Microsoft

Game of Learners

AFRICA DEVELOPMENT CENTER

Dictionaries

Dictionaries/Hash-maps and Sets/Hash-sets

Anthony Nandaa // October 2024

Preamble

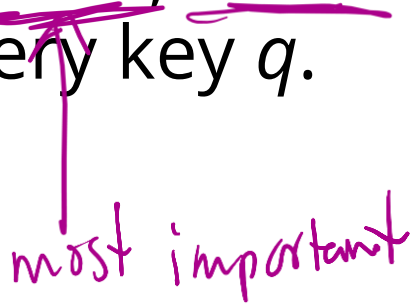
- Passing interviews is a great thing, but after that is when the real journey begins.
- It’s better to **prepare to be a great software engineer**, that kills two birds with one stone.

Outline

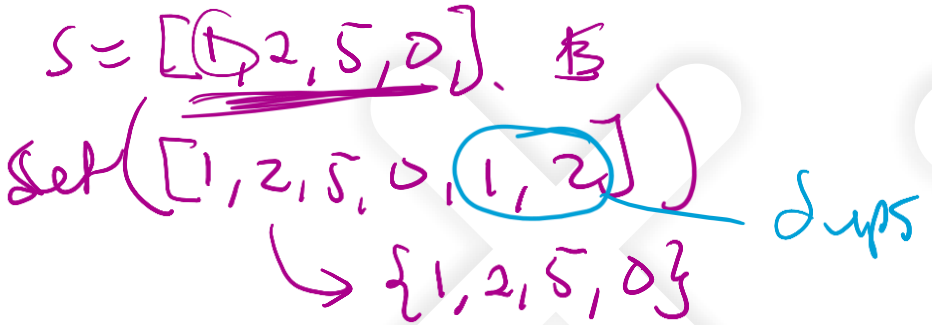
+ Introduction
+ Implementations
+ ~~Real-world applications (TBD)~~

Introduction (1)

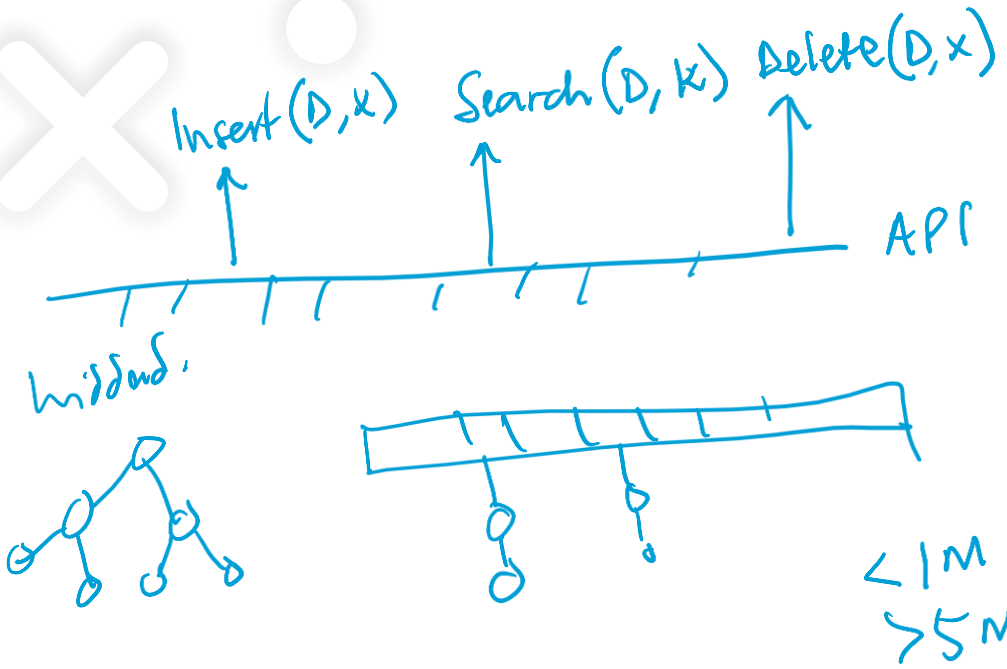
- **Input description:** A set of *n* records, each identified by one or more key fields.
- **Problem description:** Build and maintain a data structure to efficiently locate, insert and delete the record associated with any query key *q*.

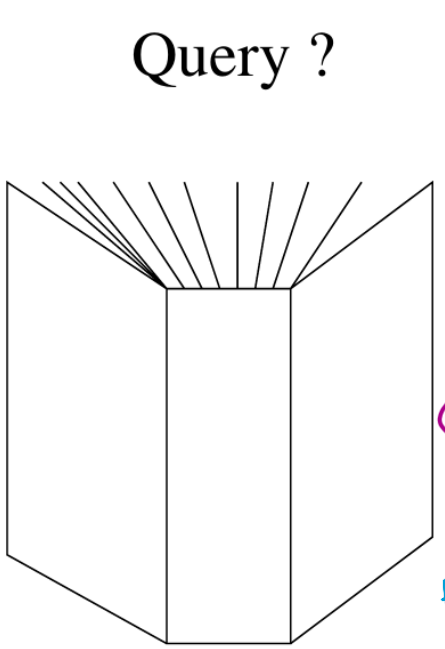


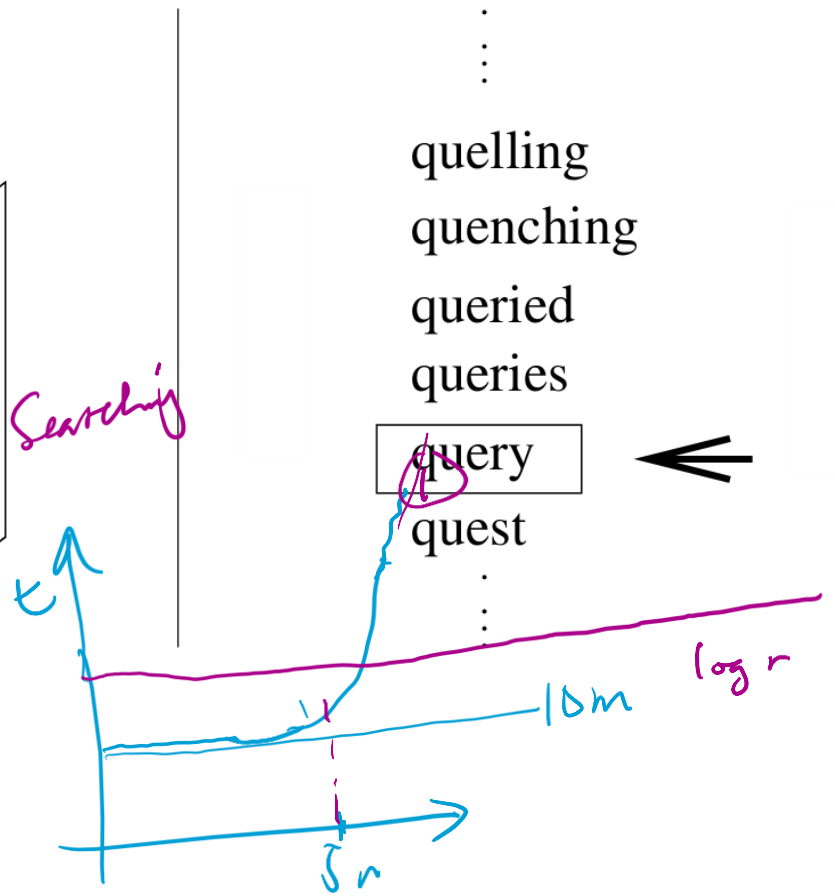
Introduction (2)



- The dictionary data structure type permits *access to data items by content*. You stick an item into a dictionary so you can find it when you need it.
- The primary operations of dictionary support are:
  - *Search(D, k)* - given a search key, *k*, return a pointer to the element in dictionary, D whose key values is *k*, if one exists.
  - *Insert(D, x)* - given a data item *x*, add it to the set in the dictionary D.
  - *Delete(D, x)* - given a pointer to a given data item *x* in the dictionary D, remove it from D.
  - *Max(D)* or *Min(D)* - retrieve the item with the largest (or smallest) key from D. This enables the dictionary to serve as a priority queue.
  - *Predecessor(D, k)* or *Successor(D, k)* - retrieve the item from D whose key is immediately before (or after) *k* in sorted order. These enables us to iterate through the elements of the data structure.





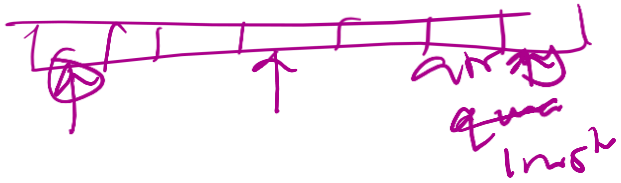


Implementations

Implementation Intro

- An essential piece of advice is to carefully isolate the implementation of the dictionary data structure from its interface.
  - Use explicit calls to methods or subroutines that initialize, search and modify the data structure, rather than embedding them within the code. This is much cleaner and also makes it easy to experiment with different implementations to see how they perform.

Considerations



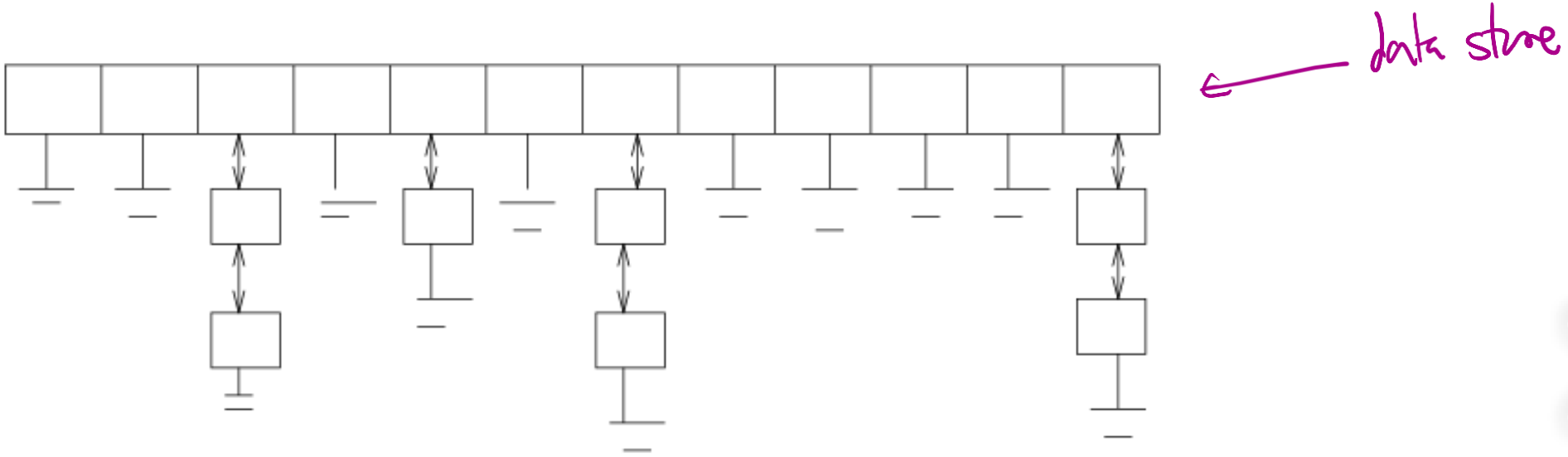
- In choosing the right data structure for your dictionary, ask yourself the following questions:
  - How many items will you have in your data structure?
  - Do you know the relative frequencies of inserts, delete, and search operations?
  - Can we assume that the access pattern for keys will be unform and random?
  - Is it critical that individual operations be fast, or only that the total amount of work done over the entire program be minimized?

Common implementations

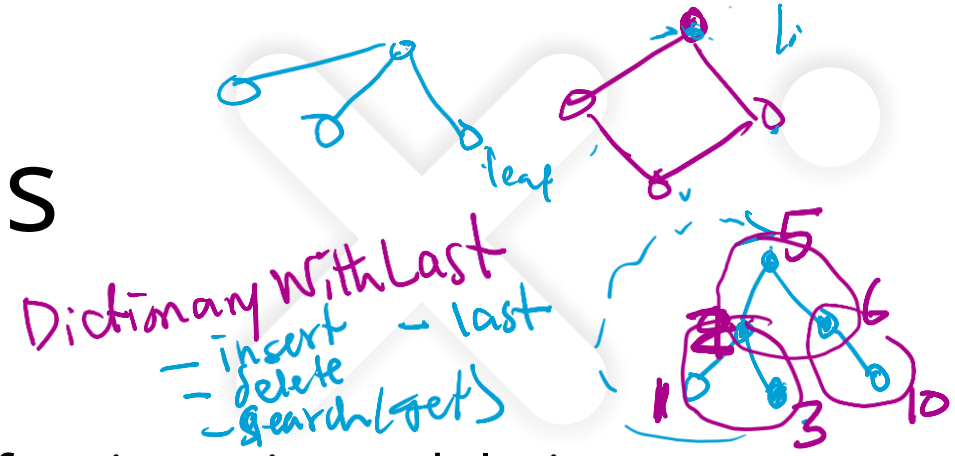
- Unsorted linked lists or arrays – simple
  - For small data sets, e.g. 50 to 100 items.
  - Useful variant is the *self-organizing list:* whenever a key is accessed or inserted, we always move it to the head of the list. Thus, if the key is accessed again sometime in the near future, it will be near the front and so require only a short search to find it.
- Sorted linked lists or arrays:
  - A sorted array will only be appropriate if and only if there are not many insertions or deletions.

Common implementations

- **Hash tables:**
  - For applications involving a moderate-to-large number of keys (say 100 to 10 M), a hash table is probably the right way to go.
  - To be discussed in detail next.



Common implementations



- **Binary search trees (BST):**
  - BSTs are elegant data structures that support fast insertions, deletions and queries.
  - The big distinction between different types of trees is whether they are explicitly rebalanced after insertion or deletion, and how the balancing is done.
  - Balanced search trees use local *rotation* operations to restructure search trees moving more distant nodes closer to the root while maintaining the in-order search structure of the tree.
  - Popular among balanced search trees are ***red-black trees***.
  - An interesting self-organizing data structure is the ***spray tree***, which uses rotations to move any accessed key to the root. Frequently used or recently accessed nodes thus sit near the top of the tree, allowing faster searches.

Common implementations

- **B-trees:**
  - For data sets so large that they will not fit in main memory (say more than 1 M items), your best bet will be some flavor of a B-tree.
  - Once a data structure has to be stored outside of main memory, the search time grows by several orders of magnitude.
  - The idea behind a B-tree is to collapse several levels of a binary search tree (BST) into a single large node, so that we can make the equivalent of several search steps before another disk access is needed.
  - With B-tree we can access enormous numbers of keys using only a few disk accesses.

Common implementations

- **Skip lists:**
  - *Out of scope, a good take-home read!*

Implementation in Language Libs (1)

- Modern programming languages provide libraries offering complete and efficient container implementations. For examples:
  - **Python:**
    - `dict`
    - `set`
    - `collections.defaultdict`
  - **Java:**
    - `HashMap` (`java.util.HashMap`) – hash table implementation
    - `HashSet` – implements a set backed by a hash map
    - `LinkedHashMap` – a version of HashMap that maintains insertion order, useful when order preservation is necessary.
    - `TreeMap` – an implementation of a Red-Black Tree, allowing for sorted key-value pairs, with *O(log n)* lookup.

Implementation in Language Libs (1)

+ JavaScript:
  - *Object*: JS objects act as dictionaries, with properties acting as keys and values, offering constant time lookups.
  - *Map*: introduced in ES6, `Map` provides an explicit dictionary structure with better support for non-string keys and maintains insertion order.
  - *Set:* also introduced in ES6, it is used for collections of unique values.
+ C++:
  - `std::unordered_map`: A hash tale that allows fast access to elements using a hash function.
  - `std::map`: implements a binary search tree with *O(log n)* lookups, storing elements in sorted order.
  - `std::unordered_set`: implements a hash set, storing elements with average O(1) lookup time.

Assignment #2

- Check the source code for the implementation of a hash-table, set or dictionary in your favorite programming language. Discuss the design decisions in the next session.

Array Implementation

- The basic dictionary operations can be implemented with the following costs on sorted and unsorted arrays:

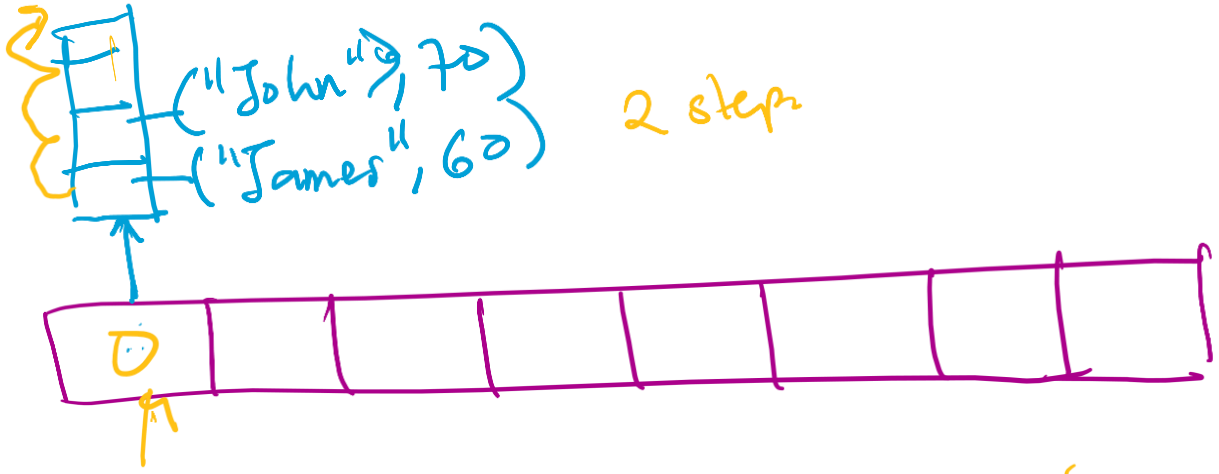
Dictionary operation	Unsorted array	Sorted array
Search(L, k)	$O(n)$	$O(\log n)$
Insert(L, x)	$O(1)$	$O(n)$
Delete(L, x)	$O(1)^*$	$O(n)$
Successor(L, x)	$O(n)$	$O(1)$
Predecessor(L, x)	$O(n)$	$O(1)$
Minimum(L)	$O(n)$	$O(1)$
Maximum(L)	$O(n)$	$O(1)$

Array Implementation

- *Deletion* is somewhat trickier, hence the *. The definition states that we are given a pointer x to the element to delete, so we need not spend any time searching for the element.
  - But removing the *xth* element from the array A leaves a hole that must be filled. We could fill the hole by moving each of the elements *A[x+1]* to *A[n]* up one position, but this requires *O(n)* time when the first element is deleted. The following idea is better: just write over *A[x]* with *A[n]*, and decrement *n*. This only takes constant time.

Hash-maps

- **Hashing and Strings (1):**
  - Hash tables exploit the fact that looking an item up in an array takes constant time once you have its index.
  - A hash function is a mathematical function that maps keys to integers.
  - We will use the value of our hash function as an index into an array, and store our item at that position.

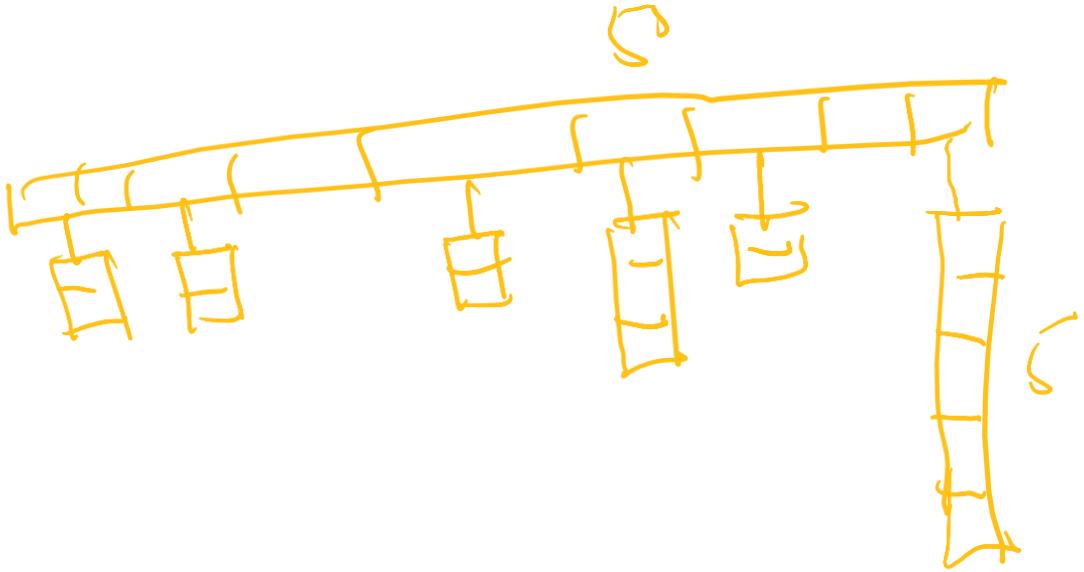


H James → f → 0

John → → 0

H.(John) → 0 (O(1).

~O(1)



Hash-maps

- **Hashing (2):**
  - The first step of the hash function is usually to map each key to a big integer. Let *α* be the size of the alphabet on which a given string *S* is written. Let *char(c)* be a function that maps each symbol of the alphabet to a unique integer from 0 to *α - 1*. The function:

$$H(S) = \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \times char(s_i)$$

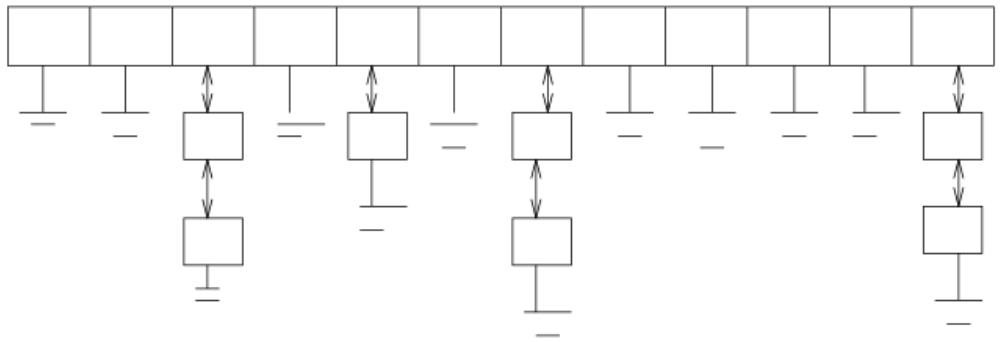
    - maps each string to a unique (but large) integer by treating the characters of the string as “digits” in a *base - α* number system.

Hash-maps

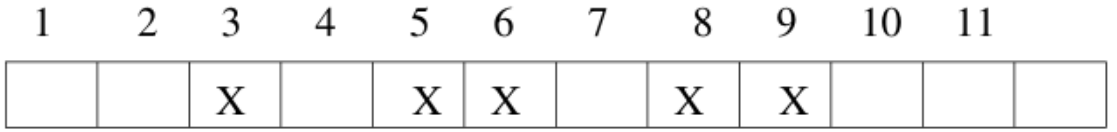
- **Hashing (3):**
  - The result is unique identifier numbers, but they are so large they will quickly exceed the number of slots in our has table (denoted by m). We must reduce this number to an integer between 0 and $m - 1$, by taking the remainder of $H(S) \bmod m$.
  - If the table size is selected with enough finesse (ideally m is a large primer not too close to $2^i - 1$), the resulting hash values should be faily uniformly distributed.

Hash-maps

- **Hashing (4):**
  - **Collusion Resolution:**
    - No matter how good our hash function is, occasionally two distinct keys will hash to the same value.
    - ***Chaining*** is the easiest approach to collision resolution. Represent the hash table as an array of m linked lists. The *ith* list will contain all the items that has to the value of *i*.



Hash-maps



- **Hashing (4):**
  - **Collusion Resolution:**
    - ***Open addressing*** is the alternative. The hash table is maintained as an array of elements (not buckets), each initialized to null. On an insertion, we check to se if the desired position is empty. If so, we insert it. If not, we must find some other place to insert it instead. The simplest possibility (called ***sequential probing***) inserts the item in the next open spot in the table.
    - Searching for a given key now involves going to the appropriate hash value and checking to see if the item there is the one we want. If so, return it. Otherwise we must keep checking through the length of the run.
    - Deletion can get ugly, since removing one element might break a chain of insertions, making some elements inaccessible. We therefore have to reinsert all tehe items in the run following the new hole.

Hash-maps

- ***Summary:*** We use a function that maps keys (be it strings, numbers or whatever) to integers between *0* and *m – 1*. We maintain an array of *m* buckets, each typically implemented using an unsorted linked list.
- If we use hash function that spreads the keys out nicely, and a sufficiently large hash table, each bucket should contain very few items, thus making linear searches acceptable.
- Insertion and deletion from a hash table reduce to insertion and deletion from the bucket/list.
- **Tip:** Regardless of which hash function you decide to use, print statistics on the distribution of keys per bucket to see how uniform it *really* is.

References

+The Algorithm Design Manual, Steven Skiena

+https://github.com/golclinics/golclinics-dsa - more listed here

Thanks

+ **Subscribe** to my YouTube channel https://youtube.com/@blackbored
+ **Email:** hey@nandaa.dev
+ **Homepage**: https://nandaa.dev

Take-home Assignments

Assignment #1

- Read up on *Skip Lists* and how they are used to implement dictionaries.

Assignment #2

- Check the source code for the implementation of a hash-table, set or dictionary in your favorite programming language. Discuss the design decisions in the next session.
- Can do a write-up or even a blog-post!

Assignment #3

- Source code ref: https://gist.github.com/profnandaa/30670efbcf77bbbc1475833cf61dd4fd
- Extend the code done in class to include *delete()* and *display()* aka. *__str__()* methods.

Assignment #4

- Read up on how dictionaries are implemented with binary search trees (BST).
- Attempt to make a new implementation of the same interfaces we had in our class example, but with BSTs (Red-Black Tree for example).

Assignment #5

- Implement DictionaryWithLast – that does all the dictionary operations but additionally with *last()* method that returns the last key that was accessed.
  - d = DictionaryWithLast()
  - d.insert(“James”, 10)
  - d.insert(“John”, 20)
  - d.last() – returns “John”
  - d.get(“James”)
  - d.last() – returns “James”
  - **Tip:** explore using a *Spray Tree*.