CMSC 341

Binary Heaps

Priority Queues

Priority Queues

- Priority: some property of an object that allows it to be prioritized with respect to other objects of the same type
- Min Priority Queue: homogeneous collection of Comparables with the following operations (duplicates are allowed). Smaller value means higher priority.
  - `void insert (Comparable x)`
  - `void deleteMin( )`
  - `void deleteMin ( Comparable min)`
  - `Comparable findMin( )`
  - `Construct from a set of initial values`
  - `boolean isEmpty( )`
  - `boolean isFull( )`
  - `void makeEmpty( )`

Priority Queue Applications

- Printer management:
  - The shorter document on the printer queue, the higher its priority.
- Jobs queue within an operating system:
  - Users' tasks are given priorities. System priority high.
- Simulations
  - The time an event "happens" is its priority.
- Sorting (heap sort)
  - An elements "value" is its priority.

Possible Implementations

- Use a sorted list. Sorted by priority upon insertion.
  - findMin( ) --> list.front( )
  - insert( ) --> list.insert( )
  - deleteMin( ) --> list.erase(list.begin( ))
- Use ordinary BST
  - findMin( ) --> tree.findMin( )
  - insert( ) --> tree.insert( )
  - deleteMin( ) --> tree.delete(tree.findMin( ))
- Use balanced BST
  - guaranteed O(lg n) for Red-Black

Min Binary Heap

- A min binary heap is a complete binary tree with the further property that at every node neither child is smaller than the value in that node (or equivalently, both children are at least as large as that node).
- This property is called a ***partial ordering*.**
- As a result of this partial ordering, every path from the root to a leaf visits nodes in a non-decreasing order.
- What other properties of the Min Binary Heap result from this property?

Min Binary Heap Performance

- Performance (n is the number of elements in the heap)
  - construction O(n)
  - findMin O(1)
  - insert O(lg n)
  - deleteMin O(lg n)
- Heap efficiency results, in part, from the implementation
  - Conceptually a complete binary tree
  - Implementation in an array/vector (in level order) with the root at index 1

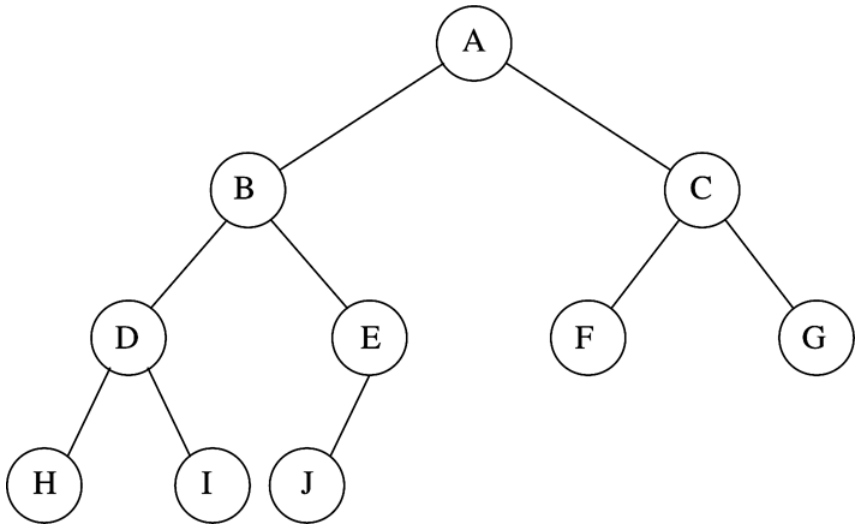
Min Binary Heap Properties

- For a node at index i
  - its left child is at index 2i
  - its right child is at index 2i+1
  - its parent is at index $\lfloor i/2 \rfloor$
- No pointer storage
- Fast computation of 2i and $\lfloor i/2 \rfloor$ by bit shifting

  i << 1 = 2i

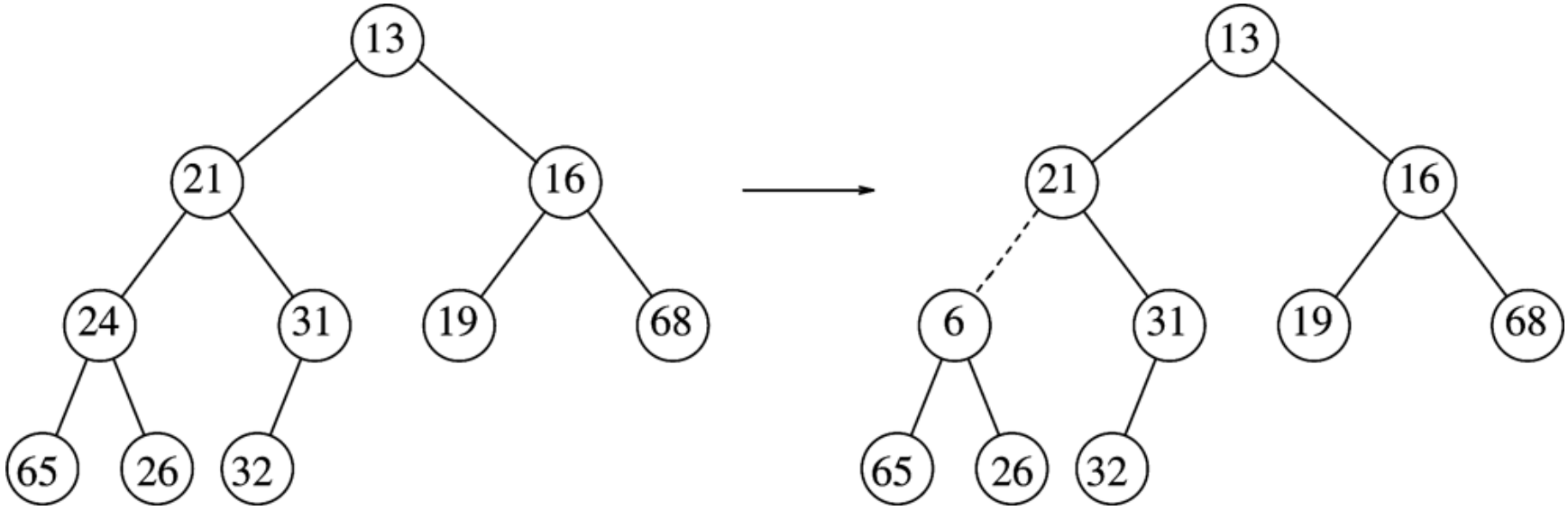
  i >> 1 = $\lfloor i/2 \rfloor$

Heap is a Complete Binary Tree



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

Which satisfies the properties of a Heap?



Min BinaryHeap Definition

```
public class
BinaryHeap<AnyType extends Comparable<? super AnyType>>
{
    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( )       { /* TBD */ }
    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){/* code online */}
}
```

Min BinaryHeap Implementation

```
public AnyType findMin( )
{
  if ( isEmpty( ) ) throw Underflow( );
      return array[1];
}
```

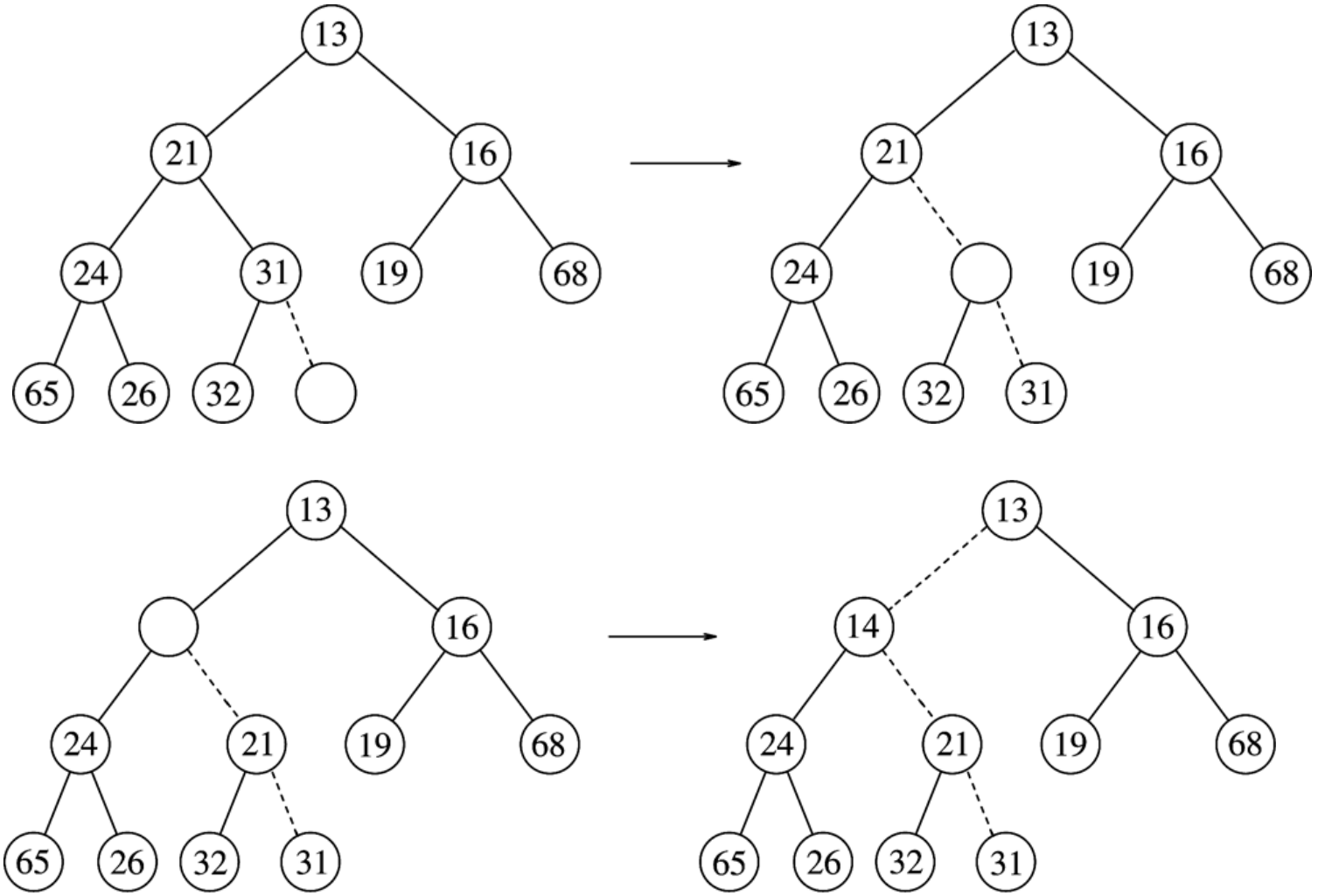
Insert Operation

- Must maintain
  - CBT property (heap shape):
    - Easy, just insert new element at “the end” of the array
  - Min heap order
    - Could be wrong after insertion if new element is smaller than its ancestors
    - Continuously swap the new element with its parent until parent is not greater than it
      - Called *sift up* or *percolate up*
- Performance of insert is O(lg n) in the worst case because the height of a CBT is O(lg n)

Min BinaryHeap Insert (cont.)

```
/**
     * 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( ; hole > 1&& x.compareTo(array[hole/2]) < 0; hole/=2 )
            array[ hole ] = array[ hole / 2 ];
   array[ hole ] = x;
}
```

Insert 14


13
21
16
24
31
19
68
65
26
32
13
21
16
24
19
68
65
26
32
31
13
16
24
21
19
68
65
26
32
31
13
14
16
24
21
19
68
65
26
32
31

Deletion Operation

- Steps
  - Remove min element (the root)
  - Maintain heap shape
  - Maintain min heap order
- To maintain heap shape, actual node removed is “last one” in the array
  - Replace root value with value from last node and delete last node
  - Sift-down the new root value
    - Continually exchange value with the smaller child until no child is smaller.

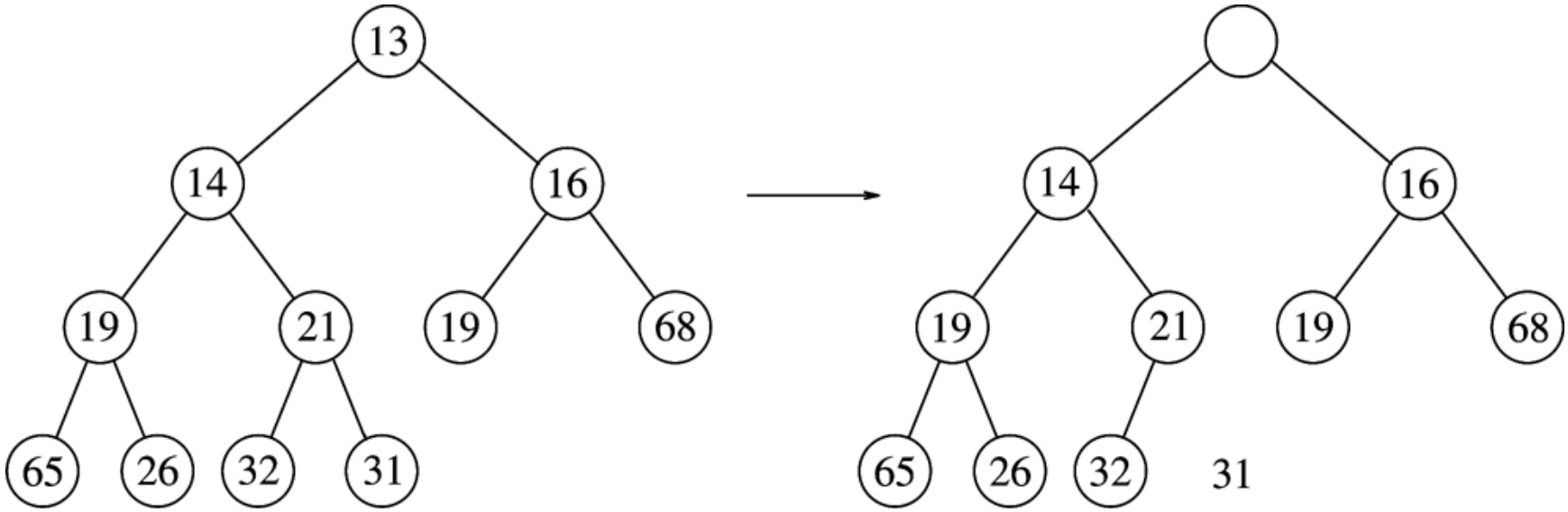
Min BinaryHeap Deletion(cont.)

```
/**
  * 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;
    }
```

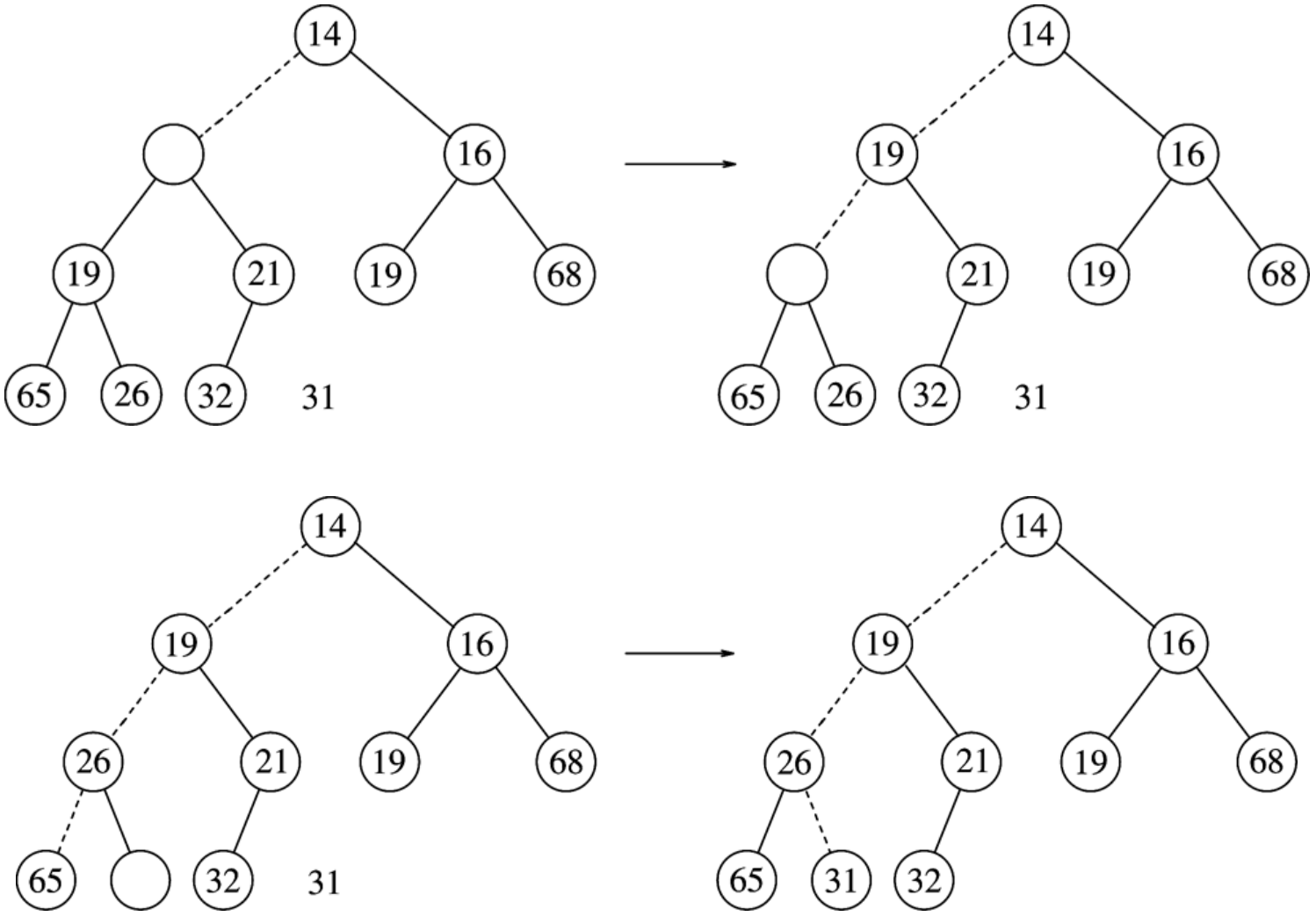
MinBinaryHeap percolateDown(cont.)

```
 /**
      * 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;
}
```

deleteMin



deleteMin (cont.)



Constructing a Min BinaryHeap

- A BH can be constructed in O(n) time.
- Suppose we are given an array of objects in an arbitrary order. Since it's an array with no holes, it's already a CBT. It can be put into heap order in O(n) time.
  - Create the array and store n elements in it in arbitrary order. O(n) to copy all the objects.
  - Heapify the array starting in the "middle" and working your way up towards the root

```
for (int index = ⌊n/2⌋ ; index > 0; index--)
  percolateDown( index );
```

Constructing a Min BinaryHeap(cont.)

```
//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 );
}
```

Performance of Construction

- A CBT has 2^{h-1} nodes on level h-1.
- On level h-l, at most 1 swap is needed per node.
- On level h-2, at most 2 swaps are needed.
- …
- On level 0, at most h swaps are needed.
- Number of swaps = S

$= 2^{h}*0 + 2^{h-1}*1 + 2^{h-2}*2 + \ldots + 2^{0}*h$

$= \sum_{i=0}^{h} 2^i (h-i) = h\sum_{i=0}^{h} 2^i - \sum_{i=0}^{h} i2^i$

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

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

$= 2^{h+1}-h-2$

Performance of Construction (cont.)

- But $2^{h+1}-h-2 = O(2^h)$
- But $n = 1 + 2 + 4 + \ldots + 2^h = \sum_{i=0}^{h} 2^i$
- Therefore, $n = O(2^h)$
- So $S = O(n)$

- A heap of n nodes can be built in O(n) time.

Heap Sort

- Given n values we can sort them in place in O(n log n) time
  - Insert values into array -- O(n)
  - heapify -- O(n)
  - repeatedly delete min -- O(lg n), n times
- Using a min heap, this code sorts in reverse (high down to low) order.
- With a max heap, it sorts in normal (low up to high) order.
- Given an unsorted array A[ ] of size n

```
for (i = n-1; i >= 1; i--)
{
        x = findMin( );
        deleteMin( );
        A[i+1] = x;
}
```

Limitations

- MinBinary heaps support `insert`, `findMin`, `deleteMin`, and `construct` efficiently.
- They do not efficiently support the `meld` or `merge` operation in which 2 BHs are merged into one. If H_1 and H_2 are of size n_1 and n_2, then the merge is in $O(n_1 + n_2)$.

Leftist Min Heap

- Supports
  - findMin -- O(1)
  - deleteMin -- O(lg n)
  - insert -- O(lg n)
  - construct -- O(n)
  - merge -- O(lg n)

Leftist Tree

- The ***null path length, npl(X),*** of a node, X, is defined as the length of the shortest path from X to a node without two children (a ***non-full*** node).
- Note that npl(NULL) = -1.

- A Leftist Tree is a binary tree in which at each node X, the null path length of X's right child is not larger than the null path length of the X's left child .
  I.E. the length of the path from X's right child to its nearest non-full node is not larger than the length of the path from X's left child to its nearest non-full node.

- An important property of leftist trees:
  - At every node, the shortest path to a non-full node is along the rightmost path.

    "Proof": Suppose this was not true. Then, at some node the path on the left would be shorter than the path on the right, violating the leftist tree definition.

Leftist Min Heap

- A ***leftist min heap*** is a leftist tree in which the values in the nodes obey heap order (the tree is partially ordered).
- Since a LMH is not necessarily a CBT we do not implement it in an array. An explicit tree implementation is used.
- Operations
  - findMin -- return root value, same as MBH
  - deleteMin -- implemented using meld operation
  - insert -- implemented using meld operation
  - construct -- implemented using meld operation

Merge

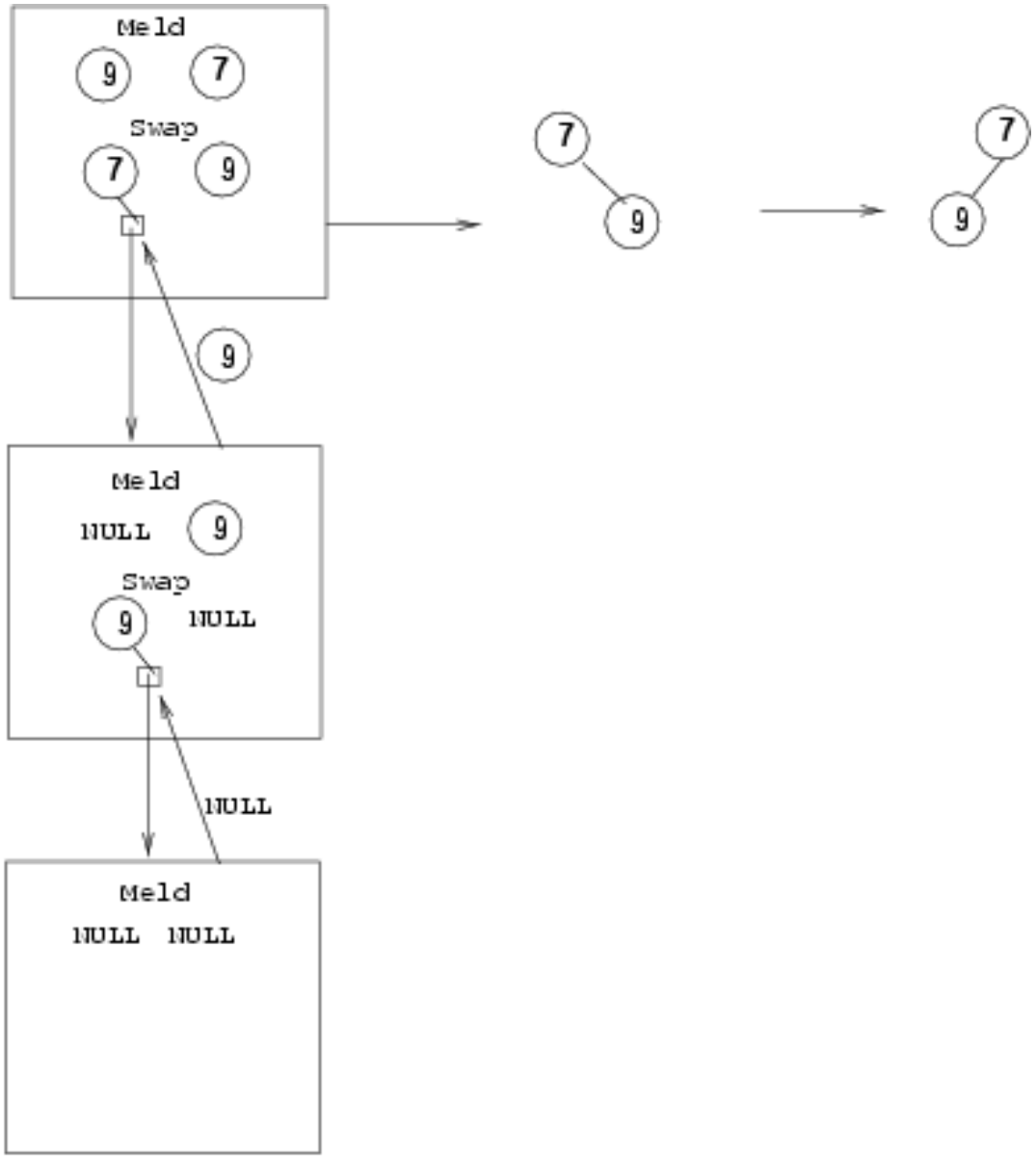
```
// Merge rhs into the priority queue.
// rhs becomes empty. rhs must be different from this.
// @param rhs the other leftist heap.
public void merge( LeftistHeap<AnyType> rhs ){
        if( this == rhs ) return; // Avoid aliasing problems
        root = merge( root, rhs.root );
        rhs.root =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

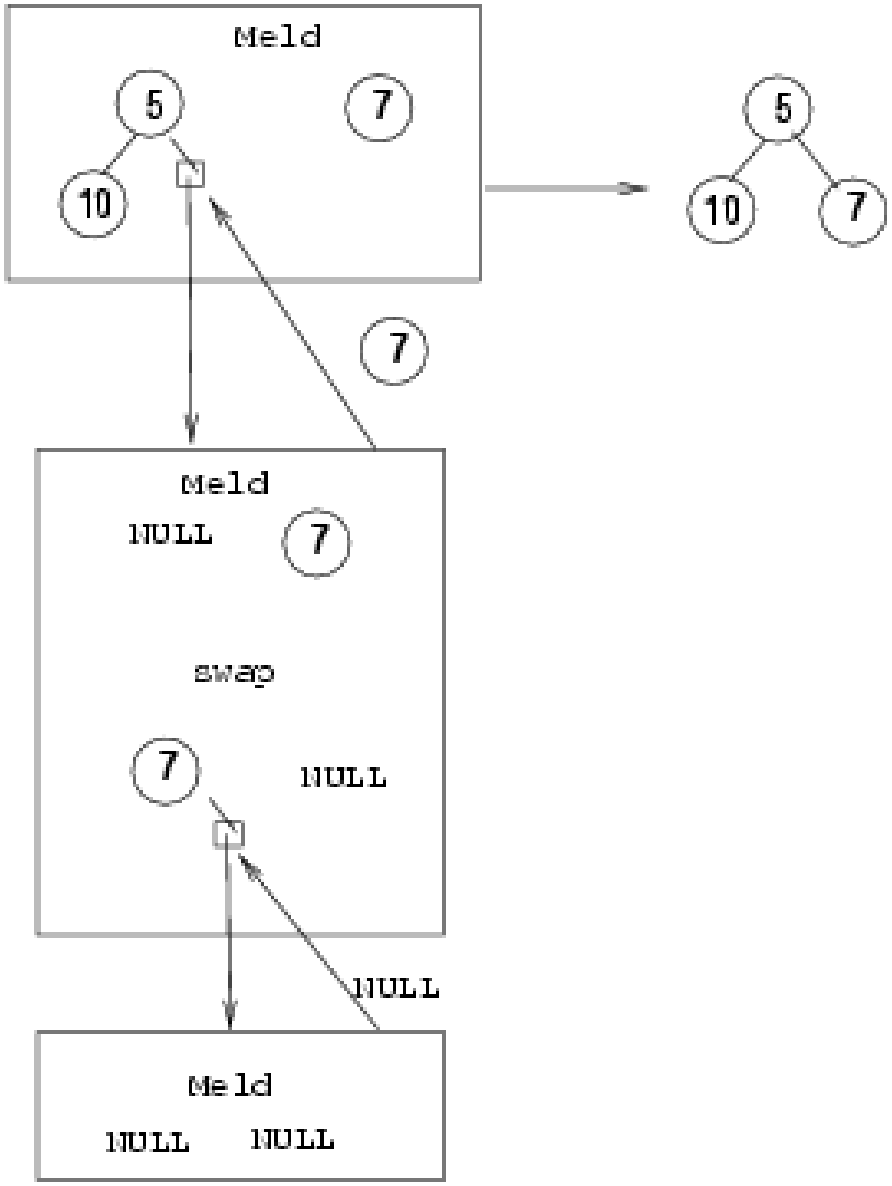
Merge (cont.)

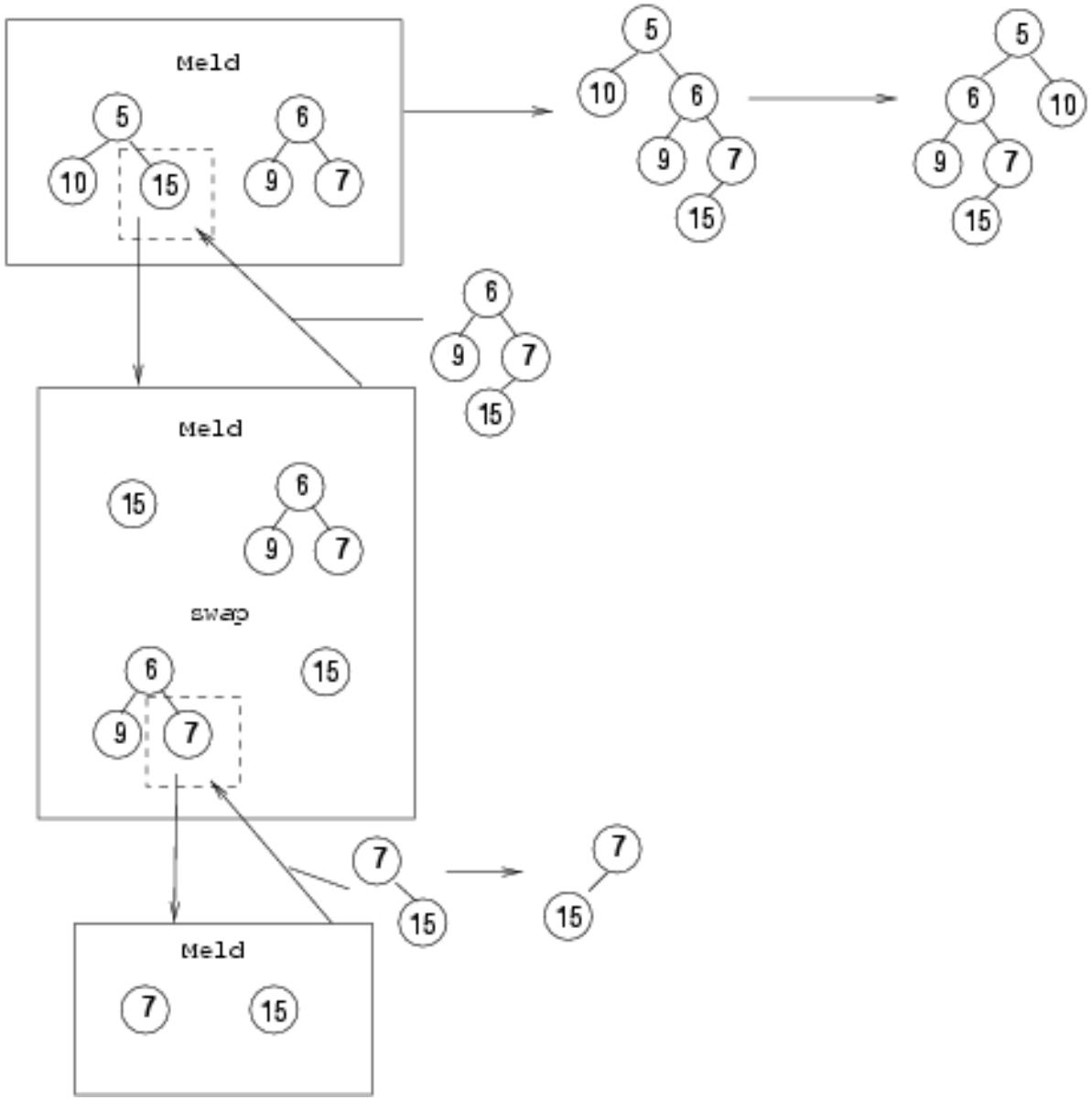
```
/**
* Internal method to merge two roots.
* Assumes trees are not empty, and h1's root contains smallest item.
*/
private Node<AnyType> merge1( Node<AnyType> h1, Node<AnyType> h2 )
{
        if( h1.left == null )   // Single node
            h1.left = h2;       // Other fields in h1 already accurate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

Merge (cont.)

- Performance: O(lg n)
  - The rightmost path of each tree has at most $\lfloor \lg(n+1) \rfloor$ nodes. So O(lg n) nodes will be involved.

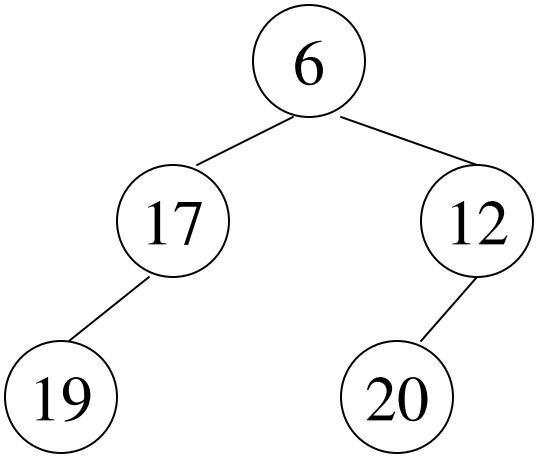

Meld
9
7
Swap
7
9
9
Meld
NULL
9
Swap
9
NULL
NULL
Meld
NULL NULL
7
9
7
9

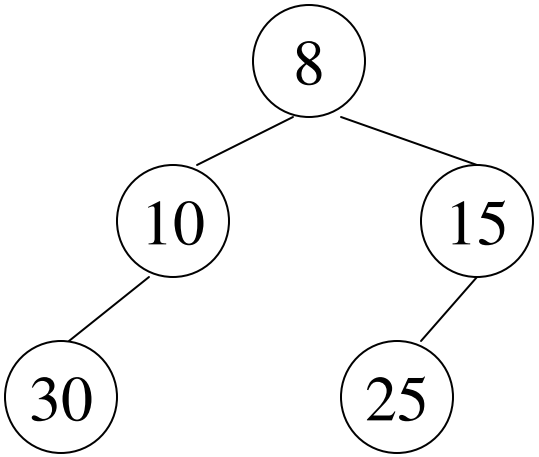

Meld
5
7
10
5
10
7
7
Meld
NULL
7
swap
7
NULL
NULL
Meld
NULL
NULL


Meld
5
10
15
6
9
7
5
10
6
9
7
15
5
6
10
9
7
15
6
9
7
15
Meld
15
6
9
7
swap
6
9
7
15
7
15
7
15
Meld
7
15

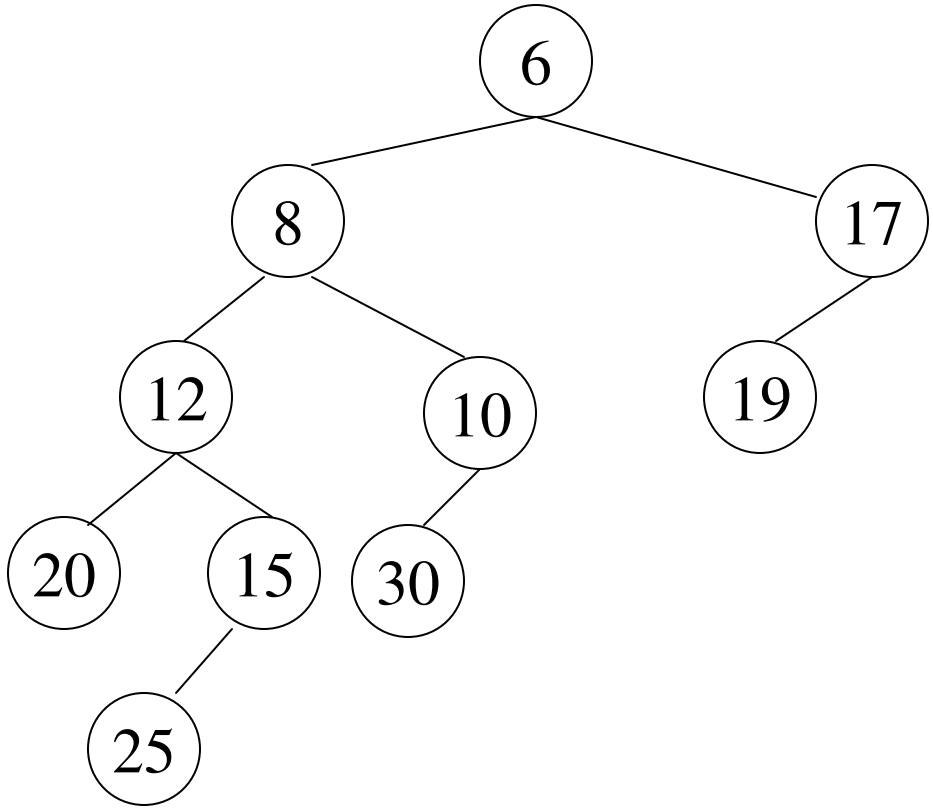
Student Exercise

- Show the steps needed to merge the Leftist Heaps below. The final result is shown on the next slide.





Student Exercise Final Result



Min Leftist Heap Operations

- Other operations implemented using Merge( )
  - insert (item)
    - Make item into a 1-node LH, X
    - Merge(this, X)
  - deleteMin
    - Merge(left subtree, right subtree)
  - construct from N items
    - Make N LHs from the N values, one element in each
    - Merge each in
      - one at a time (simple, but slow)
      - use queue and build pairwise (complex but faster)

LH Construct

- Algorithm:

  Make n leftist heaps, $H_1 \ldots H_n$ each with one data value

  Instantiate Queue<LeftistHeap> q;

  for (i = 1; i <= n; i++)

  q.enqueue(H_i);

  Leftist Heap h = q.dequeue( );

  while (!q.isEmpty( ))

  q.enqueue(merge(h, q.dequeue( )));

  h = q.dequeue( );