자료구조

Chap 3. Array

2018년 1학기

컴퓨터과학과
민 경 하

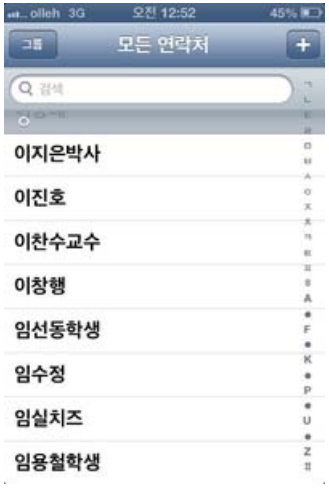
Contents

3. Array

1. Definition of list
2. Definition of array
3. Operations of array
4. Search
5. Insert
6. Delete
7. Performance analysis
8. Implementation tip

1. Definition of list

- List (目錄, in real life)
  - Arranging items in a row (by a special order)
  - The most frequently used data structure in everywhere
  - Why do we use list? <**一目瞭然**>



Phone number

Apps

Music

Bus

PoketmonGo

List

1. Definition of list

- List (in computer science)
  - a finite sequence of elements (values)

$$A = (2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ 19)$$

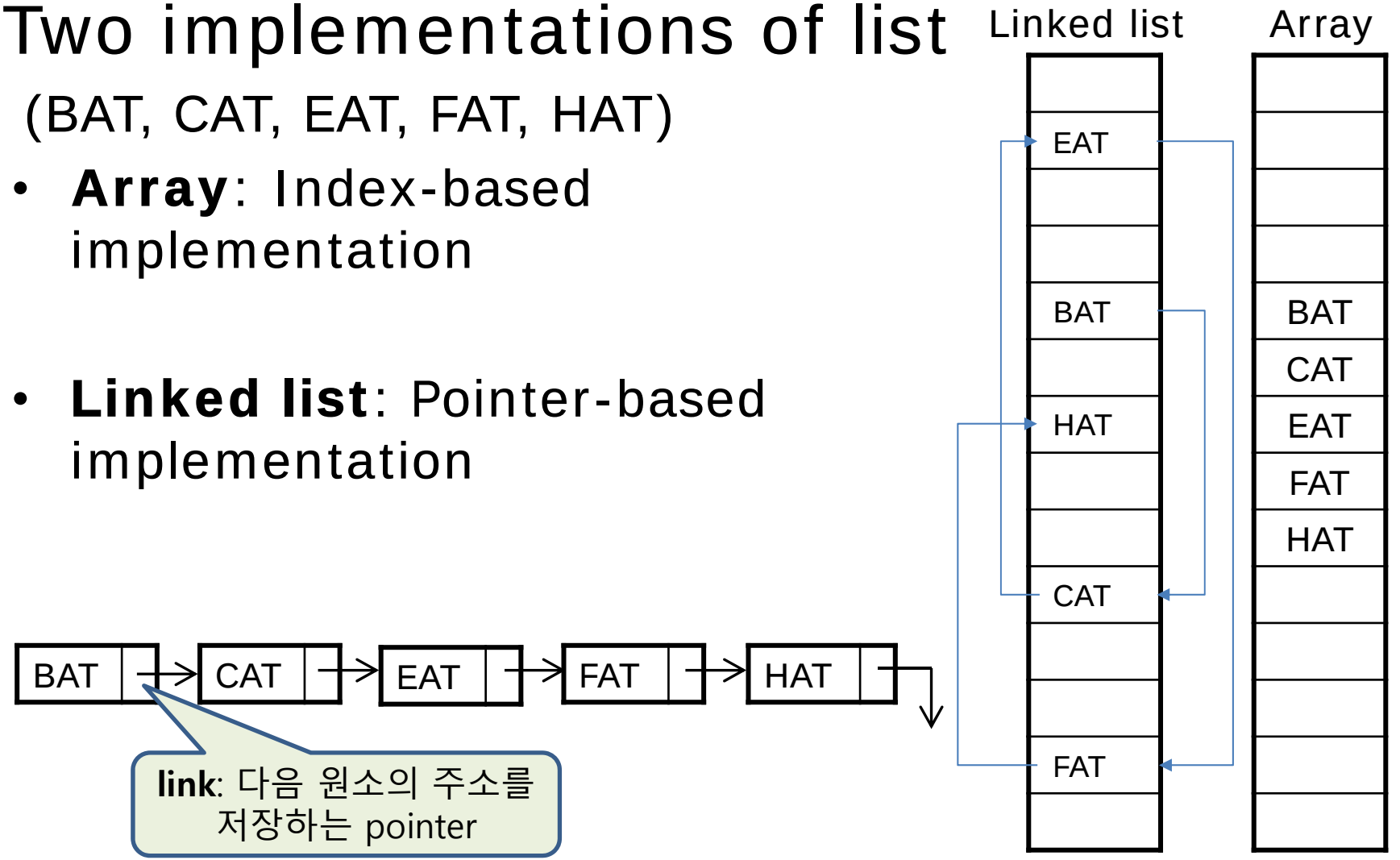
  - Each **element** in the list is mapped into an **index**
    - The most important property of a list

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
2	3	5	7	11	13	17	19

    - (i, a_i) is a pair of (index, element)

1. Definition of list

- Two implementations of list
  (BAT, CAT, EAT, FAT, HAT)
  - **Array**: Index-based implementation
  - **Linked list**: Pointer-based implementation



1. Definition of list

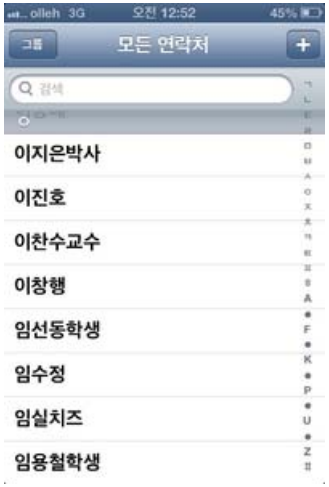
- Two types of list
  - Sorted list

$$A = (2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ 19)$$

  - Unsorted list

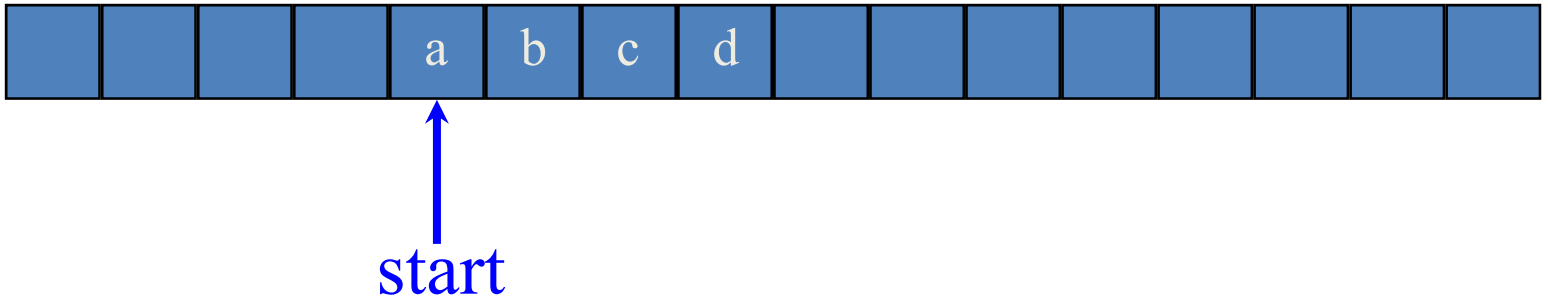
$$B = (11,\ 5,\ 19,\ 2,\ 7,\ 13,\ 17,\ 3)$$

원소의 위치를 예측할 수 있음
→ Search가 편함
→ Insert/delete가 불편함



2. Definition of array

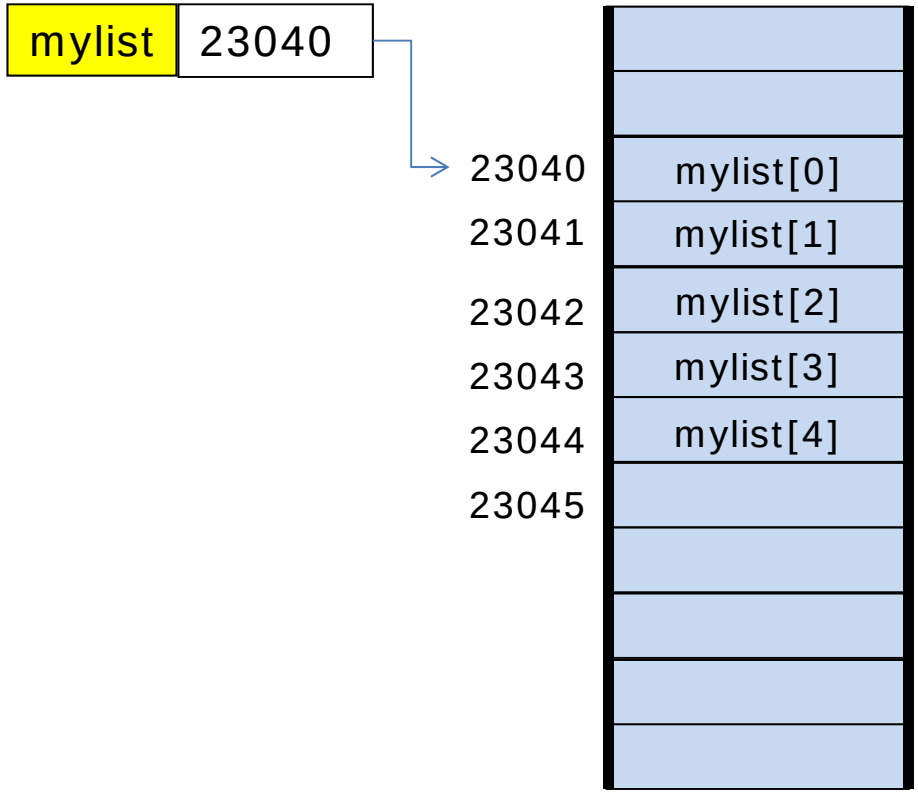
- Array
  - An index-based implementation of a list
  - a consecutive set of memory allocations
    - x = [a, b, c, d]
    - Location of x ➔ start
    - Location of x[i] = (start + i)
  - Four values are stored with **one** location



2. Definition of array

- Array
  - Implementation of array in memory

```
int mylist[5];
```



2. Definition of array (implementation tip)

⑴ Array with friends

```
{
    int A[10];
}
```

- array
  - Consecutive allocations of elements
- size
  - The size of an array
- count
  - The number of elements in an array

2. Definition of array (implementation tip)

⑴ Array with friends (local declaration)

- array
- size
- count

```
#define MAX_SIZE 100//     MAX, SIZE 등 다양한 이름으로 사용
{
    int count = 0;  //     cnt, n 등 다양한 이름으로 사용
    int A[MAX_SIZE];
    ……
    for ( i = 0; i < count; i++ )
        if ( A[i] == x )
    ……
}
```

2. Definition of array (implementation tip)

⑴ Array with friends (extern declaration)

- array
- size
- count

```
#define MAX_SIZE 100//     MAX, SIZE 등 다양한 이름으로 사용
int count = 0;       //     cnt, n 등 다양한 이름으로 사용
int A[MAX_SIZE];
{
……
    for ( i = 0; i < count; i++ )
        if ( A[i] == x )
    ……
}
```

2. Definition of array (implementation tip)

⑵ class can wrap them all

- array
- size
- count

```
class myArray {
    int MAX_SIZE;
    int count;
    int *arr;
};


    myArray A;
```

2. Definition of array (implementation tip)

⑶ Initialization of C and C++

```
#define MAX_SIZE 100
……
    int count = 0;
    int *A = (int *) calloc ( MAX_SIZE, sizeof(int) );
```

```
class myArray {
    int MAX_SIZE;
    int count;
    int *arr;
public:
    void myArray ( int MAX ) {
        MAX_SIZE = MAX;
        count = 0;
        arr = (int *) calloc ( MAX_SIZE, sizeof(int) );
    }
};
……
    myArray A ( 100 );
```

생성자 (constructor)

3. Operations of array

(1) Primary operation (implemented in PL)

- create
  - Create an array of size *n* elements
- retrieve
  - Get an *i*-th element of an array
- store
  - Store *x* in the *i*-th position of an array

array create (n)	int L[10];
element retrieve (A, i)	int x = L[5];
array store (A, i, x)	L[5] = x;

3. Operations of array

(2) Further operation (not implemented)

① **search**

② **insert**

③ **delete**

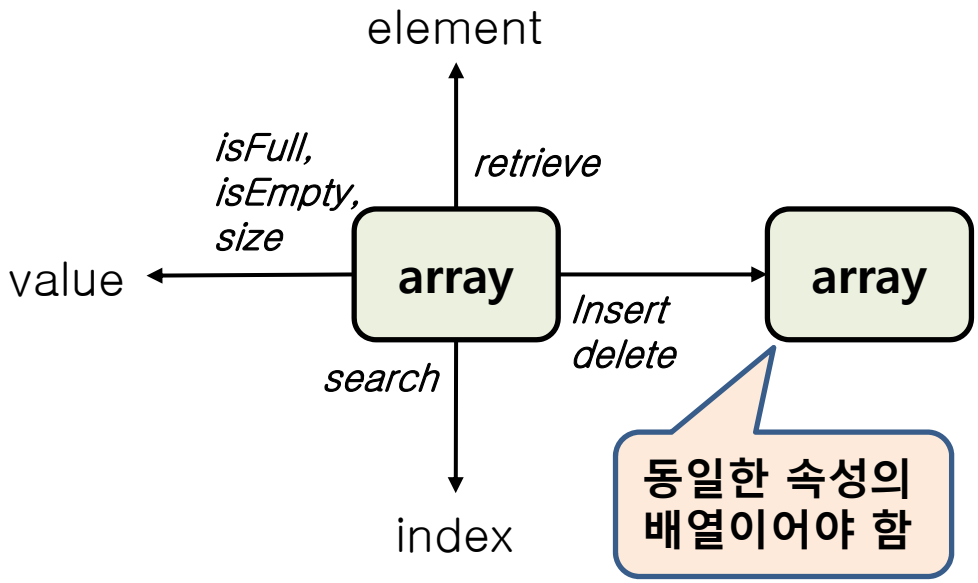
④ resize

⑤ isFull

⑥ isEmpty

⑦ size

……



3. Operations of array

<table>
<tr><th>Operation</th><th>Sorted array
A=[2, 5, 7, 9, 10]</th><th>Unsorted array
A=[5, 2, 7, 10, 9]</th></tr>
<tr><td rowspan="2">Search</td><td>linear_search (A, x)</td><td rowspan="2">linear_search (A, x)</td></tr>
<tr><td>binary_search (A, x)</td></tr>
<tr><td rowspan="3">Insert</td><td rowspan="3">insert_by_value (A, x)
(A, 8) → [2, 5, 7, 8, 9, 10]

insert ( ), insert_by_index ( ) & store ( ) are not allowed</td><td>insert (A, x)
(A, 8) → [5, 2, 7, 10, 9, 8]</td></tr>
<tr><td>insert_by_index (A, i, x)
(A, 3, 8) → [5, 2, 7, 8, 10, 9]</td></tr>
<tr><td>store (A, i, x)
(A, 3, 8) → [5, 2, 7, 8, 9]</td></tr>
<tr><td rowspan="2">Delete</td><td>delete_by_value (A, x)
(A, 5) → [2, 7, 9, 10]</td><td>delete_by_value (A, x)
(A, 5) → [2, 7, 10, 9]</td></tr>
<tr><td>delete_by_index (A, i)
(A, 3) → [2, 5, 7, 10]</td><td>delete_by_index (A, i)
(A, 3) → [5, 2, 7, 9]</td></tr>
</table>

4. Search

① Definition of search

(i) Determine whether the key element is in the array or not

(ii) Return the index of the key element

- Different approach by the type of an array
  - Unsorted array: <7, 5, 9, 3, 1, 6, 8, 4>

```
search ( A, x )
```

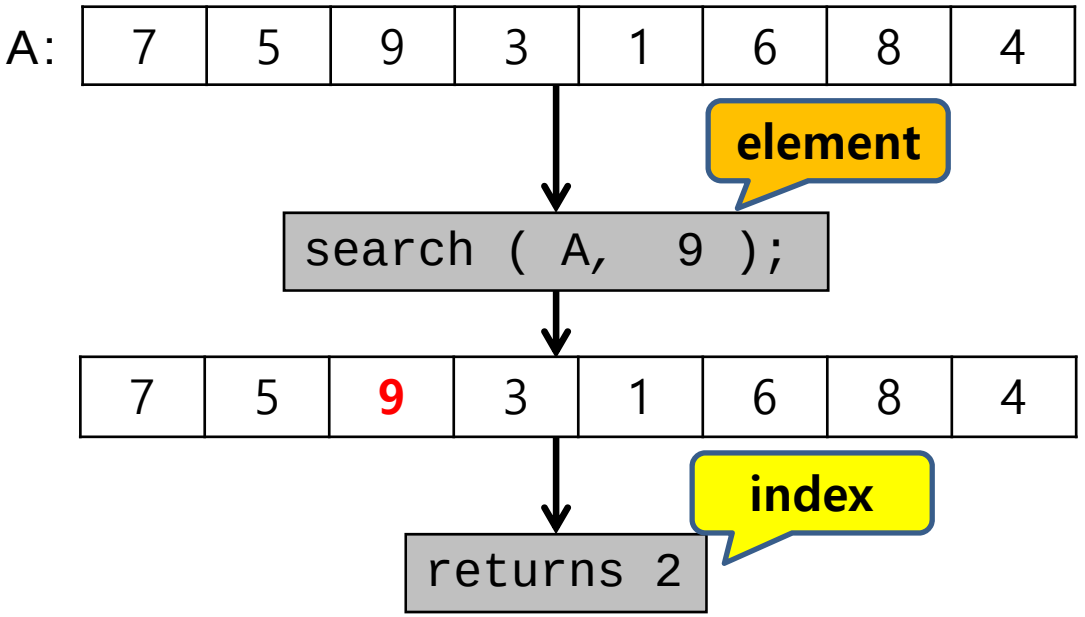
  - Sorted array: <1, 3, 4, 5, 6, 7, 8, 9>

```
search( A, x )
```

4. Search

① Definition of search

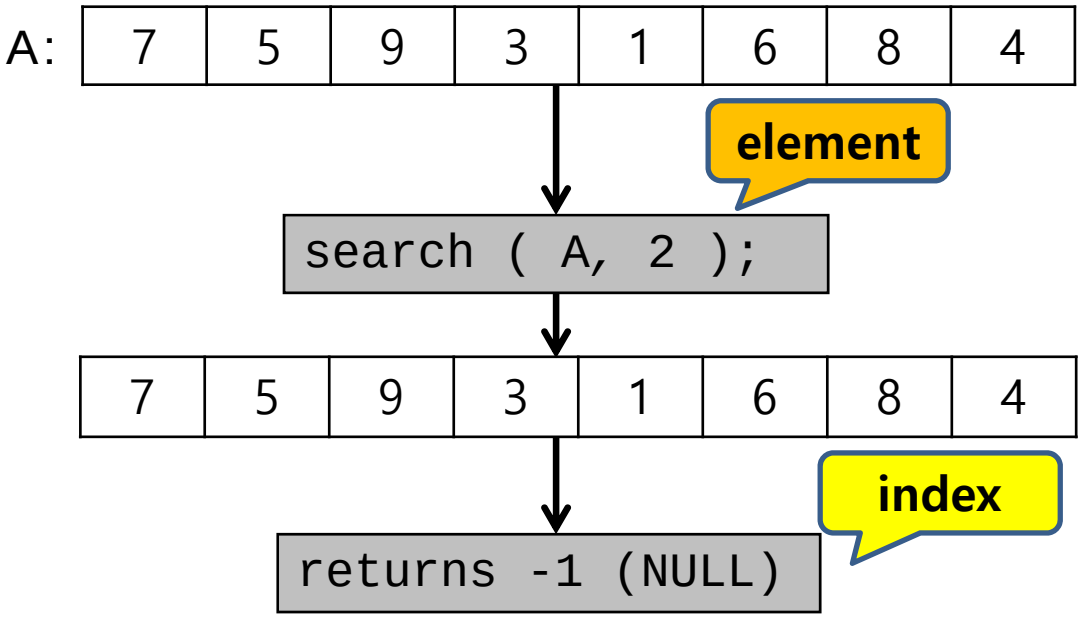
- Example of Successful search
  - The key element is in the array



4. Search

① Definition of search

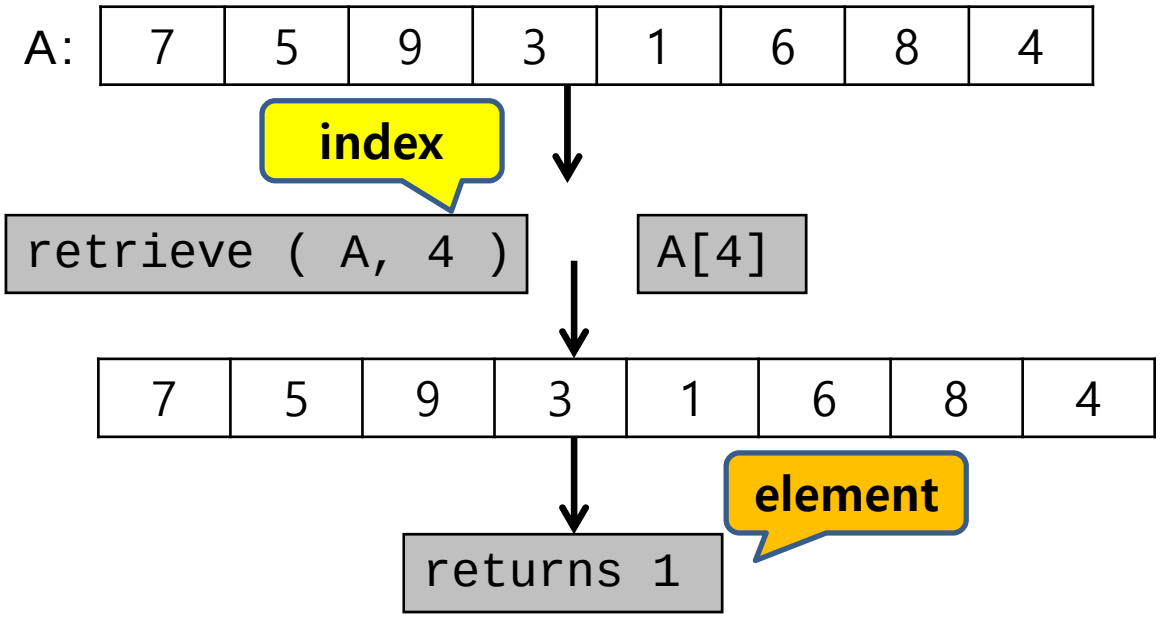
- Example of Failed search
  - The key element is not in the array



4. Search

① Definition of search

- What is the difference of *search* & *retrieve*?
  - retrieve: get an **element** from an **index**
  - search: get an **index** from an **element**



4. Search

① Definition of search

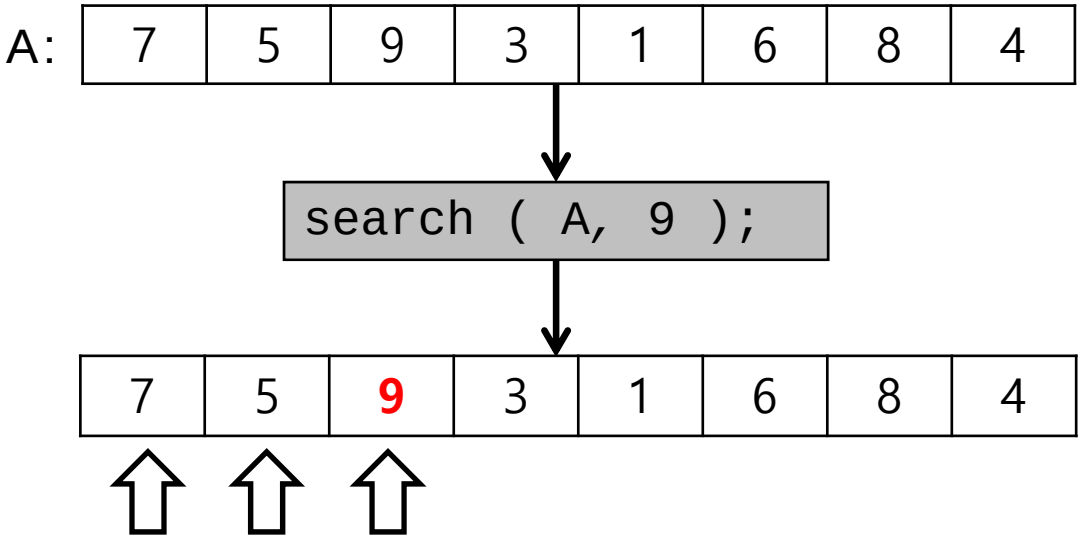
- **linear search** in an unsorted array
  - exhaustive search / sequential search
  - visit the elements in the array from the first position until we find the key element

- **binary search** in an sorted array
  - divide & conquer
  - select the middle of the array and divide the array by half

4. Search

② linear search in an unsorted array

- visit the elements in the array from the first position until we find the key element



4. Search

② linear search in an unsorted array

```
index linear_search ( Array A, elt x )
{
    for ( int i = 0; i < n; i++ )
        if ( A[i] == x )
            return i;

    return -1; //      NULL
}
```

4. Search

②linear search in an unsorted array

- C-style with local declaration

```
int linear_search ( int *A, int count, int x )
{
    for ( int i = 0; i < count; i++ )
        if ( A[i] == x )
            return i;

    return -1; //      NULL
}
```

```
    idx = linear_search ( A, count, x );
```

4. Search

②linear search in an unsorted array

- C-style with extern declaration

```
int A[MAX_SIZE];
int count;

int linear_search ( int x )
{
    for ( int i = 0; i < count; i++ )
        if ( A[i] == x )
            return i;

    return -1; //      NULL
}
```

```
    idx = linear_search ( x );
```

4. Search

②linear search in an unsorted array

- C++-style

```
int myArray::linear_search ( int x )
{
    for ( int i = 0; i < count; i++ )
        if ( arr[i] == x )
            return i;

    return -1; //      NULL
}
```

```
    idx = A.linear_search ( x );
```

4. Search

② linear search in an unsorted array

- What is the time complexity of linear search?

  O(n)

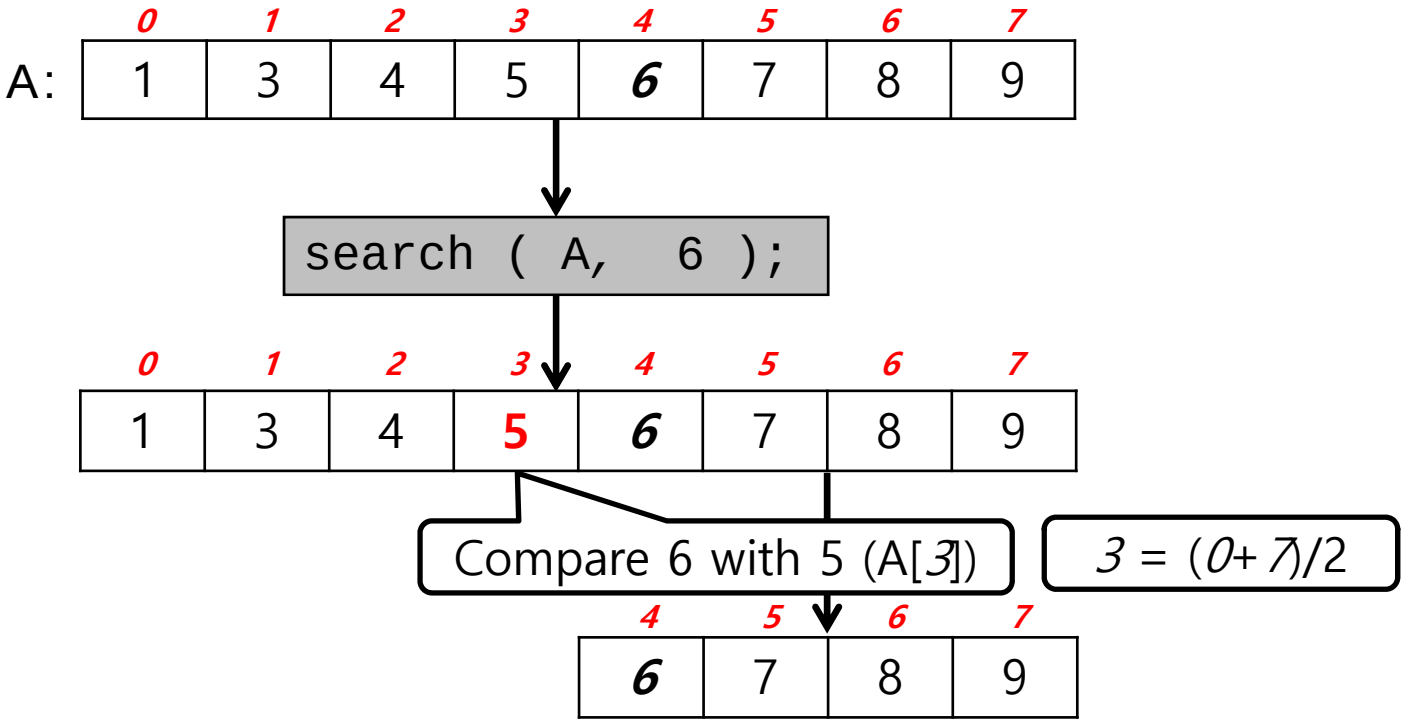
```
index linear_search ( Array A, elt x )
{
    for ( int i = 0; i < n; i++ )
        if ( A[i] == x )
            return i;

    return -1; //     NULL
}
```

4. Search

③ binary search in an sorted array

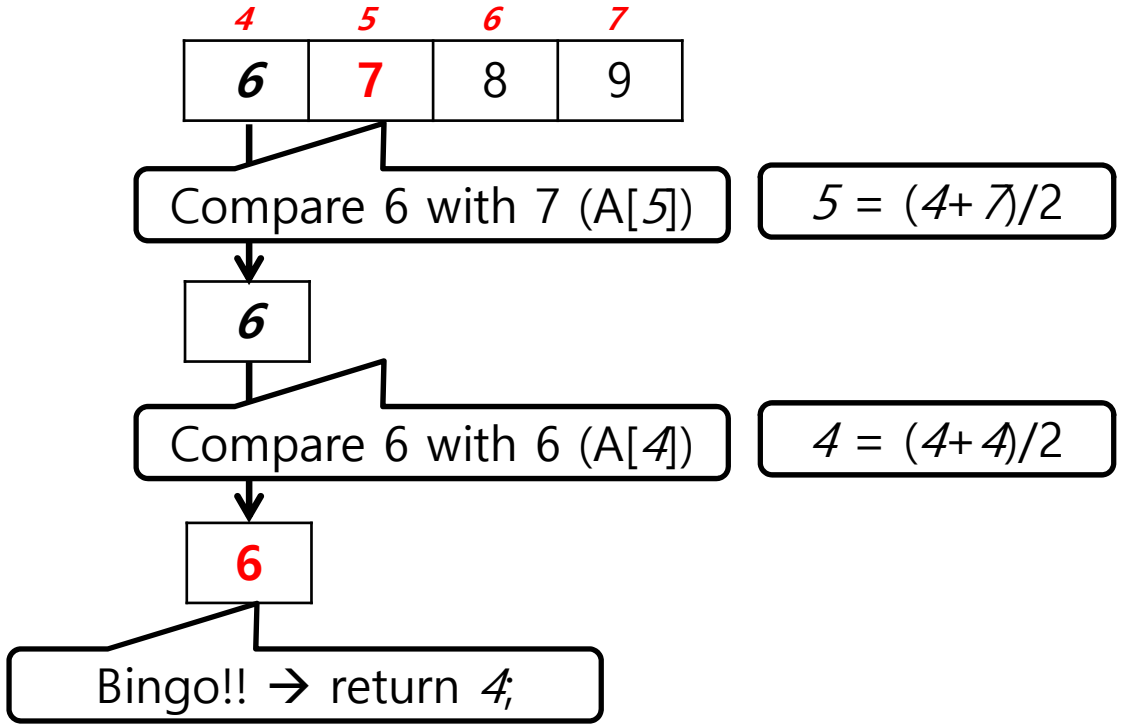
- select the middle of the array and divide the array by half



4. Search

③ binary search in an sorted array

- select the middle of the array and divide the array by half



4. Search

③ binary search in an sorted array

```
index binary_search ( Array A, index s, index e, elt x )
// s: first index, e: last index
{
    if ( s == e )
        return (A[s] == x) ? s : -1;

    int mid = (s + e)/2;
    if (x == A[mid])
        return mid;
    else if ( x < A[mid] )
        return binary_search ( A, s, mid-1, x );
    else
        return binary_search ( A, mid+1, e, x );
}
```

4. Search

③ binary search in an sorted array

- C-style with local declaration

```
int binary_search ( int *A, int s, int e, int x )
// s: first index, e: last index
{
    if ( s == e )
        return (A[s] == x) ? s : -1;

    int mid = (s + e)/2;
    if (x == A[mid])
        return mid;
    else if ( x < A[mid] )
        return binary_search ( A, s, mid-1, x );
    else
        return binary_search ( A, mid+1, e, x );
}
```

```
    binary_search ( A, 0, count-1, 10 );
```

4. Search

③ binary search in an sorted array

- C-style with extern declaration

```
int binary_search ( int s, int e, int x )
// s: first index, e: last index
{
    if ( s == e )
        return (A[s] == x) ? s : -1;

    int mid = (s + e)/2;
    if (x == A[mid])
        return mid;
    else if ( x < A[mid] )
        return binary_search ( s, mid-1, x );
    else
        return binary_search ( mid+1, e, x );
}
```

```
    binary_search ( 0, count-1, 10 );
```

4. Search

③ binary search in an sorted array

- C++-style

```
int myArray::binary_search ( int s, int e, int x )
// s: first index, e: last index
{
    if ( s == e )
        return (arr[s] == x) ? s : -1;

    int mid = (s + e)/2;
    if (x == arr[mid])
        return mid;
    else if ( x < arr[mid] )
        return binary_search ( s, mid-1, x );
    else
        return binary_search ( mid+1, e, x );
}
```

```
    A.binary_search ( 0, count-1, 10 );
```

4. Search

③ binary search in an sorted array

- What is the time complexity of binary search?

  O (log n)

T(n) ← Search on n data (n = e – s + 1)

T(n) = T(n/2) + 1 ← Telescoping

3. Operations of array

<table>
<tr><th>Operation</th><th>Sorted array
A=[2, 5, 7, 9, 10]</th><th>Unsorted array
A=[5, 2, 7, 10, 9]</th></tr>
<tr><td rowspan="2">Search</td><td>linear_search (A, x)</td><td rowspan="2">linear_search (A, x)</td></tr>
<tr><td>binary_search (A, x)</td></tr>
<tr><td rowspan="3">Insert</td><td rowspan="3">(3) insert_by_value (A, x)
(A, 8) → [2, 5, 7, 8, 9, 10]

Insert ( ), Insert_by_index ( ) & Store ( ) are not allowed</td><td>(1) insert (A, x)
(A, 8) → [5, 2, 7, 10, 9, 8]</td></tr>
<tr><td>(2) insert_by_index (A, i, x)
(A, 3, 8) → [5, 2, 7, 8, 10, 9]</td></tr>
<tr><td>store (A, i, x)
(A, 3, 8) → [5, 2, 7, 8, 9]</td></tr>
<tr><td rowspan="2">Delete</td><td>delete_by_value (A, x)
(A, 5) → [2, 7, 9, 10]</td><td>delete_by_value (A, x)
(A, 5) → [2, 7, 10, 9]</td></tr>
<tr><td>delete_by_index (A, i)
(A, 3) → [2, 5, 7, 10]</td><td>delete_by_index (A, i)
(A, 3) → [5, 2, 7, 9]</td></tr>
</table>

5. Insert

① Definition of insert

- Add a new element to an array
- Different approaches by the type of an array
  - Unsorted array: <BAT, CAT, FAT, EAT, JAT, LAT>

    (1) do not specify the insert position → insert it last

    `insert ( A, x )`

    (2) specify the insert position (insert_by_index)

    `Insert_by_index ( A, i, x )`

  - Sorted array: <BAT, CAT, EAT, FAT, JAT, LAT>

    (3) do not specify the position (insert_by_value)

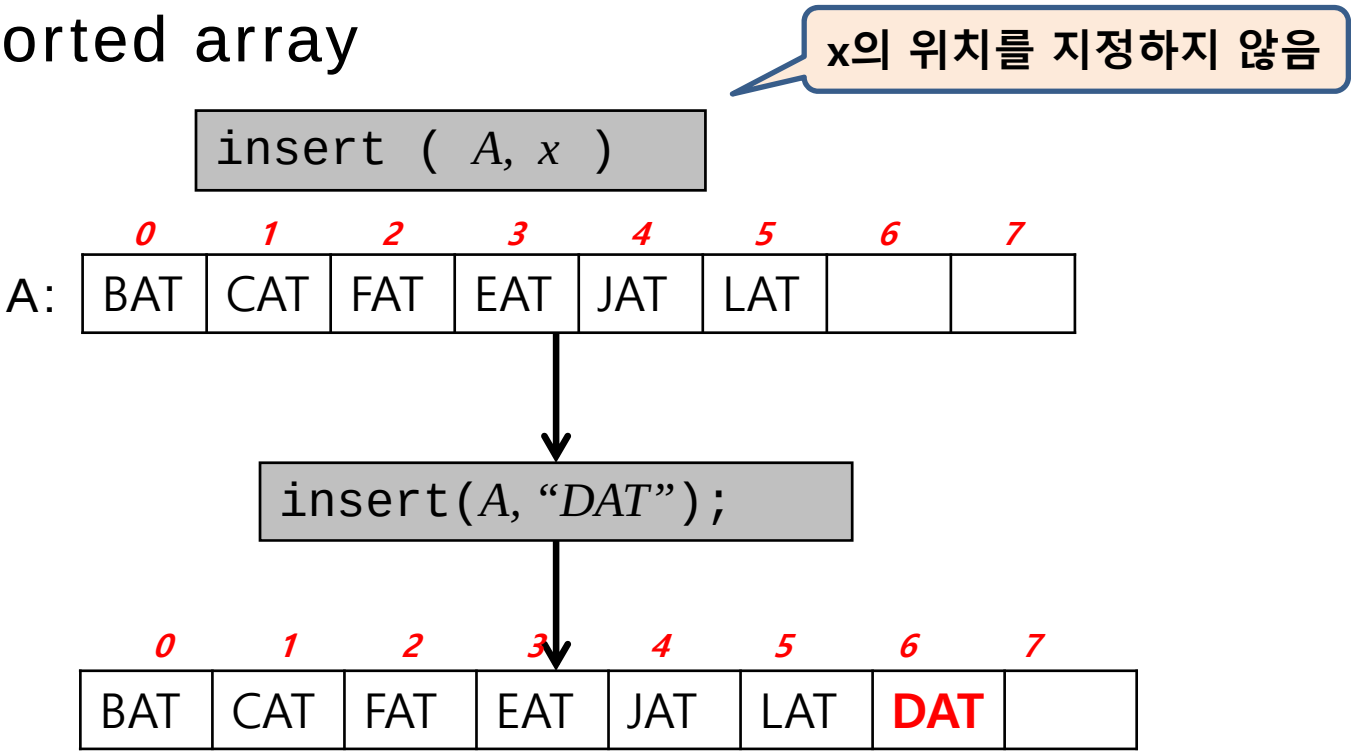
    `Insert_by_value ( A, x )`

- What is the difference of *insert* and *store*?

5. Insert

① Definition of insert (1) → Insert

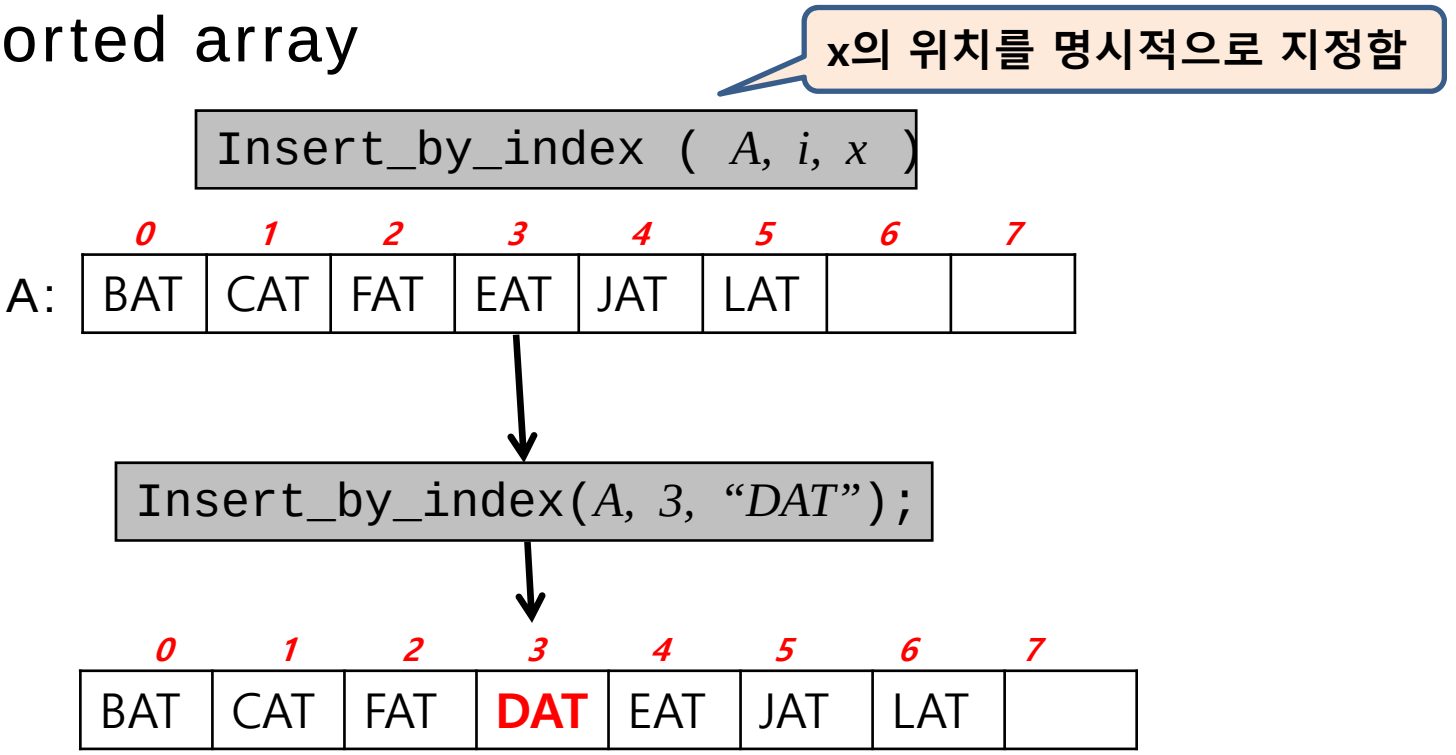
- Unsorted array



5. Insert

① Definition of insert (2) → Insert_by_index

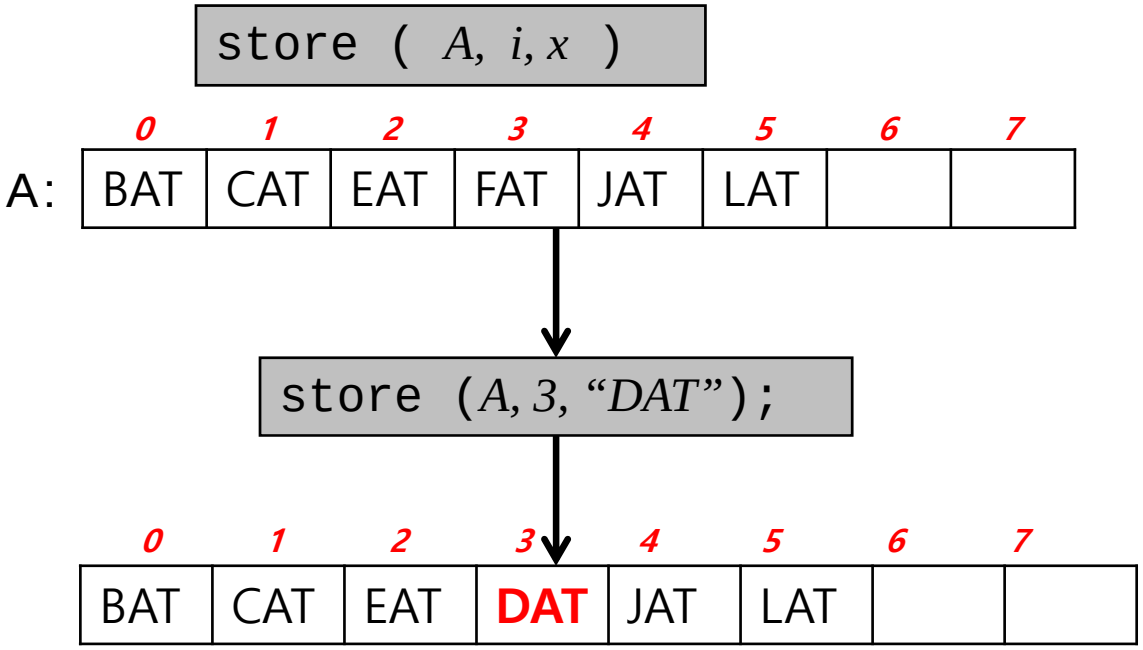
- Unsorted array



5. Insert

① Definition of insert

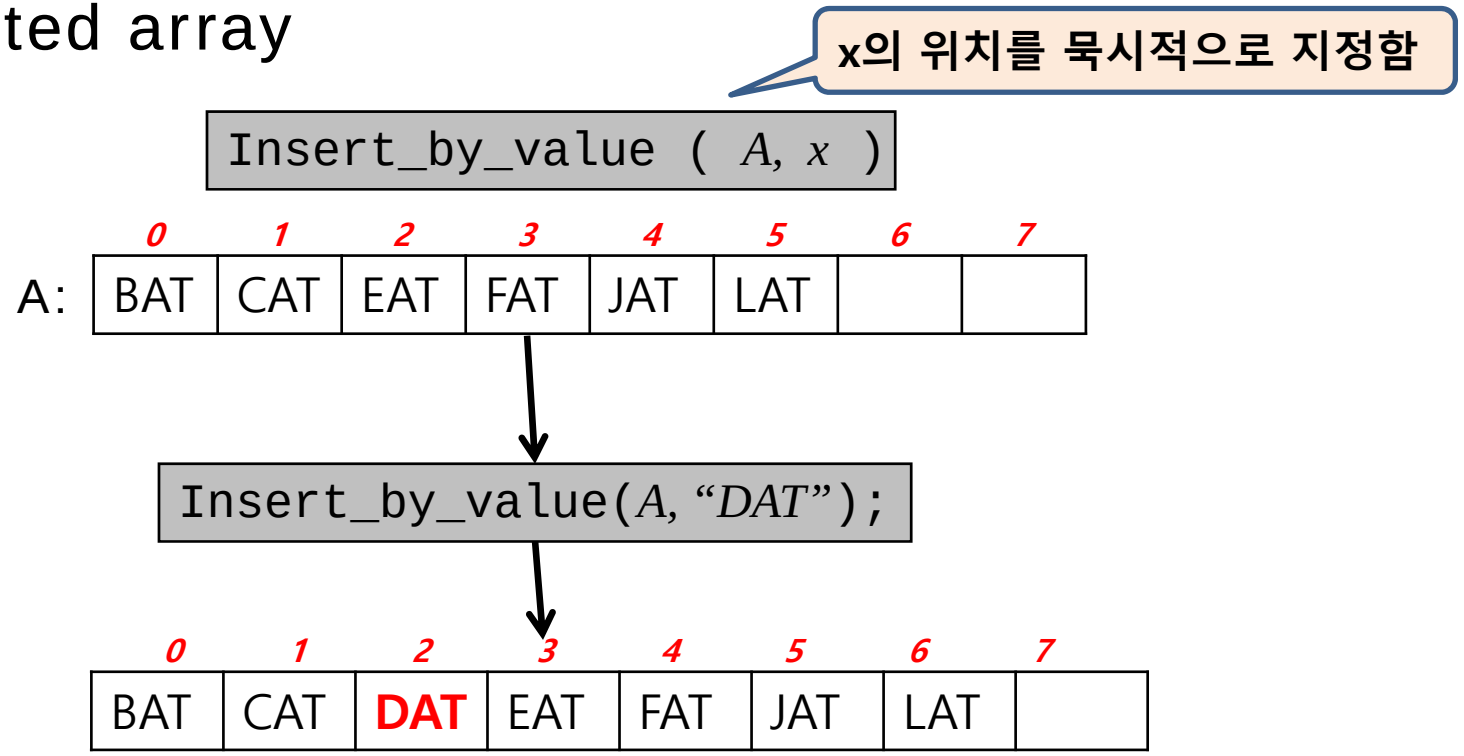
- compare to the *store* operation



5. Insert

① Definition of insert (3) → Insert_by_value

- Sorted array



5. Insert

② insert () in unsorted array

```
Array insert ( Array A, elt x )
{
// 0. Degenerate case
    if ( is_full ( A ) )
        resize ( A );

    A[n-1] = x;
    n++;
    return A;
}
```

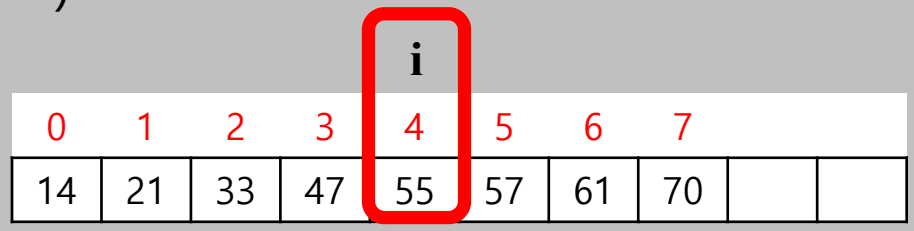
5. Insert

② insert_by_index () in unsorted array

```
Array insert ( Array A, index i,  elt x )
{
// 0. Degenerate case
    if ( i >= n )      return (“Error”);
    if ( is_full ( A ) )
        resize ( A );

// 1. Move the elements from i (→)
    for ( j = n-1; j >= i; j-- )
        A[j+1] = A[j];
// 2. Store x at A[i]
    A[i] = x;

    n++;
    return A;
}
```



5. Insert

② insert_by_index () in unsorted array

```
int insert_by_index ( int *A, int count, int i,  int x )
{
// 0. Degenerate case
    if ( i >= count )  return (“Error”);
    if ( is_full ( A ) )
        resize ( A );

// 1. Move the elements from i (→)
    for ( int j = count-1; j >= i; j-- )
        A[j+1] = A[j];
// 2. Store x at A[i]
    A[i] = x;

    return count + 1;
}
```

```
    count = insert_by_value ( A, count, 5, 4 );
```

5. Insert

② insert_by_index () in unsorted array

```
void insert_by_index ( int *A, int *count, int i,  int x )
{
// 0. Degenerate case
    if ( i >= *count ) return (“Error”);
    if ( is_full ( A ) )
        resize ( A );

// 1. Move the elements from i (→)
    for ( int j = *count - 1; j >= i; j-- )
        A[j+1] = A[j];
// 2. Store x at A[i]
    A[i] = x;

    (*count)++;
}
```

```
    insert_by_value ( A, &count, 5, 4 );
```

5. Insert

② insert_by_index () in unsorted array

```
void insert_by_index ( int i, int x )
{
// 0. Degenerate case
    if ( i >= count ) return (“Error”);
    if ( is_full ( A ) )
        resize ( A );

// 1. Move the elements from i (→)
    for ( int j = count-1; j >= i; j-- )
        A[j+1] = A[j];
// 2. Store x at A[i]
    A[i] = x;

    count++;
}
```

```
    insert_by_value ( 5, 4 );
```

5. Insert

② insert_by_index () in unsorted array

```
void myArray::insert_by_index ( int i, int x )
{
// 0. Degenerate case
    if ( i >= count )  return (“Error”);
    if ( is_full ( ) )
        resize ( );

// 1. Move the elements from i (→)
    for ( int j = count-1; j >= i; j-- )
        arr[j+1] = arr[j];
// 2. Store x at A[i]
    arr[i] = x;

    count++;
}
```

```
    A.insert_by_value ( 5, 4 );
```

5. Insert (R)

③ insert_by_value () in sorted array

```
Array insert_by_value ( Array A,  elt x )
{
 // 0. Degenerate case
    if ( is_full ( A ) )
        resize ( A );

// 1. Locate the x's position
    for ( i = 0; i < n; i++ ) {
        if ( A[i] > x )
            break;
    }
// 2. Move the elements from i (→)
    for ( j = n-1; j >= i; j-- )
        A[j+1] = A[j];
// 3. Store x at A[i]
    A[i] = x;
    n++;
    return A;
}
```

5. Insert (R)

③ insert_by_value () in sorted array

```
int insert_by_value ( int *A, int count int x )
{
 // 0. Degenerate case
    if ( is_full ( A ) )
        resize ( A );

// 1. Locate the x's position
    for ( i = 0; i < count; i++ ) {
        if ( A[i] > x )
            break;
    }
// 2. Move the elements from i (→)
    for ( j = count-1; j >= i; j-- )
        A[j+1] = A[j];
// 3. Store x at A[i]
    A[i] = x;

    return count + 1;
}
```

5. Insert

③ insert_by_value () in sorted array

```
void insert_by_value ( int x )
{
 // 0. Degenerate case
    if ( is_full ( A ) )
        resize ( A );

// 1. Locate the x's position
    for ( i = 0; i < count; i++ ) {
        if ( A[i] > x )
            break;
    }
// 2. Move the elements from i (→)
    for ( j = count - 1; j >= i; j-- )
        A[j+1] = A[j];
// 3. Store x at A[i]
    A[i] = x;
}
```

5. Insert

③ insert_by_value () in sorted array

```
void myArray::insert_by_value ( int x )
{
 // 0. Degenerate case
    if ( is_full ( ) )
        resize ( );

// 1. Locate the x's position
    for ( i = 0; i < count; i++ ) {
        if ( arr[i] > x )
            break;
    }
// 2. Move the elements from i (→)
    for ( j = count - 1; j >= i; j-- )
        arr[j+1] = arr[j];
// 3. Store x at A[i]
    arr[i] = x;
}
```

3. Operations of array

Operation	Sorted array A=[2, 5, 7, 9, 10]	Unsorted array A=[5, 2, 7, 10, 9]
Search	linear_search (A, x)	linear_search (A, x)
	binary_search (A, x)	
Insert	insert_by_value (A, x) **(A, 8) → [2, 5, 7, 8, 9, 10]**	insert (A, x) **(A, 8) → [5, 2, 7, 10, 9, 8]**
		insert_by_index (A, i, x) **(A, 3, 8) → [5, 2, 7, 8, 10, 9]**
		store (A, i, x) **(A, 3, 8) → [5, 2, 7, 8, 9]**
Delete	delete_by_value (A, x) **(A, 5) → [2, 7, 9, 10]**	delete_by_value (A, x) **(A, 5) → [2, 7, 10, 9]**
	delete_by_index (A, i) **(A, 3) → [2, 5, 7, 10]**	delete_by_index (A, i) **(A, 3) → [5, 2, 7, 9]**

6. Delete

① Definition of delete

- Remove an element from an array
- Constraint
  - An array does not have an empty slot

- Two types of delete
  - delete by index

```
delete_by_index ( A, i )
```

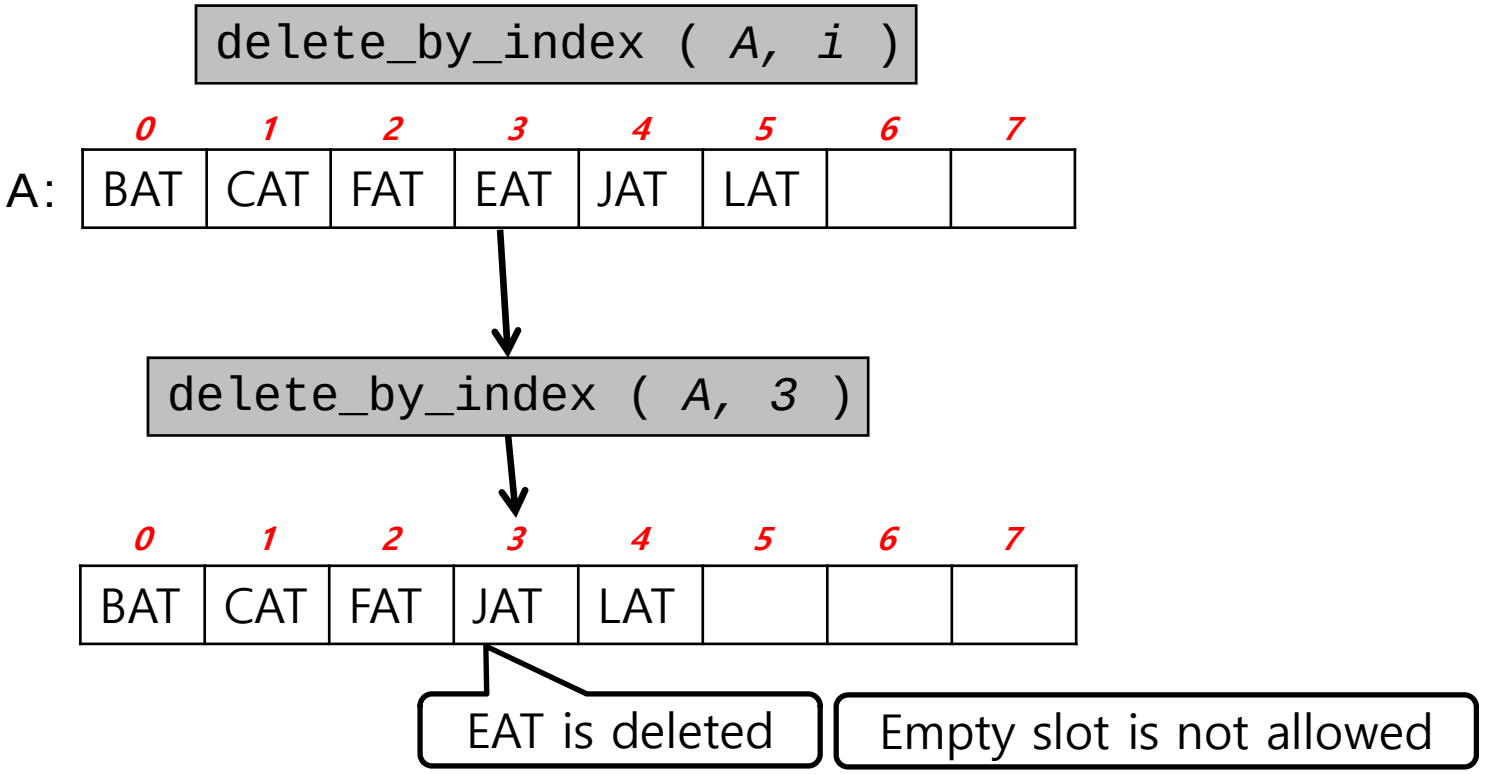
  - delete by value

```
delete_by_value ( A, x )
```

6. Delete

① Definition of delete

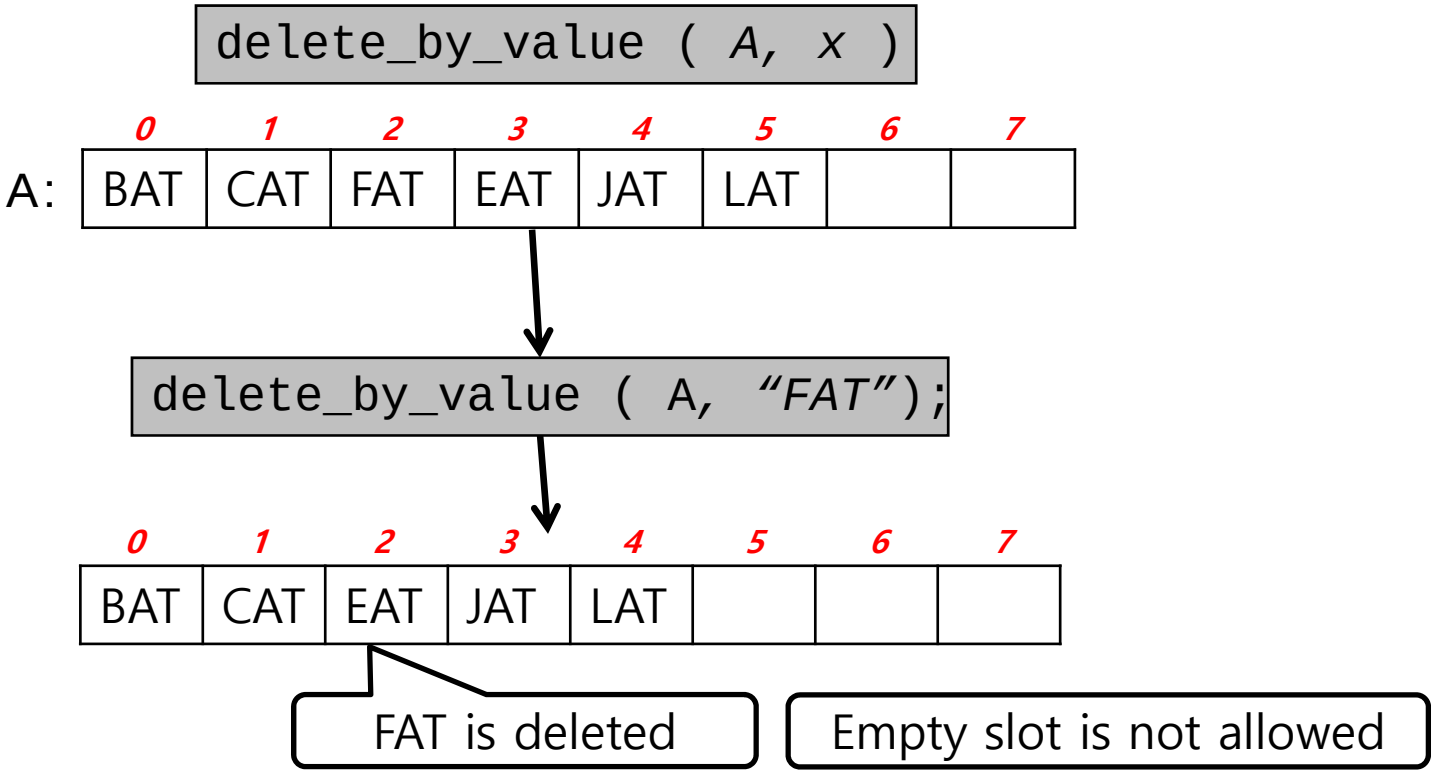
- delete by index



6. Delete

① Definition of delete

- delete by value



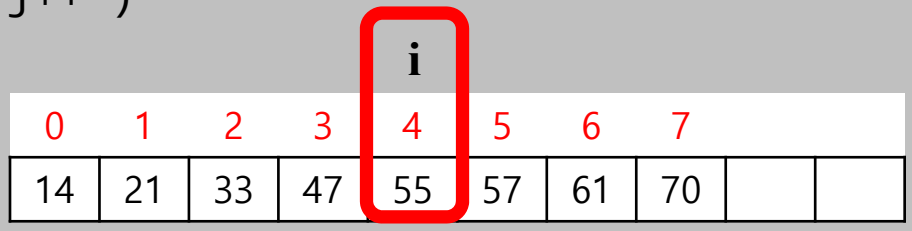
6. Delete

② delete by index

```
Array delete_by_index ( Array A, index i )
{
// 0. Degenerate case
    if ( is_empty ( A ) )
        return (“Error”);
    if ( i > n )
        return (“Error”);

// 1. Move the elements after index = i to their front positions (←)
    for ( int j = i; j < n-1; j++ )
        A[j] = A[j+1];
    n--;

    return A;
}
```



6. Delete

② delete by index

```
int delete_by_index ( int *A, int count, int i )
{
// 0. Degenerate case
    if ( is_empty ( A ) )
        return (“Error”);
    if ( i > count )
        return (“Error”);

// 1. Move the elements after index = i to their front positions (←)
    for ( int j = i; j < count-1; j++ )
        A[j] = A[j+1];

    return count - 1;
}
```

```
    count = delete_by_index ( A, count, 4 );
```

6. Delete

② delete by index

```
void delete_by_index ( int i )
{
// 0. Degenerate case
    if ( is_empty ( A ) )
       return (“Error”);
    if ( i > count )
       return (“Error”);

// 1. Move the elements after index = i to their front positions (←)
    for ( int j = i; j < count-1; j++ )
       A[j] = A[j+1];

    count--;
}
```

```
    delete_by_index ( 4 );
```

6. Delete

② delete by index

```
void myArray::delete_by_index ( int i )
{
// 0. Degenerate case
    if ( is_empty ( ) )
        return (“Error”);
    if ( i > count )
        return (“Error”);

// 1. Move the elements after index = i to their front positions (←)
    for ( int j = i; j < count-1; j++ )
        arr[j] = arr[j+1];

    count--;
}
```

```
    A.delete_by_index ( 4 );
```

6. Delete (R)

③ delete by value

```
Array delete_by_value ( Array A, elt x )
{
// 0. Degenerate case
    if ( is_empty ( A ) )
        return (“Error”);

// 1. Find the index to delete
    for ( int i = 0; i < n; i++ ) {
        if ( A[i] == x )
            break;
    }
    if ( i == n )  return;
// 2. Move the elements after index = i to their rear positions (←)
    for ( int j = i; j < n-1; j++ )
        A[j] = A[j+1];

    n--;

    return A;
}
```

6. Delete (R)

③ delete by value

```
int delete_by_value ( int *A, int count, int x )
{
// 0. Degenerate case
    if ( is_empty ( A ) )
        return (“Error”);

// 1. Find the index to delete
    for ( int i = 0; i < count; i++ ) {
        if ( A[i] == x )
            break;
    }
    if ( i == count )  return;
// 2. Move the elements after index = i to their rear positions (←)
    for ( int j = i; j < count-1; j++ )
        A[j] = A[j+1];

    return count - 1;
}
```

6. Delete (R)

③ delete by value

```
void delete_by_value ( int x )
{
// 0. Degenerate case
    if ( is_empty ( A ) )
        return (“Error”);

// 1. Find the index to delete
    for ( int i = 0; i < count; i++ ) {
        if ( A[i] == x )
            break;
    }
    if ( i == count )  return;
// 2. Move the elements after index = i to their rear positions (←)
    for ( int j = i; j < count-1; j++ )
        A[j] = A[j+1];

    count--;
}
```

6. Delete (R)

③ delete by value

```
void myArray::delete_by_value ( int x )
{
// 0. Degenerate case
    if ( is_empty ( ) )
        return (“Error”);

// 1. Find the index to delete
    for ( int i = 0; i < count; i++ ) {
        if ( arr[i] == x )
            break;
    }
    if ( i == count )  return;
// 2. Move the elements after index = i to their rear positions (←)
    for ( int j = i; j < count-1; j++ )
        arr[j] = arr[j+1];

    count--;
}
```

7. Performance of the operations

- Time complexity

Operation	Type of array	Function	Time complexity
Search	Unsorted array	linear_search (A, x)	O (n)
	Sorted array	binary_search (A, x)	**O (log n)**
Insert	Unsorted array	Insert_by_index (A, i, x)	O (n)
		insert (A, x)	**O (1)**
	Sorted array	Insert_by_value (A, x)	O (n)
Delete	Don't care	delete_by_index (A, i)	O (n)
		delete_by_value (A, x)	O (n)

8. Implementation tip

- comparison

```
#define MAX_SIZE 100//     MAX, SIZE 등 다양한 이름으로 사용
{
    int count = 0;  //     cnt, n 등 다양한 이름으로 사용
    int A[MAX_SIZE];
}
```

```
    search ( A, count, 10 );
    count = insert ( A, count, 10 );
    count = delete ( A, count, 10 );
```

```
    search ( A, count, 10 );
    insert ( A, &count, 10 );
    delete ( A, &count, 10 );
```

8. Implementation tip

- comparison

```
#define MAX_SIZE 100//     MAX, SIZE 등 다양한 이름으로 사용
int count = 0;      //     cnt, n 등 다양한 이름으로 사용
int A[MAX_SIZE];
{

}
```

```
    search ( 10 );
    insert ( 10 );
    delete ( 10 );
```

8. Implementation tip

- comparison

```
class myArray {
    int MAX_SIZE;
    int count;
    int *arr;
};
{
    myArray A;
}
```

```
    A.search ( 10 );
    A.insert ( 10 );
    A.delete ( 10 );
```

3. List

1. Definition of list
2. Definition of array
3. Operations of array
4. Search
5. Insert
6. Delete
7. Performance analysis
8. Implementation tip

Contents