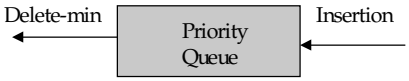
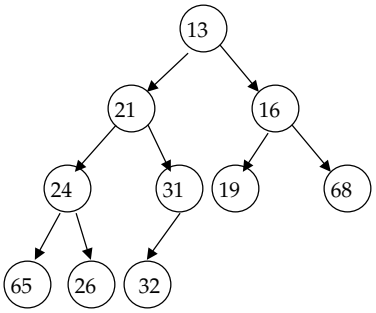
Priority Queues

- Set of records with associated **priorities**
  - jobs waited to be printed - priority may be based on size of job (print small jobs first)
  - Processes waiting to be run - priority based on expected time to completion, or real time requirements
- Data structure that supports following operations
  - insertion
  - deletion of element with minimum priority



Binary heap

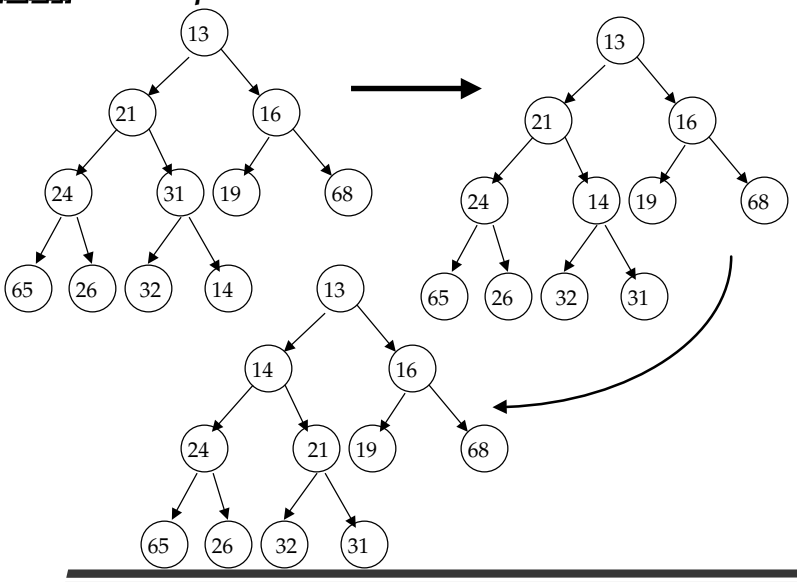
- Binary heap (or heap) is a complete binary tree having the following **heap order property**:
  - for every node X, the key in the parent of X is smaller than the key at X.
- Heaps stored using sequential representation of complete binary trees
- Smallest element is at the root of the heap



Insertion of x into a binary heap

- Create a hole in the next available location
- If x can be placed in the hole, finished
- Otherwise, **percolate x up** into it's parent's location and recurse
- Terminate if x is switched with the key at the root.

Example - Insert 14



Code for insertion

- Place a small element in position 0 of the heap to avoid testing for root
  - value known as a **sentinel**
- Routine does not use swaps as it percolates up
  - percolating up using swaps would require 3d assignments for d percolates
  - Code shown uses d+1 assignments

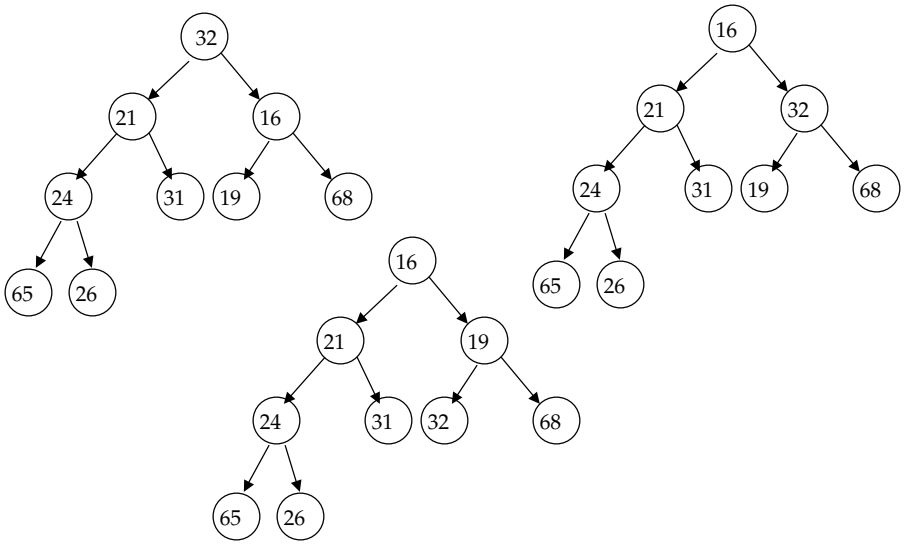
Code for insertion

```
Procedure insert (x:element to be inserted; H: priority queue);
    vari i: integer;
    begin
        if H.size = Maximum then error
        else begin
        H.size : = H.size +1
        i := H.size
        while H.element[i div2] > x do
        begin
                H.element[i] := H.element[i div 2];
                i := i div 2
        end
        H.element[i] := x
    end!
```

Delete-min

- Key,at root is always deleted
- Move last key, x, in heap into root
- **Percolate down** until it is smaller than both of its children
  - if x is smaller than both of its children, halt
  - otherwise swap x with its **smaller** child and repeat

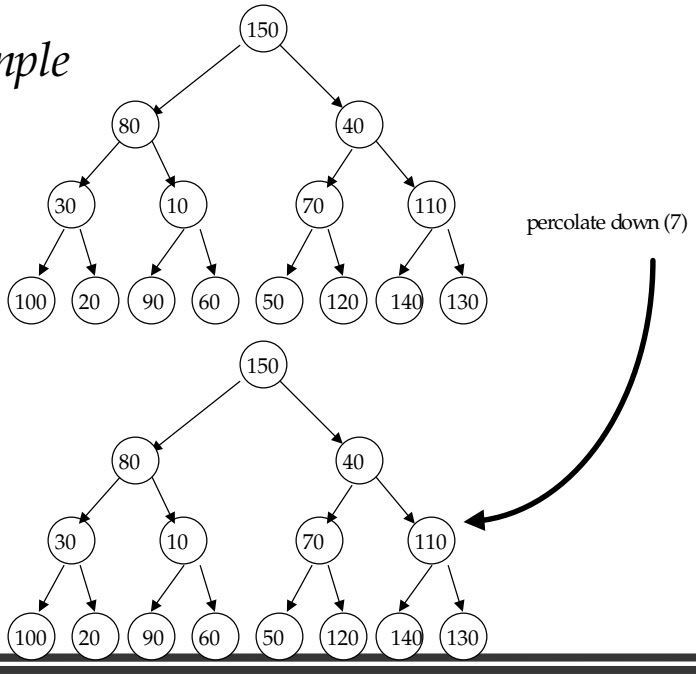
Example

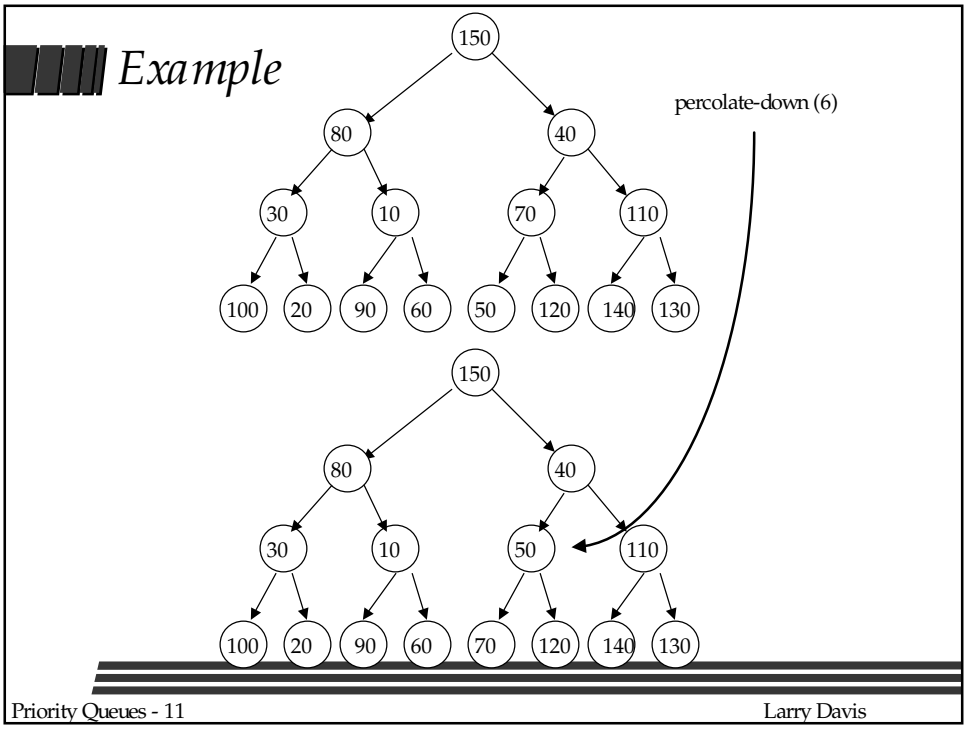


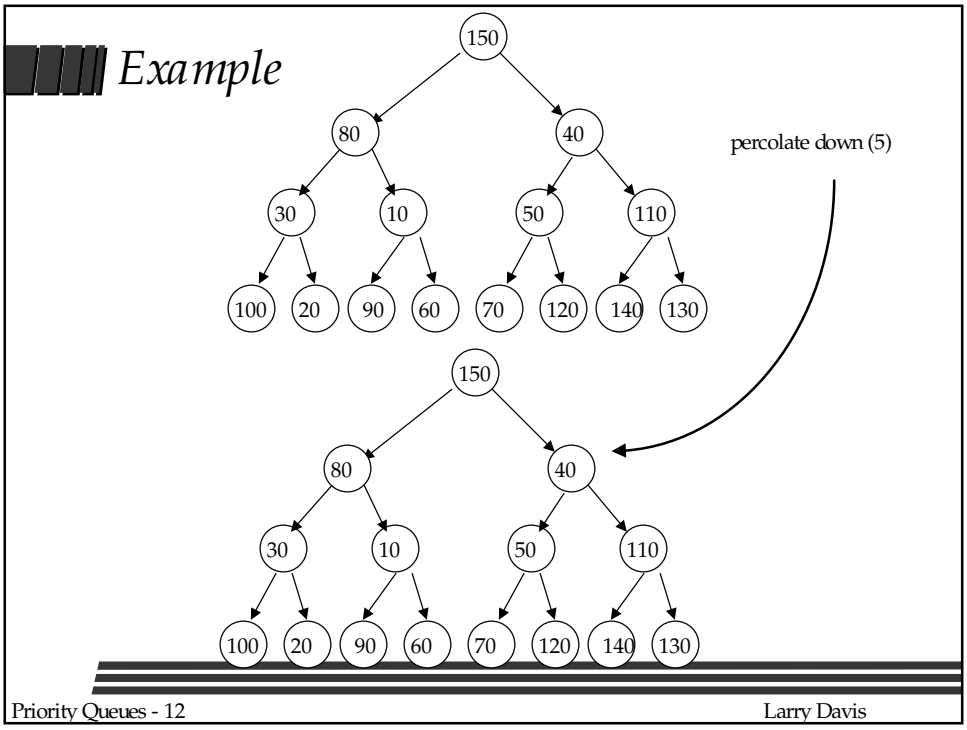
Building a heap

- A heap can be built from n keys in O(n) time
- Insert the keys in any order, maintaining the structure property (complete BT)
- Then percolate keys down from "bottom" to "top".
  - percolating a node down can only take time proportional to the height of the node
  - But the "total" height of a complete BT is O(n)

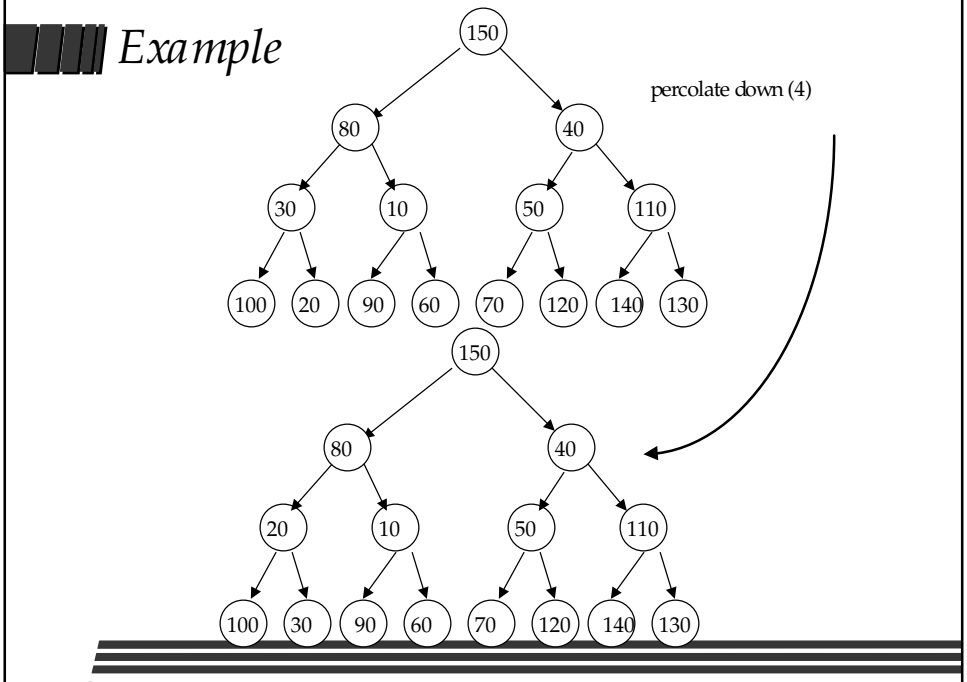
Example



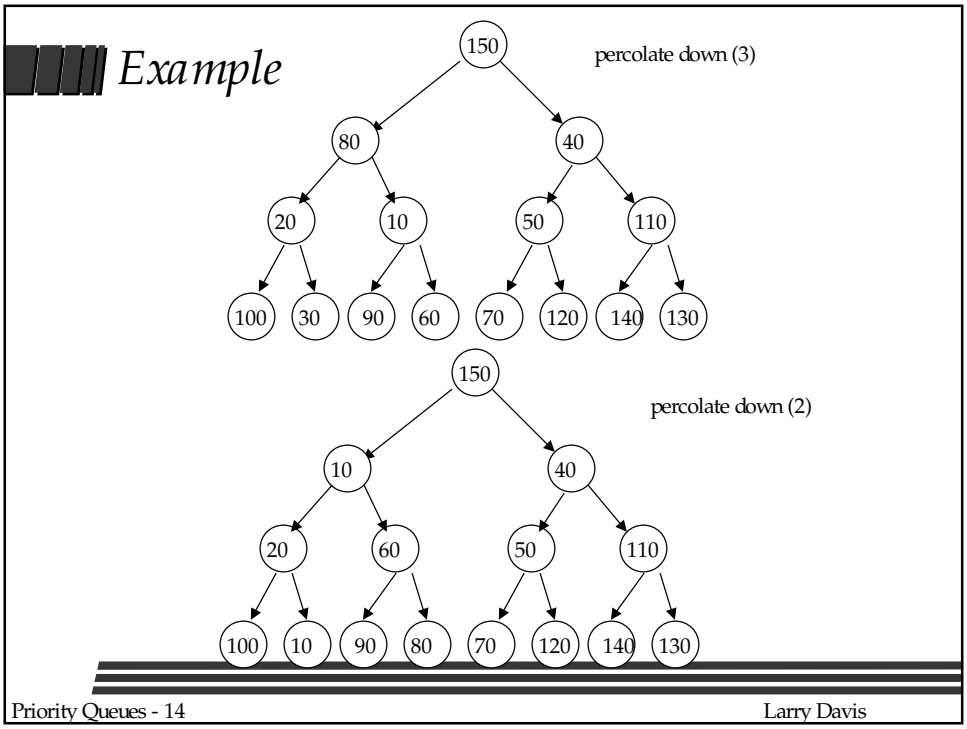

Example
percolate-down (6)
150
80
40
30
10
70
110
100
20
90
60
50
120
140
130
150
80
40
30
10
50
110
100
20
90
60
70
120
140
130


Example
percolate down (5)
150
80
40
30
10
50
110
100
20
90
60
70
120
140
130
150
80
40
30
10
50
110
100
20
90
60
70
120
140
130

Example



Example



Proof of linearity of heap creation

- For a complete binary tree of height h containing 2^{h+1} nodes, the sum of the heights of the nodes is $2^{h+1} - 1 - (h+1)$
  - There is 1 node at height h, 2 at height (h-1), 2^2 at height (h-2) and 2^i at height (h-i)

$$S = \sum_{i=0}^{h} 2^i (h-i) = h + 2(h-1) + 4(h-2) + 8(h-3) + \cdots + 2^{h-1}(1)$$

$$2S = 2h + 4(h-1) + 8(h-2) + \cdots + 2^h(1)$$

  - Subtracting these yields

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

Application - nearest neighbor finding in hierarchical structures

- Given:
  - hierarchical data structure containing points, lines, etc. - data objects (dobj)
  - a query object (qobj)
- Goal: Find the data object in the hierarchical data structure that is closest to the query object
- Approach: Recursive traversal of the data structure, using priority queue to store lower bounds on distance to nearest dobj in nodes

Initialization

- Insert root of data structure into Q
  - Elements in Q are pairs (node, cost), where
    - » node is a non-leaf node of the hierarchical data structure\
    - » cost is the lower bound on the distance between qobj and any dobj stored in node.
  - Insert(root, 0) into Q
- Nearest-obj = NULL
- Nearest-dist = ∞

While loop

- Body of algorithm is a while loop that analyzes top element from Q as long as its cost < Nearest-dist

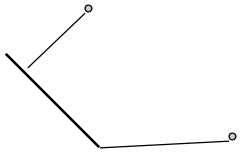
```
While (~ empty(Q)  AND   Q.first.cost < Nearest-dist
    node = delete -first(Q)
    if leaf(node) then
            for each dobj in node
            if distance(dobj,qobj) < Nearest-dist then
                    nearest-obj = dobj
                    nearest-dist = distance(dobj, qobj)
            else
            for each non-white child of node
            insert-Q (Q, child, lower-bound-distance(child, qobj))
```

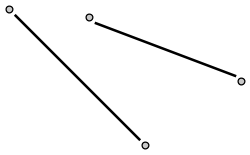
Key distance functions

- Distance
  - depends on types of objects being compared
    - » point - point
    - » line segment - point
    - » line segment -line segment
- Lower-bound-distance
  - depends both on type of qobj and data structure node
    - » consider only quadtrees, with square nodes
    - » qobj can be either a line or a point

Distance

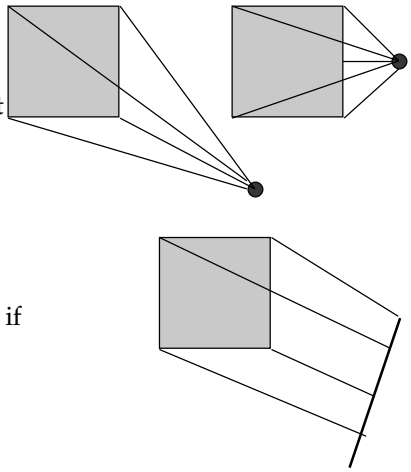
- point-point
  - just use Euclidean distance between points
- point-line segment
  - either the perpendicular distance from the point to the line segment
  - or, the shorter of the distances to the line segment end points
- line segment-line segment
  - minimum end point to line segment distance if segments do not intersect
  - 0 otherwise



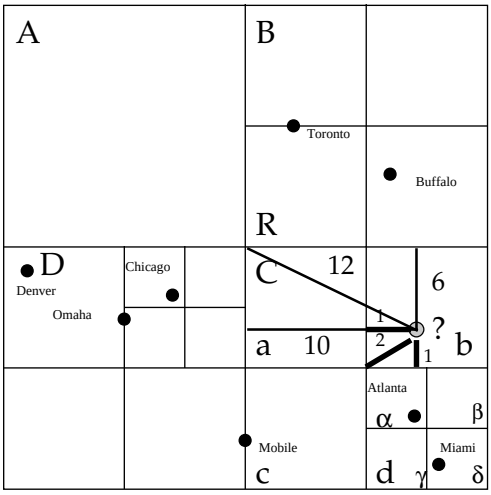


Lower bound distance

- Point to quadtree node
  - minimum distance between any corner or side of the square and the point if point is outside square
  - 0 otherwise
- Line segment to quadtree node
  - minimum distance between any corner or side of the square and the line segment if segment is outside square
  - 0 otherwise

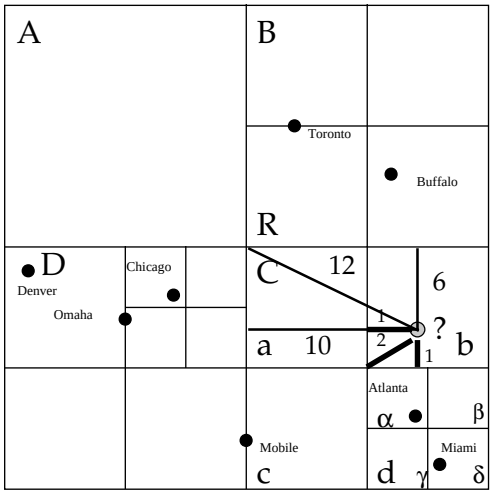


Example



- Initially, Q contains (R,0) and it is dequeued
- Then, we insert (C,0), (B,6), (D,10)
- (C,0) is dequeued
- (d,1) and (c,2) are queued
- (d,1) dequeued

Example



- (d,1) dequeued
- (α,1) and (δ,3) are queued
- (c,1) dequeued - distance from ? to Mobile (10) becomes new nearest-dist
- (α,1) dequeued and distance from ? to Atlanta computed = 2.5.
  - becomes new nearest-dist
- Done - top Q element has cost (3) > nearest-dist=2.5

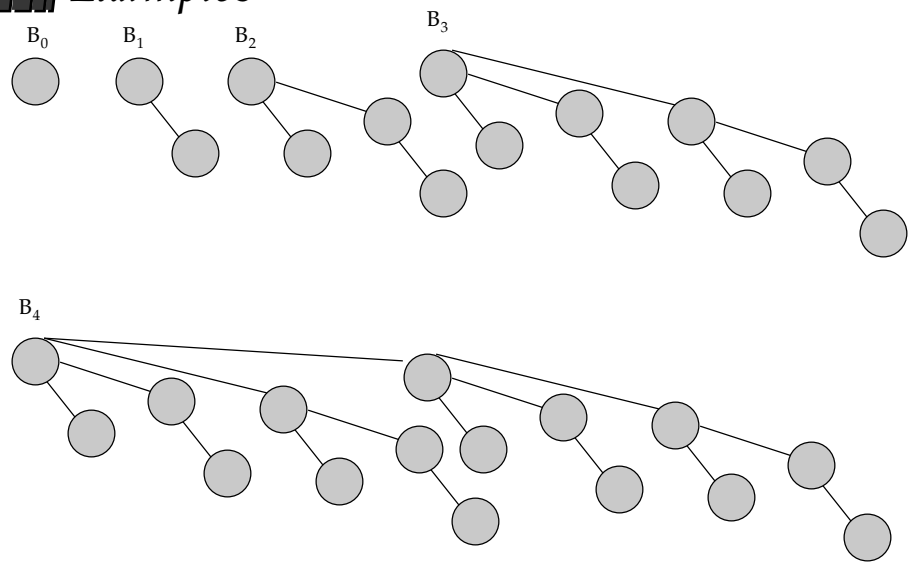
Binomial queues

- Consider problem of merging two priority queues
  - binary heap solution would require inserting the keys one at a time from H_1 into H_2. This would lead to a linear algorithm
- Binomial queues allow merging in log(n) times while still supporting fast insertion and delete-min

Binomial queues

- Binomial queue is a collection of trees
  - each tree is heap ordered
  - collection of trees represented as a forest - root node whose sons are the roots of the heap ordered trees
- Each of these trees is a **binomial tree**.
  - There is only one binomial tree of any given height.
  - B_0: binomial tree of height 0, is a single node
  - B_K: binomial tree of height k - attach a B_{K-1} to the root of another B_{K-1}

Examples

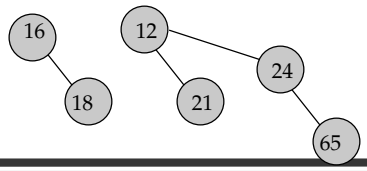


Binomial queues

- Binomial tree, B_K, consists of a root and children $B_0, B_1, ..., B_{K-1}$.
- B_K has exactly 2^k nodes
- Number nodes at depth d is the binomial coefficient $\binom{k}{d}$
- How can we represent an arbitrary priority queue as a binomial queue?
  - Expand the size of the priority queue in binary
  - Include a binary tree for each "1" in the binary representation of the size.

Representing priority queues as binomial queues

- Consider a priority queue containing 13 elements
  - 13 = 1101 in binary
  - so, include B_3, B_2 and B_0 in the forest of binomial trees representing the priority queue.
- Example: binomial queue H_1 with 6 elements



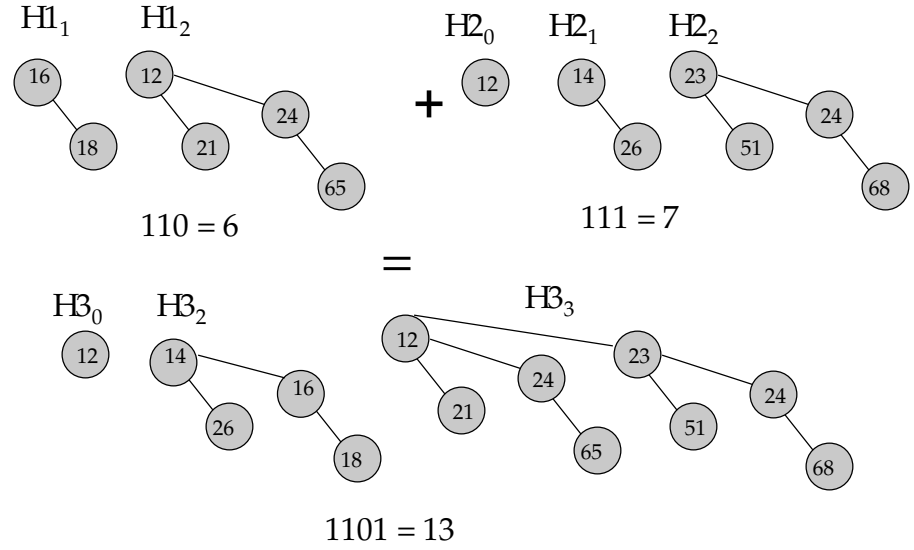
Binomial queue operations

- Merging is the basic operation
  - accomplished by "adding" the two queues together

$$\begin{array}{r} 10010 \\ +10111 \\ \hline 101001 \end{array}$$

  - merging two binomial trees takes constant time
  - only log(n) pairs of trees to merge in merging two binomial queues

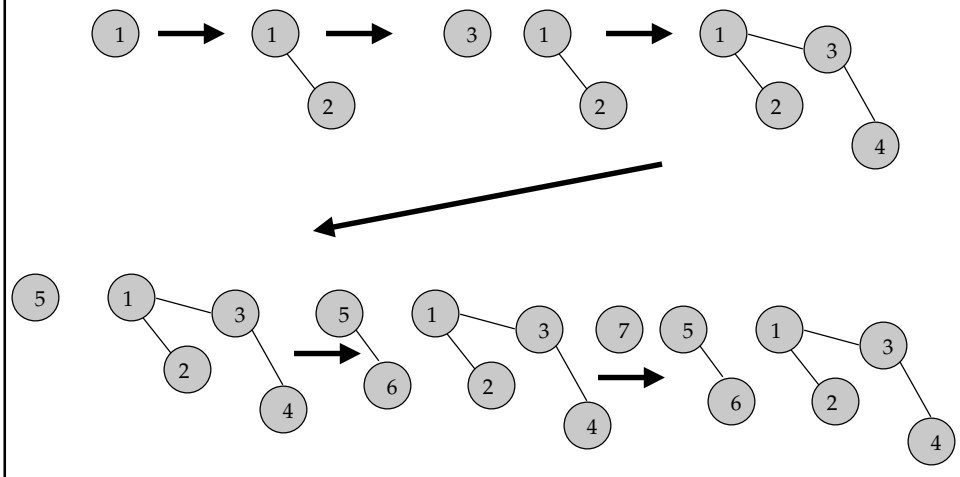
Example



Operations on binomial queues

- Insertion
  - special case of merging
  - create a one node tree and then perform the merge
  - if the priority queue into which the element is merged has the property that its smallest nonexistent binomial tree is B_i, then the time to insert is proportional to i+1
  - since each tree is present with probability 1/2, the expected time to perform an insertion is constant

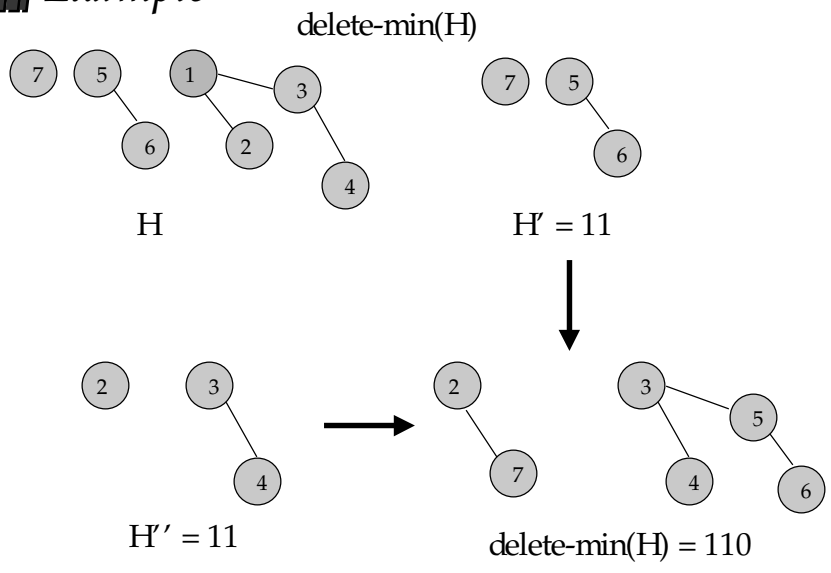
Example - inserting the keys 1-7



Operations on binomial queues

- Delete-min
  - find binomial tree, B_k, with smallest root in binomial queue H
  - Remove B_k from the forest H forming a new binomial queue H′.
  - Remove the root of B_k creating the binomial trees B'_0, B'_1, ..., B'_{k-1} which collectively form the binomial queue H′′.
  - Merge H′ with H′′ to get the answer

Example



Disjoint sets with union

- Given:
  - a fixed set T
  - partition of T into subsets S_1, S_2, ..., S_k

$$T = \bigcup_{i=1}^{k} S_i$$

- Operations to be supported are
  - Find (X) - returns the set that contains X
  - Union (R,S) - compute $R \cup S$ which replaces R and S

Disjoint sets with union - Up trees

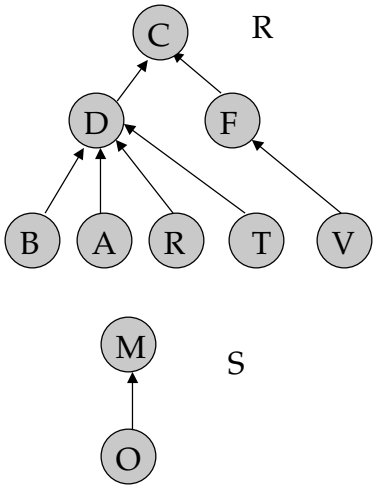
- Since the subsets are disjoint, each element belongs to only one set
- Up-tree: Node contains a single pointer to its parent in the tree representing a set S_i.
  - node can have arbitrary number of children since there is no limit on the number of pointers that can point to it.
  - sets are identified by their root nodes.

Up-trees: find

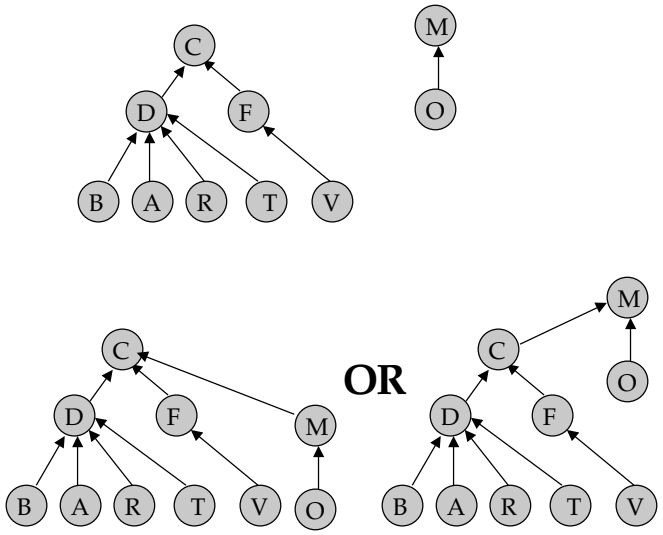
- Find(X): folow pointers from X back up to the root of the tree to which X belongs
- But how do we get to the node containing X?
  - Use some tree dictionary for the names of the set elements; in this case is takes log(n) time to find the node containing X.
  - If X is drawn from a small dictionary, then maintain a table that gives constant lookup times

Up-trees: Union

- Union(R,S) - make one set point to another
  - make the root of one point "up" to the root of the other
  - if we make the root of R point to S then we say we **merge R into S**



Merging



Merging

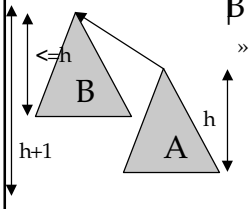
- When we merge two sets we want the height of the merged tree to be as small as possible
  - Always merge the smaller tree into the larger
  - Associate a field, Count, with the root of each tree which contains the number of nodes in the tree
- Let R be an up-tree representing a set of size n constructed from singleton sets by repeatedly forming unions using the merge smaller rule. Then the height of R is at most log n.

Merging smaller into bigger gives log height up trees

- Proof is by induction
  - We show that if R is a tree of height h created by a sequence of UNIONS, then $|R| >= 2^h$.
  - Use induction on h
- h = 0, or height(R) = 0. So, R consists of just the root, and $|R| = 1 >= 0$.
- Assume for any S, if Height(S) <= h, then $|S| >= 2^{Height(S)}$
- Consider the trees of height h+1 in the order that they are created.

Merging smaller into bigger gives log height up trees

- Let R be the first such tree ever constructed!
  - then R must have been created by merging a tree, A, of height h into a tree, B, of height at most h with $|A| <= |B|$ (or B would been merged into A)
    - » otherwise its height would be less than h+1 since if
      - $R = A \cup B$, and
      - Height(A)< h and Height(B) < h then
      - Height (R) = min(Height(A), Height(B)) + 1 < h+1
      - Notice that **neither** A nor B can have Height = h+1 because R is the **first** such tree every constructed, by assumption



  - But then $|A| >= 2^h$, so $|R| = |A| + |B| >= 2|A| >= 2^{h+1}$

Merging smaller into bigger gives log height up trees

- Suppose an R of height h+1 was created later, user one of the already constructed up trees of height h+1
  - But then by induction, that old up tree has at least 2^{h+1} nodes in it.
  - So R, which will contain this old up tree and more,
  - also has at least 2^{h+1} nodes
  - So, we are done

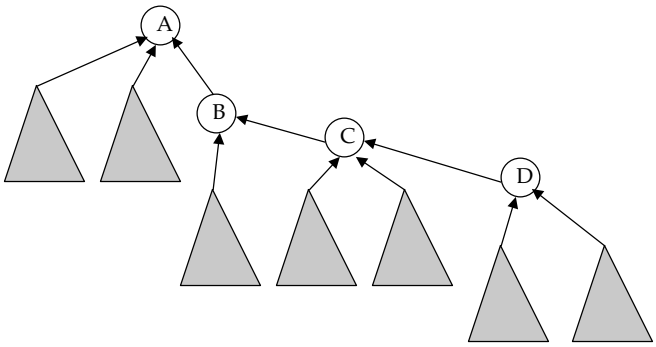
Path compression

- A simple modification to Find can make subsequent Finds faster
  - In the case where our auxiliary dictionary is a table, this will actually lower the complexity of Finds to sub-logarithmic
- Finds takes less time in a shallow tree than in a deep tree
  - trick: during a Find operation, nodes that are visited have their parent pointers updated to point to the root
  - called path compression

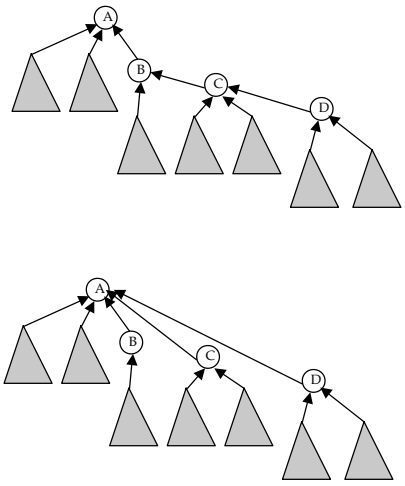
Path compression

```
function PathcompressFind (pointer P): pointer
{ return the root of the tree to which P belongs}
  R <--P
  while Parent(R) <> Nil do R <-- Parent(R)
  Q <-- P  { Now we retrace the path}
  While Q <> R do { simultaneously}
   Q <-- Parent(Q)
    Parent(Q) <-- R
  return R
```


Example



Find D

Example

- Nodes C and D which were encountered on the path to D have their pointers changed to the root.
- Subsequent Finds to them or nodes in their subtrees will be faster