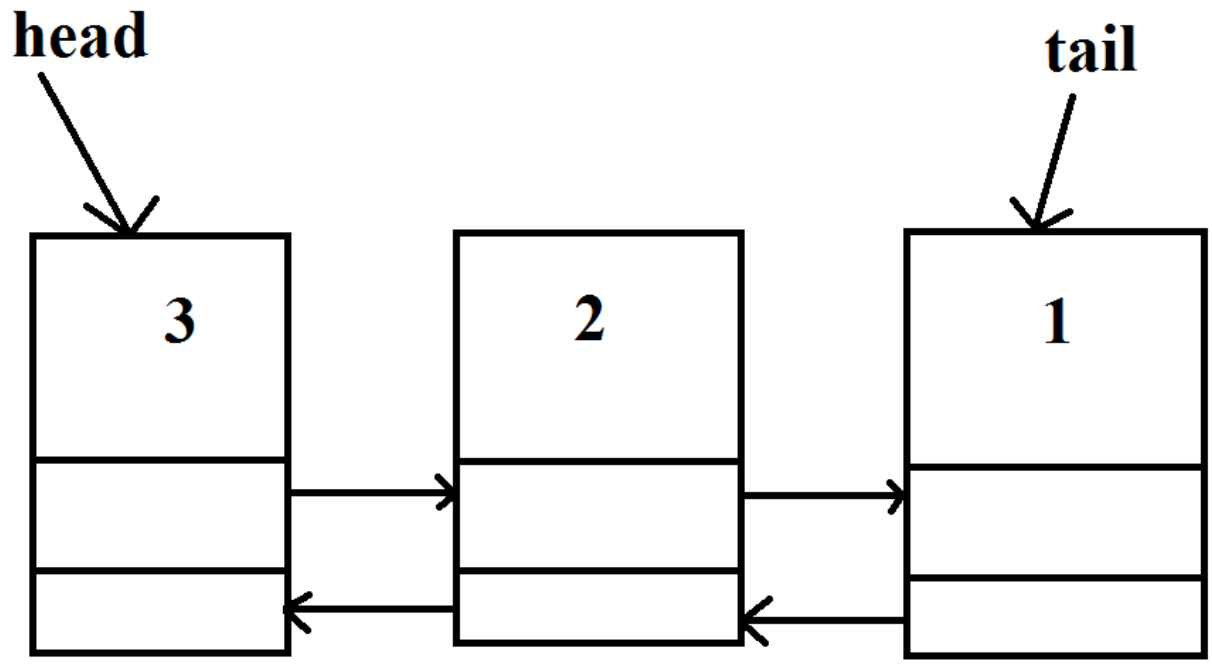
Lab 2: The BigInt

A typical programming question in interviews is how to extend the range of integers with a LinkedList. There is actually a way extend the range of real numbers also with LinkedList, but this lab will just work with Integers, as extending real numbers is more complex.

Requirements

Number Representation (Format)

Before talking about BigInt operations, let's talk about the representation we will use:



For this lab you shall implement minimal functionality in a DoubleLinkedList. A DoubleLinkedList is a list which nodes hold a reference to their next element, as well as their previous. The head's previous is and the tail's next are both equal to NULL. Each node will hold a digit of the integer, in the above example the number is 123, as we will store the smallest unit at the head of the tail. This will make easier all the arithmetic operations later.

Below an example of what you can use for the DoubleLinkedList nodes:

```
typedef unsigned int uint;

typedef unsigned long ulong;

typedef void* Object;

typedef struct _DNode{

        Object data;

        struct _DNode* prev;

        struct _DNode* next;

} DNode;
```

Then the DoubleLinkedList can look like this:

```
typedef struct _DoubleLinkedList{

        DNode* head;

        DNode* tail;

        uint length;

        uint elementSize;

} DoubleLinkedList;
```

Finally, a BigInt will only hold a DoubleLinkedList, but we shall create it as a separate structure as it will have operations that the DoubleLinkedList will not support.

```
typedef struct _BigInt{

        DoubleLinkedList* number;

} BigInt;
```

Supported Operations

DoubleLinkedList:

1. DoubleLinkedList* allocDList(uint elementSize): this function allocates the DoubleLinkList

2. void releaseDList(DoubleLinkedList* list): frees the space allocated for the DoubleLinkList

3. void insertDListElementAt(DoubleLinkedList* list, Object newElement, uint position): this function inserts an element at the position given by the parameter, the content at that position of the list is shifted to the end.

Example:

{4, 2, **5, 3, 4**} has 5 elements, therefore the positions go from 0 to 4,

calling insertDLIstElementAt(list, 7, 2) would affect the list as {4, 2, 7, **5, 3, 4**},

notice how the elements in red shifted their position by 1 towards the end.

4. void appendDList(DoubleLinkedList* list, Object newElement): inserts the new element at the end of the list. Notice that , time, since we have a DoubleLinkedList, there is a reference to the last element, so appending can be done faster

Example:

{4, 2, 7, 5, 3, 4}, calling appendDList(list, 9) affects the list as {4, 2, 7, 5, 3, 4, **9**}

5. void insertDList(DoubleLinkedList* list, Object newElement): this function works like insertDListElementAt(list, newElement, 0), meaning that will always insert at the head of the list

Example:

{4, 2, 7, 5, 3, 4, 9}, calling insertDList(list, 0) would affect the list as {**0**, 4, 2, 7, 5, 3, 4, 9}

6. DoubleLinkedList* reverseDList(DoubleLinkedList* list): this function reverses the content of the list, but returns the content in a new list.

Example:

{0, 4, 2, 7, 5, 3, 4, 9}, calling reverseDList would **return** {9, 4, 3, 5, 7, 2, 4, 0}

7. DoubleLinkedList* halfList(DoubleLinkedList* list): returns the elements that are after the midpoint of the list, in a new list. You could use the size of the list to know where the list midpoint is. However, a common programming interview is to this by looping only once over the list and not using a size variable. In class we shall see a solution for this problem.

Example:

{9, 4, 3, 5, 7, 2, 4, 0}, halfList(list) would **return** {7, 2, 4, 0}

8. Object removeDList(DoubleLinkedList* list int position): removes the element at the given position and **returns** the removed value.

   Example:

   {9, 4, 3, 5, 7, 2, 4, 0}, calling removeDList(list, 2) would keep the list as {9, 4, 5, 7, 2, 4, 0} and return a pointer to 3

9. void printDList(DoubleLinkedList* list): this function will print the list in the same format I have been using to printing the list in the lab. That is, {all list elements separated by a comma}

BigInt:

10. BigInt* allocBigInt(uint smallNumber): allocates the BigInt space, with smallNumber as its starting value
11. void releaseBigInt(BigInt* number): releases the memory that has been reserved for a BigInt
12. BigInt* addBigInt(BigInt* first, BigInt* second): adds to BigInts and returns their addition as a new BigInt. Look for tips at the end of the document.
13. BigInt* multiplyBigInt(BigInt* first, BigInt* second): adds to BigInts and returns their addition as a new BigInt. Look for tips at the end of the document.
14. BigInt* shifLeftBigInt(BigInt* number): this operation is equivalent to multiplying by 10.

Example:

123 would become 1230 as every content was shifted to the left and a 0 was added at the end of the value. From a list perspective this should be {3, 2 ,1} → {0, 3, 2, 1}

15. BigInt* shifRightBigInt(BigInt* number): this operation is equivalent to dividing by 10.

Example:

123 would become 012 and the 3 would be lost, but remember the first digit will never be 0, so keep it as 12. From a list perspective {3, 2, 1} → {2, 1}

16. int compareBigInt(BigInt* first, BigInt* second): this function will return -1 if first is smaller than second, 1 if first is larger than second, and 0 if they are both equal. Look for tips at the end.

17. bool isBigIntDividableBy(BigInt* first, uint digit): this function will test if a BigInt is dividable by the parameter digit. The only values for the function (and you can assume this as well) will be: 2, 3, 4, 5, 6, 8, 9. There are rules for testing dividable by 7 by they require subtraction which we are not building in the lab. Look for tips at the end.

18. uint toInt(BigInt* number): there will be times when a BigInt can fit on a uint variable, the library limits.h contains the MAX and MIN values for most data types actually. But since we are using uint we could find its largest value as pow(2, sizeof(uint)*8) – 1. As long as the BigInt is still smaller than the uint MAX's value then we will be able to store its value as an int. Notice that I added an unsigned long ulong typedef that you could use to calculate the largest possible uint in case you need to. If you like you could also do a ulong toLong(BigInt*), but the logic will be exactly the same as this function, so I leave it up to you if want to build it.

19. void printBigInt(BigInt* number): even though the DoubleLinkedList already has a printDList function, the output format is not suited for a BigInt. For this function the BigInt should be printed as an Integer, but keep in mind that BigInt could be much larger than an int or even long, so do not call the BigInt's function toInt or toLong and print that with printf. Instead, you can

print each individual digit (keep in mind the first digit is at the tail of the list and the last one is at the head)

Example:

If the internal list content is {3, 2, 1} then the output should be 123 without the {} or commas

Tips on the Arithmetic/Logic Functions

1. You would probably want to take the longest BigInt as the one you use as first number (do this inside of the function).
2. Use modulus as your friend in the operations. Remember that each node can only hold one digit of the BigInt. This means that 7 * 7 = 49 would be internally seen in the list as {9, 4}. In class we saw how the % and / operators could help us extract those digits. Keep in mind that dealing with digits, the worst case are:

   9 + 9 = 18 or {8, 1} and 9 * 9 = 81 or {1, 8} … in other words, you do not need to worry about 3 digits numbers.

3. Multiplication tip: in order to avoid adding many numbers at the end, which would require to have an array of BigInt (you are more than welcome to do this), a way you can multiply stuff is as follows:

1 *	2 4	3 5	Number of shifts
			123 x 5
		0	1230 x 4

What you are really doing is accumulating products, in other words 123 x 45 = 123 x 5 + 1230 x 4. So for any general case, you can start at 0, then multiply the original first number the first digit in the second number and add the result to the accumulator variable (have copies of the lists so that the operation at the end does not affect the original numbers). Then we add a 0 as the last digit the first number and we remove the last digit of the second, therefore 4 would be the last digit, and the first number should now be 1230. Repeat the process until the second number's copy is empty. This in computer architecture is known as Shift Add.

4. Compare tips:
   - A number with more digits than another is automatically larger
   - Assuming they both have the same length, comparing digits from left to right, as soon as two digits are not the same the largest digit will belong to the largest number
   - If all digits are the same then both numbers are equal
5. Dividedable tips:
   http://www.mathwarehouse.com/arithmetic/numbers/divisibility-rules-and-tests.php

Submission

You are required to write DoubleLinkedList.h, DoubleLinkedList.c, BigInt.h, and BigInt.c. You will be provided with a main.c for BigInt, but for now you can write your own main.c, just make sure that your code gives the proper answers for the main.c which will be provided along with the sample output. Once again, submit a screenshot of your running code after you test the main.c provided. Also include comments in your code along with a comment at the start of the file indicating who worked on the lab, along with the class, lab section, and purpose of each file.