UM-SJTU JOINT INSTITUTE
Data Structures and Algorithms
(VP281)

Programming Assignment

Programming Assignment Three
Priority Queue and its Application

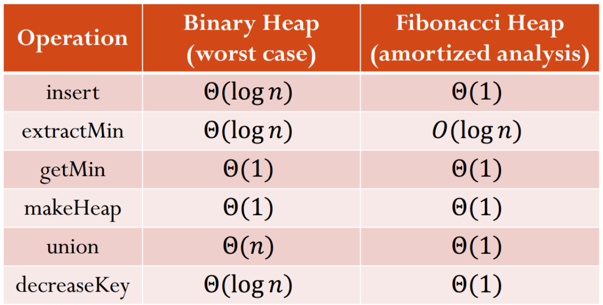
Name: Pan Chongdan
ID: 516370910121

Date: November 9, 2018

1 Introduction

The programming assignment asks me to implement three priority queue including binary heap, unsorted heap and fibonacci heap. The goal is clear, use priority queue to find the shortest path from the source cell to the ending cell

Second, thstudy the efficiency of different implementations. In lecture slides 13, professor gives a summary for the time complexity for binary heap and fibonacci heap.



Operation	Binary Heap (worst case)	Fibonacci Heap (amortized analysis)
insert	$\Theta(\log n)$	$\Theta(1)$
extractMin	$\Theta(\log n)$	$O(\log n)$
getMin	$\Theta(1)$	$\Theta(1)$
makeHeap	$\Theta(1)$	$\Theta(1)$
union	$\Theta(n)$	$\Theta(1)$
decreaseKey	$\Theta(\log n)$	$\Theta(1)$

Figure 1: Random Selection

However, we need to test the priority queue efficiency by ourselves, so I wrote a cpp file to print out the time required for each algorithm, which is included in appendix part.

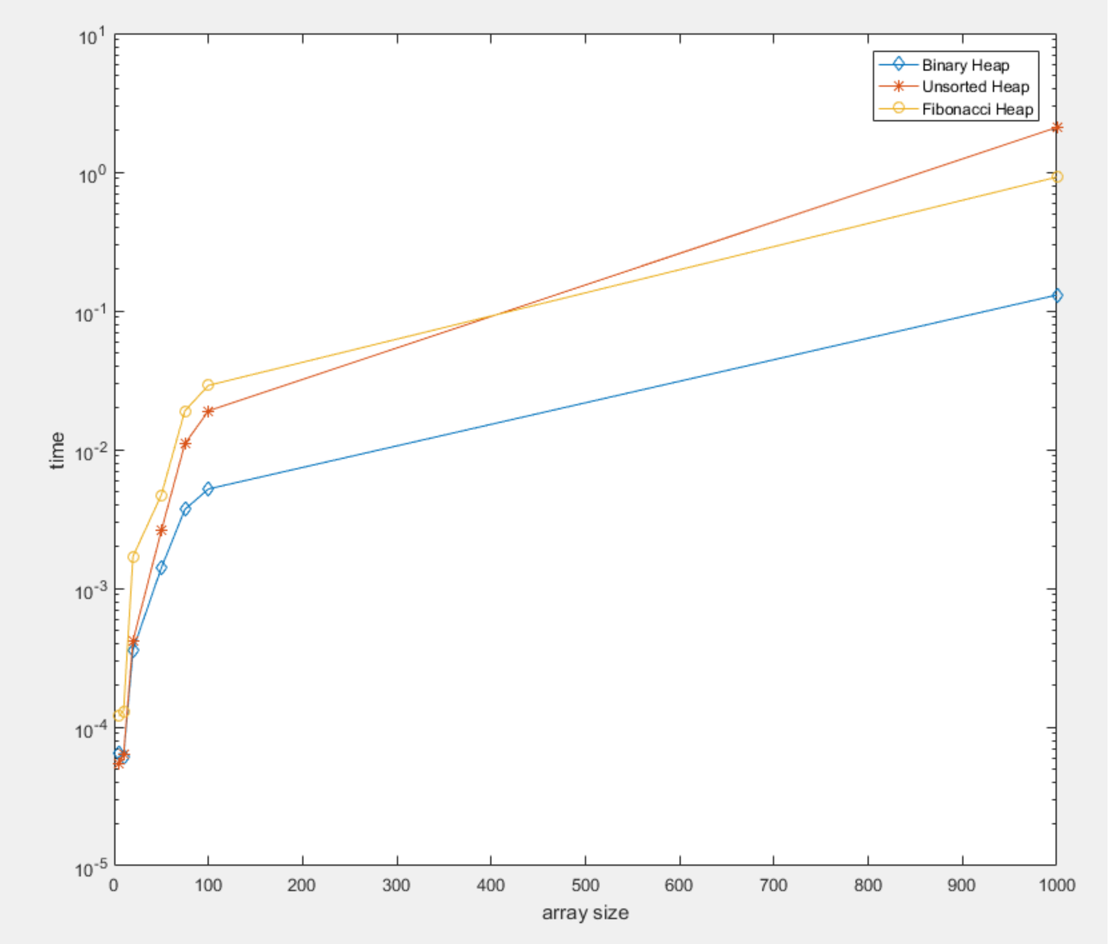
2 Result

To test time efficiency, I used clock() function to get the time required. To get rid of other disturbing factor, I used to variable start and stop to get the time right before and after the sorting complete, then their difference is the time used. In my analysis, I get 7 set of data ,from which the map's length of side is 5,10,20,50,75,100,1000 and the size is the length's square.

map Size	5	10	20	50	75	100	1000
Binary Heap	6.5e-5	6.1e-5	3.6e-4	1.4e-3	3.7e-3	5.2e-3	0.13
Unsorted Heap	5.5e-5	6.3e-5	4.2e-4	2.6e-3	1.1e-2	1.9e-2	2.1
Fibonacci Heap	1.2e-4	1.3e-4	1.7e-3	4.7e-3	1.9e-2	2.9e-2	0.92

3 Conclusion

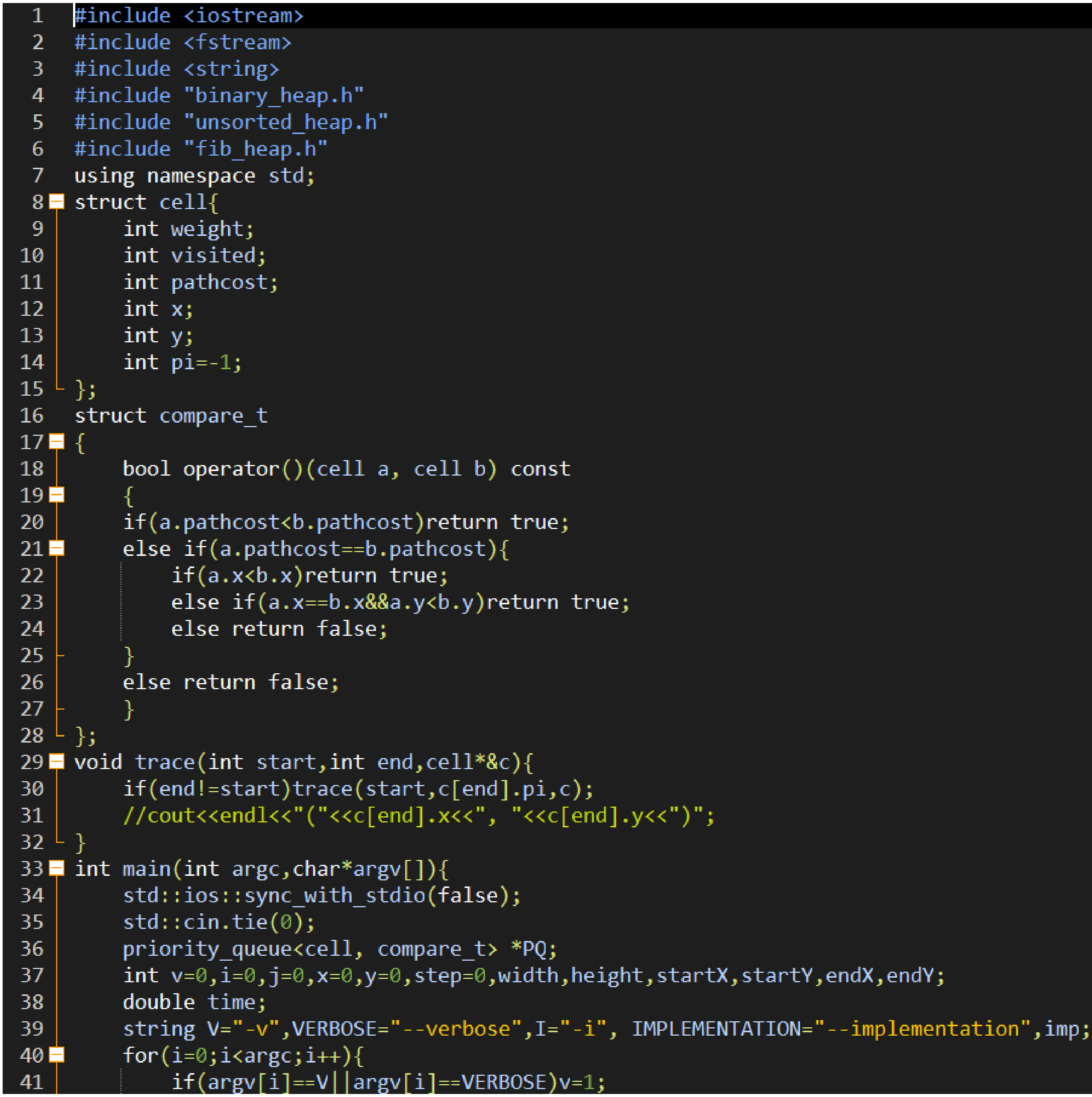
This chart is plot by matlab, showing the time efficiency of each algorithms compared to each other, and it has following characteristics.



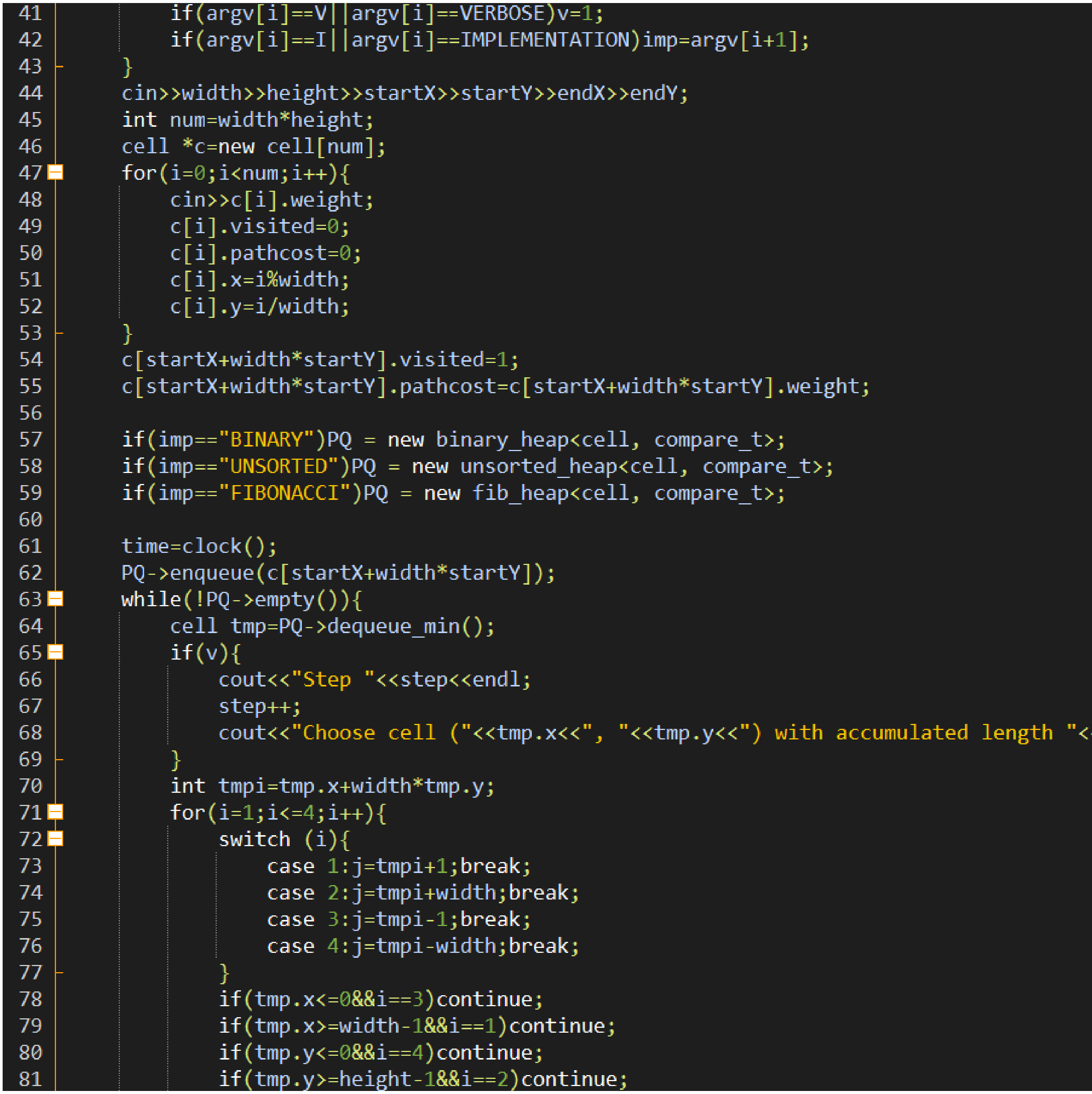
1. The average time required for each algorithm grows as the map grows bigger.
2. When the map is small, the time used for binary heap and unsorted heap is very close but binary have an advantage when the map is bigger as well as the Fibonacci.
3. Fibonacci heap always takes the longer time than Binary Heap, which is not reasonable, I think it's because it takes more time to use implement with list, but its time complexity is similar others from the picture

4 Appendix

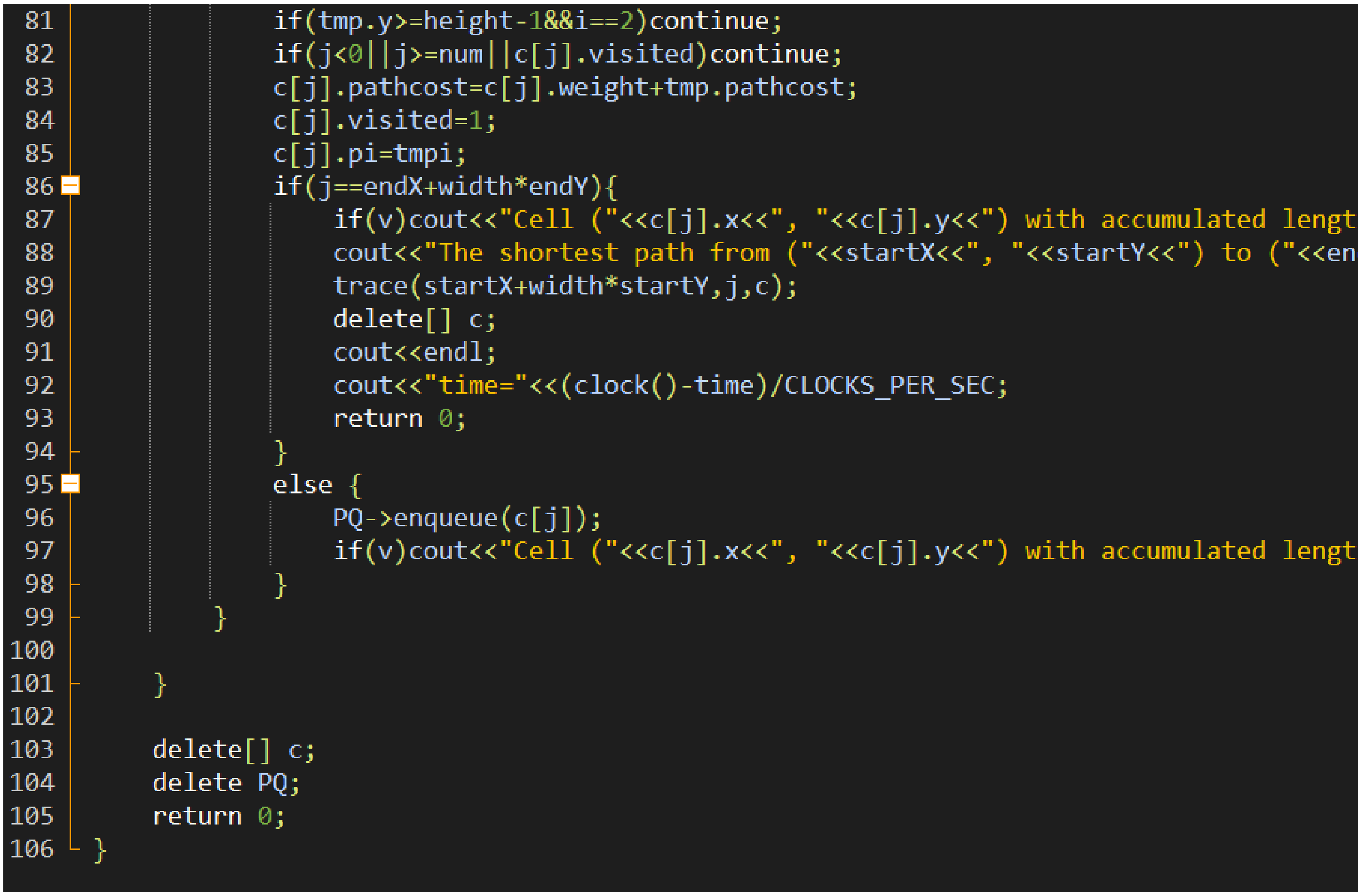
The appendix shows the cpp code for main project and time comparison. The following part is for the main algorithm



```
#include <iostream>
#include <fstream>
#include <string>
#include "binary_heap.h"
#include "unsorted_heap.h"
#include "fib_heap.h"
using namespace std;
struct cell{
    int weight;
    int visited;
    int pathcost;
    int x;
    int y;
    int pi=-1;
};
struct compare_t
{
    bool operator()(cell a, cell b) const
    {
    if(a.pathcost<b.pathcost)return true;
    else if(a.pathcost==b.pathcost){
        if(a.x<b.x)return true;
        else if(a.x==b.x&&a.y<b.y)return true;
        else return false;
    }
    else return false;
    }
};
void trace(int start,int end,cell*&c){
    if(end!=start)trace(start,c[end].pi,c);
    //cout<<endl<<"("<<c[end].x<<", "<<c[end].y<<")";
}
int main(int argc,char*argv[]){
    std::ios::sync_with_stdio(false);
    std::cin.tie(0);
    priority_queue<cell, compare_t> *PQ;
    int v=0,i=0,j=0,x=0,y=0,step=0,width,height,startX,startY,endX,endY;
    double time;
    string V="-v",VERBOSE="--verbose",I="-i", IMPLEMENTATION="--implementation",imp;
    for(i=0;i<argc;i++){
        if(argv[i]==V||argv[i]==VERBOSE)v=1;
```

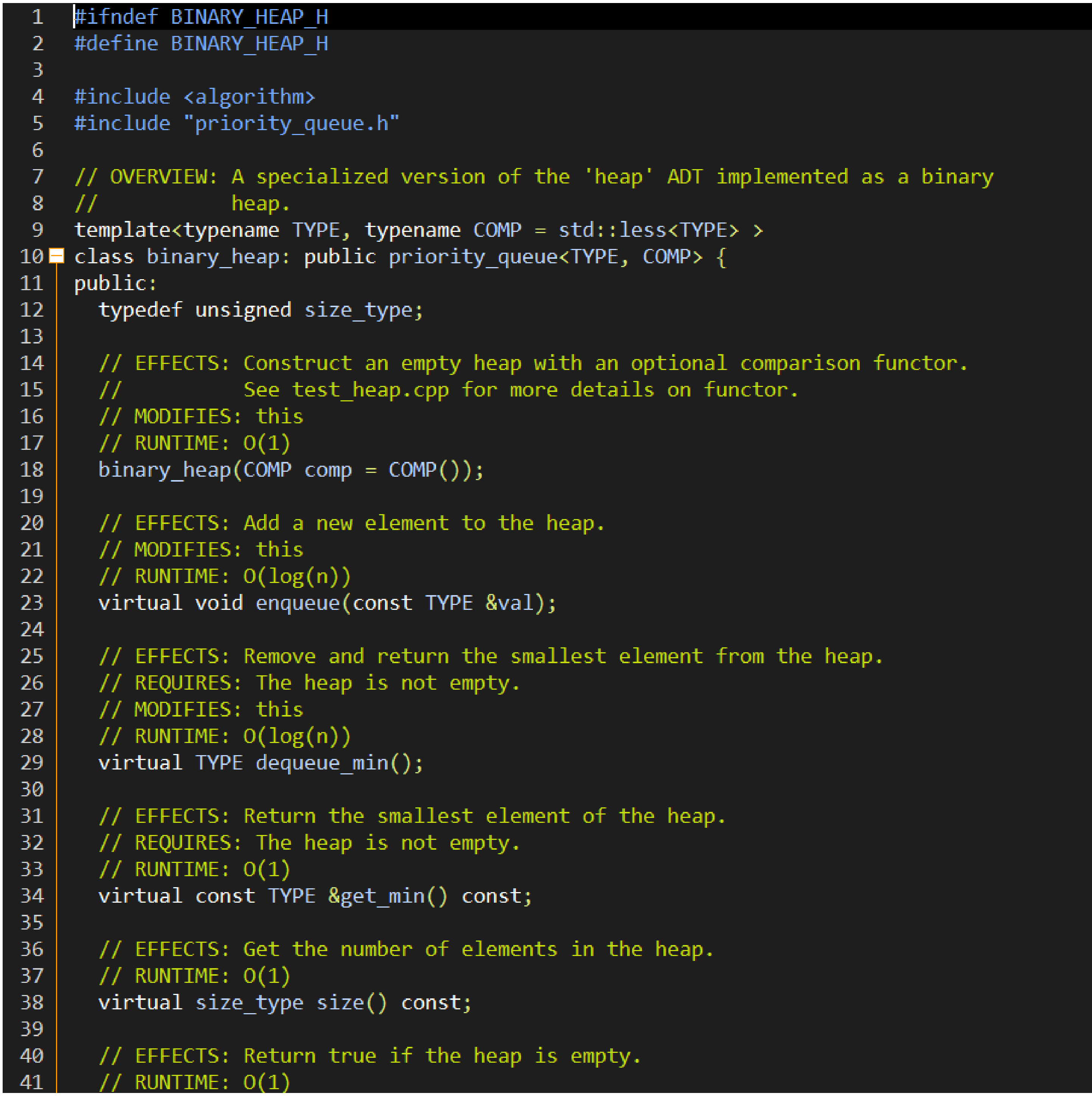


```
        if(argv[i]==V||argv[i]==VERBOSE)v=1;
        if(argv[i]==I||argv[i]==IMPLEMENTATION)imp=argv[i+1];
    }
    cin>>width>>height>>startX>>startY>>endX>>endY;
    int num=width*height;
    cell *c=new cell[num];
    for(i=0;i<num;i++){
        cin>>c[i].weight;
        c[i].visited=0;
        c[i].pathcost=0;
        c[i].x=i%width;
        c[i].y=i/width;
    }
    c[startX+width*startY].visited=1;
    c[startX+width*startY].pathcost=c[startX+width*startY].weight;

    if(imp=="BINARY")PQ = new binary_heap<cell, compare_t>;
    if(imp=="UNSORTED")PQ = new unsorted_heap<cell, compare_t>;
    if(imp=="FIBONACCI")PQ = new fib_heap<cell, compare_t>;

    time=clock();
    PQ->enqueue(c[startX+width*startY]);
    while(!PQ->empty()){
        cell tmp=PQ->dequeue_min();
        if(v){
            cout<<"Step "<<step<<endl;
            step++;
            cout<<"Choose cell ("<<tmp.x<<", "<<tmp.y<<") with accumulated length "<
        }
        int tmpi=tmp.x+width*tmp.y;
        for(i=1;i<=4;i++){
            switch (i){
                case 1:j=tmpi+1;break;
                case 2:j=tmpi+width;break;
                case 3:j=tmpi-1;break;
                case 4:j=tmpi-width;break;
            }
            if(tmp.x<=0&&i==3)continue;
            if(tmp.x>=width-1&&i==1)continue;
            if(tmp.y<=0&&i==4)continue;
            if(tmp.y>=height-1&&i==2)continue;
```

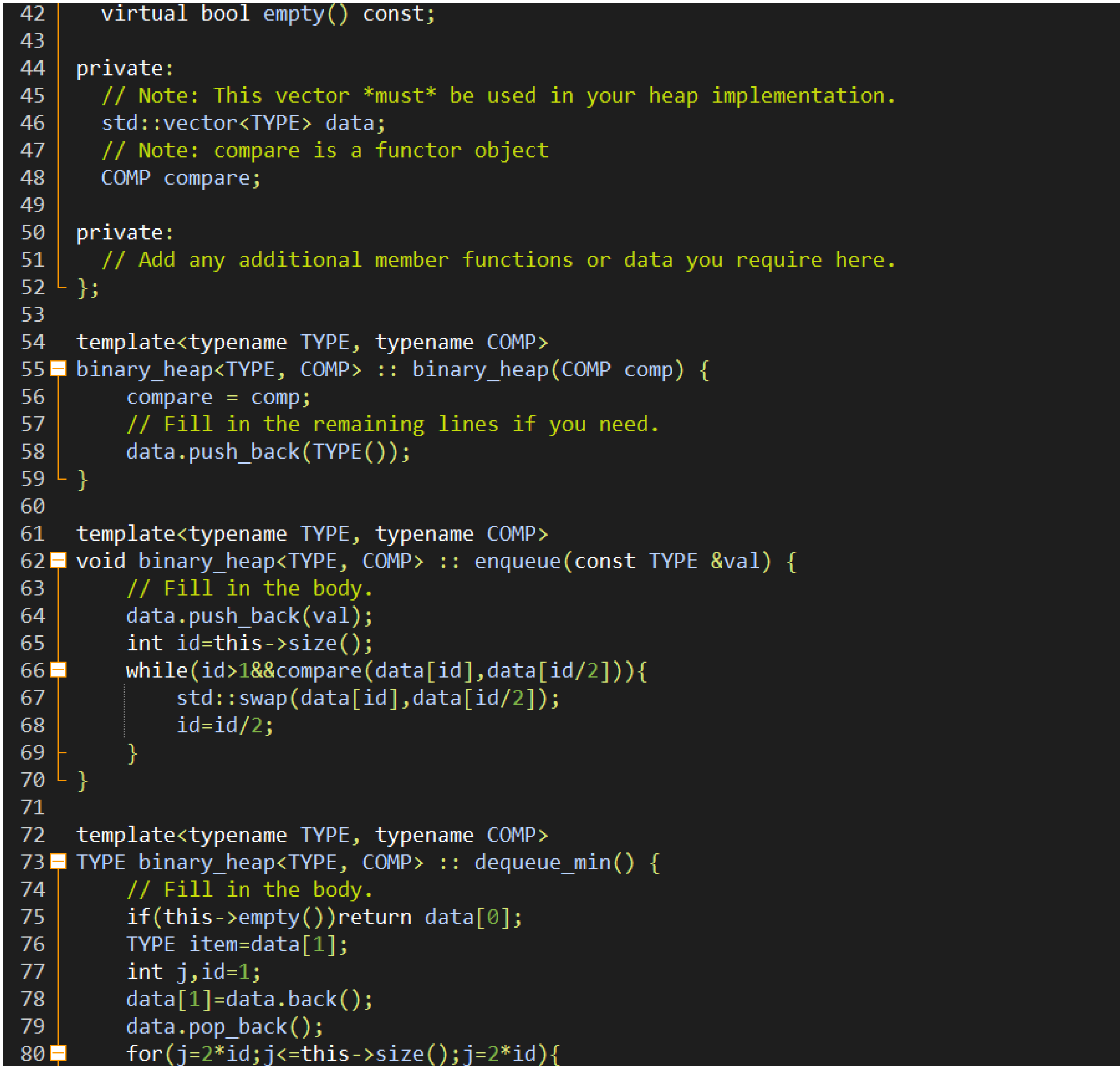


```
            if(tmp.y>=height-1&&i==2)continue;
            if(j<0||j>=num||c[j].visited)continue;
            c[j].pathcost=c[j].weight+tmp.pathcost;
            c[j].visited=1;
            c[j].pi=tmpi;
            if(j==endX+width*endY){
                if(v)cout<<"Cell ("<<c[j].x<<", "<<c[j].y<<") with accumulated lengt
                cout<<"The shortest path from ("<<startX<<", "<<startY<<") to ("<<en
                trace(startX+width*startY,j,c);
                delete[] c;
                cout<<endl;
                cout<<"time="<<(clock()-time)/CLOCKS_PER_SEC;
                return 0;
            }
            else {
                PQ->enqueue(c[j]);
                if(v)cout<<"Cell ("<<c[j].x<<", "<<c[j].y<<") with accumulated lengt
            }
        }

    }

    delete[] c;
    delete PQ;
    return 0;
}
```

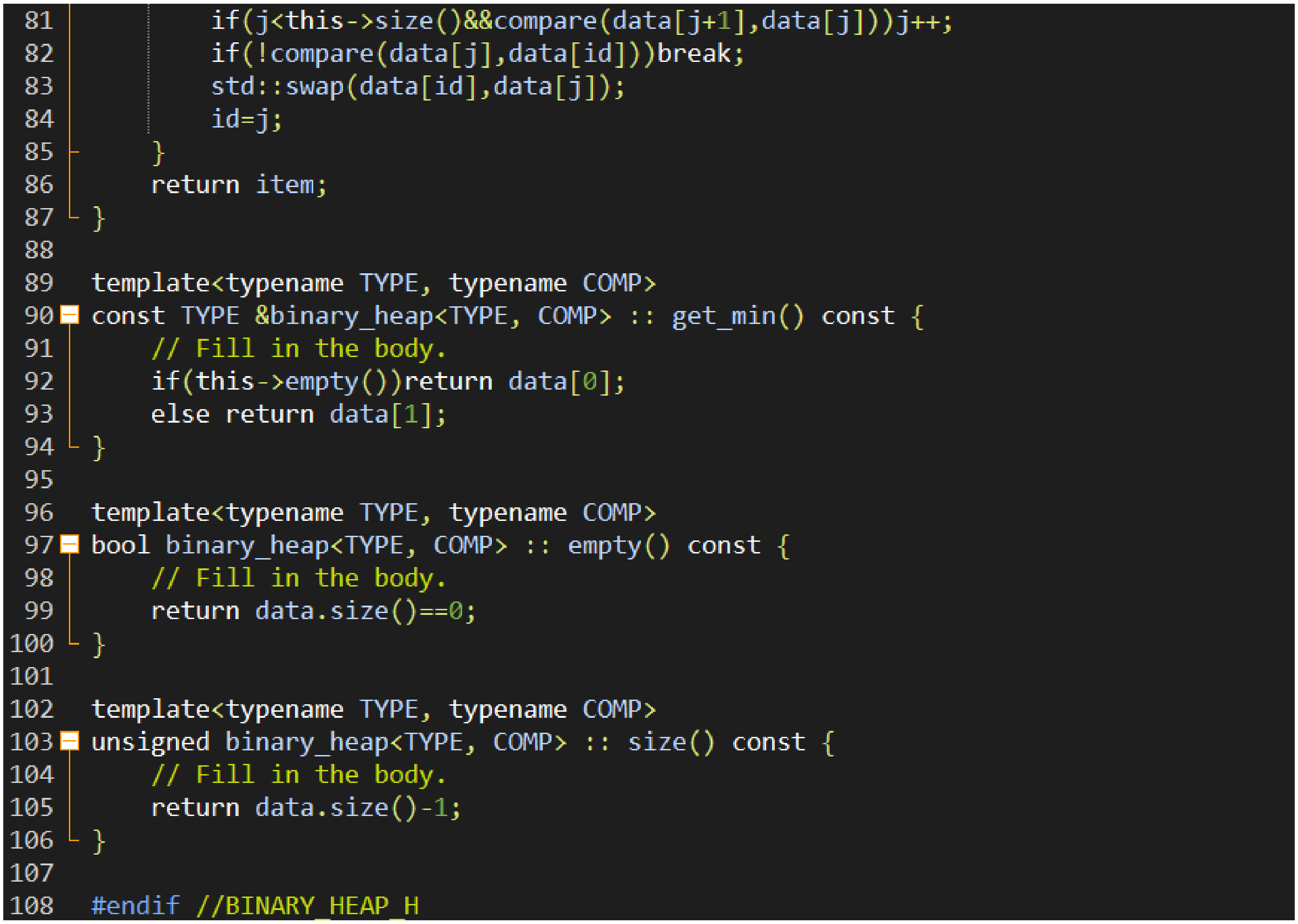
The following is for binary heap.



```
#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)
```

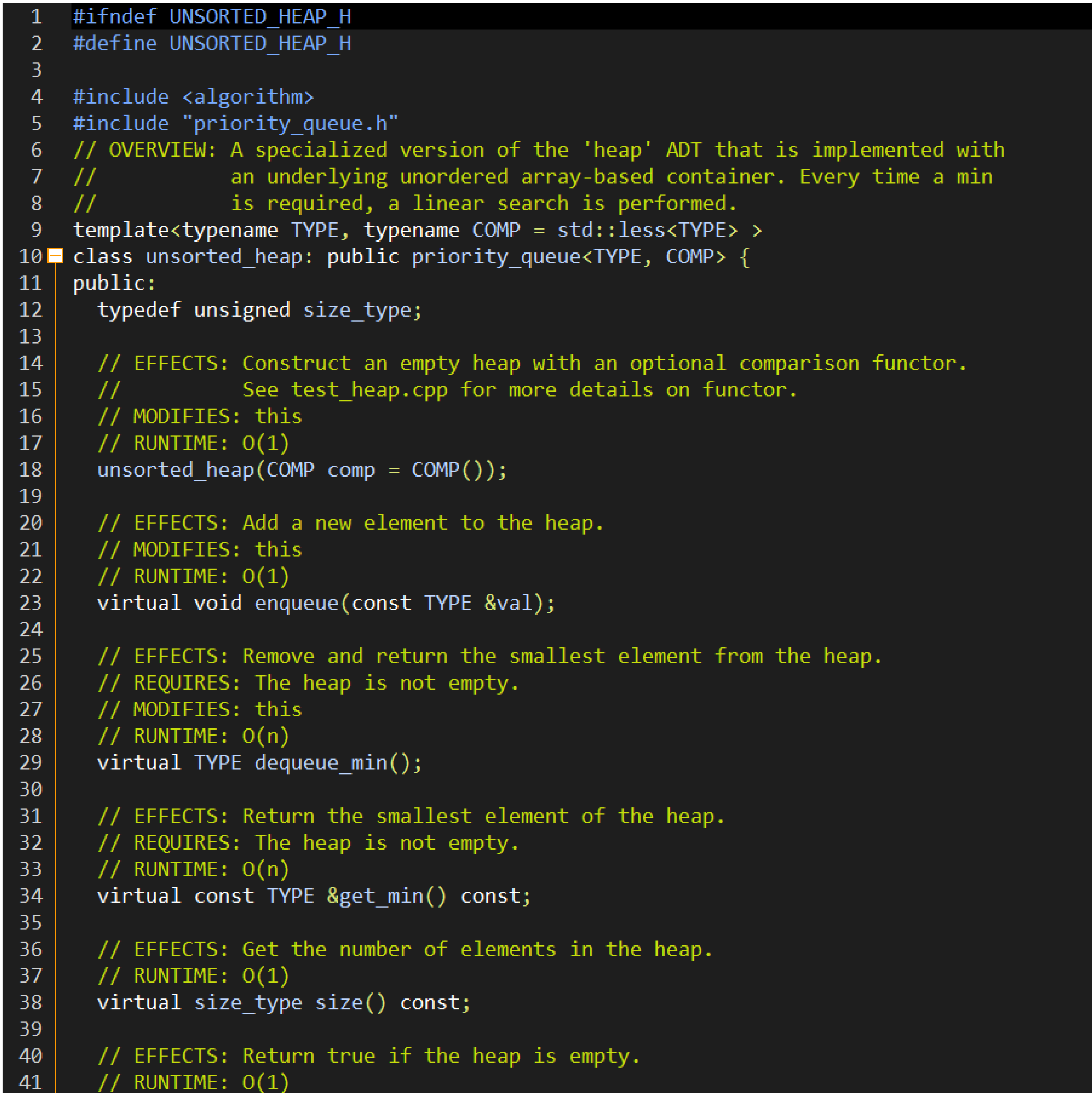


```cpp
  virtual bool empty() const;

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

template<typename TYPE, typename COMP>
void binary_heap<TYPE, COMP> :: enqueue(const TYPE &val) {
    // Fill in the body.
    data.push_back(val);
    int id=this->size();
    while(id>1&&compare(data[id],data[id/2])){
        std::swap(data[id],data[id/2]);
        id=id/2;
    }
}

template<typename TYPE, typename COMP>
TYPE binary_heap<TYPE, COMP> :: dequeue_min() {
    // Fill in the body.
    if(this->empty())return data[0];
    TYPE item=data[1];
    int j,id=1;
    data[1]=data.back();
    data.pop_back();
    for(j=2*id;j<=this->size();j=2*id){
```

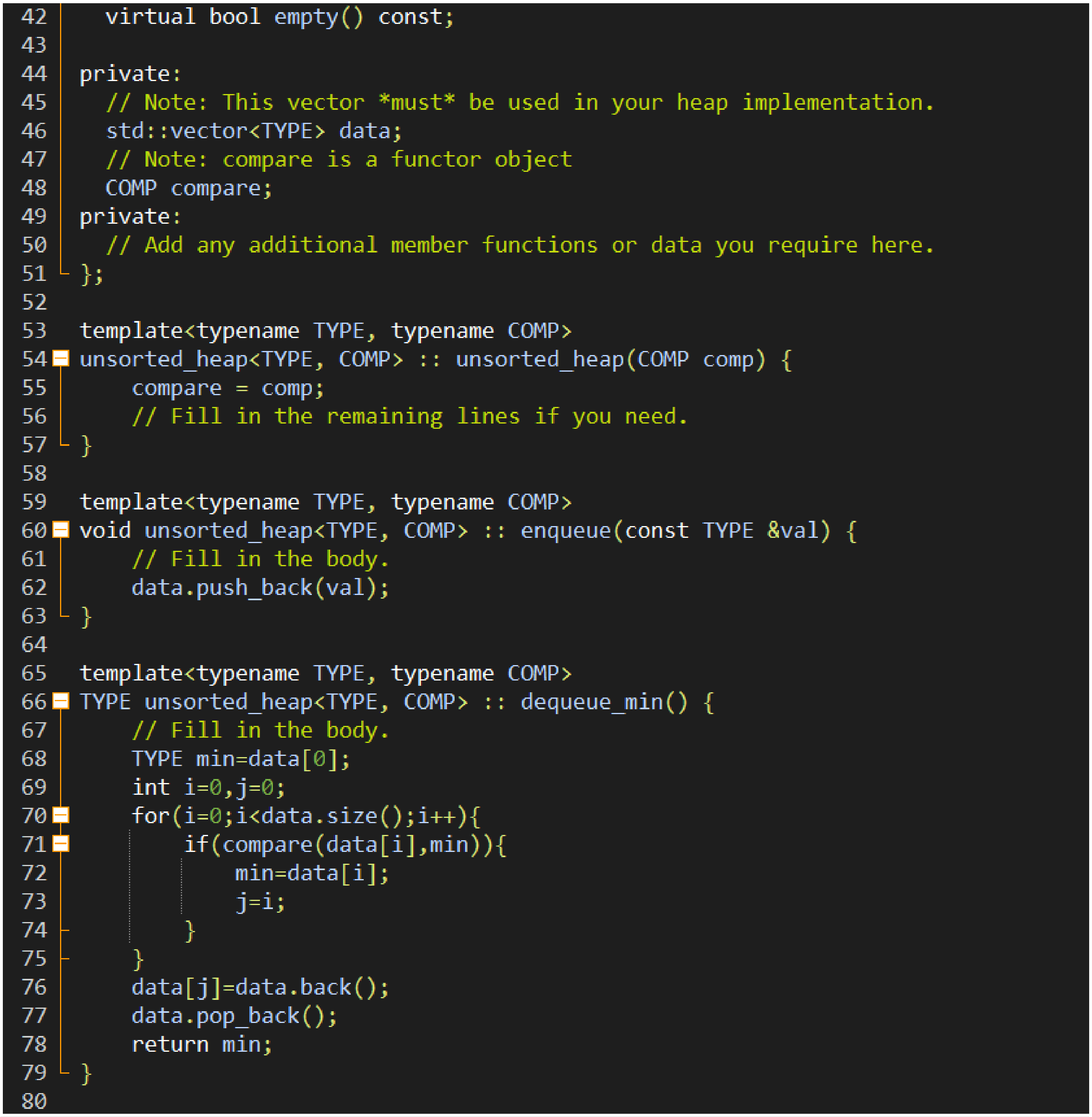


```
        if(j<this->size()&&compare(data[j+1],data[j]))j++;
        if(!compare(data[j],data[id]))break;
        std::swap(data[id],data[j]);
        id=j;
    }
    return item;
}

template<typename TYPE, typename COMP>
const TYPE &binary_heap<TYPE, COMP> :: get_min() const {
    // Fill in the body.
    if(this->empty())return data[0];
    else return data[1];
}

template<typename TYPE, typename COMP>
bool binary_heap<TYPE, COMP> :: empty() const {
    // Fill in the body.
    return data.size()==0;
}

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

#endif //BINARY_HEAP_H
```

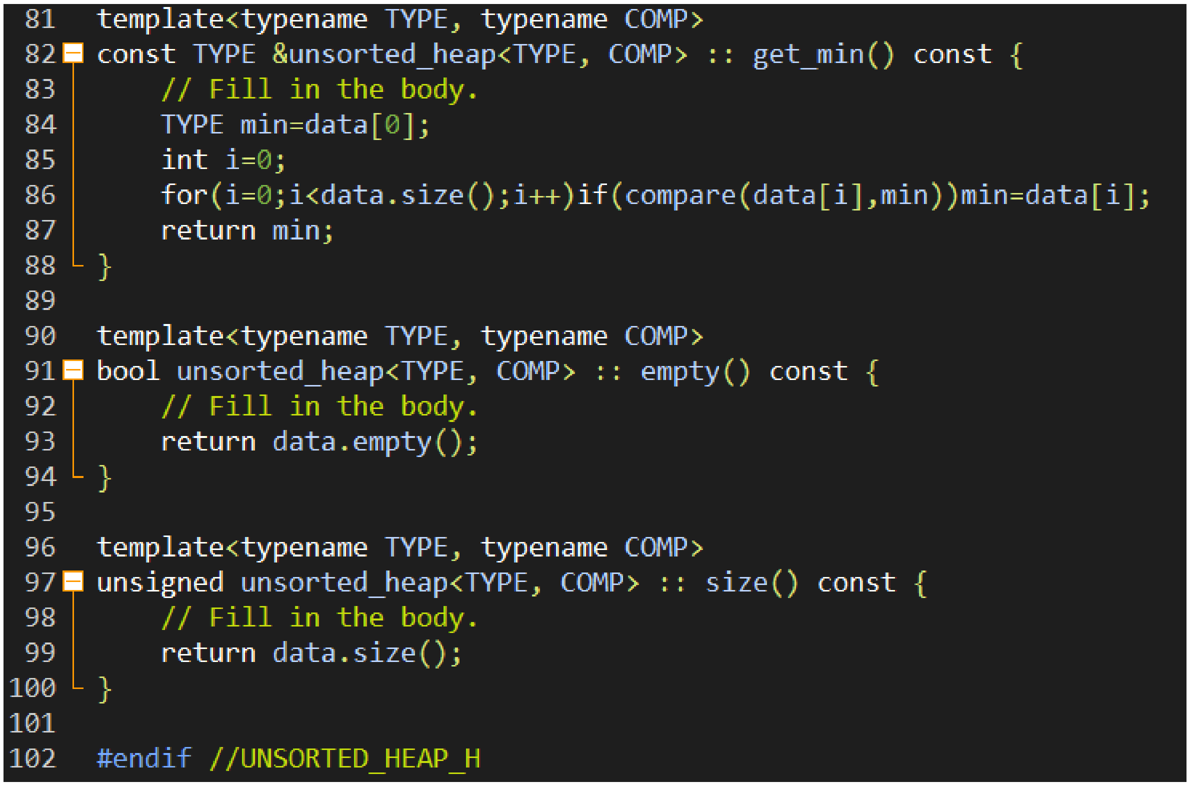
The following is for unsorted heap.



```
#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)
```

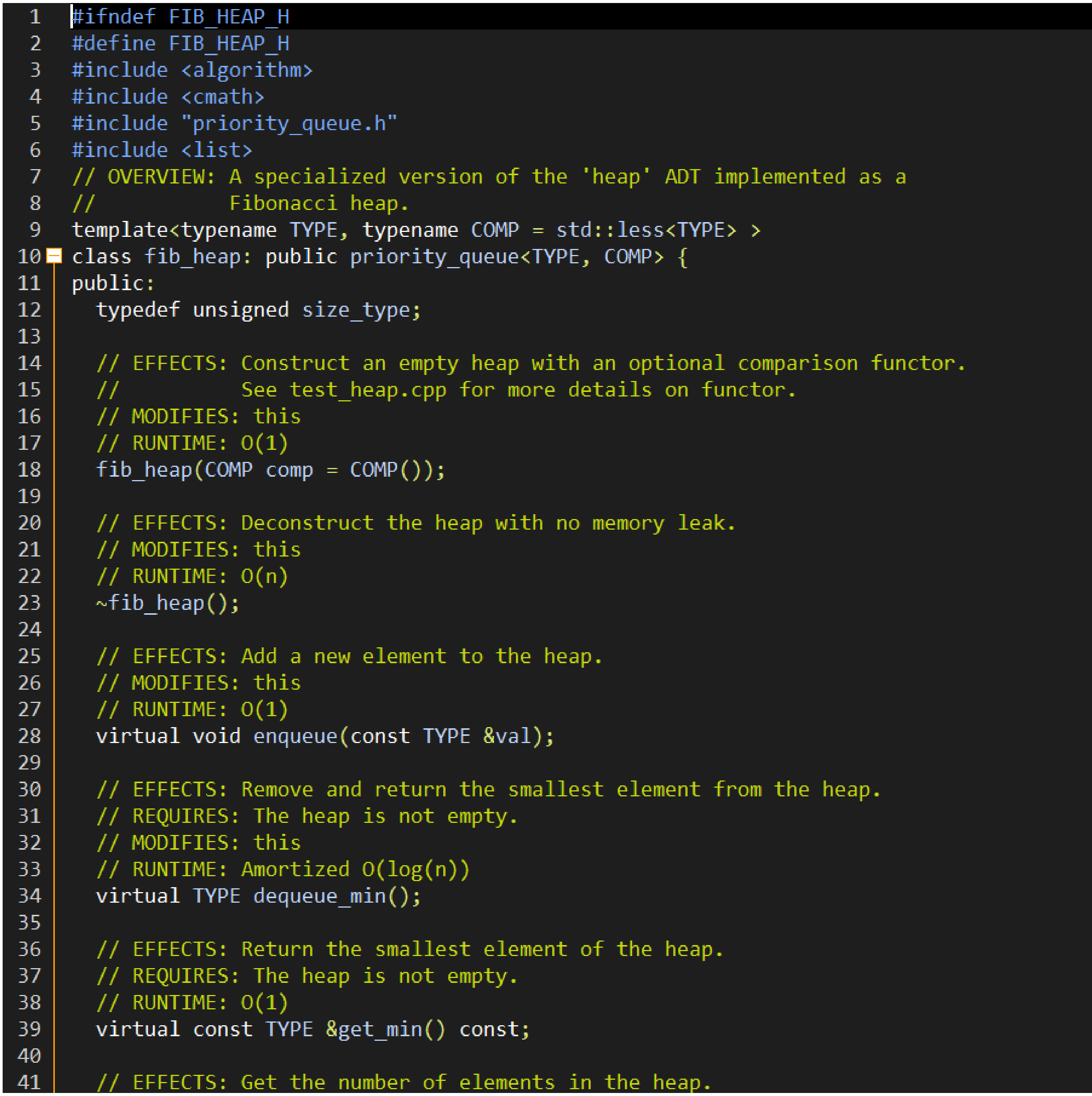


```
  virtual bool empty() const;

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 min=data[0];
    int i=0,j=0;
    for(i=0;i<data.size();i++){
        if(compare(data[i],min)){
            min=data[i];
            j=i;
        }
    }
    data[j]=data.back();
    data.pop_back();
    return min;
}

```

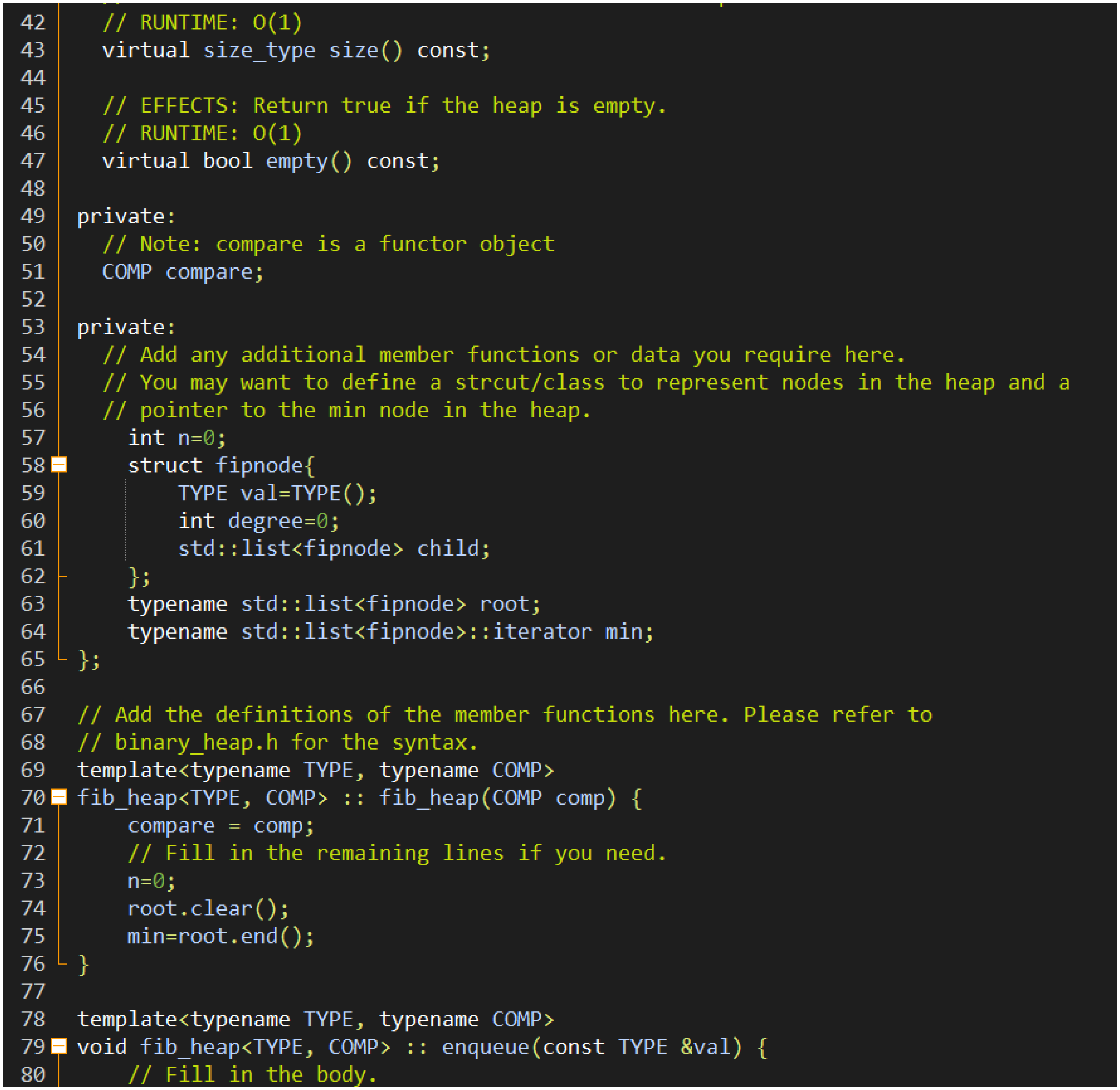


```cpp
template<typename TYPE, typename COMP>
const TYPE &unsorted_heap<TYPE, COMP> :: get_min() const {
    // Fill in the body.
    TYPE min=data[0];
    int i=0;
    for(i=0;i<data.size();i++)if(compare(data[i],min))min=data[i];
    return min;
}

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

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

#endif //UNSORTED_HEAP_H
```

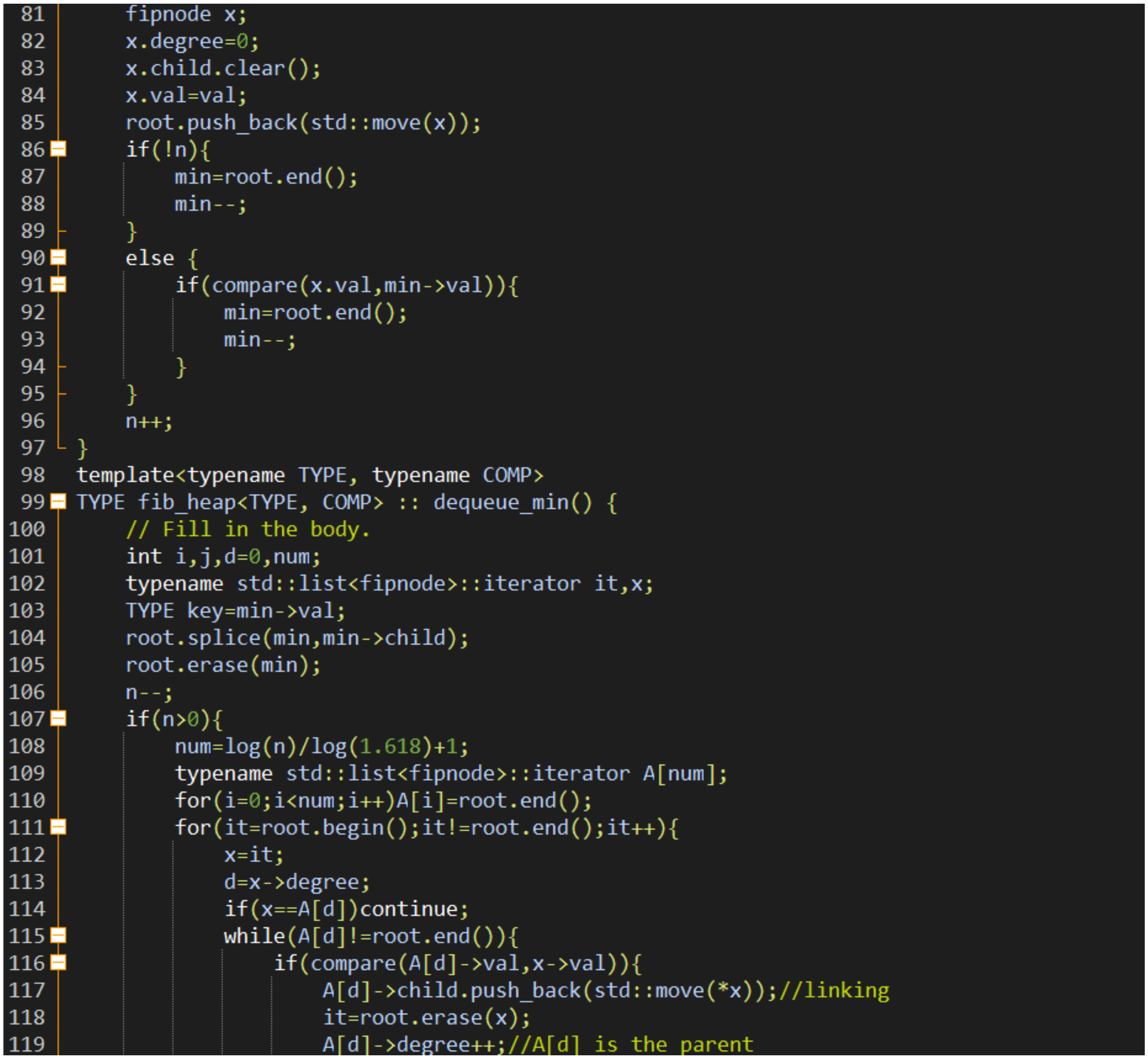
The following is for fibonacci heap.



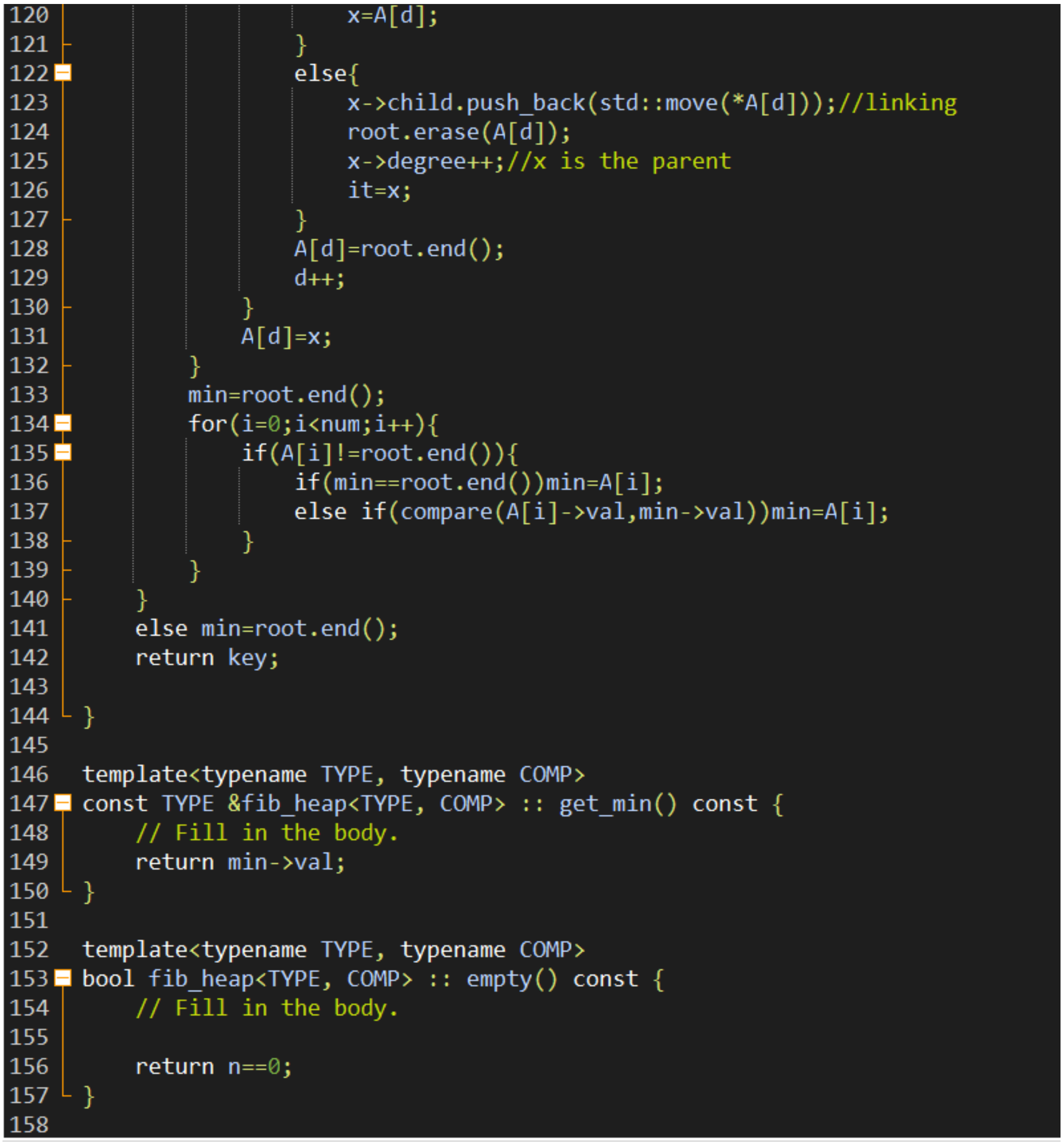
```
#ifndef FIB_HEAP_H
#define FIB_HEAP_H
#include <algorithm>
#include <cmath>
#include "priority_queue.h"
#include <list>
// OVERVIEW: A specialized version of the 'heap' ADT implemented as a
//           Fibonacci heap.
template<typename TYPE, typename COMP = std::less<TYPE> >
class fib_heap: public 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.
```



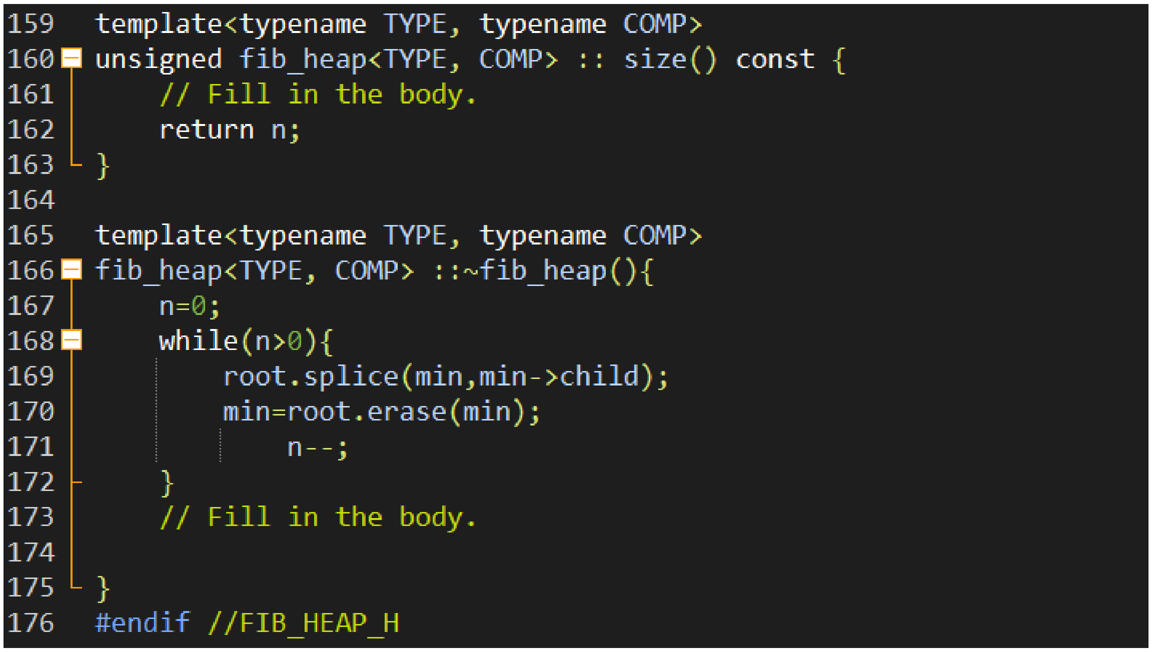
```
  // RUNTIME: O(1)
  virtual size_type size() const;

  // 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.
    int n=0;
    struct fipnode{
        TYPE val=TYPE();
        int degree=0;
        std::list<fipnode> child;
    };
    typename std::list<fipnode> root;
    typename std::list<fipnode>::iterator min;
};

// Add the definitions of the member functions here. Please refer to
// binary_heap.h for the syntax.
template<typename TYPE, typename COMP>
fib_heap<TYPE, COMP> :: fib_heap(COMP comp) {
    compare = comp;
    // Fill in the remaining lines if you need.
    n=0;
    root.clear();
    min=root.end();
}

template<typename TYPE, typename COMP>
void fib_heap<TYPE, COMP> :: enqueue(const TYPE &val) {
    // Fill in the body.
```



```
    fipnode x;
    x.degree=0;
    x.child.clear();
    x.val=val;
    root.push_back(std::move(x));
    if(!n){
        min=root.end();
        min--;
    }
    else {
        if(compare(x.val,min->val)){
            min=root.end();
            min--;
        }
    }
    n++;
}
template<typename TYPE, typename COMP>
TYPE fib_heap<TYPE, COMP> :: dequeue_min() {
    // Fill in the body.
    int i,j,d=0,num;
    typename std::list<fipnode>::iterator it,x;
    TYPE key=min->val;
    root.splice(min,min->child);
    root.erase(min);
    n--;
    if(n>0){
        num=log(n)/log(1.618)+1;
        typename std::list<fipnode>::iterator A[num];
        for(i=0;i<num;i++)A[i]=root.end();
        for(it=root.begin();it!=root.end();it++){
            x=it;
            d=x->degree;
            if(x==A[d])continue;
            while(A[d]!=root.end()){
                if(compare(A[d]->val,x->val)){
                    A[d]->child.push_back(std::move(*x));//linking
                    it=root.erase(x);
                    A[d]->degree++;//A[d] is the parent
```



```
                    x=A[d];
                }
                else{
                    x->child.push_back(std::move(*A[d]));//linking
                    root.erase(A[d]);
                    x->degree++;//x is the parent
                    it=x;
                }
                A[d]=root.end();
                d++;
            }
            A[d]=x;
        }
        min=root.end();
        for(i=0;i<num;i++){
            if(A[i]!=root.end()){
                if(min==root.end())min=A[i];
                else if(compare(A[i]->val,min->val))min=A[i];
            }
        }
    }
    else min=root.end();
    return key;

}

template<typename TYPE, typename COMP>
const TYPE &fib_heap<TYPE, COMP> :: get_min() const {
    // Fill in the body.
    return min->val;
}

template<typename TYPE, typename COMP>
bool fib_heap<TYPE, COMP> :: empty() const {
    // Fill in the body.

    return n==0;
}

```



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

template<typename TYPE, typename COMP>
fib_heap<TYPE, COMP> ::~fib_heap(){
    n=0;
    while(n>0){
        root.splice(min,min->child);
        min=root.erase(min);
            n--;
    }
    // Fill in the body.

}
#endif //FIB_HEAP_H
```