5.11

5.12

对于栈(stack)而言 先进的元素优先值（priority）小 后进的元素优先值大，

用additional variable 記錄每個元素的優先值（priority)，先進stack的優先值小，后進的優先值大。priority = 0;將一個數n push 進stack, 先講給n附上priority,然後用insert(),根據這個數的priority值放入，同時更新priority++,當下一次個數需要存入時，將新的priority賦于它。從stack pop 出一個數的時候，用min(),同時更新priority－－。

5.13

从heap的root開始，遞迴分別搜索左子樹和右子樹看是否節點中的key小於k,如果大於k，就停止搜索，這個算法O(K),因為heap不會有一個node的key比k大，但他的子樹卻比K小的情況。

```
//二叉树结点
typedef struct Heap{
    //数据
    int  data;
    //左右孩子指针
    struct Heap *lchild,*rchild;
}

void PreOrder(HeapT,int k){
    if(T->data <= k){
        //访问根节点
        Visit(T);
        //访问左子结点
        PreOrder(T->lchild);
        //访问右子结点
        PreOrder(T->rchild);
```

```
    }
}
```

5.14

簡介：
跟SimHash一樣，MinHash也是LSH的一種，可以用來快速估算兩個集合的相似度。MinHash由Andrei Broder提出，最初用於在搜索引擎中檢測重復網頁。它也可以應用於大規模聚類問題。

原理：
先定義幾個符號術語：
h(x)：把x映射成一個整數的哈希函數。
hmin(S)：集合S中的元素經過h(x)哈希後，具有最小哈希值的元素。
那麼對集合A、B，hmin(A) = hmin(B)成立的條件是A ∪ B 中具有最小哈希值的元素也在 ∩ B中。
這裏有一個假設，h(x)是一個良好的哈希函數，它具有很好的均勻性，能夠把不同元素映射成不同的整數。 所以有，Pr[hmin(A) = hmin(B)] = J(A,B)，即集合A和B的相似度為集合A、B經過hash後最小哈希值相等的概率。

有了上面的結論，我們便可以根據MinHash來計算兩個集合的相似度了。一般有兩種方法：

第一種：使用多個hash函數
為了計算集合A、B具有最小哈希值的概率，我們可以選擇一定數量的hash函數，比如K個。然後用這K個hash函數分別對集合A、B求哈希值，對每個集合都得到K個最小值。比如
Min(A)k={a1,a2,...,ak}，Min(B)k={b1,b2,...,bk}。
那麼，集合A、B的相似度為lMin(A)k ∩ Min(B)kl / lMin(A)k ∪ Min(B)kl，及Min(A)k和Min(B)k中相同元素個數與總的元素個數的比例。

第二種：使用單個hash函數
第一種方法有壹個很明顯的缺陷，那就是計算復雜度高。使用單個hash函數是怎麼解決這個問題的呢？請看：
前面我們定義過 hmin(S)為集合S中具有最小哈希值的一個元素，那麼我們也可以定義hmink(S)為集合S中具有最小哈希值的K個元素。這樣一來，我們就只需要對每個集合求一次哈希，然後取最小的K個元素。計算兩個集合A、B的相似度，就是集合A中最小的K個元素與集合B中最小的K個元素的交集個數與並集個數的比例。

優點：
如果僅僅對兩篇文檔計算相似度而言，MinHash沒有什麼優勢，反而把問題復雜化了。但是如果有海量的文檔需要求相似度，比如在推薦系統中計算物品的相似度，如果兩兩計算相似度，計算量過於龐大。但是用Minhash卻可以大大減少計算量，比如有n個文檔，每個文檔的維度為m，我們可以選取其中k個排列求MinHash，由於每個對每個排列而言，MinHash把一篇文檔映射成壹個整數，所以對k個排列計算MinHash就得到k個整數。那麼所求的MinHash矩陣為n*k維，而原矩陣為n*m維。n>>m時，計算量就降了下來。
參考文獻：
（1） http://en.wikipedia.org/wiki/MinHash
(2) http://fuliang.iteye.com/blog/1025638

5.15

二分法
string a,b
將a 二分，得到apre為字符串首元素到中間的字串，aend為字符串尾元素到中間+1的字串。
同理得到bpre,bend;
apre.size= a.size/2
aend.size = a.size - apre.size
k = a.pre.size
positionPre[first, are.size]

positionEnd[aend.size, end]
hashapre = postfixHash(apre,k),得到apre的hash值。
同理得hashbpre

判斷hashapre是否與hashbpre相等
如果相等，對aend,bend 繼續進行二分操作
并更新positionEnd
如果不相等，對apre,bpre進行二分操作
并更新positionPre

直到K<1,停止二分，此時可以得到最終position

該算法最終需要log(n)

bonus

```
" // BKDR Hash Function
" unsigned int BKDRHash(char *str)
" {
"     unsigned int seed = 131;
"     unsigned int hash = 0;
"
"     while (*str)
"     {
"         hash = hash * seed + (*str++);
"     }
"
"     return (hash & 0x7FFFFFFF);
" }
"
```

由於字符串不可能只有兩位或者三位，不可能為每個係數去人為賦值，但是字符串中又位數的順序，所以可以用不為1的係數的n次方來作為每個字符的係數，這樣就大大降低了碰撞的發生。
在seed的選擇上，一般選擇特殊的奇数2^n-1，因為不為1的奇數可以大大減小碰撞率，同時這樣在CPU的運算中移位和減法會比較快。

最終對應的結果

auto	220087971
break	1201733283
case	224240792
char	224358574
const	1489645747
continue	453557803
default	1552649209
do	13211
double	1810292477
else	228925745
enum	228960337
extern	88412644
float	218932392
for	1765077
goto	233473551
if	13857
int	1816431
long	244713212
register	1789433047
return	2105315144

short	1890957580
signed	1031268144
sizeof	1073828736
static	2109898570
struct	648861
switch	863898632
typedef	2115570911
union	345858955
unsigned	1941414773
void	267193464
volatile	880861684
while	921369849

32個words都沒有發生碰撞。這樣我們可以得到perfect hash function.

為了得到Minimal Perfect Hash Function,
我們改變seed ,得到h1（）,h2（）兩個Function，再構造一個數組實現的函數g(),使得
h()=g(h1()+h2()))mod 32, 得到的h()就是Minimal Perfect Hash Function。

5.21

測試方法:
隨機輸入一組數，測試insert（）size() pop()，例如輸入1 4 2 3 5 0 這組數，測試結構的大小size，以及pop出來這組數的順序，代碼如下:

```
int main()
{
    int size = 0;
    vector<int> output;
    BinomialHeap<int> computer;
    computer.insert(1);
    computer.insert(4);
    computer.insert(2);
    computer.insert(3);
    computer.insert(5);
    computer.insert(0);
    size = computer.hsize();
    for (int i = 0; i<6;i++ ) {
        output.push_back(computer.pop());

    }
    cout << "size =  "<<size << endl;
    for (int i = 0; i < 6; i++) {
        cout << output[i] <<endl;
    }
}
```

測試結果:

```
size =  6
5
4
3
2
1
0
```

可以證實解構正確。