

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#define MAXPAROLA 30
#define MAXRIGA 80
```

```
int main(int argc, char *argv[])
```

```
{
    int freq[MAXPAROLA]; /* vettore di contatori
delle frequenze delle lunghezze delle parole */
    char riga[MAXRIGA];
    int i, inizio, lunghezza;
    FILE *f;
```

```
for(i=0; i<MAXPAROLA; i++)
    freq[i]=0;
```

```
if(argc != 2)
```

```
{
    fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
    exit(1);
}
```

```
f = fopen(argv[1], "r");
if(f==NULL)
```

```
{
    fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
    exit(1);
}
```

```
while( fgets( riga, MAXRIGA, f ) != NULL )
```



Symbol Tables

Direct Access Tables

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Definition

- ❖ A Symbol Table is a data structure with records including a key and allowing operations such as
 - **Insertion** of a new record
 - **Search** of a record with a given key
 - Delete, select, order, union
- ❖ Sometimes symbol tables are denoted with the term **dictionary**
 - Many applications need fast searches
 - Dictionaries are very important in computer engineering

Applications

❖ Symbol tables have several applications

| Applications | Target, i.e., searching | Key | Return Value |
|---------------------|--------------------------------|------------|---------------------|
| Dictionary | Definition | Word | Definition |
| Book index | Relevant pages | Word | Page list |
| DNS Lookup | IP address given its URL | URL | IP address |
| Reverse DNS Lookup | URL given its IP address | IP address | URL |
| File system | File on disk | File name | Disk location |
| Web search | Web page | Keyword | Page list |

Implementations

❖ Symbol tables have several implementations

❖ Linear structures

➤ Direct Access Tables

➤ Arrays

- Unordered
- Ordered

➤ Lists

- Unordered
- Ordered

➤ Hash Tables

❖ Tree structures

➤ Binary Search Trees (BSTs)

➤ Balanced Trees

- 2-3-4
- RB-tree
- B-tree

Already studied - To be done - Not analyzed in this course

Complexity

❖ Different data structures have different performances

Worst case complexity

| Data Structure | Insert | Search |
|-----------------------------|----------|----------|
| Direct Access Table | 1 | 1 |
| Unordered Array | 1 | n |
| Ordered Array Linear Search | n | n |
| Ordered Array Binary Search | n | $\log n$ |
| Unordered List | 1 | n |
| Ordered List | n | n |
| BST | n | n |
| RB-tree | $\log n$ | $\log n$ |
| Hashing | 1 | n |

Complexity

Average case complexity

| Data Structure | Insert | Search | |
|-----------------------------|----------|----------|----------|
| | | Hit | Miss |
| Direct Access Table | 1 | 1 | 1 |
| Unordered Array | 1 | $n/2$ | n |
| Ordered Array Linear Search | $n/2$ | $n/2$ | $n/2$ |
| Ordered Array Binary Search | $n/2$ | $\log n$ | $\log n$ |
| Unordered List | 1 | $n/2$ | n |
| Ordered List | $n/2$ | $n/2$ | $n/2$ |
| BST | $\log n$ | $\log n$ | $\log n$ |
| RB-tree | $\log n$ | $\log n$ | $\log n$ |
| Hashing | 1 | 1 | 1 |

Direct Access Tables

- ❖ All search algorithms analyzed so far in the course are based on comparisons
 - For example searching for a key into an array, a list or a BST implies comparing this key with the element or node keys visiting the data structure with a specific logic
- ❖ **Direct Access Tables** and **Hash Tables** use a different paradigm
 - They compute the position of the key within the data structure by applying a function to the key

Direct Access Tables

❖ Problem definition

- Suppose we need to store a key **k** belonging to a universe **U** of key in a table, with
 - $k \in U$
 - No two elements have the same key
 - U has cardinality $|U|$

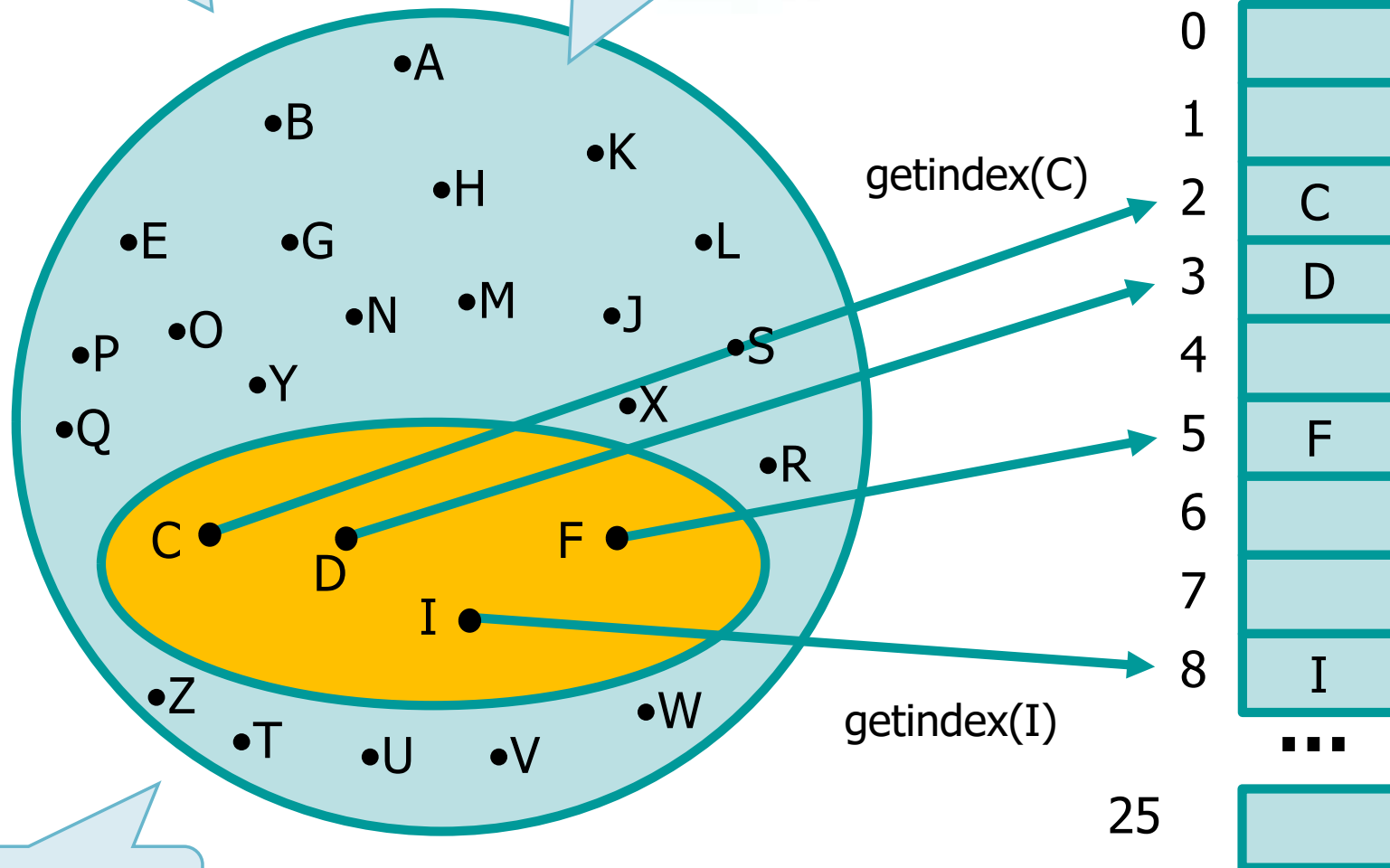
❖ Core ideas

- We can use an array to store the keys (and the related data fields)
 - The array (**st**) has size equal to $|U|$
- We need to map each key ($k \in U$) into a specific element of the array

Direct Access Tables

There is a 1:1 mapping between $k \in U$ and elements in st

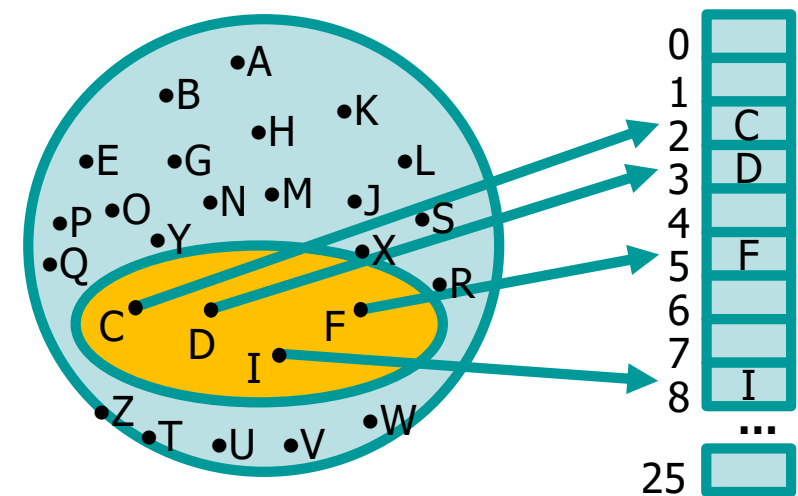
U (Universe of keys)



K (used keys)

Direct Access Tables

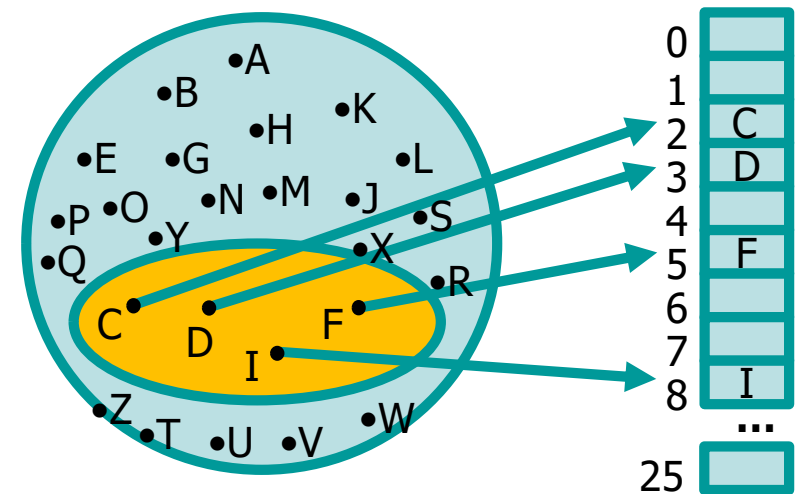
- ❖ We have two problems
 - As the array **st** has size equal to $|U|$, the cardinality of U must be small to be able to allocate the array **st**
 - We always use $|U|$ elements even when we want to store a small subset of $|U|$



Direct Access Tables

❖ We have two problems

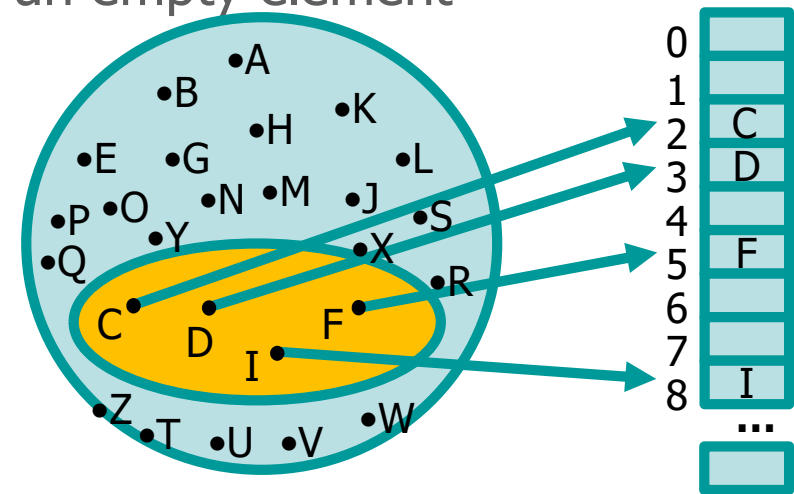
- We need to understand how to map keys into elements
 - This may be simple in specific cases, but the keys are not necessarily integer values
 - The mapping between keys and array indices may be complex



Direct Access Tables

- ❖ To create the mapping key-index we have to design a function (**getindex**) that given a key k
 - Returns an integer from 0 to $|U|-1$, acting as an array index
 - If the key k is in the table
 - **st[getindex(k)]** stores it
 - If the key k is not in the table
 - **st[getindex(k)]** stores an empty element

There is a 1:1 mapping between $k \in U$ and elements in st



Direct Access Tables

❖ This looks simple enough, but **getindex** must be general

➤ If keys are integers from 0 to $|U|-1$

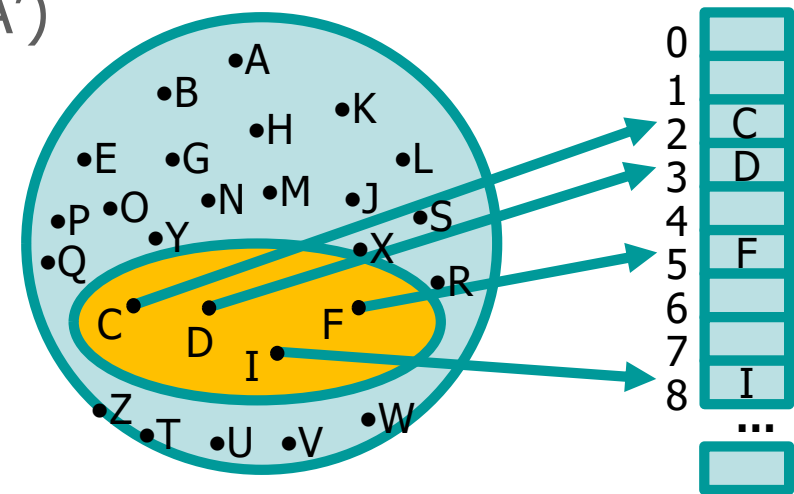
▪ $\text{getindex}(k) = k$

➤ If keys are small (capital) letters in the English alphabet (i.e., a-z or A-Z)

▪ $\text{getindex}(k) = k - ((\text{int}) 'a')$

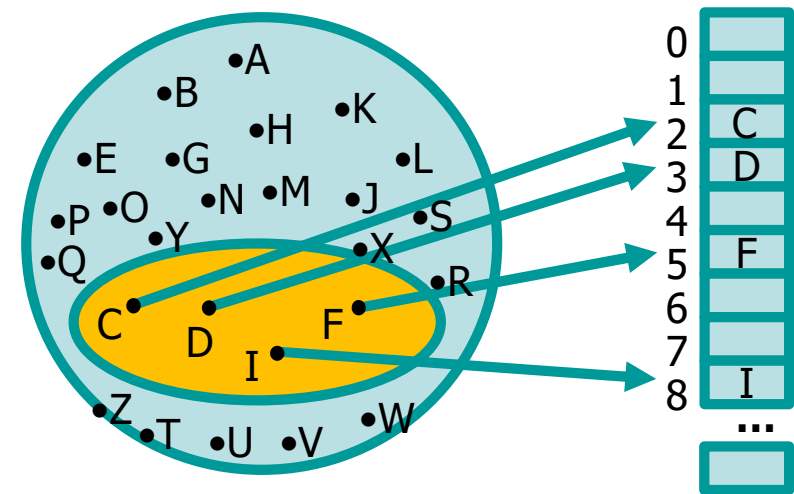
▪ $\text{getindex}(k) = k - ((\text{int}) 'A')$

ASCII for 'a' is 97, thus 'a' is mapped onto 0 and 'z' is mapped onto 26.
Same consideration for 'A' (ASCII 65).



Direct Access Tables

- If keys are generic values
 - Function **getindex** has to map those keys into integer values in the range $[0, |U|-1]$
 - This may be very complex



Advantages

- ❖ Complexity plays in favour of direct access tables
 - Insert, search, and delete complexity
 - $T(n) = \Theta(1)$
 - Init complexity
 - $T(n) = \Theta(|U|)$
 - Memory usage
 - $S(n) = \Theta(|U|)$

Disadvantages

- ❖ Limits are due to
 - For large $|U|$ the array **st** cannot be allocated
 - Direct access tables can be used only for small $|U|$
 - Thus, if $|U|$ is large direct tables cannot be used
 - If $|K| \ll |U|$ there is a memory loss
- ❖ Function **getindex** has to be properly designed depending on the key type

Disadvantages

- ❖ Direct access tables have restricted practical applications
 - Used to convert keys into integers (and vice-versa) with a cost equal to 1
- ❖ When $|U|$ is large or keys are complex, direct access tables must be extended into **Hash Tables**
 - With hash-tables the **1:1** mapping between keys and array indices **is lost**
 - We must map "many" elements in a "small" table