UNIT 3: HASHING

<u>L8. HASHING WITH CHAINING</u>

DICTIONARY PROBLEM

Abstract Data Type (ADT) — maintain a set of items, each with a key, subject to

- insert(item): add item to set
- delete(item): remove item from set
- search(key): return item with key if it exists

We assume items have distinct keys (or that inserting new one clobbers old).

Goal: $O(1)$ time per operation.

Python Dictionaries

Items are (key, value) pairs e.g. d = {`algorithms': 5, `cool': 42}

<u>Python set is really dict where items are keys (no values)</u>

MOTIVATION

Dictionaries are perhaps the most popular data structure in CS

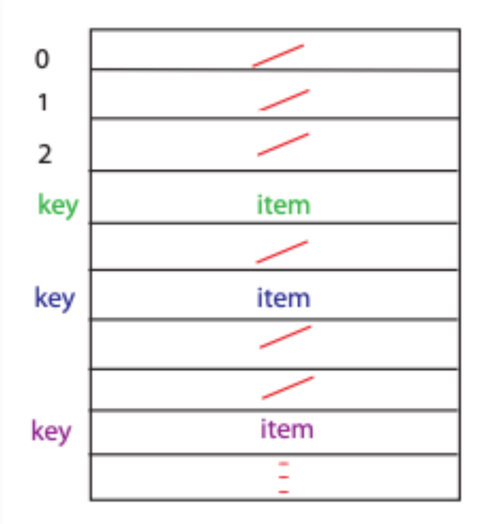
- built into most modern programming languages (Python, Perl, Ruby, JavaScript, Java, C++, C#, . . .)
- e.g. best docdist code: word counts & inner product
- implement databases: (DB HASH in Berkeley DB)
- compilers & interpreters: names → variables
- network routers: IP address → wire
- network server: port number → socket/app.
- virtual memory: virtual address → physical

Less obvious, using hashing techniques:

- substring search (grep, Google) (L9)
- string commonalities (DNA) (PS4)
- file or directory synchronization (rsync)
- cryptography: file transfer & identification (L10)

HOW DO WE SOLVE THE DICTIONARY PROBLEM?

Simple Approach: Direct Access Table



Problems

1. keys must be nonnegative integers (or using two arrays, integers)
2. large key range $\Longrightarrow$ large space - e.g. one key of 2^{256} is bad news.

2 SOLUTIONS

Solution to 1 : “prehash” keys to integers.

- In theory, possible because keys are finite $\Longrightarrow$ set of keys is countable
- In Python: $\underline{\ \mathrm{hash}\ }$ (object) (actually hash is misnomer should be "prehash") where object is a number, string, tuple, etc. or object implementing - hash_- - (default = id $=$ memory address)
- In theory, $x = y \Leftrightarrow \mathrm{hash}(x) = \mathrm{hash}(y)$
- Python applies some heuristics for practicality: for example, $\mathrm{hash}(\backslash 0B') = 64 = \mathrm{hash}('\backslash 0\backslash 0C')$
- Object's key should not change while in table (else cannot find it anymore)
- No mutable objects like lists

Solution to 2 : hashing

- Reduce universe $\mathcal{U}$ of all keys (say, integers) down to reasonable size m for table
- idea: $m \approx n = \#$ keys stored in dictionary
- hash function $\mathrm{h} : \mathcal{U} \rightarrow \{0, 1, \ldots, m-1\}$
- two keys $k_i, k_j \in K$ collide if $h(k_i) = h(k_j)$

How do we deal with collisions?

1. Chaining: TODAY
2. Open addressing: L10

CHAINING

Linked list of colliding elements in each slot of table

- Search must go through whole list $\mathrm{T}[\mathrm{h}(\mathrm{key})]$
- Worst case: all n keys hash to same slot $\Longrightarrow \Theta(n)$ per operation

Simple Uniform Hashing

An assumption (cheating): Each key is equally likely to be hashed to any slot of table, independent of where other keys are hashed.

$$\begin{aligned} \text{let } n &= \# \text{ keys stored in table} \\ m &= \# \text{ slots in table} \\ \text{load factor } \alpha &= n/m = \text{ expected } \# \text{ keys per slot} \end{aligned}$$

HASH FUNCTIONS

Division Method

$$h(k) = k \bmod m$$

This is practical when m is prime but not too close to power of 2 or 10 (then just depending on low bits/digits).

But it is inconvenient to find a prime number, and <u>division</u> is slow.

Multiplication Method

$$h(k) = [(a \cdot k) \bmod 2^w] \gg (w - r)$$

where a is random, k is w bits, and $m = 2^r$.
This is practical when a is odd $\& 2^{w-1} < a < 2^w$ & a not too close to 2^{w-1} or 2^w.

<u>Multiplication and bit extraction</u> are faster than division.

Universal Hashing

For example: $h(k) = [(ak + b) \bmod p] \bmod m$ where a and b are random $\in \{0, 1, \ldots p - 1\}$, and p is a large prime ($> |\mathcal{U}|$). This implies that for worst case keys $k_1 \neq k_2$, (and for a, b choice of h):

$$\mathrm{Pr}_{a,b}\{\text{ event } X_{k_1 k_2}\} = \mathrm{Pr}_{a,b}\{h(k_1) = h(k_2)\} = \frac{1}{m}$$

THINKING

本讲开始介绍字典，又称为关联数组（associative array）或映射（map）。

字典的主要特性为置入、删除和搜索都只需要常数时间。因此有大量的应用。

本讲从最基本的数据结构开始构建字典。

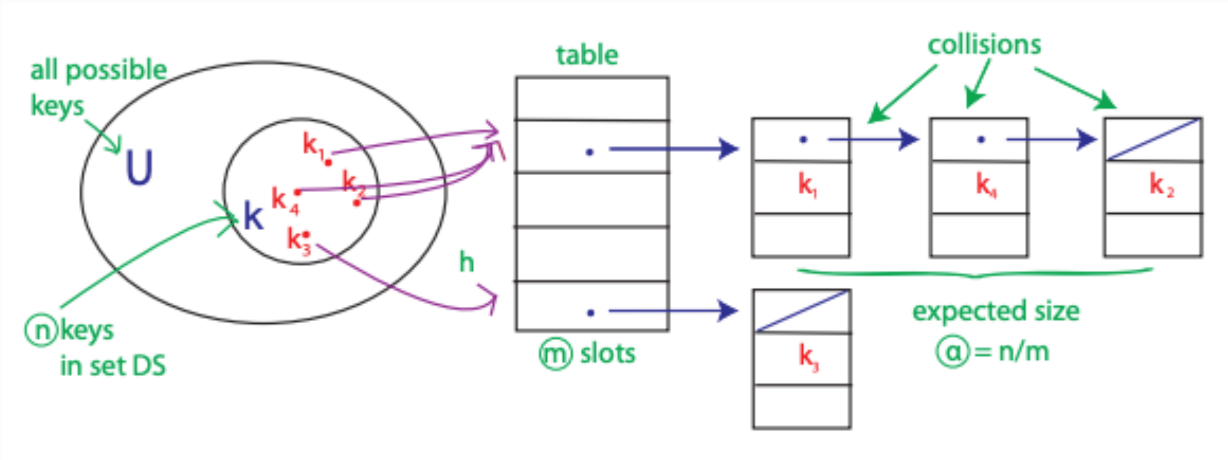
- 首先需要一个随机访问/直接访问列表（例如array）
- 有两个问题：1.键值只能为正整数，2.键值很大时需要极大的存储空间
- 解决第一个问题的方法称为prehash（预散列），即将不同类型的键值映射为正整数
- 解决第二个问题的方法为hashing（散列），通过hash function（散列函数），将预散列后的大小为n的整数空间压缩为最大值为 $m = O(n)$ 的整数空间。
- hashing的主要问题是collision（散列冲突），即两个预散列后的不同的整数经过散列后为相同的整数。
- 解决collision的一个方法为chaining，即将散列后键值相同的元素放入一个存储位置保存为链表（linked list）
    - 这个方法在实践中载荷因子（load factor）可以假定为 $\frac{m}{n}$
- 载荷因子定义为：填入表中的元素个数 / 散列表的长度
    - 另一个方法在下讲会讲到
    - 之后介绍了几种散列函数

本讲已经涉及了需要用到复杂计算的算法，教授忽略了一些计算的细节，我忽略了更多。当然，可以在PSet中遇到问题时再返回来复习。

大概念上可以看到字典问题是为了应对一个最常见的问题——查找。而能够在常数时间内完成当然是十分重要的。这当中的抽象计算模型为随机访问机（random access machine（RAM））

L9. TABLE DOUBLING, KARP-RABIN

RECALL



HOW LARGE SHOULD TABLE BE?

- want $m = \Theta(n)$ at all times
- don't know how large n will get at creation
- m too small $\implies$ slow; m too big $\implies$ wasteful

Idea:

Start small (constant) and grow (or shrink) as necessary.

Rehashing:

To grow or shrink table hash function must change (m, r)
$\Longrightarrow$ must rebuild hash table from scratch for item in old table: $\rightarrow$ for each slot, for item in slot insert into new table $\Longrightarrow \Theta(n+m)$ time $= \Theta(n)$ if $m = \Theta(n)$

How fast to grow?

When n reaches m, say

- $m+=1?$

  $\Longrightarrow$ rebuild every step

  $\Longrightarrow n$ inserts cost $\Theta(1+2+\cdots+n) = \Theta\left(n^2\right)$

- $m* = 2?m = \Theta(n)$ still $(r+=1)$

  $\Longrightarrow$ rebuild at insertion 2^i

  $\Longrightarrow n$ inserts cost $\Theta(1+2+4+8+\cdots+n)$ where n is really the next power of 2 $= \Theta(n)$

  <u>Table Doubling</u>

- a few inserts cost linear time, but $\Theta(1)$ "on average".

Delete:

- when n decreases to $m/4$, shrink to half the size $\Longrightarrow O(1)$ amortized cost

Resizable Arrays:

- same trick solves Python "list" (array)
- $\Longrightarrow$ list.append and list.pop in $O(1)$ amortized

AMORTIZED ANALYSIS

This is a common technique in data structures - like paying rent: $\$1500/\text{month} \approx \$50/$ day

- operation has amortized cost $T(n)$ if k operations cost $\leq k \cdot T(n)$
- " $T(n)$ amortized" roughly means $T(n)$ "on average", but averaged over all ops.
- e.g. inserting into a hash table takes $O(1)$ amortized time.

STRING MATCHING

Given two strings s and t, does s occur as a substring of t? (and if so, where and how many times?)

Simple Algorithm:

$\text{any}(s == t[i : i + \text{len(s)}]$ for i in range $(\text{len}(t) - \text{len}(s)))$
$-O(|s|)$ time for each substring comparison $\Longrightarrow O(|s| \cdot (|t| - |s|))$ time
$= O(|s| \cdot |t|)$ potentially quadratic

Karp-Rabin Algorithm:

- Compare $h(s) == h(t[i : i + \text{len(s)}])$
- If hash values match, likely so do strings
    - can check $s == t[i : i + \text{len}(s)]$ to be sure $\sim$ cost $O(|s|)$
    - if yes, found match $-$ done
    - if no, happened with probability $< \frac{1}{|s|}$

$\Longrightarrow$ expected cost is $O(1)$ per i

- need suitable hash function.
- expected time $= O(|s| + |t| \cdot \text{cost}(\text{h}))$
  - naively $h(x)$ costs $|x|$
  - we'll achieve $O(1)$!
  - idea: $t[i : i + \text{len}(s)] \approx t[i + 1 : i + 1 + \text{len}(s)]$.

ROLLING HASH ADT

Maintain string x subject to

- r(): reasonable hash function h(x) on string x
- r.append(c): add letter c to end of string x
- r.skip(c): remove front letter from string x, assuming it is c

Data Structure:

Treat string x as a multidigit number u in base a where a denotes the alphabet size, e.g., 256

- $r() = u \bmod p$ for (ideally random) prime $p \approx |s|$ or $|t|$ (division method)
- r stores $u \bmod p$ and $|x|$ (really $a^{|x|}$), not u
  $\Longrightarrow$ smaller and faster to work with ($u \bmod p$ fits in one machine word)
- r.append $(c) : (u \cdot a + \text{ord}(c)) \bmod p = [(u \bmod p) \cdot a + \text{ord}(c)] \bmod p$
- $r \cdot \text{skip}(c) : \left[u - \text{ord}(c) \cdot \left(a^{|u|-1} \bmod p\right)\right] \bmod p$
  $= \left[(u \bmod p) - \text{ord}(c) \cdot \left(a^{|x-1|} \bmod p\right)\right] \bmod p$

THINKING

1. 这一讲首先复习了字典，然后提出了一个尚未解决的问题：尺寸的初始化和置入过程中尺寸的维护。
    1. 解决方法也很简单，Table Doubling，因为几何级数的特性，所有doubling操作总和的复杂度是线性的。
    2. 进一步的，虽然某些insertion所引起的table doubling操作是线性的，但其余操作都是常数的，因此平摊分析（amortized analysis）中每步操作是常数的。
    - 这种尺寸维护的方法也被使用在Python List中。
2. 这一讲也介绍了另一个散列函数及其应用：使用旋转散列使字符串搜索算法为线型复杂度（而不是二次方的）

    关键在于每个单字符位移的长度为s的字符串的散列（prehashing/hashing）都可以在常数时间内完成

R9. ROLLING HASHES, AMORTIZED ANALYSIS

ROLLING HASH

AMORTIZED ANALYSIS

THINKING

这一讲详细介绍了rolling hash（旋转散列）的实现。也演示了amortized analysis（平摊分析）的实例。

平摊分析中可以看到对于BST的 `next_larger` 的平摊复杂度是常数的，因此对于 `list(l, h)` 来说，如果先找到 `l` 对应的node，再查找所有successor直到找到 `h`， 所有步骤的复杂度即为 $O(\log n + I)$

R9B: DNA SEQUENCE MATCHING

A GOOD HASH FUNCTION

PYTHON ITERATOR

- `iterator.iter()` : Returns the iterator object itself. This allows iterators to be used with the `for` and `in` statements.
- `iterator.next()` : Returns the next item. If there are no further items, raise the `StopIteration` exception.

ITERATORS VS GENERATORS

```
class Reverse:
  """Iterator for looping over a sequence backwards."""
  def __init__(self, data):
    self.data = data
    self.index = len(data)
  def __iter__(self):
    return self
  def next(self):
    if self.index == 0:
      raise StopIteration
    self.index = self.index - 1
    return self.data[self.index]
```

```
def reverse(data):  ## generator
  for index in range(len(data)-1, -1, -1):
    yield data[index]  ## create an iterator
```

PSET4

GENERATOR AND DICTIONARY

如果查找两个文件中的相同元素：

1. 一种方法是先遍历第一个文件，过程中遍历第二个文件，对比每个元素，复杂度为 $O(n^2)$
2. 更好的方法是，遍历第一个文件，建立key为元素、value为所有位置列表的字典；然后遍历第二个文件，过程中查找列表是否在字典内，如果在输出位置对。复杂度为 $O(n)$
   - **因此generator通常不重复使用，因为多次遍历过于耗时，如果需要重复查找，则建立字典**

L10. OPEN ADDRESSING, CRYPTOGRAPHIC HASHING

Readings

CLRS Chapter 11.4 (and 11.3.3 and 11.5 if interested)

OPEN ADDRESSING

- one item per slot $\Longrightarrow m \geq n$
- hash function specifies order of slots to probe (try) for a key (for insert/search/delete), not just one slot

Insert(k,v) :

Keep probing until an empty slot is found. Insert item into that slot.

```
for i in xrange(m):
  if T[h(k, i)] is None:   ## empty slot
    T[h(k, i)] = (k, v)   ## store item
    return
raise ‘full’
```

Search(k):

As long as the slots you encounter by probing are occupied by keys $\neq$ k, keep probing until you either encounter k or find an empty slot—return success or failure respectively.

```
for i in xrange(m):
  if T[h(k, i)] is None:   ## empty slot?
    return None   ## end of “chain”
  elif T[h(k, i)][0] == k:   ## matching key
    return T[h(k, i)]   ## return item
return None   ## exhausted table
```

Deleting Items?

- can't just find item and remove it from its slot (i.e. set T(h(k, i)) = None)
- example: delete(586) ⇒ search(496) fails
- replace item with `special flag: "DeleteMe"`, which Insert treats as None but Search doesn't

PROBING STRATEGIES

Linear Probing

$h(k,i) = (h'(k) + i) \bmod m$ where $h'(k)$ is ordinary hash function

- like street parking
- problem? clustering—cluster: consecutive group of occupied slots as clusters become longer, it gets more likely to grow further
- can be shown that for 0.01 < α < 0.99 say, clusters of size $\Theta(\log n)$

Double Hashing

$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m$ where $h_1(k)$ and $h_2(k)$ are two ordinary hash functions.

UNIFORM HASHING ASSUMPTION

Each key is equally likely to have any one of the m! permutations as its probe sequence

- not really true
- but double hashing can come close

Analysis

Suppose we have used open addressing to insert n items into table of size m. Under the uniform hashing assumption the next operation has expected cost of $\le \frac{1}{1-\alpha}$, where $\alpha = n/m(< 1)$.
Example: $\alpha = 90\% \implies 10$ expected probes

OPEN ADDRESSING VS. CHAINING

Open Addressing: better cache performance (better memory usage, no pointers needed)

Chaining: less sensitive to hash functions (OA requires extra care to avoid clustering) and the load factor α (OA degrades past 70% or so and in any event cannot support values larger than 1)

CRYPTOGRAPHIC HASHING

THINKING

这一讲主要介绍了处理散列函数冲突问题的第二个方法：Open Addressing（开放地址法），这样只需要1个array就可以，不再需要linked list。

- 有一个技术的细节：删除的时候需要在当前位置设置一个特殊值，insert处理这个特殊值等同于None，但search会将其当作一般值。
- 由于没有指针的需求，对于存储的使用较优。

- 需要特殊的散列函数来处理probing的问题，如双散列（double hashing）

散列也是计算机密码的重要基础，通过散列，管理员也无法获得真实密码。

- 单向性（One-Way，OW）
- 抗冲突（Collision-Resistance，CR）
- 目标抗冲突（Target Collision-Resistance，TCR）？

R10. QUIZ 1 REVIEW

DECISION TREE(LOWER BOUND)

2 possible comparison output, e.g. True or False:

$\Omega(\log n)$

3 possible comparison output, e.g. >, < or =:

$\Omega(\log_3 n)$

BUNKER HILL

Using Rolling Hash to scan a 2-D array to find Equivalence

R11. PRINCIPLES OF ALGORITHM DESIGN

k^{th} MINIMUM IN MIN-HEAP

1. $O(N \log N)$ Algorithm -- simplest
    - Heap_Sort the array
2. $O(k \log N)$ Algorithm -- parctical complexity
    - Extract_Min for k times
3. $O(N \log k)$ Algorithm -- get some insights to the problem
    - Iterate throught the array and maintain a k-elements Max-Heap (need to heapify every time a replacement is found)
4. $O(k^2)$ Algorithm (for small k) -- get more insights
    - reduce the height of the heap to k and use the 2nd algorithm
5. $O(k \log k)$ Algorithm
    - maintain a Horizon（min-heap）, and each time extract a minimum from it, add the minimum's two sub-nodes to it (**so that it always keep the roots of sub-trees and thus the minimum value**)

PRINCIPLES OF ALGORITHM DESIGN

See the Recitation Note

1. Experiment with examples.
2. Simplify the problem.
    1. Sometimes, when a problem is difficult to solve, it can be worth it to solve a related, simpler problem instead.

    2. Once you develop ideas for the simpler case, you can often apply them to handle the more complex case.
3. Look for similar problems.
4. Delegate the work.
    1. Recursion
5. Design according to the runtime.

THINKING

很多算法都和对于数据结构的理解有关，这点还是很重要的。例如上边这个例子，如果能够理解到最小堆（min-heap）的所有子堆都是最小堆的话，只需要维护一个去除掉前 i 个最小值的子堆的根（root）的最小堆即可。