

L05.01

Hashing I

50.004 Introduction to Algorithms
Ioannis Panageas (ioannis@sutd.edu.sg)
ISTD, SUTD
CLRS Ch 11.1-11.3
Slides by A.Binder and based on Dr. Simon LUI

Today topics

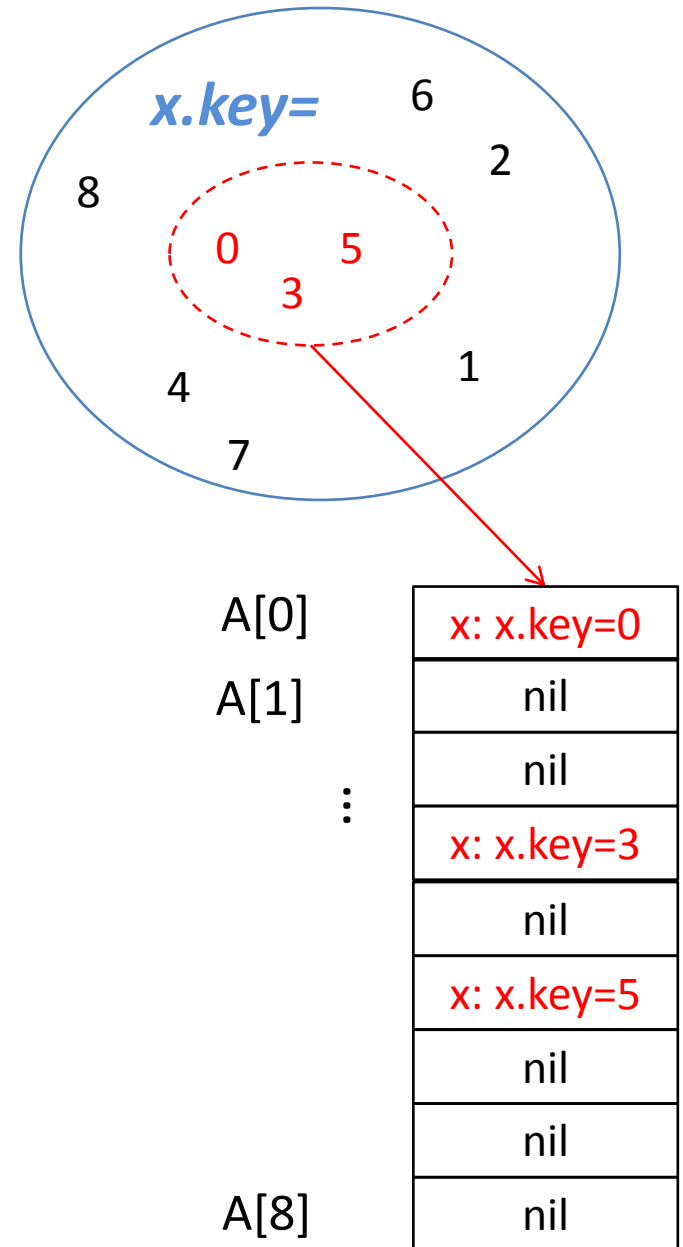
- Hash table, hash function
 - Insert, search, delete in practice very efficient
 - Insert, delete: $O(1)$
 - search: $O(1+\alpha)$ α = load factor of the hash table
 - Hash table collisions
- Hash functions for non-numbers
- Good versus bad hash functions
- Runtime Analysis
- Table Doubling tricks for making search $O(1)$ “on average” (not worst case complexity!!)

Hash tables

- A data structure that supports ‘dictionary operations’ search(key), insert(key), delete(key) **very fast** most of the time
 - **Heaps**: insert $O(\log n)$, search $O(n)$
 - **Binary search trees**: insert, search $\rightarrow O(\text{height})$, worst case $O(n)$
 - **AVL trees** insert, search $\rightarrow O(\log n)$,
 - **Hash tables**:
 - Insert $O(1)$ worst case,
 - Search: $O(n)$ worst case performance BUT
 - $O(1 + \alpha)$ average case performance
 - Delete: $O(1)$ worst case performance*
- * If you have the pointer to the object

an array

- Suppose keys $\{0, \dots, l-1\}$, object **x** has key **x.key**
- Use an **array A** of length l
 - initialize with NULL pointers (python: list with empty entries)
- Insert(x):
 $A[x.key] = x$
- Delete(x)
 $A[x.key] = \text{NULL}$
- Search(x):
 Return True or False?
 ($A[x.key] \neq \text{NULL}$)
- **All $O(1)$ operations**



Array(ii)

- Use an **array** **A** of length (**I**), initialize with NULL pointers (or python: list with empty entries)
- Insert(x):
 $A[x.key]=x$
- When this is not practicable?

Array(ii)

- When this is not practicable?
 - Set of possible objects/keys very large
 - Array of length I does not fit into memory!!
 - Don't know the values of keys in advance
 - Keys are not integers:
 - Filenames, student names, instances of an object class

Hash table + simple array

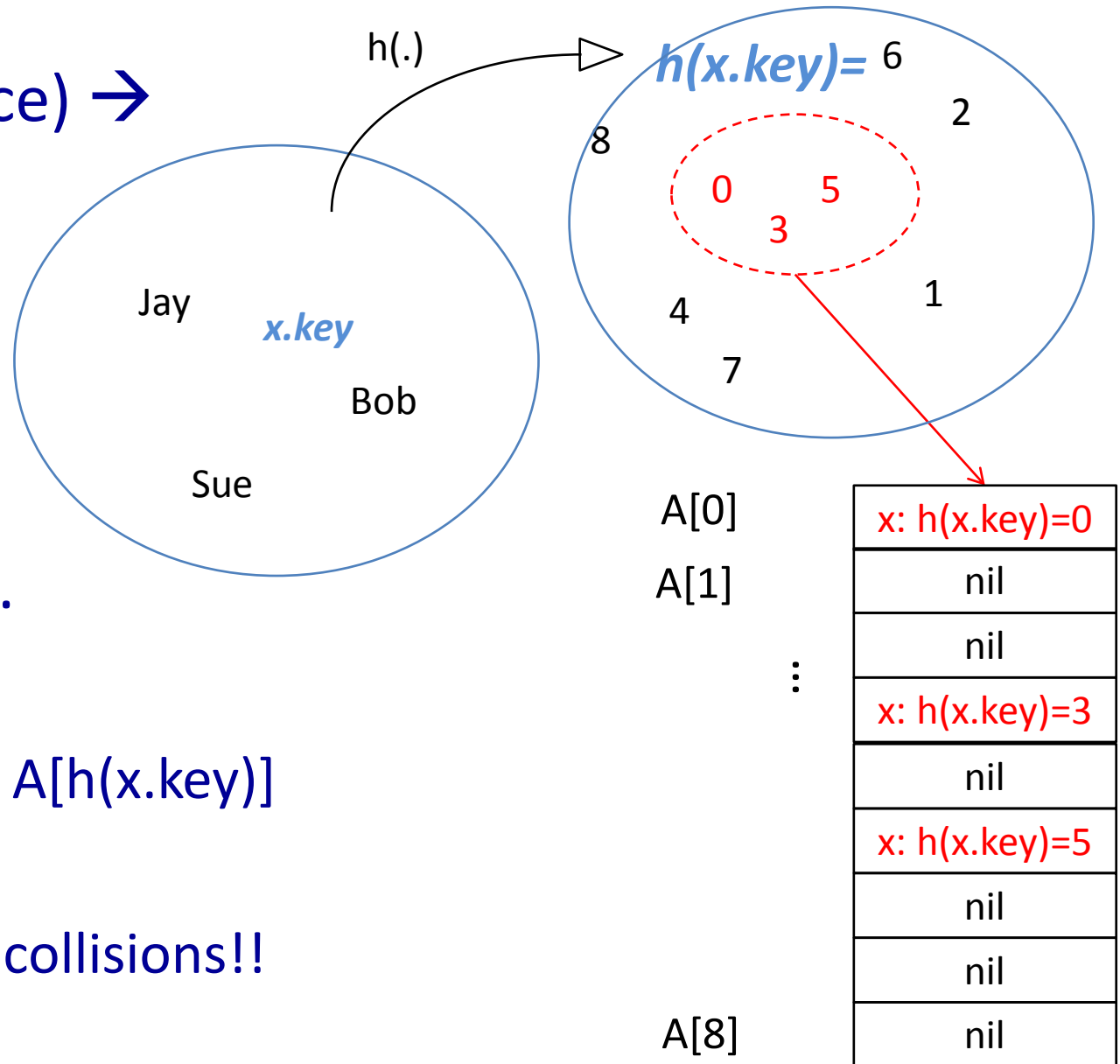
- When this is not practicable?
 - Set of possible objects/keys very large
 - Keys are not integers:
 - Filenames, student names,
- Solution: Hash function:
 - Let L be the space of all possible keys (e.g. $L = \{\text{all possible filenames}\}$)
 - Hash function ('h'): a function that maps the space of all possible keys K onto $\{0, \dots, l-1\}$
 - $h: K \rightarrow \{0, \dots, l-1\}$,
- First idea (will refine it): Hash table is will be an array of size l ,

Hash function + simple Array

- $h: K \text{ (key space)} \rightarrow \{0, \dots, l-1\}$,

$h: \text{Jay} \rightarrow 0,$
 $\text{Bob} \rightarrow 1,$
 $\text{Sue} \rightarrow 2, \dots$

- $\text{Insert}(x)$:
 - Insert x into $A[h(x.\text{key})]$
 - All happy?? collisions!!



Collisions

- Set of keys (e.g. $K = \{\text{all possible filenames}\}$) is very large,
 - hash table is an array of **length l** , l typically much smaller than set of keys
 - If $|K| > l$, then there must be two keys in L such that $h(\text{key1}) = h(\text{key2})$
 - Collision: two keys are hashed onto the same value: $h(\text{key1}) = h(\text{key2})$

$$K = \{0, \dots, 100\}$$

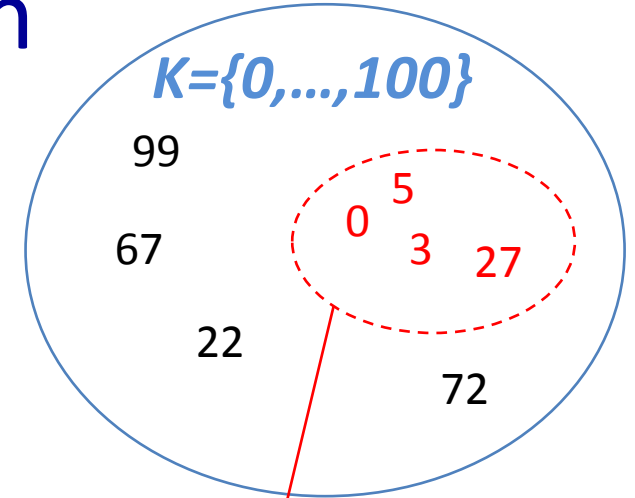
$$h(k) = k \bmod l = k \bmod 9$$

$$h(1) = h(37) = 1$$

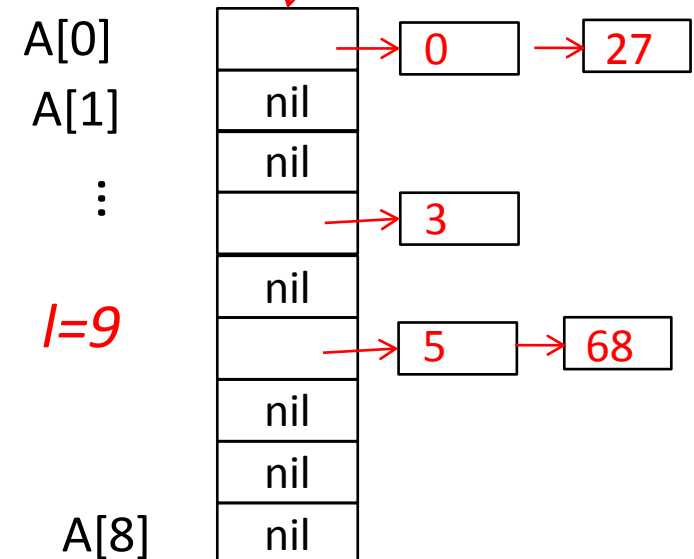
| | |
|------|-----------------------------|
| A[0] | nil |
| A[1] | x: x.key = 1, x: x.key = 37 |
| ⋮ | nil |
| | nil |
| | nil |
| | nil |
| A[6] | nil |

Hash function + Hash table with chaining -- formal definition

- Hash function:
 - K = space of all possible keys e.g. $K=\{0,\dots,100\}$
 - Hash function ('h') maps L onto $\{0,\dots,l-1\}$, where l is the size of the has table
 - $h: K \rightarrow \{0,\dots,l-1\}$,
- Hash table is an array of size l ,
 - each element in the table is a [doubly]-linked list* ("chaining")
 - Allows to insert multiple elements for the same hash value

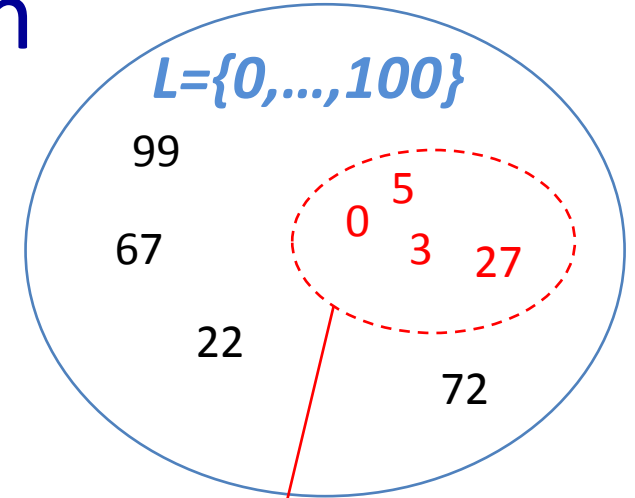


$$h(k) = k \bmod l = k \bmod 9$$

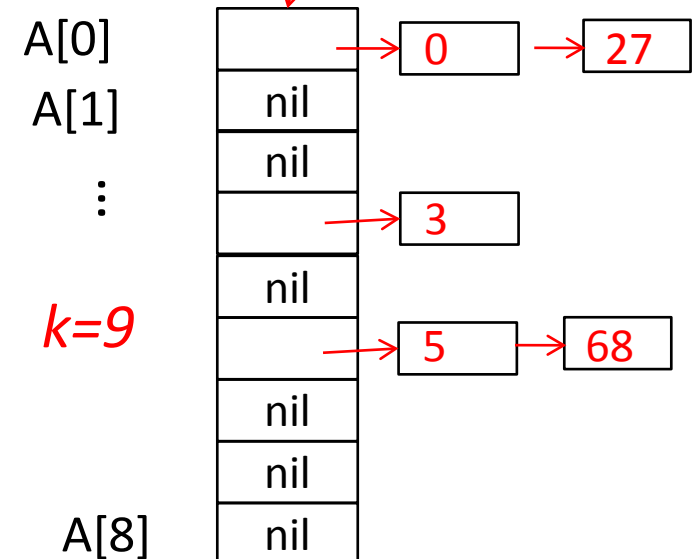


Hash function + Hash table with chaining -- formal definition

- Idea: map a very large set of keys to a small number of hash-values
- Efficient insert, delete in worst case $O(1)$
- Efficient search:
 - Worst case $O(n)$,
 - average case $O(1+\alpha)$
 - average case $O(1)^*$ under some assumptions



$$h(k) = k \bmod l = k \bmod 9$$



Hash functions

- Chained-hash-Insert(A, x) $O(1)$ – compute $h(x.key)$ + 1 insert
– Insert element x at the head of linked list $A[h(x.key)]$
- Chained-hash-search(A, k) $O(???)$
– Search for an element with key k in list $A[h(k)]$

Worst case performance is what and why?

Hash functions

- Chained-hash-Insert(A, x) $O(1)$ – compute $h(x.key) + 1$ insert
 - Insert element x at the head of linked list $A[h(x.key)]$
- Chained-hash-search(A, k) $O(???)$ – will analyze this later
 - Search for an element with key r in list $A[h(r)]$
- Chained-hash-Delete(A, x): $O(1)$ [$O(???)$ if using the key]
 - We assume here that we have a pointer to x , that the memory address of element x
 - Go to x in $A[h(x.key)]$ using the address
 - Link predecessor and successor of x in this doubly linked-list (use link to previous and next element of x in the list!)
 - Delete x from the list $A[h(x.key)]$ using the pointer

Further questions ...

- How to hash filenames and non-numerical entries?
- What is a good hash function?
- Runtimes ? Load factors?

Hashing for non-numerical input

- Latin?

$$\text{'pt'} \rightarrow (112, 116) \xrightarrow{\text{radix } 128} 112 \cdot 128^1 + 116 \cdot 128^0$$

- Written Chinese? Tamil? Jawi? Traditional Chinese? Old Javanese ?

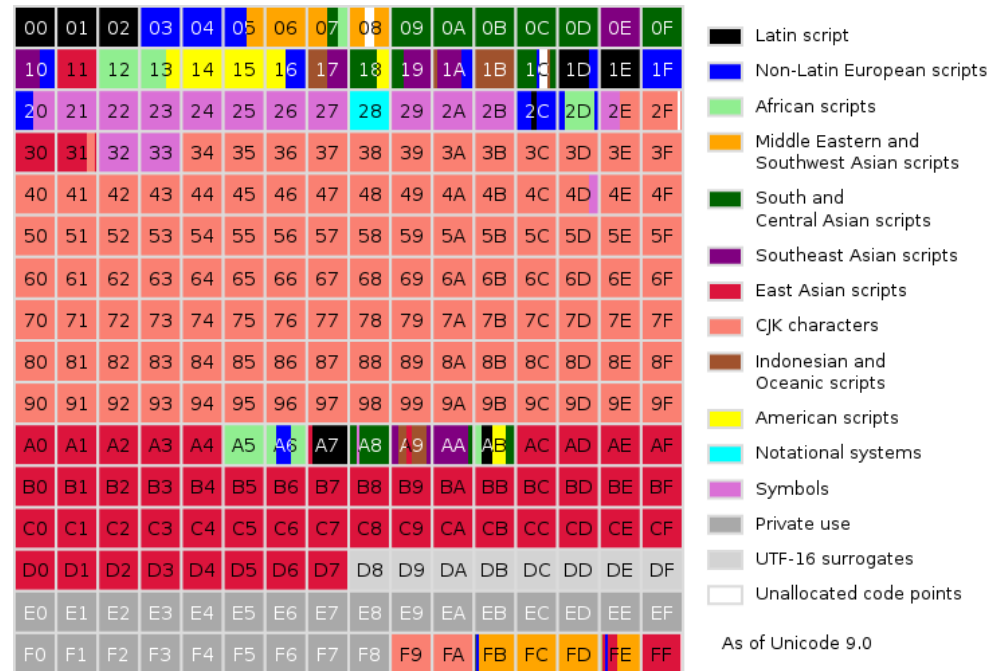
Hashing for non-numerical input

Written Chinese? Tamil? Jawi?

Old Javanese ? Traditional Chinese? Sanskrit?

Unicode <https://en.wikipedia.org/wiki/Unicode>

- Most languages encoded in 6 planes, each with 65535 entries (unicode supports 17 planes)
- Shown: plane 0

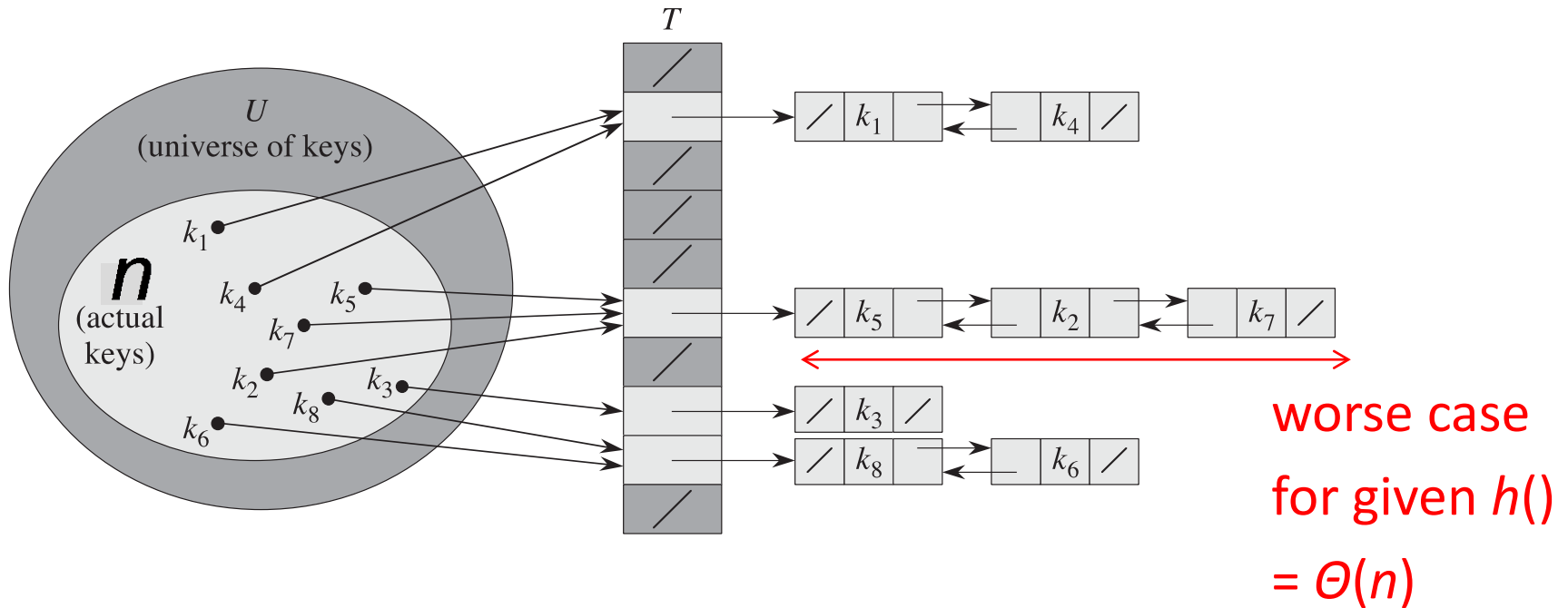




What is a good hash function?

- Scenario: N keys are coming in, unknown order
- Worst case: ??

Resolving collisions: chaining



insert(k) $\rightarrow O(1)$

delete(k) $\rightarrow O(1)$

search(k) $\rightarrow \Theta(n)$

What is a good hash function?

- Scenario: N keys are coming in, unknown order
- Worst case: hashes all keys into the same slot.
 - search: Have to go through one linked list with n elements
- Best case:
 - How n keys should be distributed onto l lists for fast search??

What is a good hash function?

- Scenario: N keys are coming in, unknown order
- Worst case: hashes all keys into the same slot.
 - search: Have to go through one linked list with n elements
- Best case: they are distributed equally likely into all slots
 - All slots are filled with approximately the same number of keys search is faster in worst case ... n/l elements
- Bad case: a few slots have high load, most of table is empty “clustering of keys”

What is a good hash function?

- Scenario: N keys are coming in, unknown order
- Worst case: hashes all keys into the same slot.
 - search: Have to go through one linked list with n elements
- Best case: they are distributed equally likely into all slots
 - All slots are filled with approximately the same number of keys
search is faster in worst case ... n/l elements
- **Simple uniform hashing assumption:** any given element is equally likely to hash into any of the l slots, independently of where any other element has hashed to.
 - Getting close to it – design of a hash function

Why it is hard to achieve the simple uniform hashing assumption?

- Distribution of incoming keys unknown
- Can have correlations, not independent
 - E.g. programmers use variable names that follow rules of their spoken languages. nobody writes code with random variable names.
 - Infections in english end all on “itis”
 - Other examples?
- Example of a hash function that satisfies U.H.A.
- K = random real numbers $0 \leq r < 1$, uniformly distributed, have l slots.
- $h(k) = \text{floor} (k l)$

- Think about: being close to simple uniform hashing assumption depends on what factors ?

Good or bad hash function?

- $H(k) = k \bmod 2^p$
 - Returns p lowest bits, when k is written in binary representation

Case 1: K = uniformly distributed integers $\{0, 2^{\{2p\}}-1\}$

Case 2: K = odd numbers, uniformly distributed

Case 3: K = { total size in byte of an array of floats, array lengths are uniformly distributed, each single entry has 4 byte length }



12



20

Why it is hard to achieve the simple uniform hashing assumption?

- Knowledge of distribution of keys helps to design better hash functions.
- array Bytelength example: p bits starting at bit #2 is better
 - lowest p bits have bad reputation in C.S. ;-)

p=10 lowest bits

- For array lengths in big data :D
- Now some hash functions
- Mostly heuristics ... hacks that work well in practice

Hash function example: The division method

$$h(k) = k \bmod l \quad \textbf{Example: } l = 19 \rightarrow h(93) = 17$$

Choice of l : $l = 2^p$ (powers of 2) is risky.

l too close to a power of 2 can have surprising effects.

Example: encode strings in base 2^p

$$\text{pt} \rightarrow (112, 116) \rightarrow 112 \cdot 2^1 + 116 \cdot 2^0$$

If $l = 2^p - 1$, then all permutations of a string result in the same hash ! (proof on request ...)

A good choice of l : A prime not close to 2^p for any integer p
Results in funny hash table sizes

Hash function example: The multiplication method

$$h(k) = \lfloor l ((kA) \bmod 1) \rfloor = \lfloor l (kA - \lfloor kA \rfloor) \rfloor$$

► Method:

- Choose constant A in the range $0 < A < 1$.
 - Multiply key k by A .
 - Extract the fractional part of kA .
 - Multiply the fractional part by m .
 - Take the floor of the result.
- In short, the hash function is $h(k) = \lfloor m (kA \bmod 1) \rfloor$, where $kA \bmod 1 = kA - \lfloor kA \rfloor = \text{fractional part of } kA$.

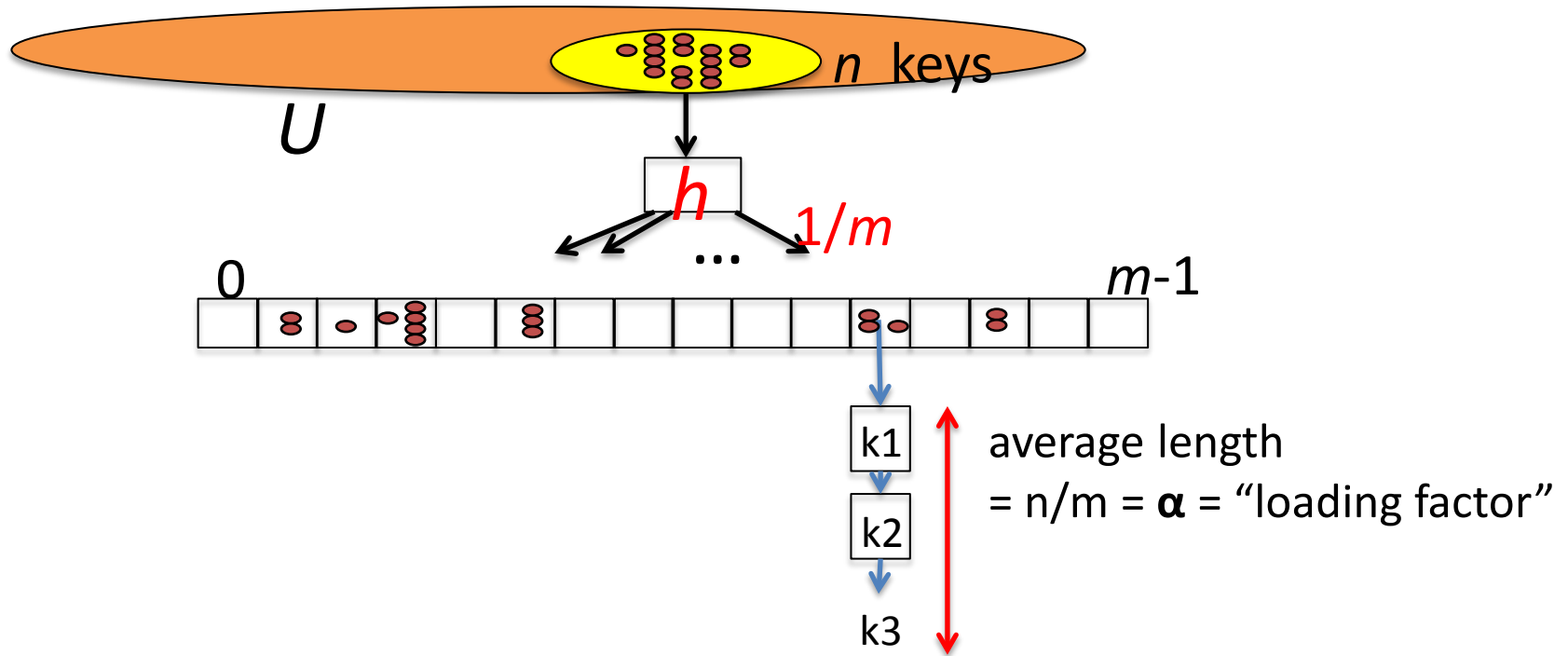
Advantage: value of m is not critical (can be 2^p)

Disadvantage: slower than division method

Average Run time analysis

- Chained-hash-search(A, k) $O(???)$
 - Search for an element with key k in list $A[h(k)]$

Under the simple uniform hashing assumption



Average Run time analysis

- Let $\alpha = N/l$ be the load factor of a hash table
- Under the simple uniform hashing assumption, the average runtime of chained-hash-search is $O(1+\alpha)$
 - This is an average runtime result,
 - Depends on the design of the hash!!

Average Run time analysis

- Case 1: Search for a key that is not in the table
- Case 2: Search for a key that is in the table
- Case 1: search in $A[h(k)]$
 - $A[h(k)]$ has on average $\alpha = N/I$ elements
 - Must go through these elements until element x is found
 - $O(1)$ from computing $h(k)$ and looking up in $A[.]$
 - So Case 1: $O(1 + \alpha)$,

Out of curriculum part

- Case 1: Search for a key that is not in the table
 - Case 2: Search for a key that is in the table
-
- Case 2: $O(1 + \alpha)$,
 - Read CLRS if you want to know more ...

Two keys collide if: $1\{h(k_i) == h(k_l)\}$
elements inserted before are those that were inserted after
element i , that's why sum $i+1 \dots n$

$$s = \frac{1}{n} \sum_{i=1}^n E \left[1 + \sum_{l=1+i}^n 1\{h(k_i) == h(k_l)\} \right]$$

Code here

- Under the **simple uniform hashing** assumption, the average runtime of chained-hash-search is $O(1+\alpha)$

- Under the **simple uniform hashing** assumption, the average runtime of chained-hash-search is $O(1+\alpha)$
that is $O(1)$ if α is upper bounded (independent of N).
- “ α being upper bounded” **Requires what?** (as one adds more and more entries into the hash table)

why resize a hash table?

- Under the **simple uniform hashing** assumption, the average runtime of chained-hash-search is $O(1+\alpha)$
that is $O(1)$ if α is upper bounded (independent of N).
- “ α being upper bounded” **Requires what?** (as one adds more and more entries into the hash table)
- Hash table needs to be resized “from time to time” when load factor α becomes too large
- We will see: for search complexity $O(1)$ we need to learn about two tricks
 - 1. the right strategy of growing a table
 - 2. only as an average over many insert steps
- $O(1)$ can only be attained on average over a large number of inserts (“amortized analysis”)
- **GrowTable(A, l')**
 - Make Table A' of size l'
 - Build new hash function h'
 - Rehash:
For item in $A.items()$:
 $A'.insert(item) \% \text{using } h' \text{ as new hash inside}$
why we do not reuse/extend the existing hash?
- **Complexity ?**

In What way resize a hash table?

- Complexity ? $O(n+l+l')$
 - l for visiting each table entry in old table
 - N for processing all objects stored in there
 - l' for making of new table
- How much make the table bigger?
- $l'=l+1$?

In What way resize a table ?

- Resize Complexity $O(n+l+l')$
- How much make the table bigger?
- resize by 1
 - $l'=l+1$ whenever $n=m$, start at $m=1$
 - Sum over resizes in l : $1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2} = O(n^2)$ Quadratic, slow
- resize: adding a constant to the size
 - $l'=l+c$ whenever $N/l = t$???

In What way resize a table ?

- Resize Complexity $O(n+l+l')$
 - How much make the table bigger?
 - resize by 1
 - $l'=l+1$ whenever $n=m$, start at $m=1$
 - Sum over resizes in l : $1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2} = O(n^2)$ Quadratic, slow
 - resize: adding a constant to the size
 - $l'=l+c$ whenever $N/l = t$ load factor hits a threshold t
→ Same problem as above!!!
 - suppose $N/l = t$, next resize time: N' such that $N'/(l+c)=t$ (= next resize)
 - $N'-N = (l+c)t - lt = ct \rightarrow$ resize after every ct steps, $N' = \{1, 2, 3, 4, 5\} ct$
 - resize complexity is $\theta(n+l+l') = \theta(N') = \text{some constant} \cdot \{1, 2, 3, 4, 5\} \cdot ct$
 - Last resize time: $\left\lfloor \frac{N_{final}}{ct} \right\rfloor$
- Overall resize complexity is:
- $$s = O(ct + 2ct + 3ct + \dots + \left\lfloor \frac{n}{ct} \right\rfloor ct) = O\left(\frac{\left\lfloor \frac{n}{ct} \right\rfloor \left(\left\lfloor \frac{n}{ct} \right\rfloor + 1\right) ct}{2}\right) = O(n^2)$$

In What way resize a table ?

- Complexity ? $O(n+l+l')$
- How much make the table bigger?
- resize by doubling
 - $l'=2l$ whenever $\alpha = N/l = t$ load factor hits a threshold t

$$\frac{N}{l} = t, \text{ next resize time: } \frac{N'}{l'} = \frac{N'}{2l} = t$$

$$\rightarrow N' - N = lt = N \rightarrow N' = 2N,$$

With some initial $N_0 = l_0 \cdot t$, (l_0 is initial table size),

In What way resize a table ?

- resize by doubling
 - $l' = 2l$ whenever $N/l = t$
 - $N/l = t$, next resize time: $N' = 2N$, starts at some initial sample size N_0
 - Resize when sample size reaches $N_0, 2N_0, 4N_0, \dots, 2^t N_0$
 - resize complexity is $\theta(n + l + l') = \theta(n)$
with $n = N_0, 2N_0, 4N_0, \dots, 2^t N_0$
 - Let be last resize time when $N_{final} = 2^{t_f}$

Overall resize Complexity is

$$s = O(N_0 + 2N_0 + 4N_0 + \dots + 2^{t_f} N_0)$$

In What way resize a table ?

- resize by doubling
 - $l' = 2l$ whenever $N/l = t$
 - Resize when sample size reaches $N_0, 2N_0, 4N_0, \dots, 2^t N_0$
 - resize complexity is $\theta(n+l+l') = \theta(n)$
with $n = N_0, 2N_0, 4N_0, \dots, 2^t N_0$
 - Let be last resize time when $N_{final} = 2^{t_f} N_0$
- overall resize Complexity is
- $$s = O(N_0 + 2N_0 + 4N_0 + \dots + 2^{t_f} N_0) = O(N_0 2^{t_f+1})$$
- !!!!!!!!!!!!
- $$= O(2N_0 2^{t_f}) = O(2N_{final})$$

In What way resize a table ?

- resize by doubling
 - $l' = 2l$ whenever $N/l = t$
 - resize complexity is $\theta(n+l+l') = \theta(n) = \theta(l)$

Overall resize Complexity is

$$S = O(N_{final}) \text{ NOT } O(N^2)$$

- That's why table doubling!

Table doubling

- Do we get the nice $O(N_{final})$ when not doubling but

$$l' = l \cdot r, r > 1 ???$$

Table doubling

- Do we get the nice $O(N_{final})$ when not doubling but

$$l' = l \cdot r, r > 1 ??$$

Yes! proof relies on the geometric series

$$r^0 + r^1 + r^2 + \dots = \sum_{k=0}^t r^k = \frac{r^{t+1} - 1}{r - 1}$$

Table doubling in general

- table doubling = resize a hash table by doubling (or exponentially by multiplying with a factor $r > 1$)
- $l' = l \cdot r, r > 1$

Overall resize Complexity for table doubling is

$$O(N_{final})$$

Now we can show under what conditions Hash-search is $O(1)$

amortized analysis

- ***amortized analysis*** = average the time required to perform a sequence of operations over the number of operations
 - $N_{final} = 2^{tf} N_0$ hash table inserts,
 - all necessary table doublings as we insert in total N_{final} many keys
- Amortized costs for insert + table growing is $O(???)$
 - Hash table inserts: $N_{final} O(1)$
 - Table doubling costs: $O(N_{final})$
 - Total costs: $O(N_{final}) + N_{final} O(1) = O(N_{final})$
- Average cost for search (IF simple uniform hashing assumption HOLDS) is:
 - Hash table search: $O(1+\alpha)$
 - α is bounded independent of N_{final} by table doubling strategy, so
Hash table search: $O(1)$!!!

amortized analysis

- ***amortized analysis*** = average the time required to perform a sequence of operations over the number of operations
 - $N_{final} = 2^{tf} N_0$ hash table inserts,
 - all necessary table doublings as we insert in total N_{final} many keys
- Amortized costs for insert + table growing is $O(???)$
 - Hash table inserts: $N_{final} O(1)$
 - Table doubling costs: $O(N_{final})$
 - Total costs: $O(N_{final}) + N_{final} O(1) = O(N_{final})$
- Average cost for search (IF simple uniform hashing assumption HOLDS) is:
 - Hash table search: $O(1+\alpha)$
 - α is bounded independent of N_{final} by table doubling strategy, so
Hash table search: $O(1)$!!!

amortized analysis

- ***amortized analysis*** = average the time required to perform a sequence of operations over the number of operations
 - $N_{final} = 2^{tf} N_0$ hash table inserts,
 - all necessary table doublings as we insert in total N_{final} many keys
- Amortized costs for insert + table growing is $O(???)$
 - Hash table inserts: $N_{final} O(1)$
 - Table doubling costs: $O(N_{final})$
 - Total costs: $O(N_{final}) + N_{final} O(1) = O(N_{final})$
- Average cost for search (IF simple uniform hashing assumption HOLDS) is:
 - Hash table search: $O(1+\alpha)$
 - α is bounded independent of N_{final} by table doubling strategy, so
Hash table search: $O(1)$!!!

amortized analysis

- ***amortized analysis*** = average the time required to perform a sequence of operations over the number of operations
 - $N_{final} = 2^{tf} N_0$ hash table inserts,
 - all necessary table doublings as we insert in total N_{final} many keys
- Amortized costs for insert + table growing is $O(???)$
 - Hash table inserts: $N_{final} O(1)$
 - Table doubling costs: $O(N_{final})$
 - Total costs: $O(N_{final}) + N_{final} O(1) = O(N_{final})$
- Average cost for search (IF simple uniform hashing assumption HOLDS) is:
 - Hash table search: $O(1+\alpha)$
 - α is bounded independent of N_{final} by table doubling strategy, so
Hash table search: $O(1)$!!!

amortized analysis

- ***amortized analysis*** = average the time required to perform a sequence of operations over the number of operations
 - $N_{final} = 2^{t_f} N_0$ hash table inserts,
 - all necessary table doublings as we insert in total N_{final} many keys
- Amortized costs for insert + table growing is $O(???)$
 - Hash table inserts: $N_{final} O(1)$
 - Table doubling costs: $O(N_{final})$
 - Total costs: $O(N_{final}) + N_{final} O(1) = O(N_{final})$
 - **Amortized (=average) cost over N_{final} insertions:**
$$\frac{O(N_{final})}{N_{final}} = O(1)$$
- Average cost for search (IF simple uniform hashing assumption HOLDS) is:
 - Hash table search: $O(1+\alpha)$
 - α is bounded independent of N_{final} by table doubling strategy, so
Hash table search: $O(1)$!!!

Final Result:

- Under the simple uniform hashing assumption:

The average search time of a chained hash table with table doubling strategy is $O(1)$

Why ? (because we do table doubling whenever the load factor α goes above a fixed threshold t (=independent of n), therefore $O(1 + \alpha) \leq O(1 + t)$ which is $O(1)$ when considered as a function of the number of keys n)

- The amortized cost for the necessary insertions and table doublings (for keeping α bounded by a constant) are also $O(1)$
- This is a double average
 - $O(1 + \alpha)$ contains averaging of distribution of keys into hash slots (simple uniform hashing assumption)
 - Amortization Analysis of Table doubling contains averaging over a large number of sequential insertions

Conclusions...

- Chained hash:
 - an array s.t. each entry is a linked list
 - a hash function that maps keys onto array indices
 - Map n keys from a very large key space onto hash table of size m
- may have hash collisions
 - Hash operations: insert, delete as $O(1)$ worst case
 - Search $O(n)$ worst, $O(1+\alpha)$ average case (simple uniform hashing assumption)
 - When combined with table doubling: $O(1)$ average case\
 - Amortized costs for table doubling+ insertions: also $O(1)$
- Good hash functions: close to **simple uniform hashing assumption** each key has equal chance to end up in any of the bins ... ensures on average equal load across the whole table