PROGRAMMING ASSIGNMENT #1
LINKED LISTS

DUE: Tuesday September 21, 2010 11:59PM

In this project we will use parallel arrays to implement linked lists. In some applications, this is the most natural method.

In linked allocation, there must generally be some pool from which new nodes can be drawn, and to which nodes that are no longer needed can be returned. Such a pool is commonly called "available storage", and is frequently implemented as a single long list. Call this list *avail*. To obtain a new node from available storage, one simply unlinks the front node of *avail*, and to return a node, simply re-link it back onto the front of *avail*. In programming languages with pointer types, this is managed by the system and one just uses library functions such as malloc (or new) and free (or dispose). However, some programming languages do not support pointers and we need to use other methods to implement linked lists.

For this assignment you will need an *avail* list. You should implement it using two arrays (named it *info* and *next*) of 100 nodes and an integer variable *avail* (-1 <= *avail* <= 99). A node will consist of an *info-next* pair. For simplicity, you may assume the *info* field contains a string. This *avail* list is the available pool of storage. You should also have an array *L* of 5 linked lists which your program will manipulate, see (4) below.

The assignment is as follows:

(1) Write a function of no arguments, *initialize* which will be called once at the beginning of your run. *initialize* should "prelink" the *info* array as shown below and initialize *avail* to point to the first available node (i.e. the node whose index is 0). Since its initial contents are irrelevant, the *info* fields should not be initialized. Use -1 to denote to null link. *initialize* should also set up the array *L* to be 7 empty linked lists. NOTE: in this project, your linked list should not have head nodes.

ARRAY INDEX	*info*	*next*
0		1
1		2
2		3
3		4
.		.
.		.
.		.
98		99
99		-1

avail is initialized to 0.

(2) Implement a function of no arguments, *fetch*, which unlinks the front node of *avail* and returns as its functional value the index of the acquired node.

(3) Implement a function of one argument, *dispose*(x), which returns the node whose index is x to the **front** of *avail.*

(4) Declare an array, $L[0..4]$, which will store 5 list pointers. Note that $-1 <= L[i] <= 99$ for $0 <= i <= 4$. In this project, these pointers are simply integers. The array L will allow us to store and manipulate up to 5 lists. You may have other variables. *Initialize* should initialize $L[0]$, ... $L[4]$ to null lists. Note that the elements of the lists need not be distinct.

(5) Implement the following functions:

(a) *addfirst*(X, i) Add a node containing information X to the front of list $L[i]$, as in *addfirst* (John, 3) which means: get a node from the *avail* list, place John into its INFO field, then link it onto the front of the list whose pointer is stored in $L(3)$.

(b) *addlast*(X, i) Add X to the end of $L[i]$. Thus, *addlast* (Mary, 3) would change the above list $L[3]$ to:

$L[3]$ --->John ---> Mary

(c) *deletefirst*(i) Delete the first node of list $L[i]$, returning the deleted node to *avail.*

(d) *deletelast*(i) Delete the last node of list $L[i]$, returning the deleted node to *avail.*

(e) *show*(i) Print the list $L[i]$ out. Print both *info* and *next* for each node by following the *link* fields.

(f) *reverse*(i) After this operation the order of elements in list $L[i]$ is reversed. For example:

$L[i]$ -----> John ----> Mary ----> David

becomes

$L[i]$ -----> David ----> Mary ----> John

The *info* field should not be copied; only the links of the nodes may be modified; no additional storage dependent on the size of the list should be used.

(g) *append* (j, k) After this operation, list $L[j]$ contains all elements of the original $L[j]$ and $L[k]$, and $L[k]$ becomes null.

(h) *sort* (j) After this operation, the names in $L[j]$ are in alphabetical order. The address of the list elements remains the same, only the next fields may be changed.

(6) Test your list functions thoroughly using your own data which you may encode any way you like. Your output must clearly indicate the sequence of the operations in your data. You may optionally print out trace messages to illustrate how various things work.

(7) Submit a copy of your program to the class Blackboard and hand in a hard copy of

- your well documented program written in a programming language of your choice
- the test data you use to test your program thoroughly indicating (handwritten is fine) the purpose of each item in your test data.
- an execution (showing output) of your program using your test data