Санкт-Петербургский политехнический университет Петра Великого
Институт металлургии, машиностроения и транспорта
Высшая школа автоматизации и робототехники

К У Р С О В А Я Р А Б О Т А

Фибоначчиева куча

по дисциплине «Объектно-ориентированное программирование»

Выполнил: студент гр. 3331506/70401 Елисеев В.В.
Преподаватель: Ананьевский М.С.

«___» _____ 2021 г.

Санкт-Петербург
2021

Содержание

1. Введение

Фибоначчиева куча — структура данных, отвечающая интерфейсу приоритетная очередь. Эта структура данных имеет меньшую амортизированную сложность, чем такие приоритетные очереди как биномиальная куча и двоичная куча.

Кучи являются основной структурой данных во многих приложениях.

Они применяются: при сортировке элементов; в алгоритмах выбора, для поиска минимума или максимума, медианы; в алгоритмах на графах, в частности, при построении минимального остовного дерева алгоритмом Крускала, при нахождении кратчайшего пути алгоритмом Дейкстры.

Недостатки:

- Большое потребление памяти на узел
- Большая константа времени работы, что делает ее малоприменимой для реальных задач
- Некоторые операции в худшем случае могут работать за $O(n)$ времени

Достоинства:

- Одно из лучших асимптотических времен работы для всех операций

2. Описание

Фибоначчиева куча является структурой данных для хранения данных позволяющей быстро производить следующие операции: добавление элемента, получение минимального элемента, удаление минимального элемента, уменьшение ключа по ссылке и удаление по ссылке. Данная структура организована следующим образам:

1) Существует явная ссылка на минимальный элемент.

2) У каждой вершины есть ссылка на правый и левый элемент в двусвязном списке содержащим эту вершину.

3) У каждой вершины есть ссылка *child* указывающая на одну из вершин спика ее детей.

4) У каждой вершины есть ссылка *parent* указывающая на родителя.

5) У каждой вершины есть булевское поле *marked* использующаяся при уменьшении ключа. Оно истинно если вершина потеряла ребенка после того, как сделалась чьим-нибудь ребенком.

6) Список содержащей минимальную вершину называется корневым списком, и родители всех его вершин отсутствуют.

3. Реализация фибоначчиевой кучи

Для возможности быстрого удаления элемента из произвольного места и объединением с другим списком будем хранить их в циклическом двусвязном списке. Также будем хранить и все уровни поддерева. Исходя из этого структура каждого узла будет выглядеть вот так.

```
struct node {
    node* parent; // указатель на родительский узел
    node* child; // указатель на один из дочерних узлов
    node* left; // указатель на левый узел того же предка
    node* right; // указатель на правый узел того же предка
    int key; // ключ
    int degree; // Степень узла
    char mark; // Отметка узла
    char c; // Флаг для помощи в функции поиска узла
};

struct node* mini = NULL; //Создание указателя на мин значение
int size = 0; //Объявить целое число для количества узлов в куче
```

Рисунок 4.1 – Создание структуры

Чтобы не дублировать код, объясню на словах, как действуют определенные функции.

Insertion()

Вставка элемента. Данная операция вставляет новый элемент в список корней правее минимума и при необходимости меняет указатель на минимум кучи.

Fibonnaci_link()

Для сливания двух Фибоначчиевых куч необходимо просто объединить их корневые списки, а также обновить минимум новой кучи, если понадобится. Вынесем в вспомогательную функцию Fibonnaci_link логику, объединяющую два списка вершины, которых подаются ей в качестве аргументов.

Consolidate()

Сливаем два корневых списка в один и обновляем минимум, если нужно.

Extract_min()

Удаление минимального элемента. Первая рассматриваемая операция, в ходе которой значительно меняется структура кучи. Здесь используется вспомогательная процедура consolidate, благодаря которой, собственно, и достигается желанная амортизированная оценка.

Cut()

Вырезание. При вырезании вершины мы удаляем ее из списка детей своего родителя, уменьшаем степень ее родителя и снимаем пометку с текущей вершины.

Cascase_cut()

Каскадное вырезание. Перед вызовом каскадного вырезания нам известно, удаляли ли ребенка у этой вершины. Если у вершины до этого не удаляли дочерний узел, то мы помечаем эту вершину и прекращаем выполнение операции. В противном случае применяем операцию cut для текущей вершины и запускаем каскадное вырезание от родителя.

Decrease_key()

Функция для уменьшения значения узла в куче.

Find()

Как бы странно это не звучало, но это функция необходима для поиска определенного узла, с целью его последующего удаления или доступа.

Deletion()

Удаление вершины реализуется через уменьшение ее ключа до 0 и последующим извлечением минимума.

display()

Показывает все элементы в куче в выстроенном порядке.

find_min()

Показывает минимум.

interview()

Функция, которая опрашивает пользователя. Некоторые функции изначально недоступны пользователю, с целью снижения загруженности интерфейса.

4. Время работы

Для анализа производительности операций введем потенциал для фибоначчиевой кучи как $\Phi = trees + 2 \bullet marked$, где *trees* — количество элементов в корневом списке кучи, а *marked* — количество вершин, у которых удален один ребенок (то есть вершин с пометкой *x.mark = true*). Договоримся, что единицы потенциала достаточно для оплаты константного количества работы.

Вставка элемента

Для оценки амортизированной стоимости операции рассмотрим исходную кучу H и получившуюся в результате вставки нового элемента кучу H'. $trees(H') = trees(H) + 1$ и $marked(H') = marked(H)$. Следовательно, увеличение потенциала составляет $(trees(H) + 1 + 2 \bullet marked(H)) - (trees(H) + 2 \bullet marked(H)) = 1$. Так как реальное время работы составляет $O(1)$, то амортизированная стоимость данной операции также равна $O(1)$.

Получение минимального элемента

Истинное время работы — $O(1)$.

Соедининение двух куч

Реальное время работы — $O(1)$. Амортизированное время работы также $O(1)$, поскольку, при объединении двух куч в одну, потенциалы обеих куч суммируются, итоговая сумма потенциалов не изменяется, $\Phi_{n+1} - \Phi_n = 0$.

Таблица 1 – Время выполнения различных операций

Операция	Амортизированная сложность
makeHeap	$O(1)$
getMin	$O(1)$
merge	$O(1)$
extractMin	$O(\log(n))$
decreaseKey	$O(1)$
delete	$O(\log(n))$
insert	$O(1)$

5. Список литературы

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн — Алгоритмы: построение и анализ. — М.: Издательский дом «Вильямс», 2005. — С. 1296. — ISBN 5-8459-0857-4
- Mehlhorn, Kurt, Sanders, Peter. 6.2.2 Fibonacci Heaps // Algorithms and Data Structures: The Basic Toolbox. — Springer, 2008. — 300 с. — ISBN 978-3-540-77978-0.
- Fibonacci Heaps — Duke University
- Fibonacci Heaps — Princeton University

Приложение А

```
#include<iostream>
#include <math.h>

using namespace std;

struct node {
	node* parent; // указатель на родительский узел
	node* child; // указатель на один из дочерних узлов
	node* left; // указатель на левый узел того же предка
	node* right; // указатель на правый узел того же предка
	int key; // ключ
	int degree; // Степень узла
	char mark; // Отметка узла
	char c; // Флаг для помощи в функции поиска узла
};

struct node* mini = NULL; //Создание указателя на мин значение
int size = 0; //Обьявить целое число для количества узлов в куче

static void insertion(int val) //Функция для вставки числа в кучу
{
	struct node* new_node = (struct node*)malloc(sizeof(struct
node)); // создаем новый узел
	new_node->key = val; // инициализируем ключ нового узла
	new_node->parent = NULL;
	new_node->child = NULL;
	new_node->left = new_node;
	new_node->right = new_node;
	if (mini != NULL) {
		(mini->left)->right = new_node;
		new_node->right = mini;
		new_node->left = mini->left;
		mini->left = new_node;
	if (new_node->key < mini->key)
		mini = new_node;
	} else {
		mini = new_node;
	}
	size++; // не забываем увеличить переменную size
}

static void Fibonnaci_link(struct node* ptr2, struct node* ptr1)
//настройка связи между родителем и дочерним узл.
{
	(ptr2->left)->right = ptr2->right;
```

```
	(ptr2->right)->left = ptr2->left;
	if (ptr1->right == ptr1)
		mini = ptr1;
	ptr2->left = ptr2;
	ptr2->right = ptr2;
	ptr2->parent = ptr1;
	if (ptr1->child == NULL)
		ptr1->child = ptr2;
	ptr2->right = ptr1->child;
	ptr2->left = (ptr1->child)->left;
	((ptr1->child)->left)->right = ptr2;
	(ptr1->child)->left = ptr2;
	if (ptr2->key < (ptr1->child)->key)
		ptr1->child = ptr2;
	ptr1->degree++;
}

static void Consolidate() //Объединение кучи
{
	int temp1;
	float temp2 = (log(size)) / (log(2));
	int temp3 = temp2;
	struct node* arr[temp3];
	for (int i = 0; i <= temp3; i++)
		arr[i] = NULL;
	node* ptr1 = mini;
	node* ptr2;
	node* ptr3;
	node* ptr4 = ptr1;
	do {
		ptr4 = ptr4->right;
		temp1 = ptr1->degree;
		while (arr[temp1] != NULL) {
			ptr2 = arr[temp1];
			if (ptr1->key > ptr2->key) {
				ptr3 = ptr1;
				ptr1 = ptr2;
				ptr2 = ptr3;
			}
			if (ptr2 == mini)
				mini = ptr1;
			Fibonnaci_link(ptr2, ptr1);
			if (ptr1->right == ptr1)
				mini = ptr1;
			arr[temp1] = NULL;
			temp1++;
		}
		arr[temp1] = ptr1;
```

```
                ptr1 = ptr1->right;
        } while (ptr1 != mini);
        mini = NULL;
        for (int j = 0; j <= temp3; j++) {
                if (arr[j] != NULL) {
                        arr[j]->left = arr[j];
                        arr[j]->right = arr[j];
                        if (mini != NULL) {
                                (mini->left)->right = arr[j];
                                arr[j]->right = mini;
                                arr[j]->left = mini->left;
                                mini->left = arr[j];
                                if (arr[j]->key < mini->key)
                                        mini = arr[j];
                        } else {
                                mini = arr[j];
                        }
                        if (mini == NULL)
                                mini = arr[j];
                        else if (arr[j]->key < mini->key)
                                mini = arr[j];
                }
        }
}

static void Extract_min() //Функция для извлечения минимального узла в
куче
{
        if (mini == NULL)
                cout << "The heap is empty" << endl;
        else {
                node* temp = mini;
                node* pntr;
                pntr = temp;
                node* x = NULL;
                if (temp->child != NULL) {
                        x = temp->child;
                        do {
                                pntr = x->right;
                                (mini->left)->right = x;
                                x->right = mini;
                                x->left = mini->left;
                                mini->left = x;
                                if (x->key < mini->key)
                                        mini = x;
                                x->parent = NULL;
                                x = pntr;
                        } while (pntr != temp->child);
```

```
                }
                (temp->left)->right = temp->right;
                (temp->right)->left = temp->left;
                mini = temp->right;
                if (temp == temp->right && temp->child == NULL)
                        mini = NULL;
                else {
                        mini = temp->right;
                        Consolidate();
                }
                size--;
        }
}

static void Cut(struct node* found, struct node* temp)
{
        if (found == found->right)
                temp->child = NULL;
        (found->left)->right = found->right;
        (found->right)->left = found->left;
        if (found == temp->child)
                temp->child = found->right;
        temp->degree = temp->degree - 1;
        found->right = found;
        found->left = found;
        (mini->left)->right = found;
        found->right = mini;
        found->left = mini->left;
        mini->left = found;
        found->parent = NULL;
        found->mark = 'B';
}

static void Cascase_cut(struct node* temp)
{
        node* ptr5 = temp->parent;
        if (ptr5 != NULL) {
                if (temp->mark == 'W') {
                        temp->mark = 'B';
                } else {
                        Cut(temp, ptr5);
                        Cascase_cut(ptr5);
                }
        }
}

static void Decrease_key(struct node* found, int val)
{
```

```
        if (mini == NULL)
                cout << "The Heap is Empty" << endl;
        if (found == NULL)
                cout << "Node not found in the Heap" << endl;
        found->key = val;
        struct node* temp = found->parent;
        if (temp != NULL && found->key < temp->key) {
                Cut(found, temp);
                Cascase_cut(temp);
        }
        if (found->key < mini->key)
                mini = found;
}

static void Find(struct node* mini, int old_val, int val)
{
        struct node* found = NULL;
        node* temp5 = mini;
        temp5->c = 'Y';
        node* found_ptr = NULL;
        if (temp5->key == old_val) {
                found_ptr = temp5;
                temp5->c = 'N';
                found = found_ptr;
                Decrease_key(found, val);
        }
        if (found_ptr == NULL) {
                if (temp5->child != NULL)
                        Find(temp5->child, old_val, val);
                if ((temp5->right)->c != 'Y')
                        Find(temp5->right, old_val, val);
        }
        temp5->c = 'N';
        found = found_ptr;
}

static void Deletion(int val)
{
        if (mini == NULL)
                cout << "The heap is empty" << endl;
        else {
                Find(mini, val, 0); // Уменьшение значения узла до 0
                Extract_min(); // Вызов функции Extract_min для удаления
узла минимального значения, равного 0
                cout << "Key Deleted" << endl;
        }
}
```

```
static void display(struct node* mini) //Показывает всю кучу
{
	node* ptr = mini;
	if (ptr == NULL)
		cout << "The Heap is Empty" << endl;
	else {
		cout << "The root nodes of Heap are: " << endl;
		do {
			cout << ptr->key;
			ptr = ptr->right;
			if (ptr != mini) {
				cout << "-->";
			}
		} while (ptr != mini && ptr->right != NULL);
		cout << endl << "The heap has " << size << " nodes" <<
endl;
	}
}

int find_min(struct node* mini) //Функция для поиска минимального узла
в куче
{
	cout << "min of heap is: " << mini->key << endl;
}

int interview(void) {
	int answer;
	int number;
	string bas[7] = {"What do you want?", "1)Add", "2)Find min",
"3)Delete min " ,"4)Show all", "5)Exit"};
	for (int i = 0; i < 7; i++) {
		cout << bas[i] << endl;
	 }
	cin >> answer;
	switch (answer)
	{
		case 1:
			cout << "What number do you want to add?" <<
endl;
			cin >> number;
			insertion(number);
			break;

		case 2:
			find_min(mini);
			break;

		case 3:
```

```
                        Extract_min();
                        break;
                case 5:
                        cout << "Goodbye))" << endl;
                        return 0;

                case 4:
                        display(mini);
                        break;
        }
}

int main()
{
        while (1) {
                if (interview() == 0){
                        return 0;}
        }
}
```