§Amortized Analysis

- An example of using potential function
  a sequence of operations: OP_1, OP_2, … OP_m
  OP_i : several pops (from the stack) and
  one push (into the stack)
  t_i : time spent by OP_i
  the average time per operation:

$$t_{ave} = \frac{1}{m}\sum_{i=1}^{m} t_i$$

e.g. p: pop , u: push

i	1	2	3	4	5	6	7	8
OP_i	1u	1u	2p 1u	1u	1u	1u	2p 1u	1p 1u
t_i	1	1	3	1	1	1	3	2

$$\begin{aligned} t_{ave} &= (1+1+3+1+1+1+3+2)/8 \\ &= \frac{13}{8} \\ &= 1.625 \end{aligned}$$

e.g. p: pop , u: push

i	1	2	3	4	5	6	7	8
OP_i	1u	1p 1u	1u	1u	1u	1u	5p 1u	1u
t_i	1	2	1	1	1	1	6	1

$$t_{ave} = (1+2+1+1+1+1+6+1)/8 = \frac{14}{8} = 1.75$$

● amortized time and potential function

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

a_i : amortized time of OP_i

Φ_i: potential function of the stack after OP_i

$\Phi_i - \Phi_{i-1}$: change of the potential

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} t_i + \sum_{i=1}^{m} (\Phi_i - \Phi_{i-1})$$

$$= \sum_{i=1}^{m} t_i + \Phi_m - \Phi_0$$

If $\Phi_m - \Phi_0 \geq 0$, then $\sum_{i=1}^{m} a_i$ represents an upper bound of $\sum_{i=1}^{m} t_i$

define: Φ_i : # of elements in the stack.

$$\Rightarrow \Phi_m - \Phi_0 \geq 0$$

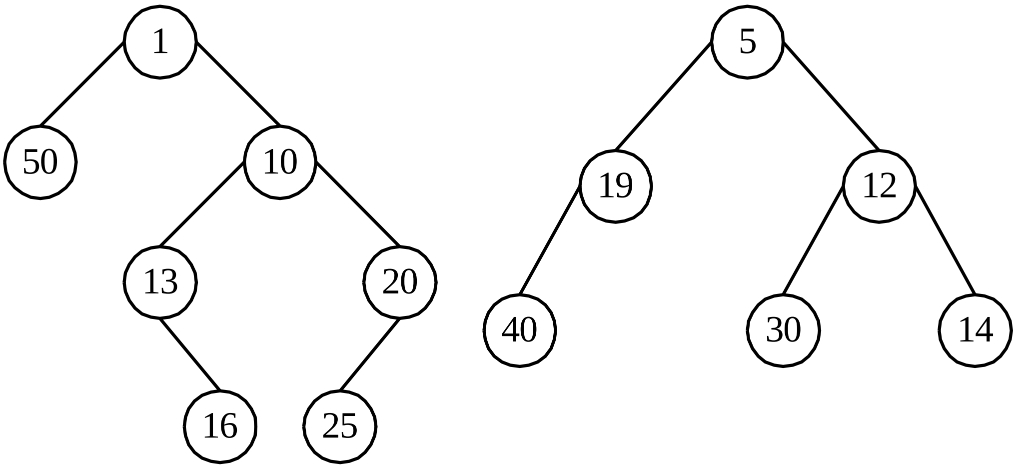
Suppose that before we execute OP_i, there are k elements in the stack and OP_i consists of n pops and 1 push . Thus,

$$\begin{aligned}
&\Phi_{i-1} = k \\
&\Phi_i = k - n + 1 \\
&t_i = n + 1 \\
&a_i = t_i + \Phi_i - \Phi_{i-1} \\
&\quad = n + 1 + (k - n + 1) - k \\
&\quad = 2 \\
&(\sum_{i=1}^{m} a_i) / m = 2
\end{aligned}$$

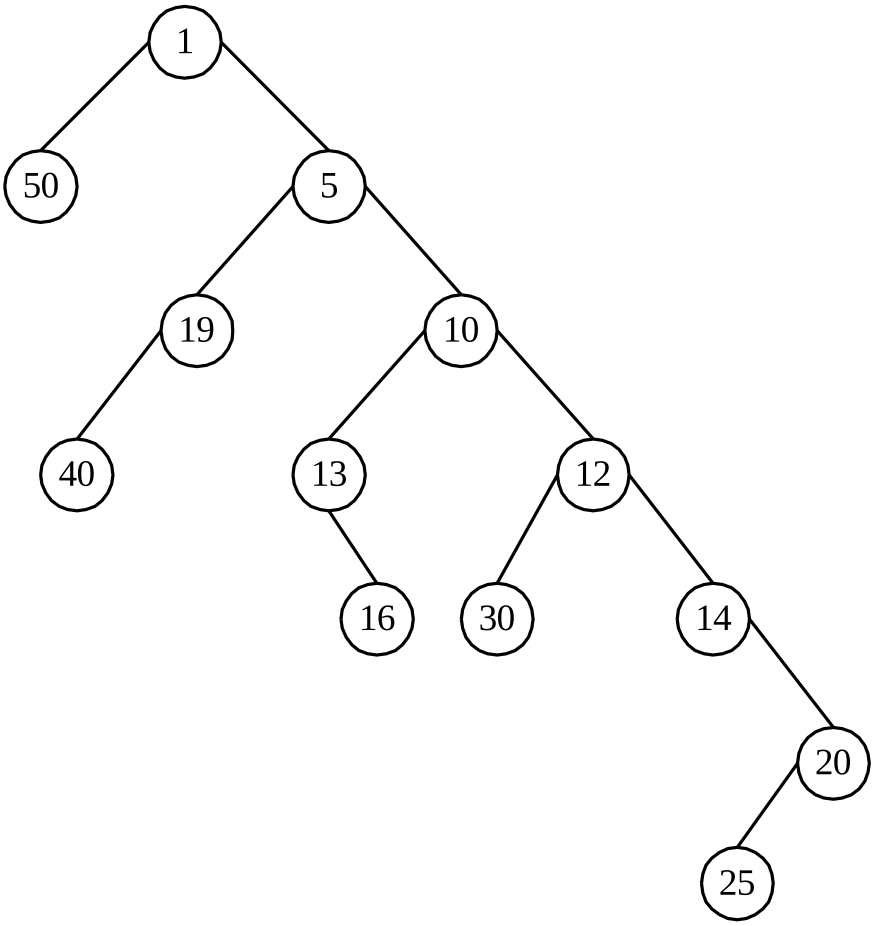
$\Rightarrow t_{ave} \leq 2$

By observation, at most m pops and m pushes are executed in m operation, hence $t_{ave} \leq 2$.

- An amortized analysis of skew heaps
  two skew heaps:

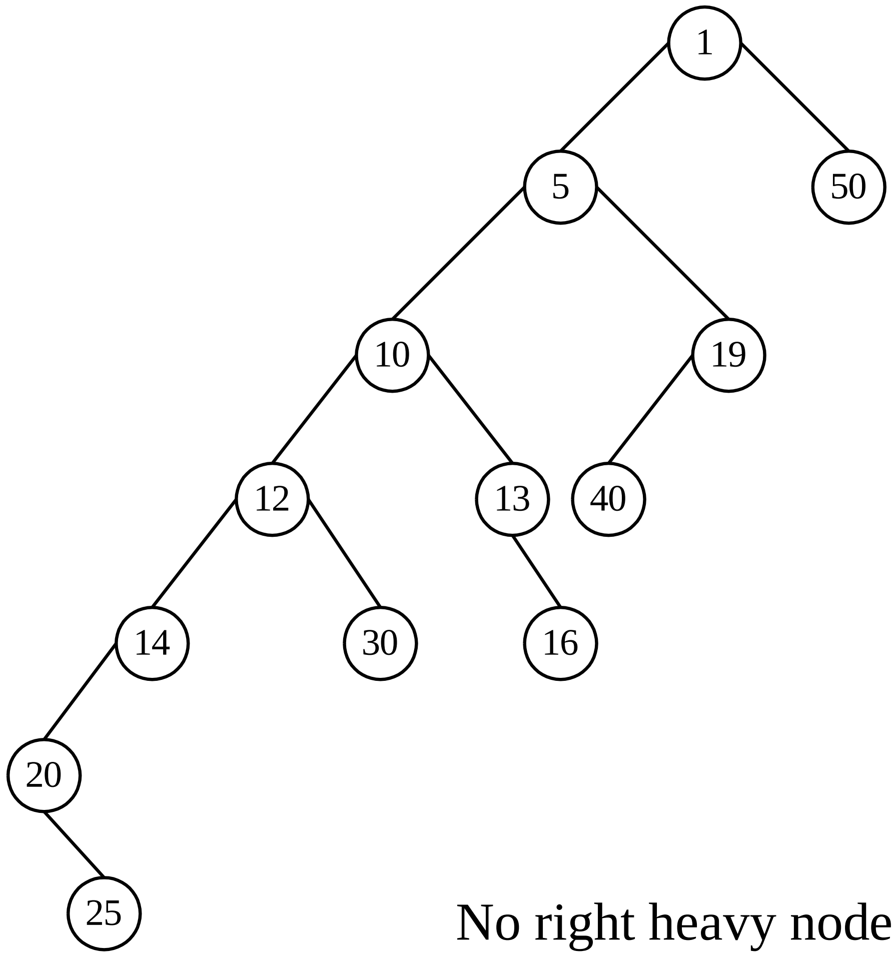


⇓ merge of the right paths



5 right heavy nodes.

⇓ swapping of children along the path formed by the merge.



No right heavy node.

meld: merge + swapping

operations on a skew heap:

1. find-min(h): find the min of a skew heap h.
2. insert(x, h): insert x into a skew heap h.
3. delete-min(h): delete the min from a skew heap h.
4. meld(h_1, h_2): meld two skew heaps h_1 and h_2.

The first three operations can be implemented by melding.

- potential function

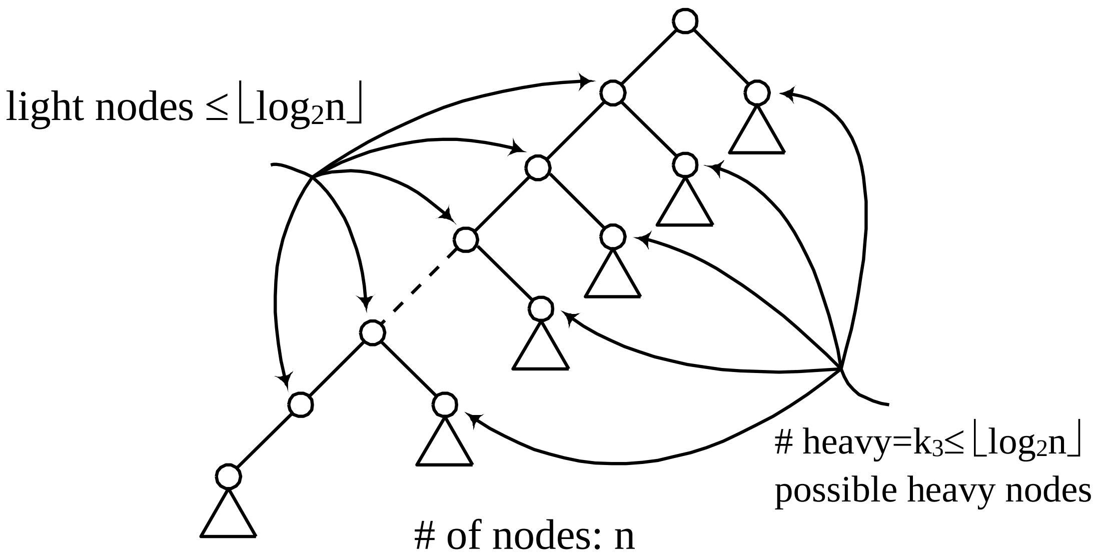
  wt(x): # of descendants of node x, including x.

  heavy node x: wt(x) > wt(p(x))/2, where p(x) is the parent node of x.

  light node: not a heavy node

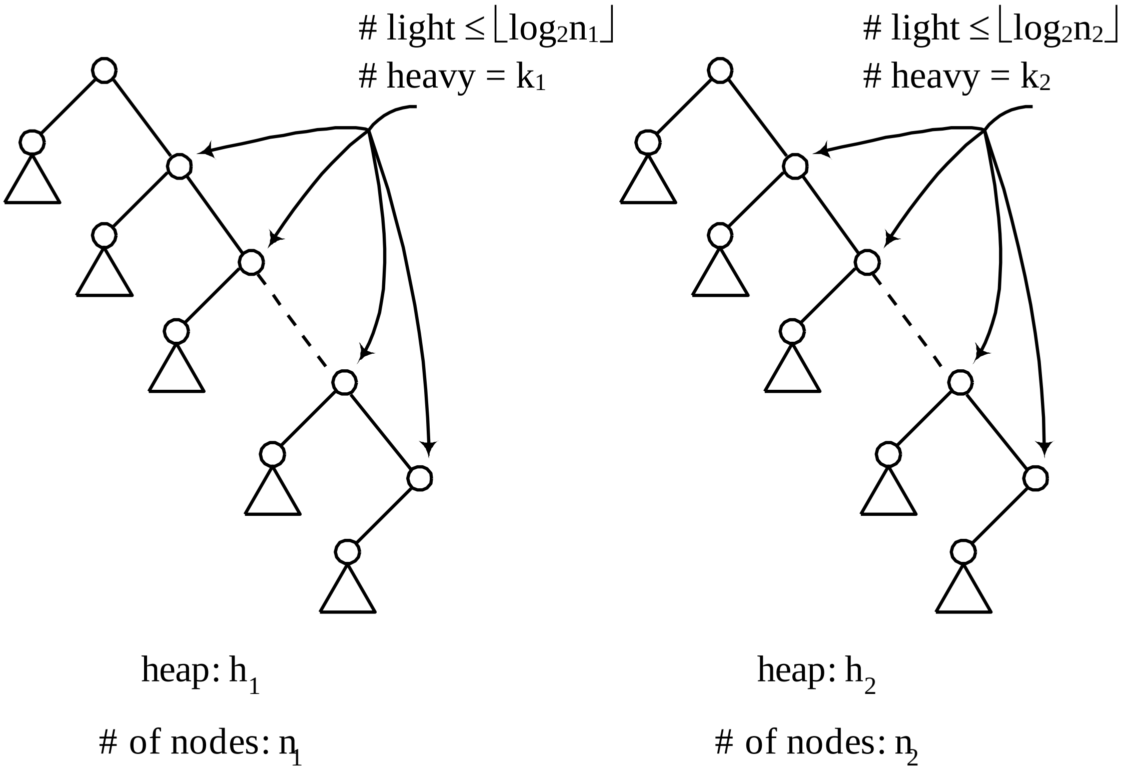
  potential function Φ_i: # of right heavy nodes of the skew heap.

- Any path in an n-node tree contains at most $\lfloor \log_2 n \rfloor$ light nodes.



- The number of right heavy nodes attached to the left path is at most $\lfloor \log_2 n \rfloor$.

- amortized time



$a_i = t_i + \Phi_i - \Phi_{i-1}$

t_i : time spent by OP_i

$t_i \le 2+\lfloor \log_2 n_1 \rfloor + k_1 + \lfloor \log_2 n_2 \rfloor + k_2$

("2" counts the roots of h_1 and h_2)

$\le 2+2\lfloor \log_2 n \rfloor + k_1 + k_2$

where $n=n_1+n_2$

$\Phi_i - \Phi_{i-1} = k_3-(k_1+k_2) \le \lfloor \log_2 n \rfloor - k_1 - k_2$

$a_i = t_i + \Phi_i - \Phi_{i-1}$

$\le 2+2\lfloor \log_2 n \rfloor + k_1 + k_2 + \lfloor \log_2 n \rfloor - k_1 - k_2$

$= 2+3\lfloor \log_2 n \rfloor$

$\Rightarrow a_i = O(\log_2 n)$

● Amortized analysis of AVL-trees

height balance of node v:

hb(v)=height of right subtree - height of left subtree

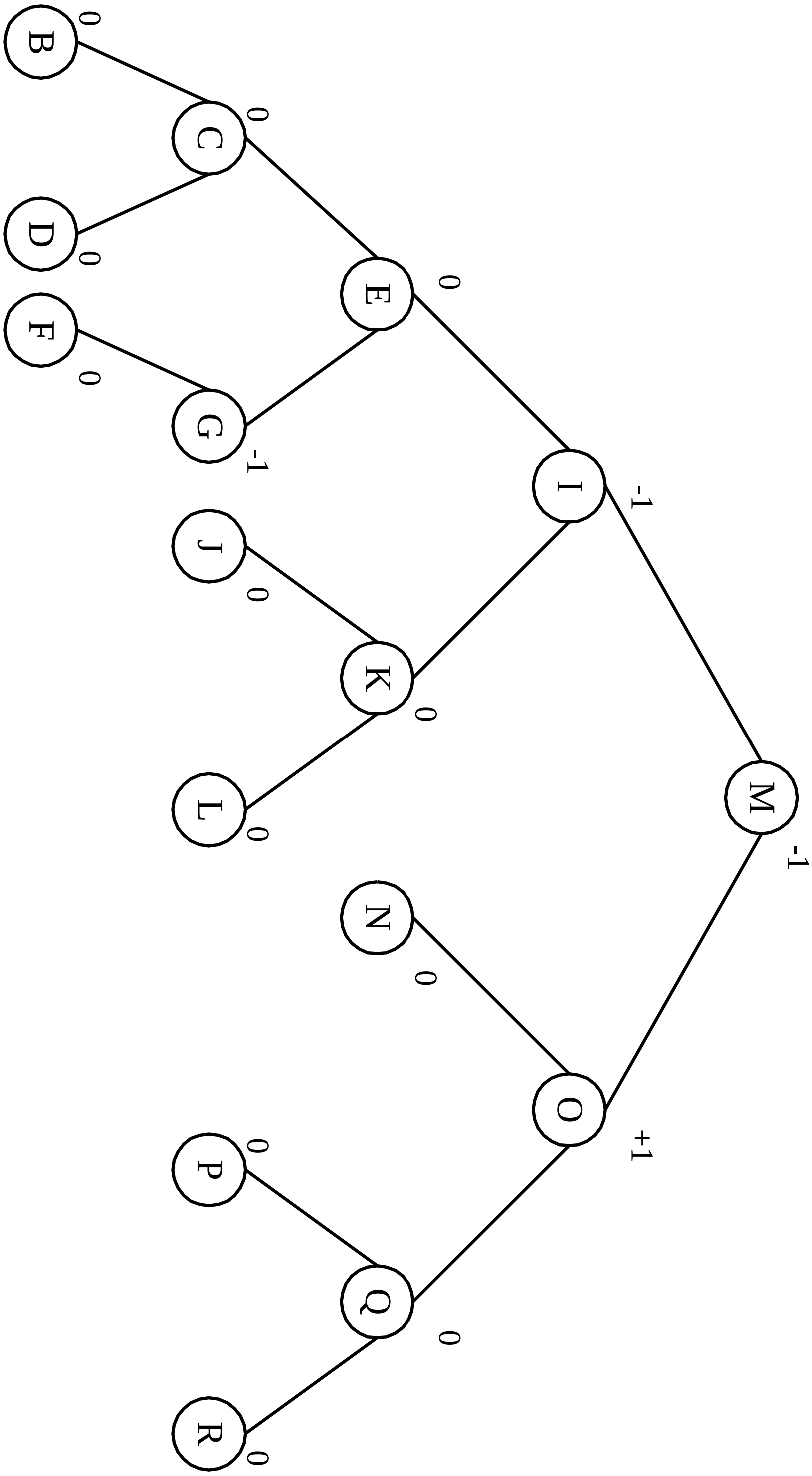
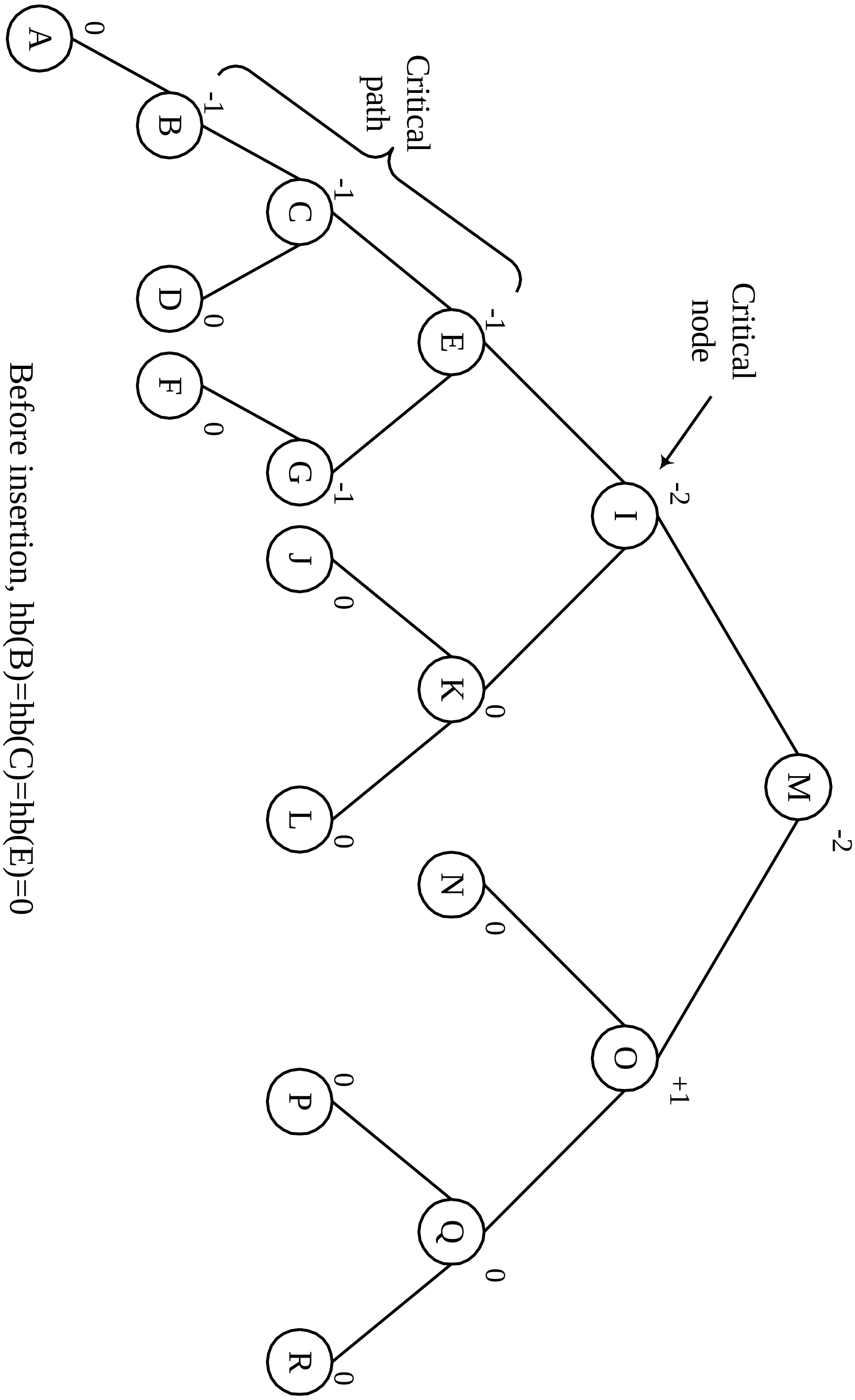


Fig. An AVL-Tree with Height Balance Labeled add a new node A:



Before insertion, hb(B)=hb(C)=hb(E)=0
hb(I)≠ 0…the first nonzero from leaves.

Fig. The New Tree with A Added

Consider a sequence of m insertions on an empty AVL-tree.

T_0: an empty AVL-tree.

T_i: the tree after the ith insertion.

L_i: the length of the critical path involved in the ith insertion.

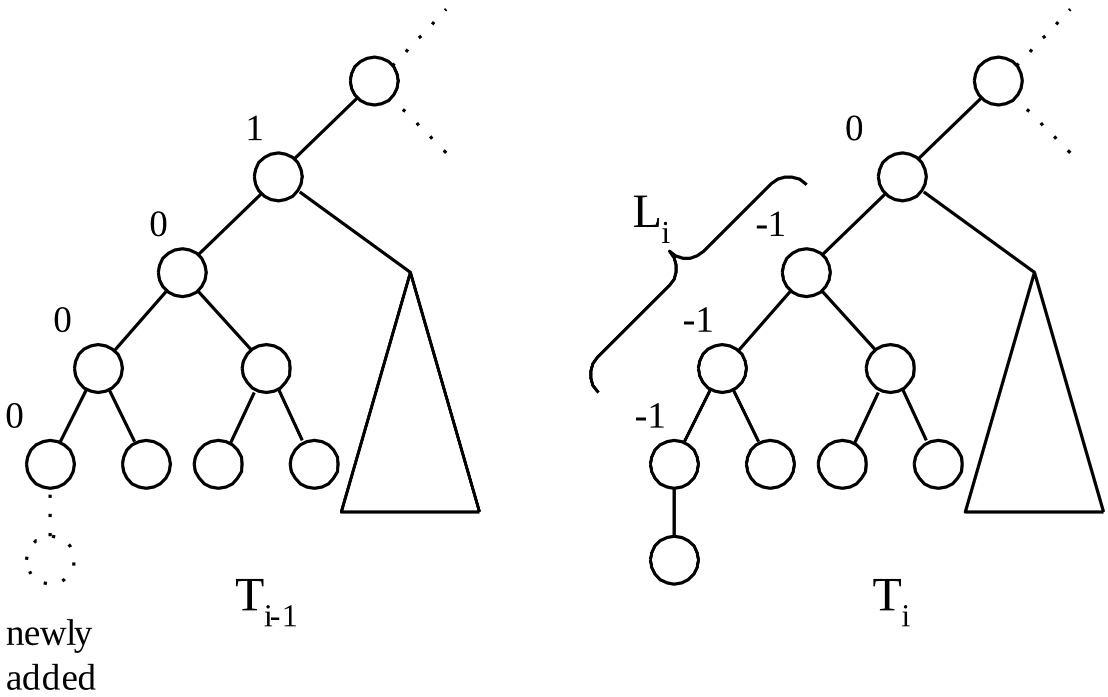
X_1: total # of balance factor changing from 0 to +1 or -1 during these m insertions

(rebalancing cost)

$X_1 = \sum_{i=1}^{m} L_i$, we want to find X_1.

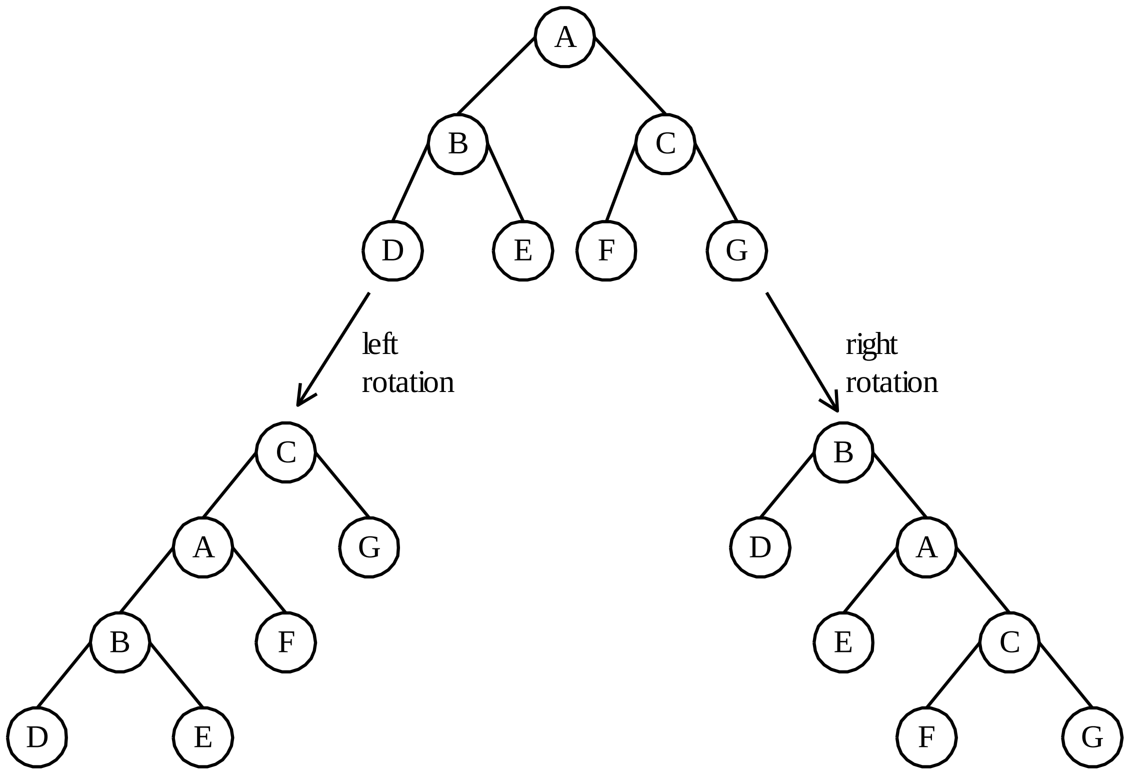
Val(T): # of unbalanced node in T
(height balance $\neq$ 0)

<u>Case 1</u>: Absorption (tree height not increased, no need of rebalancing)

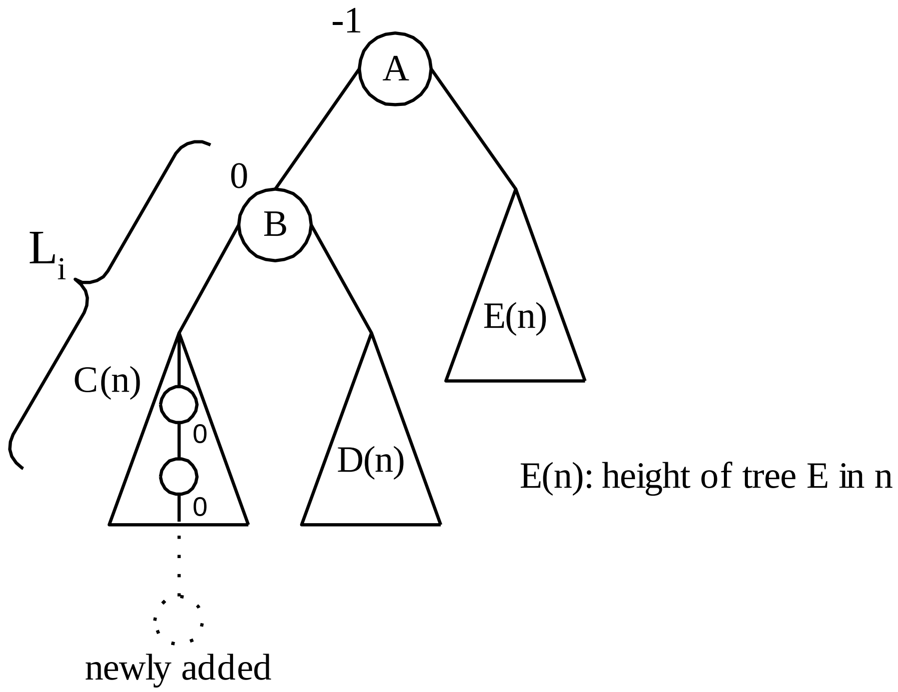


$$Val(T_i)=Val(T_{i-1})+(L_i-1)$$

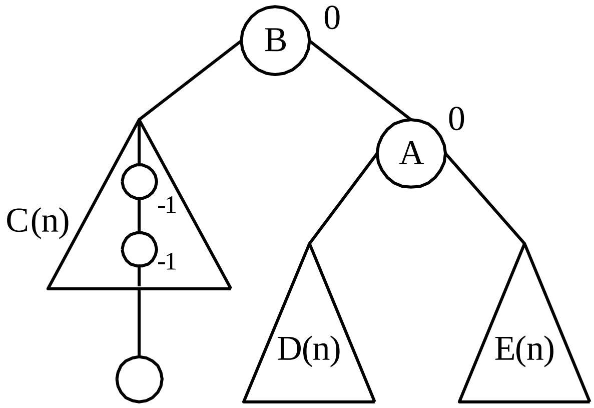
Case 2: Rebalancing the tree.



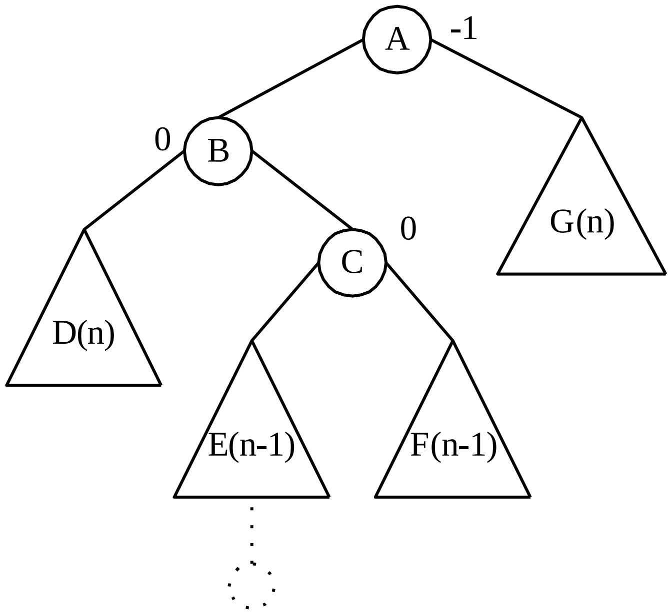
Case 2.1 single rotation



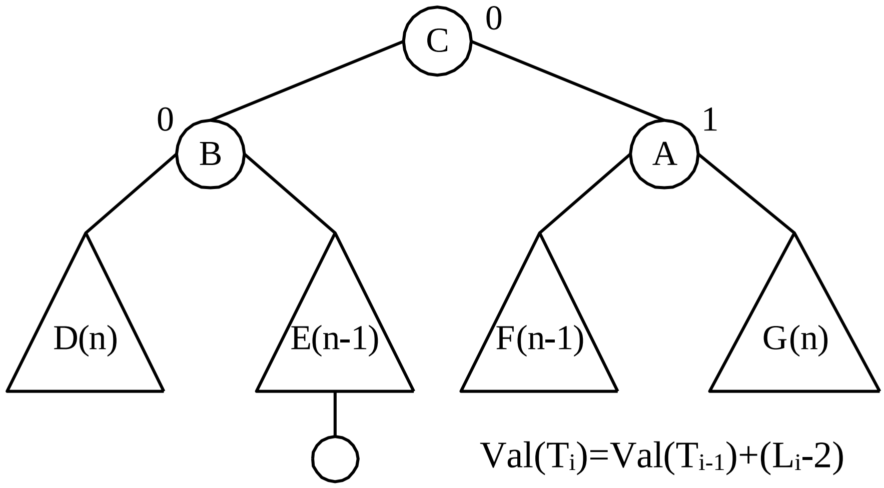
after a right rotation on the subtree rooted at A:



case 2.2 double rotation



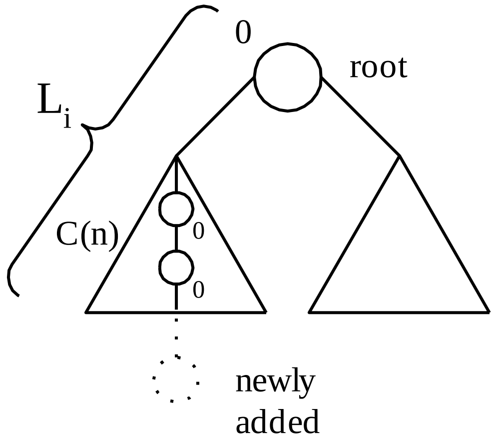
after a left rotation on the subtree rooted at B and a right rotation on the subtree rooted at A

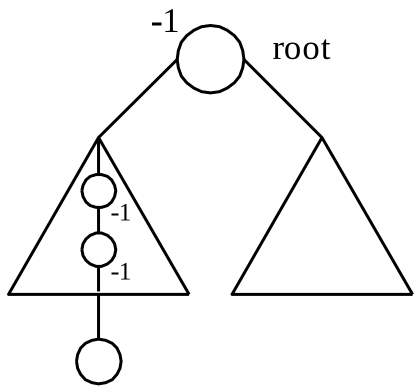


$Val(T_i)=Val(T_{i-1})+(L_i-2)$

Case 3: Height increase

(L_i is the height of the root)





$Val(T_i)=Val(T_{i-1})+L_i$

X_2: # of absorptions in case 1

X_3: # of single rotations in case 2

X_4: # of double rotations in case 2

X_5: # of height increases in case 3

$$Val(T_m) = Val(T_0)+\sum_{i=1}^{m} L_i - X_2 - 2(X_3+X_4)$$

$$=0+X_1-X_2-2(X_3+X_4)$$

$$Val(T_m) \le 0.618m \quad \text{(proved by Knuth)}$$

$$\Rightarrow X_1 = Val(T_m)+2(X_2+X_3+X_4)-X_2$$

$$\le 0.618m+2m$$

$$= 2.618m$$

- Amortized analysis of a self-organizing sequential search heuristics

3 methods for enhancing the performance of sequential search:

(1) Transpose Heuristics:

Query	Sequence
B	B
D	D B
A	D A B
D	D A B
D	D A B
C	D A C B
A	A D C B

(2) Move-to-the-Front Heuristics:

Query	Sequence
B	B
D	D B
A	A D B
D	D A B
D	D A B
C	C D A B
A	A C D B

(3) Count Heuristics: (decreasing order by the count)

Query	Sequence
B	B
D	B D
A	B D A
D	D B A
D	D B A
A	D A B
C	D A B C
A	D A B C

● analysis of the move to the front heuristics

interword comparison: unsuccessful comparison

intraword comparison: successful comparison

pairwise independent property:

For any sequence S and all pairs P and Q, # of interwod comparisons of P and Q is exactly # of comparisons made for the subsequence of S consisting of only P's and Q's.

e.g.

Query	Sequence	(A, B) comparison
C	C	
A	A C	
C	C A	
B	B C A	√
C	C B A	
A	A C B	√

# of comparisons made between A and B: 2

Consider the subsequence consisting of A and B:

Query	Sequence	(A, B) comparison
A	A	
B	B A	√
A	A B	√

# of comparisons made between A and B: 2

Query	Sequence	C	A	C	B	C	A
	(A, B)		0		1		1
	(A, C)	0	1	1		0	1
	(B, C)	0		0	1	1	
		0	1	1	2	1	2

There are 3 distinct interword comparisons:

(A, B), (A, C) and (B, C)

We can consider them separately and then add them up.

the total number of interword comparisons:

$0+1+1+2+1+2 = 7$

$C_M(S)$: # of comparisons of the move to front heuristics

$C_O(S)$: # of comparisons of the optimal static ordering

$\boxed{C_M \leq 2C_O(S)}$

Proof:

$Inter_M(S)$: # of interword comparisons of the move to the front heuristics

$Inter_O(S)$: # of interword comparisons of the optimal static ordering

Let S consist of a A's and b B's, a < b.

The optimal static ordering: BA

$$\left.\begin{array}{l} Inter_O(S) = a \\ Inter_M(S) \le 2a \end{array}\right\} \Rightarrow Inter_M(S) \le 2Inter_O(S)$$

Consider any sequence consisting of more than two items. Because of the pairwise independent property, we have $Inter_M(S) \le 2Inter_O(S)$

$Intra_M(S)$: # of intraword comparisons of the move to the front heuristics

$Intra_O(S)$: # of intraword comparisons of the optimal static ordering

$$Intra_M(S) = Intra_O(S)$$

$Inter_M(S) + Intra_M(S) \le 2Inter_O(S) + Intra_O(S)$

$\Rightarrow C_M(S) \le 2C_O(S)$

- The count heuristics has a similar result.

$C_C(S) \leq 2C_O(S)$, where $C_C(S)$ is the cost of the count heuristics.

- The transposition heuristics does not possess the pairwise independent property

We can not have a similar upper bound for the cost of the transposition heuristics.

e.g.

Consider pairs of distinct items independently.

Query	Sequence	C	A	C	B	C	A
	(A, B)		0		1		1
	(A, C)	0	1	1		0	1
	(B, C)	0		0	1	1	
		0	1	1	2	1	2

# of interword comparisons: 7 (not correct)

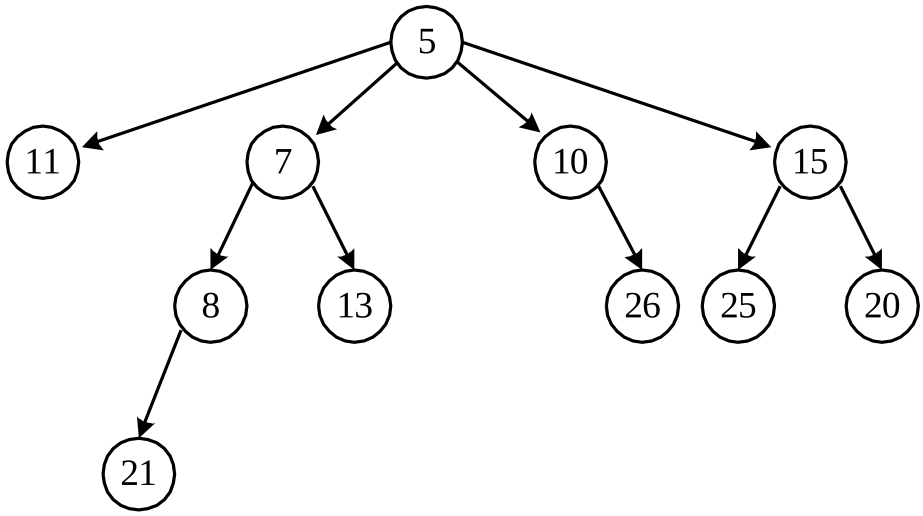
the correct interword comparisons:

Query Sequence	C	A	C	B	C	A
Data Ordering	C	AC	CA	CBA	CBA	CAB
Number of Interword Comparisons	0	1	1	2	0	2

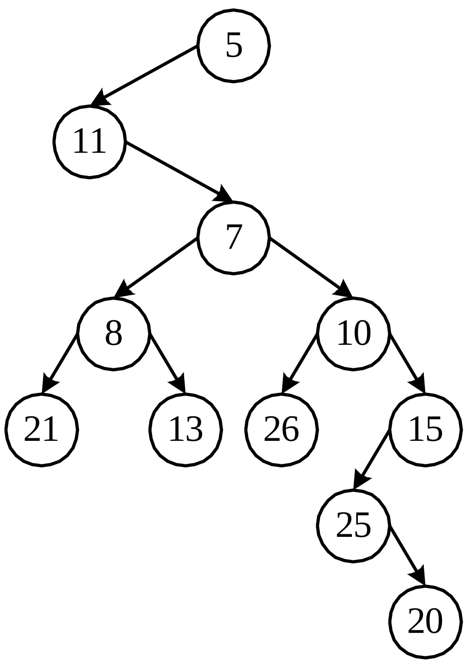
$\underbrace{0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 2}_{6}$

● The pairing heap and its amortized analysis

e.g. a pairing heap:

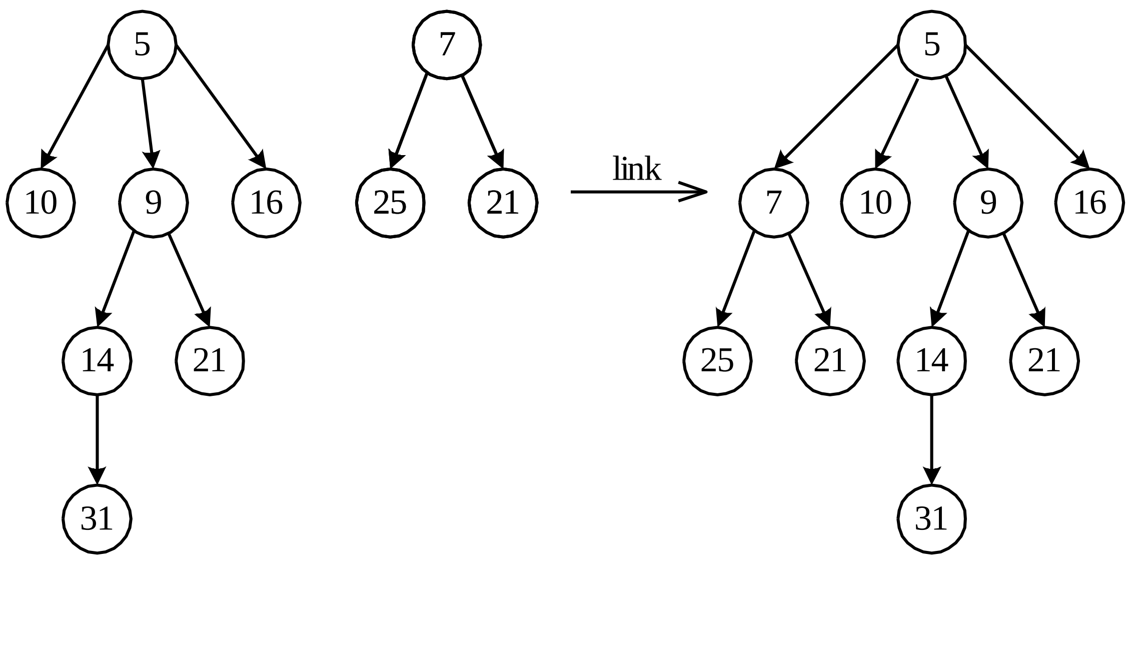


the binary tree representation:



- Basic operations for a pairing heap:
  (1) make heap(h): to create a new empty heap named h.
  (2) find min(h): to find the min. of heap h.
  (3) insert(x, h): to insert an element x into heap h.
  (4) delete min(h): to delete the min. from heap h.
  (5) meld(h_1, h_2): to create a heap by joining two heaps h_1 and h_2.
  (6) decrease key(Δ, x, h): to decrease the element x in heap h by the value Δ.
  (7) delete(x, h): to delete the element x from heap h.

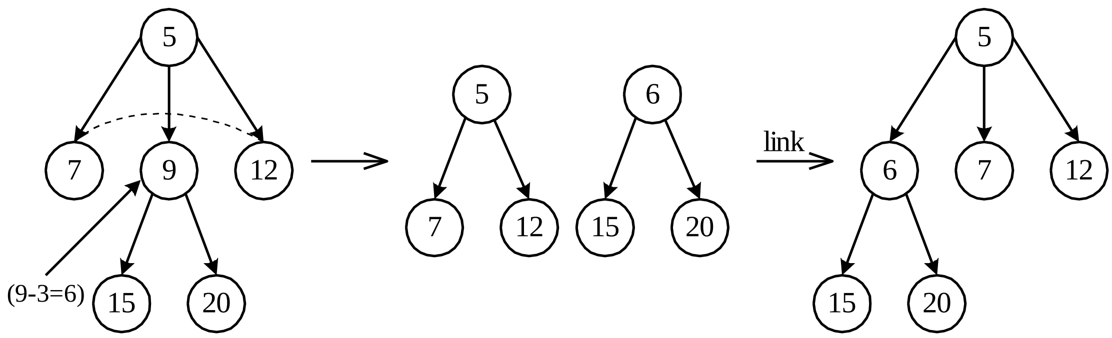
- the internal basic operation:
  link(h_1, h_2): to link two heaps into a new heap.



(1) make heap(h): trivial
(2) find min(h): trivial
(3) insert(x, h): apply link(h_1, h_2)
(4) delete min(h): delete the root, then apply many link(h_1, h_2)'s.
(5) meld(h_1, h_2): apply link(h_1, h_2)
(6) decrease(Δ, x, h):

    step 1: Subtract Δ from x.
    step 2: If x is root, then return.
    step 3: Cut the edge joining x and its parent. Then apply link(h_1, h_2)

e.g. decrease(3, 9, h):



(7) delete(x, h):

step 1: If x is root, then return (delete min(h)).

step 2: Otherwise,

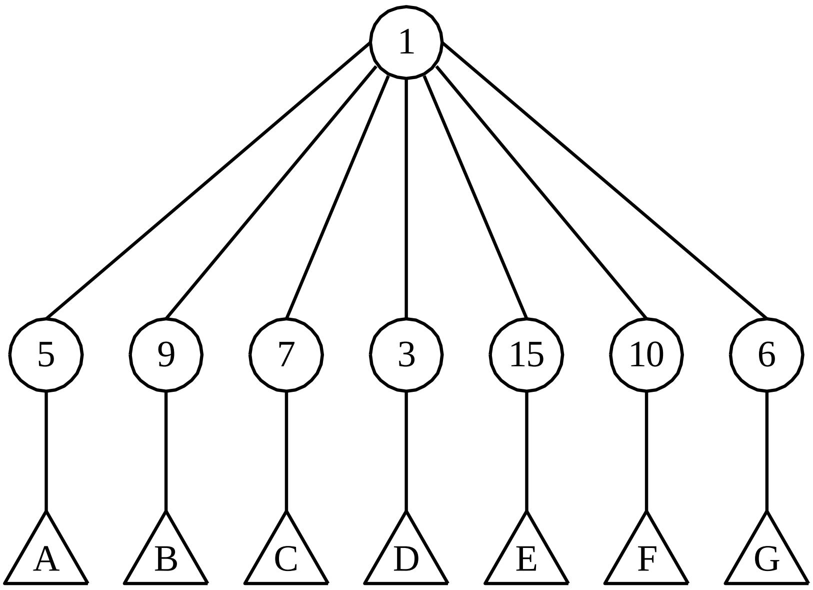
step 2.1: Cut the edge joining x to its parent.

step 2.2: Perform a delete min(h) on the tree rooted at x.

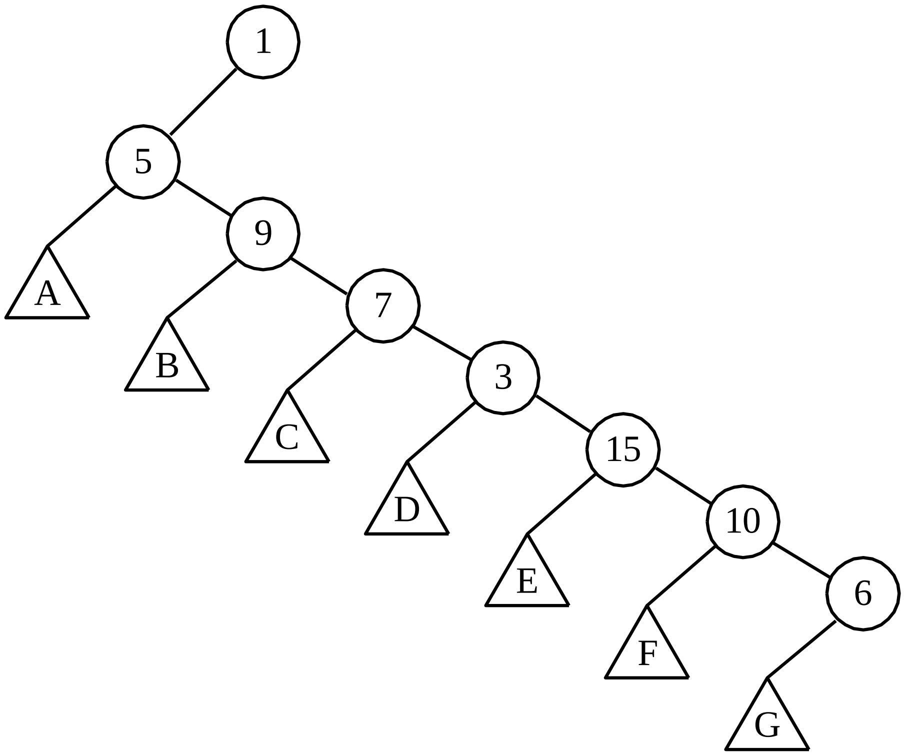
step 2.3: Link the resulting trees.

- The delete min(h) is critically important to the amortized analysis of the pairing heap.

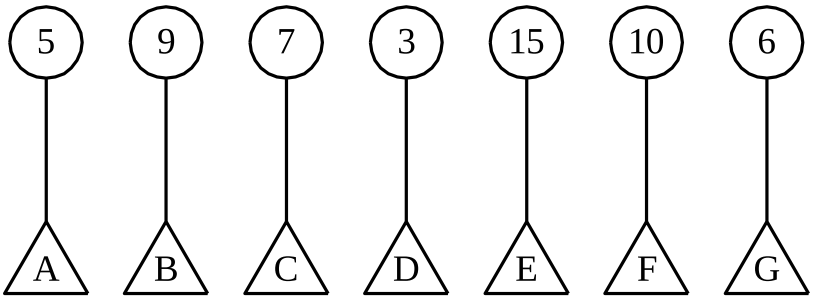
- How does the delete min(h) work?



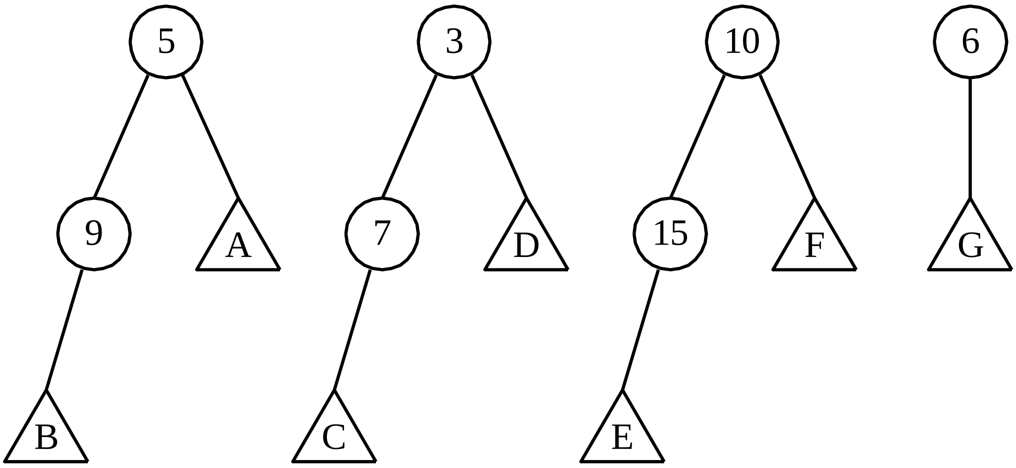
The binary tree representation:



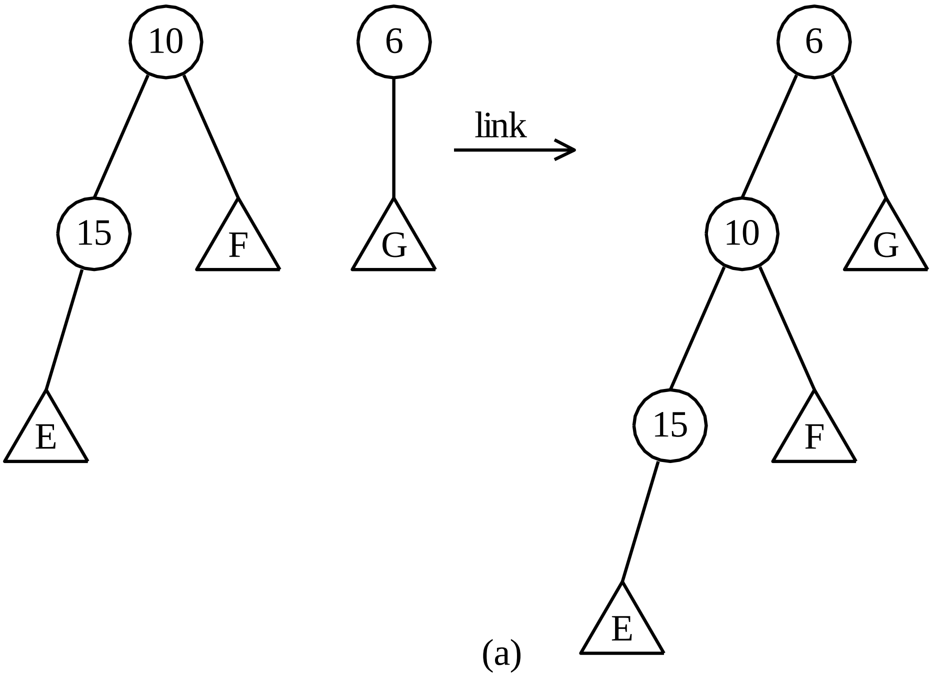
Step 1: delete the root:



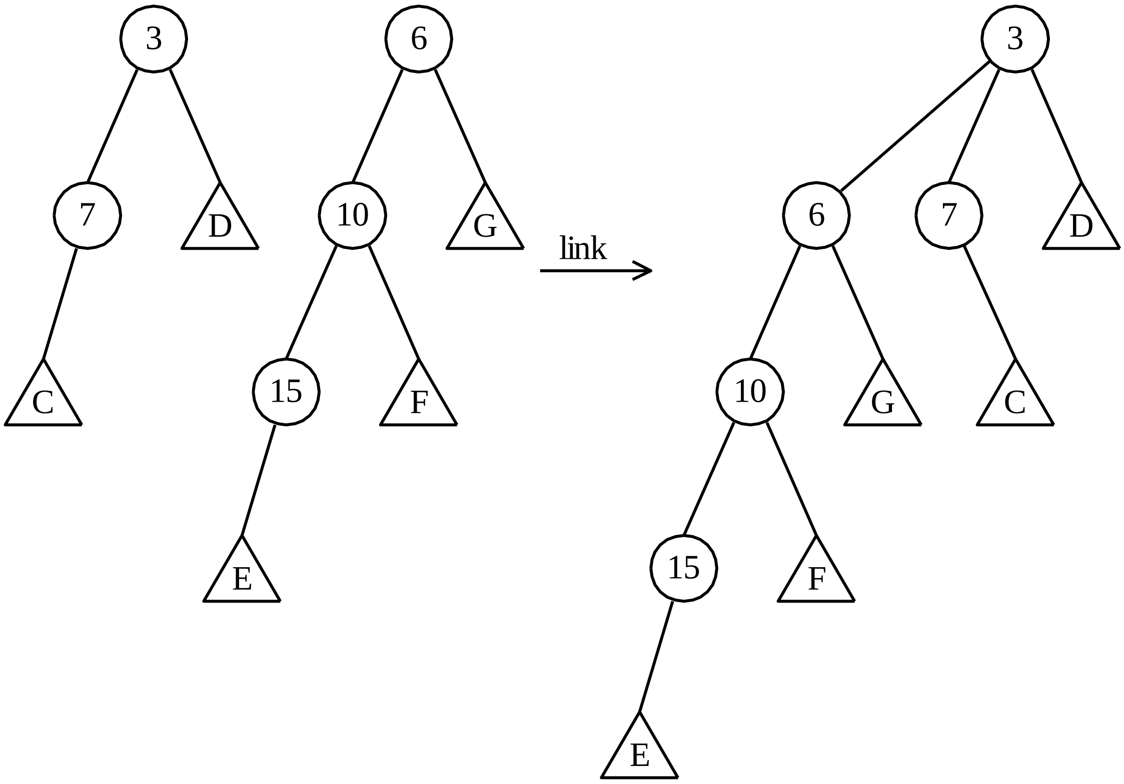
Step 2: pairwise melding:



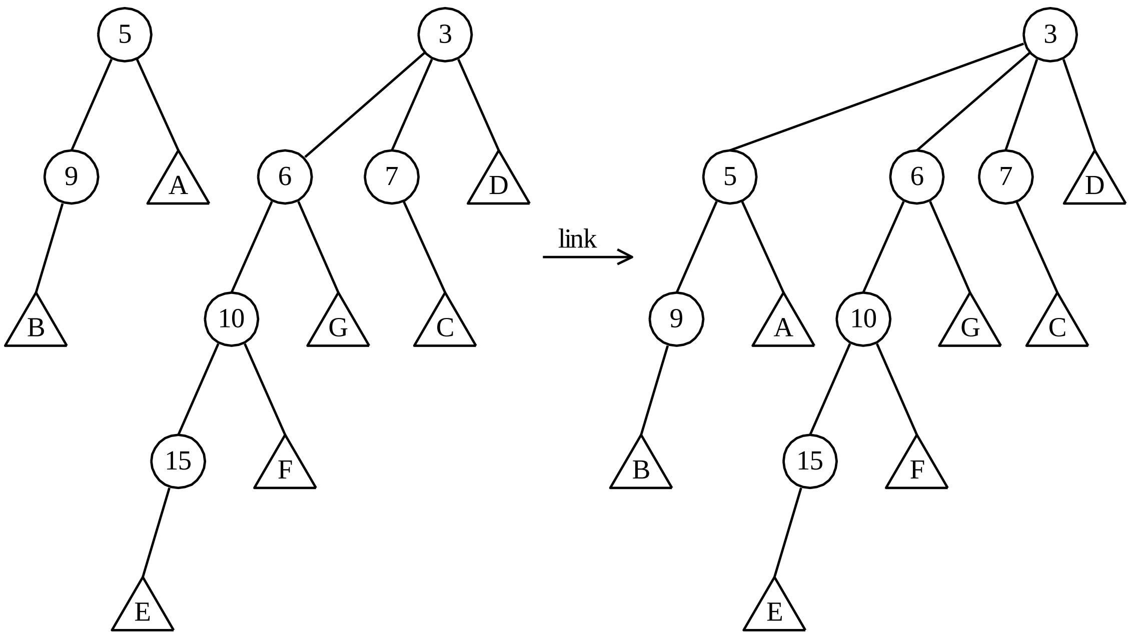
Step 3: melding with the last heap one by one:



(a)

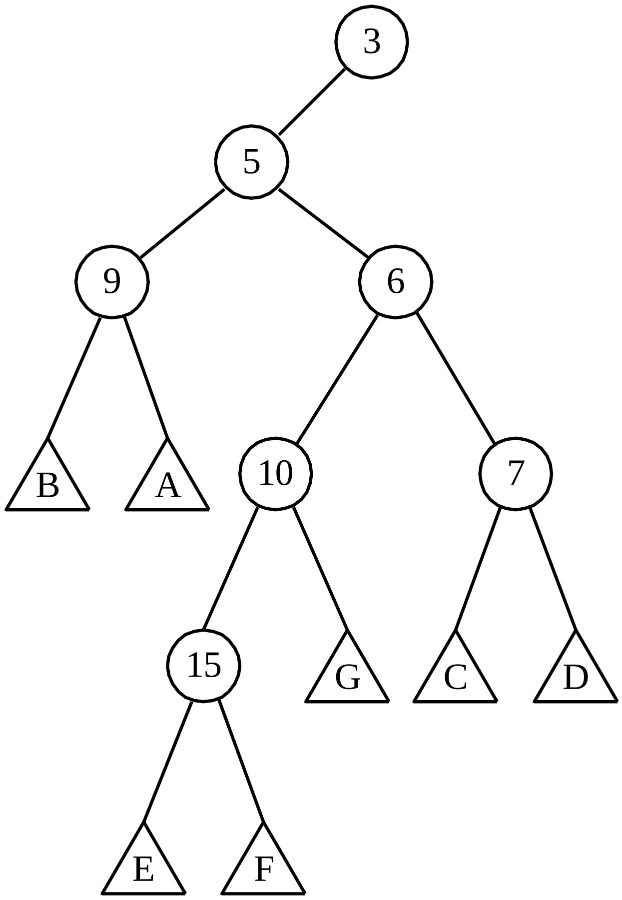


(b)



(c)

The binary tree representation of the resulting pairing heap:



- potential function

  s(x): # of nodes in the subtree rooted at x including x in a binary tree.

  r(x): rank of x, defind as log(s(x)).

  The potential of a binary tree, denoted as Φ, is the sum of the ranks of all nodes.

e.g.

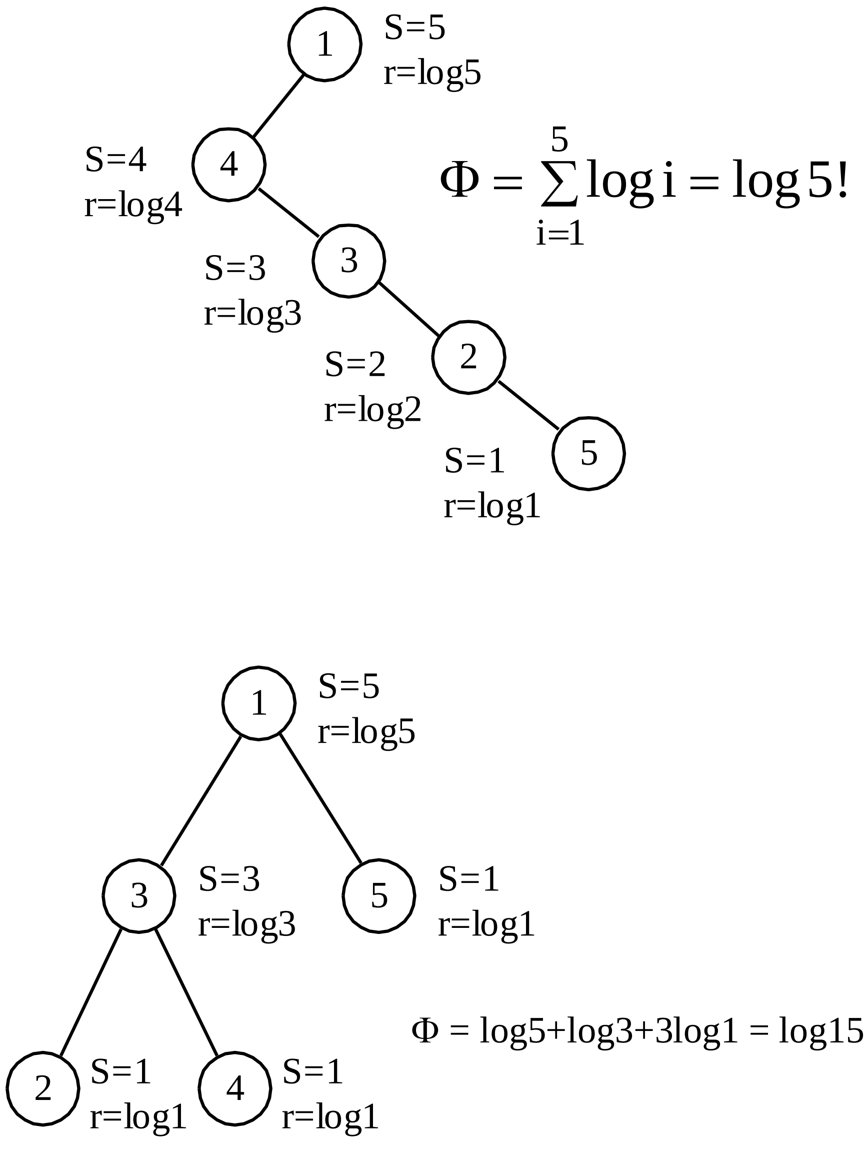


Fig. Potentials of Two Binary Trees

- the change of the potential

  $a_i = t_i + \Phi_i - \Phi_{i-1}$

  a_i : time spent for OP_i

  Φ_i : potential after OP_i

  Φ_{i-1}: potential before OP_i

step 1: deleting the root:

$\Delta\Phi_1 = -\log n$

step 2: pairwise melding:

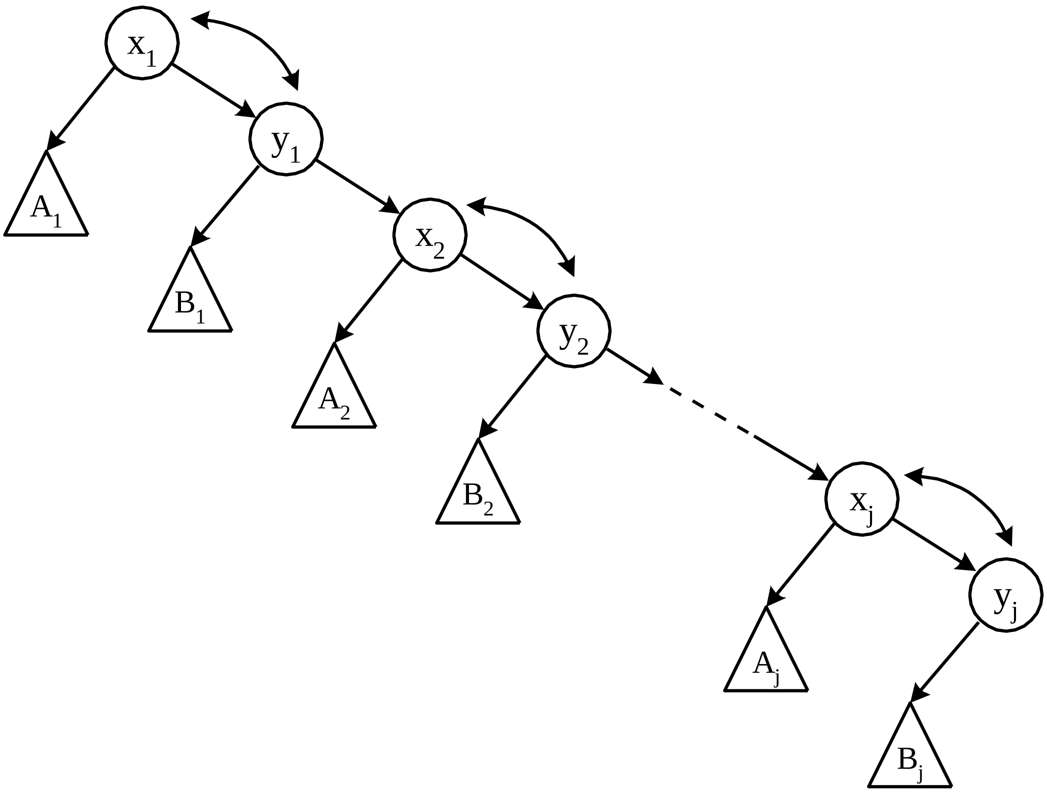
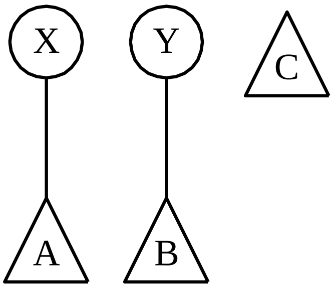
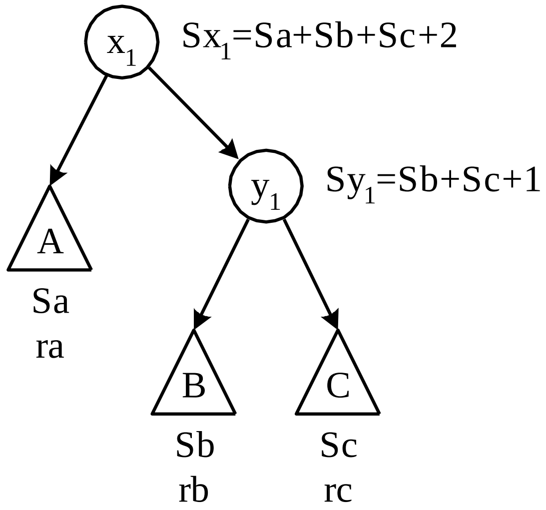


Fig. The Pairing Operations.

the melding of one pair:
pairing heap:

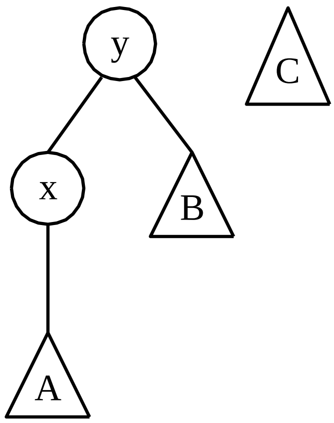


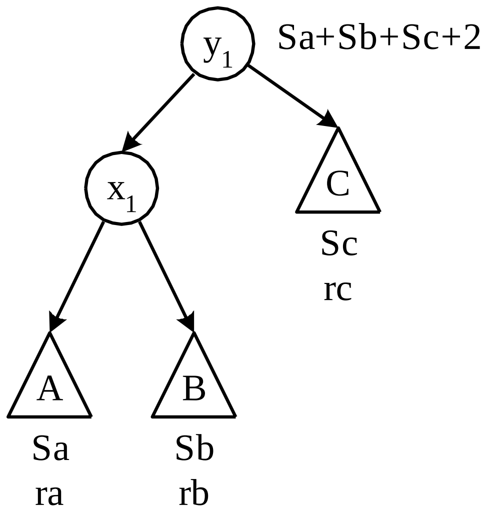


Φ_{before}=ra+rb+rc+
log(Sa+Sb+Sc+2)+
log(Sb+Sc+1)

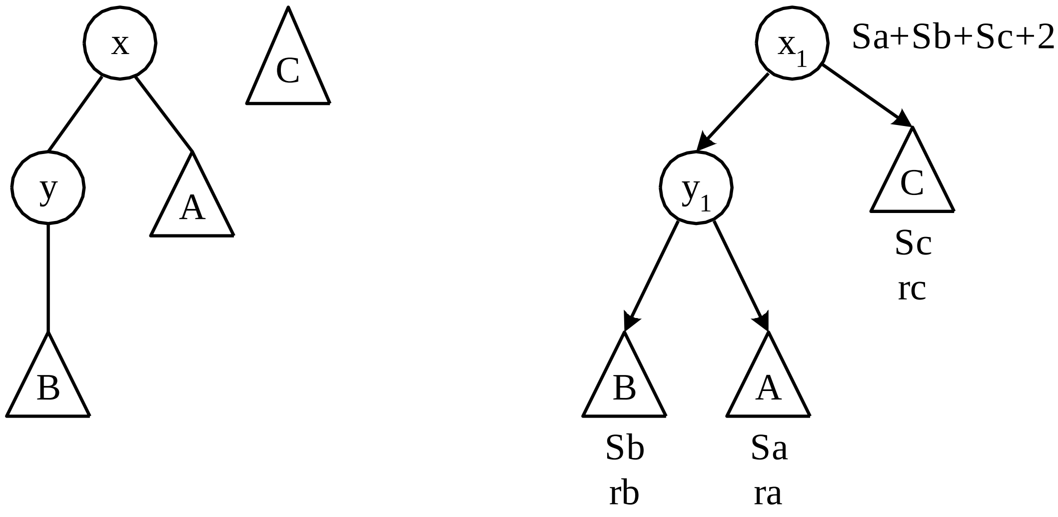
$\Downarrow$ meld

case 1: $y \leq x$





case 2: y > x



Φ_{after} = ra+rb+rc+log(Sa+Sb+Sc+2)+log(Sa+Sb+1)

$\Delta\Phi_{pair} = \Phi_{after} - \Phi_{before}$

= log(Sa+Sb+1) – log(Sb+Sc+1)

$$\boxed{\text{If } p,\ q \geq 0 \text{ and } p+q \leq 1, \text{ then } \log p + \log q \leq -2}$$

Let $p = \frac{Sa+Sb+1}{Sa+Sb+Sc+2}$, $q = \frac{Sc}{Sa+Sb+Sc+2}$

$$\log\left(\frac{Sa+Sb+1}{Sa+Sb+Sc+2}\right) + \log\left(\frac{Sc}{Sa+Sb+Sc+2}\right) \leq -2$$

$$\log(Sa+Sb+1) - \log(Sc) \leq 2\log(Sa+Sb+Sc+2) - 2\log(Sc) - 2$$

$$\log(Sa+Sb+1) - \log(Sb+Sc+1) \leq 2\log(Sa+Sb+Sc+2) - 2\log(Sc) - 2$$

$(\because \log(Sc) < \log(Sb+Sc+1))$

$\Delta\Phi_{pair} \leq 2\log(Sx) - 2\log(Sc) - 2$

the last pair, $\Delta\Phi_{pair} \leq 2\log(Sx)$ $(\because Sc=0)$

$$\Delta\Phi_{\text{total pair}} = \sum_{i=1}^{j-1}(\text{ith}\,\Delta\Phi_{\text{pair}}) + \Delta\Phi_{\text{pair}} \text{ due to the jth pair}$$

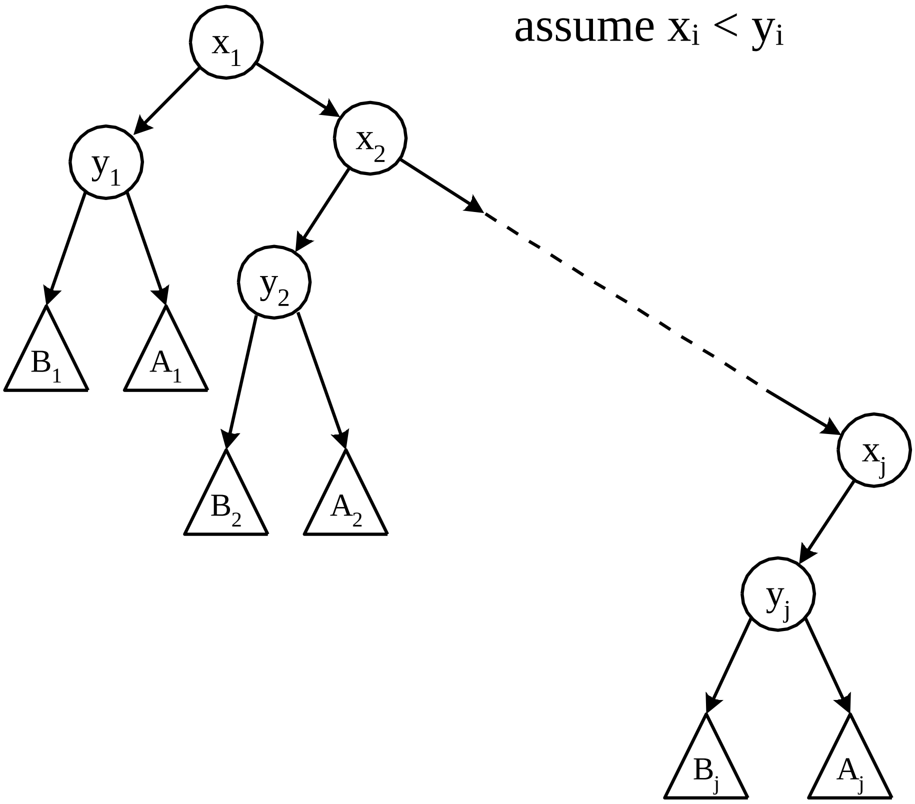
$$\le \sum(2\log(Sx_i) - 2\log(Sx_{i+1}) - 2) + 2\log(Sx_j)$$

$$= 2\log(Sx_1) - 2(j-1)$$

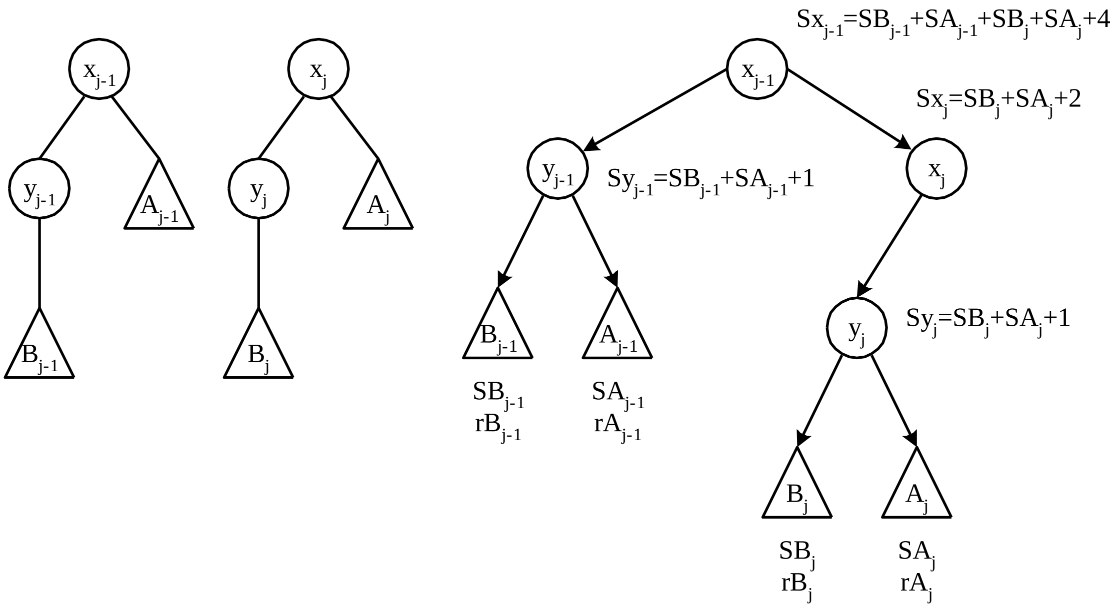
$$\le 2\log n - 2j + 2$$

step 3: melding with the last heap one by one:

assume $x_i < y_i$

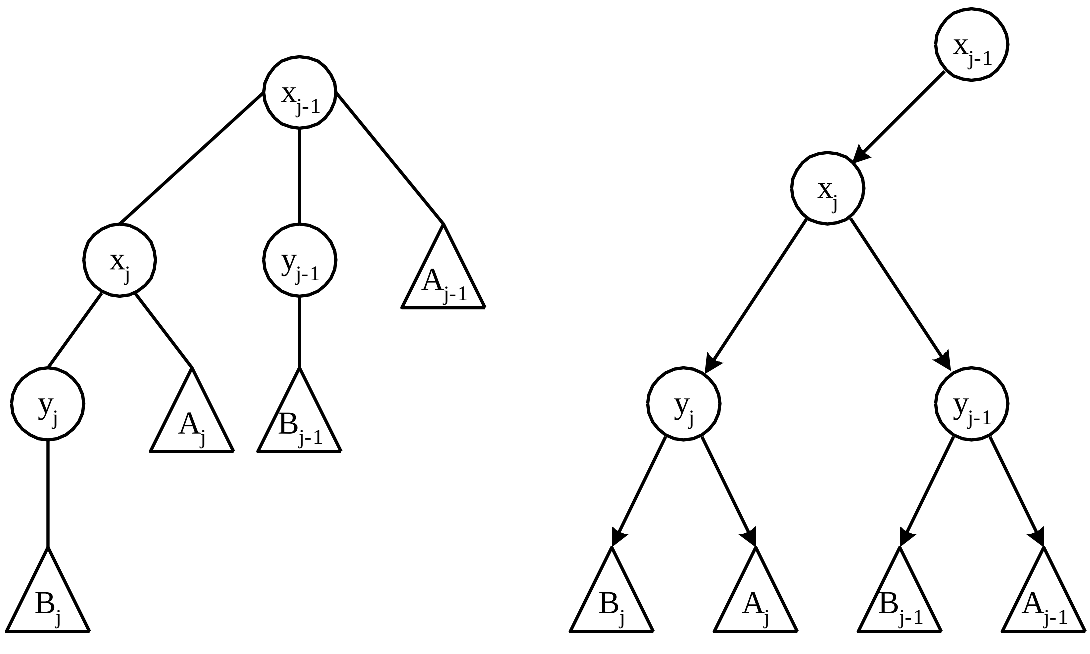


the last pair:



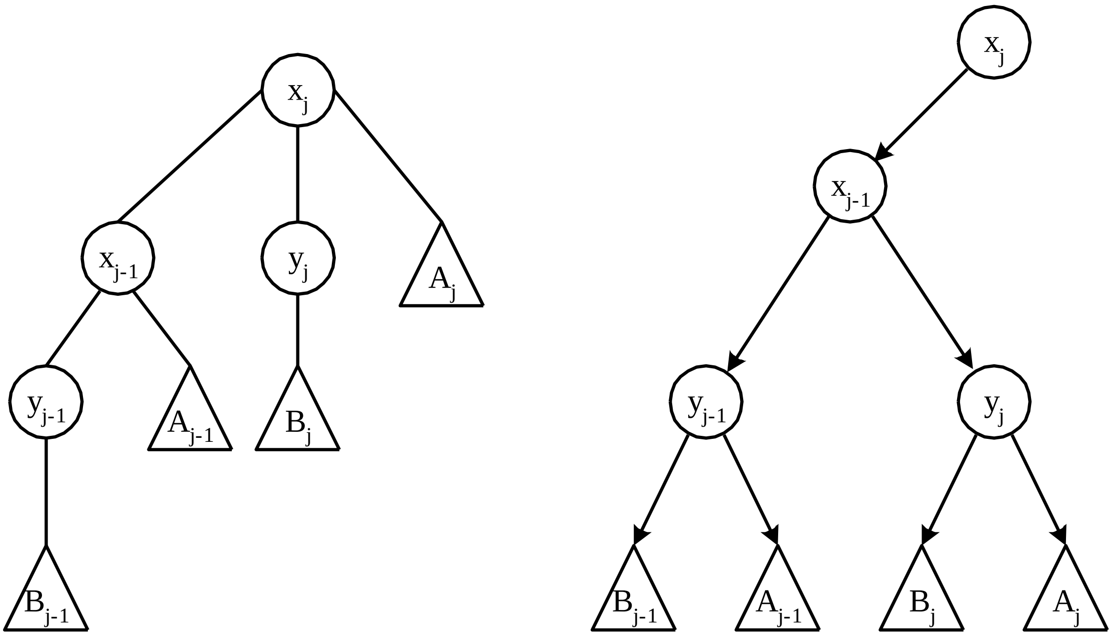
$\Downarrow$ meld

Case 1: $x_{j-1} \le x_j$



(a)

Case 2: $x_{j-1} > x_j$



(b)

$\Delta\Phi = \Phi_{after} - \Phi_{before} = \log(S_{A_{j-1}} + S_{B_{j-1}} + S_{A_j} + S_{B_j} + 3) - \log(S_{A_j} + S_{B_j} + 2)$

Let n_i = # of nodes of tree containing x_i, y_i, A_i and B_i

$$
\begin{aligned}
\Delta\Phi = &\log(n_1+n_2+\cdots+n_j-1) - \log(n_2+n_3+\cdots+n_j) \\
&+ \log(n_2+n_3+\cdots+n_j-1) - \log(n_3+n_4+\cdots+n_j) \\
&+ \cdots \\
&+ \log(n_{j-1}+n_j-1) - \log(n_j) \\
&\le \log(n-2) - \log(n_j) \\
&< \log(n-1)
\end{aligned}
$$

$$
\begin{aligned}
a &= t+\Phi'-\Phi \\
&\le 2j+1-\log n+(2\log n-2j+2)+\log(n-1) \\
&\le 2\log n+3 = O(\log n)
\end{aligned}
$$