**Q1b Justification**

Breadth First Search (BFS) is one of the programming techniques to visit unvisited nodes, mainly the parent and child nodes and then present the output according to the flow of visited nodes. The 15 values are arranged in a perfect binary tree structure with four levels. In a perfect binary tree, BFS will visit each node according to its level.

According to the code, 16 values are initialised in the array named as list. The first element, list[0]=15, shows the size of the array. The address of list is loaded into Register R0. Then, R0 is stored in R10 while R1 is stored in R11 because sorting procedure changes size and address. After all the nodes have been visited, the sorted list will be shown in memory window. R4 will be used the find element in binary tree through find function.

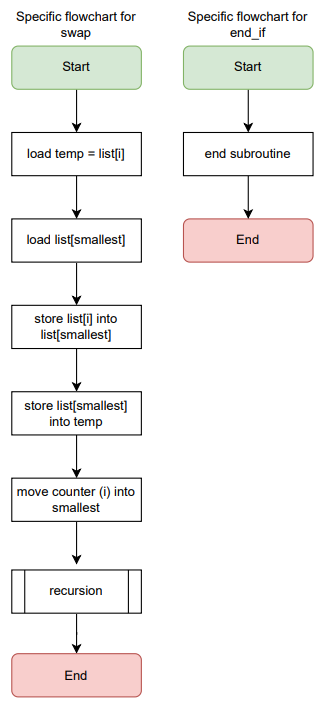
Next, the program will call sorting procedure which will then sort the array. It starts off with push link register on the stack to put data in memory. Then, R9 will be used as counter so that it will end procedure once all 15 values have been visited. The size will be stored in R12 which is used as a local parameter. In the function sorting\_for loop, it will first compare R12 to R19 to check whether it reaches the last value. If condition is met, it will end sorting by pop program counter that removes the data from the memory. If condition is not met, it will go to build\_val procedure. In this build\_val procedure, size of array is divided by two, minus by 1 and stored in R8. This part is to build array back into binary tree to compare the children and parent node. The smaller one replaces the parent.

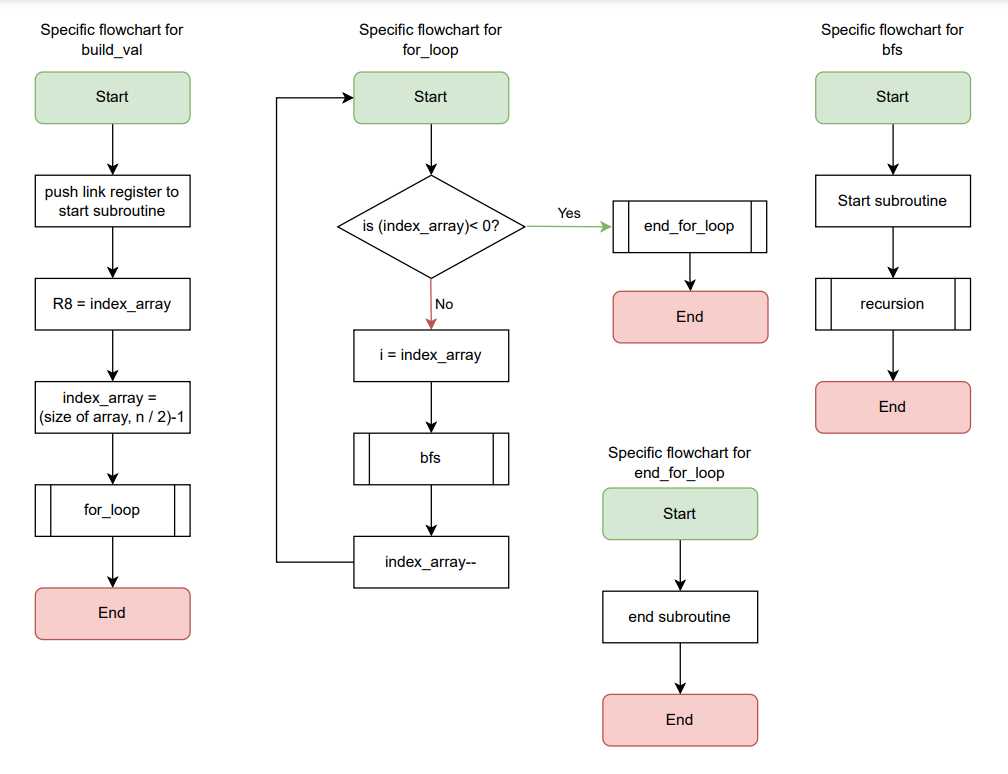
Then, R8 is compared with 0. If the value is lesser than zero, then loop ends, otherwise the value of R8 will be stored in index R2 and it will call the bfs procedure. After that, recursion will occur. The value of R2 will be stored in R3 which is used to store the smallest value in the array. R2 will be multiplied by 2 and then added with one. The value is then stored in R4 which is the left node of the binary tree as smaller value is at the left. R5 will store greater value which is at the right. The difference between R4 and R5 is we add 2 to R5 and 1 to R4 to make sure that bigger values are always store at the right nodes while smaller values at the left nodes. If R4 is larger than or equal to R1, it will start off by comparing R5 and R1 to check if it is greater than or equal to the size. If yes, then it will jump to right\_node. In right\_node, if R3 which is smallest is greater than R2, then if-condition ends and return to where it was called. However, if R4 is not bigger than R1 then it loads the value that the left value into R6 and value stored in smallest into R7. Then, it will compare if R6 is greater than or equal to R7, then it will jump to left\_node condition, else the value of smallest which is R3 is now R4 which store the current value in the left side of the binary tree. Besides, if R5 is not greater than or equal to R1, then it stores current value at right node in R6 and current smallest value in R7. If R6 is greater than or equal to R7, then jump to right\_node and if not, then R3 will store the same value as R5. In right\_node, if R3 is not greater or equal to R2, it swaps the position of the value in memory. In this situation, R6 becomes temporary storage to store current value in R2. Finally, current index of the element pointed by R2 will be the same as the index in R3. Then, the recursion happens until all the elements have been visited, subtract R1 by 1 to decrease size by 1 and add R9 which is counter by 1.

After that, ‘find’ procedure starts to put back the deleted element from the array list. Then, it will print out the sorted value in the memory window and it can be check by putting address, R10. After it is done, the procedure end where it will be at pass loop and program is completed. Hence, the above explanation has clearly shown how BFS is applied in sorting by using heap sort algorithm which uses binary tree and visit each node in the binary tree.

**Q1b Flowchart**

Diagram

Description automatically generatedDiagram

Description automatically generatedDiagram

Description automatically generated

Diagram

Description automatically generatedDiagram

Description automatically generated

Find and Search element to display back the sorted list in memory map

**Q2b. Justification**

Merge sort is defined as a divide-and-conquer sorting algorithm. For question 2a, an iterative merge sort was used to sort the 10 elements given. The array is repeatedly divided into halves until each sub-arrays obtained contain only one element. After the divisions are made, the sub-arrays with one element are sorted and merged to form pairs. These one-dimensional structures (sub-arrays) are repeatedly merged until a sorted array of the original size is achieved, as per Figure 1 below. When each merge is performed, the elements of the sub-arrays participating in the merging process are sorted to form a bigger sorted array from two unsorted sub-arrays.

For reasons of optimization, in this implementation of merge sort, the divisions conducted in a traditional merge sort are done virtually, meaning the code provided does not explicitly create the required sub-arrays. Rather, each element in the array is initially treated as a sub-array containing one element. This is because no matter the size of the array, each element in the array will be split into its sub-array after the division process. These sub-arrays are then merged with adjacent sub-arrays of the same size (size 1) to form sorted sub-arrays of size 2. Afterwards, merge sort continues normally; smaller unsorted arrays are merged to form larger sorted arrays until all elements are sorted in ascending order.

The first compare and branch statement group is used to iterate over the levels to merge the sub-arrays until an array of the original size is obtained. The size of the sub-arrays doubles with each iteration. The counter variable, I, stored in register 2 (r2) defines the index of the first element of each sub-array in a particular level, while the variable (j) stored in register 3 (r3) is used to iterate over each element in a sub-array starting at the ith element of the array. The counter variable (k) is used to do the required searches and insertions to perform the sorting of the merged sub-arrays. Register 11 (r11) and register 12 (r12) are used to store pointers to the kth and jth elements of the array respectively. When calculating the address of the jth element, register 11 is initially set to the address of the first element of the array. The count j is added to this address to get the address of the jth element. Register 11 is set to the address of the kth element of the array, while Register 9 (r9) is set to its value, to perform the required comparisons. Therefore, when the counter variable k is incremented, the pointer variable for the kth element also needs to be updated.

A screenshot of a computer

Description automatically generated with low confidence

Figure 1

**Flowchart for Merge sort**

**Q2b Flowchart**

Diagram

Description automatically generatedDiagram

Description automatically generatedDiagram

Description automatically generatedDiagram

Description automatically generatedDiagram

Description automatically generatedDiagram

Description automatically generated