|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Part** | **1** | **2** | **3** | **4** | **Total** |
| *maximum* | **25** points | **25** points | **25** points | **25** points | **100**G101010 pointsG |
| ***Your Score*** |  |  |  |  |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lists**

Reading Assignment: Thoroughly read Chapter 9 in the course textbook.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Part 1 Glossary Terms**

Define, in detail, each of these glossary terms from the realm of computer programming logic and design and computer topics, in general. If applicable, use examples to support your definitions. Consult your notes or course textbook(s) as references or by visiting Web sites such as: [**http://www.ask.com**](http://www.ask.com),[**http://www.bing.com**](http://www.bing.com), [**http://www.webopedia.com**](http://www.webopedia.com)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**(a) ArrayListIterator class**

|  |
| --- |
| A list iterator used for position-based operations and moves a “cursor” through a list to certain positions. Unlike a simple iterator, the list iterator supports movement to previous positions, directly to the 1st position, and directly to the last position and insertions, removals, and replacements at each of these positions (Lambert)  The list iterator has operations that depend on a synced up “modification count” between the list it is iterating over (it’s “backing store”) and it’s own modification count. If this is out of line, then the list iterator raises an AttributeError(“Illegal modification of backing store”). All operations / mutations done on a backing store must be done by the list iterator.   The reason the list iterator does this is that it must keep an accurate position for its cursor, and if modifications were made to the backing store that were not accounted for by the iterator, then the cursor position would be inaccurate. |

**(b) Circular Linked List**

|  |
| --- |
| A circular linked list uses TwoWayNodes with a previous and a next link, and an external head pointer (called a dummy header or sentinel node) that always has a “previous” link to the tail or final node in the list, and a “next” link to the first node in the list.  The benefit of a circular linked list is that it supports O(k) removal time for popping the first or last item in a linked list, as well as forward (next) and backwards (previous) traversal of the list with a list iterator. |

**(c) Node**

|  |
| --- |
| A node is an object that contains two elements: A data item, and a link to the next node in a collection. A two-way node contains a link to the next node as well as a link to a previous node in a collection. |

**(d) Round - Robin Scheduling**

|  |
| --- |
| Round-Robin scheduling is actually implemented using a priority queue, and not a list. In it, processes are inserted into a queue based on their priority (e.g., processes that take very little time to process take precedence in the queue while processes that take a long time take less precedence), and then popped out of queue when a slice of CPU becomes available for them.  The PVM uses Lists to allocate various segments of the Object Heap, an area of memory where the PVM allocates segments of various sizes for all new data objects (Lambert 249). Multiple lists of memory are created and sorted in an optimal way based on byte size and free space within memory, and when allocating memory for a new object, the PVM searches the first and last positions in these lists for an appropriate free space before executing a linear search. |

**(e) Time and Space Analysis**

|  |
| --- |
| Time analysis can be conducted via benchmarking, or, using a computer’s clock to judege the time complexity of an algorithm. It is not very reliable however as computers vary greatly in terms of CPU and RAM.  The running time of an algorithm is better calculated using Big O notation, and using simple algebra to determine the number of iterations an algorithm would take (e.g., constant O(k), logarithmic O(log2n), linear O(n), quadratice O(n^2), etc.)  Space analysis is conducted by analyzing the amount of memory an algorithm would require to run, according to it’s input size (n). Byte sizes are analyzed depending on object type (integer, float, string which is really a list of n-byte-sized characters, lists etc.) and space complexity is calculated using big O notation as well.  It is entirely possible for an algorithm to have trade-offs in time complexity and space complexity, where an algorithm may run faster but take up more memory, and vice-a-versa. |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Part 2 True / False Exercises**

For each of these exercises, enter True or False in the spaces provided.

**FALSE** **(1)** A list supports manipulation of items at only the head and tail within a linear collection.

**FALSE (2)** Items in a list must be sorted to make sense.

**TRUE for linked list (3)** The items in a list are logically contiguous but need not be physically contiguous in memory.

**FALSE** **(4)** For a linked implementation of a list, a singly linked structure allows movement to the next node or the previous node.

**TRUE** **(5)** In the implementation of the lister iterator on a doubly linked list, the method hasNext returns False if the cursor is less than the backing store's length.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Part 3 Multiple Choice Exercises**

Select the correct response or responses.

**(1)** Which of the following is NOT an example of a list?

(a) a recipe

**(b) a jar of marbles**

(c) words on a document

(d) a file on a disk

**(2)** Which of the following is true about lists?

(a) items must be sorted

(b) items must be physically contiguous

(c) the order of items is unimportant

**(d) items are logically contiguous**

**(3)** What is true about how lists are indexed?

(a) indices increase in both movement directions

(**b) indices decrease to the left and increase to the right**

(c) indices decrease in both movement directions

(d) indices decrease to the right and increase to the left

**(4)** Why can you refer to an item in a list via its relative position from the head of the list using an index?

(a) because it has a circular structure

(b) because it uses round - robin

**(c) because it is ordered linearly**

(d) because it's arranged in a hierarchy

**(5)** What is the typical argument to a content-based list operation?

(a) an index

(**b) an item**

(c) a list instance

(d) an array

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Part 4 Programming Exercises**

**(1)** Explain how a list iterator differs from a **for** loop.

|  |
| --- |
| A list iterator supports forward and backward list traversals of a list as well as list mutations such as remove, insert, and replace. |

**(2)** In the following code for the **\_\_iter\_\_** method in the **ArrayList** class, what is the missing code?

**def \_\_iter\_\_(self) :**

**cursor = 0**

**while cursor < len(self) :**

**yield self.items[cursor]**

**# <missing code>**

MISSING CODE IS:

cursor += 1

**(3)** Explain, in detail, what is accomplished by the following code segment.

|  |
| --- |
| The code segment defines a node as well as a LinkedList. The LinkedList class has a listprint method that iterates through each link list to print out data from each node.   The statements below the class defintions instantiate a LinkedList, mutate the list directly by setting its head value to a “First” node, and creates other instances of nodes and assigns those to list.headvalue.next, essentially creating a chain of nodes linked to one another.   The listprint() statement returns each data element in each node: First Second Third |

|  |
| --- |
| **class Node() :**  **def \_\_init\_\_(self, data = None) :**  **self.data = data**  **self.next = None**  **class Linkedlist:**  **def \_\_init\_\_(self) :**  **self.headvalue = None**  **def listprint(self):**  **printvalue = self.headvalue**  **while printvalue is not None:**  **print (printvalue.data)**  **printvalue = printvalue.next**  **list = Linkedlist()**  **list.headvalue = Node("first")**  **x2 = Node("second")**  **x3 = Node("third")**  **list.headvalue.next = x2**  **x2.next = x3**  **list.listprint()** |

**(4)** The following program segment creates a linked list.

Modify the code such that the list's head value is displayed to the user.

|  |
| --- |
| class Node():  def \_\_init\_\_(*self*, *data*):  *self*.data = *data*  *self*.next = None  class Linkedlist():  def \_\_init\_\_(*self*):  *self*.head = None  *self*.last\_node = None  def append(*self*, *data*):  *if* *self*.last\_node is None:  *self*.head = Node(*data*)  *self*.last\_node = *self*.head  *else*:  *self*.last\_node.next = Node(*data*)  *self*.last\_node = *self*.last\_node.next  def display(*self*):  curr = *self*.head  *while* curr is not None:  print(curr.data, *end*=' ')  curr = curr.next  def displayHead(*self*):  print(*self*.head.data)  my\_llist = Linkedlist()  a = int(input("How many elements would you like to add? "))  *for* i *in* range(a):  data = int(input('Enter data item: '))  my\_llist.append(data)  print("The linked list: ", *end*=' ')  my\_llist.display()  *# Adding new feature*  print("\nThe head item in the linked list is: ")  my\_llist.displayHead()  **RETURNS**  **How many elements would you like to add? 3**  **Enter data item: 1**  **Enter data item: 2**  **Enter data item: 3**  **The linked list: 1 2 3**  **The head item in the linked list is:**  **1** |

**(5)** Examine the following program code which establishes a linked list.

Modify the program that includes a new function which will display the latest head value of the list for each new item added.

|  |
| --- |
| class Node:  *# singly linked node*  def \_\_init\_\_(*self*, *data*=None):  *self*.data = *data*  *self*.next = None  class singly\_linked\_list:  def \_\_init\_\_(*self*):  *# create an empty list*  *self*.head = None  *self*.tail = None  *self*.count = 0  def iterate\_item(*self*):  *# iterate through the list*  current\_item = *self*.head  *while* current\_item:  val = current\_item.data  current\_item = current\_item.next  *yield* val  def append\_item(*self*, *data*):  *# append items on the list*  node = Node(*data*)  *if* *self*.tail:  *self*.tail.next = node  *self*.tail = node  *else*:  *self*.head = node  *self*.tail = node  *self*.count += 1  def get\_head(*self*):  *if* *self*.head:  *return* str(*self*.head.data)  *return* "No head!"  def get\_tail(*self*):  *if* *self*.tail:  *return* str(*self*.tail.data)  *return* "No tail!"  items = singly\_linked\_list()  items.append\_item("item 1")  items.append\_item("item 2")  items.append\_item("item 3")  items.append\_item("item 4")  items.append\_item("item 5")  *for* val *in* items.iterate\_item():  print(val)  print("\nhead.data: ", items.get\_head())  print("tail.data: ", items.get\_tail())  **RETURNS  item 1**  **item 2**  **item 3**  **item 4**  **item 5**  **head.data: item 1**  **tail.data: item 5** |