CS 310

Assignment 0911 - Algorithm Analysis

Bill Jin

31/35

Analysis for push function:

1.**push** function code :

```
1  void push(unsigned element)
2  {
3    heap.push_back(element);
4    bubble_up(heap.size() - 1);
5  }
```

2.**bubble_up** function code :

```
1  void bubble_up(size_t position)
2  {
3    if (position != 0)
4    {
5      size_t parent = (position - 1) / 2;
6      while (position != 0 && heap.at(position) > heap.at(parent))
7      {
8        std::swap(heap.at(position), heap.at(parent));
9        position = parent;
10       parent = (position - 1) / 2;
11     }
12   }
13 }
```

Analysis: This algorithm is used to place the node into its right place by using while loop swapping its parent node with if the node is greater than that. And for the input size will be the size of the heap we build.

For the **worst cases** (Big-O), the operations that are counted are:

In push function :

- the push_back function, count as 1 operation,

In bubble_up function :

- the if condition comparison on line 3, count as 1 operation,
- the assignment of parent, count as 1 operation,
- while loop condition, since it has 3 comparisons, count as $\lfloor \log_2(n+1) \rfloor \times 3$ operations,
- swap function, count as $\lfloor \log_2(n+1) \rfloor \times 2$ operations,
- assignment of position, count as $\lfloor \log_2(n+1) \rfloor \times 1$ operations,
- parent assignment on line 10, count as $\lfloor \log_2(n+1) \rfloor \times 1$ operations,

please use lg, not log_2

- one more for loop condition check, count as $\lfloor \log_2(n+1) \rfloor \times 3$ operations.

After sum up all, the number of times of all operations that are executed is:

$$\lfloor \log_2(n+1) \rfloor \times 10 + 3$$

For the **best cases** (Big-Ω), the operations that are counted are:

In push function :

- the push_back function, count as 1 operation.

In bubble_up function :

- we assume it only has root which means only run if condition, count as 1 operation.

After sum up all, the number of times of all operations that are executed is :

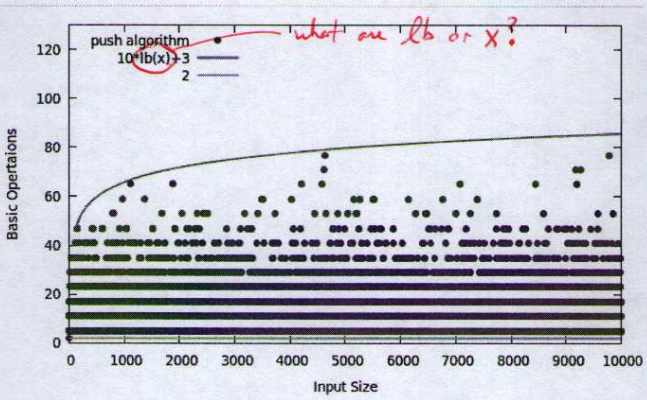
2 operations

Since there are while loop and if conditions which could cause the best and the worst cases. Therefore, we can find out that this algorithm belongs to Big-Oh and Big-Omega efficiency class.

$$T(n) \in O(\log_2(n)) \text{ for } c = 10, \text{ all } n \geq 0$$

$$T(n) \in \Omega(1)$$

And the resulting data file is plotted, we get the following. Also plotted on the same axes are the scaled standard functions $\log_2(n+1) \times 10 + 3$ and 2 which illustrate that the algorithm belongs to big-O and big-Ω .



```
1  //This is the header file for ~~main function.~~ heap class
2  #ifndef HEAP
3  #define HEAP
4  #include <cassert>
5  #include <vector>
6
7  /*
8   * a class to implement a maxheap that stores unsigned integers
9   */
10 class Heap
11 {
12  public:
13
14   /**
15    * The constructor of an empty heap has nothing to do
16    */
17   Heap() { }
18
19   /**
20    * The destructor has nothing to do
21    */
22   ~Heap() {}
23
24   /**
25    * Disallow the copy and move constructors and the copy and move
26    * assignment operators
27    */
28   Heap(const Heap& rhs) = delete;
29   Heap(Heap && rhs) = delete;
30   Heap& operator= (const Heap& rhs) = delete;
31   Heap& operator= (Heap&& rhs) = delete;
32
33   /**
34    * add an element to the heap
35    * @param element the element to add
36    */
37   void push(unsigned element)
38   {
39     heap.push_back(element);
40     bubble_up(heap.size() - 1);
41   }
42
43   /**
44    * delete and return the largest element of the heap
45    * @return the deleted element
46    */
47   unsigned pop()
48   {
49     assert(heap.size() > 0);
50     unsigned value_to_return = heap.at(0);
51     heap.at(0) = heap.at(heap.size() - 1);
52     heap.pop_back();
53     percolate_down(0);
54     return value_to_return;
55   }
56
57   /**
58    * this function will return the size of the heap
59    * @return the size of the heap
60    */
61   size_t size() const
62   {
63     return heap.size();
64   }
65
66   /**
67    * delete all the elements in the heap
68    * @return a boolean variable 0
69    */
70   bool empty() const
71   {
72     return heap.size() == 0;
73   }
```

there is no counter?
-2


```
74
75   /**
76    * just for debugging use
77    * display the contents of the heap as a vector to stdout
78    */
79   void dump() const
80   {
81     for (auto element : heap)
82     {
83       std::cout << element << ' ';
84     }
85     std::cout << std::endl;
86   }
87
88 private:
89
90   /**
91    * this function will percolate down the root element into the right
92    * position.
93    * @param position is the index of the root in heap
94    */
95   void percolate_down(size_t position)
96   {
97     //declare left and right children
98     size_t left_child_position;
99     size_t right_child_position;
100
101    //assign indexes into both children in those two variable
102    left_child_position = 2 * position + 1;
103    right_child_position = 2 * position + 2;
104
105    //declare loop control variable to see if the loop can iterate more
106    bool loop_guard = true;
107
108    //this loop continues iterates until the element find the right position.
109    //and it will always iterates if there is at least 1 child.
110    while (number_of_chidren(position) >= 1 && loop_guard)
111    {
112      //if the element has 2 children.
113      if (number_of_chidren(position) == 2)
114      {
115        if (heap.at(left_child_position) <= heap.at(right_child_position))
116        {
117          //if the element is smaller than its right child
118          if (heap.at(position) <= heap.at(right_child_position))
119          {
120            std::swap(heap.at(position), heap.at(right_child_position));
121            position = right_child_position;
122          }
123          else
124          {
125            //the element is bigger than its all children, loop ends.
126            loop_guard = false;
127          }
128        }
129        else //right child is smaller than left child
130        {
131          if (heap.at(position) <= heap.at(left_child_position))
132          {
133            std::swap(heap.at(position), heap.at(left_child_position));
134            position = left_child_position;
135          }
136          else
137          {
138            //the element is bigger than its all children, loop ends.
139            loop_guard = false;
140          }
141        }
142      }
143      else if (number_of_chidren(position) == 1) // element only has one child
144      {
145        if (heap.at(position) < heap.at(left_child_position))
146        {
```

```
147           std::swap(heap.at(position), heap.at(left_child_position));
148           position = left_child_position;
149         }
150         else
151         {
152           //the element is bigger than its all children, loop ends.
153           loop_guard = false;
154         }
155       }
156       //assign the new index of the children of the element
157       left_child_position = 2 * position + 1;
158       right_child_position = 2 * position + 2;
159       }
160     ((void)0);
161   }
162
163   /**
164    * this function will find the number of the element
165    * @return this will return the number of the element
166    */
167   unsigned number_of_chidren(size_t index)
168   {
169     //declare two children and convert size_t into unsigned
170     unsigned left_c = static_cast<unsigned>(index * 2 + 1);
171     unsigned right_c = static_cast<unsigned>(index * 2 + 2);
172     if (left_c >= heap.size() && right_c >= heap.size())
173     {
174       //there is no child for this element
175       return 0;
176     }
177     else if (right_c >= heap.size() && left_c < heap.size())
178     {
179       //there is one child for this element
180       return 1;
181     }
182     else if (right_c < heap.size() && left_c < heap.size())
183     {
184       //the element has two children
185       return 2;
186     }
187     //just for avoiding waning and this will never be executed
188     const unsigned ERROR = 3;
189     return ERROR;
190   }
191
192   /**
193    *  this function will put the push-back element into its right position
194    */
195   void bubble_up(size_t position)
196   {
197     if (position != 0)
198     {
199       size_t parent = (position - 1) / 2;
200       while (position != 0 && heap.at(position) > heap.at(parent))
201       {
202         std::swap(heap.at(position), heap.at(parent));
203         position = parent;
204         parent = (position - 1) / 2;
205       }
206     }
207   }
208
209   std::vector<unsigned> heap;
210 };
211
212 #endif
213
214
215
```

supposed to delete this line

missing @param -1

incorrect logic -1

missing @param