گزارش تمرین سری سوم

هدف ازین تمرین پیاده سازی درخت سرچ باینری می باشد.

ابتدا کلاس node را درن کلاس bst تعریف میکنیم به اسن صورت که هر نود دارای parent و right and left node و دارای یک مقدار می باشد. دو constructor برای این کلاس تعریف میکنیم که یکی هنگام صدا زدن تمامی نود های والد و چپ و راست را برابر nullptr قرار میدهد و دیگری مقداری که میخواهیم را به نود نسبت میدهد.

یک تابع show برای نود تعریف کرده ایم که در صورت وجود مقدار نود ان را چاپ و در صورت وجود والو و نود های چپ و راست آن ها را نیز چاپ کند.

برای تابع BST constructor ابتدا یک نود میسازیم و پوینتر rot که یک پوینتر کمکی است را برابر nullptr قرار میدهیم و مقدار s که نمایشگر اندازه ی درخت است را برابر صفر قرار داده و ارایه ی vec که ترتیب های مختلف درخت را در خود نگه میدارد را ریست میکنیم.

برای تابع copy constructor باید اولا درخت را کپی کنیم دوما سایز درخت را کپی کنیم. کپی کردن سایز درخت به آسانی امکان پذیر است ولی برای کپی کردن درخت نیاز به یک تابع بازگشتی می باشد که آن را با تابع copy نمایش داده ایم.

تابع بازگشتی کپی به این صورت عمل میکند که در هر مرحله یک پوینتر به نود می سازد که مشخصه های نود ساخته شده را برابر مشخصه های نود درخت اصلی قرار میدهد. روند کپی کردن درخت اینگونه است که ابتدا شاخه های سمت چپ را کپی کرده و زمانی که به null میرسد، شاخه های سمت راست را کپی میکند و این کار را به صورت بازگشتی با صدا زدن دوباره ی تابع انجام میدهیم.

برای تابع destructor نیازی به delete پوینتر های smart نیست اما برای اطمینان فضای وکتور vec را ازاد میکنیم.

تابع add: ورودی تعریف شده در صورت سوال یک عدد می باشد. اما در درون تابع add تابع دیگری صدا زده میشود که یک پوینتر نود و ورودی int را قبول میکند. ورودی این تابع نود proot می باشد که این نود پوینتری است به نود ریشه ی درخت و این پوینتر در طول کلاس همواره به ریشه ی درخت اشاره میکند و تغییر نمیکند مگر ریشه ی درخت دچار تغییر شود و برای پیمایش درخت از پوینتر کمکی rot استفاده میکنیم. قبل از صدا زدن تابع باید از اینکه ایا درختی وجود دارد یا نه اطمینان حاصل کنیم چون ممکن است حالتی پیش بیاید که همه ی نود های درخت را حذف کنیم و سپس بخواهیم دوباره نود به درخت اضافه کنیم.

تابع add: این تابع دو ورودی پوینتر به نود و عدد میگیرد. ابتدا چک میکند که ایا درختی وجود دارد یا خیر و اگر وجود نداشت پوینتر proot and rot را ست میکند و سیاز را اضافه میکند. سپس اگر درختی از قبل وجود داشت باید چک کند عدد ورودی در هر مرحله (یعنی ابتدا ریشه ی درخت سپس نود سطح ۱ و سپس نود سطح ۲) از مقدار نود سطح بالاتر خود کمتر است یا بیشتر که این با تابع بازگشتی قابل انجام است. اگر به مرحله ای رسیدیم که سطح دیگری وجود نداشت نود جدیدی ساخته و مشخصه های ان را ست میکنیم و سایز را یکی افزایش میدهیم.

تابع +operator: ابتدا از copy constructor استفاده میکنیم و سپس برای object کپی شده تابع Add را صدا زده و objectکپی شده را باز میگردانیم. در این حالت روی object اصلی تغییری صورت نمیگیرد.

تابع search: ابتدا نود های سمت چپ را بازبینی میکنیم و سپس نودهای سمت راست را به صورت بازگشتی سرچ میکنیم و در هر مرحله اگر مقدار مورد نظر را یافتیم ۱ و اگر به انتهای درخت رسیدیم و مقدار را نیافته بودیم صفر برمیگردانیم.

تابع remove: درون این تابع تابع Remove دیگری صدا زده میشود که پوینتر به نود ریشه یعنی proot و عدد را میدهیم.

تابع remove: برای این تابع یک پوینتر کمکی دیگر به اسم temp درست کرده ایم. تابع به صورت بازگشتی همان الگویی که برای سرچ داشتیم را پیاده میکنیم تا یا عددی که میخواهیم حذف کنیم در درخت پیدا کنیم یا اگر همه ی درخت را پیمایش کرده و به null رسیدیم چاپ بشود که عدد مورد نظر در درخت موجود نمیباشد. اگر عدد مورد نظر را یافتیم ۳ حالت وجود دارد:

۱) فرزندی نداشته باشد. در این حالت ۲ شرایط پیش می اید که یا خود ریشه باشد و سایز درخت ۱ بوده باشد یا سایز درخت بزگتر از یک باشد که این شرط را با پوینتر والد چک میکنیم که ایا وجود دارد یا خیر. اگر درخت فقط یک نود ان هم ریشه داشته باشد proot and rot ریست میشوند و در غیر اینصورت ان سمتی که عدد قرار گرفته از والد ریست میشود.

۲) یک فرزند داشته باشد. فرض میکنیم که فقط فرزند چپ دارد، ابتدا پوینتر کمکی را روی فرزند چپ ست میکنیم. در این حالت ۲ حالت شرایط پیش می اید که ایا عدد مورد نظر در ریشه است یا خیر. اگر در ریشه است به راحتی proot را ریست و برابر پوینتر کمکی قرار میدهیم. و اگر در ریشه نباشد جای نود مورد نظر که میخواهیم حذف شود را با فرزند خود جا به جا میکنیم برای این کار از تابع swap استفاده میکنیم.

۳)دو فرزند داشته باشد. در این حالت ابتدا باید یک قدم به سمت راست و تا جایی که میشود به سمت چپ حرکت کنیم و نود مورد نظر را بیابیم که این کار را با تابع FindMin انجام داده ایم. در این حالت نیز دو شرایط پیش می اید. اگر والد کمترین مقدار یافت شده ریشه باید و نود مورد نظر برای حذف ریشه باشد باید سمت چپ ریشه را به سمت چپ نود راست منتقل کنیم و مقدار proot را روی نود سمت راست ست کنیم. در غیر اینصورت کمترین مقدار یافته شده را با مقدار نود مورد نظر جا به جا کرده و نودی که دارای کمترین مقدار می باشد را از درخت حذف میکنیم.

تابع inorder: ابتدا باید وکتور vec را ریست کرده و تابع inorder دیگری با را با پوینتر ریشه صدا میزنیم.

تابع inorder: در این تابع ابتدا تا جایی که میتوانیم به سمت چپ حرکت میکنیم و عدد ها را به وکتور اضافه میکنیم و سپس خود ریشه را اضافه میکنیم و سپس شاخه های سمت راست را اضافه میکنیم.

تابع preorder: ابتدا باید وکتور vec را ریست کرده و تابع preorder دیگری با را با پوینتر ریشه صدا میزنیم.

تابع preorder: ابندا خود ریشه را به وکتور اضافه کرده سپس سمت چپ و سپس سمت راست را به وکتور اضافه میکنیم.

تابع postorder: ابتدا باید وکتور vec را ریست کرده و تابع postorder دیگری با را با پوینتر ریشه صدا میزنیم.

تابع postorder: ابتدا سمت چپ را اضافه کرده و سپس سمت راست و سپس خود ریشه را اضافه میکنیم.

تابع []operator: در این تابع روی درخت کنونی پیمایش inorder را انجام میدهیم و index ورودی از وکتور را return میکنیم.

تابع show : در این تابع روی درخت کنونی پیمایش inorder را انجام میدهیم و اعداد ان را چاپ میکنیم.