Priority Queues (Heaps)

- Problem: scheduling jobs in a multiuser computer system.
- Solution 1:
  - jobs are placed in a queue
  - the scheduler will take the first job on the queue, run it until finishes or its time limit is up.
  - If not finished place the job at the end of queue.

  => Very short jobs take a long time because of waiting.
- Solution 2: Shortest Job First
  - Run the shortest jobs first
- Need a special kind of queue => priority queue.

Priority queue

- Allows at least two operations:
  - insert
  - deleteMin – finds, returns, and removes the minimum element in the priority queue.

Simple Implementations

- Implementation 1:
  - Use a linked list
  - insert: insert at the front => O(1)
  - deleteMin: traverse the list and delete the minimum => O(N)

  => Too expensive
- Implementation 2:
  - Use a BST
  - insert and deleteMin => O(log N)
  - Removing the min => unbalanced tree and the time degrades
  - Variant: use AVL trees.

  => Supports operations that are not required and is complex.

Binary Heap

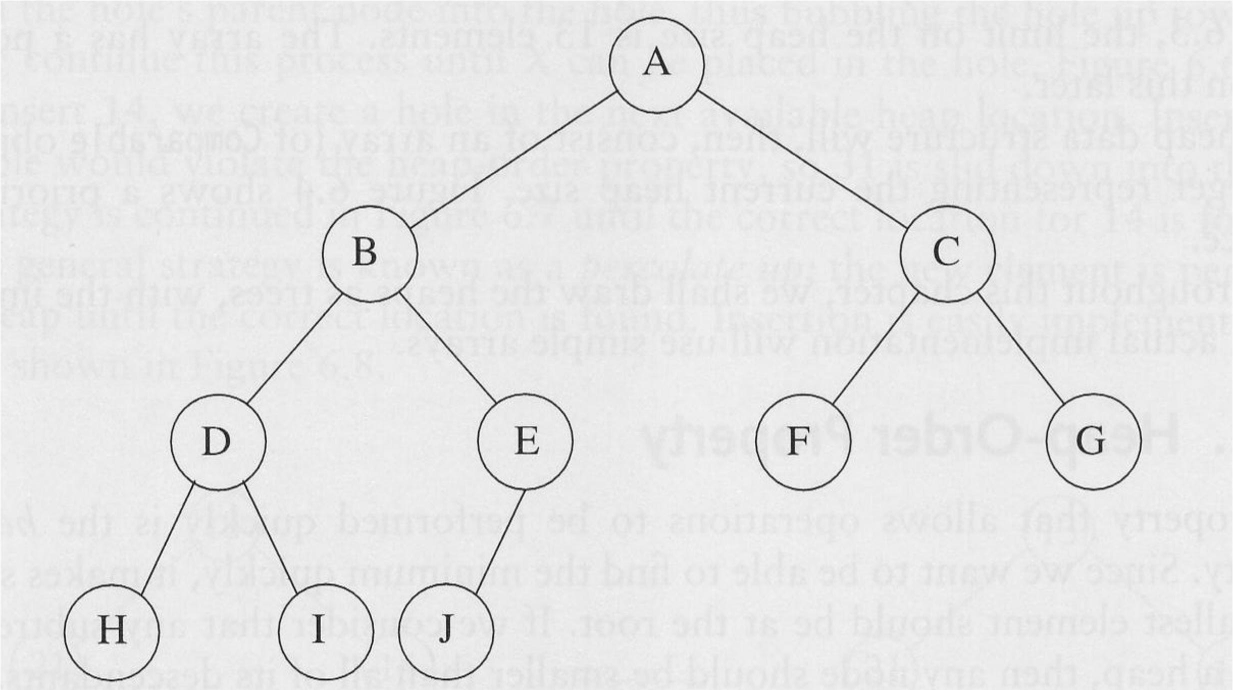
- Binary heap properties:
  1. Structure property
  2. Heap-order property
- Structure property:

  A binary heap is a complete binary tree.

  (i.e. the tree is completely filled except the last level which is filled from left to right).
- A complete tree of height h has between 2^h and 2^{h+1} -1 nodes

  => height = O(log N)
- It can be represented in an array.
- If an element is at position *i* in the array:
  – Left child: position *2i*
  – Right child: position *2i+1* => it is after the left child
  – Parent: position $\lfloor i/2 \rfloor$
- Need an estimate of the maximum heap size.

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

Binary Heap Class

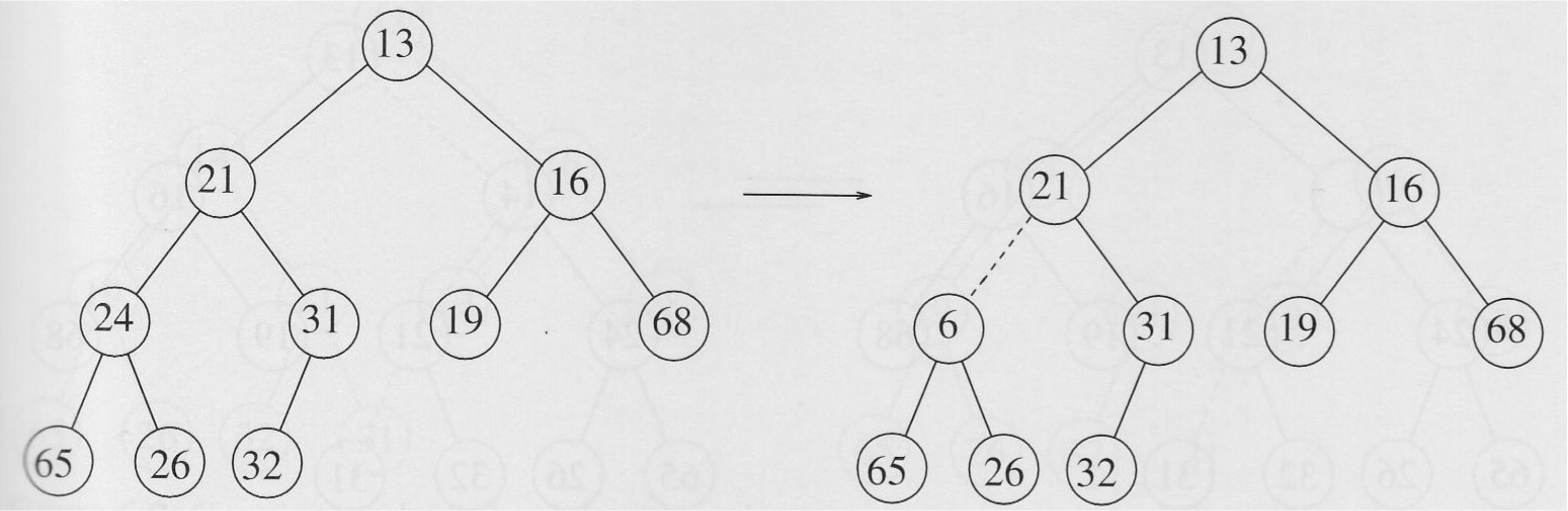
```
template <typename Comparable>
class BinaryHeap
{
        public:
                explicit BinaryHeap( int capacity = 100 );
                explicit BinaryHeap( const vector<Comparable> & items )

                bool isEmpty( ) 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 );
};
```

Heap-Order Property

- Heap-Order Property:

  In a heap for every node X, the key in the parent of X is smaller than (or equal to) the key in X, with the exception of the root.
- The minimum element always at the root

  => findMin in O(1).



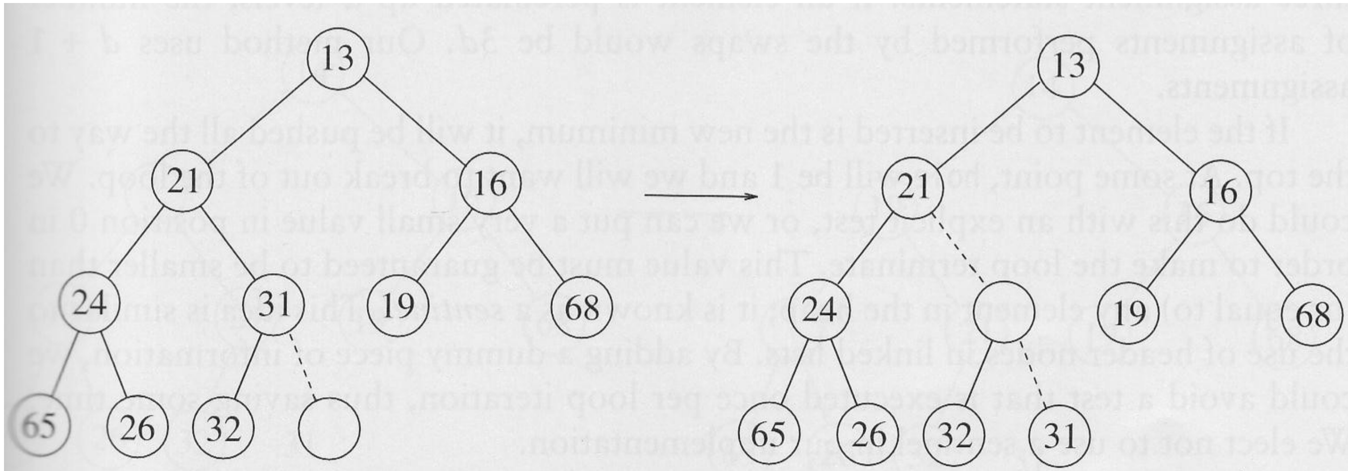
Insert

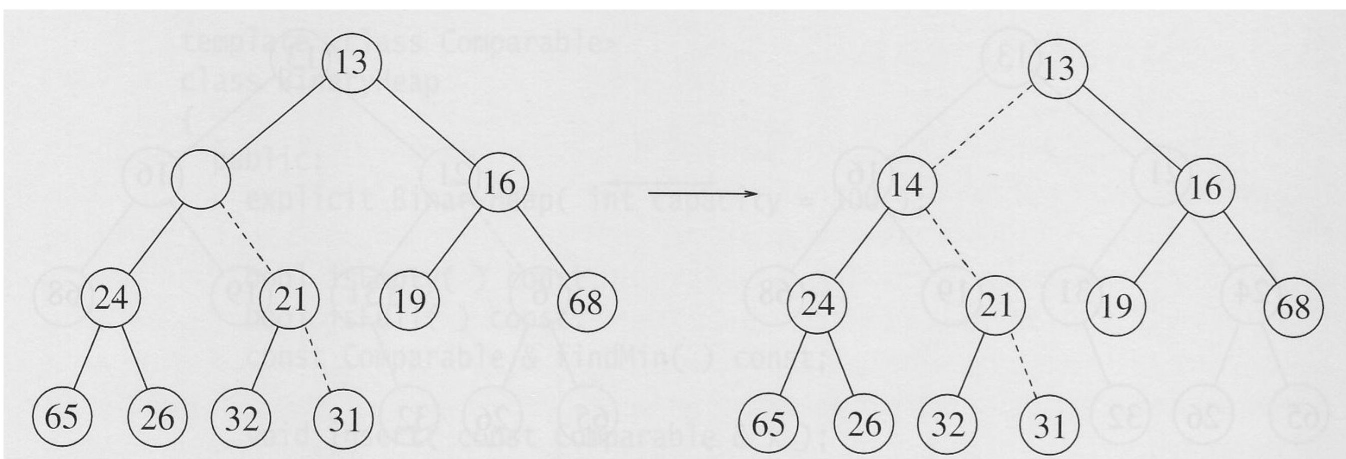
- Insert X:
  - Create a “hole” in the next available location.
  - If X can be placed without violating the heap order => place it.
  - Else slide the hole’s parent into the hole, bubbling up the hole towards the root.
  - Continue the process until X can be placed in the hole.

=> Percolate up strategy

Insert

insert 14





Insert

```
void insert( const Comparable & x )
{
        if( currentSize == array.size( ) - 1 )
                array.resize( array.size( ) * 2 );

                // Percolate up
        int hole = ++currentSize;
        for( ; hole > 1 && x < array[ hole / 2 ]; hole /= 2 )
                array[ hole ] = array[ hole / 2 ];
        array[ hole ] = x;
}
```

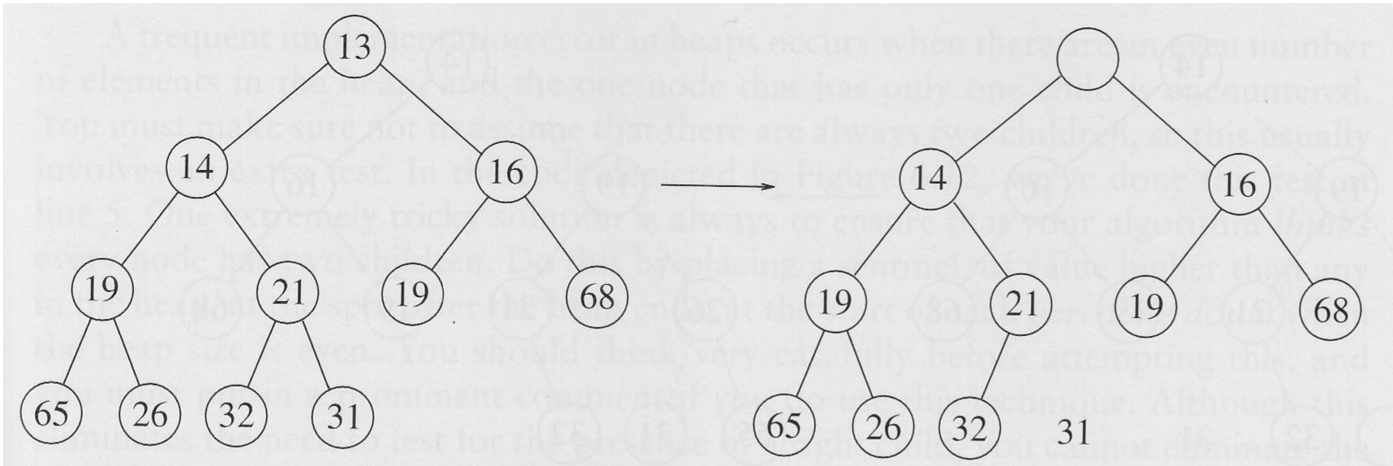
$T(N) = ?$

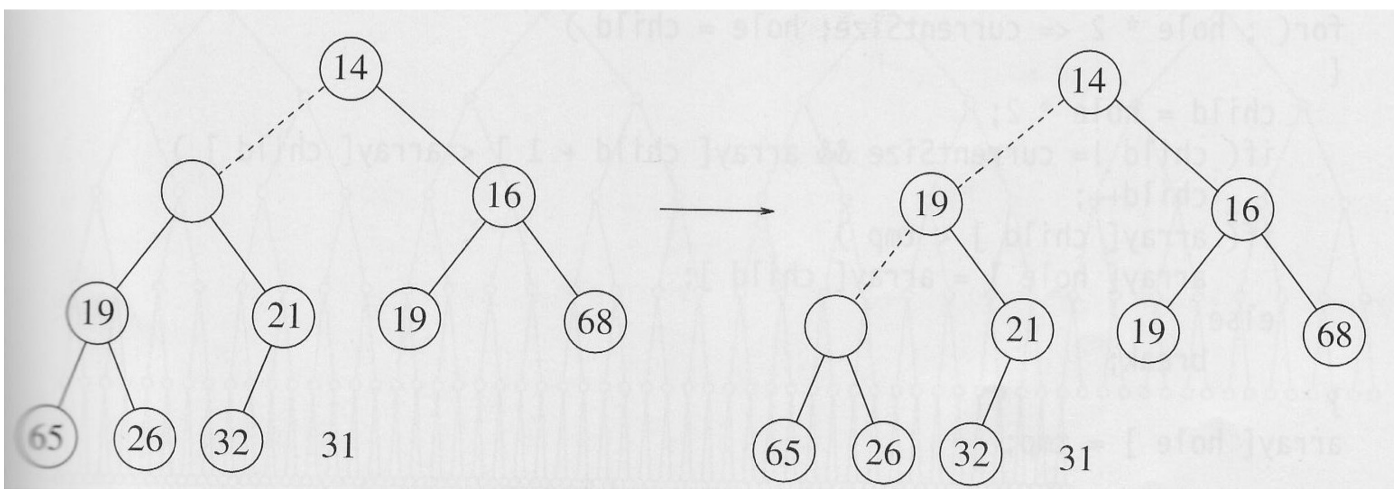
DeleteMin

- DeleteMin:
  - Remove minimum => create a hole at the root.
  - X the last element in the heap must move
  - If X can be placed in the hole => place it
  - Else slide the smaller of the hole’s children into the hole, pushing the hole down one level
  - Repeat the process until X can be placed in the hole.

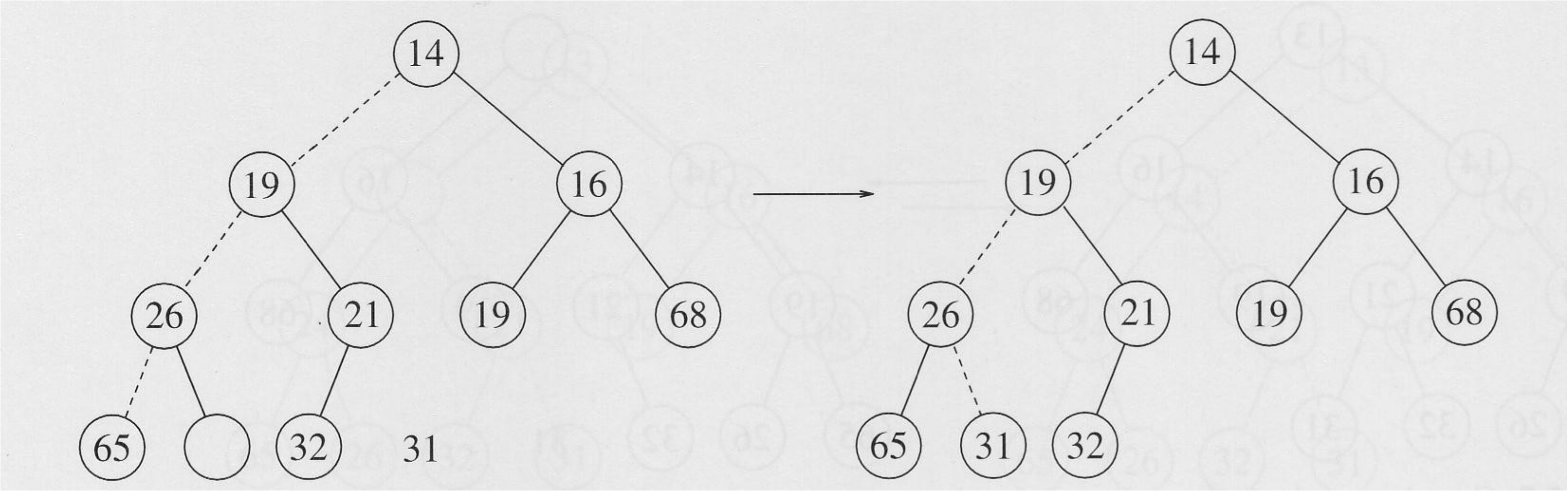
=> Percolate down strategy

DeleteMin


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


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

DeleteMin



DeleteMin

```
void deleteMin( Comparable & minItem )
{
        if( isEmpty( ) )
                throw UnderflowException( );

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

void percolateDown( int hole )
{
        int child;
        Comparable 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;
}
```

T(N) = ?

Other Heap Operations

- decreaseKey(p, d)
  - Lowers the value of the item at position p by a positive amount d.
  - Use percolate up to restore the heap-order property.
  - Operating System Example: sysadmins make their programs run with highest priority.
- increaseKey(p, d)
  - Increases the value of the item at position p by a positive amount d.
  - Use percolate down to restore the heap-order property.
  - Operating System Example: drop the priority of a process.
- remove(p)
  - Removes the node at position p.
  - decreaseKey(p, ∞) then deleteMin()
  - Operating System Example: when a process is terminated by user it is removed from the queue.

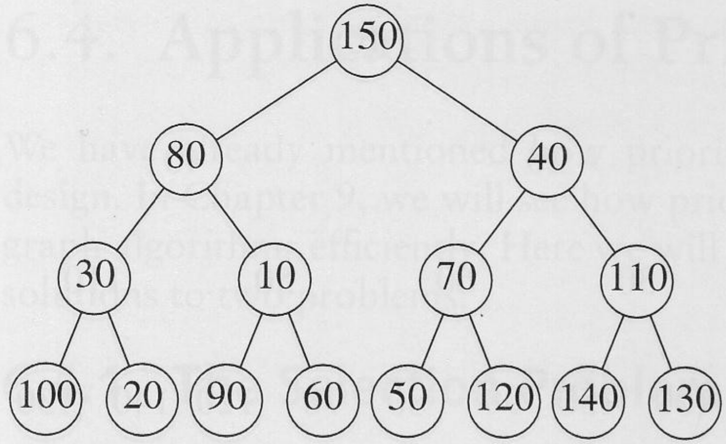
buildHeap

- Place N items into an empty heap.
- Solution 1: N successive insert operations.
- => O(N log N) worst case running time
- Solution 2: place the N items into the tree in any order and then use percolateDown(i)

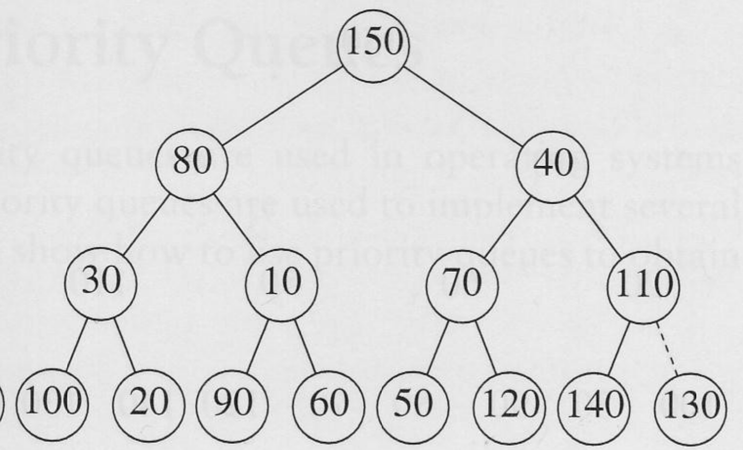
  => O(N)

```
void buildHeap( )
{
    for( int i = currentSize / 2; i > 0; i-- )
            percolateDown( i );
}
```

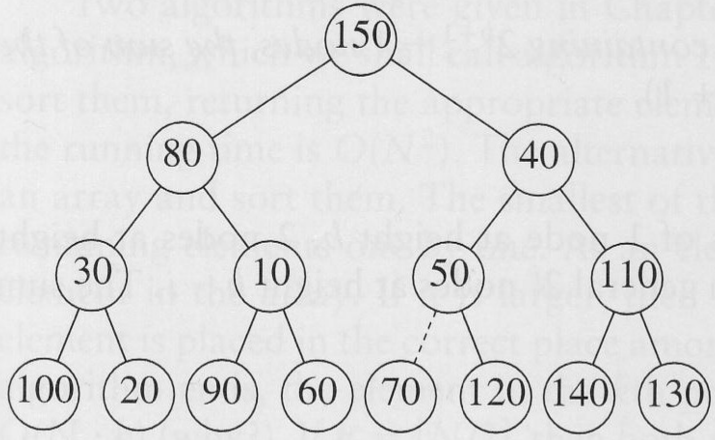
buildHeap



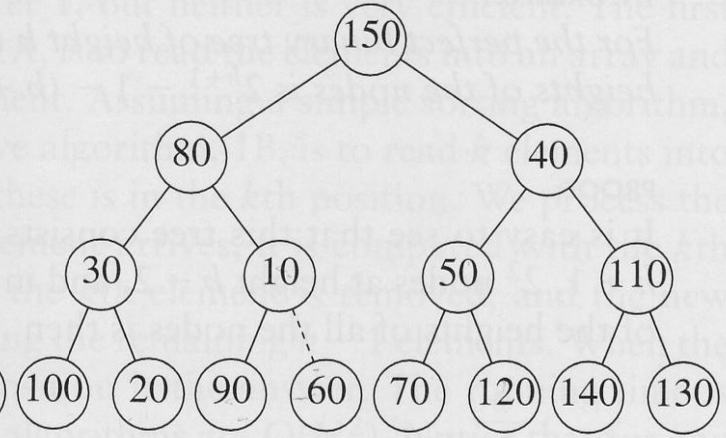
initial heap



percolateDown(7)

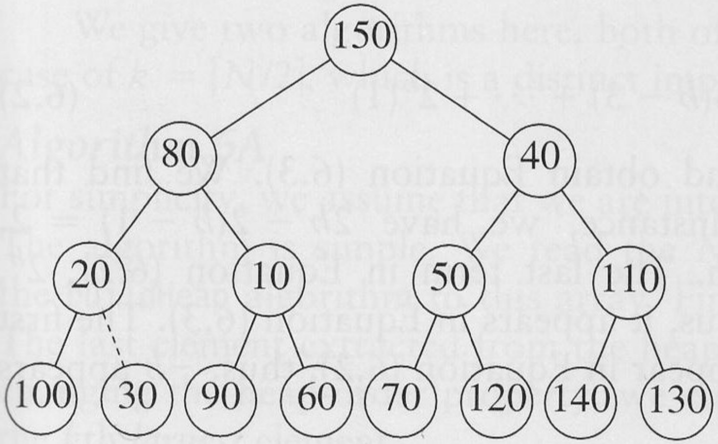


percolateDown(6)

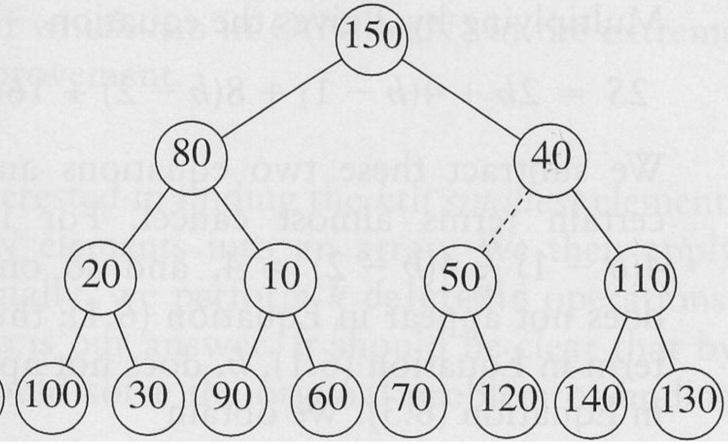


percolateDown(5)

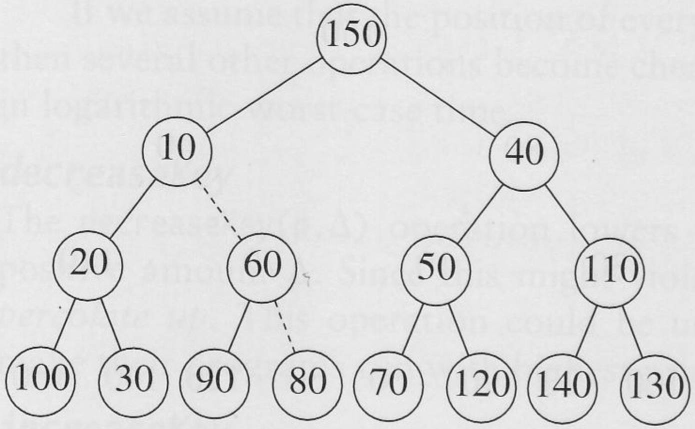
buildHeap



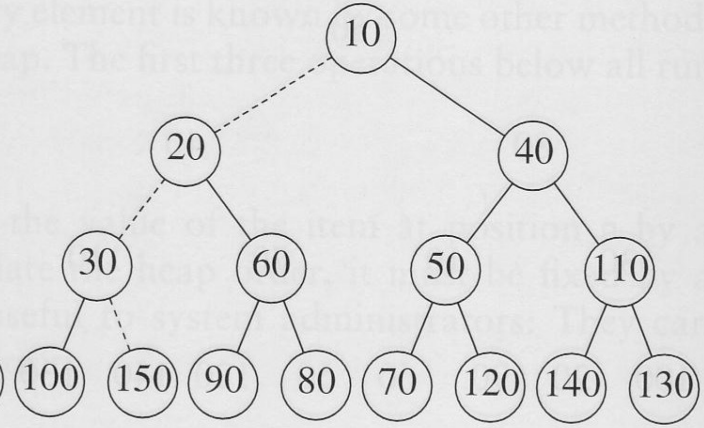
percolateDown(4)



percolateDown(3)



percolateDown(2)



percolateDown(1)

buildHeap

- Running time for buildHeap = O(N)
- Dashed lines = 2 comparisons:
  - one to find the smaller child
  - one to compare the smaller child with the node
- What is the maximum number of dashed lines?
- Maximum number of dashed lines = sum of the heights of all nodes in the heap
- Show that it is O(N)

buildHeap

- 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:

  1 node at height h, 2 nodes at height h-1, …,

  2^i nodes at height h-i

  Sum of the heights:

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

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

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

$$2\mathrm{S} - S = -h + 2 + 4 + 8 + .. + 2^{h-1} + 2^h = (2^{h+1} - 1) - (h+1)$$

h = O(log N) => S = O(N)

Applications: Selection Problem

- Selection problem: find the k-th largest element in a list of N elements.
- Algorithm A: (for k-th smallest)
  - Read N elements into an array => O(N)
  - Apply buildHeap => O(N)
  - Perform K deleteMin => O(k log N)

  The last element extracted from the heap is the answer.
- $T(N) = O(N + k \log N)$
- If $k = O(N / \log N)$ => $T(N) = O(N)$
- If $k = \lceil N/2 \rceil$ (finds the median) => $T(N) = \Theta(N \log N)$
- If $k = N$ => sorting algorithm called heapsort

Applications: Selection Problem

- Algorithm B: (for k-th largest)
  - Read first k elements into an array => O(k)
  - Apply buildHeap => O(k)
  - Compare the new element with the k-th largest => O(1)
  - If new element > k-th largest in the heap then insert => O(log k)

  The k-th largest element in the heap is the answer.
- $T(N) = O(k + (N-k) \log k) = O(N \log k)$
- If $k = \lceil N/2 \rceil$ (finds the median)

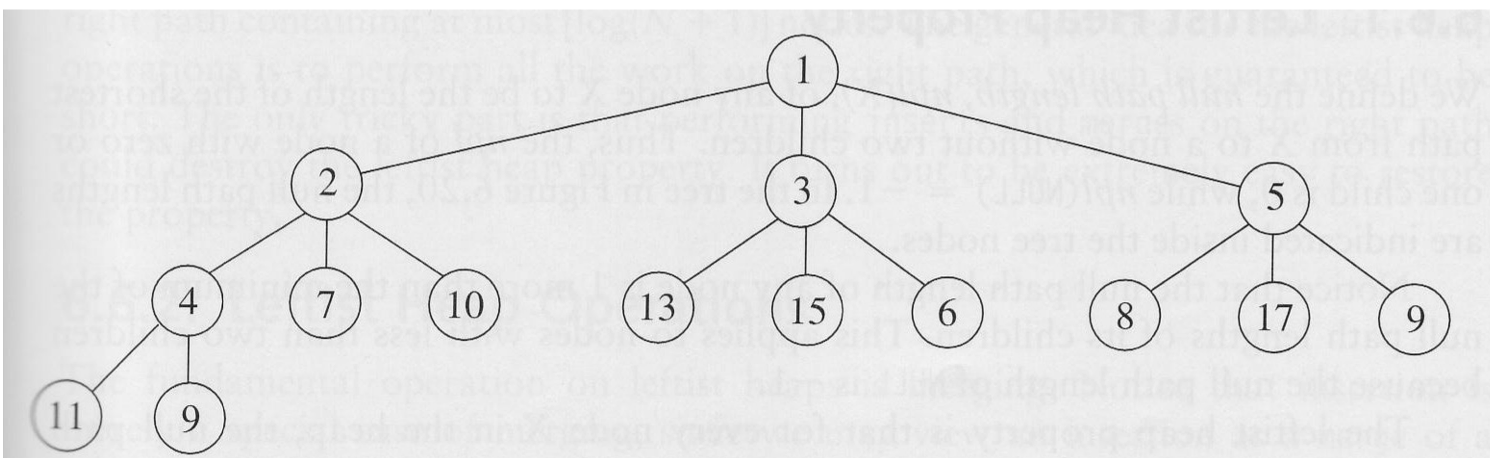
  => $T(N) = \Theta(N \log N)$

Applications: Event Simulation

- Simulate a queueing system: Customers arrive and wait in line until one of the k tellers is available.
- Simulation => processing two types of events
  - A customer arriving
  - A customer departing
- Implement the waiting line of customers as a priority queue.
- Easy to find the nearest event in the future

d-Heaps

- All nodes have d children
- Improves the running time of insert => $O(\log_d N)$
- For large d, deleteMin is more expensive => d-1 comparisons
- Used when the priority queue is too large to fit in main memory.
- Example: 3-heap



Merge Heap

- Merge heap: combine two heaps into one
- Complex operation.
- Special heap implementations to support the merge operation:
  - Leftist heaps: heaps that are kept intentionally unbalanced.
  - Skew heaps: self adjusting version of leftist heaps
  - Binomial queues: a collection of heap ordered binomial trees. The number of nodes at depth h in a binomial tree is the binomial coefficient (k,d)