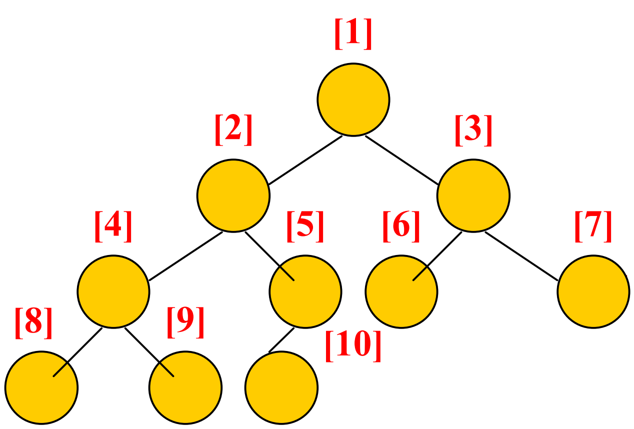
HEAPS

Heaps in Theory

- uses a graphical tree to ***represent*** un unsorted array
- the tree
  - is a RBT (Regular Binary Tree)
    - so only 2 children
  - ***is completed from the top down, left to right (called complete)***
  - each node in the tree represented in the corresponding array
- cannot have duplicates
- items are added to the array in order (or make a COMPLETE tree)
- order of inputs does have an effect on the overall order of the heap

Tree Representation



// JUST A GRAPHICAL REPRESENTATION!!!

Array Representation

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
-1	**29**	**85**	**8**	**93**	**23**	**88**	**73**	**44**	**36**

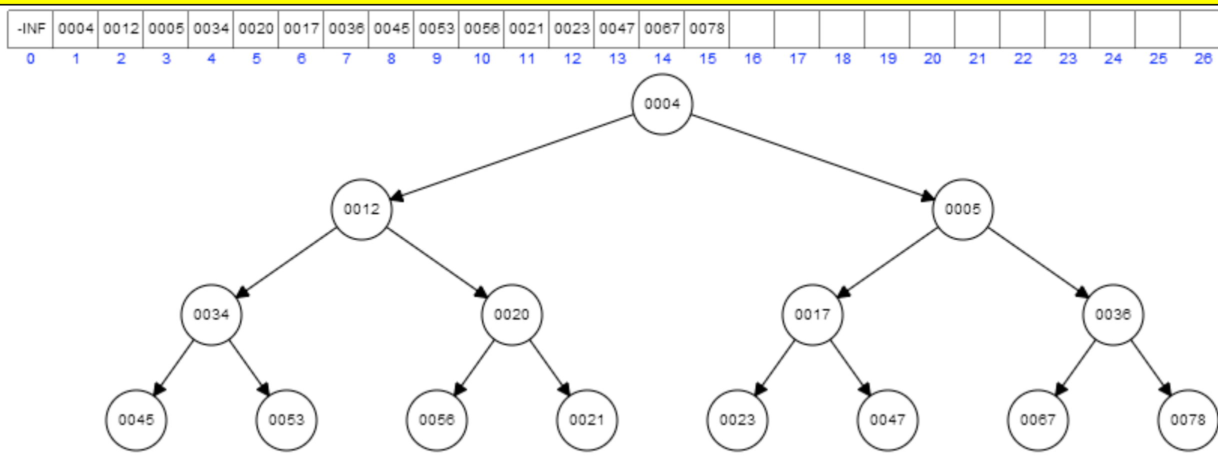
// Draw the value in the elements of the tree from the array representation

Was does complete mean?

Minimum Binary Heap

- same constructions as a heap, but the minimum value of the ***entire tree*** is stored at the root
- the further down we go in the min heap, the value increases
  - parent will ALWAYS be ***less than or equal*** in value than the kids
  - this is called partial ordering

The Min Heap Structure



Notice 4 is the smallest value so far in this heap
Anything below the parent (no matter where) is >= than the parent
Notice the max value will be SOMEWHERE near the bottom

Initial class setup – BinaryHeap

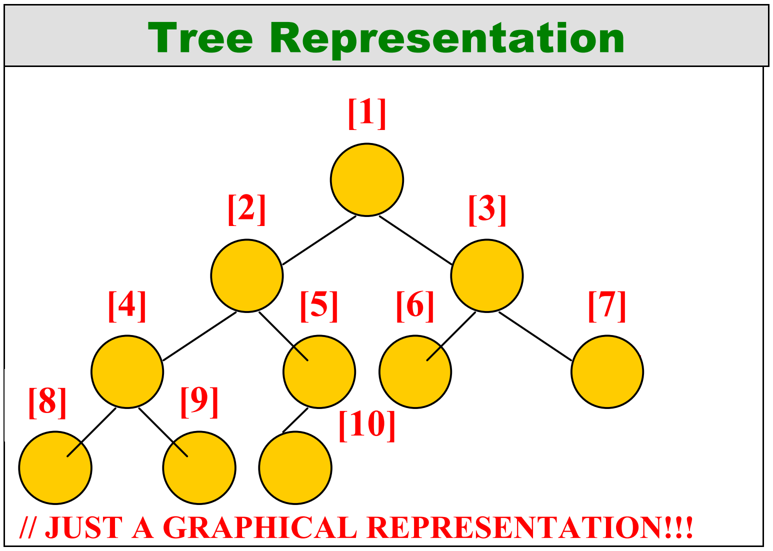
- code given uses an array
  - default size is 10
  - calls buildHeap() just to do that

MinBH Construct(or)

```
/**
 * 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( );
}
```

Determining the relationships using code/array

- Determining who is parent/child of a certain node is easy!!
  - using array notation and structure!!



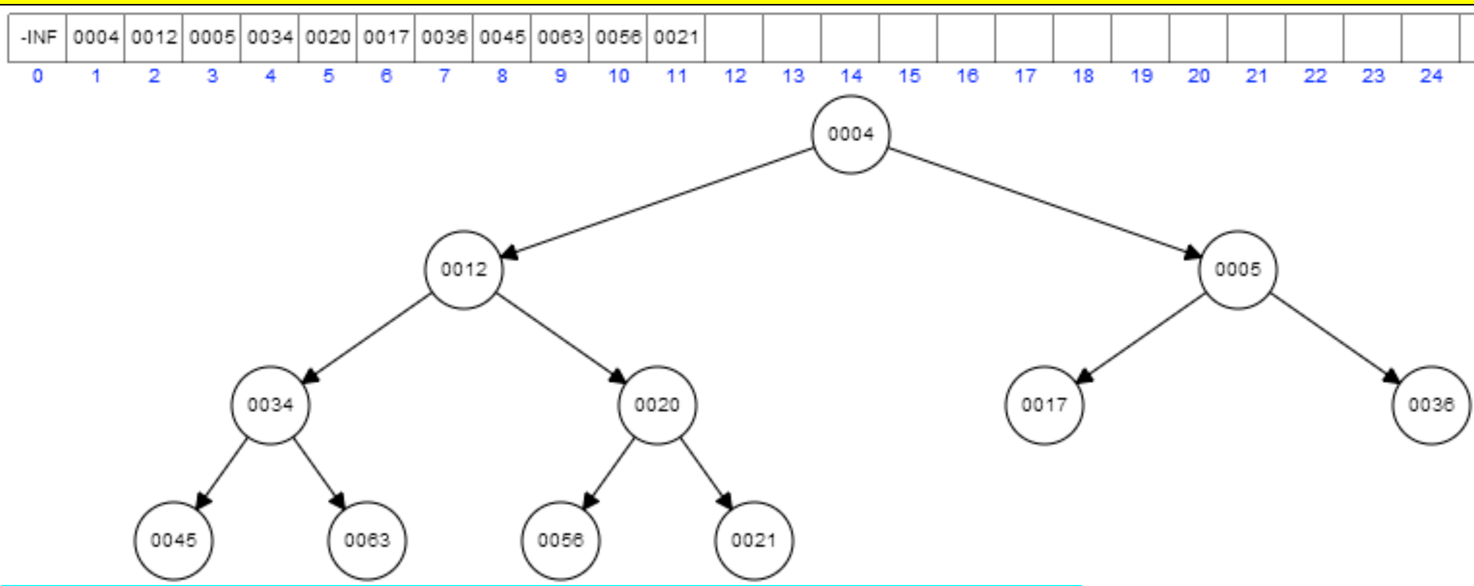
Remember this is using an ARRAY representation!!!

To Find	Formula	Example
Parent index	**floor**((index)/2)	[6]/2 = 3 6's Parent is 3
Left Child index	2(index)	2*[3] = 6 3's Left Child is 6
Right Child index	2(index) + 1	2*[3] + 1 = 7 3's Right Child is 7
9's Parent		
2's Left Child		
4's Parent		

Building and Inserting into a Min Heap

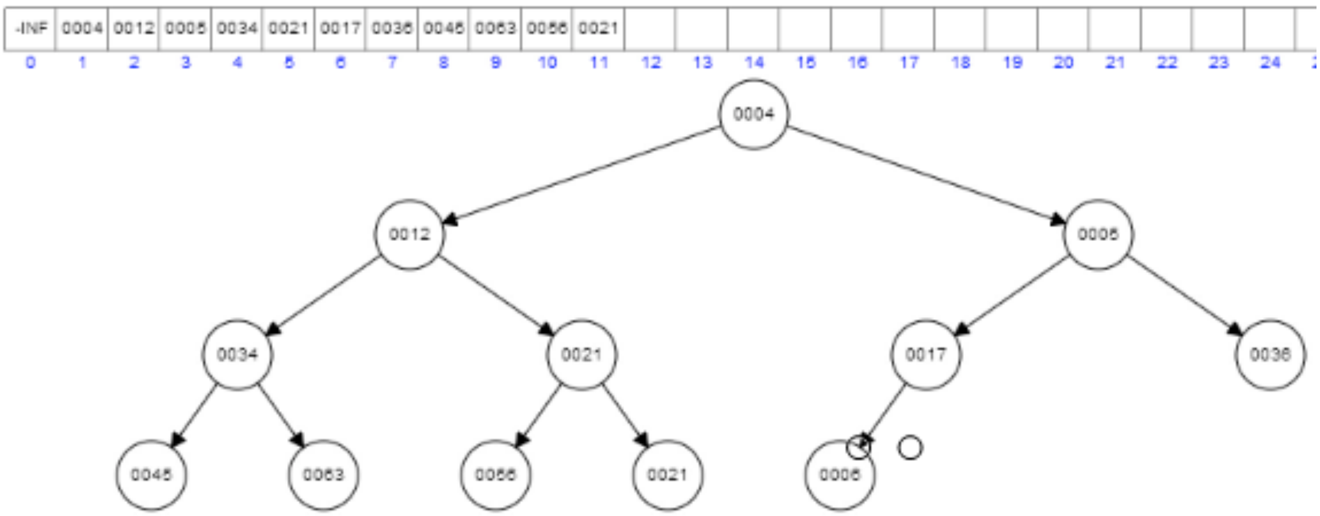
- notice I do have to ***<u>specify</u>*** Minimum Binary Heap
- algorithm
  - place new node at END of array
    - next available complete spot in BT
  - at end, could be in wrong order (parent is larger!)
    - continuously swap with parent going up the tree until parent < new node
    - this is called sift up or percolate up
  - notice that "lighter" values do bubble up, (maybe not to the root), but are in a higher position

Inserting into an establish Min Heap

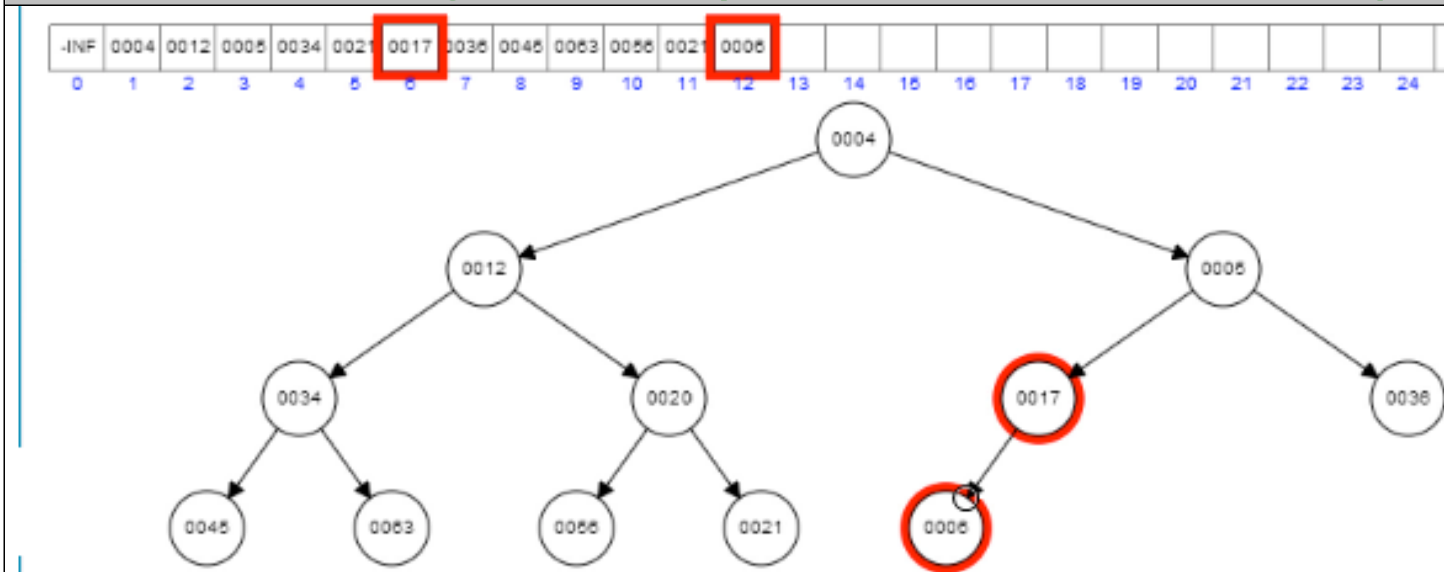


Where will the next value be first placed, no matter the value??

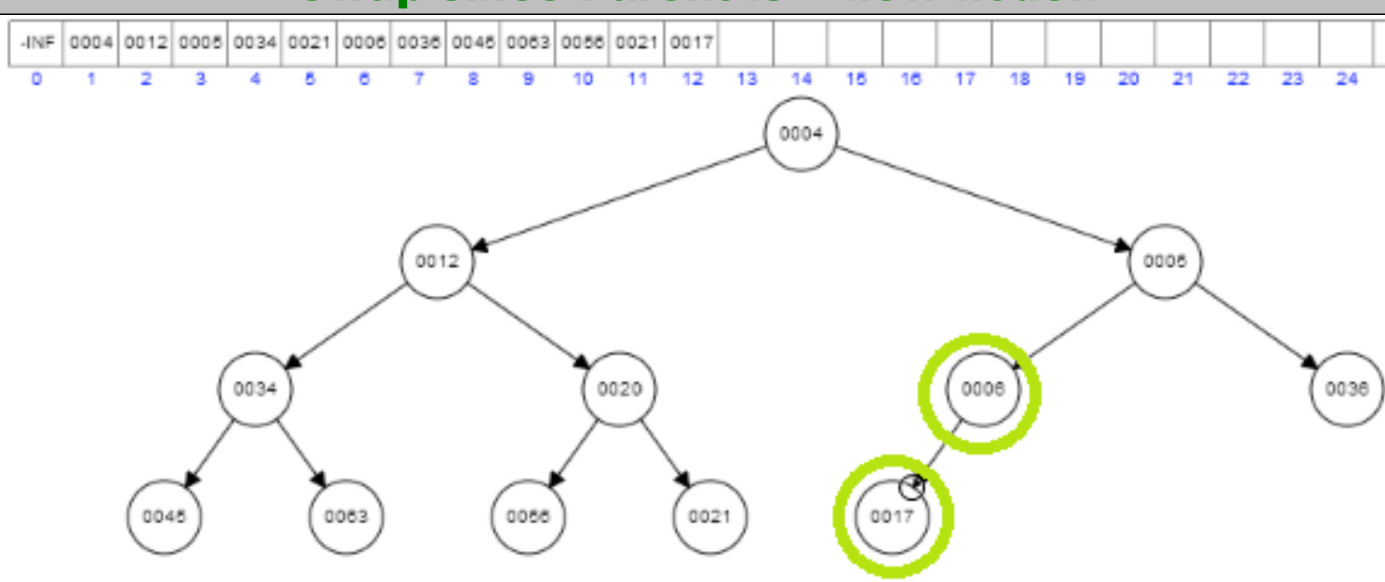
Inserting a 6



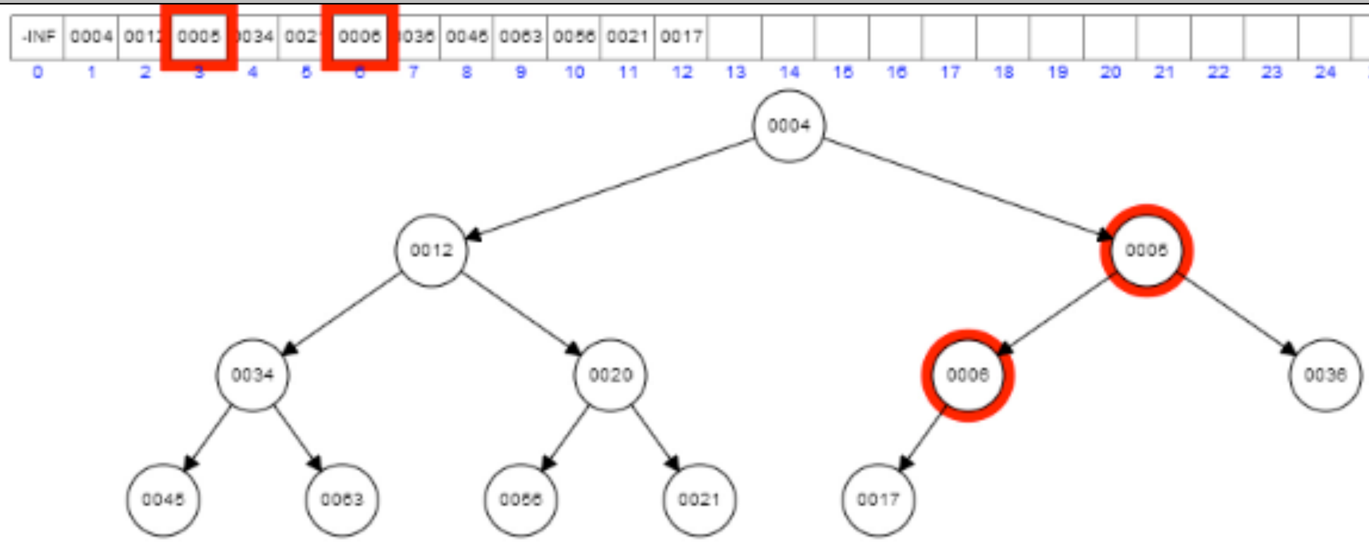
Start checking position!! (Check immediate Parent 17)



Swap since Parent is > new node!!



Start checking position!! (Check immediate Parent 17)

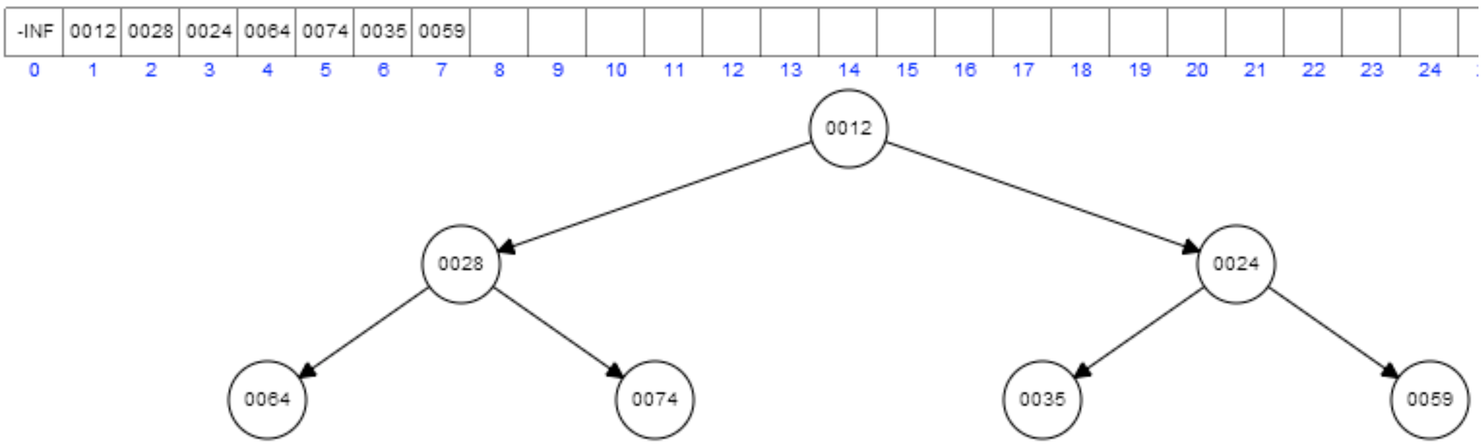


But 5 < 6, so we're good!!

I will try this one: (answer on next page)

64, 12, 35, 28, 74, 24, 59

Answer to in-class example



Try these on your own, insert in the order given:
1. 56, 43, 12, 67, 92, 4, 87, 53, 44, **93**
2. 61, 23, 57, 12, 68, 24, **14**, 96, 75, 63
3. If I added 4 to #2, how many swaps would take place?

Answers$_b$:

Insert – the function

- notice it checks the size of the array first
  - adds more if not enough
- ***temporarily*** spaces our value in [0]
- < 0 is not the value, but if there is a parent that is greater, then keep swapping

Array version of Insert for MinBH

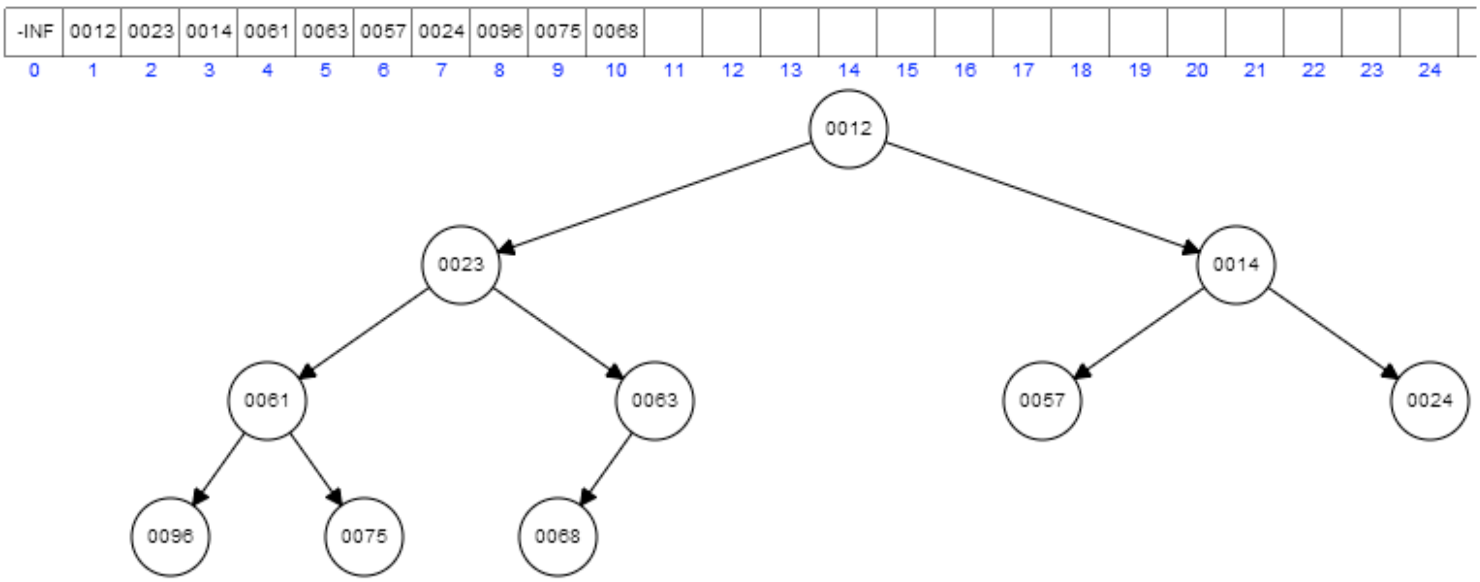
```
/**
 * Insert into the priority queue, maintaining heap order.
 * Duplicates are allowed.
 * @param x the item to insert.
 */
public void insert( AnyType x )
{
    // check if size of array is enough to hold new node
   if( currentSize == array.length - 1 )
         { enlargeArray( array.length * 2 + 1 ); }

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

    // now put our new value into the right place
    array[ hole ] = x;
}
```

Why is it /= 2?

Finding the minimum in a MinBH

- super easy!
- minimum value will ALWAYS be the root
  - if everything percolated correctly
  - [1] not [0]!!

Min is always at the top of a MinBH



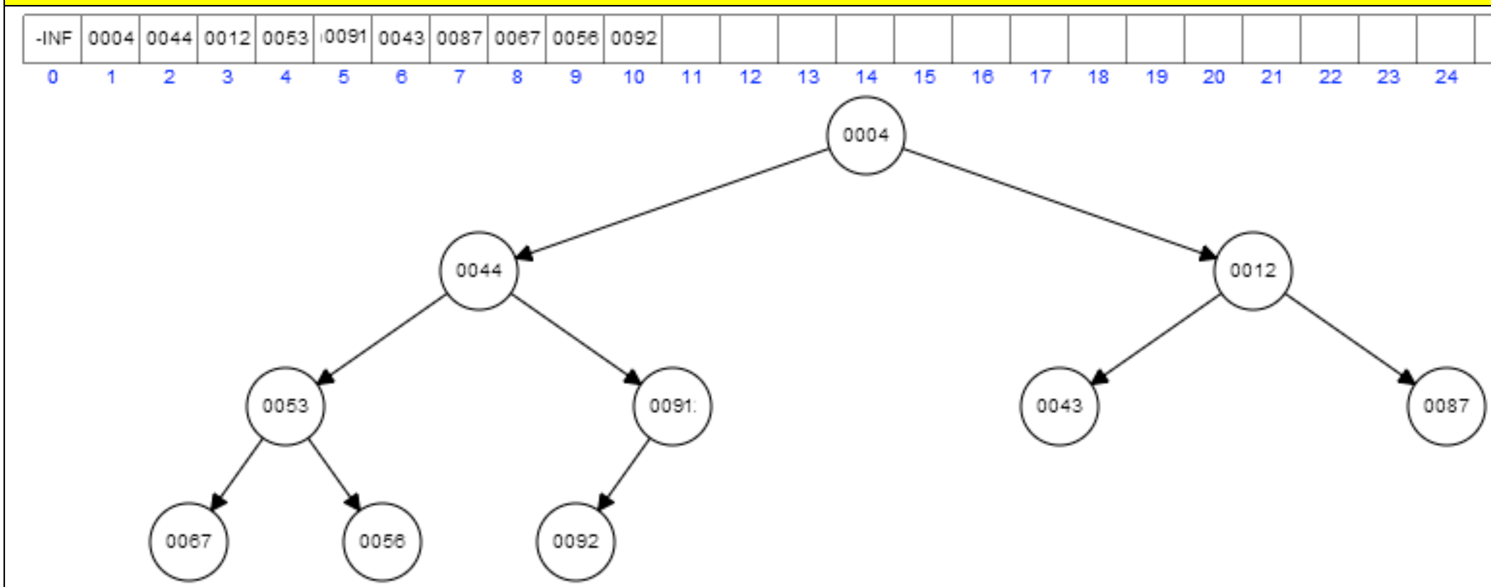
findMin the function

```
/**
 * Find the smallest item in the priority queue.
 * @return the smallest item, or throw an UnderflowException if empty.
 */
public AnyType findMin( )
{
    if( isEmpty( ) )
        { throw new UnderflowException( ); }
    return array[ 1 ];
}
```

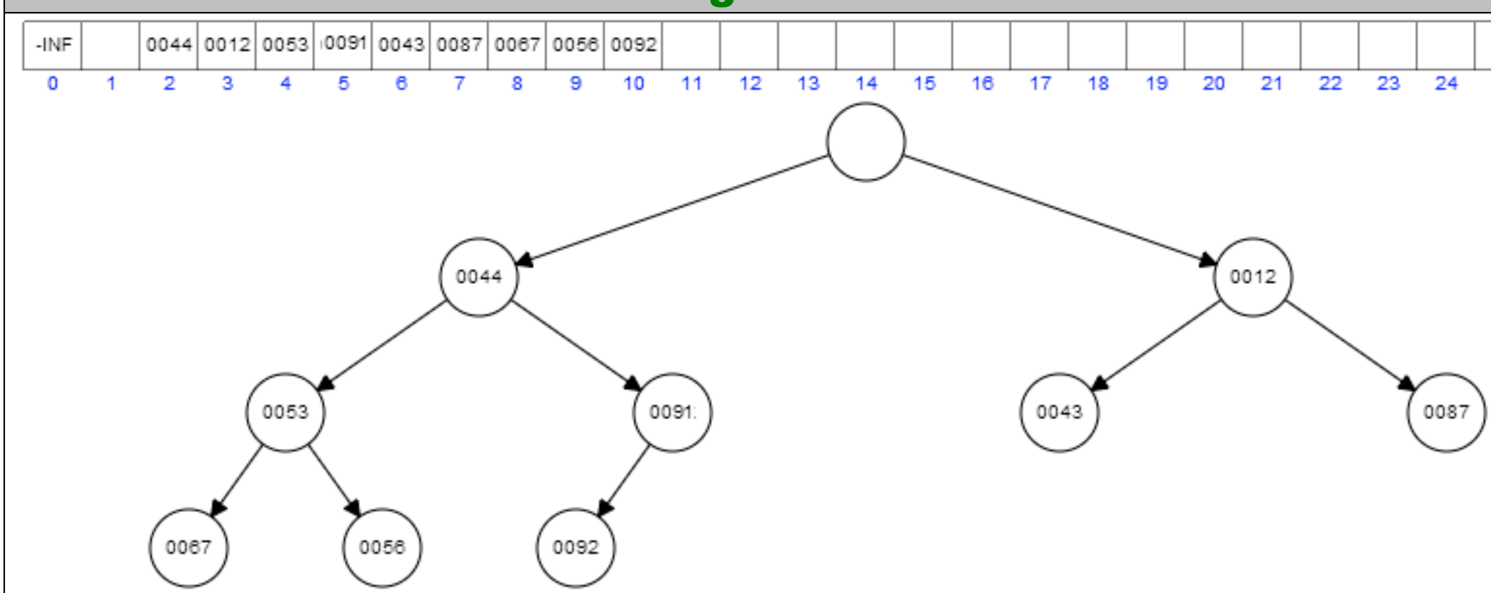
Deleting in a MinBH

- deletion is ***ONLY*** authorized for the MINIMUM value
    - not any other value
- we are not deleting the node, just replace the data inside
- now replaced with the NEXT lowest value
    - which SHOULD be close to the top of the tree
- tree must maintain it's shape
- but we will delete the LAST complete node in the tree since
    - since now it will be empty

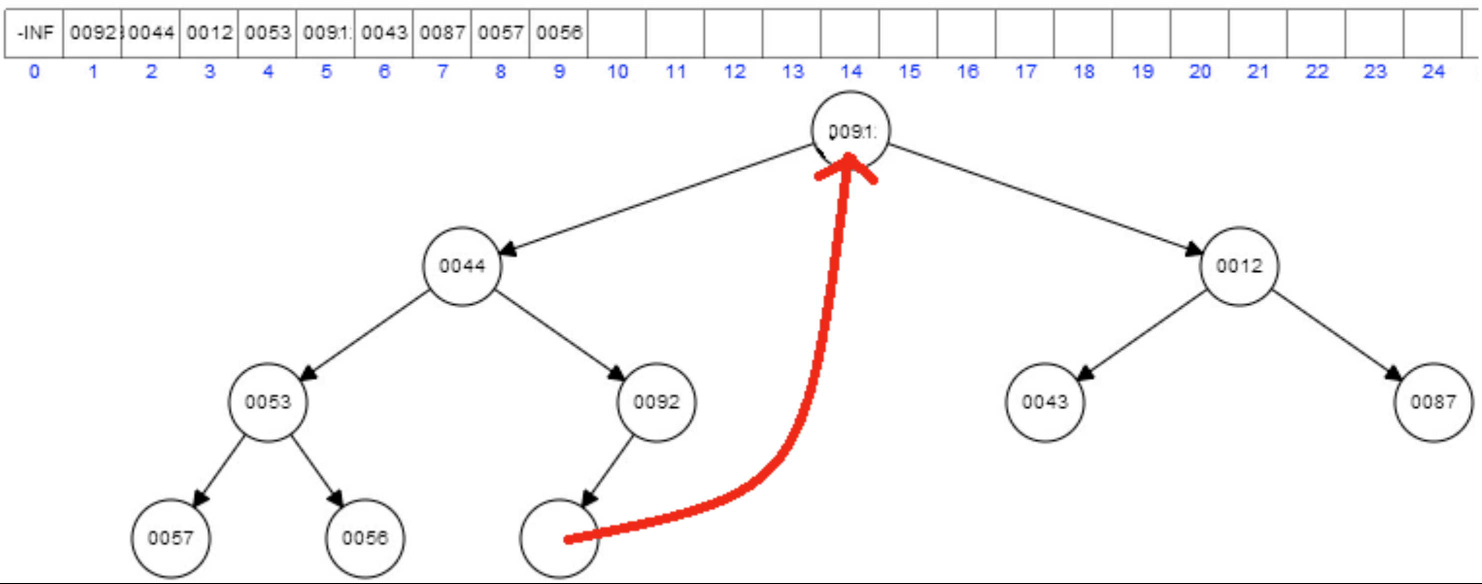
Deleting a node, and re-heaping



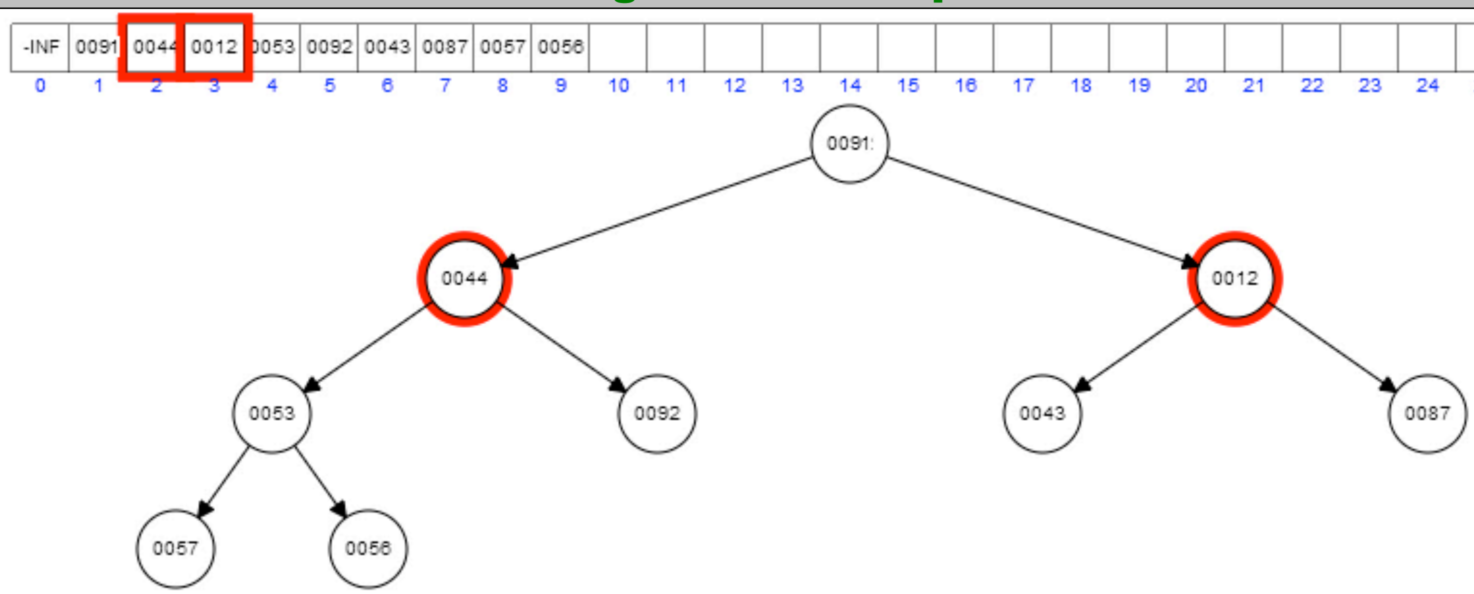
Deleting Min value



Replacing root with LAST value in tree

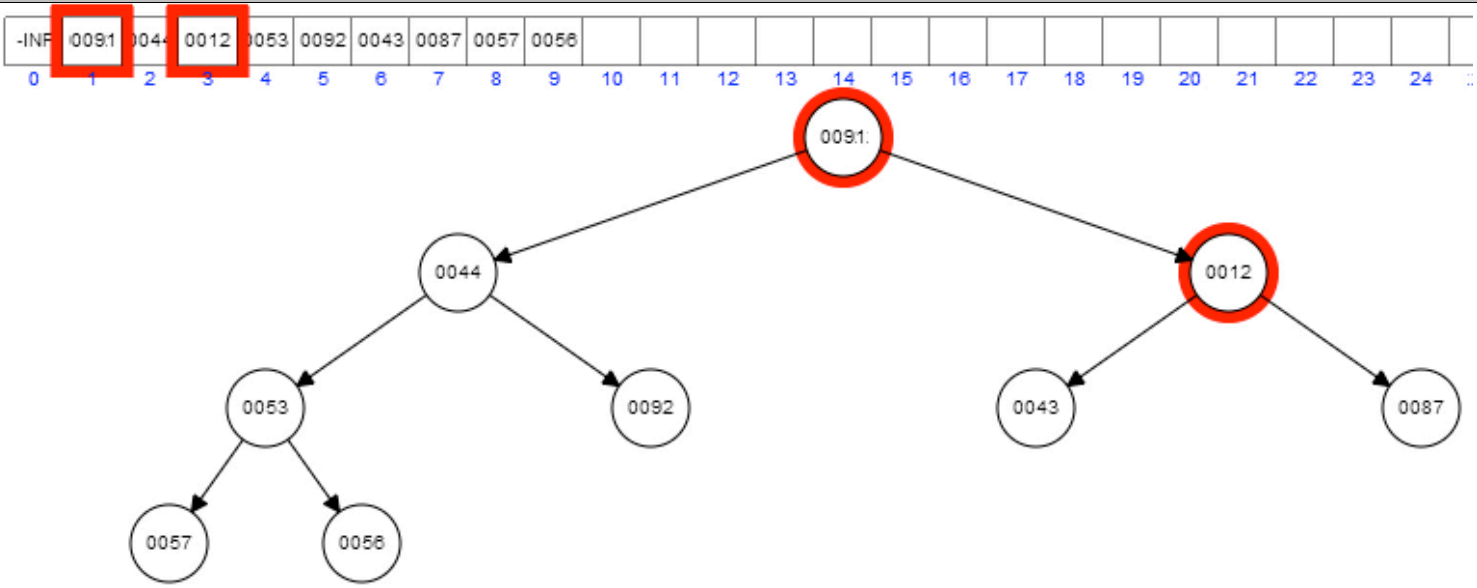


Percolating Down - Comparison

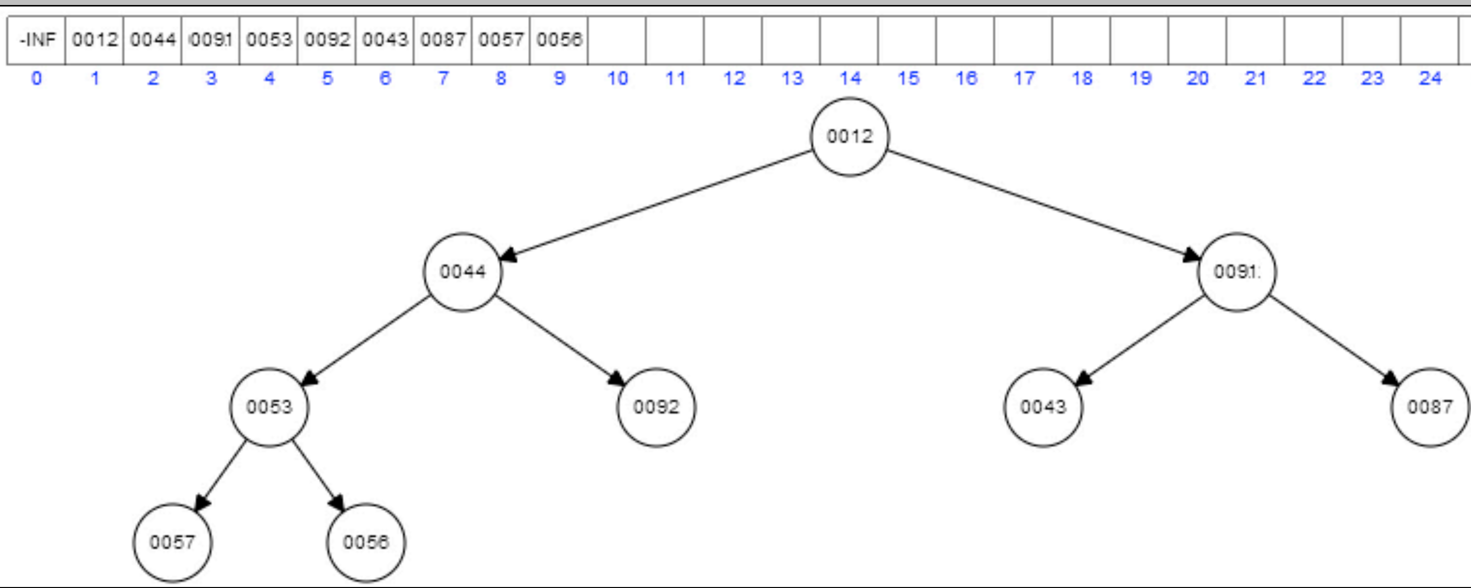


Comparing 91 (root) with whatever child is smallest

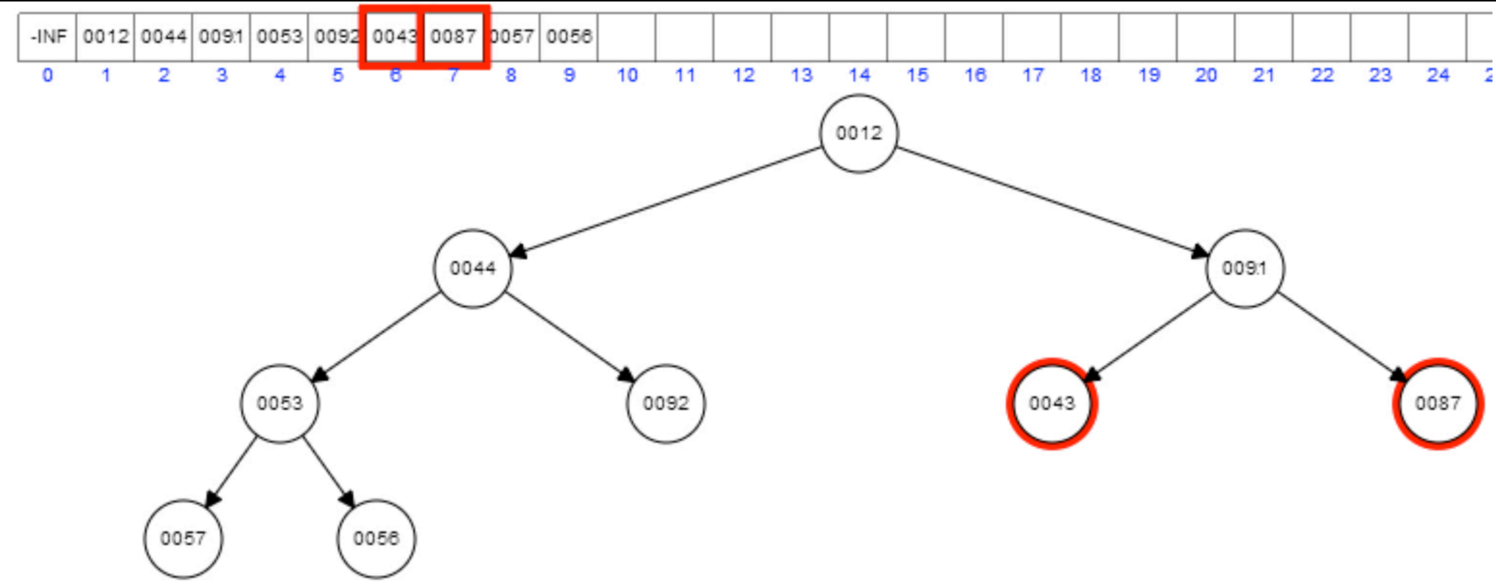
Percolating Down – Match



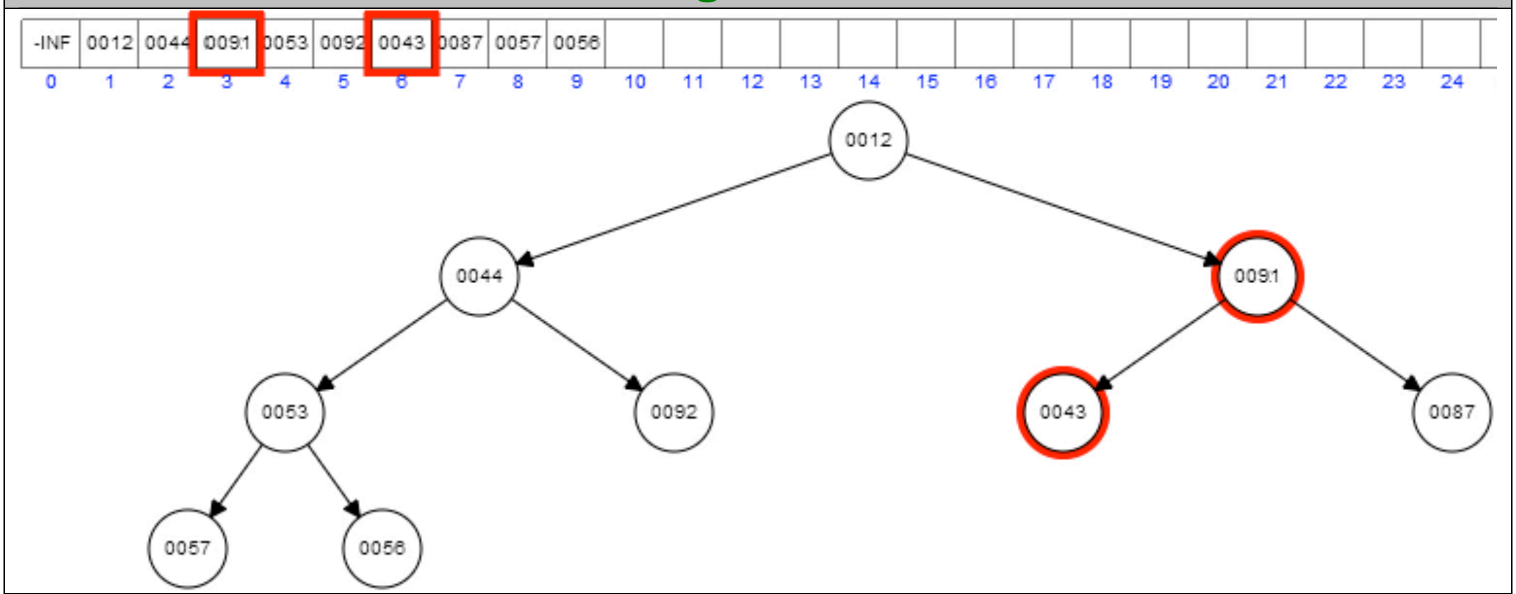
Percolating Down – Swap



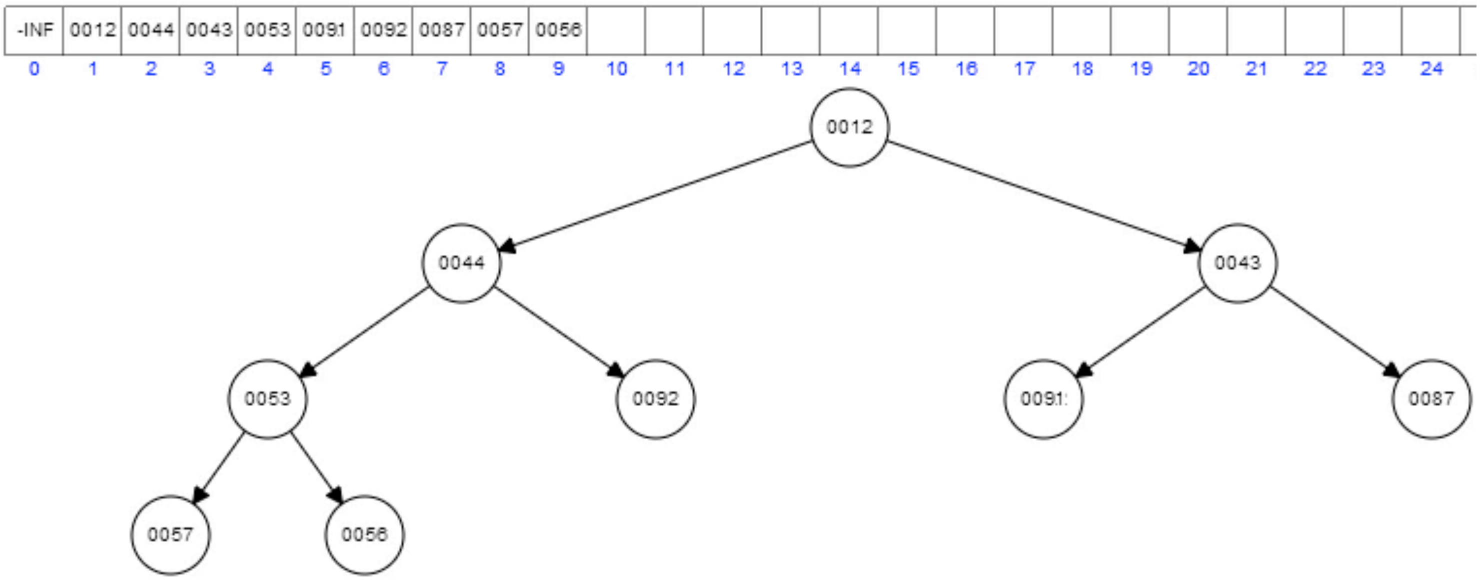
Percolating Down – Comparison



Percolating Down – Match



Percolating Down – Swap



Done since no more immediate nodes to compare to.

Delete the NEXT node using the result above.

After you’re done, click the link below for your answer:
http://userpages.umbc.edu/~slupoli/notes/DataStructures/videos/Heaps/Deleting%20from%20a%20Heap%20-%20Exercise.html

Delete – the function(s)

- deleteMin() and percolateDown()
    - deleteMin is the bootstrap to get things started
    - percolateDown is iterative in comparing and swapping
        - also called **heapify**

deleteMin() function

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

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

    return minItem;
}
```

percolateDown() function

```
/**
 * 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;
```

```
    }
```

Performance

- construction O(n)
  - even if data is out of order, we place in heap with **partial** ordering
  - still stored in a simple array!!
- findMin O(1)
- insert O(log n)
- deleteMin O(log n)

Heap Construction – the function

- lays all items into array first, no matter order in construction
  - done in constructor
- then "builds the heap" (sorts, partially) in buildHeap
  - notice that buildHeap uses percolateDown starting at middle of the array
  - this is enough to have the real minimum value "rise" to the top of the heap
- neither of these functions are recursive

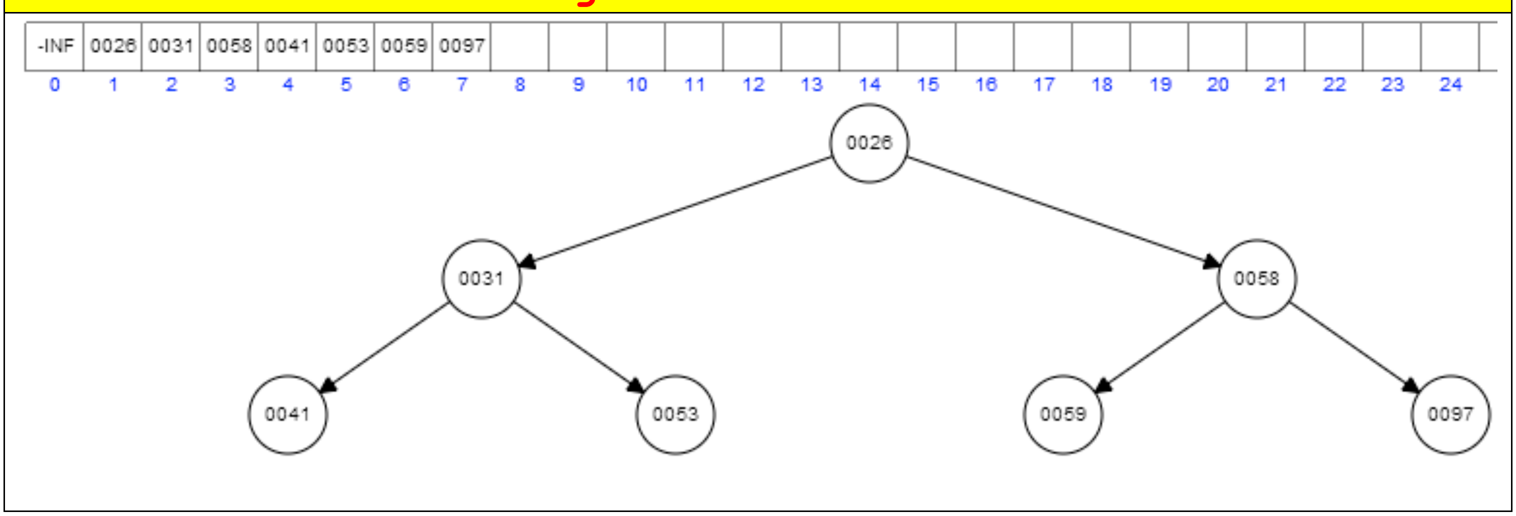
BuildHeap function

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

Sorting a Heap – MinBH

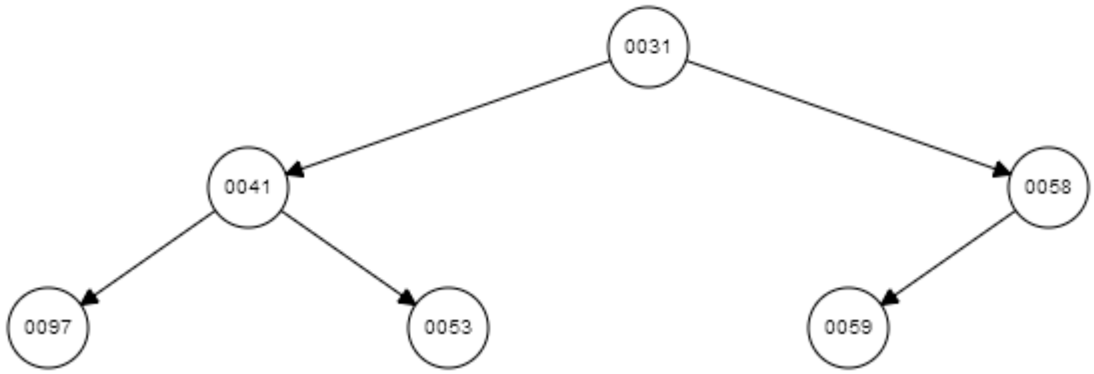
- given a list of n values, we can build and sort in O(n log n)
    - insert ***random*** values = O(n)
    - heapify = O(n)
    - repeatedly delete min and re-heapify O(log n) * n times
- heapify
    - re-ordering the values so Parent is <= it's kids in a MinBH
- delete
    - retrieves the CURRENT minimum node in a MinBH
        - value is saved in another array
    - ***automatically calls percolateDown()***
- this looped OVER and OVER will return a sorted list of items
- this means we will need another array of the same size, just to hold the cast offs
    - UNLESS, we store the values casted back in the deleted code's position
    - but this will have everything backwards!!

Using a MinBH to sort



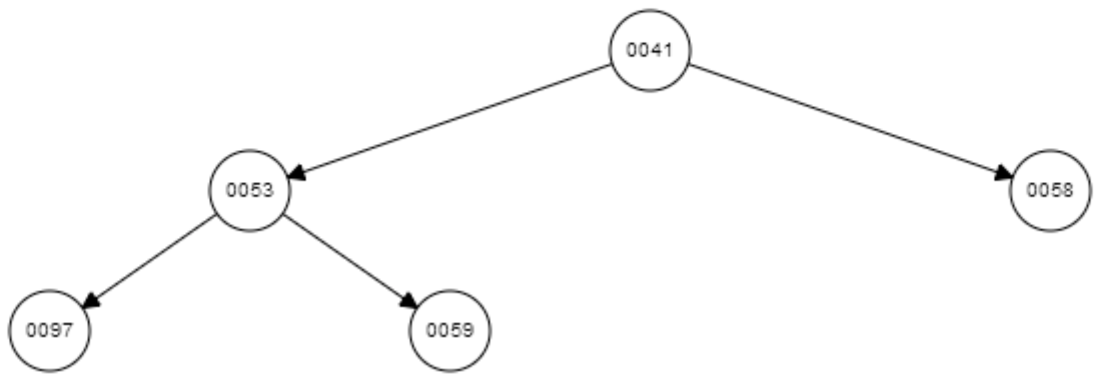
Take 26 (current Min), now heapify

-INF	0031	0041	0058	0097	0053	0059	26														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21



Take 31 (current Min) , now heapify

-INF	0041	0053	0058	0097	0059	31	26														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

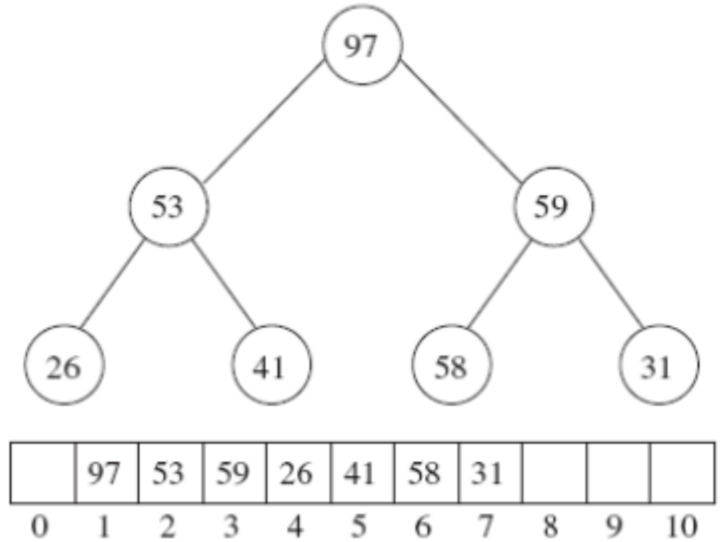


Perform the next two deletions on the heap above. Make sure to draw the tree AND the array

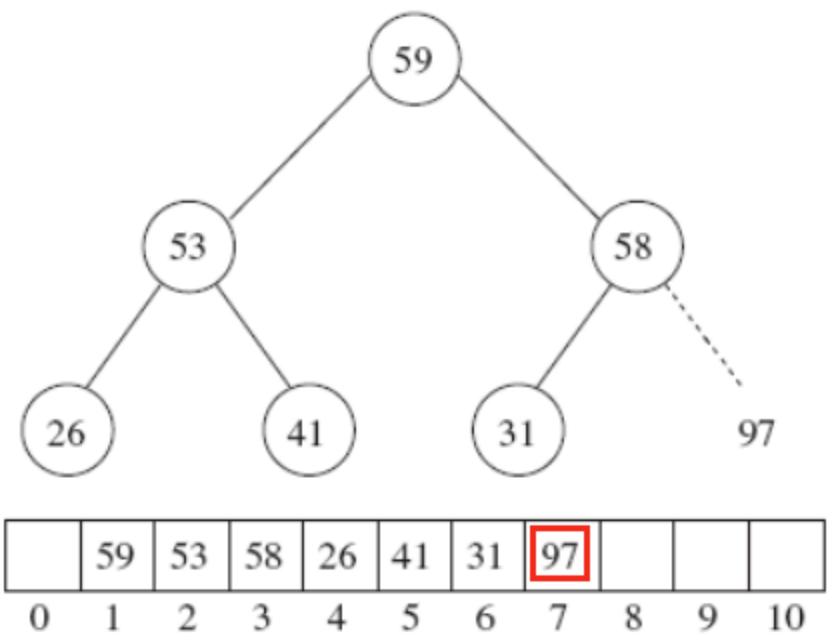
Sorting a Heap – MaxBH

- here we avoid the “backward” issue
- now the parent is >= it’s kids
  - so HIGHEST value is at the top of the heap

Max Heap Example



Deleting 97 (current Max), now heapify



Perform the next two deletions on the heap above

In it’s entirety

http://nova.umuc.edu/~jarc/idsv/lesson3.html

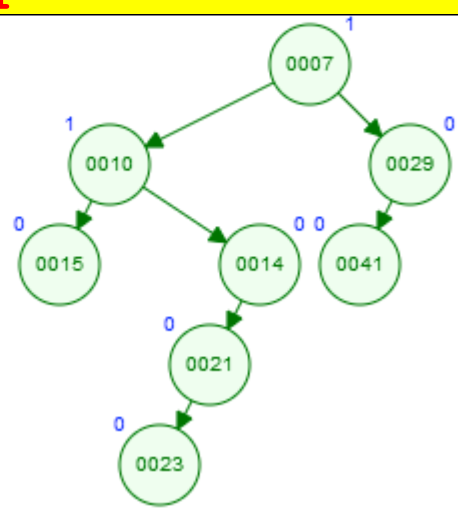
MinBH (using arrays) shortcomings

- sorting, wrong order
- merge
    - merging two arrays, no real shortcut
    - so $n_1 + n_2$

Leftist Min Heaps

- uses a BT!!
- merging heaps is much easier and faster
    - may use already established links to merge with a new node
        - why so much faster
    - ***because we are using Binary Trees!!***
- values STILL obey a heap order (partially ordered)
- uses a null path length to maintain the structure (covered later)
    - the null path of and node's ***left child is >= null path of the right child***
- at every node, the shortest path to a non-full node is along the rightmost path
- this overall ADT supports
    - findMin = O(1)
    - deleteMin = O(log n)
    - insert = O(log n)
    - construct = O(n)
    - merge = O(lg n)

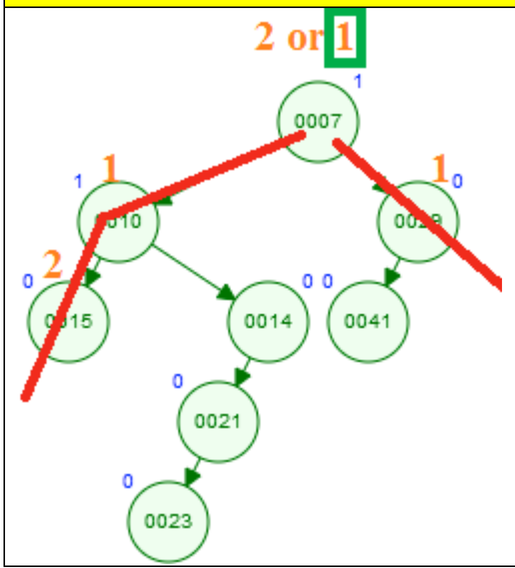
Example of a Leftist Heap

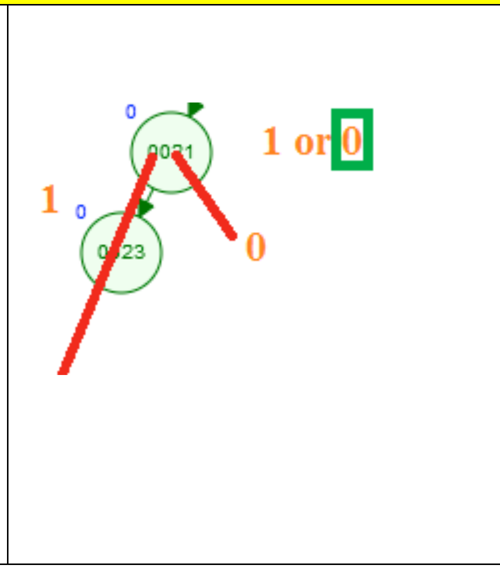


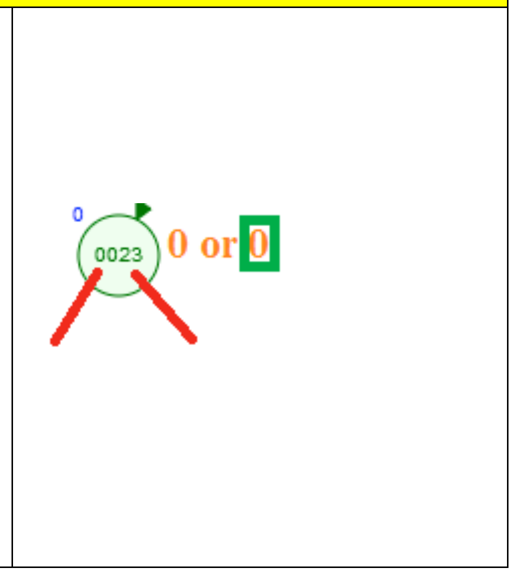
Null Path Length (npl)

- length of **shortest** path from current node (X) to a node **without** 2 children
  - value is store IN the node itself
- leafs = 0
- nodes with only 1 child = 0

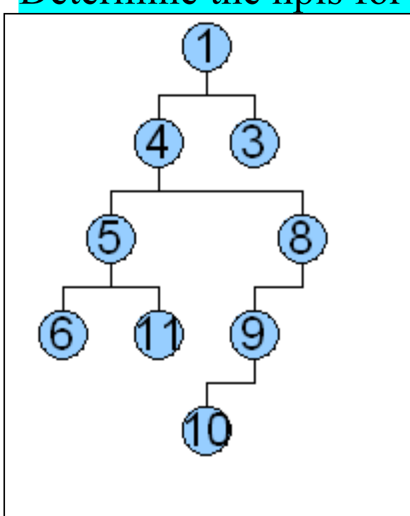
Determining the npl for a node

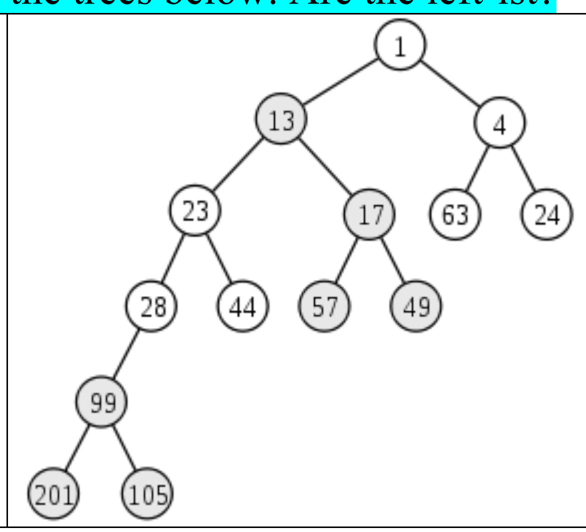


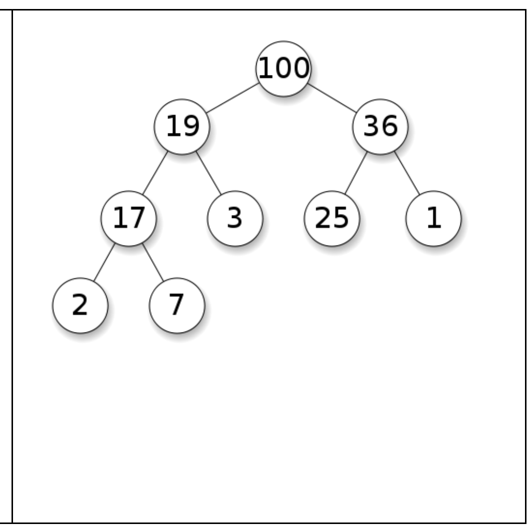




Determine the npls for the trees below. Are the left-ist?







The Leftist Node

- the node will have many data members this time
  - links (left and right
  - element (data)
  - npl
- by default, the LeftistHeap sets and empty one as the root

The leftist Node Class and Code

```
    private LeftistNode<AnyType> root;    // root

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

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

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

Building a Left-ist Heap

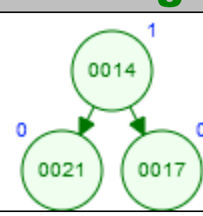
- value of node STILL matters, lowest value will be root, so still a min Heap
- data entered is random
- uses CURRENT npl of a node to determine where the next node will be placed
- algorithm
  - add new node to right-side of tree, in order
  - if new node is to be inserted as a parent (parent > children),
    - make new node parent
    - link children to it
    - link grandparent down to new node
  - if leaf, attach to right of parent
  - if no left sibling, push to left (hence left-ist)
    - why?? (answer in a second)
  - else left node is present, leave at right child
  - update all ancestors' npls
  - check each time that all nodes left npl < right npls
    - if not, swap children or node where this condition exists
- this is really using heaps and links!!

Building a leftist Heap

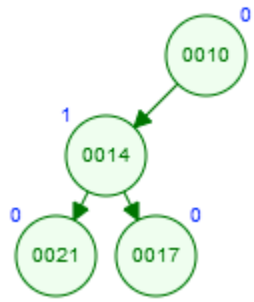
21, 14, 17, 10, 3, 23, 26, 8

inserting 14

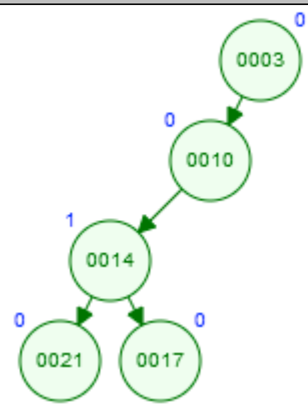
inserting 17



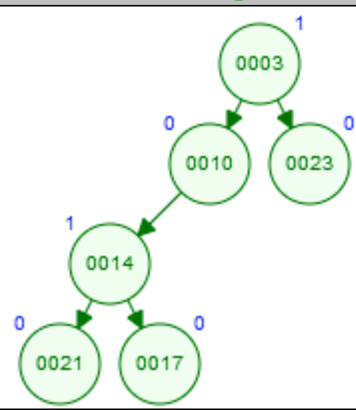
inserting 10



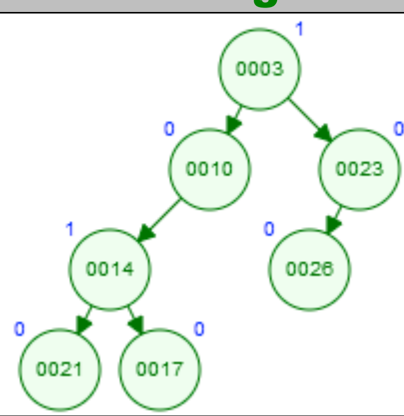
inserting 3



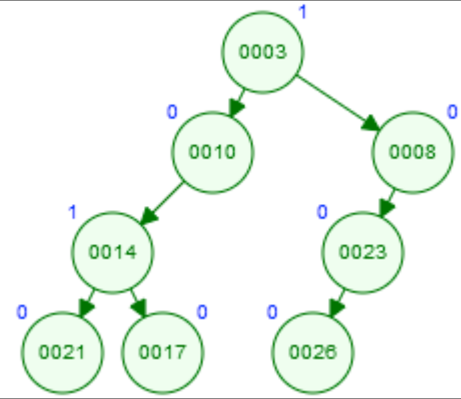
inserting 23



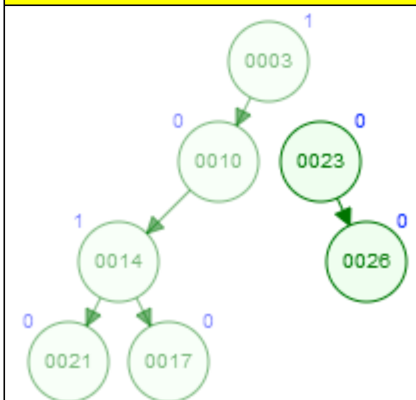
inserting 26



inserting 8



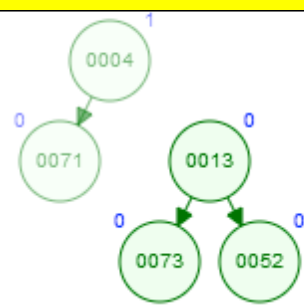
Why can we NOT have this??



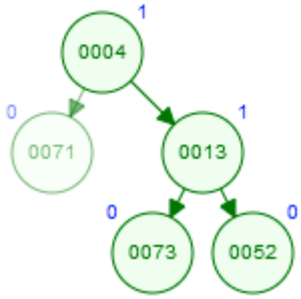
Why does 26 HAVE to be moved left?

- we can have it where a node's left npl is greater than it's right npl
  - simply, we swap children

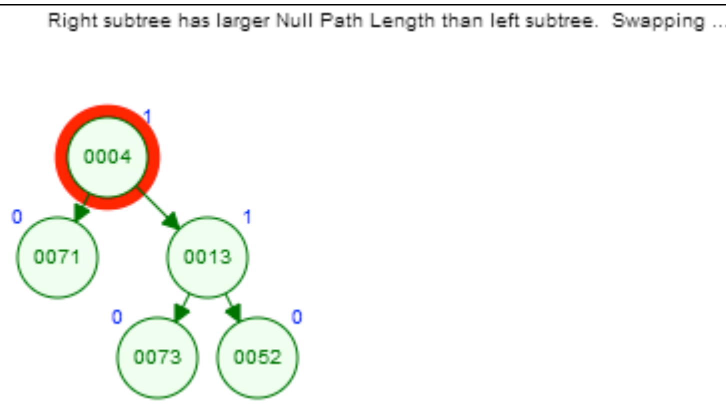
Swapping children to save a leftist tree

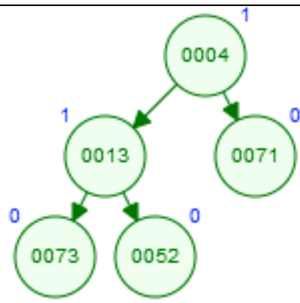


Just inserted 52 into the leftist heap



What's the issue now?





Try creating these leftist heaps n your own:

```
75, 91, 97, 9, 39, 87, 34, 8, 86, 58
24, 80, 98, 30, 77, 35, 65, 2, 48, 92, 18, 37, 67, 96
71, 4, 13, 73, 52, 20, 50, 63, 85, 23, 1, 44, 32, 53, 14, 17, 82, 76, 27, 83, 11, 81, 90, 62
```

Answers$_b$:

Inserting – the function

- in the code, adding a single node is treated at merging a heap (just one node) with an established heap's root
    - and work from that root as we just went over
- we will go over merging whole heaps momentarily

The Insert function

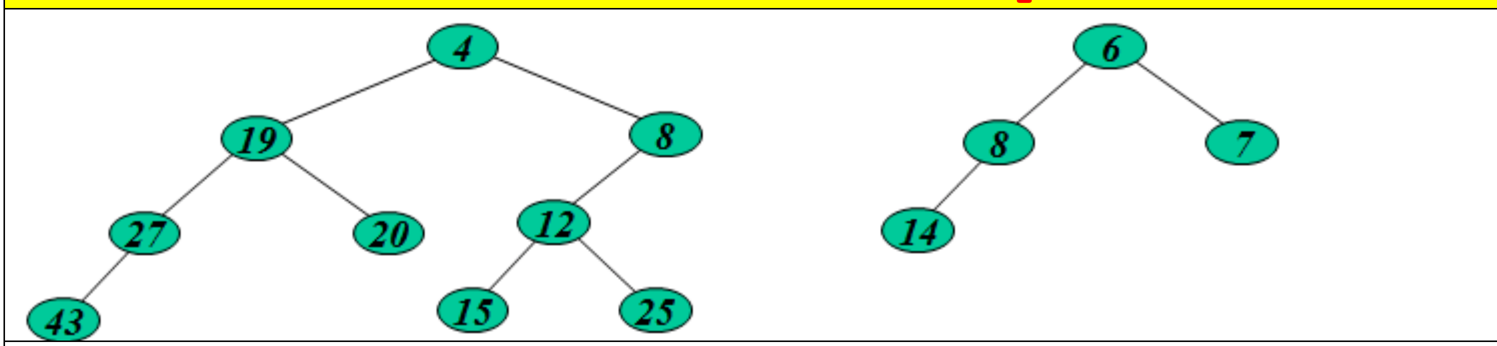
```
/**
 * Insert into the priority queue, maintaining heap order.
 * @param x the item to insert.
 */
public void insert( AnyType x )
{
    root = merge( new LeftistNode<>( x ), root );
 }
```

Merging Left-ist Heaps

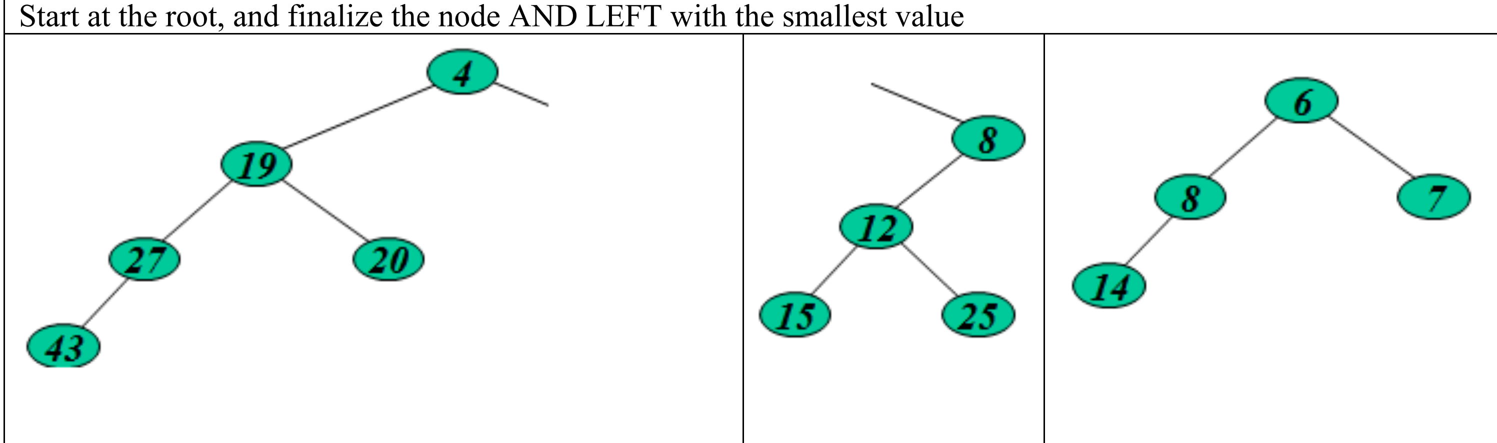
- the heaps we are about to merge must be left-ist
- at end we will get a heap that is
  - a min-heap
  - left-ist
- algorithm
  - Start at the (sub) root, and finalize the node AND LEFT with the smallest value
  - REPEADLY, until no lists left unmerged.
    - Start at the **rightmost** root of the sub-tree, and finalize the node AND LEFT with the **next** smallest value in leftist lists.
    - Add to RIGHT of finalized tree.
  - Verify that it is a Min Heap!! (Parent < Children)
  - Verify a leftist heap! (left npl <= right npl)
    - if not, swap troubled node with sibling

I will try:

Initial Left-ist Heaps

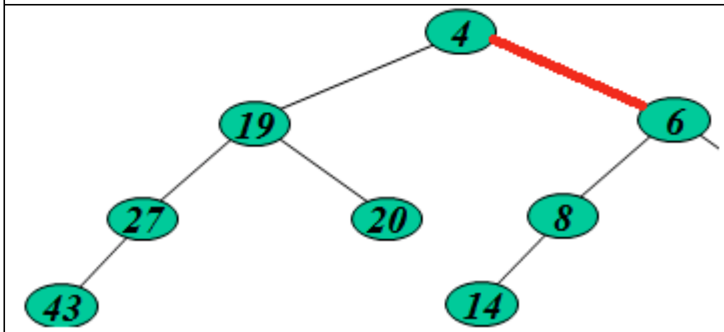


Start at the root, and finalize the node AND LEFT with the smallest value

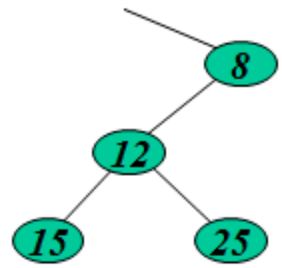


finalized	option 1	option 2

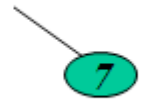
Start at the root of the sub-tree, and finalize the node AND LEFT with the **next** smallest value. Add to RIGHT of finalized tree



finalized

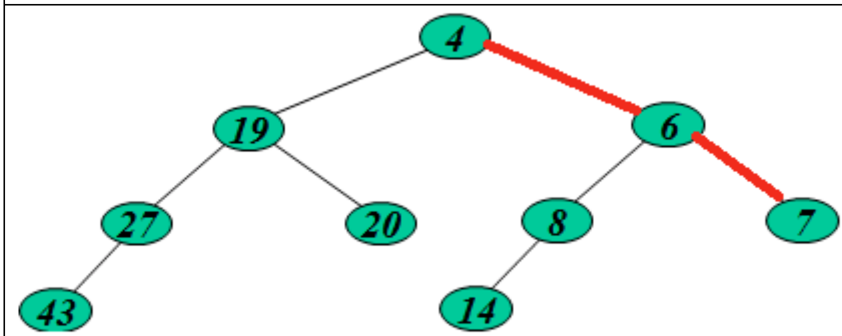


option1

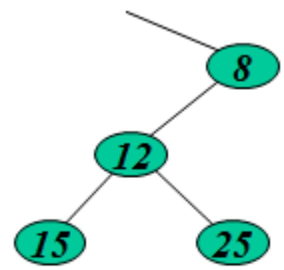


option2

Start at the root of the sub-tree, and finalize the node AND LEFT with the **next** smallest value. Add to RIGHT of finalized tree



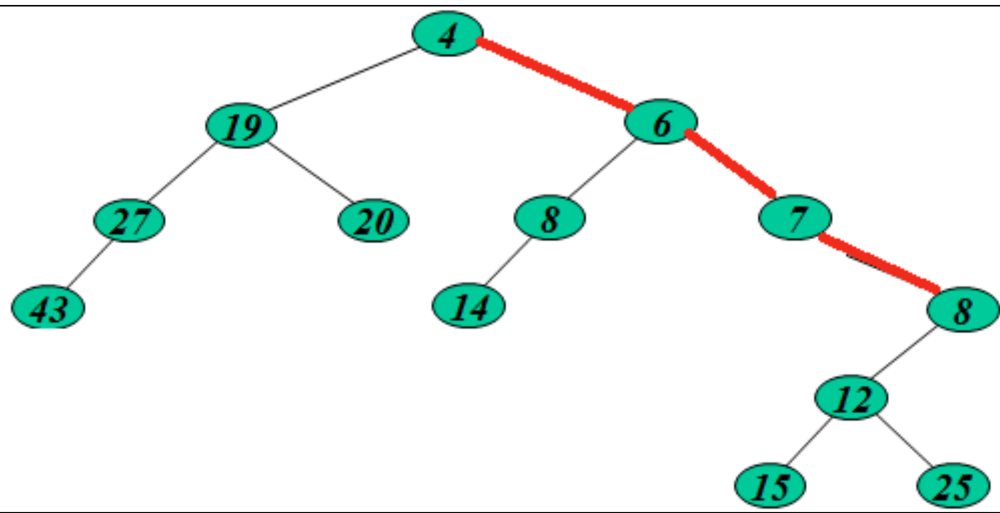
finalized



option 1

option 2

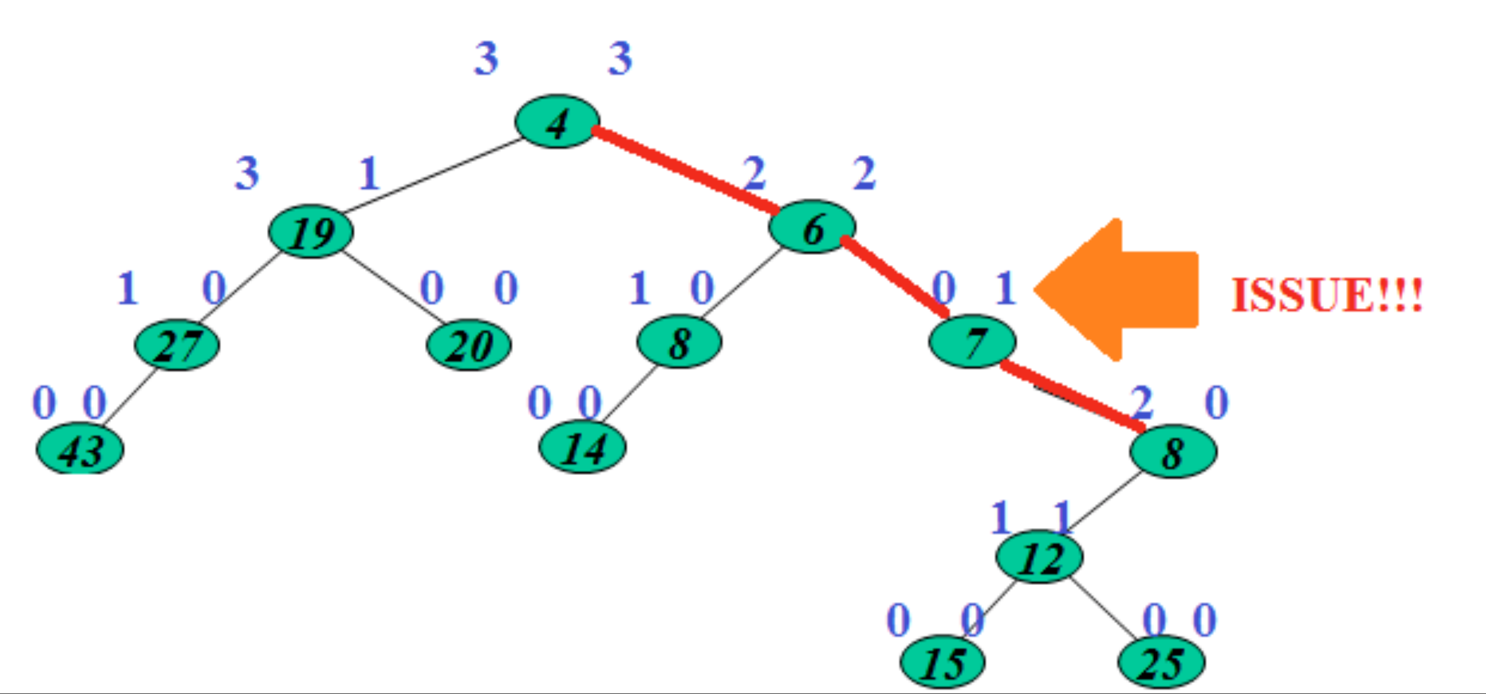
Start at the root of the sub-tree, and finalize the node AND LEFT with the **next** smallest value. Add to RIGHT of finalized tree



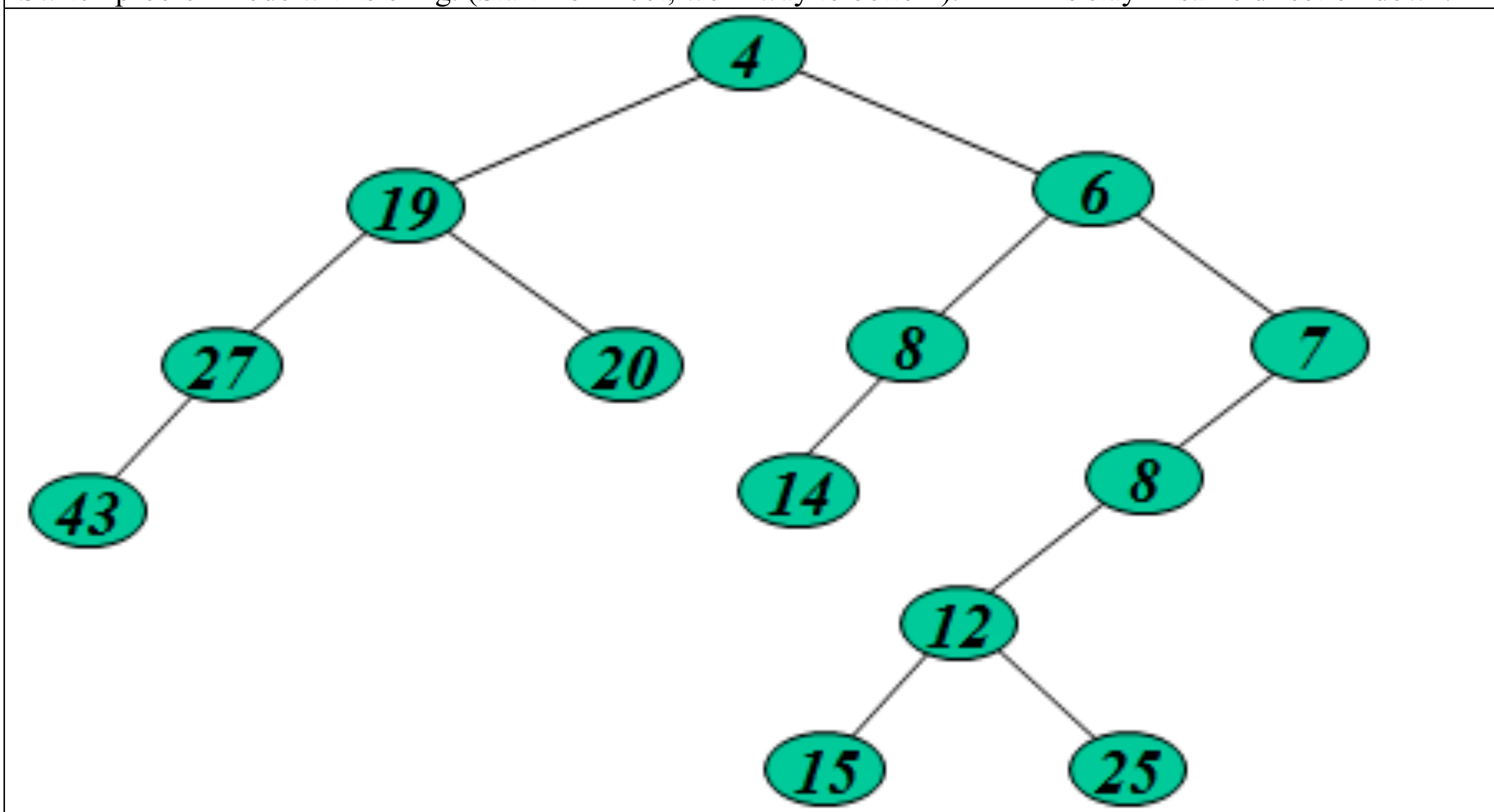
Verify that it is a Min Heap!! (Parent < Children)

Yup

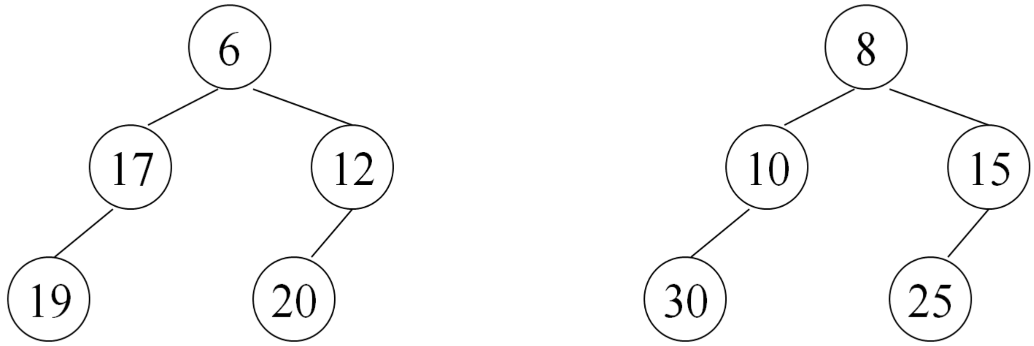
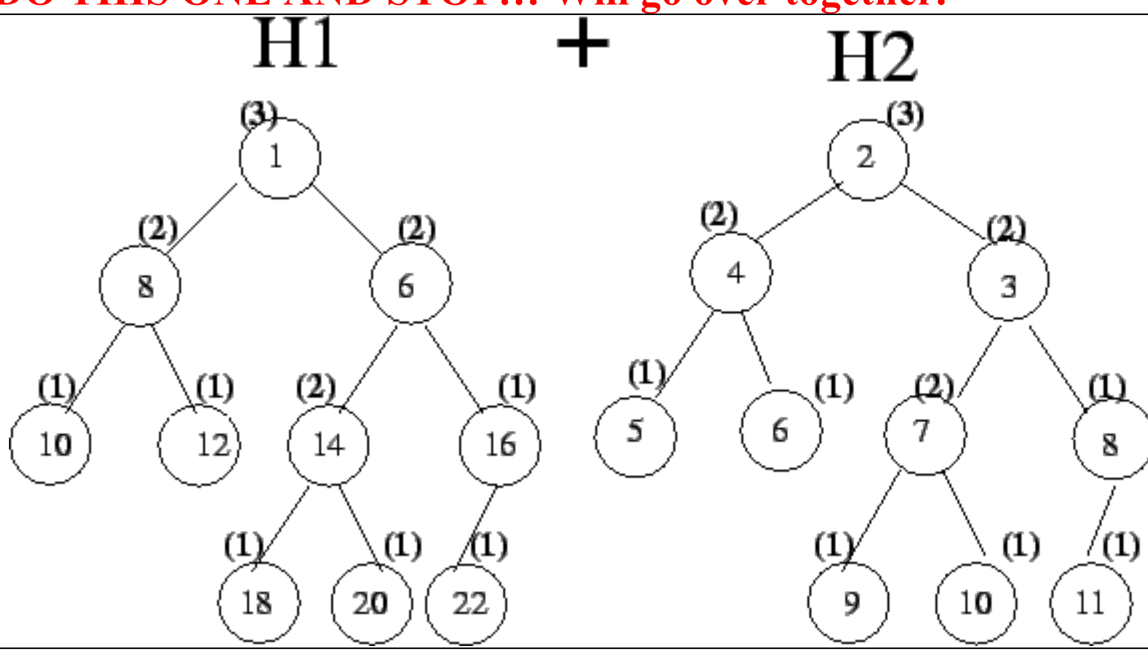
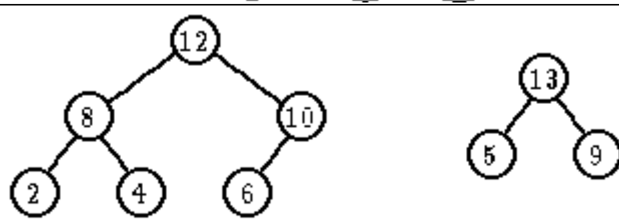
Verify a leftist heap! (left npl <= right npl)



Switch problem node with sibling. (Start from root, work way to bottom). All links stay in same direction down.



Try these:

#1	
DO THIS ONE AND STOP!!! Will go over together.	
#2 a lot of work on this one	
#3 I know!! Heap is a MAX heap!! Try it anyway! (max first!)	

Merging – the function

- notice it is recursive!
- merge()
  - version 1 – copy rhs to root
  - version 2 - is the function to set up the order between left and right heaps
- merge1() is the function to actually do the linking and swapping if **left** npl > **right** npl
  - notice npl is a private variable

Merging Heaps

```
    /**
     * 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 LeftistNode<AnyType> merge( LeftistNode<AnyType> h1,
LeftistNode<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 );
    }

    /**
```

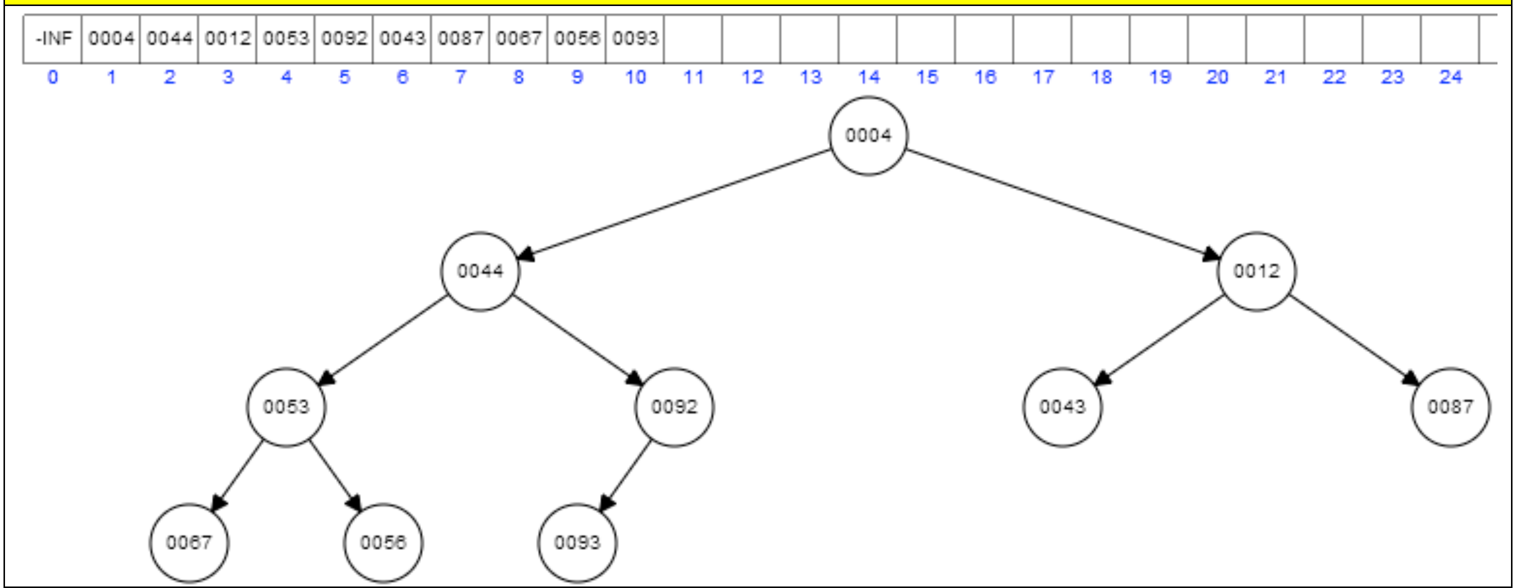
```java
     * Internal method to merge two roots.
     * Assumes trees are not empty, and h1's root contains smallest item.
     */
    private LeftistNode<AnyType> merge1( LeftistNode<AnyType> h1,
LeftistNode<AnyType> h2 )
    {
        if( h1.left == null )   // Single node
            h1.left = h2;       // 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;
    }

    /**
     * Swaps t's two children.
     */
    private static <AnyType> void swapChildren( LeftistNode<AnyType> t )
    {
        LeftistNode<AnyType> tmp = t.left;
        t.left = t.right;
        t.right = tmp;
    }
```

So why did we do this?

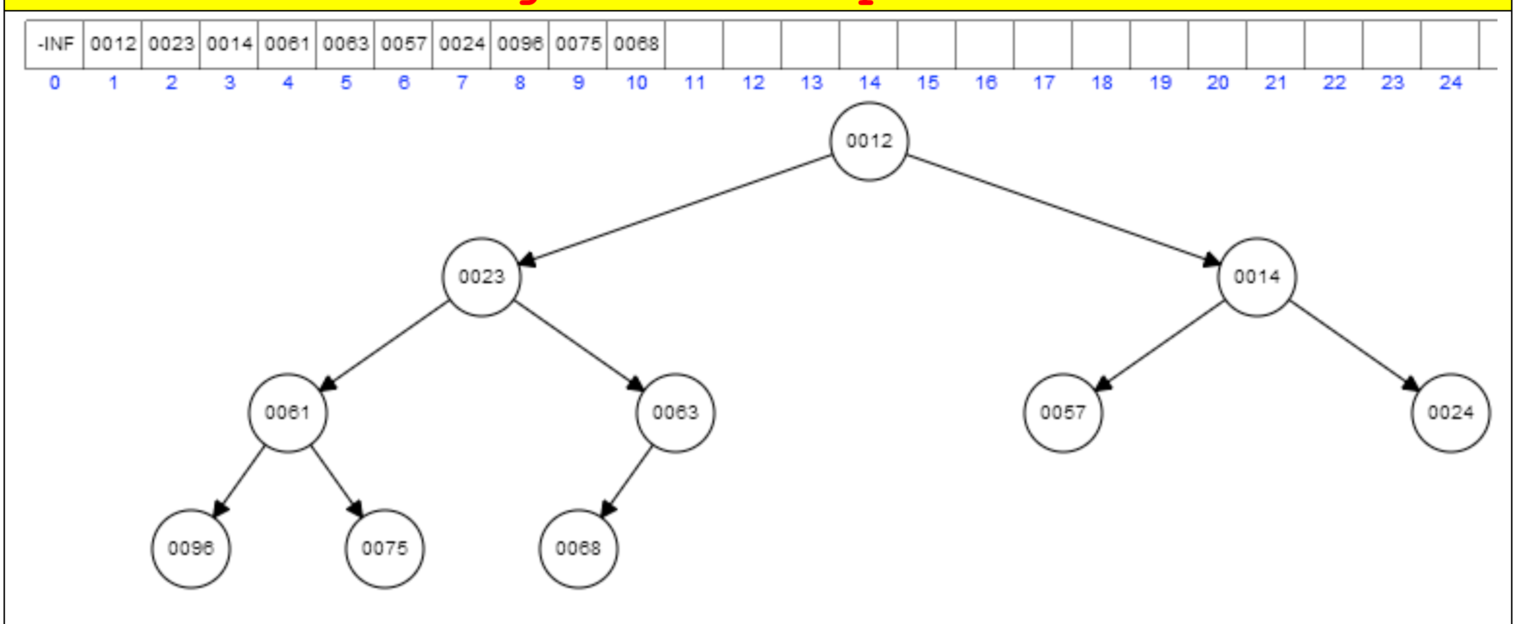
- fast!
  - merge with two trees of size n
    - O(log n), we are not creating a totally new tree!!
    - some was used as the LEFT side!
  - inserting into a left-ist heap
    - O(log n)
    - same as before with a regular heap
  - deleteMin with heap size n
    - O(log n)
    - remove and return root (minimum value)
    - merge left and right subtrees
- real life application
  - priority queue
    - homogenous collection of comparable items
    - **smaller** value means higher priority

Answers:

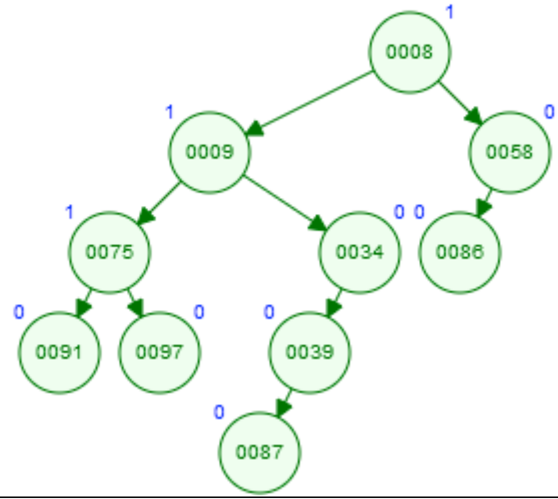
Inserting into a Heap Exercise #1



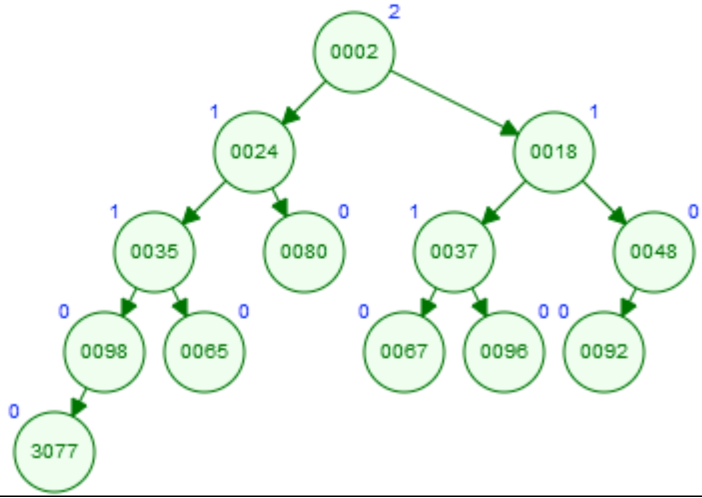
Inserting into a Heap Exercise #2



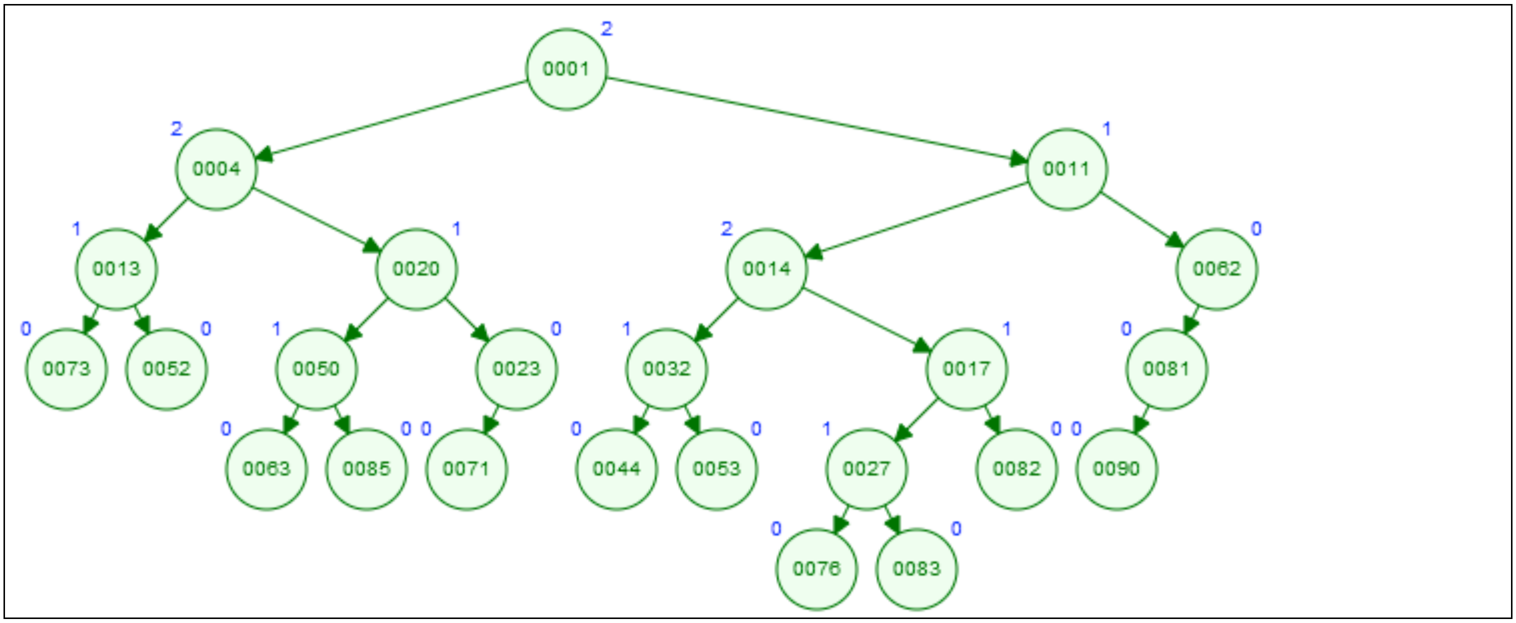
Leftist Heap Creation Exercise #1



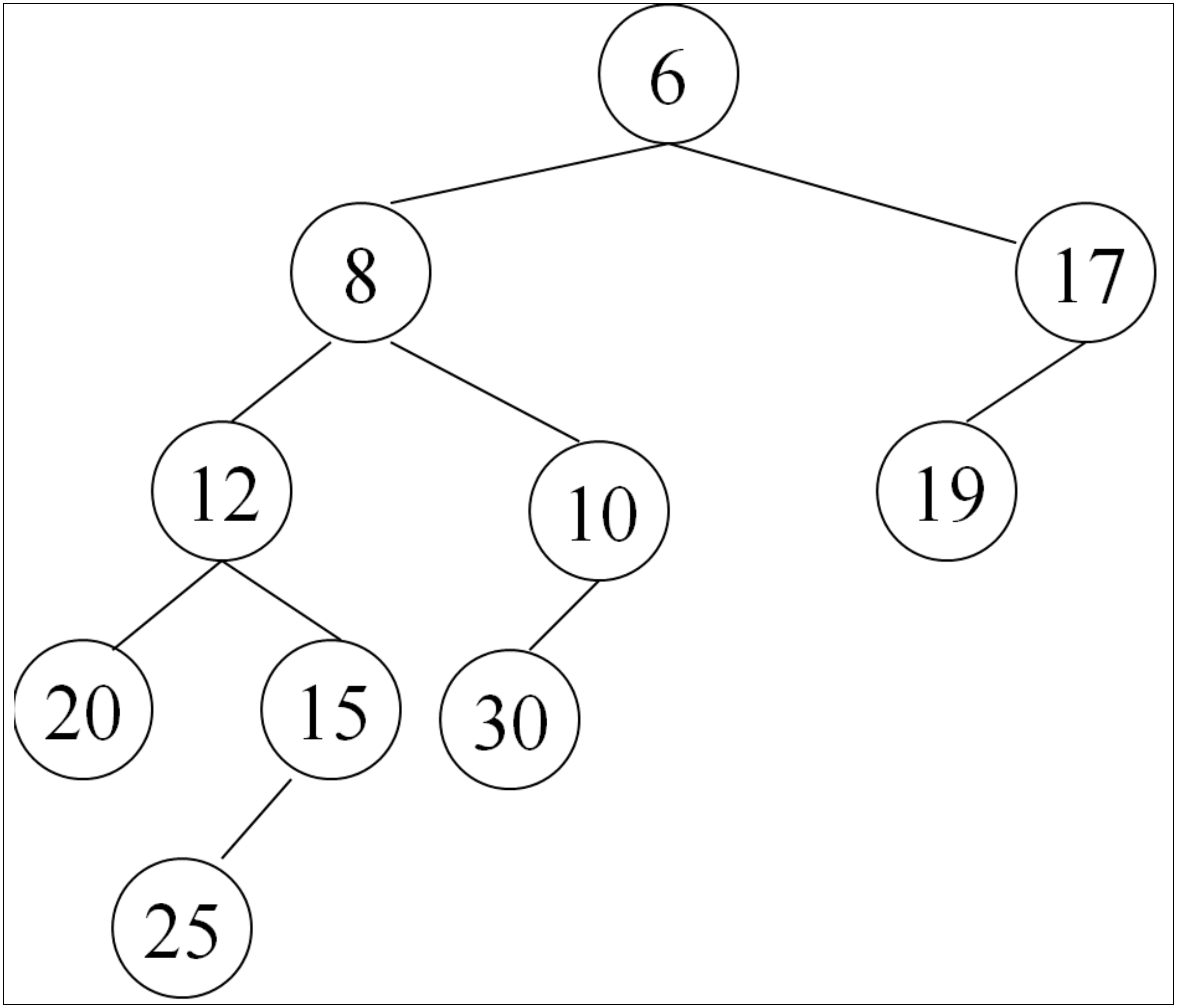
Leftist Heap Creation Exercise #2



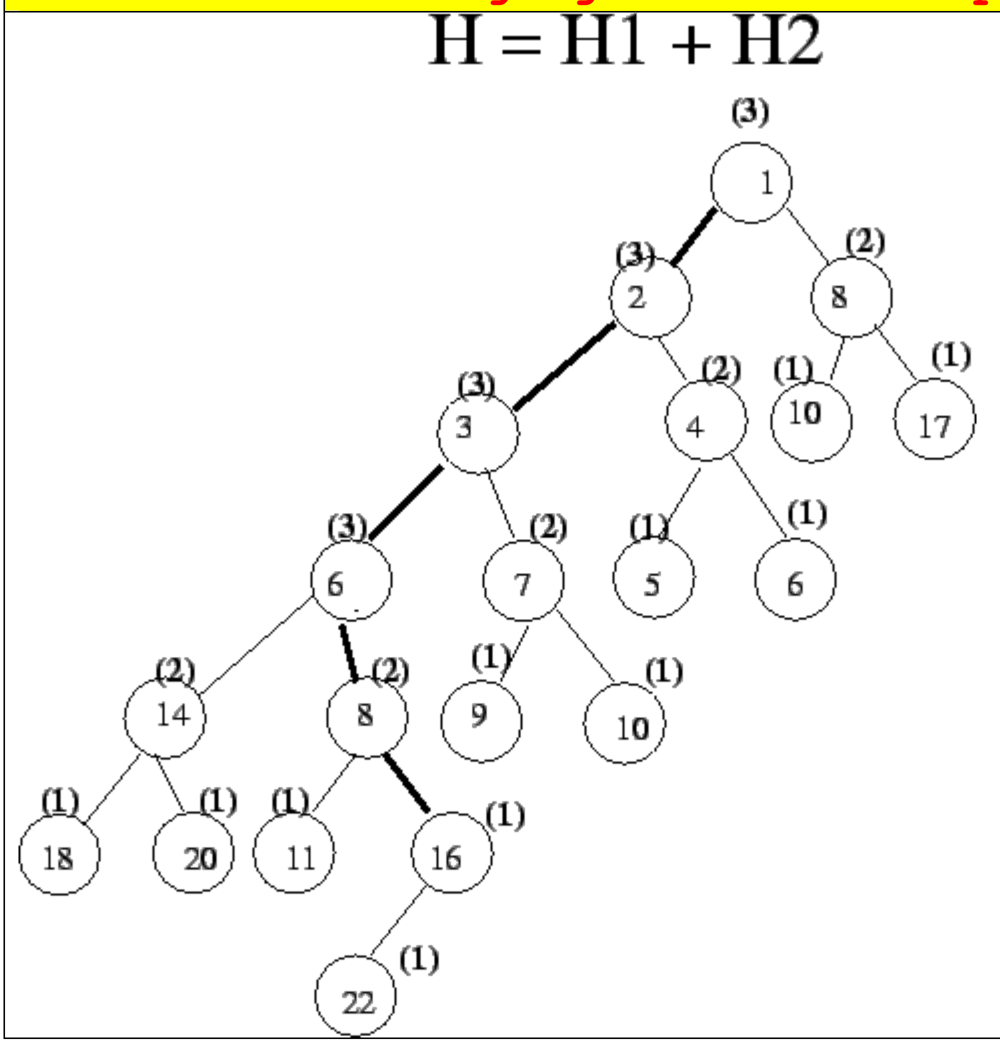
Leftist Heap Creation Exercise #3



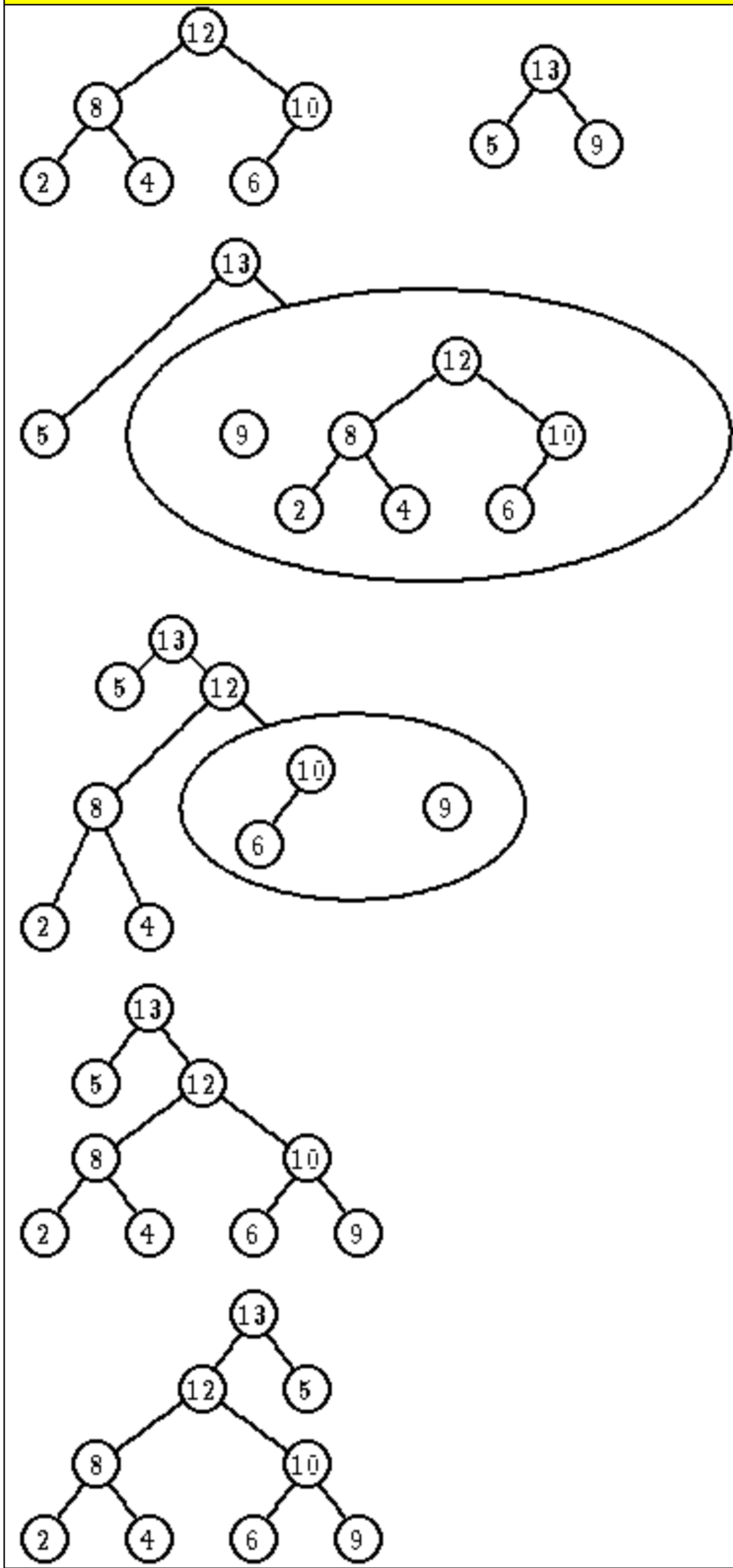
Merging Left-ist Heaps #1


6
8
17
12
10
19
20
15
30
25

Merging Left-ist Heaps #2



Merging Left-ist Heaps #3



Sources:

In General
http://courses.cs.washington.edu/courses/cse332/13wi/lectures/cse332-13wi-lec04-BinMinHeaps-6up.pdf

Maximum HeapSort
http://www.cs.usfca.edu/~galles/visualization/HeapSort.html

Show and Tell Heap building
http://www.cs.usfca.edu/~galles/visualization/Heap.html

Random Heapsort
http://www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heapSort.html
http://nova.umuc.edu/~jarc/idsv/lesson3.html

Building a Leftist Heap
http://www.cs.usfca.edu/~galles/visualization/LeftistHeap.html