Lecture 5

Heap, BST

吴蔚琪 2022.11.11

991《数据结构与算法》考纲

5、树

⑴树的概念和性质。

⑵二叉树的概念、性质和实现。

⑶二叉树的顺序存储结构和链式存储结构。

⑷遍历二叉树。

⑸树和森林的存储结构、遍历。

⑹堆与优先队列。

⑺二叉排序树。

⑻平衡二叉树。

⑼哈夫曼(Huffman)树和哈夫曼编码。

6. 并查集

⑴并查集的概念与实现。

Binary Heap

Priority Queue

Background & Review

We have discussed Abstract Lists with explicit linear orders

- Arrays, linked lists, strings

We saw three cases which restricted the operations:

- Stacks, queues, deques

Following this, we looked at search trees for storing implicit linear orders: Abstract Sorted Lists

- Run times were generally $\Theta(\ln(n))$

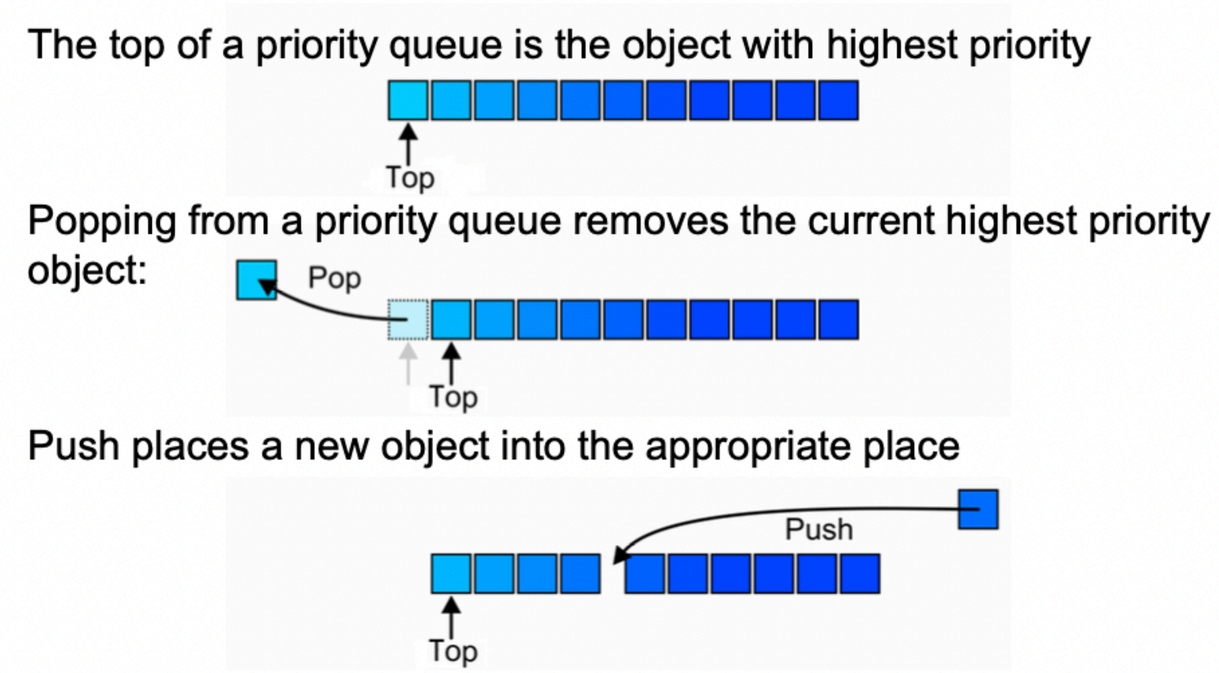
We will now look at a restriction on an implicit linear ordering:

- Priority queues

Priority Queue

Background & Review

- Queue: FIFO
- Priority queues
  - Each object is associated with a priority
    - The value o has the highest priority.
    - The higher the number, the lower the priority
  - We pop the object which has the highest priority
- How to implement a priority queue?



Multiple Queue

One way for Implementation

- Goal: make the run time of each operation as close to O(1) as possible
- Multiple Queues: one for each priority

Assume there is a fixed number of priorities, say M

- Create an array of M queues
- Push a new object onto the queue corresponding to the priority
- Top and pop find the first non-empty queue with highest priority

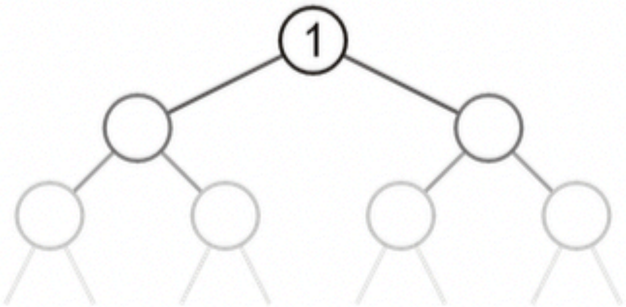
- Push is O(1), Top and Pop are both O(M)
- Cons: restricts the range of priorities, the memory requirement is O(M+N)

Heap

Can we do better?

We need a *heap*

- A tree with the top object at the root
- We will look at binary heaps
- Numerous other heaps exists:
  - d-ary heaps
  - Leftist heaps
  - Skew heaps
  - Binomial heaps
  - Fibonacci heaps
  - Bi-parental heaps



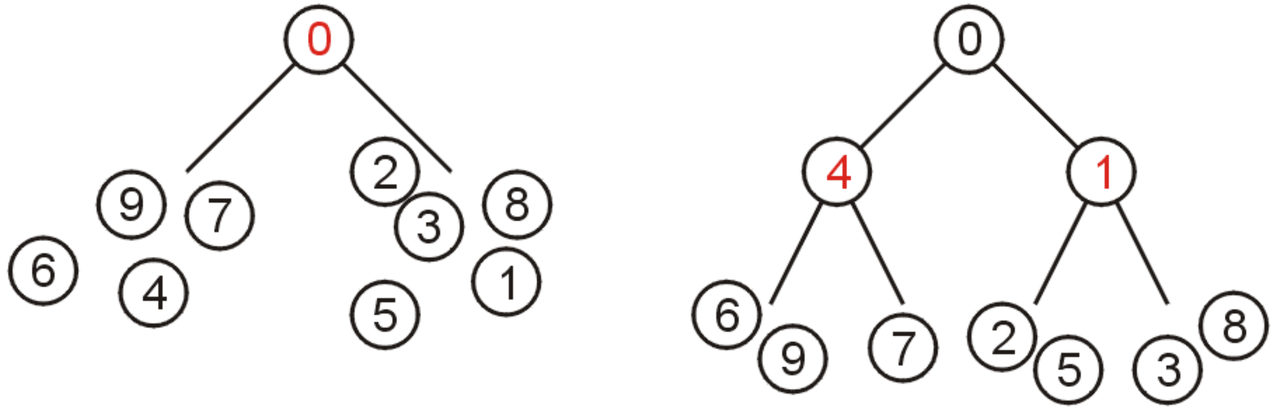
Contents for Heap

Definition, Operation and Application

- Heap Definition
- Heap Operation
  - Top: O(1)
  - Push()
  - Pop()
  - Build Heap
    - Simple method: repeatedly push()
    - Floyd's Method
- Heap Sort

2-1. Heap Definition

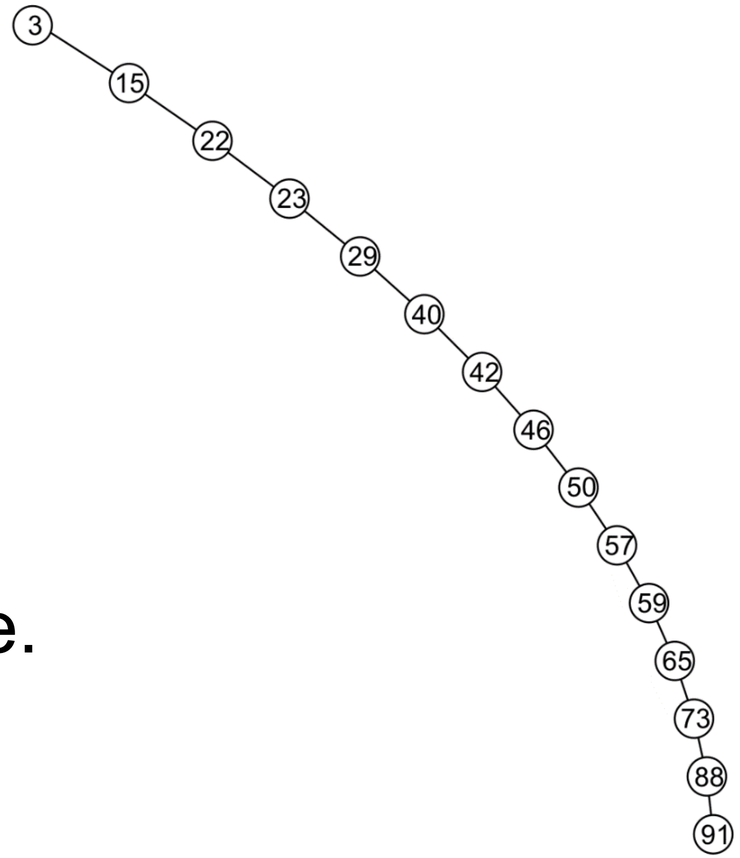
- A non-empty tree is a min-heap if
  - The key associated with the root is less than or equal to the keys associated with the sub-trees (if any).
  - The sub-trees (if any) are also min-heaps. (Recursive)



****There is no other relationship between the elements in the subtrees!*

2-1. Heap Storage

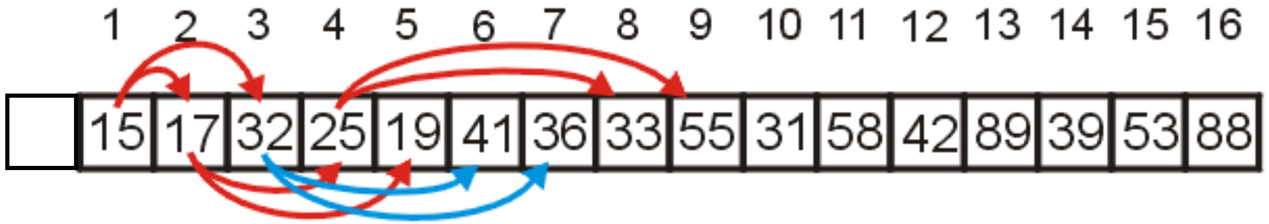
- If we use a simple tree structure, then the time complexity may be bad.
  - $O(n)$
  - Worst case: the binary tree is highly unbalanced



- Can we do better?
  - Keep balance – a complete tree structure.

2-1. Heap Storage - Array

- We start at index 1 when filling the array. (like complete tree)



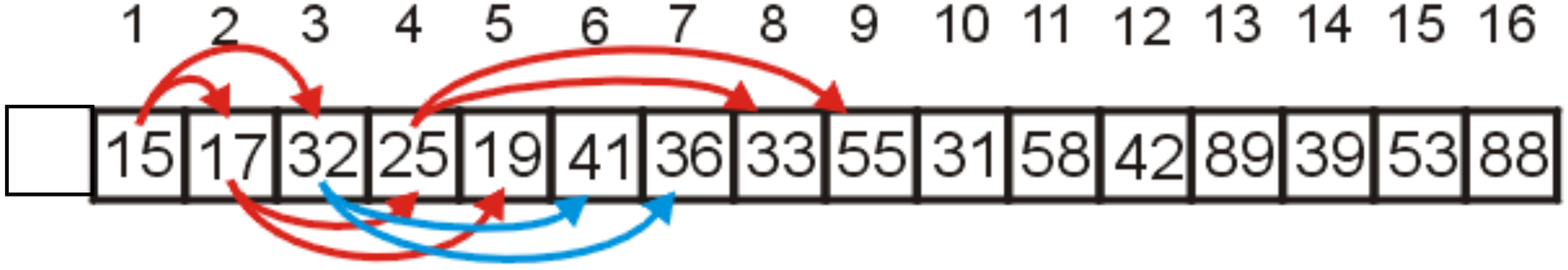
- Given the entry at index k, it follows that:
  - The parent of node is a $k/2$
  - The children are at $2k$ and $2k+1$

```
parent = k >> 1;

left_child = k << 1;
right_child = left_child + 1;
```

2-1. Heap Storage – Array Implementation

- If the heap-as-array has $\boldsymbol{count}$ entries, then the next empty node in the corresponding complete tree is at location $\boldsymbol{posn = count + 1}$
- We compare the item at location $\boldsymbol{posn}$ with the item at $\frac{\boldsymbol{posn}}{\boldsymbol{2}}$
- If they are out of order
  - Swap them
  - Set `posn /= 2` and repeat



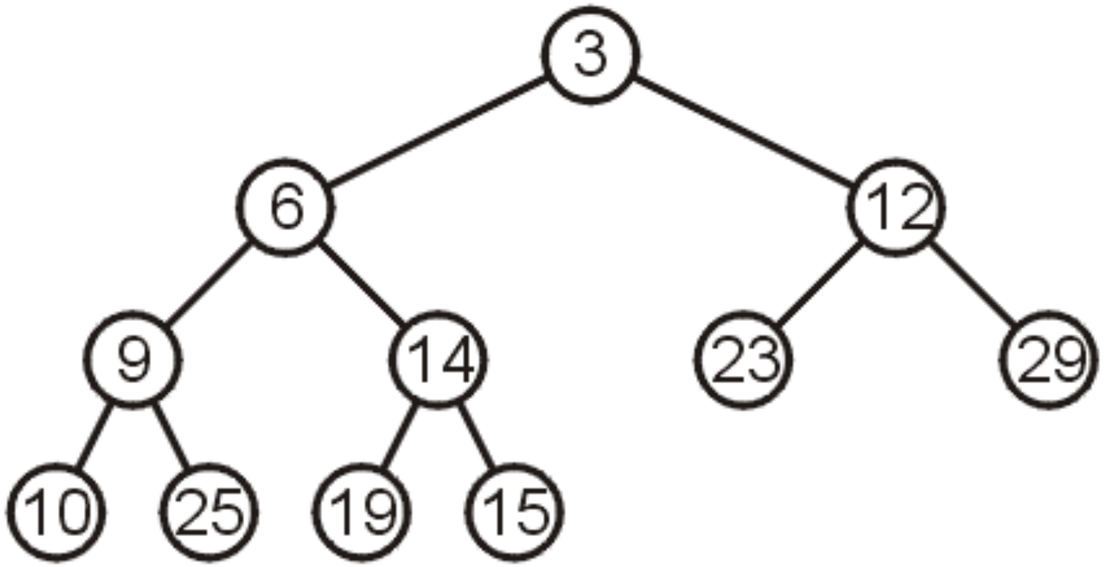
2-1. Heap Operations Guidelines

- We have such following operations on heap:
  - Push -> Based on Priority Queue
  - Pop -> Based on Priority Queue
  - Build Heap

- For each operation, the heap properties should always be satisfied.
- **`Heapify()`**

2-1. Heap Operations Example

- Consider the following heap, both as a tree and in its array representation.

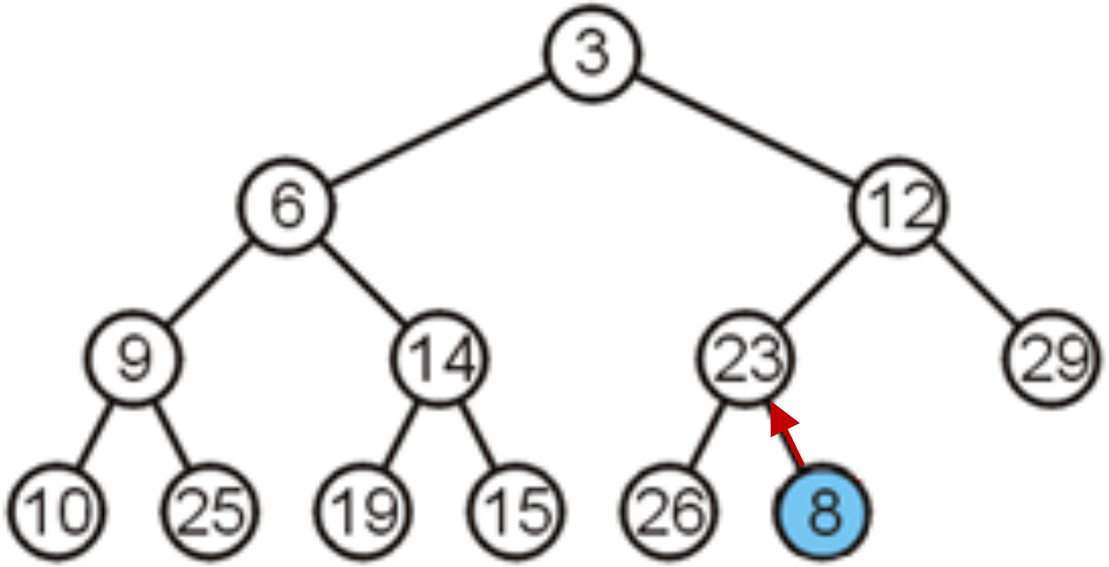


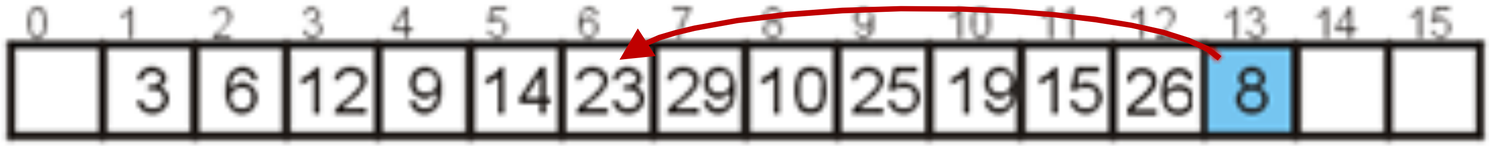
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	6	12	9	14	23	29	10	25	19	15				

push()

2-1. Heap Operations – Push()

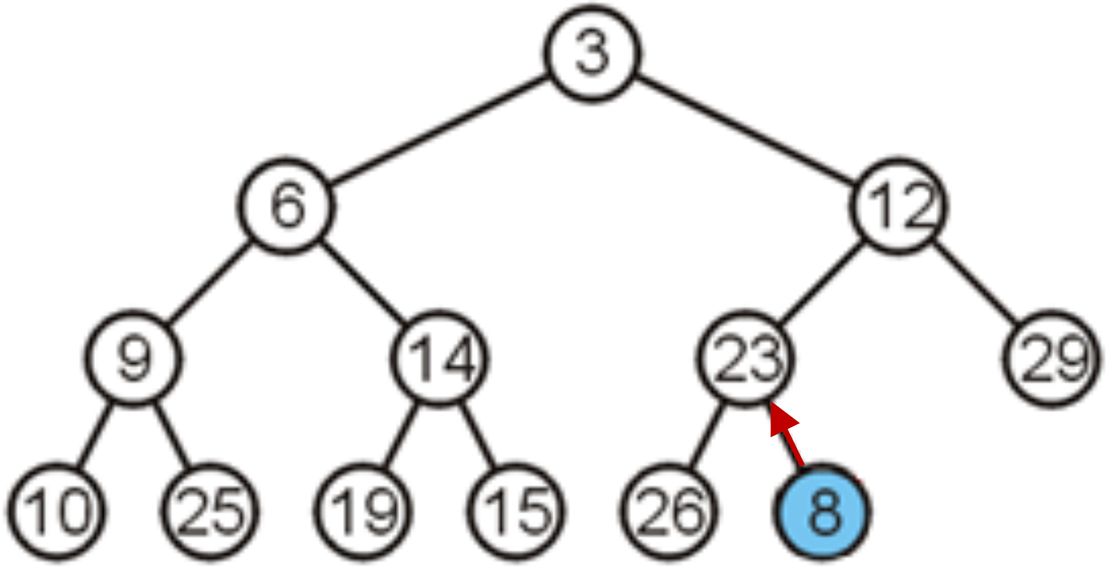
- Suppose we want a min-heap, and we want to insert 8.
  - First, insert it next to the last element in the array.

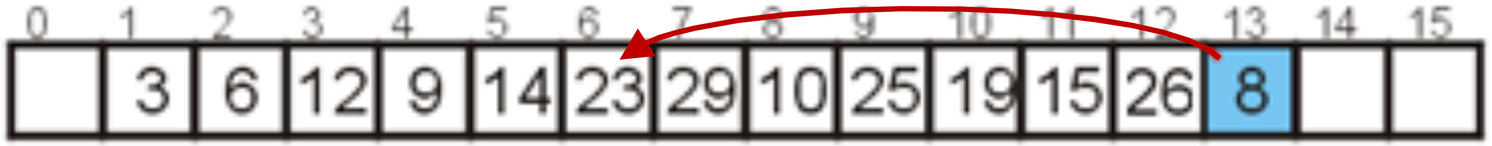




2-1. Heap Operations – Push()

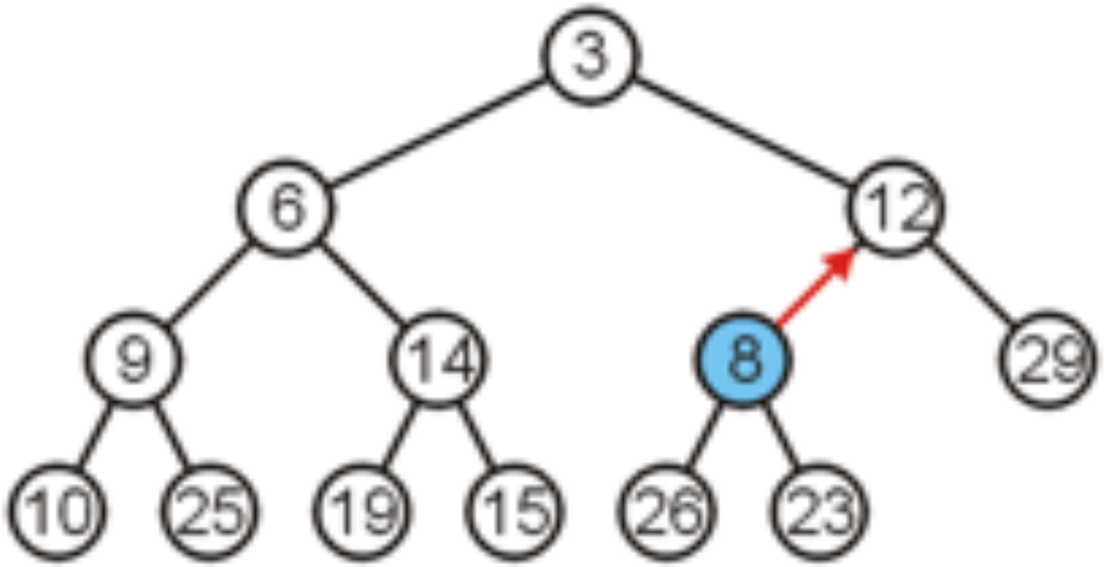
- Inserting 8 requires a few percolations:
  - (1) Swap 8 and 23.

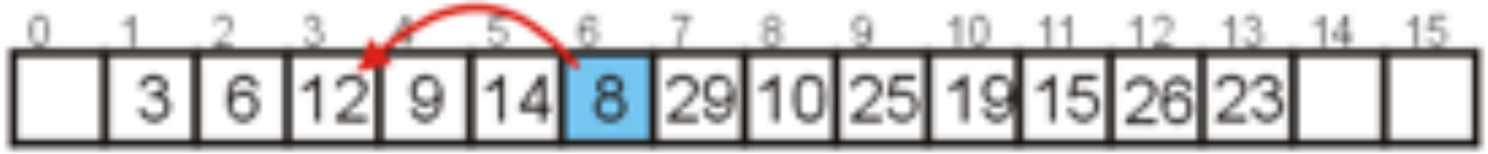




2-1. Heap Operations – Push()

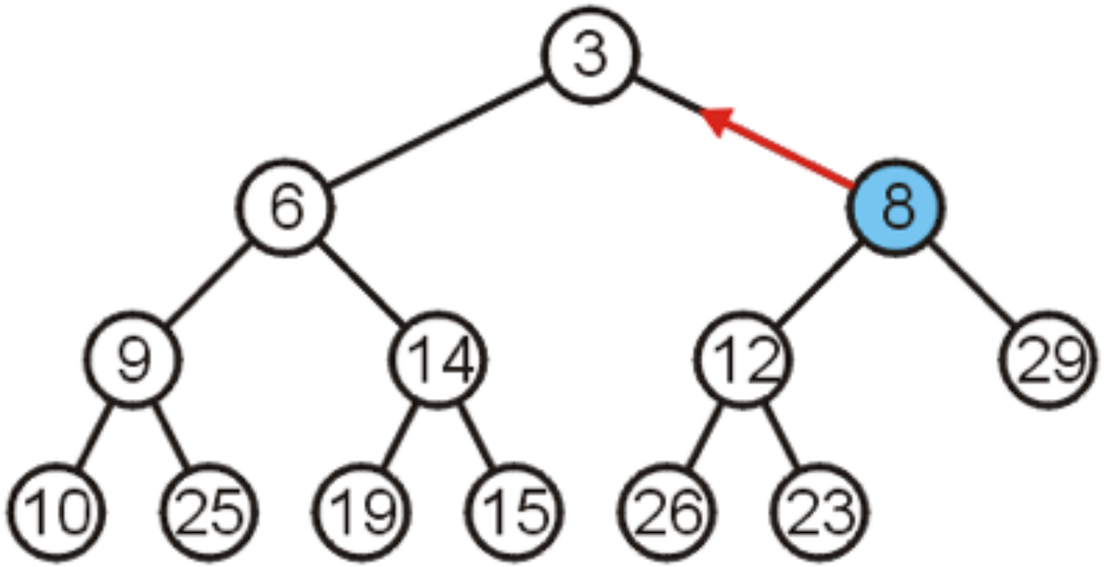
- Inserting 8 requires a few percolations:
  - (2) Swap 8 and 12.

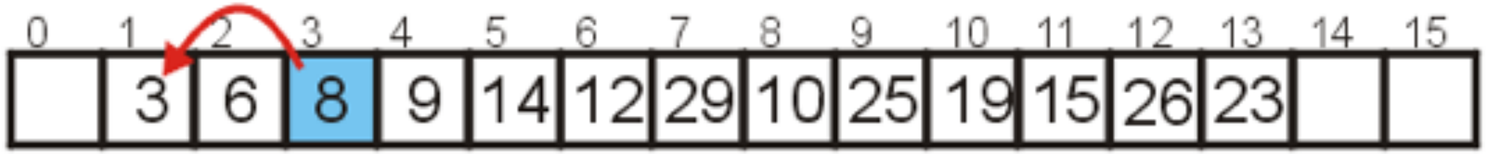




2-1. Heap Operations – Push()

- At this point, it is greater than its parent, so we are finished.





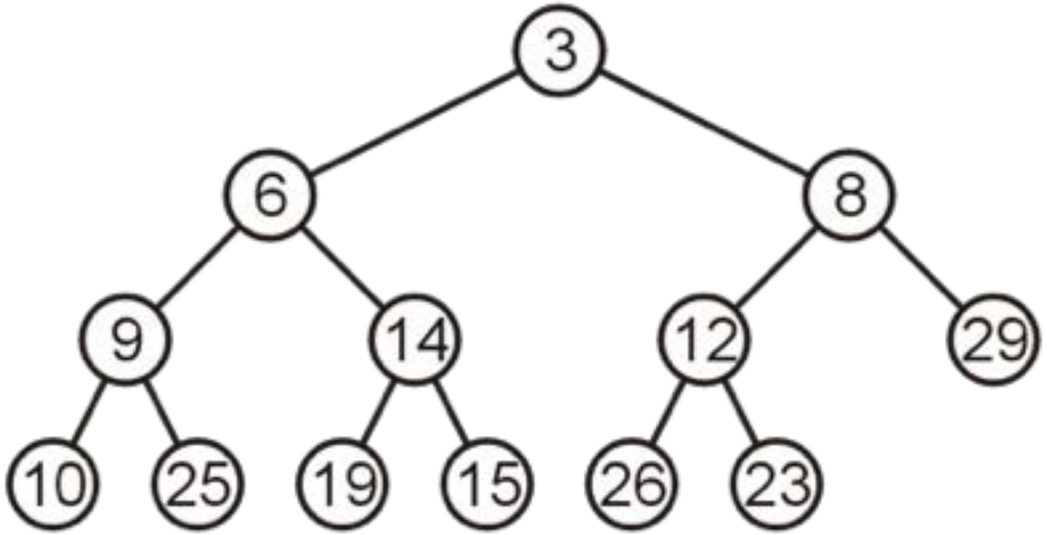
2-1. Heap Operations – Push() Algorithm

1. Insert it next to the last element in the array.
2. Swap with its parent if the current node is smaller than its parent repeatedly.
    1. If the current node is the root, then the algorithm will be terminated.
    2. If the current node is bigger than its parent, then the algorithm will be terminated.

pop()

2-1. Heap Operations – Pop()

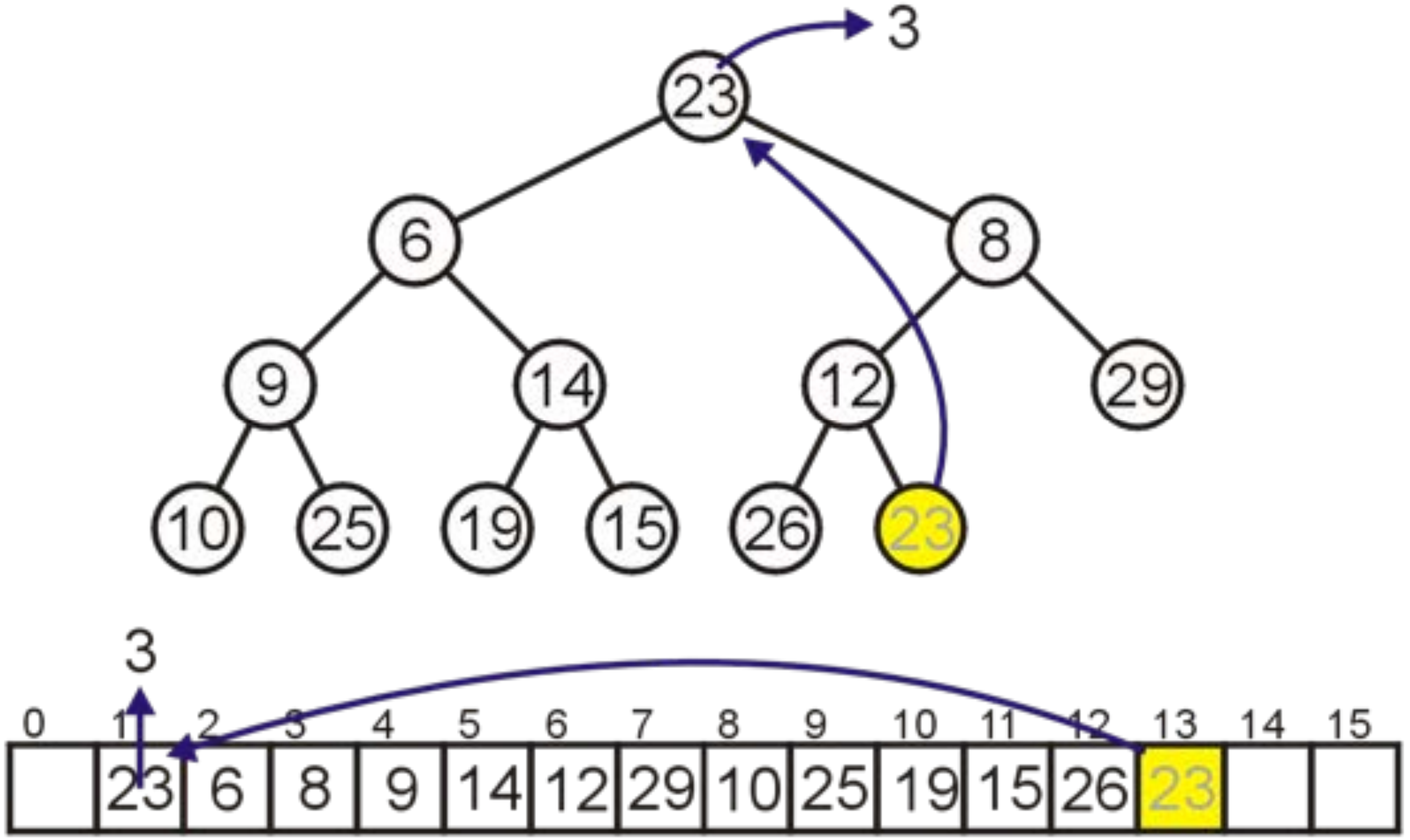
- As before, popping the top element at index 1.
- Move the last entry to the top, whose index is 1.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	6	8	9	14	12	29	10	25	19	15	26	23		

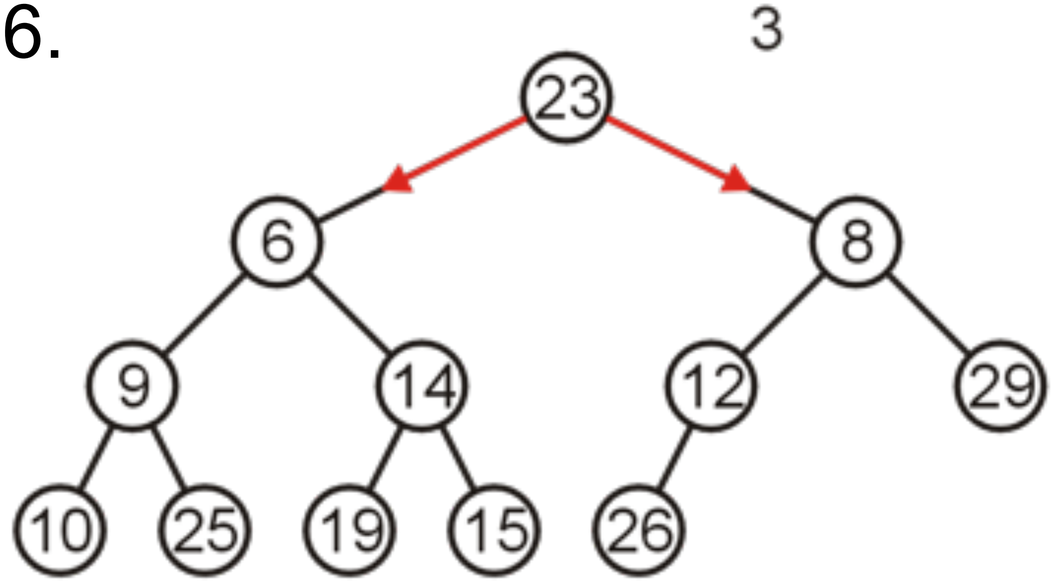
2-1. Heap Operations – Pop()

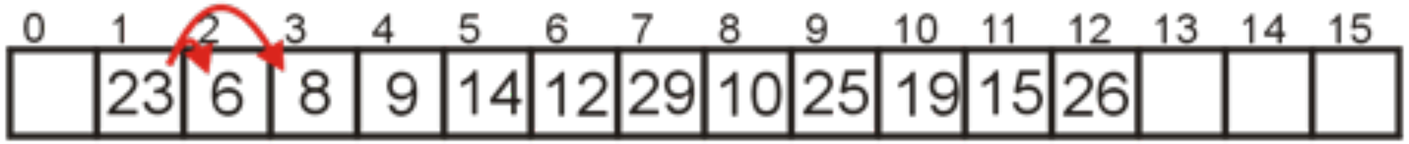
- As before, popping the top element at index 1.
- Move the last entry to the top, whose index is 1.



2-1. Heap Operations – Pop()

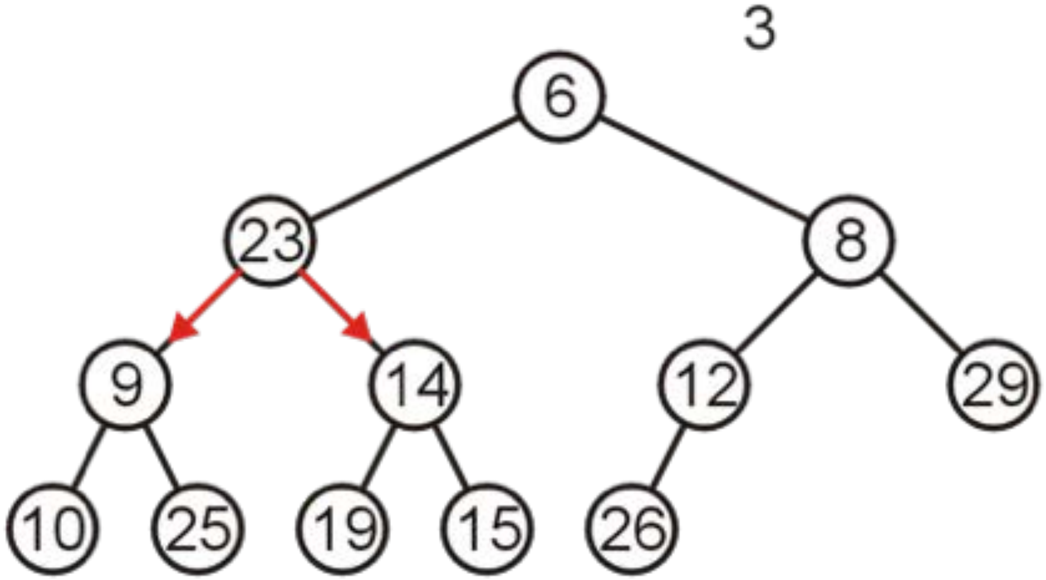
- Now percolate down. Compare Node 1 with its children: Nodes 2 and 3. Swap with the smallest one.
  - Swap 23 and 6.

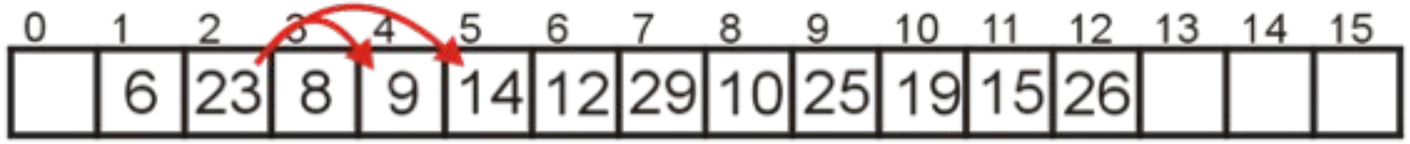




2-1. Heap Operations – Pop()

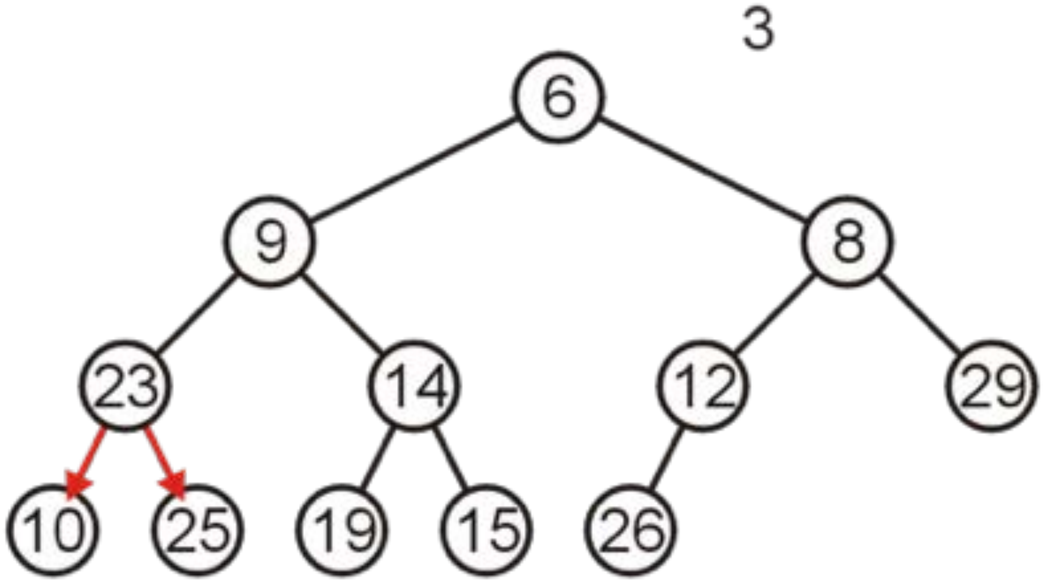
- Compare Node 2 with its children: Nodes 4 and 5
  - Swap 23 and 9.

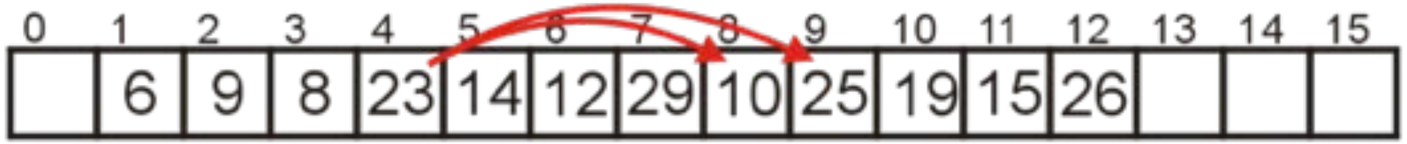




2-1. Heap Operations – Pop()

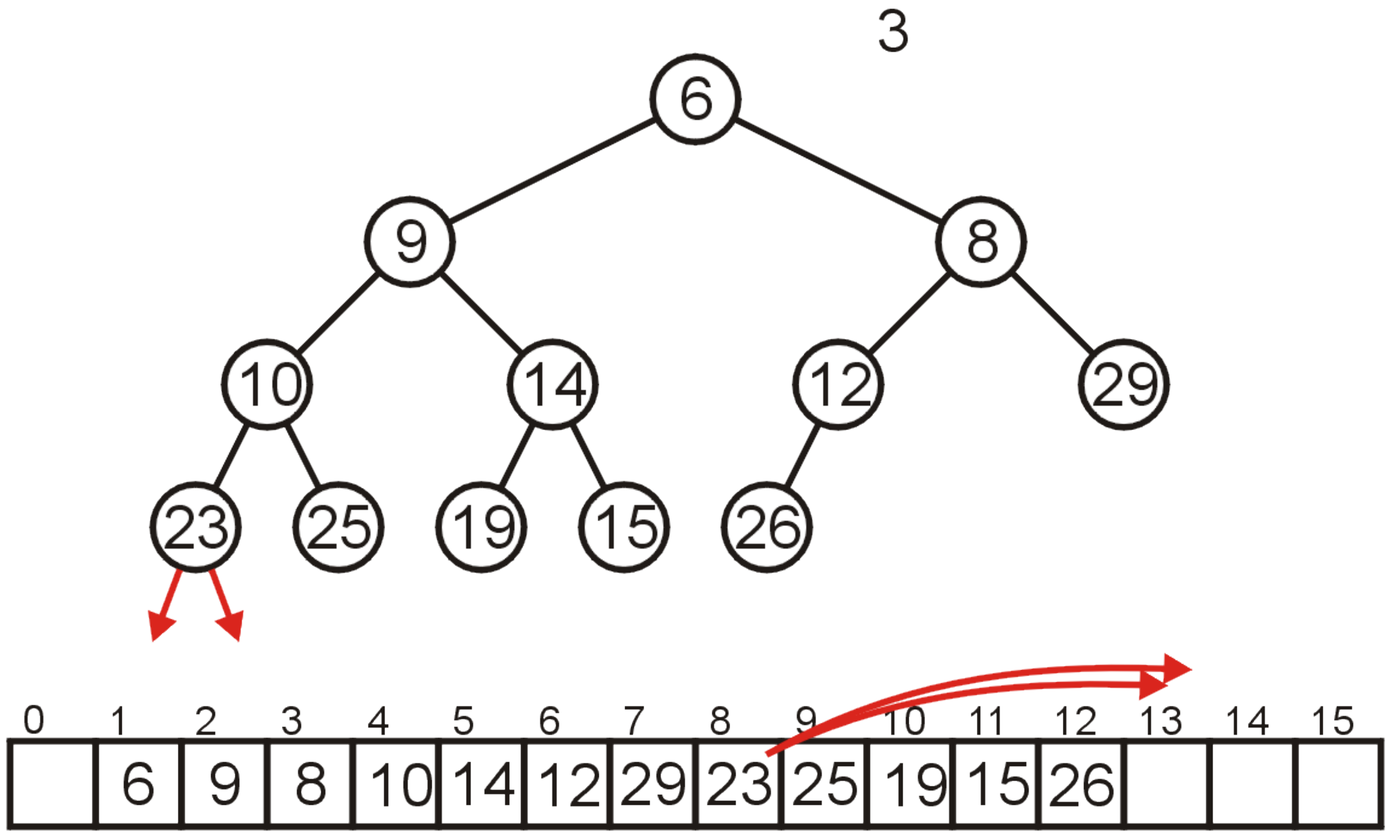
- Compare Node 4 with its children: Nodes 8 and 9.
  - Swap 23 and 10.





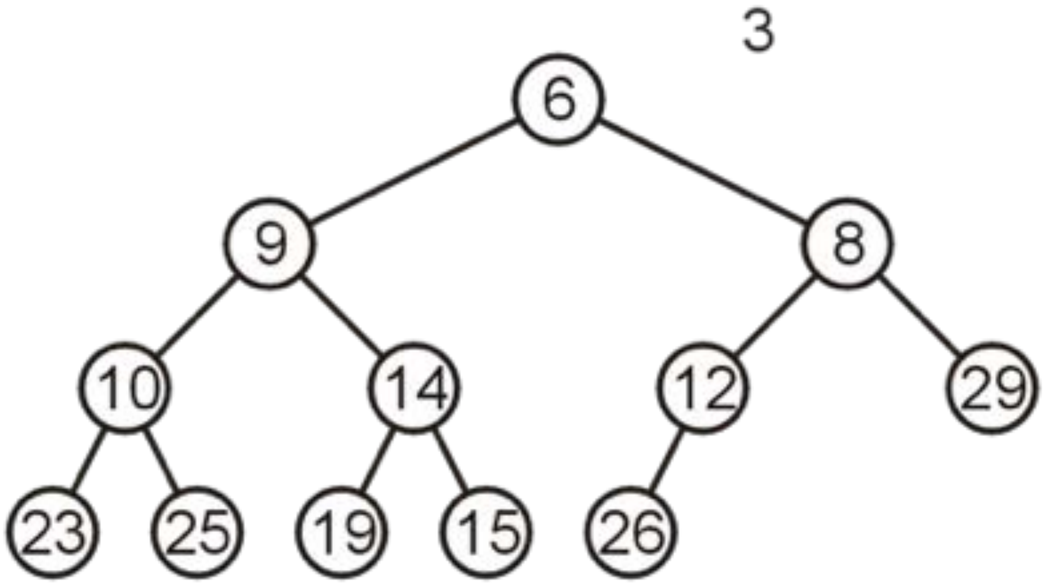
2-1. Heap Operations – Pop()

- The children of Node 8 are beyond the end of the array:
  - Stop.



2-1. Heap Operations – Pop()

- The result is a min-heap. It is still a complete tree.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	6	9	8	10	14	12	29	23	25	19	15	26			

2-1. Heap Operations – Pop() Algorithm

1. Pop the top element at index 1.
2. Move the last entry to the top, whose index is 1.
3. Compare the current nodes with its two children, and swap with the smallest one, repeatedly.
    1. If the current node is smaller than both two children, then the algorithm will be terminated.
    2. If the current node has no children, then the algorithm will be terminated.

Run-time Analysis

- Accessing the top object is $\Theta(1)$, since it is an array.
- Popping the top object is $\mathbf{O}(\ln(n))$
  - We copy something that is already in the lowest depth—it will likely be moved back to the lowest depth
- Pushing an object is also $\mathbf{O}(\ln(n))$
  - If we insert an object less than the root, it will be moved up to the top
- Space complexity $\mathbf{O}(n)$
- *So binary heap is a better implementation of priority queue*

Run-time Analysis (Push)

- (Worst) If we are inserting an object less than the root (at the front), then the run time will be $\Theta(\ln(n))$
- (Best) If we insert at the back (greater than any object) then the run time will be $\Theta(1)$
- (Average) The run time will be $\Theta(1)$. **Why?**

- For a complete tree, $h = \Theta(\ln(n))$.

Run-time Analysis (Average - Push)

- With each percolation, it will move an object past half of the remaining entries in the tree
  - Therefore after one percolation, it will probably be past half of the entries, and therefore *on average* will require no more percolations

$$\frac{1}{n}\sum_{k=0}^{h}(h-k)2^k = \frac{2^{h+1}-h-2}{n}$$

$$= \frac{n-h-1}{n} = \Theta(1)$$

- Therefore, we have an average run time of $\Theta(1)$
- What would be the average run time of pop then?
  - The answer is O(ln(n))

$$\frac{1}{n}\sum_{k=0}^{h} k \cdot 2^k$$

Build Heap

2-1. Build Heap – Simple Method

- Task: Given a set of n keys, build a heap all at once
- Approach 1:
  - Repeatedly perform push
- Complexity
  - $O(n \ln(n))$

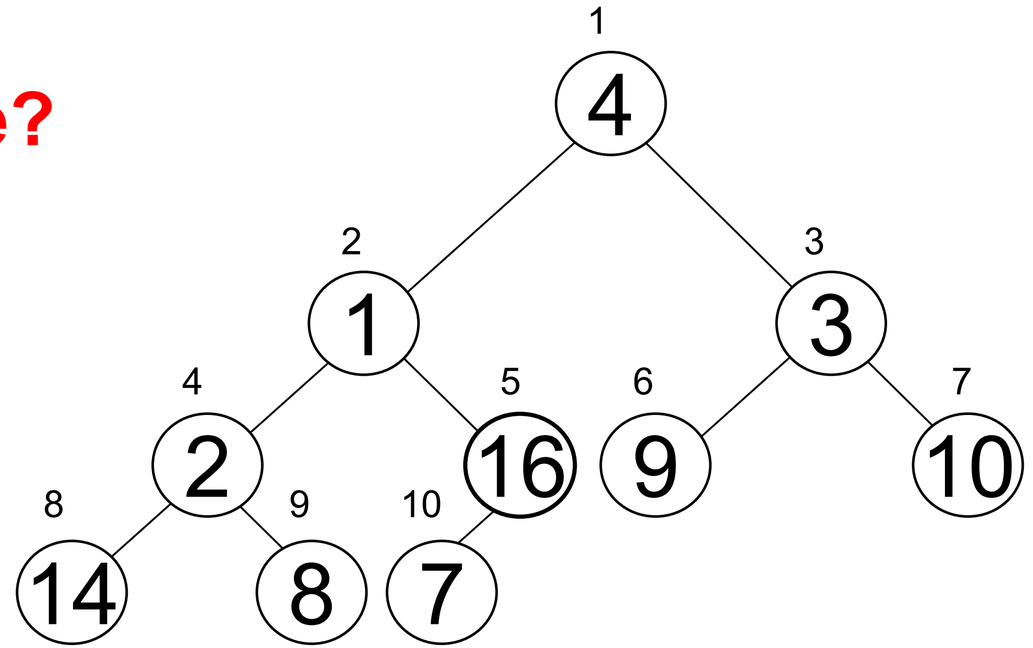
2-1. Build Heap – Floyd's Method

- Task: Given a set of n keys, build a heap all at once
- Approach 2:
  - Floyd's Method
  - Put all the keys in a binary tree first and then fix the heap property!
  - You only need to do it at most half keys. Why?
- **Complexity !!!**
  - $\boldsymbol{O(n)}$

2-1. Build Heap – Floyd's Method

- Task: Given a set of n keys, build a heap all at once
- Approach 2:
  - Floyd's Method. **Why half of size?**

```
buildHeap(){
   for (i=size/2; i>0; i--)
      percolateDown(Array[i]);
}
```

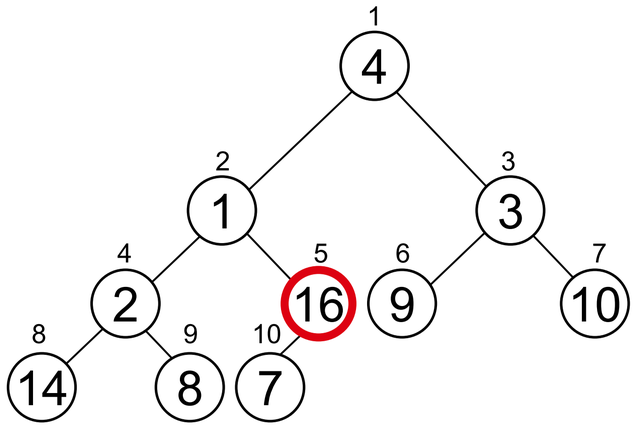


A:

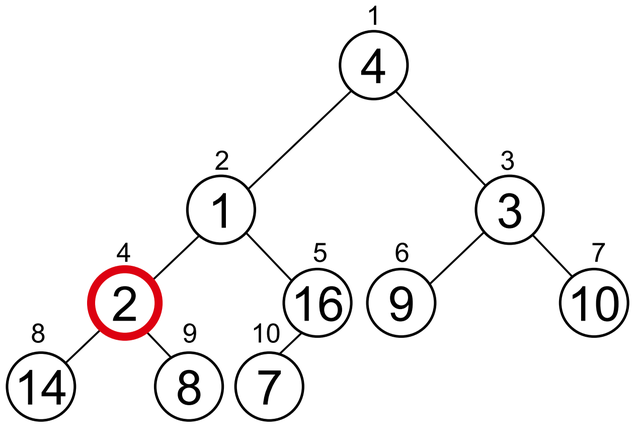
4	1	3	2	16	9	10	14	8	7

2-1. Build Heap – Floyd's Method (max heap)

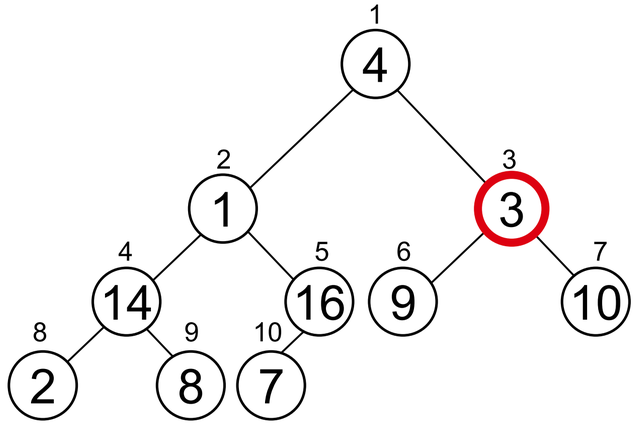
$i = 5$



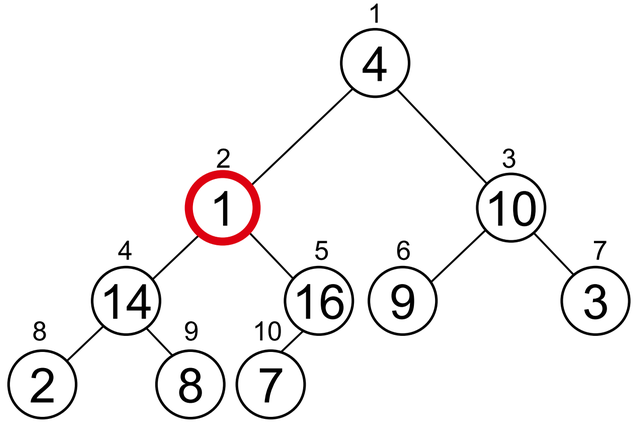
$i = 4$



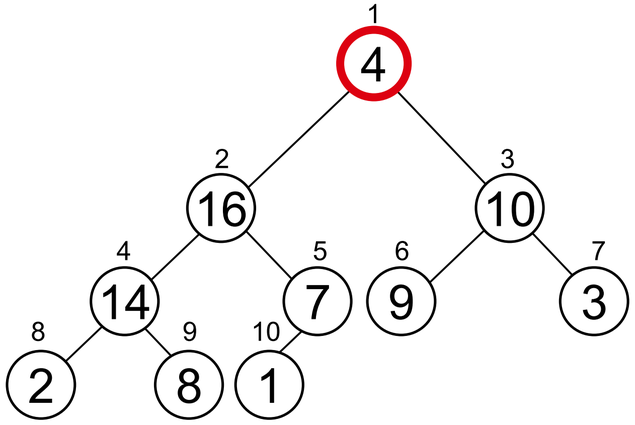
$i = 3$

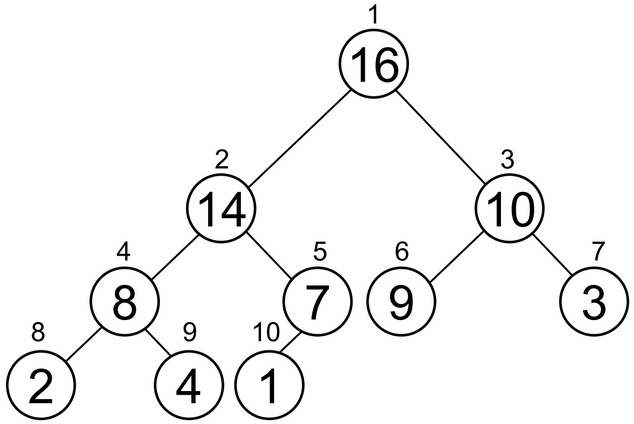


$i = 2$



$i = 1$

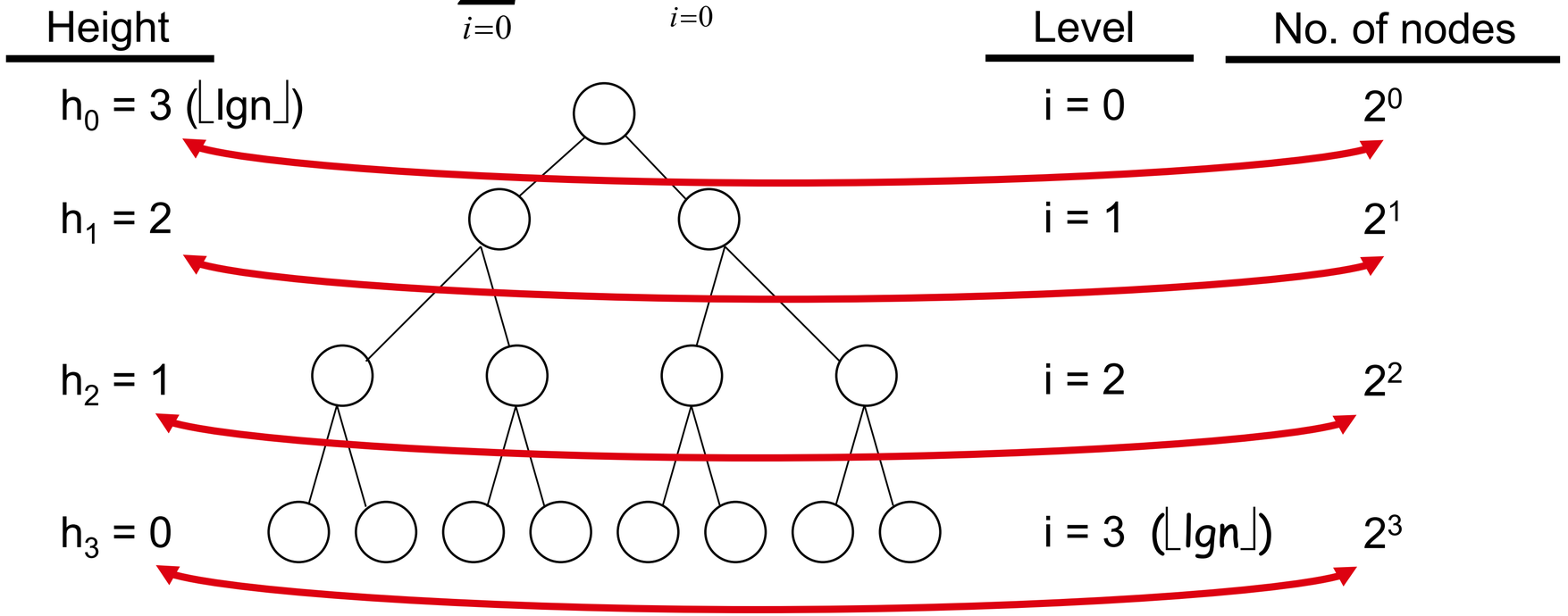




2-1 Running Time of Floyd's Method

- HEAPIFY takes O(h) $\Rightarrow$ the cost of HEAPIFY on a node i is proportional to the height of the node i in the tree

$$\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i = \sum_{i=0}^{h} 2^i (h-i) = O(n)$$



$h_i = h - i$ height of the heap rooted at level i

$n_i = 2^i$ number of nodes at level i

2-1 Running Time of Floyd's Method

$$T(n) = \sum_{i=0}^{h} n_i h_i$$ Cost of HEAPIFY at level i $*$ number of nodes at that level

$$= \sum_{i=0}^{h} 2^i (h - i)$$ Replace the values of n_i and h_i computed before

$$= \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$$ Multiply by 2^h both at the nominator and denominator and write 2^i as $\frac{1}{2^{-i}}$

$$= 2^h \sum_{k=0}^{h} \frac{k}{2^k}$$ Change variables: k = h - i

$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$$ The sum above is smaller than the sum of all elements to ∞ and h = lgn

$$= O(n)$$ The sum above is smaller than 2

Running time of BUILD-MAX-HEAP: T(n) = *O*(n)

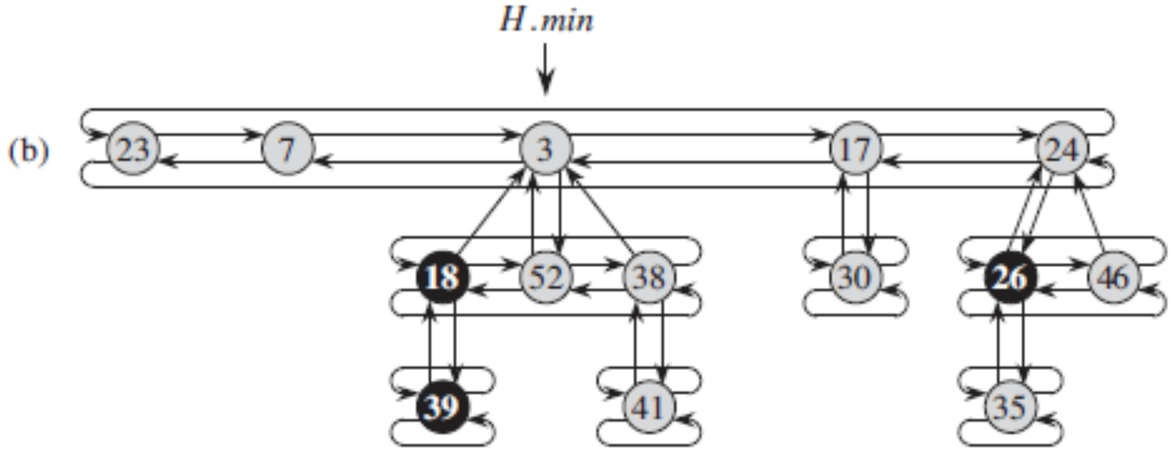
2-1. Build Heap – Floyd's Method

- No percolation for the leaf nodes (n/2 nodes)
- At most n/4 nodes percolate down 1 level
  At most n/8 nodes percolate down 2 levels
  At most n/16 nodes percolate down 3 levels
  …

$$1\frac{n}{4}+2\frac{n}{8}+3\frac{n}{16}+\cdots=\sum_{i=1}^{\log n} i\frac{n}{2^{i+1}}=\frac{n}{2}\sum_{i=1}^{\log n}\frac{i}{2^{i}}=n=\Theta(n)$$

Summary 2

- Binary heap
  - Based on priority queue, complete binary tree.
  - Implementation using arrays
  - *Heap != Tree.



Fibonacci Heap

Summary 2 – Binary Heap

	Average	**Worst**
Top	$\Theta(1)$	$\Theta(1)$
Push	$\underline{\Theta(1)}$	$\Theta(\log n)$
Pop	$\mathrm{O}(\log n)$	$\Theta(\log n)$
Build Heap		$\Theta(n)$

6. In a binary heap (a heap that is implemented using a complete binary tree), if a node A is the ancestor of a node B, then we can deduce that the insertion of node A is performed before the insertion of node B. （ ）
在一个二叉堆（用完全二叉树实现的堆）中，节点 A 是节点 B 的祖先，那么可以推断出节点 A 先于节点 B 插入这个二叉堆中。（ ）

Heap Sort

2-2. Heapsort

- Heapsort
  - Place the objects into a heap
    - $O(n)$ time
  - Repeatedly popping the top object until the heap is empty
    - $O(n \ln(n))$ time
  - Time complexity: $O(n \log n)$

2-2. Heapsort

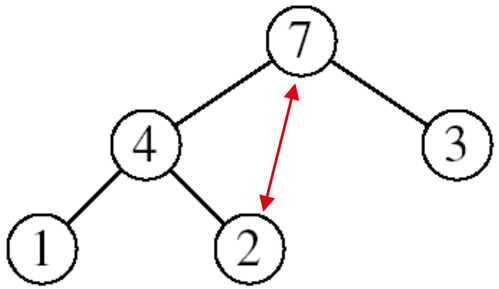
- Goal:
  - Sort an array using heap representations (increasing order)
- Idea:
  - Convert the unordered array to a **max-heap.**
  - Swap the root (the maximum element) with the last element in the array *-> like the pop() operation*.
  - “Discard” this last node by decreasing the heap size.
  - Call MAX-HEAPIFY on the new root. *-> like the pop() operation*
  - Repeat this process until only one node remains.

2-2. Heapsort Algorithm

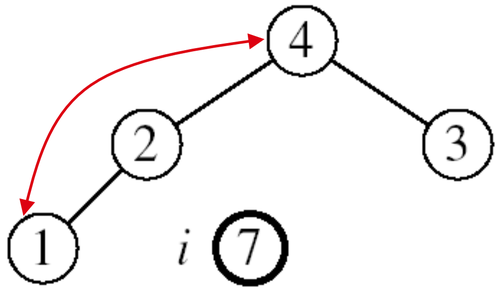
```
1.  BUILD-MAX-HEAP(A)                          O(n)
2.   for i ← length[A] downto 2
3.        do exchange A[1] ⇔ A[i]
4.             MAX-HEAPIFY(A,1,i-1)           O(lgn)
```

n-1 times

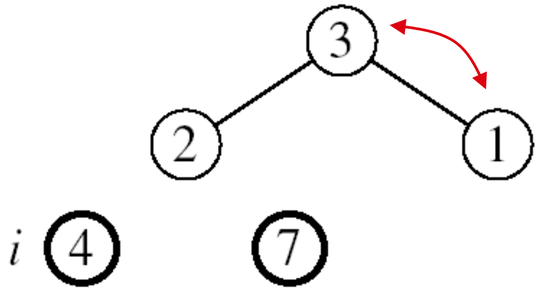
2-2. Heap Sort Example 1



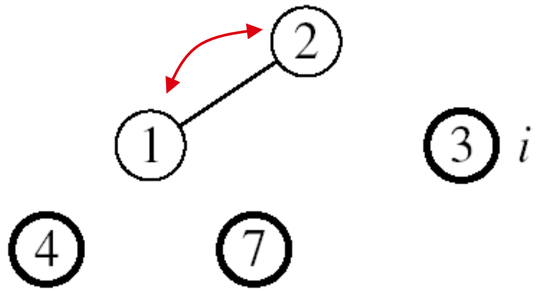
MAX-HEAPIFY(A, 1, 4)

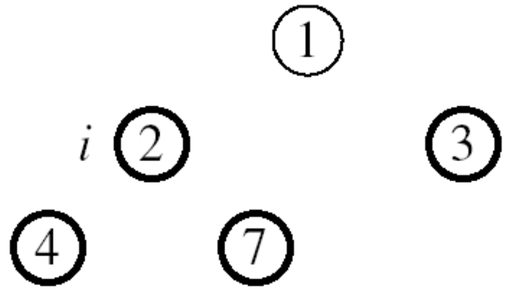


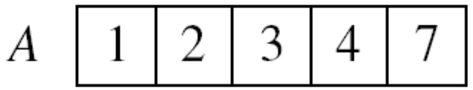
MAX-HEAPIFY(A, 1, 3)



MAX-HEAPIFY(A, 1, 2)







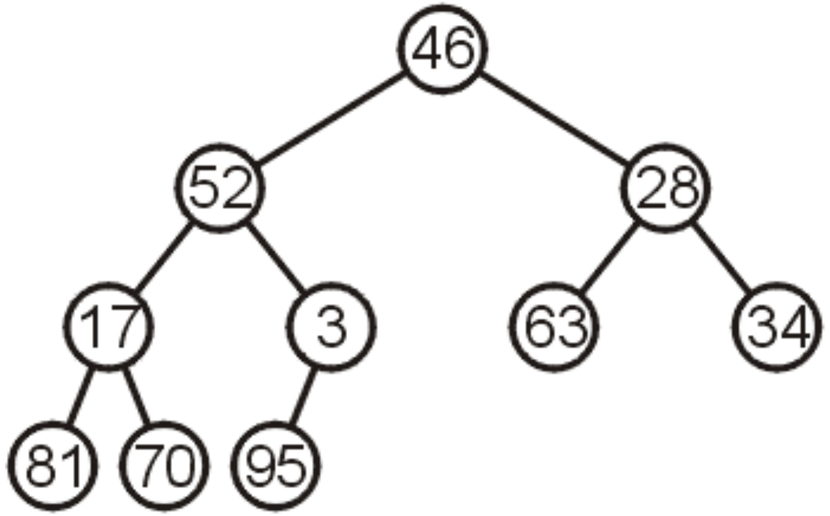
MAX-HEAPIFY(A, 1, 1)

2-2. Heap Sort Example 2

- First, we must convert the unordered array with $n = 10$ elements into a max-heap.

46	52	28	17	3	63	34	81	70	95

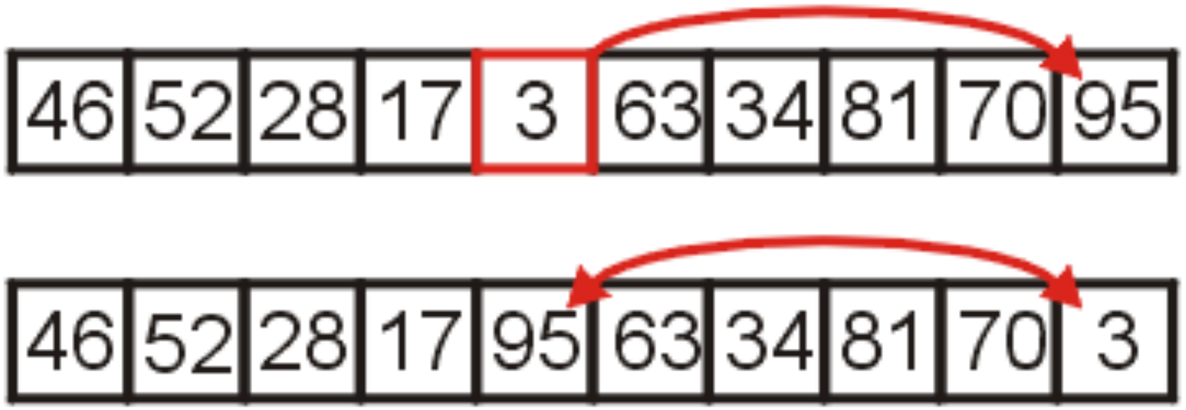
- None of the leaf nodes need to be percolated down, and the last non-leaf node is in position $n/2$



- Thus we start with position $10/2\text{-}1 = 5$

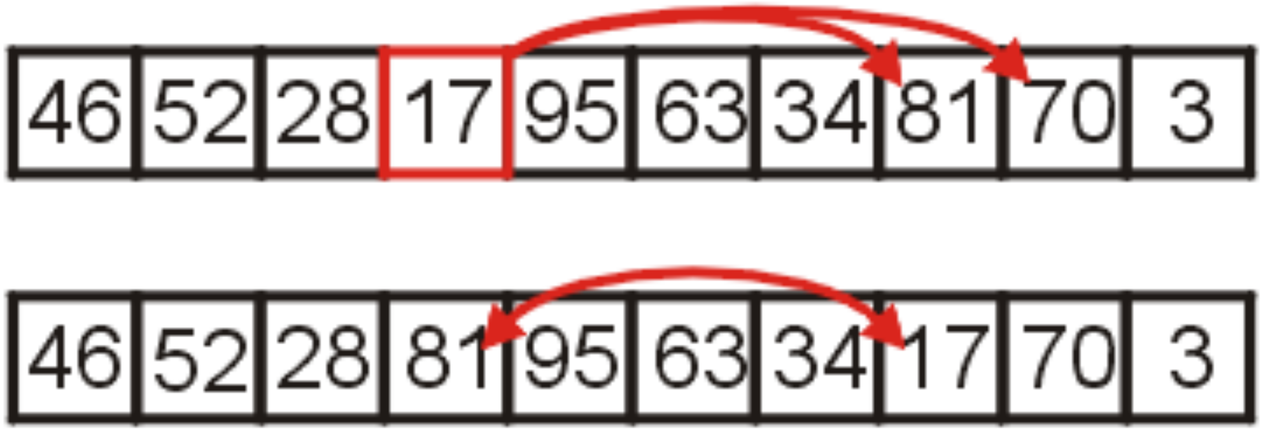
2-2. Example Heap Sort – Build Heap

- We compare 3 with its child and swap them



2-2. Example Heap Sort – Build Heap

- We compare 17 with its two children and swap it with the maximum child (81)

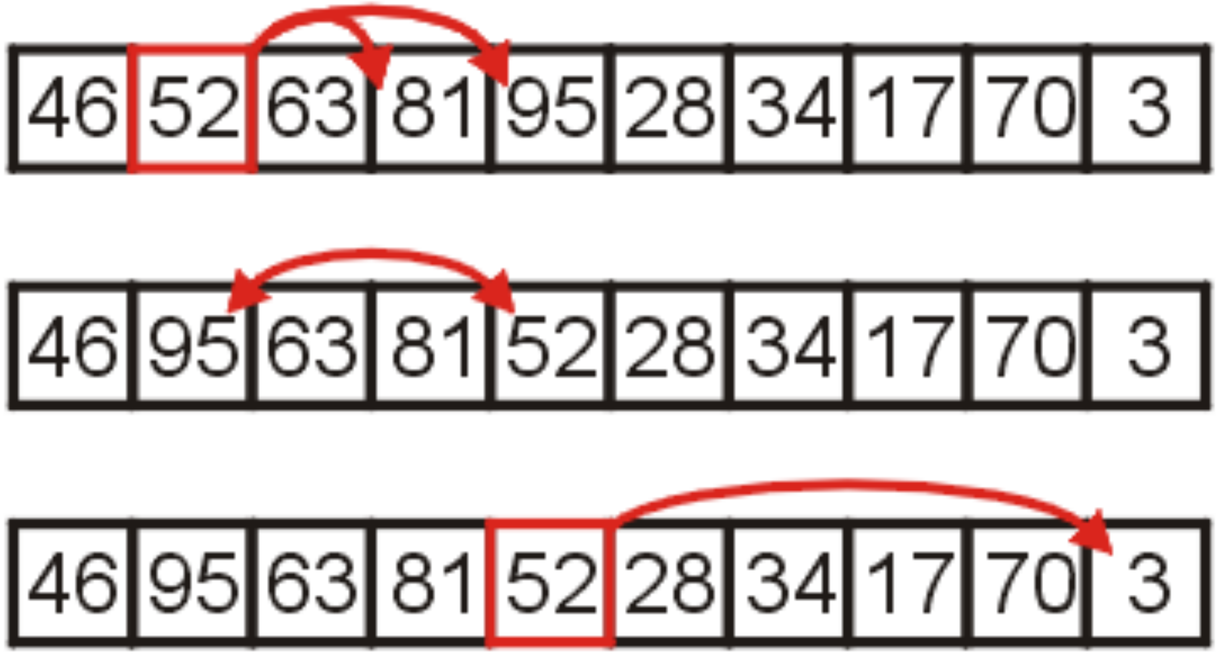


2-2. Example Heap Sort – Build Heap

- We compare 28 with its two children, 63 and 34, and swap it with the largest child.

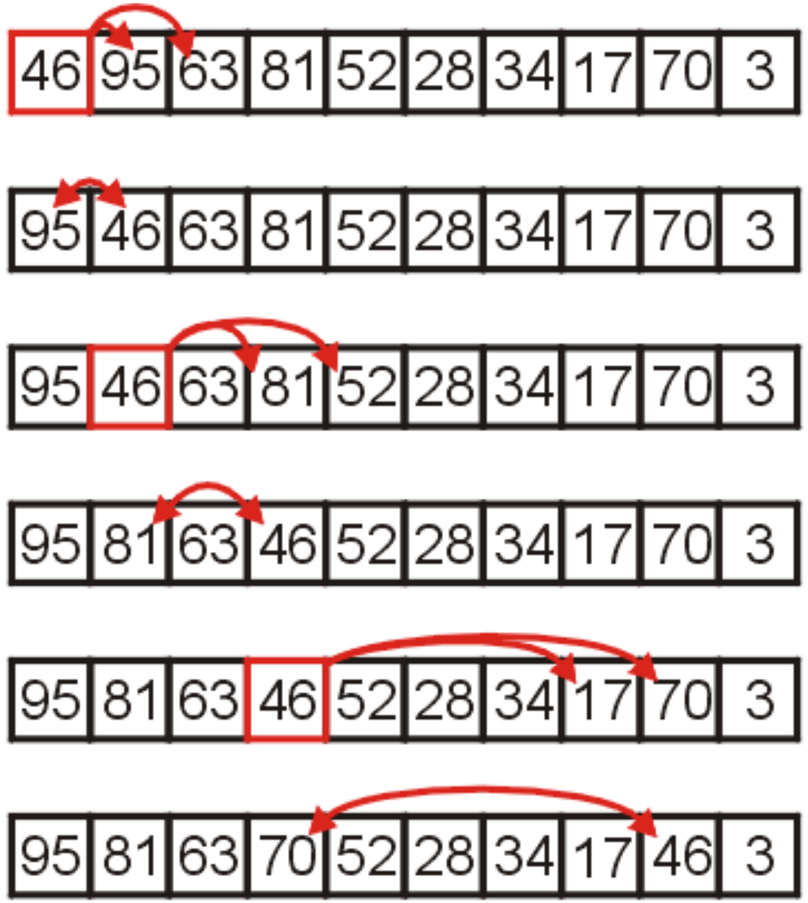
2-2. Example Heap Sort – Build Heap

- We compare 52 with its children, swap it with the largest
  - Recursing, no further swaps are needed



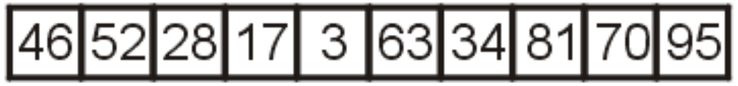
2-2. Example Heap Sort – Build Heap

- Finally, we swap the root with its largest child, and recurse, swapping 46 again with 81, and then again with 70

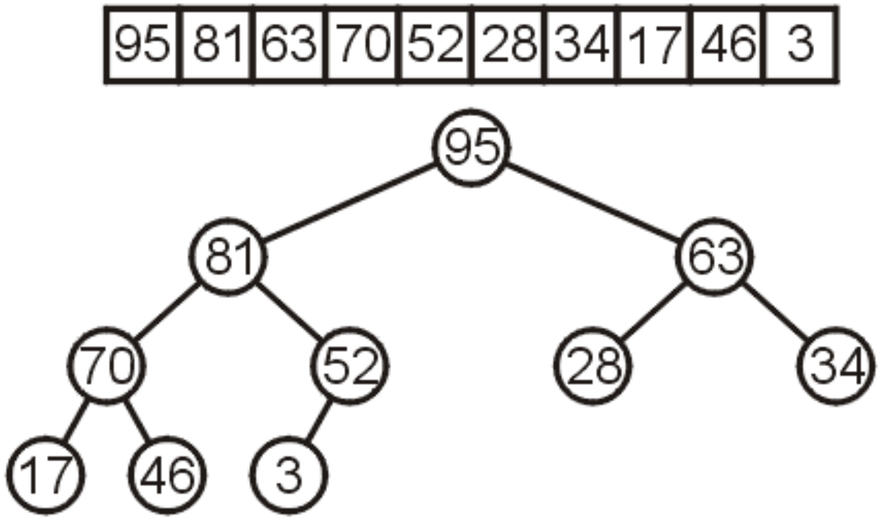


2-2. Example Heap Sort – Build Heap

- We have now converted the unsorted array:



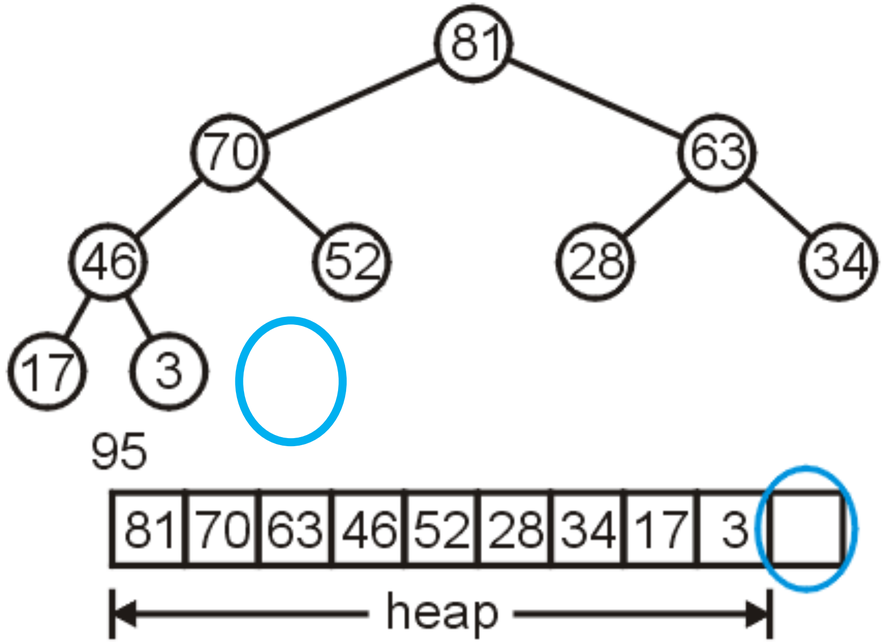
into a max-heap:



2-2. Example Heap Sort – Get Largest

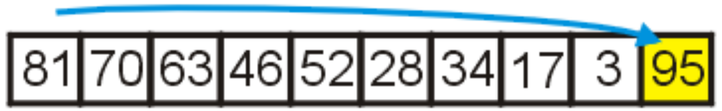
- We pop the maximum element of this heap

- This leaves a gap at the back of the array:

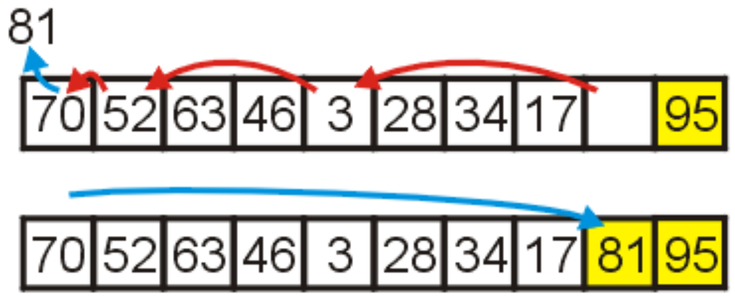


2-2. Example Heap Sort – Get Largest

- This is the last entry in the array, so why not fill it with the largest element?

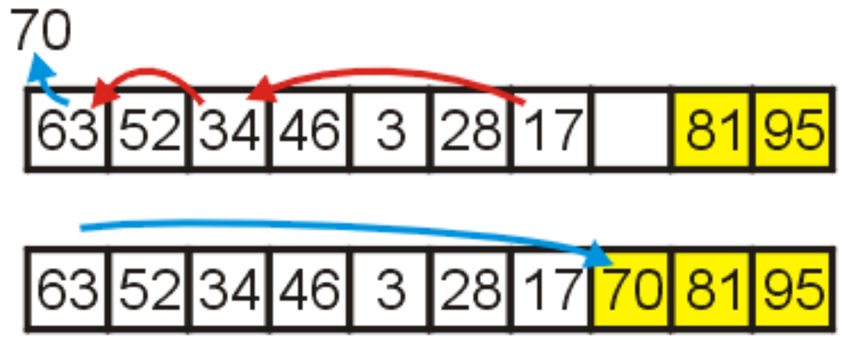


- Repeat this process: pop the maximum element, and then insert it at the end of the array:

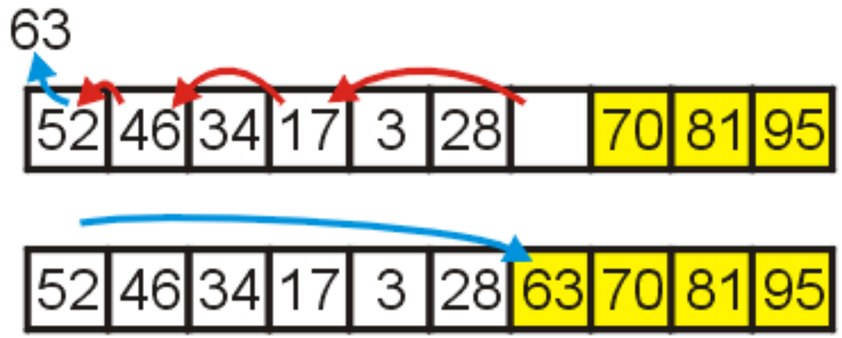


2-2. Example Heap Sort – Get Largest

- Repeat this process
  - Pop and append 70

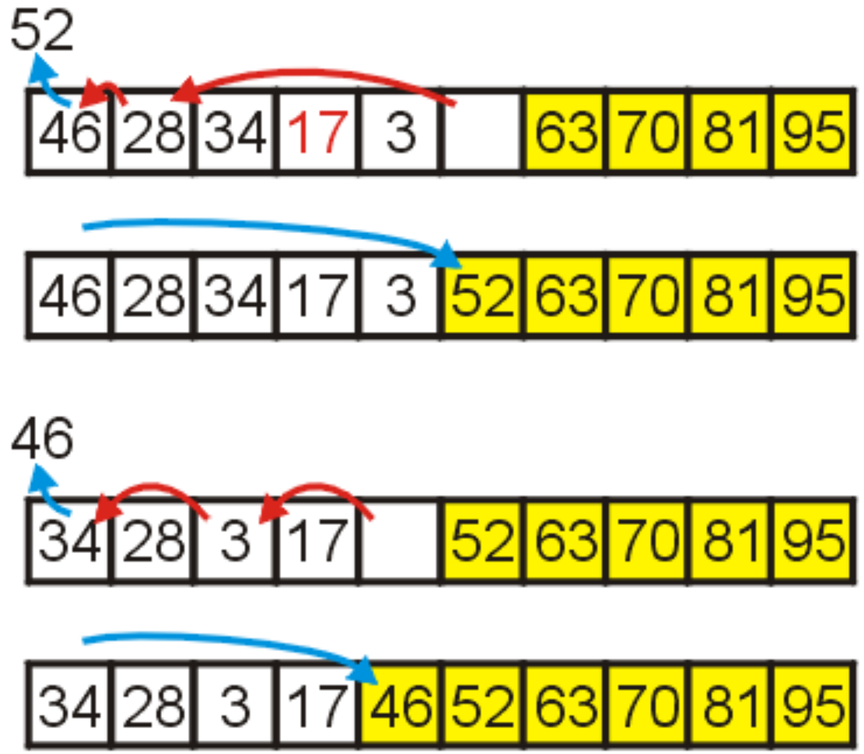


  - Pop and append 63



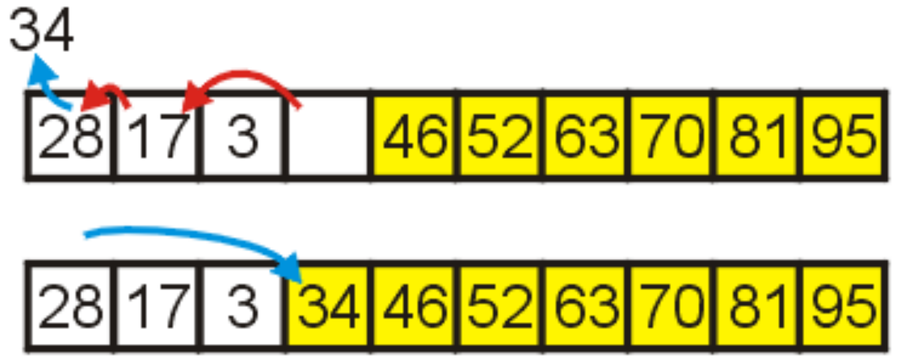
2-2. Example Heap Sort – Get Largest

- We have the 4 largest elements in order
  - Pop and append 52
  - Pop and append 46

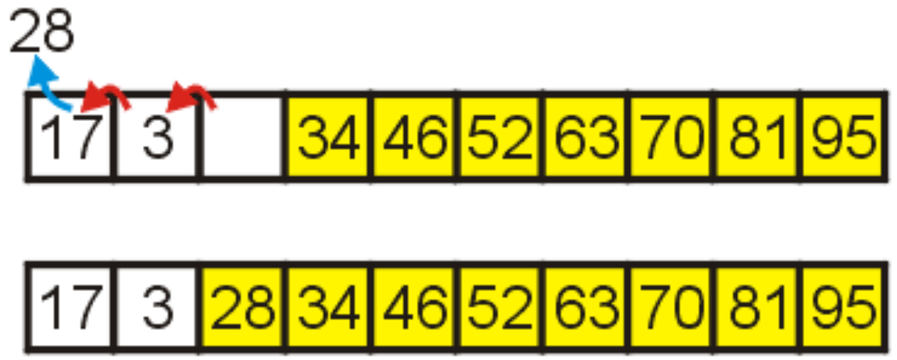


2-2. Example Heap Sort – Get Largest

- Continuing...
  - Pop and append 34

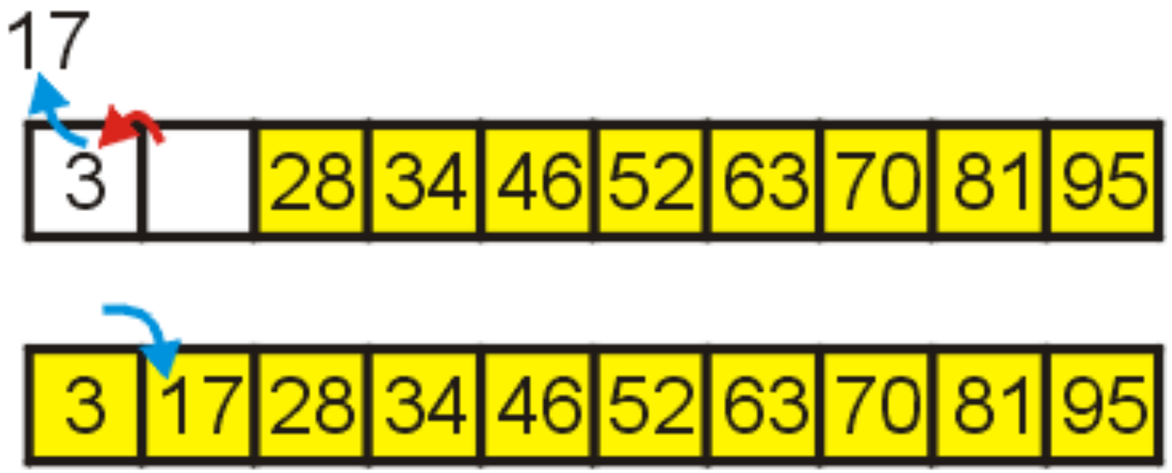


  - Pop and append 28



2-2. Example Heap Sort – Get Largest

- Finally, we can pop 17, insert it into the 2nd location, and the resulting array is sorted.



Summary 3

- Heap Sort
  - Time Complexity: O(nln(n))
  - Step: (1) Build Heap (2) Pop and Add it to the back
  - Additional Space Complexity: O(1), in-place sorting!
    - Not stable

Binary Search Tree

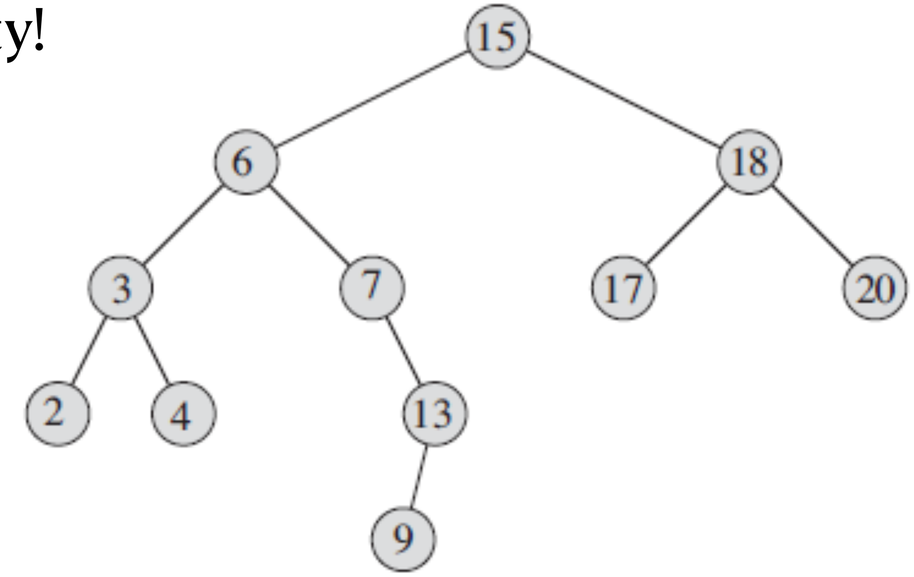
What is the purpose of Binary Search Trees?

- If we implement an Abstract Sorted List using an **array** or a **linked list**, we will have operations which are **O**(*n*).
- Binary Search Trees have operations which are **O**(h), where h is the height of the tree.

Binary Search Trees

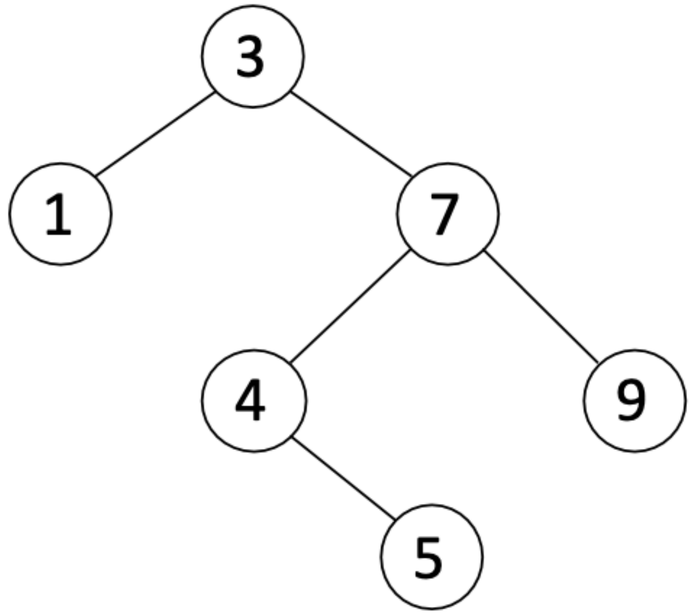
- All objects in the left sub-tree to be less than the object stored in the root node.
- All objects in the right sub-tree to be greater than the object in the root object.
- The two sub-trees are themselves binary search trees.

BST can be empty!



4. The following tree is a binary search tree. （ ）

下列树是二叉搜索树。（ ）

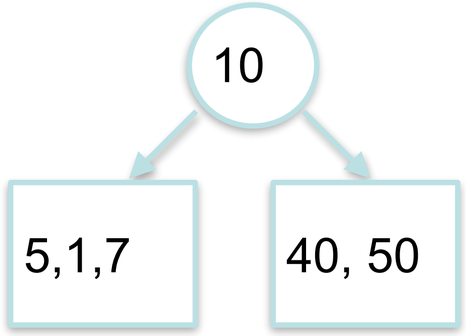


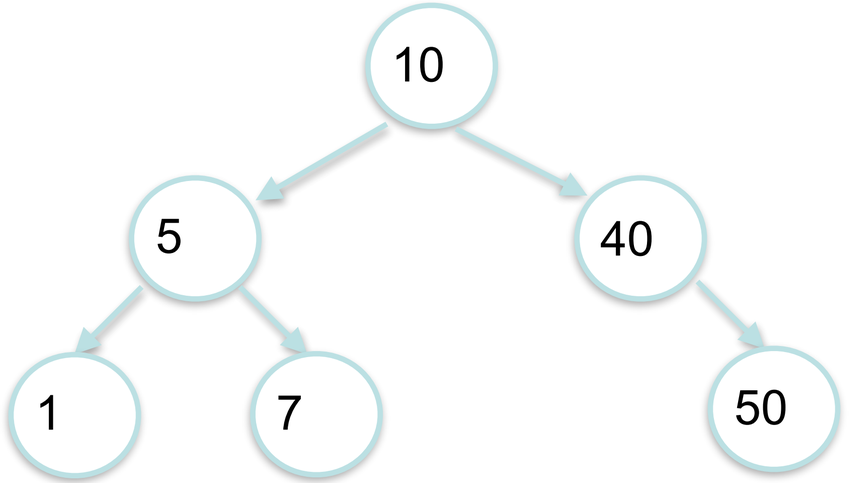
Properties of BST

- Given a pre-order(left to right) sequence of the BST, we can uniquely certain the structure of it.

Question:
The preorder traversal of a binary search tree is {10, 5, 1, 7, 40, 50}, construct the BST.

Method:
- The first element of preorder traversal is always root.
- Let the index of first element which is greater than root is “i”.
- The value between root and “i” belongs to the left subtree.
- The value from “i” to n-1 belongs to the right subtree.

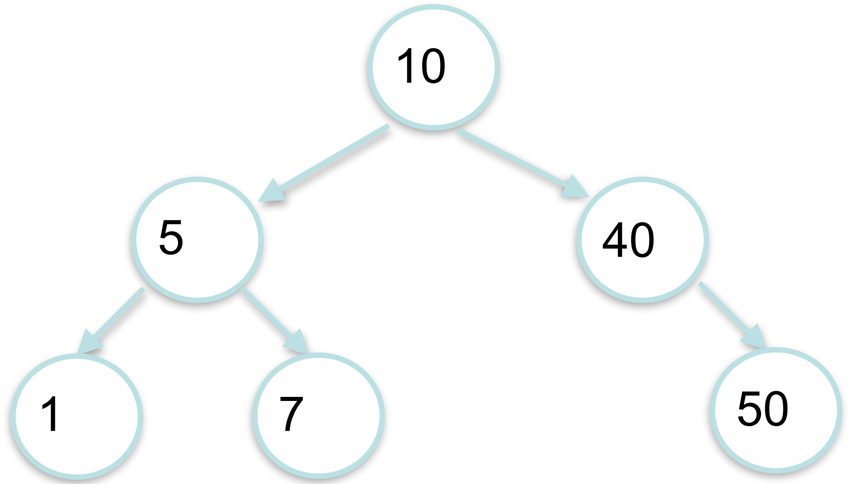




Properties of BST

- The in-order sequence of BST are ascendant.

Example:



In-order sequence: 1, 5, 7, 10, 40, 50

Question:
(T or F) For any two BST with same keys, they always have the same in-order sequence.

Answer: T

For two BST with same keys, do they always have the same tree structure?

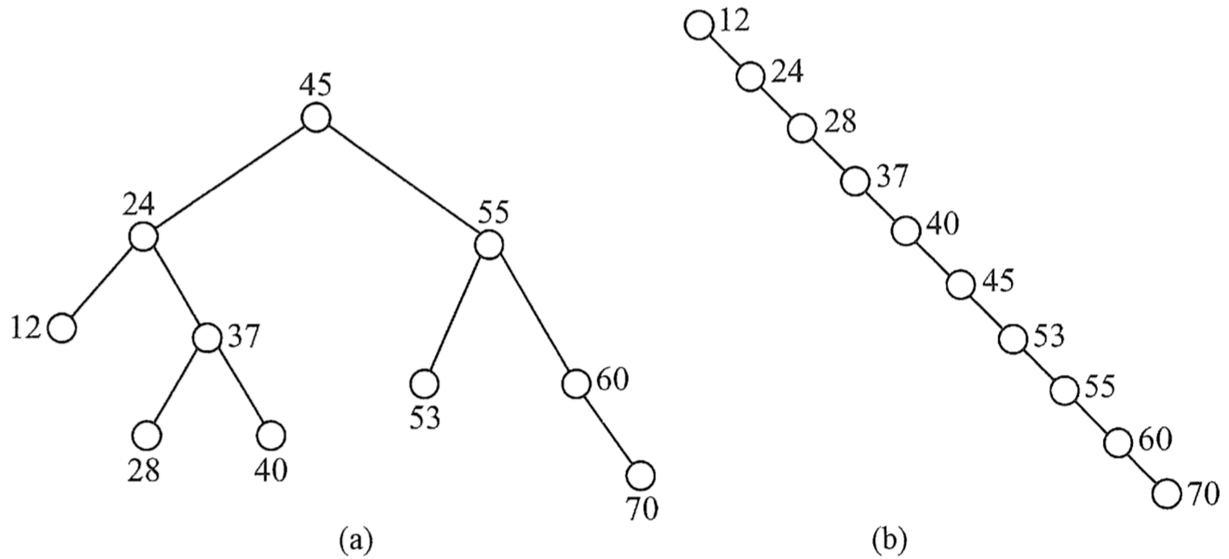
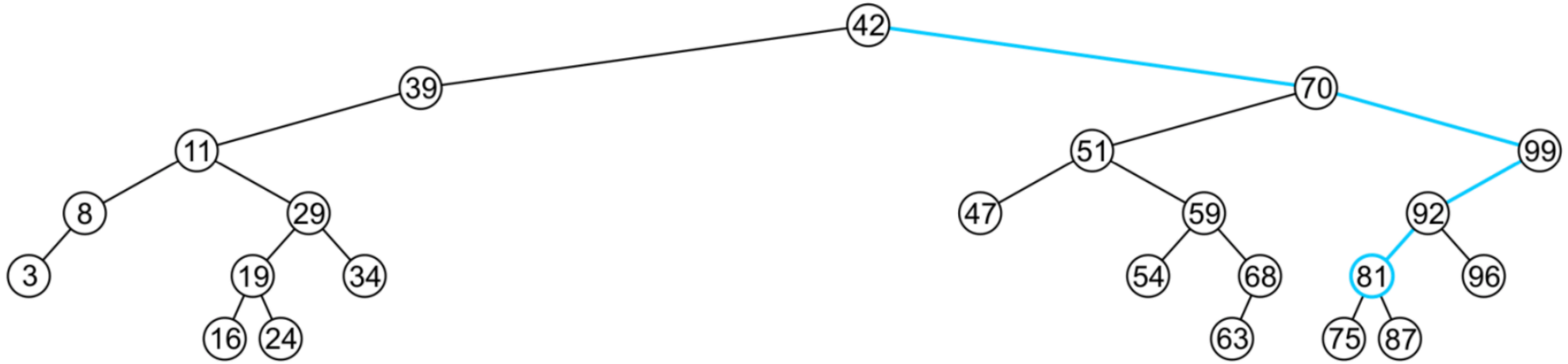


图 5.25　相同关键字组成的不同二叉排序树

Operations in BST

- Search
- Insert
- Erase
- Next & Previous

Search in a BST



Recursive:

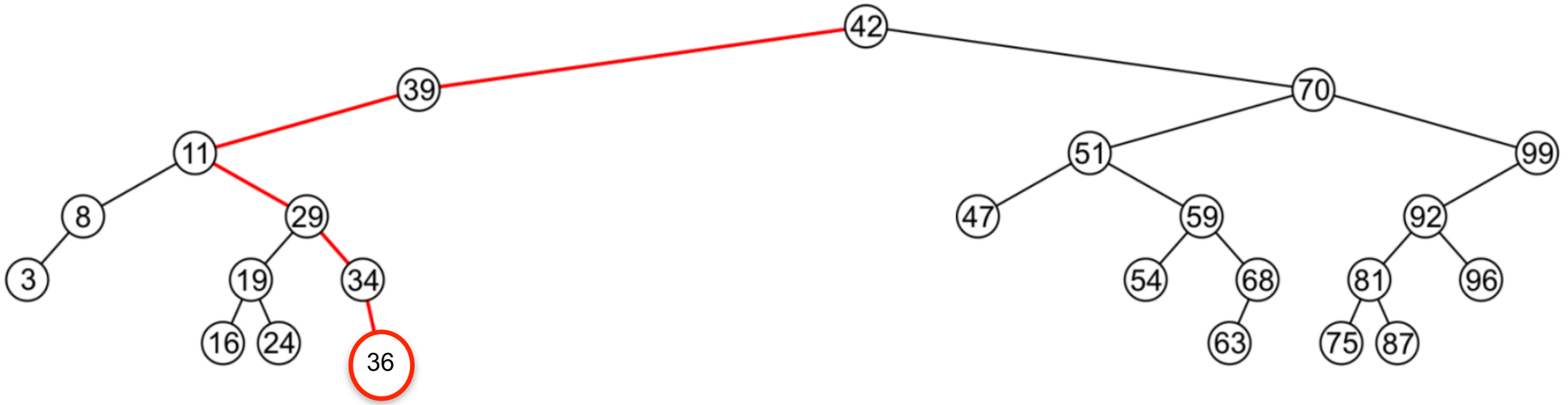
```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2        return x
3  if k < x.key
4        return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

Iterative:

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2        if k < x.key
3              x = x.left
4        else x = x.right
5  return x
```

Insertion

- Search for “L” ends at a null link.
- Create new node.
- Reset links (and increment counts on the way up).

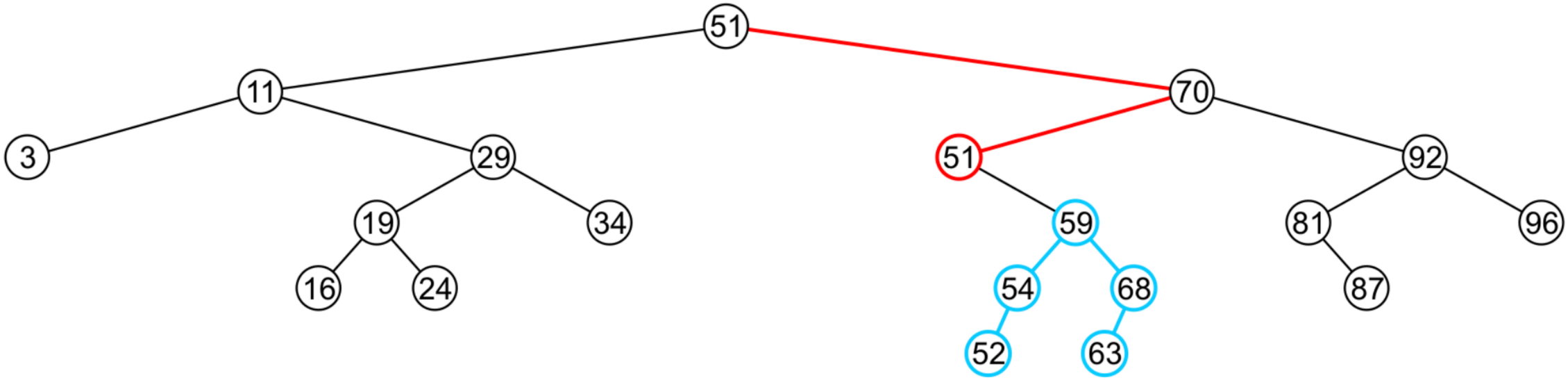


二叉排序树插入操作的算法描述如下：

```
int BST_Insert(BiTree &T,KeyType k){
    if(T==NULL){                          //原树为空，新插入的记录为根结点
        T=(BiTree)malloc(sizeof(BSTNode));
        T->key=k;
        T->lchild=T->rhild=NULL;
        return  1;                        //返回 1，插入成功
    }
    else if(k==T->key)                    //树中存在相同关键字的结点，插入失败
        return  0;
    else if(k<T->key)                     //插入到 T 的左子树
        return  BST_Insert(T->lchild,k);
    else                                  //插入到 T 的右子树
        return  BST_Insert(T->rchild,k);
}
```

Erase

- For 0 children node, erase it directly.
- For 1 children node, promote the sub-tree associated with the child
- For 2 children node:
  - Replace it with the minimum object in the right sub-tree.
  - Erase that object from the right sub-tree.



Question:

- Staring from an empty binary search tree, insertion of the following integers in sequence {10, 20, 15, 25, 30, 16}. What does the BST look like?
- Deleting 20, what does the BST look like?

Question:

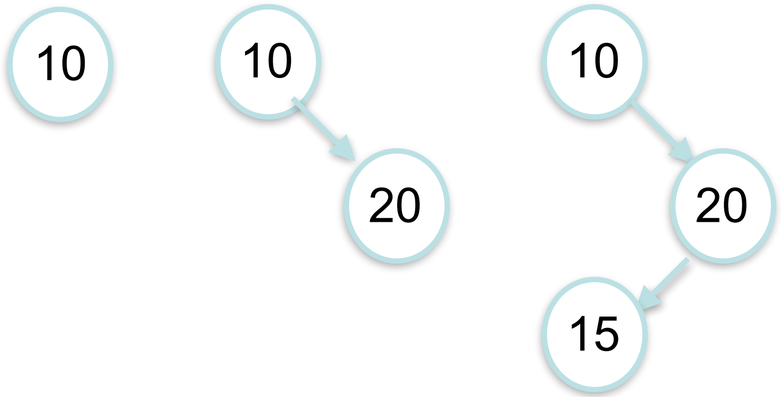
- Staring from an empty binary search tree, insertion of the following integers in sequence {10, 20, 15, 25, 30, 16}. What does the BST look like?
- Deleting 20, what does the BST look like?

Question:

- Staring from an empty binary search tree, insertion of the following integers in sequence {10, 20, 15, 25, 30, 16}. What does the BST look like?
- Deleting 20, what does the BST look like?

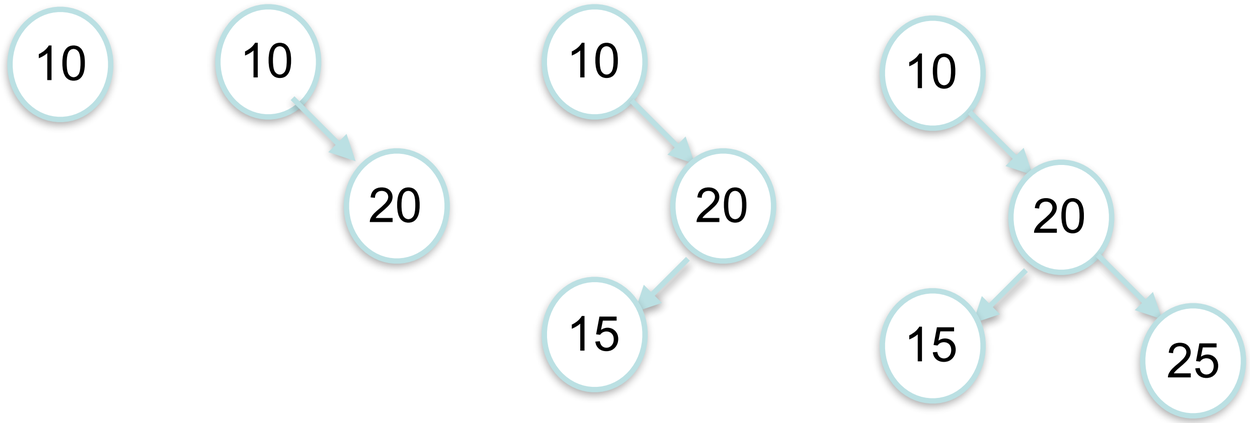
Question:

- Staring from an empty binary search tree, insertion of the following integers in sequence {10, 20, 15, 25, 30, 16}. What does the BST look like?
- Deleting 20, what does the BST look like?



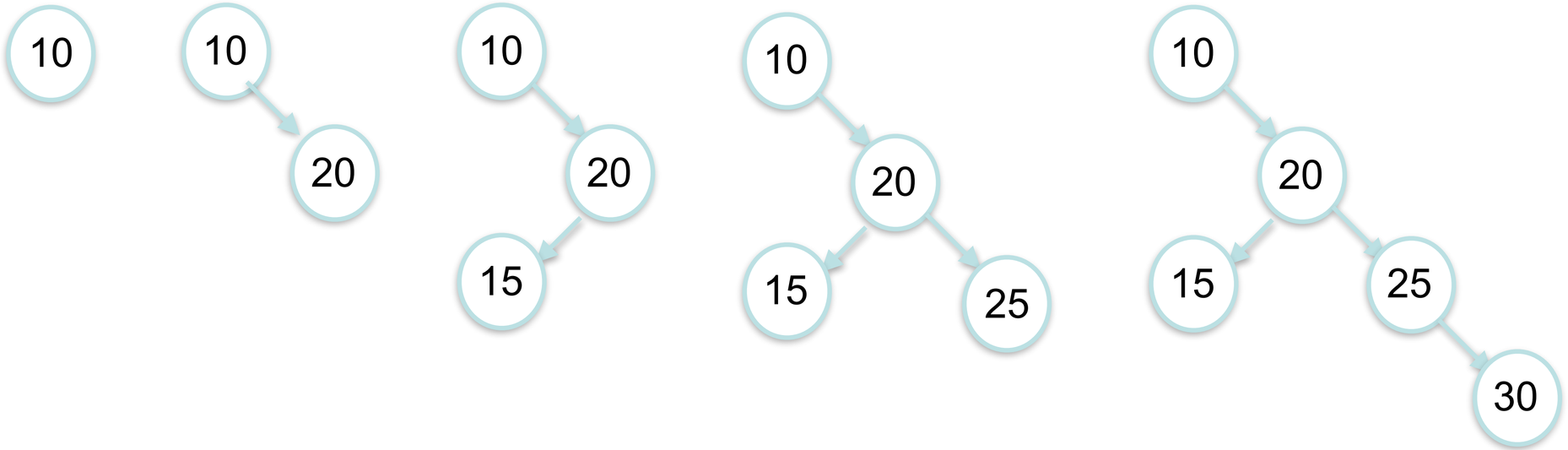
Question:

- Staring from an empty binary search tree, insertion of the following integers in sequence {10, 20, 15, 25, 30, 16}. What does the BST look like?
- Deleting 20, what does the BST look like?



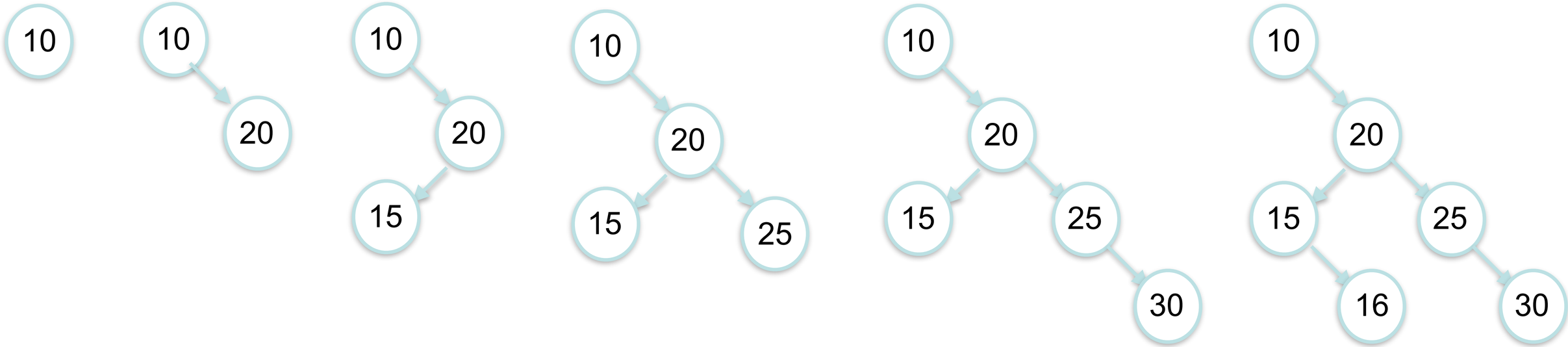
Question:

- Staring from an empty binary search tree, insertion of the following integers in sequence {10, 20, 15, 25, 30, 16}. What does the BST look like?
- Deleting 20, what does the BST look like?



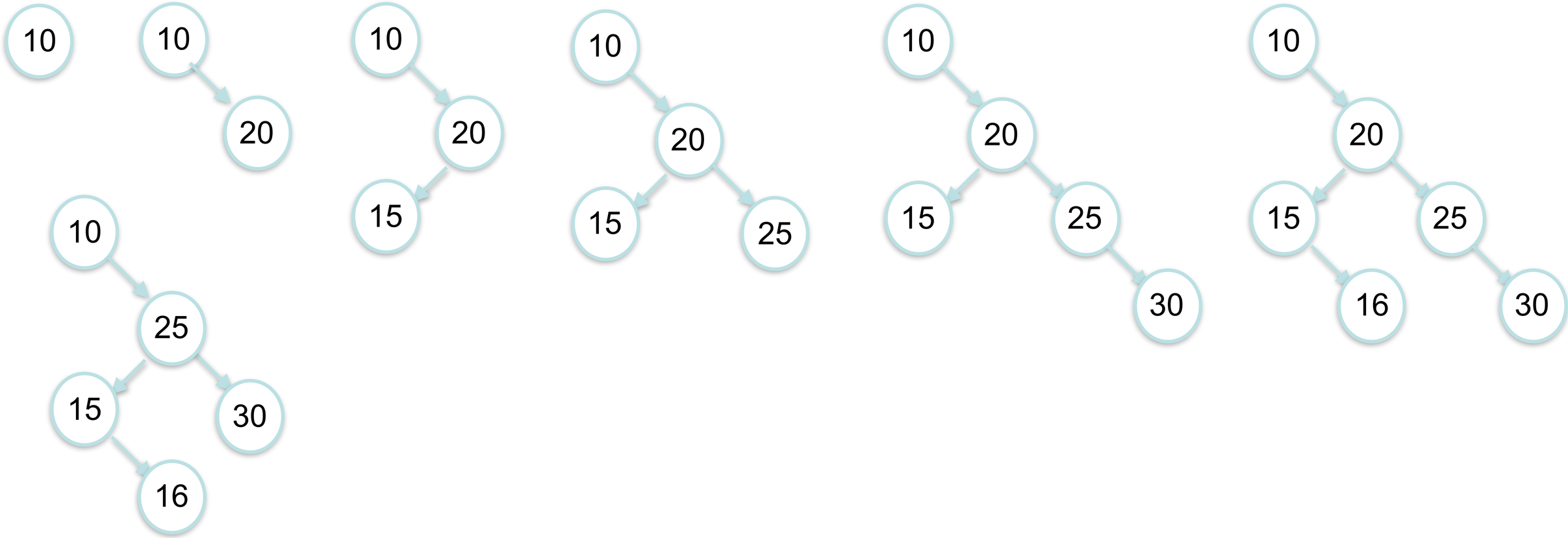
Question:

- Staring from an empty binary search tree, insertion of the following integers in sequence {10, 20, 15, 25, 30, 16}. What does the BST look like?
- Deleting 20, what does the BST look like?



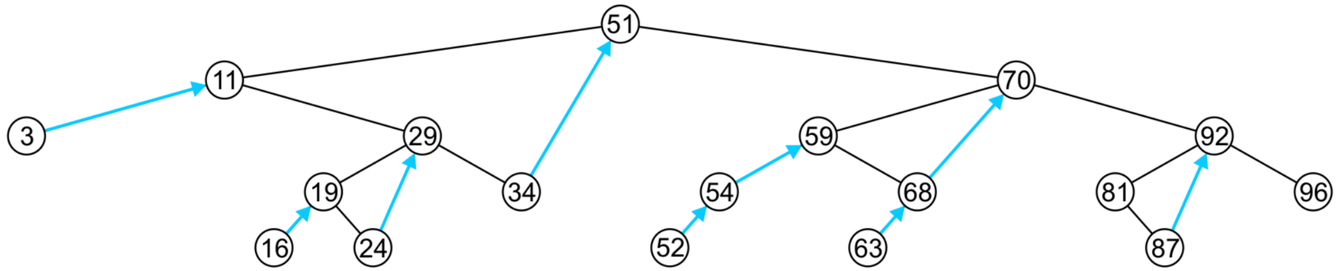
Question:

- Staring from an empty binary search tree, insertion of the following integers in sequence {10, 20, 15, 25, 30, 16}. What does the BST look like?
- Deleting 20, what does the BST look like?



Next

- If the node has the right sub-tree, find the min in the right sub-tree.
- If it does not have the right sub-tree, find the first larger object (if any) that exists in the path from the node to the root




```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2        return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5        x = y
6        y = y.p
7  return y
```

The "**previous**" operation is similar.

3. The in-order depth-first traversal of a binary search tree (BST) produces elements in descending order. （ ）
二叉搜索树的中序深度优先遍历可以产生一个降序的有序序列。（ ）

Disjoint Set

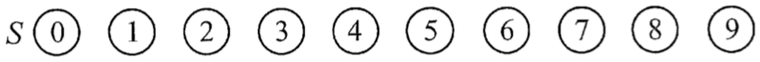
Definition

- 多叉树的森林结构，并且每棵树之间无交集
- 并查集应用场景
  - 在一些应用问题中，需要将n个不同的元素划分成一些不相交的集合。
  - 开始时，每个元素自成一个单元素集合，然后按一定的规律将归于同一组元素的集合合并。在此过程中要反复用到查询某一个元素归属于哪个集合的运算。
- 并查集的数组表示
  - 数组的下标对应集合中元素的编号
  - 数组中如果为负数，负号代表根，数字代表该集合中元素个数
  - 数组中如果为非负数，代表该元素双亲在数组中的下标

通常用树（森林）的双亲表示作为并查集的存储结构，每个子集合以一棵树表示。所有表示子集合的树，构成表示全集合的森林，存放在双亲表示数组内。通常用数组元素的下标代表元素名，用根结点的下标代表子集合名，根结点的双亲结点为负数。

例如，若设有一个全集合为 $S=\{0,1,2,3,4,5,6,7,8,9\}$，初始化时每个元素自成一个单元素子集合，每个子集合的数组值为-1，如图 5.18 所示。

经过一段时间的计算，这些子集合合并为 3 个更大的子集合 $S_1=\{0,6,7,8\}$，$S_2=\{1,4,9\}$，$S_3=\{2,3,5\}$，此时并查集的树形表示和存储结构如图 5.19 所示。

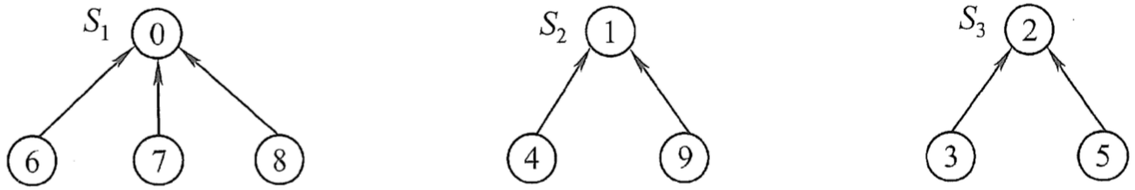


(a) 全集合S初始化时形成一个森林

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

(b) 初始化时形成的(森林)双亲表示

图 5.18　并查集的初始化



(a) 集合的树形表示

0	1	2	3	4	5	6	7	8	9
-4	-3	-3	2	1	2	0	0	0	1

(b) 集合S_1、S_2和S_3的(森林)双亲表示

图 5.19　用树表示并查集

并查集是一种简单的集合表示，它支持以下 3 种操作：

1）Union(S,Root1,Root2)：把集合 S 中的子集合 Root2 并入子集合 Root1。要求 Root1 和 Root2 互不相交，否则不执行合并。

2）Find(S,x)：查找集合 S 中单元素 x 所在的子集合，并返回该子集合的名字。

3）Initial(S)：将集合 S 中的每个元素都初始化为只有一个单元素的子集合。

并查集的结构定义如下：

```
#define SIZE 100
int     UFSets[SIZE];                    //集合元素数组（双亲指针数组）
```

并查集的初始化操作（S 即为并查集）：

```
void Initial(int S[]){
    for(int i=0;i<size;i++)              //每个自成单元素集合
        S[i]=-1;
}
```

Find 操作（函数在并查集 S 中查找并返回包含元素 x 的树的根）：

```
int Find(int S[],int x){
    while(S[x]>=0)                       //循环寻找 x 的根
        x=S[x];
    return  x;                           //根的 S[]小于 0
}
```

Union 操作（函数求两个不相交子集合的并集）：

```
void Union(int S[],int Root1,int Root2){
//要求 Root1 与 Root2 是不同的，且表示子集合的名字
    S[Root2]=Root1;                    //将根 Root2 连接到另一根 Root1 下面
}
```

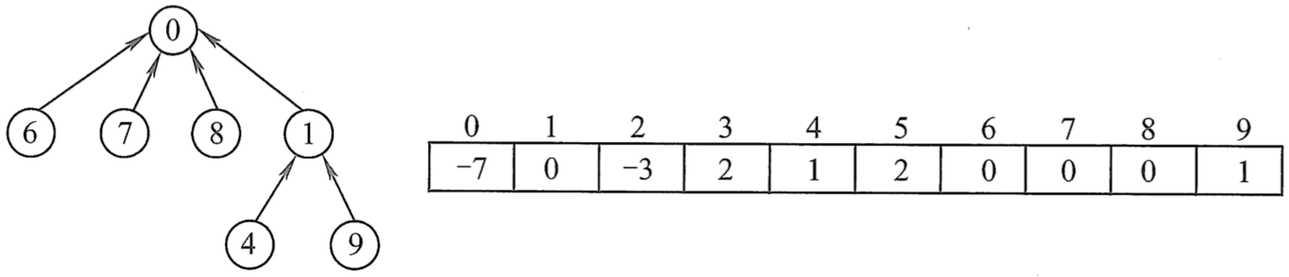


图 5.20　$S_1 \cup S_2$ 可能的表示方法

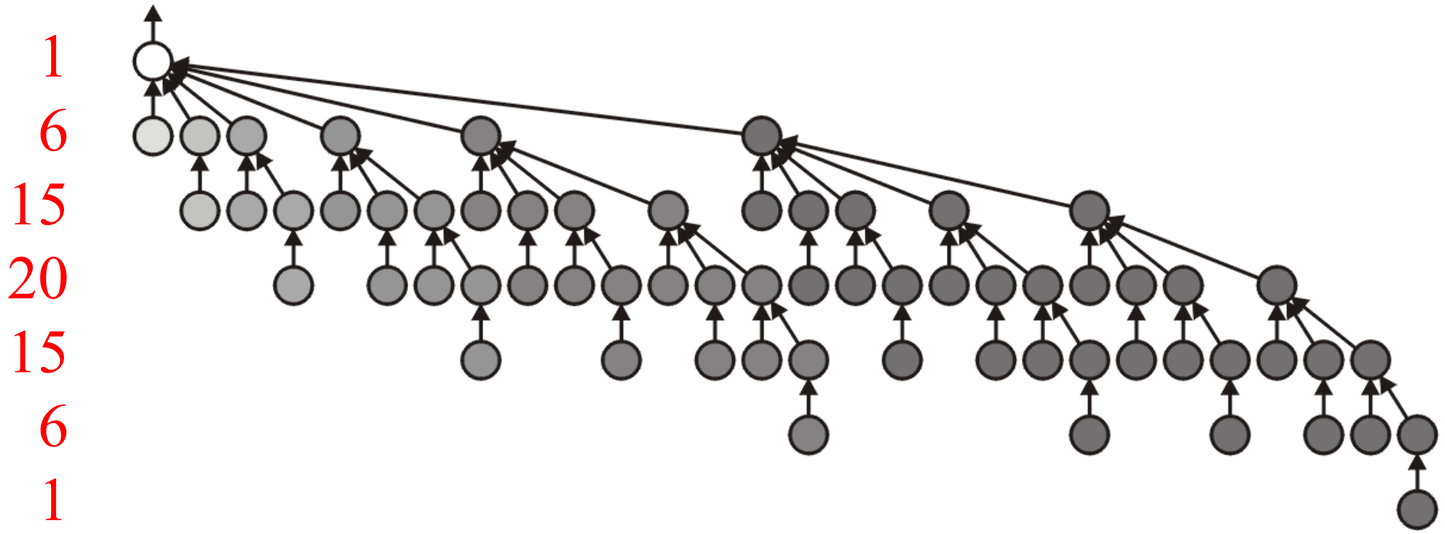
Worst-Case Scenario

- Let us consider creating the worst-case disjoint set
  - The tallest tree with the least number of nodes

- The worst case tree of height h must result from taking union of two worst case trees of height h–1

Worst-Case Scenario

These are *binomial trees*



Worst-Case Scenario

From the construction, it should be clear that this would define Pascal's triangle

- The *binomial* coefficients

$$\binom{n}{m} = \begin{cases} 1 & m = 0 \text{ or } m = n \\ \binom{n-1}{m} + \binom{n-1}{m-1} & 0 < m < n \end{cases}$$

$$= \frac{n!}{m!(n-m)!}$$

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

1 6 15 20 15 6 1

Worst-Case Scenario

- Thus, suppose we have a worst-case tree of height h
  - The number of nodes is $\sum_{k=0}^{h} \binom{h}{k} = 2^h = n$
  - The sum of node depth is $\sum_{k=0}^{h} k\binom{h}{k} = h2^{h-1}$
  - Therefore, the average depth is $\frac{h2^{h-1}}{2^h} = \frac{h}{2} = \frac{\lg(n)}{2}$
  - The height and average depth of the worst case are O(ln(n))

Optimization 1

- Problem:
  - The height of the tree may grow very large

- To optimize both find and set_union, we must minimize the height of the tree
  - Therefore, point the root of the shorter tree to the root of the taller tree
  - The height of the taller will increase if and only if the trees are equal in height

Union By Size

- Maintain a subtree count for each node, initially 1.
- Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

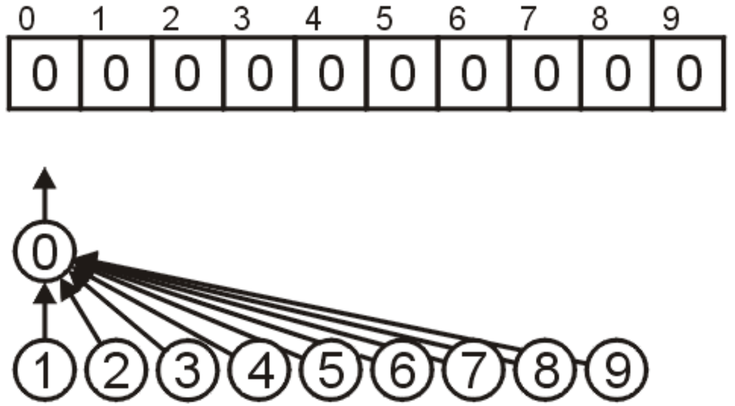
```
MAKE-SET (x)
parent(x) ← x.
size(x) ← 1.
```

```
FIND (x)
WHILE (x ≠ parent(x))
    x ← parent(x).
RETURN x.
```

```
UNION-BY-SIZE (x, y)
r ← FIND (x).
s ← FIND (y).
IF (r = s) RETURN.
ELSE IF (size(r) > size(s))
    parent(s) ← r.
    size(r) ← size(r) + size(s).
ELSE
    parent(r) ← s.
    size(s) ← size(r) + size(s).
```

Best-Case Scenario

- In the best case, all elements point to the same entry with a resulting height of O(1):



Optimization 2: Path Compression

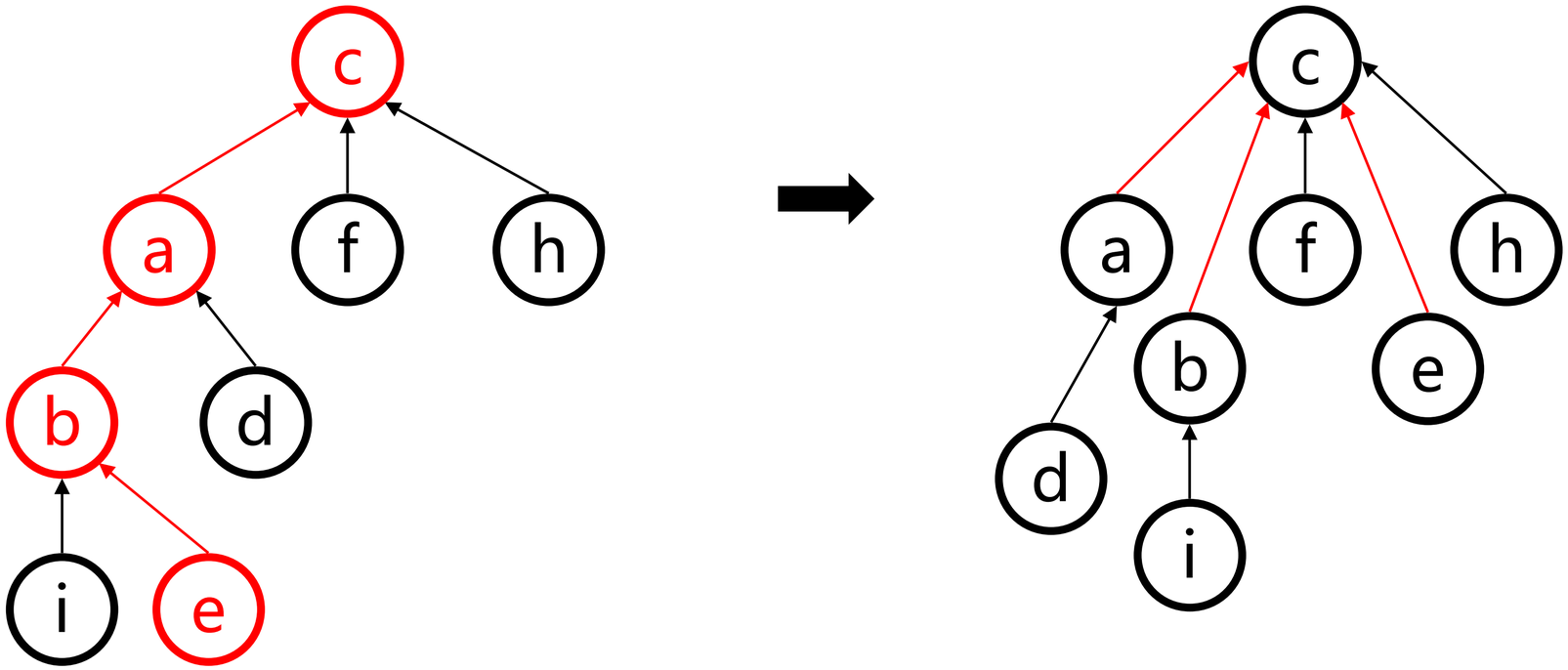
- Another optimization is that, whenever find is called, update the object to point to the root

```
size_t Disjoint_set::find( size_t n ) {
    if ( parent[n] == n ) {
        return n;
    } else {
        parent[n] = find( parent[n] );
        return parent[n];
    }
}
```

- The next call to find(n) is Q(1)
- The cost is O(h) memory

Optimization 2: Path Compression

- find(e)



Time complexity

- With both optimization methods, could it be any better than O(log(n))?
- Result in Θ(1) time complexity

Summary

- 查找元素属于哪个集合
  - 沿着数组表示树形关系以上一直找到根(即：树中中元素为负数的位置)
- 查看两个元素是否属于同一个集合
  - 沿着数组表示的树形关系往上一直找到树的根，如果根相同表明在同一个集合，否则不在
- 将两个集合归并成一个集合
  - (1)将两个集合中的元素合并 (2)将一个集合名称改成另一个集合的名称
- 集合的个数
  - 遍历数组，数组中元素为负数的个数即为集合的个数。