Data Structure-Homework4

A1115513 劉沛辰

1. 心得:

這次作業給我的挑戰蠻大的，一開始閱讀功課要求時完全不知道從何下手，後來想了一下決定先做好 BCD 加法，回來做 BCD 加法讓我複習了位元運算，把後 4 位進行 AND 之後做 OR，取得 4 位元做加法，得到 carryin 跟 carryout，之後把後面 16 位都做完。

後面再做 linked list，把物件設為剛剛做的 BCD，這裡遇到很大的 input 問題，分割輸入的字串放入 linked list，然後分別帶入之前做好的 BCD ADD。

最後再進行 reverse，讓他從高位輸出到低位。

2. The advantages of Big number with BCD encoding::

Linked list 的好處就是好插入和增加元素，正好與大數運算一樣，因為無法預測輸入的數字能有多大，所以用 linked list 做儲存，並且原本用 array 儲存就要做尋訪了，反而 array 的優勢(快速定位元素)的功能沒有顯現出來。

3. Design of your linked list:

```
class ListNode {
public:
    BCD64 obj;
    ListNode *next;
    ListNode* reverseLinkedList(ListNode* head);

    ListNode(BCD64 obj) {
        this->obj = obj;
        next = nullptr;
    }
};
```

用自創的 BCD64 做為 LinkedList 的節點資料，還創建了一個 reverse 方便輸出(因為原本 BCD 處理是由低位到高位，輸出時要由高到低)。

4. Review/Improve class BCD64:

Review/Improve(此為更新過的):

```
class BCD64 {
    public:
    BCD64();

    BCD64(unsigned long long bcd);

    BCD64(const BCD64 &bcd_obj);

    BCD64(string &num_str);

    BCD64 add(BCD64 &num, int &carryout, int &carryin) ;

    BCD64 sub(BCD64 &num, int &carryout, int &carryin) ;

    friend ostream &operator<<(ostream &sout, BCD64 &num);

    //private:
    unsigned long long bcd;
};
```

定義多種 constructor，讓使用者能輸入多種資料來建立，分別有空的，輸入 int 的，輸入 BCD member 的，和輸入 string 的，而下面定義了兩個 function，分別是加法和減法，放入 bcd、carryin，carryout，讓輸入的 class bcd 可以和原本的 class bcd 加減法

```
BCD64 BCD64::add(BCD64 &num, int &carryout, int &carryin) {
    BCD64 result;
    unsigned long long sum = 0;

    for(int i = 0; i < 16; ++i) { // 應該確保16位元結果，每個BCD數字4位元組
        int a = bcd & 0x0f;
        int b = num.bcd & 0x0f;
        sum = a + b + carryin;
        if(sum > 15) {
            carryout = 1;
            sum -= 16;
        } else {
            carryout = 0;
        }

        result.bcd |= (sum << (i * 4)); // 將加法結果設置到對應位置
        carryin = carryout;

        // 右移位準備處理下一組BCD位元
        bcd >>= 4;
        num.bcd >>= 4;
    }

    return result;
}
```

利用 and(位元運算子)，把原本 unsigned long long 分成 16 個 part，分割的方法是用和 4b’ 1111 做 and 的方式，取出最後 4 個 bits，然後設一個 sum 做加法，當他 overflow 時，把 sum-=16 並把 carryout 設為 1，然後把位元進行右移再繼續運算。

而減法則是 sum 用減的，當小於 0 把 carryout 設為 1(借位)。

5. Comparison with other data structure:

使用 python 生成了不同長度的加法(資料由小到大排列)

使用 unsigned long long int:

```
Transpose execution time: 291 milliseconds
Transpose execution time: 1208 milliseconds
Transpose execution time: 9322 milliseconds
Transpose execution time: 84165 milliseconds
Transpose execution time: 920292 milliseconds
```

使用 int:

```
Transpose execution time: 314 milliseconds
Transpose execution time: 1190 milliseconds
Transpose execution time: 10691 milliseconds
Transpose execution time: 112444 milliseconds
Transpose execution time: 1047052 milliseconds
```

可以看到一開始差距不大，但是到後面因為分割的次數較多，造成延遲被放大，所以執行時間也變久