داده ساختارها

بخش چهارم

برای پیاده سازی داده ساختارها به کدهای ارائه شده در کلاس مراجعه کنید.

صف اولویت یا هرم

یک درخت دودویی کامل (complete binary tree)

برگ‌های سطح آخر آن از سمت چپ چیده شده‌اند.

کلید هر عنصر از کلیدهای فرزندانش کوچک‌تر نیست.

به این داده‌ساختار درخت نیمه‌مرتب (Partially Ordered Tree)، هرم بیشینه (max-heap)، یا max-priority queue نیز می‌گویند

متناظر با هرم کمینه (min-heap) است.

داده‌ساختاری برای اعمال

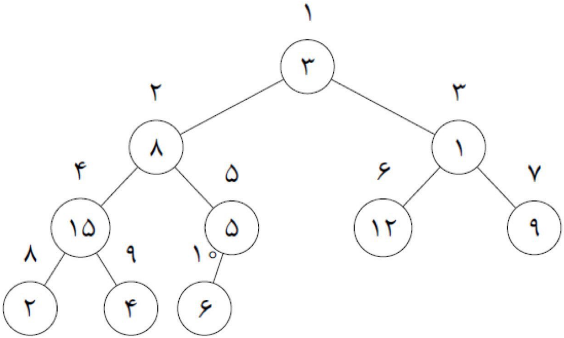
- درج
- حذف بزرگ‌ترین (کوچک‌ترین) عنصر
- افزایش (کاهش) مقدار کلید یک عنصر

همه در $\mathcal{O}(\lg n)$

پیاده سازی هرم با آرایه

$A =$

۳	۸	۱	۱۵	۵	۱۲	۹	۲	۴	۶



یک آرایه و هرم متناظر با آن

- آرایه‌ی `A[1..n]`
- ریشه در `A[1]`
- فرزند چپ عنصر iام در `A[2i]` (اگر $۲i \leq n$)
- فرزند راست آن در `A[2i+1]` (اگر $۲i + ۱ \leq n$)
- پدرش در $\mathtt{A}[\lfloor \frac{\mathtt{i}}{2} \rfloor]$

function $\textsc{Parent}(i)$
    return $\left\lfloor \frac{i}{2} \right\rfloor$
function $\textsc{LeftChild}(i)$
    return $2i$
function $\textsc{RightChild}(i)$
    return $2i + 1$

هرم بیشینه

- ریشه بزرگ‌ترین عنصر است.
- ارتفاع یک هرم (بیشینه یا کمینه) با n عنصر $\lfloor \lg n \rfloor$ است.
- اعمال درج، حذف بزرگ‌ترین و افزایش کلید از مرتبه‌ی $\lg n$ انجام می‌شوند.

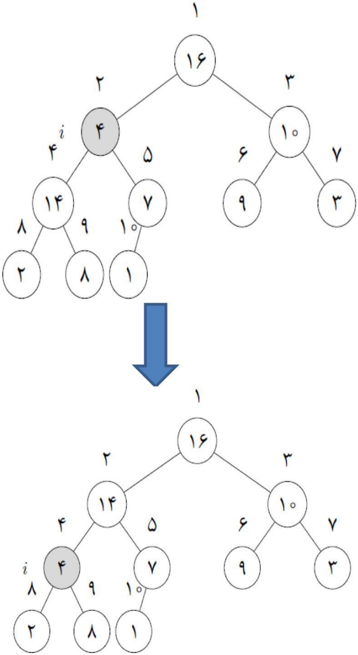
الگوریتم مرتب‌سازی هرمی نیاز به الگوریتمی دارد که ابتدا با استفاده از آن بتوان یک هرم را به یک هرم بیشینه تبدیل کرد. پس ابتدا لازم است با الگوریتم‌های زیر آشنا شویم:

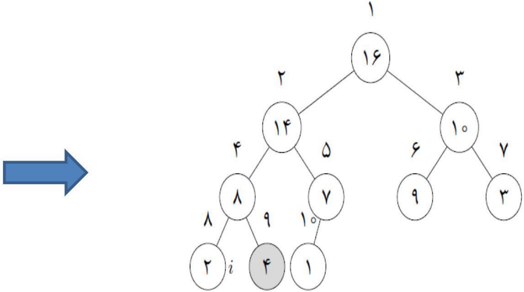
الگوریتم MaxHeapify

```
function MaxHeapify(array A, i)
    l ← LeftChild(i)
    r ← RightChild(i)
    if l ≤ n and A[l] > A[i] then
        largest ← l
    else
        largest ← i
    if r ≤ n and A[r] > A[i] then
        largest ← r
    if largest ≠ i then
        Exchange A[i] with A[largest]
        MaxHeapify(A,largest)
```

الگوریتم MaxHeapify یک آرایه A و اندیس i را به عنوان ورودی می‌گیرد و فرض می‌کنیم زیر درخت‌های چپ و راست گره i خود هرم بیشینه باشند. اگر $A[i] \leq A[\textsc{LeftChild}(i)]$ یا $A[i] \leq A[\textsc{RightChild}(i)]$ باشد به $A[i]$ اجازه می‌دهد که در هرم بیشینه به سمت پایین حرکت کند تا ویژگی هرم بیشینه حفظ شود.

اجرای MaxHeapify





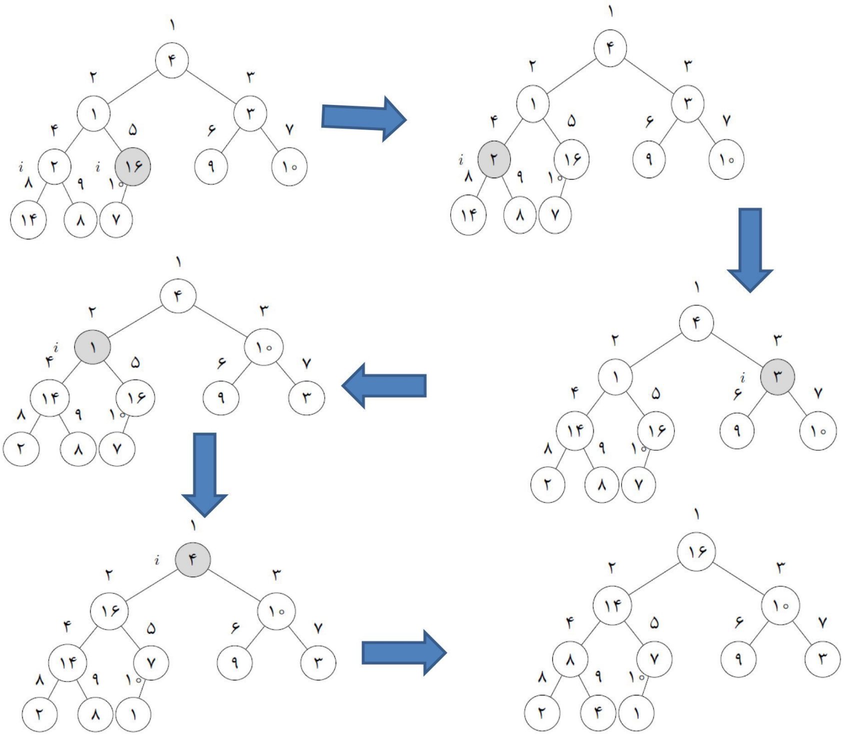
با استفاده از MaxHeapify می‌توانیم آرایه $A[۱ \cdots n]$ را به یک هرم بیشینه تبدیل کنیم. الگوریتم BuildMaxHeap از گره‌های داخلی (غیر برگ) درخت می‌گذرد و برروی هر گره MaxHeapify را اجرا می‌کند. دقت کنید که عنصرهای زیر آرایه $A[\lfloor \frac{n}{۲} \rfloor + ۱ \cdots n]$ تماماً برگ‌های درخت هستند که خود هرم بیشینه هستند. بنابراین نیاز به اجرای الگوریتم MaxHeapify روی آن‌ها نیست.

ساختن هرم

```
function BuildMaxHeap(array A[1 ··· n])
    for i = ⌊n/2⌋ downto 1 do
        MaxHeapify(A, i)
```

آرایه زیر داده شده است. یک هرم بسازید.

۴	۱	۳	۲	۱۶	۹	۱۰	۱۴	۸	۷



مرتب‌سازی هرمی (heapsort)

```
BUILD-HEAP(A)
for i ← ⌊length[A]/2⌋ downto 1
    do HEAPIFY(A, i)
```

```
HEAPSORT(A)
BUILD-HEAP(A)
for i ← length[A] downto 2
    do swap(A[1], A[length[A]])
       length[A] ← length[A] − 1
       HEAPIFY (A, 1)
```

مثال

	1	2	3	4	5	6	7	8	9	10
SWAP(1,10)	۱۸	۹	۱۸	۶	۷	۱	۹	۳	۵	۷
HEAPIFY(1,9)	۷	۹	۱۸	۶	۷	۱	۹	۳	۵	۱۸
SWAP(1,9)	۱۸	۹	۹	۶	۷	۱	۷	۳	۵	۱۸
HEAPIFY(1,8)	۵	۹	۹	۶	۷	۱	۷	۳	۱۸	۱۸
SWAP(1,8)	۹	۷	۹	۶	۵	۱	۷	۳	۱۸	۱۸
HEAPIFY(1,7)	۳	۷	۹	۶	۵	۱	۷	۹	۱۸	۱۸
SWAP(1,7)	۹	۷	۳	۶	۵	۱	۷	۹	۱۸	۱۸
HEAPIFY(1,6)	۷	۷	۳	۶	۵	۱	۹	۹	۱۸	۱۸
SWAP(1,6)	۷	۷	۳	۶	۵	۱	۹	۹	۱۸	۱۸
HEAPIFY(1,5)	۱	۷	۳	۶	۵	۷	۹	۹	۱۸	۱۸
SWAP(1,5)	۷	۶	۳	۱	۵	۷	۹	۹	۱۸	۱۸
HEAPIFY(1,4)	۵	۶	۳	۱	۷	۷	۹	۹	۱۸	۱۸
SWAP(1,4)	۶	۵	۳	۱	۷	۷	۹	۹	۱۸	۱۸
HEAPIFY(1,3)	۱	۵	۳	۶	۷	۷	۹	۹	۱۸	۱۸
SWAP(1,3)	۵	۱	۳	۶	۷	۷	۹	۹	۱۸	۱۸
HEAPIFY(1,2)	۳	۱	۵	۶	۷	۷	۹	۹	۱۸	۱۸
SWAP(1,2)	۳	۱	۵	۶	۷	۷	۹	۹	۱۸	۱۸
	۱	۳	۵	۶	۷	۷	۹	۹	۱۸	۱۸

درهم سازی

آدرس‌دهی مستقیم

فرض کنید می‌خواهیم اطلاعات ۱۰۰۰۰ دانشجو را ذخیره کنیم. همچنین فرض کنید شماره‌های دانشجویی، اعدادی بین ۱ تا یک میلیون هستند.

آرایه‌ای مانند T با یک میلیون خانه و برای هر شماره‌ی دانشجویی یک خانه از T را در نظر می‌گیریم.

DIRECT-ADDRESS-SEARCH (T, k)

Return $T[k]$

DIRECT-ADDRESS-INSERT (T, x)

$T[x.key] \longleftarrow x$

DIRECT-ADDRESS-DELETE (T, x)

$T[x.key] \longleftarrow null$

آدرس‌دهی مستقیم

- اگر کلیدها متمایز و $K \subseteq \{۰, ۱, \cdots, m-۱\}$،

$\Longleftarrow$ آرایه‌ی $T[۰..m-۱]$

$$T[k] = \begin{cases} x & \text{if } k \in K \text{ and } key[x] = k \\ \textbf{null} & \text{otherwise} \end{cases}$$

- اعمال در $\mathcal{O}(۱)$

داده‌ساختاری به نام جدول درهم‌سازی

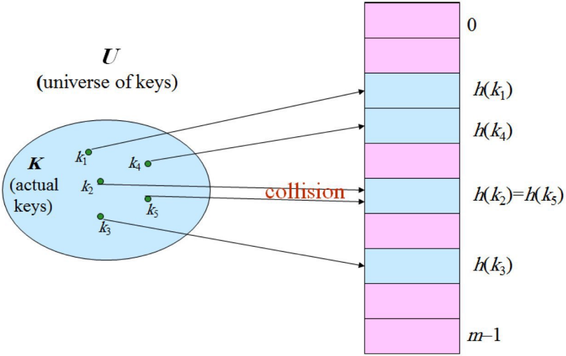
ایده‌ی کلی بر این اساس است که به جای در نظر گرفتن خود کلیدها، برای آدرس‌دهی از تابعی از کلیدها استفاده و هر داده را در اندیس متناظر با مقدار آن تابع ذخیره کنیم. حال اگر برد این تابع از مرتبه‌ی تعداد داده‌ها باشد و این تابع روی کلیدهای مورد نظر به صورت "تقریبا" یک‌به‌یک عمل کند، تمامی اطلاعات را می‌توان در آرایه‌ای به اندازه‌ی داده‌ها ذخیره کرد که مطلوب‌ترین حالت ممکن است.

تابع درهم‌سازی (hashing function)

$$h : U \to \{0, 1, \dots, m-1\}$$

تعریف: اگر دو عنصر توسط تابع درهم‌سازی به یک درایه از جدول متناظر شوند، می‌گوییم یک برخورد (collision) روی داده است.

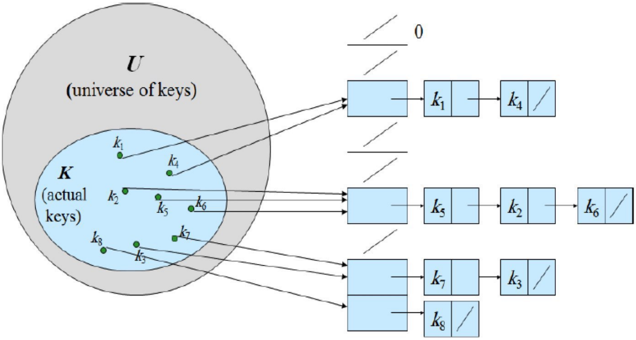
برخورد



در این‌جا کلیدهای $k_۵$ و $k_۲$ برخورد دارند چون تحت h به یک درایه نگاشته می‌شوند.

روش زنجیرهای(Chaining) برای حل مشکل برخورد

یکی از راه‌حل‌های رفع مشکل برخورد، ذخیره‌ی اندیس‌هایی که برخورد کرده‌اند در یک لیست پیوندی است. درایه‌های خالی جدول هم لیست‌های تهی ($null$) هستند. به چنین جدولی "جدول درهم‌سازی با روش زنجیره‌ای" می‌گوییم.



روش زنجیره‌ای (اعمال)

CHAINED-HASH-INSERT (T, x)
insert x at the head of the list $T[h(x.k)]$

CHAINED-HASH-SEARCH (T, k)
search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE (T, x)
delete x from list $T[h(x.k)]$

$\Theta(۱)$

- فرض: جدول درهم‌سازی (T) به‌اندازه‌ی m، n عنصر را ذخیره می‌کند.
- تعریف: α ضریب بارگذاری (load factor) T، یعنی $\alpha = \frac{n}{m}$
- α میانگین تعداد عناصری است که در یک زنجیره قرار می‌گیرند.

- بدترین حالت: n عنصر در یک درایه قرار گیرند.
- بدترین زمان برای جست‌وجو:

$\mathcal{O}(n)$ به‌علاوه زمان لازم برای محاسبه تابع درهم‌سازی.

حال برای مثال میانگین زمان جست‌وجو را محاسبه می‌کنیم. بدترین حالت زمانی اتفاق می‌افتد که کلید k که به دنبال آن هستیم ذخیره نشده باشد. در این صورت باید کل لیست $h(k)$ را پیمایش کنیم. بنابر تعریف، طول این لیست به‌طور متوسط برابر است با عامل بار:

$$E[n_{h(k)}] = \alpha$$

پس زمان جست‌وجو در حالت ناموفق از مرتبه‌ی $\Theta(۱+\alpha)$ است. زمان ۱ برای این است که ببینیم کلید در کدام خانه‌ی

حافظه است و در زمان α باید آرایه‌ی مربوط به آن خانه را جست‌وجو کنیم. در حالت جست‌وجوی موفق نیز با همین استدلال و با توجه به این که کلید k به طور متوسط در فاصله‌ی $\frac{\alpha}{۲}$ از ابتدای لیست قرار دارد، زمان جست‌وجوی از مرتبه‌ی $\Theta(۱+\frac{\alpha}{۲})$ است.

در مورد زمان درج و حذف اگر از لیست پیوندی دوسویه استفاده کنیم، هر دو عمل در $\Theta(۱)$ انجام می‌شوند. بنابراین در مجموع اگر $\alpha = O(۱)$ آن‌گاه همان‌طور که انتظار داشتیم هر سه عمل مورد نظر در $\Theta(۱)$ انجام خواهند شد.

توابع درهم‌سازی

- یک تابع درهم‌سازی خوب باید یک‌نواخت و ساده simlpe uniform hashing
- بسیاری از توابع درهم‌سازی فرض می‌کنند کلید عناصر اعداد طبیعی هستند.
- بنابراین اگر کلید عناصری که می‌خواهیم ذخیره کنیم طبیعی نباشد باید روشی برای متناظر کردن آن‌ها با اعداد طبیعی بیابیم.
- برای مثال، اگر کلید عناصر رشته‌هایی از نویسه‌ها باشد، می‌توانیم کلید را مجموع کد اسکی نویسه‌های آن رشته در نظر بگیریم.

- $h(k) = k \bmod m$.
- باید از انتخاب مقادیر خاصی برای m اجتناب کنیم.
- مثلاً m نباید توانی از ۲ باشد، زیرا اگر $m = ۲^p$، $h(k)$ معادل p بیت کم‌ارزش k است.
- به‌تر است از مقادیر همه‌ی بیت‌ها برای محاسبه‌ی $h(k)$ استفاده شود.
- به‌طور کلی یک عدد اول که نزدیک به توانی از ۲ نباشد انتخاب مناسبی برای m است.

آدرس دهی باز

ایده‌ی کلی آن است که اگر برخورد رخ داد، به خانه‌ی بعدی برویم و همین روند را ادامه دهیم تا در نهایت به یک خانه‌ی خالی برسیم و یا این‌که کل آرایه پر شده باشد.

در آدرس‌دهی باز هر عنصر در یک درایه جدول ذخیره می‌شود

درج

برای مشخص کردن این‌که کدام درایه‌ها باید بررسی شوند تابع درهم‌سازی را به این صورت تعریف می‌کنیم:

$$h : U * \{0, 1, ..., m-1\} \rightarrow \{0, 1, ..., m-1\}$$

در آدرس‌دهی باز برای هر کلید k نیاز به چک کردن متوالی درایه‌های زیر داریم:

$$< h(k, 0), h(k, 1), ..., h(k, m-1) >$$

حال برای درج داده با کلید k کافی است با همین ترتیب جدول را وارسی و داده را در اولین خانه‌ی خالی ذخیره کنیم. به طریق مشابه، برای جست‌وجوی داده با کلید k دوباره با همین ترتیب خانه‌ها را وارسی می‌کنیم. اگر کلید k را یافتیم که جست‌وجو موفق بوده است و اگر به خانه‌ای خالی رسیدیم، دیگر نیازی به ادامه‌ی وارسی نیست و جست‌وجو ناموفق بوده است. در زیر شبه‌کدهای متناظر با آن‌چه گفته شد، آورده

```
HASH-INSERT (T, k)
i ← 0
repeat
   j ← h(k, i)
   if T[j] = null
      T[j] ← k
      return j
   else i ← i + 1
until i = m
error "hash table overflow"
```