0.1 Fibonacci Heap

- *Algorithm:* Construct and maintain a Fibonacci Heap
- *Input:* one Fibonacci Heap
- *Complexity:* $\mathcal{O}(\log n)$ for *extract_min* opearation and $\Theta(1)$ for other operations
- *Data structure compatibility:* Fibonacci Heap
- *Common applications:* Priority queue, Improvement for graph search algorithm like, Dijkstra Algorithm, A* Algorithm

Fibonacci Heap

Fibonacci heap is a data structure for priority queue operations. which is made up of a collection of rooted trees that are min-heap ordered. It has a better amortized running time than many other priority queues since most of the opearations in Fibonacci Heap take constant time.

Description

Operations of a Fibonacci Heap

Each element in a Fibonacci heap has a *key*. The heap supports the following seven opearations. [1]

- $\mathit{Make\_heap}()$ creates and then returns an empty new heap.
- $\mathit{Insert}(H, x)$ inserts an element x into the heap H.
- $\mathit{Minimum}(H)$ returns a pointer to the element in H with the minimum key.
- $\mathit{Union}(H_1, H_2)$ creates then returns a new heap that contains all elements of H_1 and H_2, with H_1,H_2 no longer used afterward.
- $\mathit{Decrease\_key}(H, x, k)$ assigns to element x within H a new key value k, where k should be no greater than x's current key value.
- $\mathit{Delete}(H, x)$ deletes element x from H

Amortized time complexity

Procedure	Amortized time complexity
Make_heap	$\Theta(1)$
Insert	$\Theta(1)$
Extract_min	$\mathcal{O}(\log n)$
Union	$\Theta(1)$
Decrease_key	$\Theta(1)$
Delete	$\mathcal{O}(\log n)$

Table 1: Running times for operations onFibonacci heap. The number of items in the heap(s) at the time of an operation is denoted by n.

Structure of Fibonacci heaps

An element in a Fibonacci heap has 7 atrributes. [1]

- key: the key of the element.
- value: value stored in the element
- left: Pointer that points to the element's right siblings. If this element is an only child, its left is itself.
- right: Pointer that points to the element's right siblings. If this element is an only child, its right is itself.
- parent: Pointer thatpoints to the element's parent. For element in the root list of a Fibonacci heap, its parent is None.
- child: Pointer that points to one of its child. Its children are linked together in a circular, doubly linked list, which is called the child list of x.
- mark: A mark modified in $Decrease\_key$ and used in potential calculation.

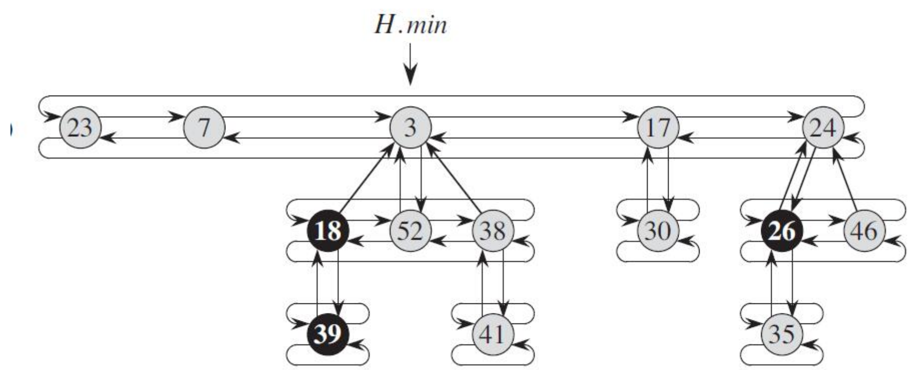

Figure 1: An example of a Fibonacci heap

Make an empty heap

Make an empty heap H is to set the $H.min$ to be None and the current size of H to be 0.

Insert a node

When a node is to be inserted, it is assumed to have already been allocated with its key already set.

Algorithm 1: Insert(x)

```
Input  : A node x to be inserted
Output: The minimum element in H and modified H
x.degree = 0
x.parent = None
x.child = None
x.mark = False
if H.min==None then
    create a root list for H with just x
    H.min = x
end if
else
    insert x into H's root list
    if x.key < H.min.key then
        H.min = x
    end if
end if
H.size+ = 1
```

Union two Fibonacci heaps

When uniting two Fibonacci heaps into one new Fibonacci heap, the things to do are simply concatenating the root lists of both Fibonacci heaps and then determining the new minimum node. After such union operations, the old two Fibonacci heaps will not be used any more.

Algorithm 2: Union(H_1, H_2)

```
Input  : Fibonacci heapH1, H2
Output: A new Fibonacci heap H, which is the union heap
H = make_heap()
H.min = H1.min
concatenate the root list of H2 with the root list of H
if H1.min == None or (H2.min ≠ None and H2.min, key < H1.min.key) then
    H.min = H2.min
end if
H.size = H1.size + H2.size
return H
```

Extract the minimum node

This process is the most complicated opearations of a Fibonacci heap. It is where the maintainence of heap property and the work of consolidating trees in the root list occurs.

Algorithm 3: Extrcact_min(H)

```
Input   : Fibonacci Heap H
Output: The minimum element in H and modified H
z = H.min
if z ≠ None then
    for each child of z do
        add x to the root list of H
        x.p = None
    end for
end if
remove z from the root list of H if z == z.right then
    H.min = None
end if
else
    H.min = z.right
    Consolidate(H)
end if
H.size− = 1
return z
```

The upper bound $D(n)$ on the degree of any node in an n-node Fibonacci heap meets that

$$D(n) \leq \lfloor \log_{\phi} n \rfloor$$

where ϕ is the golden ratio, defined as

$$\phi = \frac{\sqrt{5}+1}{2}$$

Algorithm 4: Consolidate(H)

```
Function Heap_link():
    remove y from the root list of H
    make y a child of x
    incrementing x.degree
    y.mark = FALSE
end
let A be a new array with size D(n)
Set each element of A None
for each node w in the root list of H do
    x = w
    d = x.degree
    while A[d] ≠ None do
        y = A[d] if x.key > y.key then
            exchange x with y
        end if
        Heap_link(y, x)
        A[d] = None d = d + 1
    end while
    A[d] = x
end for
H.min = None
for every element i in A do
    if A[i] ≠ None then
        if H.min == None then
            create a new root list of H that only contains i
            H.min = i
        end if
        else
            insert i into H root list
            if i.key < H.min.key then
                H.min=i
            end if
        end if
    end if
end for
```

Decrease a key and delete a node

Decrease_key opearation decreases the key of one certain node. All of such operations are done under the assumption that removing a node from a linked list won't change any of its structural atrributes.

What *Delete* does is decrease the key of the element to be deleted and call *Extract_min*. Assume that there is no key value of $-\infty$ currently in the Fibonacci heap.

There is a question that in pratical use we cannot actually find the address of or the pointer itself. The first solution I think is Ostrich Algorithm, that the probability of having to delete another element besides the minimum within the heap is very small since we have already decided to use Fibonacci heap rather than other heaps. A more relatively solution is to set up a dictionary for the Fibonacci heap, where the key of the dictionary is the value of all elements, the value of the dictionary is the element or the address of the element. So we can delete an element by its value.

Algorithm 5: Decrease_key(x, k)

Input : The target node x, the value to decrease k
Output: Nothing

```
Function Cut(H, x, y):
    remove x from the child list of y
    y.degree−= 1
    add x to the root list of H
    x.parent = Null
    x.mark = false
end
Function Cascading_cut(H, y):
    z = y.parent
    if z ≠ None then
        if y.mark == false then
            y.mark == true
        end if
        else
            Cut(H, y, z)
            Cascading_cut(H, z)
        end if
    end if
end
if k > x.key then
    this causes error.
end if
x.key = k
y = x.parent
if y ≠ None and x.key < y.key then
    Cut(H, x, y)
    Cascading_cut(H, y)
end if
if x.key < H.min.key then
    H.min = x
end if
```

Algorithm 6: Delete(H, x)

Input : Fibonacci heap$H$, the element to be deleted x
Output: Nothing

```
Decrease_key(H, x, −∞)
Extract_min(H)
```

Amortized Analysis

1. Notations and the potential function. The notations used in analysis is shown in the table below.

notation	meaning
n	number
rank(x)	number of children of node x
Extract_min	max rank of any node in heap H
trees(H)	number of trees in heap H
marks(H)	number of marked nodes in heap H

Table 2: Notations in amortized analysis

The potential function is [2]

$$\Phi(H) = \mathsf{trees}(H) + 2 \cdot \mathsf{marks}(H)$$

2. *Insert*:
The actual cost is $c_i = \mathcal{O}(1)$. Change in potential is $\Delta\Phi = 1$. The amortized cost is calculated as $\hat{c}_i = c_i + \Delta\Phi = \Theta(1)$.

3. *Extract_min*:
The actual cost is $c_i = \mathcal{O}(rank(H)) + \mathcal{O}(trees(H))$. Change in potential is $\Delta\Phi \leq \mathsf{rank}\left(H'\right) + 1 - \mathsf{trees}(H)$. The amortized cost is calculated as $\hat{c}_i = c_i + \Delta\Phi = \mathcal{O}(rank(H)) + \mathcal{O}\left(rank\left(H'\right)\right) = \mathcal{O}(\log n)$, where the rank of heap with size n is $\mathcal{O}(\log n)$.

4. *Decrease_key*:
The actual cost $c_i = \mathcal{O}(c)$, where c is the number of cuts. Change in potential is $\Delta\Phi \leq c+2\cdot(-c+2) = \mathcal{O}(1)-c$. The amortized cost is calculated as $\hat{c}_i = c_i + \Delta\Phi = \mathcal{O}(1)$

References.

[1] Leiserson Charles E. Cormen Thomas H. and etc. *Introduction to Algorithms – Third Edition.* The MIT Press, 2009 (cit. on pp. 1, 2).

[2] Xiaofeng Gao. *VE281 – Data Structure and Algorithms (lecture slides)*. 2019 (cit. on p. 7).