cs 5800 - hw9

Yijing Xiao

November 2021

1 Exercise 17.3-3

Let the potential function to be $\Phi(D_i) = kn_i * ln(n_i)$ where n_i is the number of elements in the binary heap, and k is a big enough constant. The amortized cost $\widehat{C}_i$ of the ith operation with respect to potential function Φ is defined by $\widehat{C}_i = C_i + \Phi(D_i) - \Phi(D_{i-1})$

a. INSERT
If the ith operation inserts into a nonempty heap, then $n_i = n_{i-1} + 1$ and the amortized cost is

$$\begin{aligned}\widehat{C}_i &= C_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq kln(n_i) + kn_iln(n_i) - kn_{i-1}ln(n_{i-1}) \\ &< 2klnn_i + kn_iln(\frac{n_i}{n_{i-1}})\end{aligned}$$

For $nln(\frac{n}{n-1})$, we have

$$\begin{aligned}nln(\frac{n}{n-1}) &= nln(1 + \frac{1}{n-1}) \\ &= ln(1 + \frac{1}{n-1})^n \\ &= ln(e^{\frac{1}{n-1}})^n \\ &= ln(e^{\frac{n}{n-1}}) \\ &= \frac{n}{n-1} \leq 2\end{aligned}$$

If $n \geq 2$, then $nln(\frac{n}{n-1}) \leq 2$

$$\begin{aligned}\widehat{C}_i &< 2kln(n_i) + kn_iln(\frac{n_i}{n_{i-1}}) \\ &\leq 2kln(n_i) + 2k \\ &= O(lgn_i)\end{aligned}$$

b. EXTRACT-MIN
If the ith operation extracts from a heap with more than 1 item, then $n_i = n_{i-1} - 1$ and the amortized cost is

$$\begin{aligned}
\widehat{C}_i &= C_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq kln(n_{i-1}) + kn_i ln(n_i) - kn_{i-1} ln(n_{i-1}) \\
&\leq kln(n_{i-1}) + k(n_{i-1} - 1) ln(n_{i-1} - 1) - kn_{i-1} ln(n_{i-1}) \\
&= kln(n_{i-1}) + kn_{i-1} ln(n_{i-1} - 1) - kln(n_{i-1} - 1) - kn_{i-1} ln(n_{i-1}) \\
&= kln(\tfrac{n_i - 1}{n_{i-1} - 1}) + kn_{i-1} ln(\tfrac{n_{i-1} - 1}{n_{i-1}}) \\
&< kln(\tfrac{n_i - 1}{n_{i-1} - 1}) \\
&\leq kln2 = O(1)
\end{aligned}$$

2 Exercise 17.3-6

Algorithm 1 Implement a queue using two stacks

```
1: function ENQUEUE-STACK(stack1, stack2, value)
2:     stack1.push(value)
3: end function
4:
5: function DEQUEUE-STACK(stack1, stack2)
6:     if stack2 is not empty then
7:         return stack2.pop()
8:     else
9:         while stack1 is not empty do
10:            stack2.push(stack1.pop())
11:        end while
12:        if stack2 is empty then
13:            return −1
14:        else
15:            return stack2.pop()
16:        end if
17:    end if
18: end function
```

Use accounting method:

The actual costs of the operations were:

PUSH	**1**
POP	**1**
MULTIPOP	**min(k,s)**

where k is the argument supplied to MULTIPOP and s is the stack size when it is called.
Let's assign the following amortized cost:

PUSH	**2**

POP **0**
MULTIPOP **0**

Amortized cost of ENQUEUE = 1+2 = 3
Amortized cost of DEQUEUE:
if stack 2 is empty
Amortized cost = $2^0 = 1$
if stack 2 is nonempty
Amortized cost = 1+0 = 1
Therefore, the amortized cost of both ENQUEUE and DEQUEUE are $O(1)$.

3 Exercise 19.2-1

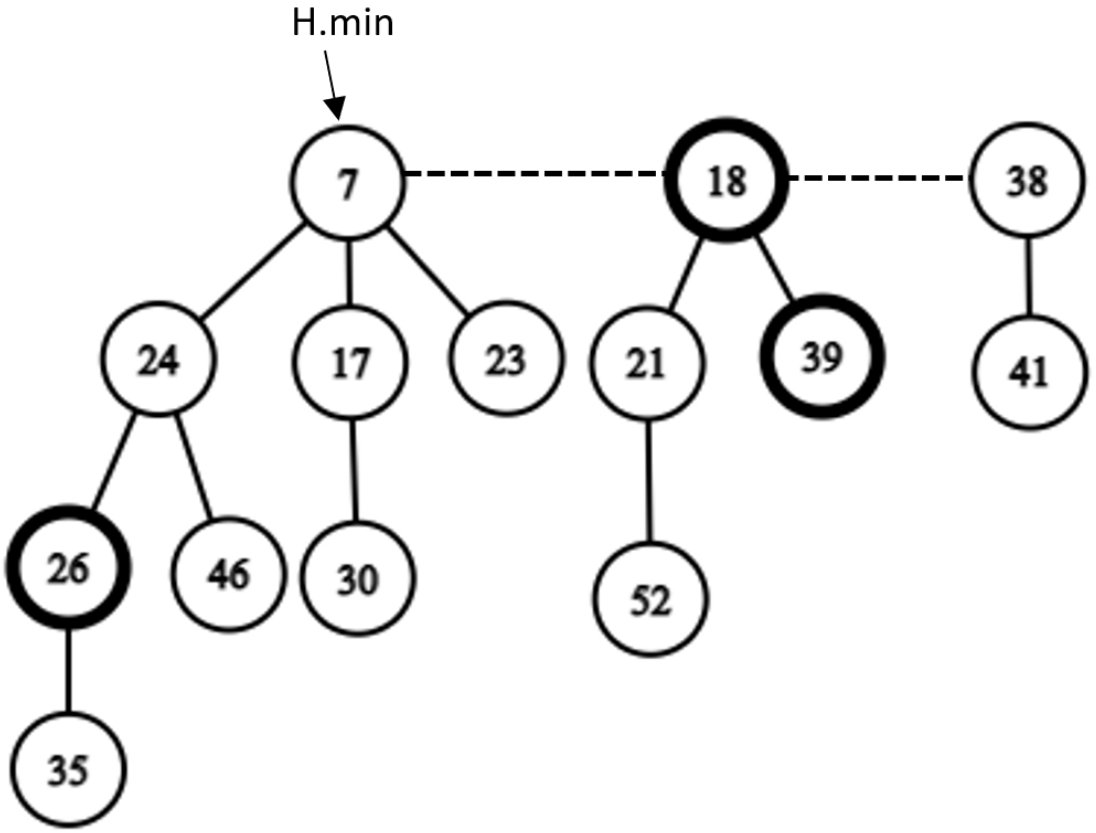

Figure 1: initial heap

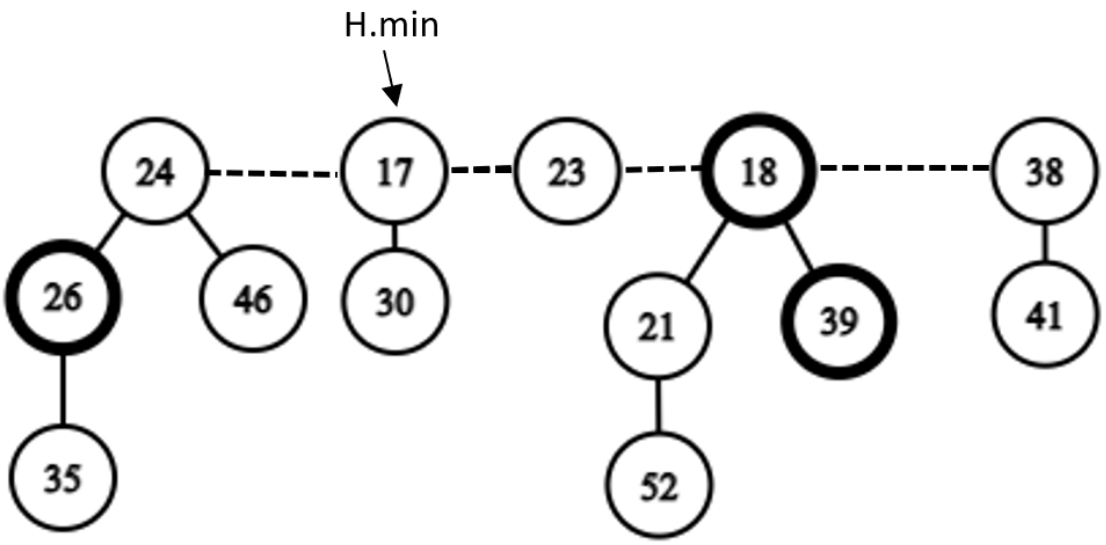

Figure 2: step 1

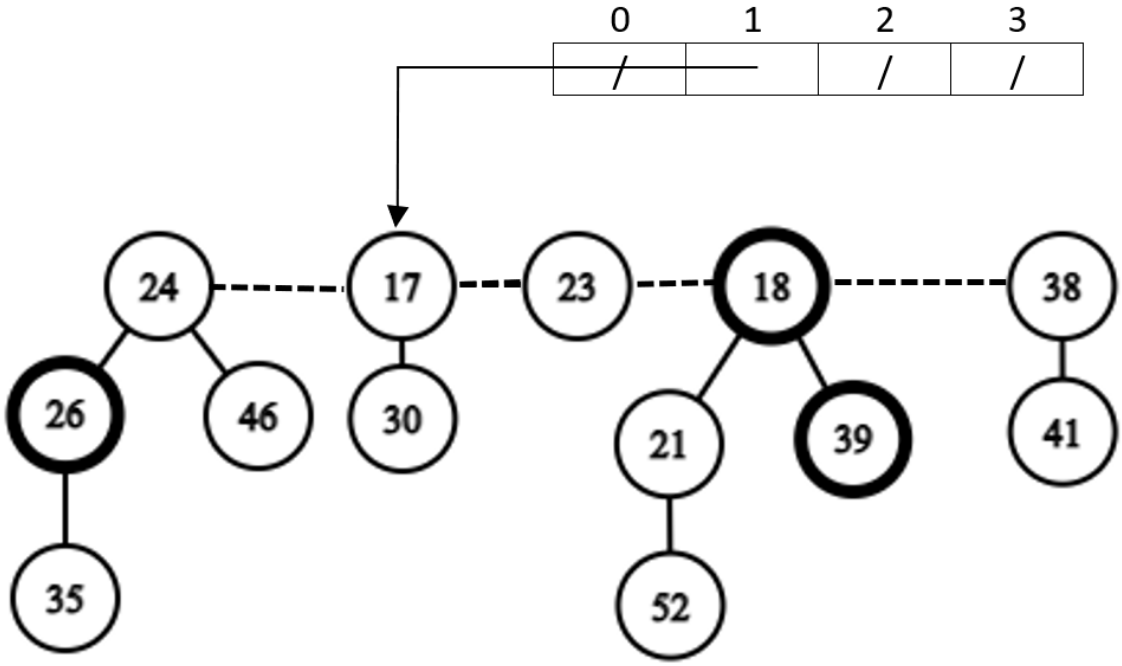

Figure 3: step 2

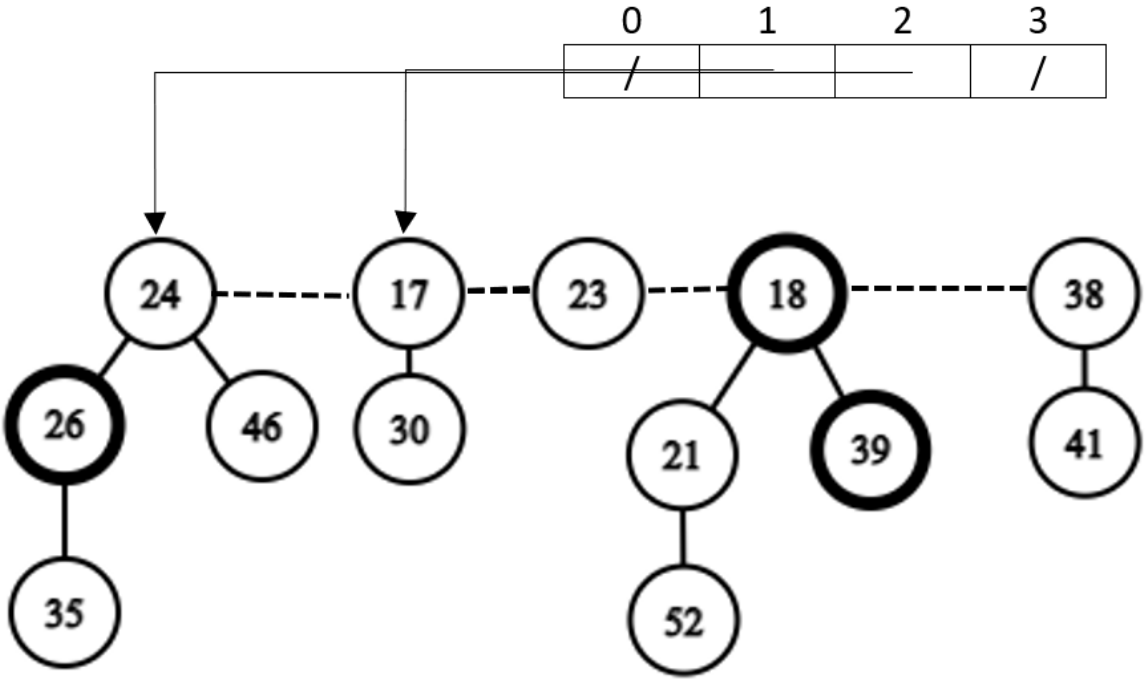

Figure 4: step 3

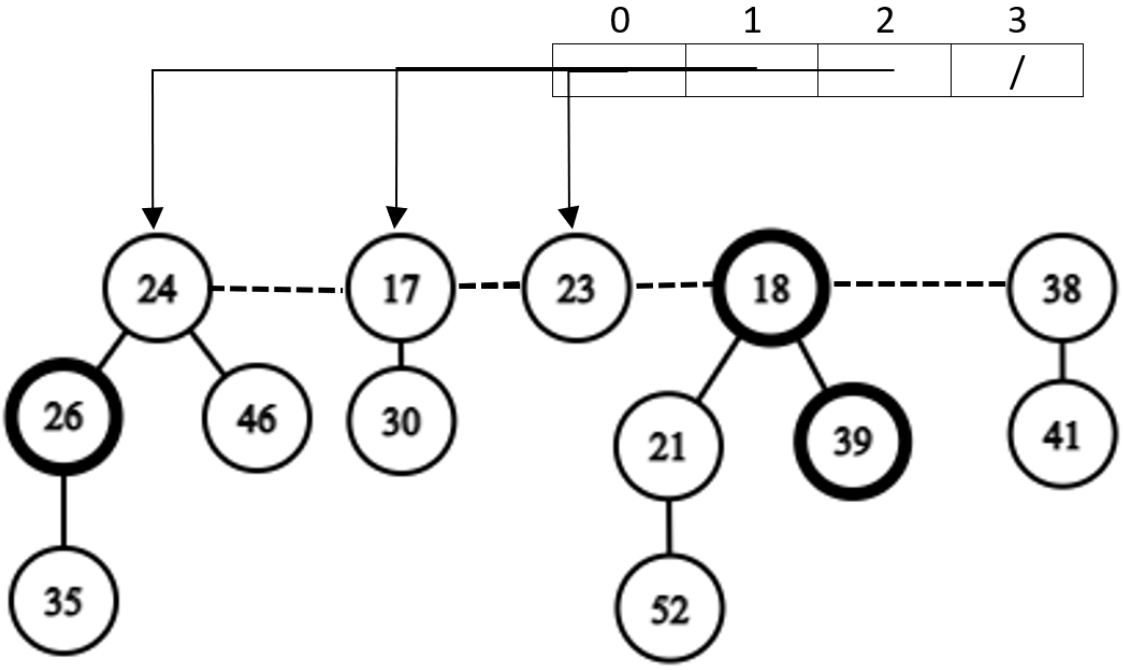

Figure 5: step 4

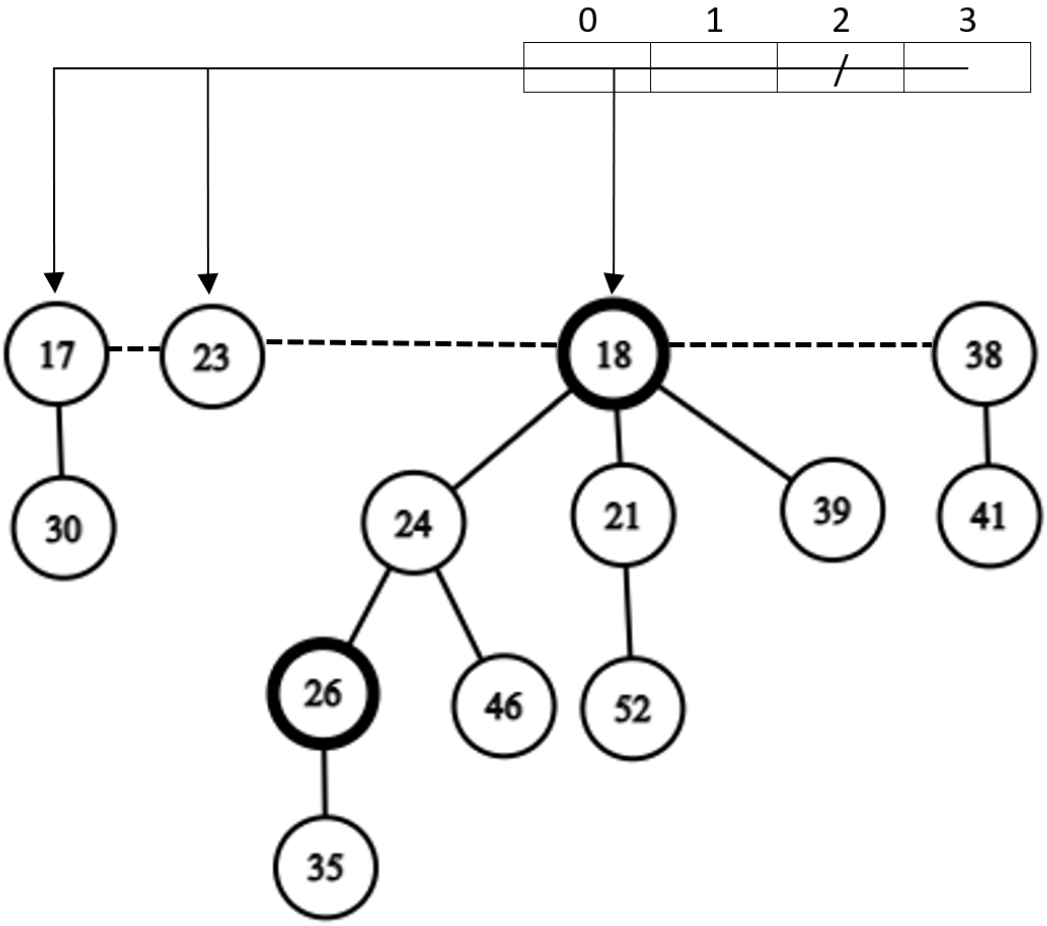

Figure 6: step 5

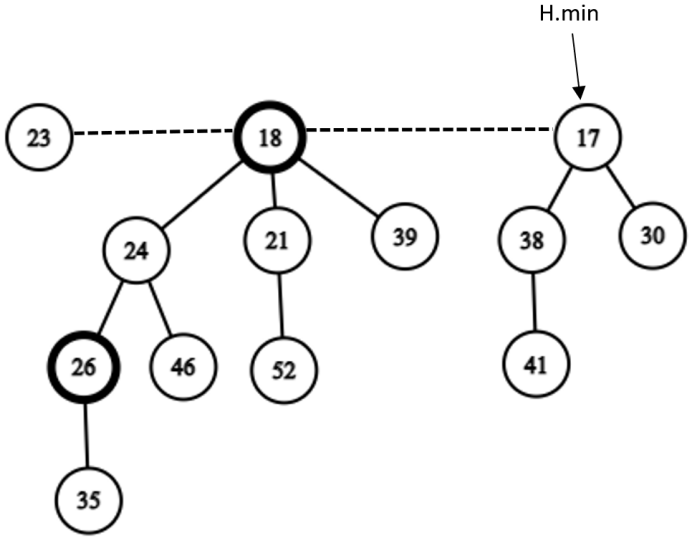

Figure 7: after consolidate

4 Implement binomial heaps

```
1  class Node:
2          def __init__(self, k=None):
3              self.p = None
4              self.key = k
5              self.degree = 0
6              self.child = None
7              self.sibling = None
8
9  class binomialHeap:
10     def __init__(self, head=None):
11         self.head = head
12
13     def make_heap(self):
14         heap = binomialHeap()
15         return heap
16
17     def minimum(self):
18         y = None
19         x = self.head
20         mini = float('inf')
21         while x != None:
22             if x.key < mini:
23                 mini = x.key
24                 y = x
25             x = x.sibling
26         return y
27
28     def link(self,y, z):
29         y.p = z
30         y.sibling = z.child
31         z.child = y
32         z.degree += 1
33
34     def heap_merge(self, h1, h2):
35         node = None
36         p = None
37         p1 = h1.head
38         p2 = h2.head
39
40         if p1 is None:
41             return h2.head
42         if p2 is None:
```

```
43              return h1.head
44
45          if p1.degree < p2.degree:
46              p = p1
47              p1 = p1.sibling
48          else:
49              p = p2
50              p2 = p2.sibling
51          node = p
52
53          while p1 and p2:
54              if p1.degree < p2.degree:
55                  p.sibling = p1
56                  p1 = p1.sibling
57              else:
58                  p.sibling = p2
59                  p2 = p2.sibling
60              p = p.sibling
61
62          if p2:
63              p.sibling = p2
64          else:
65              p.sibling = p1
66          return node
67
68      def union(self, h1, h2):
69          h = self.make_heap()
70          h.head = self.heap_merge(h1, h2)
71          #free the object h1 and h2 but not the lists they point to
72          del h1
73          del h2
74
75          if h.head is None:
76              return h
77
78          prev_x = None
79          x = h.head
80          next_x = x.sibling
81
82          while next_x is not None:
83              if (x.degree != next_x.degree) or (next_x.sibling is
                    not None and next_x.sibling.degree == x.degree):
84                  prev_x = x                          #case 1 and 2
85                  x = next_x                          #case 1 and 2
86              elif x.key <= next_x.key:
```

```
87                  x.sibling = next_x.sibling      #case 3
88                  h.link(next_x, x)               #case 3
89              else:
90                  if prev_x is None:
91                      h.head = next_x             #case 4
92                  else:
93                      prev_x.sibling = next_x     #case 4
94                  h.link(x, next_x)               #case 4
95              next_x = x.sibling                  #case 4
96          return h
97
98      def heap_insert(self, x):
99          h = self.make_heap()
100         x.p = None
101         x.child = None
102         x.sibling = None
103         x.degree = 0
104         h.head = x
105         heap = self.union(self, h)
106         return heap
107
108     def insert(self, key):
109         return self.heap_insert(Node(key))
110
111     def extract_min(self):
112         #find the root x with the minimum key in the root list of
                heap
113         p = self.head
114         x = None
115         p_prev, x_prev = None, None
116
117         if p is None:
118             return p
119         x = p
120         mini = p.key
121         p_prev = p
122         p = p.sibling
123         while p is not None:
124             if p.key < mini:
125                 x_prev = p_prev
126                 x = p
127                 mini = p.key
128             p_prev = p
129             p = p.sibling
130         if x == self.head:
```

```
131             self.head = x.sibling
132         elif x.sibling is None:
133             x_prev.sibling = None
134         else:
135             x_prev.sibling = x.sibling
136         child_x = x.child
137 
138         #if the minimum node has no child
139         if child_x is not None:
140             """if the node has subtree, then insert them into a
                    new heap,
141                  and union this new heap with old"""
142             h = self.make_heap()
143             child_x.p = None
144             h.head = child_x
145             p = child_x.sibling
146             child_x.sibling = None
147             while p is not None:
148                 p_prev = p
149                 p = p.sibling
150                 p_prev.sibling = h.head
151                 h.head = p_prev
152                 p_prev.p = None
153             self = self.union(self, h)
154         return self
155 
156     def decrease_key(self, x, k):
157         if k > x.key:
158             print("new key is greater than current key")
159             return
160 
161         x.key = k
162         y = x
163         z = y.p
164 
165         while z is not None and y.key < z.key:
166             #do exchange
167             #if y and z have satellite fields, exchange them, too
168             y.key = z.key
169             z.key = k
170             y = z
171             z = y.p
172 
173     def search(self, k):
174         x = self.head
```

```
175             while x is not None:
176                 if x.key == k:
177                     return x
178                 else:
179                     if x.key < k and x.child is not None:
180                         x = x.child
181                     elif x.key > k or x.child is None:
182                         while x.sibling is None:
183                             x = x.p
184                             if x is None:
185                                 return None
186                         x = x.sibling
187             return None
188 
189     def delete(self, x):
190         self.decrease_key(x, -float('inf'))
191         return self.extract_min()
192 
193 def test():
194     print("Binomial Heap test:\n")
195     #1. make heap test
196     print("1. make heap test")
197     heap = binomialHeap().make_heap()
198     if heap:
199         print("make heap successfully\n")
200 
201     #2. insert test
202     print("2. insert test")
203     heap = heap.insert(5)
204     heap = heap.insert(8)
205     heap = heap.insert(2)
206     heap = heap.insert(7)
207     heap = heap.insert(6)
208     heap = heap.insert(9)
209     heap = heap.insert(4)
210     if heap.head is not None:
211         print("insert to heap successfully\n")
212 
213     #3. search key test
214     print("3. search key test")
215     key = 2
216     print("find key", key)
217     node = heap.search(key)
218     if node is not None:
219         print(node.key, "is in the binomial heap")
```

```
220             print("search key successfully\n")
221         else:
222             print("Cannot find the key", key, "\n")
223 
224         #4. minimum key test
225         print("4. minimum key test")
226         print("The minimum:", heap.minimum().key)
227         print("minimum key successfully\n")
228 
229         #5. extract-min test
230         print("5. extract-min test")
231         heap = heap.extract_min()
232         print("After extract-min, the minimum:", heap.minimum().key)
233         node = heap.search(key)
234         if node is None:
235             print("After extract-min, the old minimum", key, "is not
                    in the heap")
236             print("exctract-min successfully\n")
237 
238         #6. decrease key test
239         print("6. decrease key test")
240         key = 9
241         decrease = 2
242         print("we are going to decrease the key", key, "to", decrease)
243         node = heap.search(key)
244         #make sure the key we want to decrease exist
245         if node is not None:
246             print(node.key, "is in the binomial heap")
247             heap.decrease_key(node, decrease)
248         else:
249             print("Cannot find the key", key)
250         #check the update value exist
251         node1 = heap.search(key)
252         node2 = heap.search(decrease)
253         if node1 is None and node2 is not None:
254             print("after decrease key,", node2.key, "is in the
                    binomial heap and", key, "is not in the binomial heap")
255             print("decrease key successfully\n")
256         else:
257             print("decrease key is not successfully\n")
258 
259         #7. delete key test
260         print("7. delete key test")
261         print("before delete:")
262         delete = 7
```

```
263         node = heap.search(delete)
264         if node is not None:
265             print(delete, "is in the binomial heap")
266             heap.delete(node)
267         else:
268             print(delete, "is not in the binomial heap")
269         print("after delete:")
270         node = heap.search(delete)
271         if node is None:
272             print(delete, "is not in the binomial heap")
273             print("delete successfully\n")
274         else:
275             print("delete is not successfully\n")
276 
277 if __name__ == '__main__':
278     test()
```