تحلیل پیچیدگی زمانی راه حل ۲:

$$T(n) = \sum_{h=0}^{\log n} h \times 2^{(\log n)-h}$$

چون $h=0$ منظور سطح آخر است و به سمت ریشه حرکت می‌کنیم (ریشه $h=\log n$) و هر سطح h نیز $\log n - h$ نود داریم.

$$T(n) = \sum_{h=0}^{\log n} h \times 2^{(\log n)-h}$$

$$= \sum_{h=0}^{\log n} h * \frac{n}{2^h} = n \sum_{h=0}^{\log n} \frac{h}{2^h} = n * A^*$$

ادعا می‌کنیم که A^* یک عدد ثابت است و در نتیجه $T(n)=O(n)$ می‌باشد.

یادآوری: اگر $|a|<1$ باشد آنگاه $\sum_{k=0}^{\infty} a^k = \frac{1}{1-a}$

$$A^* \leq \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \cdots$$

می‌دانیم که: $\frac{1}{2} + (\frac{1}{2})^2 + (\frac{1}{2})^3 + (\frac{1}{2})^4 + \cdots = 1$

Sum:

$$(\tfrac{1}{2})^2 + (\tfrac{1}{2})^3 + (\tfrac{1}{2})^4 + \cdots = 1 - \tfrac{1}{2} = \tfrac{1}{2}$$

$$(\tfrac{1}{2})^3 + (\tfrac{1}{2})^4 + \cdots = \tfrac{1}{2} - \tfrac{1}{4} = \tfrac{1}{4}$$

$$A^* = \sum_{k=0}^{\infty} (\tfrac{1}{2})^k = 2$$

$$T(n) = 2n = O(n)$$

نکته مهم: ایده راه حل دوم این بود که برای هر عنصر، $\log n$ هزینه نکنیم، به عبارت دیگر:

عنصر 1 $\longrightarrow \log n$

عنصر 2 $\longrightarrow (\log n) - 1$

عنصر 4 $\longrightarrow (\log n) - 2$

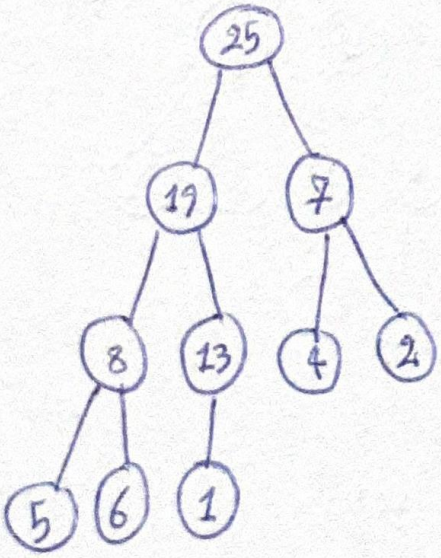
⋮

عمل افزایش کلید یک عنصر:

Increas(x, key)

در این عملیات می‌خواهیم اگر گره x کلید key مقداری کمتر از x دارد، مقدار آن را به x افزایش دهیم.

مثال: درخت MaxHeap زیر را در نظر بگیرید:



① فراخوانی Increase(10, 8) چه تغییری در درخت MaxHeap پالا ایجاد می‌کند؟

② فراخوانی Increase(7, 8) چه تغییری ایجاد می‌کند؟

③ فراخوانی Increse(27, 8) چه تغییری ایجاد می‌کند؟

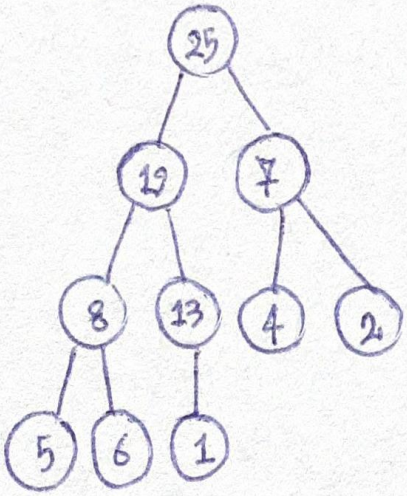
جمع بندی: متناسب با مقدار x از نود حاوی کلید key کاری که می‌بایست ساختار MaxHeap را به‌روزرسانی کنیم. پیچیدگی این عملیات از مرتبه $O(\log n)$ است، جایی که n تعداد نودهای درخت MaxHeap است.

عملیات درج عنصر در MaxHeap:

Insert (key)

در این عملیات می‌خواهیم عنصر جدید key را به نحوی به درخت MaxHeap اضافه کنیم که درخت حاصل همچنان MaxHeap بماند.

مثال: درخت Max Heap زیر را در نظر بگیرید:

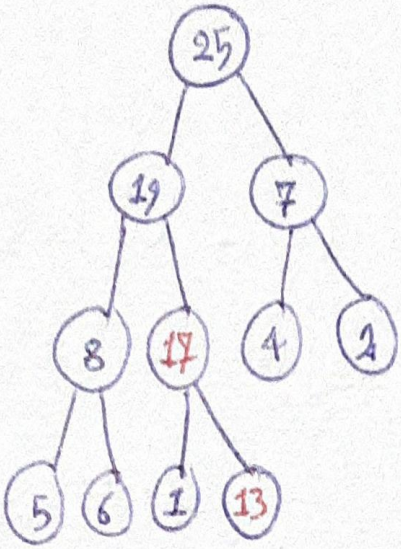


① فراخوانی Insert (10) چه تغییری در درخت Max Heap بالا ایجاد می‌کند؟

ایده: یک نود با کلید ۱۰ در سطح آخر و به عنوان سمت راست‌ترین نود ایجاد می‌کنیم و اشاره‌گر پدر آن را به طور مناسبی بروز می‌کنیم.

سپس با فراخوانی Increase (17, 10)، درخت MaxHeap نهایی ساخته می‌شود.

درخت حاصل از Insert(17) به صورت زیر بدست می آید:

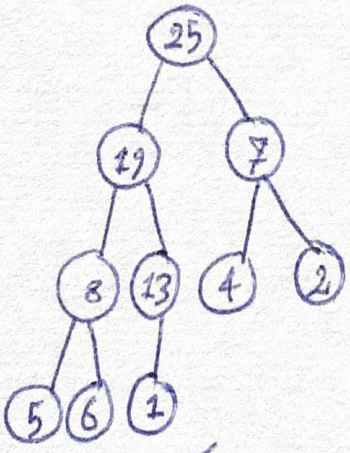


عملیات حذف عنصر Max از درخت Max Heap:

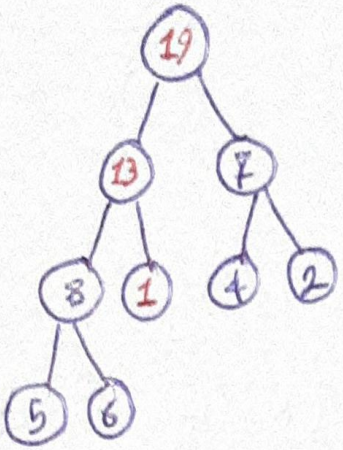
Delete-Max(T)

در این عملیات می خواهیم عنصر بیشینه را به نوعی حذف کنیم که درخت حاصل همچنان Max Heap بماند.

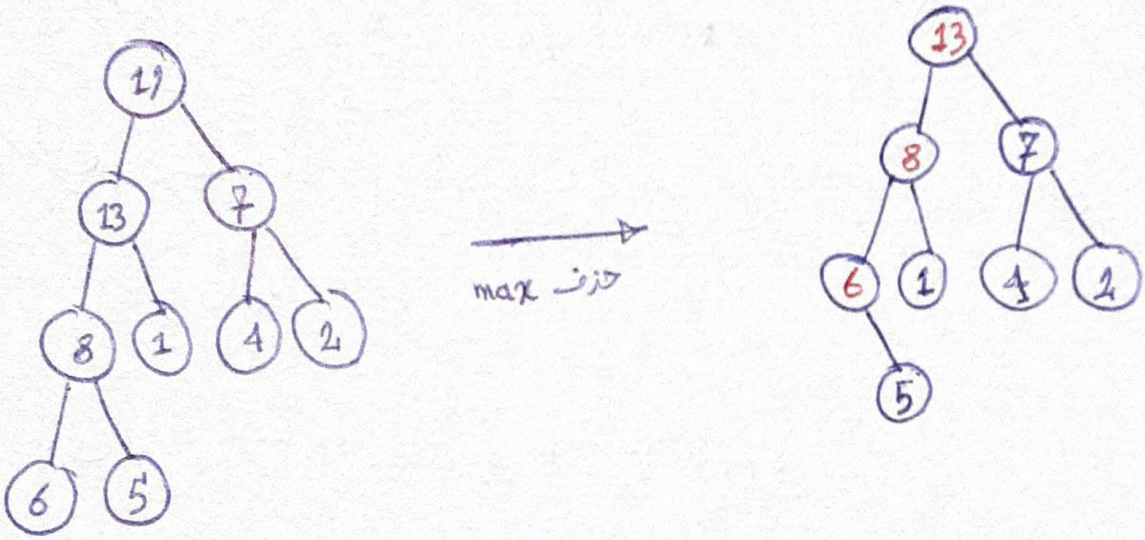
مثال: درخت Max Heap زیر را در نظر بگیرید:



راه حل پیشنهادی!: ابتدا ریشه را حذف می کنیم و سپس از ریشه به سمت برگ ها حرکت کرده و در صورت نیاز برای حفظ درخت Max Heap بودن، عنصر ماکزیمم را به سطح بالاتر انتقال می دهیم.



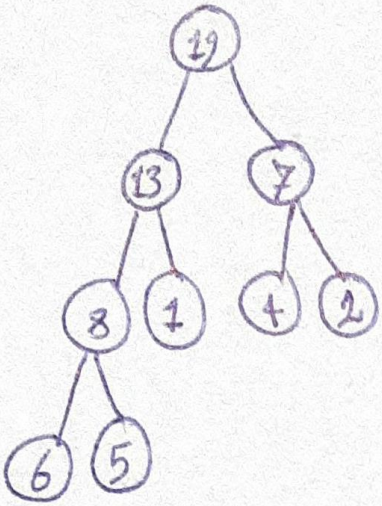
نکته: راه حل پیشنهادی! ضمانتی برای درخت حریصانه کامل بودن با رعایت پر بودن از سمت چپ به راست نودها را ندارد به مثال زیر دقت کنید:



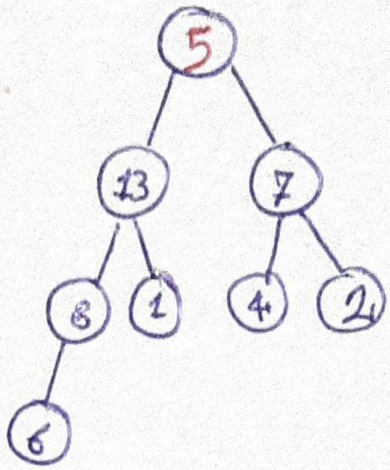
به وضوح درخت بالا، درخت Max Heap هست ولی اگر ساختار Max Heap را رعایت نکرده است.

راه حل ۲: ابتدا ریشه را حذف کرده و با سمت راست ترین برگ جایگزین می کنیم، سپس از ریشه تا برگ Max Heap بودن را بررسی و در صورت نیاز جابجایی های لازم را انجام می دهیم.

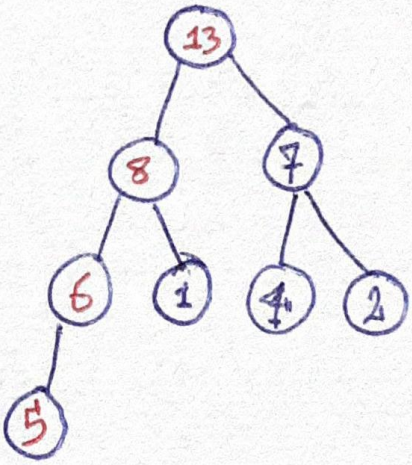
مثال: درخت Max Heap زیر را درنظر بگیرید:



گام ۱: حذف ریشه و جایگزینی کلید راست‌ترین برگ به عنوان ریشه



گام ۲: Max Heap کردن درخت حاصل از گام ۱



نکته: پیچیدگی زمانی الگوریتم فوق برابر با ارتفاع درخت از مرتبه $O(\log n)$ است.

پیاده سازی درخت Max Heap:

با توجه به خصیصه‌های ساختاری Max Heap (تقریباً کامل بودن و پر شدن از چپ به راست برگ‌ها)، لیست یک گزینه بسیار مناسب برای پیاده سازی درخت Max Heap است. به عنوان مثال برای هر نود در اندیس i لیست، فرزندان آن در اندیس‌های $2i$ و $2i+1$ قرار می‌گیرند و هدر رفت حافظه‌ای هم اتفاق نمی‌افتد.