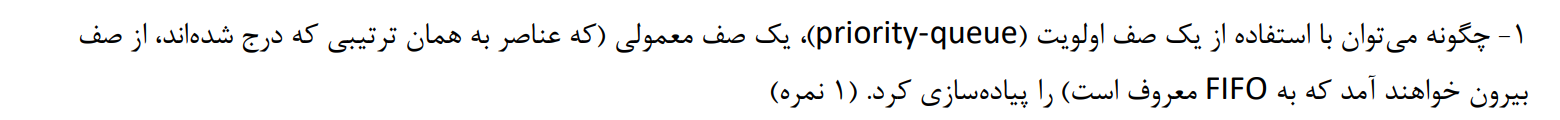
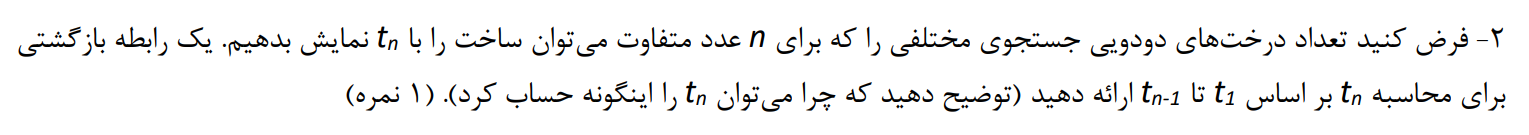
به نام خدا

تمرین شماره ۳ درس ساختمان داده‌ها و الگوریتم‌ها

چمران معینی : ۹۹۳۱۰۵۳



می‌دانیم که صف اولویت‌دار، چیزی که افزون بر صف معمولی دارد این است که می‌توانیم اولویتی را برای هر عنصرِ جدید هنگام insert تعیین بکنیم. با این حساب، کافی‌ست هربار که عنصری را اضافه می‌کنیم، آن را در آخرین اولویت بگذاریم، به این ترتیب مشابه یک صف معمولی عمل می‌شود و عناصر هرچه زودتر گذاشته شده باشند، زودتر خارج می‌شوند.



اگر فقط یک عنصر داشته باشیم، مشخص است که فقط یک راه برای تبدیل آن به درخت دودویی جست و جو داریم،‌ یعنی:

اگر دو عنصر داشته باشیم، دو انتخاب برای ریشه داریم. بعد از انتخابِ ریشه هم t1 حالت برای چینش بقیه‌ی عناصر، یعنی:

در حالتی هم که سه عنصر داشته باشیم، سه انتخاب برای ریشه داریم. اگر بزرگ‌ترین یا کوچک‌ترین عنصر ریشه‌مان باشد هر دو عنصرِ دیگر در یک طرف قرار می‌گیرند، یعنی t2 حالت برای چینشِ بقیه‌ی اعضا داریم. اگر هم عوض میانی را انتخاب کرده باشیم t1 \* t1 حالت برای چینشِ بقیه‌ی اعضا داریم، پس:

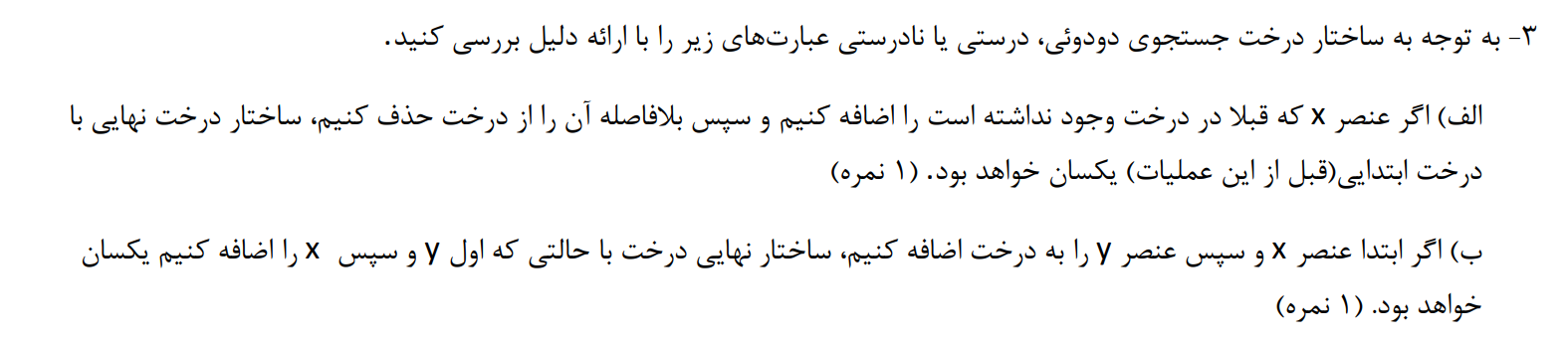
در حالتی که چهار عنصر داریم، چهار انتخاب برای ریشه داریم. در حالتی که بزرگ‌ترین یا کوچک‌ترین عضو را انتخاب کرده باشیم، هر سه عضو در یک طرف قرار می‌گیرند و t3 حالت برای چینش‌شان خواهیم داشت. در حالتی که عضو دوم یا سوم را انتخاب کرده باشیم هم t1 \* t2 حالت برای چینشِ بقیه‌ی اعضا داریم، پس:

وقتی پنج عنصر داشته باشیم هم، پنج انتخاب برای ریشه داریم. اگر بزرگ‌ترین یا کوچک‌ترین عضو باشد t4 حالت برای چینشِ بقیه‌ی عناصر داریم. اگر عضو دوم یا چهارم باشد t1 \* t3 حالت برای چینشِ بقیه‌ی اعضا داریم. اگر عضو سوم ریشه بشود هم t2 \* t2 حالت برای چینشِ بقیه‌ی اعضا داریم، پس:

به همین ترتیب و با همین منطق خواهیم داشت:

پس tn را به این شکل می‌نویسیم:

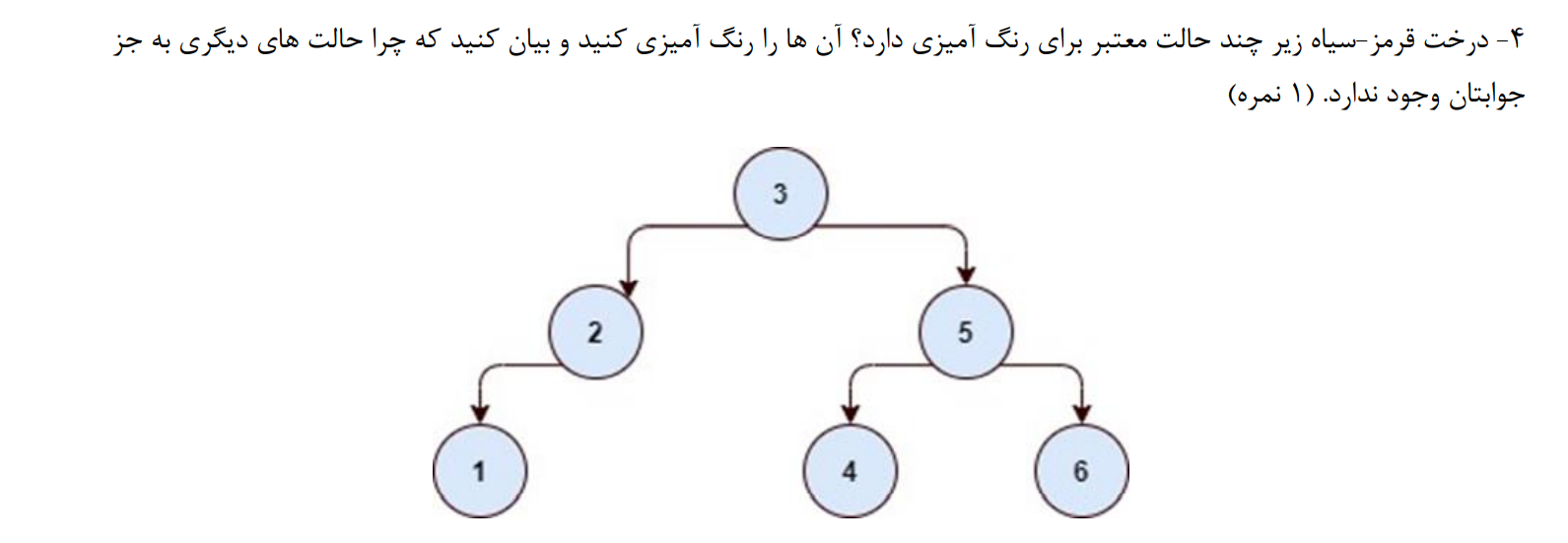
می‌توان این رابطه را به این شکل توضیح داد که از اولین جمله، مربوط به حالتی‌ست که بزرگ‌ترین عنصر را به عنوان ریشه انتخاب کرده باشیم، در این حالت تمام n-1 در یک سمت قرار می‌گیرند و می‌توان آن‌ها را به t(n-1) حالت چید. دومین جمله، مربوط به زمانی‌ست که دومین عنصر بزرگ را به عنوان ریشه انتخاب کنیم، در این حالت یک عنصر در یک سمت قرار می‌گیرد که t1 چینش دارد، بقیه‌ی اعضا هم در سمت دیگر قرار می‌گیرند که t(n-2) چینش خواهند داشت، الی آخر.



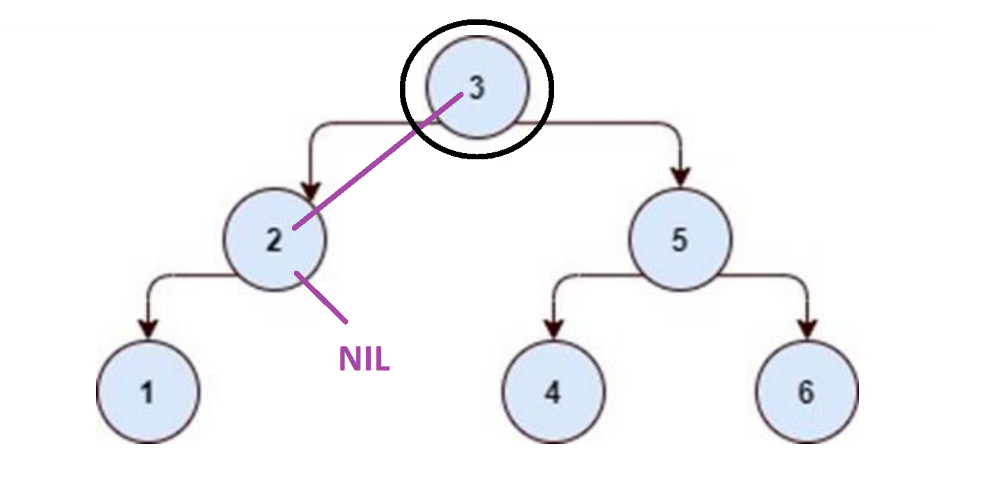
الف) می‌دانیم که عملیاتِ insert در یک ددج، تغییری در جایگاهِ قبلیِ عناصر ایجاد نمی‌کند. تنها کاری که می‌کنیم این است که می‌گردیم و خانه‌ی نال مناسب را پیدا می‌کنیم و عنصرِ جدید را به جای آن می‌گذریم.

اما می‌دانیم که عملیاتِ حذف می‌تواند با تغییراتی در عناصرِ دیگر هم همراه باشد، اما در چه صورت؟ در صورتی که عنصرمان فرزندانی داشته باشد، اما می‌دانیم که عنصری که تازه اضافه شده، به عنوان یک برگ اضافه می‌شود و فرزندی ندارد و با توجه به این که بلافاصله بعد از اضافه کردن اقدام به حذف شده، در نتیجه فقط یک لحظه عنصری به عنوان فرزندِ یکی از بزرگ‌ها اضافه می‌شود و سپس دوباره حذف می‌شود و درخت دقیقا مانند قبل خواهد بود، پس این گزاره صحیح است.

ب) اشتباه است. برای نقض آن، یک مثال نقض کافی‌ست. فرض کنید ددج‌ای داشته باشیم با ریشه‌ی ۱۰، اگر اول ۹ را اضافه کنیم و بعد ۸ را، ۹ فرزندِ ریشه می‌شود و ۸ هم فرزندِ ۹ می‌شود، اما در صورتی که اول ۸ را اضافه کرده باشیم، ۸ فرزندِ ریشه می‌شود و ۹ در سمت راستِ ۸ قرار می‌گیرد.



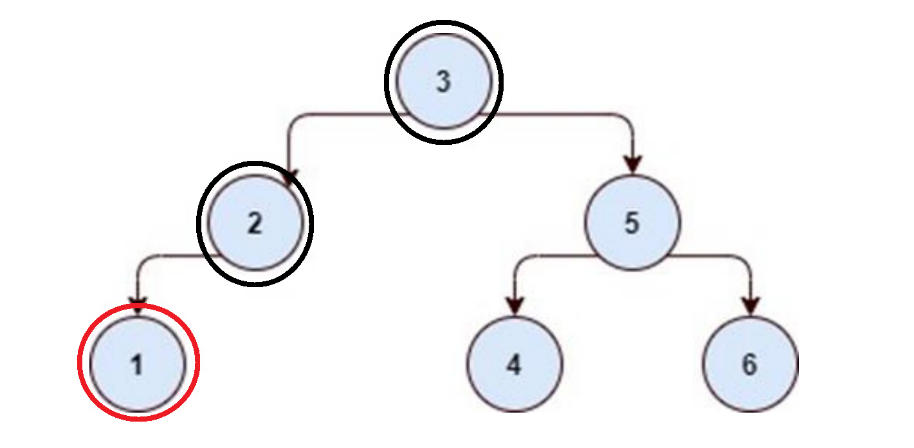
می‌دانیم که ریشه باید سیاه باشد، پس ابتدا آن را رنگ می‌کنیم و بعد سراغ بقیه می‌رویم.



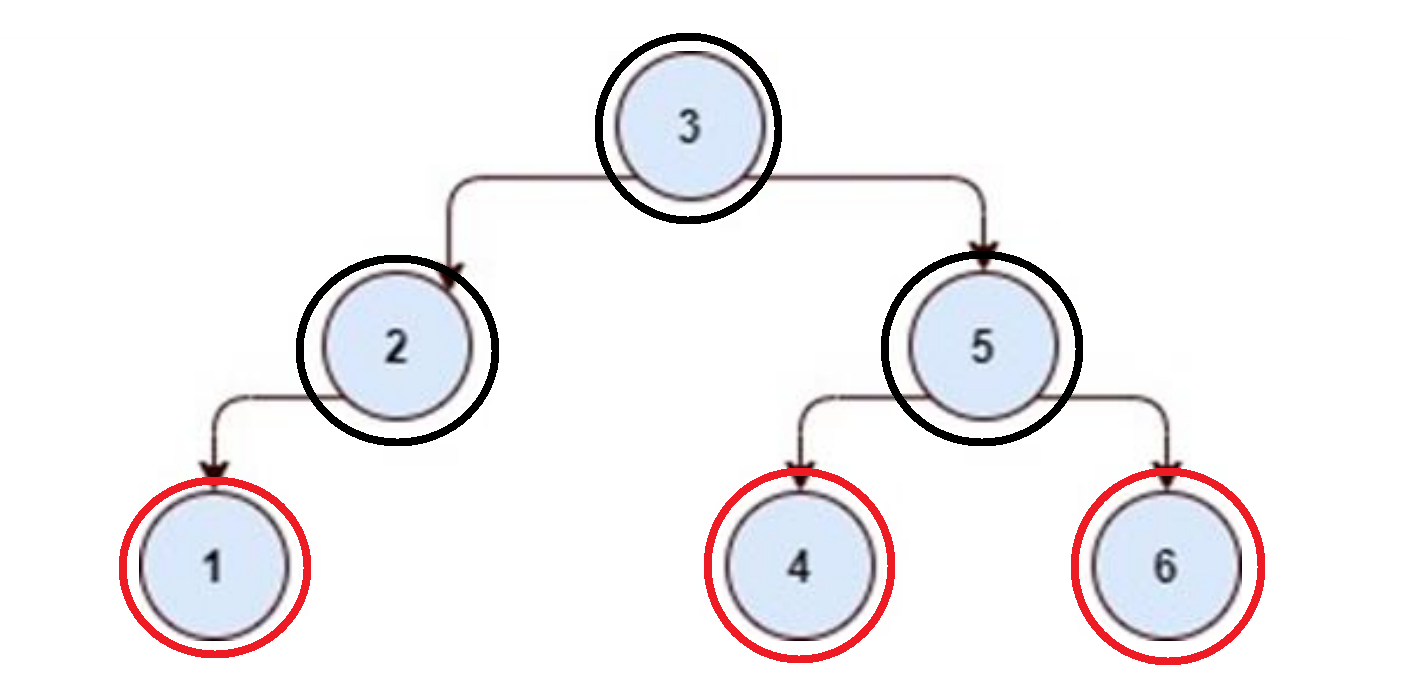
گره ۲ را در دو حالتی که قرمز و سیاه باشد بررسی می‌کنیم.

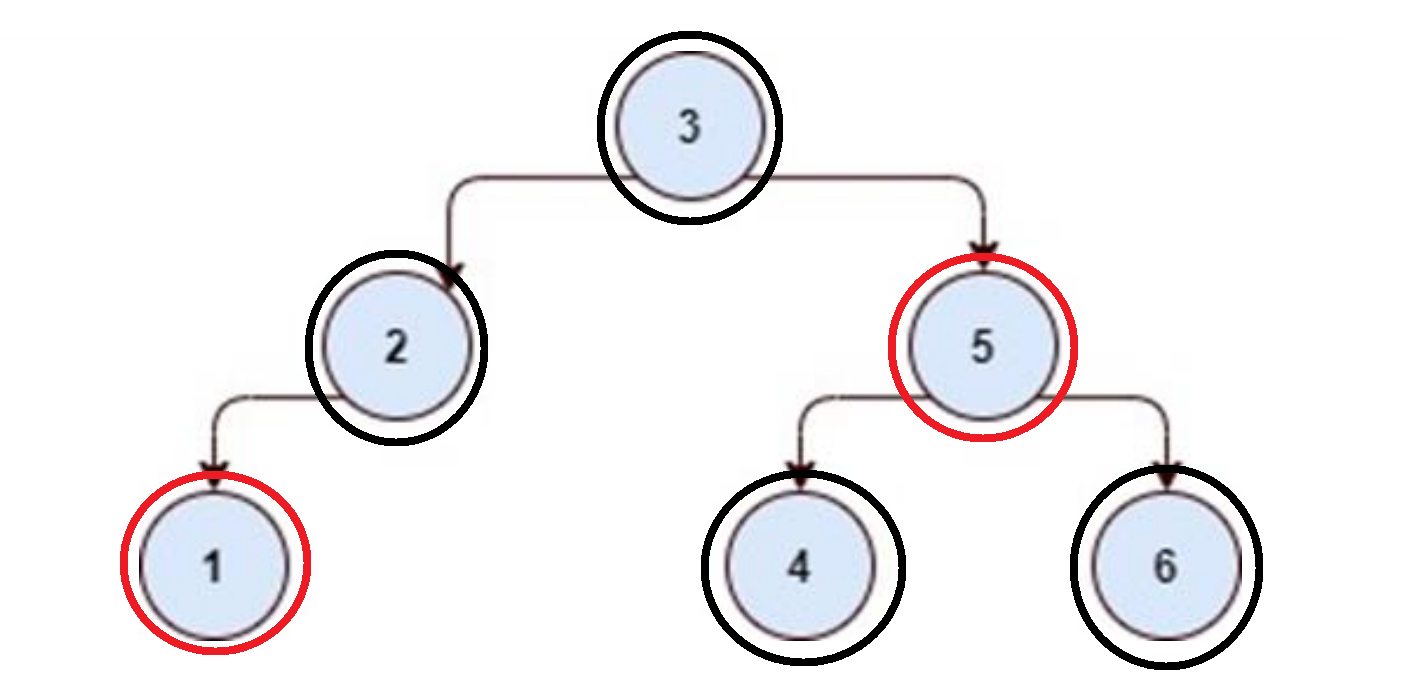
اگر قرمز باشد، مسیر مشخص شده (که یک مسیر root-NIL است)، دارای یک گره قرمز خواهد شد، پس بقیه‌ی مسیرهای ریشه تا نال هم باید تنها یک گره مشکی داشته باشند که ممکن نیست، برای مثال در همین مسیر ۳-۲-۱ ، مجبور می‌شویم گره ۱ را هم قرمز کنیم که آن‌گاه گره ۲ و فرزندش ۱ هر دو قرمز می‌شوند که غیرمجاز است، پس این حالت رد می‌شود.

اگر ۲ سیاه باشد، مسیرِ مشخص شده دارای دو گره سیاه خواهد بود و همه‌ی گرههای ریشه تا نال باید دو گره سیاه داشته باشند. پس گره ۱ را هم قرمز می‌کنیم که مسیر ۳-۲-۱ دارای دو گره سیاه باشد. تا این‌جا درختمان به این شکل در آمد که تنها حالتِ مجاز برای خانه‌های رنگ شده است:

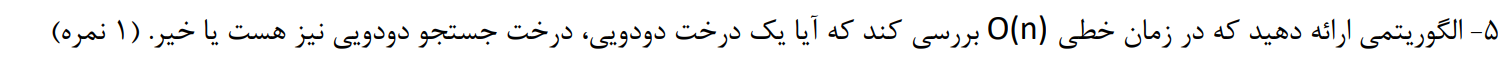


دو مسیر ریشه تا نالِ دیگر داریم ۳-۵-۶ و ۳-۵-۴ که هرکدام یک خانه‌ی مشکی‌اش تثبیت شده، و نیاز به یک خانه‌ی مشکیِ دیگر دارد. می‌توانیم ۵ را قرمز کنیم و بقیه را مشکی، و همچنین می‌توانیم پنج را مشکی کنیم و بقیه را قرمز.



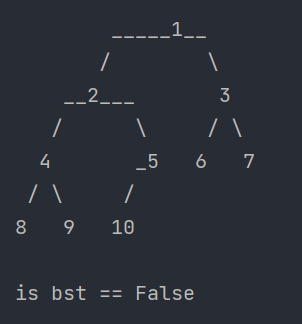


با توجه به این که از ابتدا حالت‌های غیرمجاز را حذف کردیم و حالت‌های مختلف برای انتخاب‌های مجاز را بررسی کردیم، حالتِ‌ مجاز دیگری غیر از این دو وجود ندارد.



*from* typing *import* Optional  
*from* binarytree *import* build, Node  
  
  
*def* is\_bst(tree: Optional[Node], min\_val, max\_val):  
  
 *if* tree *is None*:  
 *return True  
  
 if* tree.val < min\_val *or* max\_val < tree.val:  
 *return False  
  
 return* is\_bst(tree.left, min\_val, tree.val-1) *and* is\_bst(tree.right, tree.val+1, max\_val)  
  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
  
 list\_of\_nodes = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
 my\_tree = build(list\_of\_nodes)  
  
 my\_tree.pprint()  
 print('is bst == ' + str(is\_bst(my\_tree, -10000, +10000)))

برای توضیح الگوریتم‌مان، آن را با مثالی مانندِ درختِ زیر توضیح می‌دهیم:



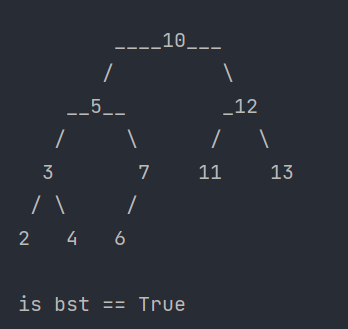
در این الگوریتم، هر گره یک بار با ماکسیمم و مینیمم مقدارِ مجازی که می‌تواند داشته باشد مقایسه می‌شود. در صورتی که در این میان یکی از اعضا بزرگ‌تر یا کوچک‌تر از حد مجاز باشد، الگوریتم به پایان می‌رسد، و اگر آخرین عضو غیرمجاز باشد یا درختمان ددج باشد، تمام اعضا چک می‌شوند که همان O(n) زمان می‌گیرد.

ابتدا از ریشه شروع می‌کنیم، می‌دانیم که عملا محدودیتی برای مقدارِ این گره وجود ندارد، پس حداکثر و حداقل مقدار مجاز برای تایپِ مقادیرِ dataی هر گره را (در این‌جا برای وضوح منفی و مثبت ده‌هزار در نظر گرفته شده است) را به همراه ریشه، به تابع‌مان می‌فرستیم.

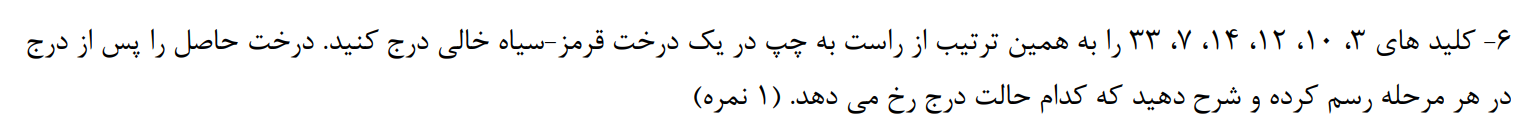
تابع اول از همه بررسی می‌کند که گرهِ فرستاده شده، نیل هست یا خیر، تا اگر در یک مسیر به ریشه رسید، دیگر ادامه ندهد. سپس مقدار گره‌مان را با مقادیر مجاز مقایسه می‌کند، که برای ریشه در این مرحله مشکلی نخواهیم داشت.

حال تابع‌مان را دو بارِ دیگر صدا می‌کنیم تا نوادگان سمت راست و چپِ گرهِ فعالی را بررسی کند. مقدار مجاز برای گره سمت راست، اعداد بزرگ‌تر از مقدار فعلی‌ست تا بزرگ‌ترین مقدار ممکن یعنی ۲ تا ۱۰۰۰۰، و مقدار مجاز برای گره سمت چپ، اعدادِ کوچک‌تر از گره فعلی‌ست تا کوچک‌ترین مقدار ممکن یعنی ۱۰۰۰۰- تا ۰ ، که ۲ خارج از این محدوده است پس تابع‌مان False را برمی‌گرداند.

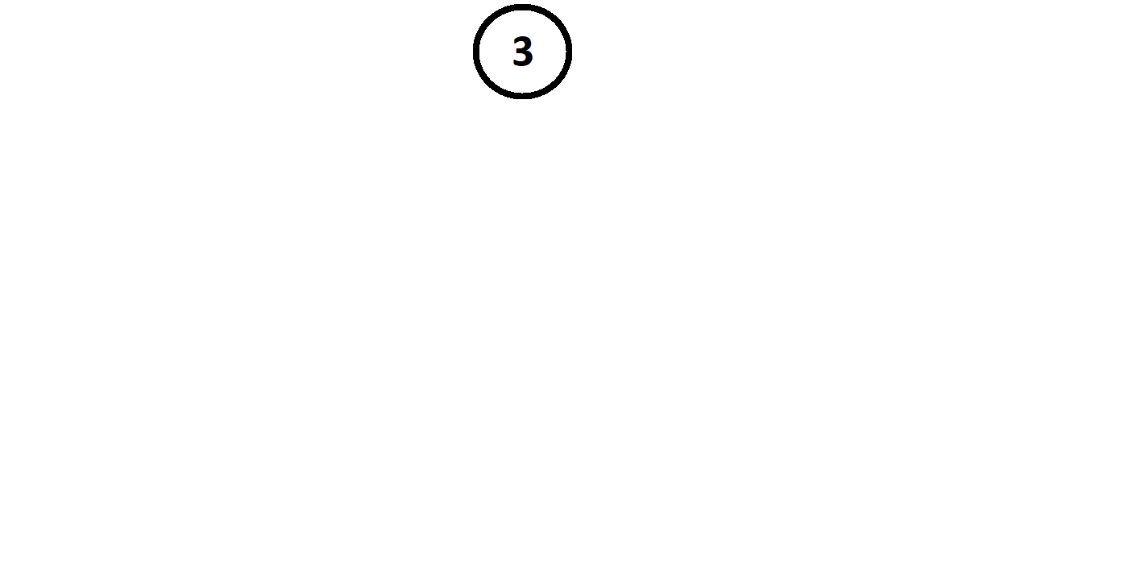
برای مثال زیر هم:



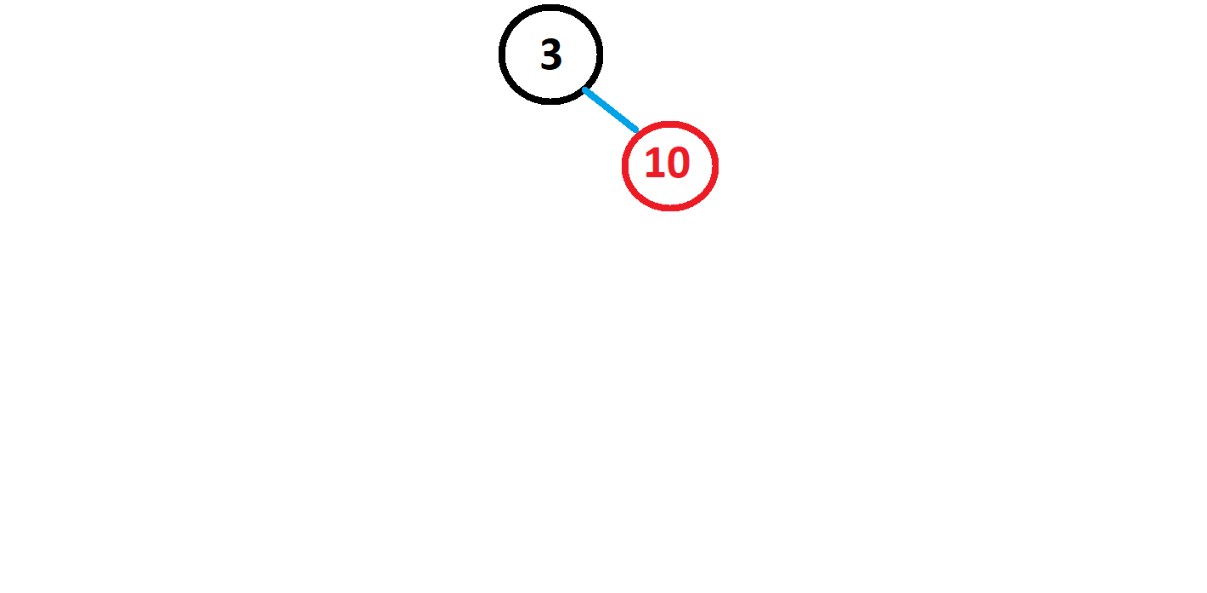
مقدار ۱۰ باید بین ۱۰۰۰۰ تا ۱۰۰۰۰- باشد که هست. مقدار ۵ باید بین ۱۰۰۰۰- تا ۱۰ باشد که هست، ۳ بین ۱۰۰۰۰- تا ۵ ، ۲ بین ۱۰۰۰۰- تا ۳، ۴ بین ۳ و ۵ باشد، ۷ بین ۵ و ۱۰ باشد، ۶ بین ۵ و ۷ باشد، ۱۲ بین ۱۰ و ۱۰۰۰۰ باشد، ۱۳ بین ۱۲ و ۱۰۰۰۰ باشد و ۱۱ هم بین ۱۰ و ۱۲ باشد که همگی هستند پس درخت‌مان ددج است.



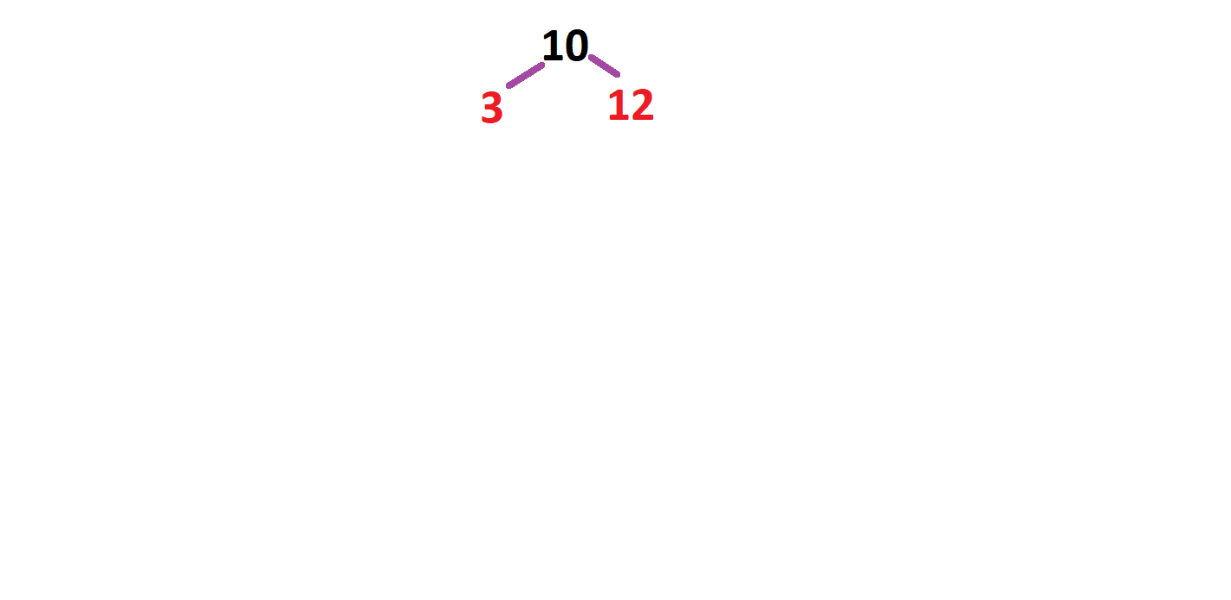
ابتدا ۳ را به عنوان ریشه‌ی سیاه اضافه می‌کنیم:



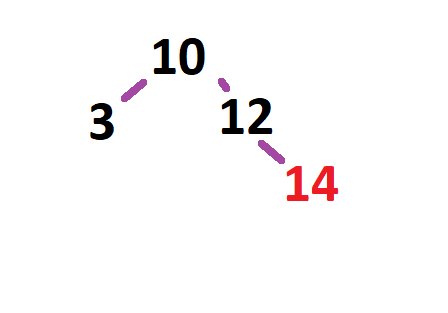
حال ۱۰ را اضافه می‌کنیم که به فرزندیِ راست ۳ می‌رود و چون پدرش سیاه است، به راحتی آن را قرمز می‌کنیم و تمام.



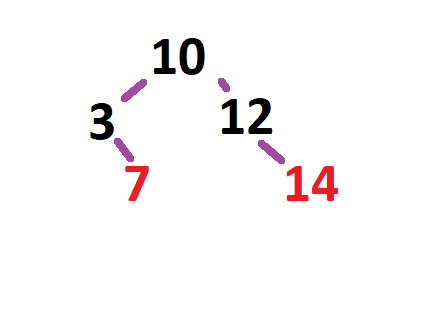
نوبت به ۱۲ می‌رسد که به فرزندیِ راست ۱۰ می‌رود. چون پدرش قرمز است اوضاع کمی پیچیده می‌شود. به سراغ عموی ۱۰ می‌رویم، یعنی فرزندِ دیگرِ‌ ۳ که برابر با نال است، پس مشابه کیس۲ عمل می‌کنیم و پدر یعنی ۱۰ را rotate می‌کنیم تا به جای پدربزرگ برود و درخت‌مان به این شکل می‌شود:



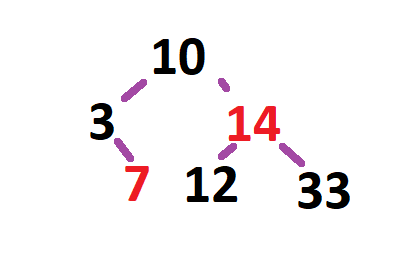
حالا چهارده را اضافه می‌کنیم که به فرزندیِ راستِ ۱۲ می‌رود. پدرش یعنی ۱۲ قرمز است، پس عمو را بررسی می‌کنیم که آن هم قرمز است، پس مطابق کیس۱ عمل می‌کنیم و پدر و عمو را مشکی می‌کنیم. از آن‌جا که پدر‌بزرگ که ۱۰ باشد ریشه است، آن را همان مشکی رها می‌کنیم و از آن‌جایی که ریشه روی همه‌ی مسیرها تاثیر یک‌سان دارد، مشکلی نخواهیم داشت.

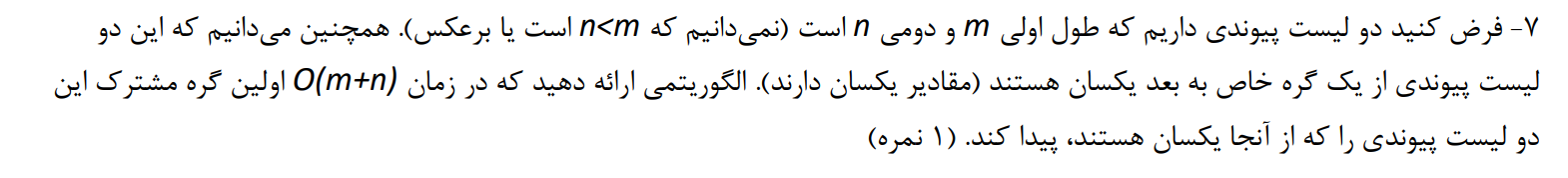


نوبت به هفت می‌رود که به فرزندیِ راست ۳ می‌رود. چون پدرش سیاه است، آن را قرمز اضافه می‌کنیم.



نوبت به آخرین عنصر یعنی ۳۳ می‌رد که باید به فرزندیِ راست ۱۴ برود. چهارده پدر قرمز است، پس عمو را بررسی می‌کنیم که نیل است، پس مطابق کیس دو عمل می‌کنیم و چهارده و دوازده را دوران می‌دهیم و فرزند جدید را مشکی اضافه می‌کنیم. با توجه به بقیه‌ی اعضا، می‌بینیم که نیازی به رنگ کردنِ دوباره هم نیست و همه‌ی مسیرهای ریشه-نیل هم دقیقا دو گره سیاه دارند.





در این الگوریتم از دو اشاره‌گر استفاده می‌کنیم که هرکدام به اولین عضوِ یکی از لیست‌ها اشاره می‌کند، هرگاه این دو اشاره‌گر به یک گره یک‌سان اشاره بکنند، یعنی دو لیست پیوندی‌مان به یکدیگر رسیده‌اند.

ابتدا تفاضل طولِ دو لیست را محاسبه می‌کنیم، سپس در لیستِ طولانی‌تر با اشاره‌گرمان به اندازه‌ی تفاضل طولِ دو لیست پیش می‌رویم، حال از هر دو لیست،‌ به یک اندازه گره باقی مانده‌ست. همزمان روی هردو لیست یکی یکی پیش می‌رویم و در هر مرحله پوینتری که روی روی این دو لیست داریم را با یکدیگر مقایسه می‌کنیم، اگر برابر نبودند هر کدام را یک گره پیش می‌بریم، و اولین باری که برابر شدند، هردو به اولین گره مشترک اشاره می‌کنند.

بدترین حالت برای این الگوریتم، زمانی‌ست که آخرین اعضای این دو لیست با یکدیگر برابرند که در این حالت الگوریتم‌مان O(m+n) زمان خواهد گرفت.

اگر منظور از این که نمی‌دانیم m<n یا برعکس، این است که توانایی یا اجازه‌ی مقایسه کردن و محاسبه‌ی آن را نداریم، می‌توانیم از الگوریتم زیر استفاده کنیم:

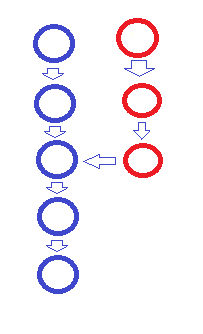
یک دور روی تمام عناصر یکی از لیست‌ها پیمایش می‌کنیم (طول این لیست را m در نظر می‌گیریم) و تمام این اعضا را در یک هش ذخیره می‌کنیم.

می‌دانیم که ذخیره کردن هر عضو در هش O(1) زمان می‌گیرد، پس کل این مرحله O(m) زمان می‌گیرد.

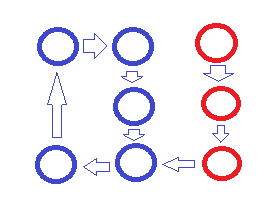
سپس سراغ لیست دوم می‌رویم و یکی یکی عناصر را می‌پیماییم و چک می‌کنیم که در هش هست یا نه، اولین عنصر از لیست دوم که در هش هم بود، جوابِ ماست. می‌دانیم که چک کردنِ این که هر عضو در هش هست یا نیست O(1) زمان می‌گیرد، در نتیجه بدترین حالت زمانی خواهد بود که عضو مشترک، آخرین عضو از دو لیست باشد، در این صورت باید تمام عناصر لیست دوم را هم تست کنیم که آن‌گاه O(n) زمان خواهد گرفت.

یک راه حل جالب دیگر:

دو لیستِ ما چنین شکلی دارند:



سراغ لیست اول می‌رویم (لیست‌ آبی‌رنگ به طول m ). آدرس عضو اول از این لیست را ذخیره می‌کنیم و سپس تا آخرین عضو پیش می‌رویم، سپس next آخرین عضو را برابر با اولین عضو قرار می‌دهیم، یعنی اولین لینکدلیستمان را تبدیل به یک حلقه کردیم، به این شکل:

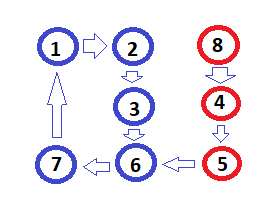


می‌دانیم که تعداد گره‌های این حلقه، برابر است با همان m یعنی طول لیست اول.

مرحله‌ای که توضیح داده شد، مشخصا به اندازه‌ی طول لیستمان یعنی O(m) زمان می‌گیرد.

حال سراغ لیست دوم (که اعضای متمایزِ آن با رنگ قرمز مشخص شده‌اند) می‌رویم.

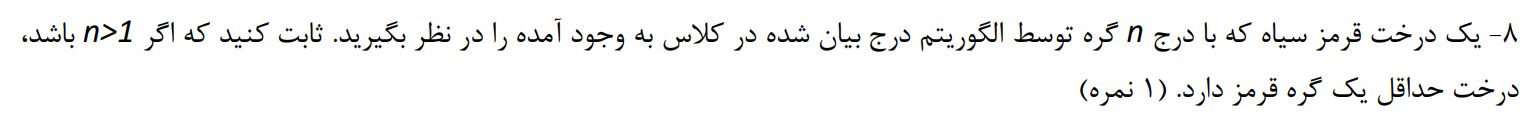
برای واضح بودن توضیحات، اعدادی بی‌معنا را به گره‌ها نسبت می‌دهیم.



سراغ پوینتری می‌رویم که به اولین گره از دومین لیست داریم، یعنی به گره ۸ ، آدرسِ این گره را ذخیره می‌کنیم، سپس به اندازه‌ی طول لیست اول، یعنی m ، جلو می‌رویم، پس به گره ۱ می‌رسیم.

در این جا دو پوینتر داریم، یکی به عضو اول از لیست دوم اشاره می‌کند و دیگری به m عضو بعد از عضوِ اول از لیست دوم.

حال هم‌زمان یکی یکی این دو اشاره‌گر را پیش می‌بریم و بعد از هر پیش‌روی با یکدیگر مقایسه‌شان می‌کنیم و اگر به یک گره اشاره نمی‌کردند، یک گرهِ دیگر پیش می‌بریم هردو را. اولین باری که به یک گره مشترک اشاره می‌کردند، آن گره، شروع اشتراکِ این دو لیست است. این مرحله هم در بدترین حالت تا آخرین عنصر لیست دوم (که طول آن برابر n بود) طول می‌کشد، یعنی O(n) زمان می‌گیرد.



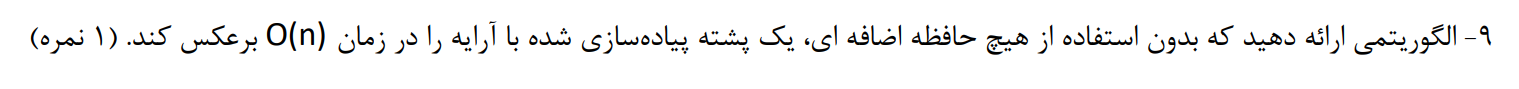
اولین عضو به عنوان ریشه‌ی سیاه درج خواهد شد. دومین عضو قطعا یکی از فرزندانِ این ریشه‌ی سیاه خواهد بود و طبق الگوریتمی که در کلاس بیان شد، هرگاه پدر سیاه بود، فرزند را قرمز می‌کنیم.

حال برای این که مطلوب را ثابت کنیم، کافی‌ست نشان دهیم که هیچ درجی نمی‌تواند این فرزندِ‌ قرمز را سیاه کند، بدونِ این که یک گره قرمز دیگر اضافه کند.

تغییر رنگ این فرزندِ قرمز، هنگامی رخ می‌دهد که یک فرزند به آن اضافه کنیم. اگر تا آن موقع یک برادرِ قرمز به این فرزند قرمز اضافه شده باشد، این دو برابر را مشکی می‌کنیم و فرزندِ جدید را قرمز می‌کنیم، پس در این کیس همچنان حداقل یک گره قرمز، که همان فرزندِ جدید باشد، داریم همچنان.

حالتِ دیگر این است که هنگامی که فرزندی به این فرزندِ قرمز اضافه می‌کنیم، عموی‌ش مشکی یا نیل باشد. در این حالت هم می‌دانیم که بعد از روتیت و درج، گره‌ای که اضافه می‌شود، قرمز اضافه می‌شود.

به این ترتیب تمام حالاتی که برای تغییر رنگ گره قرمزمان داشتیم را بررسی کردیم و دیدیم که در تمام این حالات، حداقل یک گره قرمز دیگر اضافه می‌شود، پس همواره گره قرمز خواهیم داشت.



با توجه به این که داده‌های ما در یک آرایه ریخته شده‌اند، پس به تمام اعضا دسترسی داریم، در نتیجه، دو شمارنده در نظر می‌گیریم، یکی را i می‌نامیم و آن را برابر با صفر قرار می‌دهیم و دیگری را j می‌نامیم و برابر با ایندکسِ آخرین عضو یا n – 1 (با این فرض که طول آرایه‌مان همان n است)، سپس مقادیر عضو i ام و عضو j ام را با یکدیگر جابه‌جا می‌کنیم، و یک واحد به i اضافه می‌کنیم و یک واحد از j کم می‌کنیم و سپس باز هم به همین منوال ادامه می‌دهیم، عضو i ام را با j ام جابه‌جا می‌کنیم و دوباره یکی از i کم می‌کنیم و یکی به j اضافه می‌کنیم. همین حلقه را ادامه می‌دهیم تا هنگامی که i کوچک از j باشد.

اگر هم فرض کنیم هدفمان برعکس کردنِ پشته‌ای باشد که با لینکدلیست پیاده‌سازی شده است، یک اشاره‌گر به عضو اول می‌سازیم، یک اشاره‌گرِ موقت هم می‌سازیم، اشاره‌گر اصلی را به گرهِ بعدی می‌فرستیم و اشاره‌گر موقتِ دیگری را به این گره می‌سازیم. دوباره اشاره‌گر اصلی را یک گره جلوتر می‌رویم، سپس next اشاره‌گر دوم‌مان را به اشاره‌گر اول وصل می‌کنیم. این کار را تا جایی ادامه می‌دهیم که اشاره‌گر اصلی‌مان به نیل برسد، در این حالت اشاره‌گر موقت دوم به عضو آخر اشاره می‌کند، و اشاره‌گر موقت اولمان به عضو یکی مانده به آخر اشاره می‌کند و next عضو آخر را برابر با عضو یکی مانده به آخر قرار می‌دهیم، با این کار تمام اشاره‌گر‌هایمان به عضو‌های قبلی خود اشاره می‌کنند، یعنی لیست‌مان برعکس شده.

به بیانی، نیاز به سه اشاره‌گر داریم که به سه گرهِ پیاپی اشاره می‌کنند. گرهی که جلوتر است، یکی یکی جلو می‌رود و به کمکِ آن گره‌های قبلی را مقداردهی می‌کنیم و دو گرهِ قبلی هم، که اشاره‌گری از عنصر عقب‌تر به عنصر جلوتر اشاره می‌کند، در هر مرحله اشاره‌گرشان را برعکس می‌کنیم.

*class* Node:  
 *def* \_\_init\_\_(*self*, val):  
 *self*.val = val  
 *self*.next = *None  
  
  
def* reverse(linked\_list: Node) -> Node:  
 main\_pointer = linked\_list  
 prev\_prev\_main = main\_pointer  
  
 main\_pointer = main\_pointer.next  
 prev\_main = main\_pointer  
  
 main\_pointer = main\_pointer.next  
  
 prev\_prev\_main.next = *None  
  
 while* main\_pointer *is not None*:  
 prev\_main.next = prev\_prev\_main  
  
 prev\_prev\_main = prev\_main  
 prev\_main = main\_pointer  
  
 main\_pointer = main\_pointer.next  
  
 prev\_main.next = prev\_prev\_main  
 *return* prev\_main  
  
  
*def* print\_linked\_list(linked\_list: Node):  
 pointer = linked\_list  
  
 print('\nLinked List: ', end='')  
 *while* pointer *is not None*:  
 print(pointer.val, end=' ')  
 pointer = pointer.next  
  
 print()  
  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 linked\_list = Node(1)  
 linked\_list.next = Node(2)  
 linked\_list.next.next = Node(3)  
 linked\_list.next.next.next = Node(4)  
 linked\_list.next.next.next.next = Node(5)  
  
 print\_linked\_list(linked\_list)  
 linked\_list = reverse(linked\_list)  
 print\_linked\_list(linked\_list)

خروجی این کد به این شکل است:

