فرهاد امان ۹۹۳۱۰۰۶

۱- از دو استک استفاده می‌کنیم استک اول برای نگه داری مراحل بازگشت که استفاده می‌شود و از استک دوم برای ذخیره خروجی استفاده می‌شود چراکه خروجی باید به صورت معکوس چاپ شود.

```
stract Node
    data
    Node left, right

Postorder (Node root)
    if root = null
        return
    stack s
    stack o
    s.push(root)
    while (s.notempty())
        Node curr ← s.top()
        s.pop()
        if (curr.have_left)
            s.push (curr.left)
        if (curr.have_right)
            s.push (curr.right)
        o.push (curr.data)
    end while
    while (o.notempty())
        print (o.top())
        o.pop()
    endwhile
```

شبه کد تنها دارای ۲ حلقه می‌باشد در حلقه اول Node ها به ترتیب در ۲ استک push و pop می‌شوند.

چون حرکت رو به بالا در درخت نداریم می‌دانیم که هر Node تنها یک بار در هر استک push خواهد شد.

و همچنین تنها یک بار از هر استک pop می‌شود. بنابراین پیچیدگی کد ما $\boxed{O(n)}$ خواهد بود.

فرهاد امان ۹۹۴۱۰۰۶

۲ - الگوریتم از ۲ مرحله تشکیل شده است. ابتدا بروی درخت پیمایش inorder را اعمال کنیم و المان های به دست آمده را به ترتیب در یک آرایه یا استک اضافه می کنیم. در مرحله ی دوم عناصر داخل آرایه یا استک را به صورت خطی پیمایش کرده و چک می کنیم که صورت صعودی برای آرایه یا نزولی برای استک باشند. می دانیم که اگر درخت باینری ما BST باشد نتیجه پیمایش inorder ها به صورت مرتب شده خواهد بود زیرا در هر Node ابتدا زیر درخت چپ پیش خود Node و در انتها زیر درخت راست پیمایش خواهد شد.

```
inorder (node root, array a)
    if (root = null)
        return
    inorder(root.left, a)
    a.addend (root.data)
    inorder (root.right, a)

is_bst (node root)
    array a
    inorder (root, a)
    for i=1 to a.size -1
        if (a[i-1] > a[i])
            return False
    return True
```

پیمایش inorder در $O(n)$ انجام می شود.
چک کردن صعودی بودن آرایه در $O(n)$ است
بنابراین کل الگوریتم $O(n)$ می باشد.

فرهاد امان ۹۹۳۱۰۰۶

۳ - الف) بر روی درخت T_1 یک از پیمایش های درخت را انجام میدهیم. هنگام عبور از هر node با استفاده از الگوریتم find چک می کنیم که مقدار گره عبوری از درخت T_1 در درخت T_2 هم وجود دارد یا خیر برای اینکه $S_1 \subseteq S_2$ باید تمام اعضای T_1 در T_2 موجود باشد. پیمایش درخت T_1 از $O(n_1)$ و هر عملیات find از $O(\lg n_2)$ می باشد پس مرتبه الگوریتم از $O(n_1 \lg n_2)$ است.

چون به جز چند متغیر از حافظه دیگری استفاده نکرده ایم حافظه اضافه از $O(1)$ است.

```
find (node root, key)
    if(root = null)
        return False
    if(root.value = key)
        return True
    if(root.value < key)
        return find(root.right, key)
    if(root.value > key)
        return find(root.left, key)
```

```
traverse T1 for each node u in T1
    if( find(T2.root, u.value) = False)
        print ( S1 ⊈ S2 )
        end
print ( S1 ⊆ S2 )
```

فرهاد امان ۹۹۴۱۰۰۶

۳ - ب) آرایه a را به اندازه n_1 و آرایه b را به اندازه n_2 در نظر می‌گیریم.
به ترتیب بر روی T_1 و T_2 پیمایش inorder را انجام می‌دهیم و مقادیر را بدون آرایه
های a و b می‌ریزیم. می‌دانیم ۲ آرایه مرتب شده خواهند بود.
حالا از روش Two pointers استفاده می‌کنیم به این صورت که یک pointer بر روی a
و یکی بر روی b قرار می‌دهیم و اینکه b شامل a باشد را چک می‌کنیم.

```
i = 0
j = 0
while ( i < n1 and j < n2 )
    if ( a[i] = b[j] )
        i = i + 1
        j = j + 1
    else
        j = j + 1
if ( i = n1 )
    Print True
else
    Print False
```

در هر بار اجرای حلقه یکی از پوینتر ها جلو می‌رود. پس در حداکثر $O(n_2)$ بار
پایان خواهد یافت. پیمایش روی درخت ها هم
از $O(n)$ در درخت است.
پس در کل $O(n_1+n_2)$ می‌باشد.
صورتی هم تنها از ۲ آرایه با سایزهای n_1 و n_2 استفاده
کردیم.

فرهاد امان ۹۹۳۱۰۰۶

۴- دو تابع heapify-up و heapify-down را تعریف می‌کنیم و از آن برای بخش‌های الف و ب و ج استفاده می‌کنیم.

heapify-down همان heapify عادی است.

```
heapify_down (array A, i)
    left ← 2i
    right ← 2i+1
    largest ← i
    if left ≤ heap_size
        if A[left] > A[largest]
            largest ← left
    if right ≤ heap_size
        if A[right] > A[largest]
            largest ← right
    if largest ≠ i
        swap (A[i], A[largest])
        heapify_down (A, largest)
```

heapify-up یک نود را تا هنگامی که از پدر خود بزرگتر است به سمت بالا می‌برد.

```
heapify_up (array A, i)
    Par ← ⌊i/2⌋
    if Par > 0
        if A[i] > A[Par]
            swap(A[i], A[Par])
            heapify_up (A, Par)
```

حال با استفاده از این ۲ تابع حالت‌های الف و ب و ج را انجام می‌دهیم.

فرهاد امان ۹۹۴۱۰۰۶

۴ - الف) در مرحله اول با فرض اینکه می‌خواهیم گره i را حذف کنیم. گره آخر heap را به جای گره i قرار می‌دهیم. پس سایز heap را یک واحد کاهش می‌دهیم.

حال ۳ حالت وجود دارد اگر مقدار جدید گره i برابر مقدار قبلی بود کار تمام است.

اگر مقدار جدید بیشتر از مقدار قبلی بود از heapify-up و اگر کمتر بود از heapify-down استفاده می‌کنیم.

```
i // delete i th node from heap
oldi ← A[i]
A[i] ← A[heap-size]
heap-size ← heap-size - 1
if A[i] > oldi
    heapify-up (A, i)
elsif A[i] < oldi
    heapify-down (A, i)
```

ب) ابتدا یک واحد به اندازه heap اضافه می‌کنیم و مقدار جدید را در انتهای heap قرار می‌دهیم سپس از heapify-up استفاده می‌کنیم.

```
new-value // insert new-value to heap
heap-size ← heap-size + 1
A[heap-size] ← new-value
heapify-up (A, heap-size)
```

فرهاد امان ۹۹۴۱۰۰۶

۴ - ج) مقدار قبلی گره مورد نظر را ذخیره می‌کنیم و پس آن را عوض می‌کنیم با دیگر ۳ حالت وجود دارد اگر مقدار جدید برابر مقدار قدیمی است کار تمام است . اگر مقدار جدید بزرگتر بود از heapify-up و اگر کمتر بود از heapify-down استفاده می‌کنیم .

```
i
new-value // change i th node to new-value
oldi ← A[i]
A[i] ← new-value
if A[i] > oldi
    heapify-up (A,i)
elseif A[i] < oldi
    heapify-down(A,i)
```