Data Structures

Lecture 12: Search

Dongbo Min
Department of Computer Science and Engineering
Ewha Womans University, Korea
E-mail: dbmin@ewha.ac.kr

Contents

- Search definition

- Search in unordered array
  - Sequential search, improved sequential search

- Search in ordered array
  - Sequential search, binary search, indexed sequential search, interpolation search

Search

- Search
  - Finding data from multiple sources
  - Computational efficiency does matter

- Search key
  - A key that distinguishes items

- Data structure used for search
  - Array, linked list, tree, graph, etc

Sequential Search

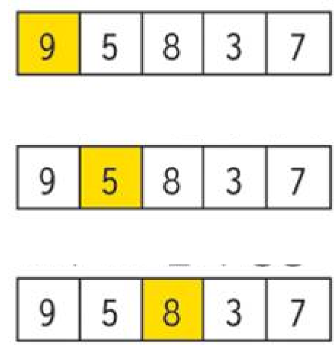
- Sequential search
  - The simplest search method
  - Check an unordered array sequentially

- The number of comparisons (for n inputs)
  - For successful search: $(n+1)/2$ comparisons (on *average*)
  - For failed search: n comparisons (tight bound)

Time complexity: $O(n)$

```c
int seq_search(int key, int low, int high)
{
        int i;
        for (i = low; i <= high; i++)
                if (list[i] == key)
                        return i;
        return -1;
}
```

Search for 8



Improved Sequential Search

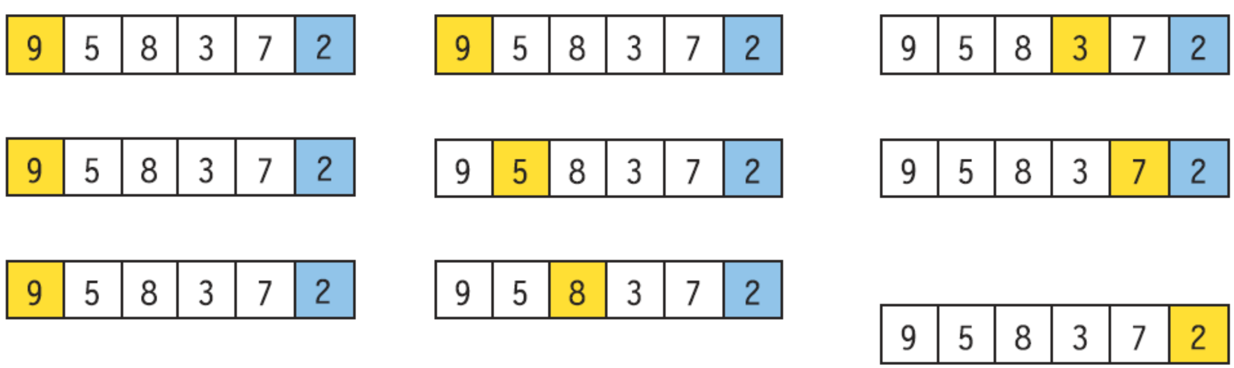
- Save the search key at the end of list
- Terminate the loop when finding key value

```
int seq_search2(int key, int low, int high)
{
        int i;
        list[high + 1] = key;
        for (i = low; list[i] != key; i++)
              ;
        if (i == (high + 1)) return -1;
        else      return i;
}
```

# of comparisons is smaller than sequential search.

Search for 8

Search for 2



Successful search

Failed search

Binary Search

- Suitable for searching in an *ordered* array

- Procedure
  1. Start at the center of the array
  2. If the search key is identical to the value at center, terminate it.
  3. Otherwise, go to left or right sub-array
  4. Repeat 2 and 3 until finding the key or the sub-array is null

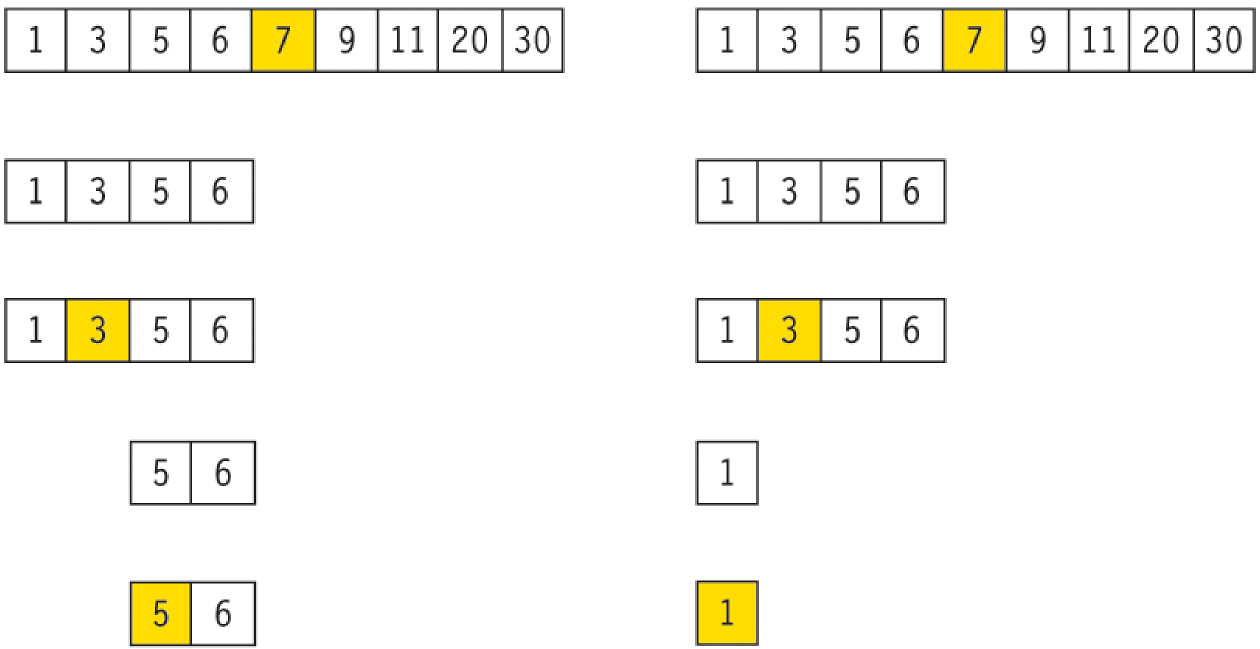
  Time complexity: $O(log_2 n)$

- Example) Search for a specific name in a billion people
  - Binary search: 30 comparisons $(log_2 1{,}000{,}000{,}000)$
  - Sequential search: 0.5 billion comparisons (on average)

```
search_binary(list, low, high)
       middle ← (low + high)/2
       if( key = list[middle] ) return TRUE;
       else if (key < list[middle] )
              return search from list[low] to list[middle-1];
       else if (key > list[middle] )
              return search from list[middle+1] to list[high];
```

Search for 5

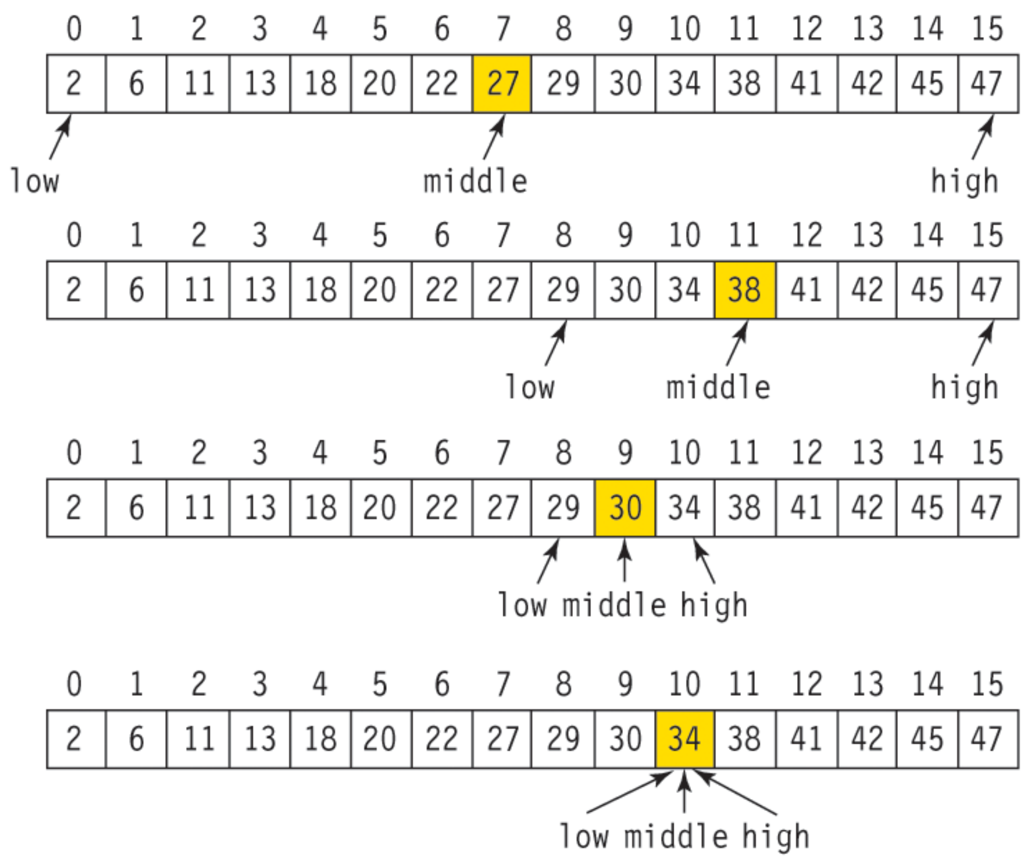
Search for 2



Search success

Search fail

```c
int search_binary2(int key, int low, int high)
{
        int middle;
        while (low <= high) { // when there are elements to be checked
                middle = (low + high) / 2;
                if (key == list[middle]) return middle; // search success
                else if (key > list[middle]) low = middle + 1; // go to left sub-array
                else high = middle - 1; // go to right sub-array
        }
        return -1; // search fail
}
```



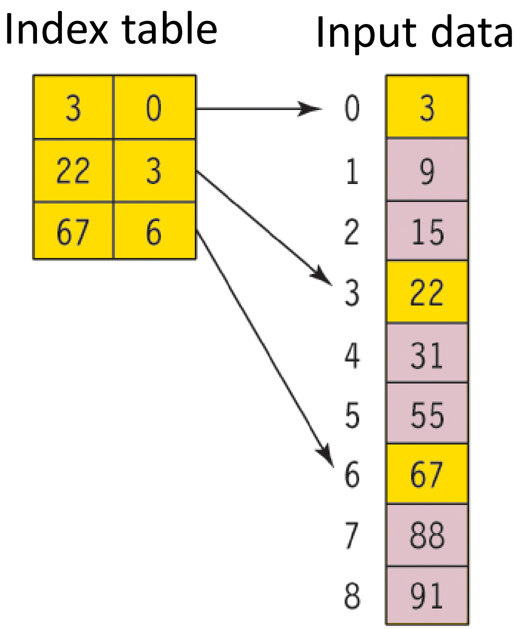
Indexed Sequential Search

- Index table
  - is used to increase the efficiency of sequential search
  - stores data *regularly* sampled from the input data list (every $\lceil n/m \rceil$)

- Assumption
  - Both the input data list and the index table are sorted

Time complexity: $O(m + n/m)$

m: Index table size
n: size of main data list



```c
#define INDEX_SIZE    4  //index table size
#define INPUT_SIZE    18 //input data size

typedef struct itable {
           int key;
           int index;
} itable;
itable index_list[INDEX_SIZE];
int *list;

int seq_search(int key, int low, int high)
{
           int i;
           for (i = low; i <= high; i++)
                      if (list[i] == key)
                                 return i;
           return -1;
}

void generate_index_table()
{
           int step = ceil((float)INPUT_SIZE / (float)INDEX_SIZE);
           for (int i = 0; i < INDEX_SIZE; i++)
           {
                      index_list[i].index = i*step;
                      index_list[i].key = list[i*step];
           }
}
```

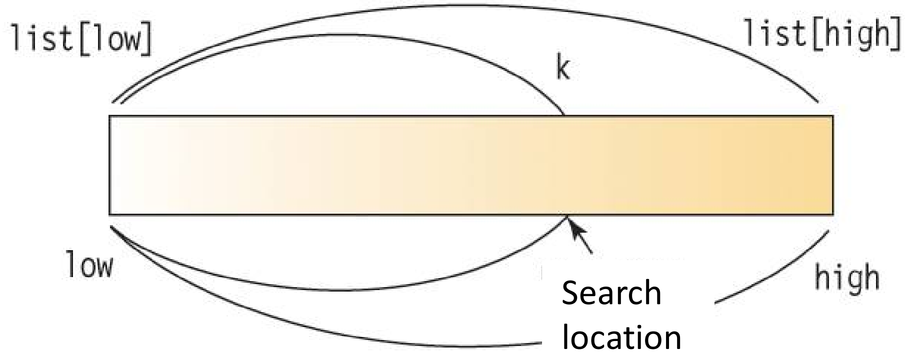
```
int search_index(int key)
{
	int i, low, high;
	if (key<list[0] || key>list[INPUT_SIZE - 1]) {
		return -1;
	}
	for (i = 0; i < INDEX_SIZE; i++)
		if (index_list[i].key <= key && index_list[i + 1].key > key)
			break;
	if (i == INDEX_SIZE)
		return -1;
	else if (i == INDEX_SIZE - 1) {
		low = index_list[i].index;
		high = INPUT_SIZE - 1;
	}
	else {
		low = index_list[i].index;
		high = index_list[i+1].index;
	}

	return seq_search(key, low, high);
}

void main()
{
	int list_tmp[INPUT_SIZE]={1, 3, 6, 9, 10, 12, 15, 20, 24, 28, 29, 32, 35, 39, 45, 60, 68, 75};
	list = list_tmp;

	generate_index_table();
	int pos = search_index(75);
	if (pos == -1)
		printf("search failed\n");
	else
		printf("search success. position: %d\n", pos);
}
```

Interpolation Search

- A method for *predicting* the location of a search key
- It is similar to the binary search, but the list is *unevenly* divided
- It works well when the data is relatively evenly distributed.

Example) Search in dictionary
Words starting with ‘z’ are found at the end of the dictionary
Words starting with 'a' are found at the beginning

$$(list[high] - list[low]) : (k - list[low]) = (high - low) : search\ location - low$$



$$search\ location = \frac{(k - list[low])}{(list[high] - list[low])}(high - low) + low$$

Interpolation Search

$$search\ location = \frac{(k - list[low])}{(list[high] - list[low])}(high - low) + low$$

$search\ location$ = (55-3)/(91-3)*(9-0) + 0 = 5.31 ≒ 5

0	1	2	3	4	5	6	7	8	9
3	9	15	22	31	55	67	88	89	91

```c
#define INPUT_SIZE18//input data size
int *list;

int search_interpolation(int key)
{
            int low, high, j;
            low = 0;
            high = INPUT_SIZE - 1;
            while ((list[high] >= key) && (key > list[low])) {
                j = (float)(key - list[low]) / (float)(list[high] - list[low])*(float)(high - low) + low;
                if (key > list[j])
                        low = j + 1;
                else if (key < list[j])
                        high = j - 1;
                else low = j;
            }
            if (list[low] == key)return low;
            else return -1;
}

void main()
{
            int list_tmp[INPUT_SIZE]={1,3, 6, 9, 10, 12, 15, 20, 24, 28, 29, 32, 35, 39, 45, 60, 68, 75 };
            list = list_tmp;

            int pos = search_interpolation(60);
            if (pos == -1)
                        printf("search failed\n");
            else
                        printf("search success. position: %d\n", pos);
}
```