Fibonacci Heap

Penadillo Lazares Wenses Johan

UNIVERSIDAD NACIONAL DE INGENIERÍA

20 de septiembre de 2021

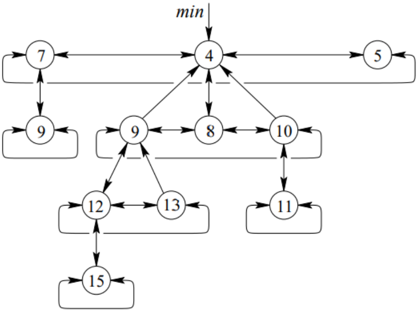
Índice

Explicación

El montón de Fibonacci es una estructura de datos que implementa el tipo de datos abstractos de cola de prioridad, al igual que el ordinario montón (heap) pero más complicado y asintóticamente más rápido para algunas operaciones.

Fibonacci heap

Es una colección de arboles de montones ordenados. Los hermanos y las raíces están organizados según una lista circular doblemente enlazada, cada nodo tiene un puntero hacia su padre y uno de sus hijos. Tambien almacenan una llave(valor), el grado y un bit que se usa para marcar y desmarcar el nodo.



Potential function

Se usara una función potencial para analizar el costo amortizado de un montón de fibonacci inicialmente vació. Sea r_i el numero de raíces en la lista circular de raíces y m_i el numero de nodos marcados. El potencial después de la i-esima operación es $\Phi_i = r_i + 2m_i$. Cuando trabajemos con una colección de montones de fibonacci, definiremos su potencial como la suma de sus potencias individuales. c_i sera el costo actual y $a_i = c_i + \Phi_i - \Phi_{i-1}$ el costo amortizado de la i-esima operación. Como $\Phi_0 = 0$ y $\Phi_i \geq 0$:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} a_i = r_n + 2m_n + \sum_{i=1}^{n} c_i$$

Delete min

1. Remover el nodo con el menor valor de la lista circular de raíces.
2. fusionar la lista circular de raíces con la lista circular de los hijos del nodo removido.
3. Si hay dos raíces con el mismo grado, las enlazaremos.
4. Recalcular el puntero al mínimo valor.

Delete min

Para analizar el costo amortizado de eliminar el mínimo. Sea D_n el máximo grado posible de un nodo en un montón de fibonacci de n nodos. El numero de operaciones de enlace en el paso 3 es el numero de raíces con las que empezamos, que es menor que $r_{i-1} + D(n)$, menos el numero de raíces con las que terminamos que es r_i. Después del paso 3, todas las raíces tendrán grados diferentes, lo que implica que $r_i \leq D(n) + 1$. Entonces el costo actual para los 4 pasos es $c_i \leq 2 + r_{i-1} + 2D(n) - r_i$. El cambio potencial es $\Phi_i - \Phi_{i-1} = r_i - r_{i-1}$. Entonces
$a_i = c_i + \Phi_i - \Phi_{i-1} \leq 2D(n) + 2$
Luego se puede probar que $D(n) < 2\log_2(n+1)$, lo que implica que eliminar el mínimo, tiene un costo amortizado logarítmico.

Decrease key and delete

La primera operación reemplaza la llave x almacenada en un nodo v por $x - \triangle$, donde $\triangle$ es un numero real no negativo.

1. Desvincular el arbol enraizado en v.
2. Dsiminuir la llave en v por $\triangle$.
3. Añadir v a la lista circular de raíces y posiblemente actualizar el puntero al mínimo.
4. Hacer cortes en cascada.

Para la segunda operación debemos revisar si $v = min$, realizaremos *delete min*, sino.

1. Desvincular el árbol enraizado en v.
2. fusionar la lista circular de raíces con la lista circular del los hijos de v.
3. Eliminar v.
4. Hacer cortes en cascada.

Cascading cuts

Sea v un nodo que se convierte en hijo de otro nodo en el tiempo t. Marcaremos v cuando pierda su primer hijo después del tiempo t. Luego desmarcaremos v,desenlazaremos y lo añadiremos a la lista circular de raíces cuando pierda a su segundo hijo. Esta operación se llamara corte y sera en cascada porque un corte puede causar otro y ese otro puede causar otro y así sucesivamente.

Análisis

El costo del paso 4 en *decrease key and delete* es el numero de cortes, c_i. El cambio potencial debido a que hay c_i nuevas raíces y c_i menos nodos marcados sera:
$\Phi_i - \Phi_{i-1} = r_i + 2m_i - r_{i-1} - 2m_{i-1} \leq c_i - 2c_i + 2 = -c_i + 2.$
El costo amortizado de $a_i = c_i + \Phi_i - \Phi_{i-1} \leq c_i - (2 - c_i) = 2.$
Los 3 primeros pasos de la operacion *decrease key* toman solo una cantidad constante de tiempo e incrementan el potencial en una cantidad constante. Entonces el costo amortizado de *decrease key* incluyendo el corte en cascada en el paso 4, es constante. Similarmente la operación de eliminar tiene un tiempo constante pero el paso 2 incrementa el potencial a lo mas $D(n)$. Lo que implica que el costo amortizado de la operación de eliminar es $O(\log n)$.

Resumen

Costos temporales de las operaciones:

encontrar el mínimo	$O(1)$
fusionar dos montones	$O(1)$
añadir un nuevo nodo	$O(1)$
eliminar el mínimo	$O(\log n)$
disminuir el valor de un nodo	$O(1)$
eliminar un nodo	$O(\log n)$

```python
class FibonacciHeap:

    # internal node class
    class Node:
        def __init__(self, data):
            self.data = data
            self.parent = self.child = self.left = self.right = None
            self.degree = 0
            self.mark = False

    # function to iterate through a doubly linked list
    def iterate(self, head):
        node = stop = head
        flag = False
        while True:
            if node == stop and flag is True:
                break
            elif node == stop:
                flag = True
            yield node
            node = node.right

    # pointer to the head and minimum node in the root list
```

```python
    root_list, min_node = None, None

    # maintain total node count in full fibonacci heap
    total_nodes = 0

    # return min node in O(1) time
    def find_min(self):
        return self.min_node

    # extract (delete) the min node from the heap in O(log n) time
    # amortized cost analysis can be found here (http://bit.ly/1ow1Clm)
    def extract_min(self):
        z = self.min_node
        if z is not None:
            if z.child is not None:
                # attach child nodes to root list
                children = [x for x in self.iterate(z.child)]
                for i in range(0, len(children)):
                    self.merge_with_root_list(children[i])
                    children[i].parent = None
            self.remove_from_root_list(z)
            # set new min node in heap
            if z == z.right:
                self.min_node = self.root_list = None
```

```
            else:
                self.min_node = z.right
                self.consolidate()
            self.total_nodes -= 1
        return z

    # insert new node into the unordered root list in O(1) time
    def insert(self, data):
        n = self.Node(data)
        n.left = n.right = n
        self.merge_with_root_list(n)
        if self.min_node is None or n.data < self.min_node.data:
            self.min_node = n
        self.total_nodes += 1

    # modify the data of some node in the heap in O(1) time
    def decrease_key(self, x, k):
        if k > x.data:
            return None
        x.data = k
        y = x.parent
        if y is not None and x.data < y.data:
            self.cut(x, y)
            self.cascading_cut(y)
```

```
        if x.data < self.min_node.data:
            self.min_node = x

    # merge two fibonacci heaps in O(1) time by concatenating the root
    # the root of the new root list becomes equal to the first list and
    # list is simply appended to the end (then the proper min node is d
    def merge(self, h2):
        H = FibonacciHeap()
        H.root_list, H.min_node = self.root_list, self.min_node
        # fix pointers when merging the two heaps
        last = h2.root_list.left
        h2.root_list.left = H.root_list.left
        H.root_list.left.right = h2.root_list
        H.root_list.left = last
        H.root_list.left.right = H.root_list
        # update min node if needed
        if h2.min_node.data < H.min_node.data:
            H.min_node = h2.min_node
        # update total nodes
        H.total_nodes = self.total_nodes + h2.total_nodes
        return H

    # if a child node becomes smaller than its parent node we
    # cut this child node off and bring it up to the root list
```

```
    def cut(self, x, y):
        self.remove_from_child_list(y, x)
        y.degree -= 1
        self.merge_with_root_list(x)
        x.parent = None
        x.mark = False

    # cascading cut of parent node to obtain good time bounds
    def cascading_cut(self, y):
        z = y.parent
        if z is not None:
            if y.mark is False:
                y.mark = True
            else:
                self.cut(y, z)
                self.cascading_cut(z)

    # combine root nodes of equal degree to consolidate the heap
    # by creating a list of unordered binomial trees
    def consolidate(self):
        A = [None] * self.total_nodes
        nodes = [w for w in self.iterate(self.root_list)]
        for w in range(0, len(nodes)):
            x = nodes[w]
```

```
            d = x.degree
            while A[d] != None:
                y = A[d]
                if x.data > y.data:
                    temp = x
                    x, y = y, temp
                self.heap_link(y, x)
                A[d] = None
                d += 1
            A[d] = x
        # find new min node - no need to reconstruct new root list belo
        # because root list was iteratively changing as we were moving
        # nodes around in the above loop
        for i in range(0, len(A)):
            if A[i] is not None:
                if A[i].data < self.min_node.data:
                    self.min_node = A[i]

    # actual linking of one node to another in the root list
    # while also updating the child linked list
    def heap_link(self, y, x):
        self.remove_from_root_list(y)
        y.left = y.right = y
        self.merge_with_child_list(x, y)
```

```python
            x.degree += 1
            y.parent = x
            y.mark = False

    # merge a node with the doubly linked root list
    def merge_with_root_list(self, node):
        if self.root_list is None:
            self.root_list = node
        else:
            node.right = self.root_list.right
            node.left = self.root_list
            self.root_list.right.left = node
            self.root_list.right = node

    # merge a node with the doubly linked child list of a root node
    def merge_with_child_list(self, parent, node):
        if parent.child is None:
            parent.child = node
        else:
            node.right = parent.child.right
            node.left = parent.child
            parent.child.right.left = node
            parent.child.right = node
```

```
    # remove a node from the doubly linked root list
    def remove_from_root_list(self, node):
        if node == self.root_list:
            self.root_list = node.right
        node.left.right = node.right
        node.right.left = node.left

    # remove a node from the doubly linked child list
    def remove_from_child_list(self, parent, node):
        if parent.child == parent.child.right:
            parent.child = None
        elif parent.child == node:
            parent.child = node.right
            node.right.parent = parent
        node.left.right = node.right
        node.right.left = node.left
```