Data Structures

# Lesson 9

# Heaps

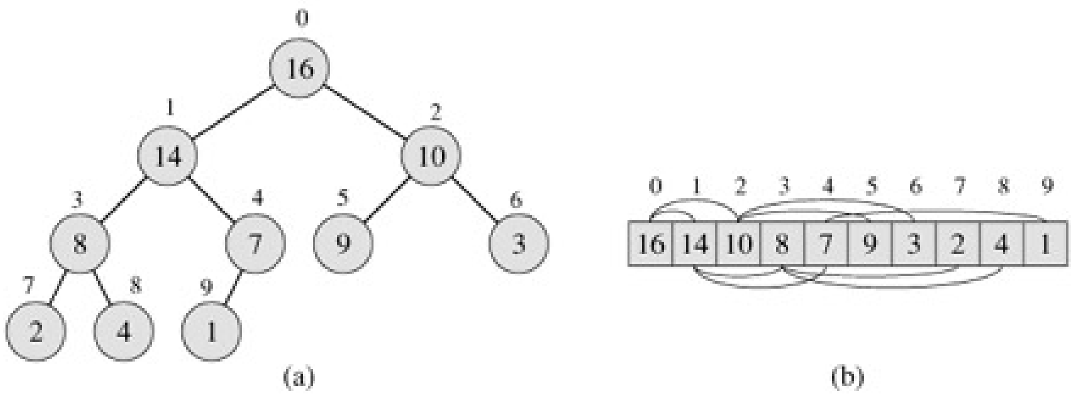
James C.C. Cheng
Department of Computer Science
National Chiao Tung University

# Overview

- Heap
  - A complete binary tree
  - It is implemented by an array
  - Its node has four member data

```
template <class T>
struct  HeapNode{
    int left, right, parent
    T key;
    HeapNode(const T& k = T(), int L=-1, int R = -1, int P = -1):
       key(T), left(L), right( R ), parent(P){}
}
```

  - Max-heap
    - for every node *i* other than the root, A[A[i].parent].key ≥ A[i].key
  - Min-heap
    - for every node *i* other than the root, A[A[i].parent].key ≤ A[i].key



# Heapify

- The kernel function of Heap maintaining

```
// A is a heap data array, i is an index of node
// heap_size is the number of nodes in A
template <class T>
void    MaxHeapify (HeapNode <T>*  A,  int i, int heapSize){
            int L =A[i].left,     R = A[i].right;
            int largest;

            if (L < heapSize and A[L] > A[i])  largest  = L;
            else largest = i;

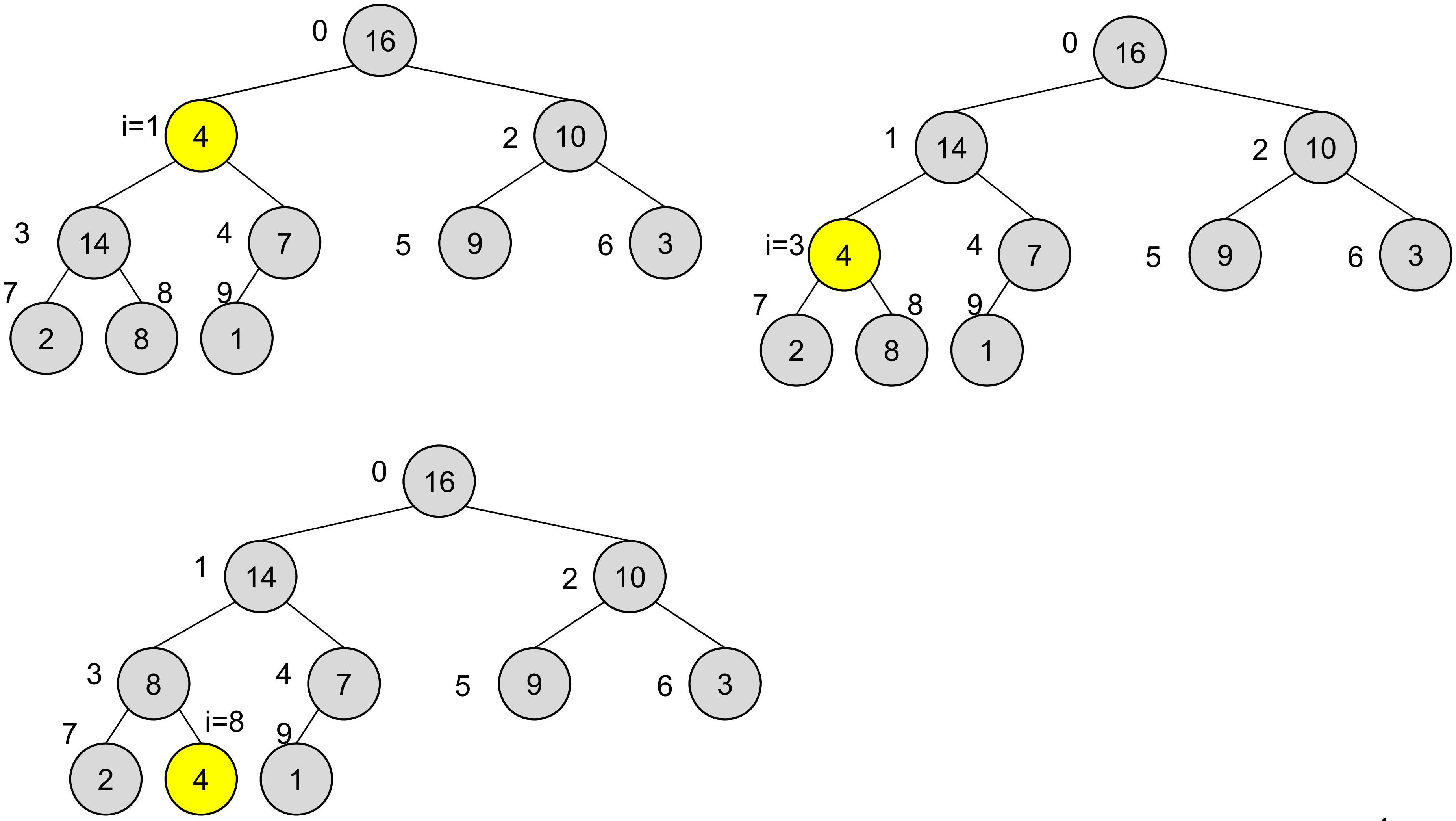
            if (R < heapSize and A[R] > A[largest] ) largest = R;

            if (largest != i ){
                        swap( A[i], A[largest]);
                        MaxHeapify (A, largest, heapSize);
            }
}
```

  - Time = $O(\log n)$, where $n$ is the number of nodes

# Heapify

- Example:



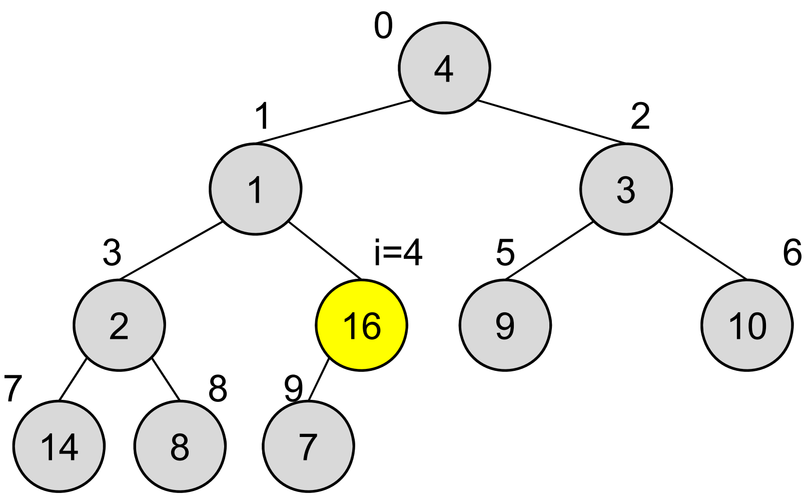
# Heap Building

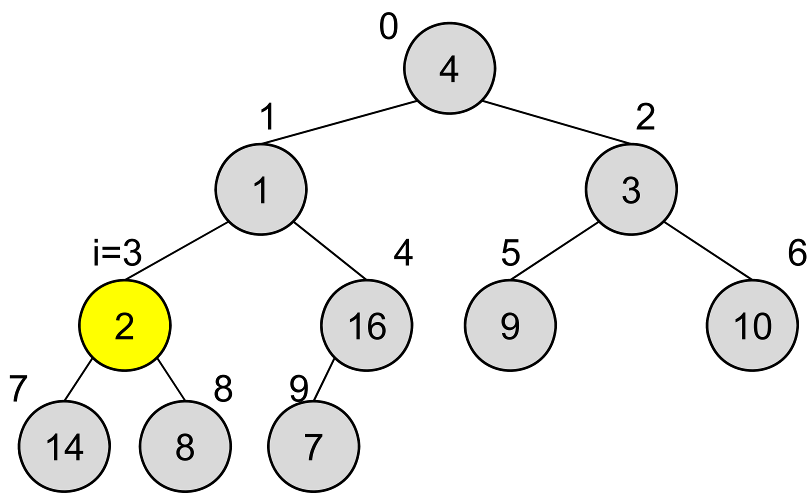
- Bottom-up heap building:

```
template <class T>
void BuildMaxHeap (HeapNode<T>* A, int heapSize){
    for(int i = (heapSize-1) / 2; i >=0; --i)
            MaxHeapify (A, i, heapSize)
}
```

- Example: A

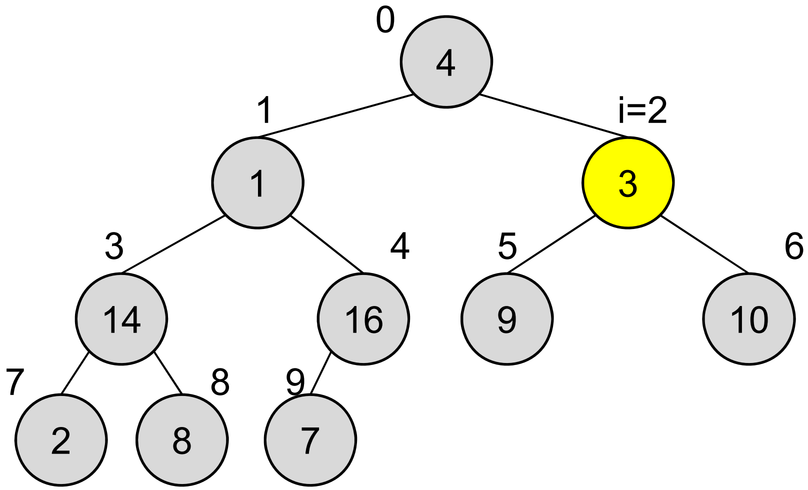
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |

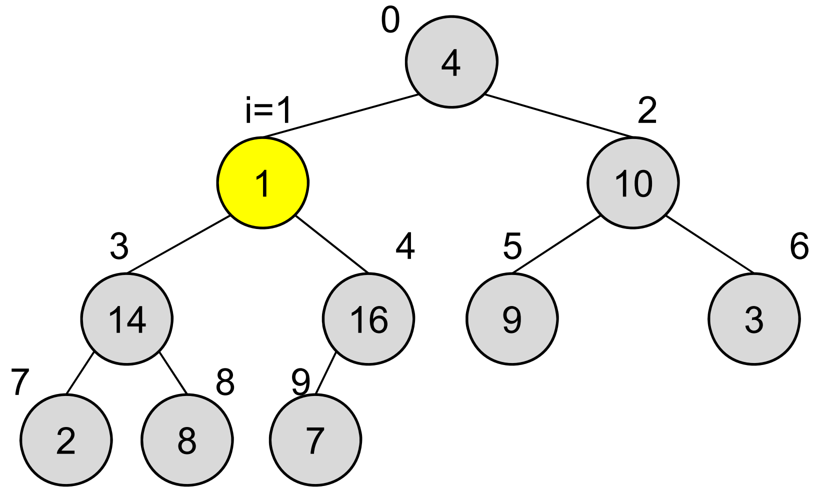


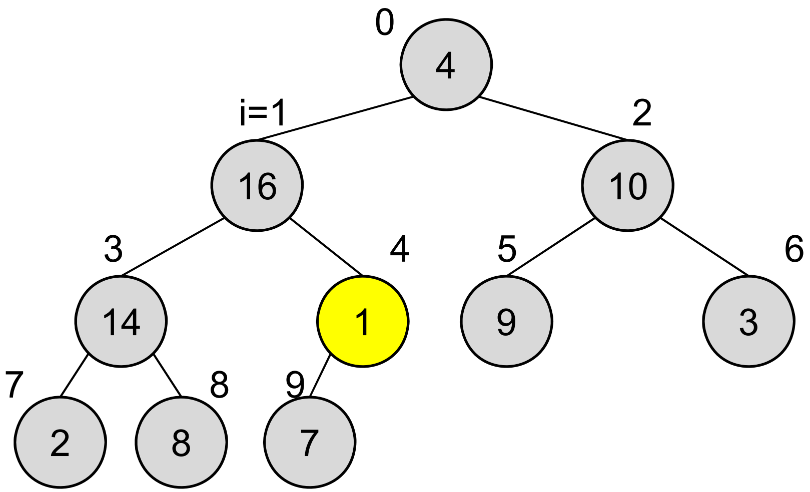



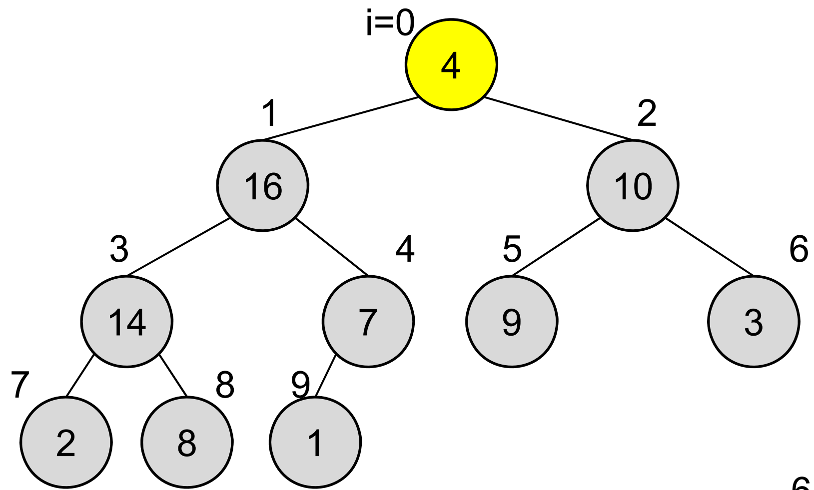
# Heap Building

- Example:



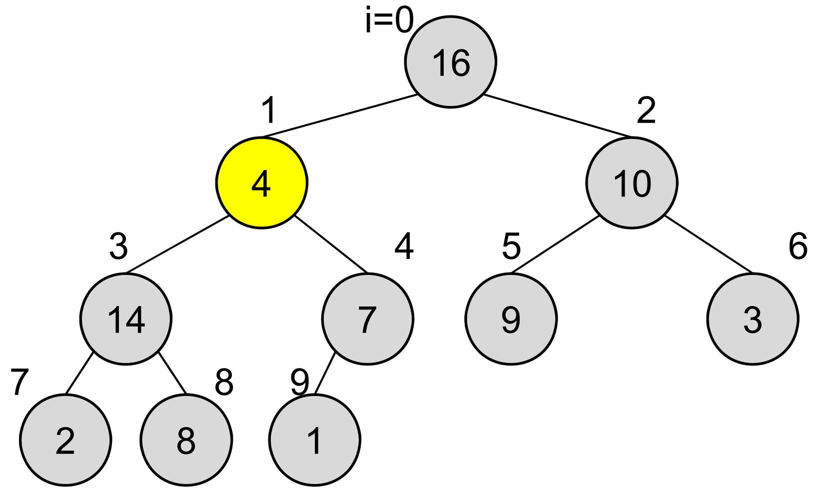



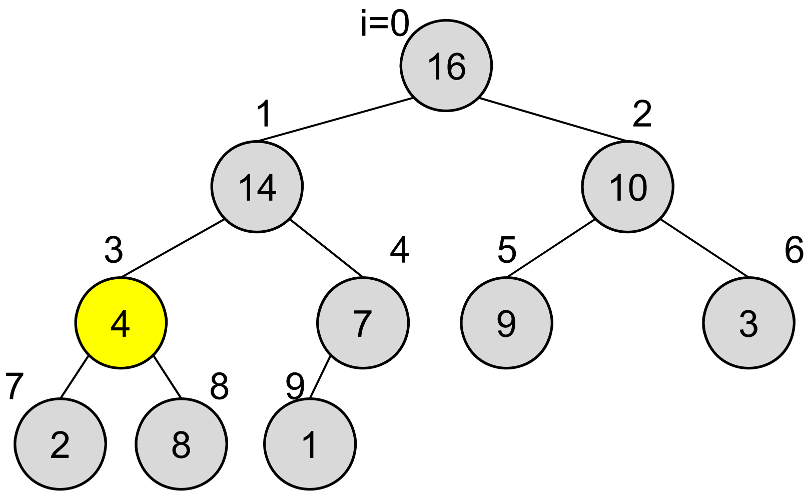



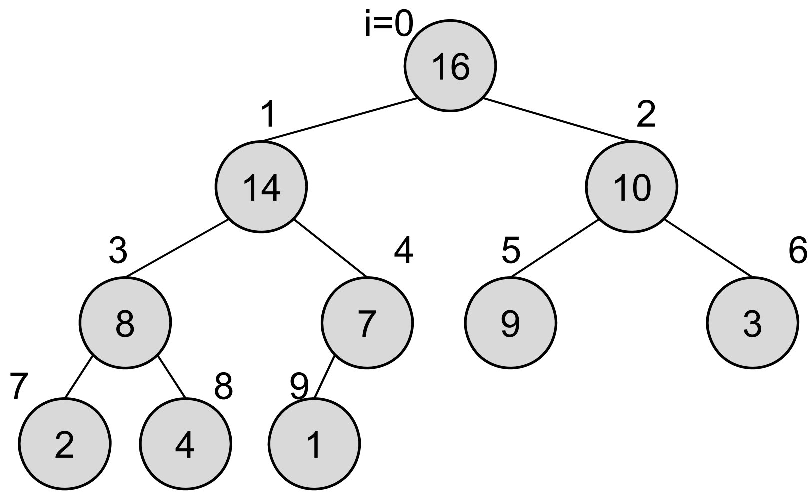



# Heap Building

- Example:



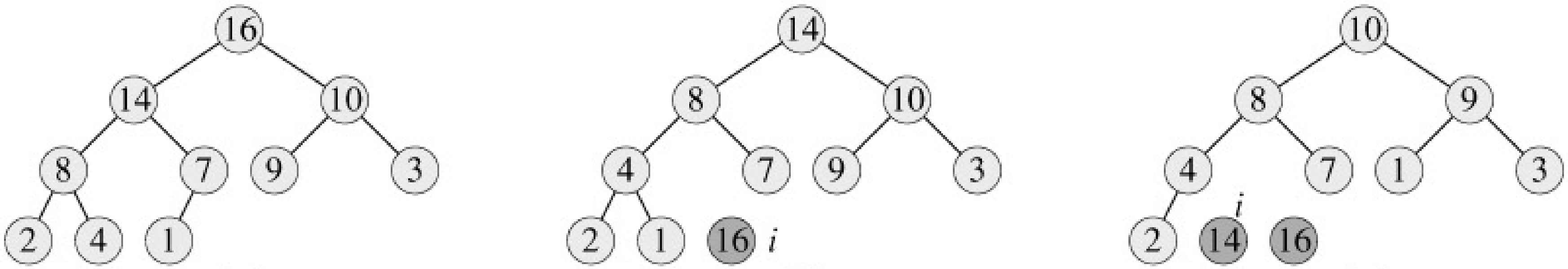
# Heap Building

- Time complexity
  - Given n nodes, and
    - the height of heap tree $h = 1 + \lfloor \log n \rfloor$
    - the depth of node i: $h_i = 1 + \lfloor \log(i+1) \rfloor$
  - The maximum time of **MaxHeapify**() on node i: $h - h_i$

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{(n-1)/2} (h - h_i) \\
&\le \sum_{i=0}^{(n-1)/2} (\lfloor \log n \rfloor - \lfloor \log(i+1) \rfloor) = \frac{n-1}{2} \lfloor \log n \rfloor - \sum_{i=0}^{(n-1)/2} \lfloor \log(i+1) \rfloor \\
&\le \frac{n}{2} \log n - \int_1^{\frac{n}{2}} \log x dx = \frac{n}{2} \log n - \frac{n}{2} \log n + \frac{n}{2} \log 2 + \frac{n}{2} + 1 \\
&= O(n)
\end{aligned}
$$

# Heap Sort

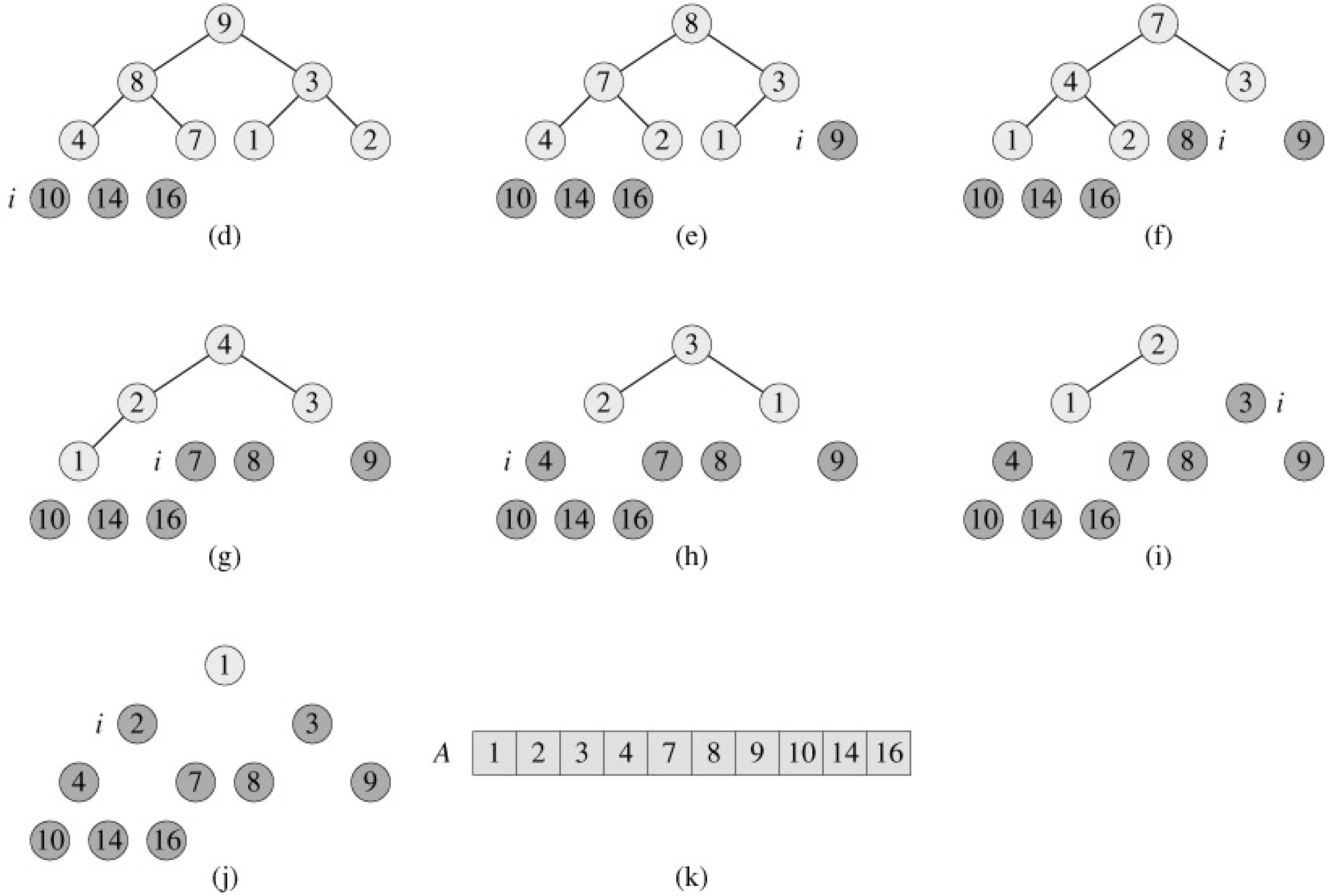
- ```
  template <class T>
  void HeapSort (HeapNode<T>* A, int heapSize, int arraySize){
          BuildMaxHeap(A, heapSize)
          for( int i = arraySize – 1; i>=1; --i){
                  swap(  A[0],  A[i] );
                  --heapSize;
                  MaxHeapify( A, 0, heapSize);
          }
  }
  ```
  - Time complexity: $O(n \log n)$,
- Example




(a) (b) (c)
  ```

# Heap Sort

- Example



# Priority Queues

- A ***priority queue*** is a data structure for maintaining a set A of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations.
  - T& ***HeapMax*** (A) ------ Θ(1)
    - returns the element of A with the largest key.
  - T ***HeapExtractMax*** (A , int& heapSize) ------ O(log$n$)
    - removes and returns the element of A with the largest key.
  - void ***HeapIncreaseKey*** (A, int i, const T& k) ------ O(log$n$)
    - increases the value of node $i$'s key to the new value $k$, where $k$ > A[i].
  - void ***HeapInsert*** (A, const T& x , int& heapSize ) ------ O(log$n$)
    - inserts the element $x$ into the set A = A ∪ {$x$}.

# Priority Queues

- ***HeapMax***: the root of a max heap is the largest key

```
template <class T> inline T&  HeapMax (HeapNode<T>*  A) { return A[0]; }
```

- ***HeapExtractMax***

```
template <class T>
T  HeapExtractMax(HeapNode<T>* A , int& heapSize){
        if (heapSize <= 0) return T();
        T  m = A[0];
        A[0]  = A[ --heapSize ];
        MaxHeapify(A, 0, heapSize);
        return m;
}
```

# Priority Queues

- ***HeapIncreaseKey***

```
template <class T>
void  HeapIncreaseKey (HeapNode<T>* A,  int i, const T& k){
        if (k > A[i]){
            A[i] = key
            while( i >= 0 && A[ A[i].parent ] < A[i] ){
                swap( A[i], A[ A[i].parent ] );
                i = A[i].parent ;
            }
        }
}
```

- Time: the maximum running time of the while loop is the height of heap tree
  - O(log*n*)

# Priority Queues

- ***HeapInsert***

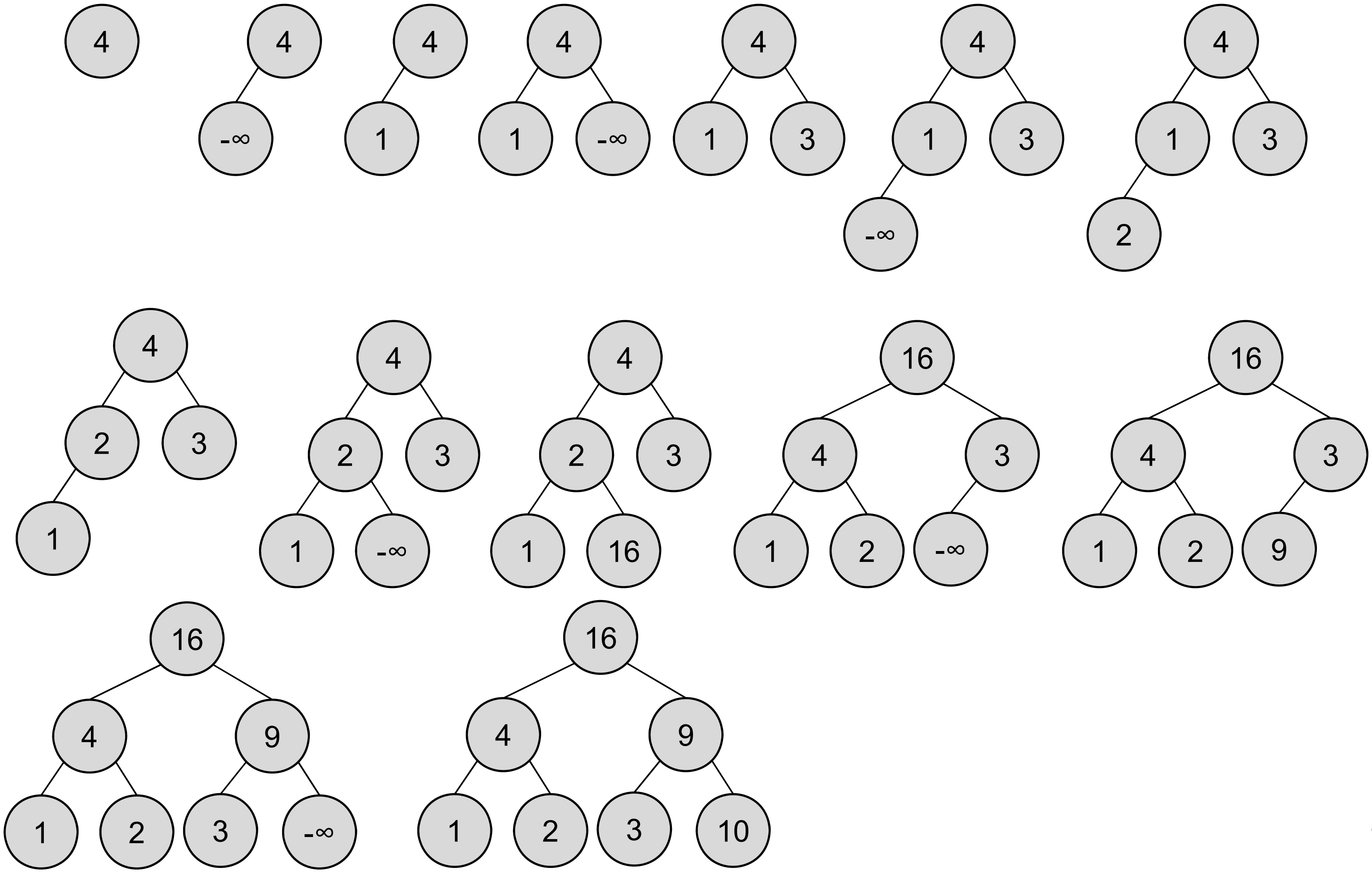
```
template <class T>
void  HeapInsert (HeapNode<T>*  A, const T& x , int& heapSize ){
              A[heapSize++] = -∞;
               HeapIncreaseKey ( A, heapSize - 1, x)
}
```

  - Time = O(log*n*)

# Priority Queues

- Example of insertion

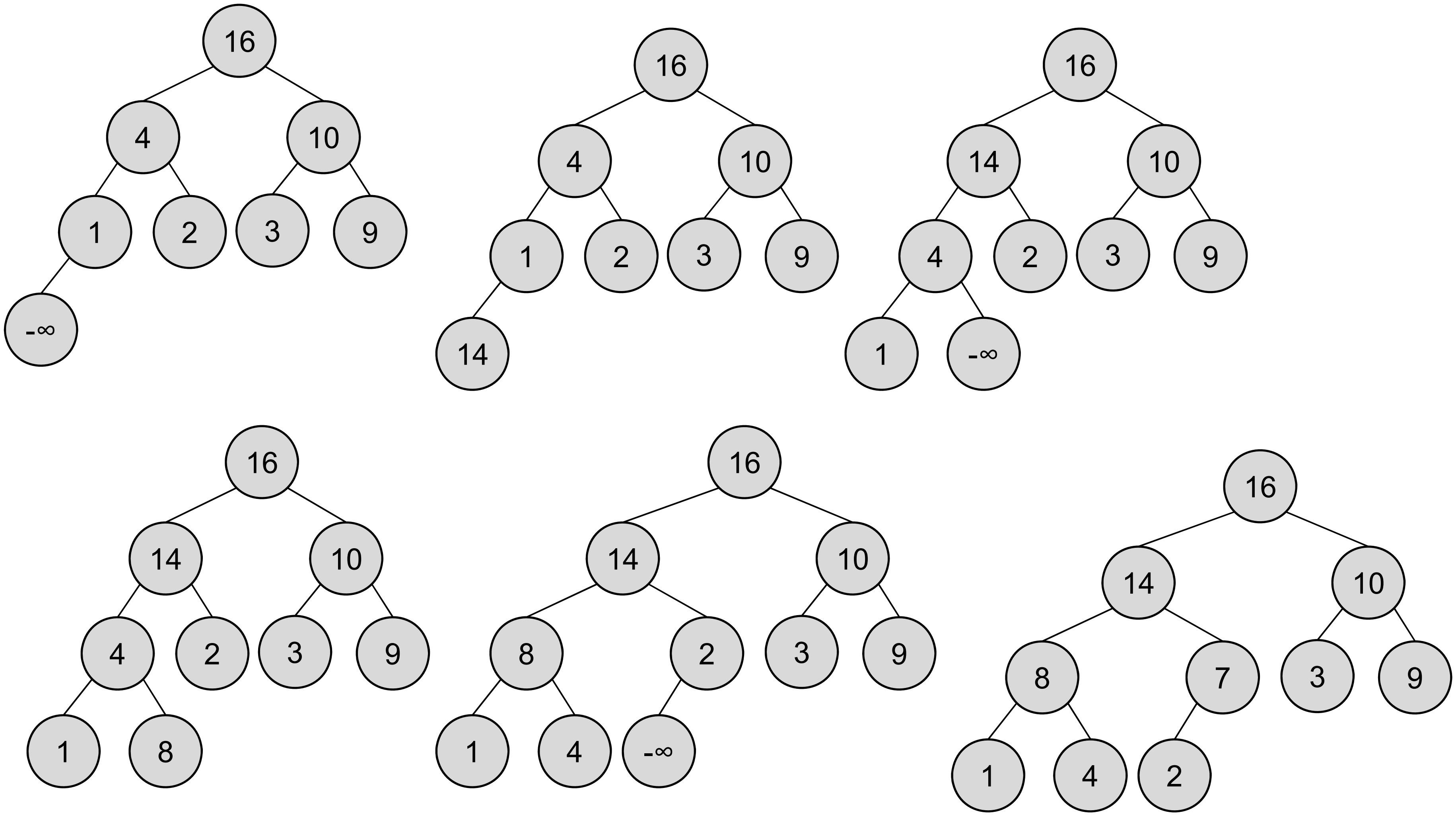
| A | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |



# Priority Queues

- Example of insertion

| A | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |



# Priority Queues

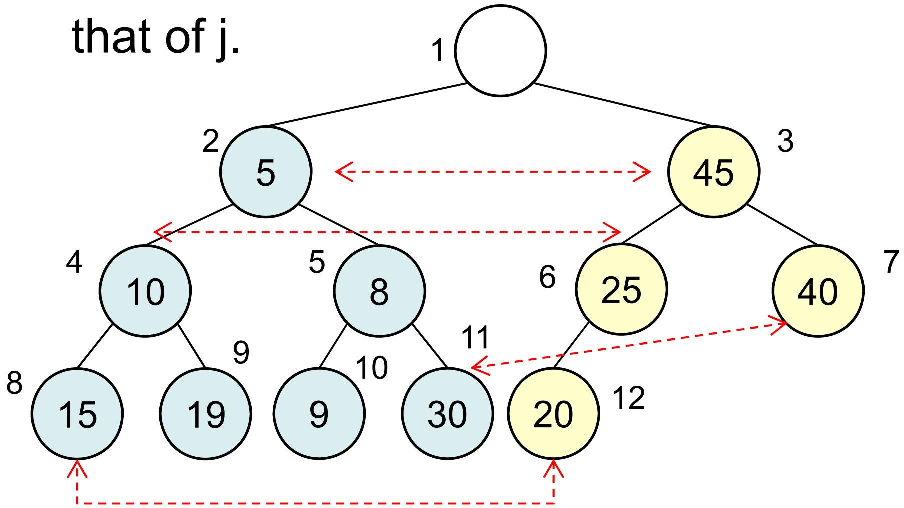
- Top-down heap building
  - Using ***HeapInsert***
  - Time:
    - Best case: $O(n)$
    - worst case: $\Theta(n \lg n)$

# Exercise

- Finish the deletion operation for max-heap
  - template <class T> void
    ***HeapDelete*** (HeapNode<T>* A, int i, int& heapSize );
    - A is the array of heap, i is the node to be deleted and heapSize is the number of nodes
    - Time: O(log*n*)
    - It is similar to ***HeapInsert***
- Using a boolean variable, MaxMin, to switch Max-heap and Min-heap
  - If MaxMin is true ➔ max-heap
  - Else ➔ min-heap

# Symmetric Min-Max Heap, SMMH

- Double-Ended Priority Queue, DEPQ
- Properties (old version)
  - The root is empty
  - The Left subtree is a min-heap
  - The Right subtree is a max-heap
  - If the right subtree is not empty, then let i be any node in the left subtree. Let j be the **corresponding node** in the right subtree. If such node j does not exist, then let j be the node in the right subtree that corresponds to the parent of i. The key in node i is less than or equal to that of j.




root: i= 1

$$j = i + 2^{\lfloor \log_2 i \rfloor - 1}, if\,(j > n)\, j = \left\lfloor \frac{j}{2} \right\rfloor$$

$$i = j - 2^{\lfloor \log_2 j \rfloor - 1}$$

# Symmetric Min-Max Heap, SMMH

- Check whether the node *i* is in the max-heap, $\Theta(1)$
  - The maximum number of nodes on height *h* is $2^{h-1}$, $h \geq 1$
  - The maximum number of nodes in a binary tree of height *h* is $2^h - 1$, $h \geq 1$
  - The index of mid node *m* on height *h* is

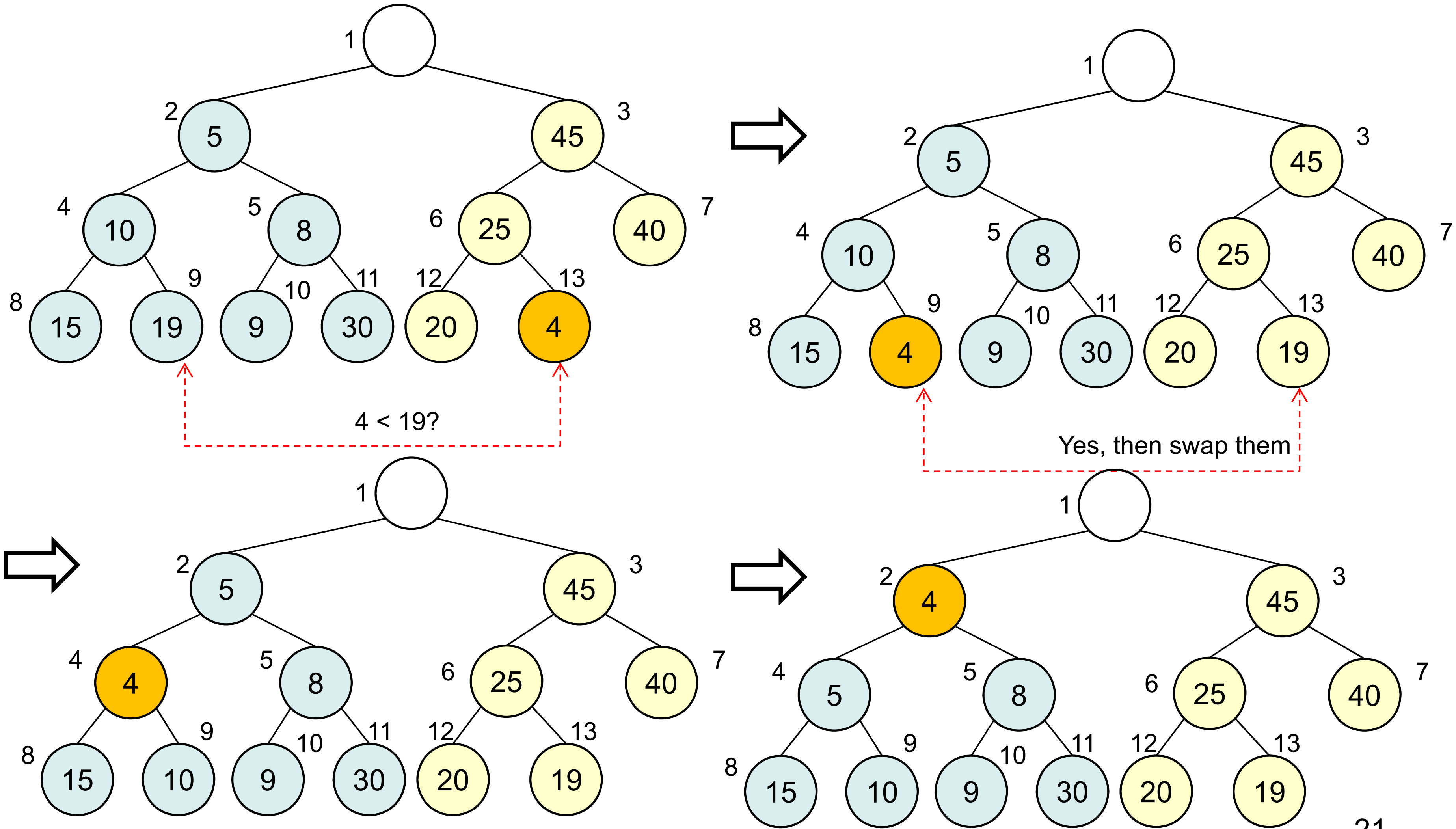
$$m = (2^{h-1} - 1) + 2^{h-2} = 3 \times 2^{h-2} - 1$$

$$, h \geq 2$$

```
bool MaxHeap(int i){
   if ( i > 3 * pow(2, floor(log2i) - 1) – 1 ) return true;
   else return false;
}
```

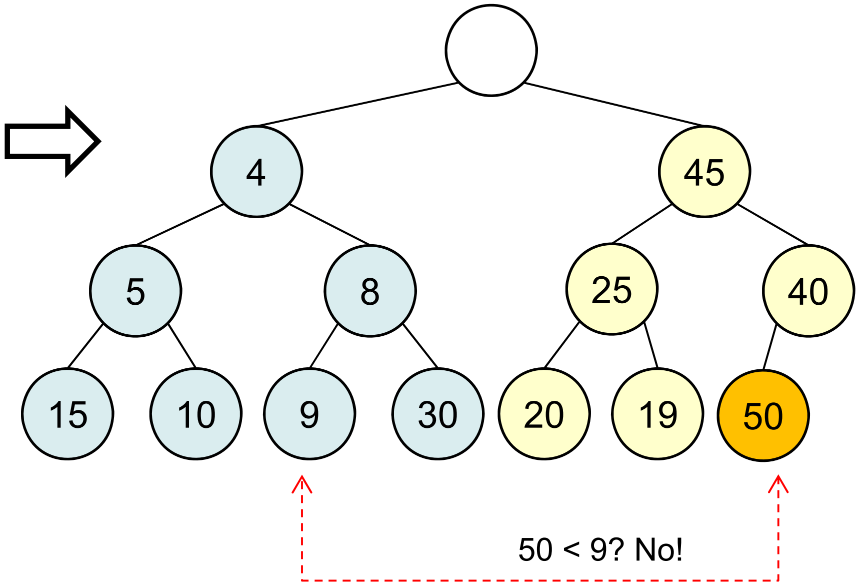
# Symmetric Min-Max Heap, SMMH

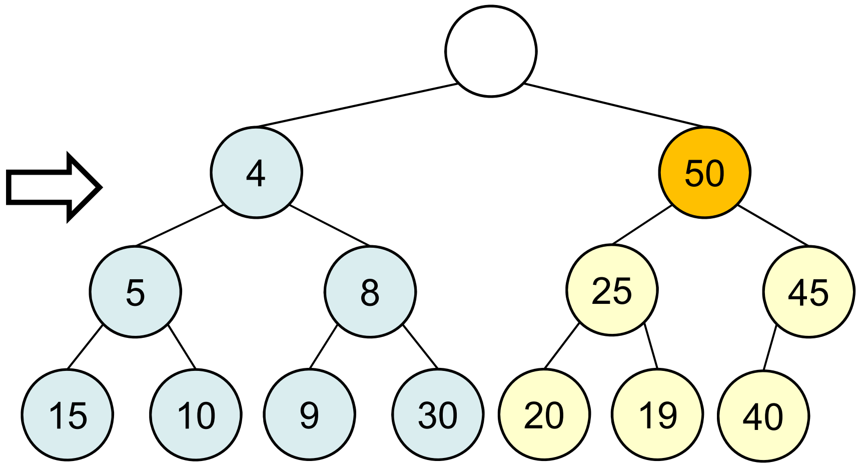
- Insertion



# Symmetric Min-Max Heap, SMMH

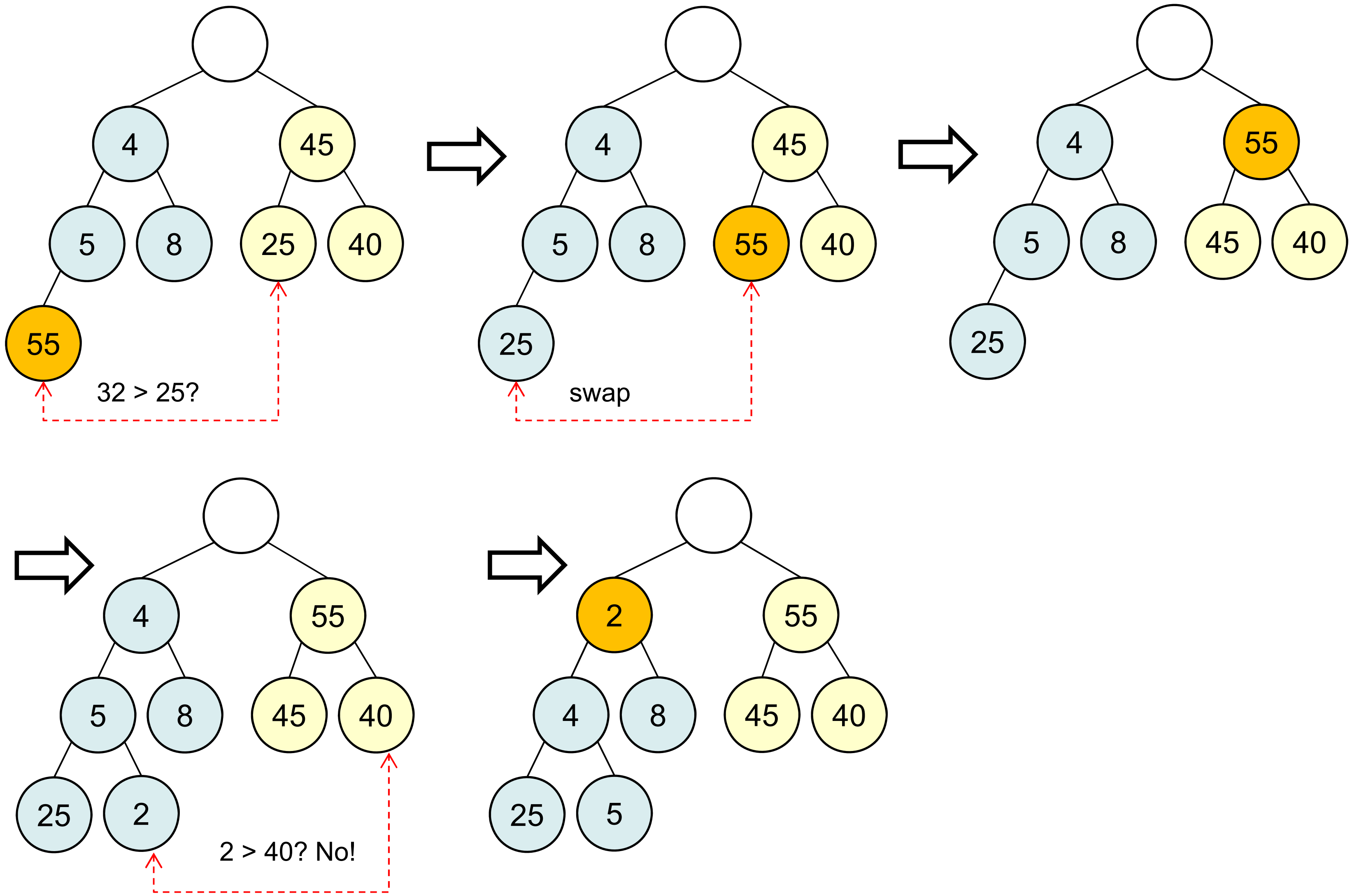
- Insertion



# Symmetric Min-Max Heap, SMMH

- Insertion

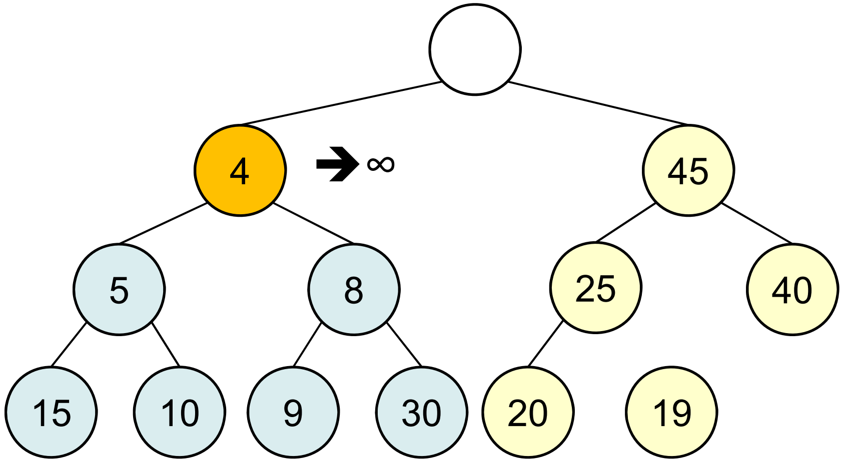


# Symmetric Min-Max Heap, SMMH

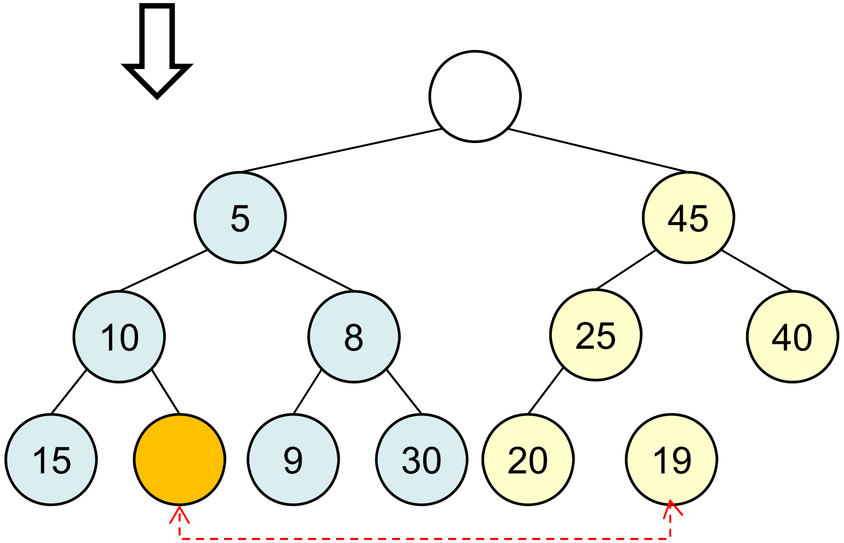
- Insertion
  - Time complexity = $O(\log n)$
    - Check whether the node is in max-heap: $\Theta(1)$
    - Find the corresponding node: $\Theta(1)$
    - Swap: $\Theta(1)$
    - Heap adjustment: Heapify(): $O(\log n)$

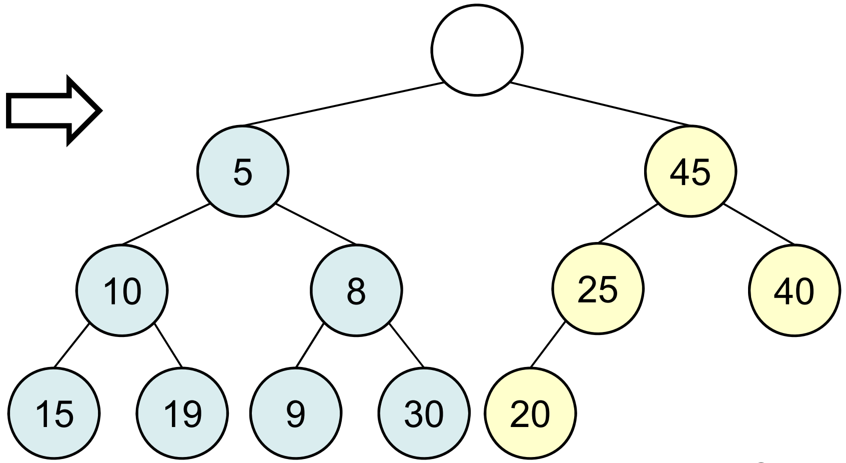
# Symmetric Min-Max Heap, SMMH

- Deletion
  - Only the maximum node and the minimum node can be removed




Remove the minimum node:
1. Clear it.
2. Move the last node as t
3. Top-down min heap adjustment.
4. Insert t to the empty node.



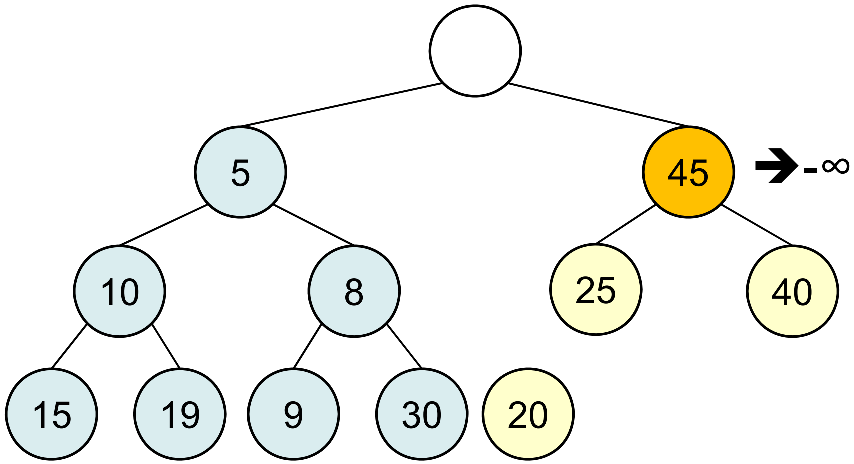



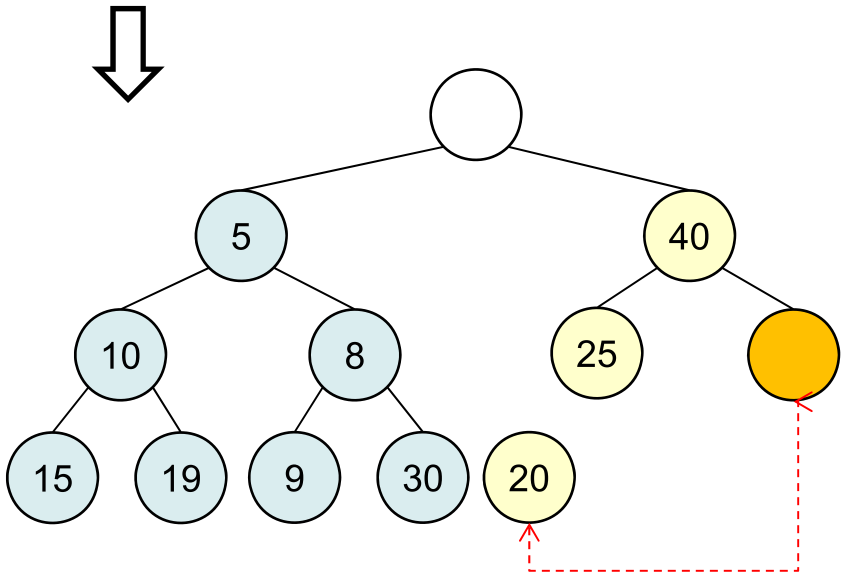
# Symmetric Min-Max Heap, SMMH

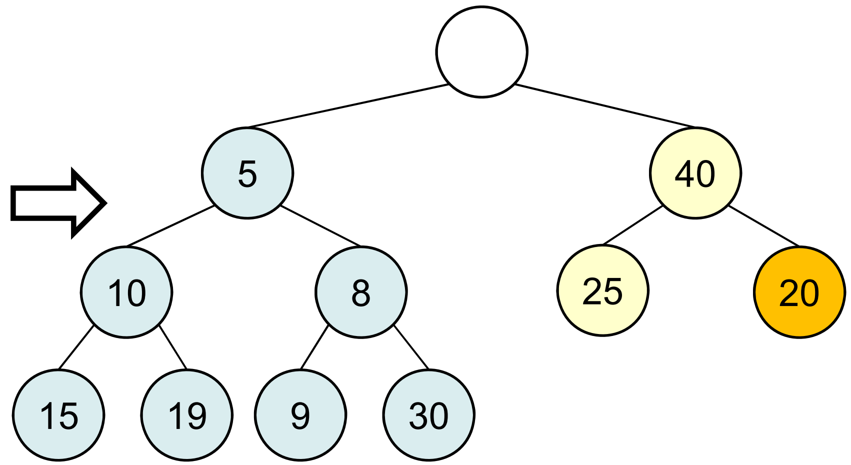
- Deletion

Remove the maximum node:

1. Clear it.
2. Move the last node as t
3. Top-down max-heap adjustment.
4. Insert t to the empty node

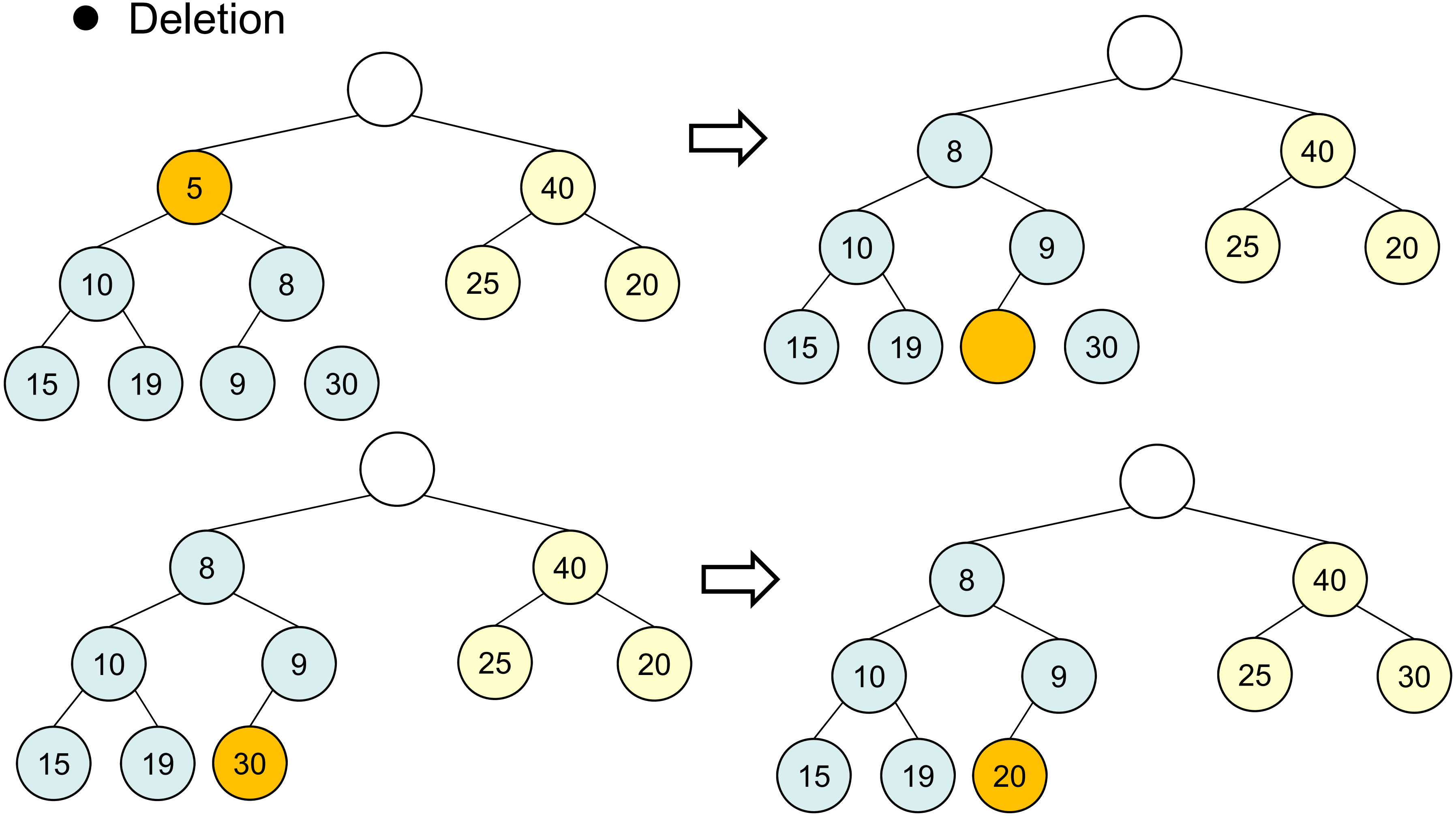


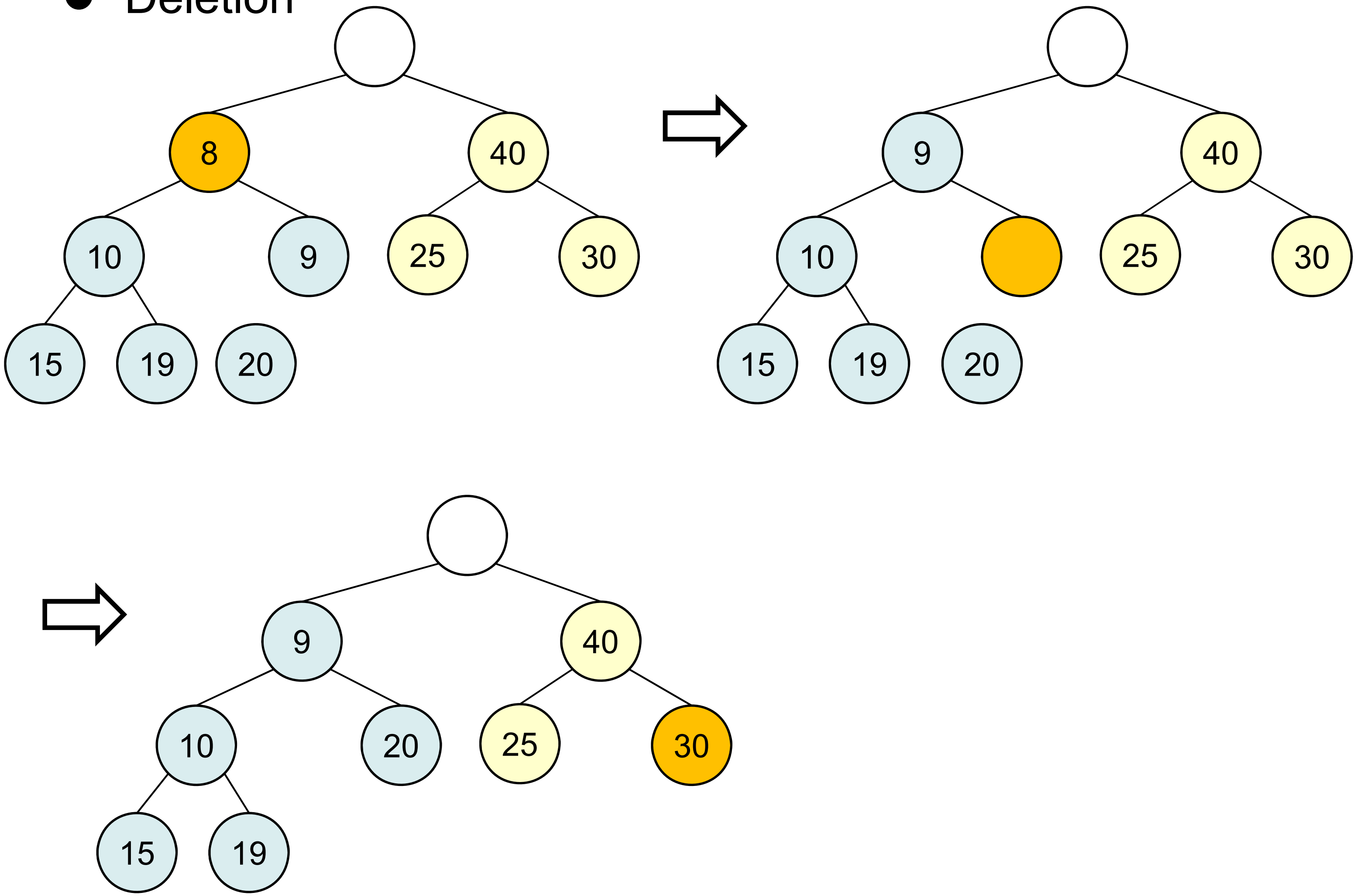
# Symmetric Min-Max Heap, SMMH

- Deletion



# Symmetric Min-Max Heap, SMMH

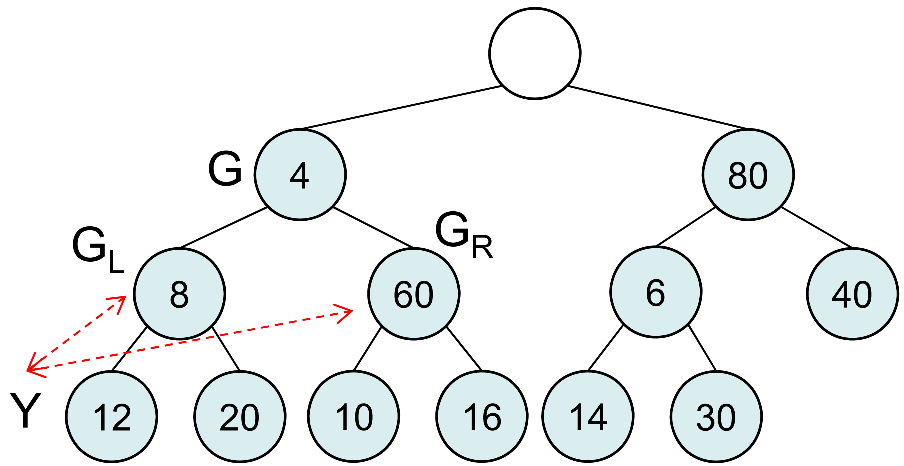
- Deletion



# SMMH v2.0

- Double-Ended Priority Queue, DEPQ
- Properties (new version)
  - The root is empty
  - Let X be any node in SMMH
  - X's Left subtree is a min-heap
  - X's Right subtree is a max-heap
  - Left sibling ≤ Right sibling

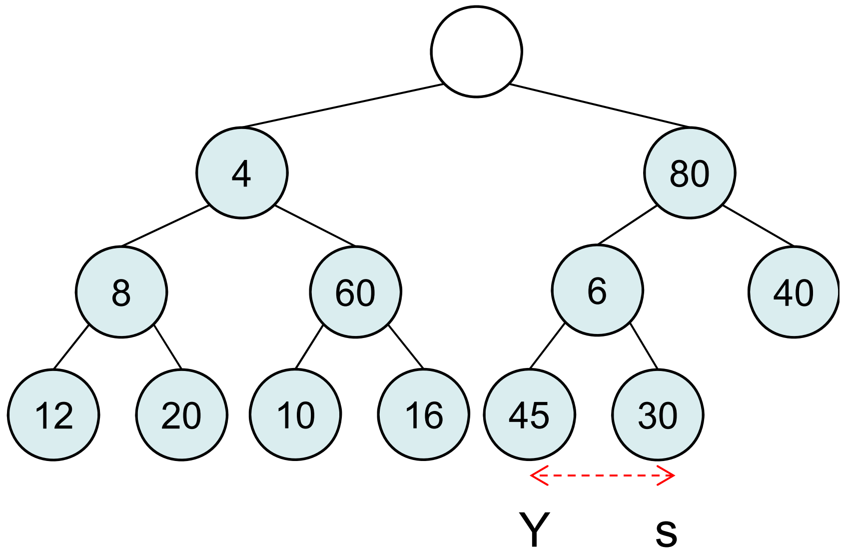
Let Y be any node in SMMH and Y has grandparent G
- $G_L$, the left child of G ≤ Y
- $G_R$, the right child of G ≥ Y



# SMMH v2.0

- Sibling adjustment

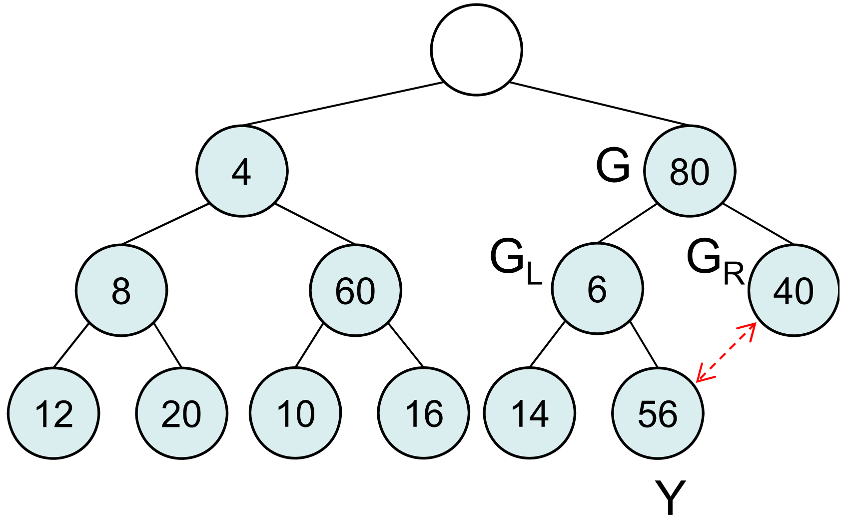
```
// Note that the index of root is 0;
int AdjSibling(T* smmh, int Y, int MaxSize ){
    int s;
    if (Y is left child){
            s = Y + 1;
            if( s >= MaxSize) return Y;
            if(smmh[Y] > smmh[s]) {
                    swap(smmh[Y], smmh[s]);
                    return s;
            }
    }
    else{ // Y is right child
            s = Y – 1;
            if(smmh[Y] < smmh[s]){
                    swap(smmh[Y], smmh[s]);
                    return s;
            }
    }
    return Y;
}
```



# SMMH v2.0

- Grandparent adjustment

```
// Note that the index of root is 0;
int AdjG(T* smmh, int Y, int MaxSize ){
    if (Y <= 2) return Y;
    int G = Y's grand parent;
    int GL = G's left child, GR = G's right child;
    if(smmh[GL] > smmh[Y]){
            swap(smmh[GL], smmh[Y]);
            return GL;
    }
    else if(smmh[GR] < smmh[Y]) {
            swap(smmh[GR], smmh[Y]);
            return GR;
    }
    return Y;
}
```



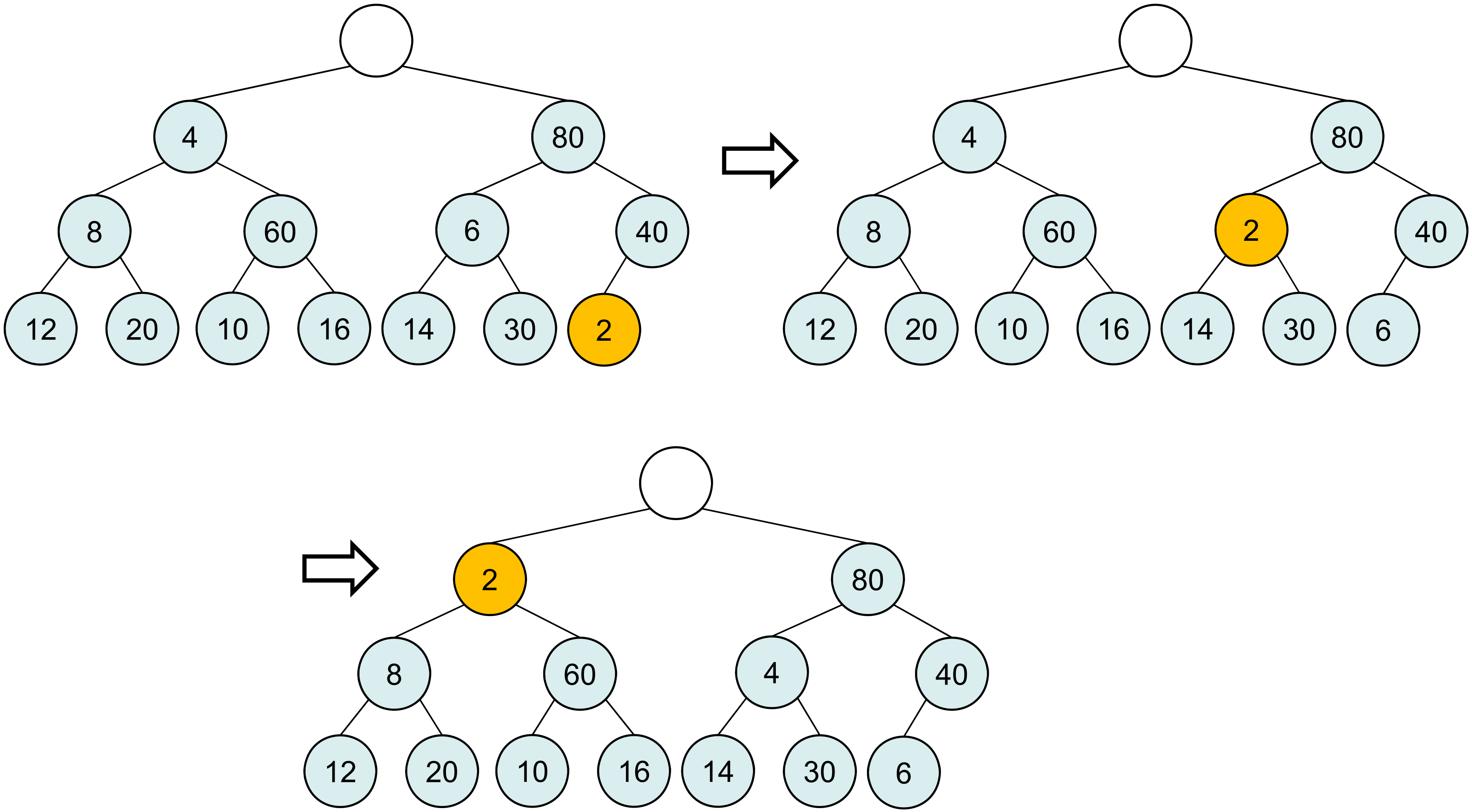
# SMMH v2.0

- Insertion
  1. Expand the size of the complete binary tree by one node. Assume that the id of new node is Y
  2. Y = AdjSibling(Y);
  3. X = AdjG(Y);
  4. If X == Y ➔ stop;
  5. Else Y ← X and repeat step 2

  Time complexity: O(log *n*)

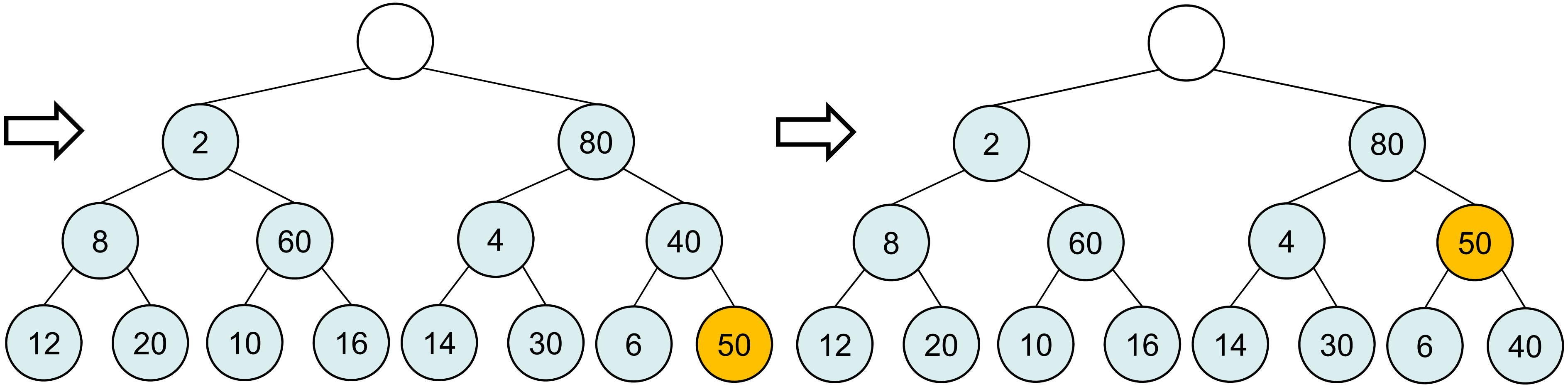
# SMMH v2.0

- Insertion example



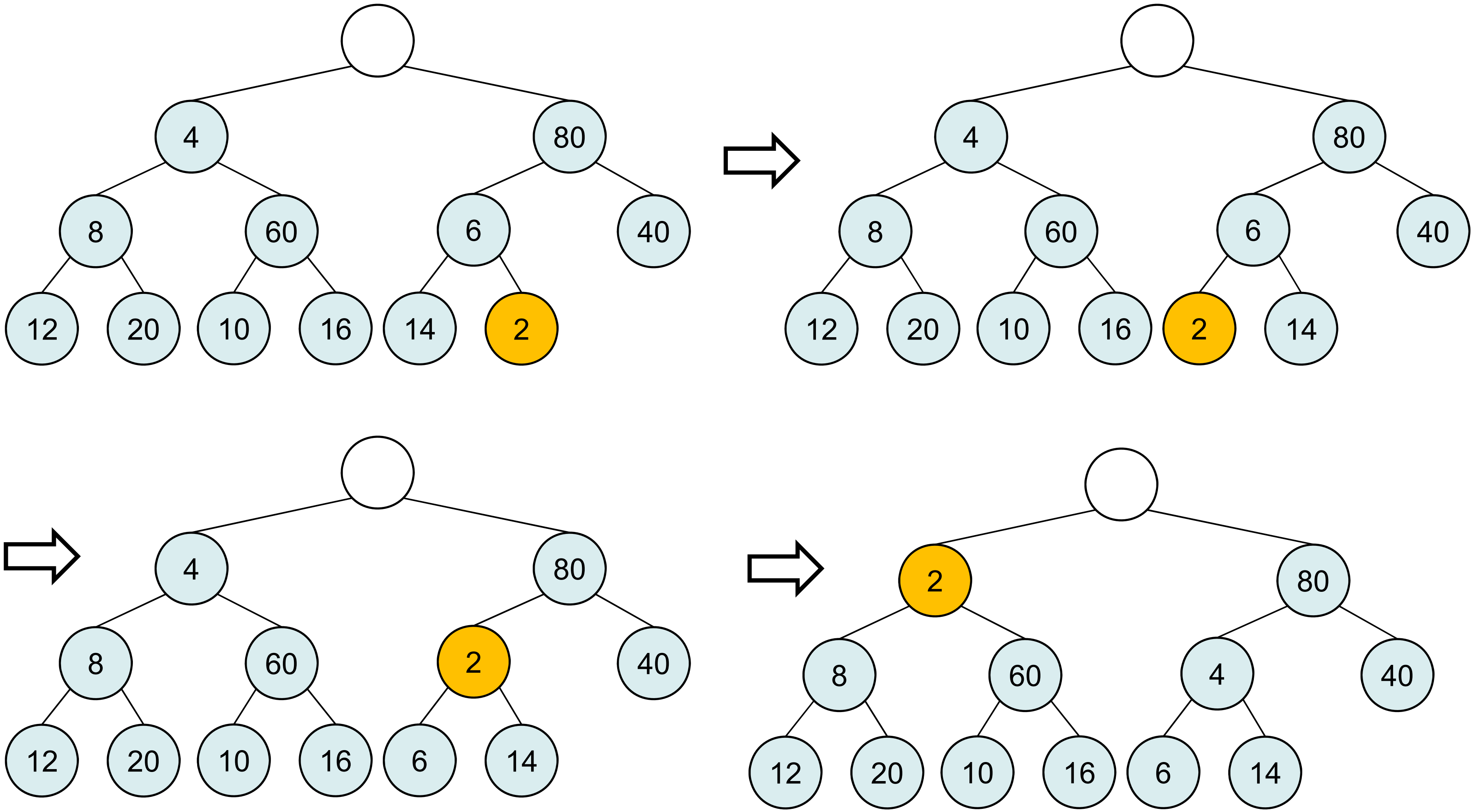
# SMMH v2.0

- Insertion example



# SMMH v2.0

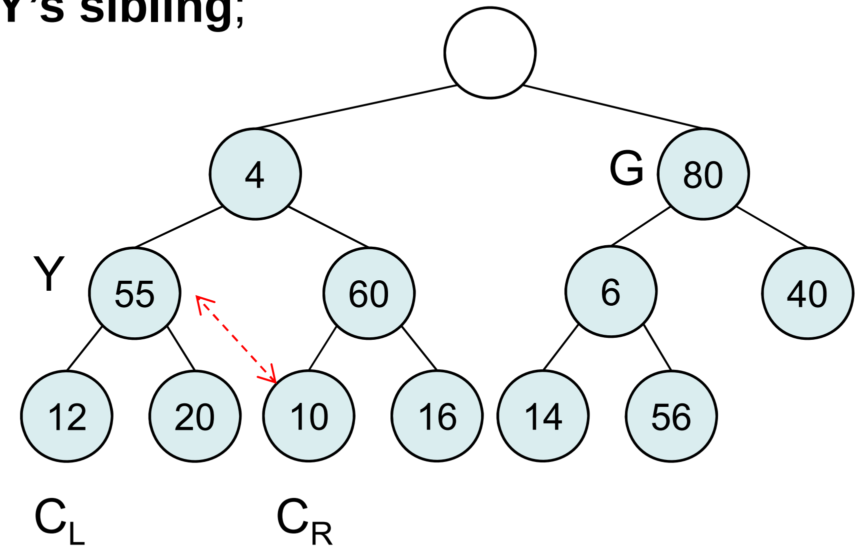
- Insertion example



# SMMH v2.0

- Grandchild adjustment

```
// Note that the index of root is 0;
int AdjGC(T* smmh, int Y, int MaxSize ){
    if (Y is a leaf) return Y;
    if( Y is a left child){
        int CL = Y's left child, CR = right child of Y's sibling;
        int C = CL;
        if(smmh[CR] < smmh[CL]) C = CR;
        if(smmh[C] < smmh[Y]){
            swap(smmh[C], smmh[Y]);
            return C;
        }
    }
    else{ // Y is a rightchild
    /* exercise */
    }
    return Y;
}
```



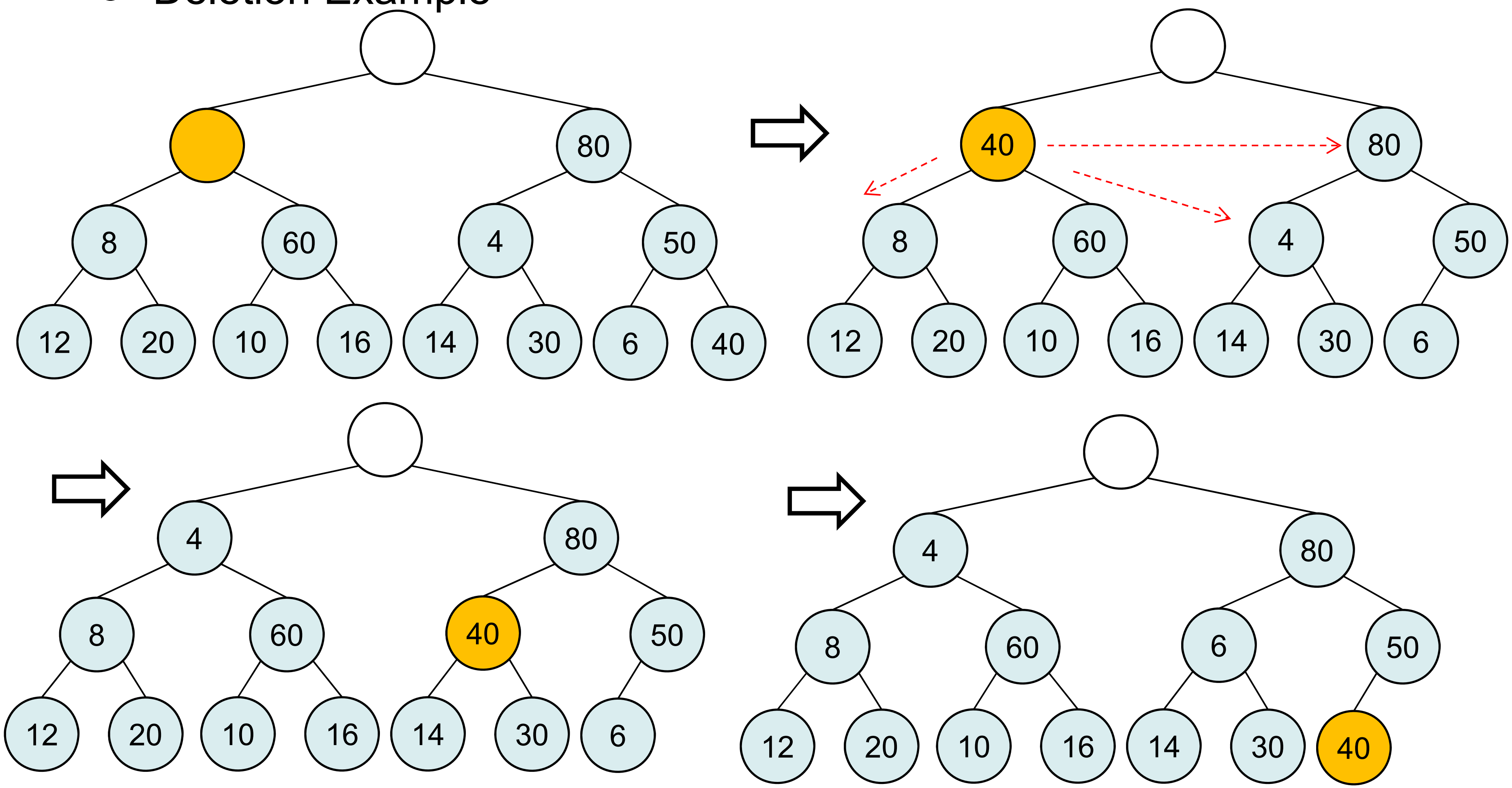
# SMMH v2.0

- Deletion
  1. Clear the target node, and move the last node to the target node
  2. AdjSibling(Y);
  3. X = AdjGC(Y);
  4. If X == Y ➔ stop;
  5. Else  Y = X and repeat step 2

  Time complexity: O(log *n*)

# SMMH v2.0

- Deletion Example



# SMMH v2.0

- Deletion Example