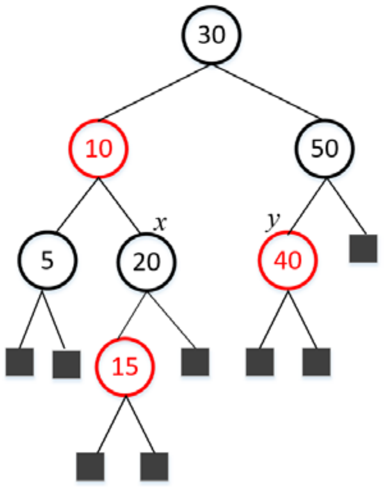
Csci 242. Algorithms and Data Structures **Fall, 2018**

Date: October 9th, 2018

Exam 1: Solution

Q1. [20] Data Structure

In the given tree,



(1). [5] Give (A) a **height** of a node *x* and (B) a **black-depth** of the node *y*.

Chap. 2.3 and Chap. 4.3: Height(x) = 2, Black-Depth(y) = 2

(2). [5] Suppose that the colors of the nodes of keys 10, 15 and 40 are also black, i.e. every node has the same color. Is the tree an AVL tree? Justify your answer.

Chap. 4.2

Yes, it's an AVL tree because the height of the left-subtree of the root(30) is 3 while that of the right-subtree is 2 whose difference is 1, while both its left-subtree and the right-subtree are also an AVL trees, recursively.

(3). [5] Traverse the tree by ***Preorder Traversal*** and print the keys.

Chap. 2.3: 30 10 5 20 15 50 40

(4). [5] Is the tree a ***binary maximum heap***? Justify your answer.

Chap. 5:

No, the key of root (30) is not the maximum key and each subtree of the root is not a maximum heap, either, etc.

Q2. [20] Short Answer. Explain your answer clearly.

(1). [5] What is a ***running time* of *deletion*** and that of ***insertion*** operation of ***max-priority queue*** of *n* elements implemented by a ***heap,*** respectively? Give them in big-Oh(O) notation.

Chap. 5.3

Both the running time of deletion and that of insertion is O(log *n*)

(2). [5] What is the ***maximum* number of *internal nodes*** of a binary ***heap*** of height *h,* that stores *n* elements? Explain your answer by drawing such a heap.

Chap. 5.3, Theorem 5.1 – Case 2

When the binary tree is complete: every internal nodes has two children.

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$

(3). [5] Give a **definition** of a ***Red-Black tree*** and describe its **property.**

Chap. 4.3

A red-black tree is a height balanced binary search tree that satisfies the following property:

1. Root property: the root is black.
2. External property: every leaf is black.
3. Internal property: the children of a red node are black.
4. Black-depth property: all the leaves have the same black depth, i.e. the same # of black nodes as proper ancestors.

(4). [5] Explain (A) an ***open addressing*** is in the hashing and (B) give *three methods* for open addressing.

Chap. 6.3

Open addressing is the mechanism of collision handling which places a colliding item in a different cell of the hash table, rather using the extra memory space for the linked list or another hash table to store the colliding keys.

Linear probing, Quadratic probing and Double hashing are the methods for open addressing.

Q3. [10] Write a ***recursive*** algorithm, **Tree-Sum(*v*),** that returns the sum of the ***keys of internal nodes*** in the tree rooted at a node *v*. A tree is implemented as a *linked list structure* and all of the external nodes are dummy with no key.

```
Tree-Sum (v)
Input: a node v
Output: the sum of the keys of internal nodes

if v = nil then return 0
return v.key + Tree-Sum(v.left) + Tree-Sum(v.right)
```

Q4. [10] (A) Write an insertion algorithm **put(*k*)** which **inserts** a key *k* to the hash table A of size N using the hash function *h* and handling a collision with **linear probing**. (B) Then, insert a key 17 to the given hash table of size 11

0	1	2	3	4	5	6	7	8	9	10
	23			15		28	7	39	18	

.

Chap. 6.3

- put(k, v):

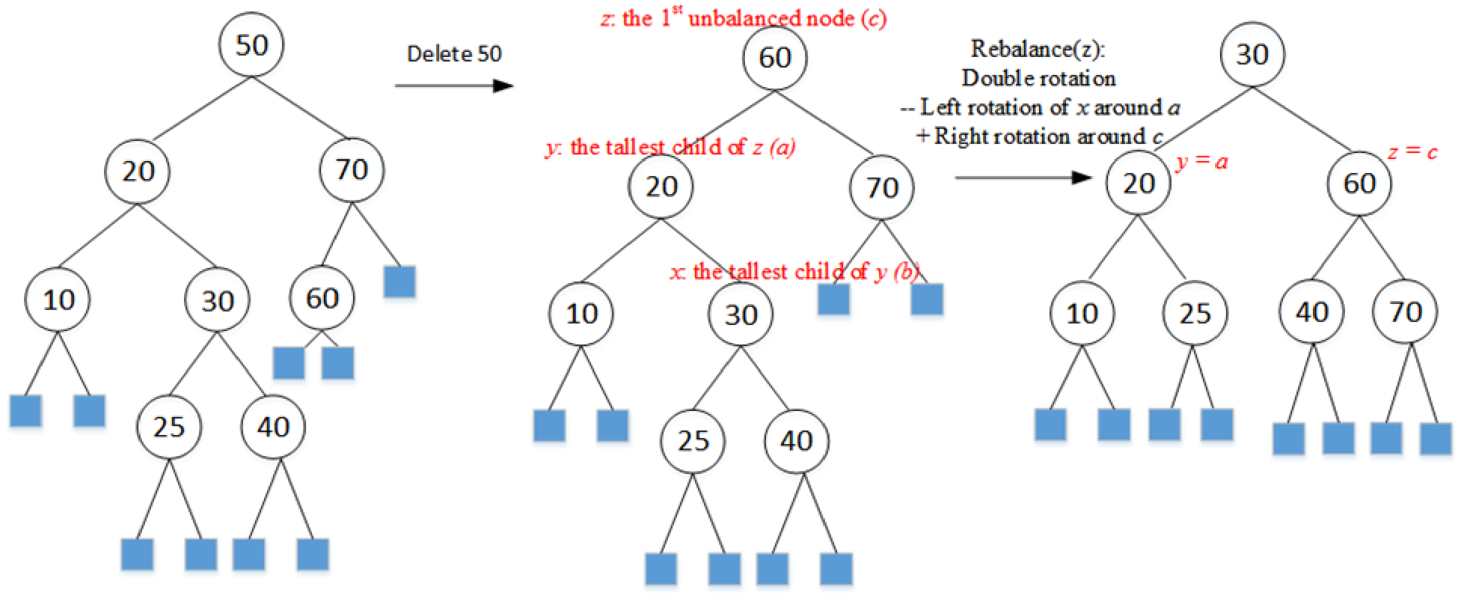
```
    i ← h(k)
    while A[i] ≠ NULL do
        if A[i].key = k then
            A[i] ← (k, v)        // replace the old (k, v')
        i ← (i + 1) mod N
    A[i] ← (k, v)
```

h(17) = 17 mod 11 = 6. Collision at A[6], A[7], A[8], A[9], thus put it at A[10]

Q5. [10] Draw the ***AVL tree*** after **deleting** the key **50** from the following AVL tree.
Show each step of deletion & rotation(s) to restore the AVL property for the final tree.

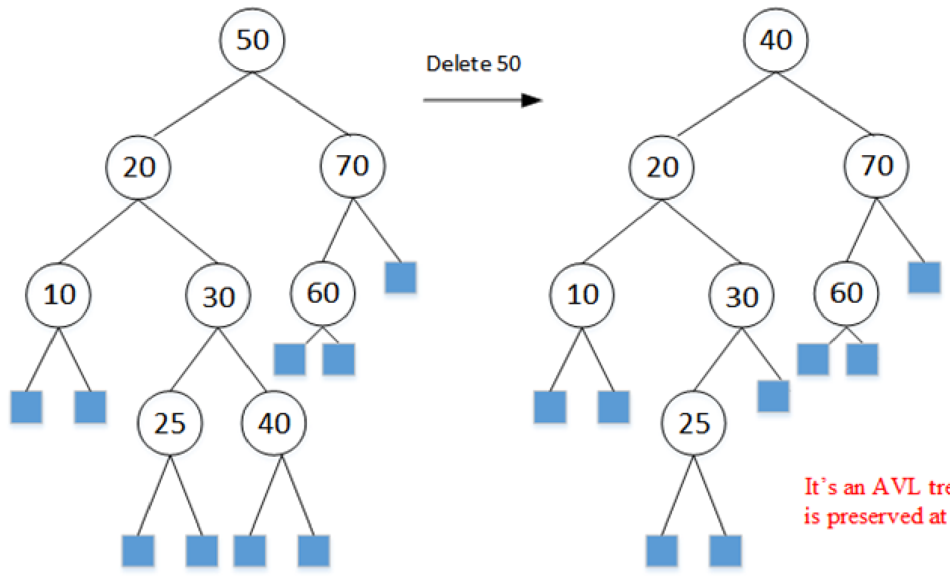
Chap. 4.2

After deleting 50, it's replaced by its immediate successor 60, actually deleting a node of 60.


50
20
70
10
30
60
25
40
Delete 50
z: the 1st unbalanced node (c)
60
y: the tallest child of z (a)
20
70
10
30
x: the tallest child of y (b)
25
40
Rebalance(z):
Double rotation
-- Left rotation of x around a
+ Right rotation around c
30
20
y = a
60
z = c
10
25
40
70

a : a node with the smallest key
b : a node with the middle key
c : a node with the largest key
among three nodes *x*, *y*, *z*.

After deleting 50, it's replaced by its immediate predecessor 40, actually deleting a node of 40.



It's an AVL tree after deletion since AVL property is preserved at every node.

Q6. [10] Write a ***recursive algorithm*** that computes $\sum_{i=0}^{n} x^i$ in $\mathbf{O}(n)$ time.

Chap. 6.2. The same idea from the polynomial accumulation using Horner's rule in the hash code.

$\sum_{i=0}^{n} x^i = 1 + x + x^2 + x^3 + \ldots + x^n$

$= 1 + (1 + x + x^2 + x^3 + \ldots + x^{n-1})\, x$

$= 1 + (1 + (1 + x + x^2 + x^3 + \ldots + x^{n-2}) \cdot x) \cdot x$

$= 1 + (1 + (1 + (1 + x + x^2 + x^3 + \ldots + x^{n-3}) \cdot x) \cdot x) \cdot x$

....

$= 1 + (1 + (1 + (1 + \ldots (1 + x) \cdot x \ldots . \cdot x) \cdot x) \cdot x) \cdot x$

i.e. $P_n(x) = \sum_{i=0}^{n} x^i$

where $P_0(x) = 1$ for $i = 0$

$P_i(x) = 1 + P_{i-1}(x) \cdot x$ for $i = 1, 2, \ldots, n-1$

So, the recursive algorithm is

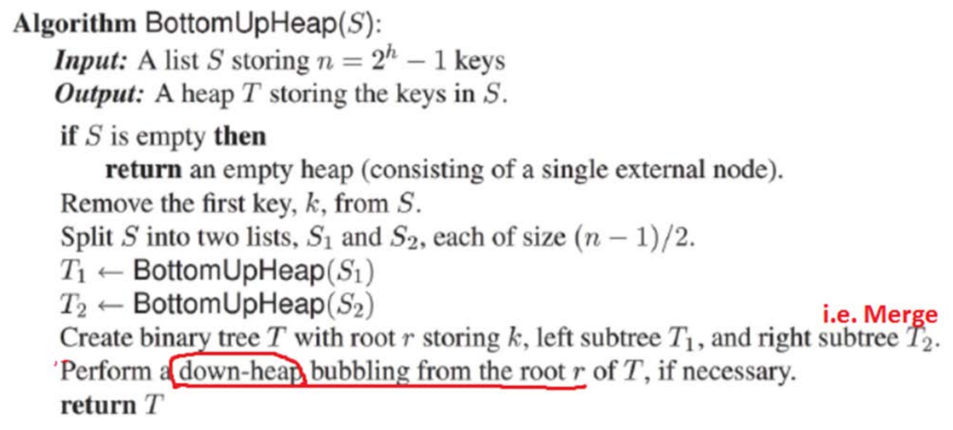
```
G-Sum(x, n):
Input:  (the fixed value of x and) the highest order n of the geometric series
Output: the sum of a geometric series, $\sum_{i=0}^{n} x^i$
If n=0 return 1
return 1 + x·G-Sum(x, n-1)
```

Q7. [20] Min-Heap Chap. 5.4: slide#45 - #53

In the given array A[1.. n] = [15, 9, 7, 12, 5, 6, 3],

(1) [10] A heap can be constructed by merging two heaps. Write an in-place **recursive algorithm BottomUpHeap(A)** that constructs a **min-heap** in the array A[1 .. n] by merging two sub-heaps with a new key. Suppose that the number of stored keys $n=2^h-1$ where h is the height of a heap.

Chap. 5.4



```
Algorithm BottomUpHeap($S$):
    Input: A list $S$ storing $n = 2^h - 1$ keys
    Output: A heap $T$ storing the keys in $S$.
    if $S$ is empty then
        return an empty heap (consisting of a single external node).
    Remove the first key, $k$, from $S$.
    Split $S$ into two lists, $S_1$ and $S_2$, each of size $(n-1)/2$.
    $T_1 \leftarrow$ BottomUpHeap($S_1$)
    $T_2 \leftarrow$ BottomUpHeap($S_2$)
                                                        i.e. Merge
    Create binary tree $T$ with root $r$ storing $k$, left subtree $T_1$, and right subtree $T_2$.
    Perform a down-heap bubbling from the root $r$ of $T$, if necessary.
    return $T$
```

This algorithm of the textbook has to be modified to handle the indices for an in-place algorithm in A: Construct a heap from the element at A[i], where i is the index of the root.

I)

```
Algorithm BottomUpHeap(A, $i$ $j$)  // construct a heap in A[i .. $j$]
If $i \geq j$ then return $i$ (or A[i])   // $i$ is the index of a leaf, which is a heap of single element, so return
                                          the index $i$, i.e. the root of the single heap.
~~(Key ← A[i] )~~                        // T1 & T2 will be merged with a new key of root.
T1 ← BottomUpHeap(A, $i+1, \lceil (j+1)/2 \rceil$) // construct a heap for the left-subtree of the root.
T2 ← BottomUpHeap(A, $\lceil (j+1)/2 \rceil +1, j$)// construct a heap for the right-subtree.
~~root ← i~~                             // T1 & T2 are merged with a new root at A[i].
$n \leftarrow j$
DownHeap(A, ~~root~~ $i$, $j$ )          // restore the min-heap property from the root $i$.
return ~~root~~ $i$ (or A)               // return the index $i$, i.e. the root of the

Algorithm DownHeap(A, root, $n$)
                          // return the index of the root after restoring the heap property
```

The DownHeap algorithm has to be further modified to handle the indices properly.

II)

```
Algorithm BottomUpHeap(A, i):
    If $i > \lfloor \frac{n}{2} \rfloor$ then return i (or A)   // i is the index of a leaf, which is a heap of single element, so return
                                                                 the index i, i.e. the root of the single heap.
    ~~{Key ← A[i] }~~                      // T1 & T2 will be merged with a new key of root.
    T1 ← BottomUpHeap(A, 2i)               // construct a heap for the left-subtree of the root.
    T2 ← BottomUpHeap(A, 2i+1)             // construct a heap for the right-subtree of the root.
    ~~root ← i~~                           // T1 & T2 are merged with a new root at A[i].
    DownHeap(A, ~~root~~ i )               // restore the min-heap property from the root
    return ~~root~~ i (or A)               // return the index i, i.e. the root of the
```

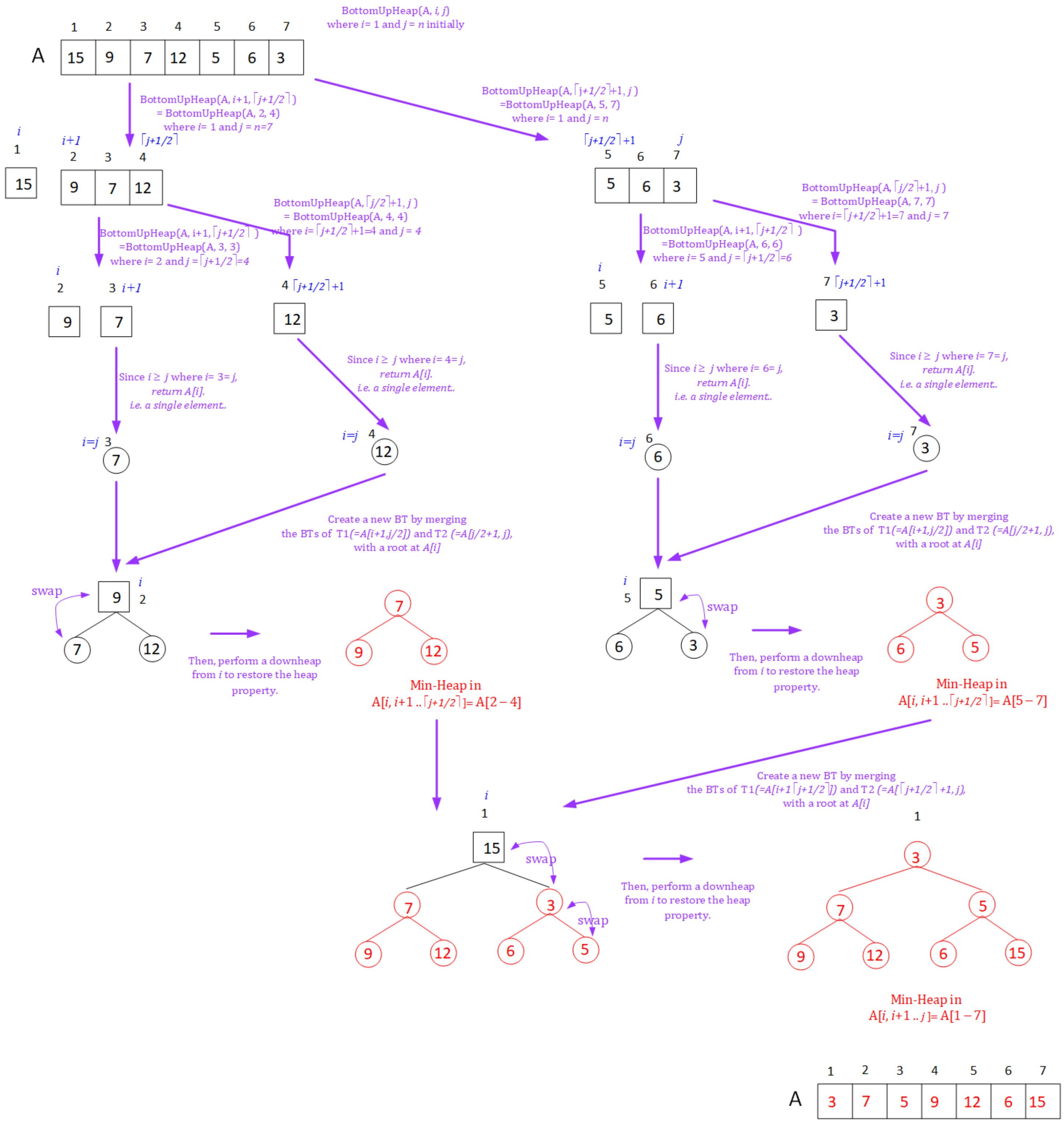
```
Algorithm DownHeap(A, root)      // return the index of the root after restoring the heap property
                                 // from the root
```

DownHeap

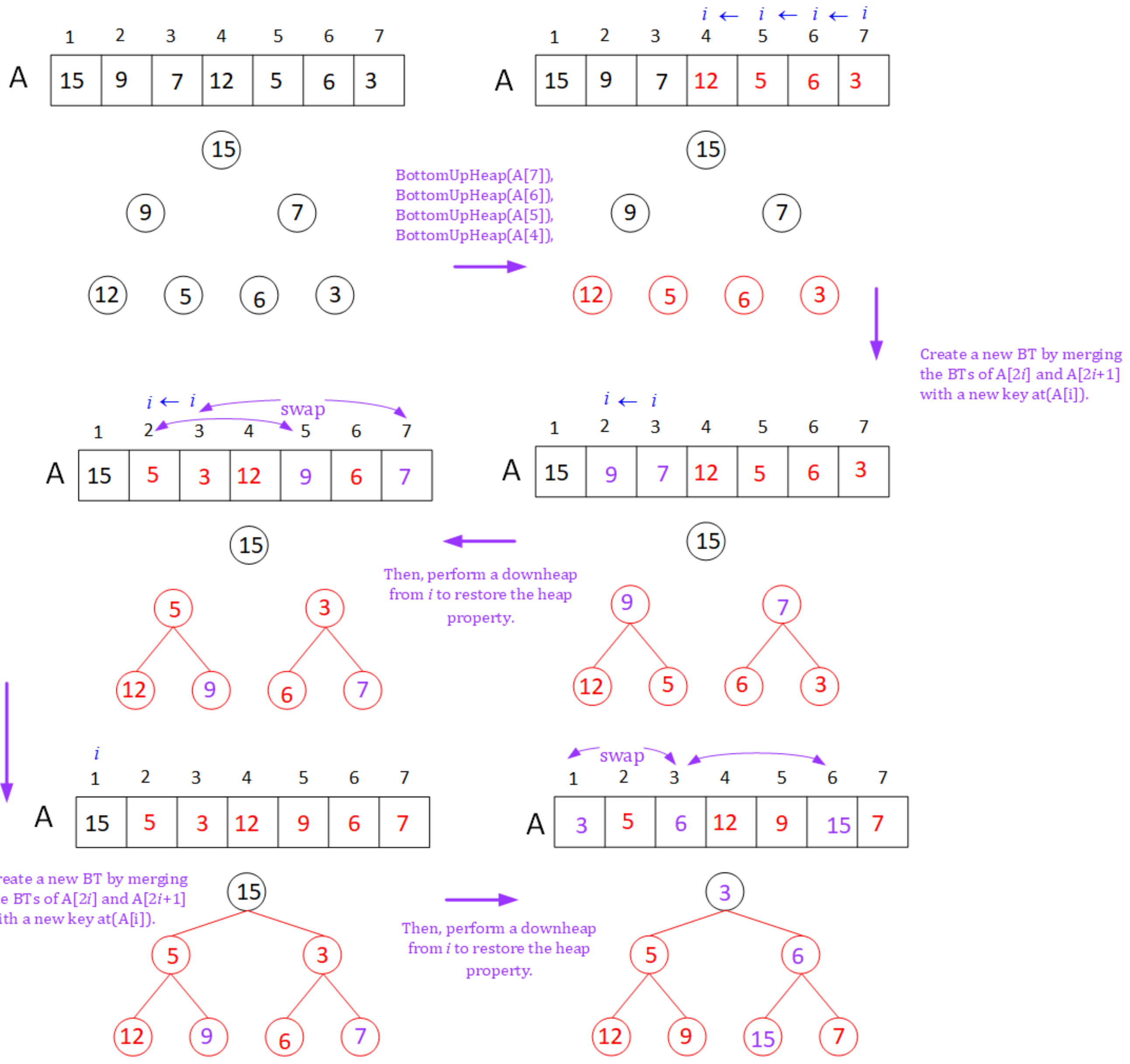
```
$i \leftarrow$ root
while $i < n$ do
    if $2i + 1 \leq n$ then   // this node has two internal children
        if $A[i] \leq A[2i]$ and $A[i] \leq A[2i+1]$ then      // if key(parent) < keys(both children)
            return root      // we have restored the heap-order property
        else
            Let $j$ be the index of the smaller of $A[2i]$ and $A[2i+1]$   // j = a child of the smaller key
            Swap $A[i]$ and $A[j]$
            $i \leftarrow j$                                            // along the downward path
    else   // this node has zero or one internal child
        if $2i \leq n$ then   // this node has one internal child (the last node)
            if $A[i] > A[2i]$ then
                Swap $A[i]$ and $A[2i]$
        return root   // we have restored the heap-order property
return root   // we reached the last node or an external node
```

(2) [10] Construct a min-heap in the given array A by BottomUpHeap. Show the **final min-heap** in the **array A**.

I) By Algorithm in I), as it is in the example of the **slide**:


BottomUpHeap(A, i, j)
where i= 1 and j = n initially
A: 15 9 7 12 5 6 3
BottomUpHeap(A, i+1, ⌈j+1/2⌉) = BottomUpHeap(A, 2, 4) where i= 1 and j = n=7
BottomUpHeap(A, ⌈j+1/2⌉+1, j) =BottomUpHeap(A, 5, 7) where i= 1 and j = n
BottomUpHeap(A, i+1, ⌈j+1/2⌉) =BottomUpHeap(A, 3, 3) where i= 2 and j = ⌈j+1/2⌉=4
BottomUpHeap(A, ⌈j/2⌉+1, j) = BottomUpHeap(A, 4, 4) where i= ⌈j+1/2⌉+1=4 and j = 4
BottomUpHeap(A, i+1, ⌈j+1/2⌉) =BottomUpHeap(A, 6, 6) where i= 5 and j = ⌈j+1/2⌉=6
BottomUpHeap(A, ⌈j/2⌉+1, j) = BottomUpHeap(A, 7, 7) where i=⌈j+1/2⌉+1=7 and j = 7
Since i ≥ j where i= 3= j, return A[i]. i.e. a single element.
Since i ≥ j where i= 4= j, return A[i]. i.e. a single element.
Since i ≥ j where i= 6= j, return A[i]. i.e. a single element.
Since i ≥ j where i= 7= j, return A[i]. i.e. a single element.
Create a new BT by merging the BTs of T1(=A[i+1,j/2]) and T2 (=A[j/2+1, j]), with a root at A[i]
swap
Then, perform a downheap from i to restore the heap property.
Min-Heap in A[i, i+1 ..⌈j+1/2⌉]= A[2 – 4]
Min-Heap in A[i, i+1 ..⌈j+1/2⌉]= A[5 – 7]
Create a new BT by merging the BTs of T1(=A[i+1⌈j+1/2⌉]) and T2 (=A[⌈j+1/2⌉+1, j]), with a root at A[i]
Min-Heap in A[i, i+1 .. j]= A[1 – 7]
A: 3 7 5 9 12 6 15

II) Or, By Algorithm in II),



BottomUpHeap(A[7]),
BottomUpHeap(A[6]),
BottomUpHeap(A[5]),
BottomUpHeap(A[4]),

Create a new BT by merging the BTs of A[2*i*] and A[2*i*+1] with a new key at(A[i]).

Then, perform a downheap from *i* to restore the heap property.

Create a new BT by merging the BTs of A[2*i*] and A[2*i*+1] with a new key at(A[i]).

Then, perform a downheap from *i* to restore the heap property.

(3) [10, optional] What is the ***running time*** of **BottomUpHeap** in its asymptotic ***upper bound,*** big-Oh (O)? Write its recurrence equation of T(n) and solve it.

Chap. 5.4

$$T(n) = \begin{cases} O(1) & n = 0 \\ 2T\left(\frac{n}{2}\right) + O(\log n) & \text{if } n > 0 \end{cases} \quad \Rightarrow \text{Solution: } T(n) = O(n \log n)$$

$$T(n) = 2T\left(\frac{n}{2}\right) + c\cdot\log n = 2\left(2T\left(\frac{n}{2^2}\right) + c\cdot\log\frac{n}{2}\right) + c\log n$$

$$= 2^2T\left(\frac{n}{2^2}\right) + c\cdot\left(\log\frac{n}{2} + \log n\right) = 2^2(2T\left(\frac{n}{2^3}\right) + c\cdot\log\frac{n}{2^2}) + c\cdot\left(\log\frac{n}{2} + \log n\right)$$

$$= 2^3T\left(\frac{n}{2^3}\right) + c\cdot\left(\log\frac{n}{2^2} + \log\frac{n}{2} + \log n\right) = \ldots\ldots$$

$$= 2^hT\left(\frac{n}{2^h}\right) + c\cdot\left(\log\frac{n}{2^{h-1}} + \cdots + \log\frac{n}{2^2} + \log\frac{n}{2} + \log n\right) = \ldots.$$

$$= n\cdot T(1) + c\cdot\left(\log\frac{n}{2^{\log n-1}} + \cdots + \log\frac{n}{2^2} + \log\frac{n}{2} + \log n\right) \text{ until } \frac{n}{2^h} = 1 \Leftrightarrow h = \log_2 n$$

$$\leq n\cdot T(1) + c\cdot n\log n = O(n) + O(n\log n) = O(n \log n)$$

Q8. [10, optional] Analysis of Algorithm

For an algorithm whose total running time is T(n) = n^2+ 2n – 8, (A) give its ***asymptotic upper bound*** in ***big-Oh*** notation (**O**) and (B) prove it by the definition of big-Oh.

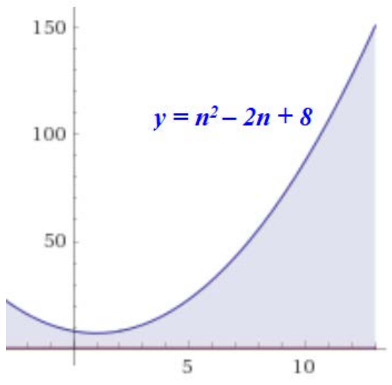
From the previous exam, HW

Let's show that $T(n) = n^2 + 2n - 8 = O(n^2)$.

Then, we have to $T(n) = n^2 + 2n - 8 \leq c\cdot g(n)$ for $n \geq n_0$ for a positive constant c and n_0.

where $g(n) = n^2$

i.e. $n^2 + 2n - 8 \leq c\cdot n^2$



Choose $c = 2$. Then, $n^2 + 2n - 8 \leq 2n^2 \Leftrightarrow 0 \leq n^2 - 2n + 8$.

So, for any $n \geq 1$, $\Leftrightarrow n^2 + 2n - 8 \leq 2n^2$.

Thus, $n^2 + 2n - 8 \leq c\cdot n^2$ when $c = 2$ and $n \geq n_0 = 1$.

Therefore, $T(n) = n^2 + 2n - 8 = O(n^2)$.