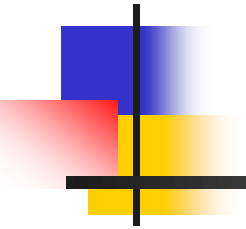
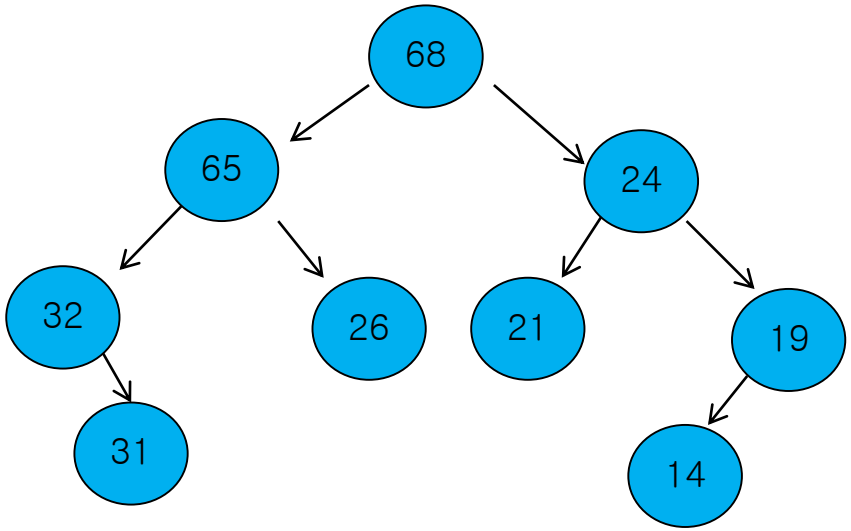


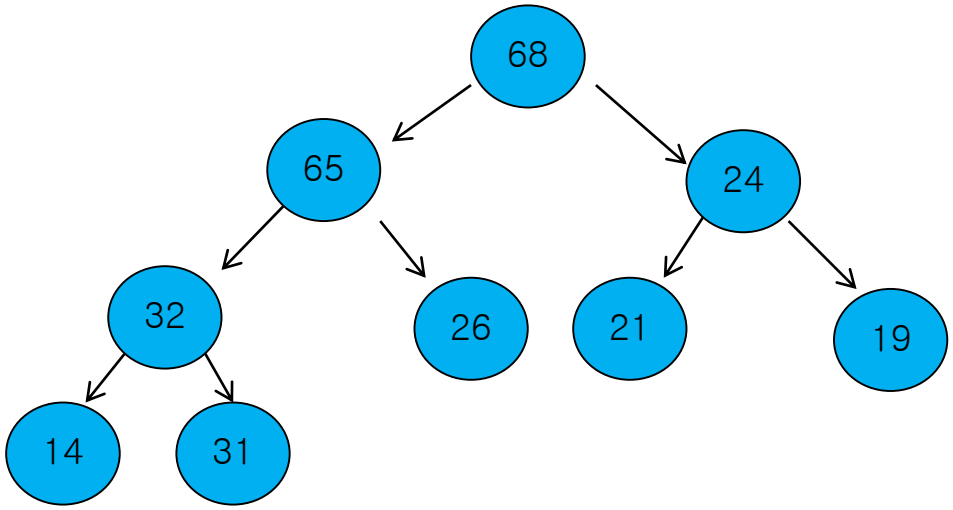
Heap

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

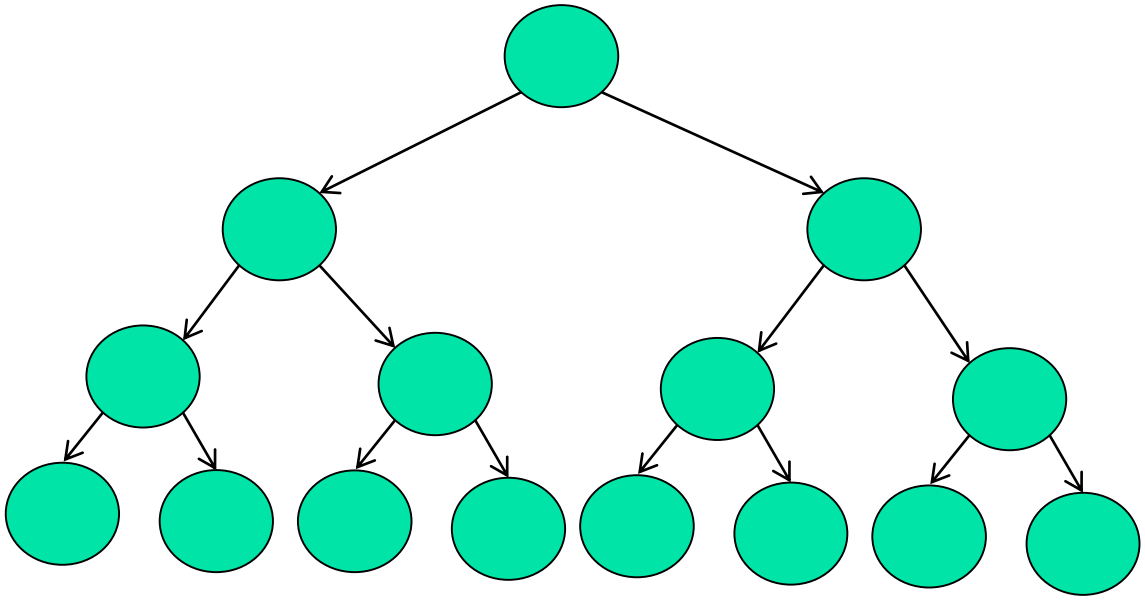
An Example of a Max Heap



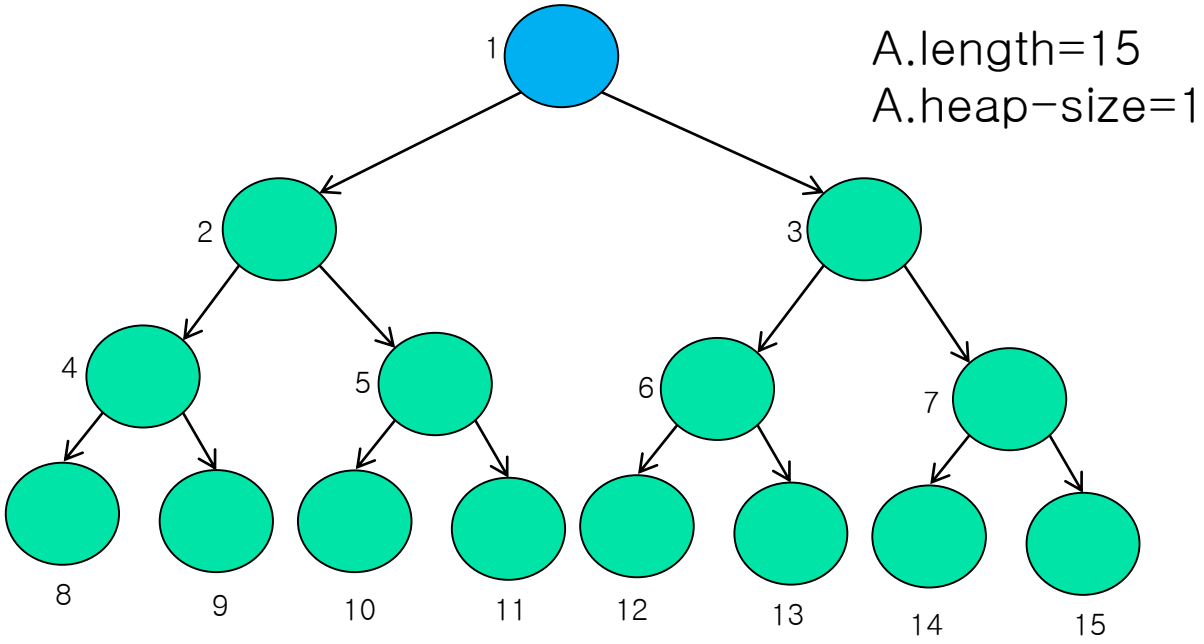
Another Example of a Max Heap



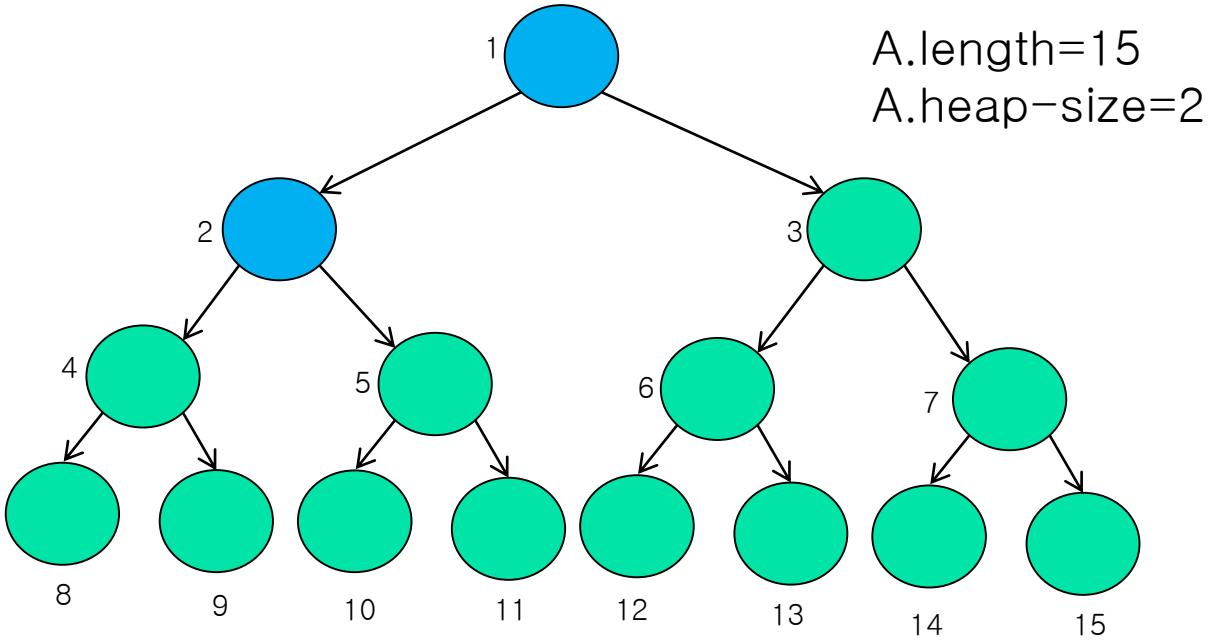
A Complete Binary Tree



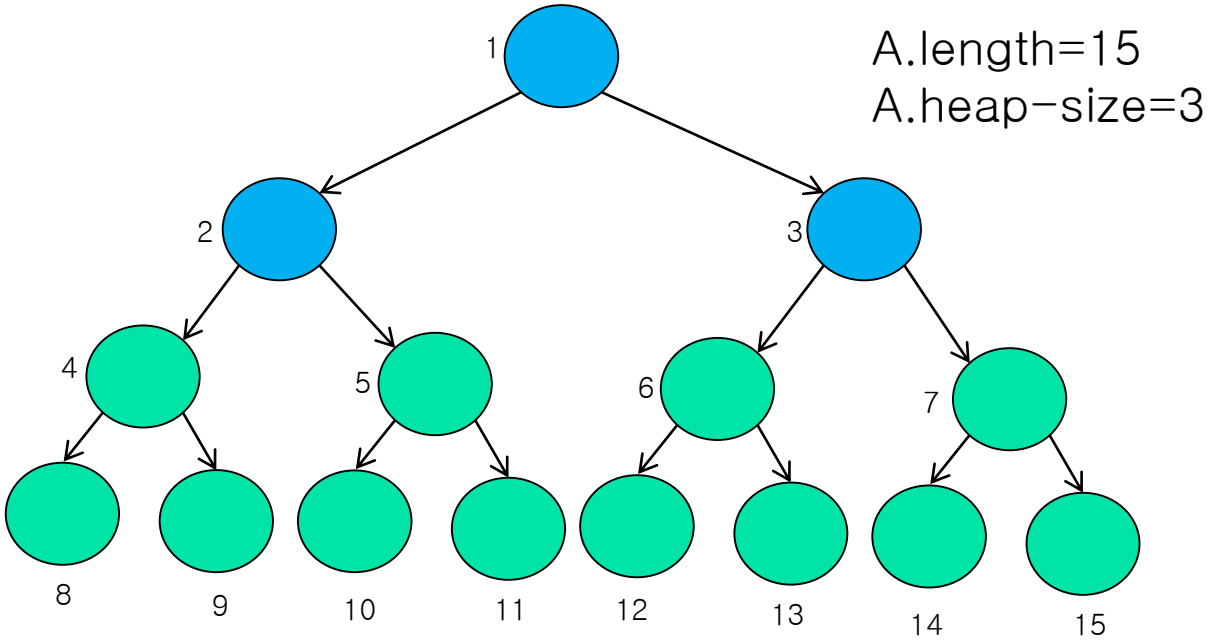
A Possible Heap



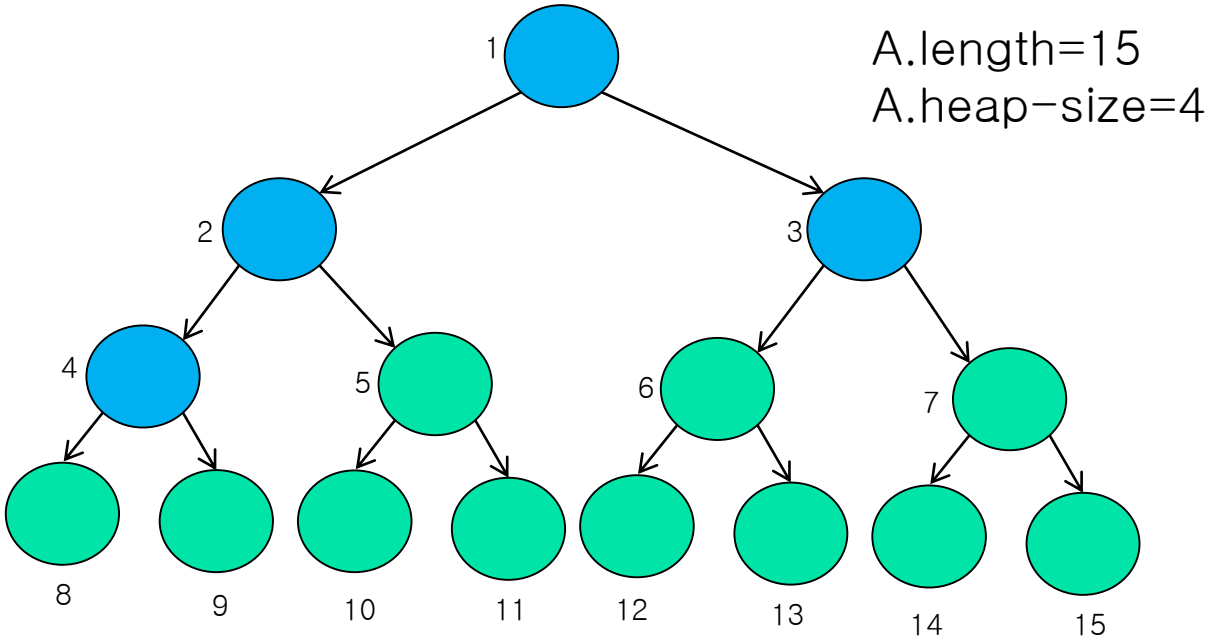
A Possible Heap



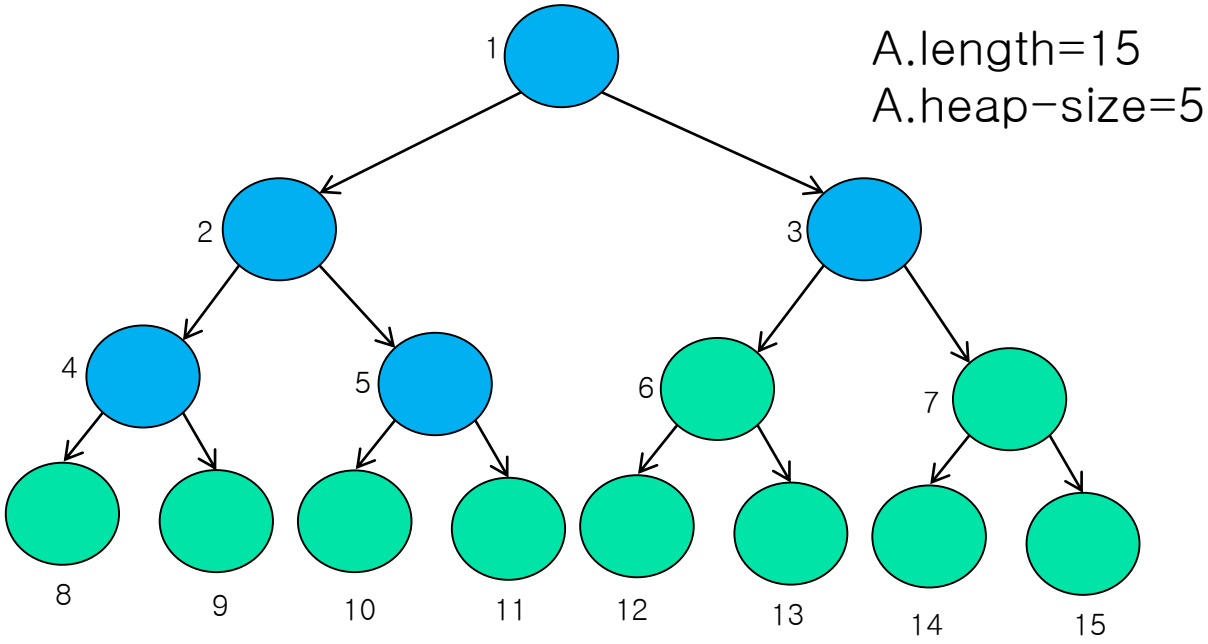
A Possible Heap



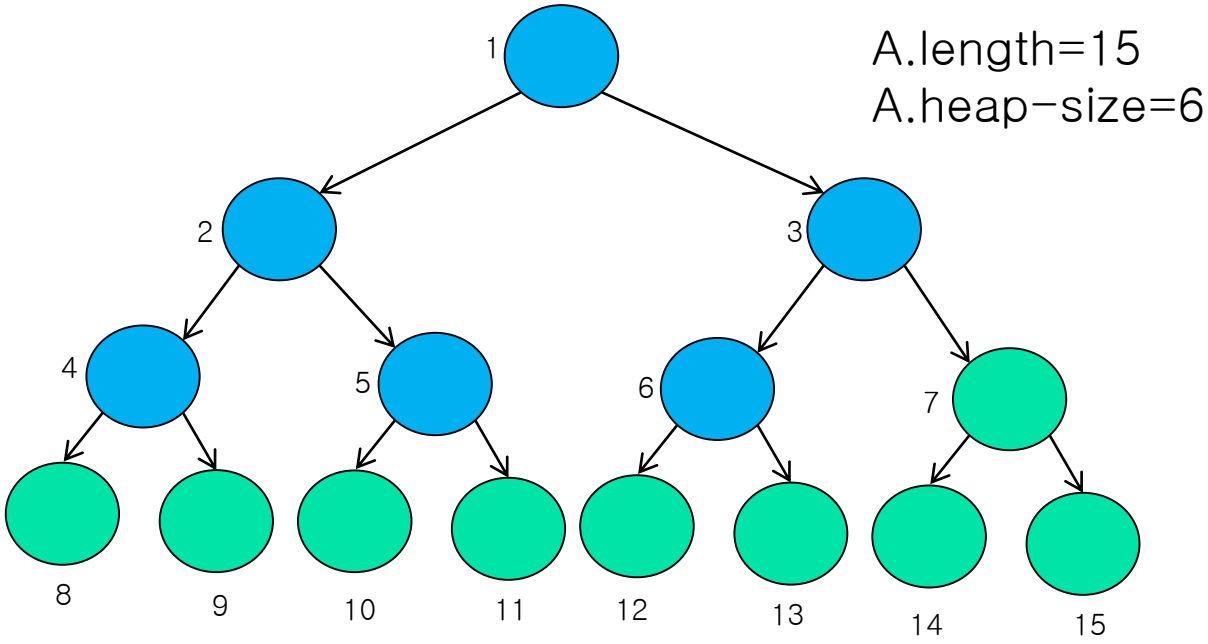
A Possible Heap



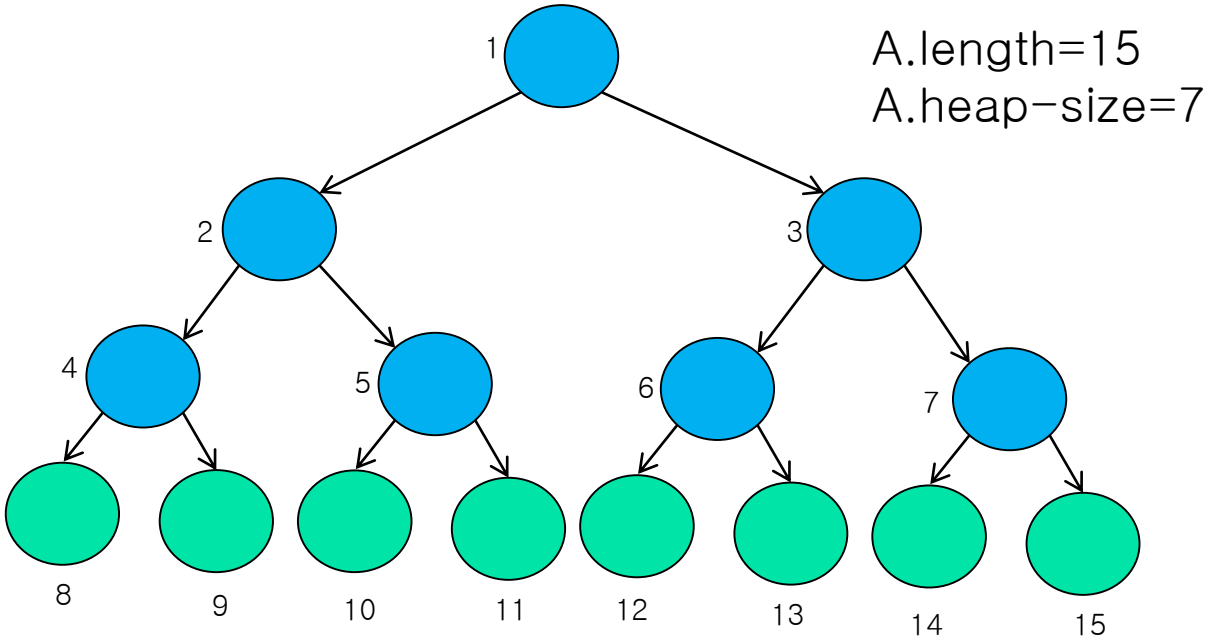
A Possible Heap



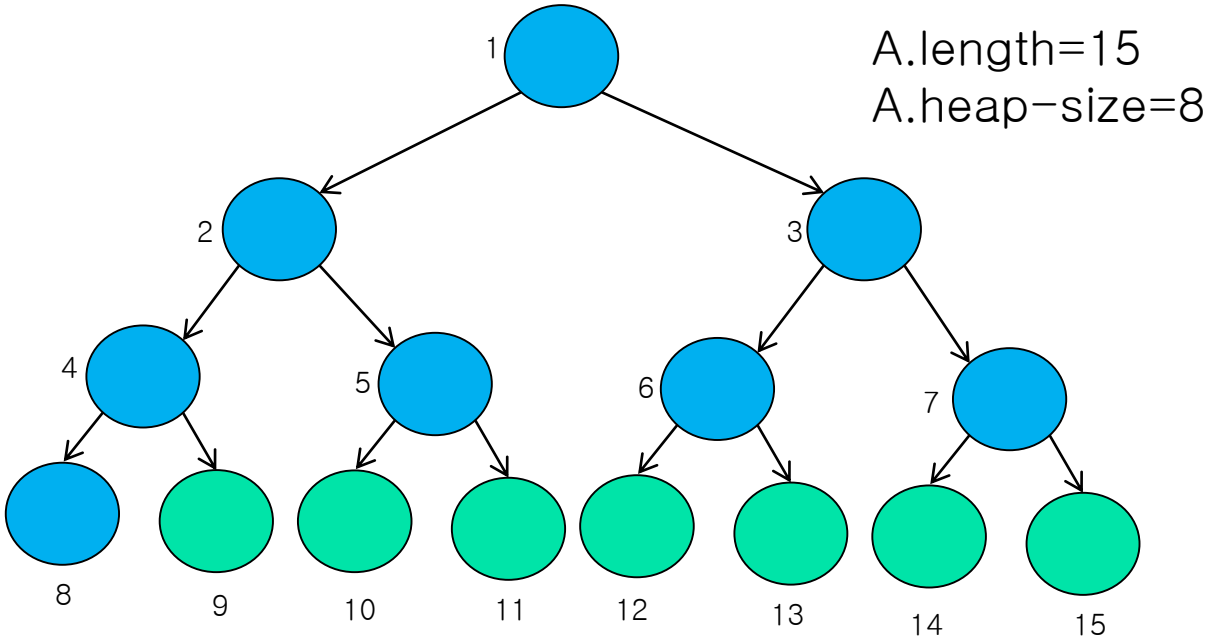
A Possible Heap

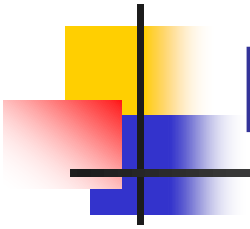


A Possible Heap



A Possible Heap





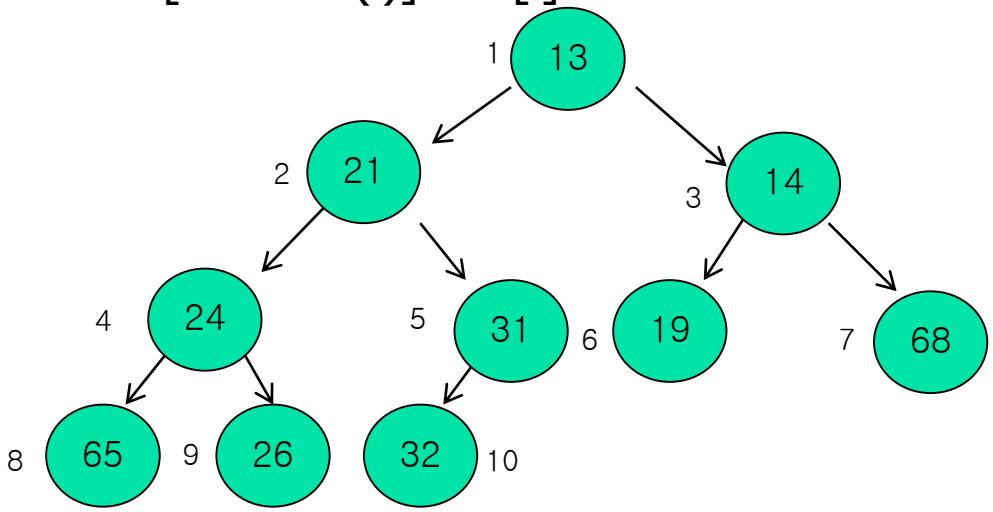
Heaps

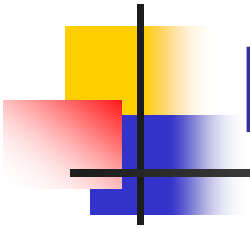
- The binary heap is an array that we can view as a nearly complete binary tree.
- Each node of the tree corresponds to an element of the array.
- The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.
- An array A that represents a heap is an object with two attributes
  - A.length: the number of elements in the array
  - A.heap-size: the number of elements in the heap
- Only the elements in A[1…A.heap-size], where 0 ≤ A.heap-size ≤ A.length, are valid elements of the heap.
- The array element A[1] is the root of the tree
- Given the index i, the indices of its parent, left child and right child are
  - LEFT(i)=2i (if 2i≤A.heap-size)
  - RIGHT(i)=2i+1 (if 2i+1≤A.heap-size)
  - PARENT(i)=$\lfloor i/2 \rfloor$

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Min Heap

- For every node i other than the root, A[PARENT(i)] ≤ A[i]





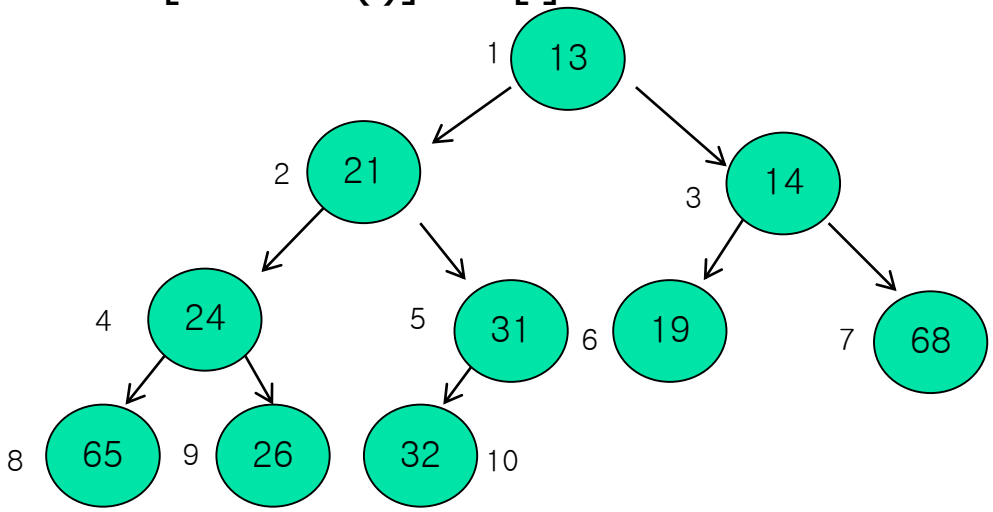
Heap

- The height of a node in a heap is the number of edges on the longest simple downward path from the node to a leaf.
- The height of the heap is the height of its root.
- Since a heap of n elements is based on a complete binary tree, its height is O(lg n)

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Min Heap

- For every node i other than the root, A[PARENT(i)] ≤ A[i]



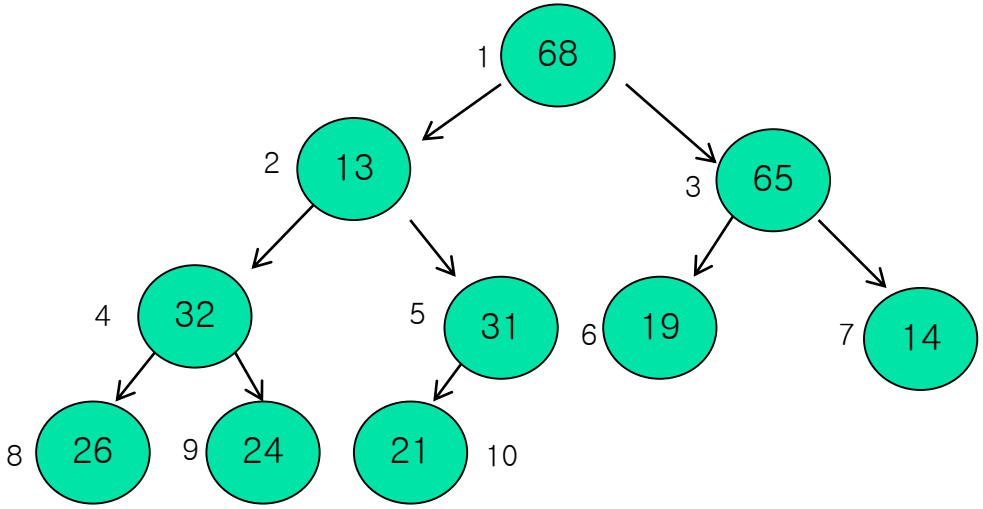
MAX-HEAP

- The binary trees rooted at LEFT(i) and RIGHT(i) are max heaps and A[i] might violate the max-heap property.
- MAX-HEAPIFY lets the value at A[i] float down so that the subtree rooted at index i becomes a max-heap.
- MAX-HEAPIFY(A, i)

```
1.    L  = Left(i)
2.    R  = Right(i)
3.    if L ≤ A.heap-size and A[L] > A[i]
4.       largest = L
5.    else largest  = i
6.    if R ≤ A.heap-size and A[R] > A[largest]
7.       then largest = R
8.    if largest ≠ i
9.       exchange A[i] with A[largest]
10.      MAX-HEAPIFY(A, largest)
```

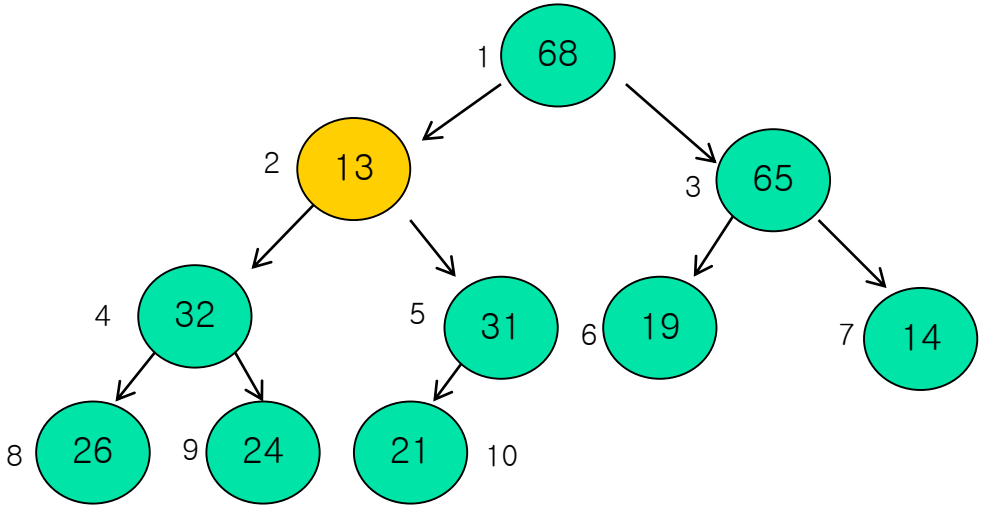
Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Action of MAX-HEAPIFY(A,2)



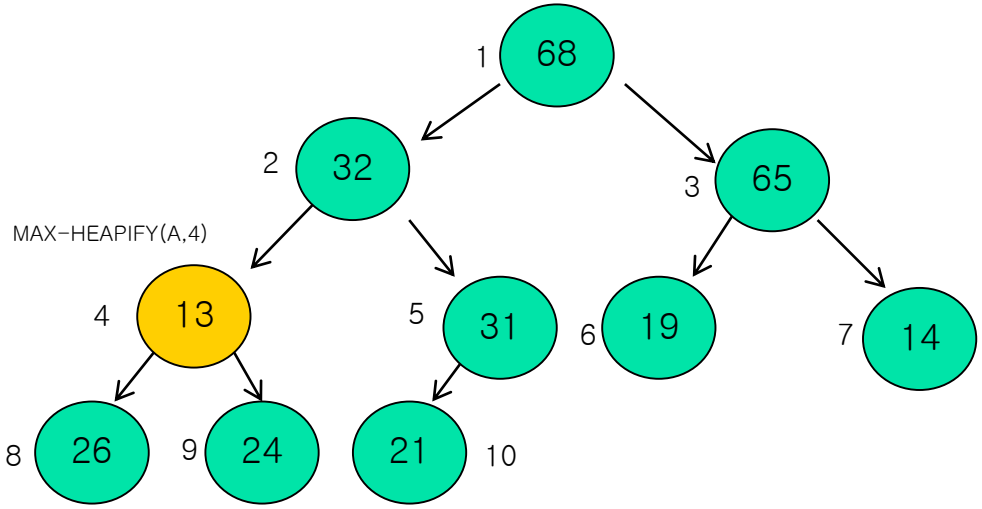
array		**68**	**13**	**65**	**32**	**31**	**19**	**14**	**26**	**24**	**21**		
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Action of MAX-HEAPIFY(A,2)



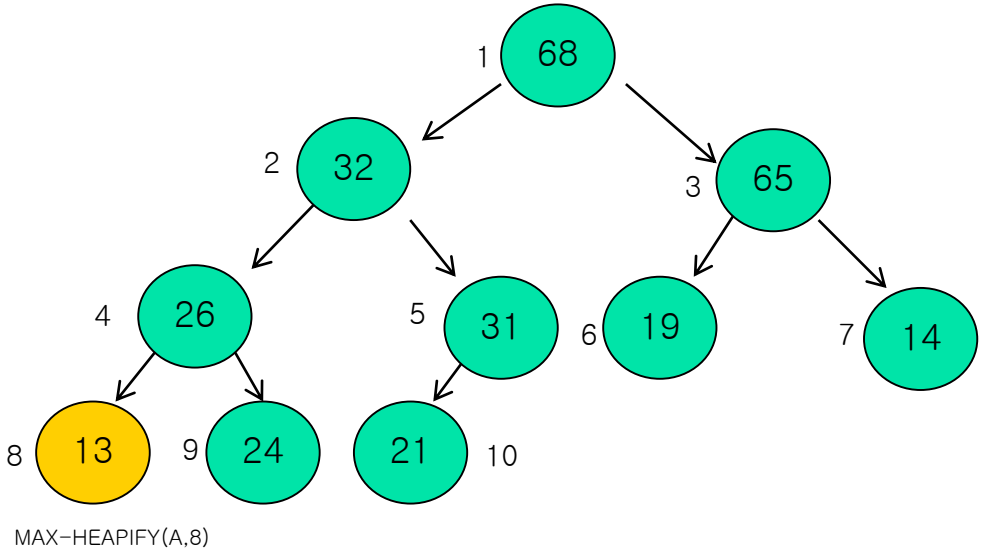
array		68	13	65	32	31	19	14	26	24	21		
index	0	1	2	3	4	5	6	7	8	9	10	11	

Action of MAX-HEAPIFY(A,2)



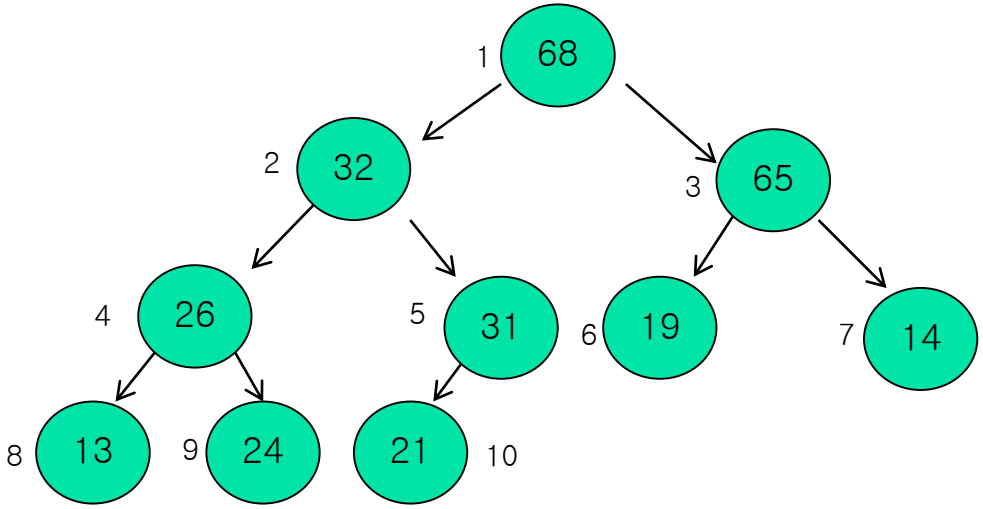
array		68	32	65	13	31	19	14	26	24	21	
index	0	1	2	3	4	5	6	7	8	9	10	11

Action of MAX-HEAPIFY(A,2)



array		68	32	65	26	31	19	14	13	24	21		
index	0	1	2	3	4	5	6	7	8	9	10	11

Action of MAX-HEAPIFY(A,2)



array		**68**	**32**	**65**	**13**	**31**	**19**	**14**	**26**	**24**	**21**		
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

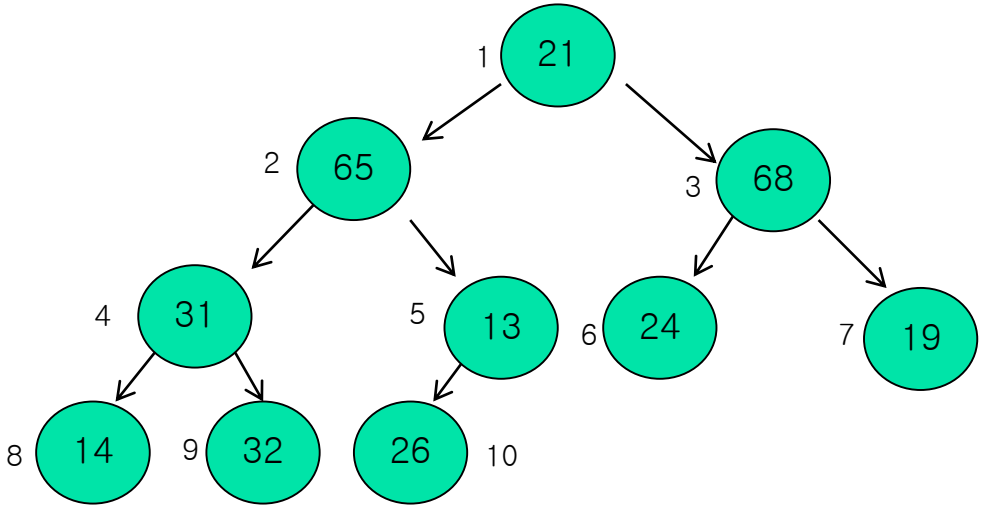
BUILD-MAX-HEAP(A)

- We can use MAX-HEAPIFY in a bottom-up manner to convert array A[1..n], where n = A.length, into a max-heap

BUILD-MAX-HEAP(A)

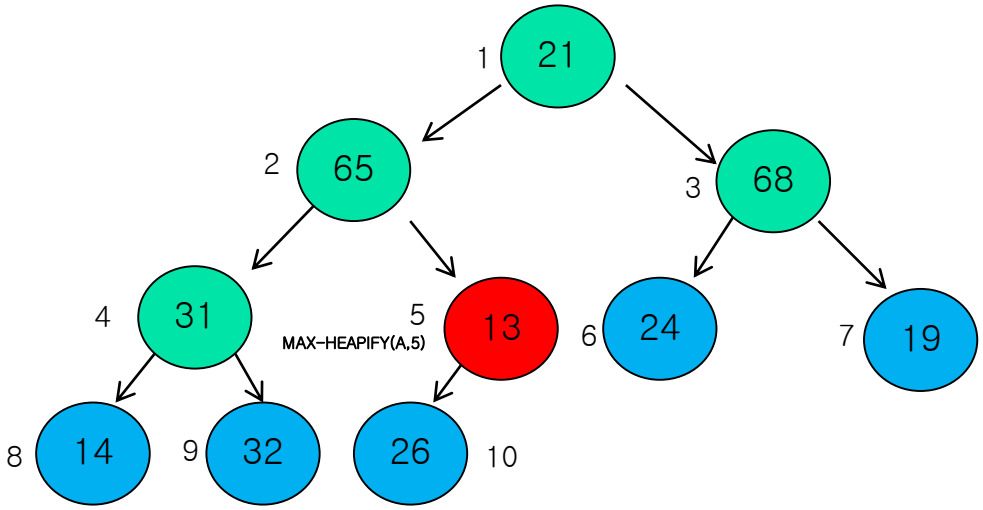
1. A.heap-size = A.length
2. **for** i = $\lfloor A.length/2 \rfloor$ **downto** 1
3. MAX-HEAPIFY(A, i)

BUILD-MAX-HEAP



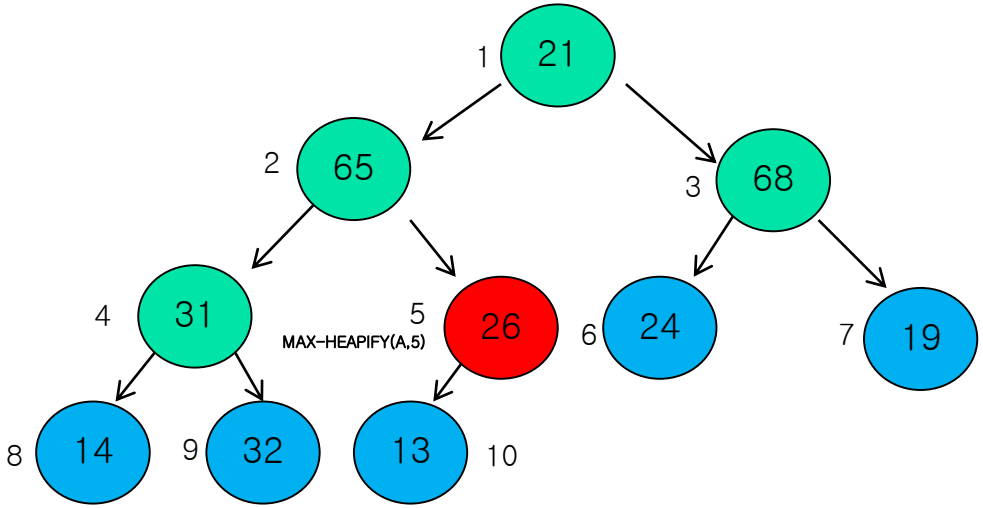
array		**21**	**65**	**68**	**31**	**13**	**24**	**19**	**14**	**32**	**26**		
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

BUILD-MAX-HEAP



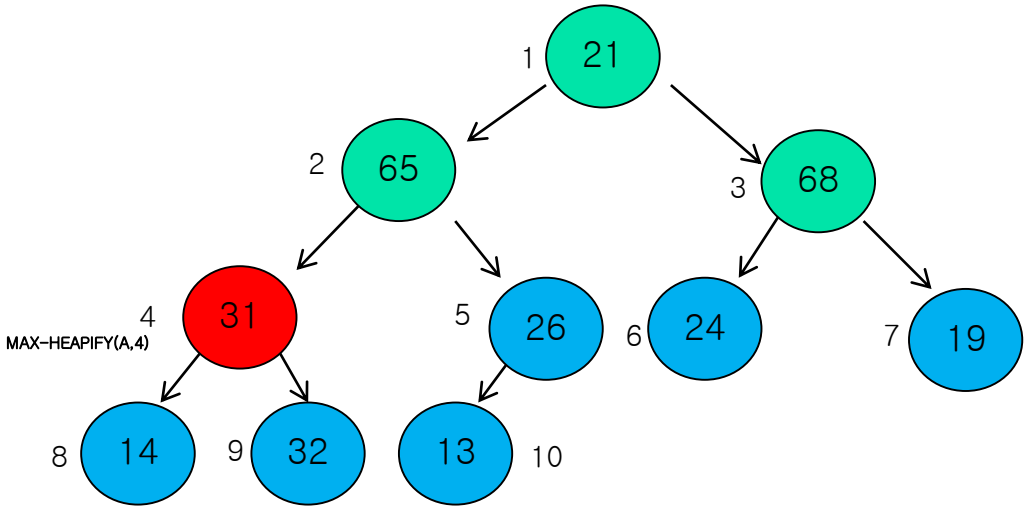
array		21	65	68	31	13	24	19	14	32	26		
index	0	1	2	3	4	5	6	7	8	9	10	11

BUILD-MAX-HEAP



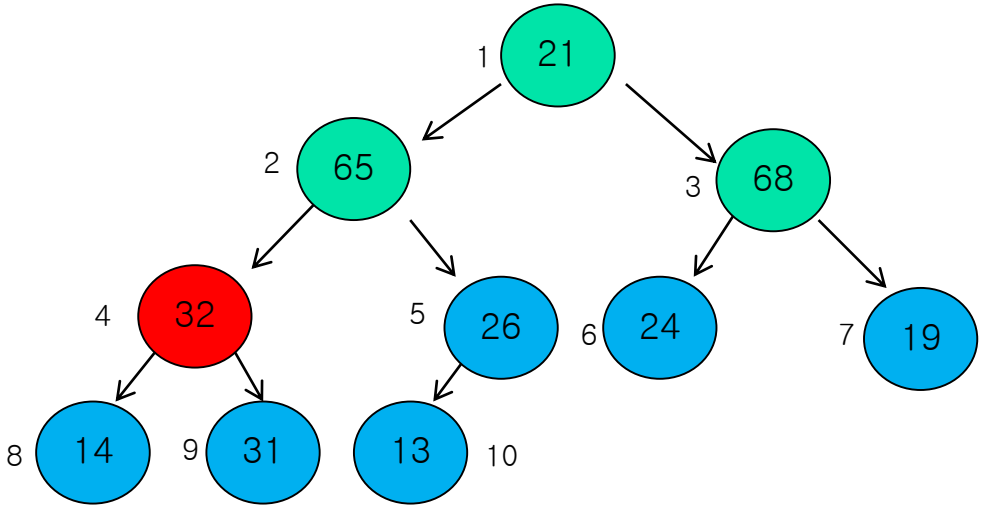
array		21	65	68	31	26	24	19	14	32	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

BUILD-MAX-HEAP



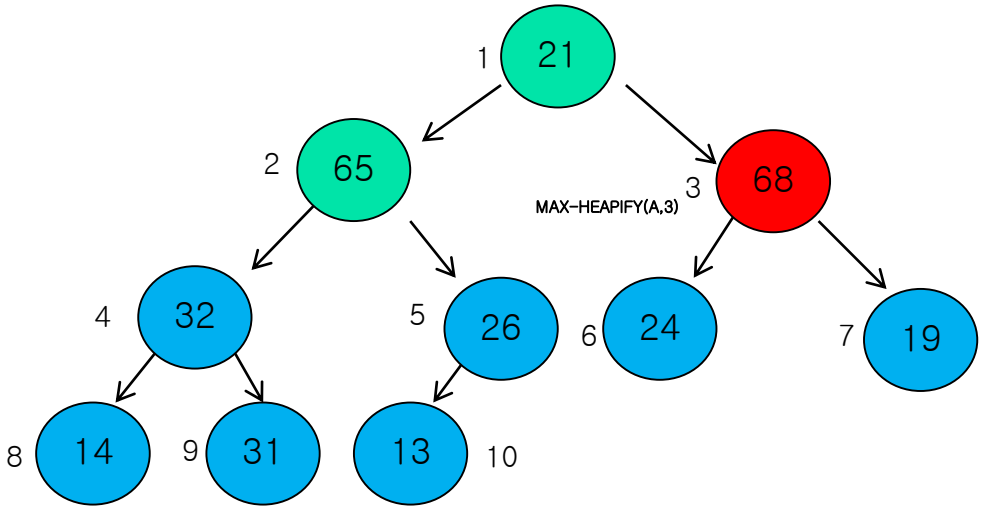
array		21	65	68	31	26	24	19	14	32	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

BUILD-MAX-HEAP



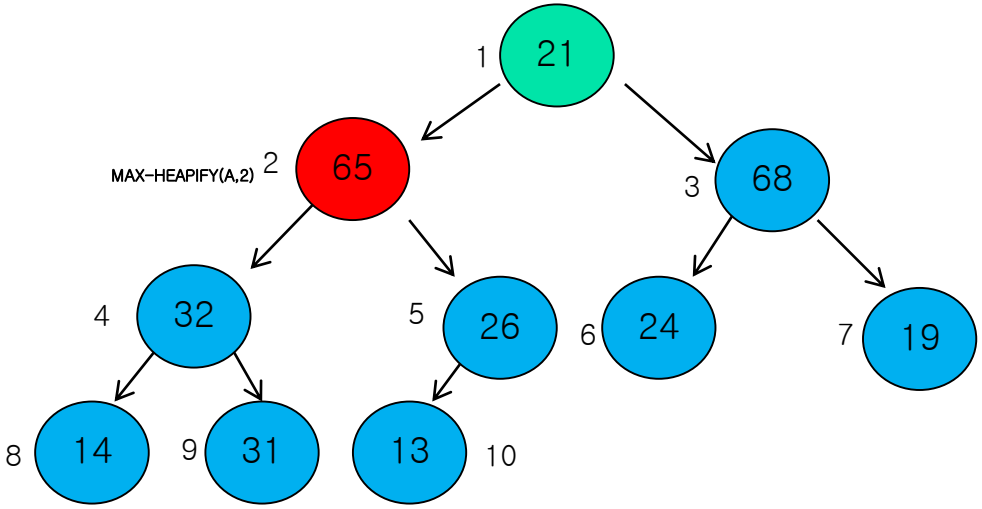
array		**21**	**65**	**68**	**32**	**26**	**24**	**19**	**14**	**31**	**13**		
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

BUILD-MAX-HEAP



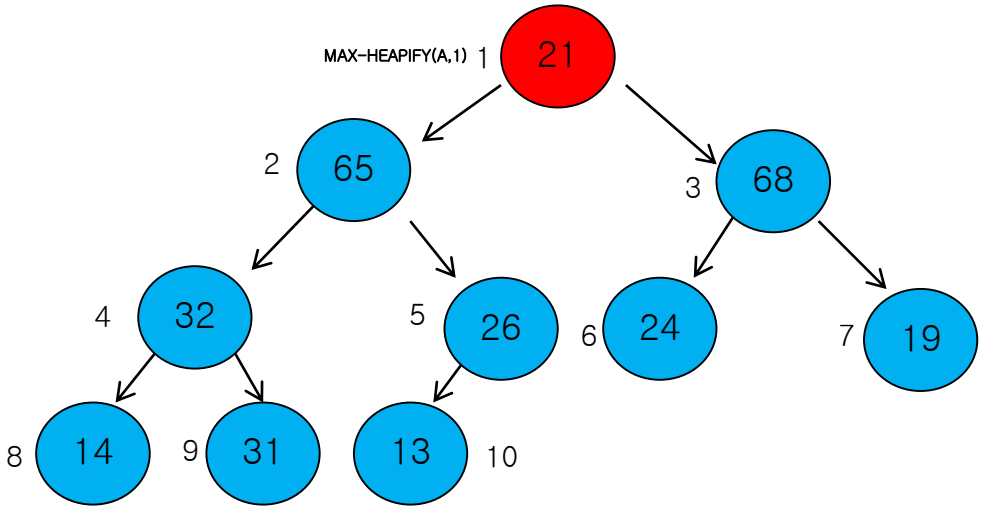
array		21	65	68	32	26	24	19	14	31	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

BUILD-MAX-HEAP



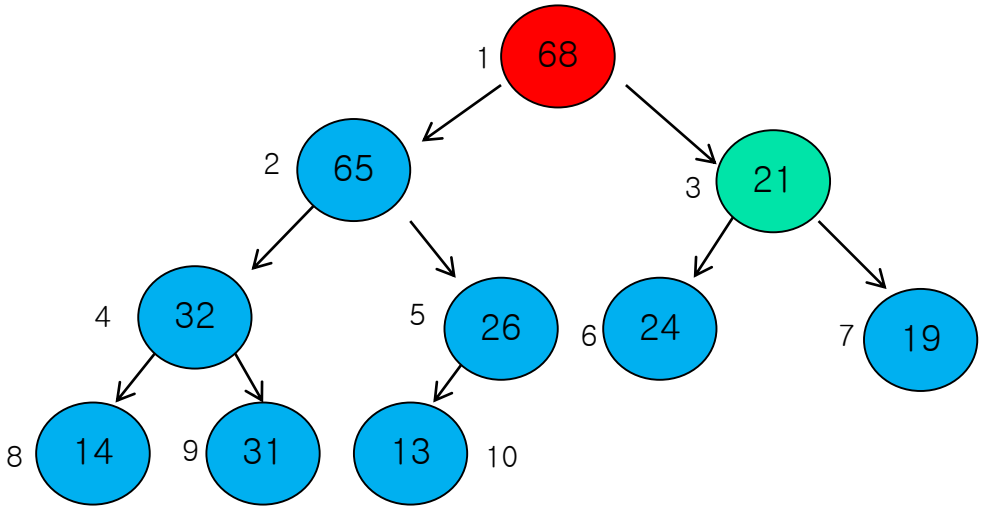
array		21	65	68	32	26	24	19	14	31	13		
index	0	1	2	3	4	5	6	7	8	9	10	11

BUILD-MAX-HEAP



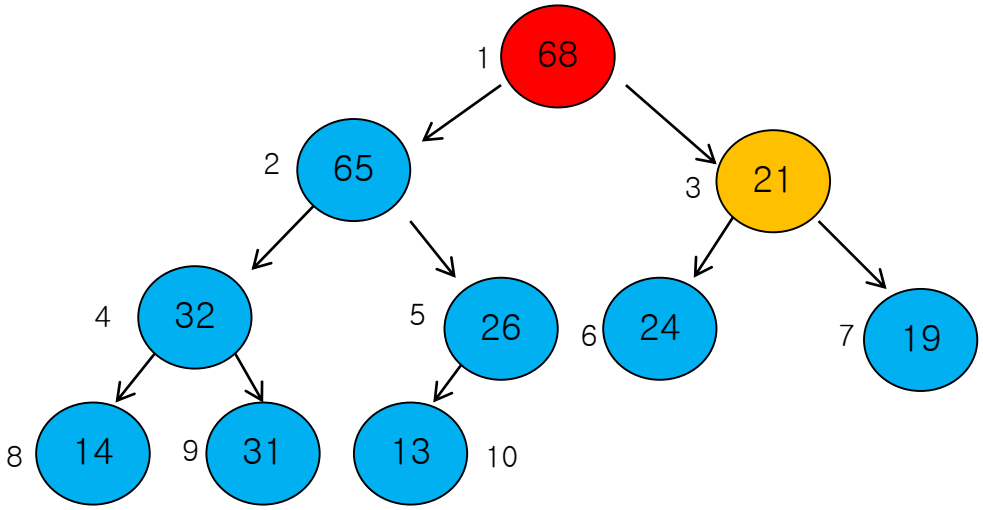
array		21	65	68	32	26	24	19	14	31	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

BUILD-MAX-HEAP



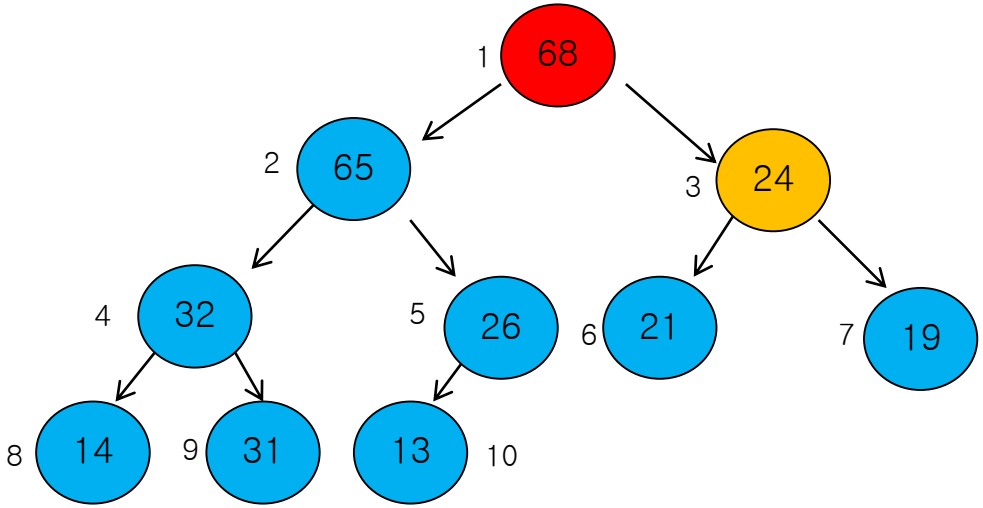
array		68	65	21	32	26	24	19	14	31	13		
index	0	1	2	3	4	5	6	7	8	9	10	11

BUILD-MAX-HEAP



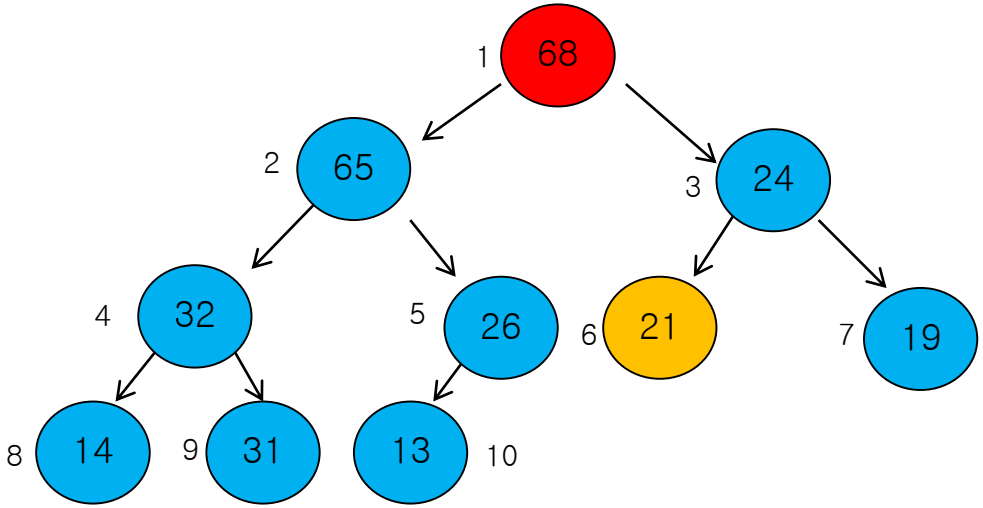
array		68	65	21	32	26	24	19	14	31	13		
index	0	1	2	3	4	5	6	7	8	9	10	11

BUILD-MAX-HEAP



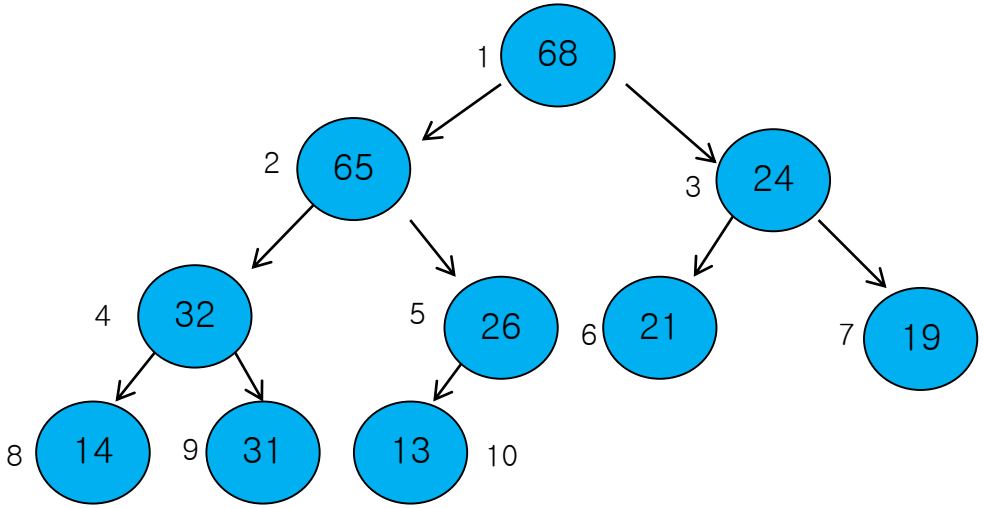
array		68	65	24	32	26	21	19	14	31	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

BUILD-MAX-HEAP



array		68	65	24	32	26	21	19	14	31	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

BUILD-MAX-HEAP



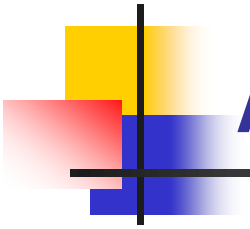
array		68	65	24	32	26	21	19	14	31	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

BUILD-MAX-HEAP(A)

- We can use MAX-HEAPIFY in a bottom-up manner to convert array A[1..n], where n = A.length, into a max-heap

BUILD-MAX-HEAP(A)

1. A.heap-size = A.length
2. **for** i = $\lfloor A.length/2 \rfloor$ **downto** 1
3. MAX-HEAPIFY(A, i)



A Simple Upper Bound

- BUILD-MAX-HEAP takes O(n lg n) time.
  - MAX-HAEP makes O(n) calls to MAX-HEAPIFY which costs O(lg n) time.

Another Analysis of Building a Heap

- Let's assume the tree is complete : $n = 2^h - 1$
  - There is one key at the height=h ,which might shift down h levels
  - There is two keys at the height=h-1 ,which might shift down h-1 levels
  - There is four key at the height=h-2 ,which might shift down h-2 levels
- Overall costs $S = h + 2(h-1) + 2^2(h-2) + \cdots + 2^{h-1}(1)$

$$2S = 2h + 2^2(h-1) + 2^3(h-2) + \cdots + 2^h(1)$$

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

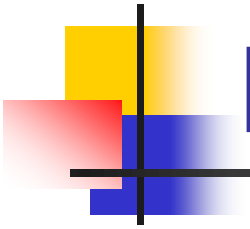
$$S = -h + (2^1 + 2^2 + \cdots + 2^h)$$

$$= -h - 1 + \left(2^0 + 2^1 + \cdots + 2^{h-1} + 2^h\right)$$

$$= 2^{h+1} - h - 2 = 2 \cdot 2^{\lg n} - \lg n - 2$$

$$\leq 2n$$

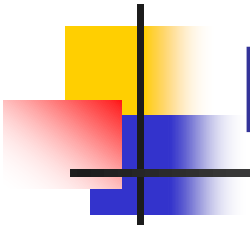
$\alpha^{\lg \beta} = \beta^{\lg \alpha}$



Heapsort

- Use BUILD-MAX-HEAP to build a max-heap.
- Since the maximum element of the array is stored at the root A[1], put it into its correct position by exchanging it with A[n].
- Restore the max-heap property by calling MAX-HEAPIFY(A, 1) to generate a max-heap in A[1..n-1].
- Repeat the steps for the max-heap of size n-1 down to a heap of size 2

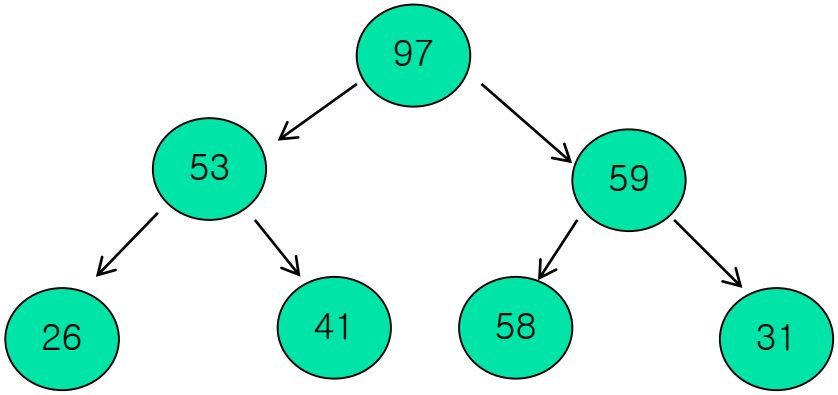
Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009



Heapsort(A)

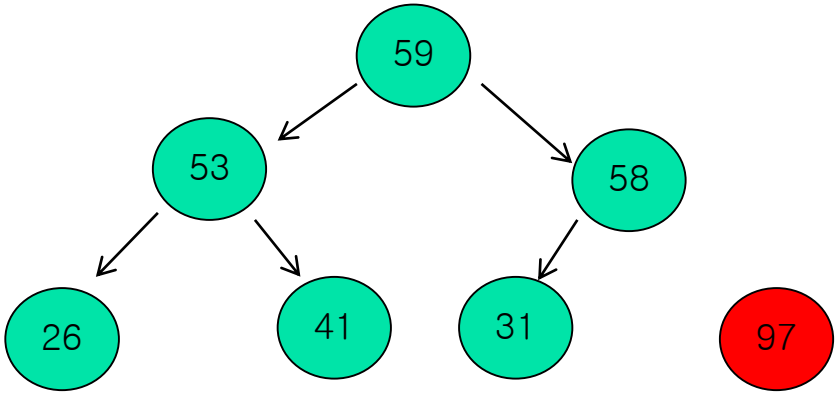
```
HAEPSORT(A)
1. BUILD-MAX-HEAP(A)
2. for i = A.length downto 2
3.     exchange A[1] with A[i]
4.     A.heap-size = A.heap-size-1
5.     MAX-HEAPIFY(A,1)
```

Operation of Heap Sort



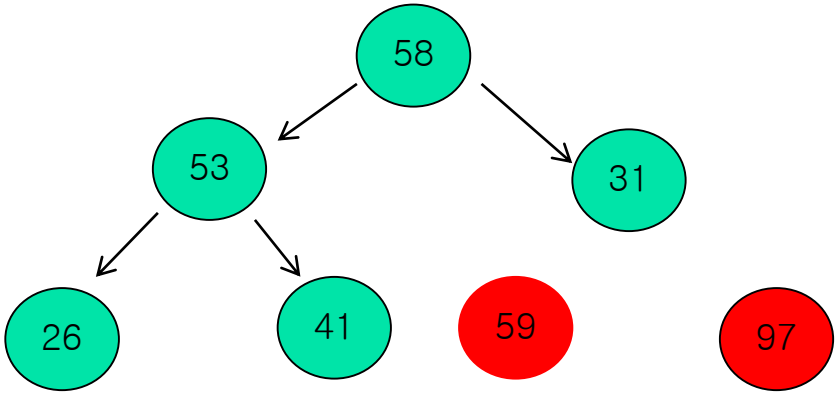
array		**97**	**53**	**59**	**26**	**41**	**58**	**31**				
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Operation of Heap Sort



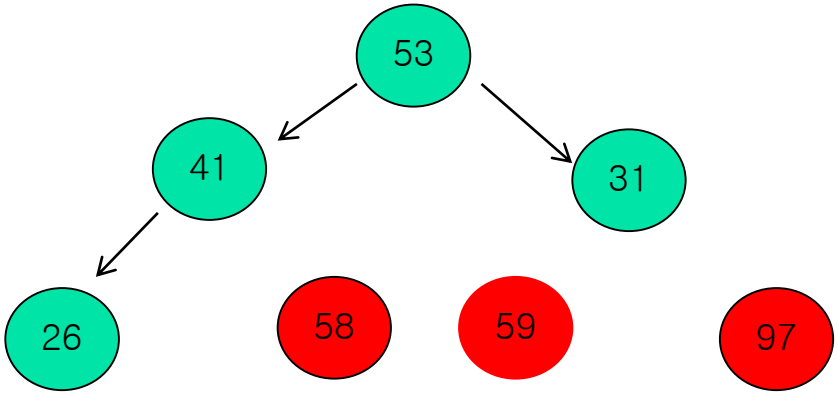
array		**59**	**53**	**58**	**26**	**41**	**31**	**97**				
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Operation of Heap Sort



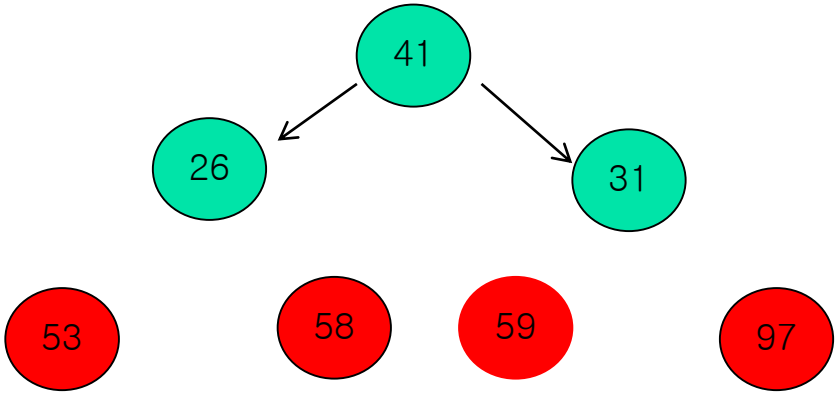
array		**58**	**53**	**31**	**26**	**41**	**59**	**97**				
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Operation of Heap Sort



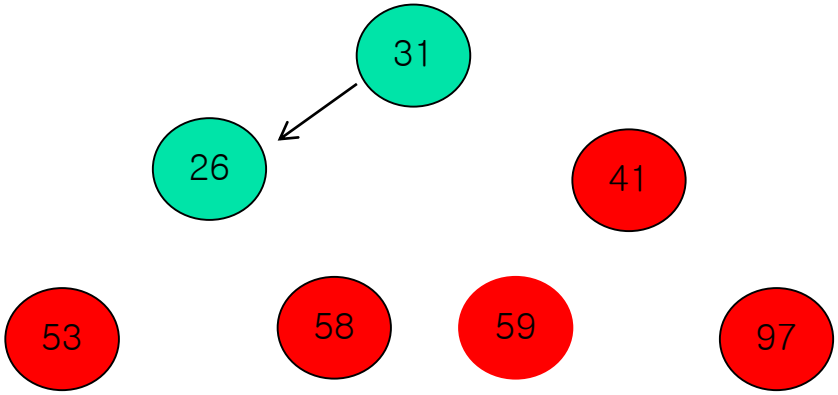
array		53	41	31	26	58	59	97				
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of Heap Sort



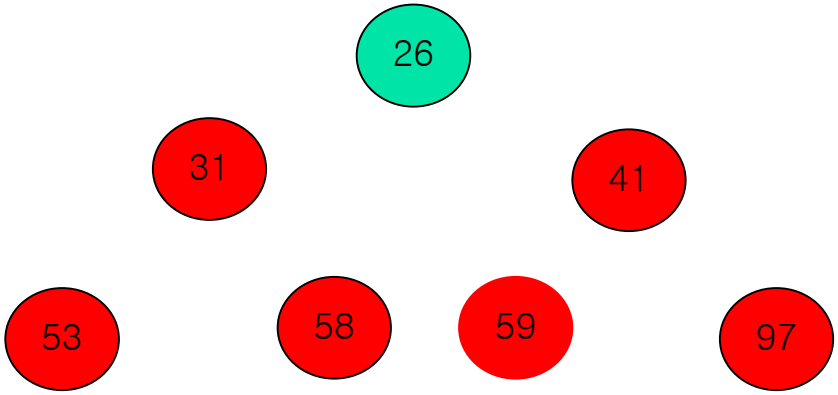
array		**41**	**26**	**31**	**53**	**58**	**59**	**97**				
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Operation of Heap Sort

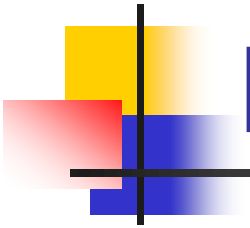


array		**31**	**26**	**41**	**53**	**58**	**59**	**97**				
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Operation of Heap Sort



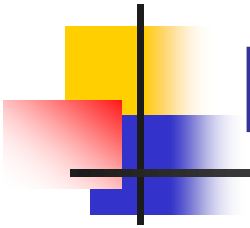
array		**26**	**31**	**41**	**53**	**58**	**59**	**97**				
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**



Heapsort(A)

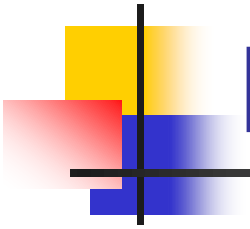
HAEPSORT(A)

1. BUILD-MAX-HEAP(A)
2. **for** i = A.length **downto** 2
3. exchange A[1] with A[i]
4. A.heap-size = A.heap-size-1
5. MAX-HEAPIFY(A,1)



Heapsort

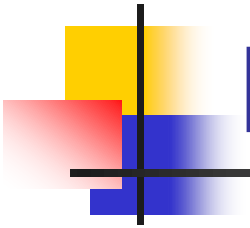
- It takes O(n lg n) time:
  - Calls to BUILD-MAX-HEAP takes O(n) time.
  - Each of the n-1 calls to MAX-HEAPIFY takes O(lg n) time.



Priority Queues

- HEAP-MAXIMUM(A)

1. Return A[1]

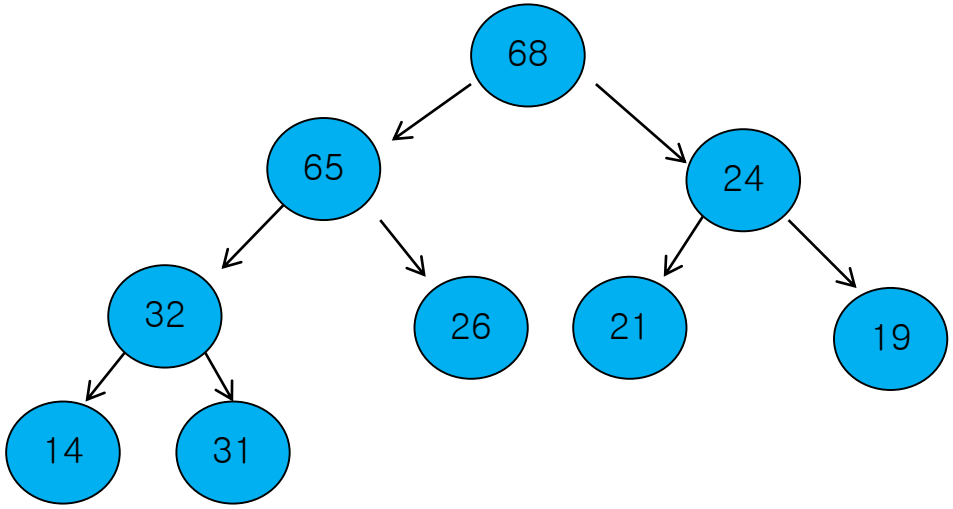


Priority Queues

- HEAP-EXTRACT-MAX(A)

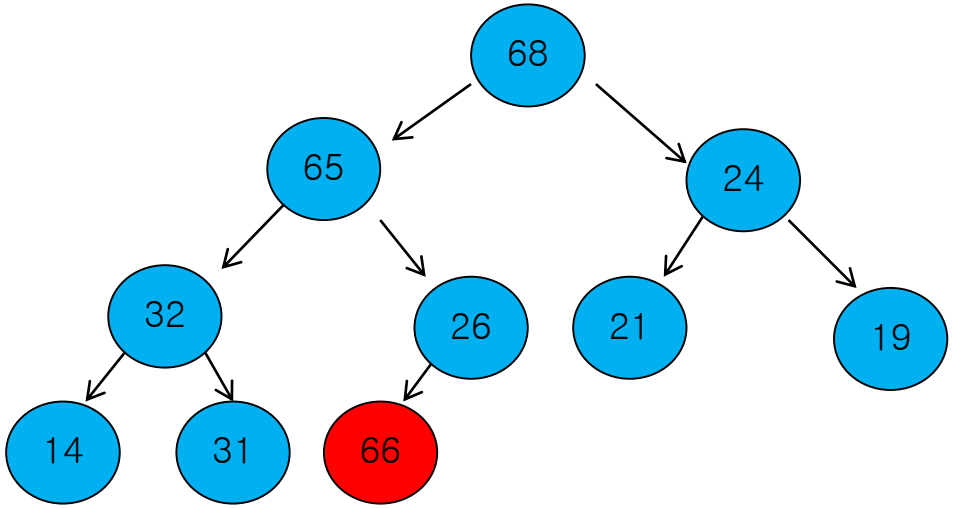
```
1.  if A.heap-size < 1
2.      error "heap underflow"
3.  max = A[1]
4.  A[1] = A[A.heap-size]
5.  A.heap-size = A.heap-size – 1
6.  MAX-HEAPIFY(A, 1)
7.  return max
```

INSERT



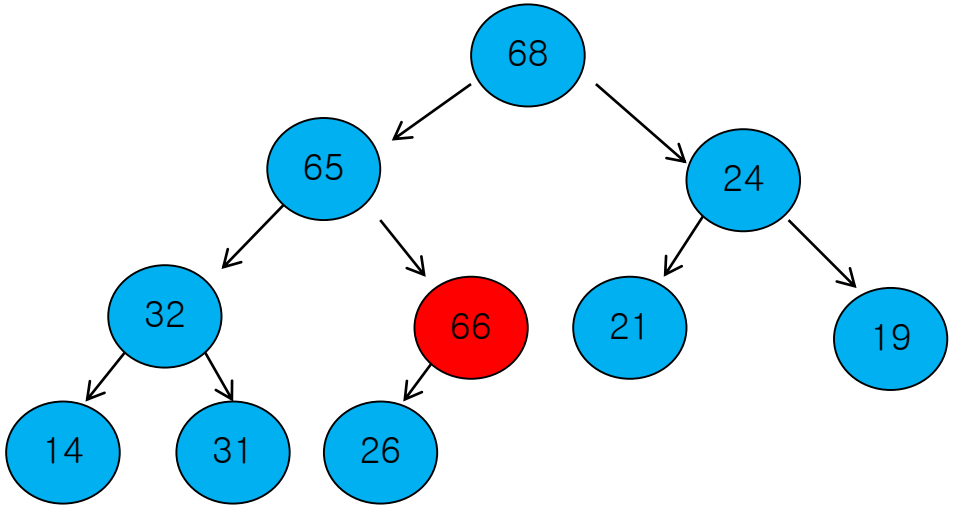
array		68	65	24	32	26	21	19	14	31		
index	0	1	2	3	4	5	6	7	8	9	10	11

INSERT



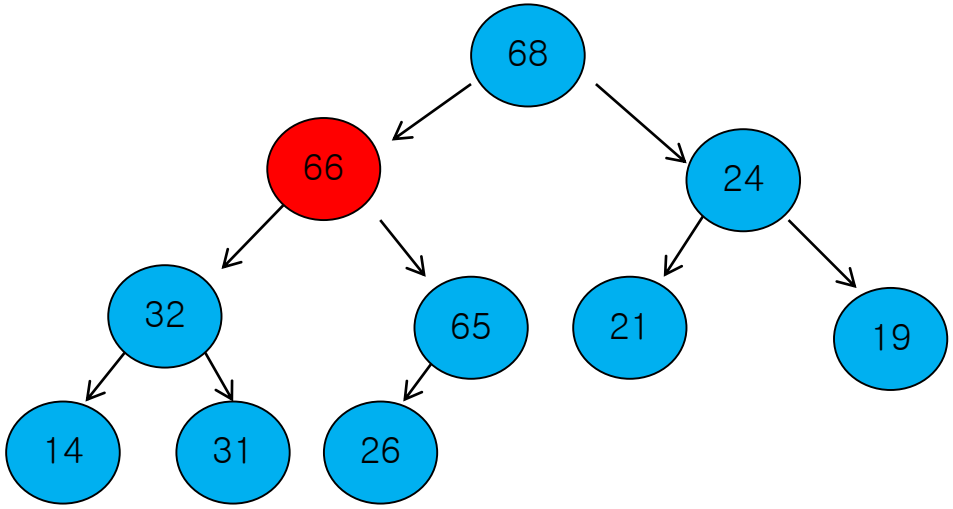
array		**68**	**65**	**24**	**32**	**26**	**21**	**19**	**14**	**31**	**66**		
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

INSERT

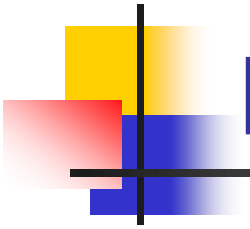


array		**68**	**65**	**24**	**32**	**26**	**21**	**19**	**14**	**31**	**66**		
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

INSERT

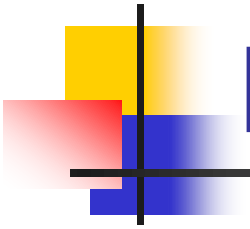


array		**68**	**66**	**24**	**32**	**26**	**21**	**19**	**14**	**31**	**65**		
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**



Heap: Priority Queues

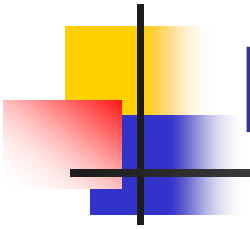
- Max(min) priority queue
  - An element with highest(lowest) priority is deleted
  - An element with arbitrary priority can be inserted
  - Frequently implemented using max(min) heap



Priority Queues

- Abstract class in C++

```
template <class T>
class MaxPQ {
public:
    virtual ~MaxPQ(){}
                // virtual destructor
    virtual bool IsEmpty() const = 0;
                // return true if the priority queue is empty
    virtual const T& Top() const = 0;
                // return reference to max element
    virtual void Push(const T&) = 0;
                // add an element to the priority queue
    virtual void Pop() = 0;
                // delete element with max priority
 };
```



Definition of a Heap

- Max(min) tree
  - A tree in which the key value in each node is no smaller (larger) than the key values in its children (if any)
  - The key in the root is the largest (smallest)
- Max(min) heap
  - A complete binary tree that is also a max(min) tree

Definition of a Heap

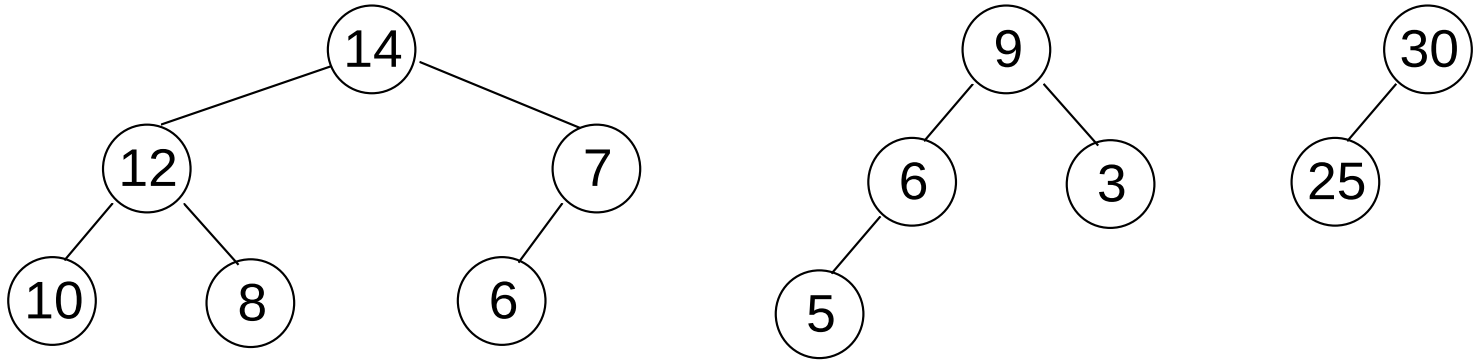


Figure 5.24 : Max heaps

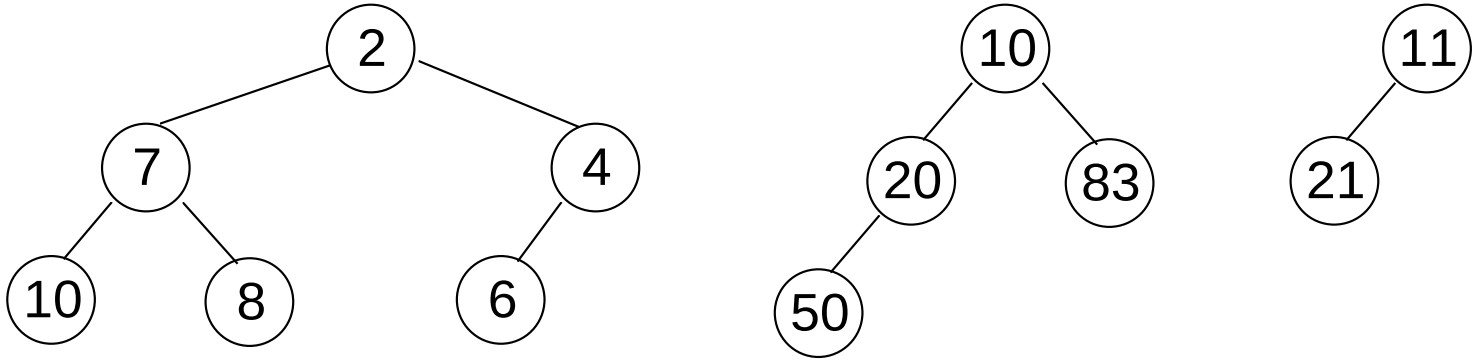
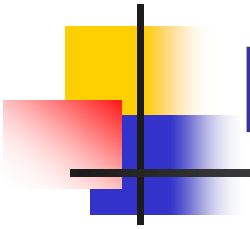


Figure 5.25 : Min heaps



Definition of a Heap

- Basic operations of a max heap
  - Creation of an empty heap
  - Insertion of a new element into the heap
  - Deletion of the largest element from the heap
- Private data members of class MaxHeap

```
private:
    T *heap;          // element array
    int heapSize;     // number of elements in heap
    int capacity;     // size of the array heap
```

Definition of a Heap

```
template <class T>
MaxHeap<T>::MaxHeap(int theCapacity = 10)
{
        if (theCapacity < 1) throw “Capacity must be >= 1”;
        capacity = theCapacity;
        heapSize = 0;
        heap = new T[capacity+1]; // heap[0] is not used
}
```

Program 5.15 : Max heap constructor

Insertion into Max Heap

- Examples

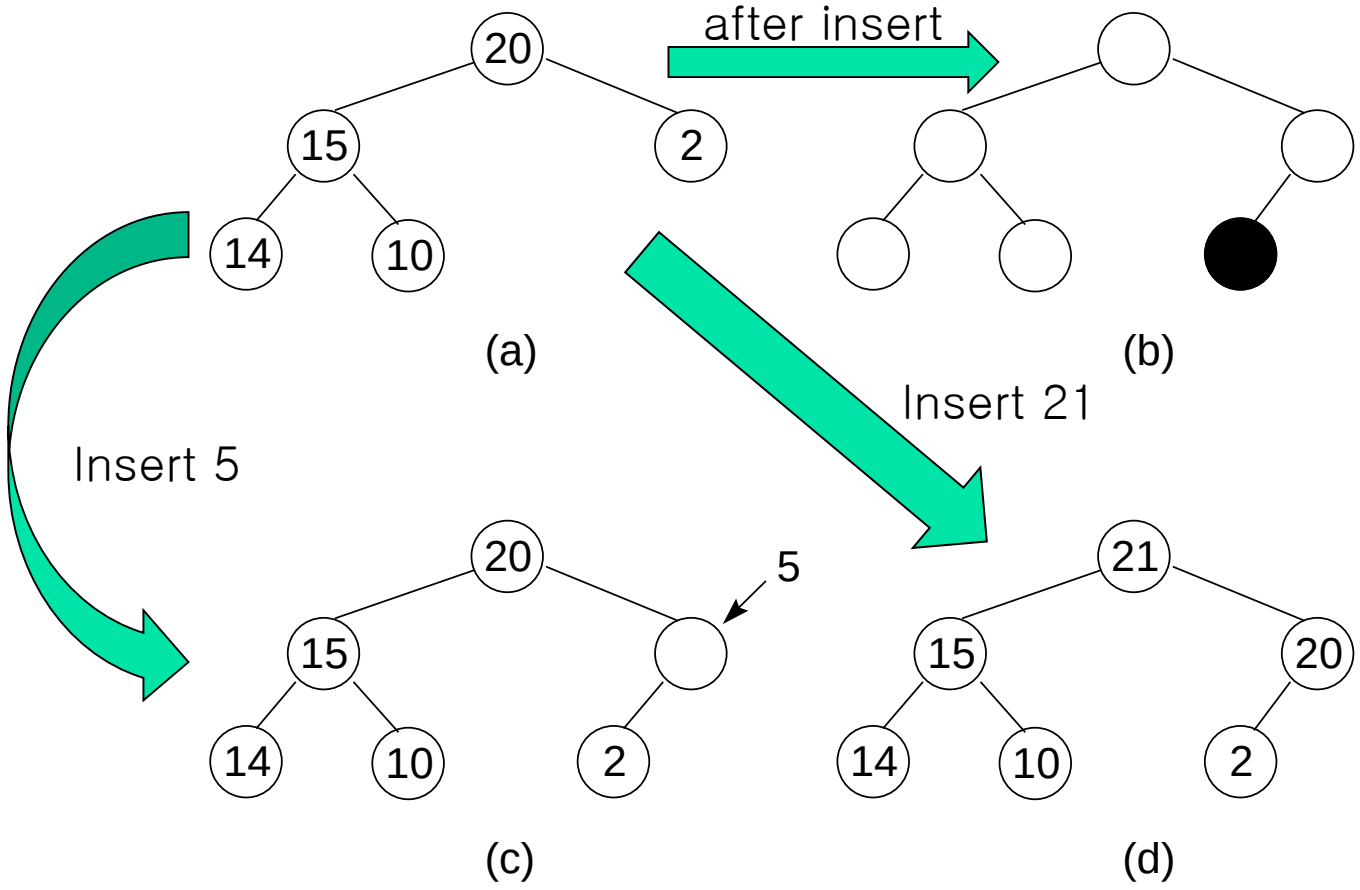
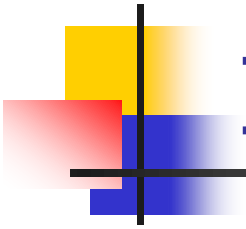


Figure 5.26 : Insertion into a max heap



Insertion into Max Heap

- Implementation
  - Need to move from child to parent
  - Heap is complete binary tree
    - Use formula-based representation
    - Lemma 5.4 : parent(i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$
  - Complexity is O(log n)

Insertion into Max Heap

```
template <class T>
void MaxHeap<T>::Push(const T& e)
{ // Insert e into the max heap
        if (heapSize == capacity) { // double the capacity
                ChangeSize1D(heap, capacity, 2*capacity);
                capacity *= 2;
        }
        int currentNode = ++heapSize;
        while (currentNode != 1 && heap[currentNode / 2] < e)
        { // bubble up
                heap[currentNode] = heap[currentNode / 2];
                                        // move parent down
                currentNode /= 2;
        }
        heap[currentNode] = e;
}
```

Program 5.16 : Insertion into a max heap

Deletion from Max Heap

- Example

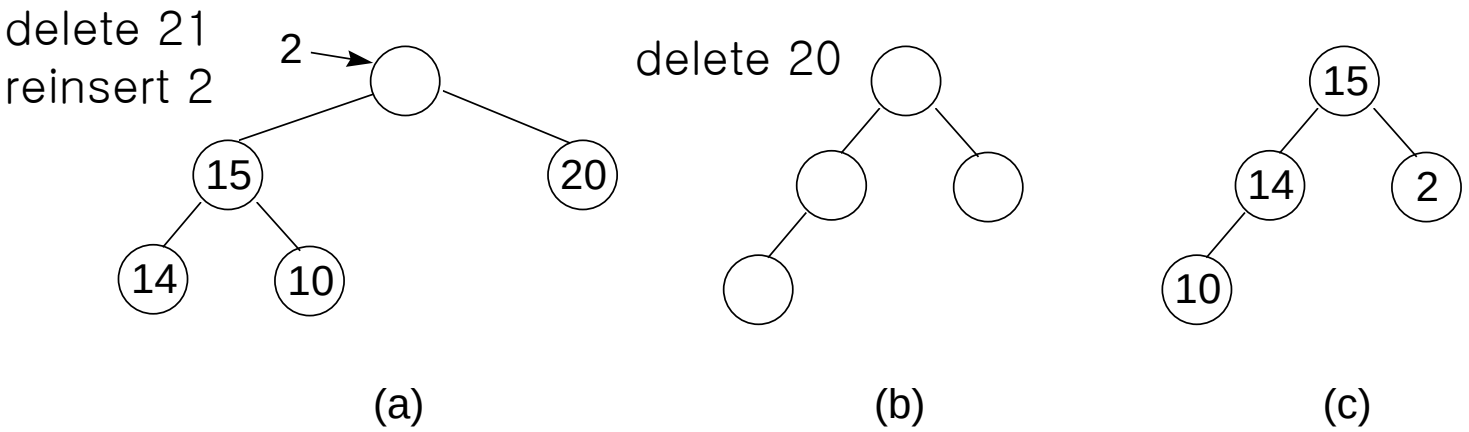
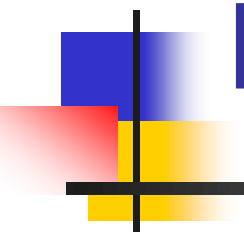


Figure 5.27 : Deletion from a heap

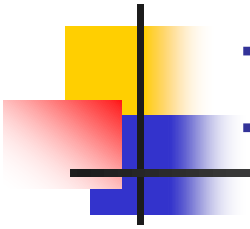
Deletion from Max Heap

```
template <class T>
void MaxHeap<T>::Pop()
{ // Delete max element
        if (IsEmpty()) throw "Heap is empty. Cannot delete.";
        heap[1].~T(); // delete max element

        // remove last element from heap
        T lastE = heap[heapSize--];

        // trickle down
        int currentNode = 1;        // root
        int child = 2;              // a child of currentNode
        while (child <= heapSize)
        {
                // set child to larger child of currentNode
                if (child<heapSize && heap[child]<heap[child+1]) child++;
                // can we put lastE in current Node?
                if (lastE>=heap[child]) break; // yes

                // no
                heap[currentNode] = heap[child];
                currentNode = child; child *= 2;
        }
        heap[currentNode] = lastE;
}
```

Program 5.17 : Deletion from a max heap

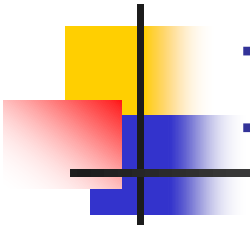


Representing Disjoint Sets



Introduction

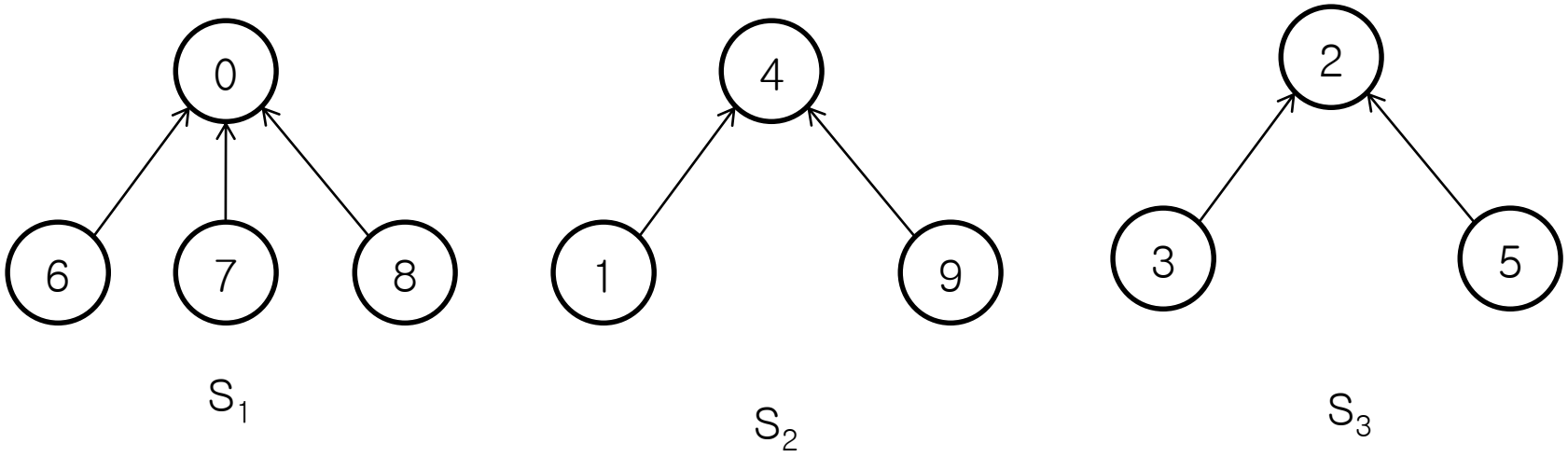
- Use of trees in the representation of sets.
- Assume
  - Elements of the sets are the numbers 0,1,2,3,…,n-1
  - Pairwise disjoint (S_i and S_j, $i \neq j$, there is no element that is in both S_i and S_j)



Introduction (cont.)

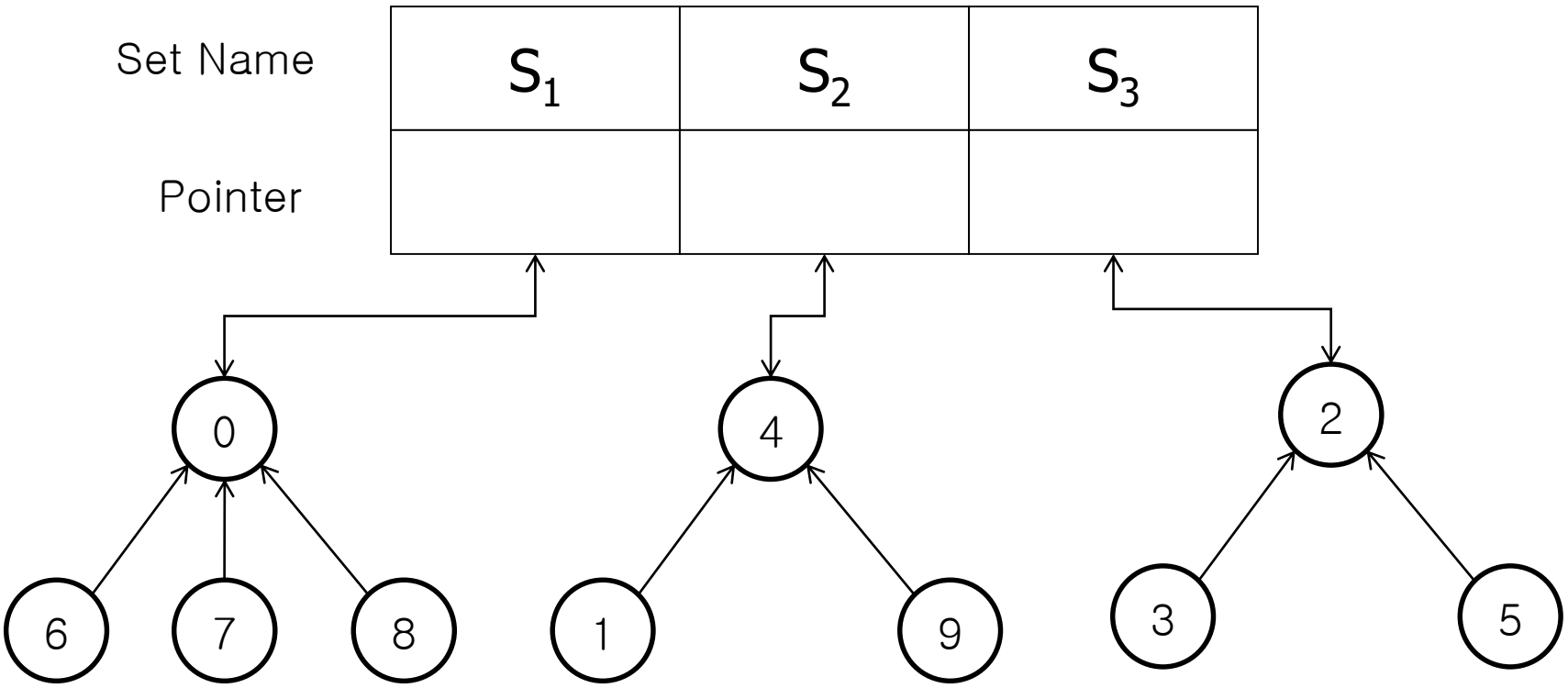
- Operation
  1) Disjoint set union. If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ = { all elements x such that x in in S_i or S_j }
  2) Find(i). Find the set containing element i.

Introduction (cont.)



5.36 : Possible tree representation of sets

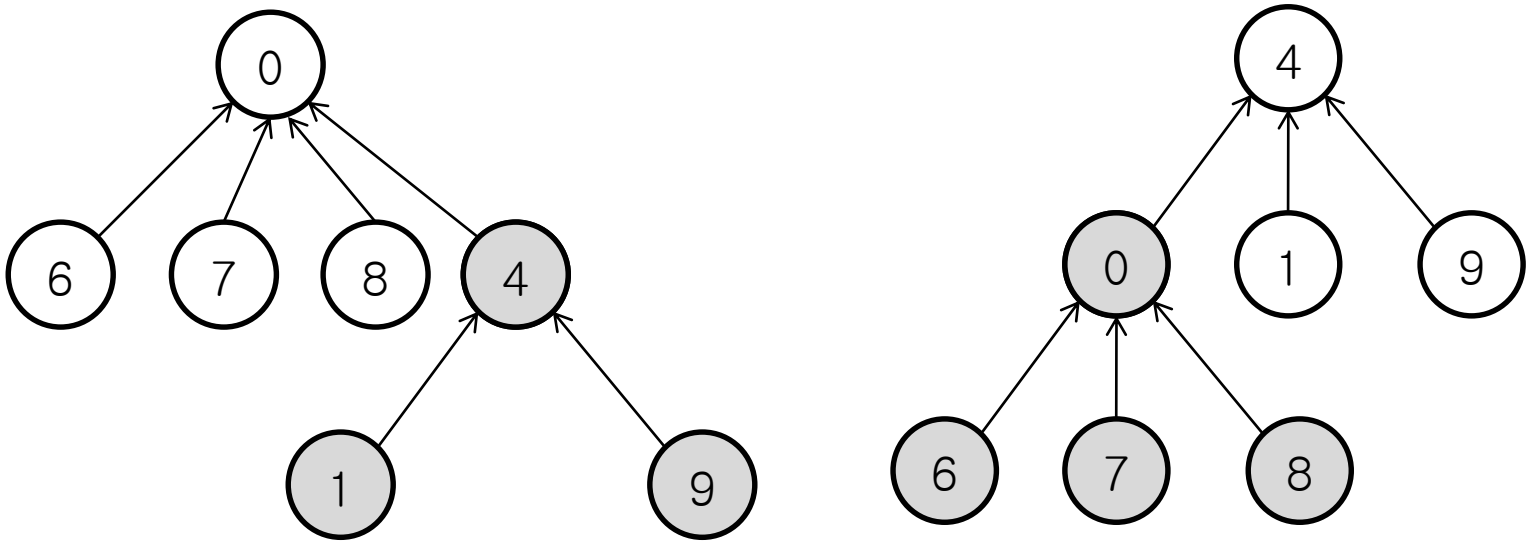
Introduction (cont.)



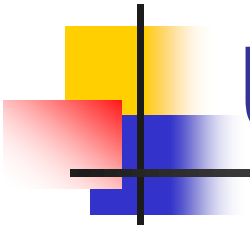
Data representation for S_1, S_2, and S_3

Union and Find Operations

- Union
  - Union of S_1 and S_2



Possible representations of $S_1 \cup S_2$

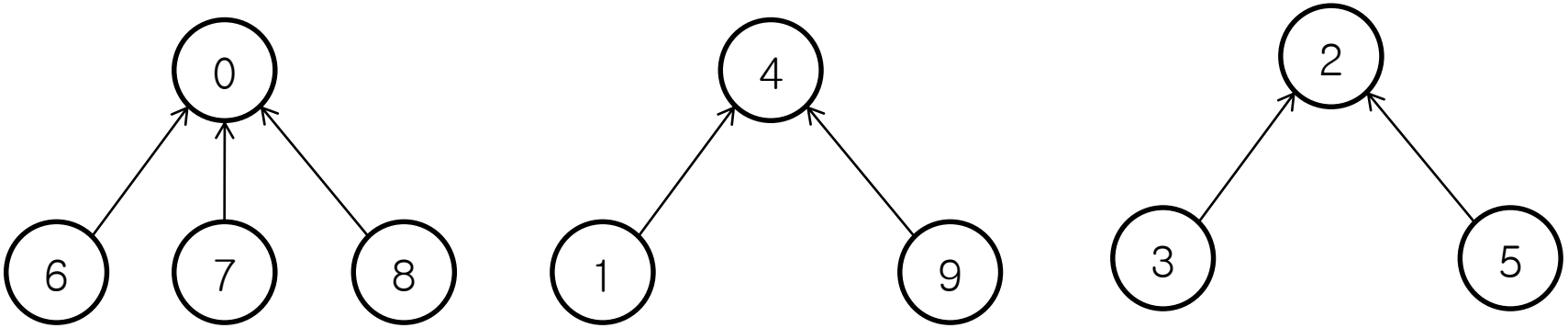


Union and Find Operations

- Union
  - Set parent field of one of the roots to the other root.

Union and Find Operations

- Since set elements are numbered 0 through n-1, we represent the tree nodes using an array parent[n].
- This array element gives the parent pointer of the corresponding tree node.



i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

Array Representation of S_1, S_2, and S_3 of 5.36

Union and Find Operations

- Find(i)
  - Ex. Find(5)
  - Start at 5 -> moves to 5's parent, 2 -> parent[2]=-1, we have reached root.
- Union(i,j)
  - We pass in two trees with roots i and j, i≠j -> parent[i]=j

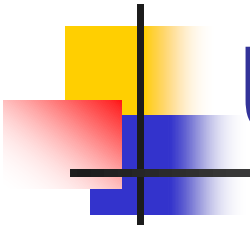
i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

Class Definition and Constructor for Sets

```
class Sets{
pulib:
        //set operations follow
        .
        .
private:
        int *parent;
        int n; //number of set elements
};

Sets::Sets(int numberOfElements)
{
        if (numberOfElements<2) throw “Must have at least 2 elements”;
        n=numberOfElements;
        parent=new int[n];
        fill(parent,parent+n,-1);
}
```

Simple Functions for Union and Find

```
void Sets::SimpleUnion(int i, int j)
{//Replace the disjoint sets with roots i and j, i!=j with their union.
        parent[i]=j;
}

int Sets::SimpleFind(int i)
{//Find the root of the tree containing element i.
        while (parent[i]>=0) i=parent[i];
        return i;
}
```



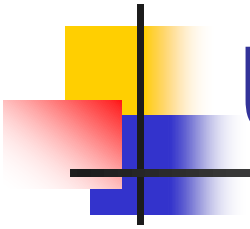
Union and Find operations

- Analysis of SimpleUnion and SimpleFind
  - Start off with n elements each in a set of its own(i.e., $S_i=\{i\}$, $0 \le i < n$) -> Initial configuration consists of a forest with n nodes, and parent[i]=-1, $0 \le i < n$

Union and Find Operations

- Process the following sequence of operations:
- Union(0,1), Union(1,2),…,Union(n-2,n-1)
- Find(0),Find(1),…,Find(n-1)
- Time Taken for a union is constant : n-1 unions in time O(n).
- Each find operation requires following a sequence of parent pointers from the element to be found to the root.

$$O\left(\sum_{i=1}^{n} i\right) = O(n^2)$$

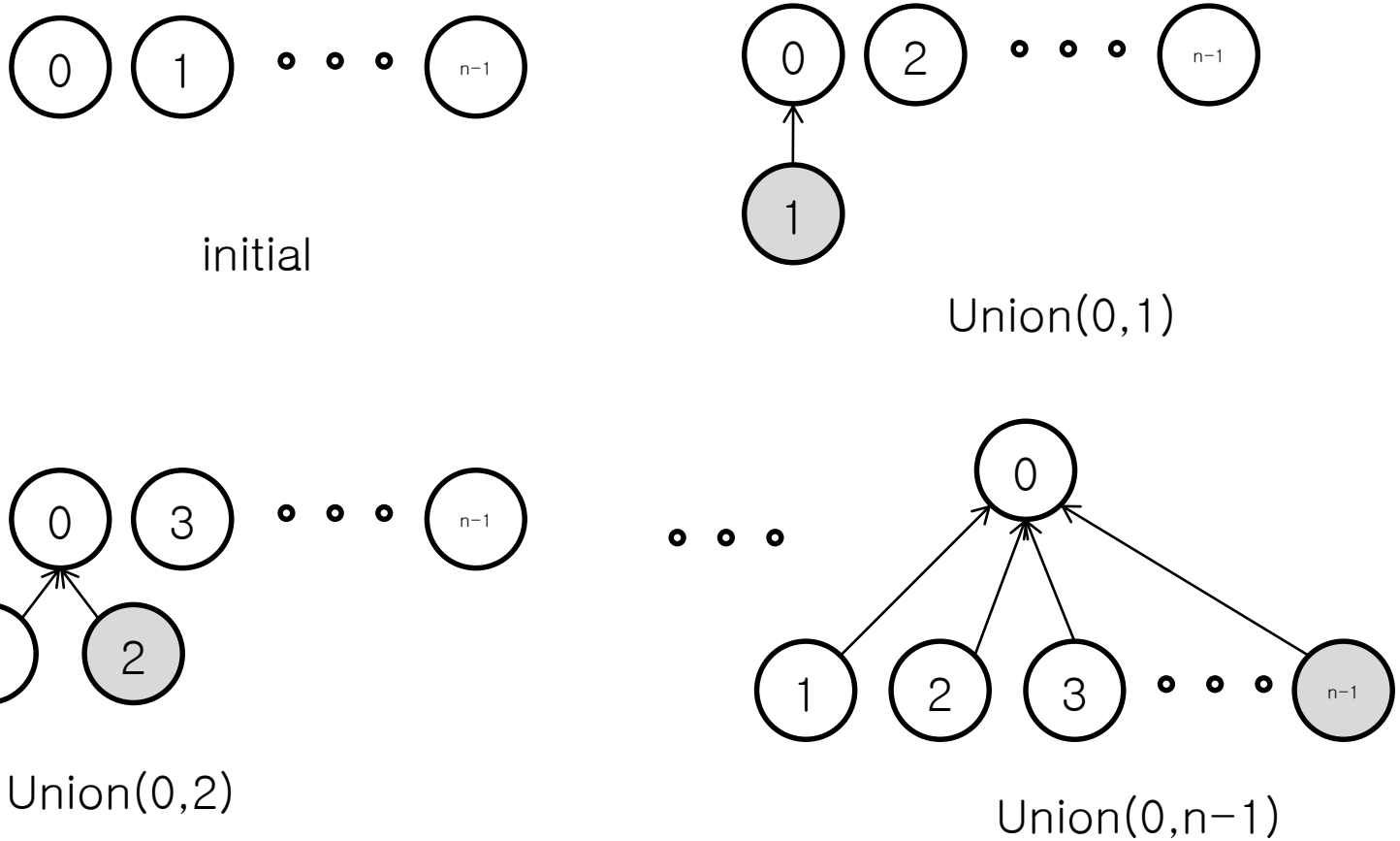


Union and Find Operations

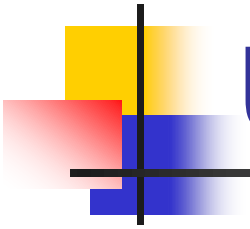
- Avoiding the creation of degenerate trees.
- Definition [Weighting rule for Union(i,j)]:

  If the number of nodes in the tree with root i is less than the number in the tree with root j, then make j the parent of i; otherwise make i the parent of j

Union and Find operations



Trees obtained using the weighting rule



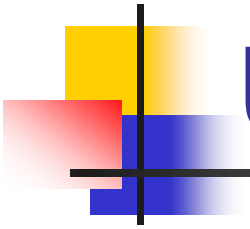
Union and Find Operations

- Unions have been modified so that the input parameter values correspond to the roots of the trees to be combined.

Union Function with Weighting rule

```
void Sets::WeightedUnion(int i, int j)
//Union sets with roots i and j, i≠j using the weighting rule.
//parent[i]=-count[i] and parent[j]= -count[j]
{
        int temp=parent[i]+parent[j];
        if (parent[i]>parent[j]){//i has fewer nodes
                parent[i]=j;
                parent[j]=temp;
        }
        else {//j has fewer nodes(or i and j have the same
              //                        number of nodes)
                parent[j]=i;
                parent[i]=temp;
        }
}
```

Union and Find Operations



- Analysis of WeightedUnion and Simple Find
  - The time required to perform a union has increased somewhat but is still bounded by a constant.
  - The maximum time to perform a find is determined by Lemma 5.5
- Lemma 5.5: Assume that we start with a forest of trees, each having one node. Let T be a tree with m nodes created as a result of a sequence of unions each performed using function WeightedUnion. The Height of T is no greater than

$$\lfloor \log_2 m + 1 \rfloor$$

Union and Find Operations (cont.)

- Proof :
  - True for m=1
  - Assume true for all trees with i nodes, i≤m-1
    -> show that also true for i=m
  - Let T be a tree with m nodes created by function WeightedUnion.
  - Consider the last union operation performed, Union(k,j).
  - Let a be the number of nodes in tree j and m-a the number in k
  - Without loss of generality we may assume 1≤a≤m/2
  - Height of T is either the same as that of k or is one more than that of j
  - Former : $T \le \text{floor}(\log_2(m-a))+1 \le \text{floor}(\log_2 m)+1$
  - Latter : $T \le \text{floor}(\log_2 a)+2 \le \text{floor}(\log_2 m/2)+2 \le \text{floor}(\log_2 m)+1$

Union and Find Operations

- Definition [Collapsing rule] :

  If j is a node on the path from i to its root and parent[i]≠root(i), then set parent[j] to root(i).

```
int Sets::CollasingFind(int i)
{//Find the root of the tree containing element i.
//Use of collapsing rule to collapse all nodes from i to the root.
        for (int r=i; parent[r]>=0; r=parent[r]);//find foot
        while (i!=r){//collapse
                int s=parent[i];
                parent[i]=r;
                i=s;
        }
        return r;
}
```

Union and Find Operations

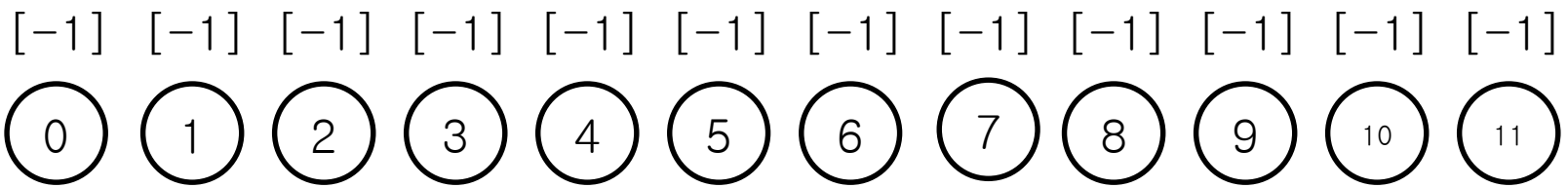
- Analysis of WeightedUnion and CollapsingFind
  - Use of the collapsing rule roughly doubles the time for an individual find.
  - It reduces the worst case time over a sequence of finds.
  - The worst-case complexity of processing a sequence of unions and finds is stated in Lemma 5.6.

- Lemma 5.6[Tarjan and Van Leeuwed]: Assume that we start with a forest of trees, each having one node. Let T(f,u) be the maximum time required to process any intermixed sequence of f finds and u unions. Assum that $u \geq n/2$.

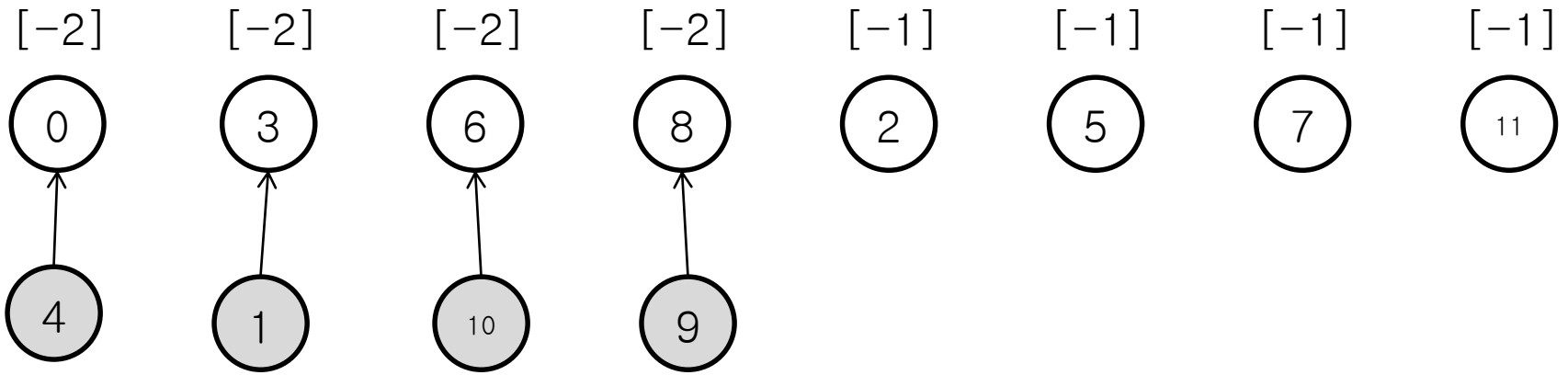
  Then $k_1(n+f*\alpha(f+n,n)) \leq T(f,u) \leq k_2(n+f*\alpha(f+n,n))$ for some positive constants k_1 and k_2.

- Note that $\alpha(p,q) \leq 4$ for all practical cases.

Application to Equivalence Class

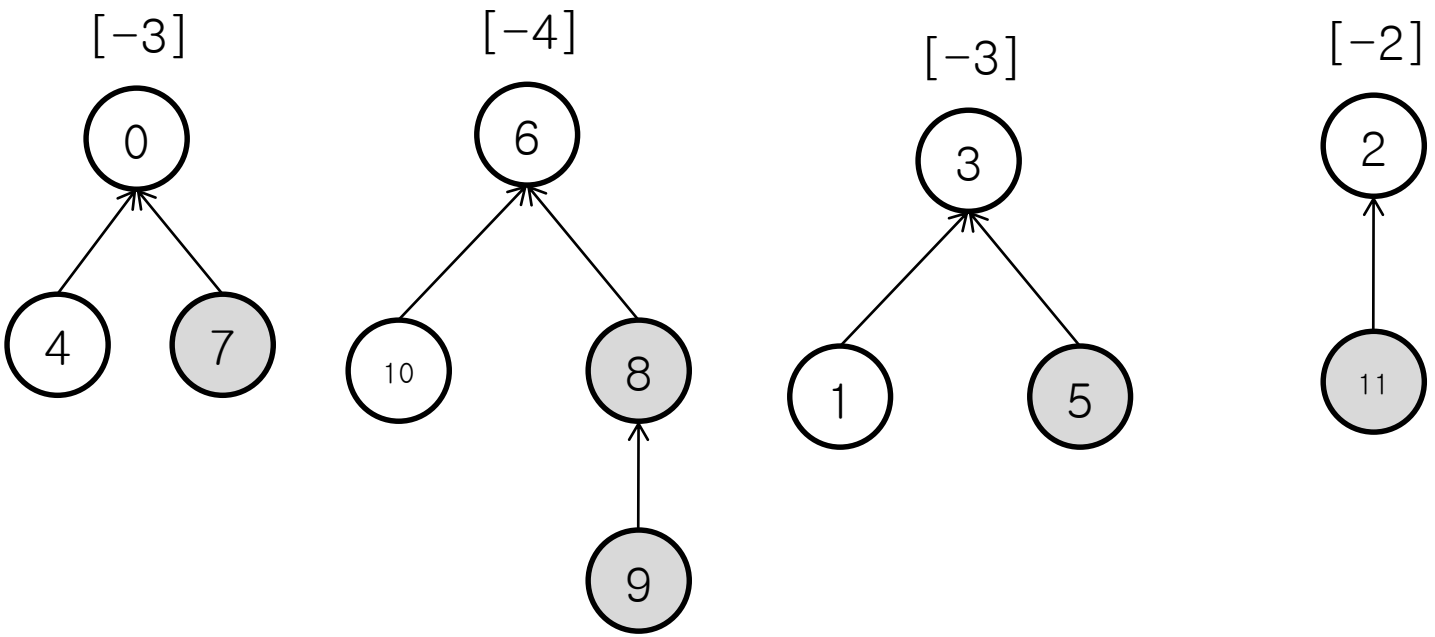


Initial Trees



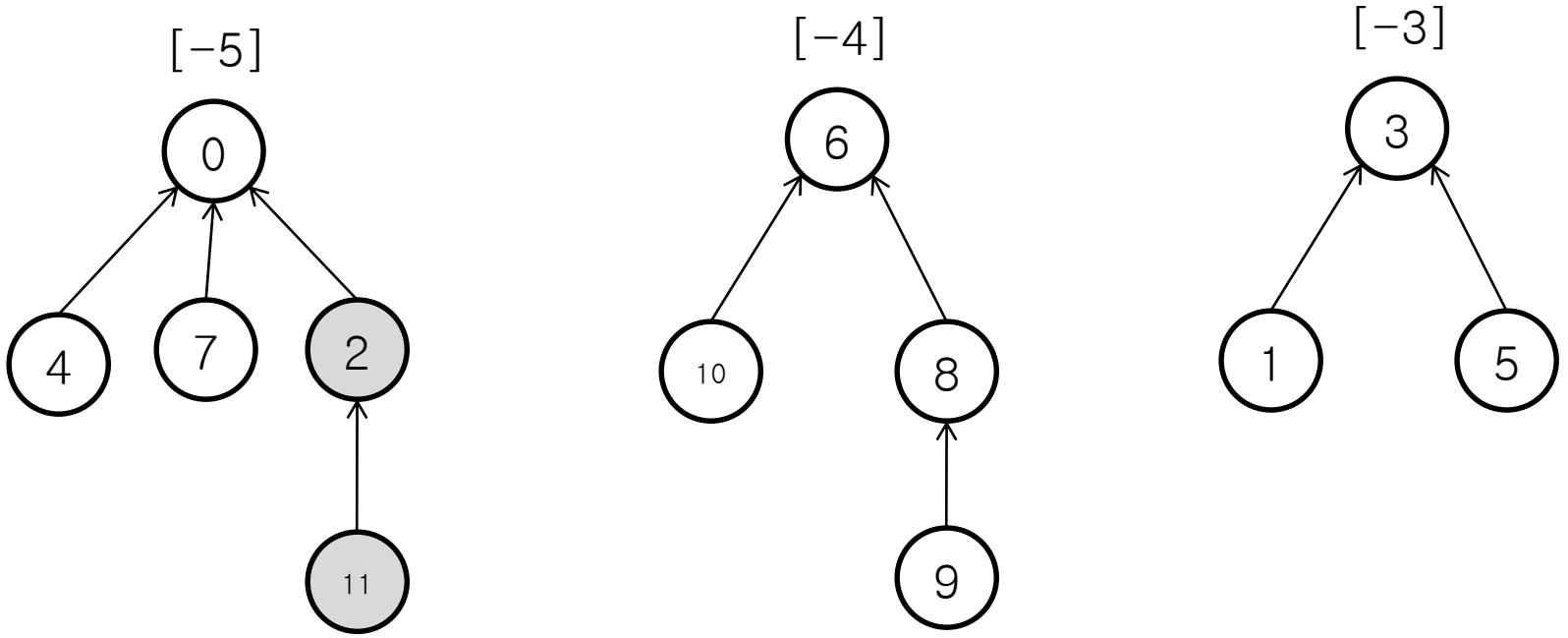
Height-2 trees following 0≡4, 3≡1, 6≡10, 8≡9

Application to Equivalence Class



Trees following 7≡4, 3≡5, 6≡8, 2≡11

Application to Equivalence Class



Trees following 11≡0