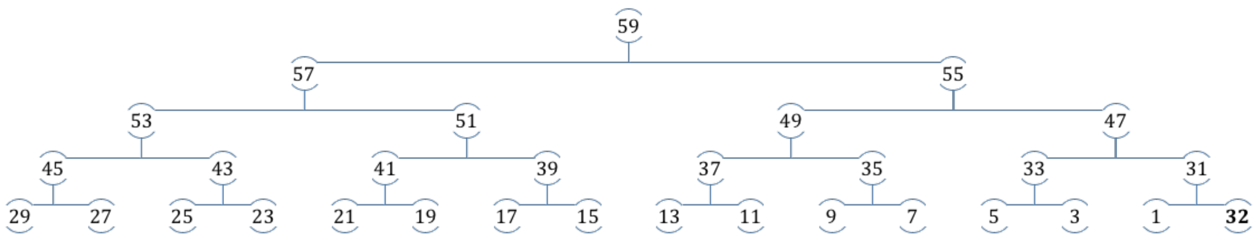
Data Structures and Algorithms, Spring 2015
Homework #5
B03901023 許秉鈞

5.1 (1) R-8.24

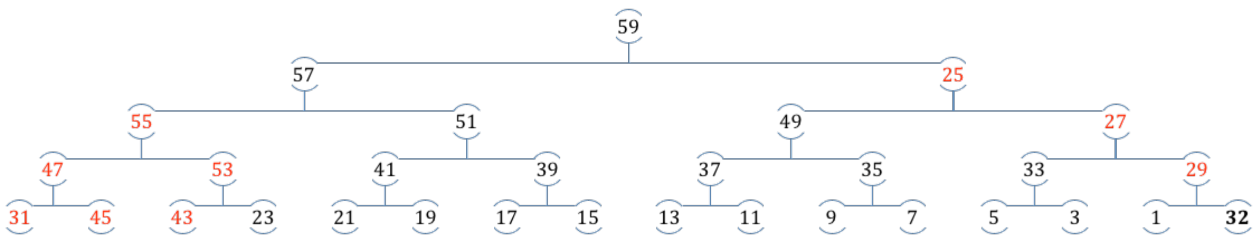
Draw an example of a heap whose keys are all the odd numbers from 1 to 59 (with no repeats), such that the insertion of an entry with key 32 would cause up-heap bubbling to proceed all the way up to a child of the root (replacing that child's key with 32).



[Fig. 1]

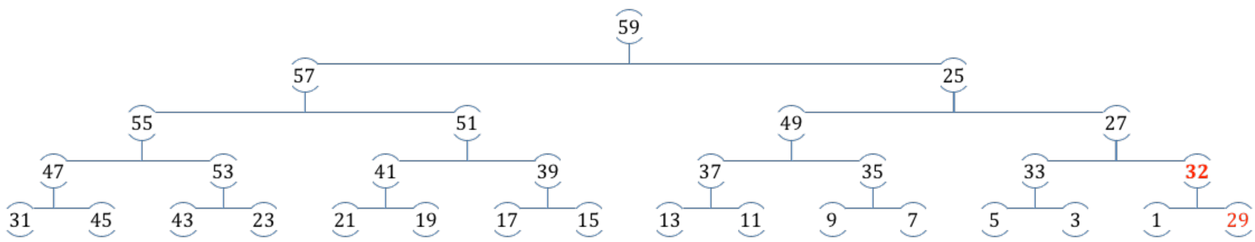
The original (trivial) case of such heap is in [Fig. 1].

Modification on [Fig.1] :
(1) pick up 31, 47, 55
(2) replaced with 29, 27, 25
(3) maintain the heap, and then we can get the desired heap [Fig. 2]

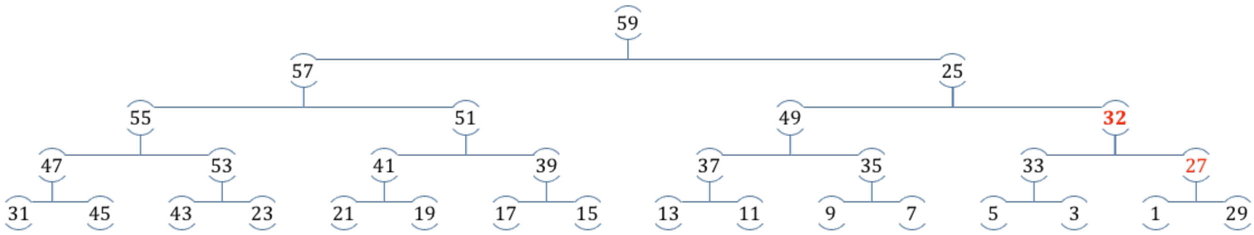


[Fig. 2] -> Answer!

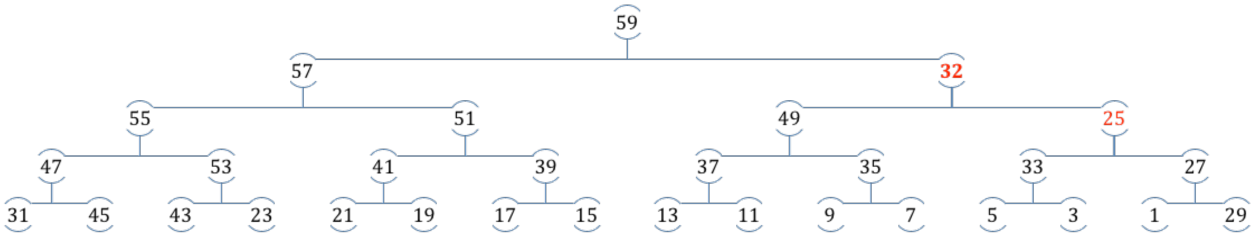
Process of up-heap bubbling:



[Fig. 3]



[Fig. 4]



[Fig. 5]

5.1 (2) C-8.4

Show how to implement the stack ADT using only a priority queue and one additional member variable.

```
class Stack {

    class Node {

        Node(int p, DATATYPE e)
            :priority(p), element(e){};
        int priority; //key
        DATATYPE element; //value
    };

    priority_queue< Node > Q; // operator'<' is defined by node's priority
    int top_priority = 0; //one additional member variable

    void push(DATATYPE input)
    {
        Q.push( Node(top_priority++, input) );
    }
    DATATYPE pop()
    {
        top_priority--;
        return q.pop().element;
    }
};
```

[Fig. 6]

5.1 (3) C-8.14

Given a heap T and a key k, give an algorithm to compute all the entries in T with a key less than or equal to k. For example, given the heap of Figure 8.12(a) and query $k = 7$, the algorithm should report the entries with keys 2, 4, 5, 6, and 7 (but not necessarily in this order). Your algorithm should run in time proportional to the number of entries returned.

```
// suppose heap T is realized by "Level Numbering" method
// that is: (see p.295)
// If v is the root of T,then f(v)= 1
// If v is the left child of node u, then f(v) = 2f(u)
// If v is the right child of node u, then f(v) = 2f(u) + 1

class Node {

    Node(int _k, DATATYPE _v)
        :key(_k), value(_v){};
    int key;
    DATATYPE value;
};
void dfs( Node heap[], int id, int k)
{
    if( heap[id].key > k ) return;
    cout << heap[id].value << " ";
    dfs( heap, id * 2, k);
    dfs( heap, id * 2 + 1, k);
}
void findKeyLowerThanK( Node T[], int k )
{
    dfs( T, 1, k ); //1-based
}
```

[Fig. 7]

Ans. (see Fig.7)

In order to run in time proportional to the number of entries returned and NOT modifying the original heap, I chose depth-first search to realize the algorithm “findKeyLowerThanK”.

5.1 (4)

Main Idea:

A technique for quickly estimating how similar two sets are.

Application:

1. Being applied in large-scale clustering problems, such as *clustering documents by the similarity of their sets of words*.
2. In data mining, Cohen et al. (2001) use MinHash as a tool for *association rule learning*.

Techniques:

1. MinHash Signature for Sets

將這些比較長、比較大的每一個集合都以一個特殊的*signature*來代替，單純透過比較這些*signatures*，就能知道其所代表的集合間的 *Jaccard similarity*；如此一來，原本計算過程中需要對上萬個元素進行比較的運算就能替換為*signature*長度的運算。

2. Locality-Sensitive Hashing for Documents

*MinHash*中，我們透過*LSH*將具有類似*signature*的集合分到相同的*bucket*；由於較相似的集合會比不相似的集合們更有機會分配到同一個*bucket*，我們只需要在輸出結果前檢查那些被放在同一個*bucket*中的集合即可。

Processes:

1. 首先須決定*minhash signature*的長度 *n*，並對各個集合算出其*minhash signature*。
2. 依據應用情境選擇 *b*與 *r*值，要記住 *b*與 *r*決定了集合被納入*candidate pairs*的機率；如果我們希望避免*false negative*，則減少 *r*值，增加 *b*值；如果希望限制*false positive*，則可增加 *r*，減少 *b*。
3. 以選擇的參數進行*LSH*運算，取得*candidate pairs*。
4. 對各個*candidate pair*的*minhash signature*計算其 *Jaccard similarity*，確認是否相似。
5. 若前述步驟都進行完畢尚有運算資源可運用，則可實際取得集合內容，再次確認其相似性。

Jaccard similarity Def:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$

ref:

http://web.stanford.edu/class/cs276b/handouts/minhash.pdf

http://en.wikipedia.org/wiki/MinHash

http://shihpeng.org/tag/minhash/

5.1 (5)

```
typedef unsigned long long int longint;

// O(1) Time Complexity
longint postfixHash(string str,int k);

int binary_search(string str_1, string str_2)
{
    int left = 0;
    int right = str.size() - 1;
    while( left <= right )
    {
        int mid = (left + right) / 2;

        if(postfixHash(str_1, mid) != postfixHash(str_2, mid))
        {
            if(str_1.size() == 1)
                return mid;
            else
                left = mid + 1;
        }
        else // (postfixHash(str_1, mid) == postfixHash(str_2, mid))
            right = mid - 1;
    }
    return "KEY_NOT_FOUND";
}
```

[Fig. 8]

Ans. (see Fig.8)

//Discuss and justify the time complexity of your algorithm:
我的做法是，引用二元搜尋樹的概念加上postfixHash()，目標為找到在兩長度相同的字串中，找到唯一不同字元的位置(因此回傳 int)。Binary_search的時間複雜度為O(log N)，而postHash()以由題目訂定為O(1)，所以總共為O(log N)。

5.1 (6)

```
typedef unsigned long long int longint;

longint hash(string s) //BONUS
{
    longint out = 0;
    for(int i = 0; i < s.size(); i++)
    {
        out *= 27;
        out += s[i] - 'a' + 1; //hash("a") == 1
    }
    return out;
}
// consider hash("register"):
// 190329075127, still in the range of longint
// consider hash("volatile"):
// 236112196676, still in the range of longint
```

[Fig. 9]

Ans. (see Fig.9)

為了方便計算，我使用27進位而非26進位，讓hash(“a”) 為1。
如此一來，這個hash可以用*linear time*的時間算出一個給定string 的hash value，有效率地計算此值，且發生collision的機率極低，在給定的32 個 standard words中無碰撞發生。