UM–SJTU Joint Institute

Data Structures and Algorithms

(VE281)

Homework 3

Name: Ji Xingyou ID: 515370910197
Date: 6 November 2017

Contents

1 Theoretical Data

As is discussed in the class, we can get the following table summarizing the time complexity for each priority queues.

1.1 binary_heap

enqueue	$O(logN)$
dequeue_min	$O(logN)$
get_min	$O(1)$

Table 1: Time complexity of binary_heap.

1.2 unsorted_heap

enqueue	$O(1)$
dequeue_min	$O(N)$
get_min	$O(N)$

Table 2: Time complexity of unsorted_heap.

1.3 fib_heap

enqueue	$O(1)$
dequeue_min	$O(logN)$
get_min	$O(1)$

Table 3: Time complexity of fib_heap.

In this report, we will first implement all these three priority queues, and then test the run time for each of them to see whether the above table makes sense.

The implementation of the priority queues is attached in the appendix.

2 Result Analysis

After finishing implementing the above three priority queues, I wrote another program to test the run time of each priority queues. In this program, I set two clocks, noting the starting and finishing instance. To avoid uncertainty in the data, I wrote a while loop to run the program 10 times so that I could get the average value.

The grid size I chose is 16, 25, 100, 225, 400, 625, 900, 1600, 2500, 3600, 4900, 6400, 8100, 10000.

The run time data for each priority queues is listed in table 2.

Grid size	binary_heap	unsorted_heap	fib_heap
16	281	337	332
25	376	343	397
100	416	530	1386
225	493	691	2066
400	648	800	4493
625	793	926	8120
900	838	1175	11461
1600	867	1372	32847
2500	1417	2270	68225
3600	2180	3107	106249
4900	2329	4587	168253
6400	2364	5894	292624
8100	3397	8836	356320
10000	5692	9915	492132
22500	5849	26533	1891842
40000	9129	59409	3676906

Table 4: Run time of priority queues

The run time comparison is shown in figure 1.

Combining the table and figure above, we can conclude the following points.

1. For each priority queue, the run time increases as the size of the grid increases.

2. For different grid size, binary_heap has the best the performance among all while the fib_heap has the worst.

However, for the reason that my fib_heap is not well implemented, which I myself does not know where goes wrong, this does not cover the real situation.

Below I will illustrate the real case, which can not be derived from my figure. The real cases are from the discussion with my classmates. All the conversation is conducted in a normal way which does not involve an Honor Code violation. Reference is available if requested.

1. When the grid size is small (less than 100), unsorted_heap has a better performance than fib_heap.

2. When the array size becomes gradually larger (greater than 100), fib_heap has a better performance than unsorted_heap.

3. No matter what the grid size is, binary_heap always has the best performance among these three.

This inspires me that in future learning, when dealing with priority queues, I should make a wise choice on which form to apply. For example, when the grid size is small, I should use binary_heap or unsorted_heap. When the grid size is large, I should choose Fibonacci_heap or binary_heap.

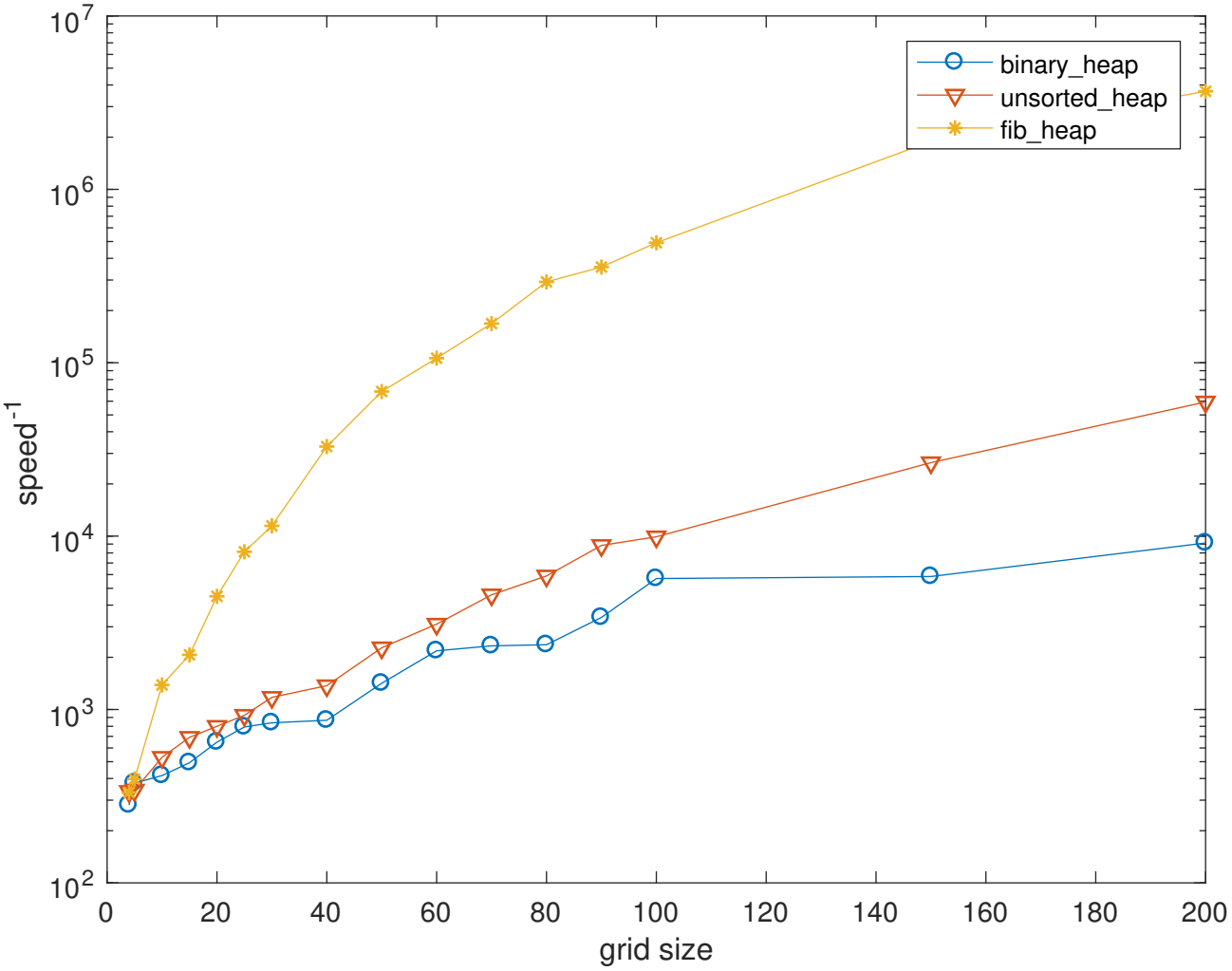

Figure 1: Run time comparison

3 Appendix

3.1 Priority queues

3.1.1 main.cpp

```
1 //
2 //  main.cpp
3 //  project3
4 //
```

```
5  //  Created by           on 2017/10/27.
6  //  Copyright    2017             . All rights reserved.
7  //
8
9  #include <iostream>
10 #include <getopt.h>
11 #include "priority_queue.h"
12 #include "binary_heap.h"
13 #include "fib_heap.h"
14 #include "unsorted_heap.h"
15
16 using namespace std;
17
18 class point {
19 public:
20     int x;
21     int y;
22     int cellweight=0;
23     int pathcost=0;
24     bool reached=false;
25     point *predecessor=NULL;
26     friend bool operator==(const point &p1,const point &p2)
27     {
28         return (p1.x==p2.x&&p1.y==p2.y&&p1.cellweight==p2.cellweight&&p1.pathcost==p2.pathcos
29     }
30     friend bool operator<(const point &p1,const point &p2)
31     {
32         return p1.pathcost<p2.pathcost;
```

```
33      }
34      friend bool operator>(const point &p1,const point &p2)
35      {
36          return p1.pathcost>p2.pathcost;
37      }
38      struct compare_t
39      {
40          bool operator()(const point &a, const point &b) const
41          {
42              return (a.pathcost<b.pathcost)||((a.pathcost==b.pathcost)&&(a.x<b.x))||((a.pathco
43          }
44      };
45  };
46
47  void trace_back_path(point *p);
48
49  int main(int argc,char* argv[])
50  {
51      int width,height=0;
52      cin>>width;
53      cin>>height;
54      int start_point_x,start_point_y,end_point_x,end_point_y;
55      cin>>start_point_x>>start_point_y>>end_point_x>>end_point_y;
56      point p_array[height][width];
57      for(int h=0;h<height;++h)
58      {
59          for(int w=0;w<width;++w)
60          {
```

```
61              cin>>p_array[h][w].cellweight;
62          }
63      }
64      for(int h=0;h<height;++h)
65      {
66          for(int w=0;w<width;++w)
67          {
68              p_array[h][w].x=w;
69              p_array[h][w].y=h;
70              p_array[h][w].pathcost=p_array[h][w].cellweight;
71          }
72      }
73      bool verbose=false;
74      string mode;
75      while(true)
76      {
77          static struct option long_options[]=
78          {
79              {"verbose",no_argument,NULL,'v'},
80              {"implementation",required_argument,NULL,'i'},
81              {0, 0, 0, 0}
82          };
83          int c=getopt_long(argc,argv,"vi:",long_options,NULL);
84          if(c==-1)
85          {
86              break;
87          }
88          if(c=='v')
```

```
89          {
90              verbose=true;
91          }
92          if(c=='i')
93          {
94              mode=optarg;
95          }
96      }
97      priority_queue<point,point::compare_t> *PQ;
98      if(mode=="BINARY")
99      {
100         PQ=new binary_heap<point,point::compare_t>();
101     }
102     else if(mode=="UNSORTED")
103     {
104         PQ=new unsorted_heap<point,point::compare_t>();
105     }
106     else if(mode=="FIBONACCI")
107     {
108         PQ=new fib_heap<point,point::compare_t>();
109     }
110     else
111     {
112         exit(0);
113     }
114     p_array[start_point_y][start_point_x].reached=true;
115     PQ->enqueue(p_array[start_point_y][start_point_x]);
116     int Step=0;
```

```
117     while(PQ->empty()==false)
118     {
119         point C=PQ->dequeue_min();
120         if(verbose==true)
121         {
122             cout<<"Step "<<Step<<endl;
123             cout<<"Choose cell ("<<p_array[C.y][C.x].x<<", "<<p_array[C.y][C.x].y<<") with ac
124         }
125         Step++;
126         int delta_x[4]={1,0,-1,0};
127         int delta_y[4]={0,1,0,-1};
128         for(int i=0;i<4;++i)
129         {
130             int N_x=p_array[C.y][C.x].x+delta_x[i];
131             int N_y=p_array[C.y][C.x].y+delta_y[i];
132             if(N_x<0||N_x>width-1||N_y<0||N_y>height-1||p_array[N_y][N_x].reached==true)
133             {
134                 continue;
135             }
136             p_array[N_y][N_x].pathcost=p_array[C.y][C.x].pathcost+p_array[N_y][N_x].cellweigh
137             p_array[N_y][N_x].reached=true;
138             p_array[N_y][N_x].predecessor=&p_array[C.y][C.x];
139             if(p_array[end_point_y][end_point_x].x==p_array[N_y][N_x].x&&p_array[end_point_y]
140             {
141                 if(verbose==true)
142                 {
143                     cout<<"Cell ("<<p_array[N_y][N_x].x<<", "<<p_array[N_y][N_x].y<<") with a
144                 }
```

```
145                 cout<<"The shortest path from ("<<p_array[start_point_y][start_point_x].x<<",
146                 cout<<"Path:"<<endl;
147                 trace_back_path(&p_array[end_point_y][end_point_x]);
148                 delete PQ;
149                 return 0;
150             }
151             else
152             {
153                 PQ->enqueue(p_array[N_y][N_x]);
154                 if(verbose==true)
155                 {
156                     cout<<"Cell ("<<p_array[N_y][N_x].x<<", "<<p_array[N_y][N_x].y<<") with a
157                 }
158             }
159         }
160     }
161     delete PQ;
162     return 0;
163 }
164
165 void trace_back_path(point *p)
166 {
167     if(p!=NULL)
168     {
169         trace_back_path(p->predecessor);
170         cout<<"("<<p->x<<", "<<p->y<<")"<<endl;
171     }
172     else
```

```
173     {
174         return;
175     }
176 }
```

3.1.2 binary_heap.h

```
1  #ifndef BINARY_HEAP_H
2  #define BINARY_HEAP_H
3
4  #include <algorithm>
5  #include "priority_queue.h"
6
7  // OVERVIEW: A specialized version of the 'heap' ADT implemented as a binary
8  //           heap.
9  template<typename TYPE, typename COMP = std::less<TYPE>>
10 class binary_heap:public priority_queue<TYPE,COMP>
11 {
12 public:
13     typedef unsigned size_type;
14
15     // EFFECTS: Construct an empty heap with an optional comparison functor.
16     //          See test_heap.cpp for more details on functor.
17     // MODIFIES: this
18     // RUNTIME: O(1)
19     binary_heap(COMP comp=COMP());
20
21     // EFFECTS: Add a new element to the heap.
22     // MODIFIES: this
```

```
23      // RUNTIME: O(log(n))
24      virtual void enqueue(const TYPE&val);
25
26      // EFFECTS: Remove and return the smallest element from the heap.
27      // REQUIRES: The heap is not empty.
28      // MODIFIES: this
29      // RUNTIME: O(log(n))
30      virtual TYPE dequeue_min();
31
32      // EFFECTS: Return the smallest element of the heap.
33      // REQUIRES: The heap is not empty.
34      // RUNTIME: O(1)
35      virtual const TYPE&get_min() const;
36
37      // EFFECTS: Get the number of elements in the heap.
38      // RUNTIME: O(1)
39      virtual size_type size() const;
40
41      // EFFECTS: Return true if the heap is empty.
42      // RUNTIME: O(1)
43      virtual bool empty() const;
44
45  private:
46      // Note: This vector *must* be used in your heap implementation.
47      std::vector<TYPE> data;
48      // Note: compare is a functor object
49      COMP compare;
50
```

```
51 private:
52     // Add any additional member functions or data you require here.
53     virtual void percolateUp(int id);
54 
55     virtual void percolateDown(int id);
56 
57 };
58 
59 template<typename TYPE,typename COMP>
60 void binary_heap<TYPE,COMP>::percolateUp(int id)
61 {
62     while(id>1&&compare(data[id],data[id/2]))
63     {
64         std::swap(data[id/2],data[id]);
65         id=id/2;
66     }
67 }
68 
69 template<typename TYPE,typename COMP>
70 void binary_heap<TYPE,COMP>::percolateDown(int id)
71 {
72     int j;
73     int size=this->size();
74     for(j=2*id;j<=size;j=2*id)
75     {
76         if(j<size&&compare(data[j+1],data[j]))
77         {
78             j++;
```

```
79          }
80          if(compare(data[id],data[j]))
81          {
82              break;
83          }
84          std::swap(data[id],data[j]);
85          id=j;
86      }
87  };
88
89  template<typename TYPE,typename COMP>
90  binary_heap<TYPE,COMP>::binary_heap(COMP comp)
91  {
92      compare=comp;
93      // Fill in the remaining lines if you need.
94      data.push_back(TYPE());
95  }
96
97  template<typename TYPE,typename COMP>
98  void binary_heap<TYPE,COMP>::enqueue(const TYPE&val)
99  {
100     // Fill in the body.
101     data.push_back(val);
102     int id=this->size();
103     this->percolateUp(id);
104 }
105
106 template<typename TYPE,typename COMP>
```

```
107 TYPE binary_heap<TYPE,COMP>::dequeue_min()
108 {
109     // Fill in the body.
110     if(this->empty())
111     {
112         return data[0];
113     }
114     TYPE val=data[1];
115     data[1]=data.back();
116     data.pop_back();
117     this->percolateDown(1);
118     return val;
119 }
120
121 template<typename TYPE,typename COMP>
122 const TYPE&binary_heap<TYPE,COMP>::get_min() const
123 {
124     // Fill in the body.
125     if(this->empty())
126     {
127         return data[0];
128     }
129     else
130     {
131         return data[1];
132     }
133 }
134
```

```
135 template<typename TYPE,typename COMP>
136 bool binary_heap<TYPE,COMP>::empty() const
137 {
138     // Fill in the body.
139     return this->size()==0;
140 }
141
142 template<typename TYPE,typename COMP>
143 unsigned binary_heap<TYPE,COMP>::size() const
144 {
145     // Fill in the body.
146     return data.size()-1;
147 }
148
149 #endif //BINARY_HEAP_H
```

3.1.3 unsorted_heap.h

```
1  #ifndef UNSORTED_HEAP_H
2  #define UNSORTED_HEAP_H
3
4  #include <algorithm>
5  #include "priority_queue.h"
6
7  // OVERVIEW: A specialized version of the 'heap' ADT that is implemented with
8  //           an underlying unordered array-based container. Every time a min
9  //           is required, a linear search is performed.
10 template<typename TYPE,typename COMP = std::less<TYPE>>
11 class unsorted_heap:public priority_queue<TYPE,COMP>
```

```
12 {
13 public:
14     typedef unsigned size_type;
15 
16     // EFFECTS: Construct an empty heap with an optional comparison functor.
17     //          See test_heap.cpp for more details on functor.
18     // MODIFIES: this
19     // RUNTIME: O(1)
20     unsorted_heap(COMP comp=COMP());
21 
22     // EFFECTS: Add a new element to the heap.
23     // MODIFIES: this
24     // RUNTIME: O(1)
25     virtual void enqueue(const TYPE&val);
26 
27     // EFFECTS: Remove and return the smallest element from the heap.
28     // REQUIRES: The heap is not empty.
29     // MODIFIES: this
30     // RUNTIME: O(n)
31     virtual TYPE dequeue_min();
32 
33     // EFFECTS: Return the smallest element of the heap.
34     // REQUIRES: The heap is not empty.
35     // RUNTIME: O(n)
36     virtual const TYPE&get_min() const;
37 
38     // EFFECTS: Get the number of elements in the heap.
39     // RUNTIME: O(1)
```

```
40      virtual size_type size() const;
41
42      // EFFECTS: Return true if the heap is empty.
43      // RUNTIME: O(1)
44      virtual bool empty() const;
45
46  private:
47      // Note: This vector *must* be used in your heap implementation.
48      std::vector<TYPE> data;
49      // Note: compare is a functor object
50      COMP compare;
51  private:
52      // Add any additional member functions or data you require here.
53      TYPE empty_element=TYPE();
54  };
55
56  template<typename TYPE,typename COMP>
57  unsorted_heap<TYPE,COMP>::unsorted_heap(COMP comp)
58  {
59      compare=comp;
60      // Fill in the remaining lines if you need.
61  }
62
63  template<typename TYPE,typename COMP>
64  void unsorted_heap<TYPE,COMP>::enqueue(const TYPE&val)
65  {
66      // Fill in the body.
67      data.push_back(val);
```

```
68 }
69
70 template<typename TYPE,typename COMP>
71 TYPE unsorted_heap<TYPE,COMP>::dequeue_min()
72 {
73     // Fill in the body.
74     if(this->empty())
75     {
76         return empty_element;
77     }
78     auto it=data.begin();
79     TYPE min=data[0];
80     TYPE max=data[0];
81     for(it=data.begin();it!=data.end();++it)
82     {
83         if(compare((*it),min))
84         {
85             min=*it;
86         }
87     }
88     for(it=data.begin();it!=data.end();++it)
89     {
90         if(compare(max,(*it)))
91         {
92             max=(*it);
93         }
94     }
95     if(compare(min,max)==false)
```

```
96      {
97          min=max;
98      }
99      auto key=it;
100     for(it=data.begin();it!=data.end();++it)
101     {
102         if((*it)==min)
103         {
104             key=it;
105             break;
106         }
107     }
108     data.erase(key);
109     return min;
110 }
111
112 template<typename TYPE,typename COMP>
113 const TYPE&unsorted_heap<TYPE,COMP>::get_min() const
114 {
115     // Fill in the body.
116     if(this->empty())
117     {
118         return data[0];
119     }
120     auto it=data.begin();
121     TYPE min=data[0];
122     TYPE max=data[0];
123     for(it=data.begin();it!=data.end();++it)
```

```
124     {
125         if(compare((*it),min))
126         {
127             min=(*it);
128         }
129     }
130     for(it=data.begin();it!=data.end();++it)
131     {
132         if(compare(max,(*it)))
133         {
134             max=(*it);
135         }
136     }
137     if(compare(min,max)==false)
138     {
139         min=max;
140     }
141     auto key=it;
142     for(it=data.begin();it!=data.end();++it)
143     {
144         if((*it)==min)
145         {
146             key=it;
147             break;
148         }
149     }
150     return *key;
151 }
```

```
152
153 template<typename TYPE,typename COMP>
154 bool unsorted_heap<TYPE,COMP>::empty() const
155 {
156     // Fill in the body.
157     return data.empty();
158 }
159
160 template<typename TYPE,typename COMP>
161 unsigned unsorted_heap<TYPE,COMP>::size() const
162 {
163     // Fill in the body.
164     return data.size();
165 }
166
167 #endif //UNSORTED_HEAP_H
```

3.1.4 fib_heap.h

```
1  #ifndef FIB_HEAP_H
2  #define FIB_HEAP_H
3
4  #include <algorithm>
5  #include <cmath>
6  #include "priority_queue.h"
7  #include <list>
8
9  // OVERVIEW: A specialized version of the 'heap' ADT implemented as a
10 //           Fibonacci heap.
```

```
11 template<typename TYPE,typename COMP = std::less<TYPE>>
12 class fib_heap:public priority_queue<TYPE,COMP>
13 {
14 public:
15     typedef unsigned size_type;
16
17     // EFFECTS: Construct an empty heap with an optional comparison functor.
18     //          See test_heap.cpp for more details on functor.
19     // MODIFIES: this
20     // RUNTIME: O(1)
21     fib_heap(COMP comp=COMP());
22
23     // EFFECTS: Deconstruct the heap with no memory leak.
24     // MODIFIES: this
25     // RUNTIME: O(n)
26     ~fib_heap();
27
28     // EFFECTS: Add a new element to the heap.
29     // MODIFIES: this
30     // RUNTIME: O(1)
31     virtual void enqueue(const TYPE&val);
32
33     // EFFECTS: Remove and return the smallest element from the heap.
34     // REQUIRES: The heap is not empty.
35     // MODIFIES: this
36     // RUNTIME: Amortized O(log(n))
37     virtual TYPE dequeue_min();
38
```

```
39      // EFFECTS: Return the smallest element of the heap.
40      // REQUIRES: The heap is not empty.
41      // RUNTIME: O(1)
42      virtual const TYPE&get_min() const;
43
44      // EFFECTS: Get the number of elements in the heap.
45      // RUNTIME: O(1)
46      virtual size_type size() const;
47
48      // EFFECTS: Return true if the heap is empty.
49      // RUNTIME: O(1)
50      virtual bool empty() const;
51
52  private:
53      // Note: compare is a functor object
54      COMP compare;
55
56  private:
57      // Add any additional member functions or data you require here.
58      // You may want to define a strcut/class to represent nodes in the heap and a
59      // pointer to the min node in the heap.
60      struct fib_node
61      {
62          TYPE val;
63          typename std::list<fib_node> child;
64          int degree=0;
65      };
66      typename std::list<fib_node> root;
```

```
67      typename std::list<fib_node>::iterator H_min;
68      int H_n=0;
69      TYPE empty_element=TYPE();
70  };
71
72  // Add the definitions of the member functions here. Please refer to
73  // binary_heap.h for the syntax.
74
75  template<typename TYPE,typename COMP>
76  fib_heap<TYPE,COMP>::fib_heap(COMP comp)
77  {
78      compare=comp;
79      // Fill in the remaining lines if you need.
80      H_min=root.begin();
81      H_n=0;
82  }
83
84  template<typename TYPE,typename COMP>
85  fib_heap<TYPE,COMP>::~fib_heap()
86  {
87      typename std::list<fib_node>::iterator it;
88      for(it=root.begin();it!=root.end();++it)
89      {
90          root.erase(it);
91      }
92  }
93
94  template<typename TYPE,typename COMP>
```

```
95  void fib_heap<TYPE,COMP>::enqueue(const TYPE&val)
96  {
97      fib_node n;
98      n.val=val;
99      n.degree=0;
100     if(root.empty()==true)
101     {
102         root.push_back(n);
103         H_min=root.begin();
104     }
105     else
106     {
107         if(compare(val,(*H_min).val))
108         {
109             auto it=root.insert(root.end(),n);
110             H_min=it;
111         }
112         else
113         {
114             root.insert(root.end(),n);
115         }
116     }
117     H_n++;
118 };
119
120 template<typename TYPE,typename COMP>
121 TYPE fib_heap<TYPE,COMP>::dequeue_min()
122 {
```

```
123     if(root.empty()==true)
124     {
125         return empty_element;
126     }
127
128 //     std::cout<<std::endl;
129
130     fib_node z;
131     z=*H_min;
132     typename std::list<fib_node>::iterator temp;
133     if(H_min!=root.end())
134     {
135         temp=z.child.begin();
136         while(temp!=z.child.end())
137         {
138             root.push_back(*temp);
139             temp=z.child.erase(temp);
140         }
141         H_min=root.erase(H_min);
142         H_n--;
143         if(H_n==0)
144         {
145             H_min=root.end();
146         }
147         else
148         {
149             int size=int((log(H_n))/(log((1+sqrt(5))/2)))+1;
150             typename std::list<fib_node>::iterator A[size];
```

```
151             for(int i=0;i<size;++i)
152             {
153                 A[i]=root.end();
154             }
155             typename std::list<fib_node>::iterator x;
156             typename std::list<fib_node>::iterator y;
157             int d=0;
158             typename std::list<fib_node>::iterator it;
159             for(it=root.begin();it!=root.end();++it)
160             {
161
162 //                  std::cout<<"item in A ";
163 //                  for(int i=0;i<size;++i)
164 //                  {
165 //                      if(A[i]==root.end()){
166 //                          std::cout<<"NULL"<<" ";
167 //                      }
168 //                      else{
169 //                          std::cout<<(*A[i]).val<<" ";
170 //                      }
171 //                  }
172 //                  std::cout<<std::endl;
173
174                 d=(*it).degree;
175                 while(A[d]!=root.end())
176                 {
177                     y=A[d];
178
```

```
179 //                      std::cout<<"item in A ";
180 //                      for(int i=0;i<size;++i)
181 //                      {
182 //                          if(A[i]==root.end()){
183 //                              std::cout<<"NULL"<<" ";
184 //                          }
185 //                          else{
186 //                              std::cout<<(*A[i]).val<<" ";
187 //                          }
188 //                      }
189 //                      std::cout<<std::endl;
190
191                       if(compare((*y).val,(*it).val))
192                       {
193                           root.insert(y,*it);
194                           root.insert(it,*y);
195                           it=root.erase(it);
196                           y=root.erase(y);
197                           it--;
198                           y--;
199                       }
200
201 //                      std::cout<<"item in A ";
202 //                      for(int i=0;i<size;++i)
203 //                      {
204 //                          if(A[i]==root.end()){
205 //                              std::cout<<"NULL"<<" ";
206 //                          }
```

```
207 //                          else{
208 //                              std::cout<<(*A[i]).val<<" ";
209 //                          }
210 //                      }
211 //                      std::cout<<std::endl;
212
213                       (*it).child.push_back((*y));
214                       y=root.erase(y);
215                       (*it).degree++;
216                       A[d]=root.end();
217
218 //                      std::cout<<"item in A ";
219 //                      for(int i=0;i<size;++i)
220 //                      {
221 //                          if(A[i]==root.end()){
222 //                              std::cout<<"NULL"<<" ";
223 //                          }
224 //                          else{
225 //                              std::cout<<(*A[i]).val<<" ";
226 //                          }
227 //                      }
228 //                      std::cout<<std::endl;
229 //                      typename std::list<fib_node>::iterator ttt;
230 //                      int testt=0;
231 //                      for(ttt=root.begin();ttt!=root.end();++ttt)
232 //                      {
233 //                          printf("The %d item in rootlost is %d\n",testt,(*ttt).val);
234 //                          typename std::list<fib_node>::iterator tttt;
```

```
235 //                          for(tttt=(*ttt).child.begin();tttt!=(*ttt).child.end();++tttt)
236 //                          {
237 //                              std::cout<<(*tttt).val<<" ";
238 //                          }
239 //                          testt++;
240 //                          std::cout<<std::endl;
241 //                      }
242 //                      std::cout<<"END"<<std::endl;
243
244                         d++;
245                     }
246                     A[d]=it;
247                 }
248                 H_min=root.end();
249                 for(int i=0;i<size;++i)
250                 {
251                     if(A[i]!=root.end())
252                     {
253                         if(H_min==root.end())
254                         {
255                             H_min=A[i];
256                         }
257                         else
258                         {
259                             if(compare((*A[i]).val,(*H_min).val))
260                             {
261                                 H_min=A[i];
262                             }
```

```
263                     }
264                 }
265             }
266         }
267     }
268     return z.val;
269 };
270 
271 template<typename TYPE,typename COMP>
272 const TYPE&fib_heap<TYPE,COMP>::get_min() const
273 {
274     if(this->empty())
275     {
276         return empty_element;
277     }
278     else
279     {
280         return (*H_min).val;
281     }
282 };
283 
284 template<typename TYPE,typename COMP>
285 bool fib_heap<TYPE,COMP>::empty() const
286 {
287     return this->size()==0;
288 };
289 
290 template<typename TYPE,typename COMP>
```

```
291 unsigned fib_heap<TYPE,COMP>::size() const
292 {
293     return this->H_n;
294 };
295
296 #endif //FIB_HEAP_H
```