

Data Structures and Algorithms: Homework #5

Due on May 26, 2015 at 16:20

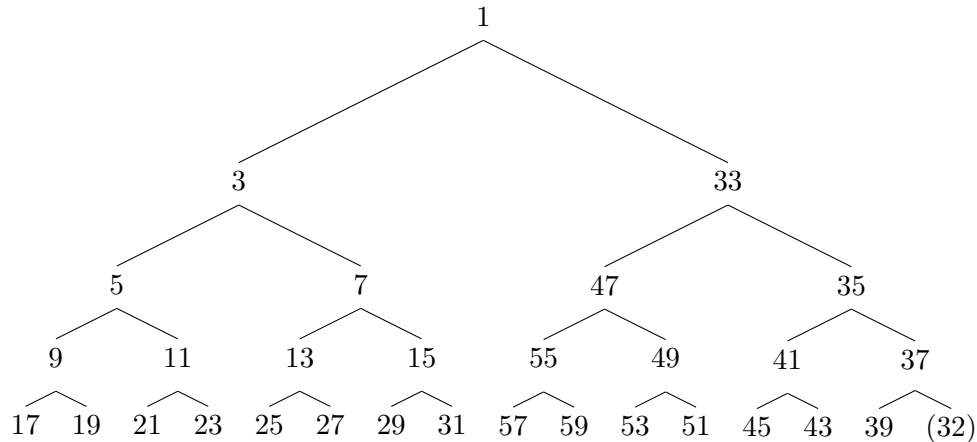
Instructors Hsuan-Tien Lin, Roger Jang

Tim Liou (b03902028)

5.1 Heap and Hash

(1) Complete Exercise R-8.24 of the textbook.

Consider a min heap. The insertion of an entry with key 32 would cause up-head bubbling to proceed all the way up to a child of the root in this heap.



(2) Complete Exercise C-8.4 of the textbook.

Combine each key we want to insert to the stack with a priority variable. These combinations would be insert to the MaxPriorityQueue which compare elements by the priority variable. Note that the basic idea of this method is the fact that every new element is pushed in stack with a priority higher than the current one. Then, MaxPriorityQueue will behave in the LIFO way, which is what we want.

```

1  class Stack {
2      class Element { int priority, Key element; };
3      MaxPriorityQueue<Element> MaxPQ;
4      int top_priority = 0;
5
6      void push(Key key) { MaxPQ.insert(Element(top_priority++, key)); }
7      void pop() { top_priority--; MaxPQ.removeMax(); }
8      const Key& top() { return MaxPQ.max(); }
9      int size() { return top_priority; }
10     bool empty() { return (top_priority == 0); }
11 };

```

(3) Complete Exercise C-8.14 of the textbook.

```

1: function FINDLEQELEMENTINHEAP(heap, key)
2:   while heap.top() ≤ key do
3:     insert heap.top() into the output list, and do heap.removeMin()
4:   end while
5:   return the output list
6: end function

```

Algorithm 1: Compute all the entries in a heap with a key less than or equal to the value

(4) Hash function is everywhere. Use any search engine to study the term MinHash Explain to the TAs what it is and why it is useful. Also, cite the website that you learn the term from.

I learn MinHash from <http://shihpeng.org/tag/minhash/> and <http://en.wikipedia.org/wiki/MinHash>.

MinHash is a technique to estimate how similar two sets are. We can use Jaccard similarity coefficient to find out the similarity between two sets.

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

However, it is slow to compute $J(A, B)$ by the the way to check every elements in this two sets. MinHash was designed to compute $J(A, B)$ fast.

Consider a case, $S_1 = \{b, c\}$, $S_2 = \{a, b\}$, $S_3 = \{b\}$

Element	S_1	S_2	S_3
a	0	1	0
b	1	1	1
c	1	0	0
d	0	0	0

And we permuate the element,

Element	S_1	S_2	S_3
c	1	0	0
d	0	0	0
b	1	1	1
a	0	1	0

Let h_{min}^k be a hash function, which k is a permutation of element. In this case, k is $cdba$. And h_{min}^k is defined as the symbol of the first non-zero row. In this case, $h_{min}^{cdba}(S_1) = c$, $h_{min}^{cdba}(S_2) = b$, $h_{min}^{cdba}(S_3) = a$. We can find that the value of $J(A, B)$ is as same as the possibility of $h_{min}^k(A) = h_{min}^k(B)$. The following is the proof.

For any two sets, we have three possible result for every elements.

Type 1 both are zero, that is, both sets do not contain this element.

Type 2 one is zero and the other is one, that is, one set contains this element.

Type 3 both are one, that is, both sets contain this element.

The number of Type 1 is M_1 , the number of Type 2 is M_2 , and the number of Type 3 is M_3 .

$$J(A, B) = \frac{M_3}{M_3 + M_2}$$

The possibility of $h_{min}^k(A) = h_{min}^k(B)$ is

$$\begin{aligned} P(\text{Type 3} \mid (\text{Type 2 or Type 3})) &= \frac{\frac{M_3}{M_1+M_2+M_3}}{\frac{M_2+M_3}{M_1+M_2+M_3}} \\ &= \frac{M_3}{M_3 + M_2} = J(A, B) \end{aligned}$$

If we choose enough number of k , the possibility of $h_{min}^k(A) = h_{min}^k(B)$ will be closer to $J(A, B)$.

This method would be more efficient than the naive algorithm. The time complexity change from $O(M)$ to $O(1)$ since we don't have to go through all elements. This could be useful for the sets contain lots of elements.

(5) Describe an algorithm to find out the position that the two strings differ efficiently. Briefly discuss and justify the time complexity of your algorithm.

```

1: function FINDDIFFPOSTION( $s1, s2$ )
2:    $\text{start} \leftarrow 0$ 
3:    $\text{end} \leftarrow \text{the length of string} - 1$ 
4:    $\text{middle} \leftarrow \lfloor (\text{start} + \text{end}) / 2 \rfloor$ 
5:   while  $\text{start} \neq \text{end}$  do
6:     if  $\text{POSTFIXHASH}(s1, \text{end} - \text{middle}) \neq \text{POSTFIXHASH}(s2, \text{end} - \text{middle})$  then
7:        $\text{start} \leftarrow \text{middle} + 1$ 
8:        $\text{middle} \leftarrow \lfloor (\text{start} + \text{end}) / 2 \rfloor$ 
9:     else
10:       $\text{end} \leftarrow \text{middle}$ 
11:       $\text{middle} \leftarrow \lfloor (\text{start} + \text{end}) / 2 \rfloor$ 
12:    end if
13:  end while
14:  return  $\text{start}$ 
15: end function
```

Algorithm 2: Find the position that the two strings differ efficiently

The time complexity of this algorithm is $O(\log n)$, which n is the length of two strings, because we check whether the hash values of the last half of two strings are the same. If they are same, it

means the position that two strings differ is in first half. Otherwise, the position is in the last half. After one loop in **while**, the length of string we need to check becomes half. This implies that the time complexity of this algorithm is $O(\log n)$.

(6) Construct a perfect hash function that is efficiently computable for the following 32 standard keywords in C. You need to explain why the hash function is perfect and why it is efficiently computable to get the full bonus.

Construct a simple hash function which would not make the values of these 32 keywords collision. For example, $h(s) = s.front() + 27 \times s.back() + 27^2 \times s.length()$. It can show that these hashcodes are not the same.

```
1  if: 4317
2  do: 4555
3  for: 5367
4  int: 5424
5  void: 5734
6  case: 5742
7  else: 5744
8  long: 5805
9  enum: 5960
10 auto: 6010
11 goto: 6016
12 char: 6093
13 while: 6491
14 break: 6632
15 union: 6732
16 const: 6876
17 float: 6879
18 short: 6892
19 static: 7162
20 signed: 7189
21 double: 7201
22 sizeof: 7243
23 switch: 7297
24 extern: 7445
25 return: 7458
26 struct: 7621
27 typedef: 7973
28 default: 8335
29 unsigned: 8649
30 continue: 8658
31 volatile: 8677
32 register: 9024
```

Then, call these hashcodes $S = \{a_1, a_2, \dots, a_{32}\}$. Note that the values in S are sorted. Use binary search to find the value in S , and the perfect hashcode is the order number of the value. For instance, the hash function value of the keyword 'struct' is 7621, then the perfect hashcode is 26.

This method is efficient because we can build the table in advance. Besides, by using this method, we can change types of objects which are hard to be searched by binary search to number, which is easy to be searched by binary search.

5.2 Distributed System

(1) Finish/rewrite the BinomialHeap class, and describe how you test whether the data structure is correct.

I wrote a simple cpp file to test this data structure.

```
1  #include <iostream>
2  #include "binomial_heap.h"
3
4  #define ArraySize(A) (sizeof((A)) / sizeof(*(A)))
5
6  int main()
7  {
8      BinomialHeap<int> bh, bh2;
9
10     int input[] = {1, 29, 33, 234, 234, 92, 11};
11     int input2[] = {2, 30, 31, 34, 92, 0, 111};
12
13     std::cout << "*****" << std::endl;
14     for (unsigned int i = 0; i < ArraySize(input); i++)
15     {
16         bh.insert(input[i]);
17         std::cout << "insert [" << input[i] << "] to bh,\t"
18             << "bh.size = " << bh.size << std::endl;
19     }
20     std::cout << "*****" << std::endl;
21
22     std::cout << "*****" << std::endl;
23     for (unsigned int i = 0; i < ArraySize(input2); i++)
24     {
25         bh2.insert(input2[i]);
26         std::cout << "insert [" << input[i] << "] to bh2,\t"
27             << "bh2.size = " << bh2.size << std::endl;
28     }
```

```

29     std::cout << "*****" << std::endl;
30
31     std::cout << "*****" << std::endl;
32     std::cout << "bh.size = " << bh.size << std::endl;
33     std::cout << "bh2.size = " << bh2.size << std::endl;
34     bh.merge(bh2);
35     std::cout << "Merge bh2 to bh" << std::endl;
36     std::cout << "bh.size = " << bh.size << std::endl;
37     std::cout << "bh2.size = " << bh2.size << std::endl;
38     std::cout << "*****" << std::endl;
39
40     std::cout << "*****" << std::endl;
41     while (bh.size)
42     {
43         std::cout << "bh.top() is " << bh.top() << " and do bh.pop()" << std::endl
44             ;
45         bh.pop();
46     }
47     std::cout << "bh.size = " << bh.size << std::endl;
48     std::cout << "*****" << std::endl;
49     return 0;
50 }

```

I can check the correctness of this data structure by the output.

```

1  *****
2  insert [1] to bh, bh.size = 1
3  insert [29] to bh, bh.size = 2
4  insert [33] to bh, bh.size = 3
5  insert [234] to bh, bh.size = 4
6  insert [234] to bh, bh.size = 5
7  insert [92] to bh, bh.size = 6
8  insert [11] to bh, bh.size = 7
9  *****
10 *****
11 insert [1] to bh2, bh2.size = 1
12 insert [29] to bh2, bh2.size = 2
13 insert [33] to bh2, bh2.size = 3
14 insert [234] to bh2, bh2.size = 4
15 insert [234] to bh2, bh2.size = 5
16 insert [92] to bh2, bh2.size = 6
17 insert [11] to bh2, bh2.size = 7
18 *****

```

```
19 *****
20 bh.size = 7
21 bh2.size = 7
22 Merge bh2 to bh
23 bh.size = 14
24 bh2.size = 0
25 *****
26 *****
27 bh.top() is 234 and do bh.pop()
28 bh.top() is 234 and do bh.pop()
29 bh.top() is 111 and do bh.pop()
30 bh.top() is 92 and do bh.pop()
31 bh.top() is 92 and do bh.pop()
32 bh.top() is 34 and do bh.pop()
33 bh.top() is 33 and do bh.pop()
34 bh.top() is 31 and do bh.pop()
35 bh.top() is 30 and do bh.pop()
36 bh.top() is 29 and do bh.pop()
37 bh.top() is 11 and do bh.pop()
38 bh.top() is 2 and do bh.pop()
39 bh.top() is 1 and do bh.pop()
40 bh.top() is 0 and do bh.pop()
41 bh.size = 0
42 *****
```