02/2021

Date :

Module 3

Heap data structure (priority Queue)

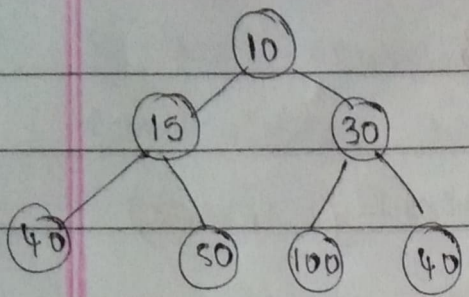
* a tree based data structure in which all the nodes of the tree are in a Special order.
* It is a almost complete binary tree
* Two types of heap data structure are:
  1) max-heap
  2) Min-heap

Max-heap :

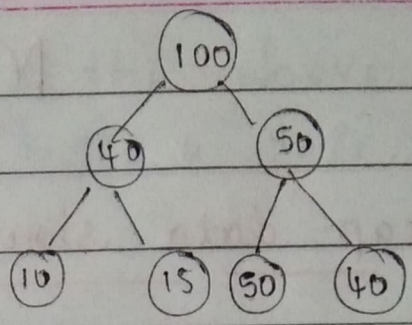
In a max-heap, the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in the binary Tree.

Min-heap :

In a min-heap, the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in the binary tree.

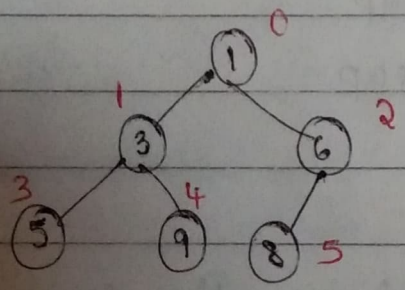


Min-Heap



Max-Heap

The traversal method used to achieve Array Representation is Level Order



0	1	2	3	4	5
1	3	6	5	9	8

Applications of heaps

1) Heap sort
2) Priority Queue
3) Graph Algorithms

MERGABLE HEAP TREE

A mergable heap supports the usual heap operations.

- Make-Heap() → Returns an empty heap
- Insert(H, x, k) → insert an item x with key k into the heap H
- Find-min(H) → Return item with min key
- Extract-min(H) → Extract & return the minimum element
- Merge(H1, H2) → combine the element of H1 & H2 into a single heap.

Examples : 1) Binomial Heap
2) Fibonacci Heap

BINOMIAL HEAP

* Binomial heap is an extension of binary heap.
* It provides faster union or merge operation together with other operations provided by binary heap.
* A binomial heap is a collection of binomial tree.
* No need to rebuild everything when union is performed.

Page No :
Date

Binomial tree

→ A binomial tree of order 0 has 1 node.

→ A binomial tree K can be constructed by taking two binomial trees of order K-1 and making one as leftmost child of other.

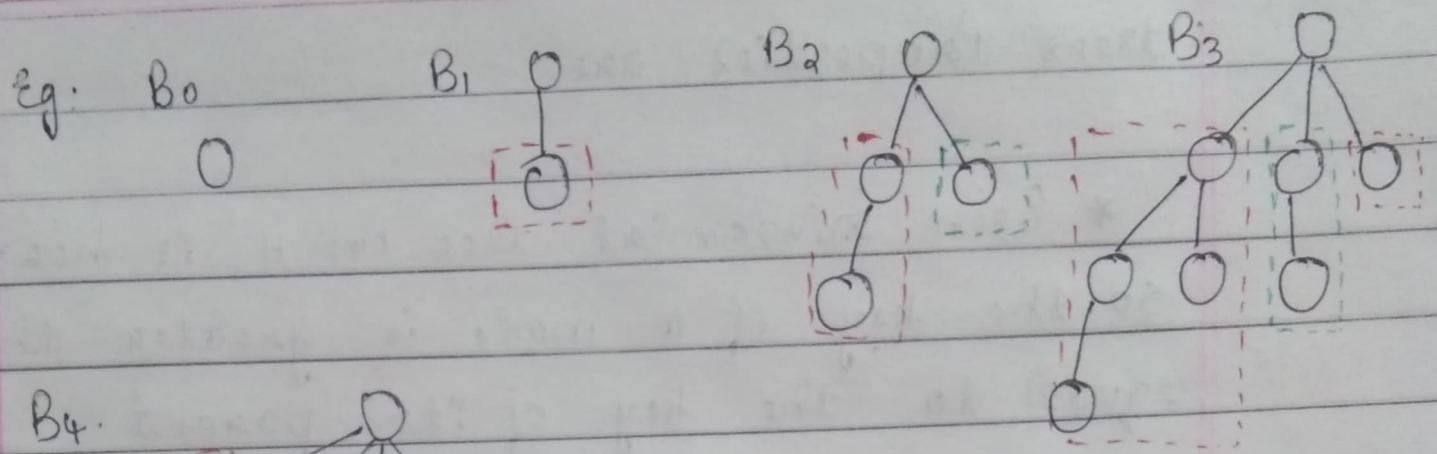
→ A binomial tree is a rooted tree with 2^n nodes defined recursively.

Properties of binomial tree

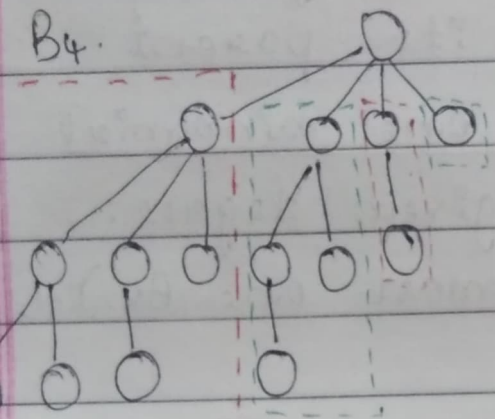
The binomial tree B_K has the following properties:

- It has 2^K nodes
- It has height K
- ~~There are exactly~~ degree(root) = K
  degree(other nodes) < K
- Children of root, from left to right, are $B_{K-1}, B_{K-2} \cdots B_1, B_0$
- Exactly c(K,i) nodes at depth I

Eg: B_0 B_1 B_2 B_3



B_4.



Here the binomial heap H consists of the binomial tree B_0, B_1, B_2, B_3; which have 1, 2, 4, 8 nodes respectively.

The root of binomial trees are linked by a linked list in order of increasing degree.

Consequences of the definition :

⇒ The root of a heap ordered tree contains the smallest key in the tree.

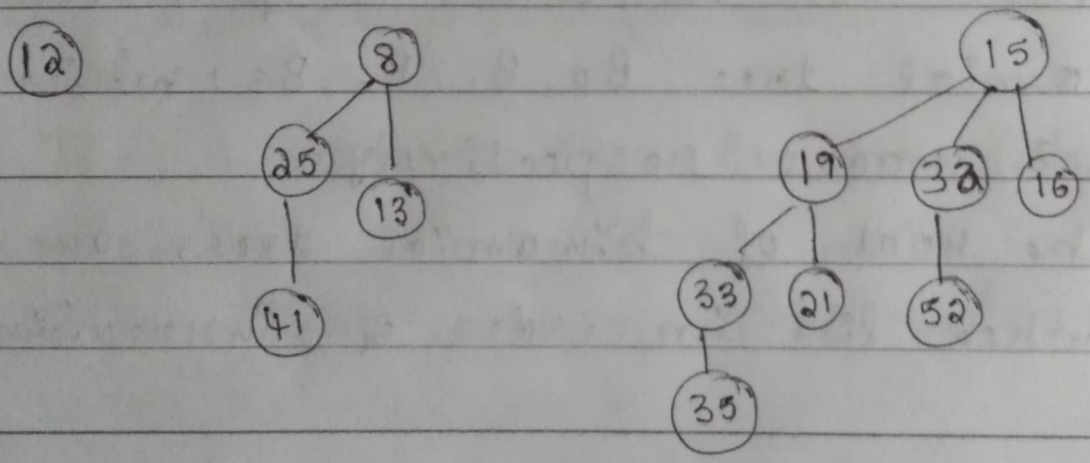
Binomial heap :

A binomial heap H is a set of binomial

trees. Properties are

* Each binomial tree in H is heap ordered so the key of a node is greater than or equal to the key of its parent.
* There is at most one binomial tree in H, whose root has a given degree.
(ie, for each k, at most one B_k)

Eg: A binomial heap with 13 elements



The first property tells us that the root of a heap-ordered tree contains the smallest key in the tree.

A binomial heap with n nodes has the number of binomial trees equal to the number of set bits in the binary

representation of n.

Eg: let n be 13.

Binary Representation: 1101 ⇒ B_3 B_2 B_0

We can conclude that there are $O(\log n)$ binomial trees in a binomial heap with 'n' nodes.

- Binary heaps can be implemented by using doubly linked lists to store the root nodes.
- Each node stores information about the parent pointer, left & right sibling pointer, left most child pointer, the number of children it has & its key.

Representation of Binary Heap

→ left child right Sibling Representation
→ Each node stores its degree.
→ Collection of binomial trees stored in ascending order of size.

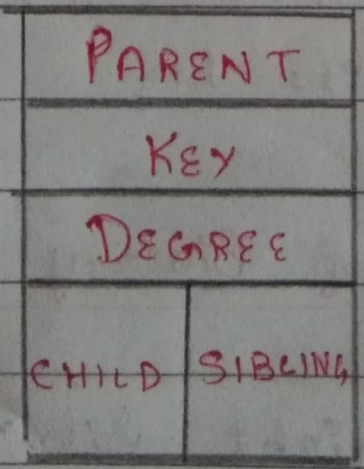
→ The root list in the heap is a linked list of roots of binomial heap.

→ Degree of the nodes of the roots increases on traversing the root list.

Fields in Each Node :

Each node in a binomial heap has 5 fields

1) Pointer to parent
2) Key
3) Degree
4) Pointer to child (Left most child)
5) Pointer to Sibling which is immediately to its right



Operations on Binomial Heap

1) Make-Heap()
2) Find-Min()
3) Union
4) Decrease-Key

1) Make-Heap()

Running time = O(1)

To make an empty binomial heap, MAKE-BINOMIAL-HEAP procedure simply allocates & returns an object H, where head(H) = NIL.

2) Finding Minimum key

The procedure BINOMIAL-HEAP-MINIMUM returns a pointer to the node with the minimum key in an n-node binomial heap H.

BINOMIAL-HEAP-MINIMUM(H)

```
y ← NIL
x ← head(H)
min ← ∞
while x ≠ NIL
    do if key[x] < min
        then min ← key[x]
             y ← x
    x ← siblings[x]
return y
```

→ Since a binomial heap is min-heap-ordered the minimum key must reside in a root node.

→ BINOMIAL-HEAP-MINIMUM procedure checks all roots, which number at most $\lfloor \lg n \rfloor + 1$, saving the current minimum in min & a pointer to current minimum in y.

→ Running time → $O(\log n)$

3) Union

* The operation of uniting two binomial heaps is used as a subroutine by most of the remaining operations.

* BINOMIAL-HEAP-UNION procedure repeatedly links binomial trees whose roots have same degree.

```
BINOMIAL-LINK(y, z)
p[y] ← z
sibling[y] ← child[z]
child[z] ← y
degree[z] ← degree[z] + 1
```

It makes node y the new head of the linked list of node z's children in O(1) time. It works because the left-child, right-sibling representation of each binomial tree matches the ordering property of the ~~root~~ tree.

BINOMIAL-HEAP-UNION (H_1, H_2)

```
1.  H ← MAKE-BINOMIAL-HEAP()
2.  head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
3.  free the objects H1 & H2 but not the lists they
    point to
4.  if head[H] = NIL
5.      then return H
6.  prev-x ← NIL
7.  x ← head[H]
8.  next-x ← sibling[x]
9.  while next-x ≠ NULL
10.     do if (degree[x] ≠ degree[next-x] or
11.        (sibling[next-x] ≠ NIL & degree[sibling[next-x]]
           = degree[x])
12.         then prev-x ← x                    case 1 and 2
13.              x ← next-x                    case 1 & 2
14.         else if key[x] ≤ key[next-x]
15.              then sibling[x] ← sibling[next x]   case 3
16.                   BINOMIAL-LINK(next-x, x)       case 3
17.              else if prev-x = NIL                 case 4
18.                   then head[H] ← next-x           case 4
19.                   else sibling[prev-x] ← next-x   case 4
20.                   BINOMIAL-LINK(x, next-x)        case 4
21.              x ← next-x                           case 4
22.     next x ← sibling(x)
```

22. return H

Binomial Heap Union Algorithm

Given a binomial heap H_1 & H_2

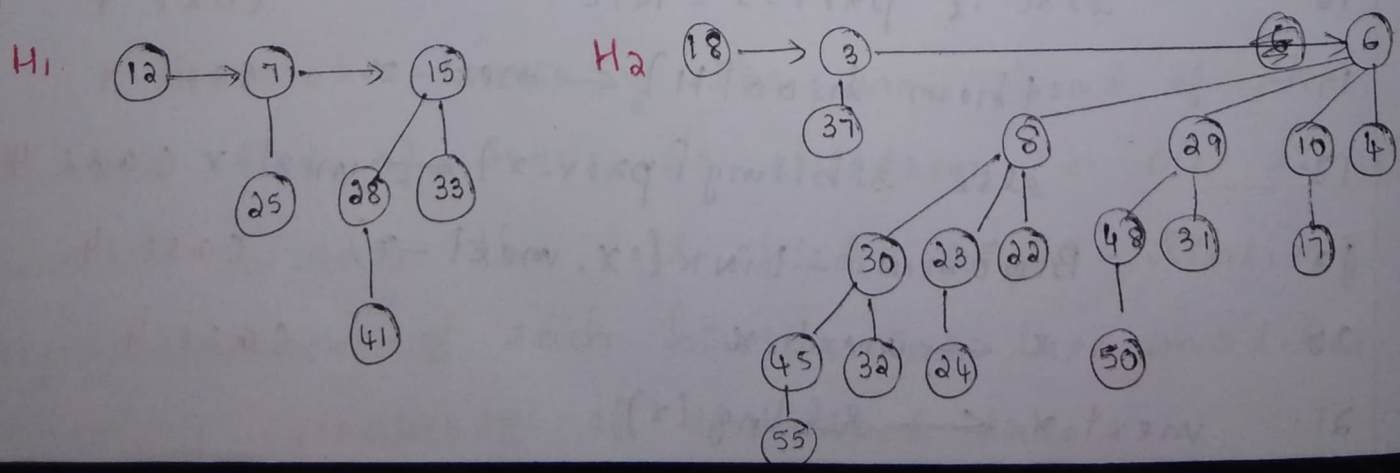
Step 1 : Merge H_1 & H_2 i.e, link the roots H_1 & H_2 in non-decreasing order

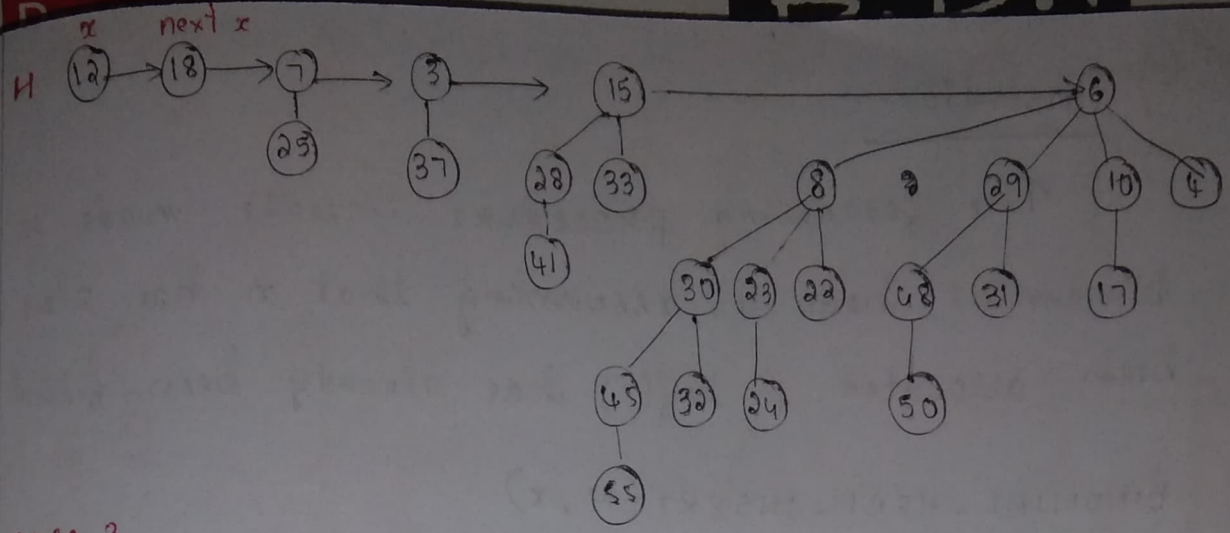
case 1 : If degree[x] ≠ degree[next x], then move pointers ahead.

case 2 : If degree[x] = degree[next x] = degree[sibling[next x]] move pointers ahead.

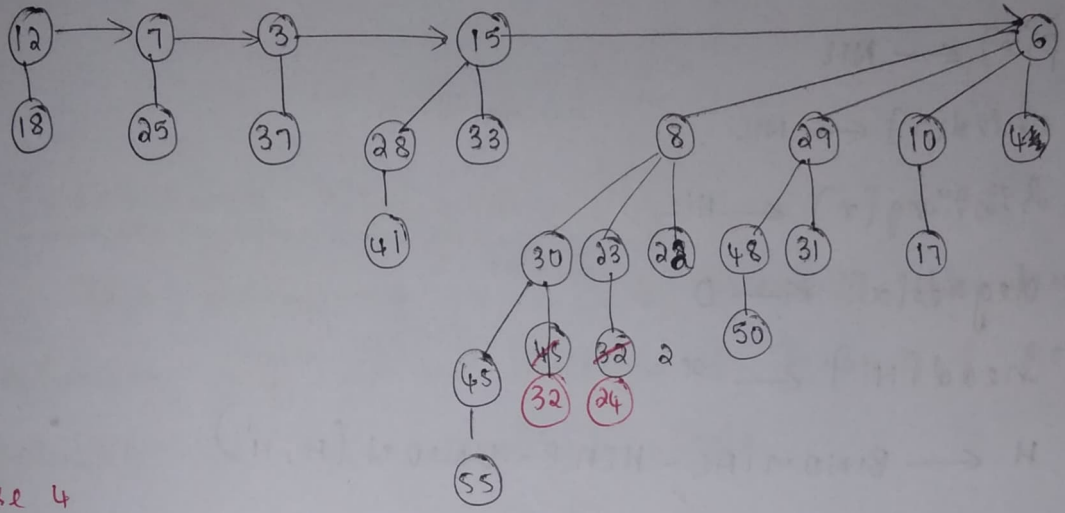
case 3 : If degree[x] = degree[next x] ≠ degree[sibling[next x]] and key[x] < key[next x] then remove next[x] from root & attach to x.

case 4 : If degree[x] = degree[next x] ≠ degree[sibling[next x]] & key[x] > key[next x] then remove x from root & attach to (next x).

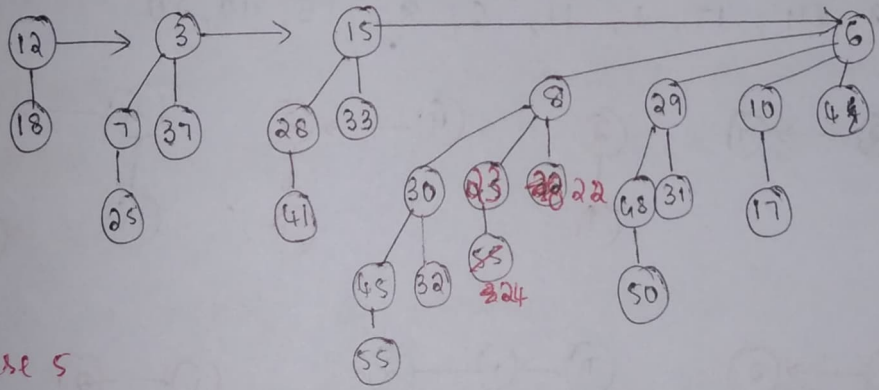




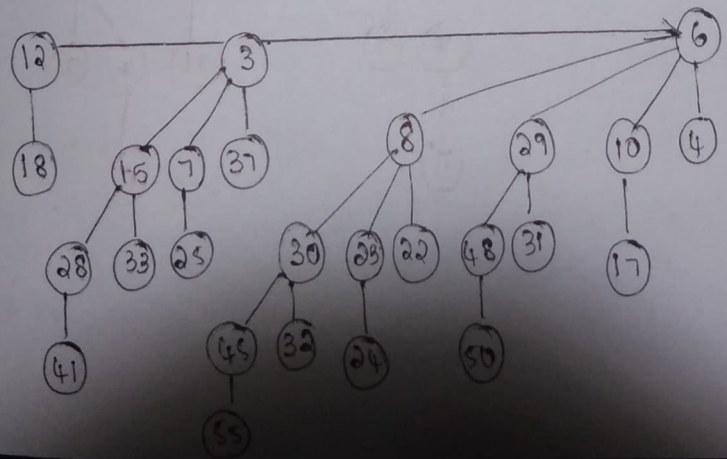
case 3



case 4



case 5



2021

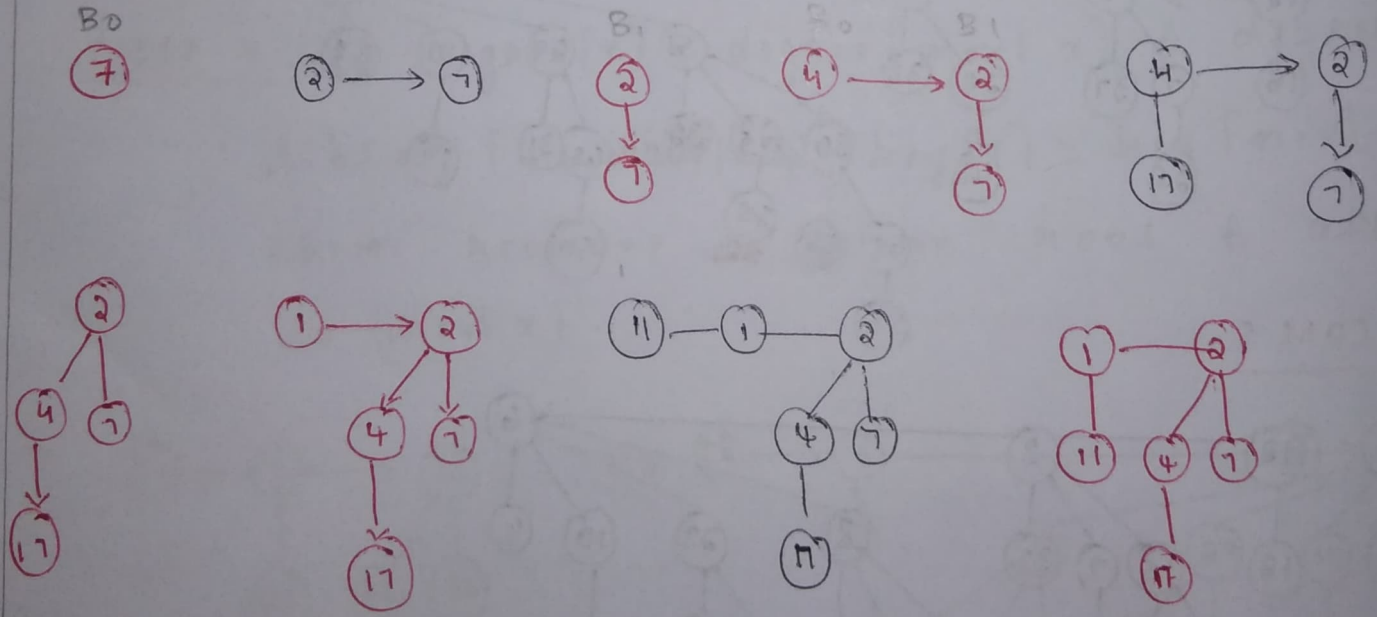
4) Insertion

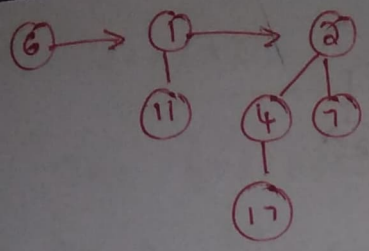
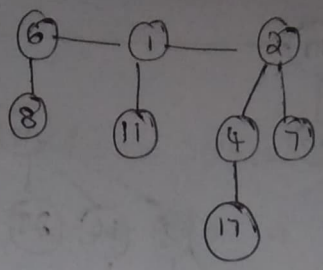
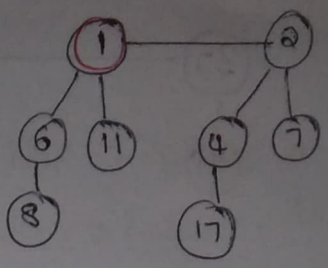
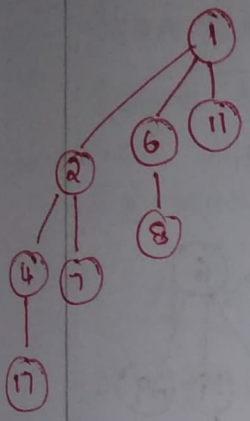
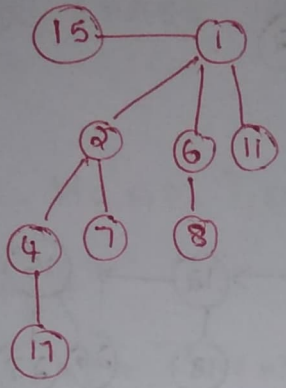
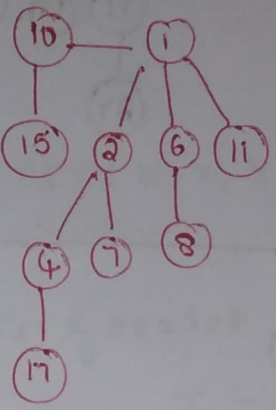
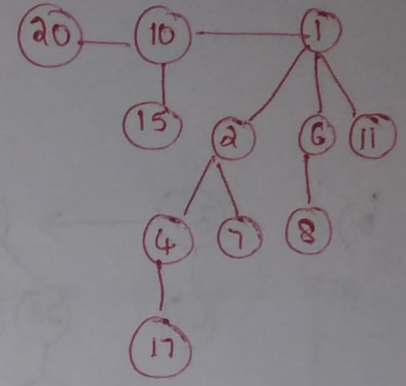
The following procedure inserts node x into binomial heap H, assuming that x has already been allocated & key[x] has already been filled in.

```
BINOMIAL-HEAP-INSERT(H, x)
1. H' ← MAKE-BINOMIAL-HEAP()
2. p[x] ← NIL
3. child[p] ← NIL
4. sibling[x] ← NIL
5. degree[x] ← 0
6. head[H'] ← x
7. H ← BINOMIAL-HEAP-UNION(H, H')
```

(2–4: every thing NIL; 5: degree 0; 6: link of call union)

Eg: Insert 7, 2, 14, 17, 1, 11, 6, 8, 15, 10, 20



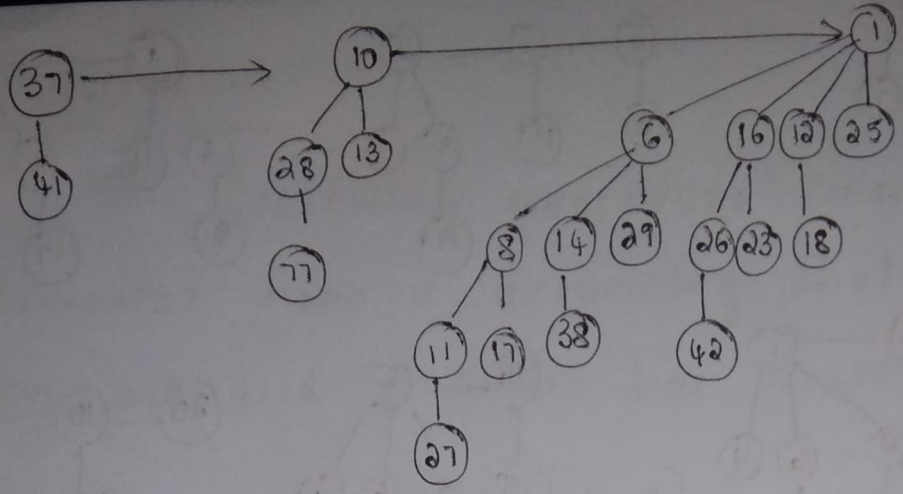
5) Extracting Minimum key

The following procedure extracts the node with minimum key from binomial heap H & returns a pointer to the extracted node.

BINOMIAL-HEAP-EXTRACT(H)

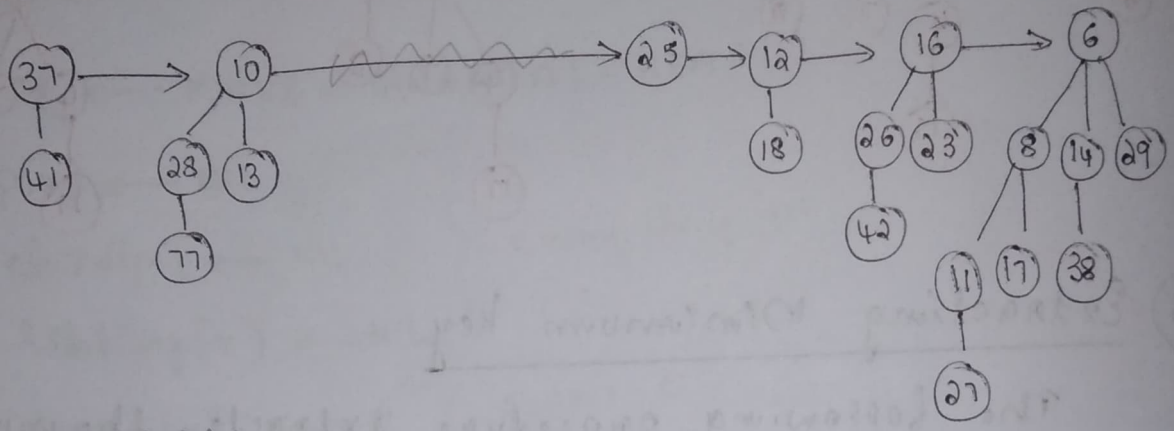
1. Find the root x with the minimum key in the root list of H, & remove x from the rootlist of H.
2. H' ← MAKE-BINOMIAL-HEAP()
3. Reverse the order of the linked list of x's children & set head[H'] to point to the head of the resulting list.
4. H ← BINOMIAL-HEAP-UNION(H, H')
5. return x

a)

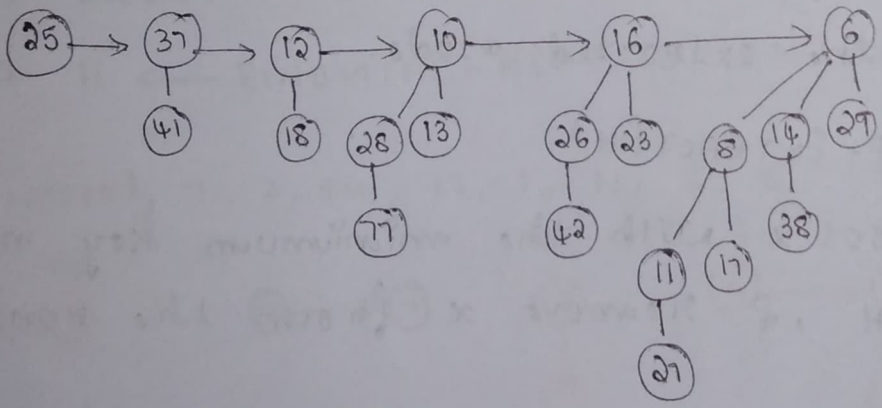


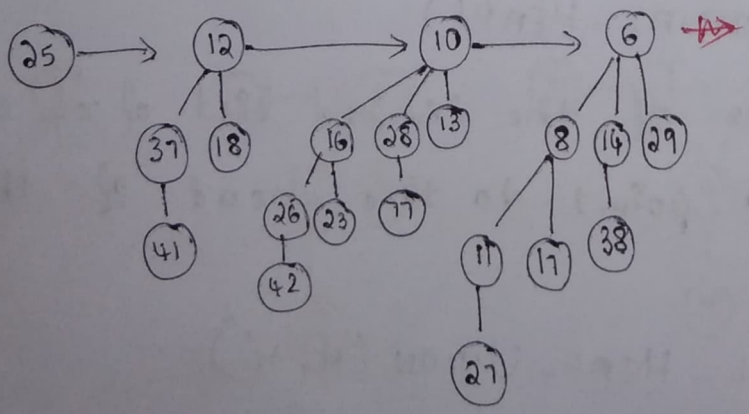
extract min
Find min
Split & reverse

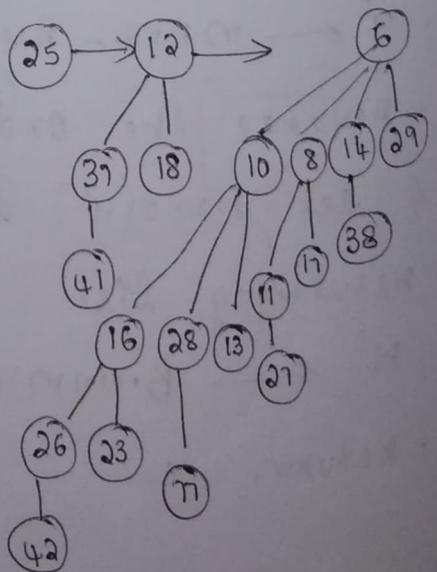
b)



Union – Merge







5) Decreasing a Key

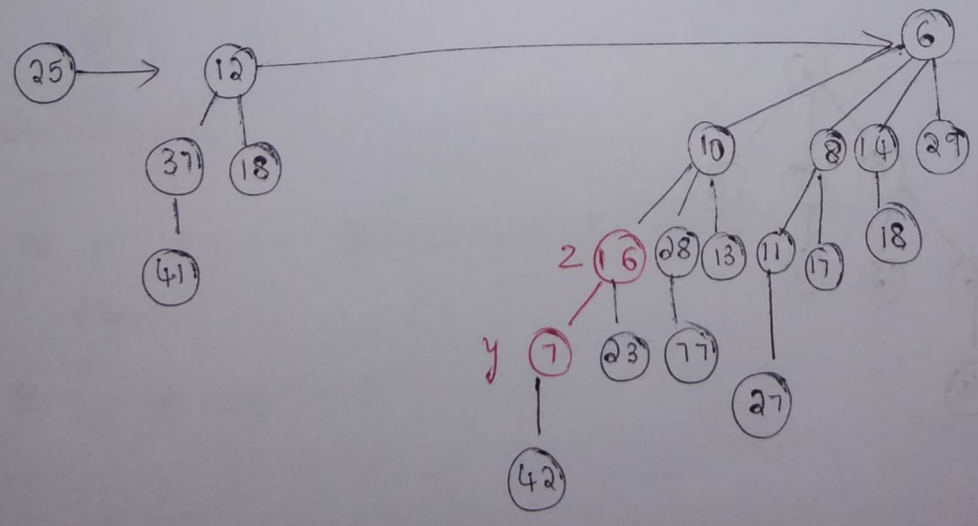
The following procedure decreases the key of a node x in a binomial heap H to a new value k. It signals an error if k is greater than x's current key.

BINOMIAL-HEAP-DECREASE-KEY(H, x, k)

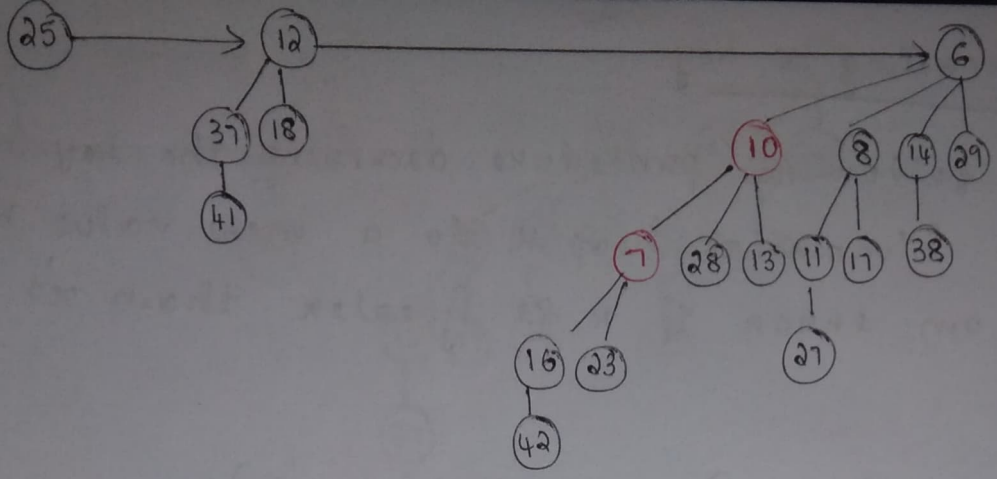
1. if $k > key[x]$
2. then error "new key is greater than current key"
3. $key[x] \leftarrow k$
4. $y \leftarrow x$
5. $z \leftarrow p[y]$
6. while $z \neq NIL$ and $key[y] < key[z]$
7. do exchange $key[y] \leftrightarrow key[z]$
8. if y & z have satellite fields, exchange them too
9. $y \leftarrow z$
10. $z \leftarrow p[y]$

$x \rightarrow$ old
$y \rightarrow$ new
$z \rightarrow$ parent

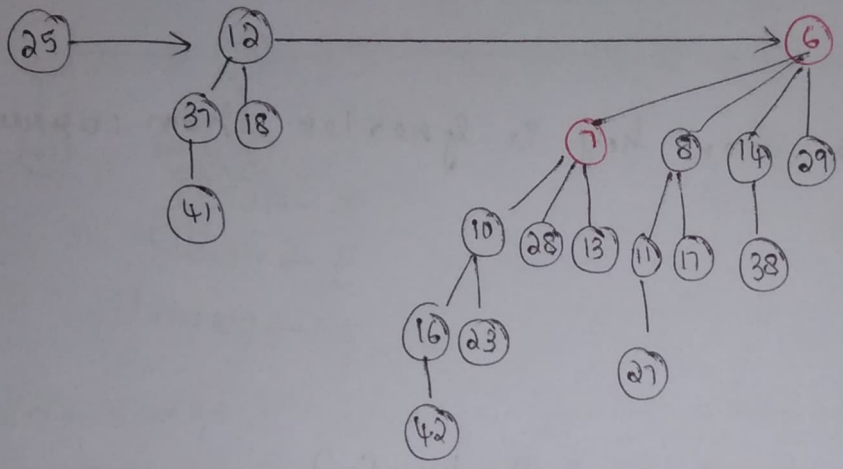
9)



b)



c)

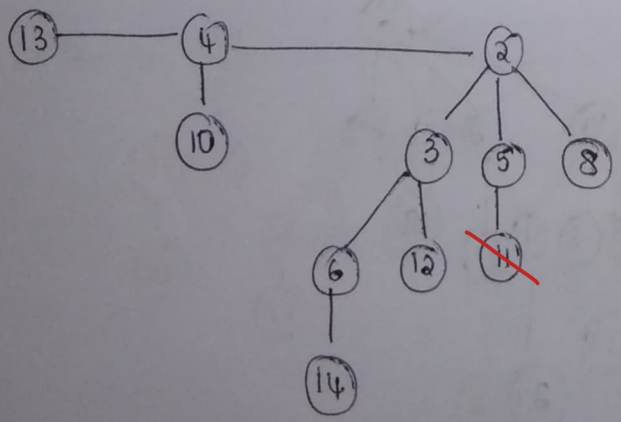


6) Deleting a key

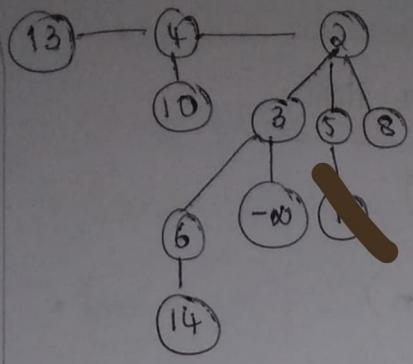
BINOMIAL-HEAP-DELETE(H, x)

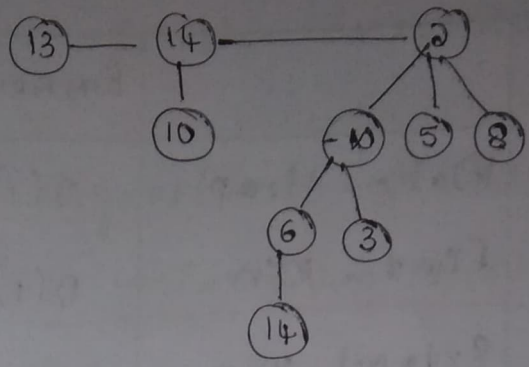
1. BINOMIAL-HEAP-DECREASE-KEY(H, x, $-\infty$) → smallest value
2. BINOMIAL-HEAP-EXTRACT-MIN(H)

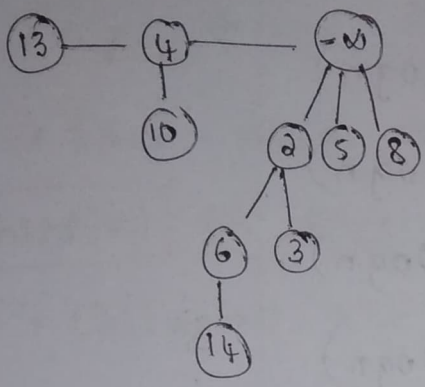
g)

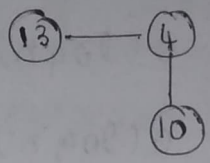


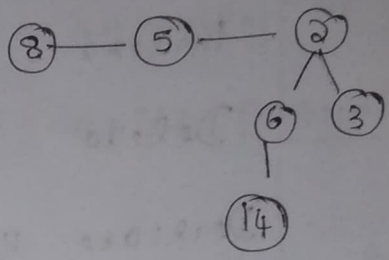
delete (12)



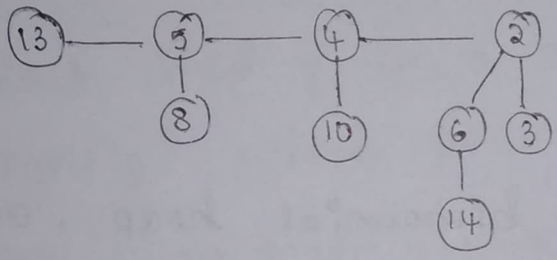


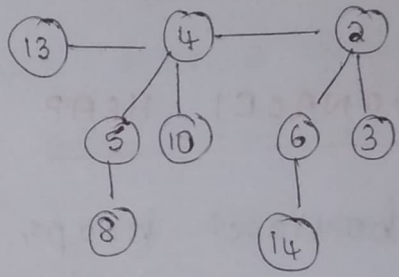




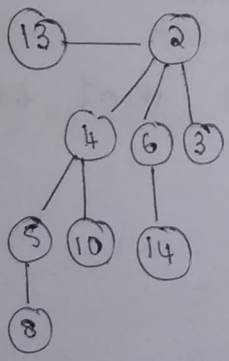


=





=



Analysis

* n node binomial heap have $\log n + 1$ binomial trees.
* heap(minimum) = $\log n$

	BINARY HEAP	BINOMIAL HEAP
Make - Heap	O(1)	O(1)
Find - Min	O(1)	O(logn)
Extract - Min	O(logn)	O(logn)
Insert	O(logn)	O(logn)
Delete	O(logn)	O(logn)
Decrease Key	O(logn)	O(logn)
Union	O(n)	O(logn)

6/02/2021

2. FIBONACCI HEAP

* Fibonacci heaps, like binomial heap, are a collection of min-heap ordered trees.

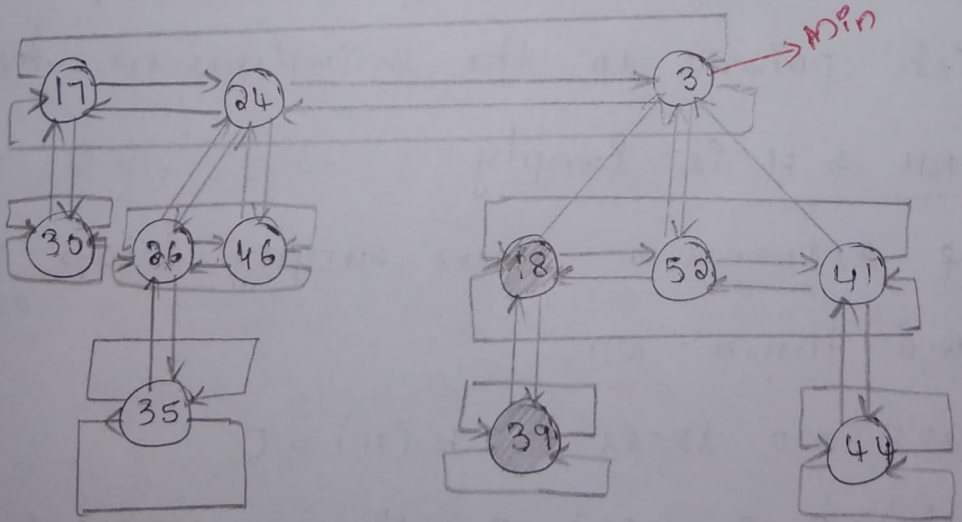
* The trees in fibonacci heap are not constrained to be a binomial tree.

properties

- Unlike binomial trees, fibonacci heap can have many trees of same degree & it doesnot contain 2^k nodes.
- Nodes in a fibonacci heap are not Ordered (by degree) in the root list or as siblings.
- Root & sibling list kept as circular ~~doubly~~ linked list.

- Allows constant time deletion/insertion/concatenation.
- Each node stores its degree (no. of children).
- min(H) is a pointer to minimum root in root list.
- n(H) keeps number of nodes currently in H.
- Each node x has pointer p[x] to its parent & child[x] to one of its children.
- Children are linked together in a doubly linked circular list which is called the child list of x.
- Each child y in a childlist has a pointer left[y] & right[y] which points to left & right siblings.
- left[y] == right[y] == y, then y is the only child.
- Each node also has degree[x] indicating the no. of children of x.

Eg:



• Each node also has mark[x], a boolean field indicating whether x has lost a child since the last time x was made the child of another node.

• Some nodes will be marked

i) A node x will be marked if x has lost a child since the last time that x was made a child of another node.

ii) Newly created nodes are unmarked.

iii) When node x becomes child of another node it becomes unmarked.

Operations

i) Creating a new Fibonacci Heap:

- To make an empty F.H, the MAKE-FIB HEAP procedure allocates & returns the F.H object H.
- The entire heap is accessed by a pointer min[H] which points to the minimum key root
- Min(H) = NIL ⇒ H is Empty
- creates & returns a new empty heap with H.n = 0 and H.min = NIL.
- There are no trees in H. (H) = 0
- Because t(H) = 0 & m(H) = 0, the potential of the empty F.H. $\phi(H) = 0$.
- Amortized cost = actual cost = $O(1)$.

2) Inserting a Node :

FIB-HEAP-INSERT(H; x):

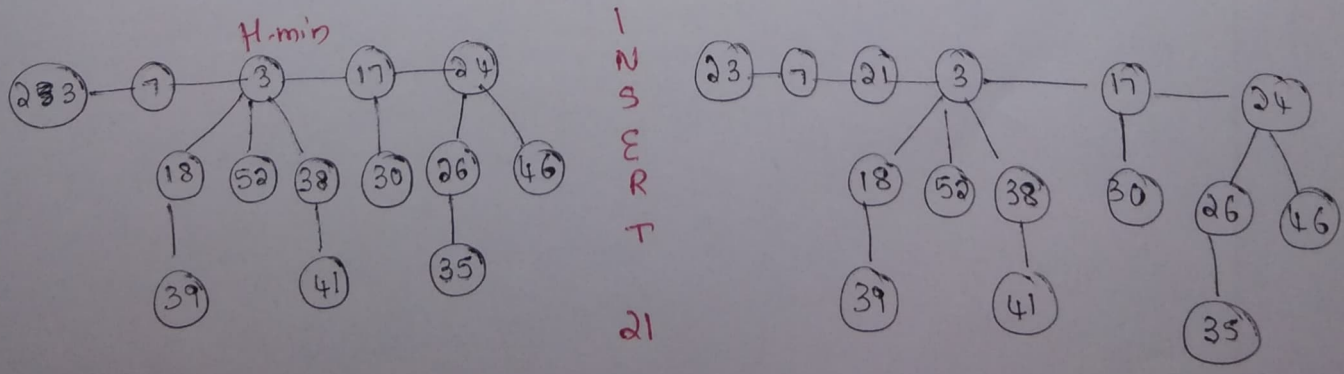
- Creates a new singleton tree
- Add to root list
- Update min pointer

Algorithm :

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

Eg:



(2021) To determine the amortised cost of FIB-HEAP-INSERT, let H be the input F.H & H' be the resulting ~~F.B~~ F.H. Then $t(H') = t(H) + 1$ & $m(H') = m(H)$ & increase in potential is

Trees + 1 Marked Before Inserting

$$((t(H)+1) + 2m(H)) - (t(H) + 2m(H)) = 1$$

Since Actual cost is $O(1)$, the amortised cost is $O(1) + 1 = O(1)$ //

Am cost = Actual cost + difference in pot
= 1 + 1 = 2 → constant → $O(1)$

3) Finding Minimum Node

FIB-HEAP-MINIMUM(H):

The minimum node of a F.H H is given by a pointer H.min, so we can find the minimum node in $O(1)$ actual time. Because the potential of H doesn't change, the amortised cost of this operation is equal to its $O(1)$ actual cost

4) Extracting Minimum Node

- FIB-HEAP-EXTRACT-MIN makes a root of each of the minimum node children & removes the minimum node from the root list.
- Then it consolidates the root list by linking roots of equal degree until at most one root remains of each degree.
- Consolidate(H) consolidates the root list of H

by executing repeatedly the following steps untill every ~~no~~ root in the root list has a distinct degree value.

* Find two roots x and y from the root list with the same degree & with x → key y → key.

* Link y to x, Remove y from the root list & make y a child of x. This operation is performed by FIB-HEAP-LINK.

• Consolidate(H) uses an auxillary array A[0...D(H.n)] to keep track of roots according to their degrees.

If A[i] = y, then y is currently a root with y. degree = i.

FIB-HEAP-EXTRACT-~~NODE~~ MIN(H):

This is the operation where all work delayed by other operation is done.

delayed work = consolidation (merging) of trees.

- Extract min & concatenate its children into root list.

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 rootlist of H
5. x.p = NIL

```
6.  remove z from the rootlist 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
```

CONSOLIDATE(H)

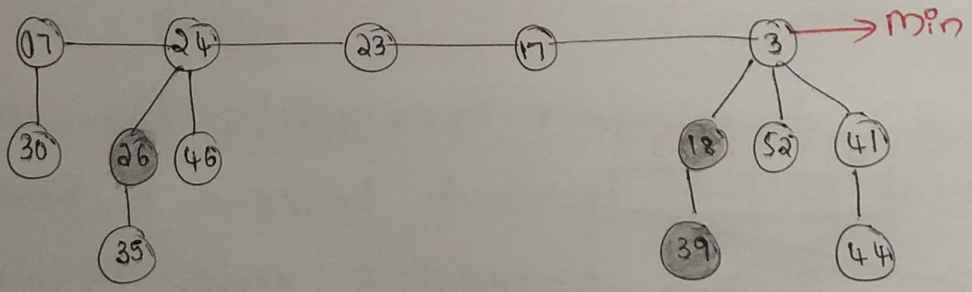
```
1.  let A[0..D(H.n)] be a new array
2.  for i = 0 to D(H.n)
3.      A[i] = NIL
4.  for each node w in the root list of H
5.      x = w
6.      d = x.degree
7.      while A[d] ≠ NIL
8.          y = A[d]        // another node with same degree as x
9.          if x.key > y.key
10.             exchange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

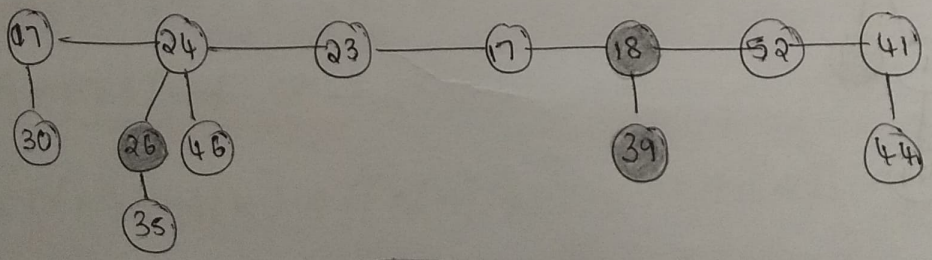
FIB-HEAP-LINK(H, y, x)

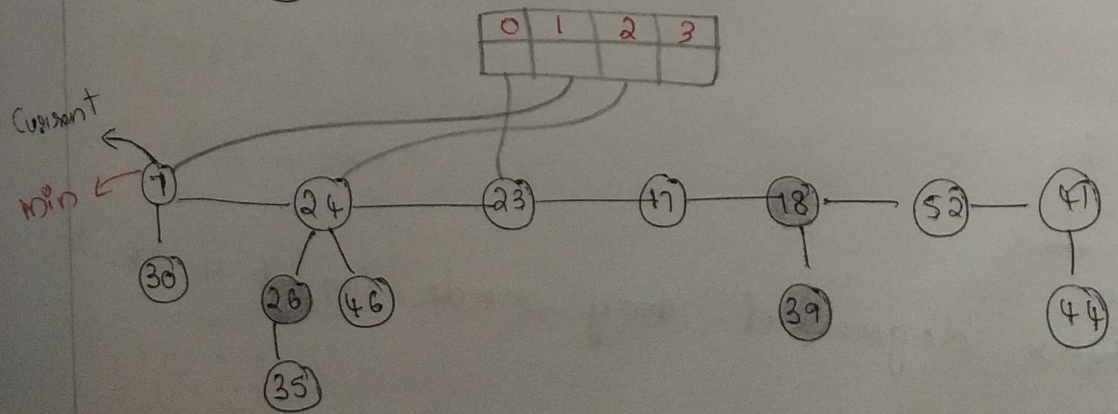
1. Remove y from the root list of H
2. Mark y a child of x, incrementing x, degree
3. y.mark = FALSE

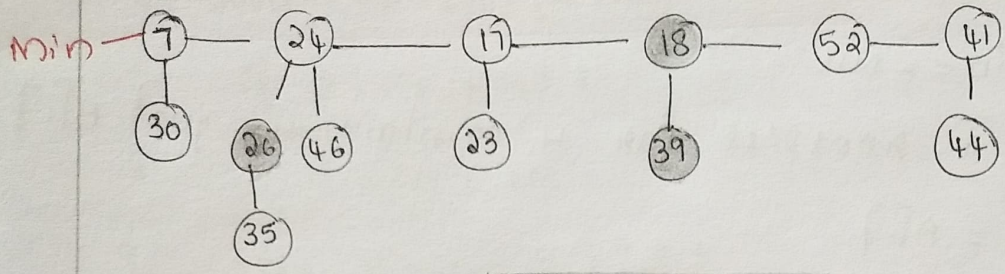
Eg:

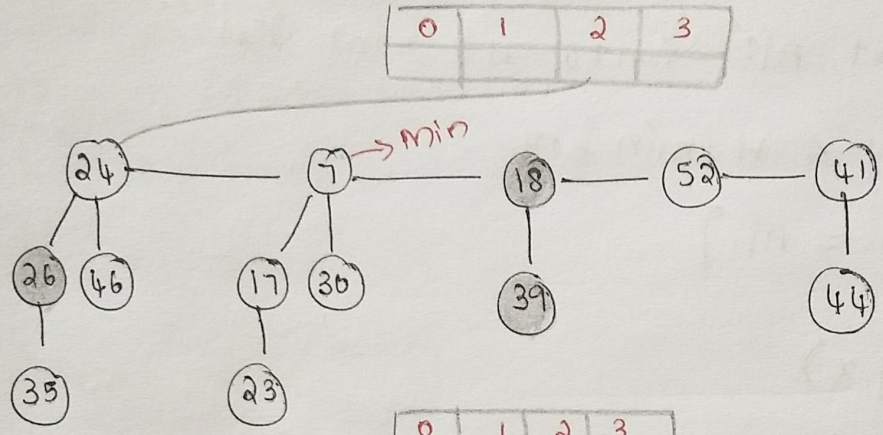


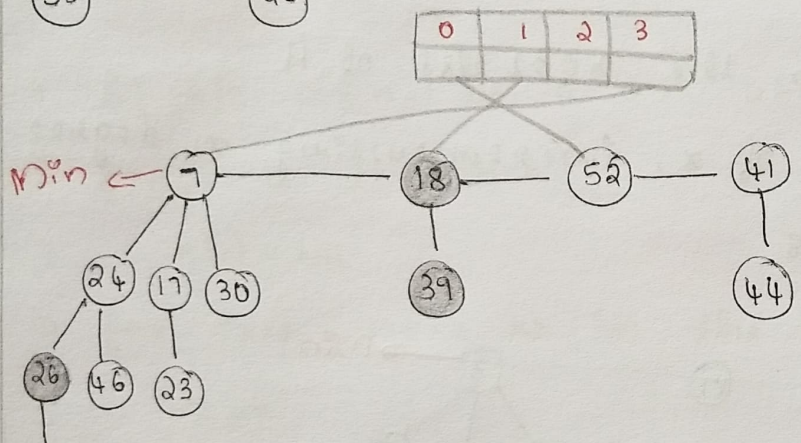
Remove = 3

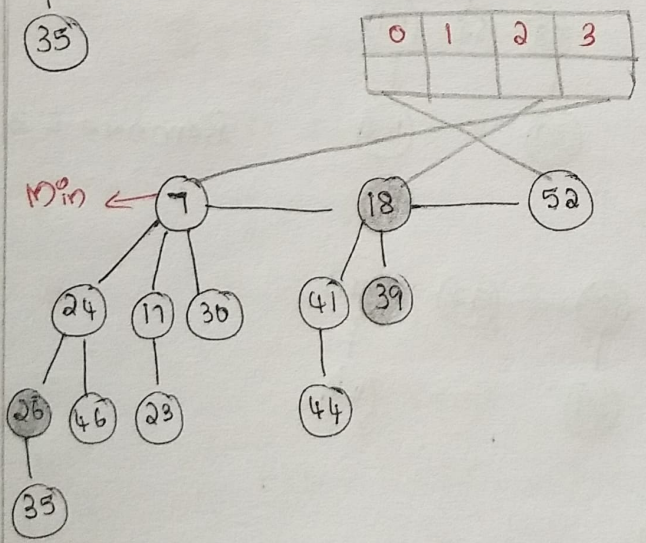












Analysis :

Notation —

D(n) = max degree of any node in F-H with n nodes.

$t(H)$ = # trees in heap H

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

- Actual cost : $O(D(n) + t(H))$

- $O(D(n))$ work adding min's children into root list & updating min.

  * atmost $D(n)$ children of min node.

- $O(D(n)) + t(H)$ work consolidating trees

  * work is proportional to size of root list since number of roots decrease by one after each merging.

  * $\leq D(n) + t(H) - 1$ root node at begining of consolidation.

→ Potential before extracting the minimum node is $t(H) + 2m(H)$

→ Atmost $D(n)+1$ root remain & no nodes become ~~marked~~ marked during the operation ⇒ the potential after extracting the minimum ~~no~~ node is $\leq (D(n)+1) + 2m(H)$

→ Amortized cost =

$$P(D(n) + t(H) + ((D(n)+1) + 2m(H)) - (t(H) + 2m(H))$$

$$= O(D(n)) + O(t(H)) - t(H) = O(D(n))$$

$Dn = \log n$

$\Rightarrow O(\lg n)$

5) UNION OF FIBONACCI HEAP

FIB-HEAP-UNION (H_1, H_2)

1) Consolidate the root list of H_1 & H_2 into new root list H.

2) Set the minimum node of H

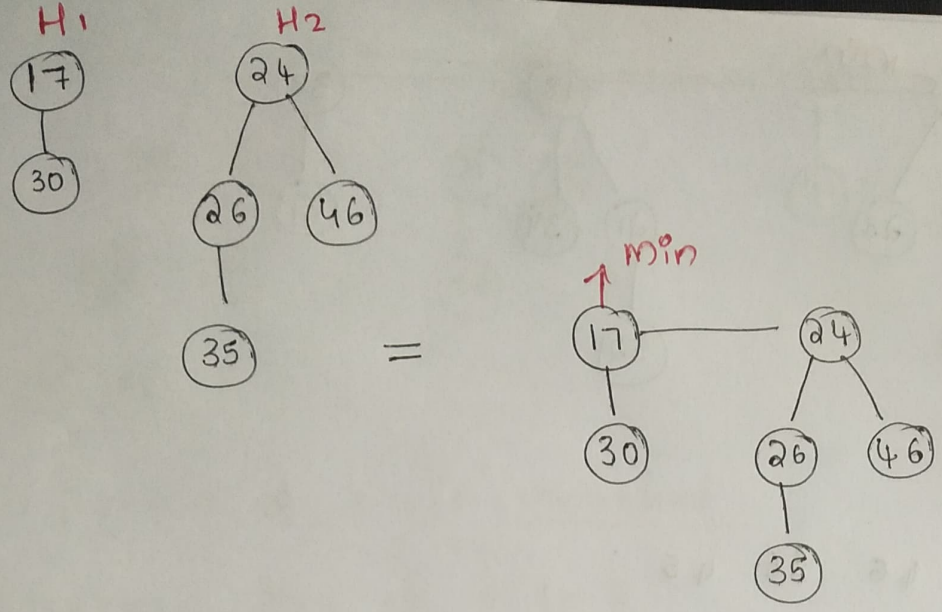
3) Set n[H] to total number of nodes

It simply concatenates the root lists H_1 & H_2, ~~destroy~~ and then determining the new minimum node. Then objects representing H_1 & H_2 will never be used again.

FIB-HEAP-UNION (H_1, H_2)

1. H = MAKE-FIB-HEAP()
2. Hmin = 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 ≠ 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

Eg:



Analysis:

Change in potential is

$\phi(H) - (\phi(H_1) + \phi(H_2))$

$= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + ((t(H_2) + 2m(H_2))$

$= 0$ (since $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$)

$\Rightarrow$ Amortised cost = Actual cost = $O(1)$

10/2/2021

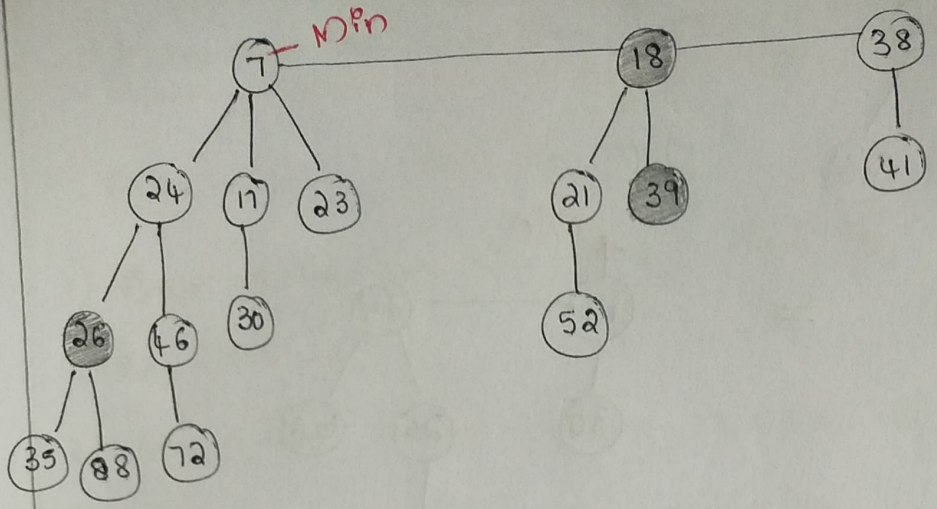
6) Fib heap decrease

To decrease the value of any element in the heap, we follow following algorithm.

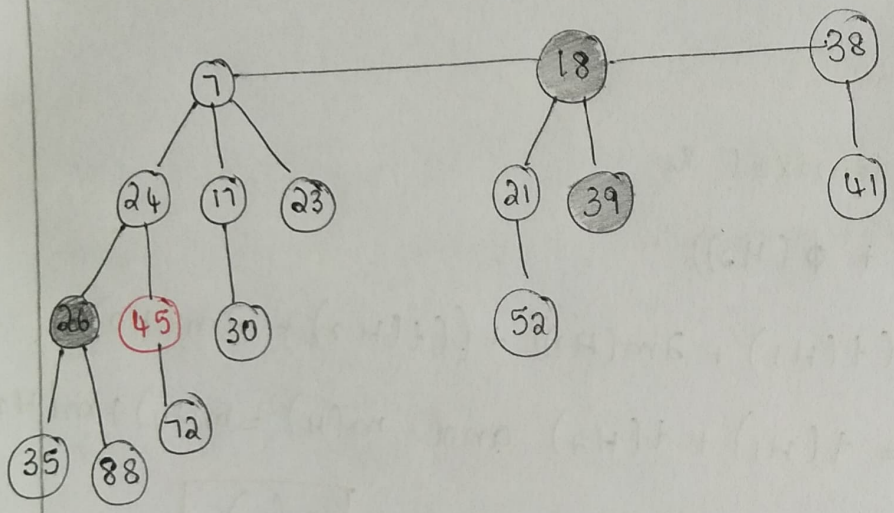
* Decrease the value of the node 'x' to the new chosen value.

CASE 1

1) If min-heap property is not violated,
   * update min-pointer if necessary
   * decrease key of x to k



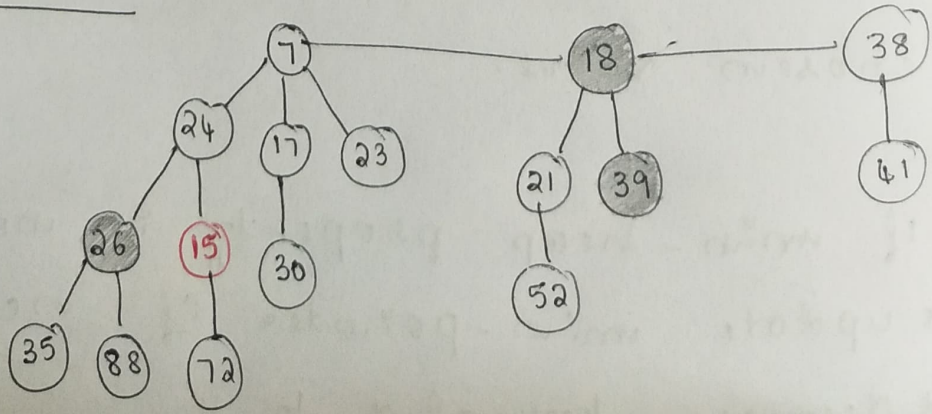
Decrease 46 to 45 :

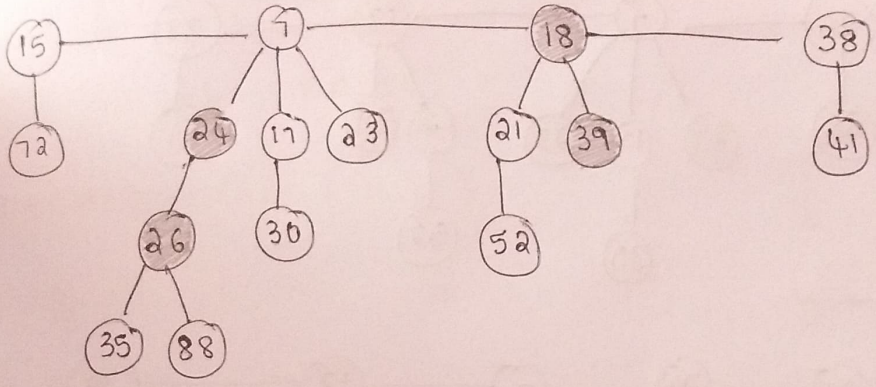


Case 2 : Parent of x is unmarked

- decrease key of x to k
- cut off link between
- mark parent
- add tree rooted at x to root list, updating heap min pointer.

Decrease 45 to 15

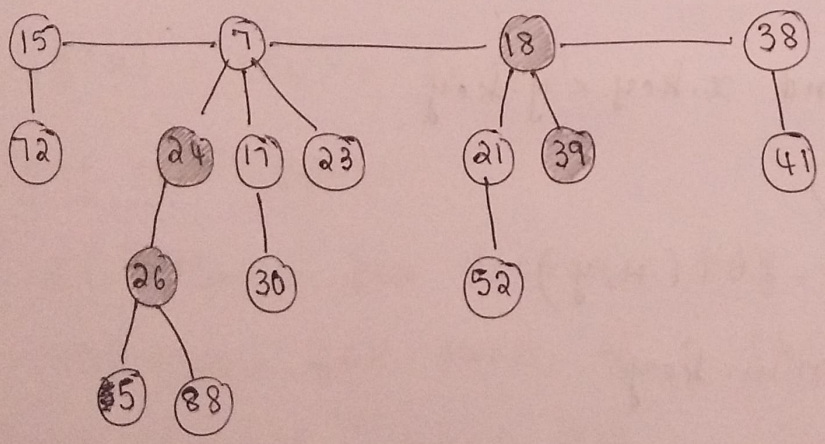


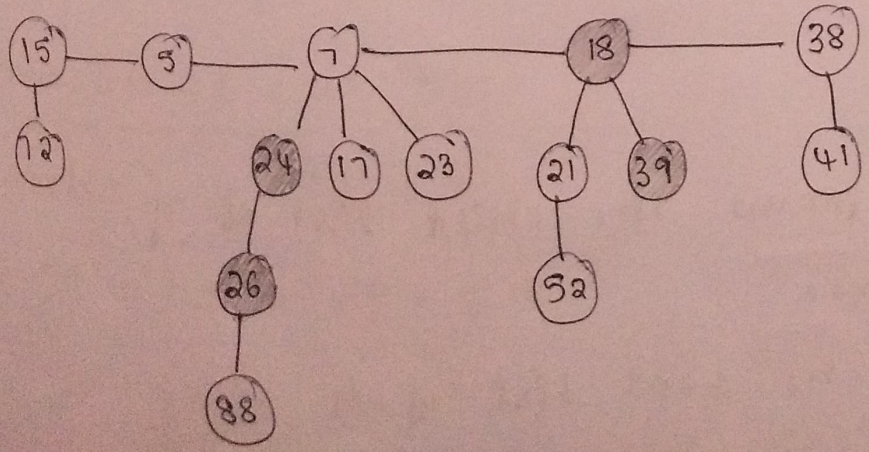


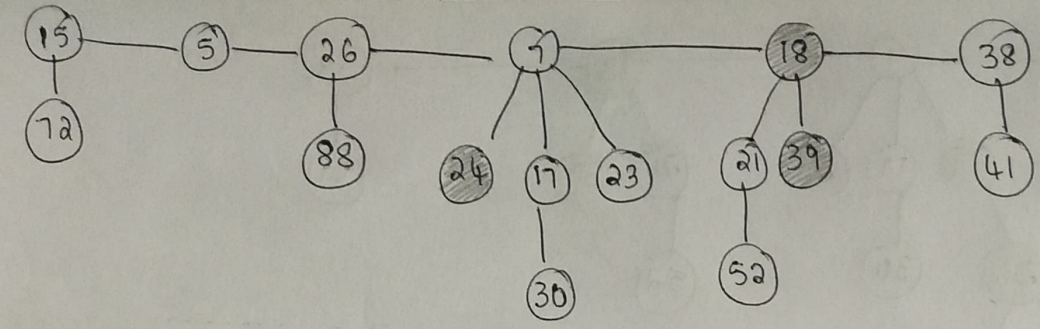
case 3 : parent of x is marked

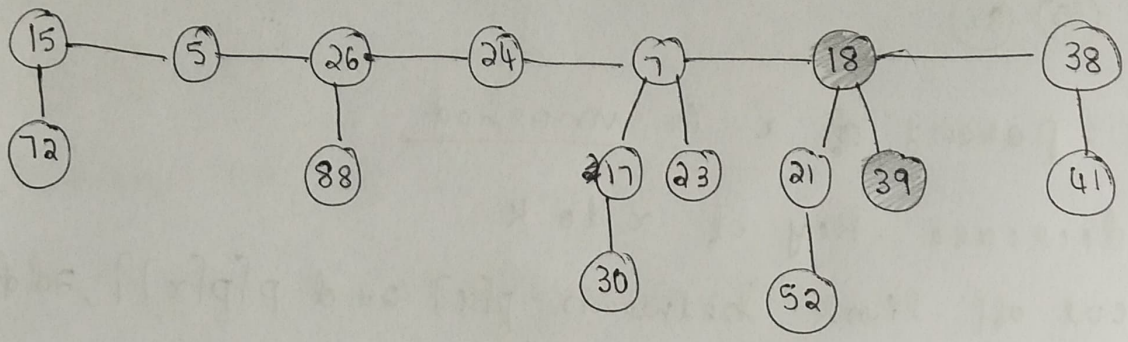
- decrease key of x to k
- cut off link between p[x] and p[p[x]], add p[x] to root list.
  - * if p[p[x]] unmarked, then mark it
  - * if p[p[x]] marked, cut of p[p[x]], unmark & repeat.

decrease 35 to 5









~~Algorithm~~

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)

x = element

y = Parent

z = Grand Parent

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)

Analysis :

Notation :

t(H) = # trees in heap H

m(H) = # marked nodes in heap H

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

Actual cost : O(c) :

O(1) time for decrease key

O(1) time for each of c cascading cuts, plus reinserting in root list.

Amortised cost : O(1)

$t(H') = t(H) + c$

$m(H') \leq m(H) - c + 2$

- each cascading cut, unmark a node
- last cascading cut could potentially mark

a node.

$$\Delta\phi \leq c + a(-c+a) = 4 - c$$

7) Delete a key

The following pseudocode deletes a node from an n-node F.H in $O(D(n))$ amortised time. We assume that there is no key value of $-\infty$ currently in FH.

FIB-HEAP-DELETE(H, x)

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

• FIB-HEAP-DELETE makes x become the minimum node by giving $-\infty$.

• FIB-HEAP-EXTRACT-MIN procedure then removes node x from F.H

• The amortised time of FIB-HEAP-DELETE is the sum of $O(1)$ amortised time of FIB-HEAP-DECREASE KEY & $O(D(n))$ amortised time of FIB-HEAP-EXTRACT-MIN.

$\therefore D(n) = O(\lg n)$