CHAPTER 9 HEAP STRUCTURES

9.1 Min-max heaps

- Definition
  - A *double-ended priority queue* is a data structure that supports the following operations:

    (1) Insert an element with arbitrary key

    (2) Delete an element with the largest key

    (3) Delete an element with the smallest key
  - When only insertion and one of the two deletion operations is supported, we may use a min heap or max heap. A min-max heap however supports all of the operations just described.

– Definition:

A *mix-max* heap is a complete binary tree such that if it is not empty, each element has a field called *key*. Alternating levels of this tree are min levels and max levels, respectively. Let *x* be any node in a min-max heap. If *x* is on a min level then the element in *x* has the minimum key from among all elements in the subtree with root *x*. We call this node a *min* node. Similarly, if *x* is on a max level then the element in *x* has the maximum key from among all elements in the subtree with root *x*. We call this node a *max* node.

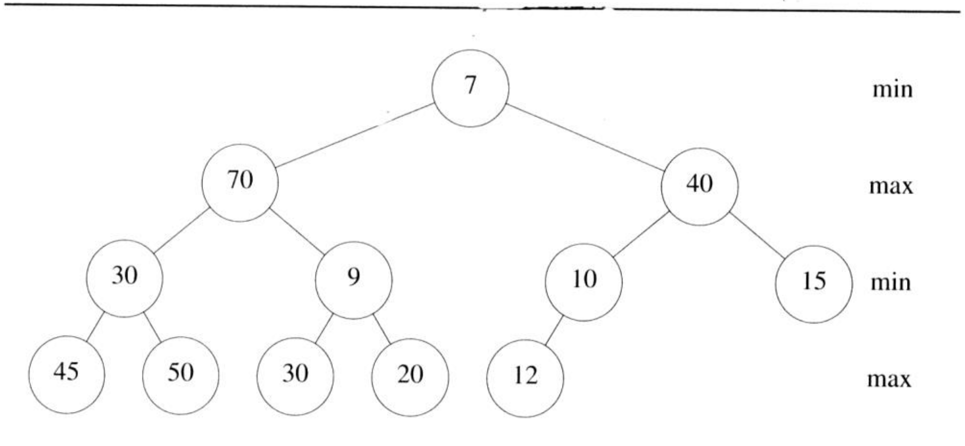


Figure 9.1: A 12 element min-max-heap

– Insertion into a min-max heap

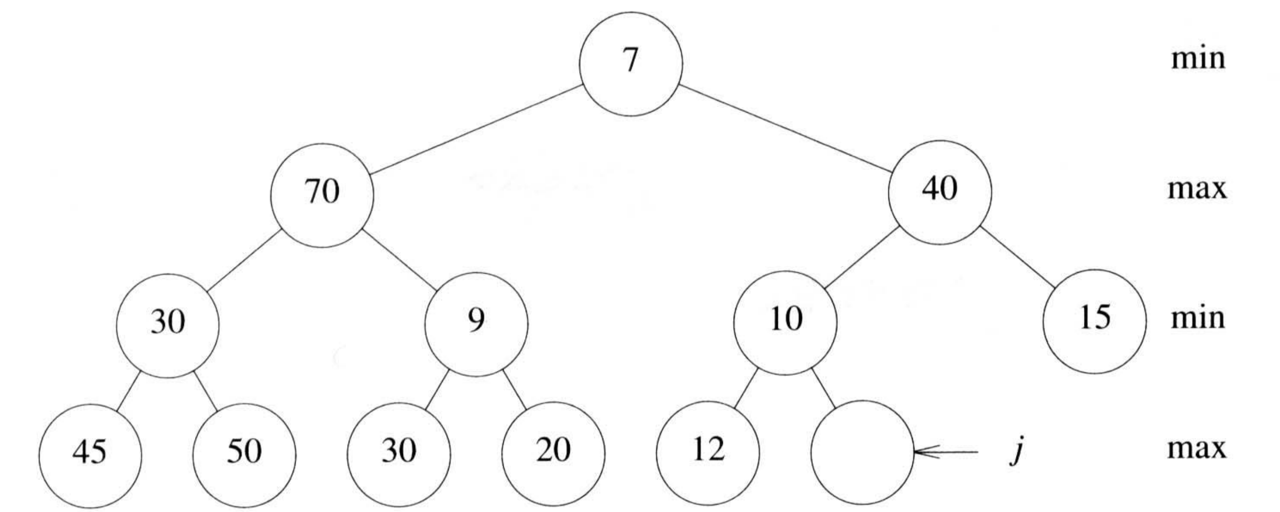
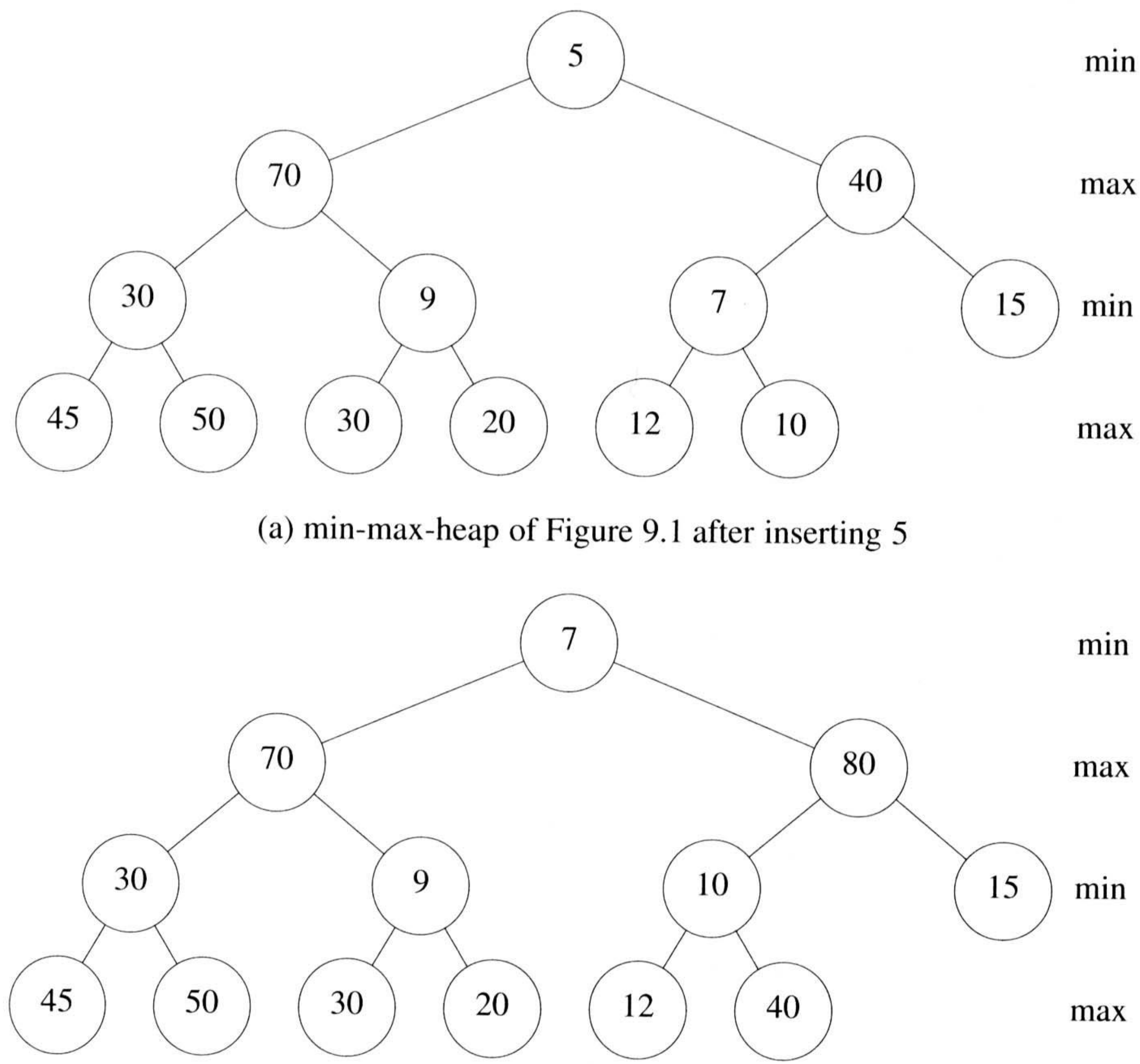


Figure 9.2: Min-max-heap of Figure 9.1 with new node *j*

Insertion at a "max" level: (Insertion at a "min" level 以類比作法)

1. **If it is smaller than its father (a "min"), then it must be smaller than all "max" above. So simply check the "min" ancestors.**
2. **If it is greater than its father (a "min"), then it must be greater than all "min" above. So simply check the "max" ancestors.**



(a) min-max-heap of Figure 9.1 after inserting 5

(b) min-max-heap of Figure 9.1 after inserting 80

Figure 9.3: Insertion into a min-max-heap

- **Analysis of min_max_insert:**
  - Since a min-max heap with *n* elements has O(log *n*) levels, the complexity of the *min_max_insert* function is O(log *n*).

– Deletion of min element

- If we wish to delete the element with smallest key, then this element is in the root.
- In the case of the min-max-heap of Figure 9.1, we are to delete the element with key 7.
  - Following the deletion, we will be left with a min-max-heap that has 11 elements. Its shape is shown in Figure 9.4.
  - The node with key 12 is deleted from the heap and the element with key 12 is reinserted into the heap.
  - As in the case of deletion from a min or max-heap, the reinsertion is done by examining the nodes of Figure 9.4 from the root down towards the levels.
- In general situation, we are to reinsert an element *item* into a min-max-heap, *heap*, whose root is empty. We consider the two cases:

– (1) *The root has no children*. In this case *item* is to be inserted into the root.

– (2) *The root has at least one child*. Now, the smallest key in the min-max-heap is in one of the children or grandchildren of the root. We determine which of these nodes has the smallest key. Let this be node *k*. The following possibilities need to be considered:

» *item.key* ≤ *heap*[*k*].*key*. Item may be inserted into the root as there is no element in heap with key smaller than item.key.

» *item.key* > *heap*[*k*].*key* and *k* is a child of the root. Since *k* is a max node, it has no descendants with key larger than *heap*[*k*].*key*. Hence, node *k* has no descendants with key larger than *item.key*. So, the element *heap*[*k*] may be moved to the root and *item* inserted into node *k*.

» *item.key* > *heap*[*k*].*key* and *k* is a grandchild of the root. In this case too, *heap*[*k*] may be moved to the root. Let *parent* be the parent of *k*. If *item.key* > *heap*[*parent*].*key*, then *heap*[*parent*] and *item* are to be interchanged. This ensures that the max node *parent* contains the largest key in the sub-heap with root *parent*. At this point, we are faced with the problem of inserting *item* into the sub-heap with root *k*. the root of this sub-min-max-heap is presently empty. This is quite similar to our initial situation where we were to insert *item* into the min-max-heap *heap* with root 1 and node 1 is initially empty. Therefore, we repeat the above process.

Delete a "min" – i.e., root:

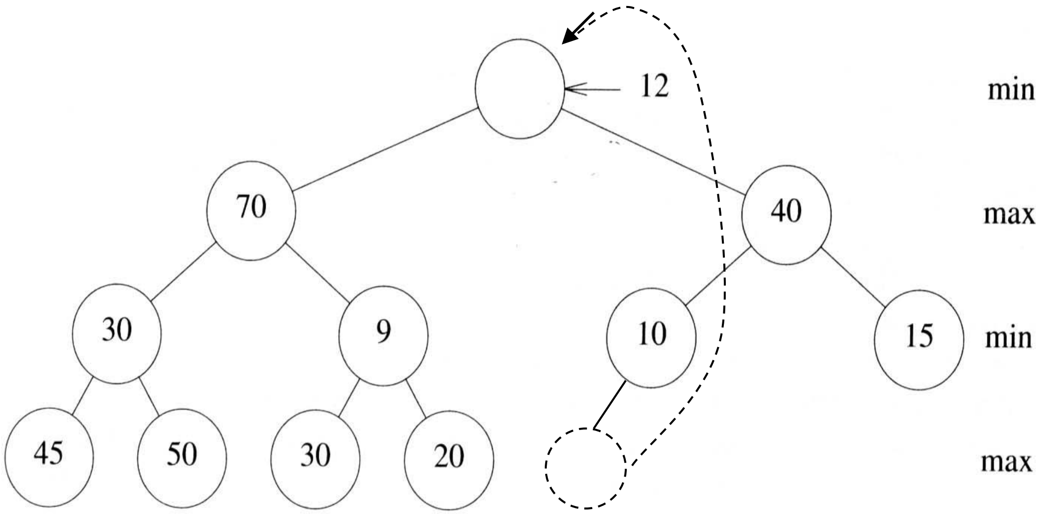


Figure 9.4: Shape of Figure 9.1 following a delete min

1. Single node case.
2. 找出兒、孫層最小值 heap[k].key.
   (a) "12" <= heap[k].key,
   12直接放入 root
   (b) "12" > heap[k].key and **k is a child**, 則 "12" 與 k 對調 (因為這case 一定只有一層(兒子)，沒有孫子層了。)
   (c) "12" > heap[k].key and k is a **grandchild**, 則 "12" 與 k 對調，之後，k不必去與其他的 min 再比對，因 k 已經是兒孫層最小的了。
   但 "12" 要與其下的 subtree 再做比較調整。調整的方式，就是以 "12" 為root, 並與上面作一樣的動作即可。

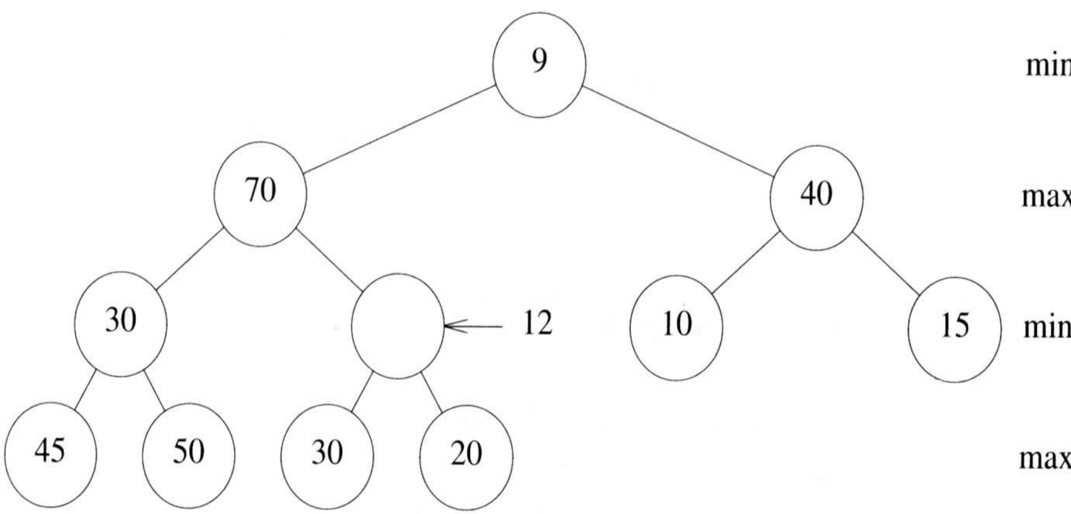


Figure 9.5: Figure 9.4 following the move of the element with key 9

Delete a "max": 則由它以下之兩個level找最大的，然後與上面一樣作類似的調整。

- **Analysis of *delete_min*:**
  - Since a min-max heap is a complete binary tree, *heap* has O(log *n*) levels. Hence, the complexity of *delete_min* is O(log *n*).

9.2 Deaps

- Definition
  - A deap is a double-ended heap that supports the double-ended priority queue operations of insert, delete min, and delete max.
  - As in the case of the min-max heap, these operations take logarithmic time on a deap. However, the deap is faster by a constant factor and algorithms are simpler.

– Definition:

- A *deap* is a complete binary tree that is either empty or satisfies the following properties:
  - The root contains no element.
  - The left subtree is a min-heap.
  - The right subtree is a max-heap.
  - If the right subtree is not empty, then let *i* be any node in the left subtree. Let *j* be the corresponding node in the right subtree. If such a *j* does not exist, then let *j* be the node in the right subtree that corresponds to the parent of *i*. The key in node *i* is less than or equal to the key in *j*.

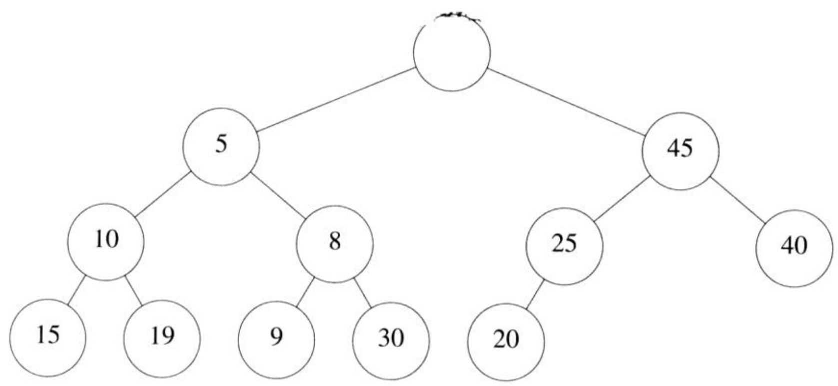


Figure 9.6: An 11 element deap

– In an n element deap, $n > 1$, the min element is in the root of the min-heap while the max element is in the root of max-heap.

– If $n = 1$, then the min and max elements are the same and are in the root of the min heap.

– Since a deap is a complete binary tree, it may be stored as an implicit data structure in a one dimensional array.

- In the case of a deap, position 1 of the array is not utilized.
- Let n denote the last occupied position in this array. Then the number of elements in the deap is n-1.
- If i is a node in the min-heap, then its corresponding node in the max-heap is $i+2^{\lfloor \log_2 i \rfloor -1}$. Hence the j defined in property (4) of the definition is given by:

$$j = i+2^{\lfloor \log_2 i \rfloor -1};$$

$$\text{if } (j > n)\ j/=2;$$

- Notice that if property (4) of the deap definition is satisfied by all leaf nodes *i* of min-heap, then it is satisfied by all remaining nodes of the min-heap too.

- Insertion into a deap
  - Suppose we wish to insert an element with key 4 into the deap of Figure 9.6. Following this insertion, the deap will have 12 elements in it and will thus have the shape shown in Figure 9.7. *j* points to the new node in the deap.

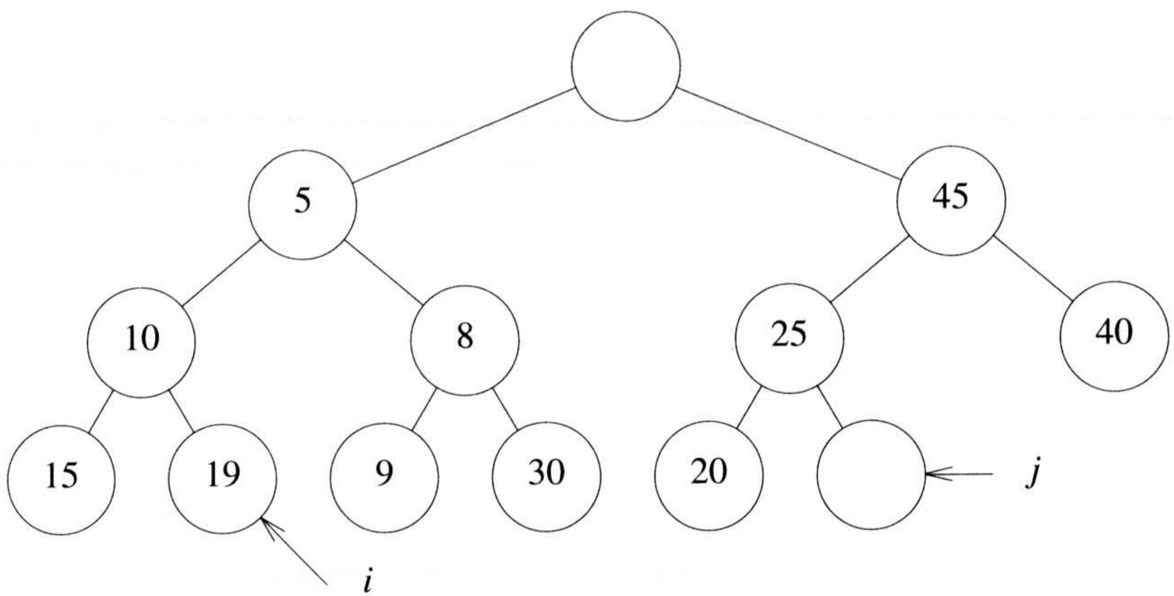


Figure 9.7: Shape of a 12 element deap

Insert 4 into the deap

- The insertion process begins by comparing the key 4 to the key in *j*'s corresponding node, *i*, in the min-heap. The node contains a 19.
- To satisfy property (4), we move the 19 to node *j*. Now, if we use the min-heap insertion algorithm to insert 4 into position *i*, we get the deap of Figure 9.8.

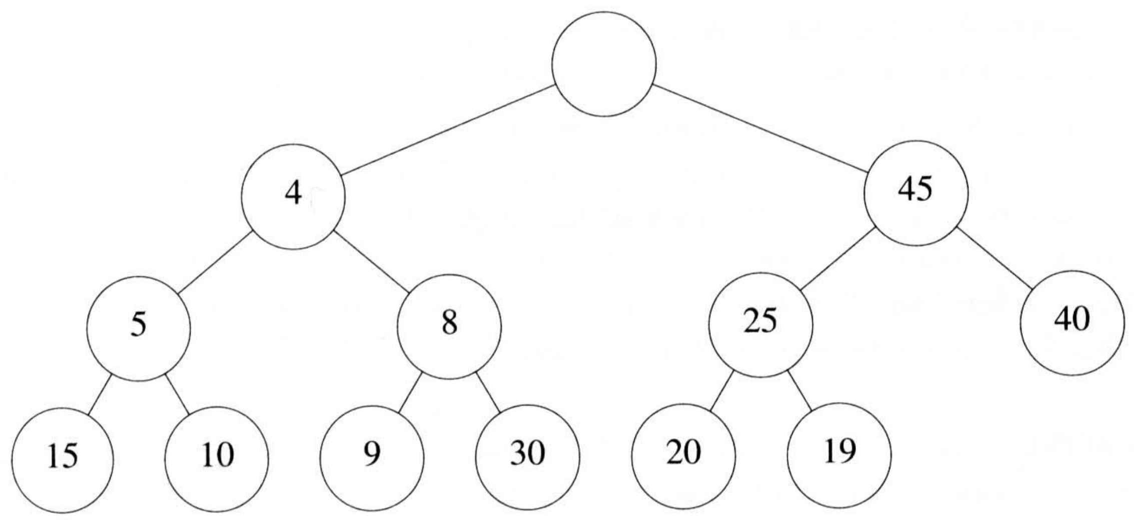


Figure 9.8: Deap of Figure 9.6 following the insertion of 4

- If instead of inserting a 4, we were to insert a 30 into the deap of Figure 9.6, then the resulting deap has the same shape as in Figure 9.7.
- Comparing 30 with the key 19 in the corresponding node *i*, we see that property (4) may be satisfied by using the max-heap insertion algorithm to insert 30 into position *j*. This results in the deap of Figure 9.9.

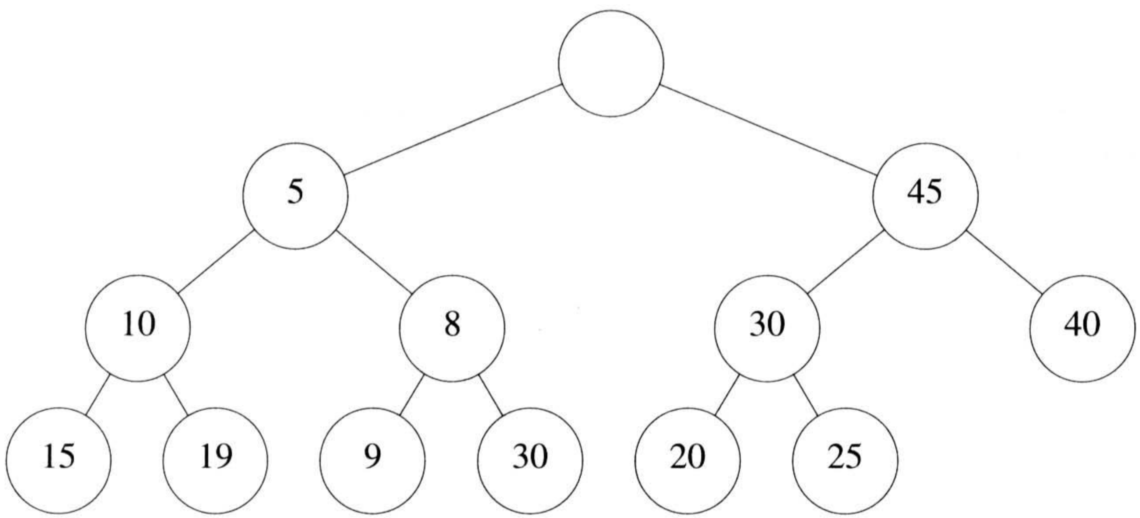


Figure 9.9: Deap of Figure 9.6 following the insertion of 30

Insert 30 into the deap of Figure 9.7

– **Analysis of *deap_insert*:**

- Time complexity of *deap_insert* is O(log *n*) as the the height of the deap is O(log *n*).

- Deletion of min element
  - **Example:** Suppose that we wish to remove the minimum element from the deap of Figure 9.6.
    - – We first place the last element (the one with key 20 (Figure 9.7)) in the deap into a temporary element, temp, since the deletion removes this node from the heap structure.
    - – Next, we fill the vacancy created in the min-heap root (node 2) by the removal of the minimum element. To fill this vacancy we move along the path from the root to a leaf node.
      - » Prior to each move, we place the smaller of the elements in the current node's children into the current node. We then move to the node previously occupied by the moved element.
      - » In this example, we first move 8 to node 2. Then we move 9 into the node formerly occupied by 8.
    - – Now, We have an empty leaf and proceed to insert 20 into this. We compare 20 with the key 40 in its max partner. Since 20 < 40, no exchange is needed and we proceed to insert 20 into the min-heap beginning at the empty position. This operation results in the deap of Figure 9.10.

» *max_partner*(n). This function computes the max-heap node that corresponds to the parent of the min-heap position n.

$$max\_partner(n) = (n + 2^{\lfloor \log_2 n \rfloor - 1})/2$$

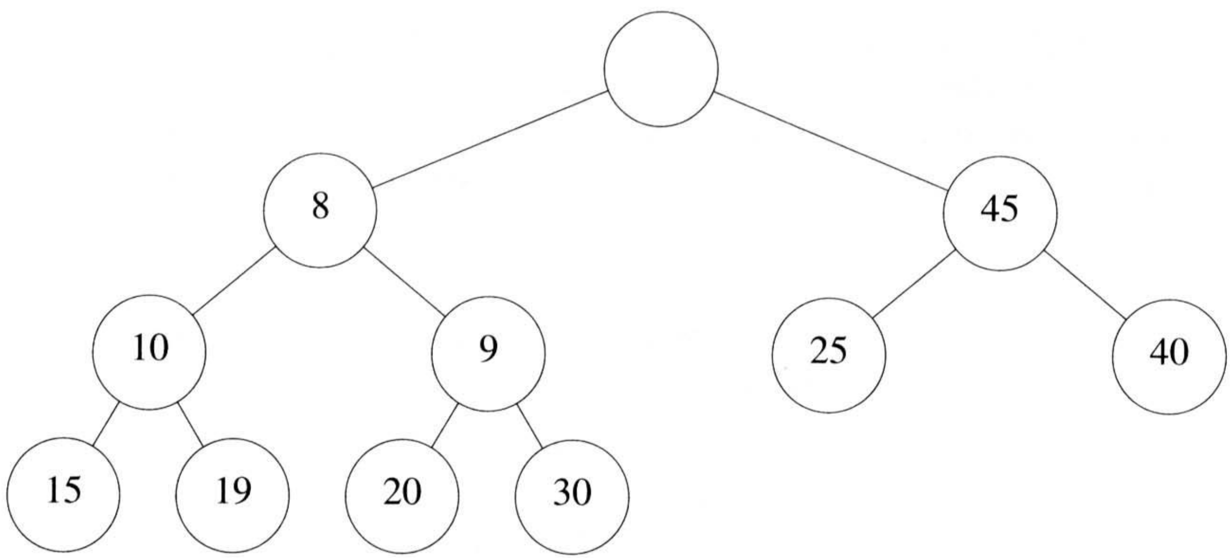


Figure 9.10: Deap of Figure 9.6 following a delete min

– **Analysis of *deap_delete_min*:**

- Time complexity of *deap_delete_min* is O(log *n*) as the the height of the deap is O(log *n*).

9.3 Leftist trees

- In the preceding section we extended the definition of a priority queue by requiring that both delete max and delete min operations be permissible.
- In this section, we consider a different extension, *combine*. This requires us to combine two priority queues into a single priority queue.
  - One application for this is when the server for one priority queue shuts down. At this time, it is necessary to combine its priority queue with that of a functioning server.

- Simple comparisons
  - Let *n* be the total number of elements in the two priority queues that are to be combined.
  - If heaps are used to represent priority queues, then the combine operation takes O(*n*) time.
  - Using a leftist tree, the combine operation as well as the normal priority queues operations take logarithmic time.
- In order to define a leftist tree, we need to introduce the concept of an extended binary tree.
  - An *extended binary tree* is a binary tree in which all empty binary subtrees have been replaced by a square node.
    - Figure 9.11 shows two example binary trees. There corresponding extended binary tree are shown in Figure 9.12.
  - The square nodes in an extended binary tree are called *external nodes*. The original (circular) nodes of the binary tree are called *internal nodes*.

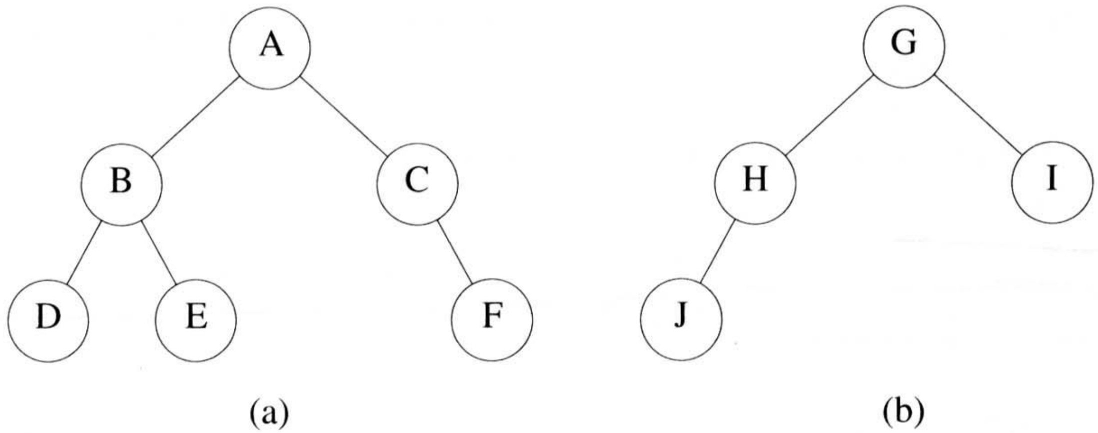


Figure 9.11: Two binary trees

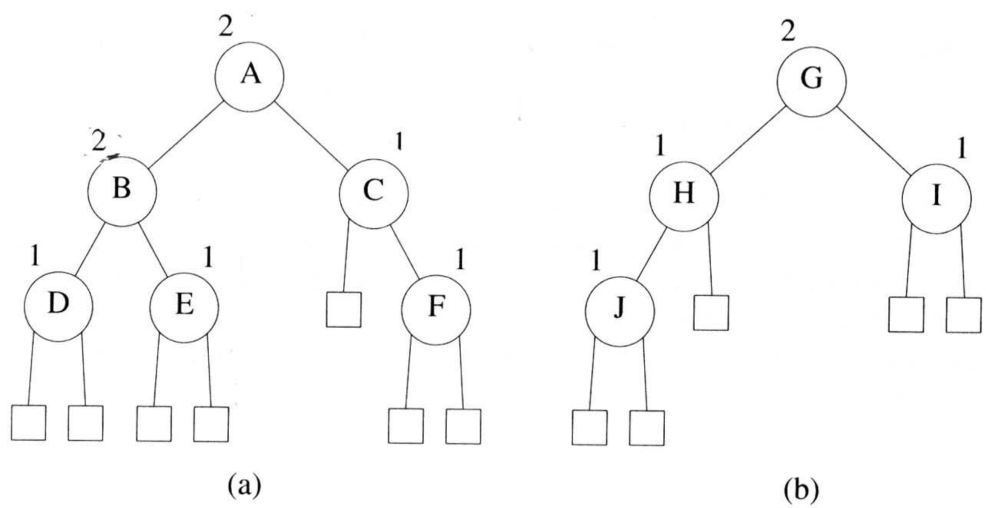


Figure 9.12: Extended binary trees corresponding to Figure 9.11

- Let *x* be a node in an extended binary tree. Let *left_child* (*x*) and *right_child* (*x*), respectively, denote the left and right children of the internal node *x*. Define *shortest* (*x*) to be the length of a shortest path from *x* to an external node.

$$shortest(x) = \begin{cases} 0, & if\ x\ is\ an\ external\ node \\ 1 + \min\{shortest(left\_child(x)), shortest(right\_child(x))\}, & otherwise \end{cases}$$

– Definition:

A *leftist tree* is a binary tree such that if it is not empty, then *shortest* (*left_child* (x)) ≥ *shortest* (*right_child* (x)) for every internal node x.

– Example:

The binary tree of Figure 9.11(a) which corresponds to the extended binary tree of Figure 9.12(a) is not a leftist tree as *shortest* (*left_child*(C)) = 0 while *shortest* (*right_child*(C)) = 1. The binary tree of Figure 9.11(b) is a leftist tree.

– Lemma 9.1:

Let x be the root of a leftist tree that has n (internal) nodes.

$$n \geq 2^{shortest(x)} - 1$$

The rightmost root to external node path is the shortest root to external node path. Its length is *shortest*(x).

– Definition:

- A *min-leftist tree* (*max leftist tree*) is a leftist tree in which the key value in each node is no larger (smaller) than the key values in its children (if any). In other words, a min (max) leftist tree is a leftist tree that is also a min (max) tree.

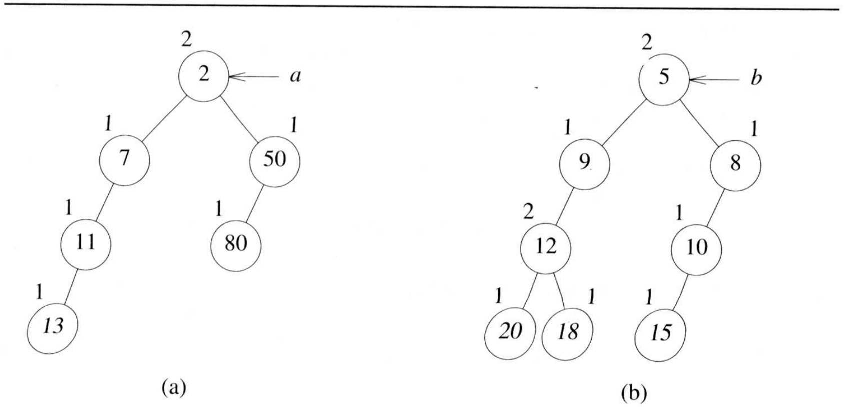


Figure 9.13: Example min leftist trees

– Both insert and delete min operations can be implemented by using the combine operation.

- To insert an element, *x*, into a min-leftist tree, *a*, we first create a min-leftist tree, *b*, that contains the single element *x*. Then we combine the min-leftist trees *a* and *b*.
- To delete the min element from a nonempty min-leftist tree, *a*, we combine the min-leftist trees *a->left_child* and *a->right_child* and delete node *a*.

– Combine the min-leftist trees *a* and *b*.

- We obtain a new binary tree containing all elements in *a* and *b* by following the rightmost paths in *a* and/or *b*. This binary tree has the property that the key in each node is no larger than the keys in its children (if any).
- Next, we interchange the left and right subtrees of nodes as necessary to convert this binary tree into a leftist tree.

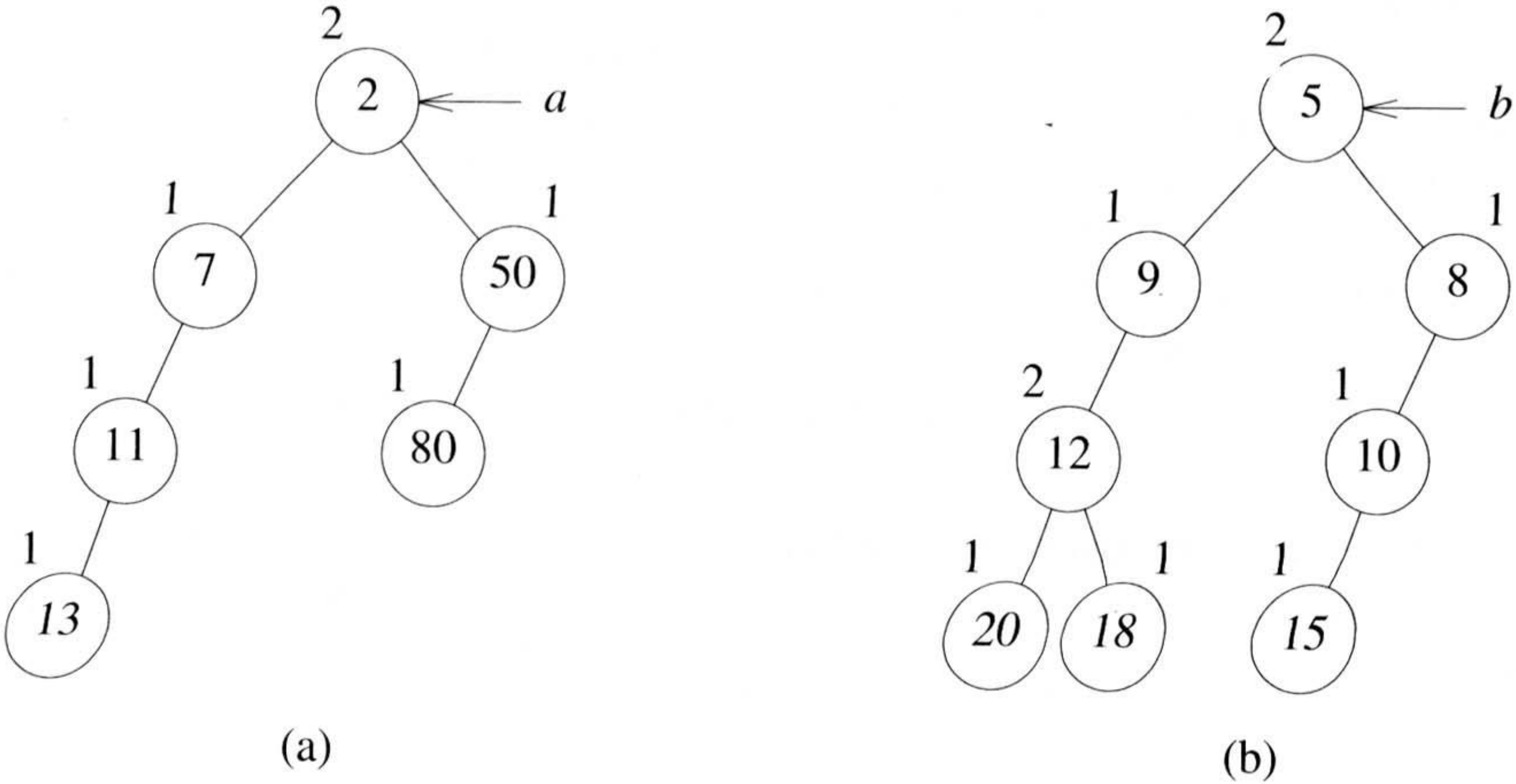


Figure 9.13: Example min leftist trees

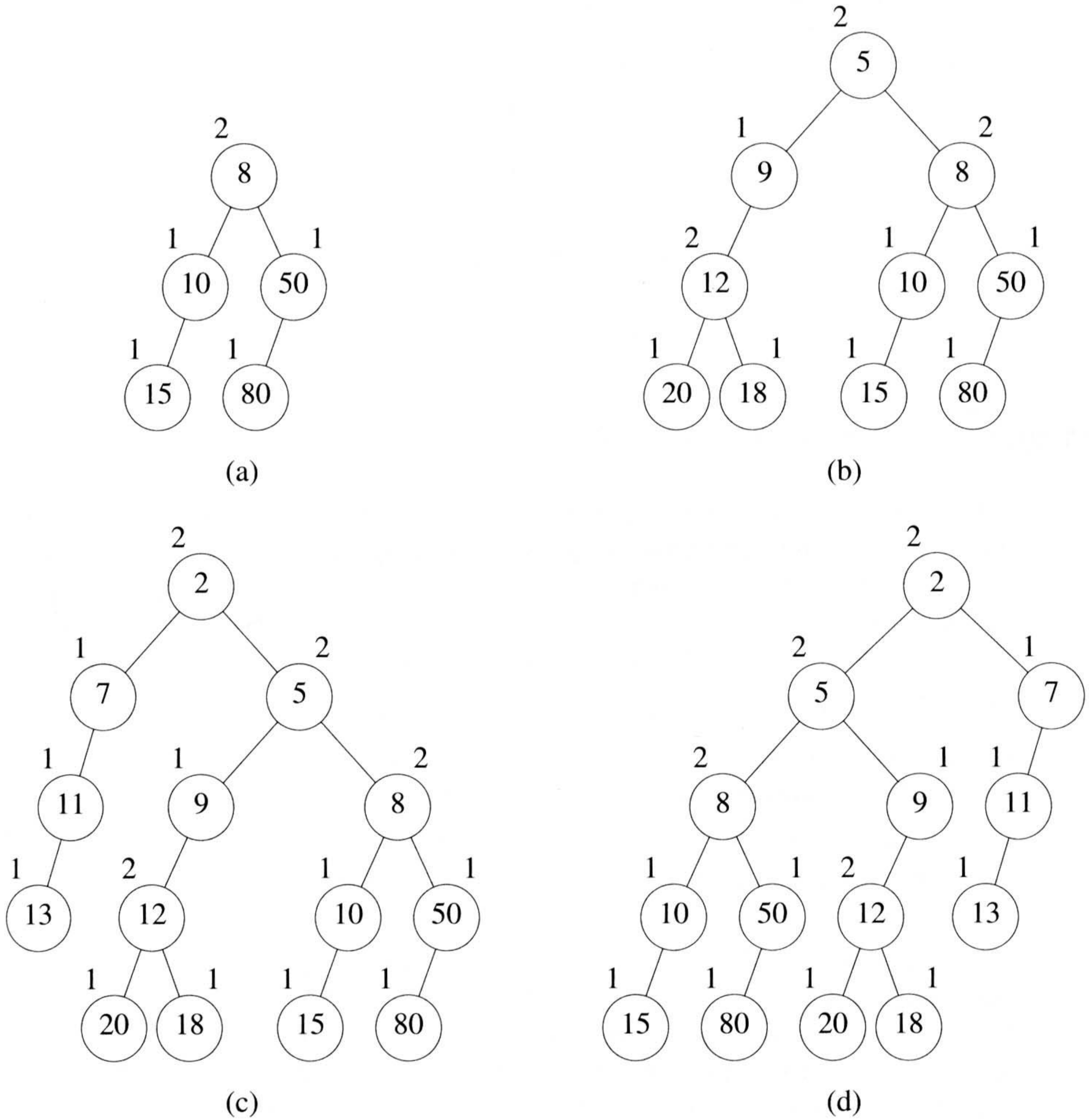


Figure 9.14: Combining the min leftist trees of Figure 9.13

– Combine方式

1. 比較兩個min-liftest tree 的root, 較小的root a 是合併後的 root.
2. 合併後root a的left subtree不變.
3. Right subtree則以root a 的right subtree與另一棵tree合併而得.
4. 必要時可以對調left與right subtrees以符合leftest tree的條件.

Insert : reduced to a combine operation.

Delete : delete the root and combine the two subtrees.

9.4 Binomial heaps

- Cost amortization
  - A binomial heap is a data structure that supports the same functions as supported by leftist trees.
  - Unlike leftist trees where an individual operation can be performed in O(log$n$) time, if we amortize part of the cost of expensive operations over the inexpensive ones, then the amortized complexity of an individual operation is either O(1) or O(log$n$) depending on the type of the operation.

	I1	I2	D1	I3	I4	I5	I6	D2	I7	
actual cost	1	1	8	1	1	1	1	10	1	Total = 25
amortized cost	2	2	6	2	2	2	2	6	1	Total = 25

- We can make the claim that the actual cost of any insert/delete min sequence is no more than $2*i+6*d$ where i and d are respectively, the number of insert and delete min operations in the sequence.
- Suppose that actual cost of a delete min is no more than ten, while that of an insert is one. Using actual costs, we can conclude that the sequence cost is no more than $i*10*d$.
- Combine these two bounds, we obtain $\min\{2*i+6*d, i+10*d\}$ as a bound on the sequence cost. Hence, using the notion of cost amortization it is possible to obtain tighter bounds on the complexity of a sequence of operations.

– We shall use the notion of cost amortization to show that while individual delete operations on an binomial heap may be expensive, the cost of any sequence of binomial heap operations is actually quite small.

- Definition of binomial heaps
  - As in case of heaps and leftist trees, there are two varieties of binomial heaps: min and max.
  - A *min-binomial heap* is a collection of min-trees while a *max-binomial heap* is a collection of max-trees.
    - We shall explicitly consider min-binomial heaps only. These will be referred to as *B-heaps*.

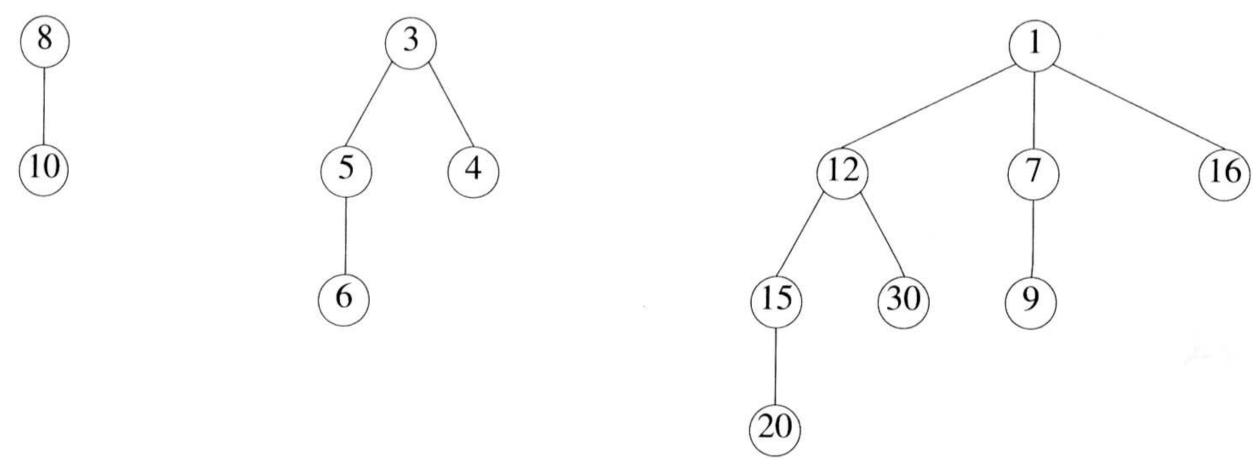


Figure 9.15: A B-heap with three min-trees

- Using B-heaps, we can perform an insert and a combine in O(1) actual and amortized time and a delete min in O(log*n*) amortized time.
- B-heaps are represented using nodes that have the fields: *degree, child, left_link, right_link*, and *data*.
  - The *degree* of a node is the number of children it has.
  - The *child* field is used to point to any one of its children (if any).
  - The *left_link* and *right_link* fields are used to maintain doubly linked circular list of siblings.
  - All the children of a node form a doubly linked circular list and the node points to one of these children.
  - The roots of the min-trees that comprise a B-heap are linked to form a doubly linked circular list.
  - The B-heap is then pointed at by a single pointer to the min-tree root with smallest key.

– Figure 9.16 shows the representation for the example of Figure 9.15.

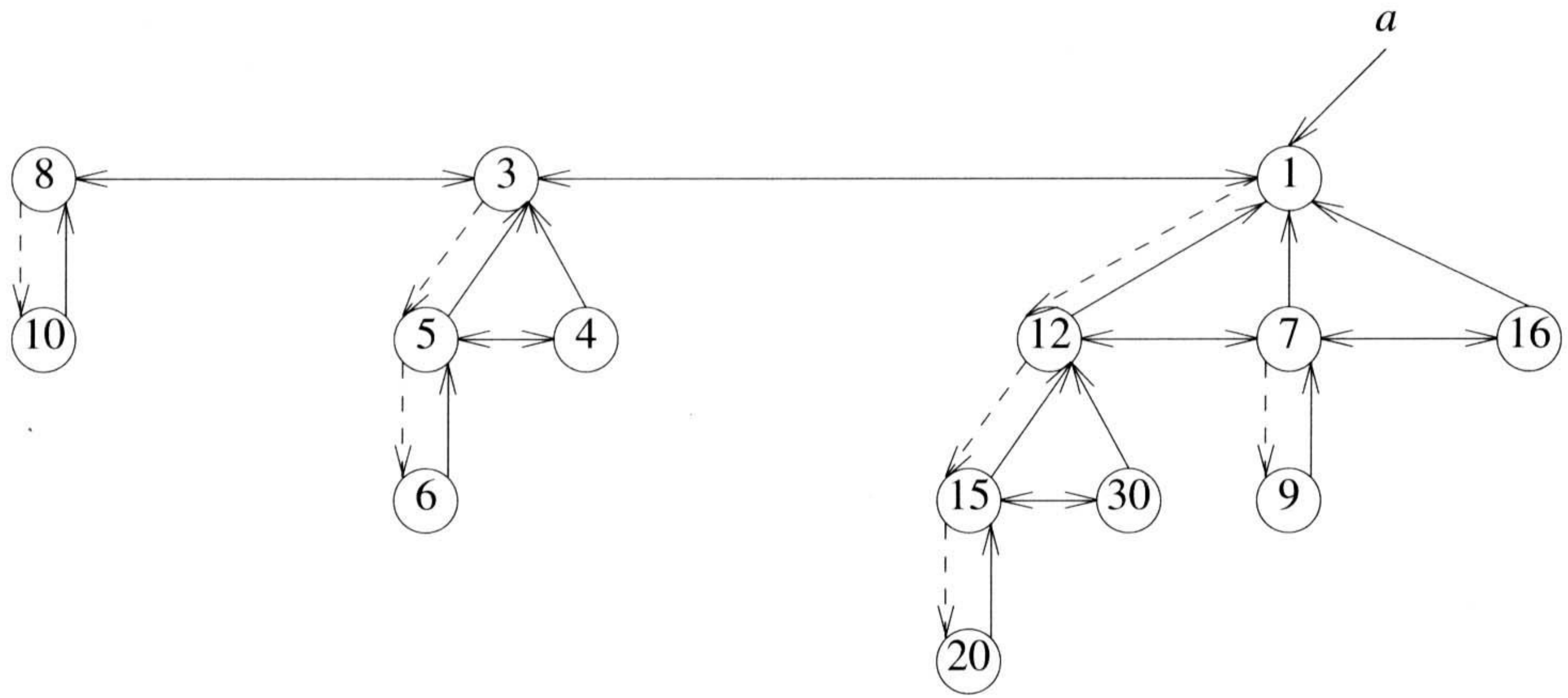


Figure 9.16: B-heap of Figure 9.15 showing parent pointers and sibling lists

- Insertion into a binomial heap
  - We insert an element, *x*, into an B-heap, *a*, by first putting *x* into a new node and placing this node into the doubly linked circular list pointed at by *a*.
  - We reset *a* to this new node only if *a* is *NULL* or *x*’s key is smaller than the key in the node pointed at by *a*.
  - It is evident that we can perform these insertion steps in O(1) time.

Insert 時，不作 combine 的動作！

- Combine
  - To combine two nonempty B-heaps *a* and *b*, we combine the top doubly linked circular list of *a* and *b* into single double linked circular list.
  - The new B-heap pointer is either *a* or *b* depending on which has the smaller key. This can be determined with a single comparison.
  - Since two doubly linked circular lists can be combined into a single one in O(1) time, a combine takes only O(1) time.

- Deletion of min element
  - Let *a* be the pointer of the B-heap from which the min element is to be deleted.
    - If *a* is *NULL*, then the B-heap is empty and a deletion cannot be performed.
    - Assume that a is not NULL.

      a points to the node that contains the min element.

      This node is deleted from its doubly linked circular list.

      The new B-heap consists of the remaining min-trees and the sub min-trees of the deleted root. (Figure 9.17 shows the situation for the example of Figure 9.15)

      Repeatedly join together pairs of min-trees that have the same degree. (The resulting min-tree collection is that of Figure 9.19)

      Form the doubly linked circular list of min-tree roots. We also reset the B-heap pointer so as to point to the min-tree root with smallest key.

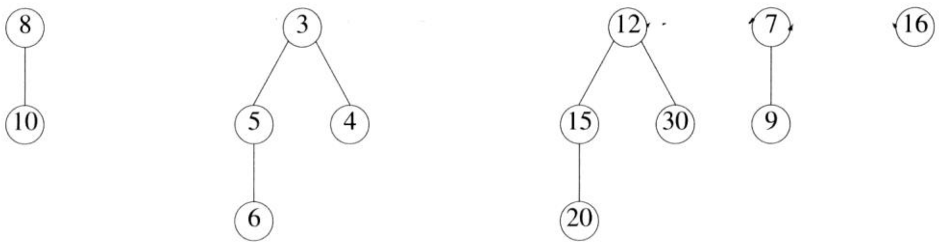


Figure 9.17: The B-heap of Figure 9.15 following the deletion of the min element

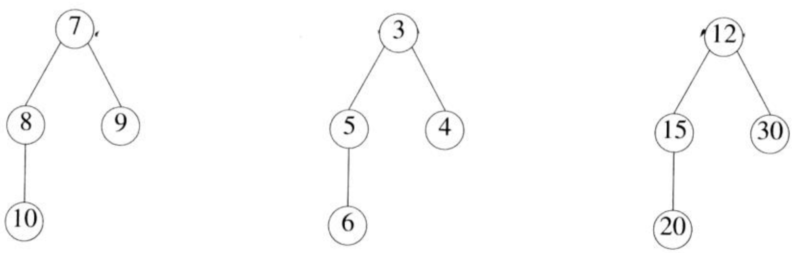


Figure 9.18: The B-heap of Figure 9.17 following the joining of the two degree one min-trees

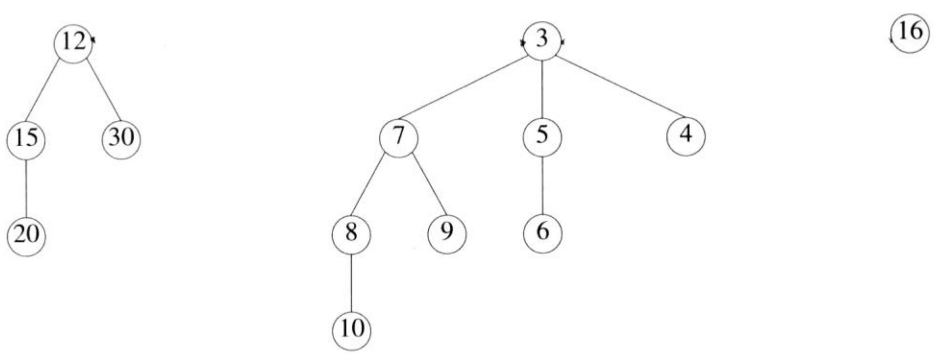


Figure 9.19: The B-heap of Figure 9.18 following the joining of two degree two min-trees

Root 的 degree:

- 是0的，有2^0個nodes (B_0)
- 是1的，有2^1個nodes (B_1)
- 是2的，有2^2個nodes (B_2)
- 是3的，有2^3個nodes (B_3)

整個 B-heap 就成為

$a_0 B_0 + a_1 B_1 + a_2 B_2 + a_3 B_3 \ldots$

– How to repeatedly join together pairs of min-trees that have the same degree?
  - The degree of a nonempty min-tree is the degree of its root.
  - This min-tree joining is done by making the min-tree whose root has larger key a subtree of the other.
  - When two min-trees are joined, the degree of the resulting min-tree is one larger than the original degree of each min-tree and the number of min-trees decreases by one.
    – Example: We may first join the min-trees with roots 8 and 7. The min-tree with root 8 is made a subtree of the min-tree with root 7. We now have the min-tree collection of Figure 9.18.

{Delete the min element from a B-heap *a*, this element is returned in *x*}

Step 1: [Handle empty B-heap] if (*a* = *NULL*) *deletion* _ *error* else perform steps 2 - 4;

Step 2: [Deletion from nonempty B-heap] *x* = *a*−>*data*; *y* = *a*−>*child*; Delete *a* from its doubly linked circular list; Now, *a* points to any remaining node in the resulting list; If there is no such node, then *a* = *NULL*;

Step 3: [Min-tree joining] Consider the min-trees in the lists *a* and *y*; Join together pairs of min-trees of the same degree until all remaining min-trees have different degree;

Step 4: [Form min-tree root list] Link the roots of the remaining min-trees (if any) together to form a doubly linked circular list; Set *a* to point to the root (if any) with minimum key;

Program 9.8: Steps in a delete min

- Analysis
  - **Definition**

    The *binomial tree*, B_k, of *degree* k is a tree such that if $k = 0$, then the tree has exactly one node and if $k > 0$, then it consists of a root whose degree is k and whose subtree are $B_0, B_1, \ldots, B_{k-1}$.

    - Example: The min-trees of Figure 9.15 are B_1, B_2, and B_3, respectly.
  - One may verify that B_k has exactly 2^k nodes.
  - Further, if we start with a collection of empty B-heaps and perform only the operations insert, combine, and delete min, then the min-trees in each B-heap are binomial trees.

– **Lemma 9.2:**

Let a be a B-heap with n elements that results from a sequence of insert, combine, and delete min operations performed on initially empty B-heaps. Each min-tree in a has degree $\leq \log_2 n$. Consequently, $MAX\_DEGREE \leq \lfloor \log_2 n \rfloor$ and the actual cost of a delete min is

$O(\log_2 n + s)$,

where s be the number of min-trees in a and y (y = a->*child*).

– **Theorem 9.1:**

If a sequence of n insert, combine, and delete min operations is performed on initial empty B-heaps, then we can amortize costs such that the amortized time complexity of each insert and combine is O(1) and that of each delete min is O(log$n$).

9.5 Fibonacci heaps

- Definition （e.g., 1, 1, 2, 3, 5, 8, 13, 21…….)
  - A Fibonacci heap is a data structure that supports the three binomial heap operations: insert, delete min, and combine as well as the operations:
    1. *delete*, delete the element in a specified node (done in O(1) amortized time)
    2. *decrease key*, decrease the key of a specified node by a given positive amount (in O(log*n*) amortized time).
  - The binomial heap operations can be performed in the same asymptotic times using a Fibonacci heap as using a binomial heap. (作動作的時間 F-heap = B-heap)

- There are two varieties of Fibonacci heaps: min and max. A *min-Fibonacci heap* is a collection of min-trees while a *max-Fibonacci heap* is a collection of max-trees.
  - We shall explicitly consider min-Fibonacci heaps only.
  - These will be referred to as *F-heaps*.
- B-heaps are a special case of F-heaps.
- To represent an F-heap, the B-heap representation is augmented by adding two fields: *parents* and *child_cut* to each node.
  - The *parents* field is used to point to the node's parent.
  - The significance of the *child_cut* field will be described later.
- The basic operations: insert, delete min, and combine are performed exactly as for the case of B-heaps.

- Deletion from an F-heap
  - To delete an arbitrary node *b* from the F-heap a, we do the following:

    If $a = b$, then do a delete min; otherwise do steps 2, 3, and 4 below.

    Delete *b* from the doubly linked list it is in.

    Combine the doubly linked list of *b*'s children with the doubly linked list of *a*'s min-tree roots to get a single doubly linked list. Trees of equal degree are not joined together as in a delete min.

    Dispose of node *b*.

- Example: if we delete the node containing 12 from the F-heap of Figure 9.15, we get the F-heap of Figure 9.20. The actual cost of an arbitrary delete is O(1) unless the min element is being deleted. In this case the deletion time is the time for a delete min operation.

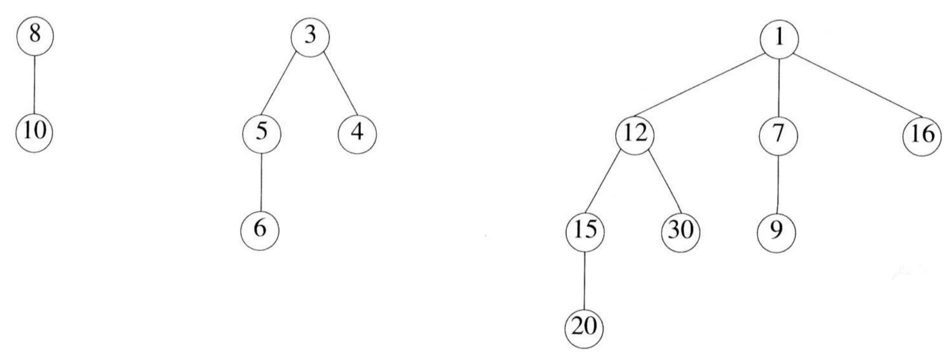


Figure 9.15: A B-heap with three min-trees

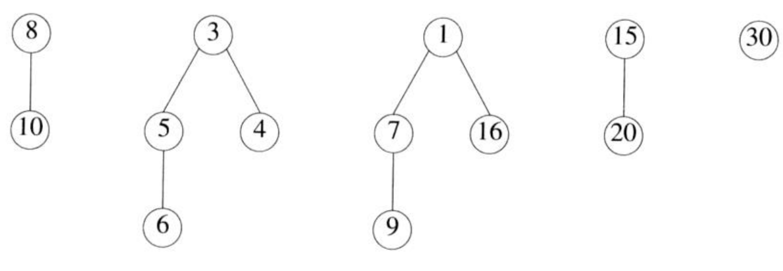


Figure 9.20: F-heap of Figure 9.15 following the deletion of 12

- Decrease key
  - To decrease the key in node *b* we do the following:

    Reduce the key in *b*.

    If *b* is not a min-tree root and its key is smaller than that in its parent, then delete *b* from its doubly linked list and insert it into the doubly linked list of min-tree roots.

    Change *a* to point to *b* in case the key in *b* is smaller than that in *a*.

- Example: suppose we decrease the key 15 in the F-heap of Figure 9.15 by 4. The resulting F-heap is shown in Figure 9.21. The cost of performing a decrease key is O(1).

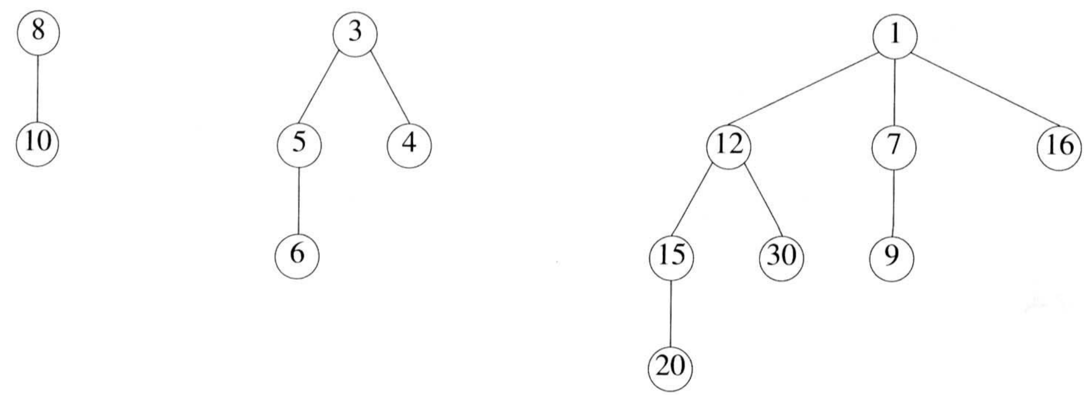


Figure 9.15: A B-heap with three min-trees

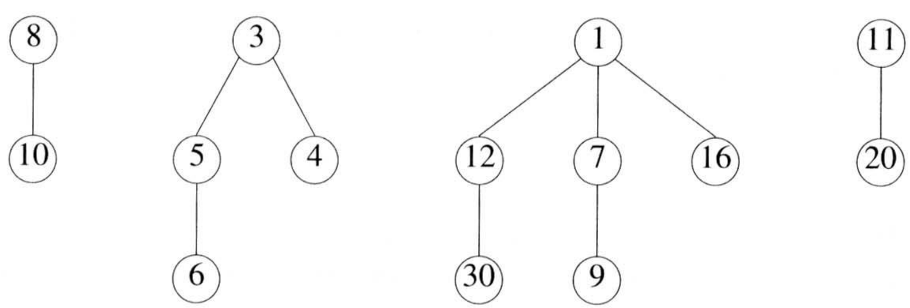


Figure 9.21: F-heap of Figure 9.15 following the reduction of 15 by 4

- If an F-heap is used to represent *S'*, the complexity of the shortest path algorithm becomes $O(n\log n + e)$. (e: the number of edges)
  - This is an asymptotic improvement over the implementation discussed in Chapter 6 if the graph does not have $\Omega(n^2)$ edges.
- If this single source algorithm is used n times, once with each of the n vertices in the graph as the source, then we can find a shortest path between every pair of vertices in $O(n^2\log n + ne)$.
  - Once again, this represents an asymptotic improvement over the $O(n^3)$ dynamic programming algorithm of Chapter 6 for graphs that do not have $\Omega(n^2)$ edges.
- It is interesting to note that $O(n\log n + e)$ is the best possible implementation of the single source algorithm of Chapter 6 as the algorithm must examine each edge and may be used to sort n numbers (which requires $O(n\log n)$ time).