Algorithms and Data Structures – Assignment 1

Module Code 55-400212

* 1. Consider the following algorithm fragment written in ADL and calculate its complexity with justification.

for i ← 1 to n by 1 do

  for j ← 1 to i by 1 do

    for k ← 1 to j by 1 do

      x = x + 1

    end

  end

end

The complexity of the algorithm is n3.

If we look at the algorithm, we see three nested for-loops. If we name the loops *a*, *b*, and *c* from the bottom up and the code inside the last loop *x* then we can describe the running time of the algorithm segment as such:

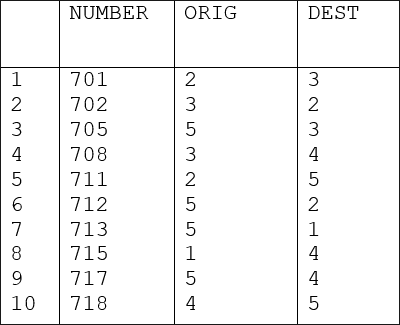
(((*xt* \* *at*) \* *bt*) \* *ct*)

Since we are describing the complexity, and the block of code nested inside the loops is of constant complexity and not dependent on the ever-exponentiating for loop conditions, we can discard *x*;

((*at*) \* *bt*) \* *ct*)

Since the number of iterations of *a* depend on the current iterative value of *b*, and likewise between *b* and *c*, we can say that in the context of describing the complexity, the total number of iterations is *n \* n \* n*, because *n* determines the number of iterations for loop *c* and the other loops are dependent on that. Conclusively, n \* n \* n is more simply written as **n3**.

* 1. Consider the data shown below, which gives the different flights of an airline. Discuss different ways of storing the data so as to decrease the time in executing the following:



* + 1. Find the origin and destination of a flight, given the flight number

The data is already stored in a fairly ideal way to do this; the process of execution could be sped up by storing the name of the origin and destination city directly in the table, but this would take up a lot more storage space and be far less maintainable/manageable due to repetition of data as opposed to referencing unique items many times (using relationships).

* + 1. Given city A and city B, find whether there is a flight from A to B, and if there is, find and return its flight number

One way to achieve this faster would be to have a table for each city which lists departures from there; then one could simply filter for the destination and look at the flight number.

Such a table could look like this:

|  |  |
| --- | --- |
| **Departures from Boston** | |
| Destination | Flight Number |
| 3 | 701 |
| 5 | 711 |
| 4 | 746 |

Again, while this would reduce the execution time of certain operations, a massive amount of storage would be required – 2 extra cells just for every flight. In a system of thousands of flights a day, this would quickly amount to a lot of storage used for little increase in execution speed.

1. Consider the following the data structure and three procedural abstractions that support it:

declare DATA [1..20], top, n

procedure abstractionONE(OUT top, OUT n)

top ← 0

n ← 20

end

procedure abstractionTWO(IN item, IN n, IN top, INOUT DATA[])

if top ≥ n then CALL DATA\_FULL()

else

top ← top + 1

DATA(top) ← item

endif

end

procedure abstractionTHREE(OUT item, IN top, INOUT DATA[])

if top ≤ 0 then CALL DATA\_EMPTY()

else

item ← DATA(top)

top ← top - 1

endif

end

Consider further that the following application has been written:

1. CALL abstractionONE(top, n)
2. read (item1, item2, item3, item4)
3. CALL abstractionTWO(item1, n, top, DATA[])
4. CALL abstractionTWO(item2, n, top, DATA[])
5. CALL abstractionTWO(item3, n, top, DATA[])
6. CALL abstractionTHREE(item, top, DATA[])
7. print(item)
8. CALL abstractionTHREE(item, top, DATA[])
9. print(item)
10. CALL abstractionTWO(item4, n, top, DATA[])
11. CALL abstraction THREE(item, top, DATA[])
12. print(item)
13. Pretend to be a processor and execute the above algorithms top-down (i.e. by starting from the application) by using the code walkthrough technique taught in lectures.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Line #** | **top** | **n** | **data[]** | **item1** | **item2** | **item3** | **item4** | **item** |
| **1** | 0 | 20 | [-] | - | - | - | - | - |
| **2** | 0 | ^ | [-] | 1 | 2 | 3 | 4 | - |
| **3** | 1 | ^ | [1] | ^ | ^ | ^ | ^ | - |
| **4** | 2 | ^ | [1, 2] | ^ | ^ | ^ | ^ | - |
| **5** | 3 | ^ | [1, 2, 3] | ^ | ^ | ^ | ^ | - |
| **6** | 2 | ^ | ^ | ^ | ^ | ^ | ^ | 3 |
| **7** | ^ | ^ | ^ | ^ | ^ | ^ | ^ | ^ |
| **8** | 1 | ^ | ^ | ^ | ^ | ^ | ^ | 2 |
| **9** | ^ | ^ | ^ | ^ | ^ | ^ | ^ | ^ |
| **10** | 2 | ^ | [1, 4, 3] | ^ | ^ | ^ | ^ | ^ |
| **11** | 1 | ^ | ^ | ^ | ^ | ^ | ^ | 4 |
| **12** | 1 | 20 | [1, 4, 3] | 1 | 2 | 3 | 4 | 4 |

1. What does this application do? Give a more meaningful name to each abstraction?

The data structure and supporting abstractions simulate a sort of tape where it can be written to and moved across, or read from and moved back the other way.

abstractionONE is simply used to set default values for some variables, so could alternatively be called something like ‘setup’ or ‘initiateValues’.

abstractionTWO writes to the array of values and shifts the position forwards by 1 (stored in ‘top’; the highest index that relevant data is written to). It could be called something like ‘writeAndShiftRight’, or simply ‘write’.

abstractionTHREE reads from the array and shifts the position backwards by 1. It could be called ‘readAndShiftLeft’, or simply ‘read’.

The application shown simply writes and reads some values; the final result is numbers 1, 4 and 3 in the array and numbers 3, 2 and 4 printed out.

1. In the light of your answer in (ii), devise an algorithm as a functional abstraction in ADL that determines whether or not DATA[] is empty.

procedure isDataEmpty(IN DATA[], IN n, OUT empty)

declare pos, empty

pos <- n

empty <- true

for i <- n to 0 by -1 do

if DATA(i) is not null do

empty = false

endif

end

end

1. How can you incorporate the functional abstraction you devised in (iii) into the application above? Modify the above application to demonstrate your answer.
2. CALL abstractionONE(top, n)
3. isDataEmpty(DATA[], n, empty)
4. if empty is true do
5. read (item1, item2, item3, item4)
6. CALL abstractionTWO(item1, n, top, DATA[])
7. CALL abstractionTWO(item2, n, top, DATA[])
8. CALL abstractionTWO(item3, n, top, DATA[])
9. else do
10. top = 3
11. endif
12. CALL abstractionTHREE(item, top, DATA[])
13. print(item)
14. CALL abstractionTHREE(item, top, DATA[])
15. print(item)
16. CALL abstractionTWO(item4, n, top, DATA[])
17. CALL abstraction THREE(item, top, DATA[])
18. print(item)
19. Suppose a secondary school keeps a record for each student which contains the following data: ID, which is unique for each Student, Name, Telephone Number, Father, Mother. Here, Father and Mother contain, respectively, the names of the student's father and mother.

Eight such records may be as follows:

**ID Name Telephone Father Mother**

ID1 Adams, John; 01142253175 Richard Mary

ID2 Bailey, Susan; 01142256900 Steven Sheila

ID3 Clark, Bruce; 0161248653 XXXX Barbara

ID4 Adams, Jane 01142253175 Richard Mary

ID5 Adams, William 01142253175 Richard Mary

ID6 Adams, Donald 01142253175 Richard Mary

ID7 Clark, David 0161248653 XXXX Sheila

ID8 Clark, Lisa 0161248653 XXXX Sheila

Here, XXXX means that the parent is not living with the student. The following operations may be applied to the above data:

* Add a new student record
* Delete a student's record
* Given the names of a father and mother, print their children who attend the school

In this context, carry out the following task:

1. Design a set of data structure(s) that will hold data similar to the ones above and justify the purpose of each data structure in your set. If you are able to come up with alternative data structures and choose the most appropriate one based on time-space trade-off, then additional marks will be awarded.

The most straightforward way to achieve this is to have a set of one-dimensional arrays; one per table column. This way, the data stored in the table could be directly translated to array storage record-by-record, by adding one value to 5 different arrays (5 different attributes in the table). This would look something like…

declare ID[1..20]

declare NAME[1..20]

declare TELEPHONE[1..20]

declare FATHER[1..20]

declare MOTHER[1..20]

ID(1) <- ID1

NAME(1) <- “Adams, John”

TELEPHONE(1) <- “01142253175”

FATHER(1) <- “Richard”

MOTHER(1) <- “Mary”

STUDENT\_ID1\_NAME <- NAME(1)

Alternatively, a two-dimensional array can be used. This would achieve a slightly greater efficiency since all of the data is stored together rather than in separate memory objects. Additionally, the data would be easier to read in an algorithm as only one array must be referred to in order to access any attribute of any record. The two-dimensional array solution here would look like…

declare STUDENTS[20, 5]

STUDENTS[1, 1] <- “ID1”

STUDENTS[1, 2] <- “Adams, John”

STUDENTS[1, 3] <- “01142253175”

STUDENTS[1, 4] <- “Richard”

STUDENTS[1, 5] <- “Mary”

STUDENT\_ID1\_NAME <- STUDENTS[1, 2]

My solution will work with the two-dimensional array option, as it allows for more efficient code and more efficient storage of the data.

1. Using the data structures you designed in (i), devise algorithms written in ADL as suitable abstractions for the following operations. You must explain your problem-solving strategy for each problem.
2. Add a new student record

To add a record, we must simply append a row to the end of the array. This is achieved using a variable ‘lastIndex’ which keeps track of the array index of the most recently added record – this method avoids having to read the array every time we want to do almost anything with it, as we know where the existing data ends.

The information regarding the new row of data is read in and, using the aforementioned ‘lastIndex’ variable, a few lines of code assign the values to the correct column in the final row of the array.

Finally, we increment the ‘lastIndex’ variable by 1 to keep track of the new number of records.

addRecord(INOUT STUDENTS[], INOUT lastIndex, IN maxLength)

read(id, name, telephone, father, mother)

lastIndex <- lastIndex + 1

STUDENTS[lastIndex, 1] <- id

STUDENTS[lastIndex, 2] <- name

STUDENTS[lastIndex, 3] <- telephone

STUDENTS[lastIndex, 4] <- father

STUDENTS[lastIndex, 5] <- mother

end

1. Delete a student’s record

To delete a record, we first must find it. I did this by using a simple linear search function; a for loop iterates through each record searching for the one we wish to delete. Once it is located, its position is stored in a variable to use in the next pass over the array.

Now that we know where the record occurs in the array, we can iterate through again and start to remove it. We do this by starting an iteration through the array starting at the index of the record to be deleted, and with each loop we shift the next record back one space, overwriting whatever is there.

When finished, this results in all but one record being correctly adjusted – the final record in the array cannot have the next one shifted into its position because there is no next one. Instead, after iterating, we specifically select the final record and set its contents to null, appropriately adjusting the variable used to track the current number of records after doing so.

deleteRecord(INOUT STUDENTS[], INOUT lastIndex, IN maxLength, IN DEL\_ID)

declare delPos

for i <- 1 to maxLength do

if STUDENTS[i, 1] = DEL\_ID do

delPos <- i

endif

end

for i <- delPos to (maxLength – 1) do

for j <- 1 to 5 do

STUDENTS[i, j] <- STUDENTS[i + 1, j]

end

end

for i <- 1 to 5 do

STUDENTS[maxLength, i] <- null

end

lastIndex <- lastIndex - 1

end