算法基础第二次实验

PB19000352 易元昆

实验设备和环境

- Windows 11 专业版 21H2
- Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz   2.59 GHz 16GB
- 编译环境 `clang --version 13.0.0` c++版本 `-std=c++20` (tips:使用了c++17及以上版本才支持的 `std::filesystem` 用于确定文件路径)
- 调试环境 `GNU gdb(GDB) 10.2`

EX1 斐波那契堆

实验内容

- 通过 `insert` 建立斐波那契堆 `H1~H4`
- 在H1到H4中分别完成10个ops，然后合成H5，并再进行10个ops
- 输出时间并进行分析

方法与步骤

1. 建立存储斐波那契堆的数据结构

```
//斐波那契结点ADT
struct FibonacciHeapNode {
    int                 key;     //结点
    int                 degree;  //度
    FibonacciHeapNode* left;    //左兄弟
    FibonacciHeapNode* right;   //右兄弟
    FibonacciHeapNode* parent;  //父结点
    FibonacciHeapNode* child;   //第一个孩子结点
    bool                marked;  //是否被删除第1个孩子
};

typedef FibonacciHeapNode FibNode;

//斐波那契堆ADT
struct FibonacciHeap {
    int                 keyNum;          //堆中结点个数
    FibonacciHeapNode*  min;             //最小堆，根结点
    int                 maxNumOfDegree;  //最大度
    FibonacciHeapNode** cons;            //指向最大度的内存区域
};
```

2. 初始化斐波那契堆

```
//初始化一个空的Fibonacci Heap
FibHeap* FibHeapMake() {
    FibHeap* heap = NULL;
    heap         = (FibHeap*)malloc(sizeof(FibHeap));
    if (NULL == heap) {
        puts("Out of Space!!");
        exit(1);
    }
    memset(heap, 0, sizeof(FibHeap));
    return heap;
}
```

初始化后通过`FibHeapInsertKey`(先新建一个节点，然后将key插入该节点，最后插入堆中)完成数据插入:

```
//堆结点x插入fibonacci heap中
int FibHeapInsert(FibHeap* heap, FibNode* x) {
    if (0 == heap->keyNum) {
        heap->min = x;
    } else {
        FibNodeAdd(x, heap->min);
        x->parent = NULL;
        if (x->key < heap->min->key) {
            heap->min = x;
        }
    }
    heap->keyNum++;
    return heap->keyNum;
}
//将值插入Fibonacci Heap
static int FibHeapInsertKey(FibHeap* heap, int key) {
    FibNode* x = NULL;
    x          = FibHeapNodeMake();
    x->key     = key;
    FibHeapInsert(heap, x);
    return heap->keyNum;
}
```

3. 抽取最小节点

   因为建立的斐波那契堆中已经存储了最小节点，因此只需要抽取该节点就行，删除其孩子并附加到原先的堆上

```
//抽取最小结点
FibNode* FibHeapExtractMin(FibHeap* heap) {
    FibNode *x = NULL, *z = heap->min;
    if (z != NULL) {
        //删除z的每一个孩子
        while (NULL != z->child) {
            x = z->child;
            FibNodeRemove(x);
            if (x->right == x) {
                z->child = NULL;
            } else {
                z->child = x->right;
            }
            FibNodeAdd(x, z);  // add x to the root list heap
            x->parent = NULL;
        }

        FibNodeRemove(z);
        if (z->right == z) {
            heap->min = NULL;
        } else {
            heap->min = z->right;
            FibHeapConsolidate(heap);
        }
        heap->keyNum--;
    }
    return z;
}
```

4. 减少一个节点的key

```
//减小一个关键字
void FibHeapDecrease(FibHeap* heap, FibNode* x, int key) {
    FibNode* y = x->parent;
    if (x->key < key) {
        puts("new key is greater than current key!");
        exit(1);
    }
    x->key = key;

    if (NULL != y && x->key < y->key) {
        //破坏了最小堆性质，需要进行级联剪切操作
        FibHeapCut(heap, x, y);
        FibHeapCascadingCut(heap, y);
    }
    if (x->key < heap->min->key) {
        heap->min = x;
    }
}
```

5. 删除节点

先找到该节点(提前完成)，然后将该节点的设为最小，然后抽取即可

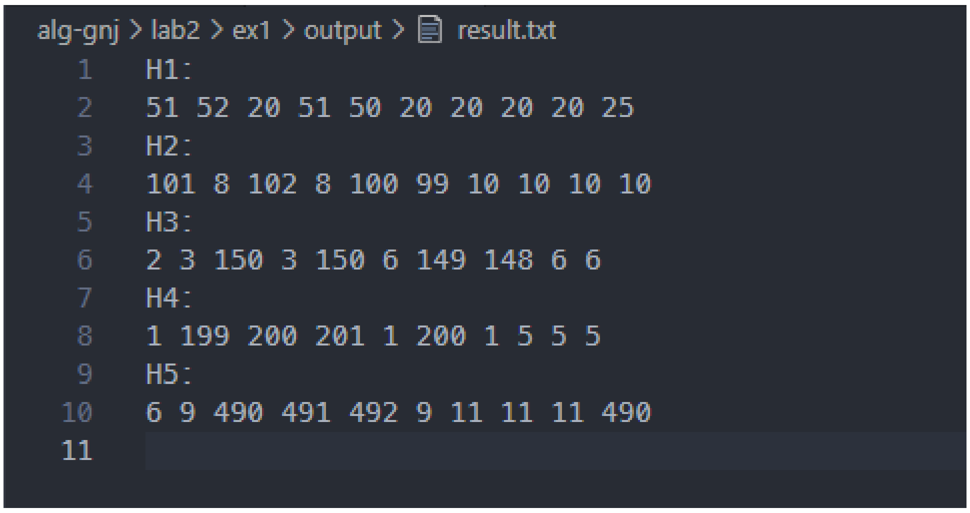
```
//删除结点
int FibHeapDelete(FibHeap* heap, FibNode* x) {
    FibHeapDecrease(heap, x, INT_MIN);
    FibHeapExtractMin(heap);
    return heap->keyNum;
}
```

6. 集合合并

```
//合并堆
FibHeap* FibHeapUnion(FibHeap* heap1, FibHeap *heap2){
    FibHeap *tmp;
    if(heap1 ==NULL){
        return heap2;
    }
    if(heap2 ==NULL){
        return heap1;
    }
    if(heap2->maxNumOfDegree> heap1->maxNumOfDegree){
        tmp = heap1;
        heap1 = heap2;
        heap2 = tmp;
    }
    if(heap1->min ==NULL){
        heap1->min = heap2->min;
        heap1->keyNum = heap2->keyNum;
        free(heap2->cons);
        free(heap2);
    }
    else if(heap2->min==NULL){
        free(heap2->cons);
        free(heap2);
    }
    else{
        FibNodeAdd(heap1->min, heap2->min);
        if(heap1->min->key>heap2->min->key){
            heap1->min = heap2->min;
        }
        heap1->keyNum +=heap2->keyNum;
        free(heap2->cons);
        free(heap2);
    }
    return heap1;
}
```

结果与分析

1. 代码运行结果



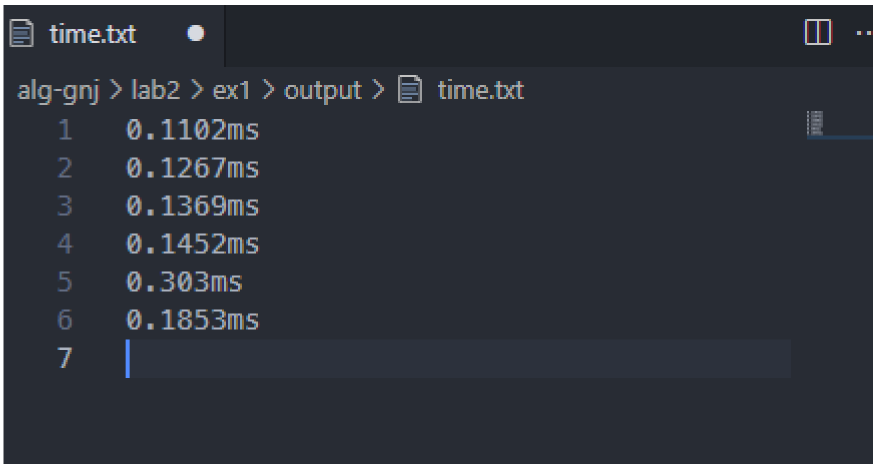
alg-gnj > lab2 > ex1 > output > result.txt

```
H1:
51 52 20 51 50 20 20 20 20 25
H2:
101 8 102 8 100 99 10 10 10 10
H3:
2 3 150 3 150 6 149 148 6 6
H4:
1 199 200 201 1 200 1 5 5 5
H5:
6 9 490 491 492 9 11 11 11 490

```

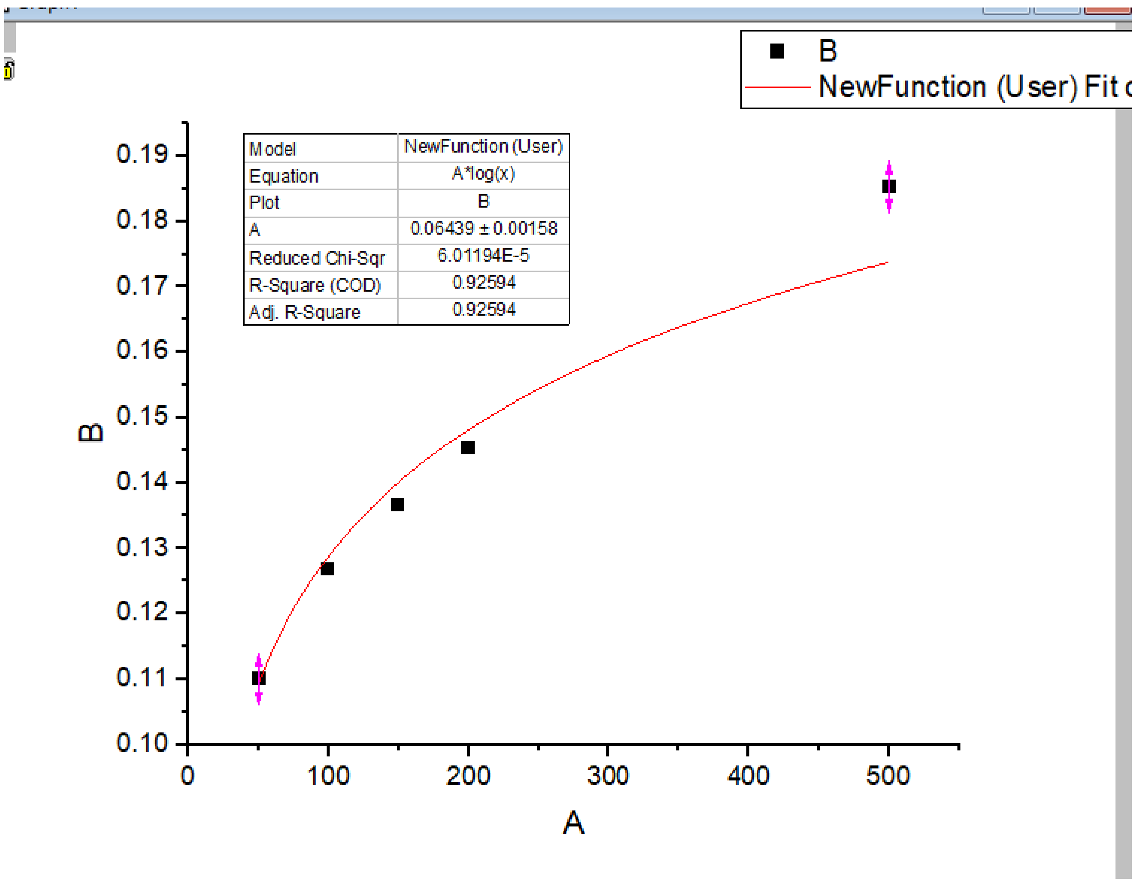
符合预期实验结果

2. 实验时间



```
0.1102ms
0.1267ms
0.1369ms
0.1452ms
0.303ms
0.1853ms
```

进行函数拟合可得



Model	NewFunction (User)
Equation	A*log(x)
Plot	B
A	0.06439 ± 0.00158
Reduced Chi-Sqr	6.01194E-5
R-Square (COD)	0.92594
Adj. R-Square	0.92594

EX2 家族数

实验内容

- 通过并查集解决查找家族数问题
- 实现按秩合并，路径压缩两种优化手段

方法与步骤

1. 建立并查集

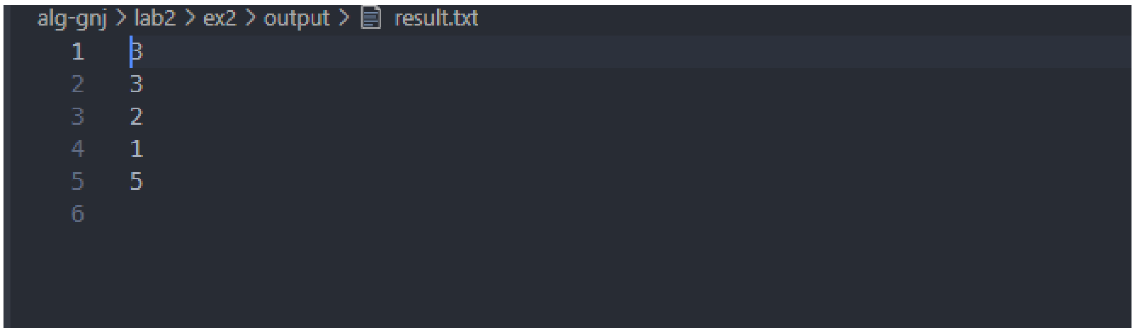
```
private:
    vector<int> parent;
    vector<int> rank;  // 秩
```

2. 实现优化

```
public:
    DisjSet(int max_size)
        : parent(vector<int>(max_size)), rank(vector<int>(max_size, 0)) {
        for (int i = 0; i < max_size; ++i)
            parent[i] = i;
    }
    // 查找根节点
    int find(int x) {
        return x == parent[x] ? x : (parent[x] = find(parent[x]));
    }
    //合并 同时进行路径优化和按秩合并
    void to_union(int x1, int x2) {
        int f1 = find(x1);
        int f2 = find(x2);
        if (rank[f1] > rank[f2])
            parent[f2] = f1;
        else {
            parent[f1] = f2;
            if (rank[f1] == rank[f2])
                ++rank[f2];
        }
    }
    bool is_same(int e1, int e2) {
        return find(e1) == find(e2);
    }
};
```

结果与分析

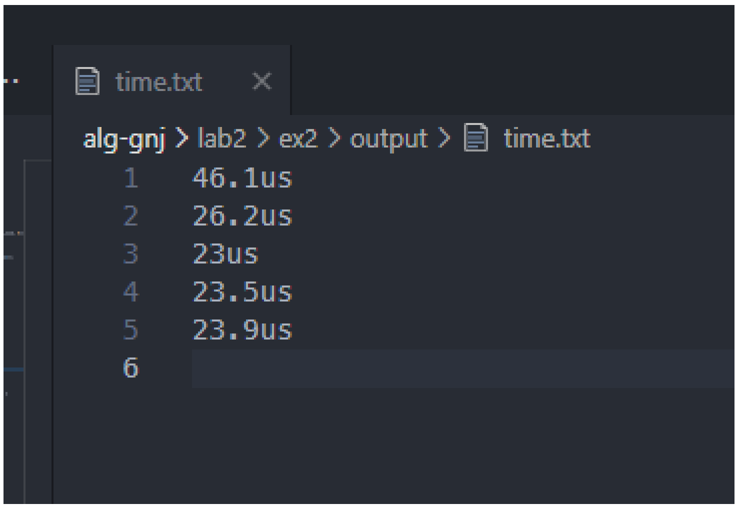
1. 运行结果



alg-gnj > lab2 > ex2 > output > result.txt

```
3
3
2
1
5
```

2. 时间分析



time.txt

alg-gnj > lab2 > ex2 > output > time.txt

```
46.1us
26.2us
23us
23.5us
23.9us
```

根据输出的时间来看(第一个多次运行时间仍然远大于其他时间，估计是初始化的时间过长导致，微秒级太短)，运行时间复杂度为$O(1)$

源代码

- EX1

```
#include <chrono>
#include <climits>
#include <cmath>
#include <cstdlib>
#include <cstring>
#include <filesystem>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

using namespace std;
using namespace filesystem;
using namespace chrono;

//斐波那契结点ADT
struct FibonacciHeapNode {
    int                 key;     //结点
    int                 degree;  //度
    FibonacciHeapNode* left;    //左兄弟
    FibonacciHeapNode* right;   //右兄弟
    FibonacciHeapNode* parent;  //父结点
    FibonacciHeapNode* child;   //第一个孩子结点
    bool                marked;  //是否被删除第1个孩子
};

typedef FibonacciHeapNode FibNode;

//斐波那契堆ADT
struct FibonacciHeap {
    int                  keyNum;          //堆中结点个数
    FibonacciHeapNode*  min;             //最小堆，根结点
    int                  maxNumOfDegree;  //最大度
    FibonacciHeapNode** cons;            //指向最大度的内存区域
};

typedef FibonacciHeap FibHeap;

//将x从双链表移除
inline void FibNodeRemove(FibNode* x) {
    x->left->right = x->right;
    x->right->left = x->left;
}

//将x堆结点加入y结点之前(循环链表中)
void FibNodeAdd(FibNode* x, FibNode* y) {
    x->left        = y->left;
    y->left->right = x;
    x->right       = y;
    y->left        = x;
```

```
}

//初始化一个空的Fibonacci Heap
FibHeap* FibHeapMake() {
    FibHeap* heap = NULL;
    heap         = (FibHeap*)malloc(sizeof(FibHeap));
    if (NULL == heap) {
        puts("Out of Space!!");
        exit(1);
    }
    memset(heap, 0, sizeof(FibHeap));
    return heap;
}

//初始化结点x
FibNode* FibHeapNodeMake() {
    FibNode* x = NULL;
    x          = (FibNode*)malloc(sizeof(FibNode));
    if (NULL == x) {
        puts("Out of Space!!");
        exit(1);
    }
    memset(x, 0, sizeof(FibNode));
    x->left = x->right = x;
    return x;
}

//将堆的最小结点移出，并指向其有兄弟
static FibNode* FibHeapMinRemove(FibHeap* heap) {
    FibNode* min = heap->min;
    if (heap->min == min->right) {
        heap->min = NULL;
    } else {
        FibNodeRemove(min);
        heap->min = min->right;
    }
    min->left = min->right = min;
    return min;
}

//将x根结点链接到y根结点
void FibHeapLink(FibHeap* heap, FibNode* x, FibNode* y) {
    FibNodeRemove(x);
    if (NULL == y->child) {
        y->child = x;
    } else {
        FibNodeAdd(x, y->child);
    }
    x->parent = y;
    y->degree++;
    x->marked = false;
}

//堆结点x插入fibonacci heap中
int FibHeapInsert(FibHeap* heap, FibNode* x) {
    if (0 == heap->keyNum) {
        heap->min = x;
    } else {
        FibNodeAdd(x, heap->min);
        x->parent = NULL;
        if (x->key < heap->min->key) {
            heap->min = x;
        }
    }
    heap->keyNum++;
    return heap->keyNum;
}

//将值插入Fibonacci Heap
static int FibHeapInsertKey(FibHeap* heap, int key) {
    FibNode* x = NULL;
    x          = FibHeapNodeMake();
    x->key     = key;
    FibHeapInsert(heap, x);
    return heap->keyNum;
}

//将数组内的值插入Fibonacci Heap
void FibHeapInsertKeys(FibHeap* heap, int keys[], int keyNum) {
    for (int i = 0; i < keyNum; i++) {
        FibHeapInsertKey(heap, keys[i]);
    }
}

//开辟FibHeapConsolidate函数哈希所用空间
static void FibHeapConsMake(FibHeap* heap) {
    int old                 = heap->maxNumOfDegree;
    heap->maxNumOfDegree = int(log(heap->keyNum * 1.0) / log(2.0)) + 1;
    if (old < heap->maxNumOfDegree) {
        //因为度为heap->maxNumOfDegree可能被合并,所以要maxNumOfDegree + 1
        heap->cons = (FibNode**)realloc(
            heap->cons, sizeof(FibHeap*) * (heap->maxNumOfDegree + 1));
        if (NULL == heap->cons) {
            puts("Out of Space!");
            exit(1);
        }
    }
}

//合并左右相同度数的二项树
void FibHeapConsolidate(FibHeap* heap) {
    int      D, d;
    FibNode *w = heap->min, *x = NULL, *y = NULL;
    FibHeapConsMake(heap);  //开辟哈希所用空间
    D = heap->maxNumOfDegree + 1;
    for (int i = 0; i < D; i++) {
        *(heap->cons + i) = NULL;
    }

    //合并相同度的根节点，使每个度数的二项树唯一
    while (NULL != heap->min) {
        x = FibHeapMinRemove(heap);
        d = x->degree;
        while (NULL != *(heap->cons + d)) {
            y = *(heap->cons + d);
            if (x->key > y->key) {  //根结点key最小
```

```
                swap(x, y);
            }
            FibHeapLink(heap, y, x);
            *(heap->cons + d) = NULL;
            d++;
        }
        *(heap->cons + d) = x;
    }
    heap->min = NULL;  //原有根表清除

    //将heap->cons中结点都重新加到根表中，且找出最小根
    for (int i = 0; i < D; i++) {
        if (*(heap->cons + i) != NULL) {
            if (NULL == heap->min) {
                heap->min = *(heap->cons + i);
            } else {
                FibNodeAdd(*(heap->cons + i), heap->min);
                if ((*(heap->cons + i))->key < heap->min->key) {
                    heap->min = *(heap->cons + i);
                }  // if(<)
            }      // if-else(==)
        }          // if(!=)
    }              // for(i)
}

//抽取最小结点
FibNode* FibHeapExtractMin(FibHeap* heap) {
    FibNode *x = NULL, *z = heap->min;
    if (z != NULL) {
        //删除z的每一个孩子
        while (NULL != z->child) {
            x = z->child;
            FibNodeRemove(x);
            if (x->right == x) {
                z->child = NULL;
            } else {
                z->child = x->right;
            }
            FibNodeAdd(x, z);  // add x to the root list heap
            x->parent = NULL;
        }

        FibNodeRemove(z);
        if (z->right == z) {
            heap->min = NULL;
        } else {
            heap->min = z->right;
            FibHeapConsolidate(heap);
        }
        heap->keyNum--;
    }
    return z;
}

//修改度数
void renewDegree(FibNode* parent, int degree) {
    parent->degree -= degree;
    if (parent->parent != NULL) {
```

```c
        renewDegree(parent->parent, degree);
    }
}

//切断x与父节点y之间的链接，使x成为一个根
static void FibHeapCut(FibHeap* heap, FibNode* x, FibNode* y) {
    FibNodeRemove(x);
    renewDegree(y, x->degree);
    if (x == x->right) {
        y->child = NULL;
    } else {
        y->child = x->right;
    }
    x->parent = NULL;
    x->left = x->right = x;
    x->marked          = false;
    FibNodeAdd(x, heap->min);
}

//级联剪切
static void FibHeapCascadingCut(FibHeap* heap, FibNode* y) {
    FibNode* z = y->parent;
    if (NULL != z) {
        if (y->marked == false) {
            y->marked = true;
        } else {
            FibHeapCut(heap, y, z);
            FibHeapCascadingCut(heap, z);
        }
    }
}

//减小一个关键字
void FibHeapDecrease(FibHeap* heap, FibNode* x, int key) {
    FibNode* y = x->parent;
    if (x->key < key) {
        puts("new key is greater than current key!");
        exit(1);
    }
    x->key = key;

    if (NULL != y && x->key < y->key) {
        //破坏了最小堆性质，需要进行级联剪切操作
        FibHeapCut(heap, x, y);
        FibHeapCascadingCut(heap, y);
    }
    if (x->key < heap->min->key) {
        heap->min = x;
    }
}

//删除结点
int FibHeapDelete(FibHeap* heap, FibNode* x) {
    FibHeapDecrease(heap, x, INT_MIN);
    FibHeapExtractMin(heap);
    return heap->keyNum;
}
```

```c
//被FibHeapSearch调用
static FibNode* FibNodeSearch(FibNode* x, int key) {
    FibNode *w = x, *y = NULL;
    if (x != NULL) {
        do {
            if (w->key == key) {
                y = w;
                break;
            } else if (NULL != (y = FibNodeSearch(w->child, key))) {
                break;
            }
            w = w->right;
        } while (w != x);
    }
    return y;
}

//堆内搜索关键字
FibNode* FibHeapSearch(FibHeap* heap, int key) {
    return FibNodeSearch(heap->min, key);
}

//被FibHeapDestory调用
static void FibNodeDestory(FibNode* x) {
    FibNode *p = x, *q = NULL;
    while (p != NULL) {
        FibNodeDestory(p->child);
        q = p;
        if (p->left == x) {
            p = NULL;
        } else {
            p = p->left;
        }
        free(q->right);
    }
}

//销毁堆
void FibHeapDestory(FibHeap* heap) {
    FibNodeDestory(heap->min);
    free(heap);
    heap = NULL;
}

//合并堆
FibHeap* FibHeapUnion(FibHeap* heap1, FibHeap *heap2){
    FibHeap *tmp;
    if(heap1 ==NULL){
        return heap2;
    }
    if(heap2 ==NULL){
        return heap1;
    }
    if(heap2->maxNumOfDegree> heap1->maxNumOfDegree){
        tmp = heap1;
        heap1 = heap2;
        heap2 = tmp;
    }
    if(heap1->min ==NULL){
        heap1->min = heap2->min;
        heap1->keyNum = heap2->keyNum;
        free(heap2->cons);
        free(heap2);
    }
    else if(heap2->min==NULL){
        free(heap2->cons);
        free(heap2);
    }
    else{
        FibNodeAdd(heap1->min, heap2->min);
        if(heap1->min->key>heap2->min->key){
            heap1->min = heap2->min;
        }
        heap1->keyNum +=heap2->keyNum;
        free(heap2->cons);
        free(heap2);
    }
    return heap1;
}


int main() {
    path p0(__FILE__);
    auto p  = p0.parent_path().parent_path();
    auto p1 = p0.parent_path().parent_path();
    auto p2 = p0.parent_path().parent_path();
    p2 += "\\output\\time.txt";
    p1 += "\\output\\result.txt";
    p += "\\input\\2_1_input.txt";
    cout << "Loading File from " << p << endl;
    ifstream file;
    file.open(p, ios::in);
    int         i = 0;
    vector<int> input;
    while (i < 500) {
        int number = 0;
        file >> number;
        input.push_back(number);
        i++;
    }

    int key1[50], key2[100], key3[150], key4[200];
    for (i = 0; i < 50; i++) {
        key1[i] = input[i];
    }
    for (; i < 150; i++) {
        key2[i - 50] = input[i];
    }
    for (; i < 300; i++) {
        key3[i - 150] = input[i];
    }
    for (; i < 500; i++) {
        key4[i - 300] = input[i];
    }
    FibHeap *heap1, *heap2, *heap3, *heap4;
    FibNode* x = NULL;
```

```cpp
    heap1       = FibHeapMake();
    heap2       = FibHeapMake();
    heap3       = FibHeapMake();
    heap4       = FibHeapMake();
    FibHeapInsertKeys(heap1, key1, 50);
    FibHeapInsertKeys(heap2, key2, 100);
    FibHeapInsertKeys(heap3, key3, 150);
    FibHeapInsertKeys(heap4, key4, 200);
    cout << "Finish building FibHeap" << endl << endl;

    ofstream timefile, outputfile;
    timefile.open(p2, ios::out);
    outputfile.open(p1, ios::out);

    // heap1
    steady_clock::time_point t1 = steady_clock::now();
    outputfile << "H1:" << endl
               << FibHeapInsertKey(heap1, 249) << " "
               << FibHeapInsertKey(heap1, 830) << " ";
    outputfile << heap1->min->key << " ";
    x = FibHeapSearch(heap1, 127);
    FibHeapDelete(heap1, x);
    outputfile << heap1->keyNum << " ";
    x = FibHeapSearch(heap1, 141);
    FibHeapDelete(heap1, x);
    outputfile << heap1->keyNum << " " << heap1->min->key << " ";
    x = FibHeapSearch(heap1, 75);
    FibHeapDecrease(heap1, x, 61);
    outputfile << heap1->min->key << " ";
    x = FibHeapSearch(heap1, 198);
    FibHeapDecrease(heap1, x, 169);
    outputfile << heap1->min->key << " ";
    x = FibHeapExtractMin(heap1);
    outputfile << x->key << " ";
    x = FibHeapExtractMin(heap1);
    outputfile << x->key << " " << endl;
    steady_clock::time_point t2    = steady_clock::now();
    auto                     time1 = t2 - t1;
    timefile << duration<double, milli>(time1).count() << "ms" << endl;

    // heap2
    t1 = steady_clock::now();
    outputfile << "H2: " << endl
               << FibHeapInsertKey(heap2, 816) << " " << heap2->min->key <<
" "
               << FibHeapInsertKey(heap2, 345) << " ";
    x = FibHeapExtractMin(heap2);
    outputfile << x->key << " ";
    x = FibHeapSearch(heap2, 504);
    outputfile << FibHeapDelete(heap2, x) << " ";
    x = FibHeapSearch(heap2, 203);
    outputfile << FibHeapDelete(heap2, x) << " ";
    x = FibHeapSearch(heap2, 296);
    FibHeapDecrease(heap2, x, 87);
    outputfile << heap2->min->key << " ";
    x = FibHeapSearch(heap2, 278);
    FibHeapDecrease(heap2, x, 258);
    outputfile << heap2->min->key << " " << heap2->min->key << " ";
    x = FibHeapExtractMin(heap2);
    outputfile << x->key << endl;
    t2    = steady_clock::now();
    time1 = t2 - t1;
    timefile << duration<double, milli>(time1).count() << "ms" << endl;

    t1 = steady_clock::now();
    x  = FibHeapExtractMin(heap3);
    outputfile << "H3: " << endl
               << x->key << " " << heap3->min->key << " "
               << FibHeapInsertKey(heap3, 262) << " ";
    x = FibHeapExtractMin(heap3);
    outputfile << x->key << " " << FibHeapInsertKey(heap3, 830) << " "
               << heap3->min->key << " ";
    x = FibHeapSearch(heap3, 134);
    outputfile << FibHeapDelete(heap3, x) << " ";
    x = FibHeapSearch(heap3, 177);
    outputfile << FibHeapDelete(heap3, x) << " ";
    x = FibHeapSearch(heap3, 617);
    FibHeapDecrease(heap3, x, 360);
    outputfile << heap3->min->key << " ";
    x = FibHeapSearch(heap3, 889);
    FibHeapDecrease(heap3, x, 353);
    outputfile << heap3->min->key << endl;
    t2    = steady_clock::now();
    time1 = t2 - t1;
    timefile << duration<double, milli>(time1).count() << "ms" << endl;

    //heap4
    t1 = steady_clock::now();
    x = FibHeapSearch(heap4, 708);
    outputfile<<"H4: "<<endl<<heap4->min->key<<" "<<FibHeapDelete(heap4, x)
<<" "<<FibHeapInsertKey(heap4, 281)<<" "<<FibHeapInsertKey(heap4, 347)<<" "
<<heap4->min->key<<" ";
    x = FibHeapSearch(heap4, 415);
    outputfile<<FibHeapDelete(heap4, x)<<" ";
    x = FibHeapExtractMin(heap4);
    outputfile<<x->key<<" ";
    x = FibHeapSearch(heap4, 620);
    FibHeapDecrease(heap4, x, 354);
    outputfile<<heap4->min->key<<" ";
    x = FibHeapSearch(heap4, 410);
    FibHeapDecrease(heap4, x, 80);
    outputfile<<heap4->min->key<<" ";
    x = FibHeapExtractMin(heap4);
    outputfile<<x->key<<endl;
    t2 = steady_clock::now();
    time1 = t2 - t1;
    timefile << duration<double, milli>(time1).count() << "ms" << endl;

    //step5
    t1 = steady_clock::now();
    auto heap5 = FibHeapUnion(heap1, heap2);
    heap5 = FibHeapUnion(heap5, heap3);
    heap5 = FibHeapUnion(heap5, heap4);
    cout<<"Heap Union step complete"<<endl;
    t2 = steady_clock::now();
    time1 = t2 - t1;
    timefile << duration<double, milli>(time1).count() << "ms" << endl;

    //step6
    t1 = steady_clock::now();
    x = FibHeapExtractMin(heap5);
    outputfile<<"H5: "<<endl<<x->key<<" "<<heap5->min->key<<" ";
    x = FibHeapSearch(heap5, 800);
    outputfile<<FibHeapDelete(heap5, x)<<" "<<FibHeapInsertKey(heap5, 267)
<<" "<<FibHeapInsertKey(heap5, 351)<<" ";
    x = FibHeapExtractMin(heap5);
    outputfile<<x->key<<" ";
    x = FibHeapSearch(heap5, 478);
    FibHeapDecrease(heap5, x, 444);
    outputfile<<heap5->min->key<<" ";
    x = FibHeapSearch(heap5, 559);
    FibHeapDecrease(heap5, x, 456);
    outputfile<<heap5->min->key<<" "<<heap5->min->key<<" ";
    x = FibHeapSearch(heap5, 929);
    outputfile<<FibHeapDelete(heap5, x)<<endl;
    return 0;
}
```

- EX2

```
#include <algorithm>
#include <chrono>
#include <filesystem>
#include <fstream>
#include <iostream>
#include <vector>

using namespace std;
using namespace chrono;
using namespace filesystem;

class DisjSet {
  private:
    vector<int> parent;
    vector<int> rank;  // 秩

  public:
    DisjSet(int max_size)
        : parent(vector<int>(max_size)), rank(vector<int>(max_size, 0)) {
        for (int i = 0; i < max_size; ++i)
            parent[i] = i;
    }
    // 查找根节点
    int find(int x) {
        return x == parent[x] ? x : (parent[x] = find(parent[x]));
    }
    //合并 同时进行路径优化和按秩合并
    void to_union(int x1, int x2) {
        int f1 = find(x1);
        int f2 = find(x2);
        if (rank[f1] > rank[f2])
```

```
            parent[f2] = f1;
        else {
            parent[f1] = f2;
            if (rank[f1] == rank[f2])
                ++rank[f2];
        }
    }
    bool is_same(int e1, int e2) {
        return find(e1) == find(e2);
    }
};

int BuildSet(int number, int size, vector<int> input) {
    DisjSet* set;
    set         = new DisjSet(size);
    int number1 = 0;
    for (int i = 1; i < size; i++) {
        for (int j = 0; j < i; j++) {
            number1 = number + size * i + j;
            if (input[number1] == 1) {
                set->to_union(i, j);
            }
        }
    }
    vector<int> vec;
    for (int i = 0; i < size; i++) {
        int value = set->find(i);
        if (find(vec.begin(), vec.end(), value) == vec.end()) {
            vec.push_back(value);
        }
    }
    return vec.size();
}

int main() {
    path p0(__FILE__);
    auto p1 = p0.parent_path().parent_path();
    auto p2 = p0.parent_path().parent_path();
    auto p3 = p0.parent_path().parent_path();
    p1 +="\\input\\2_2_input.txt";
    p2 +="\\output\\result.txt";
    p3 +="\\output\\time.txt";
    cout << "Loading File from " << p1 << endl;
    ifstream file;
    file.open(p1, ios::in);
    vector<int> N = {10, 15, 20, 25, 30};
    vector<int> input;
    int         number = 0;
    while (!file.eof()) {
        file >> number;
        input.push_back(number);
    }
    file.close();
    number = 0;
    ofstream outfile, timefile;
    outfile.open(p2, ios::out);
    timefile.open(p3, ios::out);
    for(int i=0;i<N.size();i++){
        steady_clock::time_point t1 = steady_clock::now();
        outfile<<BuildSet(number, N[i], input)<<endl;
        steady_clock::time_point t2    = steady_clock::now();
        auto                     time = t2 - t1;
        timefile << duration<double, micro>(time).count() << "us" << endl;
        number +=N[i]*N[i];
    }
}
```