Part 1

1a

What merging and sorting algorithms have you considered?

I chose the insertion sort out of the sorting algorithm’s we have covered in class as opposed to merge sort (which has an average case complexity of O(nlogn)) because, I had previously decided against working with dynamic data structures which is needed to take full advantage of merge sort. While insertion sort’s worst case complexity is O(n^2), it is the most efficient of the other two sorting algorithm’s we learned (insertion and bubble) because it’s adaptive for data sets that are already substantially sorted making it have a possible time complexity of O(*n* + *d*), where *d* is the number of inversions. This makes it more efficient than the two aforementioned algorithm’s because it gives the algorithm a potential best case of O(n).[1] The merging algorithm I chose is a modified version of the mergesort’s merge algorithm which allows two sorted arrays to be merged into a larger array.[2]

For my second solution for implementing merging and sorting of the students I decided to use quicksort along with the same merging algorithm I decided on in solution 1 (the modified merge sort merge function.

A major reason why I chose this algorithm is because quicksort requires no dynamic allocations. It can work in-place on the original array, using only O(log n) stack space (worst-case if implemented correctly) to store the stack frames necessary for recursion. This lines up with my choice not to implement dynamic data structures. Also quicksort will have a very small number of cache misses, which on modern architectures is critical for performance. The only tradeoff with using mergesort is the possibility that it degrades to O(n2), which on large data sets can be very serious.[3]

1b What data structure(s) would be most suited?

After thorough analysis I decided to use a non dynamic array in conjunction with records for the data structures.

I first weighed up the positives and negatives of implementing an array or a linked list as well as a linked list of a stack or queue with that of an array implementation. I first checked the positives/negatives of using an array vs linked list [4] where I found a simple breakdown of when to use one over the other.

I came to the conclusion that array implementation would be best suited for the assignment because I will need indexed/random access to the elements when sorting them, I know the exact number of students needed to sort and merge, I will be iterating through all the elements in sequence when sorting and merging and while the number of elements is unlikely to lead to memory problems, cutting down on memory use will make the program more efficient.

The stack and the Queue are more advanced ways to handle a collection than the array itself, which doesn't establish any order in the way the elements behave inside the collection. I felt that they weren’t needed, given the required task because they have the same basic trade off as any array vs dynamic data structure.

While a linked queue/stack has flexible high speed insertions/deletions when implemented, it requires more storage than an array. Insertions/deletions are inexpensive at the ends of an array until you run out of space an array implementation of a queue or stack will require more work to resize, since you'd need to copy the original into a larger structure

I decided to use record structure’s for storing each individual’s students personal information (student id, name, previous college)

1c. Explain/illustrate both solutions.

Solution 1

Records will be created for each of the four colleges (TU, DIT, ITT, ITB). Each record will contain the students name, id and college. (Aside from the TU record which will contain their name, id and previous college). Each college will be assigned an integer array with size corresponding to the number of students in the year for each year of each college, i.e. the DIT record will be assigned an integer array for year 1 called dit\_year 1 with a size of 50 to hold the 50 students. These records will then be filled in by file input from an external file containing the details of all the students.

The program will then call the insertion sort.

Once inside the insertion sort algorithm, the function will be sorting the student elements by their student id’s in order from lowest to highest by comparing two indexes against each other iteratively and sorting them. This function will be called for each year of each of the different college’s records. This will be done in the order DIT(years 1-4), ITT(years 1-4) and ITB(years 1-4). i.e. dit\_year1’s record will be the first to be called into insertion sort. Once all years of all three college’s have been sorted, the merge function will be called. The merge function will first merge DIT’s year 1 record with ITT year 1 record into the TU record. This process will then repeat for year 2- 4 of the two college’s. A copy of the TU record will be made and merge will then be called again, this time to merge ITB with the TU record copy into the original TU record. This is to avoid overwriting addresses when merging into the TU record’s array.

Solution 2

Records will be created for each of the four colleges (TU, DIT, ITT, ITB). Each record will contain the students name, id and college. (Aside from the TU record which will contain their name, id and previous college). Each college will be assigned an integer array with size corresponding to the number of students in the year for each year of each college, i.e. the DIT record will be assigned an integer array for year 1 called dit\_year 1 with a size of 50 to hold the 50 students. These records will then be filled in by file input from an external file containing the details of all the students.

The program will then call the quick sort. Quick sort will sort the student elements by their student id’s in order from lowest to highest by finding the middle element of all the student id’s and making this the pivot and comparing the student id’s of the far left and far right elements against the pivot element. This continues to happen causing the bigger element to be pushed to the right side and lesser element to the left side. This function will be called for each year of each of the different college’s records. This will be done in the order DIT(years 1-4), ITT(years 1-4) and ITB(years 1-4). i.e. dit\_year1’s record will be the first to be called into quick sort.

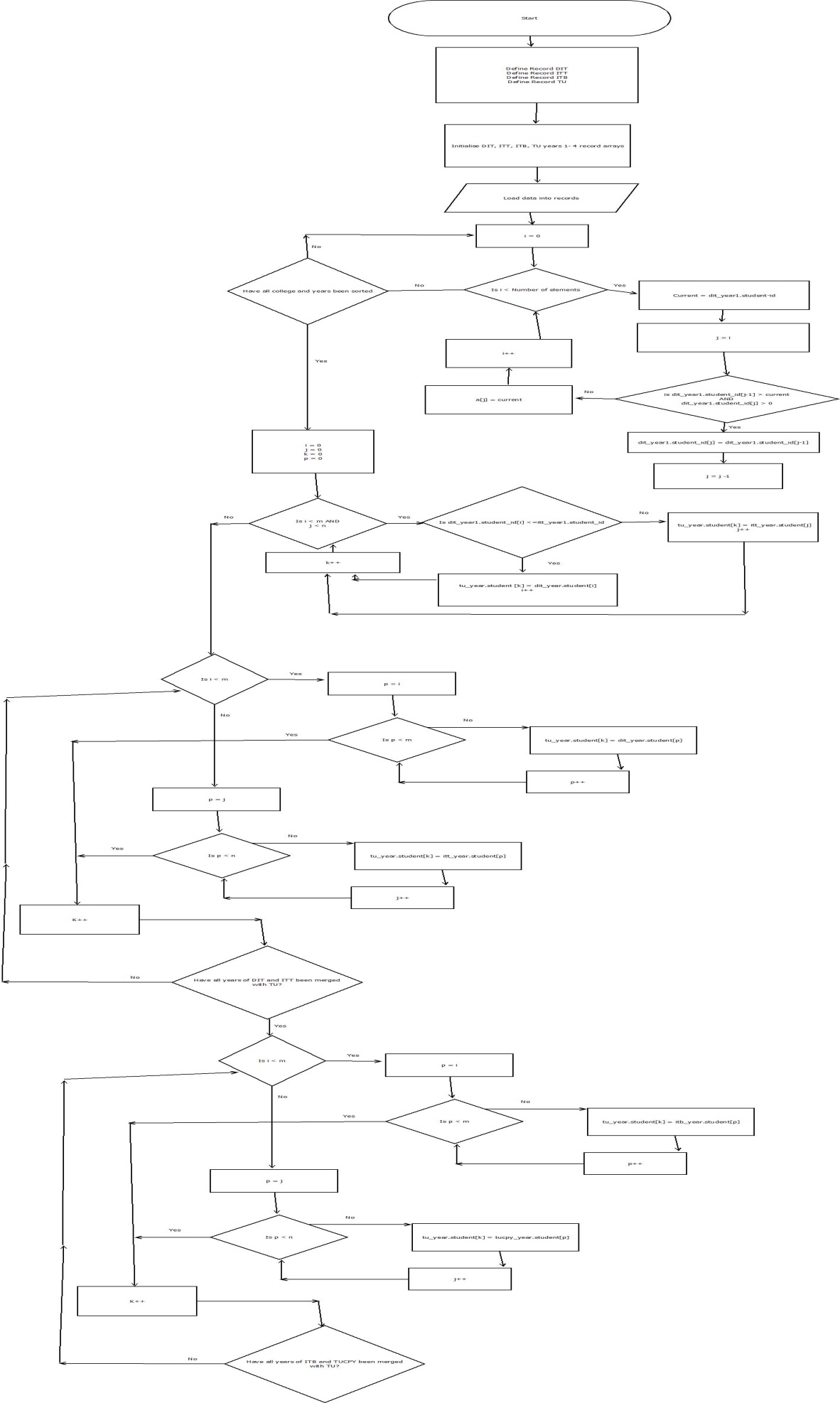
Once all years of all three college’s have been sorted, the merge function will be called. The merge function will first merge DIT’s year 1 record with ITT year 1 record into the TU record. This process will then repeat for year 2- 4 of the two college’s. A copy of the TU record will be made and merge will then be called again, this time to merge ITB with the TU record copy into the original TU record. This is to avoid overwriting addresses when merging into the TU record’s array.

1.d Justify which solution should be used.

While it is generally agreed upon that quicksort is best implemented when dealing with larger elements, I opted to choose my insertion sort solution for a number of reasons. Firstly, it is the solution with which I feel most comfortable with using. While I understand how quicksort works to a large extent, I find the simple design of quicksort much more easier to work with and implement. Insertion sort has low overhead, it can be written fairly succinctly, and it has two key benefits: it is stable, and it has a fairly fast running case when the input is nearly sorted. While in its worst case, insertion sort has a big O of O(n^2) it is more than likely that the student id’s will be entered in a semi-sorted state, the insertion sort could run as fast as linear time O(n). I didn’t feel the size and type of the problem set necessitated the use of the quick sort algorithm

1.e Write an algorithm for the best solution described in 1c.

1. Initialize college records
2. Set variables for years 1 – 4 of each college
3. Load info file for each year of each record
4. Initialize insertion sort function
5. For i = 1, i < NumElements do {
6. current = collegeandyear.student\_id[i]
7. j = i
8. while collegeandyear.student\_id [j] > 0 AND collegeandyear.student\_id [j – 1] > current
9. collegeandyear.student\_id [j] = collegeandyear.student\_id [j-1]
10. j = j-1
11. End while
12. A[j] = current
13. End for
14. Repeat steps 5 – 13 with three colleges for all years
15. Initialize 1st merge function
16. merge(m, n, dit\_year[], itt\_year[], tu\_year[])
17. i = 0
18. j = 0
19. k = 0
20. P = 0
21. while (i < m && j < n)
22. if (dit\_year.student\_id [i] <= itt\_year.student\_id [j])
23. tu\_year.student [k] = dit\_year.student[i]
24. i++
25. else
26. tu\_year.student[k] = itt\_year.student[j]
27. j++
28. End if
29. k++
30. if (i < m)
31. for p = I to p < m
32. tu\_year.student[k] = dit\_year.student[p]
33. End for
34. K++
35. else
36. for p = j to p < n
37. tu\_year.student[k] = itt\_year.student[p]
38. End for
39. K++
40. End if
41. End while
42. Repeat steps 20 – 40 with all years
43. Copy tu\_years1-4 records into tucpy\_years1-4 records
44. Initialize 2nd merge function
45. merge(m, n, itb\_year[], tucpy\_year[], tu\_year[])
46. i = 0
47. j = 0
48. k = 0
49. P = 0
50. while (i < m && j < n)
51. if (itb\_year.student\_id [i] <= tucpy \_year.student\_id [j])
52. tu\_year.student[k] = itb \_year.student[i]
53. i++
54. else
55. tu\_year.student[k] = tucpy \_year.student[j]
56. j++
57. End if
58. k++
59. if (i < m)
60. for (p = i; p < m; p++) {
61. tu\_year.student[k] = itb \_year.student[p]
62. End for
63. K++
64. else
65. for (p = j; p < n; p++)
66. tu\_year.student[k] = tucpy \_year.student[p]
67. End for
68. K++
69. End if
70. End while
71. Repeat steps 44 – 69 for all years

Flowchart of Algorithm

Part 2

Explain *one* good way of searching for a student who started their degree in IT Tallaght and is now in TU Dublin:

With the now sorted tu record from part 1 this could be implemented linearly through a linear search which will simply go through a for loop from elements current – size of all the records on tu record file read from header of file for each year. The program will prompt the user for the previous user id of the student or their nam, the loop will continue to run until it finds a student who’s id or name matches the user entered name or id.

Algorithm

Int I

Input id or name

for i = 0, i < total records do

if tu\_year1 [i]->Student\_id = ‘Id’

OR

If tu\_year[i]->First\_Name = ‘Name’

current = i

counter++

return i

i++

Repeat for all years

References

[1] <http://en.wikipedia.org/wiki/Insertion_sort>

[2] <http://www.algolist.net/Algorithms/Merge/Sorted_arrays>

[3] http://en.wikipedia.org/wiki/Quicksort

[4] http://stackoverflow.com/questions/393556/when-to-use-a-linked-list-over-an-array-array-list