CSCI 335
Software Design and Analysis III
Lecture 14: Priority Queues
Binary heaps

Professor Anita Raja

10-24-22

Announcement

- HW3 posted
- Next lecture: Staff will go over code and HW3 details.
- Office Hours this week: In-person Thursday 10/27 noon slot rescheduled to Wednesday 10/28 1-2pm zoom meeting
  - https://us02web.zoom.us/j/88365080455?pwd=RDJWZ0hoNDU3UWo5ZDIvbVcyNzBkZz09

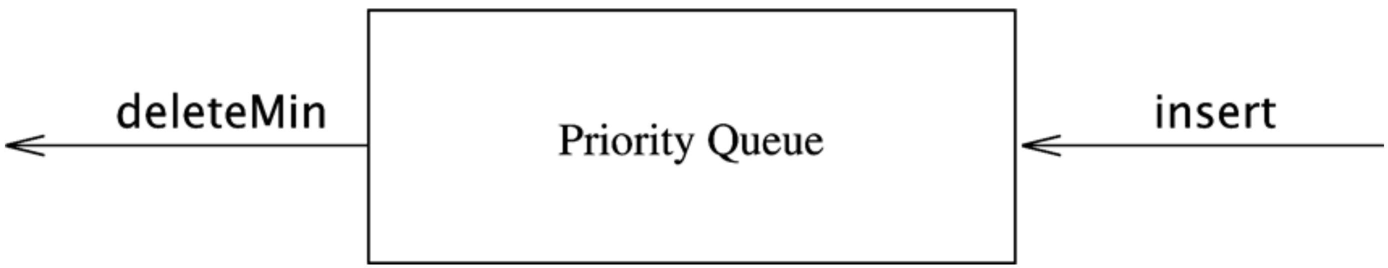
Agenda

- Hashing
  - Rehashing
  - Worst-case access
  - Extendible Hashing
- Priority Queues intro
- Binary Heap
- Heap and Hashtable
- Selection Problem
- d-heap

Priority Queues

- Queue where elements have priorities
- Efficient Implementation
- Advanced Implementations
- Uses of PQs
  - Implementation of **greedy** algorithms

Priority Queue: basic operations



Possible Implementations

- List
- Sorted List
- Binary search tree implementation
  - Appropriate for any priority queue
  - Largest item is rightmost and has at most one child

(c)

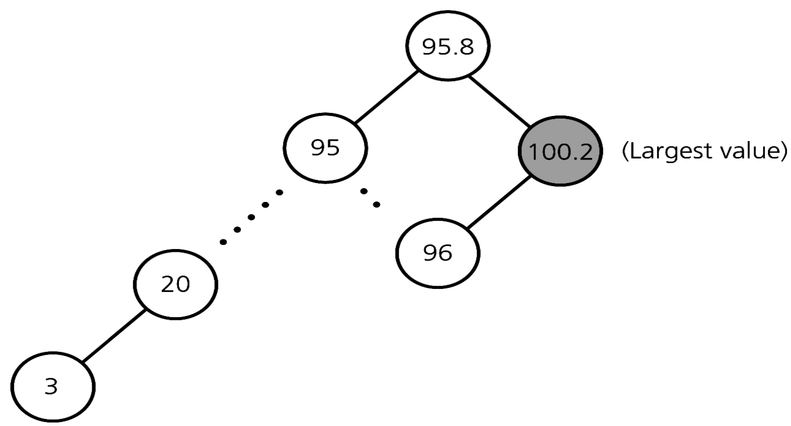


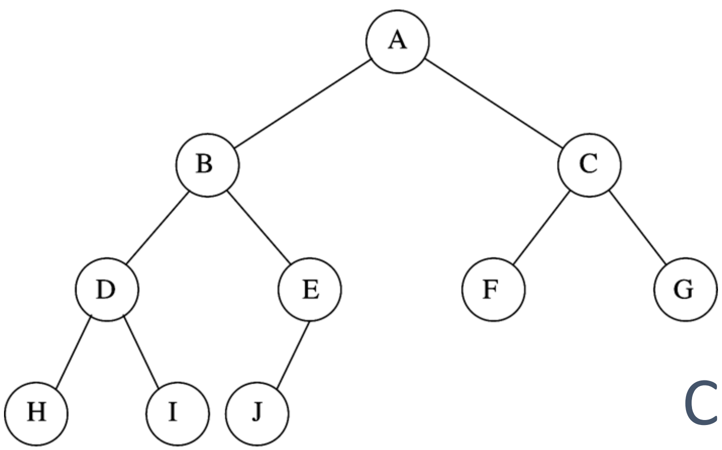
Figure 11-9c A binary search tree implementation of the ADT priority queue

First efficient implementation

- No links (pointers) required
- Very easy to implement
- O(logN) worst-case time for insert/deleteMin
- O(1) to access the min elements
- O(1) on average for insertions
- O(N) for building a queue of N items

Binary Heap: Structure property

Complete Binary Tree; Height is $\lfloor \log N \rfloor$



Left Child = 2 * Parent
Right Child = Left Child + 1

Parent = $\lfloor \text{Child} / 2 \rfloor$

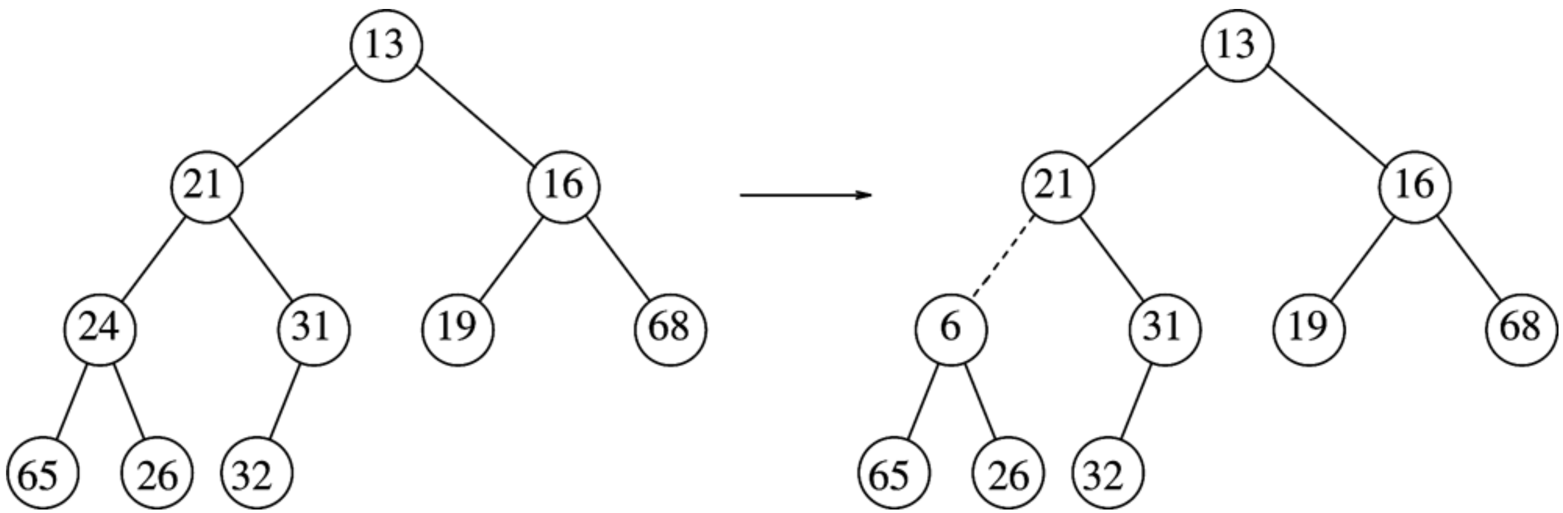
Can be implemented using an array !

	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: Heap-order property

For every node X (except root):

key(parent(X)) <= key(X)



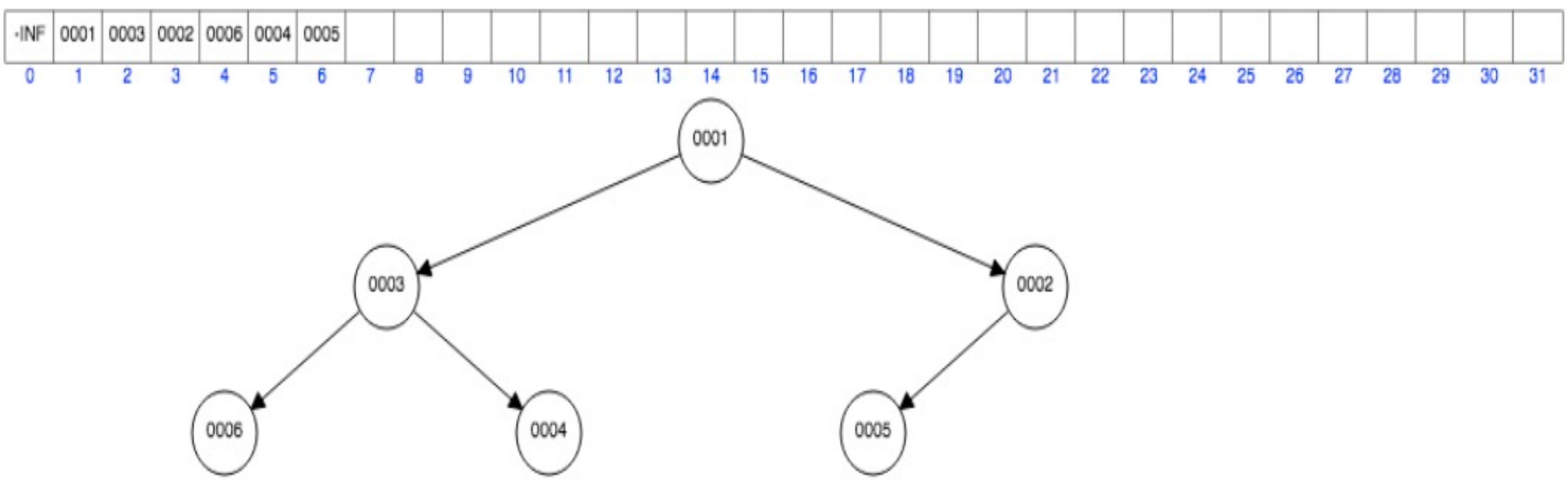
Heap

not a Heap

Basic Heap Operations

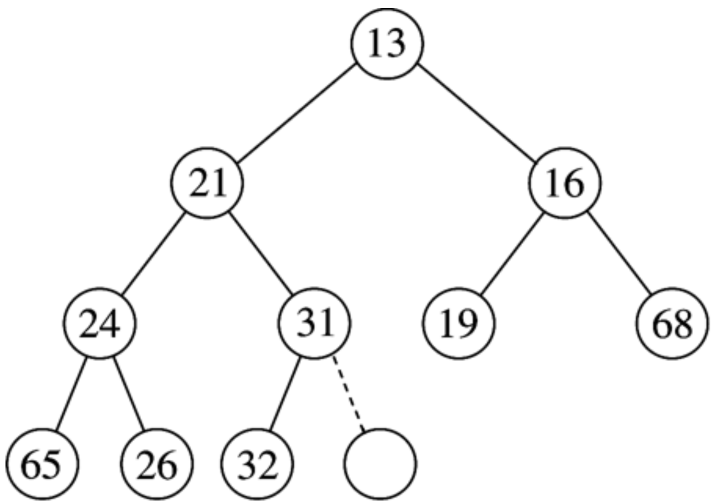
Example: Insert 6, 5, 4, 3, 2, 1 into an empty heap

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



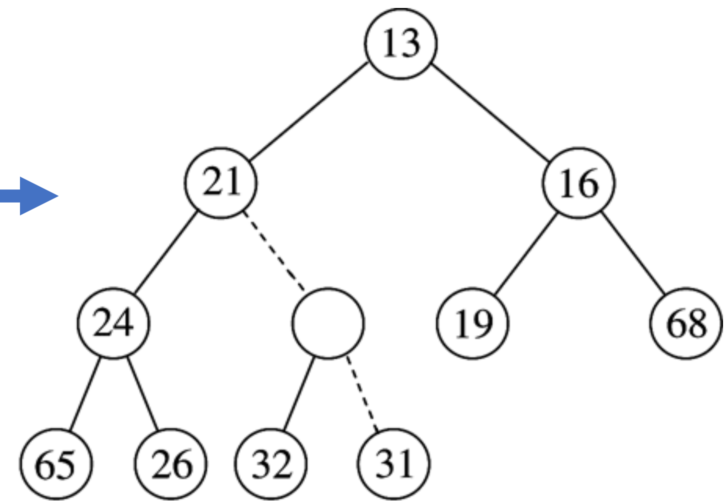
Basic Heap Operations

- Insert (percolate up)



Index: 0 1 …. … 10
Array: -, 13, 21, 16, 24, 31, 19, 68, 65, 26, 32

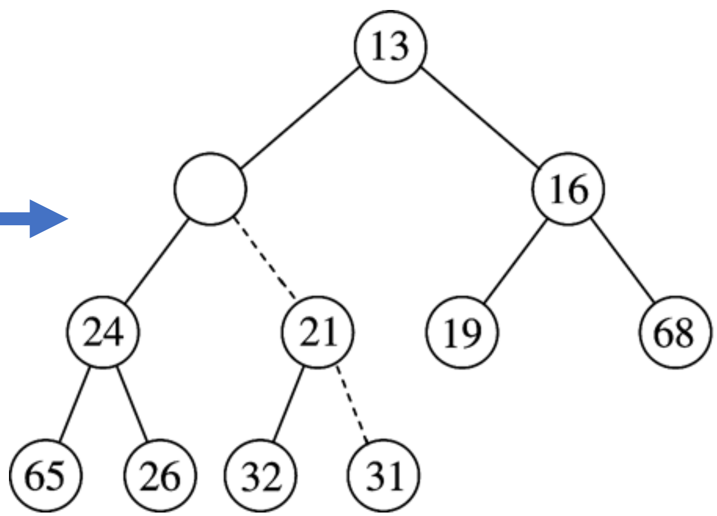
Insert 14



0 1 …. 5 … 10 11
-, 13, 21, 16, 24, H, 19, 68, 65, 26, 32, 31

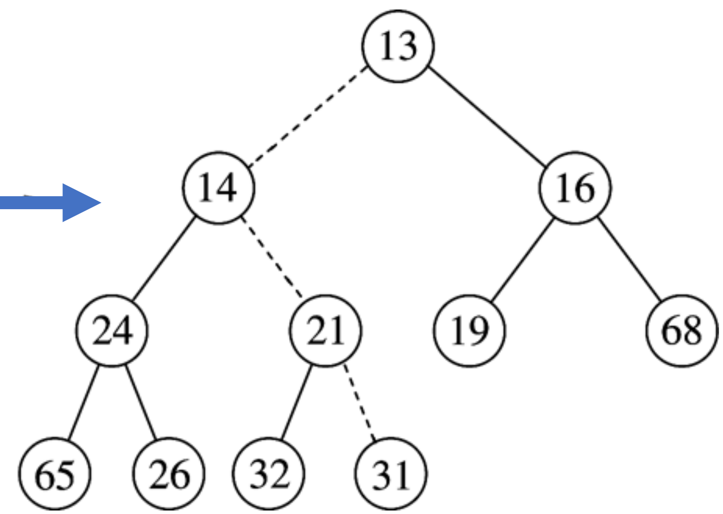
Basic Heap Operations

- Insert (percolate up)



0 1   2   5   ... 10 11
-, 13, H, 16, 24, 21, 19, 68, 65, 26, 32, 31

Insert 14



0 1   2   5   ... 10 11
-, 13, 14, 16, 24, 21, 19, 68, 65, 26, 32, 31

Basic Heap Operations

- Insert: O(logN)
  - Worst-case: key to be inserted is the new minimum

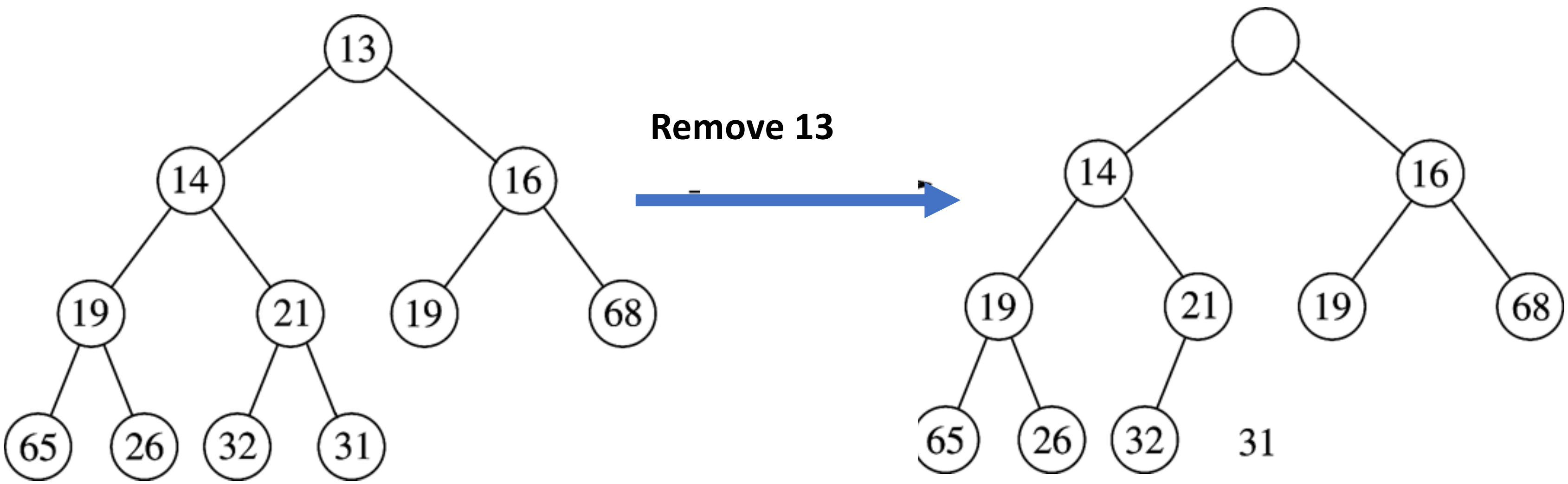
    => will be percolated up all the way to the root.

On average 2.607 comparisons are required…

```
// @x: item to be inserted into binary heap.
// Heaps starts from index 1. Location 0 is unused.
void Insert(const Comparable &x) {
     if (current_size_ == array_.size() - 1)  // Heap full.
        array_.resize(array_.size() * 2);
     int hole = current_size_++;
     Comparable copy = x;
     // Save item to dead space of heap.
     // Used also as marker to stop the loop.
     array_[0] = std::move(copy);
     for ( ; x < array_[hole / 2]; hole /= 2)
       array_[hole] = std::move(array_[hole / 2]);
     array_[hole] = std::move(array_[0]);
}
```

Basic Heap Operations

- DeleteMin: Percolate down

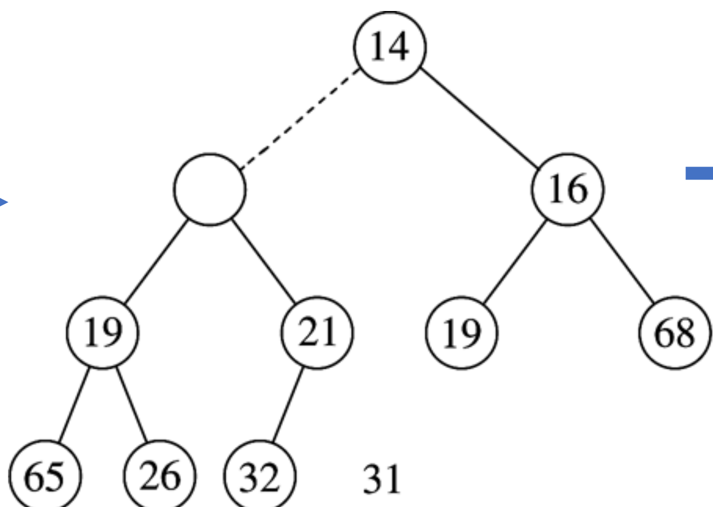


0 1 2 5 … 10 11
-, 13, 14, 16, 19, 21, 19, 68, 65, 26, 32, 31

0 1 2 5 10
-, 31, 14, 16, 19, 21, 19, 68, 65, 26, 32

Basic Heap Operations

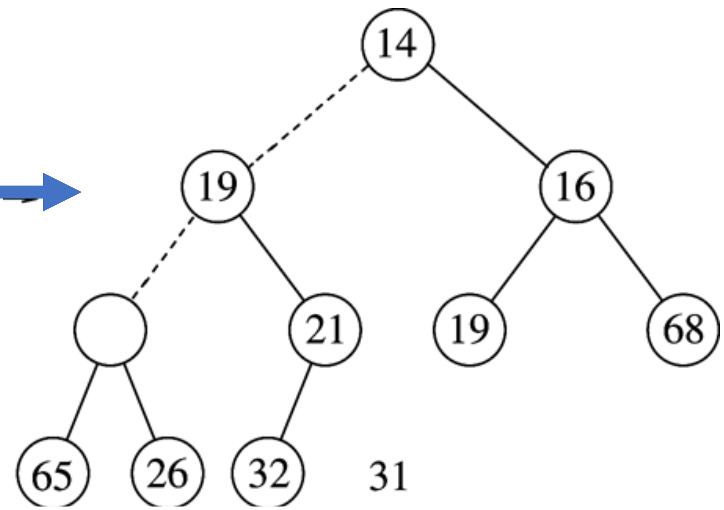
- DeleteMin: Percolate down



tmp = 31

0 1 2 3 5 10
-, 14, 14, 16, 19, 21, 19, 68, 65, 26, 32

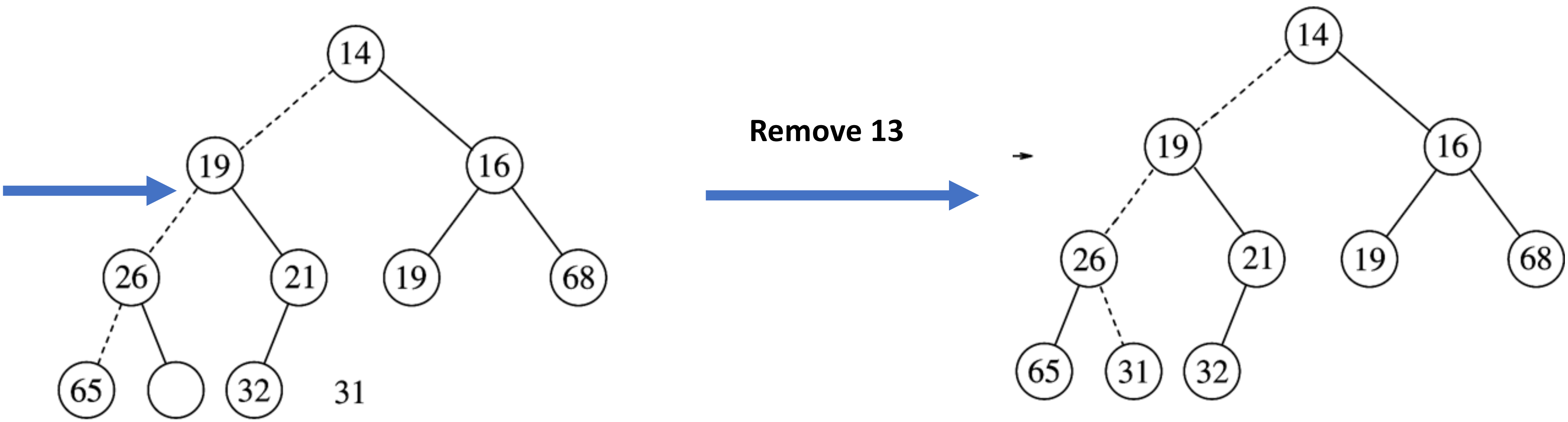
Remove 13



0 1 2 3 4 5 8 9 10
-, 14, 19, 16, 19, 21, 19, 68, 65, 26, 32

Basic Heap Operations

- DeleteMin: Percolate down



0 1 2 3 4 5 8 9 10
-, 14, 19, 16, 26, 21, 19, 68, 65, *26*, 32

0 1 2 3 4 5 8 9 10
-, 14, 19, 16, 26, 21, 19, 68, 65, 31, 32

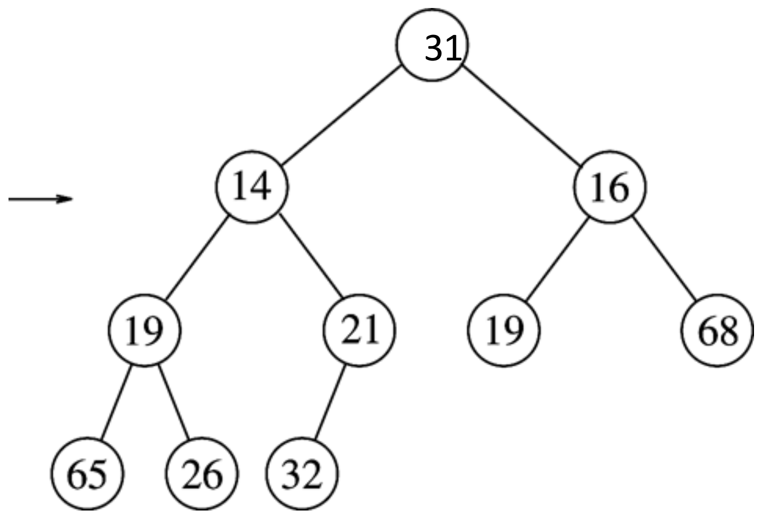
Basic Heap Operations

- DeleteMin: O(logN) worst- and average-case

```
// @param hole: index of element on array_
// Percolates down element stored in array_[hole].
void PercolateDown(int hole) {
     Comparable tmp = std::move(array_[hole]);
     int child;
     for(; hole * 2 <= current_size_; hole = child) {
     child = hole * 2;  // Index of left child.
        if (child != current_size_ &&
            array_[child + 1] < array_[child]) ++child;
     // child is the index of the minimum of the two children.
        if (array_[child] < tmp)
            array_[hole] = std::move(array_[child]);
        else
            break;  // Stop percolating down.
      }  // End for
      array_[hole] = std::move(tmp);
}
----

For DeleteMin() need to call PercolateDown(1)
```

Run PercolateDown code

- PercolateDown(1)



0 1 2 5 10
-, 31, 14, 16, 19, 21, 19, 68, 65, 26, 32

Other Operations

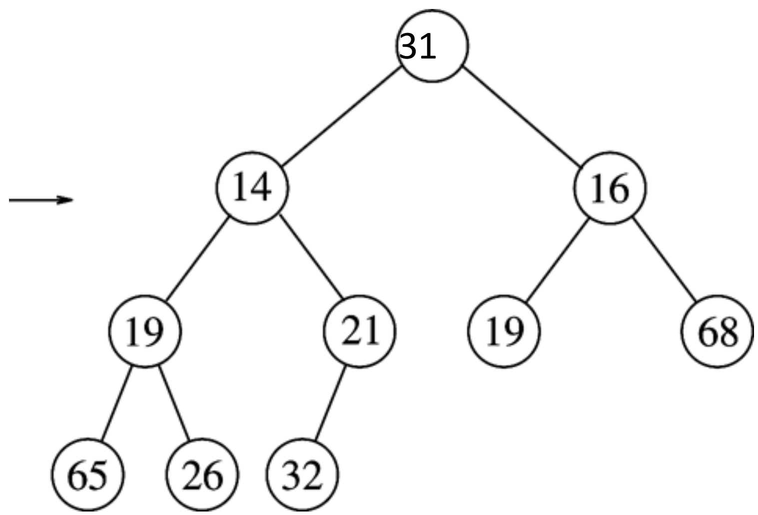
- decreaseKey(p, Δ):
  - lower key at position p by positive Δ
  - Might violate heap order, so use percolate up.
  - Useful for sysadmin to make their programs run with highest priority
- increaseKey(p, Δ): increase key at position p by positive Δ
  - Increase key at position p by positive Δ
  - Done with percolate down
  - Many schedulers automatically drop the priority of process that consumes excessive CPU time
- remove(p):

  decreaseKey(p, ∞)

  deleteMin

Question: How to find position p of key?

Heap and hash table



0 1 2 5 10

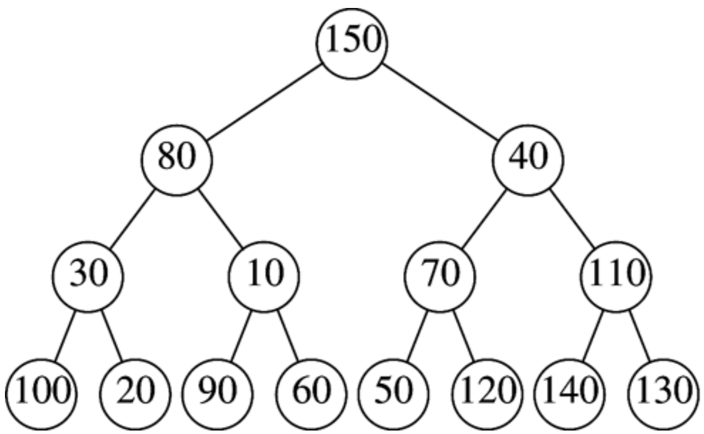
-, 31, 14, 16, 19, 21, 19, 68, 65, 26, 32

buildHeap from array

buildHeap

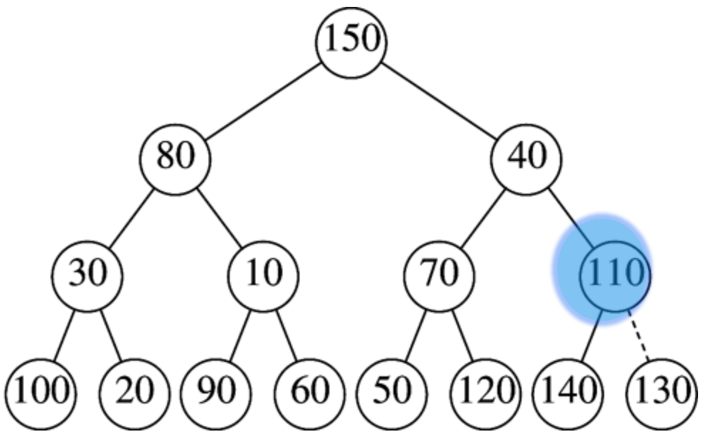
```
explicit BinaryHeap(const vector<Comparable> &items)
  : array_(items.size() + 10), current_size_{items.size()} {
    for (int i = 0; i < items.size(); i++ )
       array_[i + 1] = items[i];  // Create initial array
    BuildHeap();  // Make it a heap.
}

void BuildHeap() {
     for (int i = current_size_ / 2; i > 0; --i)
      PercolateDown(i);
}
```

buildHeap from array



Initial array :
| | 150 | 80 | 40 | 30 | 10 | 70 | 110 | 100 | 20 | 90 | 60 | 50 | 120 | 140 | 130 |
current_size_ = 15

Build Heap from array



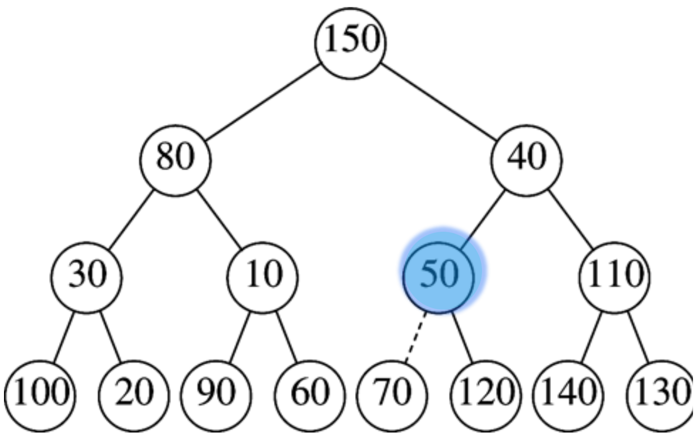
PercolateDown(7)

Initial array :

| | 150 | 80 | 40 | 30 | 10 | 70 | 110 | 100 | 20 | 90 | 60 | 50 | 120 | 140 | 130 |

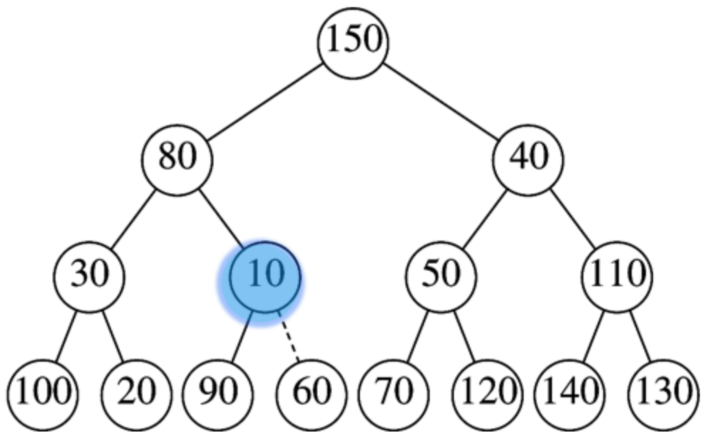
current_size_ = 15 (=> current_size_ / 2 = 7)

Build Heap from array



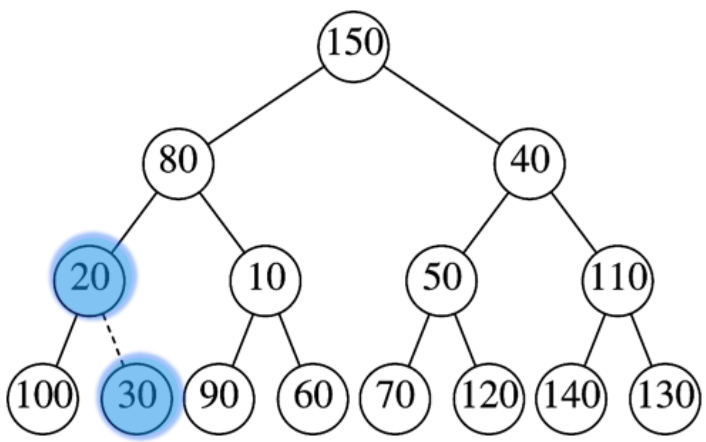
PercolateDown(6)

Build Heap from array

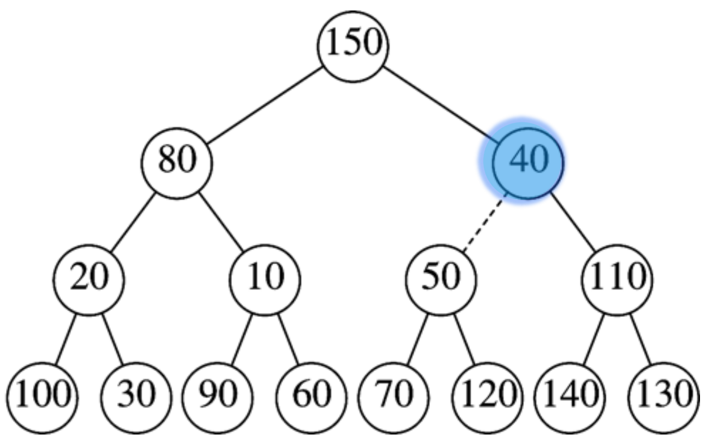


PercolateDown(5)

Build Heap from array

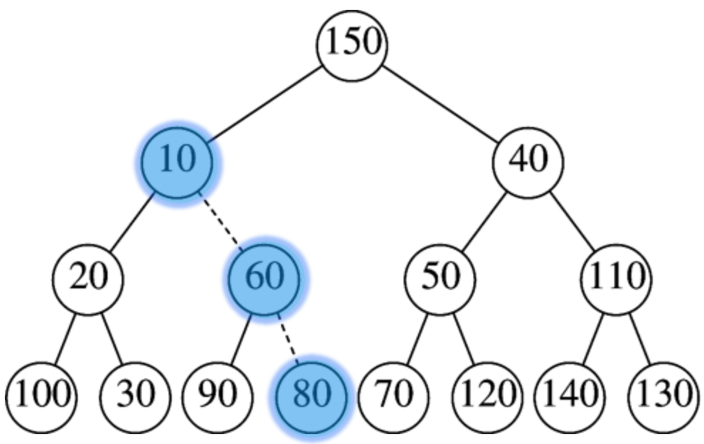


PercolateDown(4)

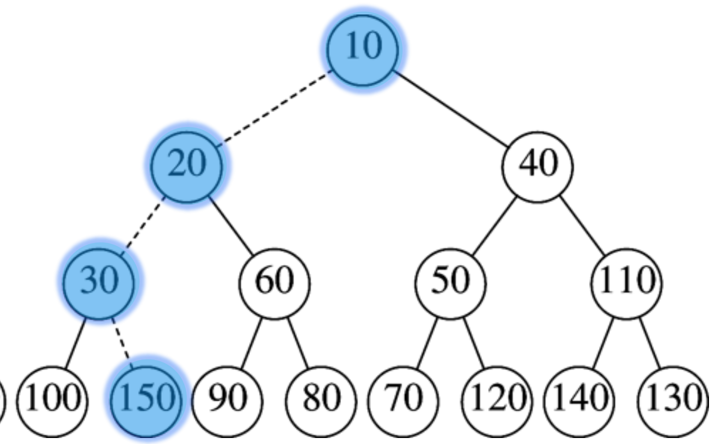


PercolateDown(3)

Build Heap from array



PercolateDown(2)



PercolateDown(1)

buildHeap: running time?

- Each dash line corresponds to 2 comparisons
  - => 20 comparisons in previous example
- In worst-case: find total number of dash lines

  i.e. find the sum of heights of all nodes

buildHeap: running time?

Theorem:

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

Proof: … (next slide)

Running Time: A complete tree has between 2^h and 2^{h+1} nodes

=> **buildHeap is O(N)** where N is number of nodes

Full binary tree of height h (in class)

buildHeap running time (in class)

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

Proof:

buildHeap running time (in class)

- **buildHeap is O(N) worst case**
- **Complete binary tree has between**

buildHeap Running Time

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

There is 1 node at height h, 2 nodes at height $h - 1$, ..., 2^i at $h - i$.

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

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

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

$$S = -h + 2^{h+1} - 1 - 1$$

$$S = \left(2^{h+1} - 1\right) - (h+1)$$

buildHeap Running Time

- buildHeap is $O(N)$ worst-case
- Complete binary tree has between 2^h and 2^{h+1} nodes
- $S = \left(2^{h+1} - 1\right) - (h + 1) = O(N) - O(\log N) = O(N)$

The Selection Problem

Given a list of N elements, and integer k

Find the k^{th} largest (or smallest) element.

Ideas?

Try: 4^{th} smallest or 4^{th} largest

2, 1, 3, 5, 10, 4, 7, 0

How will you use heaps to come up with O(NlogN) algorithm?

The Selection Problem

Ideas:

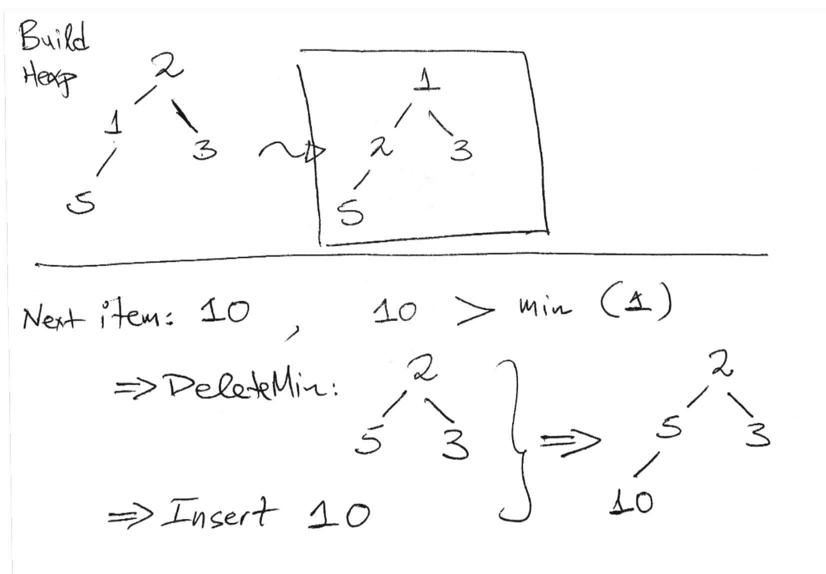
1. Sort N elements and find index k.
2. Build a heap of N elements with revised heap property where largest is root.
3. Do k deleteMax (similar to deleting root for deleteMin).
4. Last element extracted from heap is the answer.

k DeleteMax

- Build heap of n items O(n)
- k DeleteMaxs O(k logn)
- Total: O(n + k logn)

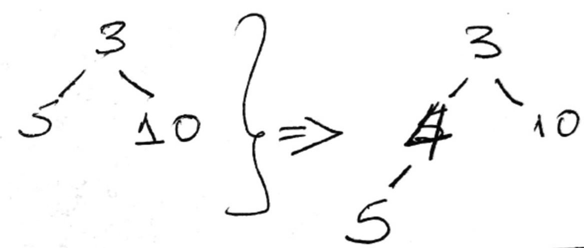
Example: 4th largest in : 2, 1, 3, 5, 10, 4, 7, 0
1st DeleteMax -> 10
2nd -> 7
3rd -> 5
4th -> 4

Alternative (4th largest)



Next item: 4 > min(2)

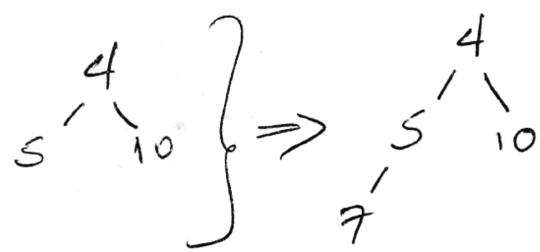
⇒ DeleteMin:



⇒ Insert 4

Next item: 7 > min(3)

⇒ Deletemin:

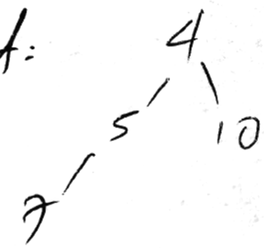


⇒ Insert 7

Next item: $0 < \min(4)$

$\Rightarrow$ Do nothing.

Final result:



$\Rightarrow$ 4^{th} largest item is 4.

Online algorithm: Answer (i.e. 4^{th} largest) as items are being read.
Complexity?

The Selection Problem

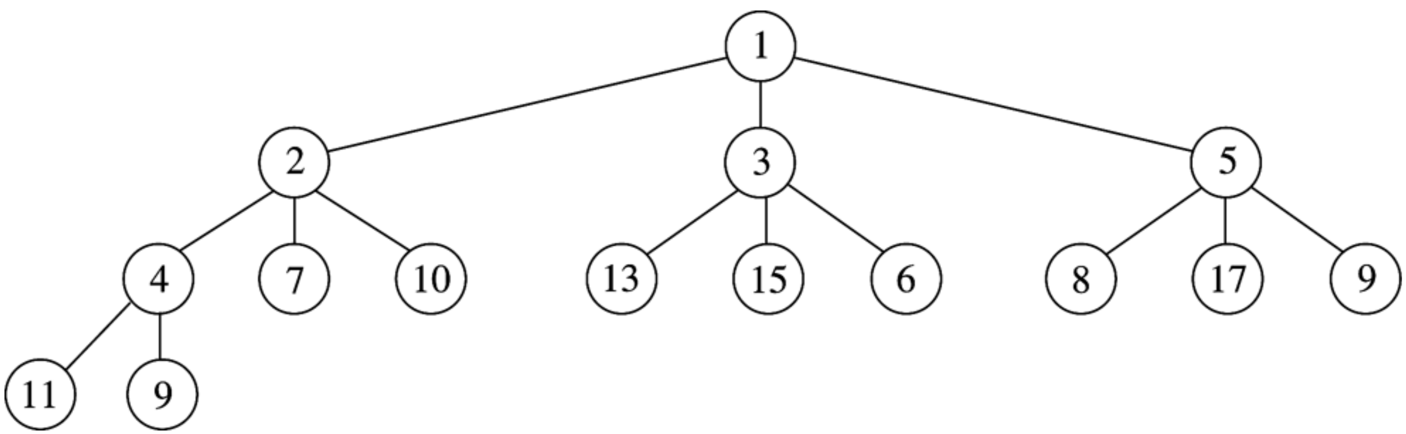
- Build heap of k items O(k)
- N-k items
  - test item goes in sequence O(1);
  - DeleteMins O(logn)
  - Insert if necessary O(logN)
  - Total: O((N-k)logN)
- Total: O(k+(N-k)logN) = O(Nlogk)
- Also gives bound of **Θ**(NlogN) for finding the median.
- Chapter 7 sorting algorithm in O(N) average time
- Chapter 10 elegant, albeit impractical algorithm to solve O(N) worst-case time.

d-Heaps

d children

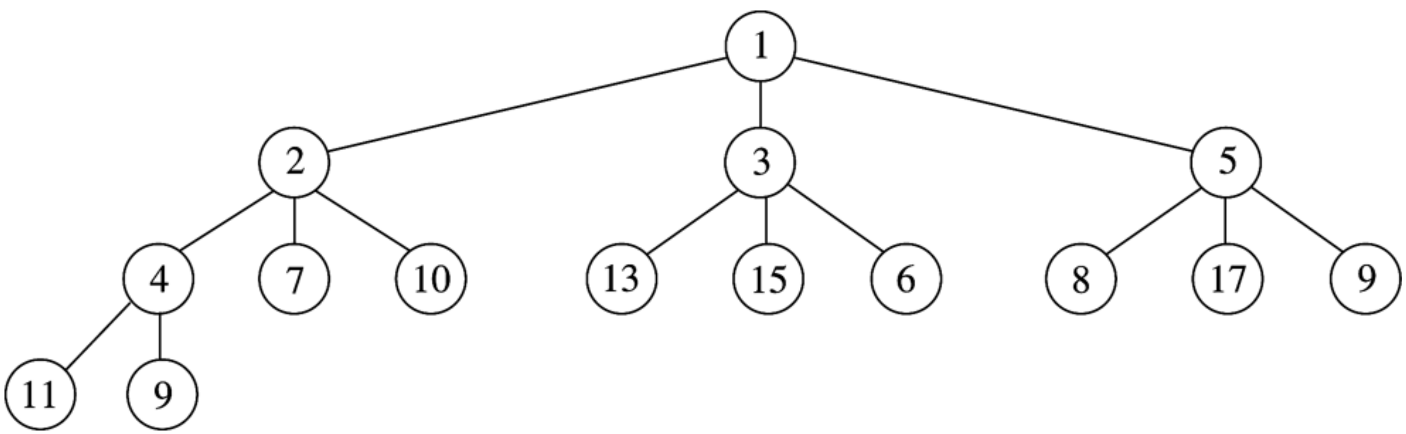
Height is now O($\log_d$ N)=>less deep

Is it better than a 2-heap?



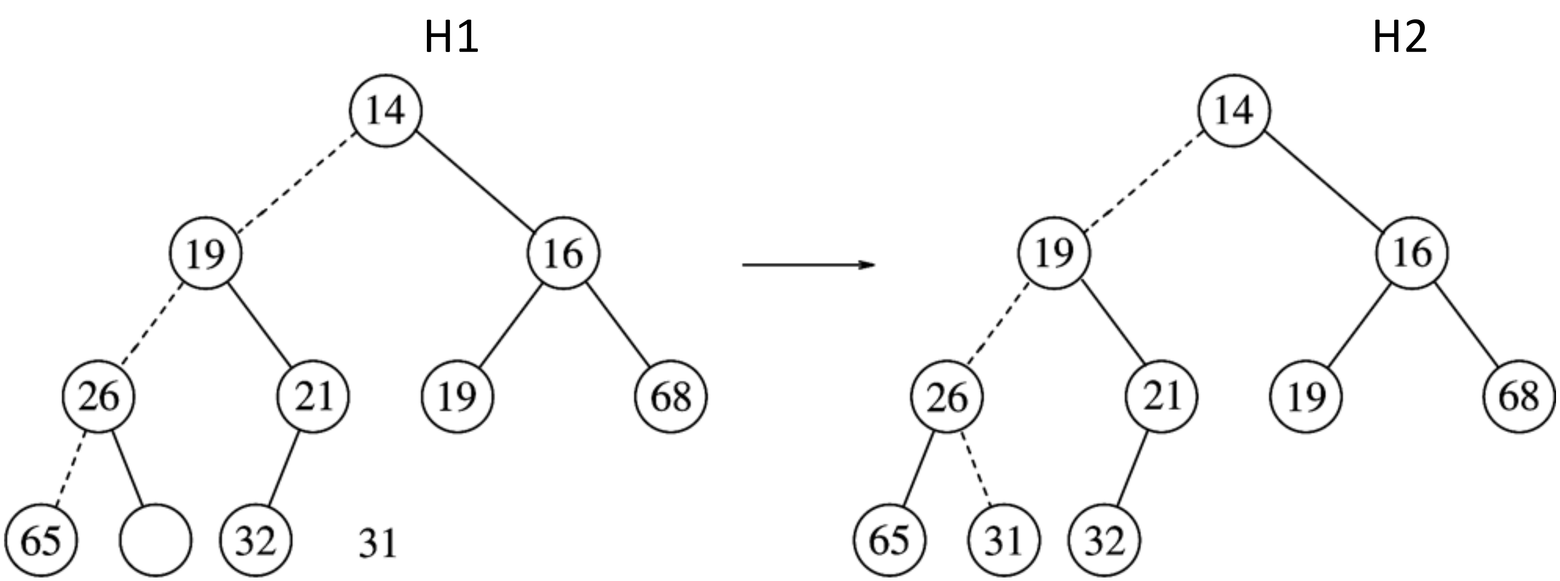
d-Heaps

- d children
- Height is now O($\log_d$ N)=>less deep
- Is it better than a 2-heap?
  - Good for external (disk) implementation
  - Good when more inserts than deleteMins
  - In general 4-heaps outperform 2-heaps



The Merge operation

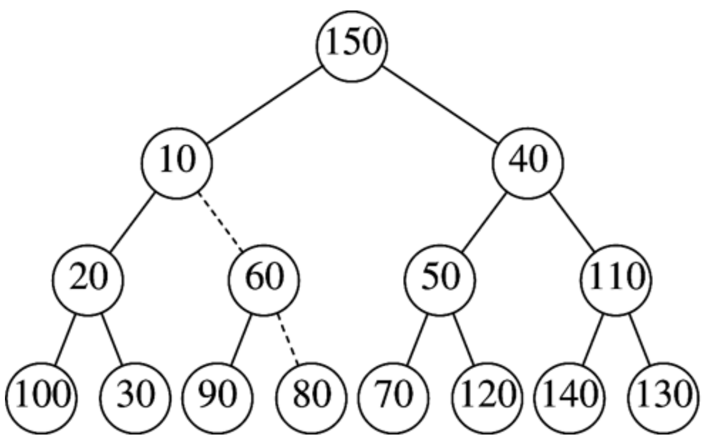
- Given two heaps H1 and H2, merge them into heap H
- In a binary heap, what is the complexity of merge?
- Need efficient merge



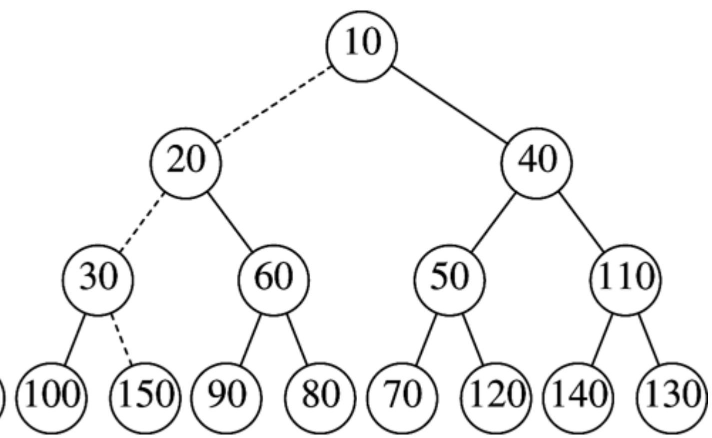
The Merge operation

- Given two heaps H1 and H2, merge them into heap H
- In a binary heap, what is the complexity of merge?
- Need efficient merge

H1



H2



Next class

- d-heap
- Skew heaps