Documentation for List.h and List.c

Steven Andrews, © 2015

Description

This library supports basic list management for sorted and unsorted lists. These lists are designed to be very simple to use, somewhat like the Python list data type. They use dynamic memory allocation, so that they expand themselves as needed.

The code that calls these lists should call the functions here for adding elements to lists and manipulating lists. However, it is fine for the code to read the list directly rather than through functions.

Currently supported list types are as follows

abbreviation	list
li	long integers
v	void*
dd	double, double
ULVD4	unsigned long long int, void*, double[4]

Dependencies

```
List.h
string2.h
```

History

10/7/15	Started. Wrote support for long integers.
8/22/16	Updated. Added void* list type.
12/3/20	Added double-double list type.
5/13/21	Added ULVD4 list type.

Data structures

```
typedef struct liststructli{
  int max;
  int n;
  long int *xs;
  } *listptrli;

typedef struct liststructv{
  int max;
  int n;
  void **xs;
  } *listptrv;
```

```
typedef struct liststructdd{
   int maxrow;
   int nrow;
   int maxcol;
   int ncol;
   double **data;
   } *listptrdd;

typedef struct liststructULVD4{
   int max;
   int n;
   unsigned long long *dataul;
   void **datav;
   double *datad4[4];
   } *listptrULVD4;
```

In the data structure, `max` gives the allocated size of the list. It is allowed to equal 0, in which case `xs` is equal to `NULL`. `n` is the number of elements currently in the list. `xs` is the actual list, which is allocated to size `max` and filled from element 0 to element `n`-1. The dd data structure is for a matrix of doubles, with `nrow` rows (allocated to `maxrow`) and `ncol` columns (allocated to `maxcol`). The data are stored in a single array (`row*maxcol+col`), rather than a vector of vectors.

Code documentation

Internal functions

`int List_ExpandLI(listptrli list,int spaces);`
: Expands memory allocated for existing list `list` by `spaces` spaces, without changing list contents. `spaces` is allowed to be negative for list shrinking. The list can be shrunk sufficiently that some contents are lost. Returns 0 for success or 1 for unable to allocate memory.

`int List_ExpandV(listptrv list,int spaces);`
: Expands memory allocated for existing list `list` by `spaces` spaces, without changing list contents. `spaces` is allowed to be negative for list shrinking. The list can be shrunk sufficiently that some contents are lost. Returns 0 for success or 1 for unable to allocate memory.

`int ListExpandDD(listptrdd list,int addrows,int addcols)`
: Expands or shrinks memory allocated for new or existing list `list` by `addrows` rows and `addcols` columns without changing list contents. Inputs are allowed to be negative for list shrinking, possibly with loss of data. Returns 0 for success or 1 for inability to allocate memory.

`int ListExpandULVD4(listptrULVD4 list,int addrows);`

Expands or shrinks memory allocated for new or existing list `list` by `addrows` rows without changing list contents. Inputs are allowed to be negative for list shrinking, possibly with loss of data. Returns 0 for success or 1 for inability to allocate memory.

Memory management

`listptrli List_AllocLI(int max);`
Allocates a new empty list, set up for `max` spaces. `max` is allowed to equal 0. Returns the list or `NULL` if unable to allocate memory.

`listptrv List_AllocV(int max);`
Allocates a new empty list, set up for `max` spaces. `max` is allowed to equal 0. Returns the list or `NULL` if unable to allocate memory.

`listptrdd ListAllocDD(int maxrow,int maxcol)`
Allocates a new empty double-double list, set up for `maxrow` rows and `maxcol` columns, either or both of which are allowed to equal 0. Returns the list or `NULL` if unable to allocate memory.

`listptrdd ListAllocULVD4(int max)`
Allocates a new empty list, set up for `max` spaces. `max` is allowed to equal 0. Returns the list or `NULL` if unable to allocate memory.

`void List_FreeLI(listptrli list);`
Frees all memory allocated for list `list`. `list` is allowed to be `NULL`, in which case this does nothing.

`void List_FreeV(listptrv list);`
Frees all memory allocated for list `list`. `list` is allowed to be `NULL`, in which case this does nothing.

`void ListFreeDD(listptrdd list)`
Frees all memory allocated for list `list`. `list` is allowed to be `NULL`, in which case this does nothing.

`void ListFreeULVD4(listptrULVD4 list)`
Frees all memory allocated for list `list`. `list` is allowed to be `NULL`, in which case this does nothing.

Reading lists

`int List_MemberLI(const listptrli list,long int x);`
Tests to see if `x` is a member of list `list`, returning 1 if so and 0 if not. Does not assume list is sorted.

Adding elements to lists

`listptrli List_ReadStringLI(char *string);`
: Reads a string of elements of the given type (e.g. long integers for the LI suffix) from `string` and returns them in a newly created list. Returns the list for success or `NULL` for failure, where this failure could arise from a memory allocation error (unlikely) or a string reading error (likely).

`listptrv ListAppendItemV(listptrv list,void *newitem);`
: Appends item `newitem` to end of list `list`, returning `list` on success or `NULL` on failure to allocate memory. If `list` is entered as `NULL`, then a new list is created and is returned.

`listptrdd ListAppendItemsDDv(listptrdd list, int newrow, int narg, va_list items);`
`listptrdd ListAppendItemsDD(listptrdd list, int newrow, int narg, ...);`
: Appends one or more items to end of list `list`, returning `list` on success or `NULL` on failure to allocate memory. If `list` is entered as `NULL`, then a new list is created and is returned. Send in `newrow` as 1 to start a new row in the data table or to 0 to append to the end of the last row. Send in the number of items in `narg` and then that many items, each of which should be a double. This automatically allocates memory as needed. Also, the number of columns in the list is automatically expanded to be the longest row length. These two functions are essentially the same (and in fact the latter version simply calls the former version), except that the variable arguments are pre-grouped in the former version.

`int List_InsertItemULVD4(listptrULVD4 list,unsigned long long xdataul,void *xdatav,const double *xdatad4,int mode)`
: This tests to see if an element is in the list, which is assumed to be sorted in ascending order for the `dataul` component, and can insert it in the correct place if not. The new or test element has the three components `xdataul`, `xdatav`, and `xdatad4`. If `mode` is 0, then this simply looks for this test element, based solely on the `xdataul` value, and returns the index within the list of the element if it's found or -1 if it's not found. If `mode` is 1, then this looks for the test element, returns the index if it's found, and adds it and returns its index if it wasn't found; it can also return -2 for out of memory. If `mode` is 2, then this adds the new element, whether it was found or not, again returning its index or -2 for out of memory. Note that the `xdatad4` component is a const variable, so it gets copied over into the data structure.

Removing elements

`void List_CleanULVD4(listptrULVD4 list);`
: Scans through a list and looks for `void*` pointers, in the `datav` element, that are equal to `NULL`. These values are fully erased, and the list is compacted so that all the blank entries are at the end of the list, and the `list->n` value is then reduced to the number of used entries. The order of the list is not changed otherwise.

Combining lists

```
int List_AppendListLI(listptrli list,const listptrli newstuff);
```
Appends the contents of the list `newstuff` to the end of the existing list `list`, expanding `list` as needed. Returns 0 for success or 1 for memory allocation error.

```
int List_RemoveListLI(listptrli list,const listptrli remove);
```
Removes items that are in the list `remove` from the list `list`. For each item that is in `remove`, if it is in `list` in multiple copies, then only the last copy is removed. If an item that is in `remove` is not found in list, then this simply continues on to the next item. Returns the number of items that were removed from list `list`.

```
void ListClearDD(listptrdd list);
```
Clears the contents of the list but without freeing any memory. All this really does is to set the number of row and columns to 0 (along with the `nextcol` element).

List output

```
void ListPrintDD(listptrdd list);
```
Prints out all of the list data structure values to stdout. As written currently, this function is designed strictly for debugging, although it could be usefully repurposed for output. I'm not aware of any functions that call this function.