CHAPTER 6

Data Structures and Algorithm Analysis in Java 3rd Edition by Mark Allen Weiss

PRIORITY QUEUES (Heaps)

- A priority queue is a data structure that allows at least the following two operations:
  - insert
  - deleteMin

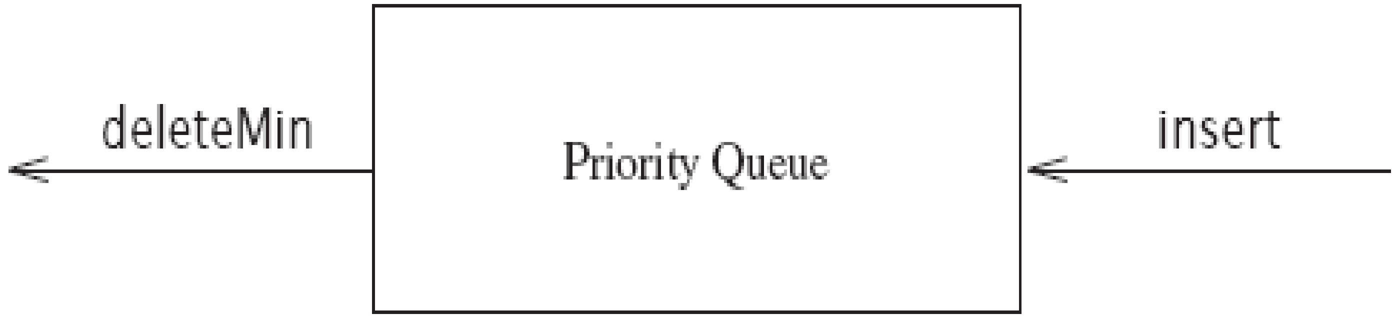


Figure 6.1 Basic model of a priority queue

IMPLEMENTATION

- Linked List
  - insertions at the front takes O(1)
  - finding and deleting minimum will take O(N) time

- Binary Search Tree
  - Both operations take O(log N) average time.

- Basic data Structure using arrays and no links
  - Both operations in O(log N) worst time.

BINARY HEAP

- The implementation is known as binary heap.

- Heaps have two properties
  - Structure Property
  - Heap Order Property

STRUCTURE PROPERTY

- A heap is a binary tree that is completely filled with the possible exception of the bottom level, which is filled from left to right. Such a tree is known as **complete binary tree**.

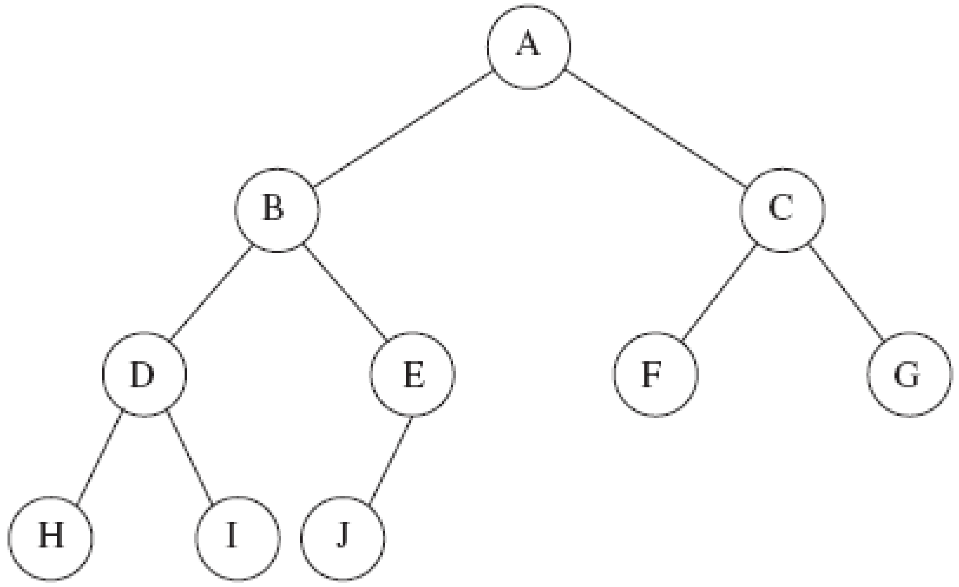
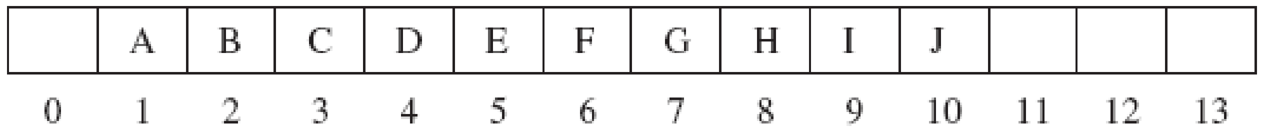


Figure 6.2 A complete binary tree

STRUCTURE PROPERTY

- Complete binary tree of height h has between 2^h and 2^{h+1} -1
- Height of the binary tree is O(log N)
- It can be represented in an array and no links are necessary.
- For any element in array position i, the left child is in position 2i, the right child is in 2i+1 and parent is in i/2.



	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figure 6.3 Array implementation of complete binary tree

Heap-Order Property

Smallest element should be at the root, any node should be smaller than all of its descendants.

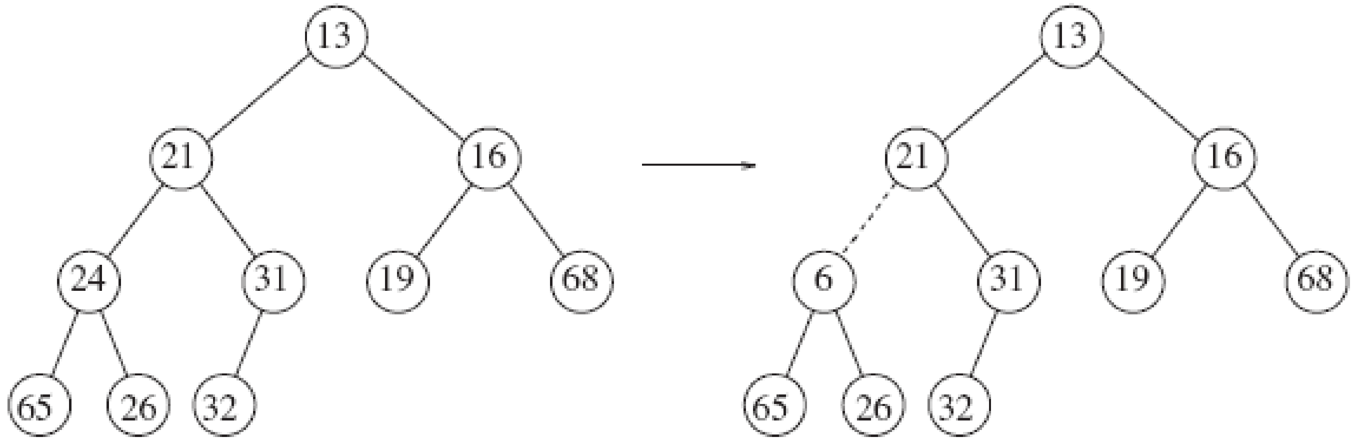


Figure 6.5 Two complete trees (only the left tree is a heap)

```
public class BinaryHeap<AnyType extends Comparable<? super AnyTy
{
    public BinaryHeap( )
      { /* See online code */ }
    public BinaryHeap( int capacity )
      { /* See online code */ }
    public BinaryHeap( AnyType [ ] items )
      { /* Figure 6.14 */ }

    public void insert( AnyType x )
      { /* Figure 6.8 */ }
    public AnyType findMin( )
      { /* See online code */ }
    public AnyType deleteMin( )
      { /* Figure 6.12 */ }
    public boolean isEmpty( )
      { /* See online code */ }
    public void makeEmpty( )
      { /* See online code */ }

    private static final int DEFAULT_CAPACITY = 10;

    private int currentSize;      // Number of elements in heap
    private AnyType [ ] array;    // The heap array

    private void percolateDown( int hole )
      { /* Figure 6.12 */ }
    private void buildHeap( )
      { /* Figure 6.14 */ }
    private void enlargeArray( int newSize )
      { /* See online code */ }
}
```

Figure 6.4 Class skeleton for priority queue

insert : Operation

- To insert element X into the heap we create hole in the next available position.
  - if X can be placed in the hole without violating heap order, then we do so.
  - Else slide the element that is in the hole’s parent node into the hole. (bubbling the hole up towards the root)
  - Continue this until X can be placed in the hole.
  - This general strategy is known as **percolate up**

EXAMPLE: INSERT 14

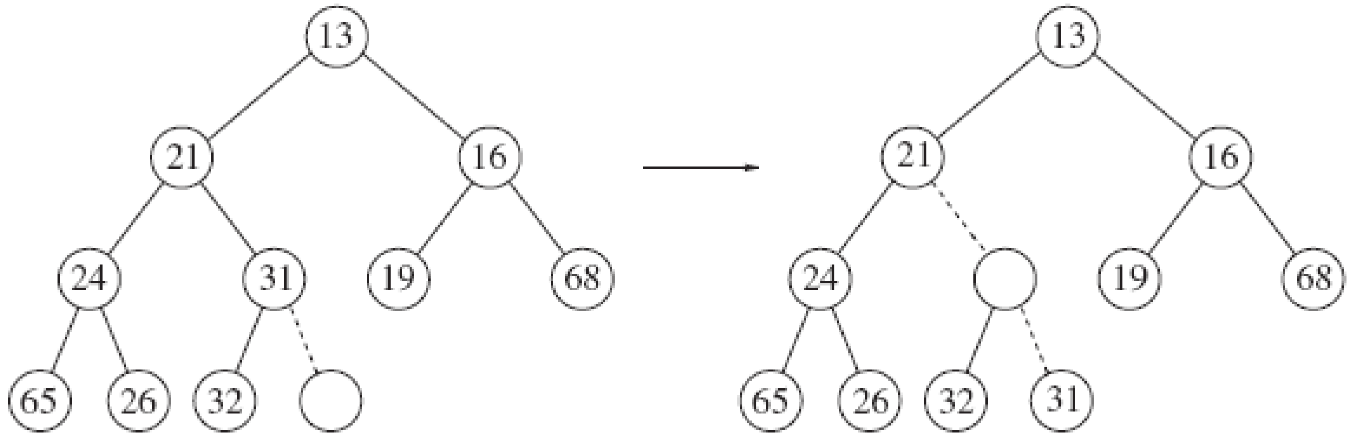


Figure 6.6 Attempt to insert 14: creating the hole and bubbling the hole up

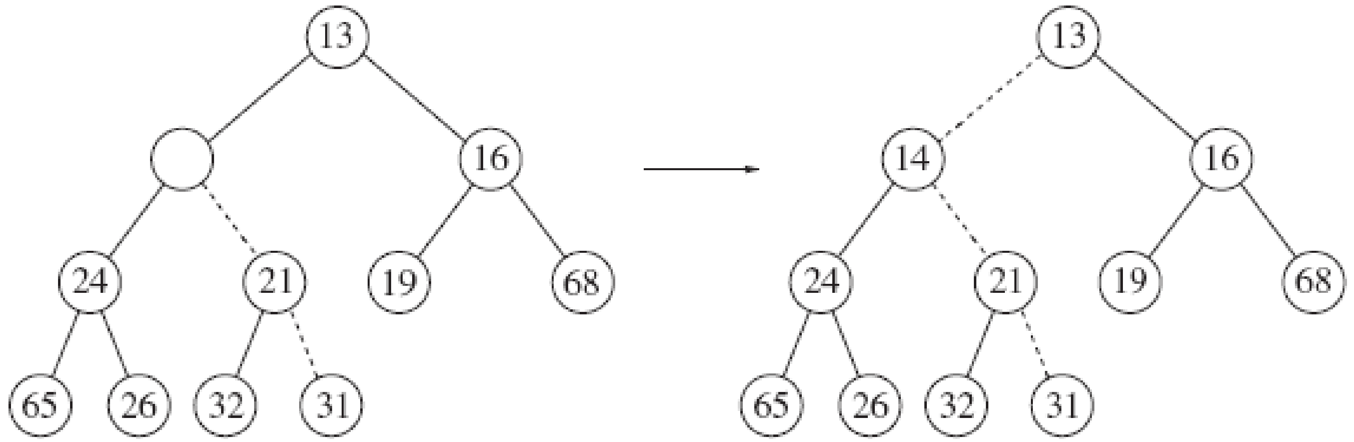


Figure 6.7 The remaining two steps to insert 14 in previous heap

insert : Operation

```java
/**
 * Insert into the priority queue, maintaining heap order.
 * Duplicates are allowed.
 * @param x the item to insert.
 */
public void insert( AnyType x )
{
    if( currentSize == array.length - 1 )
        enlargeArray( array.length * 2 + 1 );

        // Percolate up
    int hole = ++currentSize;
    for( array[ 0 ] = x; x.compareTo( array[ hole / 2 ] ) < 0; hole /= 2 )
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;
}
```

Figure 6.8 Procedure to insert into a binary heap

deleteMin: Operation

- Finding the minimum is easy, removing it is hard.
- Heap now becomes one smaller, so the last element X in the heap must move somewhere in the heap .

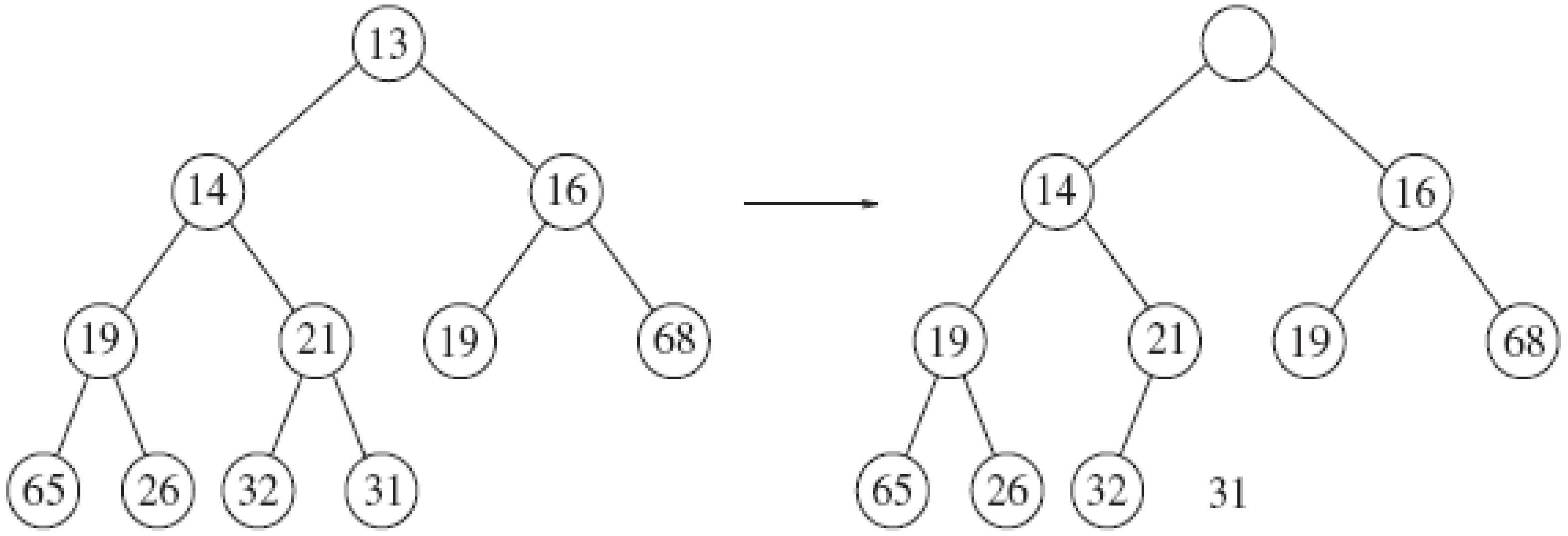


Figure 6.9 Creation of the hole at the root

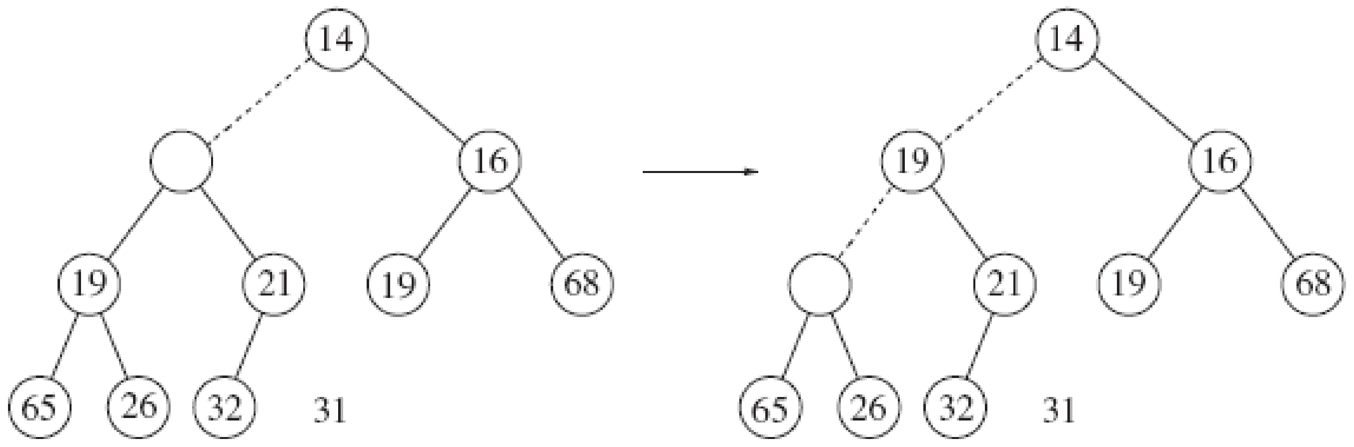


Figure 6.10 Next two steps in `deleteMin`

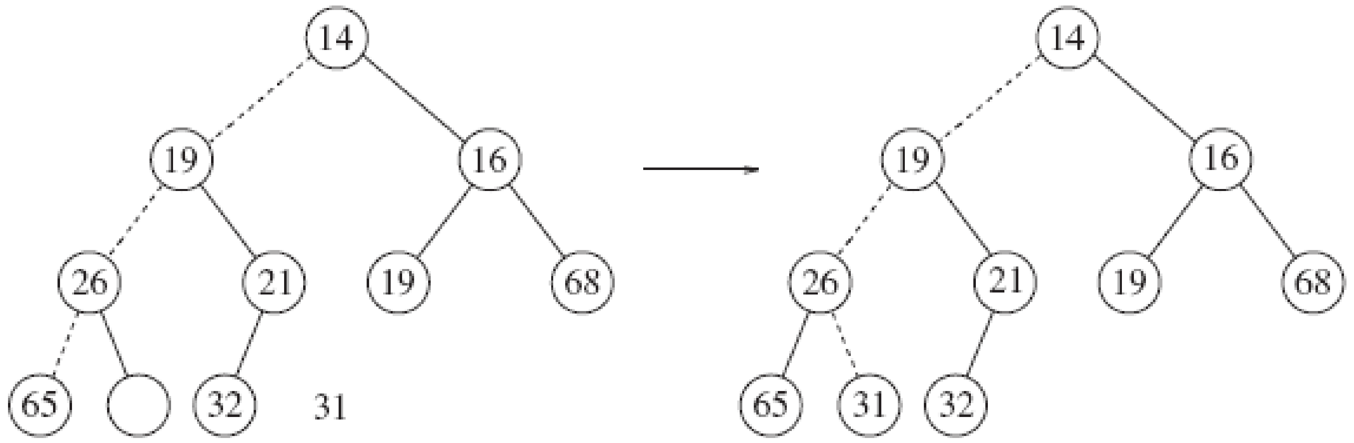


Figure 6.11 Last two steps in `deleteMin`

```
    /**
     * Remove the smallest item from the priority queue.
     * @return the smallest item, or throw UnderflowException, if empty.
     */
    public AnyType deleteMin( )
    {
        if( isEmpty( ) )
            throw new UnderflowException( );

        AnyType minItem = findMin( );
        array[ 1 ] = array[ currentSize-- ];
        percolateDown( 1 );

        return minItem;
    }

    /**
     * Internal method to percolate down in the heap.
     * @param hole the index at which the percolate begins.
     */
    private void percolateDown( int hole )
    {
        int child;
        AnyType tmp = array[ hole ];

        for( ; hole * 2 <= currentSize; hole = child )
        {
            child = hole * 2;
            if( child != currentSize &&
                    array[ child + 1 ].compareTo( array[ child ] ) < 0 )
                child++;
            if( array[ child ].compareTo( tmp ) < 0 )
                array[ hole ] = array[ child ];
            else
                break;
        }
        array[ hole ] = tmp;
    }
```

Figure 6.12 Method to perform deleteMin in a binary heap

OTHER HEAP OPEATIONS

- A min heap is used to search for minimum element it has no information on the maximum element, it can be in any of the leaf nodes. In a very large binary tree, this is hard to find.

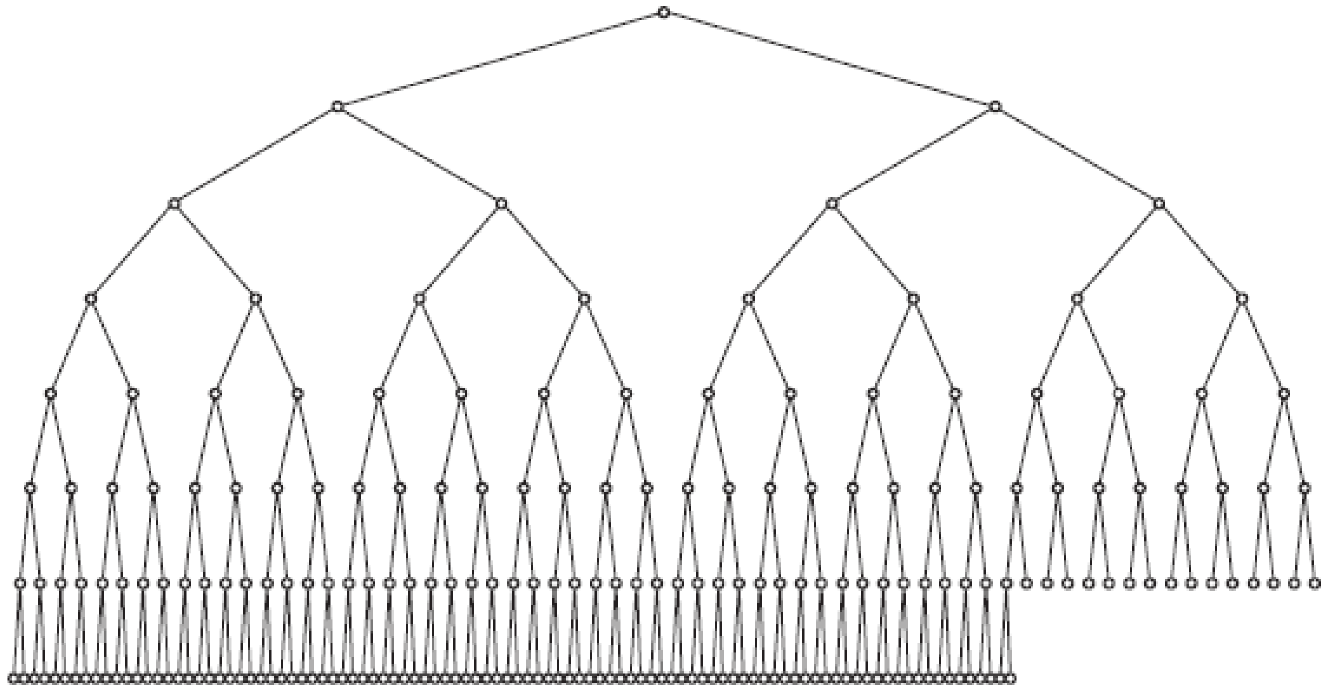


Figure 6.13 A very large complete binary tree

OTHER HEAP OPEATIONS

- decreaseKey:
  - decreaseKey(p, Δ) operation lowers the value of the item at position p by a positive amount Δ.
  - Heap order must be fixed by percolate up.
- increaseKey:
  - increaseKey(p, Δ) operation increases the value of the item at position p by a positive amount Δ.
  - Heap order must be fixed by percolate down.
- delete:
  - delete(p) operation removes the node st position p from heap.
  - Done by first performing decreaseKey(p, ∞) and then performing deleteMin().

buildHeap

- Binary heap is sometimes constructed from an initial collection of items.
- Running time for this algorithm is O(N) average and O(NlogN) worst.
- The algorithm is to place N items into the tree in any order maintaining the structure property. Then , if percolateDown(i) percolates down from the node i, for i from the last parent to the root.

Example

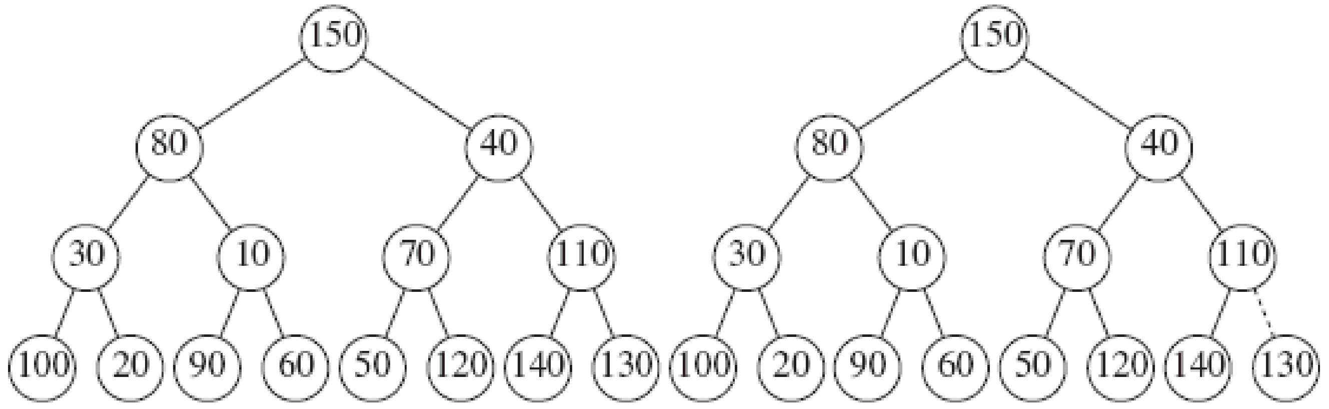


Figure 6.15 Left: initial heap; right: after `percolateDown(7)`

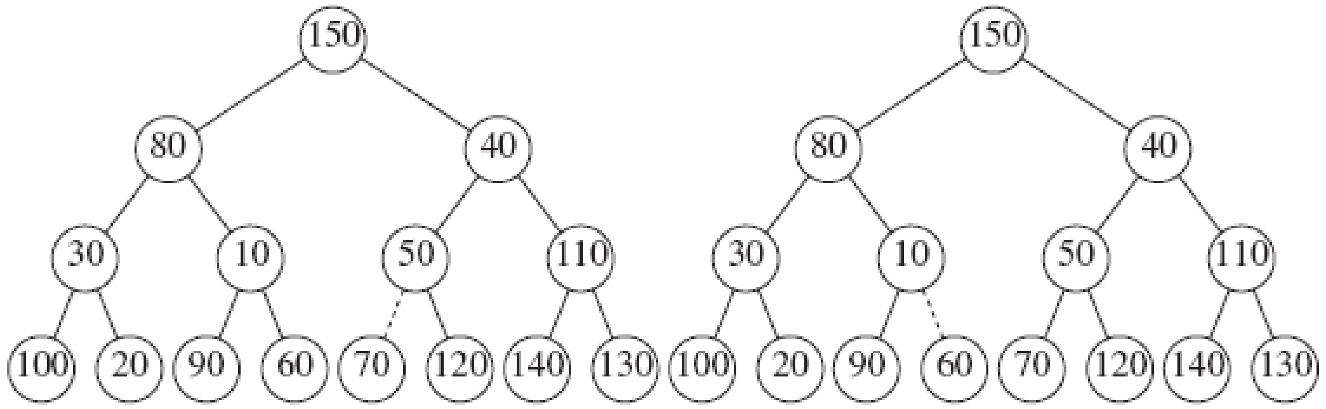


Figure 6.16 Left: after `percolateDown(6)`; right: after `percolateDown(5)`

Example

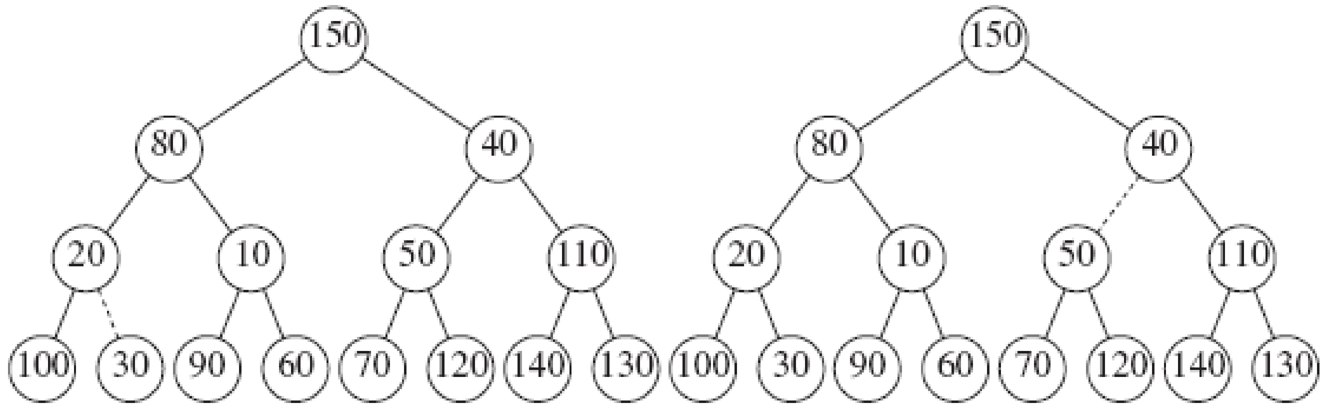


Figure 6.17 Left: after `percolateDown(4)`; right: after `percolateDown(3)`

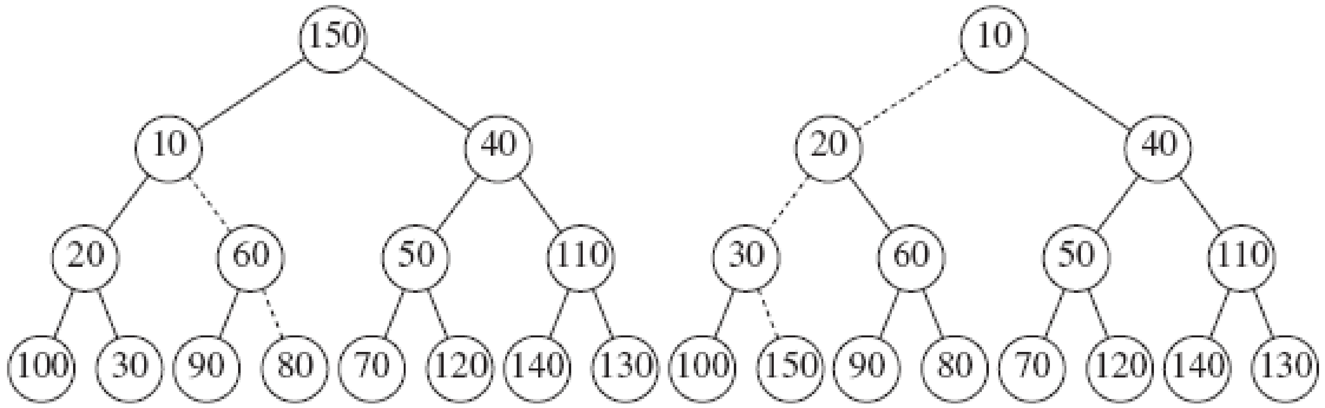
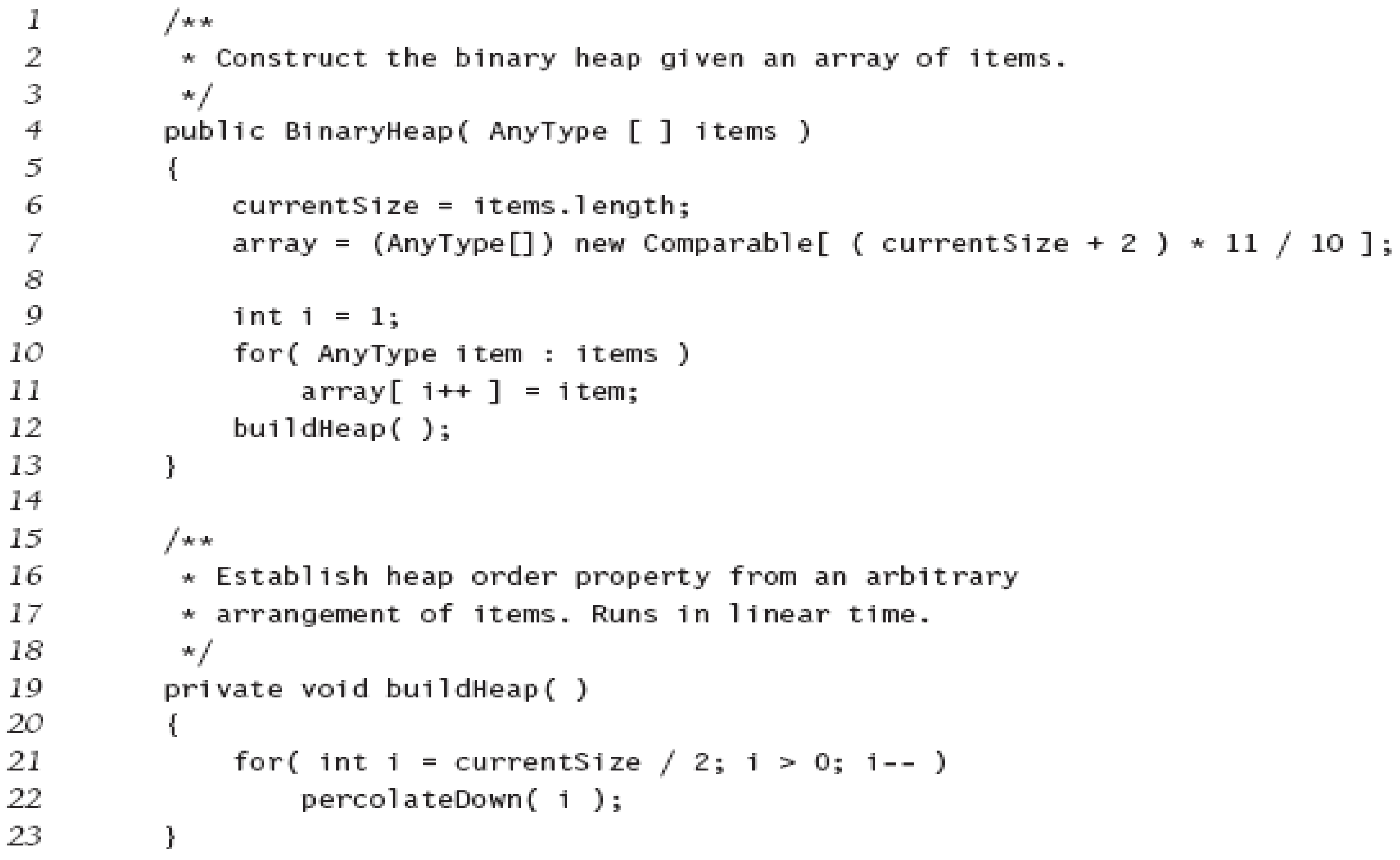


Figure 6.18 Left: after `percolateDown(2)`; right: after `percolateDown(1)`

buildHeap



```
/**
 * Construct the binary heap given an array of items.
 */
public BinaryHeap( AnyType [ ] items )
{
    currentSize = items.length;
    array = (AnyType[]) new Comparable[ ( currentSize + 2 ) * 11 / 10 ];

    int i = 1;
    for( AnyType item : items )
        array[ i++ ] = item;
    buildHeap( );
}

/**
 * Establish heap order property from an arbitrary
 * arrangement of items. Runs in linear time.
 */
private void buildHeap( )
{
    for( int i = currentSize / 2; i > 0; i-- )
        percolateDown( i );
}
```

Figure 6.14 Sketch of buildHeap

RUNNING TIME ANALYSIS

- The running time of the algorithm is bounded by number of comparisons (dashed lines). Which is bounded by the sum of heights of the all the nodes in the heap
- Theorem: For the perfect binary tree of height h containing 2^{h+1} -1 nodes , the sum of heights of the nodes is 2^{h+1} -1 – (h+1)

S = $\sum_{i=0}^{h} 2^{i}(h-i)$

Cont.

$S = h + 2(h-1) + 4(h-2) + 8(h-3) + 16(h-4) + \ldots + 2^{h-1}$

Find 2S

S= 2S –S

$S = -h + 2 + 4 + 8 + \ldots. + 2^{h-1} + 2^{h} \quad = (2^{h+1} - 1) - (h+1)$

which is O(N)

- Complete binary tree will have between 2^{h} and 2^{h+1} nodes. This theorem implies that this sum is O(N) where N is the number of nodes.

Applications Of Priority Queues

- The priority queues are used in operating system design and several graph algorithms implementation.

- Two other applications are
  - The Selection Problem
  - Event Simulation

The Selection Problem

- To find the k th largest element in a list of N elements

- Solutions without priority queue
  - Algorithm 1: Sort the entire array and return the k th index O(N^2)

  - Algorithm 2:
    - Read k elements into a new array and sort it
    - Process the other elements one by one
    - Read a new element compare it with the k th element , if it is larger than k, then k th element is removed and the new element is placed in the appropriate array position.
    - The running time is O(N.k)

Algorithm with Priority Queues

- Lets assume we are finding k th smallest element
- Read N elements into the array
- buildheap
- perform k deleteMin operations. The last element extracted from the heap is the answer.
- The running time is O(N+klogN) if k = N/2 then O(NlogN)

Algorithm with Priority Queues

- At any point in time we will maintain a set S of the k largest elements.
- After first k elements are read, when a new element is read it is compared with kth largest element.
- The first k elements are placed into the heap in total time O(k) with a call to buildHeap.
- The time to process each of the remaining elements is O(1) to test if element goes into S , plus O(log k) to delete and insert the new element, if this is necessary

Event Simulation

- In any kind of system (such as a bank) customers arrive wait in line until one of the k resources (tellers) is available. We can use computer models to simulate the operation of the system.
- A simulation consist of processing events
- One way to do this simulation is to start a simulation clock at zero ticks. We then advance the clock one tick at a time, checking to see if there is an event. When there are no customers left in lone and all tellers are free. The simulation is over.
- Another way is not to use clock ticks and use events to advance from one stage to another. Like
  - the next customer in the input line arrives
  - A customer at a teller leaves.

d-Heaps

- A simple generalization of binary heap is a d-heap, which is exactly like a binary heap except that all nodes have d children.

- Running time of insert $\rightarrow$ $O(\log_d N)$
- However, for large d, the *deleteMin* operation is more expensive, because even though the tree is shallower, the minimum of d children must be found which takes d-1 comparisons.
- d-heap can be advantageous in same way as B-trees

d-Heaps

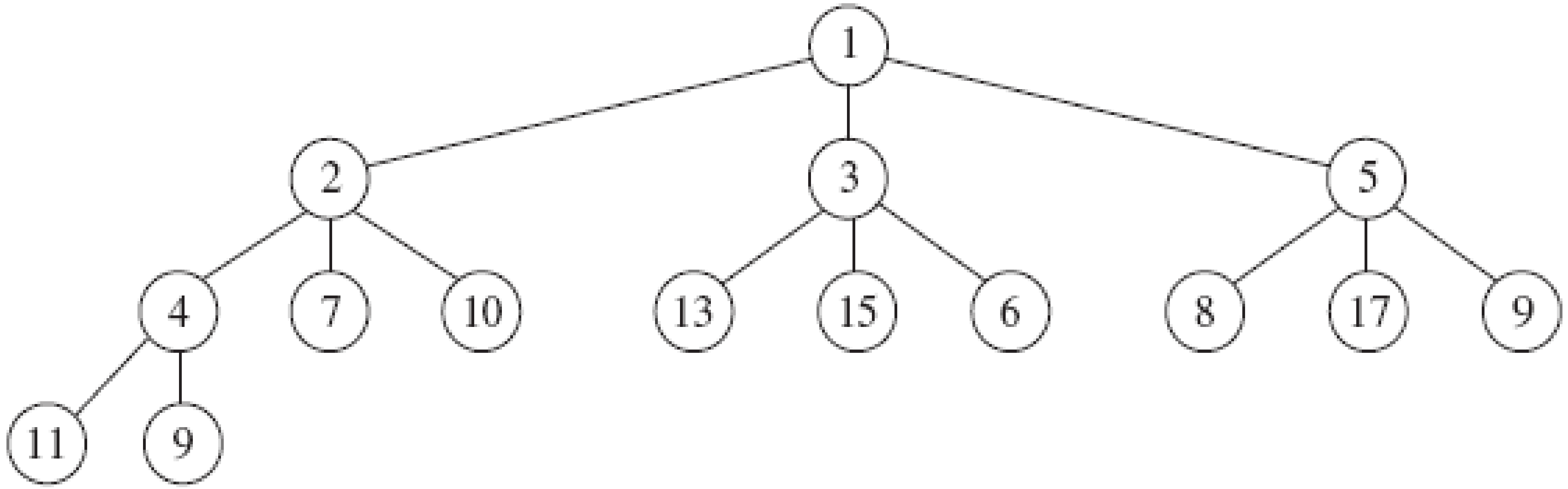


Figure 6.19 A *d*-heap

Leftist Heap

- This is a data structure that supports efficient merging, and requires the use of linked data structure.
- Leftist heap has both a structural property and an ordering property.

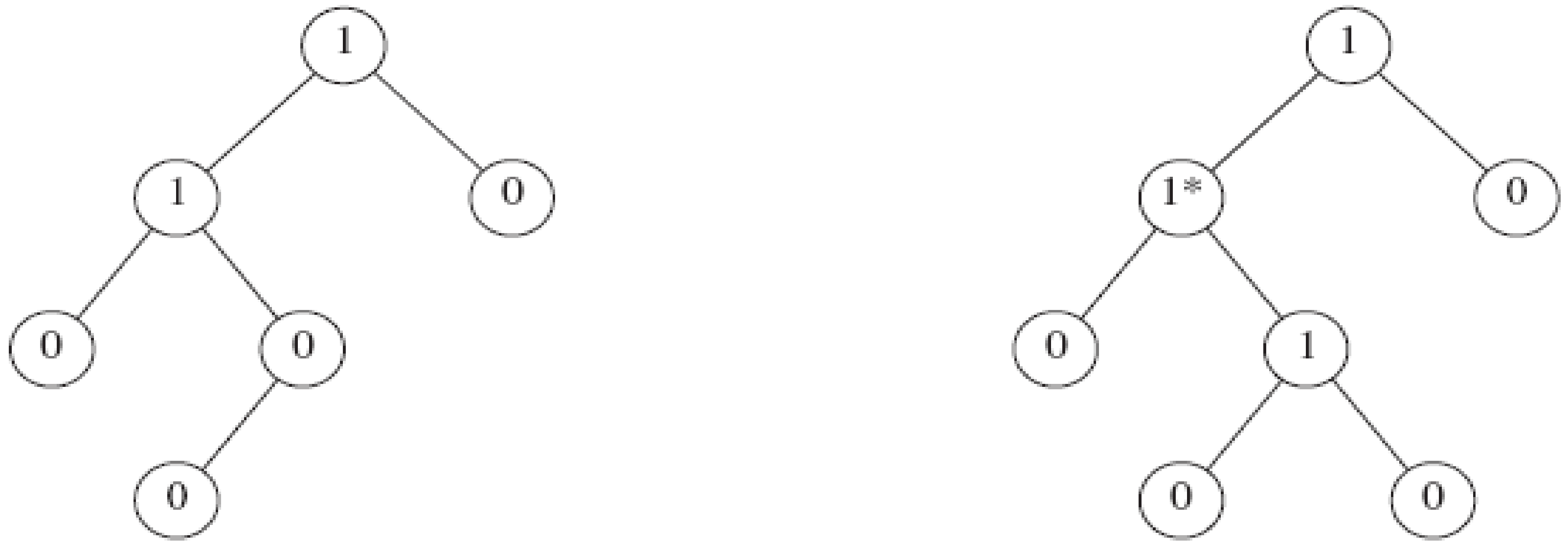


Figure 6.20 Null path lengths for two trees; only the left tree is leftist

Leftist Heap Property

- *npl*(X) - Null path length of any node X to be the length of the shortest path from X to a node without two children. *npl* of a node with 0 or 1 child is o and *npl*(null) = -1.
- *npl*(X) = Min(*npl*(X.left), *npl*(X.right)) +1
- The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.

Leftist Heap Property

- Theorem: A leftist tree with r nodes on the right path must have at least 2^r – 1 nodes.
- prove by induction if r =1 then it will have at least one tree node.
- Assume it is true for r = k has at least 2^k – 1 nodes.
- if r = k + 1 then its right subtree will have at most k nodes so whole right tree has at least 2^k – 1 nodes and the left tree has at least 2^k – 1 nodes. Combining both and 1 for root we have 2^{k+1} – 1. Hence proved
- From this theorem, it follows immediately that a leftist tree of N nodes has a right path containing at most $\lfloor \log(N+1) \rfloor$

Leftist Heap Operations

- Merging :
- Each node has data fields for the element, left and right references and an entry to indicate null path length.
- We merge two leftist heaps H_1 and H_2,
  - if either one is empty return the other
  - Otherwise compare their roots, first merge the heap with large root with the right subheap of the heap with smaller root, then set its right subheap as the newly formed heap.
  - Make sure the newly formed heap is a leftist if not swap its right and left children

Leftist Heap Operations

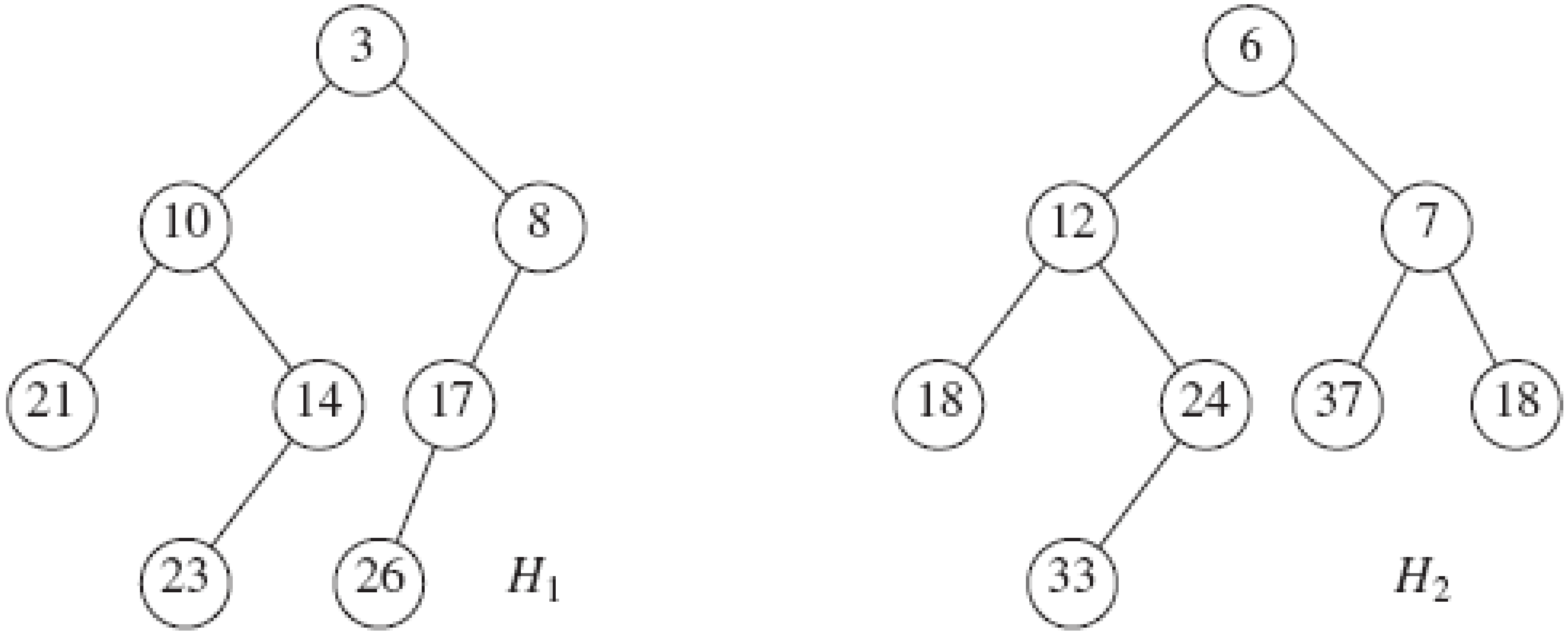


Figure 6.21 Two leftist heaps H_1 and H_2

Leftist Heap Operations

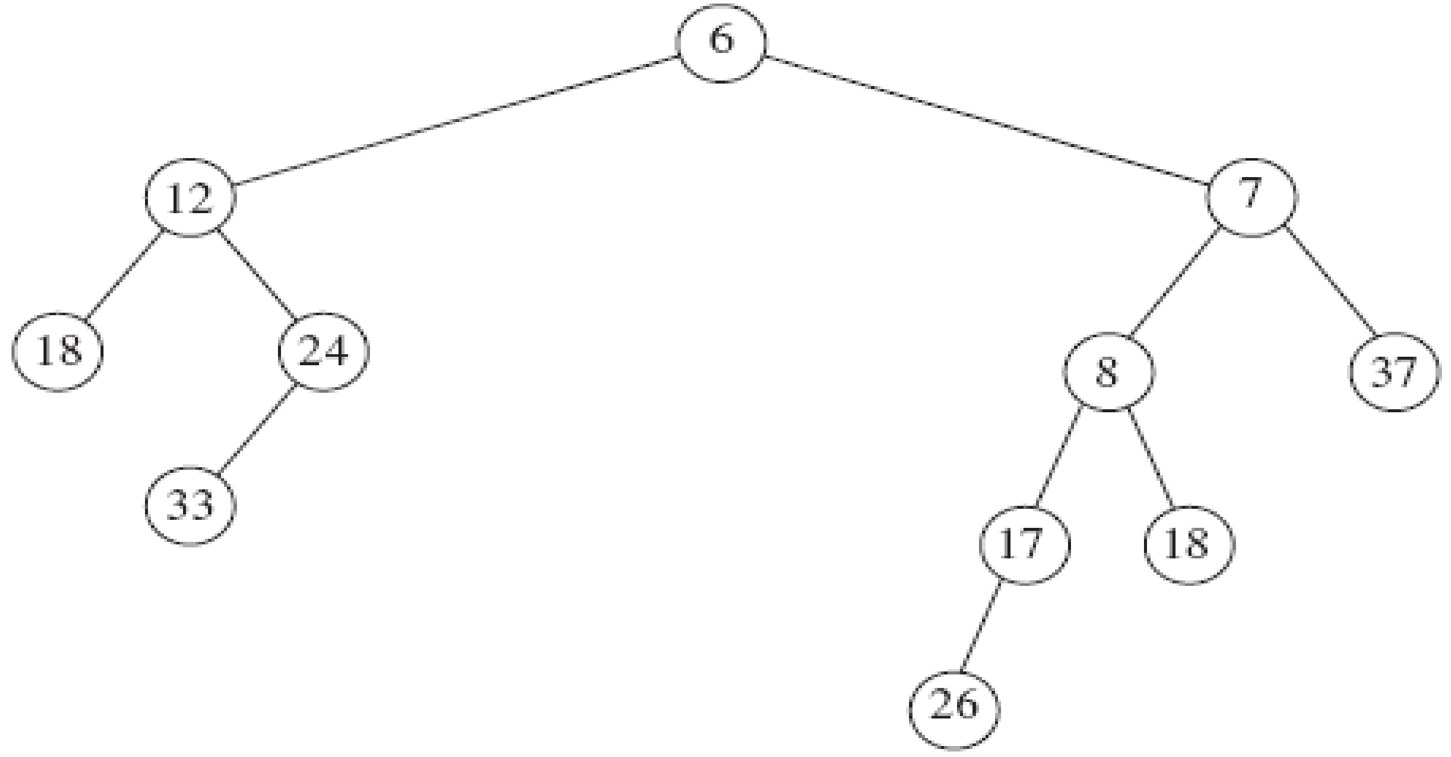


Figure 6.22 Result of merging H_2 with H_1's right subheap

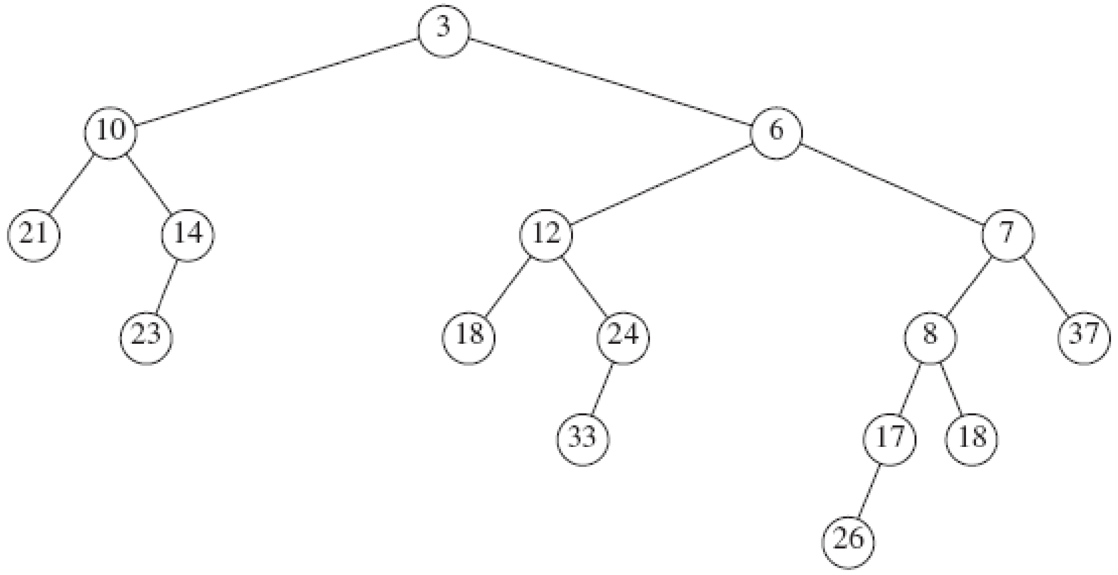


Figure 6.23 Result of attaching leftist heap of previous figure as H_1's right child

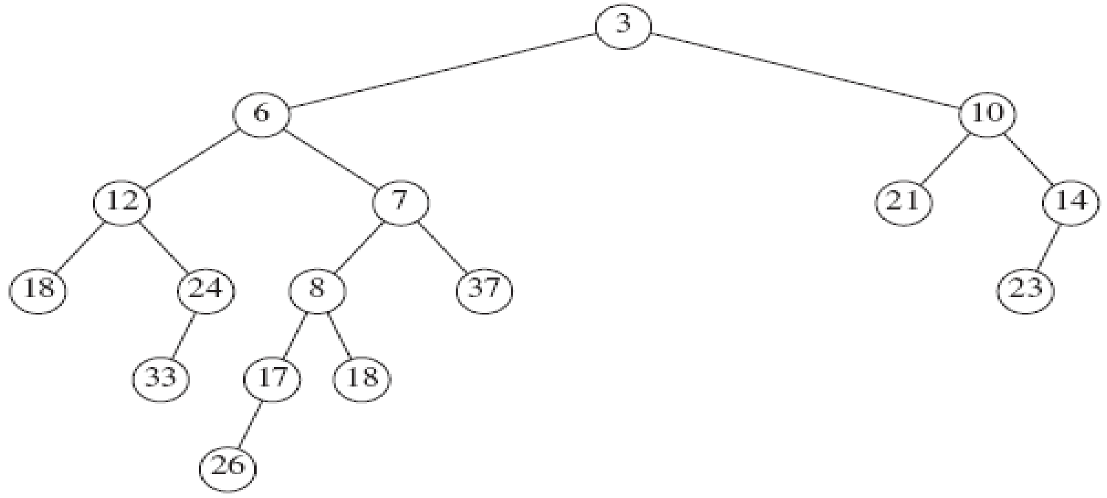


Figure 6.24 Result of swapping children of H_1's root

```java
public class LeftistHeap<AnyType extends Comparable<? super AnyType>>
{
    public LeftistHeap( )
      { root = null; }

    public void merge( LeftistHeap<AnyType> rhs )
      { /* Figure 6.26 */ }
    public void insert( AnyType x )
      { /* Figure 6.29 */ }
    public AnyType findMin( )
      { /* See online code */ }
    public AnyType deleteMin( )
      { /* Figure 6.30 */ }

    public boolean isEmpty( )
      { return root == null; }
    public void makeEmpty( )
      { root = null; }

    private static class Node<AnyType>
    {
            // Constructors
        Node( AnyType theElement )
          { this( theElement, null, null ); }

        Node( AnyType theElement, Node<AnyType> lt, Node<AnyType> rt )
          { element = theElement; left = lt; right = rt; npl = 0; }

        AnyType        element;      // The data in the node
        Node<AnyType> left;          // Left child
        Node<AnyType> right;         // Right child
        int            npl;          // null path length
    }

    private Node<AnyType> root;    // root

    private Node<AnyType> merge( Node<AnyType> h1, Node<AnyType> h2 )
      { /* Figure 6.26 */ }
    private Node<AnyType> merge1( Node<AnyType> h1, Node<AnyType> h2 )
      { /* Figure 6.27 */ }
    private void swapChildren( Node<AnyType> t )
      { /* See online code */ }
}
```

Figure 6.25 Leftist heap type declarations

```
/**
 * Merge rhs into the priority queue.
 * rhs becomes empty. rhs must be different from this.
 * @param rhs the other leftist heap.
 */
public void merge( LeftistHeap<AnyType> rhs )
{
    if( this == rhs )    // Avoid aliasing problems
        return;

    root = merge( root, rhs.root );
    rhs.root = null;
}

/**
 * Internal method to merge two roots.
 * Deals with deviant cases and calls recursive merge1.
 */
private Node<AnyType> merge( Node<AnyType> h1, Node<AnyType> h2 )
{
    if( h1 == null )
        return h2;
    if( h2 == null )
        return h1;
    if( h1.element.compareTo( h2.element ) < 0 )
        return merge1( h1, h2 );
    else
        return merge1( h2, h1 );
}
```

Figure 6.26 Driving routines for merging leftist heaps

```
    /**
     * Internal method to merge two roots.
     * Assumes trees are not empty, and h1's root contains smallest item.
     */
    private Node<AnyType> merge1( Node<AnyType> h1, Node<AnyType> h2 )
    {
        if( h1.left == null )   // Single node
            h1.left = h2;       // Other fields in h1 already accurate
        else
        {
            h1.right = merge( h1.right, h2 );
            if( h1.left.npl < h1.right.npl )
                swapChildren( h1 );
            h1.npl = h1.right.npl + 1;
        }
        return h1;
    }
```

Figure 6.27 Actual routine to merge leftist heaps

```
    /**
     * Insert into the priority queue, maintaining heap order.
     * @param x the item to insert.
     */
    public void insert( AnyType x )
    {
        root = merge( new Node<>( x ), root );
    }
```

Figure 6.29 Insertion routine for leftist heaps

```
    /**
     * Remove the smallest item from the priority queue.
     * @return the smallest item, or throw UnderflowException if empty.
     */
    public AnyType deleteMin( )
    {
        if( isEmpty( ) )
            throw new UnderflowException( );

        AnyType minItem = root.element;
        root = merge( root.left, root.right );

        return minItem;
    }
```

Figure 6.30 `deleteMin` routine for leftist heaps

Non-Recursive version

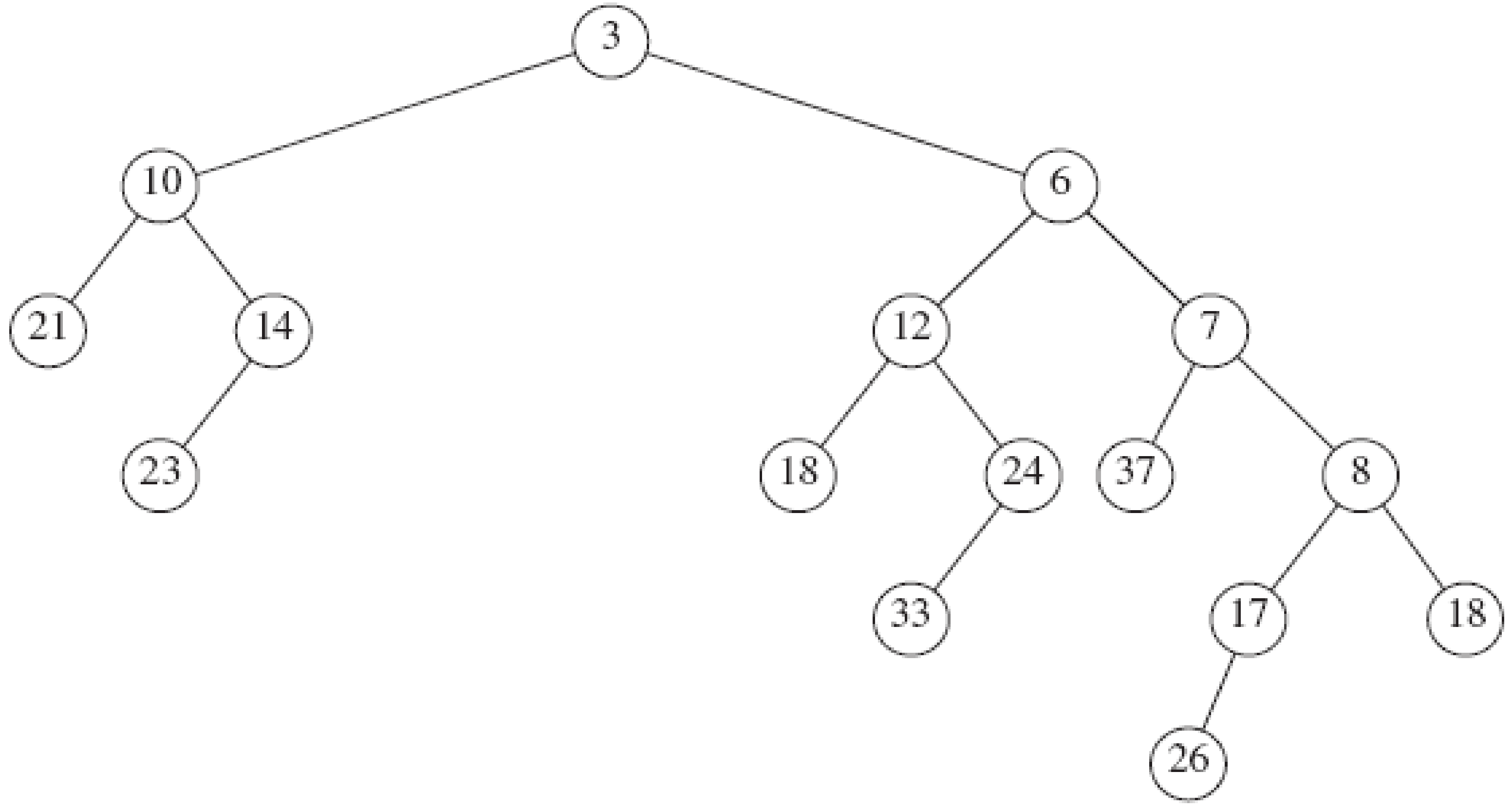


Figure 6.28 Result of merging right paths of H_1 and H_2

Skew Heaps

- A skew heap is a self adjusting version of leftist heap, except it maintains no information about null path length of any node.
- The worst-case running time is O(N) but the amortized running time O(MlogN) for M consecutive operations.
- For skew heap the swap is unconditional, we always do it, with one exception the largest of all nodes on the right paths does not have its children swapped.

Skew Heaps

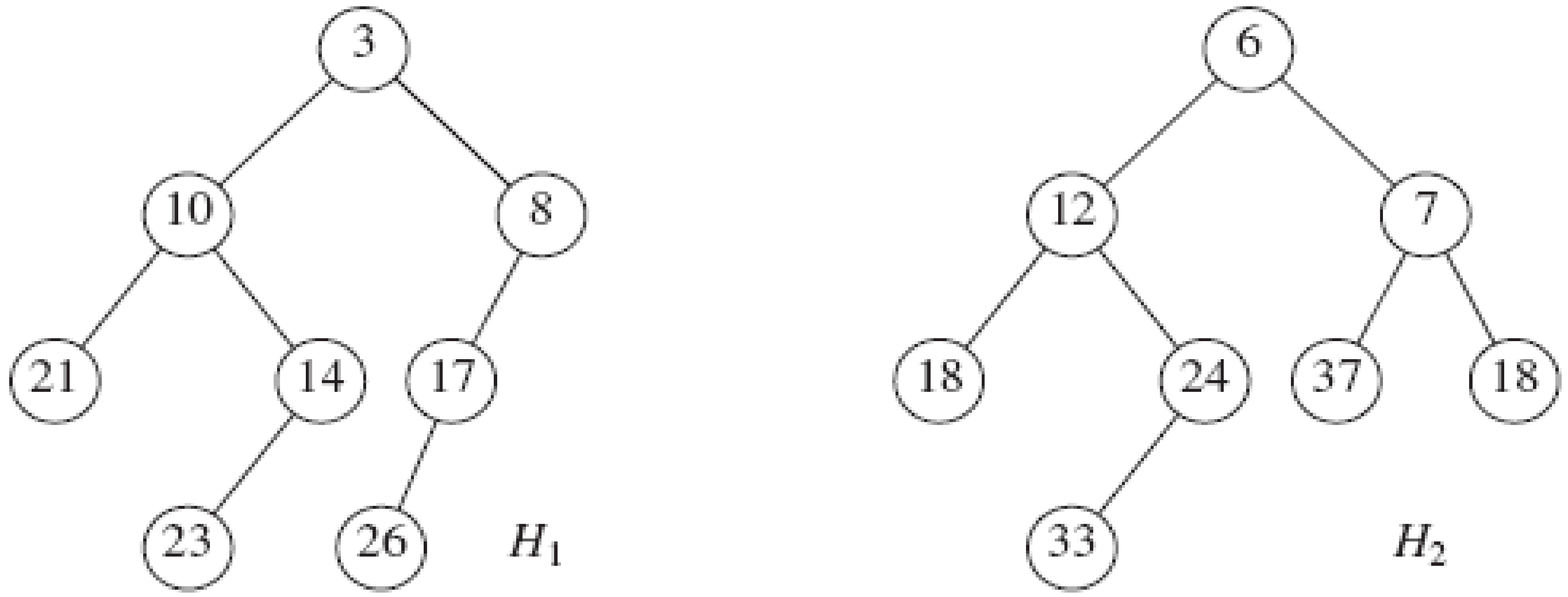


Figure 6.31 Two skew heaps H_1 and H_2

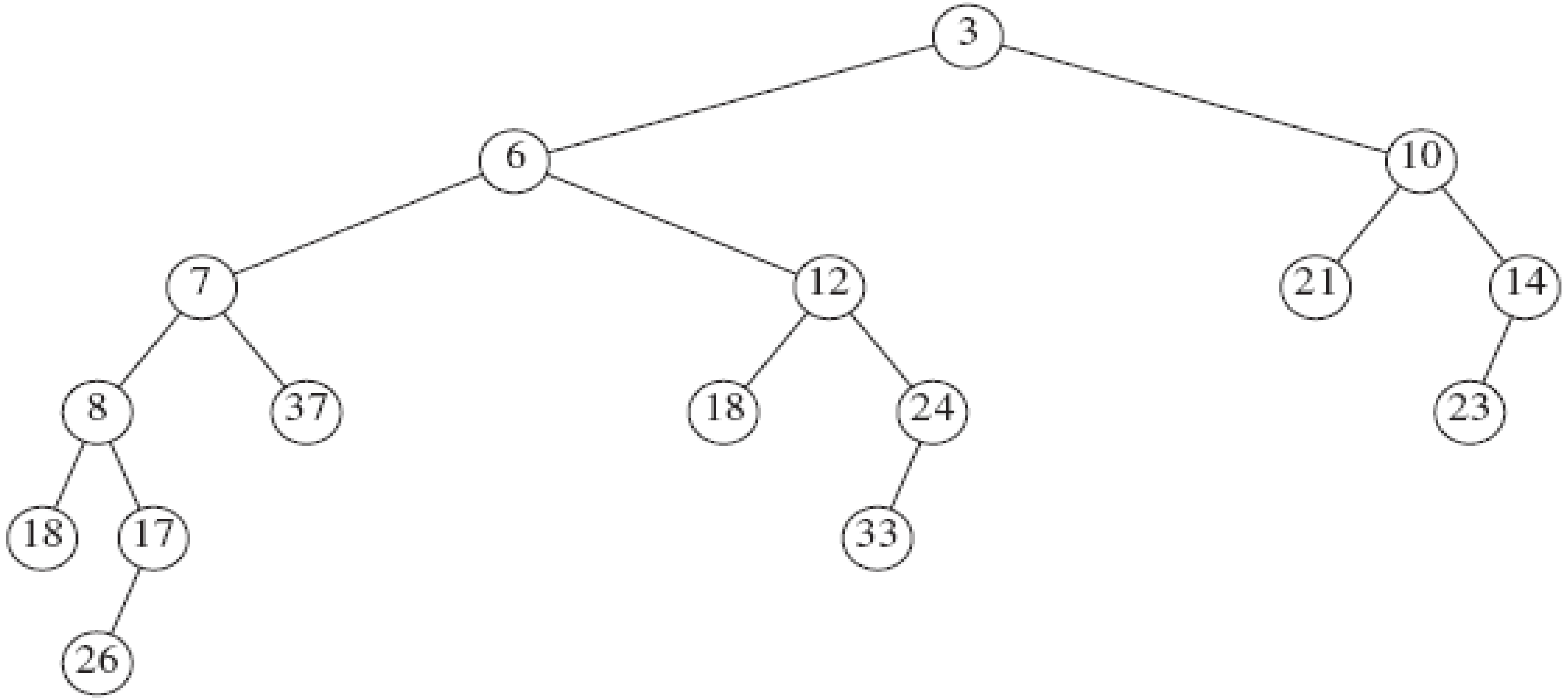


Figure 6.33 Result of merging skew heaps H_1 and H_2

Binomial Queues

- They are a collection of heap ordered trees, known as a forest
- Each heap ordered tree is a constrained form known as a binomial tree.
- There is at most one binomial tree of every height
- Insertion, deleteMin and merging is supported in O(logN) worst case and insertion is constant time on average.
- A binomial tree of height 0 is a one-node tree
- Binomial tree B_k, of height is formed by attaching a binomial tree B_{k-1}, to the root of another binomial tree B_{k-1}

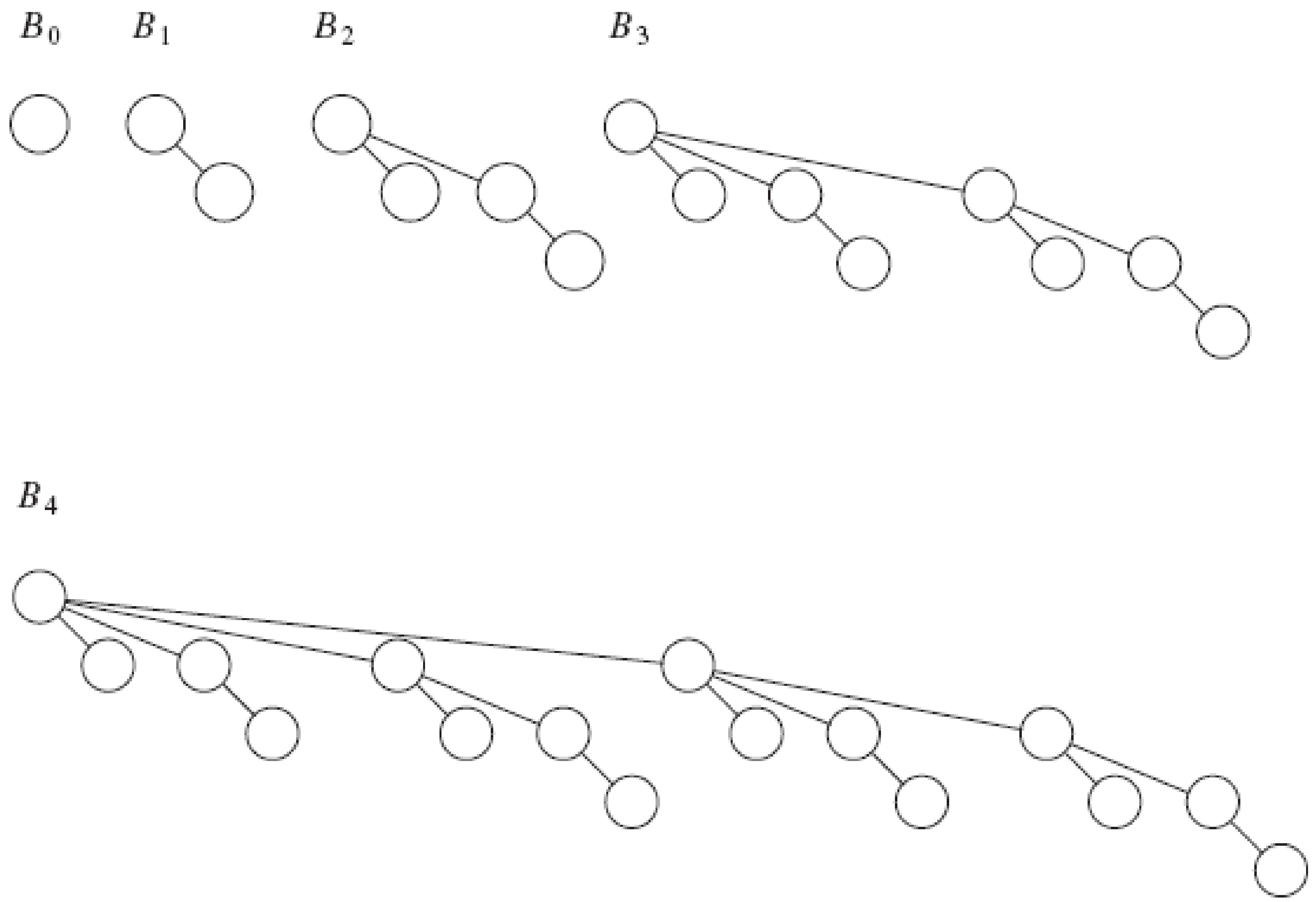


Figure 6.34 Binomial trees B_0, B_1, B_2, B_3, and B_4

Binomial Queue

- Binomial trees of height k have exactly 2^k
- We can represent a priority queue of any size by a collection of binomial trees. For example, a priority queue of size 13 could be represented by the forest B_3, B_2, B_0. which is represented as 1101

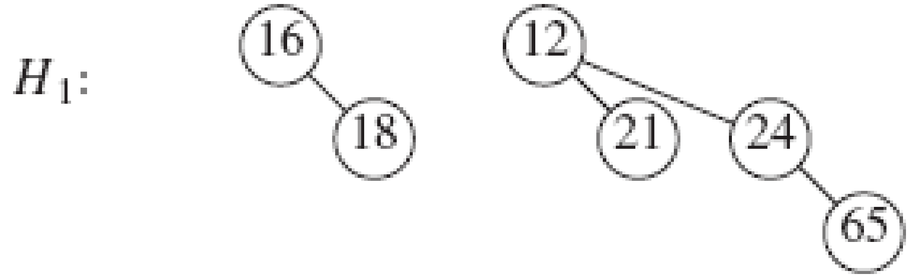


Figure 6.35 Binomial queue H_1 with six elements

Binomial Queue Operations

- The minimum element can be found by scanning the roots of all the trees. There are logN trees so minimum can be found in O(logN)
- Merging 2 binomial queues is performed by essentially adding two queues together.
- Merging Two trees takes constant time. There are O(log N) binomial trees, the merge takes O(logN) time worst case.

Merging Two Queues

- For example merging two queues H_1 and $H_{2,}$ Let H_3 be new queue

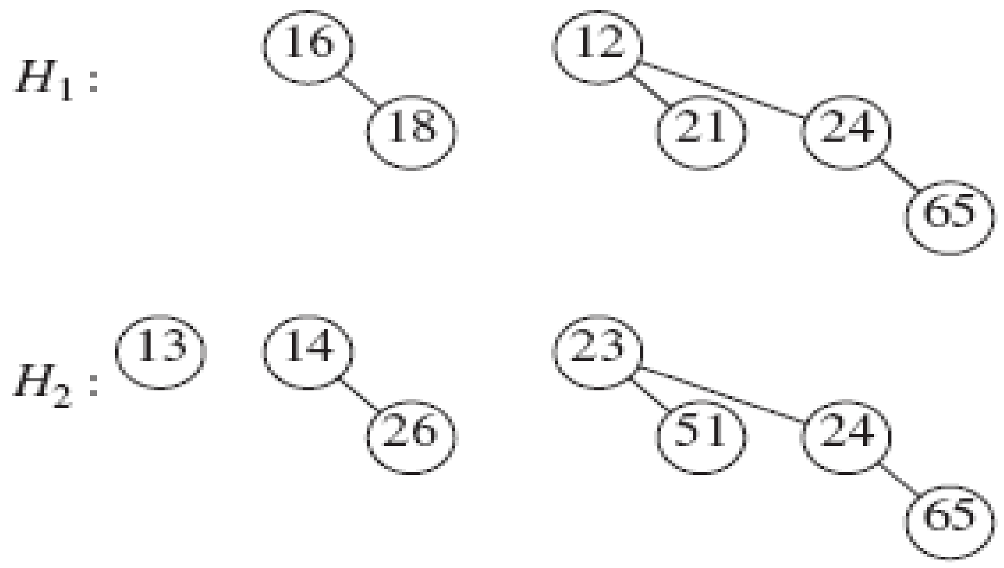


Figure 6.36 Two binomial queues H_1 and H_2

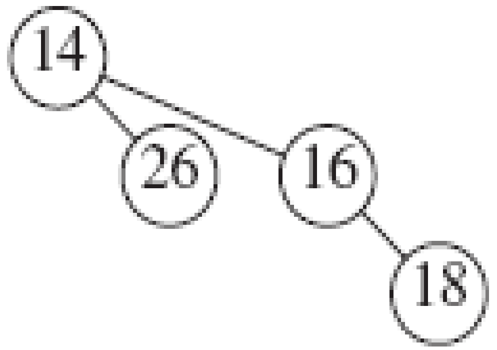


Figure 6.37 Merge of the two B_1 trees in H_1 and H_2

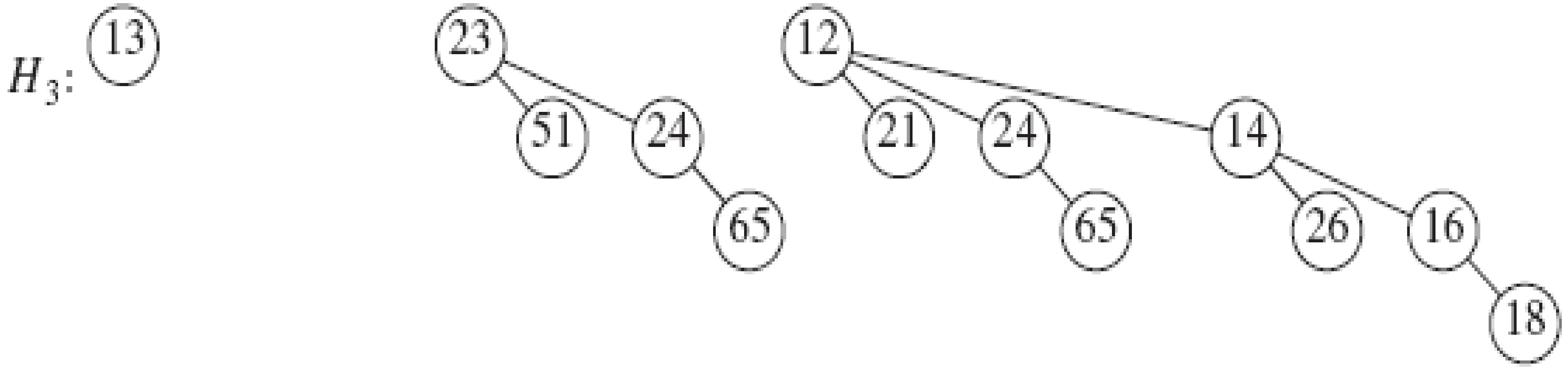


Figure 6.38 Binomial queue H_3: the result of merging H_1 and H_2

Insertion

- Insertion is just a special case of merging, we create a one-node tree and perform merge. The worst case time for this operation is O(logN).
- If the priority queue into which the element is inserted has the property that the smallest nonexistent binomial tree is B_i the running time is proportional to i+1.
- Since each tree in a binomial queue is present with probability of ½, it follows that we expect an insertion to terminate in two step, so the average time is constant.

Example

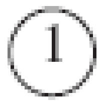


Figure 6.39 After 1 is inserted

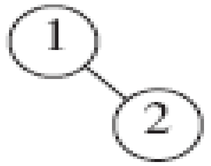


Figure 6.40 After 2 is inserted

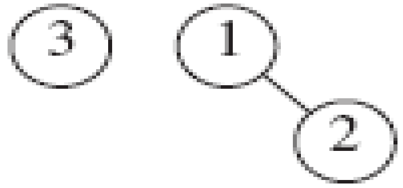


Figure 6.41 After 3 is inserted

Example

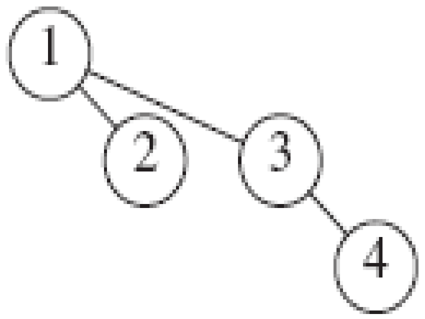


Figure 6.42 After 4 is inserted

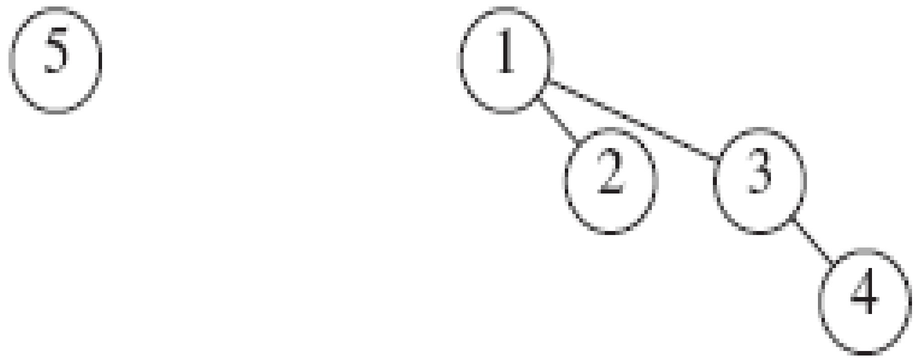


Figure 6.43 After 5 is inserted

Example

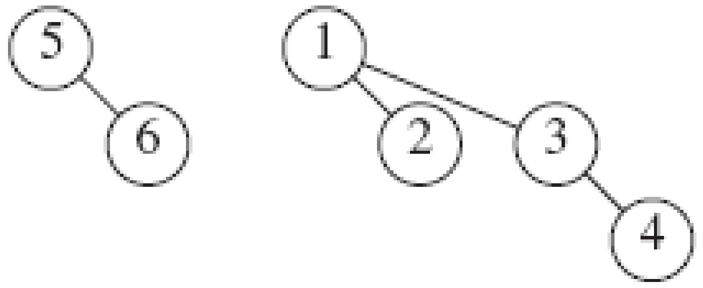


Figure 6.44 After 6 is inserted

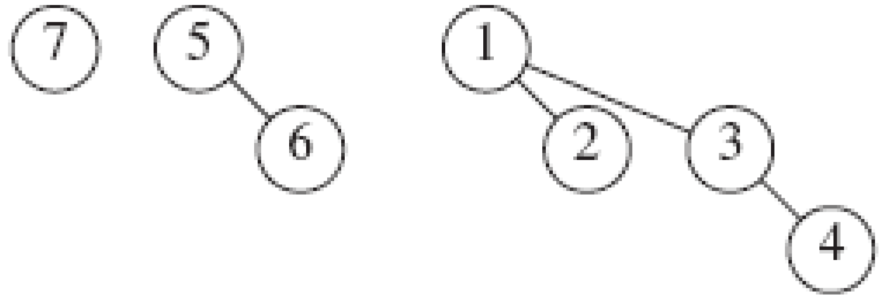


Figure 6.45 After 7 is inserted

deleteMin Funtion

- it can be performed by first finding the binomial tree with the smallest root.
- Let this tree be B_k and let the original priority queue be called H
- Remove B_k from H, forming the new binomial queue H'
- Remove the root of B_k creating binomial trees B_0, B_1,…. B_{k-1}, which collectively form the binomial queue H'' .
- Merge H' and H'' .

Example: perform deleteMin

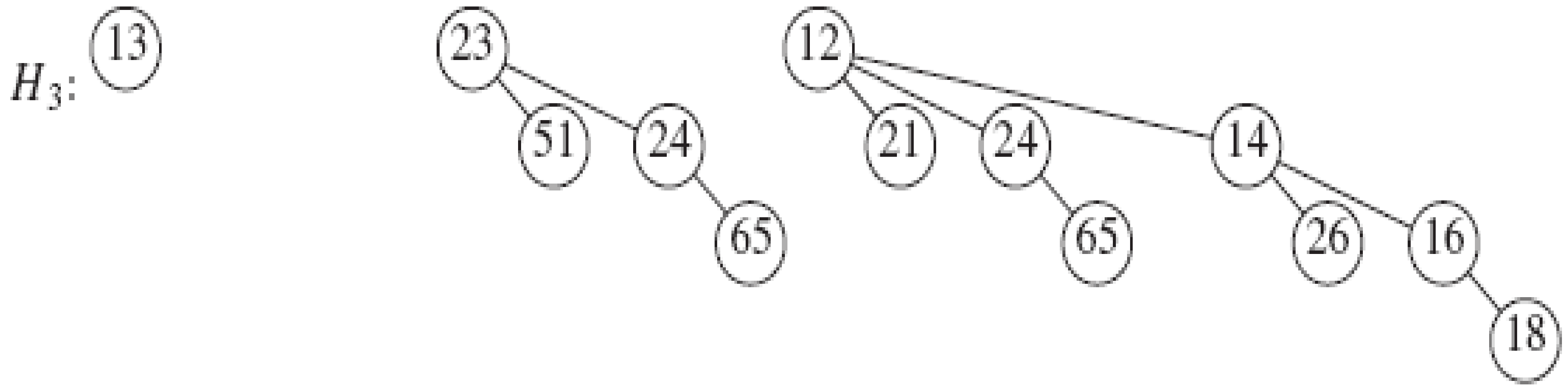


Figure 6.46 Binomial queue H_3

Example:

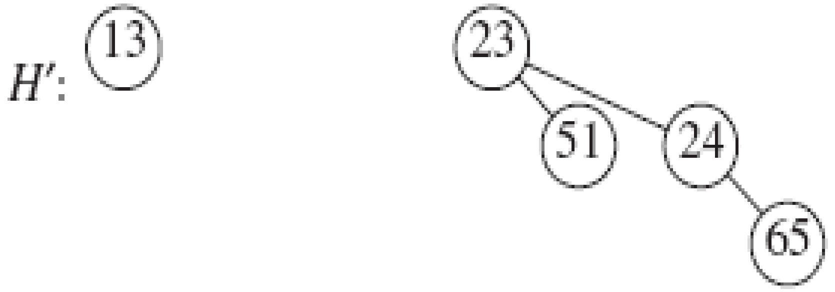


Figure 6.47 Binomial queue H', containing all the binomial trees in H_3 except B_3

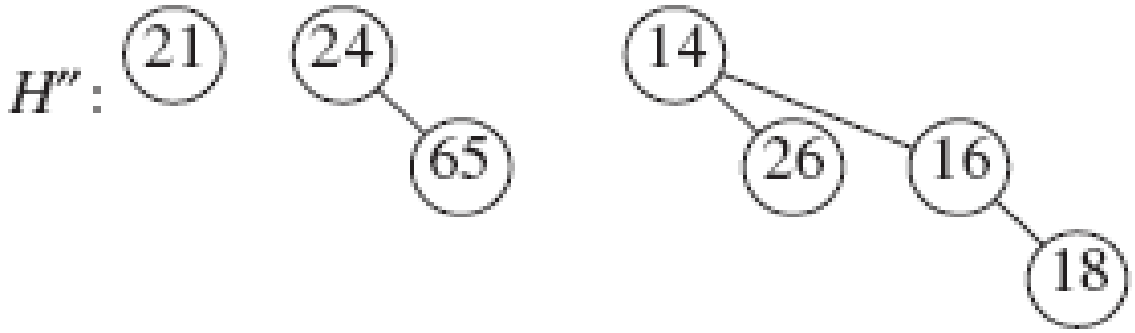


Figure 6.48 Binomial queue H'': B_3 with 12 removed

Example:

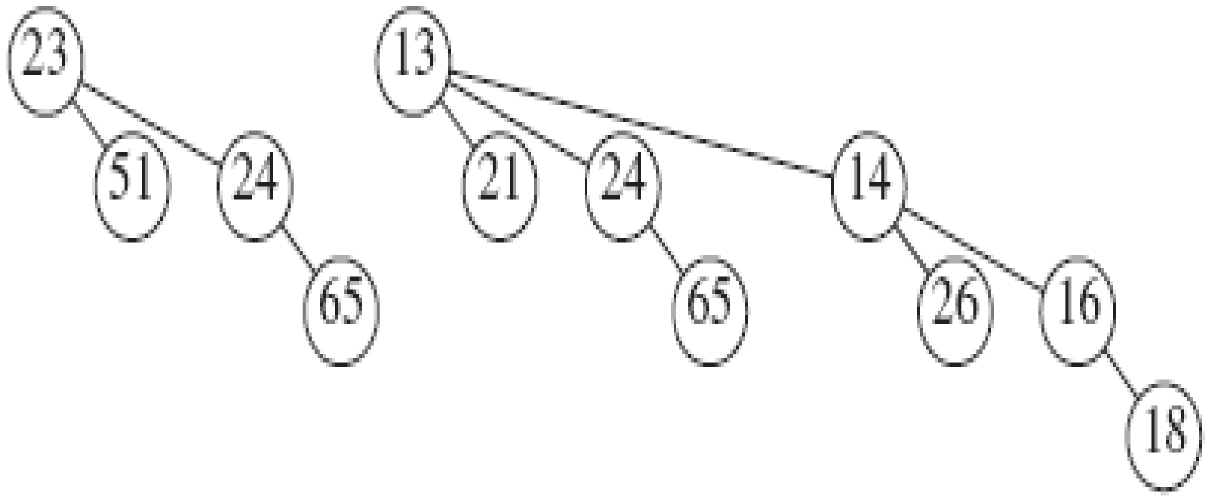


Figure 6.49 Result of applying `deleteMin` to H_3

Analysis

- The deleteMin operation breaks the original binomial tree into two.
- It takes O(logN) time to find the tree containing the minimum element and to create the queue H' and H'' .
- Merging these two queues takes another O(logN)
- so total is O(logN)

Implementation

- The deleteMin operation requires the ability to find all the subtrees of the root quickly, so the standard representation of the general tree is required.
- The children of each node are kept in a linked list and each node has a reference to its first child.
- The Binomial queue will be array of binomial trees.
- Each node in the a binomial tree will contain the data, first child and right sibling.

Example: Binomial Queue

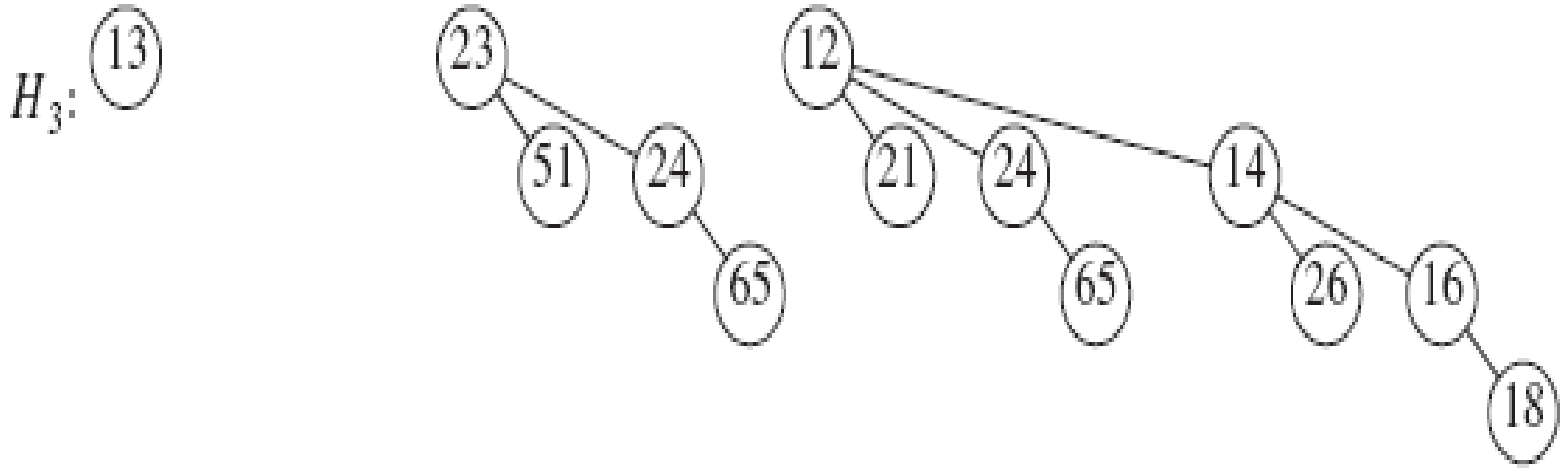


Figure 6.50 Binomial queue H_3 drawn as a forest

Example: Binomial Queue Implementation

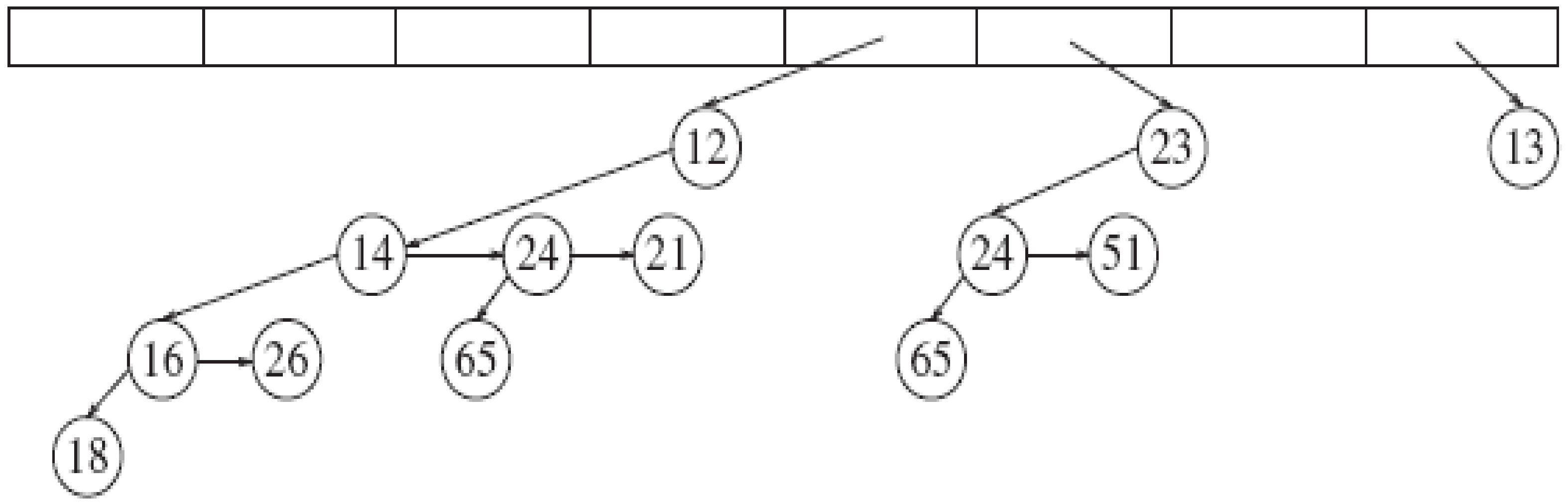


Figure 6.51 Representation of binomial queue H_3

```
public class BinomialQueue<AnyType extends Comparable<? super AnyType>>
{
    public BinomialQueue( )
      { /* See online code */ }
    public BinomialQueue( AnyType item )
      { /* See online code */ }

    public void merge( BinomialQueue<AnyType> rhs )
      { /* Figure 6.55 */ }
    public void insert( AnyType x )
      { merge( new BinomialQueue<>( x ) ); }
    public AnyType findMin( )
      { /* See online code */ }
    public AnyType deleteMin( )
      { /* Figure 6.56 */ }

    public boolean isEmpty( )
      { return currentSize == 0; }
    public void makeEmpty( )
      { /* See online code */ }

    private static class Node<AnyType>
    {
            // Constructors
        Node( AnyType theElement )
          { this( theElement, null, null ); }

        Node( AnyType theElement, Node<AnyType> lt, Node<AnyType> nt )
          { element = theElement; leftChild = lt;  nextSibling = nt; }

        AnyType        element;      // The data in the node
        Node<AnyType> leftChild;    // Left child
        Node<AnyType> nextSibling; // Right child
    }

    private static final int DEFAULT_TREES = 1;

    private int currentSize;                 // # items in priority queue
    private Node<AnyType> [ ] theTrees;  // An array of tree roots

    private void expandTheTrees( int newNumTrees )
      { /* See online code */ }
    private Node<AnyType> combineTrees( Node<AnyType> t1, Node<AnyType> t2 )
      { /* Figure 6.54 */ }

    private int capacity( )
      { return ( 1 << theTrees.length ) - 1; }
    private int findMinIndex( )
      { /* See online code */ }
}
```

Figure 6.52 Binomial queue class skeleton and node definition

Merge

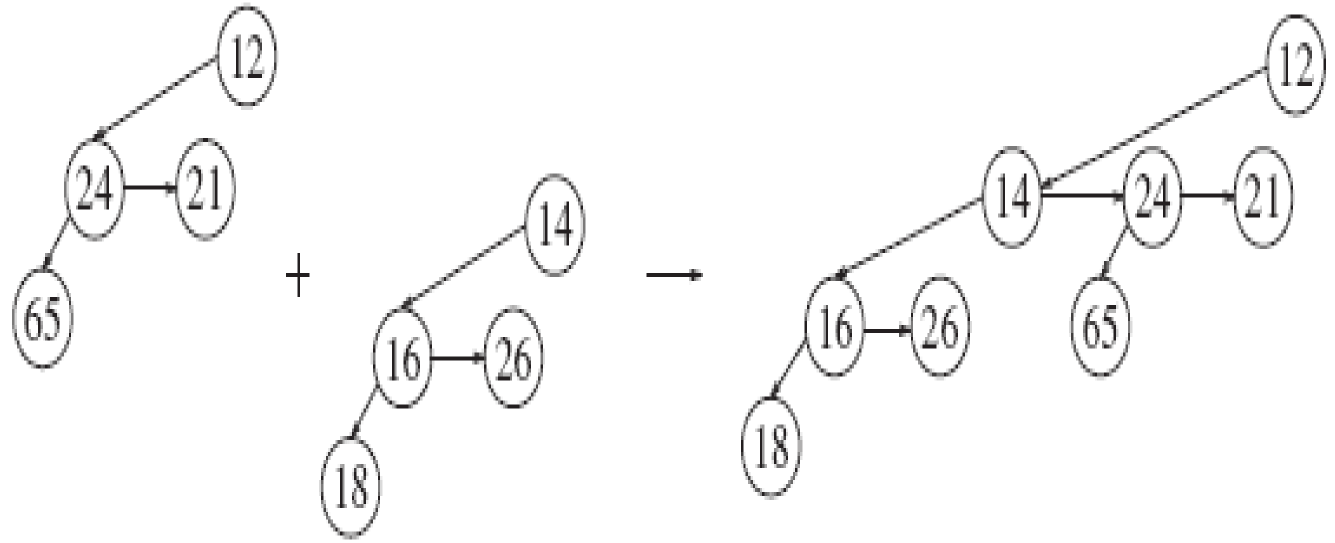


Figure 6.53 Merging two binomial trees

Merge

- To merge two tree efficiently we need to place them in decreasing order in the array.
- To merge two queues we routinely merge two binomial tees of the same size.
- H_1 is represented by the current object and H_2 by rhs. The routine combines H_1 and H_2, placing the result in H_1 and making H_2 empty

Merge

```java
/**
 * Return the result of merging equal-sized t1 and t2.
 */
private Node<AnyType> combineTrees( Node<AnyType> t1, Node<AnyType> t2 )
{
    if( t1.element.compareTo( t2.element ) > 0 )
        return combineTrees( t2, t1 );
    t2.nextSibling = t1.leftChild;
    t1.leftChild = t2;
    return t1;
}
```

Figure 6.54 Routine to merge two equal-sized binomial trees

```java
    /**
     * Merge rhs into the priority queue.
     * rhs becomes empty. rhs must be different from this.
     * @param rhs the other binomial queue.
     */
    public void merge( BinomialQueue<AnyType> rhs )
    {
        if( this == rhs )    // Avoid aliasing problems
            return;

        currentSize += rhs.currentSize;

        if( currentSize > capacity( ) )
        {
            int maxLength = Math.max( theTrees.length, rhs.theTrees.length );
            expandTheTrees( maxLength + 1 );
        }

        Node<AnyType> carry = null;
        for( int i = 0, j = 1; j <= currentSize; i++, j *= 2 )
        {
            Node<AnyType> t1 = theTrees[ i ];
            Node<AnyType> t2 = i < rhs.theTrees.length ? rhs.theTrees[ i ] : null;

            int whichCase = t1 == null ? 0 : 1;
            whichCase += t2 == null ? 0 : 2;
            whichCase += carry == null ? 0 : 4;

            switch( whichCase )
            {
              case 0: /* No trees */
              case 1: /* Only this */
                break;
              case 2: /* Only rhs */
                theTrees[ i ] = t2;
                rhs.theTrees[ i ] = null;
                break;
              case 4: /* Only carry */
                theTrees[ i ] = carry;
                carry = null;
                break;
              case 3: /* this and rhs */
                carry = combineTrees( t1, t2 );
                theTrees[ i ] = rhs.theTrees[ i ] = null;
                break;
```

Figure 6.55 Routine to merge two priority queues

```
                case 5: /* this and carry */
                  carry = combineTrees( t1, carry );
                  theTrees[ i ] = null;
                  break;
                case 6: /* rhs and carry */
                  carry = combineTrees( t2, carry );
                  rhs.theTrees[ i ] = null;
                  break;
                case 7: /* All three */
                  theTrees[ i ] = carry;
                  carry = combineTrees( t1, t2 );
                  rhs.theTrees[ i ] = null;
                  break;
              }
          }

          for( int k = 0; k < rhs.theTrees.length; k++ )
              rhs.theTrees[ k ] = null;
          rhs.currentSize = 0;
      }
```

Figure 6.55 *(continued)*

```
    /**
     * Remove the smallest item from the priority queue.
     * @return the smallest item, or throw UnderflowException if empty.
     */
    public AnyType deleteMin( )
    {
        if( isEmpty( ) )
            throw new UnderflowException( );

        int minIndex = findMinIndex( );
        AnyType minItem = theTrees[ minIndex ].element;

        Node<AnyType> deletedTree = theTrees[ minIndex ].leftChild;

        // Construct H''
        BinomialQueue<AnyType> deletedQueue = new BinomialQueue<>( );
        deletedQueue.expandTheTrees( minIndex + 1 );

        deletedQueue.currentSize = ( 1 << minIndex ) - 1;
        for( int j = minIndex - 1; j >= 0; j-- )
        {
            deletedQueue.theTrees[ j ] = deletedTree;
            deletedTree = deletedTree.nextSibling;
            deletedQueue.theTrees[ j ].nextSibling = null;
        }

        // Construct H'
        theTrees[ minIndex ] = null;
        currentSize -= deletedQueue.currentSize + 1;

        merge( deletedQueue );

        return minItem;
    }
```

Figure 6.56 deleteMin for binomial queues, with findMinIndex method