Note: I have put the cpp file containing every piece of code written in this word document (with their questions marked as comments for easy search) in case it’s hard to read from the word document, or you want to test these codes out.



# Q1) One Dimensional Array Vs. Linked List

1. Element allocation:

Array:

An array stores its elements in the heap as a contiguous block; where all the addresses of the array’s elements are next to each other, and the difference between any two consecutive addresses is the size of the element’s datatype. The memory allocation for storing the elements occurs at compile time. Note from the code below that when you output arr by itself, it gives the address of the first element in this contiguous block, and when you increase the address by 1, it outputs an address greater than the first one by 4 (the address to the second element). This is because the type of the element here is int which holds 4 bytes. One can deduct from this that arr acts as a pointer to the first element in a series of elements in a contiguous block. Also, the compiler needs to know how much memory this contiguous block will hold exactly before runtime. We can specify the reason for this with an example: suppose you defined an array that will hold three integers, and let their addresses of the elements be any 3 random consecutive numbers : 2000, 2004, 2008. If you were to add a variable x in your program, there will be a small but possible chance that the address of this variable will be at 2012 which is next to the address of the array’s last element. This is problematic, because if we want to add another element in the array during runtime, this element will access an address that already contains the variable’s value.

Demonstration by code:

int arr[5] = { 1, 2, 3, 4, 5 };

cout << arr << endl; //Output : 001EF9A4

cout << arr + 1 << endl; //Output : 001EF9A8

cout << arr[0] << endl; //Output : 1

cout << arr[1] << endl; //Output : 2

cout << \*(arr) << endl; //Output : 1

cout << \*(arr + 1) << endl; //Output : 2

Linked List:

A linked list also stores its elements in the heap, but in random locations instead of a contiguous block. This is due to the nature of linked list’s node structure that contains the data to be stored (int, double, string, etc…), and a variable (pointer) created specifically to point to the location of the next node in the linked list so they can all be accessed by getting just the address of the first node (the head). A new element address is allocated in memory during runtime. This is because each element (node) isn’t next to each other (in terms of their addresses), so a new memory allocation can be created at any random address in the heap during runtime. This is in contrast with the array memory allocation, as the compiler needs to know how much memory this contiguous block will hold exactly before runtime. The code below shows how linked list elements are stored, and shows that the elements’ addresses are not consecutive.

Demonstration by code:

struct node

{

int data;

node \*next;

};

int main()

{

node \*LinkedListHead = new node;

LinkedListHead->data = 1;

node\* SecondElement = new node;

LinkedListHead->next = SecondElement;

SecondElement->data = 2;

cout << LinkedListHead << endl; //Output : 00604590

cout << LinkedListHead->data << endl; //Output : 1

cout << SecondElement << endl; //Output : 004A4220

cout << SecondElement->data << endl; //Output : 2

cout << LinkedListHead->next << endl; //Output : 004A4220 , same as SecondElement

return 0;

}

1. Extension of number of elements

Array:

Increasing the size:

The size of the array (number of elements) must be determined before compile time and cannot be changes afterwards. If we try to access the value inside the address directly after the last address in the contiguous block, the value returned will be garbage. Moreover, trying to add another element at that location will give a runtime error. When typing the code below, the compiler will show a warning stating “Index ‘5’ is out of valid index range ‘0’ to ‘4’ for possible stack allocated buffer ‘arr’.”

Demonstration by code:

int arr[5] = { 1, 2, 3, 4, 5 }; //The array has 0-based index, so the elemnts' indexes are 0,1,2,3,4

arr[5] = 6;

cout << arr[5]; //Compile time error

However, one can argue that there is a method to increase the array’s size, by making an array with bigger size, then copying all the values of the smaller array to the bigger array, but by doing this, we are still not resizing our array, we are just creating a new bigger one. Nonetheless, the code below shows the method to do that by two ways: the first one is setting the size of the new array manually, and the second one is to dynamically allocate the size of the new array.

Code by demonstration:

const int aSize = 5;

int arr[aSize] = { 1, 2, 3, 4, 5 };

cout << "Enter new element: ";

int x; cin >> x; //Input 6

int arrBig[aSize + 1] = { 0 };

int i;

for (i = 0; i < aSize; i++) //This will loop till i == 5

arrBig[i] = arr[i]; //arrBig change: {1,0,0,0,0,0} -> {1,2,0,0,0,0} -> {1,2,3,0,0,0} -> {1,2,3,4,0,0} -> {1,2,3,4,5,0}

arrBig[i] = x; //{1,2,3,4,5,6}

///\*\*\*Another method using dynamic memory allocation\*\*\*\\\

const int aSize = 2;

int arr[aSize] = { 1, 2 };

cout << "Enter the size of your new array: ";

int bigSize; cin >> bigSize; //Input 4

int\* arrBig = new int[bigSize]; //"new" allocates memory to the heap of size sizeof(int) \* n bytes and return the address of the memory where it's stored to the variable (pointer) arrBig, this why it can accept variables which are not const

int i;

for (i = 0; i < aSize; i++) //This will loop till i == 2 (it will assign till arrBig[1], but will exit the loop with i == 2)

arrBig[i] = arr[i]; //arrBig change: {1,0,0,0} -> {1,2,0,0}

for (i = i; i < bigSize; i++) //This will loop till i == 4

{

cout << "Enter the value to be inserted at position: " i + 1 << " : ";

int x; cin >> x; //Input: 3 4 5 6

arrBig[i] = x; //{1,2,3,0} -> {1,2,3,4}

}

delete []arrBig; //Since the memory is dynamically allocated to the heap using new, we have to deallocate it manually by writing this line (when we don't need anymore, of course):

Decreasing the size:

The same concepts of increasing the array’s size applies here. Since the size is determined only at compile time, we can’t decrease its size during runtime. However, we can still copy the elements of the array to a newer and smaller one, after deleting one element or more of course. The code below shows how this can be done.

Demonstration by code:

const int aSize = 3;

int arr[aSize] = { 1, 2, 3};

cout << "Enter the value of the element that you want to delete: ";

int x; cin >> x; //Input 2

for (int i = 0 ; i < aSize ; i++)

if (arr[i] == x)

{

arr[i] = 0; //arr change: {1,0,3}

break;

}

int arrSmall[aSize - 1] = { 0 };

for (int i = 0, j = 0; i < aSize; i++) //This will loop till i == 3, then j == 2

if (arr[i] != 0)

{

arrSmall[i] = arr[j]; //arrSmall change: {1,3}

j++;

}

///\*\*\*Another method using dynamic memory allocation\*\*\*\\\

const int aSize = 5;

int arr[aSize] = { 1, 2, 3, 4, 5 };

cout << "Enter the size of your new array: ";

int smallSize; cin >> smallSize; //Input 3

int numOfDeletedElements = aSize - smallSize;

int\* valsToBeDeleted = new int[numOfDeletedElements]; //Creates a dynamically allocated array with size of the difference between the original and the smaller arrays

int\* arrSmall = new int[smallSize]; //New dynamically allocated smaller array

cout << "Enter the values that you want to delete with spaces between each value: ";

for (int j = 0; j < numOfDeletedElements; j++) //Puts all the values to be deleted in an array

cin >> valsToBeDeleted[j]; //Input: 2 4

for (int i = 0; i < aSize; i++)

for (int j = 0; j < numOfDeletedElements; j++)

if (arr[i] == valsToBeDeleted[j])

{

arr[i] = 0;

valsToBeDeleted[j] = 0; //arrSmall change: {1,0,3,4,5} -> {1,0,3,0,5}

break;

}

for (int i = 0, j = 0; i < aSize; i++) //This will loop till i == 5, then j == 2

if (arr[i] != 0)

{

arrSmall[i] = arr[j]; //arrSmall change: {1,3,5}

j++;

}

delete[]arrSmall;

Linked List:

The size of the linked list can increase and decrease during runtime as they are dynamically allocated in the heap.The code below shows how to dynamically allocate new element addresses and their deletion during runtime.

Demonstration by code:

struct node

{

int data;

node\* next = nullptr;

};

int main()

{

int uData;

cout << "enter data to be inserted at the first node: ";

cin >> uData; //Input : 1

node \*LinkedListHead = new node; //Address 00604590 is chosen at runtime to insert uData value to it

LinkedListHead->data = uData; //Assigns uData to first node

cout << "enter data to be inserted at the second node: ";

cin >> uData; //Input : 2

node\* SecondElement = new node; //the linked list size increases as Address 004A4220 is chosen to insert uData value to it

LinkedListHead->next = SecondElement; //Assigns 004A4220 as the address of the node that follows the node with address 00604590

SecondElement->data = uData; //Assigns uData to second node

node\* ptr = LinkedListHead; //Initializes pointer ptr to the head of the linked list (its first node)

while (ptr != nullptr) //Pointer ptr loops through and outputs the whole linked list

{

cout << ptr->data << " "; //Output : 1 2

ptr = ptr->next; //Makes ptr point to the next node in the linked list, note that after the loop finishes, the linked list contains 2 nodes, so its size increased during runtime

}

delete SecondElement; //Deletes second node

LinkedListHead->next = nullptr; //Makes the first node point to null again

ptr = LinkedListHead;

while (ptr != nullptr) //Pointer ptr loops through the linked list again

{

cout << ptr->data << " "; //Output : 1

ptr = ptr->next; //Note that now the linked lists contains only 1 node, so its size decreased during runtime

}

return 0;

}

1. Accessibility by adding, modifying, and deleting elements

Array:

1. Access: Direct access occurs, as you simply specify the index of the element you want to access. Faster than the linked list.

Demonstration by code:

int arr[5] = { 1,2,3,4,5 };

cout << "Enter the postition you position of the value you want to access: ";

int pos; cin >> pos; //Input: 3

cout << "The value at " << pos << " is: " << arr[pos - 1]; //pos -1 as index is 0-based, Output: 3

1. Addition: If we mean substituting garbage values (or the initial values, supposedly 0) in the array with new values, or removing garbage (or initial) values to be able to enter the new value in a certain position by shifting the other non-garbage values, then it’s possible; however, it’s much slower than in linked lists, and have three case scenarios that all require for at least one garbage (or initial) value to be present within the boundries of the array:

Case 1: Inserting new element at the start of the array

To do this, we shift all the elements, starting from the last element, to the right. After that, we insert the new element at the first position in the array (at index 0).

Case 2: Inserting new element after the last non-garbage element (or 0) (end of array)

Same as case 1, but we shift the elements, starting from the first element, to the left this time. After that, we insert the new element at the last position in the array (at index s – 1, where s is the number of elements in the array).

Case 3: Inserting new element at the nth position of the array

Same with case 1, but we shift the elements, starting from the last element, to the right, and stop this shifting at the element with index n – 1. After that, we insert the new element at this position in the array.

Note:

All these cases work with the exception that there is at least one empty element in the array, but if the array was full, then adding new element will not be possible, because as previously stated in b), the size of the array cannot changed once it has been set. However, we can technically still accomplish this by creating a bigger array and copy the original array’s values to the new array, the code for this is covered in b).

Demonstration by code:

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Case 1\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

const int s = 5; //Array size = s = 5

int inVal = 0; //Initial value for array elements = inVal = 0

int arr[s] = { inVal };

arr[0] = 1, arr[1] = 2, arr[2] = 3; //Array now equals { 1, 2, 3, 0, 0 }

int lastE;

for (lastE = s - 1; arr[lastE] == inVal; lastE--); //Loop that will stop at index 2

for (int i = lastE; i >= 0; i--)

arr[i + 1] = arr[i]; //{ 1, 2, 3, 3, 0 } -> { 1, 2, 2, 3, 0 } -> { 1, 1, 2, 3, 0 }

cin >> arr[0]; //Input: 4 -> { 4, 1, 2, 3, 0 }

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Case 2\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

const int s = 5; //Array size = s = 5

int inVal = 0; //Initial value for array elements = inVal = 0

int arr[s] = { inVal };

arr[0] = 1, arr[1] = 2, arr[2] = 3; //Array now equals { 1, 2, 3, 0, 0 }

int lastE;

for (lastE = s - 1; arr[lastE] == inVal; lastE--); //Loop that will stop at index 2

cin >> arr[lastE + 1]; //Input: 4 -> { 1, 2, 3, 4, 0 }

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Case 3\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

const int s = 5 //Array size = s = 5

int inVal = 0; //Initial value for array elements = inVal = 0

int arr[s] = { inVal };

arr[0] = 1, arr[1] = 2, arr[2] = 3;//Array now equals {1, 2, 3, 0, 0}

int lastE, n;

cout << "Enter the position where you want to enter your value from 1 to " << s << ": ";

cin >> n; //Input: 2

n--; //makes n 0 based (1)

for (lastE = s - 1; arr[lastE] == inVal; lastE--); //Loop that will stop at index 2

for (int i = lastE; i >= n; i--)

arr[i + 1] = arr[i]; //{ 1, 2, 3, 3, 0 } -> { 1, 2, 2, 3, 0 }

cout << "Enter your value: ";

cin >> arr[n]; //Input: 4 -> { 1, 4, 2, 3, 0 }

for (int i = 0 ; i < s ; i++)

cout << arr[i] << " ";

1. Modification: array modification also occurs directly and is faster than linked list, as you specify the index where you want a new value to be placed, and then that value replaces whatever was in that position. Note that here, it is okay for the array to be full, as the new value will replace one of them anyway.

Demonstration by code:

const int s = 5; //Array size = s = 5

int inVal = 0; //Initial value for array elements = inVal = 0

int arr[s] = { inVal };

arr[0] = 1, arr[1] = 2, arr[2] = 3; //Array now equals { 1, 2, 3, 0, 0 }

cin >> arr[0] >> arr[3]; //Input: 4 5 -> { 4, 2, 3, 5, 0}

1. Deletion: the availability of element deletion in arrays depends on how you define “element deletion” :

Definition 1: return the element of any chosen position to its initial value

In this case, it’s faster than linked lists, as it works just like modification of array elements.

Demonstration by code:

const int s = 5; //Array size = s = 5

int inVal = 0; //Initial value for array elements = inVal = 0

int pos;

int arr[s] = { inVal };

arr[0] = 1, arr[1] = 2, arr[2] = 3; //Array now equals { 1, 2, 3, 0, 0 }

cin >> pos; //Input: 1 -> { 0, 2, 3, 0, 0}

Definition 2: Deleting an element, then making the rest of the elements next to each other without any garbage (or 0) value between them, in order for their addresses to be still close to each other.

The three cases by this definition are deletion of first, last, and nth element. The code below shows how each of these cases are handled.

Demonstration by code:

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Delete first\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

const int s = 5;//Array size = s = 5

int inVal = 0; //Initial value for array elements = inVal = 0

int arr[s] = { inVal };

arr[0] = 1, arr[1] = 2, arr[2] = 3; //Array now equals { 1, 2, 3, 0, 0 }

for (int i = 1; i < s; i++)

arr[i - 1] = arr[i]; //{ 2, 2, 3, 0, 0 } -> { 2, 3, 3, 0, 0} -> { 2, 3, 0, 0, 0} -> remains the same till i == s-1

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Delete last\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

const int s = 5; //Array size = s = 5

int inVal = 0; //Initial value for array elements = inVal = 0

int arr[s] = { inVal };

arr[0] = 1, arr[1] = 2, arr[2] = 3; //Array now equals { 1, 2, 3, 0, 0 }

int lastE, n;

for (lastE = s - 1; arr[lastE] == inVal; lastE--); //Loop that will stop at index 2

arr[lastE] = 0; //{ 1, 2, 0, 0, 0 }

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Delete nth\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

const int s = 5; //Array size = s = 5

int inVal = 0, n; //Initial value for array elements = inVal = 0

int arr[s] = { inVal };

arr[0] = 1, arr[1] = 2, arr[2] = 3; //Array now equals { 1, 2, 3, 0, 0 }

cout << "Enter the position of the value you want to delete: ";

cin >> n; //Input: 2

n--; //Makes n 0-based (1)

for (int i = n; i < s-1; i++)

arr[i] = arr[i + 1]; //{ 1, 3, 3, 0, 0 } -> { 1, 3, 0, 0, 0} -> remains the same till i == s-1

for (int i = 0; i < s; i++)

cout << arr[i] << " ";

Definition 3: same as definition 2, but decreasing the size of the array to fit only the elements with non-garbage (or 0) value.

This is impossible for the array, because as previously stated, the size of the array cannot be changed at runtime and can be set once during compile time. However, we can technically still accomplish this by creating a smaller array and copy the original array’s values (with the exception of the elements that we want to delete) to the new array, the code for this is covered in b).

Linked List:

1. Access: Accessing the value in a certain position (node) in linked list takes longer time than doing so in an array, this is because the only way to access the linked list is through the address of the first node (head pointer).
2. Addition: adding a new element (node) in linked list is quicker than in an array, as you don’t have to the rest of the elements in the array, you just need to adjust the next pointers of the nodes in proximity with the place where you are going to insert your new node.
3. Modification: Takes the same time as accessing an element (node) value, as you still iterate from the first node till the requested mode, and when you access it, you can change its value, so modification here is still slower than in an array.
4. Deletion: Takes the same time as Adding an element, as the only thing you need to change still are the next pointers of the nodes in proximity with the place where you want to delete a node.

All these manipulations will be shown in the code below.

Demonstration by code:

class node // created a class in order to efficiently hold all the manipulations of the linked list inside one class

{

protected: //This is done as the singleNode struct won't be accessed outside of node class, so there is no need to put it in public. Also, we put it as protected in order to be available for usage by public functions.

struct singleNode

{

int data;

singleNode\* next;

singleNode(int d, singleNode\* n = nullptr) //second parameter here sets next to nullptr if no second argument is inserted in main function

{

data = d;

next = n;

}

};

singleNode\* head;

public:

node() { head = nullptr; }

~node() //Destructor that iterates through all nodes and deletes them, as we don't want remaining space in the heap

{

singleNode\* ptr = head;

while (ptr != nullptr)

{

singleNode\* garbage = ptr;

ptr = ptr->next;

delete garbage;

}

}

void displayList() //Display the values inside the linked list

{

singleNode\* ptr = head;

while (ptr != nullptr)

{

cout << ptr->data << " ";

ptr = ptr->next;

}

}

int accessPos(int num) //returns position of a node that contains data num which could then be displayed with cout

{

singleNode\* ptr = head;

int pos = 1;

while (ptr->data != num)

{

ptr = ptr->next;

pos++;

}

return pos;

}

int accessVal(int pos) //returns data of a node at position pos which could then be displayed with cout

{

singleNode\* ptr = head;

for (int i = 0 ; i < pos ; i++)

ptr = ptr->next;

return ptr->data;

}

void modify(int newNum, int pos) //takes newNum value and makes it replace the value in the node with position pos

{

singleNode\* ptr = head;

for (int i = 0; i < pos; i++)

ptr = ptr->next;

ptr->data = newNum;

}

void addBegNode(int num) //Adds new node at the beginning of the linked list

{

if (head = nullptr)

head = new singleNode(num);

else

head = new singleNode(num, head); //Note that since the compiler reads from right to left, this line is saying : create a new node with data passed in the parameter and next pointing to the previous head, then assign the head pointer to this node

}

void addEndNode(int num) //Adds new node at the end of the linked list

{

if (head = nullptr)

head = new singleNode(num);

else

{

singleNode\* ptr = head;

while (ptr->next != nullptr)

ptr = ptr->next;

ptr->next = new singleNode(num);

}

}

void addNthNode(int num, int pos) //Adds new node at the nth position of the linked list

{

singleNode\* ptr = head, \* prev;

if (pos == 1)

head = new singleNode(num, head);

else

{

for (int i = 1; i < pos; i++)

{

prev = ptr;

ptr = ptr->next;

}

ptr = new singleNode(num, ptr);

prev->next = ptr;

}

}

void removeVal(int num) //Deletes node that contains data inserted by the user

{

singleNode\* ptr, \* prev;

if (head == nullptr)

return;

if (head->data == num)

{

ptr = head;

head = head->next;

delete ptr;

}

else

{

ptr = head;

while (ptr != nullptr && ptr->data != num)

{

prev = ptr;

ptr = ptr->next;

}

if (ptr != nullptr)

{

prev->next = ptr->next;

delete ptr;

}

}

}

void removePos(int pos) //Deletes node that contains position inserted by the user

{

pos--;

singleNode\* ptr, \* prev;

if (head == nullptr)

return;

if (pos == 0)

{

ptr = head;

head = head->next;

delete ptr;

}

else

{

for (int i = 0 ; i < pos ; i++)

{

prev = ptr;

ptr = ptr->next;

}

if (ptr != nullptr)

{

prev->next = ptr->next;

delete ptr;

}

}

}

};

# Q2) sortArray() && sortLinked()

1. sortArray() function:

class Data //1

{

private:

int id; //2

string name; //3

public:

int getId() { return id; } //4

string getName() { return name; } //5

};

void sortArray(Data dataArray[], int arraySize) //6

{

for (int i = 0; i < arraySize - 1; i++) //7

for (int j = i + 1; j < arraySize; j++) //8

if (dataArray[j].getId() < dataArray[i].getId()) //9

{

Data temp = dataArray[i]; //10

dataArray[i] = dataArray[j]; //11

dataArray[j] = temp; //12

}

}

1. sortArray() function clarification:

Note: I added the lines //1,2,3 here, even though they are not required, in order to show that I made the member variables private, and called them by member functions in the public section. I put them here in order not to show any confusion on where getId() and getName() came from. Although the problem said that the class Data contains id and name, and didn’t specify any member functions, I added them as this was how we dealt with classes throughout the semester. In addition, if these weren’t needed, the only change that will happen is that the class will comprise only of 2 public member variables which will be called directly in the sort functions. Now, regarding the question; since the question didn’t elaborate on the variable of the Data object that we should consider to sort our object list by, I decided to use int id as the deciding variable as it makes more sense and it’s always unique, while names can be the same. However, if we were to program the function by choosing string name as the deciding variable, the aforementioned code will be exactly the same, except that instead of //9, we will type something like this if(dataArray[j].name.at(0)<dataArray[i].name.at(0))

Which only sorts by the first letter of the name, but it is sufficient nonetheless. And now, a line by line explanation for the above code:

//6 Sets the function sortArray with an array dataArray[] containing Data elements as first parameter and integer arraySize containing the size of dataArray[].

//7 Creates a for loop with i as dataArray[]’s starting element and stops when i equals the element before the last element. The reason for this is explained in //8

//8 Creates a for loop with j as the index of the element right after I, and stops when j equals the last element. Since the bubble sort’s main principle is comparing an element with the element after it, setting i to reach the last element will create a problem, as in the last for loop in //7, the value of j will be an index which resides after the last element of the array, so that final loop will compare the value of the last element with a garbage value and that cannot happen.

//9 Checks if the id of dataArray[j] is less than that of dataArray[i].

//10,11,12 if //9 is true, then it will swap the Data object inside dataArray[i] with that of dataArray[j] by introducing a temporary Data variable called temp to store Data of dataArray[i] before it’s overwritten by dataArray[j]. And for the clarification of how the two for loops with swap acts as bubble sort, the best explanation can be done through tracing: let the arraySize = 4, and the ids are {4,3,2,1} respectively. Then

i=0,j=1:{4,3,2,1} -> {3,4,2,1} -> {2,4,3,1} --i=1,j=2--> {1,4,3,2} -> {1,3,4,2} -> {1,2,4,3} --i=2,j=3--> {1,2,3,4}

1. sortLinked() function:

class dataLL //1

{

public:

Data dataInLL; //2

dataLL\* next; //3

};

void sortLinked(dataLL \*dataLinked, int linkedListSize) //4

{

dataLL\* iptr, \* jptr; //5

for (iptr = dataLinked; iptr != nullptr; iptr = iptr->next) //6

for (jptr = iptr->next; jptr != nullptr; jptr = jptr->next)//7

if (jptr->dataInLL.getId() < iptr->dataInLL.getId()) //8

{

Data temp = iptr->dataInLL; //9

iptr->dataInLL = jptr->dataInLL; //10

jptr->dataInLL = temp; //11

}

}

1. sortLinked() function clarification:

Note: the lines //1,2,3 are not required yet I included them to clarify where the name dataInLL comes from.

//4 Sets the function sortLinked with a linked list dataLL \*dataLinked (which is a pointer to the first node of dataLinked, which is of type class dataLL that contains Data element and next pointer) as first parameter and integer arraySize containing the size of dataLinked (number of nodes that it is connected to).

//5 Creates a for loop with i as dataLinked’s starting element and stops when i equals the element before the last element. The reason for this is explained in //6

//6 Creates a for loop with j as the index of the element right after I, and stops when j equals the last element. Since the bubble sort’s main principle is comparing an element with the element after it, setting i to reach the last element will create a problem, as in the last for loop in //5, the value of j will be an index which resides after the last element of the array, so that final loop will compare the value of the last element with a garbage value and that cannot happen.

//7 Checks if the id of dataLinked node which is in the jth position is less than that of the ith position.

//8,9,10 if //7 is true, then it will swap the Data object inside dataLinked of ith position with that of dataLinked of jth position by introducing a temporary Data variable called temp to store Data of dataLinked of ith position before it’s overwritten by dataLinked of jth position.

# Q3) Students program + getStudent() function

Brief summary of the code:

I will divide the program into three sections and summarize each one:

1. The initialization section:

Initialized sNum = 10 which is the number of students based on the problem.

struct student which contains the student’s number, name, and grade.

struct studentRecordLocation which is similar to student struct but instead of storing his number, it stores the location of the number in the text file in numLocation. Although I could have defined this variable in the student struct instead of creating another struct, I didn’t do that because the problem specified that student contains only the number, name, and grade of the student.

student sArray[sNum + 1] which is an array of type student, and contains 11 elements.

studentRecordLocation chosenStudent[sNum + 2] which is an array of type studentRecordLocation, and contains 12 elements.

And finally, fstream sRecords("FinalAssignment.txt", ios::in | ios::out), which is an fstream object that reads and writes from “FinalAssignment.txt”. All of these are initialized globally for easy access throughout the program.

1. The functions section:

void saveToFileAndArray(fstream& sRecords, student sArray[]):

It takes the fstream object, and the sArray[] as parameters and returns nothing. Note that these parameters aren’t necessary as they are defined globally, but I included them for extra functionality, in case these functions were used in another program where the parameter aren’t defined globally. This function will do two things:

if the file is empty, it enters new data and saves it in the text file, sArray[], and chosenStudent[] simultaneously. If the file is not empty, it will ask the user whether he wants to update the data or not, if yes, then call the function again (recursion), else not, then call saveDataFromFile function which is explained next. Note that the function stores only the location of students' numbers, as to not fill the chosenStudents[] with useless information (as we will only access 1 element of it when the user enter the number of the student whose data he wants to access).

void saveDataFromFile(fstream& sRecords, student sArray[], studentRecordLocation chosenStudent[]):

The explanation for the parameters (and its return type) is similar to the first function, with the addition of parameter studentRecordLocation chosenStudent[]. This function is used in case the file already has data written in it and the user doesn't want to update it, so it stores these data in sArray[], and chosenStudent[].

studentRecordLocation getStudent(int num):

This returns all the information of the student (as the problem didn’t specify what member variable of student should we exactly return) whose “number” is the input parameter “num”. The use of the seekg() function has been implemented in it as requested in the problem (the comments next to its implementation are highlighted in yellow).

1. The main function section

Since most of the code was implemented in functions outside of main(), the main() function code is small and clean. It comprises of the following:

A call to the function saveToFileAndArray(sRecords, sArray).

A prompt for the user to enter the number of a student to get his/her name and grade.

Initializing num as an int which stores number inputted by user.

An if condition that checks if the member variable “numLocation”, which is the location of the number inputted by the user in the text file, is equal to 0 or not. If it isn’t, then display the name and grade of the student that has the same number as the one inputted by the user. If it’s equal to 0, then end the program. Note that this check works as the studentRecordLocation struct has the numLocation initialized to 0, so this means that throughout the getStudent() function, no number matched the one entered by the user, and that the user didn’t attempt to enter a correct number.

I have attached a text file as a test sample, feel free to use instead of manually typing one.



Below is the whole program with comments explaining how the program works:

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\\

//Author: Ashraf Adel ID: 196280 Group: A-6 \\

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\\

#include <iostream>

#include <string>

#include <fstream>

#include <sstream>

#include <iomanip>

using namespace std;

//Q3

const int sNum = 10; //Number of students

struct student

{

int number = 0;

string name = "";

float grade = 0.0;

};

struct studentRecordLocation //Created this as the problem didn't specify whether we should return the name or the grade of the chosen number, so instead, getStudent() returns an object containing his information

{

int numLocation = 0; //This is the location of the number in the text file

string name = "";

float grade = 0.0;

};

student sArray[sNum + 1]; //sNum + 1 as counting starts from 1 in the loops

studentRecordLocation chosenStudent[sNum + 2]; //sNum + 2 just for extra space and to assure that the array won't go out of bounds

fstream sRecords("FinalAssignment.txt", ios::in | ios::out);

void saveDataFromFile(fstream& sRecords, student sArray[], studentRecordLocation chosenStudent[]) //This function is used in case the file already has data written in it and the user doesn't want to update it, so it stores these data in sArray[], and chosenStudent[]

{

sRecords.open("FinalAssignment.txt", ios::in | ios::beg);

if (!(sRecords.is\_open())) //check if file is not valid

{

cout << "sorry but the file is not valid..." << endl;

sRecords.close();

return;

}

else

{

string notNeeded = ""; //this is created to store unrequired data in the text file, just to move the cursor (offset) to the relevant data

for (int i = 1 ; i <= sNum && !(sRecords.eof()) ; i++)

{

sRecords >> notNeeded >> sArray[i].number >> notNeeded >> sArray[i].name >> notNeeded;

if (notNeeded != "Grade:")

{

sArray[i].name = sArray[i].name + " " + notNeeded;

sRecords >> notNeeded >> sArray[i].grade;

}

else

sRecords >> sArray[i].grade;

chosenStudent[i].name = sArray[i].name, chosenStudent[i].grade = sArray[i].grade;

sRecords.ignore(100, '\n'); //ignores rest of characters in the line until reaching a new line (thus the offset of the file stops at the new line

int endLineLoc = sRecords.tellg(); //Stores the length of the line to know where to start seekg() in getStudent()

if (i != sNum)

{

chosenStudent[i + 1].numLocation = endLineLoc;

chosenStudent[i + 1].numLocation += 7; //The +7 is done as every line starts with Number:

}

if (i == 1)

chosenStudent[i].numLocation = 7; //In the first number only, store location of number in text file as 7 (which is after Number:)

}

}

}

void saveToFileAndArray(fstream& sRecords, student sArray[]) //if the file is empty, then enter new data and save it in text file, sArray[], and chosenStudent[] simultaneously. If not, call saveDataFromFile()

{

sRecords.open("FinalAssignment.txt", ios::out); //In here we made ios::out, because if we don't, and the user decides to update the data, the function will recurse and then sRecords.tellg() will still not be equal to zero, so the file won't be updated if we remove ios::out

if (!(sRecords.is\_open())) //check if file is not valid

{

cout << "sorry but the file is not valid..." << endl;

sRecords.close();

return;

}

else

{

sRecords.close();

sRecords.open("FinalAssignment.txt", ios::in | ios::ate);

if (sRecords.tellg() == 0) //check if file is empty, then create the first record of students

{

sRecords.close();

sRecords.open("FinalAssignment.txt", ios::out | ios::app); //Write to file

int snum; string sname; float sgrade;

cout << "Enter the data for " << sNum << (sNum == 1 ? " student." : " students.") << endl;

for (int i = 1; i <= sNum; i++)

{

cout << "Enter the number of student " << i << ": ";

cin >> snum; cin.ignore();

cout << "Enter the name of student " << i << ": ";

getline(cin, sname);

cout << "Enter the grade of student " << i << ": ";

bool loop = 0;

do //loops if grade is invalid

{

if (loop)

{

cout << "Enter a grade from 0 to 100: ";

cin >> sgrade;

}

else

cin >> sgrade;

loop = 1;

}while(sgrade < 0 || sgrade > 100)

cin.ignore();

sArray[i].number = snum, sArray[i].name = sname, sArray[i].grade = sgrade; //Store in sArray[]

sRecords << "Number: " << setw(15) << left << snum << "\t" //Store in text file

<< "Name: " << setw(15) << left << sname << "\t"

<< "Grade: " << sgrade << endl;

sRecords.seekg(0, ios::end); //Using seekg() function to reach the end of line, so the next number that we will get will be at that location +7

int endLineLoc = sRecords.tellg();

if (i != sNum) //In these lines, we will store only the location of students' numbers as to not fill the chosenStudents with useless information (as we will only access 1 element of it)

chosenStudent[i + 1].numLocation = endLineLoc + 7; //The +7 is done as every line starts with Number:

if (i == 1)

chosenStudent[i].numLocation = 7; //In the first number only, store location of number in text file as 7 (which is after Number:)

}

cout << "Records were saved successfully..." << endl;

}

else

{

sRecords.close();

bool choice = 0;

cout << "Sorry but the file already contains records of students, would you like to update the file? (0 for no, 1 for yes): ";

cin >> choice;

if (choice)

saveToFileAndArray(sRecords, sArray); //Recurse the function again to update file with new data

else

{

saveDataFromFile(sRecords, sArray, chosenStudent); //Calls a function to store the data of text file to sArray[] and chosenStudent[]

cout << "Returning..." << endl;

}

}

}

sRecords.close(); //This is important to do after modifying text

}

studentRecordLocation getStudent(int num) //The required function that gets the information of the student with the number in the parameter in the form of an object

{

sRecords.open("FinalAssignment.txt");

if (!(sRecords.is\_open())) //check if file is not valid

{

cout << "sorry but the file is not valid..." << endl;

sRecords.close();

return chosenStudent[0]; //Empty, as entry in this array starts from chosenStudent[1]

}

else

{

sRecords.close();

sRecords.open("FinalAssignment.txt", ios::in | ios::ate);

if (sRecords.tellg() == 0) //check if file is empty, then create the first record of students

saveToFileAndArray(sRecords, sArray); //Calls saveToFIleAndArray() which gets the locations of the numbers in the text file as required by seekg() as required in the problem

bool again = 0;

do

{

sRecords.open("FinalAssignment.txt");

if (!(sRecords.is\_open())) //check if file is not valid

{

sRecords.close();

return chosenStudent[0];

}

else

{

sRecords.close();

sRecords.open("FinalAssignment.txt", ios::in | ios::ate);

if (sRecords.tellg() == 0) //check if file is empty, then call SaveToFileAndArray()

saveToFileAndArray(sRecords, sArray);

sRecords.close();

sRecords.open("FinalAssignment.txt", ios::in | ios::beg);

int numTest;

string lNameTest = "";

for (int i = 1; i <= sNum && !(sRecords.eof()); i++)

{

int loc = chosenStudent[i].numLocation; //Note that seekg() was used (as required) and it has been included in both the next line, and in the saveToFileAndArray() function

sRecords.seekg(loc, ios::beg); //sets the offset of seekg to the location of student i's number in the text file

sRecords >> numTest;

if (numTest == num)

{

sRecords >> chosenStudent[i].name >> chosenStudent[i].name >> lNameTest;

//The reason why this is done 3 times can be explained by visualizing the text file:

//Number: num Name: Ashraf Adel Grade: 55

// sRecords >> 1 ^^^^ 2^^^^^^ ^^^^3

//the next if statement checks whether the name has last name or not

if (lNameTest != "Grade:")

{

chosenStudent[i].name = chosenStudent[i].name + " " + lNameTest;

sRecords >> lNameTest;

sRecords >> chosenStudent[i].grade;

}

else

sRecords >> chosenStudent[i].grade;

return chosenStudent[i];

}

}

sRecords.close();

cout << "Invalid number. Would you like to try again? (0 for no, 1 for yes): ";

cin >> again;

if (again)

{

cout << "Enter the number of the student to get his/her name: ";

cin >> num;

}

}

} while (again);

if (!again)

{

sRecords.close();

return chosenStudent[0];

}

}

}

int main()

{

saveToFileAndArray(sRecords, sArray);

cout << "Enter the number of a student to get his/her name and grade: ";

int num; cin >> num;

if (getStudent(num).numLocation != 0)

cout << "Name: " << getStudent(num).name << "\nGrade: " << getStudent(num).grade << "\n";

else

cout << "Closing program...\n";

sRecords.close();

return 0;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\\

Final note: I never had this much fun while doing an assessment before. This is because in this assessment and the last one, I have learned so much more about C++ and I feel like the material of this course will greatly benefit me in future programming modules! I would like to thank all the T.As for their continuous support and providing us with clear explanations of each lab and new concept, and thank Dr. Mostafa for providing us with two assessments that truly expanded our knowledge!...... and for putting up with all my emails. Can’t wait to see you again in the Algorithms module! <3

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\\