به نام خدا

مشخصات :

نام : مهدی جواهری صابر

شماره دانشجویی : ۹۲۴۳۰۸۸۰۱۷

EMAIL : MEHDYCZ1994@GMAIL.COM

ساختمان داده های مورد استفاده :

در این پروژه به غیر از ساختمان داده ی درخت دودویی جستجو (که جزو اجزای غیر قابل اجتناب این پروژه میباشد) از ساختمان داده ی پشته استفاده شده است

دلیل استفاده از پشته :

در تابع افزودن و حذف کردن یک گره ، نیاز به افزودن و کاستن جمعیت گره های پدر وجود دارد تا درخت جدید در تعریف درخت مرتبه آماری صدق کند

پشته برای تابع Add() :

برای این کار ما یک پشته تعریف میکنیم و در حین یافتن محل گره برای افزودن گره ی جدید ، گره هایی را که پیموده ایم داخل پشته قرار میدهیم تا در انتهای عملیات افزودن ، جمعیت گره های داخل پشته را بیافزاییم
(هشدار : نمی توانیم همین که در حین پیمایش برای یافتن محل گره برای افزودن هستیم جمعیت پدران را بیافزاییم ، چرا که ما نمیدانیم داده تکراری است یا نه ، برای همین منظور از پشته استفاده میکنیم تا در انتها که مطمئن شدیم افزودن موفقیت آمیز بود جمعیت ها را تغییر دهیم)

پشته برای تابع Remove() :

در تابع حذف نیاز به پشته محسوس تر میگردد چرا که در حالتی که اگر گره ی مورد نظر برای حذف دارای دو فرزند باشد باید بار دیگر پیمایشی برای یافتن گره ی بعد از آن در حالت میان ترتیب انجام دهیم و جای دو گره را تغییر دهیم و در این حالت است که پشته کاربردی میشود

توضیحات توابع استفاده شده :

Add() : در تابع ما یک حلقه While تعریف میکنیم تا محل قرارگیری گره ی جدید را بیابیم
در حین یافتن محل گره ی جدید گره هایی را که پیموده ایم را داخل پشته قرار میدهیم
یک تابع **SizeChanger()** نیز برای خالی کردن پشته و افزودن جمعیت گره های پدر میباشد که توضیحات لازم آن داخل کد نوشته شده است

Remove() : در این تابع برای حالت حذف برگ و حذف گره ی دارای ۱ فرزند ، یک تابع مجزا **NodeRemover()** نوشتم تا از دوباره نویسی کد جلوگیری شود (چون حالت حذف گره ی دارای ۲ فرزند نیز در باطنش این تابع وجود دارد)
و همچنین برای یافتن محل گره ای که میخواهیم حذف کنیم ، یک تابع دیگر **rsearch()** برای جستجو تعریف کردم تا گره های پدر را بتوانیم داخل پشته قرار دهیم
تذکر : در حالتی که گره دارای ۲ فرزند است گره ی بعد از آن در پیمایش میان ترتیب ، در چپ ترین گره ی گره ی سمت راست میباشد

Search() : در این تابع اگر گره ی ریشه برابر داده بود که جستجو پایان یافته است
اگر کوچکتر از ریشه بود سمت چپ ریشه را به صورت بازگشتی جستجو میکنیم
اگر بزرگتر از ریشه بود سمت راست ریشه را به صورت بازگشتی جستجو میکنیم

Trace() : در این تابع ابتدا سمت چپ ریشه را به صورت بازگشتی پیمایش میکنیم و سپس خود ریشه را ملاقات کرده و
بعد سمت راست را به صورت بازگشتی پیمایش میکنیم

K_index() : این تابع k امین عنصر کوچکتر در پیمایش میانترتیب را برمیگرداند
برای این منظور یک اشاره گر کمکی static به نام P ایجاد میکنیم ، تا در آن پدر ریشه ی فعلی نگهداری شود
ابتدا k با جمعیت ریشه مقایسه میشود و در صورتی که k بزرگتر از آن بود NULL برمیگرداند
و زمانی که k صفر شد(حالتی که به اندازه کافی روی درخت جلو رفته ایم و به گره ی مورد نظر رسیده ایم) داده ی گره ی فعلی را برمیگردانیم
شرط بعدی بررسی نال بودن گره ی فعلی است و در صورتی که نال باشد از روی شرط رد شده و NULL برمیگرداند یعنی زمانی که مثلا به آخرین گره ی سمت چپ رسیده باشیم
در صورتی که جمعیت ریشه فعلی بزرگتر یا مساوی K باشد اگر سمت چپ نال نباشد به سمت چپ گره میرویم ولی اگر نال باشد k امین عنصر حتما در سمت راست قرار دارد ، به سمت راست میرویم و یکی از k کم میکنیم زیرا یک گره را رد کردیم

در حالتی که جمعیت ریشه فعلی کوچکتر از k باشد ، نیازی به بررسی زیر شاخه های بعدی نیست ، به گره ی پدر برمیگردیم و سمت راست ان را بررسی میکنیم و در اینجا چون زیر شاخه ها را رد کردیم ، باید جمعیت آن را نیز از k کم کنیم و چون به سمت راست گره ی پدر رفتیم ، یعنی عملا از روی گره ی پدر رد شدیم ، باید یکی از k کم کنیم

این تابع به صورت بازگشتی آنقدر میگردد و از k کم میکند ، تا روی گره ی مذکور برود و داده ی آن را برگرداند

X_rate() : این تابع تعداد عناصر درخت که از X کوچکتر یا مساوی هستند را بازمیگرداند و به صورت بازگشتی پیاده سازی شده است

در این تابع یک حلقه while قرار دارد که داخل آن ابتدا بررسی میکنیم گره فعلی کوچکتر مساوی مرتبه هست یا نه اگر کوچکتر نبود حتما سمت چپی های آن باید کوچکتر یا مساوی باشند

در حالتی که گره فعلی کوچکتر یا مساوی مرتبه باشد :

اگر گره دقیقا مساوی مرتبه باشد پس جمعیت سمت چپ آن + یک را برمیگردانیم

اگر گره کوچکتر از مرتبه باشد جمعیت سمت چپ آن + یک + جمعیت((بررسی سمت راست آن)) را برمیگردانیم