CSIS216 Data Structures Project Spring 24-25

Introduction
A linked list is a linear collection of data elements whose order is not given by their physical placement in memory. Instead, each element in the list points to the next.
In this course, we have seen how pointers and dynamic allocation and deallocation operations can be used in implementing linked lists. A different way to implement linked lists without using pointers and dynamic allocation and deallocation is by using arrays of structs (or classes). In this project, you are asked to provide an array-based implementation of a linked list.

Using the ideas presented below you are required to:

1. develop a template library named "NodePool.h" that manages the array **arrNode** and the list of free nodes along with functions to get a new node or to return one to the free list.

2. define the template list class, using the .h file defined in **1.**

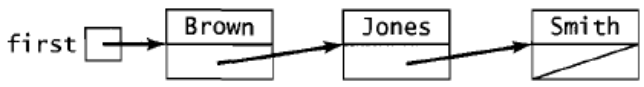
3. write a user-friendly tester to test the class list defined in **2.**

Node Structure
Nodes in a linked list contain two parts: a data part that stores an element of the list; and a next part that points to a node containing the successor of this list element or that is null if this is the last element in the list. This suggests that each node can be represented as a struct and that the linked list can be represented as an array of structs. Each struct will contain two members: a data member that stores a list element and a next member that will point to its successor by storing its index in the array. Thus, appropriate declarations for the array-based storage structure for linked lists are as follows:

```
/*** Node Declarations ***/
struct NodeType
{
    ElementType data;
    int next;
} ;

const int NULL_VALUE = -1;          //a nonexistent location
/*** The Storage Pool ***/
const int NUM_NODES = 2048;
NodeType node[NUMNODES];
int free;                           //points to a free node
```

To illustrate, consider a linked list containing the names Brown, Jones, and Smith in this order:

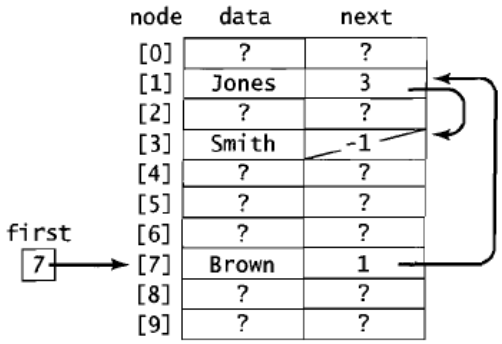


Here first is a variable of type int and points to the first node by storing its location in the storage pool node. Suppose for simplicity that NUMNODES is 10, so that the array node constructed by the preceding declarations consists of 10 structs, each of which has a data member for storing a name and a next member for storing the location of its successor. The nodes of the linked list can be stored in any three of these array locations, provided that the links are appropriately set and first is maintained as a pointer to the first node.

For example, the first node might be stored in location 7 in this array, the second node in location 1, and the third in location 3. Thus, first would have the value 7, node[7].data would store the string "Brown", and node[7].next would have the value 1. Similarly, we would have node[1].data = "Jones" and node[1].next = 3. The last node would be stored in location 3 so that node[3].data would have the value "Smith".

Since there is no successor for this node, the next field must store a null pointer to indicate this fact; that is, node [3].next must have a value that is not the index of any array location, and for this, the value -1 is a natural choice for the null value.

The following diagram displays the contents of the array node and indicates how the next members connect these nodes. The question marks in some array locations indicate undetermined values because these nodes are not used to store this linked list.



To traverse this list, displaying the names in order, we begin by finding the location of the first node by using the pointer first. Since first has the value 7, the first name displayed is "Brown", stored in node[7].data.

Following the next member leads us to array location node[7].next = 1, where the name "Jones" is stored in node [1].data, and then to location node[1].next = 3, where "Smith" is stored.

The null value -1 for node[3].next signals that this is the last node in the list.

In general, to traverse any linked list, we use the method given in the traversal algorithm:

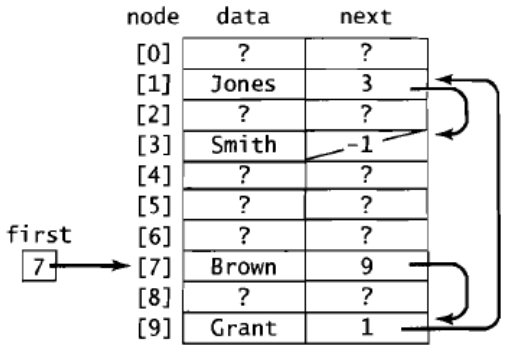
1. Initialize ptr to first.
2. While (ptr != NULL_VALUE) do the following:

a. Process the data part of the node pointed to by ptr.
b. Set ptr equal to the next part of the node pointed to by ptr.

End while

Now suppose we wish to insert a new name into this list, for example, to insert "Grant" after "Brown". We must first obtain a new node in which to store this name.

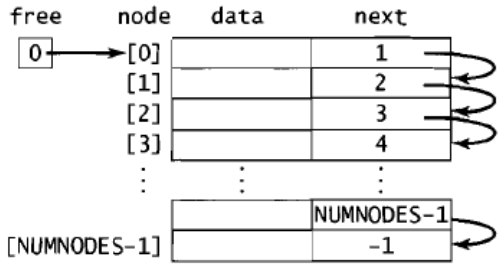
Seven locations are available, namely, positions 0, 2, 4, 5, 6, 8, and 9 in the array. Let us assume for now that this storage pool of available nodes has been organized in such a way that a call to a function **newNode()** returns the index 9 as the location of an available node. The new name is then inserted into the list using the method described in the preceding section: node[9].data is set equal to "Grant"; node[9].next is set equal to 2 so that it points to the successor of "Brown"; and the link field node[7].next of the predecessor is set equal to 9.



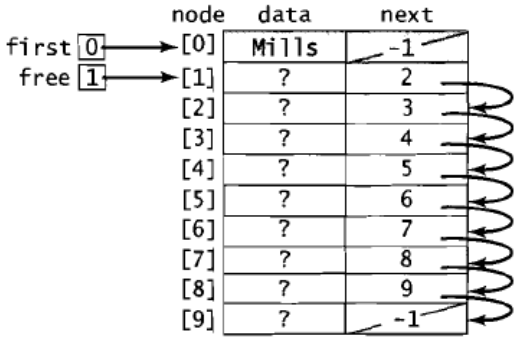
Organizing the Storage Pool

This example illustrates that the elements of the array node are of two kinds. Some of the array locations-namely 1,3,7, and 9-are used to store nodes of the linked list. The others represent unused "free" nodes that are available for storing new items as they are inserted into the list. We have described in detail how the nodes used to store list elements are organized, and we must now consider how to structure the storage pool of available nodes.

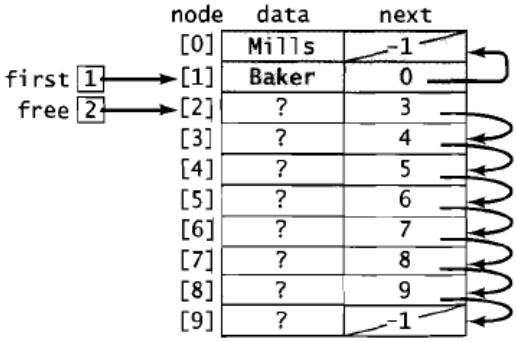
One simple way is to begin by linking all the nodes together to form the storage pool by letting the first node point to the second, the second to the third, and so on. The link field of the last node will be null, and a pointer free is set equal to 0 to provide access to the first node in this storage poot:



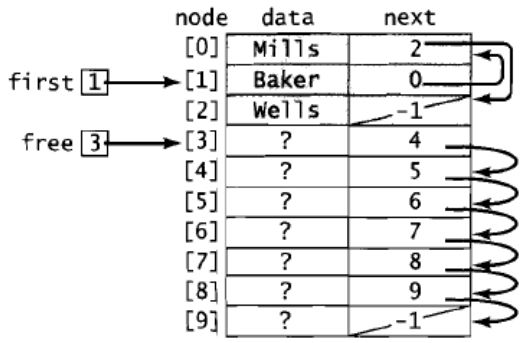
A function call ptr = newNode() returns the location of a free node by assigning to ptr the value free and removing that node from the free list by setting free equal to node[free].next. Thus, if "Mills" is the first name to be inserted into a linked list, it will be stored in the first position of the array node, because free has the value 0; first will be set to 0; and free will become 1:



If "Baker" is the next name to be inserted, it will be stored in location 1 because this is the value of free, and free will be set equal to 2. If the list is to be maintained in alphabetical order, first will be set equal to 1; node[l].next will be set equal to 0; and node [0].next will be set equal to -1:

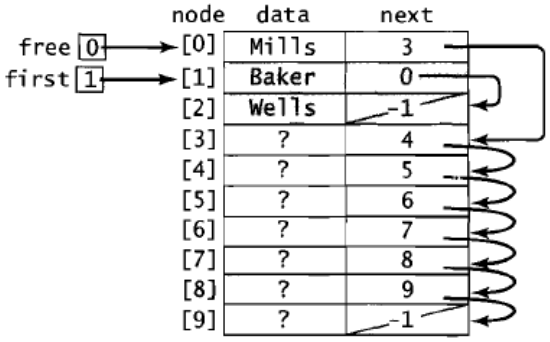


If "We115" is the next name inserted into the list, the following configuration will result:



When a node is deleted from a linked list, it should be returned to the storage pool of free nodes so that it can be reused later to store some other list element. A function call **deleteNode(ptr)** simply inserts the node pointed to by ptr at the beginning of the free list by first setting node[ptr].next equal to free and then setting free equal to ptr.

For example, deleting the name "Mi11s" from the preceding linked list produces the following configuration:



Note that it is not necessary to actually remove the string "Mills" from the data part of this node because changing the link of its predecessor has *logically* removed it from the linked list. This string "Mills" will be overwritten when this node is used to store a new name.