libADT

An abstract data type library written in C.

1 Lists

Lists are an abstract data type which are used to store a collection of (often) related objects.
They can be ordered or unordered and are an example of a more general ADT called a 'container. Functions for the list ADT include:

- Constructor - to instantiate a new list.
- Destructor - to remove an instance of a list from memory.
- Add - to add an item to the end of the list.
- Read - to read the value at a given index of the list into a variable.
- Remove - to remove an item in a list at a given index.
- Size - to return the number of items in the list.
- isEmpty - to return true if the list is empty, otherwise false.
- Search - to see if a given value exists within the list.
- Resize - resize the list as long as size is bigger.

ArrayList

Constructor(int size)

To instantiate a new ArrayList first define an List pointer.
Then instantiate it with the constructor method *listConst()*.
When calling the constructor pass the size of the ArrayList you want to make.

```
struct List* newList;
newList = listConst(5); //Creates list of size 5
```

Destructor(void)

To free a list from memory call the *listDest()* method and pass the list as an argument.

```
struct List* newList;
newList = listConst();

listDest(newList);
```

listAdd(struct List* list, int entity)

To add an item to the ArrayList use the *listAdd()* method and pass the list and item as arguments.

```
struct List* newList;
newList = listConst();

listAdd(newList, 3);
//ArrayList now reads [3] + 0's for empty initialised indexes.
```

listRead(struct List* list, int index, int* var)

To add an item to the ArrayList use the *listRead()* method and pass the list and item as arguments.

```
int var;
int newList;
listConst(newList);

listRead(newList, 3, &var);

printf("%d", var); //prints 3
```

listRem(struct List* list, int entity)

To remove an item from the ArrayList use the *listRem()* method and pass the list and index to remove item.

```
struct List* newList;
newList = listConst();

listAdd(newList, 3); newList = [3]
listRem(newList, 1); //Removes 3 from list.
```

Objects will cascade down ArrayLists when removed too.

```
struct List* newList;
newList = listConst();

listAdd(newList, 5);
listAdd(newList, 1);
listAdd(newList, 3);

//List reads [5,1,3] + '0's for empty initialised indexes.

listRem(newList, 1); //Removes '1' from index 1 of list.

//List now reads [5,3] + '0's for empty initialised index.
```

listPrint(struct List* list)

The *listPrint()* method prints all the values that have been added to the list in square brackets.

```
struct List* newList;
newList = listConst(5);

listAdd(newList, 3);
listAdd(newList,2);

listPrint(newList); //prints [3 2]
```

listPrintAll(struct List* list)

The *listPrintAll()* method prints the value contained at all indexes. Even ones not yet occupied.

```
struct List* newList;
newList = listConst(5);

listPrintAll(newList); //prints [0 0 0 0 0]

listAdd(newList, 2);

listPrint(newList); //prints [2]
listPrintAll(newList); //prints [2 0 0 0 0]
```

listSize(struct List* list)

To get the number of items currently stored in the ArrayList use the *listSize()* method.

```
struct List* newList;
newList = listConst();

for(int i=0; i<5; i++)
{
    listAdd(newList, i);
}

printf("%d", listSize(newList)); //prints 5
```

listIsEmpty(struct List* list)

The *listIsEmpty()* method returns 1 if the ArrayList is empty otherwise 0.

```
struct List* newList;
newList = listConst();

print("%d", listIsEmpty(newList)); //prints 1

listAdd(newList, 5); //newList = [5]

print("%d", listIsEmpty(newList)); //prints 0
```

listResize(struct List* list, int size)

Resizes an ArrayList.
If the size chosen is greater than the number of items in the ArrayList, the extra space is initialised with zeroes.

```
struct list* newList;
newList = listConst(5);

for(int i=0; i<5; i++)
{
    listAdd(l,i);
}

listPrintAll(newList); //prints [0 1 2 3 4]

listResize(10);
listPrintAll(newList); //prints [0 1 2 3 4 0 0 0 0 0]

listResize(7);
listPrintAll(newList) //prints [0 1 2 3 4 0 0]

listResize(lnewList, 1);
listPrintAll(newList) //prints [0 1 2 3 4] - list isn't truncated.
```

If the size chose is lower than the number of indices occupied, the list will be resized to preserve occupied spaces.
That is to say that the *listResize()* does not truncate ArrayLists.

listTruncate(struct List* list, int size)

The *listTruncate()* method operates in the same manner as the *listResize()* method except it wil truncate a list.

```
struct list* newList;
    newList = listConst(5);

    for(int i=0; i<5; i++)
    {
        listAdd(l,i);
    }

    listPrintAll(newList); //prints [0 1 2 3 4]

    listResize(10);
    listPrintAll(newList); //prints [0 1 2 3 4 0 0 0 0 0]

    listResize(7);
    listPrintAll(newList) //prints [0 1 2 3 4 0 0]

    listTruncate(newList, 1);
    listPrintAll(newList) //prints [0 ] - list has been truncated.
```

Single Linked List

Constructor()

A new SLL struct is instantiated via the *singleListConst()* method.

```
struct SingleList* list = singleListConst();
```

Destructor()

A SLL is freed from memory using the *singleListDest()* method/

```
struct SingleList* list = singleListConst();
singleListDest(list); // Frees list from memory
```

singleListAdd()

To append an item to the end of the SLL.

```
struct SingleList* list = singleListConst();
singleListAdd(list, 1); //List reads [1]
singleListAdd(list, 3); //List reads [1,3]
```

singleListAddFront()

To add an item to the front of the SLL.

```
struct SingleList* list = singleListConst();
singleListAdd(list, 1); //List reads [1]
singleListAdd(list, 3); //List reads [1,3]
singeListAddFront(list, 2) //List reads [2, 1, 3]
```

singleListRead()

To read an item at a given index into a variable.

```
int i;
struct SingleList* list = singleListConst();

singleListAdd(list, 3); //List reads [3]
singleListAddFront(list, 2); //List reads [2, 3]

singleListRead(list, 1, &i); //Value 3 is stored in variable i
```