

实验11 斐波拉契堆

姓名：邵宁录

学号：2018202195

目录

1. 问题描述
2. 算法基本思路
3. 算法复杂度分析
4. 源码
5. 运行结果截图
6. 问题与总结

一、问题描述

实现斐波拉契堆，至少包括插入，合并，抽取最小点操作。

二、算法基本思路

算法基本思路基本参考算法导论第19章的内容，在细节上略有修改。

插入

插入操作主要分为以下几个步骤：

1. 根据传入的值 x 新建一个结点。
2. 若堆原本为空，则将 Min 指向它即可。
3. 若堆原本不为空：
 1. 先将该结点插入 Min 结点的左边
 2. 再比较新结点与 Min 结点的 key 的大小，若新结点比较小，则将 Min 指向新结点。
4. 堆的结点数量 $+1$

合并

合并操作主要分为以下几个情况：

1. 若两个堆都是空的，则合并后仍然是个空堆。
2. 若只有 A 是空堆，则合并后直接变为 B 。
3. 若只有 B 是空堆，则合并后直接变为 A 。
4. 若都不为空：
 1. 先将当前的 Min 指向 A 的 Min 。
 2. 然后将 B 的根结点全部加入 A 的左端。
 3. 最后比较 A, B 的 key 的大小，若 B 的更小，则将 Min 指向 B 。
5. 最后将当前结点数 n 更新一下。

抽取最小点

抽取最小点主要分为以下几个情况：

1. 若堆为空，则返回 $null$ 。
2. 若堆内只有一个结点。则就返回当前结点，并删除它。
3. 其他情况时，先将 Min 的 $child$ 加入根链表，然后将其 Min 指向当前右边的结点。最后 $Consolidate$ 。

三、算法复杂度分析

使用摊还分析中的势能法分析其复杂度：首先定义整个斐波那契堆的势能函数为： $\Phi(H) = t(H) + 2m(H)$ 。

其中 $t(H)$ 为 H 中根链表中的节点数， $m(H)$ 为 H 中已标记节点的和。

在 $|H| = 0$ 时， $\Phi(H) = 0$ ，同时在 $|H| > 0$ 时恒有 $\Phi(H) > 0$ ，因此摊还代价 $\sum \hat{c}_i = \sum (c_i + \Phi(H_i) - \Phi(H_{i-1}))$ 永远是实际代价的上界。

接下来分析每一个操作的摊还代价：

1. 插入操作

$$\begin{aligned}t(H') &= t(H) + 1 \\m(H') &= m(H) \\ \Delta\Phi(H) &= (t(H') + 2m(H')) - (t(H) + 2m(H)) = 1\end{aligned}$$

又因为链表本身的插入操作代价为 $O(1)$ ，于是摊还代价为 $O(1) + 1 = O(1)$ ；

2. 合并操作

$$\begin{aligned}t(H) &= t(H1) + t(H2) \\m(H) &= m(H1) + m(H2) \\ \Delta\Phi &= \Phi(H) - (\Phi(H1) + \Phi(H2)) = (t(H) + 2m(H)) - (t(H1) + 2m(H1) + t(H2) + 2m(H2)) = 0\end{aligned}$$

由于链表本身的合并操作代价为 $O(1)$ ，于是摊还代价为 $O(1) + 0 = O(1)$ ；

3. 提取最小操作

$$\begin{aligned}\Phi(H) &= t(H) + 2m(H) \\ \Phi(H') &\leq D(n) + 1 + 2m(H) \\ \Delta\Phi &\leq D(n) - t(H)\end{aligned}$$

因为 $D(n) = O(\lg n)$ ，又因为链表本身的抽取操作代价为 $O(D(n) + t(H))$ ，所以抽取最小节点的摊还代价为 $O(D(n) + t(H)) - t(H) = O(D(n)) = O(\lg n)$ 。

四、源码

Fibonacci.h

```
/*
 * @Description: 算法导论第19章斐波那契堆实现
 * @Author: rainym00d
 * @Github: https://github.com/rainym00d
 * @Date: 2020-11-25 21:54:13
 * @LastEditors: rainym00d
 * @LastEditTime: 2020-11-27 14:16:14
 */
#pragma once

#include <vector>
#include <cstring>
#include <algorithm>
#include <map>
#include <iostream>
using namespace std;

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;
    // 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();

    bool Empty();
    T Top();

    void Insert(T x);
    T ExtractMin();
    // void Decrease_Key(T k);
    void HeapUnion(Fibonacci_Heap <T> &A, Fibonacci_Heap <T> &B);
};

```

```

template <class T>
Fibonacci_Heap<T>::Fibonacci_Heap() {
    this->Min = nullptr;
    this->n = 0;
}

```

```

template <class T>
Fibonacci_Heap<T>::~~Fibonacci_Heap() {
    // this->mp.clear();
    Node *ptr = this->Min;
    if (ptr == nullptr) return;
    do {
        Del_Tree(ptr);
        ptr = ptr -> right;
    } while(ptr != this->Min);
}

```

```

template <class T>
void Fibonacci_Heap<T>::Del_Tree(Node *root) {
    if (root->child != nullptr) {
        Node *ptr = root->child;
        do {
            this->Del_Tree(ptr);
            ptr = ptr->right;
        } while(ptr != root->child);
    }
    delete root;
}

```

```

template <class T>
bool Fibonacci_Heap<T>::Empty() {
    if (this->n == 0)
        return true;
    return false;
}

```

```

template <class T>
T Fibonacci_Heap<T>::Top() {
    return this->Min->key;
}

```

```

template <class T>
void Fibonacci_Heap<T>::Link(Node *y, Node *x) {
    // 把y从根链表中移除
    Node* l = y->left;

```

```

Node* r = y->right;
l->right = r;
r->left = l;
// 把y作为x的子结点, 增加x.degree
x->degree ++;
y->p = x;
y->mark = false;
if (x->child != nullptr) {
    Node* t = x->child->right;
    t->left = y;
    y->right = t;
    y->left = x->child;
    x->child->right = y;
}
else {
    x->child = y;
    y->left = y;
    y->right = y;
}
}
}

```

```

template <class T>
void Fibonacci_Heap<T>::Consolidate() {
    vector<Node*> root_list;
    vector<Node*> A;
    Node* cur = this->Min->right;
    root_list.push_back(this->Min);
    while (cur != this->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);
        }
        while (A[d] != nullptr) {
            Node* y = A[d];
            if (x->key > y->key) {
                Node* tmp = x;
                x = y;
                y = tmp;
            }
            this->Link(y, x);
            A[d] = nullptr;
            d++;
        }
        while (d + 5 > A.size()) {
            A.push_back(nullptr);
        }
        A[d] = x;
    }
    this->Min = nullptr;
    for (int i = 0; i < A.size(); i++) {
        if (A[i] != nullptr) {
            if (this->Min == nullptr) {
                this->Min = A[i];
                this->Min->left = this->Min;
                this->Min->right = this->Min;
            }
            else {
                Node* Min_left = this->Min:

```

```

        Min_left->right = A[i];
        this->Min->left = A[i];
        A[i]->left = Min_left;
        A[i]->right = this->Min;
        if (A[i]->key < this->Min->key) {
            this->Min = A[i];
        }
    }
}
}

```

```

template <class T>
void Fibonacci_Heap<T>::Insert(T x) {
    // while (id >= this->mp.size()) {
    //     this->mp.push_back(nullptr);
    // }
    Node* point = new Node(x);
    // this->mp[id] = point;

    // 当堆为空时
    if (this->Empty()) {
        this->Min = point;
    }
    // 当堆不为空时
    else {
        // 把新插入的结点放到Min结点的左边
        Node* tmp = this->Min->left;
        tmp->right = point;
        this->Min->left = point;
        point->left = tmp;
        point->right = this->Min;
        // 如果新结点的key比较小, 那就将Min指向它
        if (point->key < this->Min->key) {
            this->Min = point;
        }
    }
    // 结点数量+1
    this->n ++;
}

```

```

template <class T>
T Fibonacci_Heap<T>::ExtractMin() {
    // 如果是空
    if (this->Empty()) {
        return -1;
    }
    // 如果只有一个
    this->n --;
    if (this->Empty()) {
        // cout << "in " << endl;
        T ans = this->Min->key;
        // cout << "11 " << ans << endl;
        // printf("%p\n", &Min);
        delete this->Min;
        // cout << "111" << endl;
        this->Min = nullptr;
        return ans;
    }
    // 将子结点加入到根链表
    Node* tmp = this->Min->child;
    vector <Node *> child_list;
    if (tmp != nullptr) {
        do {
            child_list.push_back(tmp);
            tmp = tmp->right;
        } while (tmp != nullptr);
    }
}

```

```

        } while (tmp != this->Min->child);
    }
    for (auto &child : child_list) {
        Node* Min_left = this->Min->left;
        child->p = nullptr;
        Min_left->right = child;
        this->Min->left = child;
        child->left = Min_left;
        child->right = this->Min;
    }
    // 将Min从根链表中删除
    Node* Min_left = this->Min->left;
    Node* Min_right = this->Min->right;
    Min_left->right = Min_right;
    Min_right->left = Min_left;
    T ans = this->Min->key;
    delete this->Min;
    this->Min = Min_left;
    // printf("%p %p\n", &Min, &Min_left);
    // cout << "left " << Min_left->key << endl;
    this->Consolidate();
    return ans;
}

template<class T>
void Fibonacci_Heap<T>::HeapUnion(Fibonacci_Heap<T> &A, Fibonacci_Heap<T> &B) {
    // 若两个堆都是空的, 则合并后仍是个空堆
    if (A.Min == nullptr && B.Min == nullptr) {
        return;
    }
    // 若只有A是空的, 则直接变成B
    if (A.Min == nullptr) {
        this->Min = B.Min;
    }
    // 只有B是空的, 则直接变成A
    else if (B.Min == nullptr) {
        this->Min = A.Min;
    }
    // 若都不空
    else {
        // 先将Min指向A.Min
        this->Min = A.Min;
        // 先将B的结根点全部插入A的跟链表
        Node* tmp = B.Min;
        vector<Node*> root_list;
        do {
            // cout << "aaa " << tmp->key << endl;
            root_list.push_back(tmp);
            tmp = tmp->right;
        } while (tmp != B.Min);

        for (auto &root : root_list) {
            Node* Min_left = this->Min->left;
            Min_left->right = root;
            this->Min->left = root;
            root->left = Min_left;
            root->right = this->Min;
        }
        // 若B.Min比A.Min的key小, 则把Min指向B.Min
        if (B.Min->key < A.Min->key) {
            this->Min = B.Min;
        }
    }
    // 更新结点数量
    this->n = A.n + B.n;
}

```

main.cpp

```
/*
 * @Description:
 * @Author: rainym00d
 * @Github: https://github.com/rainym00d
 * @Date: 2020-11-27 12:20:03
 * @LastEditors: rainym00d
 * @LastEditTime: 2020-11-27 13:52:59
 */

#include "Fibonacci.h"

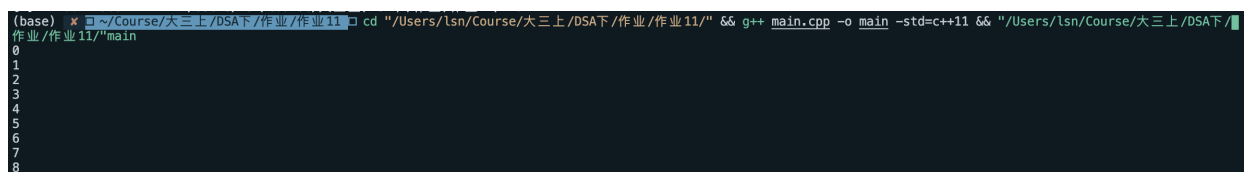
using namespace std;

int main(int argc, char const *argv[])
{
    Fibonacci_Heap <int> A;
    Fibonacci_Heap <int> B;
    Fibonacci_Heap <int> H;
    for (int i = 0; i < 5; i++) {
        A.Insert(2 * i);
        B.Insert(2 * i + 1);
    }
    H.HeapUnion(A, B);
    for (int i = 0; i < 9; i++) {
        cout << H.ExtractMin() << endl;
    }
    return 0;
}
```

五、运行结果截图

操作描述：

1. 对 A 插入了 0, 2, 4, 6, 8
2. 对 B 插入了 1, 3, 5, 7
3. 最后再对他们合并，并执行提取最小操作将其全部输出



```
(base) * ~ /Course/大三上/DSA下/作业/作业11/ cd "/Users/lsn/Course/大三上/DSA下/作业/作业11/" && g++ main.cpp -o main -std=c++11 && "/Users/lsn/Course/大三上/DSA下/
作业/作业11/"main
0
1
2
3
4
5
6
7
8
```

六、问题与总结

总的来说，本次实验难度较大，原因在于斐波那契堆其本身的数据结构设计就很抽象难以理解。但其有着优越的性质，是非常值得学习的一个数据结构！