实验目标

实现斐波拉契堆并与二叉堆比较效率（Dijkstra算法）

设计思路

二叉堆

数据存储设计

~~基本和之前写过的二叉堆一样~~

- 用数组来存储堆中元素
- 数组下标为元素的序号
  - 左儿子为2*i+1,右儿子为2*i+2

```
template <class T>
class MinHeap{
    private:
        T *mHeap;        // 数据
        int mCapacity;    // 总的容量
        int mSize;        // 实际容量

    private:
        // 最小堆的向下调整算法
        void filterdown(int start, int end);
        // 最小堆的向上调整算法(从start开始向上直到0，调整堆)
        void filterup(int start);
    public:
        MinHeap();
        MinHeap(int capacity);
        ~MinHeap();

        int getIndex(T data);// 返回data在二叉堆中的索引
        int remove(T data);// 即extract_min
        int insert(T data);// 将data插入到二叉堆中
        int decrease_key(T pos,T data);
        int empty();
        T find_min();
};
```

函数实现

- decrease_key
    - 该元素存在的情况下，将元素key值减小，再从该元素开始向上调整（调用filterup)

```
template <class T>
int MinHeap<T>::decrease_key(T pos,T data){
    int index = getIndex(pos);
    if(index == -1)return -1;
    if(mHeap[index] <= data)return -1;
    mHeap[index] = data;
    filterup(index);
    return 0;
}
```

- filterdown
    - 根据左右儿子的有无分类讨论
        - 右儿子不存在时，只比较左儿子
        - 右儿子存在时，选左右儿子中最小的，与父节点比较判断是否需要交换并进一步向下调整

```
template <class T>
void MinHeap<T>::filterdown(int start, int end)
{
    T temp;
    int left = 2*start + 1;
    int right = 2*start + 2;
    if(left > end)return;
    if(right > end)
    {
        if(mHeap[left] < mHeap[start])
        {
            temp = mHeap[left];
            mHeap[left] = mHeap[start];
            mHeap[start] = temp;
            filterdown(left,end);
            return;
        }else{
            return;
        }
    }else{
        if(mHeap[right] <= mHeap[left]&&mHeap[right] < mHeap[start])
        {
            temp = mHeap[right];
            mHeap[right] = mHeap[start];
            mHeap[start] = temp;
            filterdown(right,end);
            return;
        }else if(mHeap[left] < mHeap[right]&&mHeap[left] < mHeap[start])
```

```
        {
            temp = mHeap[left];
            mHeap[left] = mHeap[start];
            mHeap[start] = temp;
            filterdown(left,end);
            return;
        }
        else
        {
            return;
        }

    }
}
```

- filterup
  - 不断的比较当前节点与其父节点的key值大小来判断是否需要调整，直到当前节点key值>=父节点的key值为止或者是当前节点是root

```
template <class T>
void MinHeap<T>::filterup(int start){
    int father = (start - 1 ) / 2;
    while(father >= 0 && mHeap[father] > mHeap[start] )
    {
        T temp = mHeap[father];
        mHeap[father] = mHeap[start];
        mHeap[start] = temp;
        start = father;
        father = (start - 1) / 2;
    }
}
```

斐波那契堆

基本按照给的模板完成，略作修改，将map<T, Node*> mp换作vector<Node*>mp;

并在一些函数的参数中作了修改。

节点设置

```
struct Node {
    T key;
    int degree;
    bool mark;
    Node *p, *child, *left, *right;
    Node(T k) : key(k), degree(0), mark(false) {
      p = child = nullptr;
      left = right = this;
    }
  };
```

数据存储

用Node* Min来寻找根链表

用vector<Node*>mp来保证在O（1）时间内寻找到某节点

n记录节点个数

```
template <class T>
class Fibonacci_Heap {
private:
  struct Node {
    T key;
    int degree;
    bool mark;
    Node *p, *child, *left, *right;
    Node(T k) : key(k), degree(0), mark(false) {
      p = child = nullptr;
      left = right = this;
    }
  };

  Node *Min;
  int n;
  //map<T, Node*> mp;
    vector<Node*>mp;
  void Del_Tree(Node *root);
  void Consolidate();
  void Link(Node *y, Node *x);
  void Cut(Node *x, Node *y);
  void Cascading_Cut(Node *y);
public:
  Fibonacci_Heap();
  ~Fibonacci_Heap();

  void Push(int id,T x);
  bool Empty();
  T Top();
```

```
    void Pop();
    void Decrease_Key(int id, T k);
};
```

函数实现

- Push
    - 将x放在mp中，为了以后实现decrease_key
    - 将x插入到根链表中
    - 更新Min

```
template <class T>
void Fibonacci_Heap<T>::Push(int id,T x) {
    while(id >= mp.size()){
        mp.push_back(nullptr);
    }
    Node* point = new Node(x);
    mp[id] = point;

    if(n == 0){
        Min = point;
    }else{
        Node* tmp = Min->left;
        tmp->right = point;
        Min->left = point;
        point->left = tmp;
        point->right = Min;
        if(Min->key > point->key)
        {
            Min = point;
        }
    }
    n++;
}
```

- Pop
    - 先判断是否有最小节点可以pop
    - 只有一个节点，pop掉它就好
    - 有多个节点，pop以后要维护斐波那契堆堆性质
        - 将Min的每一个子节点添加到根链表中
        - 将Min移除
        - 重置Min节点
        - 调用consolidate

```
template <class T>
void Fibonacci_Heap<T>::Pop() {

    if(n == 0) return;
    n--;
    if(n == 0) {
        delete Min;
        Min = nullptr;
        return;
    }
    Node *tmp = Min -> child;
    vector <Node *> chdlist;
    if(tmp != nullptr)
        do{
            chdlist.push_back(tmp);
            tmp = tmp -> right;
        }while(tmp != Min -> child);
    for(int i = 0; i < chdlist.size(); i++){
        Node *iterat = chdlist[i];
        Node *Mleft = Min -> left;
        iterat -> p = nullptr;
        Mleft -> right = iterat;    Min -> left = iterat;
        iterat -> left = Mleft;     iterat -> right = Min;
    }
    Node *l = Min -> left;
    Node *r = Min -> right;
    l -> right = r;
    r -> left = l;
    delete Min;
    Min = l;
    Consolidate();
}
```

- Decrease_key
    - 判断是否可以使用此操作
    - 调用cut和cascading-cut来维护斐波那契堆的性质
    - 更新Min

```
template<class T>
void Fibonacci_Heap<T>::Decrease_Key(int id, T k) {

    if(id>=mp.size()||mp[id] == nullptr){
        cout<<"The target  doesn't exit"<<endl;
        return;
    }
    if(mp[id]->key < k){
```

```
        cout<<"The key of target  is higher than you thought"
<<endl;return;
    }
    Node* target = mp[id];
    target->key = k;
    Node* fa = target->p;
    if(fa!=nullptr && target->key < fa->key)
    {
        Cut(target,fa);
        Cascading_Cut(fa);
    }
    if(target->key < Min->key)
        Min = target;
}
```

- Consolidate
    - 用vector<Node*>A来按照度的大小存储根节点中的元素
        - 几处的while循环是为了保证vector足够大，能够满足度数不会大于A.size()

        ```
        while(cur->degree+1 > A.size())
                { A.push_back(nullptr); }
        ```

    - 将根链表中每一个元素取出，放在root_listl里面
    - 用for循环实现 对每一个root_list中元素进行判断：是否有元素具有与它相同的degree
        - 若具有相同的degree,合并两个元素，调用Link将key值较小的节点插在Key值较大节点的子链表中；插入后可能新生成的节点与其他节点degree相同，故重复此过程直到不相同为止。（while实现）
    - 用A中的元素创建一个新的根链表

```
template <class T>
void Fibonacci_Heap<T>::Consolidate() {
    //cout<<"consolidate"<<endl;
    vector<Node*>root_list;
    vector<Node*>A;
    Node* cur = Min->right;
    root_list.push_back(Min);
    while(cur != Min)
    {
        while(cur->degree+1 > A.size())
        { A.push_back(nullptr); }
        root_list.push_back(cur);
        cur = cur->right;
    }

    for(int i = 0;i < root_list.size();i++)
```

```
    {
        Node* x = root_list[i];
        int d = x->degree;
        while(d + 10>A.size()){A.push_back(nullptr);}
        //cout<<d<<endl;
        //cout<<A.size()<<endl;
        while(A[d] != nullptr)
        {
            Node* y = A[d];
            if(x->key > y->key)
            {
                Node* swap = x;
                x = y;
                y = swap;
            }
            Link(y,x);
            A[d] = nullptr;
            d++;
        }
        while(d+5 > A.size()){ A.push_back(nullptr);}
        A[d] = x;
    }
    Min = nullptr;
    for(int j = 0;j < A.size();j++)
    {
        if(A[j]!=nullptr){
            if(Min == nullptr)
            {
                Min = A[j];
                Min->left = Min;Min->right = Min;
            }
            else
            {

                Node* le = Min->left;
                le -> right = A[j];
                Min -> left = A[j];
                A[j]->right = Min;
                A[j]->left = le;
                if(Min->key > A[j]->key)Min = A[j];
            }

        }
    }
}
```

- Link

    - 将y从根链表中移除
    - 将y添加到x的子节点中，跟新x的degree
    - 置y的mark为false

```
template <class T>
void Fibonacci_Heap<T>::Link(Node *y, Node *x) {
    //remove y from the root list
    Node* l = y->left;
    Node* r = y->right;
    l->right = r;
    r->left = l;

    //make y a child of x, incrementing x.degree
    x->degree++;
    y->p = x;
    y->mark = false;
    if(x->child != nullptr){
        Node* t = x->child->right;
        t->left = y;
        y->left = x->child;
        y->right = t;
        x->child->right = y;
    }else
    {
        x->child = y;
        y->left = y;y->right = y;
    }

}
```

- Cut
    - 将x从y的子链表中移除，更新y的degree
    - 将x添加到根链表中
    - 跟新x的p和mark

```
template <class T>
void Fibonacci_Heap<T>::Cut(Node *x, Node *y) {
    y->degree--;


    //remove x from child list of y
    Node* tmp = x;
    if(y->degree == 0){y->child = nullptr;}
    else{
        if(y->child == x)
            y->child = x->right;
```

```
        Node* left = x->left;
        Node* right = x->right;
        right->left = left;
        left->right = right;

    }

    //add x to the root list
    x->p = nullptr;
    x->mark = false;
    Node* temp = Min->left;
        temp->right = x;
        Min->left = x;
        x->left = temp;
        x->right = Min;
}
```

- Cascading_Cut
  - y的mark未被标记
    - 标记y
  - y的mark被标记
    - 递归查看y的父节点
      - cut（y，z）
      - cascadign-cut（z）

```
template <class T>
void Fibonacci_Heap<T>::Cascading_Cut(Node *y) {
    Node* z = y->p;
    if(z!=nullptr){
        if(y->mark == false)
        {
            y->mark = true;
        }else
        {
            Cut(y,z);
            Cascading_Cut(z);
        }

    }
```

实验结果截图

```
[329] 0->643->390->998
[488] 0->643->88->495->999
-----------------
Time binary heap: 0.209971 s
Time fibonacci heap: 0.048829 s
```

总结

斐波那契堆的时间复杂度小于二项堆的时间复杂度。但是斐波那契堆的编程复杂性远高于二项堆的编程复杂性。