# STUDENT OUTLINE

## Lesson 31 – Linked-List Algorithms

**INTRODUCTION:** The linked list in this lesson will have the following special characteristic: the random order of the incoming data will be stored in ordered fashion based on a key field. After creating the linked list, a variety of algorithms will be discussed and solved.

The key topics for this lesson are:

A. Building an Ordered Linked List
B. Linked-List Algorithms
C. Static vs. Dynamic Data Structures

**VOCABULARY:** INTERNAL POINTER EXTERNAL POINTER

**DISCUSSION:** A. <u>Building an Ordered Linked List</u>

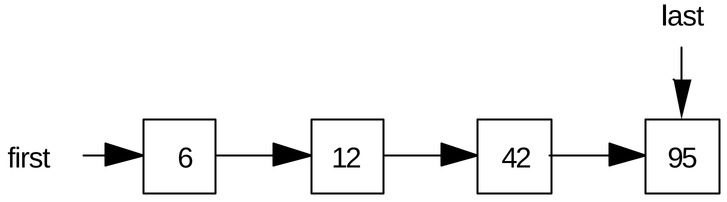
1. If data is supplied in unordered fashion, building an ordered linked list becomes a harder problem. Consider the following cases for inserting a new value into the linked list:

   a. Insert the new value into an empty list.
   b. Insert the new value at the front of the list.
   c. Insert the new value at the tail of the list.
   d. Insert the new value between two nodes.

2. Suppose the data was supplied in this order:

   42 6 95 12 <eof>

   The resulting ordered linked list looks like this:



3. Three of the cases are easy to identify and solve: the empty list case, placing the new value at the front of the list, or adding the value to the end of the list.

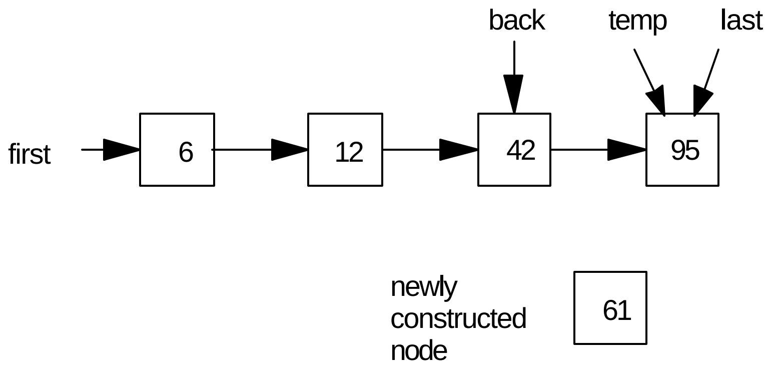
4. To assist us in our discussion of linked list algorithms, some definitions will be helpful. An internal pointer is one that exists inside a node. An internal pointer joins one node to the next in the linked list. An external pointer is one that points to a node from outside the list. Every linked data structure must have at least one external pointer that allows access to the data structure. Our linked list in the student outline has two external pointers, `first` and `last`.

5. Suppose a fifth value, 61, is to be inserted into the list. We can see in the diagram that it will go between the 42 and 95. To help the computer "see" this, we are going to use some helping external pointers, also called auxiliary pointers.

6. It is possible to find the attachment point using just one auxiliary pointer, but we will use two called `temp` and `back`. Using two external pointers makes the hookup easier. Finding the attachment point involves something like this:

```
while we haven't found the attachment point
{
   back = temp;             //move back up to temp
   temp = temp.getNext();  //advance temp one node ahead
}
```

For attaching the value 61, `back` and `temp` will end up pointing to these locations:



B. <u>Linked-List Algorithms</u>

1. Searching an ordered linked list is a sequential search process. A linked list is not a random access data structure. You cannot jump to the middle of a linked list. Only sequential moves are possible. The search function could return a value or a pointer to that cell.

2. Deleting a value involves the following steps:
   a. Locating the value (if it exists) to be deleted.

b. Rehooking pointers around the node to be deleted.

C. <u>Static vs. Dynamic Data Structures</u>

1. An *array* is a somewhat static data structure that has these advantages and disadvantages:

| Advantages of an *array* | Disadvantages of an *array* |
|---|---|
| 1. Easy to implement and use.<br>2. Fast, random access feature. | 1. Memory is usually wasted. |

2. A linked list (LL) is a dynamic data structure which has these advantages and disadvantages.

| Advantages of LL | Disadvantages of LL |
|---|---|
| 1. Memory is allocated when the program is run, therefore the data structure is only as big as it needs to be.<br>2. Memory is conserved. | 1. Each node of the list takes more memory.<br>2. Processing is slower.<br>3. The data structure is not random access.<br>4. Processing must be done in sequential order. |

**SUMMARY/ REVIEW:** We now have two different data structures to implement lists: arrays or linked lists. Each has its appropriate place in a programmer's tool kit. We will discuss variations of singly linked lists in later lessons.

**ASSIGNMENT:** Lab Exercise L.A.31.1, *OrderedList*

## LAB EXERCISE

## OrderedList

**Background:**

This lab will use a text file, *'file20.txt'*, which is similar to the one used in the binary search lab in Lesson 28. The file has been saved in random order by `id` number. Your program must build an ordered linked list based on the `id` field. The ordered list should be implemented as a `SinglyLinkedList` of type `ListNode`.

**Assignment:**

1. Here are some of the specifications for the methods to be added to the `SinglyLinkedList` class:

   a. You are to write a method `insert` that builds the linked list in order based on the `id` value.

   b. A `find` method will check the list for a specific id value, returning a reference to such a node if the value exists in the list. If the value is not found, the method should return a **`null`** value.

   c. A `remove` method will remove unwanted data from the linked list.

   d. Write a `clear` method that clears the entire list.

   e. Write a `size` method that returns the number of nodes in the list. Avoid copying this code from the last lesson and write it from scratch.

   f. Reading the data file is an identical process to that used in *Store.java* in Lesson 27.

   g. Printing the list involves the same code as used in the previous lesson on linked lists.

   h. Code a recursive `printBackward` method that prints out the linked list contents in reverse order.

2. If your instructor chooses, you will be provided with a program shell consisting of a *main* menu, testing methods, and stubbed methods. Here are some of the specifications of this program shell.

   a. A method `addLast` is provided which builds the list just as in lab L.A.31.1, `List1`. The list will be built in the same order as it exists in the data file. The routine that reads the data from the text file has been separated from the method to insert the information into the linked list. See the code in the program shell.

   b. The `find` method returns a **`null`** value.

   c. The `remove` method returns a **`null`** value.

   d. The `clear` method is stubbed out as a print statement.

e. The `printBackwards` method is stubbed out as a print statement.

f. The `size` method returns 0.

g. Methods to read the data file and print the list are provided.

**<u>Instructions:</u>**

1. Modify and write code as necessary to satisfy the above specifications.

2. Print out the entire source code.

3. Include a printed run output of the following:

    a. The entire list printed in ascending order. Include a call of the `size` method and print the number of nodes.

    b. Print the list backward using the recursive `printBackward` method. Do not worry about printing a line number in this algorithm.

    c. The `id` 's you searched for and their corresponding inventory amounts. Your instructor will specify which `id` values to search for.

    d. Delete `id` values as specified by your instructor. Print the abbreviated list after values have been deleted and include a call to `size`.

    e. Clear the entire list of all nodes. Call the `size` method one last time.