CS 315 March 18

Goals:

- Heap (Chapter 6)
  - priority queue
  - definition of a heap
  - Algorithms for
    - *Insert*
    - *DeleteMin*
    - *percolate-down*
    - *Build-heap*

Priority queue

- primary operations:
    - Insert
    - deleteMin

- secondary operations:
    - Merge (union)
    - decreaseKey
    - increaseKey

- Performance goal:
    - O(log n) for all operations (worst-case)

Linked list implementation of priority queue

- option 1: keep the array sorted
  - insert O(n) time
  - deleteMin O(1) time

- option 2: array is not sorted
  - insert O(1) time
  - deleteMin O(n) time

Similar performance can be observed with sorted array and unsorted array.

All these options require O(n) operations for one of the two operations.

More efficient implementations of priority queue

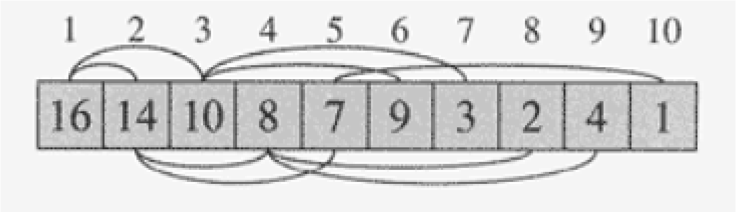
- binary heap
- skew heap
- binomial heap

No pointers are used (like in hashing)

- binary search tree

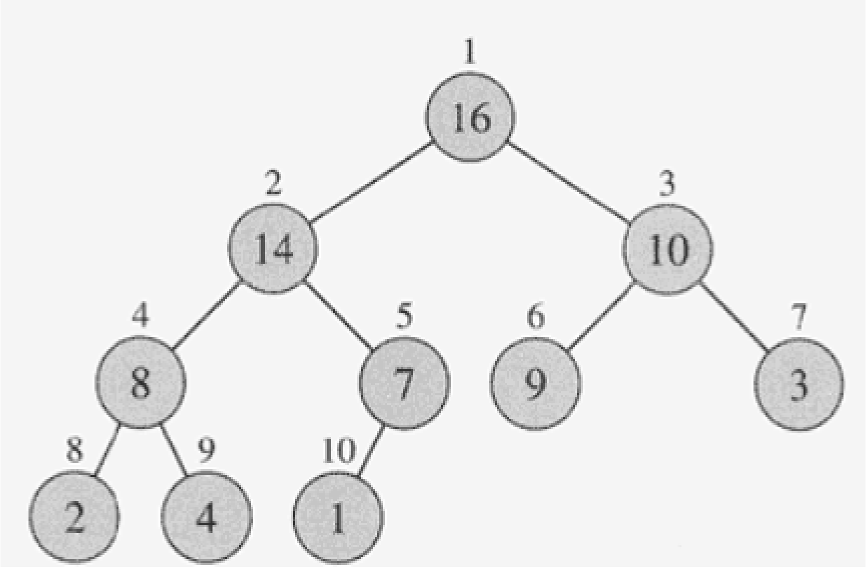
Uses pointer (like a linked list)

Binary Heap as an array and as a tree



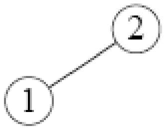
Array of 10 keys stored in A[1] through A[10]

Can be viewed as a tree. Index j has index 2j and 2j+1 as children.

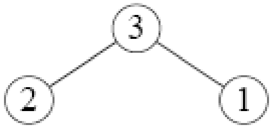


Connection between size and height of a heap

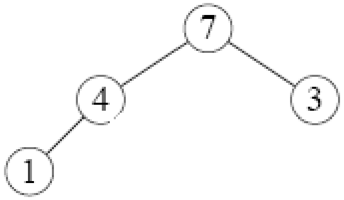
h = 1



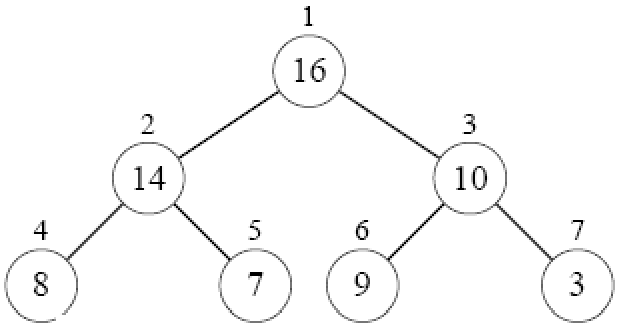
h = 2



h = 2



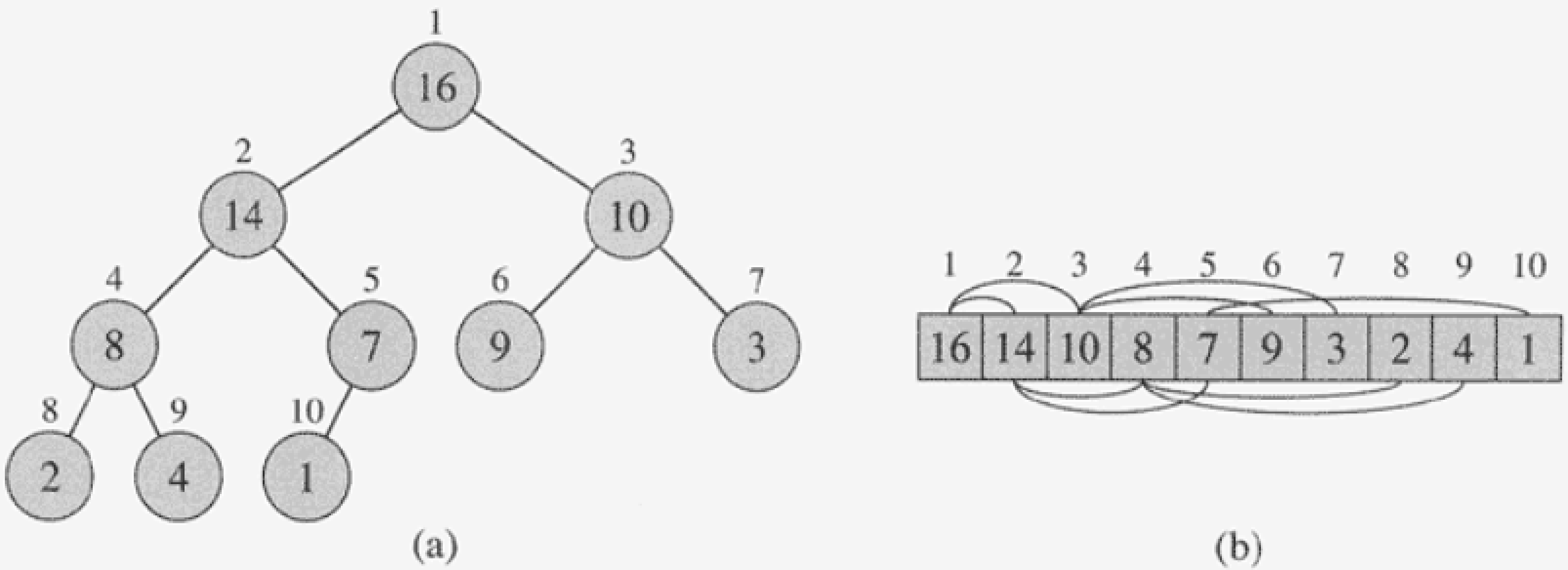
h = 3



h = 3

In general, if the number of nodes is N, height is at most $\lfloor \log_2 (N+1) \rfloor$ = O(log N)

Max-Heap property



(a)

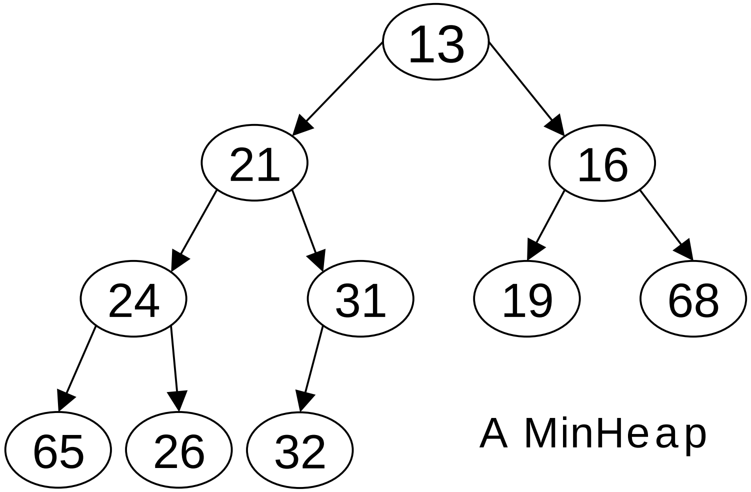
(b)

Key at each node must be bigger than or equal to its two children.

Min-heap: parent key < = child key.

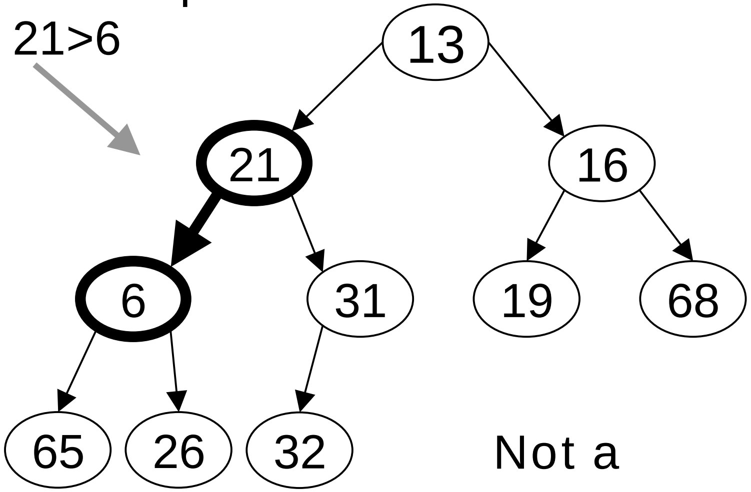
Left-child can be < or = or > than right-child. See above example.

MinHeap and non-Min Heap examples

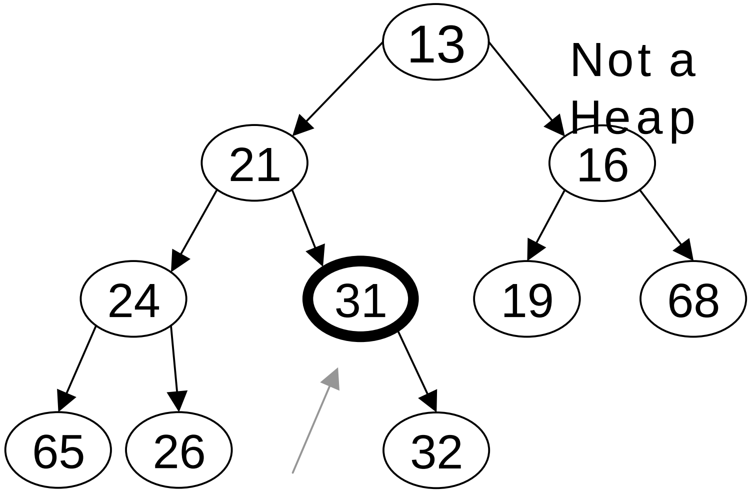


A MinHeap

Violates MinHeap property 21>6

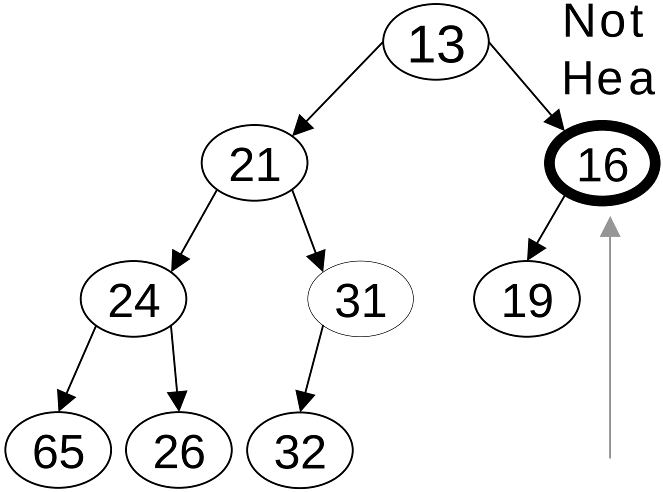


Not a Heap



Not a Heap

Violates heap structural property

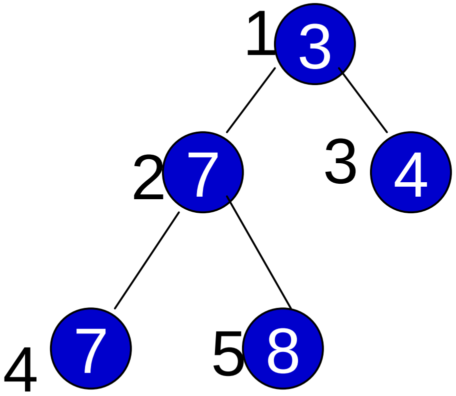


Not a Heap

Violates heap structural property

Heap class definition

```
class BinaryHeap
   {private:
    vector<Comparable> array;
    int currentSize;
    public:  // member functions
      }
```

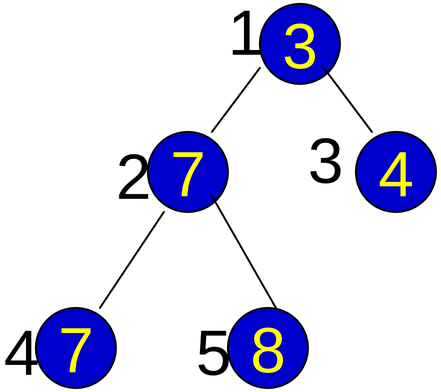


A

1	2	3	4	5
3	7	4	7	8

Note: currentSize is different from the size of the vector (which is the number of allocated memory cells).

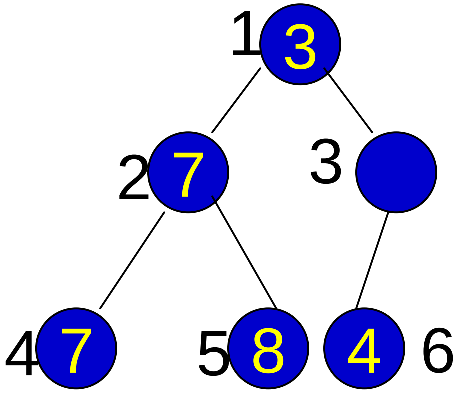
Heap Insertion Example



Insert 2 into the heap

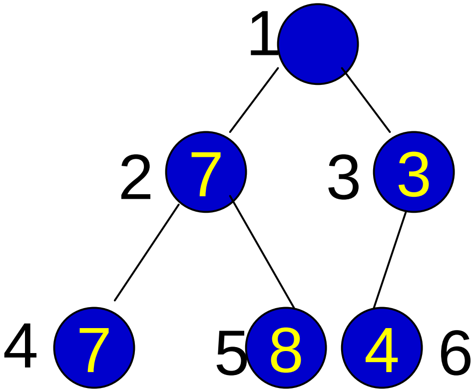
Create a hole and **percolate it up**.

Size = 6, so initially, j= 6; Since H[j/2]= 4 > 2, H[3] is copied to H[6]. Thus the heap now looks as follows:



The new value of j=3. Since H[j/2] =H[1] = 3 is also greater than 2, H[1] copied to H[3]. Now the heap looks as in the next slide.

Heap Insertion Example continued

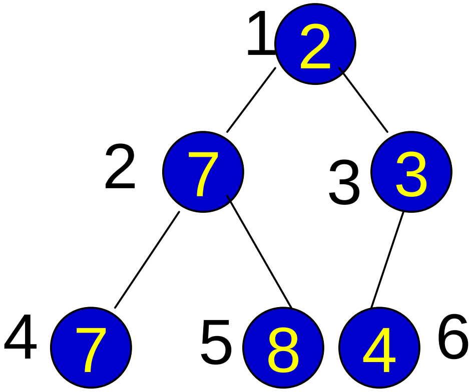


The new value of j = [3/2] = 1.

Now, j/2 = 0 so the iteration stops.

The last line sets H[j] = H[1] = 2.

The final heap looks as follows:



Code for insert

```
void insert(const Comparble & 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;

}
```

Try some more examples and make sure that the code works correctly.

Complexity of insert operation

The height of a heap with n nodes is O(log n). The insert procedure percolates up along a path from a leaf to the root.

Along the path at each level, one comparison and one data movement is performed.

Conclusion: **Insert** performs *O (log n)* elementary operations (comparisons and data movements) to insert a key into a heap of size *n* in the worst-case.

DeleteMin Operation

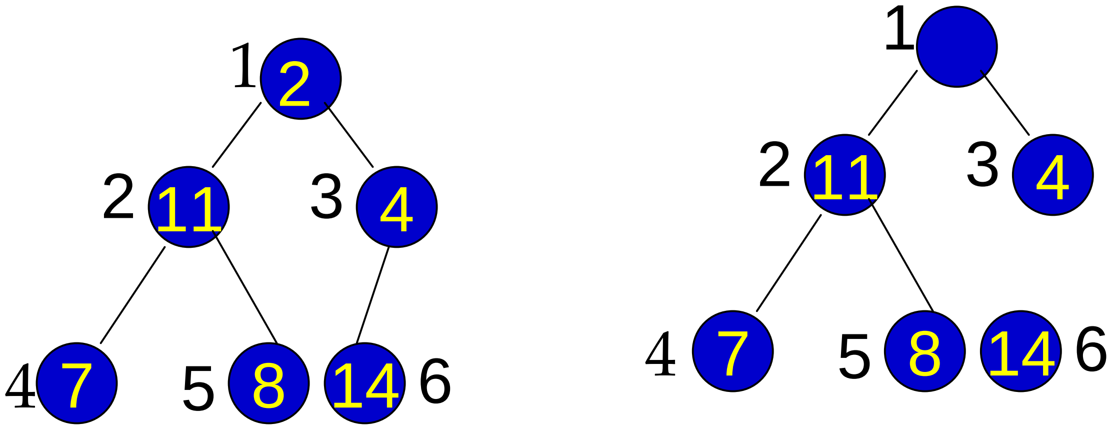
- Observation: The minimum key is at the root.

- FindMin() can be performed in O(1) time:

```
Comparable findMin() {
  if (currentSize == 0)
    cout << “heap is empty.” << endl;
  else return array[1];
 }
```

- Deleting this key creates a hole at the root and we perform **percolate down** to fill the hole.

Example of DeleteMin

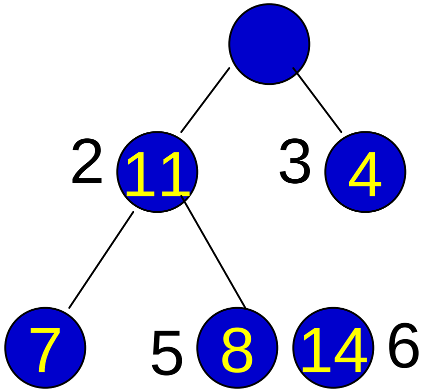


After deleting the min key, the hole is at the root. How should we fill the hole?

We also need to vacate the index 6 since the size of the heap now reduces to 5.

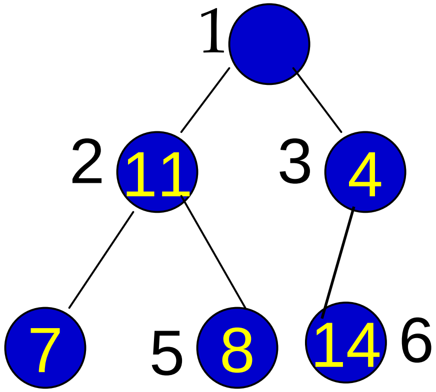
We need to find the right place for key 14.

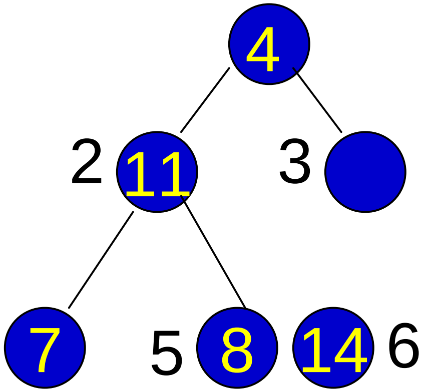
Example of DeleteMin

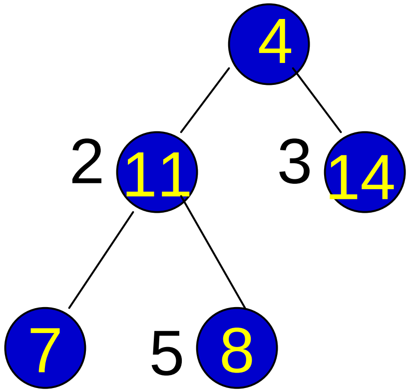


Hole should be filled with smallest key, and it is in array[2] or array[3].

Example of DeleteMin

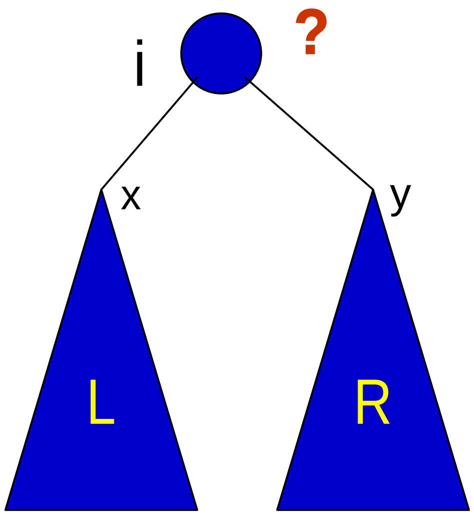






Try some more examples

DeleteMin – Outline of general case



Vacated
key = k

Hole is currently in index i.
its children are 2i and 2i+1.

Suppose array[2i] = x,
array[2i+1]= y.

- Case 1: (k <= x) && (k <= y) In this case, put k in array[I] and we are done.

- Case 2: (k > x) && (y > x). Thus, x is the minimum of the three keys k, x and y. We move the key x up so the hole percolates to 2i. Now the hole percolates down to 2i.

- Case 3: (k >x) && (x >y). Move y up and the hole percolates down to 2i+1.

- Case 4: the hole is at a leaf. put k there and this terminates

Code for delete-min

```
/**
     * Remove the minimum item and place it in minItem.
     * Throws Underflow if empty.
     */
    void deleteMin( Comparable & minItem )
    {
        if( isEmpty( ) )
            throw UnderflowException( );
        minItem = array[ 1 ];
        array[ 1 ] = array[ currentSize-- ];
        percolateDown( 1 );
    }
```

Code for percolateDown

```
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;
    }
```