CS F364: Design & Analysis of Algorithms

Lecture-kt16: Fibonacci Heap (contd..) + Shortest path in Graph

Dr. Kamlesh Tiwari,
Assistant Professor,
Department of Computer Science and Information Systems,
BITS Pilani, Rajasthan-333031 INDIA

Mar 02, 2017 (Campus @ BITS-Pilani Jan-May 2017)

Recap: Fibonacci Heaps

It support **mergeable heaps** operations in constant amortized time

Procedure	MAKE-HEAP	INSERT	MINIMUM	EXTRACT-MIN	UNION	DECREASE-KEY	DELETE
Binary Heap	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
Fibonacci Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$

Recap: Fibonacci Heaps

It support **mergeable heaps** operations in constant amortized time

Procedure	MAKE-HEAP	INSERT	MINIMUM	EXTRACT-MIN	UNION	DECREASE-KEY	DELETE
Binary Heap	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
Fibonacci Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$

- Maintains circular, doubly linked lists

Recap: Fibonacci Heaps

It support **mergeable heaps** operations in constant amortized time

Procedure	MAKE-HEAP	INSERT	MINIMUM	EXTRACT-MIN	UNION	DECREASE-KEY	DELETE
Binary Heap	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
Fibonacci Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$

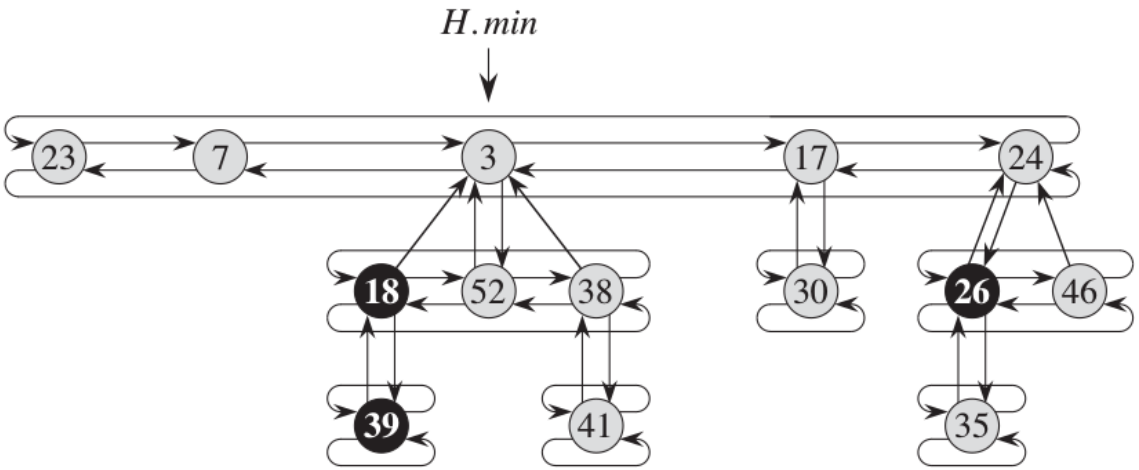
- Maintains circular, doubly linked lists
- $H.min$ points to minimum item of root-list therefore, **find minimum is** $O(1)$ operation

Recap: Fibonacci Heaps

It support **mergeable heaps** operations in constant amortized time

Procedure	MAKE-HEAP	INSERT	MINIMUM	EXTRACT-MIN	UNION	DECREASE-KEY	DELETE
Binary Heap	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
Fibonacci Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$

- Maintains circular, doubly linked lists
- *H.min* points to minimum item of root-list therefore, **find minimum is** $O(1)$ operation

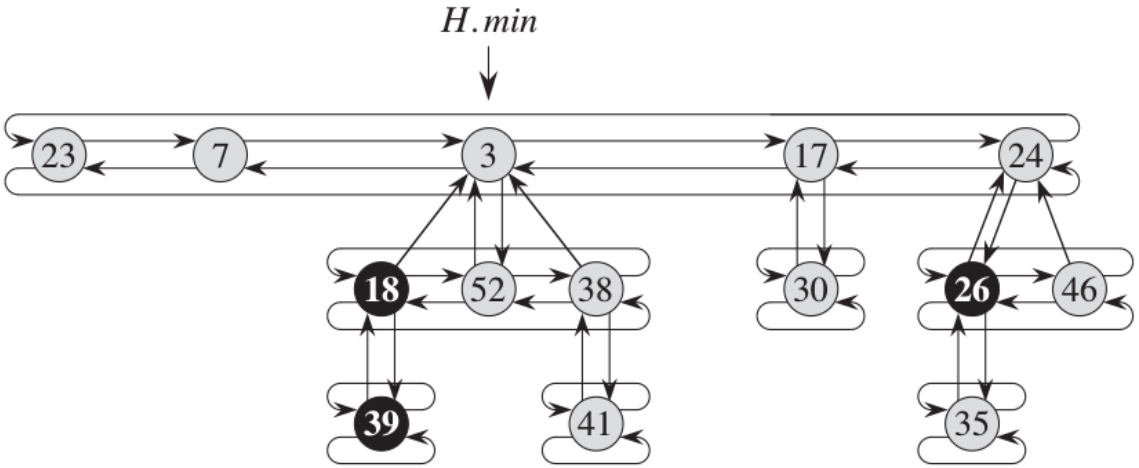


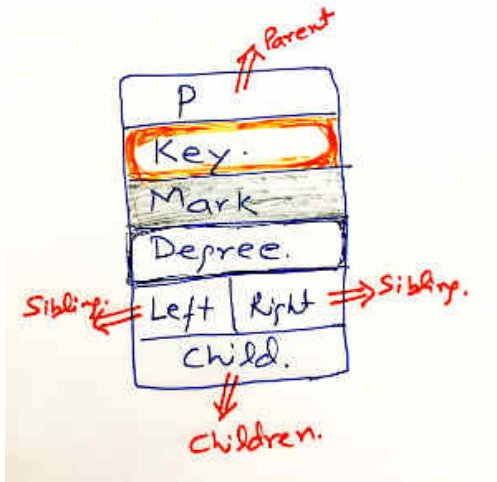
Recap: Fibonacci Heaps

It support **mergeable heaps** operations in constant amortized time

Procedure	MAKE-HEAP	INSERT	MINIMUM	EXTRACT-MIN	UNION	DECREASE-KEY	DELETE
Binary Heap	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
Fibonacci Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$

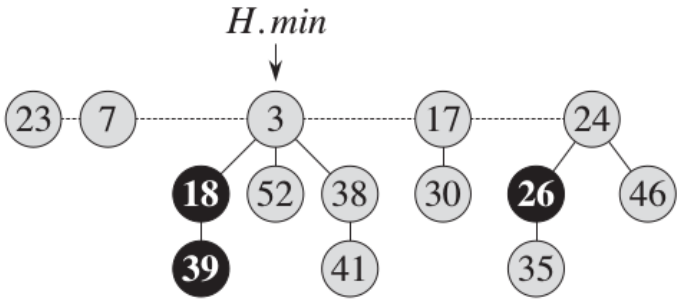
- Maintains circular, doubly linked lists
- *H.min* points to minimum item of root-list therefore, **find minimum is** $O(1)$ operation

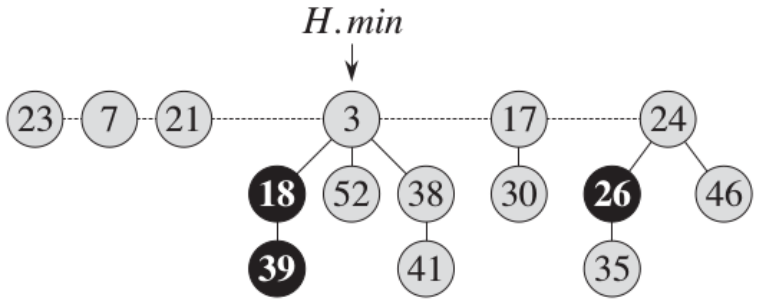




Recap: Fibonacci Heap Insertion

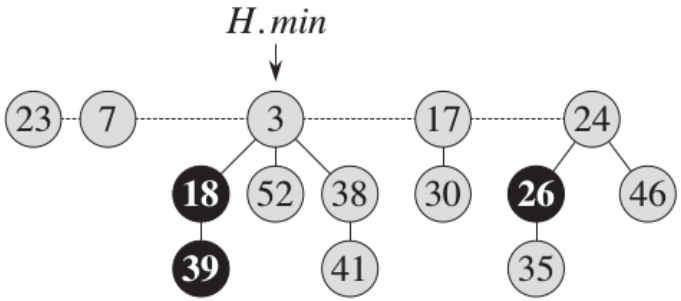
Inserting item with key 21

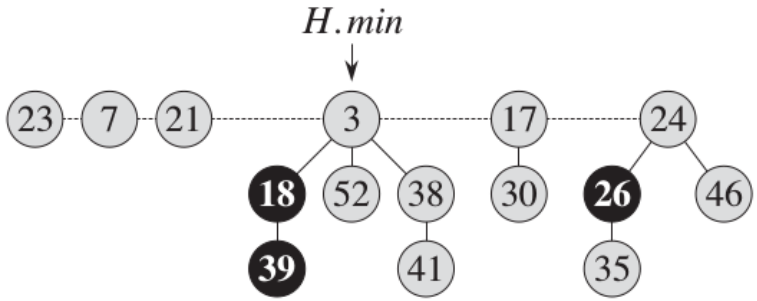




Recap: Fibonacci Heap Insertion

Inserting item with key 21

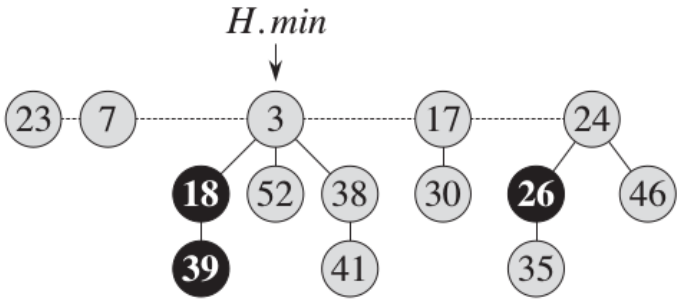


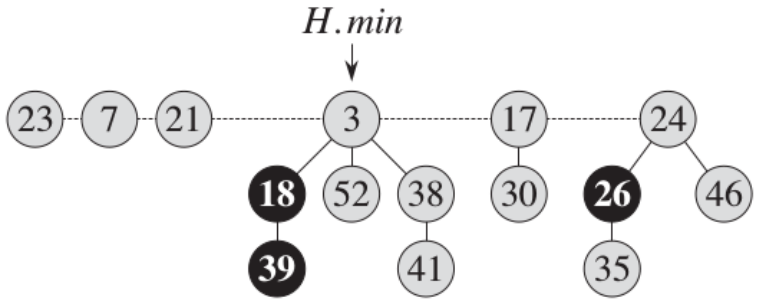


- Add the item in root-list
  Its pointer is known (H.min)

Recap: Fibonacci Heap Insertion

Inserting item with key 21

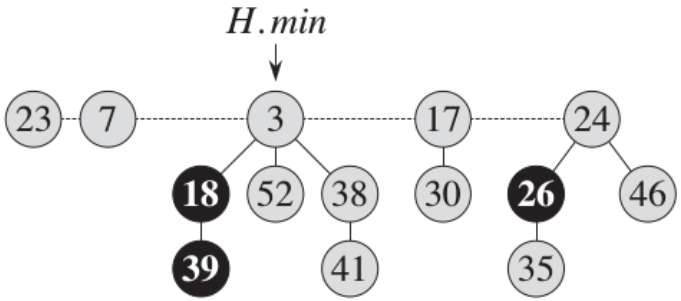


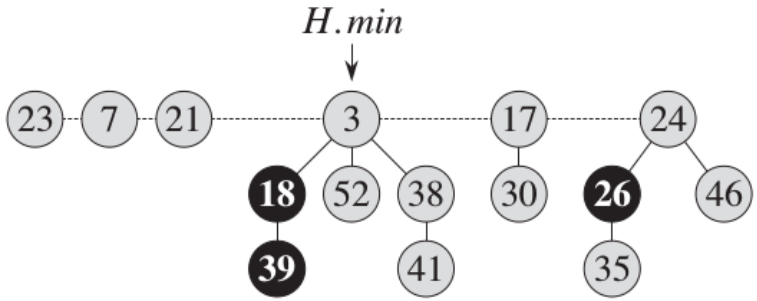


- Add the item in root-list
  Its pointer is known (H.min)
- Update H.min if needed

Recap: Fibonacci Heap Insertion

Inserting item with key 21

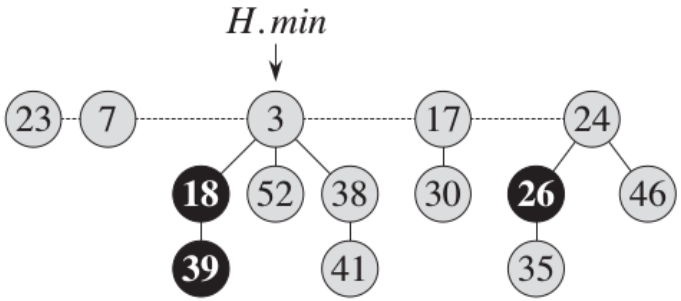


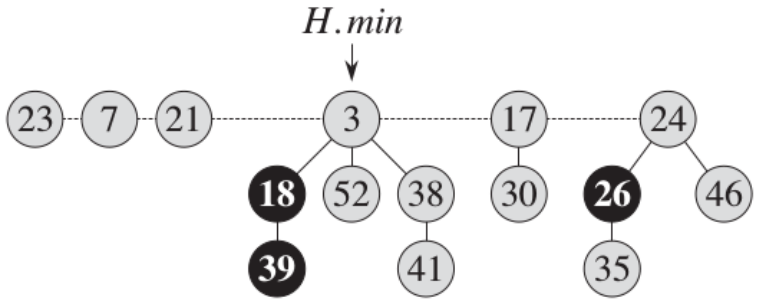


- Add the item in root-list
  Its pointer is known (H.min)
- Update H.min if needed
- Takes $O(1)$ time

Recap: Fibonacci Heap Insertion

Inserting item with key 21





- Add the item in root-list
  Its pointer is known (H.min)
- Update H.min if needed
- Takes $O(1)$ time

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
```

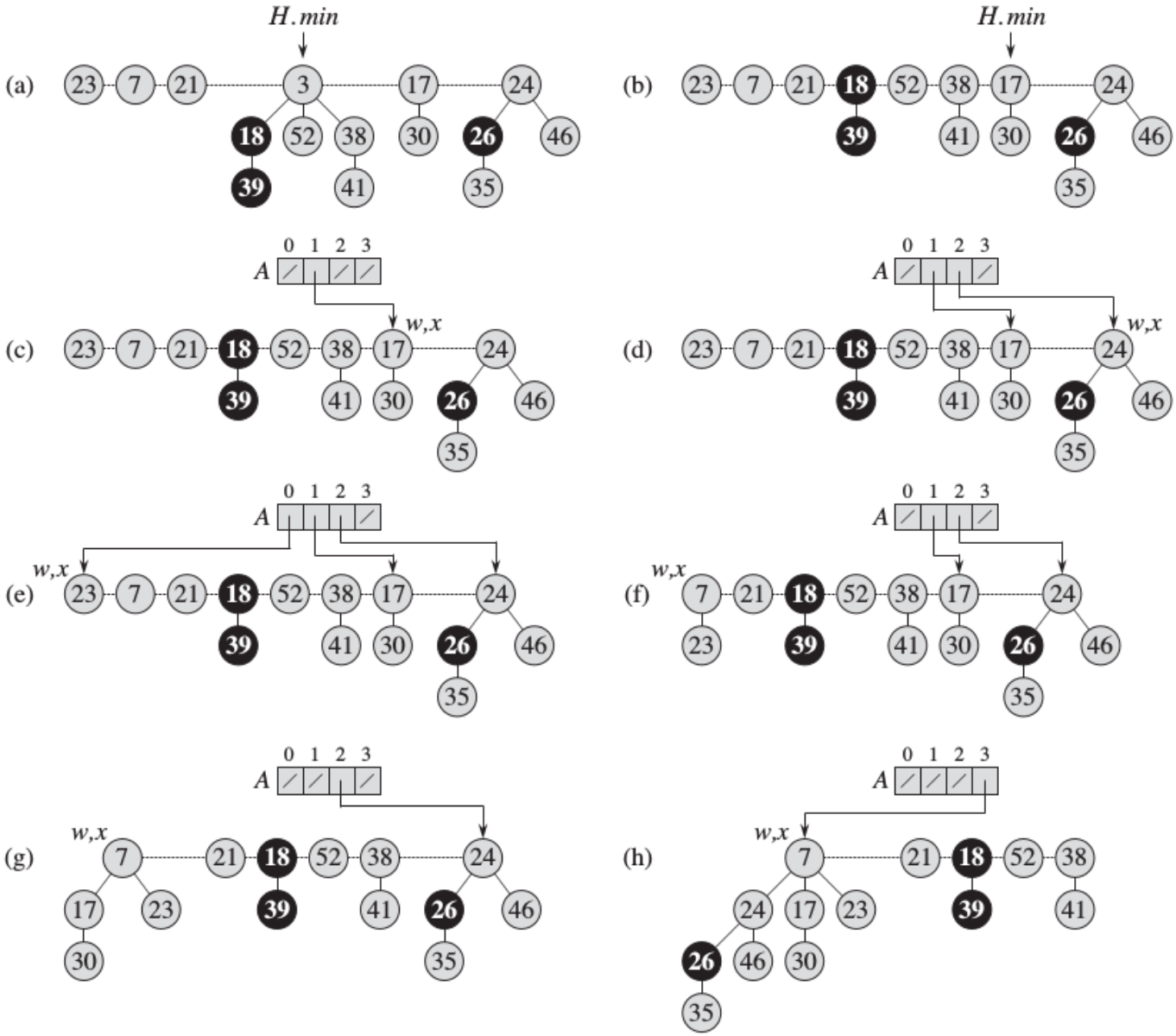
Fibonacci Heap Union

- Concatenate two root-lists in $O(1)$ time
- Update the pointer to minimum
- Overall time is $O(1)$

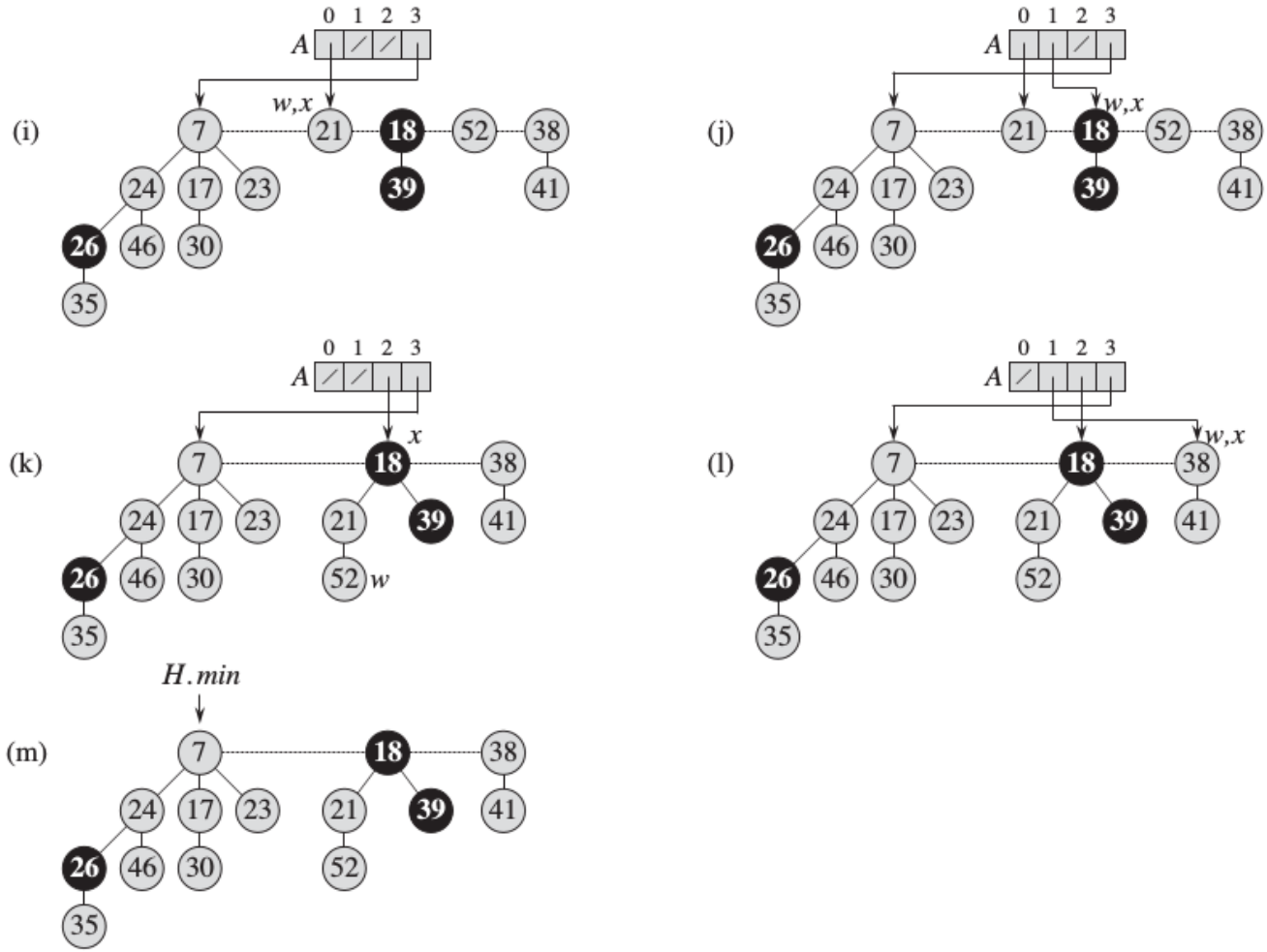
FIB-HEAP-UNION(H_1, H_2)

1 H = MAKE-FIB-HEAP()
2 $H.min = H_1.min$
3 concatenate the root list of H_2 with the root list of H
4 **if** ($H_1.min$ == NIL) or ($H_2.min \neq$ NIL and $H_2.min.key < H_1.min.key$)
5 $H.min = H_2.min$
6 $H.n = H_1.n + H_2.n$
7 **return** H

Extracting the minimum node



Extracting the minimum node



Extracting the minimum node

```
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
```

Extracting the minimum node

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
```

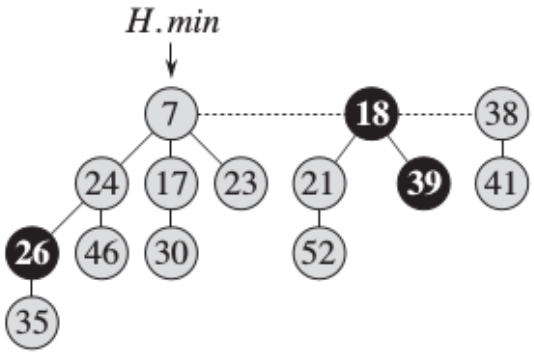
CONSOLIDATE(H)

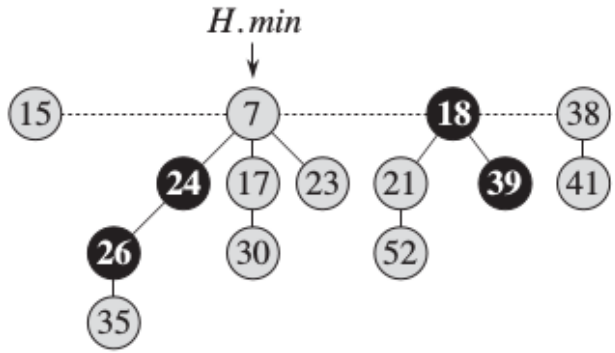
```
1   let A[0..D(H.n)] be a new array
2   for i = 0 to D(H.n)
3       A[i] = NIL
4   for each node w in the root list of H
5       x = w
6       d = x.degree
7       while A[d] ≠ NIL
8           y = A[d]        // another node with the same degree as x
9           if x.key > y.key
10              exchange 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]
```

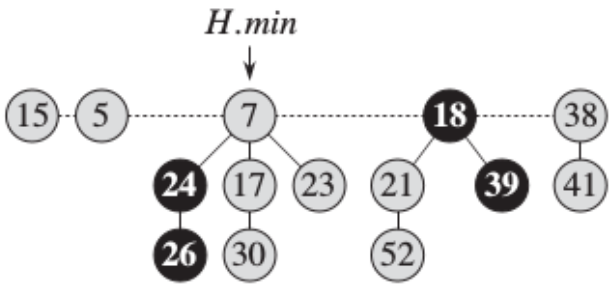
FIB-HEAP-LINK(H, y, x)

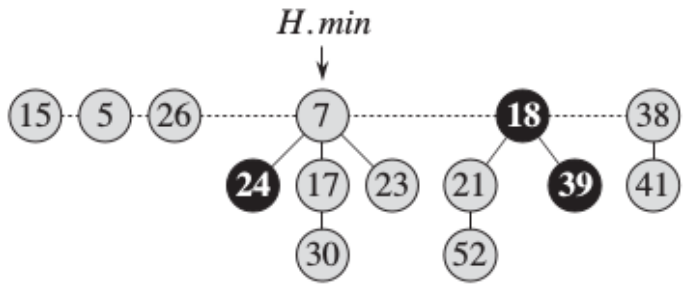
```
1   remove y from the root list of H
2   make y a child of x, incrementing x.degree
3   y.mark = FALSE
```

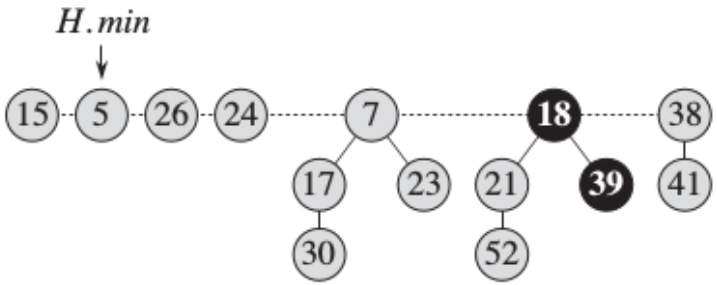
Decreasing a key











Decreasing key of 46-to-15 and 35-to-5

Decreasing a key

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
```

Decreasing a key

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
```

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
```

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)
```

Deletion

Algorithm 1: Delete(H,x)

1 Fib-Heap-Decrease-Key(H,x,$-\infty$)
2 Fib-Heap-Extract-Min(H)

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
```

Deletion

Algorithm 2: Delete(H,x)

1 Fib-Heap-Decrease-Key(H,x,$-\infty$)
2 Fib-Heap-Extract-Min(H)

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
```

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
```

Graph

- Defined as $G = (G.V, G.E)$

Graph

- Defined as $G = (G.V, G.E)$
- where $G.V$: Set of vertices, and $G.E$: set of edges

Graph

- Defined as $G = (G.V, G.E)$
- where $G.V$: Set of vertices, and $G.E$: set of edges
- **Single-source shortest-paths problem:** given a graph $G = (G.V, G.E)$, we want to find a shortest path from a given source vertex $s \in G.V$ to each vertex $v \in G.V$.

Graph

- Defined as $G = (G.V, G.E)$
- where $G.V$: Set of vertices, and $G.E$: set of edges
- **Single-source shortest-paths problem:** given a graph $G = (G.V, G.E)$, we want to find a shortest path from a given source vertex $s \in G.V$ to each vertex $v \in G.V$.
- Shortest path between two vertices contains other shortest paths

Graph

- Defined as $G = (G.V, G.E)$
- where $G.V$: Set of vertices, and $G.E$: set of edges
- **Single-source shortest-paths problem:** given a graph $G = (G.V, G.E)$, we want to find a shortest path from a given source vertex $s \in G.V$ to each vertex $v \in G.V$.
- Shortest path between two vertices contains other shortest paths

If negative weight are allowed,
negative-weight cycle would be an issue

Graph

- Defined as $G = (G.V, G.E)$
- where $G.V$: Set of vertices, and $G.E$: set of edges
- **Single-source shortest-paths problem:** given a graph $G = (G.V, G.E)$, we want to find a shortest path from a given source vertex $s \in G.V$ to each vertex $v \in G.V$.
- Shortest path between two vertices contains other shortest paths

If negative weight are allowed, negative-weight cycle would be an issue

Algorithm 8: Bellman-Foard(G,w,s)

```
1 Initialize-Single-Source(G,s)
2 for i = 1 to |G.V| − 1 do
3     for each edge (u, v) ∈ G.E do
4         Relax(u,v,w)
5 for each edge (u, v) ∈ G.E do
6     if v.d > u.d + w(u, v) then
7         return FALSE
8 return TRUE
```

Graph

Algorithm 9: Initialize-Single-Source(G,s)

1 **for** *each vertex* $v \in V$ **do**
2 $\quad v.d = \infty$ and $v.\pi = Nil$
3 $s.d = 0$

Graph

Algorithm 11: Initialize-Single-Source(G,s)

```
1 for each vertex v ∈ V do
2     v.d = ∞ and v.π = Nil
3 s.d = 0
```

Algorithm 12: Relax(u,v,w)

```
1 if v.d > u.d + w(u, v) then
2     v.d = u.d + w(u, v)
3     v.π = u
```

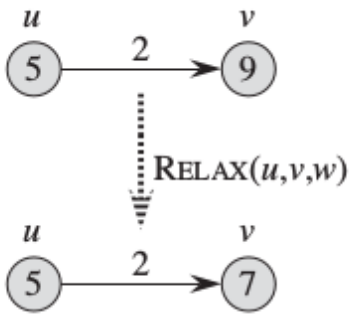
Graph

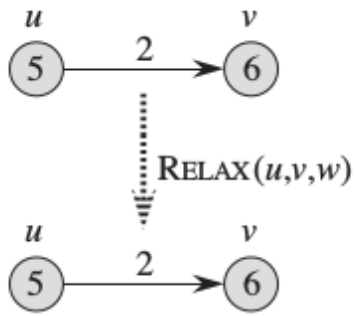
Algorithm 13: Initialize-Single-Source(G,s)

```
1 for each vertex v ∈ V do
2     v.d = ∞ and v.π = Nil
3 s.d = 0
```

Algorithm 14: Relax(u,v,w)

```
1 if v.d > u.d + w(u, v) then
2     v.d = u.d + w(u, v)
3     v.π = u
```





Thank You!

Thank you very much for your attention! (Reference[1])

Queries ?

[1] [1] Book - *Introduction to Algorithm*, By THOMAS H. CORMEN, CHARLES E. LEISERSON, RONALD L. RIVEST, CLIFFORD STEIN