Searching

Definition

- **Searching:** Given a piece of information (key), asked to find a record that contains other information associated with the key.

Examples:

- Given a name, find personnel records.
- Given a student id number, find student record.

Algorithms for Searching

- The algorithm used for searching depends on the organization and structure of data.

Data Structure	Algorithm to be used
Linked List	Sequential Search
Sorted Array	Binary Search
Unsorted Array	Sequential Search
Hash Table	Hashing

- Searching over a file stored on disk → external searching
- Searching over an array, or a linked list → internal searching

Concepts

Record: a collection of related fields.

Field: the smallest logically meaningful unit of information in a file.

Key: a field, or a combination of fields, that may be used in a search.

- **Primary key**: a key that uniquely identifies a record (e.g., student id number) → |Result| <= 1
- **Secondary key**: other keys that may be used for search. → 0<= |Result| <= n
  - Student name
  - age
  - Student name + age

Sequential Search for a Primary Key

```
int sequentialSearch(int D[], int key, int n){
    int i;
  for (i = 0; i < n; i++){
      if (D[i]== key)
          return i;
    }
  return -1;
}
```

Analysis of Sequential Search for a Primary Key

```
int sequentialSearch(int D[], int key, int n){
   int i;
  for (i = 0; i < n; i++){
      if (D[i]== key)
  return i;
    }
  return -1;
}
```

Unsuccessful Search: ➔ O(n)

Successful Search:

Best-Case: *key* is in the first location of the array ➔O(1)

Worst-Case: *key* is in the last location of the array ➔O(n)

Average-Case: The number of key comparisons 1, 2, …, n

➔ O(n) $\frac{\sum_{i=1}^{n} i}{n} = \frac{(n^2 + n)/2}{n}$

Sequential Search for a Secondary Key

- **Example**: search for a particular surname from a student array.

Define student record as follows:

```
typedef struct st{
    int student_id;
    char name[15];
    char surname[20];
    char department[40];
    float gpa;
}STUDENT;
```

Sequential Search for a Secondary Key

```
int secondaryKeySearch(STUDENT D[],
                       int N,
                       int Result[],
                       char *key){
   int i=0, k;
   for (k=0; k<N; k++){
      if (strcmp(D[k].surname,key)==0){
         // found, insert into Result array
         Result[i]=k;
         i++;
      }
   }
   return i;
}
```

Motivation for Binary Search

- Sequential search is
  - easy to implement,
  - efficient for short lists, but
  - disaster for long ones.

Example: Find «Thomas Smith» in a large phone book.

To find any item in a long sorted list

→use binary search.

Binary Search Algorithm

- Compare key with the middle element of the list.
- If key is equal to the middle element, then found, return the address of the middle element.
- If key is less than the middle element, restrict attention to the first half of the list.
- Otherwise, continue with the second half of the list.
- Continue this process until the key is found, or the sublist becomes empty.

Binary Search

```
int binarySearch(int D[], int N, int key) {
    int left =0;
    int right = N -1;
    int middle;

    while (left <= right) {
      middle = (left + right)/2;
      if (key == D[middle])
          return middle;
      else if (key > D[middle])
         left = middle + 1;
      else
           right  = middle - 1;
    }
    return -1;
}
```

Binary Search – Analysis

- For an unsuccessful search:
  - The number of iterations in the loop is $\lfloor \log_2 n \rfloor + 1$

    ➔ $O(\log_2 n)$
- For a successful search:
  - ***Best-Case:*** The number of iterations is 1. ➔ $O(1)$
  - ***Worst-Case:*** The number of iterations is $\lfloor \log_2 n \rfloor + 1$ ➔ $O(\log_2 n)$
  - ***Average-Case:*** The avg. # of iterations < $\log_2 n$ ➔ $O(\log_2 n)$

0	1	2	3	4	5	6	7	← an array with size 8
3	2	3	1	3	2	3	4	← # of iterations

The average # of iterations = 21/8 < $\log_2 8$

How much better is $O(log_2 n)$?

n	***O(log₂n)***
16	4
64	6
256	8
1024 (1KB)	10
16,384	14
131,072	17
262,144	18
524,288	19
1,048,576 (1MB)	20
1,073,741,824 (1GB)	30