ARITHMETIC OPERATIONS ON MULTIPLE BYTE INTEGERS

By: *Ishraga Mustafa Awad Allam*

A Thesis submitted in fulfillment of the requirements for the degree of Master of Science, in Computer Science,

Faculty of Mathematical Sciences,

University of Khartoum.

2009.

Acknowledgement

For Dr. Mohsen Hassan Abdalla Hashim(Dean of faculty of Mathematical sciences); with sincere gratitude and appreciation:

I am so incredibly fortunate to have the benefit from his invaluable guidance, support and continual advice, without which this work would not have seen the light.

I am greatly indebted to Prof. Sami Mohamed Sharif and Dr. Nuha Mudithir Behariy.

Also my gratitude is extended to my family with all my heart and love.

Abstract

Big integers are very essential in many applications. Cryptography is one of these applications. In this study, the objective is to create a multiple byte integer type, with its arithmetic operations defined. The operations are: addition, subtraction, multiplication, division and modular exponentiation are overloaded, to work on this multiple byte integer type.

The creation of the multiple byte integer is done by using doubly linked lists, a well known technique in data structure. The reason is that doubly linked lists enable us to create integer of unlimited size. That is, you do not have to pre-specify the size of the arrays storing these integers.

This is done by dynamically allocating the memory to store the digits constructing the integers.

The operations on these integers are defined using the simple and straight forward techniques, learnt in school.

The results obtained are satisfactory and reliable. The type could be extended to help define multiple byte floating point numbers.

In this work, an improvement has been made to the work of BH Flowers.

مستخلص

الأرقام الكبيرة مهمة لعدد كبير من التطبيقات وعلم التشفير هو واحد من تلك التطبيقات. الهدف من هذه الدراسة، تعريف نوع الرقم المتعدد البايت مع تعريف العمليات الرياضية على هذا النوع ايضاً. تلك العمليات هى: الجمع، الطرح، الضرب، القسمة وحساب الأسى بمقياس، تتم تحميلها للعمل على نوع الرقم المتعـــدد البايت.

إنشاء الرقم المتعدد البايت كان باستعمال القوائم مزدوجة الاتصال، يمكننا من إنشاء الرقم ذى السعة الغير محدودة، يعني ذلك، إنه لا يجب علينا تحديد سعة لتخزين مسبقاً. وهذه يتم فعله بتوزيع متحرك للذاكرة لتخزين الأحاديات المكونة للرقم.

العمليات على هذه الأرقام تعرف باستعمال التقنيات البسيطة والتى درست فى المدارس.

النتائج التى تم تحصيلها مرضية وموثوقة. هذا النوع من الاعداد يمكن استخدامه لتعريف انواع تساعد على تمثيل الاعداد الحقيقية.

من نتائج البحث، تحسين العمل الذى قام به فلورز.

Contents:

List of Tables

List of Figures:

Abbreviations:

IEEE: Institute of Electrical and Electronics Engineers.

Fortran: FORmula TRANslation Language.

AlGol: ALGOrithmic Language.

Basic: Beginners All Symbolic Instruction Code Language.

O(b) : Order of b.

Chapter One

Introduction

1. Introduction:

The computer consists of bits and bytes. The important thing in computer is its word size which can be data or some computer operation, represented by an integer for everything. The largest integers we have been able to work with have been confined to short, an int or a long. In C++ language, for example, these are one byte, two bytes and four bytes long, respectively. For all types, advanced arithmetical operations are made standard for only that domain of type, otherwise the compiler just ignore the function output, no overflow message. In principle, however, there is no reason to confine an integer to a specific number of bytes: the concept of a list allows us to work with any number of bytes, dynamically determined according to the transient needs of the program.

Integers limit in size made files vulnerable to attacks, even thought they were encrypted, because the size of the private key is limited. Also in real life, the size of amounts of large integers is limited to just more than 4 millions.

1.1. Research Objectives:

Is to create a type of integer having unlimited size. This is very essential in cryptographic application, and when extended to define floating point numbers can be useful in doing accurate computations.

1.2. The Methodology:

Double linked list are used to store the digits used to define the integers. The power of C++ is used to define functions to construct list and manipulate these integers. The simple and basic arithmetic techniques are overloaded on these integers.

1.3. The Contents of the Thesis:

The thesis contains foury chapters summarized as follows:

Chapter two is about integers and some operations on them for cryptography, and also giving information about double-linked lists. It was followed by giving a preview to all previous works done on the subject.

Chapter three is about multiple byte integers and how their arithmetic operations are done with proof for my assumptions. Also it gives information about floating point numbers, revealing the importance of the need to do precision floating point numbers.

Chapter four is the Conclusion and recommendation.

Chapter Two

Literature Review

2.1. Introduction:

This chapter gives a review to the previous work done on the subject of the thesis. Then it is followed by giving a definition of integers with some of them used in cryptography. This is followed by introducing lists; because the work is done on integers stored in doubly linked lists.

2.2. Previous Studies:

There are so many works that have been carried out in the area of integers, which will be discussed in details in the following sub sections.

2.2.1. BH Flowers's work:

Double linked list were used to store the digits, used to define the integers. The power of C++ is used to define functions to construct list. A decimalizing constructor is made that convert any stored list in decimal number. This number is returned as strings as the limit of integers were exceeded. There are also forward and backward constructors, traversing of these lists.

Future Work:

(i). Arithmetic operators on multiple byte integer.

(ii). Multiple byte float point numbers and they arithmetic operations.

Conclusion:

The sign character is not showing in the final result of the decimal string. The Boolean type BH Flowers made, did not work for me. If the first digit of the string is zero, the decimal will be zero. The error message function contains errors on the standard.h header file.

2.2.2. Chew Keong Tan's Work:

The implementation of asymmetrical cryptographic schemes often requires the use of numbers that are many times larger than the integer data types that are supported natively by the compiler. In this article, an introduction was given to the implementation of arithmetic operations involving large integers. No attempt was tried to give a full coverage of this topic since it is both complex and lengthy. For a more detailed treatment, the reader is referred to the listed references of his.

The source code that accompanies his article implements the BigInteger class supporting large integer arithmetic operations. Overloaded operators includes +, -, *, /, %, >>, <<, ==, !=, >, <, >=, <=, &, |, ^, ++, -- and ~. Other additional features such as modular exponential, modular inverse, pseudoprime generation and probabilistic primality testing are also supported.

Features

a. Arithmetic operations involving large signed integers in 2's complement representation.
b. Prime number tests using Fermat's Little Theorem, Rabin Miller's method and Solovay Strassen's method.
c. Modular exponential with Barrett reduction.
d. Modular inverse.
e. Random Pseudoprime generation.
f. Random Coprime generation.
g. Greatest common divisor.

Future Work

(i). Faster implementation of arithmetic operations.

(ii). More robust primality testing methods.

(iii). Faster pseudoprime generation

Conclusion

In his article, he has provided a short introduction to the topic of large integer arithmetic. Then he has looked at how large integer addition, subtraction and multiplication can be implemented. Also he examined the problem of primality testing and introduced the concept of primality testing based on Fermat's Little Theorem. His implementation of BigInteger class can be downloaded from his page and provides the overloading of most arithmetic operators. He has pointed out the limitations of his implementations of primality testing and is working towards more robust primality testing methods and faster implementation of arithmetic operators[4].

2.2.3. BigInteger of Java:

The BigInteger class in the java.math package provides a way of doing computations with arbitrary large integer types. Unlike C++, Java does not overload operators, so to do basic math, you have to call methods to perform mathematical operations such as addition. All of the operations create new BigInteger objects as results. The objects used in the operations remain untouched. For instance

BigInteger b1 = new BigInteger(“1”);

BigInteger b2 = new BigInteger(“2”);

BigInteger b3 = b1.add(b2);

At the end, b1 is still 1, b2 is still 2, and b3 is what you could expect it to be, 3. Modifying an object requires explicit code. For instance, if we wanted b1 to contain the result of the addition, we would use self-assignment like this:

b1 = b1.add(b2); // b1 is now 3

While the notation is different, the behavior is no different than using normal operators. Adding 1 + 2 doesn’t change the original values of 1 or 2.

All of the usual arithmetic operations are available: add, subtract, multiply, divide, and remainder. Another method, divideAndRemainder, returns a two-element BigInteger array where element 0 is the division result and element 1 is the remainder. A complete list and comparison of BigInteger operations is given in Table 2.1.

Table 2.1 BigInteger Bit Operations

BIT OPERATION	NATIVE TYPE NOTATION	BIGINTEGER NOTATION
AND	a & b	a.and(b)
ANDNOT	a & ~b	a.andNot(b)
NOT(complement)	~a	a.not(b)
OR	a | b	a.or(b)
XOR	a ^ b	a.xor(b)
Shift left n bits, signed	a << n	a.shiftLeft(n)
Shift right n bits, signed	a >> n	a.shiftRight(n)
Test bit n	(a & (1 << n) != 0)	a.testBit(n)
Set bit n	(a | (1 << n))	a.setBit(n)
Flip bit n	(a ^ (1 << n))	a.flipBit(n)

Likewise, all the usual bit operations are available, as described in Table 2.1. There is no need for a special unsigned right-shift operator, since BigInteger has unlimited capacity; the result of shifting n places to the right is the equivalent of dividing by 2^n, regardless of sign. In addition are a few new methods that simplify working with bits: testBit, setBit, and flipBit. Two others called bitCount and bitLength need a bit more explaining. For positive integers the results are what you could expect: bitLength returns the minimal number of bits required to represent the integer, and BitCount returns the number of one-bits in the representation. For negative numbers, bitLength gives a minimal length excluding the sign bit, and bitCount returns the number of zeros in the representation.

Creating and Converting:

BigInteger objects can be converted by using a string representation of a number, a native type, or with a byte array. Using a native type, a string, or a byte array representation of a number creates BigInteger objects.

Strings:

You can create BigInteger objects by using a construction that takes a string representation of a number. By default, the contructor expects the string representation to be in decimal (base 10) form. However, you can also use the standard base 16 representations, as follows:

BigInteger(String base10Rep)

BigInteger(String representation, int radix)

Conversely, you can retrieve a string representation in an appropriate base by using the toString method:

String toString()

String toString(int radix)

Numeric Types:

The BigInteger class can be created from a long by using the static method:

Long val = 123456789123;

BigInteger b1 = BigInteger.valueof(val);

Once you have the BigInteger object, it can be converted directly into a numeric type by using intValue, longValue, floatValue, or doubleValue. Not that we will mind, but there is no method to directly convert into a short or char value. If the value is too large to fit into the numeric type, a silent narrowing conversion is done (the high bits are chopped off).

Byte Arrays:

Generating byte arrays and constructing byte *a*. Assuming you have a BigInteger object; a byte array can be generated with:

Byte[] toByteArray()

And the results of this output can be used to create a new BigInteger object:

BigInteger(byte[] array)

However, because of signing issues, typically this format is not used for cryptographic purposes. If the bit length is a multiple of 8, and it always is for cryptographic applications, the byte array starts with an extra byte indicating the (0 for positive). Either your application can deal with a leading zero, or you can strip it away, as in the following:

Byte ba[] = bi.toByteArray();

```
If (ba[0] == 0) {
        Byte[] tmp = new byte(ba.length – 1);
        System.arraycopy(ba, 1, tmp, 0, tmp.length)
        ba = tmp;
}
```

A lower-speed option is to use strings as the common medium:

```
Byte ba[] = Hexify.decode(bi.toString(16));
```

Since BigInteger expects a sign-bit, you will need to use a special constructor and manually indicate the sign by passing in 1, -1, or 0 for positive, negative, or zero:

```
BigInteger(int signum, byte[] magnitude)
```

BigInteger and Cryptography:

BigInteger has a few special methods specifically designed for cryptography, listed in Table 2.2.

Table 2.2 BigInteger Operations Useful in implementing Cryptographic Algorithms

OPERATION	BIGINTEGER NOTATION
Create a random nonnegative integer Uniformly distributed from 0 to 2^n - 1	BigInteger (int numBits, Random r)_
Create a random integer that is prime With certainly 1 – 2^n	Random r = …; Int bitLength = …; BigInteger a = BigInteger(bitLength, r)
Check if prime with certainly using IEEE standard of 1 - 2^{100}	a.isProbablePrime()
a mod b	a.mod(b)
a^n mod b	a.modPow(b, n); // n is an int, not BigInteger
Find a^{-1}, such that aa^{-1} = 1 mod b	a.modInv(b)
Greatest common denominator of a, b	a.gcb(b)

Secret Methods in BigInteger:

For Sun Microsystems to effectively test for primality, many generic methods and algorithms had to be implemented; however, they are not part of the public API. For people working in numbers theory or who are developing more advanced cryptographic algorithms, these "hidden" methods may be useful. Unfortunately, they are declared private; but with a few modifications to the source code, you can create your own enhanced BigInteger variant.

The most interesting of these are the following methods:

```
int jacobiSymbol(int p, BigInteger n); // this is "package protected"
private boolean passesMillerRabin(int iterations)
private boolean passesLucasLehmer()
private static BigInteger lucasLehmerSequence(int z,
        BigInteger k, BigInteger n)
```

In addition, there is an implementation of sieve for testing primality of small numbers in the class BitSieve.
To "free" them:
1. Copy all the source files from this directory into a new directory.
2. Edit each file and change the package name from sun.math to one of your own choosing.
3. Make the BitSieve class "public."
4. Change the desired methods and classes to public. Note that the method jacobiSymbol in BigInteger and the class BitSieve do not have an access modifier, so you have to add public in front of the method.
5. Compile[2].

2.3. Integers:

Numeric information cannot efficiently be stored using the ASCII format. Imagine storing the number 123,769 using ASCII. This would consume 6 bytes, and it would be difficult to tell if the number was positive or negative (though we could precede it with the character + or -)[3].

The Scheme language has a notion of integer data type that is particularly convenient for the programmer: Scheme integers correspond directly to mathematical integers and there

are standard functions that correspond to the standard arithmetic operations on mathematical integers.

While convenient for the programmer, this causes headaches for those who have to implement the language. Computer hardware has no built-in data type that corresponds directly to the mathematical integers, and the language implementer must build such a type out of more primitive components.

Historically, therefore most “traditional” programming languages don’t provide full mathematical integers either, but instead give programmers something that corresponds to the hardware’s built-in data types. As a result, what passes for integer arithmetic in these languages is at least quite fast. What can be called “traditional” languages include FORTRAN, AlGol dialects, Pascal, C, C++, Java, Basic, and many others. Here, a discussion the integer types provided by C++, which is in many ways typical[9].

A more efficient way of storing numeric information is to use a different encoding scheme. The encoding scheme is commonly use is shown below,

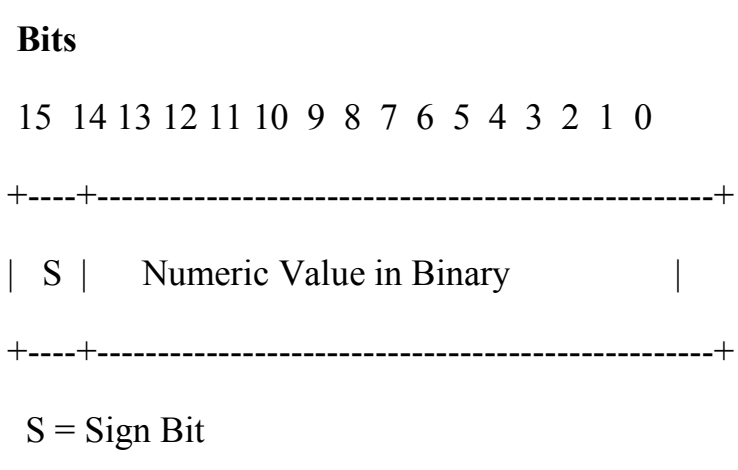


Figure 2.1. Sign Bit

Integers store whole numbers only! They do not contain fractional parts. Consider the examples below,

Valid	Invalid
123	.987
0	0.0
278903	123.09

The sign bit (which is bit 15) indicates whether the number is positive or negative. Logic 1 indicates negative, logic 0 indicates positive.

The number is converted to binary and stored in bits 0 to 14 of the two bytes.

Example

Store the value +263 as an integer value.

1) The sign bit is 0.

2) The decimal value +263 is 100000111 in binary.

Thus the storage in memory of the integer +263 looks like$_3$,

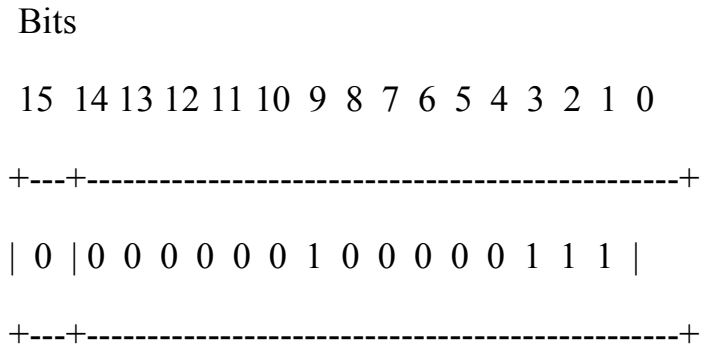


Figure 2.2. Storing Bits

2.3.1. Manipulating Bits:

One can look at a number as a bunch of bits, as shown in the last section. Java (like C and C++) provides operators for treating numbers as bits. The bitwise operators—&, |,

ˆ, and ˜— all operate by lining up their operands and then performing some operation on each bit or pair of corresponding bits, according to the following tables:

Table 2.3. Operand Bits (L, R)

Operation	(0, 0)	(0, 1)	(1, 0)	(1, 1)
& (and)	0	0	0	1
| (or)	0	1	1	1
ˆ (xor)	0	1	1	0

	Operand Bit	
Operation	0	1
˜ (not)	1	0

The “XOR” (exclusive or) operation also serves the purpose of a “not equal” operation: it is 1 if and only if its operands are not equal.

In addition, the operation x<<N produces the result of multiplying x by 2 (or shifting zeros in on the right). x>>>N produces the result of shifting zeros in on the left, throwing away bits on the right. Finally, x>>N shifts copies of the sign bit in on the left, throwing away bits on the right. This has the effect dividing by 2^N and rounding down (toward -∞).

For example,

```
int x = 42; //          == 0...0101010 base 2
int y = 7;  // == 0...0000111
```

The dots show that the integer can be of different size; i.e., eight bits or 16 bits.

x & y == 2 // == 0...0000010

The And operator of x and y.

x << 2 == 168 // == 0...10101000

The shifting operator used on x, shifting from the right two zeros.

x | y == 47 // == 0...0101111

The Or operator of x and y.

x >> 2 == 10 // == 0...00001010

Again the shifting operator on x, but this time shifting from the left.

x ^ y == 45 // == 0...0101101

The Xor operator of x and y.

~y << 2 == -32 // == 1...11100000

~y == -8 // == 11...111000 | ~y >> 2 == -2 // == 1...11111110

| ~y >>> 2 == 2^{30} - 2| // == 00111...1110

| (-y) >> 1 == -4

As it can be seen, even though these operators manipulate bits, whereas ints are supposed to be numbers, no conversions are necessary to "turn ints into bits." This isn't surprising, given that the internal representation of an int actually is a collection of bits, and always has been; these are operations that have been carried over from the world of machine-language programming into higher-level languages. They have numerous uses; some examples follow:

2.3.1.1. Masking:

One common use for the bitwise-and operator is to mask off (set to 0) selected bits in the representation of a number. For example, to zero out all but the least significant 4 bits of x, one can write

```
x = x & 0xf; // or x &= 0xf;
```

To turn off the sign bit (if x is an int):

```
x &= 0x7fffffff;
```

To turn off all but the sign bit:

```
x &= 0x80000000;
```

In general, if the expression x&˜N masks o_ exactly the bits that x&N does not.

If $n \geq 0$ is less than the word length of an integer type (32 for int, 64 for long), then the operation of masking off all but the least-significant n bits of a number of that type is (as it can be seen), the same as computing an equivalent number modulo 2^n that is in the range 0 to $2^n - 1$. One way to form the mask for this purpose is with an expression like this:

```
/** Mask to remove all but the N least-significant bits, 0<=N<32. */
int MASK = (1<<n) - 1;
```

[This can be worked with the same value of MASK, the statement

```
xt = x & ˜MASK;
```

The above has the interesting effect of truncating x to the next smaller multiple of 2^n; that is because while

```
xr = (x + ((1 << n) >>> 1)) & ~MASK;
// or
xr = (x + ((~MASK>1) & MASK)) & ˜MASK;
```

// or, if n > 0, just:

xr = (x + (1 << (n-1))) & ~MASK;

The above rounds x to the nearest multiple of 2^n.

2.3.1.2. Packing:

Sometimes one wants to save space by packing several small numbers into a single int. For example, it may be might known that w, x, and y are each between 0 and 2^9 - 1. They could be packed into a single int with

z = (w<<18) + (x<<9) + y;

Or

z = (w<<18) | (x<<9) | y;

From this z, it can be extracted w, x, and y with

w = z >>> 18; x = (z >>> 9) & 0x1ff; y = z & 0x1ff;

(In this case, the >> operator would work just as well.) The hexadecimal value 0x1ff(or 11111111_2 in binary) is used here as a mask; it suppresses (masks out) bits other than the nine that was on interest. Alternatively, you can extract x with

x = (z & 0x3fe00) >>> 9;

In order to change just one of the three values packed into z, it has to be essentially taking it apart and reconstruct it. So, to set the x part of z to 42, we could use the following assignment:

z = (z & ~0x3fe00) | (42 << 9);

The mask ~0x3fe00 is the complement of the mask that extracts the value of x; therefore z&~0x3fe00 extracts everything but x from z and leaves the x part 0. The right operand is simply the new value, 42, shifted over into the correct position for the x component (It

could have been written 378 instead, but the explicit shift is clearer and less prone to error, and compilers will generally do the computation for you so that it does not have to be re-computed when the program runs). Likewise, to add 1 to the value of x, if we know the result won't overflow 9 bits, we could perform the following assignment:

```
z = (z & ~0x3fe00) | ((((z & 0x3fe00) >>> 9) + 1) << 9);
```

Actually, in this particular case, it could have been just written

```
z += 1 << 9;
```

2.3.1.3. Packed Arrays:

We can extend the same idea to arrays. This is especially helpful when one must store an array of Boolean values; since there are only two possible Boolean values, one bit each ought to suffice. If one is forced instead to store each Boolean value in a byte, one wastes 87.5% of the storage space. Suppose that we wish to store a vector of 2^{20} one-bit values, b_0, …., $b_{1048575}$. We can store these in an array, B of 2^{17} bytes by storing the values b_{8k}; $b_{8k+1, \ldots}$, b_{8k+7} in byte B[k], where $0 \leq k < 2^{17}$. We represent b_{8k} by the units (least-significant) bit of B[k], and b_{8k+7} by the 128s (most-significant) bit. To retrieve the value bj, therefore, we can use the expression

```
(B[j/8] >> (j%8)) & 1
```

[Does it matter whether we use >> or >>> here? Why?] To set bj to a value x (0 or 1), we could use:

```
B[j/8] &= ~(1 << (j%8)); //      Mask off the bit containing bj
B[j/8] |= x << (j%8);    //      Shift x to the correct position
                         //      and place it in bj.
```

To be honest, the code in this section makes some demands on one's optimizing compiler. I assume, for example, that j%8 and j/8 are computed just once each, and that the

compiler uses the most efficient means to do so. Hard-core bit hackers will prefer to introduce temporary variables instead and to use j>>3 and j&7 rather than j/8 and j%8.

On some machines, it is probably better to work in units of 32-bit words instead, so that B is an array of 2^{15} int values. With this representation,

(B[j/32] >> (j%32)) & 1

Retrieves bj, and

B[j/32] &= ~(1 << (j%8));

B[j/32] |= x << (j%32);

Stores into it.

I'll leave to the reader the modifications needed to handle quantities containing more than One bit.

2.3.1.4. Flags:

A trick you will sometimes see in C and C++ programs is that of passing a bunch of boolean quantities in a single argument (where they are usually known as flags).For example, you might have some kind of formatting function that takes a number of yes/no options (is the argument hexadecimal, is it signed, is it left or right justified, etc.). If you define these flags as powers of two:

final static int HEX = 1, DEC = 2, OCT = 4, UNSIGNED = 8, ...;

...

/** Return a printable rendition of X, formatted according to

* FLAGS. */

String formatNumber(int x, int flags) ...

then the user can write

```
formatNumber(z, HEX | UNSIGNED);
```

to indicate that z is supposed to be formatted into an unsigned, hexadecimal number. Inside formatNumber, you can test for the presence of these flags with conditions like this[9]:

```
if ((flags & UNSIGNED) != 0) ...
```

2.4. Linked Lists:

A single integer can be very useful if we need a counter, a sum, or an index in a program, but generally we must also deal with data that have lots of parts, such as a list. We describe the logical properties of this collection of data as an abstract data type; we call the concrete implementation of the data a data structure. When a program's information is made up of component parts, we have to consider an appropriate data structure.

Data structures have a few features worth noting. First, they can be "decomposed" into their component elements. Second, the arrangement of the elements is a feature of the structure that affects how each element is accessed. Third, both the arrangement of the elements and the way they are accessed can be encapsulated[8].

First we will discuss linked lists in an Abstract manner; i.e., without using a computer language or a type of storage structure.

A linked list is a collection of elements (nodes), each containing:

1. Some data,
2. A pointer to the next element (node) of the list.

An external pointer to the first node is needed so that we can access the whole list.

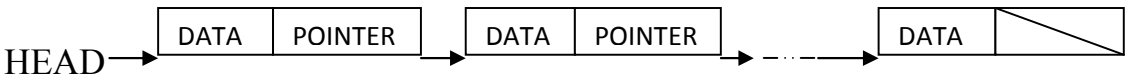
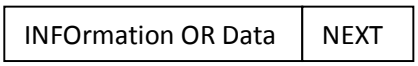


Figure 2.3. Linked List

2.4.1. Pointer:

A variable which stores the address (location) of another variable.



The next field of the last element (node) contains a special value (null) which is not a valid address.

Figure 2.4. Linked List Node

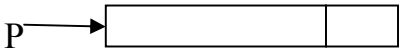
2.4.2. Notation:

Head: External pointer to 1st element.

P : A pointer, a variable

Node(p) : Refers to the whole element.

[node] pointed to by p.



Node(p)

Figure 2.5. Notation of a Linked List Node

INFO(p) : Data part of node(p) [or Data(p)].

Next(p) : Pointer field of node(p).

2.4.3. Doubly Linked Lists and Dynamic Storage Management:

So far we have mentioned singly linked linear lists but they got a restrictive problem. One difficulty with these lists is that if we are pointing to a specific node, says

P, then we can easily move only in the direction of the links. The only way to find the node which precedes P is to start back at the beginning of the list. The same problem arises when one wishes to delete an arbitrary node from a singly linked list. One must know the preceding node. If we have a problem where moving in either direction is often necessary, then it is useful to have doubly linked lists. Each node now has two link fields, one linking in the forward direction and one in the backward direction.

A node in a doubly linked list has at least 3 fields, say DATA, LLINK(left link) and RLINK(right link). A doubly linked list may or may not be circular. There is a special node added to the list, called a head node, pointing to the first node in the list and a tail node which is the RLINK pointing to of the last node in the list[6].

Chapter Three

Multiple Byte Integers and Floating Point Numbers

3.1. Introduction:

This chapter is about integers stored into doubly linked lists, multiple byte integers, and about their implementation details which will happen if they are to give right results with their mathematical proof. Also it explains my arithmetic operations done on multiple byte integers with their algorithms. Then it is followed by definition of float point numbers.

3.2. Multiple Byte Integers:

The largest integers we have been able to work with have been confined to an int or a long. In Turbo C++, for example, these are two 8-bit bytes and four 8-bit bytes long, respectively.

For both types, advanced arithmetical operations are standard provisions of the language, which makes their use overwhelmingly desirable wherever possible. In principle, however, there is no reason to confine an integer to a specific number of bytes: the concept of a list allows us to work with any number of bytes, dynamically determined according to the transient needs of the program.

The lists we have discussed so far have been singly linked: each node contains a single pointer to the succeeding node in the list. Multiple byte integers, however, are not efficiently dealt with by means of single linkages: we shall find it necessary to scan a list

both forwards and backwards, and this is most easily done if each node contains a pointer to the previous node as well as a pointer to the succeeding node. Morever, it is no longer desirable to work with pointers to the elements of a fixed data array: since we do not know in advance the number of bytes required by an integer, it would be most inconvenient to work with fixed arrays; we need to work with the bytes themselves.

An 8-bit byte, unsigned, can accommodate the integers 0, 1, …, 255. If we set B = 256, we may represent any non-negative integer N as a polynomial in B:

$$N = a_m B_m + \ldots + a_1 B + a_0, \quad \ldots\ldots 3.1.$$

Or $$N = (a_m, \ldots, a_1, a_0)B, \quad \ldots\ldots 3.2.$$

Where, in the second formula, we have recognized B as the radix, and the integer coefficients a_i, which must obey the restriction $0 < a_i < B$, as the digits of the integer to the base B. There may in principle be any number digits. If B were equal to 10, we should have the usual decimal representation, but we shall take B to be 256.

If we define a digit to be a node containing *a* byte b and two pointers, a multiple byte integer (which we shall call mult) may be represented by a list of digits. In diagrammatic form as shown in figure 3.1.[1]:

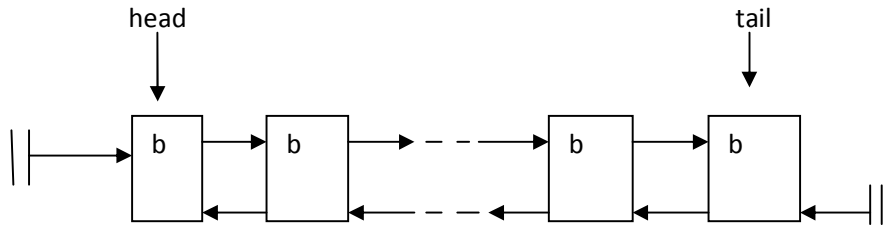


Figure 3.1

3.3. Computer Operations with Integers:

Before computers were invented mathematicians did computations either by hand or by using mechanical devices. Either way, they were only able to work with integers of limited size. Many number theoretic computations, such as factoring and primality testing, require computations with integers with as many as 50 or even 100 digits. In this section we will study some of the basic algorithms for doing computer arithmetic. Then we will study the number of basic computer operations required to carry out these algorithms.

It had been mentioned that computers internally represent numbers using bits, or binary digits. Computers have a built-in limit on the size of integers that can be used in machine arithmetic. This upper limit is called the word size, which we denote by w. The word size is usually a power of 2, such as 2^{35}, although sometimes the word size is a power of 10[5].

To do arithmetic with integers larger than the word size, it is necessary to devote more than one word to each integer. To store an integer $n > w$, we express n in base w notation, and for each digit of this expansion we use one computer word. For instance, if the word size is 2^{35}, using ten computer words we can store integers as large as $2^{350} - 1$, since integers less than 2^{350} have no more than ten digits in their base 2^{35} expansions. Also note that to find the base 2^{35} expansion of an integer, we need only group together blocks of 35 bits[5].

The first step in discussing computer arithmetic with large integers is to describe how the basic arithmetic operations are methodically performed.

We will describe the classical methods for performing the basic arithmetic operations with integers in base r notation where $r > 1$ is an integer. These methods are examples of algorithms (The word "algorithm" has an interesting history, as does the evolution of the concept of an algorithm. "Algorithm" is a corruption of the original term

"algorism" which originally comes from the book Kitab Al Jabr w'Al-Maqabala{Rules of Restoration and Reduction} written by Abu Ja'far Mohammed ibn Musa Al-Khowarizmi, in the ninth century. The word "algorism" originally referred only to the rules of performing arithmetic using Arabic numerals. The word "algorism" evolved into "algorithm" by the eighteenth century. With growing interest in computing machines, the concept of an algorithm became more general, to include all definite procedures for solving problems, not just the procedures for performing arithmetic with integers expressed in Arabic notation).

Definition: An algorithm is a specified set of rules for obtaining a desired result from a set of input[5].

We will describe for performing addition, subtraction, and multiplication of two n-digit integers $a = (a_{n-1}\, a_{n-2} \ldots a_1\, a_0)_r$ and $b = (b_{n-1}\, b_{n-2} \ldots b_1\, b_0)_r$. The algorithms described are used both for binary arithmetic with integers less than the word size of a computer, and for multiple precision arithmetic with integers larger than the word size w, using w as the base[5].

The implementation of asymmetrical cryptographic schemes often requires the use of numbers that are many times larger than the integer data types that are supported natively by the compiler. In this article, we give an introduction to the implementation of arithmetic operations involving large integers.

Implementation Details: The most common way of representing numbers is by using the positional notation system. Numbers are written using digits to represent multiples of powers of the specified base. The base that we are most familiar with and use every day, is base 10. When we write the number 12345 in base 10, it actually means

$$12345_{10} = 1x10^4 + 2x10^3 + 3x10^2 + 4x10^1 + 5x10^0 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 3.3.$$

Similarly, if the same number is specified in base 16, then

$$12345_{16} = 1x16^4 + 2x16^3 + 3x16^2 + 4x16^1 + 5x16^0 \text{……..3.4.}$$

Hence, it is important to know the base that the number is specified in, since the same representation could represent different values when interpreted in different bases. In general, a positive number can be written as a polynomial of order k, where k is non-negative and a_k != 0.

$$n = a_k b^k + a_{k-1} b^{k-1} + ... + a_1 b^1 + a_0 \text{…….……..3.5.}$$

b represents the base with 0 <= a <= b-1.

For base 10 representation, b = 10 and the digits allowed at each position is 0 <= a <= 9_4. But to get full advantage of these multiple byte integers, as the weight is set as byte, the best radix (b) is 256 and the digits allowed at each node is 0 <= a <= 255[1].

3.3.1. Addition of Multiple Byte Integers:

The addition of two numbers can be represented as the addition of two lists (polynomials) as shown below.

Let

$$n = a_k b^k + a_{k-1} b^{k-1} + ... + a_1 b^1 + a_0 \text{ ……....…3.6.}$$

$$m = c_k b^k + c_{k-1} b^{k-1} + ... + c_1 b^1 + c_0 \text{……..…..3.7.}$$

Then

$$n + m = (a_k + c_k) b^k + (a_{k-1} + c_{k-1}) b^{k-1} + ... + (a_1 + c_1) b^1 + (a_0 + c_0) \text{[1]…....3.8.}$$

They don't have to be equal in length; i.e., n has k while m has j, and k != j. But the radix is the same, for all arithmetical operations done.

However, it is often easier to visualize the addition of two large numbers as the digit-by-digit addition at each position. When we add the digits at a particular position, the largest resulting value is 2b – 2[4].

Proof: Since the maximum value of each digit is b - 1, we have

$$(b - 1) + (b - 1) = 2b - 2 \quad \text{.....3.9.}$$

Since 2b - 2 < 2b, the maximum value that we have to carry over to the next higher position is 1. For example, if we add 9 and 9 in base 10, we get 18, where 8 remains in the current position and the 1 gets *carried on* to the next higher position. When adding the two digits in the next position, we must remember to add the carry as well4.

We first discuss the algorithm for addition. When we add *a* and b, we obtain the sum

$$a + b = \sum_{j=0}^{n-1} a_j r^j + \sum_{j=0}^{n-1} b_j r^j = \sum_{j=0}^{n-1} (a_j + b_j) r^j, \quad \text{.....3.10.}$$

To find the base r expansion of $a + b$, first note that by the division algorithm, there are integers C_0 and s_0 such that

$$a_0 + b_0 = C_0 r + s_0, \; 0 \leq s_0 < r. \quad \text{.....3.11.}$$

Because a_0 and b_0 are positive integers not exceeding r, we know that $0 \leq a_0 + b_0 \leq 2r - 2$, so that $C_0 = 0$ or 1; here C_0 is the carry to the next place. Next, we find that there are integers C_1 and s_1 such that

$$a_1 + b_1 + C_0 = C_1 r + s_1, \; 0 \leq s_1 < r. \quad \text{.....3.12.}$$

Since $0 \leq a_1 + b_1 + C_0 \leq 2r - 1$, we know that $C_1 = 0$ or 1. Proceeding inductively, we find integers C_i and s_i for $1 \leq I \leq n - 1$ by

$$a_i + b_i + C_{i-1} = C_i r + s_i, \; 0 \leq s_i < r, \quad \text{.....3.13.}$$

with $C_i = 0$ or 1. Finally, we let $s_n = C_{n-1}$, since the sum of two integers with n digits has n + 1 digits when there is a carry in the n^{th} place. We conclude that the base r expansion for the sum is $a + b = (s_n s_{n-1} \ldots s_1 s_0)_r$.

When performing base r addition by hand, we can use the same familiar technique as is used in decimal addition[5].

The two number lists must be traversed from right to left; i.e., from the smallest weight, adding the two corresponding nodes and the carry digit, what is greater than the radix is then added to the next nodes sum. Every addition result is inserted into a node of the resultant number list. Finally if there is also a carry, a node is again inserted and attached at the front of the list representing the sum of the two number lists.

This is precisely what happens in the definition of the operator+().

```
mult& mult::add(mult ms1, mult ms2)
 {
        int a;              // The value of adding the two digits.
        int carry = 0;      // The carry of if the sum of the two digits is greater than the radix, initially is zero.
        mult ms3;           // The product list of addition.
          multscan thisptr(ms1, NEXT);  // Traverse from right to left, the first is the most smallest weight
          dgtp pr1;
          multscan thatptr(ms2, NEXT);
          dgtp pr2;
            while(((pr1 = thisptr()) != 0) && ((pr2 = thatptr()) != 0))        // While both are equal in length.
                {
                a = int(pr1 -> dgt) + int(pr2 -> dgt) + carry;
                if(((pr1 -> dgt) + (pr2 -> dgt) + carry) >= radix)
                        {
                                        carry = a / radix;
                                        a = a % radix;
                        }
                   else carry = 0;
                ms3.append(a);
```

```
                }
            if(ms1.numdgt > ms2.numdgt)// If I didn't do this loop, the next digit of the first list will be lost.
                {
                a = int(pr1 -> dgt) + carry;
                if(((pr1 -> dgt) + carry) >= radix)
                                {
                                        carry = a / radix;
                                        a = a % radix;
                                }
                        else carry = 0;
                ms3.append(a);
                }
          while((pr1 = thisptr()) != 0)         // While the first list is greater in length than the second list.
                {
                a = int(pr1 -> dgt) + carry;
                if(a >= radix)
                                {
                                        carry = a / radix;
                                        a = a % radix;
                                }
                        else carry = 0;
                ms3.append(a);
                }
         while((pr2 = thatptr()) != 0)          // While the second list is greater in length than the first list.
                {
```

```
            a = int(pr2 -> dgt) + carry;
            if(a >= radix)
                        {
                                carry = a / radix;
                                a = a % radix;
                        }
                    else carry = 0;
            ms3.append(a);
            }
 if(carry >= 1)  ms3.append(carry);  // If the last digit addition carry is not zero, an attached node.
                                                // Is appended to the product list.
 if((ms1.sgn == -1) && (ms2.sgn == +1))      // Checking if the two lists, doesn't have the same sign.
     {
            ms1.sgn = ms2.sgn = +1;  // Changing the signs first.
            ms3 = ms2 - ms1;          // A SWAP is done with a subtraction, but the sign will be negative.
            ms1.sgn = -1;              // The product will have a negative sign.
            cout << "  The Result value : ";
            printmult(ms3);
            exit(0);
   }
if((ms1.sgn == +1) && (ms2.sgn == -1))       // Checking if the two lists, doesn't have the same sign.
   {
            ms1.sgn = ms2.sgn = +1;          // Changing the signs first.
            ms3 = ms1 - ms2;          // A subtraction is done here.
            ms2.sgn = -1;              // The product will have a negative sign.
            cout << "  The Result value : ";
```

```
                printmult(ms3);
                exit(0);
            }
        cout << "\n The addition : \n  ";
        rs = ms3;    // I need two variables, because sometimes in an addition, I also perform a subtraction.
        printmult(ms3);      // One is local and the other is global to return the result.
        return rs;
    }
```

The inline operator does the inline + operator between two multiple-Byte integers, where m1 is the first number list, m2 is the second number list, pa is a temporary resultant number list.

```
inline mult operator + (mult m1, mult m2) {   return (pa.add(m1, m2)); }
```

The lists in number1 and number2 are traversed from right to left using list <Byte int> reverse iterators that are moved through the lists synchronously by using the increment operator. The carry digit and the blocks at each position of the iterator are added, the new carry digit and sum block are calculated, and the sum block inserted at the front of the list in sum using list's insert() operator.

Sign Test for Addition:

a. If the sign of the first number list is positive and the sign of the second number list is positive, a normal addition will be done with the resultant sign, will be positive.

b. If the sign of the first number list is negative and the sign of the second number list is positive, a subtraction operation is done using the subtraction operator,

after treating that both signs are positive and the first number list will be the second number list for the subtraction operation; i.e., reversing numbers.

c. If the sign of the first number list is positive and the sign of the second number list is negative, a subtraction operation is done after treating that both signs are positive.

d. The length of numbers can be tested here as addition is no good as subtraction for the subtraction has a fixed rule the state:

   (i). If first's sign is negative and second's sign is negative for subtraction, the resultant sign will be negative and the operation will be subtraction.

   (ii). If first's sign is negative and second's sign is positive for subtraction, the resultant sign will be negative and the operation will be addition.

   (iii). If first's sign is positive and second's sign is negative for subtraction, the resultant sign will be positive and the operation will be addition.

   (iv). If first's sign is positive and second's sign is positive for subtraction, the resultant operation will depend on the length of each of the number lists size. If the second length is greater, the resultant sign will be negative otherwise it will be positive.

3.3.2. Subtraction of Multiple Byte Integers:

The subtraction of two numbers is very similar to addition and can be easily implemented as byte-by-byte subtraction. However, when subtracting two digits, we often encounter the situation where the first digit is smaller than the second. In this case, we have to borrow 1 from the digit in the next position. This is actually done by subtracting 1 from the digit in the next higher position and adding the value of the base to the current digit[4].

We consider

$$a - b = \sum_{j=0}^{n-1} a_j r^j - \sum_{j=0}^{n-1} b_j r^j = \sum_{j=0}^{n-1} (a_j - b_j) r^j, \quad \text{.....3.14.}$$

Where we assume that $a > b$. Note that by the division algorithm, there are integers B_0 and d_0 such that

$$a_0 - b_0 = B_0 r + d_0, \; 0 \leq d_0 < r, \quad \text{.....3.15.}$$

and since a_0 and b_0 are positive integers less than r, we have

$$-(r - 1) \leq a_0 - b_0 \leq r - 1. \quad \text{.....3.16.}$$

When $a_0 - b_0 \geq 0$, we have $B_0 = 0$. Otherwise, when $a_0 - b_0 < 0$, we have $B_0 = -1$; B_0 is the borrow, from the next place of the base r expansion of a. We use the division algorithm again to find integers B_1 and d_1 such that

$$a_1 - b_1 + B_0 = B_1 r + d_1, \; 0 \leq d_1 < r. \quad \text{.....3.17.}$$

From this equation, we see that the borrow $B_1 = 0$ as long as $a_1 - b_1 + B_0 \geq 0$, and $B_1 = -1$ otherwise, since $-r \leq a_1 - b_1 + B_0 \leq r - 1$. We proceed inductively to find integers B_i and d_i, such that

$$a_i - b_i + B_{i-1} = B_i r + d_i, \; 0 \leq d_i \leq r \quad \text{.....3.18.}$$

with $B_i = 0$ or -1, for $1 \leq i \leq n - 1$. We see that B_{n-1}. We see that $B_{n-1} = 0$, since $a > b$. We can conclude that

$$a - b = (d_{n-1} \, d_{n-2} \ldots d_1 \, d_0)_r. \quad \text{.....3.19.}$$

When performing base r subtraction by hand, we use the same familiar technique as is used in decimal subtraction[5].

The two number lists must be traversed from right to left; i.e., from the smallest weight, subtracting the two corresponding nodes and the borrow digit if the result is negative, will be taken a radix in size from the exceeding first number digit; i.e., will be subtracted from the next subtraction of the other two corresponding digits.

```
mult& mult::sub(mult mt1, mult mt2)
```

```
{
 int borrow = 0;       // The borrow if the first list digit was smaller than the second list digit, initially is zero.
 int at;       // The value of subtracting the two digits.
     mult mt3, buf;              // The buf is needed when the second list length is greater than the first list.
        multscan thisptr(mt1, NEXT); // Traverse from right to left, the first is the most smallest weight
       dgtp pt1;
       multscan anotherptr(mt2, NEXT);
      dgtp pt2;
         while(((pt1 = thisptr()) != 0) && ((pt2 = anotherptr()) != 0))
              {
                        at = int(pt1 -> dgt) - int(pt2 -> dgt) - borrow;
                        if((pt1 -> dgt) < (pt2 -> dgt))
                        {
                                    at = at + radix;
                                    borrow = 1;
                          }
                       else borrow = 0;
                        mt3.append(at);
           }
       if(mt1.numdgt > mt2.numdgt)
              {
                        at = int(pt1 -> dgt) - borrow;
                        if(at == 0)
                                  {
                                    at = at + radix;
                                    borrow = 1;
```

```
                    }
                else borrow = 0;
               if(at <= 0) borrow = 1;
               mt3.append(at);
          }
 while((pt1 = thisptr()) != 0)          // While the first list is greater in length than the second list.
          {
               at = int(pt1 -> dgt) - borrow;
               if(at <= 0) borrow = 1;
               mt3.append(at);
          }
 mt3.sgn = +1;
 if((mt1.sgn == -1) && (mt2.sgn ==  -1))  mt3.sgn = -1;
if((mt1.sgn == -1) && (mt2.sgn == +1))        // Checking if the two lists, doesn't have the same sign.
          {
               mt1.sgn = mt2.sgn = +1;// Changing the signs first.
               mt3 = mt1 + mt2;
               mt3.sgn = -1;                    // The product will have a negative sign.
               mt3 = rs;
          }
if((mt1.sgn == +1) && (mt2.sgn == -1))        // Checking if the two lists, doesn't have the same sign.
          {
               mt1.sgn = mt2.sgn = +1;// Changing the signs first.
               mt3 = mt1 + mt2;                 // Addition is done here.
               mt3.sgn = +1;                    // The product will have a positive sign.
               mt3 = rs;
```

```
            }
    if(mt1.numdgt < mt2.numdgt)        // If the second list length is greater than the first list. SWAP
            {
                    mt3 = mt2 - mt1;
                    mt3.sgn = -1;                    // The product will have a negative sign.
                    mt3 = rs;
            }
        cout << "\n The subtraction : \n  ";
        rs = mt3;
        printmult (mt3);
        return rs;
}
```

Sign Test for Subtraction:

a. If the sign of the first number list is negative and the sign of the second number list is positive, an addition operation is done after treating both signs as positive. The resultant sign will be set to negative.

b. If both signs were negative, then the sign of the resultant is also negative.

c. If the sign of the first number list is positive and the sign of the second number list is negative, an addition operation is done after treating both signs as positive. The sign of the resultant is set to positive.

d. If the length of the first number list is less than that of the first, then a message will appear and the numbers will be swapped; i.e., the first will be the second for the subtraction operation. The resultant will have a negative sign.

3.3.3. Multiplication of Multiple Byte Integers:

The simplest way to implement multiplication is by using repeated addition. To compute *a* * b, we set the result to 0, then we repeatedly add *a* to the result for b number of times. Although this approach is simple, its complexity is O(b) large integer additions and is equivalent to O(bn) byte-by-byte additions, where n is the number of digits in the large integer. For large b, O(bn) > $O(n^2)$. Hence, this approach does not scale to large multipliers.

The next easiest approach is by using techniques that we have learnt in elementary schools. More precisely, we multiply *a* by the each digit of b and sum the partial results[4]. It is clear that the time complexity of multiplying two large integers is $O(n^2)$ byte multiplications where n is the number of digits in the number. The process of adding the partial results has a complexity of O(n) large integer additions, equivalent to $O(n^2)$ byte additions. Thus, the total complexity is $O(n^2)$ byte multiplications and $O(n^2)$ byte additions[4]. But before discussing multiplication, we describe shifting. To multiply $(a_{n-1} \ldots a_1 a_0)_r$ by r^m, we need only shift the expansion left m places, appending the expansion with m zero digits.

Example: To multiply $(101101)_2$ by 2^5, we shift the digits to the left five places and append the expansion with five zeros, obtaining $(10110100000)_2$. □

To deal with multiplication, we first discuss the multiplication of an n-place integer by a one-digit integer. To multiply $(a_{n-1} \ldots a_1 a_0)_r$, we first note that

$$a_0 b = q_0 r + p_0,\ 0 \le p_0 < r, \quad \ldots\ldots 3.20.$$

and $0 \le q_0 \le r - 2$, since $0 \le a_0 b \le (r-1)^2$. Next, we have

$$a_1 b + q_0 = q_1 r + p_1,\ 0 \le p_1 < r, \quad \ldots\ldots 3.21.$$

and $0 \leq q_1 \leq r - 1$. In general, we have

$a_i b + q_{i-1} = q_i r + p_i$, $0 \leq p_i < r$ ……...…..3.22.

and $0 \leq q_i \leq r - 1$. Furthermore, we have $p_n = q_{n-1}$. This yields $(a_{n-1} \ldots a_1 a_0)_r$

$(b)_r = (p_n p_{n-1} \ldots p_1 p_o)_r$. ……..……..3.23.

To perform a multiplication of two n-place integers we write

$$ab = a \sum_{j=0}^{n-1} b_j r^j = \sum_{j=0}^{n-1} (ab_j) r^j, \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 3.24.$$

For each j, we first multiply *a* by the digit b_1, then shift to the left j places, and finally add all of the n integers we have obtained to find the product.

We multiplying two integers with base r expansion, we use the familiar method of multiplying decimal integers by hand[5].

Again the two number lists were traversed from right to left, taking the first digit of the second number list and multiply it with each digit of the first number list. Each multiplication if greater than the radix, a carry digit will be added to the next proceeding nodes' result. A node will be created when the last digit of the first number list is reach and there was a carry, at the front of the list. The next multiplication of the next digit of the second number list will be done in the same way except that a zero digit will be inserted first at the most right node of the result. The next other ones, every time an increment is done; i.e., two zero's will be inserted in the most right nodes of the result, and so one, using an iterator. Every resultant of each multiplication of a digit of the second number list is added to the one before it, and the sum will be the final resultant of the multiplication operation. The addition operator will be used here, but without the sign test. Both signs will be treated as positive.

The Sign Test results only in the final resultant sign, and if the sign of the two numbers lists were different, then the resultant sign will be negative, otherwise it is positive.

```
mult& mult::mul(mult m1, mult m2)
 {
        int carrym;
       long unsigned int x, count = 0;
       long int ssum;
       mult mmp, mb; // mb to store each last addition of the multiplications done.
         multscan anotherptr(m2, NEXT);        // Traverse from right to left, the first is the most smallest weight
         dgtp p2;
           while((p2 = anotherptr()) != 0)        // For every digit of the second list, it multiple all the first list.
                 {
                 carrym = 0;
                 mult ma;          // Each time, a new initiated variable list is needed.
                   for(x = 0; x < count; x++)   ma.append(0);        // Shifting the first leading zeros.
                     multscan otherptr(m1, NEXT);
                    dgtp p1;
                   while((p1 = otherptr()) != 0)
                                 {
                                          ssum = int(p1 -> dgt) * int(p2 -> dgt) + carrym;
                                         if(ssum >= radix)
                                                 {
                                                          carrym = ssum / radix;
                                                          ssum = ssum % radix;
                                                 }
                                            else carrym = 0;
                                         ma.append(ssum);
```

```
                    }
                if(carrym >= 1) ma.append(carrym);      // Attaching the last carry to the product list.
                if(count == 0) mmp = ma;          // Initiale addition to the multiplication.
                  else                    // Else adding every last multiplications to the next one.
                {
                        mmp = ma + mb;
                        mmp = rs;
                }
                count++;                  // The shifting of the leading zeros is incremented.
                mb = mmp;                 // The value of storing additions, is updated.
                printmult(mmp);
            }
        mmp.sgn = +1;
       if((m1.sgn == -1) && (m2.sgn == +1))   mmp.sgn = -1;          // Only the sign will be negative.
        if((m1.sgn == +1) && (m2.sgn == -1))   mmp.sgn = -1;
        if((m1.sgn == -1) && (m2.sgn == -1))    mmp.sgn = +1;
         cout << "\n The multiplication : \n  ";
         rs = mmp;
         printmult(mmp);
         return rs;
}
```

3.3.4. Division of Integers:

We now discuss integer division. We wish to find the quotient q in the division algorithm

$a = bq + R,\ 0 \le R < b.$ ……………………………………………….…3.25.

If the base r expansion of q is $q = (q_{n-1}\, q_{n-2} \dots q_1\, q_0)_r$, then we have

$a = b \sum_{j=0}^{n-1} q_j r^j + R,\ 0 \le R < b.$ ………………………………………..3.26.

To determine the first digit q_{n-1} of q, notice that

$a - bq_{n-1} r^{n-1} = b \sum_{j=0}^{n-2} q_j r^j + R.$ ……………………………………3.27.

The right-hand side of this equation is not only positive, but also less than br^{n-1},

Since $\sum_{j=0}^{n-2} q_j r^j \le \sum_{j=0}^{n-2} (r-1) r^j = \sum_{j=0}^{n-1} r^j - \sum_{j=0}^{n-2} r^j = r^{n-1} - 1.$

Therefore, we know that

$0 \le a - bq_{n-1} r^{n-1} < br^{n-1}.$ …………………………………………..…3.28.

This tells us that

$q_{n-1} = [a / (br^{n-1})].$ …………………………………………………….……3.29.

We can obtain q_{n-1} by successively subtracting br^{n-1} from a until a negative result is obtained, and then q_{n-1} is one less than the number of subtractions.

To find the other digits of q, we define the sequence of partial remainders R_i by $R_0 = a$ and

$R_i = R_{i-1} - bq_{n-1} r^{n-i}$ …………………………………………………..…3.30.

For i = 1, 2, …, n. By mathematical induction, we show that

$R_i = \sum_{j=0}^{n-i-1} q_j r^j\ b + R.$ ………………………………………………..3.31.

For i = 0, this is clearly correct, since $R_0 = a = qb + R$. Now assume that

$R_k = \sum_{j=0}^{n-k-1} q_j r^j\ b + R.$ ……………………………………………….3.32.

Then

$R_{k+1} = R_k - bq_{n-k-1} r^{n-k-1}$ ……………………………………….…3.33.

$$= \sum_{j=0}^{n-k-1} q_j r^j \; b + R - b q_{n-k-1} r^{n-k-1} \quad \text{.....3.34.}$$

$$= \sum_{j=0}^{n-(k+1)-1} q_j r^j \; b + R. \quad \text{.....3.35.}$$

establishing *, which by it, we see that $0 \leq R_i < r^{n-i} b$, for i = 1, 2, …, n, since $\sum_{j=0}^{n-i-1} q_j r^j \leq r^{n-i} - 1$. Consequently, since $R_i = R_{i-1} - b q_{n-i} r^{n-i}$ and $0 \leq R_i < r^{n-1} b$, we see that the digit q_{n-i} is given by $[R_{i-1} / (br^{n-i})]$ and can be obtained by successively subtracting br^{n-i} from R_{i-1} until a negative result is obtained, and then q_{n-i} is one less than the number of subtractions. This is how we find the digits of q[5].

Again the two number lists must be traversed from right to left, subtracting the two corresponding digits. Here I'm not using the subtraction operator because I need a flag, which after each subtraction if there was a negative result, the flag is set to positive, indicating the end of the operation. The second number list is then subtracted from the resultant of the last subtraction until the flag is positive. The number of these subtractions while the flag is negative is the final resultant, been converted to a list itself; i.e., each time this iterator which count these subtraction is reach the radix size, a node of the final resultant is created and the modulo of the division of this iterator is inserted into a new node, creating the final division number list. The iterator was declared as long as to be able to present as large as possible for a really big integers as it can be for the resultant.

```
mult& mult::dvd(mult mr1, mult mr2)
{
    int borrow = 0, d2, d1;
    long unsigned int n = 0;
    int ar, ps = 0;
    mult mr3, mdval, buff = mr1;
    if(mr2.numdgt > buff.numdgt)   // If the second list length was greater than the first list.
    {
```

```
                fp = 1;   // This is a flag to show that the result will contain floating points.
                cout << "\n Float division not integers \n";
                mr3 = mdval = 0;          // The result will be zero.
                ps = 1;              // To terminate this operation, and not to go to next loop.
                n = 0;
        }
while(ps == 0)
        {               // Traverse from right to left, the first is the most smallest weight
                multscan thisptr(mr1, NEXT);
                  dgtp ptr1;
                  multscan anotherptr(mr2, NEXT);
                  dgtp ptr2;       // I didn't use the subtraction operator because I needed a flag.
                        while(((ptr1 = thisptr()) != 0) && ((ptr2 = anotherptr()) != 0))
                                {
                                        d2 = int(ptr2 -> dgt);
                                        d1 = int(ptr1 -> dgt);
                                        ar = d1 - d2 - borrow;
                                        if((ptr1 -> dgt) < (ptr2 -> dgt))
                                                {
                                                        ar = ar + radix;
                                                        if(ptr1 -> next == 0) ar = 0;
                                                        borrow = 1;
                                                }
                                           else borrow = 0;
                                        ptr1 -> dgt = ar;
                                        if(ar < 0) ps = 1; // The flag is on if the result is negative.
                                }
                        if(mr1.numdgt > mr2.numdgt)   // When the first list is greater than the second.
```

```
        {                       // Not to lost the next digit of the first list.
                ar = int(ptr1 -> dgt) - borrow;
                if((ptr1 -> next != 0) && (ar == 0))
                        {
                                ar = ar + radix;
                                 if(ptr1 -> next == 0) ar = 0;
                                borrow = 1;
                        }
                    else borrow = 0;
                if(ar <= 0) borrow = 1;
                ptr1 -> dgt = ar; // Change the first list digit.
                if(ar < 0) ps = 1;
        }
while((ptr1 = thisptr()) != 0)      // When the first list is greater than the second.
        {
                ar = int(ptr1 -> dgt) - borrow;
                ptr1 -> dgt = ar;
                borrow = 0;
        }
if(fp == 1)             // A suggested loop for Floating Point operation.
        {
            mult buf;
            multscan thisptr(mr1, NEXT);
            dgtp ptrf;
                buf.sgn = mr2.sgn;
                buf.append(0);          // Shifting the first list to increase it.
                while((ptrf = thisptr()) != 0) buf.append(int(ptrf -> dgt));
                mr1 = buf;
```

```
                    cout << "Fractional mult is ";
                    printmult(mr1);
                    //if(n == 65000) ps = 1;
                }
            n = n + 1;                  // Counting the number of subtractions.
            if((mr1.numdgt == mr2.numdgt) && (d2 > ar))  ps = 1;    // Floating Point.
            if((buff.numdgt == mr2.numdgt) && (d2 > d1))
                {
                    ps = 1;
                    n = 0;
                }
            if(ar <= 0) mr1.numdgt = mr1.numdgt - 1;        // If last digit is less than zero,
            if(mr1.numdgt == 0) ps = 1;                 // then the length of it decreases.
            if(mr1.numdgt < mr2.numdgt) ps = 1;             // Floating Point result...
    }
mdval = mr1;
mdval.sgn = +1;                     // The sign of the modulo is always positive.
pr = mdval;
mr1 = buff;
while(n > 0)        // Converting the decimal number of the counting into a multiple Byte List.
    {
            ar = n % radix;
            n = n / radix;
            mr3.append(ar);
    }
 mr3.sgn = +1;
 if((mr1.sgn == -1) && (mr2.sgn == +1)) mr3.sgn = -1;        // Only the sign changes.
```

```
        if((mr1.sgn == +1) && (mr2.sgn == -1)) mr3.sgn = -1;
        if((mr1.sgn == -1) && (mr2.sgn == -1)) mr3.sgn = +1;
        cout << "\n The module : \n  ";
        //if(fp == 1) mdval = 0;
        printmult (mdval);
        cout << "\n The dividor : \n  ";
        rs = mr3;
        if(fp == 1)   // A suggested loop for Floating Point operation.
            {
                cout << "  This is a fraction : 0.";
                for(n = 0; n < fc; n++) cout << "0";
            }
    printmult(mr3);
    if(fp == 1) cout << "  Sorry but this fraction can not be returned. This opens up a new research.
    \n";
    return rs;
}
```

A test is done first to see if the second number length of nodes is greater than the first. If it is so, a message will appear, telling that there will be a Floating-Point result, which is another research to be done. If both numbers' lengths are equal, traverse from left to right, here is the advantage or 'use' of the doubly-linked list structure reason of being there, to check if the second number is greater than the first; again there is the message of a Floating-Point research.

The Sign Test results only in the final resultant sign, and if the signs of the two numbers' lists were different, then the resultant sign will be negative, otherwise it is positive. Just like multiplication.

3.3.5. Exponential Operation:

Here the first number list is multiplied by itself, using the multiplication operator, according to the second number digits times. The second number is traverse from right to left, each digit is multiplied by the radix raised to its weight, and this result number control the number of iteration of the multiplication of the first number list. This number if it is greater that the maximum of integers for the mathematic function then there will be a problem as these functions were built on integers, log or pow, which raise the radix to its coefficient.

```
mult& mult::expo(mult me1, mult me2)
{       long unsigned int x, count = 0;
        mult me3;
        me3 = me1;
        multscan thatptr(me2, NEXT);  // Traverse from right to left, the first is the most smallest weight
        dgtp pe2;
        while((pe2 = thatptr()) != 0)
        {
                for(x = 0; x < ((int(pe2 -> dgt) * (exp((count) * (log(radix))))) - 1); x++)
                me3 = me3 * me1;
                count++;
        }
        if(me2.sgn == -1)
        {
                me3 = 0;
```

```
            cout << "This is a fractional, beyond this research \n";
        }
        cout << "\n The exponential : \n  ";
        rs = me3;
        printmult(me3);
        return rs;
    }
```

Warning: The second number digit if it is zero and length one, is trivially known that any number, raised to zero is one. No need to do it here.

The second number sign can not be negative as Floating-Point numbers are outside this scope of research.

3.3.6. Comparison Operation:

Here, the lengths of the two numbers are compared, if their signs are the same. The larger is the big one.

```
mult& mult::cmp(mult mc1, mult mc2)
 {
        int pr = 0, d2, d1;
        mult mc3;
          multscan (mc2, LAST); // Traverse from left to right, the first is the most greatest weight
          dgtp p2;                          // Here shows why it was a doubly-linked list
          multscan otherptr(mc1, LAST);
          dgtp p1;
                if((mc1.sgn == +1) && (mc2.sgn == -1))
                  {
```

```
                pr = ptrgt = 1;              // A flag so as not to perform the next loop.
                mc3 = mc1;       // The greatest will be the first list.
            }
        if((mc1.sgn == -1) && (mc2.sgn == +1)
          {
                pr = ptrlt = 1;     // A flag of the less than is set on.
                mc3 = mc2;       // The greatest will be the second list.
          }
if((mc1.numdgt < mc2.numdgt) && (mc1.sgn == mc2.sgn)
        {
                pr = ptrlt = 1;
                mc3 = mc2;
        }
    if((mc1.numdgt > mc2.numdgt) && (mc1.sgn == mc2.sgn))
        {
                pr = ptrgt = 1;
                mc3 = mc1;
        }
    if((mc1.numdgt == mc2.numdgt) && (mc1.sgn == mc2.sgn))
         {
                 while(((p1 = otherptr()) != 0) && ((p2 = anotherptr()) != 0))
                        {
                                d2 = int(p2 -> dgt);
                                d1 = int(p1 -> dgt);
                                cout << "D1 " << d1 << "  D2 " << d2 << "\n";
                                 if(d2 > d1)
```

```
                    {
                        pr = ptrlt = 1;
                        mc3 = mc2;
                        cout << "M2 > M1 \n";
                    }
                else if(d2 < d1)
                    {
                        pr = ptrgt = 1;
                        mc3 = mc1;
                        cout << "M1 > M2 \n";
                    }
            }
if(pr == 0)
    {
        cout << "They are Equal 'But' : ";
        if(mc1.sgn == mc2.sgn) mc3 = 0;
        ptreq = 1;                  // A flag of the equal than is set on.
    }
cout << "\n The Greater is : \n  ";
if(mc3.sgn == +1) ch = '+';
else ch = '-';
rs = mc3;
printmult(mc3);
return rs;
}
```

If one is negative, then it is trivially that the other number is the bigger whatever is the length is of its nodes.

If the lengths are equal, same sign, here the number lists are traversed from left to right, digits are compared, and the first one which is bigger than the other, then this one is the bigger. If there are all the same, then these numbers are equal, totally.

3.4. Floating Point Numbers:

There are two problems with integers; they cannot express fractions, and the range of the number is limited to the number of bits used. An efficient way of storing fractions is called the floating point method, which involves splitting the fraction into two parts, an exponent and a mantissa.

The exponent represents a value raised to the power of 2.

The mantissa represents a fractional value between 0 and 1.

Consider the number

12.50

The number is first converted into the format[3]

2n * 0.xxxxxx

Where **n** represents the exponent and 0.xxxxx is the mantissa.

The computer industry agreed upon a standard for the storage of floating point numbers. It is called the IEEE 754 standard, and uses 32 bits of memory (for single precision) called float in C family languages and real or real*4 in Fortran language: gives significant precision 24 bits and accuracy to approximations 7 decimal digits, or 64 bits (for double precision accuracy) called double in C family languages and double precision or real*8 in Fortran language: occupied 64 bits and has significant precision to 53 bits which gives accuracy to approximations 16 decimal digits. The single precision format looks like,

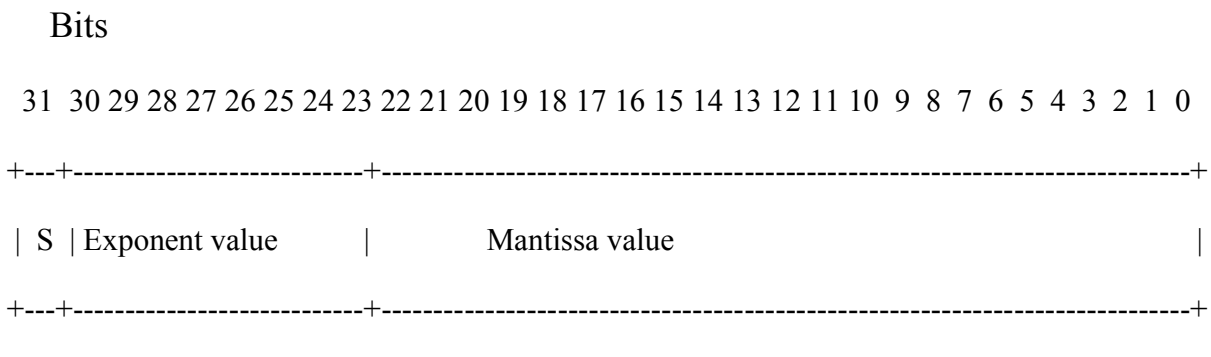


Figure 3.2. Floating Point Sign Bit

The sign bit is 1 for a negative mantissa, and 0 for a positive mantissa.

The exponent uses a bias of 127.

The mantissa is stored as a binary value using an encoding technique.

Working out the Floating Point bit patterns: The number we have are 12.5, which expressed as fraction to the power of 2 is,

12.5 / 2 = 6.25

6.25 / 2 = 3.125

3.125 / 2 = 1.5625

1.5625 / 2 = 0.78125

NOTE: Keep dividing by 2 till a fraction between 0 and 1 result. The fraction is the mantissa value; the number of divisions is the exponent value.

Thus our values now are,

0.78125 * (2*2*2*2) (comment: this is 2 to the power of 4)

The exponent bit pattern is stored using an excess of 127. This means that this value is added to the exponent when storing (and subtracted when removing).

The exponent bit pattern to store is,

4 + 127 = 131 = '10000011'

As the mantissa is a positive value, the sign bit is 0.

The mantissa is a little more complicated to work out. Each bit represents 2 to the power of a negative number. It looks like,

1st bit of mantissa = 0.5

2nd = 0.25

3rd = 0.125

4th = 0.0625

5th = 0.03125

etc

The mantissa number value we have is 0.78125, which in binary is

11001000000000000000000 (0.5 + 0.25 + 0.03125)

How-ever, to make matters even more complicated, the mantissa is normalized, by moving the bit patterns to the left (each shift subtracts one from the exponent value) till the first 1 drops off[3].

The resulting pattern is then stored.

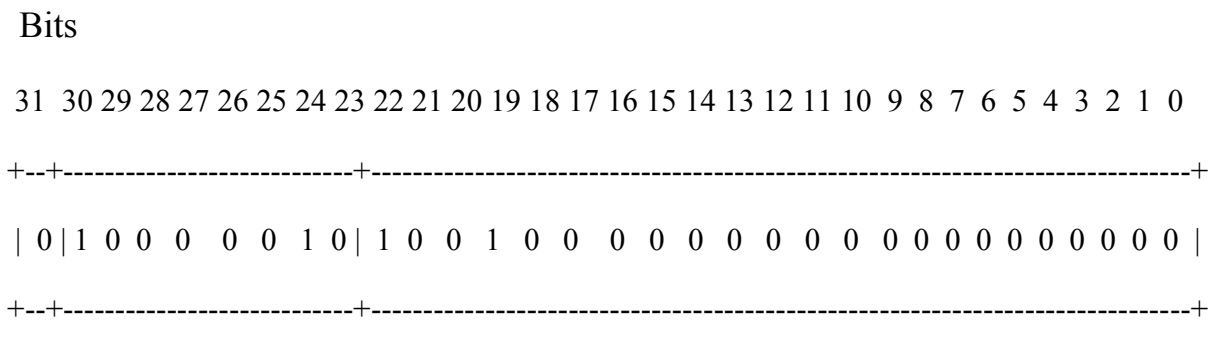
The mantissa now becomes

10010000000000000000000

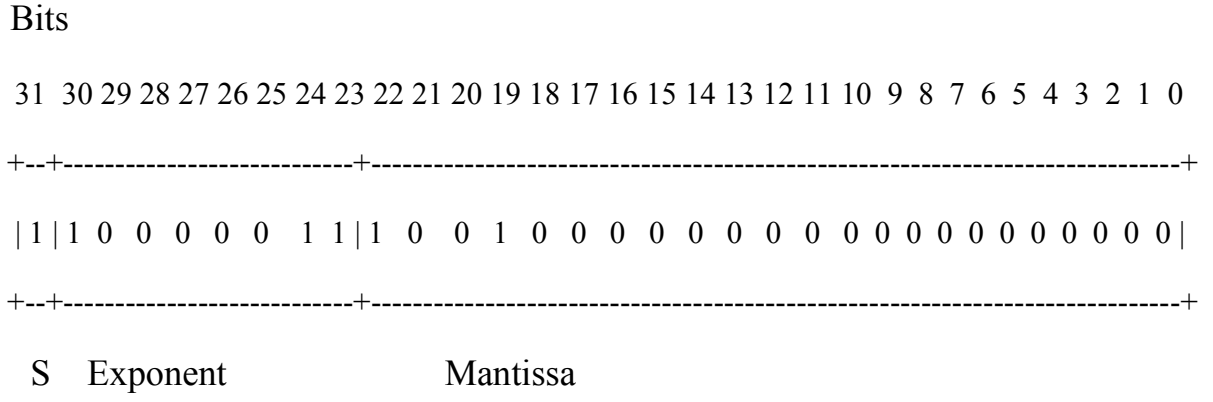
And the exponent is adjusted to become

131 - 1 = 130 = '10000010'

The final assembled format is,



Now let's convert the following storage format back into a decimal number.



This gives a negative number with an exponent value of,

$$131 - 127 = 4$$

And, a mantissa value of,

$$1.0 + 0.5 + 0.0625 = 1.5625$$

(The 1.0 comes from the bit which was shifted off when the mantissa was normalized), thus the number is,

$$-1.5625 * 2^4 = -25.0$$

The numeric range for floating point numbers using the IEEE-754 [3]method, using 32 bits, is

$$+- 0.838860808 * 2^{-128} \text{ to } +- 0.838860808 * 2^{127}$$

or, in decimal[3],

$$2.4652 * 10^{-39} \text{ to } 1.4272 * 10^{38}$$

3.4.1. The Advantages of Storing Floating Point Numbers in this Way:

a. Multiplication is performed by adding exponents and mantissas.
b. Division is performed by subtracting exponents and mantissas.
c. It is easy to compare two numbers to see which is the greater or lesser.
d. Large number ranges are stored using relatively few bits.

3.4.2. The Disadvantages of the Storage Format:

a. Errors are created by moving the mantissa bits.
b. Conversion backwards and forwards takes time.
c. There are fractions in the real world that can not be represented binary like 1/3.
d. There are also fractions so very small found in the real world applications approximated but they are important and cause big differences.
e. Sometimes the fraction occupies a small number of bits than the one set wasting space.
f. Approximations done for smallest of space causes inaccuracies that make fractions equal while in real thing they are not. That is why the comparisons operations are not accurate.

3.5. Results:

These are some examples done by my program:

3.5.1. Addition:

{+, 56, 78, 98, 245, 160} + {+, 45, 3, 68} = {+, 101, 81, 166, 245, 160}

In decimal = + 435161593248.

{+, 34, 120, 36, 86} + {-, 100, 200} = {+, 190, 175, 35, 86} ≡ + 3199148886.

{-, 34, 25, 78, 90, 200} + {+, 200, 220, 98} = {-, 90, 60, 235, 89, 200} ≡ - 387569113544.

{-, 120, 38, 98} + {-, 36, 200} = {+, 156, 238, 98} ≡ + 10284642.

3.5.2. Subtraction:

{+, 200, 39, 87, 65, 55} – {+, 48, 98, 120} = {+, 152, 197, 222, 64, 55} ≡ + 656154705975.

{-, 46, 24, 33, 78, 200} – {+, 47, 75, 98} = {-, 93, 99, 131, 78, 200} ≡ - 401101508296.

{+, 40, 39, 28, 36, 20} – {-, 200, 180} = {+, 240, 219, 28, 36, 20} ≡ + 1034468205588.

{-, 230, 48, 170, 78} – {-, 49, 160, 120, 35, 84} = {-, 75, 111, 206, 212, 83} ≡ - 323998372947.

3.5.3. Multiplication:

{+, 160, 147, 222} * {+, 250, 59} = {+, 64, 10, 98, 37, 52} ≡ + 275052111156.

{+, 20, 49, 120, 223} * {-, 98, 78, 57} = {-, 168, 225, 106, 24, 47, 6, 50}

≡ - 47535641875908746.

{-, 206, 187, 135} * {+, 24, 69, 47} = {-, 80, 33, 102, 203, 146, 51} ≡ - 88104388760115.

{-, 201, 167, 96, 58} * {-, 222, 189, 49} = {+, 78, 229, 41, 25, 204, 87, 11}

≡ + 22207012872673035.

3.5.4. Division & Modulus:

{+, 222, 190, 211} / {+, 208, 199} = {+, 16, 1} ≡ + 4097.

{+, 222, 190, 211} % {+, 208, 199} = {+, 221, 113, 0} ≡ + 14512384.

3.5.5. Exponential:

{+, 87, 75, 129, 89, 221} ^ {+, 4} = {+, 33, 96, 31, 76, 246, 132, 214, 67, 181, 98, 89, 82, 83, 145, 234, 146, 69, 220, 24, 143} ≡ + 1905402919448373960240517833737307453304505685 91.

3.5.6. Comparison:

{+, 39, 67, 128, 226} < {-, 87, 75, 129, 89, 221} = Greater {+, 39, 67, 128, 226} < False.

Chapter Four
Conclusion and Recommendation

The new data type described in this study combines the advantages of algorithm of double-linked lists and the precision is more than the decimal system. The concept of a list preserves space and memory space which the integer has to hold.

The sign test is taken from the way it is done in calculators. I noticed that the subtraction showed a rule that is fixed for all types of conditions, which can be used in addition also.

We created our own Boolean type using the same way as BH Flowers did.

I do recommend a further work to complete the definition of multi-precision type for floating point representation of real numbers. Also the basic operations should overload.

References

[**1**]. B.H. Flowers, **An Introduction to Numerical Methods in C++,** Oxford University Press, 1995.

[**2**]. **Bits and Bytes**, Sun Microsystems, at http://www.java.sun.com, accessed on 27 March 2007.

[**3**]. Brian Brown, **Data Structures and Number Systems,** 1984-2000, at http://goforit.unk.edu, accessed on 3 December 2007.

[**4**]. Chew Keong Tan, The Code Project – **C# BigInteger Class** – C# Programming, 2002, at http://www.codeproject.com, accessed on 6 August 2005.

[**5**]. Kenneth H. Rosen, **Elementary Number Theory and its Applications,** AT&T Bell laboratories and Kenneth H. Rosen, 1993.

[**6**]. Ellis Horowitz and Sartaj Sahri, **Fundamentals of Data Structures**, Computer Science Press, Inc., 1983.

[**7**]. Greg Goebel, **Computer Numbering Formats,** at

http://www.vertorsite.net, accessed on 12 June 2005.

[**8**]. Neil Dale, **C++ Plus Data Structures**, By Jones and Bartlett Publishers, inc., 1999.

[**9**]. Paul N. Hilfinger, **Numbers,** University of California, Department of Electrical Engineering and computer sciences, computer sciences division, C561B, Fall 1999.

[**10**]. Wikipedia, the free encyclopedia, **Computer numbering formats** – at

http://en.wikipedia.org, accessed on 3 May 2007.

Appendix

```
/* A MSc. RESEARCH, A C++ PROGRAM OF A NEW WAY OF DIGITALLY
STORING LARGE NUMBERS  'A DOUBLE LINKED LIST'  IN COMPUTERS,  AND
THE ARITHMETIC OPERATIONS ON THEM.
        Email : IshragaAllam@GMail.com   */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <iostream.h>
#include <string.h>
#include <math.h>

#define TRUE     1
#define FALSE    0
#define NEXT  TRUE
#define LAST FALSE

typedef unsigned char byte;

static int radix = 256;
const  int POSV  =  +1;
const  int NEGV  =  -1;
```

```
int fp = 0, ptreq = 0, ptrgt = 0, ptrlt = 0;
long unsigned int fc = 0;
char ch;

void error(char *s)
  {
    cout << s;
  }

class digit;
typedef digit *dgtp;

class digit
  {
    friend class mult;
    friend class multscan;
    private:
      byte dgt;
      dgtp next;
      dgtp last;
      digit(int );
```

```
  public:
    friend ostream& operator<<(ostream&, digit&);
    friend istream& operator>>(istream&, digit&);
};

digit::digit(int i)
 {
  if(i < 0 || i >= radix)
    error("Btyes only in digit::digit(int)");
  dgt = (byte)i;
  next = last = 0;
 };

ostream& operator<<(ostream &s, digit &d)
 {
  s << int(d.dgt);
  return s;
 }

istream& operator>>(istream &s, digit &d)
 {
  int i;
```

```
    s >> i;
    if(i < 0 || i >= radix)
      error("Enter a byte in op>>(os&, dig&)");
    d.dgt = (byte)i;
    return s;
  }

class mult
  {
    friend class multscan;
    private:
      int sgn;
      dgtp head;
      dgtp tail;
      int numdgt;
      void clearmult();
    public:
      mult() { sgn = POSV; head = tail = 0; numdgt = 0; }
      mult(int );
      //mult(mult& );
     // ~mult() { clearmult(); }
      void setsign(char );
```

```
    char getsign();
    void chsgn();
    void prepend(int );
    void append(int );
    int gethead();
    int gettail();
    int lastdec();
    mult& subdec(int );
    mult& divten();
    char *decimalize();
    mult& sub(mult , mult );
    mult& add(mult , mult );
    mult& mul(mult , mult );
    mult& dvd(mult , mult );
    mult& expo(mult , mult );
    mult& cmp(mult , mult );
  }rs, pa, ps, pm, pd, pr, pe;

inline mult operator + (mult m1, mult m2)
 {
  return (pa.add(m1, m2));
 }
```

```
inline mult operator - (mult m1, mult m2)
  {
    return (ps.sub(m1, m2));
  }
inline mult operator * (mult m1, mult m2)
  {
    return (pm.mul(m1, m2));
  }

inline mult operator / (mult m1, mult m2)
  {
    return (pd.dvd(m1, m2));
  }

inline mult operator % (mult m1, mult m2)
  {
    pd = pd.dvd(m1, m2);
    return pr;
  }

inline mult operator ^ (mult m1, mult m2)
  {
```

```
    return (pe.expo(m1, m2));
  }

inline mult operator < (mult m1, mult m2)
  {
    return (pa.cmp(m1, m2));
  }

 class multscan
  {
    private:
      mult *mp;
      dgtp dp;
      int dir;
    public:
      multscan(mult&, int );
      dgtp operator()();
  };

 multscan::multscan(mult &m, int direction)
  {
    mp = &m;
```

```
    dir = direction;
    dp = dir ? m.head : m.tail;
  }
 dgtp multscan::operator()()
  {
    dgtp p = dp;
      if(dir) dp = p ? dp -> next : 0;
       else dp = p ? dp -> last : 0;
      return p;
  }
/* void mult::clearmult()
  {
    dgtp dp = head;
      while(dp)
        {
          dgtp ddp = dp;
            if((dp = dp -> next) != 0) dp -> last = 0;
            delete ddp;
        }
      head = tail = 0;
      numdgt = 0;
  } */
```

```
mult::mult(int i)
  {
   if(i < 0 || i >= radix) error("Enter a byte!");
   sgn = POSV;
   head = tail = new digit(i);
   numdgt = 1;
  }
void mult::setsign(char ch)
  {
   if(ch == '+') sgn = POSV;
    else if(ch == '-') sgn = NEGV;
     else error("Enter '+' or '-' only!");
  }
char mult::getsign()
  {
   return (sgn == POSV) ? '+' : '-';
  }
void mult::chsgn()
  {
   sgn = (sgn == POSV) ? NEGV : POSV;
  }
void mult::prepend(int i)
```

```
  {
   dgtp p = new digit(i);
   if(!p) error("Allocation error in mult::prepend(int )");
   if(!head) head = tail = p;
    else
      {
        head -> last = p;
        p -> next = head;
        head = p;
      }
   ++numdgt;
  }
void mult::append(int i)
  {
   dgtp p = new digit(i);
   if(!p) error("Allocation error in mult::append(int )");
   if(!tail) head = tail = p;
    else
      {
        tail -> next = p;
        p -> last = tail;
        tail = p;
```

```
        }
    ++numdgt;
  }
int mult::gethead()
  {
    if(!head) error("Can't get from empty list in gethead()");
    dgtp p = head;
    int i = int(p -> dgt);
    if((head = head -> next) != 0) head -> last = 0;
    delete p;
    --numdgt;
    return i;
  }
/*mult::mult(mult &m)
  {
    head = tail = 0;
    sgn = m.sgn;
    numdgt = 0;
    while(m.head) append(m.gethead());
  }  */
void makemult(mult &m)
  {
```

```
    char ch;
          int n;
      cout << "  Enter sign : ";
      cin >> ch;
      m.setsign(ch);
      cout << "  Number of bytes : ";
      cin >> n;
      for(int i = 0; i < n; ++i)
        {
         int b;
         cout << "    Enter byte : ";
         cin >> b;
         m.append(b);
        }
  }
void printmult(mult &m)
  {
    multscan thisptr(m, NEXT);
    cout << "   {" << m.getsign();
    dgtp p;
    while((p = thisptr()) != 0) cout << *p << ",";
          cout << "\b}\n";
```

```
  }
int mult::lastdec()
  {
    int d = 0;
    int B = radix;
    int C = B % 10;
      multscan thisptr(*this, NEXT);
      dgtp p;
      while((p = thisptr()) != 0)
        d = (C * d + (p -> dgt % 10)) % 10;
      return d;
  }
mult& mult::subdec(int i)
  {
    int B = radix;
    int carry = FALSE;
      multscan thisptr(*this, LAST);
      dgtp p = thisptr();
      if(!p) error("Can't subtract from empty list! ");
            int j = int(p -> dgt) - i;
      if(j < 0)
        {
```

```
        j += B;
        carry = TRUE;
      }
    p -> dgt = (byte)j;
    while((p = thisptr()) != 0)
      {
        j = int(p -> dgt);
        if(carry)
          {
            --j;
            carry = FALSE;
          }
        if(j < 0)
          {
            j += B;
            carry = TRUE;
          }
                    p -> dgt = (byte)j;
      }
    if(!head -> dgt) gethead();
    return *this;
};
```

```
mult& mult::divten()
  {
    int B = radix;
    int c = 0;
      multscan thisptr(*this, NEXT);
      dgtp p;
      while((p = thisptr()) != 0)
        {
          int i = int(p -> dgt);
          i += (c * B);
          p -> dgt = (byte)(i / 10);
          c = i % 10;
        }
      if(!head -> dgt) gethead();
      return *this;
        }
char *chstr(char ch, char *str)
  {
    int n = strlen(str);
    char *newstr = new char[n + 2];
      newstr[0] = ch;
      newstr[1] = '\0';
```

```
    str = strcat(newstr, str);
    //newstr = 0;
    return str;
  }
char *mult::decimalize()
  {
   char *number = "";
    while(head)
      {
       int d = lastdec();
       char ch = char(d + '0');
         number = chstr(ch, number);
         subdec(d);
                        if(head) divten();
      }
    char s = (sgn == POSV) ? '+' : '-';
    return number;
      }

mult& mult::add(mult ms1, mult ms2)
  {
   int a;
```

```
int carry = 0;
mult ms3;
  multscan thisptr(ms1, NEXT);
  dgtp pr1;
  multscan thatptr(ms2, NEXT);
  dgtp pr2;
  while(((pr1 = thisptr()) != 0) && ((pr2 = thatptr()) != 0))
    {
     a = int(pr1 -> dgt) + int(pr2 -> dgt) + carry;
     if(((pr1 -> dgt) + (pr2 -> dgt) + carry) >= radix)
       {
         carry = a / radix;
         a = a % radix;
       }
      else carry = 0;
     ms3.append(a);
     //cout << ' ' << a << ' ';
    }
  if(ms1.numdgt > ms2.numdgt)
    {
     a = int(pr1 -> dgt) + carry;
     if(((pr1 -> dgt) + carry) >= radix)
```

```
                {
                        carry = a / radix;
                        a = a % radix;
                }
                else carry = 0;
    ms3.append(a);
    //cout << a << ' ';
  }
while((pr1 = thisptr()) != 0)
  {
    a = int(pr1 -> dgt) + carry;
    if(a >= radix)
                {
                        carry = a / radix;
                        a = a % radix;
                }
                else carry = 0;
    ms3.append(a);
    //cout << a << ' ';
  }
while((pr2 = thatptr()) != 0)
  {
```

```
        a = int(pr2 -> dgt) + carry;
        if(a >= radix)
                        {
                                carry = a / radix;
                                a = a % radix;
                        }
                        else carry = 0;
        ms3.append(a);
        //cout << a << ' ';
      }
   if(carry >= 1)
      {
        ms3.append(carry);
        //cout << carry << ' ';
      }
//if ((ms1.sgn == -1) && (ms2.sgn == -1)) ms3.sgn = -1;
 if((ms1.sgn == -1) && (ms2.sgn == +1))
     {
      cout << "First is negative \n";
      ms1.sgn = ms2.sgn = +1;
      ms3 = ms2 - ms1;
      ms1.sgn = -1;
```

```
    cout << "  The Result value : ";
    printmult(ms3);
    cout << "\n" << "In Decimal : " << ch << ms3.decimalize() << "\n";
    cout << "Press any key to finish : ";
    cin >> ch;
    exit(0);
   }
if((ms1.sgn == +1) && (ms2.sgn == -1))
  {
    cout << "Second is negative \n";
    ms1.sgn = ms2.sgn = +1;
    ms3 = ms1 - ms2;
    ms2.sgn = -1;
    cout << "  The Result value : ";
    printmult(ms3);
    cout << "\n" << "In Decimal : " << ch << ms3.decimalize() << "\n";
    cout << "Press any key to finish : ";
    cin >> ch;
    exit(0);
   }
 cout << "\n The addition : \n  ";
 rs = ms3;
```

```
    if(ms3.sgn == +1) ch = '+';
     else ch = '-';
    printmult(ms3);
    return rs;
  }

mult& mult::sub(mult mt1, mult mt2)
 {
  int borrow = 0;
  int at;
  mult mt3, buf;
    cout << "M1 numdgt " << mt1.numdgt;
    cout << "M2 numdgt " << mt2.numdgt << "\n";
    multscan thisptr(mt1, NEXT);
    dgtp pt1;
    multscan anotherptr(mt2, NEXT);
    dgtp pt2;

    while(((pt1 = thisptr()) != 0) && ((pt2 = anotherptr()) != 0))
      {
       at = int(pt1 -> dgt) - int(pt2 -> dgt) - borrow;
       if((pt1 -> dgt) < (pt2 -> dgt))
```

```
      {
        at = at + radix;
        borrow = 1;
      }
     else borrow = 0;
    mt3.append(at);
    //cout << at << ' ';
  }
if(mt1.numdgt > mt2.numdgt)
  {
    at = int(pt1 -> dgt) - borrow;
    if(at == 0)
      {
        at = at + radix;
        borrow = 1;
                                }
     else borrow = 0;
    if(at <= 0) borrow = 1;
    mt3.append(at);
    //cout << at << ' ';
  }
while((pt1 = thisptr()) != 0)
```

```
  {
   at = int(pt1 -> dgt) - borrow;
   if(at <= 0) borrow = 1;
   mt3.append(at);
   //cout << at << ' ';
  }

mt3.sgn = +1;
if((mt1.sgn == -1) && (mt2.sgn == -1)) mt3.sgn = -1;
if((mt1.sgn == -1) && (mt2.sgn == +1))
   {
      mt1.sgn = mt2.sgn = +1;
      mt3 = mt1 + mt2;
      mt3.sgn = -1;
      cout << "  The Result value : ";
      printmult(mt3);
      ch = '-';
      cout << "\n" << "In Decimal : " << ch << mt3.decimalize() << "\n";
      cout << "Press any key to finish : ";
      cin >> ch;
      exit(0);
      mt3 = rs;
```

```
    }
if((mt1.sgn == +1) && (mt2.sgn == -1))
   {
     mt1.sgn = mt2.sgn = +1;
     mt3 = mt1 + mt2;
     mt3.sgn = +1;
     ch = '+';
     cout << "  The Result value : ";
     printmult(mt3);
     cout << "\n" << "In Decimal : " << ch << mt3.decimalize() << "\n";
     cout << "Press any key to finish : ";
     cin >> ch;
     exit(0);
     mt3 = rs;
   }
if(mt1.numdgt < mt2.numdgt)
  {
   cout << "M1 is < than M2 so swap \n";
   mt3 = mt2 - mt1;
   mt3.sgn = -1;
   cout << "  The Result value : ";
   printmult(mt3);
```

```
        ch = '-';
        cout << "\n" << "In Decimal : " << ch << mt3.decimalize() << "\n";
        cout << "Press any key to finish : ";
        cin >> ch;
        exit(0);
        mt3 = rs;
       }
     cout << "\n The subtraction : \n  ";
     rs = mt3;
     if(mt3.sgn == +1) ch = '+';
      else ch = '-';
     printmult(mt3);
     return rs;
  }

mult& mult::mul(mult m1, mult m2)
  {
          int carrym;
    long unsigned int x, count = 0;
    long int ssum;
    mult mmp, mb;
```

```
multscan anotherptr(m2, NEXT);
dgtp p2;
while((p2 = anotherptr()) != 0)
  {
    carrym = 0;
    mult ma;
    for(x = 0; x < count; x++) ma.append(0);
    multscan otherptr(m1, NEXT);
    dgtp p1;
    while((p1 = otherptr()) != 0)
      {
        ssum = int(p1 -> dgt) * int(p2 -> dgt) + carrym;
        if(ssum >= radix)
          {
            carrym = ssum / radix;
           ssum = ssum % radix;
          }
         else carrym = 0;
        ma.append(ssum);
                        }
    if(carrym >= 1) ma.append(carrym);
                cout << "ma = ";
```

```
      printmult(ma);
      cout << "mb = ";
      printmult(mb);
      if(count == 0) mmp = ma;
       else
         {
            mmp = ma + mb;
            mmp = rs;
            cout << "In and in" << "\n";
         }
      count++;
      mb = mmp;
      cout << "mmp = ";
      printmult(mmp);
      printmult(mb);
    }
mmp.sgn = +1;
if((m1.sgn == -1) && (m2.sgn == +1)) mmp.sgn = -1;
if((m1.sgn == +1) && (m2.sgn == -1)) mmp.sgn = -1;
if((m1.sgn == -1) && (m2.sgn == -1)) mmp.sgn = +1;
cout << "\n The multiplication : \n  ";
rs = mmp;
```

```
   if(mmp.sgn == +1) ch = '+';
      else ch = '-';
   printmult(mmp);
   return rs;
       }

mult& mult::dvd(mult mr1, mult mr2)
        {
         int borrow = 0, d2, d1;
         long unsigned int n = 0;
         int ar, ps = 0;
         mult mr3, mdval, buff = mr1;
   cout << "M1 : ";
   printmult(buff);
   if(mr2.numdgt > buff.numdgt)
     {
         fp = 1;
         /*mult buf;
         multscan thatptr(buff, NEXT);
         dgtp ptrf;
          buf.sgn = buff.sgn;
          buf.append(0);
```

```
  while((ptrf = thatptr()) != 0) buf.append(int(ptrf -> dgt));
  fc = fc + 1;
  mr1 = buf;
  cout << "Buff is ";
  printmult(mr1); */
  cout << "\n Float division not integers \n";
  /*mr3 = mr1 / mr2;
  cout << "  The Result value : ";
  printmult(mr3);
  cout << "\n" << "In Decimal : " << ch << mr3.decimalize() << "\n";
  cout << "Press any key to finish : ";
  cin >> ch;
  exit(0); */
  mr3 = mdval = 0;
  ps = 1;
 }
        while(ps == 0)
{
 multscan thisptr(mr1, NEXT);
 dgtp ptr1;
 multscan anotherptr(mr2, NEXT);
  dgtp ptr2;
```

```
while(((ptr1 = thisptr()) != 0) && ((ptr2 = anotherptr()) != 0))
  {
    d2 = int(ptr2 -> dgt);
    d1 = int(ptr1 -> dgt);
    ar = d1 - d2 - borrow;
    if((ptr1 -> dgt) < (ptr2 -> dgt))
      {
        ar = ar + radix;
        if(ptr1 -> next == 0) ar = 0;
        borrow = 1;
      }
     else borrow = 0;
    cout << ar << ' ';
    ptr1 -> dgt = ar;
    if(ar < 0) ps = 1;
  }


if(mr1.numdgt > mr2.numdgt)
  {
    ar = int(ptr1 -> dgt) - borrow;
     if((ptr1 -> next != 0) && (ar == 0))
              {
```

```
                    ar = ar + radix;
                    if(ptr1 -> next == 0) ar = 0;
                    borrow = 1;
                }
          else borrow = 0;
        if(ar <= 0) borrow = 1;
        ptr1 -> dgt = ar;
        cout << ar << ' ';
        if(ar < 0) ps = 1;
      }
    while((ptr1 = thisptr()) != 0)
      {
        ar = int(ptr1 -> dgt) - borrow;
        ptr1 -> dgt = ar;
       cout << ar << ' ';
        borrow = 0;
        //if(ar < 0) ps = 1;
      }
if(fp == 1)
  {
    mult buf;
    multscan thisptr(mr1, NEXT);
```

```
    dgtp ptrf;
    buf.sgn = mr2.sgn;
    buf.append(0);
     while((ptrf = thisptr()) != 0) buf.append(int(ptrf -> dgt));
     mr1 = buf;
     cout << "Fractional mult is ";
     printmult(mr1);
     if(n == 65000) ps = 1;
   }
if(n % 16 == 0)
  {
    cout << "Press any key to continue : ";
    getch();
  }
n = n + 1;
cout << "   No. : " << n << "  Numdgt = " << mr1.numdgt << "\n";
if((mr1.numdgt == mr2.numdgt) && (d2 > ar))
  {
    cout << "Equal and is less \n";
    ps = 1;
  }
if((buff.numdgt == mr2.numdgt) && (d2 > d1))
```

```
  {
    cout << "Equal but M2 is greater than M1 \n";
    ps = 1;
    n = 0;
  }
 if(ar <= 0) mr1.numdgt = mr1.numdgt - 1;
 if(mr1.numdgt == 0) ps = 1;
 if(mr1.numdgt < mr2.numdgt) ps = 1;
 }
mdval = mr1;
mdval.sgn = +1;
pr = mdval;
mr1 = buff;
while(n > 0)
  {
   ar = n % radix;
   n = n / radix;
   mr3.append(ar);
   cout << ar << ' ';
  }
 mr3.sgn = +1;
                    if((mr1.sgn == -1) && (mr2.sgn == +1)) mr3.sgn = -1;
```

```
                if((mr1.sgn == +1) && (mr2.sgn == -1)) mr3.sgn = -1;
                if((mr1.sgn == -1) && (mr2.sgn == -1)) mr3.sgn = +1;

    cout << "\n The module : \n  ";
    //if(fp == 1) mdval = 0;
    printmult(mdval);
    cout << "\n The dividor : \n  ";
    rs = mr3;
    if(fp == 1)
      {
       cout << "  This is a fraction : 0.";
       for(n = 0; n < fc; n++) cout << "0";
      }
    if(mr3.sgn == +1) ch = '+';
     else ch = '-';
    printmult(mr3);
    if(fp == 1) cout << "  Sorry but this fraction can not be returned. This opens up a
new research. \n";
    return rs;
  }

mult& mult::expo(mult me1, mult me2)
```

```
{
long unsigned int x, count = 0;
mult me3;
  me3 = me1;
  multscan thatptr(me2, NEXT);
  dgtp pe2;
  while((pe2 = thatptr()) != 0)
    {
     for(x = 0; x < ((int(pe2 -> dgt) * (exp((count) * (log(radix))))) - 1); x++)
       me3 = me3 * me1;
     count++;
                }
  if(me2.sgn == -1)
    {
     me3 = 0;
     cout << "This is a fractional, beyond this research \n";
    }
 cout << "\n The exponential : \n  ";
 rs = me3;
 printmult(me3);
 if(me3.sgn == +1) ch = '+';
   else ch = '-';
```

```
            return rs;
        }

mult& mult::cmp(mult mc1, mult mc2)
   {
        int pr = 0, d2, d1;
    mult mc3;
      multscan anotherptr(mc2, LAST);
      dgtp p2;
      multscan otherptr(mc1, LAST);
      dgtp p1;
      if((mc1.sgn == +1) && (mc2.sgn == -1))
        {
         mc3 = mc1;
         pr = ptrgt = 1;
        }
      if((mc1.sgn == -1) && (mc2.sgn == +1))
        {
          mc3 = mc2;
          pr = ptrlt = 1;
        }
      if((mc1.numdgt < mc2.numdgt) && (mc1.sgn == mc2.sgn))
```

```
  {
   pr = ptrlt = 1;
   mc3 = mc2;
  }
if((mc1.numdgt > mc2.numdgt) && (mc1.sgn == mc2.sgn))
  {
   pr = ptrgt = 1;
   mc3 = mc1;
  }
if((mc1.numdgt == mc2.numdgt) && (mc1.sgn == mc2.sgn))
  {
   while(((p1 = otherptr()) != 0) && ((p2 = anotherptr()) != 0))
     {
      d2 = int(p2 -> dgt);
      d1 = int(p1 -> dgt);
      cout << "D1 " << d1 << "  D2 " << d2 << "\n";
       if(d2 > d1)
         {
           pr = ptrlt = 1;
           mc3 = mc2;
           cout << "M2 > M1 \n";
         }
```

```
            else if(d2 < d1)
              {
               pr = ptrgt = 1;
               mc3 = mc1;
               cout << "M1 > M2 \n";
              }
        }
  }
if(pr == 0)
  {
    cout << "They are Equal 'But' : ";
    if(mc1.sgn == mc2.sgn) mc3 = 0;
    ptreq = 1;
  }
cout << "\n The Greater is : \n  ";
if(mc3.sgn == +1) ch = '+';
   else ch = '-';
rs = mc3;
 printmult(mc3);
return rs;
    }
```

```
int main()
  {
    mult m1, m2, m;
     textbackground(BLUE);      clrscr();
     cout<< "\n\n" << "      Radix = " << radix << "\n\n";
     cout << "Your operation ('+' OR '-' OR '*' OR '/' OR '%' OR '^' OR '<') : ";
     cin >> ch;
     cout << "\n  Enter your first number list : \n";
     makemult(m1);
     printmult(m1);
     cout << "  Enter your second number list : \n";
     makemult(m2);
     printmult(m2);
     cout << "\n";

     if(ch == '+') m = m1 + m2;
     if(ch == '-') m = m1 - m2;
     if(ch == '*') m = m1 * m2;
     if(ch == '/') m = m1 / m2;
     if(ch == '%') m = m1 % m2;
     if(ch == '^') m = m1 ^ m2;
     if(ch == '<')
```

```
    {
      m = m1 < m2;
      cout << "Your comparison operation ('<' OR '=' OR '>') : ";
      cin >> ch;
      if(ch == '<') return ptrlt;
      if(ch == '>') return ptrgt;
      if(ch == '=') return ptreq;
    }
  cout << "\n  The Result value : \n";
  printmult(m);
  cout << "\n" << "In Decimal : " << ch << m.decimalize() << "\n";
  cout << "Press any key to finish : ";
  cin >> ch;
  return 0;
}
```