Revised:  This file is updated with the following changes. So, you can use this file. The table below is simply a revision history, hopefully to avoid any confusion.

|  |  |  |
| --- | --- | --- |
| Part I.a | “For” loop | “I  <  3” should be “i  >  3” |
| Part II.a | “While” loop | Remove “}” between “sum += \_\_\_\_\_” and “node\_prt = \_\_\_\_\_\_” |
| Part II.b.ii | “node\_prt” should be “node\_ptr” |  |

**Name**:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Grading**:

|  |  |  |
| --- | --- | --- |
| **Assigned** | **Value** | **Completed** |
| Part I | 4 |  |
| Part II | 9 |  |
| Part III | 7 |  |
| Total | 20 points |  |

**Objectives**:

(1) Compare and contrast three data structures (arrays, vectors, linked list)

 (2) Use Big-O for evaluating chosen implementations.

**Turn In**: This booklet with answers to questions – hard copy.

**Part I**: [4] Answer the following questions.

**a.** [2] Trace the execution of the following code:

           int an\_array[] = {0, 1, 2, 3, 4, 5, 6, 7};   // initialize array of integers containing the specified values

                                                                              // index values of 0 to 7

          for (int i = 3; i < 7; i++)                           // loop through the array from index 0 to 7

          an\_array[i + 1] = an\_array[i];                 // new values for an\_array={0,0,0,0,0,0,0,0}

           int an\_array[] = { 0, 1, 2, 3, 4, 5, 6, 7 }; // initialize array of integers containing the specified values

           for (int i = 7; i > 3; i--){                 // set i to 7 then loop through by decramenting, then don't loop index 3

                an\_array[i] = an\_array[i - 1]; // new array values {0,1,2,3,3,4,5,6}

           }

* What are the contents of “an\_array” after the execution of each loop?
  + new values for an\_array={0,0,0,0,0,0,0,0}
  + new values for an\_array={0,1,2,3,3,4,5,6}

**b.** [2] Write the following function:

/\*\*       Finds the first occurrence of target in “a\_vector”.

            @return           The index of the first occurrence of the target

                                    or size if the target is not found

\*/

int find(const vector<string>& a\_vector, const string& target) {

     for(int i = 0, i < a\_vector.size(); i++) {

           if(a\_vector[i]!=target)

               return;

          else

               return i;

     }

}

**Part II**: [9] Answer the following questions.

**a.** [2] (Two parts)

 (1) Draw a single-linked list of “int” values containing the integers 5, 10, 7, and 30 and referenced by “head.”

 (2) Complete the following fragment, which adds all “int” values in a list. Your fragment should walk down the list, adding any integers to “sum” and ignoring any objects in the list that are not “int” objects.

**[The node structure of “node\*” is defined in the textbook.]**

     int  sum = 0;

     Node\*  node\_ptr = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;

     while (node\_ptr != NULL) {

          sum += \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;

          node\_ptr = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;

     }

**b.** [3] For the single-linked list in Figure 4.16 (textbook), assume that you have a pointer variable “head (type Node\*)” that references the first node.

Explain the effect of each statement in the following fragments.

i.

head = new Node(“Shakira”, head->next)     // creates a new node with data component “Shakira” and its next component is the current heads —> next element

ii.

Node\* node\_ptr = head->next;

node\_ptr->next = node\_ptr->next->next;

iii.

Node\* node\_ptr = head;

while (node\_ptr->next != NULL)

            node\_ptr = node\_ptr->next;

node\_ptr->next = new node(“Tamika”);

**c.** [2] For the double-linked list in Figure 4.20 (textbook), explain the **effect of each statement in the following fragments.**

i.

DNode\* node\_ptr = tail->prev;

node\_ptr->prev->next = tail;

tail->prev = node\_ptr->prev;

ii.

DNode\* node\_ptr = head;

head = new DNode(“Tamika”);

head->next = node\_ptr;

node\_ptr->prev = head;

**d.** [2] Using the single-linked list shown in Figure 4.16, and assuming that “head” references the first “Node” and “tail” references the last “Node”, write statements to do the following:

**Insert “Bill” before “Tom”.**

**Part III**: [7] Suppose you have two implementations of a doubly linked list, “SimpleList” and “SentinelList”. For purposes of discussion, let “firstNode” denote the node holding the first element, and let “lastNode” denote the node holding the last element.

* An instance of SimpleList has two fields: first = firstNode, and last = lastNode.
* An instance of SentinelList has two fields: header = new Node(), and trailer = new Node(). These are distinct, special nodes created by the SentinelList constructor, and they stand permanently at either end of the list. We have header.next = firstNode, and trailer.prev = lastNode.

**a.** [3] If the list is empty, then firstNode and lastNode don’t exist.

Then what should go in SimpleList’s first and last fields?

How about SentinelList’s header and trailer fields?

How about header.next and trailer.prev?

|  |  |
| --- | --- |
| SimpleList.first |  |
| SimpleList.last |  |
| SentinelList.header |  |
| SentinelList.trailer |  |
| SentinelList.header->next |  |
| SentinelList.trailer->prev |  |

**b.** [2] You would like to maintain a “sorted” sequence of integers. So you need an “insert(e)” method that will insert the new element “e” at the correct position in the current sorted sequence.

To do this involves two parts:

(1) locating the correct position in sequence and

(2) inserting the element into the sequence.

Using Big-O notation, how long will each part take if you implement with an array?

|  |  |  |
| --- | --- | --- |
|  | Array | Linked List |
| Locate position |  |  |
| Insert |  |  |

**c.** [1] In the previous question, suppose you expect that insertions into the middle of the list will be very rare. Most insertions will be quite close to the end of the list.

Does this affect your decision about whether to use a linked list or array in your implementation?

**d.** [1]Arrays are fast at jumping quickly to a given element, whereas linked list are fast at insertion after a given element. You would like to design a hybrid data structure that is fast at both.

How about an array whose elements were the nodes of a linked list, so that you could use the array to jump right to the 173rd node and then use the linked list to insert a new node at position 174? Does this work?

Explain.