Thread-Safe-List Documentation

Thread-Safe-List is a library that allows to create a list with some practical methods which use thread-pooling.
The list has homogeneous values but can handle two types of value: long double and char[8].
A testing suite and a makefile to compile it are provided.

Made by: Elisabetta Russo [1986112] and Andrea Petruzzi [1982874]

Testing Suite

By default, the makefile compiles using gcc with the -g flag set
Usage:

```
$ make
$ ./test
```

Repository

https://github.com/GreyJolly/RDC-Thread-Safe-List.git

Data structures

```
// Basic node of the list
typedef struct baseNode
{
    struct baseNode *prev;      // Pointer to the next node in the list
    struct baseNode *next;      // Pointer to the previous node in the list
} baseNode;


// Specialized nodes containing the possible values for the list
typedef struct
{
    baseNode base;
    unsigned char value[TYPE_CHAR_LENGTH];
} charNode;

typedef struct
{
    baseNode base;
    long double value;
} ldoubleNode;


// The list
typedef struct
{
    unsigned char listType;     /* The type of nodes contained in the list, canzbe
                                TYPE_CHAR or TYPE_LONGDOUBLE */
    baseNode *head;             /* Pointer to the first element of the list */
    baseNode *tail;             /* Pointer to the last element of the list */
    baseNode *lastAccess;       /* Pointer to the last value of the list which has been
                                accessed */
    threadPool *pool;           /* Pointer to the thread pool that does computation for
                                the list functions */
    sem_t accessing;            /* Semaphore to allow for synchronization of accesses
                                to the list */
} list;
```

Functions

```
list *createList(unsigned char type);
void deleteList(list *l);
baseNode *getAt(list *l, unsigned int index);
baseNode *insert(list *l, void *value);
baseNode *insertAt(list *l, unsigned int index, void *value);
baseNode *removeFromList(list *l);
baseNode *removeFromListAt(list *l, unsigned int index);
list *map(list *l, void *(*function)(void *));
void *reduce(list *l, void*(*function)(void *, void*));
```

`list *createList(unsigned char type);`	
Description	Creates a list of a given type.
Parameters	**type** specifies the type of the values in the list. TYPE_CHAR for char[8] TYPE_LONGDOUBLE for long double.
Return Value	Returns the address to the new list, in case of error returns NULL and errno is set.

`void deleteList(list *l);`	
Description	Deletes a list
Parameters	**l** is the address of the list to be deleted, in case of error errno is set.

`baseNode *getAt(list *l, unsigned int index);`	
Description	Returns the address of the node of a list in a given index. **l->lastAccess** gets updated to point to the returned node.
Parameters	**l** is the address of the list where to search for the node **index** is the index of the node in the list
Return Value	Returns the address of the node or NULL if the list is too short, in case of error returns NULL and errno is set.

`baseNode *insert(list *l, void *value);`	
Description	Creates and inserts a new node as the first of the list **l->lastAccess** gets updated to point to the new node.
Parameters	**l** is the address of the list where to insert the node ***value** contains the value to be assigned to the new node: If the list is of type TYPE_CHAR it should be of type (char*) If the list is of type TYPE_LONGDOUBLE it should be of type (long double*)
Return Value	Returns the address of the new node, in case of error returns NULL and errno is set.

`baseNode *insertAt(list *l, unsigned int index, void *value);`	
Description	Creates and inserts a new node as in the list at a specific index **l->lastAccess** gets updated to point to the new node.
Parameters	**l** is the address of the list where to insert the node ***value** contains the value to be assigned to the new node: If the list is of type TYPE_CHAR it should be of type (char*) If the list is of type TYPE_LONGDOUBLE it should be of type (long double*) **index** is the index of the list the new node should be placed at
Return Value	Returns the address of the new node, in case of error returns NULL and errno is set.

`baseNode *removeFromList(list *l);`	
Description	Removes the first node from the list If **l->lastAccess** pointed to the node that was removed, it gets set to NULL.
Parameters	**l** is the address of the list where to remove the node
Return Value	Returns the address of the node following the removed node (NULL if there is none), case of error returns NULL and errno is set.

`baseNode *removeFromListAt(list *l, unsigned int index);`	
Description	Removes the node at a specific index from the list If **l->lastAccess** pointed to the node that was removed, it gets set to NULL.
Parameters	**l** is the address of the list where to remove the node **index** is the index of the node to remove
Return Value	Returns the address of the node following the removed node (NULL if there is none) or NULL if the list is too short, in case of error returns NULL and errno is set.

`list *map(list *l, void *(*function)(void *));`	
Description	Applies a function to every value in the list and creates a new list of the same type with the resulting values
Parameters	**l** is the address of the list whose nodes the function is to be applied **function** is the address of the function to be applied to the node values, its return value will be put in a new list of the same type: If the list is of type TYPE_CHAR it should be of type (char*)(*f)(char*) If the list is of type TYPE_LONGDOUBLE it should be of type (long double*)(*f)(long double*). Its result should be dynamically allocated.
Return Value	Returns the address of the new list containing the values resulting from the function applied to the values contained in the previous list, in case of error returns NULL and errno is set.

`void *reduce(list *l, void*(*function)(void *, void*));`

Description	Reduces the whole list to a single value by applying a function to every member of the list. Reduce works best with commutative functions.
Parameters	**l** is the address of the list whose nodes's values have to be reduced **function** is the address of the function used for the reduction, its return value will be put in a new list of the same type: If the list is of type TYPE_CHAR it should be of type (char*)(*f)(char*, char*) If the list is of type TYPE_LONGDOUBLE it should be of type (long double*)(*f)(long double*, long double*). Its result should be dynamically allocated.
Return Value	Returns the resulting cumulative return of the function, in case of error returns NULL and errno is set. If the list is of type TYPE_CHAR the return value will be of type char*. If the list is of type TYPE_LONGDOUBLE the return value will be of type long double*.