محمدحسین نظری - ۹۹۳۱۰۹۱

۱) در این روش از Iterative postorder traversal استفاده می‌کنیم

که با استفاده از دو استک این کار را انجام می‌دهیم.

روال کار به این شکل است که یک نود به اسم temp داریم و آن را برابر با root درخت

قرار داده و بر روی آن کار می‌کنیم. مقدار نود temp را ابتدا استک دوم (S2)، Push می‌کنیم،

حال فرزندهای چپ و راست (به ترتیب) temp را در S1، Push می‌کنیم، بعد از آن

temp را برابر S1.pop قرار داده و در S2، Push می‌کنیم و مانند مرحله قبل دوباره

فرزندهای چپ و راست (به ترتیب و در صورت وجود) temp را در S1 push کرده و همین

حلقه را تا زمانی ادامه می‌دهیم تا S1 خالی شود و در این مرحله S2 جواب

مسئله است که با pop کردن آن به پیمایش postorder درخت باینری می‌رسیم.

* برای اینکه در حلقه while بتوانیم کارمان را انجام دهیم، نود اولیه root را

در S1، push کرده و هر بار node temp را برابر S1.pop قرار

می‌دهیم.

* متد isempty اگر استک خالی باشد true و در غیر این صورت false می‌دهد.

* برای چاپ postorder، S2 را تا زمانی که خالی شود، pop می‌کنیم.

Iterative postorder traversal (root)

```
1  stack sl, sr;
2  if (root == null) {
3      break; }
4  sl.push(root);
5  while (!sl.isempty()) {
6      node temp = sl.pop;
7      sr.push(temp);
8      if (temp.left != null) {
9          sl.push(temp.Left); }
10     if (temp.right != null) {
11         sl.push(temp.right); } }
12 while (!sr.isempty()) {
13     node temp = sr.pop;
14     print temp.key; }
```

تحلیل زمانی

اس تعداد گره‌های درخت را n در نظر بگیریم

خطوط 1 ، 2 ، 3 ، 4 از مرتبه O(1) هستند .

خطوط 5 تا 11 از مرتبه O(n) هستند.

خطوط 12 تا 14 از مرتبه O(n) هستند (دقیقاً n بار حلقه‌ی while تکرار می‌شود ، غیر از n+1 بار حلقه‌ی for تکرار می‌شود)

پس کل الگوریتم از مرتبه زمانی O(n) است.

④ یک راه برای اینکه یک درخت باینری، BST هست یا نه این است که بیایم کوچکترین مقدار زیردرخت چپ از ریشه آن کوچکتر است (در هر مرحله و به کل بازگشتی) و به همین صورت کوچکترین مقدار زیردرخت راست از ریشه آن بزرگتر است (در هر مرحله و به کل بازگشتی). برای بازده و سرعت بیشتر الگوریتم برای اینکه مقدار یک نود چندین بار محاسبه نشود، از یک تابع به نام isBSTUtil استفاده می‌کنیم و مقدارهای گفته شده در بالا را برای هر نود در زیردرخت‌هایش از پایین به بالا حساب می‌کنیم و در نود بالاتر از نتایج قبلی استفاده می‌کنیم.

* از تابع‌های پیش‌فرض Min و Max برای بدست آوردن کوچکترین و بزرگترین مقدار درخت استفاده می‌کنیم.

* درخت هیچ مقدار تکراری نباید داشته باشد

* پیچیدگی زمانی کد از مرتبه O(n) است

* توابع Min و Max هرکدام از مرتبه O(h) یا O(log n) هستند پس در کل باعث افزایش پیچیدگی زمانی کد نمی‌شوند که همان O(n) است چونکه باید برای همه‌ی نودهای درخت یکبار تعداد n عملیات بازدید انجام شود.

* درخت تهی، BST به حساب آورده شده.

② isBST (t.root):

```
isBSTUtil (node, min, max) {
    if (node == null) {
        return True ;}
    if (node.key < min || node.key > max) {
        return False ; }
    if (isBSTUtil (node.left , min , (node.key)-1 &&
        isBSTUtil (node.right , (node.key)+1 , max)) {
        return True ; }
    else {
        return False;} }

return isBSTUtil (t.root , Min(t) , Max(t)) ;
```

(۳)

الف) برای رسیدن به جواب در این مسئله و با پیچیدگی زمانی گفته شده، می‌توانیم جستجوی هر نود در T_2 همانند bST، $\lg n_2$ بار طول می‌کشد، جستجوی هر n نود در درخت جدید هم n بار طول می‌کشد، پس گره‌های درخت T_1 را در درخت T_2 جستجو کنیم و اگر جواب همه مثبت بود پس زیرمجموعه آن است

$$\overbrace{O(n_1)}^{\text{گشتن عناصر } T_1} \text{ و } \overbrace{O(\lg(n_2))}^{\text{جستجو در } T_2} \longrightarrow O(n_1 \lg(n_2))$$

$O(1)$ هم برای تعریف متغیرها ... استفاده می‌شود.

تشبیهی که بیان شد: مثلاً inorder یا inorder(i) یعنی عنصر iام حاصل از inorder در نظر بگیریم.

```
issubset (n1 ; n2 , T1, T2):
  int k = 0 ;
  boolean r ;
  for (i = 0 ; i < n1 ; i++) {
     r = search T2 (inorder T1 (i)) ;
     if (r == true)
        k++ ; }
  if (k == n1)
     return true
  else
     return false
```

ج) دو آرایه تعریف می‌کنیم برای هردو درخت که پیمایش با ترتیب یکسانی داشته باشند. مثلا سورت شده صعودی یا نزولی باشند یا پیمایش inorder یا ... باشند.

حال کم‌کم عناصر T_1 یعنی n_1 تا را به ترتیب از اول T_2 شروع به مقایسه می‌کنیم و اگر به تعداد مساوی رسیدیم، مقایسه برای عنصر بعدی T_1 در T_2 را از index یا تعداد قبلی برابر شده در T_2 ادامه می‌دهیم نه از ابتدا چونکه هردو از یک پیمایش یا سورت هستند.

برای آرایه اول قطعاً به n_1 بار پیمایش نیاز داریم و در آرایه دوم حداکثر به n_2 پیمایش یا چک کردن نیاز هست پس در کل به $O(n_1+n_2)$ مرتبه نیاز داریم.

```
issubset:
  a1[] = T1.sort;
  a2[] = T2.sort;
  int t=0, k=0;
  for( int i=0; i<n1; i++) {
     if (t <= n2) {
        while (a1[i] != a2[t]) {
            t++; }
        k++; }
     else
        break; }

if (k == n1)
    return true
else
    return false
```

*حافظه اضافی هم برای درختن(?) n_1 و n_2 عنصر در آرایه و ساختن آرایه مورد نیاز است.

(۲)

الف) برای حذف گره دلخواه از یک max heap ، ابتدا آن گره را با آخرین عنصر آرایه heap جایگزین کرده (راست ترین برگ) ، سپس تعداد اعضای max heap را یک واحد کاهش می دهیم تا عنصر مورد نظر حذف گردد و در آخر هم max heapify می کنیم تا جایگاه گره ریشه max heap بودن را نقض نکند .

```
delete(A, i):

A[i] = A[heap.size] ;

heap.size -- ;

max_heapify (A,i) ;
```

ب) ابتدا سایز آرایه max.heap را یک واحد افزایش می دهیم و سپس گره مورد نظر را به انتهای آرایه اضافه کرده و آرایه نهایی را max heapify می کنیم .

```
insert (A, x):

heap.size ++ ;

A[heap.size] = x ;

max_heapify(A, heap.size) ;
```

ج) مسئله را به دو بخش تقسیم می‌کنیم، در بخش اول اگر مقدار به گره‌ای که تغییر کند افزایش یابد، جای گره را با والدش عوض می‌کنیم تا زمانی که کوچکتر از والدش باشد (یعنی به سمت بالا می‌رود).

در بخش دوم اگر مقدار کوچکتر شود، عکس حالت قبلی را انجام می‌دهیم. به پایین حرکت می‌کنیم تا زمانی که به بچه‌ی کوچکتر برسیم.

```
update(node, x):

  if(x > node.key){
      node.key = x;
      while(node.key > node.parent.key){
          swap(node, node.parent);}}

  if(x < node.key){
      node.key = x;
      while(node.key < max(node.Leftchild.key, node.rightchild.key)){
          swap(node, max(node.children));}}
```

در این جا منظور گره‌ای از فرزندان است که مقدار بزرگتری دارد و null نیست.