SCC.121: **Indexed Retrieval**

Amit K. Chopra

amit.chopra@lancaster.ac.uk

Earlier…

- Searching linear arrays:
  - binary searching efficient for sorted arrays
  - for unsorted arrays, have to use linear search

- Linear search – O(*N*): linear time

- Binary search – O($\log_2 N$): logarithmic

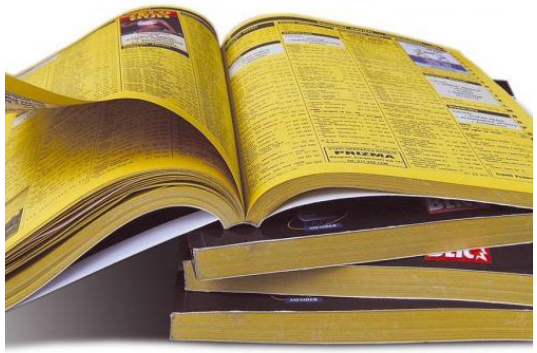
Today

- Indexed retrieval: for storing and retrieving objects or records
- Examples based on alphabetic keys
- Collisions and collision-free storage

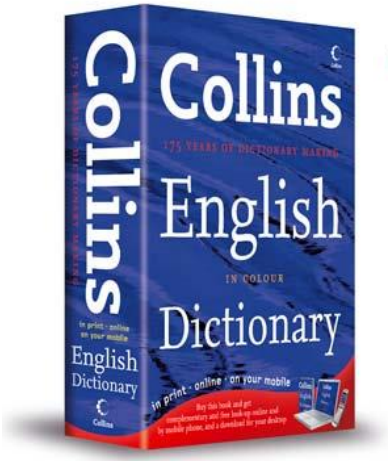
Next lecture

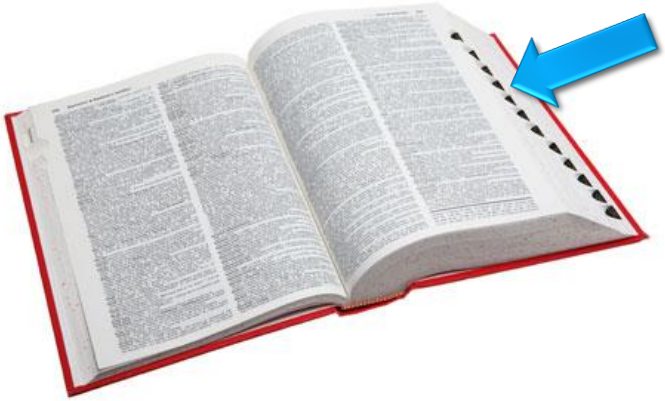
- Hashing

Indexed Retrieval



Alphabetical indexes





Keys

- Indexed retrieval requires an identifier to store and retrieve each *object* or *record*.
- An attribute of each object can be used as a *key* for storage and retrieval
- *The **key** is a **unique** name or **identifier** for each object.*
- The use of keys allows a set of objects / records to be **stored in sorted order**, and so binary search can be used for retrieval

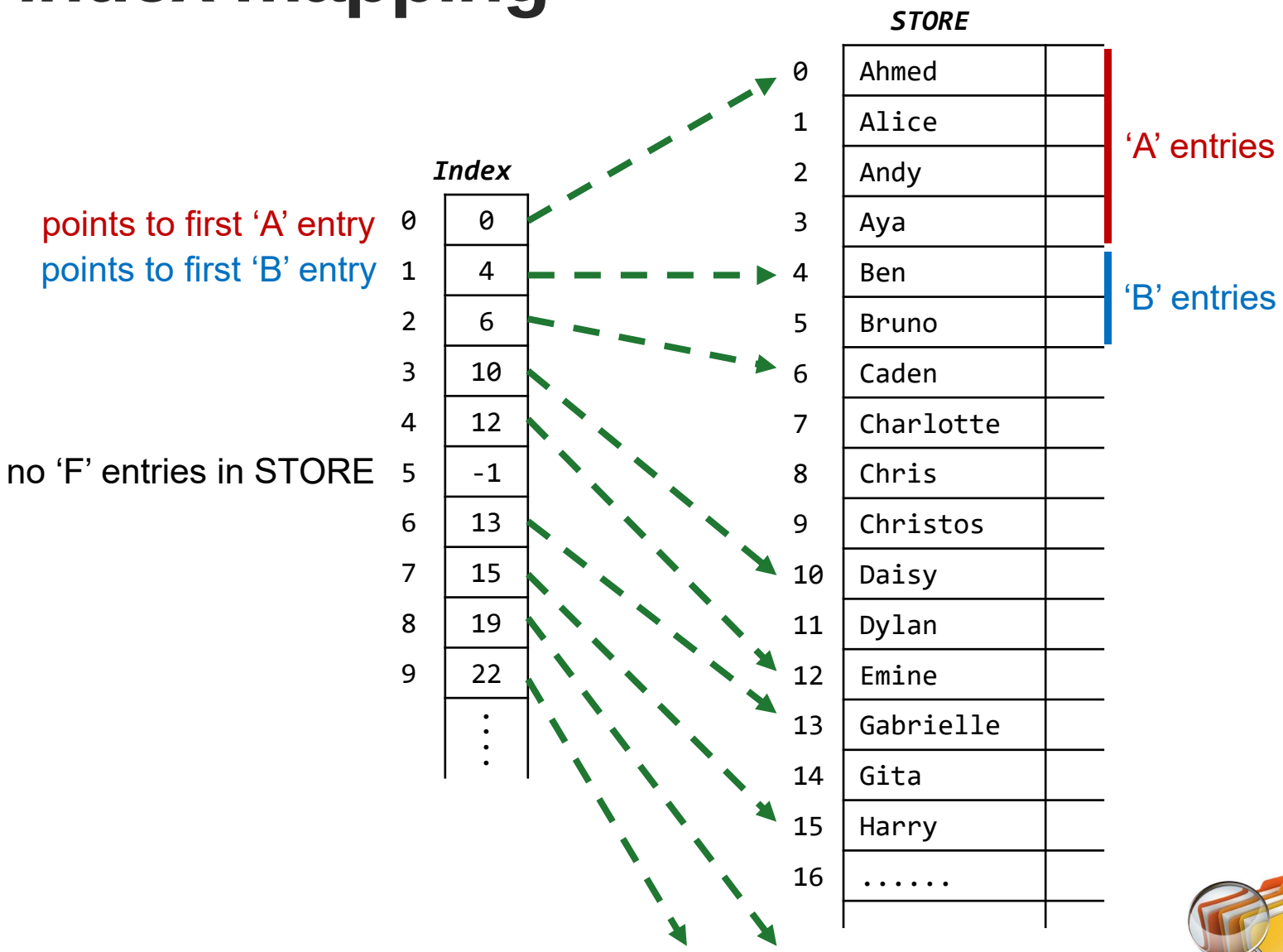
Keys into indexes

- An array of sorted keys can fall into natural ‘sections’, e.g. for alphabetic keys (people’s names):
  - → some “A” records,
  - → some “B” records,
  - → some “C” records…
- A new array can be constructed to index the first element of each section:

  ***index*[0]** holds the index of the first “A” record
  ***index*[1]** holds the index of the first “B” record
  ***index*[2]** holds the index of the first “C” record
- Note: if there are no keys with a specific letter then

  ***index*[*n*] = -1**

The index mapping



Indexed retrieval with a linear search

Given ***name***, look for record in *STORE* with matching key:

```
take first letter L from name;
convert L to numeric value V;
P = Index[V];
if (P<0) then no records start with letter L → fail!
while (first letter of key of STORE[P]==L)
{
  if (key==name)                    //  record found!
  {  return STORE[P];}
  P++;                              //  move to next record
}
//  if loop exhausts, name was not found.
```

Indexed retrieval with linear search: Efficiency

linear search of the section starting at `STORE[P]`

Worst case:

All keys start with same letter – all records are in the same section:

linear search through the whole array; O(*N*).

In practice:

Depends on the size of sections (for each letter); if all sections are roughly equal size, it can be up to 26x faster than linear search without indexing.

Of course, because some names beginning with some letters are more common than others, the gain will be probably smaller

Indexed retrieval with binary search: Efficiency

if some of sections are large, we can do a ***binary search*** of the section starting at `STORE[P]`

Worst case:

Everything in one section: binary search whole array, $O(\log_2 N)$

In Practice:

Depends on size of sections, but could up to 26x times faster than binary search without indexing.

Collisions

Collision-free storage

Test whether a word is in a set of words:

break case data for if make null quit return save test value while

They all start with different letters

- Hold the set in an array of 26 elements with the index of each word determined by its first letter: e.g. A[1]="break", A[2]="case", A[22]="while"
- Take the first letter of the word to test
- Check it in the appropriate array element.
- It *either* matches *or* the word is definitely not in the set.

Collision-free storage: algorithm

```
Given name, look for record in STORE with
 matching key:
    take first letter L from name;
    convert L to numeric value V;
    if (STORE[V].key == name)
    {
     return STORE[V];  // record found
    }
    else ......        // no matching key
```

Note: only *one* element of *STORE* is tested; if it does not match, the retrieval *fails*. Hence, this is a fast, *constant time*, O(1), algorithm.

Dealing with collisions using indexed retrieval

Solution 1 – Indexed retrieval (as above)

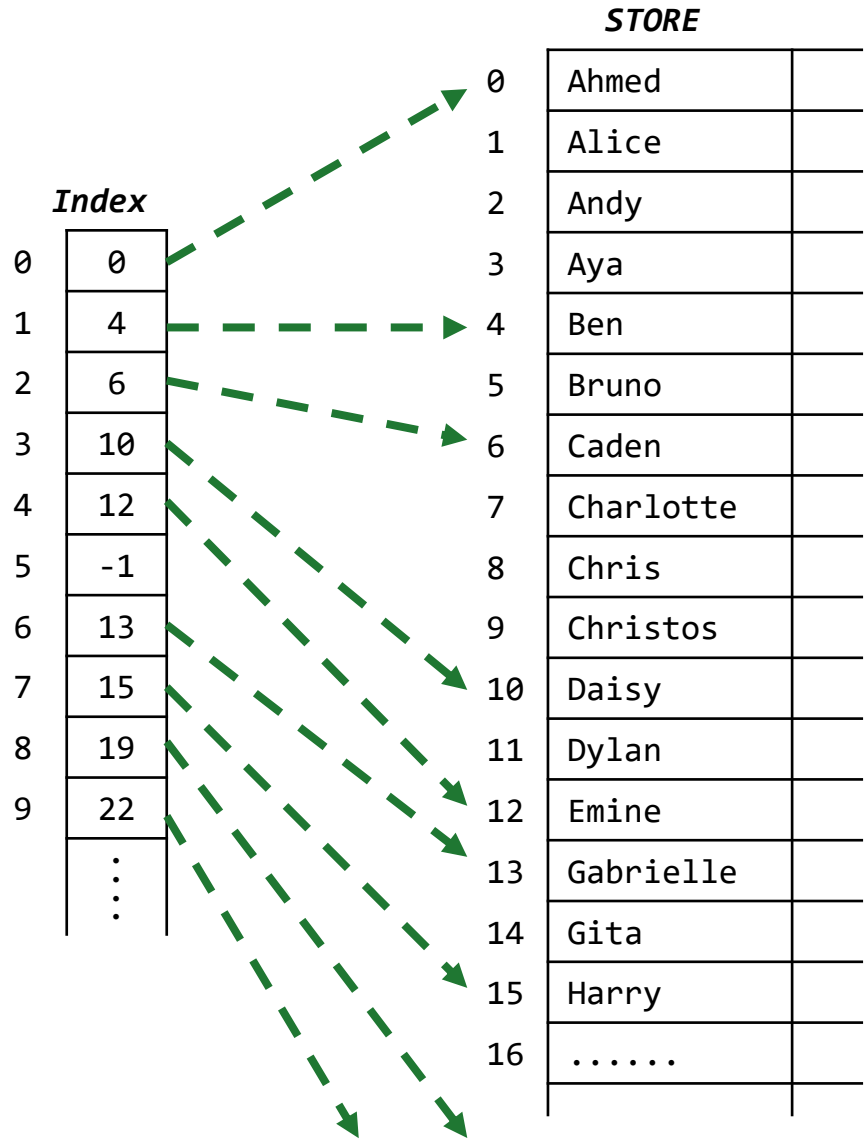
- put keys into a sorted array
- use an index array to find relevant section

but

to insert extra data, the index array must be recomputed

Indexed retrieval: adding an entry (1)

What if we need to add an entry for Christophe?

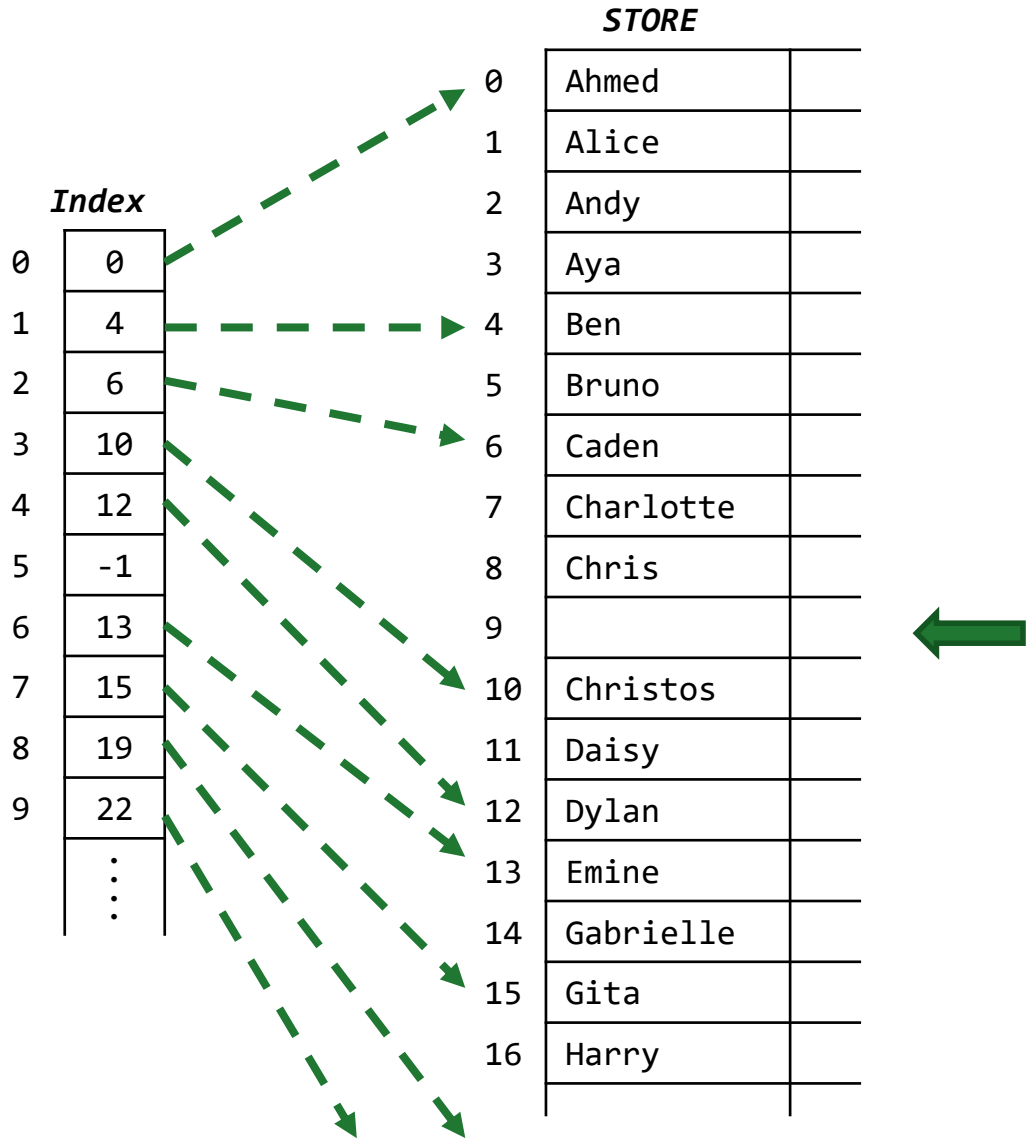


Indexed retrieval: adding an entry (2)

What if we need to add an entry for Christophe?

First we have to make room for the entry in the store.

This means all the entries from position 9 onwards have to be shifted into the next slot.



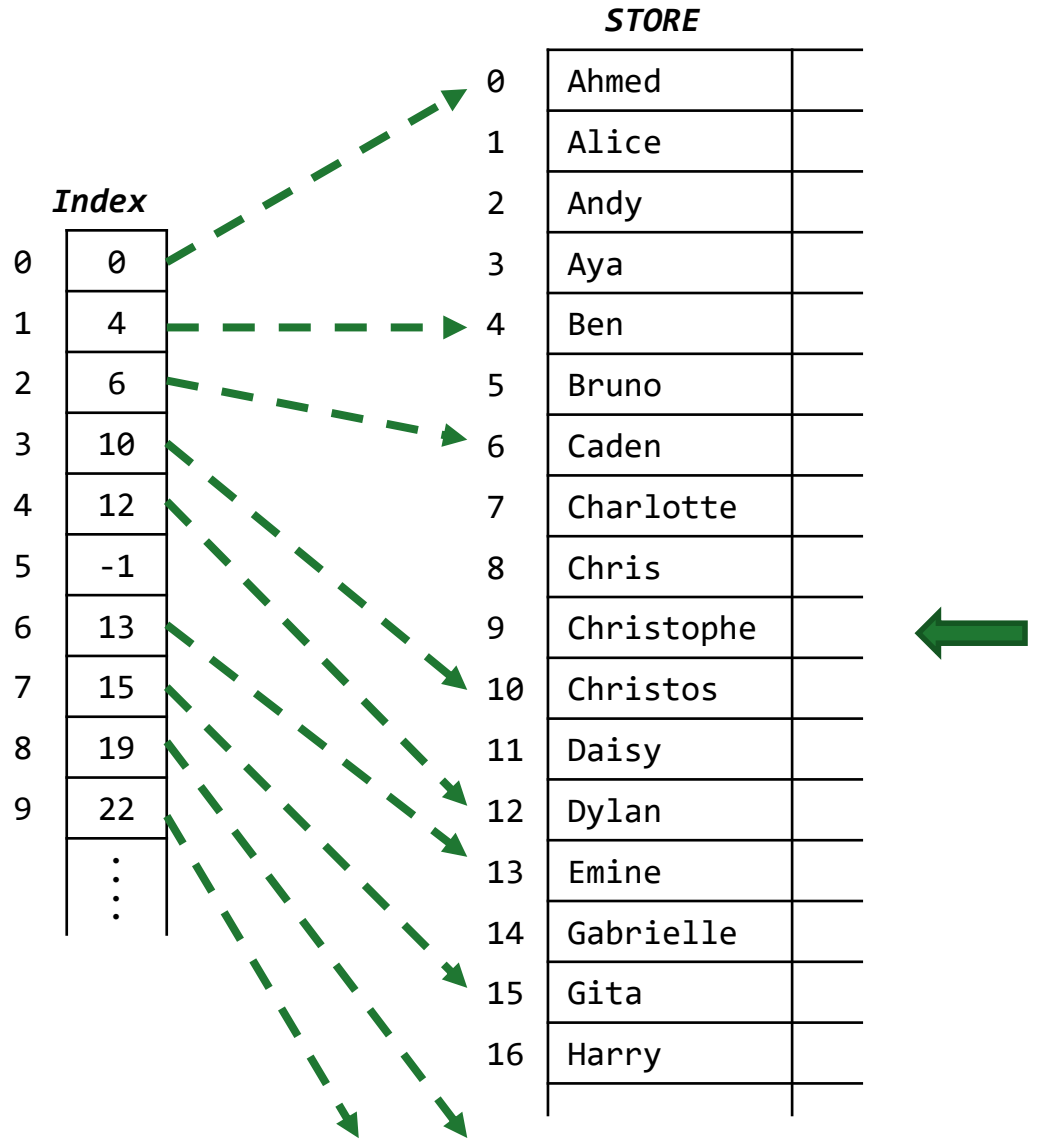
Indexed retrieval: adding an entry (3)

What if we need to add an entry for Christophe?

First we have to make room for the entry in the store.

This means all the entries from position 9 onwards have to be shifted into the next slot.

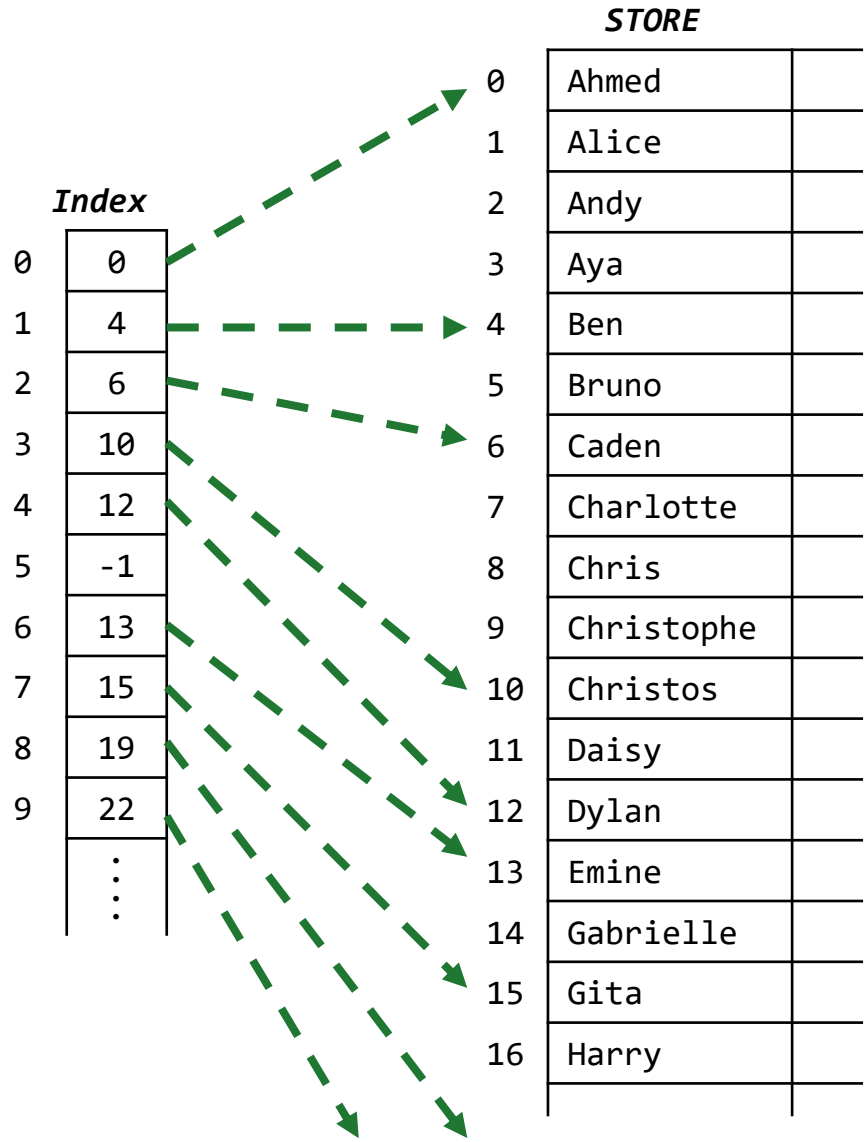
Now we can add Christophe.



Indexed retrieval: adding an entry (4)

What if we need to add an entry for Christophe?

Now the Index is wrong, from element 3 onwards.

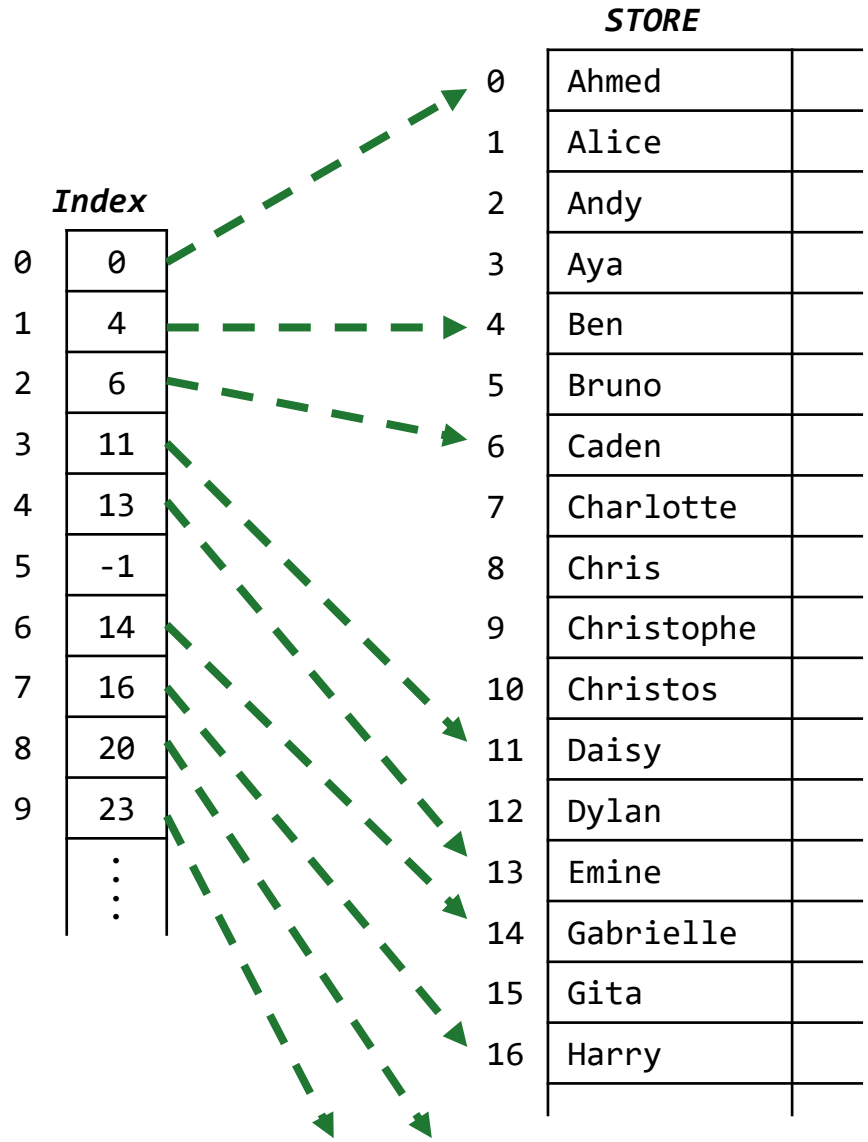


Indexed retrieval: adding an entry (5)

What if we need to add an entry for Christophe?

Now the Index is wrong, from element 3 onwards.

We have to recompute the Index. Adding 1 to every positive value, starting at element 3.



Dealing with collisions using chains (linked lists)

Solution 2 – Using chains

For storage based on first letter of each key:

use a linear array *STORE* of 26 elements representing the alphabet with each element pointing to the appropriate chain; i.e. *STORE*[0] points to a chain of objects whose keys starts with ‘A’

```
STORE[0] → Ahmed → Alice → Andy → Aya
STORE[1] → Ben → Bruno
STORE[2] → Caden → Charlotte → Chris → Christos
STORE[3] → Daisy → Dylan
STORE[4] → Emine
STORE[5]
STORE[6] → Gabrielle → Gita
 .......
```

Indexed retrieval algorithm using chains

Given ***name***, look for record in *STORE* with matching key:

```
take first letter L from name;
convert L to numeric value V;
obtain chain pointer  P  from  STORE[V];
while (P!=null)
{
 if (P.key==name)            //  record found!
  {  return P; }
 P=P.next;                 //  move to next record
}
//  if loop exhausts, name was not found.
```

Dealing with collisions using chains: Efficiency

***Worst case*:**
Every object is on the same chain
Like linear search, O(*N*)

Practically: Much faster – if the objects are distributed across many chains, so same improvement as with using arrays.

Advantage

- Chains are dynamic, so extra objects can be inserted more easily
- No need to recompute index