به نام خدا

# ساختمان داده

آرش شفیعی

# مرتب‌سازی

## مقدمه

- در این بخش در مورد چند الگوریتم مرتب‌سازی صحبت خواهیم کرد و داده ساختارهای مورد نیاز برای این الگوریتم‌ها را معرفی خواهیم کرد.
- مسئله مرتب‌سازی [1] به شرح زیر است :
- ورودی : دنباله‌ای از $n$ عدد به صورت $\langle a_1, a_2, \cdots, a_n \rangle$
- خروجی : ترتیبی (یا جایگشتی) از ورودی‌ها به صورت $\langle a'_1, a'_2, \cdots, a'_n \rangle$ به طوری‌که $a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n$
- دنباله ورودی معمولا به صورت یک آرایه $n$ عنصری نشان داده می‌شود، گرچه می‌تواند با استفاده از داده ساختارهای دیگری مانند لیست پیوندی نیز مدلسازی شود.

[1] sorting problem

## مقدمه

- معمولا در عمل عناصر یک آرایه مقادیر مجزا نیستند، بلکه هرکدام نمایندهٔ مجموعه‌ای از داده‌ها هستند که آن را رکورد [1] می‌نامیم.
- یک رکورد مجموعه‌ای از داده‌ها با نوع‌های داده‌ای متفاوت است که با یک کلید مشخص می‌شود. برای مثال اطلاعات یک دانشجو یک رکورد را تشکیل می‌دهد و شماره دانشجویی کلید رکورد است.
- در عمل وقتی کلیدهای تعدادی رکورد مرتب می‌شوند، داده‌های رکوردهای آنها هم به همراه کلیدها جابجا می‌شوند. اگر اطلاعات یک رکورد زیاد باشند می‌توان در کنار کلید اشاره‌گری به اطلاعات رکورد مربوط به کلید ذخیره کرد و اشاره‌گر را به همراه کلید در رکورد جابجا نمود.
- در این قسمت تنها در مورد الگوریتم‌های مرتب‌سازی صحبت می‌کنیم صرف نظر از اینکه چه نوع اطلاعاتی را مرتب می‌کنیم. هنگام توصیف الگوریتم‌ها فرض می‌کنیم ورودی از اعداد تشکیل شده است.

[1] record

# مقدمه

- بسیاری از دانشمندان علوم‌کامپیوتر بر این باورند که مرتب‌سازی یکی از پایه‌ای‌ترین و مهم‌ترین مسائل در مطالعه الگوریتم‌هاست. دلایل زیادی مسبب این اهمیت اند از جمله این که :
- بسیاری از برنامه‌ها نیاز به مرتب‌سازی اطلاعات دارند. برای مثال، در یک سامانه دانشگاهی نیاز داریم دانشجویان را بر اساس یک شاخص مثلاً شماره دانشجویی و یا نام‌خانوادگی مرتب کنیم.
- بسیاری از الگوریتم‌های کامپیوتری نیاز به مرتب‌سازی دارند. برای مثال، یک برنامه کامپیوتری که اشیاء گرافیکی را پردازش می‌کند برای به تصویر کشیدن اشیاء نیاز دارد آنها را با اساس فاصلهٔ آنها از بیننده تصویر مرتب کند.

## مقدمه

- می‌توانیم روش‌ها و رویکردهای مختلف الگوریتمی را توسط الگوریتم‌های مرتب‌سازی مورد بررسی قرار دهیم. در طی سالیان، الگوریتم‌های مرتب‌سازی مختلفی با رویکردها و روش‌های گوناگون به وجود آمده‌اند که مطالعه هرکدام به فهم بهتر مبحث الگوریتم‌ها و داده‌ساختارها کمک می‌کند.
- می‌توانیم روش‌های مختلف مرتب‌سازی را براساس عملکردشان با یکدیگر مقایسه کنیم. با توجه به پراکندگی و ساختار ورودی مورد نظر برای جستجو، الگوریتم مناسب را انتخاب کنیم. برای هریک از روش‌های مرتب‌سازی می‌توانیم کران زمان اجرا را محاسبه و اثبات و با الگوریتم‌های دیگر مقایسه کنیم.

# الگوریتم‌های مرتب‌سازی

- در مقدمه تحلیل الگوریتم‌ها، مرتب‌سازی درجی [1] را معرفی کردیم که در بدترین حالت یک آرایه $n$ عنصری را در زمان $\Theta(n^2)$ مرتب می‌کند. این الگوریتم به فضای اضافی برای مرتب‌سازی نیاز ندارد و آرایه را درجا [2] مرتب می‌کند که می‌تواند برای آرایه‌های بسیار بزرگ یک مزیت محسوب شود.

[1] insertion sort
[2] in place

# الگوریتم‌های مرتب‌سازی

- در این بخش درمورد یک الگوریتم مرتب‌سازی دیگر صحبت خواهیم کرد.
- مرتب‌سازی هرمی [1] آرایه‌ای از $n$ عنصر را درجا در زمان $O(n \lg n)$ مرتب می‌کند.
- مرتب‌سازی هرمی از یک داده ساختار مهم به نام هرم یا هیپ [2] استفاده می‌کند که به معرفی این داده ساختار و موارد استفاده از آن خواهیم پرداخت.

[1] heap sort
[2] heap

# الگوریتم‌های مرتب‌سازی

- مرتب‌سازی هرمی و مرتب‌سازی درجی در دسته‌ای از الگوریتم‌های مرتب‌سازی به نام مرتب‌سازی‌های مقایسه‌ای قرار می‌گیرند. دو الگوریتم مرتب‌سازی مقایسه‌ای مهم دیگر که در مبحث طراحی الگوریتم‌ها به آن پرداخته می‌شود مرتب‌سازی ادغامی [1] و مرتب‌سازی سریع [2] هستند. می‌توان اثبات کرد که هر نوع مرتب‌سازی مقایسه‌ای نمی‌تواند در زمان کمتر از $n \lg n$ انجام شود بنابراین کران پایین زمان اجرای مرتب‌سازی‌های مقایسه‌ای $\Omega(n \lg n)$ است.

[1] merge sort
[2] quick sort

# الگوریتم‌های مرتب‌سازی

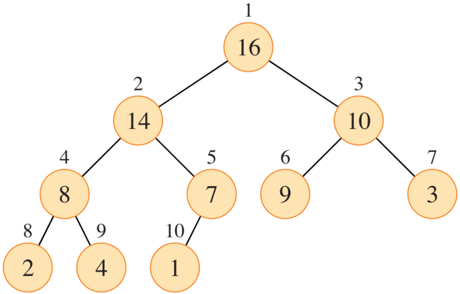
- اگر مجموعه‌ای که می‌خواهیم مرتب کنیم شامل اعداد حسابی $\{0, 1, \cdots, k\}$ باشد با استفاده از یک حافظه اضافی می‌توانیم زمان اجرا را به $\Theta(k + n)$ کاهش دهیم. این الگوریتم مرتب‌سازی را الگوریتم مرتب‌سازی شمارشی [1] می‌نامیم. اگر $k = O(n)$ باشد، مرتب‌سازی شمارشی اعداد را در زمان خطی نسبت به تعداد اعداد آرایه مرتب می‌کند.

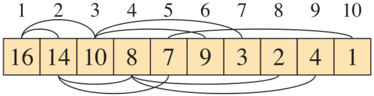
- الگوریتم مرتب‌سازی سطلی [2] الگوریتم دیگری است که در دسته الگوریتم‌های مرتب‌سازی خطی قرار می‌گیرد. اگر اطلاعاتی از توزیع احتمالی اعداد در آرایه ورودی داشته باشیم و بدانیم $n$ عدد ورودی دارای توزیع احتمالی یکنواخت [3] هستند می‌توانیم از مرتب‌سازی سطلی استفاده کنیم که در حالت میانگین اعداد را در زمان $O(n)$ مرتب می‌کند.

[1] counting sort
[2] bucket sort
[3] uniform distribution

# داده ساختار هرم

- داده ساختار هرم [1] یا هرم دودویی [2] آرایه‌ای از اشیاء است که می‌توانیم آن را به صورت یک درخت دودویی نشان دهیم.
- در شکل زیر یک نمونه از داده ساختار هرم نشان داده شده است.

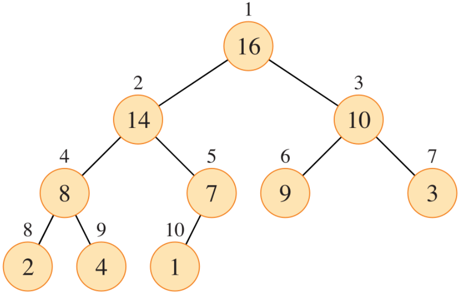


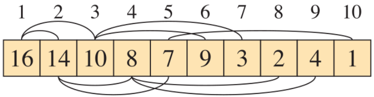




[1] heap
[2] binary heap

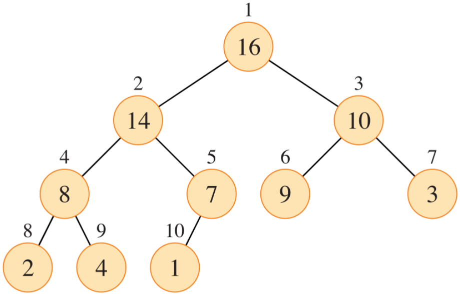
# داده ساختار هرم

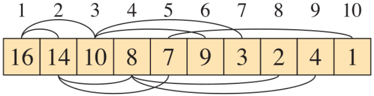
- هر رأس در درخت متناظر با یکی از عناصر آرایه است. درخت در همهٔ سطوح به غیر از سطح آخر یا سطح برگ‌ها کامل است و در سطح آخر از چپ به راست پر شده است.



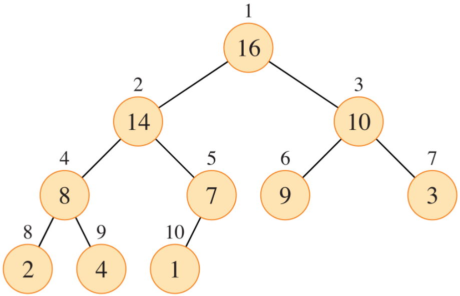
- آرایه $A[1:n]$ که یک هرم را نمایش می‌دهد یک شیء با ویژگی $A.heap\text{-}size$ است که تعداد عناصر هرم را که در آرایه A ذخیره شده است نمایش می‌دهد. بنابراین اگرچه $A[1:n]$ می‌تواند از اعدادی تشکیل شده باشد، تنها عناصر $A[1:A.heap\text{-}size]$ به طوری‌که $0 \leqslant A.heap\text{-}size \leqslant n$ عناصر معتبر هرم هستند. اگر $A.heap\text{-}size = 0$ باشد، هرم خالی است.

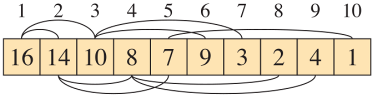


# داده ساختار هرم

- ریشه درخت $A[1]$ است و به ازای اندیس $i$ از یک رأس در درخت، روش ساده‌ای برای محاسبه اندیس‌های پدر، فرزند چپ و فرزند راست آن رأس وجود دارد.



# داده ساختار هرم

- در زیر روش محاسبه اندیس‌های پدر، فرزند چپ و فرزند راست نشان داده شده است.

**Algorithm** Parent

```
   function PARENT(i)
1: return ⌊ i/2 ⌋
```

**Algorithm** Left

```
   function LEFT(i)
1: return 2i
```

**Algorithm** Right

```
   function RIGHT(i)
1: return 2i + 1
```

# داده ساختار هرم

- در بسیاری از کامپیوترها مقدار $2i$ می‌تواند با یک دستور ماشین توسط انتقال عدد دودویی متناظر با $i$ یک واحد به سمت چپ محاسبه شود. همچنین مقدار $i/2$ توسط یک انتقال به راست محاسبه می‌شود.
- دو نوع هرم دودویی وجود دارد : هرم بیشینه [1] و هرم کمینه [2].
- هر دو نوع هرم دارای ویژگی هرم [3] هستند.

---

[1] max heap
[2] min heap
[3] heap property

# داده ساختار هرم

- در هرم بیشینه ویژگی هرم بیشینه [1] بدین صورت است که به ازای هر رأس $i$ به غیر از ریشه داریم :

$$A[\mathrm{Parent}(i)] \geqslant A[i]$$

- این ویژگی بدین معناست که مقدار یک رأس در درخت حداکثر برابر با مقدار رأس پدر آن رأس است . بنابراین بزرگ‌ترین عنصر در یک هرم بیشینه در ریشه ذخیره می‌شود و یک زیردرخت دارای یک ریشه مقداری بزرگ‌تر از ریشه را شامل نمی‌شود.

[1] max heap property

# داده ساختار هرم

- یک هرم کمینه به روش عکس هرم بیشینه ذخیره‌سازی را انجام می‌دهد. ویژگی هرم کمینه [1] بدین صورت است که به ازای هر رأس $i$ به غیر از ریشه، داریم :

$$A[\mathrm{Parent}(i)] \leqslant A[i]$$

- بنابراین دریک هرم کمینه کوچکترین عنصر در ریشه ذخیره می‌شود.

[1] min heap property

# داده ساختار هرم

- الگوریتم مرتب‌سازی هرمی و صف اولویت [1] دو مورد از کاربردهای داده‌ساختار هرم هستند که درمورد آن‌ها بعدها صحبت خواهیم کرد.

[1] priority queue

- اگر یک هرم را به صورت یک درخت نمایش دهیم، می‌توانیم ارتفاع [1] یک رأس از درخت را تعریف کنیم.
- ارتفاع یک رأس از درخت، تعداد یال‌هایی است که برروی بلندترین مسیر ساده از یکی از برگ‌های درخت به آن رأس قرار دارد. ارتفاع یک هرم ارتفاع ریشه هرم است.
- از آنجایی که یک هرم دارای $n$ عنصر بر پایهٔ درخت دودویی کامل [2] است، ارتفاع آن $\Theta(\lg n)$ است.
- عملیات اصلی که بر روی هرم اجرا می‌شوند حداکثر در زمانی متناسب با ارتفاع هرم اجرا می‌شوند و در نتیجه حداکثر به زمان $O(\lg n)$ نیاز دارند.

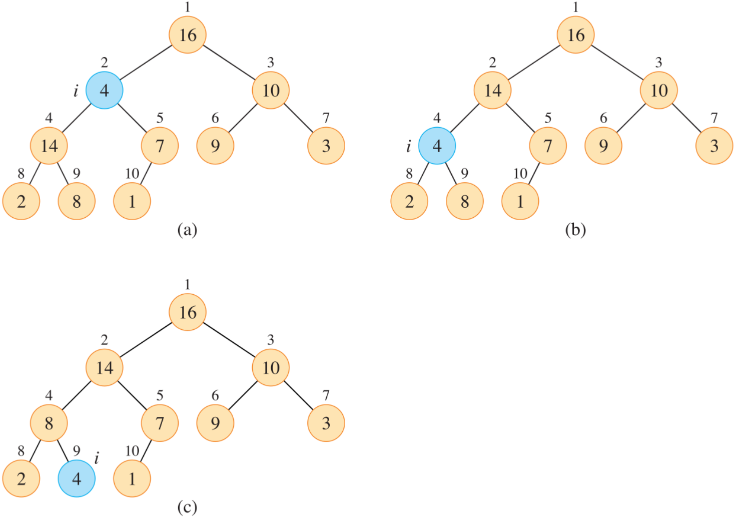
[1] height
[2] complete binary tree

# حفظ ویژگی هرم

- تابع Max–Heapify تابعی است که در صورتی که ریشه یک هرم بیشینه دارای ویژگی هرم نباشد (با فرض اینکه فرزندان هر دو هرم بیشینه باشند)، ویژگی هرم بیشینه را بازمی‌گردارند.
- ورودی آن یک آرایه A با اندازه heap–size و یک اندیس i در آرایه است. وقتی این تابع فراخوانی می‌شود، تابع فرض را بر این می‌گذارد که دو درخت دودویی با ریشه‌های Left(i) و Right(i) هرم بیشینه هستند، اما A[i] ممکن است کوچک‌تر از فرزندانش باشد. تابع Max–Heapify مقدار A[i] را در هرم بیشینه پایین می‌برد تا جایی که ویژگی هرم بیشینه به دست آید.

# حفظ ویژگی هرم

- شکل زیر عملیات Max-Heapify را نشان می‌دهد.



# حفظ ویژگی هرم

- تابع Max-Heapify به صورت زیر است.

**Algorithm** Max Heapify

```
function MAX-HEAPIFY(A,i)
l = Left(i)
r = Right(i)
if l ⩽ A.heap-size and A[l] > A[i] then
   largest = l
else  largest = i
if r ⩽ A.heap-size and A[r] > A[largest] then
   largest = r
if largest ≠ i then
   exchange A[i] with A[largest]
   Max-Heapify (A, largest)
```

# حفظ ویژگی هرم

- در هرگام، تابع بیشترین مقدار عناصر A[i] و A[Left(i)] و A[Right(i)] را تعیین می‌کند و اندیس بزرگ‌ترین عنصر را در متغیر largest ذخیره می‌کند. اگر A[i] بزرگ‌ترین عنصر است، زیر درخت با ریشه i یک هرم بیشینه است و نیاز به انجام هیچ عملیاتی نیست. در غیر اینصورت، یکی از دو فرزند دارای بیشترین مقدار است و مکان i و largest تعویض می‌شود که باعث می‌شود رأس i و فرزندانش ویژگی هرم بیشینه را حفظ کنند. اما اکنون زیر درخت با رأس largest ممکن است ویژگی هرم بیشینه را نقض کند، پس دوباره تابع Max-Heapify به صورت بازگشتی باید فراخوانی شود.

## حفظ ویژگی هرم

- برای تحلیل تابع Max-Heapify ، فرض کنید $T(n)$ زمان اجرا در بدترین حالت باشد که تابع برای یک زیر درخت با اندازه $n$ صرف می‌کند.
- برای یک درخت با ریشه $i$ زمان اجرا برای مقایسه عناصر $A[i]$ ، $A[Left(i)]$ و $A[Right(i)]$ برابر با $\Theta(1)$ است. تابع به صورت بازگشتی برای یکی از زیردرخت‌ها فراخوانی می‌شود. هرکدام از زیردرخت‌ها حداکثر $2n/3$ رأس دارند و بنابراین می‌توانیم زمان اجرای تابع Max-Heapify را با رابطه بازگشتی زیر توصیف کنیم.

$$T(n) \leqslant T(2n/3) + \Theta(1)$$

- با حل کردن این رابطه بازگشتی توسط قضیه اصلی به دست می‌آوریم $T(n) = O(\lg n)$
- همچنین می‌توانیم بگوییم زمان اجرای تابع Max-Heapify بر روی رأسی به ارتفاع $h$ برابر با $O(h)$ است.

# حفظ ویژگی هرم

- یک درخت دودویی کاملاً پُر [1] درختی است که در آن هر رأس میانی دقیقا دو فرزند دارد و عمق همهٔ برگ‌ها یکسان است.
- تعداد برگ‌های یک درخت دودویی کاملا پر با $n$ رأس میانی برابر با $n+1$ است.
- با فرض اینکه تعداد رئوس زیر درخت سمت راست ریشه در یک هرم برابر با $k$ است، تعداد رئوس زیردرخت سمت چپ حداکثر برابر با $2k+1$ است. در چنین درختی تعداد کل رئوس برابر است با $3k+2$ .
- بنابراین خواهیم داشت : $2k+1 \leq 2n/3 \leq 2(3k+2)/3$ .

[1] perfect binary tree

# ساختن هرم

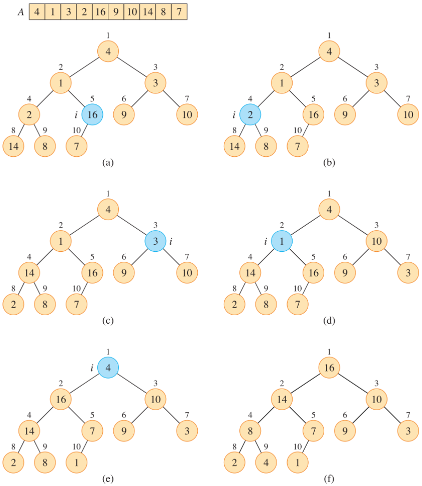
- تابع Build-Max-Heap آرایه $A[1:n]$ را به یک هرم بیشینه تبدیل می‌کند.
- عناصر زیر آرایه $A[\lfloor n/2 \rfloor + 1 : n]$ همه برگ‌های درخت هستند.
- تابع Build-Max-Heap به ازای همه عناصر غیر برگ تابع Max-Heapify را فراخوانی می‌کند.

**Algorithm** Build Max Heap

```
function BUILD-MAX-HEAP(A,n)
A.heap-size = n
for i = ⌊ n/2 ⌋ downto 1 do
    Max-Heapify(A,i)
```

# ساختن هرم

- شکل زیر یک مثال از عملیات تابع Build–Max–Heap را نشان می‌دهد.



## ساختن هرم

- برای اینکه نشان دهیم چرا تابع Build–Max–Heap درست کار می‌کند، باید اثبات کنیم در ابتدای حلقه for هر یک از رأس‌های $i+1, i+2, \cdots, n$ ریشه یک هرم بیشینه است.

- باید نشان دهیم این گزاره در ابتدای اولین تکرار حلقه درست است و در هر تکرار حلقه اگر گزاره در ابتدای یک تکرار درست باشد، در ابتدای تکرار بعدی نیز درست است. بدین ترتیب به روش استقرایی اثبات می‌کنیم الگوریتم درست عمل می‌کند.

**Algorithm** Build Max Heap

```
function BUILD-MAX-HEAP(A,n)
A.heap-size = n
for i = ⌊ n/2 ⌋ downto 1 do
   Max-Heapify(A,i)
```

# ساختن هرم

- پایهٔ استقرا : قبل از اولین تکرار حلقه، $i = \lfloor n/2 \rfloor$ . هریک از رئوس $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \cdots, n$ یک برگ است و بنابراین به طور بدیهی ریشه یک هرم بیشینه است.

- گام استقرا : اندیس فرزندان رأس $i$ بزرگ‌تر از رأس $i$ است. طبق فرض استقرا هردوی فرزندان ریشه هرم بیشینه هستند. این دقیقا شرایطی است که برای فراخوانی تابع Build-Max-Heap(A،i) نیاز داریم تا رأس $i$ را در ریشه هرم بیشینه قرار دهیم. همچنین فراخوانی تابع Max-Heapify ویژگی هرم را حفظ می‌کند به طوری‌که رئوس $i+1, i+2, \cdots, n$ همه ریشه هرم بیشینه خواهند بود. بنابراین در تکرار بعدی حلقه این رئوس هریک ریشه یک هرم بیشینه خواهند بود.

- خاتمه‌پذیری : حلقه دقیقا $\lfloor n/2 \rfloor = i$ بار تکرار می‌شود بنابراین قطعا الگوریتم خاتمه می‌یابد. در پایان $i = 0$ است بنابراین طبق اثبات استقرایی هریک از رئوس $1, 2, \cdots, n$ ریشه یک هرم بیشینه هستند و رأس $1$ ریشه هرم بیشینه به دست آمده است.

# ساختن هرم

- می‌توانیم یک کران بالای ساده بر روی زمان Build-Max-Heap به صورت زیر محاسبه کنیم. زمان اجرای هر فراخوانی Max-Heapify $O(\lg n)$ است و Build-Max-Heap تعداد $O(n)$ فراخوانی انجام می‌دهد. بنابراین زمان اجرا $O(n \lg n)$ است. این کران بالا اگرچه درست است، اما به میزان کافی دقیق نیست.

- می‌توانیم یک کران مجانبی دقیق‌تر برای این تابع محاسبه کنیم. مشاهده می‌کنیم که زمان اجرای Max-Heapify بستگی به ارتفاع درختی دارد که برروی آن اجرا می‌شود. در یک هرم شامل $n$ عنصر ارتفاع $\lfloor \lg n \rfloor$ است و حداکثر تعداد $\lceil n/2^{h+1} \rceil$ رأس با ارتفاع $h$ وجود دارد.

- زمان مورد نیاز برای تابع Max-Heapify وقتی بر روی رأسی با ارتفاع $h$ فراخوانی می‌شود برابراست با $O(h)$.

# ساختن هرم

- فرض کنید $c$ یک ضریب ثابت در تحلیل مجانبی باشد، آنگاه زمان اجرای Build–Max–Heap را به صورت زیر می‌توانیم توصیف کنیم.

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil ch$$

- به ازای $0 \leqslant h \leqslant \lfloor \lg n \rfloor$ داریم $n/2^{h+1} \geqslant 1/2$.

$$\lg n \geqslant \lfloor \lg n \rfloor \geqslant h \implies 2^{\lg n} \geqslant 2^{\lfloor \lg n \rfloor} \geqslant 2^h \implies n \geqslant 2^h \implies n/2^{h+1} \geqslant 1/2$$

- همچنین از آنجایی که به ازای هر $x \geqslant 1/2$ داریم $\lceil x \rceil \leqslant 2x$ ، بنابراین $\lceil n/2^{h+1} \rceil \leqslant n/2^h$.

# ساختن هرم

- بنابراین می‌توانیم رابطه زیر را به دست آوریم.

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil ch \leqslant \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch < cn \sum_{h=0}^{\infty} \frac{h}{2^h} < cn \frac{1/2}{(1-1/2)^2} = O(n)$$

- بنابراین می‌توانیم یک هرم بیشینه از یک آرایه غیر مرتب در زمان خطی بسازیم.
- برای ساختن هرم کمینه از تابع Build-Min-Heap استفاده می‌کنیم که شبیه Build-Max-Heap است با این تفاوت که تابع Min-Heapify فراخوانی می‌شود.

# مرتب‌سازی هرمی

- الگوریتم مرتب‌سازی هرمی [1] با فراخوانی تابع Build-Max-Heap یک هرم بیشینه بر روی آرایه ورودی $A[1:n]$ می‌سازد. از آنجایی که مقدار بیشینه عناصر در ریشه $A[1]$ ذخیره شده است، الگوریتم مرتب‌سازی هرمی می‌تواند این مقدار بیشینه را در مکان درست آن یعنی $A[n]$ قرار دهد.
- حال اگر تابع رأس $n$ را از هرم بردارد یا به عبارت دیگر `A.heap-size` را یک واحد کاهش دهد، فرزندان ریشه همچنان هرم بیشینه باقی می‌مانند. برای حفظ ویژگی هرم تابع `Max-Heapify(A,1)` فراخوانی می‌شود و در نتیجه $A[1:n-1]$ یک هرم بیشینه می‌شود. این عملیات ادامه می‌یابد تا جایی که اندازه هرم برابر با 2 شود.

[1] heapsort algorithm

# مرتب‌سازی هرمی

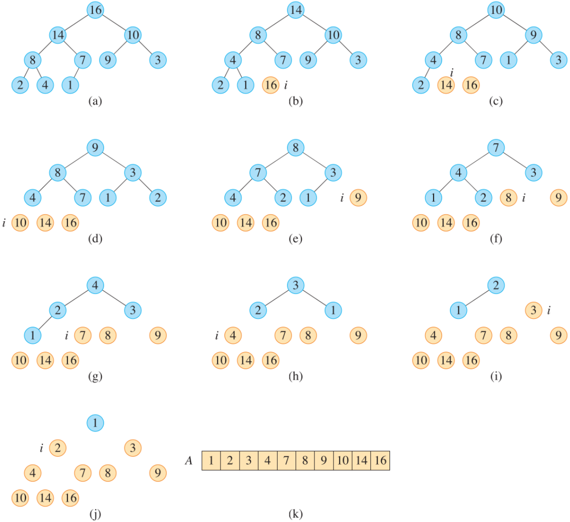
- الگوریتم مرتب‌سازی هرمی به صورت زیر است.

**Algorithm** Heapsort

```
function HEAPSORT(A,n)
Build-Max-Heap (A,n)
for i = n downto 2 do
    exchange A[1] with A[i]
    A.heap-size = A.heap-size-1
    Max-Heapify (A,1)
```

# مرتب‌سازی هرمی

- شکل زیر یک مثال از عملیات Heapsort را نشان می‌دهد.



# مرتب‌سازی هرمی

- مرتب‌سازی هرمی در زمان $O(n \lg n)$ اجرا می‌شود، زیرا فراخوانی تابع Build-Max-Heap به زمان $O(n)$ نیاز دارد و هریک از $n - 1$ فراخوانی تابع Max-Heapify در زمان $O(\lg n)$ اجرا می‌شود.

# صف اولویت

- داده ساختار هرم کاربردهای زیادی در الگوریتم‌های مختلف دارد. یکی از کاربردهای آن صف اولویت [1] است.
- در اینجا درمورد یکی از انواع صف اولویت که صف اولویت بیشینه [2] است صحبت می‌کنیم که براساس هرم بیشینه است. صف اولویت کمینه [3] شبیه صف اولویت بیشینه است که از هرم کمینه استفاده می‌کند.
- یک صف اولویت داده ساختاری است برای نگهداری مجموعه S از عناصری که هرکدام با یک کلید [4] مشخص شده‌اند. صف اولویت بیشینه عملیات زیر را پشتیبانی می‌کند.

---

[1] priority queue
[2] max priority queue
[3] min priority queue
[4] key

# صف اولویت

- عملیات درج $\text{Insert}(S،x،k)$ : عنصر $x$ با کلید $k$ را در مجموعه $S$ درج می‌کند که معادل عملیات $S = S \cup \{x\}$ است.
- عملیات بیشینه $\text{Maximum}(S)$ : عنصری از $S$ که بزرگ‌ترین کلید را دارد را بر می‌گرداند.
- عملیات استخراج بیشینه $\text{Extract–Max}(S)$ : عنصری از $S$ که بزرگ‌ترین کلید را دارد را از مجموعه حذف کرده، و باز می‌گرداند.
- عملیات افزایش کلید $\text{Increase–key}(S،x،k)$ : مقدار کلید عنصر $x$ را با مقدار جدید $k$ افزایش می‌دهد فرض بر این است که مقدار $k$ بزرگ‌تر یا مساوی مقدار فعلی کلید $x$ است.

# صف اولویت

- صف اولویت می‌تواند کاربردهای زیادی داشته باشد. برای مثال واحدهای کاری [1] در یک سیستم‌عامل به هنگام اجرا وارد یک صف می‌شوند، اما یک زمانبند [2] ممکن است بخواهد آنها را براساس اولویت‌شان از صف خارج کند که در اینجا می‌توانیم از یک صف اولویت بیشینه استفاده کنیم. هرگاه یک واحدکاری به اتمام رسید، زمانبند واحد کاری با بیشترین اولویت را توسط Extract-Max از صف اولویت خارج می‌کند. هرگاه یک واحدکاری جدید توسط کاربر تعریف شد، توسط Insert وارد صف اولویت می‌شود.
- یک صف اولویت کمینه نیز عملیات Insert ، Minimum ، Extract-Min و Decrease-Key را پشتیبانی می‌کند.
- ممکن است یک زمانبند بخواهد واحدهای کاری که کمترین زمان اجرا دارند زودتر اجرا کند. در این صورت کلید عناصر زمان اجرای مورد نیاز آنهاست و از یک صف اولویت کمینه استفاده می‌شود.

---
[1] jobs or tasks
[2] scheduler

# صف اولویت

- تابع Max-Heap-Maximum عملیات Maximum را در زمان $\Theta(1)$ انجام می‌دهد.

**Algorithm** Max Heap Maximum

```
function MAX-HEAP-MAXIMUM(A)
if A.heap-size < 1 then
    error "heap underflow"
return A[1]
```

# صف اولویت

- تابع Max-Heap-Extract-Max عملیات Extract-Max را پیاده سازی می‌کند.

**Algorithm** Max Heap Extract Max

```
function MAX-HEAP-EXTRACT-MAX(A)
max = Max-Heap-Maximum(A)
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
Max-Heapify(A,1)
return max
```

- زمان اجرای Max-Heap-Extract-Max برابر با $O(\lg n)$ است، زیرا تنها تعدادی عملیات در زمان ثابت علاوه بر عملیات Max-Heapify انجام می‌دهد.

# صف اولویت

- در این الگوریتم‌ها فرض می‌کنیم Max-Heapify عناصر صف اولویت را بر اساس ویژگی کلید آنها مقایسه می‌کند و همچنین وقتی دو عنصر در آرایه جابجا می‌شوند، درواقع اشاره‌گری به عناصر آنها جابجا می‌شود.

# صف اولویت

- تابع Max–Heap–Increase–Key عملیات Increase–Key را انجام می‌دهد.

**Algorithm** Max Heap Increase Key

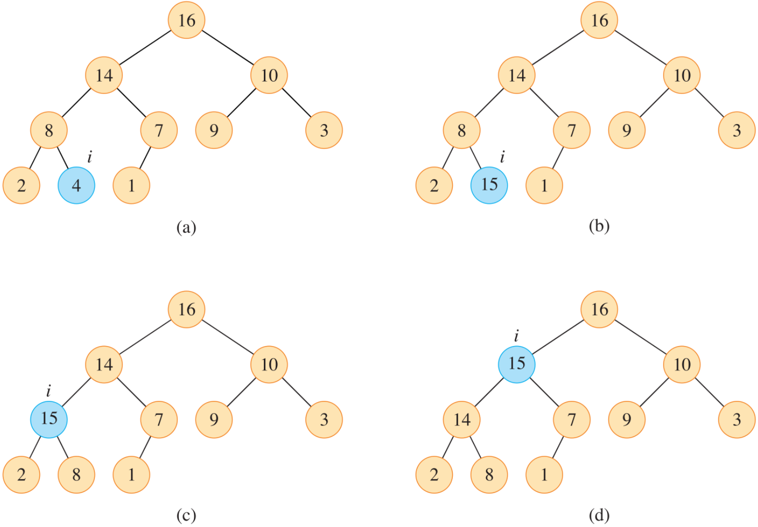
```
function MAX-HEAP-INCREASE-KEY(A,x,k)
if k < x.key then
    error "new key is smaller than current key"
x.key = k
i = x.i  ▷ find the index i in array A where object x occurs. Index i may be given as an input of the function.
while i > 1 and A[Parent(i)].key < A[i].key do
    exchange A[i] with A[Parent(i)]      ▷ and also update the information that maps priority queue objects to array indices
    i = Parent(i)
```

# صف اولویت

- این تابع ابتدا بررسی می‌کند که کلید جدید $k$ کلید فعلی $x$ را کاهش نمی‌دهد و در غیر این‌صورت، مقدار کلید $x$ را تغییر می‌دهد. سپس این تابع اندیس $i$ از $x$ را در آرایه پیدا می‌کند به طوری که $A[i]$ برابر با $x$ باشد.
- از آنجایی که افزایش کلید $A[i]$ ممکن است ویژگی هرم بیشینه را نقض کند، تابع یک مسیر ساده از رأس به سمت ریشه طی می‌کند تا مکان مناسب رأس را با کلید جدید پیدا کند.
- وقتی تابع Max-Heap-Increase-Key این مسیر را طی می‌کند، به طور مکرر کلید یک عنصر را با کلید پدر آن مقایسه می‌کند و در صورتی که کلید عنصر بزرگ‌تر بود، اشاره‌گرهای آنها با جابجا می‌کند و در صورتی که کلید عنصر از کلید پدر کوچک‌تر بود، الگوریتم به پایان می‌رسد، زیرا ویژگی هرم بیشینه برقرار خواهد بود.

# صف اولویت

- شکل زیر یک مثال از عملیات Max-Heap-Increase-Key را نشان می‌دهد.



# صف اولویت

- زمان اجرای عملیات افزایش کلید Max-Heap-Increase-Key برای هر هرم با $n$ عنصر برابر با $O(\lg n)$ است، زیرا مسیری که رأس تا ریشه طی می‌کند $O(\lg n)$ است.

# صف اولویت

- تابع Max-Heap-Insert عملیات درج Insert را پیاده‌سازی می‌کند.

**Algorithm** Max Heap Insert

```
function MAX-HEAP-INSERT(A,x,n)
if A.heap-size == n then
    error "heap overflow"
A.heap-size = A.heap-size + 1
k = x.key
x.key = -∞
A[A.heap-size] = x   ▷ map x to index heap-size in the array
Max-Heap-Increase-Key (A,x,k)
```

# صف اولویت

- این تابع به عنوان ورودی آرایه $A$ را که یک هرم بیشینه را پیاده‌سازی کرده است، به همراه عنصر جدید $x$ دریافت می‌کند و آن را در آرایه $A$ با اندازه $n$ درج می‌کند. این تابع بررسی می‌کند مکان کافی در آرایه برای درج عنصر جدید وجود دارد. سپس یک برگ با کلید $-\infty$ به هرم اضافه می‌کند. سپس تابع Max-Heap-Increase-Key را فراخوانی می‌کند تا کلید عنصر اخیرا افزوده شده را تنظیم و آن را در مکان مناسب خود قرار دهد تا ویژگی هرم برقرار شود.
- زمان اجرای Max-Heap-Increase بر روی یک هرم با $n$ عنصر برابر با $O(\lg n)$ است.

# مرتب‌سازی در زمان خطی

- الگوریتم‌های مرتب‌سازی بهینه یک آرایه $n$ تایی را در زمان $O(n \lg n)$ مرتب می‌کنند.
- برای مثال الگوریتم‌های مرتب‌سازی هرمی و ادغامی در بدترین حالت و مرتب‌سازی سریع در زمان متوسط مرتب‌سازی $n$ عنصر را در زمان $O(n \lg n)$ انجام می‌دهند.
- این الگوریتم‌ها که با مقایسه عناصر آرایه مرتب‌سازی را انجام می‌دهند در دسته الگوریتم‌های مرتب‌سازی مقایسه‌ای [1] قرار می‌گیرند.
- در این قسمت در مورد چند الگوریتم مرتب‌سازی صحبت می‌کنیم که آرایه را در زمان خطی نسبت به تعداد عناصر آرایه مرتب می‌کنند.

[1] comparison sort

# کران پایین مرتب‌سازی مقایسه‌ای

- اثبات می‌کنیم که هر الگوریتم مرتب‌سازی مقایسه‌ای کران پایین زمان اجرای $\Omega(n \lg n)$ در بدترین حالت دارد. به عبارت دیگر هیچ الگوریتم مقایسه‌ای وجود ندارد که آرایه $n$ عنصری را در زمانی کمتر از $n \lg n$ بتواند مرتب کند.

- یک مرتب‌سازی مقایسه‌ای تنها با مقایسه عناصر دنباله $\langle a_1, a_2, \cdots, a_n \rangle$ در مورد ترتیب عناصر آن اطلاعات کسب می‌کند. بدین معنی که به ازای دو عنصر $a_i$ و $a_j$ این الگوریتم‌ها یکی از مقایسه‌های $a_i < a_j$ ، $a_i \leqslant a_j$ ، $a_i = a_j$ ، $a_i \geqslant a_j$ و یا $a_i > a_j$ را انجام می‌دهند تا ترتیب نسبی آن دو را تعیین کنند.

- برای اثبات کران پایین [1] مرتب‌سازی‌های مقایسه‌ای بدون از دست دادن عمومیت اثبات [2] ، فرض می‌کنیم عناصر ورودی متمایز هستند. بنابراین فرض می‌کنیم همهٔ مقایسه‌ها به صورت $a_i \leqslant a_j$ هستند.

[1] lower bound
[2] without loss of generality

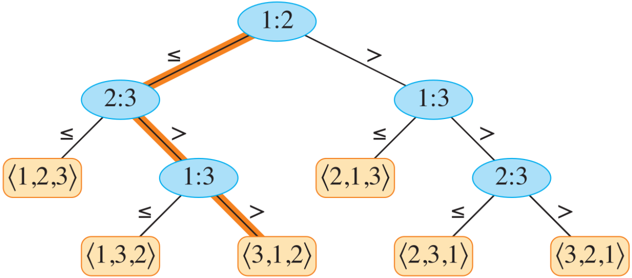
# کران پایین مرتب‌سازی مقایسه‌ای

- می‌توانیم مرتب‌سازی مقایسه‌ای را به صورت درخت‌های تصمیم [1] مدلسازی کنیم.
- یک درخت تصمیم یک درخت دودویی پر [2] است (بدین معنی که هر رأس یا یک برگ است و یا دو فرزند دارد) که مقایسه‌های بین عناصر آرایه را نمایش می‌دهد که توسط یکی از انواع مرتب‌سازی‌های مقایسه‌ای بر روی عناصر ورودی انجام شده است.

[1] decision trees
[2] full binary tree

# کران پایین مرتب‌سازی مقایسه‌ای

- شکل زیر یک درخت تصمیم را برای الگوریتم مرتب‌سازی درجی نشان می‌دهد.



# کران پایین مرتب‌سازی مقایسه‌ای

- هر یک از رئوس میانی درخت تصمیم با $i : j$ به ازای $1 \leqslant i$ و $n \leqslant j$ نمایش داده شده‌اند به طوری که $n$ تعداد عناصر دنباله ورودی است.
- هر یک از برگ‌ها با یک جایگشت $\langle \Pi(1), \Pi(2), \cdots, \Pi(n) \rangle$ نمایش داده شده‌اند.
- اندیس‌هایی که در رئوس درخت تصمیم نوشته شده‌اند مکان اصلی عناصر را نشان می‌دهند اجرای درخت تصمیم بر روی یک ورودی متناظر با یک مسیر ساده از ریشه به یکی از برگ‌هاست.
- هر رأس میانی یک مقایسه $a_i \leqslant a_j$ انجام می‌دهد. زیر درخت سمت چپ شرایطی را بررسی می‌کند که در آن $a_i \leqslant a_j$ و زیر درخت سمت راست شرایطی را بررسی می‌کند در آن $a_i > a_j$ است. با رسیدن به یک ریشه ترتیب $a_{\Pi(1)} \leqslant a_{\Pi(2)} \leqslant \cdots \leqslant a_{\Pi(n)}$ تعیین شده است.
- هر یک از $n!$ جایگشت $n$ عنصر باید در یکی از برگ‌های درخت تصمیم ظاهر شود.

# کران پایین مرتب‌سازی مقایسه‌ای

- طول بلندترین مسیر بین ریشه و برگ‌ها زمان اجرای الگوریتم مرتب‌سازی مقایسه‌ای در بدترین حالت را تعیین می‌کند. بنابراین بیشترین تعداد مقایسه‌ها در یک مرتب‌سازی مقایسه‌ای برابر با ارتفاع درخت تصمیم است.

- کران پایین برروی ارتفاع همه درخت‌های تصمیم که در آنها یک ترتیب در برگ‌های ظاهر می‌شود، درواقع کران پایین برروی زمان اجرای هرنوع مرتب‌سازی مقایسه‌ای است.

# کران پایین مرتب‌سازی مقایسه‌ای

- قضیه : هر الگوریتم مرتب‌سازی مقایسه‌ای نیاز به حداقل $\Omega(n \lg n)$ مقایسه در بدترین حالت برای مرتب‌سازی $n$ عنصر دارد.
- اثبات : مرتب‌سازی $n$ عنصر در یک الگوریتم مقایسه‌ای متناظر با یک درخت تصمیم با ارتفاع $h$ است. تعداد $n!$ جایگشت وجود دارد و یک درخت دودویی با ارتفاع $h$ بیشتر از $2^h$ برگ ندارد. بنابراین داریم :

$$n! \leqslant 2^h$$

- با گرفتن لگاریتم از دو طرف داریم $h \geqslant \lg(n!)$ .
- می‌توان ثابت کرد $\lg(n!) = \Omega(n \lg n)$ بنابراین $h = \Omega(n \lg n)$ .
- بنابراین مرتب‌سازی هرمی یک الگوریتم مرتب‌سازی بهینه است.

# مرتب‌سازی شمارشی

- مرتب‌سازی شمارشی [1] فرض می‌کند که هریک از $n$ عنصر ورودی یک عدد صحیح در بازه $0$ تا $k$ است. این الگوریتم در زمان $\Theta(n+k)$ اجرا می‌شود، بنابراین وقتی $k = O(n)$ باشد، مرتب‌سازی شمارشی در زمان $\Theta(n)$ اجرا می‌شود.
- مرتب‌سازی شمارشی ابتدا به ازای هر عنصر $x$ تعداد عناصر کوچک‌تر یا مساوی $x$ را تعیین می‌کند. سپس از این اطلاعات استفاده می‌کند تا مکان عنصر $x$ را در آرایه خروجی تعیین کند.
- برای مثال اگر تعداد ۱۷ عنصر کوچک‌تر یا مساوی $x$ هستند، مکان $x$ در آرایه خروجی مکان ۱۷ است. همچنین باید شرایطی که چند عنصر می‌توانند مساوی باشند را در نظر بگیریم زیرا نمی‌خواهیم همه عناصری که مقدارشان مساوی یکدیگر است را در یک مکان حافظه قرار دهیم.

[1] countiong sort

# مرتب‌سازی شمارشی

- تابع Counting-Sort آرایه $A[1:n]$ را که اندازه آن $n$ است به همراه مقدار حدی $k$ که یک عدد صحیح غیر منفی است را دریافت می‌کند و آرایه $B[1:n]$ شامل عناصر مرتب‌شده را باز می‌گرداند. در این مرتب‌سازی از آرایه $C[0:k]$ به عنوان فضای کمکی استفاده می‌کند.

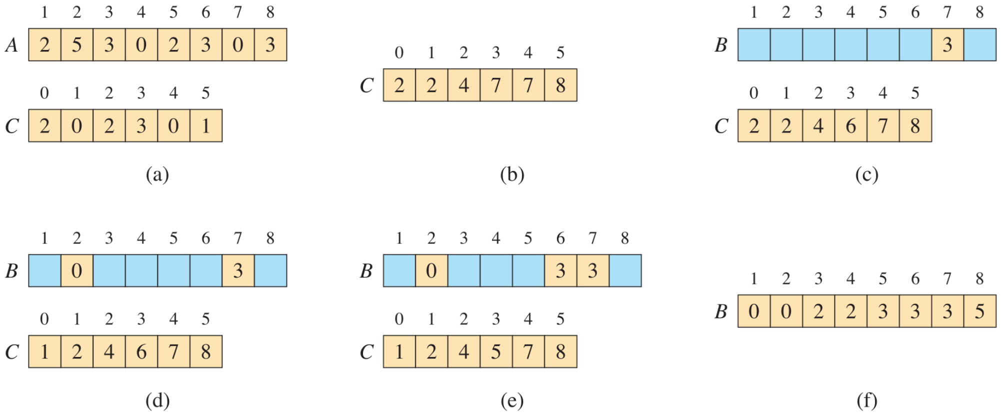
# مرتب‌سازی شمارشی

**Algorithm** Counting Sort

```
function COUNTING-SORT(A, n, k)
let B[1:n] and C[0:k] be new arrays
for i = 0 to k do
    C[i] = 0
for j = 1 to n do
    C[A[j]] = C[A[j]] + 1
▷ C[i] now contains the number of elements equal to i.
for i = 1 to k do
    C[i] = C[i] + C[i-1]
▷ C[i] now contains the number of elements less than or equal to i.
▷ Copy A to B, starting from the end of A.
for j = n downto 1 do
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1      ▷ to handle duplicate values
return B
```

# مرتب‌سازی شمارشی

- شکل زیر نحوه اجرای الگوریتم مرتب‌سازی شمارشی را نشان می‌دهد.



# مرتب‌سازی شمارشی

- بعد از حلقه خطوط ۲ و ۳ که آرایه $C$ را با صفر مقداردهی اولیه می‌کند، حلقه خطوط ۴ و ۵ یک بار آرایه $A$ را بررسی می‌کند تا اطلاعاتی درمورد آرایه به دست آورد. هربار یک عنصر ورودی که مقدار آن $i$ است پیدا می‌شود. مقدار $C[i]$ یک واحد افزایش پیدا می‌کند. بنابراین بعد از خط ۵، مقدار $C[i]$ تعداد عناصر ورودی است که مقدارشان برابر با $i$ است به ازای $i = 0, 1, \cdots, k$.
- خطوط ۷ و ۸ به ازای هر $i = 0, 1, \cdots, k$ تعیین می‌کند چند عنصر ورودی کوچک‌تر یا مساوی $i$ وجود دارند.
- در نهایت حلقه خطوط ۱۱ تا ۱۳ یک بار دیگر آرایه $A$ را بررسی می‌کنند، اما این‌بار از آخر. در این بررسی هر عنصر $A[j]$ در مکان درست خود در آرایه $B$ قرار می‌گیرد.

# مرتب‌سازی شمارشی

- اگر همه $n$ عنصر متمایز باشند، وقتی وارد خط ۱۱ می‌شویم، به ازای هر $A[j]$ مقدار $C[A[j]]$ مکان درست $A[j]$ در آرایه خروجی است زیرا $C[A[j]]$ عنصر کوچک‌تر یا مساوی $A[j]$ وجود دارد.
- از آنجایی که عناصر ممکن است متمایز نباشند، حلقه مقدار $C[A[j]]$ را در هربار که مقدار $A[j]$ را در $B$ قرار می‌دهد کاهش می‌دهد. کاهش $C[A[j]]$ باعث می‌شود عنصر قبلی $A$ با مقدار $A[j]$ به مکان قبل از $A[j]$ در آرایه $B$ برود.

# مرتب‌سازی شمارشی

- برای تحلیل زمانی الگوریتم مرتب‌سازی شمارشی، مشاهده می‌کنیم که حلقه خطوط ۲ و ۳ در زمان $\Theta(k)$ انجام می‌شود و حلقه خطوط ۴ و ۵ در زمان $\Theta(n)$ و حلقه خطوط ۷ و ۸ در زمان $\Theta(k)$ و حلقه خطوط ۱۱ تا ۱۳ در زمان $\Theta(n)$ انجام می‌شوند. بنابراین زمان کل مورد نیاز $\Theta(n+k)$ است.
- در عمل تنها وقتی از مرتب‌سازی شمارشی استفاده می‌کنیم که داشته باشیم $k = O(n)$ که در این‌صورت زمان اجرا برابر با $\Theta(n)$ خواهد بود.

# مرتب‌سازی شمارشی

- مرتب‌سازی شمارشی از مرتب‌سازی‌های مقایسه ای که کران پایین زمان اجرای آنها $\Omega(n \lg n)$ است بهتر عمل می‌کند. هیچ مقایسه‌ای بین عناصر در این الگوریتم انجان نمی‌شود و بنابراین این الگوریتم یک الگوریتم مقایسه‌ای نیست.

# مرتب‌سازی شمارشی

- یک ویژگی مهم مرتب‌سازی شمارشی این است که پایدار [1] است، بدین معنا که عناصری که مقدارشان برابر است در آرایه خروجی با همان ترتیب ورودی ظاهر می‌شوند. ویژگی پایداری مهم است بدین دلیل که ممکن است علاوه بر کلیدی که مرتب‌سازی برروی آن انجام می‌شود، داده‌های دیگری همراه کلید وجود داشته باشند و نخواهیم ترتیب آنها تغییر کند.

[1] stable

# مرتب‌سازی مبنایی

- مرتب‌سازی مبنایی [1] در مواقعی استفاده می‌شود که می‌دانیم اعداد d رقمی هستند و هریک از ارقام k مقدار متفاوت می‌توانند داشته باشند.
- هریک از ارقام اعداد در مبنای ۱۰ می‌توانند ۱۰ مقدار متفاوت داشته باشند و ارقام اعداد در مبنای ۲ تنها ۲ مقدار متفاوت دارند.
- این الگوریتم اعداد را به ترتیب کم ارزش‌ترین رقم آنها (از رقم راست به چپ) مرتب می‌کند.

[1] radix sort

# مرتب‌سازی مبنایی

- در شکل زیر یک نمونه از این مرتب‌سازی نشان داده شده است.

| | | | | | | |
|---|---|---|---|---|---|---|
| 329 | | 720 | | 720 | | 329 |
| 457 | | 355 | | 329 | | 355 |
| 657 | | 436 | | 436 | | 436 |
| 839 | → | 457 | → | 839 | → | 457 |
| 436 | | 657 | | 355 | | 657 |
| 720 | | 329 | | 457 | | 720 |
| 355 | | 839 | | 657 | | 839 |

## مرتب‌سازی مبنایی

- برای این که این مرتب‌سازی درست عمل کند، الگوریتم مرتب‌سازی ارقام باید پایدار باشد.
- تابع Radix-Sort آرایه $A[1:n]$ را که هرکدام از عناصر آن $\mathrm{d}$ رقم دارند مرتب می‌کند. رقم اول کم‌ارزش‌ترین یعنی رقم سمت راست . رقم $\mathrm{d}$ ام پرارزش‌ترین رقم یعنی رقم سمت چپ است.

**Algorithm** Radix Sort

```
function RADIX-SORT(A, n, d)
for  i = 1 to d do
   use a stable sort to sort array A[1:n] on digit i
```

# مرتب‌سازی مبنایی

- اگرچه در الگوریتم مرتب‌سازی مبنایی الگوریتم مرتب‌سازی ارقام تعیین نشده است، اما معمولا از مرتب‌سازی شمارشی استفاده می‌شود.

## مرتب‌سازی مبنایی

- قضیه : به ازای $n$ عدد $d$ رقمی به طوری که هررقم بتواند $k$ مقدار داشته باشد، مرتب‌سازی مبنایی اعداد را در زمان $\Theta(d(n+k))$ مرتب می‌کند اگر مرتب‌سازی ارقام در زمان $\Theta(n+k)$ انجام شود.
- اثبات : هریک از ارقام در زمان $\Theta(n+k)$ مرتب می‌شود و بنابراین تعداد $d$ رقم در زمان $\Theta(d(n+k))$ مرتب می‌شود.

# مرتب‌سازی مبنایی

- اگر $d$ ثابت باشد و $k = O(n)$ باشد، مرتب‌سازی مبنایی در زمان خطی آرایه را مرتب می‌کند.

## مرتب‌سازی سطلی

- مرتب‌سازی سطلی [1] فرض می‌کند داده‌های ورودی دارای یک توزیع احتمالی یکنواخت [2] هستند.
- مرتب‌سازی سطلی بازه $[0, 1]$ را به $n$ زیربازه مساوی که هرکدام یک سطل [3] نامیده می‌شوند تقسیم می‌کند و $n$ عدد ورودی را در این سطل‌ها توزیع می‌کند. از آنجایی که داده‌های ورودی با توزیع احتمالی یکنواخت پراکنده شده‌اند، تعداد اعدادی که در یک سطل قرار می‌گیرند زیاد نخواهد بود. برای مرتب‌سازی اعداد به ترتیب از زیربازه اول شروع می‌کنیم و اعداد درون بازه‌ها را مرتب می‌کنیم و بدین ترتیب همه اعداد مرتب می‌شوند.

[1] bucket sort
[2] uniform distribution
[3] bucket

# مرتب‌سازی سطلی

- تابع Bucket-Sort در زیر نشان داده شده است. این تابع آرایه $A[1:n]$ را که هر یک از عناصر $A[i]$ در محدوده $0 \leqslant A[i] < 1$ است را دریافت می‌کند. این تابع از یک آرایه اضافی $B[0:n-1]$ که سطل‌ها هستند استفاده می‌کند.

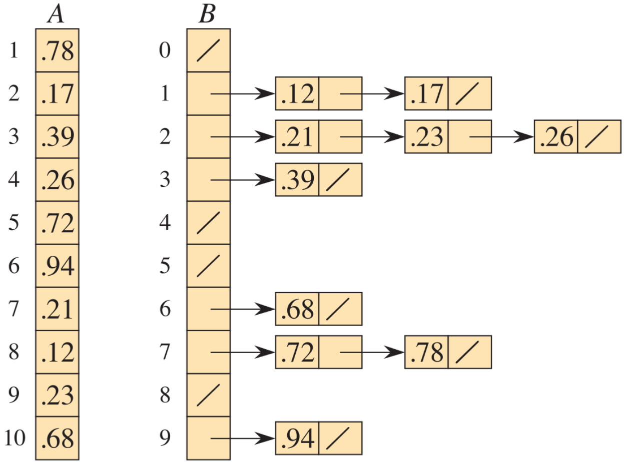
# مرتب‌سازی سطلی

**Algorithm** Bucket Sort

```
function BUCKET-SORT(A, n)
let B[0:n-1] be a new array
for i = 0 to n-1 do
    make B[i] an empty list
for j = 1 to n do
    insert A[i] into list B[⌊ n · A[i] ⌋]
for i = 0 to n-1 do
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n-1] together in order
return the concatenated lists
```

# مرتب‌سازی سطلی

- شکل زیر عملیات مرتب‌سازی سطلی را برای یک آرایه شامل ۱۰ عنصر نشان می‌دهد.



# مرتب‌سازی سطلی

- برای اثبات درستی این الگوریتم، دو عنصر $A[i]$ و $A[j]$ را در نظر بگیرید. بدون از دست دادن عمومیت اثبات، فرض کنید $A[i] \leqslant A[j]$ . از آنجایی که $\lfloor n.A[i] \rfloor \leqslant \lfloor n.A[j] \rfloor$ ، یا عنصر $A[i]$ در همان سطل $A[j]$ قرار می‌گیرد و یا به یک سطل با اندیس کمتر وارد می‌شود. اگر $A[i]$ و $A[j]$ به یک سطل یکسان بروند، حلقه خطوط ۶ و ۷ ترتیب درست را به دست می‌آورد و اگر در دو سطل متفاوت وارد شوند، خط ۸ آنها را در ترتیب درست قرار می‌دهد. پس الگوریتم درست عمل می‌کند.

- برای تحلیل زمان اجرای این الگوریتم، مشاهده کنید که همه خطوط به غیر از خط ۷ در زمان $O(n)$ اجرا می‌شوند. بنابراین باید زمان اجرای $n$ فراخوانی تابع مرتب‌سازی درجی را در خط ۷ را بیابیم.

# مرتب‌سازی سطلی

- فرض کنید $n_i$ یک متغیر تصادفی باشد که تعداد عناصر در سطل $B[i]$ را نشان می‌دهد. مقدار مورد انتظار برای متغیر تصادفی $n_i$ برابر است با ۱ ، بنابراین داریم $E[n_i] = 1$ .

- از آنجایی که مرتب‌سازی درجی در زمان مربعی اجرا می‌شود، زمان اجرای مرتب‌سازی سطلی برابر است با

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

- با محاسبه این عبارت به دست می‌آوریم $T(n) = \Theta(n)$ .