Ve 281

P3 Report

By
Dong Jing 515370910182
Nonmember 7, 2017

1. Object

I. Learn three different priority queue data structures, binary heap, unsorted heap and Fibonacci heap;
II. Implement three different data structures;
III. Use these data structures to solve the shortest weight path problem;
IV. Compare the performance of different data structures with different input size.

2. Design

Note: all program are executed on Ubuntu (64-bit) with memory equal to 2048 MB on Oracle VM VirtualBox 5.1.22 r115126.

I select ten different values of width, 5, 10, 20, 50, 100, 200, 300, 400, 500 and 600. The value of height is equal to the value of width. Then I use mrand48() to generate random numbers which represent the weight of every node. To make the weight correct, I calculate the random number mod 100 to get a nonnegative random number. For each input size, I generate 5 different graphs to find the path with minimal cost. For each graph, the start point is (0,0) while the end point is (n-1,n-1), where n is the value of width.

3. Result

Table 1 shows are data we recorded. The unit of all data is CLOCK_PER_SEC.

Input Size	Binary	Unsorted	Fibonacci
5	3	2	6
10	8	6	13
20	76	106	99
50	371	2469	533
100	2188	51102	2110
200	15128	2305501	15618
300	17205	3984744	16354
400	43336	15747025	39359
500	49486	32454825	45816
600	131586	139405585	98839

Table 1: Runtime comparison of three priority queue data structures

With this data, I use Matlab to plot three curves representing the runtime of three data structures on different input size.

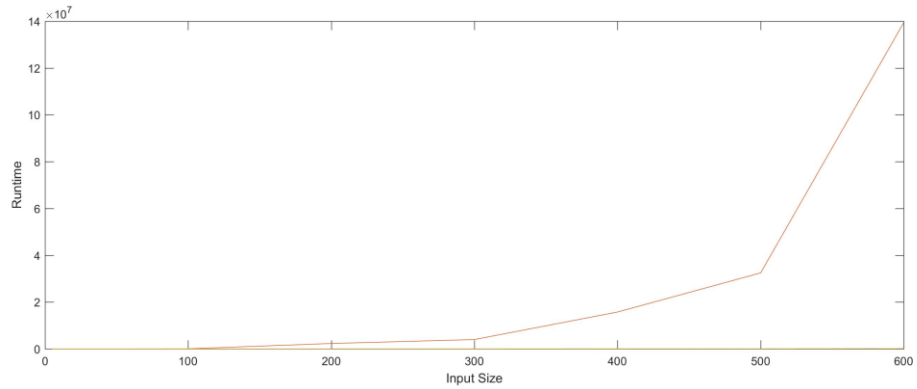

Figure 1. Comparison of three data structures

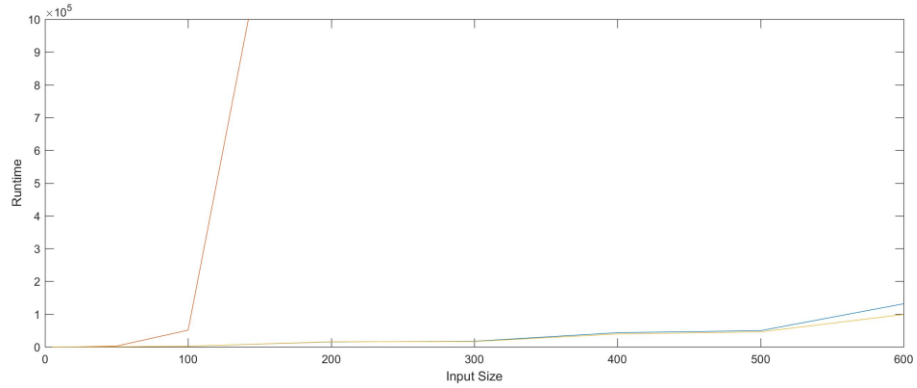

Figure 2. Comparison between binary heap and Fibonacci heap on large size

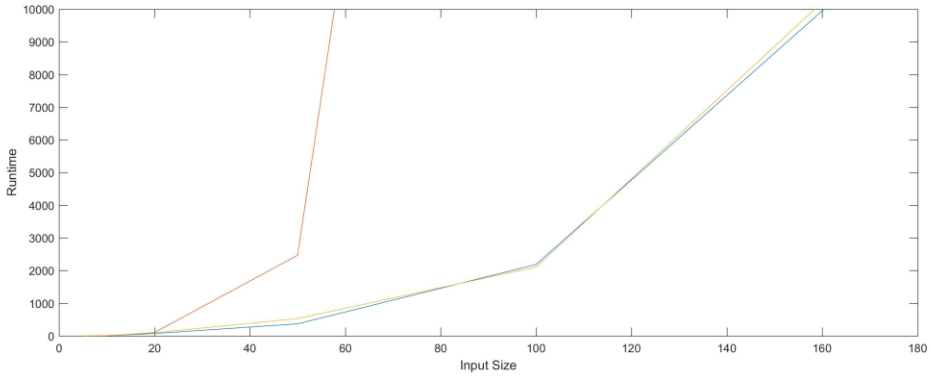

Figure 3. Comparison between binary heap and Fibonacci heap on small size

In all three figures, blue curve represents the runtime of binary heap. Orange curve represents the runtime of unsorted heap. And yellow curve is on behalf of the runtime of Fibonacci heap.

As we can see, the runtime of unsorted heap increases quick as the input size increases. It is much slower than other two data structures when the input size is greater than 20*20. And the blue curve representing the runtime of binary heap is close to the yellow curve which represents the runtime of Fibonacci heap. When the input size is small, binary heap is faster than Fibonacci heap. When the input size is large, binary heap is slower than Fibonacci heap. The difference is not big for the input size I tested. But I observe the tendency and I can conclude that when the input

size goes larger and larger, Fibonacci heap will become faster and faster compared with binary heap, which satisfies the amortized time complexity in theorem. Table 2 shows the time complexity of different operands of different data structures.

Time Complexity	Binary heap	Unsorted heap	Fibonacci heap
enqueue	O(log(n))	O(1)	O(1)
dequeue_min	O(log(n))	O(n)	O(log(n))

Table 2. Time complexity of operands needed

4. Discussion and Conclusion

From three figures, we can find when the input size is small unsorted heap is the fastest, because the enqueue operands is fast and when the input size is small the difference between log(n) and n is small. As the input size goes large, unsorted heap is much slower than other two data structures due to the large time used to dequeue_min operand.

When the input size is small, we find binary heap is a little faster than Fibonacci heap. I think it is because the complex operations on the node in Fibonacci heap. Although the time complexity of enqueue operand of Fibonacci heap is O(1), actually it need to build a new node and link the node into the root list, which is complex. When the input size becomes large, the runtime of Fibonacci heap is a little faster than binary heap. And I think if the input size goes larger, like 1000*1000, the difference between the runtime of Fibonacci heap and the runtime of binary heap will also become large.

In conclusion, I think it is improper to use unsorted heap to solve shortest weight path problem. Both binary heap and Fibonacci heap is suitable for this problem. In this problem, the advantage of binary heap is that it is easy to implement in code. I think the biggest advantage of Fibonacci heap does not show in this problem because it just needs two operands of Fibonacci heap. The advantage will become more obvious if we use more operands of Fibonacci heap, such as Union and Decrease-Key. Fibonacci heap is useful for Dijkstra's algorithm for computing the shortest path between two nodes in a graph and matching of weight bipartite graph. The most important problem of Fibonacci heap is the difficulty of programming it due to many pointers and complex operands to implement different functions.

In the project, I have a better understanding about priority queue and know more about how different priority queues works. The comparison of performance of three data structures shows the correctness of time complexity. Implementing Fibonacci heap also improves my programming ability.

5. Appendix

time.cpp

```
#pragma GCC optimize(3)
#include <iostream>
#include <fstream>
```

```
#include <sstream>
#include <string>
#include <cstdlib>
#include <climits>
#include <ctime>
#include <cassert>
#include <getopt.h>
#include "binary_heap.h"
#include "priority_queue.h"
#include "unsorted_heap.h"
#include "fib_heap.h"

using namespace std;

struct point
{
	int x;
	int y;
	int key;
	int weight;
	point *prev;
	bool mark;
};

struct compare_t
{
	bool operator()(point *a, point *b) const
	{
		if (a->key<b->key) return true;
	else if (a->key==b->key && a->x<b->x) return true;
	else if (a->key==b->key && a->x==b->x && a->y<b->y) return true;
	else return false;
	}
};

int main(int argc, char *argv[])
{
	int width,height,start_x,start_y,end_x,end_y;
	width=5;
	height=5;
	int t=0;
	int k=0;
	int num=0;
	clock_t time;
```

```
clock_t ave[3];
point *p,*tem;
while (t<10)
{
cout<<"Width is "<<width<<" and height is "<<height<<endl;
point *pos=new point[width*height];
start_x=0;start_y=0;
end_x=width-1;end_y=height-1;
while (k<5){
      for (int j=0;j<height;j++)
            for (int i=0;i<width;i++)
            {
                pos[j*width+i].weight=mrand48()%100;
      pos[j*width+i].mark=false;
           pos[j*width+i].x=i;
           pos[j*width+i].y=j;
           pos[j*width+i].prev=NULL;
            }
      while (num<3){
      pos[start_x+start_y*width].prev=&pos[start_x+start_y*width];
      pos[start_x+start_y*width].key=pos[start_x+start_y*width].weight;
      priority_queue<point*, compare_t> *PQ=NULL;
      if (num==0) PQ=new binary_heap<point*, compare_t>;
      if (num==1) PQ=new unsorted_heap<point*, compare_t>;
      if (num==2) PQ=new fib_heap<point*, compare_t>;
      p=&pos[start_x+start_y*width];
      time=clock();
      PQ->enqueue(p);
      while (PQ->empty()==false)
      {
       p=PQ->dequeue_min();
       if (p->x+1<width && pos[p->x+1+p->y*width].mark==false)
       {
            pos[p->x+1+p->y*width].mark=true;
      pos[p->x+1+p->y*width].prev=p;
      pos[p->x+1+p->y*width].key=p->key+pos[p->x+1+p->y*width].weight;
           tem=&pos[p->x+1+p->y*width];
           if (tem->x==end_x && tem->y==end_y) break;
           else PQ->enqueue(tem);
       }
       if (p->y+1<height && pos[p->x+(p->y+1)*width].mark==false)
       {
            tem=&pos[p->x+(p->y+1)*width];
      tem->mark=true;
```

```
            tem->prev=p;
            tem->key=p->key+tem->weight;
                if (tem->x==end_x && tem->y==end_y) break;
                else PQ->enqueue(tem);
            }
            if (p->x-1>=0 && pos[p->x-1+p->y*width].mark==false)
            {
                tem=&pos[p->x-1+p->y*width];
            tem->mark=true;
            tem->prev=p;
            tem->key=p->key+tem->weight;
                if (tem->x==end_x && tem->y==end_y) break;
                else PQ->enqueue(tem);
            }
            if (p->y-1>=0 && pos[p->x+(p->y-1)*width].mark==false)
            {
                tem=&pos[p->x+(p->y-1)*width];
            tem->mark=true;
            tem->prev=p;
            tem->key=p->key+tem->weight;
                if (tem->x==end_x && tem->y==end_y) break;
                else PQ->enqueue(tem);
            }
        }
        time=clock()-time;
        ave[num]=(k*ave[num]+time)/(k+1);
        num++;
        while (PQ->size()>0) PQ->dequeue_min();
        delete PQ;
        for (int j=0;j<height;j++)
                for (int i=0;i<width;i++)
                {
            pos[j*width+i].mark=false;
                pos[j*width+i].x=i;
                pos[j*width+i].y=j;
                pos[j*width+i].prev=NULL;
            pos[j*width+i].key=0;
                }
        }
        num=0;
        k++;
}
        k=0;
        delete [] pos;
```

```
        cout<<"The average runtime of binary heap is "<<ave[0]<<endl;
        cout<<"The average runtime of unsorted heap is "<<ave[1]<<endl;
        cout<<"The average runtime of fibonacci heap is "<<ave[2]<<endl;
        if (t==0) {width=10;height=10;}
        if (t==1) {width=20;height=20;}
        if (t==2) {width=50;height=50;}
        if (t==3) {width=100;height=100;}
        if (t==4) {width=200;height=200;}
        if (t==5) {width=300;height=300;}
        if (t==6) {width=400;height=400;}
        if (t==7) {width=500;height=500;}
        if (t==8) {width=600;height=600;}
        t++;
        }
}
```

priority_queue.h

```
#ifndef PRIORITY_QUEUE_H
#define PRIORITY_QUEUE_H

#include <functional>
#include <vector>

// OVERVIEW: A simple interface that implements a generic heap.
//                 Runtime specifications assume constant time comparison and
//                 copying. TYPE is the type of the elements stored in the priority
//                 queue. COMP is a functor, which returns the comparison result of
//                 two elements of the type TYPE. See test_heap.cpp for more details
//                 on functor.
template<typename TYPE, typename COMP = std::less<TYPE> >
class priority_queue {
public:
   typedef unsigned size_type;

   virtual ~priority_queue() {}

   // EFFECTS: Add a new element to the heap.
   // MODIFIES: this
   // RUNTIME: O(n) - some implementations *must* have tighter bounds (see
   //                 specialized headers).
   virtual void enqueue(const TYPE &val) = 0;

   // EFFECTS: Remove and return the smallest element from the heap.
   // REQUIRES: The heap is not empty.
```

```
    //                  Note: We will not run tests on your code that would require it
    //                  to dequeue an element when the heap is empty.
    // MODIFIES: this
    // RUNTIME: O(n) - some implementations *must* have tighter bounds (see
    //                 specialized headers).
    virtual TYPE dequeue_min() = 0;

    // EFFECTS: Return the smallest element of the heap.
    // REQUIRES: The heap is not empty.
    // RUNTIME: O(n) - some implementations *must* have tighter bounds (see
    //                 specialized headers).
    virtual const TYPE &get_min() const = 0;

    // EFFECTS: Get the number of elements in the heap.
    // RUNTIME: O(1)
    virtual size_type size() const = 0;

    // EFFECTS: Return true if the heap is empty.
    // RUNTIME: O(1)
    virtual bool empty() const = 0;

};

#endif //PRIORITY_QUEUE_H
```

binary_heap.h

```
#ifndef BINARY_HEAP_H
#define BINARY_HEAP_H

#include <algorithm>
#include "priority_queue.h"

// OVERVIEW: A specialized version of the 'heap' ADT implemented as a binary
//                 heap.
template<typename TYPE, typename COMP = std::less<TYPE> >
class binary_heap: public priority_queue<TYPE, COMP> {
public:
    typedef unsigned size_type;

    // EFFECTS: Construct an empty heap with an optional comparison functor.
    //                 See test_heap.cpp for more details on functor.
    // MODIFIES: this
    // RUNTIME: O(1)
    binary_heap(COMP comp = COMP());
```

```
    // EFFECTS: Add a new element to the heap.
    // MODIFIES: this
    // RUNTIME: O(log(n))
    virtual void enqueue(const TYPE &val);

    // EFFECTS: Remove and return the smallest element from the heap.
    // REQUIRES: The heap is not empty.
    // MODIFIES: this
    // RUNTIME: O(log(n))
    virtual TYPE dequeue_min();

    // EFFECTS: Return the smallest element of the heap.
    // REQUIRES: The heap is not empty.
    // RUNTIME: O(1)
    virtual const TYPE &get_min() const;

    // EFFECTS: Get the number of elements in the heap.
    // RUNTIME: O(1)
    virtual size_type size() const;

    // EFFECTS: Return true if the heap is empty.
    // RUNTIME: O(1)
    virtual bool empty() const;

        void percolateup(int id);

        void percolatedown(int id);

private:
    // Note: This vector *must* be used in your heap implementation.
    std::vector<TYPE> data;
    // Note: compare is a functor object
    COMP compare;

private:
    // Add any additional member functions or data you require here.
};

template<typename TYPE, typename COMP>
binary_heap<TYPE, COMP> :: binary_heap(COMP comp) {
        compare = comp;
        // Fill in the remaining lines if you need.
}
```

```
template<typename TYPE, typename COMP>
void binary_heap<TYPE, COMP> :: enqueue(const TYPE &val) {
	// Fill in the body.
	if (data.size()==0) data.push_back(val);
	data.push_back(val);
	percolateup(data.size()-1);
}

template<typename TYPE, typename COMP>
TYPE binary_heap<TYPE, COMP> :: dequeue_min() {
	// Fill in the body.
	int id=data.size();
	if (id>1)
	{
		TYPE tem=data[1];
		data[1]=data[id-1];
		data[id-1]=tem;
		data.pop_back();
		percolatedown(1);
		return tem;
	}
}

template<typename TYPE, typename COMP>
const TYPE &binary_heap<TYPE, COMP> :: get_min() const {
	// Fill in the body.
	if (data.size()>1) return data[1];
}

template<typename TYPE, typename COMP>
bool binary_heap<TYPE, COMP> :: empty() const {
	// Fill in the body.
	if (data.size()<2) return true;
	else return false;
}

template<typename TYPE, typename COMP>
unsigned binary_heap<TYPE, COMP> :: size() const {
	// Fill in the body.
	if (data.size()==0) return 0;
	else return data.size()-1;
}
```

```
template<typename TYPE, typename COMP>
void binary_heap<TYPE, COMP>:: percolateup(int id)
{
	TYPE tem;
	while (id>1 && compare(data[id],data[id/2]))
	{
		tem=data[id];
		data[id]=data[id/2];
		data[id/2]=tem;
		id=id/2;
	}

}

template<typename TYPE, typename COMP>
void binary_heap<TYPE, COMP>::percolatedown(int id)
{
	TYPE tem;
	int size=data.size()-1;
	for (int j=2*id;j<=size;j=2*id)
	{
		if (j<size && compare(data[j+1],data[j])) j++;
		if (compare(data[id],data[j])) break;
		tem=data[id];
		data[id]=data[j];
		data[j]=tem;
		id=j;
	}
}

#endif //BINARY_HEAP_H
```

unsorted_heap.h

```
#ifndef UNSORTED_HEAP_H
#define UNSORTED_HEAP_H

#include <algorithm>
#include "priority_queue.h"

// OVERVIEW: A specialized version of the 'heap' ADT that is implemented with
//		an underlying unordered array-based container. Every time a min
//		is required, a linear search is performed.
template<typename TYPE, typename COMP = std::less<TYPE> >
class unsorted_heap: public priority_queue<TYPE, COMP> {
```

```
public:
    typedef unsigned size_type;

    // EFFECTS: Construct an empty heap with an optional comparison functor.
    //          See test_heap.cpp for more details on functor.
    // MODIFIES: this
    // RUNTIME: O(1)
    unsorted_heap(COMP comp = COMP());

    // EFFECTS: Add a new element to the heap.
    // MODIFIES: this
    // RUNTIME: O(1)
    virtual void enqueue(const TYPE &val);

    // EFFECTS: Remove and return the smallest element from the heap.
    // REQUIRES: The heap is not empty.
    // MODIFIES: this
    // RUNTIME: O(n)
    virtual TYPE dequeue_min();

    // EFFECTS: Return the smallest element of the heap.
    // REQUIRES: The heap is not empty.
    // RUNTIME: O(n)
    virtual const TYPE &get_min() const;

    // EFFECTS: Get the number of elements in the heap.
    // RUNTIME: O(1)
    virtual size_type size() const;

    // EFFECTS: Return true if the heap is empty.
    // RUNTIME: O(1)
    virtual bool empty() const;

private:
    // Note: This vector *must* be used in your heap implementation.
    std::vector<TYPE> data;
    // Note: compare is a functor object
    COMP compare;
private:
    // Add any additional member functions or data you require here.
};

template<typename TYPE, typename COMP>
unsorted_heap<TYPE, COMP> :: unsorted_heap(COMP comp) {
```

```
        compare = comp;
        // Fill in the remaining lines if you need.
}

template<typename TYPE, typename COMP>
void unsorted_heap<TYPE, COMP> :: enqueue(const TYPE &val) {
        // Fill in the body.
        data.push_back(val);
}

template<typename TYPE, typename COMP>
TYPE unsorted_heap<TYPE, COMP> :: dequeue_min() {
        // Fill in the body.
        TYPE tem;
        int j=0;
        for (int i=0;i<data.size();i++)
                if (compare(data[i],data[j])) j=i;
        tem=data[j];
        data[j]=data[data.size()-1];
        data[data.size()-1]=tem;
        data.pop_back();
        return tem;
}

template<typename TYPE, typename COMP>
const TYPE &unsorted_heap<TYPE, COMP> :: get_min() const {
        // Fill in the body.
        int j=0;
        for (int i=0;i<data.size();i++)
                if (compare(data[i],data[j])) j=i;
        return data[j];
}

template<typename TYPE, typename COMP>
bool unsorted_heap<TYPE, COMP> :: empty() const {
        // Fill in the body.
        if (data.size()>0) return false;
        else return true;
}

template<typename TYPE, typename COMP>
unsigned unsorted_heap<TYPE, COMP> :: size() const {
        // Fill in the body.
        return data.size();
```

```
}

#endif //UNSORTED_HEAP_H
```

fib_heap.h

```
#ifndef FIB_HEAP_H
#define FIB_HEAP_H

#include <algorithm>
#include <cmath>
#include "priority_queue.h"
#include <iostream>

using namespace std;


// OVERVIEW: A specialized version of the 'heap' ADT implemented as a
//                    Fibonacci heap.
template<typename TYPE, typename COMP = std::less<TYPE> >
class fib_heap: public priority_queue<TYPE, COMP> {
public:
   typedef unsigned size_type;

   // EFFECTS: Construct an empty heap with an optional comparison functor.
   //                  See test_heap.cpp for more details on functor.
   // MODIFIES: this
   // RUNTIME: O(1)
   fib_heap(COMP comp = COMP());

// EFFECTS: Deconstruct the heap with no memory leak.
   // MODIFIES: this
   // RUNTIME: O(n)
   ~fib_heap();

   // EFFECTS: Add a new element to the heap.
   // MODIFIES: this
   // RUNTIME: O(1)
   virtual void enqueue(const TYPE &val);

   // EFFECTS: Remove and return the smallest element from the heap.
   // REQUIRES: The heap is not empty.
   // MODIFIES: this
   // RUNTIME: Amortized O(log(n))
   virtual TYPE dequeue_min();
```

```
    // EFFECTS: Return the smallest element of the heap.
    // REQUIRES: The heap is not empty.
    // RUNTIME: O(1)
    virtual const TYPE &get_min() const;

    // EFFECTS: Get the number of elements in the heap.
    // RUNTIME: O(1)
    virtual size_type size() const;

    // EFFECTS: Return true if the heap is empty.
    // RUNTIME: O(1)
    virtual bool empty() const;

private:
    // Note: compare is a functor object
    COMP compare;

private:
    // Add any additional member functions or data you require here.
    // You may want to define a strcut/class to represent nodes in the heap and a
    // pointer to the min node in the heap.
        struct node
        {
            TYPE key;
            node *parent;
            node *child;
            node *left;
            node *right;
            int degree;
            bool mark;
        };

        int n;
        node *min;

        void Consolidate();
};

// Add the definitions of the member functions here. Please refer to
template<typename TYPE, typename COMP>
fib_heap<TYPE, COMP> :: fib_heap(COMP comp) {
        compare = comp;
        min=NULL;
```

```
    n=0;
    // Fill in the remaining lines if you need.
}

template<typename TYPE, typename COMP>
fib_heap<TYPE, COMP>::~fib_heap()
{
    while (n>0)
    {
        n--;
        dequeue_min();
    }
}

template<typename TYPE, typename COMP>
void fib_heap<TYPE, COMP>::enqueue(const TYPE &val)
{
    node *np=new node;
    np->key=val;
    np->degree=0;
    np->mark=false;
    np->parent=NULL;
    np->child=NULL;
    np->left=NULL;
    np->right=NULL;
    if (min==NULL)
    {
        min=np;
        min->right=min;
        min->left=min;
        n=1;
    }
    else
    {
        min->right->left=np;
        np->right=min->right;
        np->left=min;
        min->right=np;
        n++;
        if (compare(np->key,min->key)) min=np;
    }
}

template<typename TYPE, typename COMP>
```

```
TYPE fib_heap<TYPE, COMP>::dequeue_min()
{
	node *np=min;
	TYPE tem;
	if (min!=NULL)
	{
		node *np1=np->child;
		node *np2,*np3;
		if (np1!=NULL)
		{
			np2=np1->right;
			np1->parent=NULL;
			min->right->left=np1;
			np1->right=min->right;
			np1->left=min;
			min->right=np1;
			while (np2!=np1)
			{
				np3=np2->right;
				np2->parent=NULL;
				min->right->left=np2;
				np2->right=min->right;
				np2->left=min;
				min->right=np2;
				np2=np3;
			}
		}
		np->right->left=np->left;
		np->left->right=np->right;
		if (np==np->right)
			min=NULL;
		else
		{
			min=np->right;
			Consolidate();
		}
		n--;
		tem=np->key;
		delete np;
		return tem;
	}
}

template<typename TYPE, typename COMP>
```

```
const TYPE &fib_heap<TYPE, COMP>::get_min() const {
	if (n>0) return min->key;
}

template<typename TYPE, typename COMP>
bool fib_heap<TYPE, COMP> :: empty() const {
	// Fill in the body.
	if (n==0) return true;
	else return false;
}

template<typename TYPE, typename COMP>
unsigned fib_heap<TYPE, COMP> :: size() const {
	// Fill in the body.
	return n;
}

template<typename TYPE, typename COMP>
void fib_heap<TYPE, COMP>::Consolidate()
{
	int D=log(n)/log((1+sqrt(5))/2)+1;
	node *A[D];
	for (int i=0;i<D;i++) A[i]=NULL;
	A[min->degree]=min;
	int t=1;
	node *np=min->right;
	while (np!=min)
	{
		t++;
		np=np->right;
	}
	node *root[t];
	np=min;
	int k=0;
	while (k<t)
	{
		root[k]=np;
		np=np->right;
		k++;
	}
	k=1;
	node *x,*y,*tem;
	int d;
	while (k<t)
```

```
{
    x=root[k];
    d=x->degree;
    k++;
    while (A[d]!=NULL)
    {
        y=A[d];
        if (compare(y->key,x->key))
        {
            tem=y;
            y=x;
            x=tem;
        }
        y->left->right=y->right;
        y->right->left=y->left;
        y->parent=x;
        x->degree++;
        y->mark=false;
        if (x->child==NULL)
        {
            x->child=y;
            y->left=y;
            y->right=y;
        }
        else
        {
            x->child->right->left=y;
            y->right=x->child->right;
            y->left=x->child;
            x->child->right=y;
        }
        A[d]=NULL;
        d++;
    }
    A[d]=x;
}
min=NULL;
for (int i=0;i<D;i++)
{
    if (A[i]!=NULL)
    {

        if (min==NULL)
        {
```

```
                    min=A[i];
                    min->left=min;
                    min->right=min;
                }
                else
                {
                    min->right->left=A[i];
                    A[i]->right=min->right;
                    A[i]->left=min;
                    min->right=A[i];
                    if (compare(A[i]->key,min->key))
                        min=A[i];
                }
            }
        }
}
#endif //FIB_HEAP_H
```