ESTRUCTURA DE DATOS 1
Código ST0245

Laboratory practice No. X: Linked lists and Dynamic vectors

Mariana Quintero Montoya
Universidad Eafit
Medellín, Colombia
mquintero3@eafit.edu.co

Isabella Pérez Serna
Universidad Eafit
Medellín, Colombia
iperezs2@eafit.edu.co

3) Practice for final project defense presentation

3.1

Exercise	ArrayList	LinkedList
1.1	O((n^2)*m)	O(n*m)
1.4		O(n*m)

3.2

To solve the problem of the broken keyboard, we must create: a linked list of strings, a variable in charge of passing through each character of the original text, a string that will be a temporary variable which will be inserted into the list and finally a variable that allows us to determine how true or false the content should be inserted at the end or beginning of the list respectively. Now we must implement a cycle that goes through each character of the string received and also identifies whether the character you are reading is a "[" or "]", because if it is, first insert the temporary string at the end and then at the beginning of the list according to the value of the boolean variable described above, if the current character is then "[" represent that the variable is false, that is that the next content would go to the beginning, and if it is "]" would mean the opposite. To finish, it is necessary to give the temporary variable an empty string so that it can continue to the next word to be treated, now, if the character is none of those, then it will be inserted to the temporary variable. And in the case that no character "[" or "]" is entered, the comparison of the value of the boolean variable should be done in order to give the instruction to insert at the end if it is true or at the beginning if it is false. Once the process is finished, simply store each node in the ordered list in a string variable.

3.3

The complexity of point 2.1 is given by $T(n,m) = n + m + 1 = n + m = O(n + m)$

3.4

n: Number of characters of the string to be ordered
m: Represents the nodes of the list, that is, the separate words of the chain to be sorted.

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473

4) Practice for midterms

4.1 *[OPC]*
4.11 B
4.12 B

4.2 **[OPC]** B

4.3 *It is empty.*

4.4
4.41 output.append(token).append(‘ ’);
4.42 B

4.5 **[OPC]** A

4.6 **[OPC]** A

4.7 It is empty.

4.8 C
4.81 A
4.82 C
4.83 B

4.9 **[OPC]**
4.91 D
4.92 A
4.93 B

4.10
4.101 C
4.102 B

4.11 Empty

4.12
4.121 Line 13: while(¡s1.isEmpty());
4.122 Line 14: s2.push(s1.pop());
4.123 Line 17: return s1.head;
4.124 iv)
4.125 i)

4.13
4.131 iii)
4.132 iii)

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473

4.14 **[OPC]** iii)

5) Recommended reading (optional)

Chapter 3: Linked Lists

3.2 Individual link lists In previous sections of the text they presented the data structure of the set and some of its applications. Arrays are nice and simple for storing things in a certain order, but they have the disadvantage of not being very adaptable, since we have to set the N-size of the array in advance. However, there are other ways to store a sequence of items that do not have this drawback. In this section, we explore an important alternative implementation, which is known as the individual link list. A linked list, in its simplest form, is a collection of nodes that together form a linear order. The order is determined as in the children's game "Follow the Leader", in which each node is an object that stores a reference to an element and a reference, called "next", to another node. In Figure 3.10 of the original text: Example of a joined list whose elements are strings that indicate airport codes. The following pointers to each node are shown as arrows. The null object is denoted as ¬. It may seem odd for one node to reference another node, but such a scheme works easily. The following reference within a node can be seen as a link or pointer to another node. Similarly, moving from one node to another following a reference is known as a linkage or pointer jump. The first and last node in a list of links are usually called the head and tail of the list, respectively. So, we can link the jump through the list starting from the head and ending from the tail. We can identify the queue as the node that has a null following reference, which indicates the end of the list. A linked list defined in this way is known as an individually linked list. Like an array, an individually linked list keeps its elements in a certain order. This order is determined by the chain of subsequent links from each node to its successor in the list. Unlike an array, a single-link list has no fixed predetermined size, and uses space proportional to the number of its elements. Likewise, we don't track any index numbers for the nodes in a linked list. So we cannot tell just by looking at a knot whether it is the second, fifth, or twentieth knot in the list. Implementing an individually linked list To implement an individually linked list, we define a node class. Here we assume that the elements are strings. In chapter 5, we describe how to define nodes that can store arbitrary element types. Given the Node class, we can define a class, SLinkedList, shown in Code Snippet 3.13 of the text, which defines the actual list of links. This class keeps a reference to the main node and a variable that counts the total number of nodes. Code Snippet 3.12 of the text: Implementation of a node from an individually linked list. Code snippet 3.13 of the text: Partial implementation of the class for an individually linked list.
3.2.1 Insertion in a simple link list When using a simple link list, we can easily insert an element in the list header, as shown in figure 3.11 in the text and code snippet 3.14 in the text. The main idea is that we create a new node, set its next link to refer to the same object as the header, and then set the header to point to the new node. Figure 3.11 of the text: Inserting an element into the head of a list of individual links: a) before insertion; b) creating a new node; c) after insertion. Code fragment 3.14 from the original text: Insert a new v-node at the beginning of a list of individual links. Note that this method works even if the list is empty. Note that we set the following pointer for the new v-node before making the variable head point to v. Inserting an element into the queue of an individually linked list We can also easily insert an element into the

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473

queue of the list, as long as we keep a reference to the queue node, as shown in figure 3.12 of the text. In this case, we create a new node, assign its next reference to point to the null object, set the next queue reference to point to this new object and then assign the queue reference itself to this new node. We give the details in Code Snippet 3.15. Figure 3.12: Inserting an individually linked list into the queue: a) before insertion; b) creating a new node; c) after insertion. Observe that we establish the following link for the queue in (b) before assigning the queue variable to point to the new node in (c). Code snippet 3.15: Inserting a new node at the end of a list of individual links. This method works also if the list is empty. Note that we set the following pointer to the old tail node before making the tail variable point to the new node. 3.2.2 Removing an element from an individually linked list The reverse operation of inserting a new element into the header of a linked list removes an element from the header. This operation is illustrated in figure 3.13 and detailed in code snippet 3.16. Figure 3.13: Removing an element in the header of an individually linked list: a) before removal; b) "linking" the new old node; c) after removal. Code fragment 3.16: Elimination of the node at the head of an individually linked list. Unfortunately, we cannot easily remove the tail node from an individually linked list. Even if we have a tail reference directly to the last node in the list, we must be able to access the node before the last node in order to remove the last node. But we cannot reach the node before the queue by following the queue links below. The only way to access this node is to start at the top of the list and search through the entire list. But such a sequence of link hopping operations could take a long time.

3.3 Double-linked lists g an element in the queue of an individually linked list is not easy In fact, it takes a long time to remove any node other than the head - front and back - in a linked list. This is the double-linked list. As we saw in the previous section, deleting in an individually linked list is not a quick way to access the node before the one we want to delete. In fact, there are many applications where we do not have quick access to such a predecessor node. For such applications, it would be good to have a way to go both ways in a linked list. There is a type of linked list that allows us to go in both varieties of quick update operations, including insertion and removal at both ends, to in the middle. A node in a double-linked list stores two references: a next link pointing to the next node in the list, and a previous link, pointing to the previous node in the list. A Java implementation of 3.17, where we assume that the elements are strings. In chapter 5, we discuss how to define nodes for arbitrary element types. Code snippet 3.17: Java class DNode that represents a node from a double-linked list that stores a string. Header and Trailer Sentinels To simplify programming, it is convenient to add special nodes to both ends of a double-linked list: a header node just before the head of the list, and a trailer node just after the tail of the list. These dummy nodes do not store any items. The header has a valid next reference but a null previous reference, while the trailer has a valid previous reference but a null next reference. Figure 3.14 shows a list doubly linked to these sentinels. Note that a linked list object would simply need to store references to these two sentinels and a size counter that keeps track of the number of items (not counting sentinels) in the list. Figure 3.14: A double-linked list with sentinels, header and trailer, marking the ends of the list. An empty list would have these sentinels pointing at each other. We don't show the previous null pointer for the header, nor do we show the next null pointer for the trailer. Inserting or removing items at either end of a double-linked list is very simple. In fact, pre-linking eliminates the need to traverse the list to get to the node just before the queue. We show the removal of a double-linked list from the queue in figure 3.15 and the details of this operation in code snippet 3.18. Figure 3.15: Node removal at the end of a double-linked list with header and towing sentries: a) before the tail

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473

removal; b) tail removal; c) after the removal. Code fragment 3.18: Elimination of the last node in a double-linked list. The variable size keeps a record of the current number of elements in the list. Note that this method also works if the list is size one. In the same way, we can easily perform an insertion of a new element at the beginning of a double-linked list, as shown in Figure 3.16 and in Code Snippet 3.19. Figure 3.16: Adding an element at the beginning: a) during; b) after. Code snippet 3.19: Inserting a new node v at the beginning of a double-linked list. The variable size keeps a record of the current number of elements in the list. Note that this method also works in an empty list.

3.3.1 Inserting in the middle of a double-linked list Double-linked lists are useful for more than just inserting and removing items in the list header and queue, however. They are also convenient for maintaining a list of items while allowing insertion and removal in the middle of the list. Given a node v in a double-linked list (which could possibly be the header but not the trailer), we can easily insert a new node z immediately after v. Specifically, let the node w be following v: 1. make the previous link from z refers to v 2. make the next link from z refers to w 3. make the previous link from w refers to z 4. make the next link from v refers to z This method is given in detail in Code Snippet 3.20, and is illustrated in Figure 3.17. Recalling our use of head and trailer sentries, note that this algorithm works even if v is the tail node (the node just before the trailer). Code snippet 3.20: Insert a new z-node after a given v-node in a double-linked list. Figure 3.17: Adding a new node after the node that stores the JFK: a) creating a new node with the BWI element and linking it; b) after the insertion.

3.3.2 Deleting in the middle of a double-linked list In the same way, it's easy to delete a v-node in the middle of a double-linked list. We access the u and w nodes on either side of v using the getPrev and getNext methods of v (these nodes must exist, as we are using sentinels). To remove the v node, we simply have u and w pointing at each one instead of a v. We refer to this operation as the link outside of v. We also override the prev and next pointers of v so as not to retain old references in the list. This algorithm is shown in code fragment 3.21 and is illustrated in figure 3.18. Code snippet 3.21: Deleting a v-node in a double-linked list. This method works even if v is the first, last or only unlinked node. Figure 3.18: Removing the node that stores PVD: a) before removal; b) linking the old node; c) after removal (and garbage collection).

3.3.3 An implementation of a double-linked list In code snippets 3.22-3.24 we show an implementation of a double-linked list with nodes that store string elements. Code snippet 3.22: Java class DList for a double-linked list whose nodes are DNode class objects (see code snippet 3.17) that store character strings. (Continued in Code Snippet 3.23.) 176 Code Snippet 3.23: Java class DList for a double-linked list. (Continued in Code Snippet 3.24.) Code Snippet 3.24: A class of double-linked list. (Continued in Code Snippet 3.23.) We make the following observations about the class DList above. - Objects of the DNode class, which store String elements, are used for all the nodes in the list, including the header and trailer sentinels. - We can use the DList class for a double-linked list of String objects only. To build a linked list of other types of objects, we can use a generic declaration, which we will discuss in chapter 5. - The getFirst and getLast methods provide direct access to the first and last node of the list. - The getPrev and getNext methods allow you to browse the list. - The hasPrev and hasNext methods detect the limits of the list. - The addFirst and addLast methods add a new node to the beginning or end of the list. - Methods, add Before and add After add a new node before or after an existing node. - Having only one removal method, remove, is not really a restriction, since we can remove at the beginning or at the end of a double-linked list L by executing L.remove(L.getFirst()) or L.remove(L.getLast()), respectively. - The toString method for converting an entire list into a string is useful for testing and debugging purposes.

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473

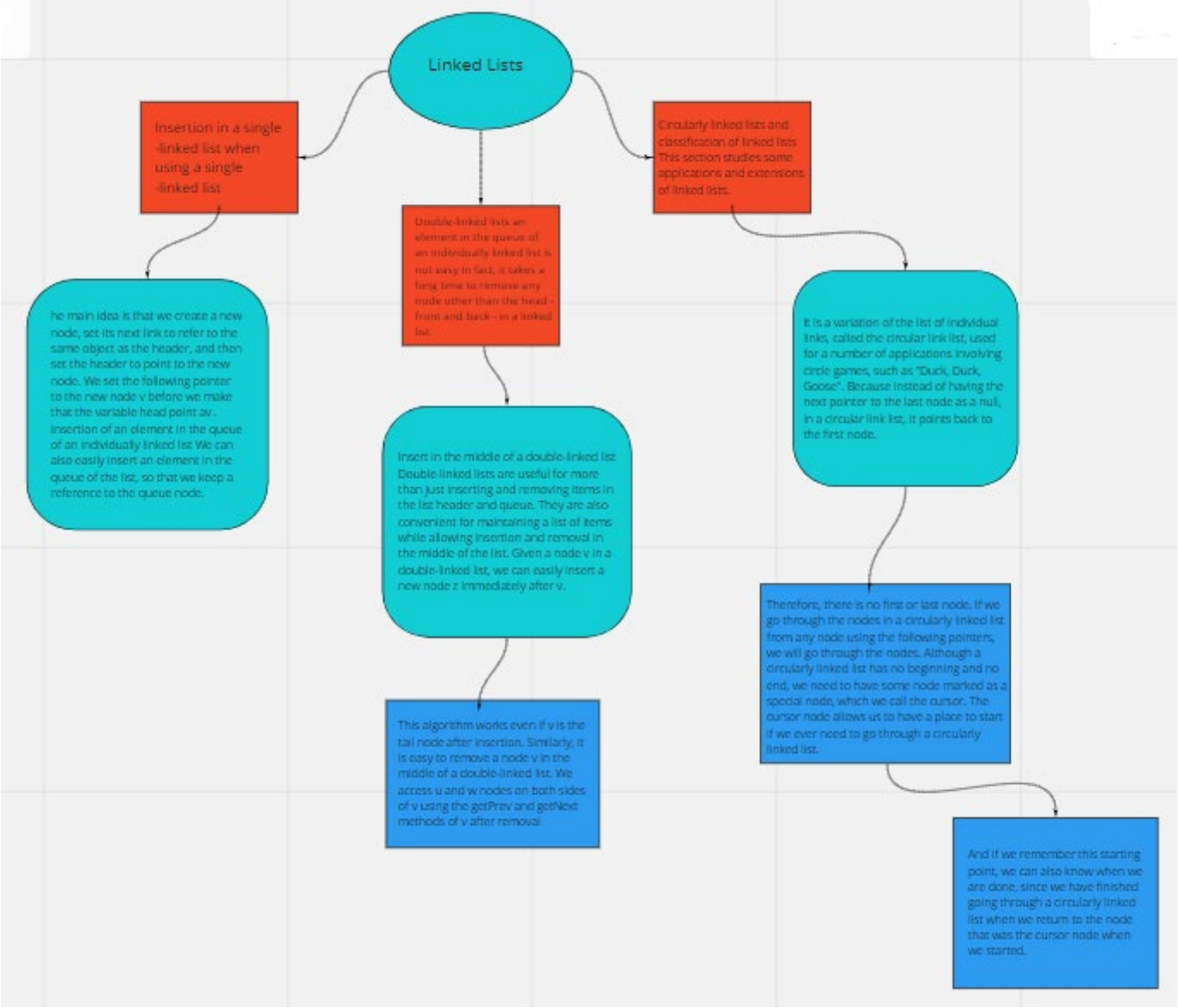
3.4 Circularly linked lists and classification of linked lists This section studies some applications and extensions of linked lists.
3.4.1 Circularly linked lists and Duck, Duck, Goose The children's game "Duck, Duck, Goose" is played in many cultures. Children in Minnesota play a version called "Duck, Duck, Goose" (but please don't ask us why.) In Indiana, this game is called "The Mosh Pot". And children in the Czech Republic and Ghana play versions of songs known respectively as "Pesek" and "Antoakyire". ." A variation of the individual link list, called the circular link list, is used for a series of applications involving circle games, such as "Duck, Duck, Goose". We discuss this type of list and the application of circle games below. A circle-linked list has the same type of nodes as an individually linked list. That is, each node in a circularly linked list has a next pointer and a reference to an element. But there is no head or tail in a circularly linked list. Because instead of having the next pointer to the last node as a null, in a circularly linked list, it points back to the first node. Therefore, there is no first or last node. If we go through the knots in a circularly linked list from any node using the following pointers, we will cycle through the knots. Although a circularly linked list has no beginning or end, we need some node to be marked as a special node, which we call the cursor. The cursor node allows us to have a place to start from if we ever need to go through a circularly linked list. And if we remember this starting point, then we can also know when we have finished, since we have finished traversing a circularly linked list when we return to the node that was the cursor node when we started. We can then define some simple update methods for a circularly linked list: add(v): Insert a new v node immediately after the cursor; if the list is empty, then v becomes the cursor and its next pointer points to itself. remove(): Remove and return the v node immediately after the cursor (not the cursor itself, unless it is the only node); if the list is empty, the cursor is set to null. 180 advance(): Advances the cursor to the next node in the list. In Code Fragment 3.25, we show a Java implementation of a circularly linked list, which uses the Node class of Code Fragment 3.12 and also includes a toString method to produce a chain representation of the list. Code fragment 3.25: A class of circularly linked list with single nodes. Some observations about the CircleList class There are some observations we can make about the CircleList class. It's a simple program that can provide enough functionality to simulate circle games like Duck, Duck, Goose, as we will show soon. However, it is not a robust program. In particular, if a list of circles is empty, then calling up the advance or removal of that list will cause an exception. (Which one?) Exercise R-3.5 deals with this exception-generating behavior and ways to better manage this empty-list condition. Duck, Duck, Goose In child's play, Duck, Duck, Goose, a group of children sit in a circle. One of them is chosen "it" and that person walks around the outside of the circle. The person who is "it" pats each child on the head, saying "Duck" each time, until they reach a child that the person "it" identifies as "Goose. At this point there is a crazy struggle, as the "Goose" and the person doing it run around the circle. Whoever returns to the place of the Goose first stays in the circle. The loser of this race is the most important person for the next round of play. The game continues like this until the children get bored or an adult tells them it's time for snack, at which point the game ends. (See Figure 3.19.) Figure 3.19: The game of Duck, Duck, Goose: a) the choice of the "Goose"; b) the race to the place of the "Goose" between the "Goose" and the "it". The simulation of this game is an ideal application of a list of circular links. Children can represent nodes from the list. The "it" person can be identified as the person sitting behind the cursor, and can be removed from the circle to simulate the walk around. We can advance the cursor with each "Duck" that the "it" person identifies, which we can simulate with a random decision. Once a "Goose" is identified, we can remove this knot

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473

from the list, make a random choice to simulate whether the "Goose" or the "person" wins the race, and insert the winner again in the list. We can then advance the cursor and insert the "it" person again to repeat the process (or do it if it's the last time we play the game). Using a circularly linked list to simulate Duck, Duck, Goose We give you Java code for a simulation of Duck, Duck, Goose in code snippet 3.26. Code snippet 3.26: The main method of a program that uses a circularly linked list to simulate the children's game Duck, Duck, Goose We show an example of output from a run of the program Duck, Duck, Goose in figure 3.20. Figure 3.20: Example output of the program Duck, Duck, Goose. Note that each iteration in this particular run of this program produces a different result, due to different initial settings and the use of random choices to identify ducks and geese. Similarly, whether the "Duck" or the "Goose" wins the race is also different, depending on the random choices. This execution shows a situation where the next child after the "it" person is immediately identified as the "Goose", as well as a situation where the "it" person walks all the way around the group of children before identifying the "Goose". Such situations also illustrate the usefulness of using a circularly linked list to simulate circular games such as Duck, Duck, Goose.

3.4.2 Sorting a linked list in code fragment 3.27 we show the insertion sorting algorithm (section 3.1.2) for a double-linked list. A Java implementation is shown in code snippet 3.28. Code snippet 3.27: High-level description of the insertion-sorting pseudocode in a double-linked list. Code snippet 3.28: Java implementation of the insertion-sorting algorithm in a double-linked list represented by the class DList.

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473



PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473