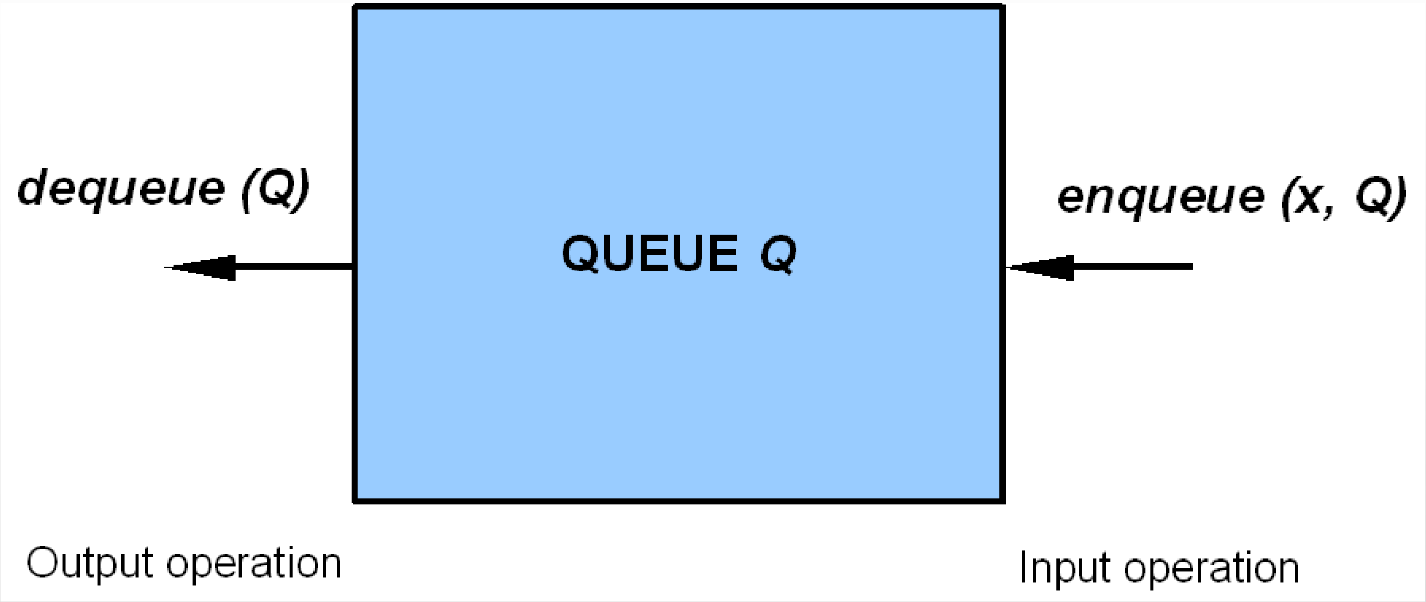
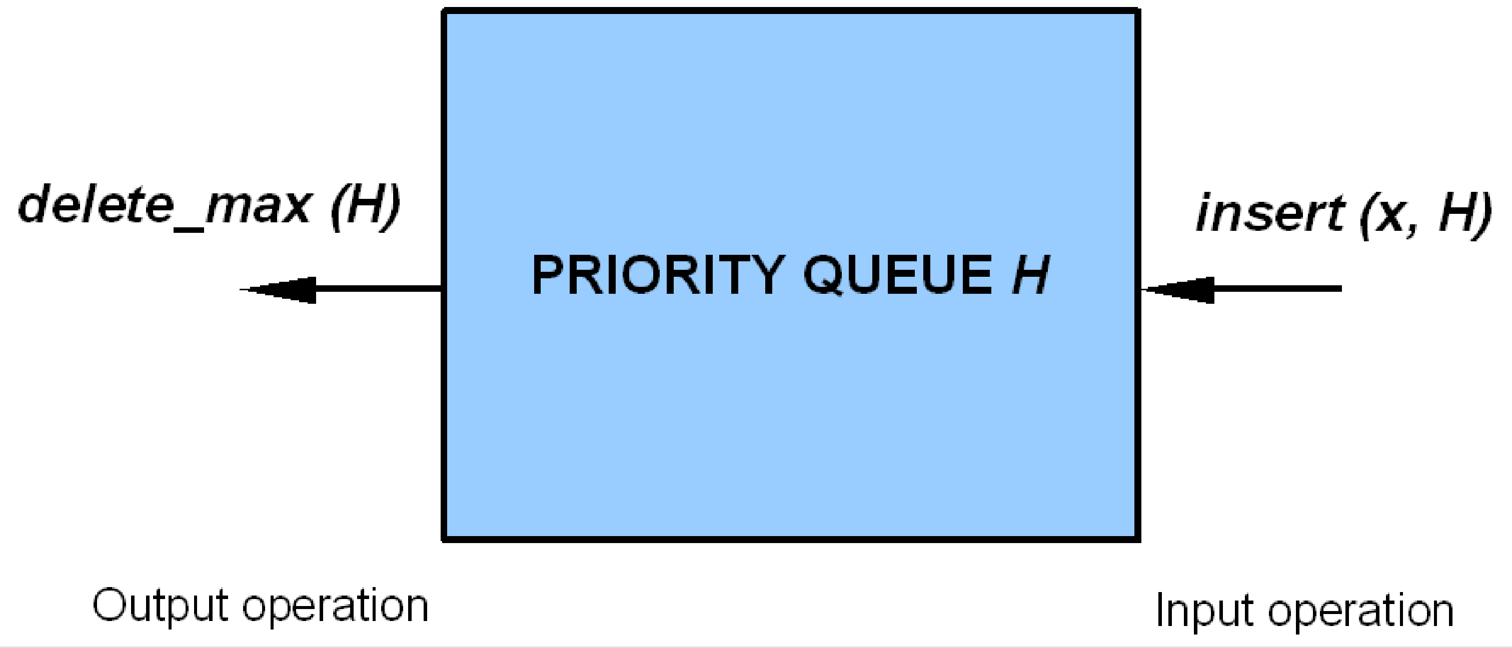
6. Heaps

Priority Queue

- Recall Queue ADT Model



Priority Queue Model



Priority Queue

Basic Operations:

- Insert a key
- Delete the maximum

Possible Implementations:

- Linked List
- BST
- Heap

Binary Heap

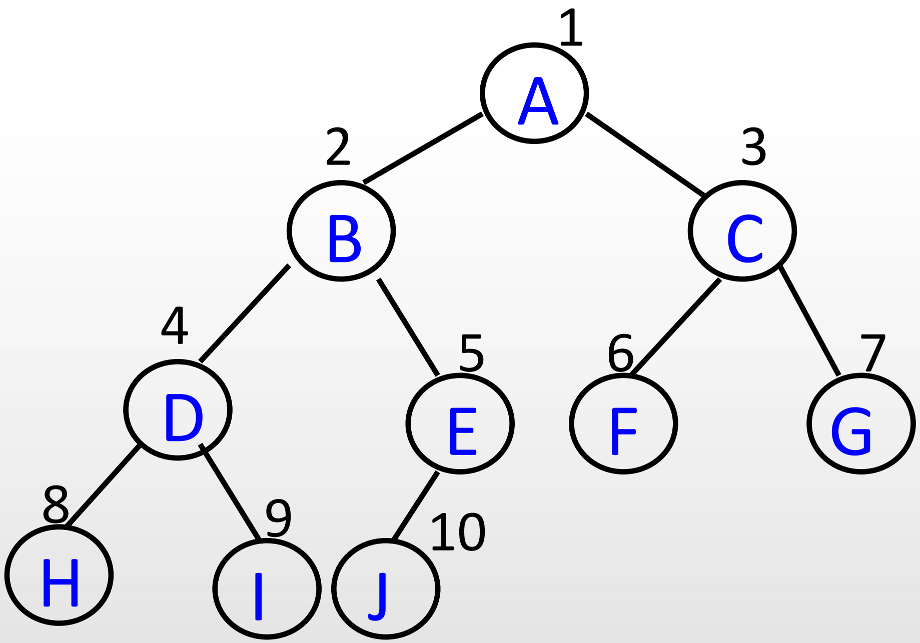
Two important properties:

1. Structure property

   - A heap is a complete binary tree (i.e. completely filled, with the possible exception of the bottom level, which is filled from left to right)

   - Since a complete binary tree is regular, it can be represented in an array and no pointers are necessary.

Complete Binary Tree



For any element in array[i], the left child is in position 2i, the right child is in 2i+1, and the parent is in $\lfloor i/2 \rfloor$.

	A	B	C	D	E	F	G	H	I	J		
0	1	2	3	4	5	6	7	8	9	10	11	12

Priority Queue

```
typedef struct node{
   int max_heap_size;
   int size;
   int *elements;
}heap;
```

Priority Queue

```
heap *create(int max){
  heap *h;

   h = (heap *)malloc(sizeof(heap));
   if(h==NULL)
       error(“Out of space!”);

   h->elements = (int *)malloc(sizeof(int)*(max+1));
   if(h->elements==NULL)
       error(“Out of space!”);

   h->max_heap_size = max;
   h->size = 0;

   return h;
}
```

Binary Heap

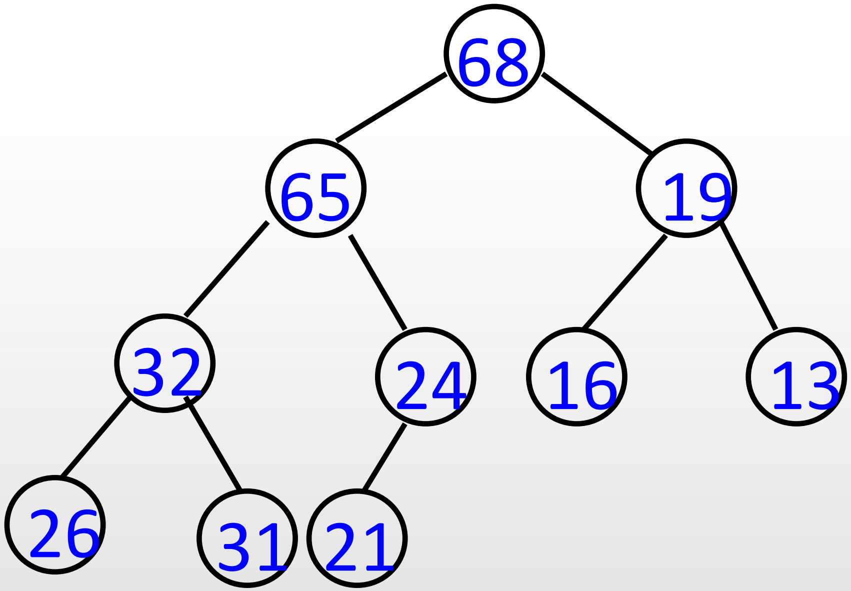
Two important properties:

2. Heap order property

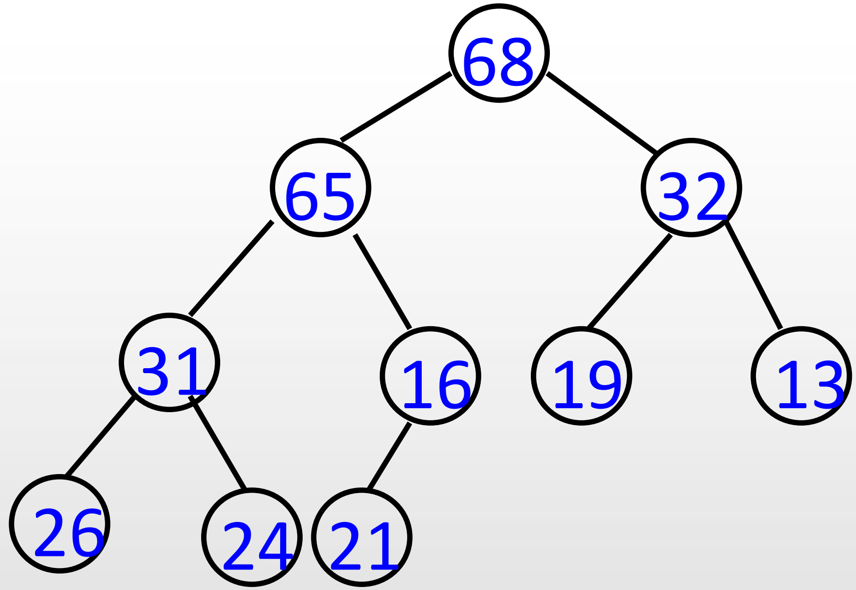
   - In a heap, for every node X, the key in the parent of X is larger than the key in X, except the root.

   - By this property, the maximum element can always be found at the root.

Heap



A



B

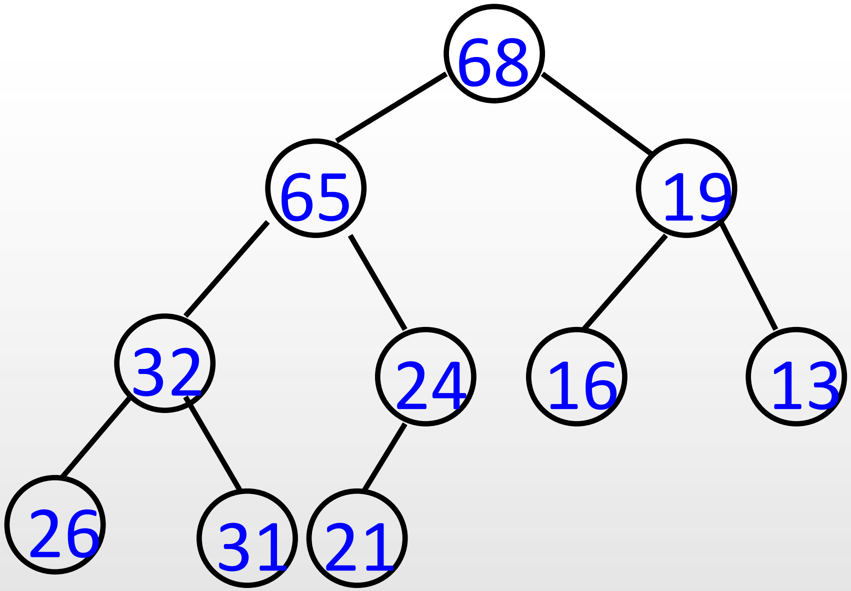
Binary Heap

Operations:

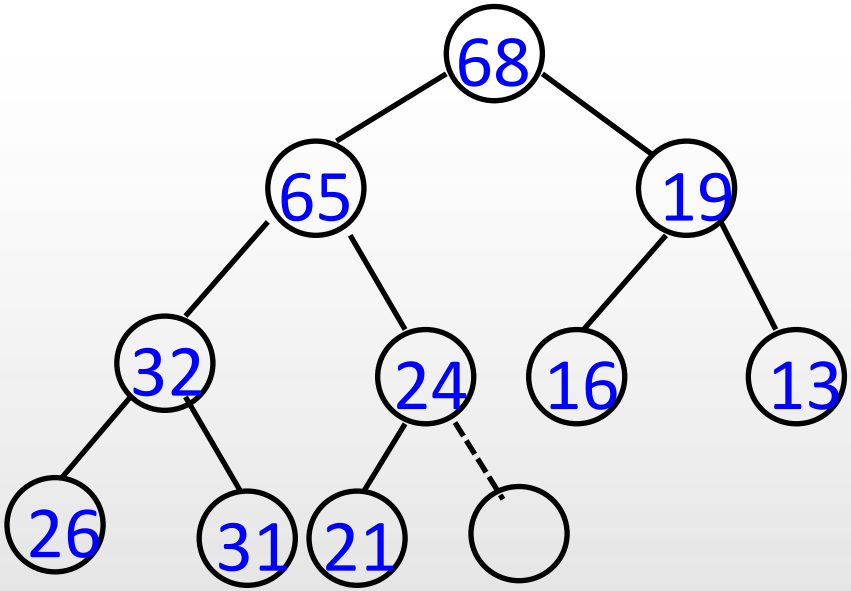
- Insert
- Delete max

* Build heap

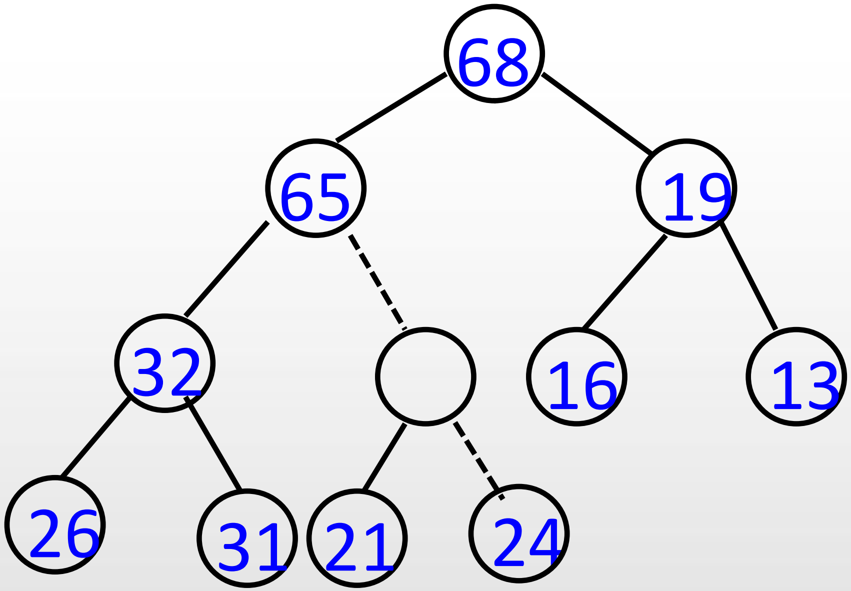
Heap - Insert



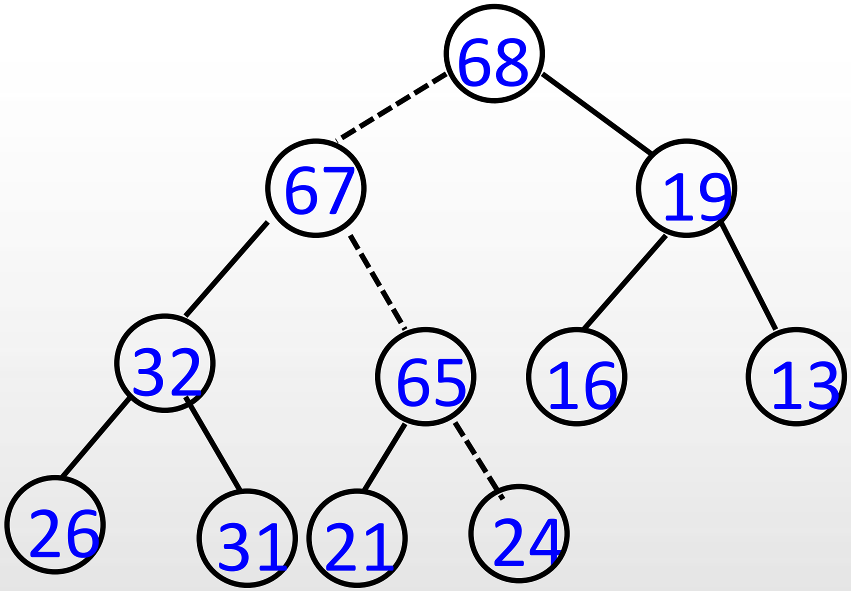
Heap – Insert 67



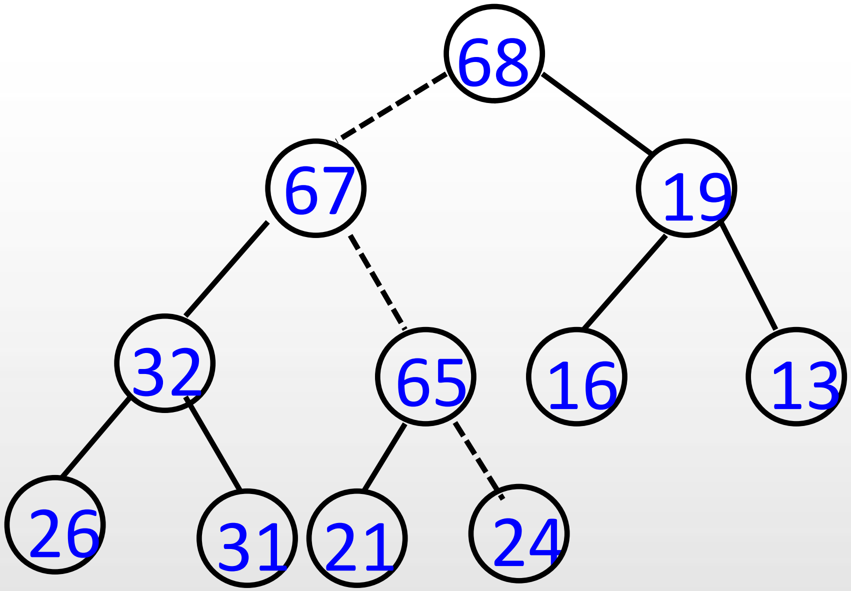
Heap – Insert 67



Heap – Insert 67



Heap – Insert 67



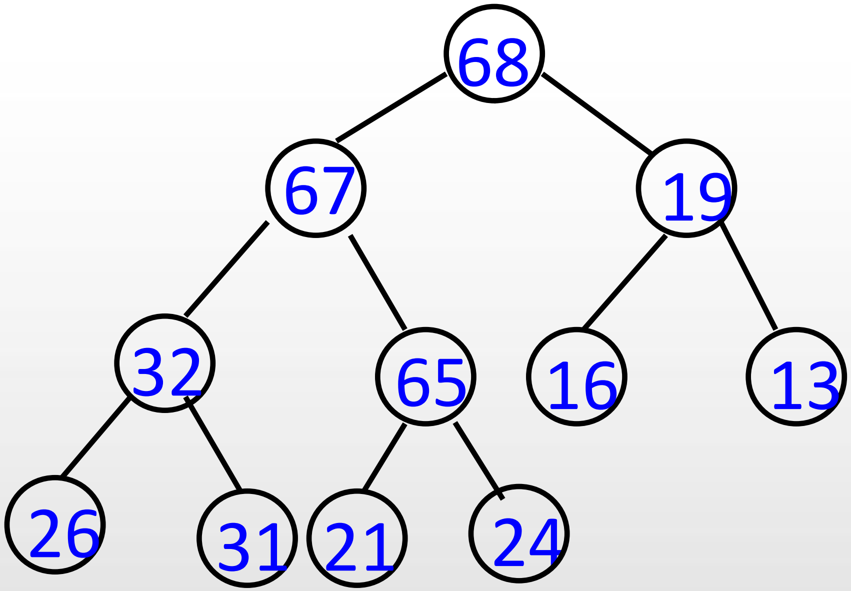
- Strategy: "percolate up"

Priority Queue

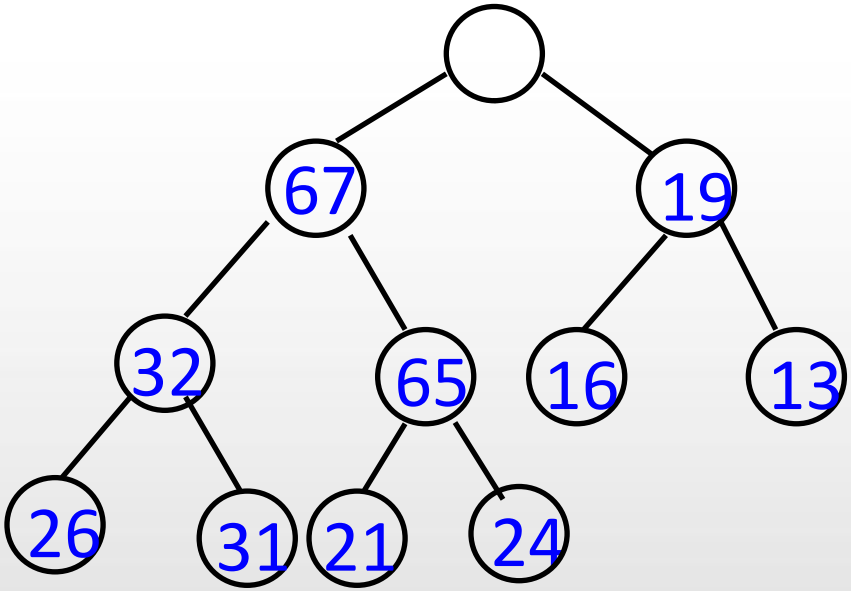
```
void insert(int x, heap *h){
  int i;

  if(is_full(h))
     error("Priority Queue is full!");
  else{
     i = ++h->size;
     while(h->elements[i/2]<x){
          h->elements[i]=h->elements[i/2];
          i/=2;
     }
     h->elements[i]=x;
  }
}
```

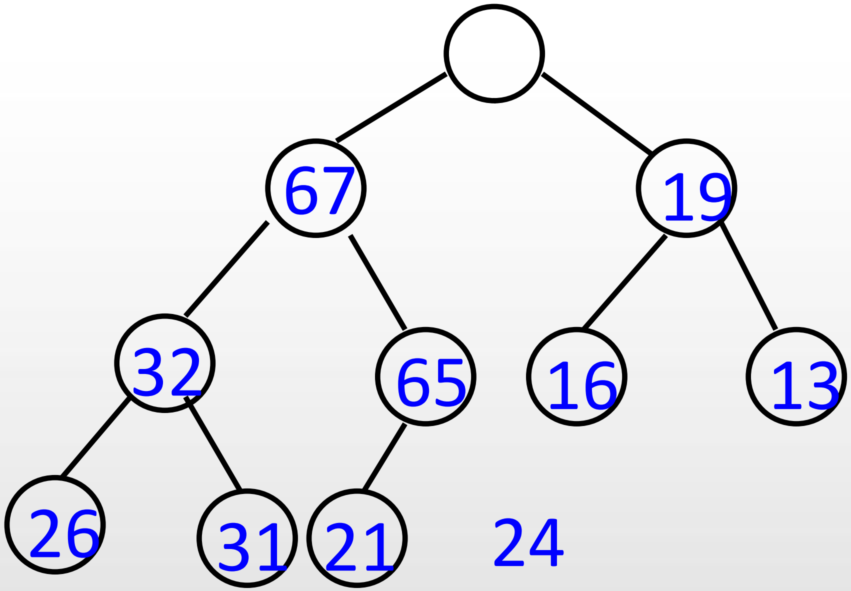
Heap – Delete max



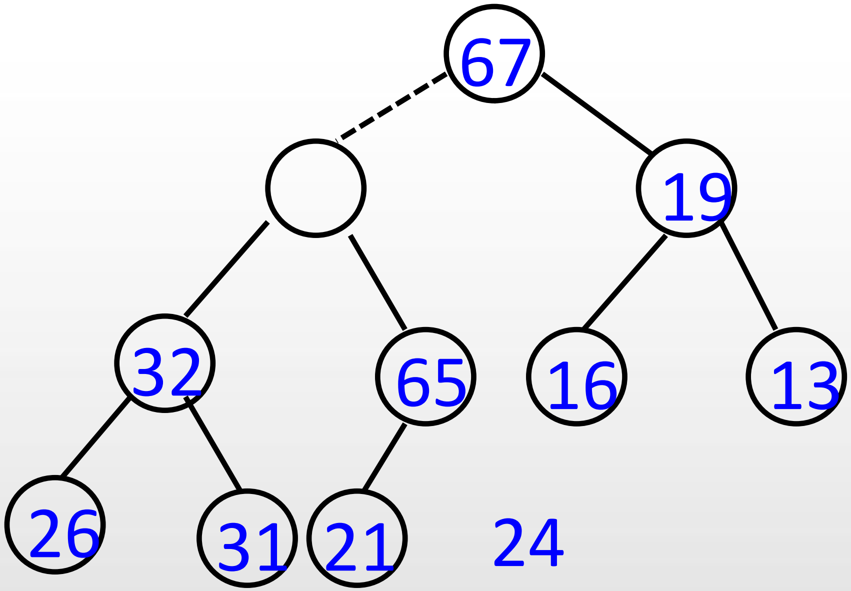
Heap – Delete max



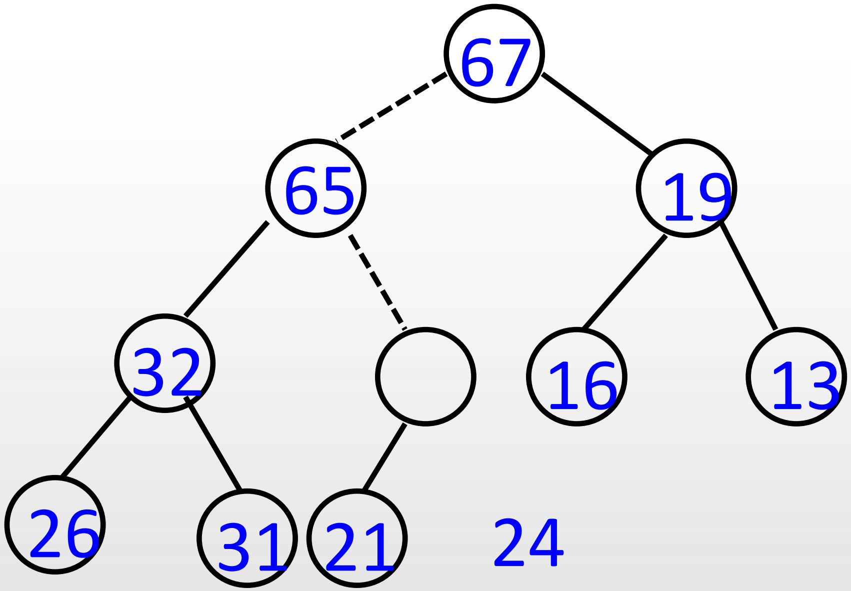
Heap – Delete max



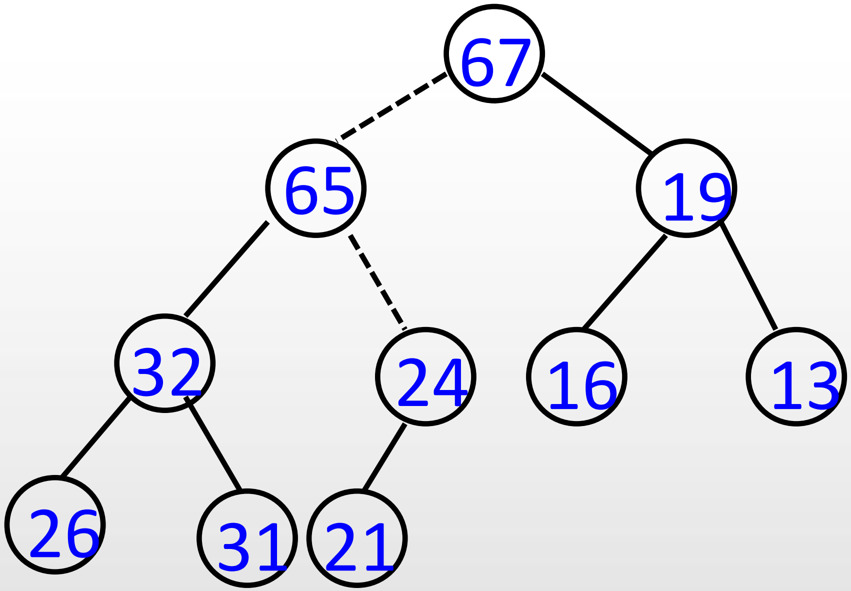
Heap – Delete max



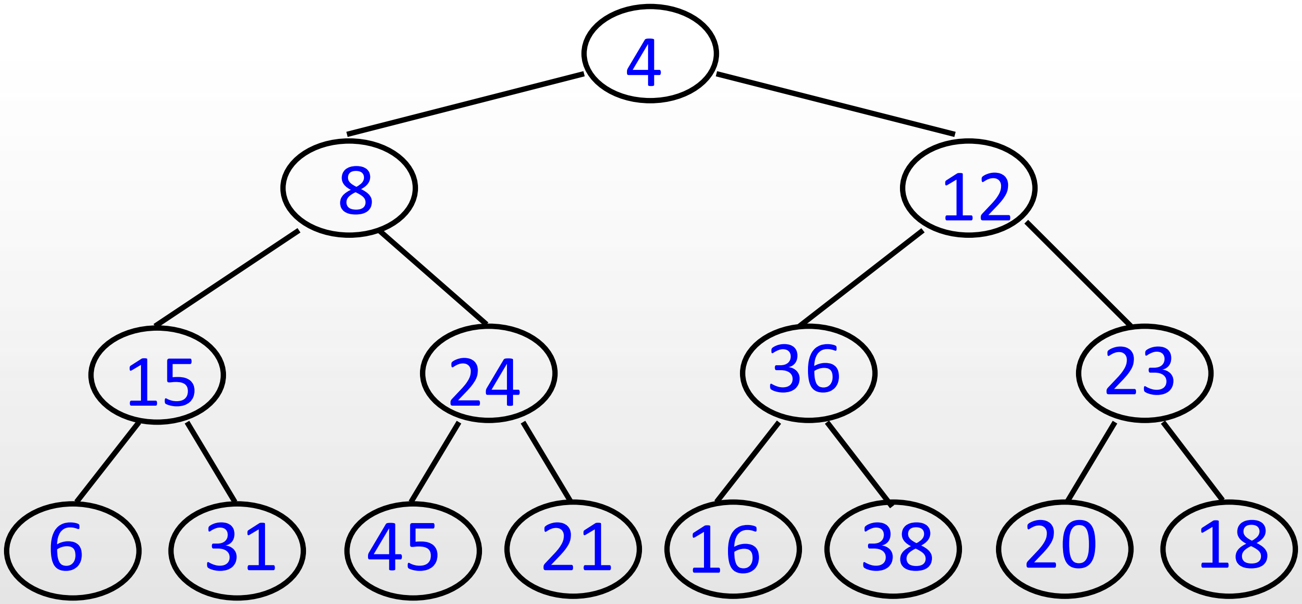
Heap – Delete max



Heap – Delete max

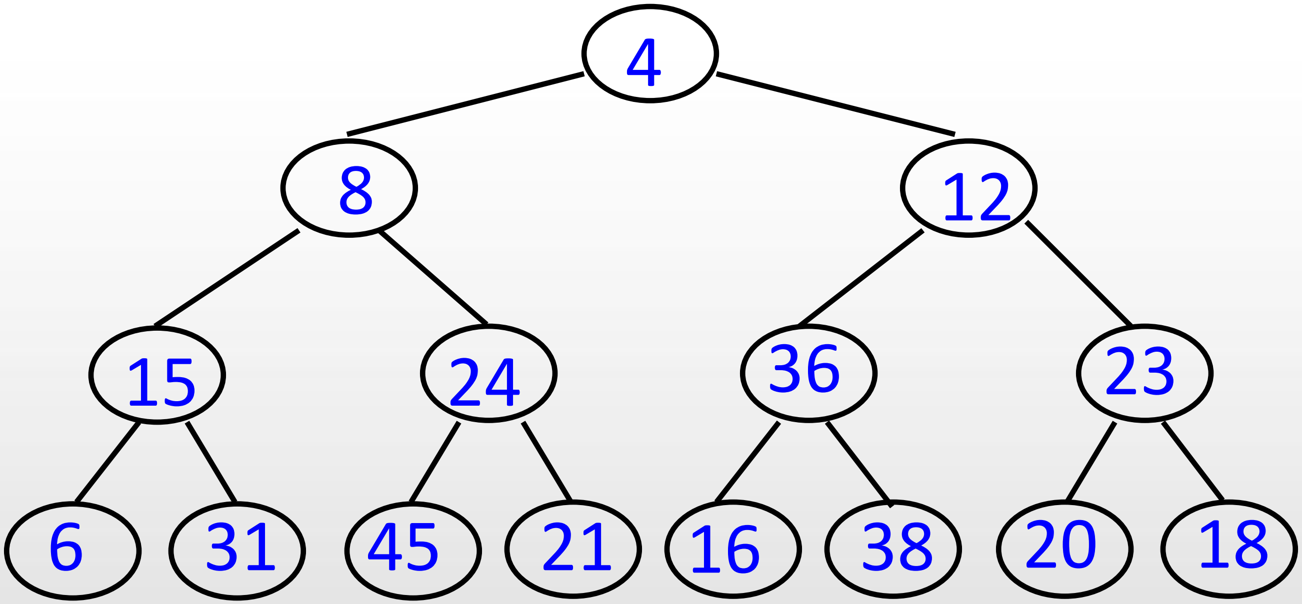


Build heap



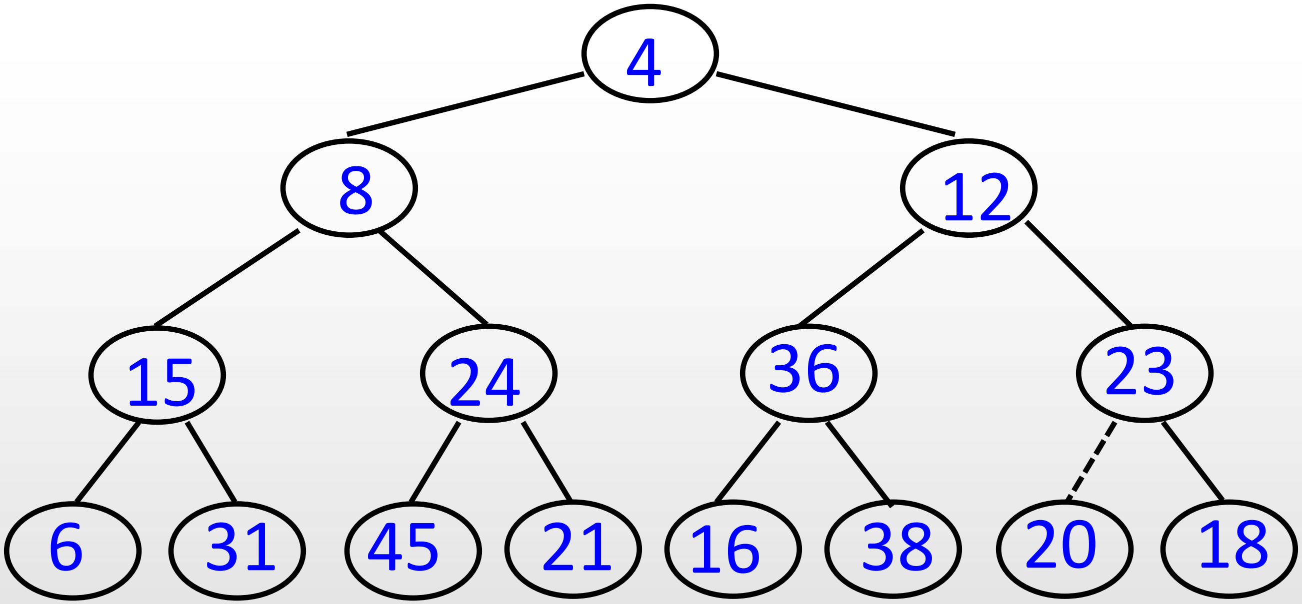
- n successive insertions ≈ O(n log n)

Build heap



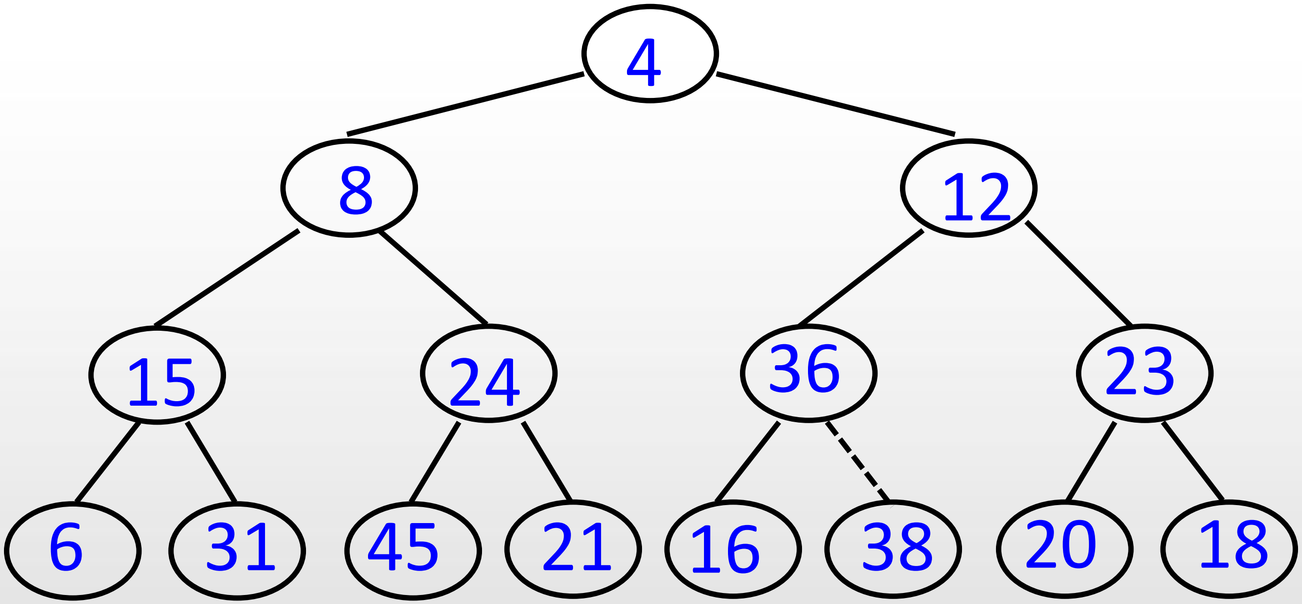
- Strategy: "percolate down" from n/2

Build heap



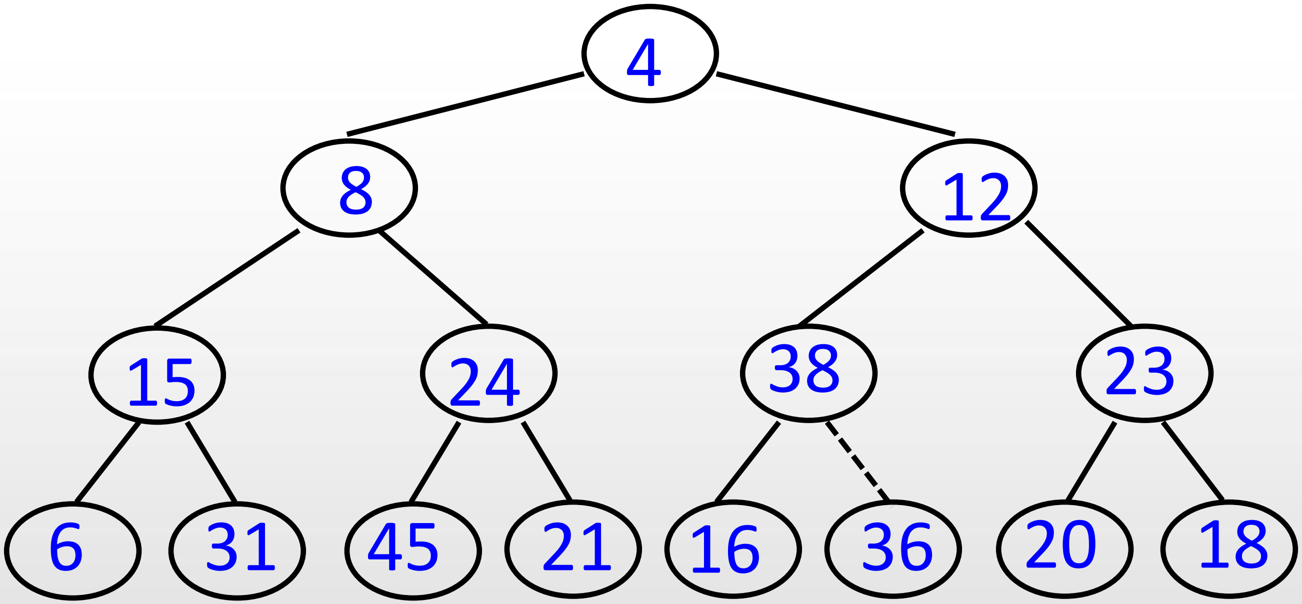
- Strategy: percolate down(7)

Build heap



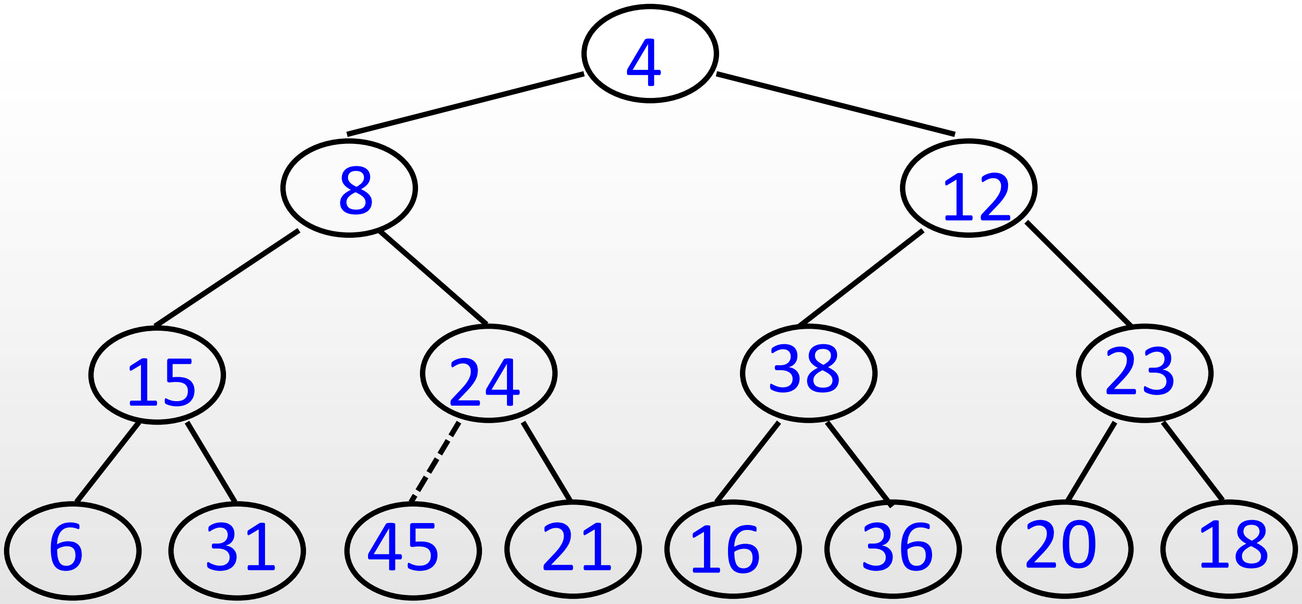
- Strategy: percolate down(6)

Build heap



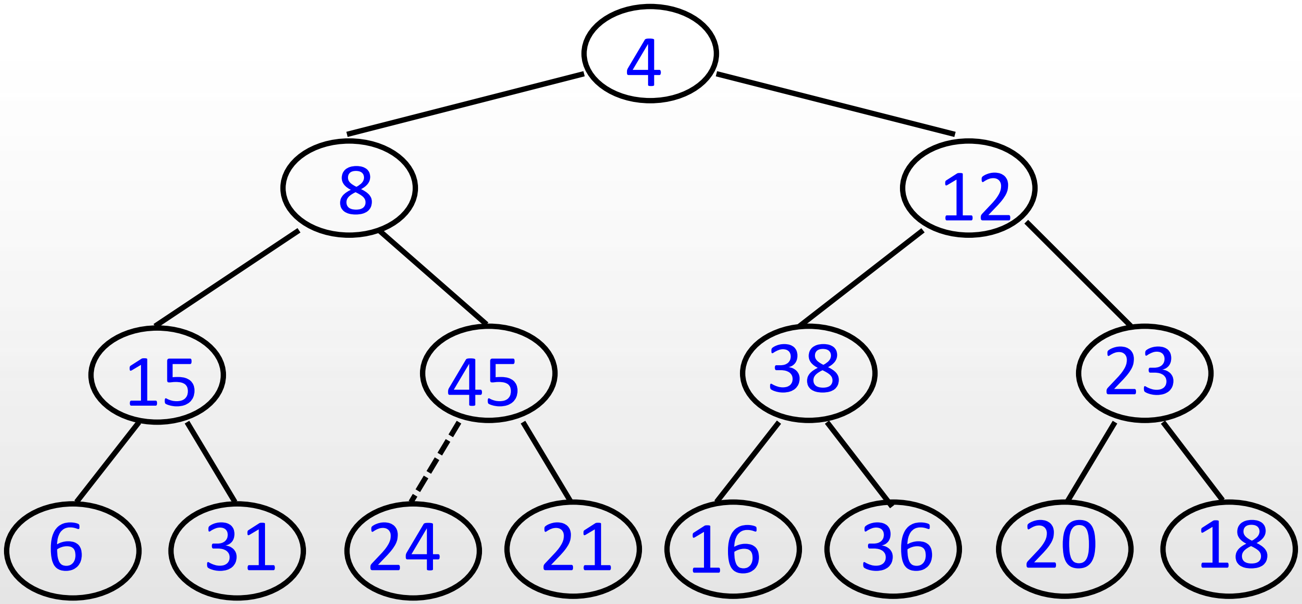
- Strategy: percolate down(6)

Build heap



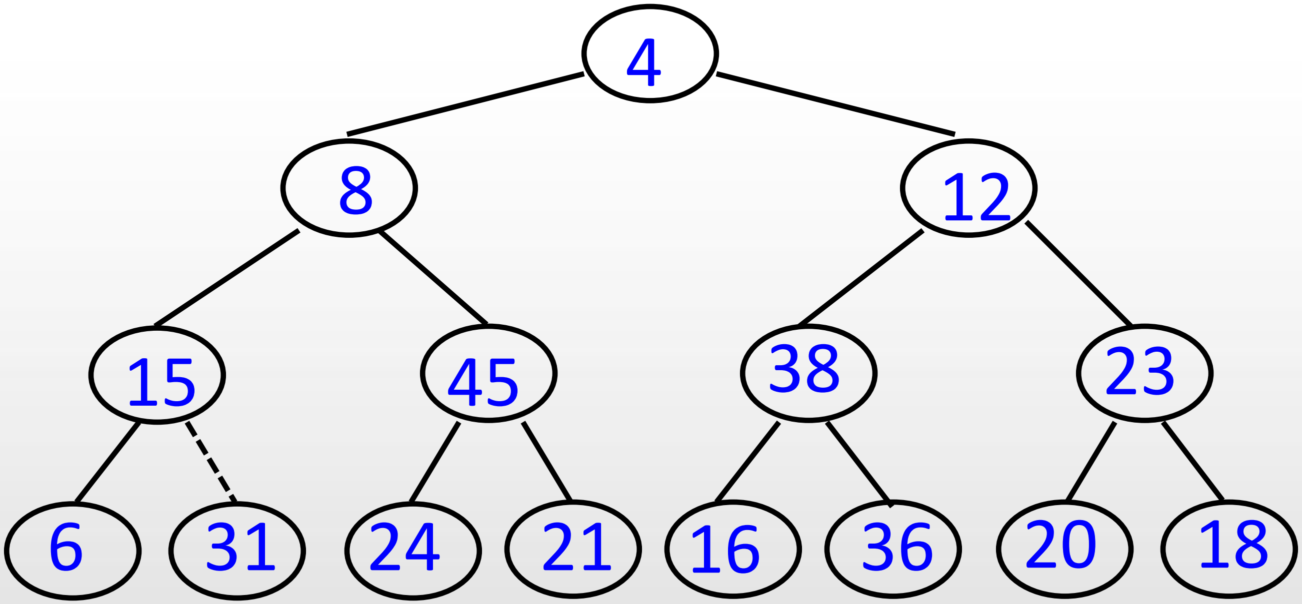
- Strategy: percolate down(5)

Build heap



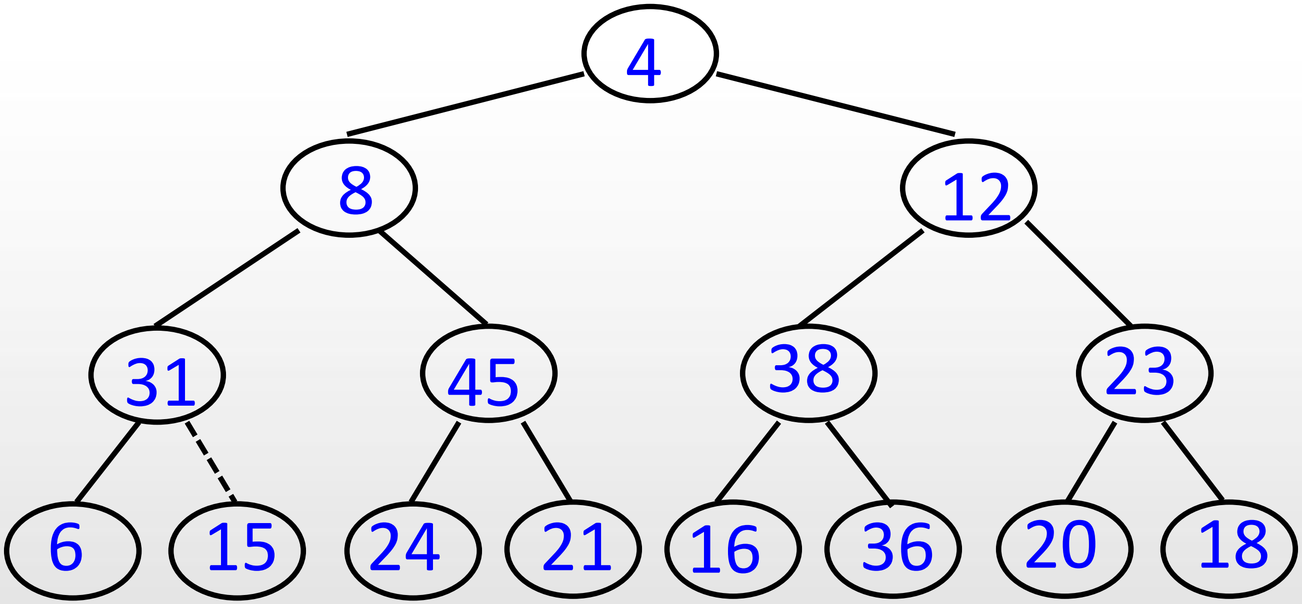
- Strategy: percolate down(5)

Build heap



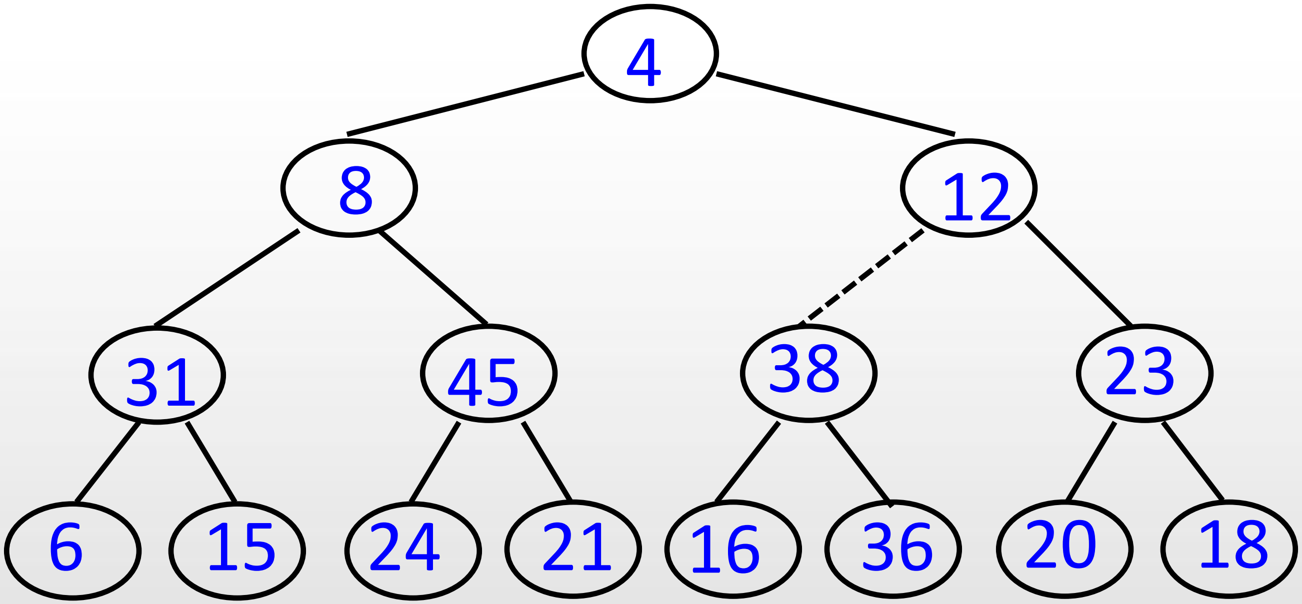
- Strategy: percolate down(4)

Build heap



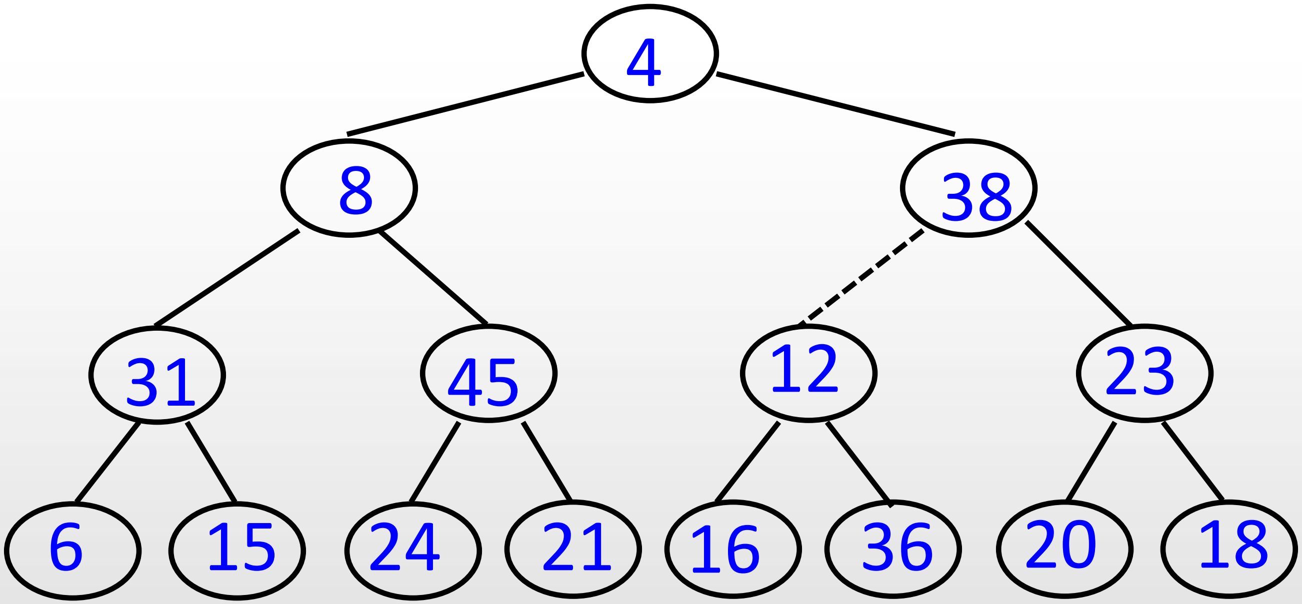
- Strategy: percolate down(4)

Build heap



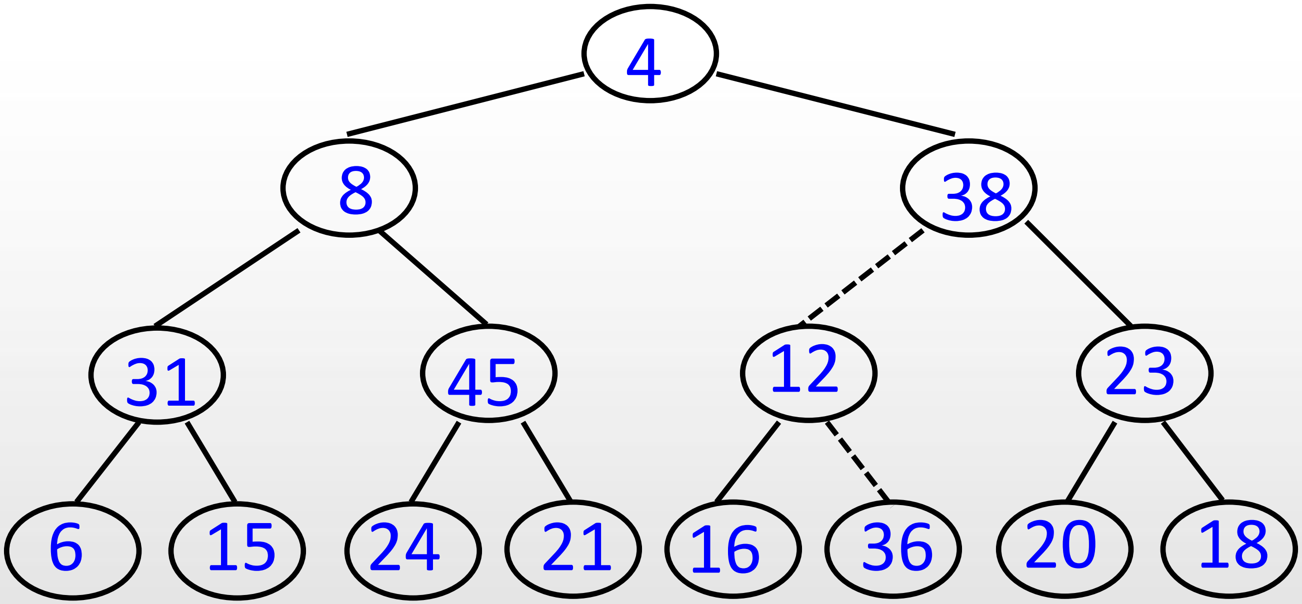
- Strategy: percolate down(3)

Build heap



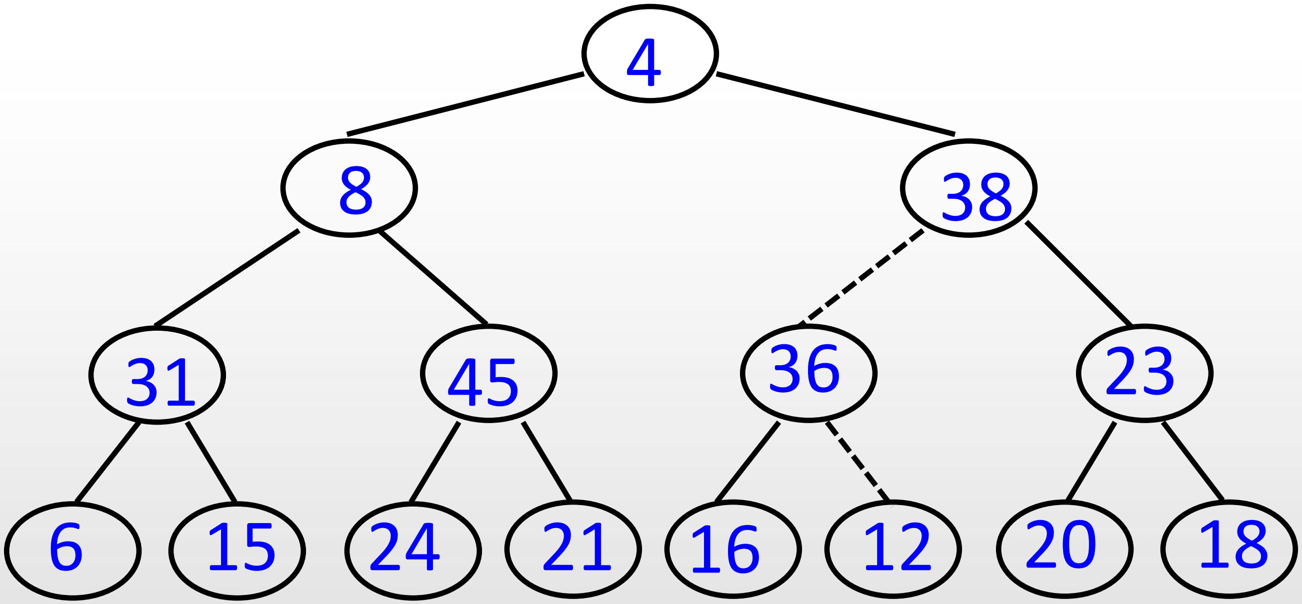
- Strategy: percolate down(3)

Build heap



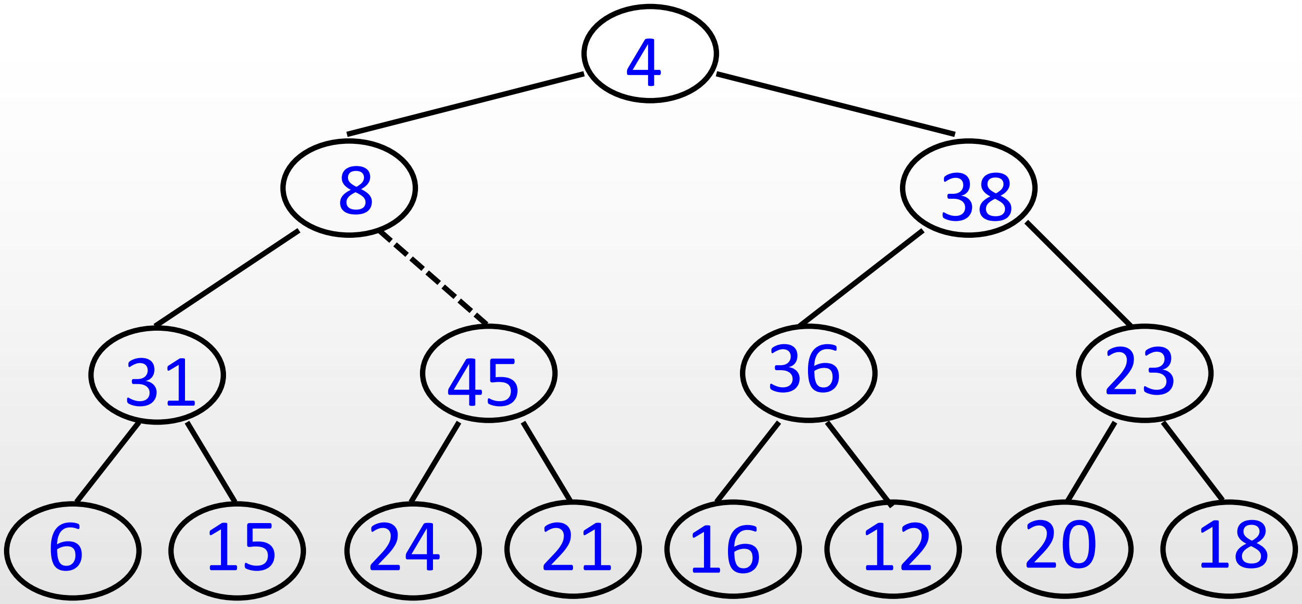
- Strategy: percolate down(3)

Build heap



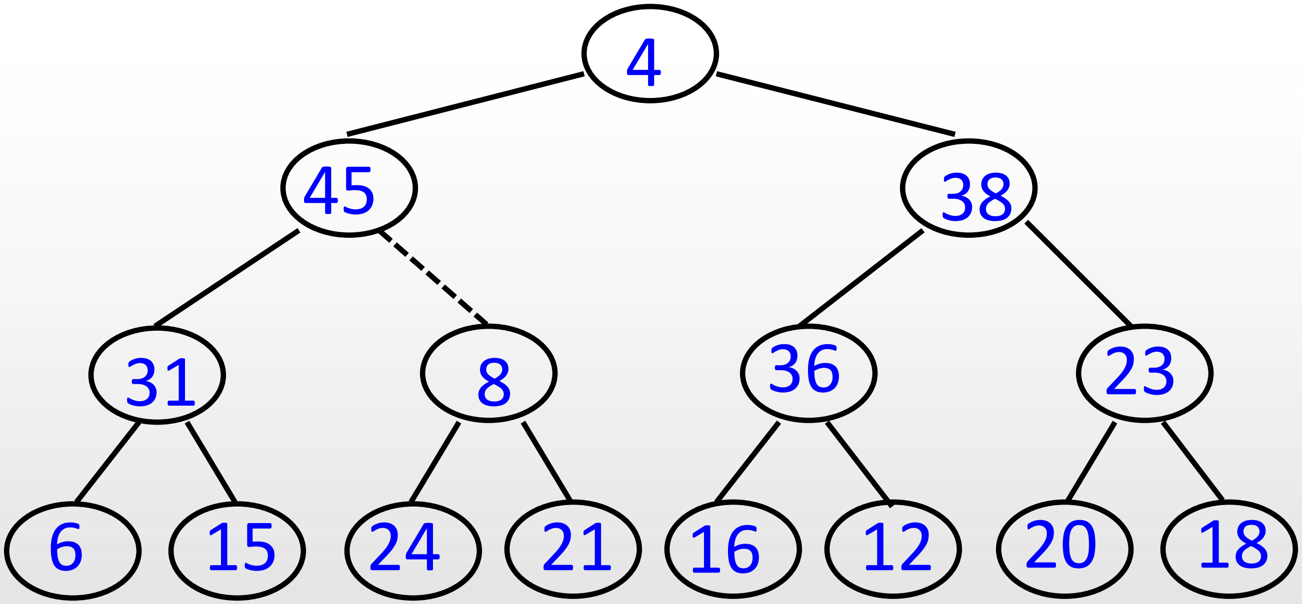
- Strategy: percolate down(3)

Build heap



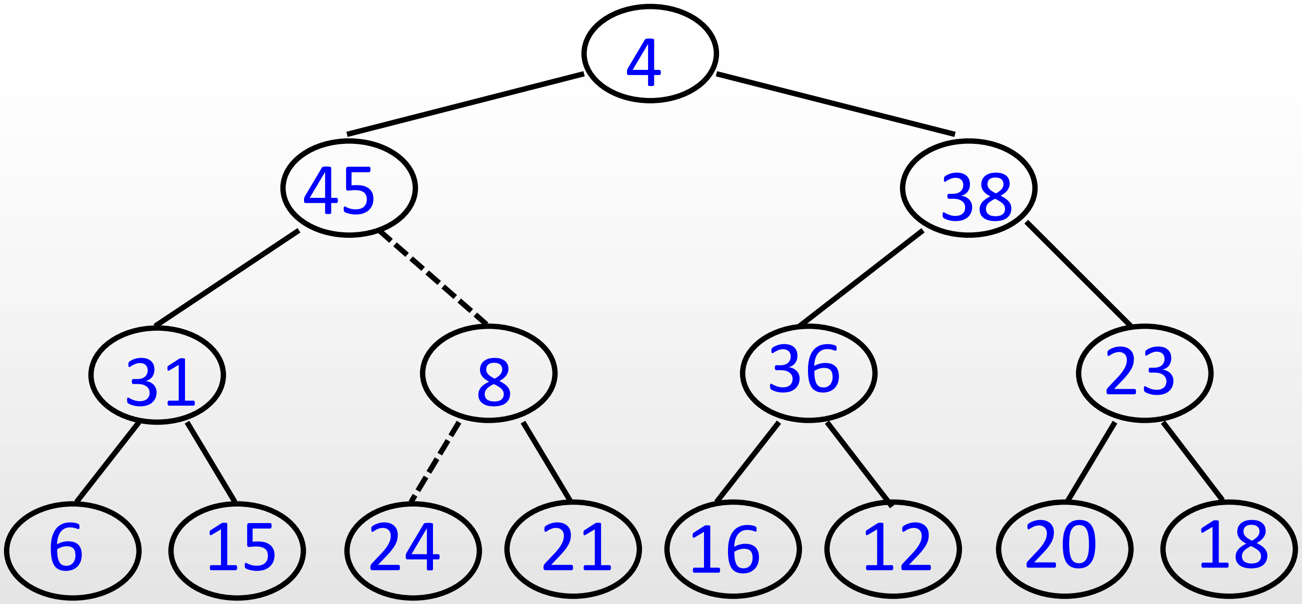
- Strategy: percolate down(2)

Build heap



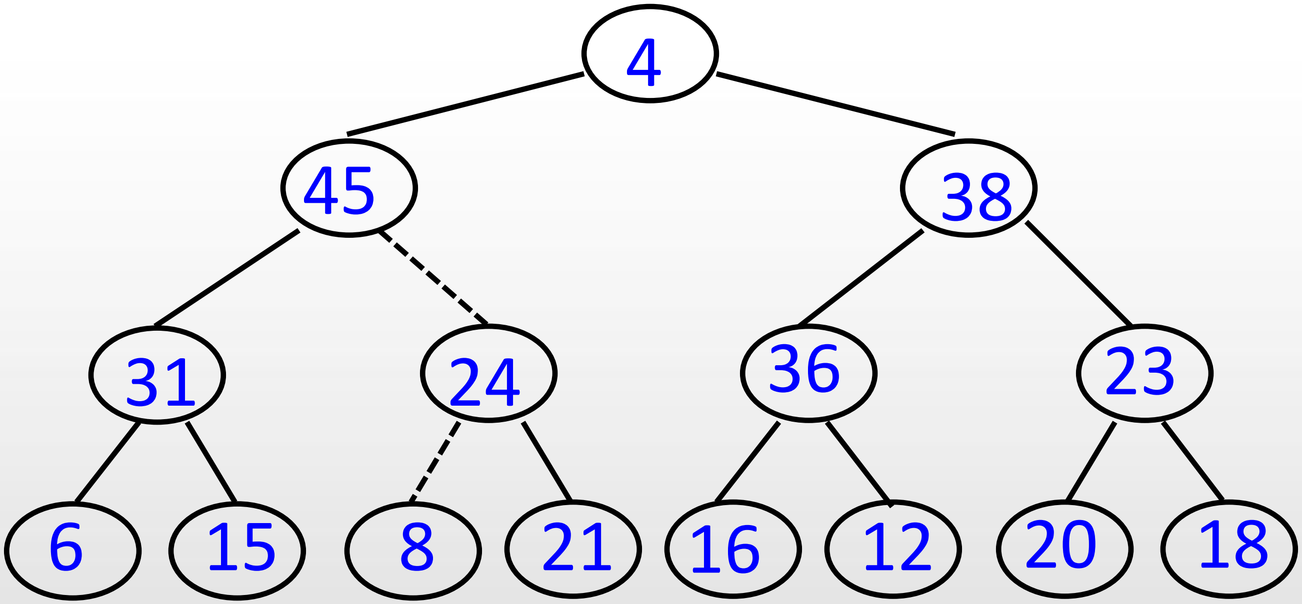
- Strategy: percolate down(2)

Build heap



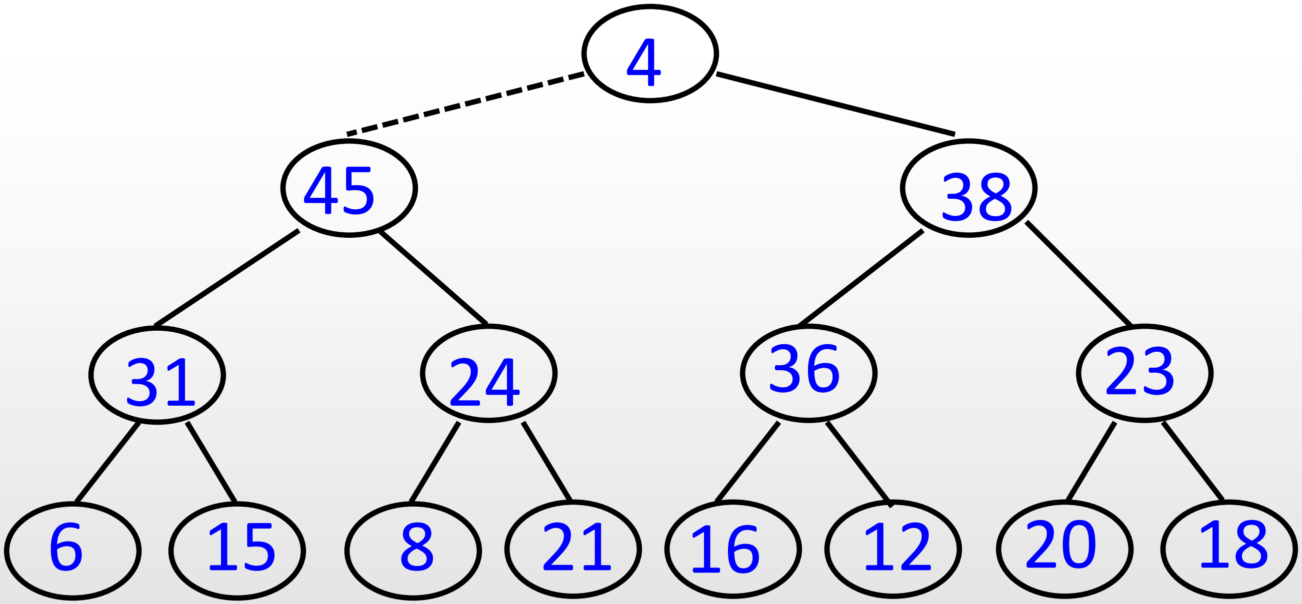
- Strategy: percolate down(2)

Build heap



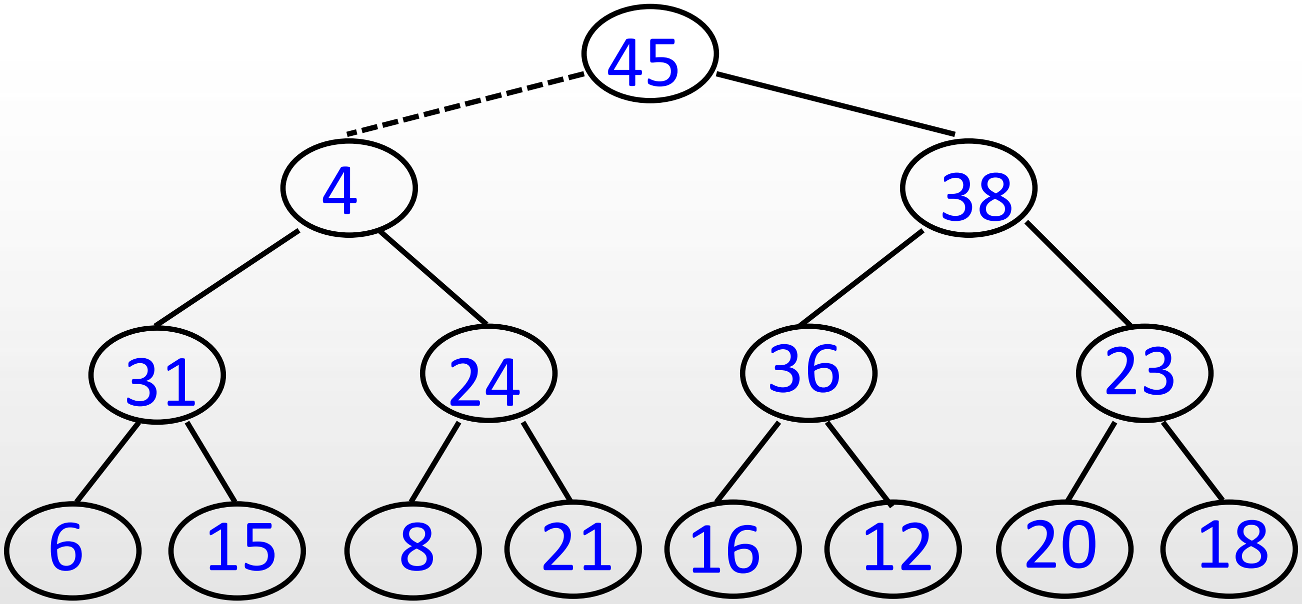
- Strategy: percolate down(2)

Build heap



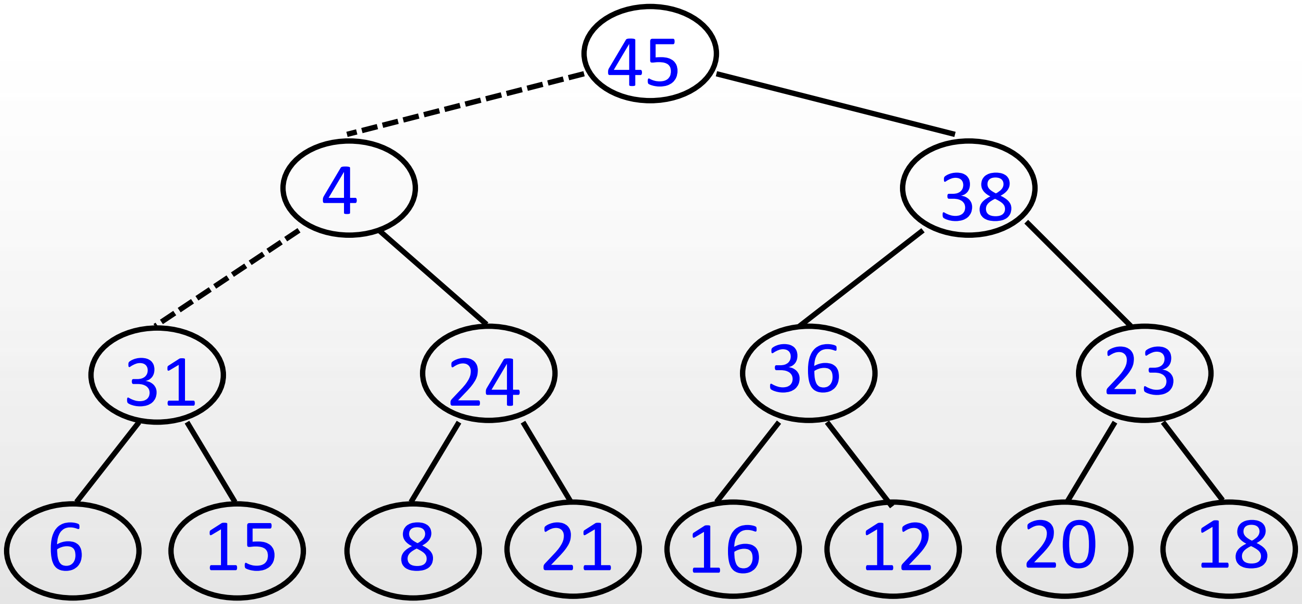
- Strategy: percolate down(1)

Build heap



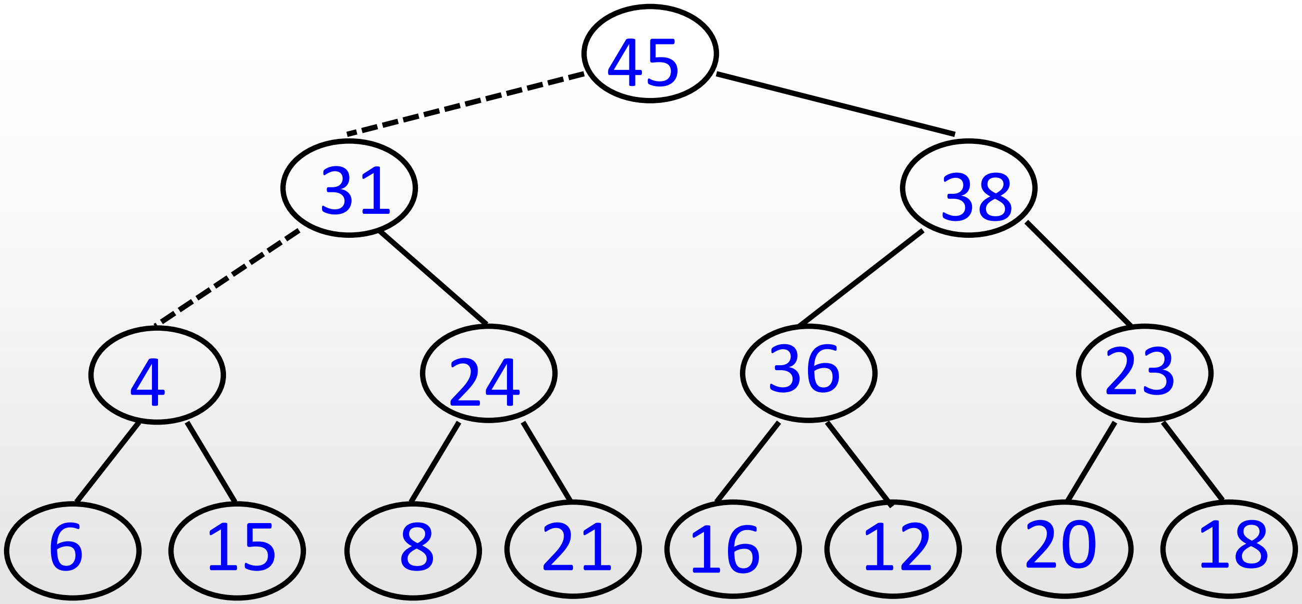
- Strategy: percolate down(1)

Build heap



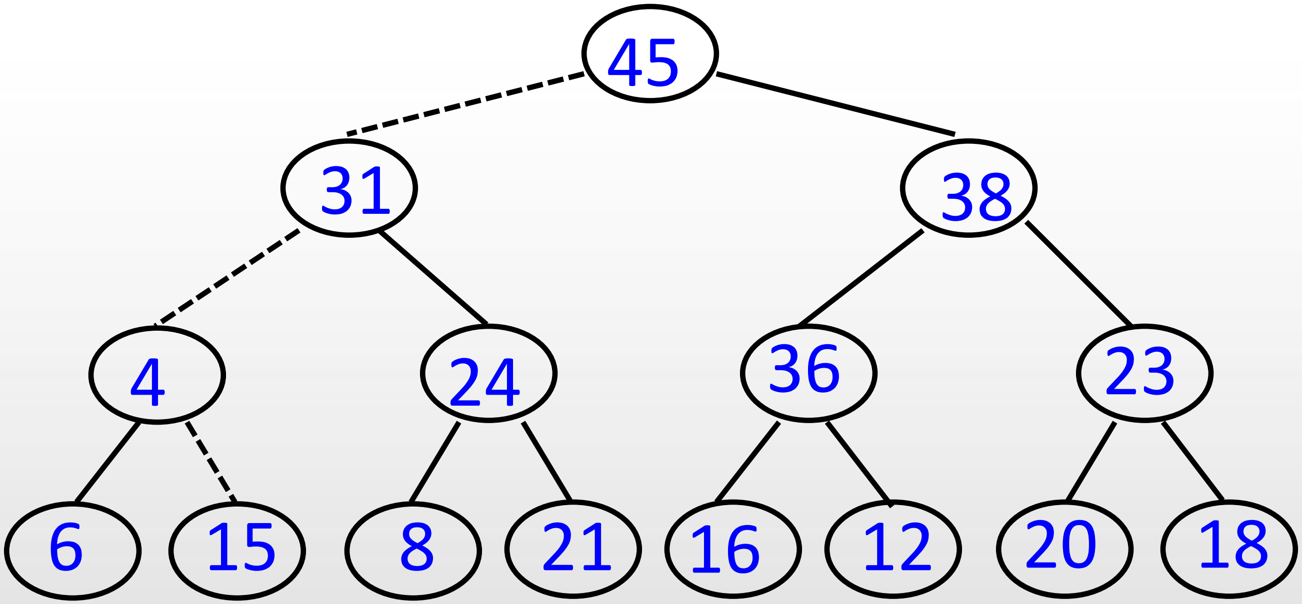
- Strategy: percolate down(1)

Build heap



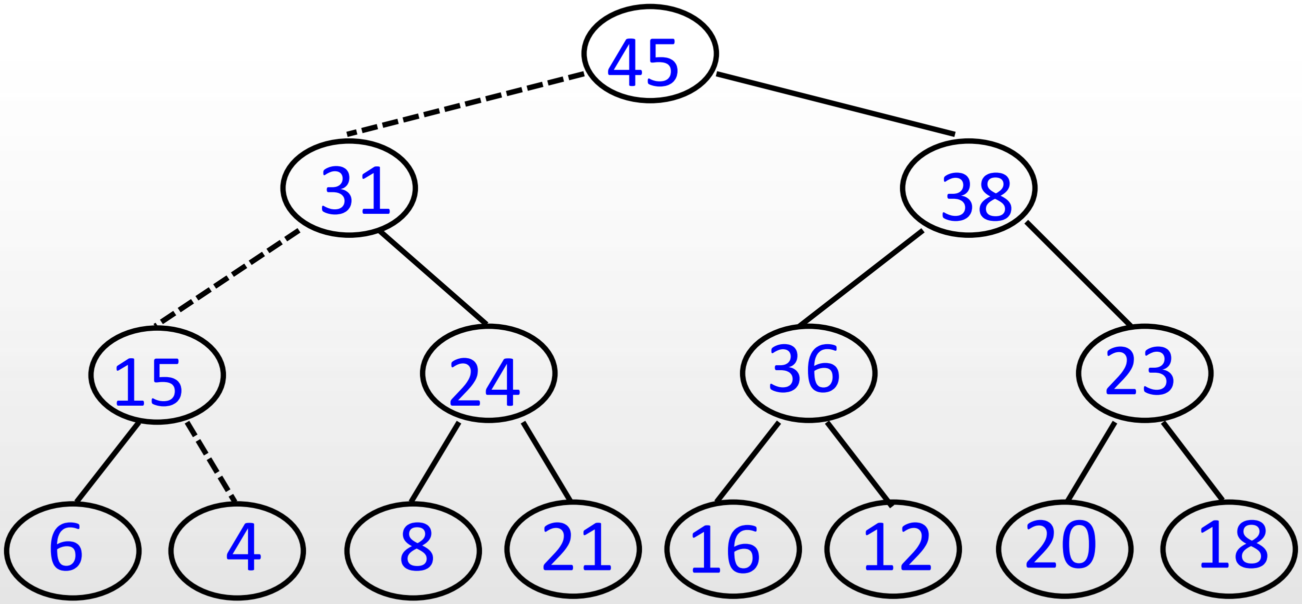
- Strategy: percolate down(1)

Build heap



- Strategy: percolate down(1)

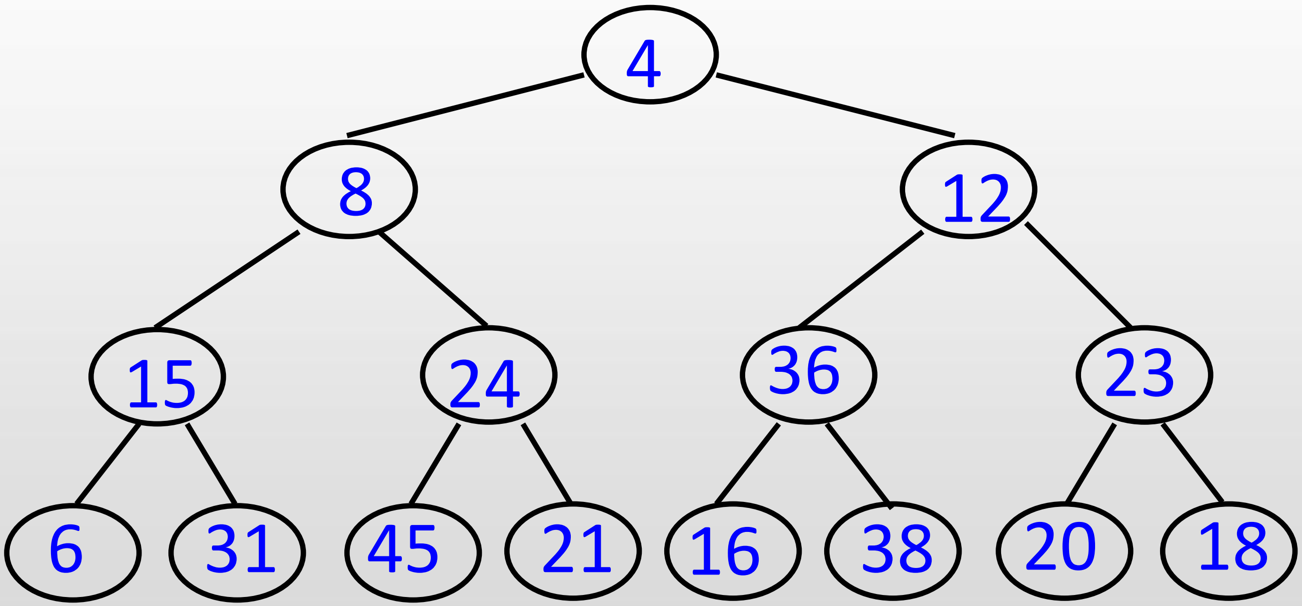
Build heap



- Strategy: percolate down(1)

Build heap

```
for(i=n/2;i>0;i--)
    percolate_down(i);
```

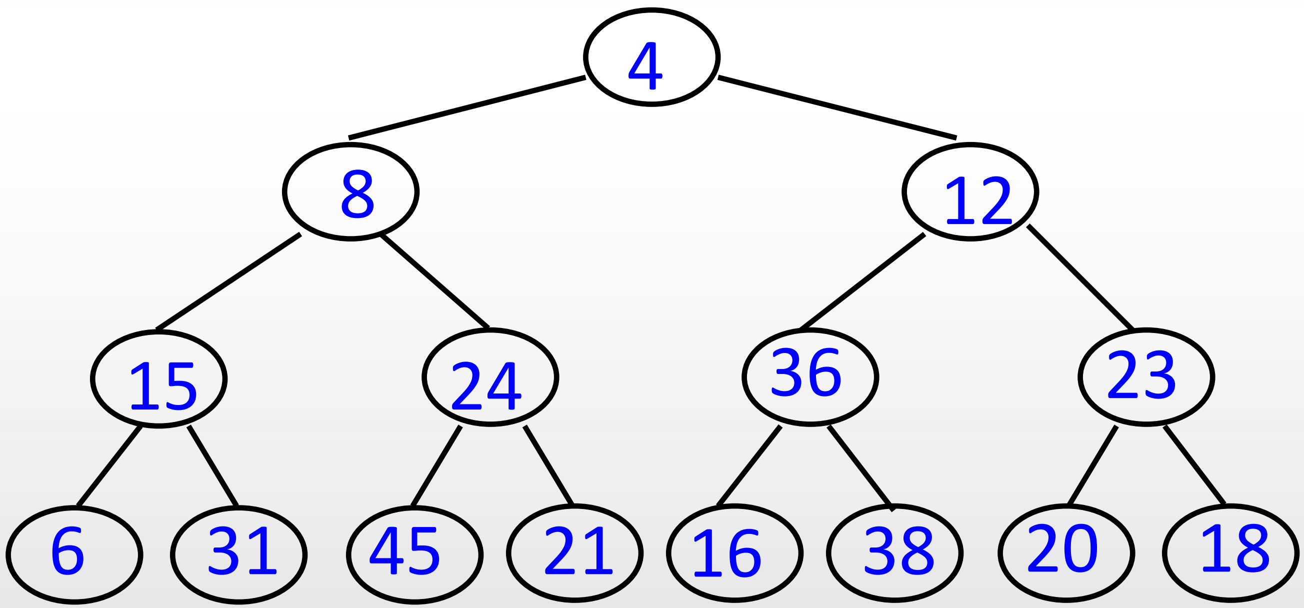


Build heap

```
for(i=n/2;i>0;i--)
    percolate_down(i);
```

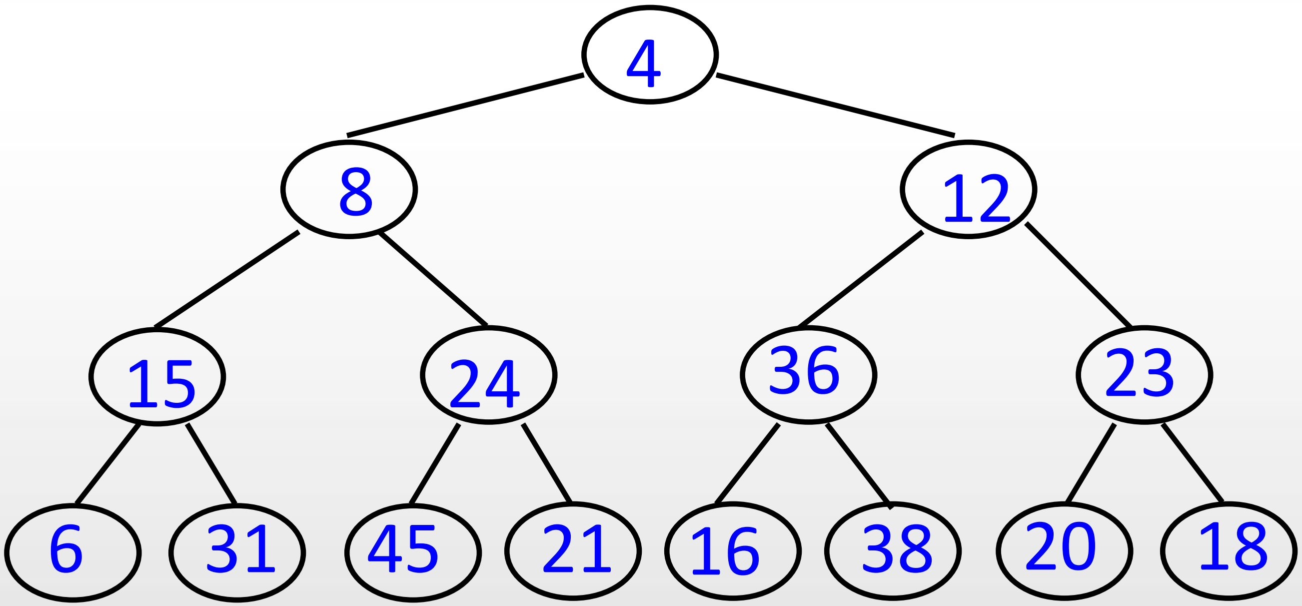
- To bound the running time of build_heap, we must bound the number of possible swaps.
- This can be done by computing the sum of the heights of all the nodes in the heap.

Build heap



- This tree consists of 1 node at height h, 2 nodes at height h-1, 2^2 nodes at height h-2, and in general 2^i nodes at height h-i.

Build heap



- The sum of the heights of all the nodes is then

$$S = \sum_{i=0}^{h} 2^i (h - i)$$

Build heap

- The sum of the heights of all the nodes is then

$$S=\sum_{i=0}^{h}2^{i}(h-i)$$

$$=h+2(h-1)+4(h-2)+8(h-3)+16(h-4)+\ldots+2^{h-1}(1)$$

$$2S=2h+4(h-1)+8(h-2)+16(h-3)+\ldots+2^{h}(1)$$

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

$$=(2^{h+1}-1)-(h+1)$$

Build heap

- The sum of the heights of all the nodes is then

$$= (2^{h+1} - 1) - (h+1)$$

$$= 2^{h}$$

$$= 2^{\log_2 n}$$

$$= n^{\log_2 2}$$

$$= O(n)$$

Application

- The Selection Problem

  Input: List of n elements, which can be totally ordered, and an integer k

  Output: Find the kth largest element

  Algo1:

  - Read elements into an array and sort them, returning the appropriate element. ≈ $O(n^2)$

Application

- The Selection Problem

  Algo2:

  - Read k elements into an array and sort them, the smallest is in the kth position.

  - Process the remaining elements one by one. As an element arrives, it is compared with kth element in the array.

  - If it is larger, the kth element is removed, and the new element is placed on the correct place among the remaining k-1 elements.

  - When the algorithm ends, the element in the kth position is the answer. $\approx O(n*k) \approx O(n^2)$

Application

- The Selection Problem

  Algo3:

  - Read n elements into an array.

  - Apply the build_heap algorithm.

  - Perform k delete_max operations.

  - The last element extracted from the heap is the answer.

  $\approx O(n \log n)$

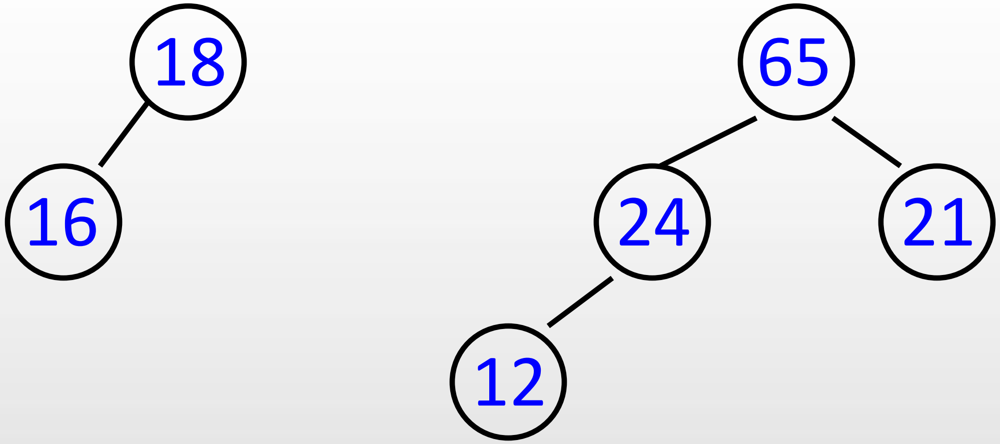
Application

- The Selection Problem

  Algo3:

  - build_heap $\approx O(n)$
  - delete_max $\approx O(\log n)$
  - k delete_max $\approx O(k \log n)$
  - total $\approx O(n + k \log n)$
  - if $k = \lceil n/2 \rceil \approx O(n \log n)$

Merging



6. Heaps

Binomial Queues

Binomial Queues

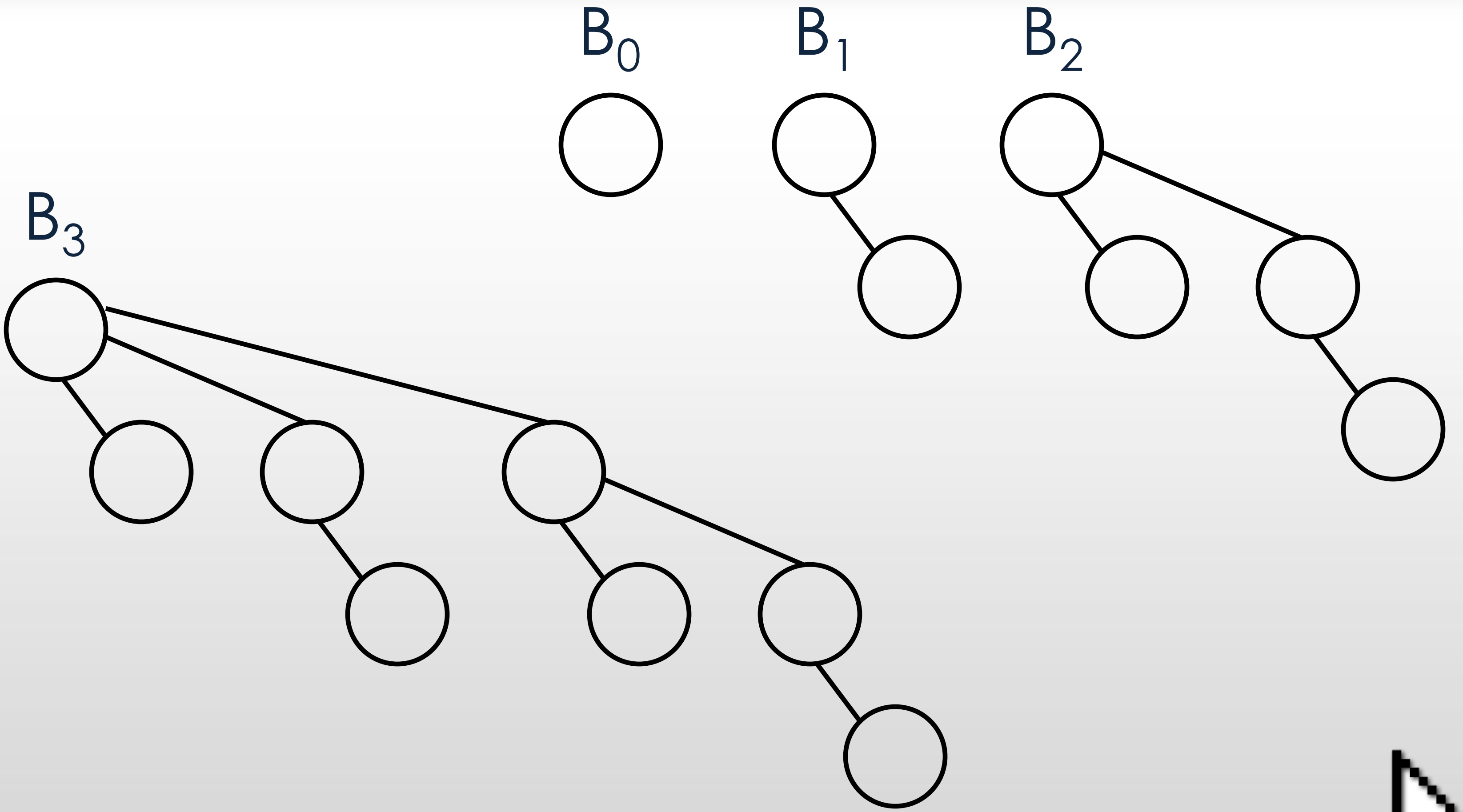
- Support merging, insertion, and delete_max in O(log n) worst-case time per operation.

- Collection of heap-ordered trees, known as a *forest*.
- Each of the heap-ordered trees are of a constrained form known as a *binomial tree*.

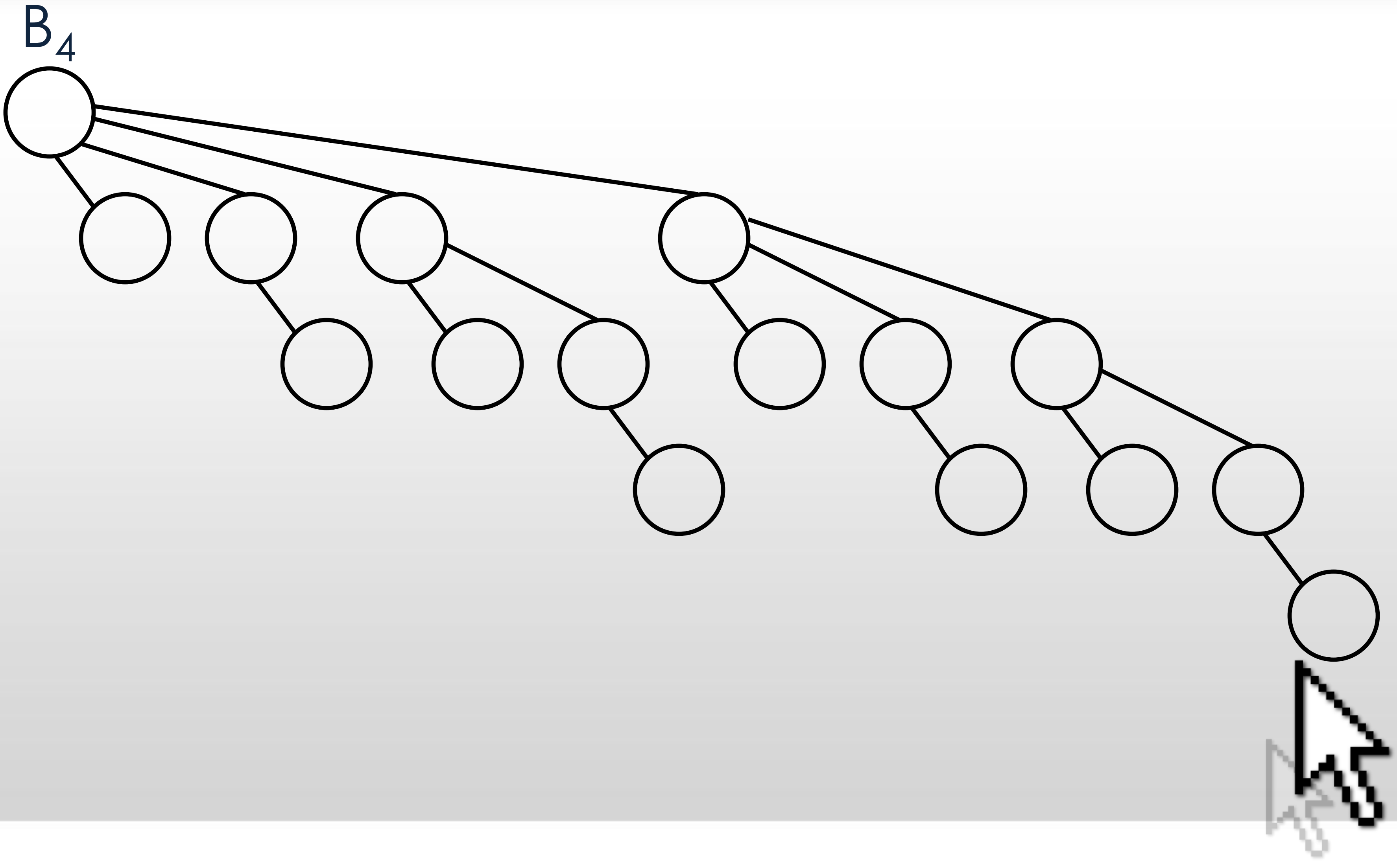
Binomial Queues

- There is at most one binomial tree of every height.
- A binomial tree of height 0 is a one-node tree; a binomial tree, B_k, of height k is formed by attaching a binomial tree, B_{k-1}, to the root of another binomial tree, B_{k-1}.
- Binomial trees of height k have exactly 2^k nodes.

Binomial trees



Binomial trees



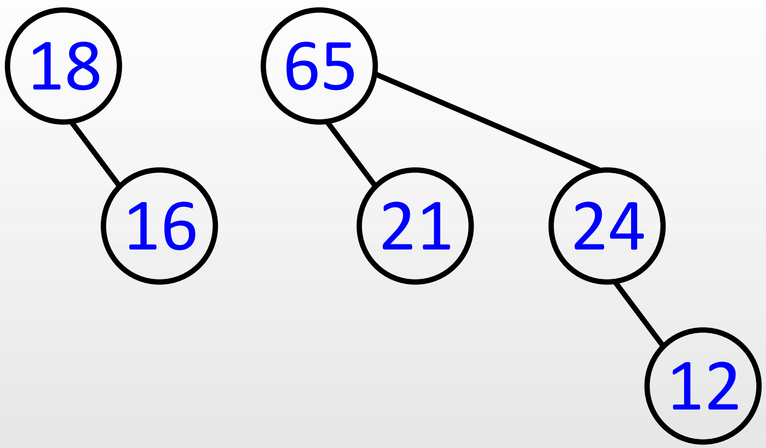
Priority Queue

- If we impose heap order on the binomial trees and allow at most one binomial tree of any height, we can uniquely represent a priority queue of any size by a collection of binomial trees
- A priority queue of size 13 could be represented by the forest B_3, B_2, B_0.
- Can also be represented as 1101

Binomial Queue Operations

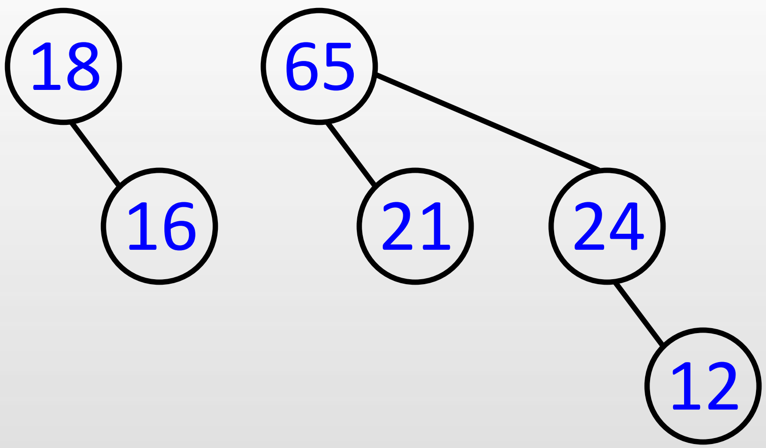
- Maximum element – scan the roots of all trees $\approx O(\log n)$

Binomial trees



Binomial trees

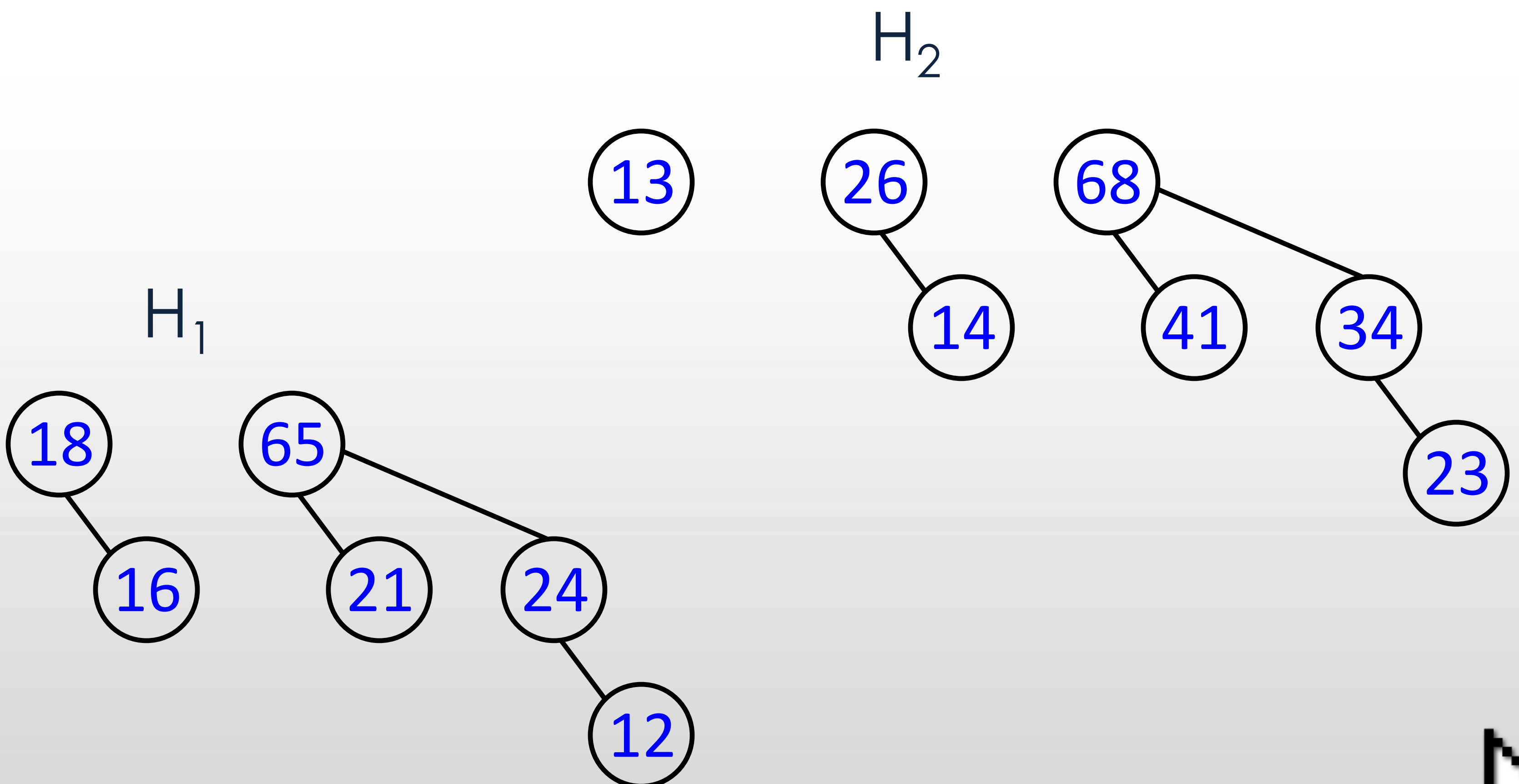
Priority Queue of size: 6 = 110



Binomial Queue Operations

- Maximum element – scan the roots of all trees $\approx O(\log n)$
- Merging two binomial queues $\approx O(\log n)$

Binomial trees

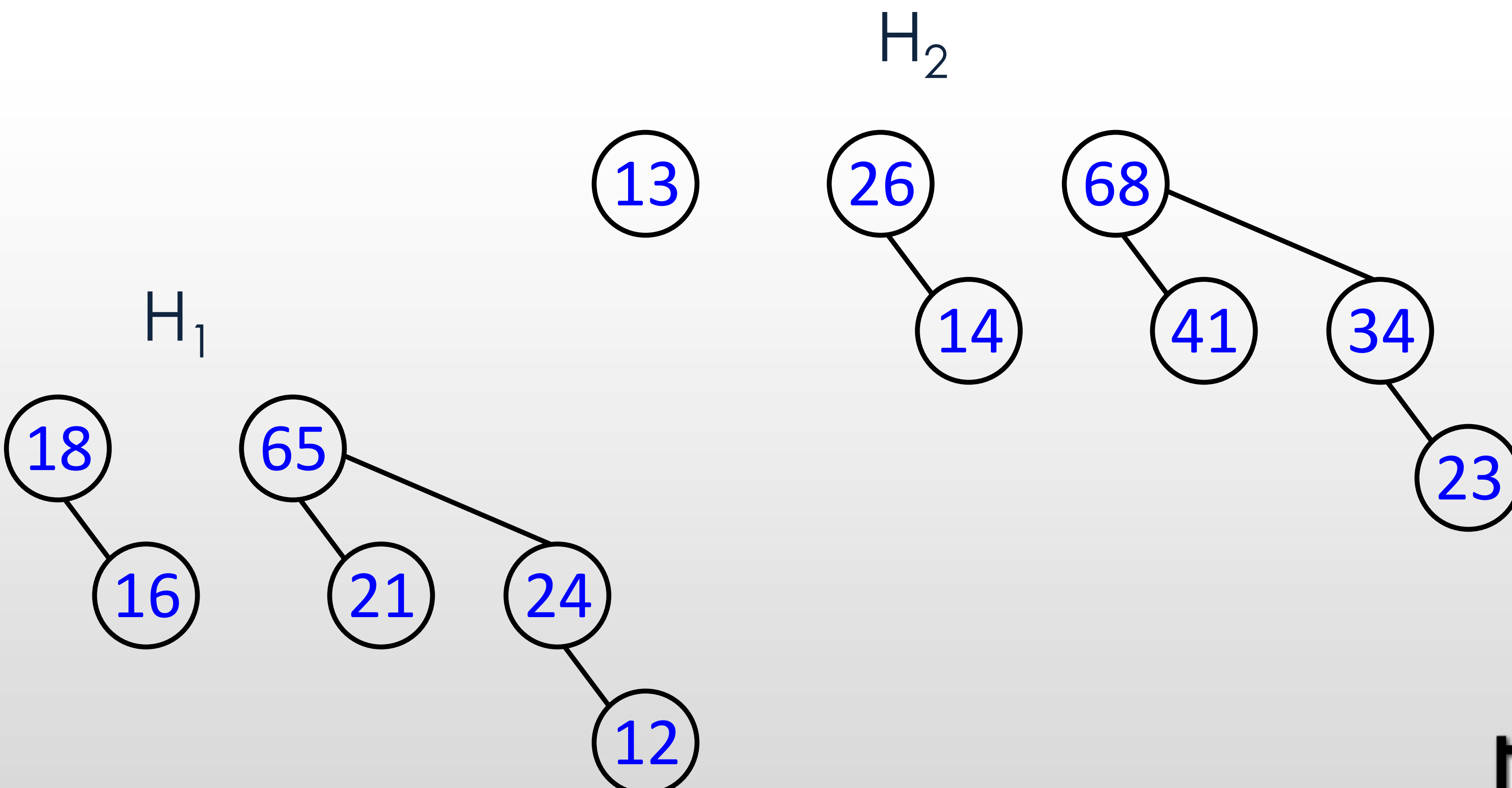


Binomial trees

H_3

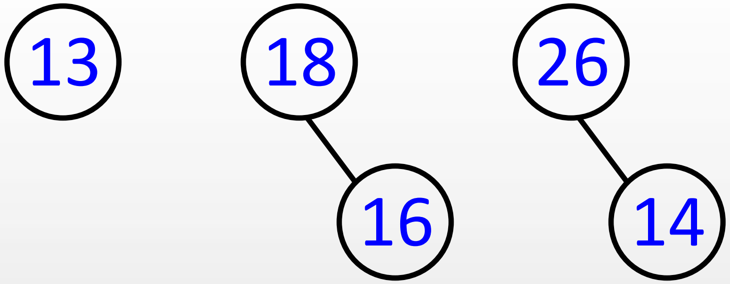
13

Binomial trees



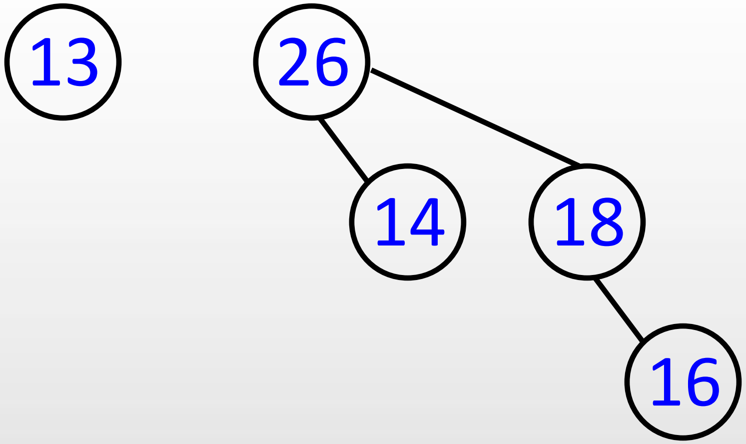
Binomial trees

H_3

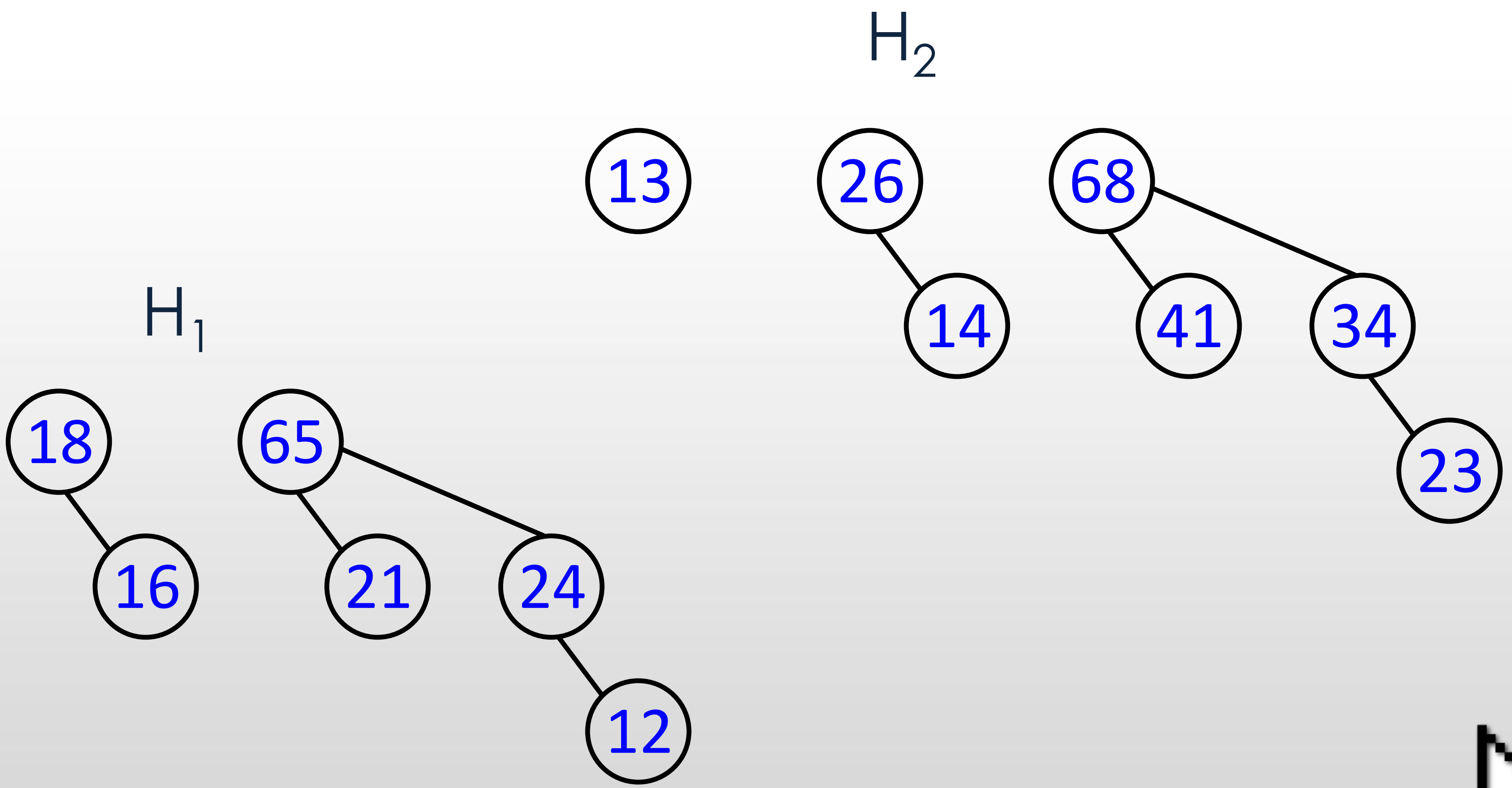


Binomial trees

H_3

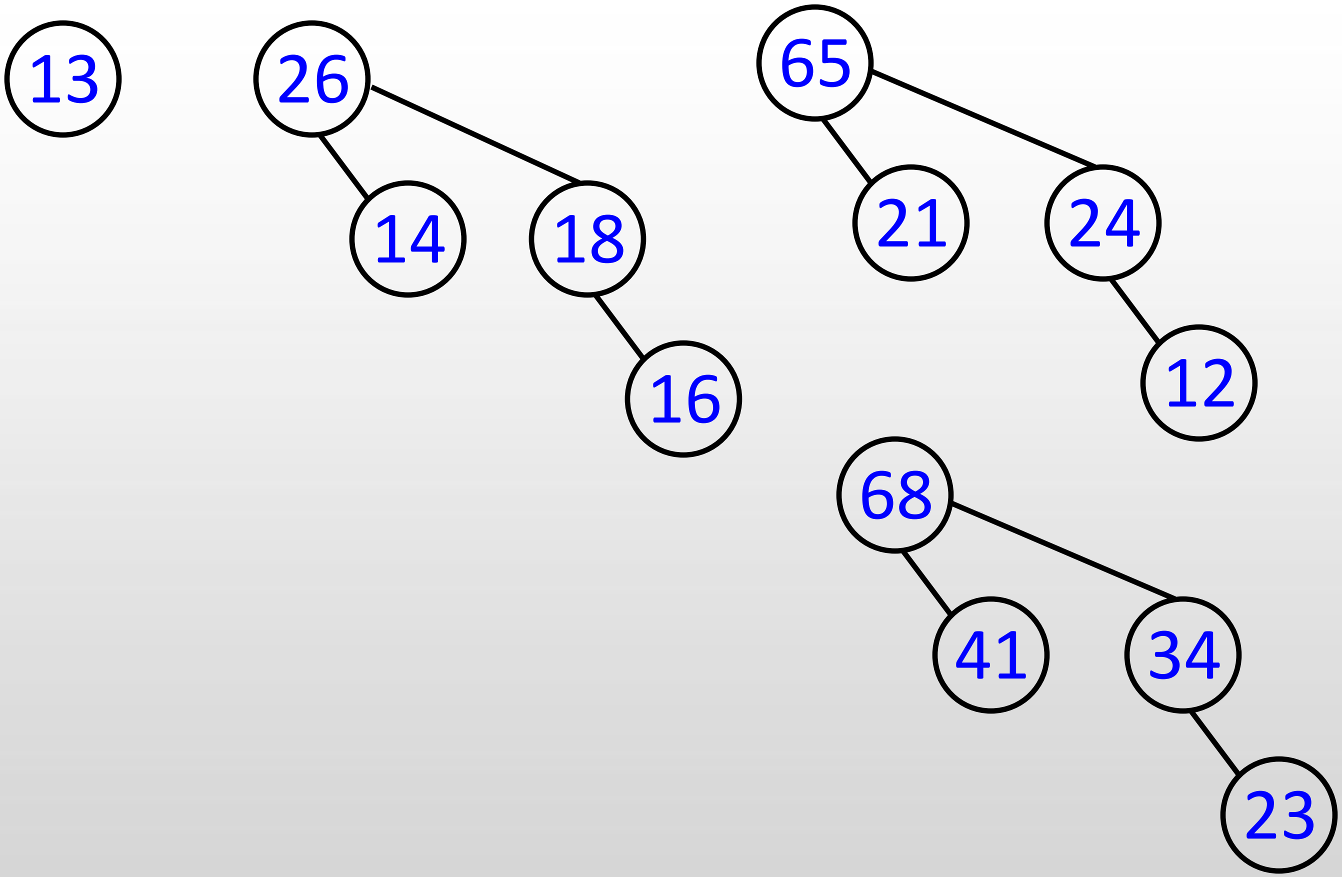


Binomial trees



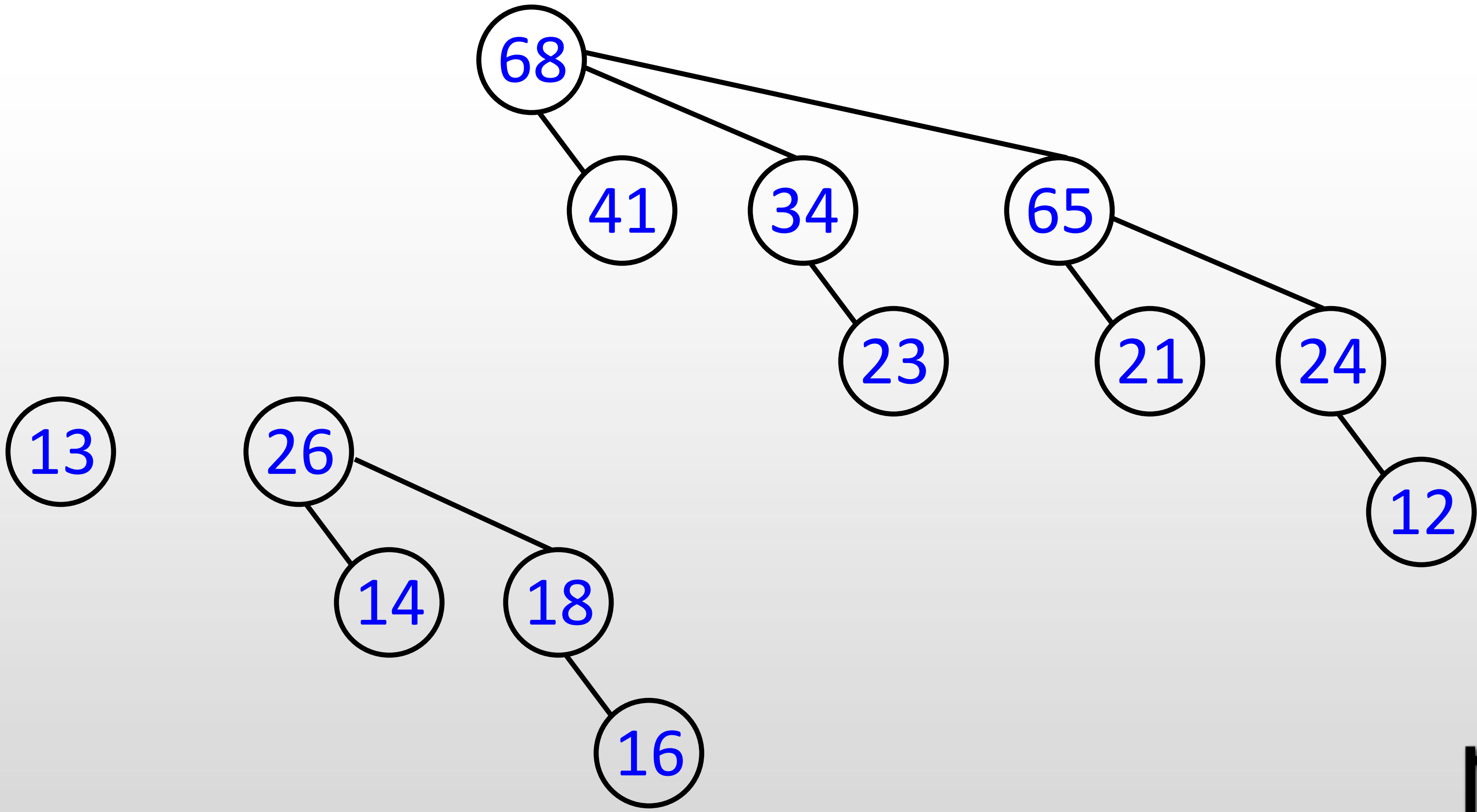
Binomial trees

H_3



Binomial trees

H_3

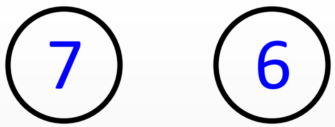


Binomial Queue Operations

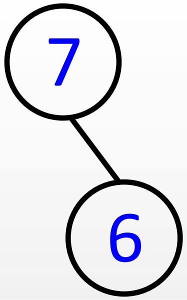
- Maximum element – scan the roots of all trees
  $\approx O(\log n)$
- Merging two binomial queues $\approx O(\log n)$
- Insertion – creates one-node tree and merge
  $\approx O(\log n)$

Binomial trees

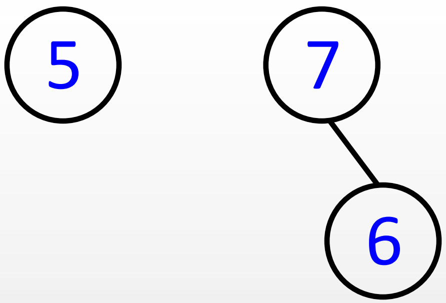
Binomial trees



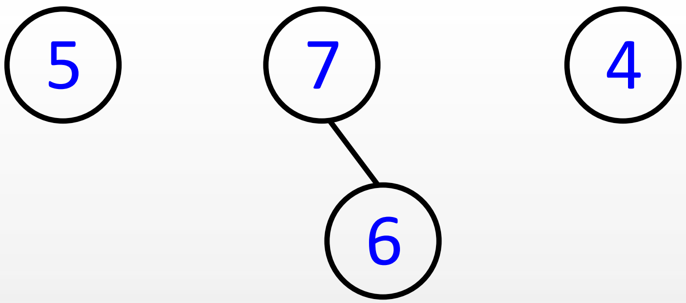
Binomial trees



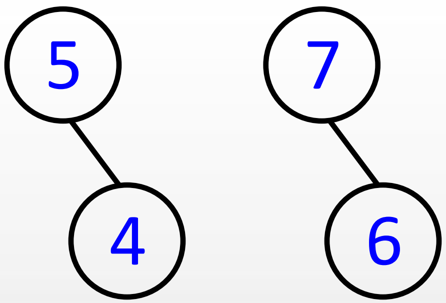
Binomial trees



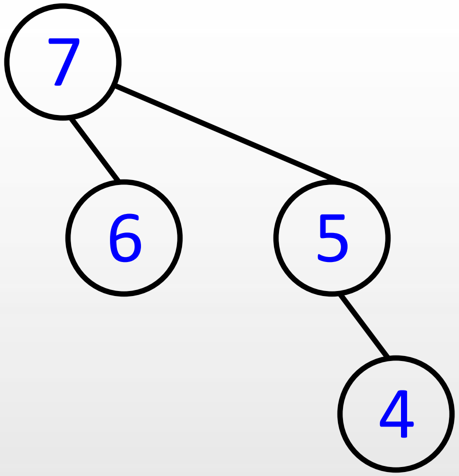
Binomial trees



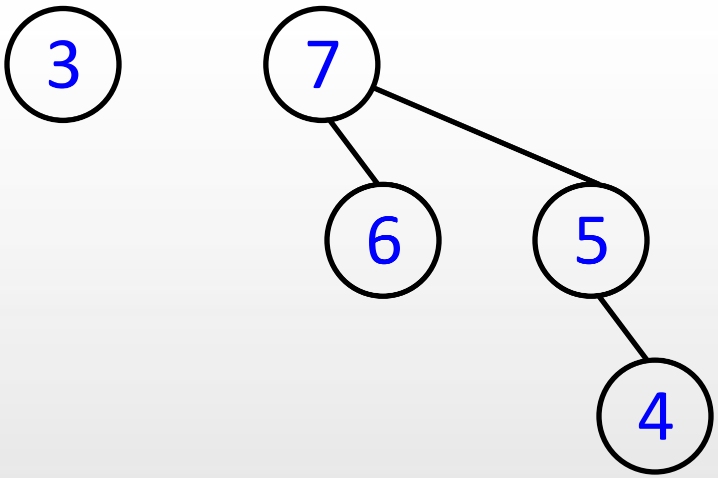
Binomial trees



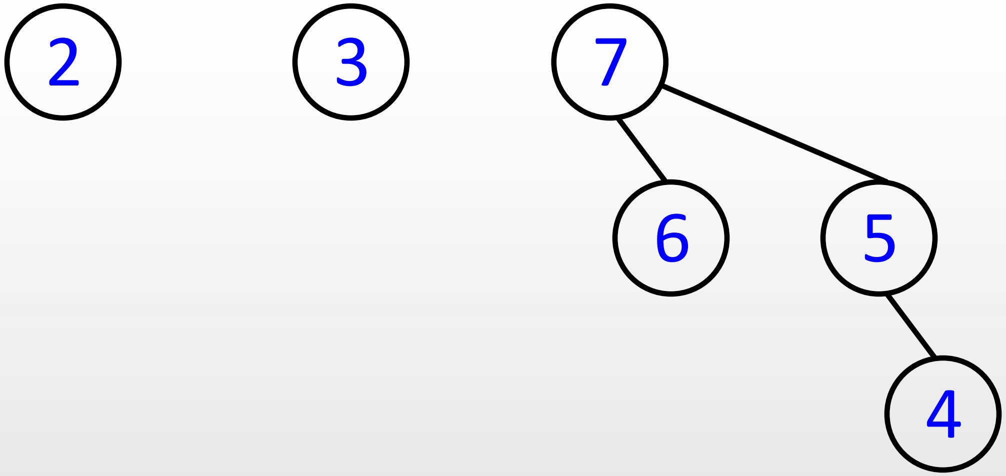
Binomial trees



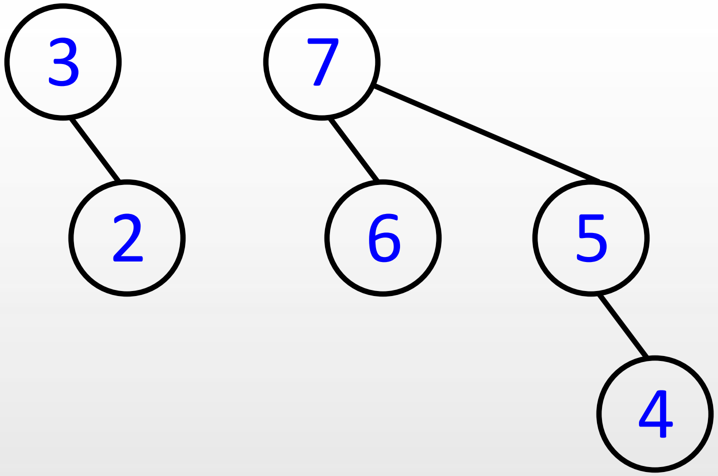
Binomial trees



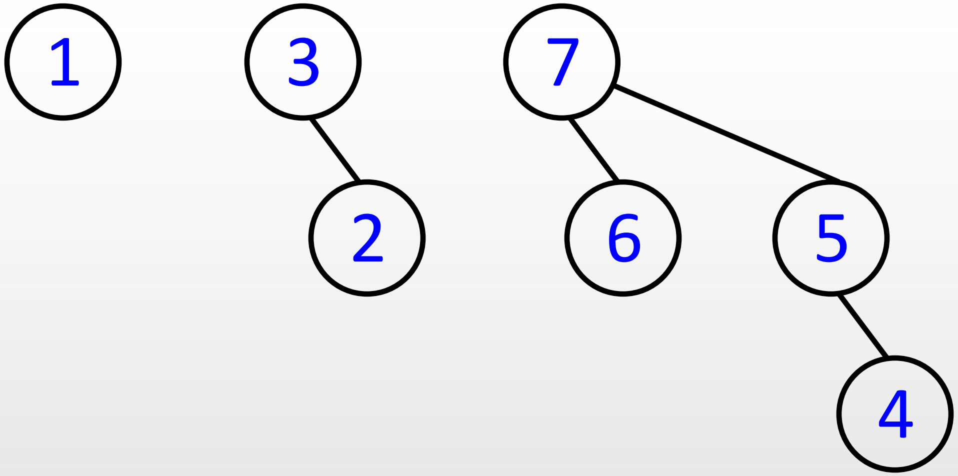
Binomial trees



Binomial trees



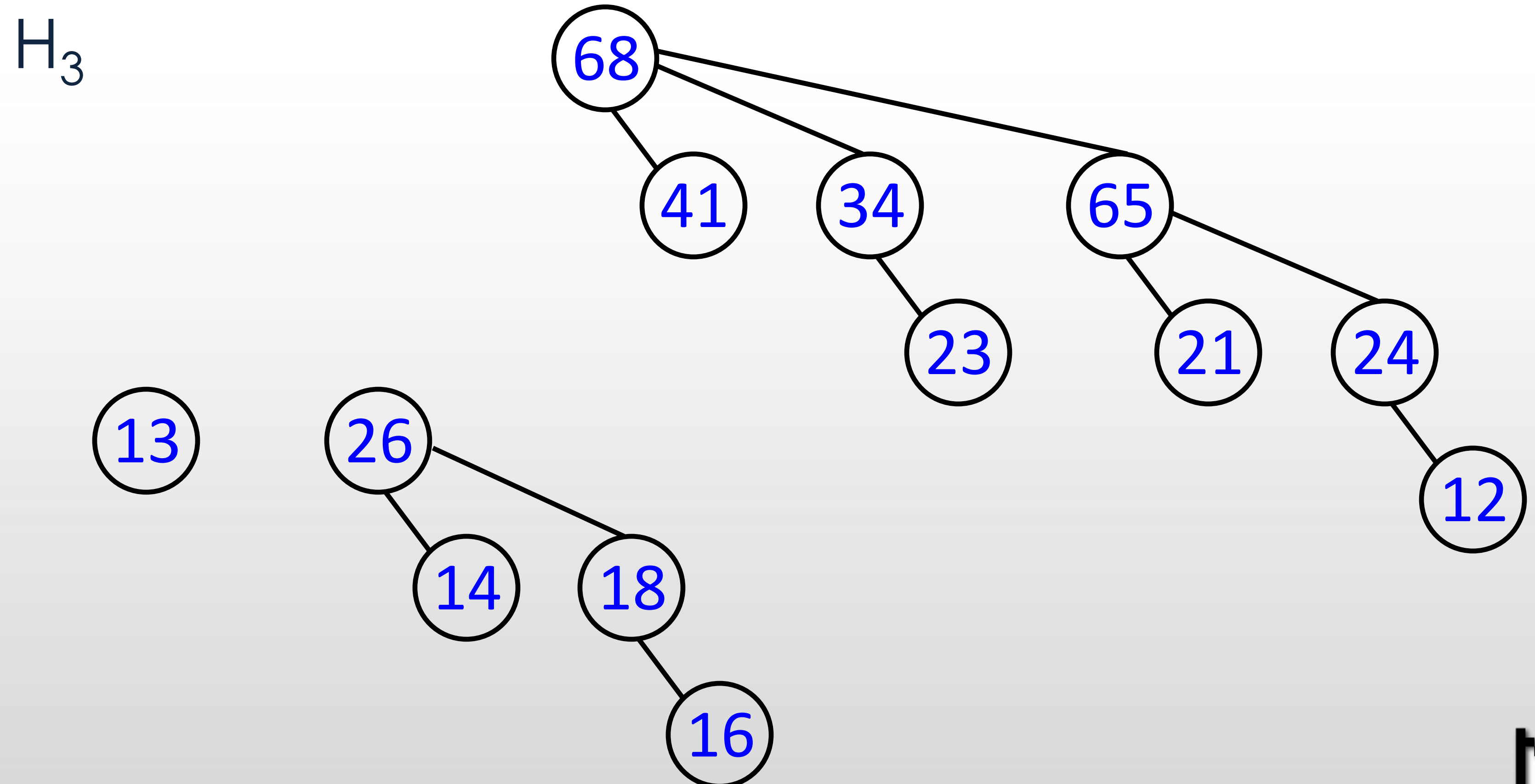
Binomial trees



Binomial Queue Operations

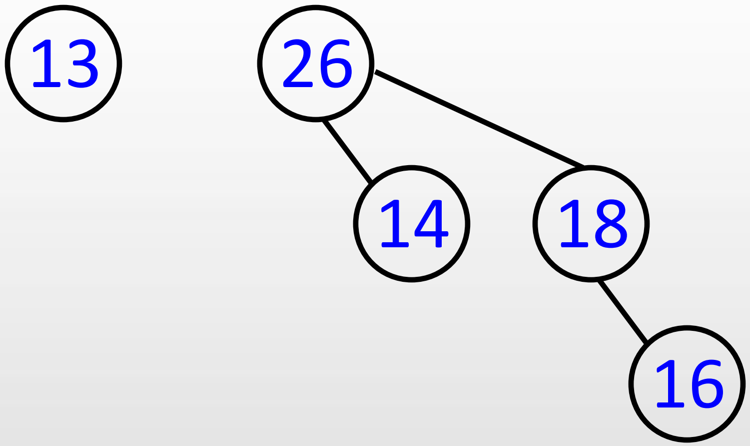
- Maximum element – scan the roots of all trees $\approx O(\log n)$
- Merging two binomial queues $\approx O(\log n)$
- Insertion – creates one-node tree and merge $\approx O(\log n)$
- Delete_max $\approx O(\log n)$

Binomial trees



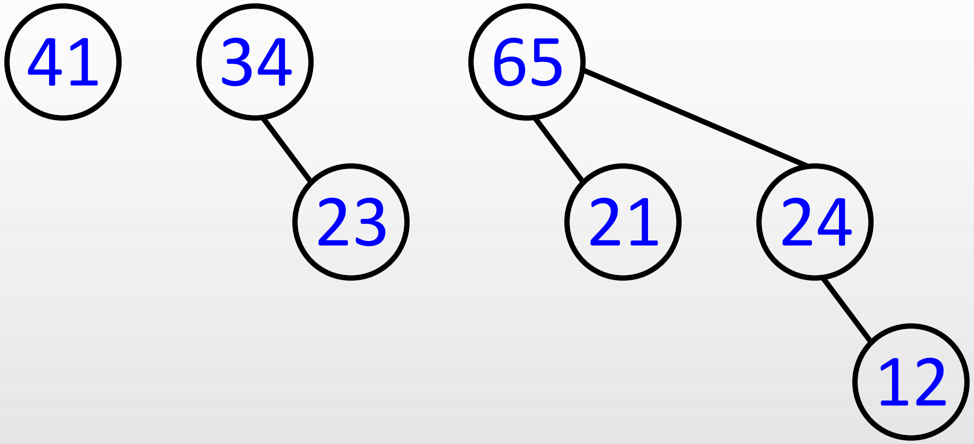
Binomial trees

H’

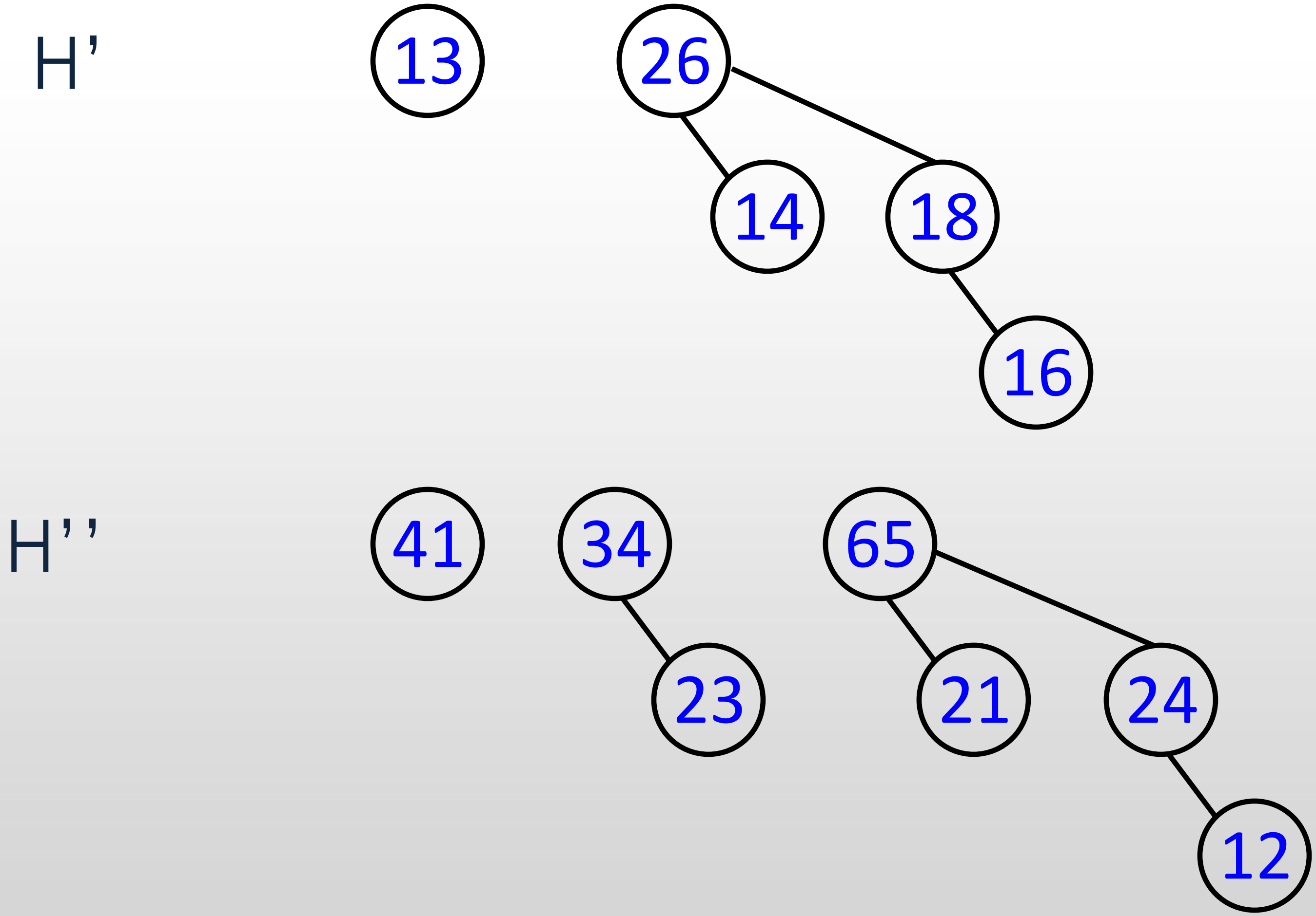


Binomial trees

H''

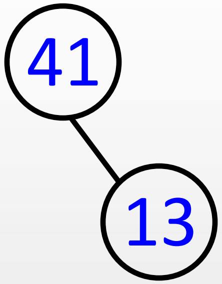


Binomial trees



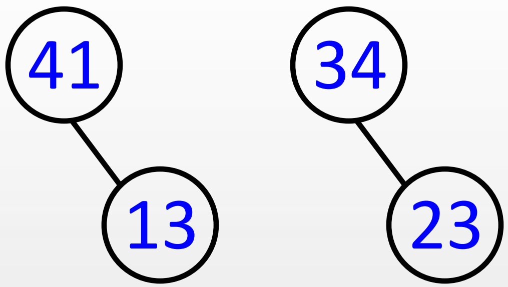
Binomial trees

H_3



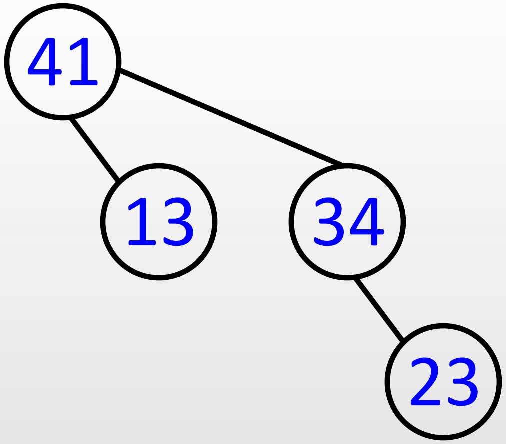
Binomial trees

H_3



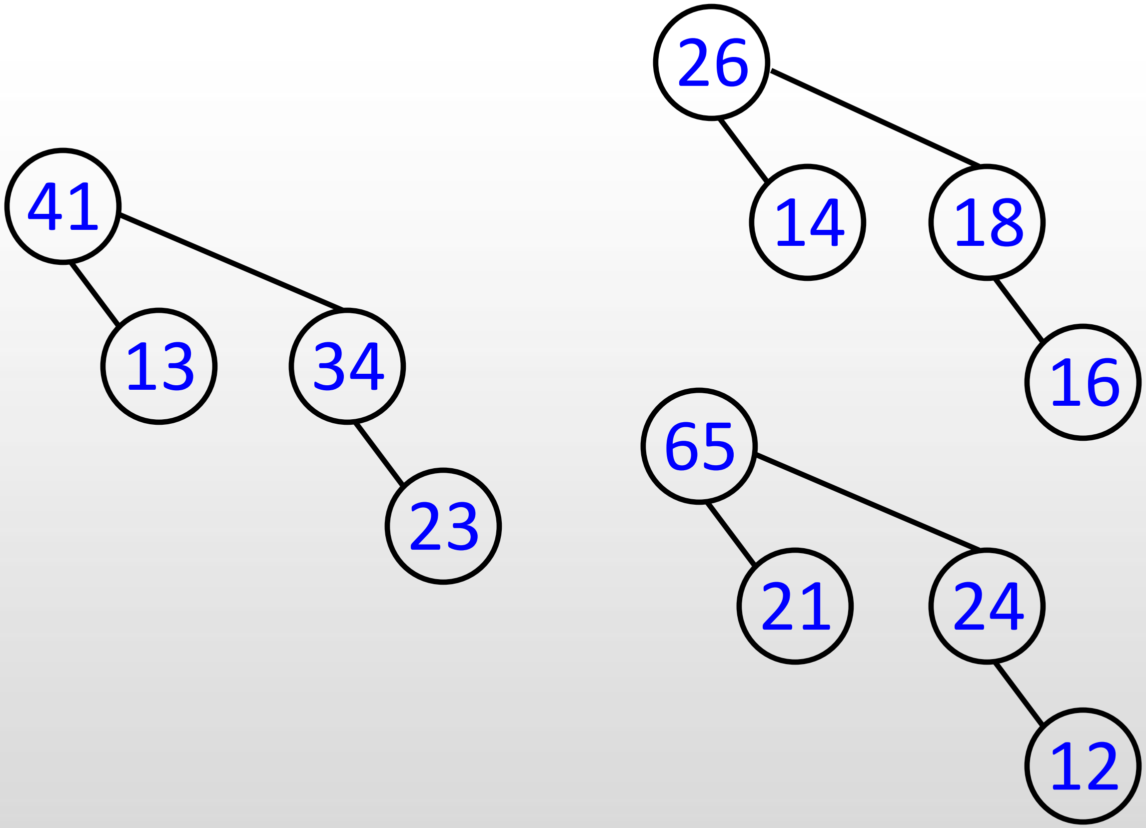
Binomial trees

H_3



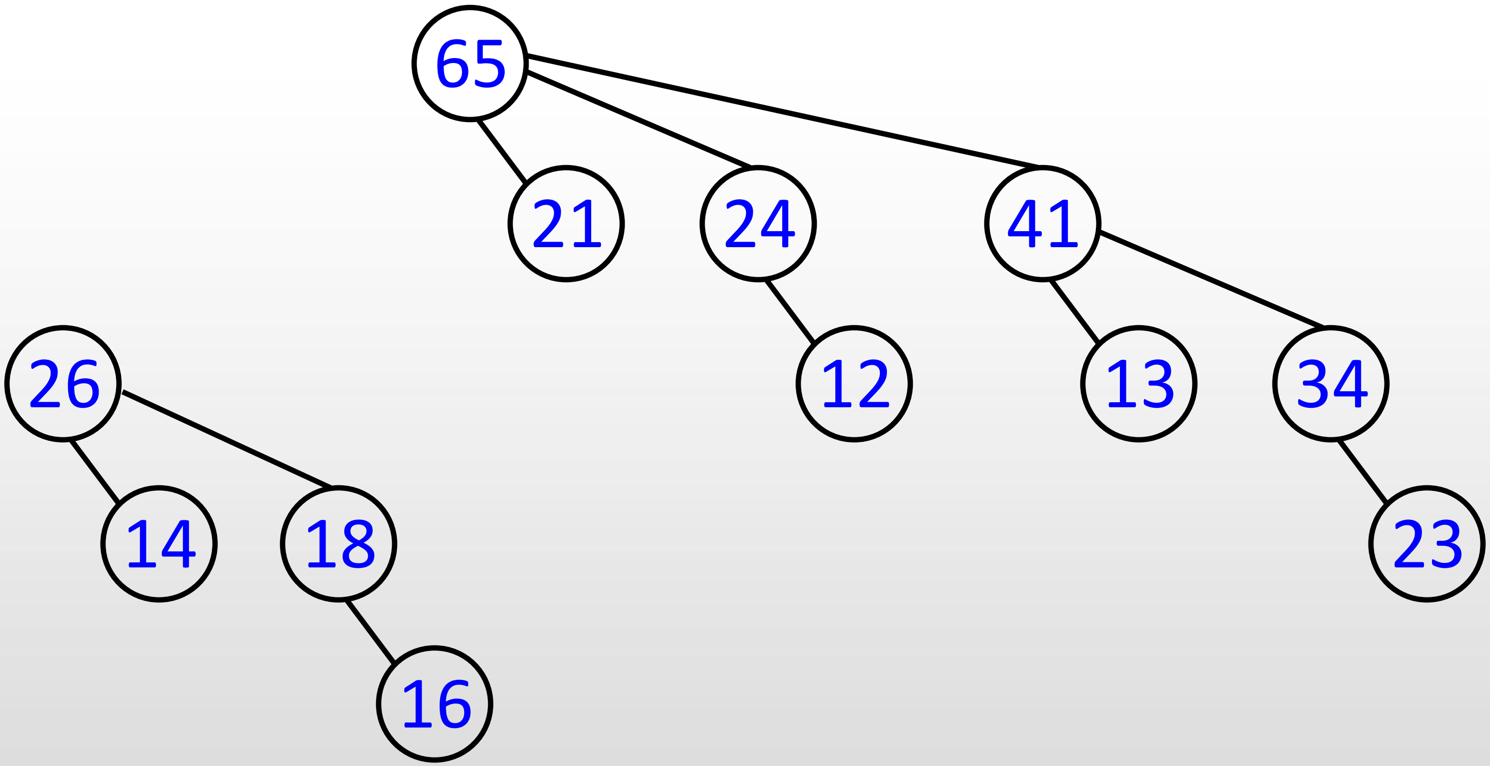
Binomial trees

H_3



Binomial trees

H_3



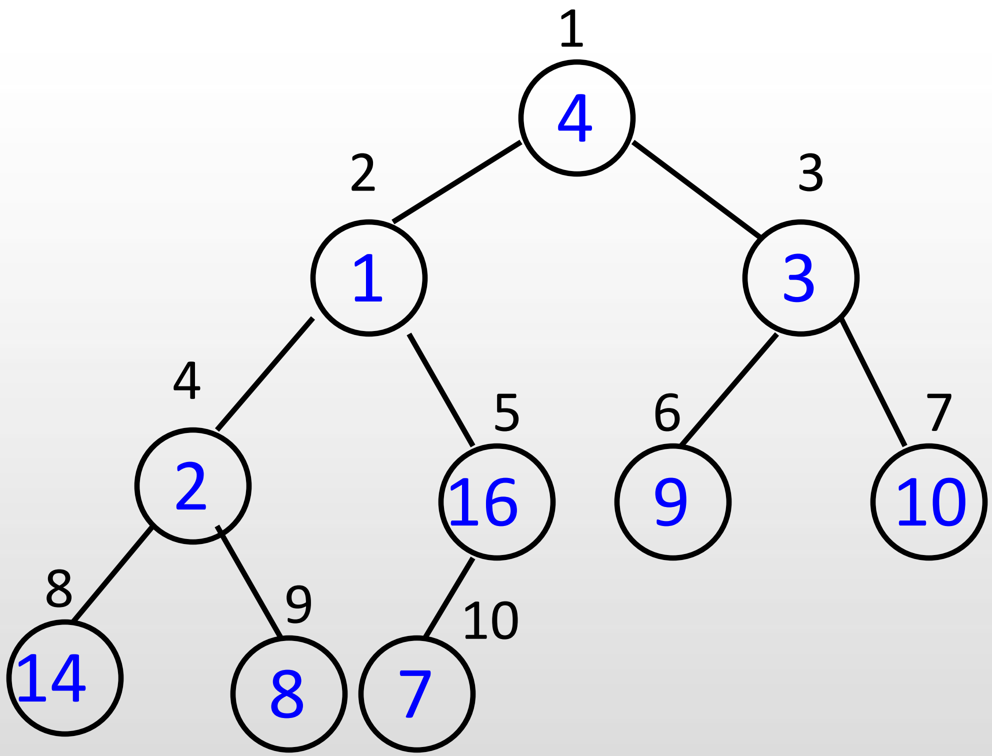
6. Heaps

Heapsort

Heapsort

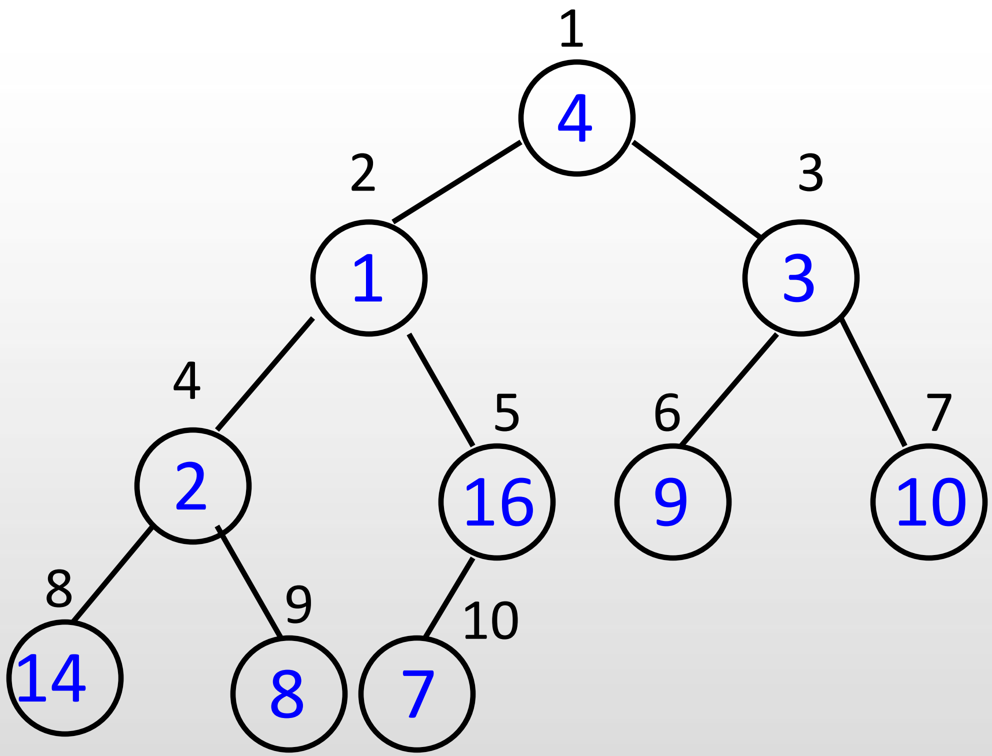
```
HeapSort(A,n)
 begin
   BuildHeap(A)
   for i=n downto 2 do
        swap(A[1],A[i])
        Heapify(A,1,(i-1))
end
```

Sort 4,1,3,2,16,9,10,14,8,7



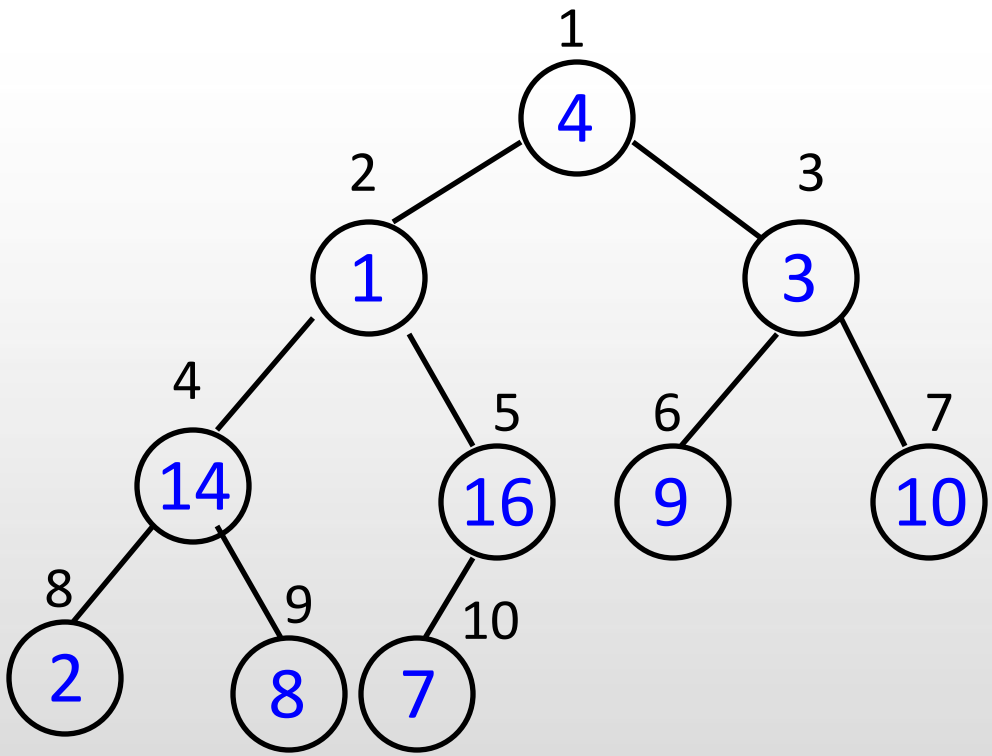
BUILDHEAP(A)

Sort 4,1,3,2,16,9,10,14,8,7



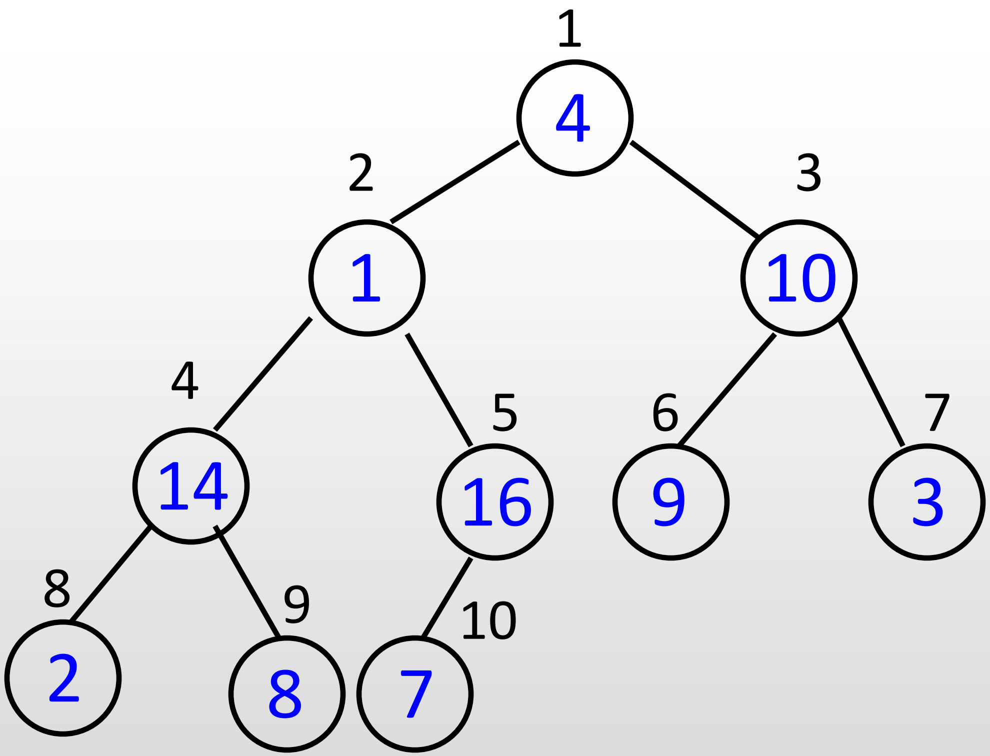
BUILDHEAP(A)

Sort 4,1,3,2,16,9,10,14,8,7



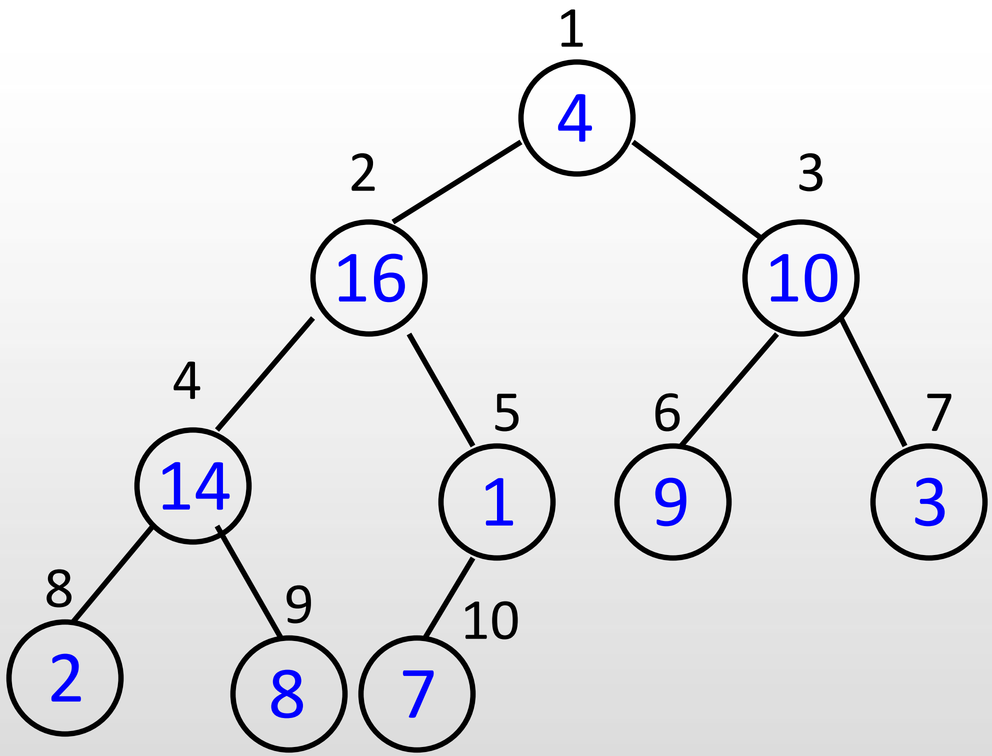
BUILDHEAP(A)

Sort 4,1,3,2,16,9,10,14,8,7



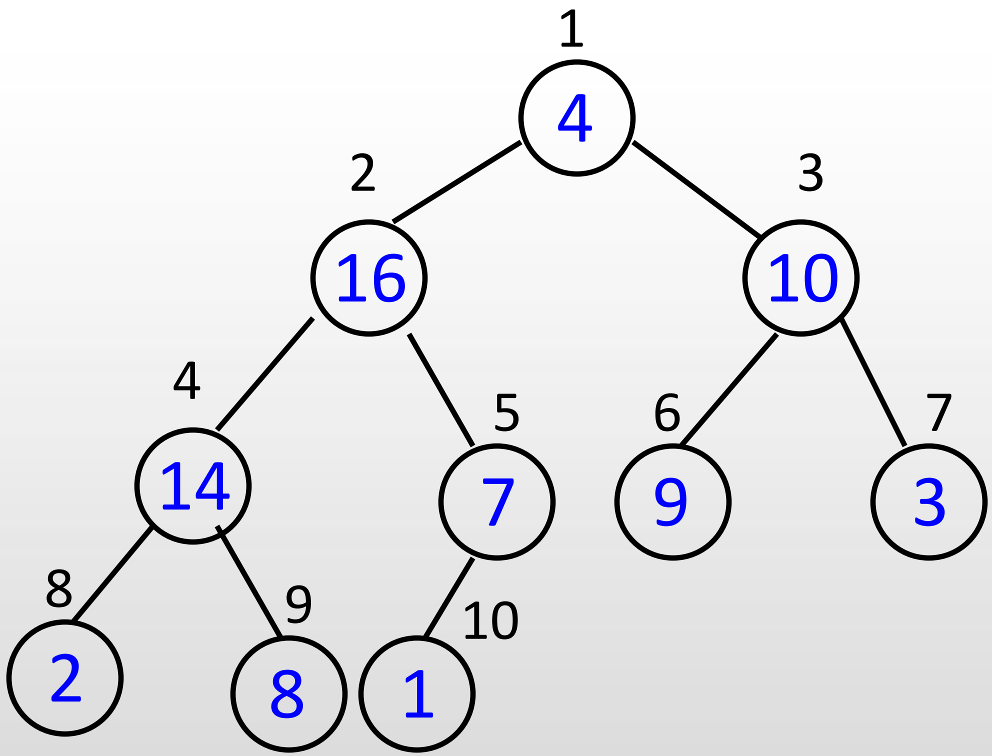
BUILDHEAP(A)

Sort 4,1,3,2,16,9,10,14,8,7



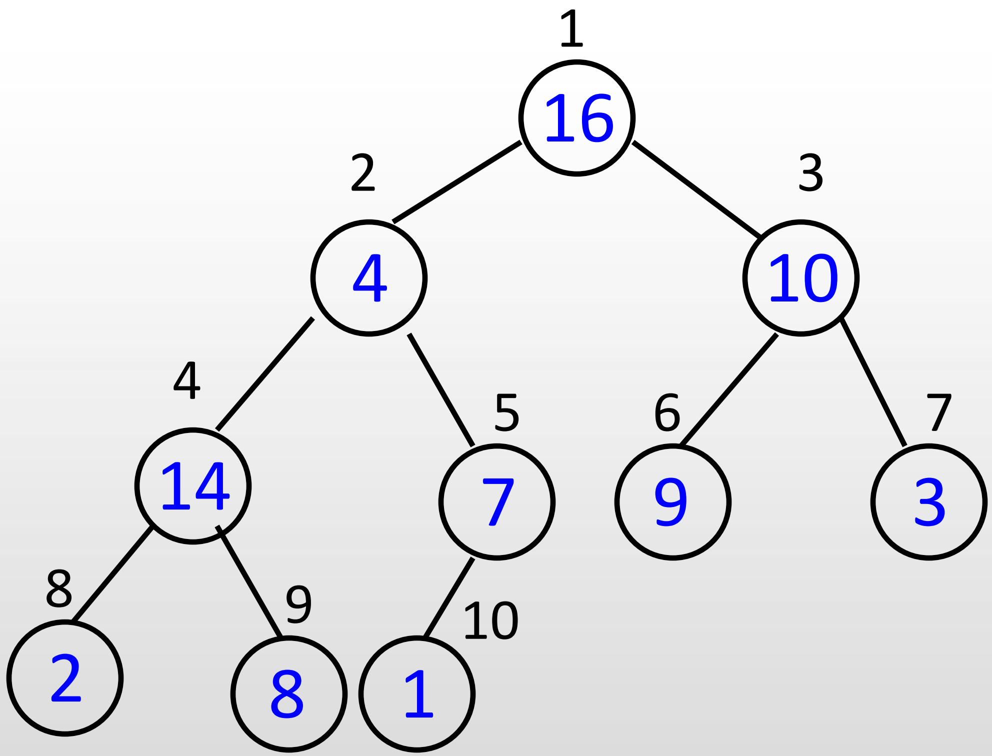
BUILDHEAP(A)

Sort 4,1,3,2,16,9,10,14,8,7



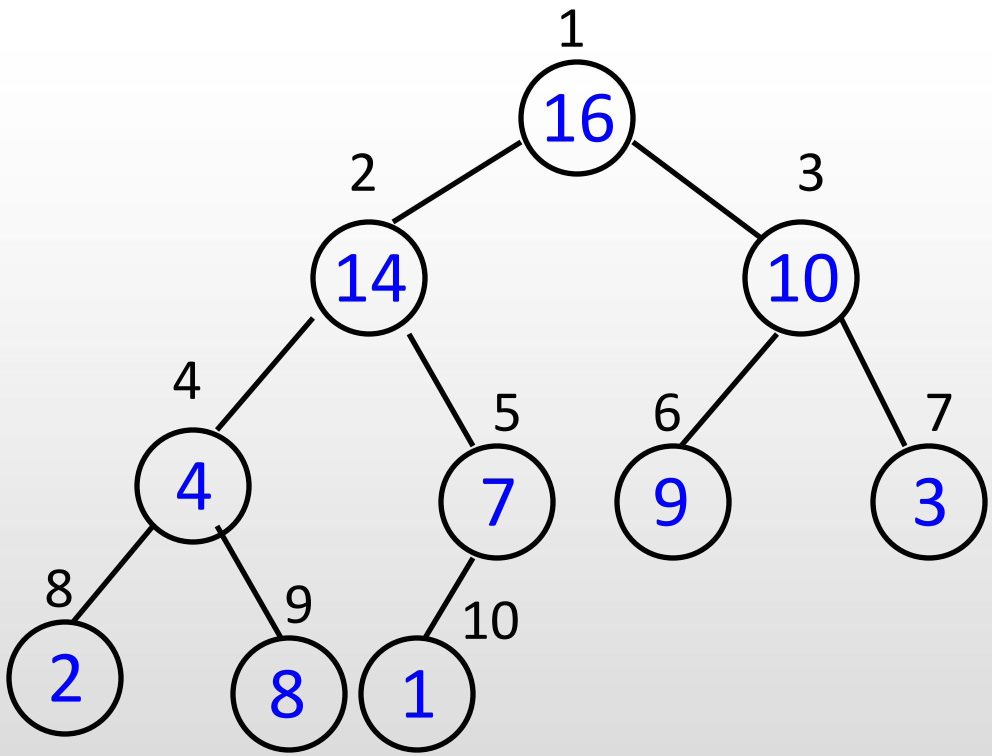
BUILDHEAP(A)

Sort 4,1,3,2,16,9,10,14,8,7



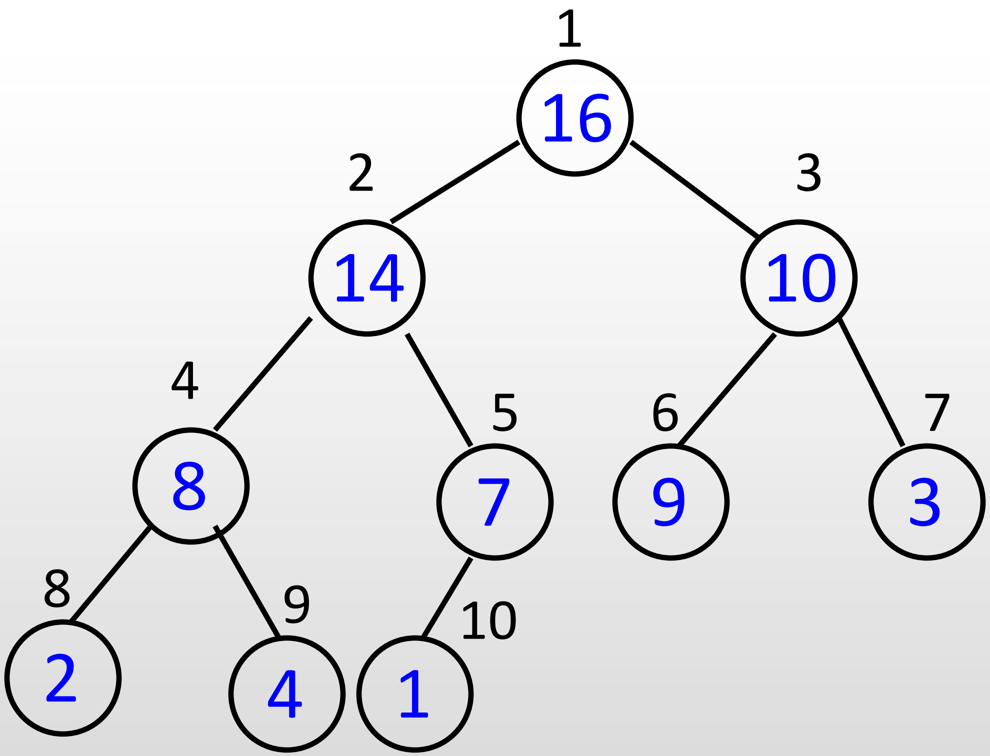
BUILDHEAP(A)

Sort 4,1,3,2,16,9,10,14,8,7

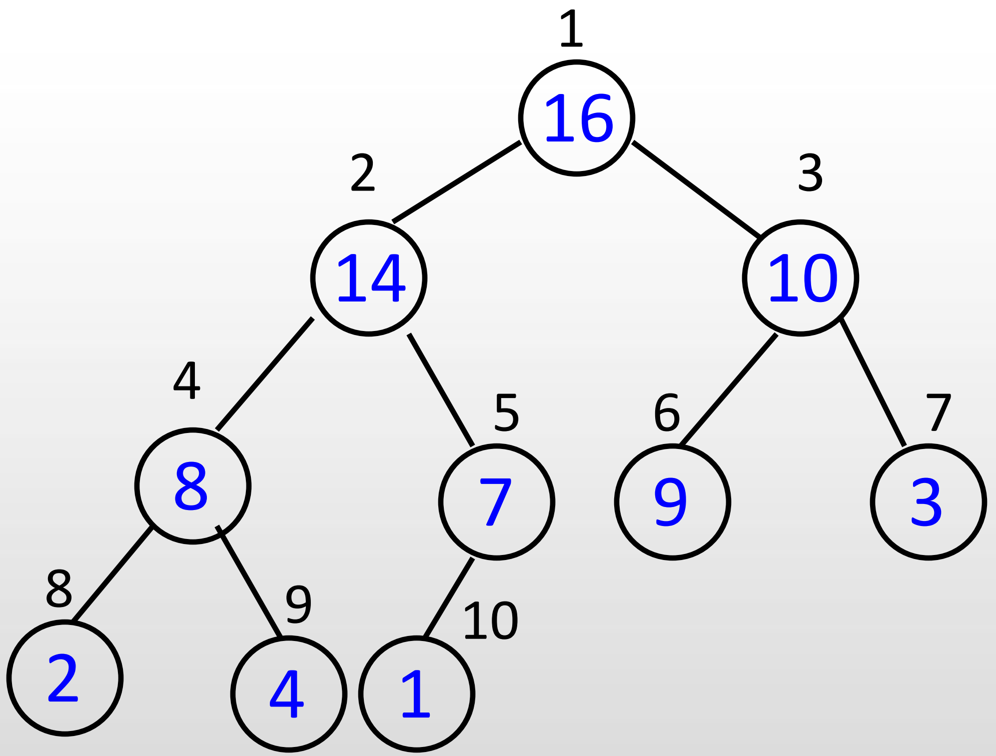


BUILDHEAP(A)

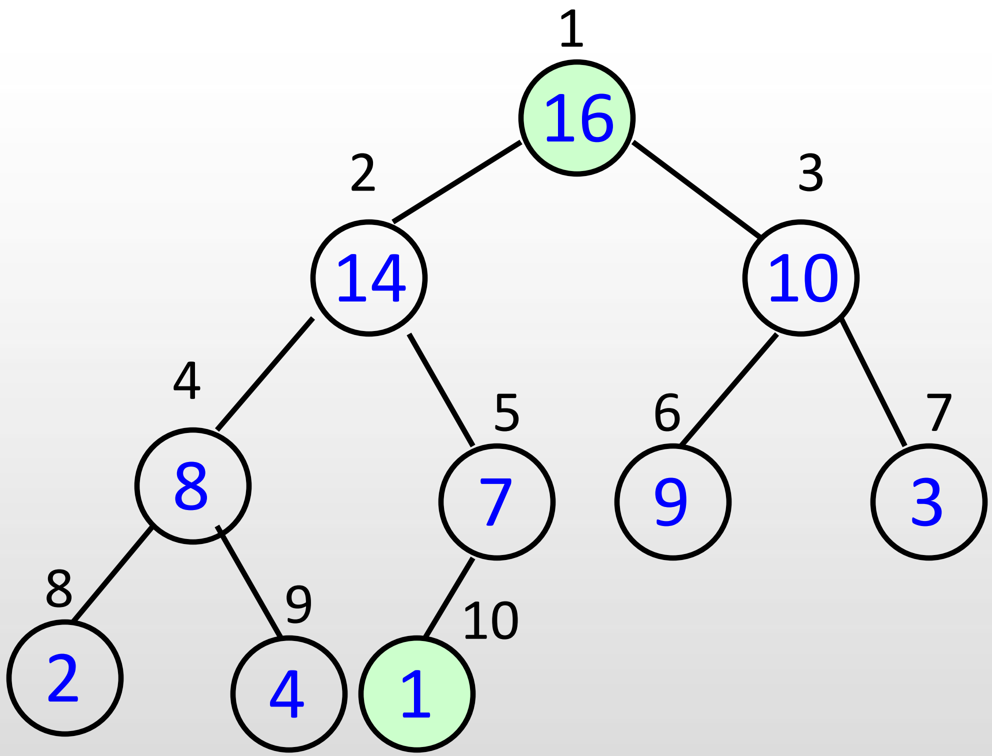
Resulting Max-Heap



1st pass: i=10

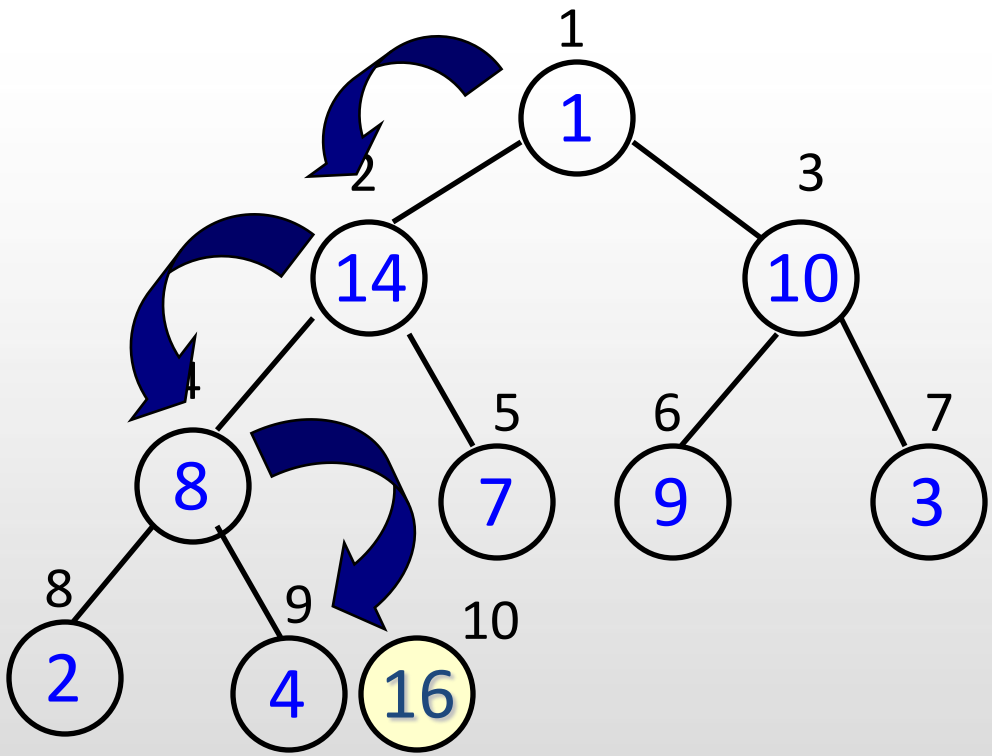


1st pass: i=10



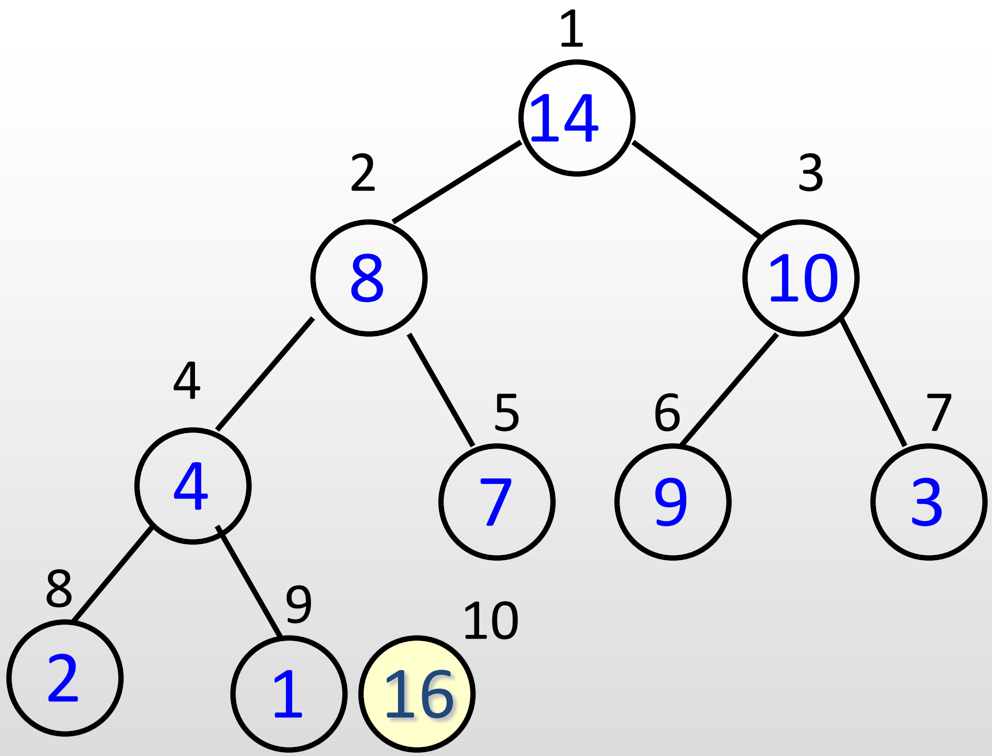
swap(A[1],A[i])

1st pass: i=10

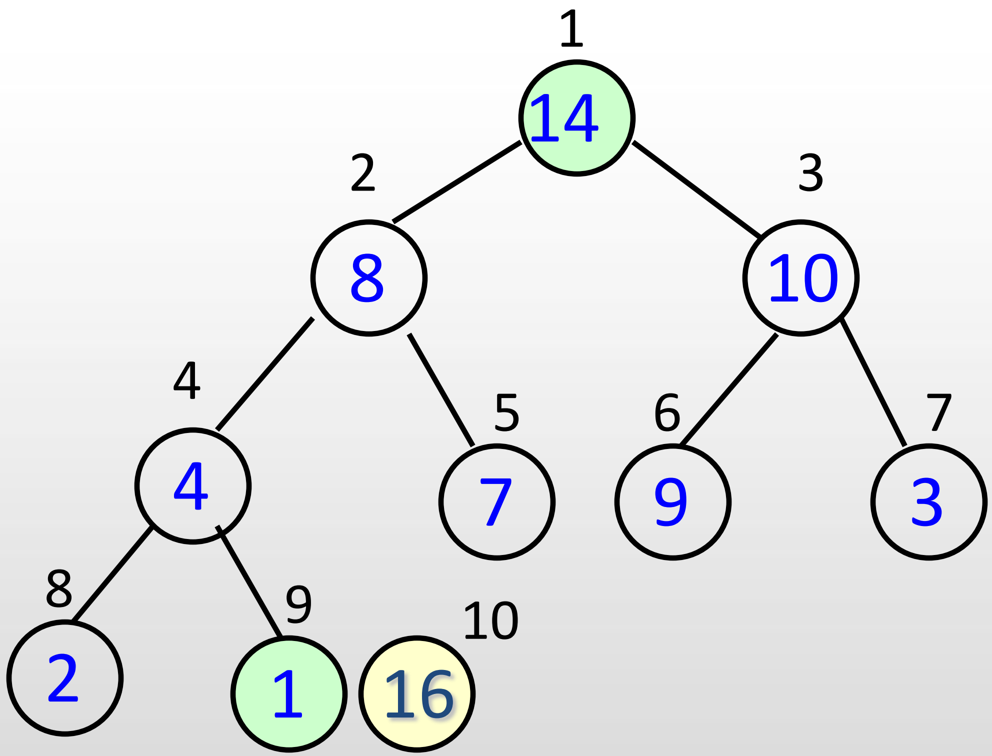


Heapify(A,1,9)

2nd pass: i=9

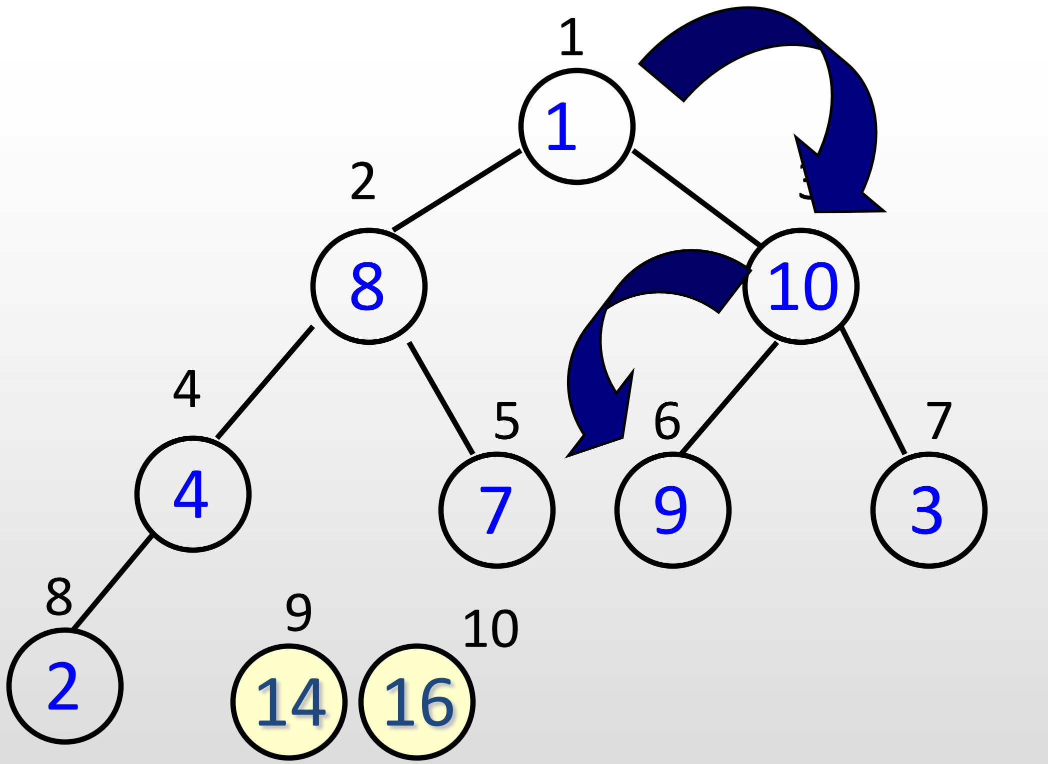


2nd pass: i=9



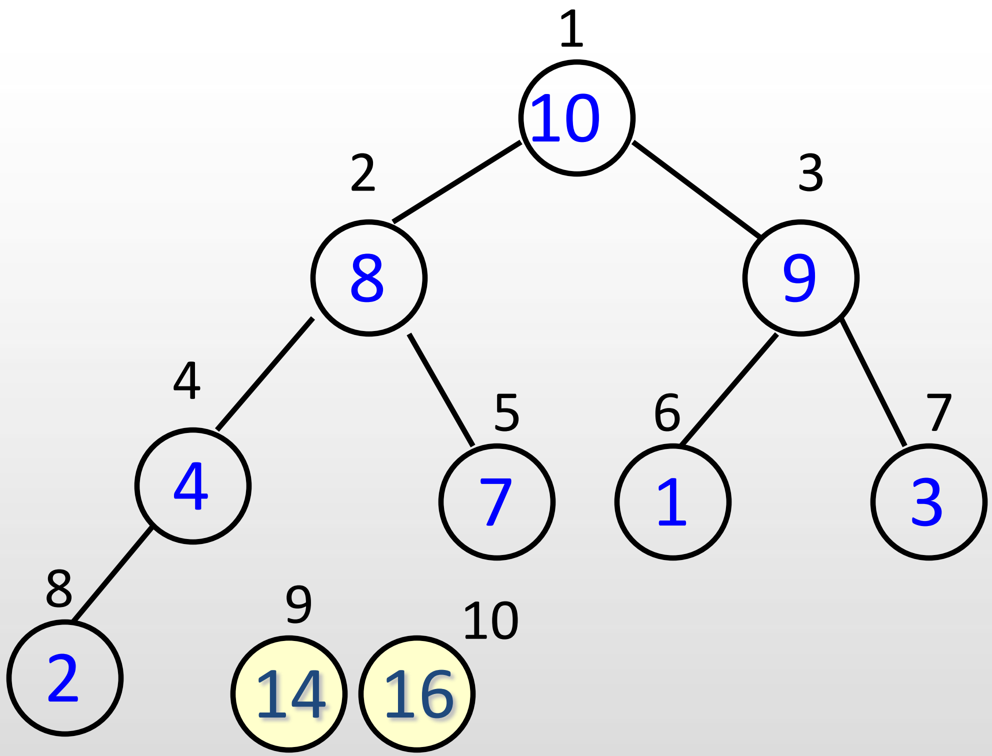
swap(A[1],A[9])

2nd pass: i=9

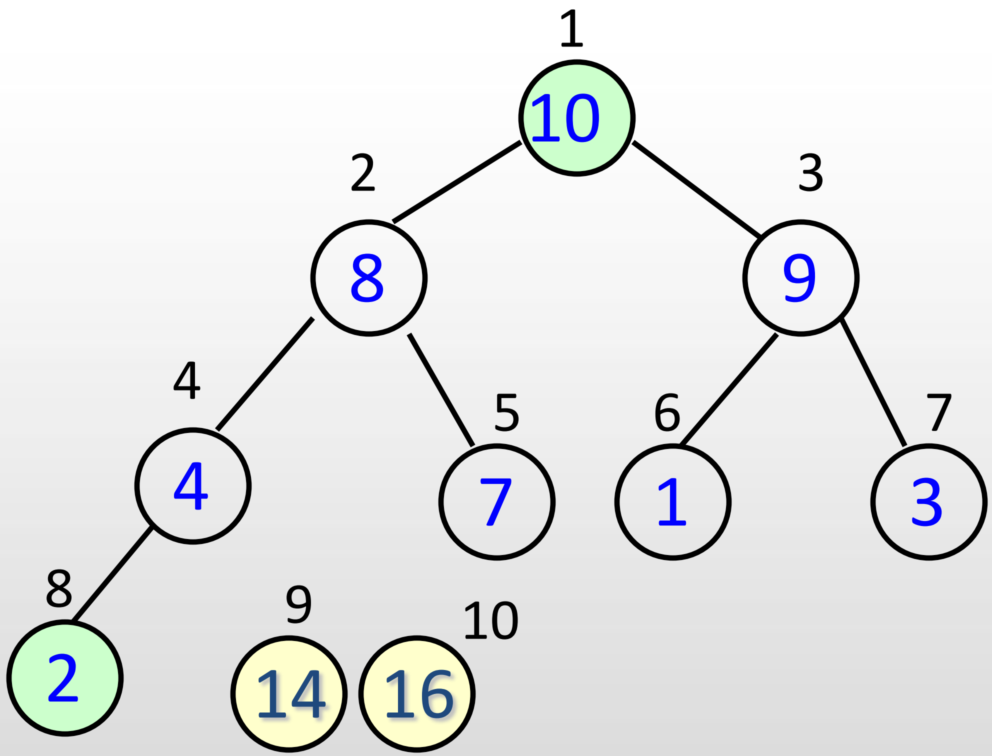


Heapify(A,1,8)

3rd pass: i=8

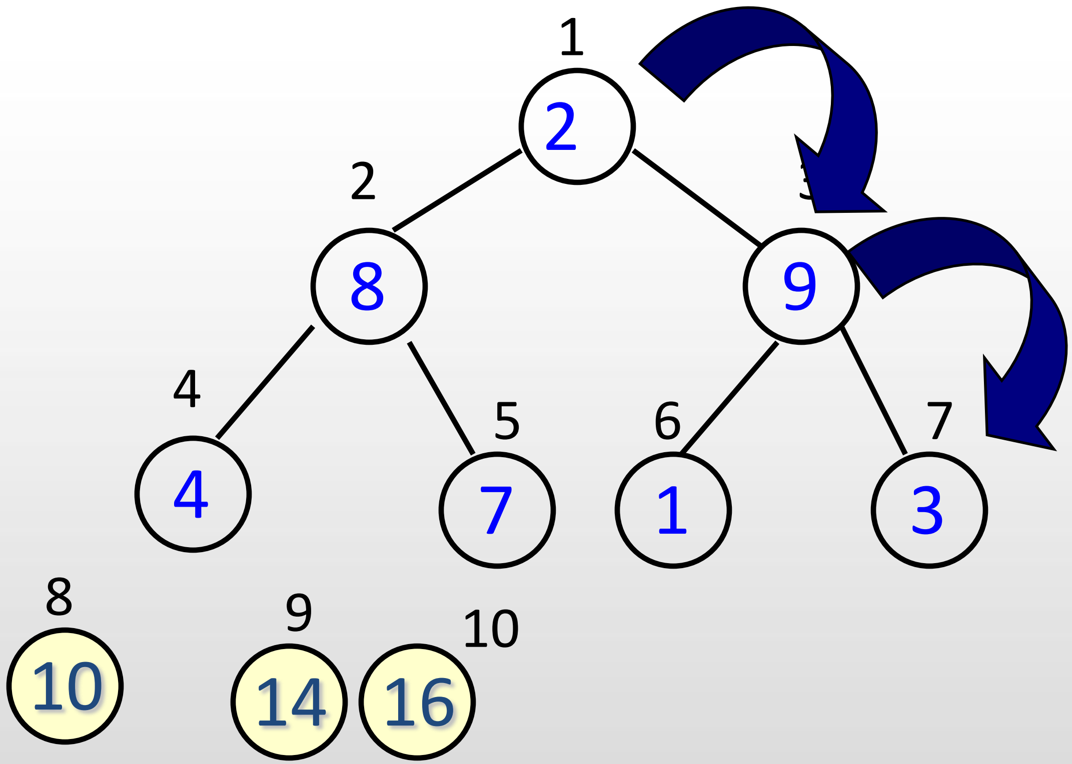


3rd pass: i=8



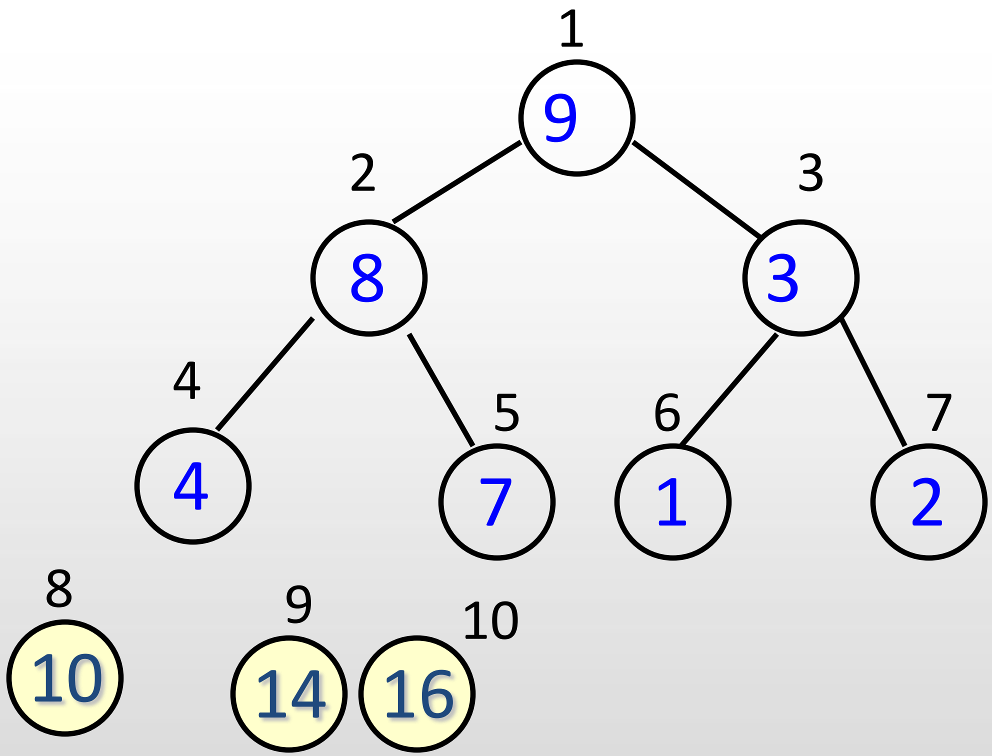
swap(A[1],A[8])

3rd pass: i=8

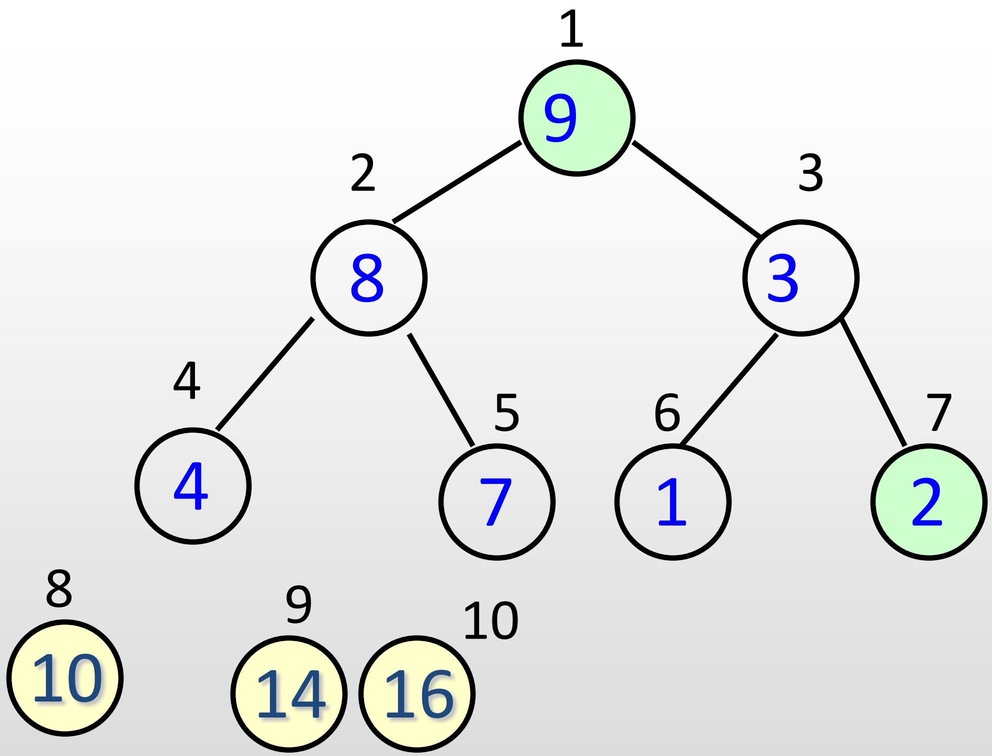


Heapify(A,1,7)

4th pass: i=7

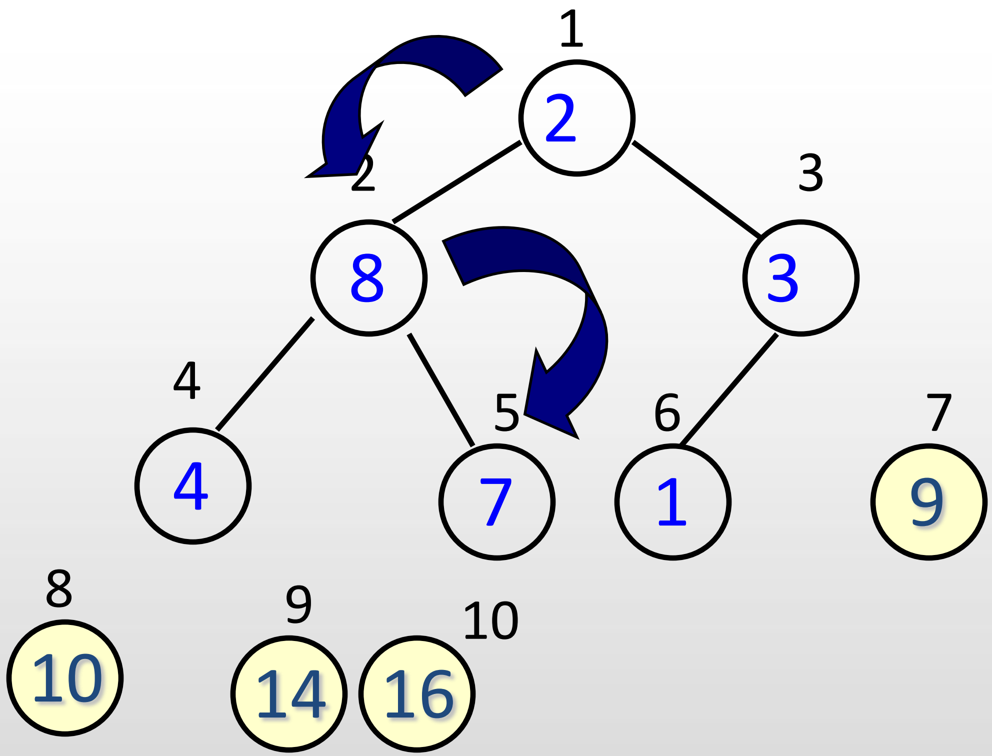


4th pass: i=7



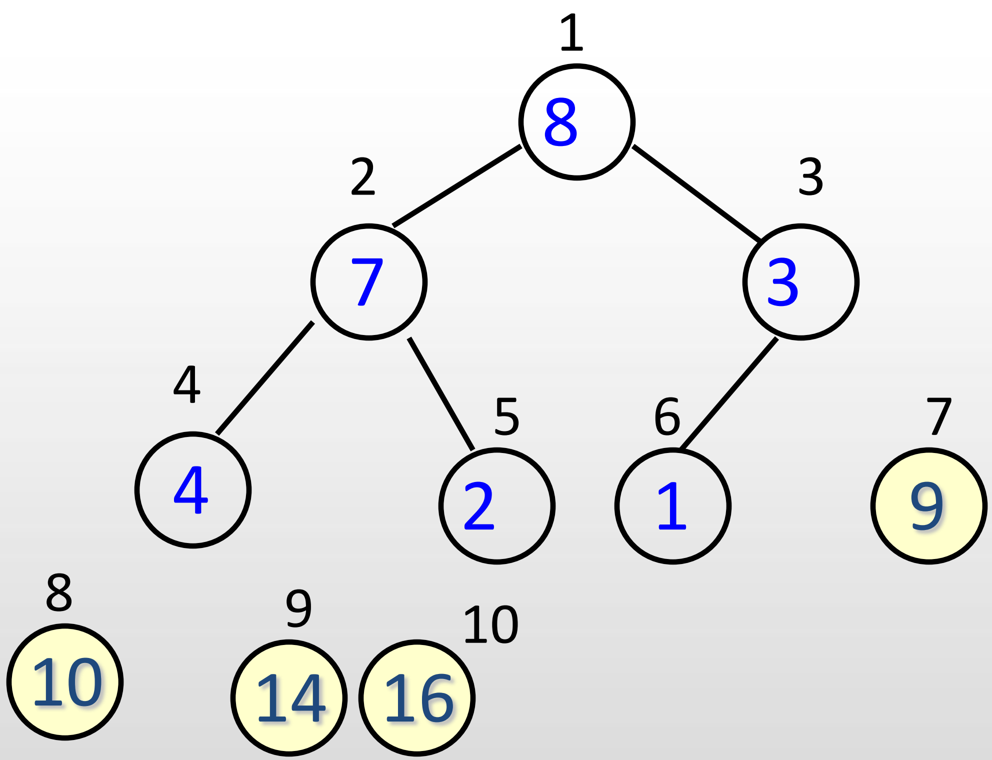
swap(A[1],A[7])

4th pass: i=7

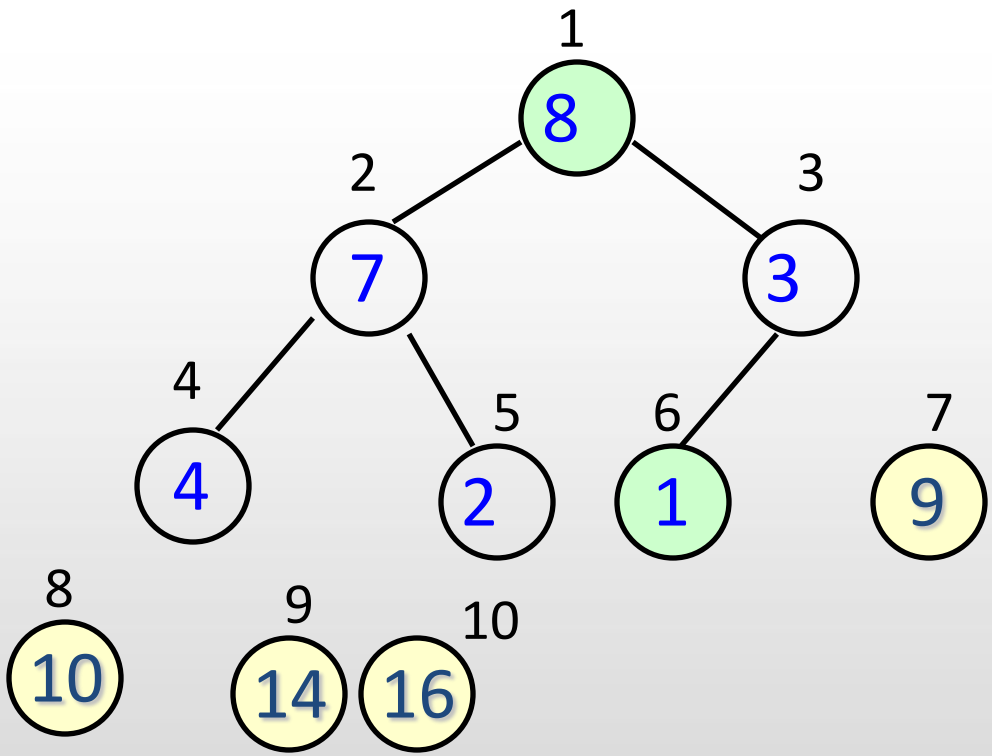


Heapify(A,1,6)

5th pass: i=6

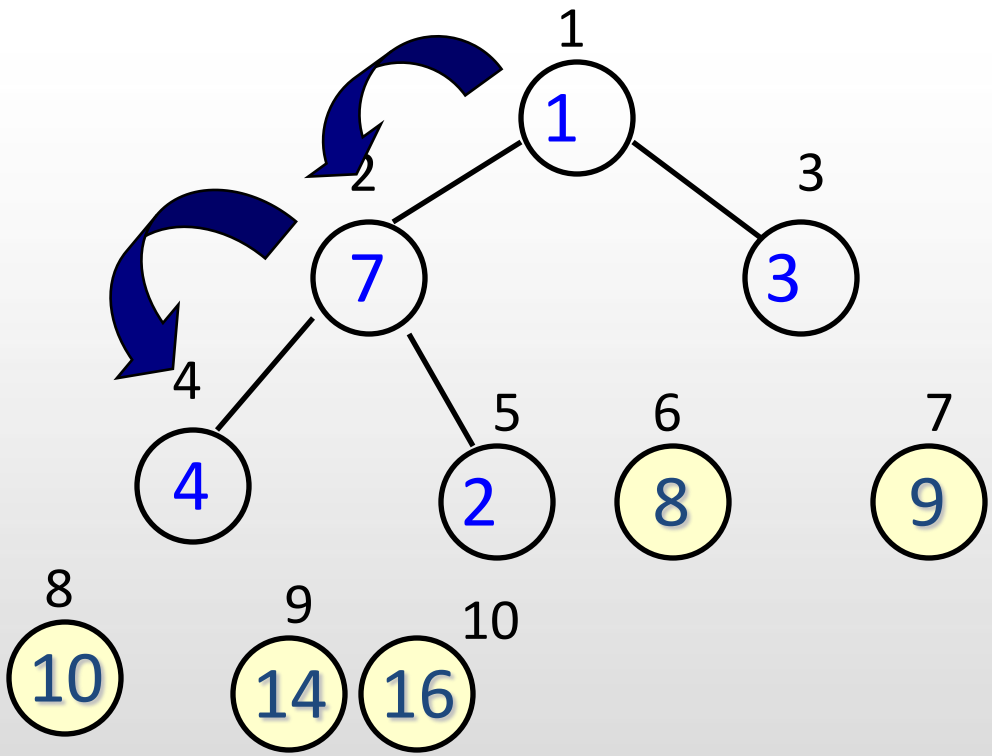


5th pass: i=6



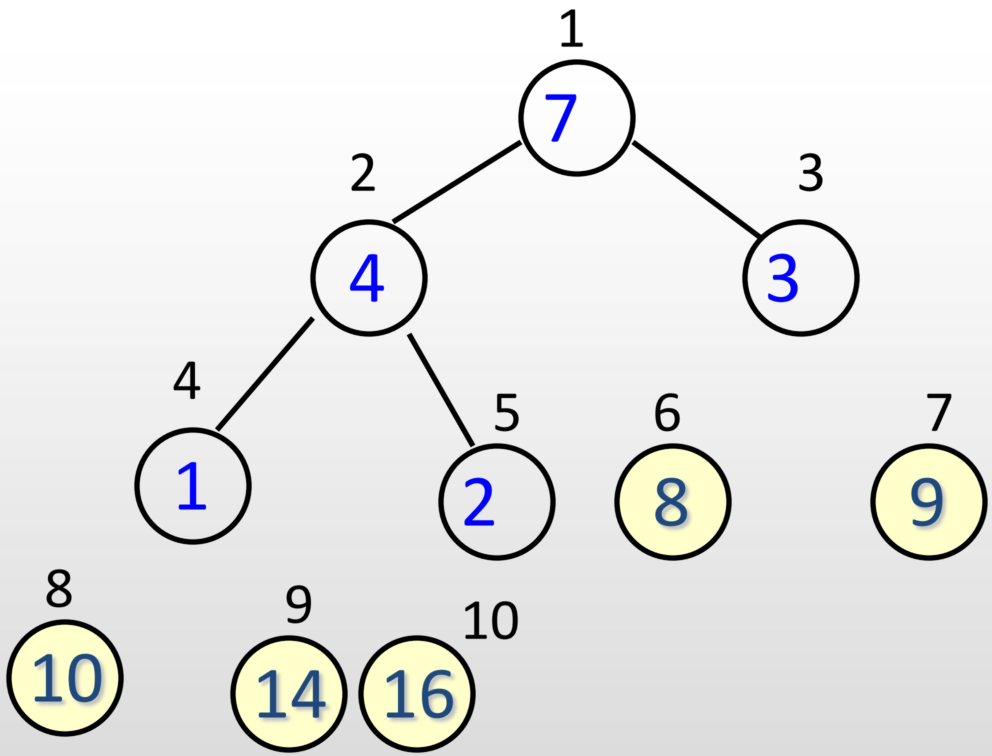
swap(A[1],A[6])

5th pass: i=6

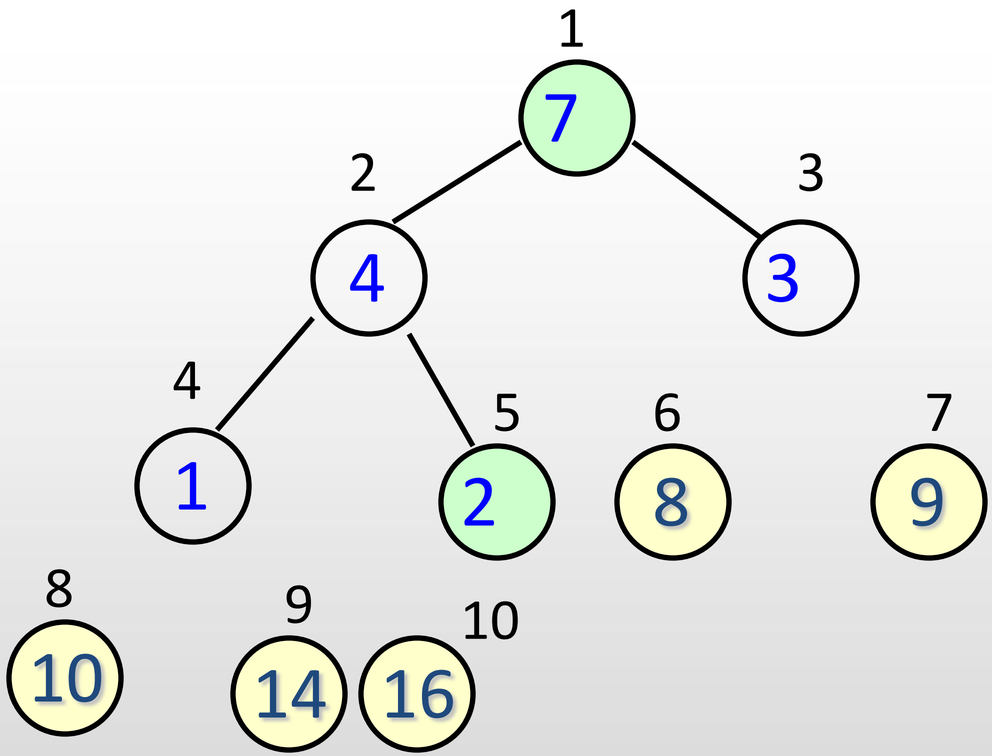


Heapify(A,1,5)

6th pass: i=5

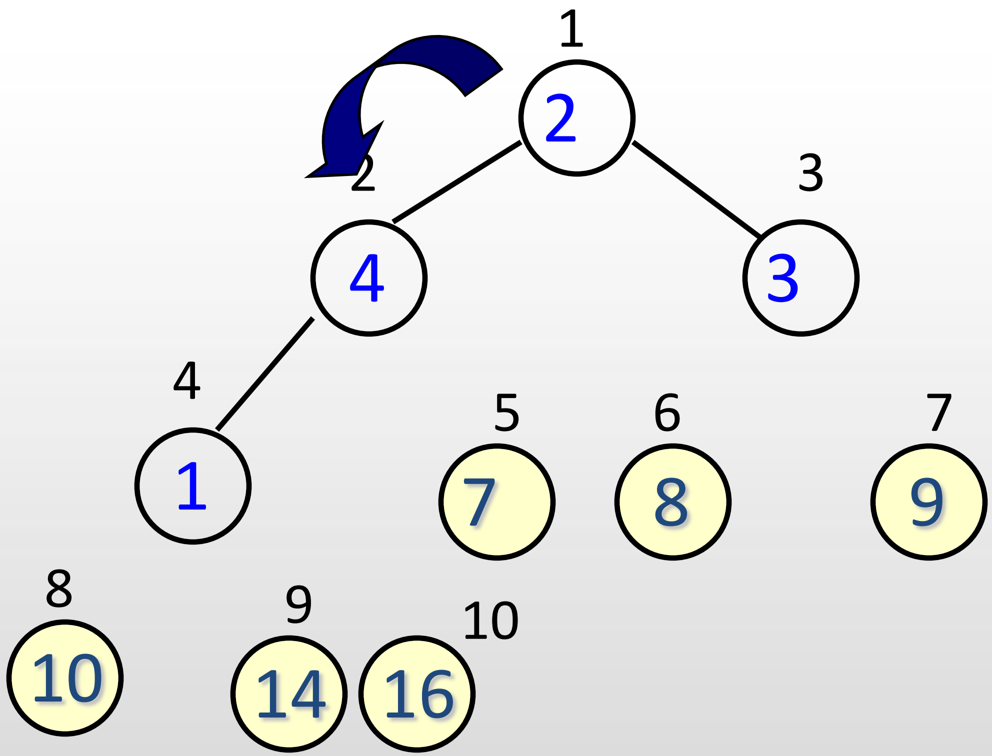


6th pass: i=5



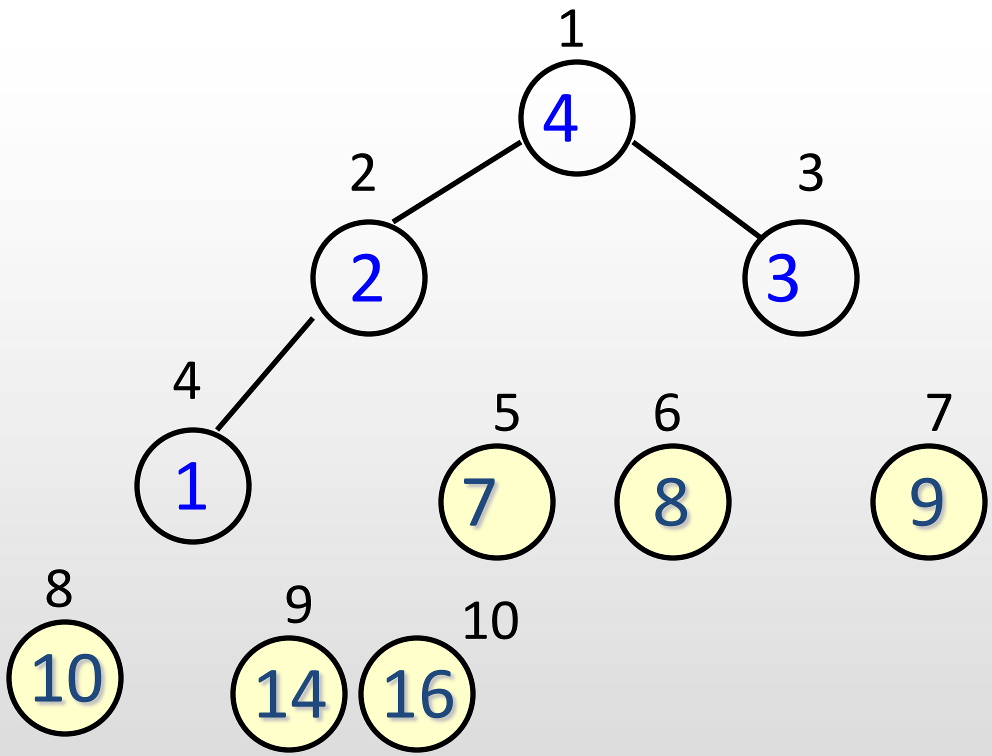
swap(A[1],A[5])

6th pass: i=5

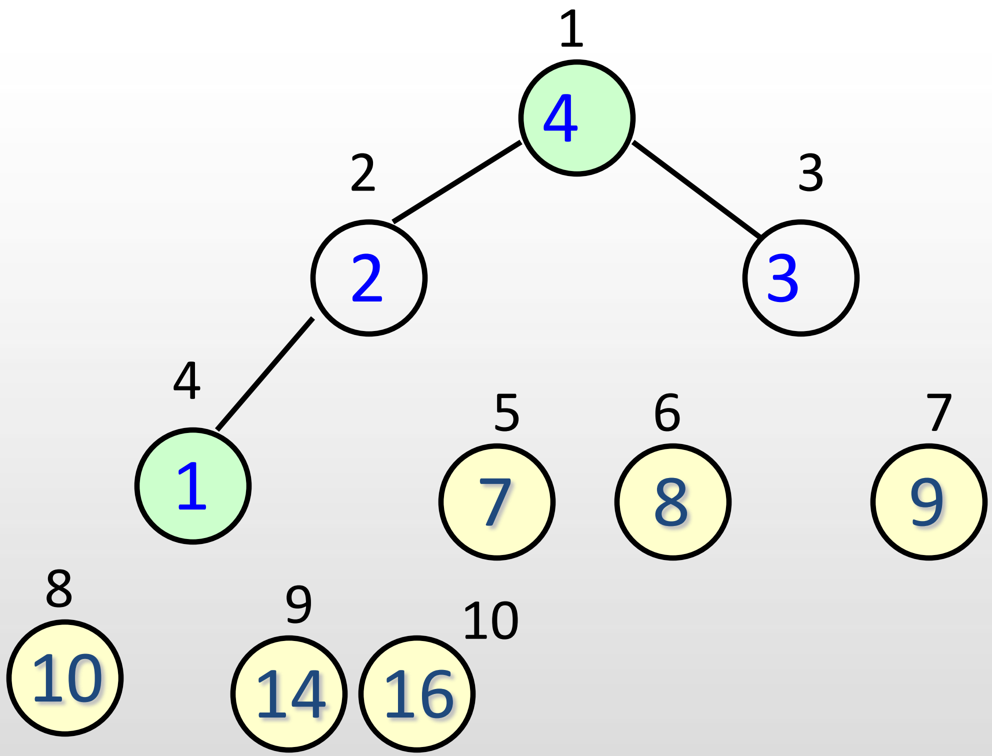


Heapify(A,1,4)

7th pass: i=4

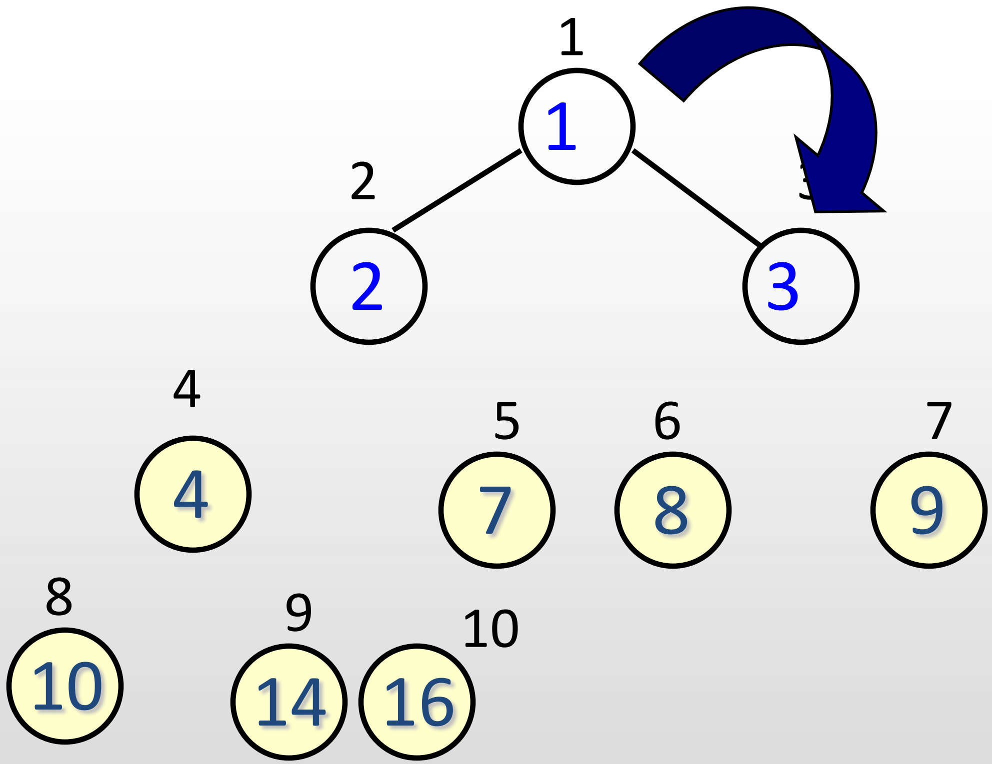


7th pass: i=4



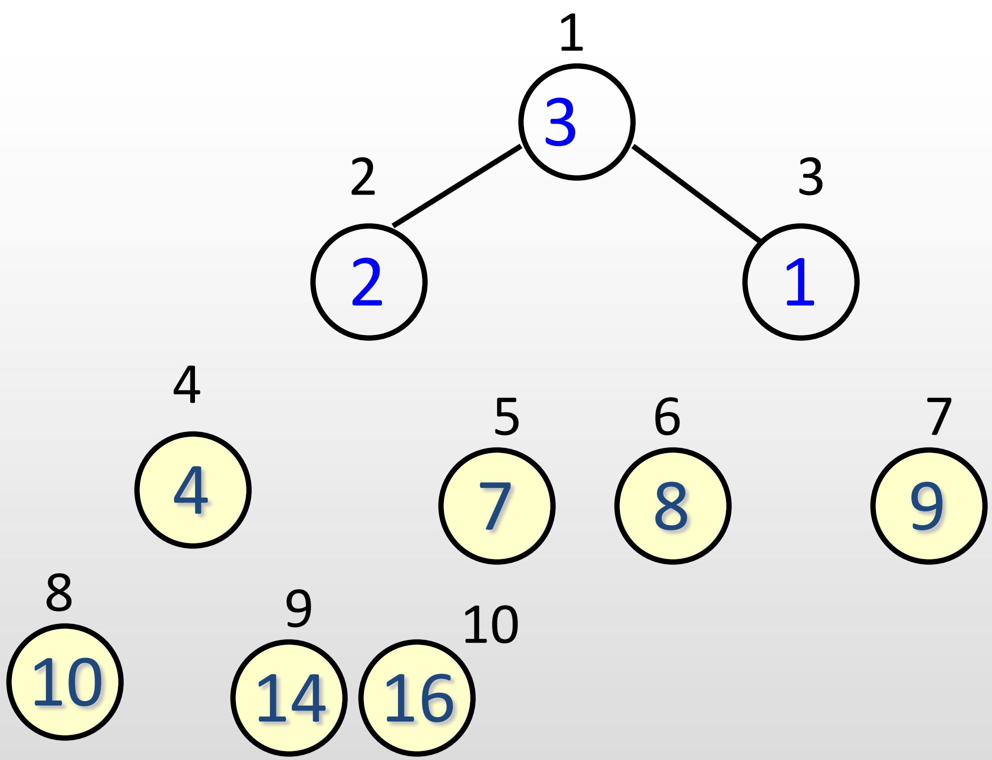
swap(A[1],A[4])

7th pass: i=4

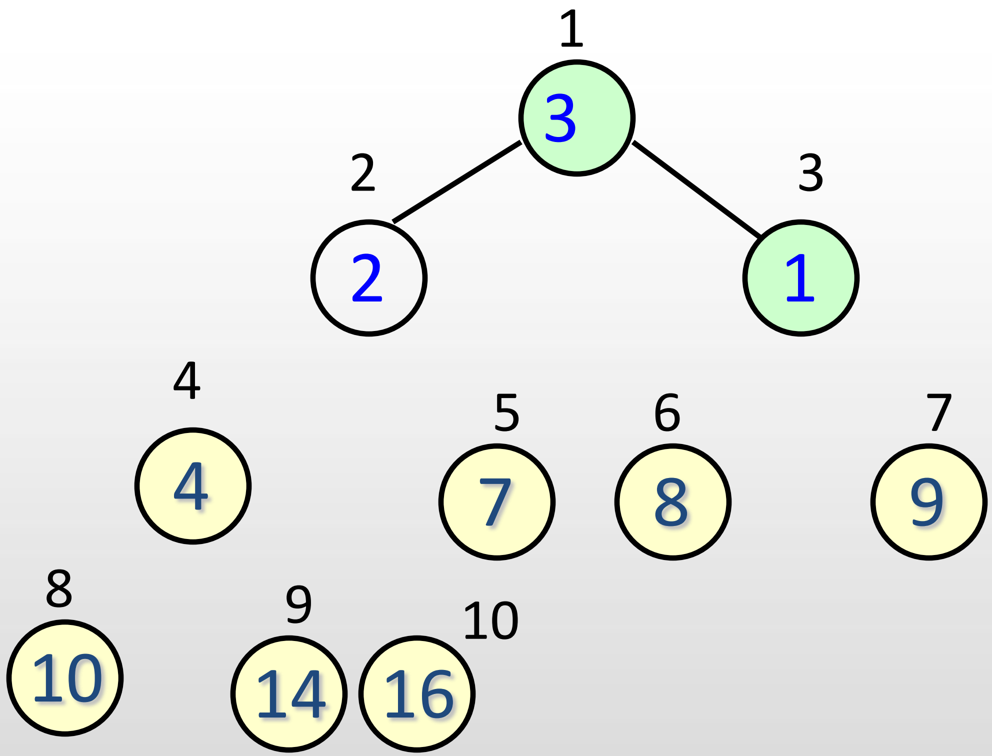


Heapify(A,1,3)

8th pass: i=3

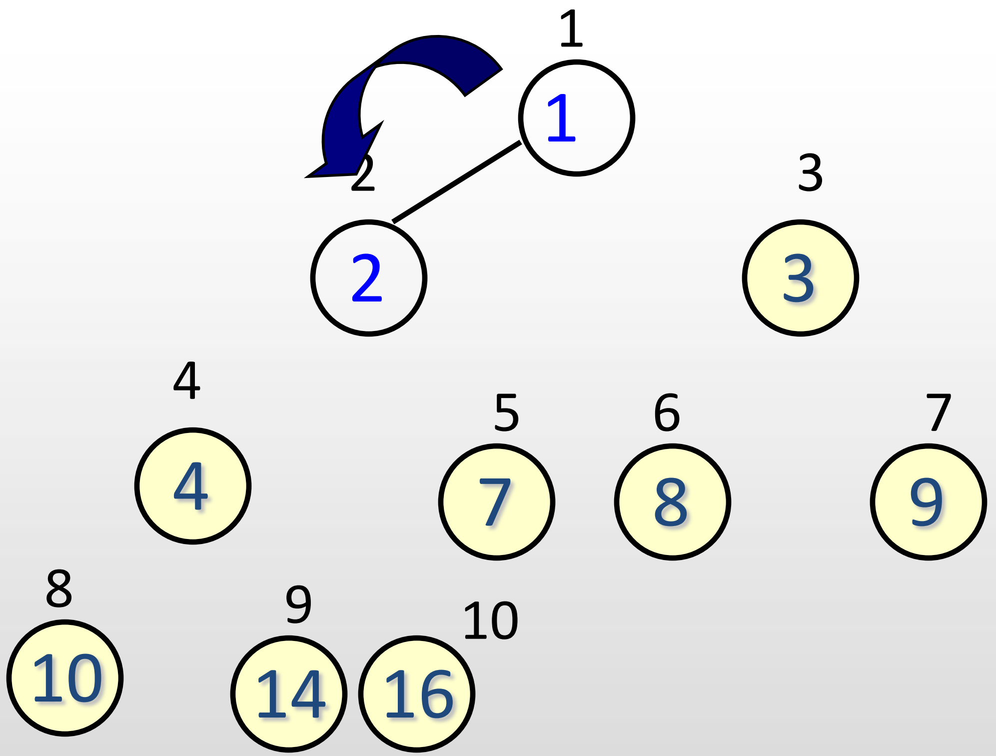


8th pass: i=3



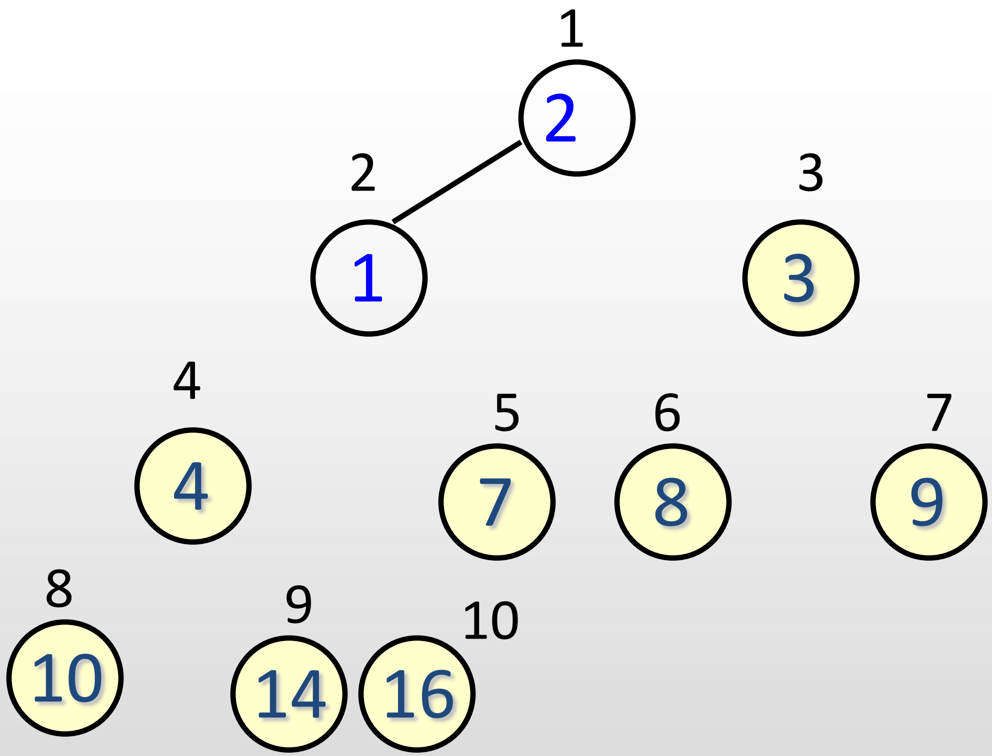
swap(A[1],A[3])

8th pass: i=3

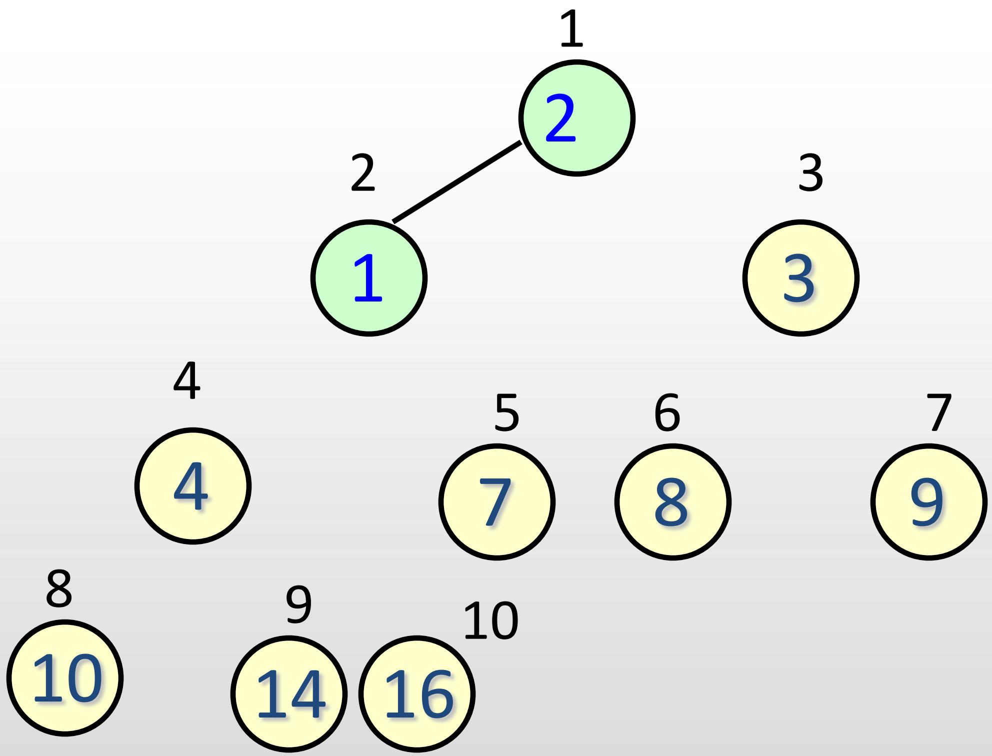


Heapify(A,1,2)

9th pass: i=2

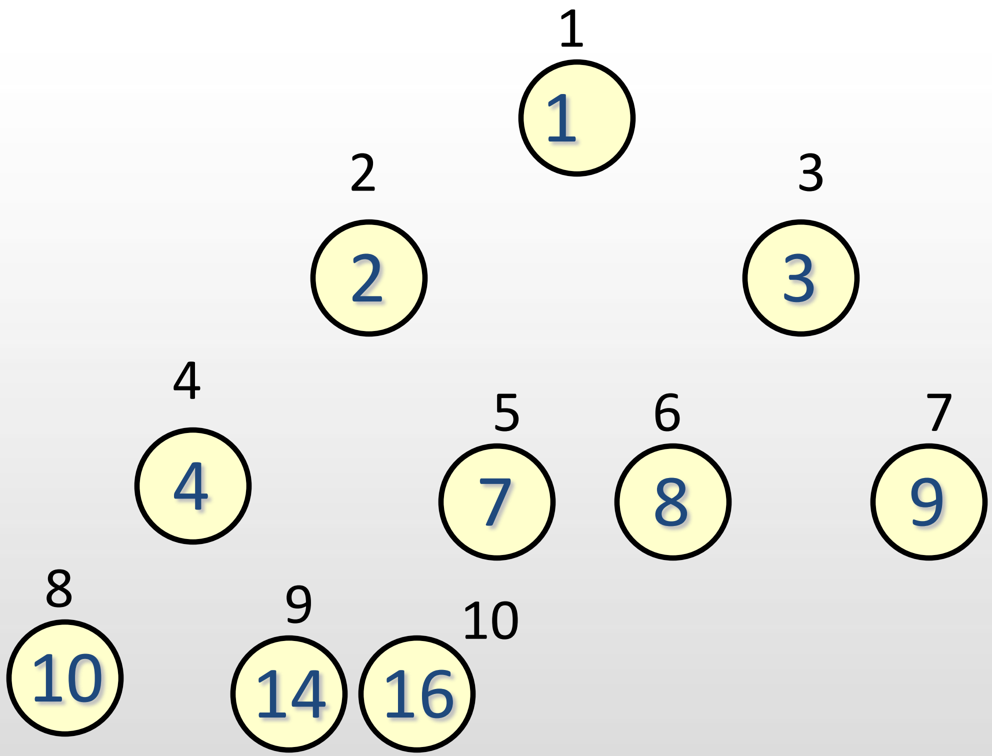


9th pass: i=2



swap(A[1],A[2])

9th pass: i=2



A									
1	2	3	4	7	8	9	10	14	16

Quiz

- Build heap:

  15, 8, 4, 3, 1, 7, 11, 10, 20, 9, 6, 5, 12, 14, 13

- In the array implementation (first element is at index 1), what is the value at:
  1. index 3?
  2. index 7?
  3. index 14?