, 5 mod 7 equals to 5, which causes a clash with 12 we just added earlier, so the Quadratic Probing requires us to add 1\*1 to the original value, so 5 plus 1 is 6 and 6 mod 7 is 6 so 5 is set into index 6 of the table. Third we have 19, so 19 mod 7 is 5 and it causes a clash, so 19+1 is 20 mod 7 is 6 which is also clash, then 19+2\*2 is 23 mod 7 is 2 which is empty so 19 is added to index 2. After that we have 2, 2 mod 7 is 2 which is a clash with the 19, then we apply 2+1 is 3 mod 7 is 3 and that index is empty so 2 is added to that index. Finally, we 23, 23 mod 7 is 2 which is a clash, so 23+1 is 24 mod 7 is 3 which is also a clash, then 23+2\*2 is 27 mod 7 is 6 which also happens to clash, we add 23+3\*3 which is 32 mod 7 is 4 which is free, resulting in all the values being inserted in the table.

Quadratic Probing Table:

|  |  |
| --- | --- |
| Index | Value |
| 0 |  |
| 1 |  |
| 2 | 19 |
| 3 | 2 |
| 4 | 23 |
| 5 | 12 |
| 6 | 5 |

Next, we have linear probing, it calculates the index using the hash function, and if the insertion failed it will go to the next index until it is inserted successfully. In this example we table with a s size of 10, values (9,18,19,27,8,17), and a hashing function of k mod table size. So, we start with 9, 9 mod 10 is 9 so it’s inserted at index 9. Then we have 18, 18 mod 10 is 8 so it’s inserted at index 8. After that, 19 mod 10 is 9 which causes a clash, so we go to the next available index is 0 so its inserted there. Next 27 mod 10 is 7 so its inserted there. After that we have 8, 8 mod 10 is 8 which causes a clash, the next indexes are 9, and 0 which both are occupied so it’s inserted at the next available index which is 1. Finally, we have 17, 17 mod 10 is 7 which is occupied, indexes 8,9,10,0, and 1 are all unavailable so it is placed at index 2.

Insert Linear Probing Table:

|  |  |
| --- | --- |
| Index | Value |
| 0 | 19 |
| 1 | 8 |
| 2 | 17 |
| 3 |  |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 | 27 |
| 8 | 18 |
| 9 | 9 |

# Task 5:

This task focused on representing a graph using 2 methods, the first being an adjacency matrix and the second being the adjacency list. Both representing the nodes and their connections to each other in the graph. The adjacency list representation is used to show a node, and then show a list of all the other nodes connected to the original node. The adjacency matrix is a representation that shows a matrix with all the nodes shown, and the intersection of two nodes shows if there a connection or not.

Diagram

Description automatically generated

Since the graph given in the example is a weighted graph, some changes in the general implementation will be applied. The adjacency matrix will have the weight of the connection be represented instead of the one that was used to represent if they were connected at all or not, and instead of zero we will have negative 1 indicating that the nodes are not connected. Each node will have an index (A is 0, B is 1 etc.)

For the adjacency matrix approach, we start by creating the AdjacencyMatrixGraph class which has two attributes, the total number of nodes, and a 2-d array that is size is set to the number of nodes in the graph and initialize all its values to negative one, then we start by adding the edges to the matrix using the add\_edge function, this function takes three arguments (start, end, weight). After all nodes are added the resulting matrix is printed with the print\_graph function. The resulted matrix is configured where we set the rows to be the source, the columns to be the destination, and the intersection being the weight of the connection between the two nodes.

A picture containing text, keyboard

Description automatically generated

For the adjacency list approach, we will have a linked list that has all the adjacent nodes for every node. We start by creating the AdjacencyListEdge class which will be the node in the linked list, this class has three attributes (start, end, weight), then we have the AdjacnecyListGraph class itself which has two attributes (nodes\_total, adjacency\_list). After that, we initialize the adjacency\_list which is a linked list with the size of the total number of nodes, then we loop around the adjacency\_list and initialize each element to be a linked list of itself to create a nested linked list. The first linked list represents each node and the second linked list is for each node and represents the adjacent node. For each edge we use the add\_edge function which takes the same parameters as the one in the adjacency matrix, it takes the start node, finds its place in the adjacencly\_list linked list using the index, then adds the node to the second internal linked list. Finally, the adjacency list is printed representing each connection and the weight.

Text

Description automatically generated

The full code for this task is in Task5.java

# Task 6:

In this problem, we are required to implement the Breadth-First and the Depth-First searching algorithms, and then test them using the graphs we have created back in Task 5. The Breadth-First algorithm traverses a tree or a graph by levels, it does that by starting at the root, then it finds all adjacent nodes and explores them all before moving onto the next level, while the Depth-First starts at the root and explores to the maximum depth before backtracking. We will apply both algorithms using the Adjacency List we defined in the previous task.

To start with the BFS, we create an empty array sized according to the number of nodes to indicate whether a node has been visited or not. A queue, implemented using a linked list, is used to add the nodes and remove them in order of first in first out. We add the start node to the queue and mark that node as visited, then we enter a while loop that checks whether the queue is empty or not. If there are elements in the queue, it will take the top element then pop it out of the queue then extract the adjacent nodes to it from the adjacency list. These nodes are checked whether they were visited or not, if they are un-visited, then they are marked as visited and added to the queue, if they were visited then the node is skipped.

Order when starting with 0:



Order when starting with 1:



For the DFS, we will have a recursive approach, so it reaches the end of the graph then starts to backtrack. We will have the DFS function, which takes the start node as an input, initializes the visited array, and then calls the main function DFSUtil and gives it the start node and the visited array. The DFSUtil with then mark the node as visited, print it out, then get all the nodes adjacent to the start node and loop on them. If the node is not visited it will be marked as visited, then the DFSUtil is called again given the new node and the updated visited array.

Output when starting with 0:



Order when starting with 1:



The full code for this task is also in Task5.java, the BFS, DFS, and DFSUtil are all in the AdjacencyListGraph.

# Task 7:

In this task, we are given a graph and required to use prim’s algorithms to find the maximum spanning tree and output the results. In general, prim’s is used in undirected weighted graphs to find the minimum spanning tree, but here we will be modifying the algorithm to find the maximum. We will be implementing the algorithm using the adjacency matrix discussed in Task 4 but tested using a new undirected graph.

To start, we create an array of visited nodes, set the starting node to 0, and set the visited value for the index of the starting node to 0, then we enter a while loop that exists when the starting node number reaches the number of total nodes (the starting node is incremented every cycle). Normally we will be having a minimum value (set to a high number which is the Integer.MAX\_VALUE in java) that we compare the weights to so we can find the minimum, but here we will be switching it and having a maximum (set to a low number which is Integer.MIN\_VALUE in java) to get the biggest weight. We loop around the visited array to find a visited node, if yes, we then go into another loop which loops around the visited array to find unvisited nodes that are connected to the original node, we loop and compare each weight to the maximum, if its bigger than the set maximum we change it so the new weight becomes the new maximum. After that we print the nodes and the weight, set the new node that was the maximum in the visited array to be true, then we increment the start node number and repeat the process again.

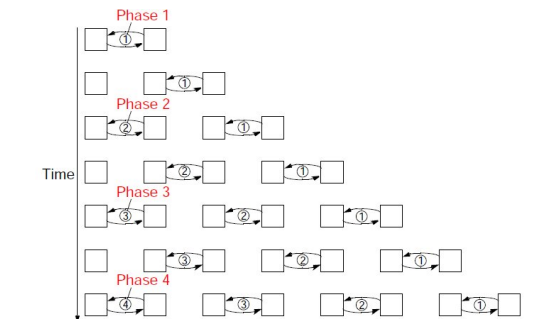
A picture containing text

Description automatically generated

The full code for this task is also in Task5.java, prim,s algorithm is in the AdjacencyMatrixGraph. And its testing has another input( the one provided in the coursework) which is inserted in another object (AMG2)

# Task 8:

Multithreading is the basis of parallel computing as it allows multiple processes to be simultaneously running at the same time, unlike when there is one core/thread where for multiple processes seem to be running simultaneously but, there is only one task is running and tasks switch out of the CPU using context switching which is handled by the operating system. In multithreading, if the code has multiple functions sequentially but aren’t dependent upon each other, then reformatting the code to allow it to be multithreaded allow them to be ran in parallel and utilizing the multi-core CPUs that are now the standard in every CPU architecture. One way to utilize this concept is in some sorting algorithms, if an algorithm works by dividing the incoming list into parts and then apply a function to solve them then it can be multithreaded as there is no need to wait for each part to finish and then combined, they can all be solved simultaneously then combined at the end. Examples of this are the Parallel Bubble Sort algorithm. The original algorithm takes the first element in a list and compares it the next element to see which is bigger than switch if the first element is bigger than the second and keeps on comparing the new second element to the third and so on so after the first loop the largest element is at the end of the list, then the process is repeated for the second largest and the third until the list is sorted. In the parallel version, the second loop can begin even if the first isn’t finished, and then the third can begin even if both the previous loops aren’t completed, running these iterations in a pipeline fashion.



Another algorithm is the quicksort algorithm. This algorithm uses a divide and conquer approach by selecting a pivot number, and the dividing the list into two subsections where in the first subsection all elements are smaller than the pivot while in the second the elements are larger than the pivot. This process is repeated for every sub-list created until the elements are sorted.

A picture containing text

Description automatically generated

Its parallel version, it utilizes the tree structure of the algorithm and assigns a process to each of the newly created lists. Thus, allowing for the division of these lists to go faster.(Rocha & Silva, n.d.)

Diagram

Description automatically generated

The final algorithm we will talk about is the parallel merge sort. The original algorithm also uses the same divide and conquer approach, it works by splitting the list into two halves then splitting the resultant lists into twos again until single elements are remaining, then they are ordered and merged until the final list is sorted. The algorithm starts by determining the number of cores in the CPU that its running on and the number of elements in the array, if they divide perfectly then all cores have the same workload, if not, then it’s divided upon number of cores-1 and the remaining workload goes to the last core. the next step is that we divide the list into subsets and a new thread is created for each subset and then the threads apply the merge sort algorithm to each of these subsets concurrently and after that the threads are joint together to have all the subsets done and sorted. Finally, the sorted subsets are then merged so the final sorted array can be printed.

The full code for this task is in Task8.java

# Conclusion :

To conclude, we have passed by concepts and implementations of multiple algorithms, discussed them and implemented them along side testing them with some scenarios. This approach has solidified my understanding of these topics that will definitely be an important asset on my journey.

# References:

Rocha, R., & Silva, F. (n.d.). *Parallel Sorting Algorithms*. 41.