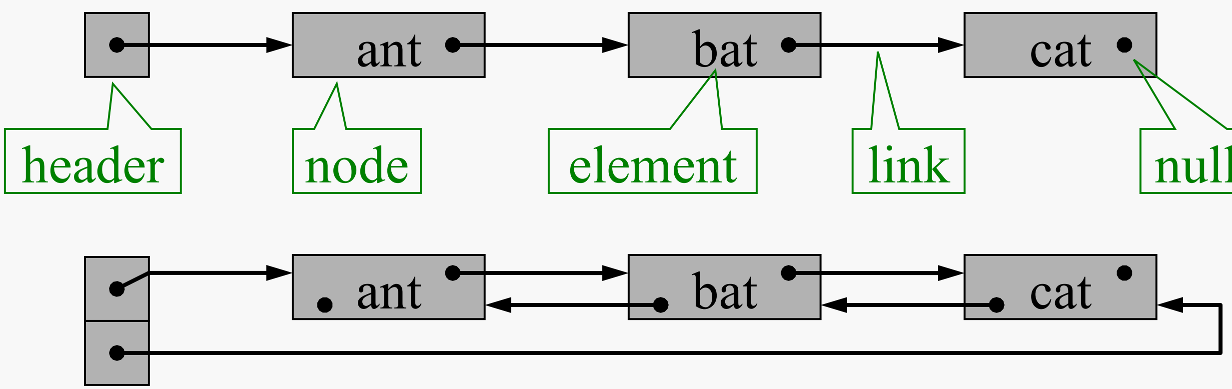
Data structures I – Linked lists, and Hash Tables

LINKED LISTS

Linked lists (1)

- A **linked list** consists of a sequence of **nodes** connected by **links**, plus a **header**.
- Each node (except the last) has a **successor**, and each node (except the first) has a **predecessor**.
- Each node contains a single **element** (object or value), plus links to its successor and/or predecessor.

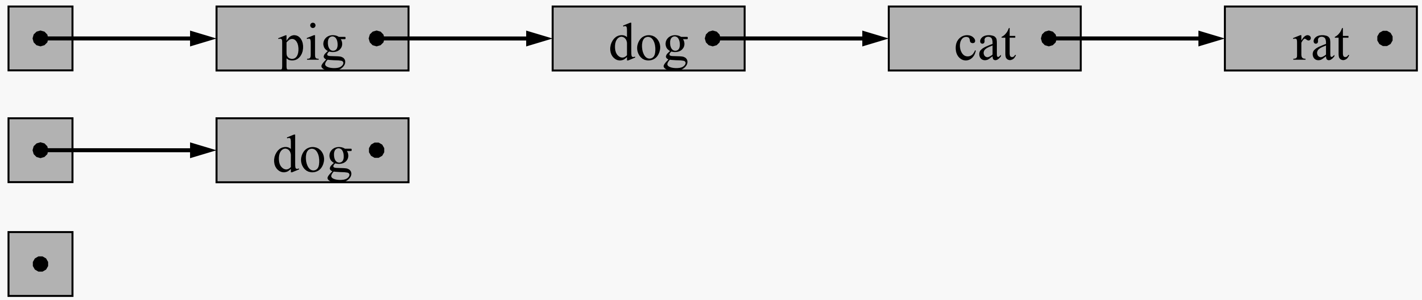


Linked lists (2)

- The **length** of a linked list is the number of nodes.
- An **empty** linked list has no nodes.
- In a linked list:
  - We can manipulate the individual elements.
  - We can manipulate the links, thus changing the linked list's very structure! (This is impossible in an array.)

Singly-linked lists

- A **singly-linked list** (**SLL**) consists of a sequence of nodes, connected by links in one direction only.
- Each SLL node contains a single element, plus a link to the node's successor (or a null link if the node has no successor).
- An SLL header contains a link to the SLL's first node (or a null link if the SLL is empty).



The _SingleLinkedBase Class for a node containing a String (1)

```
class _SingleLinkedBase:
  #A base class providing a singly linked list representation

  #-------------------------- nested _Node class --------------------------
  # nested _Node class
  class _Node:
    #Lightweight, nonpublic class for storing a singly linked node
    __slots__ = '_element', '_next'                 # streamline memory

    def __init__(self, element, next):              # initialize node's fields
      self._element = element                       # user's element
      self._next = next                             # next node reference
```

The _SingleLinkedBase Class for a node containing a String (2)

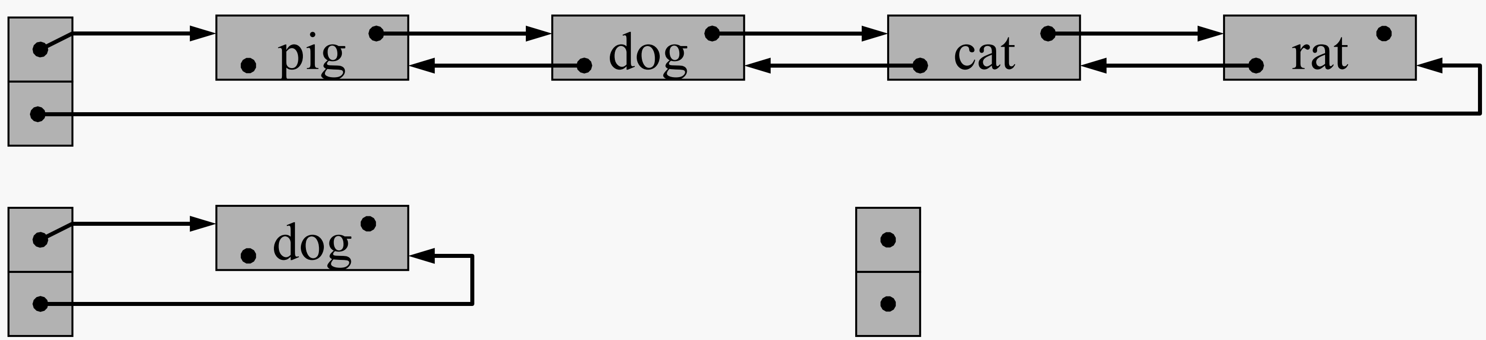
```
#--------------------- list constructor --------------------------
  def __init__(self):
    #Create an empty list
    self._head = self._Node(None,  None)
    self._size = 1                           # number of elements

  #-------------------- public accessors --------------------------
  def __len__(self):
    #Return the number of elements in the list
    return self._size

  def is_empty(self):
    #Return True if list is empty
    return self._size == 0
```

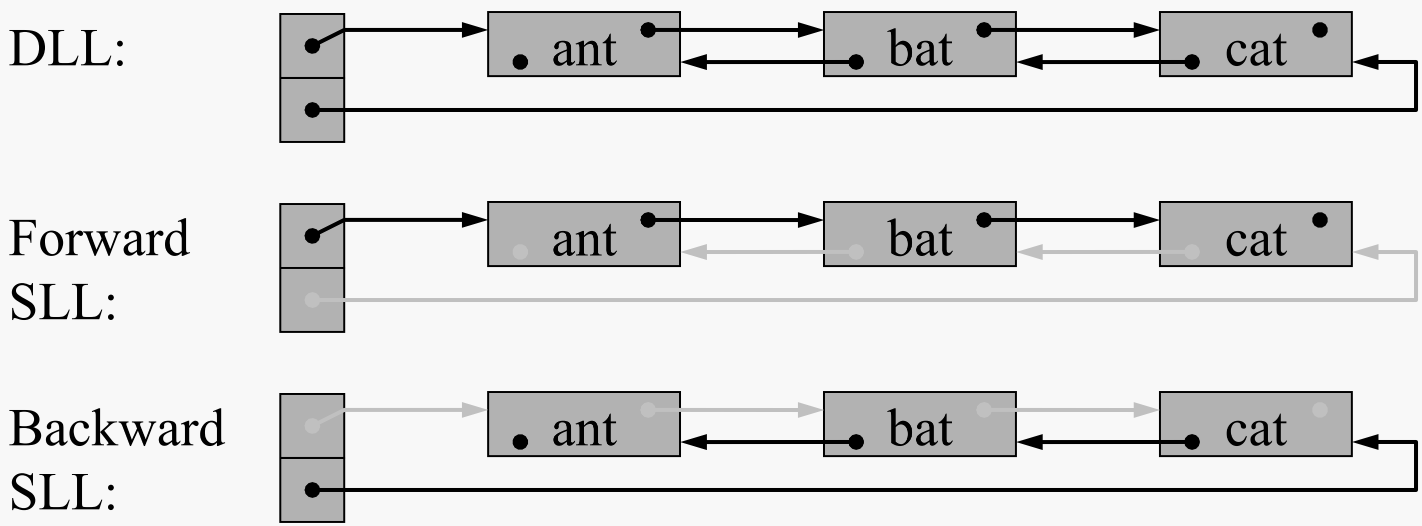
Doubly-linked lists

- A **doubly-linked list (DLL)** consists of a sequence of nodes, connected by links in both directions.
- Each DLL node contains a single element, plus links to the node's successor and predecessor (or null link(s)).
- The DLL header contains links to the DLL's first and last nodes (or null links if the DLL is empty).



DLL = forward SLL + backward SLL

- View a DLL as a **backward SLL** superimposed on a **forward SLL**:



Insertion

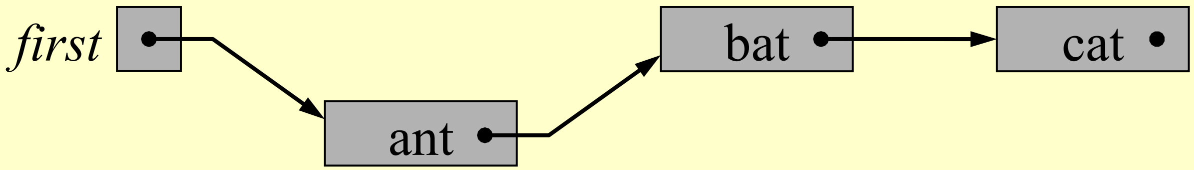
- Problem: Insert a new element ***at a given point*** in a linked list.
- Four cases to consider:
  1) insertion in an empty linked list;
  2) insertion before the first node of a nonempty linked list;
  3) insertion after the last node of a nonempty linked list;
  4) insertion between nodes of a nonempty linked list.
- The insertion algorithm needs links to the new node's successor and predecessor.

SLL insertion (1)

- Animation (insertion before first node):

To insert *elem* at a given point in the SLL headed by *first*:

1. Make *ins* a link to a newly-created node with element *elem* and successor null.
2. If the insertion point is before the first node:
   2.1. Set node *ins*'s successor to *first*.
   2.2. Set *first* to *ins*.
3. If the insertion point is after the node *pred*:
   3.1. Set node *ins*'s successor to node *pred*'s successor.
   3.2. Set node *pred*'s successor to *ins*.
4. Terminate.

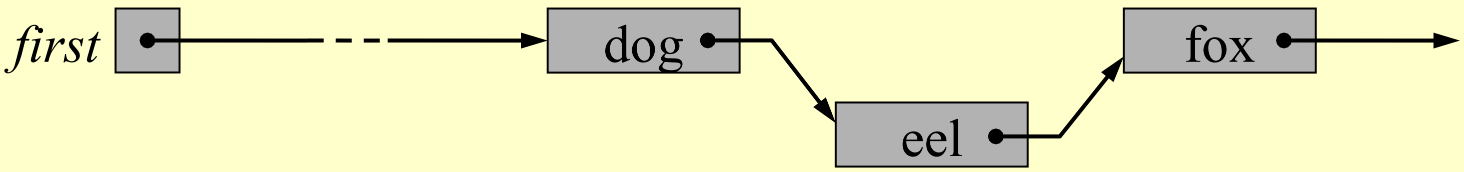


SLL insertion (2)

- Animation (insertion after intermediate node):

To insert *elem* at a given point in the SLL headed by *first*:

1. Make *ins* a link to a newly-created node with element *elem* and successor null.
2. If the insertion point is before the first node:
   - 2.1. Set node *ins*'s successor to *first*.
   - 2.2. Set *first* to *ins*.
3. If the insertion point is after the node *pred*:
   - 3.1. Set node *ins*'s successor to node *pred*'s successor.
   - 3.2. Set node *pred*'s successor to *ins*.
4. Terminate.



DLL insertion (1)

- **DLL insertion algorithm**:

  To insert *elem* ***at a given point*** in the DLL headed by (*first*, *last*):

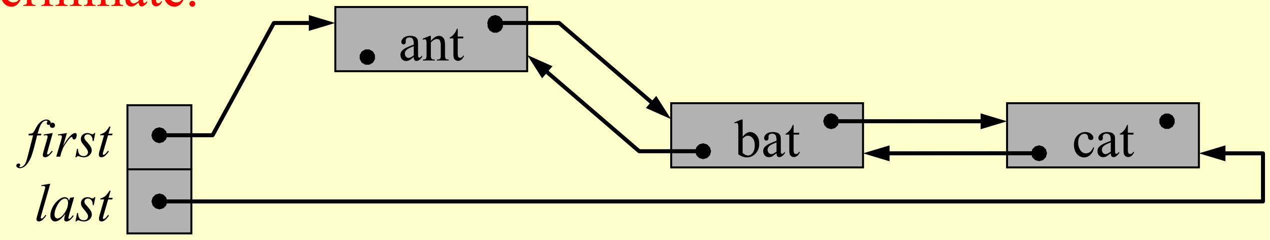
  1. Make *ins* a link to a newly-created node with element *elem*, predecessor null, and successor null.
  2. Insert *ins* at the insertion point in the forward SLL headed by *first*.
  3. Let *succ* be *ins*'s successor (or null if *ins* has no successor).
  4. Insert *ins* after node *succ* in the backward SLL headed by *last*.
  5. Terminate.

DLL insertion (2)

- Animation (insertion before the first node):

To insert *elem* at a given point in the DLL headed by (*first*, *last*):

1. Make *ins* a link to a newly-created node with element *elem*, predecessor null, and successor null.
2. Insert *ins* at the insertion point in the forward SLL headed by *first*.
3. Let *succ* be *ins*'s successor (or null if *ins* has no successor).
4. Insert *ins* after node *succ* in the backward SLL headed by *last*.
5. Terminate.

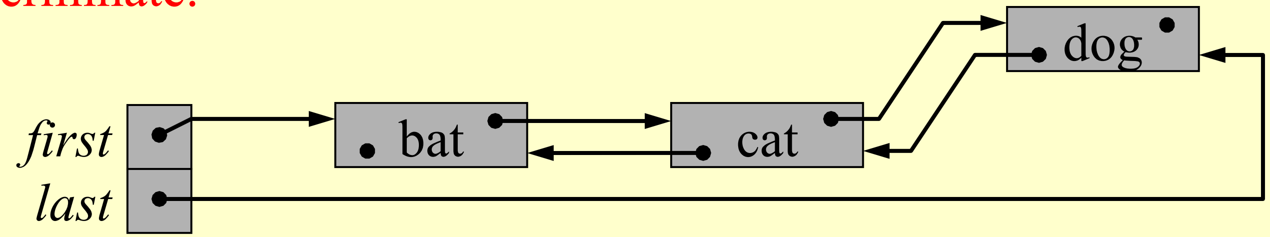


DLL insertion (3)

- Animation (insertion after the last node):

To insert *elem* at a given point in the DLL headed by (*first*, *last*):

1. Make *ins* a link to a newly-created node with element *elem*, predecessor null, and successor null.
2. Insert *ins* at the insertion point in the forward SLL headed by *first*.
3. Let *succ* be *ins*'s successor (or null if *ins* has no successor).
4. Insert *ins* after node *succ* in the backward SLL headed by *last*.
5. Terminate.

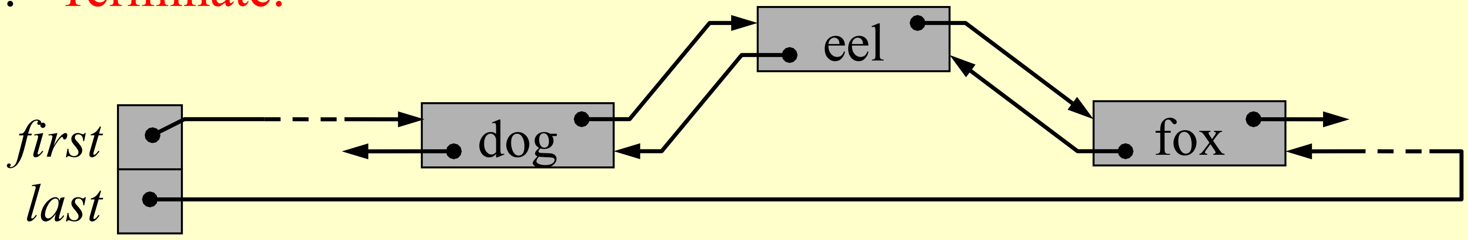


DLL insertion (4)

- Animation (insertion between nodes):

To insert *elem* at a given point in the DLL headed by (*first*, *last*):

1. Make *ins* a link to a newly-created node with element *elem*, predecessor null, and successor null.
2. Insert *ins* at the insertion point in the forward SLL headed by *first*.
3. Let *succ* be *ins*'s successor (or null if *ins* has no successor).
4. Insert *ins* after node *succ* in the backward SLL headed by *last*.
5. Terminate.



Deletion

- Problem: Delete ***a given node*** from a linked list.
- Four cases to consider:

  1) deletion of a singleton node;

  2) deletion of the first (but not last) node;

  3) deletion of the last (but not first) node;

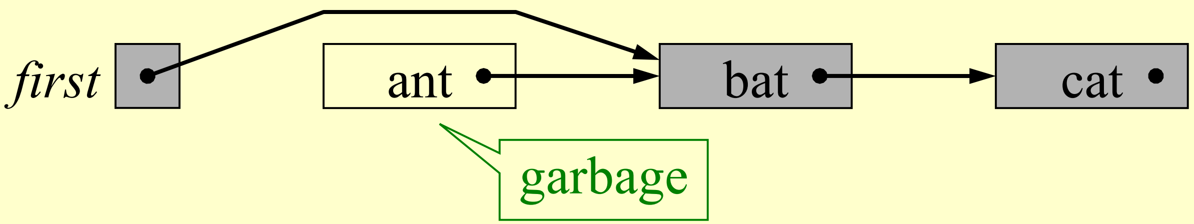
  4) deletion of an intermediate node.
- The deletion algorithm needs links to the deleted node's successor and predecessor.

SLL deletion (1)

- Animation (deleting the first node):

To delete node *del* from the SLL headed by *first*:

1. Let *succ* be node *del*'s successor.
2. If *del* = *first*:
   2.1. Set *first* to *succ*.
3. Otherwise (if *del* ≠ *first*):
   3.1. Let *pred* be node *del*'s predecessor.
   3.2. Set node *pred*'s successor to *succ*.
4. Terminate.

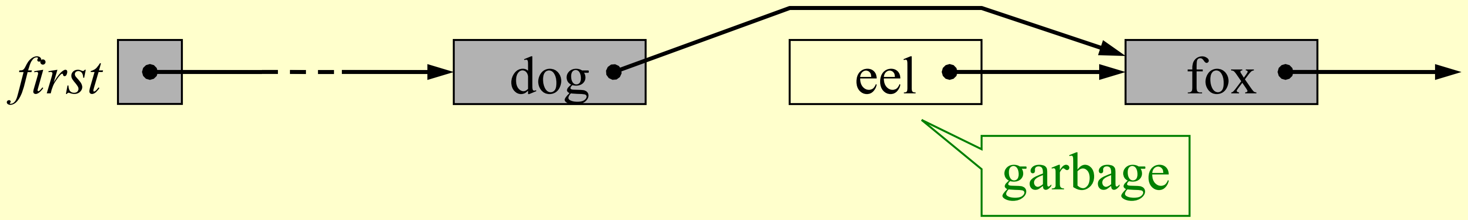


SLL deletion (2)

- Animation (deleting an intermediate (or last) node):

To delete node *del* from the SLL headed by *first*:

1. Let *succ* be node *del*'s successor.
2. If *del* = *first*:
   2.1. Set *first* to *succ*.
3. Otherwise (if *del* ≠ *first*):
   3.1. Let *pred* be node *del*'s predecessor.
   3.2. Set node *pred*'s successor to *succ*.
4. Terminate.



SLL deletion (3)

- Analysis:

  Let n be the SLL's length.

  Step 3.1 must visit all nodes from the first node to the deleted node's predecessor. There are between 0 and n–1 such nodes.

  At most, no. of nodes visited $= n - 1$

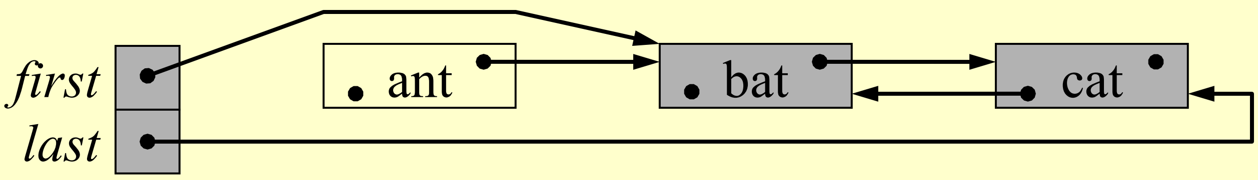
  Time complexity is $O(n)$.

DLL deletion (2)

- Animation (deleting the first (but not last) node):

To delete node *del* from the DLL headed by (*first*, *last*):

1. Let *pred* and *succ* be node *del*'s predecessor and successor.
2. Delete node *del*, whose predecessor is *pred*, from the forward SLL headed by *first*.
3. Delete node *del*, whose successor is *succ*, from the backward SLL headed by *last*.
4. Terminate.

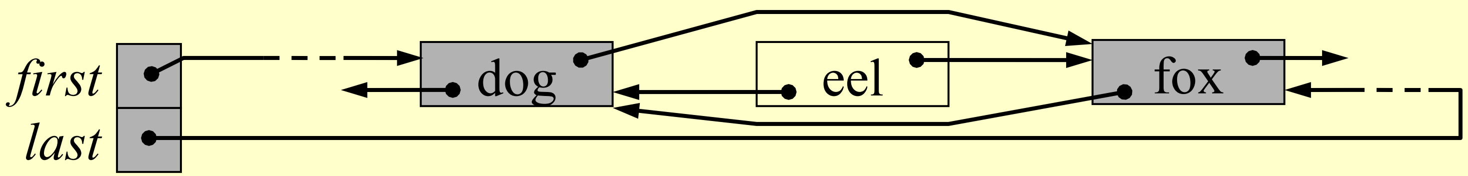


DLL deletion (3)

- Animation (deleting an intermediate node):

To delete node *del* from the DLL headed by (*first*, *last*):

1. Let *pred* and *succ* be node *del*'s predecessor and successor.
2. Delete node *del*, whose predecessor is *pred*, from the forward SLL headed by *first*.
3. Delete node *del*, whose successor is *succ*, from the backward SLL headed by *last*.
4. Terminate.



Comparison of insertion and deletion algorithms

Algorithm	SLL	DLL
Insertion	$O(1)$	$O(1)$
Deletion	$O(n)$	$O(1)$

Searching (1)

- Problem: Search for a given target value in a linked list.
- **Unsorted SLL linear search algorithm**:

  To find which (if any) node of the SLL headed by *first* contains an element equal to *target*:

  1. For each node *curr* in the SLL headed by *first*, repeat:
     1.1. If *target* is equal to node *curr*'s element, terminate with answer *curr*.
  2. Terminate with answer *none*.

- DLL linear search is similar, except that we can search from last to first if preferred.

Searching (2)

- Analysis (counting comparisons):

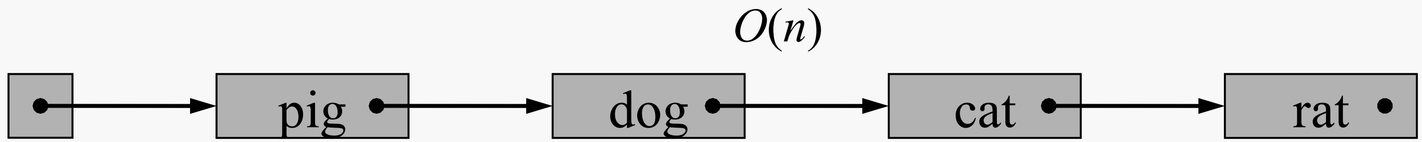
  Let n be the SLL's length.
- If the search is **successful**:

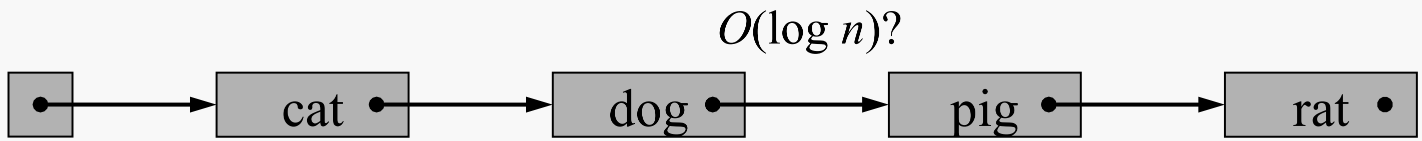
  At most, no. of comparisons $= n$
- If the search is **unsuccessful**:

  No. of comparisons $= n$
- In either case, time complexity is $O(n)$.

Searching (3)

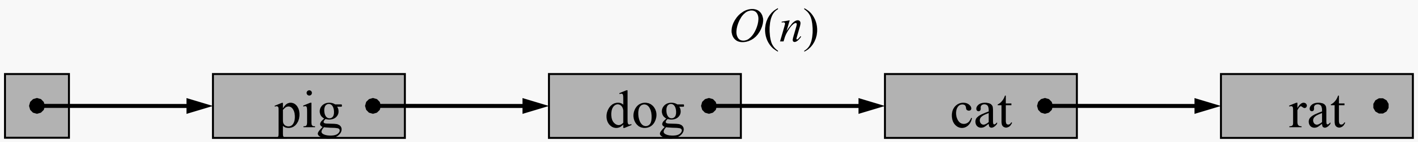
- **(Binary) search in a *sorted* SLL**
  - $O(\log n)$?
  - Locating the middle node, $O(n)$

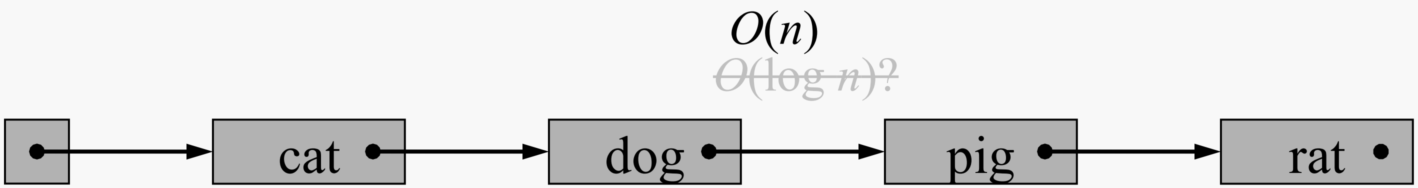




Searching (4)

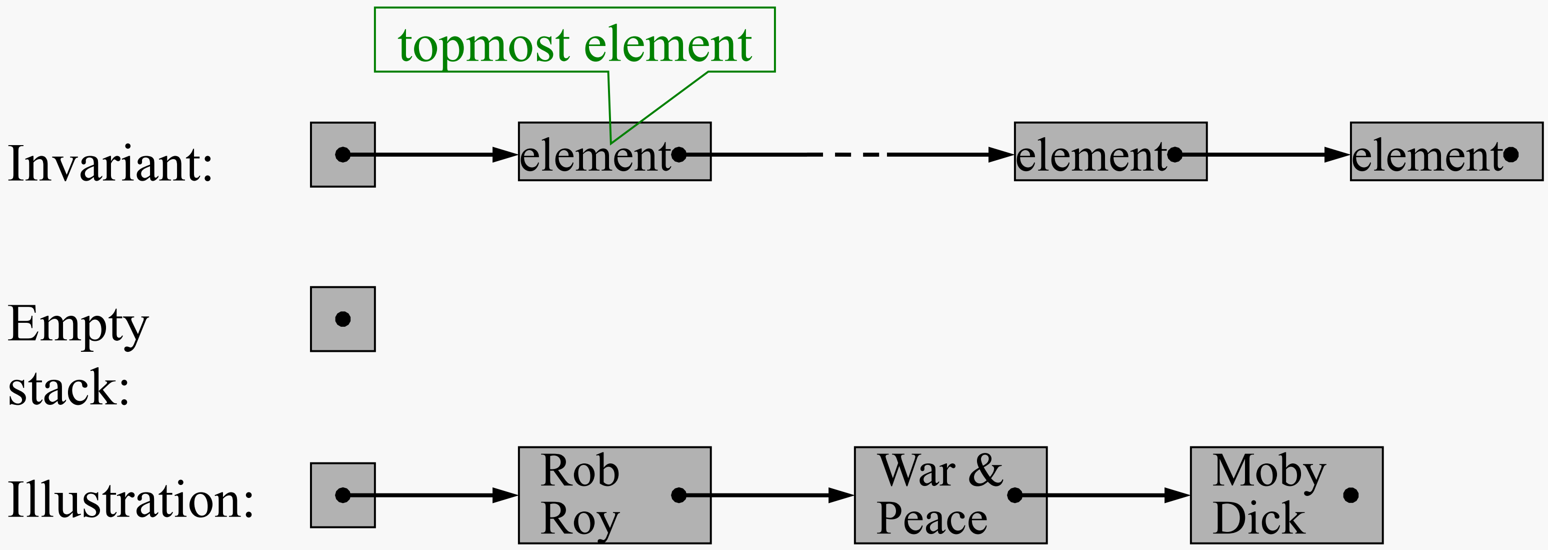
- (~~Binary~~) **Search in a *sorted* SLL**
  - $\cancel{O(\log n)}$~~?~~
  - ~~Locating the middle node,~~ $\cancel{O(n)}$
  - Linear search, $O(n)$





Implementation of stacks using SLLs (1)

- Represent an (unbounded) stack by an SLL, such that the first node contains the topmost element.

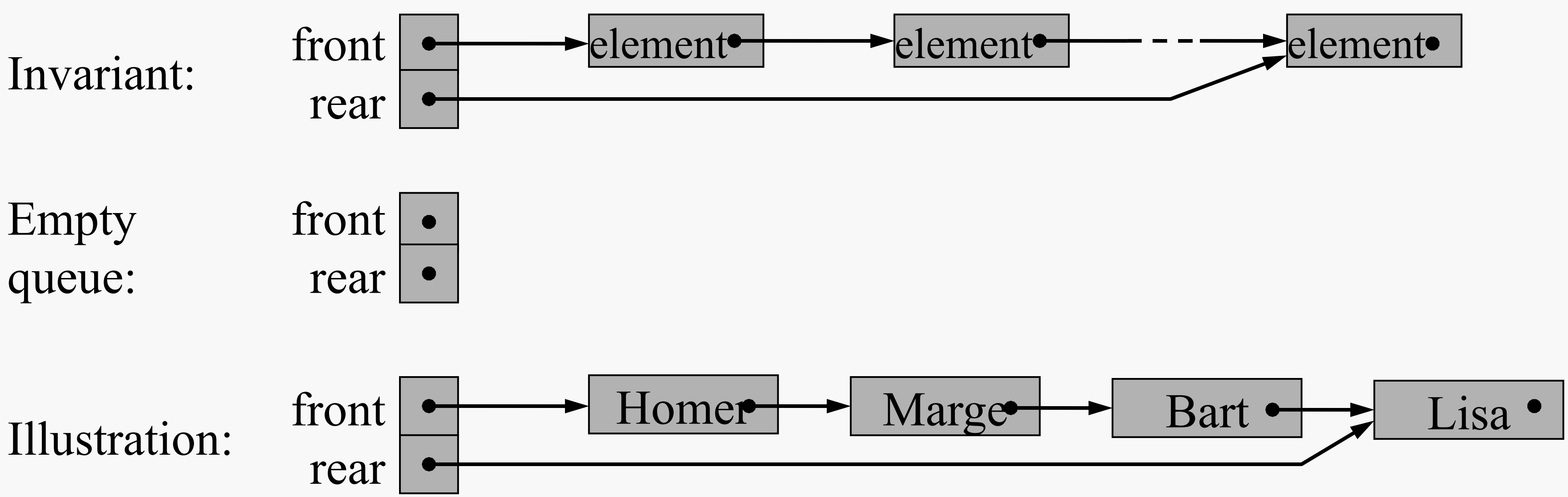


Implementation of stacks using SLLs (2)

- Analysis:
  - All operations have time complexity $O(1)$.

Implementation of queues using SLLs (1)

- Represent an (unbounded) queue by:
  - an SLL, whose first node contains the front element, and whose header contains links to the first node (*front*) and last node (*rear*).
  - a variable *length* (optional).



Implementation of queues using SLLs (2)

- Analysis:
  - Most operations have time complexity $O(1)$, but `size` is $O(n)$.
  - However, `size` too would be $O(1)$ if we used a variable *length*.

HASH TABLES

Maps

- A **map** models a searchable collection of **key-value** entries
- The main operations of a map are for searching, inserting, and deleting items (seen with linked lists)
- Multiple entries with the same key are **not** allowed
- Applications:
  - address book
  - student-record database
- Python's **dict** class is a very important data structure in the language - representing an abstraction known as a dictionary in which unique keys are mapped to associated values
- A map is a more general form of the dict abstract data type

The Map ADT

- Some map ADT methods:
  - `M[k]=v`: associate value `v` with key `k` in map `M`, replacing the existing value if the map already contains an entry with key `k`

    e.g. `M['Alison'] = 02085551234`
  - `M[k]`: if the map `M` has an entry with key `k`, return its associated value `v`; else, raise a `KeyError`

    e.g. `M['Alison']` returns `02085551234`
  - `Del M[k]`: remove from map `M` the item with key `k`; if key `k` is not already in `M`, then raise a `KeyError`
  - `k in M`: if the map `M` has an entry with key `k` then return `True`
  - `len(M)`: return the number of items in the map `M`.
  - `M.pop(k,d=None)`: remove item with key `k` and return its value `v`. If `k` is not in map return `d` or raise `KeyError` if `d` set to `None`
  - `M.keys()`: return set-like view of all keys of `M`

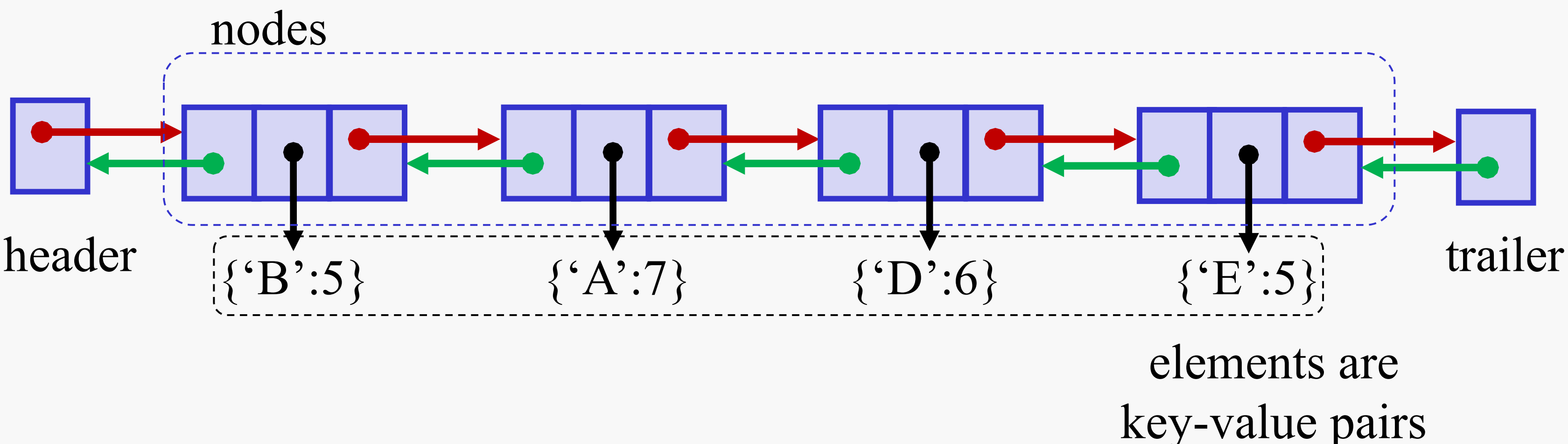
  …Lots of others

Example use of a Map

Operation	*Output*	*Map*
`Len(M)`	`0`	`Ø`
`M[‘A’]=5`	`-`	`{’A’:5}`
`M[‘B’]=7`	`-`	`{'A':5,'B':7}`
`M[‘C’]=2`	`-`	`{'A':5,'B':7,'C’:2}`
`M[‘D’]=8`	`-`	`{'A':5,'B':7,'C':2,'D’:8}`
`M[‘C’]=9`	`-`	`{'A':5,'B':7,’C’:9,'D’:8}`
`M[‘B’]`	`7`	`{'A':5,'B':7,’C’:9,'D’:8}`
`M[‘X’]`	`KeyError`	`{'A':5,'B':7,’C’:9,'D’:8}`
`len(M)`	`4`	`{'A':5,'B':7,’C’:9,'D’:8}`
`Del M[‘A’]`	`-`	`{‘B':7,’C’:9,'D’:8}`
`M.pop(‘B’)`	`7`	`{‘C’:9,'D’:8}`
`M.keys()`	`‘C’,’D’`	`{‘C’:9,'D’:8}`

A Simple List-based Map

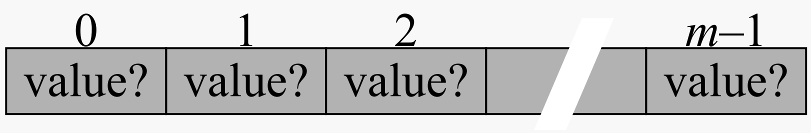
- We can (inefficiently) implement a map `M` using an unsorted list
- We store the items of the map in a list (based on a doubly-linked list), in arbitrary order – not good for large maps
- Inserting an item takes $O(1)$ if we insert at beginning/end of list
- Searching for an item can take $O($`len[M]`$)$



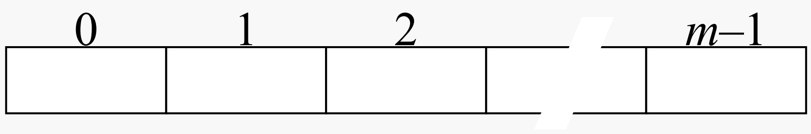
Implementation of small-integer-key maps using key-indexed arrays (1)

- What about a more efficient storage and retrieval of information than a list-based map?
- If the keys are known to be small **integers**, in the range 0…m–1, represent the map by:
  - an array *vals* of length m, such that *vals*[k] contains a value v if and only if (k, v) is a entry of the map.

Invariant:



Empty map:



Implementation using key-indexed arrays *(2)*

Illustration (m = 20):

code	module
01	CS1
02	CS2
10	DB
11	OOP
12	ADS
14	OS
16	HCI

is represented by

0	1	2	3		10	11	12	13	14	15	16	17	18	19
	CS1	CS2			DB	OOP	ADS		OS		HCI			

Implementation using key-indexed arrays *(3)*

Operation	Algorithm	Time complexity
`search`	inspect array component	$O(1)$
`insert`	update array component	$O(1)$
`delete`	make array component null	$O(1)$

Implementation using key-indexed arrays *(3)*

Illustration ($m = 20$):

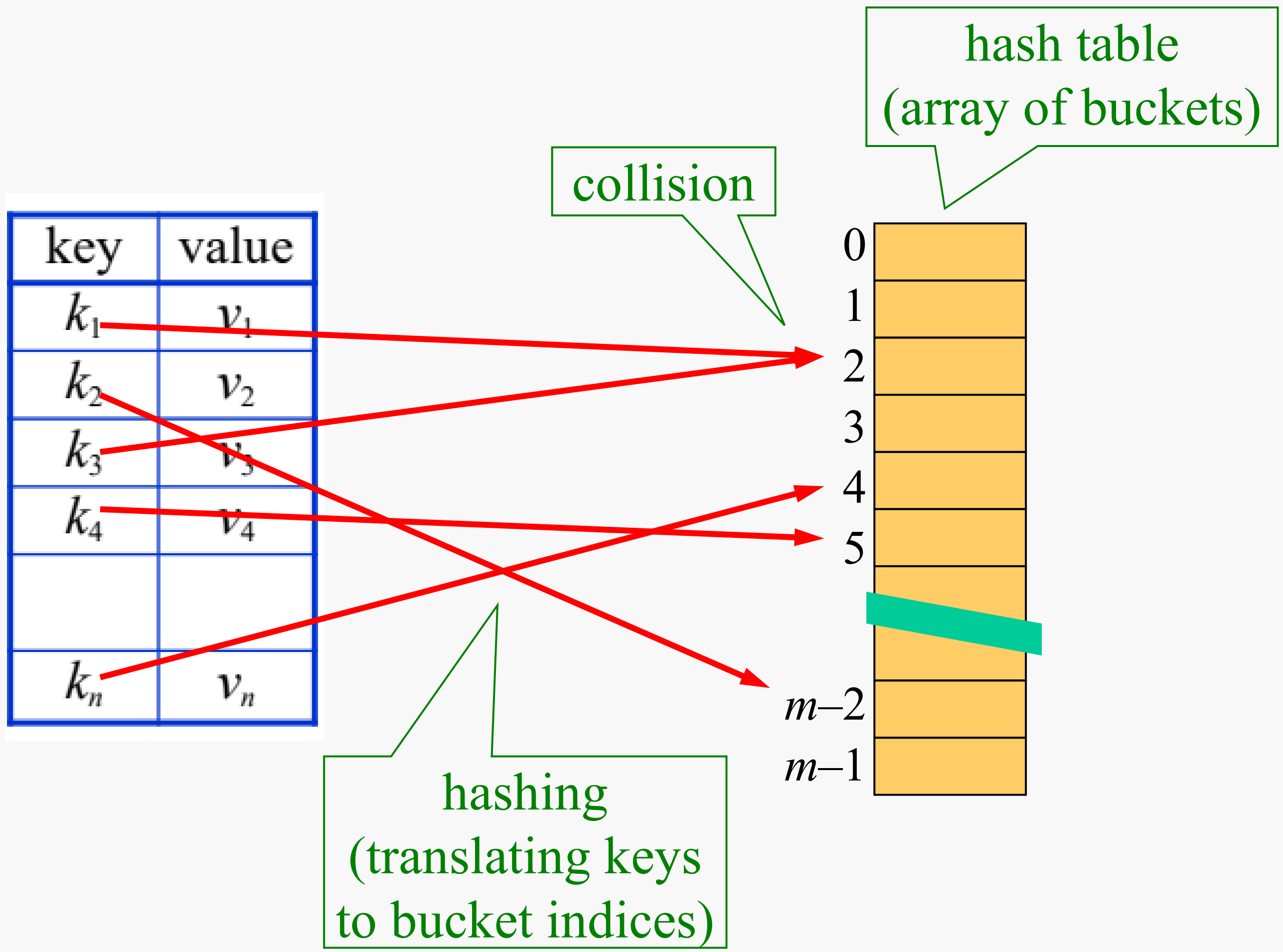
module	code
CS1	01
CS2	02
DB	10
OOP	11
ADS	12
OS	14
HCI	16

?

Hash Tables Principles (1)

- If a map's keys are small integers, we can represent the map by a key-indexed array. Search, insertion, and deletion then have time complexity $O(1)$.
- Surprisingly, we can approach this performance with keys of other types!
- **Hashing**: translate each key to a small integer, and use that integer to index an array.
- A **hash table** is an array of m **buckets**, together with a **hash function** $hash(k)$ that translates each key k to a bucket index (in the range $0 \ldots m-1$).

Hash Tables Principles (2)



Hash Tables Principles (3)

- Each key k has a home bucket in the hash table, namely the bucket with index $hash(k)$.

- To **insert** a new entry with key k into the hash table, assign that entry to k's home bucket.

- To **search** for an entry with key k in the hash table, look in k's home bucket.

- To **delete** an entry with key k from the hash table, look in k's home bucket.

Hash Tables Principles (4)

- The hash function must be **consistent**:

  $k_1 = k_2$ implies $hash(k_1) = hash(k_2)$.

- In general, the hash function is many-to-one.
- Therefore, different keys may share the same home bucket:

  $k_1 \neq k_2$ but $hash(k_1) = hash(k_2)$.

  This is called a **collision**.

- Always prefer a hash function that makes collisions relatively infrequent.

Example: a Hash Function for Words

- Suppose that the keys are English words.
- Possible hash function:
- $$m = 26$$
  $$hash(w) = (\text{initial letter of } w) - \text{‘A’}$$
- All words with initial letter ‘A’ share bucket 0;
  …
  all words with initial letter ‘Z’ share bucket 25.
- This is a convenient choice for illustrative purposes.
- But it is a poor choice for practical purposes: collisions are likely to be frequent in some buckets.

Closed-bucket vs. Open-bucket Hash Tables

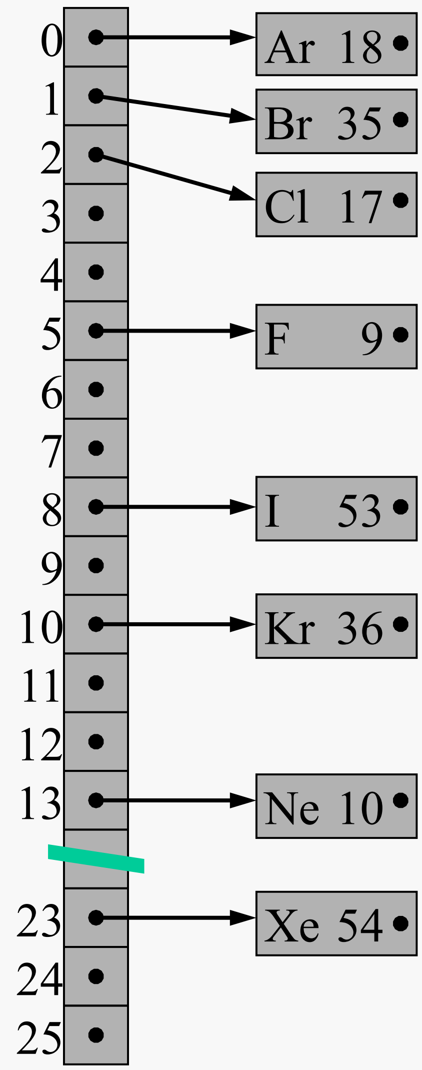
- **Closed-bucket hash table (CBHT)**:
  - Each bucket may be occupied by several entries.
  - Buckets are completely separate: *separate chaining*.

- **Open-bucket hash table (OBHT)**:
  - Each bucket may be occupied by at most one entry.
  - Whenever there is a collision, displace the new entry to another bucket: *linear probing*.

Closed-bucket Hash Tables

Element	Number
F	9
Ne	10
Cl	17
Ar	18
Br	35
Kr	36
I	53
Xe	54

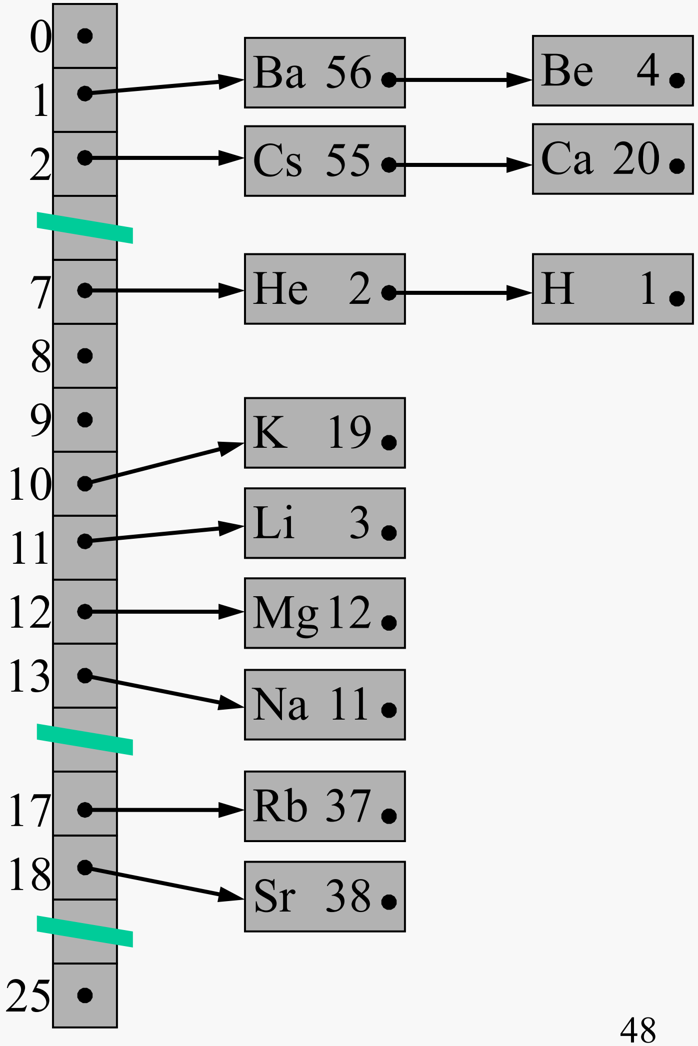
is represented by



CBHT: Collisions

Element	Number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56

With collisions



CBHT: Algorithms & Analysis

- Set bucket *b* to *hash(target-key)* and search/insertion/deletion algorithms of SLL
- Analysis of the CBHT search/insertion/deletion algorithms (counting comparisons): Let the number of entries be *n*.
- Set bucket *b* to *hash(target-key)* and search/insertion/deletion algorithms of SLL.
- In the best case, no bucket contains more than (say) 2 entries: **Best-case** time complexity is O(1).
- In the worst case, one bucket contains all n entries: **Worst-case** time complexity is O(*n*).

CBHT: Design

- CBHT design consists of:
  - choosing the number of buckets m
  - choosing the hash function *hash*.

- Design aims:
  - collisions should be infrequent
  - entries should be distributed evenly among the buckets, such that few buckets contain more than about 2 entries.

CBHT: Number of Buckets

- The load factor of a hash table is the average number of entries per bucket, n/m.

- If n is (roughly) predictable, choose m such that the load factor is likely to be between 0.5 and 0.75.
  - A low load factor wastes space.
  - A high load factor tends to cause some buckets to have many entries.

- Choose m to be a prime number.

CBHT: Hash Function

- The hash function should be efficient (performing few arithmetic operations).

- The hash function should distribute the entries evenly among the buckets, regardless of any patterns in the keys.

- Possible trade-off:
  - Speed up the hash function by using only part of the key.
  - But beware of any patterns in that part of the key.

Example: Hash Table for Words (1)

- $hash(w)$ can depend on any of w's letters and/or length.

- Consider $m = 20$, $hash(w)$ = length of $w - 1$.
  - Far too few buckets. Load factor = 1000/20 = 50.
  - Very uneven distribution.

- Consider $m = 26$, $hash(w)$ = initial letter of w – 'A'.
  - Far too few buckets.
  - Very uneven distribution.

Example: Hash Table for Words (2)

- Consider $m = 520$, $hash(w) = 26 \times$ (length of $w - 1$) + (initial letter of w – 'A').
  - Too few buckets. Load factor = 1000/520 $\approx$ 1.9.
  - Very uneven distribution. Since few words have length 0–2, buckets 0–51 will be sparsely populated. Since initial letter Z is uncommon, buckets 25, 51, 77, 103, … will be sparsely populated. And so on.
- Consider $m = 1499$, $hash(w)$ = (weighted sum of letters of w) modulo m

  i.e., ($c_1 \times$ 1st letter of w + $c_2 \times$ 2nd letter of w + …) modulo m

  + Good number of buckets. Load factor $\approx$ 0.67.
  + Reasonably even distribution.

Open-bucket Hash Table (1)

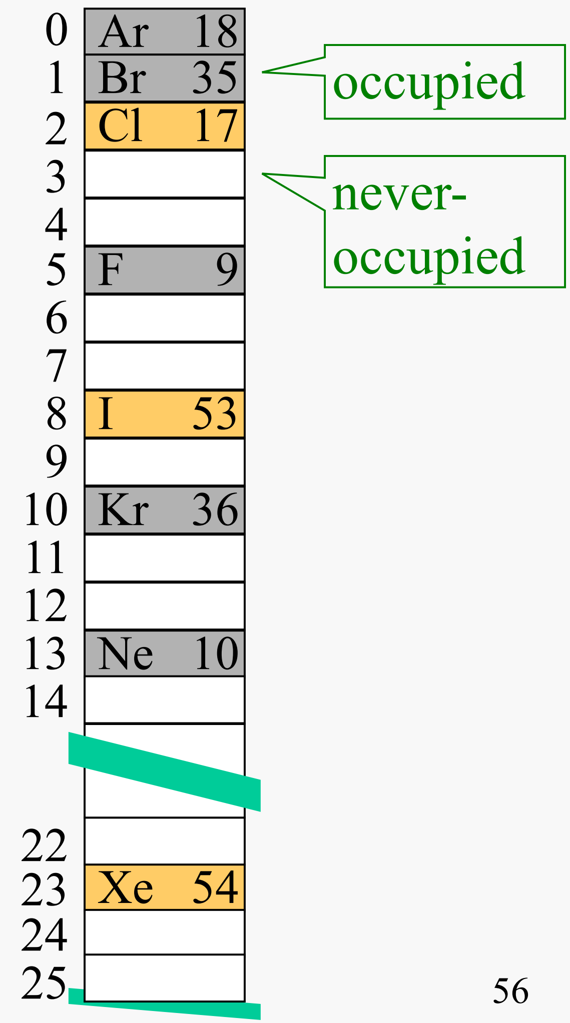
- **Open-bucket hash table (OBHT)**:
  - Each bucket may be occupied by at most one entry.
  - Whenever there is a collision, displace the new entry to another bucket: *linear probing*.

- Each bucket has three possible states:
  - **occupied** (currently contains an entry)
  - **never-occupied** (has never contained an entry)
  - **formerly-occupied** (previously contained an entry, which has been deleted and not yet replaced).

Open-bucket Hash Table (2)

Element	Number
F	9
Ne	10
Cl	17
Ar	18
Br	35
Kr	36
I	53
Xe	54

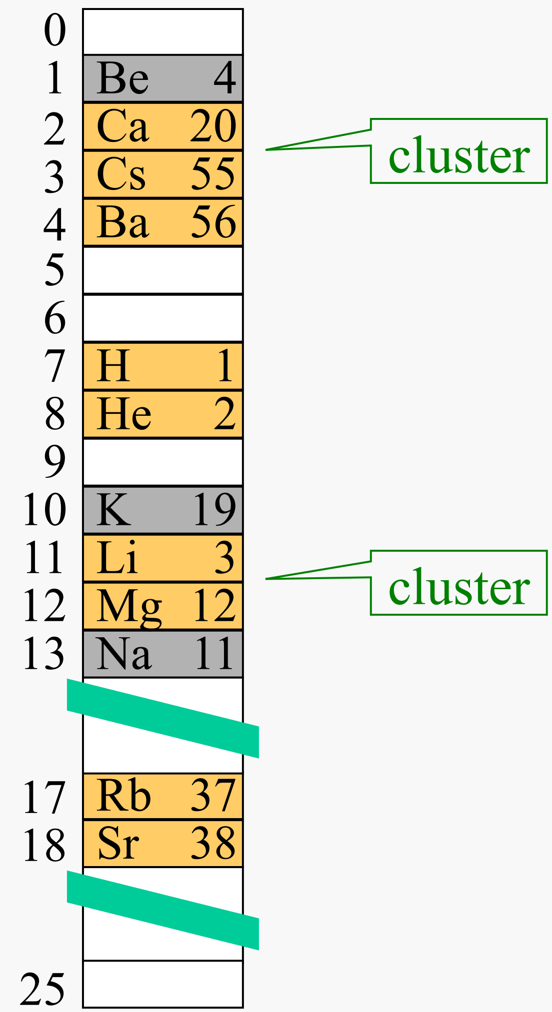
With no collisions



OBHT: Collisions (1)

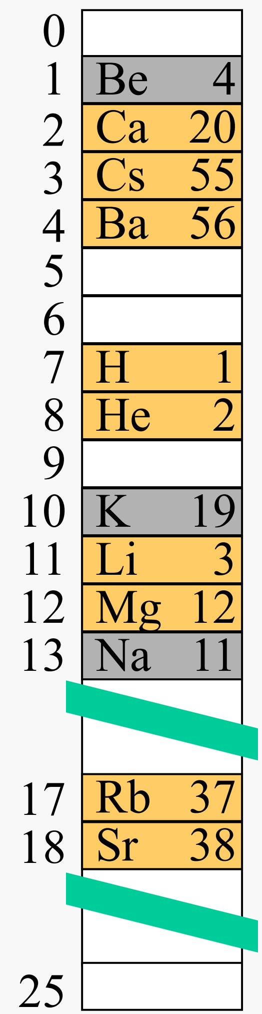
Element	Number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56

With collisions



OBHT: Collisions (2)

Element	Number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



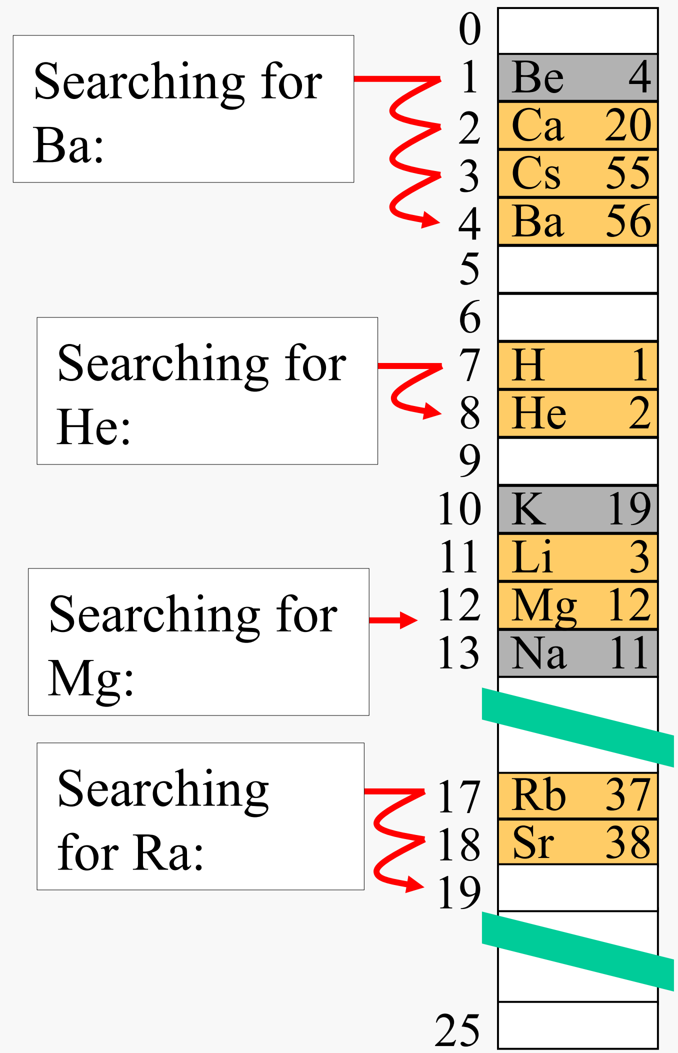
OBHT: Search

1. Set *b* to *hash*(*target-key*).

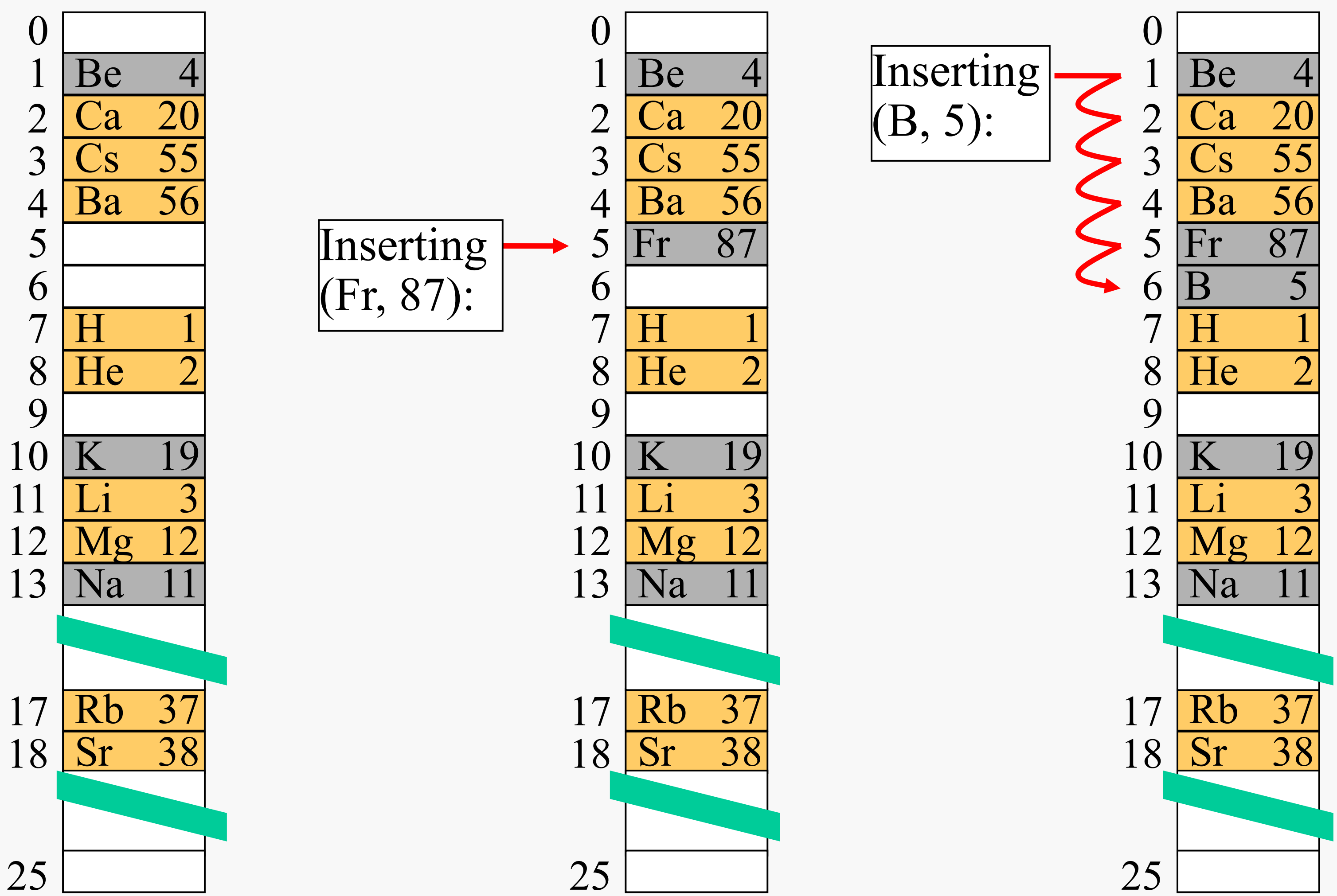
2. Repeat:
   2.1. If *b* is never-occupied:
      2.1.1. Terminate with answer *none*.

   2.2. If key(*b*) == *target-key*:
      2.2.1. Terminate with answer *b*.

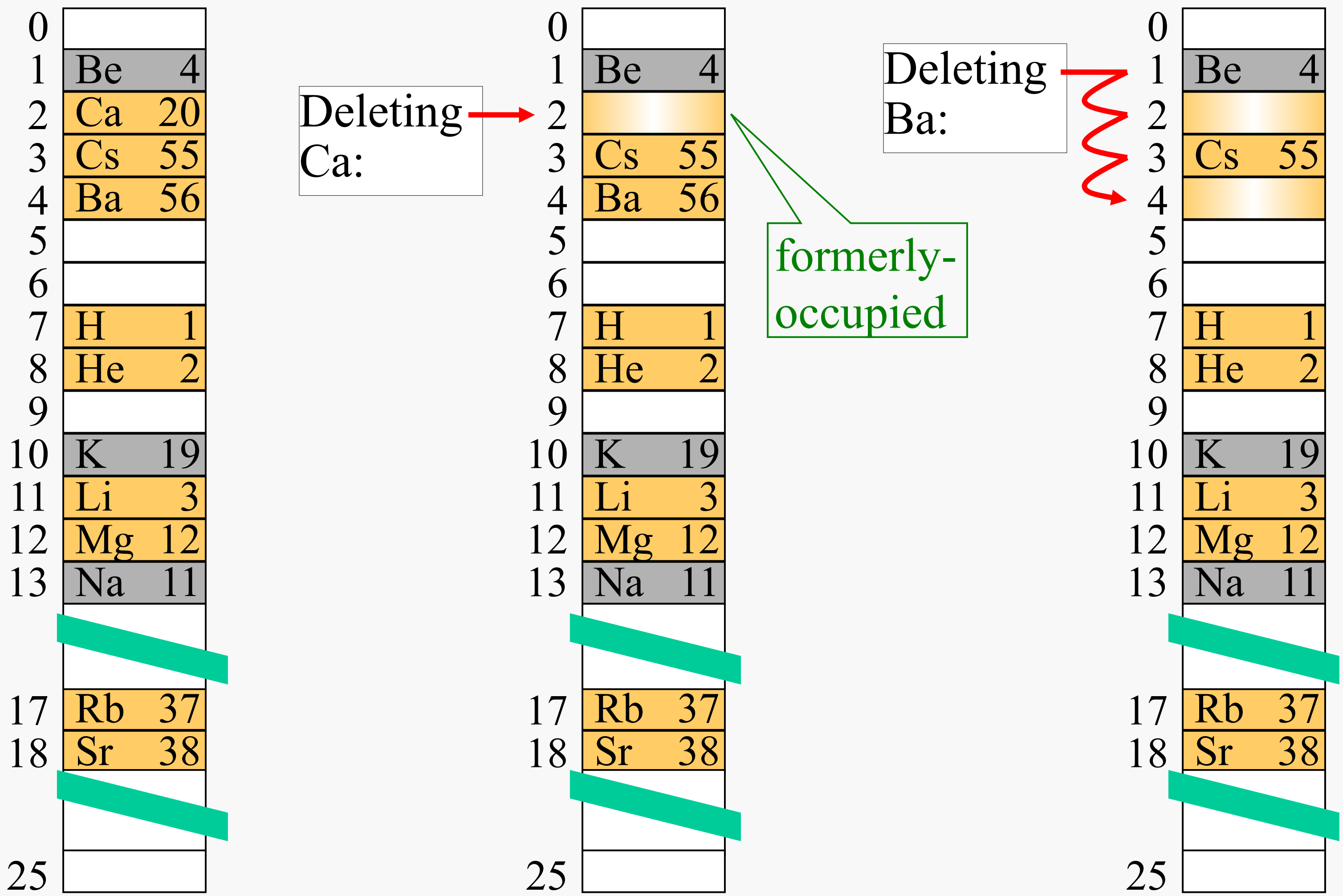
   2.3. If *b* is formerly-occupied, or key(*b*) != *target-key*:
      2.3.1. Increment *b* modulo *m*.



OBHT: Insertion



OBHT: Deletion



OBHT: Analysis

- Analysis of OBHT search/insertion/deletion algorithm (counting comparisons): Let the number of entries be n.
- In the **best case**, no cluster contains more than (say) 4 entries:

  Max. no. of comparisons $= 4$

  Best-case time complexity is $O(1)$.
- In the **worst case**, one cluster contains all n entries:

  Max. no. of comparisons $= n$

  Worst-case time complexity is $O(n)$.

Acknowledgements

- Some of this material has been taken from a variety of sources
- the most notable of which is from

Tamassia, Goldwasser and Goodrich "Data Structures and Algorithms in Python " John Wiley & Sons.

Watt, and Brown"Java Collections: An Introduction to Abstract Data Types, Data Structures and Algorithms" John Wiley & Sons.