TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

———————— * ————————

BÀI TẬP LỚN

TÌM HIỂU VỀ FIBONACCI HEAP

Giảng viên hướng dẫn
TS. Đỗ Phan Thuận

Sinh viên thực hiện
Trần Phương Nam

HÀ NỘI
Ngày 23 tháng 12 năm 2018

Mục lục

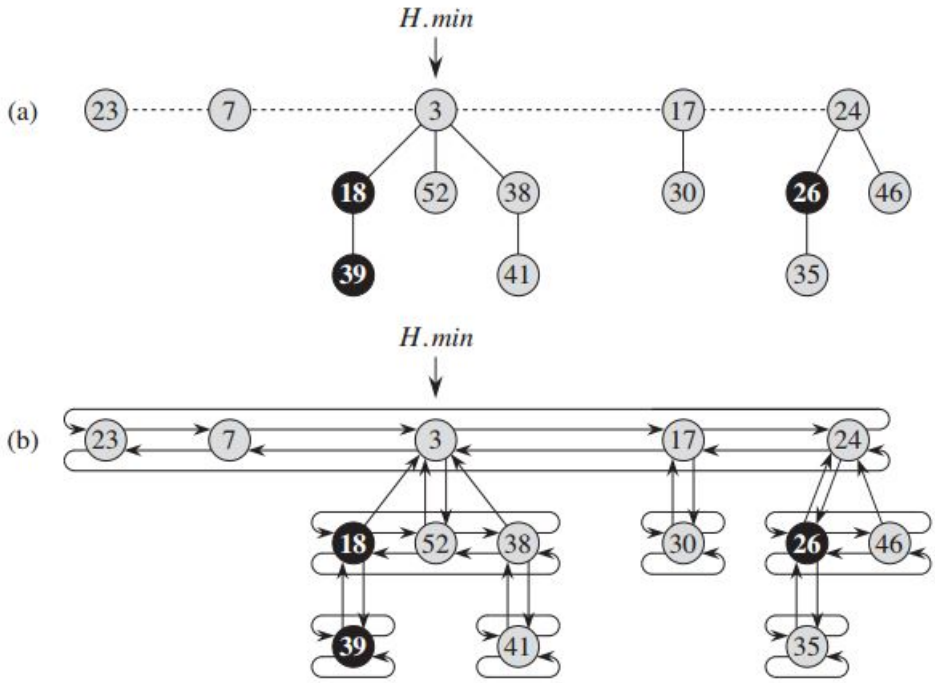
1 Giới thiệu chung

Fibonacci heap là một cấu trúc dữ liệu nâng cao, có nhiều ưu điểm so với các cấu trúc dữ liệu thông thường. Đầu tiên, đây là một cấu trúc heap (đống), là cấu trúc dữ liệu dựa trên cấu trúc cây và có tính chất: nếu B là nút con của A thì $key(B) \geq key(A)$ (đối với min-heap). Thứ hai, ưu điểm của Fibonacci heap so với các heap thông thường là nó cho phép thực hiện 1 số phép toán đặc trưng của heap trong thơi gian hằng số, điều này rất phù hợp với các ứng dụng mà có lời gọi đến các phép toán này thường xuyên.
Tuy nhiên, trên thực tế, để cài đặt fibonacci heap hoàn chỉnh là rất phức tạp. Điều này khiến cho fibonacci heap ít được ứng dụng nhiều trong thực tế, ngoài trường hợp quản lí số lượng lớn dữ liệu. Nếu có thể cải thiện, làm cấu trúc dữ liệu này đơn giản hơn mà vẫn giữ được các giới hạn tính toán thì fibonacci heap sẽ được ứng dụng nhiều hơn trong thực tế.

2 Cấu trúc của Fibonacci heap

2.1 Cấu trúc của Fibonacci heap

Một fibonacci heap là một tập các cây thoả mãn tính chất min-heap (khoá của nút con không bé hơn khoá của nút cha).



Hình 1: Một ví dụ về fibonacci heap

Hình trên là một ví dụ về fibonacci heap. Nó bao gồm 5 cây con, gốc của mỗi cây được trỏ tới nhau bởi danh sách liên kết 2 chiều, gọi là *root list* (danh sách gốc). Các node anh, em cũng được trỏ tới nhau bởi danh sách liên kết 2 chiêu, giúp việc chèn, sửa, xoá được thực hiện dễ dàng hơn. Các node được bôi đậm là các node có thuộc tính *mark* là TRUE

Mỗi node trong fibonacci heap sẽ chứa các thông tin:

1. `parent`: node cha của node hiện tại. Nếu x là node thuộc *root list* thì $x.parent = NULL$
2. `left, right`: node anh, em trái, phải của node hiện tại
3. `degree`: số lượng node con của node hiện tại
4. `child`: 1 node con của node hiện tại. Nếu x không có con thì $x.child = NULL$
5. `key`: giá trị khoá tại node hiện tại
6. `mark`: giá trị logic thể hiện việc node đó đã bị mất node con nào hay chưa. Khi khởi tạo thì mọi node đều có giá trị *mark* là FALSE. Do trong fibonacci heap, mỗi node chỉ bị mất tối đa 1 node con nên khi x là node bị mất 1 con thì $x.mark =$ TRUE

Một fibonacci heap sẽ được xác định thông qua 2 thuộc tính:

1. `min`: trỏ đến node có thuộc tính khoá nhỏ nhất trên heap. Nếu heap H là rỗng thì $H.min = NULL$
2. `n`: số lượng node hiện có của heap

2.2 Hàm thế năng

Để đánh giá thời gian thực hiện các phép toán trên fibonacci heap thì ta sẽ sử dụng hàm thế năng:

$$\phi(H) = t(H) + 2m(H)$$

Trong đó:

1. $t(H)$: số lượng node có trong root list
2. $m(H)$: số lượng node có thuộc tính *mark* là TRUE

2.3 Đánh giá số lượng con lớn nhất của 1 node

Gọi $D(n)$ là cận trên cho số lượng con của 1 node (max degree) trong 1 fibonacci heap có số node là n. Khi đó, ta có đánh giá:

$$D(n) \leq \lg n$$

Chứng minh:

1. *Tính chất của số Fibonacci*

   Số Fibonacci thứ k được định nghĩa như sau:
   $F_0 = 0$
   $F_1 = 1$
   $F_k = F_{k-1} + F_{k-2}$ với $k \geq 2$

   Bằng quy nạp, ta có thể chứng minh được $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$
   Gọi ϕ là nghiệm dương của phương trình $\phi^2 = \phi + 1$
   Sử dụng quy nạp một lần nữa, ta có thể chứng minh được $F_{k+2} \geq \phi^k$.

2. *Tính chất của các node con*

   x là một node bất kì trong heap và $x.degree = k$. Gọi $y_1, y_2, ..., y_k$ là các node con được nối với x theo đúng thứ tự đó từ sớm nhất đến muộn nhất. Ta có tính chất $y_1.degree \geq 0, y_i.degree \geq i - 2$ với $i \geq 2$.

   Hiển nhiên $y_1.degree \geq 0$
   Ta có nhận xét, khi y_i được nối với x thì các node $y_1, y_2, ..., y_{i-1}$ đã là node con của x, vậy nên $x.degree \geq i - 1$. Vì node y_i được nối với x nên $y_i.degree = x.degree \geq i - 1$. Vì mỗi node chỉ có thể mất tối đa 1 node con nên $y_i.degree \geq i - 2$ (đpcm).

3. Gọi $size(x)$ là tổng số node của cây con có gốc là x, với x là node bất kì. Đặt $x.degree = k$. Ta có tính chất $size(x) \geq F_{k+2} \geq \phi^k$

   Gọi s_k là node có $size$ nhỏ nhất và có k node con. Ta thấy $s_0 = 1, s_1 = 2$.
   Vì thao tác nối 2 node không làm giảm $size$ của node cha, nên giá trị s_k là không giảm đối với k, do đó, $s_k \geq size(x)$.
   Vì node x có các node con là $y_1, y_2, ..., y_k$ nên ta có:
   $s_k \geq y_1.degree + \sum_{i=2}^{k} s_{y_i}.degree + 1 \geq 2 + \sum_{i=2}^{k} s_{i-2}$
   Bằng quy nạp, ta có thể chứng minh được $s_k \geq F_{i+2}$
   Do đó, $size(x) \geq s_k \geq F_{k+2} \geq \phi^k$ (đpcm)

4. Cận trên của $D(n)$ là $O(\lg n)$

   Gọi x là node bất kì của heap có n node và $x.degree = k$. Ta có $n \geq size(x) \geq \phi^k$, do đó $k \leq \log_\phi n$.
   Vậy số node con tối đa của 1 node là $O(\lg n)$.

   Nhờ sự liên hệ giữa heap và tính chất của các số Fibonacci nên cấu trúc dữ liệu này được gọi là *Fibonacci heap*

3 Các phép toán trên heap

Các phép toán trên heap được thiết kế để có độ trễ càng lâu càng tốt. Như khi thêm 1 node mới vào heap, ta chỉ việc khởi tạo 1 node và chèn node đó vào root list. Khi phép toán lấy ra min node được thực hiện (extract min), cấu trúc heap mới được tổ chức lại, khi đó số lượng node trên root list tối đa là $D(n) + 1$

3.1 Tạo heap mới (`MAKE-FIB-HEAP()`)

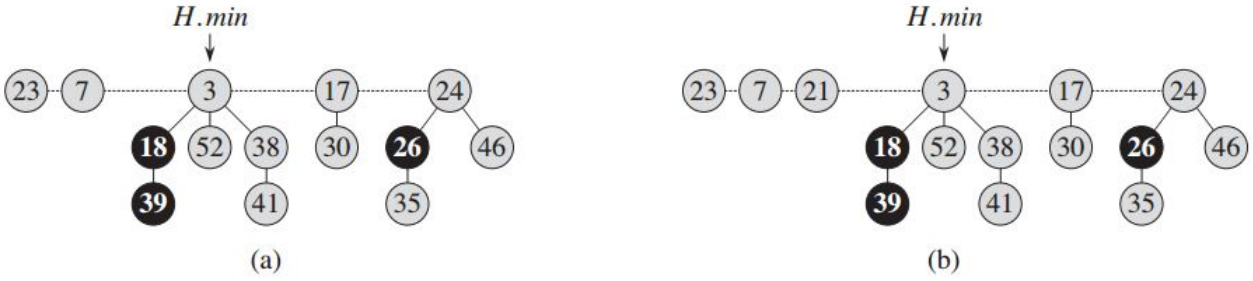
Hàm này sẽ trả về một heap H với các giá trị được khởi tạo $H.n = 0$ và $H.min = NULL$. Hàm thế năng $\phi(H) = t(H) + m(H) = 0$. Do đó, phép toán được thực hiện trong thời gian là $O(1)$.

3.2 Thêm node mới vào heap (`FIB-HEAP-INSERT(H, x)`)

Hàm này sẽ khởi tạo node mới x và chèn node mới này vào root list. Nếu node mới có giá trị khoá nhỏ hơn min node thì giá trị $H.min$ sẽ được cập nhật lại.

```
FIB-HEAP-INSERT(H, x)
 1  x.degree = 0
 2  x.p = NIL
 3  x.child = NIL
 4  x.mark = FALSE
 5  if H.min == NIL
 6       create a root list for H containing just x
 7       H.min = x
 8  else insert x into H's root list
 9       if x.key < H.min.key
10            H.min = x
11  H.n = H.n + 1
```

Hình 2: Mã giả thao tác FIB-HEAP-INSERT



Hình 3: Ví dụ về thao tác FIB-HEAP-INSERT

Hình trên là ví dụ minh hoạ khi chèn thêm node mới có giá trị khoá là 21 vào heap. Ở trường hợp này, node mới sẽ được chèn vào bên trái min node.

Thời gian khấu hao:
$\phi(H') - \phi(H) = (t(H') + m(H')) - (t(H) + m(H)) = (t(H) + 1 + m(H)) - (t(H) + m(H)) = 1$
Do đó, thời gian thực hiện thao tác này là $O(1) + 1 = O(1)$.

3.3 Hợp nhất 2 heap (`FIB-HEAP-UNION`(H_1, H_2))

Thao tác này tạo ra 1 heap mới với root list chính là danh sách kết hợp 2 root list của 2 heap và có số node trên heap bằng tổng số node của 2 heap. Giá trị min node sẽ được cập nhật cho heap mới.

```
FIB-HEAP-UNION(H1, H2)
1  H = MAKE-FIB-HEAP()
2  H.min = H1.min
3  concatenate the root list of H2 with the root list of H
4  if (H1.min == NIL) or (H2.min ≠ NIL and H2.min.key < H1.min.key)
5      H.min = H2.min
6  H.n = H1.n + H2.n
7  return H
```

Hình 4: Mã giả thao tác FIB-HEAP-UNION

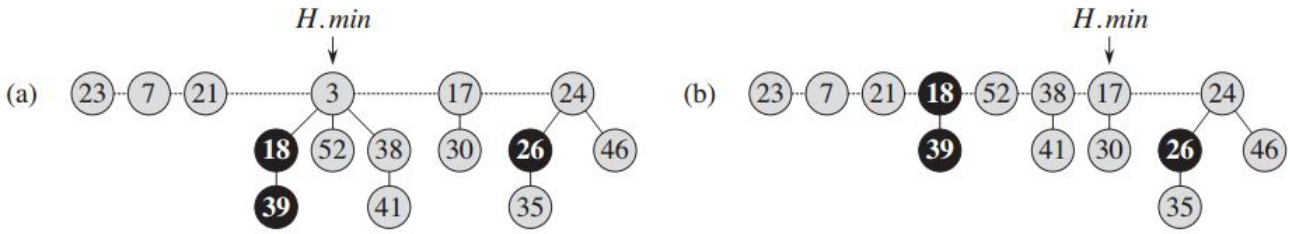
Thời gian khấu hao:
$\phi(H) - (\phi(H_1) + \phi(H_2)) = (t(H) + m(H)) - ((t(H_1) + m(H_1)) + (t(H_2) + m(H_2)) = 0$
Do đó, thời gian thực hiện thao tác là $O(1)$

3.4 Lấy ra min node (`EXTRACT-MIN(H)`)

Đây là thao tác phức tạp nhất đối với fibonacci heap. Đây cũng là thao tác giúp tổ chức lại cấu trúc cây sau hàng loạt các thao tác trì hoãn công việc này.
Trước khi lấy ra min node, chúng ta cần thêm tất cả các node con của min node vào root list, đặt giá trị `parent` là NULL. Xoá min node và đặt giá trị min node mới bằng anh em trái (hoặc phải) của min node cũ. Sau đó, thao tác này sẽ tiến hành gọi thủ tục `CONSOLIDATE(H)` để tổ chức lại cấu trúc cây.

```
FIB-HEAP-EXTRACT-MIN(H)
 1  z = H.min
 2  if z ≠ NIL
 3      for each child x of z
 4          add x to the root list of H
 5          x.p = NIL
 6      remove z from the root list of H
 7      if z == z.right
 8          H.min = NIL
 9      else H.min = z.right
10          CONSOLIDATE(H)
11      H.n = H.n − 1
12  return z
```

Hình 5: Mã giả thao tác EXTRACT-MIN



Hình 6: Hình trạng heap sau khi xoá min node

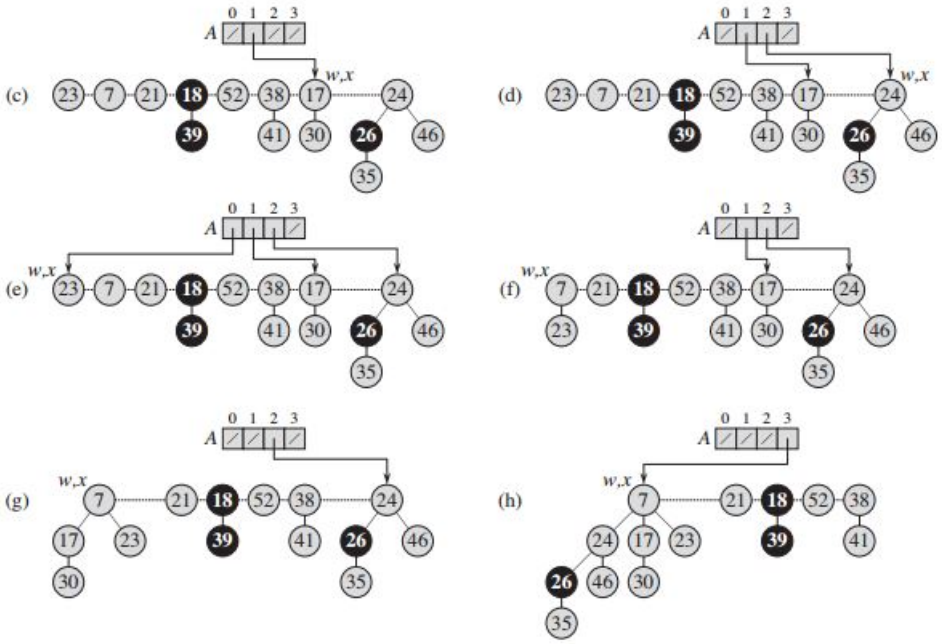
Hình trên là minh hoạ cho thao tác xoá node 3 (min node) ra khỏi heap ban đầu. Đầu tiên, chúng ta sẽ thêm các node có khoá 18, 52, 38 lên root list, xoá node có khoá là 3 ra khỏi heap và đặt lại con trỏ min node sang node bên cạnh (node 17)

3.4.1 `CONSOLIDATE(H)`

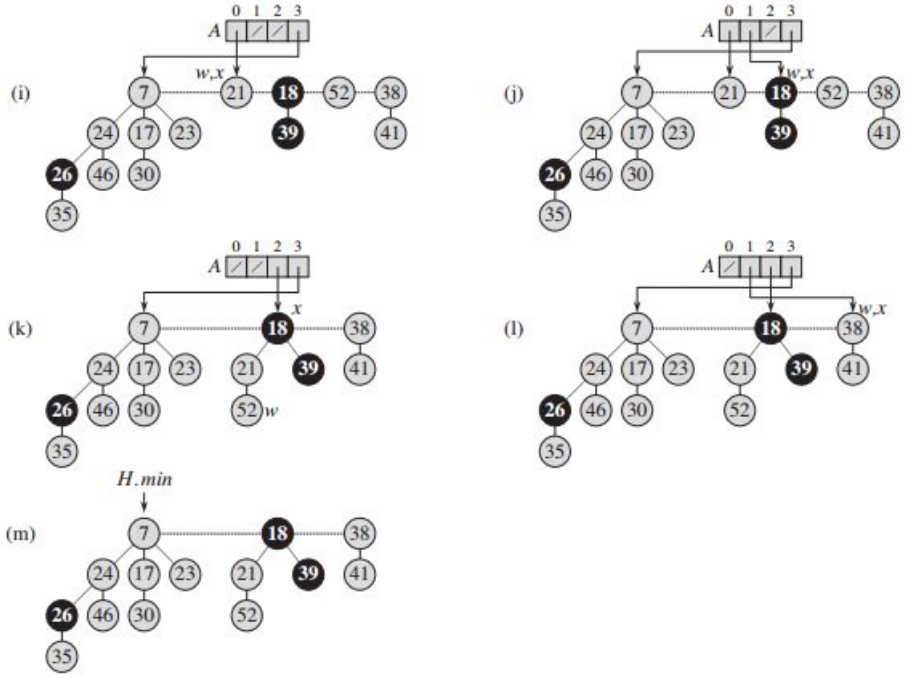
Thao tác này đảm bảo các node trong root list sẽ không có cùng số node con, tức là 2 node khác nhau sẽ có số lượng node con khác nhau. Do số node con tối đa có thể có của 1 node là $D(n) \leq \lg n$, nên số node tối đa trong root list là $\lg n + 1$.
Thao tác này trước nhất sẽ khởi tạo 1 mảng với số phần tử là số node con tối đa có thể có của 1 node. Bắt đầu duyệt từ min node, sau đó đi theo anh em phải (hoặc trái) của node vừa được duyệt. Nếu node được duyệt có số node con là duy nhất tính đến thời điểm duyệt, thì giá trị node này sẽ được lưu vào vị trí trương ứng với số node con trong mảng. Nếu node được duyệt có số node con trùng với bất kì node nào đã duyệt trước đó thì thao tác `FIB-HEAP-LINK` sẽ được gọi, thao tác này sẽ biến 1 node thành con của node kia. Kết thúc thao tác này, cấu trúc cây sẽ được tổ chức lại một cách hợp lí.

```
CONSOLIDATE(H)
 1  let A[0..D(H.n)] be a new array
 2  for i = 0 to D(H.n)
 3      A[i] = NIL
 4  for each node w in the root list of H
 5      x = w
 6      d = x.degree
 7      while A[d] ≠ NIL
 8          y = A[d]          // another node with the same degree as x
 9          if x.key > y.key
10              exchange x with y
11          FIB-HEAP-LINK(H, y, x)
12          A[d] = NIL
13          d = d + 1
14      A[d] = x
15  H.min = NIL
16  for i = 0 to D(H.n)
17      if A[i] ≠ NIL
18          if H.min == NIL
19              create a root list for H containing just A[i]
20              H.min = A[i]
21          else insert A[i] into H's root list
22              if A[i].key < H.min.key
23                  H.min = A[i]
```

Hình 7: Mã giả thủ tục `CONSOLIDATE(H)`



Hình 8: Ví dụ thủ tục `CONSOLIDATE(H)`



Hình 9: Cấu trúc heap được tổ chức lại

3.4.2 `FIB-HEAP-LINK(H, y, x)`

Thủ tục này sẽ biến node y thành node con của node x và đặt lại giá trị *mark* của y.
y.mark = FALSE

FIB-HEAP-LINK(H, y, x)
1 remove y from the root list of H
2 make y a child of x, incrementing $x.degree$
3 $y.mark$ = FALSE

Hình 10: Mã giả thủ tục FIB-HEAP-LINK

3.4.3 Thời gian khấu hao

Thời gian để đặt các node con của min node lên root list là $O(D(n))$
Thời gian duyệt qua các node trong min node là $t(H)$
Do đó, tổng thời gian xoá min node ra khỏi heap là $O(D(n)+t(H))$ với hệ số của $t(H) \leq 1$ do ta duyệt qua nhiều nhất $t(H)$ node.

Hàm thế năng trước thao tác `EXTRACT-MIN` là $t(H)+2m(H)$.
Hàm thế năng sau khi thực hiện thao tác `EXTRACT-MIN` là $(D(n)+1)+2m(H)$. Vì lúc đó, trên root list có ít nhất $D(n)+1$ node và trong quá trình thực hiện thao tác, không có node nào được *marked*.

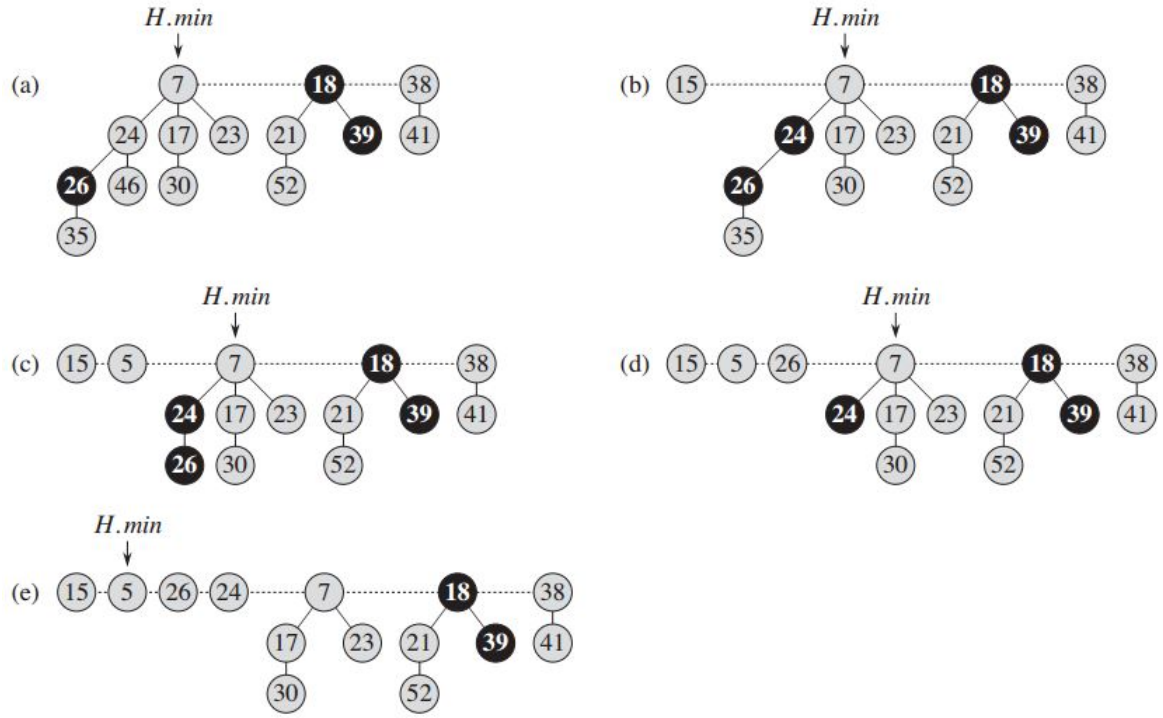
Tổng thời gian khấu hao:
$O(D(n)+t(H))+((D(n)+1)+2m(H))-(t(H)+2m(H)) = O(D(n)+t(H))-t(H) = O(D(n))$
Do $D(n) \leq \lg n$ nên thời gian thực hiện thao tác là $O(\lg n)$

3.5 Giảm giá trị khoá (`FIB-HEAP-DICREASE-KEY(H, x, k)`)

Thao tác này sẽ gán giá trị khoá mới k cho node x. Sau đó, nó sẽ gọi đến 2 thủ tục là `CUT` và `CASCADING-CUT` tổ chức lại cấu trúc heap để thoả mãn tính chất min-heap.

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > x.key
2      error "new key is greater than current key"
3  x.key = k
4  y = x.p
5  if y ≠ NIL and x.key < y.key
6      CUT(H, x, y)
7      CASCADING-CUT(H, y)
8  if x.key < H.min.key
9      H.min = x
```

Hình 11: Mã giả thao tác FIB-HEAP-DICREASE-KEY



Hình 12: VÍ dụ thao tác FIB-HEAP-DICREASE-KEY

Hình trên minh hoạ thao tác DECREASE-KEY. Ở đây, ta giảm các node $46 \to 15, 35 \to 5$

3.5.1 `CUT(H, x, y)`

Thủ tục này có xoá x khỏi danh sách node con của y, và đặt x lên root list.

```
CUT(H, x, y)
1  remove x from the child list of y, decrementing y.degree
2  add x to the root list of H
3  x.p = NIL
4  x.mark = FALSE
```

Hình 13: Mã giả thủ tục CUT

3.5.2 `CASCADING-CUT(H, y)`

Thủ tục này sẽ được gọi một cách đệ quy, nhằm thoả mãn ràng buộc của fibonacci heap, rằng mỗi node chỉ được mất tối đa 1 con. Nếu 1 node đã mất con tiếp tục bị mất thêm 1 node con khi thao tác DICREASE-KEY thực hiện thì nó bị "cắt" khỏi node cha và đặt lên root list, đồng thời đặt lại giá trị $mark = TRUE$.

```
CASCADING-CUT(H, y)
1  z = y.p
2  if z ≠ NIL
3       if y.mark == FALSE
4            y.mark = TRUE
5       else CUT(H, y, z)
6            CASCADING-CUT(H, z)
```

Hình 14: Mã giả thủ tục CASCADING-CUT

3.5.3 Thời gian khấu hao

Mỗi lần gọi thủ tục `CASCADING-CUT` tốn $O(1)$ Giả sử thao tác này gọi đến c lần thủ tục `CASCADING-CUT`, độ phức tạp $O(c)$.
Sau c lần gọi thủ tục `CASCADING-CUT` thì root list có thêm c node, đồng thời số lượng mark node giảm đi $c - 2$ node.

Tổng thời gian khấu hao:
$O(c) + ((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = O(c) + 4 - c = O(1)$

3.6 Xoá một node khỏi heap (`FIB-HEAP-DELETE(H, x)`)

Thao tác này tương đương với quy trình:

1. `FIB-HEAP-DICREASE-KEY(H, x, `$-\infty$`)`
2. `FIB-HEAP-EXTRACT-MIN(H)`

3.7 Tổng hợp và so sánh

Fibonacci heap giúp cải thiện đa số các thao tác thực hiện trên heap trong thời gian hằng số

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

Hình 15: Độ phức tạp các thao tác

4 Ứng dụng trong bài toán tìm đường đi ngắn nhất

4.1 Bài toán

Cho đồ thị vô hướng có trọng số. Các đỉnh được đánh số từ 1 đến n.
Bài toán đặt ra là tìm đường đi có tổng trọng số trên các cạnh là bé nhất từ đỉnh 1 đến đỉnh n

Input: Dòng đầu tiên gồm 2 số nguyên n, m ($2 \leq n \leq 10^5, 0 \leq m \leq 10^5$), với n là số đỉnh, m là số cạnh. m dòng tiếp theo gồm cạnh qua mỗi đỉnh a_i, b_i, w_i ($1 \leq a_i, b_i \leq n, 1 \leq w_i \leq 10^6$).
Output: In ra đường đi ngắn nhất, nếu không tồn tại thì in ra -1.

input Copy

```
5 6
1 2 2
2 5 5
2 3 4
1 4 1
4 3 3
3 5 1
```

output Copy

```
1 4 3 5
```

Hình 16: Minh hoạ bài toán

4.2 Thuật toán Dijkstra

Thuật toán Dijkstra áp dụng cho đồ thị vô hướng $G(V, E)$ có trọng số và một đỉnh nguồn s. Thuật toán sẽ tính toán đường đi ngắn nhất từ đỉnh s đến tất cả các đỉnh trong đồ thị.

Các bước thực hiện thuật toán:

- Dùng 1 hàm số $d(u)$ để lưu trữ độ dài đường đi ngắn nhất từ đỉnh nguồn s đến đỉnh u. Khởi tạo ban đầu $d(s) = 0$ còn lại $d(v) = \infty$
- Mỗi lần duyệt và tính được $d(v)$, ta tiến hành cập nhật lại độ dài đường đi đến tất cả các đỉnh.
  Kí hiệu $X^-(u)$ là tập các đỉnh có cạnh đi tới u. Với mọi v thuộc $X^-(u)$ đã xác định $d(v)$ thì:

$$d(u) = min\{d(u), d(v) + w(v, u)\}$$

- Các đỉnh cần được theo dõi trạng thái trong quá trình thực hiện thuật toán và lưu lại giá trị đỉnh liền trước nó trong đường đi ngắn nhất từ đỉnh nguồn s

Cấu trúc dữ liệu thường được sử dụng trong bài toán này là dùng một hàng đợi ưu tiên.

Các thao tác khi thực hiện thuật toán:

- Thêm đỉnh mới vào hàng đợi: (1)
- Lưu trữ đỉnh có độ dài đường đi ngắn nhất: (2)
- Cập nhật lại độ dài đường đi đến các đỉnh: (3)

Độ phức tạp thuật toán:
$O(|V| * (1) + |V| * (2) + |E| * (3))$

4.3 Ứng dụng fibonacci heap

4.3.1 Khởi tạo heap

```
// build heap, one vertice one node
for(int i=0; i<G->num_vertices; i++){
    parent[i]=-1;
    marked[i] = false;
    node_array[i] = new fib_node;

    if(i==src){
        insert(H, node_array[i], 0);
    } else{
        insert(H, node_array[i], INF);
    }

    node_array[i]->index = i;
}
```

Hình 17: Khởi tạo heap

4.3.2 Lưu trữ đỉnh có độ dài đường đi ngắn nhất

```
min_node = fib_heap_extract_min(H); //extract min distance
if(min_node!=NULL){
    min_node_index = min_node->index; // get node
```

Hình 18: Lưu trữ đỉnh có độ dài đường đi ngắn nhất

4.3.3 Cập nhật heap

```
if(node_array[adj_node_index]->key>node_array[min_node_index]->key+(*it)->weight){
    parent[adj_node_index] = min_node_index; // save parent
    fib_heap_decrease_key(H, node_array[adj_node_index],
    node_array[min_node_index]->key+(*it)->weight); // update distance
}
```

Hình 19: Cập nhật các giá trị trên heap

4.3.4 Độ phức tạp

Bài toán chúng ta sử dụng những thao tác:

- `FIB-HEAP-INSERT`: $O(1)$
- `EXTRACT-MIN`: $O(\lg n)$
- `DECREASE-KEY`: $O(1)$

Do đó, độ phức tạp cả bài là $O(|V|.\lg|V| + |E|)$

Hình 20: Kết quả

5 Kết luận

- Fibonacci heap là một cấu trúc dữ liệu nâng cao giúp cải thiện đa số các phép toán trên heap thực hiện trong thời gian hằng số
- Fibonacci heap có cài đặt phức tạp
- Trong 1 số bài toán với dữ liệu nhỏ, sự cải thiện của fibonacci heap là không đáng kể

6 Tài liệu tham khảo

1. *Introduction to Algorithms, Third Edition* - Thomas H.Cormen, Charles E. Leiserson, Ronald R. Rivest, Clifford Stein

2. `https://www.cs.princeton.edu/~wayne/teaching/fibonacci-heap.pdf`