CSE 3100 Systems Programming
Homework #3

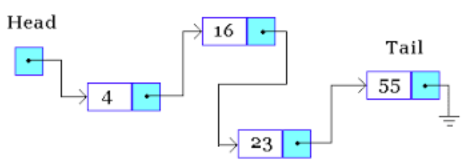
For instructions on how to work on the assignment on Mimir, please review materials in Lab 0. Do not forget to submit your work before the deadline.

1 Vector, List and Stack

In this part of the assigment, we will implement a linked list and a vector, and then use each of them in a stack implementation. We will assume that the vector, list and stacks all contain integers.

1.1 List

First, recall that a linked list looks like this:



Now, open the `list.h` file. You'll notice that we've defined an `ILNode` structure, which holds a number, and a pointer to the next node in the list. The `next` field of the tail node is `NULL`. The `IntList` holds the `head` node, `tail` node and length of the list.
You'll also notice that we've declared, but not implemented, several functions, in `list.h`. Now, open `list.c` with a text editor. This will provide the implementation for the functions declared in the `list.h` file. Your first job is to fill in the body of each incomplete function.

1.1.1 Init and Free

In the `initList` function, you are passed a pointer to an `IntList`. You just need to set the `head` and `tail` fields to NULL, and the `len` field to 0. For the `freeList`, you need to iterate through the list, and call `free` on each node.

1.1.2 Push and Pop

For the *push* functions, you're given a number. You need to create (`malloc`) a new node, store the number in the node, and put the node into the list. The `pushBackList` function puts the new node at the end of the list, while the `pushFrontList` function puts the node at the beginning of the list. For the *pop* functions, you need to remove the head node or tail node from the list, `free` it, and return the number contained in the removed node.

1.1.3 Sort

I've already provided the `sortList` function. However, it relies on the `sortedInsert` function, which takes a list that is already sorted, and a node, and inserts the node into the list so that the list stays sorted. You need to finish the `sortedInsert` function.

1.1.4 Testing List

Once you're done implementing the list functions, open the `listTest.c` file with a text editor. This is a program that uses your implementation to build a list and perform various manipulations on it. You can compile and run this program as follows:

```
$ cc -o listTest list.c listTest.c
$ ./listTest
()
Pushing 1 to back of list: (1 )
Pushing 2 to back of list: (1 2 )
Pushing 3 to front of list: (3 1 2 )
Pushing 8 to front of list: (8 3 1 2 )
Pushing 10 to back of list: (8 3 1 2 10 )
Pushing 4 to back of list: (8 3 1 2 10 4 )
Sorting list (1 2 3 4 8 10 )
Pop back 10: (1 2 3 4 8 )
Pop back 8: (1 2 3 4 )
Pop front 1: (2 3 4 )
Pop front 2: (3 4 )
```

1.2 Vector

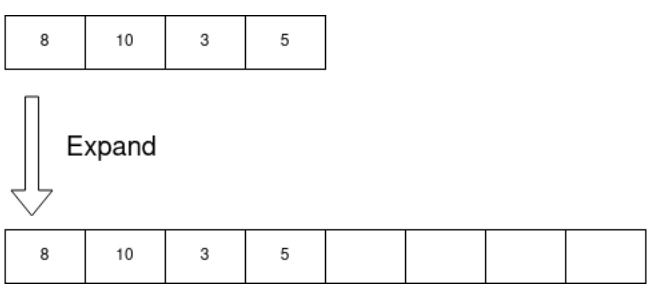
Here, we're implementing a data structure called a *vector* (also known as a dynamic array), which is basically an array that grows to accommodate additional elements. Open `vector.h` with a text editor. You'll notice an `IntVector` structure; it consists of an array of integers (`numbers`), a `size` and a `capacity`. The `size` holds the number of integers stored in the `numbers` array, while `capacity` holds the number of integers that *can* be stored in `numbers`. Now, open `vector.c`. There are a number of functions you need to implement:

1.2.1 Init and Free

In the `initVector` function, you are passed a pointer to an `IntVector`, and an integer `cap`, which is the initial capacity of the vector. Allocate space for `cap` integers, and fill in the fields of `vec`; `size` should be 0, and `capacity` should be set to `cap`. The `freeVector` function just needs to free the `numbers` array.

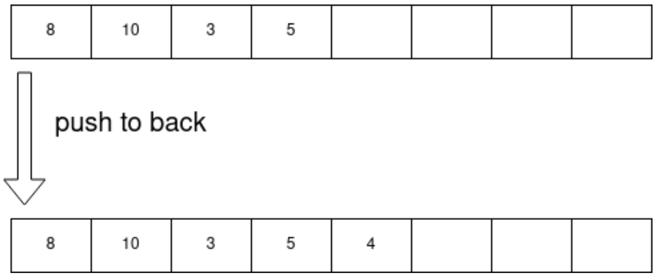
1.2.2 Expand Vector

The `expandVector` function is called when an element is added to a vector that is already at capacity. You need to call `realloc` to double the capacity of the vector:

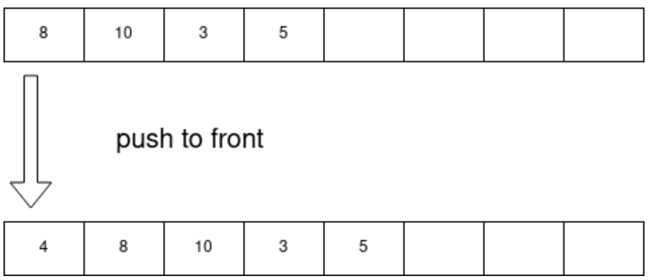


1.2.3 Push

For the *push* functions, you're given a `number` to push to the front or end of the vector. Start by checking if the vector is filled to capacity; that is, check if `size` is equal to `capacity`. If it is, then call the `expandVector` function. For `pushBackVector`, you set the last element of the array to `number`, and increment the `size` field:



For `pushFrontVector`, you need to shift the elements of the vector one place to the end, and put the `number` into the first position of the array:



1.2.4 Pop

For the `popBackVector` function, you need to decrement the `size` field, and return the last element in the array. For example, if the vector holds $\{4, 8, 10, 3, 5\}$, then you would decrement the size from 5 to 4, and return the last element, which is 5.

For the `popFrontVector` function, you need to return the first element in the vector, and shift the elements of the vector toward the front. For example, if the vector holds $\{4, 8, 10, 3, 5\}$, then you would return 4, and the vector would become $\{8, 10, 3, 5\}$. The `size` field would be decremented from 5 to 4. To be clear, you do *not* need to change the amount of memory allocated for the vector; that is, the vector grows as needed, but does *not* shrink.

1.2.5 Sort

In the `sortVector` function, sort the vector in ascending order. You should implement insertion sort. I've provided a `swap` function, which will be useful here.

1.2.6 Testing Vector

Once you're done implementing the vector functions, open the `vectorTest.c` file with a text editor. This is a program that uses your implementation to build a vector and perform various manipulations in it. You can compile and run this program as follows:

```
$ cc -o vectorTest vector.c vectorTest.c
$ ./vectorTest
()
Pushing 2 to back: (2 )
Pushing 1 to back: (2 1 )
Pushing 20 to front: (20 2 1 )
Pushing 25 to front: (25 20 2 1 )
Pushing 8 to back: (25 20 2 1 8 )
```

```
Pushing 3 to back: (25 20 2 1 8 3 )
Pushing 10 to back: (25 20 2 1 8 3 10 )
Sorting vector: (1 2 3 8 10 20 25 )
Pop front 1: (2 3 8 10 20 25 )
Pop back 25: (2 3 8 10 20 )
```

1.3 List Stack

Open `LStack.h` with a text editor. Notice that we defined a `Stack` struct, which contains a linked list (`IntList`). We've also declared functions to initialize and free, push to and pop from, and print the stack. Now, open `LStack.c`. You'll need to fill in each of the functions with a single line of code, each of which calls a function found in `list.c`. I've provided a test program called `LStackTest.c`, which accepts integers from standard input, and pushes them onto a stack, and then pops them off. You can compile it as follows:

```
$ cc -o LStackTest list.c LStack.c LStackTest.c
```

When you run the program, enter some numbers, seperated by spaces, and then press `enter` and `ctrl-d`. Here, I've entered `8 3 10 5 2`:

```
$ ./LStackTest
Enter some numbers: 8 3 10 5 2
Stack: (8 3 10 5 2 )
Pop: 2
Pop: 5
Pop: 10
Pop: 3
Pop: 8
Stack: ()
```

1.4 Vector Stack

Open `VStack.h` with a text editor. We've defined a `Stack` struct, which contains a vector (`IntVector`). We've also declared functions to initialize and free, push to and pop from, and print the stack. Now, open `VStack.c`. You'll need to fill in each of the functions with a single line of code, each of which calls a function found in `vector.c`. I've provided a test program called `VStackTest.c`, which accepts integers from standard input, and pushes them onto a stack, and then pops them off. You can compile it as follows:

```
$ cc -o VStackTest vector.c VStack.c VStackTest.c
```

When you run the program, enter some numbers, seperated by spaces, and then press `enter` and `ctrl-d`. Here, I've entered `8 3 10 5 2`:

```
$ ./VStackTest
Enter some numbers: 8 3 10 5 2
Stack: (8 3 10 5 2 )
Pop: 2
Pop: 5
Pop: 10
Pop: 3
Pop: 8
Stack: ()
```

1.5 Queue Using 2 Stacks

In this part of the assignment, we'll implement a queue using 2 (list based) stacks. A queue operates as FIFO: first in, first out, whereas a stack operates as LIFO: last in, first out. Enque refers to pushing an item into the queue, and deque refers to removing an item from the queue. To enque an item, push the item onto stack 1. To deque, check if stack 2 is empty. If it is empty, then transfer all of the items in stack 1 onto stack 2. Then, pop from stack 2, and return the popped item.
For example, suppose we did the following operations: enqueue(1), enqueue(2), deque(), enqueue(3), deque(), deque(). The three deque calls should return: 1, 2 and 3, in that order. Our two stacks would start off empty, and would change as follows:

```
Stack 1: [], Stack 2: []
enqueue(1)
Stack 1: [1], Stack 2: []
enqueue(2)
Stack 1: [1, 2], Stack 2: []
deque()
First, we transfer from stack 1 to stack 2, yielding: Stack 1: [], Stack 2: [2, 1]
Then, we pop from stack 2, yielding: Stack 1: [], Stack 2: [2]
enqueue(3)
Stack 1: [3], Stack 2: [2]
deque()
Stack 1: [3], Stack 2: []
deque()
First, we transfer from stack 1 to stack 2, yielding: Stack 1: [], Stack 2: [3]
Then, we pop from stack 2, yielding: Stack 1: [], Stack 2: []
```

You must implement the functions in `queue.c`. You can compile and run `QTest.c` as follows:

```
$ cc -o QTest list.c LStack.c queue.c QTest.c
$ ./QTest
Enter some numbers: 1 2 3 4
Dequeue: 1
Dequeue: 2
Dequeue: 3
Dequeue: 4
```

Don't forget to submit on Mimir!