Algoritmos y Estructuras de Datos

Tema 4: Estructuras de Datos

Grado Imat. Escuela ICAI

Juan C. Aguí García

January 2024

Estructuras de Datos: Interfaz e Implementación

Las estructuras de datos son un conjunto organizado de datos y algoritmos que soportan ciertas operaciones sobre los datos.

Interfaz (API)	**Estructura de Datos**
Especificación que describe las Operaciones soportadas sobre los. Describen la funcionalidad de la estructura de datos.	Algoritmos que soportan las operaciones. Estructura de almacenamiento de la información, y código de implementación interna de las mismas.
$\Rightarrow$ **El Qué**	$\Rightarrow$ **El Cómo**

La implementación de las estructuras de datos está optimizada al modelo de computación y al lenguage de programación soportado, cambiando drásticamente
La interfaz, sin embargo, es mucho más estable y común entre plataformas

Tipos Básicos de Estructuras: Secuencias y Sets

Estudiaremos dos interfaces clave, **Secuencias**, y **Sets**, ambas enfocas a la gestión de grandes conjuntos de elementos (Magnitudes, Objetos, Registros, etc...) habitualmente homogéneos[1]

Secuencias El ordenamiento es extrínseco a los datos, el direccionamiento está orientado a índices. Ej: $(x_0, x_1, x_2 ... x_{n-1})$. Ojo **Zero indexing**
Tipos Básicos: Arrays Státicos, Arrays Dinámicos, Listas, Pilas, Colas

Sets El ordenamiento es intrínseco a los datos. El direccionamiento de los mismos es en base a una **key**. (Ojo, la **key** puede ser el mismo objeto en sí)
Tipos Básicos: Python Sets, Diccionarios

[1] aunque no necesariamente

Especificación de las Interfaces: Secuencia

Para las secuencias, la definición de la interfaz es la siguiente:

Estructura	*build*(*X*)	construye la secuencia en base al iterable X
	len()	devuelve el número de elementos en la secuencia
Estáticos	get_at(i)	retorna el i^{th} elemento
	set_at(i,x)	reemplaza el i^{th} con x
	iter_seq()	retorna los elementos de la secuencia, de uno en uno, en su orden
Dinámicos	insert_at(i,x)	Añade x como el i^{th} elemento
	delete_at(i)	Elimina y devuelve el i^{th} elemento
	insert_first(x)	Añade x como primer elemento
	insert_last(x)	Añade x como último elemento
	delete_first()	Elimina el primer elemento
	delete_last()	Elimina el último elemento

La interfaz

Especificación de las Interfaces: Set

Para los Sets, la definición de la interfaz es la siguiente:

Estructura	*build*(*X*)	construye la secuencia en base al iterable X
	len()	devuelve el número de elementos en la secuencia
Estáticos	find(k)	retorna el item asociado a la clave *k*
Dinámicos	insert(x)	Añade *x* al set, reemplanzdo el objeto con clave *x.key* si existe
	delete(k)	Elimina el objeto con clave k
Orden	iter_ord()	Iterador que produce los elementos en el orden de k
	find_min()	Retorna el objecto con la clave más pequeña
	find_max()	Retorna el objecto con la clave más grande
	find_next(k)	Retorna el objeto con la clave siguiente a k
	find_prev(k)	Retorna el objeto con la clave siguiente a k

Python Arrays implementation

Las estructuras básicas de Python implementan de forma eficiente, una versión de esta interfaz.

List A collection which is ordered and changeable. Allows duplicate members.

Tuple A collection which is ordered and unchangeable. Allows duplicate members.

Set A collection which is unordered, unchangeable*, and unindexed. No duplicate members.

Dictionary A collection which is ordered and changeable. No duplicate members. Key based

Python hint

En el intérprete de python, ejecuta: `help(Set)` o `help(list)`, etc para ver el detalle de la interfaz de las clases de python

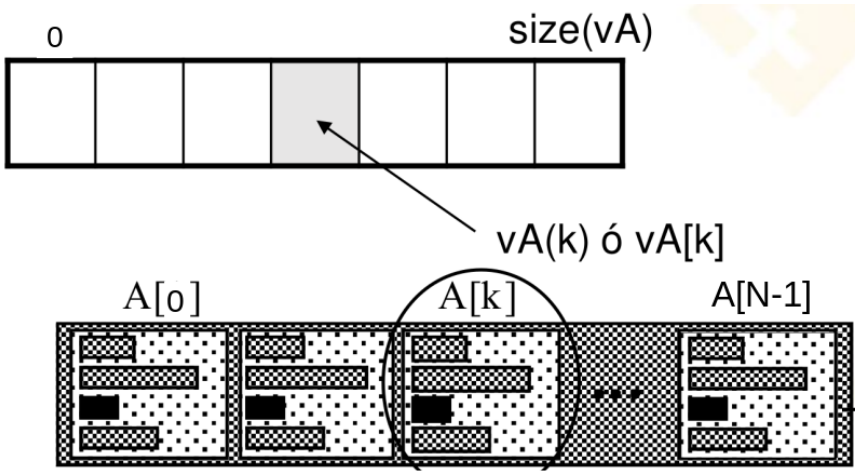
Our roadmap into basic data Structures

- **Static Arrays. Python arrays**
  Espacios de memoria estáticos y contiguos que se se direccionan en una o varias dimensiones
- **Dynamic Arrays List**
  One, or n-dimensional array that works as an extensible, dynamic collection of pointers to general objects. Generally known as lists in Python
- **Linked List**
  A collection of objects where everyone is connected to the next of previous, an in a chain manner. Can be single, double, or circular linked list. Not a Python object, the structure is a wrapper or container for data objects.
- **Key Generation: Hashing**
  Mapping the world of general *keys* to numbers that can be used to store pairs of (key, object) using general structures like the above
- **Sets and Dictionaries**
  Set are unordered list of unique objects. Dictionaries allow the storage and addressing of objects by key. Sets are dictionaries where there is no value, just the key

Static Arrays

Estructura básica fundamental disponible en casi todos los lenguajes.

- Suele implementarse como un pointer a un espacio **fijo** de memoria, en el que se almacenan, consecutivamente, entidades homogéneas de tamaño constante (int, float, Complex, estructuras de datos complejas, etc..)
- Direccionamiento rápido (Simple aritmetica de pointers a memoria. $loc(x[i]) = loc(x) + sizeOf(x) \times i$
- Los Arrays estáticos tiene tamaño fijado en su momento de creación. Redimesionamiento puede ser tan costoso como su construcción.
- Buffers en C, array en Python, matrices en Fortran, etc...



Static Arrays: Performance

La performance de las arrays estáticas es buena para los get/set, pero es poco elástica por que la inserción o borrado de un elemento precisa de la regeneración $\mathbf{O}(N)$ del array en su conjunto.

Data Structure	Coste Operación $\mathbf{O}(.)$				
	Container	Static	Dynamic		
	build(X)	get_at(i) set_at(i)	insert_first(x) delete_first(x)	insert_last(x) delete_last(x)	insert_at(i, x) delete_at(i)
Static Array	$\mathbf{O}(N)$	$\mathbf{O}(1)$	$\mathbf{O}(N)$	$\mathbf{O}(N)$	$\mathbf{O}(N)$

Python hint

Ver módulo Array en Python. Execute:

```
import array
help(array.array)
```

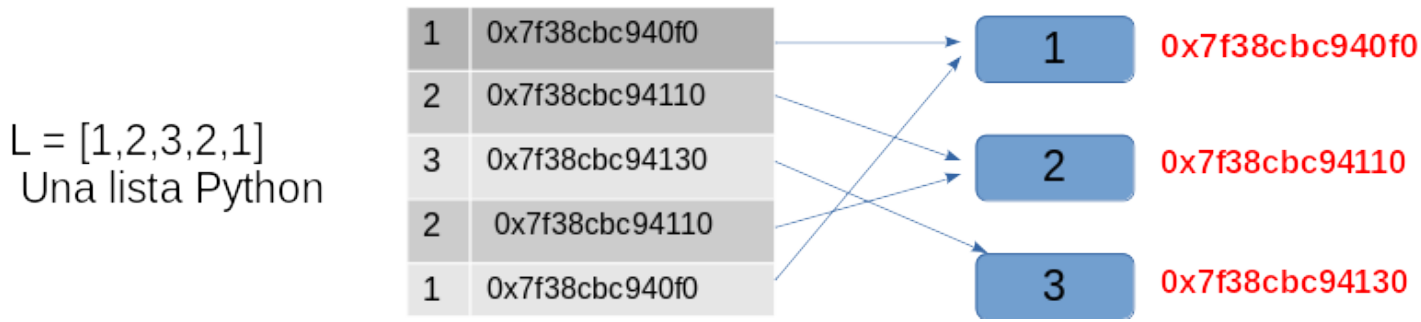
Dynamic Arrays: Lists

Estructura de datos que implementa la interfaz de **Secuencia** de forma mutable y dinámica

- **Mutable**. Lo elementos de la lista puede ser intercambiados `Ej. a[i] = 2 ; a[i] = 3`
- **Dinámica** Gestiona de forma dinámica las inserciones, y borrados (del, append, insert).

Los elementos de una lista son objetos de cualquier tipo (en realidad la **lista gestiona una colección de pointers** a ellos:

Ej: `a =['Nombre´, edad, foto]` siendo nombre un string, edad: int, y foto un puntero a un objeto imagen.



Dynamic Arrays: Lists Performance

Cómo funcionan las listas en Python ?

- **Idea!** Alocar espacio ´extra' tal que las operaciones dinámicas no requieran reorganziación de los datos y se puedan realizar en $\mathbf{O}(1)$
- Cuando el espacio alocado se llena, re-alocar con $\Theta(n)$ espacio adicional ($\sim N/2$)
- Cuando esto ocurren , una simple operación costará $\mathbf{O}(N)$, sin embargo una secuencia de $\mathbf{O}(N)$ operations costará, en media $\mathbf{O}(N)$ tiempo.
- Por tanto, en media, el **coste amortizado** de la operaciones dinámicas ser á de $\mathbf{O}(1)$

Data Structure	Coste Operación $\mathbf{O}(.)$				
	Container	Static	Dynamic		
	build(X)	get_at(i) set_at(i)	insert_first(x) delete_first(x)	insert_last(x) delete_last(x)	insert_at(i, x) delete_at(i)
Static Array	$\mathbf{O}(N)$	$\mathbf{O}(1)$	$\mathbf{O}(N)$	$\mathbf{O}(N)$	$\mathbf{O}(N)$
Dynamic Array	$\mathbf{O}(N)$	$\mathbf{O}(1)$	$\mathbf{O}(N)$	$\mathbf{O}(1)_a$	$\mathbf{O}(N)$

Arrays multidimensionales

- Elementos de una lista pueden ser, a su vez, listas.
  A = [1,[2,3],[5,2,8,4]]
- Por tanto, un Array Bidimensional es **una lista de listas**

$$A = [[a,b],[c,d]] \Rightarrow A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$A[0][1] = b \; ; \; A[0] = [a,b]$$

- Sin embargo, estas matrices, aunque son muy ágiles, **No son eficientes** Usa las arrays del módulo **Numpy** que gestiona eficientemente matrices multidimensionales con datos **homogéneos**

Python hint

Ver módulo Numpy en Python. Execute:

```
import numpy

help(numpy)
```

Consulta `https://numpy.org/`

Tuplas: Listas inmutables

- Una tupla no puede modificarse de ningún modo después de su creación (no se pueden añadir, modificar, o eliminar elementos)
- Una tupla se visualiza del mismo modo que una lista, salvo que el conjunto se encierra entre paréntesis en lugar de entre corchetes.
- Formas de crear una tupla:
  - Mediante el formato de tupla directamente
    `aTupla = (1,'a',2,"bc")`
  - Mediante el constructor de la clase `tuple`, que se alimenta de cualquier iterable:
    `aTupla = tuple([1,'a',2,"bc"])`
    `aTupla1 = tuple(range(1,10,2))`
- Los elementos de una tupla tienen un orden definido, como los de una lista. Los mismos conceptos de indexación y slicing de las listas se aplican aquí[2]

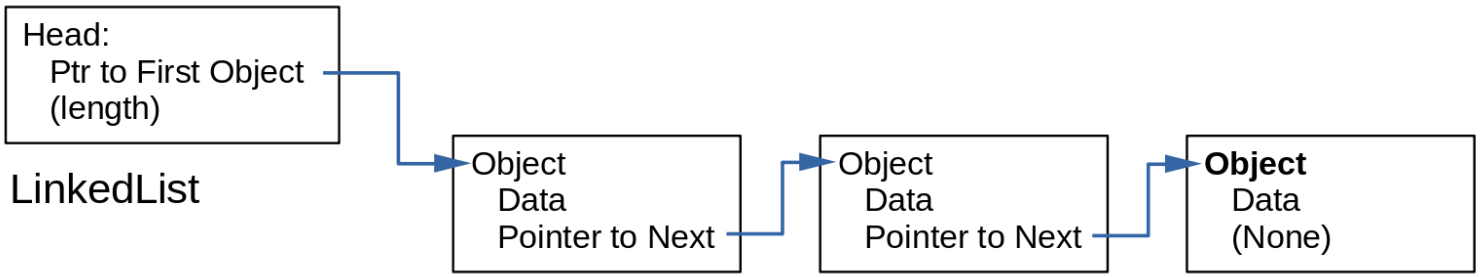
  `print (f"aTupla1[4],aTupla[2:]")` $\Rightarrow$ `9,(2,"bc")`

[2] Good Programmers use **named tuples** as simple lightweight objects. Google it !!

Listas Enlazadas (o "Linked-List")

Estructura de datos que implementa una secuencia de objetos, y que se estructura en base a:

- La cabeza (head) que contiene el puntero al primer objeto (o None) y (Opcionalmente) el número de objetos en la secuencia
- El puntero que cada objeto tiene hacia el objeto siguiente en la secuencia



- Las listas enlazadas se benefician del Programación Orientada a Objetos, donde el Objeto es un nodo, que incluye los datos (otro objeto ??) y la estructura de punteros que forman la lista enlazada.

Python Hint

Ver `https://realpython.com/linked-lists-python/`

Listas Enlazadas: Algoritmos básicos

Insert First

insertion in a LinkedList at the first position

```
function LL_INSERT_FIRST(data)
    newNode ← LLNODE(data)                          ▷ Wrap data in LLNode
    if self.firstNode != None  then                 ▷ If LL Not empty
        newNode.nextNode ← self.firstNode           ▷ link existing node after newNode
    end if
    self.firstNode ← newNode                        ▷ Place NewNode first
    increase self.length
    return
end function
```

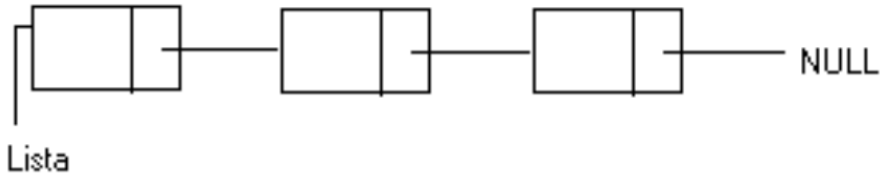
Delete position i

Removing (not deletion) of node at position i

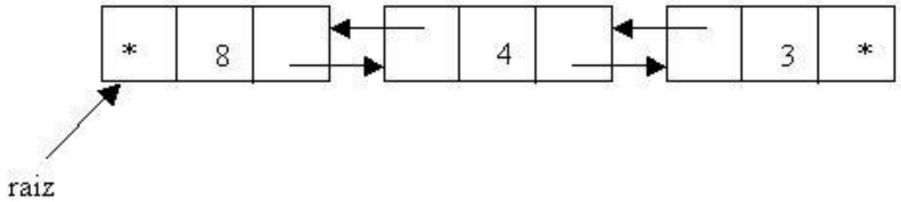
```
function LL_DELETE_AT(i)
    baseNode ←LL_GETNODE(i-1)                       ▷ Get (i-1)th node O(i) ∼ O(N)
    baseNode.nextNode ← baseNode.nextNode.nextNode  ▷ O(1)
    decrease self.length
    return
end function
```

Listas Enlazadas: tipos

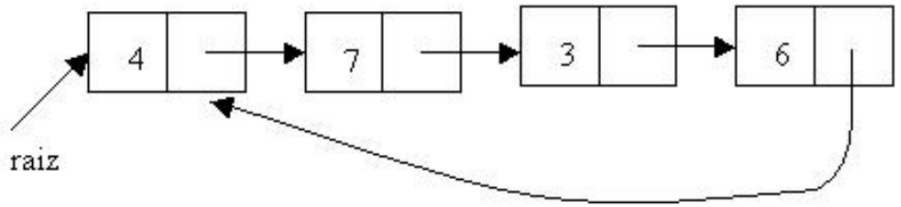
1. Simples



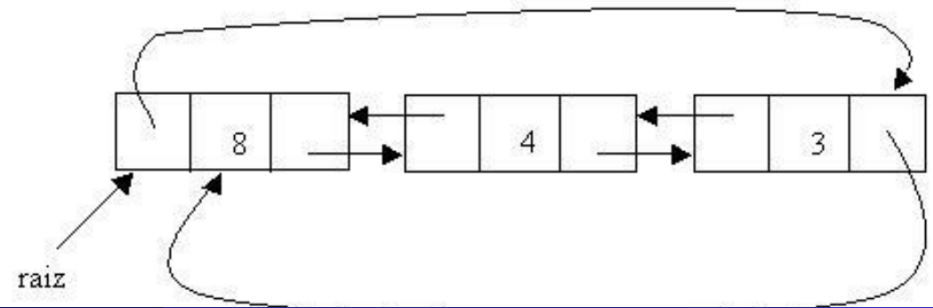
2. Doblemente enlazada



3. Circulares



4. Dobles Circulares



Linked Lists: performance

Las listas enlazadas soportan operaciones dínamicas de forma diferente a las estructuras secuenciales anteriores:

- permiten inserciones en la cabeza en $\mathbf{O}(1)$
- Los borrados e inserciones en posicion i cuestan de $\mathbf{O}(i) \sim \mathbf{O}(N)$ por el coste de navegación hasta el objeto
- si ya ubicamos el elemento en la lista, los borrados/inserciones son $\mathbf{O}(1)$ ya que es un mero reajuste de punteros.

Data Structure	Coste Operación $\mathbf{O}(.)$				
	Container	Static	Dynamic		
	build(X)	get_at(i) set_at(i)	insert_first(x) delete_first(x)	insert_last(x) delete_last(x)	insert_at(i, x) delete_at(i)
Static Array	$\mathbf{O}(N)$	$\mathbf{O}(1)$	$\mathbf{O}(N)$	$\mathbf{O}(N)$	$\mathbf{O}(N)$
Dynamic Array	$\mathbf{O}(N)$	$\mathbf{O}(1)$	$\mathbf{O}(N)$	$\mathbf{O}(1)_a$	$\mathbf{O}(N)$
Linked Lists	$\mathbf{O}(N)$	$\mathbf{O}(N)$	$\mathbf{O}(1)$	$\mathbf{O}(N)$	$\mathbf{O}(N)$

Arrays vs Listas Enlazadas

Operación	Arrays	Listas Enlazadas
Inserción/Borrado	En un array no se pueden borrar celdas intermedias, salvo coste de regenerar el array en su conjunto.	Cada objeto intermedio puede ser fácilmente eliminado o insertado en la lista.
Acceso	Indexado $\mathbf{O}(1)$	Secuencial $\mathbf{O}(N)$
Tipo Objeto	**Listas**: Objetos generales **Array Estática**: Datos homogéneos	Objetos generales encapsulados en clases Nodo
Tamaño	Estático.	Dinámico

Colas (Queues)

Son estructuras lineales optimizadas que gestionan datos principlamente mediante inserción y retirada por los extremos.
Hay tres tipos esenciales[3]:

Pilas (o Stack) en las que la inserción y retirada ocurren por el mismo extremo en secuencia **LIFO Last-In-First-Out**

Colas (Queues) en las que la inserción y retirada ocurren por extremos diferentes en secuencia **FIFO First-In-First-Out**

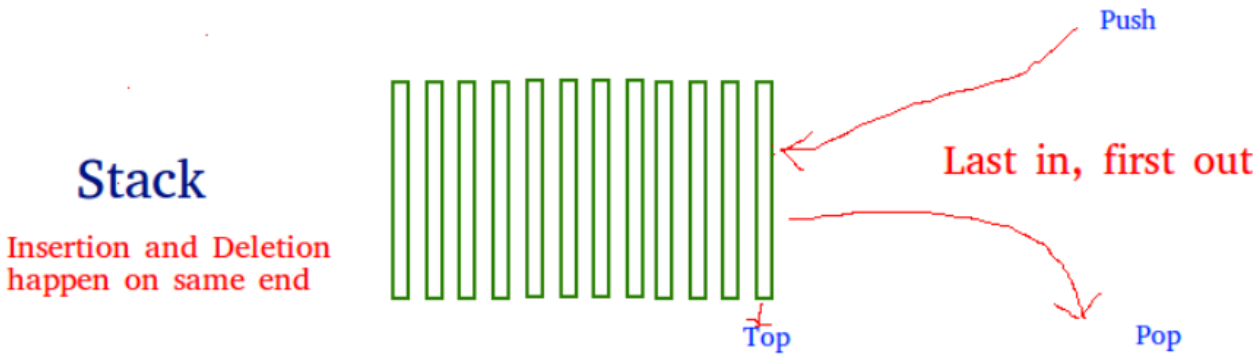
Colas de Prioridad (Priority Queues). Similares a las colas FIFO, excepto que mantiene los elementos ordenados (bajo algún criterio) y el elemento retirado es siempre **el menor** de la cola.

Los métodos básicos de las colas son:

- **Push()** añade el objeto a la cola (Complexity **O**(1))
- **Pop()** retira el objeto de la cola correspondiente al tipo de cola (`FIFO, LIFO, PRIORITY`) (Complexity **O**(1))

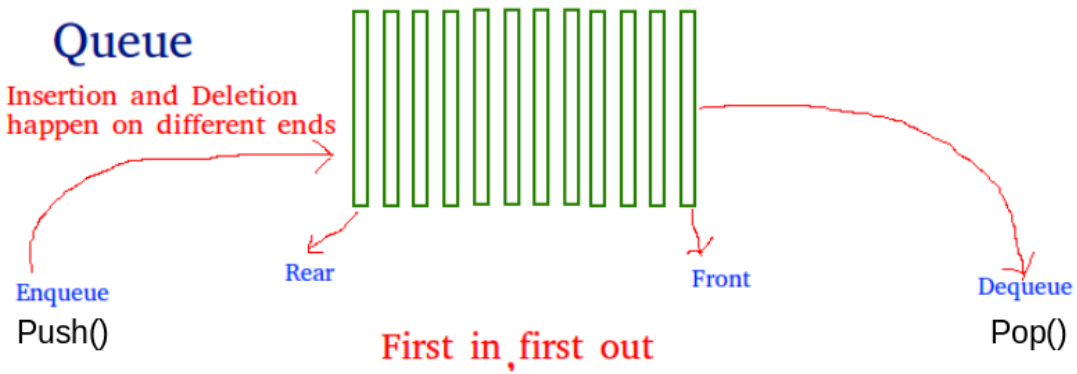
[3] Hay más, pero aquí no entraremos en ellos

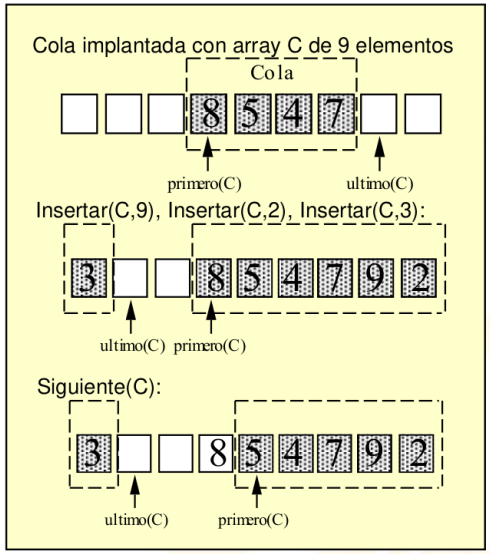
Pilas, o Stack



- Dinámica LIFO.
  Con estricción importante en el acceso a los datos
- Aplicaciones:
  - Almacenamiento de datos intermedios (Pila de la CPU, Notación RPN)
  - Almacenamiento y llamadas recursivas
- Implementación:
  - Con listas enlazadas
  - Con array (conocido el tamaño máximo de la pila)

Colas, o Queues





- Dinámica FIFO.
  Con estricción importante en el acceso a los datos
- Aplicaciones:
  - Gestión de recursos limitados (Alocación de CPU a procesos)
  - Asincronía, buffers de I/O
- Implementación:
  - Con listas enlazadas
  - Con arrays (conocido el tamaño máximo de la pila)

COMILLAS ICAI

Colas en Python

Python nos ofrece una implementación de colas completa

Ver `https://docs.python.org/es/3/library/queue.html`

Alternativamente la clase `collections.deque` (Double ended queue) ofrece una interfaz sencilla

Ver `https://docs.python.org/3/library/collections.html#collections.deque`

$\Rightarrow$ ver Python Help function on `queues.Queue` y `collections.deque`

COMILLAS ICAI

Part I

Matrices de Asociación: Sets y Diccionarios

Almacenamiento de pares Key-Values

Funciones Hash

Sets y Diccionarios

Set **Colección no ordenada de objetos no-mutables, en la que no se admiten duplicados.**

$\Rightarrow$ Las operaciones principales: inserción, borrado y extracción de elementos, arg $\leftarrow$ elemento.

Diccionario **Colección de pares `(Key,Value)` donde las claves son no-mutables y únicas (en un dictionario dado)**

$\Rightarrow$ Las operaciones principales: almacenanamiento del par `(Key , Value)` y la extracción del valor, data la clave.

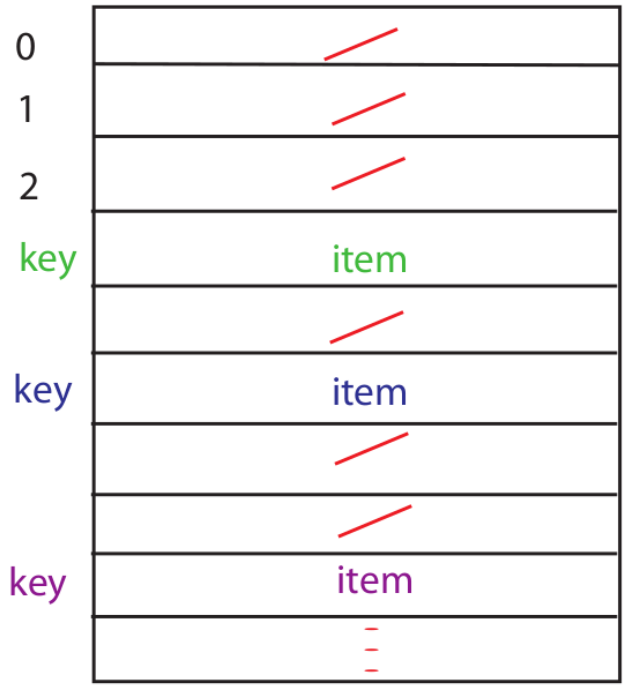
Operación	**Sets**	**Dictionaries**
Creación	{val1, val2,..}	{key1:val1 , key2:val2, ... }
Acceso	N/A	val $\leftarrow$ d[key] val $\leftarrow$ d.get(key)
Insert Remove	add(val) remove (val)	d[key] = value val $\leftarrow$ d.pop(key)
Iterate	iterable	d.keys() d.items()

Sets son diccionarios con Keys exclusivamente (sin Values)

Implementacion de Sets y Diccionarios: Tabla Hash

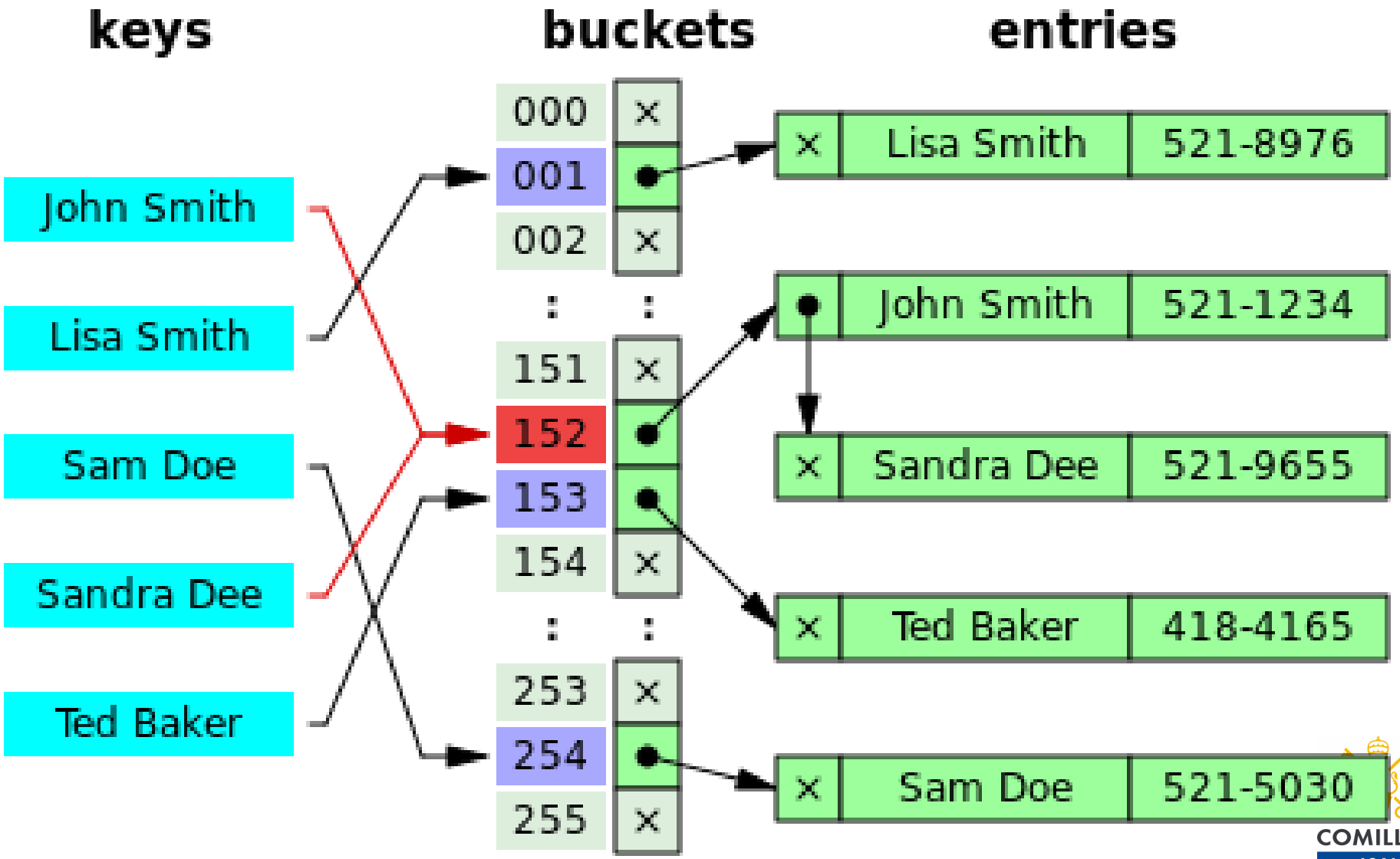
Aunque requerimos la no-mutabilidad de las Keys, implementar los Diccionarios de forma eficiente mediante un array de acceso directo conlleva dos problemas:

1. Las claves deberían ser Enteros positivosn (pero no los son, son cualquier objeto no mutable), para actuar como elementos de direccionamiento
2. Si las claves son elementos no-mutables, pero generales, el espacio es potencialmente infinito!



Resueltos estos problemas, la búsqueda en diccionarios será de $\mathbf{O}(1)$ puesto que la ubicación del Value es un cálculo de dirección $\Rightarrow$ al costo de dejar mucho espacio libre. **Trading Memory vs Performance !!**

tabla Hash


keys
buckets
entries
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker
000
001
002
151
152
153
154
253
254
255
Lisa Smith
521-8976
John Smith
521-1234
Sandra Dee
521-9655
Ted Baker
418-4165
Sam Doe
521-5030

PreHash y Hash: Solución a los problemas de la Tabla Hash

1. **Prehash** Keys (objetos no-mutables) a enteros: $\Rightarrow$ usar Python `hash()` function.
   - Strings, Enteros, y tuplas son objetos no-mutables que son directamente *hashables*
   - Objetos, en general, deberán implementa la función `__hash__()` en base a propiedades no mutables del objeto
   - hash('Francisco') $\rightarrow$ 18908510572443595 22 $\in \mathcal{U}$

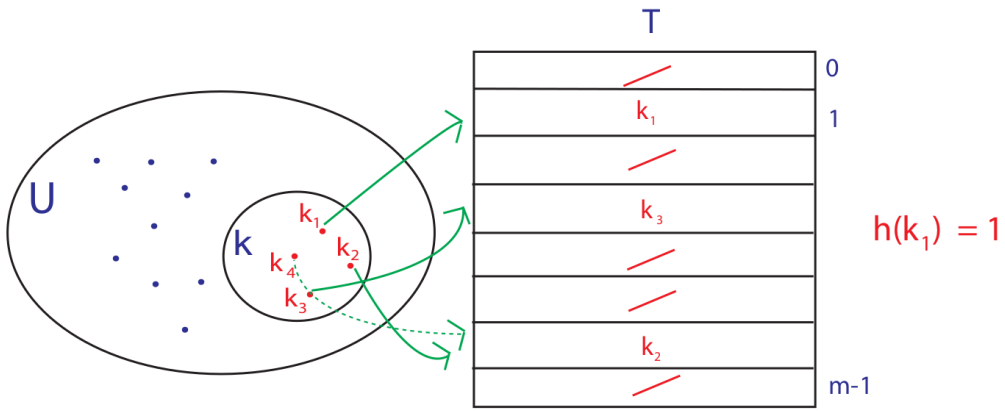
2. **Hashing**. Mapear todas las keys (un número infinito de enteros) a un rango dado de tamaño m. esto es:
   - Reduce el universo $\mathcal{U}$ de todas las claves (ya enteras) a un tamaño razonable m para la tabla hash
   - Hacer $m \approx n$ donde n es el número de claves que preveo almacenar el el diccionario
   - definir una función `hash` $h : \mathcal{U} \rightarrow \{0, 1, ..., m-1\}$

Hashing, Hash Functions

Hay muchas funciones de hashing (ver CRLS). Veremos simplemente una, la función **Universal Hashing** que viene definida como:

$$h(k) = [(a * k + b) \bmod p] \bmod m$$

donde a y $b \in \{0, 1, ...p-1\}$ y p es un número primo $(> |\mathcal{U}|)$



Aloca una tabla de tamaño m con la expectativa de almancenar n elementos.

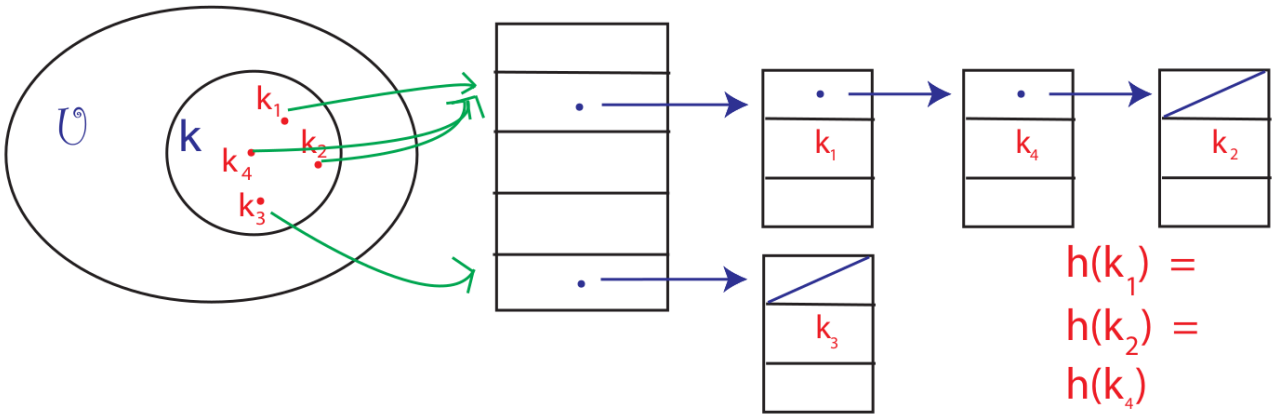
Cómo garantizamos que dos claves no resulten en el mismo hash ??

No podemos $\Rightarrow$ se pueden producir **Colisiones**,

esto es dos claves diferentes recaen en la misma posición de la tabla hash.

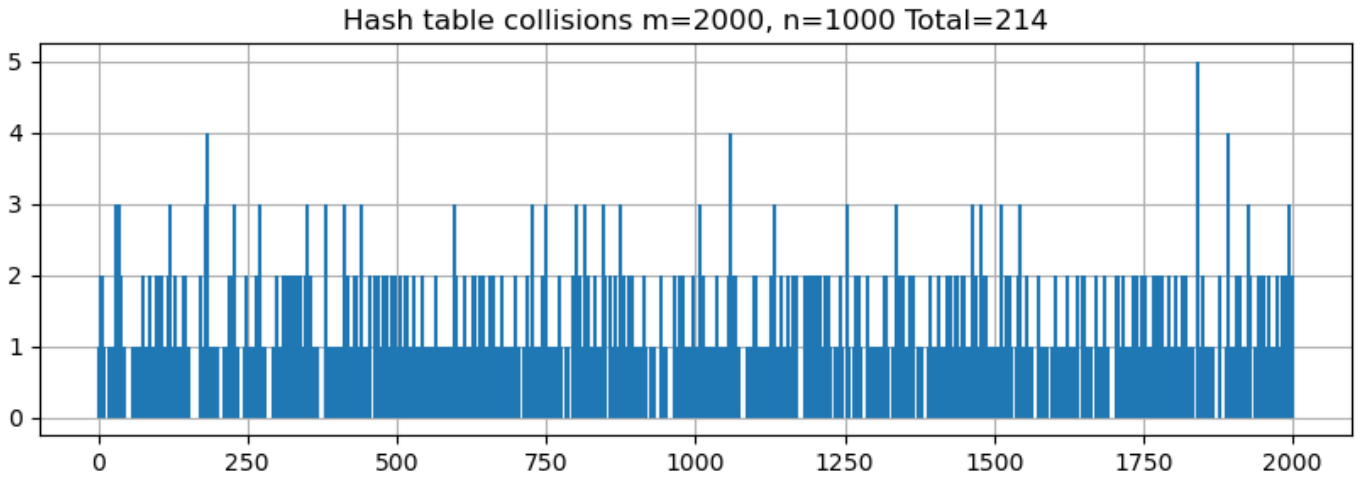
- Se define Factor de Carga $\alpha = n/m$.
- Se puede demostrar que la $E_{a,b}$ [# colisiones con k_1] $= n/m$

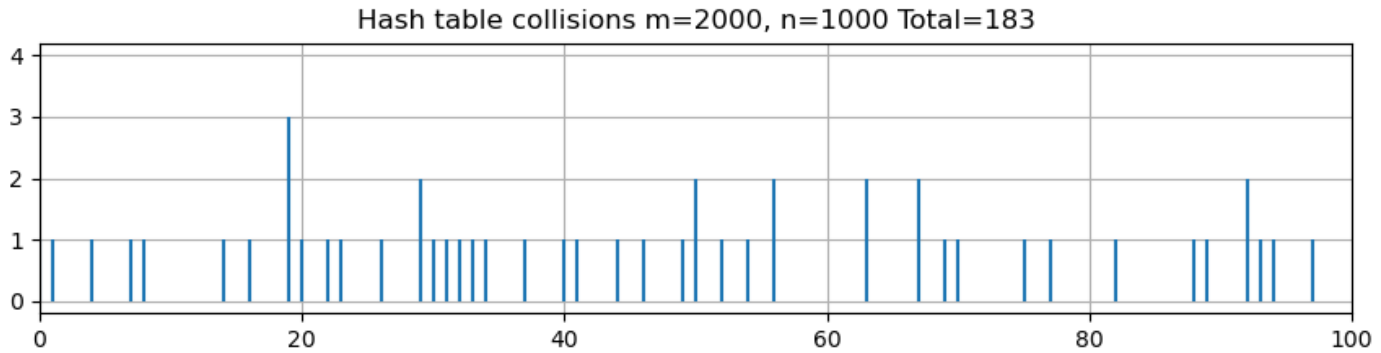
La solución es **Chaining** que substituye las entradas de la tabla con colisiones por listas enlazadas (α es la longitud esperable de la cadena).



En consecuencia, la complejidad de la operación de búsqueda es $\mathbf{\Theta}(1+\alpha)$ que se convierte en $\mathbf{O}(1)$ si $\alpha = \mathbf{O}(1)$ esto es, si $m = \mathbf{\Omega}(n)$

Universal Hashing performance





Usos

Los dicccionarios son una de las estructuras de datos más utilizadas en Computer Science

- **Son flexibles**
  - El concepto `(key,value)` permite un direccionamiento simbólico de la información
  - Acogen todo tipo de valores, mutable, o no ....
  - Permiten anidamiento: diccionario con diccionarios. Ej. Tkinter, xml, json
  - Son iterables, y pueden ser ordenados (normalmente no lo son)
- **son rápidos**
  - Permiten la inserción, y borrado en $\mathbf{O}(1)$ en contra del resto de las estructuras secuenciales
  - Permiten la búsqueda por clave `k` y la extracción del Valor en $\mathbf{O}(1)$

Usos principales de Diccionarios y HashTables

- Agrupaciones Jerárquicas de datos: Ficheros de configuración
- Criptografía. Almancenamiento de passwords.