EECS560 Data Structures Kong

Topic 6: Concatenated Queues

Read: Chpt. 6 & 11, Weiss.

Q: Let S_1 and S_2 be two sets of data objects to be organized as a PQ Q1 and Q2. How can we merge/concatenate these two priority queues Q1 and Q2 together to form a new PQ containing the objects in S_1 and S_2?

Observe that if we have implemented a function *concate(Q1,Q2)* that will allow us to merge two PQ Q1 and Q2 together, then we can use this concate function to implement the standard PQ operations insert and delete as follow.

insert(x,Q):
Let x be a PQ and merge it with Q.

deleteMin(Q):
Delete min from Q and decompose Q–{x} into a collection of PQs. Then merge all PQs together.

Remark: A concatenated queue is a priority queue that supports concate operation effectively and also uses the concate operation to implement other standard PQ operations.

ADT: Concatenated (Mergeable) queue.

A collection class with the following operations:

1. ***insert(x,Q)***
2. ***findMin(Q)***
3. ***deleteMin(Q)***
4. ***concate(Q1,Q2)***
5. create(Q)
6. destroy(Q)
7. isEmpty(Q)

Q: How efficiently can we implement the concate(Q1,Q2) operation?

Assuming that $|S_1| = m$ and $|S_2| = n$, $m \leq n$:

Approach 1: We can merge Q1 and Q2 together by inserting the objects from Q1 into Q2.

Previous data structures ineffective since

BST:	$T_w(m,n) = O(mn) = O(n^2)$.
k-Heap:	$T_w(m,n) = O(m\lg n) = O(n\lg n)$.
2-3 Tree:	$T_w(m,n) = O(m\lg n) = O(n\lg n)$.
Minmax Heap:	$T_w(m,n) = O(m\lg n) = O(n\lg n)$.

Approach 2: We can build a new PQ using objects from both Q1 & Q2.

Previous data structures again are ineffective since

BST:	$T_w(m,n) = O((n+m)^2) = O(n^2)$.
k-Heap:	$T_w(m,n) = O(n+m) = O(n)$.
2-3 Tree:	$T_w(m,n) = O((m+n)\lg(m+n)) = O(n\lg n)$.
Minmax Heap:	$T_w(m,n) = O(n+m) = O(n)$.

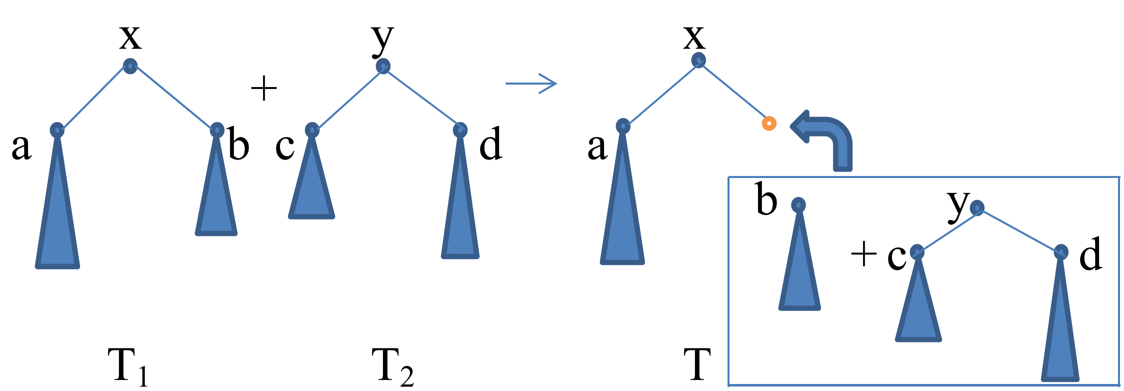
Q: Can we do it better?

Concatenated (Mergeable) Heaps:
A concatenated heap is a class of concatenated queues formed by using (min) heap-ordered trees.

Q: How do we merge two heap-ordered trees together?
It depends on the structure of the heap-ordered trees.

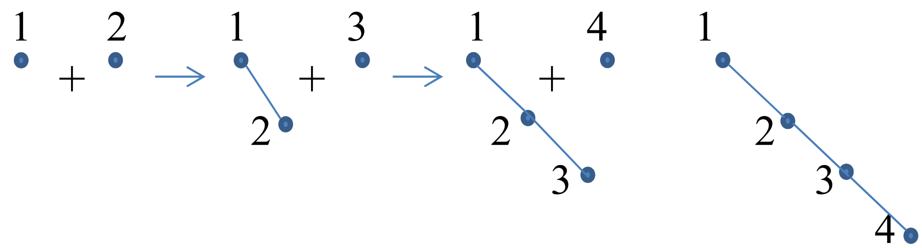
Two Basic Approaches in Merging Heap-Ordered Trees:
1. If the trees are binary trees, use *recursive merging*.

Approach: Assume that $x \leq y$. The new binary tree T is formed using
(1) x is the root of T,
(2) the left subtree of T is the old left subtree of x, and
(3) the right subtree of T is formed by recursively merging the old right subtree of x with the remaining tree T_2.



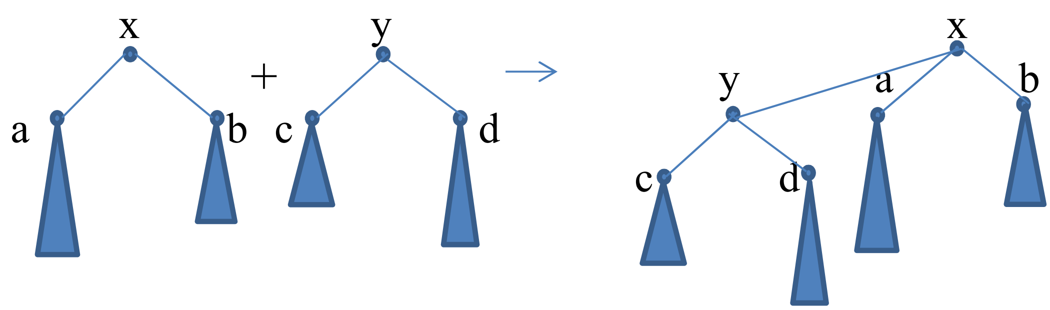
Remark: Complicate merging process, may result in skew tree.

Example: Recursive merging resulting in skew tree.

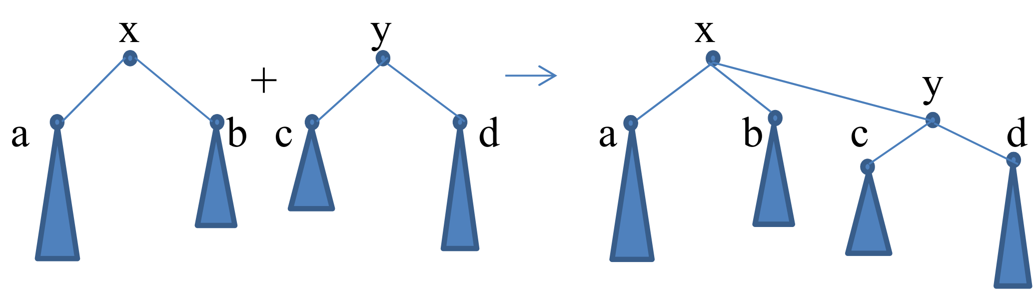


2. If the trees are general trees, use *lazy merging*.

Approach: Assume that $x \leq y$. The new binary tree T is formed by making y a new child of x. Depending on the underlying data structure, y may become either the first child, or the last child, of x.



Or,



Remark: Extremely efficient merging process with cost O(1) but resulting in much more complicate tree structure.

Some Useful Concatenated Heap structures:

Unless specified otherwise, we will always be using min heap-ordered trees in discussion.

I. **Leftist heap** (C.A. Crane):

A leftist heap is a heap-ordered leftist tree.

Dfn: Given a binary tree T. For each node x in T, define the *rank* of x as follow:

rank(x) = length of a shortest path going from x to an external node in T_E, where T_E is the extended binary tree of T.

Remark: If y is an external node in T_E, we can define rank(y) = 0.

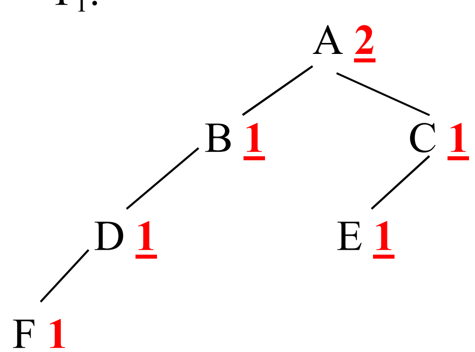
Dfn: A *leftist tree* T is a binary tree such that

(1) $T = \varnothing$, or

(2) If $T \neq \varnothing$, for every node $x \in T$, we have rank(left_child(x)) $\geq$ rank(right_child(x)).

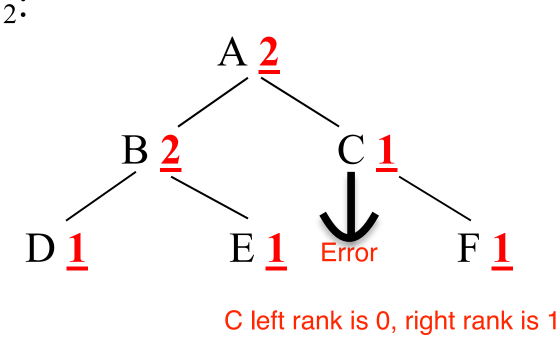
Example: Two binary trees with ranks.

T_1:



T_1 is a leftist tree.

T_2:



T_2 is not a leftist tree

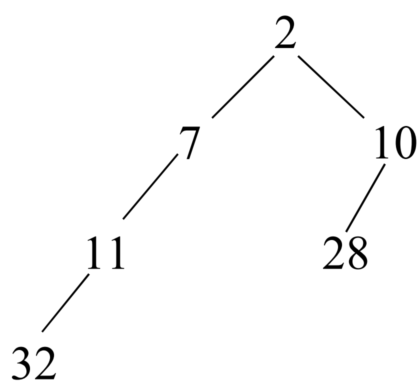
A *leftist heap* satisfies the following two properties:

Structural property: A leftist tree

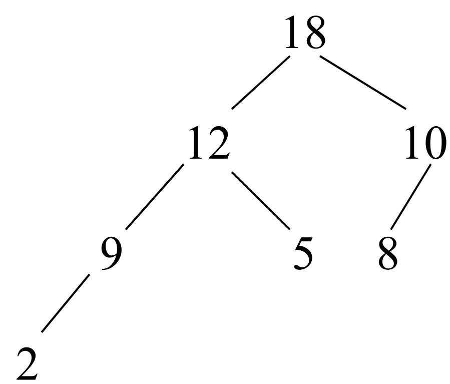
Relational property: Heap-ordered tree

Examples of leftist heaps:

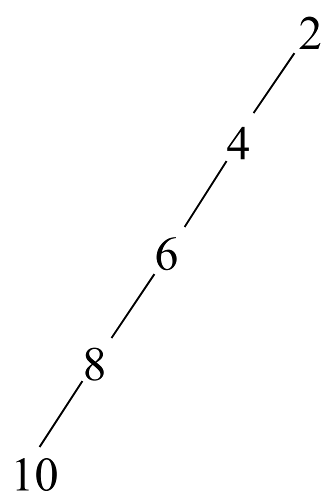
Min leftist heap:



Max leftist heap:



A skewed min leftist heap:



Remarks:

(1) A leftist tree may not be a balanced binary tree.

(2) If a leftist heap operation depends on the height of the tree, we will have $T_w(n) = O(n)$ complexity, which offers no gain in performance.

(3) Since a leftist tree is "left-heavy," all operations should always avoid using the left subtree (path) of a node.

(4) Leftist tree operations will always operate on the right subtree of a node in T.

(5) No path from the root to an external node is shorter than the path that always uses the right child in going from the root to an external node.

Theorem: Let r be the root of a leftist tree with n nodes. We have $n \geq 2^{\mathrm{rank}(r)} - 1$.

Corollary: A leftist tree with n nodes has a right path going from the root to an external node containing at most $\lfloor \lg(n+1) \rfloor$ nodes.

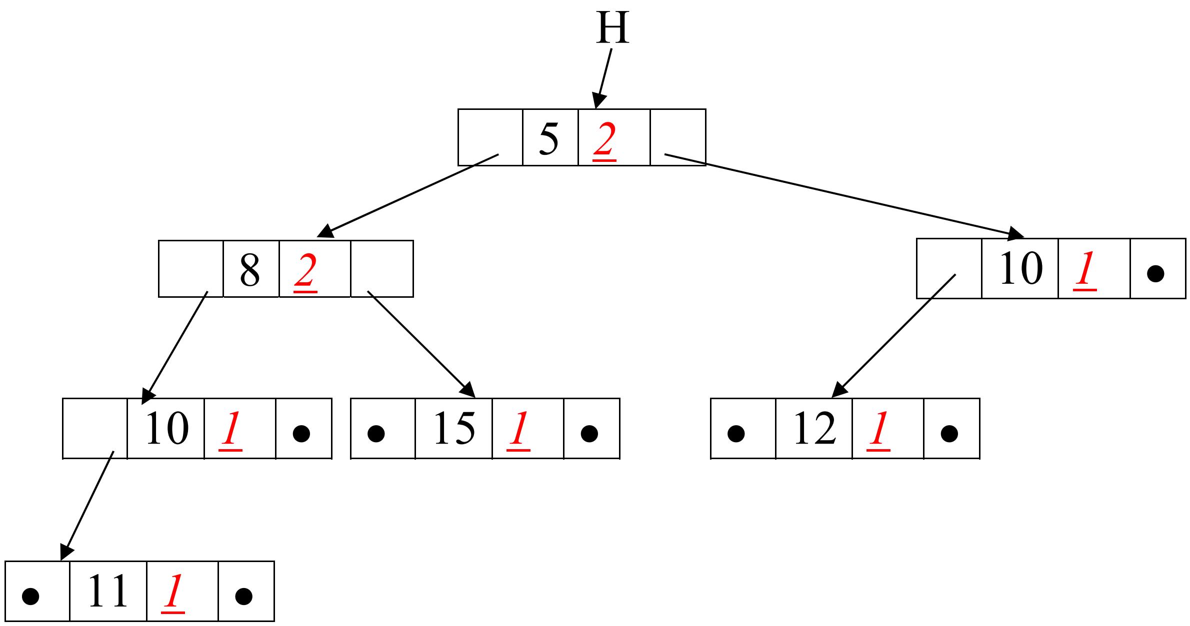
Implementation:

Array implementation infeasible, use pointers! (Why?)

Node:

lchild	**rank**	**key**	rchild

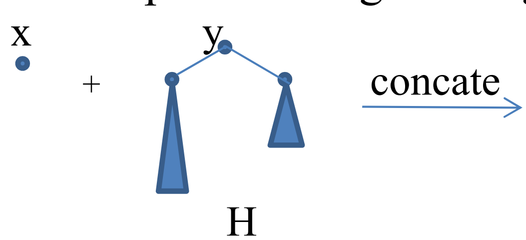
Example: A min leftist heap.



Leftist heap operations:

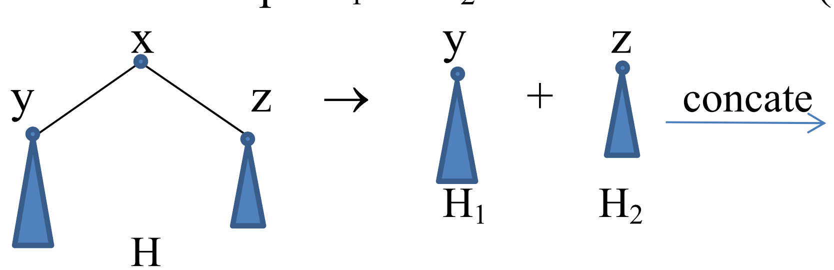
1. *insert(H,x)*:

Observe that a single element by itself is a leftist heap. Hence, we can perform concate(H1,H), where H1 is the leftist heap containing the single element x.



2. *deletemin(H)*:

After executing the deleteMin operation, we have left with two leftist heaps H_1 & H_2. Perform concate(H1,H2).

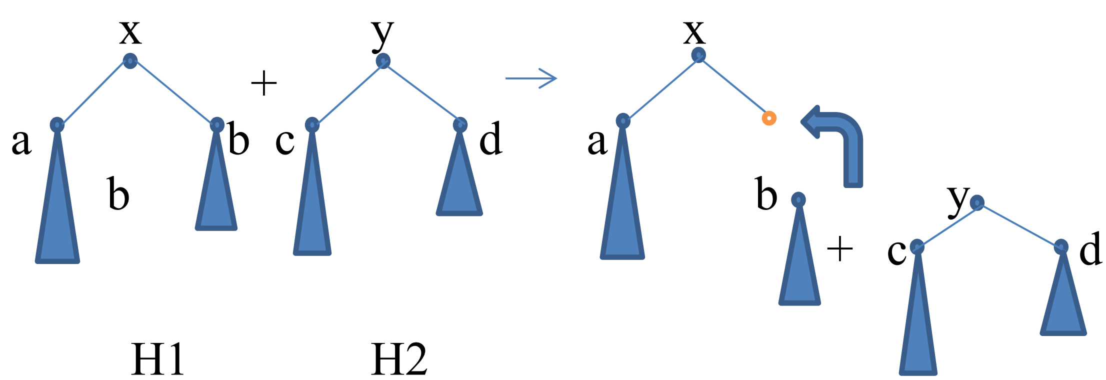


3. *concate(H1,H2)*:

Merge the two leftist heaps H1 and H2 together and then stored the resulting leftist heap as H1.

Q: How do we merge two leftist heaps together?

Assuming that we are using min leftist heap and the roots of H1 and H2 are x, and y, respectively with x ≤ y.



Remark: We will always operate on the right side of a leftist heap.

Algorithm:

if H1or H2 = $\varnothing$

then return the other heap;

else compare x with y;

if x > y

then swap(H1,H2)

endif;

merge H2 with the ***RIGHT*** *subheap of x;*

attach the resulting tree as the right child of x;

compute rank(x);

if (rank(lchild-of-x) < rank(rchild-of-x))

then swap(lchild-of-x,rchild-of-x)

endif

endif;

Q: Do we always obtain a heap-ordered tree after the merging of H2 with the **right** subheap of H1?

Yes.

Q: Do we always get back a leftist tree?

Not necessarily!

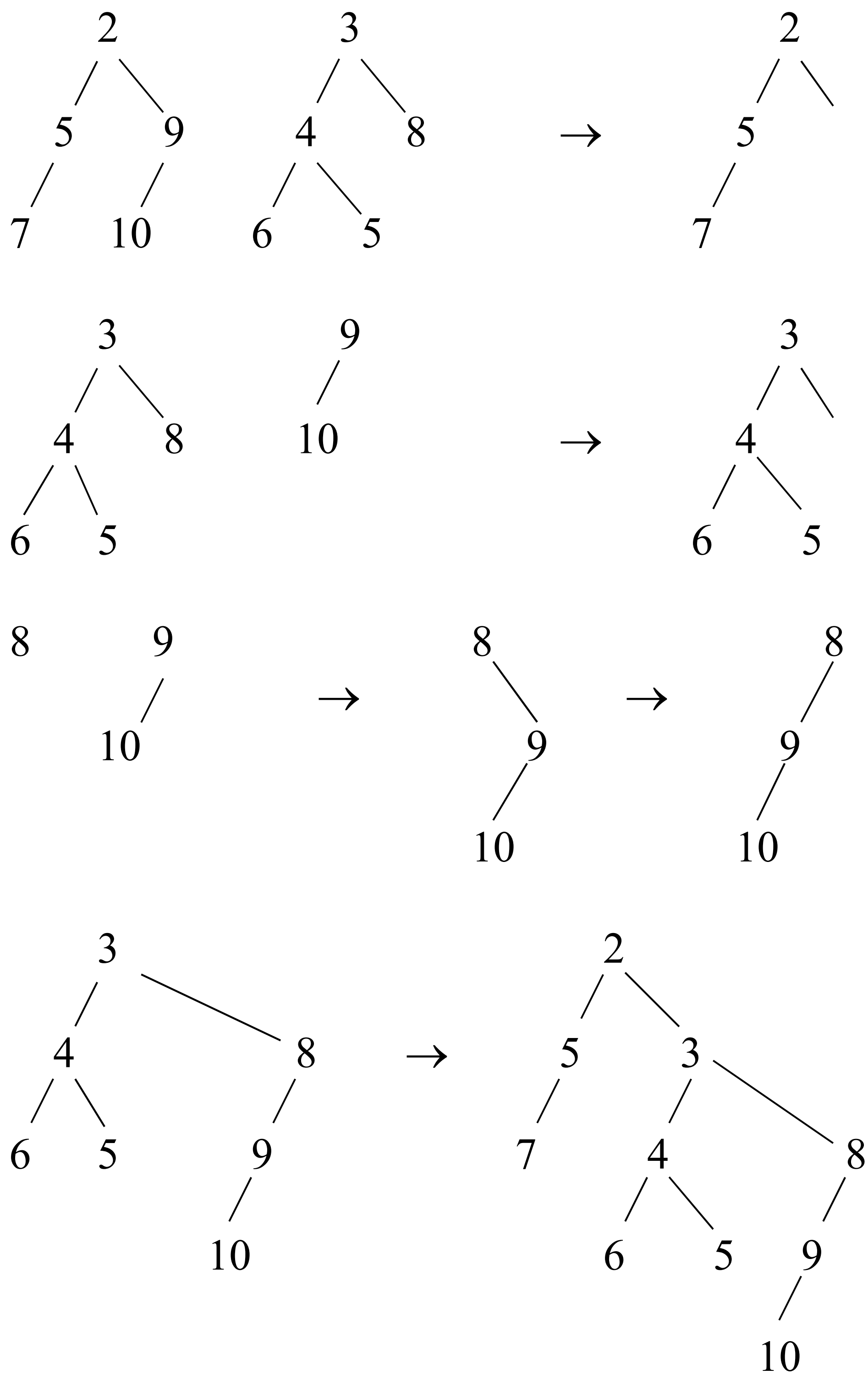
Remedy:

One must always check the ranks of the two children of x after each re-attachment and swap the two subtrees of x whenever leftist heap property is not satisfied.

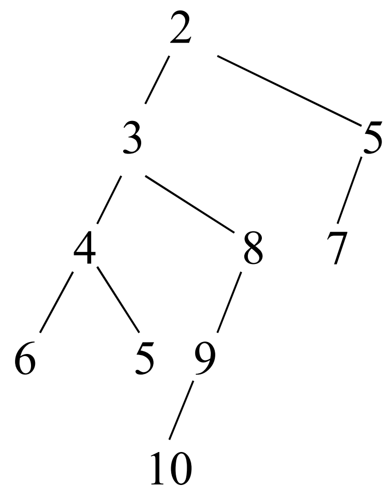
Merging two lefties heaps:

```
Merge(Node *H1, Node *H2)
   if  H1 = null
      then return H2;
      else  if H2 = null
               then return H1;
            else if (key(H1) > key(H2))
                     then swap(H1,H2)
                  endif;
                  H1→rchild := Merge(H1→rchild, H2);
                  adjust rank(H1);
                  if rank(H1→lchild) < rank(H1→rchild)
                        then swap(H1→lchild,H1→rchild)
                  endif;
                  return H1;
            endif;
   endif;
end Merge;
```

Example: Merging two leftist heaps.



Final min leftist heap:



Complexity for Leftist Heap:

concate, insert, deletemin: $T_w(n) = O(\lg n)$.
findmin: $T_w(n) = O(1)$.
build_heap using insert operations: $T_w(n) = O(n\lg n)$.

Q: Can we omit the rank info so as to simplify the concate operation? If so, how do we perform concate(H1,H2) if H1 and H2 are just two heap-ordered trees with no additional structural property imposed on them?

II. **Skew Heap** (Sleator & Tarjan):

A heap-ordered binary tree with

Structural property: A binary tree,
Relational property: Heap-ordered tree.

Observation:

Since we do not have the rank info, if we merge H1 and H2 as in leftist heap, it may result in long right path!

Possible Remedy:

Always swap left_child with right_child after ***every*** merge operation.

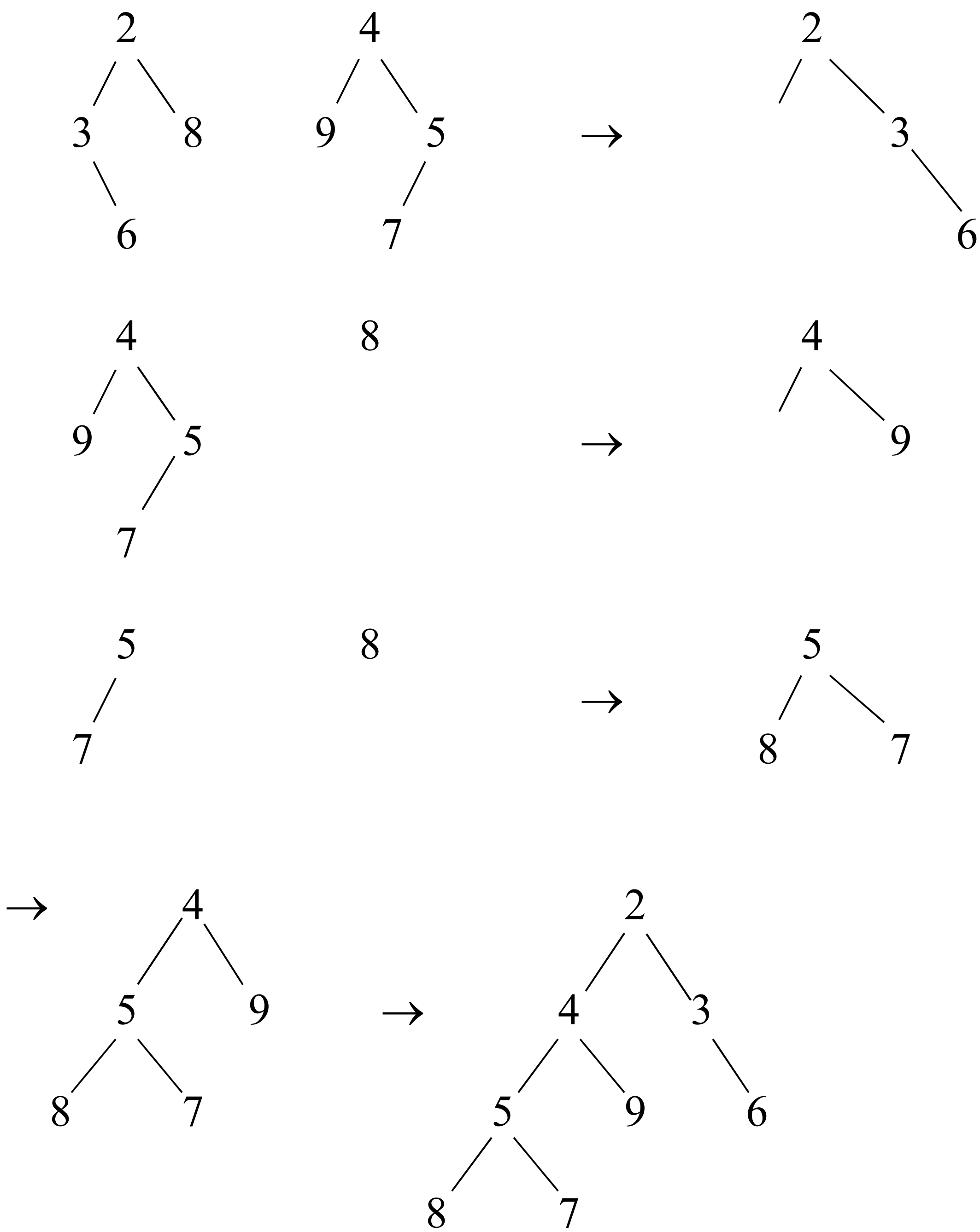
Merging two skew heaps:

Assuming that we are using min leftist heap and the roots of H1 and H2 are x, and y, respectively with $x \leq y$.

Algorithm:

```
Merge(Node *H1, Node *H2)
  if H1 = null
    then return H2;
    else if H2 = null
           then return H1;
         else if (key(H1) > key(H2))
                then swap(H1,H2)
              endif;
              *Node temp := H1→rchild;
              H1→rchild := H1→lchild;
              H1→lchild := Merge(temp, H2);
              return H1;
         endif;
  endif;
end Merge;
```

Example: Merging two skew heaps.



Complexity for Skew Heap:

concate, insert, deletemin: $T_w(n) = O(n)$.

findmin: $T_w(n) = O(1)$.

build_heap using insert operations: $T_w(n) = O(n^2)$.

Amortized Cost for Skew Heap:

For m insert, deletemin, or concate operations,

$T_w(m,n) = O(m\lg n)$.

Amortized complexity per operation:

$T_w(n) = O(\lg n)$. (Same as leftist heap)

Remark:

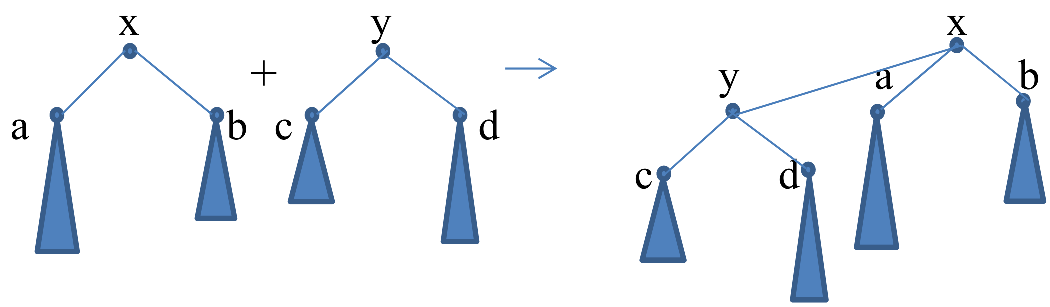
Skew heap is a class of self-adjusting concatenate queue. It uses less memory and is more efficient than leftist heap with large m and n.

Comparing leftist heap with skew heap:

	Leftist Heap	**Skew Heap**
Tree structure	Leftist tree	Binary tree
Implementation	Children pointers	Children pointers
Node structure	Need info on rank	No info on rank
Merging	Must verify rank info	No rank info
Re-attachment	May swap	Always swap

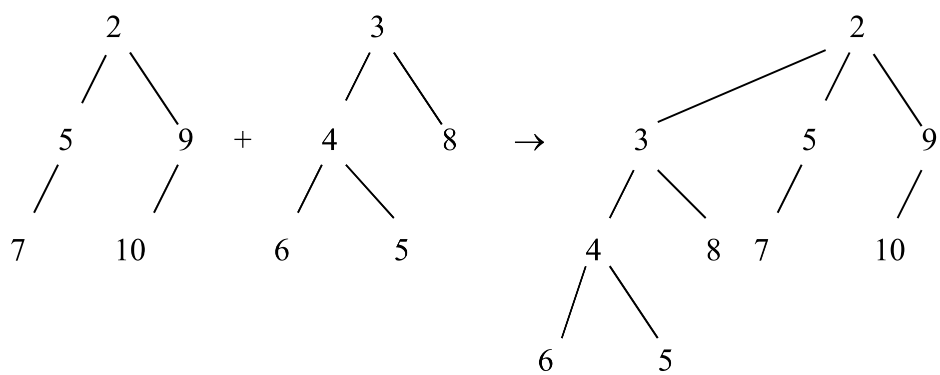
III. **Pairing Heap** (Fredman, Sedgewick, Sleator, Tarjan):
Lazy Approach in Merging Two Heap-Ordered Trees:

If $x \le y$, simply make y the first child of x.



H1 H2 concate(H1,H2)

Example: Merging two pairing heaps.



Q: How can we incorporate this simple merge concept into the design of a concatenate queue?

Pairing Heap:

Structural Property: General tree.

Relational Property: Heap-ordered tree.

Observations: Given a set of n objects to be represented using a pairing heap H.

1. Maximum degree of a node (root) in H = n−1.
2. Maximum height of H = n−1.

(Min) Pairing Heap Operations:

1. Merge(H1,H2):

   Assuming that root of H1 ≤ root of H2, make H2 the new first child of H1. If not, swap the trees and then merge.

2. Insert(x,H):

   Let x be a single-node heap and merge it with H.

3. DeleteMin(H):

   Delete the root of H to decompose H into ≤ n−1 heap-ordered trees and then merge the trees together.

Observation:

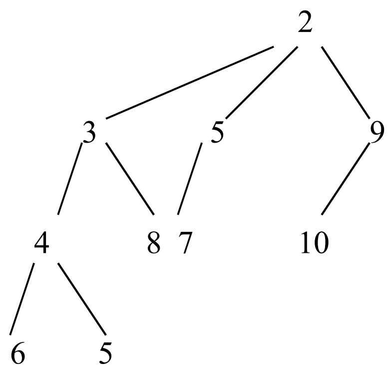
If merging two pairing heaps can be done in O(1) time, then insert can be done in O(1) time and deleteMin can be done in O(n) time.

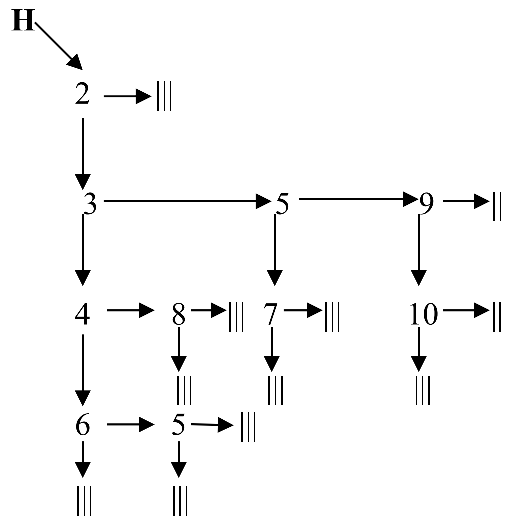
Implementation:

Array: Infeasible. (Why?)
Children pointers: Infeasible. (Why?)

We will use left-child right-sibling implementation to allow merging with O(1) cost.

Example:





Remark: Using this left-child right-sibling implementation, merge(H1,H2) can be performed in $T_w(n) = O(1)$ time as discussed above. (HW)

Remaining Problems:

How do we merge the subtrees together after a deleteMin operation? And in what order should we merge the subtrees together?

Merging Subtrees in a Pairing Heap:

1. Merging subtrees randomly:

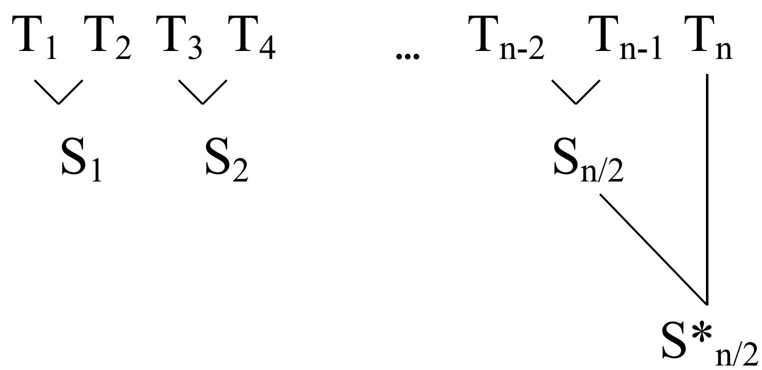
   Merge the subtrees two at a time in any order.

   Remark: Undesirable since we either have to use a random number generator or the process cannot be duplicated.

2. Two-Pass Method:

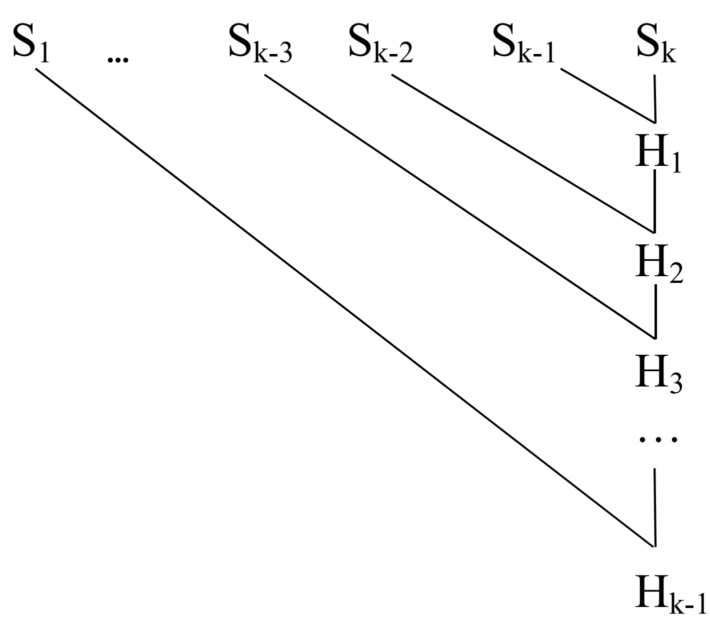
First pass:

Merge pairs of subtrees from left to right. If the number of subtrees is odd, merge the last remaining subtrees with the last newly merged subtree.



Second pass:

Starting from the last (rightmost) tree obtained above, from right to left and one at a time, merge it with the remaining trees to form a single tree.



3. Multi-Pass method:

```
create a FIFO queue Q and store all trees in Q;
while Q ≠ ∅ do
   R ← deque(Q);
   if Q ≠ ∅
         then  K ← deque(Q);
               enque(merge(R,K));
   endif;
endwhile;
return R;
```

Example: Consider merging 7 trees.

T1 T2 T3 T 4 T5 T6 T7
T3 T4 T5 T 6 T7 R1
T5 T6 T7 R1 R2
T7 R1 R2 R3
R2 R3 S1
S1 S2
H1

Return R = H1.

Complexity on Merging Trees:

Same for all three methods asymptotically with $T_w(m) = O(m)$ in merging m subtrees.

Complexity for Pairing Heap:

concate, insert, findmin: $T_w(n) = O(1)$.

deleteMin: $T_w(n) = O(n)$.

build_heap using insert operations: $T_w(n) = O(n)$.

Amortized Complexity of Pairing Heap Operations:

Concate, insert, deleteMin:

$T(n) = O(\lg n)$. (amortized)

Tw(n) insert concate deleteMin findMin build
lgn lgn lgn lgn nlgn

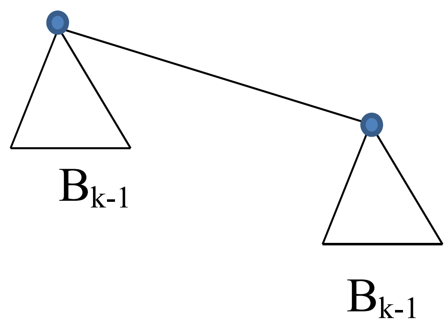
IV. Binomial Queue

Consider the following class of binomial trees (BT).

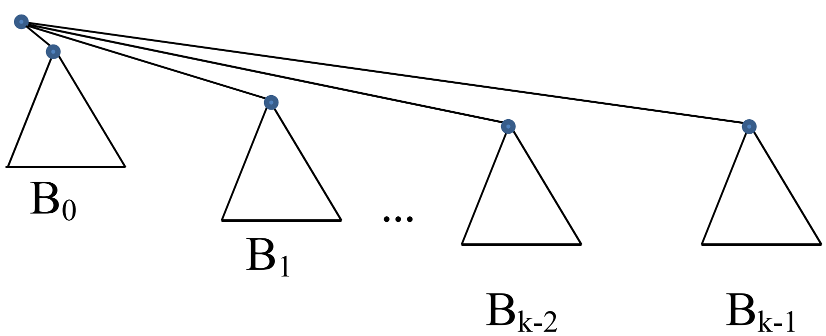
Recursive Definition of BT of Order k, $k \geq 0$:

- A single node is a BT of order 0, B_0.
- A BT of order k, $k > 0$, B_k is formed by "melding" two BTs of order k-1 together by making the root of one tree be the child of the root of the other tree.

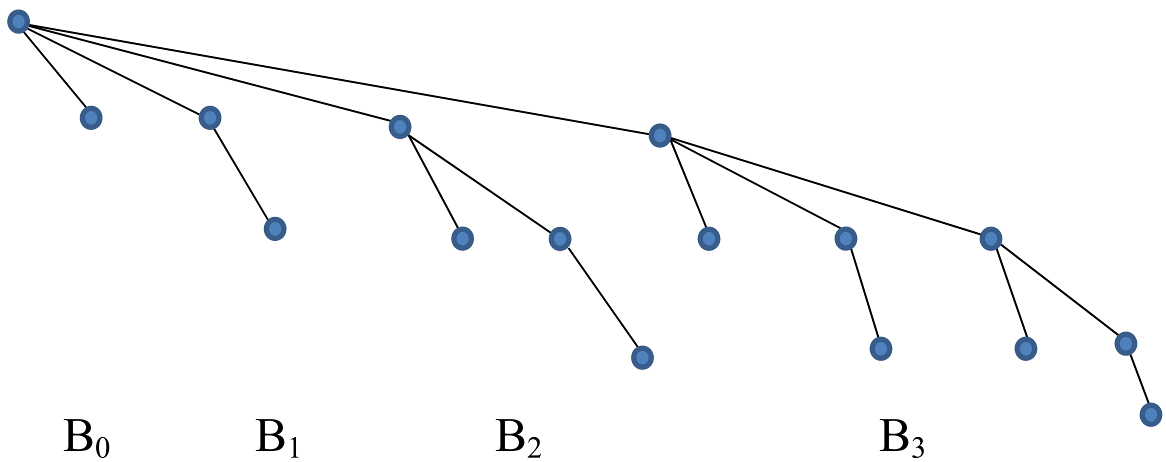
B_k:



Observation: On expanding, B_k can also be formed by making the roots of B_0, B_1, …, B_{k-2}, B_{k-2} be the children of a new root.



Example: A BT B_4 of order 4.



Observations:

Properties of B_4:

(1) The root of B_4 has exactly 4 children.
(2) B_4 has height 4.
(3) There are $2^4 = 16$ nodes in B_4.
(4) The number of nodes at each level of B_4 is given by the following binomial coefficients.

Level	# Nodes	Binomial Coeff.
0	1	C(4,0)
1	4	C(4,1)
2	6	C(4,2)
3	4	C(4,3)
4	1	C(4,4)

Remark: If one counts the number of nodes at level i, $0 \le i \le k$, of a BT of order k, B_k, it is exactly equal to the binomial coefficient $C(k,i)$. Hence, the total number of nodes in B_k is given by $\sum_{i=0}^{k} C(k,i) = \binom{k}{0} + \binom{k}{1} + \ldots + \binom{k}{k} = 2^k$.

Theorems:

(1) The root of B_k has exactly k children.
(2) The height of B_k is k.
(3) The binomial tree B_k has exactly 2^k nodes.
(4) The number of nodes in B_k at level (depth) i is given by the binomial coefficient $C(k,i)$.

Q: Given a set S of n records, $n \ge 1$, can we always store the n objects in a BT of order k?

No, unless $n = 2^k$ for some integer k.

Remedy:

Use a collection of binomial trees with total number of nodes equal to n.

Binomial Queue/Heap:

A binomial queue (BQ) is a collection of ***uniquely specified*** heap-ordered binomial trees $Q = \{B_i \mid i \in I\}$. Hence, $B_j, B_k \in Q$ implies that $i \neq j$.

Q: Using one node per record, how do we represent a set of n records $S = \{x_1, x_2, \ldots, x_n\}$ using a BQ?

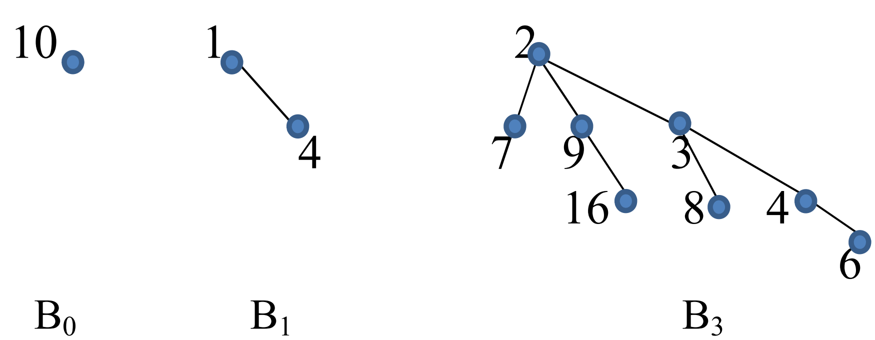
Observations:

1. If $n = 2^k$, then S can be represented by using a single BT of order k.
2. If $n \neq 2^k$, then integer n can always be represented using a binary representation $(b_k, b_{k-1}, \ldots, b_1, b_0)$ with at most k+1 bits. Since a BT of order k contains exactly 2^k nodes and can be used to represent 2^k objects in S, each binary digit $b_i = 1$ in $(b_k \ldots b_1 b_0)_2$ corresponds to a unique BT B_i in Q. We call $(b_k \ldots b_1 b_0)_2$ the binary representation of the binomial queue Q.

Theorem: There are at most $\lceil \lg(n+1) \rceil$ BTs in Q when representing a set of n records.

Example: Given S = {3,8,6,4,2,7,9,16,1,4,10}. Since $n = 11_{10} = 1011_2$, Q = {B_3, B_1, B_0}.

Example of a BQ for S:



BQ Operations:
Let's assume that we already have a function concate(Q1,Q2) that allows us to merge/concate two BQ's Q1 and Q2.

1. **Insert(x,Q):**
   Since x can be considered as a BQ with a BT B_0, ***insert(x,Q) = concate(Q1,Q2),*** where Q1 is the BT containing B_0 with one element x.

2. **Deletemin(Q):**
   Find the min element m among the roots of BTs in Q. If m is the root of a BT B_i, form new BQ Q1 = Q−{ B_i}. Also, remove m from B_i to form a new BQ Q2 by including all remaining BTs in B_i together. Perform ***concate(Q_1,Q2).***

3. **Concate(Q1,Q2):**
   Let's first consider the merging of two BTs B_k of the same order.

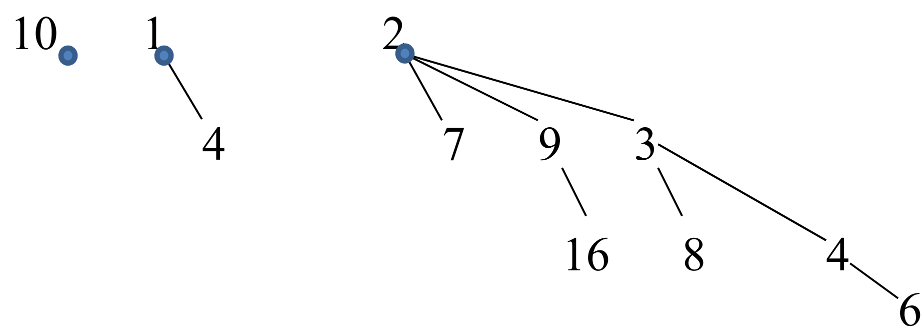
   Simplest approach:
   Compare the roots of the two BTs and then make the root of the BT with smaller root the parent of the other BT. Hence, $T_W(n) = O(1)$.

   Q: How can we extend this method to concate(Q1,Q2)?

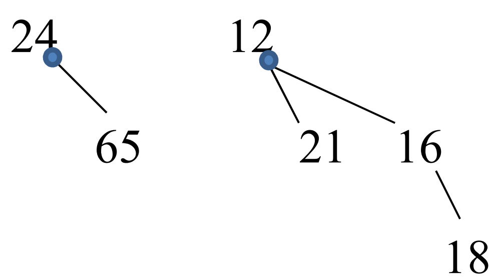
Observe that if $(b_p \ldots b_1 b_0)_2$ and $(c_p \ldots c_1 c_0)_2$ are the binary representation of |Q1| and |Q2|, then concate(Q1,Q2) results in a BQ Q3 such that |Q3| has binary representation $(d_{p+1} d_p \ldots d_1 d_0)_2$, where $(d_{p+1} d_p \ldots d_1 d_0)_2 = (b_p \ldots b_1 b_0)_2 + (c_p \ldots c_1 c_0)_2$.

Example:

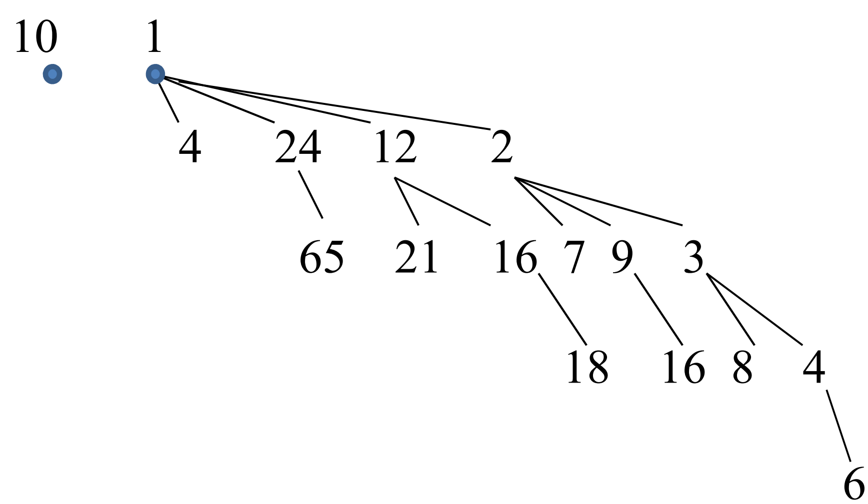
Q1:



Q2:



Concate(Q1,Q2):



Complexity for Binomial Queue:

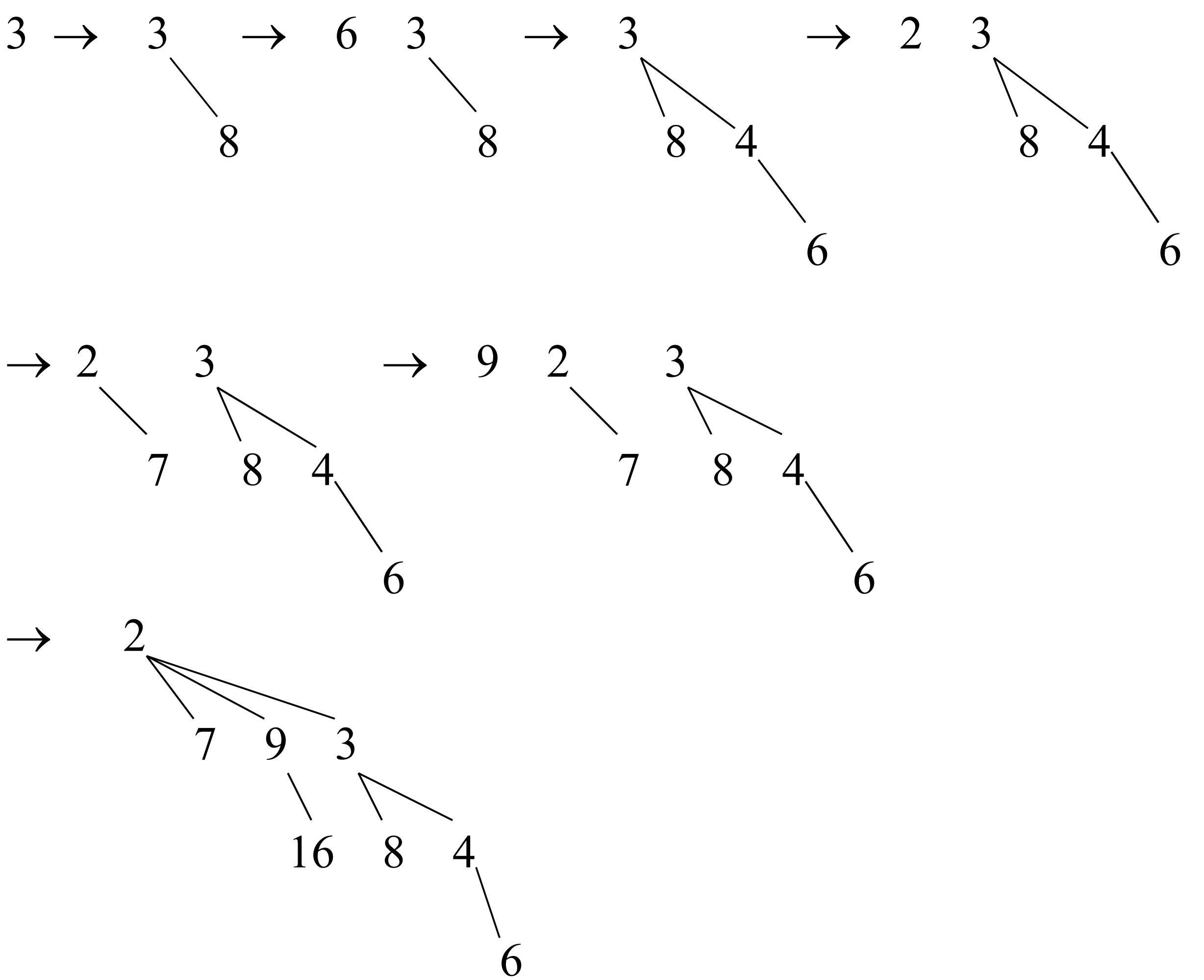
concate, insert, deletemin: $T_w(n) = O(\lg n)$.

findmin: $T_w(n) = O(\lg n)$. (No min pointer; TBA)

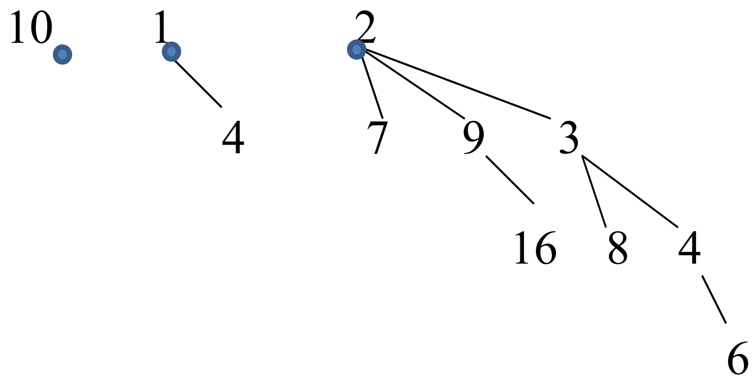
Building a BQ:

Insert the given elements one by one into an initially empty BQ. Merge two BTs B_i of order i to form a BT B_{i+1} of order i +1 if exist. Tie can be broken arbitrarily. Hence, $T_W(n) = O(n\lg n)$.

Example: Building a BQ for S = {3,8,6,4,2,7,9,16,1,4,10}.



Final binomial queue:



Implementation of BQ:

Node Structure:

order	key	item
l_sib	f_child	r_sib

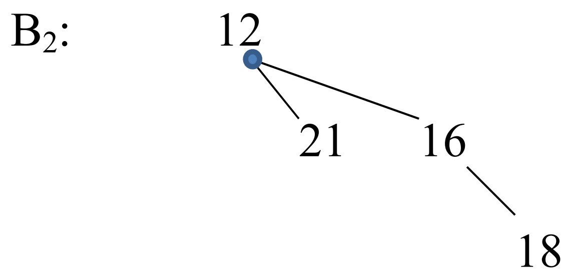
A BT is implemented using the leftmostChild-rightSibling representation with circular doubly linked list.

For any node x in a BT:

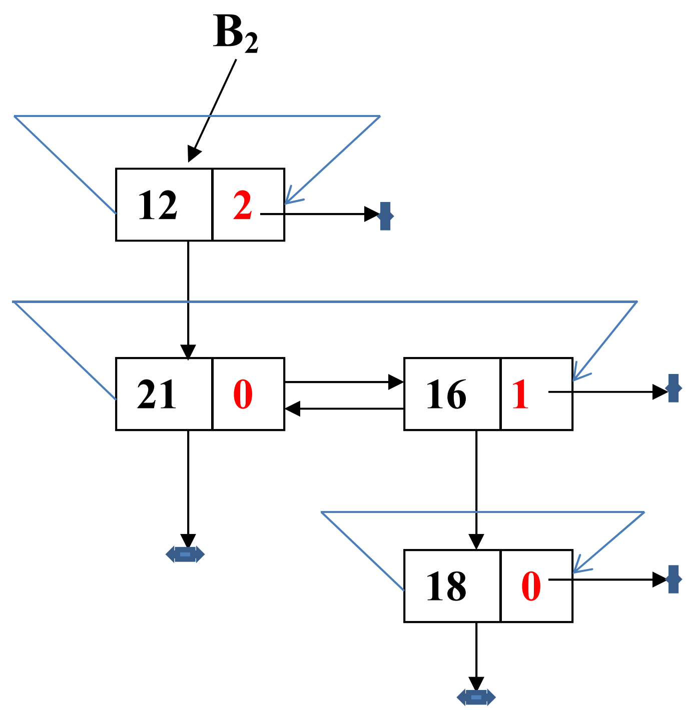
order: Order of BT rooted at x.
f_child: pointer pointing at the lowest order child of x.
r_sib: pointer pointing at the right sibling of x.
l_sib: pointer pointing at the left sibling of x.

Observe that, with this structure, siblings of x are linked together from lower to higher order. A BQ Q is then implemented by linking all binomial trees in Q together in ***increasing order of their orders***.

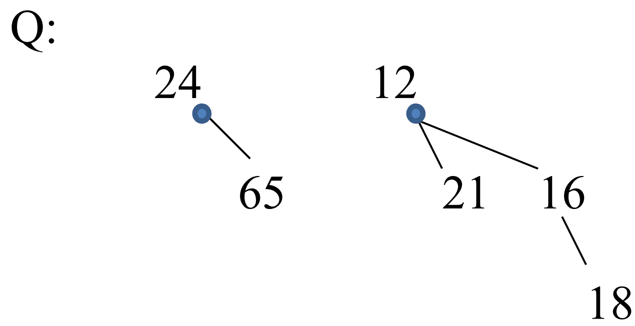
Example of Binomial Tree Implementation:



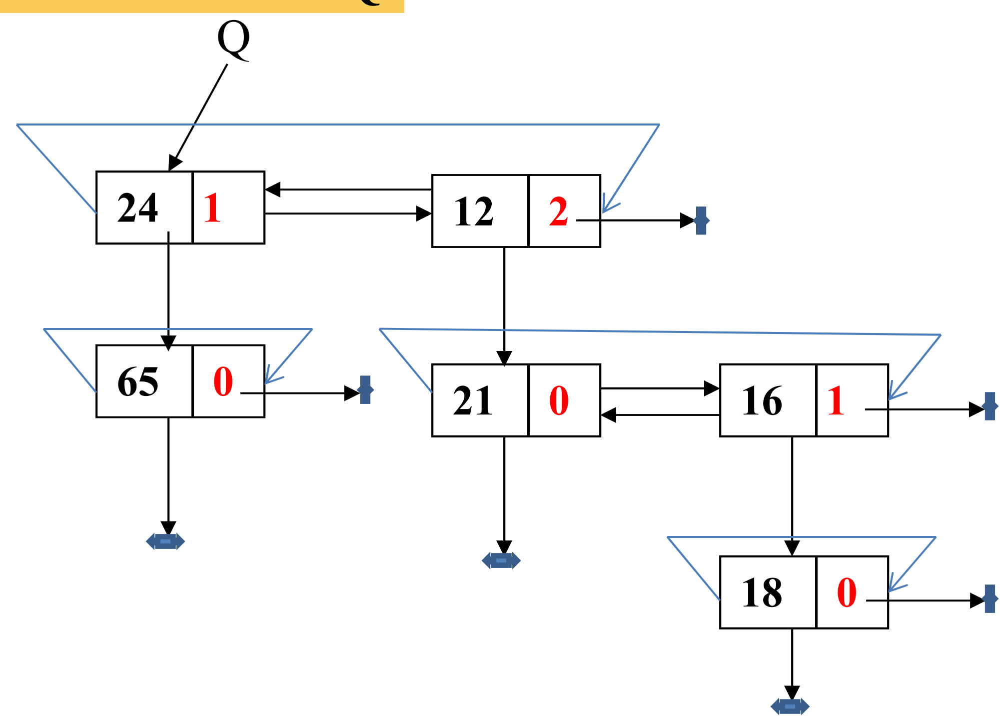
Data Structure for B_2:



Example of Binomial Queue Implementation:



Data Structure for Q:



Warning: The BQ pointer Q is pointing at the root of the BT with the lowest order, not a BT with minimum element.

Node Implementation:

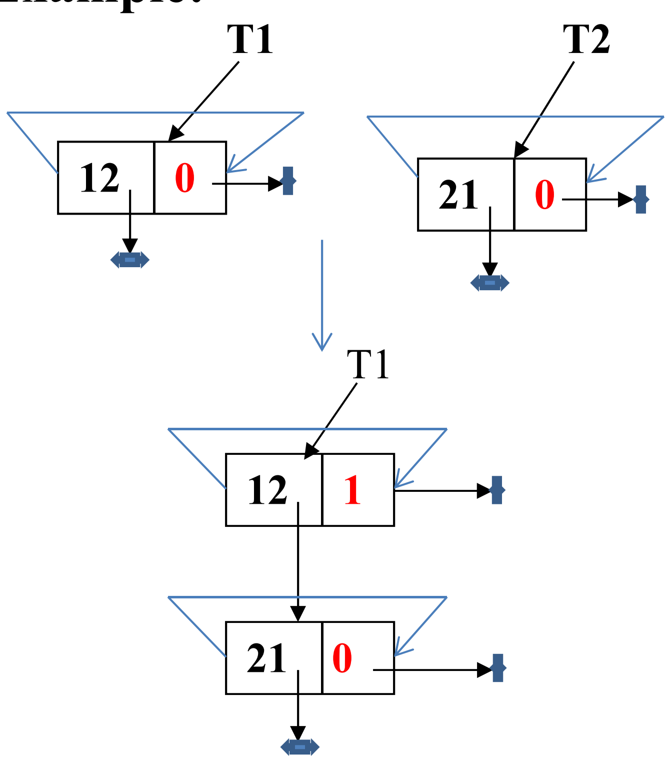
```
struct bqNode
{
   bqItemType:     item;
   int             key;
   int             order;
   bqNode          *f_child, *l_sibling, *r_sibling;
}
```

Consider the merging of two binomial trees T1 and T2 of order k, $k \geq 0$, with $T1 \rightarrow key \leq T2 \rightarrow key$.

Two cases:

(i) If $k = 0$, then $T1 \rightarrow f\_child = T2;$
$T1 \rightarrow order = T1 \rightarrow order + 1;$

Example:



(ii) If $k > 0$, then

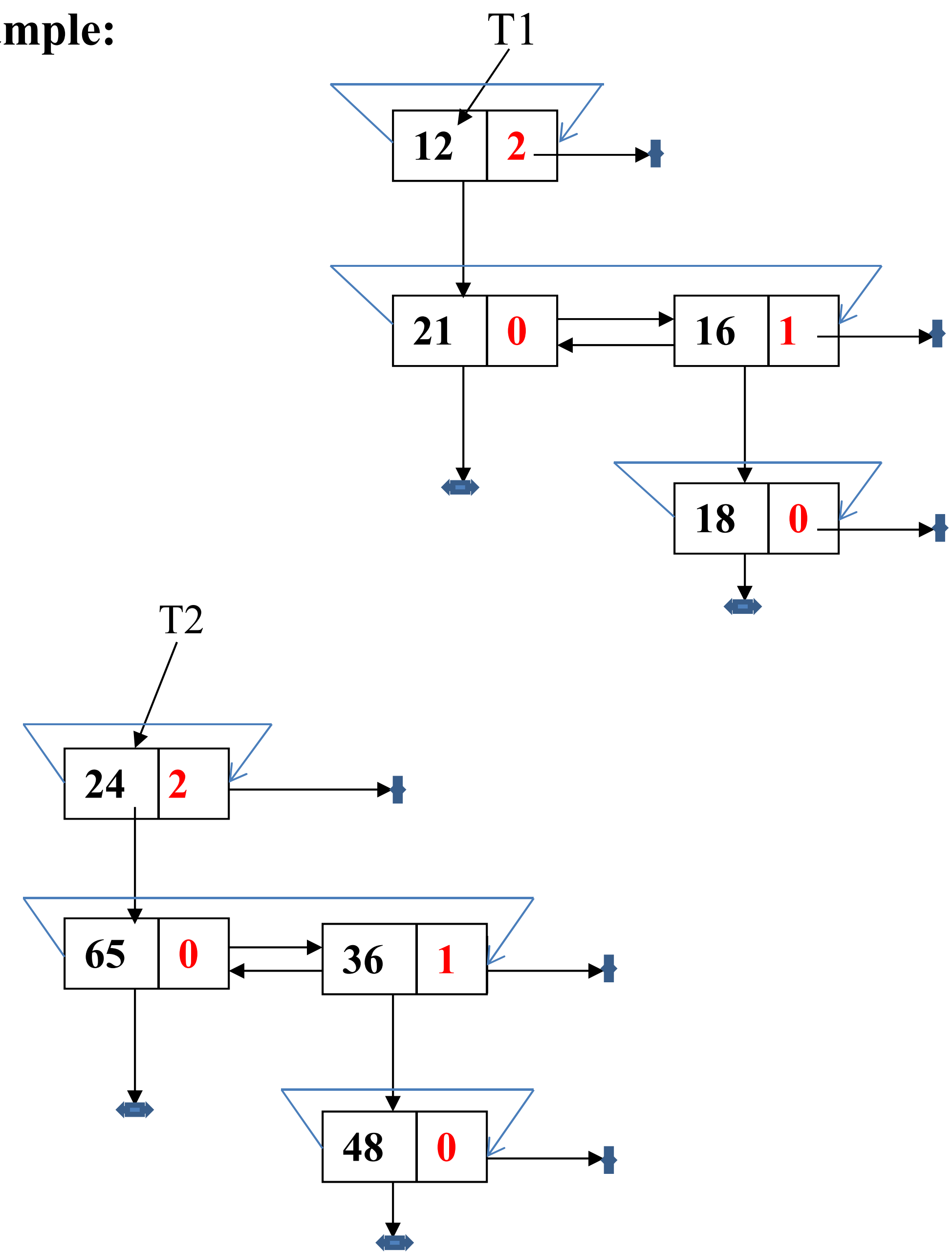
(1) $T2 \rightarrow l\_sibling = T1 \rightarrow f\_child \rightarrow l\_sibling;$

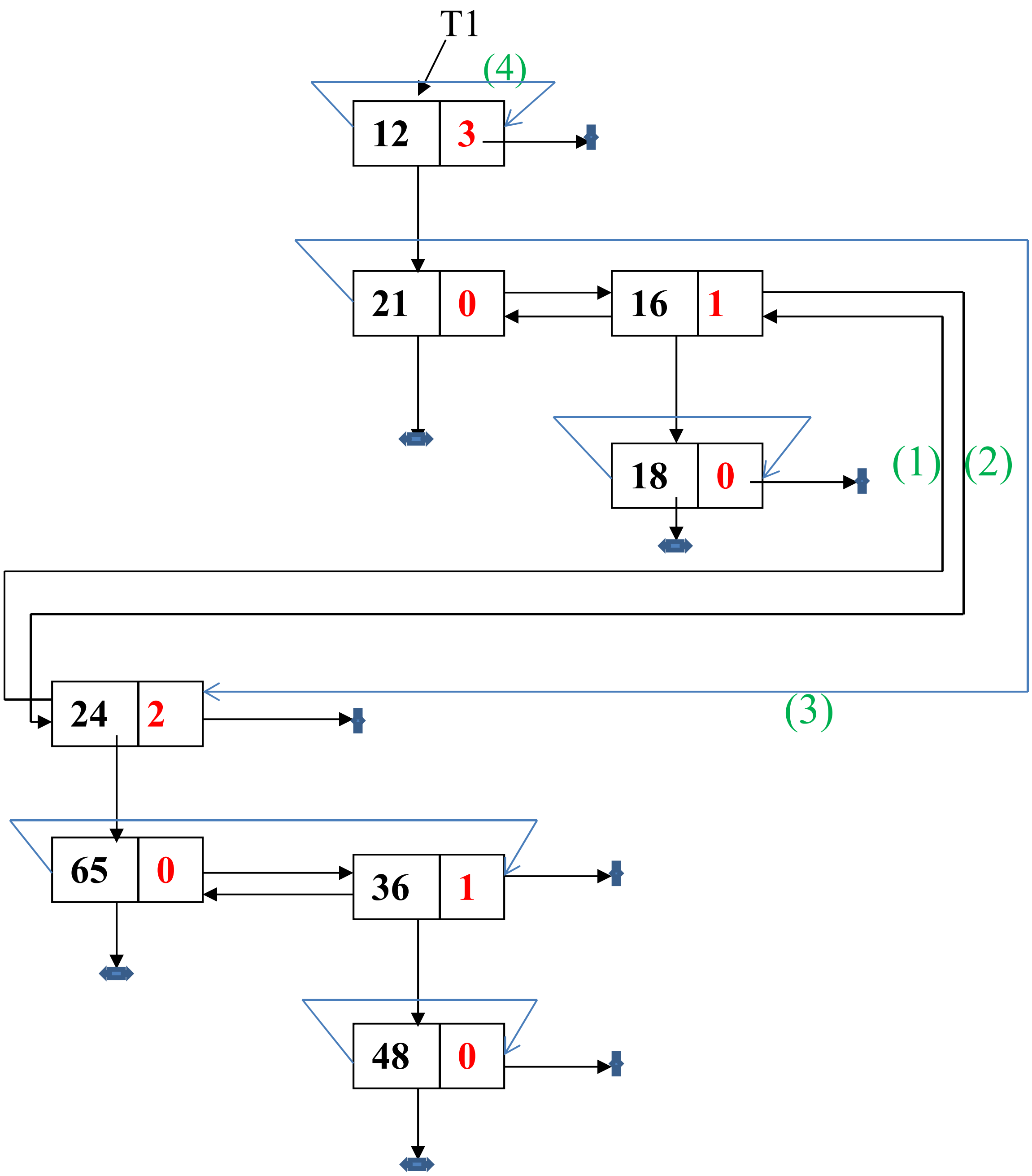
(2) $T2 \rightarrow l\_sibling \rightarrow r\_sibling = T2;$

(3) $T1 \rightarrow f\_child \rightarrow l\_sibling = T2;$

(4) $T1 \rightarrow order = T1 \rightarrow order + 1;$

Example:





HW: Implement a function to merge two binomial trees B_k of order k together and then use it to implement a function concate(Q1,Q2) to merge two BQs.

V. Fibonacci Heap (Fredman & Tarjan):

A Fibonacci Heap (FH) is an extended Binomial Queue (BQ) consisting of a collection of (min) heap-ordered trees.

Data Structure:

Implementation of FH is almost identical to the BQ structure.

(1) Each heap-ordered tree is implemented using a first_child right_sibling implementation similar to a Binomial Tree but in a totally circular fashion.

(2) The children of a node (siblings) are linked together using a circular doubly linked list structure.

(3) All the heap-ordered trees are linked together by their roots using a circular doubly linked list.

(4) Each node has a parent pointer pointing to its parent.

(5) Each node contains the **degree** of the node, which is the number of children of that node.

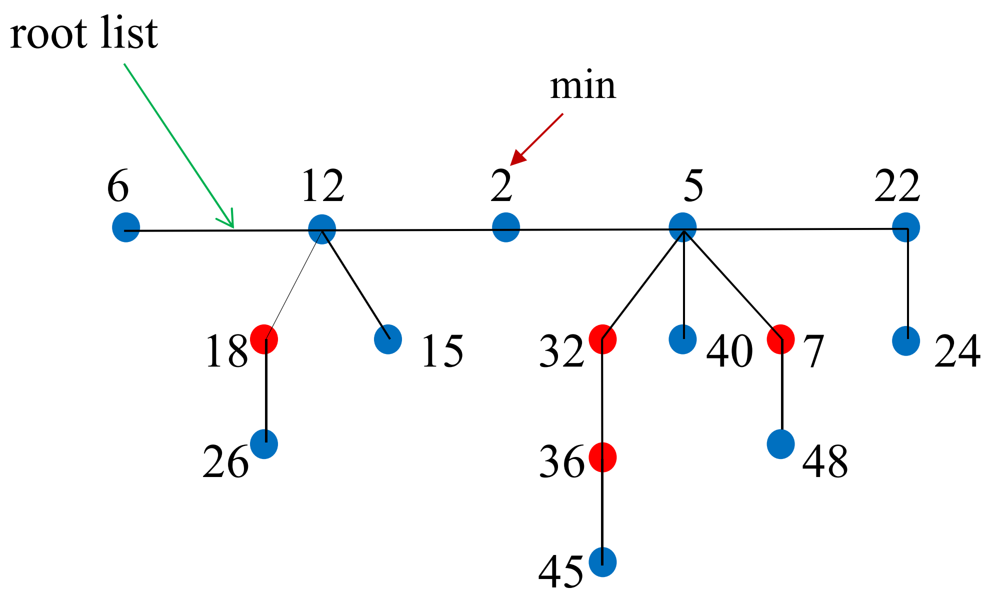
(6) A node is either marked, if a child has been removed/cut from it, or unmarked.

(7) In order to access a FH, we use a pointer pointing at the root of the tree with min key.

Node Structure of FH:

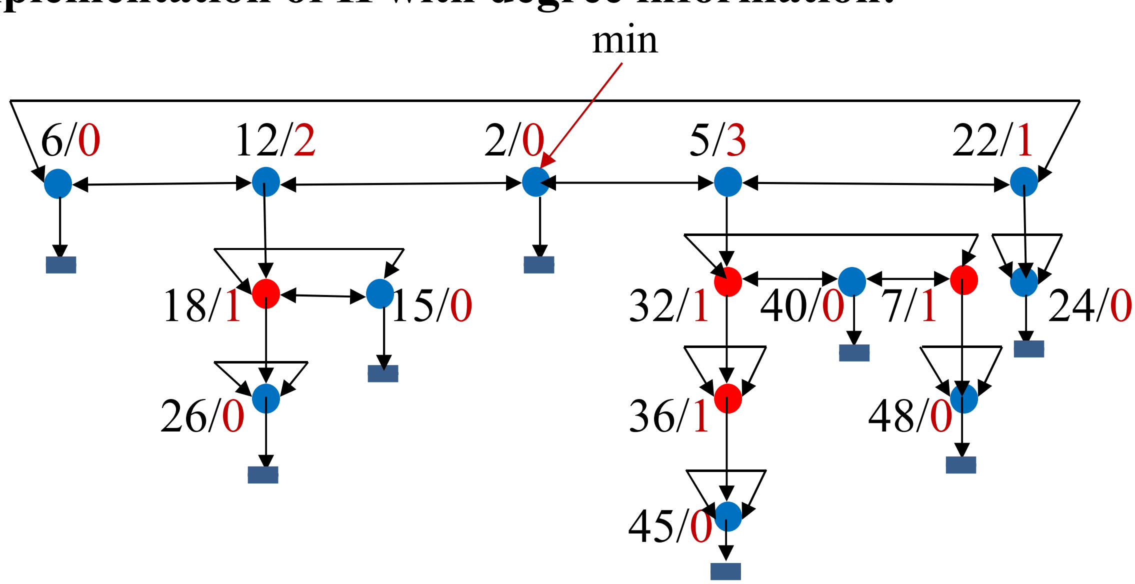
	parent	
degree	item	marked
l-sibling	f-child	r-sibling

Example: A Fibonacci heap H with five heap-ordered trees linked together by their roots.



●: Marked node.
●: Unmarked node.

Implementation of H with degree information:



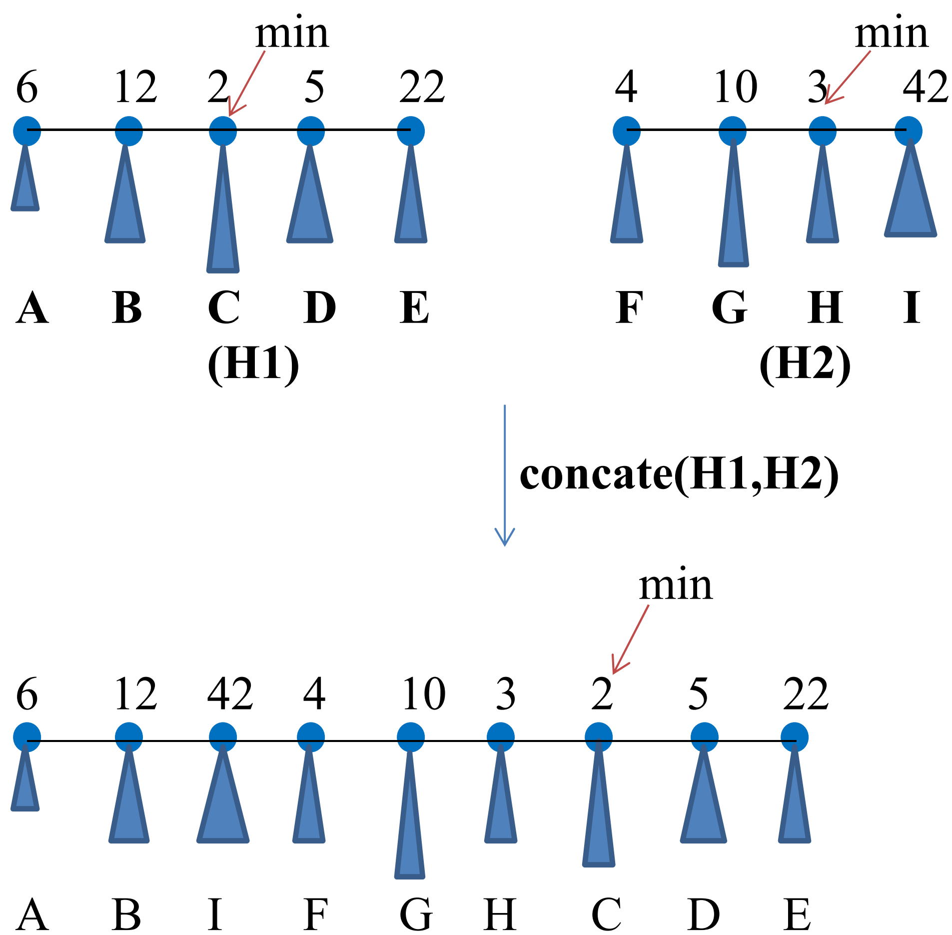
Remark: For simplicity, the parent pointer of each node is omitted.

Fibonacci Heap Operations:

FH operations are all based on "lazy" merging of FHs.

1. **Concate(H1,H2):**
   Let H1 and H2 be two item-disjoint FHs.
   Assume that min(H1) ≤ min(H2).
   Insert the FH H2 to the left of the min node in the root list of H1. $\mathbf{T_w(n) = O(1)}$.

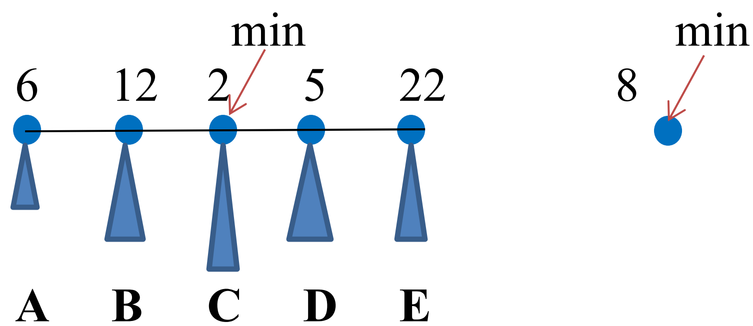
Example: Concate(H1,H2).



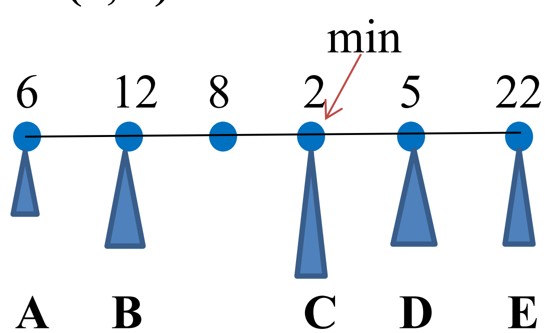
2. **Insert(x,H):**
   Create a single node FH containing x and then merge it with H. Update min pointer if necessary.
   $T_w(n) = O(1)$.

Example: Insert(x,H).

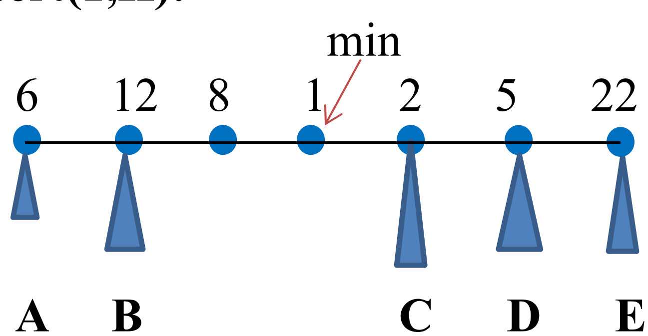
A FH H:



Insert(8,H):



Insert(1,H):



3. **DeleteMin(H):**

Step 1: Delete the min node of H and concatenate the children of the min node into the root list by replacing the min root with the children list.

Step 2: Update min pointer if necessary.

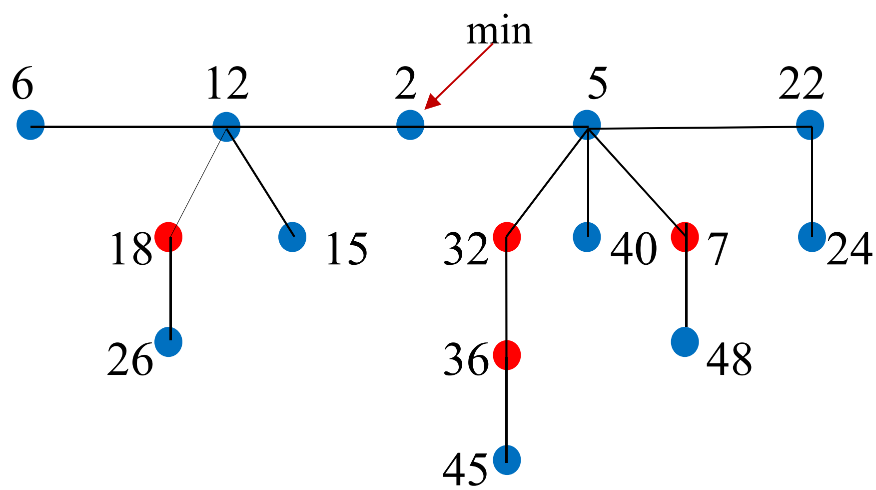
Step 3: ***Consolidate the heap-ordered trees in the FH so that no two trees will have the same degree.***

在deleteMin中多一个方法，重新整理

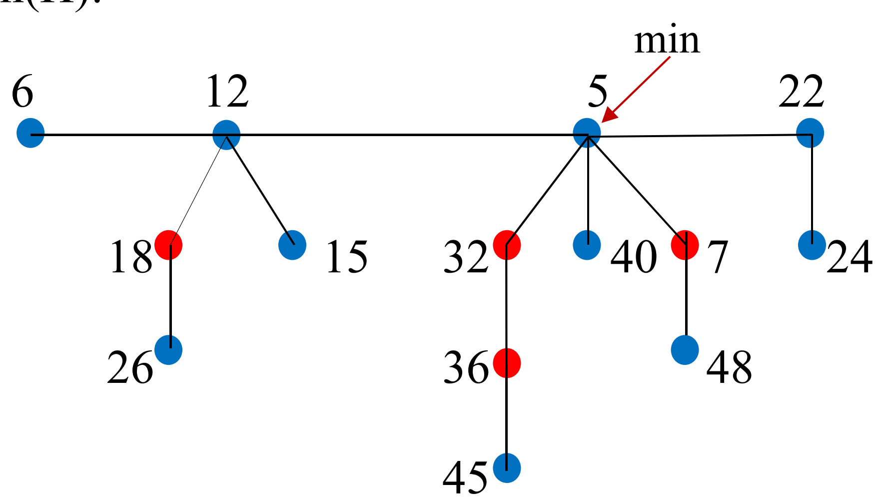
$T_w(n) = O(n)$.

Example: DeleteMin(H).

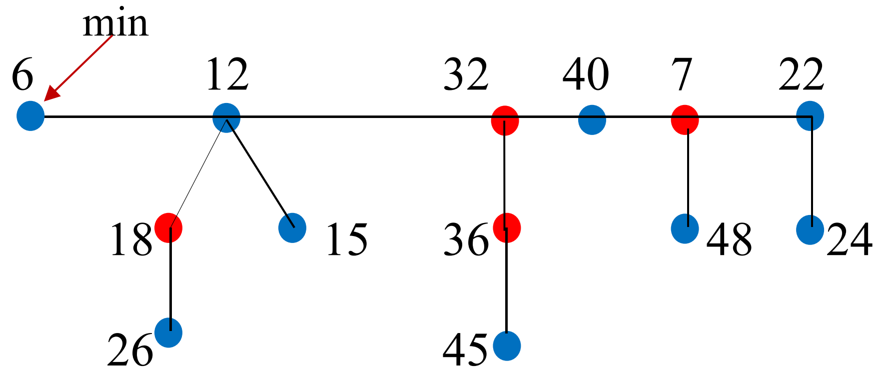
H:



DeleteMin(H):



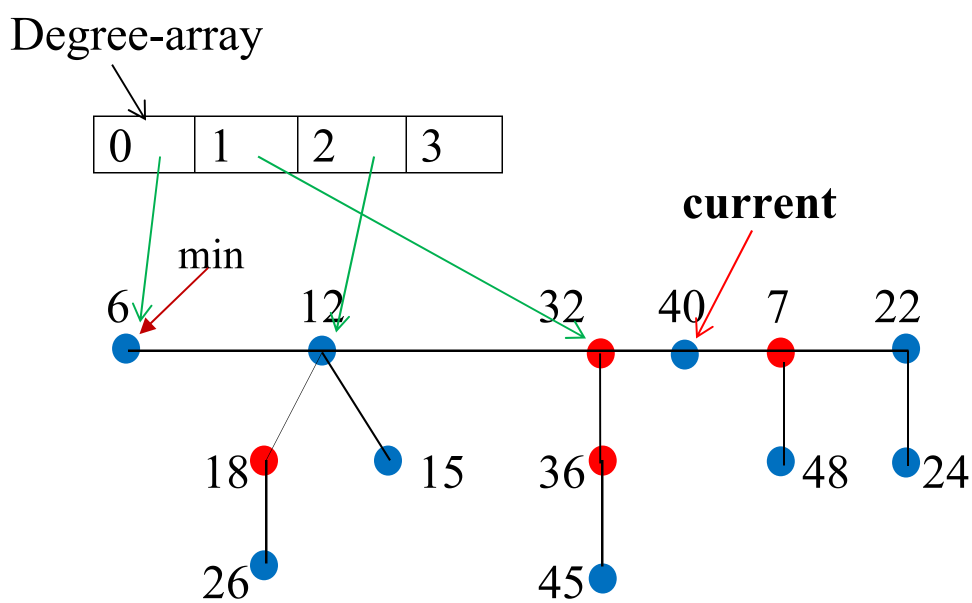
DeleteMin(H):



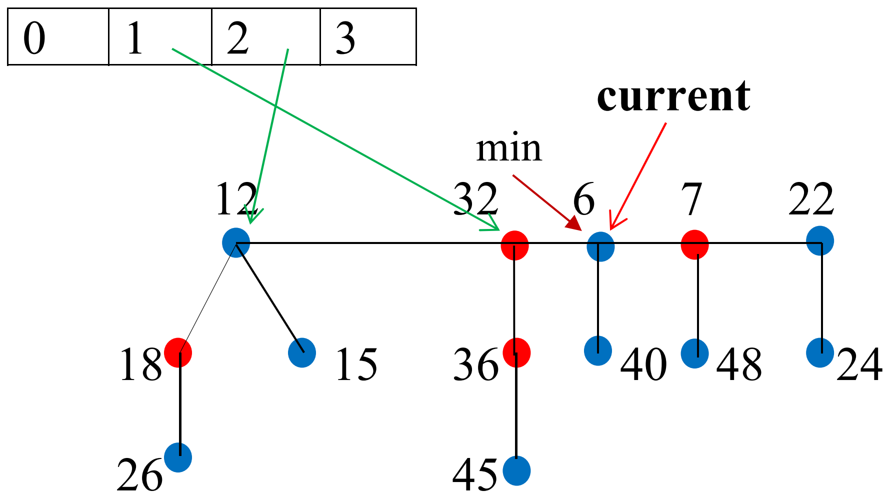
Consolidation of Heap-Ordered Trees in FH using a Degree-Array:

Starting at min node 6, consolidating trees with same degree:

(i)

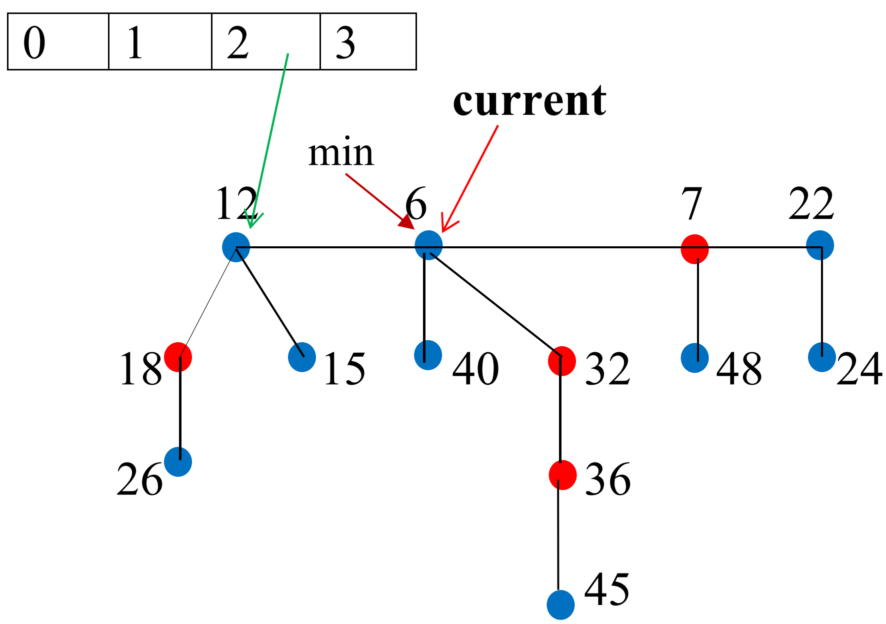


(ii)



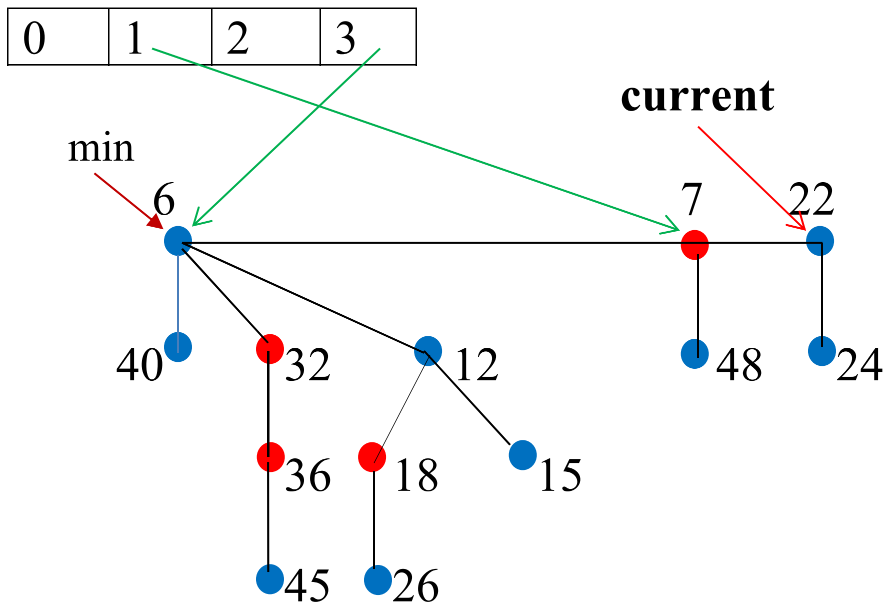
0
1
2
3
current
min
12
32
6
7
22
18
15
36
40
48
24
26
45

(iii)

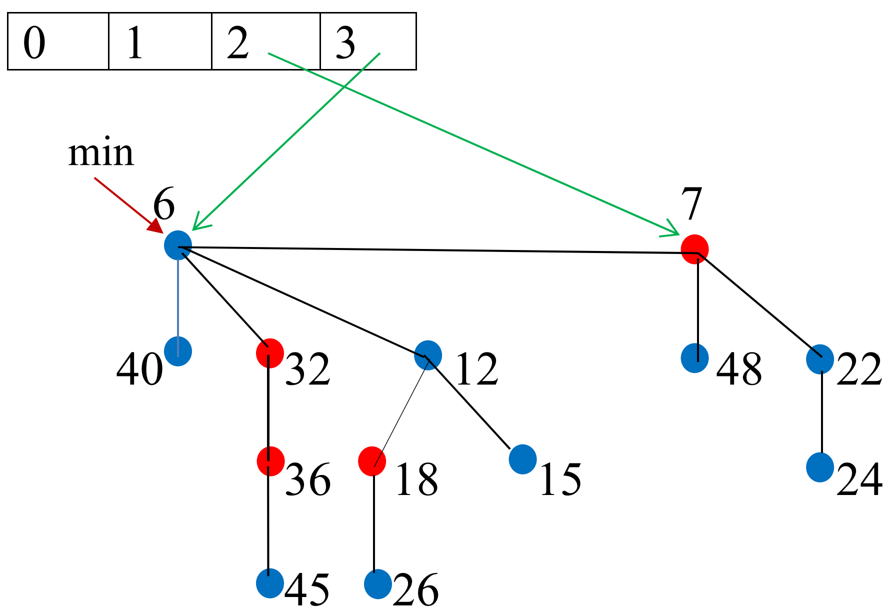


0
1
2
3
current
min
12
6
7
22
18
15
40
32
48
24
26
36
45

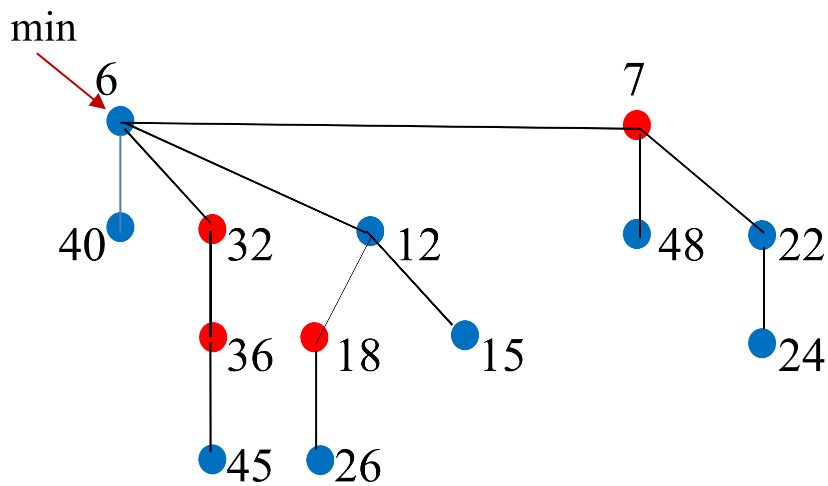
(iv)


0
1
2
3
current
min
6
7
22
40
32
12
48
24
36
18
15
45
26

(v)


0
1
2
3
min
6
7
40
32
12
48
22
36
18
15
24
45
26

Final FH after consolidating trees:



4. FindMin(H):

Follow min pointer to find min node.

$T_w(n) = O(1)$.

Remark: If only concate, insert, deleteMin and findMin operations can be performed on a FH, all the heap ordered trees in the FH are Binomial trees. (Why?)

1.不违法大小顺序，直接该值
2.如果违反顺序，把改动的值放进root list，并且marked parent；如果已经marked的parent，那就把parent放进root list，并且取消mark
在这里是把值变小不考虑变大的情况！

5. *DecreaseKey(q, k, H)*: 改变Key的值从q变成k

Step 1: Decrease the key of node x pointed at by q to k in H.

Step 2: Compare the new key of x with its parent p(x) if exists.

新的值所在的位置与父母比较两种

Case 2.1: Heap ordered tree property is not violated. Update min pointer if necessary. 直接变

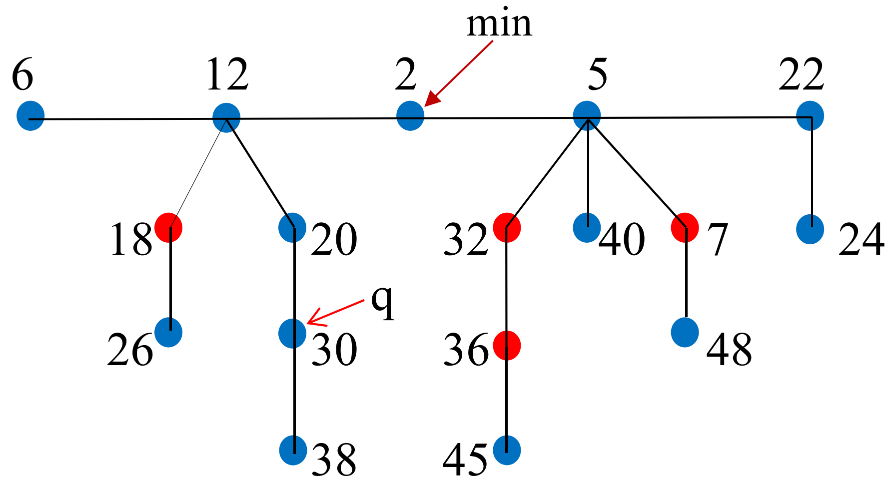
Case 2.2: Heap ordered tree property is violated. Cut the x-tree rooted at x from its parent p(x).

(i) The parent of x, p(x), is unmarked:
Mark p(x).
Add the x-tree to the root list.
Update min pointer if necessary.

(ii) The parent of x, p(x), is marked:
Add the x-tree to the root list.
Update min pointer if necessary.
If the parent of x, p(x), is the root of a tree, then unmark it. Else, cut the p(x)-tree rooted at p(x) from its parent p(p(x)), unmark p(x), and then add the p(x)-tree to the root list.
Repeat this process until an unmarked node or the root is encountered. If the root is encountered and is marked, unmark it.

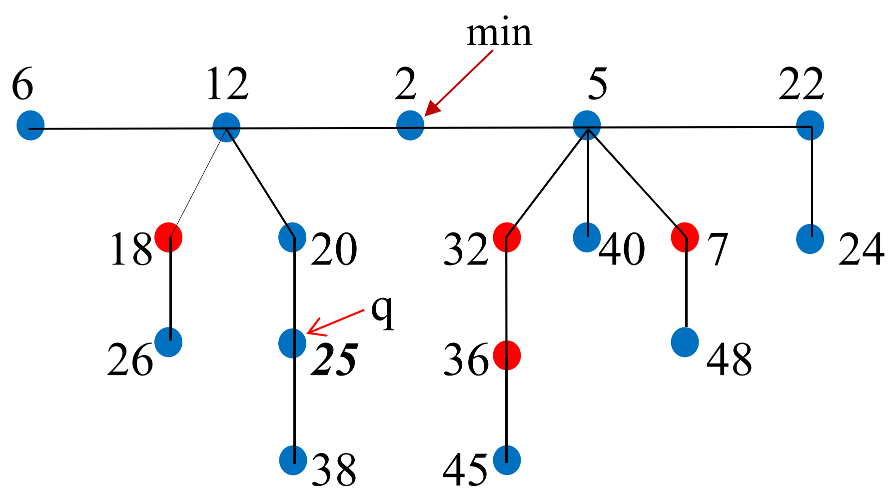
$T_w(n) = O(\lg n)$

Example: DecreaseKey(q, k, H).
A Fibonacci heap H:



DecreaseKey(q, 25, H)

DecreaseKey(q, 25, H):
Case 1: No violation of heap ordered tree property.
Change node 30 to 25.



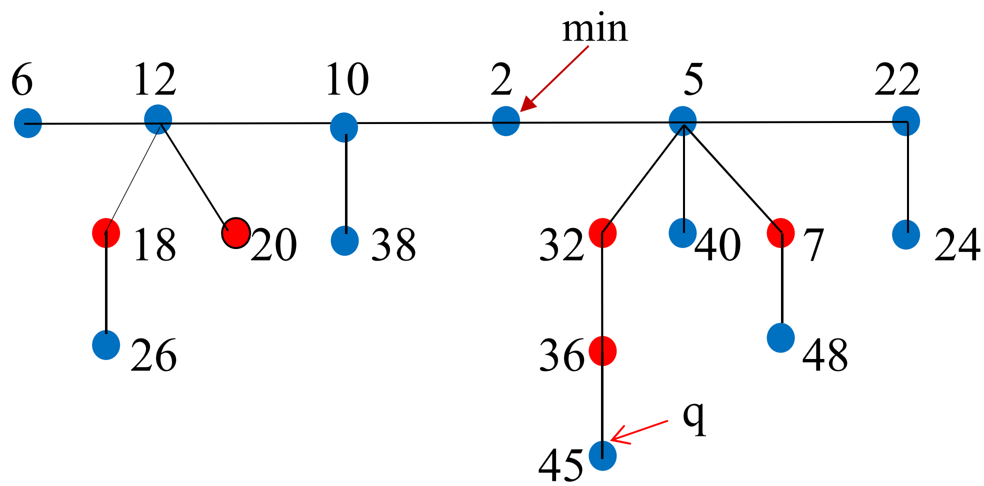
DecreaseKey(q, 10, H)

DecreaseKey(q, 10, H):

Case 2.2.(i): Violation of heap ordered tree property and p(x) is unmarked.

Case1:改变当前的值，如果没有被标记，新的subtree变成新的root list，并且标记原来parent

Change node 25 to 10.
Cut the 10-tree rooted at 10 from its parent 20.
Mark node 20.
Add the 10-tree to the root list.



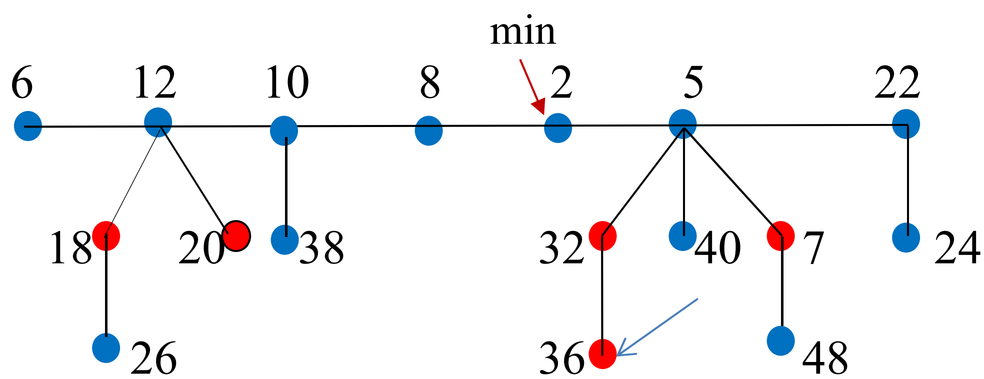
DecreaseKey(q, 8, H)

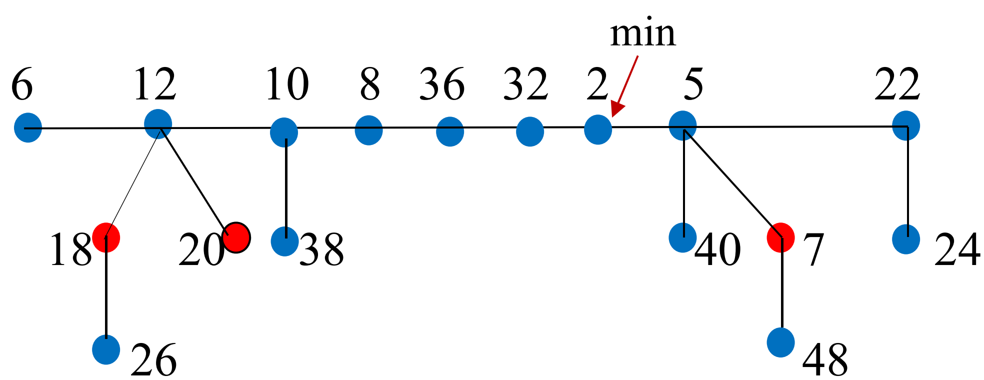
DecreaseKey(q, 8, H):

Case 2(ii): Violation of heap ordered tree property and p(x) is marked.

同上，如果遇到已经被标记的parent，全部递归到root list，root list的Node永远不被标记

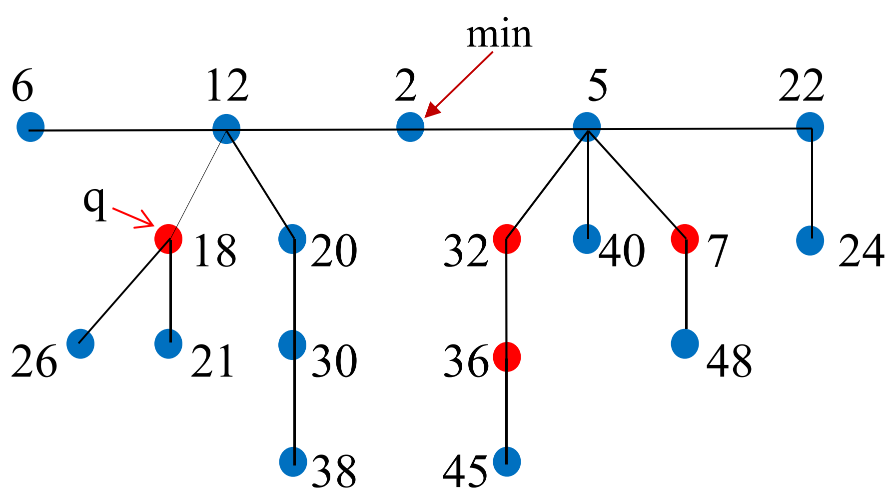
Change node 45 to 8.
Cut the 8-tree rooted at 8 from its parent 36.
Add the 8-tree to the root list.
Since node 36 is marked, unmark 36 and cut the 36-tree rooted at 36 from its parent 32.
Add the 36-tree to the root list.
Since node 32 is marked, unmark 32 and cut the 32-tree rooted at 32 from its parent 5.
Add the 32-tree to the root list.



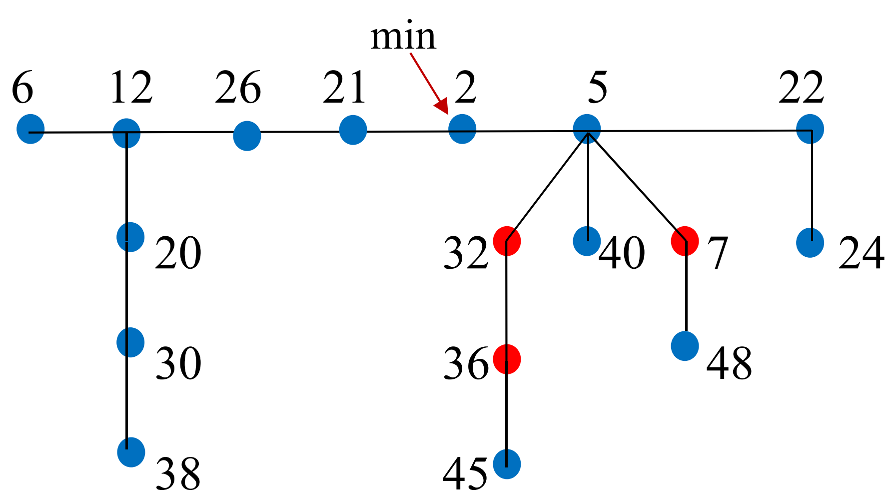


6. ***Delete(q,H)***:
Decrease the node pointed at by q to -∞ in H.
DeleteMin.
$T_w(n) = O(n)$.

Example: Delete(p,H).



Delete(p,H)



Summary:

1. FH is an extension of BQ consisting of a collection of heap ordered trees.
2. FH is a concatenate queue supporting concate, insert, findMin, and deleteMin operations efficiently.
3. If FH is used as a concatenate queue, then the collection of heap ordered trees are all Binomial trees.
4. FH also supports decreaseKey and delete operations efficiently.
5. FH is used to improve the performance of many network optimization problems.

Amortized Complexity $T_{AC}(n)$ of Fibonacci Heap:

	concate	insert	findMin	deleteMin	decreaseKey	delete
$T_{AC}(n)$	O(1)	O(1)	O(1)	O(lgn)	O(1)	O(lgn)
Tw(n)	O(1)	O(1)	O(1)	O(n)	O(lgn)	O(n)

Reference:

Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms, Michael L. Fredman & Robert E. Tarjan, JACM, Vol. 34, No. 3, 1987, 596-615.