

CS3334 - Data Structures

Lab 4

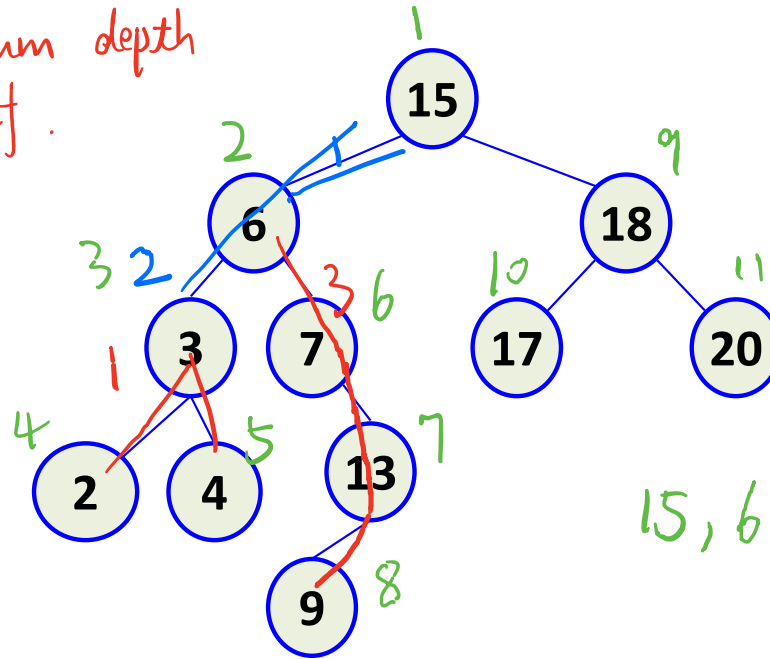
Outline

- Tree Exercises
- Assignment 757: Huffman Code
- Assignment 756: Max-Heap

Exercise 1 for Tree

Height: The maximum depth of any leaf.

Depth: from root



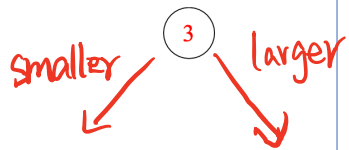
15, 6, 3, 2, 4, 7, 13, 9, 18, 17, 20

1. What's the height of node 6? 3
2. What's the height of node 3? 1
3. What is the depth of node 6? 1
4. What's the depth of node 3? 2
5. What's the preorder traversal of the tree?

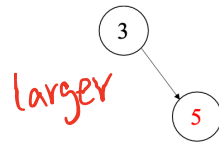
Exercise 2 for Tree

- Create a **binary search tree** according to the input insertion sequence:
{3, 5, 0, 2, 7, 6, 8, 9, 4, 1}.

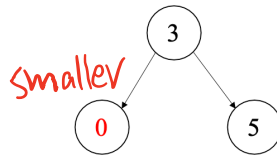
- right (larger)
- left (smaller)



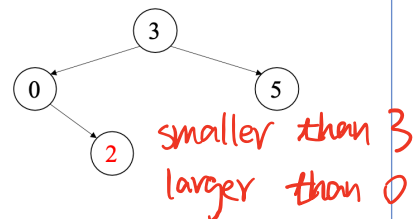
Step 1: input sequence {3,5,0,2,7,6,8,9,4,1}



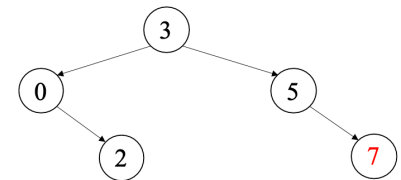
Step 2: input sequence {5,0,2,7,6,8,9,4,1}



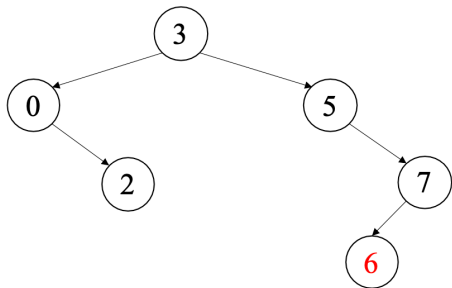
Step 3: input sequence {0,2,7,6,8,9,4,1}



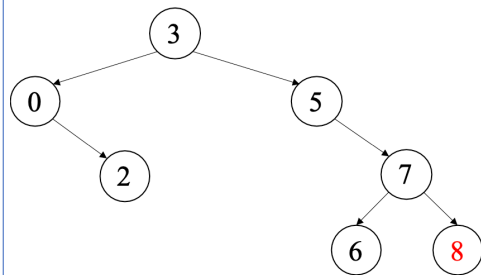
Step 4: input sequence {2,7,6,8,9,4,1}



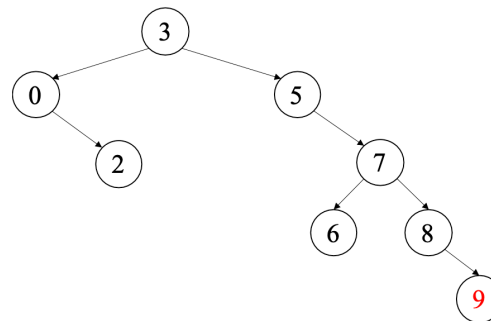
Step 5: input sequence {7,6,8,9,4,1}



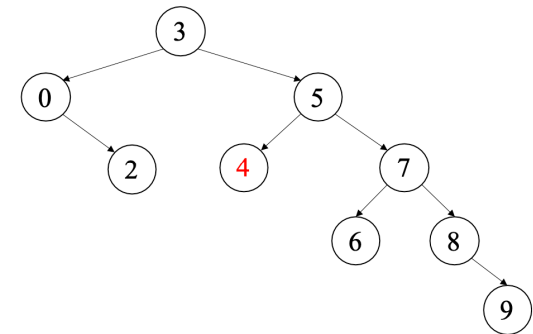
Step 6: input sequence {6,8,9,4,1}



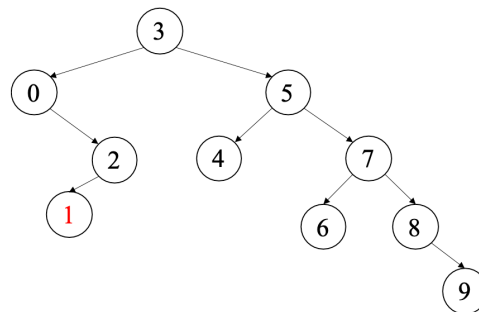
Step 7: input sequence {8,9,4,1}



Step 8: input sequence {9,4,1}



Step 9: input sequence {4,1}



Step 10: input sequence {1}

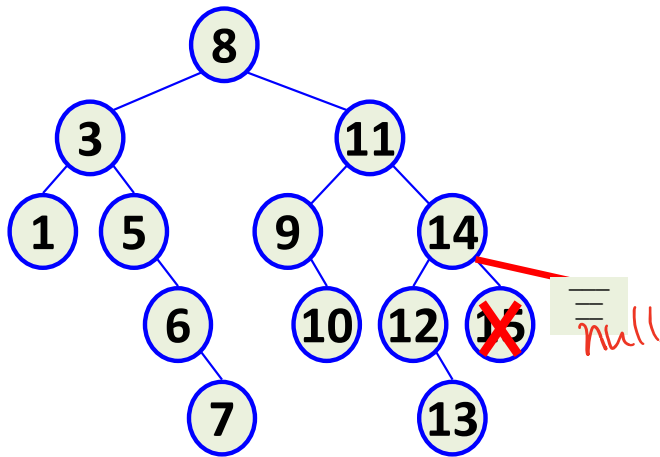
Binary Search Tree - Deletion

Delete a node in a BST:

void Mytree::delete(TreeNode* node)

Case 1:

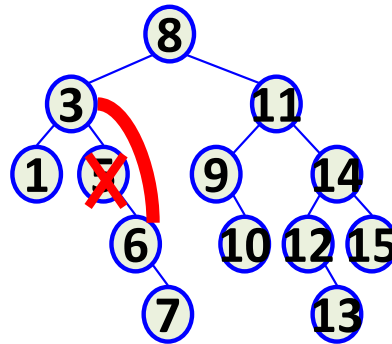
To delete a leaf node



- Set the parent node's child pointer to NULL

Case 2:

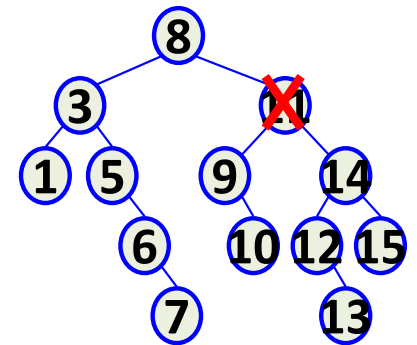
To delete a node that has 1 subtree (left or right)



- Set the parent's child pointer to root of unwanted node's subtree (left or right).

Case 3:

To delete a node that has 2 subtrees



More complicated

Binary Search Tree - Deletion

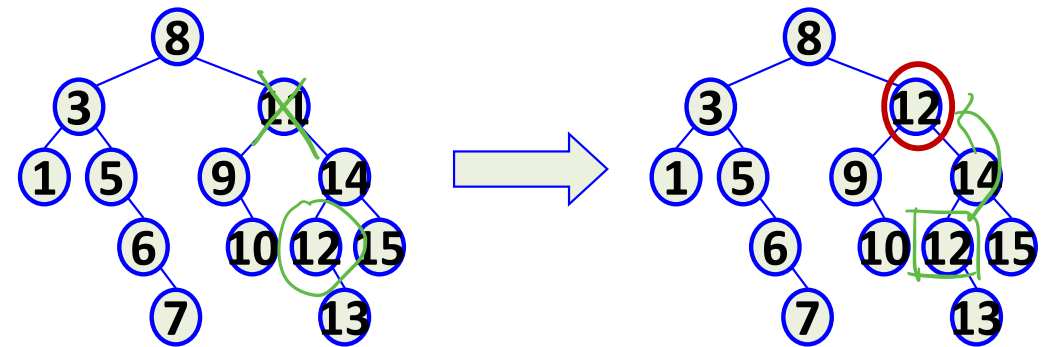
Delete a node in a BST: `void Mytree::delete(TreeNode* node)`

Case 3:

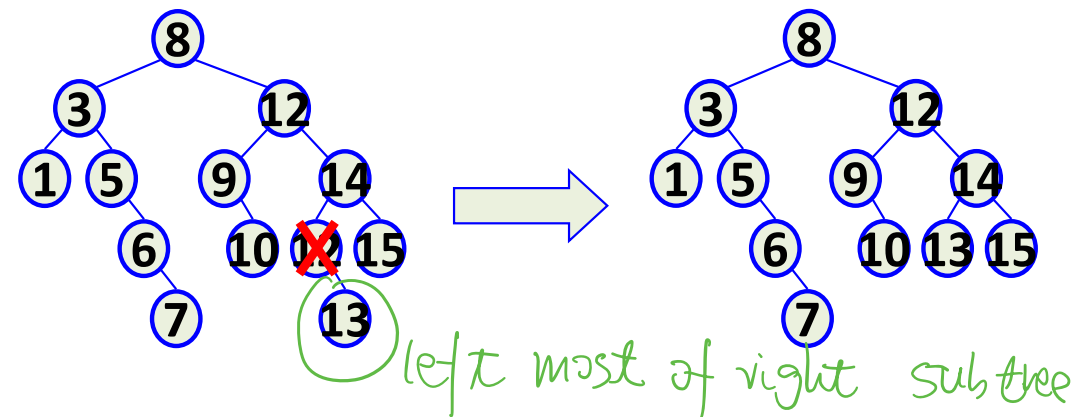
To delete a node that has 2 subtrees

- Replace its value by its inorder successor: (the least larger value)

The inorder successor is the leftmost node of right subtree. ★

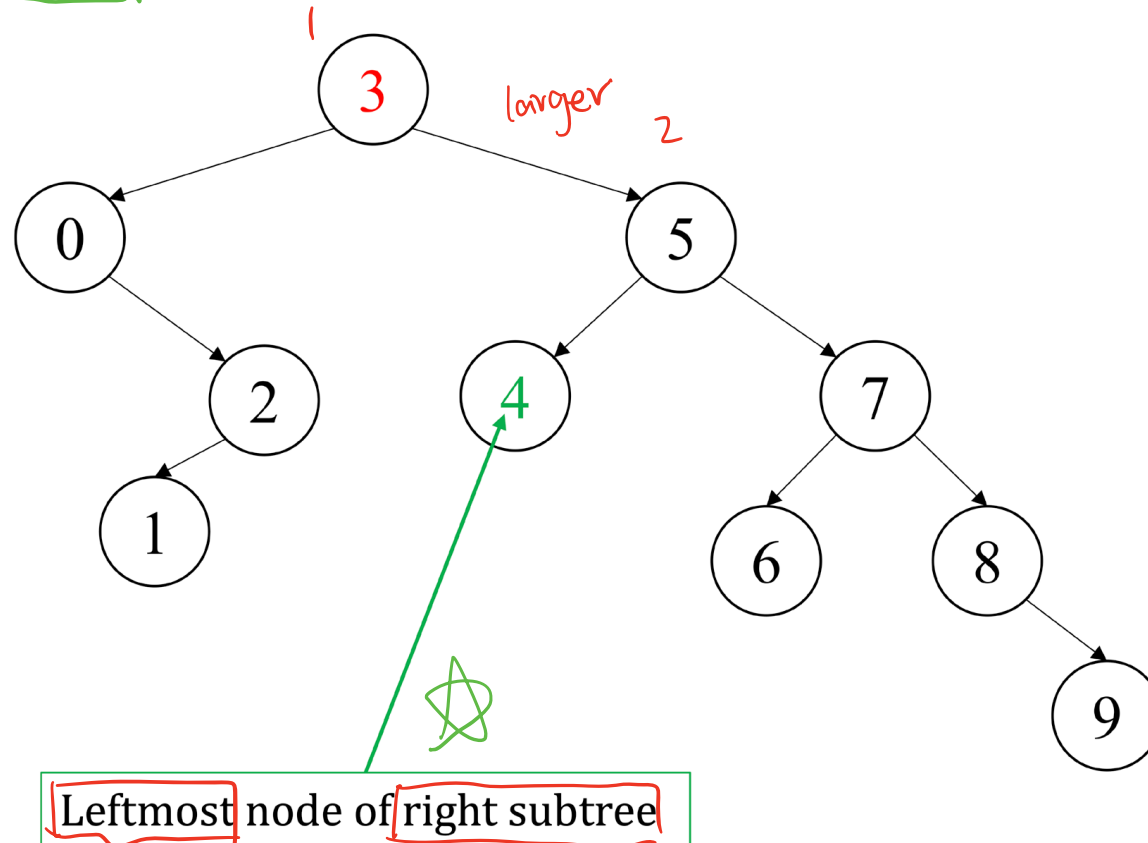


- Delete the successor in turn:



Exercise 2 for Tree

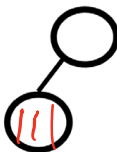
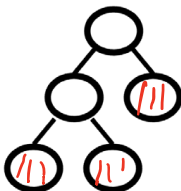
- Create a binary search tree according to the input sequence: {3, 5, 0, 2, 7, 6, 8, 9, 4, 1}.
- Then delete 3 from the constructed binary search tree.

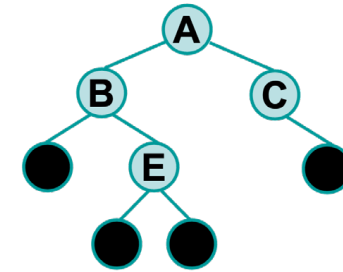


Exercise 3 for Tree

- What's the running time for the function "countleaf"?

Example:

A NULL binary tree has 0 leaf node	 A tree with 1 node has 1 leaf node	 No. of leaf nodes = 1	 No. of leaf nodes = 3
---	--	--	---



```
//To count the number of leaf nodes
int Mytree::count_leaf(TreeNode* p)
{
    if (p == NULL)
        return(0);
    else if ((p->left == NULL) && (p->right == NULL))
        return(1);
    else
        return(count_leaf(p->left) + count_leaf(p->right));
}
```

$$T(n) = 1 + 2T\left(\frac{n}{2}\right)$$

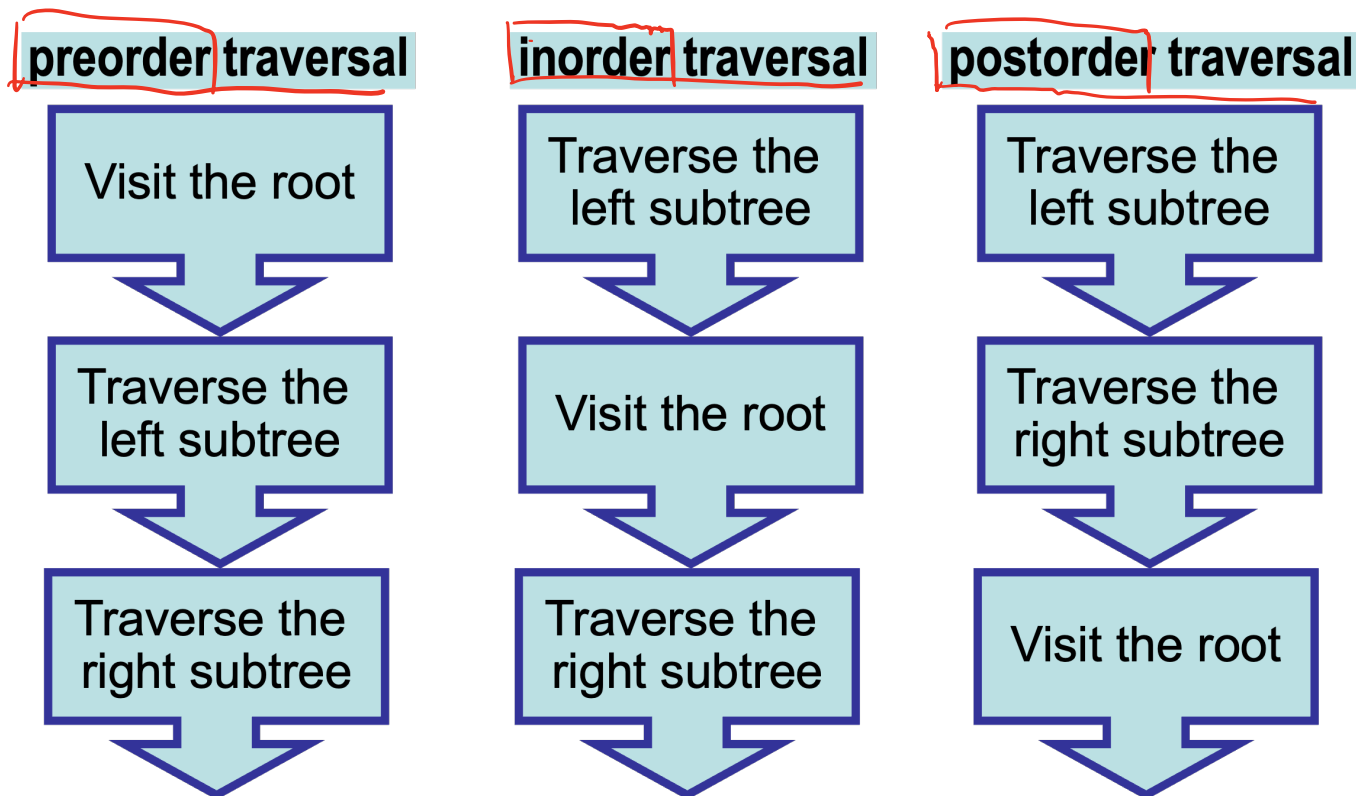
$$T(n) = O(n)$$

(visit each node once)

inclusive

Exercise 4 for Tree

- Reconstruct the binary tree whose postorder and inorder are as follows.
 - Postorder: BDFECA
 - Inorder: BACDFE

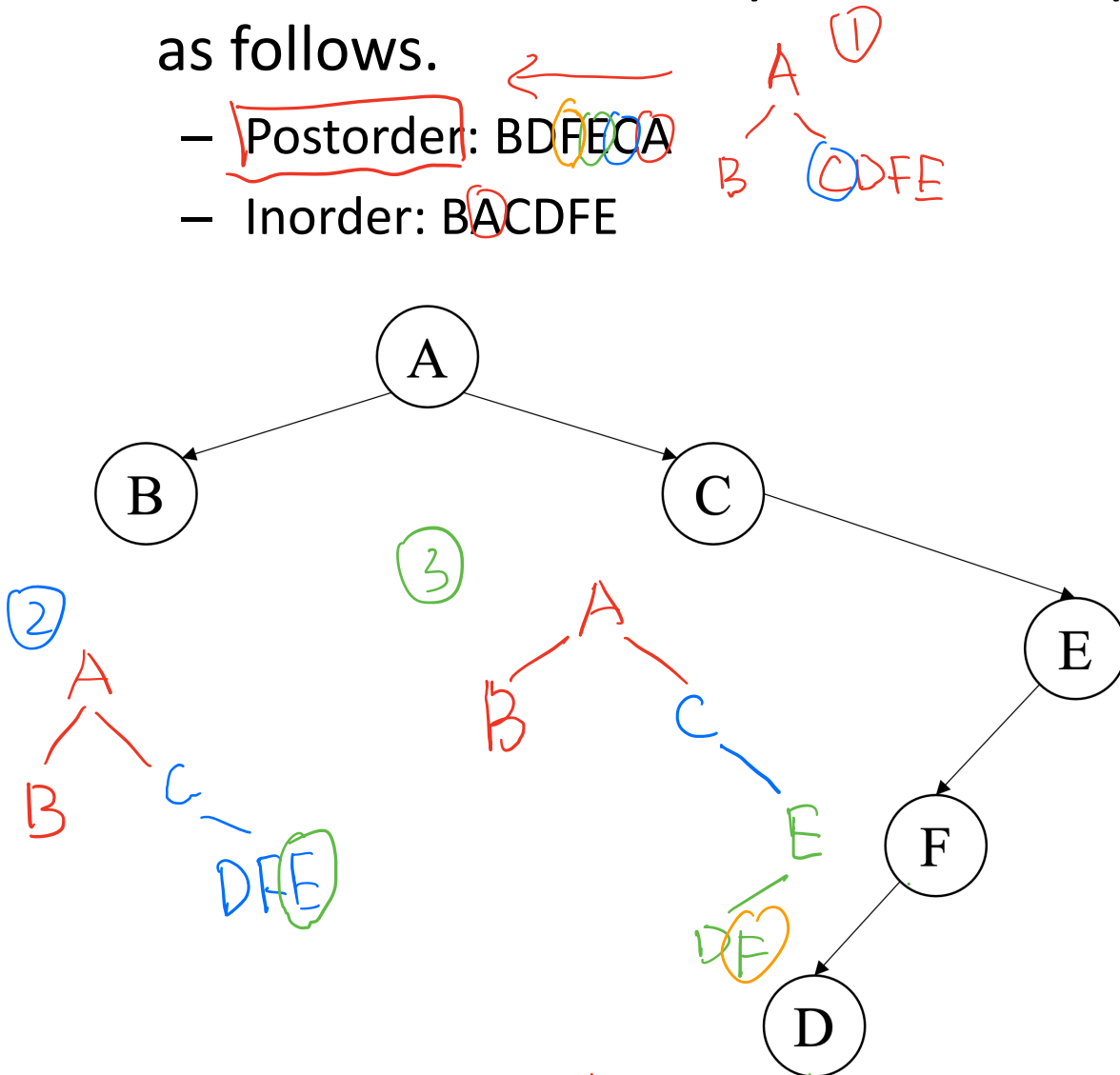


Exercise 4 for Tree

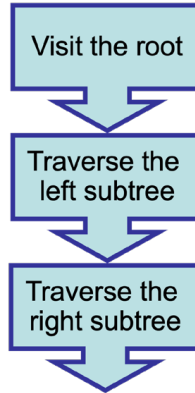
- Reconstruct the binary tree whose postorder and inorder are as follows.

– Postorder: BDFECA

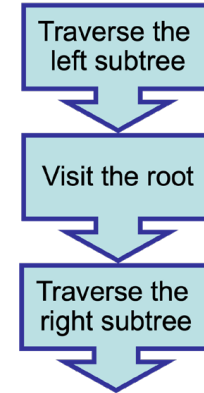
– Inorder: BACDFE



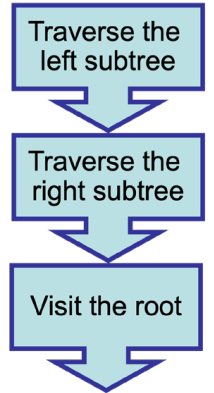
preorder traversal



inorder traversal



postorder traversal

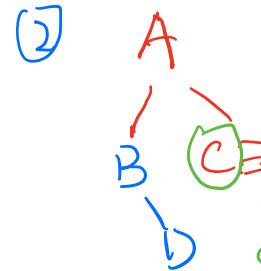
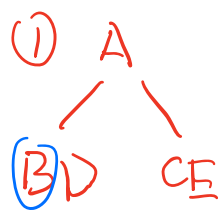


Exercise 5 for Tree

- Showing the postorder traversal of the binary tree with **preorder** and **inorder** shown as follows.

– **Preorder**: ABCDE

– **Inorder**: BDACE



(3)

⇒ No such a tree

ABC X

ABD... ✓

D should be before
E in preorder.

preorder traversal

Visit the root

Traverse the
left subtree

Traverse the
right subtree

inorder traversal

Traverse the
left subtree

Visit the root

Traverse the
right subtree

postorder traversal

Traverse the
left subtree

Traverse the
right subtree

Visit the root

Exercise 6 for Tree

- find the level of Node "ptr" in a binary tree with root node "root". Using the recursive function
- The length of the unique path from root to this node.*

```
struct Node {  
    int data;  
    struct Node *left, *right;  
};  
  
int level (Node* root, Node* ptr, int lev) {  
    ... ..  
}
```

For recursive func

Exercise 6 for Tree

- find the level of Node “ptr” in a binary tree with root node “root”. Using the recursive function.

```
struct Node {
    int data;
    struct Node *left, *right;
};

int level (Node* root, Node* ptr, int lev) {
    // base cases
    if (root == NULL)
        return 0;
    if (root == ptr)
        return lev;

    // Return level if Node is present in left subtree
    int l = level(root->left, ptr, lev + 1);
    if (l != 0)
        return l;

    // Else search in right subtree
    return level(root->right, ptr, lev + 1);
}
```

zero if not find

current visiting node

target

current level of root

not find in left subtree

757: Huffman Coding

Description

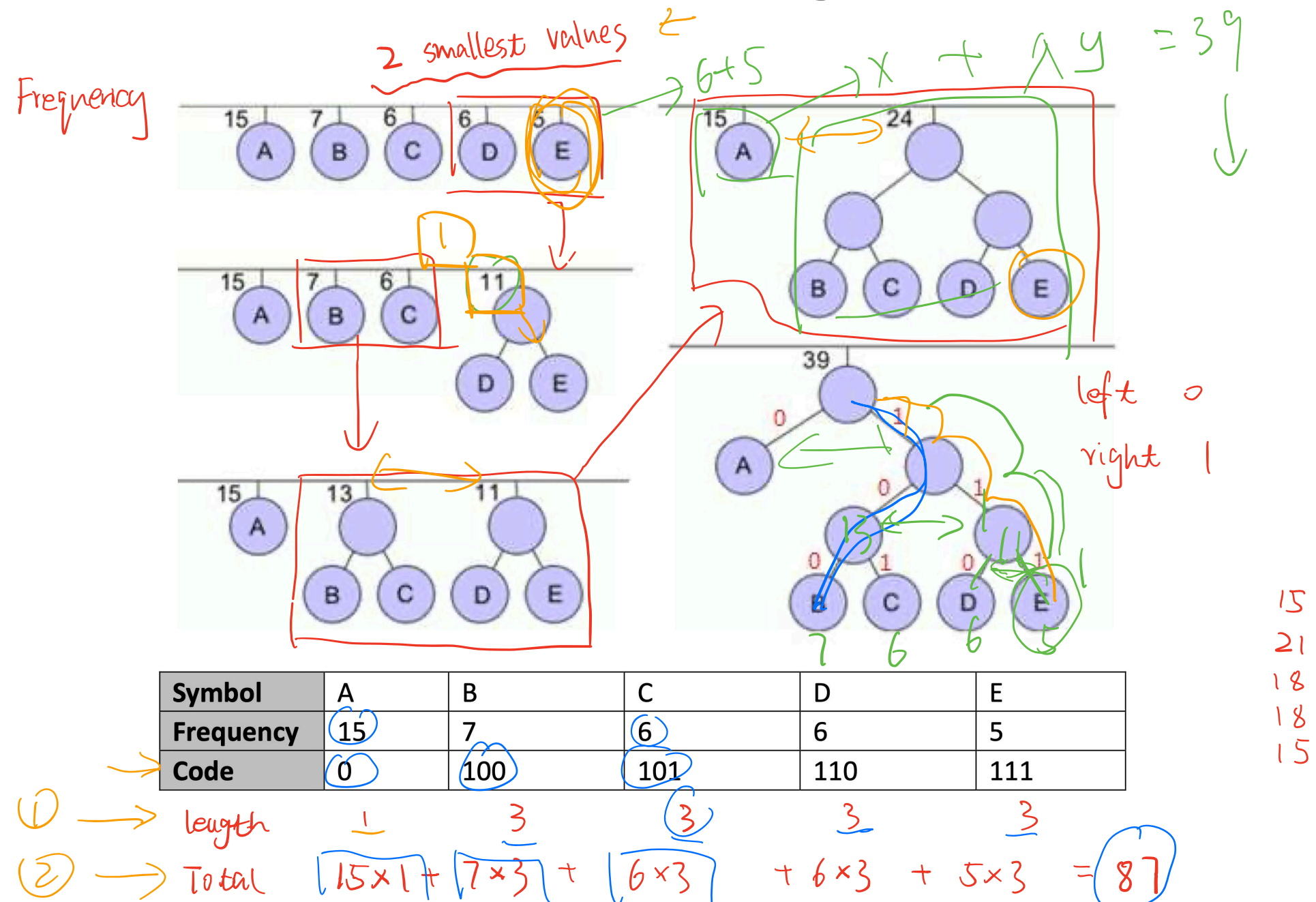
Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The idea of Huffman coding is quite simple: encode frequently used symbols with fewer bits.

Huffman code is produced from the Huffman tree, which is a binary tree.

Initially, we extract the frequency (occurrence count) of each symbol from the text file. In each step, we merge two sub-trees of the least frequency into a new binary tree, until the final binary tree is constructed.

For each symbol, its Huffman code is the path from the root to the symbol node where we encode left-tree as '0' and right-tree as '1'.

757: Huffman Coding (cont.)



757: Huffman Coding (cont.)

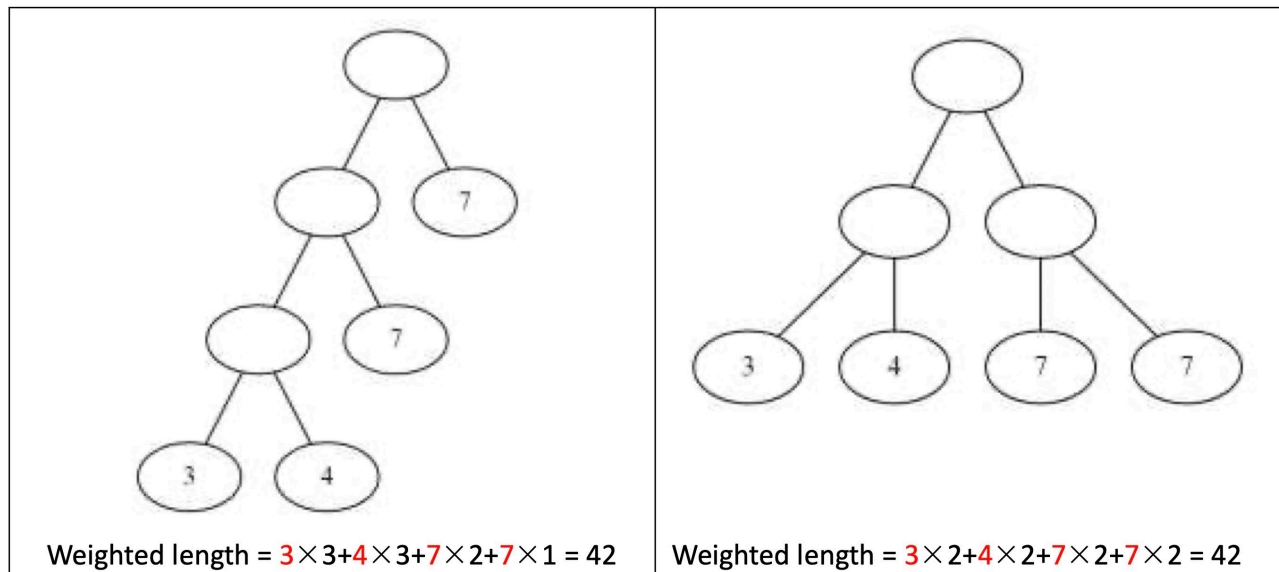
Description

In this question, given the frequency of each symbol, can you calculate the total weighted length of the Huffman code of all symbols?

For example the weighted length of the above Huffman tree is:

$$15 \times 1 + 7 \times 3 + 6 \times 3 + 6 \times 3 + 5 \times 3 = 87$$

The Huffman tree might not be unique when more than two symbols have the same frequency. However, the weighted length is always the same. In the example of four symbols with frequencies $\{3, 4, 7, 7\}$, two possible Huffman trees could be constructed as follows:



757: Huffman Coding

Input

The input contains multiple test cases.

Each test case begins with one integer n ($0 \leq n \leq 100000$), indicating the number of symbols.

The following line gives n positive integers $f_1, f_2, \dots, f_n$ representing the frequency of each symbol where $1 \leq f_i \leq 1000$.

Output

For each test case, print the total weighted length of the Huffman codes in a separate line.

757: Huffman Coding

Sample Input	Sample Output
5 ⁿ → 15 7 6 6 5 frequencies 4 3 4 7 7 6 8 6 5 5 4 3 10 8 1 7 3 6 9 4 2 3 3 - Idea: - calculate code length for every symbols - code length · frequencies - key point: How to find the smallest value in a dynamic set? 2 data structures ⇒ BST, Red-black tree (the leftmost) Heap (Top)	87 42 79 144 - solution: <pre>std::priority_queue<int> q; res = 0; for each f: // frequency q.push(-f) // we want smallest value // After we put all frequencies in the heap while q is not empty: x = q.top() q.pop() ★ y = q.top() q.pop() S = x + y q.push(S) res += S</pre> ⇒ - res You can implement it (heap)

756: Max-Heap

Description

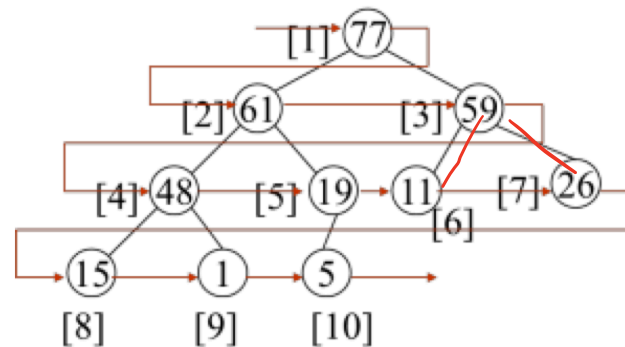
Heap is a specialized tree-based data structure that satisfies the heap property:

- if P is a parent node of C, then the key (the value) of P is either $\geq$ (in a max heap) or $\leq$ (in a min heap) the key of C.

The node at the "top" of the heap (with no parents) is called the root node.

In binary-tree based heap, it can be implemented via an array with property:

if a node in the binary tree has index i, then its two children have index $2i$, $2i+1$ respectively. The following is an example of a Max-Heap:



Index	1	2	3	4	5	6	7	8	9	10
Value	77	61	59	48	19	11	26	15	1	5

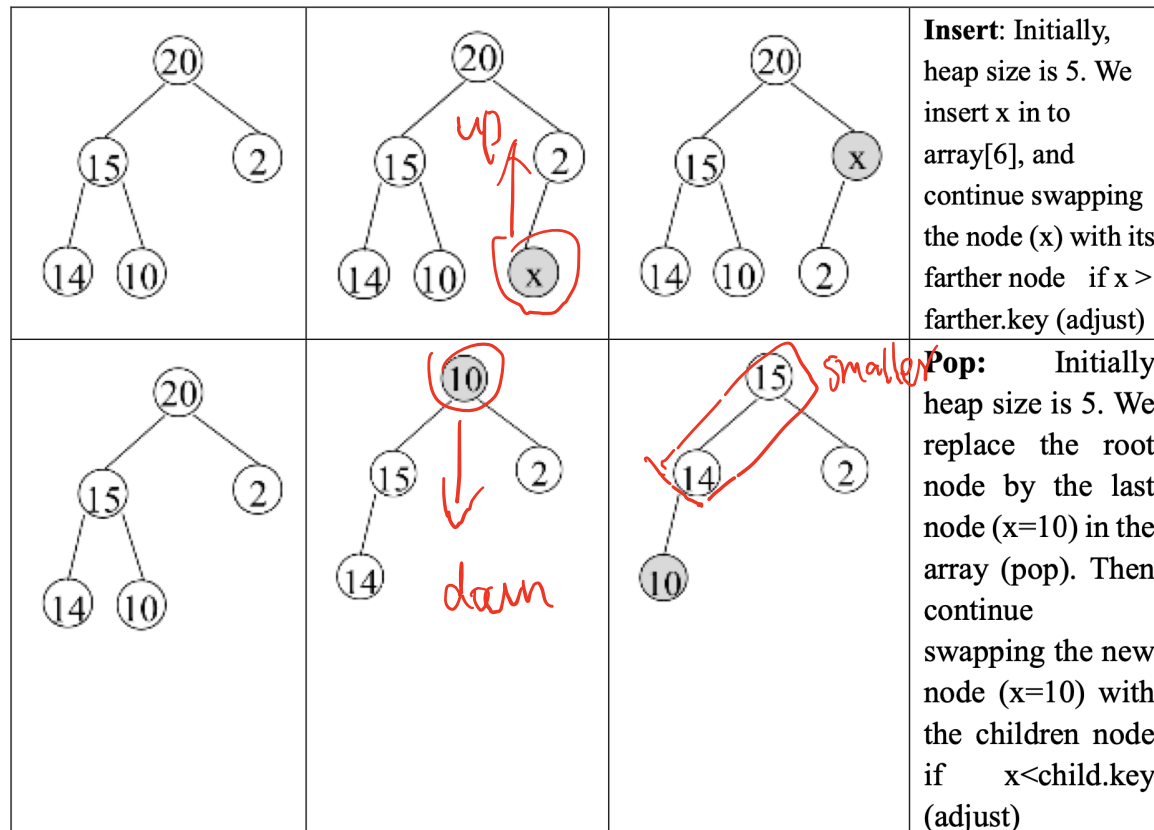
756: Max-Heap

Description

There are two main operations on heap:

Insert a new key: append new key to the end of the array, and then adjust the heap

Pop root node: replace the root by the last node in the array, and then adjust the heap



756: Max-Heap

In this question, you need to implement the data structure of Max-Heap with functions **insert()** and **pop()**.

Libraries such as <algorithm> <queue> already implements the heap (make_heap in <algorithm>, priority_queue in <queue>), therefore **you are not allowed** to use these libraries. 

Input

The input contains multiple test cases.

Each test case begins with one integer n ($0 \leq n \leq 100000$), indicating the number of operations. The following n lines give the operations on the heap, each line follows the format:

"a": **insert** a new number k to the heap, $1 \leq k \leq 1000$.

"p": **pop** (remove) the root node of the heap.

"r": **print** the sum of all numbers in the heap.

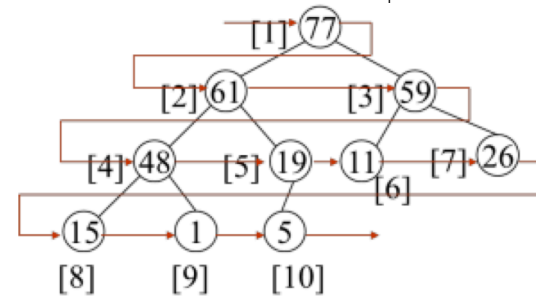
It will guarantee that the heap is not empty when encountered with pop operation.

756: Max-Heap

Output

For each “r” operation,
print the sum in a separate line.

Sample input	Sample output
13 a 61 a 1 a 77 a 19 a 26 a 15 a 59 a 5 a 48 a 11 p p r 8 a 5	184 4 5



Index	1	2	3	4	5	6	7	8	9	10
Value	77	61	59	48	19	11	26	15	1	5

left : $2i$
 right : $2i+1$
 parent : $\lfloor i/2 \rfloor$

2 assistant functions

// To maintain the heap property of the subtree of i

MAX-HEAPIFY(A, i)

$l = \text{LEFT}(i)$

$r = \text{RIGHT}(i)$

if $A[l] > A[i]$

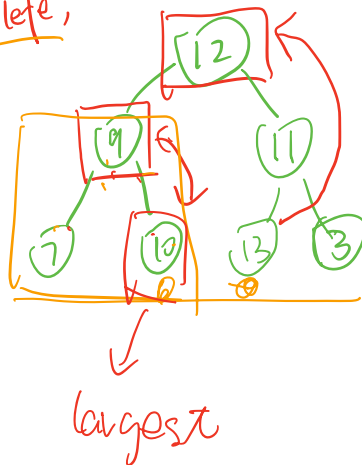
largest = l

else largest = i

if $A[r] > A[\text{largest}]$

largest = r

// find the largest value in i , left, right



if largest $\neq i$

swap($A[i]$, $A[\text{largest}]$)

MAX-HEAPIFY(A, largest)

// subtree of the updated element

// $A[i].2 \rightarrow 10$

INCREASE-KEY(A, i, key)

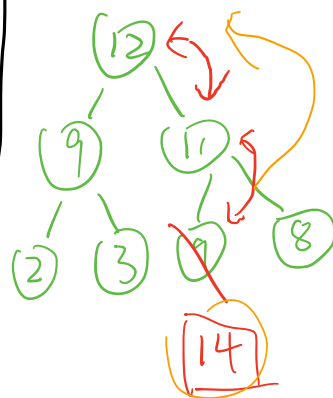
$A[i] = \text{key}$

while $i > 1$ and

$A[\text{PARENT}(i)] < A[i]$

swap($A[i]$, $A[\text{PARENT}(i)]$)

$i = \text{PARENT}(i)$



// POP

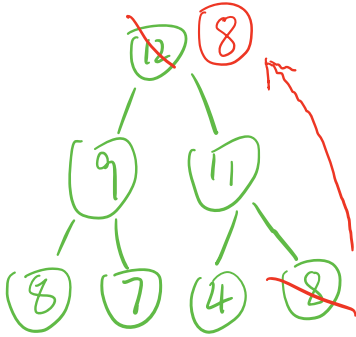
EXTRACT-MAX(X)

max = A[1]

A[1] = A[A.heap-size]

A.heap-size -= 1

MAX-HEAPIFY(A, 1)



then update

// insert key

INSERT(A, key)

A.heap-size += 1

A[A.heap-size] = -∞

INCREASE-KEY(A, A.heap-size, key)

