ساختمان داده ها

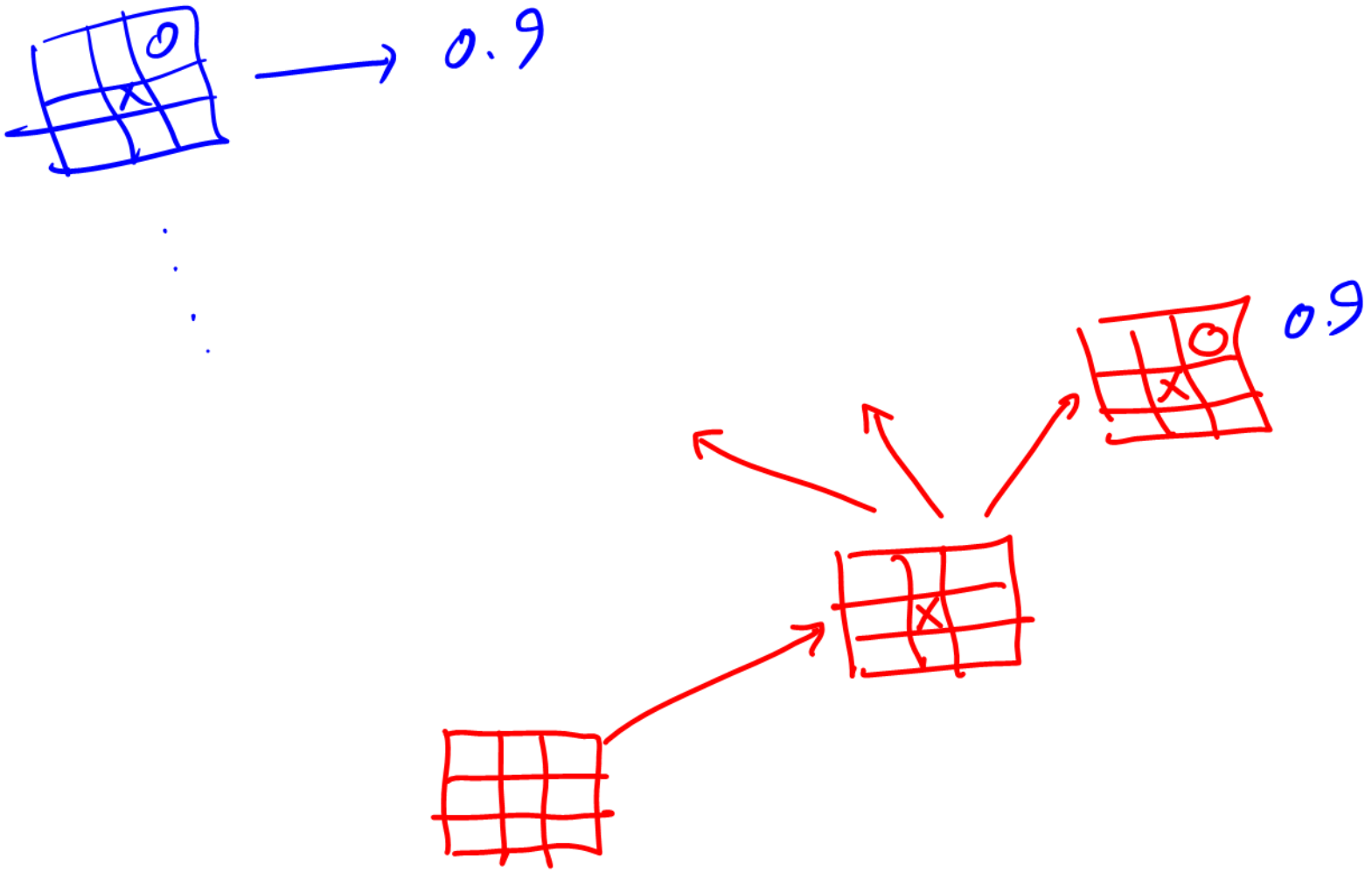
جدول درهم ریزی
(Hash Table)

مدرس: غیاثی‌شیرازی

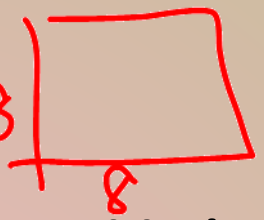
دانشگاه فردوسی مشهد

مثال هایی از کاربرد جدول درهم ریزی

- نگه داشتن مجموعه ای از قوانین برای هر آدرس ip
- Symbol Table در کامپایلر
  - کامپایلر برای هر id ای که مشاهده می کند، یک مدخل در Symbol Table باز می کند و اطلاعاتی را در مورد آن علامت (مثلا اینکه نام متغیر است یا نام کلاس است) ذخیره می کند.
- نگه داری نتایج بررسی در کاوش درخت بازی ها
  - برای هر وضعیت بازی یک مدخل درنظر گرفته می شود که حاصل تفکر در مورد آن حالت در آن ذخیره می شود. استفاده از جدول درهم ریزی مانع فکر کردن تکراری بر روی حالاتی که قبلا بررسی شده اند می شود.


0.9
0.9

ایده کلی



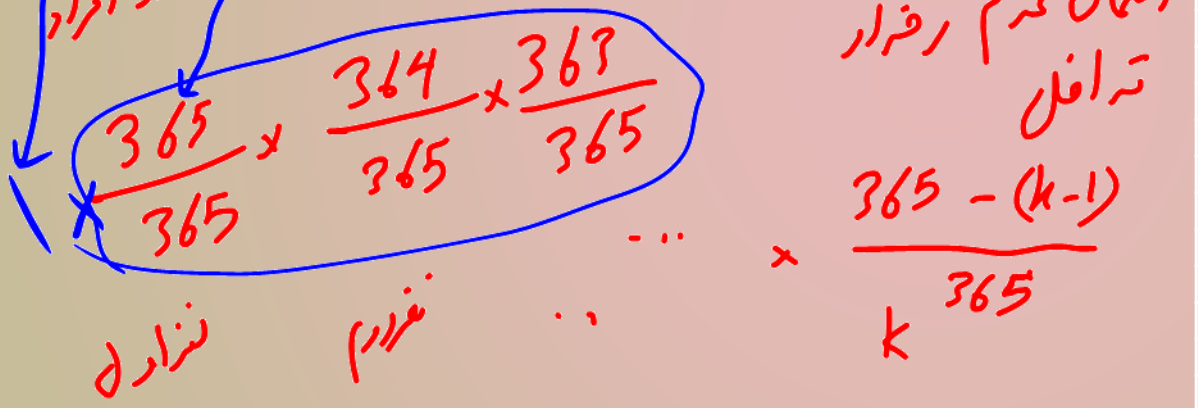
- U (از Universe) : مجموعه تمام کلیدهای ممکن
- تعداد داده هایی که در جدول درهم ریزی ذخیره می شوند بسیار بسیار کمتر از اندازه مجموعه U است.
- آرایه ای به اندازه n برای ذخیره داده ها در نظر می گیریم. به هر خانه این آرایه یک bucket گفته می شود.
- از یک تابع درهم ریزی $h: U \to \{0,1, \dots, n-1\}$ برای یافتن محل متناظر با کلید $u \in U$ استفاده می کنیم.
- داده (u,v) را در خانه h(u) ذخیره می کنیم.
- به h(u) خانه اصلی (home bucket) کلید u می گوییم.

مشکل تداخل

- مشکل تداخل:
  - اگر برای دو داده متفاوت u_1 و u_2 داشته باشیم $h(u_1)=h(u_2)$ آنگاه هر دو داده می خواهند در یک خانه آرایه ذخیره شوند.
- برای آنکه در یک جمع با احتمال ۵۰٪ روز تولد دو نفر یکسان باشد، تعداد افراد جمع باید حداقل چند نفر باشد؟

احتمال تداخل در روز تولد

```
p = 1
for k in range (0,50):
    p *= (365-k)/365
    print (k+1, 1-p)
```



نتیجه: مشکل تداخل بسیار زودتر از آنچه به نظر می رسد رخ می دهد.

n	p	n	p
1	0.000	26	0.598
2	0.003	27	0.627
3	0.008	28	0.654
4	0.016	29	0.681
5	0.027	30	0.706
6	0.040	31	0.730
7	0.056	32	0.753
8	0.074	33	0.775
9	0.095	34	0.795
10	0.117	35	0.814
11	0.141	36	0.832
12	0.167	37	0.849
13	0.194	38	0.864
14	0.223	39	0.878
15	0.253	40	0.891
16	0.284	41	0.903
17	0.315	42	0.914
18	0.347	43	0.924
19	0.379	44	0.933
20	0.411	45	0.941
21	0.444	46	0.948
22	0.476	47	0.955
23	0.507	48	0.961
24	0.538	49	0.966
25	0.569	50	0.970

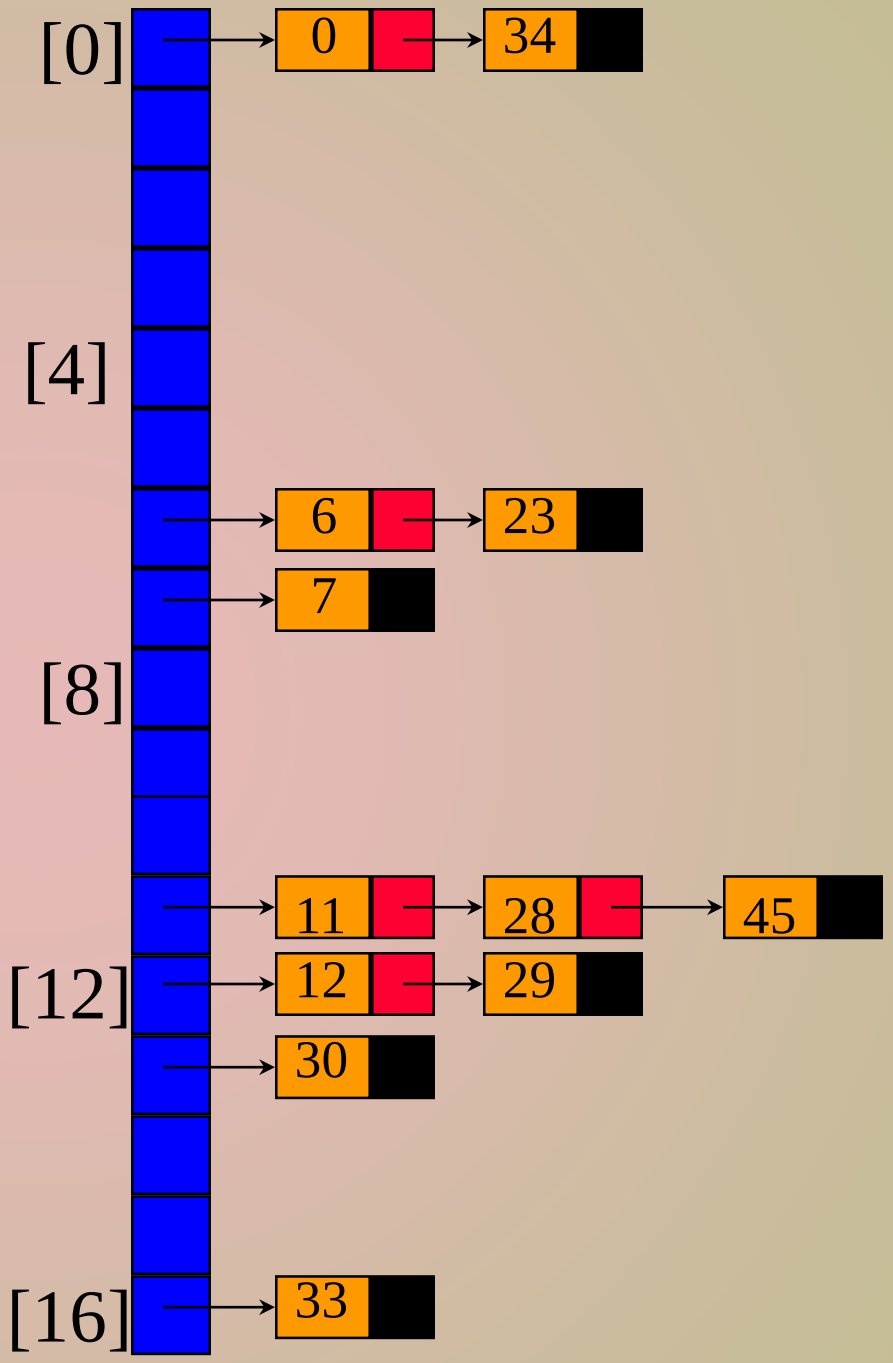
حل مشکل تداخل- راه حل ۱: Chaining

- در هر خانه آرایه یک لیست پیوندی نگه داریم.
- داده هایی که به یک خانه آرایه فرستاده می شوند را در لیست پیوندی نگهداری می کنیم.
- ممکن است برای افزایش سرعت جستجو، عناصر داخل هر لیست را مرتب کنیم.

Sorted Chains

- "Put in" pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45
- Home bucket = key % 17.

$h(k) = k \% 17$



حل مشکل تداخل- راه حل ۲: Open Addressing

- در صورت پربودن خانه اصلی عنصر u، خانه دیگری را برای آن در نظر بگیریم و این کار را تکرار کنیم.
  - Linear probing –
  - Quadratic probing –
  - Random probing –
  - Double hashing –
- http://www.cise.ufl.edu/~sahni/dsaac/enrich/c11/overflow.htm

Random Probing

Suppose that the hash table has `b` buckets. In linear open addressing the buckets are examined in the order `(f(k)+i) % b, 0 <= i < b`, where `k` is the key of the element being searched for. In random probing a pseudo-random number generator is used to obtain a random sequence `R(i), 1 <= i < b` where `R(1), R(2), ... R(b-1)` is a permutation of `[1, 2, ..., b-1]`. The buckets are examined in the order `f(k)`, `(f(k)+R(i)) % b, 1 <= i < b`.

Quadratic Probing

The bucket examination order for quadratic probing is `f(k)`, `(f(k)+i^2) % b`, `(f(k)-i^2) % b`, `1 <= i < (b-1)/2`, where `b` is the number of buckets in the table. This examination sequence covers the space of buckets whenever `b` is a prime number of the form `4j+3`, where `j` is an integer (see ``The use of quadratic residue research," by C. Radke, `Communications of the ACM`, 1970, 103-105). Some suitable values for `b` are `251` (`j = 62`), `503` (`j = 125`), and `1019` (`j = 254`).

Double Hashing

In double hashing we employ a primary hash function f_1 and a secondary hash function f_2.

The primary hash function is used to determine the home bucket $h = f_1(k)$ for the search key k.

The remaining buckets are examined using a stride $s = f_2(k)$. That is, the buckets are examined in the sequence h, $h + s$, $h + 2s$, $h + 3s$, ..., $h + (b-1)s$ (all computations are done modulo the number of buckets b).

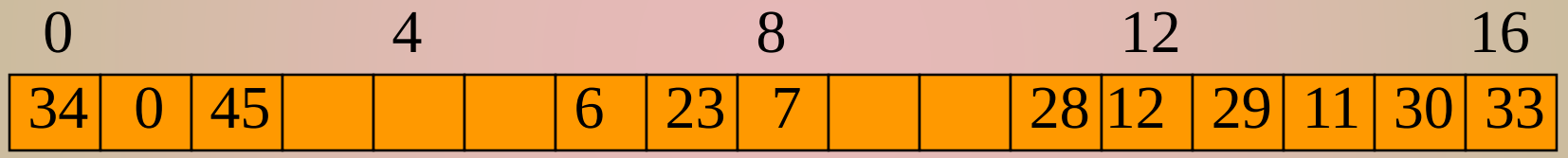
Double Hashing

When `b = 11`, we may use any of `1, 2, 3, ..., 10` as the stride. For example, if we use `s = 4`, the buckets are examined in the sequence `h, h + 4, h + 8, ..., h + 40` (modulo `11` of course). When `h = 3`, the sequence is `3, 7, 0, 4, 8, 1, 5, 9, 2, 6, 10`.

A common choice for the secondary hash function is $f_2(k) = p - (k \% p)$, where `p` is a prime number less than the number `b` of buckets in the table. Notice that $0 < f_2(k) <= p$.

Linear Probing – Get And Insert

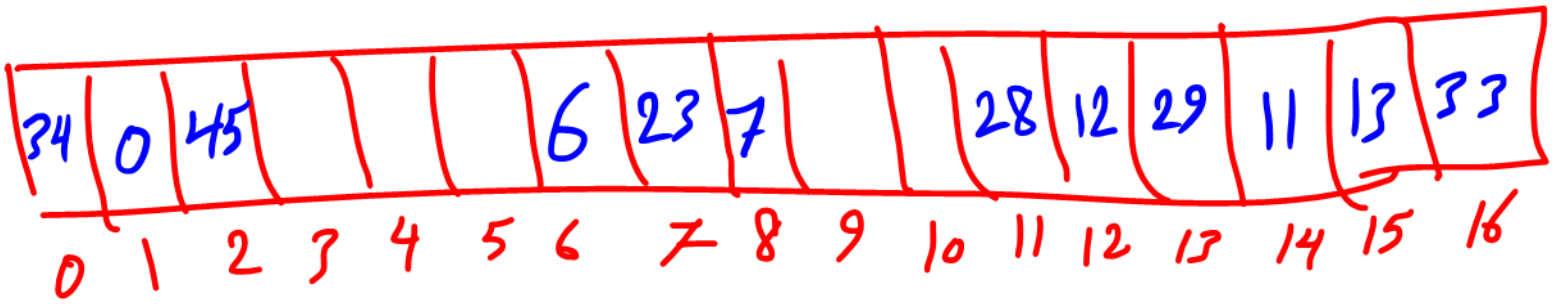
- divisor = b (number of buckets) = 17.
- Home bucket = key % 17.



- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

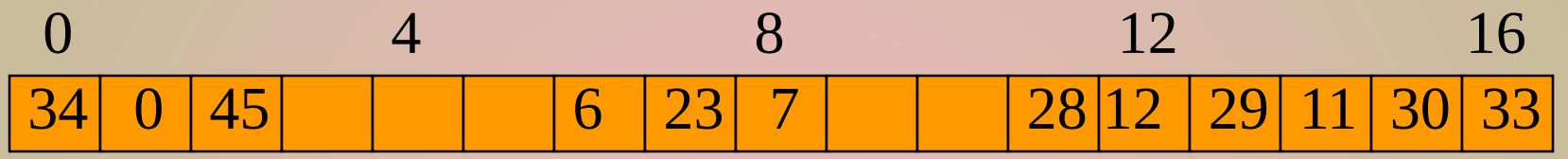
- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

$h(k) = k \% 17$

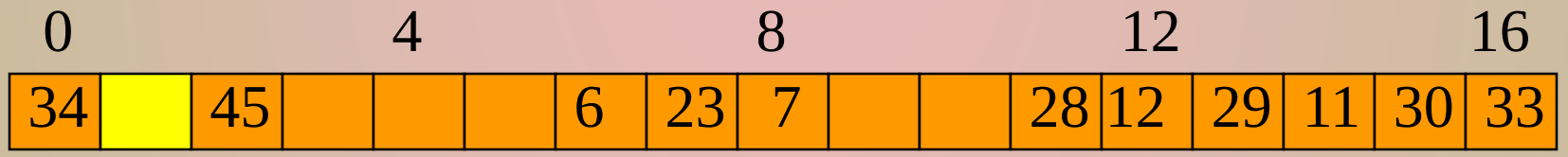


0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

Linear Probing – Delete



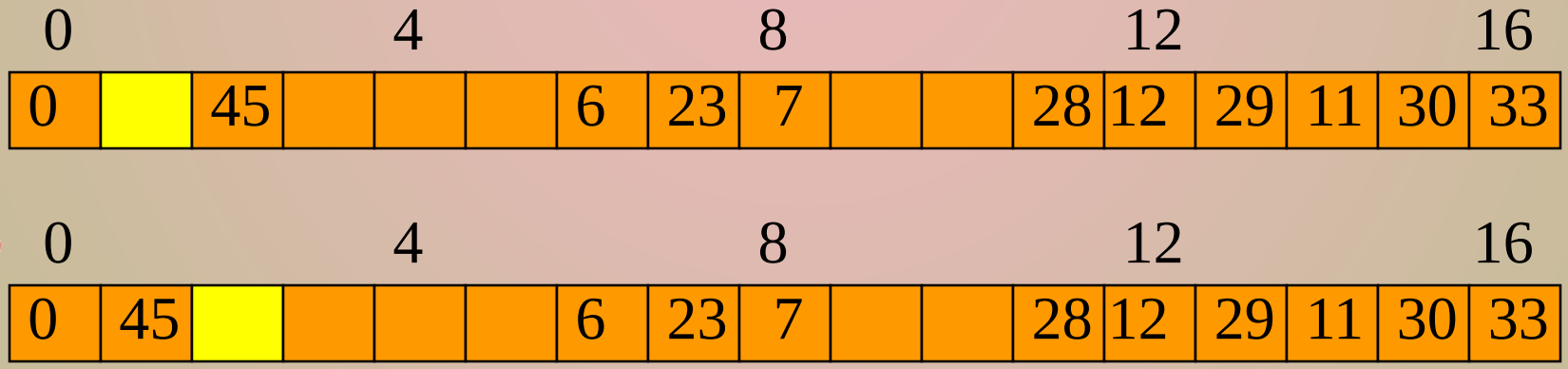
- Delete(0)



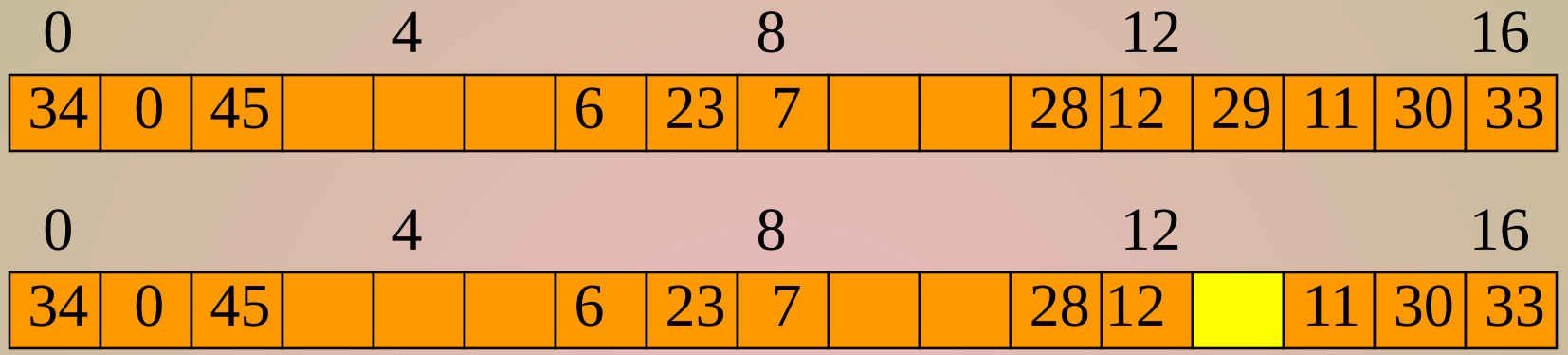
- Search cluster for pair (if any) to fill vacated bucket.

Linear Probing – Delete(34)

- Search cluster for pair (if any) to fill vacated bucket.

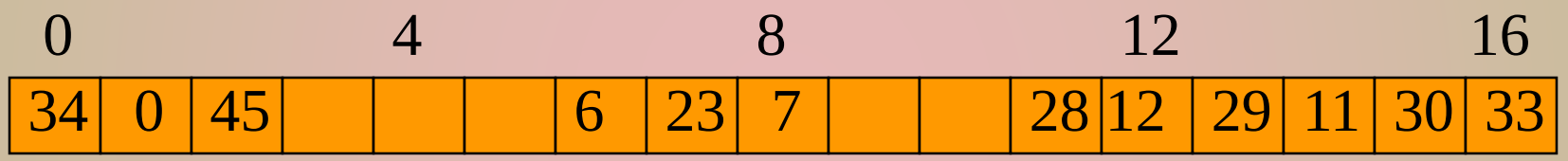


Linear Probing – Delete(29)



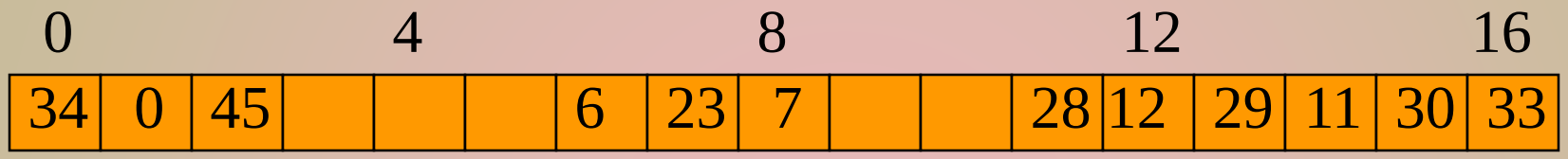
- Search cluster for pair (if any) to fill vacated bucket.

Performance Of Linear Probing



- Worst-case find/insert/erase time is $\Theta(n)$, where n is the number of pairs in the table.
- This happens when all pairs are in the same cluster.

Expected Performance



- α = loading density = (number of pairs)/b.
  - α = 12/17.
- S_n = expected number of buckets examined in a successful search when n is large
- U_n = expected number of buckets examined in a unsuccessful search when n is large
- Time to put and remove governed by U_n.

Expected Performance

- $S_n \sim \frac{1}{2}(1 + 1/(1 - \alpha))$
- $U_n \sim \frac{1}{2}(1 + 1/(1 - \alpha)^2)$
- Note that $0 <= \alpha <= 1$.

$$S_n \sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

$$U_n \sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

alpha	S_n	U_n
0.50	1.5	2.5
0.75	2.5	8.5
0.90	5.5	50.5

$\alpha <= 0.75$ is recommended.

معیارهای مناسب بودن تابع درهم ریزی

1. داده ها را به طور یکنواخت در خانه های آرایه پخش کند.
2. سرعت محاسبه آن بالا باشد.

طراحی توابع درهم ریزی ..... است

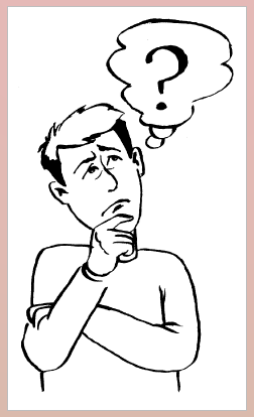
- مثالی از تابع درهم ریزی برای شماره تلفن ها:

3880 5158 ⟶ 38

- مثالی از تابع درهم ریزی برای آدرس های حافظه:

1F00B1C15 ⟶ 19

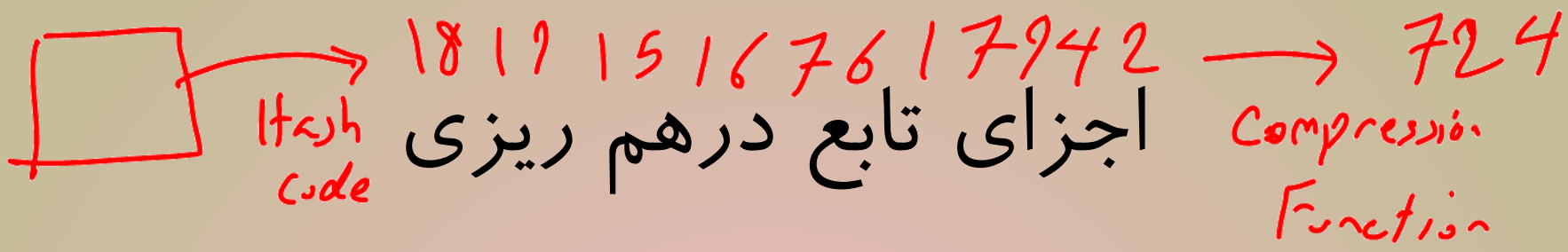
256



طراحی توابع درهم ریزی بد خیلی ساده است

- مثالی از تابع درهم ریزی بد برای شماره تلفن ها:
  - سه رقم اول شماره تلفن ها را برداریم
    - زیرا اعداد اول شماره تلفن، منطقه آن را مشخص می کنند و تعداد مشترکین مناطق مختلف یکسان نیست.
- مثالی از تابع درهم ریزی بد برای آدرس های حافظه:
  - سه رقم کم ارزش آدرس های حافظه را برداریم.
    - زیرا اکثر آدرس های حافظه زوج هستند.

۱۹
۱۸



اجزای تابع درهم ریزی

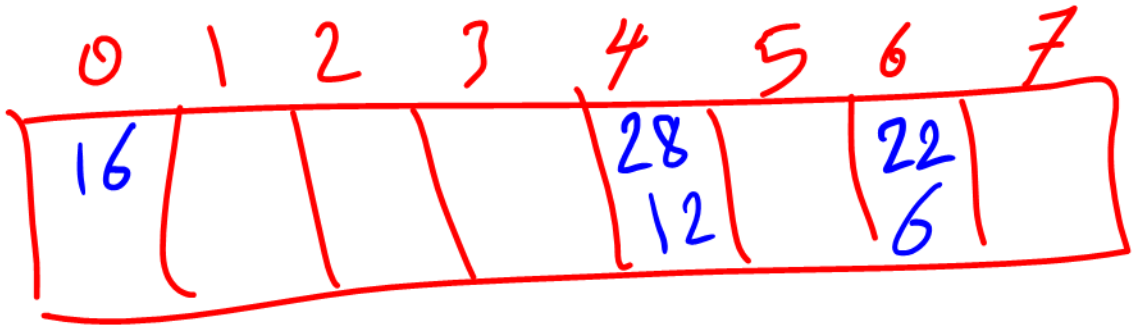
- یک تابع درهم ریزی معمولا از دو جزء تشکیل می شود:
  - Hash code: تابعی است که به هر عنصر $u \in U$ یک عدد نسبت می دهد. (توجه داریم که عناصر u می توانند هر نوعی باشند: آدرس ip، رشته متنی، وضعیت بازی شطرنج و ...)
  - تابع فشرده سازی (Compression Function): تابعی است که عدد تولید شده توسط Hash Code را (که ممکن است بسیار بزرگ باشد) به مجموعه اعداد 0 تا n-1 می نگارد.
- یک تابع فشرده سازی متداول تابع $f(m)=m \bmod n$ است.

انتخاب اندازه آرایه (n)

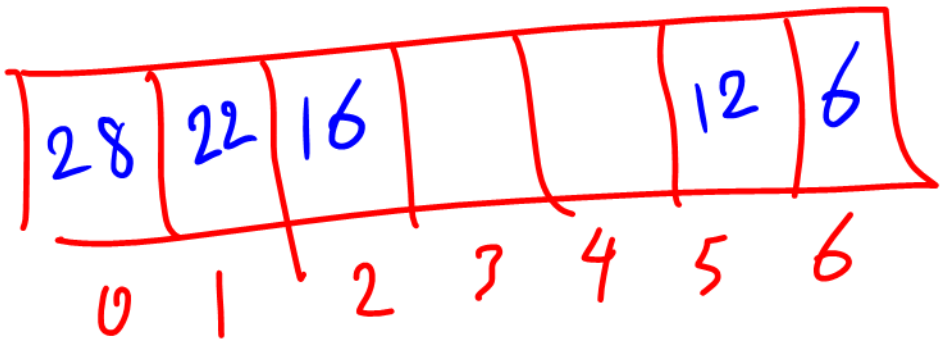
- به دلایل مختلف داده ها واقعا تصادفی نیستند و الگوهای متعددی در آنها وجود دارد.
  - برای مثال دیدیم که آدرس های حافظه از این الگو تبعیت می کنند که معمولا زوج هستند.
- اگر n مضربی از p باشد تمام داده هایی که hash code آنها با q ($0 \leq q < p$) همنهشت است، همگی به یک افراز n/p تایی از خانه های آرایه نگاشته می شوند.
- بنابراین اگر داده ها به صورت طبیعی دارای این الگو باشند که hash code آنها مضربی از p است، آنگاه تنها از n/p خانه های حافظه استفاده خواهد شد.

16, 22, 6, 28, 12 | n = 8

داده ها همه زوج باشند



داده ها همه زوج باشند n = 7



انتخاب اندازه آرایه (n)

- بنابراین ایده آل این است که اندازه جدول درهم ریزی عددی اول باشد.
  - در عمل نداشتن مقسوم علیه کوچک تر از ۲۰ کفایت می کند.
- دو نکته دیگر در انتخاب اندازه آرایه:
  - اندازه آرایه به توانی از ۲ نزدیک نباشد.
  - اندازه آرایه به توانی از ۱۰ نزدیک نباشد.

α – ضریب بار (Load factor)

- تعریف: ضریب بار یک جدول درهم ریزی (α) برابر نسبت عناصر ذخیره شده در جدول نسبت به اندازه آرایه است.
- در روش Chaining ضریب بار می تواند بزرگ تر از یک باشد. اما در Open Addressing این ضریب همواره کوچک تر مساوی ۱ است.
- برای داشتن کارایی مورد انتظار، ضریب بار باید کوچک نگه داشته شود.
  - لازم است هرگاه ضریب بار (α) به حد معینی (مثلا ۰.۷۵) می رسد, طول آرایه افزایش یابد.

STL hash_map

- Simply uses a divisor that is an odd number.
- This simplifies implementation because we must be able to resize the hash table as more pairs are put into the dictionary.
  - Array doubling, for example, requires you to go from a 1D array table whose length is b (which is odd) to an array whose length is 2b+1 (which is also odd).

Java.util.HashTable

- Simply uses a divisor that is an odd number.
- This simplifies implementation because we must be able to resize the hash table as more pairs are put into the dictionary.
  - Array doubling, for example, requires you to go from a 1D array table whose length is b (which is odd) to an array whose length is 2b+1 (which is also odd).