Chap. 6 Priority Queues (Heaps)

6.1 Model

- The priority queue ADT is similar to the queue ADT except the dequeue is done based on the priorities assigned to each element in the queue.
- Examples: printer queue, *OS* scheduling.
- Two major operations:
  - *insert:* insert a new element
  - *deleteMin*: find, return and remove the minimum (highest priority) element.

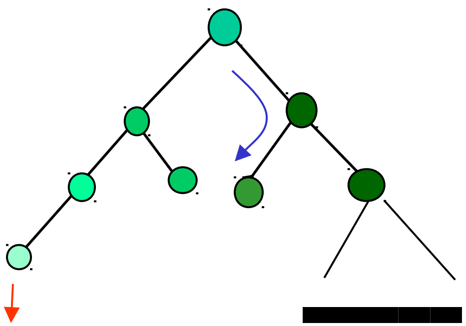
deleteMin ← Priority Queue ← insert

6.2 Simple implementation

- Linked list implementation
  - sorted list: *insert* $O(N)$, *deleteMin* $O(1)$
  - unsorted list: *insert* $O(1)$, *deleteMin* $O(N)$

– Binary search tree implementation

- insert is random, *deleteMin* is not (make right tree heavy)
- $O(logN)$, in average, for both deleteMin and insert
- An overkill (most of BST operations are not useful)



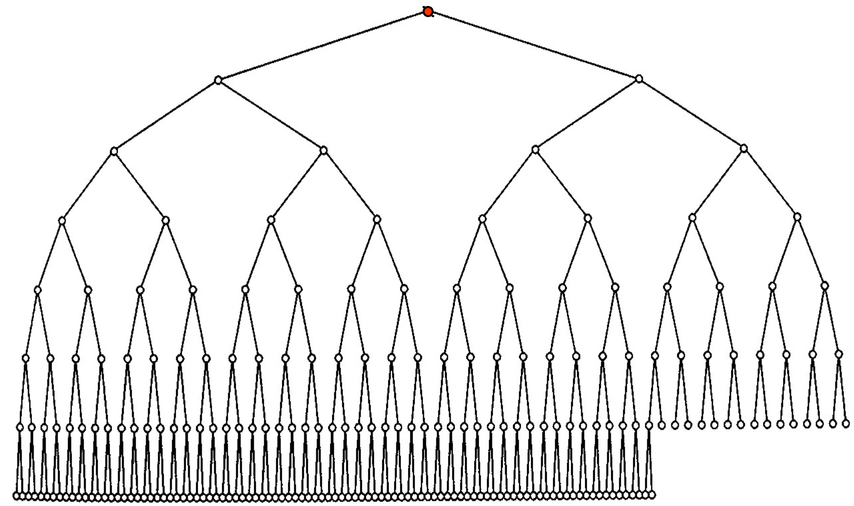


Figure 6.13 A very large complete binary tree

6.3 Binary Heaps (Heaps)

– Structure property

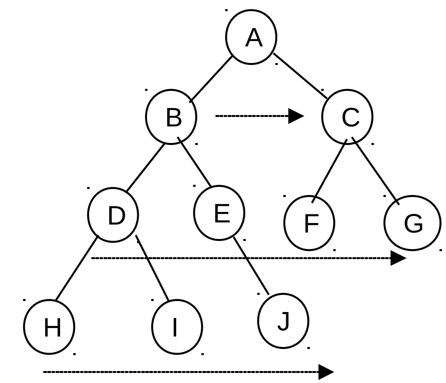
- A Heap is a binary tree that is completely filled, with the possible exception of the bottom level, which can be partially filled from left to right. Such binary tree is also called *Complete Binary Tree*.
- The height of a complete binary tree is $\lfloor logN \rfloor$, or $O(logN)$.

- Array implementation

- The array

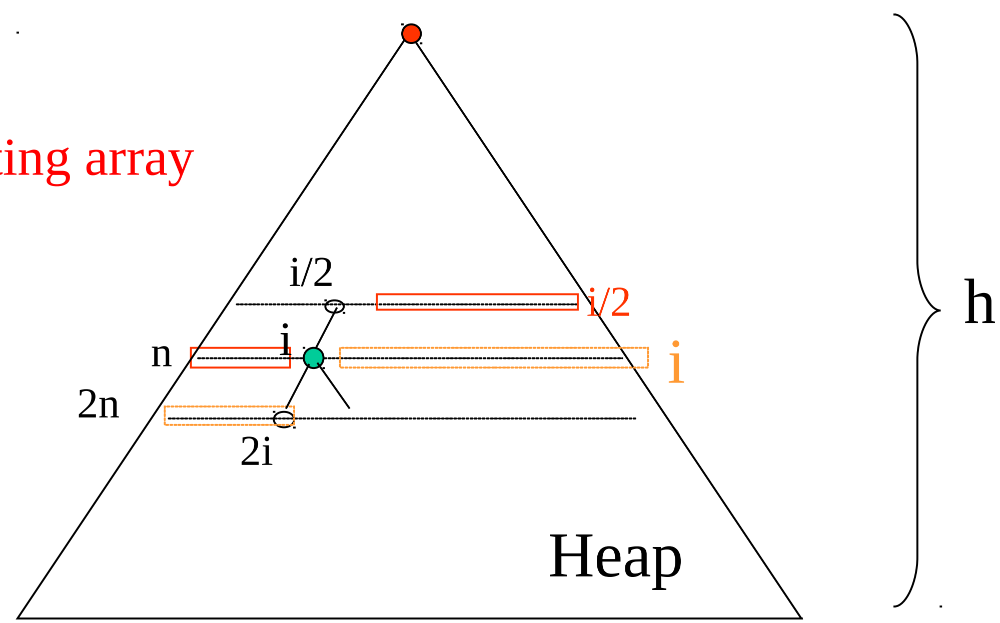
i

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



- The left child of $a[i]$ is $a[2i]$, the parent of $a[i]$ is $a[\lfloor i/2 \rfloor]$.
- Need to keep the current heap size. `[Fig. 6.4]`

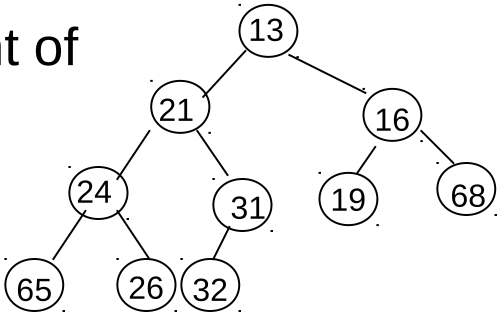
Traversing tree → manipulating array



$N=2^h-1 \sim 2^{h+1}-1$

– Heap-order property

- for every non-root node X, the key in the parent of X is smaller than (or equal to) the key in X.
- the root is the smallest element.
- *findMin* is a constant time operation.



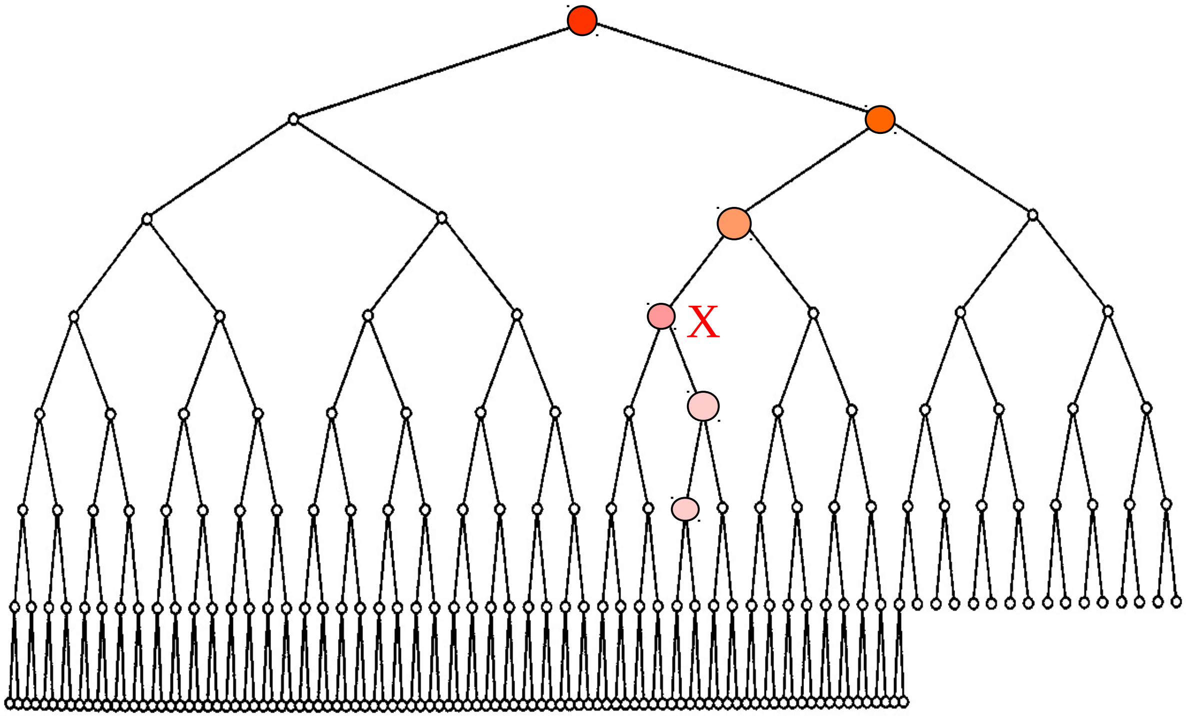


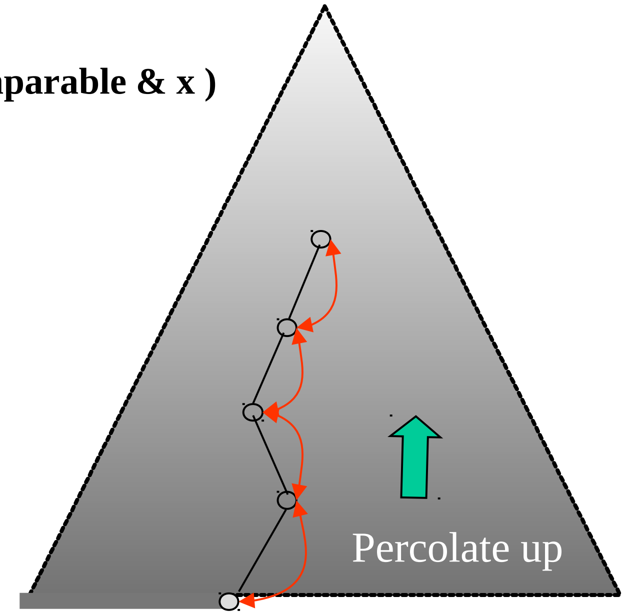
Figure 6.13 A very large complete binary tree

```
template <class Comparable>
    class BinaryHeap
    {
      public:
        explicit BinaryHeap( int capacity = 100 );
        bool isEmpty( ) const;
        bool isFull( ) const;
        const Comparable & findMin( ) const;
        void insert( const Comparable & x );
        void deleteMin( );
        void deleteMin( Comparable & minItem );
        void makeEmpty( );
      private:
        int   currentSize;  // Number of elements in heap
        vector<Comparable> array;        // The heap array
        void buildHeap( );
        void percolateDown( int hole );
    };
```

– *Insert*

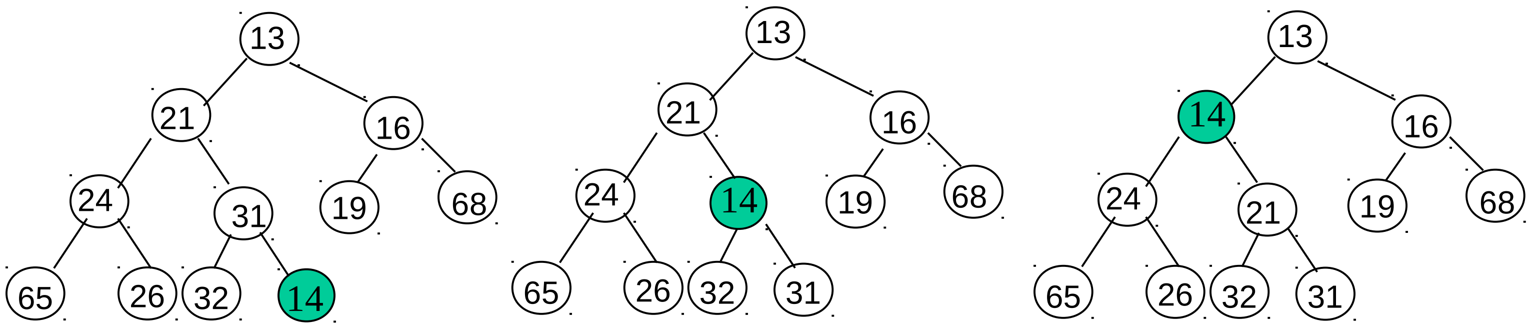
1. Generate an empty node at the end of the array
2. If the new element X can be put in the empty node without violating the heap order, do it, otherwise move the parent of the empty node into the empty node to generate a new empty node (hole) at the parent node.
3. Repeat (2) with the new empty node until X can be inserted.
   - C++ implementation `[Fig. 6.8]`

```
template <class Comparable>
    void BinaryHeap<Comparable>::insert( const Comparable & x )
    {
      if( isFull( ) )  throw Overflow( );
         // Percolate up
      int hole = ++currentSize;
      for( ; hole > 1 && x < array[ hole / 2 ]; hole /= 2 )
         array[ hole ] = array[ hole / 2 ];
      array[ hole ] = x;
    }
```



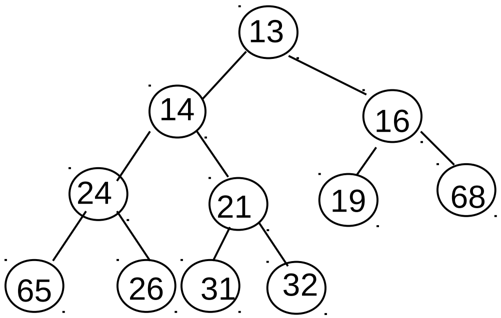
   - $O(logN)$

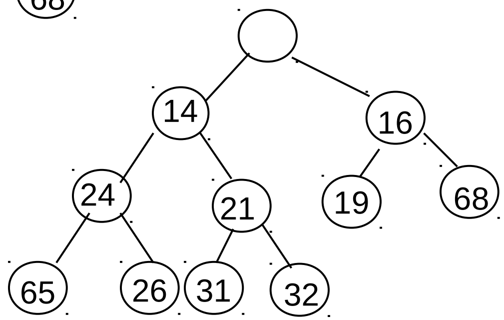
- example: insert *14* to the next heap.

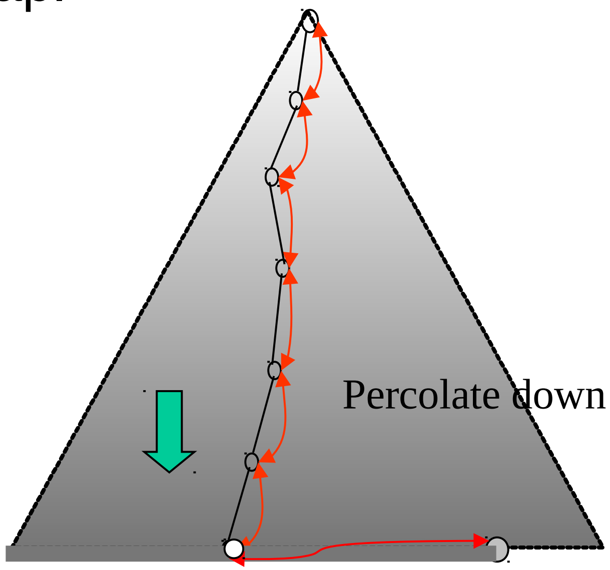


– *deleteMin*

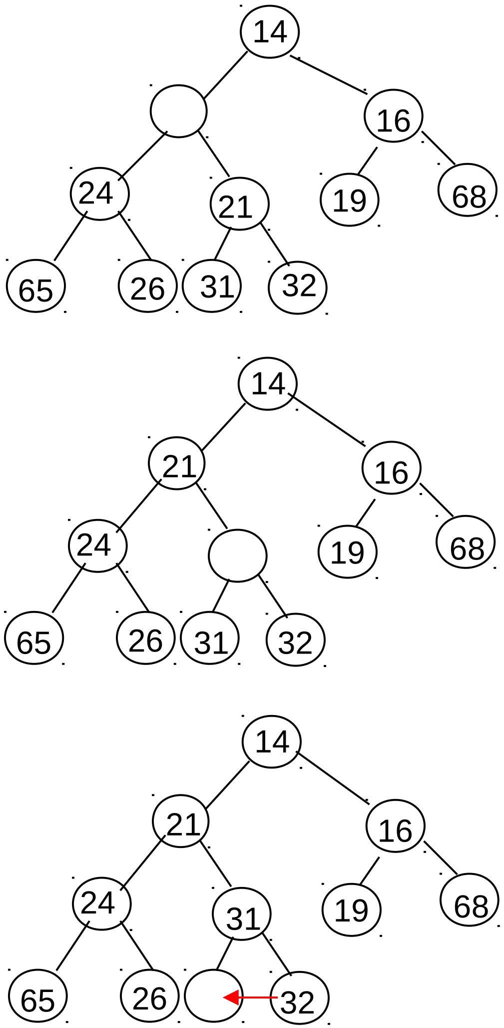
1. Remove the root, which generate a hole at the root.
2. If the last element of the heap can be moved into the hole, do it, otherwise, move the smaller child of the hole into the hole, which generates a new hole.
3. Repeat (2) with the new hole until the last element of the heap can be placed.
   - example: deleteMin from the next heap.







- C++ implementation `[Fig. 6.12]`
- *O(logN)*

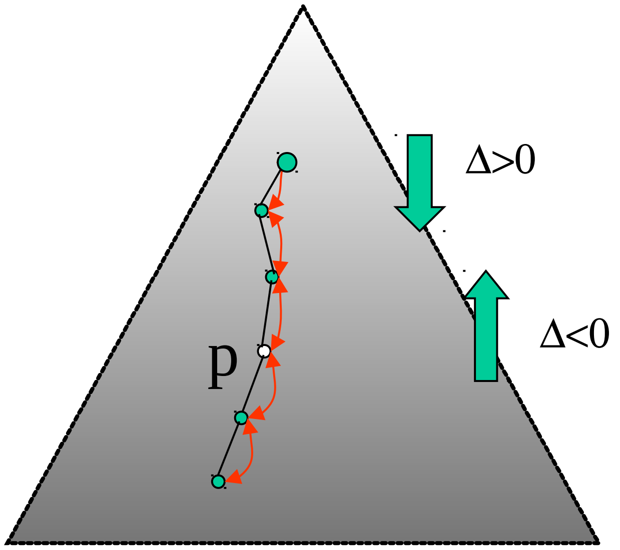



```
template <class Comparable>
    void BinaryHeap<Comparable>::deleteMin( )
    {
        if( isEmpty( ) ) throw Underflow( );
        array[ 1 ] = array[ currentSize-- ];
        percolateDown( 1 );
    }
```

```
template <class Comparable>
    void BinaryHeap<Comparable>::percolateDown( int hole )   {
/*1*/      int child;
/*2*/      Comparable tmp = array[ hole ];
/*3*/      for( ; hole * 2 <= currentSize; hole = child ) {
/*4*/          child = hole * 2;
/*5*/          if( child != currentSize && array[ child + 1 ] < array[ child ] )
/*6*/              child++;
/*7*/          if( array[ child ] < tmp )
/*8*/              array[ hole ] = array[ child ];
/*9*/          else break;
           }
/*10*/     array[ hole ] = tmp;
    }
```

– Other operation

- Need a way to identify the position of a given element (*e.g.* by hashing)
- *decresingKey* (P, Δ) : $O(logN)$
- *incresingKey* (P, Δ) : $O(logN)$
- *remove* (P) : $O(logN)$



– *BuildHeap*

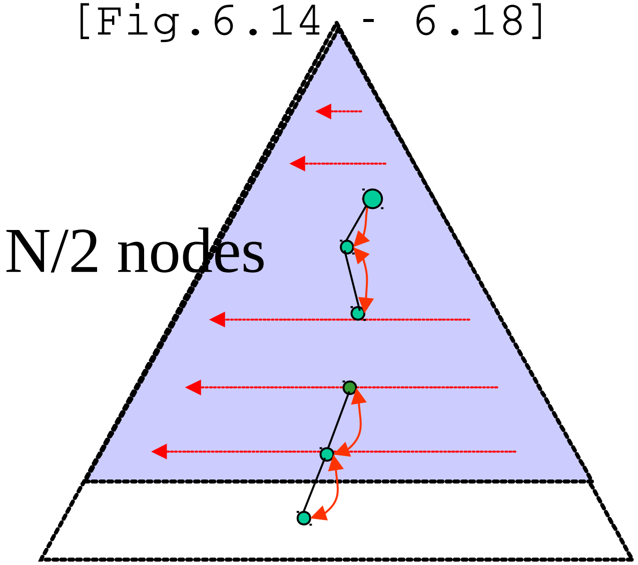
- buildHeap: build a heap from N input items
- N insert operations: $O(N)$ average, $O(N \log N)$ worst case
- A guaranteed linear algorithm:

  1.) put all element into a binary tree of arbitrary order

  2.) check all non-leaf nodes in a bottom-up order

  3.) for each node being checked, compare with its children to ensure heap-order, percolate down if necessary.

  [Fig.6.14 - 6.18]




```
template <class Comparable>
    void
BinaryHeap<Comparable>::buildHeap( )
    {
        for( int i = currentSize / 2; i > 0; i-- )
            percolateDown( i );
    }
```

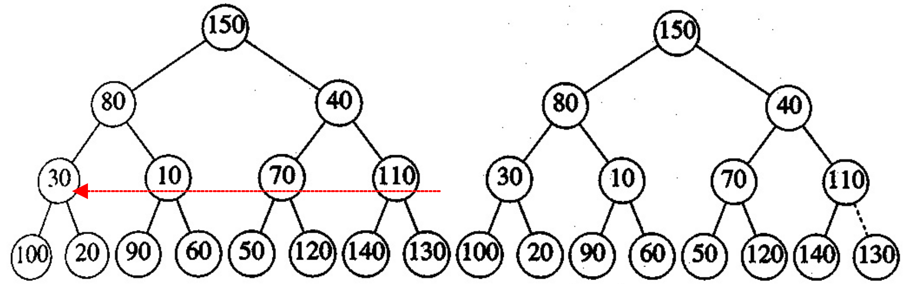


Figure 6.15 Left: initial heap; right: after percolateDown(7)

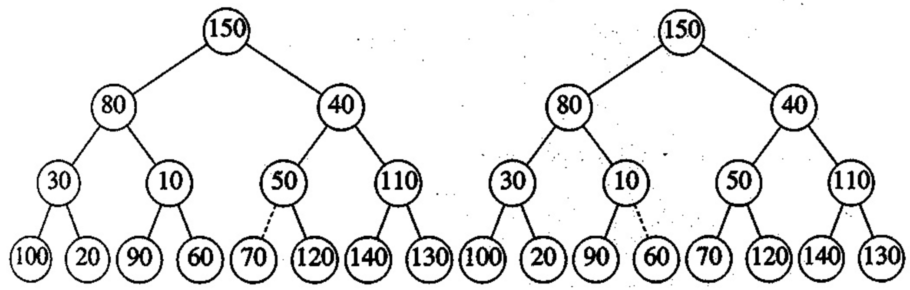


Figure 6.16 Left: after percolateDown(6); right: after percolateDown(5)

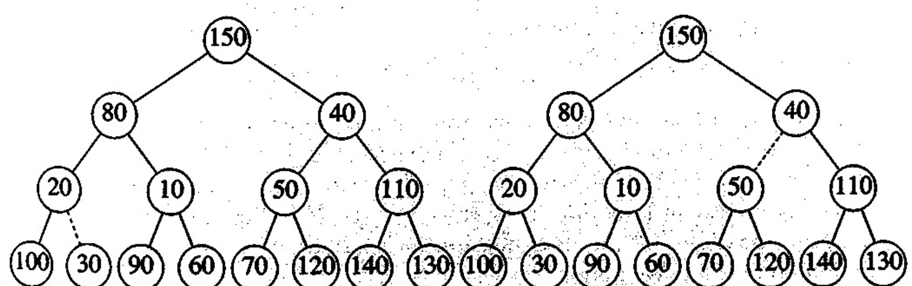
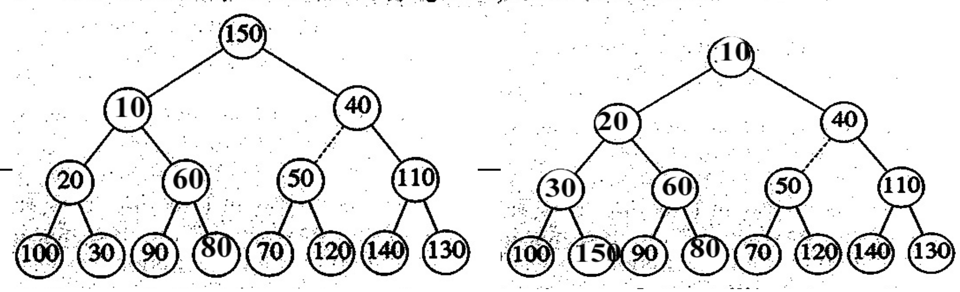


Figure 6.17 Left: after percolateDown(4); right: after percolateDown(3)



- Theorem:

  For the perfect binary tree of height h containing $2^{h+1} - 1$ nodes, the sum of the heights of the nodes is:

$$2^{h+1} - 1 - (h + 1)$$

6.4 **Application: the selection problem**

- The selection problem: given N elements, find the k^{th} smallest element.
- A simple algorithm: sort the N elements in increasing order, and take the k^{th} element. A simple sorting algorithm costs $O(N^2)$.
- The heap selection

  1.) Read the N elements into an array

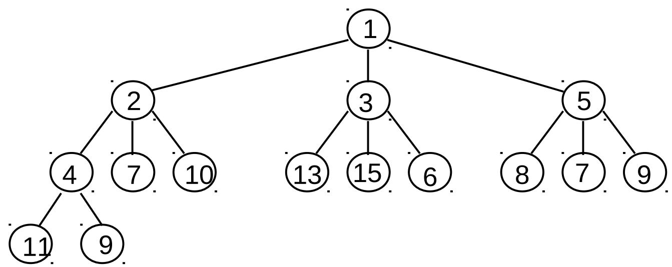
  2.) Apply the buildHeap operation

  3.) Perform k deleteMin operations.

  - Worst case time: $O(N + k\ logN) = O(N\ logN)$
  - When $k = N$, the algorithm becomes a $O(N\ logN)$ sorting algorithm.

6.5 d-Heaps

– An extension of binary heap — a *d-ary* tree structure



– *insert:* $O(\log_d N)$

– *deleteMin:* $O(d \log_d N)$ → *compare with d children*

– Array implementation is not as efficient:

Multiplication & division, in general, cannot be carried out by bit shifting.

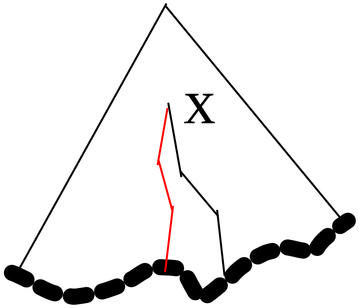
– Maybe used for disk storage (similar to *B*-trees)

– Merge operation is hard

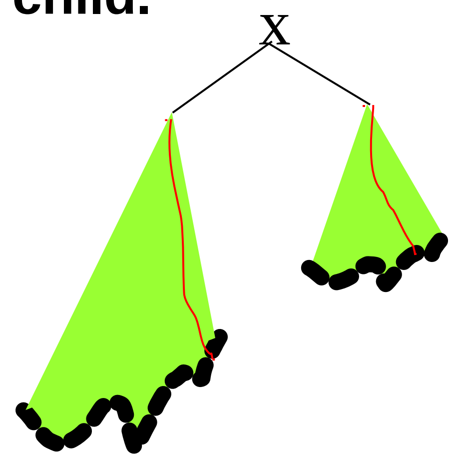
– 6.6 Leftist Heaps

- Binary tree, structure and ordering properties, unbalanced

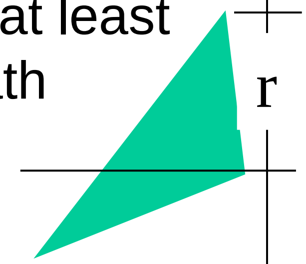
– Properties



- Designed to support merging
- **Binary tree**
- **Heap-order**
- _null path length_ of node X ($npl(X)$): the length of the shortest path from X to a node without two children.
- **Leftist heap property: for every node X, the npl of the left child is at least as large as that of the right child.**
- The rightmost path is the shortest.



- A leftist tree with r nodes on the rightmost path must have at least $2^r - 1$ nodes, *i.e.* a leftist tree of N nodes has a rightmost path containing at most $\lfloor log(N + 1) \rfloor$ nodes.



– Merge operation

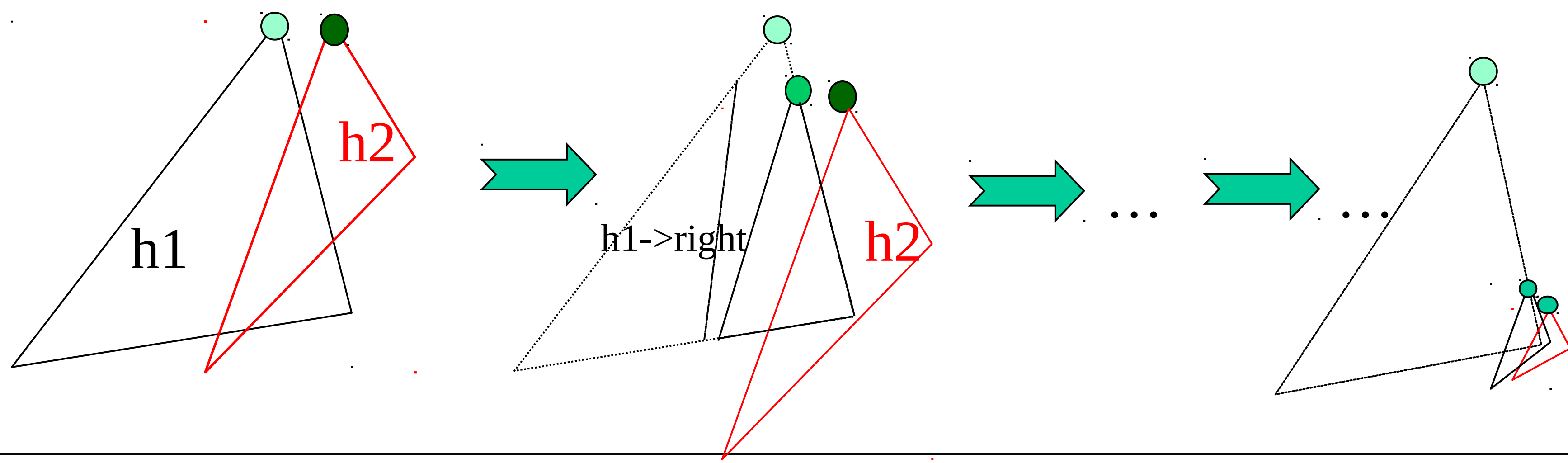
- Let h_1 and h_2 be the root nodes of the two leftist heaps, and

  h_1->*element* **<** h_2->*element*

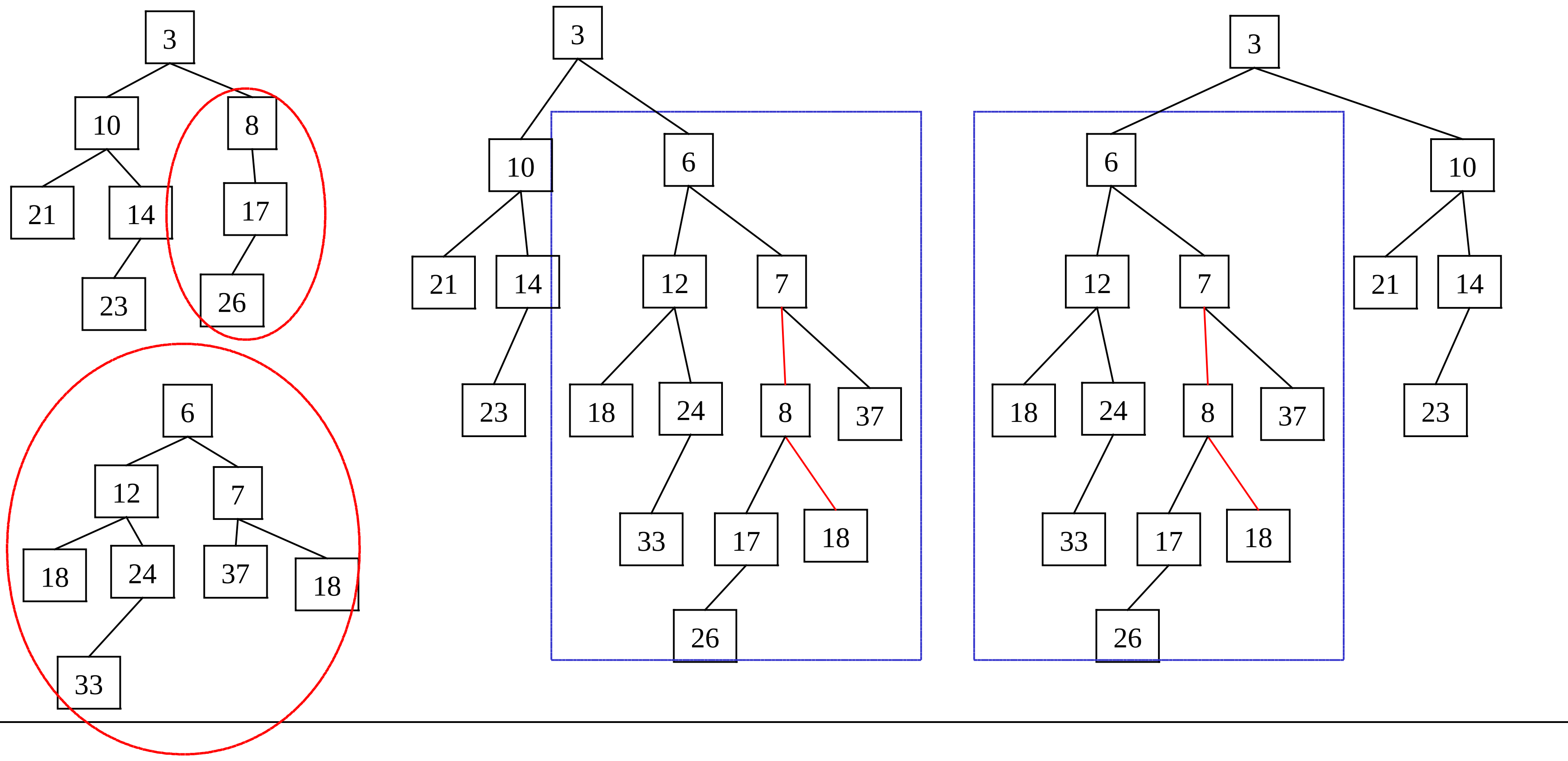
  (otherwise, swap h_1 & h_2)

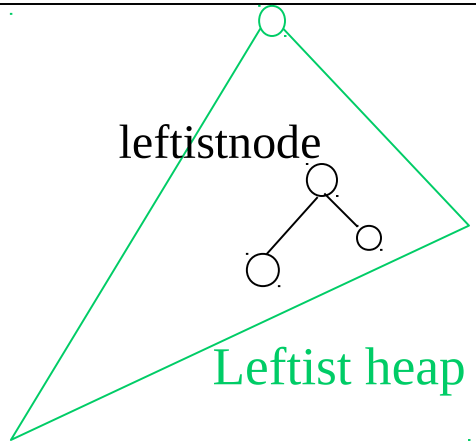
  Recursively merge h_1->*right* and h_2, and link the result back to h_1->*right*.

  If ($npl(h_1$->*left*$)$ < $npl(h_1$->*right*$)$) swap h_1's left & right subtrees.



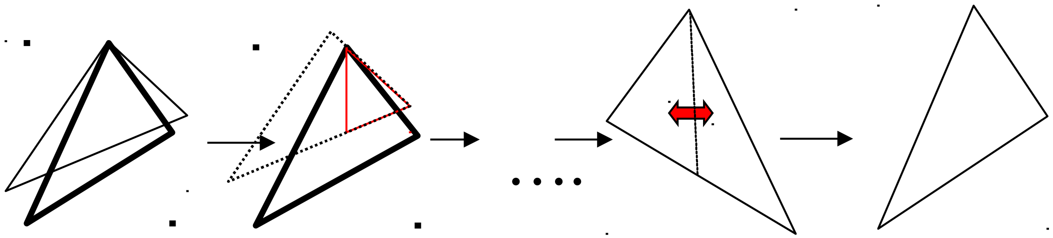
- Running time: the sum of the lengths of the rightmost paths: $O(logN)$
- example: Fig. 6.21 - 6.24
- Implementation: Fig. 6.26 - 6.27






```
template <class Comparable>
    class LeftistHeap;
    template <class Comparable>
    class LeftistNode
    {
        Comparable   element;
        LeftistNode *left;
        LeftistNode *right;
        int       npl;
        LeftistNode( const Comparable & theElement,
LeftistNode *lt = NULL, LeftistNode *rt = NULL, int np = 0 )
: element( theElement ), left( lt ), right( rt ), npl( np ) { }
        friend class LeftistHeap<Comparable>;
    };
```

```
template <class Comparable>
    class LeftistHeap {
    public:
      LeftistHeap( );
      LeftistHeap( const LeftistHeap & rhs );
      ~LeftistHeap( );
      bool isEmpty( ) const;
      bool isFull( ) const;
      const Comparable & findMin( ) const;
      void insert( const Comparable & x );
      void deleteMin( );
      void deleteMin( Comparable & minItem );
      void makeEmpty( );
      void merge( LeftistHeap & rhs );
      const LeftistHeap & operator=( const LeftistHeap & rhs );
    private:
      LeftistNode<Comparable> *root;
      LeftistNode<Comparable> * merge(
LeftistNode<Comparable> *h1,          LeftistNode<Comparable>
*h2 ) const;
      LeftistNode<Comparable> * merge1(
            LeftistNode<Comparable> *h1,
            LeftistNode<Comparable> *h2 ) const;
      void swapChildren( LeftistNode<Comparable> * t ) const;
      void reclaimMemory( LeftistNode<Comparable> * t ) const;
      LeftistNode<Comparable> * clone(
                    LeftistNode<Comparable> *t ) const;
    };
```



merge → merge1 → merge → merge1 → merge → merge1

```
template <class Comparable>
LeftistNode<Comparable> *LeftistHeap<Comparable>::merge(
LeftistNode<Comparable> * h1, LeftistNode<Comparable> * h2 ) const
    {  if( h1 == NULL ) return h2;
       if( h2 == NULL ) return h1;
       if( h1->element < h2->element ) return merge1( h1, h2 );
       else   return merge1( h2, h1 );
    }

/** * Swaps t's two children.        */
template <class Comparable>
void LeftistHeap<Comparable>::swapChildren( LeftistNode<Comparable> * t ) const
{
        LeftistNode<Comparable> *tmp = t->left;
        t->left = t->right;
        t->right = tmp;
}
```

```
/** Internal method to merge two roots.Assumes trees are not empty, and h1's root contains smallest item.        */
template <class Comparable>
LeftistNode<Comparable> *
LeftistHeap<Comparable>::merge1( LeftistNode<Comparable> * h1,
                    LeftistNode<Comparable> * h2 ) const
{
   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;
}
```

– Other operations

- insert: merge a one node heap to a leftist heap: $O(logN)$
- deleteMin: delete the root, and merge the left & right subtrees: $O(logN)$.
- Implementation: [Fig. 6.29 - 6.30]

```
template <class Comparable>
    void LeftistHeap<Comparable>::insert( const Comparable & x )
    {         root = merge( new LeftistNode<Comparable>( x ), root );
    }
/*** Find the smallest item in the priority queue. * Return the smallest
item, or throw Underflow if empty.         */
    template <class Comparable>
    const Comparable & LeftistHeap<Comparable>::findMin( ) const
    {
      if( isEmpty( ) )
        throw Underflow( );
      return root->element;
    }
```

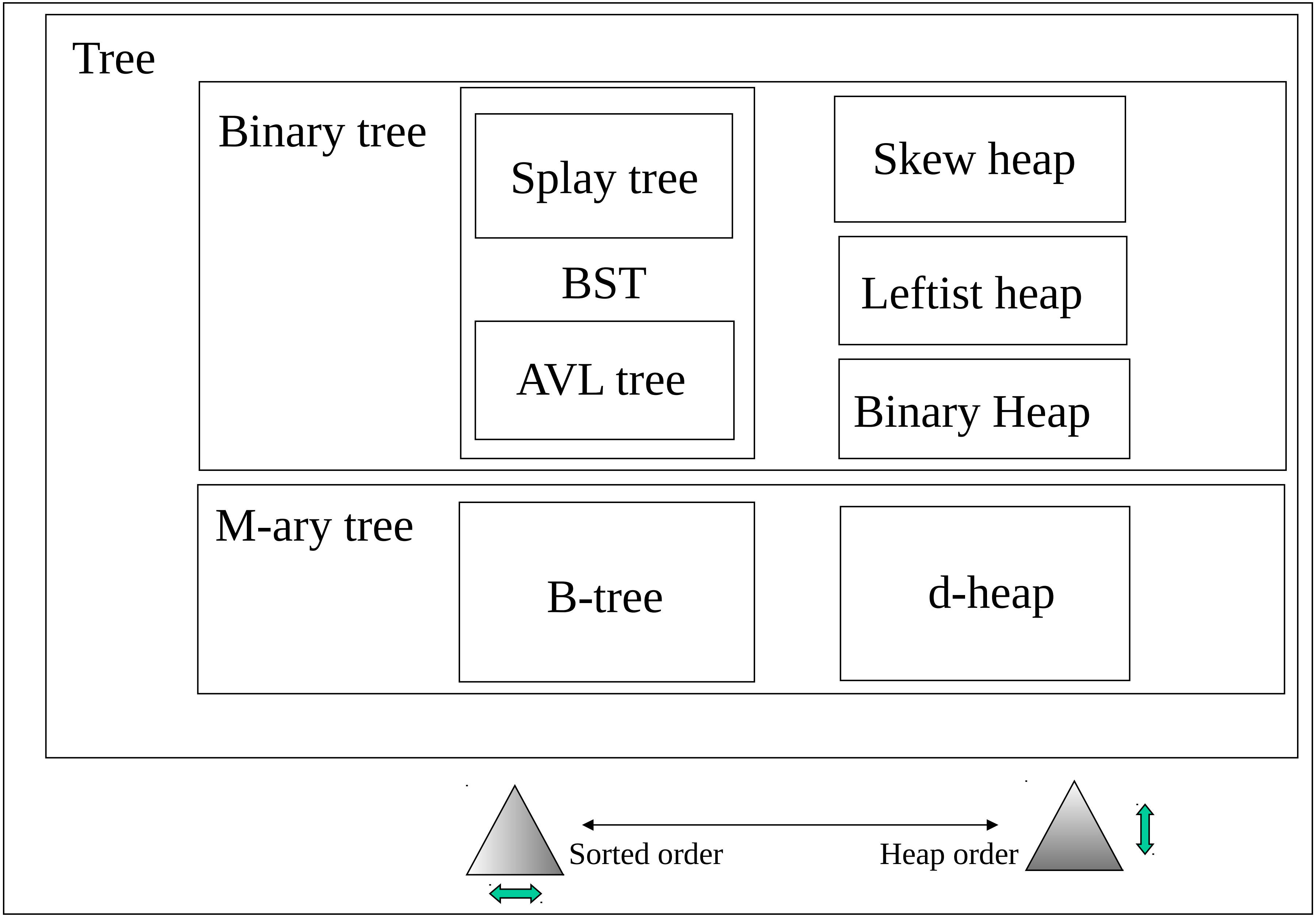
```
/*** Remove the smallest item from the priority
queue.
Throws Underflow if empty.         */
    template <class Comparable>
    void LeftistHeap<Comparable>::deleteMin( )
    {
      if( isEmpty( ) )
        throw Underflow( );
      LeftistNode<Comparable> *oldRoot = root;
      root = merge( root->left, root->right );
      delete oldRoot;
    }
```

6.7 Skew Heaps

- A skew heap is a binary tree with heap-order
- No structural constraints — worst case can be $O(N)$
- Amortized cost for node access: $O(logN)$
- Merge operation

  Same recursion process as in leftist heap, except that the left & right subtree swapping always happen.

- Example: `[Fig. 6.31 - 6.33]`


Tree
Binary tree
Splay tree
BST
AVL tree
Skew heap
Leftist heap
Binary Heap
M-ary tree
B-tree
d-heap
Sorted order
Heap order