الباب الخامس

Hash Tables

في هذا الفصل سنتحدث عن الhash tables ، و هي واحدة من أهم الdata structures , سنرى حالات استخدامها و كيف تعمل من الداخل (implementation, collision and hash function).

لنفترض أنك تعمل في متجر، و عندما يسألك أحد الزبائن عن سعر أي منتج يجب عليك البحث عنه في كتاب الأسعار الغير مرتب، هذا سيستهلك وقتا كثيرا O(n) و لكن إذا كان الكتاب مرتبا أبجديا فسيكون الأمر أسرع بكثير و سيحتاج فقط إلى O(Log(n))، و لكن الزبون لن يكون سعيدا بهذا الانتظار حتى لو قصر، لذا قمت بحفظ الكتاب بالكامل و هكذا إذا سألت عن السعر تجيب تلقائيا في O (1).

و الآن دعنا نوضح المثال بلغة الكومبيوتر، لنفترض أنك تخزن اسم المنتج و سعره في array غير مرتب، ستحتاج O(n) للوصول للمنتج أما إذا كان مرتبا أبجديا سيحتاج O(Log(n)) و لكن إذا أردت أن يصل للمنتج في O(1) حينها ستحتاج إلى ال hash tables.

دالة التجزئة (hash function):

تقوم هذه الدالة بتحويل الكلام (strings) إلى رقم يكون هو ال Index الذي تخزن فيه ال key و ال value (اسم المنتج و سعره) و تحتاج هذه الدالة إلى بعض المقومات:

- الاتساق: يجب أن تكون القيمة الرقمية متسقة لكل key فمثلا إذا أدخلنا "تفاح" و كانت النتيجة 3 سيتم تخزين التفاح و سعره في ال index 3 و يجب أن تكون قيمة التفاح دائما ب3 لنستطيع استرجاع السعر في أي وقت
- يجب أن تكون هناك قيم رقمية مختلفة للكلمات المختلفة
- يجب أن يكون الرقم هو index موجود داخل حدود ال array.

AN EMPTY HASH TABLE

book is a new hash table. Let's add some prices to book:

```
>>> book["apple"] = 0.67 <------ An apple costs 67 cents.
>>> book["milk"] = 1.49 <------ Milk costs $1.49.
>>> book["avocado"] = 1.49
>>> print book
{'avocado': 1.49, 'apple': 0.67, 'milk': 1.49}
```

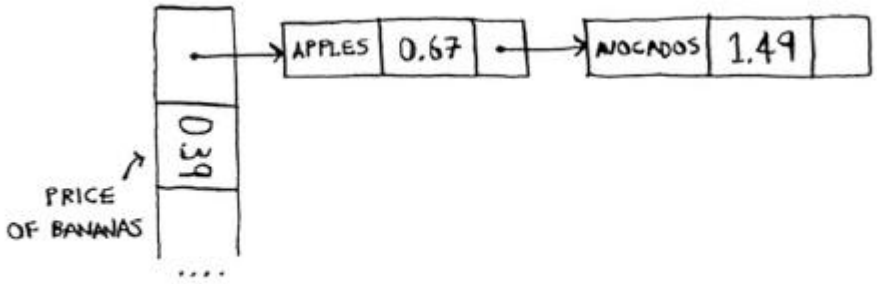
A HASH TABLE OF PRODUCE PRICES

حالات الاستخدام:

- البحث : مثال على ذلك جهات الاتصال في الهاتف، إذا أردت إضافة جهة اتصال فأنت تدخل الاسم (key) ثم الرقم الخاص به (value) ، و يمكنك بعد ذلك الوصول للرقم بسهولة من خلال البحث عن الاسم.
- DNS resolution : تحويل ال web address إلى IP address.
- منع القيم المكررة : يجب على قيم ال key أن تكون مميزة، و يمكننا التأكد إذا كان هذا الkey موجودا أم لا من خلال البحث عن وجود value خاصة به.
- Cashing : في مواقع الويب إذا كان هناك صفحة يتطلب الوصول إليها بشكل متكرر و لا تتغير من مستخدم لآخر فمن الأفضل الوصول إليها من خلال ال hash table بدلا من السيرفر لأن هذا يكون أسرع و يقلل من الضغط على السيرفر.

الصدام (collision) :

ذكرنا بأن الكلمات المختلفة تنتج قيم رقمية مختلفة و لكن هذا في الحالة المثالية فقط، في الواقع لا يوجد hash function قادرة على فعل هذا. فلنفترض مثلا أن لدينا array حجمه 28 و يتم تخزين كل key بناء على الحرف الأول في الكلمة، فإذا أضفنا "تفاح" تخزن في index 3 و لكن ماذا لو أضفنا بعد ذلك "تين"؟ سيتم استبدال التفاح و سعره بالتين و ها ما يسمى بالcollision. هناك عدة حلول لهذه المشكلة أبسطها إضافة linked list في الindex الذي يحدث فيه الصدام.



في هذه الحالة فالوصول إلى "موز" لا يزال سريعا و لكن الوصول ل"تفاح" أو "تين" أبطأ قليلا و لكن ماذا لو كان لدينا العديد من المنتجات بحرف ال"ت"؟ سيصبح الأمر أبطأ بكثير و لكن ماذا لو كانت كل المنتجات بحرف ال"ت"؟ سيصبح الhash بأكمله فارغا إلا Index 3. لذا يجب على الhash function بتوزيع القيم على الhash بشكل متساوي ة ألا تجعل الlinked list تصبح أطول من اللازم.

الأداء (Performance):

	HASH TABLES (AVERAGE)	HASH TABLES (WORST)	ARRAYS	LINKED LISTS
SEARCH	O(1)	O(n)	O(1)	O(n)
INSERT	O(1)	O(n)	O(n)	O(1)
DELETE	O(1)	O(n)	O(n)	O(1)

في الحالة المتوسطة فإن الhash table سريعة مثلا الarray في البحث و مثل ال linked list في الإضافة و الحذف و لكن في الحالة الأسوأ تكون أبطأ منهم جميعا لذا يجب أن نحاول الابتعاد عنه عن طريق تفادي ال collision من خلال:

- الحفاظ عل قيمة load factor منخفضة.
- وجود hash function جيدة.

Loading Factor:

Loading factor = number of occupied slots in an array/ number of total slots.

- إذا كانت القيمة أقل من 1 فلا يزال هناك أماكن في الarray
- إذا كانت القيمة 1 أو أكثر فالarray ممتلئ و نحتاج النقل(resize): نقوم بعمل array جديد في الغالب ضعف حجم الأول ثم نعيد إدخال القيم فيه من خلال ال has function من الأفضل القيام بها عند load factor = 0.7 .