Priority Queues (Heaps)

CS 202 – Fundamental Structures of Computer Science II

Bilkent University

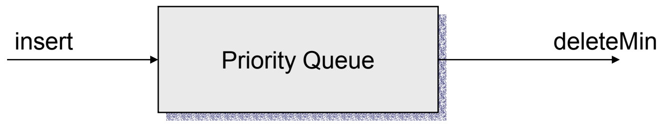
Computer Engineering Department

Need for priority based retrievel

- Some applications require different treatments for items
  - Printer queue
    - Items (files to be printed) can be prioritized depending on their size.
  - OS scheduler
    - It may be better to schedule smaller sized programs before larger sized programs.
- Priority queue structure can be used to model these kind of applications
  - So that some operations (like finding the highest priority item) can be implemented very efficiently.

Model

- A priority queue is a data structure that allows at the least the following two operations
  - Insert
    - Inserts an element to a priority queue and maintains the priority queue properties after insertion.
  - deleteMin
    - Finds, returns, and removes the minimum element in the priority queue.



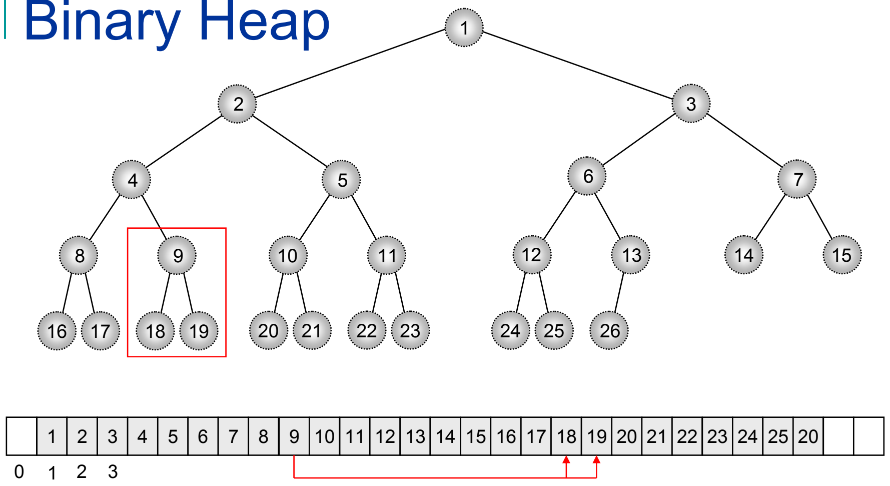
Simple Implementations of priority queues

- Use a simple linked list
  - Insert to the head of the list (O(1))
  - Search for an item by traversing the list from start to end or until item found (O(N)).
- Use a binary search tree
  - Insert into the tree. (O(logN))
  - Remove from the tree (O(logN)).

Binary Heap (or Heap)

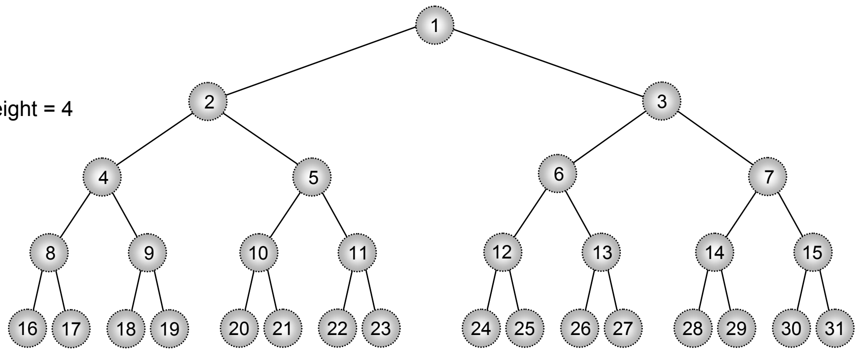
- It is a *complete* binary tree.
  - A tree that is completely filled except the bottom level, which is strictly filled from left to right.
- A complete binary tree of height h has between 2h and $2^{h+1} – 1$ nodes.
- The height of a complete binary tree is: $\lfloor \log N \rfloor$

Binary Heap



An element at position ***i*** will have children at positions **2*i*** and ***2i+1***

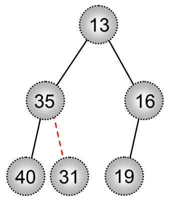
An element at position ***i*** will have parent at position $\lfloor i/2 \rfloor$

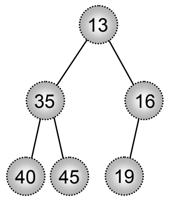


When height = 4:
- Minimum number of nodes = 16 = 2^4
- Maximum number of nodes = 31 = $2^5 - 1$

Heap Order property

- In a heap, for every node X, the key in the parent of X is smaller or equal to the key in X.
- Root has the smallest key.


A complete tree, but not a heap!


A heap

Heap Class

```
class BinaryHeap
      {
        public:
          explicit BinaryHeap( int capacity = 100 );
          bool isEmpty( ) const;
          bool isFull( ) const;
          const int & findMin( ) const;
          void insert( const int & x );
          void deleteMin( );
          void deleteMin( int & minItem );
          void makeEmpty( );

        private:
          int               currentSize;  // Number of elements in heap
          vector <int>    array;        // The heap array
          void buildHeap( );
          void percolateDown( int hole );
      };
```

Heap Insert

```
void BinaryHeap::insert( const int & 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;
      }
```

Heap Delete Minimum

```
void BinaryHeap::deleteMin( )
      {
         if( isEmpty( ) )
            throw Underflow( );

         array[ 1 ] = array[ currentSize-- ];
         percolateDown( 1 );
      }
```

Heap Delete Minimum

```
void BinaryHeap::percolateDown( int hole )
      {
         int child;
         int tmp = array[ hole ];

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

Other Heap Operations

- Heap provides fast access to minimum element
  - O(longN) worst case
- But heap does not store any other ordering information.
- Searching for an arbitrary item is not easy
  - Requires O(N)) time.
- An other data structure needs to be kept if we want to do other operations also.

Other Heap Operations

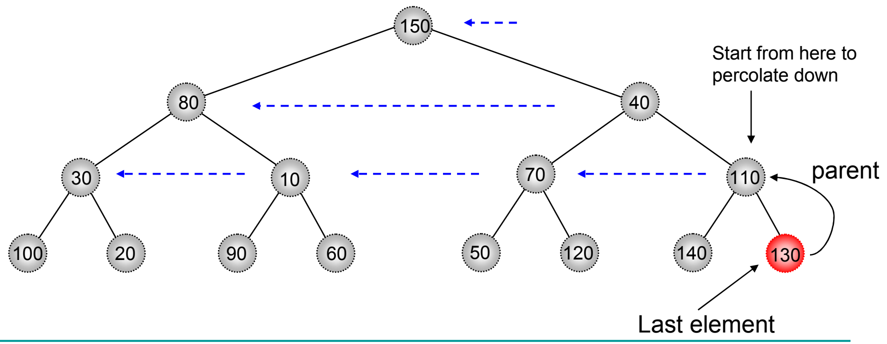
- If we assume that the position of other elements (other than minimum) is also known by some other methods, several operations become cheap
  - decreaseKey (p. Δ)
    - Lowers the value of the item at position *p* by Δ amount.
      - Requires percolate up
  - increseKey (p, Δ)
    - Increases the value of the item at position *p* by Δ amount
      - Requires perculate down
  - remove (p)
    - Removes the item at position *p*
      - *decreaseKey(p, ∞) and deleteMin();*
  - BuildHeap
    - Take N items and place them into an empty heap.

buildHeap()

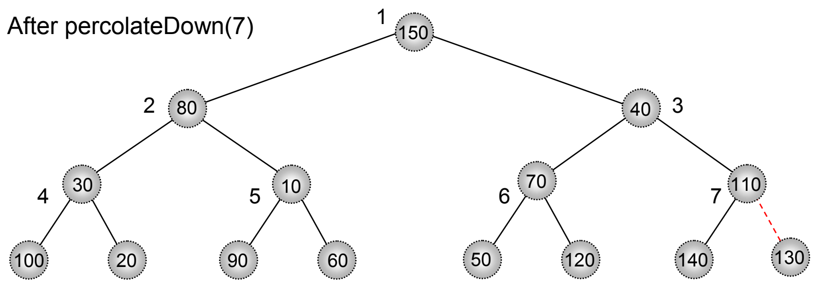
- Takes N items and builds a heap.
  - Simple method is: Insert N items successively.
- A more efficient method exists.
  - O(N) running time.
- Sketch of Algorithm
  - 1. Position = lowerbound(heapSiz/2); // initial position
  - 2. perculateDown item at that position.
  - 3. Decrement position by one.
  - 4. Repeat steps 2,3,4 until position is 0.

buildHeap()

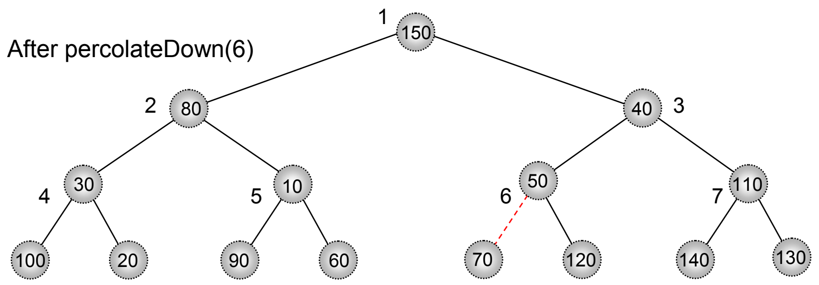
```
template <class Comparable>
      void BinaryHeap<Comparable>::buildHeap( )
      {
         for( int i = currentSize / 2; i > 0; i-- )
            percolateDown( i );
      }
```



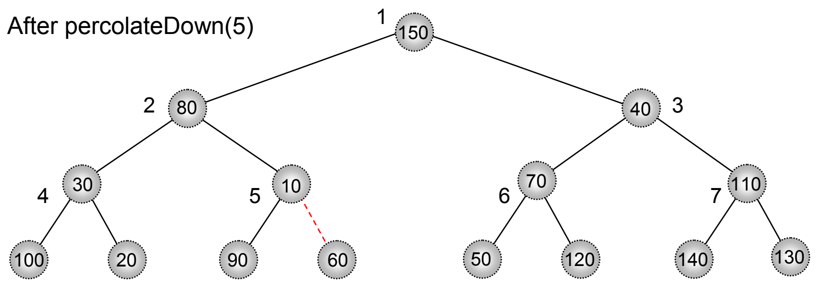
After percolateDown(7)



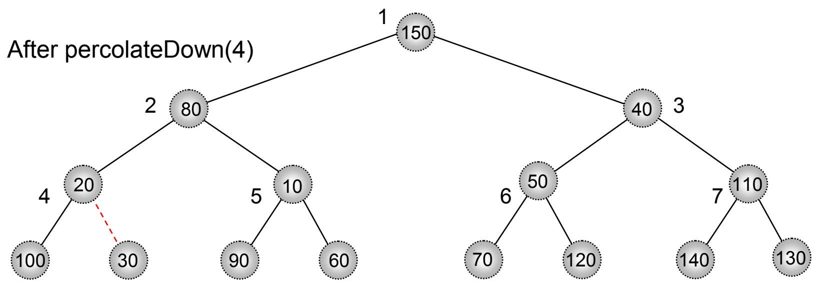
After percolateDown(6)



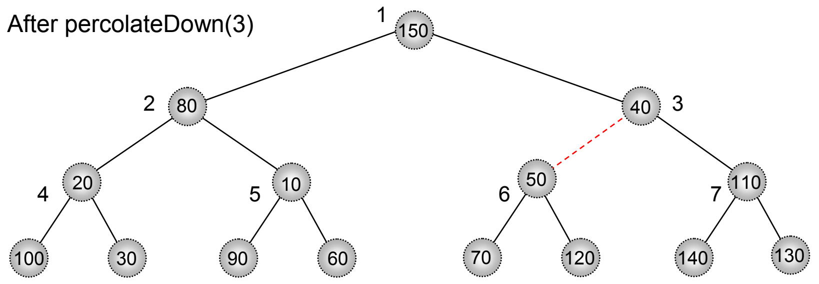
After percolateDown(5)



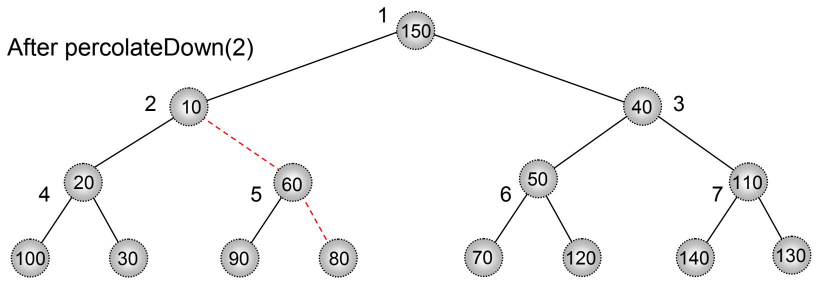
After percolateDown(4)



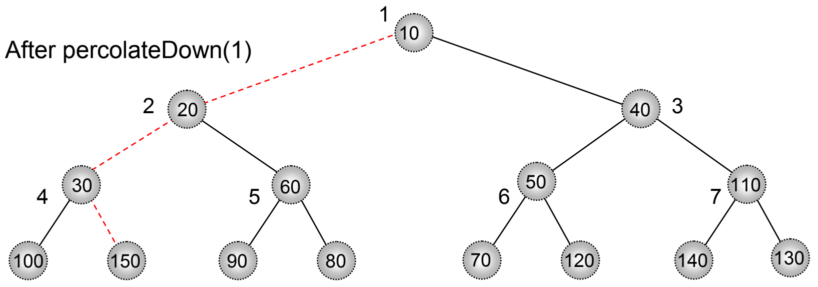
After percolateDown(3)



After percolateDown(2)



After percolateDown(1)



Analysis of BuildHeap

- The cost of BuildHeap is bounded by the number of dashed red lines, which is bounded by the sum of heights of all nodes of the heap.
- We will show that this sum is O(N), where N is the number of nodes in the heap.

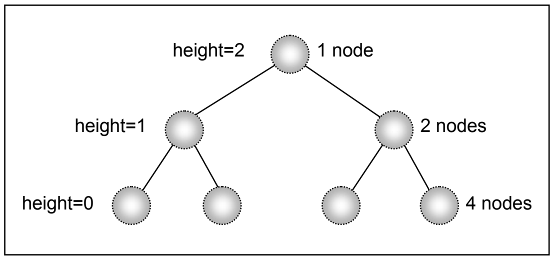
Theorem:

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

Proof:

A perfect binary contains:

1 node at height h
2 nodes at height h - 1
4 nodes at height h - 2
2^3 nodes at height h - 3.
......
2^h nodes at height 0.



Sum of heights of all nodes = $S = \sum_{i=0}^{h} 2^i (h-i)$

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

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

$$\begin{aligned} 2S - S &= -h + 2 + 4 + 8 + 16 + \ldots\ldots + 2^{h-1} + 2^{h} \\ &= -h - 1 + 1 + 2 + 4 + 8 + 16 + \ldots\ldots + 2^{h-1} + 2^{h} = \\ &= -h - 1 + 2^{h+1} - 1 \\ &= 2^{h+1} - 1 - (h+1) \end{aligned}$$

$S = 2^{h+1} - 1 - (h+1)$, where $N = 2^h$

$\Rightarrow \; S = O(N)$

- A complete tree is not a perfect binary tree, but number of nodes in a complete tree of height h is:
  - $2^h <= N < 2^{h+1}$
- The sum we came up is an upperbound on the sum of height of all nodes in a complete tree.

Application of Priority Queues

- We have talked about application in
  - Operating Systems – scheduling
  - Printer queue
- Some more applications
  - Implementation of several graph algorithms
    - Selection Problem
    - …
  - Discrete event simulation

The Selection Problem

- Given a list of N elements and an integer k (1 <= k <= N)
  - Find out the kth largest element in the list.
- Example:
  - List of elements: 4, 9, 0, 3, 5, 7, 10, 12, 2, 8
    - 1st largest element is: 12
    - 10th largest element is: 0
    - 6th largest element is: 6
      - 12 10 9 8 7 5 4 3 2 0 (sorted order of items)

The Selection Problem

- Some solutions
  - Alg-1)
    - Sort N elements: $O(N^2)$ (for a simple sort algorithm)
    - Retrieve the k^{th} largest element: O(1)
  - Alg-2)
    - 1. Read k elements into an array: O(k)
    - 2. Sort the elements in the array: $O(k^2)$
    - 3. For each of the remaining elements: (N-k)
      - 3.1 Compare it to the last element in array, if it is larger than the last element in the array, replace it with the last element and put it into correct spot in the array: O(k)
    - Total Running time: $O(k + k^{2} + (N-k) * k) = O(Nk)$.

The Selection Problem

- New algorithms
  - They will make use of heaps.
- A1
  - Assume we want to find the k^{th} smallest element.
    - 1. Read N elements into an array O(N)
    - 2. Apply BuildHeap to the array O(N)
    - 3. Perform k deleteMin operations. O(klogN)
      - Last operation will five us the k^{th} smallest one.
  - The solution is symmetric for finding k^{th} largest element
  - The total running time is O(N + klogN)

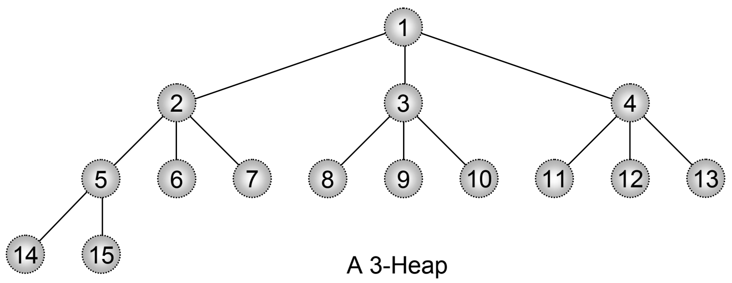
The Selection Problem

- A2
  - Use the idea of Alg-2
  - At any time maintain a set S of k largest element.
  - We want to find the kth largest element.

  - 1. Read k elements into a heap S (of size k). O(k)
  - 2. For each remaining element *(N-k) elements*
    - 2.1 Compare it with the smallest element (root) in heap S
    - 2.2 If element is larger than root, then put it into S instead of root.
    - 2.3 Percolate down the root if necessary. O(log*k*)
  - Running time = O(*k*+(N-*k*)log*k*) = O(Nlog*k*).

d-Heaps

- Exactly like a binary heap, except that all nodes have *d* children
  - A binary heap is a 2-heap.



A 3-Heap

Other Heap Operations

- Merge
  - Combining two heaps into one.
  - Is not a very simple operation
    - O(logN) cost
- We will discuss three data structure that will support merge operation efficiently
  - Leftist Heaps
  - Skew Heaps
  - Binomial Queues

Leftist Heaps

- If we use arrays to implement heaps:
  - We will have two input arrays (two heaps).
  - Combining them requires copying one array into an other: O(N) time for equal-sized heaps
- Therefore, we use a linked data structure (like a binary) to perform merge operation more efficiently.

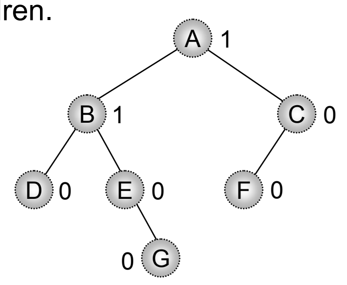
Leftist Heaps

- A leftist heap has
  - A structural property
  - An order property
- Order property is the same with ordinary binary heaps.
- Structural property is little bit different.

Leftist Heap Property

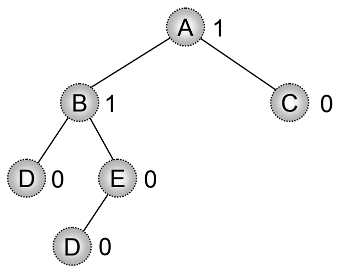
- **Definition**:
  - Null path length, npl(X), of any node X is defined as the length of the shortest path from X to a node without two children.
  - Null path length of a node with zero or one child is 0.
  - Null path length of a NULL node is defined to be -1.
- Null path length of a node is 1 more than the *minimum* of the null path lengths of its children.

$$\begin{aligned} npl(A) &= 1 + \min\{npl(B), npl(C)\} \\ &= 1 + \min\{1,0\} \\ &= 1 + 0 = 1 \end{aligned}$$

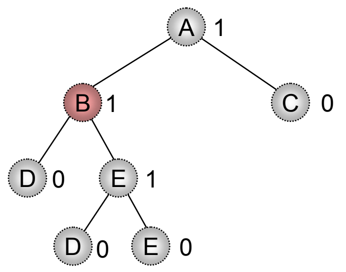


Leftist Heap Property

- Leftist heap property is: 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.



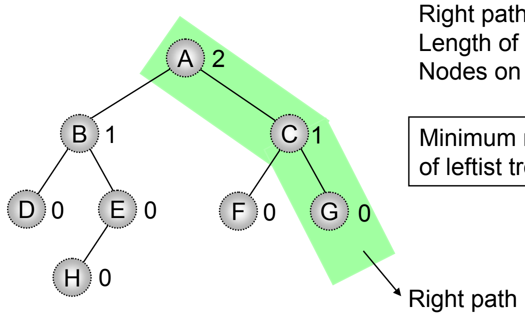
A Leftist Heap



Not a Leftist Heap

Theorem

- Theorem:
  - *A leftist tree with r nodes on the right path must have at least 2^r-1 nodes.*



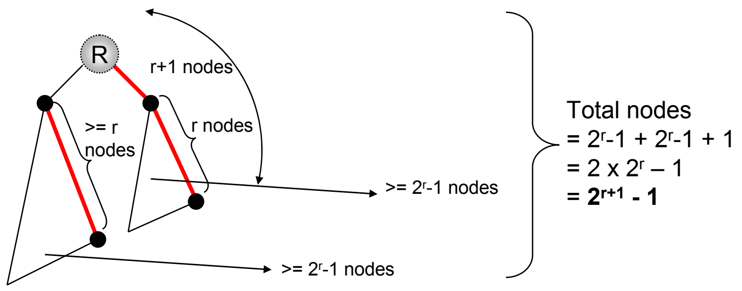
Right path: A, C, G
Length of the right path = 2
Nodes on the right path = 3

Minimum number of nodes of leftist tree: 2^3-1 = 7

npl(root) = legth_of_right_path

Theorem - Proof

- Proof:
  - By induction
    - If r = 1 then, there has to be at least one node
      - 2^r-1 =2^1-1 = 1
    - Assume theorem is true for 1,2,3,…r.
    - A tree with r+1 right path nodes would look like the following:



From the previous theorem it follows that the maximum number of nodes in the right path of a *leftist heap* of size *N* nodes is:

$$\lfloor \log(N+1) \rfloor$$

General idea for leftist heap operations is to perform all the work on the right path, which is the shortest.

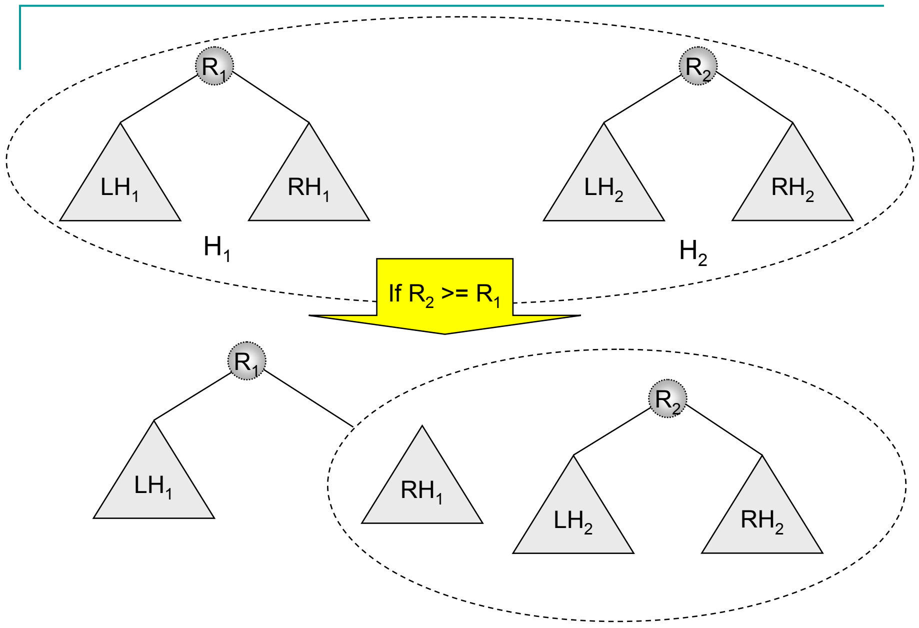
Since, *inserts* and *merges* on the right path could destroy the Leftist heap property, we need to restore the property.

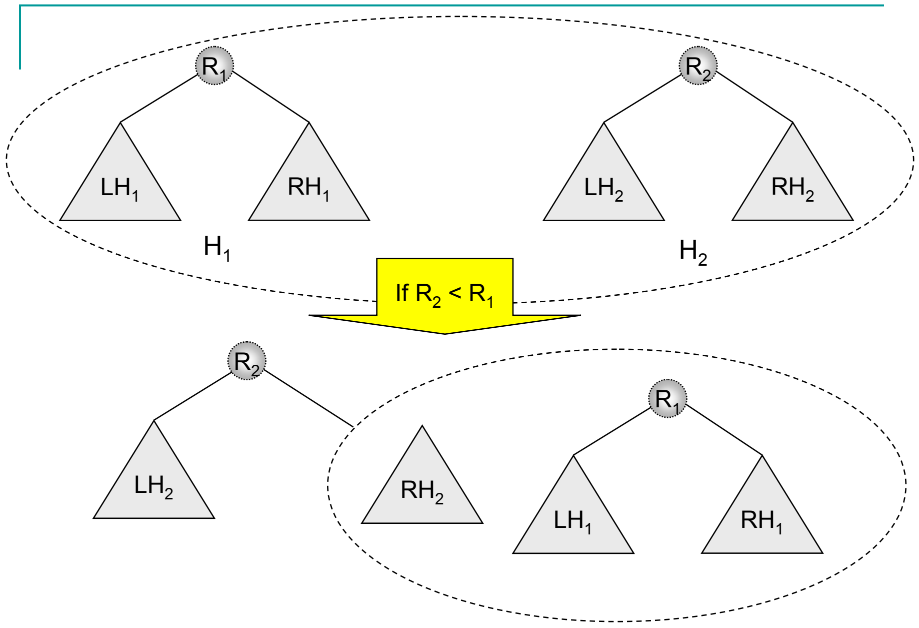
Operations

- We will see the following leftist heap operations
  - Merge
    - Merges two leftist heaps into a single one.
  - Insert
  - DeleteMin

Merge

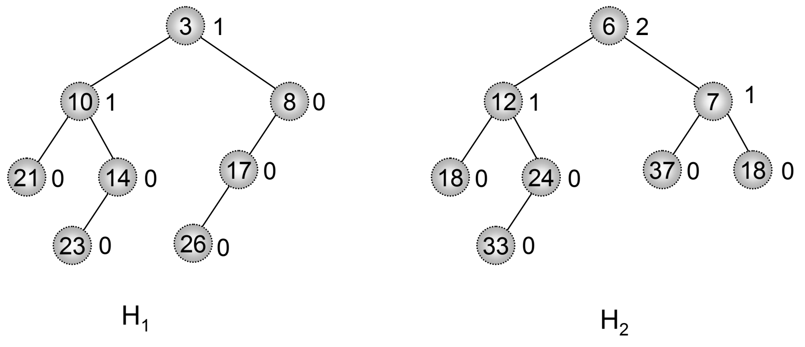
- Merge two heaps H1 and H2.
- A recursive algorithm
  - 1. If either one is empty than it is trivial to merge: return the non-empty heap as the result of merge.
  - 2. Else:
    - 2. 1 Merge the heap with the larger root with the right sub-heap of the heap with the smaller root. (this is ca recursive call)
    - 2.2 Make the resulting heap as the right child of the heap with the smaller root.
    - Swap the children of the root of the new heap, if npl(right-child) is greater than npl(left-child).
    - Update npl(root): npl(root) = npl(right-child) + 1;


R_1
R_2
LH_1
RH_1
LH_2
RH_2
H_1
H_2
If R_2 >= R_1
R_1
R_2
LH_1
RH_1
LH_2
RH_2


R_1
R_2
LH_1
RH_1
LH_2
RH_2
H_1
H_2
If R_2 < R_1
R_2
R_1
LH_2
RH_2
LH_1
RH_1

Example – Merge the following heaps

merge (H_1, H_2)

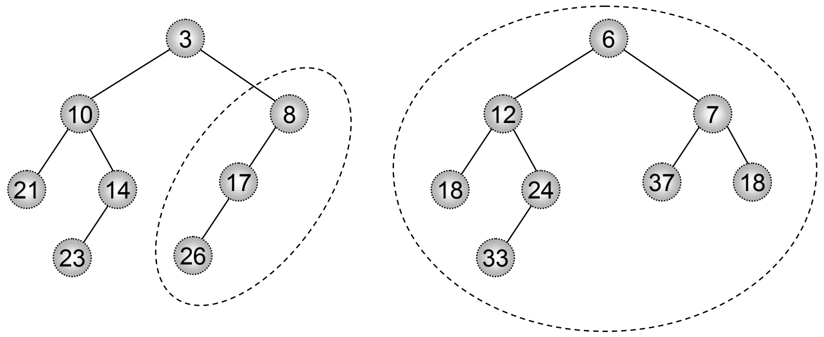


Both are *leftist heaps*

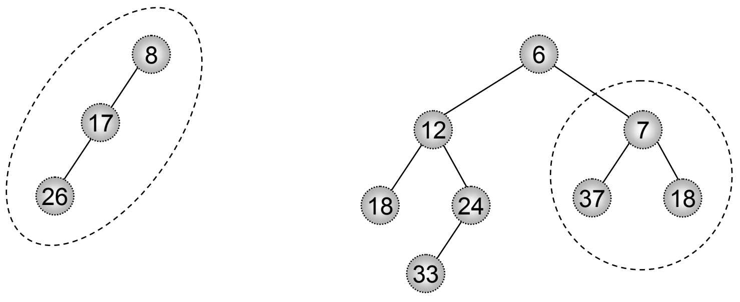
Level 1

Since 6 is greater than 3, we should recursively merge H_2 with Right subheap of H_1

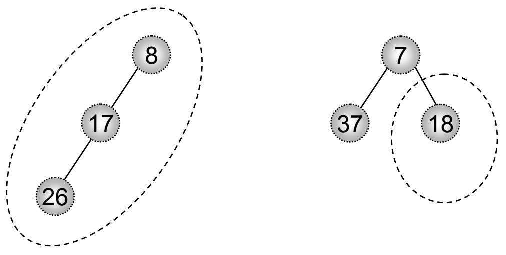
merge (H_1->right, H_2)



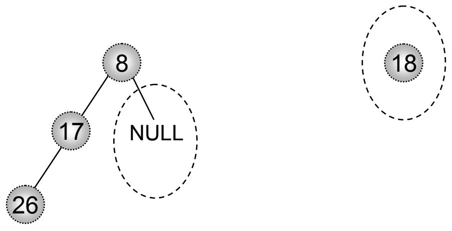
Level 2



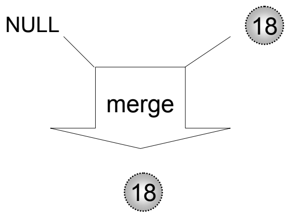
Level 3



Level 4

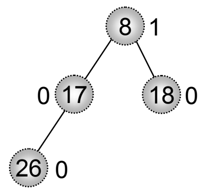


Level 5

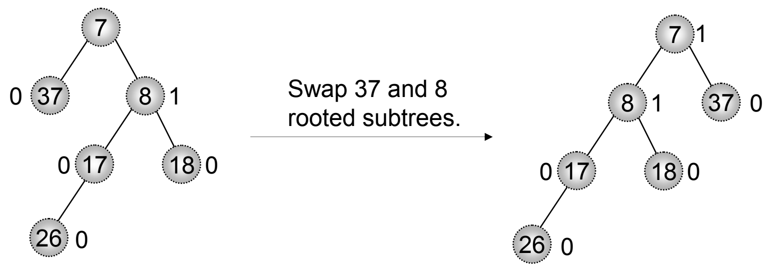


Now, start backtracking

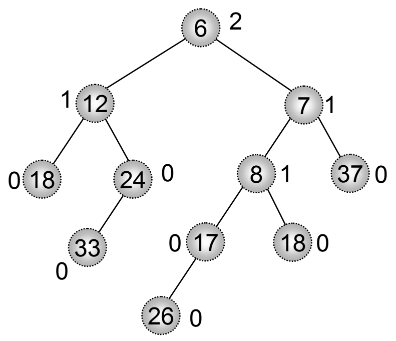
Back to Level 4



Back to level 3

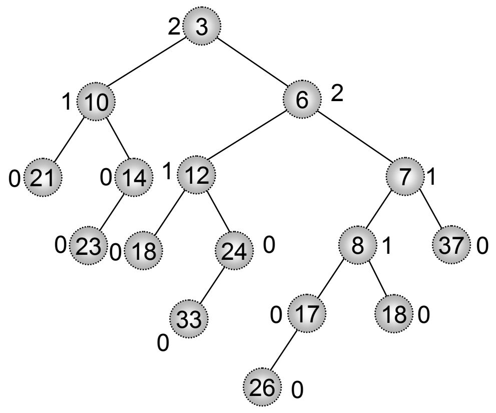


Back to level 2



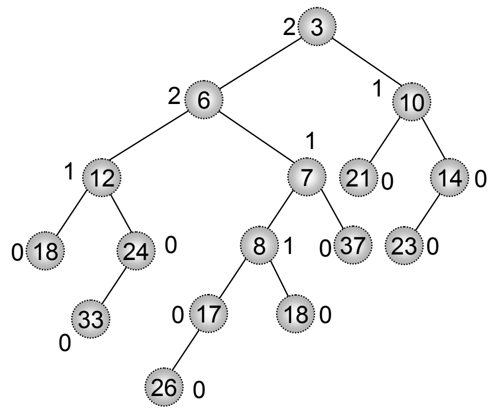
No need for swapping, since
npl(12) = npl(7)

Back to level 1



Swap 10 and 6

Back to level 1



This is the new heap after merge!

Implementation

```
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;
          void insert( const Comparable & x );
          void deleteMin( );
          void deleteMin( Comparable & minItem );
          void merge( LeftistHeap & rhs );

        private:
          LeftistNode<Comparable> *root;   /* data member */
          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;
};
```

```
// Merge right handside heap into this heap. rhs becomes empty.

template <class Comparable>
      void LeftistHeap<Comparable>::merge( LeftistHeap & rhs )
      {
         if( this == &rhs )    // Avoid aliasing problems
            return;

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

```
// calls merge1 to to actual merge. Makes sure that when merge1 is called,
// h1 is the node that has smaller root than h2.


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 );
      }
```

```
// assumes heaps are not empty
// assumes that h1 has smaller root.

 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;
      }
```