CMSC 341
Lecture 15 Leftist Heaps

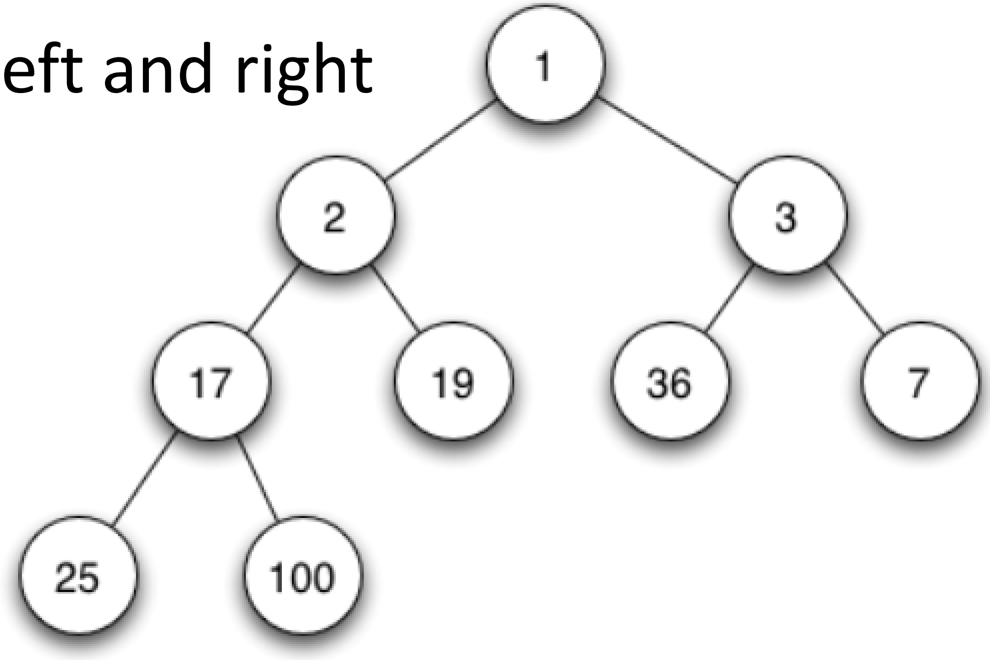
Prof. John Park

Based on slides from previous iterations of this course

Review of Heaps

Min Binary Heap

- A **min binary heap** is a…
  - Complete binary tree
  - Neither child is smaller than the value in the parent
  - No order between left and right
- In other words, smaller items go above larger ones



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 )`**

Introduction to Leftist Heaps

Leftist Heap Performance

- Leftist Heaps support:
  - construct = **O(n)**
  - **`findMin()`** = **O(1)**
  - **`insert()`** = **O(log n)**
  - **`deleteMin()`** = **O(log n)**
  - **`merge()`** = **O(log n)**

Leftist Heap Concepts

- Structurally, a leftist heap is a **min tree** where each node is marked with a **rank** value
  - The *rank* of a node is the depth of the nearest leaf
- Uses a binary tree

  The tree is not balanced, however—just the opposite
- Use a true tree
  - May use already established links to merge with a new node

Null Path Length (npl)

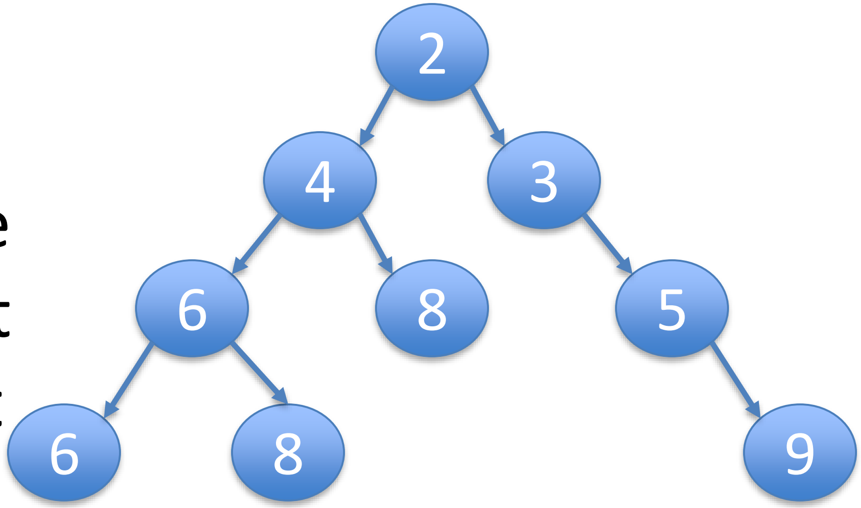
- Length of **shortest** path from current node (X) to a node **without** 2 children
  - analogous to shortest path to a dummy leaf in full tree w/internal value nodes, minus 1
- leaves: npl = 0
- nodes with only 1 child: npl = 0

Leftist Heap Concepts

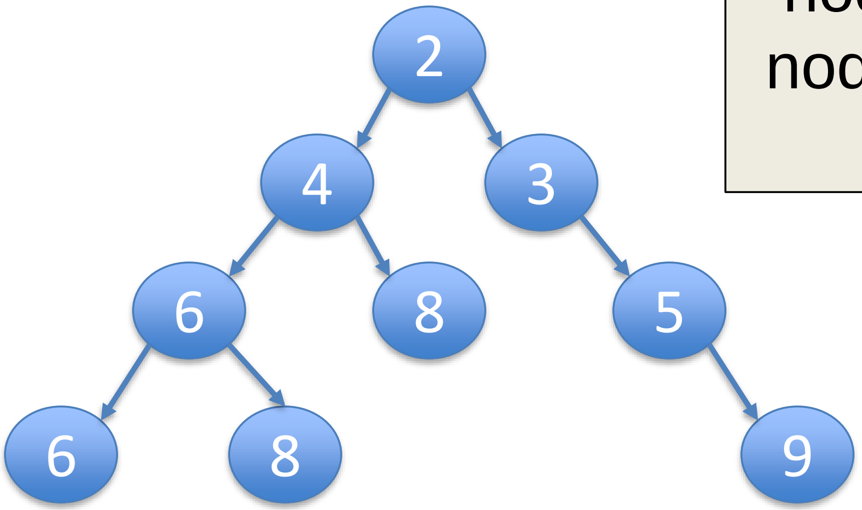
- True heap: values do obey heap order
- Uses *null path length* (*npl*) to maintain the structure (related to *s-value* or *rank*)
  - Additional constraint: the npl of a node's left child is >= npl of the right child
- →At every node, the shortest path to a non-full node is along the rightmost path

Leftist Heap Example

- A leftist heap, then, is a purposefully **unbalanced** binary tree (leaning to the left, hence the name) that keeps its smallest value at the top
- Benefit: has an inexpensive merge operation

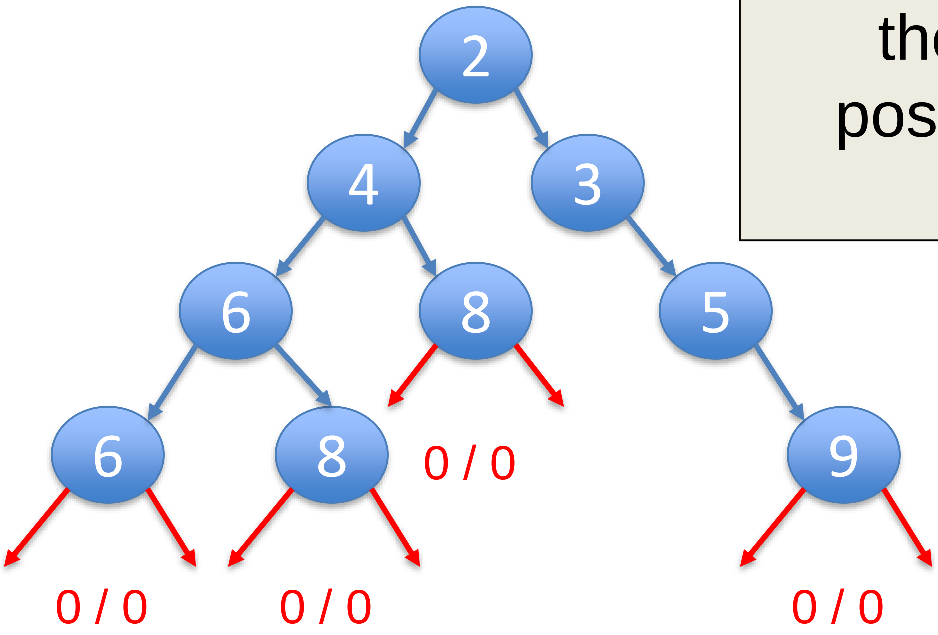


Null Path Length (npl) Calculation



To calculate the npl for each node, we look to see how many nodes we need to traverse to get to an open node

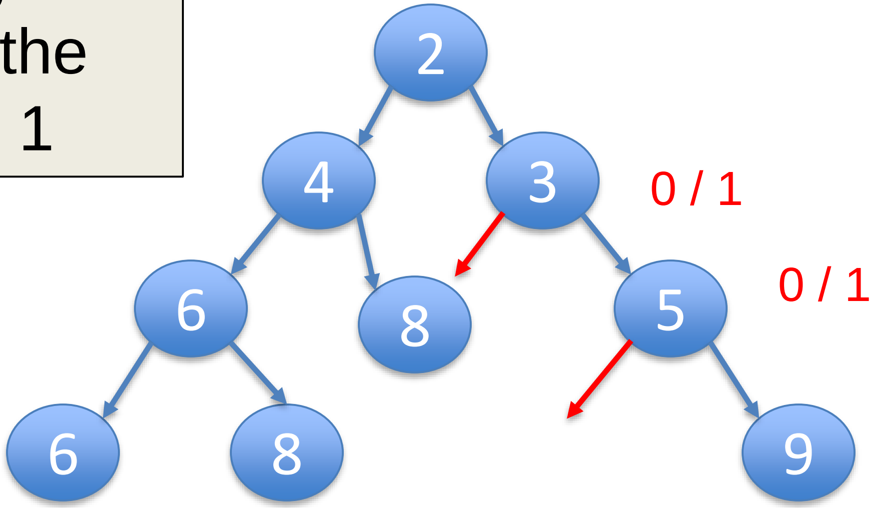
Null Path Length (npl) Calculation



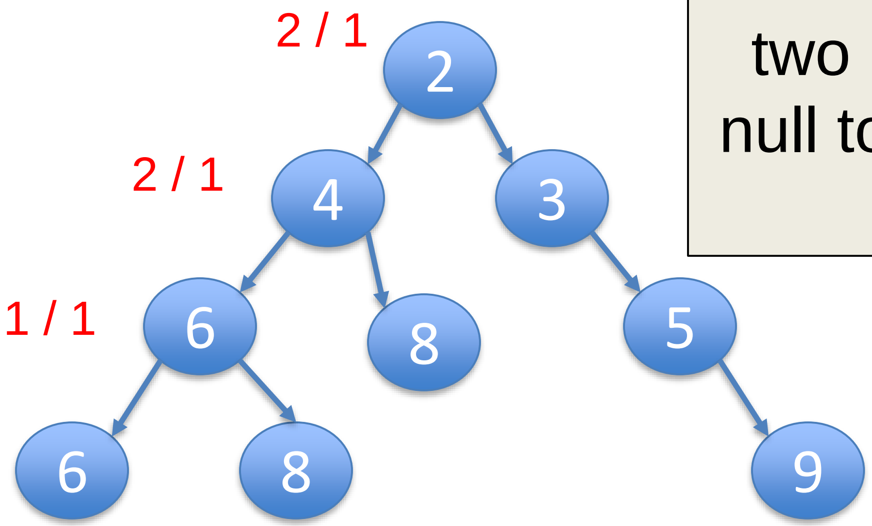
In the leaves case, there is a null position 0 nodes away

Null Path Length (npl) Calculation

In these cases, one side is 0 and the other side is 1



Null Path Length (npl) Calculation



In the root, it will take two levels to get to null to the left; one to the right.

Leftist Node

- The node for a leftist heap will have an additional member variable tracking npl
  - links (left and right)
  - element (data)
  - npl

Leftist Node Code

Looks like a binary tree node except the npl being stored.

```
private:
    struct 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 ) { }
    };

    LeftistNode *root;
```

Building a Leftist Heap

Building a Leftist Heap

- Value of node still matters
  - Still a min Heap, so min value will be root
- Data entered is random

- Uses current npl of a node to determine where the next node will be placed

Merging Nodes/Subtrees

- Insertion and merge are the same: have two nodes in hand, which are single node or root of (sub)tree
- Place lower value as (sub)root, higher value as right child. If the lower-valued node already has right child, then recursively merge the higher-valued node with that right child
  (ultimately equiv. to “place as far right as possible”, but the value being carried down might have swapped)

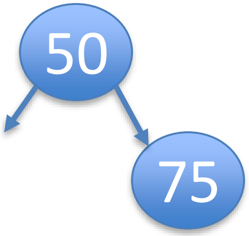
Merging Nodes/Subtrees (cont)

- After merging, the npl of the lower valued node might have changed, so recompute (this is cheap: npl of left and (new) right stay same)
- If the lower-valued node does not have left child, swing right child to the left
- If lower-valued node does have left child, then order children so left has higher npl

Moves in Building Leftist Heap

New leftist heap with one node: 50

Moves in Building Leftist Heap

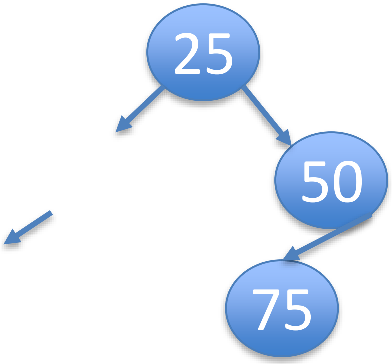


Normal insertion of new node 75 into the tree:

First place as far **right** as possible.

Then swing left to satisfy npls.

Moves in Building Leftist Heap

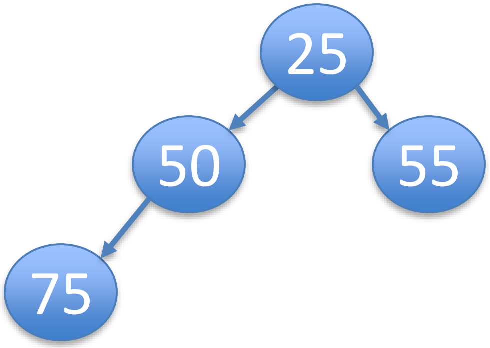


Normal insertion of new node 25 into the tree:

As this is a min Tree, 25 is the new root.

Then swing left to satisfy npls.

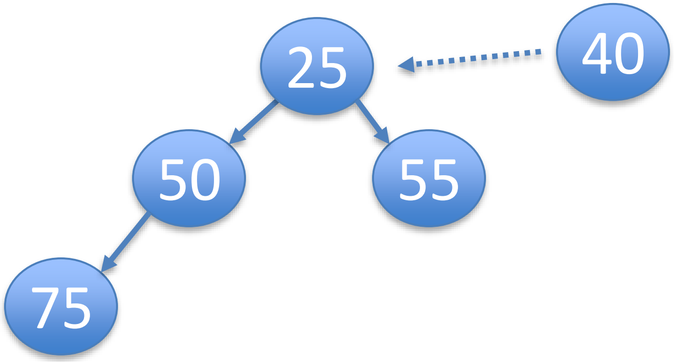
Moves in Building Leftist Heap



Normal insertion of new node 55 into the tree:

No swing required.

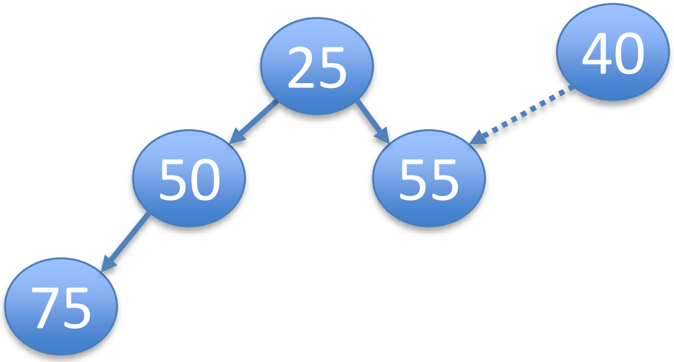
Moves in Building Leftist Heap



Normal insertion of new node 40 into the tree:

40 is larger than 25, but 25 has right child, so merge 40 w/55

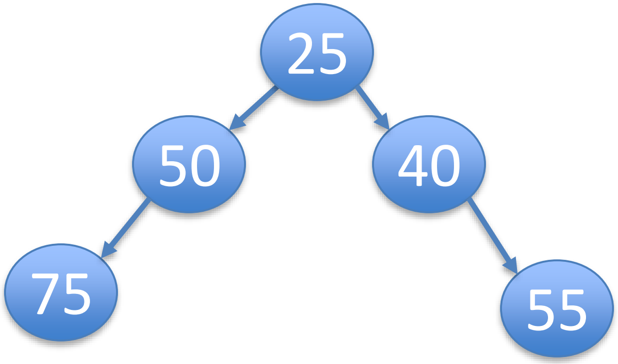
Moves in Building Leftist Heap



Normal insertion of new node 40 into the tree:

40 is smaller than 55, so it becomes new subroot (child of 25), w/55 as its right child

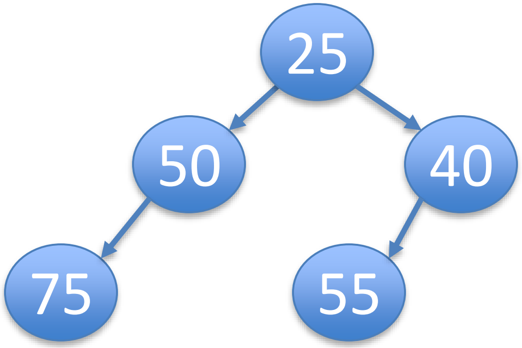
Moves in Building Leftist Heap



Normal insertion of new node 40 into the tree:

40 is smaller than 55, so it becomes new subroot (child of 25), w/55 as its right child

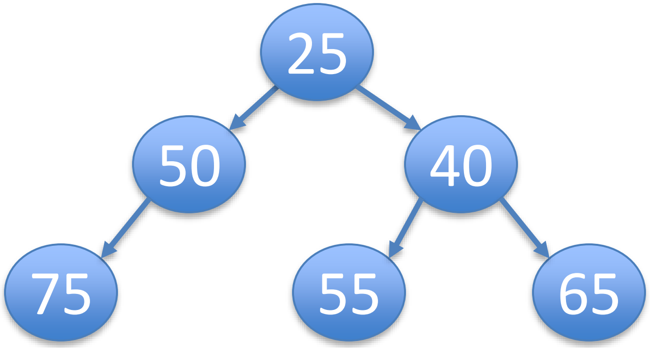
Moves in Building Leftist Heap



Normal insertion of new node 40 into the tree:

Then, swing 55 over to left

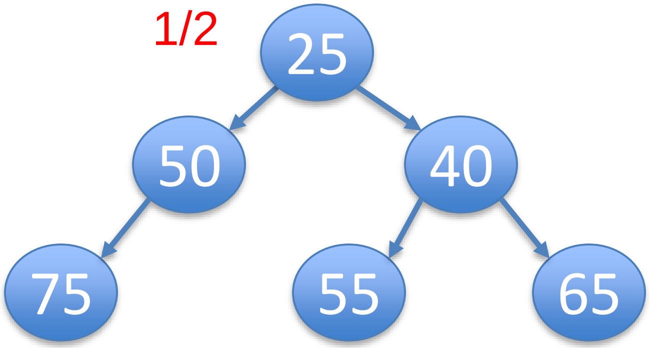
Moves in Building Leftist Heap



Normal insertion of new node 65 into the tree:

Insert as usual: ends up as sibling of 55 (Note: later, we might consider swap w/55...)

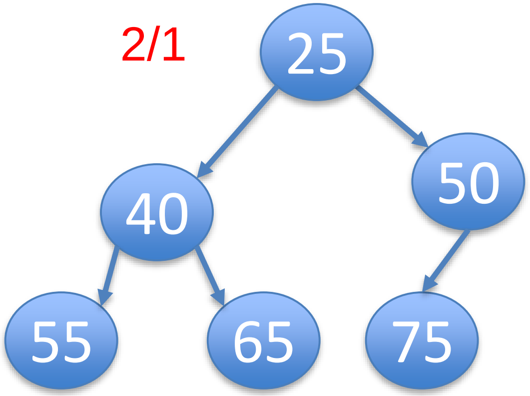
Moves in Building Leftist Heap



Normal insertion of new node 65 into the tree:

Note: must recompute npl of nodes above, resulting in noncompliant root!

Moves in Building Leftist Heap



We need change this from 1/2 to 2/1 so that it remains leftist.

To do this, we switch the left and the right subtrees.

After we do the swap, the npl of the root is compliant.

Leftist Heap Algorithm

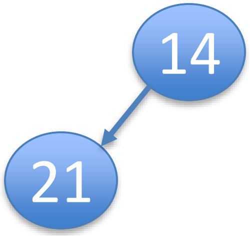
- Add new node to right-side of tree, in order
- If new node is to be inserted as a parent (parent < children)
  - make new node parent
  - link children to it
  - link grandparent down to new node **(now new parent)**
- If leaf, attach to right of parent
- If no left sibling, push to left (hence left-ist)
- 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

Building a Leftist Heap Example

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

Building a Leftist Heap Example

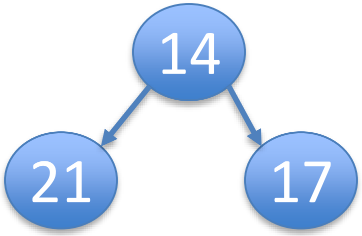
~~21~~, ~~14~~, 17, 10, 3, 23, 26, 8



Insert 14 as the new root

Building a Leftist Heap Example

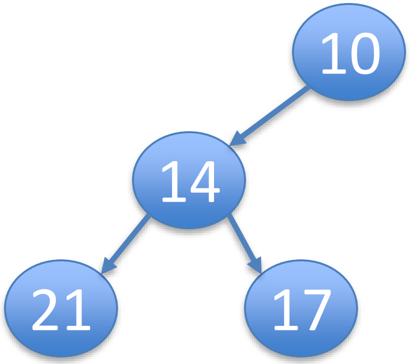
~~21~~, ~~14~~, ~~17~~, 10, 3, 23, 26, 8



Insert 17 as the right child of 14

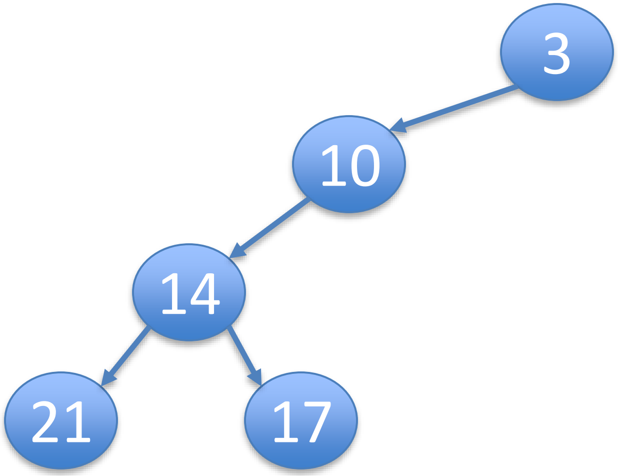
Building a Leftist Heap Example

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



Insert 10 as the new root

Building a Leftist Heap Example

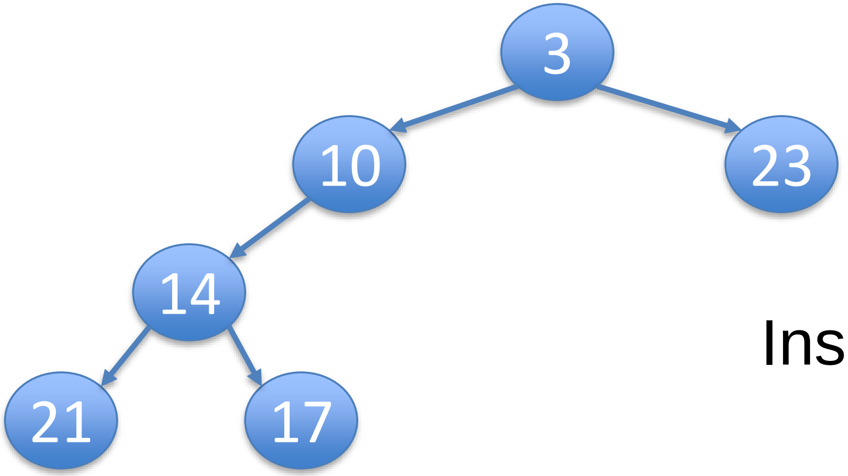


~~21~~, ~~14~~, ~~17~~, ~~10~~, ~~3~~, 23, 26, 8

Insert 3 as the new root

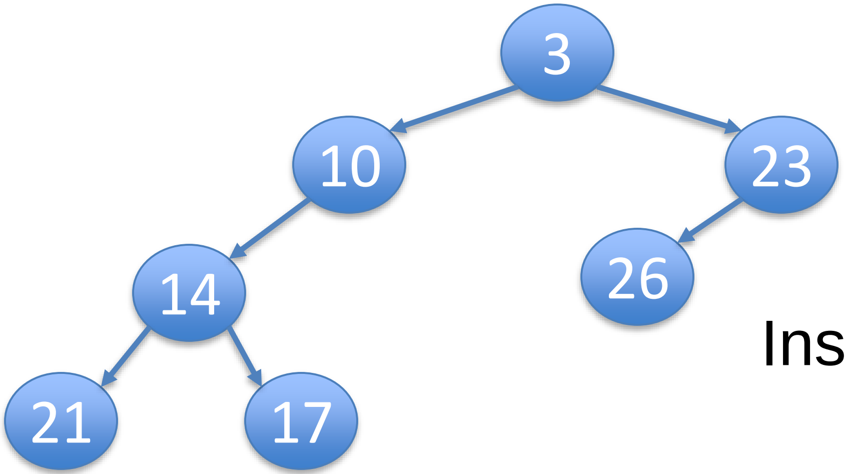
Building a Leftist Heap Example

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



Insert 23 as the right child of 3

Building a Leftist Heap Example

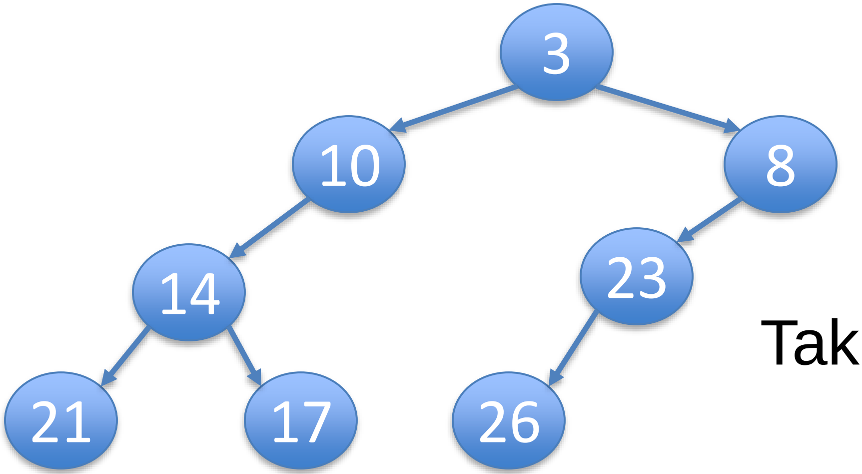


~~21~~, ~~14~~, ~~17~~, ~~10~~, ~~3~~, ~~23~~, ~~26~~, 8

Insert 26 as the right child of 23

Swing 26 to the left

Building a Leftist Heap Example



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

Take the right subtree of root 3: nodes 23 & 26

Insert 8 as the new root (parent of 23)

Reattach to original root

Leftist Heap Animation

- https://www.cs.usfca.edu/~galles/visualization/LeftistHeap.html

Merging Leftist Heaps

Merging Leftist Heaps

- Leftist heaps actually optimized for merging entire trees
- Adding a single node is treated as special case: merging a heap of one node with an existing heap’s root

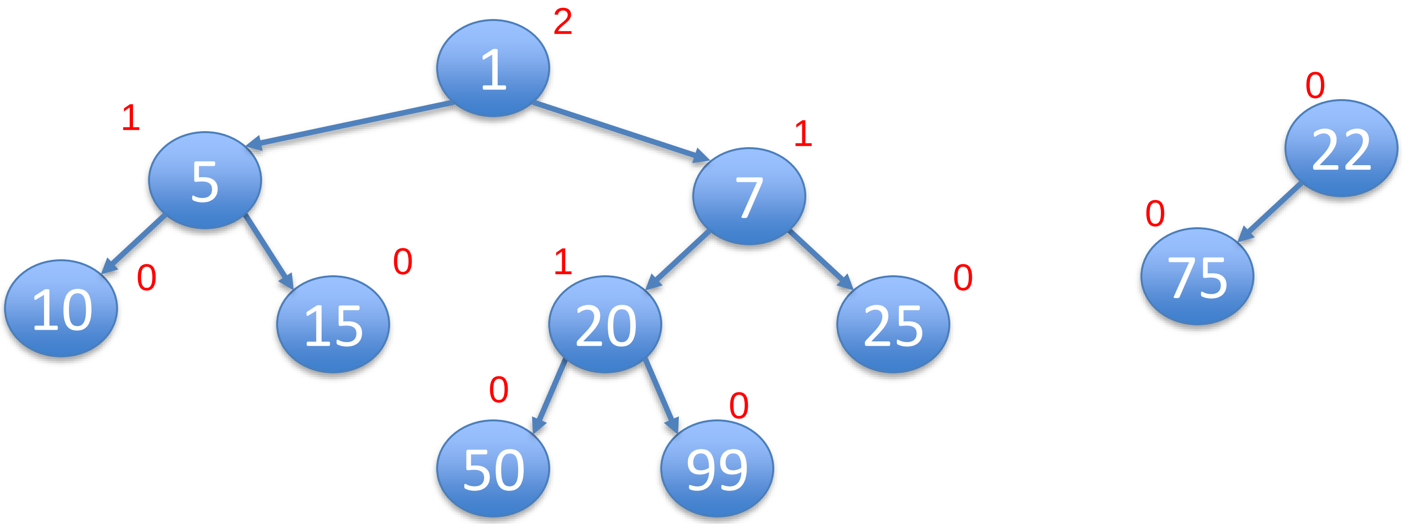
Merging Leftist Heaps

- The two constituent heaps we are about to merge must already be leftist heaps
- Result will be a new leftist heap

Merging Leftist Heaps

- The Merge procedure takes two leftist trees, A and B, and returns a leftist tree that contains the union of the elements of A and B. In a program, a leftist tree is represented by a pointer to its root.

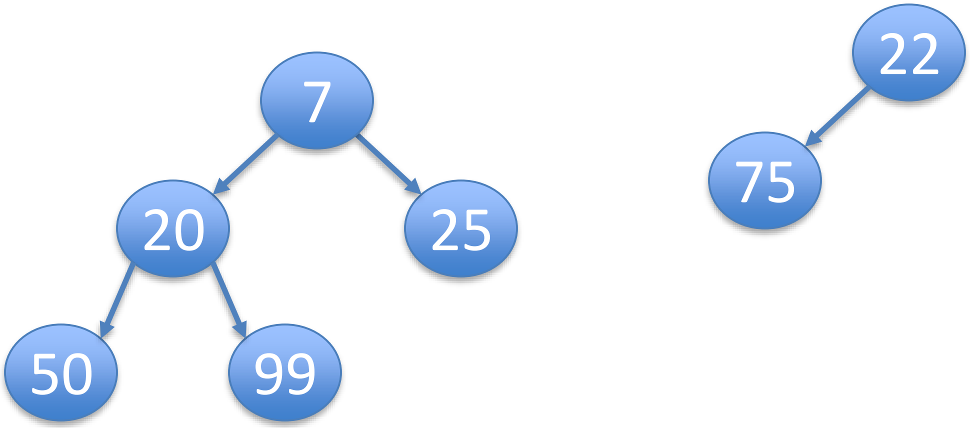
Merging Leftist Heaps Example



We start by attempting to merge 1 and 22

1 is smaller, so we attempt to merge…

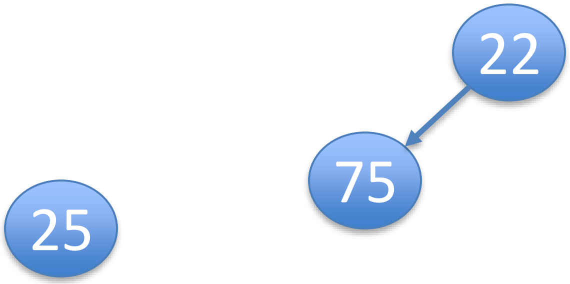
Merging Leftist Heaps Example



the right subtree 7 with 22

7 is smaller, so we attempt to merge…

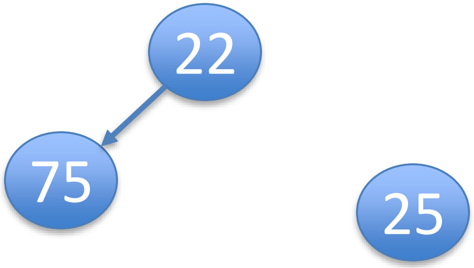
Merging Leftist Heaps Example



the right subtree 25 with 22

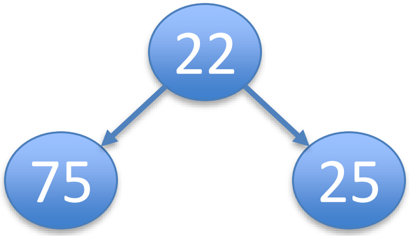
This time, 22 is smaller, so we…

Merging Leftist Heaps Example



merge 25 with the right subtree of 22: empty, so…

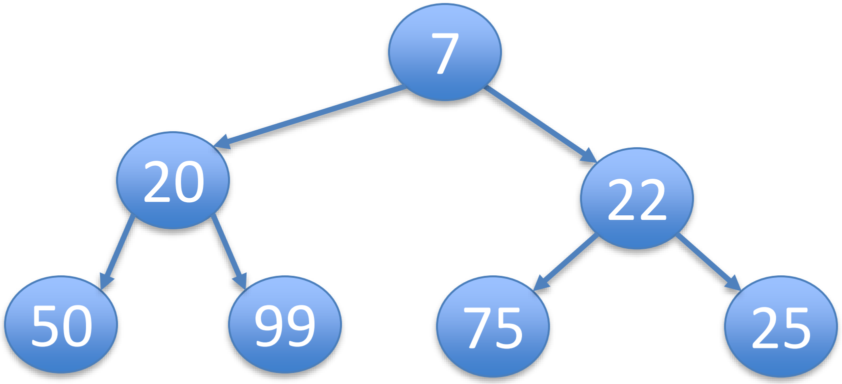
Merging Leftist Heaps Example



Add as child of 22

npl‘s same, so no swap

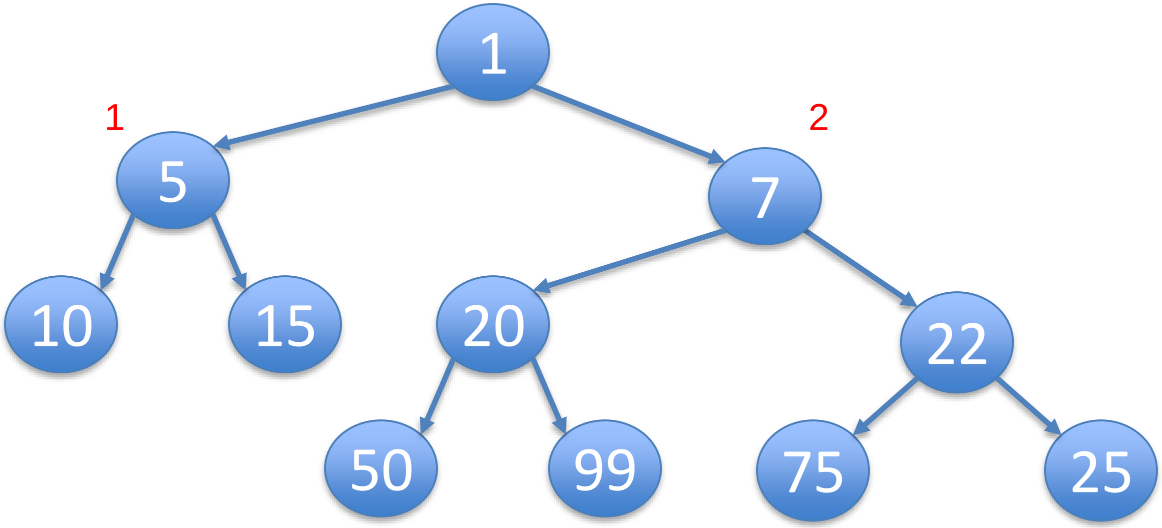
Merging Leftist Heaps Example



Next level of the tree:
add 22 as child of 7

npl‘s same, so no swap

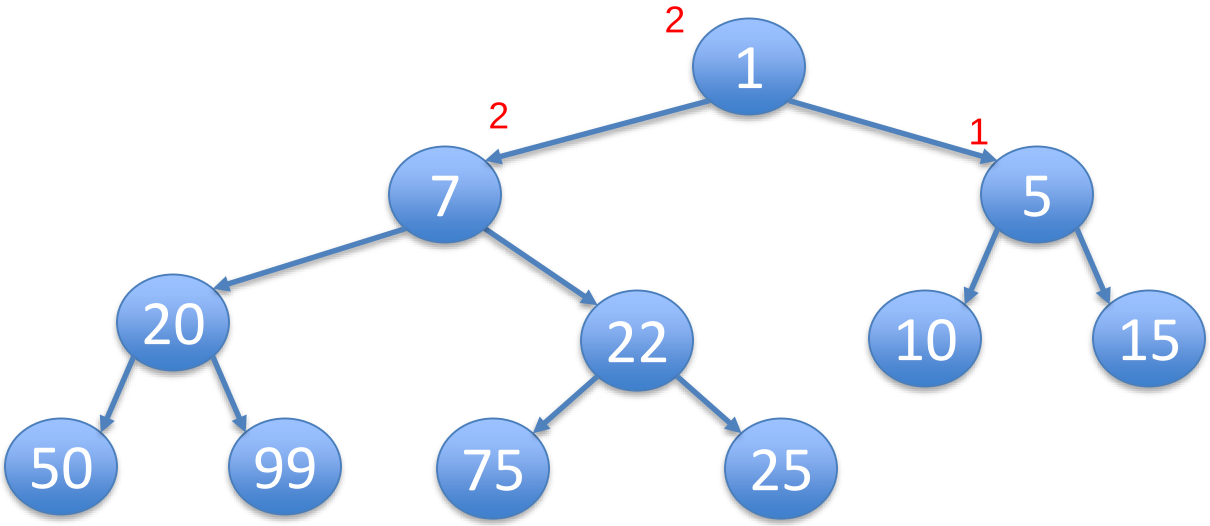
Merging Leftist Heaps Example



Next level of the tree:
add 7 as child of 1

right npl > left npl, so swap

Merging Leftist Heaps Example



Now the highest npl
is on the left.

Merging Leftist Heaps Code

```
/**
    * Merge rhs into the priority queue.
    * rhs becomes empty. rhs must be different from this.
    */
   void merge( LeftistHeap & rhs )
   {
       if( this == &rhs )    // Avoid aliasing problems
           return;

       root = merge( root, rhs.root );
       rhs.root = NULL;
   }
```

Merging Leftist Heaps Code

```
/**
 * Internal method to merge two roots.
 * Deals with deviant cases and calls recursive merge1.
 */
LeftistNode * merge( LeftistNode *h1, LeftistNode *h2 )
{
    if( h1 == NULL )
        return h2;
    if( h2 == NULL )
        return h1;
    if( h1->element < h2->element )
        return merge1( h1, h2 );
    else
        return merge1( h2, h1 );
}
```

Merging Leftist Heaps Code

```
/**
 * Internal method to merge two roots.
 * Assumes trees are not empty, & h1's root contains smallest item.
 */
LeftistNode * merge1( LeftistNode *h1, LeftistNode *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;
}
```

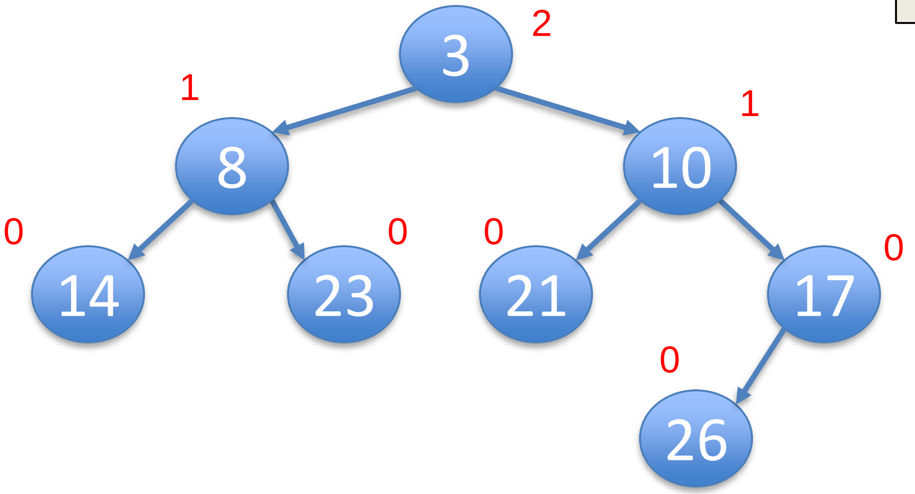
Deleting from Leftist Heap

Deleting from Leftist Heap

- Simple to just remove a node (since at top)
  - this will make two trees
- Merge the two trees like we just did

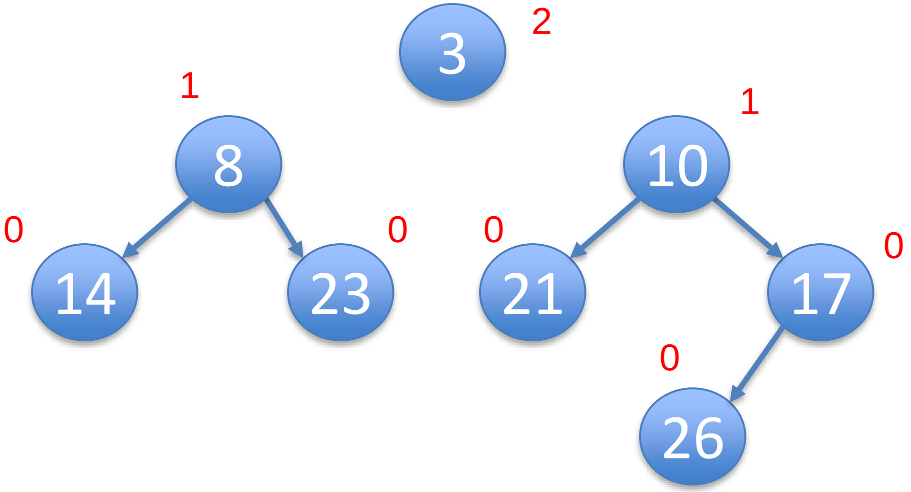
Deleting from Leftist Heap

We remove the root.



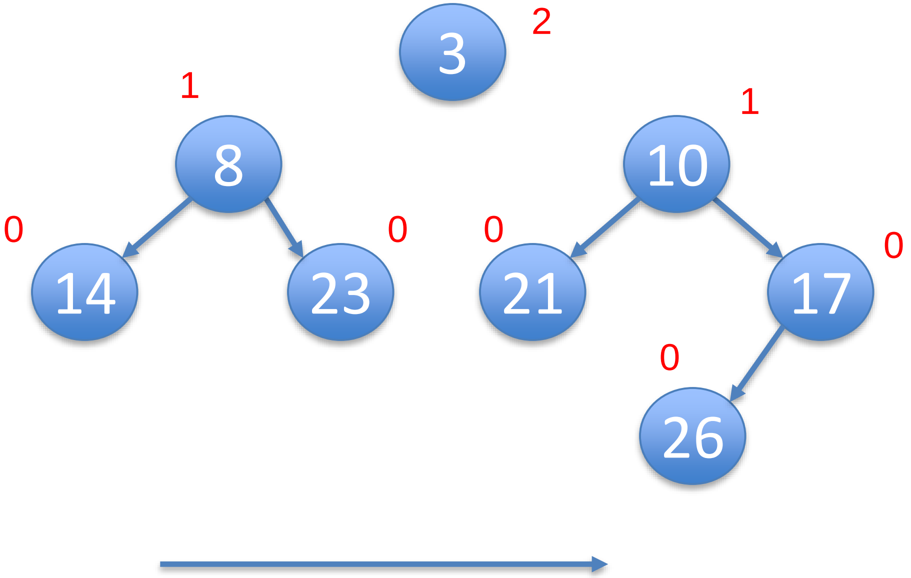
(To build this tree, insert in order: 8, 14, 23, 3, 10, 21, 17, 26)

Deleting from Leftist Heap



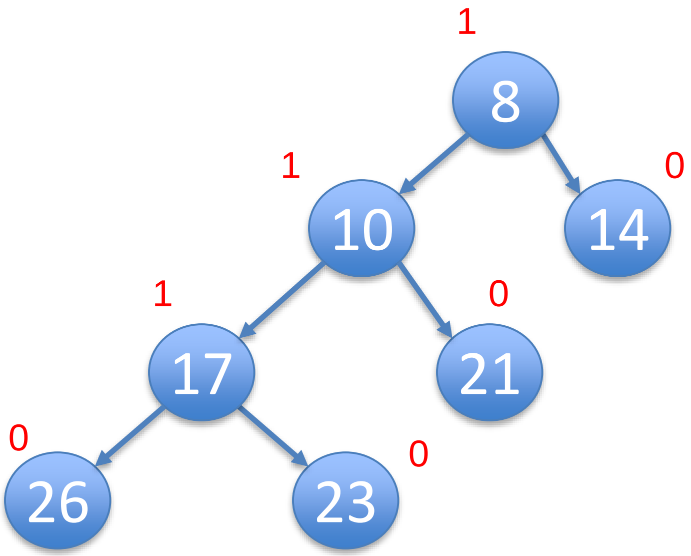
Then we do a merge and because min is in left subtree, we recursively merge right into left

Deleting from Leftist Heap



Then we do a merge and because min is in left subtree, we recursively merge right into left

Deleting from Leftist Heap



After Merge

Leftist Heaps

- 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