TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chapter 2. Basic data structures

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Contents

2.1. Array

2.2. Record

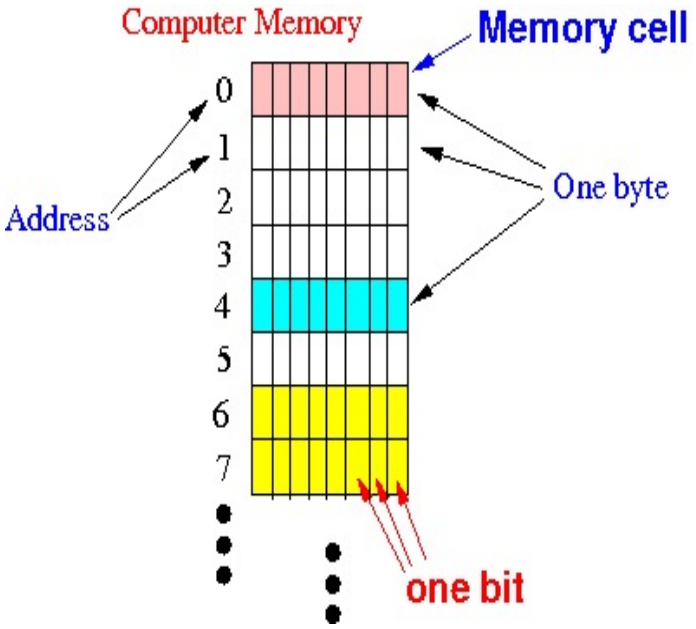
2.3. Linked List

2.4. Stack

2.5. Queue

Computer memory

- Computer memory is made of a long sequence of memory cells, each 8 bits (one byte) long
- Associated with each memory cell is an address
- Variable names in a program are addresses, i.e. the cell where the data is stored
  - **int** myvar;
- The **type** of a variable "**int**" defines the number of consecutive memory cells used to store the data
  - **int** means 4 consecutive cells are reserved to store the data of myvar



Variable declaration

```
#include <stdio.h>
#include <stdlib.h>

int main(){
   int myvar;
  printf("addr myvar %8u\n",&myvar);
}
```

```
 #include <stdio.h>
 #include <stdlib.h>

 int main(){
  int myvar;

  myvar = 16;

  printf("addr myvar %8u, and value stored at
 the addr of myvar %d\n",&myvar,myvar);
 }
```

- This first program (basic1.c) has a single variable, this variable has a memory cell address
- The second program (basic1-1.c) assigned the number 16 to the 4 memory cells starting at addr of myvar
  - Now there is data in the next 4 cells starting at the address of myvar

Variable assignment

```
#include <stdio.h>
#include <stdlib.h>

int main(){
  int var1;
   int var2;
   var1 = 16;
   var2= var1;
  printf("addr var1 %8u, value
var1 %d, value var2
%d\n",&var1,var1,var2);
}
```

- This program (basic3.c) copies the value stored in memory cells starting at addr var1 into memory cells starting at the addr of var2

Example

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int var1;
  double var2;
  int var3[5];
  printf("addr var1 int %8u\n",&var1);
  printf("addr var2 double %8u\n",&var2);
  printf("addr var3 array of int %8u\n",&var3);
```

- This program (basic2.c) shows that the "type", "**int**" "**double**", etc, in front of a variable name specifies the number of consecutive memory cells reserved for the variable
- Print the addr of
  - The first cell of var1
  - The first cell of var2
  - The first cell of the array var3

Array storage

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int var1;
  int array1[3];
  int array2[3];

  printf("addr var1%8u\n",&var1);
  printf("addr array1 %8u\n",&array1);
  printf("addr array2 %8u\n",&array2);

}
```

- This program (basic4.c) declares 3 variables, two of them are arrays
- It shows the consecutive memory cells used by arrays

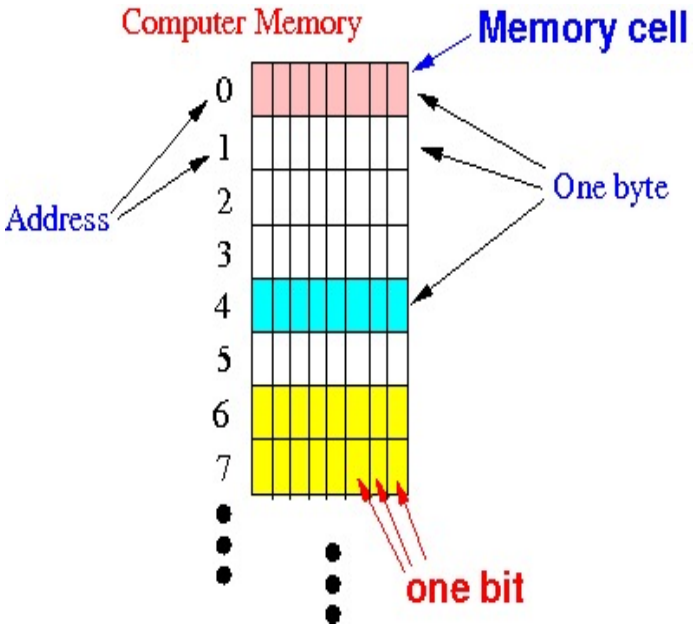
How computers compute index addresses

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int var1;
  int array1[3];
  int array2[3];
  var1 = 16;
  array1[2] = var1;

 printf("addr var1 %8u value var1
%d\n",&var1,var1);
 printf("addr array1 %8u, addr
array1[2]  %8u, value array1[2]
%d\n",&array1, &array1[2],array1[2]);
}
```

- (basic5.c) array1 is the address of the memory cell where this data structure starts
- However, array1 has storage for 3 **int**
- In order to store the value stored in var1 in array1[2], the computer needs to find the addr of the memory cell where the third **int** starts
- This is array1 + 8

Pointers

- Memory cells may also store addresses
- In this case we declare a variable of type pointer
  - int* mypointer;
- The previous declaration said that the cells starting at addr mypointer will contain the addr of a memory cell which is the beginning of an **int**



Example: pointers

- Print the addr of the memory cells for the var “pointer” and “var1”.
- Print the contain of memory cell “pointer”, empty. Assign the addr of the memory cell “vars” to the cell of pointer
- Print the contain of memory cell “pointer” and the contain of the addr stored in memory cell “pointer” (basic6.c)

```
#include <stdio.h>
#include <stdlib
int main(){
  int var1 = 16;
  int* pointer;

  printf("addr of the memory cell for pointer %8u\n\n",&pointer);
  printf("addr of the memory cell for var1 %8u\n\n",&var1);
  printf("value stored in the memory cell pointer %8u\n\n",pointer);

  pointer = &var1;
  printf("value stored in the memory cell pointer %8u\n\n",pointer);
  printf("value of the addr stored in  memory cell pointer %8u\n",*pointer);
}
```

Dynamic memory allocation

```
int main(){
  int var1 = 16;
  int *array2 = NULL;

  printf("first array2 %8u\n",array2);

  array2 = (int*)malloc(3*sizeof(int));

  printf("addr array2 %8u\n",array2);

  array2[2] = var1;

  printf("addr array2 %8u,  array2[2]
%8u, value array2[2]
%d\n",&array2[0],&array2[2],array2[2]);
}
```

- (basic7.c) array2 is of type pointer
- malloc allocate consecutive memory cells to the program for 3 integers
- malloc returns the addr of the first cell of the consecutive memory cells for the integers
- Print
  - The addr of the first cell of array2
  - The addr of the first cell of the third integer in array2 (array2[2])
  - The value stored in the array2[2]

ARRAY

- An array is always a sequence of consecutive memory cells
- The number of memory cells is
  - size of array * sizeof(type)

2.1. Array

- Imagine that we have 100 scores. We need to read them, process them and print them. We must also keep these 100 scores in memory for the duration of the program. We can define a hundred variables, each with a different name, as shown in Figure1.

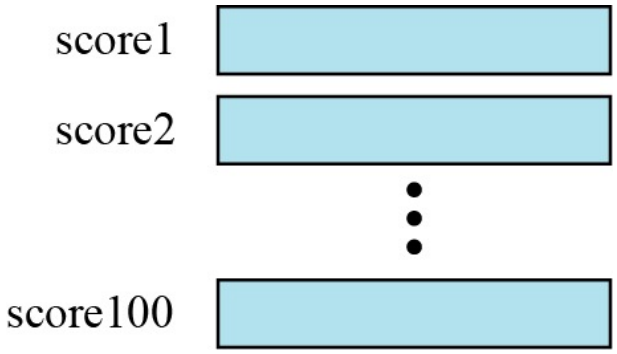


Figure 1 A hundred individual variables

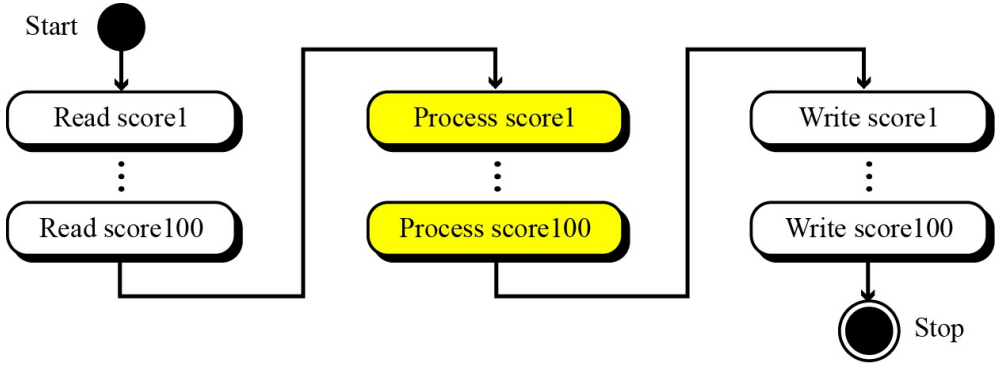


Figure 2 Processing individual variables

- But having 100 different names creates other problems. We need 100 references to read them, 100 references to process them and 100 references to write them. Figure 2 shows a diagram that illustrates this problem.

2.1. Array

- An array is a sequenced of elements, normally of the same data type, although some programming languages accept arrays in which elements are of different types.
- We can refer to the elements in the array as the first element, the second element and so forth, until we get to the last element.

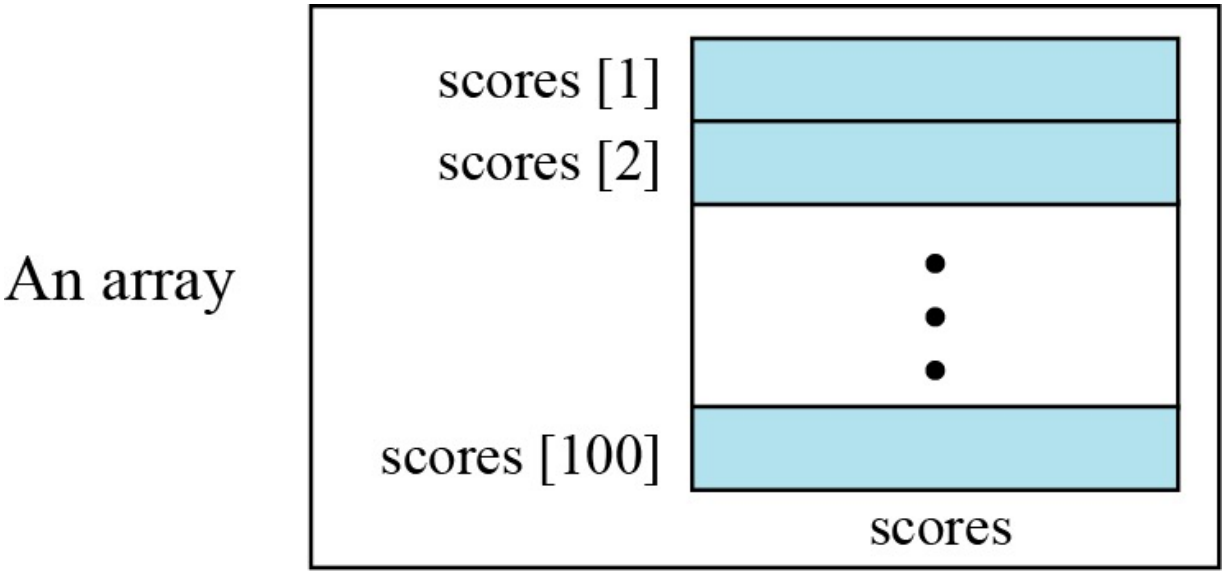
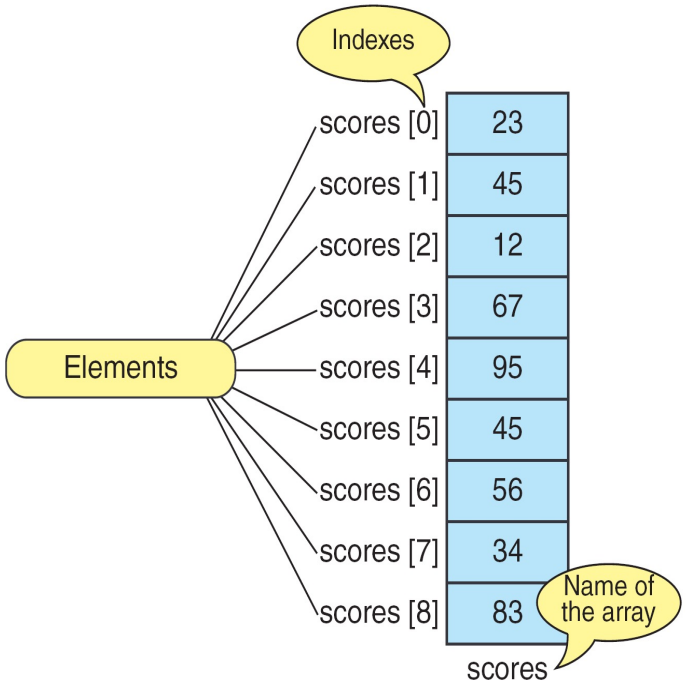


Figure 3. Arrays with indexes

2.1. Array

Basic definitions

- An array is a fixed size sequential collection of elements of identical types.
- Array: a set of pairs (index and value)
  - data structure: for each index, there is a value associated with that index.
  - representation: implemented by using consecutive memory.
- In C/C++/Java: the element in an array are indexed by the integers 0 to *n*- 1, where *n* is the size of the array



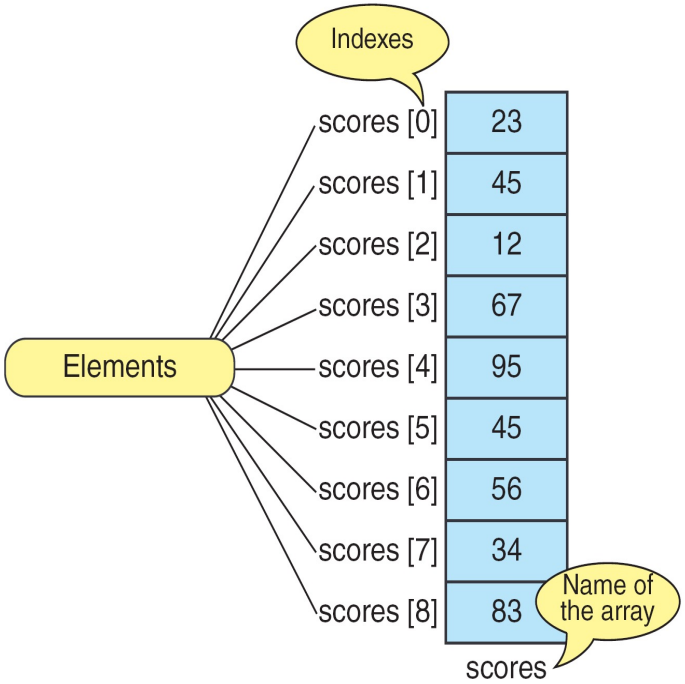
Array name versus element name

In an array we have two types of identifiers:

- the name of the array
- the name of each individual element.

The name of the array is the name of the whole structure, while the name of an element allows us to refer to that element.

Example:



the name of the array is *scores,* and name of each element is the name of the array followed by the index, for example, *scores*[0], *scores*[1], and so on.

Array types in C

C supports two types of arrays:

- ✓ ***Fixed Length Arrays*** : The programmer "hard codes" the length of the array, which is fixed at run-time.
- ✓ ***Variable-Length Arrays :*** The programmer doesn't know the array's length until run-time.

Declaring an one-dimensional array

To declare an array, we need to specify its data type, the array's identifier and the size:

```
type arrayName [arraySize];
```

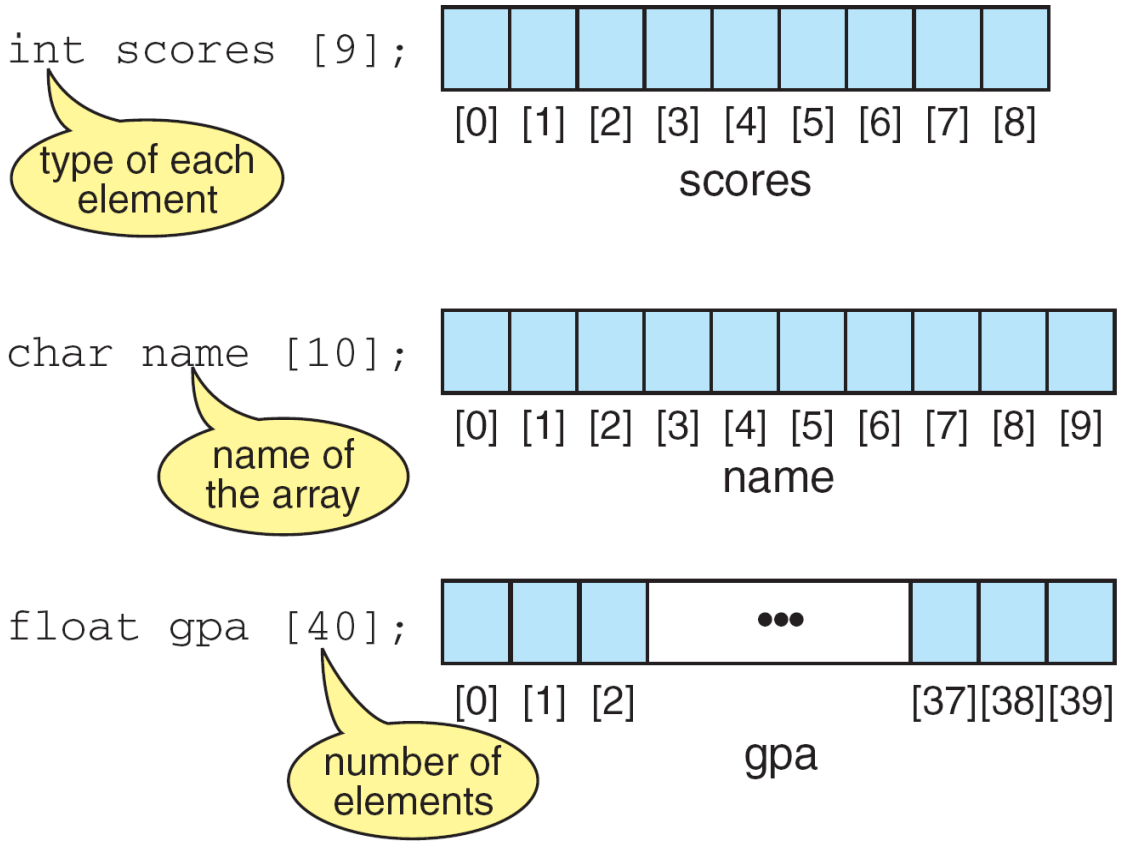
Example: declare `int A[5];`

to create an array A having 5 elements of integer type (4 bytes for each element)

- The **arraySize** can be a constant (for fixed length arrays) or a variable (for variable-length arrays)
  - Example:
```
double A[10];
int n;
double A[n];
```

- Before using an array (even if it is a variable-length array), we must declare and initialize it!

Declaration example: fixed length array

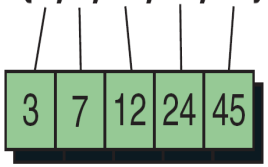


Declaring a one-dimensional array

- We can initialize fixed-length array elements when we define an array.
- If we initialize fewer values than the length of the array, C assigns zeroes to the remaining elements.

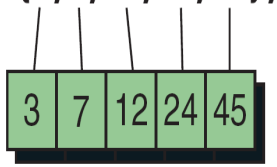
(a) Basic Initialization

```
int numbers[5] = {3,7,12,24,45};
```



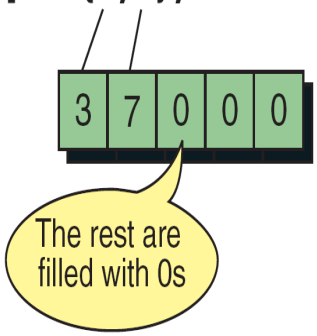
(b) Initialization without Size

```
int numbers[ ] = {3,7,12,24,45};
```



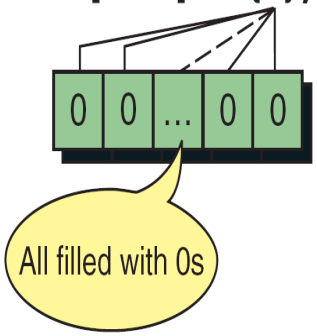
(c) Partial Initialization

```
int numbers[5] = {3,7};
```



(d) Initialization to All Zeros

```
int lotsOfNumbers [1000] = {0};
```

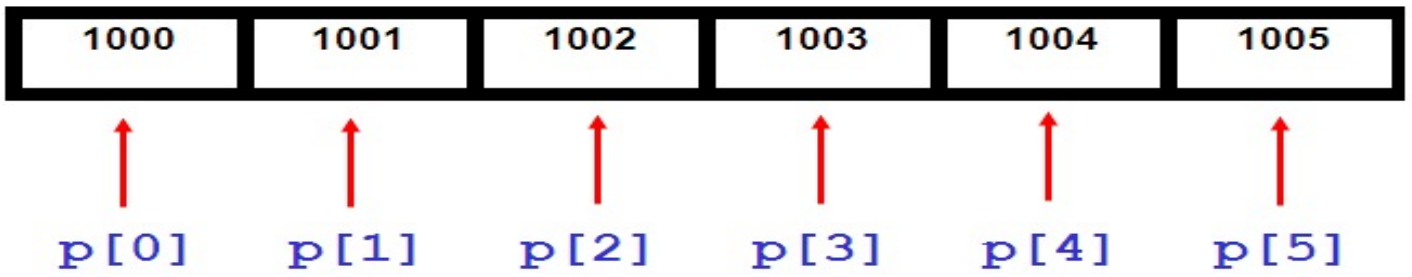


Declaring an Array

- So what is actually going on when you set up an array?

Memory:

- Each element is held in the next location along in memory
- Essentially what the computer is doing is looking at the **FIRST** address (which is **pointed** to by the variable **p**) and then just counts along.



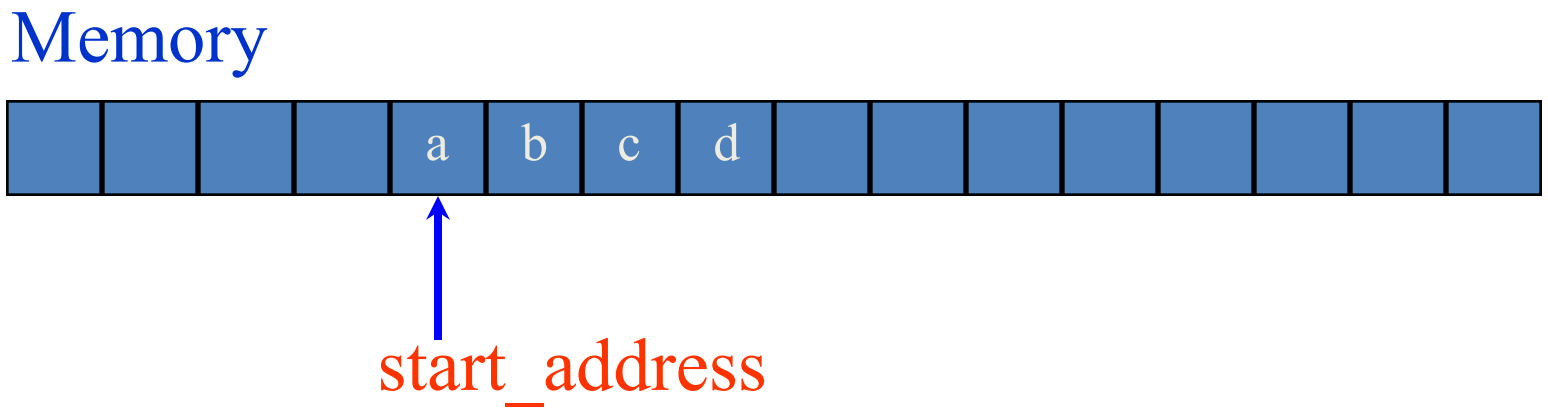
Accessing Elements

To access an array's element, we need to provide an integral value to identify the index we want to access.

- We can do this using a constant: `scores[0];`
- We can also use a variable:

```
for(i = 0; i < 9; i++)
    scoresSum += scores[i];
```

1D Array Representation in C



Example: 1-dimensional array x = [a, b, c, d]

- Map into contiguous memory locations
- Location($x[i]$) = start_address + W*i

where

- start_address: the address of the first element in the array
- W: size of each element in the array

Example

Write a C program that gives the address of each element of an 1D array:

```
#include <stdio.h>
int main()
{    int A[ ] = {5, 10, 12, 15, 4};
      int rows=5;
      /* print the address of 1D array using pointer   */
    int *ptr = A;
    printf("Address     Contents\n");
    for (int i=0; i < rows; i++)
          printf("%8u %5d\n", ptr+i, *(ptr+i));
}
```

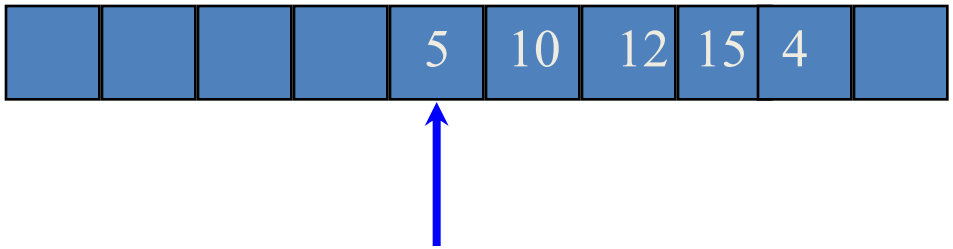
Result in DevC
(sizeof(int)=4)

```
Address     Contents
 6487536        5
 6487540       10
 6487544       12
 6487548       15
 6487552        4
```

ptr+i : address of element A[i]
*(ptr+i) : content of element A[i]

Memory

Location(A[i]) = start_address + W*i



Result in turboC
(sizeof(int)=2)

```
Address     Contents
   65516        5
   65518       10
   65520       12
   65522       15
   65524        4
```

Arrays in C

```
int list[5], *plist[5];
```

list[5]: five integers
list[0], list[1], list[2], list[3], list[4]
*plist[5]: five pointers to integers
plist[0], plist[1], plist[2], plist[3], plist[4]

Implementation of 1-D array

list[0]	start address = α
list[1]	α + sizeof(int)
list[2]	α + 2*sizeof(int)
list[3]	α + 3*sizeof(int)
list[4]	α + 4*sizeof(int)

- Compare `int *list1` and `int list2[5]`:
  - Same: list1 and list2 are pointers.
  - Difference: list2 reserves five locations.
  - Notations:
    - list2 : a pointer to list2[0]
    - (list2 + i) : a pointer to list2[i] (&list2[i])
    - *(list2 + i) : content of list2[i]

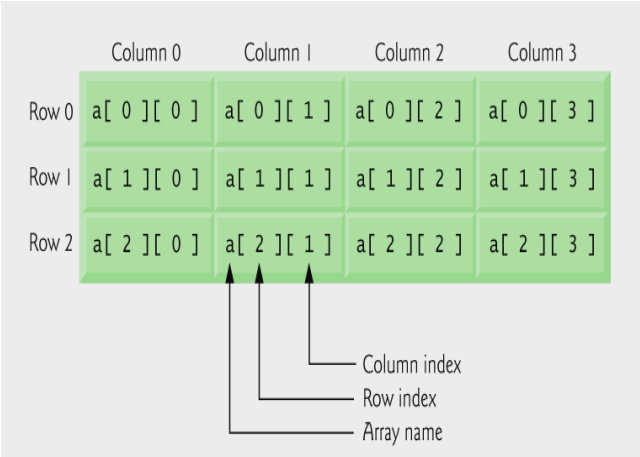
Declaring two-dimensional array

- How to declare:

```
<element-type> <arrayName> [size1][size2];
```

Example: `double a[3][4];`

may be shown as a table



- Using the two-dimensional array initializer

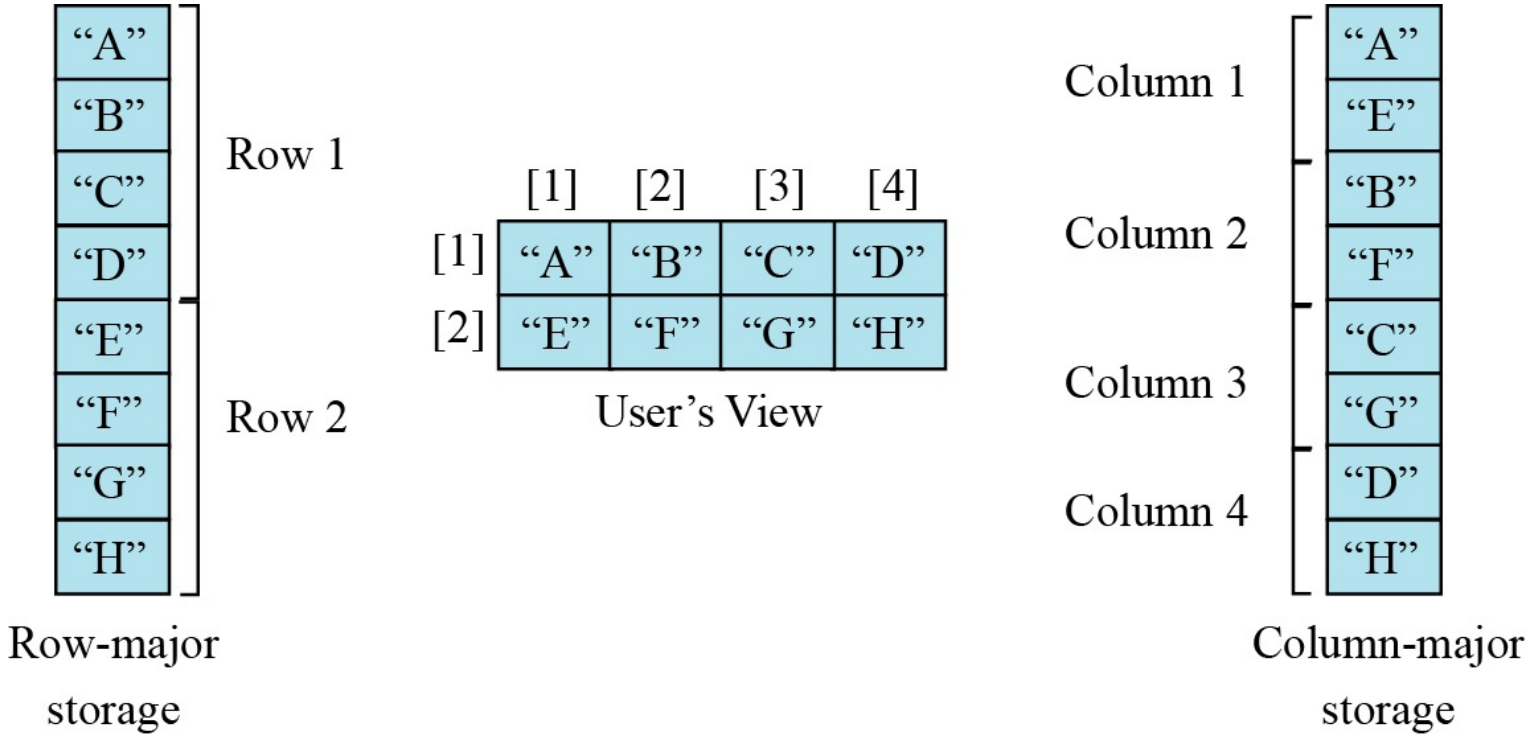
Example: `int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};`

- Access to element of array: a[2] [1];

a[0][0] = 1	a[0][1]=2	a[0][2]=3	a[0][3]=4
a[1][0] = 5	a[1][1]=6	a[1][2]=7	a[1][3]=8
a[2][0] = 9	a[2][1]=10	a[2][2]=11	a[2][3]=12

Representation of Arrays

- Multidimensional arrays are usually implemented by one dimensional array via either row major order or column major order.

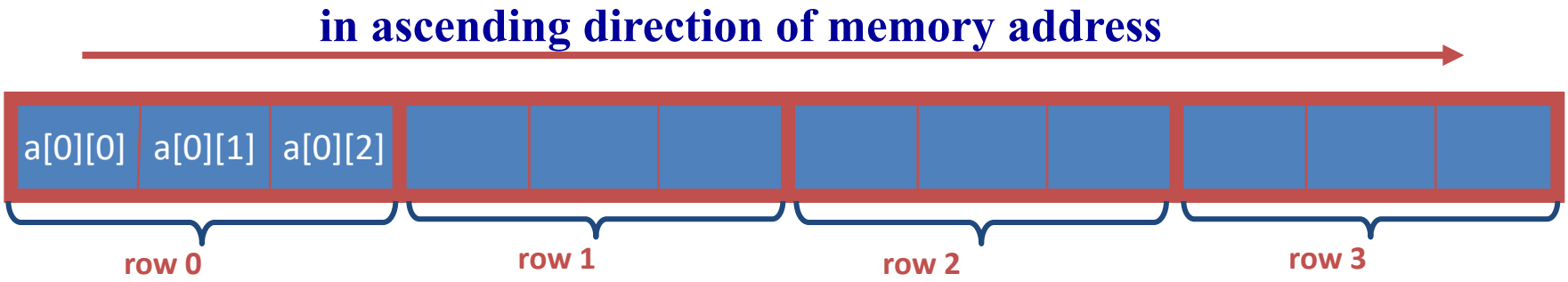


Row-Major Mapping (e.g. Pascal, C/C++)

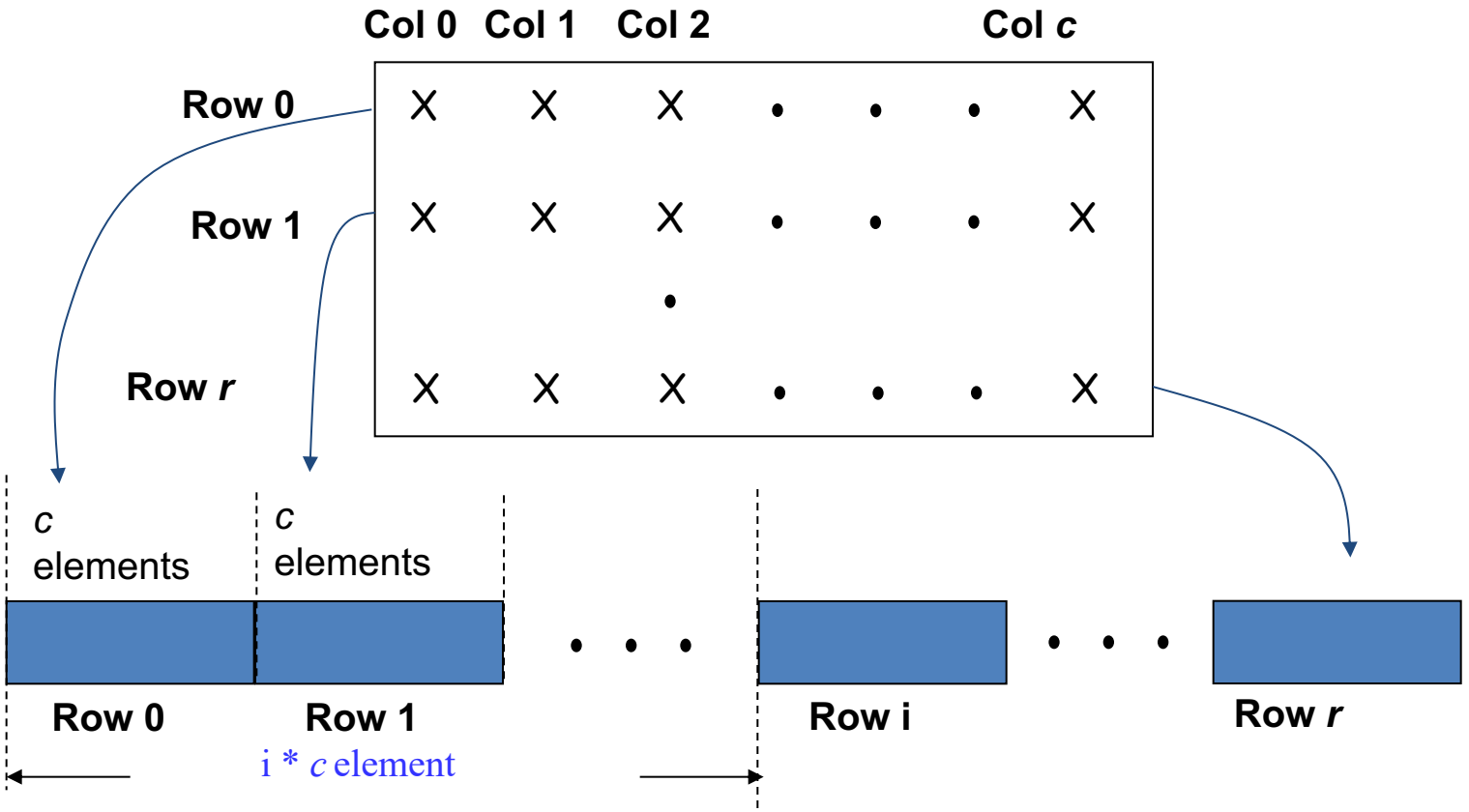
- Row- major order is a method of representing multi-dimensional array in sequential memory. In this method, elements of an array are arranged sequentially row by row. Thus, elements of the first row occupies the first set of memory locations reserved for the array, elements of the second row occupies the next set of memory and so on.

Elements of Row 0	Elements of Row 1	Elements of Row 2	....	Elements of Row i	.....

- Example: int a[4][3]



Two Dimensional Array Row Major Order



Column-Major Mapping (e.g. Matlab, Fortran)

- In this method, elements of an array are arranged sequentially column by column. Thus, elements of the first column occupies the first set of memory locations reserved for the array, elements of the second column occupies the next set of memory and so on.

- Example 3 x 4 array:

a b c d

e f g h

i j k l

Convert into 1D array **Y** by collecting elements by columns.

- Within a column elements are collected from top to bottom.
- Columns are collected from left to right.

Thus, we get **Y**[ ] =

{a, e, i, b, f, j, c, g, k, d, h, l}

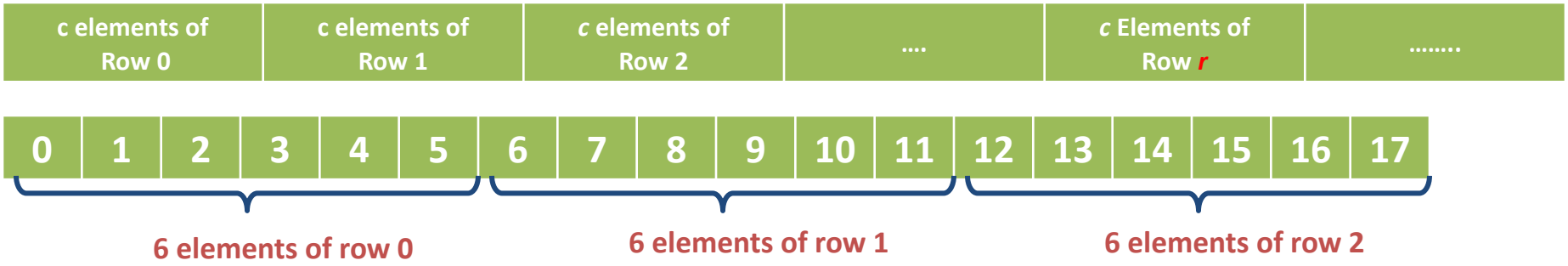
Row- and Column-Major Mappings

2D array: *r* rows, *c* columns

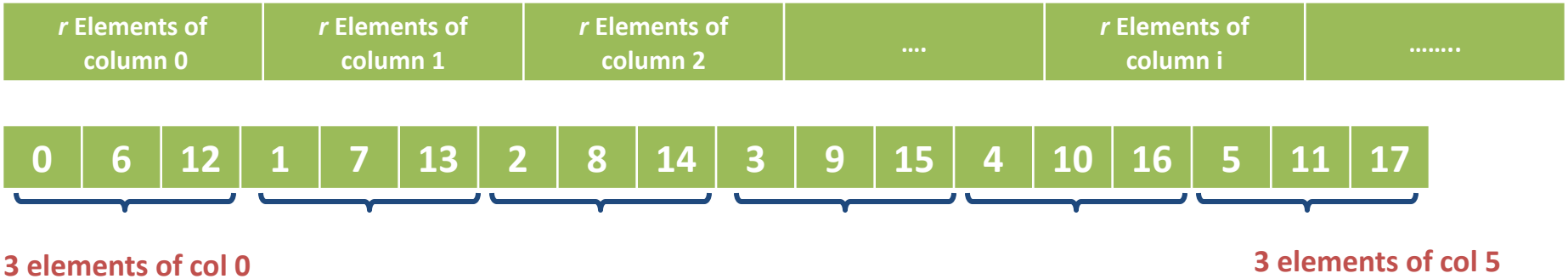
Example: int a[3][6]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17};

a[0][0]=0	a[0][1]=1	a[0][2]=2	a[0][3]=3	a[0][4]=4	a[0][5]=5
a[1][0]=6	a[1][1]=7	a[1][2]=8	a[1][3]=9	a[1][4]=10	a[1][5]=11
a[2][0]=12	a[2][1]=13	a[2][2]=14	a[2][3]=15	a[2][4]=16	a[2][5]=17

Memory: row-major order



Memory: column-major order



Locating Element x[i][j]: row-major order

- Assume x:
  - has ***r*** rows and ***c*** columns (thus, each row has ***c*** elements)

- Locating element $x[i][j]$:
  - i rows to the left of row 0 ➔ so $i*c$ elements to the left of $x[i][0]$
  - $x[i][j]$ is mapped to position: $i*c + j$ of the 1D array
  - The location of element $x[i][j]$:

  Location($x[i][j]$) =start_address + W * ($i*c + j$)

  Where
  - start_address: the address of the first element ($x[0][0]$) in the array
  - W: is the size of each element
  - c: number of columns in the array

Example

Write a C program that gives the address of each element of a 2D array:

```
#include <stdio.h>
int main()
{   int  a[3] [4] = {1,2,3,4,5,6,7,8,9,10,11,12};
    int rows=3, cols =4;
    /* print the address of 2D array by using pointer  */
    int *ptr = a;
    printf("Address    Contents\n");
    for (int i=0; i < rows; i++)
       for (int j=0; j < cols; j++)
        printf("%8u %5d\n", ptr +((i*cols)+j), *(ptr + ((i*cols)+j)) );}
```

Result in DevC

(sizeof(int)=4)

Address	Contents
6487488	1
6487492	2
6487496	3
6487500	4
6487504	5
6487508	6
6487512	7
6487516	8
6487520	9
6487524	10
6487528	11
6487532	12

Memory

Location(a[i][j]) = start_address + W*[(i*cols) + j]

Example row-major order: Memory allocation for 2D array (type int)

- Address (location) of elements in 2D array:

  int a[4][3]

a[0][0]	address = α
a[0][1]	α + 1*sizeof(int)
a[0][2]	α + 2*sizeof(int)
a[1][0]	α + 3*sizeof(int)
a[1][1]	α + 4*sizeof(int)
a[1][2]	α + 5*sizeof(int)
a[2][0]	α + 6*sizeof(int)

. . .

General: Declare

int a[m][n];

- Assume: the address of the first element (a[0][0]) is α.
- Then, the address of element a[i][j] is:

  α + (i*n + j)*sizeof(int)

Locating Element x[i][j]: column-major order

- Assume x:

  - has ***r*** rows and ***c*** columns (thus, each column has ***r*** elements)
- Locating element $x[i][j]$:
  - j columns to the left of column 0 ➔ so $j*r$ elements to the left of $x[0][j]$
  - $x[i][j]$ is mapped to position: $j*r + i$ of the 1D array
  - The location of element $x[i][j]$:

    Location($x[i][j]$) = start_address + W * ($j*r + i$)

    Where
    - start_address: the address of the first element in the array
    - W: is the size of each element
    - c: number of columns in the array

Example: array : `int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};`

Determine the address of the element a[1][2] if start_address = 6487488

Example 1: Row- and Column-Major Mappings

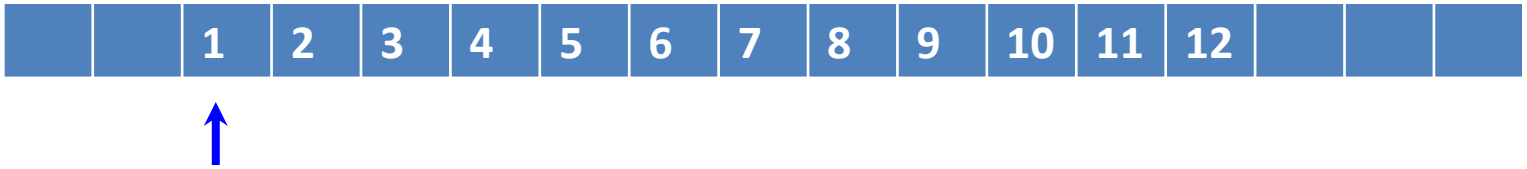
2D array:

int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};

a[0][0] = 1	a[0][1]=2	a[0][2]=3	a[0][3]=4
a[1][0] = 5	a[1][1]=6	a[1][2]=7	a[1][3]=8
a[2][0] = 9	a[2][1]=10	a[2][2]=11	a[2][3]=12

Memory: row-major order

Location(a[i][j]) = start_address + W*[(i*cols) + j]



start_address=6487488

Location(a[1][2]) = ?

Memory: column-major order

Location(a[i][j]) = start_address + W*[(j*rows) + i]

Location(a[1][2]) = ?

start_address=6487488

Example 2: Row- and Column-Major Mappings

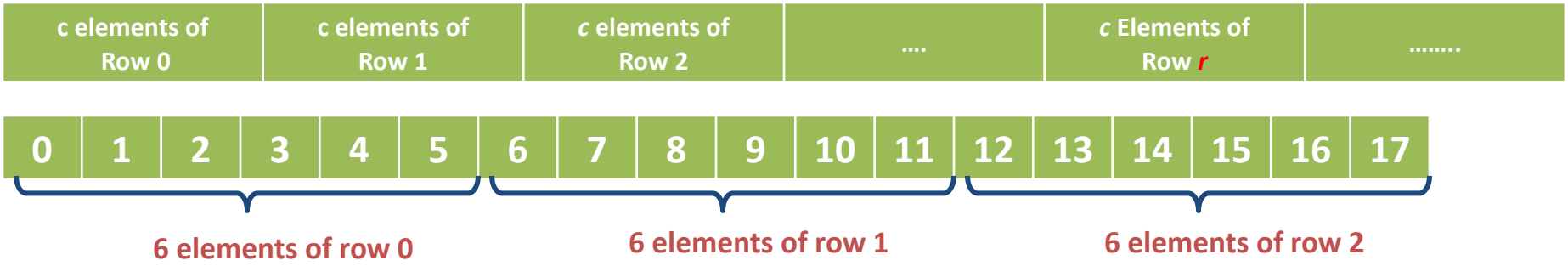
2D array: *r* rows, *c* columns

Example: int a[3][6]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17};

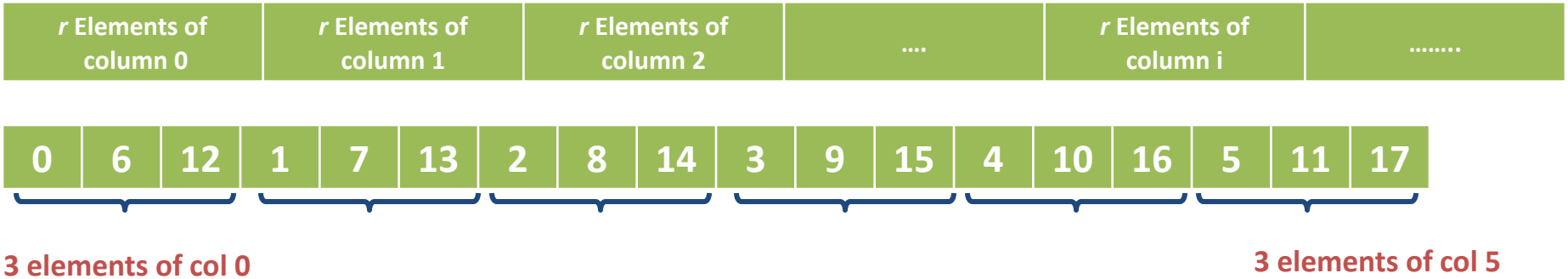
a[0][0]=0	a[0][1]=1	a[0][2]=2	a[0][3]=3	a[0][4]=4	a[0][5]=5
a[1][0]=6	a[1][1]=7	a[1][2]=8	a[1][3]=9	a[1][4]=10	a[1][5]=11
a[2][0]=12	a[2][1]=13	a[2][2]=14	a[2][3]=15	a[2][4]=16	a[2][5]=17

Memory: row-major order

start_address = 1000 ➔ Location(a[1][4]) = ?

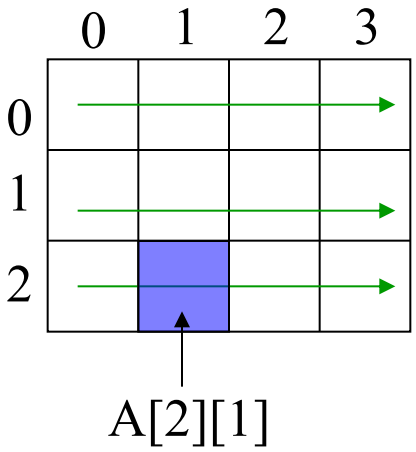


Memory: column-major order

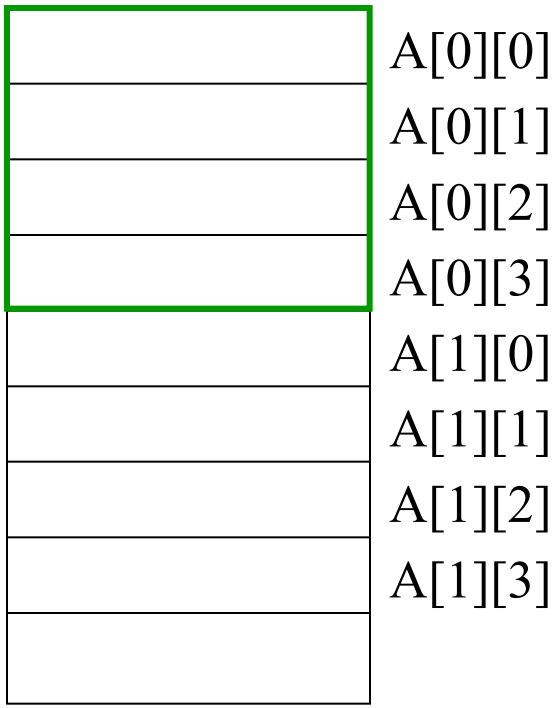


Example: Memory mapping

- `char A[3][4];` // row-major
- logical structure



physical structure



Mapping (location):
Location(A[i][j]) = Location(A[0][0])+ i*4+j

Example

We have stored the two-dimensional array students in memory. The array is 100 × 4 (100 rows and 4 columns). Show the address of the element students[5][3] assuming that the element student[0][0] is stored in the memory location with address 1000 and each element occupies only two bytes memory location. The computer uses row-major storage.

Solution:

Operations on the array

- The common operations on arrays are searching, insertion, deletion, retrieval and traversal.

Example: Given an array *S* consists of *n* integers: S[0], S[1], …, S[*n*-1]

- **Search operation**: search a value whether appears in the array S or not

  `function Search(S,value)` returns true if ***value*** appears in S; false otherwise

- **Retrieval operation**: get the value of the element at index ***i*** of the array S

  `function Retrieve(S, i)`: returns the value S[i] if 0 <= *i* <= *n*-1

- **Traversal operation**: print the value of all elements in the array S

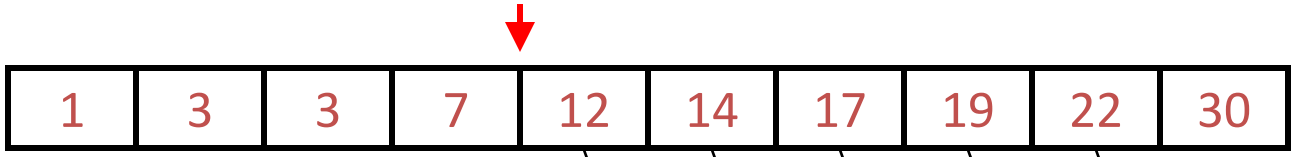
  `function PrintArray(S, n)`

- **Insert operation**: insert a value into the array S
- **Delete operation**: delete the element at index ***i*** of the array S

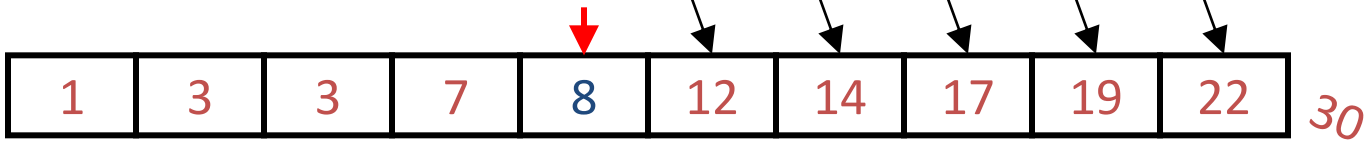
- Although searching, retrieval and traversal of an array is an easy job, insertion and deletion is time consuming. The elements need to be shifted down before insertion and shifted up after deletion.

Inserting an element into an array

- Assume we need to insert 8 into an array already be sorted in ascending order:



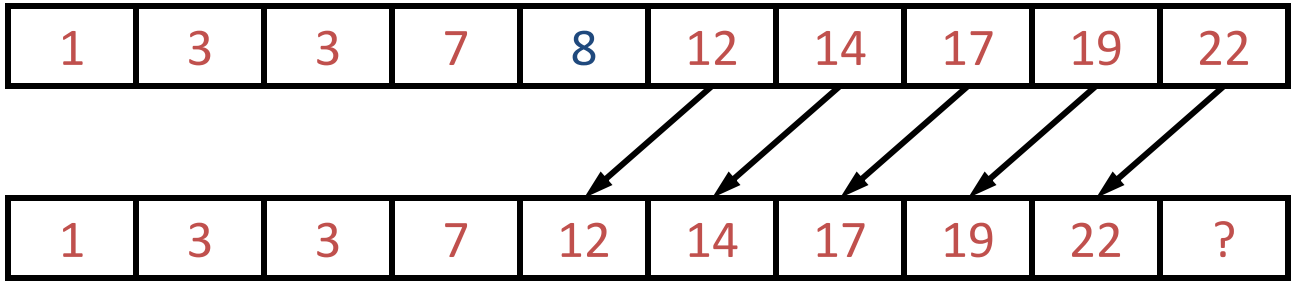
- We can do it by shifting to the right one cell for all the elements after the mark
  - It thus need to remove 30 from the array



• Moving all elements of the array is a slow operation (requires linear time O(n) where n is the size of array)

Deleting an element from an array

- In order to delete an element, we need to shift to the left all previous elements



- Delete operation is a slow operation.
- Regular implementation of this operation is undesirable.
- Delete operation makes the last index free
  - How we could mark the last index of the array being free?
    - We need variable to store the size of the array

    Example: variable size is used to store the size of the array. Before deletion, size = 10. After deletion, we need to update the value of size: size = 10 – 1 = 9

Operations on the array

Thinking about the operations discussed in the previous section gives a clue to the application of arrays. If we have a list in which a lot of insertions and deletions are expected after the original list has been created, we should not use an array. An array is more suitable when the number of deletions and insertions is small, but a lot of searching and retrieval activities are expected.

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

Represent Matrices based on arrays

- ***m x n* matrix** is a table with *m* rows and *n* columns, but numbering begins at 1 rather than 0.
- *M(i,j)* denotes the element in row *i* and column *j*.
- Common matrix operations
  - transpose
  - addition
  - Multiplication

	col 1	col 2	col 3	col 4
row 1	7	2	0	9
row 2	0	1	0	5
row 3	6	4	2	0
row 4	8	2	7	3
row 5	1	4	9	6

- Shortcomings Of Using A 2D Array For A Matrix:
  - Indexes are off by 1.
  - C arrays do not support matrix operations such as add, transpose, multiply, and so on.
    - Suppose that x and y are 2D arrays. Can't do x + y, x –y, x * y, etc. in C.

    ➔We need to develop functions to support all matrix operations.

Sparse Matrix

	col 1	col 2	col 3
row 1	-27	3	4
row 2	6	82	-2
row 3	109	-64	11
row 4	12	8	9
row 5	48	27	47

5*3

15/15

	col1	col2	col3	col4	col5	col6
row1	15	0	0	22	0	−15
row2	0	11	3	0	0	0
row3	0	0	0	−6	0	0
row4	0	0	0	0	0	0
row5	91	0	0	0	0	0
row6	0	0	28	0	0	0

6*6

8/36

sparse matrix

data structure?

Sparse Matrix

(1) Represented by a two-dimensional array (e.g. `int M[6][6];`)

- Sparse matrix wastes space.

(2) Each element is characterized by <row, col, value>.

- The terms in A should be ordered based on <row, col>

	col1	col2	col3	col4	col5	col6
row1	15	0	0	22	0	−15
row2	0	11	3	0	0	0
row3	0	0	0	−6	0	0
row4	0	0	0	0	0	0
row5	91	0	0	0	0	0
row6	0	0	28	0	0	0

6*6

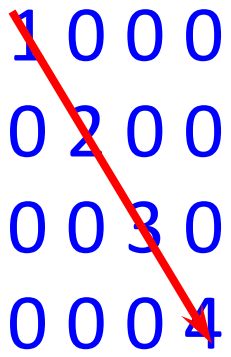
	row	*col*	*value*
A [0]	1	1	15
[1]	1	4	22
[2]	1	6	-15
[3]	2	2	11
[4]	2	3	3
[5]	3	4	-6
[6]	5	1	91
[7]	6	2	28

```
A[0][0] = 1;
A[0][1] = 1;
A[0][2] = 15
```

```
A[5][0] = 3;
A[5][1] = 4;
A[5][2] = -6;
```

```
/* Array representation of sparse matrix
  //[ ][0] represents row
  //[ ][1] represents col
  //[ ][2] represents value */
   int MAX = 8; //number of elements != 0 in sparse matrix
   int A[MAX][3];
```

Diagonal Matrix



- An ***n*** **x** ***n*** matrix in which all nonzero terms are on the diagonal.
  - M(i, j) is on diagonal iff $i = j$
  - number of diagonal elements in an n x n matrix is n
  - non diagonal elements are zero
- Store diagonal only vs n^2 whole:
  - `int M[5];`
  - `int M[5][5];`

Triangular Matrix

1 0 0 0

2 3 0 0

4 5 6 0

7 8 9 10

- An n x n matrix in which all nonzero terms are either on or below the diagonal.
  - M(i,j) is part of lower triangle iff $i >= j$
  - Number of elements in lower triangle is

$$1 + 2 + \ldots + n = n(n+1)/2$$

- Store n^2 whole vs only the lower triangle:
  - Store n^2 whole:
    - Use 2D array A[n][n]
  - Store only the lower triangle by:
    - Option 1: Map lower triangular into a 1D array
    - Option 2: Irregular 2D array

Option 1: Map Lower Triangular Array into a 1D array

Use row-major order, but omit terms that are not part of the lower triangle.

For the matrix

1 0 0 0
2 3 0 0
4 5 6 0
7 8 9 10

we get 1D array:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

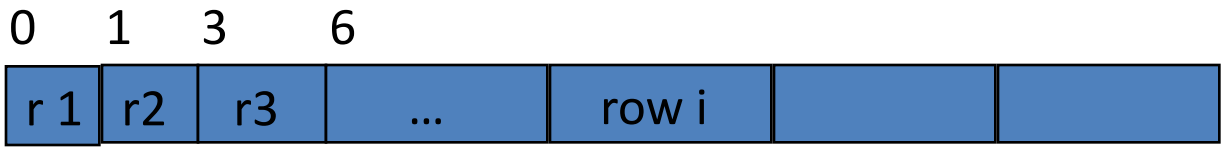
Index of element M(i,j) in 1D array

For the matrix M_{4x4}

1 0 0 0 → Row 1
2 3 0 0
4 5 6 0
7 8 9 10

we get 1D array:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10



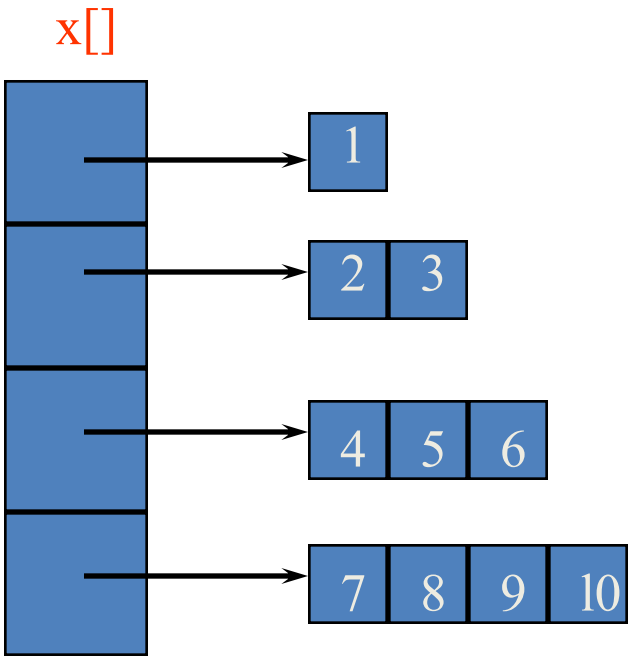
- Order is: row 1, row 2, row 3, …
  - Row *i* is preceded by rows 1, 2, …, *i*-1
- Size of row *i* (number of elements in row *i*) is *i*
- Number of elements that precede row *i* is

  $1 + 2 + 3 + \ldots + i\text{-}1 = i(i\text{-}1)/2$

➔ So element M(*i*, *j*) is at position $i(i\text{-}1)/2 + j\text{-}1$ of the 1D array

  - Example: M(3, 2) is at position $3(3\text{-}1)/2 + 2\text{-}1 = 4$

Option 2: Map Lower Triangular Array into Irregular 2D Arrays



Store only the lower triangle:

1 0 0 0

2 3 0 0

4 5 6 0

7 8 9 10

Irregular 2-D array: the length of rows is not required to be the same.

Creating and Using Irregular 2D Arrays

```
// STEP 1: declare a two-dimensional array variable
int ** iArray = new int*[numberOfRows];
OR:
 int ** iArray;
 malloc(iArray, numberOfRows*sizeof(*iArray));
//STEP 2: allocate the desired number of rows
// now allocate space for elements in each row
for (int i = 0; i < numberOfRows; i++)
    iArray[i] = new int [length[i]];
OR:
for (int i = 0; i < numberOfRows; i++)
        malloc(iArray[i], length[i]*sizeof(int));
// STEP 3: use the array like any regular array:
iArray[2][3] = 5;
iArray[4][6] = iArray[2][3]+2;
iArray[1][1] += 3;
```

Contents

2.1. Array

2.2. Record

2.3. Linked List

2.4. Stack

2.5. Queue

Record

- A record is an array where the elements have different types
- In C, the declaration of a variable of type record starts with the reserved word “struct”
- Then the type and the name of the fields in the record are defined
- Finally, the name of the variable record is given
- To assign a value to a field of a record, we must first name the record and then the field (record1.c)

```
int main(){
  struct {
    int num;
    int deno;
  }fraction;
    fraction.num = 13;
    fraction.deno = 17;
     printf("num %d, deno %d\n", fraction.num, fraction.deno);
}
```

Records: consecutive memory cells

- Show that record is made of consecutive memory cells like for arrays (record2.c)
- The difference is the fields have different types and names
- In a program we refer to those fields using names but actually the computer find the addr of cells in those fields in the same way as for arrays
- To find the addr of field x
  - Addr of the first cell of the record + number of bytes used by all the fields before x (number of bytes depend on the type of each field)

```
int main(){
  struct {
    int num;
    int deno;
  }fraction;
 printf("addr fraction %8u, addr num %8u, addr deno
%8u\n",&fraction,&fraction.num,&fraction.deno);
}
```

A second example of record (record3.c)

```
int main(){
  struct {
    int id;
    char* name; /*string of characters*/
    char grade;
  }student;


  printf("addr student %8u, addr id %8u, addr name  %8u,  addr grade
%8u\n",&student,&student.id,&student.name,&student.grade);

  student.id = 2021;
  student.name = "Big-X";
  student.grade= 'A';

  printf("id %d, name %s,  grade  %c\n",student.id, student.name, student.grade);
```

An array of records

- Print the addr of the first cell of each record (record4.c)
- Print the addr of the second and third field of each record
- Assig values to different fields and different records and print the assigned values

```
int main(){
  struct {
    int id;
    char* name;
    char grade;
  }student[3];
  printf("addr of student records, student[0] %8u, student[1] %8u, student[2]
%8u\n\n",&student[0],student[1],student[2]);
  printf("addr of fields in student[0], name %8u,  grade  %8u\n\n",&student[0].name,&student[0].grade);
  printf("addr of fields in student[1], name %8u,  grade  %8u\n\n",&student[1].name,&student[1].grade);
  printf("addr of fields in student[2], name %8u,  grade  %8u\n\n",&student[2].name,&student[2].grade);
  student[0].id = 2021;
  student[1].name = "Big-X";
  student[2].grade= 'A';
  printf("id %d, name %s,  grade  %c\n",student[0].id, student[1].name, student[2].grade);
}
```

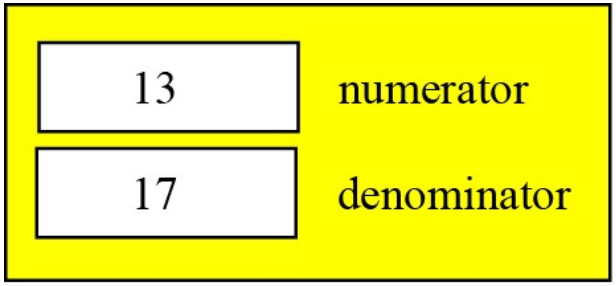
Typedef & dynamic memory allocation for rec

- Now student is a type (not a variable), does not have a memory addr (record5.c)
- Minh is a pointer, will store the addr of the first cell of a record of type student
- malloc allocate memory cells for a data structure of type student
- Print the addrs of Minh and the fields in the record. Assign values to the fields and print those values
- Since Minh is a pointer, we must reference the fields of the object to which it points using "->"

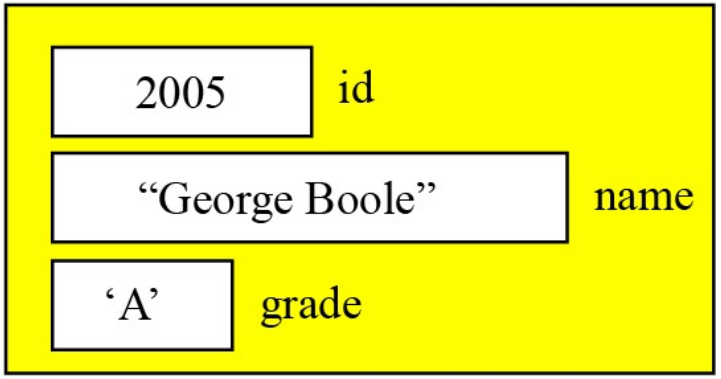
```
int main(){
  typedef struct {
    int id;
    char* name;
    char grade;
  }student;
  student* Minh;
  Minh = (student*)malloc(sizeof(student));

  printf("addr Minh %8u  addr Minh id %8u, addr Minh name %8u, addr Minh grade
%8u\n\n",&Minh,&Minh->id,&Minh->name,&Minh->grade);

  Minh->id = 2021;
  Minh->name = "Minh";
  Minh->grade= 'A';
  printf("id %d, name %s,  grade  %c\n",Minh->id, Minh->name, Minh->grade);
}
```

2.2. Record

- A record is a collection of related elements, possibly of different types, having a single name.
- Each element in a record is called a field:
  - A field has a type and exists in memory.
  - Fields can be assigned values, which in turn can be accessed for selection or manipulation.
- Example: Figure below contains two examples of records.
  - The first example: fraction has two fields: numerator and denominator, both of which are integers.
  - The second example: student has three fields (id, name, grade) made up of three different types.



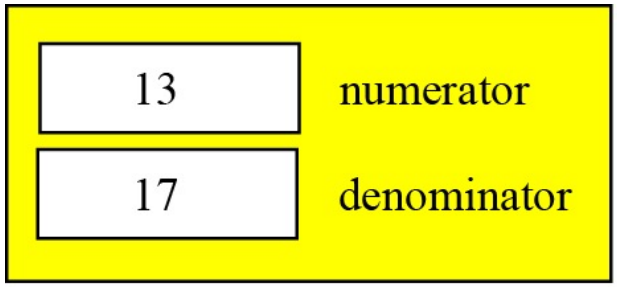
fraction



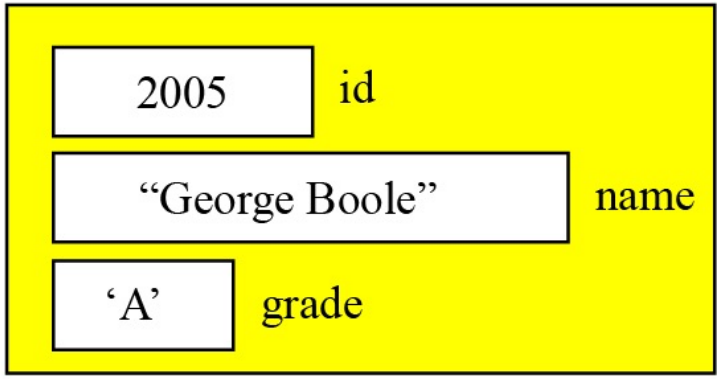
student

2.2. Record

- Example:



fraction



student

```
struct  {
    int numerator;
    int denominator;
} fraction;
fraction.numerator = 13;
fraction.denominator = 17;
```

```
struct  {
    int id;
    char* name;
    char grade;
} student;
student.id = 2005;
student.name = "George Boole";
student.grade = 'A';
```

Record name vs. field name

Just like in an array, we have two types of identifier in a record:

- the name of the record, and
- the name of each individual field inside the record.

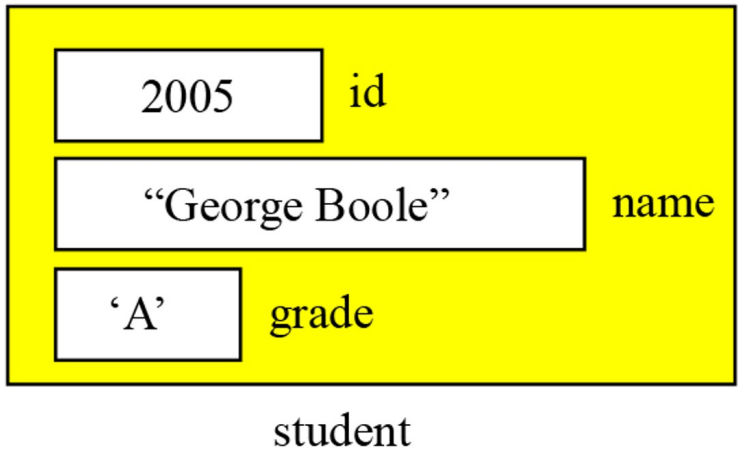
The name of the record is the name of the whole structure, while the name of each field allows us to refer to that field.

Example: in the student record:

- the name of the record is student,
- the name of the fields are student.id, student.name and student.grade.

Most programming languages use a *period* (.) to separate the name of the structure (record) from the name of its components (fields).

```
struct  {
    int id;
    char* name;
    char grade;
} student;
student.id = 2005;
student.name = "George Boole";
student.grade = 'A';
```



Comparison of records and arrays

We can compare an array with a record. This helps us to understand when we should use an array and when to use a record:

- An array defines a combination of elements, while a record defines the identifiable parts of an element.
- For example, an array can define a class of students (40 students), but a record defines different attributes of a student, such as id, name or grade.
- Array of records: If we need to define a combination of elements and at the same time some attributes of each element, we can use an array of records. For example, in a class of 30 students, we can have an array of 30 records, each record representing a student.

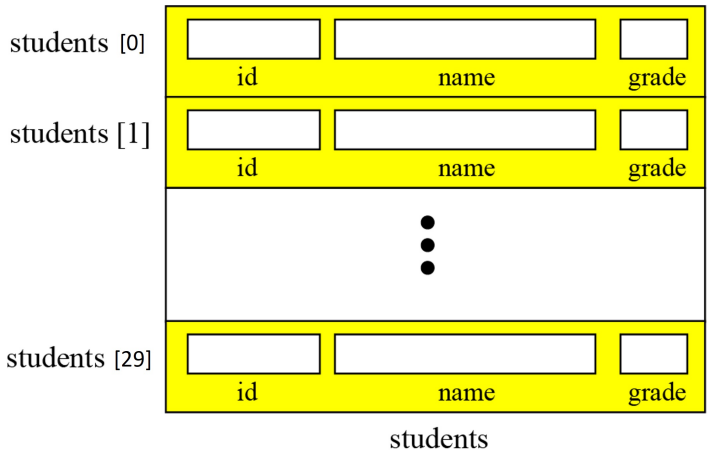


Figure 1. Array of records

```
struct  {
    int id;
    char* name;
    char grade;
} students[30];
```

```
students[0].id = 1001;
students[0].name = "J.Aron";
students[0].grade = 'A';

students[1].id = 1002;
students[1].name = "F.Bush";
students[1].grade = 'F';
…..
students[29].id = 3021;
students[29].name = "M Blair";
students[29].grade = 'B';
```

Contents

2.1. Array

2.2. Record

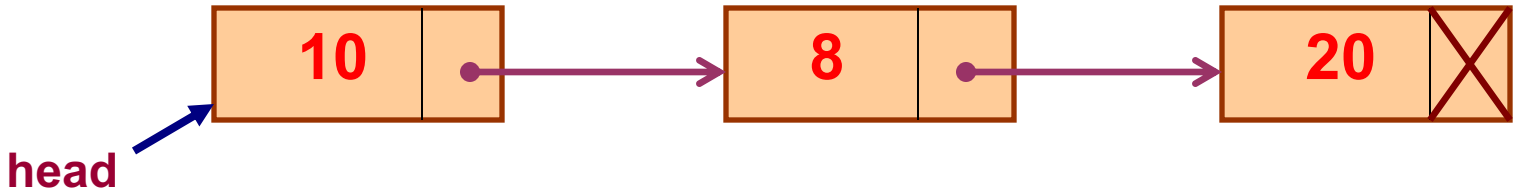
2.3. Linked List

2.4. Stack

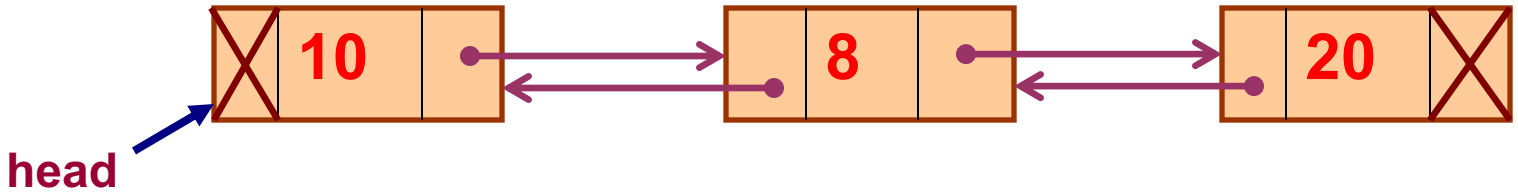
2.5. Queue

2.3. Linked list

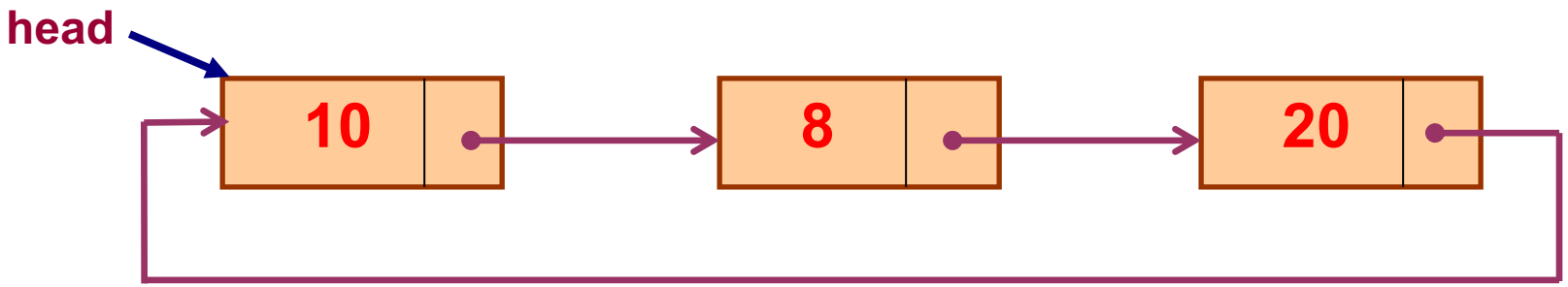
- Singly linked list



- Doubly linked list



- Circular linked list



Create the first record of type node

- Define a type record name node. The second field is a pointer to an object of type node
- Declare a variable “head” that is a pointer to an object of type node
- Then dynamically allocate memory for a record of type node. The addr of the first cell of this object is stored in head
- Print the addr of head, of fields data and next (linklist1.c)

```
int main(){
  typedef struct {
    int data;
    struct node* next;
  }node;
  node* head;
  head = (node*)malloc(sizeof(node));

  printf("addr head %8u  addr head data %8u, addr head next %8u\n\n",head,&head->data,&head->next);
}
```

Create a second record of type node

- (linklist2.c)

```
int main(){
  typedef struct {
    int data;
    struct node* next;
  }node;
  node* head;
  node* secondNode;
  head = (node*)malloc(sizeof(node));
  secondNode = (node*)malloc(sizeof(node));

  printf("addr head %8u  addr head data %8u, addr head next %8u\n\n",head,&head->data,&head->next);
 printf("addr secondNode %8u  addr secondNode data %8u, addr secondNode next %8u\n\n",secondNode,&secondNode->data,&secondNode->next);
```

Create a link list of 2 nodes

- A link list of two nodes is generated (linklist3.c)
- Assign memory addresses for two object of type node
- Connect the first node to the second one: head->next = secondNode;
- i.e., place the addr of the second node into the next field of the first node
- The link list starts with the head pointer which points to the addr of the first node
- The link list ends with the next pointer of secondNode which is NULL

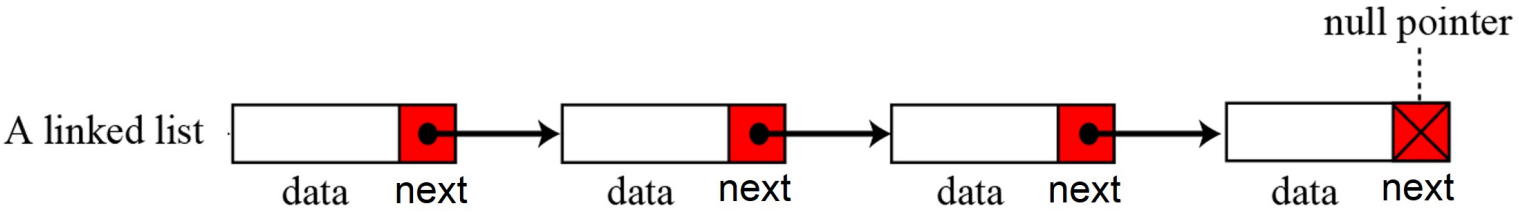
```
  node* head;
  node* secondNode;
  head = (node*)malloc(sizeof(node));
  secondNode = (node*)malloc(sizeof(node));

  head->data = 1;
  head->next = secondNode;
  secondNode->data = 2;
  secondNode->next = NULL;
}
```

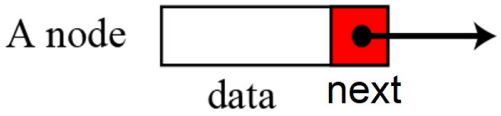
Add a third node to the link list (linklist4.c)

```
node* head;
node* secondNode;
node* thirdNode;

head = (node*)malloc(sizeof(node));
secondNode = (node*)malloc(sizeof(node));
thirdNode = (node*)malloc(sizeof(node));


head->data = 1;
head->next = secondNode;
secondNode->data = 2;
secondNode->next = thirdNode;
thirdNode->data = 3;
thirdNode->next = NULL;
```

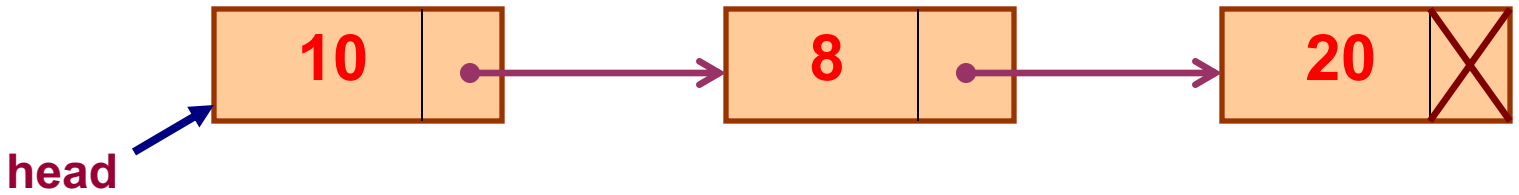
Singly Linked list

- A singly linked list is a sequences of nodes, each node contains 2 parts: data and reference (address) to the next node.
- Example: Figure shows a singly linked list contains four nodes:



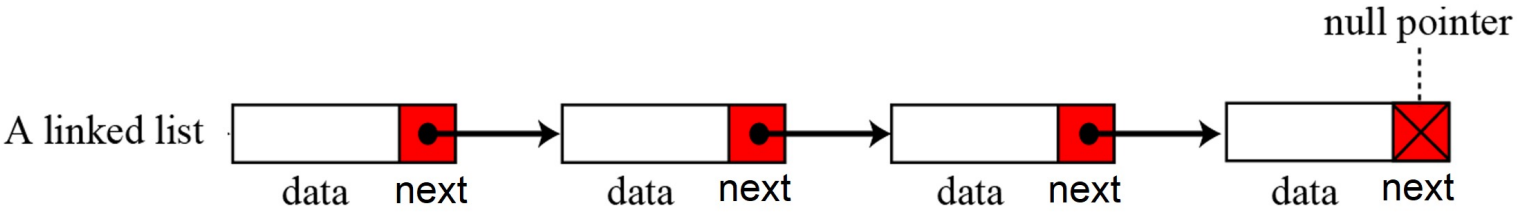


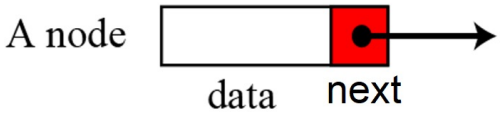
- Keeping track of a singly linked list:
  - Must know the pointer to the first element of the list (called *start*, *head*, etc.)
  - If head is NULL, the singly linked list is empty



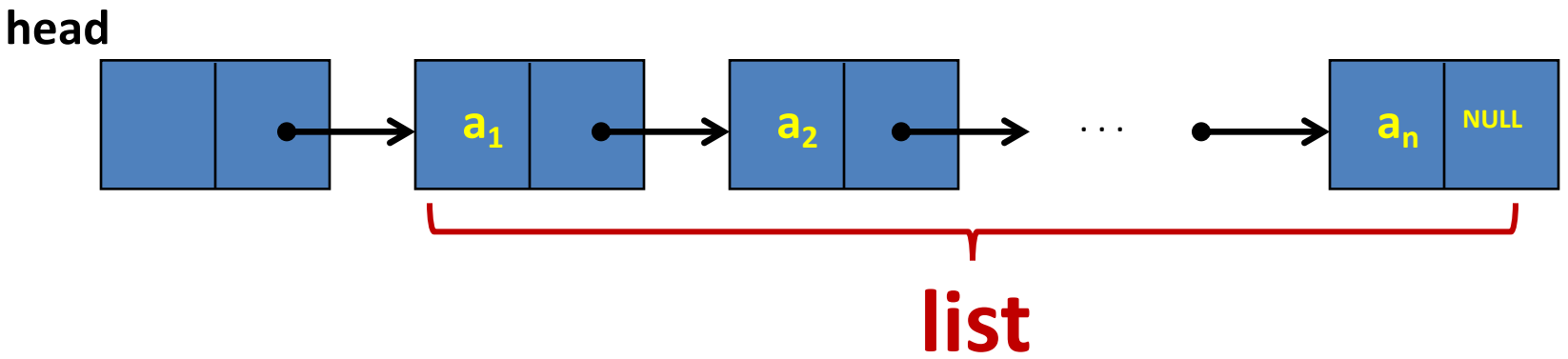
Singly Linked list

- A singly linked list is a sequences of nodes, each node contains 2 parts: data and reference (address) to the next node.
- Example: Figure shows a singly linked list contains four nodes:





- Keeping track of a singly linked list:
  - Must know the pointer to the first element of the list (called *start*, *head*, etc.)
  - If head is NULL, the singly linked list is empty



Singly Linked list

- Before further discussion of singly linked lists, we need to explain the notation we use in the figures. We show the connection between two nodes using a line. One end of the line has an arrowhead, the other end has a solid circle.

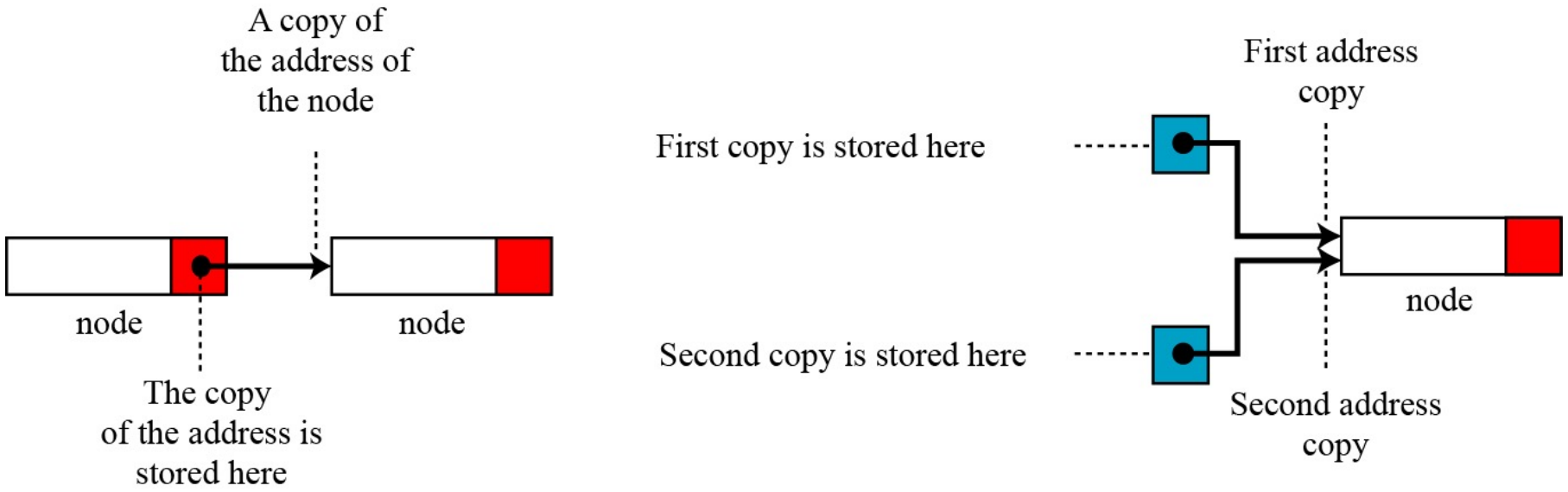
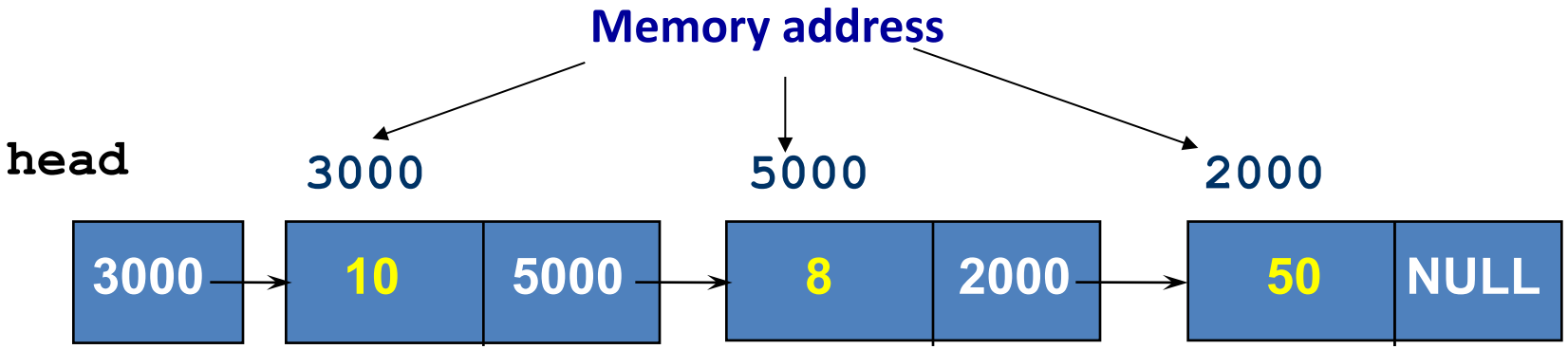
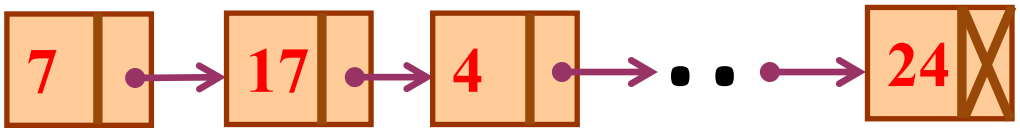


Figure. The concept of copying and storing pointers

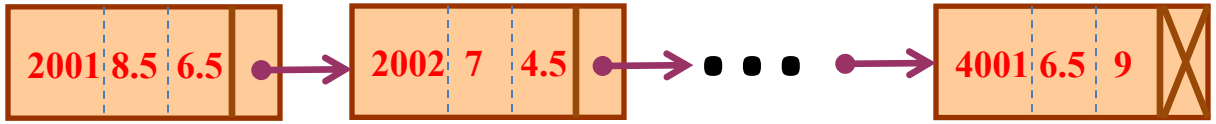


Declare singly linked list in C programming language

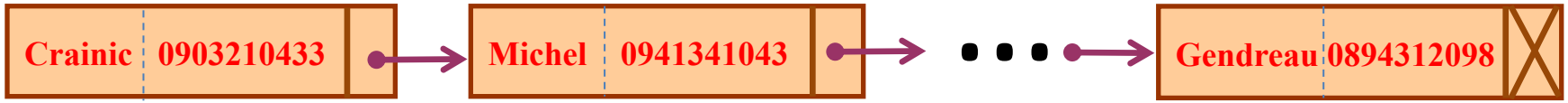
- List of integer numbers:



- List of students with data: student's ID, grade of math and physics



- List of contacts with data: name, phone number



➔ Need to declare:

- the type of data in the node first,
- then the singly linked list consists of (1) data of the node, and (2) the pointer to store the address of the next node in the list

Declare singly linked list

Need to declare:

- the type of data in the node first,
- then the linked list consists of (1) data of the node, and (2) the pointer to store the address of the next node in the list

```
typedef struct {
    .....
}NodeType;

typedef struct {
    NodeType data;
    struct node* next;
}node;
node* head;
```

Define the type of data of the node

Define the singly linked list

This declaration define `node` which is a record consisting of 2 fields:

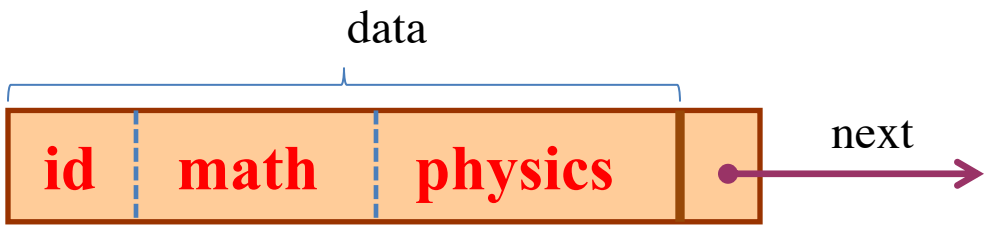
- `data` : stores data of node, has the type `NodeType` (which was defined in `typedef…NodeType`, and could consist of several attributes)
- `next` : the pointer which stores the address of the next node in the list

Pointer `head` : store address of the first node in the list

Example1: List of students with data: id of student, marks of 2 subjects: math, physics

```
typedef struct{
    char id[15];
    float math, physics;
}student;

typedef struct {
    student data;
    struct node* next;
}node;
node* head;
```

Define the type of data of the node



A node

Declare singly linked list

```
typedef struct {
    .....
}NodeType;

typedef struct {
    NodeType data;
    struct node* next;
}node;
node* head;
```

Define the type of data of the node

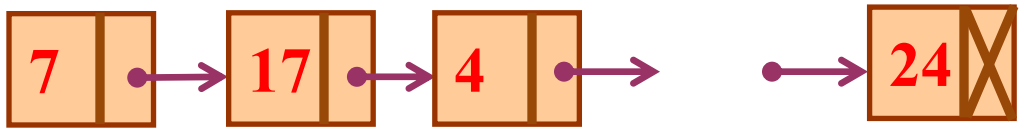
Define the singly linked list

This declaration define `node` which is a record consisting of 2 fields:

- `data` : stores data of node, has the type `NodeType` (which was defined in `typedef…NodeType`, and could consist of several attributes)
- `next` : the pointer which stores the address of the next node in the list

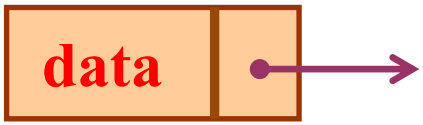
Pointer `head` : store address of the first node in the list

Example2: List of integer numbers




```
typedef struct {
    int data;
    struct node* next;
}node;
node* head;
```

"`int`" is the type of node, so do not need to use "`typedef….NodeType`" to define the type



A node

Declare singly linked list

Need to declare:

- the type of data in the node first,
- then the linked list consists of (1) data of the node, and (2) the pointer to store the address of the next node in the list

```
typedef struct {
    .....
}NodeType;

typedef struct {
    NodeType data;
    struct node* next;
}node;
node* head;
```

Define the type of data of the node

Define the singly linked list

This declaration define `node` which is a record consisting of 2 fields:

- `data` : stores data of node, has the type `NodeType` (which was defined in `typedef…NodeType`, and could consist of several attributes)
- `next` : the pointer which stores the address of the next node in the list

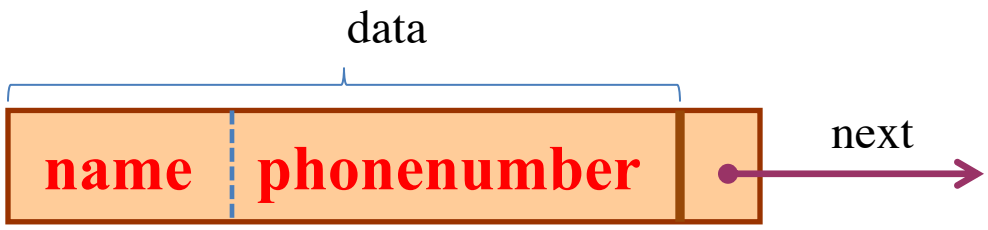
Pointer `head` : store address of the first node in the list

Example 3: List of contacts with data: name and phone number

```
typedef struct{
    char name[15];
    char phone[20];
}contact;
```

Define the type of data of the node

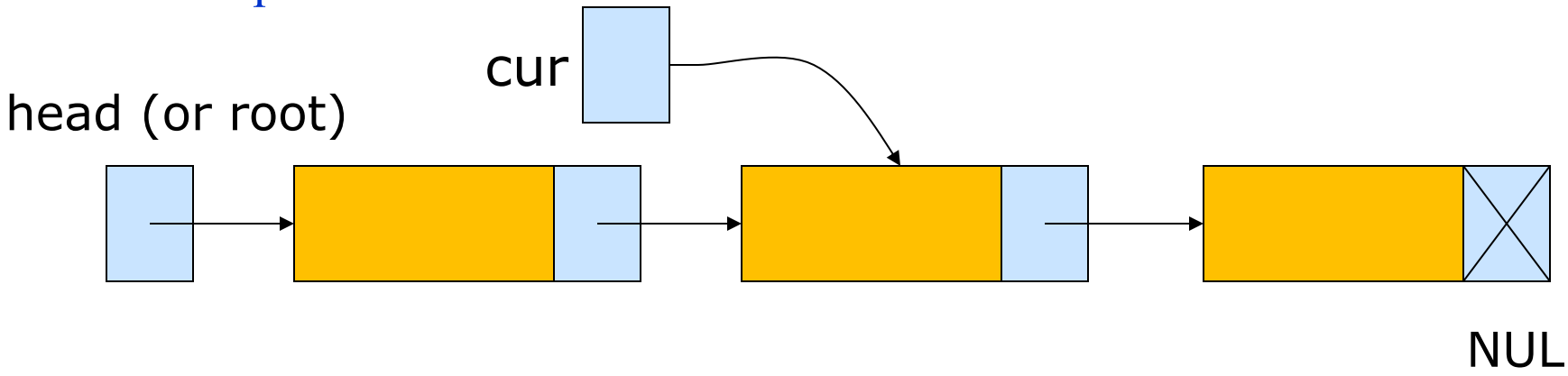
```
typedef struct {
    contact data;
    struct node* next;
}node;
node* head;
```



A node

Important elements of singly linked list

- head: store the address of the first node in the linked list
- NULL: value of the pointer of the last node in the linked list
- cur: the pointer stored the address of current node



- Allocate memory for a new node pointed by the pointer `newNode` in the list:

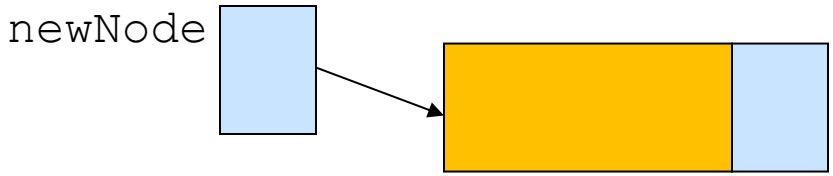
```
node *newNode = (node *) malloc(sizeof(node));
```

- Access to the data of the node pointed by pointer `newNode` :

```
newNode->data
```

- Free memory allocated for node pointed by pointer `newNode` :

```
free(newNode);
```



Operations on singly linked Lists

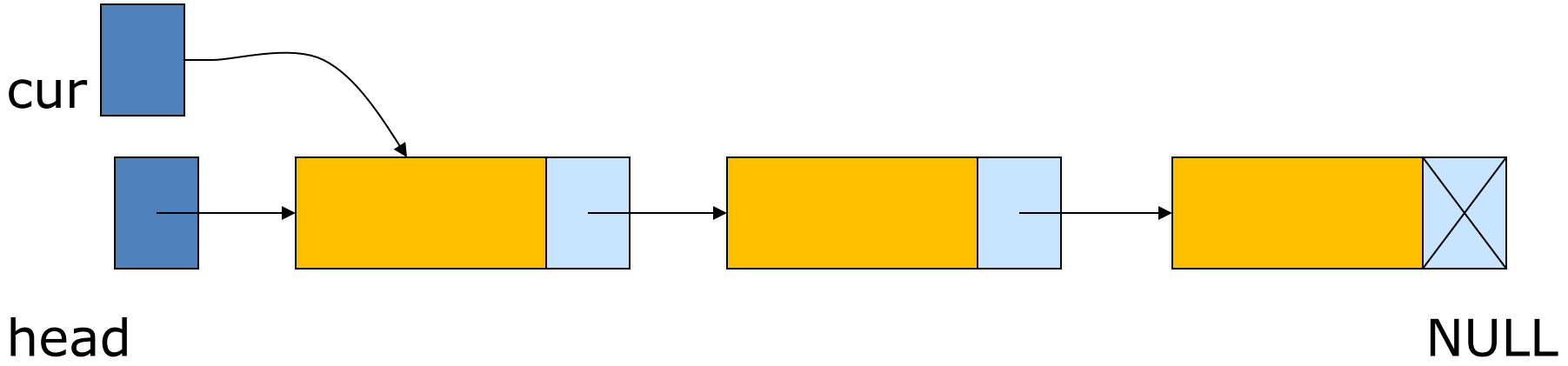
- Traverse the singly linked list
- Insert a node into the singly linked list
- Delete a node from the singly linked list
- Search data in the singly linked list

Operations on singly linked Lists

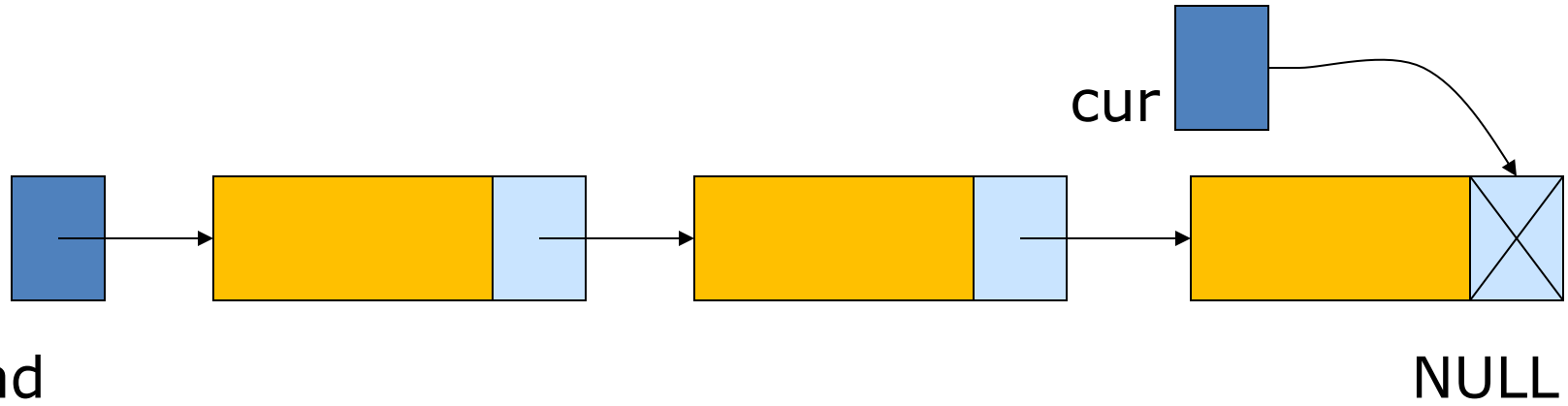
- **Traverse the singly linked list**
- Insert a node into the singly linked list
- Delete a node from the singly linked list
- Search data in the singly linked list

Traversing a singly linked list

```
for ( cur = head; cur != NULL; cur = cur->next )
        showData_Of_Current_Node( cur->data );
```



- Change the value of the pointer cur
- Finish to browse the list when the NULL value is encountered



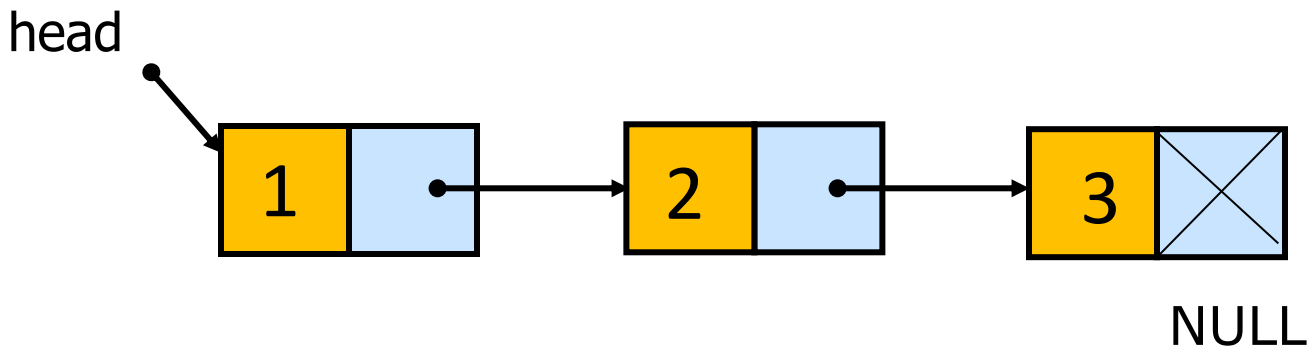
Exercise 1

- A sequence of integers is stored by a singly linked list.

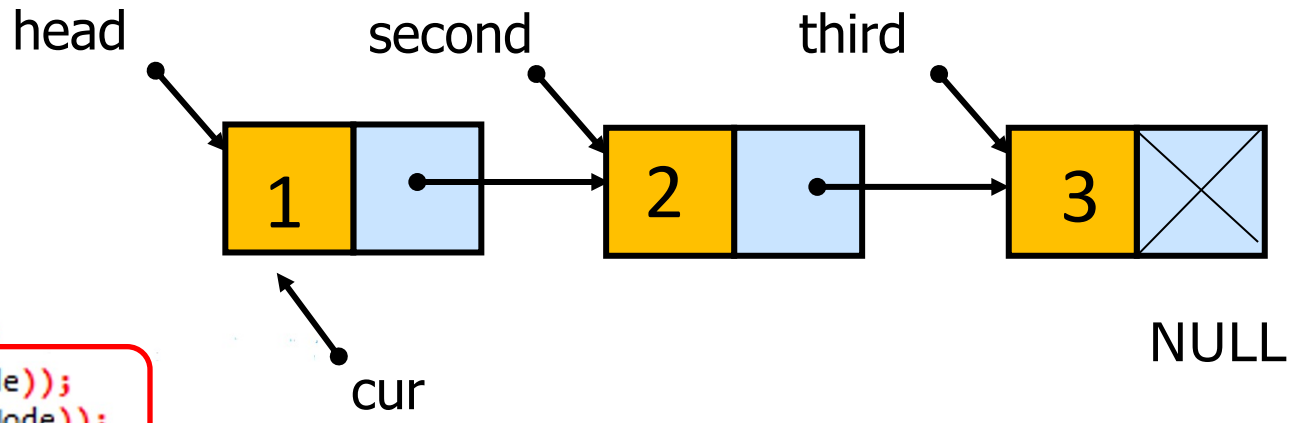
```
typedef struct Node{
    int data;
    struct Node *next;
}Node;
Node *head;
```

a. Create a list stored 3 integers: 1, 2, 3

b. Print the list of these 3 integers



```c
#include<stdio.h>

typedef struct
{
  int data;
  struct Node *next;
}Node;

void showData_Of_Current_Element(int data){
   printf("Data = %d\n",data);
}

int main()
{
  struct Node* head;
  Node* second = NULL;
  Node* third = NULL;

  //allocate memory for 3 nodes in the heap
  head = (Node*)malloc(sizeof(Node));
  second = (Node*)malloc(sizeof(Node));
  third = (Node*)malloc(sizeof(Node));

  head->data = 1; //assign data for the first node in the list
  head->next = second; //connect the first node to the second node

  second->data = 2; //assign data for the 2nd node in the list
  second->next = third; //connect the 2nd node to the 3rd node

  third->data = 3; //assign data for the 3rd node
  third->next = NULL;

  Node* cur;
  for (cur = head; cur != NULL; cur = cur->next)
     showData_Of_Current_Element(cur->data);
  return 0;
}
```



Data = 1

Data = 2

Data = 3

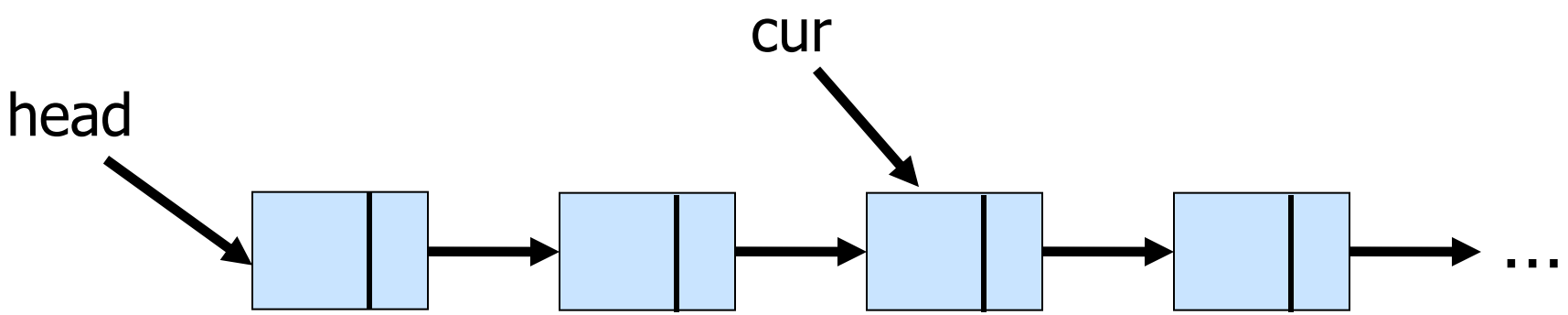
Operations on singly linked lists

- Traverse the singly linked list
- **Insert a node into the singly linked list**
- Delete a node from the singly linked list
- Search data in the singly linked list

Operations on singly linked list: Insertion

Insert a new node :

- At the beginning of the list
- After the position pointed by the pointer cur
- Before the position pointed by the pointer cur
- At the end of the list

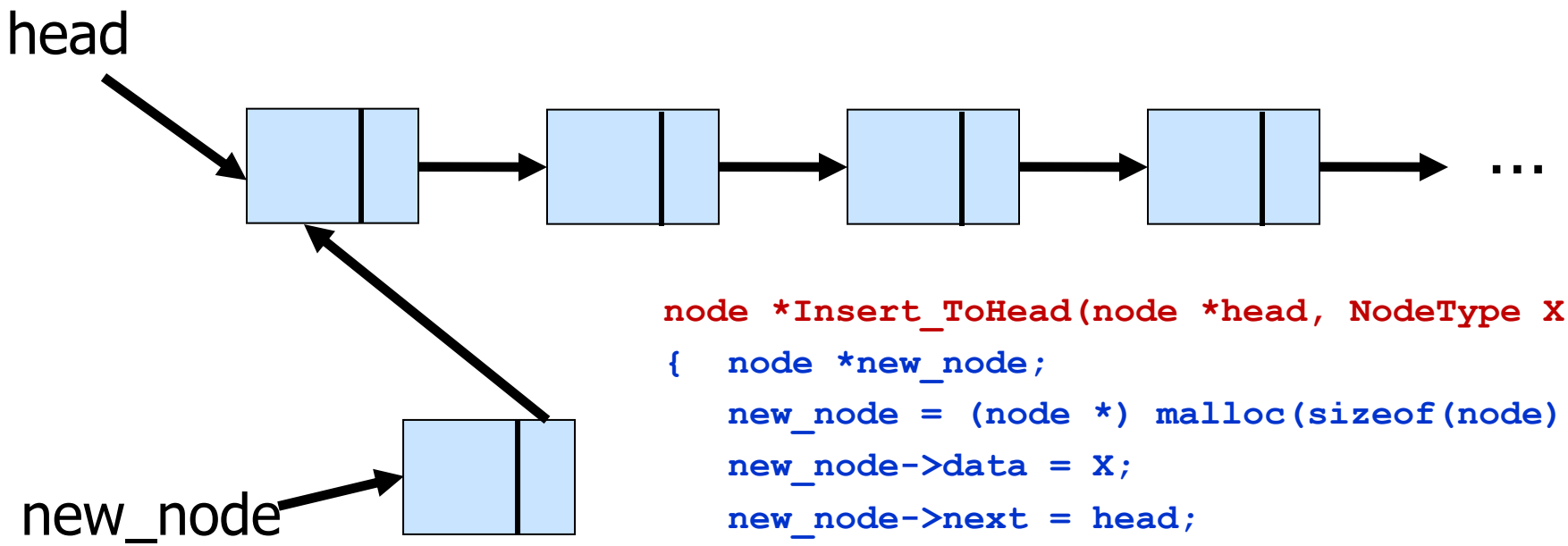


Operations on singly linked list: Insertion

Insert a new node:

- At the beginning of the list

```
<create a new node new_node>;
new_node ->next = head;
head= new_node;
```

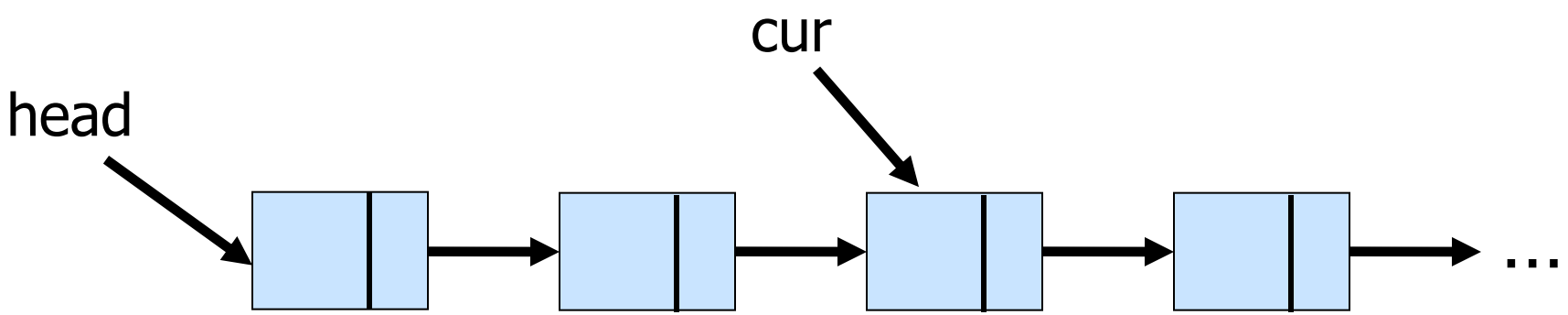



```
node *Insert_ToHead(node *head, NodeType X)
{   node *new_node;
    new_node = (node *) malloc(sizeof(node));
    new_node->data = X;
    new_node->next = head;
    head=new_node;
    return head;
}
```

Operations on singly linked list: Insertion

Insert a new node :

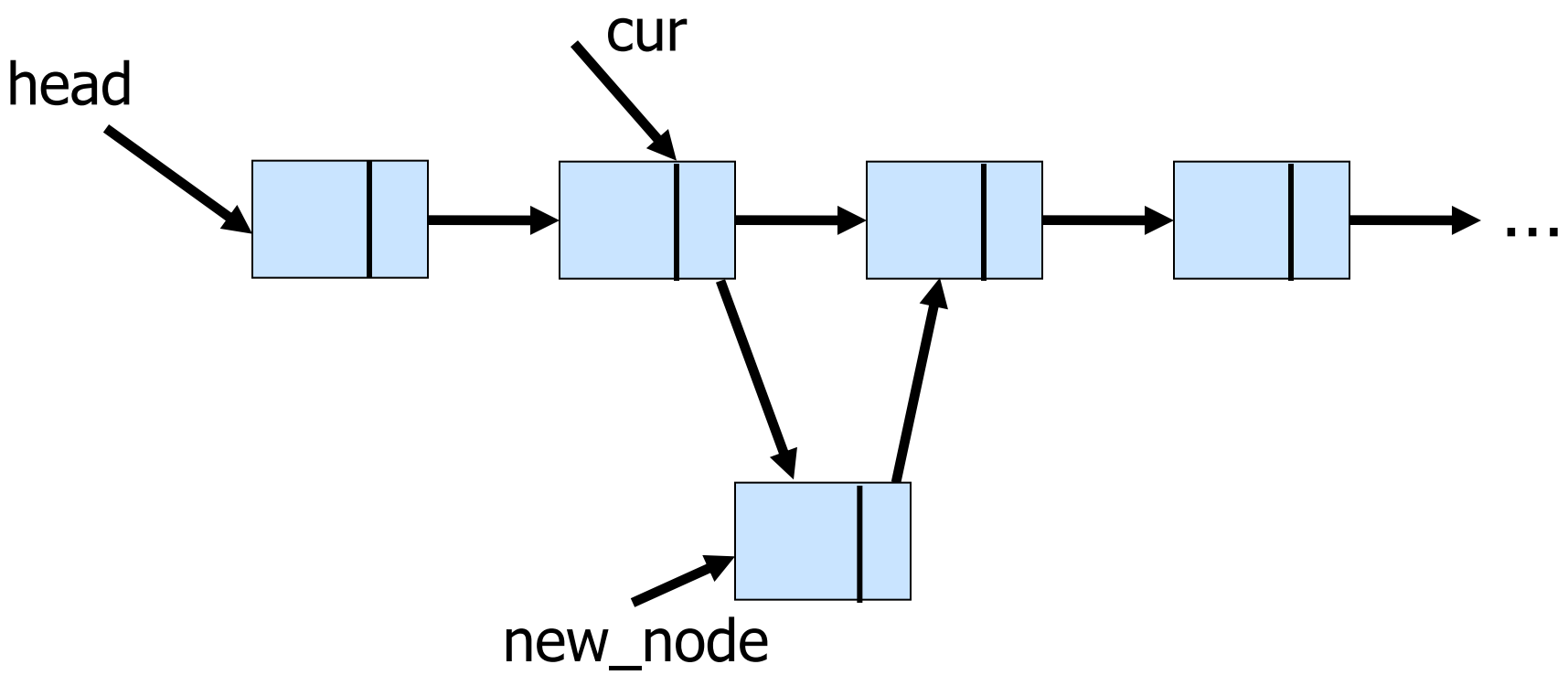
- At the beginning of the list
- **After the position pointed by the pointer cur**
- Before the position pointed by the pointer cur
- At the end of the list



Operations on singly linked list: Insertion

- Insert a new node after the node pointed by the pointer cur:

```
<create a new node new_node>;
new_node ->next = cur->next;
cur->next = new_node;
```



Operations on singly linked list: Insertion

- Insert a new node after the node pointed by the pointer cur:

```
<create a new node new_node>;
new_node ->next = cur->next;
cur->next = new_node;
```

Write a function to insert a node with data = *X* (having the type «NodeType » after the node pointed by the pointer cur. The function returns the address of the new node:

```
node *Insert_After(node *cur, NodeType X)
{       node *new_node;
        new_node = (node *) malloc(sizeof(node)); //(1)
        new_node -> data = X;                     //(1)
        new_node->next = cur->next;               //(2)
        cur->next = new_node;                     //(3)
        return new_node;
}
```

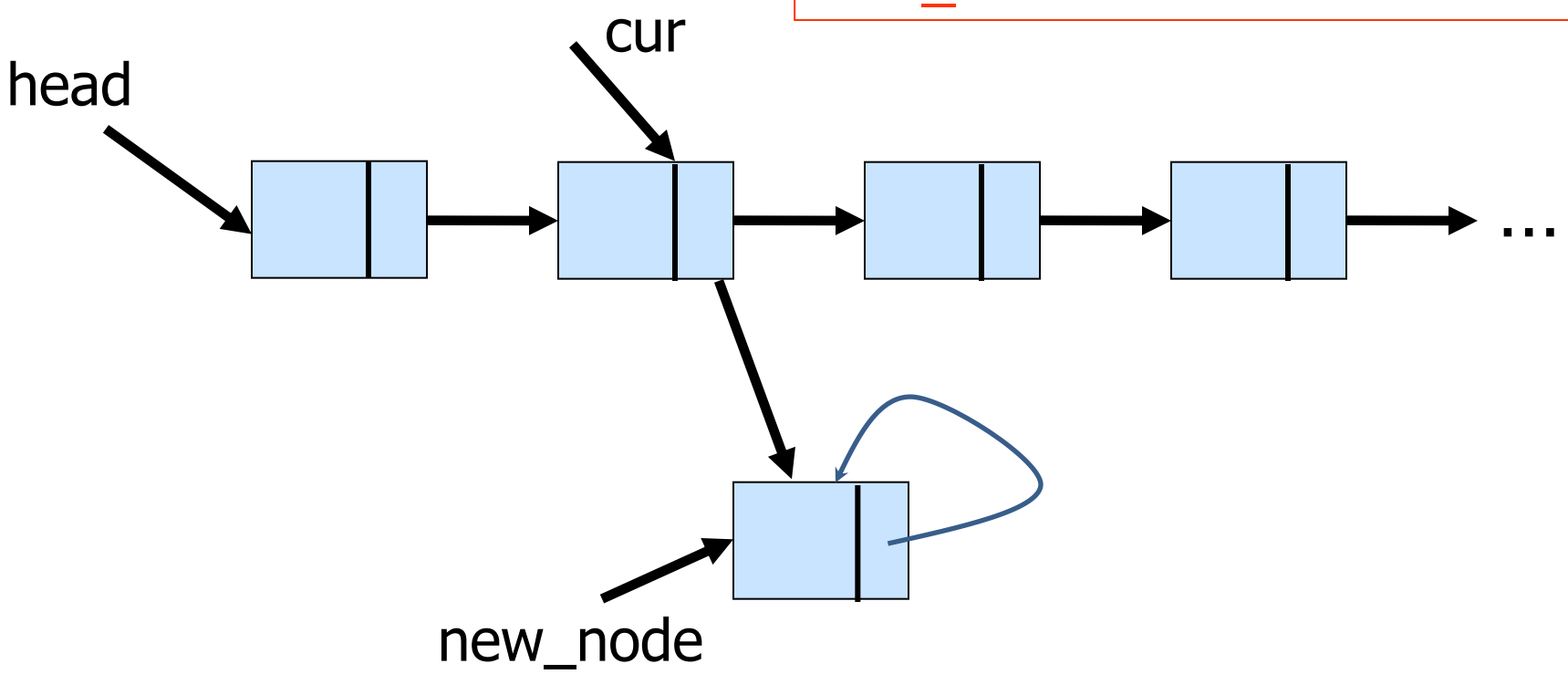
Operations on singly linked list: Insertion

- Insert a new node after the node pointed by the pointer cur:

```
<create a new node new_node>;
new_node ->next = cur->next;
cur->next = new_node;
```

?? Empty list

```
// wrong implementation:
cur->next = new_node;
new_node ->next = cur->next;
```



Operations on singly linked list: Insertion

- Insert a new node after the node pointed by the pointer cur:

```
<create a new node new_node>;
new_node ->next = cur->next;
cur->next = new_node;
```

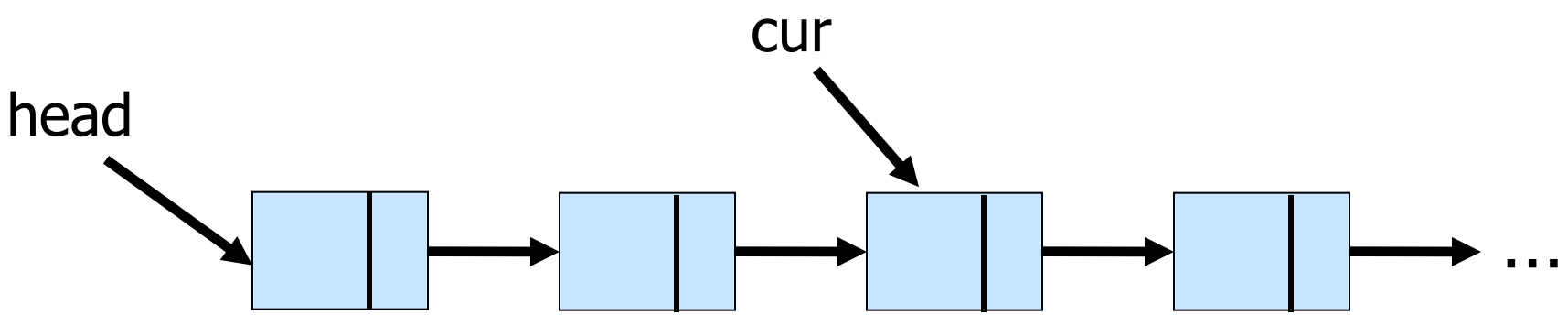
?? Empty list

```
<create a new node new_node>;
if (head == NULL) { /* list does not have any node yet */
     head = new_node;
     cur = head;
}
else {
    new_node ->next = cur->next;
    cur->next = new_node;
}
```

Operations on singly linked list: Insertion

Insert a new node :

- At the beginning of the list
- After the position pointed by the pointer cur
- **Before the position pointed by the pointer cur**
- At the end of the list



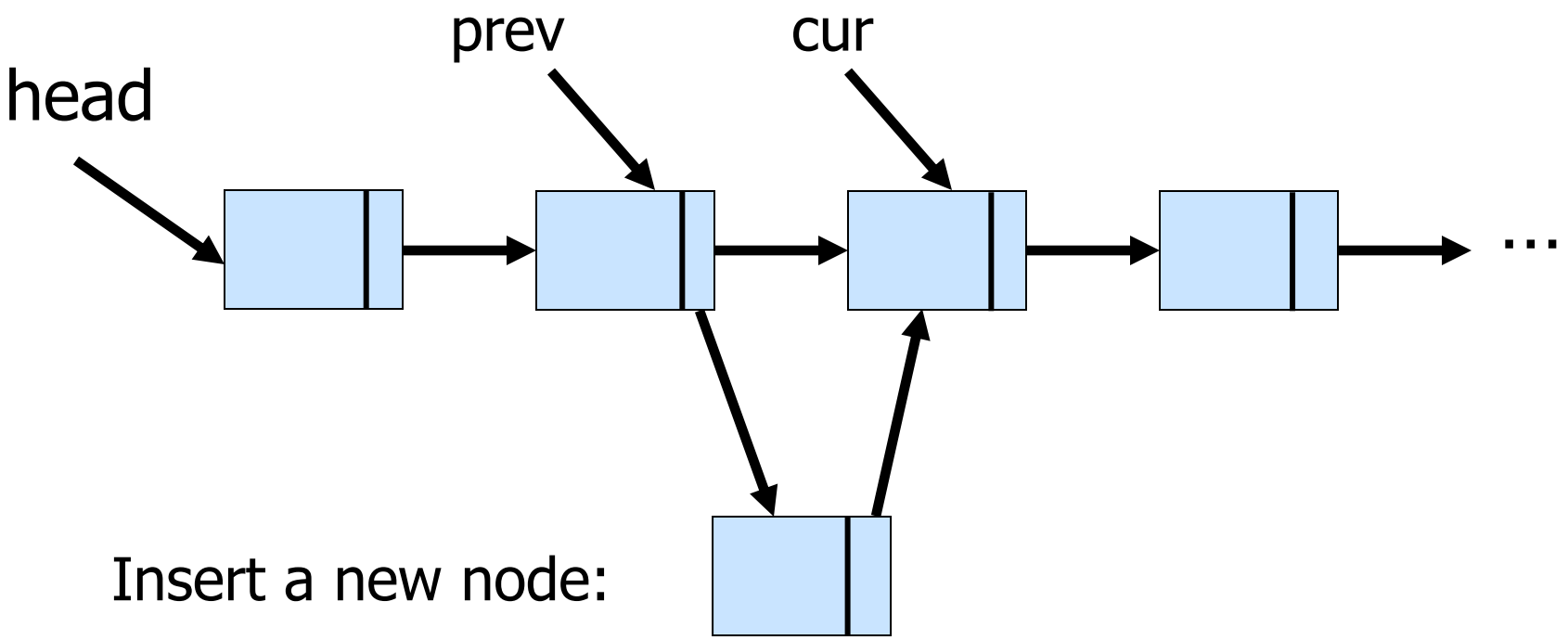
Operations on singly linked list: Insertion

Insert a new node **before the node pointed by the pointer cur**

```
<create a new node new_node>;
prev->next = new_node;
new_node>next = cur;
```

?? List does not have any node yet

?? cur is the first node in the list



Operations on singly linked list: Insertion

Insert a new node **before the node pointed by the pointer cur**

```
<create a new node new_node>;
prev->next = new_node;
new_node->next = cur;
```

?? List does not have any node yet

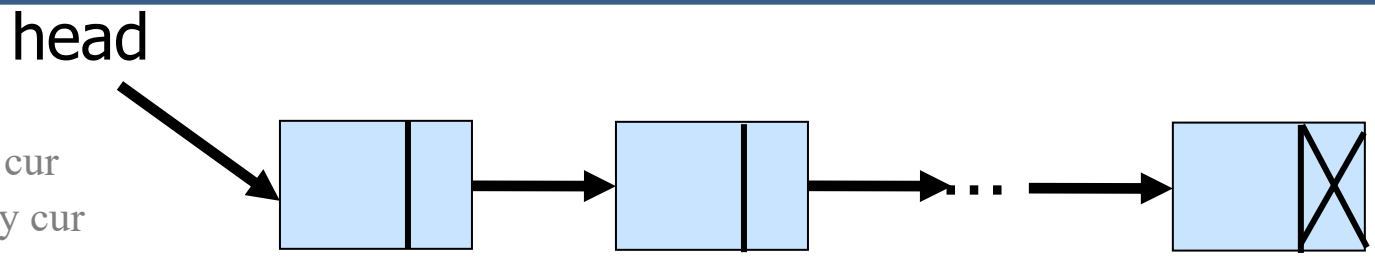
?? cur is the first node in the list

```
<create a new node new_node>;
if (head == NULL) { /* list does not have any node yet */
    head = new_node;
    cur = head;
}
else if (cur == head) { //cur us the first node in the list
    head = new_node;
    new_node->next = cur;
}
else {
   prev->next = new_node;
   new_node->next = cur;
}
```

Operations on singly linked list: Insertion

Insert a new node:

- At the beginning
- After the node pointed by cur
- Before the node pointed by cur
- **At the end of the list**




```
<create a new node new_node>;
if (head == NULL) { /* list does not have any node yet*/
    head = new_node;
}
else {
    //move the pointer to the end of the list:
     node *last =head;
     while (last->next != NULL) last = last->next;
     //Change the pointer next of the last node:
   last->next = new_node;
}
```

Complexity is ….

```
node *Insert_ToLast(node *head, NodeType X)
{   node *new_node;
    new_node = (node *) malloc(sizeof(node));
    new_node->data = X;
    if (head == NULL) head = new_node;
    else
    {
        node *last;
        last=head;
        while (last->next != NULL)   // move to the last node
            last = last->next;
        last->next = new_node;
    }
    return head;
}
```

Operations on singly linked Lists

- Traverse the singly linked list
- Insert a node into the singly linked list
- **Delete a node from the singly linked list**
- Search data in the singly linked list

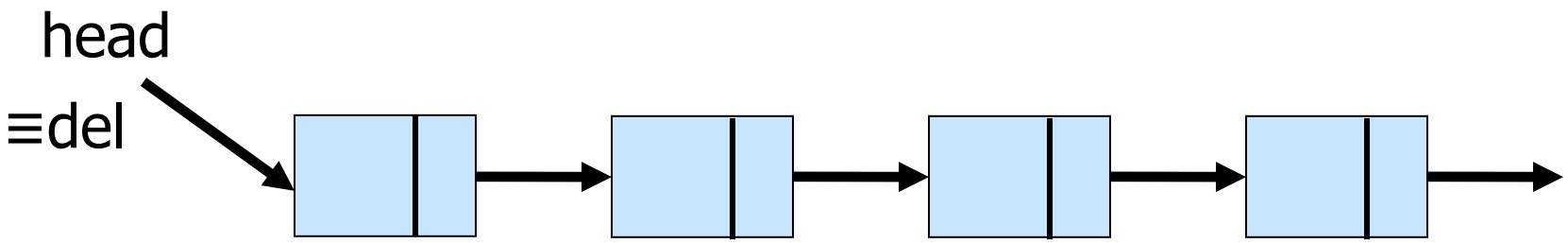
Operations on singly linked lists: Deletion

- **Delete a node**
- Delete all nodes of the list

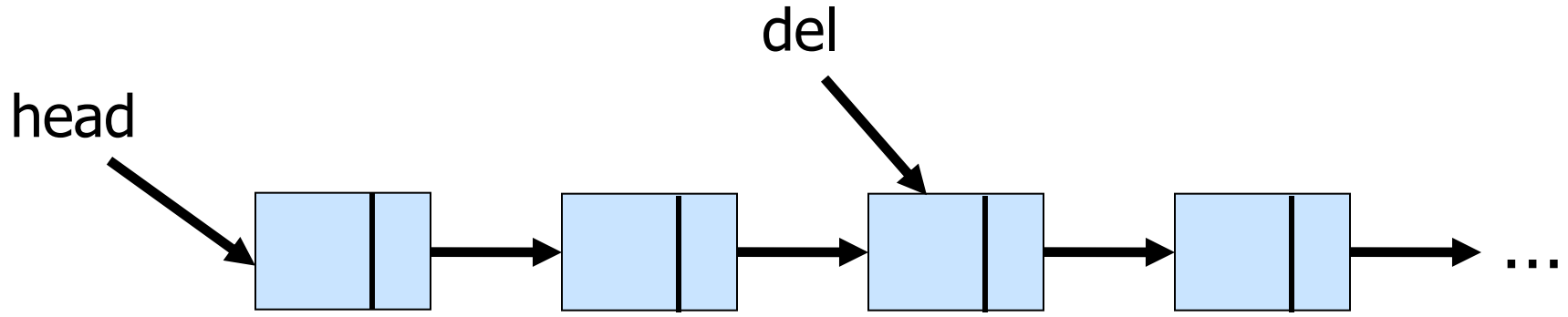
Operations on singly linked lists: Deletion

Delete a node:

- The first node of the list



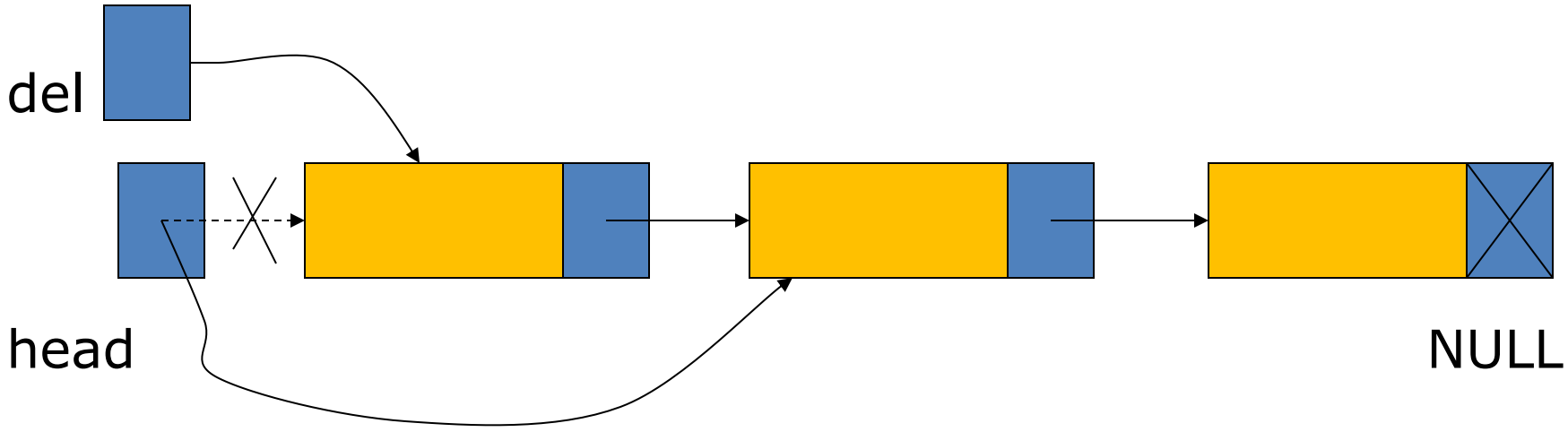
- Middle/last node of the list



Delete the first node of the list

- Delete the node `del` that is currently the first node of the list:

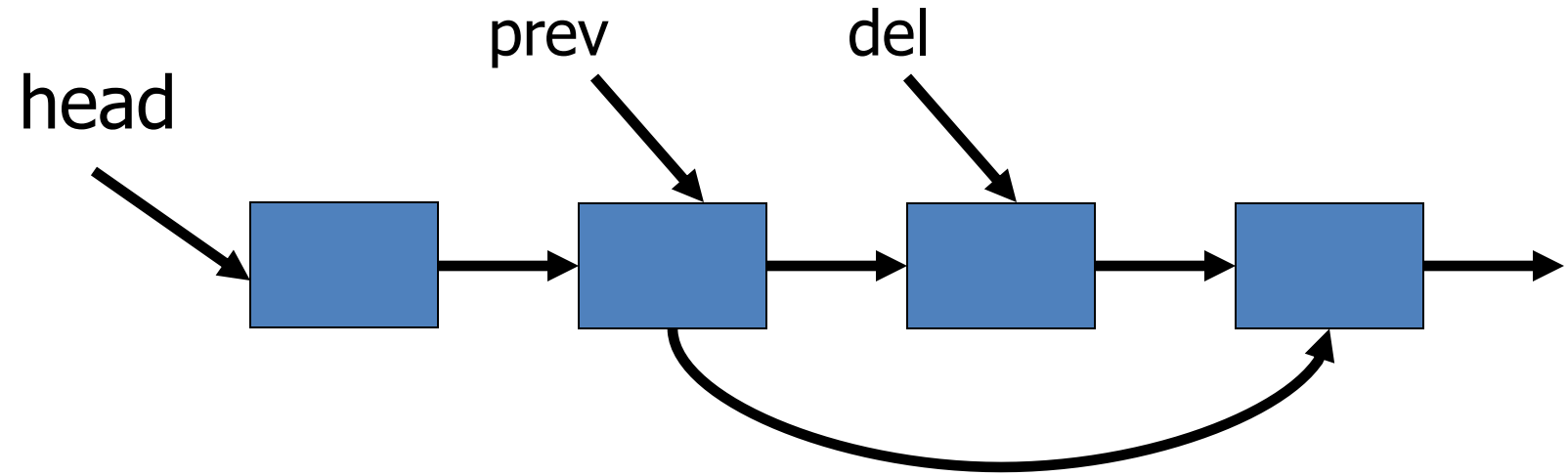
```
head = del->next;
free(del);
```

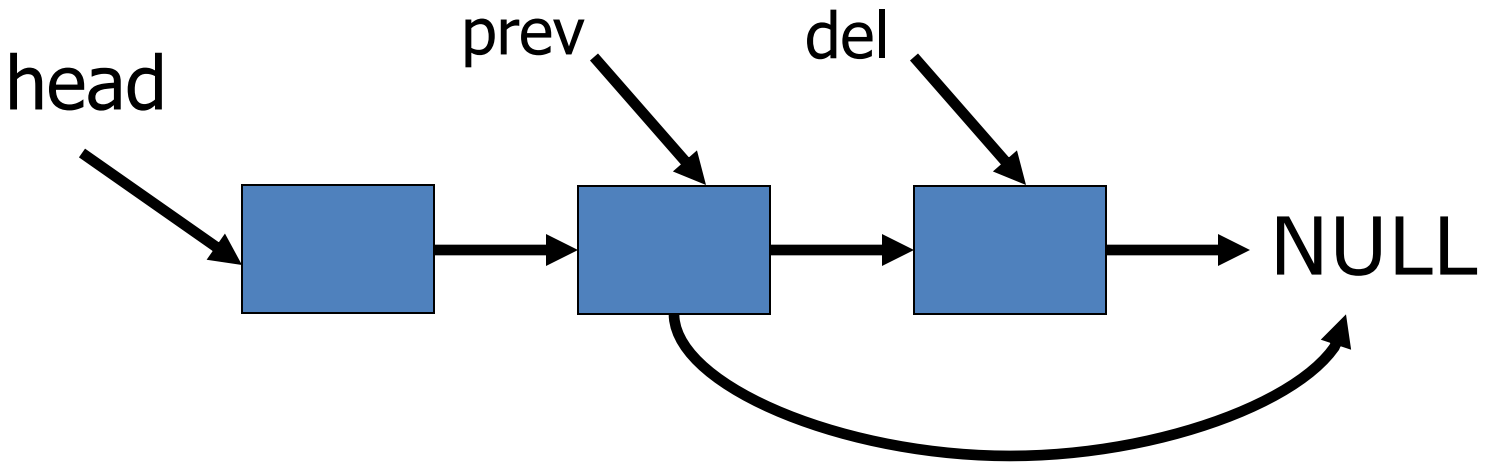


Delete the node in the middle/end of the list

Delete node `del` that is currently the middle/last node of the list:

```
<Determine the pointer prev pointed to the previous node of del>;
prev->next = del->next; //modify the link
free(del); //delete node del to free memory
```



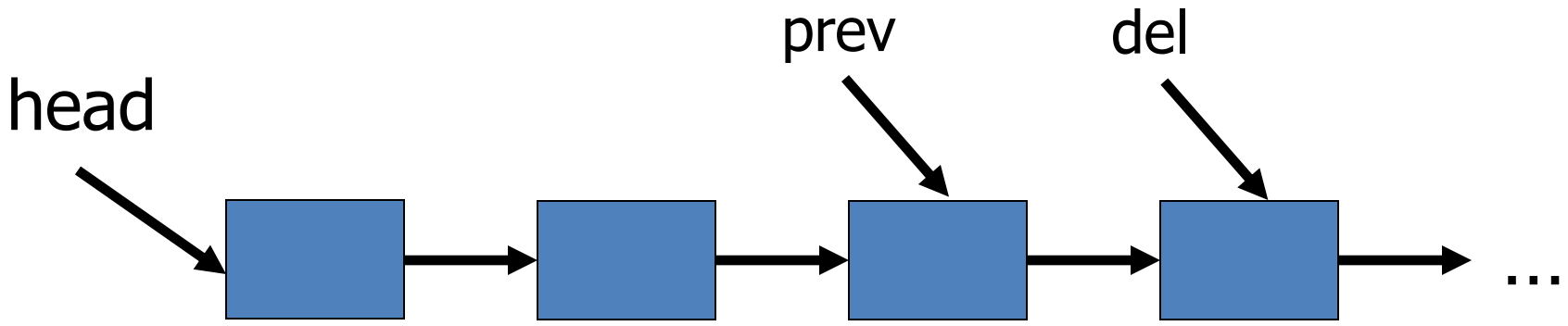


Delete the node at the middle/end of the list

Delete node `del` that is currently the middle/last node of the list:

```
<Determine the pointer prev pointed to the previous node of del>;
prev->next = del->next; //modify the link
free(del); //delete node del to free memory)
```

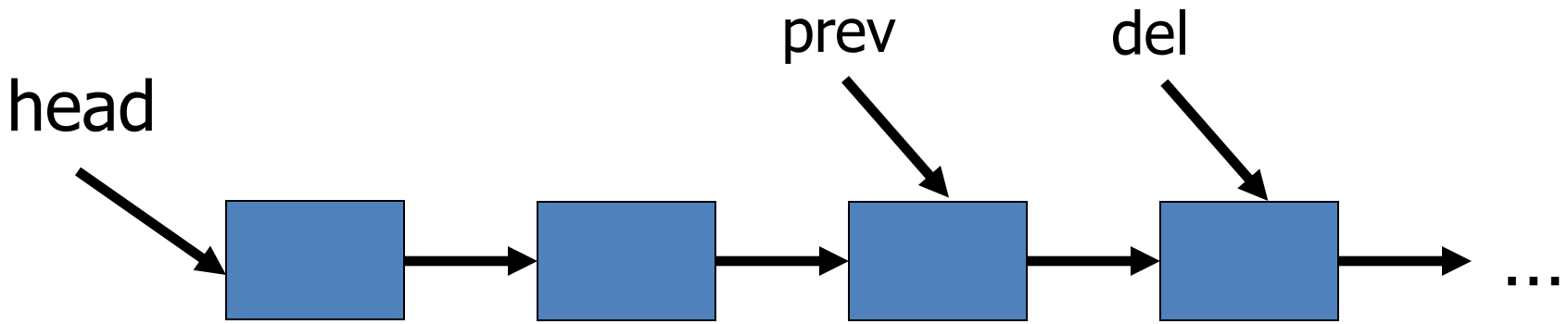
```
Node *prev =head;
while (prev->next != del) prev = prev->next;
```



Delete the node at the middle/end of the list

Delete node `del` that is currently the middle/last node of the list:

```
<Determine the pointer prev pointed to the previous node of del>;
prev->next = del->next; //modify the link
free(del); //delete node del to free memory)

Node *prev =head;
while (prev->next != del) prev = prev->next;
```



Delete a node pointed by the pointer del

Write the function **`node *Delete_Node(node *head, node *del)`**
to delete a node pointed by the pointer "del" of the list with the first node pointed by the pointer "head".
The function returns the address of the first node in the list after deletion:

- Delete the node `del` that is currently the first node of the list:

```
head = del->next;
free(del);
```

Delete node `del` that is currently the middle/last node of the list:

```
<Determine the pointer prev pointed to the previous node of del>;
prev->next = del->next; //modify the link
free(del); //delete node del to free memory)
```

```
Node *prev =head;
while (prev->next != del) prev = prev->next;
```

```
node *Delete_Node(node *head, node *del)
{
    if (head == del) //del is the first node of the list:
    {
        head = del->next;
        free(del);
    }
    else{
        node *prev = head;
        while (prev->next != NULL) prev = prev->next;
        prev->next =del->next;
        free(del);
    }
    return head;
}
```

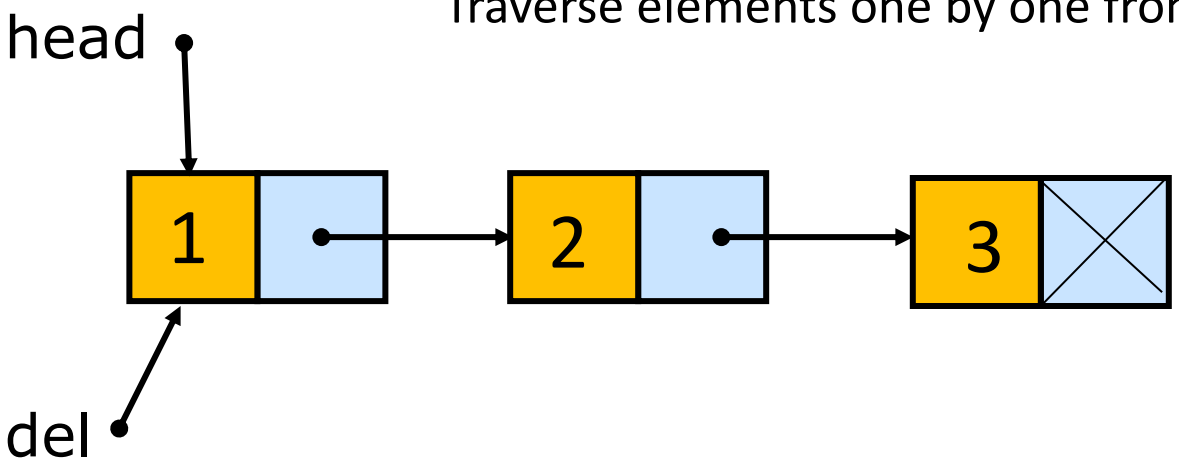
Operations on singly linked lists: Deletion

- Delete a node
- **Delete all nodes of the list**

Freeing all nodes of a list

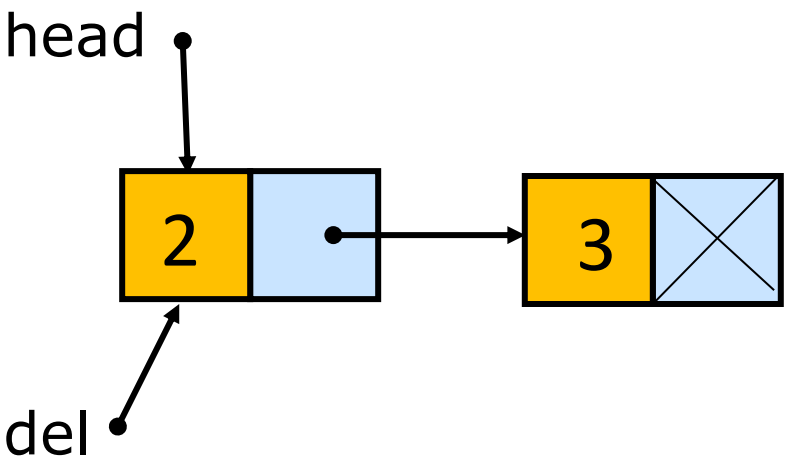
```
del = head ;
while (del != NULL)
{
    head = head->next;
    free(del);
    del = head;
}
```

Traverse elements one by one from the head till the end



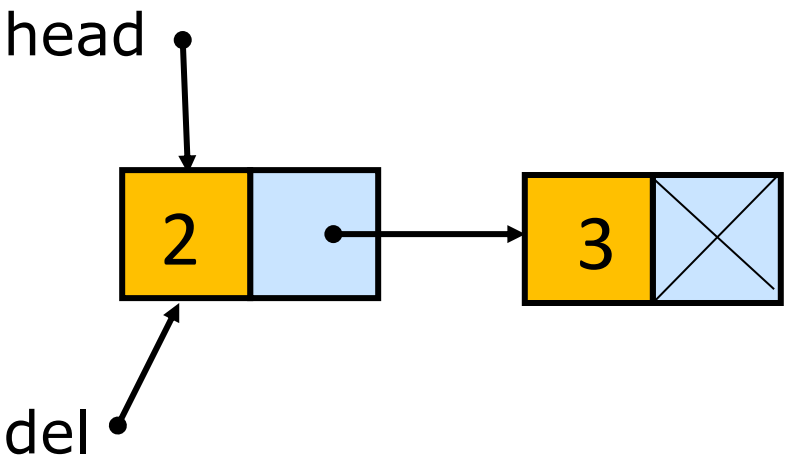
Freeing all nodes of a list

```
del = head ;
while (del != NULL)
{
    head = head->next;
    free(del);
    del = head;
}
```



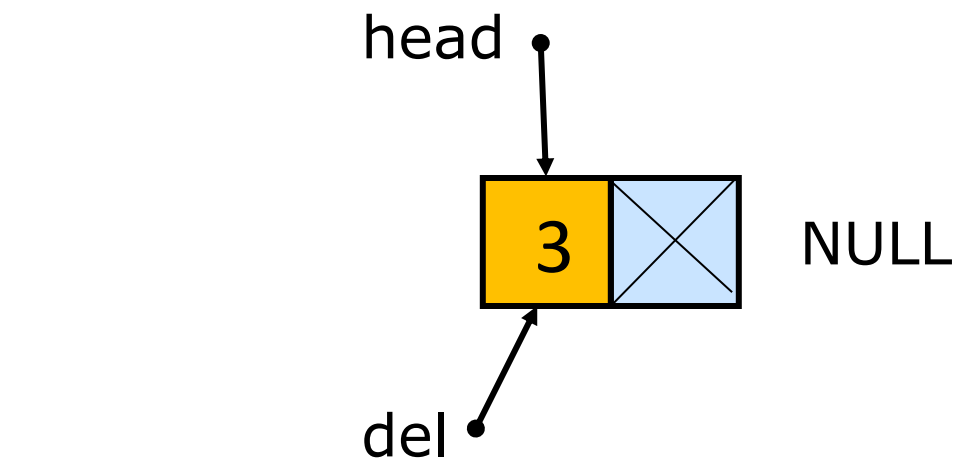
Freeing all nodes of a list

```
del = head ;
while (del != NULL)
{
    head = head->next;
    free(del);
    del = head;
}
```



Freeing all nodes of a list

```
del = head ;
while (del != NULL)
{
    head = head->next;
    free(del);
    del = head;
}
```



Freeing all nodes of a list

Write the function `node* deleteList(node* head)`
to delete all the node in the list having the first node pointed by the pointer `head`
The function returns the pointer `head` after deletion

```
node* deleteList(node* head)
{
   node *del = head ;
   while (del != NULL)
   {
      head = head->next;
      free(del);
      del = head;
   }
   return head;
}
```

Check whether the singly linked list is empty or not

Write the function `int IsEmpty(node *head)`

to check whether the singly linked list is empty or not (the pointer head pointed to the first node of the list).

The function returns 1 if the list is empty; 0 otherwise

```
int IsEmpty(node *head) {
     if (head == NULL)
          return 1;
     else return 0;
}
```

Operations on singly linked Lists

- Traverse the singly linked list
- Insert a node into the singly linked list
- Delete a node from the singly linked list
- **Search data in the singly linked list**

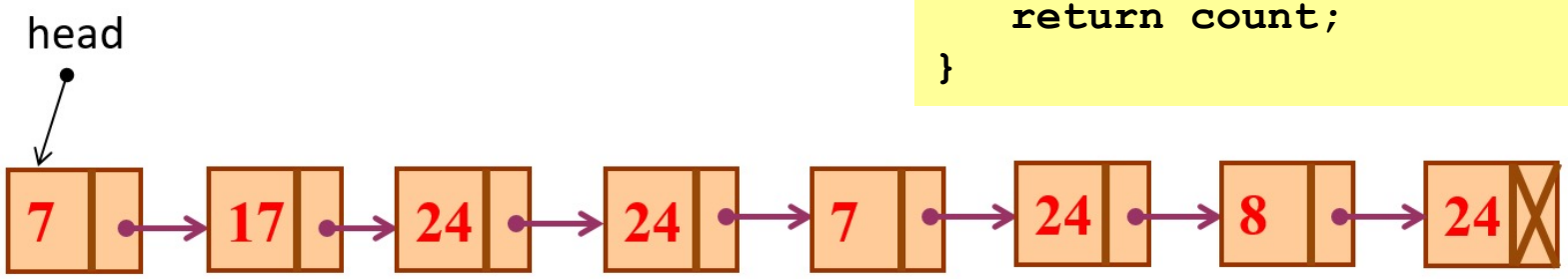
Searching

- To search for an element, we traverse from head until we locate the object or we reach the end of the list.

Example: Given a linked list consisting of integer numbers. Count the number of nodes with data field equal to number x.

```
typedef struct {
    int data;
    struct node* next;
}node;
node* head;
```

```
int countNodes(int x){
    int count = 0;
    node* e = head;
    while(e != NULL){
        if(e->data == x) count++;
        e = e->next;
    }
    return count;
}
```




```
int Result1 = countNodes(24);

int a =7;
int Result2 = countNodes(a);
```

Result1 = ?

Result2 = ?

Time Complexity: Singly-linked lists vs. 1D-arrays

Operation	**ID-Array Complexity**	**Singly-linked list Complexity**
Insert at beginning	O(n)	O(1)
Insert at end	O(1)	O(1) if the list has **tail** reference O(n) if the list has no **tail** reference
Insert at middle*	O(n)	O(n)
Delete at beginning	O(n)	O(1)
Delete at end	O(1)	O(n)
Delete at middle*	O(n): O(1) access followed by O(n) shift	O(n): O(n) search, followed by O(1) delete
Search	O(n) linear search O(log n) Binary search	O(n)
Indexing: What is the element at a given position k?	O(1)	O(n)

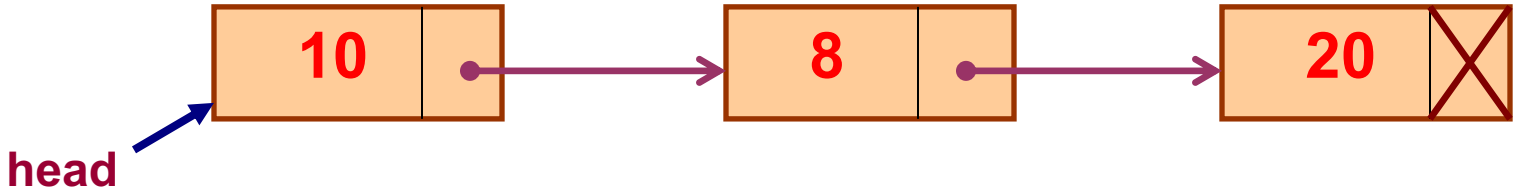
* middle: neither at the beginning nor at the end

Singly-linked lists vs. 1D-arrays

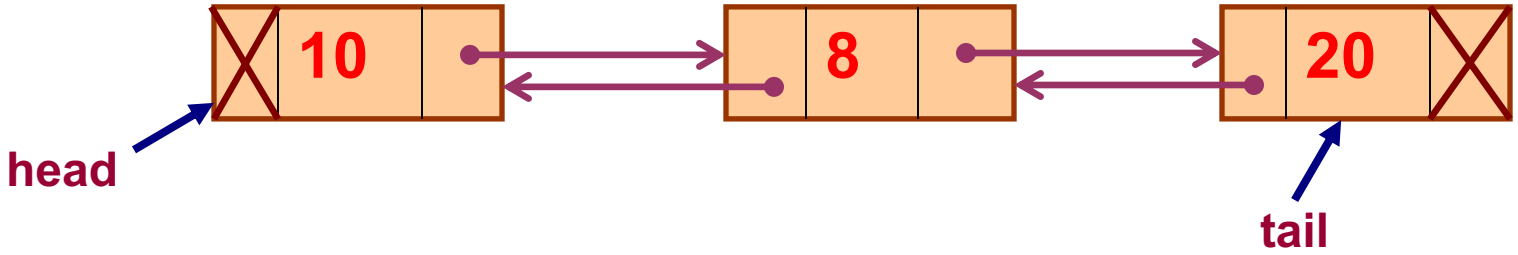
ID-array	Singly-linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Extra storage needed for references; however uses exactly as much memory as it needs
Sequential access is faster because of greater locality of references [Reason: Elements in contiguous memory locations]	Sequential access is slow because of low locality of references [Reason: Elements not in contiguous memory locations]

2.3. Linked list

- Singly linked list

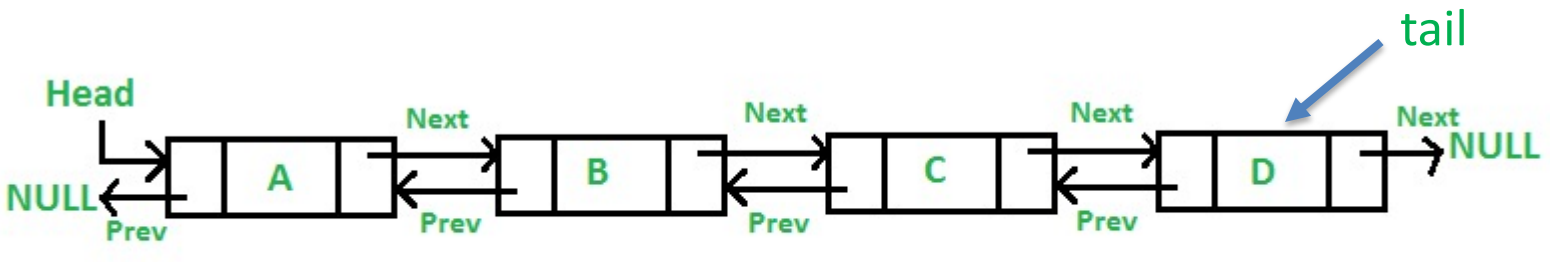


- **Doubly linked list**



Doubly linked list

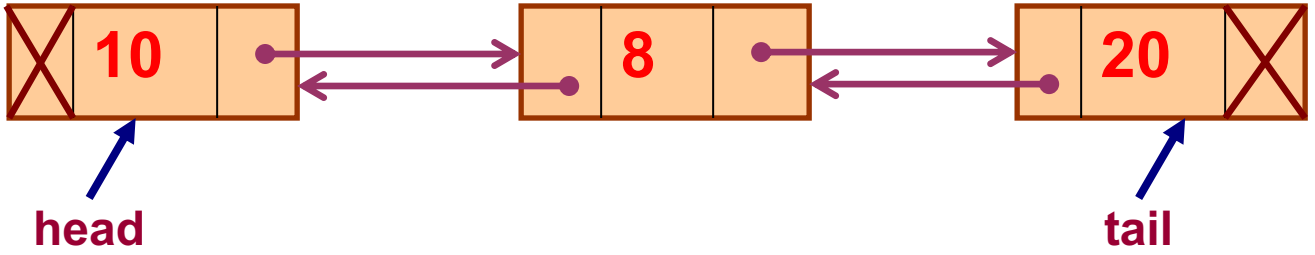
- A **D**oubly **L**inked **L**ist (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list



- 2 special nodes: `tail` and `head`
  - `head` has pointer `prev = null`
  - `tail` has pointer `next = null`
- Basic operations are considered similar as in the singly linked list

Doubly linked list

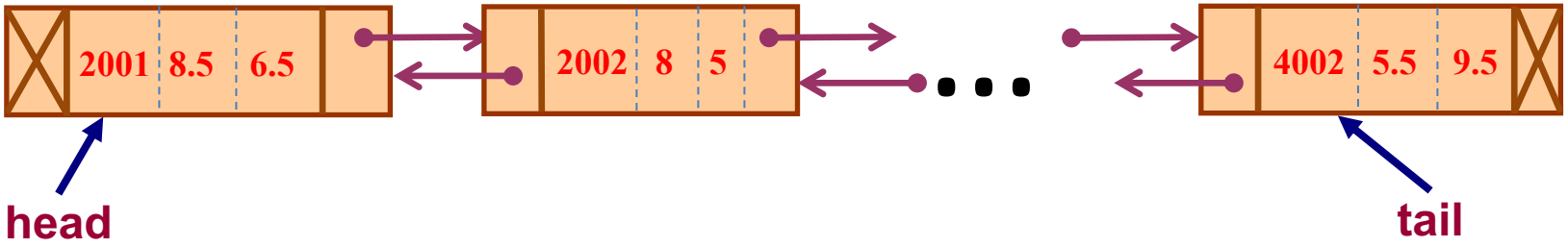
- Declare doubly linked list to store integer numbers:




```
typdedef struct {
  int number;
  struct dllist *next;
  struct dllist *prev;
} dllist;
dllist *head, *tail;
```

Doubly linked list

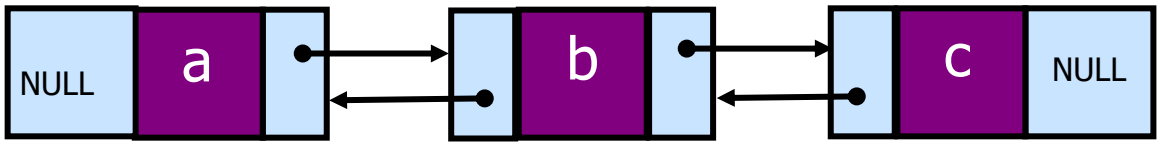
- Declare doubly linked list store data of students: ID, marks of math and physics




```
typedef struct{
    char id[15];
   float math, physics;
}student;

typedef struct {
   student data;
   struct ddlist* next;
   struct ddlist* prev;
}dllist;
dllist *head, *tail;
```

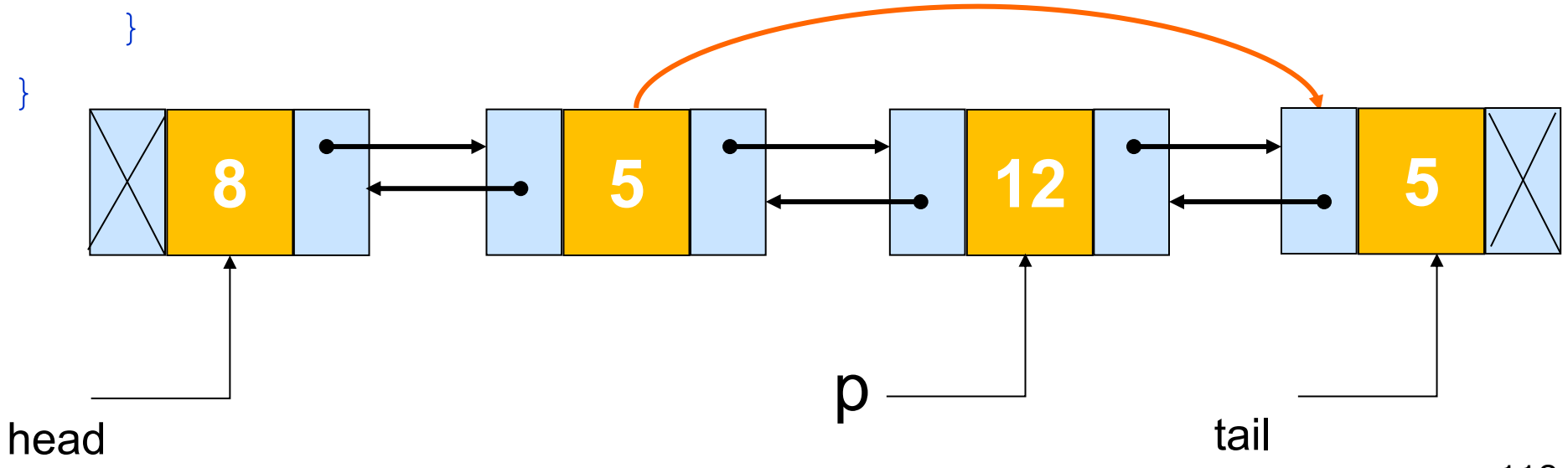
Doubly linked list – Example

```
typedef struct {
    char data;
    struct dblist *prev;
    struct dblist *next;
}dblist;
dblist node1, node2, node3;

node1.data='a';
node2.data='b';
node3.data='c';
node1.prev=NULL;
node1.next=node2;
node2.prev=node1;
node2.next=node3;
node3.prev=node2;
node3.next=NULL;
```



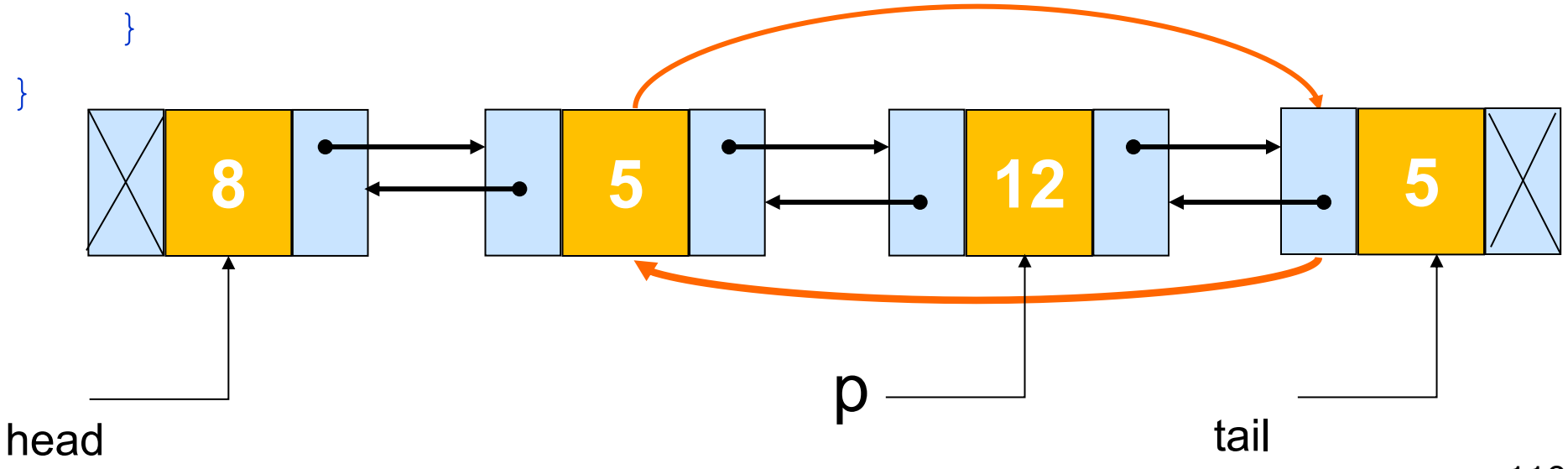
Delete a node pointed by the pointer p

```
void Delete_Node (ddlist *p){
    if (head == NULL) printf("Empty list");
    else {
      if (p==head) head = head->next; //Delete first element
      else p->prev->next = p->next;
      if (p->next!=NULL) p->next->prev = p->prev;
      else tail = p->prev;
      free(p);
    }
}
```



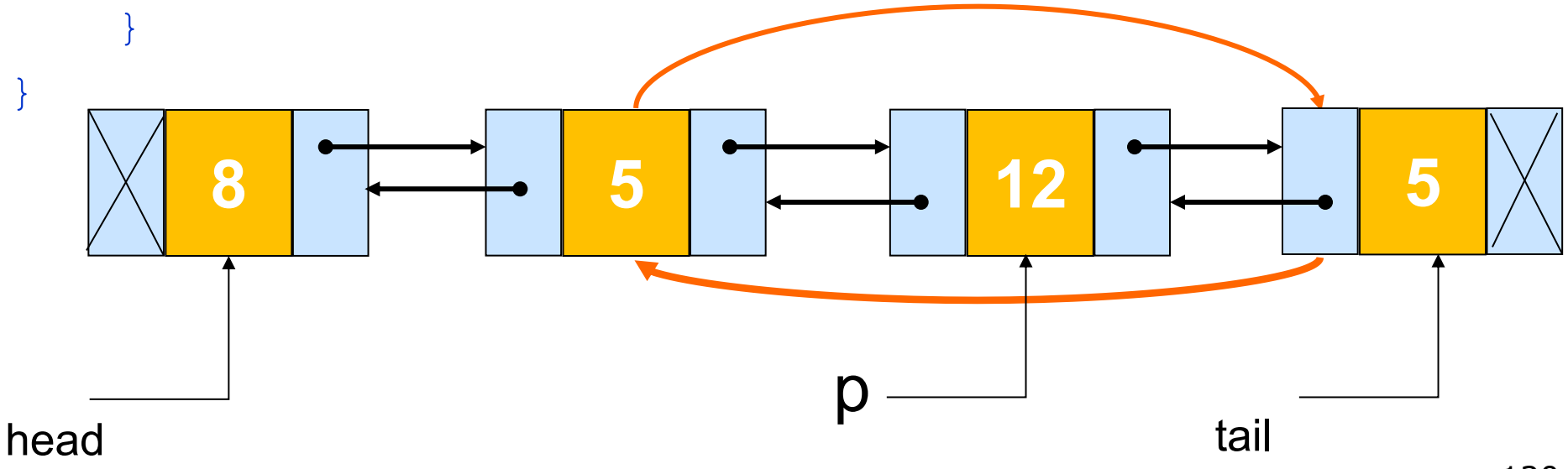
Delete a node pointed by the pointer p

```
void Delete_Node (ddlist *p){
    if (head == NULL) printf("Empty list");
    else {
      if (p==head) head = head->next; //Delete first element
      else p->prev->next = p->next;
      if (p->next!=NULL) p->next->prev = p->prev;
      else tail = p->prev;
      free(p);
    }
}
```



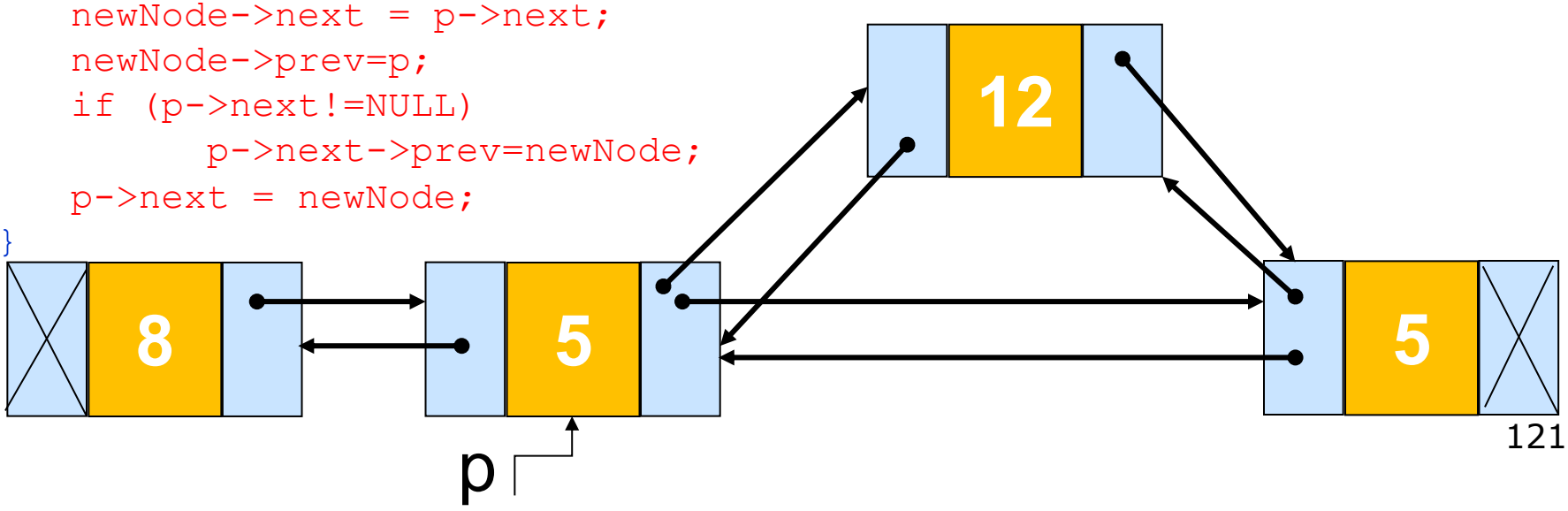
Delete a node pointed by the pointer p

```
void Delete_Node (ddlist *p){
    if (head == NULL) printf("Empty list");
    else {
      if (p==head) head = head->next; //Delete first element
      else p->prev->next = p->next;
      if (p->next!=NULL) p->next->prev = p->prev;
      else tail = p->prev;
      free(p);
    }
}
```



Insert a node after the node pointed by pointer p

```
void Insert_Node (NodeType X, ddlist *p){
   if (head == NULL){  // List is empty
        head =(ddlist*)malloc(sizeof(ddlist));
        head->data = X;
        head->prev =NULL;
        head->next =NULL;
   }
   else{
       ddlist *newNode;
       newNode=(ddlist*)malloc(sizeof(ddlist));
       newNode->data = X;
      newNode->next = NULL;

       newNode->next = p->next;
       newNode->prev=p;
       if (p->next!=NULL)
             p->next->prev=newNode;
       p->next = newNode;
   }
}
```



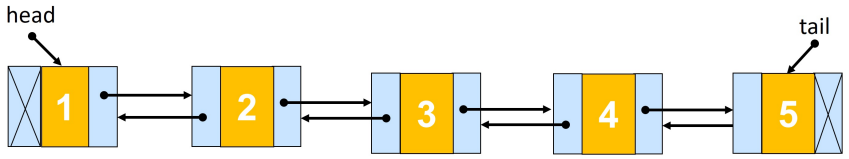
Exercise 2: Create a double linked list store integer numbers

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

typedef struct {
 int number;
 struct dllist *next;
 struct dllist *prev;
} dllist;
dllist *head, *tail;

/* Insert a new node p at the end of the list */
void append_node(dllist *p);
/* Insert a new node p after a node pointed by the pointer after */
void insert_node(dllist *p, dllist *after);
/* Delete a node pointed by the pointer p */
void delete_node(dllist *p);
```

```
/* Insert a new node p at the end of the list */
void append_node(dllist *p) {
   if(head == NULL)
   {
      head = p;
      p->prev = NULL;
   }
   else {
     tail->next = p;
     p->prev = tail;
   }
   tail = p;
   p->next = NULL;
}
```

```
/* Insert a new node p after a node pointed by the pointer after */
void insert_node(dllist *p, dllist *after) {
 p->next = after->next;
 p->prev = after;
 if(after->next != NULL)
     after->next->prev = p;
 else
     tail = p;
 after->next = p;
}
/* Delete a node pointed by the pointer p */
void delete_node(dllist *p) {
 if(p->prev == NULL)
     head = p->next;
 else  p->prev->next = p->next;
 if(p->next == NULL)
     tail = p->prev;
 else  p->next->prev = p->prev;
}
```

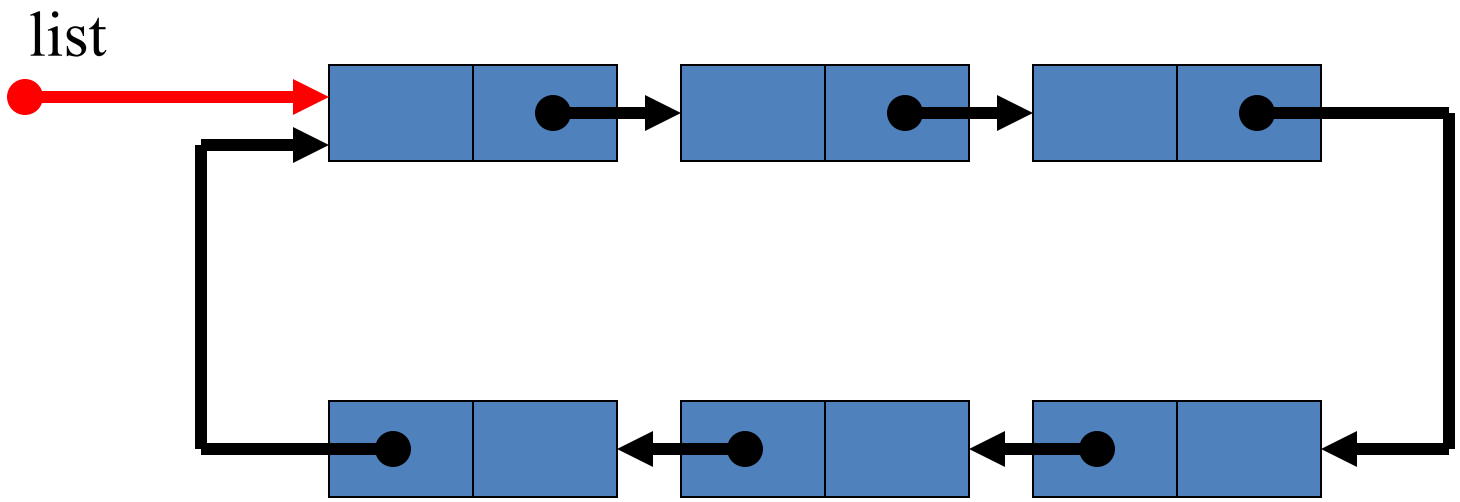
```
int main( ) {
dllist *tempnode; int i;
/* add some numbers to the double linked list */
for(i = 1; i <= 5; i++) {
    tempnode = (dllist *)malloc(sizeof(dllist));
    tempnode->number = i;
    append_node(tempnode);
}
/* print the dll list forward */
  printf(" Traverse the dll list forward \n");
  for(tempnode = head; tempnode != NULL; tempnode = tempnode->next)
       printf("%d\n", tempnode->number);

 /* print the dll list backward */
 printf(" Traverse the dll list backward \n");
 for(tempnode = tail; tempnode != NULL; tempnode = tempnode->prev)
       printf("%d\n", tempnode->number);
/* destroy the dll list */
 while(head != NULL) delete_node(head);
 return 0;
}
```



Several variants of linked lists

- Some common variants of linked list:
  - Circular Linked Lists
  - Circular Doubly Linked Lists
  - Linked Lists of Lists
- Basic operations on these variants are built similarly to the singly linked list and the doubly linked list that we consider above.

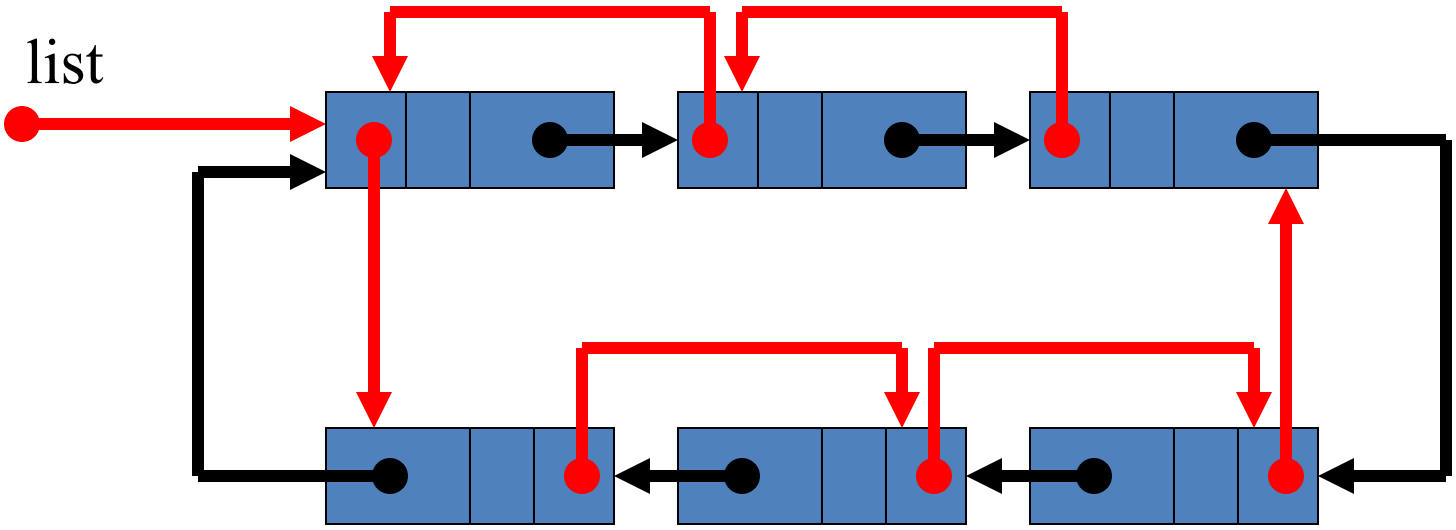
Circular linked list




```
typedef struct {
 NodeType data;
 struct node * next;
}next;
```

Store data

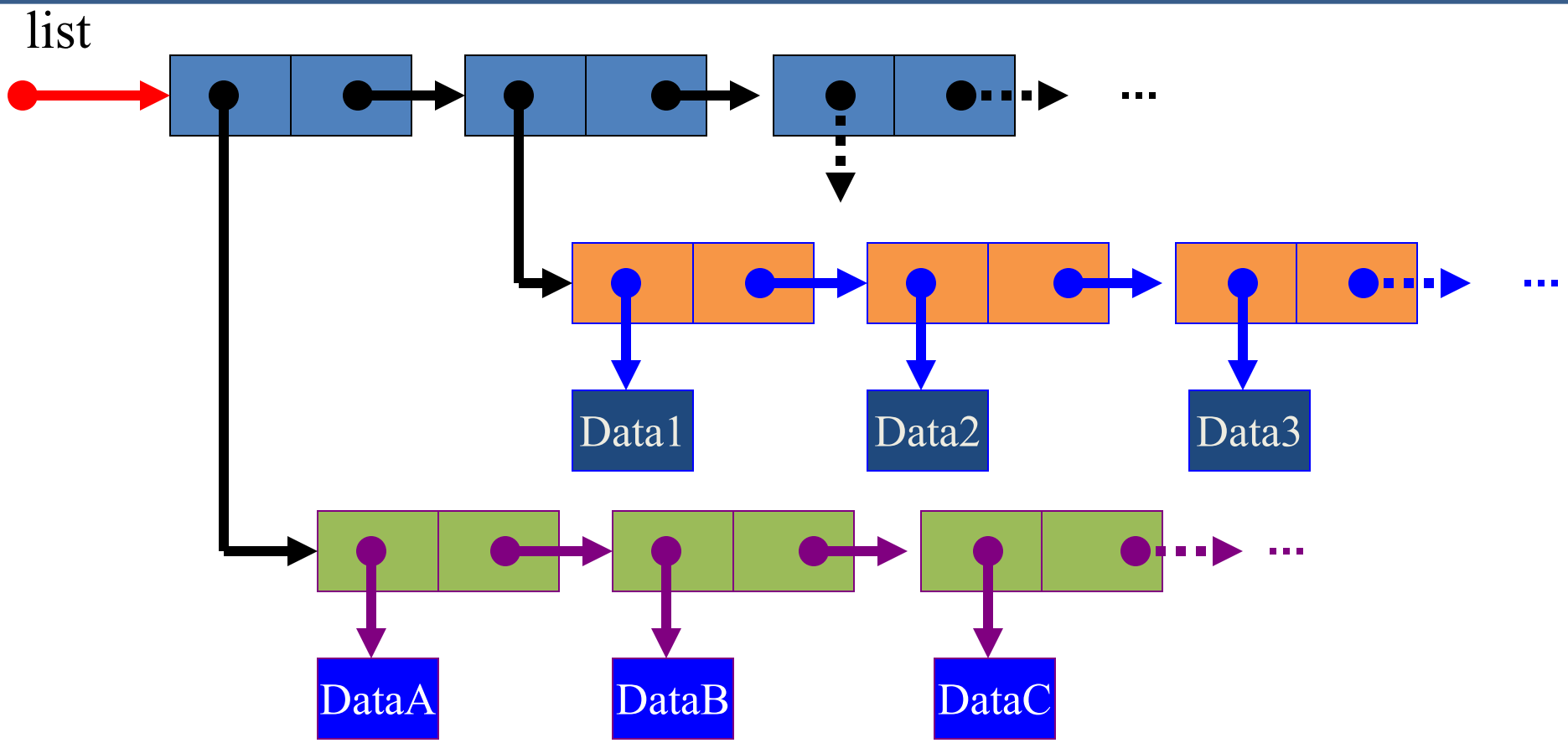
Circular Doubly Linked Lists




```
typedef struct {
  NodeType data;
  struct node * prev;
  struct node * next;
}node;
```

Store data

Linked Lists of Lists



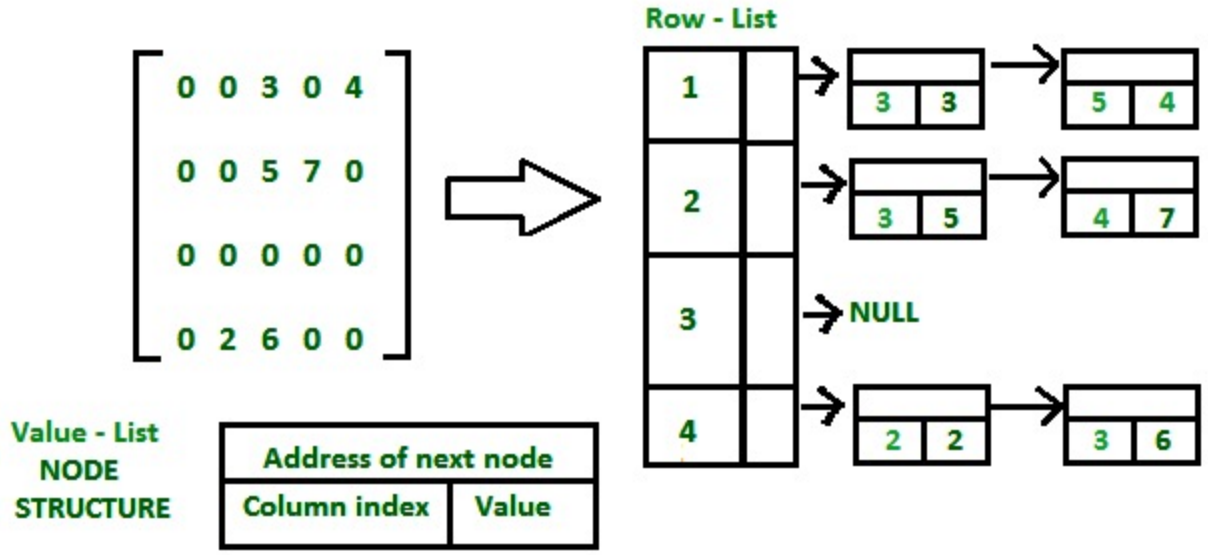
Linked list of lists Application – Sparse matrix

To represent sparse matrix, we could use linked list of lists which consists two lists:

- one list is used to represent the rows and each row contains the list of **triples: Column index, Value(non – zero element) and address field,** for non – zero elements.

For the best performance both lists should be stored in order of ascending keys.

```
// Node to represent triples
typdedef struct
{
    int column_index;
    int value;
    struct value_list *next;
}value_list;

typedef struct
{
    int row_number;
    struct row_list *link_down;
    struct value_list *link_right;
} row_list;
```



Exercise: Polynomial Addition

Polynomials: defined by a list of coefficients and exponents

- *degree* of polynomial = the largest exponent in the polynomial

$$p(x) = a_1 x^{e_1} + a_2 x^{e_2} + \cdots + a_n x^{e_n}$$

Example:

Polynomials $A(x) = 3x^{10}+2x^5+6x^4+4$

$B(x) = x^4+10x^3+3x^2+1$

To represent a polynomial, the easiest way is to use array $a[i]$ to store the coefficient of x_i. Operations with polynomials such as: Add two polynomials, Multiply two polynomials, ..., are thus possible to install simply.

Array a ~A(x)	a[10]	a[9]	a[8]	a[7]	a[6]	a[5]	a[4]	a[3]	a[2]	a[1]	a[0]
	3	0	0	0	0	2	6	0	0	0	4

Array b ~B(x)	b[4]	b[3]	b[2]	b[1]	b[0]
	1	10	3	0	1

$C(x) = A(x) + B(x) = 3x^{10}+2x^5+7x^4+3x^2+5$

Array c ~C(x)	c[10]	c[9]	c[8]	c[7]	c[6]	c[5]	c[4]	c[3]	c[2]	c[1]	c[0]
	=a[10] =3	=a[9] =0	=a[8] =0	=a[7] =0	=a[6] =0	=a[5] =2	=a[4]+b[4] =6+1=7	=a[3]+b[3] =0+10=10	=a[2]+b[2] =0+3=3	=a[1]+b[1] =0+0=0	=a[0]+b[0] =4+1=5

Exercise: Polynomial Addition

$A(x)=2x^{1000}+x^3$

$B(x)=x^4+10x^3+3x^2+1$

Use an array to keep track of the coefficients for all exponents:

…	2	…	0	1	0	0	0	A
…	0	…	1	10	3	0	1	B
	1000	…	4	3	2	1	0	

$A(x) + B(x) =$

advantage: easy implementation

disadvantage: waste space when sparse

In the case of sparse polynomial (with many coefficients equal to 0), we could represent polynomial by the linked list: We will build a list containing only the coefficients with the value not equal to zero together with the exponent. However, it is more complicated to implement the operations.

Contents

2.1. Array

2.2. Record

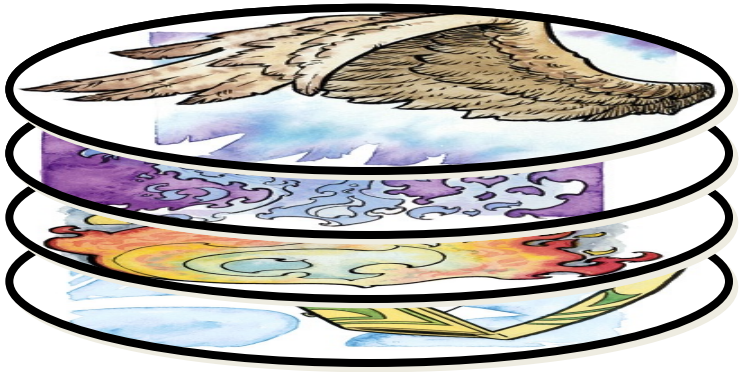
2.3. Linked List

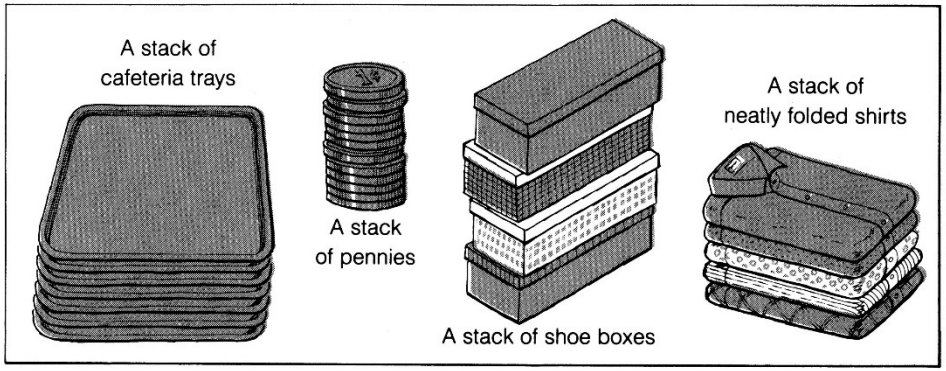
2.4. Stack

2.5. Queue

What is a stack?

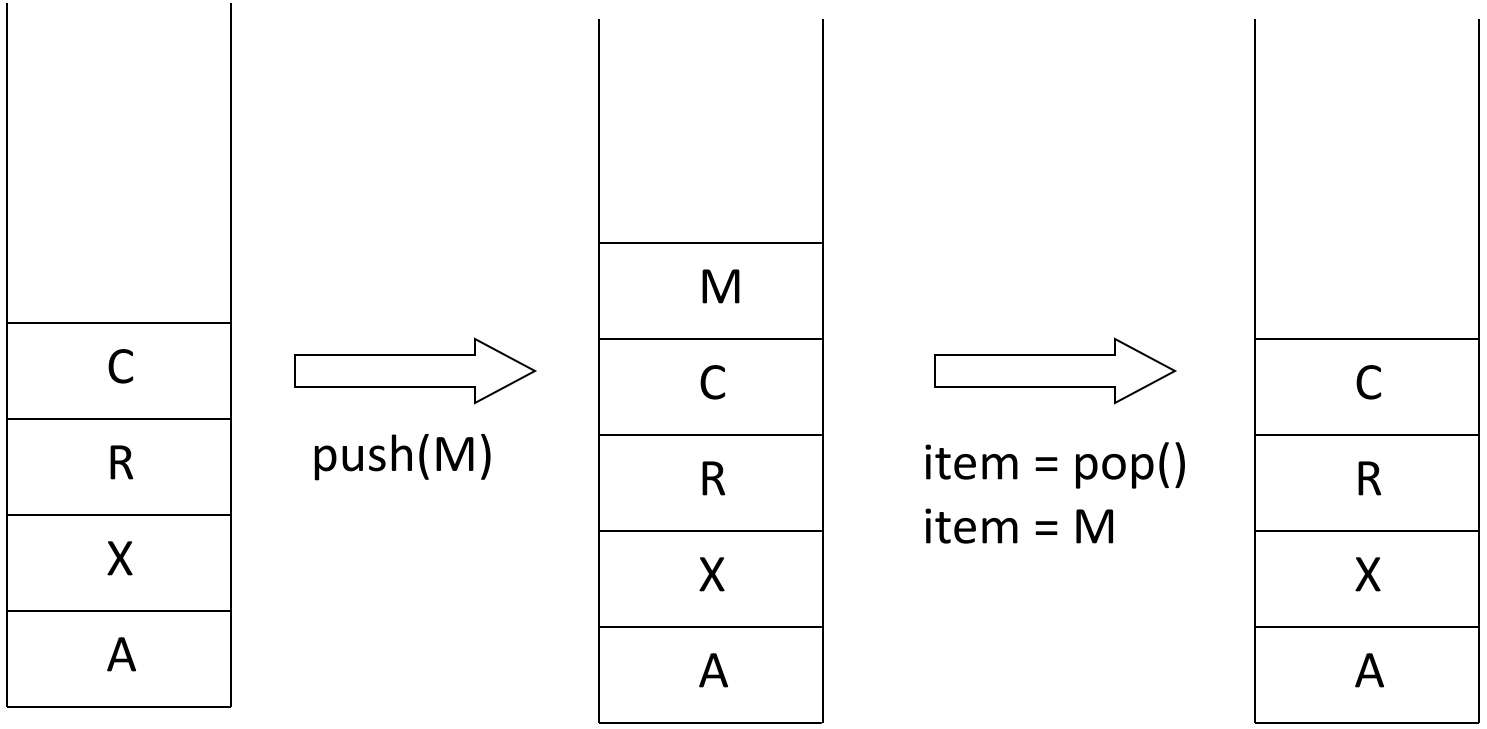
- A stack is a data structure that only allows items to be inserted and removed at ***one end***
  - We call this end the ***top*** of the stack
  - The other end is called the bottom
- Access to other items in the stack is not allowed
- The last element to be added is the first to be removed (**LIFO**: Last In, First Out)





Operation on stack

- *Push*: the operation to place a new item at the top of the stack
- *Pop:* the operation to remove the next item from the top of the stack

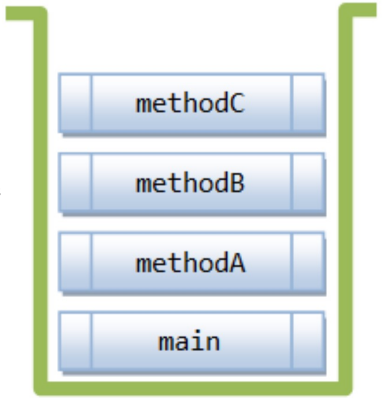


What Are Stacks Used For?

- Real life (Pile of books, Plate trays, etc.)
- More applications related to computer science
  - Program execution stack: Most programming languages use a "call stack" to implement function calling
    - When a method is called, its line number and other useful information are pushed (inserted) on the call stack
    - When a method ends, it is popped (removed) from the call stack and execution restarts at the indicated line number in the method that is now at the top of the stack

```
void methodC()
{
    printf("Enter methodC ");
}
void methodB()
{
    methodC();
}
void methodA()
{
    methodB();
}
void main() {
    methodA();
}
```

1. main pushed onto call stack, before invoking methodA
2. methodA pushed onto call stack, before invoking methodB
3. methodB pushed onto call stack, before invoking methodC
4. methodC pushed onto call stack, invoked then popped out from the call stack when completes
5. methodB popped out from call stack when completes.
6. methodA popped out from the call stack when completes.
7. main popped out from the call stack when completes. Program exits.



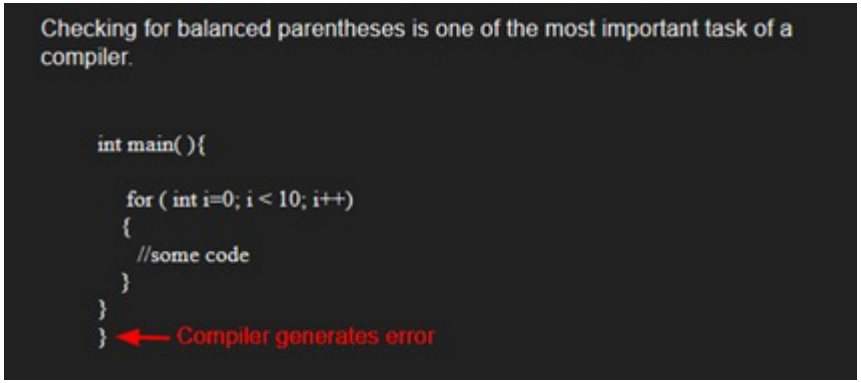
Method Call Stack
(Last-in-First-out Queue)

  - Evaluating expressions (e.g. (4/(2-2+3))*(3-4)*2)

What Are Stacks Used For?

- More applications related to computer science
  - compilers
    - parsing data between delimiters (brackets)
    - Check: each “(”, “{”, or “[” has to pair with “)”, “}”, or “]”

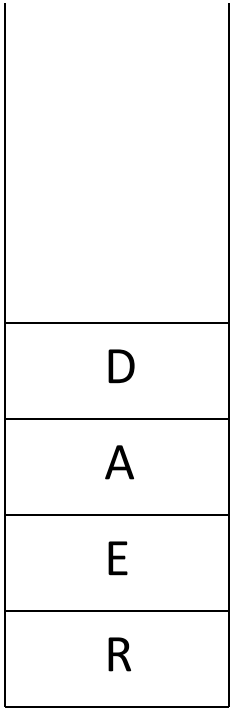
    Example:
      - correct: ( )(( )){([( )])}
      - correct: ((( )(( )){([( )])}
      - incorrect:)(( )){([( )])}
      - incorrect: ({[ ])}
      - incorrect: (



  - virtual machines
    - manipulating numbers
      - pop 2 numbers off stack, do work (such as add)
      - push result back on stack and repeat
  - artificial intelligence
    - finding a path

Example: Reversing a Word

- We can use a stack to reverse the letters in a word.
- How?
- Example: READ



Push(R)

Push(E)

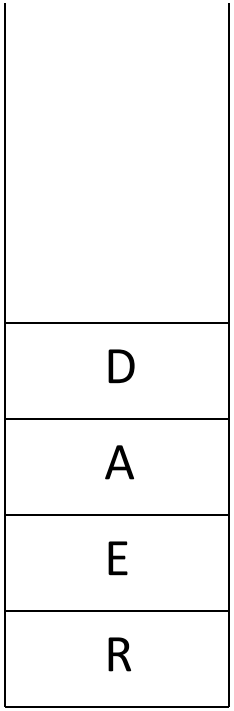
Push(A)

Push(D)

- Read each letter in the word and push it onto the stack

Example: Reversing a Word

- We can use a stack to reverse the letters in a word.
- How?
- Example: READ



Push(R) Pop(D)

Push(E) Pop(A)

Push(A) Pop(E)

Push(D) Pop(R)

DAER

- Read each letter in the word and push it onto the stack
- When you reach the end of the word, pop the letters off the stack and print them out

Implementing a Stack

- At least two different ways to implement a stack
  - array
  - linked list
- Which method to use depends on the application
  - what advantages and disadvantages does each implementation have?

Stack: Array Implementation

- If an array is used to implement a stack what is a good index for the top item?
  - Is it position 0?
  - Is it position numItems-1?
- Note that push and pop must both work in O(1) time as stacks are usually assumed to be extremely fast
- Implementing a stack using an array is fairly easy:
  - The bottom of the stack is at `S[0]`
  - The top of the stack is at `S[numItems-1]`
  - *push* onto the stack at `S[numItems]`
  - *pop* off of the stack at `S[numItems-1]`

Stack: Array Implementation

Basic operations:

- void STACKinit(int);
- int STACKempty();
- void STACKpush(Item);
- Item STACKpop();

```
typedef   .... Item;
static Item *s;
static int maxSize;//maximum number of elements that the stack could have
static int numItems; //current number of elements on stack
void STACKinit(int maxSize)
{
    s = (Item *) malloc(maxSize*sizeof(Item));
    N = 0;
}
int STACKempty(){return numItems==0;}
int STACKfull() {return numItems==maxSize;}
```

```
void STACKpush(Item item)
{
   if (Stackfull()) ERROR("Stack is full");
   else
   {   s[numItems] = item;
       numItems++;
   }
}
Item STACKpop()
{
   if (STACKempty()) ERROR("Stack is empty")
   else
   {
      numItems--;
      return s[numItems+1];
   }
}
```

Array Implementation Summary

- Advantages
  - Easy to implement
  - best performance: push and pop can be performed in O(1) time
- Disadvantage
  - fixed size: the size of the array must be initially specified because
    - The array size must be known when the array is created and is fixed, so that the right amount of memory can be reserved
    - Once the array is full no new items can be inserted
- If the maximum size of the stack is not known (or is much larger than the expected size) a dynamic array can be used
  - But occasionally push will take O(*n*) time

maxSize: maximum number of elements in the array

Stack overflow

- The condition resulting from trying to push an element onto a full stack.

```
if(STACKfull())
        STACKpush(item);
```

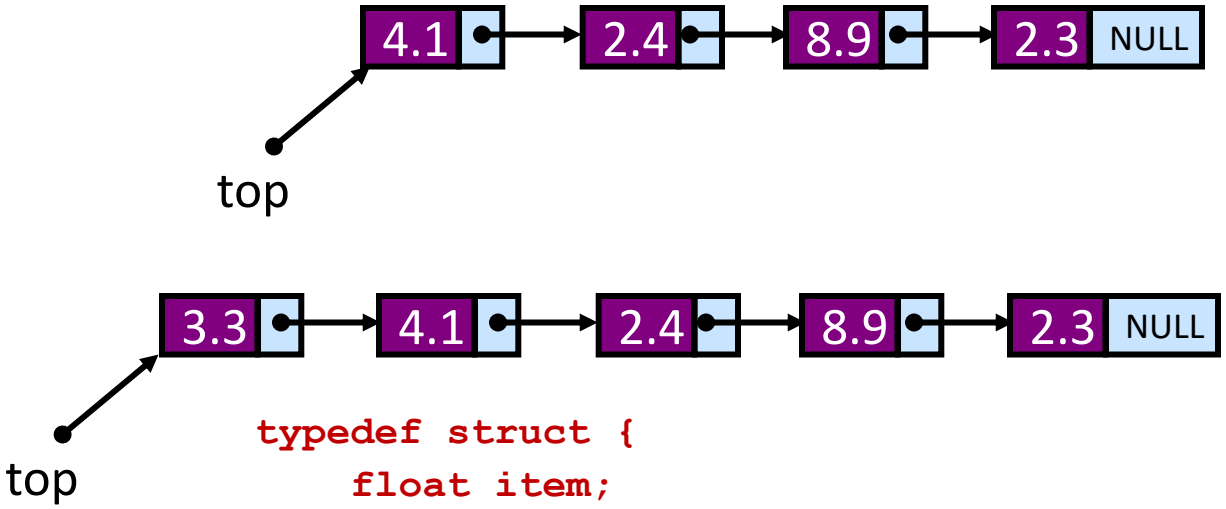
Stack underflow

- The condition resulting from trying to pop an empty stack.

```
if (STACKempty())
      STACKpop(item);
```

Implementing a Stack: using linked list

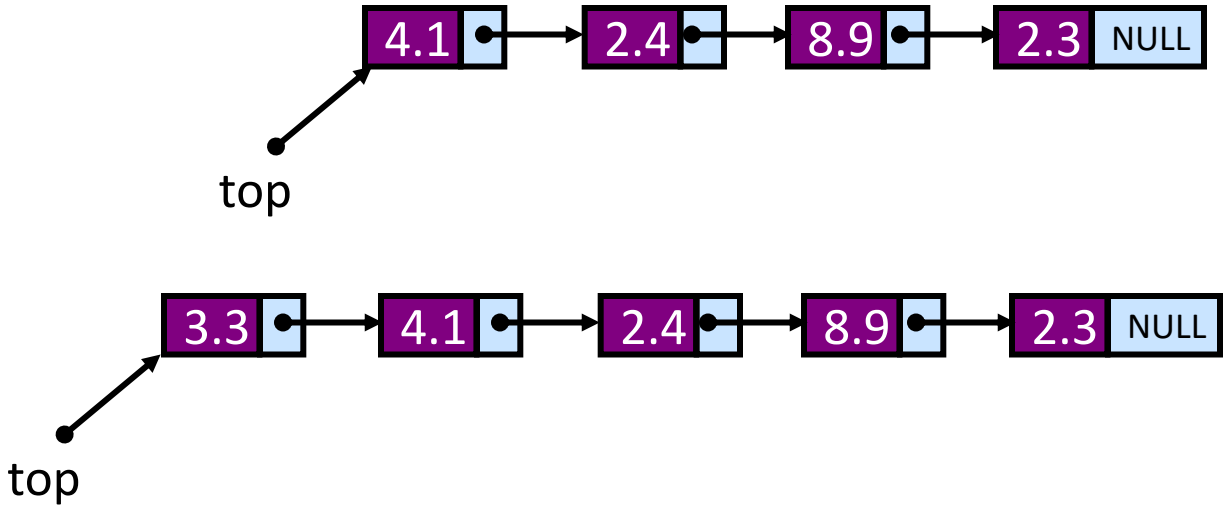
- Store the items in the stack in a linked list
- The top of the stack is the head node, the bottom of the stack is the end of the list
- *push* by adding to the front of the list
- *pop* by removing from the front of the list




```
typedef struct {
    float item;
    struct StackNode *next;
} StackNode;
typedef struct {
    StackNode *top;
}Stack;
```

Implementing a Stack: using linked list

- Store the items in the stack in a linked list
- The top of the stack is the head node, the bottom of the stack is the end of the list
- *push* by adding to the front of the list
- *pop* by removing from the front of the list



Operations

1. Init:

```
Stack *StackConstruct();
```

2. Check empty:

```
int StackEmpty(Stack* s);
```

3. Check full:

```
int StackFull(Stack* s);
```

4. Insert a new item into stack (Push): *insert a new item at the top of stack*

```
int StackPush(Stack* s, float* item);
```

5. Remove an item from stack (Pop): *remove and return the item at the top of stack:*

```
float pop(Stack* s);
```

6. Print out all items of stack

```
void Disp(Stack* s);
```

Initialize stack

```
Stack *StackConstruct() {
    Stack *s;
    s = (Stack *)malloc(sizeof(Stack));
    if (s == NULL) {
      return NULL;  // No memory
    }
    s->top = NULL;
    return s;
}

/****   Destroy stack   *****/
void StackDestroy(Stack *s) {
    while (!StackEmpty(s)) {
      StackPop(s);
    }
    free(s);
}
```

```
/*** Check empty   ***/
int StackEmpty(const Stack *s) {
    return (s->top == NULL);
}

/*** Check full  ***/
int StackFull() {
    printf("\n NO MEMORY! STACK IS FULL");
    return 1;
}
```

Display all items in the stack

```
void disp(Stack* s) {
   StackNode* node;
   int ct = 0; float m;
   printf("\n\n  List of all items in the stack \n\n");
   if (StackEmpty(s))
       printf("\n\n  >>>>>  EMPTY STACK   <<<<<\n");
   else {
      node = s->top;
      do {
         m = node->item;
         printf("%8.3f \n", m);
         node = node->next;
       } while (!(node == NULL));
   }
}
```

Push

Need to do the following steps:

(1) Create new node: allocate memory and assign data for new node

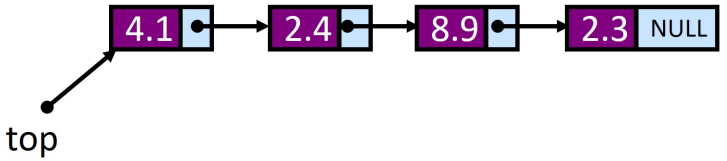
(2) Link this new node to the top (head) node

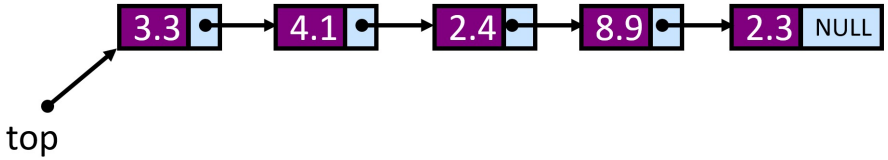
(3) Assign this new node as top (head) node

```
int StackPush(Stack *s, float item) {
  StackNode *node;
  node = (StackNode *)malloc(sizeof(StackNode)); //(1)
  if (node == NULL) {
    StackFull(); return 1;  // overflow: out of memory
  }
  node->item = item;          //(1)
  node->next = s->top;        //(2)
  s->top = node;              //(3)
  return 0;
}
```

Pop

1. Check whether the stack is empty
2. Memorize address of the current top (head) node
3. Memorize data of the current top (head) node
4. Update the top (head) node: the top (head) node now points to its next node
5. Free the old top (head) node
6. Return data of the old top (head) node






```
float StackPop(Stack *s) {
    float data;
    StackNode *node;
    if (StackEmpty(s))          //(1)
      return NULL;          // Empty Stack, can't pop
    node = s->top;              //(2)
    data = node->item;          //(3)
    s->top = node->next;        //(4)
    free(node);                 //(5)
    return item;                //(6)
}
```

Experimental program

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
// all above functions of stacks are put here
int main() {
   int ch,n,i;   float m;
   Stack* stackPtr;
   while(1)
   {  printf("\n\n======================\n");
      printf(" STACK TEST PROGRAM \n");
      printf("======================\n");
      printf(" 1.Create\n 2.Push\n 3.Pop\n 4.Display\n 5.Exit\n");
      printf("----------------------\n");
      printf("Input number to select the appropriate operation: ");
      scanf("%d",&ch); printf("\n\n");
```

```
  switch(ch) {
     case 1:     printf("INIT STACK");
                 stackPtr = StackConstruct(); break;
     case 2:     printf("Input float number to insert into stack: ");
                 scanf("%f",&m);
                 StackPush(stackPtr, m); break;
     case 3:     m=StackPop(stackPtr);
                 if (m != NULL)
                   printf("\n Data Value of the popped node: %8.3f\n",m);
                 else {
                   printf("\n  >>> Empty Stack, can't pop <<<\n");}
                 break;
     case 4:     disp(stackPtr); break;
     case 5:     printf("\n  Bye! Bye! \n\n");
                 exit(0);    break;
     default:    printf("Wrong choice");
    } //switch
  } // end while
} //end main
```

Implementing a Stack: using linked list

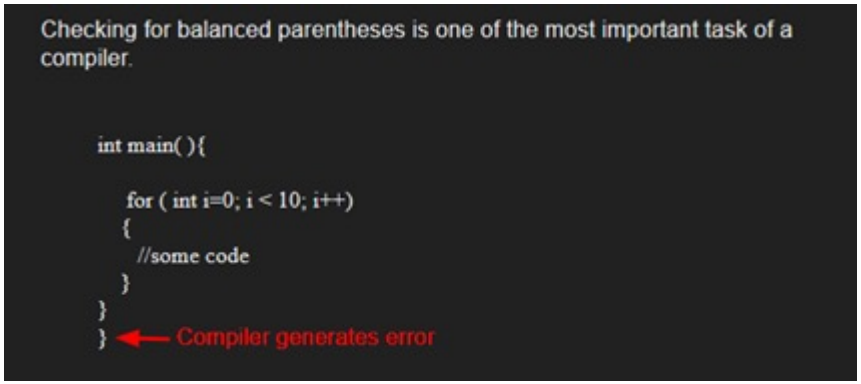
- Advantages:
  - always constant time to push or pop an element
  - can grow to an infinite size
- Disadvantages
  - Difficult to implement

Application 1: Parentheses Matching

Check for balanced parentheses in an expression:

Given an expression, write a program to examine whether the pairs match and the order is correct of “{“,”}”,”(“,”)”,”[“,”]”.

For example, the program should print true for expression = “[()]{}{[()()]()}” and false for expression = “[(])”



Algorithm:

1) Declare a character stack S.

2) Now traverse the string expression

a) If the current character is a starting bracket (‘(‘ or ‘{‘ or ‘[‘) then push it to stack.

b) If the current character is a closing bracket (‘)’ or ‘}’ or ‘]’) then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.

3) After complete traversal, if there is some starting bracket left in stack then “not balanced”

Application 2: HTML Tag Matching

- In HTML, each <name> has to pair with </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

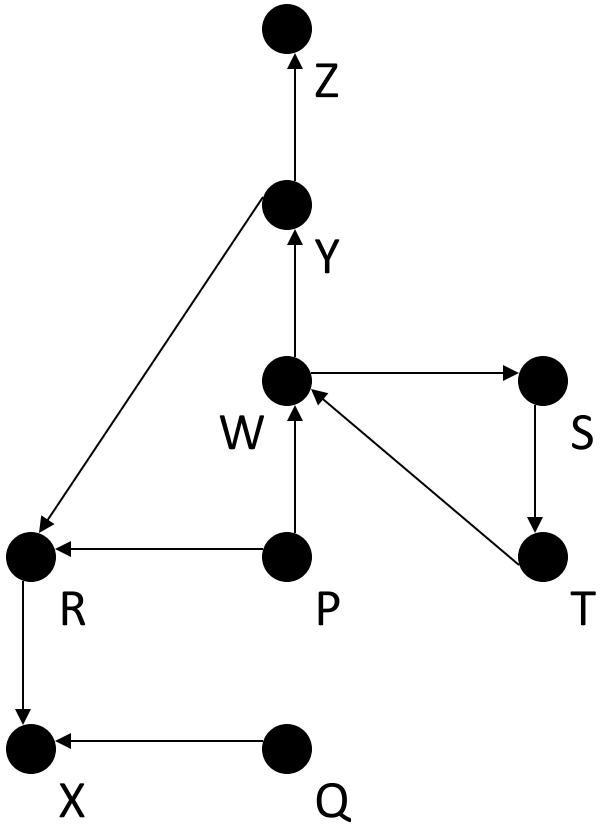
The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Application 3: Finding a Path using stack

- Consider the following graph of flights

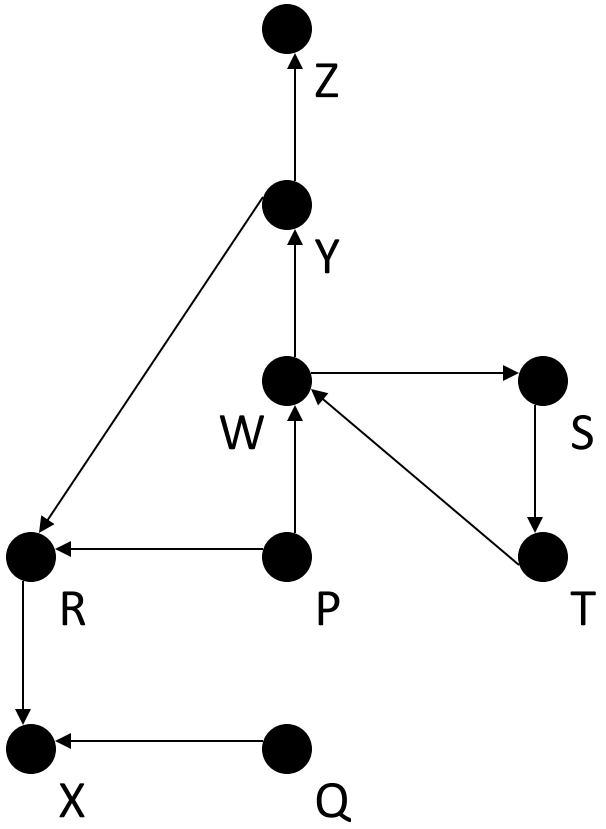


Application 3: Finding a Path using stack

- If it exists, we can find a path from any city C_1 to another city C_2 using a stack
  - place the starting city on the bottom of the stack
    - mark it as visited
    - pick any arbitrary arrow out of the city
      - city cannot be marked as visited
    - place that city on the stack
      - also mark it as visited
    - if that's the destination, we're done
    - otherwise, pick an arrow out of the city currently at
      - next city must not have been visited before
      - if there are no legitimate arrows out, pop it off the stack and go back to the previous city
    - repeat this process until the destination is found or all the cities have been visited

Application 3: Finding a Path using stack

- Consider the following graph of flights



- Want to go from P to Y
  - push P on the stack and mark it as visited
  - pick R as the next city to visit (random select)
    - push it on the stack and mark it as visited
  - pick X as the next city to visit (only choice)
    - push it on the stack and mark it as visited
  - no available arrows out of X – pop it
  - no more available arrows from R – pop it
  - pick W as next city to visit (only choice left)
    - push it on the stack and mark it as visited
  - pick Y as next city to visit (random select)
    - this is the destination – all done

Contents

2.1. Array

2.2. Record

2.3. Linked List

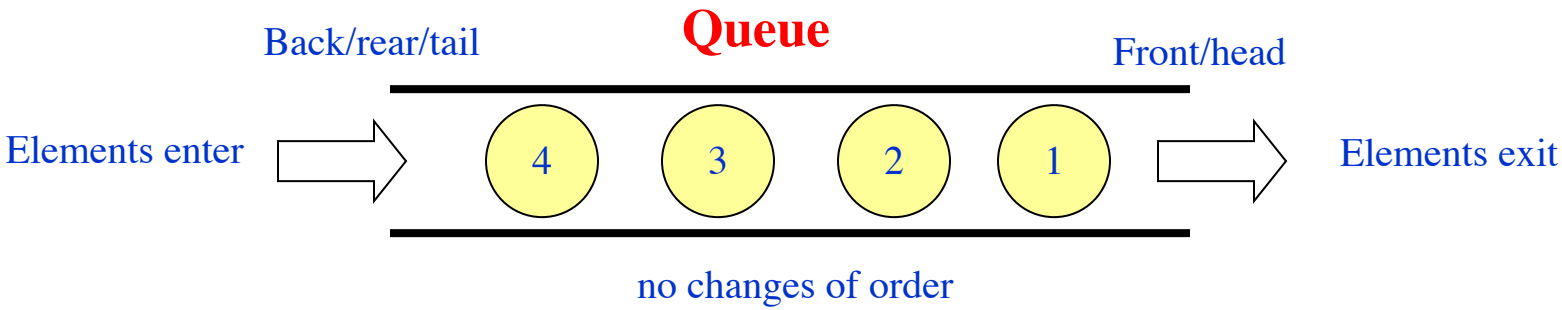
2.4. Stack

2.5. Queue

What is a Queue?

Queues

- What is a queue?
  - A data structure of ordered items such that items can be inserted only at one end and removed at the other end.



Example: A line at the supermarket

- What can we do with a queue?
  - Enqueue - Add an item to the queue
  - Dequeue - Remove an item from the queue

These operations are also called insert and getFront in order to simplify things.

- A queue is called a FIFO (First in-First out) data structure.

Queue specification

Definitions: (provided by the user)

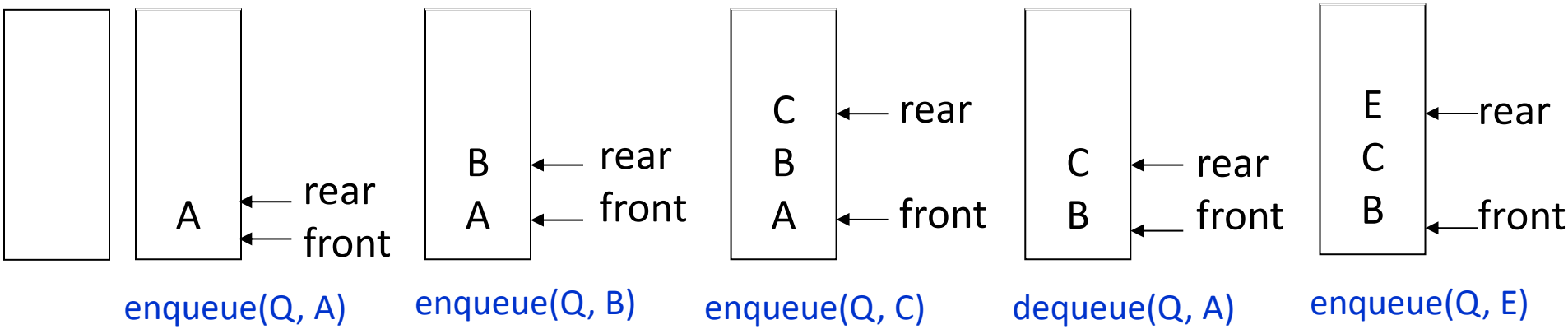
- *maxSize:* Max number of items that might be on the queue
- *ItemType*: Data type of the items on the queue

Operations:

- `Q = init();` initialize empty queue Q
- `isEmpty(Q);` returns "true“ if queue Q is empty
- `isFull(Q);` returns "true“ if Q is full, indicates that we already use the maximum memory for queue; otherwise returns “false”
- `frontQ(Q);` returns the item that is in front (head) of queue Q or returns error if queue Q is empty.
- `enqueue(Q,x);` inserts item x into the back (rear) of queue Q. If before making insertion, the queue Q is full, then give the notification about that.
- `x = dequeue(Q);` deletes the element at the front (head) of the queue Q, then returns x which is the data of this element. If the queue Q is empty before dequeue, then give the error notification.
- `print(Q);` gives the list of all elements in the queue Q in the order from the front to the back.
- `sizeQ(Q);` returns the number of elements currently in the queue Q.

FIFO


Add/
Enqueue
(Remove/Dequeue)
Back/Rear
Front/Head


A
rear
front
enqueue(Q, A)
B
A
rear
front
enqueue(Q, B)
C
B
A
rear
front
enqueue(Q, C)
C
B
rear
front
dequeue(Q, A)
E
C
B
rear
front
enqueue(Q, E)

Queues everywhere!!!!

Queues

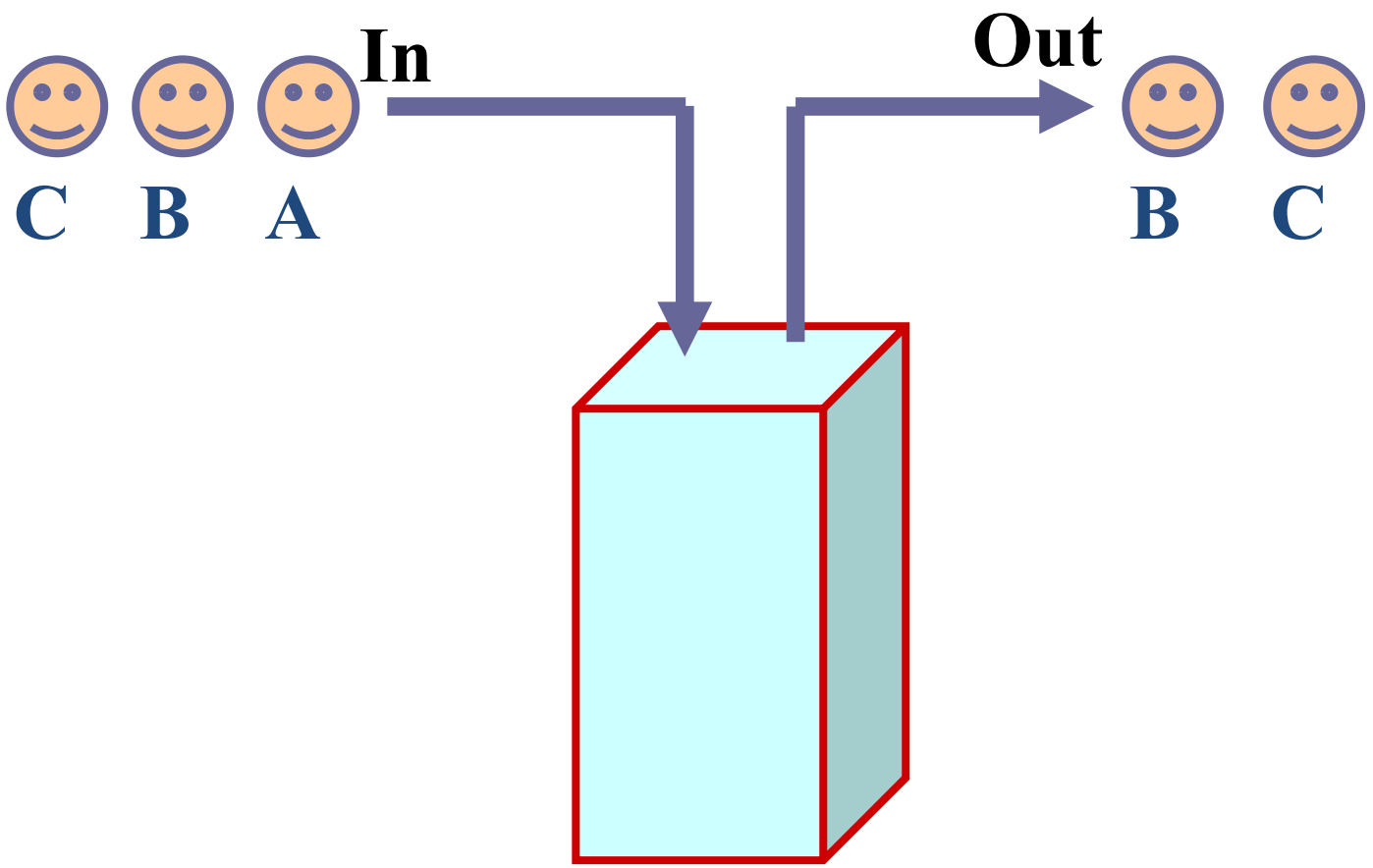
- What are some applications of queues?
  - Round-robin scheduling in processors
  - Input/Output processing
  - Queueing of packets for delivery in networks

Given the sequence of operations on queue Q as following. Determine the output and the data on the queue Q after each operation:

	Operation	Output	Queue Q
1	`enqueue(5)`	-	(5)
2	`enqueue(Q,3)`	-	(5, 3)
3	`dequeue(Q)`	5	(3)
4	`enqueue(Q,7)`	-	(3, 7)
5	`dequeue(Q)`	3	(7)
6	`front(Q)`	7	(7)
7	`dequeue(Q)`	7	( )
8	`dequeue(Q)`	error	( )
9	`isEmpty(Q)`	true	( )
10	`size(Q)`	0	( )
11	`enqueue(Q,9)`	-	(9)
12	`enqueue(Q,7)`	-	(9, 7)
13	`enqueue(Q,3)`	-	(9, 7, 3)
14	`enqueue(Q,5)`	-	(9, 7, 3, 5)
15	`dequeue(Q)`	9	(7, 3, 5)

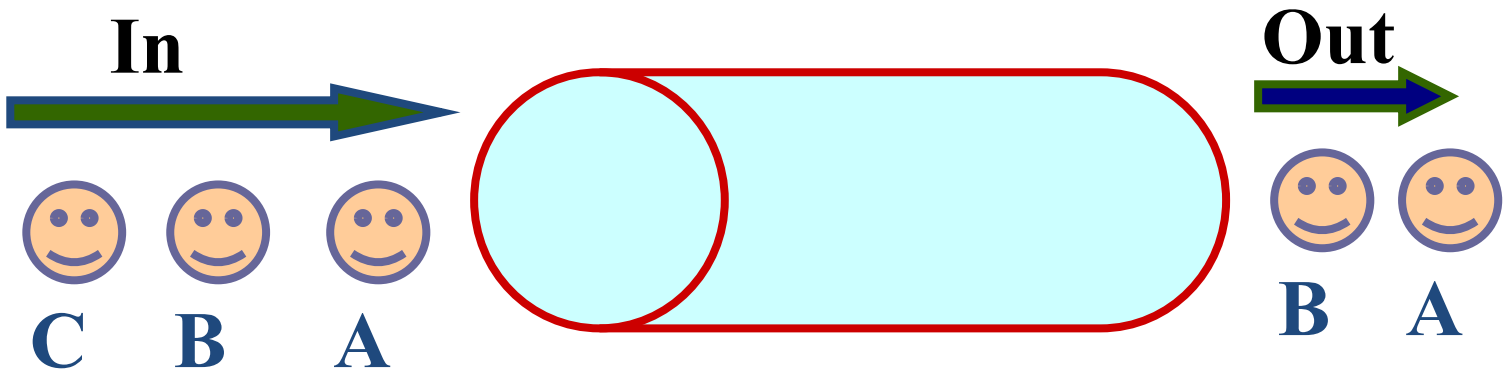
Stack

Data structure with Last-In First-Out (LIFO) behavior



Queue

Data structure with First-In First-Out (FIFO) behavior

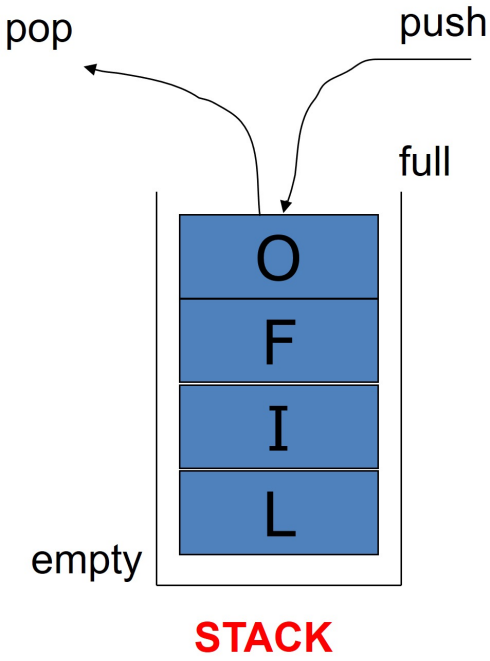


Implementing a Queue

- Just like a stack, we can implementing a queue in two ways:
  - Using an array
  - Using a linked list

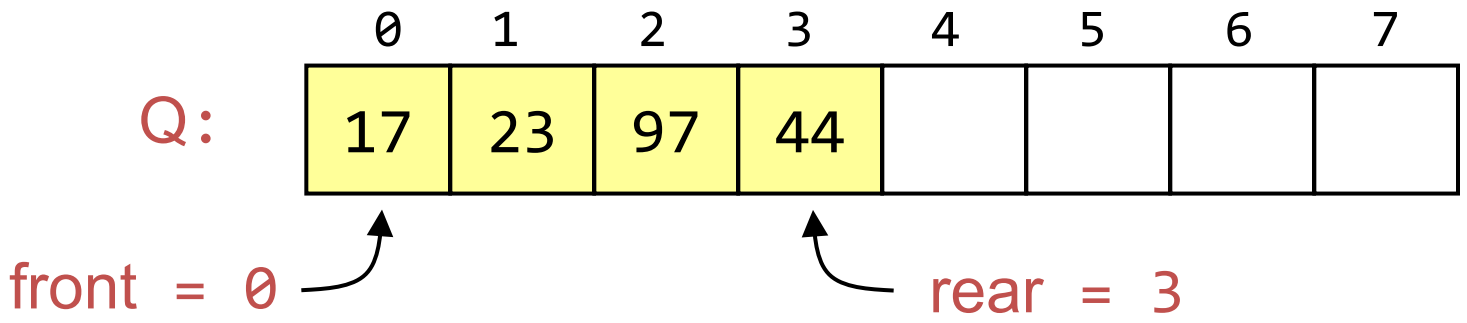
Implementing a Queue: using Array

- Using an array to implement a queue is significantly harder than using an array to implement a stack. Why?
  - A stack: we add and remove at the same end,
  - A queue: we add to one end and remove from the other.



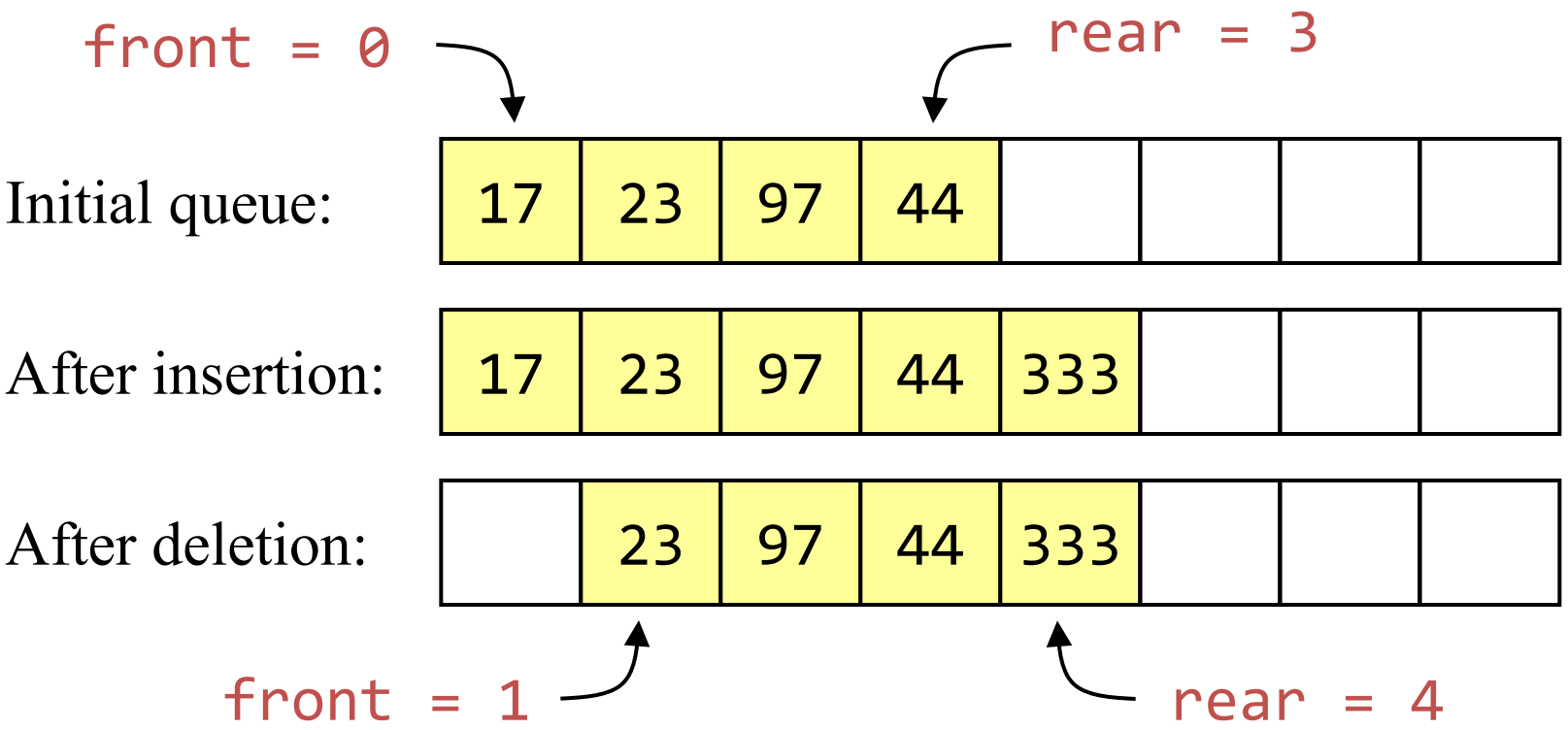
Array implementation of queues

- An array “Q” of size n
- Two pointers each representing one end of the queue
  - front: the end side where items are removed from the queue
  - rear: the end side from which items are added to the queue



- Enqueue(Q,x): put item x in the queue
  - rear++; Q[rear] = x
- Dequeue(Q): remove item x from the queue
  - Dequeue Q[front]; front++;

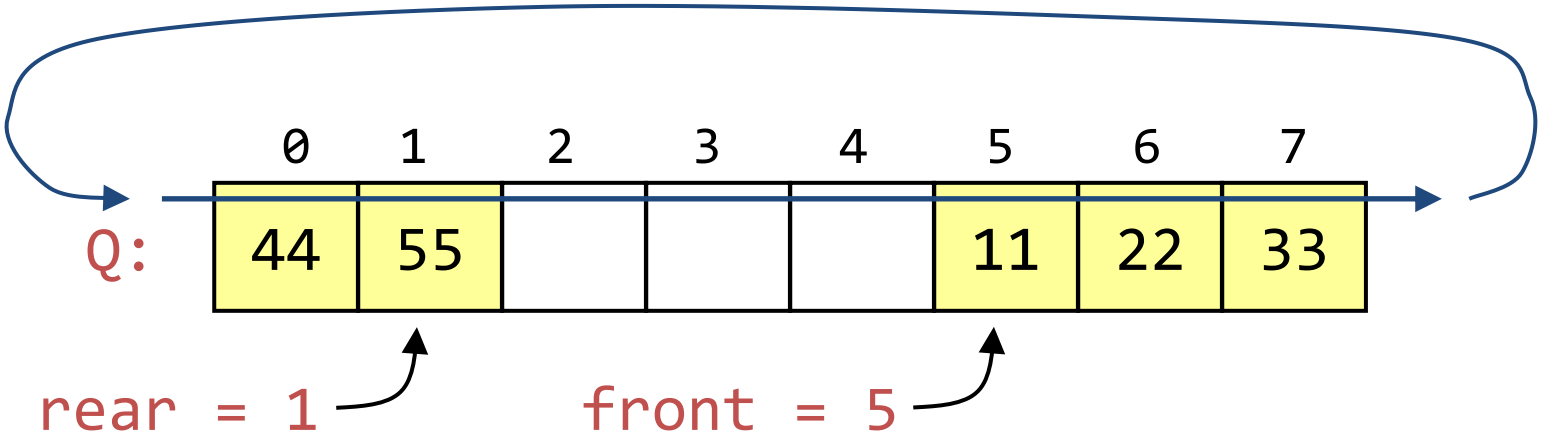
Array implementation of queues



- Notice the content of the array moves to the right as items are inserted and deleted
- This will be a problem after a while!

Circular arrays

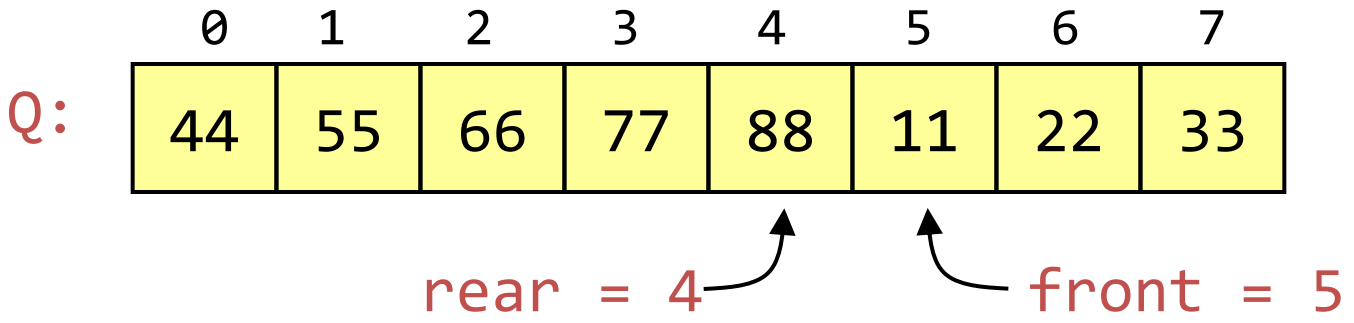
- We can treat the array holding the queue elements as circular (joined at the ends)



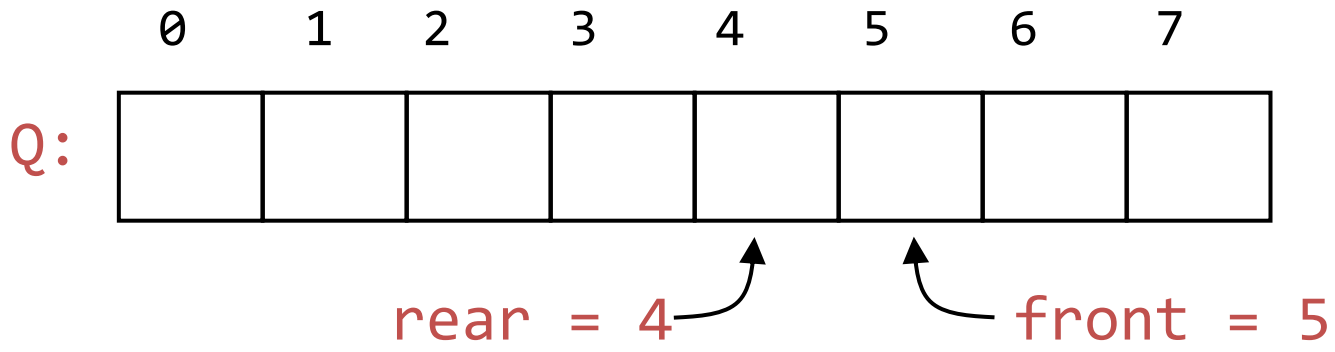
- Elements were added to this queue in the order 11, 22, 33, 44, 55, and will be removed in the same order
- Dequeue(Q) : Dequeue `Q[front]; front = (front + 1) % n;`
- Enqueue(Q,x): `rear = (rear + 1) % n; Q[rear] = x;`

Queue full or empty

- If the queue become completely full, it would look like this:



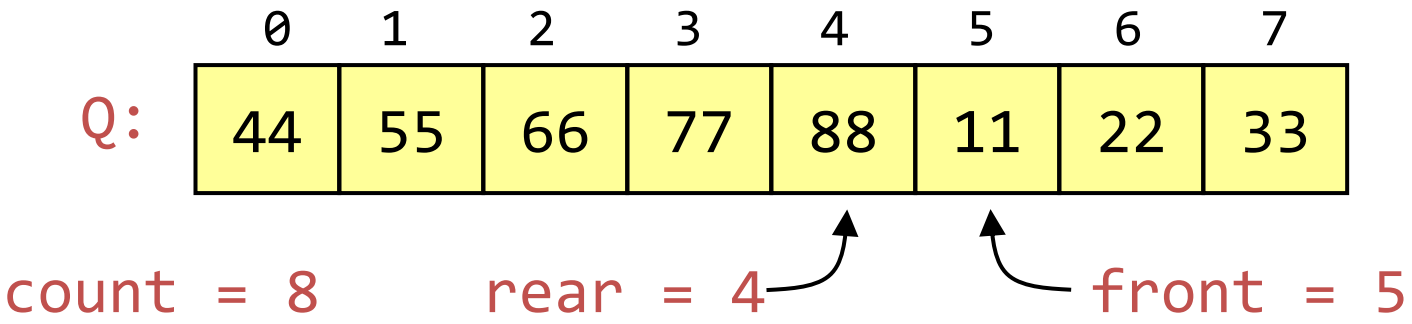
- If we remove all eight items, making the queue completely empty, it would look like this:



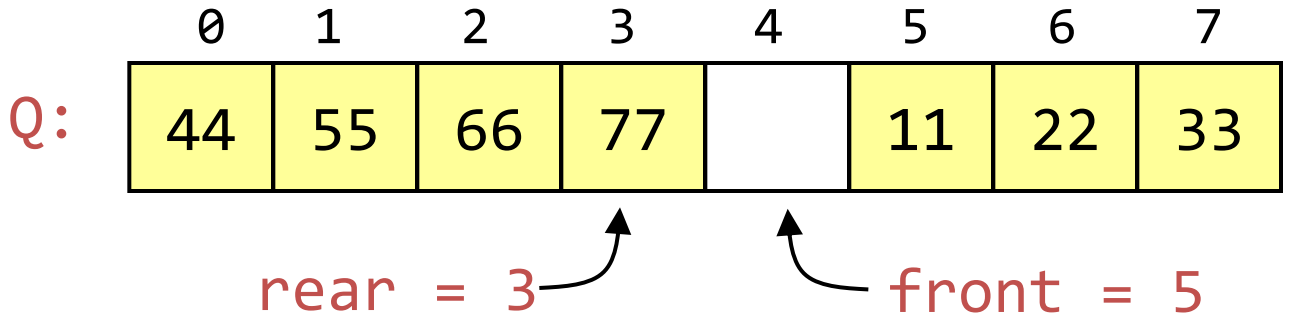
- Can't tell whether the queue is full or empty

Queues full or empty: solutions

- **Solution 1:** Keep an additional variable count which stores the current number of items in the queue

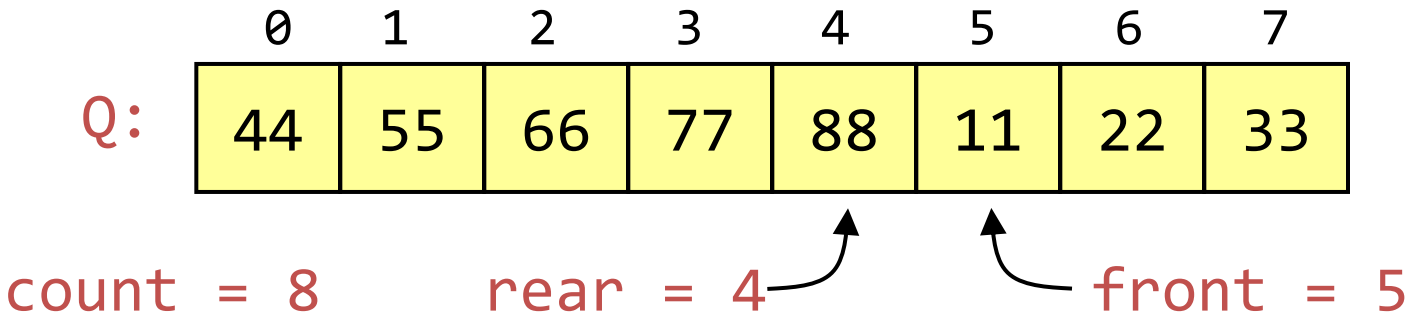


- **Solution 2:** Keep a gap between elements: consider the queue full when it has n-1 elements



Implementation of solution 1:

- **Solution 1:** Keep an additional variable



- Dequeue(Q) : `if (count == 0) return 'queue is empty';`

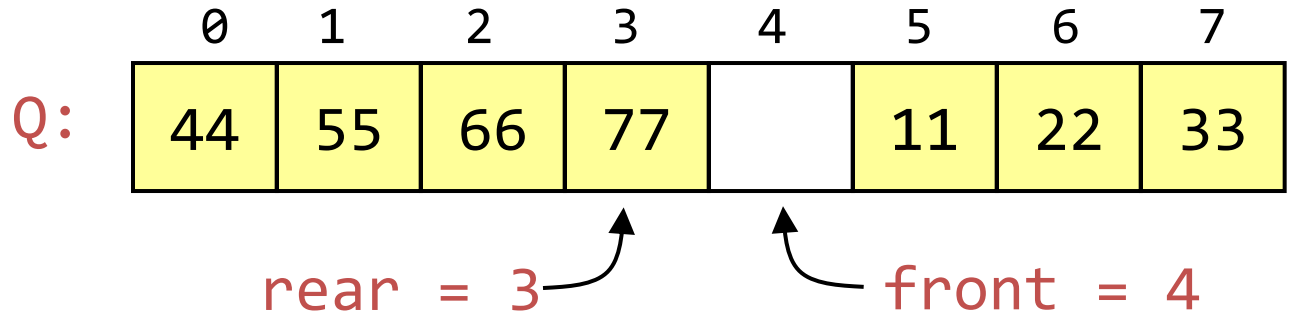
`else Dequeue Q[front]; front = (front + 1) % n; count--;`

- Enqueue(Q,x): `if (count == n) return 'queue is full';`

`else rear = (rear + 1) % n; Q[rear] = x; count++;`

Implementation of solution 2:

- **Solution 2:** the front pointer always point to the gap entry in the array



- Dequeue(Q) : `if (rear == front) return 'queue is empty';`

`else front = (front + 1) % n; Dequeue Q[front];`

- Enqueue(Q,x): `if (rear+1 == front) return 'queue is full';`

`else rear = (rear + 1) % n; Q[rear] = x;`

Implementing a Queue using Array: Examples

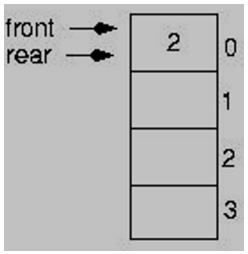
- Circular queue ["wrap around"]

Example 1: The array used to represent queue has maxSize = 4

initialize: Q is empty: front = 0; rear = -1;

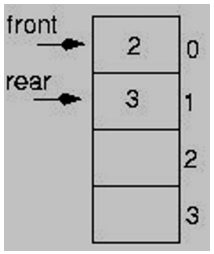
- Q[front]: the first item of the queue
- Q[rear]: the last item of the queue
- Add item to the Q (Enqueue): rear+=1;Q[rear]=item;
- Remove item from Q (Dequeue): remove Q[front]; then front+=1

enqueue(Q,2)



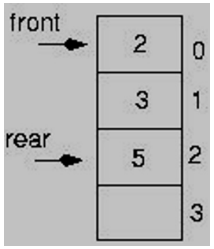
rear ++;
Q[rear] = 2;
Q = (2)

enqueue(Q,3)



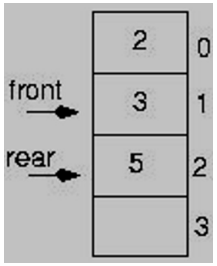
rear++;
Q[rear] = 3;
Q = (2, 3)

enqueue(Q,5)



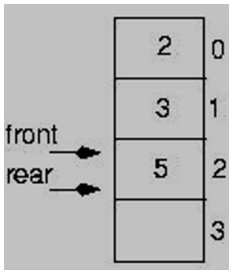
rear++;
Q[rear] = 5;
Q = (2, 3, 5)

dequeue(Q)



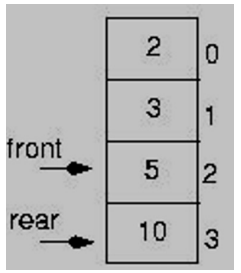
Dequeue Q[front]
Q = (3, 5)
front++;

dequeue(Q)



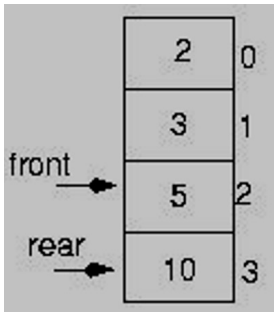
Dequeue Q[front]
Q = (5)
front++;

enqueue(Q,10)



rear++;
Q[rear] = 10;
Q = (5,10)

enqueue(Q,20) ???



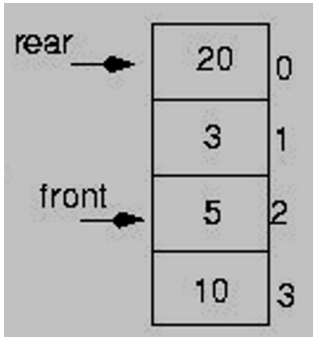
rear ++;
Q[rear] = 20;
➔ rear = 4 = maxSize
➔ Array Q over flow

Let the queue elements "wrap around"

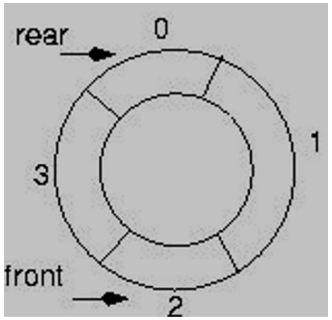
```
If (rear == maxSize -1)
     rear = 0;
else
      rear = rear +1;
```

Or

```
rear = (rear + 1) % maxSize;
```



Q = (5,10, 20)



Circular queue

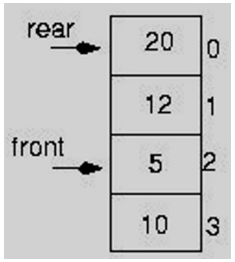
Implementing a Queue: using Array

- Circular queue ["wrap around"]

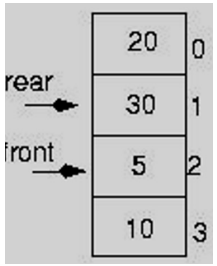
Example 2: The array used to represent queue has maxSize = 4

Q consists of 3 elements: Q = (5, 10, 20)

- Q[front]: the first item of the queue
- Q[rear]: the last item of the queue
- Add item to the Q (Enqueue): rear+=1;Q[rear]=item;
- Remove item from Q (Dequeue): remove Q[front]; then front+=1

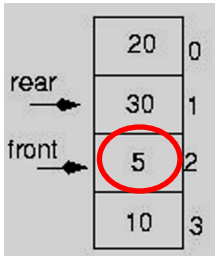


enqueue(Q,30)



rear++;
Q[rear] = 30;
Q = (5, 10, 20, 30)

enqueue(Q,50) ?? When Q is full already



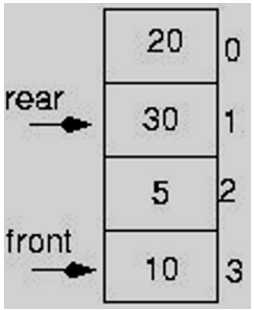
rear++;
Q[rear] = 50;
➔ **rear = 2 = front**

The queue Q is full!!!
What is the condition to determine that the queue is full ?

`rear + 1 == front`

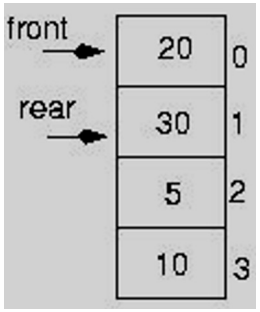
`We can not distinguish between the two cases: EMPTY and FULL !!!!!`

dequeue(Q)



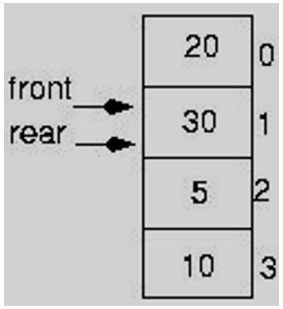
Dequeue Q[front]
Q = (10, 20, 30)
front++;

dequeue(Q)



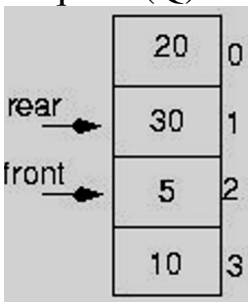
Dequeue Q[front]
Q = (20, 30)
front++;
=> front = 4=maxSize
=> "wrap around"
⇒ front = 0

dequeue(Q)



Dequeue Q[front]
Q = (30)
front++;

dequeue(Q)



Dequeue Q[front]
Q = empty
front++;

The queue Q is empty!!!
What is the condition for an empty queue ?

`rear + 1 == front`

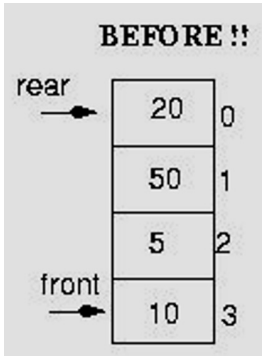
Implementing a Queue: using Array

- Circular queue ["wrap around"]

Solution 2: Make *front* point to the element **preceding** the front element in the queue (one memory location will be wasted).

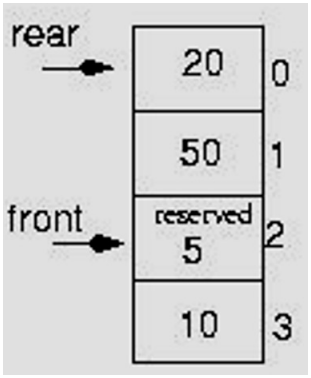
Example 3: illustration for solution 2

Solution 2: Make *front* point to the element **preceding** the front element in the queue (one memory location will be wasted).

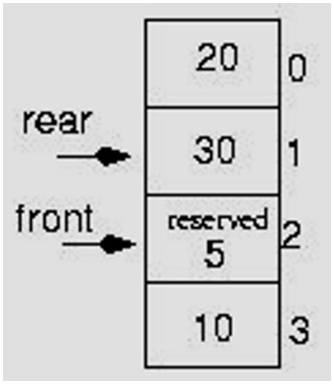


Q = (10, 20)

enqueue(Q, 30)



Q = (10, 20)

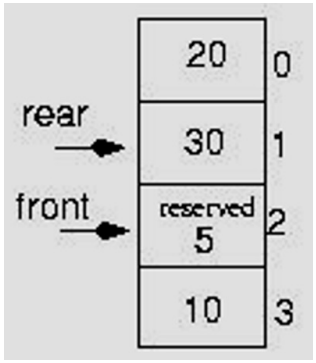


rear++; Q[rear] = 30;
Q = (10, 20, 30)

The queue Q is full!!!
What is the condition to determine that the queue is full ?

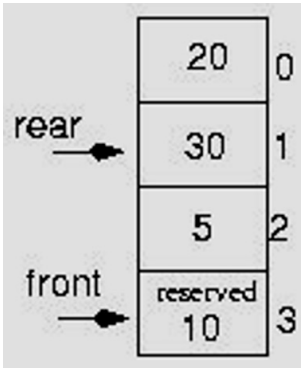
`rear + 1 == front`

dequeue(Q)



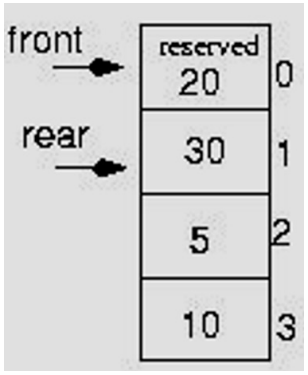
Q = (10, 20, 30)

dequeue(Q)



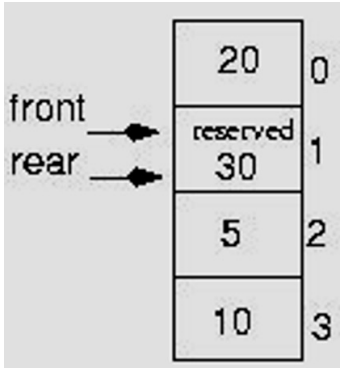
front++;
Dequeue Q[front]
Q = (20, 30)

dequeue(Q)



front++;
⇒ front = 4 = maxSize
⇒ wrap around: front = 0
Dequeue Q[front]
Q = (30)

dequeue(Q)



front++;
Dequeue Q[front]
Q = empty

The queue Q is empty!!!
What is the condition for an empty queue ?

`rear == front`

Based on this solution 1, one memory location on the array Q is wasted!!!

Implementing a Queue: using Array

- Circular queue ["wrap around"]

Solution 2: Make *front* point to the element **preceding** the front element in the queue (one memory location will be wasted).

Then what are the initial values for *front* and *rear*?

```
front = rear = maxSize – 1;
```

- Q[front+1]: the first item of the queue
- Q[rear]: the last item of the queue
- Add item to the Q (Enqueue): rear+=1; Q[rear]=item
- Remove item from Q (Dequeue): front+=1; then remove Q[front] from queue

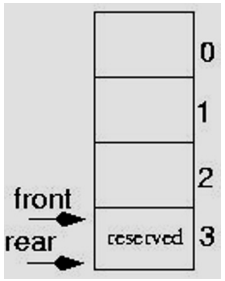
Example 4: The array used to represent queue has maxSize = 4

Initialize: `front = rear = maxSize – 1 = 3;`

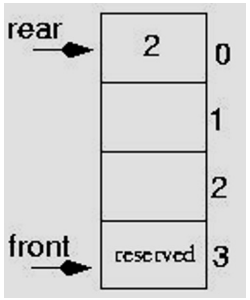
Queue Q is empty

illustration for solution 1

Init: `front = rear = 3;`

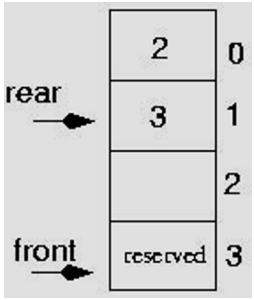


enqueue(Q, 2)



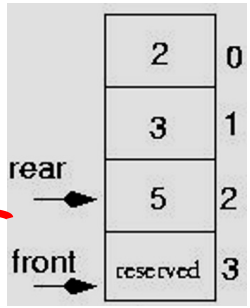
rear++;
Q[rear] = 2;
Q= (2)

enqueue(Q, 3)



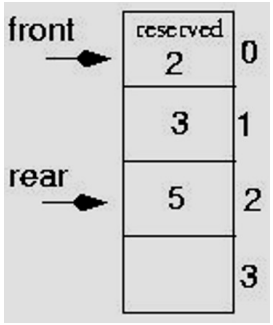
rear++;
Q[rear] = 3;
Q= (2, 3)

enqueue(Q, 5)



rear++;
Q[rear] = 5;
Q= (2, 3, 5)

dequeue(Q)



front++;
=> front = 4 = maxSize
=> wrap around : front = 0
Dequeue Q[front]
Q= (3, 5)

`rear + 1 == front`

`Queue Q full!!!`

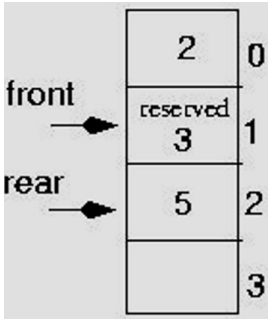
In general:

`(rear + 1)% maxSize == front`

enqueue(Q, 30) ??

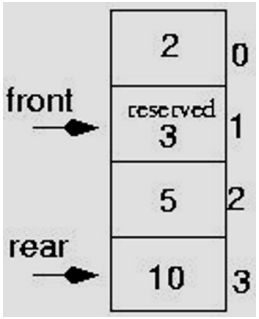
ERROR: Queue is full

dequeue(Q)



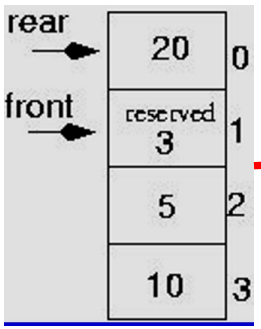
front++;
dequeue Q[front]
Q= (5)

enqueue(Q, 10)



rear++;
Q[rear] =10;
Q= (5, 10)

enqueue(Q, 20)



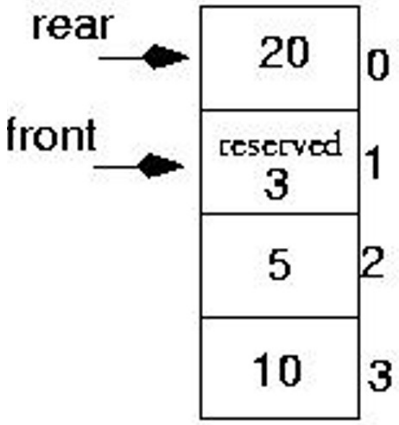
rear++;
Q[rear] =20;
Q= (5, 10, 20)

Example 4: The array used to represent queue has maxSize = 4

Initialize: `front = rear = maxSize – 1 = 3;`

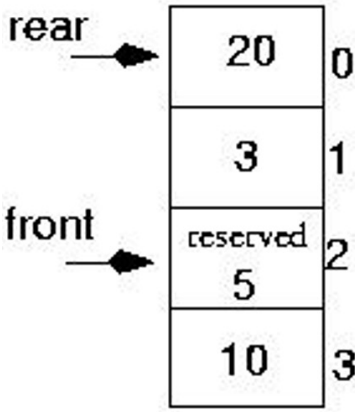
Queue Q is empty

illustration for solution 1



Q= (5, 10, 20)

dequeue(Q)

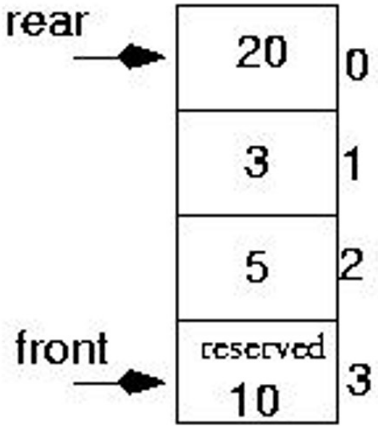


front++;

Dequeue Q[front]

Q= (10, 20)

dequeue(Q)

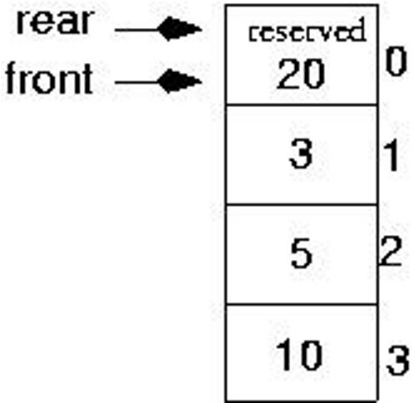


front++;

Dequeue Q[front]

Q= (20)

dequeue(Q)



front++;

=>front = 4 = maxSize

=> wrap around: front =0

Dequeue Q[front]

Q empty

Queue Q is empty now !!!

`rear == front`

Implementing a Queue: using Array

- Circular queue ["wrap around"]

Solution 2: Make *front* point to the element **preceding** the front element in the queue (one memory location will be wasted).

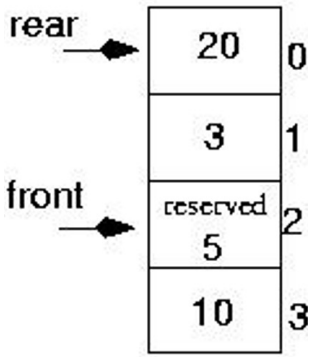
- The initial values for *front* and *rear* :

```
front = rear = maxSize – 1;
```

- Q[front+1]: the first item of the queue
- Q[rear]: the last item of the queue
- Add item to the Q (Enqueue):
  - rear+=1; `if (rear == maxSize) rear = 0;`
  - Q[rear]=item
- Remove item from Q (Dequeue):
  - `front = (front + 1) % maxSize;`
  - then remove Q[front] from queue
- Detect queue is empty: `rear == front`
- Detect queue is full: `(rear + 1) % maxSize == front`

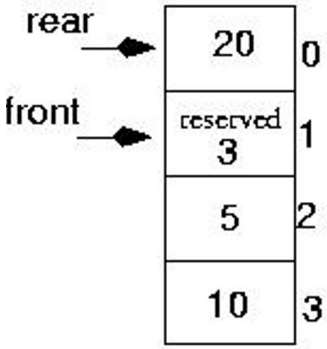
Solution 2: Make *front* point to the element **preceding** the front element in the queue (one memory location will be wasted)

```
init(int max) //initialize empty queue Q
{
    maxSize = max;
    front = maxSize – 1;
    rear = maxSize – 1;
    Q = new ItemType[maxSize];
}
int sizeQ(Q) //returns the number of elements currently in the queue Q
{
      int size = (maxSize – front + rear) % maxSize;
      return size;
}
```



Q = (10, 20)

#elements = (4 – 2+ 0) % 4 = 2



Q = (5, 10, 20)

#elements = (4 – 1+ 0) % 4 = 3

Solution 2: Make *front* point to the element **preceding** the front element in the queue (one memory location will be wasted)

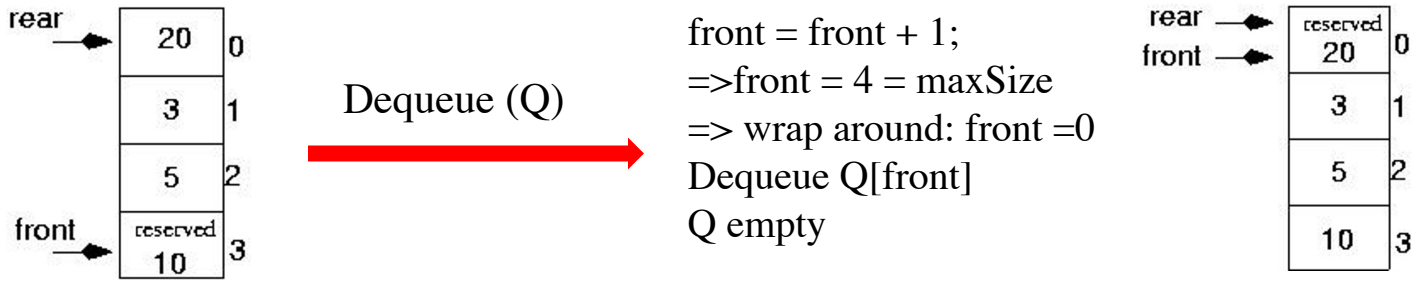
```
isEmpty(Q) //  returns "true“ if queue Q is empty
{
    if (rear == front) return true;
    else return false;
}
isFull(Q)  /*returns "true“ if Q is full, indicates that we already use  the maximum memory for queue; otherwise returns “false” */
{
    if ((rear + 1) % maxSize == front) return true;
    else return false;
}
frontQ(Q)  //returns the item that is in front (head) of queue Q or returns error if queue Q is empty.
{
    return Q[front + 1];
}
```

Solution 2: Make *front* point to the element **preceding** the front element in the queue (one memory location will be wasted)

```
enqueue(Q,x)  /*inserts item  x into the back (rear) of queue Q. If the queue is full before making insertion, then give the notification about that*/
{
   if (isFull(Q)) ERROR("Queue is FULL");
   else
   {  rear ++;
      if (rear == maxSize) rear = 0;
      Q[rear] = x;
   }
}
enqueue(Q,x)
{
   if (isFull(Q)) ERROR("Queue is FULL");
   else
   {   rear = (rear + 1) % maxSize;
       Q[rear] = x;
   }
}
```

Solution 2: Make *front* point to the element **preceding** the front element in the queue (one memory location will be wasted)

```
dequeue(Q)  /*deletes the element at the front (head) of the queue Q, then returns x which is the data of this element. If the queue Q is empty before dequeue, then give the error notification*/
{
   if (isEmpty(Q)) ERROR("Queue is EMPTY");
   else
   {    front = (front + 1);
        if (front == maxSize) front = 0;
        return Q[front];
   }
}
dequeue(Q)
{  if (isEmpty(Q)) ERROR("Queue is EMPTY");
   else
   {   front = (front + 1) % maxSize;
       return Q[front];
   }
}
```



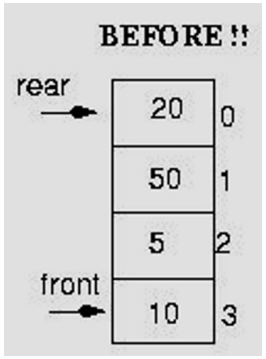
Implementing a Queue: using Array

- Circular queue ["wrap around"]

Solution 2: Make *front* point to the element **preceding** the front element in the queue (one memory location will be wasted).

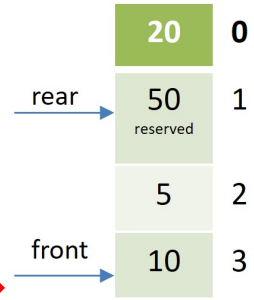
Solution 3: Make rear point to the element **posterior** the rear element in the queue (one memory location will be wasted).

Example 5: illustration for solution 3



Q = (10, 20)

Solution 3: Make *rear* point to the element **posterior** of the rear element in the queue (one memory location will be wasted).



Q = (10, 20)

enqueue(Q, 30)

?

Q[rear] = 30; rear++;
Q = (10, 20, 30)

The queue Q is full!!!
What is the condition to determine that the queue is full ?

`rear + 1 == front`

dequeue(Q)

?

front++;
⇒ front = 4 = maxSize
⇒ wrap around: front = 0
Q = (20, 30)

dequeue(Q)

?

front++;
Q = (30)

dequeue(Q)

?

front++;
Q = empty

The queue Q is empty!!!
What is the condition for an empty queue ?

`rear == front`

Based on this solution 3, one memory location on the array Q is wasted!!!

Implementing a Queue: using Array

- Circular queue ["wrap around"]

Solution 1: Make *front* point to the element **preceding** the front element in the queue (one memory location will be wasted).

Then for solution 1: what are the initial values for *front* and *rear* ?

```
front = rear = maxSize – 1;
```

Solution 3: Make rear point to the element **posterior** the rear element in the queue (one memory location will be wasted).

Then for solution 3: what are the initial values for *front* and *rear* ?

```
front = rear = 0;
```

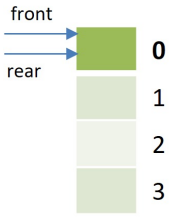
Example 6: The array used to represent queue has maxSize = 4

Initialize: `front = rear = 0;`

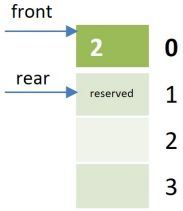
Queue Q is empty

illustration for solution 3

Init: `front = rear = 0;`



enqueue(Q, 2)



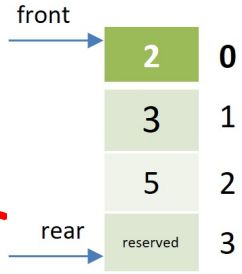
Q[rear] = 2;
rear++;
Q= (2)

enqueue(Q, 3)

?

Q[rear] = 3;
rear++;
Q= (2, 3)

enqueue(Q, 5)



Q[rear] = 5;
rear++;
Q= (2, 3, 5)

dequeue(Q)

?

Dequeue Q[front]
front++;
Q= (3, 5)

`(rear + 1) % maxSize == front`

`Queue Q full!!!`

dequeue(Q)

?

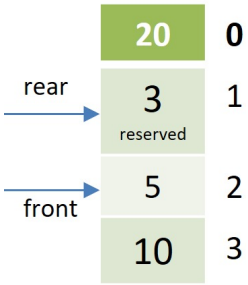
dequeue Q[front]
front++;
Q= (5)

enqueue(Q, 10)

?

Q[rear] =10;
rear++;
Q= (5, 10)

enqueue(Q, 20)



Q[rear] =20;
rear++;
Q= (5, 10, 20)

enqueue(Q, 30) ??

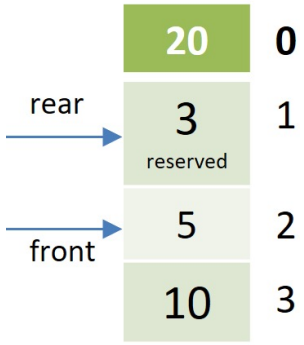
ERROR: Queue is full

Example 6: The array used to represent queue has maxSize = 4

Initialize: `front = rear = 0;`

Queue Q is empty

illustration for solution 3



Q= (5, 10, 20)

dequeue(Q)

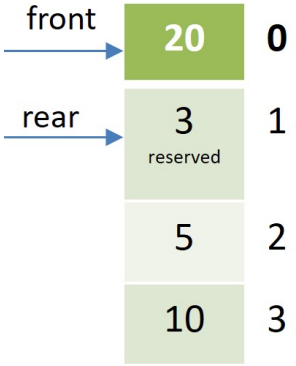
?

Dequeue Q[front]

front++;

Q= (10, 20)

dequeue(Q)



Dequeue Q[front]

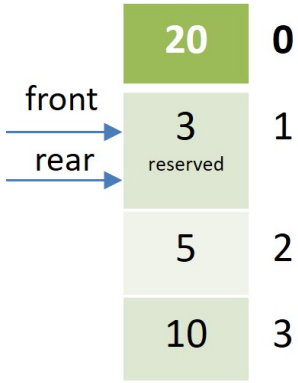
front++;

=>front = 4 =maxSize

=> wrap around: front =0

Q= (20)

dequeue(Q)



Dequeue Q[front]

front++;

Q empty

Queue Q is empty now !!!

`rear == front`

Solution 3: Make *rear* point to the element **posterior** the rear element in the queue (one memory location will be wasted)

Exercise: Write following functions for queue Q in the case of Solution 3

- `isEmpty(Q);` returns "true“ if queue Q is empty
- `isFull(Q);` returns "true“ if Q is full, indicates that we already use the maximum memory for queue; otherwise returns “false”
- `frontQ(Q);` returns the item that is in front (head) of queue Q or returns error if queue Q is empty.
- `enqueue(Q,x);` inserts item x into the back (rear) of queue Q. If before making insertion, the queue Q is full, then give the notification about that.
- `x = dequeue(Q);` deletes the element at the front (head) of the queue Q, then returns x which is the data of this element. If the queue Q is empty before dequeue, then give the error notification.
- `sizeQ(Q);` returns the number of elements currently in the queue Q.

Application 1: recognizing palindromes

- A *palindrome* is a string that reads the same forward and backward.

Example: NOON, DEED, RADAR, MADAM

Able was I ere I saw Elba

- How to recognize a given string is a palindrome or not:
  - Step 1: We will put all characters of the string into both a stack and a queue.
  - Step 2: Compare the contents of the stack and the queue character-by-character to see if they would produce the same string of characters:
    - If yes: the given string is a palindrome
    - Otherwise: not palindrome

Example 1: Whether "RADAR" is a palindrome or not

Step 1: Put "RADAR" into Queue and Stack:

Current character	Queue (front on the left, rear on the right)	Stack (top on the left)
R	R	R
A	R A	A R
D	R A D	D A R
A	R A D A	A D A R
R	R A D A R	R A D A R

front rear top

Example 1: Whether “RADAR” is a palindrome or not

Step 2: Delete “RADAR” from Queue and Stack:

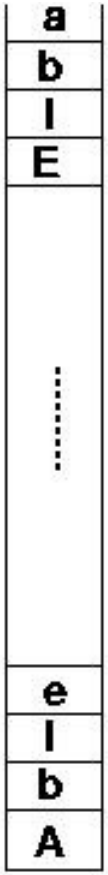
- Dequeue until the queue is empty
- Pop the stack until the stack is empty

Queue (front on the left)	Front of Queue	Top of Stack	Stack (top on the left)
R A D A R	R	R	R A D A R
A D A R	A	A	A D A R
D A R	D	D	D A R
A R	A	A	A R
R	R	R	R
empty	empty	empty	empty

Conclusion: String "RADAR" is a palindrome

Example 2: recognizing palindromes

Able was I ere I saw Elba



Stack

Queue

Application 2: Convert a string of digits into a decimal number

The algorithm is described as following:

```
// Convert sequence of digits stored in queue Q into decimal number n
// Remove empty space if any
do {      dequeue(Q, ch)
} until ( ch != blank)
//  ch is now the first digit of the given string
// Calculate n from sequence of digit in the queue
n = 0;
done = false;
do {      n = 10 * n + decimal number that ch represents;
          if (! isEmpty(Q) )
                    dequeue(Q,ch)
          else
                    done = true
}  until ( done  || ch != digit)
// Result: n is the decimal number need to be found
```

Implementing a Queue

- Just like a stack, we can implementing a queue in two ways:
  - Using an array
  - **Using a linked list**

Implementing a Queue: using a linked list

```
typedef struct {
        DataType element;
        struct node *next;
} node;
typedef struct {
      node *front;
      node *rear;
} queue;
```

where `DataType` is data type of the object need to store in the queue; `DataType` need to be declared before declaring the queue.

- Implementing a queue using a linked list:
  - Front of the queue is stored as the head node of the linked list, rear of the queue is stored as the tail node.
  - *Enqueue* by adding to the end of the list
  - *Dequeue* by removing from the front of the list.