درس ساختمان داده‌ها
دانشکده ریاضی. دانشگاه صنعتی خواجه نصیرالدین طوسی — بهار ۹۹
تدریس توسط: حسین جوهری

صف اولویت

۱ ساختار داده صف اولویت

از صف اولویت برای ذخیره‌سازی عناصری شامل دو قسمت اصلی کلید و قسمت داده استفاده می‌شود. عناصر بر اساس قسمت کلید بطور ضمنی در یک صف مرتب شده‌اند. عنصر با بیشترین کلید (بیشترین اولویت) در جلوی صف قرار گرفته است. البته در این ساختار داده (بر خلاف BST) وجود عناصر با کلیدهای تکراری مجاز است. در این حالت یکی از عناصری که کلید بیشینه دارد در جلو صف قرار گرفته است. ساختار داده صف اولویت اعمال اصلی زیر را پشتیبانی می‌کند .

۱.۱ اعمال اصلی ویژه صف اولویت

- add(k,v) : اضافه کردن یک عنصر با کلید k و داده v
- max() : برگرداندن یک عنصر با کلید بیشینه
- remove_max(): حذف یک عنصر با کلید بیشینه

Operation	Return Value	Priority Queue
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B), (5,A), (9,C)}
P.add(7,D)		{(3,B), (5,A), (7,D), (9,C)}
P.**max**()	(9,C)	{(3,B), (5,A), (7,D), (9,C)}
P.remove_**max**()	(9,C)	{(3,B), (5,A), (7,D)}
P.remove_**max**()	(7,D)	{(3,B), (5,A)}
len(P)	2	{(3,B), (5,A)}

۲.۱ پیاده‌سازی صف اولویت با آرایه نامرتب

می‌توانیم عناصر را با همان ترتیبی که وارد شده‌اند در یک آرایه ذخیره کنیم. مثلا عنصری که تازه وارد شده به انتهای آرایه اضافه شود. در این حالت آرایه نامرتب خواهد بود و لذا پیدا کردن عنصر با کلید بیشینه (و به تبع آن حذف عنصر با کلید بیشینه) زمانبر خواهد بود. جدول زیر زمان اجرای اعمال اصلی صف اولویت در این شیوه پیاده‌سازی را نشان می‌دهد.

Operation	Running Time
len	$O(1)$
is_empty	$O(1)$
add	$O(1)$
max	$O(n)$
remove_max	$O(n)$

۳.۱ پیاده‌سازی صف اولویت با آرایه مرتب

اگر عناصر صف را در یک آرایه مرتب نگه‌داریم، هزینه پیدا کردن کلید بیشینه به $O(1)$ تقلیل می یابد. عناصر را با ترتیب صعودی با توجه به مقدار کلید در یک آرایه مرتب نگه می‌داریم. در این پیاده‌سازی در همه حال عنصر با کلید بیشینه در انتهای آرایه قرار می‌گیرد و لذا حذف آن نیز کم هزینه خواهد بود. اما این کار یک هزینه سنگین به عمل درج عنصر جدید وارد می‌کند. عنصر جدید باید در جای مناسب خود در آرایه مرتب شده وارد شود که می‌تواند منجر به شیفت و جابجایی تعداد زیادی از عناصر شود. جدول زیر زمان اجرای اعمال اصلی صف در این پیاده‌سازی را نشان می‌دهد.

Operation	Unsorted List	Sorted List
len	$O(1)$	$O(1)$
is_empty	$O(1)$	$O(1)$
add	$O(1)$	$O(n)$
max	$O(n)$	$O(1)$
remove_ max	$O(n)$	$O(1)$

۴.۱ پیاده‌سازی صف اولویت با استفاده از هرم بیشینه

دو پیاده‌سازی با استفاده از آرایه مرتب و آرایه نامرتب در واقع دو سر طیف یک راه حل را نشان می‌دهد. در یکی زمان اضافه کردن عنصر جدید $O(1)$ و زمان پیدا کردان کلید بیشینه $O(n)$ است در حالیکه در دیگری برعکس این برقرار است (زمان اضافه کردن $O(n)$ است و زمان پیدا کردن کلید بیشینه $O(1)$ است). در ادامه با راه حل دیگری آشنا می‌شویم که بطور همزمان هر سه عمل اصلی صف اولویت را در زمان $O(\log n)$ پیاده‌سازی می‌کند. در این راه حل، عناصر در یک ساختار درختی به نام هرم بیشینه (یا هرم کمینه) ذخیره می‌شوند.

۲ هرم بیشینه MAX-Heap

هرم بیشینه یا MAX-Heap یک درخت دودویی متوازن (تقریبا کامل) است. این درخت دارای ویژگی‌های زیر است:

- تعداد راسهایی که فقط یک فرزند سمت چپ دارد از ۲ کمتر است. بقیه راسهای غیربرگ دو فرزند چپ و راست دارند.
- برگها حداکثر در دو سطح هستند.
- برگهای سطح آخر از سمت چپ درخت پر شده‌اند. در واقع ساختار درخت برای هر مقدار n منحصر بفرد است.
- کلید هر راس از کلید فرزندان کوچکتر نیست. بنابرین ریشه بزرگترین عنصر در هرم بیشینه است. به همین ترتیب k امین بزرگترین عنصر نمی‌تواند در سطحی بیش از k قرار گیرد.
- هرم می‌تواند عناصری با کلید یکسان داشته باشد (برخلاف BST)
- ارتفاع یک هرم با n عنصر برابر است با $\lfloor \log n \rfloor$
- اعمال درج، حذف بزرگترین عنصر، افزایش و کاهش کلید یک عنصر در زمان $\Theta(\log n)$ قابل انجام است.

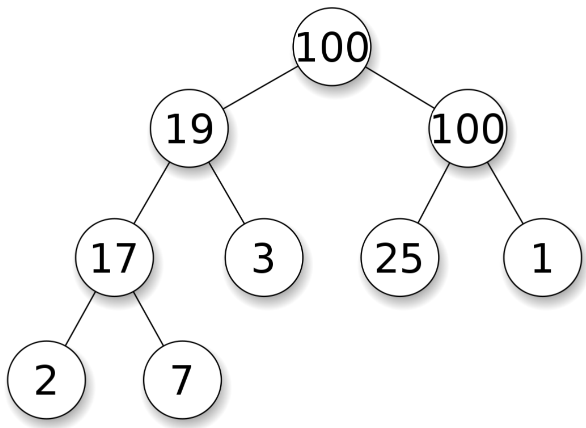


Figure ۱: یک هرم بیشینه با ۹ عنصر

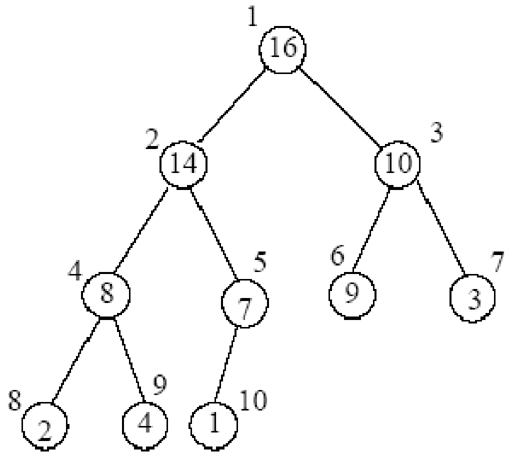
۱.۲ پیاده‌سازی هرم بیشینه با استفاده از آرایه

هرم بیشینه H با n عنصر را می‌توان در آرایه A به طول n ذخیره کرد. ریشه در اندیس $A[1]$ قرار می‌گیرد. فرزند چپ عنصر i ام در $A[2i]$ و فرزند راست در $A[2i+1]$ قرار می‌گیرد. با این توصیف، پدر عنصر iام در $A[\lfloor i/2 \rfloor]$ ذخیره می‌شود.

```
Parent(i)
  return i/2

LeftChild(i)
  return 2i

RightChild(i)
  return 2i+1
```



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Figure ۲: یک هرم بیشینه با ۱۰ عنصر و آرایه متناظر با آن

۲.۲ افزایش کلید یک راس در هرم بیشینه

فرض کنیم می‌خواهیم کلید یک راس را افزایش دهیم. کلید جدید را با کلید پدر مقایسه می‌کنیم. اگر کمتر یا مساوی بود تغییری در هرم ایجاد نمی‌کنیم درغیر این صورت جای پدر و فرزند را عوض می‌کنیم. مقایسه میان پدر و فرزند را ادامه می‌دهیم تا اینکه نیازی به تعویض جا نباشد. شبه کد جزئیات این الگوریتم را بیان می‌کند. در الگوریتم زیر key مقدار کلید جدید برای عنصر iام هرم است.

```
MAX-Heap-Increase-Key(A,i,key)
if (key < A[i])
      print error: new key is smaller than the old key
else
      A[i] = key
      while( i>1 and A[i] > A[Parent(i)])
            swap(A[i],A[Parent(i)])
            i = Parent(i)
```

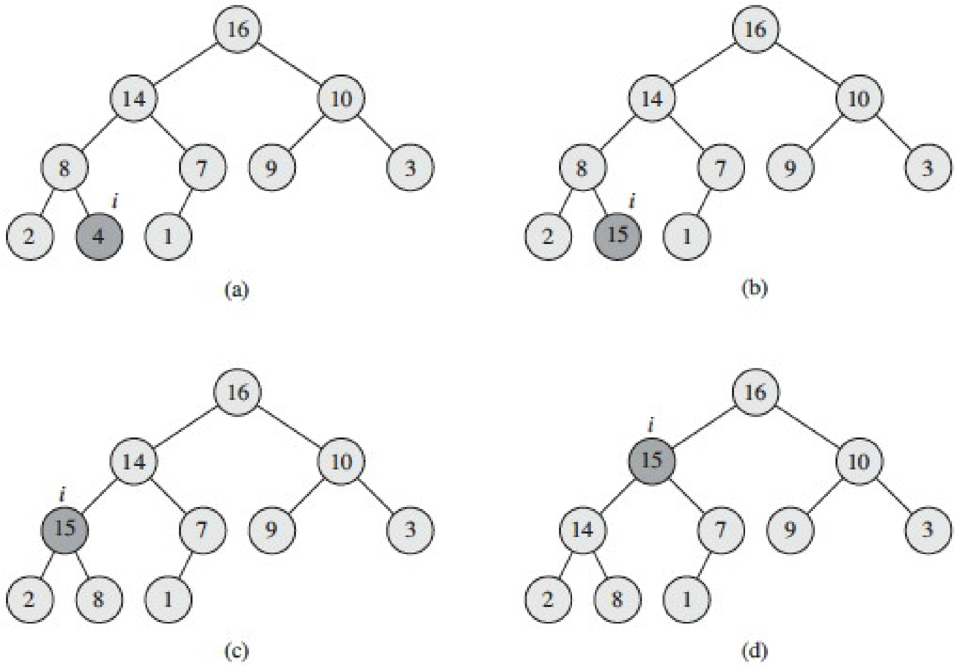


Figure ۳: کلید 4 به 15 افزایش پیدا کرده است و باعث جابجایی در هرم شده است

۳.۲ درج عنصر در MAX-Heap

برای پیاده‌سازی درج عنصر جدید از رویه افزایش کلید عنصر کمک می‌گیریم. کلید عنصر جدید را در $A[n+1]$ قرار می‌دهیم. این مثل افزایش کلید عنصر $n+1$ ام است (فرض کنید مقدار قبلی‌اش $-\infty$ بوده است.) حال رویه افزایش کلید عنصر را انجام می‌دهیم.

```
MAX-Heap-Insert(A,x)
Increase the length of A by 1  # Len(A) = Len(A)+1
A[Len(A)] = - infinity
MAX-Heap-Increase-Key(A, Len(A), key)
```

۴.۲ ساخت هرم بیشینه

فرض کنید آرایه‌ای نامرتب A با n عنصر (قابل مقایسه) داریم که می‌خواهیم تبدیل به یک هرم بیشینه‌اش کنیم. برای انجام این کار می‌توانیم یک هرم بیشینه تهی ایجاد کنیم و n عنصر آرایه را به ترتیب در هرم بیشینه درج کنیم. اما این رویه می‌تواند کند باشد (در بدترین حالت به $\Theta(n \log n)$ مقایسه نیاز دارد). رویه‌ای سریعتر برای تبدیل یک آرایه به هرم بیشینه وجود دارد که در بدترین حالت به $\Theta(n)$ مقایسه نیاز دارد. این رویه با نام Build-Heap در شبه کد زیر بیان شده است. رویه Build-Heap یک زیررویه اصلی به نام MAX-Heapify دارد که کار اصلی ساخت هرم را انجام می‌دهد. زیررویه MAX-Heapify(A،i) با فرض اینکه زیردرختهای عنصر iام آرایه A قبلا به هرم بیشینه تبدیل شده‌اند، زیردرخت با ریشه $A[i]$ را تبدیل به هرم بیشینه می‌کند. شکل زیر یک مثال کاربرد MAX-Heapify را نشان می‌دهد. برای انجام این کار کافی است فرزندان iامین عنصر (ریشه‌های دو زیردرخت عنصر iام، در مثال شکل 9 و 10) با $A[i]$ مقایسه شوند. در صورت لزوم باید جابجایی انجام شود و بزرگترین عنصر در این میان باید در ریشه قرار گیرد. جابجایی ممکن است یکی از هرمهای زیرین را از حالت بیشینه خارج سازد و در نتیجه MAX-Heapify باید روی هرم پایینتر اعمال شود تا زمانی که به جابجایی نیازی نباشد.

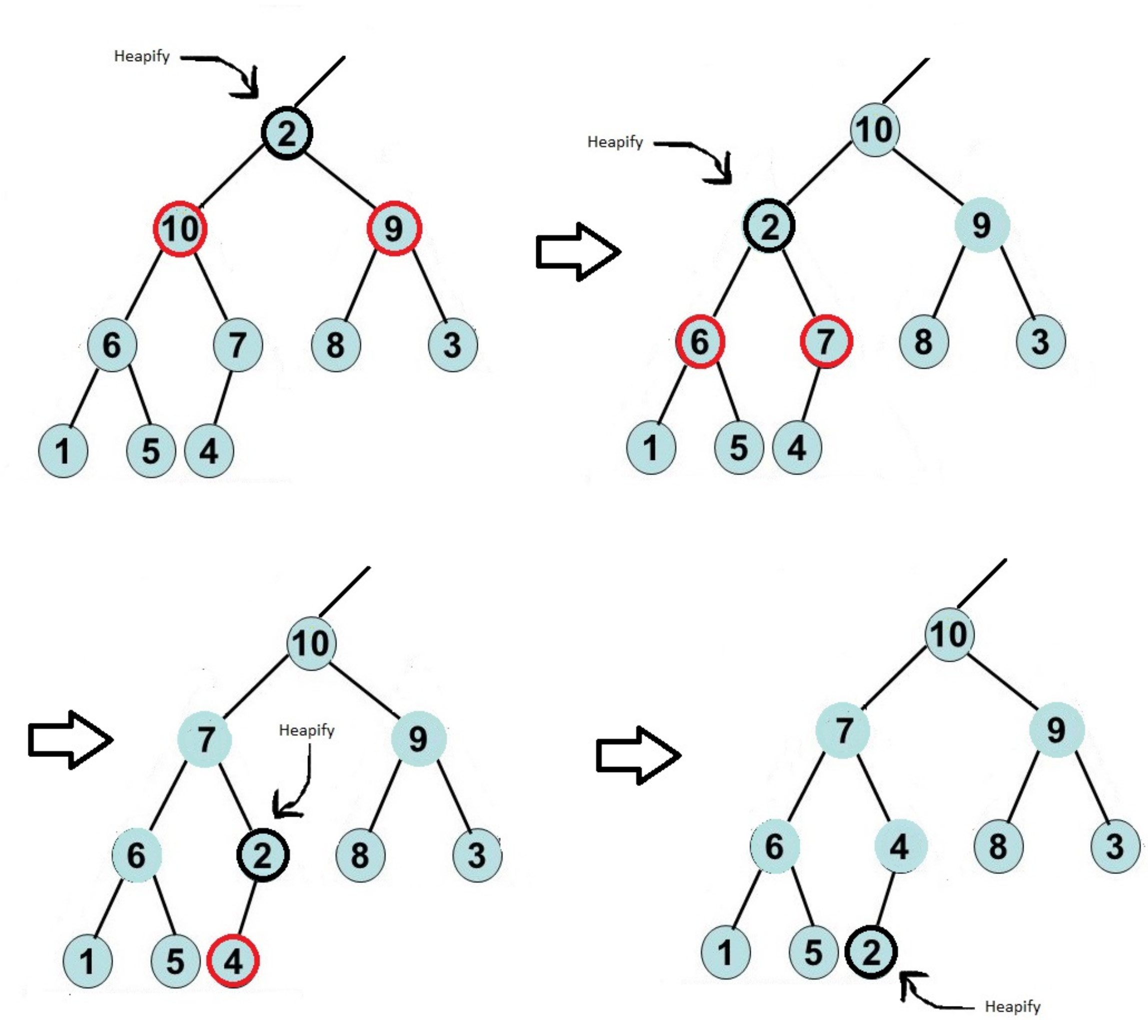


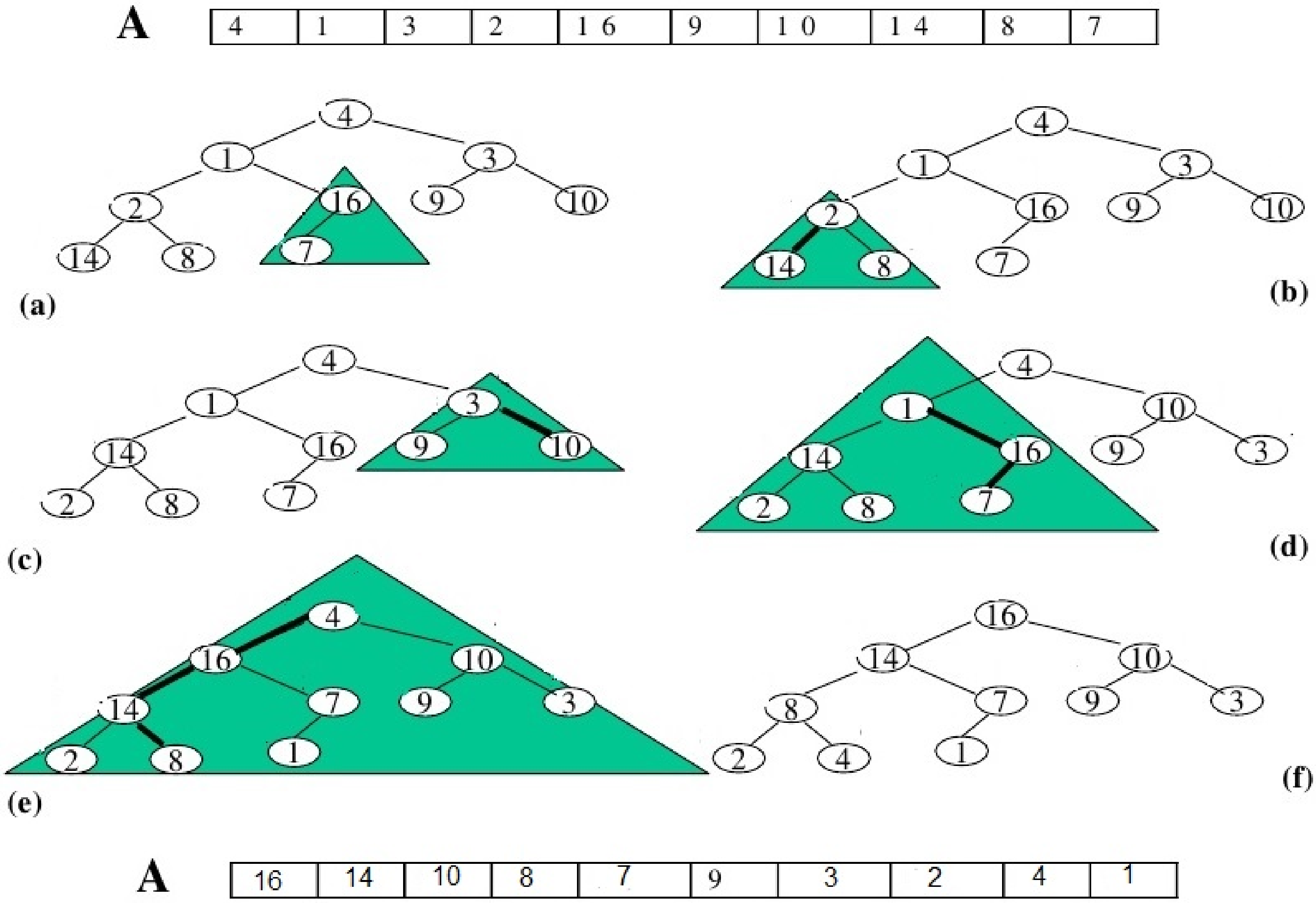
Figure ۴: MAX-Heapify برای راس با کلید 2 انجام می‌شود. زیردرختهای این راس قبلا بصورت هرم بیشینه درآمده‌اند

```
MAX-Heapify(A,i)
l= LeftChild(i)
r= RightChild(i)
if(l <= Length(A) and A[l] > A[i])
       bigchild = l
else
       bigchild = i
if(r <= Length(A)  and  A[r] > A[bigchild])
       bigchild= r
if(bigchild != i)
       swap(A[i], A[bigchild])
       MAX-Heapify(A, bigchild)
```

برای تبدیل کل آرایه به هرم بیشینه، الگوریتم Build-Heap از آخرین راس غیربرگ که در اندیس $\lfloor n/2 \rfloor$ قرار دارد، پروسه MAX-Heapify را شروع می‌کند و این رویه را برای همه راسهای غیربرگ از پایین به بالا انجام می‌دهد.

```
Build-Heap(A)
for i = Len(A)/2  downto 1
       MAX-Heapify(A,i)
```

شکل زیر مراحل انجام الگوریتم Build-Heap برای یک آرایه با ۱۰ عنصر را نشان می‌دهد. زیررویه MAX-Heapify برای هر زیردرخت انجام می‌شود.



۳ تحلیل زمان اجرای الگوریتم Build-Heap

الگوریتم Build-Heap که در درس قبل مطالعه کردیم یک آرایه با n عنصر را تبدیل به یک هرم بیشینه می‌کند. این الگوریتم زیررویه MAX-Heapify را برای هر زیردرخت (در واقع برای زیردرختهای با ریشه غیربرگ) اجرا می‌کند. زمان اجرای زیررویه MAX-Heapify به ارتفاع زیردرخت داده شده بستگی دارد. اگر ارتفاع زیردرخت h باشد زیررویه MAX-Heapify به تعداد $O(h)$ عمل اصلی انجام می‌دهد. چون ارتفاع هر زیردرخت حداکثر $\log n$ است، پس می‌توان گفت زمان اجرای الگوریتم Build-Heap از مرتبه $O(n \log n)$ است. اما با یک تحلیل دقیقتر می‌توان ثابت کرد که زمان اجرای الگوریتم Build-Heap در واقع از مرتبه $O(n)$ است. برای اثبات این ادعا ابتدا لم زیر را ثابت می‌کنیم.

مشاهده. در یک هرم با n راس، تعداد برگها دقیقا برابر با $\lceil \frac{n}{2} \rceil$ است.
اثبات: داریم

$$\text{تعداد برگها} = \{i \mid i \leq n \text{ و } 2i > n\}$$

دقیقا $\lceil n/2 \rceil$ اندیس با این شرایط می‌تواند وجود داشته باشد. $\square$

لم. در یک هرم بیشینه با n راس، حداکثر $\lceil \frac{n}{2^{h+1}} \rceil$ راس ارتفاع h دارند.

اثبات: یک راس ارتفاع h دارد وقتی فاصله‌اش تا دورترین برگ در زیردرختش برابر با h باشد. یک برگ ارتفاع صفر دارد. ادعای لم را با استفاده از استقرا روی ارتفاع درخت اثبات می‌کنیم.

پایه استقرا: $h = 0$ برای راسهای با ارتفاع صفر (برگها) اثبات حکم از مشاهده بالا نتیجه میشود.

حال فرض کنید برای هر ارتفاع $h - 1$ و کمتر ادعای لم درست باشد. ثابت می‌کنیم برای ارتفاع h نیز درست است. فرض کنید n_h تعداد راسهای با ارتفاع h در درخت T باشد ($h \geq 1$). برگهای T را حذف می‌کنیم و اسم درخت حاصل را T' می‌گذاریم. دقت کنید راسهایی که در T ارتفاع h داشتند اکنون در T' ارتفاع $h - 1$ دارند. فرض کنید T' به تعداد n' راس داشته باشد. داریم

$$n' = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$$

با استفاده از فرض استقرا داریم

$$n_h = n'_{h-1} \leq \lceil \frac{n'}{2^{(h-1)+1}} \rceil = \lceil \frac{\lfloor n/2 \rfloor}{2^h} \rceil \leq \lceil \frac{n/2}{2^h} \rceil \leq \lceil \frac{n}{2^{h+1}} \rceil$$

$\square$

قضیه. زمان اجرای Build-Heap برای یک آرایه با n عنصر از مرتبه $O(n)$ است.

اثبات: می‌دانیم که هزینه MAX-Heapify برای یک راس با ارتفاع h برابر با $O(h)$ است. اگر $T(n)$ زمان اجرای Build-Heap باشد، با توجه به لم قبلی داریم

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil \times O(h) \leq \sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) < \sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^h} O(h) \leq O(n) \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}$$

برای اثبات یک کران بالا برای $\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}$ میتوانیم از نامساوی زیر استفاده میکنیم.

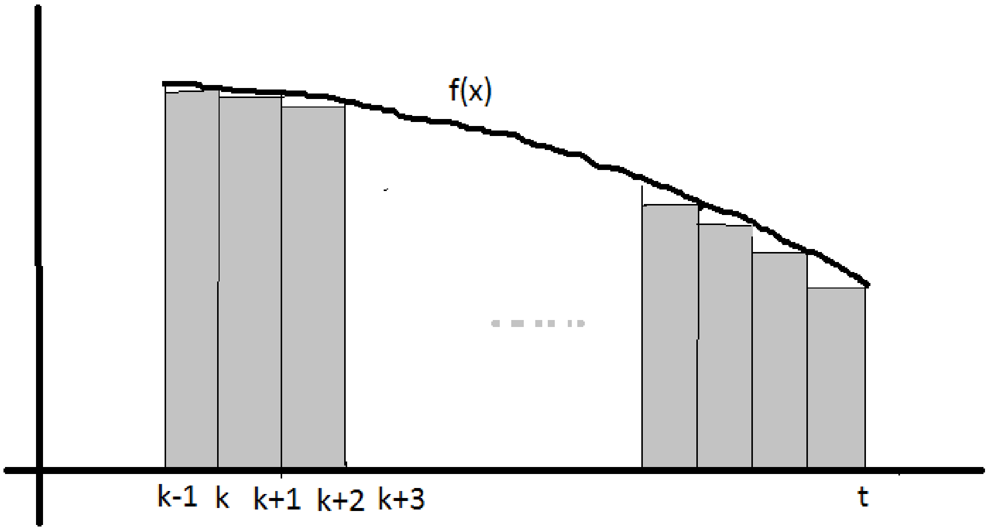
$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} < \int_0^\infty \frac{h}{2^h} dh$$

دقت کنید بطور کلی اگر $f(x)$ یک تابع انتگرال پذیر، پیوسته و نزولی در بازه $[k-1, t]$ باشد میتوانیم بنویسیم

$$\sum_{x=k}^{t} f(x) \leq \int_{k-1}^{t} f(x)dx \leq \int_{k-1}^{\infty} f(x)dx$$

به تصویر زیر دقت کنید. مقدار $\int_k^{t+1} f(x)dx$ برابر با مساحت زیر نمودار $f(x)$ است در حالیکه $\sum_{x=k}^{t} f(x)$ برابر با مجموع مساحت ستونها میباشد.

داریم



$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} = \frac{1}{2} + \sum_{h=2}^{\lfloor \log n \rfloor} \frac{h}{2^h}$$

چون $\frac{h}{2^h}$ در بازه $[1, \infty)$ نزولی است پس

$$\sum_{h=2}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq \int_1^\infty \frac{h}{2^h} dh$$

از طرف دیگر داریم

$$\int \frac{h}{2^h} dh = -\frac{h \ln 2 + 1}{2^h \ln^2 2} + C$$

لذا

$$\sum_{h=2}^{\lfloor \log n \rfloor} \frac{h}{2^h} < \frac{1}{2} + \int_1^\infty \frac{h}{2^h} dh \leq \frac{1}{2} + \frac{\ln 2 + 1}{2 \ln^2 2} = O(1)$$

پس داریم

$$T(n) \leq O(n) \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq O(n) \times O(1) = O(n)$$

□

یک راه بهتر برای محاسبه $\sum_0^\infty \frac{h}{2^h}$

وقتی $|x| < 1$ داریم

$$\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$$

از طرفین مشتق میگیریم

$$\sum_{n=0}^{\infty} nx^{n-1} = \frac{1}{(1-x)^2}$$

ضرب طرفین در x

$$\sum_{n=0}^{\infty} nx^n = \frac{x}{(1-x)^2}$$

لذا برای $x = \frac{1}{2}$ بدست می آید

$$\sum_{n=0}^{\infty} n(\frac{1}{2})^n = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$$