Julius-Maximilians-UNIVERSITÄT WÜRZBURG

CAIDAS

WüNLP

ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

Hashing

Prof. Dr. Goran Glavaš

13.11.2023

Content

- ADS: Associative Array
- Hash Table
- Hash Functions

Dynamic sets

- We go back to our **dynamic sets**
  - We need to store a set of data points (simple data types or complex ones)

- We've already seen several ADS that can store **dynamic sets**
  - **List**
  - **Stack**
  - **Queue**
  - **Priority Queue (Heap)**

- From these, only **list** has no constraints on insertion into and removal of elements from the **dynamic set**

Dynamic sets

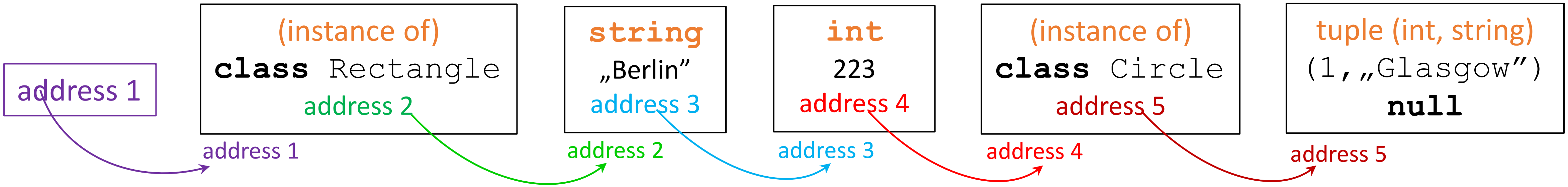
- In many applications/algorothms we need only **three basic operations** for manipulating the content of **dynamic sets**

1. **INSERT – add new element to the dynamic set**
   - In general, in no particular order
   - Constraints on positioning of elements: stacks, queues, heaps, search trees...

2. **SEARCH –** answer the question „is element X in the set”?
3. **DELETE –** remove an element from the set
   - In general, any element from the set can be removed
   - Constraints on (order of) element removal – stacks, queues, heaps, search trees...

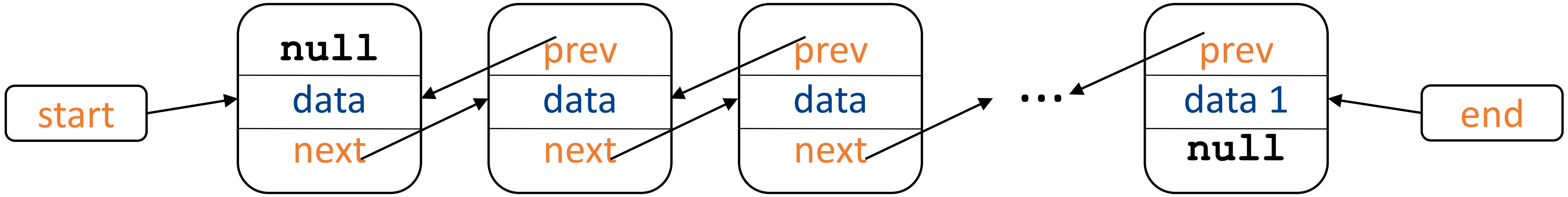
Recap – **Lists**: Arrays vs. Linked Lists

- ADT: **List** – a **linear** sequence of elements
  - When we design algorithms, we typically think in terms of ADTs
- **Linked List**
  - Consists of **nodes**: nodes contain both the data (values) and a **pointer** to the next node in the list
  - Nodes can contain values of different types
  - Dynamic data structure: „resizable” at run time
    - Non-contiguous memory allocation possible, space for **new nodes** can be allocated dynamically (on „per-need” basis)



Dynamic Set with List

- Dynamic set operations: INSERT, DELETE, SEARCH
- List implemented as a (bidirectional) **linked list**:



- Runtimes (in Big-O notation):
  - INSERT (assuming no constraints on where the element is to be inserted)?
  - SEARCH?
  - DELETE (assuming no constraints on where the element is to be inserted)?

Abstract Data Types

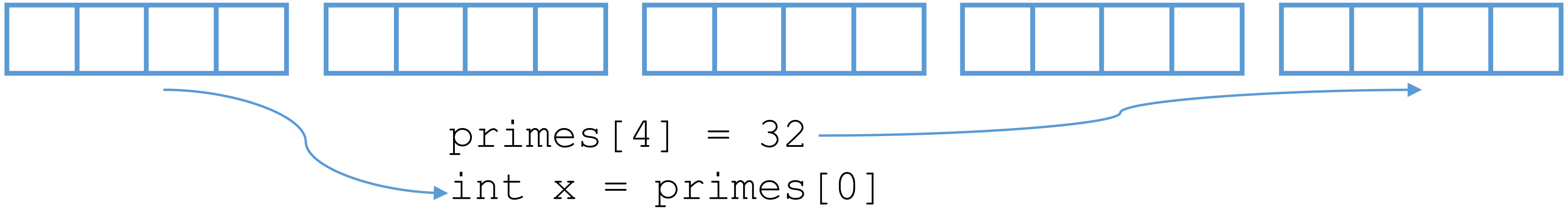
Abstract Data Type	Other Common Names	Commonly implemented as
List	**Sequence**	**Array, Linked List**
Queue		**Array, Linked List**
Double-ended Queue	Dequeue, Deque	Array, Doubly-linked List
Stack		**Array, Linked List**
Associative Array	**Dictionary, Hash Map, Map**	**Hash Table**
Set		Red-black Tree or Hash Table
Priority Queue	**Heap**	**Heap**

Associative Array

Associative array

An ADS for representing dynamic sets containing **(key, value)** pairs such that each key is unique in the array. Associative array, also known as Dictionary or Map, supports **direct addressing**: computation of memory location of **value** directly from the **key**.

- Isn't a regular **array** associative by default?
  - *key* = index of the array at which we find the element
  - Given the *key* (i.e., index), we can compute the memory address of the value



Associative Array

Associative array

An ADS for representing dynamic sets containing **(key, value)** pairs such that each key is unique in the array. Associative array, also known as Dictionary or Map, supports **direct addressing**: computation of memory location of **value** directly from the **key**.

- Isn't a regular **array** (aka **direct-access table**) associative by default?
  - *key* = index of the array at which I find the element
- What if we need to store **a very large number of elements**?
  - Memory reservation for **every possible** *key* → large memory occupance
- What if the the space/universe of keys is virtually unlimited?
  - E.g., any string?

Associative Array

Associative array

An ADS for representing dynamic sets containing **(key, value)** pairs such that each key is unique in the array. Associative array, also known as Dictionary or Map, supports **direct addressing**: computation of memory location of **value** directly from the **key**.

- Let **S** be the set (possibly infinite) of all allowed keys
  - Defined commonly with some primitive data type: `int`, `string`, `float`
  - This basically means that key can be **any** value of the primitive type
- Let K be the set of keys we would have in any concrete dynamic set
  - In most practical problems, K << |**S**|
  - If we would implement the dynamic set as the direct-access table (i.e., array), we'd have to allocate |S| slots in the memory and only K would be „used"
    - Waste of memory

Content

- ADS: Associative Array
- Hash Table
- Hash Functions

Hash table

Hash table

Hash table is a data structure (tightly coupled with a corresponding **hash function**) used for implementing **runtime and memory efficient** **associative arrays**.

- Direct-access tables store the element with **key** k at **index** k
- **Hash tables** store the element with key k at index *h*(k)
- ***h*(k)** is the **hash(ing) function** that computes the array index at which to store/find the element's value v directly from its key k
- Assuming a table (array) **T** of size m elements, *h* maps the universe of keys **S** to indices of the array

$$h\text{: } \mathbf{S} \rightarrow \{0, 1, \ldots, m\text{-}1\}$$

Hash table

- Direct-access tables would need |S| elements
- Because of the hashing (i.e., mapping) function *h*, Hash table T can have m << |S| elements
- Hashing reduces the range of the indices, i.e., the size of the array
  - Memory (space saving)
- **Caveat**: what if two keys „hash" to the same value? $h(k_2) = h(k_5)$
  - **Q**: How **likely** is that to happen? What does it depend on?

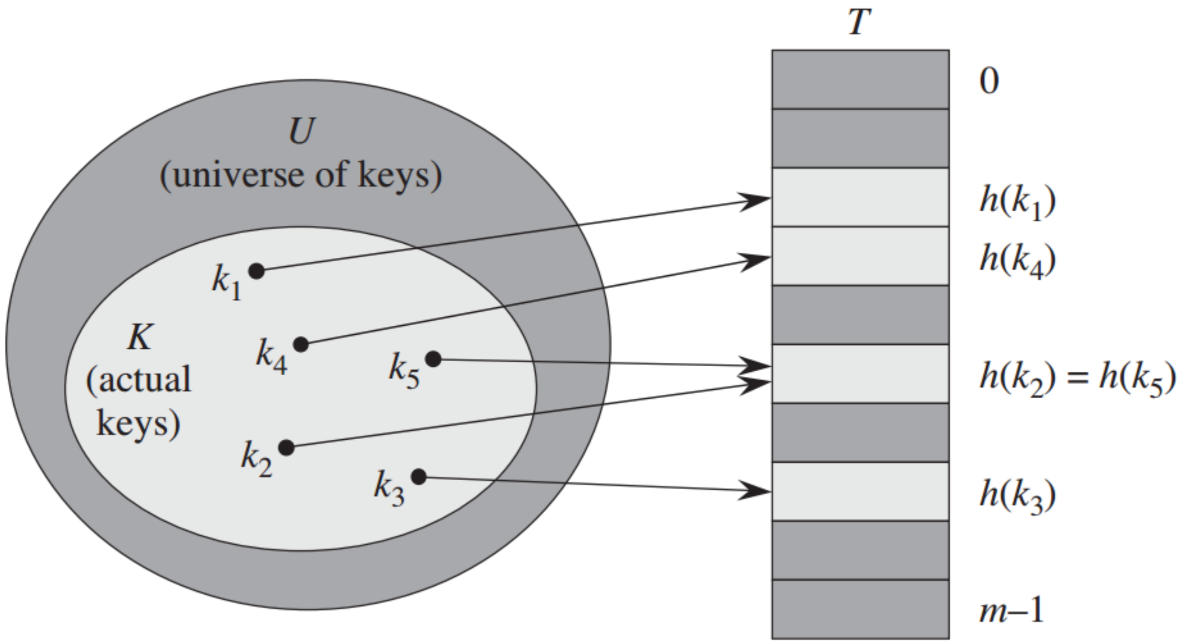


Image source: Cormen et al.

Collisions and hash functions

- **Collision**: when two (or more) keys get the same **hash,** $h(k_1) = h(k_2)$
- The frequency of collisions depends on
  - **(1)** The size m of the table (array) **T**
  - **(2)** The properties of the hashing function *h*
  - **(3)** The concrete set of keys to be hashed, **K** = $\{k_1, k_2, \ldots, k_K\}$
- Obviously, we want to **avoid** or at least **minimize** collisions
- We typically have no control over (3), some control over (1), and only full control over (2): the selection/design of the hash function

Collisions and hash functions

- **Crucial**: hash functions must be **deterministic**. What does that mean?
- We typically have no control over the set of keys **K** for which the values need to be stored in the hash table
  - Because of this, we cannot **guarantee** the absence of collisions
  - We can reduce the probability of collisions. How?

  - If m (size of T) $\geq$ |**K**| it is theoretically possible to store all allowed keys without collision
    - **Problem**: we don't actually know **K** in advance, we know only **S** (the universe of all possible/allowed keys)

Collisions and hash functions

- We typically have no control over the set of keys **K** for which the values need to be stored in the hash table
  - Because of this, we cannot **guarantee** the absence of collisions
  - **Q1**: If m (size of T) < |**K**| what is the minimal possible number of collisions?
  - **Q2**: If keys in **K** are integers {0, 1, 2, ..., |**K**|-1} and the size of the hash table T is m < |**K**|, find a hash *h* that guarantees this minimal number of collisions?
- Since we cannot guarantee no collisions, how do we handle them?

Collision resolution by chaining

- At each *hash slot* (*index in* {0, ..., m-1}, aka **bucket**) we put a pointer to the beginning of a **linked list**

- Slot/bucket *i* ∈ {0, ..., m-1} contains **a pointer to the head** of the linked list of all stored elements whose keys hash to *i*
  - If no keys have been hashed to *i*, the pointer is `null`

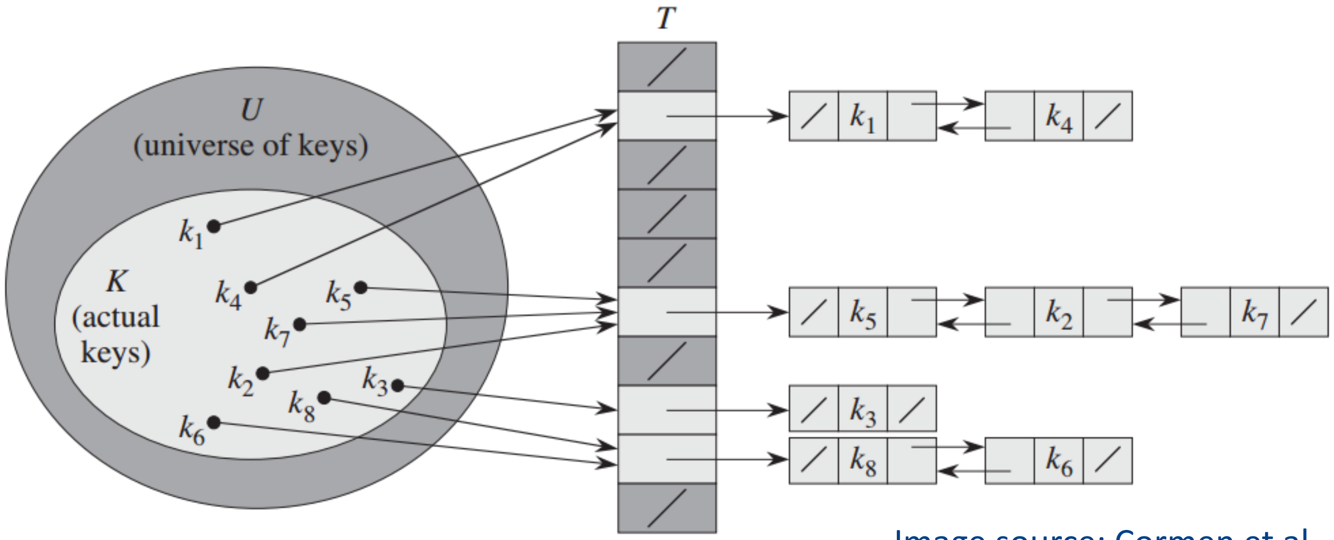


Image source: Cormen et al.

Collision resolution by chaining

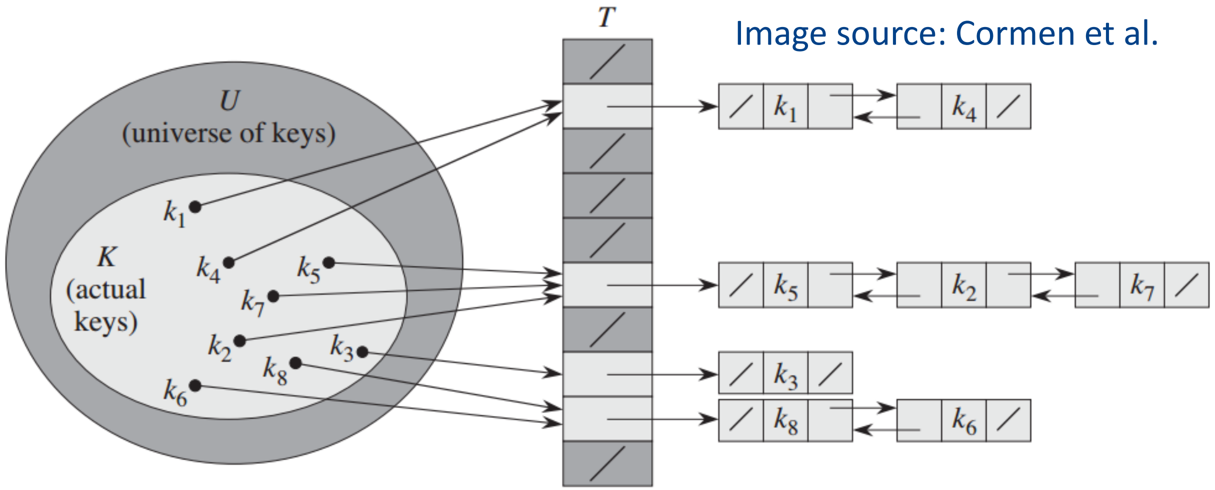


Image source: Cormen et al.

- Let x be a set element with key x.*key* and value x.*value*
  - x.*next* the pointer to the next element in the collision chain of the slot
- Let T be the hash table with collision resolution by chaining
- **INSERT**: add to the beginning of the chain

```
Insert(T, x):
    pointer = T[h(x.key)]
    x.next = pointer
    pointer = address(x)
```

Q1: Worst case running time?

Q2: What if x already in the list?

Collision resolution by chaining

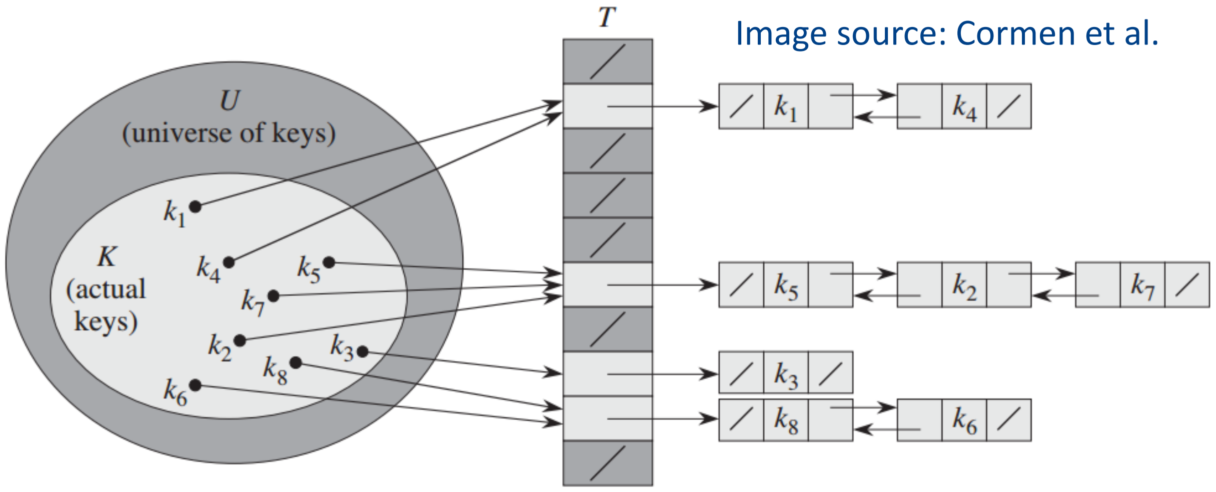


Image source: Cormen et al.

- Let x be a set element with key *x.key* and value *x.value*
  - *x.next* the pointer to the next element in the collision chain of the slot
- Let T be the hash table with chaining resolution of collisions

- **SEARCH**: find in the linked list of the chain

```
Search(T, key):
  pointer = T[h(key)]
  while pointer != null
    x = read_memory(pointer)
    if x.key == key
      return x # or x.value
    else
      pointer = x.next
  return null
```

Q: Worst case running time?

Collision resolution by chaining

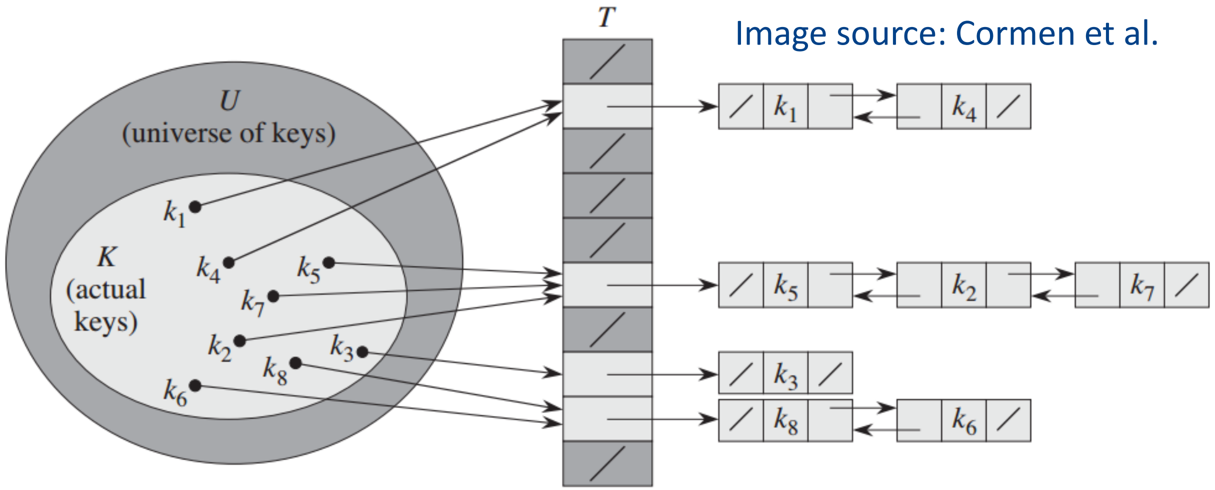


Image source: Cormen et al.

- Let x be a set element with key *x.key* and value *x.value*
  - x.*next* the pointer to the next element in the collision chain of the slot
- Let T be the hash table with chaining resolution of collisions
- **DELETE**: find in the linked list of the chain

```
Delete(T, key):
  pointer = T[h(key)]
  while pointer != null
    x = read_memory(pointer)
    if x.key == key
      # & - at the address where the pointer points
      # we set the value x.next
      &pointer = x.next
      return
    else
      pointer = x.next
```

Q: Worst case running time?

Collision resolution by chaining

- **Runtimes** for hash tables with chaining
- **Load factor**: $\alpha = |K| / m$
  - The average number of elements chained in linked lists (chains)
  - Runtime analysis in terms of α

- What is the **worst case** for hashing with chaining?
  - A hash function *h* that would map all $n = |K|$ keys into the same hash slot
  - `Search` and `Delete`: **O(n)**
  - `Insert` is always **O(1)**

Collision resolution by chaining

- **Runtimes** for hash tables with chaining
- **Load factor**: $\alpha = |K| / m$
  - The average number of elements chained in linked lists (chains)
  - Runtime analysis in terms of α

- What is the **average case** for hashing with chaining?
  - Depends on how well the hashing function h distributes keys across the m hash slots/buckets
  - Assumption: **simple uniform hashing**
    - *h* equally likely to map a key to any of the m buckets
    - Buckets will have roughly the same number of elements: α
    - Runtime of `Search` (and `Delete`) is **O(α)**

Content

- ADS: Associative Array
- Hash Table
- Hash Functions

Hash functions

- What would be a **good** hash function *h*?
  - One that creates the minimal number of collisions
- If it satisfies the assumption of **simple uniform hashing (SUH)**, it will create a small number of collisions
  - Keys are equally likely to be hashed into any of the m buckets, independently of where the other keys have been hashed
  - **As if** you were randomly drawing a bucket for every key
    - Though we cannot do that, as this is not deterministic
  - If **SUH**, then we can always reduce the runtime (of `Search`) by increasing m: for large enough m (**trading space for time**), **O(1)**
  - Unfortunately, no easy way to prove/check if some *h* results with **SUH**
    - It also depends on the actual keys being hashed

Designing hash functions

- **Heuristic design**
  - Kind of **„trial and error"** – we create *h* that „makes intuitive sense"
  - Then we test it on various key sets **K** to see whether it roughly exhibits the SUH property – that all bucket chains are similarly long, roughly $\alpha = |\mathbf{K}|/m$
- Most hashing functions assume **integer keys**
  - Other types (e.g., `float`, `string`) are converted to (natural) integers first
  - For a `string`, let's assume a charset of **N** characters (e.g., N = 128)
    - Each character corresponds to one integer between 0 and 127
    - „$c_1c_2c_3$" -> $128^2 * c_1 + 128 * c_2 + c_3$

Hash function: division method

- Hash is the **remainder of the division** of the key with the size of the array m: ***h*(key) = key % m**

- It is common to avoid certain values of m, as this can violate the assumption of **simple uniform hashing**
  - Or make the violation more likely for arbitrary key sets
  - For example, we avoid $m = 2^p$ because the value $h(key) = key \% 2^p$ depends only on the lowest p bits of the key
    → this puts all keys with the same lowest p bits into the same bucket

- **Good choice** for m: a prime number not too close to any power of 2
  - Example: 3000 keys, and we'd be ok with average chains of length $\alpha = 4$
  → Good m would be 751 or 757

Hash function: multiplication method

- Simple heuristic hashing like division, but choice of m less critical
  1. We choose a multiplier real number M, $0 < M < 1$
  2. We multiply the key k with M and take the **decimal remainder** r
     - We will denote the decimal remainder of a float f as f%1
  3. We multiply r with the hash table size m and take the first integer smaller than that number
     - We denote the first integer smaller than a float f with $\lfloor f \rfloor$ („floor” of f)

$$h(k) = \lfloor (k * M \% 1) * m \rfloor$$

- For **multiplication hashing**, it’s actually desirable that $m = 2^p$. **Q**: Why?

Hashing functions: **universal hashing**

- Multiplication and division hashing lead to uniform hashing as long as the input keys to be hashed are not „rigged" in a particular way
- Assume there's an **adversary** who wants to slow down your program that works with a hash table
  - By making **all/many** keys hashed to the same value
  - If they guess your hash function, they can easily do that
  - E.g., if they know you use division method with $m = 751$, they can send keys that leave the same remainder when divided with 751: {2, 753, 1504, ...}
- **Universal hashing**: no fixed hashing function
  - In every execution, choosing h randomly from a set/family of **carefully designed** hash functions H (but in a way **independent** of the keys)
  - Algorithm behaves differently in every execution, so no single input will always cause worst case running time

Re-Hashing

- Assume we set a hash table of size m in advance and then start hashing incoming keys and store corresponding values

- Assume we receive $|\mathbf{K}| = n >> m$ to hash, so $\alpha = n/m$ becomes larger
  - Search becomes slower

- **Re-hashing**
  - If we could increase m, that would reduce α
  - Creating a new hash table with larger m, re-hashing all existing keys
  - Trading space for time

Questions?

Awọn ibeere?
ਸਵਾਲ?
Küsimusi?
Turite klausimų?
Pitanja?
Sorusu olan?
Dúvidas?
質問は？
有问题吗？
Fragen?
¿Preguntas?
Questions?
Frågor?
Pytania?
Domande?
Питання?
Vragen?
Ερωτήσεις;
Porandukuéra?
คำถาม?
أسئلة؟