Heaps & Priority Queues

Fall 2020

School of Software Engineering

South China University of Technology

Content

- Binary Heaps
- Hashing

Binary Heaps

Readings

- Reading
  - Sections 6.1-6.4

Revisiting FindMin

- Application: Find the smallest (or highest priority) item quickly
  - Operating system needs to schedule jobs according to priority instead of FIFO
  - Event simulation (bank customers arriving and departing, ordered according to when the event happened)
  - Find student with highest grade, employee with highest salary etc.

Priority Queue ADT

- Priority Queue can efficiently do:
  - FindMin (and DeleteMin)
  - Insert
- What if we use…
  - Lists: If sorted, what is the run time for Insert and FindMin? Unsorted?
  - Binary Search Trees: What is the run time for Insert and FindMin?
  - Hash Tables: What is the run time for Insert and FindMin?

Less flexibility → More speed

- Lists
  - If sorted: FindMin is O(1) but Insert is O(N)
  - If not sorted: Insert is O(1) but FindMin is O(N)
- Balanced Binary Search Trees (BSTs)
  - Insert is O(log N) and FindMin is O(log N)
- Hash Tables
  - Insert O(1) but no hope for FindMin
- BSTs look good but…
  - BSTs are efficient for all Finds, not just FindMin
  - We only need FindMin

Better than a speeding BST

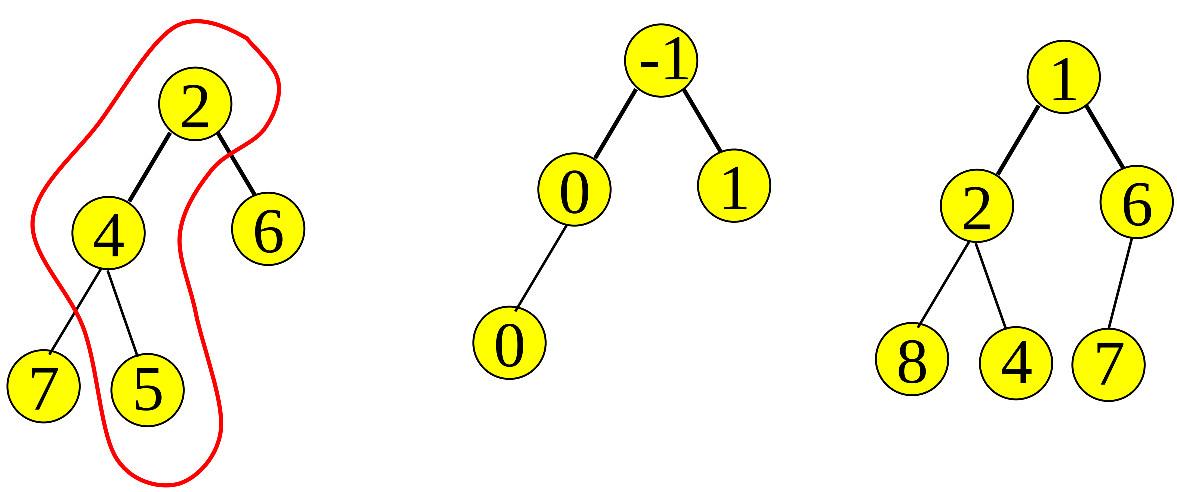
- We can do better than Balanced Binary Search Trees?
  - Very limited requirements: Insert, FindMin, DeleteMin.
  - The goals are: FindMin is O(1)
  - Insert is O(log N)
  - DeleteMin is O(log N)

Binary Heaps

- A binary heap is a binary tree (NOT a BST) that is:
  - Complete: the tree is completely filled except possibly the bottom level, which is filled from left to right
  - Satisfies the heap order property
    - every node is less than or equal to its children
    - or every node is greater than or equal to its children
- The root node is always the smallest node
  - or the largest, depending on the heap order

Heap order property

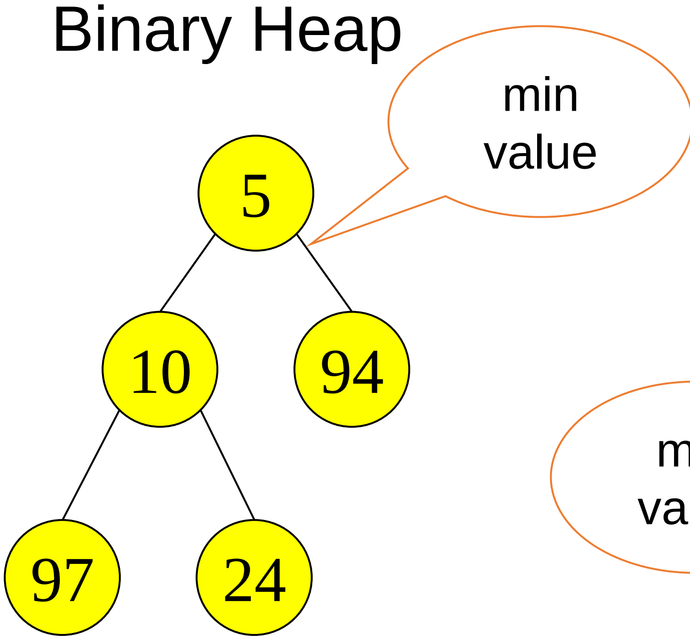
- A heap provides limited ordering information
- Each *path* is sorted, but the subtrees are not sorted relative to each other
  - A binary heap is NOT a binary search tree



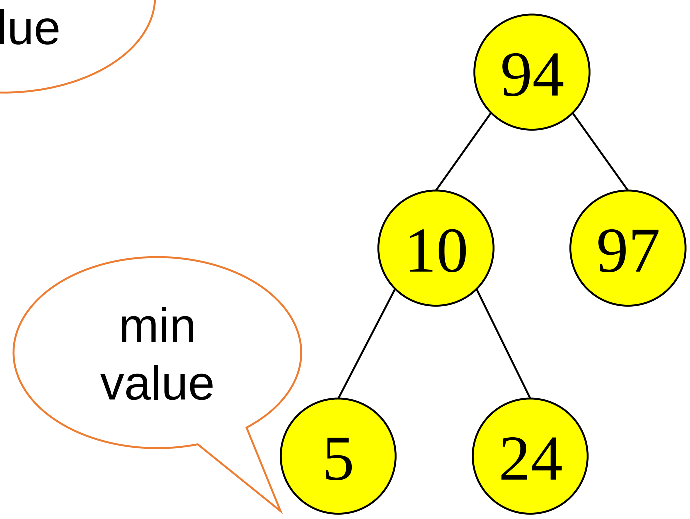
These are all valid binary heaps (minimum)

Binary Heap vs Binary Search Tree

Binary Heap



Binary Search Tree

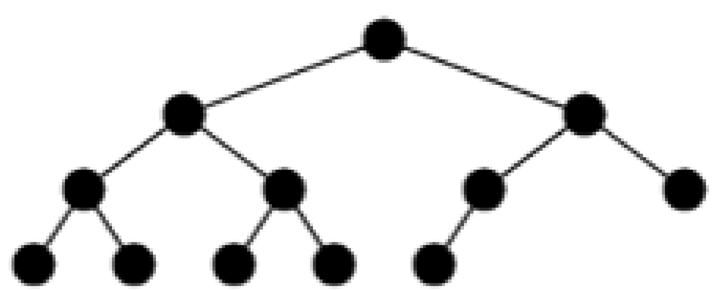


Parent is less than both left and right children

Parent is greater than left child, less than right child

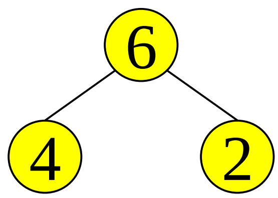
Structure property

- A binary heap is a complete tree
  - All nodes are in use except for possibly the right end of the bottom row

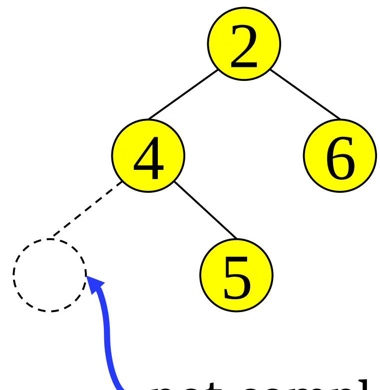


A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes.
The height of a complete binary tree is $\lfloor \log N \rfloor$

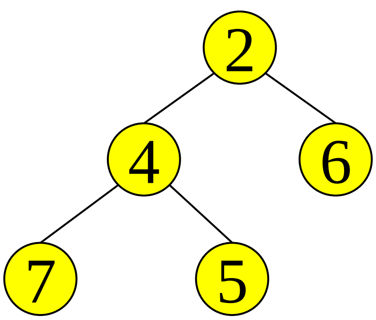
Examples



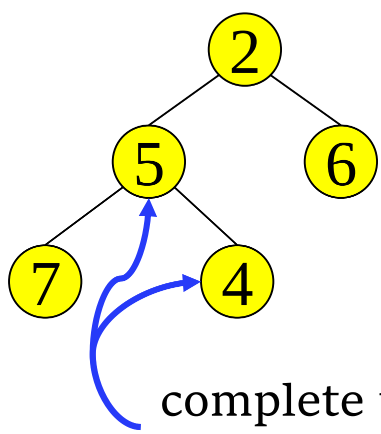
complete tree,
heap order is "max"



not complete



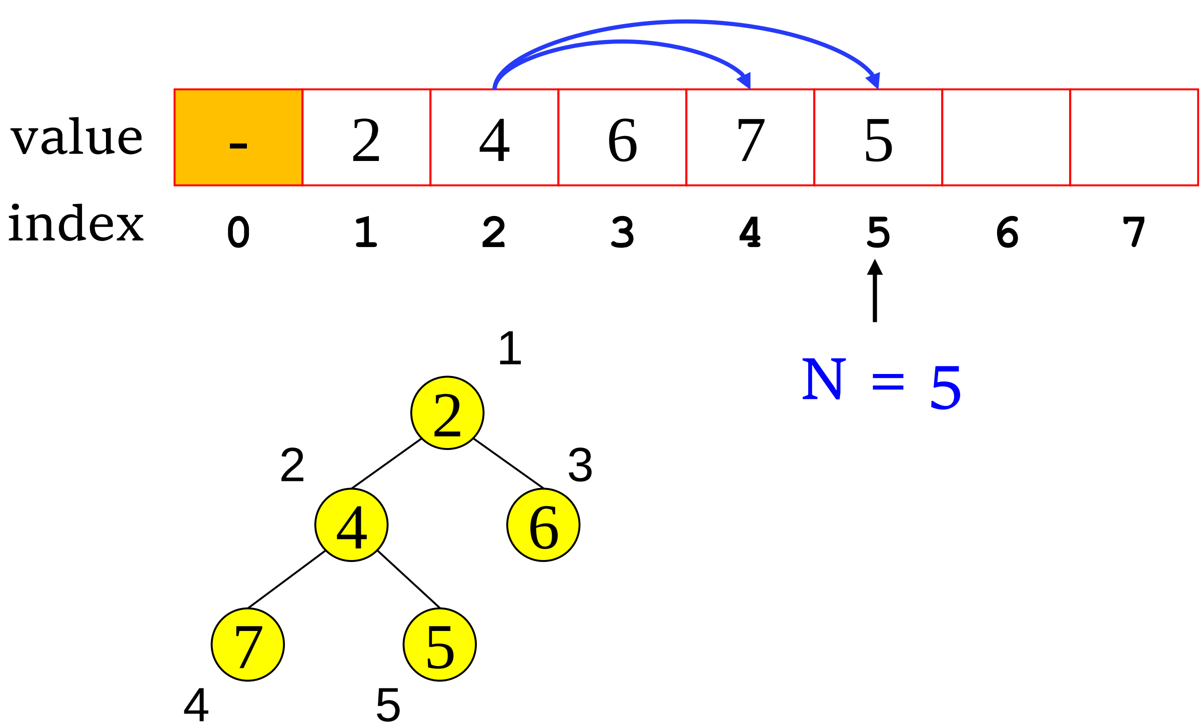
complete tree,
heap order is "min"



complete tree, but min
heap order is broken

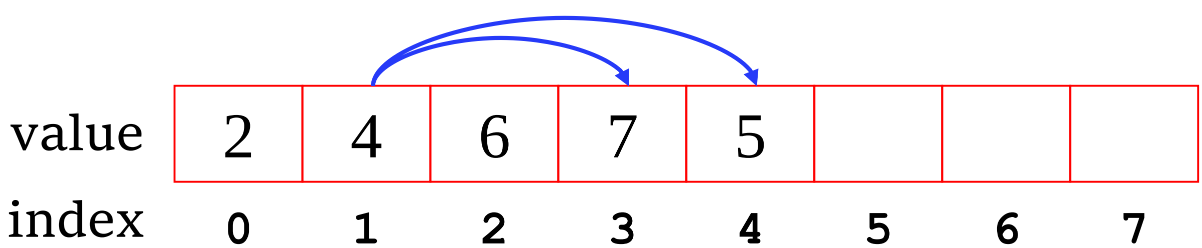
Array Implementation of Heaps (Implicit Pointers)

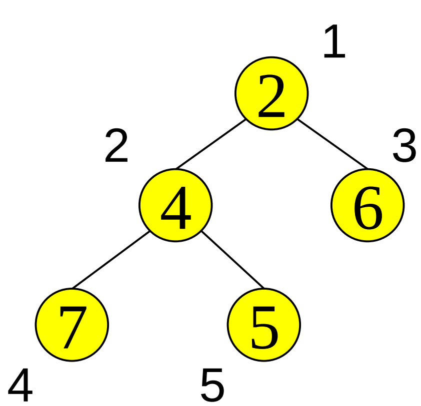
- Calculate the array indices of the various relatives of a node
  - Root node = A[1]
  - Children of A[i] = A[2i], A[2i + 1]
  - Parent of A[j] = A[j/2]
- Keep track of current size N (number of nodes)



Array Implementation of Heaps

- Another calculation
  - Root node = A[0]
  - Parent (r) = $\lfloor (r - 1) / 2 \rfloor$ if $r \neq 0$ and $r < n$
  - Left child(r) = $2r + 1$ if $2r+1 < n$
  - Right child(r) = $2r + 2$ if $2r + 2 < n$
  - Left sibling(r) = $r - 1$ if r is even, $r > 0$ and $r < n$.
  - Right sibling(r) = $r + 1$ if r is odd, $r + 1 < n$



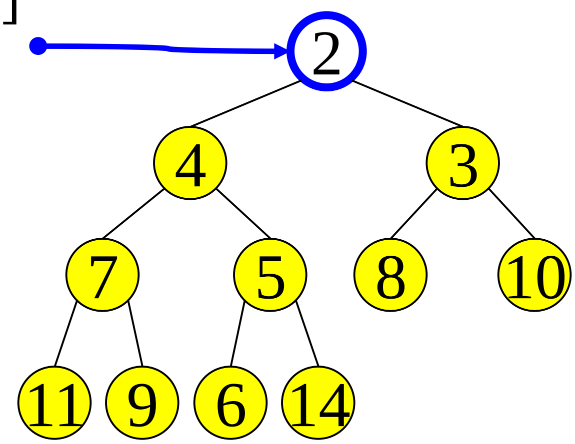


Array Implementation of Heaps

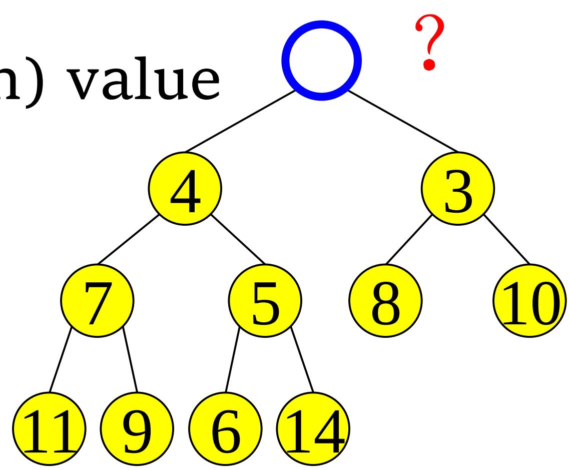
```
template <typename Comparable>
class BinaryHeap {
public:
  explicit BinaryHeap( int capacity = 100 );
  explicit BinaryHeap( const vector<Comparable> & items );

  const Comparable & findMin( ) const;
  void insert( const Comparable & x );
  void insert( Comparable && x );
  void deleteMin( );
  void deleteMin( Comparable & minItem );

private:
   int currentSize; // Number of elements in heap
   vector<Comparable> array; // The heap array

  void buildHeap( );
  void percolateDown( int hole );
};
```

FindMin and DeleteMin

- FindMin: Easy!
  - Return root value A[1]
  - Run time = ?

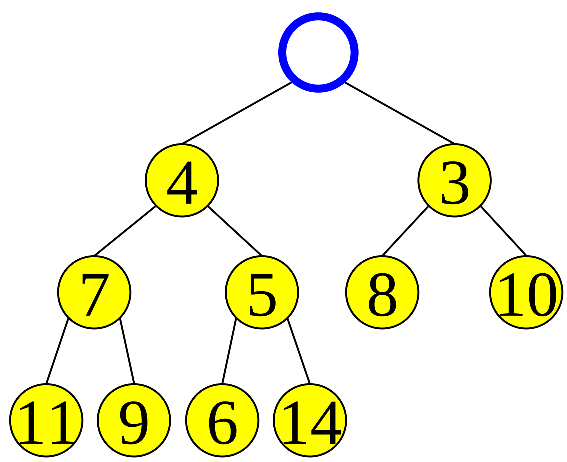


- DeleteMin:
  - Delete (and return) value at root node



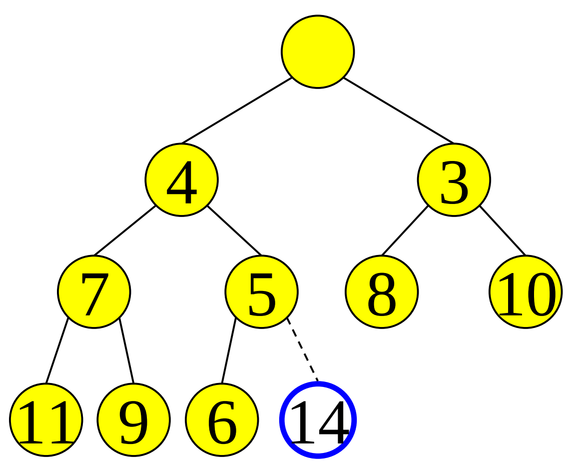
Maintain the Structure Property

- We now have a “Hole” at the root
  - Need to fill the hole with another value



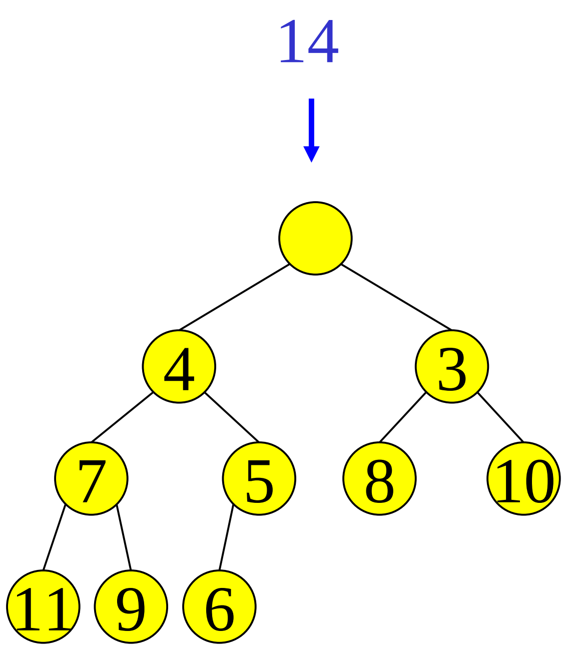
Maintain the Structure Property

- When we get done, the tree will have one less node and must still be complete



Maintain the Heap Property

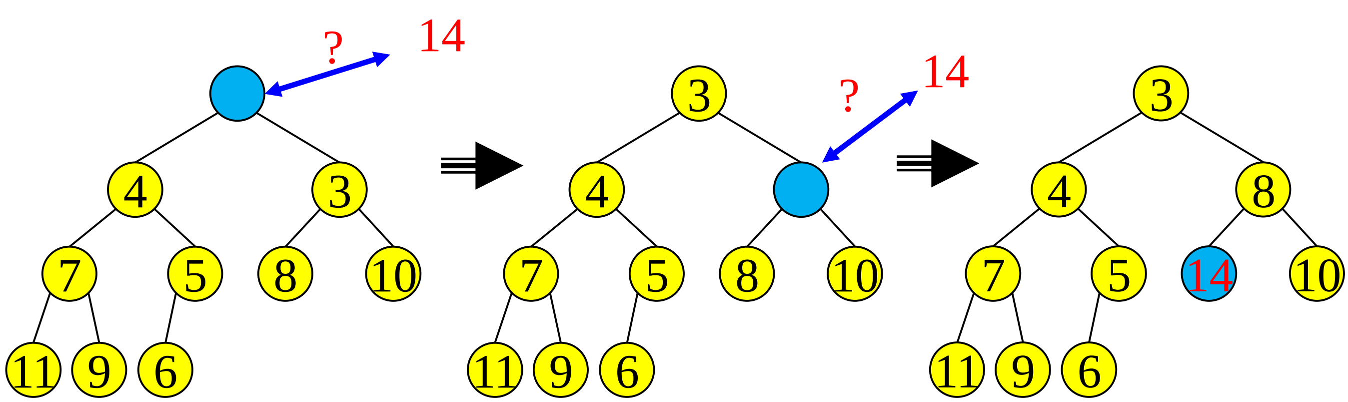
- The last value has lost its node
  - we need to find a new place for it
- We can do a simple insertion sort operation to find the correct place for it in the tree



Maintain the Heap Property

```
/**
* Remove the minimum item and place it in minItem.
* Throws UnderflowException if empty.
*/
void deleteMin( Comparable & minItem ) {
 if( isEmpty( ) )
   throw UnderflowException{ };

   minItem = std::move( array[ 1 ] );
   array[ 1 ] = std::move( array[ currentSize-- ] );

  percolateDown( 1 );
}
```

DeleteMin: Percolate Down



- Keep comparing with children A[2i] and A[2i + 1]
- Copy smaller child up and go down one level
- Done if both children are ≥ item or reached a leaf node
- What is the run time?

Percolate Down

1	2	3	4	5	6
~~6~~	10	8	13	14	25

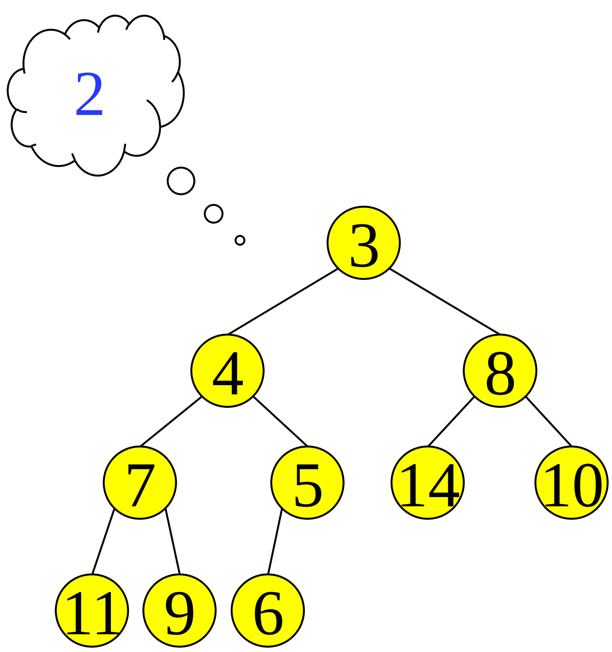
```
/**
* Internal method to percolate down in the heap.
* hole is the index at which the percolate begins.
*/
void percolateDown( int hole ) {
  int child;
  Comparable tmp = std::move( array[ hole ] );
  for( ; hole * 2 <= currentSize; hole = child ) {
    child = hole * 2; //child = leftchild
    if( child !=currentSize && array[ child + 1 ]<array[ child ] )
      ++child; //child = rightchild
    if( array[ child ] < tmp )
      array[ hole ] = std::move( array[ child ] ); // go down
    else
      break;
  }
  array[ hole ] = std::move( tmp );
}
```

DeleteMin: Run Time Analysis

- Run time is O(depth of heap)
- A heap is a complete binary tree
- Depth of a complete binary tree of N nodes?
  - depth = $\lfloor \log_2(N) \rfloor$
- Run time of DeleteMin is O(log N)

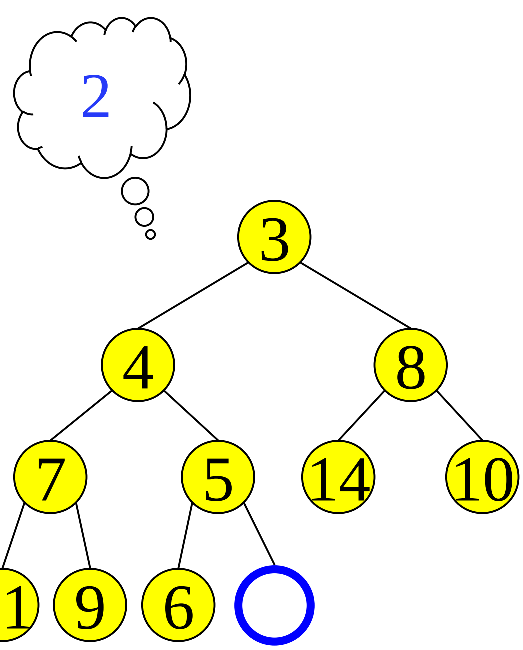
Insert

- Add a value to the tree
- Structure and heap order properties must still be correct when we are done



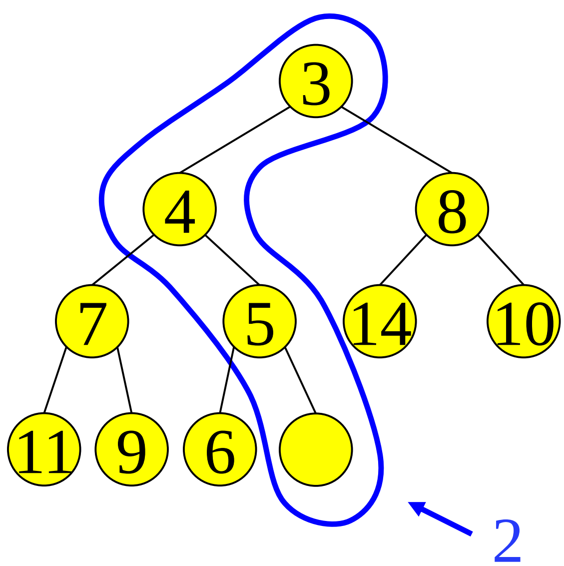
Maintain the Structure Property

- The only valid place for a new node in a complete tree is at the end of the array
- We need to decide on the correct value for the new node, and adjust the heap accordingly

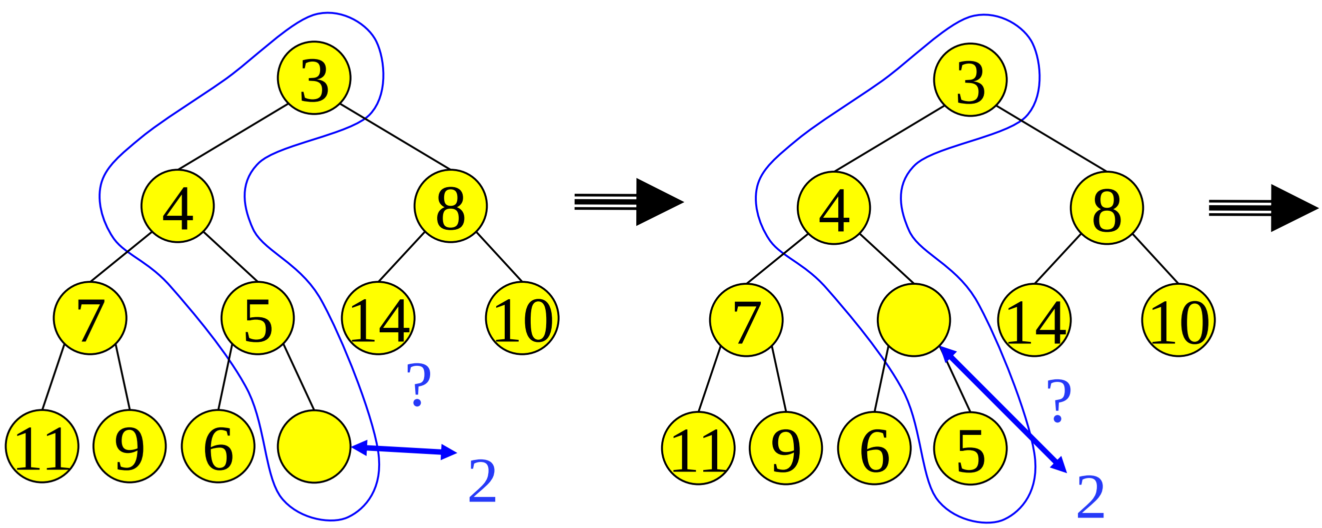


Maintain the Heap Property

- The new value goes where?
- We can do a simple insertion sort operation to find the correct place for it in the tree

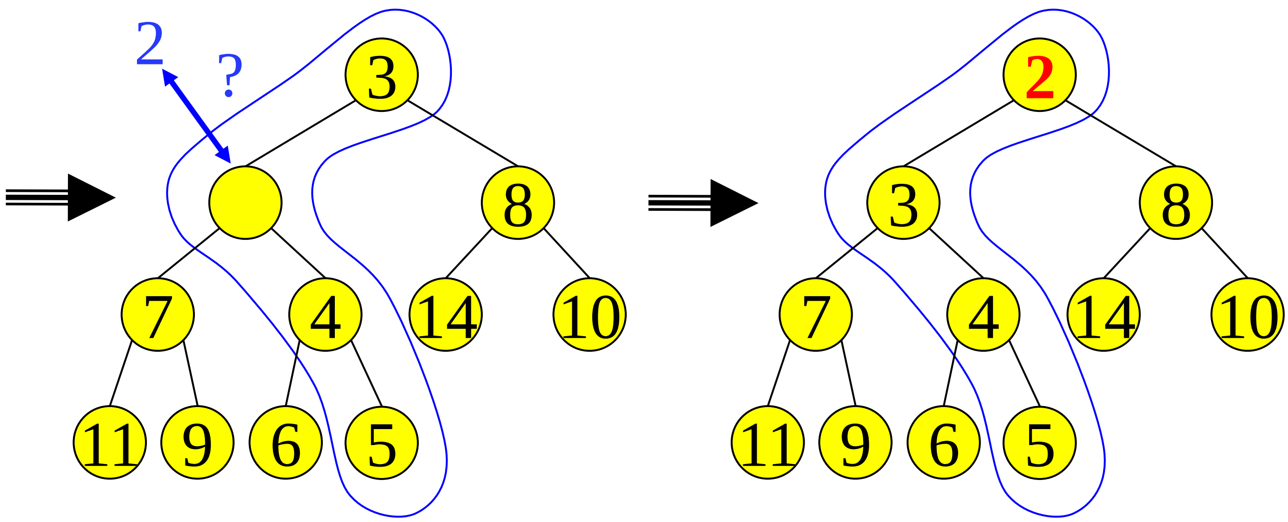


Insert: Percolate Up



- Start at last node and keep comparing with parent A[i/2]
- If parent larger, copy parent down and go up one level
- Done if parent ≤ item or reached top node A[1]
- Run time?

Insert: Percolate Up



- Done if parent ≤ item or reached top node A[1]
- Run time?

PercUp

```
/**
 * Insert item x, allowing duplicates.
*/
void insert( const Comparable & x ){
  if( currentSize == array.size( ) - 1 )
    array.resize( array.size( ) * 2 );

  // Percolate up
  int hole = ++currentSize;
  Comparable copy = x;

  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 ] );
}
```

BuildHeap

- The binary heap is sometimes constructed from an initial collection of items
  - can be done with N successive inserts.(O(N) average but O(N logN) worst-case)

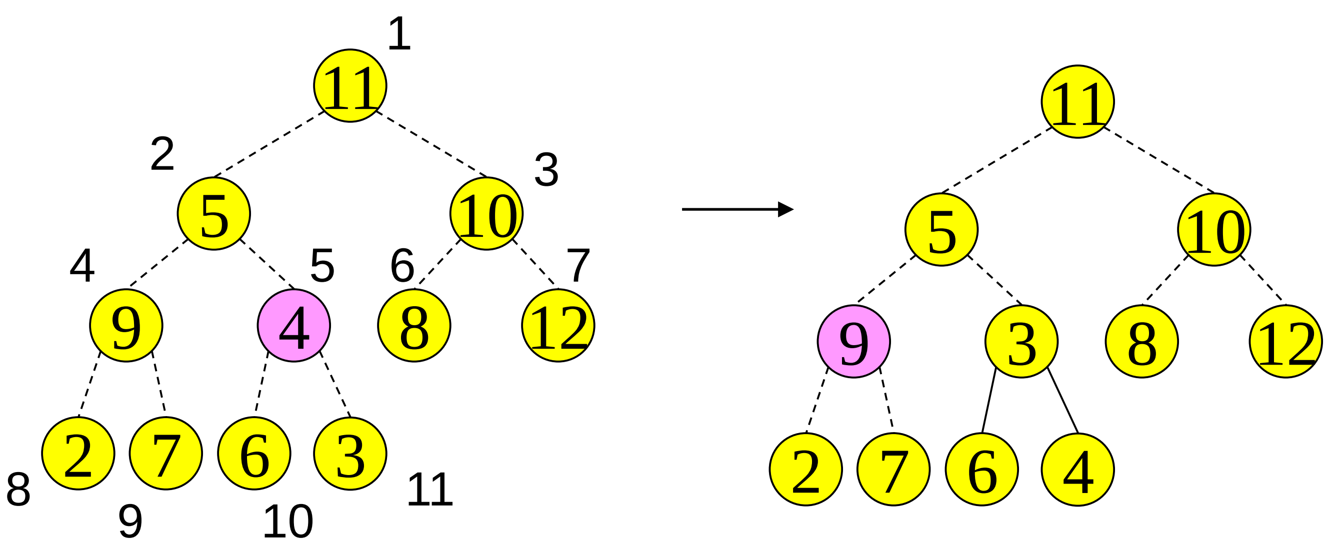
  - buildHeap routine

BuildHeap

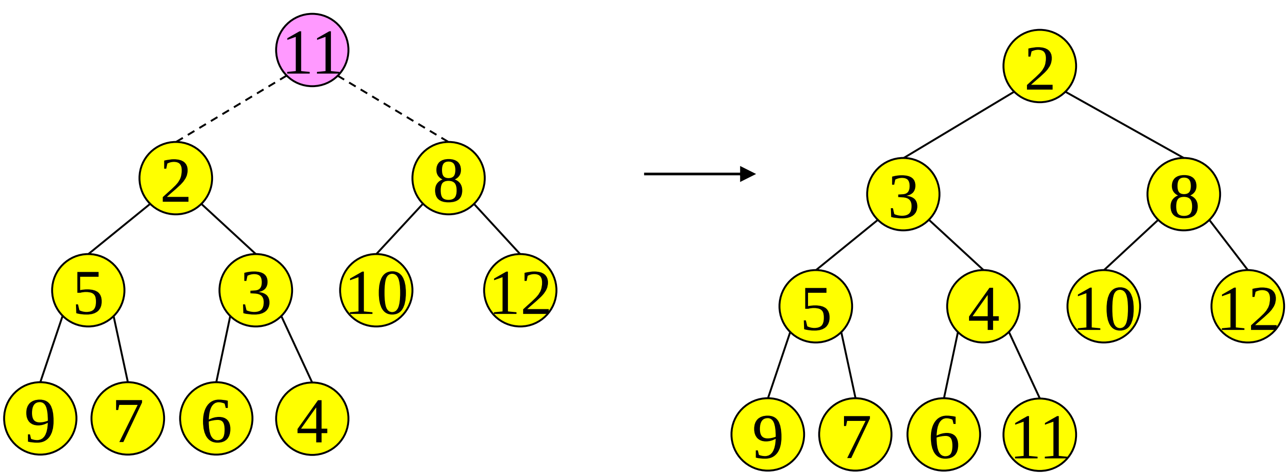
```
/**
* Establish heap order property from
* an arbitrary arrangement of items.
* Runs in linear time.
*/
void buildHeap( ) {
for( int i = currentSize / 2; i > 0; --i )
   percolateDown( i );
}
```

BuildHeap

N=11



Build Heap



Analysis of Build Heap

- Assume $N = 2^K - 1$
  - Level 1: k -1 steps for 1 item
  - Level 2: k - 2 steps for 2 items
  - Level 3: k - 3 steps for 4 items
  - Level i : k - i steps for 2^{i-1} items

$$\text{Total Steps} = \sum_{i=1}^{k-1} (k-i)2^{i-1} = 2^k - k - 1$$
$$= O(N)$$

Binary Heap Analysis

- Space needed for heap of N nodes: O(MaxN)
  - An array of size MaxN, plus a variable to store the size N
- Time
  - FindMin: O(1)
  - DeleteMin and Insert: O(log N)
  - BuildHeap from N inputs : O(N)

Homework

- Exercise 6.2, 6.3, 6.4
- Deadline: to be confirmed.