Data structures with C++

2ND SATGE

ASSIST LECTURER: ASAN BAKER

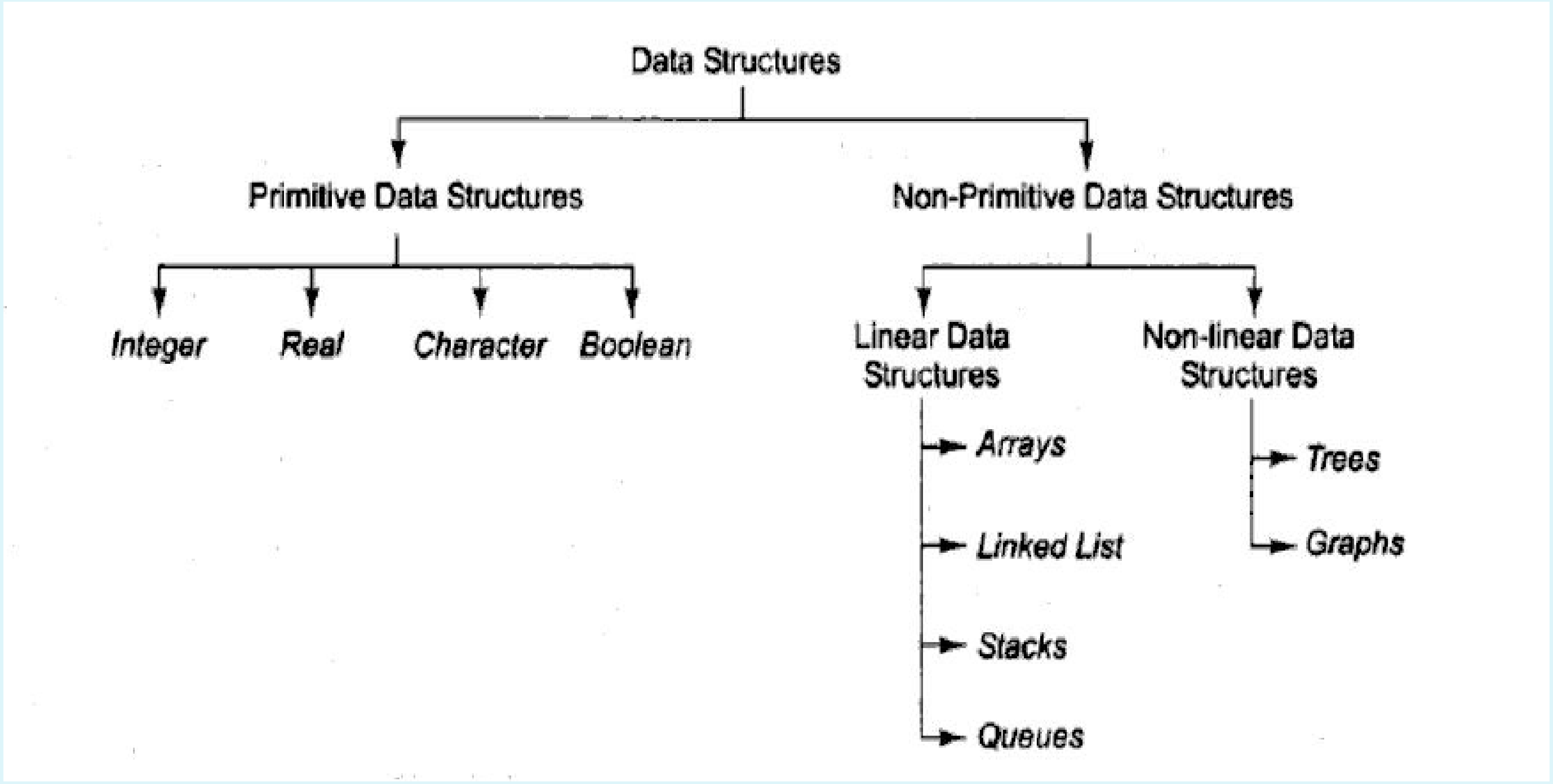
Data structures

- Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage.

Basic types of data structures

- Anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.
- Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Non-Primitive Data Structures** are :
  - Array
  - Linked List
  - Tree
  - Graph
  - Stack, Queue etc.

Diagram of types of data structures



ALGORITHMS

- Definition: An Algorithm is a method of representing the step-by-step procedure for solving a problem. It is a method of finding the right answer to a problem by breaking the problem into simple cases.
- An algorithm can be written in English like sentences or in any standard representations. The algorithm written in English language is called Pseudo code.

Algorithm must possess the following properties:

- 1. Finiteness: An algorithm should terminate in a finite number of steps.
- 2. Definiteness: Each step of the algorithm must be precisely (clearly) stated.
- 3. Effectiveness: Each step must be effective. i.e.; it should be easily convertible into program statement and can be performed exactly in a finite amount of time.
- 4. Generality: Algorithm should be complete in itself, so that it can be used to solve all problems of given type for any input data.
- 5. Input/output: Each algorithm must take zero, one or more quantities as input data and gives one of more output values.

- Example: To find the average of 3 numbers, the algorithm is as shown below.
- Step1: Read the numbers a, b, c, and d.
- Step2: Compute the sum of a, b, and c.
- Step3: Divide the sum by 3.
- Step4: Store the result in variable of d.
- Step5: End the program.

Development Of An Algorithm

The steps involved in the development of an algorithm are as follows:

- Specifying the problem statement.
- Designing an algorithm.
- Coding.
- Debugging
- Testing and Validating
- Documentation and Maintenance.

PERFORMANCE ANALYSIS

- When several algorithms can be designed for the solution of a problem, there arises the need to determine which among them is the best. The efficiency of a program or an algorithm is measured by computing its time and/or space complexities

- The time complexity of an algorithm is a function of the running time of the algorithm.
- The space complexity is a function of the space required by it to run to completion.
- The time complexity is therefore given in terms of frequency count.
- Frequency count is basically a count denoting number of times a statement execution

Best Case, Worst Case and Average Case Analysis

- If an algorithm takes minimum amount of time to run to completion for a specific set of input then it is called best case complexity.
- If an algorithm takes maximum amount of time to run to completion for a specific set of input then it is called worst case time complexity.
- The time complexity that we get for certain set of inputs is as an average same. Then for corresponding input such a time complexity is called average case time complexity.

Arrays

- An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

  That means that, for example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.

One-Dimensional Arrays

- A one-dimensional array is a list of related variables. The general form of a onedimensional array declaration is

  typevar name[size];

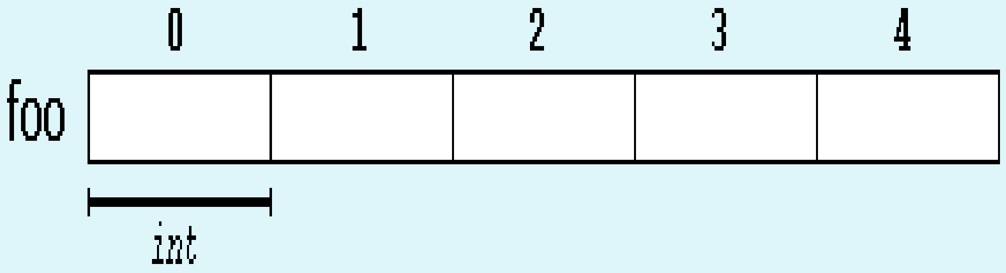
- Here, *type* declares the base type of the array. The base type determines the data type of each element that comprises the array(such as int, float…). *size* defines how many elements the array will hold. For example, the following declares an integer array named **sample** that is ten elements long:

  int sample[10];

An index identifies a specific element within an array.

- An other example, an array containing 5 integer values of type int called foo could be represented as:

int foo [5];



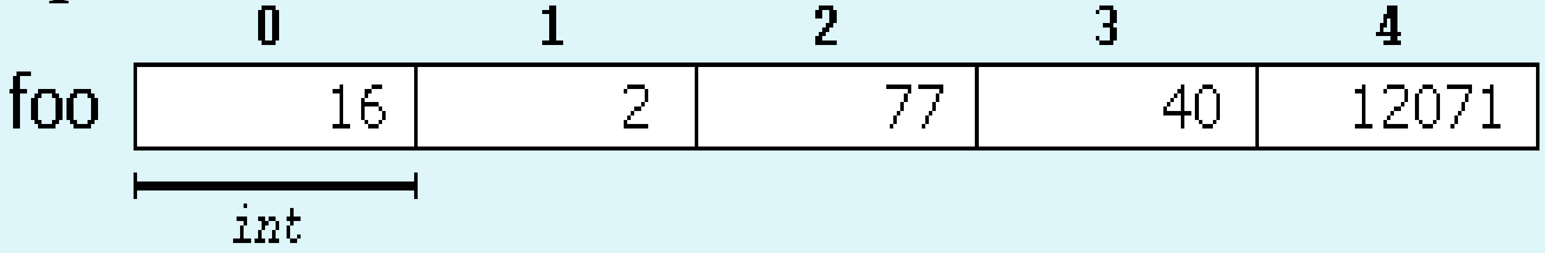
- where each blank panel represents an element of the array. In this case, these are values of type int. These elements are numbered from 0 to 4, being 0 the first and 4 the last; In C++, the first element in an array is always numbered with a zero (not a one), no matter its length.

Initializing arrays

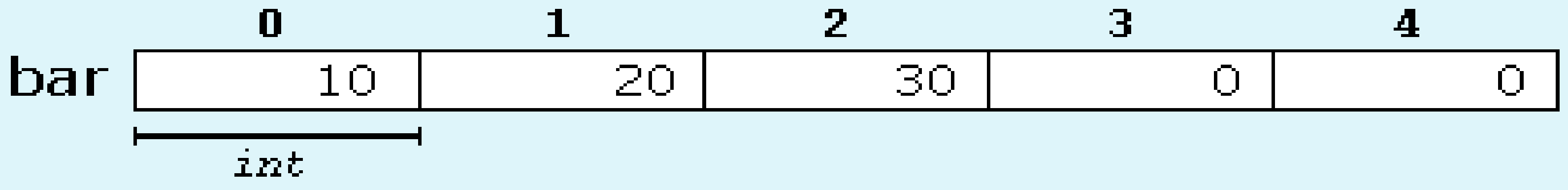
- the elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces {}. For example:

- int foo [5] = { 16, 2, 77, 40, 12071 };

- This statement declares an array that can be represented like this:



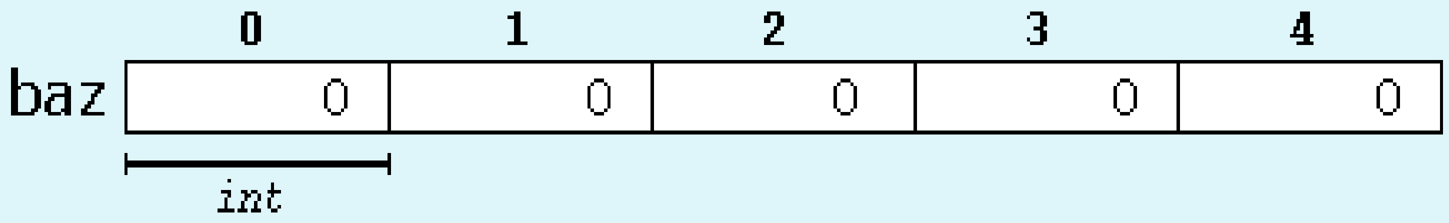
- The number of values between braces {} shall not be greater than the number of elements in the array. For example, in the example above, foo was declared having 5 elements (as specified by the number enclosed in square brackets, [], and the braces {} contained exactly 5 values, one for each element.
- If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes). For example:
- int bar [5] = { 10, 20, 30 };
  Will create an array like this:



- The initializer can even have no values, just the braces:

  int baz [5] = { };

  This creates an array of five int values, each initialized with a value of zero:



- When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty[]. In this case, the compiler will assume automatically a size for the array that matches the number of values included between the braces {}:

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

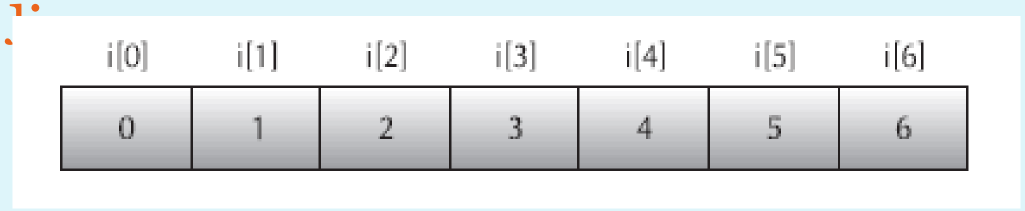
- After this declaration, array foo would be 5 int long, since we have provided 5 initialization values.

Accessing the values of an array

- In C++, all arrays consist of contiguous memory locations. (That is, the array elements reside next to each other in memory.) The lowest address corresponds to the first element, and the highest address to the last element. For example, after this fragment is run,

```
int i[7];
int j;
for(j=0; j<7; j++) i[j] = j;
```

- **i** looks like this:



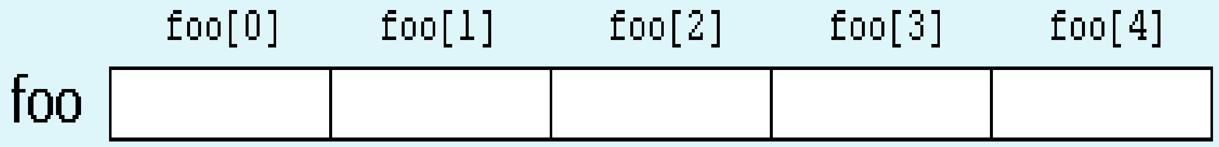
- For a one-dimensional array, the total size of an array in bytes is computed as shown here:
- total bytes = number of bytes in type × number of elements

Accessing the values of an array

- The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:

  name[index]

  Following the previous examples in which foo had 5 elements and each of those elements was of type int, the name which can be used to refer to each element is the following:



- For example, the following statement stores the value 77 in the third element of foo:

```
foo [2] = 77;
```

and, for example, the following copies the value of the third element of foo to a variable called x:

```
x = foo[2];
```

Therefore, the expression foo[2] is itself a variable of type int.

- Do not confuse these two possible uses of brackets [] with arrays.

```
int foo[5];        // declaration of a new array
foo[2] = 77;        // access to an element of the array.
```

The main difference is that the declaration is preceded by the type of the elements, while the access is not.

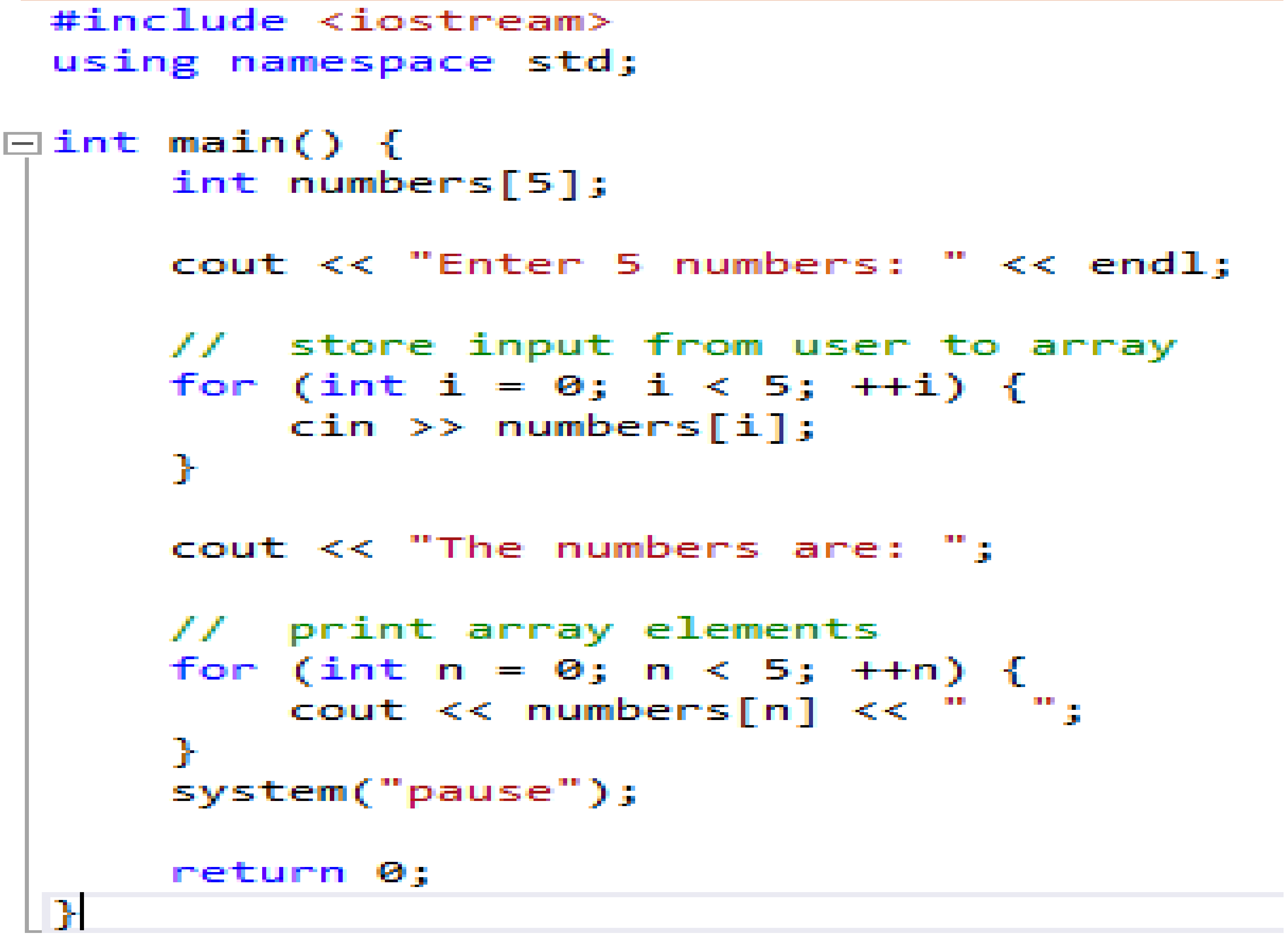
```
#include <iostream>
using namespace std;
int main()
{
int sample[10]; // this reserves 10 integer elements
int t;
// load the array
for(t=0; t<10; ++t) sample[t]=t;
// display the array
for(t=0; t<10; ++t) cout << sample[t] << ' ';
system("pause");
return 0;
}
```

Array example

```
// arrays example
#include <iostream>
using namespace std;

int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ( )
{
  for ( n=0 ; n<5 ; ++n )
  {
    result += foo[n];
  }
  cout << result;
  system("pause");
  return 0;
}
```

Take Inputs from User and Store Them in an Array

```
#include <iostream>
using namespace std;

int main() {
   int numbers[5];

   cout << "Enter 5 numbers: " << endl;

   //  store input from user to array
   for (int i = 0; i < 5; ++i) {
      cin >> numbers[i];
   }
 cout << "The numbers are: ";

   //  print array elements
   for (int n = 0; n < 5; ++n) {
      cout << numbers[n] << "  ";
   }
system("pause");
   return 0;
}
```



```cpp
#include <iostream>
using namespace std;

int main() {
    int numbers[5];

    cout << "Enter 5 numbers: " << endl;

    //  store input from user to array
    for (int i = 0; i < 5; ++i) {
        cin >> numbers[i];
    }

    cout << "The numbers are: ";

    //  print array elements
    for (int n = 0; n < 5; ++n) {
        cout << numbers[n] << "  ";
    }
    system("pause");

    return 0;
}
```

Character sequences(strigs)

- By far, the most common use for one-dimensional arrays is to create character strings. In C++, a *string* is defined as a character array that is terminated by a null. A null character is specified using **'\0'**, BackSlash and is zero. Because of the null terminator, it is necessary to declare a character array to be one character longer than the largest string that it will hold.
- For example, if you want to declare an array **str** that could hold a 10-character string, you would write:

```
char str[11];
```

Specifying the size as 11 makes room for the null at the end of the string.

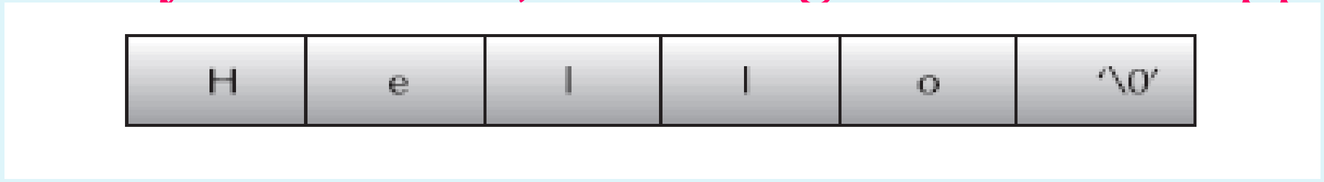
A string is a null-terminated character array.

- As you learned earlier, C++ allows you to define a string literal. Recall that a *string literal* is a list of characters enclosed in double quotes. Here are some examples:

  "hello there"   "I like C++"

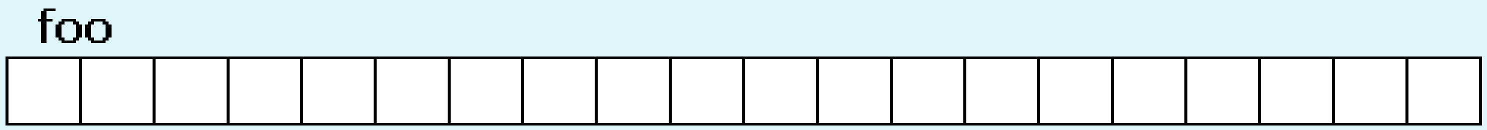
  "#$%@@#$"   ""

- The last string shown is "". This is called a *null string*. It contains only the null terminator, and no other characters. Null strings are useful because they represent the empty string.
- It is not necessary to manually add the null onto the end of string constants; the C++ compiler does this for you automatically. Therefore, the string "Hello" will appear in



char foo [20];

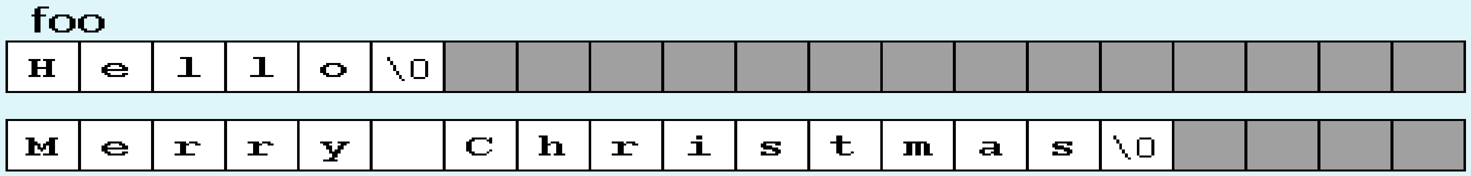
is an array that can store up to 20 elements of type char. It can be represented as:



Therefore, this array has a capacity to store sequences of up to 20 characters. But this capacity does not need to be fully exhausted: the array can also accommodate shorter sequences. For example, at some point in a program, either the sequence "Hello" or the sequence "Merry Christmas" can be stored in foo, since both would fit in a sequence with a capacity for 20 characters.

- By convention, the end of strings represented in character sequences is signaled by a special character: the *null character*, whose literal value can be written as '\0' (backslash, zero).

  In this case, the array of 20 elements of type char called foo can be represented storing the character sequences "Hello"and "Merry Christmas" as:

  Notice how after the content of the string itself, a null character ('\0') has been added in order to indicate the end of the sequence. The panels in gray color represent char elements with undetermined values.

Initialization of null-terminated character sequences

- Because arrays of characters are ordinary arrays, they follow the same rules as these. For example, to initialize an array of characters with some predetermined sequence of characters, we can do it just like any other array:

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

- The previous declares an array of 6 elements of type char initialized with the characters that form the word "Hello" plus a*null character* '\0' at the end.

  As mentioned previously arrays of character elements have another way to be initialized: using *string literals* directly.

  string literals have already shown up several times. These are specified by enclosing the text between double quotes ("). For example:

  "the result is: "

- Sequences of characters enclosed in double-quotes (") are *literal constants*. And their type is, in fact, a null-terminated array of characters. This means that string literals always have a null character ('\0') automatically appended at the end.

  Therefore, the array of char elements called myword can be initialized with a null-terminated sequence of characters by either one of these two statements:
- char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
- char myword[] = "Hello";

- In both cases, the array of characters myword is declared with a size of 6 elements of type char: the 5 characters that compose the word "Hello", plus a final null character ('\0'), which specifies the end of the sequence and that, in the second case, when using double quotes (") it is appended automatically.

NULL-TERMINATED

```
char array1[10] = {‘c’,’u’,’p’};
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
c	u	p	?	?	?	?	?	?	?

```
char array2[10] = “cup”;
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
c	u	p	\0	?	?	?	?	?	?

Reading a String from the Keyboard

- The easiest way to read a string entered from the keyboard is to make the array that will receive the string the target of a **cin** statement. For example, the following program reads a string entered by the user:

```
// Using cin to read a string from the keyboard.
#include <iostream>
using namespace std;
int main()
{
char str[80];
cout << "Enter a string: ";
cin >> str; // read string from keyboard
cout << "Here is your string: ";
cout << str;
System("pause");
return 0;
}
```

- Although this program is technically correct, there is still a problem. To see what it is, examine the following sample run.

```
Enter a string: This is a test
Here is your string: This
```

- As you can see, when the program redisplays the string, it shows only the word "This", not the entire sentence that was entered. The reason for this is that the >> operator stops reading a string when the first *whitespace* character is encountered. Whitespace characters include spaces, tabs, and newlines

- One way to solve the whitespace problem is to use another of C++'s library functions, **gets( )**. The general form of a **gets( )** call is:

  gets(*array-name*);

- If you need your program to read a string, call **gets( )** with the name of the array, without any index, as its argument. Upon return from **gets( )**, the array will hold the string input from the keyboard. The **gets( )** function will continue to read characters until you press ENTER. The header used by **gets( )** is **<cstdio>.**

  This version of the preceding program uses **gets( )** to allow the entry of strings containing spaces.

```
// Using gets() to read a string from the keyboard.
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
char str[80];
cout << "Enter a string: ";
gets(str); // read a string from the keyboard
cout << "Here is your string: ";
cout << str;
System("pause");
return 0;
}
```

- Now, when you run the program and enter the string "This is a test", the entire sentence is read and then displayed, as this sample run shows.

```
Enter a string: This is a test
Here is your string: This is a test.
```

Example

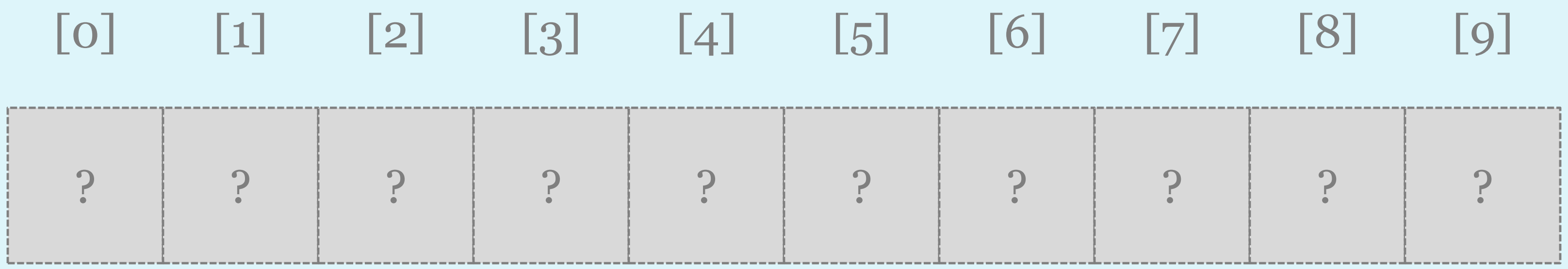
```
#include <iostream>
#include <string>
using namespace std;
int main ( )
{
  char question1[] = "What is your name? ";
  string question2 = "Where do you live? ";
  char answer1 [80];
  string answer2;
  cout << question1;
  cin >> answer1;
  cout << question2;
  cin >> answer2;
  cout << "Hello, " << answer1;
  cout << " from " << answer2 << "!\n";
  system("pause");
  return 0;
}
```

```
What is your name? Omer
Where do you live? Greece
Hello, Omer from Greece!
```

Example

```
char name[10];
cout << “enter name: “;
cin >> name;
cout << name;
```

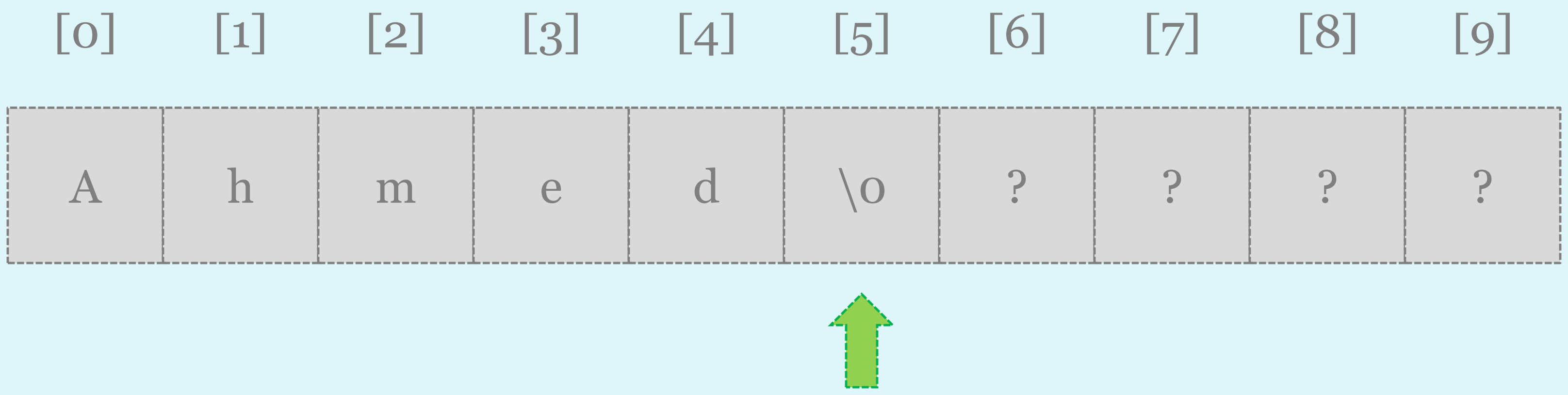
```
char name[10];
cout << “enter name: “;
cin >> name;
cout << name;
```



```
char name[10];
cout << "enter name: ";
cin >> name; // user inputs Ahmed
cout << name;
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
?	?	?	?	?	?	?	?	?	?

```
char name[10];
cout << "enter name: ";
cin >> name; // user inputs Ahmed
cout << name;
```

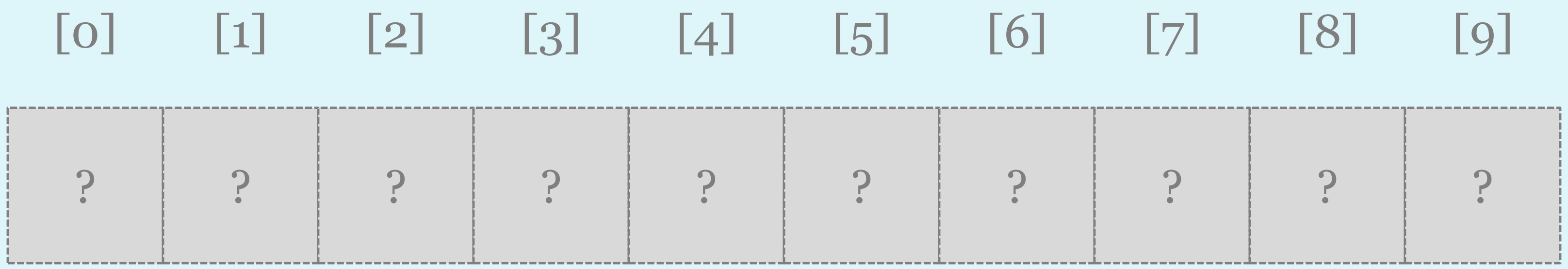


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

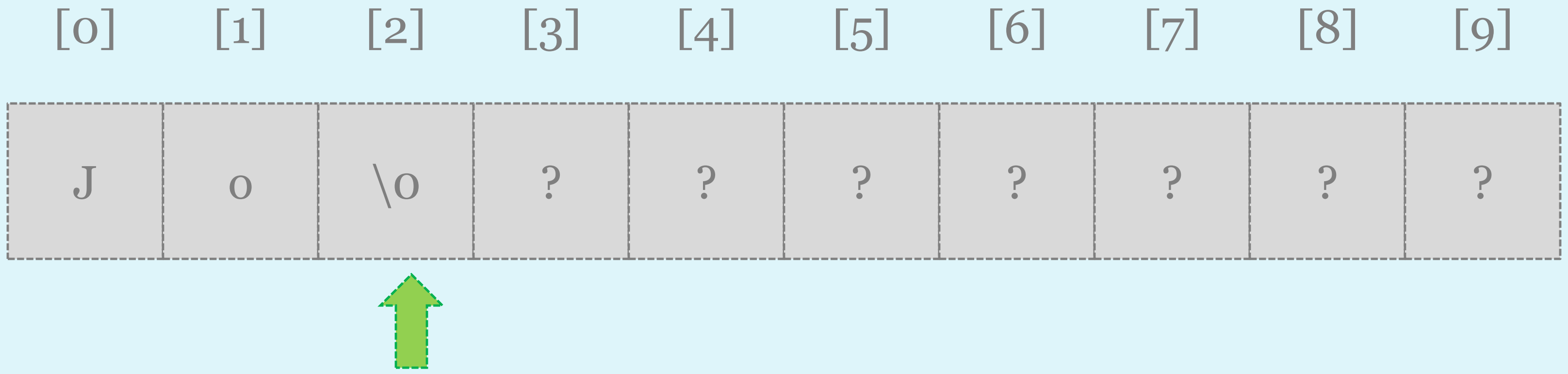
```
char name[10];
cout << "enter name: ";
cin >> name;
cout << name; //Ahmed is displayed
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

```
char name[10];
cout << "enter name: ";
cin >> name; // user inputs Jo Ann
cout << name;
```



```
char name[10];
cout << "enter name: ";
cin >> name; // user inputs Jo Ann
cout << name;
```



```
char name[10];
cout << “enter name: “;
cin >> name;
cout << name; //Jo is displayed
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
J	o	\o	?	?	?	?	?	?	?

Using the Null-Terminating Character

```
// ex1: char array must have null character at the end of data.
long string_length (const char ntca[])
{
    long length = 0;
    while (ntca[length] != ‘\0’)
        length++;

    return length;
}
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
long string_length (const char ntca[])
{
    long length = 0;
    while (ntca[length] != ‘\0’)
        length++;

    return length;
}
```

length = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
long string_length (const char ntca[])
{
    long length = 0;
    while (ntca[length] != '\0')
        length++;

    return length;
}
```

length = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
long string_length (const char ntca[])
{
    long length = 0;
    while (ntca[length] != ‘\0’)
       length++;

    return length;
}
```

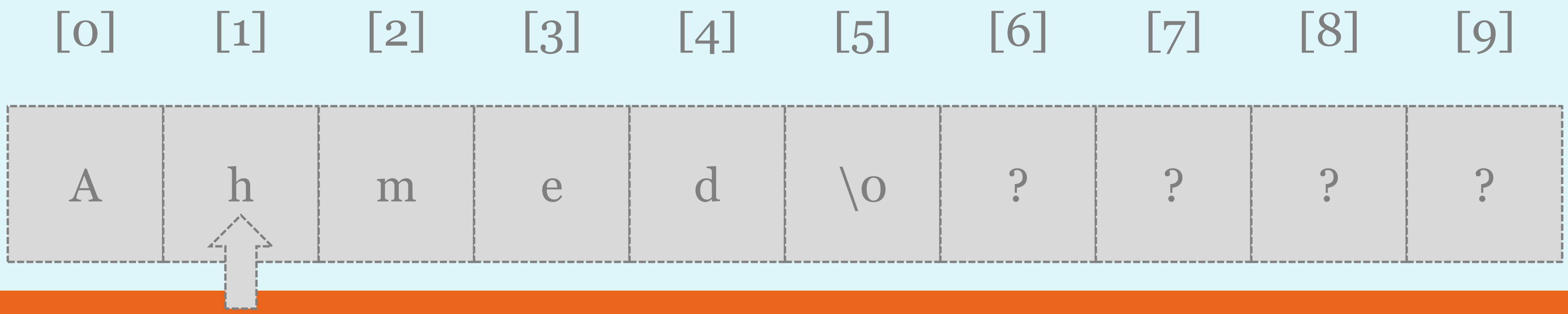
length = 1

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
long string_length (const char ntca[])
{
    long length = 0;
    while (ntca[length] != '\0')
        length++;

    return length;
}
```

length = 1



Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
long string_length (const char ntca[])
{
    long length = 0;
    while (ntca[length] != '\0')
        length++;

    return length;
}
```

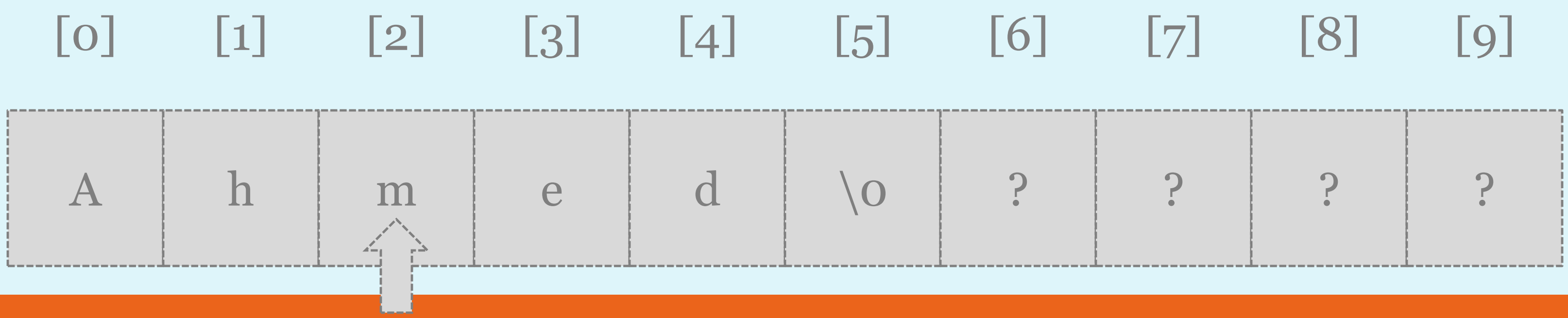
length = 2

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
long string_length (const char ntca[])
{
    long length = 0;
    while (ntca[length] != ‘\0’)
        length++;

    return length;
}
```

length = 2



Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
long string_length (const char ntca[])
{
    long length = 0;
    while (ntca[length] != '\0')
        length++;

    return length;
}
```

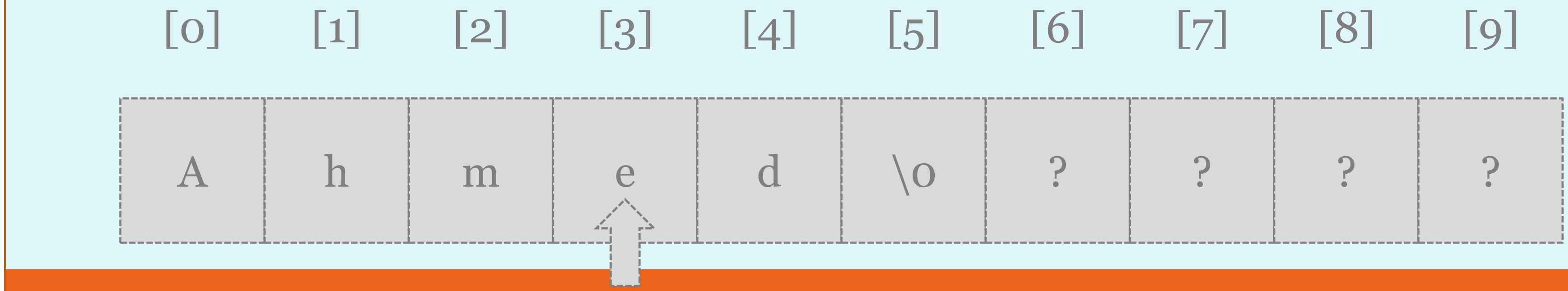
length = 3

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
long string_length (const char ntca[])
{
    long length = 0;
    while (ntca[length] != '\0')
        length++;

    return length;
}
```

length = 3



Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
long string_length (const char ntca[])
{
    long length = 0;
    while (ntca[length] != '\0')
        length++;

    return length;
}
```

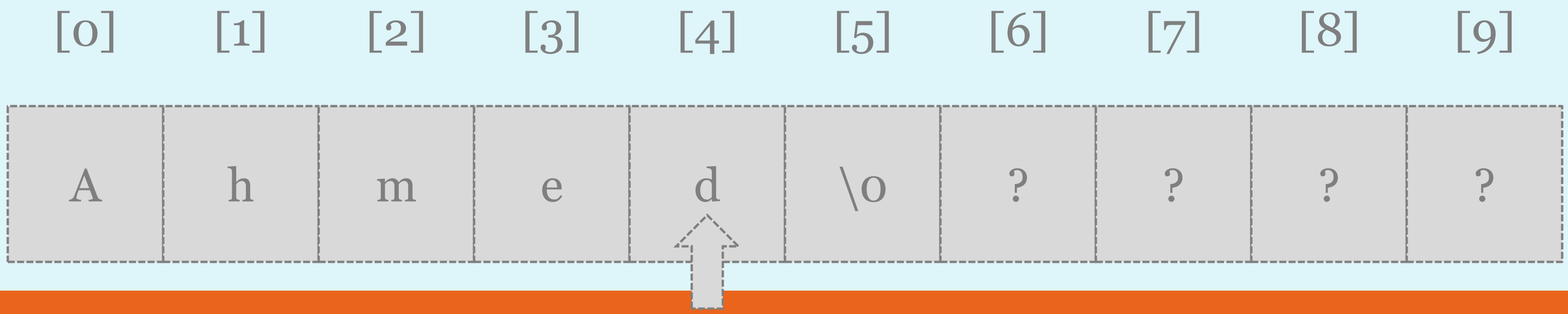
length = 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
long string_length (const char ntca[])
{
    long length = 0;
    while (ntca[length] != '\0')
        length++;

    return length;
}
```

length = 4



Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
long string_length (const char ntca[])
{
    long length = 0;
    while (ntca[length] != '\0')
        length++;

    return length;
}
```

length = 5

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
long string_length (const char ntca[])
{
    long length = 0;
    while (ntca[length] != '\0')
        length++;

    return length;
}
```

length = 5

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
long string_length (const char ntca[])
{
     long length = 0;
     while (ntca[length] != '\0')
          length++;

     return length;
}
```

length = 5

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
//ex2: char array must have null character at the end of data.
void print_reverse (const char ntca[])
{
    long length = string_length(ntca);

    for (long i = length-1; i >= 0; i--)
        cout<<ntca[i];

    return;
}
```

length =
i=
output buffer=

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
void print_reverse (const char ntca[])
{
    long length = string_length(ntca);

    for (long i = length-1; i >= 0; i--)
        cout<<ntca[i];

    return;
}
```

length = 5
i=
output buffer=

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
void print_reverse (const char ntca[])
{
    long length = string_length(ntca);

    for (long i = length-1; i >= 0; i--)
        cout<<ntca[i];

    return;
}
```

length = 5
i=4
output buffer=

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
void print_reverse (const char ntca[])
{
    long length = string_length(ntca);

    for (long i = length-1; i >= 0; i--)
        cout<<ntca[i];

    return;
}
```

length = 5
i=4
output buffer=d

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
void print_reverse (const char ntca[])
{
    long length = string_length(ntca);

    for (long i = length-1; i >= 0; i--)
        cout<<ntca[i];

    return;
}
```

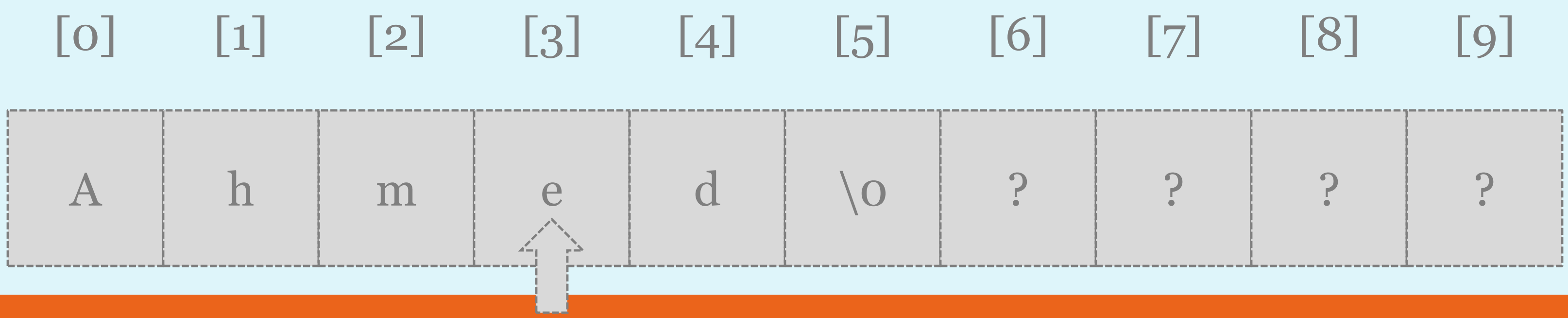
length = 5
i=3
output buffer=d

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\o	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
void print_reverse (const char ntca[])
{
    long length = string_length(ntca);

    for (long i = length-1; i >= 0; i--)
        cout<<ntca[i];

    return;
}
```

length = 5
i=3
output buffer=de



Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
void print_reverse (const char ntca[])
{
    long length = string_length(ntca);

    for (long i = length-1; i >= 0; i--)
        cout<<ntca[i];

    return;
}
```

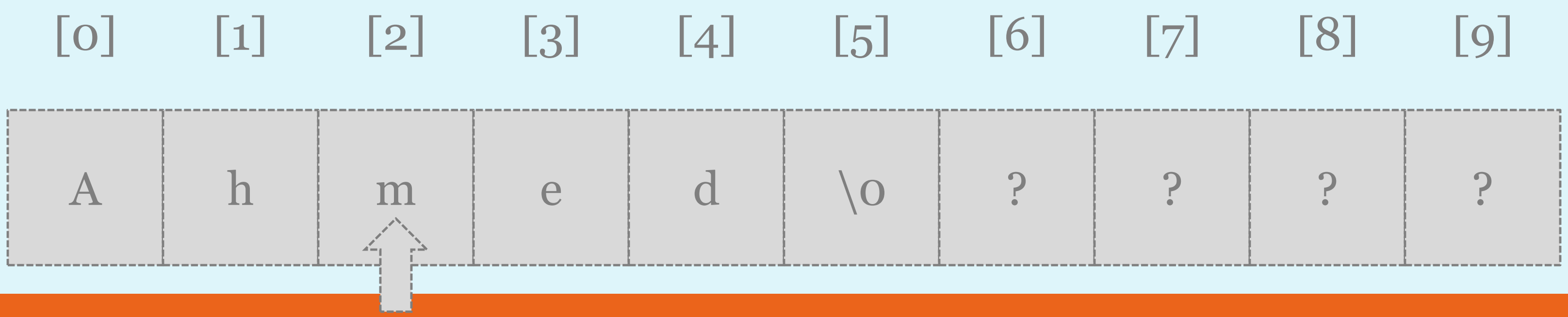
length = 5
i=2
output buffer=de

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
void print_reverse (const char ntca[])
{
    long length = string_length(ntca);

    for (long i = length-1; i >= 0; i--)
        cout<<ntca[i];

    return;
}
```

length = 5
i=2
output buffer=dem



Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
void print_reverse (const char ntca[])
{
    long length = string_length(ntca);

    for (long i = length-1; i >= 0; i--)
        cout<<ntca[i];

    return;
}
```

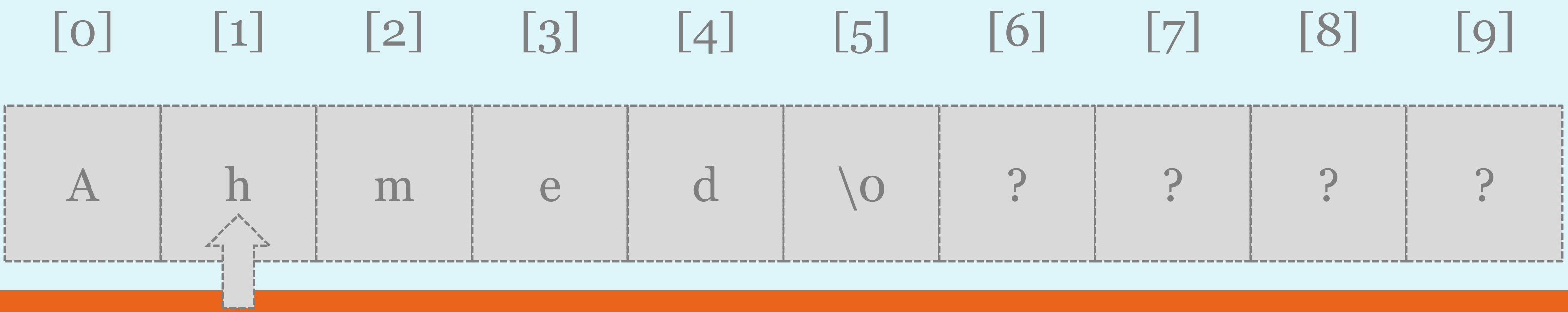
length = 5
i=1
output buffer=dem

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
void print_reverse (const char ntca[])
{
    long length = string_length(ntca);

    for (long i = length-1; i >= 0; i--)
        cout<<ntca[i];

    return;
}
```

length = 5
i=1
output buffer=demh



Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
void print_reverse (const char ntca[])
{
    long length = string_length(ntca);

    for (long i = length-1; i >= 0; i--)
        cout<<ntca[i];

    return;
}
```

length = 5
i=0
output buffer=demh

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
void print_reverse (const char ntca[])
{
    long length = string_length(ntca);

    for (long i = length-1; i >= 0; i--)
        cout<<ntca[i];

    return;
}
```

length = 5
i=0
output buffer=demhA

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Using the Null-Terminating Character

```
// Pre: char array must have null character at the end of data.
void print_reverse (const char ntca[])
{
    long length = string_length(ntca);

    for (long i = length-1; i >= 0; i--)
        cout<<ntca[i];

    return;
}
```

length = 5
i=-1
output buffer=demhA

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	h	m	e	d	\0	?	?	?	?

Two-Dimensional Arrays

- C++ allows multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array **twod** of size 10,20 you would write.

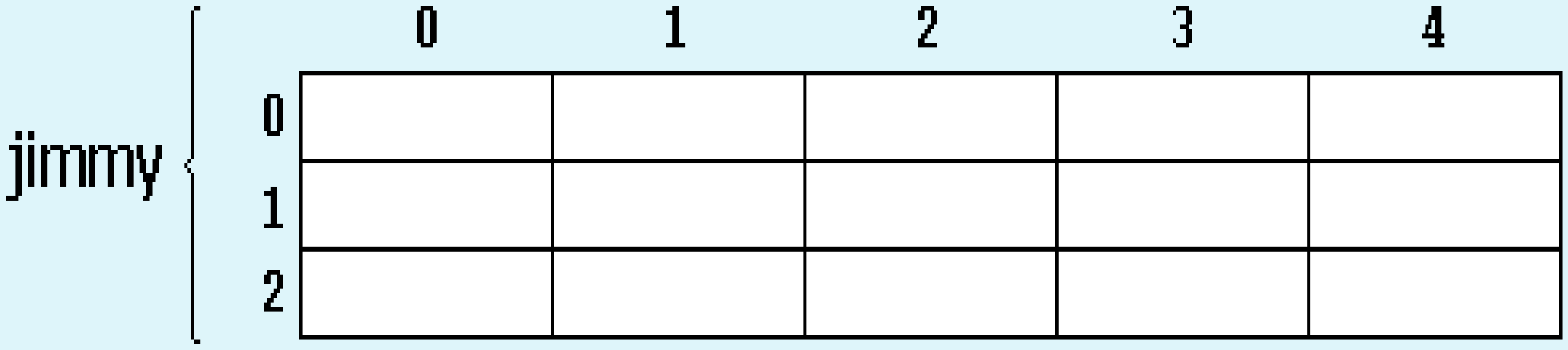
  int twod[10][20];

- Pay careful attention to the declaration. Unlike some other computer languages, which use commas to separate the array dimensions, C++ places each dimension in its own set of brackets.
- Similarly, to access point 3,5 of array **twod**, you would use.

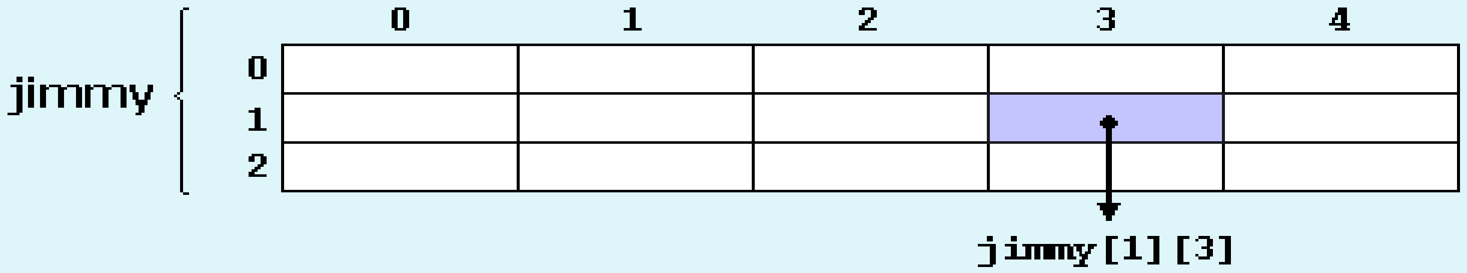
  twod [3][5].

- jimmy represents a bidimensional array of 3 per 5 elements of type int. The C++ syntax for this is:

  int jimmy [3][5];



- For example, the way to reference the second element vertically and fourth horizontally in an expression would be:
- jimmy[1][3]



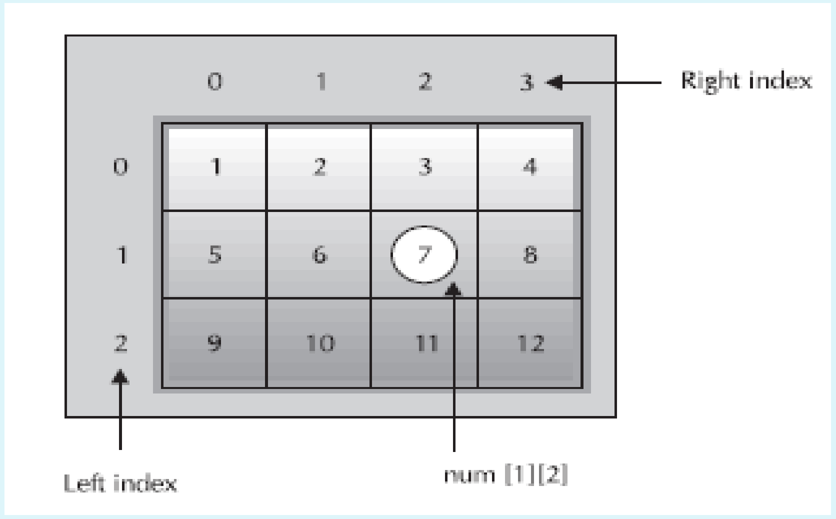
(remember that array indices always begin with zero).

Example, a two-dimensional array is loaded with the numbers 1 through 12.

```
#include <iostream>
using namespace std;
int main()
{
int t,i, num[3][4];
for(t=0; t<3; ++t) {
for(i=0; i<4; ++i) {
num[t][i] = (t*4)+i+1;
cout << num[t][i] << ' ';
}
cout << '\n';
}
system("pause");
return 0;
}
```

- In this example, **num[0][0]** will have the value 1, **num[0][1]** the value 2,
- **num[0][2]** the value 3, and so on.
- The value of **num[2][3]** will be 12.

num[t][i] = (t*4)+i+1;



Multidimensional Arrays

- C++ allows arrays with more than two dimensions. Here is the general form of a multidimensional array declaration:

  type name[*size1*][*size2*]...[*sizeN*];

- For example, the following declaration creates a 4 × 10 × 3 integer array:

  ```
  int multidim[4][10][3];
  ```

- Arrays of more than three dimensions are not often used, due to the amount of memory required to hold them.

Example

The following program uses the **sqrs** array to find the square of a number entered by the user. It first looks up the number in the array and then prints the corresponding square.

```
#include <iostream>
using namespace std;
int sqrs[10][2] = {
{1, 1},
{2, 4},
{3, 9},
{4, 16},
{5, 25},
{6, 36},
{7, 49},
{8, 64},
{9, 81},
{10, 100}
};
```

continue

```
int main()
{
int i, j;
cout << "Enter a number between 1 and 10: ";
cin >> i;
// look up i
for(j=0; j<10; j++)
if(sqrs[j][0]==i) break;
// display square
cout << "The square of " << i << " is ";
cout << sqrs[j][1];
system("pause");
return 0;
}
```

Stack Data Structures

Stacks

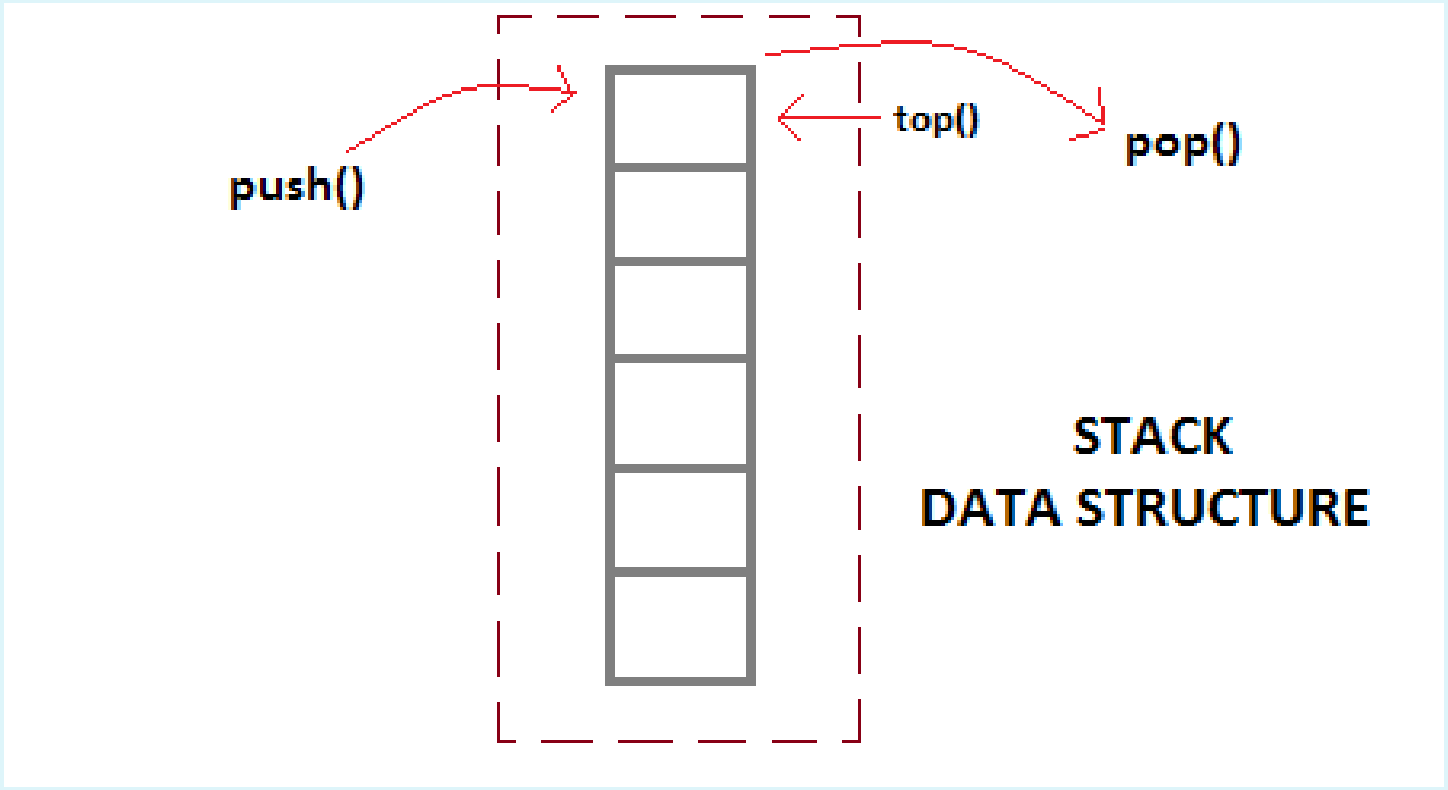
- Stack is an abstract data type with a bounded (predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.

- **Stack Syntax**
- To create a stack, we must include the <stack> header file in our code. We then use this syntax to define the std::stack:
- **Member Types**

Here are stack member types:

- value_type- The first template parameter, T. It denotes the element types.
- container_type- The second template parameter, Container. It denotes the underlying container type.
- size_type- Unsigned integral type.

Stack data structure



Basic features of Stack

- Stack is an ordered list of similar data type.
- Stack is a **LIFO** structure. (Last in First out).
- **push()** function is used to insert new elements into the Stack and **pop()** is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called **Top**.
- Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

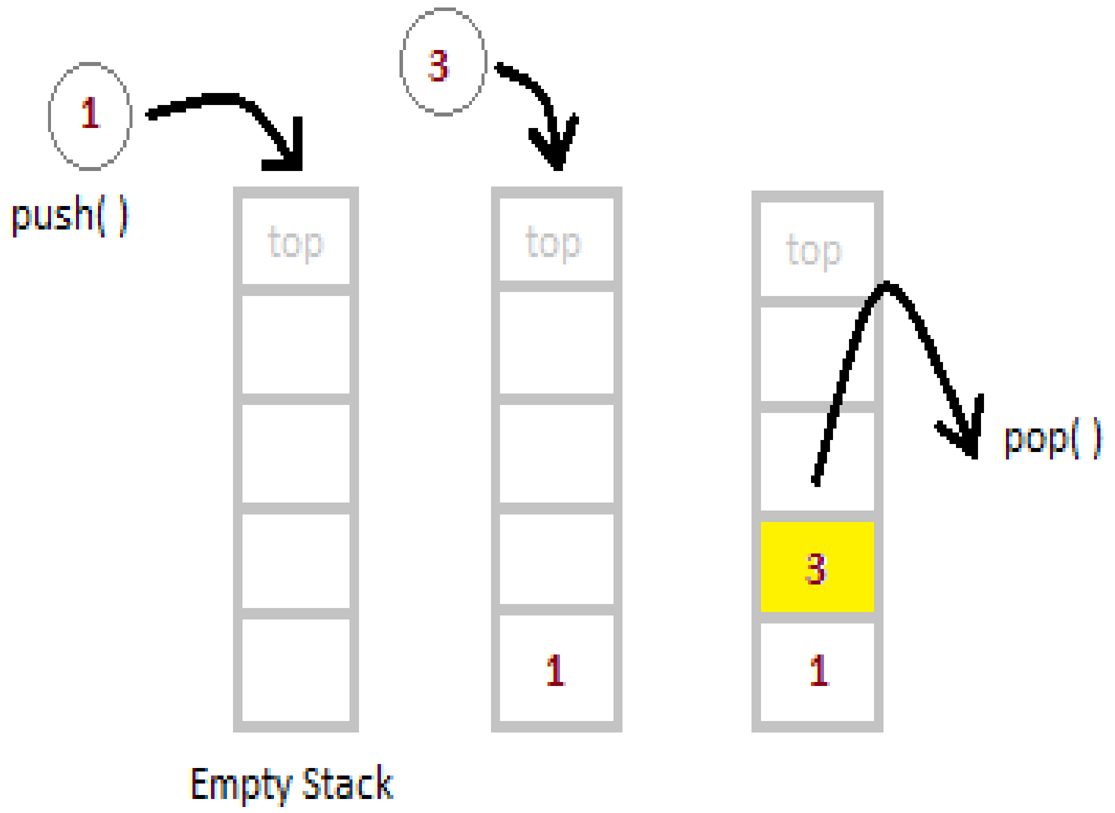
Applications of Stack

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- There are other uses also like : **Parsing**, **Expression Conversion**(Infix to Postfix, Postfix to Prefix etc) and many more.

Implementation of Stack

- Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.

STACK - LIFO Structure

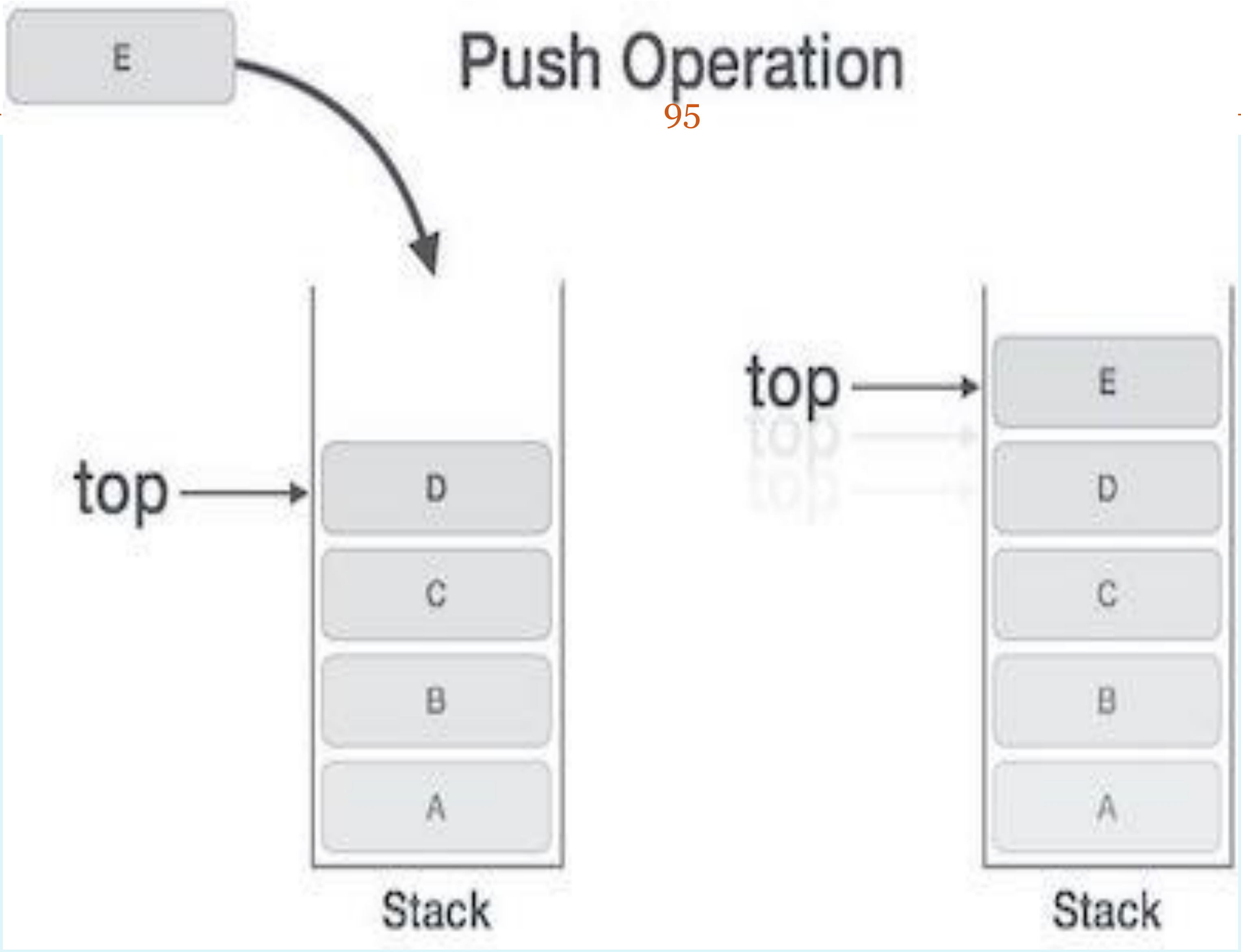


In a Stack, all operations take place at the "**top**" of the stack. The "**push**" operation adds an item to the top of the Stack.

The "**pop**" operation removes the item on top of the stack.

Push Operation

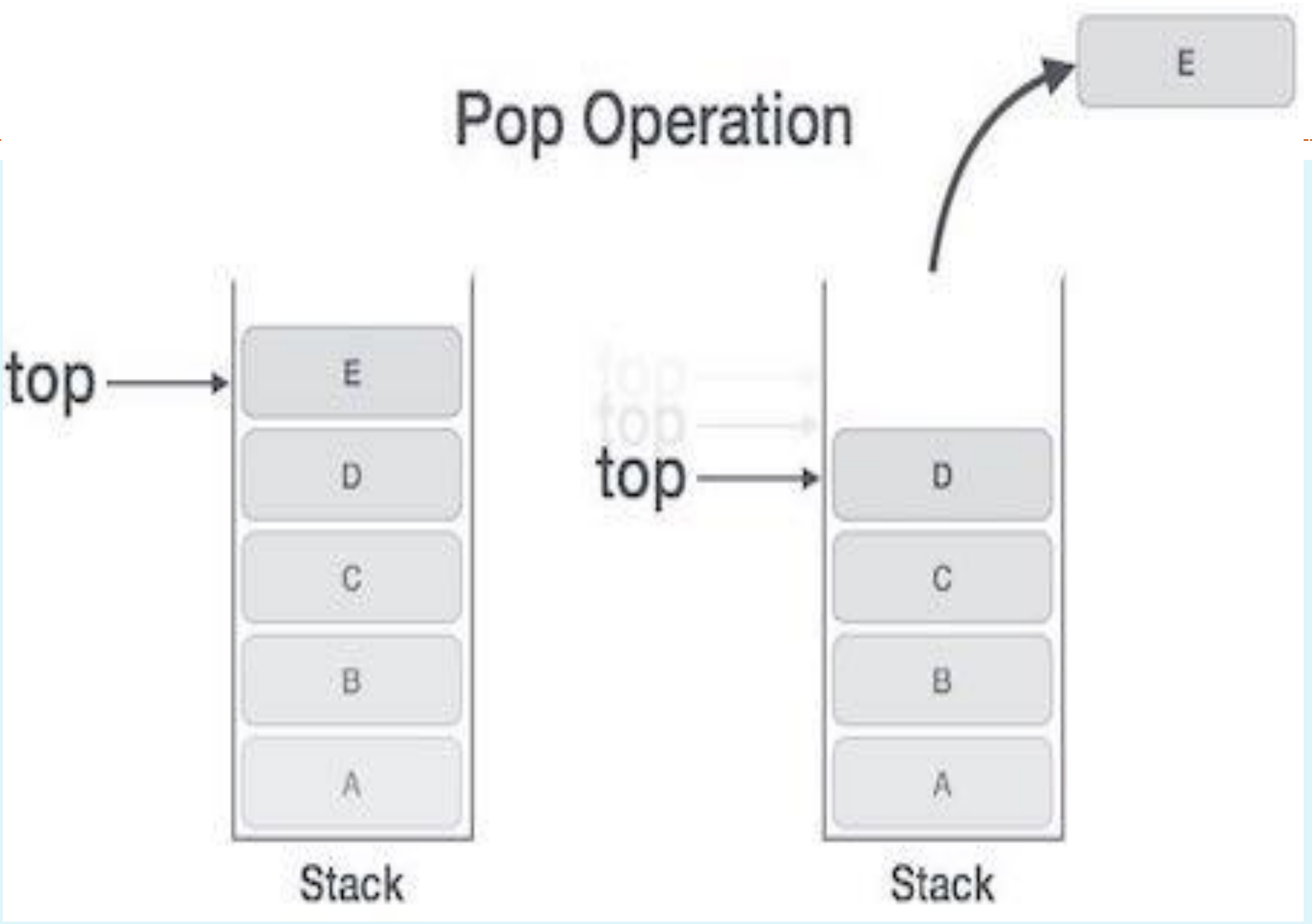
- The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –
  - **Step 1** – Checks if the stack is full.
  - **Step 2** – If the stack is full, produces an error and exit.
  - **Step 3** – If the stack is not full, increments **top** to point next empty space.
  - **Step 4** – Adds data element to the stack location, where top is pointing.
  - **Step 5** – Returns success.


E
Push Operation
top
D
C
B
A
Stack
top
E
D
C
B
A
Stack

Pop Operation

- Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

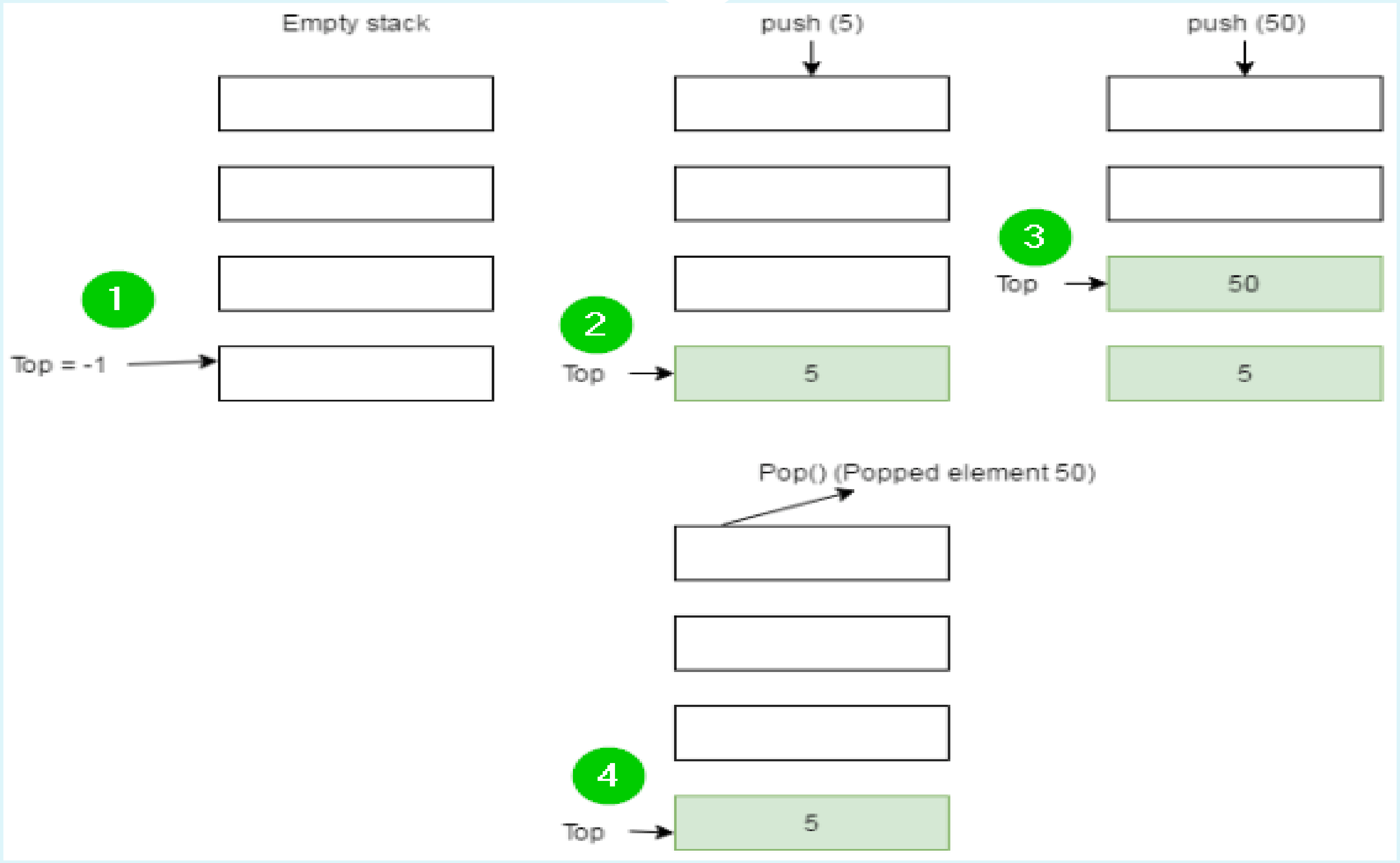
- A Pop operation may involve the following steps –
  - **Step 1** – Checks if the stack is empty.
  - **Step 2** – If the stack is empty, produces an error and exit.
  - **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
  - **Step 4** – Decreases the value of top by 1.
  - **Step 5** – Returns success.


Pop Operation
E
top
E
D
C
B
A
Stack
top
top
top
D
C
B
A
Stack

Operations in Stack

- A C++ stack supports the following basic operations:
- push – It adds/pushes an item into the stack.
- pop – It removes/pops an item from the stack.
- peek – Returns the top item of the stack without removing it.
- isFull – Checks whether a stack is full.
- isEmpty – Checks whether a stack is empty.
- LIFO Refers to the last in, first out behavior of the stack
- FILO Equivalent to LIFO

Position of Top	**Status of Stack**
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack


Empty stack
push (5)
push (50)
1
Top = -1
2
Top
5
3
Top
50
5
Pop() (Popped element 50)
4
Top
5

Stack Implementation

- **Step 1)** We initially have an empty stack. The top of an empty stack is set to -1.
- **Step 2)** Next, we have pushed the element 5 into the stack. The top of the stack will points to the element 5.
- **Step 3)** Next, we have pushed the element 50 into the stack. The top of the stack shifts and points to the element 50.
- **Step 4)** We have then performed a pop operation, removing the top element from the stack. The element 50 is popped from the stack. The top of the stack now points to the element 5.

Example

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
stack<int> st;
st.push(10);
st.push(20);
st.push(30);
st.push(40);
st.pop();
st.pop();

while (!st.empty()) {
cout << ' ' << st.top();
st.pop();
}
system("pause");
return 0;
}
```

empty(), size(), top()

Stacks have inbuilt functions that you can use to play around with the stack and its values. These include:

- empty() checks whether a stack is empty or not.
- size() returns the size of stack, that is, number of elements in a stack.
- top() accesses stack element at the top.

Example

```
#include <iostream>
#include <stack>
using namespace std;
void createStack(stack <int>
mystack)
{
stack <int> ms = mystack;
while (!ms.empty())
        {
        cout << '\t' <<
ms.top();
                ms.pop();
        }
        cout << '\n';
}
```

```
int main()
{
stack <int> st;
st.push(32);
st.push(21);
st.push(39);
st.push(89);
st.push(25);

cout << "The stack st is: ";
createStack(st);
cout << "\n st.size() : " << st.size();
cout << "\n st.top() : " << st.top();
cout << "\n st.pop() : ";
        st.pop();
        createStack(st);
        return 0;
}
```

emplace() and swap()

- These are other inbuilt stack functions:
- emplace()- constructs then inserts new element to top of stack.
- swap()- exchanges stack contents with another stack's contents.

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
stack<int> st1;
stack<int> st2;

st1.emplace(12);
st1.emplace(19);

st2.emplace(20);
st2.emplace(23);

st1.swap(st2);
cout << "st1 = ";
while (!st1.empty()) {
cout << st1.top() << " ";
st1.pop();
}

cout << endl << "st2 = ";
while (!st2.empty()) {
cout << st2.top() << " ";
st2.pop();
}
system("pause");
return 0;
}
```

Infix expression

- An infix expression is an expression in which operators (+, -, *, /) are written between the two operands. For example, consider the following expressions:
- A + B
- A + B - C
- (A + B) + (C - D)

Here we have written '+' operator between the operands A and B, and the - operator in between the C and D operand.

Postfix Expression

- The postfix operator also contains operator and operands. In the postfix expression, the operator is written after the operand. It is also known as **Reverse Polish Notation**. For example, consider the following expressions:
- A B +
- A B + C -
- A B C * +
- A B + C * D -

Algorithm to Convert Infix to Postfix Expression Using Stack

- Initialize the Stack.
- Scan the operator from left to right in the infix expression.
- If the leftmost character is an operand, set it as the current output to the Postfix string.
- And if the scanned character is the operator and the Stack is empty or contains the '(', ')' symbol, push the operator into the Stack.
- If the scanned operator has higher precedence than the existing **precedence** operator in the Stack or if the Stack is empty, put it on the Stack.
- If the scanned operator has lower precedence than the existing operator in the Stack, pop all the Stack operators. After that, push the scanned operator into the Stack.

- If the scanned character is a left bracket '(', push it into the Stack.
- If we encountered right bracket ')', pop the Stack and print all output string character until '(' is encountered and discard both the bracket.
- Repeat all steps from 2 to 8 until the infix expression is scanned.
- Print the Stack output.
- Pop and output all characters, including the operator, from the Stack until it is not empty.

- Let's translate an infix expression into postfix expression in the stack:
- Here, we have infix expression **((A * (B + D)/E) – (F * (G + H / K)))** to convert into its equivalent postfix expression:
-

((A * (B + D)/E) – (F * (G + H / K)))

Label No.	Symbol Scanned	Stack	Expression
1	(	(	
2	(	((	
3	A	((	A
4	*	((*	A
5	(	((*(	A
6	B	((*(	AB
7	+	((*(+	AB
8	D	((*(+	ABD
9	)	((*	ABD+
10	/	((*/	ABD+
11	E	((*/	ABD+E
12	)	(	ABD+E/*

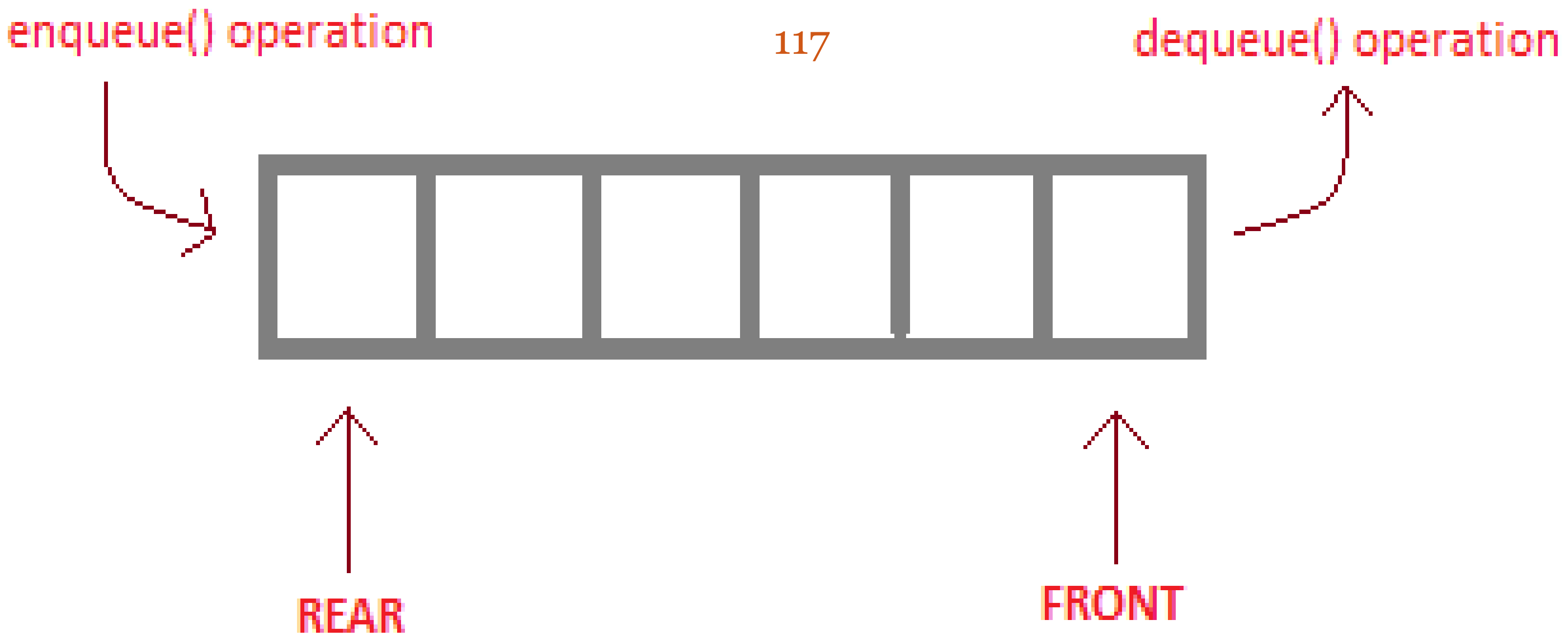
((A * (B + D)/E) – (F * (G + H / K)))

13	-	(-	ABD+E/*
14	(	(-(	ABD+E/*
15	F	(-(	ABD+E/*F
16	*	(-(*	ABD+E/*F
17	(	(-(*(	ABD+E/*F
18	G	(-(*(	ABD+E/*FG
19	+	(-(*(+	ABD+E/*FG
20	H	(-(*(+	ABD+E/*FGH
21	/	(-(*(+/	ABD+E/*FGH
22	K	(-(*(+/	ABD+E/*FGHK
23	)	(-(*	ABD+E/*FGHK/+
24	)	(-	ABD+E/*FGHK/+*
25	)		ABD+E/*FGHK/+*-

QUEUE DATA STRUCTURES

Queue Data Structures

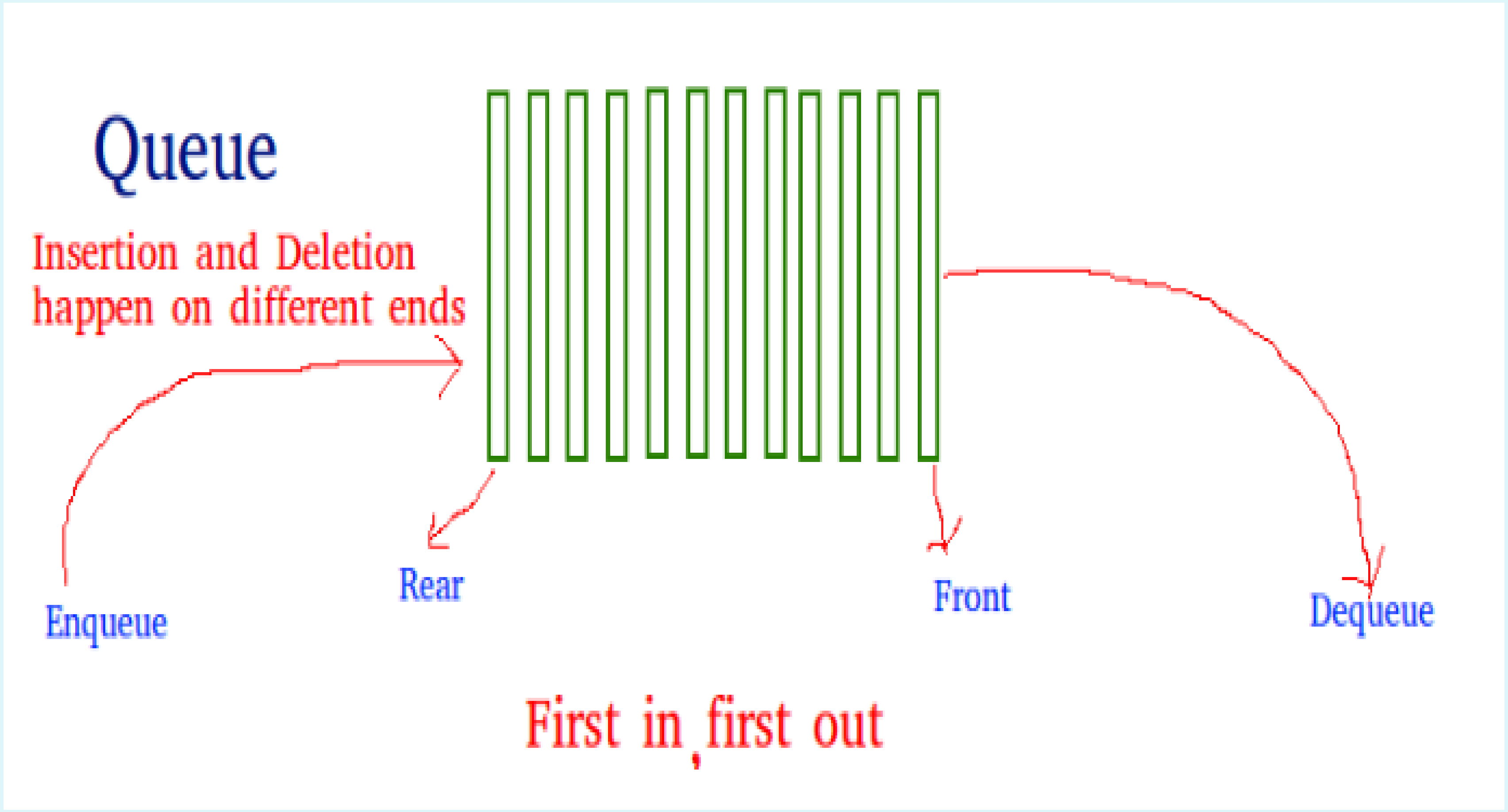
- Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called **REAR**(also called tail), and the deletion of existing element takes place from the other end called as **FRONT**(also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first.
- The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

<u>QUEUE DATA STRUCTURE</u>


Queue
Insertion and Deletion happen on different ends
Enqueue
Rear
Front
Dequeue
First in, first out

Basic features of Queue

- Like Stack, Queue is also an ordered list of elements of similar data types.
- Queue is a FIFO(First in First Out) structure.
- Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
- **peek( )** function is oftenly used to return the value of first element without dequeuing it.

Operations on Queue:

- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.
- **peek()** – Gets the element at the front of the queue without removing it.
- **isfull()** – Checks if the queue is full.
- **isempty()** – Checks if the queue is empty.
- **Front:** Get the front item from queue.
  Rear: Get the last item from queue.

- In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

peek()

- This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows –
- **Algorithm**

```
begin procedure peek
  return queue[front]
end procedure
```

Implementation of peek() function in C++ programming language –

- **Example**

```
int peek() {
return queue[front];
}
```

isfull()

- As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full.

Algorithm

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C++ programming language

Example

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty()

Algorithm

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

If the value of **front** is less than MIN, it tells that the queue is not yet initialized, hence empty.

Here's the C++ programming code –

Example

```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

Applications of Queue

- Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :

- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

- In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.

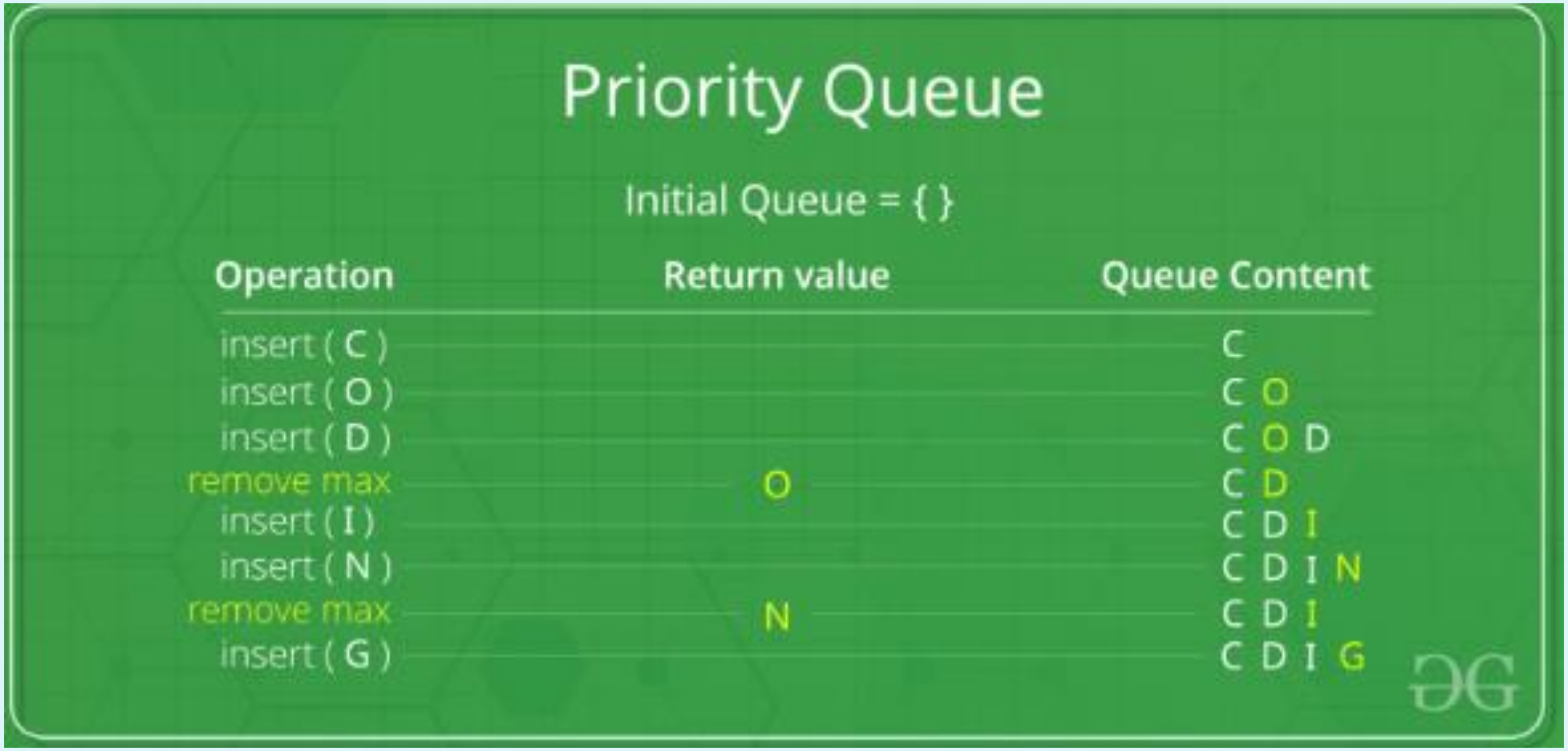
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

Priority Queue

Priority Queue is an extension of queue with following properties.

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

- In the below priority queue, element with maximum ASCII value will have the highest priority.



Priority Queue

Initial Queue = { }

Operation	Return value	Queue Content
insert (C)		C
insert (O)		C O
insert (D)		C O D
remove max	O	C D
insert (I)		C D I
insert (N)		C D I N
remove max	N	C D I
insert (G)		C D I G

A typical priority queue supports following operations.

- **insert(item, priority):** Inserts an item with given priority.
  getHighestPriority(): Returns the highest priority item.
  deleteHighestPriority(): Removes the highest priority item.

How to implement priority queue?

- ***Using Array:*** A simple implementation is to use array of following structure.

```
struct item {
 int item;
int priority;
}
```

- insert() operation can be implemented by adding an item at end of array
- getHighestPriority() operation can be implemented by linearly searching the highest priority item in array
- deleteHighestPriority() operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.

Applications of Priority Queue:

- 1) CPU Scheduling
- 2) Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
- 3) All queue applications where priority is involved.

Implementation of Queue

- Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.
- Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.

Array implementation Of Queue

- structure can be implemented using one dimensional array. But, queue implemented using array can store only fixed number of data values. The implementation of queue data structure using array is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables **'front'** and **'rear'**. Initially both **'front'** and **'rear'** are set to -1.
- Whenever, we want to insert a new value into the queue, increment **'rear'** value by one and then insert at that position. Whenever we want to delete a value from the queue, then increment 'front' value by one and then display the value at **'front'** position as deleted element.

Pros and cons of Array Implementation:

Pros of Array Implementation

- Easy to implement.

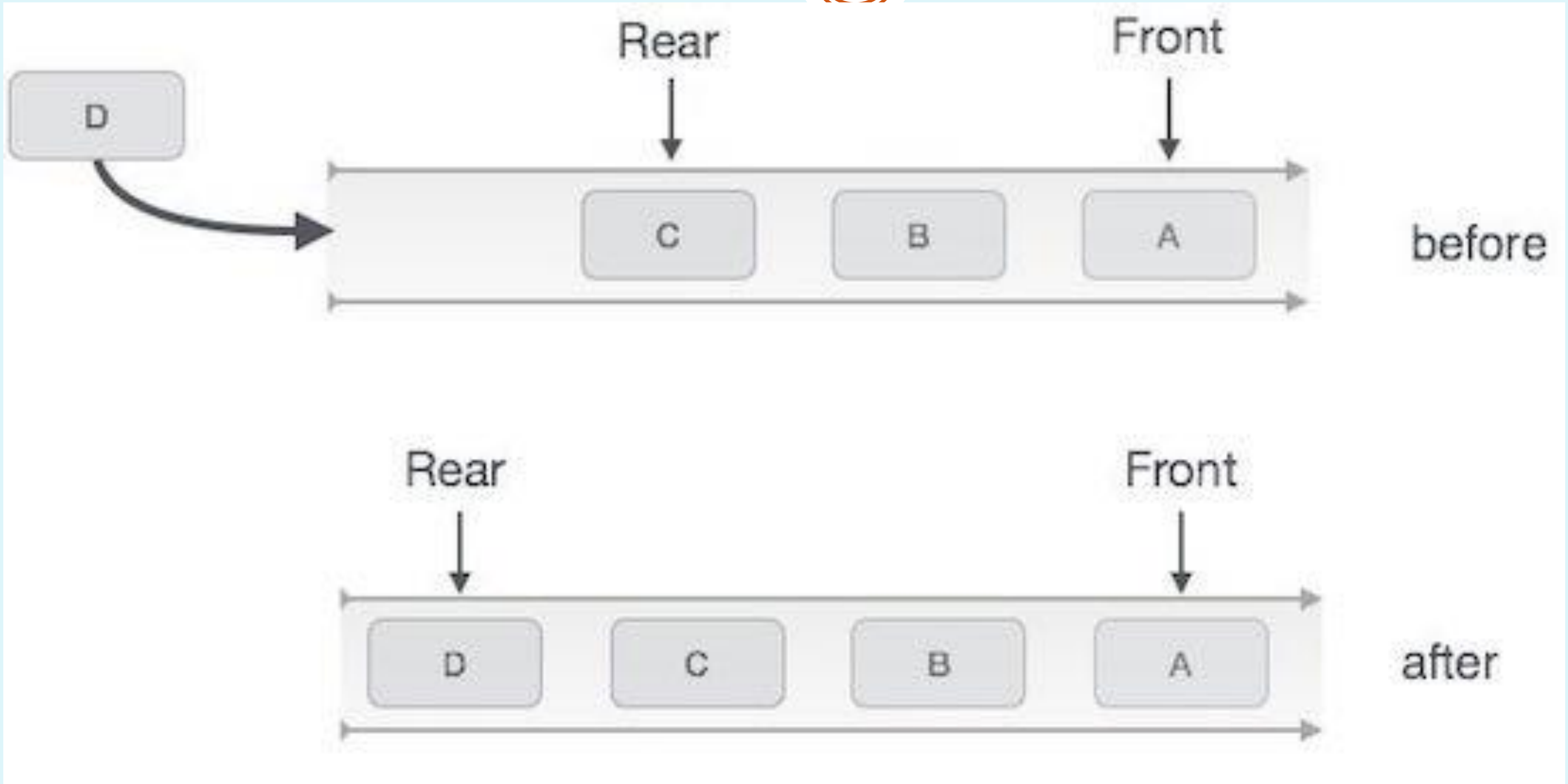
Cons of Array Implementation:

- Static Data Structure, fixed size.
- If the queue has a large number of enqueue and dequeue operations, at some point (in case of linear increment of front and rear indexes) we may not be able to insert elements in the queue even if the queue is empty (this problem is avoided by using circular queue).

enQueue(value) - Inserting value into the queue

- In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

- **Step 1** – Check if the queue is full.
- **Step 2** – If the queue is full, produce overflow error and exit.
- **Step 3** – If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** – Add data element to the queue location, where the rear is pointing.
- **Step 5** – return success



Queue Enqueue

- Algorithm for enqueue operation

```
procedure enqueue(data)

   if queue is full
      return overflow
   endif

   rear ← rear + 1
   queue[rear] ← data
   return true

end procedure
```

- Implementation of enqueue() in C++ programming language –

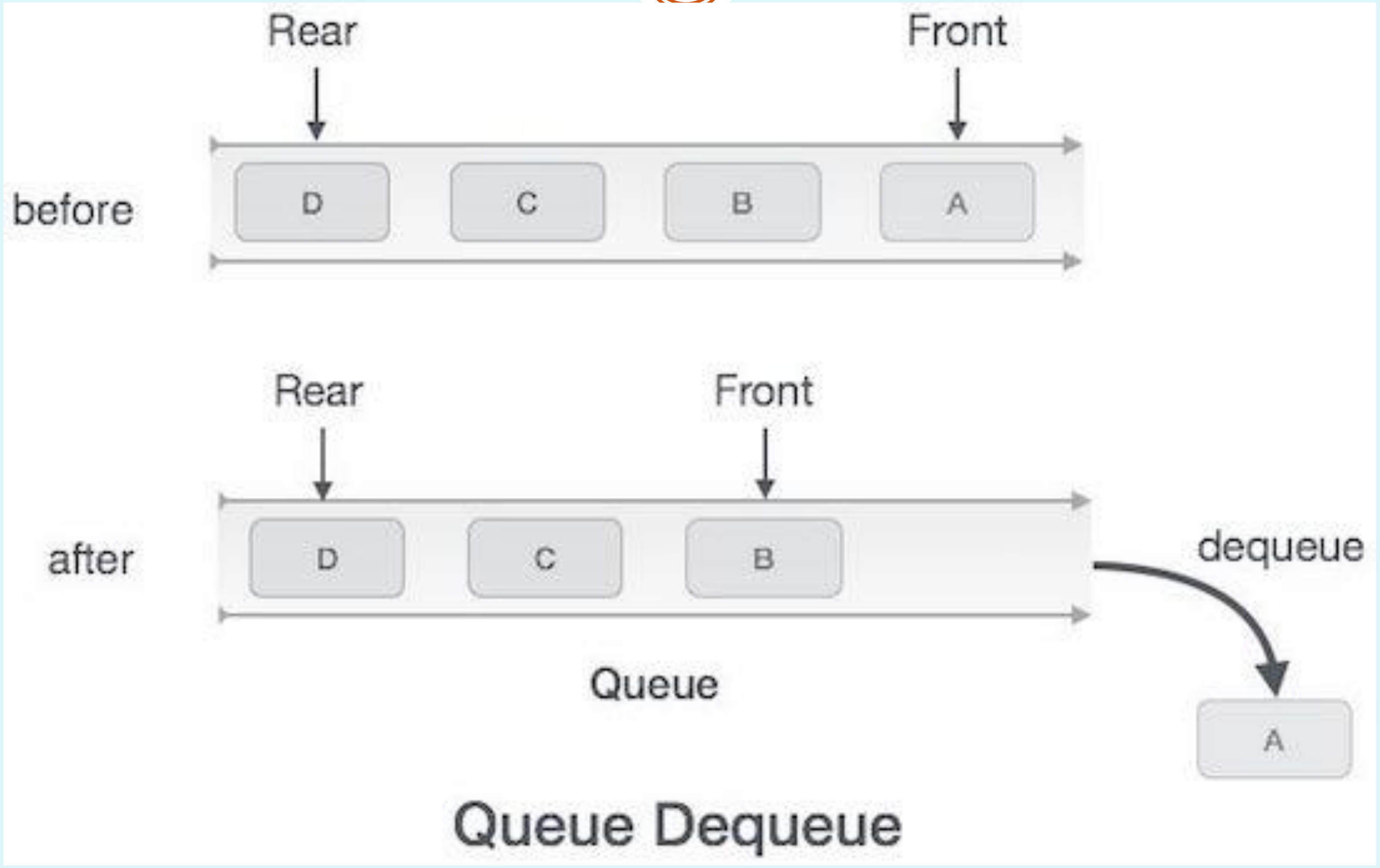
Example

```
int enqueue(int data)
   if(isfull())
      return 0;

   rear = rear + 1;
   queue[rear] = data;

   return 1;
end procedure
```

deQueue() - Deleting a value from the Queue

- In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue

   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1
   return true

end procedure
```

Implementation of dequeue() in C++ programming language –

Example

```
int dequeue() {
  if(isempty())
    return 0;

  int data = queue[front];
  front = front + 1;

  return data;
}
```

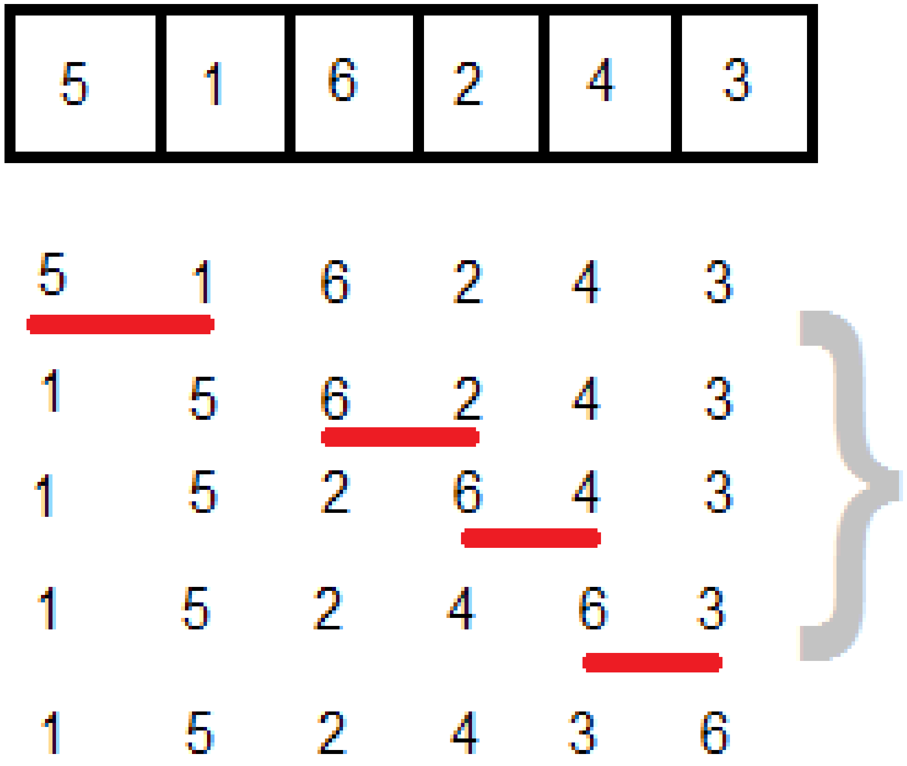
Types of Sorting Techniques

Types of Sorting Techniques

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Merge Sort
- Heap Sort

Bubble Sorting

- **Bubble Sort** is an algorithm which is used to sort **N** elements that are given in a memory for eg: an Array with **N** number of elements. Bubble Sort compares all the element one by one and sort them based on their values.
- It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.
- Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.



Lets take this Array.

Here we can see the Array after the first iteration.

Similarly, after other consecutive iterations, this array will get sorted.

5	1	12	-5	16	unsorted
5	1	12	-5	16	5 > 1, swap
1	5	12	-5	16	5 < 12, ok
1	5	12	-5	16	12 > -5, swap
1	5	-5	12	16	12 < 16, ok
1	5	-5	12	16	1 < 5, ok
1	5	-5	12	16	5 > -5, swap
1	-5	5	12	16	5 < 12, ok
1	-5	5	12	16	1 > -5, swap
-5	1	5	12	16	1 < 5, ok
-5	1	5	12	16	-5 < 1, ok
-5	1	5	12	16	sorted

Bubble Sort **ALGORITHM:**

- **Bubble_Sort (A [ ] , N)**
- Step 1: Start
- Step 2: Take an array of n elements
- Step 3: for i=0,............n-2
- Step 4: for j=i+1,.......n-1
- Step 5: if arr[j]>arr[j+1] then
- Interchange arr[j] and arr[j+1]
- End of if
- Step 6: Print the sorted array arr
- Step 7:Stop

- Note: Above is the algorithm, to sort an array using Bubble Sort. Although the above logic will sort and unsorted array, still the above algorithm isn't efficient and can be enhanced further. Because as per the above logic, the for loop will keep going for six iterations even if the array gets sorted after the second iteration.

- Hence we can insert a flag and can keep checking whether swapping of elements is taking place or not. If no swapping is taking place that means the array is sorted and we can jump out of the for loop.

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, temp;
for(i=0; i<6; i++)
{
 int flag = 0;       //taking a flag variable
 for(j=0; j<6-i-1; j++)
 {
  if( a[j] > a[j+1])
  {
   temp = a[j];
   a[j] = a[j+1];
   a[j+1] = temp;
   flag = 1;       //setting flag as 1, if swapping occurs
  }
 }
 if(!flag)         //breaking out of for loop if no swapping takes place
 {
  break;
 }
}
```

In this code, if in a complete single cycle of j iteration(inner for loop), no swapping takes place, and flag remains 0, then we will break out of the for loops, because the array has already been sorted.

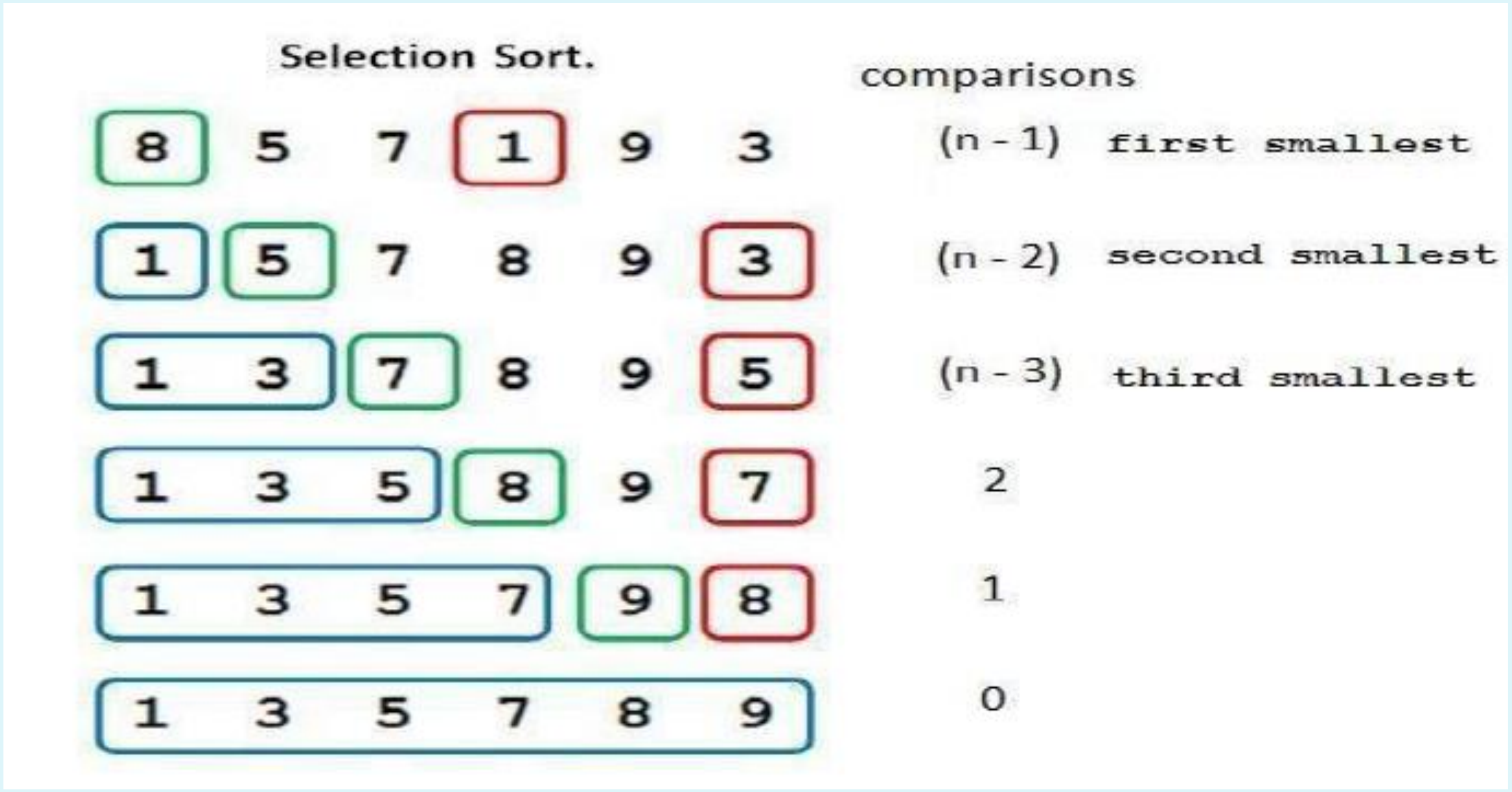
Selection Sorting

- Selection sorting is conceptually the most simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

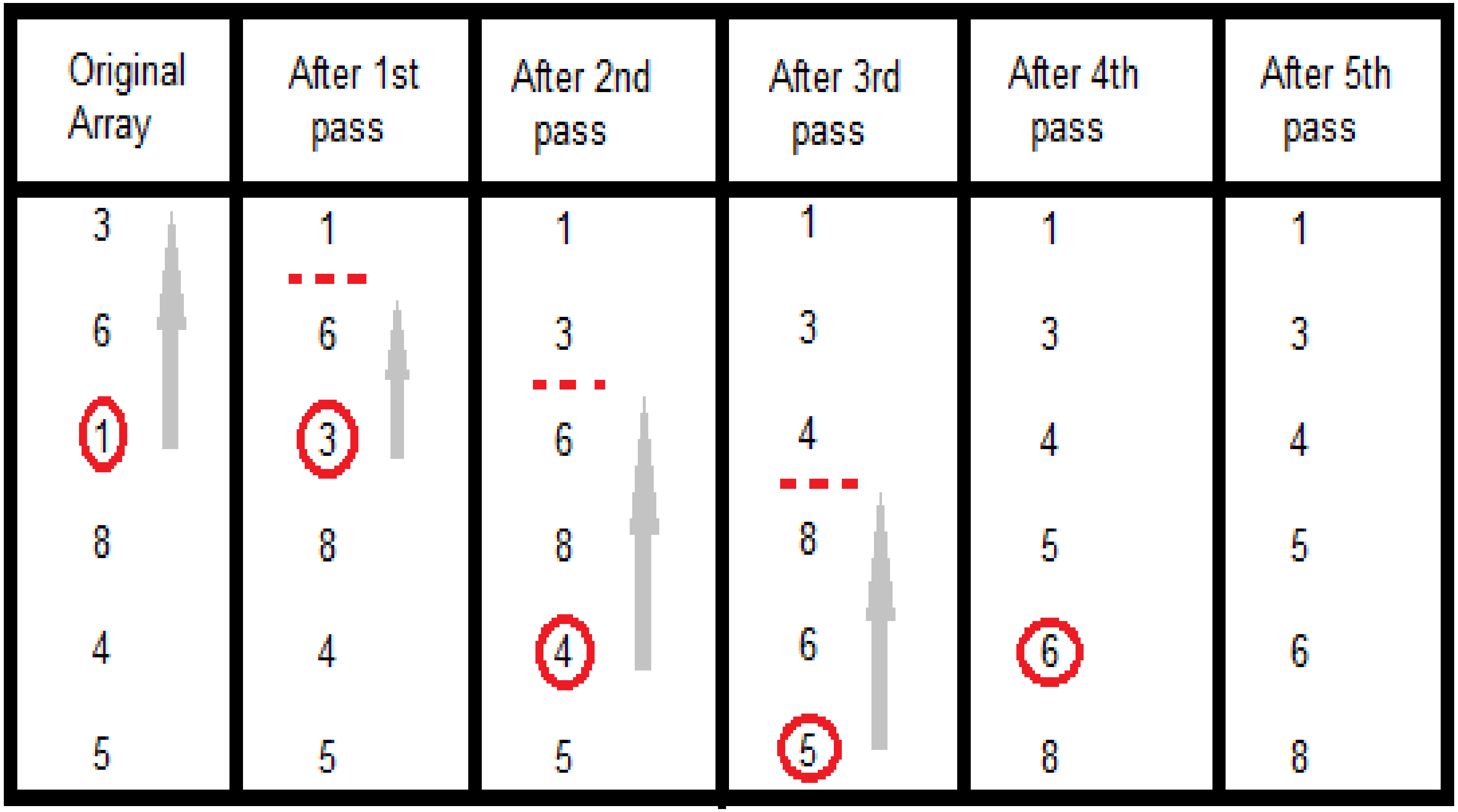
Selection sorting

- Selection sort is quite a straightforward sorting technique as the technique only involves finding the smallest element in every pass and placing it in the correct position.
- Selection sort works efficiently when the list to be sorted is of small size but its performance is affected badly as the list to be sorted grows in size.
- Hence we can say that selection sort is not advisable for larger lists of data.

How Selection Sorting Works



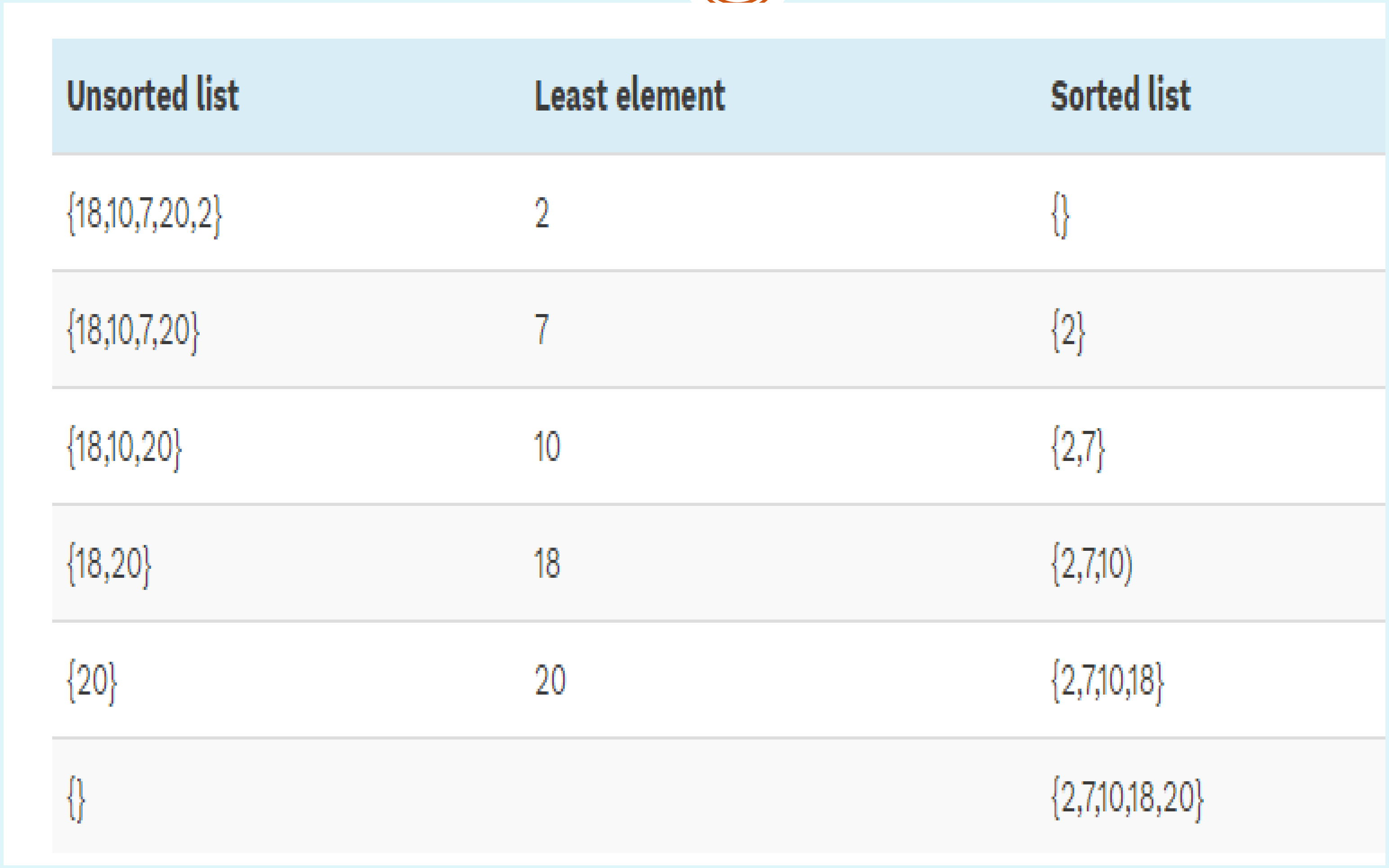
How Selection Sorting Works



Original Array	After 1st pass	After 2nd pass	After 3rd pass	After 4th pass	After 5th pass
3	1	1	1	1	1
6	6	3	3	3	3
1	3	6	4	4	4
8	8	8	8	5	5
4	4	4	6	6	6
5	5	5	5	8	8

How Selection Sorting Works

- In the first pass, the smallest element found is 1, so it is placed at the first position, then leaving first element, smallest element is searched from the rest of the elements, 3 is the smallest, so it is then placed at the second position. Then we leave 1 and 3, from the rest of the elements, we search for the smallest and put it at third position and keep doing this, until array is sorted.



Unsorted list	Least element	Sorted list
{18,10,7,20,2}	2	{}
{18,10,7,20}	7	{2}
{18,10,20}	10	{2,7}
{18,20}	18	{2,7,10)
{20}	20	{2,7,10,18}
{}		{2,7,10,18,20}

- From the illustration, we see that with every pass the next smallest element is put in its correct position in the sorted array. From the above illustration, we see that in order to sort an array of 5 elements, four passes were required. This means in general, to sort an array of N elements, we need N-1 passes in total.

Generally How Selection Sorting Works

- The first item is compared with the remaining n-1 items, and whichever of all is lowest, is put in the first position.
- Then the second item from the list is taken and compared with the remaining (n-2) items, if an item with a value less than that of the second item is found on the (n-2) items, it is swapped (Interchanged) with the second item of the list and so on.

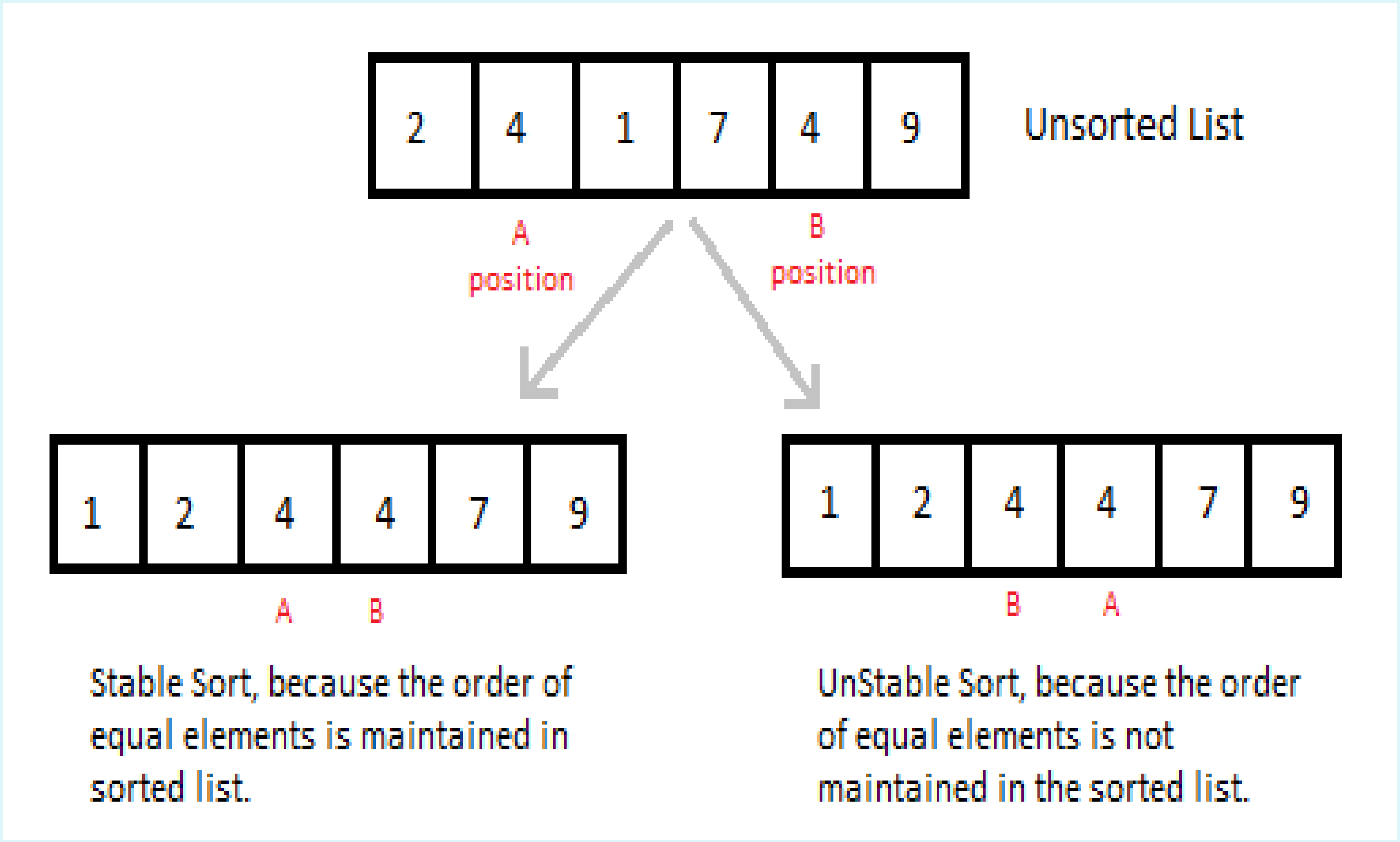
- Examples is included in practical worksheet

Insertion sort

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less, like Bubble Sorting, insertion sort also requires a single additional memory space.
6. It is **Stable**, as it does not change the relative order of elements with equal keys

Stable vs unstable sort



Stable Sort, because the order of equal elements is maintained in sorted list.

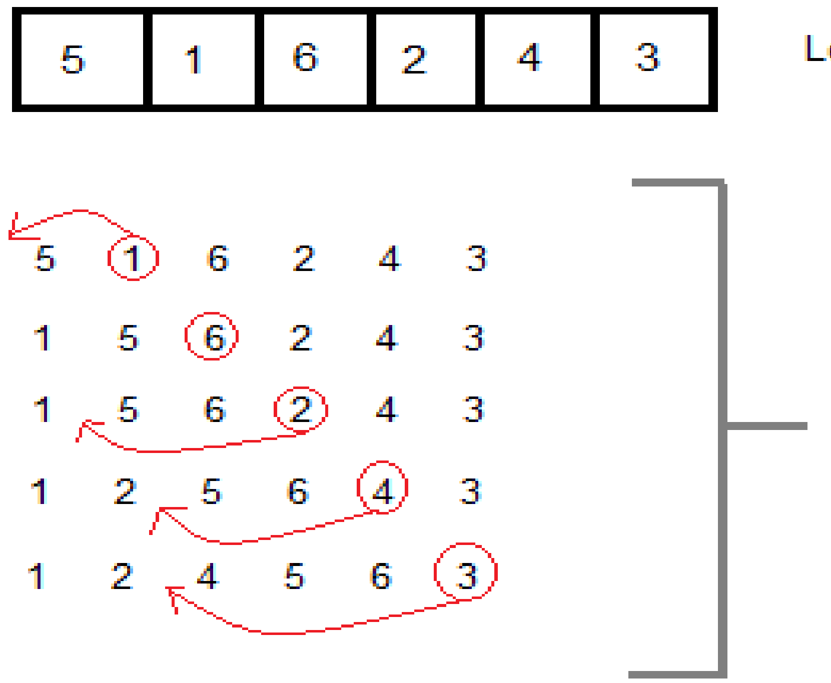
UnStable Sort, because the order of equal elements is not maintained in the sorted list.

- **Insertion sort:** It iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

ALGORITHM:

```
Step 1: start
Step 2: for i ← 1 to length(A)
Step 3: j ← i
Step 4: while j > 0 and A[j-1] > A[j]
Step 5: swap A[j] and A[j-1]
Step 6: j ← j - 1
Step 7: end while
Step 8: end for
Step9: stop
```

How insertion sort works



Lets take this Array.

As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place

5 has nothing before it.

1 is compared to 5 and is inserted before 5.

6 is greater than 5 and 1.

2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.

And this goes on...

(Always we start with the second element as key.)

Sorting using Insertion Sort Algorithm

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, key;
for(i=1; i<6; i++){
key = a[i];
j = i-1;
while(j>=0 && key < a[j])
 {
a[j+1] = a[j];
 j--;
}
  a[j+1] = key;
}
```

Sorting using Insertion Sort Algorithm

- We took an array with 6 integers. We took a variable **key**, in which we put each element of the array, in each pass, starting from the second element, that is **a[1]**.
- Then using the while loop, we iterate, until **j** becomes equal to zero or we find an element which is greater than **key**, and then we insert the key at that position.
- In the above array, first we pick 1 as key, we compare it with 5(element before 1), 1 is smaller than 5, we shift 1 before 5. Then we pick 6, and compare it with 5 and 1, no shifting this time. Then 2 becomes the key and is compared with, 6 and 5, and then 2 is placed after 1. And this goes on, until complete array gets sorted.

Insertion Sorting in C++

```
#include <iostream>
using namespace std;
 //member functions declaration
void insertionSort(int arr[], int length);
void printArray(int array[],int size);
int main()
 {
int array[5]= {5,4,3,2,1};
insertionSort(array,5);
return 0;
}
```

```
void insertionSort(int arr[], int length)
{
int i, j ,tmp;
for (i = 1; i < length; i++) {
j = i;
while (j > 0 && arr[j - 1] > arr[j])
 {
tmp = arr[j];
arr[j] = arr[j - 1];
arr[j - 1] = tmp;
j--;
}
printArray(arr,5);
}
}
```

```
void printArray(int array[], int size)
{
cout<< "Sorting the array using Insertion sort ";
   int j;
for (j=0; j < size; j++)
cout <<" "<< array[j];
cout << endl;
}
```

Quick Sort Algorithm

- Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less additional space. It is based on the rule of **Divide and Conquer**(also called *partition-exchange sort*). This algorithm divides the list into three main parts :
  1. Elements less than the Pivot element
  2. Pivot element
  3. Elements greater than the pivot element

- Quick sort :It is a divide and conquer algorithm. Developed by Tony Hoare in 1959.
- Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements.
- Quick sort can then recursively sort the sub-arrays.

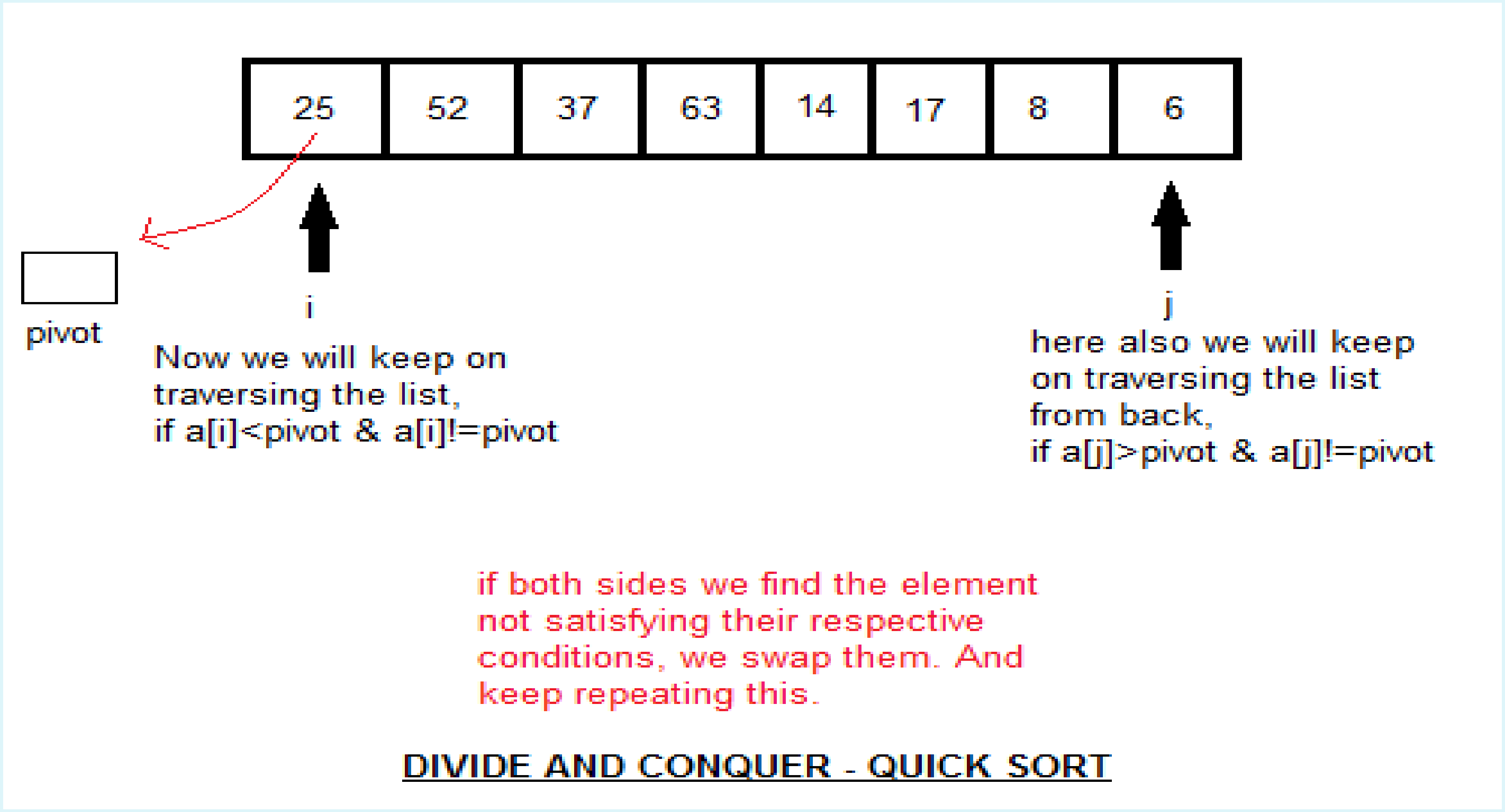
Quicksort is one the very popular sorting algorithm or technique to sort a collection of data. Quicksort is better because of few decent reasons.

- It does not need any temporary storing memory; which means if you don't need to invest any more storage capacity during the process. It makes sense when your data is quite large.
- It is very fast because it uses divide and conquers. In this algorithm, we choose a pivot and divide it into two sub-arrays and then repeat the process again.

ALGORITHM:

- Step 1: Pick an element, called a pivot, from the array.
- Step 2: Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
- Step 3: Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

How Quick Sorting Works



Now we will keep on traversing the list, if a[i]<pivot & a[i]!=pivot

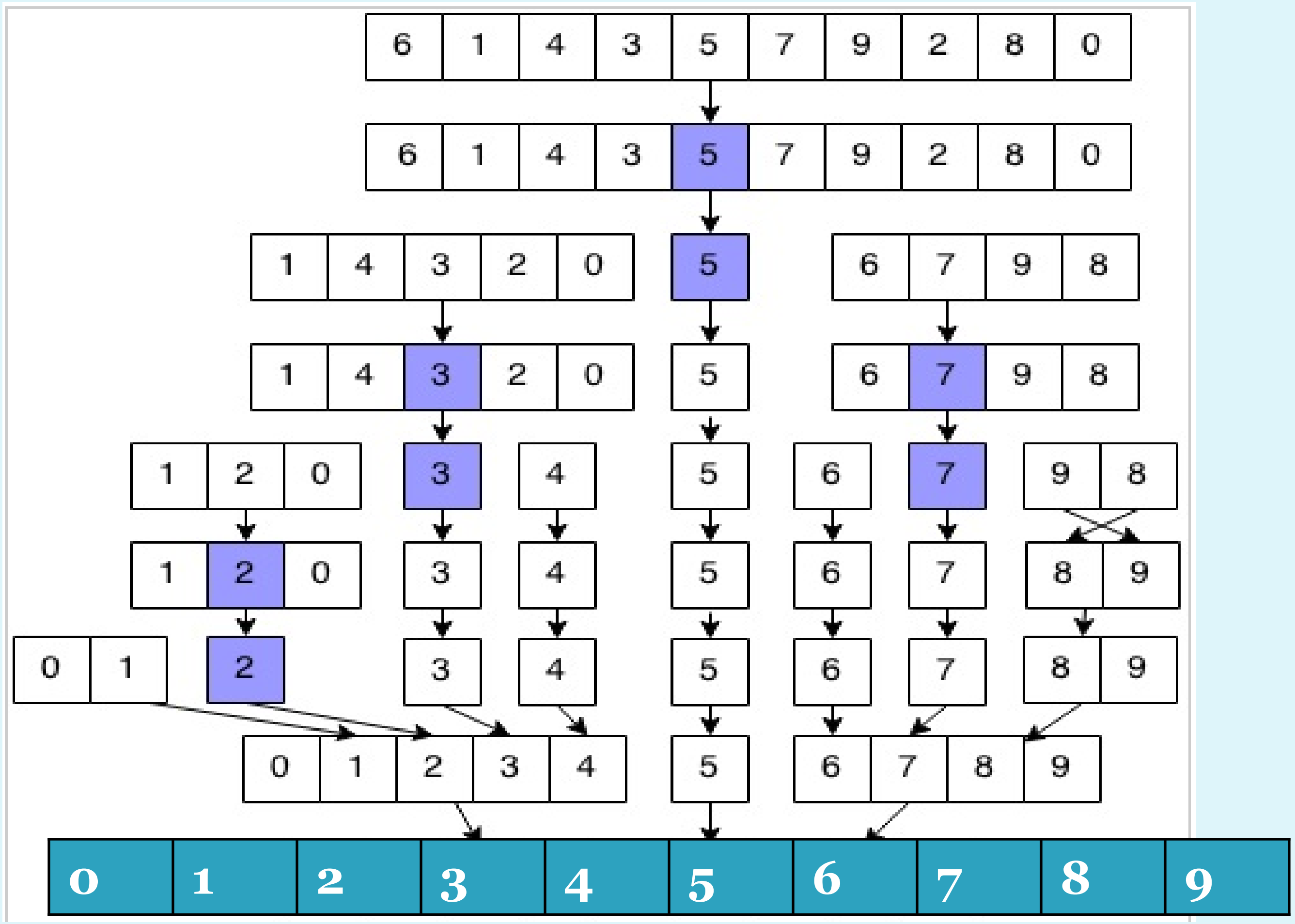
here also we will keep on traversing the list from back, if a[j]>pivot & a[j]!=pivot

if both sides we find the element not satisfying their respective conditions, we swap them. And keep repeating this.

DIVIDE AND CONQUER - QUICK SORT

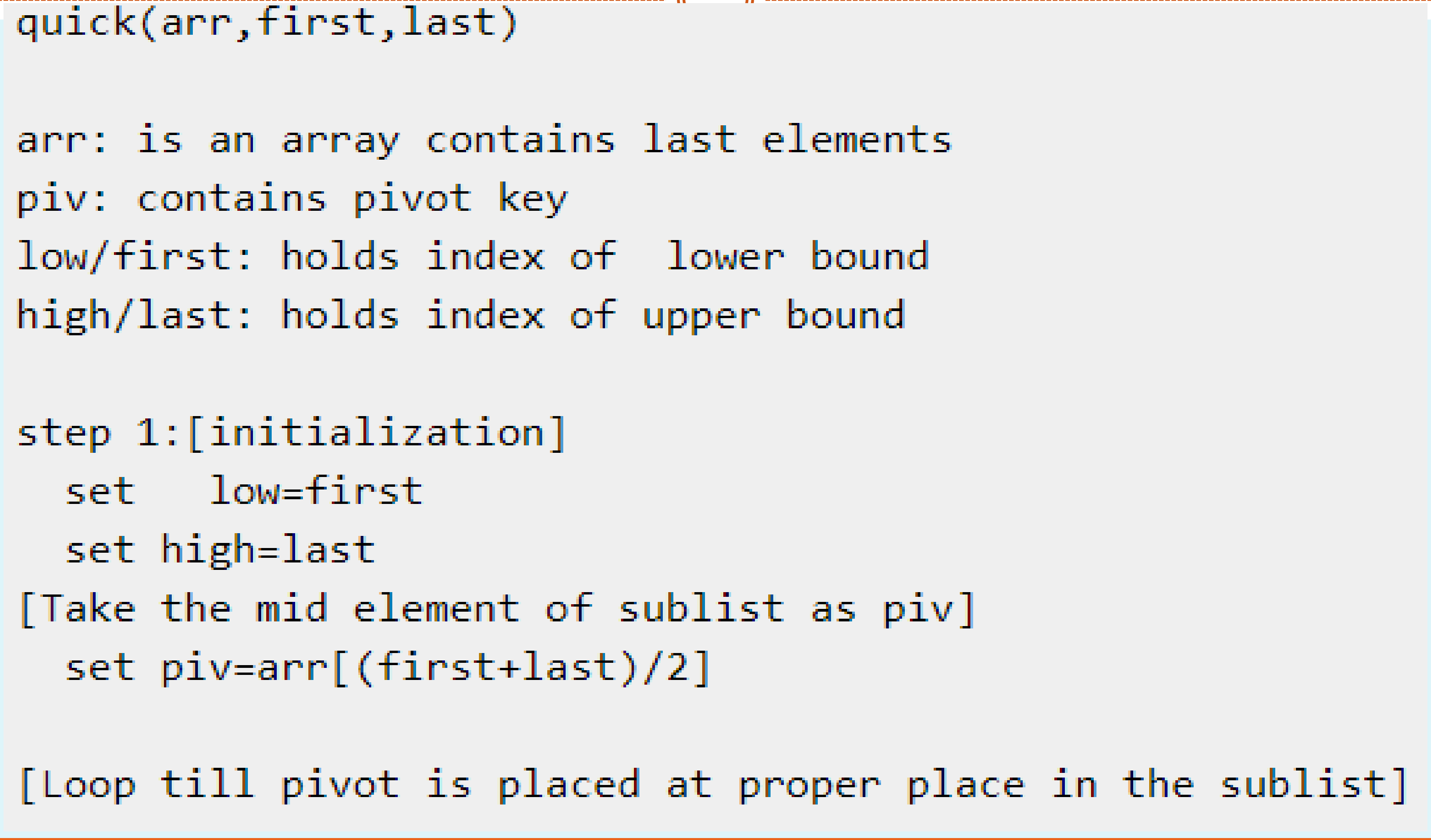
- In the list of elements, mentioned in below example, we have taken **25** as pivot. So after the first pass, the list will be changed like this.
- 6 8 17 14 **25** 63 37 52
- Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.

5 picked as pivote



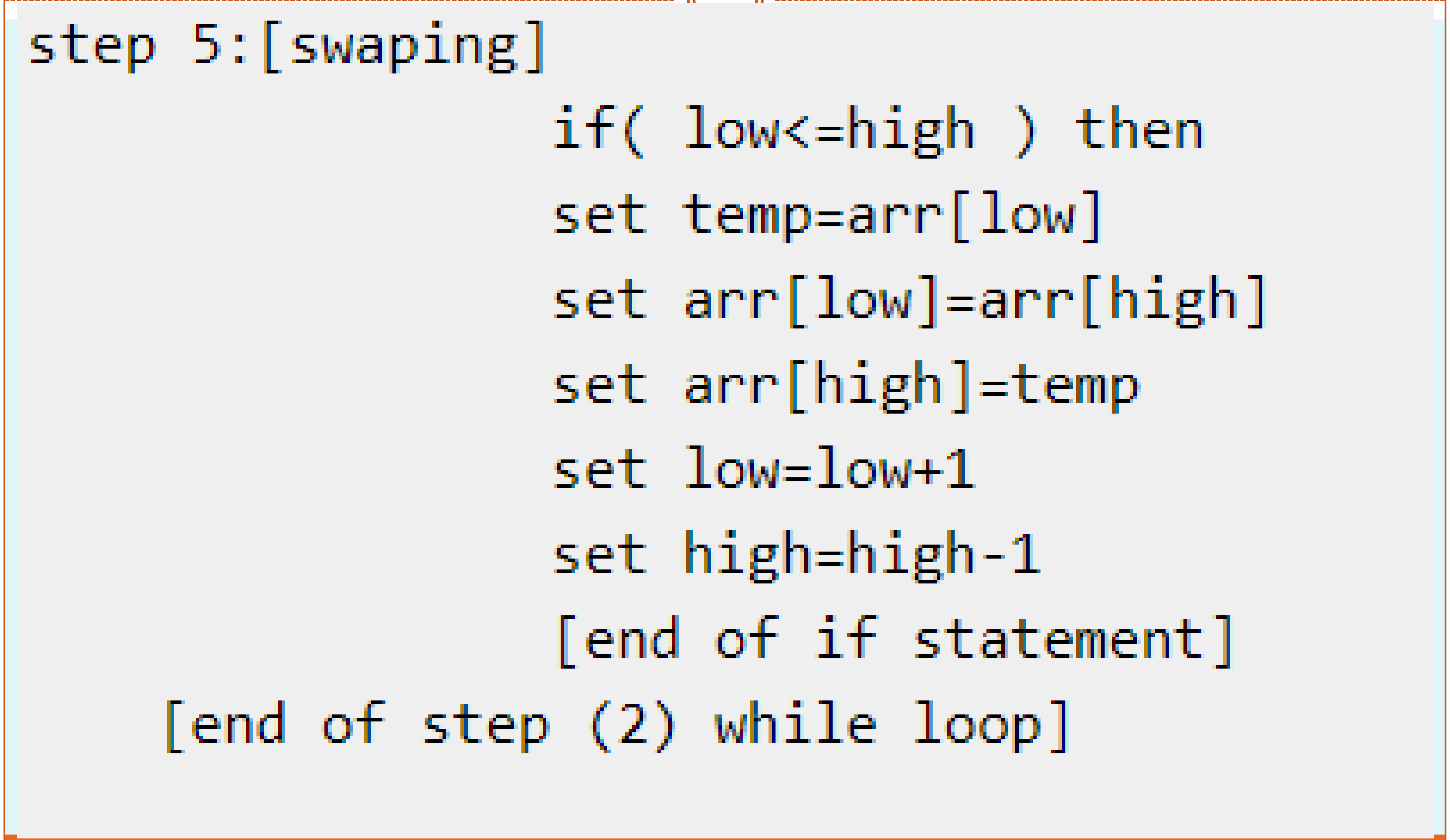
- The Above example shows it clearly that first choose the pivot which is 5 in the array 6 1 4 3 **5** 7 9 2 8 0. Now start from the first index and check if it's greater than the pivot, if you found a greater element which 6 in our case, 6 1 4 3 **5** 7 9 2 8 0. We will replace it with the element which is lower than the pivot and is in the right side of the pivot or the pivot itself in some case.
- When all the elements which are smaller than the pivot are on the left side and all the elements which are bigger than the pivot are on the right side, at that point we call the Quicksort function again with the different indices.
- This approach makes it possible to process in the sub-array without conflict the other elements of the main array.

Algorithm of Quicksort:

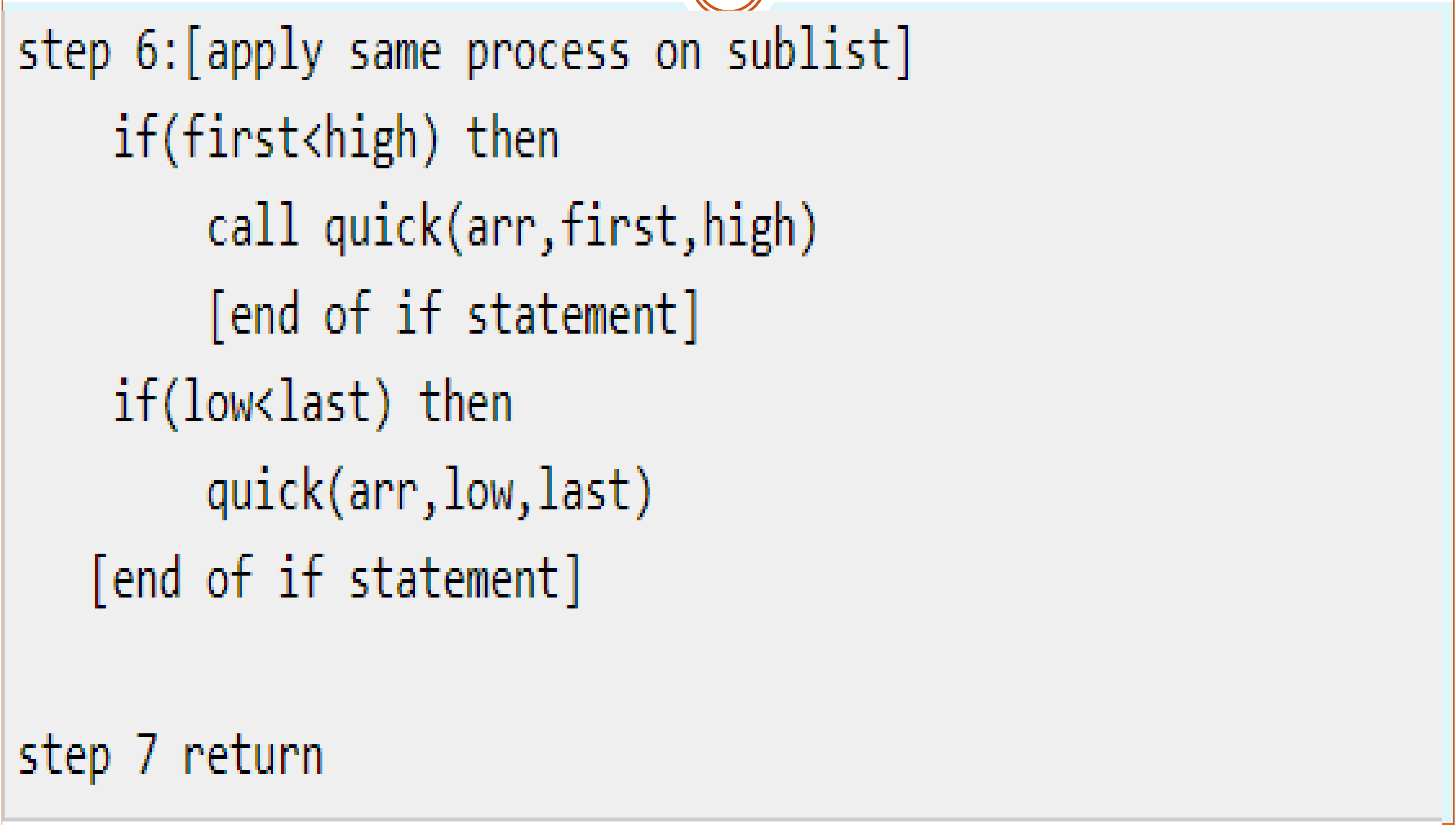


```
quick(arr,first,last)

arr: is an array contains last elements
piv: contains pivot key
low/first: holds index of  lower bound
high/last: holds index of upper bound

step 1:[initialization]
  set   low=first
  set high=last
[Take the mid element of sublist as piv]
  set piv=arr[(first+last)/2]

[Loop till pivot is placed at proper place in the sublist]
```

```
step 2: repeat step (3) to step (5) while low<=high

step 3:[Compare from  left side]
        repeat step (i)         while( arr[low]< piv )
                             i) set low=low+1
                             [end of step 3 loop]

step 4:[Compare from  right side]
                    repeat step (i)   while( arr[high]> piv )
                                   i)     set high=high -1
                                   [end of step 4 loop]
```



```
step 5:[swaping]
                   if( low<=high ) then
                   set temp=arr[low]
                   set arr[low]=arr[high]
                   set arr[high]=temp
                   set low=low+1
                   set high=high-1
                   [end of if statement]
      [end of step (2) while loop]
```



```
step 6:[apply same process on sublist]
    if(first<high) then
        call quick(arr,first,high)
        [end of if statement]
    if(low<last) then
        quick(arr,low,last)
   [end of if statement]


step 7 return
```

MERGE SORT

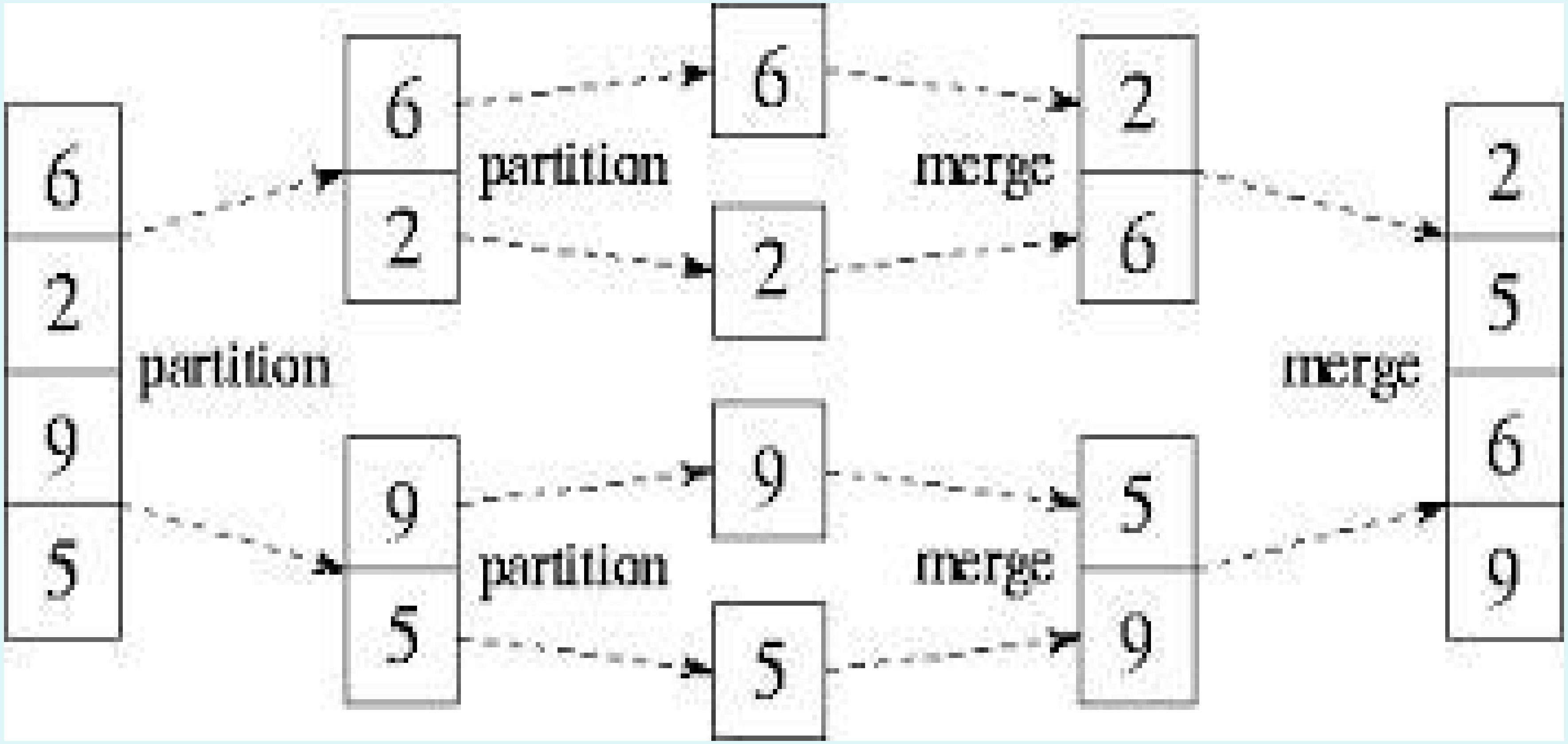
- Merge sort is a sorting technique based on divide and conquer technique. In merge sort the unsorted list is divided into N sublists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sublists, to produce new sorted sublists, and at lasts one sorted list is produced. Merge Sort is quite fast,
- It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

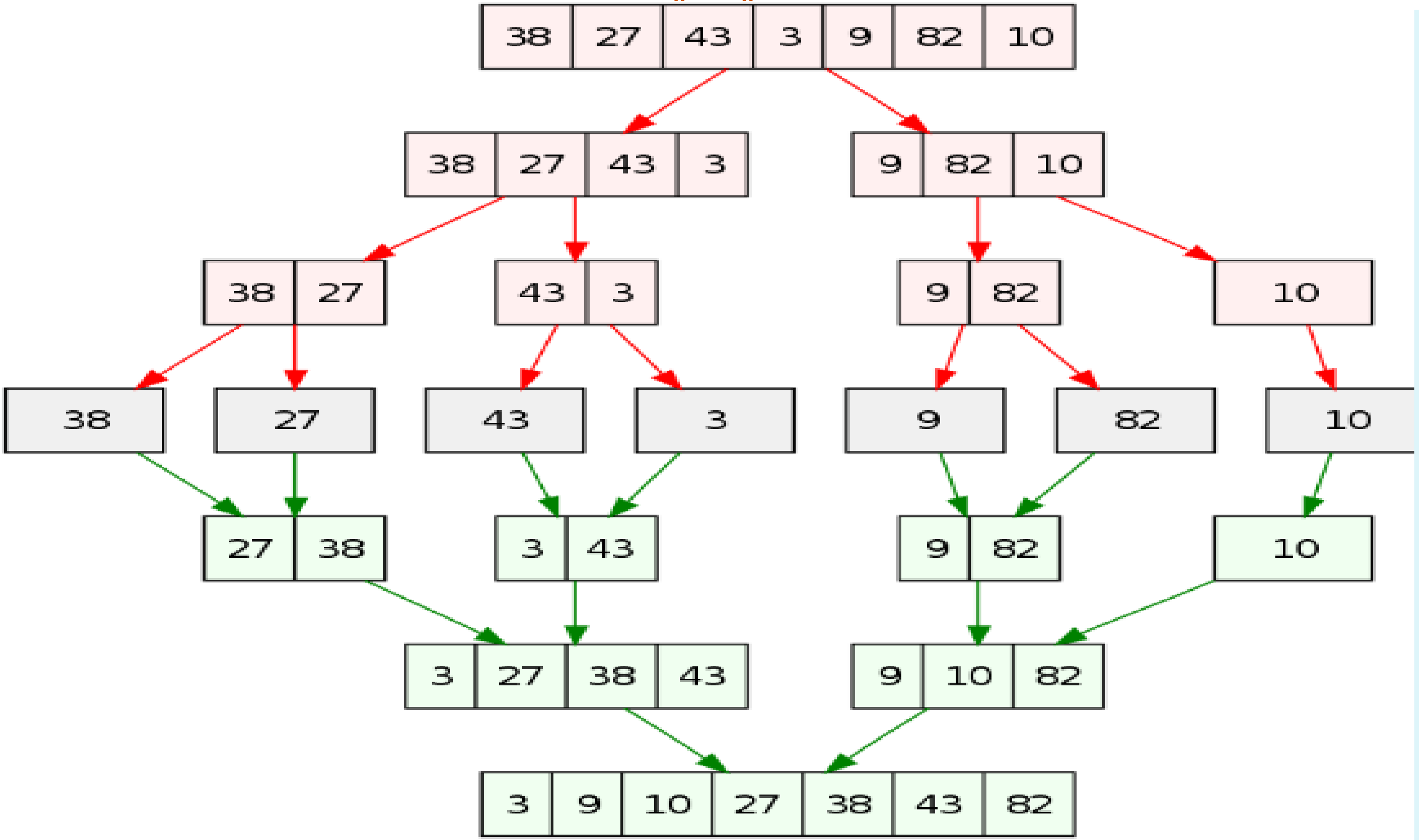
Conceptually, merge sort works as follows:

1. Divide the unsorted list into two sub lists of about half the size.

2. Divide each of the two sub lists recursively until we have list sizes of length 1, in which case the list itself is returned.

3. Merge the two sub lists back into one sorted list.


6
2
9
5
partition
6
2
9
5
partition
partition
6
2
9
5
merge
merge
2
6
5
9
merge
2
5
6
9


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

HEAP SORT

- Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heapsort algorithm uses one of the tree concepts called **Heap Tree**. In this sorting algorithm, we use **Max Heap** to arrange list of elements in Descending order and **Min Heap** to arrange list elements in Ascending order.
- It is a completely binary tree with the property that a parent is always greater than or equal to either of its children (if they exist). first the heap (max or min) is created using binary tree and then heap is sorted using priority queue.

Heap sort Algorithm

- The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

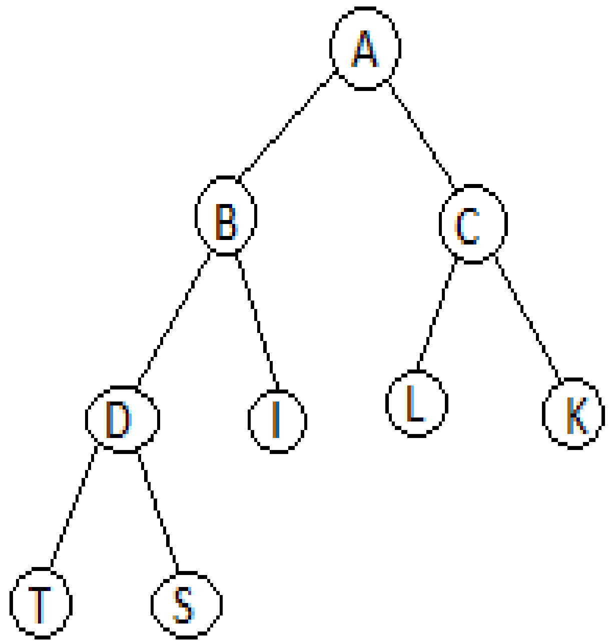
- **Step 1 -** Construct a **Binary Tree** with given list of Elements.
- **Step 2 -** Transform the Binary Tree into **Min Heap.**
- **Step 3 -** Delete the root element from Min Heap using **Heapify** method.
- **Step 4 -** Put the deleted element into the Sorted list.
- **Step 5 -** Repeat the same until Min Heap becomes empty.
- **Step 6 -** Display the sorted list.

Heap sort Algorithm

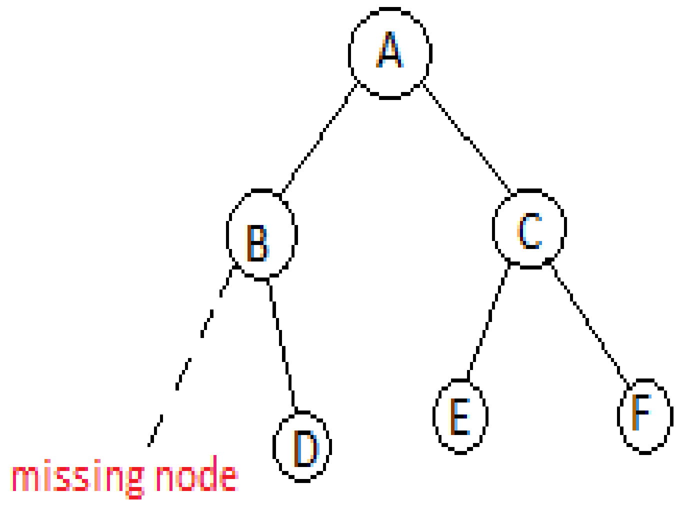
- The Heap sort algorithm to arrange a list of elements in descending order is performed using following steps...

- **Step 1 -** Construct a **Binary Tree** with given list of Elements.
- **Step 2 -** Transform the Binary Tree into **Max Heap.**
- **Step 3 -** Delete the root element from Max Heap using **Heapify** method.
- **Step 4 -** Put the deleted element into the Sorted list.
- **Step 5 -** Repeat the same until Max Heap becomes empty.
- **Step 6 -** Display the sorted list.

- Heap is a special tree-based data structure, that satisfies the following special heap properties :
- **Shape Property :** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.



Complete Binary Tree



In-Complete Binary Tree

- **Heap Property :** All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children.
- If the parent nodes are greater than their children, heap is called a **Max-Heap**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.

Heap sort steps explained in the class using white board

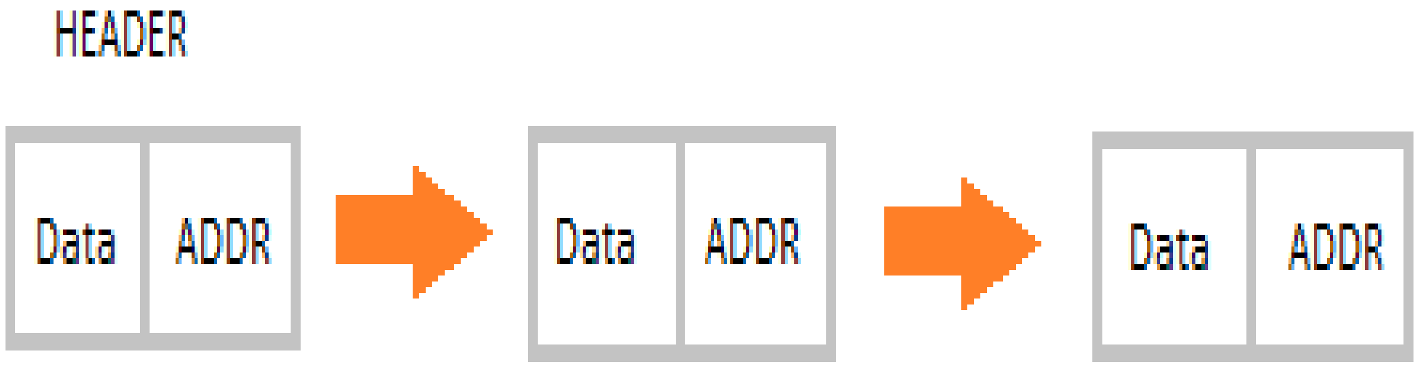
Introduction to Linked Lists

DATA STRUCTURES

2ND STAGE

Linked List

- Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs.


HEADER
Data
ADDR
Data
ADDR
Data
ADDR

Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

Disadvantages of Linked Lists

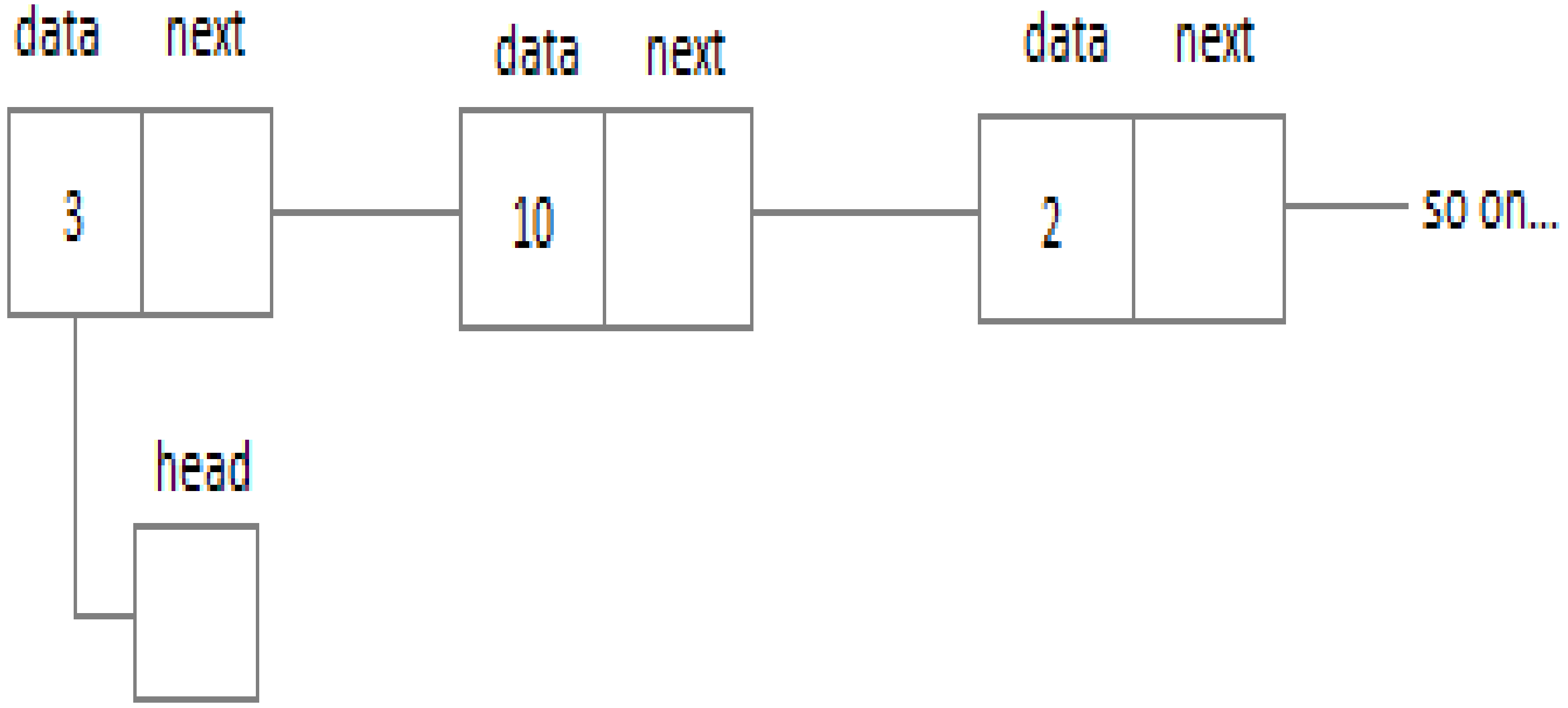
- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don’t need to know the size in advance.

Types of Linked Lists

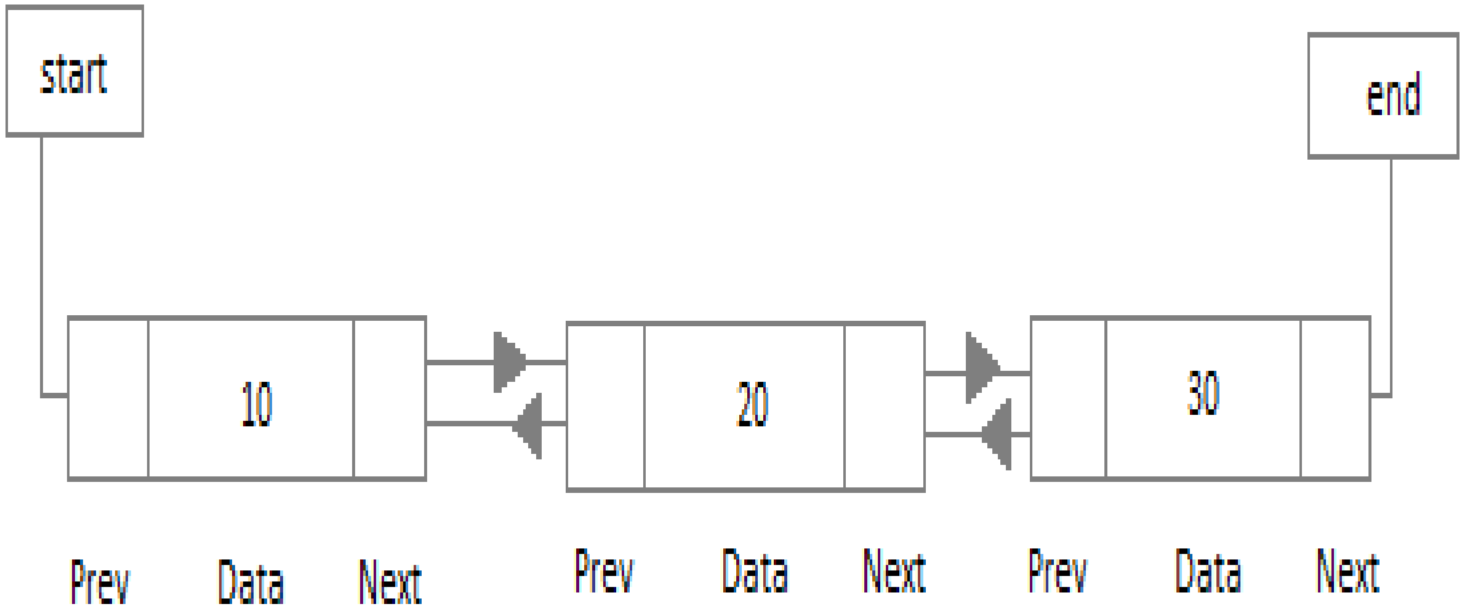
1. **Singly Linked List :** Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.



Singly Linked List

Types of Linked Lists

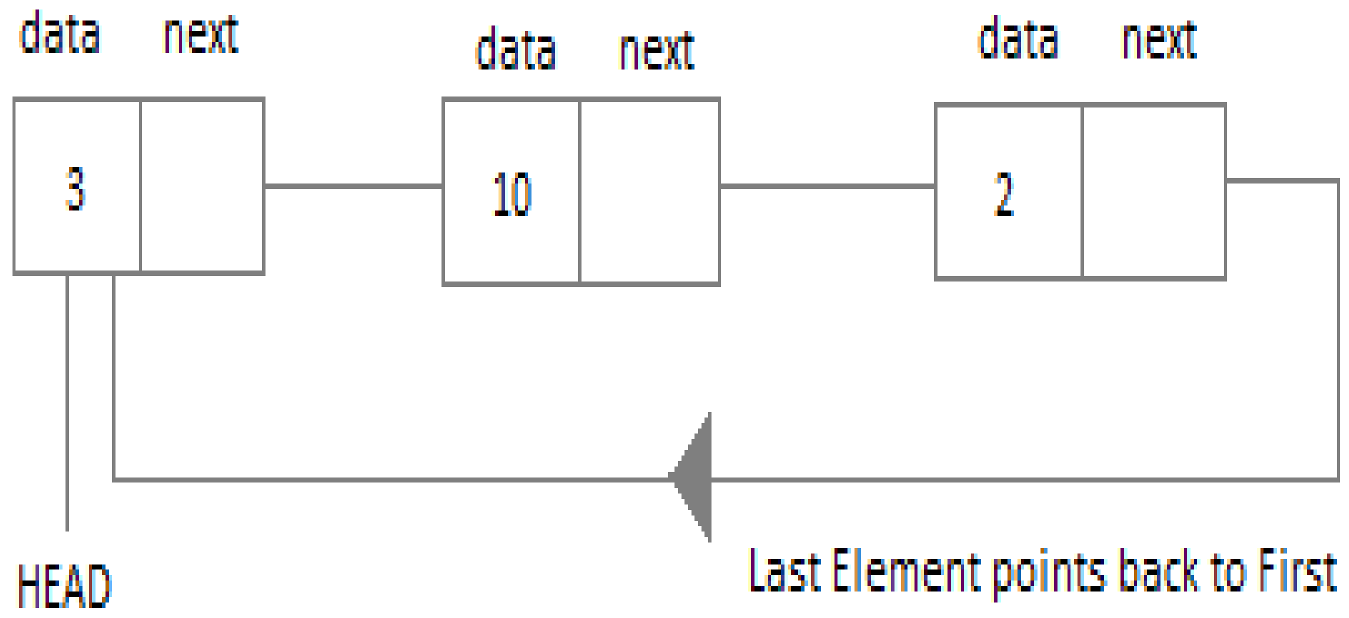
2. Doubly Linked List : In a doubly linked list, each node contains two links the first link points to the previous node and the other link points to the next node in the sequence.



Doubly Linked List

Types of Linked Lists

3.Circular Linked List : In the circular linked list the last node of the list contains the address of the first node and forms a circular chain.



Circular Linked List

Linear Linked List

- The element can be inserted in linked list in 2 ways :
  - Insertion at beginning of the list.
  - Insertion at the end of the list.
- We will also be adding some more useful methods like :
  - Checking whether Linked List is empty or not.
  - Searching any element in the Linked List
  - Deleting a particular Node from the List

Example

- Before inserting the node in the list we will create a class **Node**. Like shown below :

```
class Node {
 public:
 int data;
 //pointer to the next node
 node* next;

 node() {
  data = 0;
   next = NULL;
}
node(int x)
{
   data = x;
   next = NULL;
}
}
```

Node class basically creates a node for the data which you enter to be included into Linked List. Once the node is created, we use various functions to fit in that node into the Linked List.

Linked list class

```
class LinkedList {
 public:
 node *head;
//declaring the functions
//function to add Node at front
 int addAtFront(node *n);
 //function to check whether Linked list is empty
int isEmpty();
 //function to add Node at the End of list
 int addAtEnd(node *n);
//function to search a value
node* search(int k);
//function to delete any Node
node* deleteNode(int x);
 LinkedList() {
 head = NULL;
 }
}
```

Insertion at the Beginning

- Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the first Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

int LinkedList**addAtFront**(node *n)
 { int i = 0;
//making the next of the new Node point to Head n-
   >next = head;
 //making the new Node as Head
 head = n;
   i++;
 //returning the position where Node is added return
   i;
}

-> for accessing object member variables and methods via pointer to object

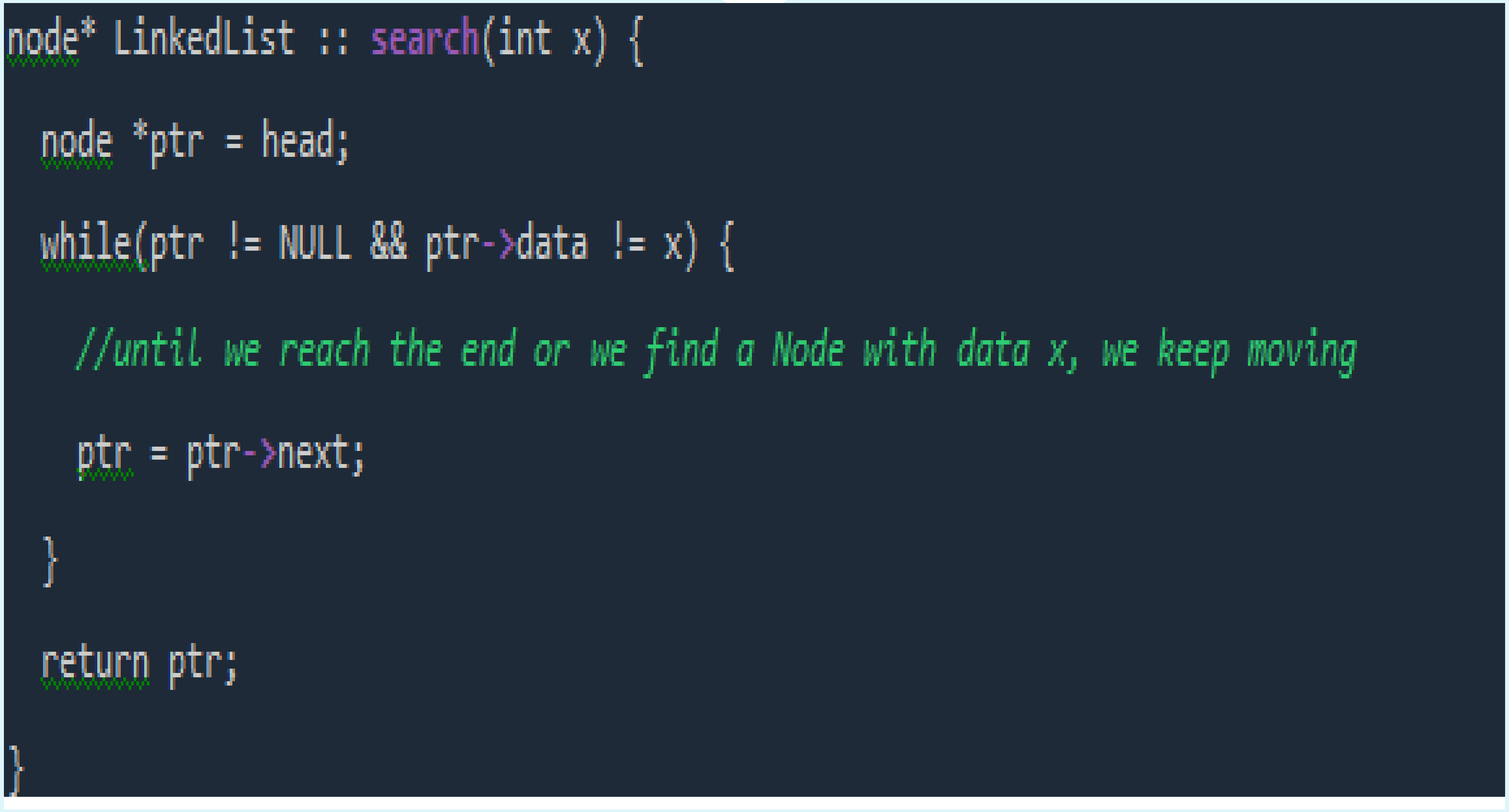
Inserting at the End

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it’s next to the new Node, hence making the new node the last Node.

```
int LinkedListaddAtEnd(node *n) {
 //If list is empty
 if(head == NULL)
{
//making the new Node as Head
head = n;
//making the next pointe of the new Node as Null
 n->next = NULL;
}
else {
 //getting the last node
 node *n2 = getLastNode();
 n2->next = n;
 }
}
 node* LinkedList getLastNode()
{
//creating a pointer pointing to Head
 node* ptr = head;
//Iterating over the list till the node whose Next pointer points to null
//Return that node, because that will be the last node.
 while(ptr->next!=NULL)
{
 //if Next is not Null, take the pointer one step forward
ptr = ptr->next;
 }
 return ptr;
}
```

Searching for an Element in the List

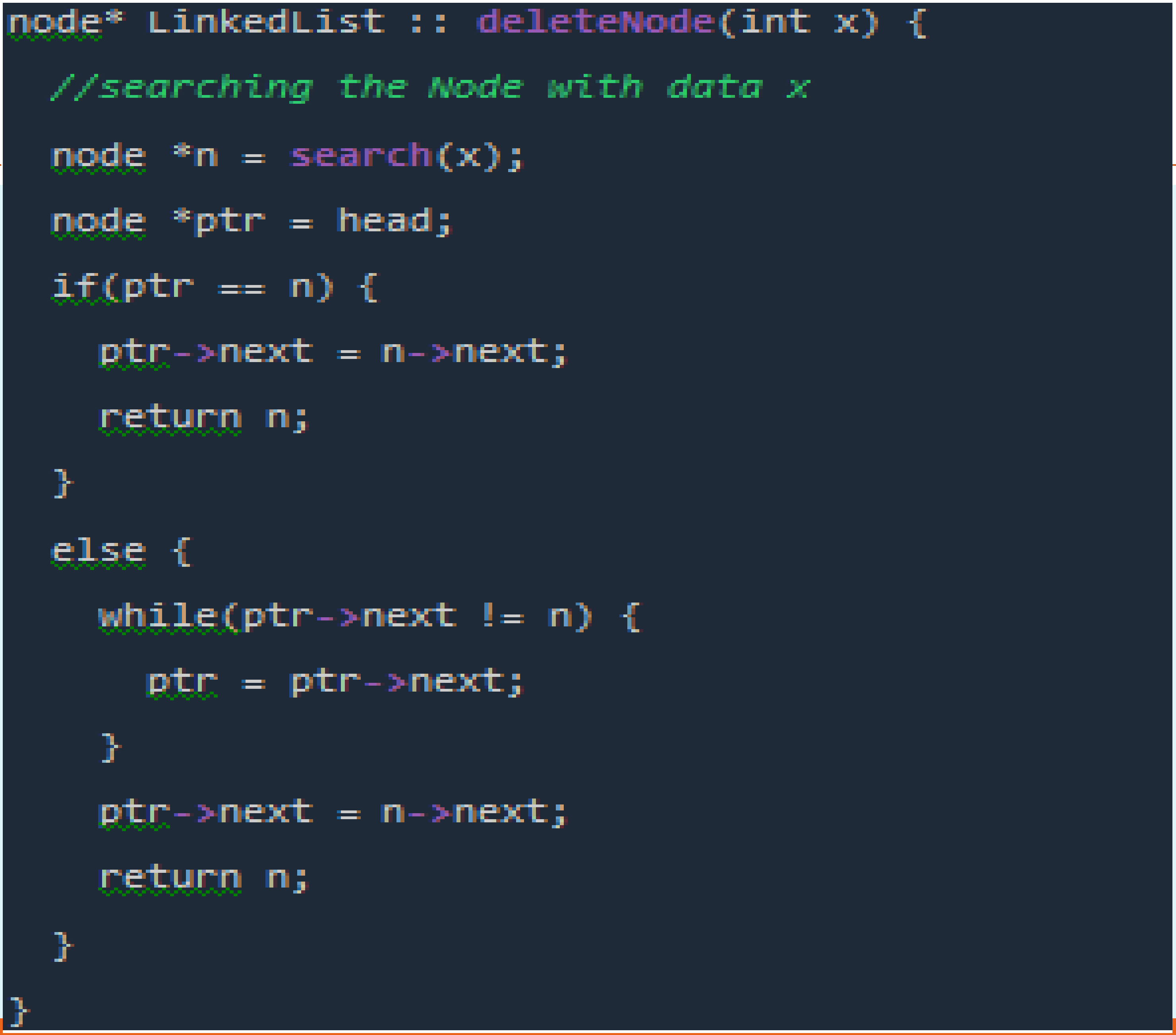
- In searching we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.



```
node* LinkedList :: search(int x) {
  node *ptr = head;
  while(ptr != NULL && ptr->data != x) {
    //until we reach the end or we find a Node with data x, we keep moving
    ptr = ptr->next;
  }
  return ptr;
}
```

Deleting a Node from the List

- Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.
  - To remove any Node from the list, we need to do the following :
  - If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted.
  - If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.



```
node* LinkedList :: deleteNode(int x) {
  //searching the Node with data x
  node *n = search(x);
  node *ptr = head;
  if(ptr == n) {
    ptr->next = n->next;
    return n;
  }
  else {
    while(ptr->next != n) {
      ptr = ptr->next;
    }
    ptr->next = n->next;
    return n;
  }
}
```

Checking whether the List is empty or not

We just need to check whether the **Head** of the List is NULL or not.

```
int LinkedListisEmpty() {
 if(head == NULL) {
return 1;
}  else
{ return 0;
}
}
```

- If you are still figuring out, how to call all these methods, then below is how your main() method will look like. As we have followed OOP standards, we will create the objects of **LinkedList** class to initialize our List and then we will create objects of **Node** class whenever we want to add any new node to the List.

```
int main() {
LinkedList L;
//We will ask value from user, read the value and add the value to our Node
 int x;
cout << "Please enter an integer value : ";
 cin >> x;
 Node *n1;
 //Creating a new node with data as x
 n1 = new Node(x);
 //Adding the node to the list
L.addAtFront(n1);
}
```

Similarly you can call any of the functions of the LinkedList class, add as many Nodes you want to your List.

Linear data structures

- Linear data structures organize their data elements in a linear fashion, where data elements are attached one after the other. Data elements in a liner data structure are traversed one after the other and only one element can be directly reached while traversing. Linear data structures are very easy to implement, since the memory of the computer is also organized in a linear fashion.
- Some commonly used linear data structures are arrays, linked lists, stacks and queues. An arrays is a collection of data elements where each element could be identified using an index. A linked list is a sequence of nodes, where each node is made up of a data element and a reference to the next node in the sequence. A stack is actually a list where data elements can only be added or removed from the top of the list. A queue is also a list, where data elements can be added from one end of the list and removed from the other end of the list.

Nonlinear data structures

- In nonlinear data structures, data elements are not organized in a sequential fashion. A data item in a nonlinear data structure could be attached to several other data elements to reflect a special relationship among them and all the data items cannot be traversed in a single run.
- Data structures like multidimensional arrays, trees and graphs are some examples of widely used nonlinear data structures.
  - A multidimensional array is simply a collection of one-dimensional arrays.
  - A tree is a data structure that is made up of a set of linked nodes, which can be used to represent a hierarchical relationship among data elements.
  - A graph is a data structure that is made up of a finite set of edges and vertices. Edges represent connections or relationships among vertices that stores data elements.

Difference between Linear and Nonlinear Data Structures

- Main difference between linear and nonlinear data structures lie in the way they organize data elements. In linear data structures, data elements are organized sequentially and therefore they are easy to implement in the computer's memory. In nonlinear data structures, a data element can be attached to several other data elements to represent specific relationships that exist among them. Due to this nonlinear structure, they might be difficult to be implemented in computer's linear memory compared to implementing linear data structures. Selecting one data structure type over the other should be done carefully by considering the relationship among the data elements that needs to be stored.

Graph

- A graph is a set of items that are connected by edges and each item is known as node or vertex. In other words, a graph can be defined as the set of vertices and there is a binary relation between these vertices.

- In implementation of a graph, the nodes are implemented as objects or structures. The edges can be represented in different ways. One of the ways is that each node can be associated with an incident edges array. If the information is to be stored in nodes rather than edges then the arrays acts as pointers to nodes and also represent edges. One of the advantages of this approach is that additional nodes can be added to the graph. Existing nodes can be connected by adding elements to arrays. But there is one disadvantage because time is required in order to determine whether there is an edge between the nodes.

- Other way to do this is to keep a two dimensional array or matrix M that has Boolean values. The existence of edge from node i to j is specified by entry Mij. One of the advantages of this method is to find out if there is any edge between two nodes.

Tree

- Tree is also a data structure used in computer science. It is similar to the structure of the tree and has a set of nodes that are linked to each other.
- A node of a tree may contain a condition or value. It can also be a tree of its own or it can represent a separate data structure. Zero or more nodes are present in a tree data structure. If a node has a child then it is called parent node of that child. There can be at most one parent of a node. The longest downward path from the node to a leaf is the height of the node. The depth of node is represented by the path to its root.

- In a tree, the topmost node is called root node. The root node has no parents as it is the top most one. From this node, all tree operations begin. By using links or edges, other nodes can be reached from the root node. The bottom-most level nodes are called leaf nodes and they don’t have any children. The node that has number of child nodes is called inner node or internal node.

Difference between graph and tree:

- A tree can be described as a specialized case of graph with no self loops and circuits.
- There are no loops in a tree whereas a graph can have loops.
- There are three sets in a graph i.e. edges, vertices and a set that represents their relation while a tree consists of nodes that are connected to each other. These connections are referred to as edges.
- In tree there are numerous rules spelling out how connections of nodes can occur whereas graph has no rules dictating the connection among the nodes.

Tree data structure

2ND STAGE

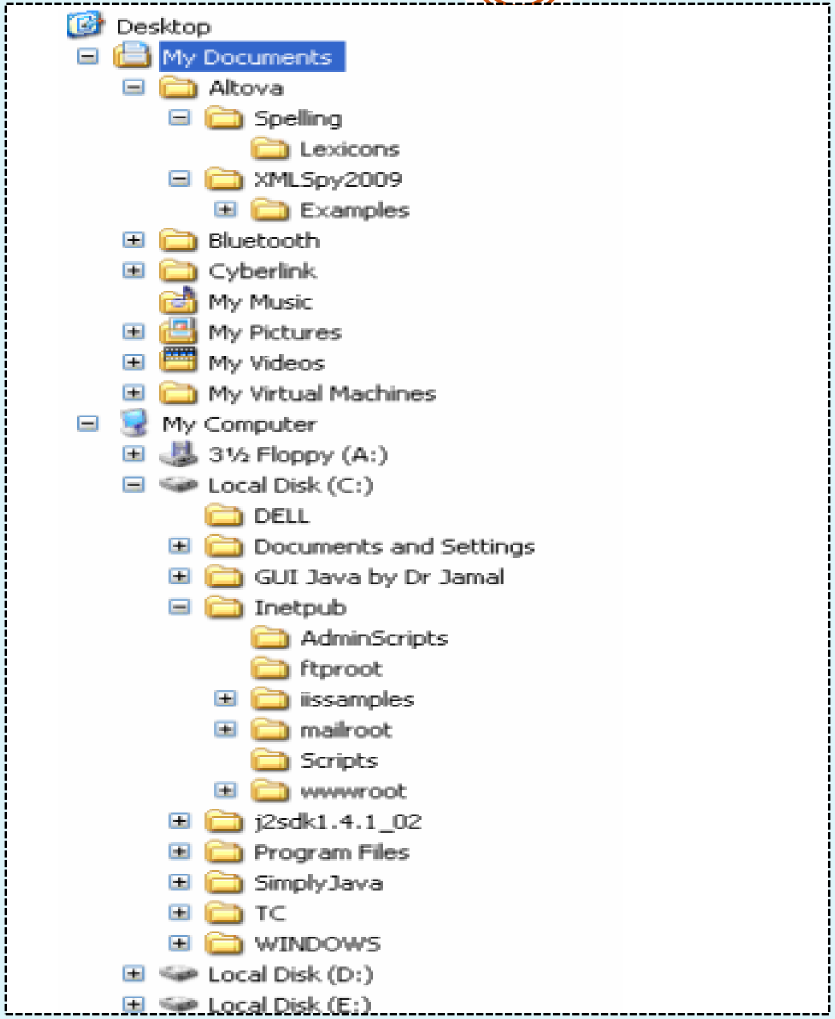
Tree

- Tree: So far, we have been studying mainly linear types of data structures: arrays, lists, stacks and queues. Now we defines a nonlinear data structure called Tree. This structure is mainly used to represent data containing a hierarchical relationship between nodes/elements e.g. family trees and tables of contents. There are two main types of tree:
- General Tree
- Binary Tree

General Tree

- General Tree: A tree where a node can has any number of children / descendants is called General Tree. For example:

Following figure is also an example of general tree where root is “Desktop”.



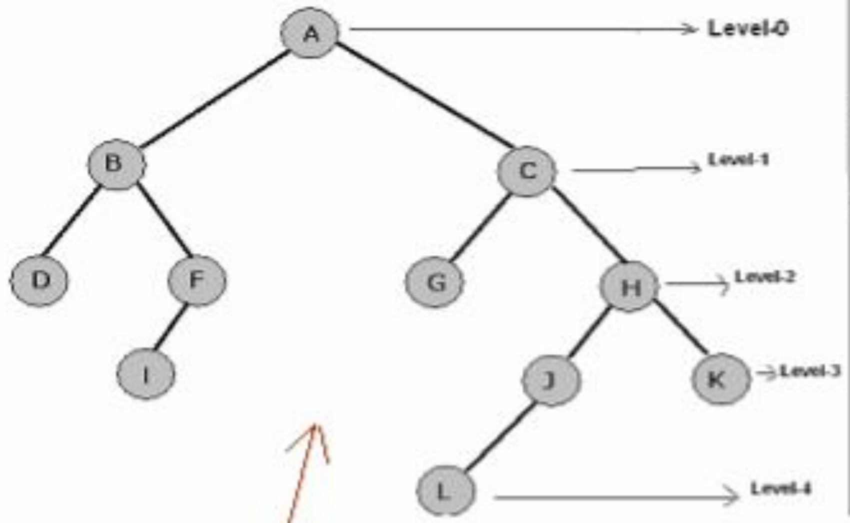
```
Desktop
  My Documents
    Altova
      Spelling
        Lexicons
      XMLSpy2009
        Examples
    Bluetooth
    Cyberlink
    My Music
    My Pictures
    My Videos
    My Virtual Machines
  My Computer
    3½ Floppy (A:)
    Local Disk (C:)
      DELL
      Documents and Settings
      GUI Java by Dr Jamal
      Inetpub
        AdminScripts
        ftproot
        iissamples
        mailroot
        Scripts
        wwwroot
      j2sdk1.4.1_02
      Program Files
      SimplyJava
      TC
      WINDOWS
    Local Disk (D:)
    Local Disk (E:)
```

Binary Tree

- Binary Tree: A tree in which each element may has 0-child , 1-child or maximum of 2-children. A Binary Tree T is defined as finite set of elements, called nodes, such that
- a) T is empty (called the null tree or empty tree.)
- b) T contains a distinguished node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T1 and T2.
- If T does contain a root R, then the two trees T1 and T2 are called, respectively, the left sub tree and right sub tree of R.

- If T1 is non empty, then its node is called the left successor of R; similarly, if T2 is non empty, then its node is called the right successor of R. The nodes with no successors are called the terminal nodes.
- If N is a node in T with left successor S1 and right successor S2, then N is called the parent(or father) of S1 and S2. Analogously, S1 is called the left child (or son) of N, and S2 is called the right child (or son) of N.
- Furthermore, S1 and S2 are said to siblings (or brothers). Every node in the binary tree T, except the root, has a unique parent, called the predecessor of N.
- The line drawn from a node N of T to a successor is called an edge, and a sequence of consecutive edges is called a path.
- A terminal node is called a leaves, and a path ending in a leaves is called a branch.

- The depth (or height) of a tree T is the maximum number of nodes in a branch of T. This turns out to be 1 more than the largest level number of T. Level of node & its generation: Each node in binary tree T is assigned a level number, as follows.
- The root R of the tree T is assigned the level number 0, and every other node is assigned a level number which is 1 more than the level number of its parent. Furthermore, those nodes with the same level number are said to belong to the same generation.



ROOT A
Left-Subtree of A i.e. {B, D, F, I}
Right-Subtre of A i.e. {C, G, H, J, K, L}
Left Successor: of A ---> B
Right Successor of A ---> C
Levels of Tree In This tree there are 0 to 4 Levels
Height / Depth of Tree 5
Terminal / External Node / **leaves**

(D, I, G, L, K)

Internal Nodes

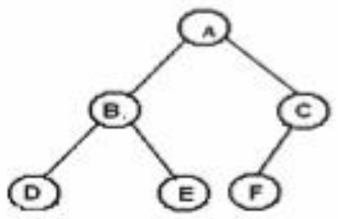
(B, C, F, H, J)

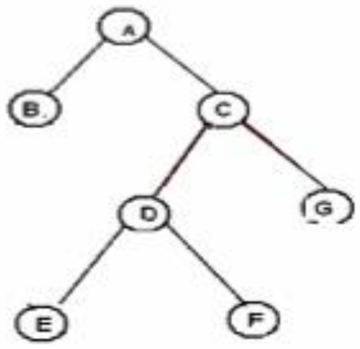
Nodes have one successor

(F, J)

Binary Tree

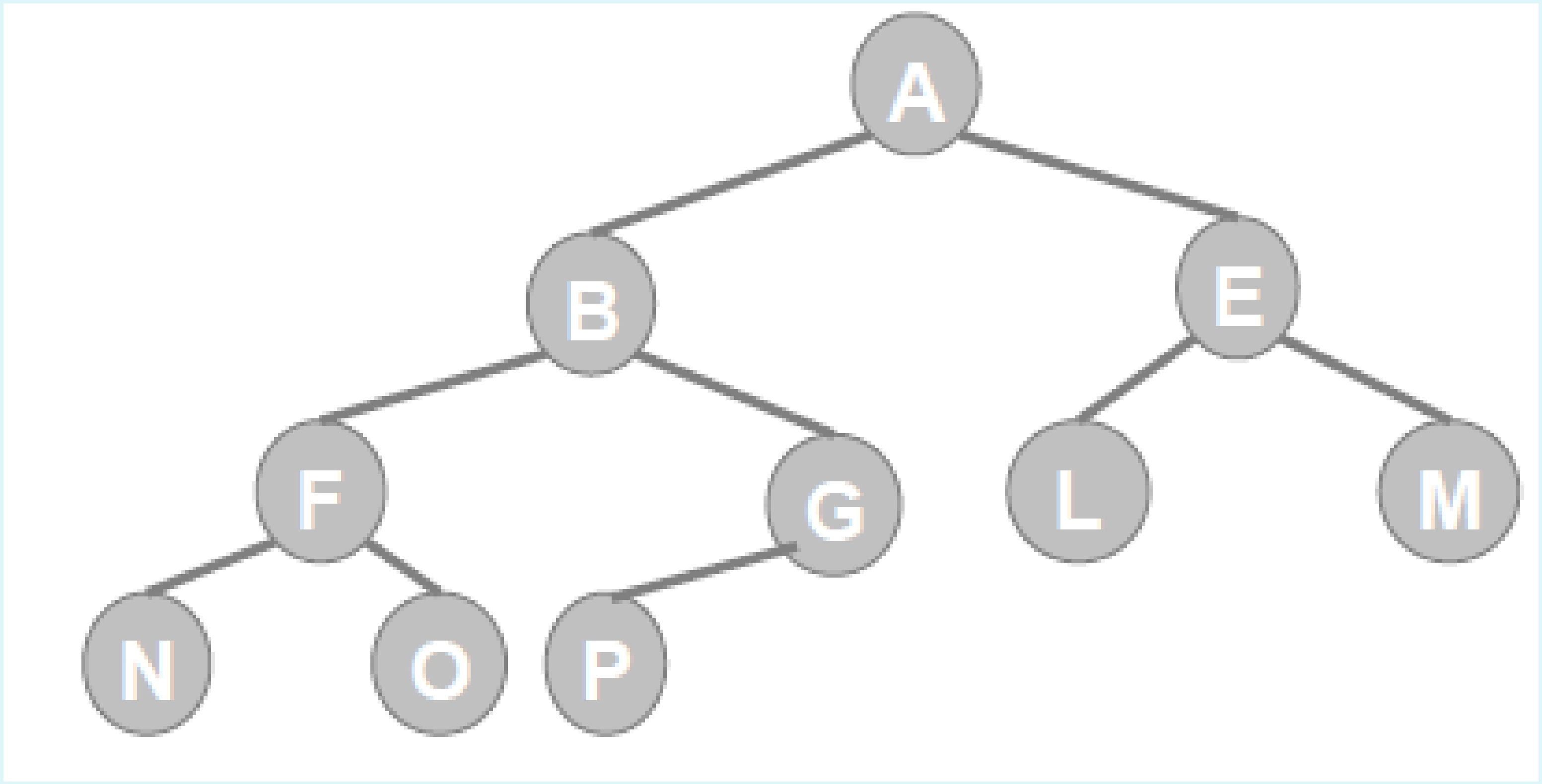
1- Simple Binary Tree
2- Complete Binary Tree
3- 2-Tree or Extended Tree





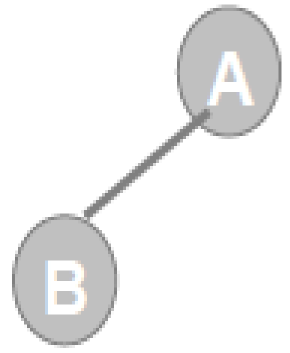
Complete Binary Tree

- Complete Binary Tree: Consider a binary tree T. each node of T can have at most two children. Accordingly. A tree is said to be complete if all its levels, except possibly the last have the maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.

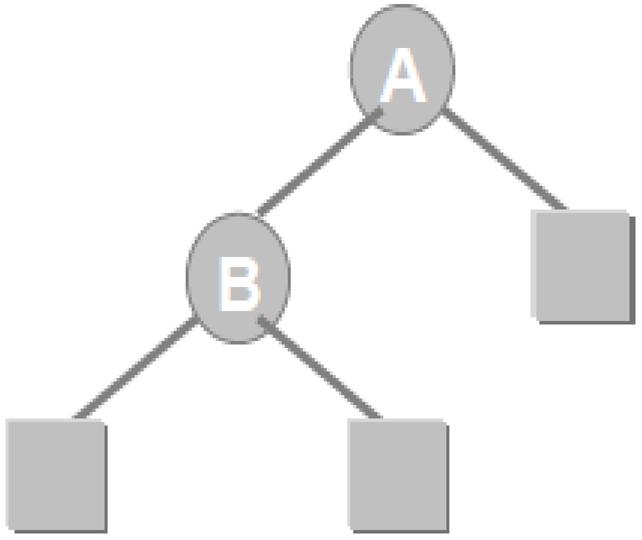

A
B
E
F
G
L
M
N
O
P

Extended Binary Tree: 2-Tree

- **Extended Binary Tree: 2-Tree:**
- A binary tree T is said to be a 2-tree or an extended binary tree if each node:
  - N has either 0 or 2 children.
  - In such a case, the nodes, with 2 children are called internal nodes,
  - node with 0 children are called external node.

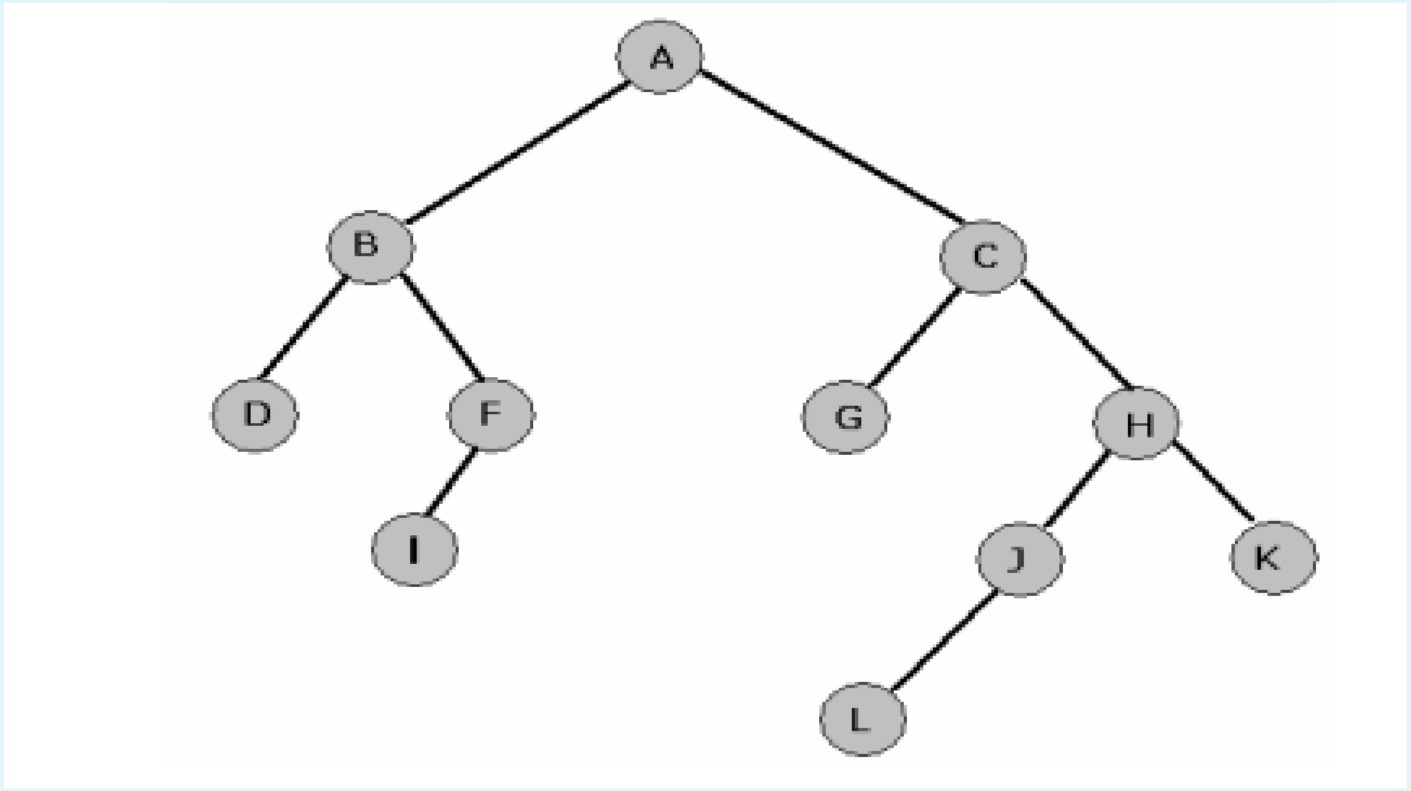


Binary Tree



Extended 2-tree

Traversing of Binary Tree:



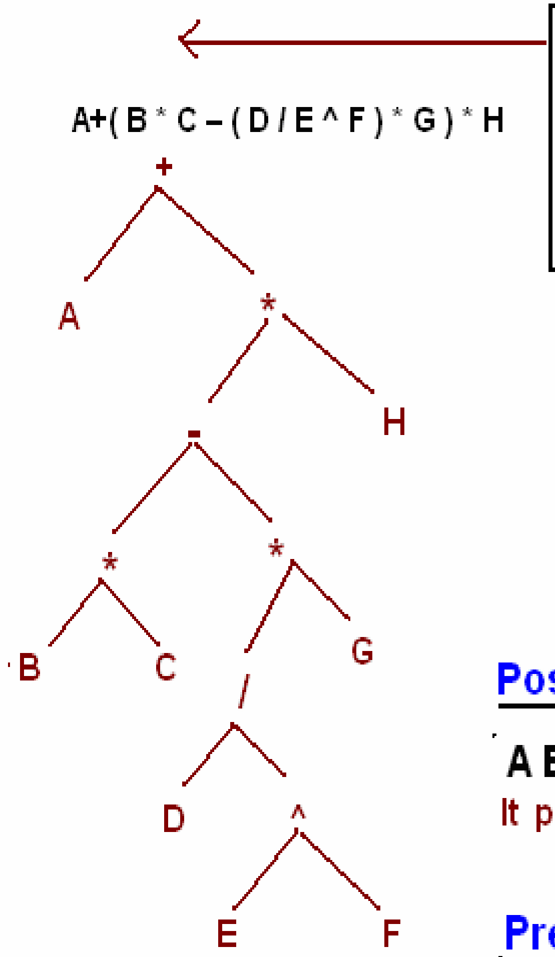
Traversing of Binary Tree:

A traversal of a tree T is a systematic way of accessing or visiting all the node of T. There are three standard ways of traversing a binary tree T with root R. these are :

- **Preorder (N L R):** **(A B D F I C G H J L K)**
  - a) Process the node/root.
  - b) Traverse the Left sub tree.
  - c) Traverse the Right sub tree.
- **Inorder (L N R):** **(D B I F A G C L J H K)**
  - a) Traverse the Left sub tree.
  - b) Process the node/root.
  - c) Traverse the Right sub tree.
- **Postorder (L R N):** **(D I F B G L J K H C A)**
  - a) Traverse the Left sub tree.
  - b) Traverse the Right sub tree.
  - c) Process the node/root.
- **Descending order (R N L):** **(K H J L C G A F I B D)**

  (Used in Binary Search Tree, will be discussed later)
  - a) Traverse the Right sub tree.
  - b) Process the node/root.
  - c) Traverse the Left sub tree.

Preparing a Tree from an infix arithmetic expression



Find operator which has low priority with respect to execution, startng from right to left. Put it on root and then continue this processor on sub-trees. The infix expression on the left side is converted into tree. Examin this tree is a 2-Tree, where each node either has two nodes or zero nodes.

All internal nodes are Operators

+ * - * * /

and all leaf nodes are Oprands

A H B C G D E F

Post Order Traversing (LRN)

A B C * D E F ^ / G * - H * +

It produce postfix expression

Pre-Order Traversing (NLR)

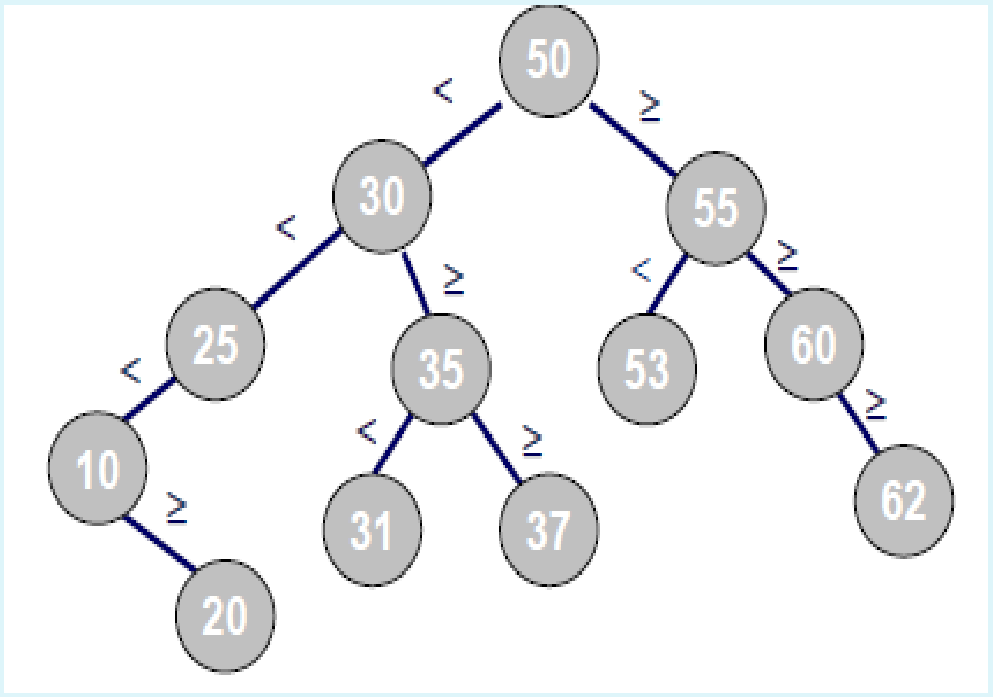
+ A * - * B C * / D ^ E F G H

it produce prefix arithmatic expression

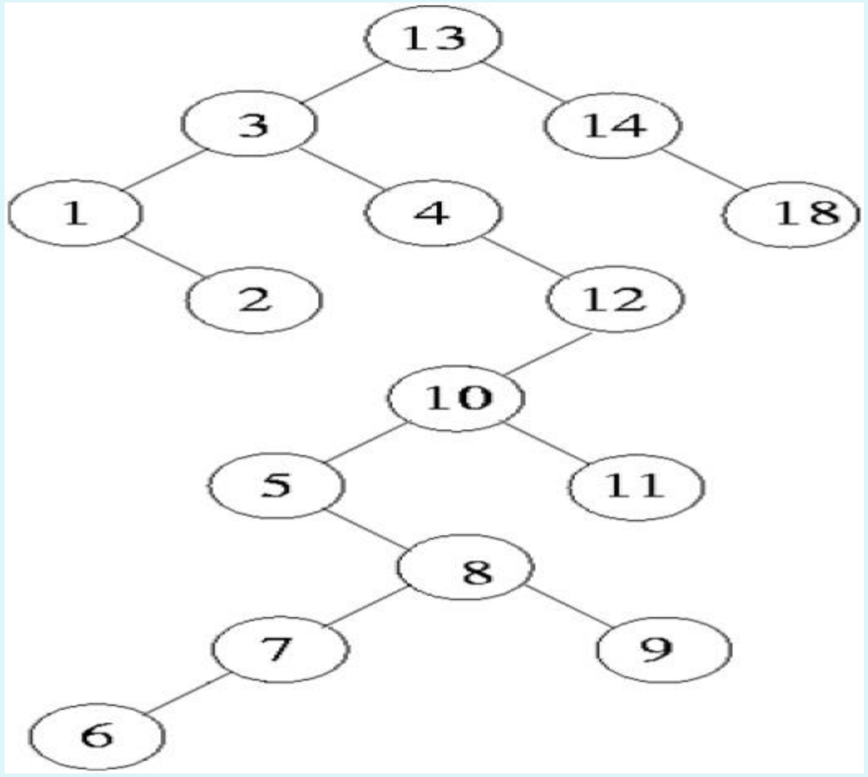
Binary Search Tree:

- Suppose T is a binary tree, the T is called a binary search tree or binary
- sorted tree if each node N of T has the following property:
    - The values of at N (node) is greater than every value in the left sub tree of
    - N and is less than every value in the right sub tree of N.

- Binary Search Tree using these values: (50, 30, 55, 25, 10, 35, 31,37, 20, 53, 60, 62)



Following figure shows a binary search tree. Notice that this tree is obtained by inserting the values 13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18 in that order, starting from an empty tree.



sorting

- **Sorting: Note that inorder traversal of a binary search tree always gives a sorted**
- sequence of the values. This is a direct consequence of the BST property.
- This provides a way of sorting a given sequence of keys: first, create a BST with these keys and then do an inorder traversal of the BST so created.
- **Inorder Travers (LNR) : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 18**

Search

- Search: is straightforward in a BST. Start with the root and keep moving left or right using the BST property. If the key we are seeking is present, this search procedure will lead us to the key. If the key is not present, we end up in a null link.

Insertion

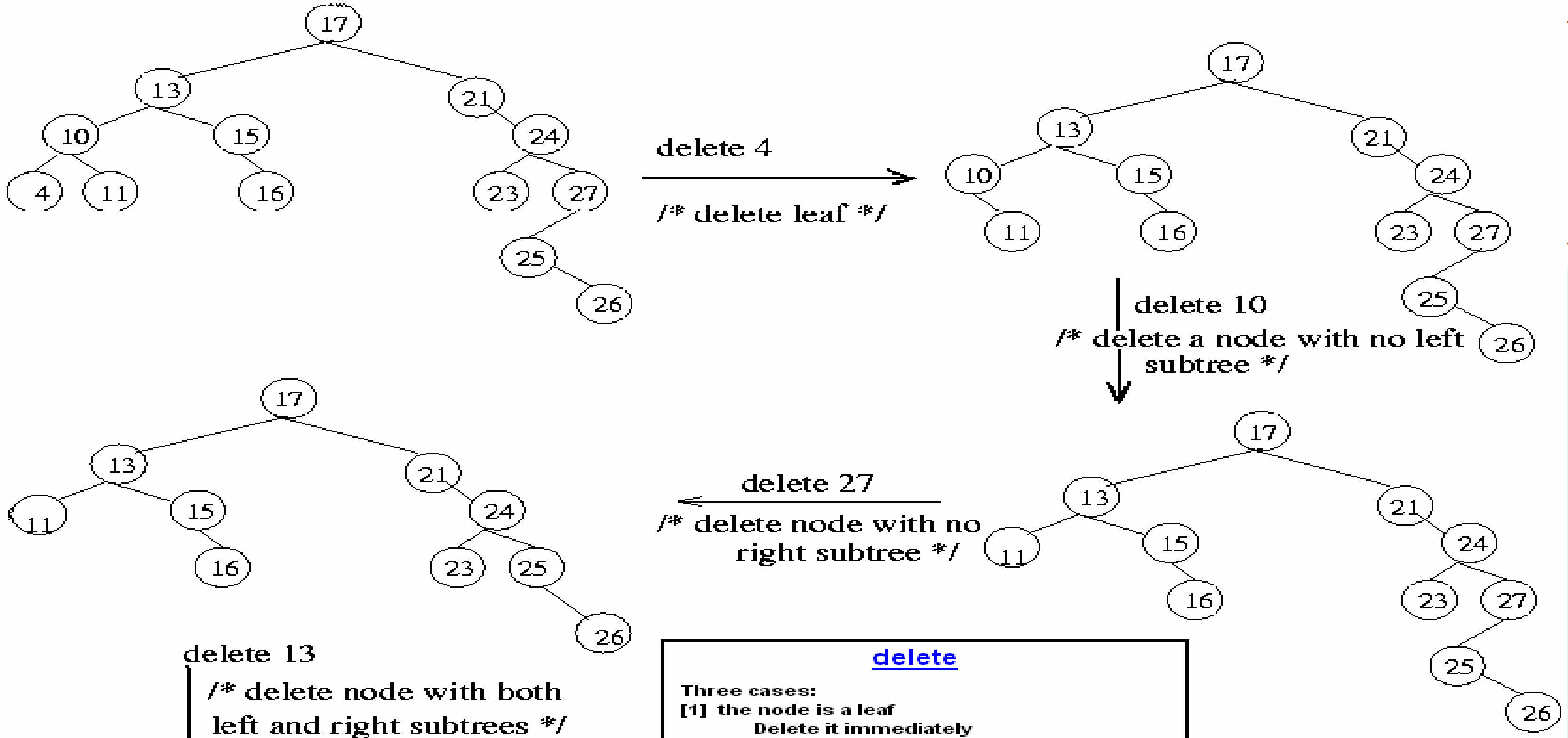
- Insertion in a BST is also a straightforward operation. If we need to insert an element n, we traverse tree starting from root by considering the above stated rules. Our traverse procedure ends in a null link. It is at this position of this null link that n will be included.

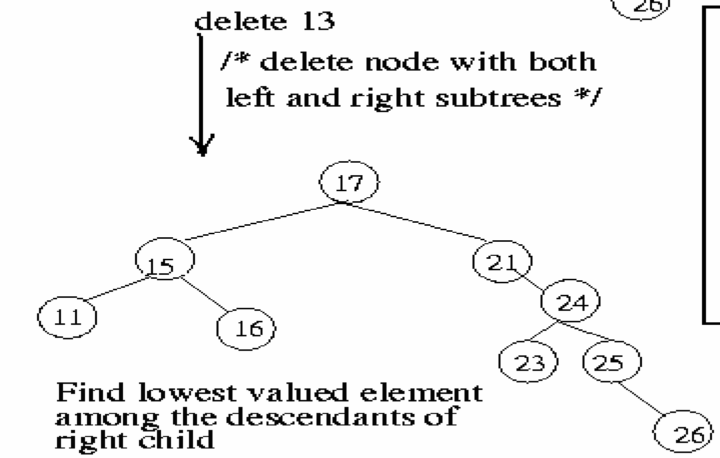
Deletion in BST

- **Deletion in BST:** Let x be a value to be deleted from the BST and let N denote the node containing the value x. Deletion of an element in a BST again uses the BST property in a critical way. When we delete the node N containing x, it would create a "gap" that should be filled by a suitable existing node of the BST. There are two possible candidate nodes that can fill this gap, in a way that the BST property is not violated:

- 1) Node containing highest valued element among all descendants of left child of N.
- 2)Node containing the lowest valued element among all the descendants of the right child of N. There are three possible cases to consider:
  - **Deleting a leaf (node with no children): Deleting a leaf is easy, as we can** simply remove it from the tree.
  - **Deleting a node with one child: Delete it and replace it with its child.**
  - **Deleting a node with two children: Call the node to be deleted "N".** Do not delete N. Instead, choose its in-order successor node "S". Replace the value of “N” with the value of “S”. (Note: S itself has up to one child.)

- As with all binary trees, a node's in-order successor is the left-most child of its right subtree. This node will have zero or one child. Delete it according to one of the two simpler cases above.


delete 4
/* delete leaf */
delete 10
/* delete a node with no left subtree */
delete 27
/* delete node with no right subtree */



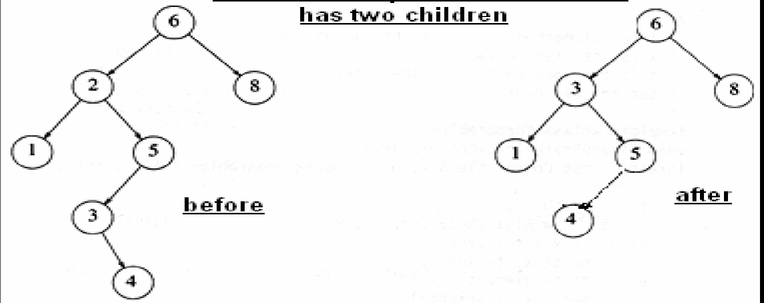
delete

Three cases:
[1] the node is a leaf
Delete it immediately
[2] the node has one child
Adjust a pointer from the parent to bypass that node
[3] the node has 2 children
replace the value of that node with the minimum element at the right subtree.
delete the minimum element Has either no child or one child.
So invoke case 1 or 2

Find lowest valued element among the descendants of right child

OR Find the in-order Successor and replace it with node to be deleted

Second example to delete node has two children



Deletion of a node (2) with two children, before and after

Graph data structure

DATA STRUCTURE

2ND STAGE

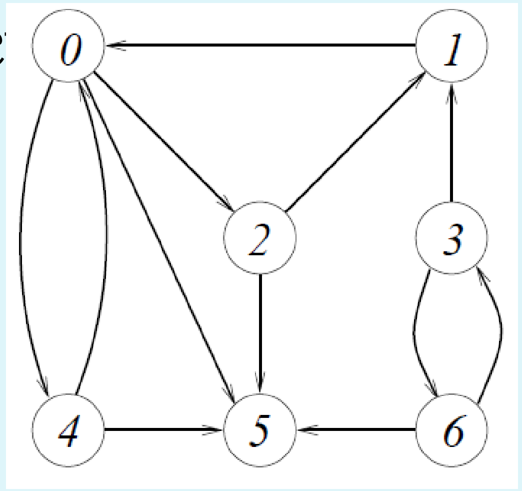
Directed and Undirected Graphs

A *graph is a mathematical structure consisting of a set of vertices and a set of edges connecting the vertices. Formally, we view the edges as pairs of vertices; an* edge (v, w) connects vertex v with vertex w. We write G = (V,E) to denote that G is a graph with vertex set V and edge set E.

A graph G = (V,E) is *undirected if for all* vertices v,w є V we have (v,w) є E if, and only if, (w, v) є E, that is, if all edges go both ways. If we want to emphasise that the edges have a direction, we call a graph *directed. For brevity, a directed graph is often called a digraph, and an undirected* graph is simply called a graph. We shall not use this convention here; for us `graph' always means `directed or undirected graph'.

When drawing graphs, we represent a vertex by a point or circle containing the name of the vertex, and an edge by an arrow connecting two vertices. When drawing undirected graphs, instead of drawing two arrows (one in each direction) between all vertices, we just draw one line connecting the vertices.

- Figure shows a drawing of the (direc
  graph G = (V,E) with



vertex set

V ={ 0,1,2, 3, 4, 5, 6}

and edge set

E = {(0, 2), (0, 4), (0, 5), (1; 0), (2,1), (2, 5), (3, 1), (3, 6), (4, 0), (4, 5), (6, 3), (6, 5)}

Use of graph

- Graphs are a useful mathematical model for numerous .real life. problems and structures. Here are a few examples:
- *Airline route maps:*

Vertices represent airports, and there is an edge from vertex A to vertex B if there is a direct flight from the airport represented by A to the airport represented by B.

- *Road Maps.*

Edges represent streets and vertices represent crossings.

- *Electrical Circuits.*

Vertices represent diodes, transistors, capacitors, switches, etc., and edges represent wires connecting them.

- *Computer Networks*.

  Vertices represent computers and edges represent network connections (cables) between them.

- *The World Wide Web*.

Vertices represent webpages, and edges represent hyperlinks.

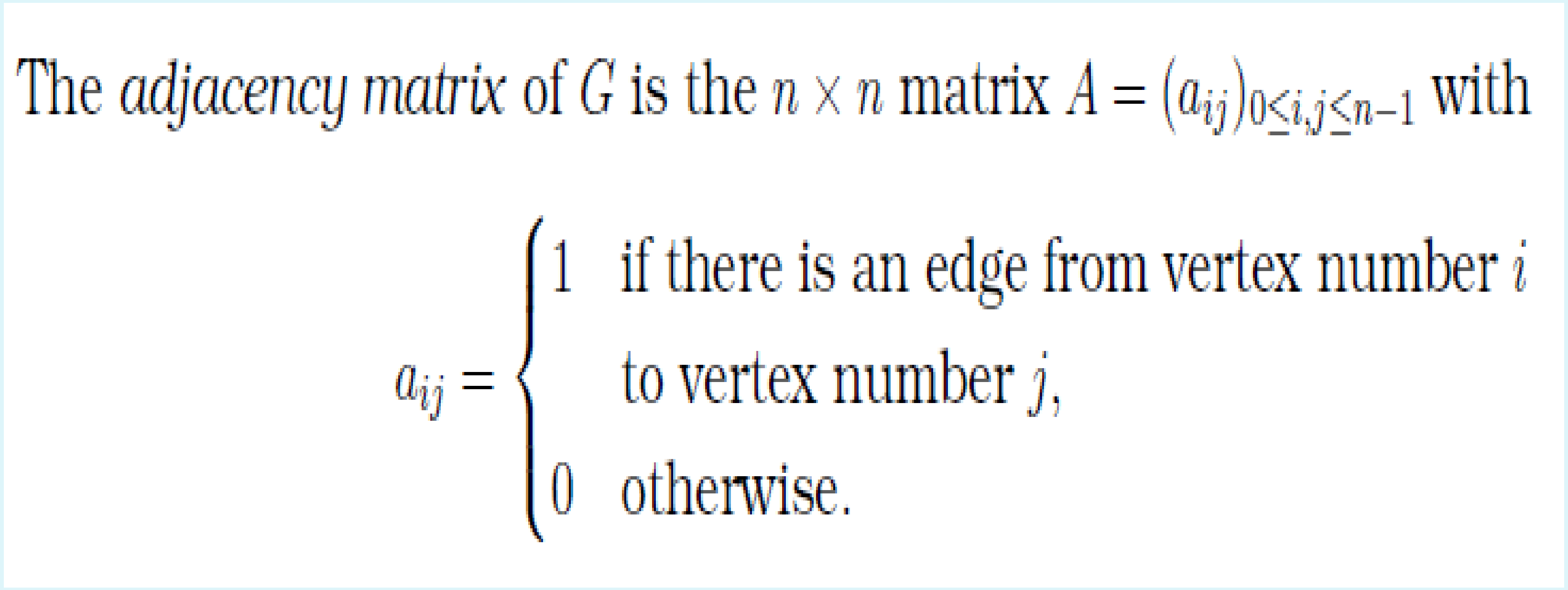
- *Flowcharts.*

A flowchart illustrates the flow of control in a procedure. Essentially, a flowchart consists of boxes containing statements of the procedure and arrows connecting the boxes to describe the flow of control. In a graph representing a flowchart, the vertices represent the boxes and the edges represent the arrows.

Data structures for graphs

- Let G = (V,E) be a graph with n vertices. We assume that the vertices of G are numbered 0, …., n - 1 in some arbitrary manner.

The adjacency matrix data structure

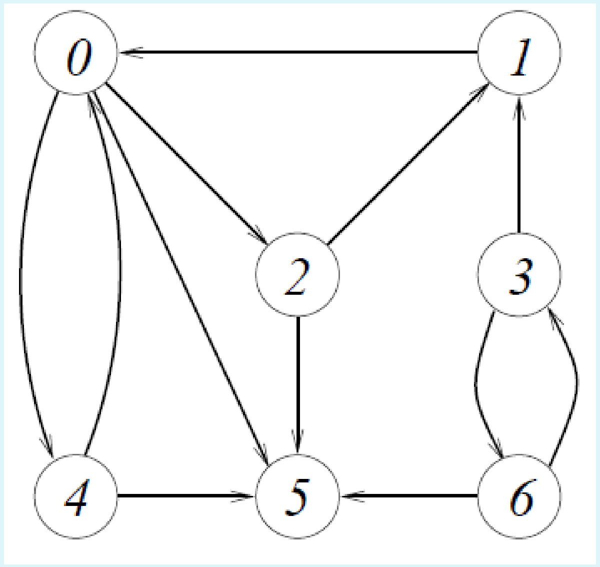


The *adjacency matrix* of G is the $n \times n$ matrix $A = (a_{ij})_{0 \le i,j \le n-1}$ with

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from vertex number } i \\ & \text{to vertex number } j, \\ 0 & \text{otherwise.} \end{cases}$$

- For example, the adjacency matrix for the graph in Figure

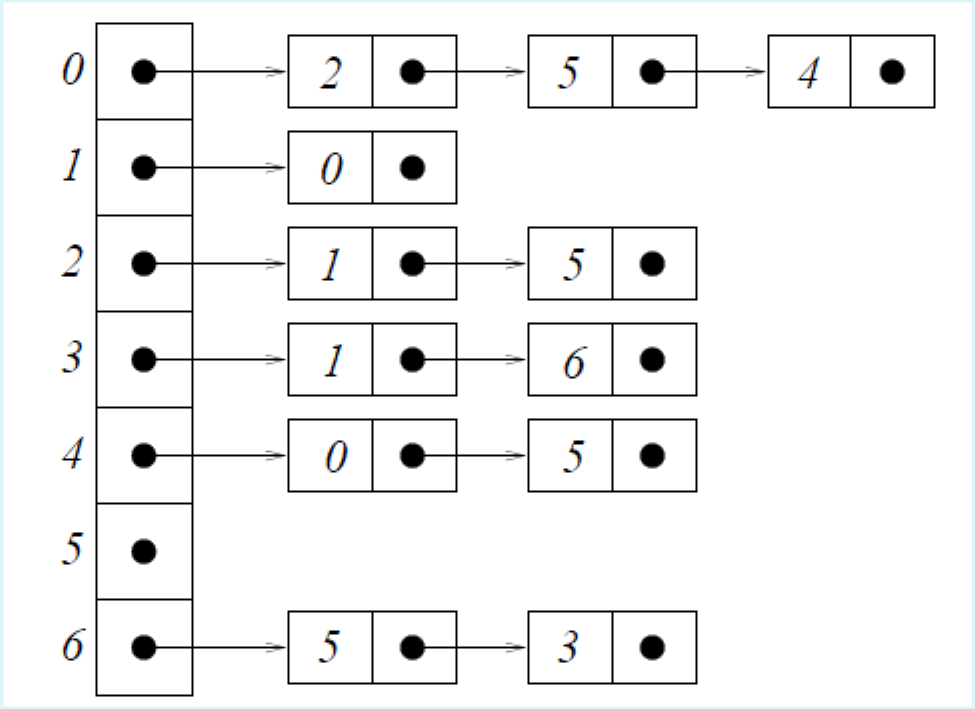
$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

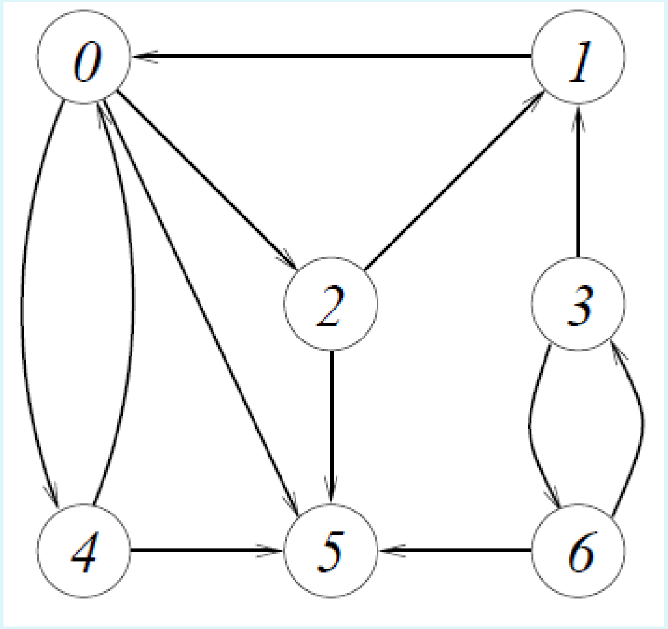


The adjacency list data structure

- The *adjacency list representation of a graph G with n vertices consists of an array vertices with n entries, one for each vertex. The entry for vertex v is a list of all* vertices w such there is an edge from v to w. We make assumptions on the order in which the vertices adjacent to a vertex v appear in the adjacency list, and our algorithms should work for any order.

- shows an **adjacency list representation of the graph in Figure**





Traversing Graphs

- Most algorithms for solving problems on graphs examine or process each vertex and each edge of the graph in some particular order. The skeleton of such an algorithm will be a *traversal of the graph, that is, a strategy for visiting the vertices* and edges in a suitable order.

- Breadth-first search (BFS) and depth-first search (DFS) are two traversals that are particularly useful. Both start at some vertex v and then visit all vertices reachable from v (that is, all vertices w such that there is a path from v to w).

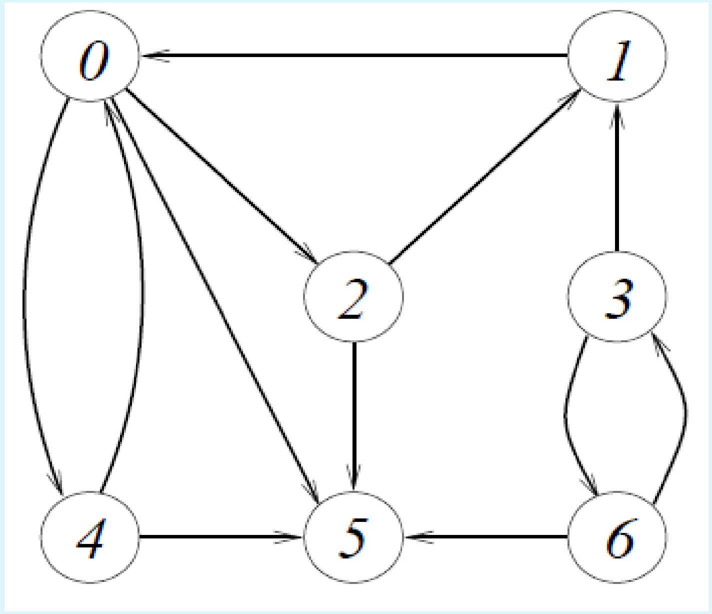
- If there are vertices that remain unvisited, that is, if there are vertices that are not reachable from v, then BFS and DFS pick a new vertex v and visit all vertices reachable from v. They repeat this process until they have finally visited all vertices.

Breadth-first search

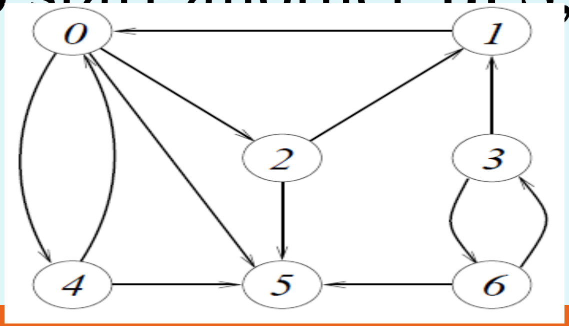
- A BFS starting at a vertex v first visits v, then it visits all neighbours of v (i.e. ,all vertices w such that there is an edge from v to w), then all neighbors of the neighbors that have not been visited before, then all neighbors of the neighbors of the neighbors that have not been visited before, et cetera.

- For example, a BFS of the graph in Figure shown bellow starting at vertex 0 would visit the vertices in the following order:

0, 2, 5, 4, 1



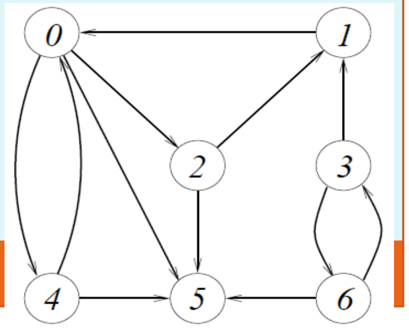
- It first visits 0, then the neighbours 2; 5; 4 of 0. Next are the neighbours of 2, which are 1 and 5. Since 5 has been visited before, only 1 is added to the list. All neighbours of 5, 4, and 1 have already been visited, so we have found all vertices that are reachable from 0. Note that there are other orders in which a BFS starting at 0 may visit the vertices of the graph, because the neighbours of 0 may be visited in a different order. An example is 0; 5; 4; 2; 1. The vertices 3 and 6 are not reachable from 0, so we have to start another BFS, say at 3. It first visits 3 and then 6.



- It is important to realize that the traversal heavily depends on the vertex we start at. For example, if we start a BFS at vertex 6 it will visit all vertices in one sweep, maybe in the following order:

6, 5, 3, 1, 0, 2, 4

Other possible orders are 6, 3, 5, 1, 0, 2, 4 and 6, 5, 3, 1, 0, 4, 2 and 6, 3, 5, 1, 0, 4, 2.



Algorithm BFS

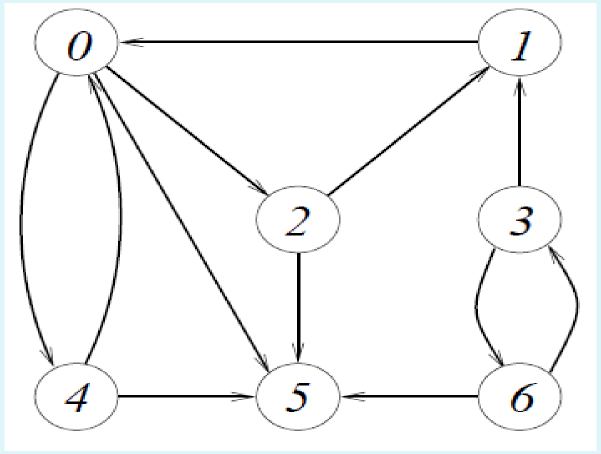
```
**Algorithm** bfsFromVertex($G, v$)

1. $visited[v] = \text{TRUE}$
2. $Q$.enqueue($v$)
3. **while not** $Q$.isEmpty() **do**
4.         $v \leftarrow Q$.dequeue()
5.         **for all** $w$ adjacent to $v$ **do**
6.                 **if** $visited[w] = \text{FALSE}$ **then**
7.                         $visited[w] = \text{TRUE}$
8.                         $Q$.enqueue($w$)
```

Depth-first search

- A DFS starting at a vertex v first visits v, then some neighbour w of v, then some neighbour x of w that has not been visited before, et cetera. Once it gets stuck, the DFS backtracks until it finds the first vertex that still has a neighbour that has not been visited before. It continues with this neighbour until it has to backtrack again.
- Eventually, it will visit all vertices reachable from v. Then a new DFS is started at some vertex that is not reachable from v, until all vertices have been visited.

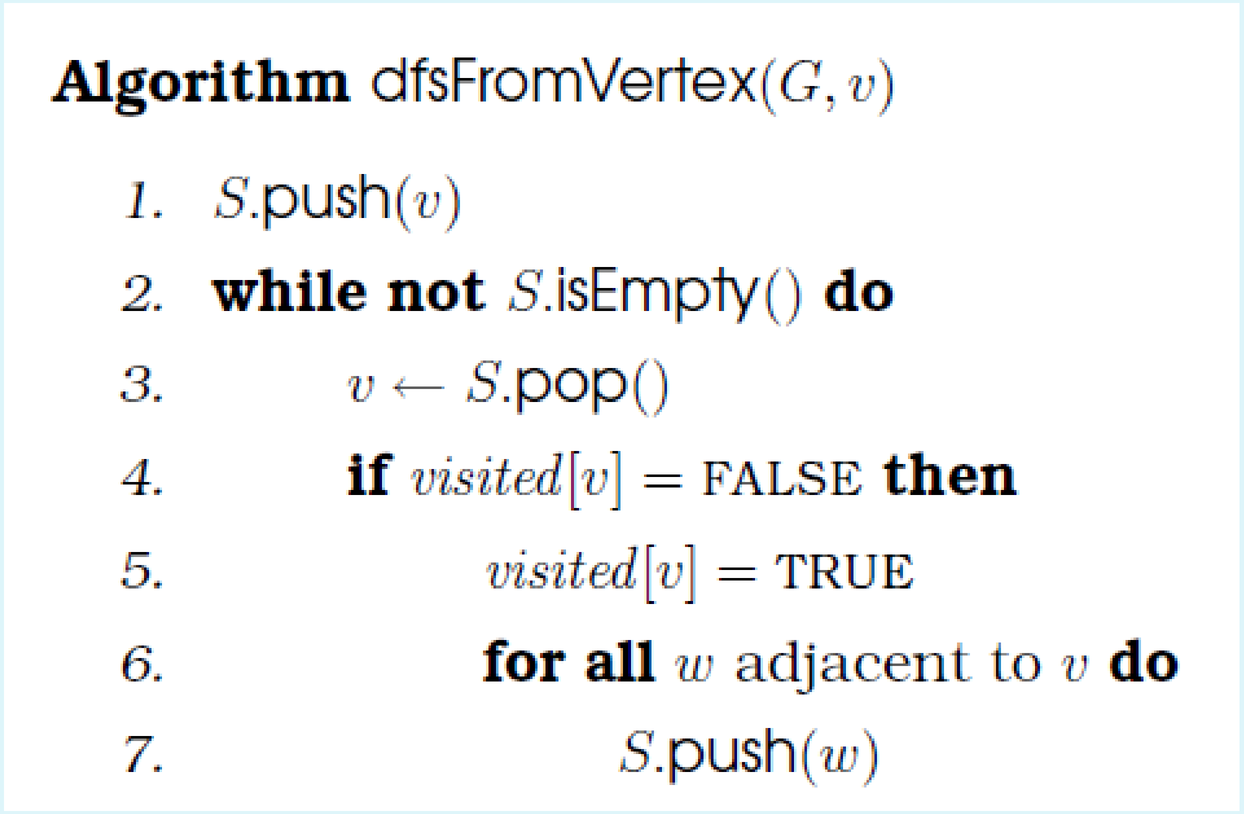
- a DFS in the graph of Figure shown starting at 0 may visit the vertices in the order

  0, 2, 1, 5, 4



- After it has visited 0; 2; 1 the DFS backtracks to 2, visits 5, then backtracks to 0, and visits 4. A DFS starting at 0 might also visit the vertices in the order 0; 4; 5; 2; 1 or 0; 5; 4; 2; 1 or 0; 2; 5; 1; 4. As for BFS, this depends on the order in which the neighbours of a vertex are processed.

Algorithm dfsFromVertex(G, v)



```
**Algorithm** dfsFromVertex($(G, v)$)

1. $S$.push($(v)$)
2. **while not** $S$.isEmpty() **do**
3.         $v \leftarrow S$.pop()
4.         **if** $visited[v] = \text{FALSE}$ **then**
5.                 $visited[v] = \text{TRUE}$
6.                 **for all** $w$ adjacent to $v$ **do**
7.                         $S$.push($(w)$)
```