CSE 203: Data Structures and Algorithms-I

Arrays: Memory Mapping, Linear and Binary Search, Linear Time Sorting (Counting Sort)

Arrays

- An array is an indexed sequence of components
  - The components of an array are all of the same type
- Typically, the array occupies sequential storage locations
- Array is a static data structure, that is, the length of the array is determined when the array is created, and cannot be changed
- Each component of the array has a fixed, unique index
  - Indices range from a lower bound to an upper bound
- Any component of the array can be inspected or updated by using its index
  - This is an efficient operation: $O(1)$ = constant time

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
7	6	11	17	3	15	5	19	30	14

Representation of Arrays in Memory

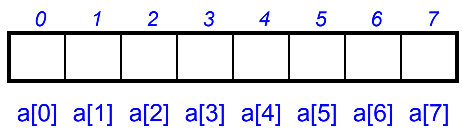
- **Linear (1 D) Arrays:**

  A 1-dimensional array a is declared as:

  int a[8];

  The elements of the array a may be shown as

  a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]



Representation of Arrays in Memory

- **2 D Arrays:**

A 2-dimensional array a is declared as:

int a[3][4];

The elements of the array a may be shown as a table

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

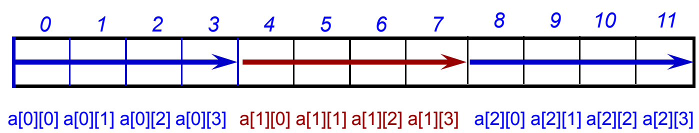
In which order are the elements stored?

- Row major order (C, C++, Java support it)
- Column major order (Fortran supports it)

Representation of Arrays in Memory

Row Major Order: the array is stored as a sequence of 1-D arrays consisting of rows

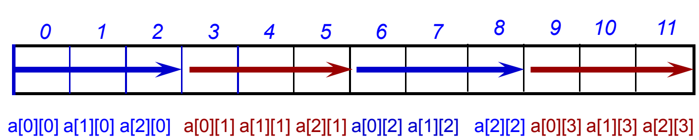
a[0][0] a[0][1] a[0][2] a[0][3] → row 0
a[1][0] a[1][1] a[1][2] a[1][3] → row 1
a[2][0] a[2][1] a[2][2] a[2][3] → row 2



Representation of Arrays in Memory

Column Major Order: The array is stored as a sequence of arrays consisting of columns instead of rows

column 0	column 1	column 2	column 3
a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]



Representation of Arrays in Memory: Parameters

- **Base Address (b):** The memory address of the first byte of the first array component.
- **Component Length (L):** The memory required to store one component of an array.
- **Upper and Lower Bounds (l_i, u_i):** Each index type has a smallest value and a largest value.
- **Dimension**

Representation of Arrays in Memory

- **Array Mapping Function (AMF)**
  - AMF converts index value to component address
- **Linear (1D) Arrays:**

  a : array $[l_1 .. u_1]$ of element_type

  $$\text{Then addr}(a[i]) = b + (i - l_1) \times L$$
  $$= c_0 + c_1 \times i$$

  Therefore, the time for calculating the address of an element is same for any value of i.

Representation of Arrays in Memory

- **Array Mapping Function (AMF): 1D Arrays**

  a : array $[l_1 .. u_1]$ of element_type

  Then addr($a[i]$) $= b + (i - l_1) \times L$

Representation of Arrays in Memory

- **Array Mapping Function (AMF): 2D Arrays**

  Row Major Order:

  a : array $[l_1 .. u_1, l_2 .. u_2]$ of element_type

  Then addr($a[i, j]$) $= b + (i - l_1)\times(u_2 - l_2 + 1)\times L + (j - l_2)\times L$

  $= c_0 + c_1 \times i + c_2 \times j$

  Therefore, the time for calculating the address of an element is same for any value of (i, j).

Example: Arrays in Memory

- **Array Mapping Function (AMF): 2D Arrays**

  Row Major Order:

  a : array $[l_1 \,..\, u_1, l_2 \,..\, u_2]$ of element_type

  Then addr($a[i, j]$) $= b + (i - l_1)\times(u_2 - l_2 + 1)\times L + (j - l_2)\times L$

Representation of Arrays in Memory

- **Array Mapping Function (AMF): 2D Arrays**

  Column Major Order:

  a : array $[l_1 \,..\, u_1, l_2 \,..\, u_2]$ of element_type

  Then addr($a[i, j]$) $= b + (j - l_2)\times(u_1 - l_1 + 1)\times L + (i - l_1)\times L$

  $= c_0 + c_1\times i + c_2\times j$

  Therefore, the time for calculating the address of an element is same for any value of (i, j).

Example: Arrays in Memory

- **Array Mapping Function (AMF)**: **2D Arrays**

  Column Major Order:

  a : array $[l_1 .. u_1, l_2 .. u_2]$ of element_type

  Then addr$(a[i, j]) = b + (j - l_2)\times(u_1 - l_1 + 1)\times L + (i - l_1)\times L$

Representation of Arrays in Memory

- **Array Mapping Function (AMF): 3D Arrays :**

  a : array $[l_1 .. u_1, l_2 .. u_2, l_3 .. u_3]$ of element_type

  Then addr$(a[i, j, k]) = b + (i - l_1)\times(u_2 - l_2 + 1)\times(u_3 - l_3 + 1)\times L +$
  $(j - l_2)\times(u_3 - l_3 + 1)\times L + (k - l_3)\times L$
  $= c_0 + c_1\times i + c_2\times j + c_3\times k$

  Therefore, the time for calculating the address of an element is same for any value of (i, j, k).

Summary on Arrays

- **Advantages:**
  - Array is a random access data structure.
  - Accessing an element by its index is very fast (constant time)
- **Disadvantages:**
  - Array is a static data structure, that is, the array size is fixed and can never be changed.
  - Insertion into arrays and deletion from arrays are very slow.

- An array is a suitable structure when
  - a lot of searching and retrieval are required.
  - a small number of insertions and deletions are required.

Searching Algorithms:

Linear Search, Binary Search

The Searching Problem

- The process of finding a particular element in an array is called searching. There two popular searching techniques:
  - *Linear search,* and
  - *Binary search*.
- The ***linear search*** compares each element in an unsorted array with the ***search key***.
  - Running time: $O(n)$
- Given a sorted array, ***Binary Search*** algorithm can be used to perform fast searching of a search key on the sorted array.
  - Running time: $O(\log n)$

Linear Search

- Each member of the array is visited until the search key is found.
- **Example:**

  Write a program to search for the search key entered by the user in the following array:

  (9, 4, 5, 1, 7, 78, 22, 15, 96, 45)

  You can use the linear search in this example.

Linear Search

```
/* This program is an example of the Linear Search*/
#include <stdio.h>
#define SIZE 10
int LinearSearch(int [], int);
int main() {
      int a[SIZE]= {9, 4, 5, 1, 7, 78, 22, 15, 96, 45};
      int key, pos;
      printf("Enter the Search Key\n");
      scanf("%d", &key);
      pos = LinearSearch(a, key);
      if(pos == –1)
            printf("The search key is not in the array\n");
      else
            printf("The search key %d is at location %d\n", key, pos);
      return 0;
}
```


Linear Search

```
int LinearSearch (int b[ ], int skey) {
      int i;
      for (i=0; i < SIZE; i++)
            if(b[i] == skey)
                  return i;
      return –1;
}
```

Binary Search

- Given a sorted array, ***Binary Search*** algorithm can be used to perform fast searching of a search key on the sorted array.
- The following program implements the binary search algorithm for the search key entered by the user in the following array:
  (3, 5, 9, 11, 15, 17, 22, 25, 37, 68)

Binary Search

```
#include <stdio.h>
#define SIZE 10
int BinarySearch(int [ ], int);
int main(){
      int a[SIZE]= {3, 5, 9, 11, 15, 17, 22, 25, 37, 68};
      int key, pos;
      printf("Enter the Search Key\n");
      scanf("%d",&key);
      pos = BinarySearch(a, key);
      if(pos == -1)
          printf("The search key is not in the array\n");
      else
          printf("The search key %d is at location %d\n", key, pos);
      return 0;
}
```

Binary Search

```
int BinarySearch (int A[], int skey){
    int low=0, high=SIZE–1, middle;
    while(low <= high){
        middle = (low+high)/2;
        if (skey == A[middle])
            return middle;
        else if(skey <A[middle])
            high = middle – 1;
        else
            low = middle + 1;
    }
    return –1;
}
```


Linear-Time Sorting Algorithms:

Counting Sort

Sorting In Linear Time

- Counting sort
  - No comparisons between elements!
  - ***But***...depends on assumption about the numbers being sorted
    - We assume numbers are in the range 1, …, k
  - The algorithm:
    - Input: A[1..n], where A[i] $\in$ {1, 2, 3, …, k}
    - Output: B[1..n], sorted (notice: not sorting in place)
    - Also: Array C[1..k] for auxiliary storage

Counting Sort

```
COUNTING-SORT(A, B, k)
for i ← 0 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
▷ C[i] now contains the number of elements equal to i.
for i ← 1 to k
    do C[i] ← C[i] + C[i − 1]
▷ C[i] now contains the number of elements less than or equal to i.
for j ← length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] − 1
```

COUNTING-SORT assumes that each of the input elements is an integer in the range 0 to k, inclusive.

A[1..n] is the input array, B[1..n] is the output array, C[0..k] is a temporary working array.

Counting Sort

```
COUNTING-SORT(A, B, k)
    for i ← 0 to k
        do C[i] ← 0
    for j ← 1 to length[A]
        do C[A[j]] ← C[A[j]] + 1
    ▷ C[i] now contains the number of elements equal to i.
    for i ← 1 to k
        do C[i] ← C[i] + C[i − 1]
    ▷ C[i] now contains the number of elements less than or equal to i.
    for j ← length[A] downto 1
        do B[C[A[j]]] ← A[j]
           C[A[j]] ← C[A[j]] − 1
```

Takes time O(k)

Takes time O(n)

What will be the running time?

- Total time: $O(n + k)$
  - Usually, $k = O(n)$
  - Thus counting sort runs in $O(n)$ time

Counting Sort: Example

```
    for i ← 0 to k        ← 5   } O(k)
        do C[i] ← 0             }
    for j ← 1 to length[A] ← n  } O(n)
        do C[A[j]] ← C[A[j]] + 1 }
```

```
    for i ← 1 to k                  } O(k)
        do C[i] ← C[i] + C[i − 1]   }
    for j ← length[A] downto 1      ← n
        do B[C[A[j]]] ← A[j]        O(n)
           C[A[j]] ← C[A[j]] − 1
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

n

	0	1	2	3	4	5
C	~~0~~ 2	0	~~0~~ 2	~~0~~ 3	0	~~0~~ 1

k

	0	1	2	3	4	5
C	~~2~~ 0	2	4	~~7~~ 6	7	~~8~~ 7

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

$2O(k) + 2O(n)$
$= O(n+k)$ ←
$= O(n+n)$
$= O(n)$ ✓

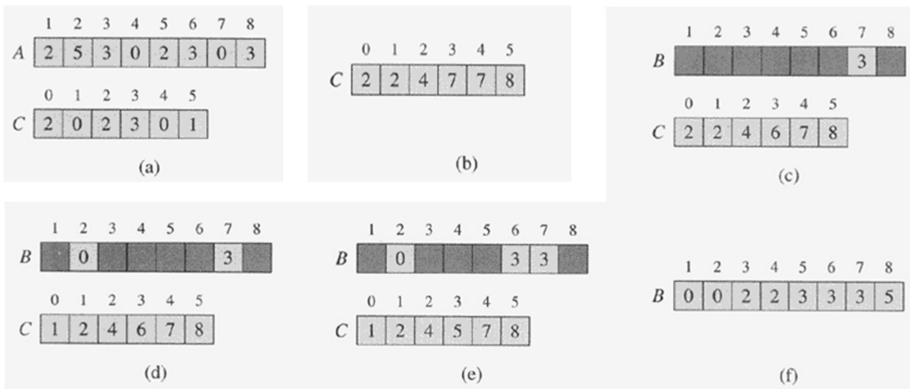
if $k = O(n)$

Counting Sort: Example

```
for i ← 0 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
```

```
for i ← 1 to k
    do C[i] ← C[i] + C[i − 1]
```

```
for j ← length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] − 1
```



Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU

Counting Sort

- *Why don't we always use counting sort?*
- Because it depends on range *k* of elements

- *Could we use counting sort to sort 32 bit integers? Why or why not?*
- Answer: No, *k* is too large (2^{32} = 4,294,967,296)

Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU