Programming Examples Using Arrays

Pei-yih Ting

Search and Sort an Array

- Two common problems in processing arrays
  - **Searching** an array to determine the location of a particular value.
  - **Sorting** an array to rearrange the array elements in numerical order.
- Examples
  - Search an array of student exam scores to determine which student, if any, got a particular score.
  - Rearrange the array elements in increasing (decreasing) order by score.
- Algorithm for searching over a sorted array is much more efficient than over an unsorted array.

Algorithm of Linear Search

(Sequential Search)

1. Assume the **target** has not been found.
2. Start with the initial array element.
3. Repeat while the target is not found and there are more array elements
   - 3.1 if the current element matches the target
     - 3.1.1 Set a flag to indicate that the target has been found

     else
     - 3.1.2 Advance to the next array element
4. if the target was found
   - 4.1 Return the target index as the search result

   else
   - 4.2 Return -1 as the search result

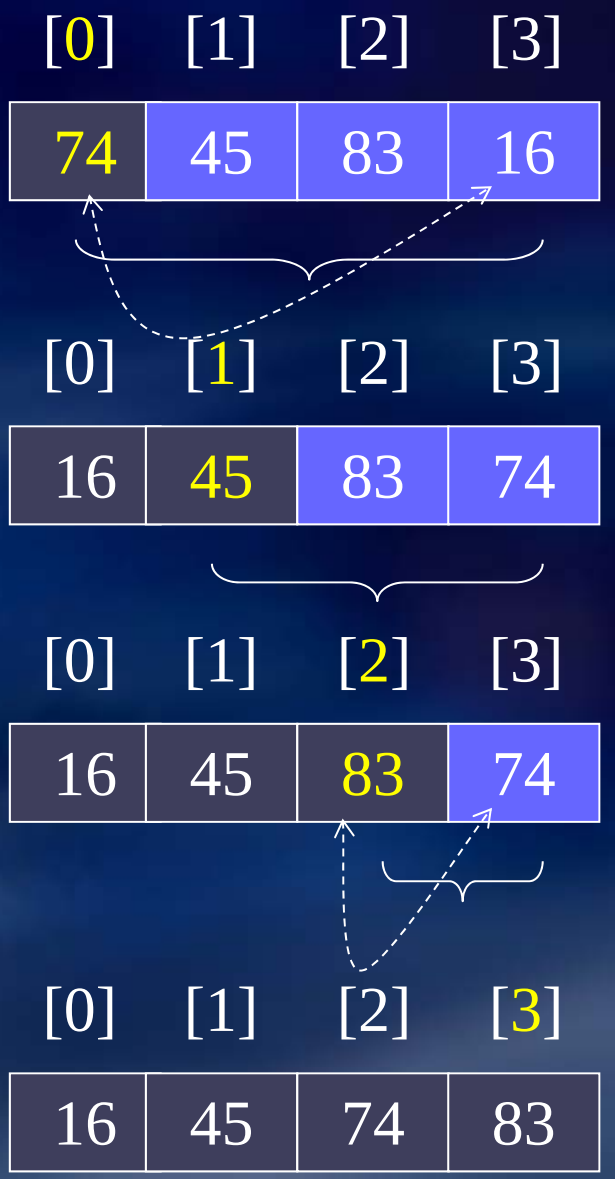
Search an Array

```
01 int search(const int array[],  /* input - array to search */
02                int      target,      /* input - value searched for */
03                int      n) {         /* input - number of elements to search */
04    int i,
05        found = 0,     /*  whether or not target has been found */
06        where;         /*  index where target found or NOT_FOUND */
07
08    /*  Compares each element to target */
09    i = 0;
10    while (!found && i < n) {
11       if (array[i] == target)
12          found = 1;
13       else
14          ++i;
15    }
16
17    /* Returns index of element matching target or NOT_FOUND */
18    if (found)
19       where = i;
20    else
21       where = NOT_FOUND;
22
23    return (where);
24 }
```

Sorting an Array

Selection sort is an intuitive sorting algorithm.

- Find the index of the smallest element in the array.
- Swap the smallest element with the first element.
- Repeat the above steps for the 2nd, 3rd, …, smallest elements.



Function select_sort

```
01 int get_min_range(int list[], int first, int last);
02 void select_sort(int list[],    /* input/output - array being sorted      */
03                  int n)         /* input - number of elements to sort     */
04 {
05       int fill,                 /* first element in unsorted subarray */
06           temp,                 /* temporary storage                  */
07           index_of_min;         /* subscript of next smallest element */
08
09       for (fill = 0; fill < n-1; ++fill) {
10          /* Find position of smallest element in the unsorted subarray */
11           index_of_min = get_min_range(list, fill, n-1);
12
13           /* Exchange elements at fill and index_of_min */
14           if (fill != index_of_min) {
15                temp = list[index_of_min];
16                list[index_of_min] = list[fill];
17                list[fill] = temp;
18           }
19       }
20 }
```

Computing Statistics

- Most common use of arrays is for storage of a collection of related data values.
- Once the values are stored, we can perform some simple statistical computations.

sum = x[0] + x[1] + ... + x[MAX_ITEM-1]

mean = sum / MAX_ITEM

sum_square = $x[0]^2 + x[1]^2 + ... + x[MAX\_ITEM-1]^2$

variance = sum_square / (MAX_ITEM-1) – $mean^2$

standard deviation = sqrt(variance)

histogram?

mode?

medium?

Computing Statistics (cont'd)

Figure 8.3

```
01 #include <stdio.h>
02 #include <math.h>
03 #define MAX_ITEM  8  /* maximum number of items in list of data    */
04 int
05 main(void)
06 {
07    double x[MAX_ITEM],   /* data list                              */
08           mean,          /* mean (average) of the data             */
09           st_dev,        /* standard deviation of the data         */
10           sum,           /* sum of the data                        */
11           sum_sqr;       /* sum of the squares of the data         */
12    int    i;
13
14    /* Gets the data   */
15    printf("Enter %d numbers separated by blanks or <return>s\n> ",
16           MAX_ITEM);
17    for  (i = 0;  i < MAX_ITEM;  ++i)
18       scanf("%lf", &x[i]);
```

```
19    /* Computes the sum and the sum of the squares of all data  */
20    sum = 0;
21    sum_sqr = 0;
22    for (i = 0; i < MAX_ITEM; ++i) {
23       sum += x[i];
24       sum_sqr += x[i] * x[i];
25 }
26
27    /* Computes and prints the mean and standard deviation     */
28    mean = sum / MAX_ITEM;
39    st_dev = sqrt(sum_sqr / MAX_ITEM - mean * mean);
30    printf("The mean is %.2f.\n", mean);
31    printf("The standard deviation is %.2f.\n", st_dev);
32
33    /* Displays the difference between each item and the mean   */
34    printf("\nTable of differences between data values and mean\n");
35    printf("Index      Item      Difference\n");
36    for (i = 0; i < MAX_ITEM; ++i)
37       printf("%3d%4c%9.2f%5c%9.2f\n", i, ' ', x[i], ' ', x[i] - mean);
38
39    return (0);
40 }
```

Computing Statistics (cont'd)

Enter 8 numbers separated by blanks or <return>s
> **16 12 6 8 2.5 12 14 -54.5**

The mean is **2.00**.
The standard deviation is **21.75**.

Table of differences between data values and mean

Index	Item	Difference
0	16.00	14.00
1	12.00	10.00
2	6.00	4.00
3	8.00	6.00
4	2.50	0.50
5	12.00	10.00
6	14.00	12.00
7	-54.50	-56.50

Matrix Operations

- Addition
  - Ex. A and B are both 3-by-5, C = A + B

$$\begin{pmatrix} 1 & 2 & 3 & 2 & 3 \\ 4 & 5 & 6 & 5 & 6 \\ 1 & 2 & 5 & 4 & 5 \end{pmatrix} + \begin{pmatrix} 7 & 2 & 3 & 2 & 6 \\ 4 & 1 & 0 & 3 & 2 \\ 1 & 4 & 4 & 2 & 2 \end{pmatrix} = \begin{pmatrix} 8 & 4 & 6 & 4 & 9 \\ 8 & 6 & 6 & 8 & 8 \\ 2 & 6 & 9 & 6 & 7 \end{pmatrix}$$

  - $C_{ij} = A_{ij} + B_{ij}$

```
int i, j, m=3, n=5;
double a_mat[3][5], b_mat[3][5];
double c_mat[3][5];
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        c_mat[i][j] = a_mat[i][j] + b_mat[i][j];
```

Matrix Operations (cont'd)

- Multiplication
  - Ex. A and B are both 3-by-5, $C = A B^T$

$$\begin{pmatrix} 1 & 2 & 3 & 2 & 3 \\ 4 & 5 & 6 & 5 & 6 \\ 1 & 2 & 5 & 4 & 5 \end{pmatrix} \begin{pmatrix} 7 & 4 & 1 \\ 2 & 1 & 4 \\ 3 & 0 & 4 \\ 2 & 3 & 2 \\ 6 & 2 & 2 \end{pmatrix} = \begin{pmatrix} 42 & 18 & 31 \\ 102 & 48 & 70 \\ 64 & 28 & 47 \end{pmatrix}$$

  - $C_{ij} = \sum_{k=1}^{5} A_{ik} B^T_{kj}$

```
int i, j, k, m=3, n=5;
double a_mat[3][5], b_mat[3][5];
double c_mat[3][3];
for (i=0; i<m; i++)
    for (j=0; j<m; j++)
        for (k=0, c_mat[i][j]=0; k<n; k++)
            c_mat[i][j] += a_mat[i][k] * b_mat[j][k];
```

Matrix Operations (cont'd)

- In-place Computation??

  $A, B \in R^{mxn}$, $A + B \rightarrow A$; $A, B \in R^{nxn}$, $A B^T \rightarrow A$

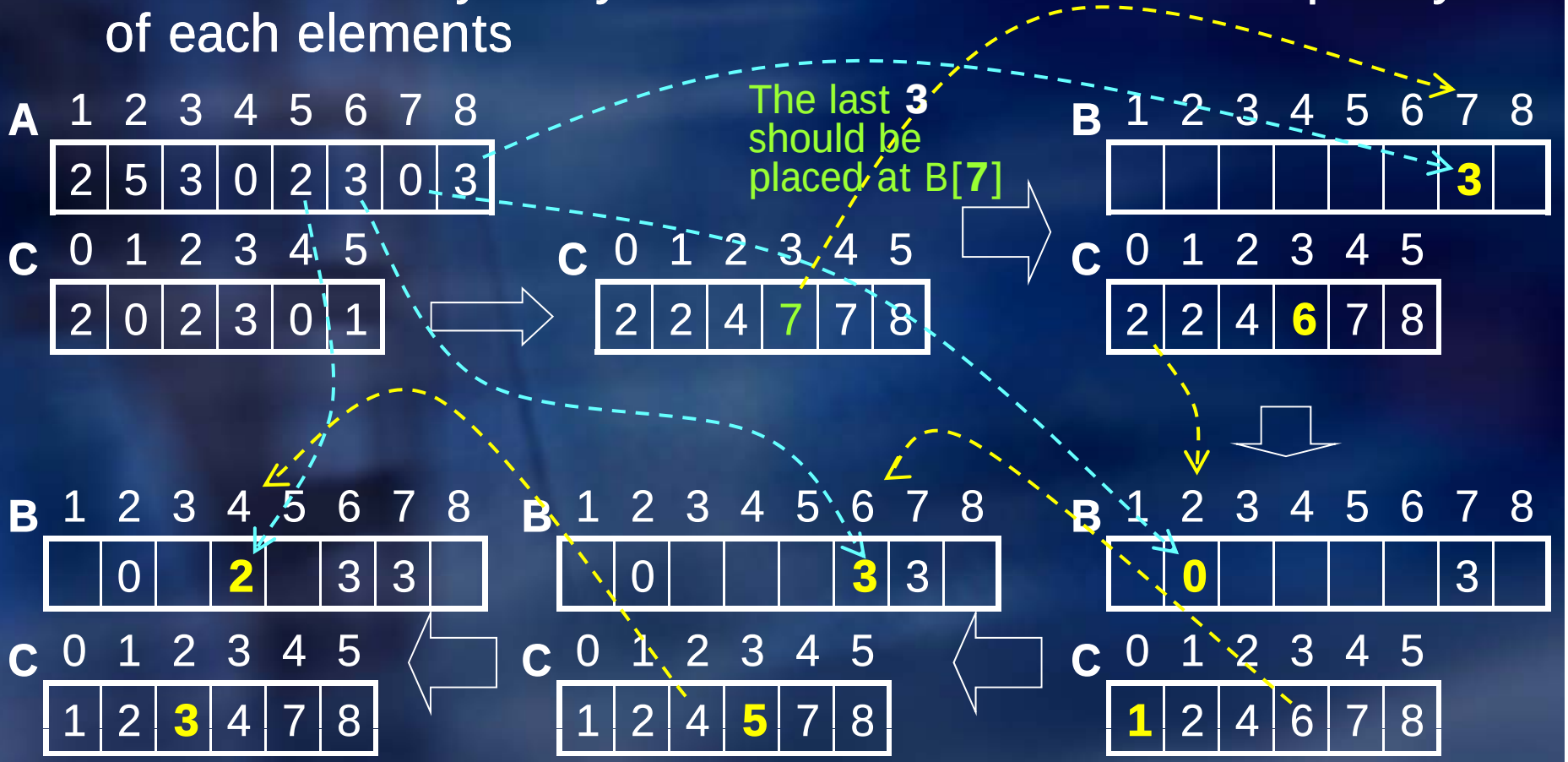
```
int i, j, k, n=3;
double a_mat[3][3];
double b_mat[3][3], sum;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
    {
        for (k=0, sum=0; k<n; k++)
            sum += a_mat[i][k] * b_mat[j][k];
        a_mat[i][j] = sum;
    }
```

```
int i, j, m=3, n=5;
double a_mat[3][5], b_mat[3][5];
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        a_mat[i][j] += b_mat[i][j];
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 1 & 2 & 5 \end{bmatrix} \quad \begin{bmatrix} 7 & 4 & 1 \\ 2 & 1 & 4 \\ 3 & 0 & 4 \end{bmatrix}$$

Counting Sort

- Elements to be sorted are in a set {0,1,...,k}
- Use an auxiliary array to count the occurrence frequency of each elements



- Non-comparison sort, stable sort

Radix Sort

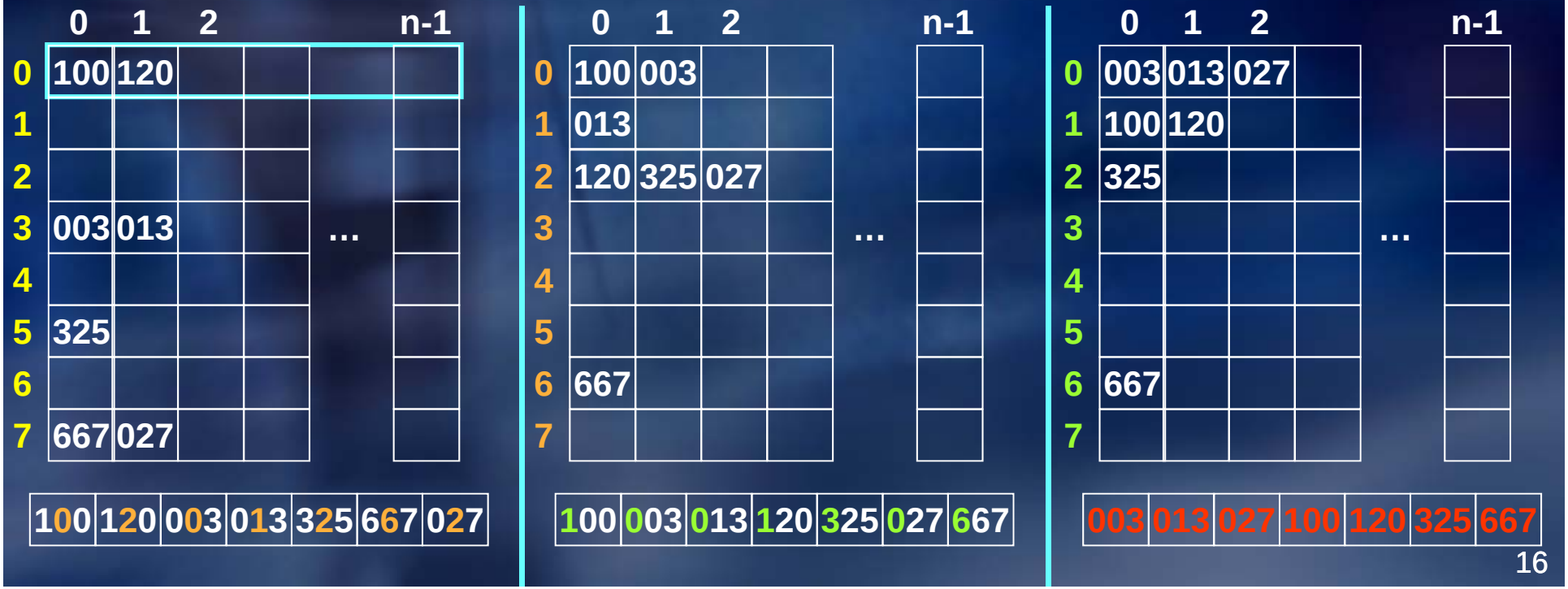
- Stably sort each digits, least significant digits first

Bucket (Bin) Sort
distribute MSB first

- Radix-sort(Array, n)
  1. for i=0 to n-1
  2. use a stable sort algorithm to sort Array on digit i

Radix-8 Sort (cont'd)

- A radix-8 sort
  - 1-dim array of positive integers to be sorted:
    - e.g. 10**0**, 00**3**, 66**7**, 02**7**, 12**0**, 01**3**, 32**5** **in octal**
  - 2-dim array of integers is used as the working space
    - rows (the buckets) indexed from 0 to 7 and
    - columns indexed from 0 to n-1



Radix-8 Sort (cont'd)

- The radix-8 sorting is done as follows:
  - **Distribute**: Place each value of the one-dimensional vector into a bucket, based on the value's rightmost octal digit. For example, 67 is placed in row 7, 3 is placed in row 3 and 100 is placed in row 0. This procedure is called a distribution pass.
  - **Gather**: Loop through the bucket vector row by row, and copy the values back to the original vector. This procedure is called a gathering pass. The new order of the preceding values in the one-dimensional vector is 100, 3 and 67.
  - **Repeat** this process for each subsequent digit position (2nd rightmost, 3rd rightmost, etc.). e.g. On the second pass, 100 is placed in row 0, 3 is placed in row 0 (3 can be seen as 003) and 97 is placed in row 9. After the gathering pass, the order of the values in the one-dimensional vector is 100, 3 and 97. On the third (3rd rightmost) pass, 100 is placed in row 1, 3 is placed in row 0 and 97 is placed in row 0 (after the 3). After this last gathering pass, the original vector is in sorted order.

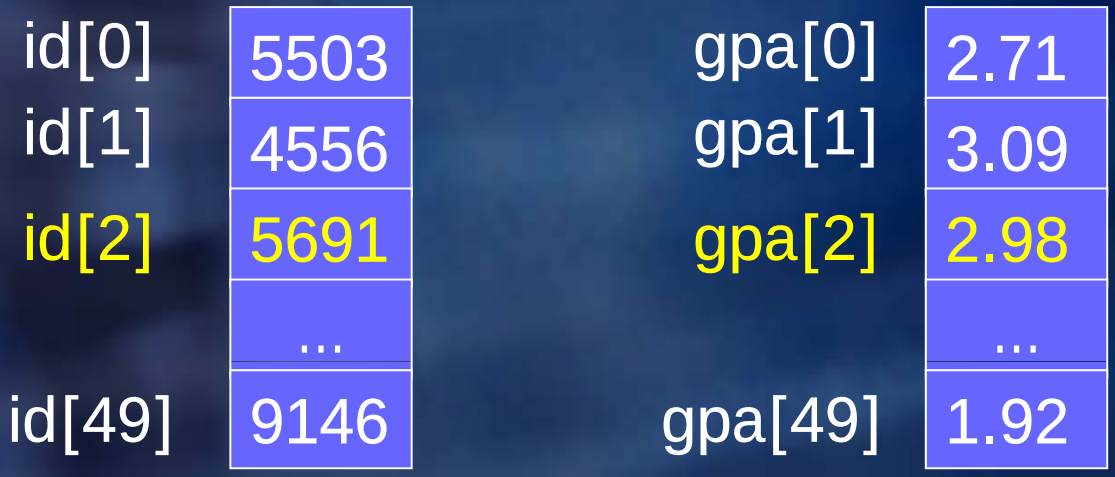
Radix Sort Implementation

```
01 void radix8Sort(int ndata, int data[]) {
02     int buckets[8][MAX], int nBucket[8];
03     int i, j, k, index, mult, iBucket;
04     int len = maxNumDigits(ndata, data); /* max number of octal digits */
05     mult = 1;
06     for (i=0; i<len; i++) {
07         for (j=0; j<8; j++) nBucket[j] = 0;
08         for (j=0; j<ndata; j++) {                    redistribute
09             iBucket = data[j] / mult % 8;
10             buckets[iBucket][nBucket[iBucket]++] = data[j];
11         }
12         for (j=0, index=0; j<8; j++)
13             for (k=0; k<nBucket[j]; k++)
14                 data[index++] =
15                         buckets[j][k];
16         mult *= 8;
17     }                                  gather
18 }
```

```
19 int maxNumDigits(int ndata,
20                                int data[]) {
21     int i, max = -1;
22     for (i=0; i<ndata; i++)
23         if (data[i] > max)
24             max = data[i];
25     return (log10(max)/log10(8))+1;
26 }
```

Parallel Arrays

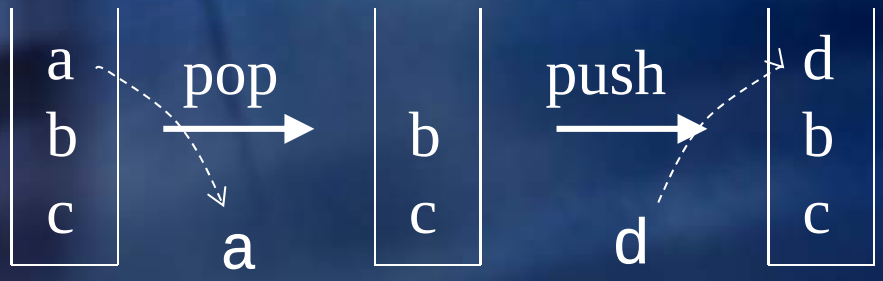
- Two or more arrays with the same number of elements used for **storing related information** about a collection of data objects
- A very common method to organize data with arrays



- id[**i**] and gpa[**i**] refer to the information related to the **i**-th student

Stacks

- A **stack** is a data structure in which only the top element can be accessed.
- For example, the plates stored in the spring-loaded device in a buffet line perform like a stack. A customer always takes the top plate; when a plate is removed, the plate beneath it moves to the top.
- Popping the stack: remove a value from a stack.
- Pushing it onto the stack: store an item in a stack.



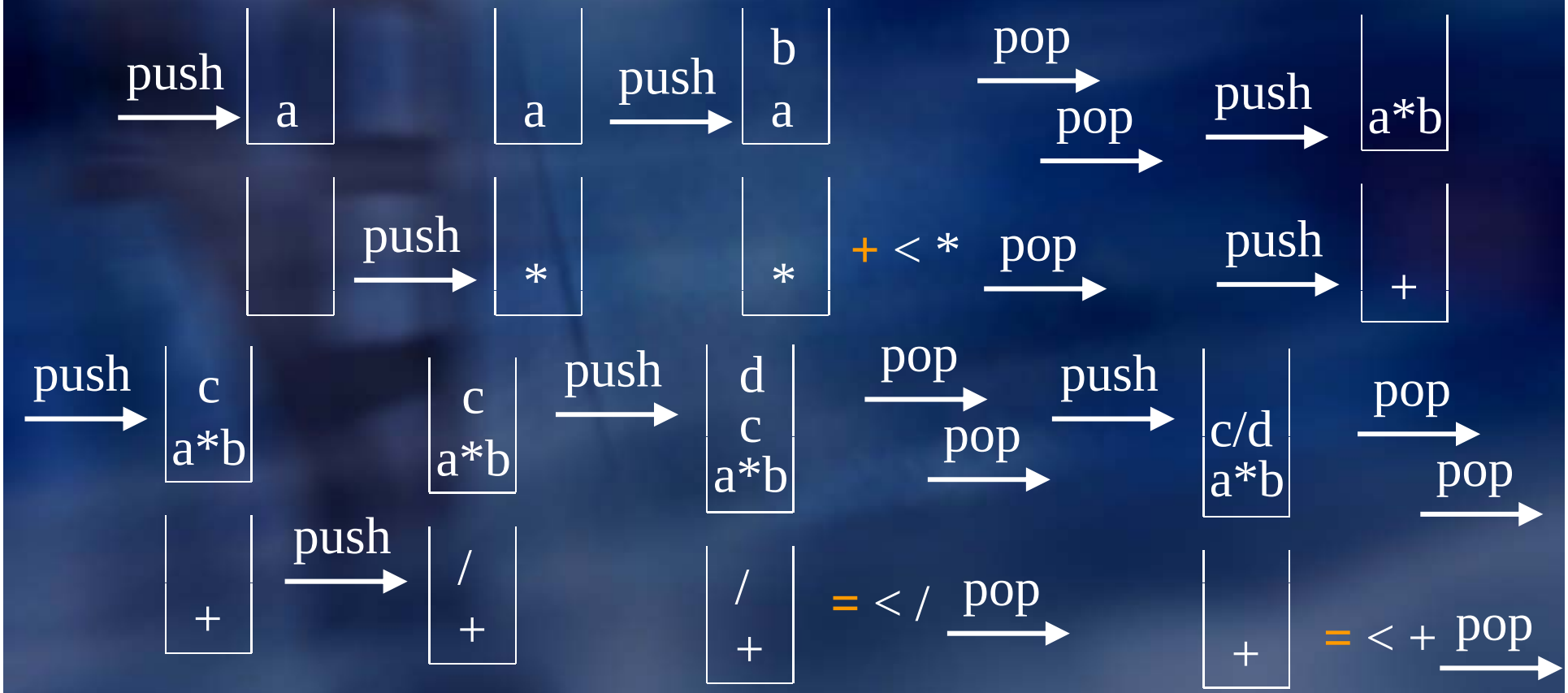
- Array is one of the approaches to implement a stack.

Algorithm Utilizing Stacks

- Expression evaluation

  a * b + c / d =

  - Two stacks: operand stack, operator stack



Push: Insert a New Element to the Top of Stack

```
#define STACK_SIZE 100
char stack[STACK_SIZE];
int top = -1; /* the position of current stack top */
```

```
push(stack, 'a', &top, STACK_SIZE);
```

```
01 void
02 push(char stack[],     /* input/output - the stack */
03        char item,      /* input - data being pushed onto the stack */
04        int  *top,      /* input/output - pointer to top of stack */
05        int  max_size)  /* input - maximum size of stack */
06 {
07        if (*top < max_size-1) {
08                ++(*top);
09                stack[*top] = item;
10        }
11 }
```

Pop: Remove from Top of Stack an Element

```
char content;
...
content = pop(stack, &top);
```

```
01 char
02 pop(char stack[],    /* input/output - the stack */
03        int  *top)     /* input/output - pointer to top of stack */
04 {
05     char item;     /* value popped off the stack */
06
07     if (*top >= 0) {
08         item = stack[*top];
09         --(*top);
10     } else {
11         item = STACK_EMPTY;
12     }
13     return item;
14 }
```

Binary Tree

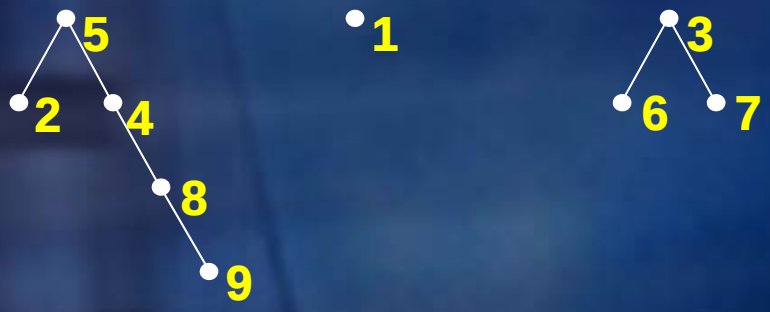
- A binary tree is a tree data structure in which each node has at most two children.

104	71	24	66	27	23	8	5	32	NIL	25	18	22	NIL	NIL
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- To represent a binary tree, the value in node label i can be stored in cell i of an array
- The parent of node label i is node label $\lfloor i/2 \rfloor$
  The left child of node label i is node label 2*i
  The right child of node label i is node label 2*i+1

Disjoint Set

- Pairwise disjoint sets X and Y satisfies $X \cap Y = \phi$
  e.g. {2, 4, 5, 8, 9}, {1}, {3, 6, 7}
- Each set can be represented as an arbitrary structured tree and the root is marked as the representative of that set



- This disjoint sets can be represented as an array – parent, parent[i] is the parent of i. A root's parent is itself.

parent

1	5	3	5	5	3	3	4	8
1	2	3	4	5	6	7	8	9