Lecture Notes

Data Structure

Contents

Chapter 1

Introduction to Data Structure

We study data structures to learn to write more efficient programs. But what is the point of programs being efficient when new computers are faster day by day? Because the more we are capable of excellence, the more our ambition grows. And to tackle that, we need to learn to represent and operate data more efficiently. By studying data structures, we will be able to store and use data more efficiently. In this chapter, we are going to explore the meaning of *data structure* as well as learn what different types of data structures there are.

1.1 Data Structure

To know the meaning of *data structure*, we first have to know what *data* constitutes. Simply put, *data* means raw facts or information that can be processed to get results. There are many variants of *data* to work with. For example, your *age* is one kind of data. Then again, so is your name. We can also say, this whole chapter is a form of data. From this, you can gather the fact that *data* is not only numeric. It takes all kinds of forms. Which brings us to its *structure*.

So, what do we mean by *structure*? Some elementary items constitute a unit and that unit may be considered as a structure. A structure may be treated as a frame where we organize some elementary items in different ways. Everything around us in our reality has a structure. Starting from the teeny-tiny atoms, to something as large as Mount Everest, everything has a structure that is built by the virtue of some other smaller structures, all the way down to some elementary items that constitute them.

Now that we have learned what *data* and *structure* mean separately, we can understand what *data structure* means. A data structure is a structure where we organize elementary data items in different ways and there exits a structural relationship among the items so that it can be used efficiently. In other words, a data structure is a means of structural relationships of elementary data items for storing and retrieving data in the computer's memory. To be more general, a data structure is any data representation and its associated operations. For example, an integer or floating-point number stored on the computer can be considered as a simple data structure. The term *data structure* is used more commonly to mean an organization or structuring for a collection of data items. A sorted list of integers stored in an array is an example of such a structuring.

1.2 Elements of Data Structure

Usually, elementary data items are the *elements* of a data structure. In a programming language, generally, the types of different elementary data items are Character, Integer, Float, etc. However, a data structure may be an element of another data structure. That means a data structure may contain another data structure. For example Array, Structure, Stack, etc. In computer science, we talk about or study data structure in two ways: Basic, Abstract Data Types (ADTs).

Basic data structures are those who have concrete implementation built into the programming language. For example, Variable, Pointer, Array, etc. When we use a variable, we do not need to "build" or "create" a variable from scratch. We just declare it and use it. The same goes for Pointer and Array. We do not need to "create" them because the fundamental of what they are, is already concretely built or implemented in respective programming languages.

ADTs are entities that are the definition of data and operation but do not have any concrete implementation. Example: List, Stack, Queue, etc. An ADT is an abstract idea that we have in our mind for a data structure. We do not have the implementation for it in the programming language built-in. For solving certain problems, we may need to think of such data structures that will be more efficient than the already built-in basic data structures in a programming language. To use those abstract data structures that we have in our minds, we need to build or create them by ourselves in the programming language.

1.3 Operations of Data Structures

Data structures have their operations to manipulate and use data. Without the operations, data structures are only good for storing data but not using them. Therefore, different operations of a data structure play a vital role in it to be useful.

There are some basic operations that all the data structures should have. Those operations are:

- Insertion (addition of a new element in the data structure)
- Deletion (removal of the element from the data structure)
- Traversal (accessing data elements in the data structure)

Most of the times the above-mentioned basic operations do not suffice when it comes to more advanced usage of data for different data structures. In such scenarios, we need to depend on additional operations. Some of those includes:

- Searching (locating a certain element in the data structure)
- Sorting (Arranging elements in a data structure in a specified order)
- Merging (combining elements of two similar data structures)

1.4 Algorithm

So far we have learned about what it means to be a data structure and several kinds of it. We also learned about their operations and such. Now, everything is well and fine but it would not make much of a sense to have all these data structures that we cannot use if you do not know what a program is. But before we dive into knowing that, we will first visit another term called an *algorithm*.

An algorithm simply means a set of instructions that can be followed to perform a task. In other words, the sequence of steps that can be followed to solve a problem. To solve a problem in a programming language, we first deduce an algorithm about it. To write an algorithm we do not strictly follow the grammar of any particular programming language. However, its language may be near to a programming language. Every algorithm can be divided into three sections:

- The first section is the *input* section, where we show which data elements are to be given or fed to the algorithm as an input.
- The second section is the most important one, which is the *operational* or *processing* section. Here we have to do all necessary operations, such as computation, taking a decision, calling other procedures (or algorithms), etc.
- The third section is *output*, where we display or get the result with the help of the previous two sections.

1.5 Program

Now that we know about algorithms, it is much easier for us to learn what a *program* constitutes. It is basically, the sequence of instructions of any programming language that can be followed to perform a task. This definition is quite similar to the one of an *algorithm*. The key difference is, now we are limiting the solution of a problem to be devised using a programming language. That means, we now have to follow a certain programming language's syntax and semantics. Otherwise, we cannot call it a program.

Like an algorithm, a program also has three sections: input, processing, and output. In a program usually, we use a large amount of data. Most of the cases these data are not only elementary items, but there also exists a structural relationship among them. That means a program uses *data structures*.

1.6 Exercises

Answer the following questions:

1. What is a *data structure*? Explain with examples.
2. What is Abstract Data Types (ADTs)? Explain with examples.
3. What do you mean by the terms *algorithm* and *program*? What are their differences? Explain with appropriate examples.

1.7 References

- "Data Structures and Algorithm Analysis", Edition 3.2 (C++ Version), Clifford A. Shaffer.
- https://en.wikipedia.org/wiki/Data_structure

Chapter 2

Array & String

In any programming language, we need a concept of list to store data items sequentially. *Array* is one of the most common and simple data structures available in C++ to do that. In this chapter, we are going to start by knowing how basic data is stored in memory with the concept of variables. Then we will explore different kinds of arrays with appropriate examples that illustrate several key behaviors of arrays. Finally, we will finish off with strings and their use in C++.

2.1 Data Storage Concept

Before we dive into learning the data structures used in any programming language, we need to know how data is stored in the memory of the computer. By having a clear understanding of that, we will be ready to venture further into the world of data structures seamlessly. Let us now start that journey by learning firstly, about variables.

2.1.1 Variables

It is well known that computers represent information using the binary number system. But we need to understand how we are going to represent information in our programming, which we will use as the basis for our computational processes. To do this, we need to decide on representations for numeric values and symbolic ones. We will use the concept of variables.

Let us think that, you are asked to retain the number 5 in your mental memory, and then you are asked to memorize also the number 2 at the same time. You have just stored two different values (5, 2) in your mental memory. Now, if you are asked to add 1 to the first number very first number, you should be retaining the numbers 6 (that is 5+1) and 2 in your memory. Now subtract the second value from the first and you obtain 4 as a result. The whole process that you have just done with your mental memory is a simile of what a computer can do with two variables. The same process can be expressed in C++ with the following instruction set:

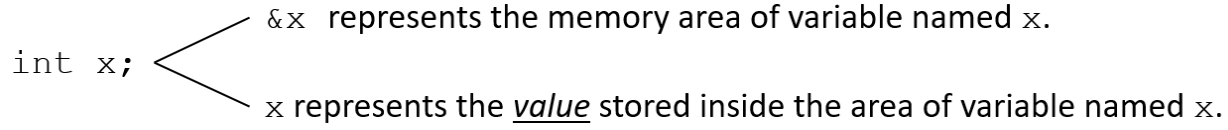
```
1. a = 5
2. b = 2
3. a = a + 1
4. result = a – b
```

This is a very simple example since we have only used two small integer values, but consider that your computer can store millions of numbers like these at the same time and conduct sophisticated mathematical operations with them. Therefore, we can define a variable as a portion of memory to store a determined value. Each variable needs an identifier that distinguishes it from the others. For example, in the previous code, the variable identifiers were *a*, *b* and *result*, but we could have called the variables any names we wanted to, as long as they were valid identifiers.

So, a variable is a storage location and an associated symbolic name (an identifier) which contains some known or unknown quantity or information, a value. The variable name is the usual way to reference the stored value; this separation of name and content allows the name to be used independently of the exact information it represents. The identifier in computer source code can be bound to a value during run time, and the value of the variable may thus change during program execution. In computing, a variable may be employed in a repetitive process: assigned a value in one place, then used elsewhere, then reassigned a new value and used again in the same way. Compilers have to replace variables' symbolic names with the actual locations of the data in the memory.

2.1.2 Memory Management

A *variable* is an area of *memory* that has been given a name. For example, if we look at the declaration `int x;`, it acquires four bytes of memory area and this area of memory that has been given the name `x`. One can use the name to specify where to store data. For example, `x=10;` is an instruction to store the data value `10` in the area of memory named `x`. The computer accesses its own memory not by using variable names but by using a memory map with each location of memory uniquely defined by a number, called the *address* of that memory location. To access the memory of a variable the program uses the `&` operator. For Example, `&x` returns the address of the variable `x`.



2.2 One-Dimensional Array

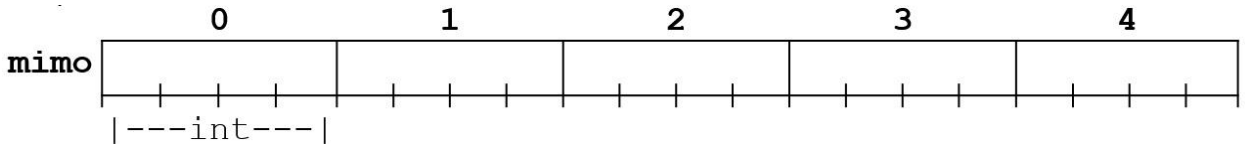
So far we have seen how a variable works and how the memory of the computer is associated with the use of a variable. We know that a variable can hold only one value of one type. A *variable* that contains one value of one type is a simple data structure. However, a lot of the time we are going to need to store a lot of values at the same time in a single variable. To do that, we need to know about another data structure called *Array*. There are mainly two kinds of arrays: One-Dimensional (1D), Multi-Dimensional (2D, 3D, 4D, etc.). We are going to be focusing on learning about 1D & 2D arrays in this chapter.

2.2.1 Definition & Structure

Let us assume you are asked to write a program where you need to store the CGPA of 1 student and output it. What approach will you take? Firstly, you will declare a float variable. Then take the CGPA of that 1 student as input and save it in that float variable. Then you are just going to use the print command in the programming language to print the variable's value. Pretty simple, right? Now, what if you are asked to store the CGPA of 200 students instead of only 1? One approach would be to declare 200 float variables and take 200 inputs separately for those 20 variables and write 200 print commands to print those 200 variables. As you can understand, this will mean the source code for this simple program will be huge and quite cumbersome. This is

neither a good nor an efficient coding practice. This simple program can be written more shortly with the use of the right data structure. In this case, that data structure should be an array.

An array can hold a *series of elements* of the *same type* placed in contiguous memory locations. Each of these elements can be individually referenced by using an index to a unique identifier. In other words, arrays are a convenient way of grouping a lot of values of the same type under a single variable name. For example, an array to contain `5` integer values of type `int` called `mimo` could be represented like this:



where each blank panel represents an element of the array, that, in this case, are integer values of type `int` with size `4` bytes. These elements are numbered/indexed from `0` to `4` since in arrays the first index is always `0`, independently of its length.

2.2.2 Declaration

Now that we know the structure of an array, we need to know how to use it in a programming language. Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

```
type name [total_number_of_elements];
```

where `type` is a valid data type (like `int`, `float` ...), `name` is a valid identifier and the elements field (which is always enclosed in square brackets `[]`), specifies how many of these elements the array can contain. Therefore, to declare an array called `mimo` as the one shown in the above diagram it is as simple as:

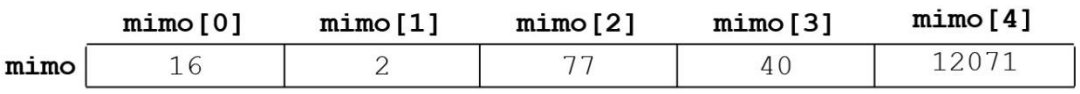
```
int mimo [5];
```

The elements field within brackets `[]` which represents the number of elements the array is going to hold, must be a *constant integer value* since arrays are blocks of non-dynamic memory whose size must be determined before execution.

2.2.3 Initialization

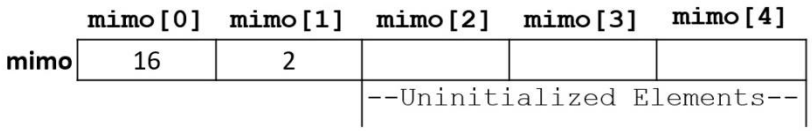
When declaring a regular array (`int mimo[5];`) of local scope (within a function, for example) its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global and static arrays are automatically initialized with their default values, zeros.

When we declare an array, we can assign initial values to each one of its elements by enclosing the values in braces `{ }` separated by commas. For example, `int mimo[5] = { 16, 2, 77, 40, 12071 };`. This declaration would have created an array like this:



The number of values between braces `{ }` must not be larger than the number of elements that we declare for the array between square brackets `[ ]`.

An array can also be partially initialized. i.e. we assign values to some of the initial elements. For example, `int mimo[5] = { 16, 2};`. This declaration would have created an array like this:



Here the first 2 values are assigned sequentially. The rest 3 elements are unassigned. Some more initializations:

```
float x[5] = {5.6, 5.7, 5.8, 5.9, 6.1};
char  vowel[6] = {'a', 'e', 'i', 'o', 'u', '\0'};
```
is equivalent to string declaration: `char vowel[6] = "aeiou";`

2.2.4 Access

It is not helpful if after declaring and initializing an array, we cannot use the values inside it for the rest of the program. To use them, we first need to have access to them. Let us now take a look at how we can do that in this section of the chapter.

The number in the square brackets `[ ]` of the array is referred to as the *'index' (plural: indices) or 'subscript'* of the array and it must be an integer number `0` to *one less than the declared number of elements*. We can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. The format is as simple as: `name[index]`. For the declaration `int mimo[5];` the five (`5`) elements in `mimo` is referred in the program by writing: `mimo[0]`, `mimo[1]`, `mimo[2]`, `mimo[3]`, `mimo[4]`. Each of the array elements of `mimo` is an integer and each of them also acts as an integer variable. That is, whatever operation we can perform with an integer variable we can do the same with these array elements using the same set of rules.

Arrays have a natural partner in programs: the `for` loop. The `for` loop provides a simple way of counting through the numbers of an index in a controlled way. The `for` loop can be used to work on an array sequentially at any time during a program.

It is important to be able to clearly distinguish between the two uses that brackets `[]` have related to arrays.

- one is to specify the size of arrays when they are declared - `char array[5];` Here the index (`5`) is always an integer constant.

- the second one is to specify indices for concrete array elements, like: `array[3] = '*';` Here the index (`3`) is always an integer or an integer expression.

Consider the following example:

```
char array[5];
array[7] = '*';
```

The assignment of `array[7]` is wrong as index `7` or element `8` in `array[5]` does not exist. But, C++ would happily try to write the character `*` at index `7`. Unfortunately, this would probably be written in the memory taken up by some other variable or perhaps even by the operating system. The result would be one of the followings:

- The value in the incorrect memory location would be corrupted with unpredictable consequences.
- The value would corrupt the memory and crash the program completely! On Unix systems, this leads to a memory segmentation *fault*.

The second of these tend to be the result of operating systems with no proper memory protection. Writing over the bounds of an array is a common source of error. Remember that the array limits run from zero to the size of the array minus one.

Below is an access demonstration example of an array in C++:

```
1.  int mimo[5], a, b, i; // declaration of a new array
2.  mimo[2] = 75;         // store 75 in the third element of mimo
3.  a = mimo[2];          // copy/assign a with the third value of mimo
4.  cin >> mimo[2]); // read a value for the third element of mimo
5.  /* read 5 values for the five elements of mimo sequentially */
6.  for(i=0; i<5; i++)
7.      cin >> mimo[i];
8.  /* print 5 values for the five elements of mimo sequentially */
9.  for(i=0; i<5; i++)     ;
10.     cin >> mimo[i];
11. /* some more interesting accesses */
12. a = 4;
13. mimo[2] = a;
14. mimo[a] = 3;
15. b = mimo[a-2] + 2;   //use of expression in index
16. mimo[mimo[a]] = mimo[2] + b;
```

2.2.5 Linear Search

Up to this point in the chapter, we should be quite clear about how an array works. In this section, we are going to take a look at one of the simplest searching techniques in computer science. It is called *Linear Search*.

A *Linear Search* is a technique of finding an element (or value) from a list (array) of elements. We can apply this technique to find an element within an array. For example, let us find `e` from an array called `lotsOfElements`. We begin searching `e` in `lotsOfElements` from its first value. We compare the first value with `e`. If `e` is equal to the first value, then we have found it. We do not need to search the rest of `lotsOfElements` anymore. However, if `e` is not equal to the first value of `lotsOfElements`, we then move on to its second value. Now we check `e` with the second value. If they are not equal we move to the next one. We keep moving to the next value of `lotsOfElements` and compare it to `e`. Until we find a match, we keep doing like this. When we find a match, we stop and do not search the rest of `lotsOfElements` anymore. However, if we reach the end of `lotsOfElements` and still could not find the desired value, then the value is not found because it is not present in `lotsOfElements`.

We can implement *Linear Search* in C++ like below:

```
1.   int mimo[10] = {32,4,5,12,5,54,6,23,3,5}; // declaration of a new array
2.   int n;
3.   cout<<"Enter the number to be searched: "<<endl;
4.   cin>>n; // inputting the number to be searched in the array
5.   for(int i=0; i<10; i++){ // searching begins
6.       if (n == mimo[i]){
7.         break; // searching ends
8.       }
9.   }
10.  cout<<n<<" was found in index "<<i<<" of the array."<<endl;
OUTPUT: 5 was found in index 2 of the array.
```

2.2.6 Insertion in Array

In the previous section, we have learned about the searching operation in an array in the form of *Linear Search*. One of the other important operations is *insertion* into an array. In this section, we are going to take a look at the insertion operation in three ways: inserting an element into an array from the back, front, and middle. For any kind of insertion into an array, we need to make sure the array has enough space for a new element to be inserted. An example illustrating these three kinds of insertions is given below in the form of a code snippet in C++ (with appropriate comments describing each line of code):

```
1.    int k, i, n=5, mimo[10]={2, 3, 5, 6, 7}; //partial initialization; n=total elements
2.    mimo[n++] = 8;          // insert value 8 at the end of the array; increase n;
3.    /* insert value 1 at the beginning of array */
4.    for(i=n; i>0; i--)      //shift all the values one index forward. i.e. the value
5.       mimo[i] = mimo[i-1]; //in index 1 goes to 2, 2 goes to 3,…, (n-1)th goes to nth.
6.    mimo[0] = 1; n++;       //1 is inserted at index 1; n increases;
7.    // insert value 4 in the middle (index k=3) of the array
8.    k = 3;
9.    for(i=n; i>k; i--)      //shift all the values one index forward. i.e. the value
10.      mimo[i] = mimo[i-1]; //in index k goes to k+1,…, (n-1)th goes to nth.
11.   mimo[k] = 4; n++;       //4 is inserted at index k; n increases;
12.   for(i=0; i<n; i++)      //printing all the values in the array after insert
13.      cout << mimo[i];
```

2.2.7 Deletion in Array

The *deletion* is another important operation performed in an array. Like *insertion* operation, it can also be done in three ways: deleting an element from the back, front, and middle. One thing to remember, we never actually delete anything from an array because there is no mechanism to free the space of an index of an array from the memory. Rather, we manipulate the array in such a way, so that when we print the array after the "deletion" operation, it prints the array without the "deleted" element. An example illustrating these three kinds of deletions is given below in the form of a code snippet in C++ (with appropriate comments describing each line of code):

```
1.    int k, i, n=8, mimo[10]={1, 2, 3, 4, 5, 6, 7, 8}; //n=total elements.
2.    n--; // Deleting the last element of the array. Decrease n; last element 8 is no
      longer part of list.
3.    /* delete value 1 from the beginning of array. */
4.    n--;          // deleting the value 1 will decrease the total elements n by one.
5.    for(i=0; i<n; i++)  //shift all the values one index backward. The value in index
6.       mimo[i] = mimo[i+1]; //2 goes to 1, 3 goes to 2,…, nth goes to (n-1)th.
7.    k = 2; // delete value 4 from the middle (index k=2) of the array
8.    n--;          // deleting the value 4 will decrease the total elements n by one.
9.    for(i=k; i<n; i++)      //shift all the values one index forward. i.e. the value
10.      mimo[i] = mimo[i+1]; //in index k+1 goes to k,…, nth goes to (n-1)th.
11.   for(i=0; i<n; i++)      //printing all the values in the array after insert
12.      cout<<mimo[i];
```

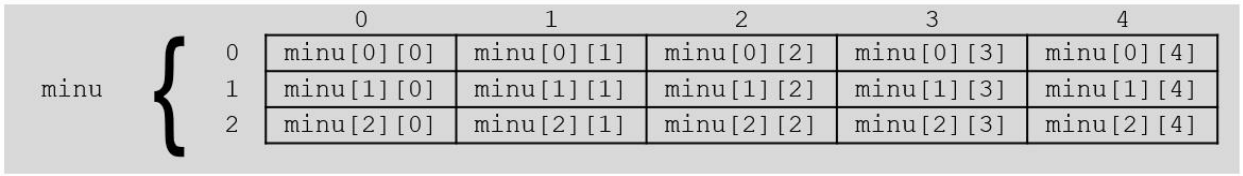
2.3 Two-Dimensional Array

We have learned about *one-dimensional (1D)* array previously and now we are going to learn about *multi-dimensional* array. To be more specific, we are going to learn about *two-dimensional (2D)* array. The basic ideas of the 1D array are still true for 2D array (sequence of elements of the same type) but the way we declare, initialize, and access a 2D array in the programming

language is different than that of a 1D array. In this section, we are going to explore the 2D array in detail.

2.3.1 Definition, Structure & Declaration

Two-dimensional (2D) arrays can be described as "arrays of arrays". For example, a 2D array can be imagined as a Two-dimensional table made of elements of the same uniform data type.



In the figure above, `minu` represents a Two-dimensional array of `3` per `5` elements of type `int`. The way to declare this array in C++ would be: `int minu [3][5];`. The way to reference the 2nd element vertically and 4th horizontally or the (2 × 4) 8th element in an expression would be: `minu [1][3];`. Generally, for two-dimensional array, 1st dimension is considered as number of rows and the 2nd dimension is considered as number of columns. Here, we have `3` rows and `5` columns.

2.3.2 Initialization

Assigning values at the time of declaring a two-dimensional array can be any one of the following ways:

```
int minu[3][5] = {1,2,3,4,5,2,4,6,8,10,3,6,9,12,15};
int minu[3][5] = {{1,2,3,4,5},{2,4,6,8,10},{3,6,9,12,15}};
int minu[3][5] =  {
  {1,2,3,4,5},
  {2,4,6,8,10},
  {3,6,9,12,15}
};
```

The internal braces are unnecessary but helps to distinguish the rows from the columns. Take care to include the semicolon at the end of the curly brace which closes the assignment. If there are not enough elements in the curly braces to account for every single element in an array, the remaining elements will be filled out with garbage/zeros. Static and global variables are always guaranteed to be initialized to zero anyway, whereas auto or local variables are guaranteed to be garbage.

2.3.3 Access

A nested loop is used to take input and give output. The input is taken in row (1st dimension) major. i.e. all the values of row `0` are scanned first, then the values of row `1`, and values of row `2`. For each row, value at column `0` is scanned first, then the value in column `1`, and value in column `2`. In the output, array a is used in row major. But array b is used in column (2nd dimension) major. i.e. all the values of column `0` are added first, then the values of column `1`,

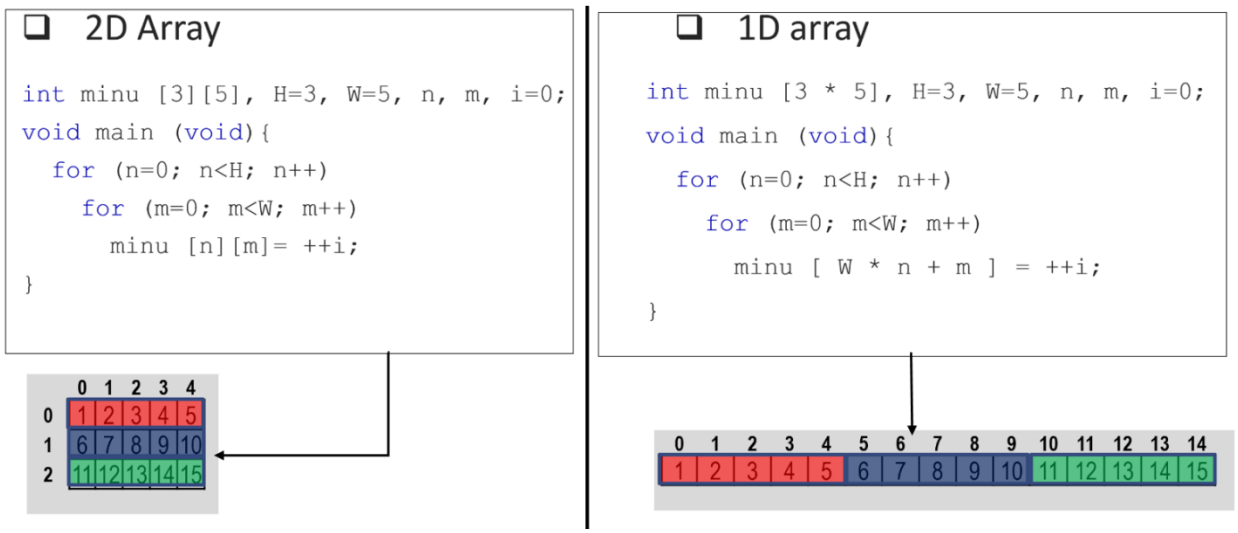
and values of column `2`. For each column, value at row `0` is added first, then the value at row `1`, and value at row `2`.

Consider the following example (the dark area at the end consists of the input and the output of this program; the yellow-colored text represents input given by the user and black colored text represents output):

```
1  // input & output of a 2D array
2  void main (void)
3  {
4    int i, j, a[3][3], b[3][3]={1,3,5,7,9,2,4,6,8};
5    for(i=0;i<3;i++)
6      for(j=0;j<3;j++)
7        cin>>a[i][j];
8    for(i=0;i<3;i++)
9      for(j=0;j<3;j++)
10       cout<<a[i][j] + b[j][i];
11 }
2 4 6 8 1 3 5 7 9
3 11 10 11 10 9 10 9 17
```

2.4 Memory Access for Both 1D & 2D Array

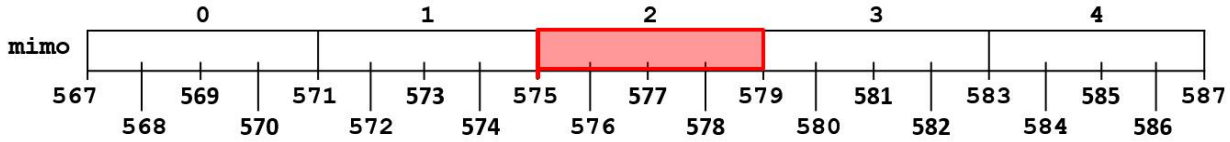
Two-dimensional arrays are just an abstraction for programmers since we can obtain the same results with a simple array just by putting a factor between its indices: `int minu [3][5];` is equivalent to `int minu [15];` (3 * 5 = 15). Below we illustrate this similarity with a code snippet:



As memory is flat, in the above code for both 1D & 2D array the values are stored sequentially in the memory (just like the 1D array). The access for the 2D array, in that case, is just as the indexing of the array,

```
[(Total_column) * (row_index) + (column_index)]
```

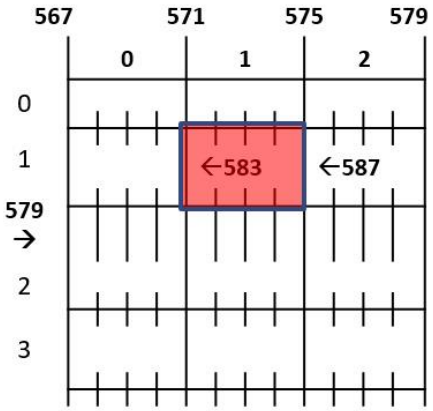
The memory of each element of an array can be accessed using the `&` operator.

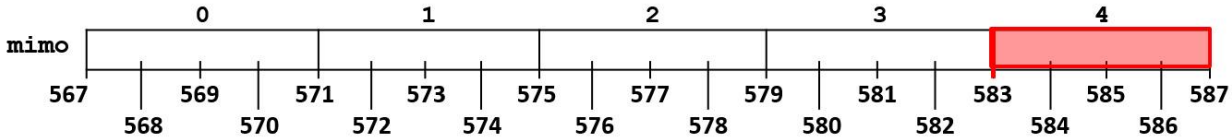


`&mimo[2]` gives the memory location of the 3rd element of the array `mimo`. If the element is more than a byte, it gives the starting byte of the element. Let us consider the starting address of `int mimo[5]` is `567`. `&mimo[2]` will give us the memory location `575`. `mimo[2]` will give us `4` bytes (`int`) of information starting from `575` to `579`. The name of an array always refers to the starting location of the array. i.e. the first element of the array. So, `mimo` and `&mimo[0]` both are the same. We can see the calculation for retrieving the memory address for `&mimo[2]` below:

```
&array[index] = start_location_array + index * size_of_data
&mimo[ 2 ] = mimo (or &mimo[0])  +  2    * sizeof(int)
&mimo[ 2 ] = 567 + 2 * 4 = 575
```

Consider a 2D array `mimo[R][C]` each element addressed by `&mimo[i][j]`, where **R** = total element in 1st dimension, **C** = total element in 2nd dimension, $0 \leq i < R, 0 \leq j < C$. Let `int mimo[4][3];`. Here, `mimo` or `&mimo[0][0]` gives us the starting memory location `567`. `mimo[1][1]` will give us `4` bytes (`int`) of information starting from `583` to `587`.





```
&array[i][j] = start_location + (i * (C * size_of_data)) + (j *
size_of_data)
&mimo[1][1] = mimo + (1 * (3 * sizeof(int))) + (1 * sizeof(int))
&mimo[1][1] = 567 + (1 * 3 * 4) + (1 * 4) = 583
```

There is a general way to access the memory location of a 2-dimensional array. For an array int mimo[R][C]; and $0 \leq i < R, 0 \leq j < C$:

- `mimo[i] = &mimo[i][0]` represents the starting address of ith row.
- `mimo[i]` skips `i` number of rows each with **C** number of elements from the `start_location` of the array.
- So, `mimo[i] = start_location + (i * C elements)`, where **C** elements are counted in bytes based on the `size_of_data`, here `int`.
- So, `mimo[i] = start_location + (i * (C * size_of_data))`.
- So, `mimo[i] = mimo(or &mimo[0][0]) + (i * (C *sizeof(int)))`.

2.5 String

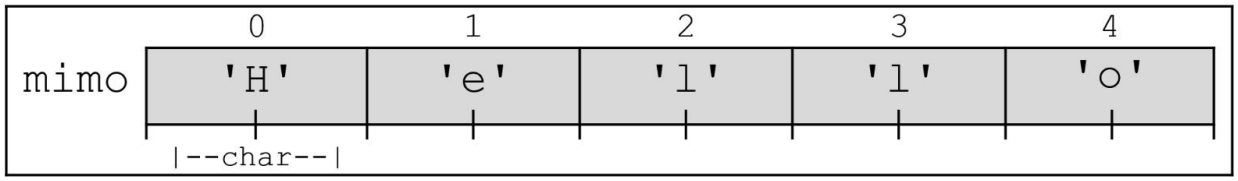
After learning arrays in detail, now it is time for us to know about one of the most important data structures in any programming language: *string*. In this section, we are going to learn about it in detail.

2.5.1 Definition & Structure

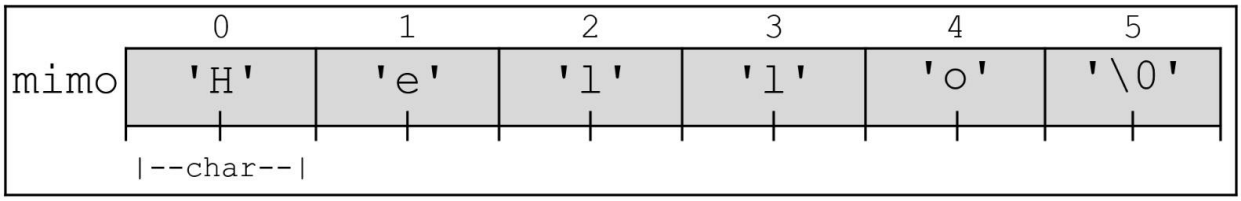
Strings are a sequence of characters representing a piece of text. In programming languages a string is represented as some characters enclosed by double quotes: `"This is a string"`. A string is mainly declared using an array of characters. The main difference between *a simple array of characters* and *an array of characters representing string* is the end marker given at the end of a string. The standard library functions can recognize this end marker as being the end of the string. The end marker is called the zero (or `NULL`) byte because it is just a byte which contains the value zero: `'\0'`. Programs rarely get to see this end marker as most functions that handle strings use it or add it automatically.

2.5.2 Declaration & Initialization

Declaration of a string is just an array of characters: `char mimo [5];`. But things are different when we initialize during declaration. An array to contain `5` character values (`'H'`, `'e'`, `'l'`, `'l'`, `'o'`) of type `char` called `mimo` could be represented like this: `char mimo [5]={ 'H', 'e', 'l', 'l', 'o'};`.



But, to represent the same text as a string in C++ double quotation (") is used to bound the text. And, a `NULL` character is added at the end of the text. So, because of this `NULL`, we need to declare an additional slot in the array. `char mimo [6]= "Hello";`. The standard library functions can recognize `NULL` (`'\0'`) as being the end of the string.



2.5.3 Access, Input, Output

Consider the following example (the dark rea at the end is the output of this program; the red-colored text represents input given by the user):

```
1  // null-terminated sequences of characters
2  void main (void)
3  {
4    char Question[]="Please, enter first name: ";
5    char Greeting[] = "Hello";
6    char FirstName[80];
7    cout<<Question;
8    cin>>FirstName;
9    cout<<Greeting<<", "<<FirstName);
10 }
```

```
Please, enter first name: John
Output: Hello, John!
```

Lines 4-6 declare three arrays of characters. The first two are initialized as strings with a `NULL` character at their end. Line 7 output the string in `Question`. The same can be found in line 8, where we have `cin` with the array `FirstName`. The address indicates the location in the memory from where the processing will start. The `NULL` character indicates where the processing will stop for `cout`. And for `cin`, after the processing (i.e, after the string input) a `NULL` character is automatically added at the end of the string.

Instead of asking for the first name only, if we would have asked for the whole name, this program might not work properly. That is if we would have given input `John Rambo` instead of only `John` the output will be as follows –

```
Please, enter your first name: John Rambo
Hello, John!
```

Still, it shows `John` after `Hello,` not `John Rambo`. As we know, that `cin` always stops at `white spaces` during taking input. So, after taking `John` as input `cin` receives a `space` indicating the end of input. So it never even goes for `Rambo`. To overcome this, there is another function `cin.get()` which takes only `2` parameters: *variable name of the string* and *its size* (maximum size of the character array). So, just writing `cin.get(FirstName,80);` instead of `cout<<Question;` will give the following output –

```
Please, enter your first name: John Rambo
Hello, John Rambo!
```

2.5.4 String Handling Functions

Strings are often needed to be manipulated by a programmer according to the need of a problem. All string manipulation can be done manually by the programmer but, this makes programming complex and large. C++ supports a large number of string handling functions. There are numerous functions defined in the "`cstring`" header file (library). For all such library functions, a `NULL` character is assumed to be at the end of the existing string and a `NULL` character is always included at the end of the new/updated/changed string. Few commonly used string handling functions are discussed below:

Function	Work Of Function
`strlen(str)`	Calculates the length of string `str`; the total number of characters from the given starting address by `str` until a `NULL` encounters.
`strcpy(s1, s2)`	Copies a string `s2` to another string `s1`; all the characters starting from the given starting address by `s2` until a `NULL` encounters, will be copied in sequentially from the given address by `s1`.
`strcat(s1, s2)`	Concatenates (joins) two strings; all the characters starting from the given starting address by `s2` until a `NULL` encounters, will be copied in sequentially from the `NULL` character at the end of `s1`.
`strcmp(s1, s2)`	Compares two strings; if `s1` and `s2` are equal `strcmp` returns `0`, if `s1` is alphabetically (compares `ASCII` value) lower than `s2` then `strcmp` returns `<0` and otherwise returns `>0`;

2.5.5 String as Object

In C++, there is another way to store a string in a variable, by using a string object. Unlike character arrays, there is no fixed length for string objects. It can be used as per the requirement

of the programmer. Unlike a string that is a character array, a string object does not have a `NULL` character at the end of it.

Instead of using `cin` or `cin.get()` functions for accepting strings with `white spaces` as input, we can use `getline()` function. The `getline()` function takes the input stream as the first parameter which is cin and string object as the second parameter.

The "`string`" header file contains library functions that can be used on string objects for manipulation. [See *Reference* at the end of the chapter for more info on that]. To use library functions of "`cstring`" on a string object, it needs to be converted to a string of character array with a `NULL` character at the end. There is a `length()` function that can get the total size of the string object. Run the following example in the IDE to see what it outputs,

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

int main(){
    string s;
    getline(cin,s);
    cout<<"String object accessed using array and loop...\n";
    for(int i=0; i<s.length(); i++){
        cout<<s[i];
    }
    cout<<endl;
    cout<<"Not using arrays or loops\n";
    cout<<s;

    return 0;
}
```

2.6 Exercises

1. Explain the differences between 1D & 2D arrays with appropriate examples.
2. What is the significance of `NULL` in a non-object string? Motivate your answer with an example.
3. What is the limitation of `cin` when printing a string in C++? What is the alternative to overcome such limitations? Explain with appropriate examples.
4. Write a C++ code to print the following array in reverse order:
   `int y[10] = {32,4,1,2,5,5,23};`
5. If you have an array, `int x[4] = {8,5,3,6}`, what will be the output of the following statements:
   `cout<<x[4];`
   `cout<<x[1];`
6. Given a 2D array, `int test[3][2] = {{21,25},{32,41},{35,61}}` write a loop in order to print the following from `test`:

`25, 41, 61`

7. What will you change in the declaration/ initialization of the 2D array `test` (in question 6) in order to make it a 1D array?
8. Again, take the array `test` (in question 6). Suppose, the starting address of the array is `1046`. What will be the address of `41`? What will be the address of `41` when you have converted `test` to a 1D array (in question 7)?
9. Write code in C++ to implement the functions: *strlen, strcmp, strcat, strcpy* by yourself.

2.7 References

1. https://en.wikipedia.org/wiki/Array_data_structure
2. https://en.wikipedia.org/wiki/Array_data_structure
3. https://www.programiz.com/cpp-programming/strings
4. https://cal-linux.com/tutorials/strings.html
5. http://www.cplusplus.com/reference/cstring/
6. http://www.cplusplus.com/reference/string/string/
7. https://cal-linux.com/tutorials/strings.html

Chapter 3

Pointer & Structure

In this chapter, we are going to explore two very powerful data structures. One of them is *pointer* and another one is *structure* or `struct` in short. We will start with the basic concept of pointer. Later we will dive deeper with lots of examples and their explanations using pointer. Finally, we will be ending this chapter by learning about structures and how powerful they are.

3.1 Pointer

Pointer deals with the address or location in the memory of the computer/ machine assigned to a certain value/ variable in the program. We are going to start by learning about the basics of pointer variables and finish the topic strongly with some advanced examples.

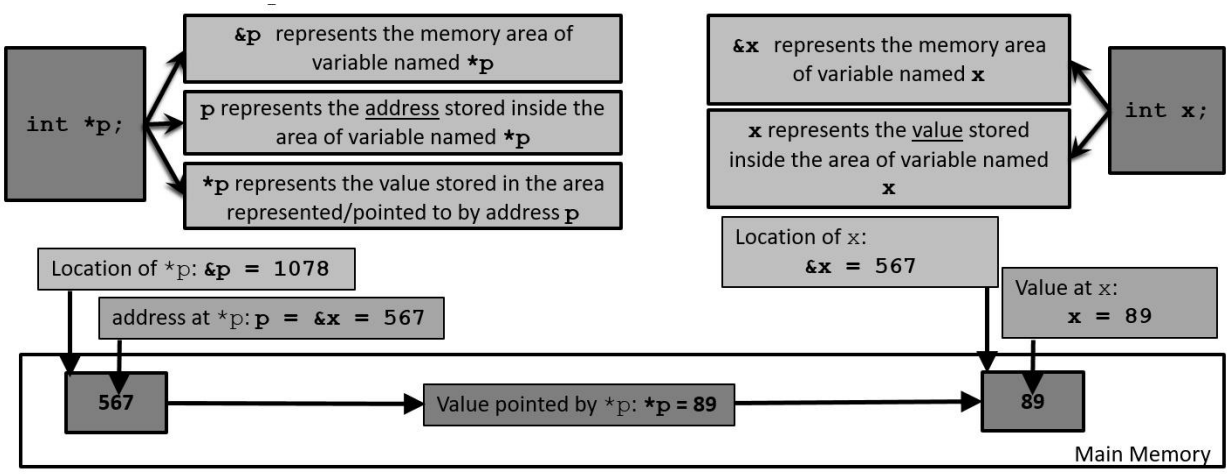
3.1.1 Variable

The computer accesses its memory not by using variable names but by using a memory map where each location of memory is uniquely defined by a number, called the address. Pointers are a very powerful, but primitive facility to avail that address.

To understand pointer let us go through the concept of variables once more. A variable is an area of memory that has been given a name. For example, `int x;` is an area of memory that has been given the name `x`. The instruction `x=10;` stores the data value `10` in the area of memory named `x`. The instruction `&x` returns the address of the location of variable `x`.

A pointer is a variable that stores the location of a memory/ variable. A pointer has to be declared. For example, `int *p;` Adding an asterisk (called the dereferencing operator) in front of a variable's name declares it to be a pointer to the declared type. Here, `int *p;` is a pointer – which can store an address of a memory location where an integer value can be stored or which can store an address of the memory location of an integer variable. For example, `int *p, q;` declares `p`, a pointer to `int`, and `q` an `int` and the instruction: `p=&q;` stores the address of `q` in `p`. After this instruction, conceptually, `p` is pointing at `q`.

After declaring a pointer `*p` variable, it can be used like any other variable. That is, `p` stores the address or pointer, to another variable; `&p` gives the address of the pointer variable itself, and `*p` is the value stored in the variable that `p` points at. The following figure illustrates the use of pointer in detail.



We can also take a look at an example in C++ below with output to better understand the basic use of pointers in programming language

```
1 // Understanding pointer variable
2 void main( void )
3 {
4  int x = 10;
5  int *p = &x;
6  int y = *p;
7  cout <<"Address of integer variable x: "<< &x <<"\n";
8  cout <<"Value stored in the memory area of x: "<< x <<"\n";
9  cout <<"Address of integer pointer variable *p: "<< &p <<"\n";
10 cout <<"Address stored in the area of pointer *p: "<< p<<"\n";
11 cout <<"Address of integer variable y: "<< &y <<"\n";
12 cout <<"Value pointed to by the pointer *p: "<< *p <<"\n";
13 cout <<"Value stored in the memory area of variable y: "<< y <<"\n";
14 }
```

```
Address of integer variable x: 0x8fbbfff0
Value stored in the memory area of x: 10
Address of integer pointer variable *p: 0x8fbbfff4
Address stored in the area of pointer *p: 0x8fbbfff0
Address of integer variable y: 0x8fbbfff8
Value pointed to by the pointer *p: 10
Value stored in the memory area of variable y: 10
```

variable	Memory Address	value	
int x	0x8f86fff0	10	x=10 &x=0x8f86fff0
	0x8f86fff1		
	0x8f86fff2		
	0x8f86fff3		
int *p	0x8f86fff4	0x8f86fff0	p=0x8f86fff0 &p=0x8f86fff4 *p=*(0x8f86fff0) =*(&x)=x=10
	0x8f86fff5		
	0x8f86fff6		
	0x8f86fff7		
int y	0x8f86fff8	10	y=*p=10 &y=0x8f86fff8
	0x8f86fff9		
	0x8f86fffa		
	0x8f86fffb		

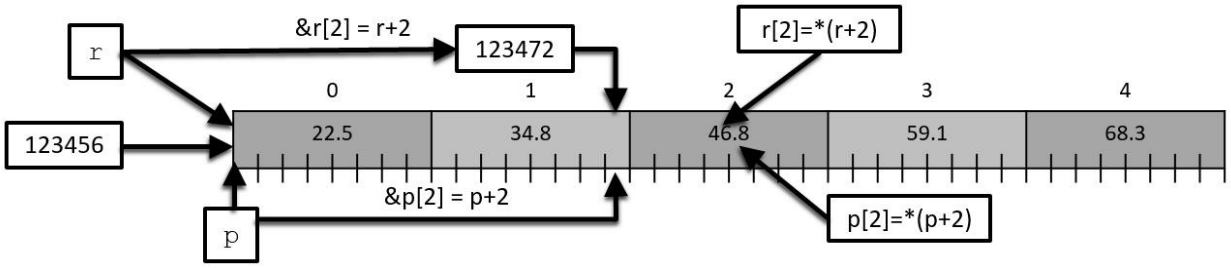
3.1.2 Pointer & Array

An array is simply a block of memory. An array can be accessed with pointers as well as with [] square brackets. The name of an array variable is a pointer to the first element in the array. So, any operation that can be achieved by an array subscription can also be done with pointers or vice-versa. The following example will clarify this idea. Take a look at the C++ code snippet below and its output.

```
1 void main( void )
2 {
3  float r[5] = {22.5,34.8,46.8,59.1,68.3};
4  cout <<"1st element: "<< r[0] <<"\n";
5  cout <<"1st element: "<< *r <<"\n";
6  cout <<"3rd element: "<< r[2] <<"\n";
7  cout <<"3rd element: "<< *(r+2)<<"\n";
8  float *p;
9  p = r; //&r[0]
10 cout <<"1st element: "<< p[0] <<"\n";
11 cout <<"1st element: "<< *p <<"\n";
12 cout <<"3rd element: "<< p[2]<<"\n";
13 cout <<"3rd element: "<< *(p+2)<<"\n";
14 for(int i=0; i<5; i++, p++)
15  cout <<"Element "<<(i+1)<<" is: "<<*p<<"\n";
16 }
```

```
1st element: 22.5
1st element: 22.5
3rd element: 46.8
3rd element: 46.8
1st element: 22.5
1st element: 22.5
3rd element: 46.8
3rd element: 46.8
Element 1 is: 22.5
Element 2 is: 34.8
Element 3 is: 46.8
Element 4 is: 59.1
Element 5 is: 68.3
```

In the above example, the array `float r[5];` or the pointer variable `*p` (after `p=r`) is a pointer to the first floating-point number in the declared array. The 1st element of the array, `22.3`, can be accessed by using: `r[0]`, `p[0]`, `*r`, or `*p`. The 3rd element, `46.8`, could be accessed by using: `r[2]`, `p[2]`, `*(r+2)` or `*(p+2)`. Now, let's examine the notation `(r+2)` and `(p+2)` below.



Assuming the starting address of the array numbers is 123456 –

```
r[0]=(r+0) starts at address,
   ⇨ r+0*sizeof(float)
   ⇨ 123456 + 0 * 8
   ⇨ 123456

r[1]=(r+1) starts at address,
   ⇨ r+1*sizeof(float)
   ⇨ 123456 + 1 * 8
   ⇨ 123464
```

```
r[2]=(r+2) starts at address,
   ⇨ r+2*sizeof(float)
   ⇨ 123456 + 2 * 8
   ⇨ 123472
```

3.1.3 Void Pointer

The `void` type of pointer is a special type of pointer which represents the absence of type. So, `void` pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereference properties). This allows `void` pointers to point to any data type, `int`, `float`, `char`, `double`, or any type of array. But the data pointed by them cannot be directly dereferenced since we have no type to dereference to. So, we need to cast the address in the `void` pointer to some other pointer type that points to a concrete data type before dereferencing it. Let us look at the following example and try to understand how void pointer works here.

```
1  // increaser                                  Y, 1603
2  void increase(void *data, int psize){
3    if ( psize == sizeof(char) ){
4       char *pchar;
5       pchar=(char*)data;
6       ++(*pchar);
7    }
8    else if (psize == sizeof(int)){
9       int *pint;
10      pint=(int*)data;
11      ++(*pint);
12   }
13 }
14 void main (void){
15   char a = 'x';
16   int b = 1602;
17   increase (&a,sizeof(a));
18   increase (&b,sizeof(b));
19   cout << a << ", " << b << endl;
20 }
```

In the example above, we start with the `main` function where two variables `char a` and `int b` is created with the values `'x'` and `1602` respectively (line 15-16). Line 17 calls the function `increase` with the address of `a` and the size of `a` as parameters. `sizeof(a) = sizeof(char) = 1`, as char type is one byte long. Line 17 gives the control to the function `increase` and it creates two (parameter) variables `void *data` and `int psize` assigned with the address value of a and `sizeof(a) = 1` of main respectively. Though `*data` contains the address of `a` in `main`, this address cannot be accessed using `*data` with type mismatch. With the `true` value of the conditional statement `psize==sizeof(char)`

another new pointer variable `char *pchar` is created and is assigned the value `*data` (line 3-5). As `*data` has no type, it must be type casted to `(char*)` before being assigned (line 5).With the statement `++(*pchar);` pointer variable `*pchar`, pointing to `a` of `main`, is increased by one. So, the value of `a` in `main` is changed to `'y'` (the ASCII value is increased by one) (line 6). Before exiting the function `increase`, the variables created by `increase` is destroyed. Then the control goes back to the function `main` (in line 17). So, we see the value `a` in `main` is changed to `'y'`.

Line 18 calls the function `increase` with address of `b` and the size of `b` as parameters. Here, `sizeof(b) = sizeof(int) = 4`, as `int` type is four bytes long. Now the control goes to the function `increase` and it creates two (parameter) variables `void *data` and `int psize` assigned with the address value of `a` and `sizeof(b) = 4` of `main` respectively. With the `true` value of the conditional statement `psize==sizeof(int)`, a new pointer variable `int *pint` is created and assigned to the value, `*data` (line 8-10). Though `*data` contains the address of `b` in main, this address cannot be accessed using `*data` with type mismatch (line 10). As `*data` has no type, it must be type cast to `(int*)` before being assigned (line 10). With the statement `++(*pint);` pointer variable `*pint`, pointing to `b` of `main`, is increased by one. So, the value of `b` in the main is changed to `1603` (line 11). Before exiting the function `increase`, the variables created by `increase` is destroyed. Then the control goes back to the function `main` (in line 17). So, we see the value `b` is changed to `1063`.

3.1.4 NULL Pointer

A `NULL` pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointing to any valid reference or memory address. This value is the result of type-casting the integer value `zero` to any pointer type.

```
1 int *p;
2 p = 0;  //can also be written, p = NULL;
3 /* p has a null pointer value */
```

Do not confuse `NULL` pointers with `void` pointers. A `NULL` pointer is a value that any pointer may take to represent that it is pointing to "nowhere", while a `void` pointer is a special type of pointer that can point to somewhere without a specific type. One refers to the value stored in the pointer itself and the other to the type of data it points to.

3.1.5 Dynamic Memory Allocation

The exact size of an array is unknown until the compile-time, i.e., the time when a compiler compiles code written in a programming language into an executable form. The size of an array declared initially can be sometimes insufficient and sometimes more than required. Also, what if we need a variable amount of memory that can only be determined during runtime? Dynamic memory allocation allows a program to obtain more memory space while running or to release

space when no space is required. C++ integrates the operators `new` and `delete` for dynamic memory allocation.

To request dynamic memory we use the operator `new`. The operator `new` is followed by a data type specifier. If a sequence of more than one memory block is required, the data type specifier is followed by the number of these memory blocks within brackets `[]`. It returns a pointer to the beginning of the new block of memory allocated. Syntax:

```
1 pointer = new vtype;
2 pointer = new vtype [number_of_elements];
```

The first expression is used to allocate memory to contain one single element of type `vtype`. The second one is used to assign a block (an array) of elements of type `vtype`, where `number_of_elements` is an integer value representing the amount of these.

Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of the operator delete, whose format is:

```
1 delete pointer;
2 delete [] pointer;
```

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements. The value passed as an argument to delete must be either a pointer to a memory block previously allocated with `new`, or a `null` pointer (in the case of a `null` pointer, `delete` produces no effect).

We have learned about dynamic programming and what the keywords `new` and `delete` do. Now, we will take a look at an example (C++ code snippet) below to better understand their functionalities. Run this in your IDE.

```
1 // new, delete
2 void main(void){
3     int n, i,*ptr, sum=0;
4     cout << "# of elements: ";
5     cin >> n;          //input 5
6     ptr = new (nothrow) int[n];
7     if(ptr==NULL){  //ptr==0
8       cout << "Error! not
9 allocated.";
10      return 1;
11    }
12    cout << "Enter elements:\n";
13    for(i=0;i<n;++i)
14    {   //input 2 6 7 4 3
15        cin >> *(ptr+i);  //ptr[i]
16        sum += *(ptr+i);
17    }
18    cout << "Sum = " << sum;
19    delete [] (ptr);
20    //memory de-allocated
  }
```

```
# of elements: 5
Enter elements:
2
6
7
4
3
Sum = 22
```

3.1.6 Pointer & Function

Passing arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. Pointer arguments enable a function to access and change objects in the function that called it. Let's consider the example below.

```
1 /* Swap two numbers using function. */
2 void swap(int *a, int *b);
3 void main(void){
4   int num1=5,num2=10;
5   swap(&num1,&num2);
6   /* address of num1, num2 is passed */
7   cout<<"Number1 = "<<num1<<"\n";
8   cout<<"Number2 = "<<num2;
9 }
10 void swap(int *a, int *b){
11 //a,b points to &num1,&num2 respectively
12   int t;
13   t = *a;
14   *a = *b;
15   *b = t;
16 }
```

```
Number1 = 10
Number2 = 5
```

The program starts with function `main` getting the control and creating two variables `num1` and `num2` with values `5` and `10` respectively (line 4). Function `main` calls the function `swap` with two parameter values `&num1` and `&num2` (line 5). So, the address of `num1` and `num2` is sent through the parameter of `swap`.

Now the control goes to the function `swap` and it creates two pointer (parameter) variables `*a` and `*b` assigned with the address values of `num1` and `num2` of main respectively. Another variable `t` is created (line 12). With the statement `t=*a;` variable `t` is assigned to the value, `num1` of `main`, pointed to by pointer `*a` (line 13). With the statement `*a = *b;` pointer variable `*a`, pointing to `num1` of `main`, is assigned to the value, `num2` in `main`, pointed to by pointer `*b` (line 14). With the statement `*b = t;` pointer variable `*b`, pointing to `num2` of `main`, is assigned to the value of `t` (line 15). Any changes we make to `*a` and `*b` in function `swap`, will change the values of `num1` and `num2` respectively in function `main` as `a` is pointing to `num1` and `b` is pointing to `num2`.

Before exiting the function `swap`, the variables created by `swap` is destroyed. Then the control goes back to the function `main` (in line 5). But nothing changes for the values of the variables `num1` and `num2` of main due to the destruction of the variables `*a` and `*b`. Because destruction only destroys the space provided for `*a` and `*b`. It does not destroy the space it was pointing to. Whatever changes were made in `swap` by `*a` and `*b`, they remain.

3.1.7 Pointer, Array & Function

We now know the strong relation pointers and arrays share. We have also seen previously that we can send pointer variables as function parameters. Now it is time for us to explore some examples of how these three might work together.

```
1 // arrays as parameters
2 void TwiceArray (int arg[], int length) {
3   for (int n=0; n<length; n++) arg[n] *= 2;
4 }
5 void PrintArray(const int arg[], int
6 length){
7   for (int n=0; n<length; n++)
8     cout<<arg[n];
9   cout<<endl;
10 }
11 void main (void){
12   int FirstArray[3] = {5, 10, 15};
13   int SecondArray[] = {2, 4, 6, 8, 10};
14   TwiceArray (FirstArray,3);
15   PrintArray (FirstArray,3);
16   PrintArray (SecondArray,5);
  }
```

In the above example, we start with the function `main` where two arrays, `FirstArray` with 3 elements and `SecondArray` with 5 elements, are declared, created, and initialized (line 11-12). Both these identifiers will hold the starting address/location of their elements. `FirstArray` holds the location of the first element, `&FirstArray[0]` and `SecondArray` holds the location of the first element, `&SecondArray[0]`.

Then, the `TwiceArray` is called with the parameters `FirstArray`, the starting location of the array itself or the location of the first element is passed to the parameter int `arg[]` and value `3` to the parameter length (line 13). The identifier `arg` holds the starting address of the `FirstArray` of the function `main`. As `arg` itself is an array, `arg` behaves like an array. The control is transferred from the function `main` to function `TwiceArray` (line 2).

Inside function `TwiceArray` another variable `n` is declared. Using `n` in *for-loop* 3 elements of `arg` are made twice of their original (line 3). As `arg` represents the array `FirstArray` on main, the 3 elements of the `FirstArray` are made twice. Hypothetically, `FirstArray[n] <- arg[n]*=2`.

Before exiting from the function `TwiceArray`, all the variables (`arg`, `length`, `n`) are destroyed and the control is returned to the `main` (line 13). Values of `FirstArray` changed in `TwiceArray` remain. Next, the function `PrintArray` is called to print the elements of the `FirstArray` and `SecondArray` consecutively. Here, it works as same in terms of parameter passing. Except for the parameter `arg` in `PrintArray` which is declared as a constant variable. As we do not need to change any elements of the array inside `PrintArray`, the parameter `arg` is declared as a constant variable. This is how any function (`main` in this example) can protect its data array (`FirstArray`) from being changed by another function (`PrintArray`).

Let us take a look at another example, where we will also know about another need of using dynamic memory allocation:

```
1 //Array Multiplication
2 int *ArrMul(int a[], int b[], int size){
3   int i,*c = new int[size]; //c[5]
4   for(i=0; i<size; i++) c[i] = a[i] * b[i];
5   return c;
6 }
7 void PrintArr(int *a, int size){
8   for(int i=0; i<size; i++) cout<<a[i]<<"\t";
9   cout << "\n ";
10 }
11 void main(void){
12   int *z, x[5]={1,2,3,4,5}, y[5]={5,6,7,8,9};
13   z = ArrMul(x, y, 5);
14   cout <<"First array:\n";
15   PrintArr( x, 5 );
16   cout <<"second array:\n";
17   PrintArr( y, 5 );
18   cout <<"result array:\n";
19   PrintArr( z, 5 );
20   delete [] (z); //memory deallocated. If you
21 look carefully, we are deallocating the same
   memory that we allocated, because that memory
   was passed to variable z from variable c;
   }
```

```
First array:
1   2   3   4   5
second array:
5   6   7   8   9
result array:
5   12  21  32  45
```

Two array `x` and `y` with five elements are multiplied index wise using the function `ArrMul`. Function `ArrMul` (line 2-6) dynamically allocates memory for the resultant array `c` of the multiplication.

Before exiting from function `ArrMul` the address stored in `*c` returned and all the variables created in `ArrMul` are destroyed. The control is transferred to `main` and `z` is assigned to the address value returned by `c` of `ArrMul` (line 13). Then the arrays represented by `x`, `y`, and `z` are printed using function `PrintArr` (line 14-19). Line 20 de-allocates the memory allocated to `*c` in function `ArrMul` (line 3) and later returned to `*z` in function `main` (line 13). A dynamically allocated memory must be de-allocated.

Consider the case (example below), if dynamic array `*c` in `ArrMul` was declared as an array `c[5]`. That is, instead of int `*c = new int[size]` in line 3, if it was `int c[5]`, the following would happen.

```
1 //Array Multiplication
2 int *ArrMul(int a[], int b[], int size){
3   int i, c[5];
4   for(i=0; i<size; i++) c[i] = a[i] * b[i];
5   return c;
6 }
7 void PrintArr(int *a, int size){
8   for(int i=0; i<size; i++)
9 cout<<a[i]<<"\t";
10  cout << "\n ";
11 }
12 void main(void){
13  int *z,x[5]={1,2,3,4,5},y[5]={5,6,7,8,9};
14  z = ArrMul(x, y, 5);
15  cout <<"First array:\n";
16  PrintArr( x, 5 );
17  cout <<"second array:\n";
18  PrintArr( y, 5 );
19  cout <<"result array:\n";
20  PrintArr( z, 5 );
21 }
```

The address represented by `c` is returned and all the variables created in `ArrMul` are destroyed and control is transferred to main at the exit from `ArrMul`. The address represented by `c` is returned and all the variables created in `ArrMul` are destroyed and control is transferred to `main` at the exit from `ArrMul`. The pointer variable `*z` is assigned to the address value returned by `c` (line 14). Then, `PrintArr` finds an error when trying to print the array `z` (line 20), as the address represented by `z` has already been destroyed at the exit of the function `ArrMul`. So, while returning a pointer to any variable or memory area, we must make sure that the returned memory area is active or not destroyed after the return.

So far in this section, we have seen examples regarding 1D array. Let us explore the case when we need to send a 2D array as a parameter to a function. We can consider the following example of that case.

```
1  #include<iostream>
2  using namespace std;
3
4  void matrixAddition(int a[][2], int b[][2]){
5      int c[2][2];
6      for(int i=0; i<2; i++){
7          for(int j=0; j<2; j++){
8              c[i][j] = a[i][j] + b[i][j];
9              cout<<c[i][j]<<"\t";
10         }
11         cout<<endl;
12     }
13 }
14
15 int main(){
16     int m[][2] = {{1,2}, {3,4}};
17     int n[][2] = {{5,6}, {7,8}};
18     matrixAddition(m,n);
19     return 0;
20 }
```

The above example is about matrix addition where two 2D arrays `m` & `n` of the same size are sent to a function `matrixAddition` as a parameter where they are added by their similar indices and saved into another 2D array `c` of same size simultaneously. In the process, `c` is also printed out. There are some new things to take away from this example. First of all, to initialize a 2D array, mentioning the 1st dimension is optional. However, the 2nd dimension must be mentioned. Also, in line 4, when receiving the array address' sent to the `matrixAddition` function as a parameter, we may use only the 2nd dimension (1st dimension is optional).

3.1.8 Pointers & Initialization

Consider the statement `int *p;` which declares a pointer `p`, and like any other variable, this space will contain garbage (random numbers), because no statement like `p = &someint;` or `p = new int;` has yet been encountered which would give it a value.

Writing a statement `int *p=2000;` is syntactically correct as `p` will point to the 2000th byte of the memory. But it might fail as byte `2000` might be being used by some other program or maybe being used by some other data type variable of the same program. So such initialization or assignment must be avoided unless the address provided is guaranteed to be safe.

Let us take a look at one of the important differences between some pointer definitions. They are given below:

```
char amsg[] = "now is the time"; /* an array */
char *pmsg = "now is the time"; /* a pointer */
```

- Here, `amsg` is an array, just big enough to hold the sequence of characters and `'\0'` that initializes it. Individual characters within the array may be changed but `amsg` will always refer to the same storage and size after declaration and initialization.
- On the other hand, `pmsg` is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.

3.2 Structure

The array takes simple data types like `int`, `char`, `double`, etc. and organizes them into a linear array of elements all of the same type. Now, consider a record card that records name, age, and salary. The name would have to be stored as a `string`, the age could be `int` and salary could be `float`. As this record is about one person, it would be best if they are all stored under one variable. At the moment the only way we can work with this collection of data is as separate variables. This isn't as convenient as a single data structure using a single name and so the C++ language provides *structure*. A *structure* is an aggregate data type built using elements of other types.

3.2.1 Defining Structure in C++

In general "structure" in C++ is defined as follows:

```
struct name{
  list of component variables
};
```

Here `struct` is the keyword, `name` is an identifier defining the structure name, `list of component variables` declares as much different type of variables as needed. The structure name works as the new data type defined by the user. The definition ends with a semicolon. For example, suppose we need to store a name, age, and salary as a single structure. You would first define the new data type using:

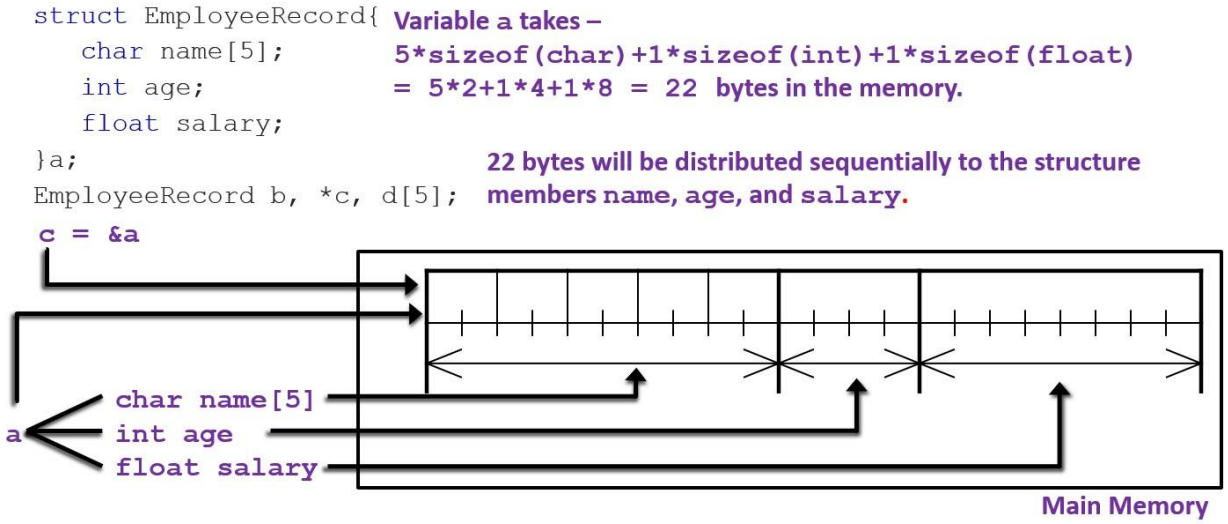
```
struct EmployeeRecord{
   char name[5];
   int age;
   float salary;
};
```

So, `EmployeeRecord` is the new user-defined data type and a variable `b` of type `EmployeeRecord` can hold a total of 22 bytes of information [5 consecutive characters (5*2=10 bytes), followed by an integer (4 bytes), and a floating-point number (8 bytes)]. Just

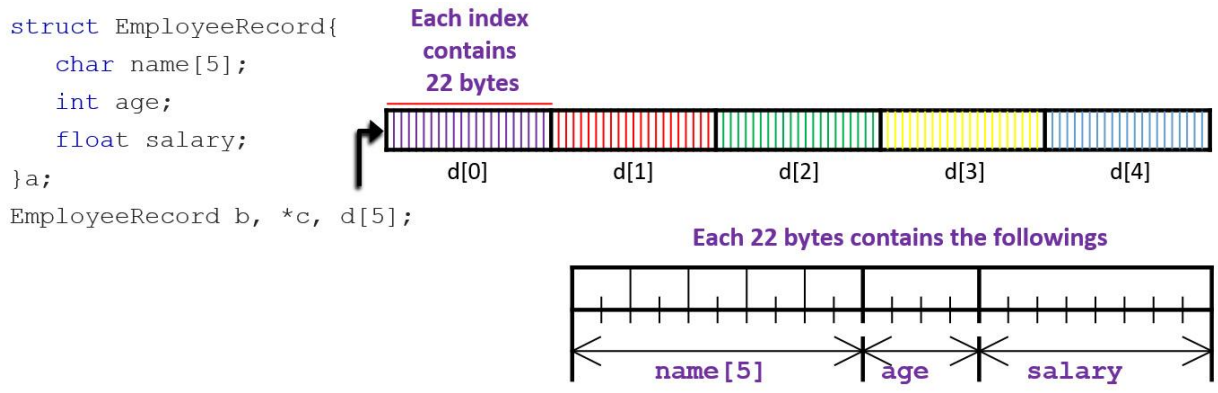
like when we say an `int` is a compiler defined data type and a variable `x` of type `int` can hold 4 bytes of an integer number.

3.2.2 Declaring Variable of a Structure

As we can declare variables for compiler defined data types (example: `int a, b, *c, d[50];`), we can do the same for user-defined data type created using `struct`.



In the above example, it is illustrated in detail how much memory is required for a variable of type `EmployeeRecord`. Also, the use of pointer is available for structure variables like any other variable of any type. Understandably, to store the address of a `struct` variable, the type of the pointer variable should be the same as the `struct` variable. In this case, `EmployeeRecord`.



Above, we can see the illustration of the memory arrangement of an array `d[5]` of type `EmployeeRecord`. As each `EmployeeRecord` is of 22 bytes in size, an array of 5 elements of such type should hold 110 bytes of data.

3.2.3 Accessing & Initializing Structure Member/ Variable

The dot (.) or combination of dash-line and greater-than sign (->) is used as operator to refer to members of struct. For example,

```
struct EmployeeRecord{
   char name[5];
   int age;
   float salary;
};
EmployeeRecord x, y[5], *p;
x.age = 22;
x.salary = 1234.56;
strcpy(x.name, "Sam");
y[2].age = 22;
p = &x;
p->age = 22;
```

In the above example, variable `x` is of type `EmployeeRecord`, `x.age` is of type `int`, `x.salary` is of type `float`, `x.name` is of type `char[5]`, `y[2].age` is of type `int` where `y[2]` is of type `EmployeeRecord` and represents the 3rd element, and `p->age` is of type `int` where `p` is a pointer pointing to variable `x` of type `EmplyeeRecord`. Operator (`->`) is used for a pointer variable of struct instead of (.). Here, `p->age` can be represented as `(*p).age` also. Member variables can be used in any manner appropriate for their data type.

To initialize a `struct` variable, follow the `struct` variable name with an equal sign, followed by a list of initializers enclosed in braces in sequential order of definition. For example,

```
EmployeeRecord x = {"Sam", 22, 1234.56};
```

Here, `"Sam"` is copied to the member `name` referred to as `x.xame`, `22` is copied to the member `age` referred to as `x.age`, and `1234.56` is copied to the member `salary` referred to as `x.salary`.

3.2.4 Some Facts About Structure

No memory (as data) is allocated for defining `struct`. Memory is allocated when its instances/variables are created. Hence `struct` stand-alone is a template… and it cannot be initialized. You need to declare a variable of type `struct`. No arithmetic or logical operation is possible on the `struct` variables unless defined by the operator overloading. For example, for `EmployeeRecord a, b;` any expression like `a == b` or `a + b`, etc. is not possible. But `a.age = b.age` is possible as both of these are of type `int`. Only assignment operation works. i.e. `a = b`; call-by-value, call-by-reference, return-with-value, return-with-reference, array-as-parameter – all works with a `struct` variable same as the normal variable concept using function in C++.

3.2.5 Nested Structure

Like any number and type of variables declared inside a structure, another structure can also be declared/defined inside another structure. We are going to take a look at two separate examples to get an idea of how that might work.

```
Example 1:
struct Appointment{
  struct AppDate{
    int day, month, year;
  }dt;

  struct AppTime{
    int minute, hour;
  }tm;

  char venue[100];
};
```

```
Example 2:
struct DateOfBirth{
  int day, month, year;
};

struct Employee{
  char EmpName[100];
  DateOfBirth dob;
};
```

In *example 1*, `AppDate` and `AppTime` is defined inside the structure `Appointment`. Here, `dt` is a variable for the structure `AppDate`. And, `tm` is a variable for the structure `AppTime`. Both `dt` and `tm` is declared inside the structure `Appointment`. We cannot access `dt` and `tm` directly from outside the structure `Appointment` because their scope is only limited to the scope of `Appointment`. As `dt` and `tm` were declared inside the structure `Appointment`, they are not directly globally accessible; only locally accessible inside `Appointment`. However, we can access `dt` and `tm`, indirectly through a variable of `Appointment`. For instance, if we declare a variable `Appointment x;`, then we can use `x` to access `dt` and `tm` like `x.dt` and `x.tm`. Also, as `AppDate` and `AppTime` both are defined inside `Appointment`, they are unable to declare their variables anywhere outside `Appointment`, again because of their scope being limited to `Appointment`.

In *example 2*, `dob` is a variable for the structure `DateOfBirth`. Here, `dob` is declared inside structure `Employee`. Here, the scope of `dob` is limited to Employee as it is declared inside `Employee`. We can access `dob` indirectly through any variable of `Employee`. On another note, unlike `AppDate` and `AppTime` in example 1, `DateOfBirth` is not defined inside a structure. Therefore, it is possible to declare its variable anywhere.

3.2.6 Self-referential Structure

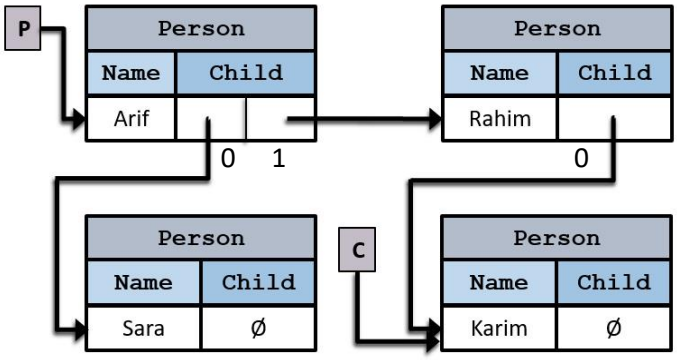
A structure member cannot be an instance of enclosing `struct`. However, a structure member can be a pointer to an instance of enclosing `struct`. This concept is called a self-referential structure. The concept of self-referential structure is used to implement some useful data structures such as linked-list, queue, stack, tree, etc. For example,

```
struct Person{
```

```
    char Name[30];
    Person *Child;
};
```

In the above example, `Person` contains a pointer variable `Child` of type `Person`. This illustrates the fact that every person may have a child which is also a person. We can illustrate a more detailed representation of the example by extending it with some variables `Person P, *C;` and their use:

```
Person P, *C;
strcpy(P.Name, "Arif");
C = P.Child = new Person[2];
strcpy(C[0].Name, "Sara");
C[0].Child = NULL;
strcpy(C[1].Name, "Rahim");
C = C[1].Child = new Person;
strcpy(C->Name, "Karim");
C->Child = NULL;
```



In the above illustration, we can simply say:

- Mr. Arif has two children – Rahim and Sara.
- Mr. Rahim has one child – Karim.
- Ms. Sara has no child.

3.3 Exercises

10. Explain the differences between NULL & Void pointers with appropriate examples.
11. What is the *dereferencing* operator? Use it in an example to explain how it works.
12. What is *dynamic memory allocation*? What operators are needed to allocate and deallocate memory during the runtime of a program?
13. Write a program in C++ to demonstrate the use of `&`(address of) and *(dereferencing) operator.
14. Assume the following definitions and initializations:

```
char c = 'T', d = 'S';
char *p1 = &c;
char *p2 = &d;
char *p3;
```

Assume further that the address of c is 6940, the address of d is 9772, and the address of e is 2224. What will be printed when the following statements are executed sequentially?

```
p3 = &d;
cout << "*p3 = " << *p3 << endl;   // (i)

p3 = p1;
```

```
cout << "*p3 = " << *p3              // (ii)
     << ", p3 = " << p3 << endl;    // (iii)

*p1 = *p2;
cout << "*p1 = " << *p1              // (iv)
     << ", p1 = " << p1 << endl;    // (v)
```

15. Consider the following statements:

```
int *p;
int i;
int k;
i = 42;
k = i;
p = &i;
```

After these statements, which of the following statements will change the value of i to 75?

a. k = 75
b. *k = 75
c. *p = 75
d. p = 75

16. Explain the error for the following statements:

```
int a = 95;
char *d = &a;
```

17. Write a function oddCount(int*, int) which receives an integer array and its size, and returns the number of odd numbers in the array.
18. Explain the *self-referential structure* with an appropriate example.
19. Explain the *nested structure* with appropriate examples.
20. Why do we need structure? State some of the major differences between structure and array with appropriate examples.
21. Can we send a `struct` variable as a parameter to a function? Try this out in your IDE and see if it works.
22. If you can successfully do what is told in *exercise 12*, go ahead and write a `Student` struct that will have the capability to store a student's name, CGPA, and credits completed. Take 10 students' info as input. Finally, create a function *printStudentInfo* that will print all the student info.

3.4 References

8. http://www.cplusplus.com/doc/tutorial/pointers/
9. http://www.cplusplus.com/doc/tutorial/structures/
10. http://www.cs.uregina.ca/Links/class-info/115/10-pointers/
11. https://condor.depaul.edu/ntomuro/courses/309/notes/pointer_exercises.html

Chapter 4

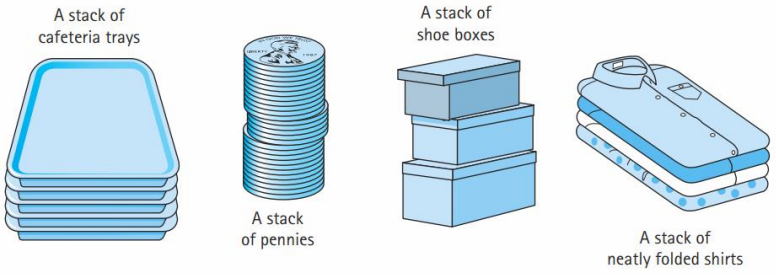
Stack & Queue

In Chapters 2 & 3 we covered the built-in data structures in C++. We did not cover any **A**bstract **D**ata **T**ype (ADT) yet. In this chapter, we are going to learn about two of the finest and simplest but most important ADTs out there: *Stack* & *Queue*. We will start with stack and explore it in detail. We are going to explore the array-based implementation of stack with both *static* and *dynamic* array in C++. Later, we will also go through the queue ADT in detail with the array-based implementation which will include both *linear queue* and *circular queue* variants in C++. Finally, we end the chapter with some applications of stack & queue. After finishing this chapter, you should be able to define stack & queue by yourself as well as learn when to apply or use them to solve problems more efficiently.

4.1 Stack

A stack is a version of a list (or array) that is particularly useful in applications involving the reversing of data sequence. In a stack data structure, all insertions and deletions of entries are made at one end, called the top of the stack. Due to this behavior stack is called a LIFO (**L**ast **I**n, **F**irst **O**ut) data structure where last inserted data remains at the top of the stack and is available to be out first.

A helpful analogy is to think of a stack of trays or plates sitting on the counter in a busy cafeteria. Throughout the lunch hour, customers take trays off the top of the stack, and employees place returned trays back on top of the stack. The tray most recently put on the stack is the first one taken off. The bottom tray is the first one put on, and the last one to be used.



A stack has two principal operations: *push* and *pop*. Inserting or adding a new element to the stack takes place from the top of the stack and thus it is called push. Similarly, an element is removed or deleted from the top of the stack making it a pop operation. Apart from these two main operations, we are going to also learn some more for a complete implementation of a stack, such as checking emptiness/ fullness of a stack, retrieving the top value/ element of the stack, and finally printing/ showing the stack elements from top to bottom.

We are going to explore an array-based implementation of stack in this chapter. This implementation opens up two possible types of stacks in terms of their size: bounded and unbounded (or non-bounded).

In case of a bounded stack, if the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state. A pop either reveals previously concealed items or results in an empty stack – which means no items are present in stack to be removed.

On the other hand, an unbounded stack is a stack where memory for the stack is allocated dynamically. As a result, there is no chance of stack overflow to take place in such stack.

4.1.1 Implementation of Bounded Stack in C++

The *stack* data structure can be implemented in C++ with the help of an array. All the stack operations we mentioned earlier, can be implemented too with that approach. Below, you will find the list of functions in C++ representing those operations.

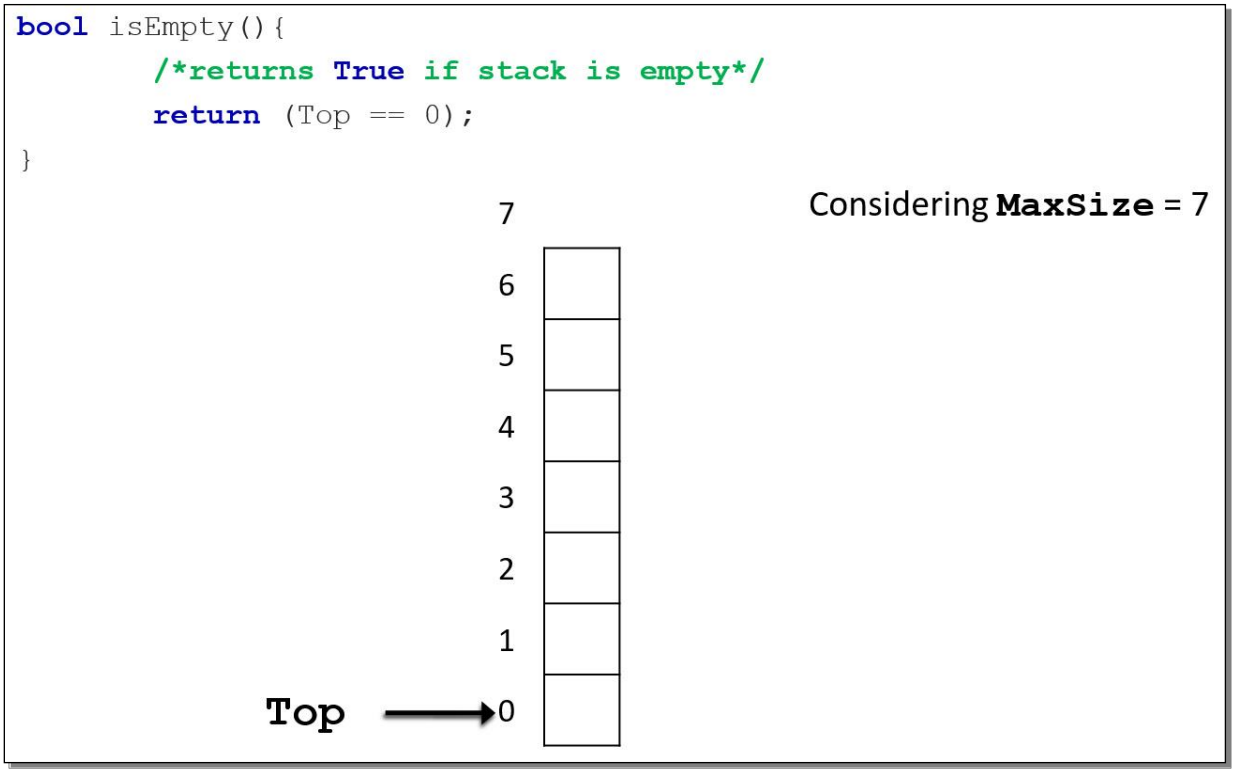
```
int Stack[100], Top=0, MaxSize=100;//Stack[] holds the elements;
Top is the index of Stack[] always holding the whereabouts of the
first/top element of the stack

bool isEmpty(); //returns True if stack has no element

bool isFull(); //returns True if stack full

bool push(int Element); //inserts Element at the top of the stack

bool pop(); //deletes top element from stack into Element

int topElement(); //gives the top element in Element

void show(); //prints the whole stack
```

Here, the array `int Stack[100]` serves the purpose of being the stack. In this case, it is an integer stack (meaning we can only hold integer values/ elements in this stack). An integer variable called `Top` keeps track of the index of the array where the top element of the stack will reside. In the beginning, the value of `Top` is 0 (zero), meaning the stack is empty but if an element were to be pushed in the stack now, they will be on the 0^{th} index of the `Stack` array. The `MaxSize` variable of type `int` holds the maximum elements that the stack can hold before it overflows as this is a bounded implementation.

Before anything is popped from the stack, it should be checked whether the stack has any element in it in the first place. If it does not, no pop operation can be performed at all. Therefore, we need to have the ability to check whether the stack is empty in our implementation. We can achieve that by writing a function called `isEmpty`. The only purpose of this function is to return `true` if the stack is empty (when the value of `Top` is `0`) and `false` if it is not. Therefore, the

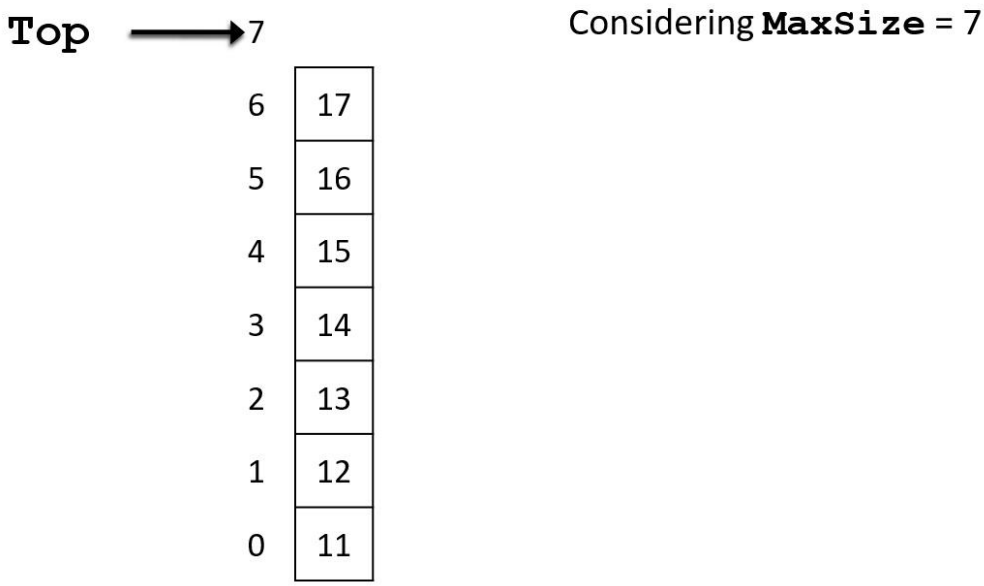
return type of this function is `bool`. Below we can see an illustration of an empty stack and when the function `isEmpty` will return `true` with the help of the implementation in C++:

```
bool isEmpty(){
        /*returns True if stack is empty*/
        return (Top == 0);
}
```



Like `isEmpty`, we also need the mechanism to check whether the stack is full. If it is, then no more elements can be pushed into the stack making the stack overflow. If it is not full, then at least one more element can be pushed. We can write a function `isFull` that will do this check every time before anything is pushed into the stack. The return type of `isFull` is `bool` as it will return `true` if the stack is full (when the value of `Top` is the same as the value of `MaxSize`) and `false` otherwise. Below, let us take a look at the C++ implementation of the function along with an illustration that depicts the stack to be full:

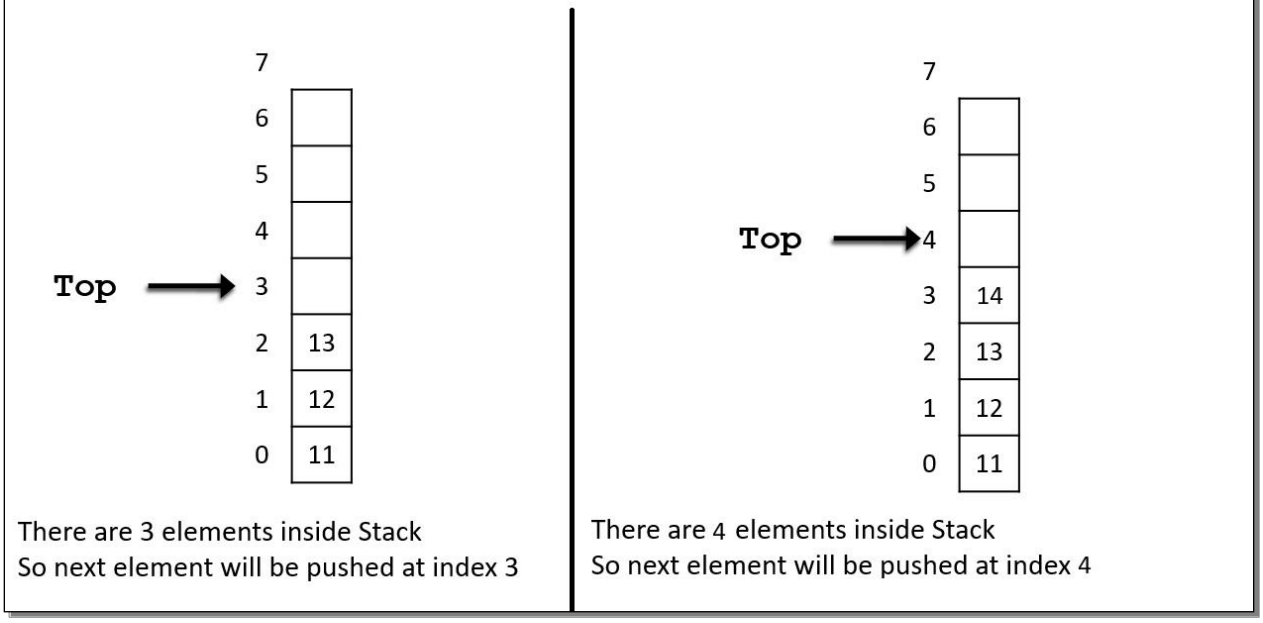
```
bool isFull(){
        /*returns True if stack is full*/
        return (Top == MaxSize);
}
```



As discussed earlier, to add or insert something into the stack, we call it *push*. We can write a function `push` in C++ to perform that operation. The function takes the element that needs to be pushed into the stack as a parameter `int Element`. The function definition will include a check for the fullness of the stack with the help of the `isFull` function defined earlier. Only when it is confirmed that the function is not full, we are then able to push new element into the stack. So, where should this new element be in the array `Stack`? Which index should it occupy in that array?

As we know, the variable `Top` keeps track of the top of the stack by remembering where the next element of the stack will be pushed. That means, if any new element is added to the stack, the index of the `Stack` array that it will occupy is what the value of `Top` is. For example, in the following illustration, the left stack is before when the element 14 is added into the stack. As it is seen there, the value of Top is then 3, meaning the next element that will be pushed into the stack will be pushed in the index 3 of the `Stack` array. After the element is pushed, we increase the value of `Top` by 1 and then as you can see in the right stack in the below-given illustration, the value of `Top` is then 4 (which means the next element that will be pushed into the stack, will occupy index 4 of the `Stack` array). So, in our implementation, we have to capture this phenomenon. And this can be done fairly simply. When we push the `Element` into the `Stack` array the statement `Stack[Top++] = Element` is the one that represents it. So, what happens here? This statement means, we are pushing the `Element` into the stack that will occupy the index `Top` (3 in the example below) of the `Stack` array. Then the index will be increased by 1 to make the value of `Top` 1 higher (4 in the example below).

```
bool push(int Element ){
     // inserts Element at the top of the stack
     if( isFull( ) ) { cout << "Stack is Full\n"; return false; }
     // push element if there is space
     Stack[ Top++ ] = Element;
     return true;
}
```



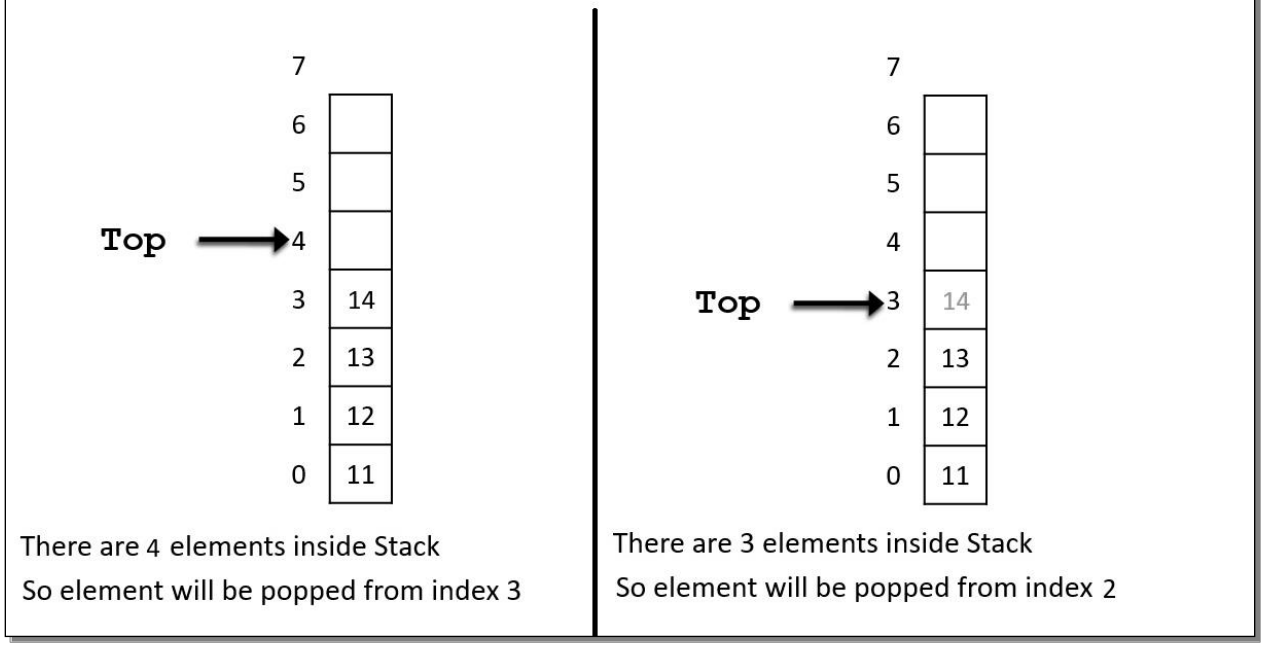
The operation of removing an element from the stack is called *pop*. We can capture this operation in our implementation with the help of a function called `pop`. The following illustration with a code-snippet shows the basics of this operation.

When the function `pop` is called, an element from the top of the stack should be removed. This will make the previous element (that was second to top of the stack before the pop operation took place) the top element of the stack now. Let us take a look at how that occurs in terms of the C++ implementation of the function.

Every time anything is popped from the stack, we need to have a check first whether the stack is empty. Because if it is, then there is nothing to pop from the stack and we get out of the `pop` function by returning `false`. That means nothing was popped from the stack. This check is done by calling the function `isEmpty`. Now, if the stack is not empty, we proceed to pop the element at the top of the stack. As this is an array-based implementation of stack and we know that we cannot delete an array element entirely by removing or freeing its memory here, we just make sure that the value of the variable `Top` is decreased by 1. That will simply mean there is one less element in the stack now. We can see in the illustration below that the stack on the left has 4 elements. After popping once, it has only 3 elements (stack on the right) and the value of `Top` is decreased by 1. Note, the element `14` remains in the `Stack` array but is not part of this particular stack data structure anymore theoretically because of where the `Top` is pointing to (as

we know the value of `Top` represents the index of the `Stack` array where the next element will be stored after being pushed. That means the index that the `Top` points to, is theoretically an empty slot for the stack data structure).

```
bool pop(){
     // removes top element from stack and puts it in
     if( isEmpty() ) { cout << "Stack empty\n"; return false; }
     Top--;
     return true;
}
```

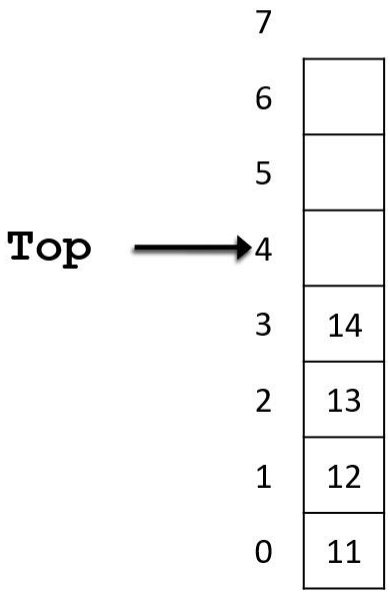


Another useful operation of a stack is to get the top value of the stack. It can be done fairly easily in C++. As `Top-1` index in the `Stack` array stores the value for the top of the stack, we just get the value of that index and then `return` it from the function. The code snippet to this is given below:

```
int topElement(){
     return Stack[ Top - 1 ];
}
```

Last but not the least, we have the operation of showing all the elements in the stack. We do this by printing all the elements in the `Stack` array from the `Top-1` index to the `0` index. We do need to check at first whether the stack is empty. For the below-given illustration, elements of the stack will be printed like `14 13 12 11`.

```
void show(){
     //prints the whole stack from top to bottom
     if(isEmpty()) { cout << "Stack empty\n"; return; }
     for( int i=Top-1; i>=0; i-- ) cout << Stack[i] << endl;
}
```



Considering **MaxSize** = 7
There are 4 elements inside Stack
So top element will be at index 3

Thus far we have explored the implementation of a bounded stack in C++. However, it is not an object-oriented implementation. This limits the use of this implemented stack in several ways. One of them is we have to be contented by just creating only one stack. But if we perform the implementation in an object-oriented manner, we can create multiple stacks by creating multiple objects of the stack class.

The first step in implementing the stack data structure in C++ with object-orientation is, there needs to be a class representing the stack data structure. The code snippet given below for such implementation has the name for that class as `MyStack`. This class will have some member variables such as `Top`, `MaxSize`, and the `Stack` array. Some member functions will also be there. These functions are the ones that we implemented previously for each operation. The newest function that will be added here is the constructor of the class. As we know a constructor initializes certain members of a class when an instance or object is created. In this scenario, when a stack object is created, generally the stack should be empty at the beginning. This will require the constructor to initialize `Top = 0` as well as assigning a size for the stack by initializing `MaxSize`. As we have already declared the `Stack` array with a size earlier, we cannot initialize `MaxSize` with a value more than the size of the array here. The member functions are not fully defined in the below-give skeleton code. Only their prototype is given. It is your task to complete this implementation.

```
class MyStack{
    int Stack[100], Top, MaxSize;
public:
    //Initializing stack
    MyStack( int Size = 100 ){ MaxSize = Size; Top = 0;}
    bool isEmpty();
    bool isFull();
    bool push(int Element);
    bool pop();
    int topElement();
    void show();
};
```

4.1.2 Dynamic Stack

Up to this point in the chapter, we have covered all the basics of a stack and gone through the implementation of it in C++. The implementation was array-based as well as of bounded stack. We are about to enhance the bounded stack to an unbounded (non-bounded) one. That will require the implementation of stack in C++ to be changed a little.

Most of the common functionalities will remain the same as before. We only have to take care of the unboundedness of the stack. And we can achieve that by making sure the array that we are basing the stack on is a *dynamic* one. That is right, we are going to be taking the help of dynamic memory allocation in order to have the capability to make the Stack array unbounded by increasing its size whenever we need in runtime. Below, we can have a look at the skeleton code of such implementation.

```
class MyStack{
    int *Stack, Top, MaxSize;
public:
    MyStack(int);
    ~MyStack();
    bool isEmpty();
    bool isFull();
    bool push(int Element);
    bool pop();
    bool topElement();
    void show();
    void resize( int size); //Resize the stack
};
```

From the above skeleton code of the *dynamic stack* implementation, it is apparent what things are new here. First of all, we are now using a pointer variable `*Stack` for the stack array, not a predefined array with size because dynamic memory allocation deals with the concept of pointer. There is a *destructor* `~MyStack`, which we did not need in earlier implementation. There is also a new function called `resize` which takes a parameter `int size`. Although it is not apparent from the skeleton code, the constructor definition is also different here which we will be dissecting now with the definition of the destructor as well. Another function that will have a slight change in definition is the `push` function.

```
MyStack::MyStack( int Size = 100 ){
    MaxSize = Size; //get Size
    Stack = new int[MaxSize]; //create array accordingly
    Top = 0; //start the stack
}

MyStack::~MyStack(){
    delete [] Stack; //release the memory for stack
}
```

As can be seen from the definition of the constructor above, we are not only initializing `Top` and `MaxSize` like bounded stack implementation but also creating the `Stack` array at runtime with the help of dynamic memory allocation. Whatever we dynamically create, we must delete it at the end of its use. Therefore, in the destructor, we are freeing up the space we created with the `new` operator using the `delete` operator.

Now, let us explore what changes are there to be made for the `push` function. Recall, this is an unbounded stack. That means, it cannot have stack overflow. Whenever there is not space in the stack for the next element to be pushed, we resize the stack by enlarging it with extra size. When it is enlarged (resized), the next element then can be pushed. This way, we avoid stack overflow and it remains unbounded. From this description, we can conclude that the job for enlarging the stack by increasing its size should be a separate function. And we can call it `resize` in this implementation. Simply put, while pushing an element into the stack, if we find the stack is full, we call the `resize` function, and that way the stack is enlarged. Then the new element can be pushed into the stack. The code snippet for the modified push function is given below.

```
void push( int Element ){
    //inserts Element at the top of the stack
    if( isFull( ) )    resize( ); //increase size if full
    Stack[ Top++ ] = Element;
}
```

Now, the only thing that remains, is to implement the `resize` function. Let us take a look at the resize function's definition by checking out the C++ code snippet of it below. Then we can discuss how it works with an illustration that should make things more clear.

```
void resize( int Size = 100 ){
    //creates a new stack with a new capacity, MaxSize + Size
    int *tempStk = new int[ MaxSize + Size ];
    //copy the elements from old to new stack
    for( int i=0; i<MaxSize; i++ ) tempStk[i] = Stack[i];
    MaxSize += Size; //MaxSize increases by Size
    delete [] Stack; //release the old stack
    Stack = tempStk; //assign Stack with new stack
}
```

Let us go through what is happening in the above code snippet by breaking it down step-by-step:

Step 1: A new `tempStk` array is created. We can assume this is a temporary stack. It is larger than our existing `Stack` array (by 100 in this example but that number can be any valid integer).

Step 2: Now, all the elements from the main stack (`Stack` array) are copied to the temporary stack `tempStk`. That means, we have two stacks now, with the same elements but one is full and smaller (`Stack`) in its total size, another is not full and larger (`tempStk`) in its total size.

Step 3: The plan was all along to increase the `MaxSize` of the Stack array so that it can hold more elements later when pushed into it. Therefore, now we increase the size of `MaxSize` with the extra size that we want the main stack (`Stack` array) to hold.

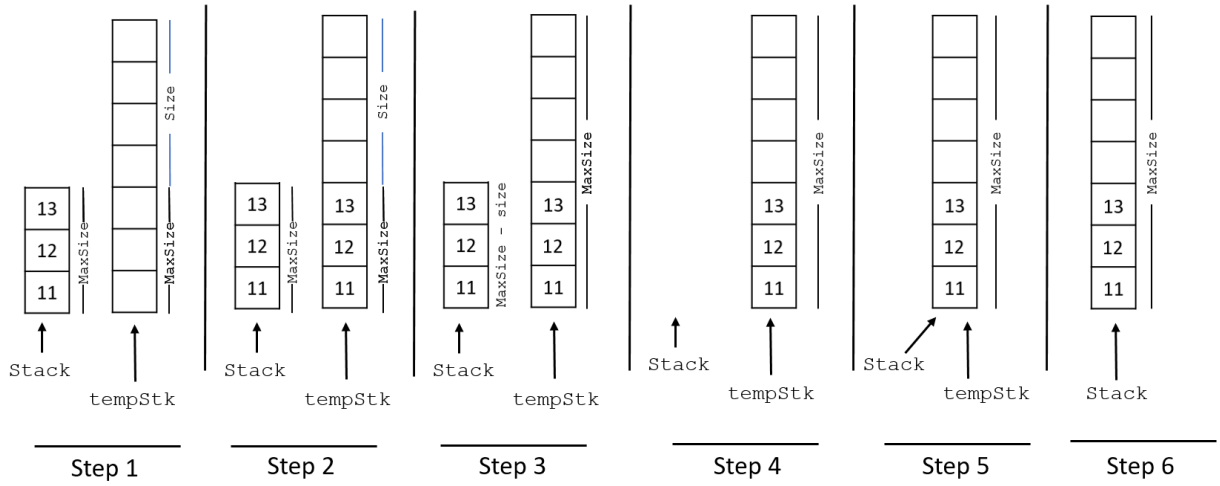
Step 4: As mentioned earlier, we have now two stacks with the same elements. But we do not need the elements of the smaller stack anymore as we have already transferred them to a much larger `tempStack`. Therefore, we release the memory where the `Stack` variable was pointing to using the `delete` operator. Recall, `Stack` is a global pointer variable that was dynamically created as an array in our implementation. Now, the `Stack` pointer is not pointing anywhere as the sequence of memory it was pointing to, was deleted.

Step 5: Both `Stack` and `tempStk` are integer pointer variables. However, `tempStk` is pointing to a sequence of memory holding all the stack elements and some extra memory. `Stack` is pointing nowhere as in the previous step that memory was removed. Recall, `Stack` is global which is working as our main stack in this implementation, and `tempStk` is a local variable to the `resize` function. That means `tempStk` is unavailable to all the functions and everything else in the implementation outside the `resize` function. And as soon as the end of the `resize` function is reached and we get out of it, all the local variable to that function will be destroyed. So, `tempStk` will also be destroyed. It only makes sense to point `Stack` to the memory that the `tempStk` is pointing to. That way, `Stack` will become an array again with all

the elements and the extra memory. That is exactly what happens when the statement `Stack = tempStk` is executed.

Step 6: We get out of the `resize` function and `tempStk` is destroyed but `Stack` remains.

Let us now take a look at an example illustration below depicting the steps above:



By following this above illustration with the steps mentioned earlier, it should be understandable how the stack gets resized.

4.2 Queue

Like stack, queue is also an ADT and is one of the simplest but most useful ones. In ordinary English, a queue is defined as a waiting line, like a line of people waiting to purchase tickets, where the first person in line is the first person served. For computer applications, we similarly define a queue to be a list in which all additions to the list are made at one end (rear), and all deletions from the list are made at the other end (front). Queues are also called first-in, first-out lists, or **FIFO** for short. An example of a real-life queue is given below.

Applications of queues are even more common than applications of stacks, since in performing tasks by computer, as in all parts of life, it is often necessary to wait for one's turn before having access to something. Within a computer system, there may be queues of tasks waiting for the printer, for access to disk storage, or even, with multitasking, for use of the CPU. Within a single program, there may be multiple requests to be kept in a queue, or one task may create other tasks, which must be done in turn by keeping them in a queue. Like a stack, a queue is a holding structure for data that we use later.

The entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front of the queue (or, sometimes, the head of the queue). Similarly, the last entry in the queue, that is, the one most recently added, is called the rear (or the tail) of the queue.

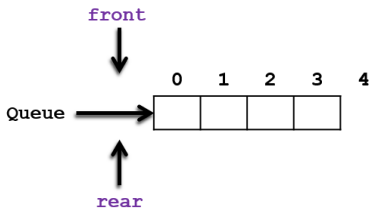
Queue has some similar operations like stack. Adding a new element in the queue is called *enqueue*. Removing an element from the queue is called *dequeue*. While implementing queue in a programming language like C++, some other helping operations are needed. For example, checking whether the queue is empty, or full. Also, we need to get the element that front of the queue for which we can write a function to get it. Then finally we need to view all the elements in the queue starting from front to rear.

We will be focusing on an array-based implementation of queue. Let us assume the array that will be used is called `Queue`. The maximum size of `Queue` is represented by a variable `MaxSize`. To denote the beginning of the queue, let us use a variable called `front`. The job of `front` is to store the index number of the `Queue` array where the first element of the queue resides. And to denote the end of the queue, let us use a variable called `rear`. The job of `rear` is to store the index number of the `Queue` array where the last element of the queue resides. With all these settled, let us take a look at the illustration of operations of *linear queue* below with the pseudo-code for each operation. But first, we have to initialize the queue as empty:

```
Initialize Queue[maxSize]; front = rear = -1;
```

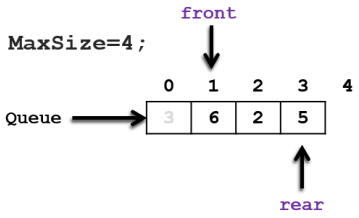
When a queue is empty, the value of `front` and `rear` is set to `-1`. This means there is no element in the `queue` array. Therefore, if we wanted to check whether a queue is empty, we could just check whether both `front` and `rear` have the value of `-1`. This can be done in C++ if the below-given pseudo-code is followed (an illustration of an empty queue is also provided).

```
isEmpty(){
 if ((front==-1) and (rear==-1)):
   then return true;
}
```



Similar to `isEmpty`, we need another function `isFull` for checking whether the queue is full. When the value of `rear` is `1` less than the value of `MaxSize`, we say the queue is full. See the pseudocode with illustration for a better understanding of this.

```
isFull(){
 if rear==(maxSize-1):
   then return true;
}
```



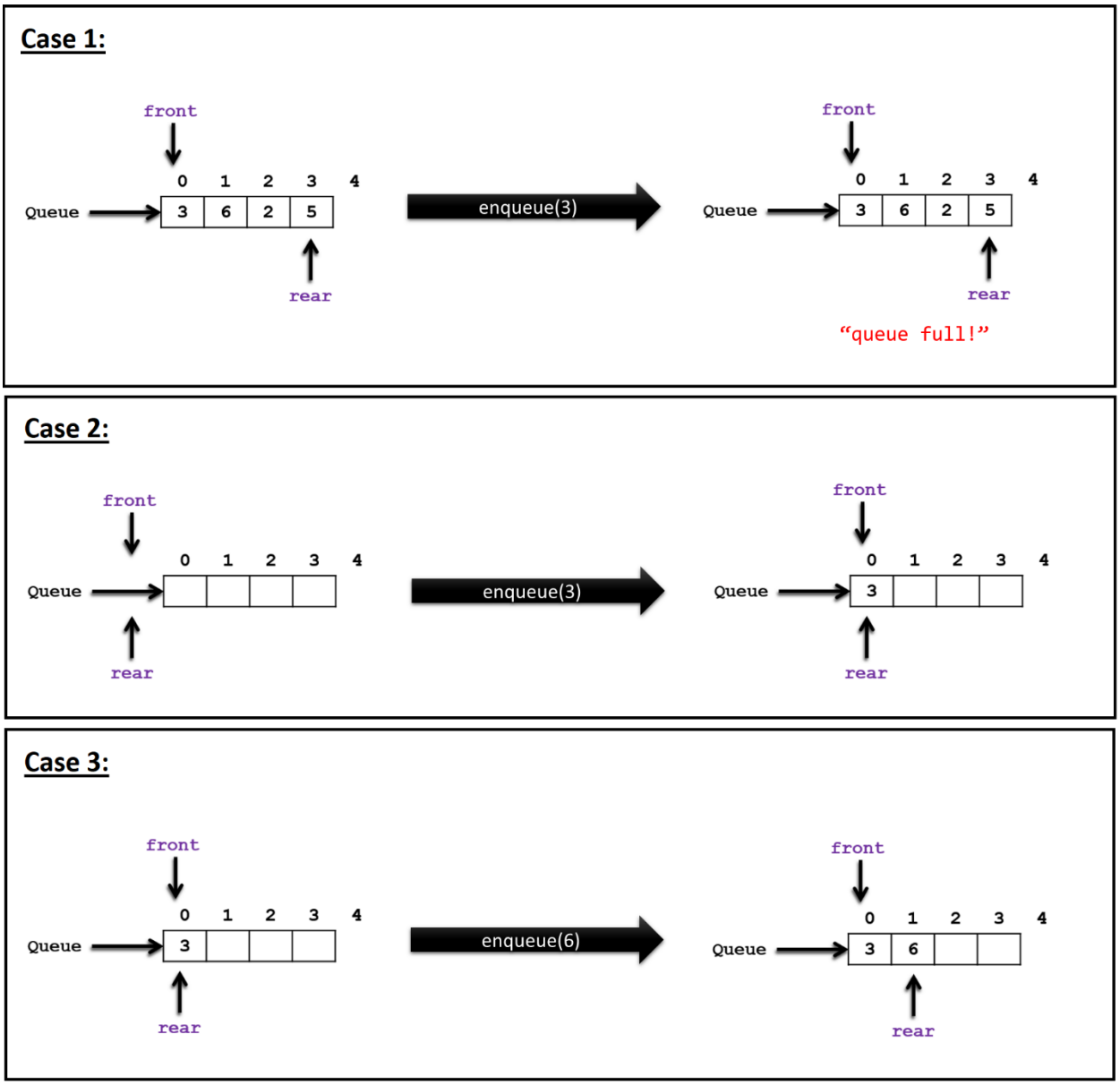
The operation of adding a new element into the queue is called *enqueue*. A new element is added at the end or `rear` of the queue. There are three cases for adding elements into the queue.

Firstly, if the queue is full, as the above illustration, it is not possible to perform enqueue operation. So, nothing gets added to the queue.

Secondly, if the queue is empty, then adding a new element would mean that is the only element in the whole queue. So, `front` and `rear` will point to the same index, which is `0` (zero).

Thirdly, if the queue is neither full nor empty, adding a new element simply means value of `rear` will be increased by 1 and that updated value of `rear` would be the index number where the new element will be added. For these three different cases, three different illustrations are provided below with the help of a pseudocode for the `enqueue` function.

```
enqueue(x){
  if(queue full): {error: "queue full!";}
  otherwise   if(queue   is   empty):   {front=rear=0;   insert   x   in
queue[rear]}
  otherwise: rear++; insert x in queue[rear];
}
```



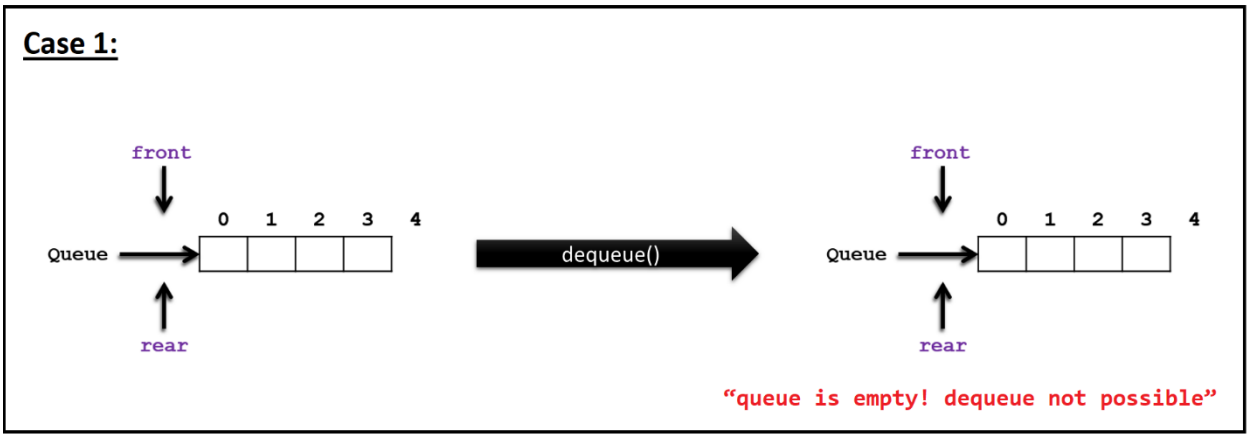
Similar to *enqueue*, there are three cases for *dequeue* too. Recall, removing an element from the queue is called *dequeue*. An element can only be removed from the front of the queue.

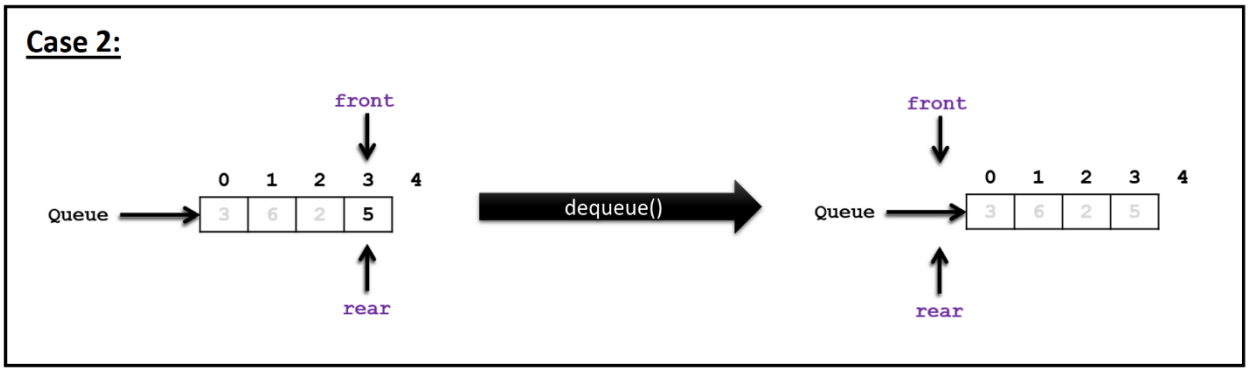
Among the three cases, firstly, if the queue is empty, nothing can be removed from the queue.

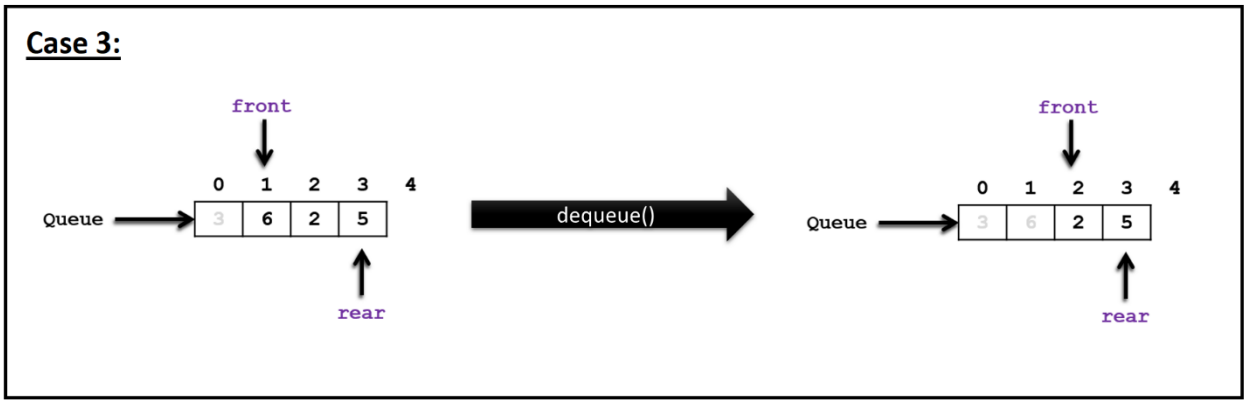
Secondly, if there is only one element in the queue (`front` and `rear` will be pointing to the same index, meaning their value would be the same), after removing that element from the queue, the queue will become empty. If that happens, the statement `front = rear = -1;` will be executed because this is what it means for the queue to be empty.

Thirdly, if there are more than one element in the queue, removing one element from the queue would mean increasing the value of `front` by `1`. See the below-given illustration with the pseudo code for a dequeue function.

```
dequeue(){
  if(queue empty): {error: "queue is empty! dequeue not possible"}
  otherwise if (front and rear are equal): {front=rear=-1;}
  otherwise: {front++;}
}
```



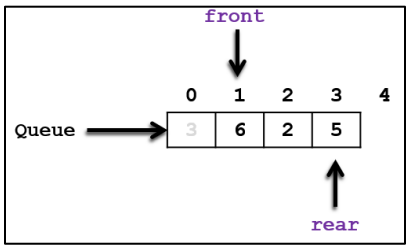




Returning the front element of the queue is also an operation. To perform that, we can write a function `frontElement` like below:

```
frontElement(){
  return queue[front];
}
```

We can do all kinds of operations on a queue, but if we are unable to show/ print all the elements of that queue, it will remain incomplete. To show all the elements of a queue, we can write a function `showQueue` which will print the queue from front to rear. For the following illustration of a queue, if it is printed from front to rear it will print `6 2 5`.

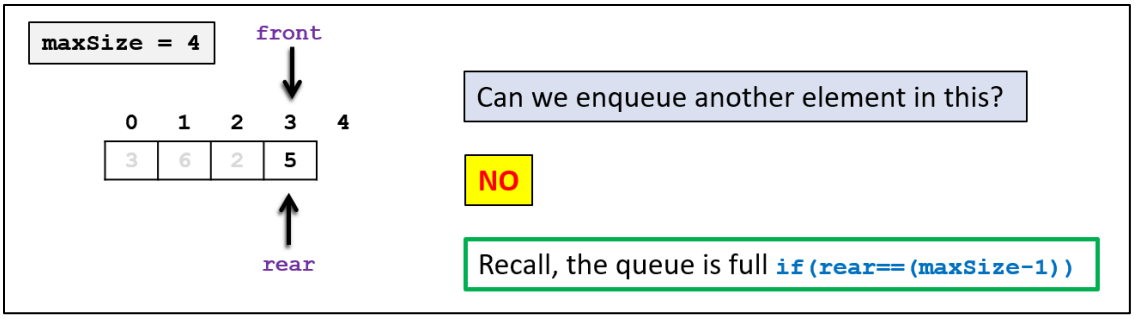



```
showQueue(){
  if (queue empty)
    error: "cannot show queue because it is empty";
  otherwise:
    for: i=front; i<=rear; i++
      output: queue[front];
}
```

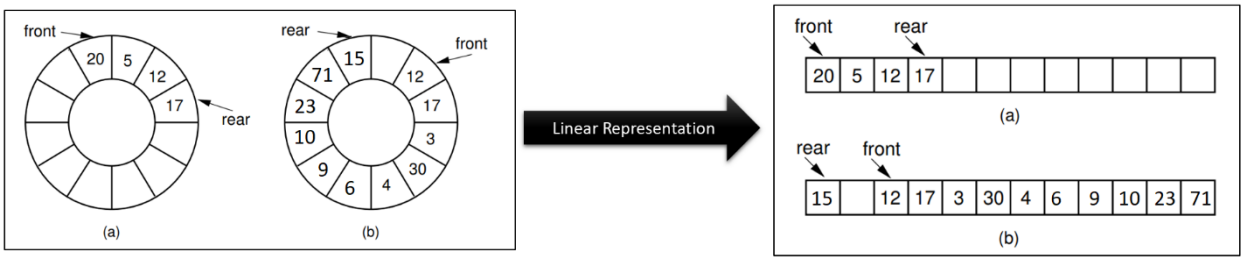
4.3 Circular Queue

The variation of queue data structure that we have studied in the last section is particularly called a *linear queue* (FIFO). That variation raises a new problem. Assume that the first element of the queue is initially at position 0 and that elements are added to successively higher-numbered positions in the array. When elements are removed from the queue, the `front` index increases. Over time, the entire queue will drift toward the higher-numbered positions in the array. Once an element is inserted into the highest-numbered (`maxSize-1`) position in the array, the queue has run out of space.

This happens even though there might be free positions at the low end of the array where elements have previously been removed from the queue. For example, in the illustration below of a linear queue, such a scenario occurs.



Due to this, the previously dequeued positions of the queue remain unused and memory is wasted. Thus, inefficient use of space occurs in the *linear queue*. This "drifting queue" problem can be overcome simply by thinking of the array (that represents the queue) as a circle rather than a straight line (see the illustration below). In this way, as entries are added and removed from the queue, the front will continually chase the rear around the array. At different times, the queue will occupy different parts of the array, but we never need to worry about running out of space unless the array is fully occupied, in which case we truly have an overflow.

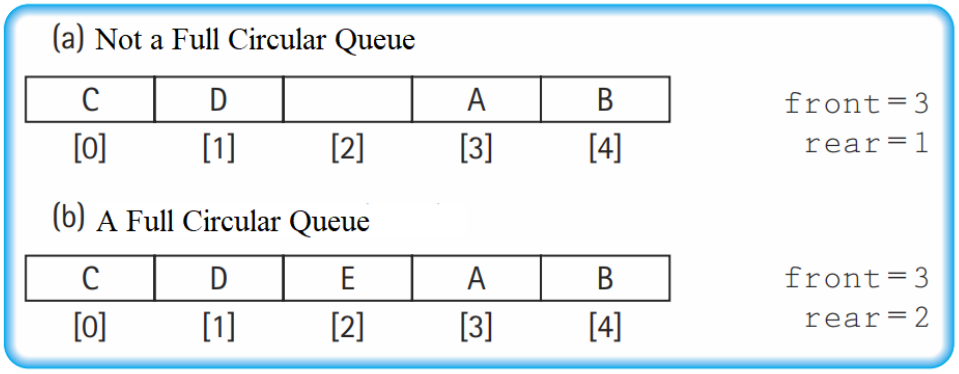


In terms of operations, like the linear queue, circular queue also has the same ones. However, while implementing them in programming language pr pseudo code, some of the definitions for the functions need to be changed because of the structural differences between a circular and a linear queue. There is nothing to be changed in the `isEmpty` and the `frontElement` functions. They work okay for both linear and circular queue just the way they are. However, for the other functions, there are some changes to be made.

Let us first start with what it means for a circular queue to be full. In linear queue, we learned that if `(rear == (maxSize-1))` is `true`, then it is full. However, the circular queue does not rely on the highest or maximum index of the array for the end of the queue. As we learned above (with some illustrations), circular queue can circle back to the beginning of the array and fill-up unused spaces. Therefore, in the array, sometimes `rear` can appear before `front`. So, how do we measure fullness of a circular queue? We do that by changing the statement `(rear == (maxSize-1))` to `(((rear+1)%maxSize) == front)` in the definition of the `isFull` function. Now this version of the function works for the circular queue.

```
isFull(){
 if (((rear+1)%maxSize)==front):
   then return true;
```

```
}
```



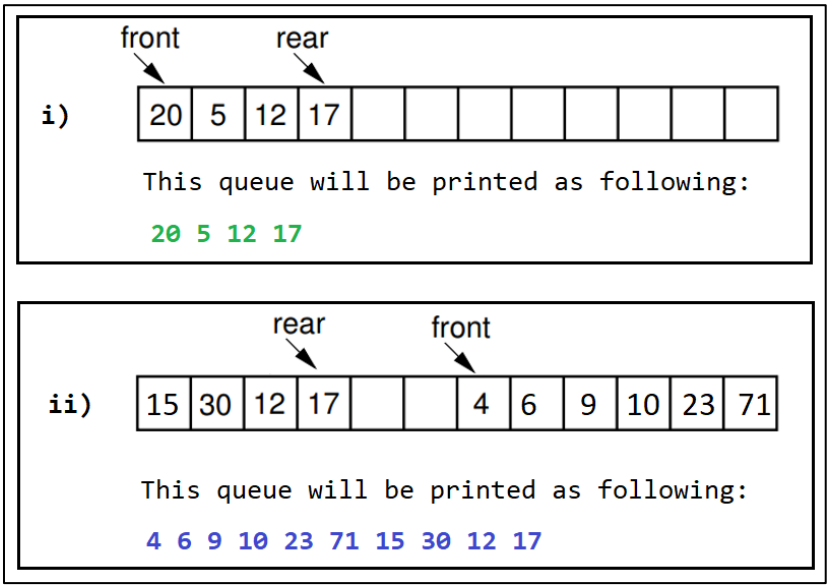
Slight changes are also in order for the functions `enqueue` and `dequeue`. As in circular queue, there is a chance of circling back to the beginning of the array, we are going to take the help of the module (`%`) operator to change certain statements in the previous definitions of `enqueue` and `dequeue` to make them suitable for circular queue. Instead of `rear++` in enqueue, we are going to use the statement `rear = (rear + 1)%maxSize`. And instead of using `front++` in dequeue, we are going to use the statement `front = (front+1)%maxSize`. Therefore, these function definitions can be re-written followingly for circular queue.

```
enqueue(x){
  if(queue full): {error: "queue full!";}
  otherwise   if(queue   is   empty):   {front=rear=0;   insert   x   in
queue[rear]}
  otherwise: rear=(rear+1)%maxSize; insert x in queue[rear];
}
```

```
dequeue(){
  if(queue empty): {error: "queue is empty! dequeue not possible"}
  otherwise if (front and rear are equal): {front=rear=-1;}
  otherwise: {front=(front+1)%maxSize;}
}
```

For printing all the elements of the linear queue, we defined the `showQueue` function earlier. The definition of this also needs to be modified to make it support the printing for a circular queue. Why do we need to change it? Because, in the circular queue, there are mainly two cases when it comes to printing all the elements of the queue: i) when `front` index is before or the same as `rear` index, ii) when `rear` index is before `front` index. We can see how both cases can be encoded in the pseudo-code and an illustration.

```
showQueue(){
  if (queue is empty)
    error: "cannot show queue because it is empty!";
  otherwise:
    if(front <= rear):
      for: i=front; i<=rear; i++
        output: queue[front];

    otherwise:
      for: i=front; i<=(maxSize-1); i++
        output: queue[front];

      for: i=0; i<=rear; i++
        output: queue[front];
}
```



4.4 Stack & Queue Applications

Now that we know how stack and queue ADTs work, we need to apply them appropriately in different problem-solving scenarios in computer science. We can use stack to solve problems such as syntax parsing, parentheses check, expression evaluation and expression conversion, banking transaction view, backtracking and implementation of recursive function, calling function, towers of Hanoi, undo mechanism in text editors, etc. Some of the applications of queue are keeping track of printing jobs, CPU task scheduling, in any customer service solution, etc. Queue can also be used alongside stack for expression evaluation and conversion. In this

chapter, we will particularly be concentrated on algebraic expression conversion and evaluation (infix to postfix and postfix evaluation). We will also be learning how applying stack can detect correct and incorrect use of parentheses in an expression.

4.4.1 Algebraic Expression

Let us revisit the land of algebraic expressions shortly before we jump into their conversion or evaluation. An algebraic expression is a legal combination of operands and operators. An operand is a quantity (unit of data) on which a mathematical operation is performed. An operand can be a variable like x,y,z, or a constant like 5,4,0,9,1, etc. An operator is a symbol that signifies a mathematical or logical operation between the operands. Example of familiar operators include +, -, *, /, ^, %. Considering these definitions of operands and operators now we can write an example of expression as `x + y * z`.

4.4.2 Infix, Postfix, and Prefix Expressions

The expressions in which operands surround the operator, i.e. the operator is in between the operands. For example, `x+y`, `6*3` etc. The infix notation is the general way we write an expression. To our surprise INFIX notations are not as simple as they seem especially while evaluating them. To evaluate an infix expression we need to consider Operators' Precedence and Associative property. For example expression 3+5*4 evaluates to:

```
32 = (3+5)*4 - Wrong
```
or
```
23 = 3+(5*4) - Correct
```

Operator precedence and associativity govern the evaluation order of expression. An operator with higher precedence is applied before an operator with lower precedence. The same precedence order operator is evaluated according to their associativity order. C++ has its operator precedence and associativity chart[1].

Postfix is also known as Reverse Polish Notation (RPN) where the operator comes after the operands, i.e. operator comes post of the operands, so the name postfix. e.g. xy+, xyz+*, etc.

Prefix is also known as Polish Notation (PN) where the operator comes before the operands, i.e. operator comes pre of the operands, so the name prefix. e.g. +xy, *+xyz, etc.

For computers, infix expressions are hard to parse because operator priorities, tiebreakers, and delimiters are needed. This makes the evaluation of expression more difficult than is necessary for the processor. Both prefix and postfix notations have an advantage over infix that while evaluating an expression in prefix or postfix form we need not consider the Precedence and Associative property. The expression is scanned from the user in infix form; it is converted into prefix or postfix form and then evaluated without considering the parentheses and priorities of the operators. So, it is easier (complexity wise) for the processor to evaluate expressions that

[1] https://en.cppreference.com/w/cpp/language/operator_precedence

are in these forms. Let us juggle our memory a little by going through the following examples of conversions from infix to postfix and prefix and their evaluations.

Infix	PostFix	Prefix
A+B	AB+	+AB
(A+B) * (C + D)	AB+CD+*	*+AB+CD
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

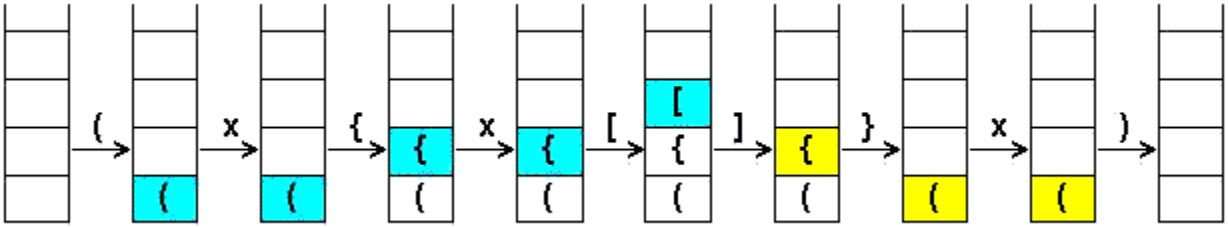
A-B/(C*D^E)	ABCDE^*/-		-A/B*C^DE	
A-B/(C*F)	ABCF*/-	D^E	-A/B*CF	D^E
A-B/G	ABG/-	C*F	-A/BG	C*F
A-H	AH-	B/G	-AH	B/G
I	I	A-H	I	A-H

4.4.3 Parentheses Check Using Stack

Using stack, we can check whether an expression has its parenthesis properly placed; i.e., whether it is opening and closing parentheses match with each other. For example, let's take the expression `(x{x[]}x)`. We will read the expression as a string starting from the first character to the last and for each character, we will do the following three things:

- Whenever we get an opening parenthesis, we will push it into the stack.
- When we get a closing parenthesis we will check that with the top of the stack. If the top of the stack has the same type of opening parenthesis, we will pop it.
- We skip the character in the string which is not a parenthesis.

Finally, if we reach the end of the expression and the stack is also empty, that means the expression is "well-formed". In any other case, the expression is "not well-formed". Let us take a look at the illustration of this process below for the parentheses check for: `(x{x[]}x)`.



This expression is well-formed.

4.4.4 Infix to Postfix Conversion

In this section of the chapter, we are going to learn the technique of converting an infix expression to a postfix one. We will be using a stack to process the input infix expression (which

will be input as a string) and use a queue to store the resultant postfix expression. This can be achieved by following the pseudo code given below:

```
Add ')' to the end of Infix;     Push( '(' );
do{
  OP = next symbol from left of Infix;
  if OP is OPERAND then EnQueue( OP );
  else if OP is OPERATOR then{
    if OP = '(' then Push( OP );
    else if OP = ')'  then{
        while TopElement() != '(' do{
          Enqueue(TopElement());
          Pop();
        }
          Pop();
    }else{
    while Precedence( OP ) <= Precedence( TopElement() ) do{
          Enqueue(TopElement());
          Pop();
    }
    Push( OP );
    }
}while !IsEmpty();
```

We can translate the above-mentioned pseudo-code in English following some steps:

a) Add a parenthesis ‘)’ to the end of the infix expression we took as an input string (let us call it **I**).
b) Push a parenthesis ‘(’ in the stack (let us call it **S**).
c) Now we start reading from the start of **I** character by character. For each character, we will be following the instructions of the next steps.
d) If the character is an *operand*, we enqueue it to the output postfix queue (let us call it **Q**).
e) Otherwise, if it is an *operator*, then we check which kind of *operator* it is.
    - If it is an opening parenthesis ‘(’ then we push it in **S**.
    - Otherwise, if it is a closing parenthesis ‘)’ then we do the following:
        - enqueue the top element of **S** into **Q**
        - pop it from **S**
        - repeat this process until we find the top element of **S** to be an opening parenthesis ‘(’
    - Otherwise, we check the precedence of it with the precedence of the element top of **S** (let us call it **t**). Then we do the following:
        - if the precedence of **t** is higher or equal, enqueue **t** to **Q** and pop it.
        - repeat this process until the precedence of **t** is lower. Then we push the operator (that we were reading from the infix expression) in **S**.
f) When the stack is empty, we are done converting the infix expression to a postfix one. When we print **Q** we will have found the postfix expression.

We can take a look at the following example (simulation) of converting an infix expression to a postfix one using the above-mentioned technique. Here, infix expression: `2*6/(4-1)+5*3`. If we add ')' at the end of it: `2*6/(4-1)+5*3)`, then a '(' will be added to stack from the beginning for that.

Symbol	Stack	Postfix (Queue)
2	(	2
*	(*	2
6	(*	26
/	(/	26*
(	(/(	26*
4	(/(	26*4
-	(/(-	26*4
1	(/(-	26*41
)	(/	26*41-
+	(+	26*41-/
5	(+	26*41-/5
*	(+*	26*41-/5
3	(+*	26*41-/53
)		26*41-/53*+

Suffice to say, this conversion technique will only work for single character operands when implemented using a programming language.

4.4.5 Evaluation of Postfix Expression

The technique of evaluating a postfix expression in mathematics is shown in section 4.3.2 of this chapter. However, in this section, we will be using a stack in combination with a queue for the evaluation. We will follow the below-given pseudo code for this. This technique will only work for single character operands when implemented using a programming language.

```
Postfix Expression: 26*41-/53*+
EnQueue( ')' );
while ( FrontElement() != ')' ) do{
    OP = FrontElement();
    DeQueue();
    if OP is OPERAND then Push( OP );
    else if OP is OPERATOR then{
          OperandRight = TopElement();
          Pop();
          OperandLeft = TopElement();
          Pop();
          x = Evaluate(OperandLeft, OP, OperandRight);
          Push(x);
    }
}
Result = TopElement();
Pop();
cout << Result;
```

An example simulation of how we can use the above-mentioned technique in practice can be seen below. Let us take the postfix expression `2,6,*,4,1,-,/,5,3,*,+` for evaluation. This postfix expression is treated as a queue, where each symbol of it is a queue element. We are using commas to separate the numbers so that there is no confusion.

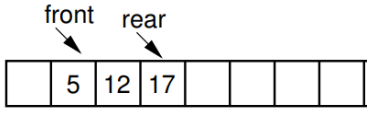
Symbol	Stack
2	2,
6	2,6
*	12
4	12,4
1	12,4,1
-	12,3
/	4
5	4,5
3	4,5,3
*	4,15
+	19
)	19

Result =19

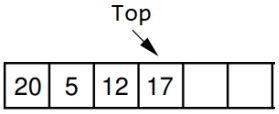
4.5 Exercises

1. Implement a Generic Stack in C++.
2. With the help of the pseudo-code available for different operations of the *linear queue*, implement it using object orientation in C++.

3. With the help of the pseudo-code available for different operations of the *circular queue*, implement it using object orientation in C++.
4. Explain the differences between Stack and Queue with appropriate examples.
5. Explain the need for a *circular queue* with appropriate examples.
6. You are given a *linear queue* of maximum size = 8. The state of the queue currently is:



   Perform simulation on this queue for the following operations:
   enqueue(3), dequeue(), enqueue(53), dequeue(), enqueue(23), dequeue(), enqueue(33), dequeue(), enqueue(32), enqueue(13).
7. Now assume the queue given in question **6** is circular. Perform the simulation of question **6** on it.
8. You are given a bounded stack of maximum size = 6. The state of the stack currently is:



   Perform simulation on this stack for the following operations:
   push(21), pop(), pop(), pop(), push(22), pop(), push(92), push(23), pop(), pop(), pop().
9. Show the simulation of coverting each of the following infix expressions to postfix:
   - `a+(b*((c-d)/c))-(x+y*z)`
   - `(a*b*c)/(d+e+f)-x-y+z`
   - `((a/b)+(c*d))*((a*b)-(c/d))`
10. Show the simulation for evaluating each of the following postfix expressions (operators and operands are comma-separated for clarity):
    - `7,4,3,*,1,5,+,/,*`
    - `5,7,+,6,2,-,*`
    - `4,2,+,3,5,1,-,*,+`
    - `5,3,2,*,+,4,-,5,+`

4.6 References

1. Data structures and Program Design in C++ by Robert L.Kruse, Alexander J.Ryba. Chapter 2, 3.
2. Nell Dale - C++ Plus Data Structures, Third Edition (2002, Jones and Bartlett Publishers, Inc.). Chapter 4.
3. Dr. Clifford A. Shaffer - Data Structures and Algorithm Analysis in C++, 3rd Edition -Dover Publications (2011). Chapter 4.2, 4.3.
4. "Advanced Data Structures", Peter Brass, Cambridge University Press, 2008. [Chapter 1: 1.1, 1:1.2]
5. https://en.wikipedia.org/wiki/Stack_(abstract_data_type)
6. https://www.cs.usfca.edu/~galles/visualization/StackArray.html
7. https://en.wikipedia.org/wiki/Queue_(abstract_data_type)
8. https://www.cs.usfca.edu/~galles/visualization/QueueArray.html
9. https://en.wikipedia.org/wiki/Circular_buffer

10. http://www.cs.uregina.ca/Links/class-info/210/Stack/
11. http://www.cs.csi.cuny.edu/~zelikovi/csc326/data/assignment5.htm

Chapter 5

Sorting and Searching

We sort many things in our everyday lives: A handful of cards when playing Bridge; bills and other piles of paper; jars of spices; and so on. And we have many intuitive strategies that we can use to do the sorting, depending on how many objects we have to sort and how hard they are to move around. Sorting is also one of the most frequently performed computing tasks. We might sort the records in a database so that we can search the collection efficiently. We might sort the records by zip code so that we can print and mail them more cheaply. . We might use sorting as an intrinsic part of an algorithm to solve some other problem, such as when computing the minimum-cost spanning tree.

5.1 Definition of Sorting

Given a set of records $r_1, r_2, \ldots, r_n$ with key values $k_1, k_2, \ldots, k_n$, the Sorting Problem is to arrange the records into any order s such that records $r_{s1}, r_{s2}, \ldots, r_{sn}$ have keys obeying the property $k_{s1} \leq k_{s2} \leq \ldots \leq k_{sn}$. In other words, the sorting problem is to arrange a set of records so that the values of their key fields are in non-decreasing order.

When comparing two sorting algorithms, the most straightforward approach would seem to be simply program both and measure their running times. When analyzing sorting algorithms, it is traditional to measure the number of comparisons made between keys.

5.2 Applications of Sorting

Commercial computing: Organizations organize their data by sorting it. Accounts to be sorted by name or number, transactions to be sorted by time or place, mail to be sorted by postal code or address, files to be sorted by name or date, or whatever, processing such data is sure to involve a sorting algorithm somewhere along the way.

Search for information: Keeping data in sorted order makes it possible to efficiently search through it using the classic binary search algorithm.

Operations research: Suppose that we have N jobs to complete. We want to maximize customer satisfaction by minimizing the average completion time of the jobs. The shortest processing time first rule, where we schedule jobs in increasing order of processing time, is known to accomplish this goal.

Combinatorial search: A classic paradigm in artificial intelligence is to define a set of configurations with well-defined moves from one configuration to the next and a priority associated with each move.

Prim's algorithm, Kruskal's algorithm and Dijkstra's algorithm are classical algorithms that process graphs. These use sorting.

Huffman compression is a classic data compression algorithm that depends upon processing a set of items with integer weights by combining the two smallest to produce a new one whose weight is the sum of its two constituents.

String processing algorithms are often based on sorting.

5.3 Bubble Sort

Bubble Sort is often taught to novice programmers in introductory computer science courses. It is a relatively slow sort. It has a poor best-case running time.

Bubble Sort consists of a simple double for loop. The first iteration of the inner for loop moves through the record array from left to right, comparing adjacent keys. If the lower-indexed key's value is greater than its higher-indexed neighbor, then the two values are swapped. Once the largest value is encountered, this process will cause it to "bubble" up to the right of the array. The second pass through the array repeats this process. However, because we know that the largest value reached the right of the array on the first pass, there is no need to compare the right two elements on the second pass. Likewise, each succeeding pass through the array compares adjacent elements, looking at one less value than the preceding pass. The following figure[2] shows an example of Bubble Sort:

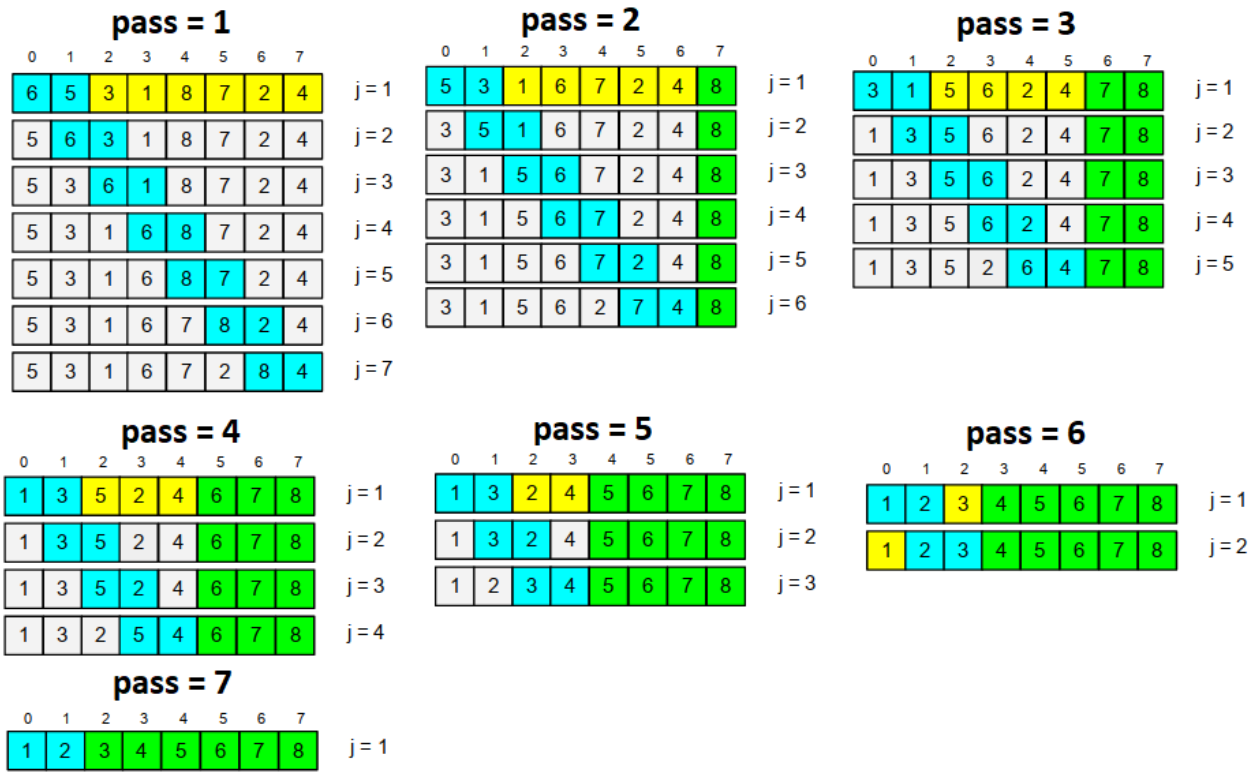


Figure 5.0.1: An illustration of Bubble Sort to sort an array in increasing order

[2] https://eleni.blog/2019/06/09/sorting-in-go-using-bubble-sort/

```
Algorithm (Bubble Sort):
Input: A (array), N (#elements)
pass =1
Step 1: u = N- pass
Compare the pairs (A[0], A[1]), (A[1], A[2]), (A[2], A[3]), … ,
(A[u-1], A[u]) and Exchange pair of elements if they are not in
order.
Step 2: pass = pass + 1. If the array is unsorted and pass < N-1
then go to step 1
```

It is to notice that if in a pass no swapping occurs then next passes are redundant. In the above illustration, it is seen that pass 7 is redundant since there was no swapping in pass 6.

Not much use in the real world, but it's easy to understand and fast to implement. It is used when a fast algorithm is needed to sort: 1) an extremely small set of data (Ex. Trying to get the books on a library shelf back in order.) or 2) a nearly sorted set of data. (Ex. Trying to decide which laptop to buy, because it is easier to compare pairs of laptops one at a time and decide which you prefer, than to look at them all at once and decide which was best.)

5.4 Selection Sort

Consider the problem of sorting a pile of phone bills for the past year. An intuitive approach might be to look through the pile until you find the bill for January, and pull that out. Then look through the remaining pile until you find the bill for February, and add that behind January. Proceed through the ever-shrinking pile of bills to select the next one in order until you are done. The *ith* pass of Selection Sort "selects" the *i*th smallest key in the array, placing that record into position *i*. In other words, Selection Sort first finds the smallest key in an unsorted list, then the second smallest, and so on. Its unique feature is that there are few record swaps. To find the next smallest key value requires searching through the entire unsorted portion of the array, but only one swap is required to put the record in place. Thus, the total number of swaps required will be $n - 1$ (we get the last record in place "for free").

Selection Sort (as written here) is essentially a Bubble Sort, except that rather than repeatedly swapping adjacent values to get the next smallest record into place, we instead remember the position of the element to be selected and do one swap at the end. Selection sort is particularly advantageous when the cost to do a swap is high, for example, when the elements are long strings or other large records. Selection Sort is more efficient than Bubble Sort (by a constant factor) in most other situations as well.

There is another approach to keeping the cost of swapping records low that can be used by any sorting algorithm even when the records are large. This is to have each element of the array store a pointer to a record rather than store the record itself. In this implementation, a swap operation need only exchange the pointer values; the records themselves do not move. The following figure shows an example of Selection Sort:

1	2	3	4	5	6	7	8	9	Remarks
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap a[i] & a[j]
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap a[i] and a[j]
45	**50**	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap a[i] and a[j]
45	**50**	**55**	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap a[i] and a[j]
45	**50**	**55**	**60**	70	80	75	85	65	Find the fifth smallest element
				i				j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	80	75	85	70	Find the sixth smallest element
					i			j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	75	85	80	Find the seventh smallest element
						i j			swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	85	80	Find the eighth smallest element
							i	J	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	**80**	**85**	The outer loop ends.

Figure 5.0.2: An illustration of Selection Sort to sort an array in increasing order

Algorithm (Selection Sort):
Input: A (array), N (#elements)

$i = 0$

Step 1: Find the smallest element from the positions starting at index i to $N-1$. Swap the smallest element with the element at index i.

Step 2: $i = i + 1$. If $i < N - 1$ go to step 1

5.5 Insertion Sort

Consider again the problem of sorting a pile of phone bills for the past year. A fairly natural way to do this might be to look at the first two bills and put them in order. Then take the third bill and put it into the right order with respect to the first two, and so on. As you take each bill, you would add it to the sorted pile that you have already made. This naturally intuitive process is the inspiration for our first sorting algorithm, called **Insertion Sort**. Insertion Sort iterates through a list of records. Each record is inserted in turn at the correct position within a sorted list composed of those records already processed.

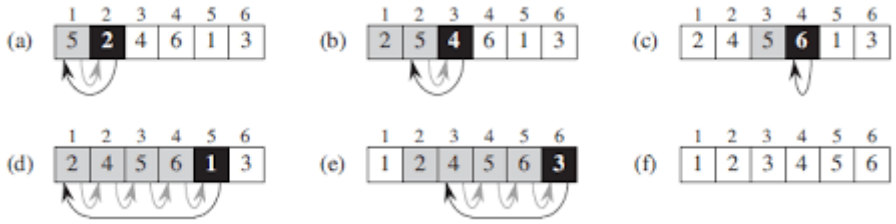


Figure 5.0.3: An illustration of Insertion Sort

It is to notice that a list is correctly sorted as quickly as can be done. Furthermore, insertion sort remains an excellent method whenever a list is nearly in the correct order and few entries are many positions away from their correct locations.

```
Algorithm (Insertion Sort):
Input: A (array), N (#elements)
i = 1
Step 1: v = A[i]
Compare v backwards with all previous (down to 0 index) elements.
If a previous element is larger shift it forward otherwise stop
comparing.
Step 2: i = i + 1. If i < N go to step 1.
```

5.6 Searching

Organizing and retrieving information is at the heart of most computer applications, and searching is surely the most frequently performed of all computing tasks. Search can be viewed abstractly as a process to determine if an element with a particular value is a member of a particular set. The more common view of searching is an attempt to find the record within a collection of records that has a particular key value, or those records in a collection whose key values meet some criterion such as falling within a range of values.

We can define searching formally as follows. Suppose that we have a collection **L** of *n* records of the form

$$(k_1, I_1), (k_2, I_2), \ldots, (k_n, I_n)$$

Where I_j is information associated with key k_j from record j for $1 \leq j \leq n$. Given a particular key value K, the **search problem** is to locate a record (k_j, I_j) in **L** such that $k_j = K$ (if one exists). **Searching** is a systematic method for locating the record (or records) with key value $k_j = K$.

A **successful** search is one in which a record with key $k_j = K$ is found. An **unsuccessful** search is one in which no record with $k_j = K$ is found (and no such record exists).

Searching in array are basically two types name sequential or linear search and binary search. Sequential or linear search is applied when a given array is unsorted but it works for

sorted array also. On the other hand, the binary search is applicable if the array is sorted. Both the sorting algorithms are described details in the following sections.

5.7 Sequential or Linear Search

Beyond doubt, the simplest way to do a search is to begin at one end of the list and scan down it until the desired key is found or the other end is reached. The simplest form of search is sequential or linear search which has already been presented in **Chapter 2**. In linear search, the major consideration is whether *K* is in list **L** at all. Thus, we have $n + 1$ distinct possible events: That *K* is in one of positions 0 to $n - 1$ in **L** (each position having its own probability), or that it is not in **L** at all.

```
Algorithm (Linear Search):
Input: Array, #elements, item (to search)
Start with the element at index = 0
Step 1: Compare the element at index with item. If it is equal
to item then return index with status "Found" otherwise go to
step 2.
Step 2: Increase index by 1. If index is less than #elements go
to step 1 otherwise return -1 with status "Not found".
```

5.8 Binary Search

Sequential search is easy to write and efficient for short lists, but a disaster for long ones. Imagine trying to find the name "Amanda Thompson" in a large telephone book by reading one name at a time starting at the front of the book! To find any entry in a long list, there are far more efficient methods, provided that the keys in the list are already **sorted** into order.

One of the best methods for a list with keys in order is first to compare the target key with one in the center of the list and then restrict our attention to only the first or second half of the list, depending on whether the target key comes before or after the central one. With one comparison of keys we thus reduce the list to half its original size. Continuing in this way, at each step, we reduce the length of the list to be searched by half. In only twenty steps, this method will locate any requested key in a list containing more than a million keys.

The method we are discussing is called binary search. This approach of course requires that the keys in the list be of a scalar or other type that can be regarded as having an order and that the list already be completely in order. The following example (in figure 5.4) illustrates the simulation of Binary Search.

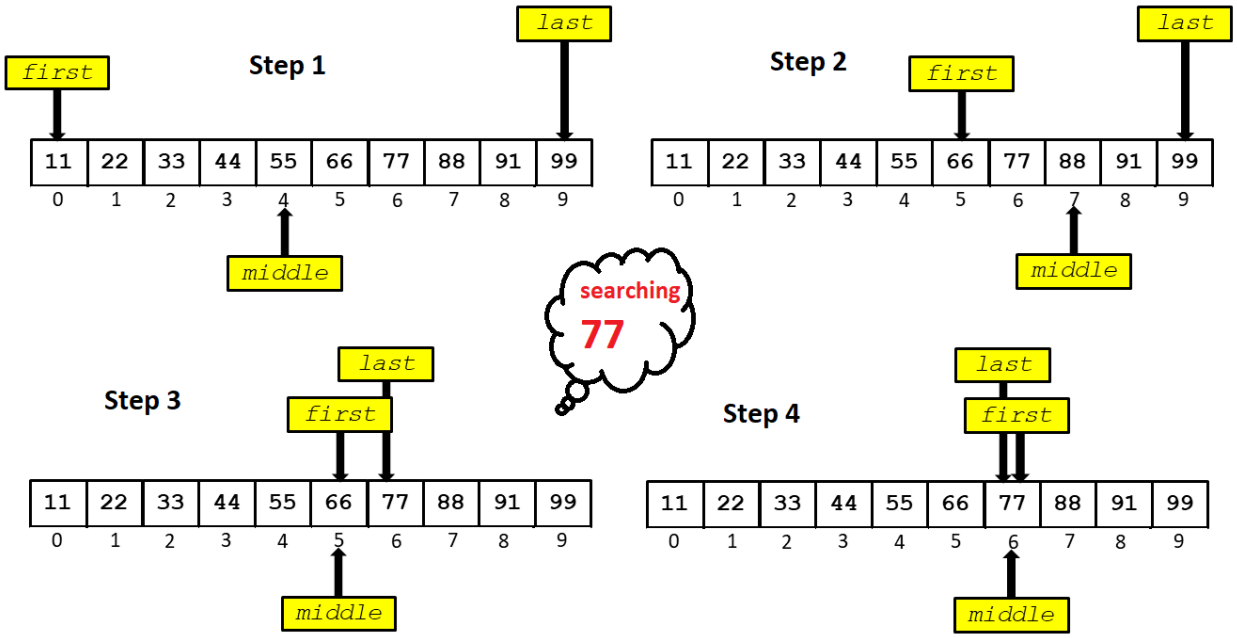


Figure 5.4: An example of Binary Search

The binary search discussed here is appropriate only for list elements stored in a sequential array-based representation. After all, how can you efficiently find the middle point of a linked list? However, you will know of a structure that allows you to perform a binary search on a linked data representation, the binary search tree. The operations used to search in a binary search tree will be discussed in **Chapter 7**.

```
Algorithm (Binary Search):
Input: Array, #elements (N), value(to search)
first= 0 and last= N-1
Step 0:
middle = ⌊(first + last)/2⌋
Step 1:
If low>high exit with status Not Found.
If Array[middle] = value then return middle with status "Found"
If Array[middle] < value then update first by middle+1 and repeat step
0.
If Array[middle] > value then update last by middle − 1 and repeat step
0.
```

5.9 References

- Kruse, R. L., Ryba, A. J., & Ryba, A. (1999). Data structures and program design in C++ (p. 280). New York: Prentice Hall.
- Shaffer, C. A. (2012). Data structures and algorithm analysis. Update, 3, 0-3.

- Dale, N. B. (2003). C++ plus data structures. Jones & Bartlett Learning.

Chapter 6

Linked List

Linked list is a data structure consisting of a group of memory space which together represents a list i.e. a sequence of data. Each data is stored in a separate memory space/block (called cell/node). Each memory block contains the data along with link/location/address to the memory location for the next data in the list. The linked list uses dynamic memory allocation, that is, it allocates memory for new list elements as needed.

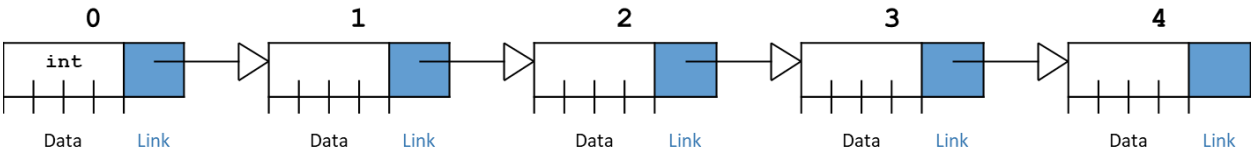


Figure 6.5: Linked List

Figure 6.1 shows a graphical depiction for a linked list storing five integers (Data). The value stored in a pointer variable (Link) is indicated by an arrow "pointing" to something.

6.1 Array and Linked List

A sequence of data can also be represented as an array. But in an array, data are stored consecutively in the memory. A linked list also represents and stores a sequence of data. But in a linked list the data may not be stored consecutively in the memory. Figure 6.2 shows a depiction of array and linked list representation. Same sequence of five values is stored in both an array and a linked list. In array, values are stored in continuous memory locations (FF00 to FF04) whereas in the linked list values are stored in random memory locations.

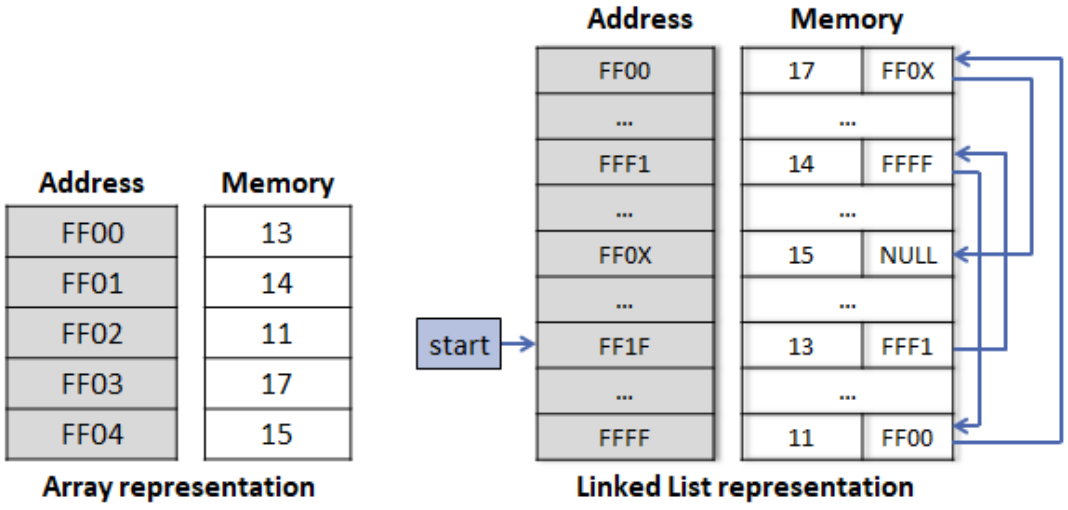


Figure 6.6: Array and Linked List representation in memory

When comparing array and linked list array allows direct access value at specific position whereas linked list requires sequential access. As an example, if we want to access the value17 from the linked list we also need to access all the previous values (ie. 13, 14 and 11).

6.2 Applications of Linked List in Computer Science

Implementation of stacks and queues: Stack and Queue both can be implemented using linked list. In fact the linked list implementation is more efficient in compare to array based implementation of Stack and Queue.

Implementation of graphs: Adjacency list representation of graphs is most popular which uses linked list to store adjacent vertices.

Dynamic memory allocation: We use linked list of free blocks which allows memory allocation on demand.

Image viewer: Previous and next images are linked, hence can be accessed by next and previous button.

Previous and next page in web browser: We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.

Music Player: Songs in music player are linked to previous and next song. You can play songs either from starting or ending of the list.

There are many other applications such as maintaining directory of names, performing arithmetic operations on long integers, manipulation of polynomials by storing constants in the node of linked list and representing sparse matrices etc.

6.3 Representation of a linked list node

A linked list node can be defined using structure. The structure is named like a normal variable which contains **data** and **link** to the next node (here it is named as next).

```
struct ListNode{
        int data;
        ListNode *next;
};
ListNode node;
```

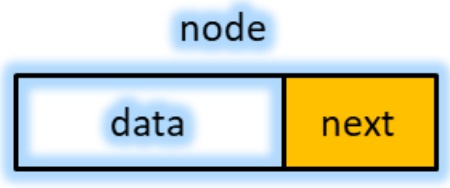


Figure 6.7: Representation of a linked list node in C/C++

Notice that node stores two types of information namely data and address to next node. In a linked list, there will be many nodes which will be connected thorough the addresses (links). A node variable can be created using the structure name. In Figure 6.3, **node** is a variable of type **ListNode**.

6.4 Linked List Traversal

Traversal is a basic operation on any data structure which requires accessing all data elements stored in the data structure. As we mentioned earlier in the chapter that linked list doesn't allow direct access though.

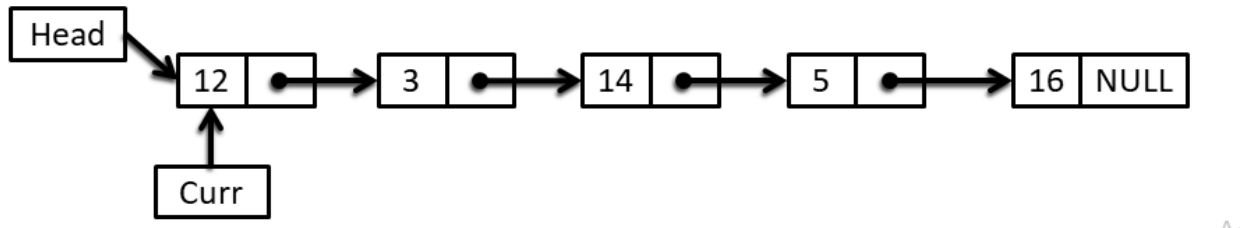


Figure 6.8: Linked List Traversal

Figure 6.4 shows a linked list which has 5 elements stored. Since linked list is always given with the address of first node we need to start from the first node only. A temporary pointer **Curr** is used to store address of visiting node. Initially the **Curr** will point to the first node (here the node containing 12). In the next step, we need to visit second node (the node containing 3). Since the first node (now pointed by **Curr**) contains the address of second node we can update or forward the **Curr** pointer to second node with the following operations:

Curr = Curr →next

Now **Curr** points to second node. In the same manner we can update **Curr** and visit to next nodes (third node, fourth node and fifth node). When **Curr** points to fifth node and we update it **Curr** will certainly become **NULL** which indicates the end of the linked list. The algorithm for linked list traversal is given as follows:

```
Algorithm (Linked List Traversal):
Input: Head (the address of first node)
Curr = Head
Step 1: if Curr == NULL exit otherwise access current node (with
address Curr)
Step 2: move Curr to next node and go to step 1
```

6.5 Searching in Linked List

Searching in linked list involves visiting each element starting from the first node and comparing each element with the searching element. Once the searching item is found the process stops otherwise keep visiting the next node. Basically, searching involves traversing the linked list partially or fully (in the worst case). The full list is visited if the searching item is absent in the list. The complexity of searching in linked list mostly like searching in an array.

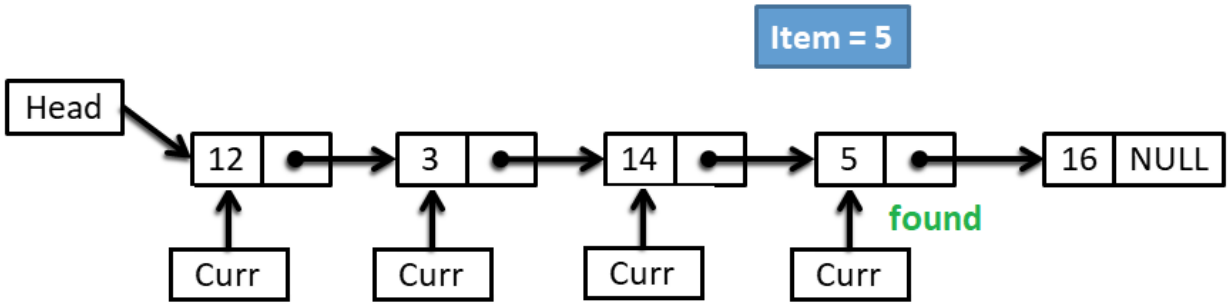


Figure 6.9: Searching in a linked list

Figure 6.5 represents an example of searching in a linked list. Like traversing a pointer Curr is assigned the address of first node (containing 12) and the data is compare with the searching element (5). If the data of node is a mismatch Curr is updated to point to the next node and the process goes on until 5 is found in 4th node.

```
Algorithm (Searching in linked list)
Input: Head (the address of first node)
Curr = Head
Step 1: if Curr == NULL print not found and exit
If Curr->data = item print found and exit
Step 2: move Curr to next node and go to step 1
```

6.6 Insertion in Linked List

Insertion in a linked list doesn't require any physical movements of data unlike insertion in an array. In a linked list, once the data is inserted the physical memory location is fixed. To insert a new data element, a new node is created in the memory and the data is placed inside the node. This doesn't complete the insertion process because the newly created node must be placed (logically) in the linked list. This process is done by updating the links (or we call it pointer, next etc.).

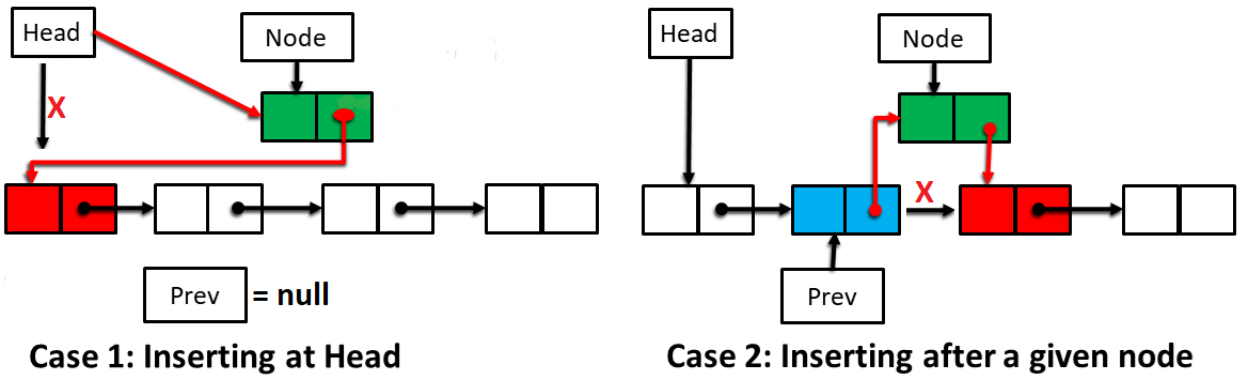


Figure 6.10: Insertion in a linked list

Figure 6.6 shows an illustrative example of insertion operation in a linked list. As already explained earlier in the section, a new node (colored green in the figure) is created which

contains the inserting data. Now, there are two cases such as **case 1)** the new node is inserted as the first node and **case 2)** the new node is inserted after a given node (blue colored in the figure).

In case 1, at first we need to make a link (colored red) between the new node and the first node and then update Head by the address of newly created node. The link is established by replacing the value at Node→next by the address of the first node (colored red).

In case 2, we need to make two new links (colored red). At first make a link between the new node and the next node (colored red) then another link between the previous node and the new node. The first link is established by replacing the value at Node→next by the address of next node (stored in Prev→next). And the second link is established by replacing the value at Prev→next by the address of the new node.

```
Algorithm (insertion in linked list)
Input:
Head (the address of first node),
Node (inserting node),
Prev (address of previous node)

Case 1:
if Prev != NULL then go to Case 2
Make a link from Node to first node
Make Node as the Head
Exit
Case 2:
Make a link from Node to the node next to Prev
Make another link from Prev to Node
```

6.7 Deletion in Linked List

Deletion in the linked list involves updating links between nodes. Like insertion, there are two cases in deletion. Case 1 occurs when we want to delete the first node and Case 2 occurs when we want to delete any other node which has a previous node.

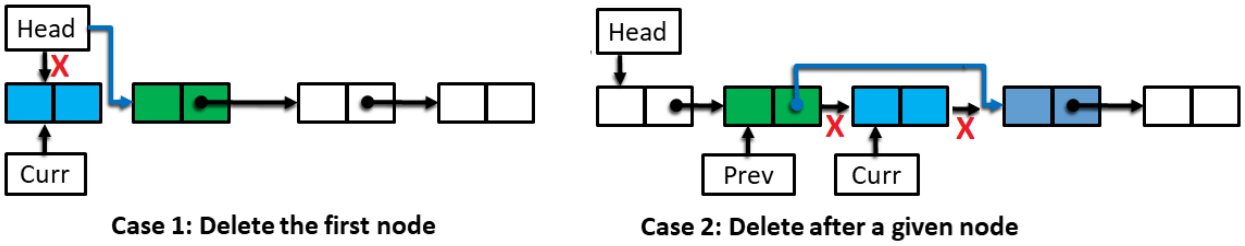


Figure 6.11: Deletion in linked list

Figure 6.7 illustrates the two cases of deletion. In the first case, the head pointer is updated by the address of second node which automatically discards the first node. In case 2, a node (pointed by Curr) is deleted by making another link between the previous node and the next node. This is done by simply replacing the value at Prev→next by the value at Curr→next (the address of next node).

```
Algorithm (deletion in linked list)
Input:
Head (the address of first node),
Prev (address of previous node)
Case 1:
if Prev = NULL then go to Case 2
Curr = Prev->next
Make a link from Prev to the node next to Curr
Exit
Case 2:
Make the node next to Head as new Head
```

6.8 Doubly Linked List

The linked list which has been so far discussed is called Singly Linked List. In Singly Linked List, a node contains data and the address of the next node which allows traversing in one direction starting from the first node to the last. Other way traversal is not possible. In contrast, a Doubly Linked List allows traversing to the both directions. Unlike a singly linked list, a doubly linked list contains data plus two pointers namely prev and next, which are addresses of the previous node and the next node respectively.

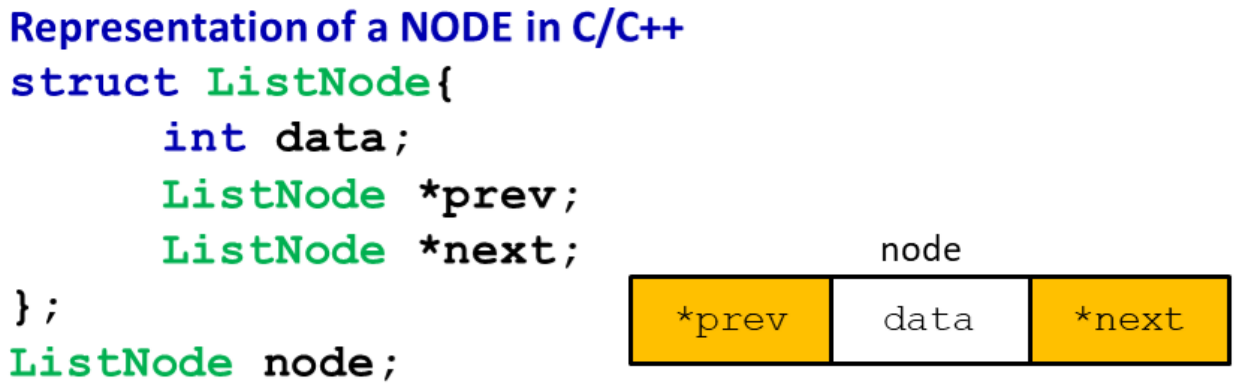


Figure 6.12: Representation of Doubly Linked List node

Figure 6.8 shows the representation of a node in doubly linked list. It is clear from the representation that from a given node traversing can be done in both directions.

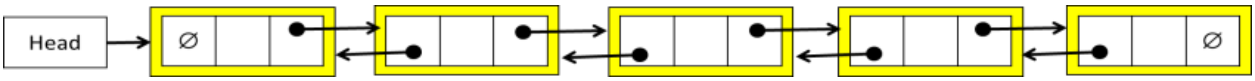


Figure 6.13: A doubly linked list

Figure 6.9 shows a representation of a doubly linked list. Note that prev of the first node is NULL and the next of the last node is NULL

6.9 References

- Kruse, R. L., Ryba, A. J., & Ryba, A. (1999). Data structures and program design in C++ (p. 280). New York: Prentice Hall.
- Shaffer, C. A. (2012). Data structures and algorithm analysis. Update, 3, 0-3.
- Dale, N. B. (2003). C++ plus data structures. Jones & Bartlett Learning.

Chapter 7

Binary Search Tree

Trees are particularly useful in computer science, where they are employed in a wide range of algorithms. For instance, trees are used to construct efficient algorithms for locating items in a list. They can be used in algorithms, such as Huffman coding, that construct efficient codes saving costs in data transmission and storage. Trees can be used to study games such as checkers and chess and can help determine winning strategies for playing these games. Trees can be used to model procedures carried out using a sequence of decisions. Constructing these models can help determine the computational complexity of algorithms based on a sequence of decisions, such as sorting algorithms.

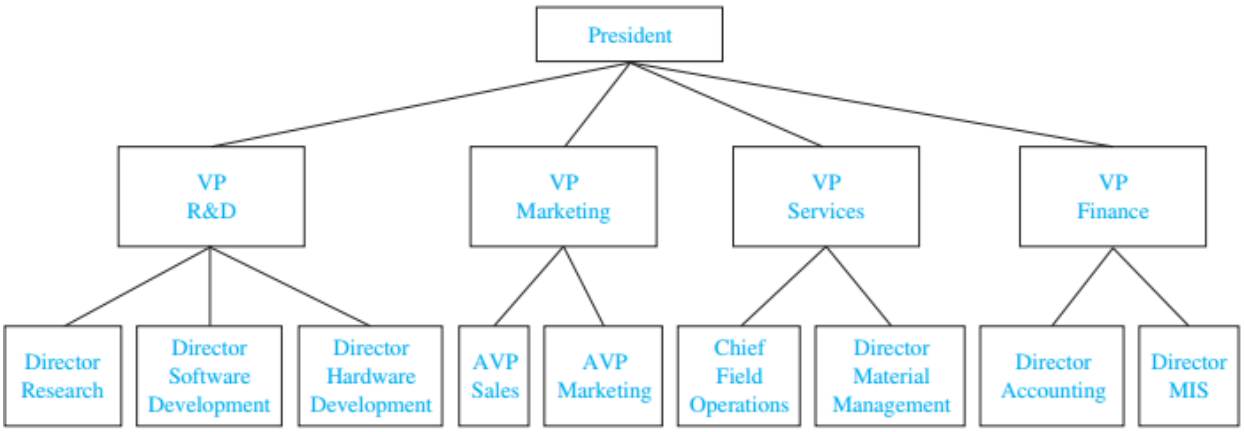


Figure 7.14: An organizational tree for a computer company.

The structure of a large organization can be modeled using a rooted tree. Each vertex in this tree represents a position in the organization. An edge from one vertex to another indicates that the person represented by the initial vertex is the (direct) boss of the person represented by the terminal vertex. The graph shown in Figure 7.1 displays such a tree. In the organization represented by this tree, the Director of Hardware Development works directly for the Vice President of R&D. The root of this tree is the vertex representing the President of the organization.

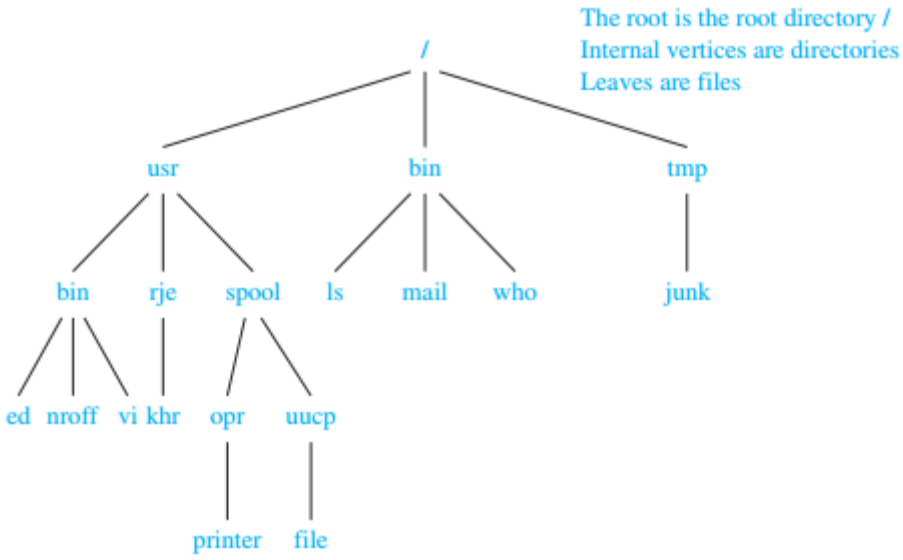


Figure 7.15: A computer file system.

Files in computer memory can be organized into directories. A directory can contain both files and subdirectories. The root directory contains the entire file system. Thus, a file system may be represented by a rooted tree, where the root represents the root directory, internal vertices represent subdirectories, and leaves represent ordinary files or empty directories. One such file system is shown in Figure 7.2. In this system, the file khr is in the directory rje. (Note that links to files where the same file may have more than one pathname can lead to circuits in computer file systems.)

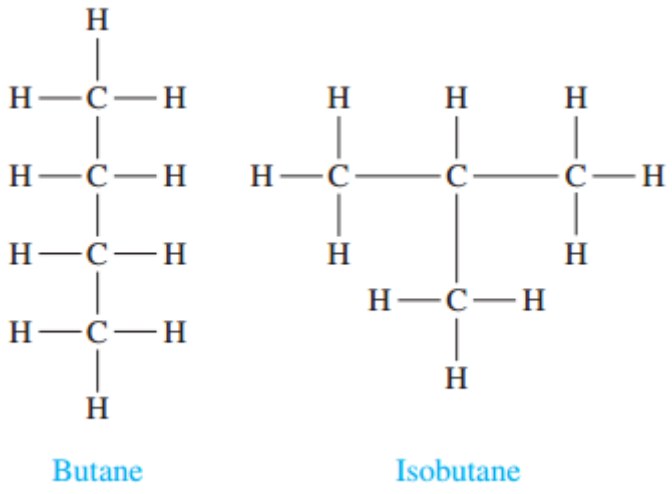


Figure 7.16: The two isomers of butane.

Graphs can be used to represent molecules, where atoms are represented by vertices and bonds between them by edges. One of the examples is shown in Figure 7.3 which are called

saturated hydrocarbons. (Isomers represent compounds with the same chemical formula but different chemical properties.)

7.1 Tree Terminologies

As a data structure, a tree consists of one or more nodes, where each node has a value and a list of references to other (its children) nodes. A tree must have a node designated as root. If a tree has only one node, that node is the root node. Root is never referenced by any other node. Having more than one node indicates that the root have some (at least one) references to its children nodes and the children nodes (might) have references to their children nodes and so on.

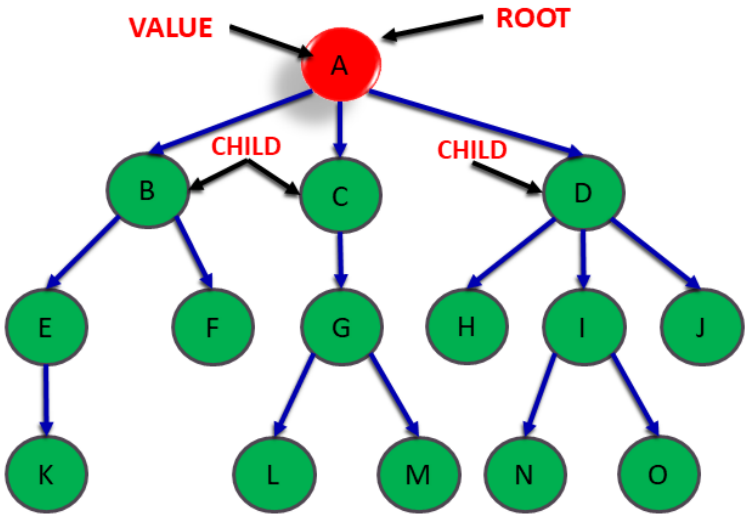


Figure 7.17: A tree

In figure 7.4, an example of tree is shown. Generally it is considered that in a tree, there is only one path going from one node to another node. There cannot be any cycle or loop. The link from a node to other node is called an edge.

The terminology for trees has botanical and genealogical origins. Suppose that T is a tree. If v is a **vertex** or **node** in T other than the **root**, the **parent** of v is the unique vertex u such that there is a directed edge from u to v. When u is the parent of v, v is called a child of u. Vertices with the same parent are called **siblings** or **children** of the parent. The ancestors of a vertex other than the root are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root (that is, its parent, its parent's parent, and so on, until the root is reached). The descendants of a vertex v are those vertices that have v as an ancestor. A vertex of a tree is called a **leaf** if it has no **children**. Vertices that have children are called **internal** vertices. The root is an internal vertex unless it is the only vertex in the graph, in which case it is a leaf.

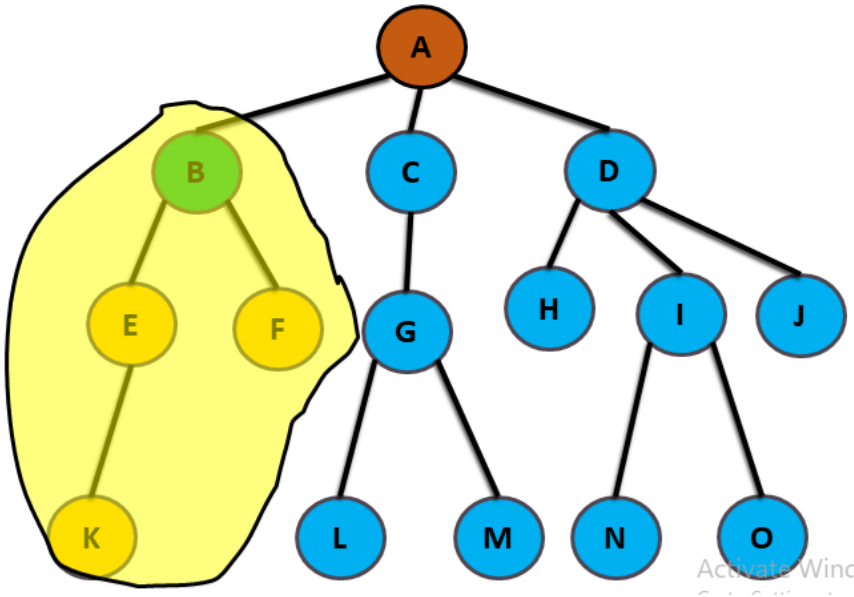


Figure 7.18: An example of Sub tree

The tree rooted by a child is called a sub tree. Thus a tree may have many sub trees. In Figure 7.5, a sub tree is shown (tree encircled in yellow colored area). Similarly, there will be other sub trees rooted at vertex C and D. And a sub tree may have many other sub trees inside it.

7.2 m-ary tree and Binary Tree

m-ary tree: A Tree is an m-ary Tree when each of its node has no more than m children.

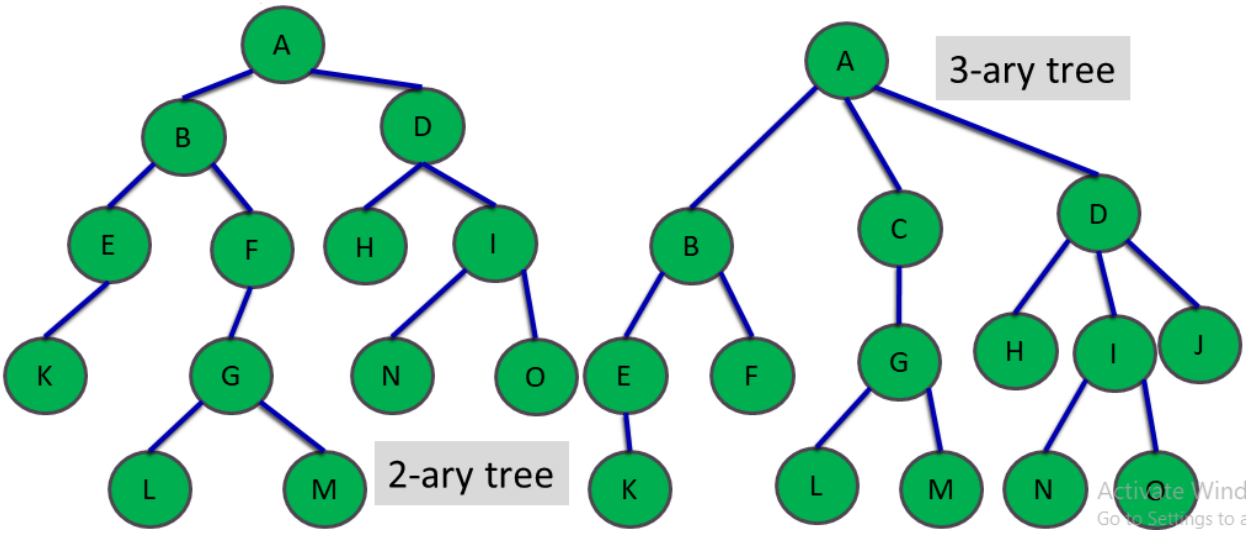


Figure 7.19: m-ary tree

A binary tree is a basically a 2-ary tree where a node can have no more than 2 children.

Complete Binary Tree: A complete binary tree is a binary tree, which is completely filled with nodes from top to bottom and left to right. In complete binary tree some of the nodes only for the bottom level might be absent (here nodes after N as shown in Figure 7.7).

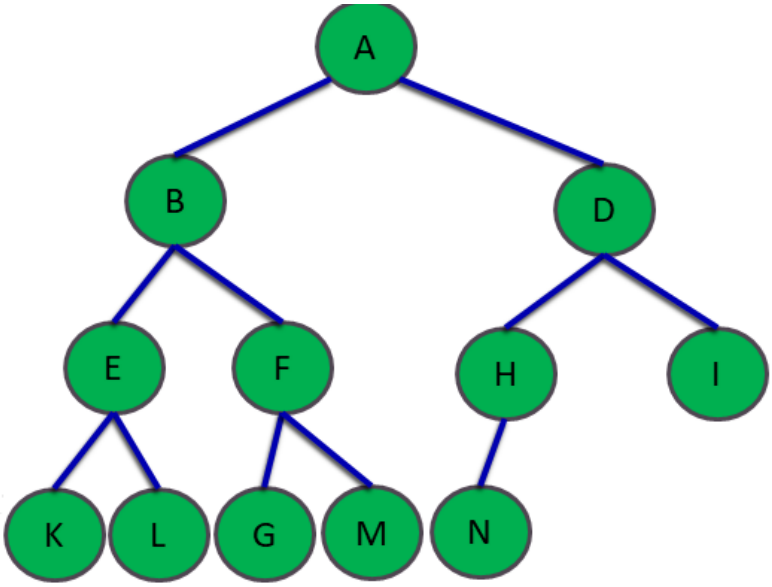


Figure 7.20: A complete binary tree

Full Binary Tree: In addition to a complete binary tree, in a full binary tree the last/bottom level must also be filled up. So it can be easily noticed that a full binary tree is always a complete binary tree but the vice-versa may not be true. Figure 7.8 shows a full binary tree where all the levels full with maximum nodes.

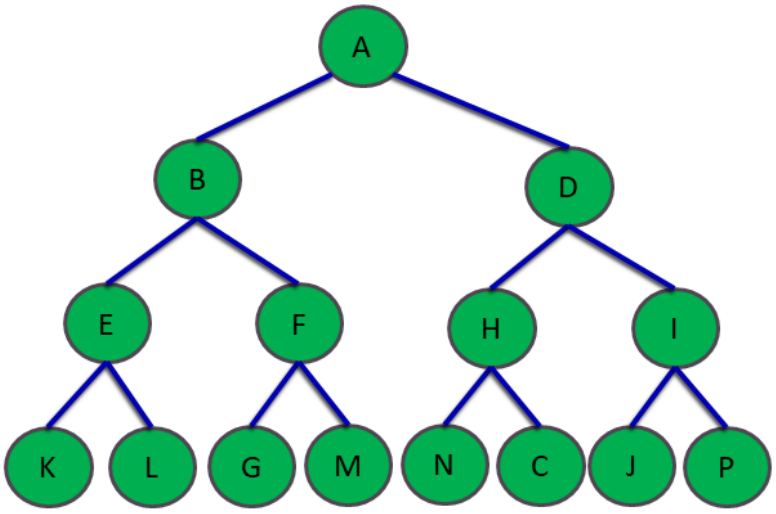


Figure 7.21: A full binary tree

In a full binary tree, number of nodes at each level l is 2^l

If there are L labels then total number of nodes will be $2^0 + 2^1 + 2^2 + \cdots + 2^{L-1} = 2^L - 1$

If total number of nodes is n then the height of the tree is $\lceil log_2 n \rceil$

7.3 Tree Traversal

Procedures for systematically visiting every vertex of a tree are called **traversal** algorithms. We will describe three of the most commonly used such algorithms, **preorder** traversal, **inorder** traversal, and **postorder** traversal. Each of these algorithms can be defined recursively. We will explain the traversal algorithms for binary tree.

Inorder traversal: In this traversal, the left sub tree is visited first then the root is visited then the right sub tree. Recursively the left sub tree and the right sub tree will be visited in the same manner.

Preorder traversal: In this traversal the root is visited first then the left sub tree is visited then the right sub tree. Recursively the left sub tree and the right sub tree will be visited in the same manner.

Postorder traversal: In this traversal, the left sub tree is visited first then the right sub tree is visited then the root. Recursively the left sub tree and the right sub tree will be visited in the same manner.

Figures[3] 7.9-7.11 shows the inorder, preorder and postorder traversal.

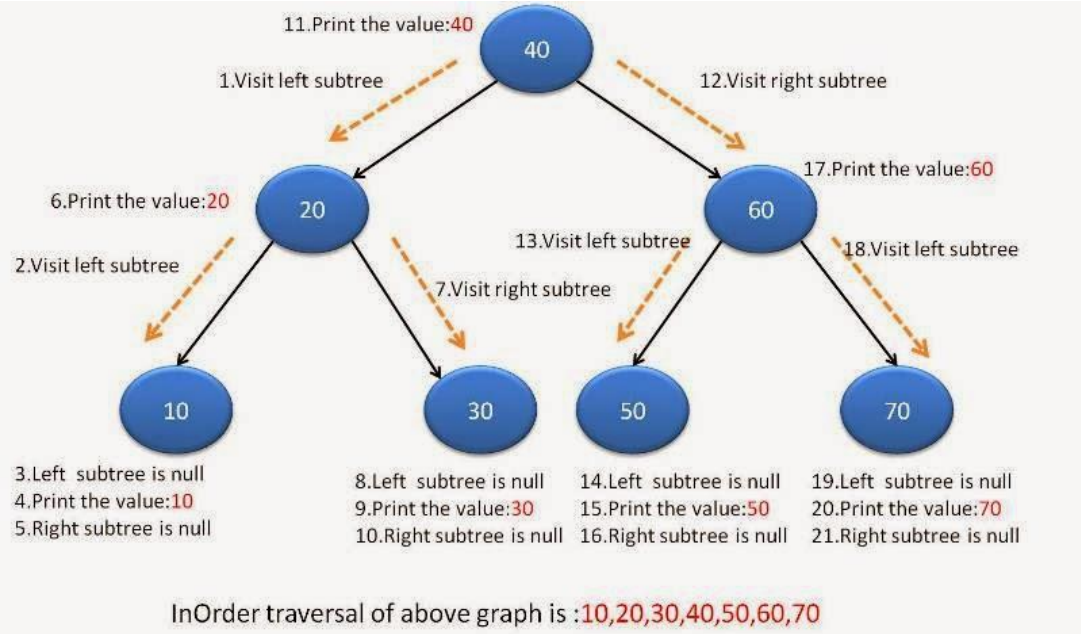


Figure 7.22: inorder traversal

[3] https://java2blog.com/binary-tree-inorder-traversal-in-java/

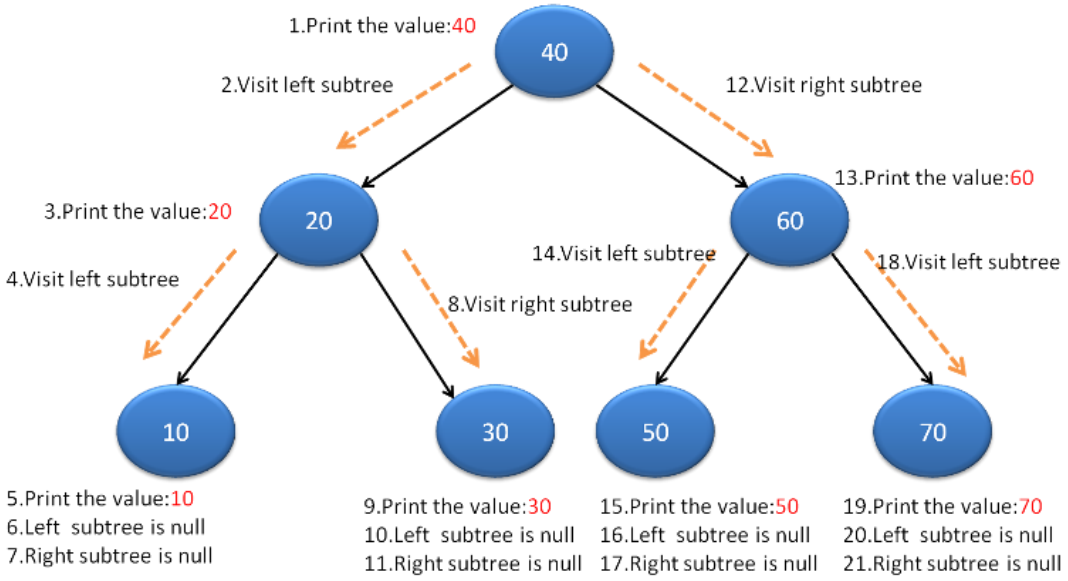


Figure 7.23: preorder traversal

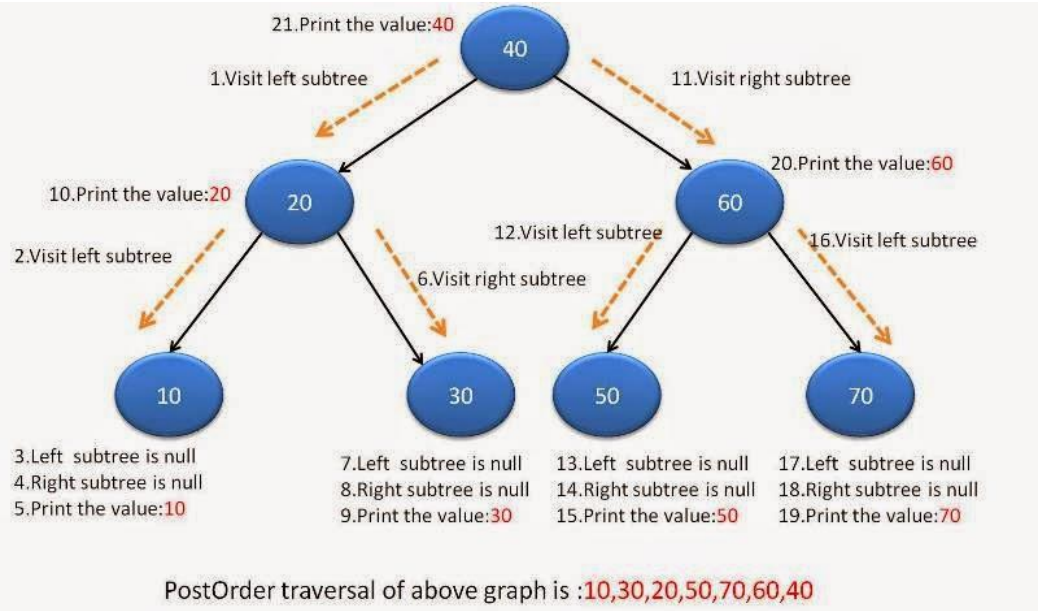


Figure 7.24: postorder traversal

7.4 Binary Search Tree

Consider the problem of searching a linked list for some target key. There is no way to move through the list other than one node at a time, and hence searching through the list must always reduce to a sequential search. As you know, sequential search is usually very slow in comparison with binary search. Hence, assuming we

can keep the keys in order, searching becomes much faster if we use a contiguous list and binary search. Suppose we also frequently need to make changes in the list, inserting new entries or deleting old entries. Then it is much slower to use a contiguous list than a linked list, because insertion or removal in a contiguous list requires moving many of the entries every time, whereas a linked list requires only adjusting a few pointers.

Binary trees provide an excellent solution to this problem. By making the entries of an ordered list into the nodes of a binary tree, we shall find that we can **search** for a target key in O(log n) steps, just as with binary search, and we shall obtain algorithms for **inserting** and deleting entries also in time O(log n).

A **binary search tree (BST)** is a binary tree that is either empty or in which every node has a key (within its data entry) and satisfies the following conditions:

1. The key of the root (if it exists) is greater than the key in any node in the left subtree of the root.
2. The key of the root (if it exists) is less than the key in any node in the right subtree of the root.
3. The left and right subtrees of the root are again binary search trees.

No two entries in a binary search tree may have equal keys.

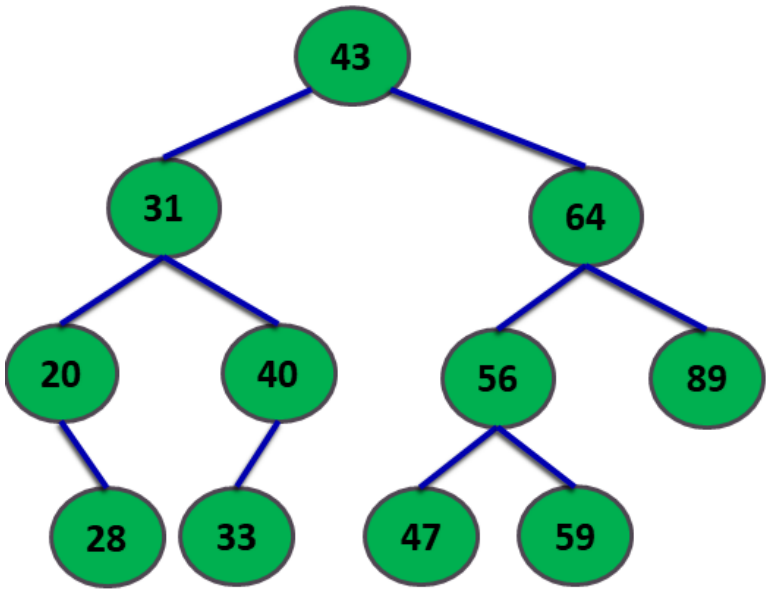


Figure 7.25: An example of a Binary Search Tree

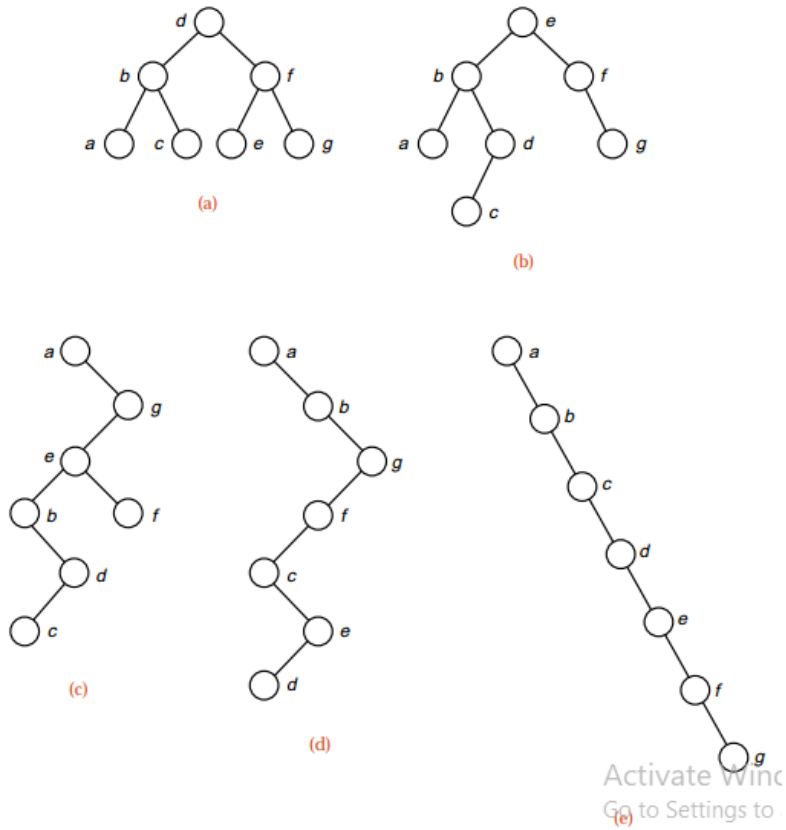


Figure 7.26: Several binary search trees with the same keys

7.5 Insertion into a BST

The next important operation for us to consider is the insertion of a new node into a binary search tree in such a way that the keys remain properly ordered; that is, so that the resulting tree satisfies the definition of a binary search tree.

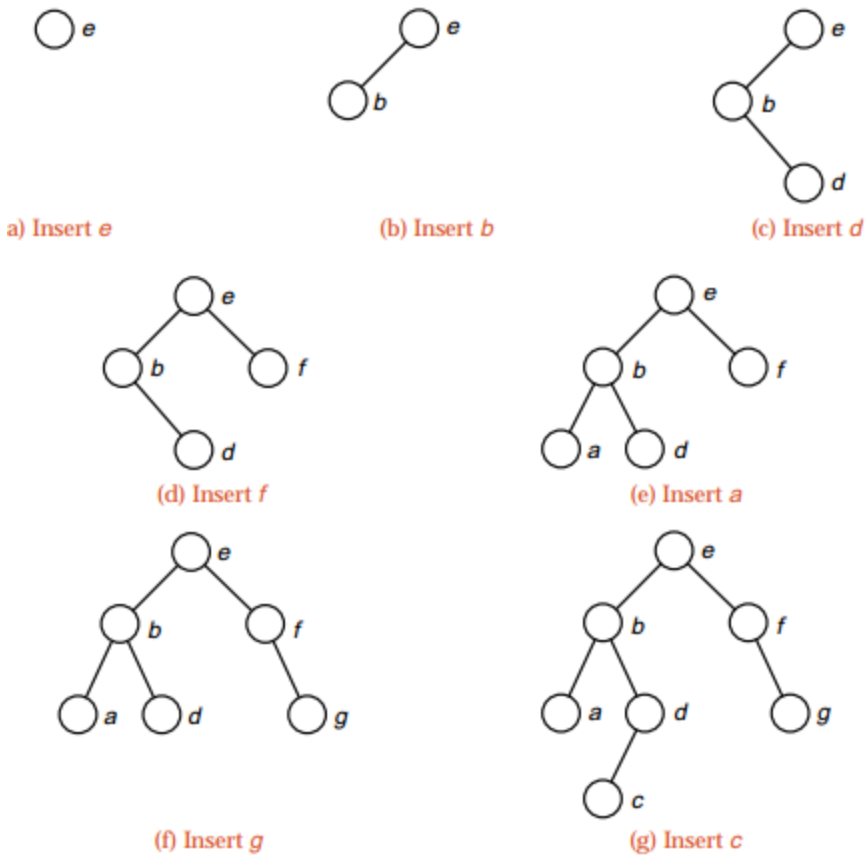


Figure 7.27: Insertions into a binary search tree

When the first entry, e, is inserted, it becomes the root, as shown in part (a). Since b comes before e, its insertion goes into the left subtree of e, as shown in part (b). Next we insert d, first comparing it to e and going left, then comparing it to b and going right. The next insertion, f, goes to the right of the root, as shown in part (d) of Figure 10.9. Since a is the earliest key inserted so far, it moves left from e and then from b. The key g, similarly, comes last in alphabetical order, so its insertion moves as far right as possible, as shown in part (f). The insertion of c, finally, compares first with e, goes left, then right from b and left from d. Hence we obtain the binary search tree shown in the last part of Figure 7.14.

It is quite possible that a different order of insertion can produce the same binary search tree. The final tree in Figure 7.14, for example, can be obtained by inserting the keys in either of the orders

e, f, g, b, a, d, c or e, b, d, c, a, f, g,

as well as several other orders.

7.6 Searching into a BST

To search for the target, we first compare it with the entry at the root of the tree. If their keys match, then we are finished. Otherwise, we go to the left subtree or right subtree as appropriate and repeat the search in that subtree.

Let us, for example, search for the key 33 in the binary search tree of Figure 7.12. We first compare 33 with the entry in the root, 43. Since 33 is smaller than 43, we move to the left and next compare 33 with 31. Since 33 is larger than 31, we move right and compare 33 with 40. Now 33 is smaller than 40, so we move to the left and find the desired target.

The tree shown as part (a) of Figure 7.13 is the best possible for searching. It is as "bushy" as possible: It has the smallest possible height for a given number of vertices. The number of vertices between the root and the target, inclusive, is the number of comparisons that must be done to find the target. The bushier the tree, therefore, the smaller the number of comparisons that will usually need to be done.

7.7 Deletion from a BST

If the node to be removed is a leaf, then the process is easy: We need only replace the link to the removed node by NULL. The process remains easy if the removed node has only one nonempty subtree: We adjust the link from the parent of the removed node to point to the root of its nonempty subtree.

When the node to be removed has both left and right subtrees nonempty, however, the problem is more complicated. To which of the subtrees should the parent of the removed node now point? What is to be done with the other subtree? This problem is illustrated in Figure 7.15. First, we find the immediate predecessor of the node under inorder traversal by moving to its left child and then as far right as possible. (The immediate successor would work just as well.) The immediate predecessor has no right child (since we went as far right as possible), so it can be removed from its current position without difficulty. It can then be placed into the tree in the position formerly occupied by the node that was supposed to be removed, and the properties of a binary search tree will still be satisfied, since there were no keys in the original tree whose ordering comes between the removed key and its immediate predecessor.

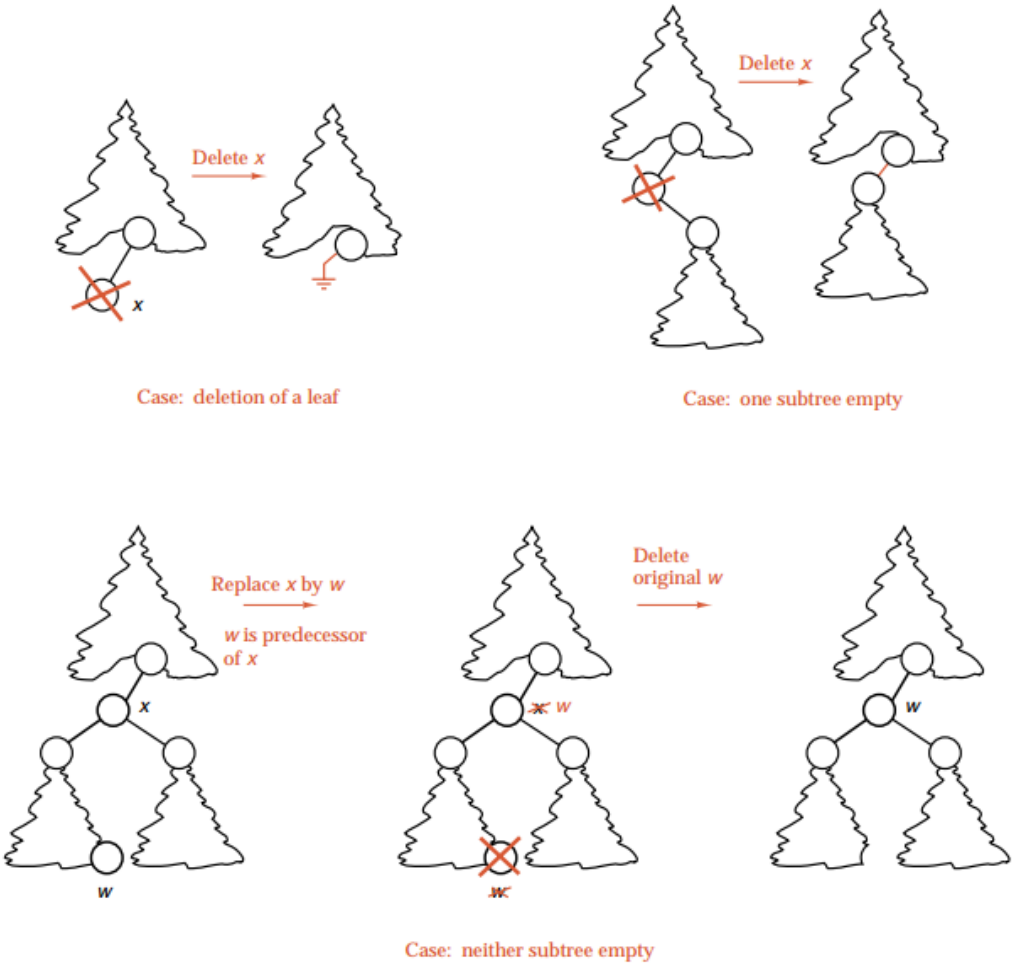


Figure 7.28: Deletion of a node from a binary search tree

7.8 References

- Kruse, R. L., Ryba, A. J., & Ryba, A. (1999). Data structures and program design in C++ (p. 280). New York: Prentice Hall.
- Shaffer, C. A. (2012). Data structures and algorithm analysis. Update, 3, 0-3.
- Dale, N. B. (2003). C++ plus data structures. Jones & Bartlett Learning.

Chapter 8

Heap

Heaps are, after the search trees, the second most studied type of data structure. As abstract structure they are also called priority queues, and they keep track of a set of objects, each object having a key value (the priority), and support the operations to `insert` an object, find the object of minimum key (`find min`), and delete the object of minimum key (`delete min`). So unlike the search trees, there are neither arbitrary find operations nor arbitrary delete operations possible. Of course, we can replace everywhere the minimum by maximum; where this distinction is important, one type is called the min-heap and the other the max-heap. If we need both types of operations, the structure is called a double-ended heap, which is a bit more complicated.

The heap structure was originally invented by Williams (1964) for the very special application of sorting, although he did already present it as a separate data structure with possibly further applications. But it was recognized only much later that heaps have many other, and indeed more important, applications. Still, the connection to sorting is important because the lower bound of $\Omega(n \log n)$ on comparison-based sorting of n objects implies a lower bound on the complexity of the heap operations. We can sort by first inserting all objects in the heap and then performing `find min` and `delete min` operations to recover the objects, sorted in increasing order. So we can sort by performing n operations each of `insert`, `find min`, and `delete min`; thus, at least one of these operations must have (in a comparison-based model) a complexity $\Omega(\log n)$. This connection works in both directions; there is an equivalence between the speed of sorting and heap operations in many models – even in which the comparison-based lower bound for sorting does not hold.

The various methods to realize the heap structure differ mainly by the additional operations they support. The most important of these are the merging of several heaps (taking the union of the underlying sets of objects), which is sometimes also called melding, and the change of the key of an object (usually decreasing the key), which requires a finger to the object in the structure.

The most important applications of heaps are all kinds of event queues, as they occur in many diverse applications: sweeps in computational geometry, discrete event systems (Evans 1986), schedulers, and many classical algorithms such as Dijkstra's shortest path algorithm.

In this chapter, we are going to explore an important data structures named heap. We will start with the basic concept of heap and types of heap. Later we will dive deeper with lots of examples and their explanations. Finally, we will be ending this chapter by learning the applications of heap.

8.1 Heap Types

A heap is an almost complete binary tree with all levels is full, except possibly the last one, which is filled from left to right.

Definition 8.1: We define "height" of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf. The height of a heap is the height of its root. A heap of `n` nodes has a height of $\lfloor \log n \rfloor$. (Hint: if a heap has height h, then the minimum and maximum possible number of elements is $2h \leq n \leq 2h + 1 - 1$.

Definition 8.2: The "max-heap property" (of a heap A) is the property we were talking about, where for every node `i` other than the root, `A[parent[i]]≥A[i]`. The "min-heap property" is defined analogously.

Max heap can be viewed as a binary tree, where each node has two (or fewer) children, and the key of each node (i.e. the number inside the node) is greater than the keys of its child nodes. (From now on I'm going to say node `i` is larger than node `j` when I really mean that the key of `i` is larger than the key of `j`.) There is also min heaps, where each node is smaller than its child nodes, but here we will talk about max heaps, with the understanding that the algorithms for min heaps are analogous.

For example, the root of a max heap is the largest element in the heap. However, note that it's possible for some nodes on level 3 to be smaller than nodes on level 4 (if they're in different branches of the tree). In figure 8.1, it doesn't matter that 4 in level 1 is smaller than 5 in level 2. Figure 8.2 shows the example of max heap and figure 8.3 shows the example of max heap and min heap.

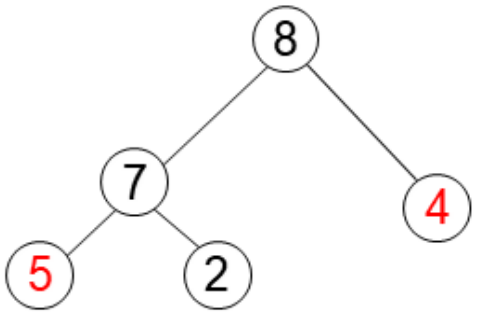


Figure 8.1: Heap (top to bottom and left to right)

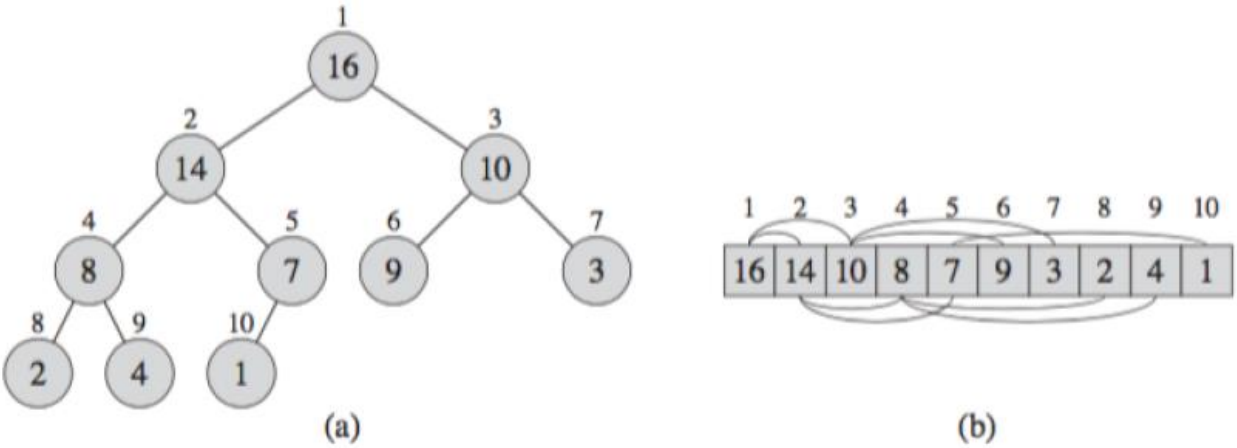


Figure 8.2 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

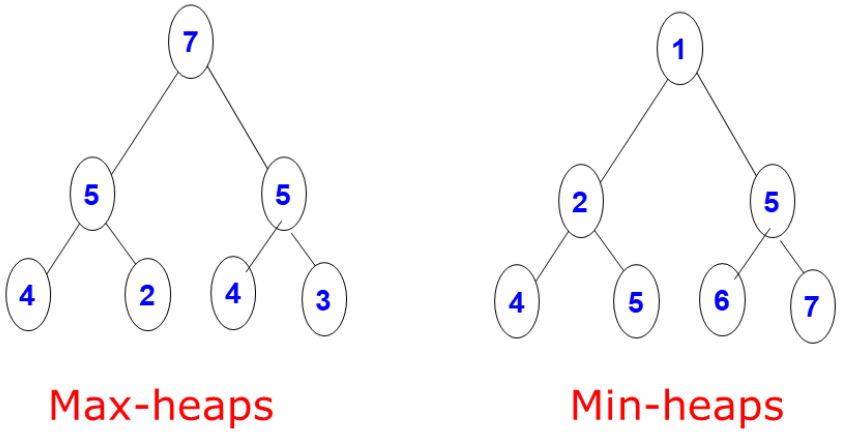


Figure 8.3: The example of max heap and min heap

8.2 Heap as an Array

Heap can be implemented as an array. This array is essentially populated by "reading of" the numbers in the tree, from left to right and from top to bottom. The root is stored at index 0, and if a node is at index i, then

- Its left child has index 2i+1
- Its left child has index 2i+2
- Its parent has index $\lfloor {}^{i-1}/_{2} \rfloor$

Furthermore, for the heap array A, we also store two properties: length n, which is the number of elements in the array, and heap size, which is the number of array elements that are actually part of the heap. Even though the array A is filled with numbers, only the elements in A[0…heapsize] are actually part of the heap. Therefore, Heapsize[A] ≤ length[A]. The elements in A[$\lfloor n/2 \rfloor$] to A[n-1] are leaves and Parents are A[0] to A[$\lfloor n/2 \rfloor - 1$]. The root have the maximum element of the heap.

8.3 Operations on Heap

The common operation involved using heaps are:

- Maintaining the heap property :
  It is a process to rearrange the elements of the heap in order to maintain the heap property. It is done when a certain node causes an imbalance in the heap due to some operation on that node. For example, Max-Heapify .

- Create a max-heap from an unordered array:
  The Build-Max-Heap function that follows, converts an array *A* which stores a complete binary tree with *n* nodes to a max-heap by repeatedly using Max-Heapify in a bottom up manner. It is based on the observation that the array elements indexed by A[$\lfloor n/2 \rfloor$] to A[n-1] are all leaves for the tree (assuming that indices start at 0) , thus each is a one-element heap. Build-Max-Heap runs Max-Heapify on each of the remaining tree nodes.

- Sort an Heap Array
  Heapsort can be thought of as an improved selection sort like selection sort, heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. Unlike selection sort, heapsort does not waste time with a linear-time scan of the unsorted region; rather, heap sort maintains the unsorted region in a heap data structure to more quickly find the largest element in each step [1].

8.3.1 Maintaining the heap property

Heaps use this one weird trick to maintain the max-heap property. Unlike most of the algorithms we've seen so far, Max-Heapify assumes a very special type of input. Given a heap A and a node i inside that heap, Max-Heapify attempts to "heapify" the subtree rooted at i, i.e., it changes the order of nodes in A so that the subtree at i satisfies the max-heap property. However, it assumes that the left subtree of i and the right subtree of i are both max-heaps in and of themselves, so the only possible violation is that i might be smaller than its immediate children.

In order to fix this violation, the algorithm finds which node is largest: i, i's left child, or i's right child. If i is largest, the max-heap property is already satisfied. Otherwise, it swaps i with the largest node, which causes the top of the heap to be fine. But this might mess

up the subtree that used to be under the largest node, because `i` might be smaller than some of the nodes in that subtree. So now we call `Max-Heapify` on that subtree (because the only possible violation of the max-heap property occurs at the top of the subtree). `Max-Heapify` runs in time `O(h)`, where `h` is the height of the node.

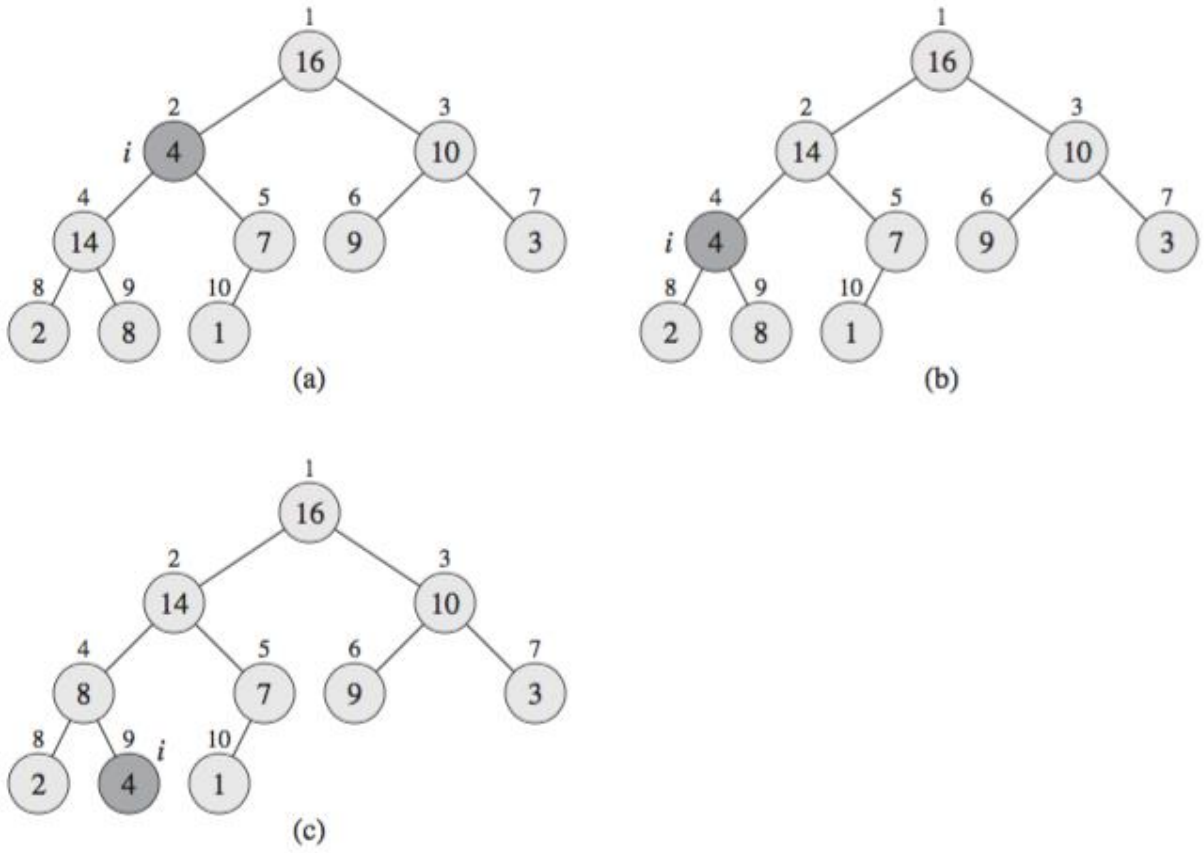


Figure 8.4 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

Recall that we are implementing heaps as arrays, so the actual code of `Max-Heapify` takes as input an array `A` and an index `i` into that array.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

8.3.2 Building a heap

The `Build-Max-Heap` procedure runs `Max-Heapify` on all the nodes in the heap, starting at the nodes right above the leaves and moving towards the root. We start at the bottom because in order to run `Max-Heapify` on a node, we need the subtrees of that node to already be heaps.

Recall that the leaves of the heap are the nodes indexed by `A[⌊n/2⌋]` ..... `A[n-1]` so when we use our array implementation we just start at the index `A[⌊n/2⌋-1]` and move towards `0`. For example in figure 8.5, we will start from index `4` and move towards `0`.

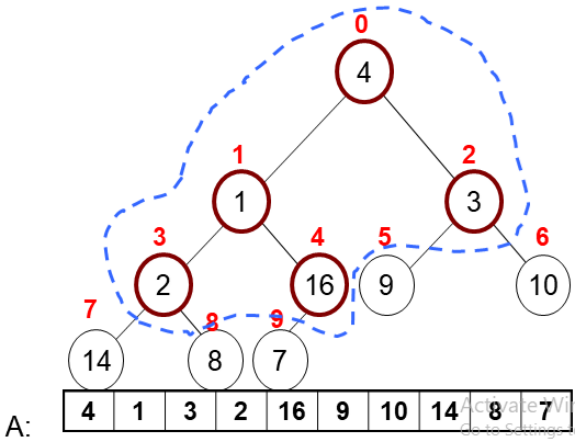


Figure 8.5: The example of Build-Max-Heap range

```
BUILD-MAX-HEAP(A)
  1. n = length of A[]
  2.  for i = ⌊n/2⌋-1 down to 0
  3.        do MAX-HEAPIFY(A, i)
```

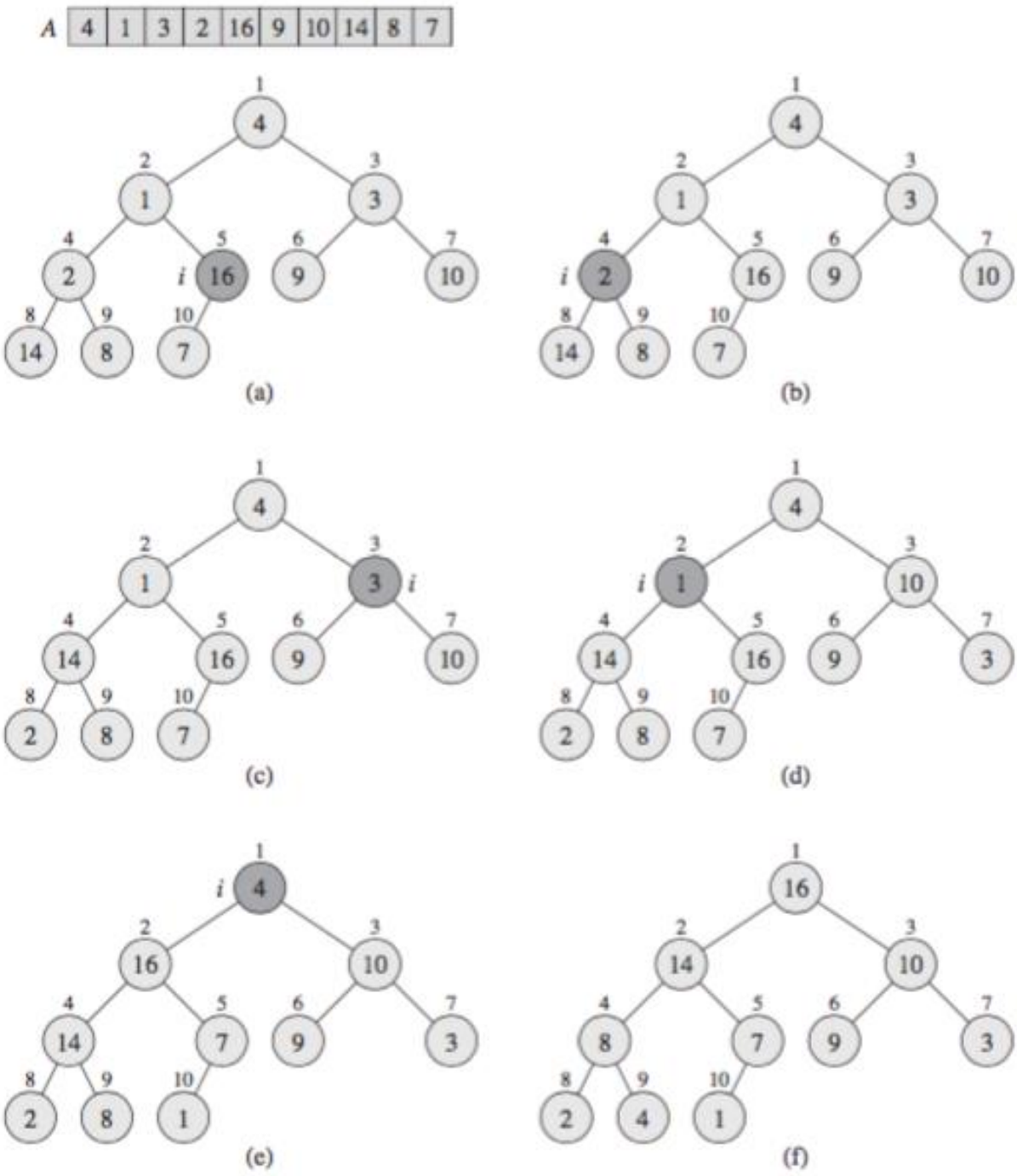


Figure 8.6 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

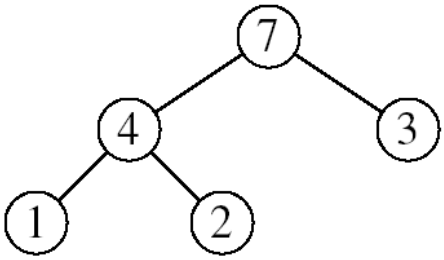
8.3.3 Heap Sort

`Heapsort` is a way of sorting arrays. First we use `Build-Max-Heap` to turn the array into a max-heap. Now we can extract the maximum (i.e. the root) from the heap, swapping it with the last element in the array and then shrinking the size of the heap so we never operate on the max element again. At this point, the heap property is violated because the root may be smaller than other elements, so we call `Max-Heapify` on the root, which restores the max heap property. Now we can keep repeating this until the whole array is sorted.

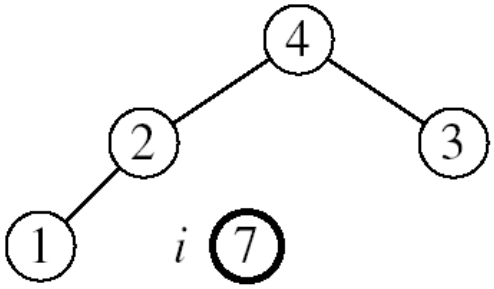
Note that each call to `Max-Heapify` takes `O(log n)` because the height of the heap is `O(log n)`, so heapsort takes `O(n log n)` overall.

```
HeapSort(A)
   1. BUILD-MAX-HEAP(A)
   2. for i ← n-1 down to 1
   3.     do exchange A[0] ↔ A[i]
   4.         n=n-1
   5.         MAX-HEAPIFY(A,0)
```

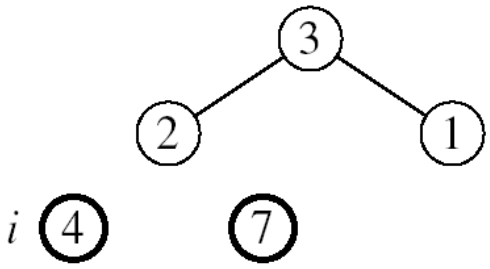
For example, we have a array `A` with values `A=[7,4,3,1,2],` figure 8.7 shows the step by step simulation of heap sort.



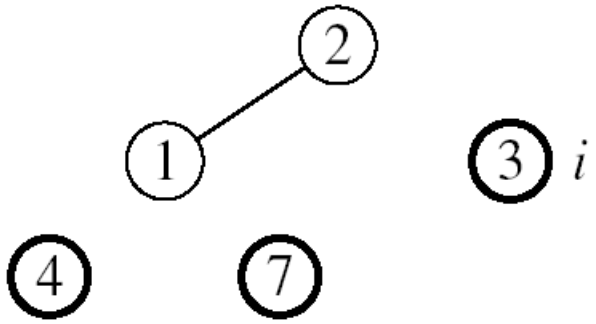
(6) n=4 , Max-Heapify(A,0)



(1) n=3 , Max-Heapify(A,0)



(5) n=2 , Max-Heapify(A,0)



(2) n=1 , Max-Heapify(A,0)

8.4 Complexity of Heapsort

There are two steps to sort an array, or list, containing `N` values, first insert each value into a heap (initially empty) and then remove each value form the heap in ascending order (this is done by `N` successive calls to get smallest).

Hence the the complexity of the Heapsort algorithm comprises of two operations like (`N` insert operations) + (`N` delete operations). Each insert and delete operation is `O(logN)` at the very worst - the heap does not always have all N values in it. So, the complexity is certainly no greater than `O(NlogN)`.

8.5 Uses of Heap

There are two main uses of heaps.

The first is as a way of implementing a special kind of queue, called a priority queue. Recall that in an ordinary queue, elements are added at one end of the queue and removed from the other end, so that the elements are removed in the same order they are added (FIFO). In a priority queue, each element has a priority; when an element is removed it must be the element on the queue with the highest priority.

A very efficient way to implement a priority queue is with a heap ordered by priority - each node is higher priority than everything below it. The highest priority element, then, is at the top of the heap.

The second application is sorting. Heapsort is especially useful for sorting arrays. Because heaps - unlike almost all other types of trees - are usually implemented in arrays, not as linked data structures!

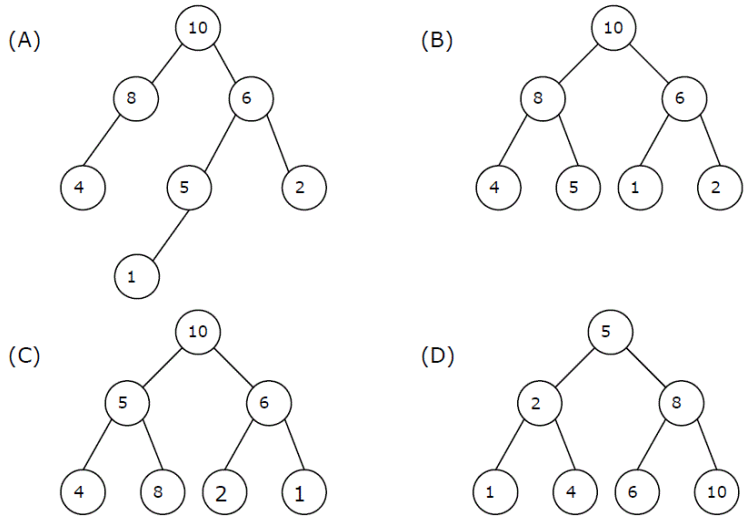
8.6 Some Important Properties of a Heap

- Heaps are based upon trees. These trees maintain the heap property.
    - The Heap invariant. The value of Every Child is greater than the value of the parent. We are describing Min-heaps here (Use less than for Max-heaps).
- The trees must be mostly balanced for the costs listed below to hold.
- Access to elements of a heap usually has the following costs.
    - The cost to find the smallest (largest) element takes constant time.
    - The cost to delete the smallest (largest) element takes time proportional to the log of the number of elements in the set.
    - The cost to add a new element takes time proportional to the log of the number of elements in the set.
- Heaps can be implemented using arrays (using the tree embedding described above) or by using balanced binary trees

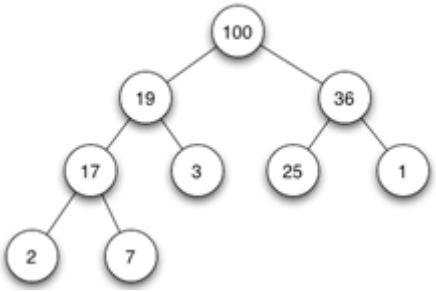
- Trees with the leftist property have the following invariant.
    - The leftist invariant. The rank of every left-child is equal to or greater than the rank of the corresponding right-child. The rank of a tree is the length of the right-most path.
- Heaps form the basis for an efficient sort called heap sort that has cost proportional to `n*log(n)` where n is the number of elements to be sorted.
- Heaps are the data structure most often used to implement priority queues.

8.7 Exercises

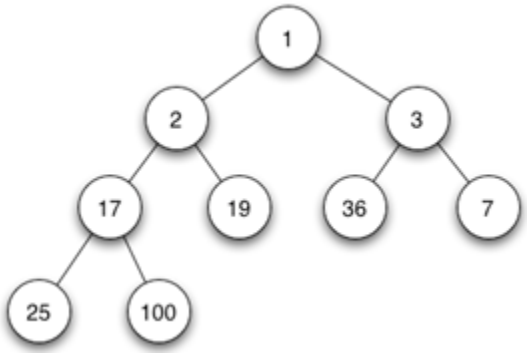
1. What is the time complexity of Build Heap operation. Build Heap is used to build a max(or min) binary heap from a given array. Build Heap is used in Heap Sort as a first step for sorting.
2. Suppose we are sorting an array of eight integers using heapsort, and we have just finished some heapify (either maxheapify or minheapify) operations. The array now looks like this: 16 14 15 10 12 27 28 How many heapify operations have been performed on root of heap?
3. Consider a binary max-heap implemented using an array. Which one of the following array represents a binary max-heap?
   (A) 25,12,16,13,10,8,14
   (B) 25,12,16,13,10,8,14
   (C) 25,14,16,13,10,8,12
   (D) 25,14,12,13,10,8,16
4. A max-heap is a heap where the value of each parent is greater than or equal to the values of its children. Which of the following is a max-heap?



5. If we implement heap as maximum heap , adding a new node of value 15 to the left most node of right subtree . What value will be at leaf nodes of the right subtree of the heap?



6. If we implement heap as min-heap, deleting root node (value 1)from the heap. What would be the value of root node after second iteration if leaf node (value 100) is chosen to replace the root at start?



8.8 References

[1]. *Skiena, Steven (2008). "Searching and Sorting". The Algorithm Design Manual. Springer. p. 109. doi:10.1007/978-1-84800-070-4_4. ISBN 978-1-84800-069-8. [H]eapsort is nothing but an implementation of selection sort using the right data structure.*

[2] https://en.wikipedia.org/wiki/Heapsort

[3] https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture4.pdf

[4] **"Schaum's Outline of Data Structures with C++"**. By John R. Hubbard (Can be found in university Library)

[5] "Data Structures and Program Design", Robert L. Kruse, 3rd Edition, 1996.

[6] "Data structures, algorithms and performance", D. Wood, Addison-Wesley, 1993

[7] "Advanced Data Structures", Peter Brass, Cambridge University Press, 2008

[8] "Data Structures and Algorithm Analysis", Edition 3.2 (C++ Version), Clifford A. Shaffer, Virginia Tech, Blacksburg, VA 24061 January 2, 2012

[9] "C++ Data Structures", Nell Dale and David Teague, Jones and Bartlett Publishers, 2001.

[10] "Data Structures and Algorithms with Object-Oriented Design Patterns in C++", Bruno R. Preiss,

Chapter 9

Graphs

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

A Graph consists of a finite set of vertices (or nodes) and set of edges which connect a pair of nodes.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender, locale etc.

In this chapter we will discuss some basic graph terminology of graphs and then define two fundamental representations for graphs, the adjacency matrix and adjacency list. Then we will discuss algorithms for finding the minimum-cost spanning tree, useful for determining lowest-cost connectivity in a network. We will end this chapter by presenting the two most commonly used graph traversal algorithms, called depth-first and breadth-first search, with their application.

9.1 Basic Terminology of Graphs

A graph G = (V,E) consists of a set of vertices V and a set of edges E, such that each edge in E is a connection between a pair of vertices in V. The number of vertices is written |V|, and the number of edges is written |E|.

Undirected Graph: A graph whose edges are unordered pairs of vertices. That is, each edge connects two vertices where edge (u, v) = edge (v, u).

Directed Graph: A graph whose edges are ordered pairs of vertices. That is, each edge can be followed from one vertex to another vertex where edge (u, v) goes from vertex u to vertex v.

Acyclic G (a) A graph with no path that starts ar ` ` ; at the same vertex.

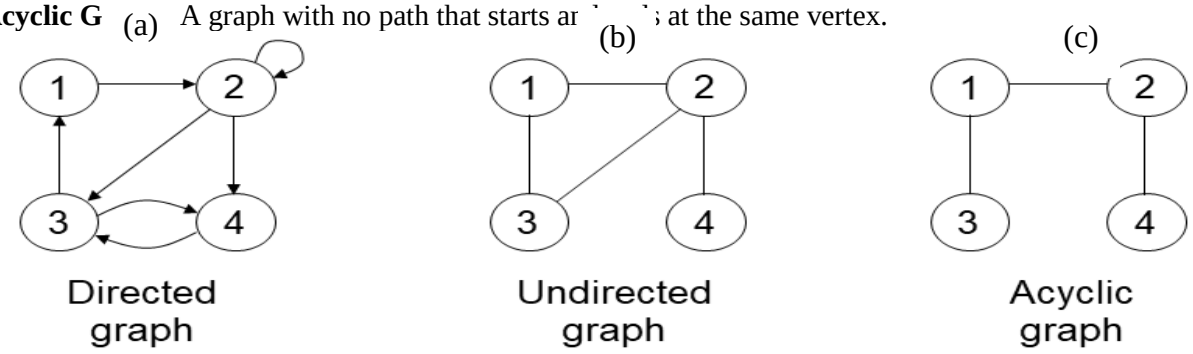


Figure 9.1: An example of (a) directed, (b) undirected and (c) acyclic graph

Another example of undirected graph G=(V,E) in figure 9.2, where V={1,2,3,4,5,6}andE={{1,2},{1,5},{2,5},{3,6}} and the vertex 4 is isolated. Vertex 1,2,5 has degree 2; 3,6 has degree 1; vertex 4 has degree 0. Vertex 3 is adjacent to vertex 6 and vice versa; {1, 5} is adjacent to 2; 4 is not adjacent to any other vertex.

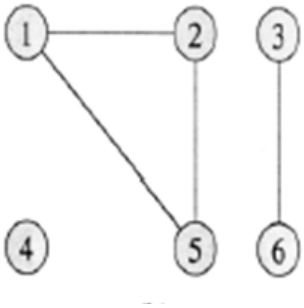


Figure 9.2: An example of undirected graph

Another example of **directed** graph G= (V, E)in figure 9.3, where V = {1, 2, 3, 4, 5, 6} and E={ (1,2), (2,2), (2,4), (2,5), (4,1), (4,5), (5,4), (6,3) }. The edge (2,2) is **self-loop**. Vertex 5 has **in-degree** 2 and **out-degree** 1. Vertex 4 is **adjacent** to vertex 5; {1, 5} is **adjacent** to 4; 3 is not **adjacent** to any other vertex except 6.

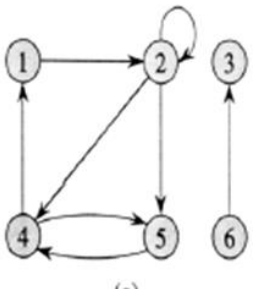


Figure 9.3: An example of directed graph

Complete graph: When every vertex is strictly connected to each other. (The number of edges in the graph is maximum).

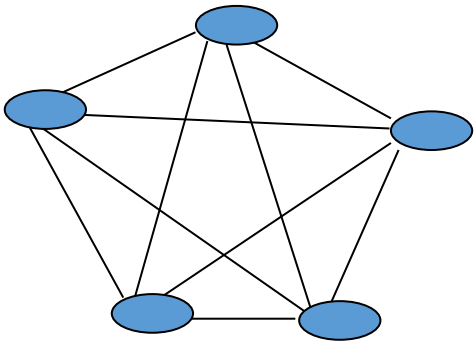


Figure 9.4: An example of Complete Graph

Dense graph: When the number of edges in the graph is close to maximum. (*adjacency matrix* is used to store info for this). A dense graph is one where there are many edges, but not necessarily as many as in a complete graph. This term is intentionally vague and is intended to convey a general sense that the number of edges can be expected to be large with respect to the number of vertices.

Sparse graph: When number of edges in the graph is very few. (*adjacency list* is used to store info for this)

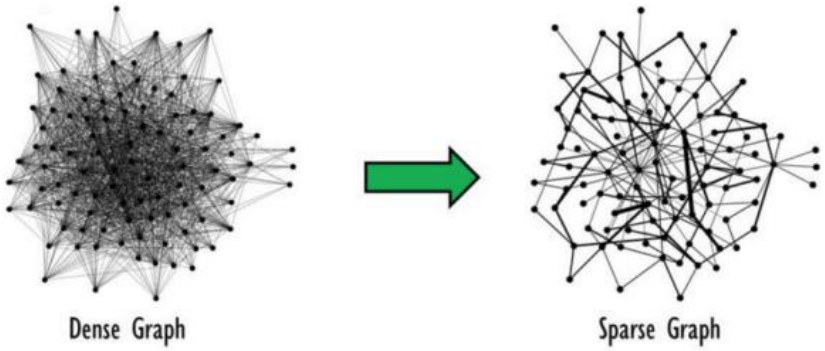


Figure 9.4: An example of dense and sparse graph

Weighted graph: associates weights with either the edges or the vertices
DAG: Directed acyclic graphs
Connected: if every vertex of a graph can *reach* every other vertex, i.e., every pair of vertices is connected by a path
Strongly connected: every 2 vertices are reachable from each other (in a digraph)
Connected Component: equivalence classes of vertices under "is reachable from" relation. Simply put, it is a subgraph in which any two vertices are **connected** to each other by paths, and which is **connected** to no additional vertices in the super graph.

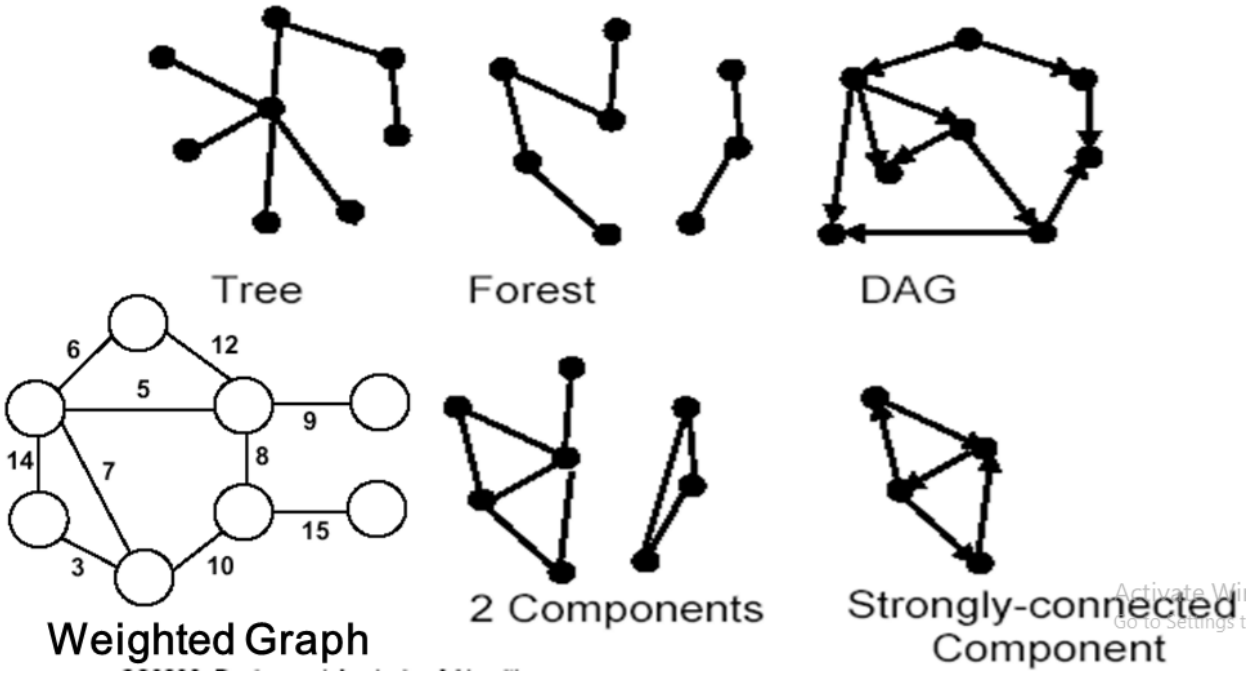


Figure 9.5: An example of tree, forest, DAG, weighted graph, components and strongly connected component

Degree of a vertex v: The degree of vertex v in a graph G, written d (v), is the number of edges incident to v, except that each loop at v counts twice (*in-degree* and *out-degree* for directed graphs). For example, in figure 9.6, d())=0, d(1)=1, d(2)=2, d(3)=3, d(4)= 3, d(5)=5, d(6)=2.

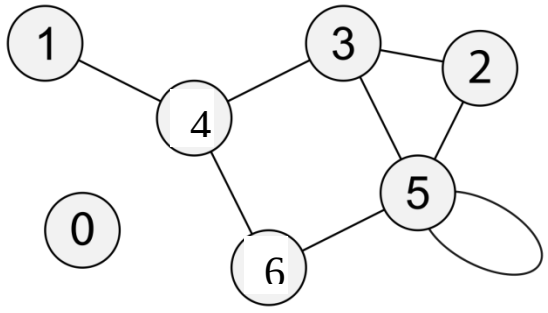


Figure 9.6: An example of undirected graph

9.1.1 Graph Applications

In **Computer science** graphs are used to represent the flow of computation. **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices. In **Facebook**, users are considered to be the vertices and if they are friends then there is

an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of **undirected graph**. In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of **Directed graph**. It was the basic idea behind Google Page Ranking Algorithm. In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur. Here we outline just some of the many applications of graphs.

- State-space search in Artificial Intelligence
- Geographical information systems, electronic street directory[Figure 9.7]
- Logistics and supply chain management
- Telecommunications network design[Figure 9.8]
- Many more industry applications
- The graphic representation of world wide web (www)
- Resource allocation graph for processes that are active in the system.
- The graphic representation of a map[Figure 9.9]
- Scene graphs: The contents of a visual scene are also managed by using graph data structure.

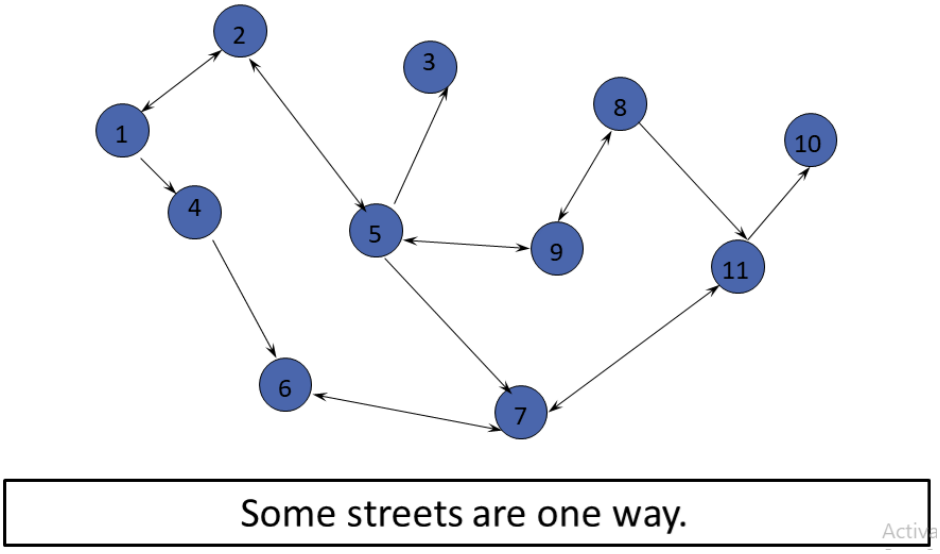


Figure 9.7: An example of directed graph which represents street map

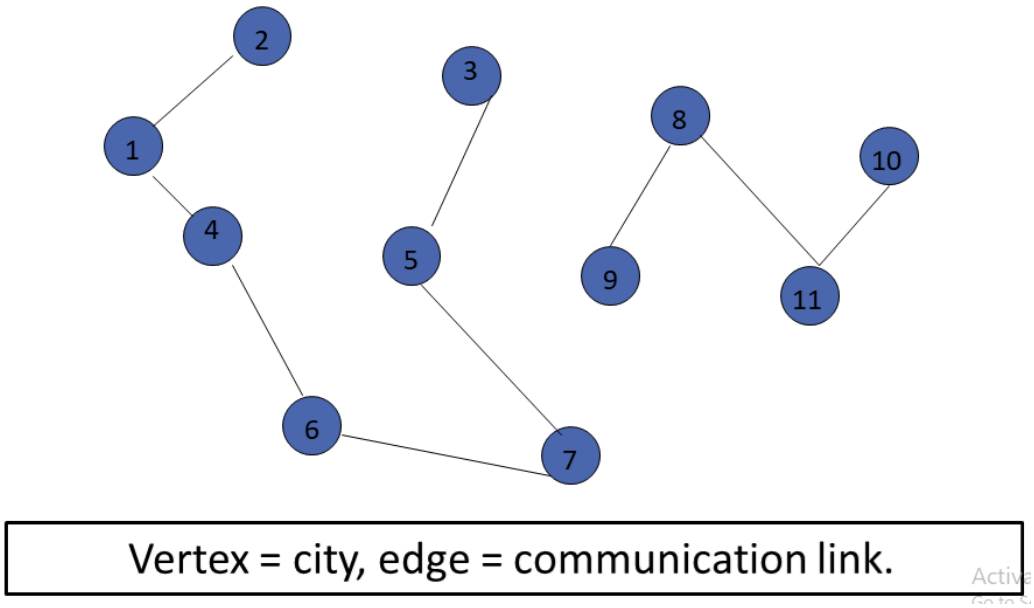


Figure 9.8: An example of undirected graph which represents Telecommunications network design

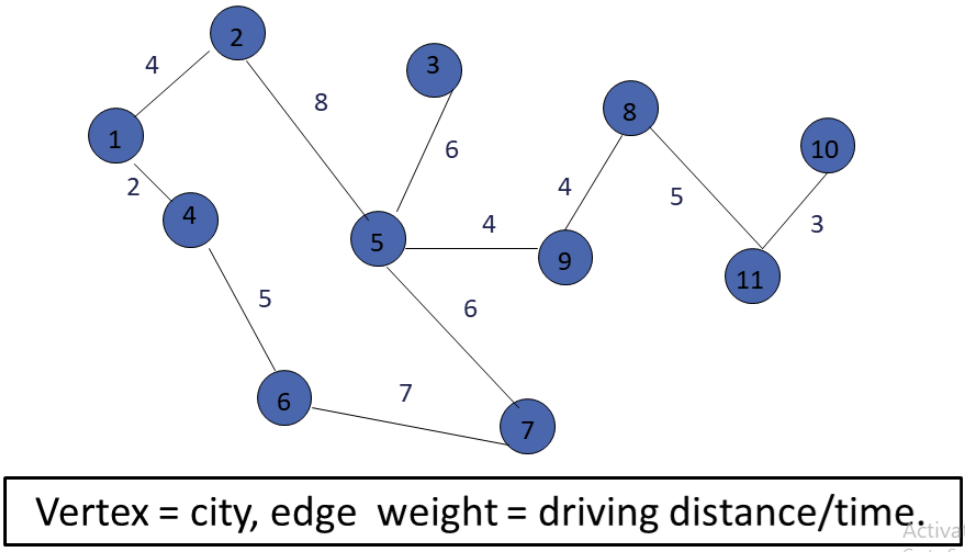


Figure 9.9: An example of undirected weighted graph which represents city map

9.1.2 Graph Representation

There are two commonly used methods for representing graphs. The adjacency matrix is illustrated by Figure 9.10(b). The adjacency matrix for a graph is a |V|x|V| array. Assume that |V| = n and that the vertices are labeled from V_0 through V_{n-1}. Row i of the adjacency matrix contains entries for Vertex V_i. Column j in row i is marked if there is an edge from V_i to V_j and is not marked otherwise. Thus, the adjacency matrix requires one bit at each position. Alternatively, if we wish to associate a number with each edge, such as the weight or distance

between two vertices, then each matrix position must store that number. In either case, the space requirements for the adjacency matrix are $\Theta(|V|^2)$.

The second common representation for graphs is the adjacency list, illustrated by Figure 9.10(c). The adjacency list is an array of linked lists. The array is |V| items long, with position i storing a pointer to the linked list of edges for Vertex V_i. This linked list represents the edges by the vertices that are adjacent to Vertex V_i.

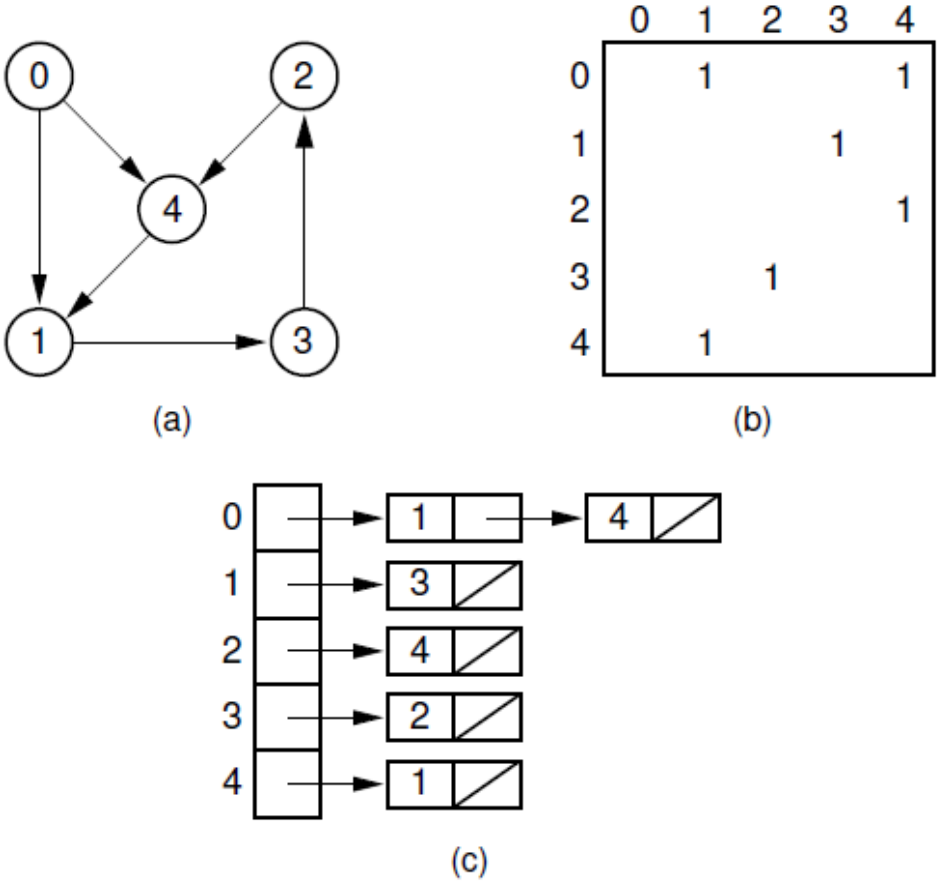


Figure 9.10 Two graph representations. (a) A directed graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

The storage requirements for the adjacency list depend on both the number of edges and the number of vertices in the graph. There must be an array entry for each vertex (even if the vertex is not adjacent to any other vertex and thus has no elements on its linked list), and each edge must appear on one of the lists. Thus, the cost is $\Theta(|V| + |E|)$.

Both the adjacency matrix and the adjacency list can be used to store directed or undirected graphs. Each edge of an undirected graph connecting Vertices U and V is represented by two directed edges: one from U to V and one from V to U. Figure 9.11 illustrates the use of the adjacency matrix and the adjacency list for undirected graphs.

Which graph representation is more space efficient depends on the number of edges in the graph. The adjacency list stores information only for those edges that actually appear in the graph, while the adjacency matrix requires space for each potential edge, whether it exists or not. However, the adjacency matrix requires no overhead for pointers, which can be a substantial cost, especially if the only information stored for an edge is one bit to indicate its existence. As the graph becomes denser, the adjacency matrix becomes relatively more space efficient. Sparse graphs are likely to have their adjacency list representation be more space efficient.

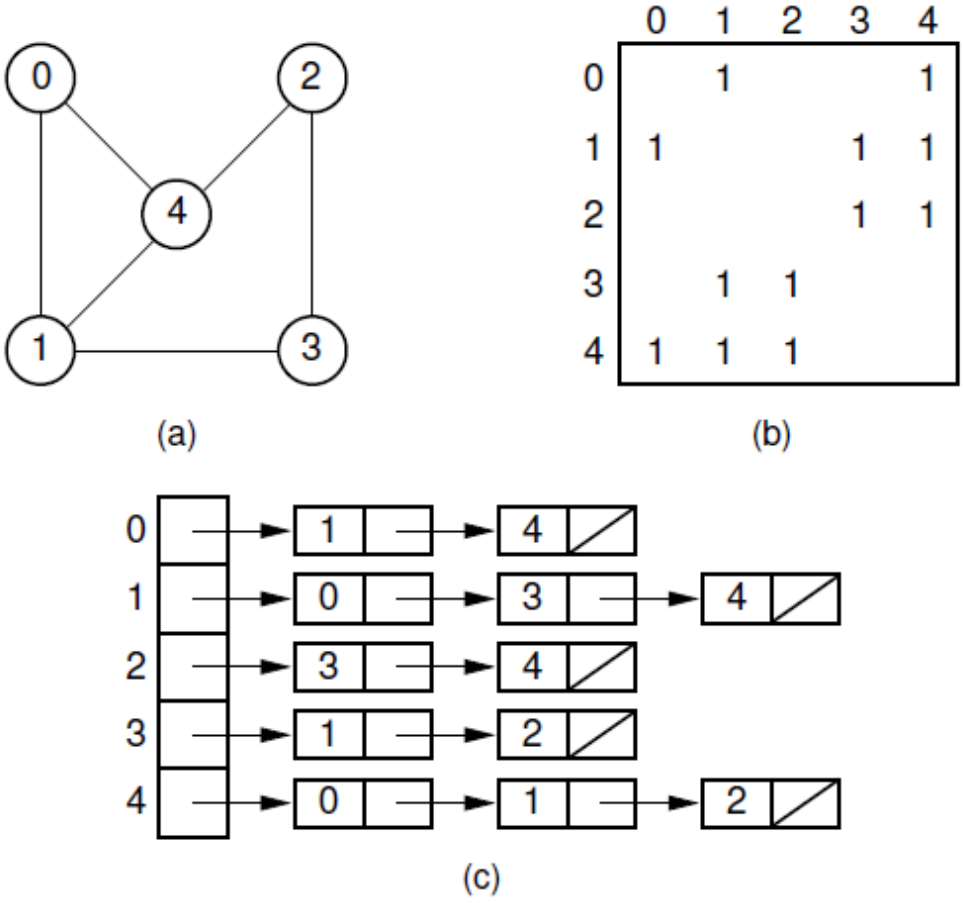


Figure 9.11 Using the graph representations for undirected graphs. (a) An undirected graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list

9.2 Minimum Spanning Tree

The **minimum-cost spanning tree (MST)** problem takes as input a connected, undirected graph G, where each edge has a distance or weight measure attached. The MST is the graph containing the vertices of G along with the subset of G's edges that (1) has minimum total cost as measured by summing the values for all of the edges in the subset, and (2) keeps the vertices connected. Applications where a solution to this problem is useful include soldering the shortest set of wires needed to connect a set of terminals on a circuit board, and connecting a set of cities by telephone lines in such a way as to require the least amount of cable.

The MST contains no cycles. If a proposed MST did have a cycle, a cheaper MST could be had by removing any one of the edges in the cycle. Thus, the MST is a free tree with |V|-1 edges. The name "minimum-cost spanning tree" comes from the fact that the required set of edges forms a tree, it spans the vertices (i.e., it connects them together), and it has minimum cost. Figure 9.12 shows the MST for an example graph.

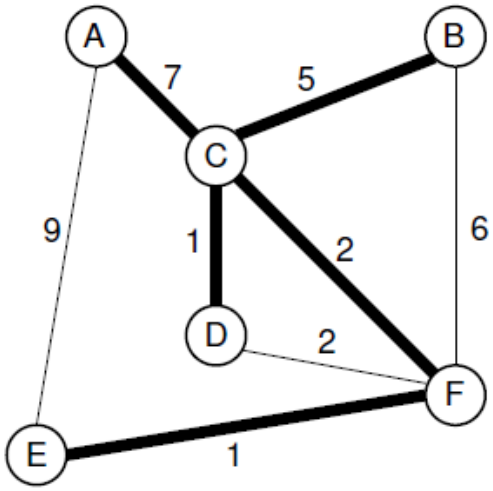


Figure 9.12 A graph and its MST. All edges appear in the original graph. Those edges drawn with heavy lines indicate the subset making up the MST. Note that edge (C, F) could be replaced with edge (D, F) to form a different MST with equal cost.

9.2.1 Applications of Minimum Spanning Tree

Minimum Spanning Tree (MST) problem: Given connected graph G with positive edge weights, find a min weight set of edges that connects all of the vertices. MST is fundamental problem with diverse applications.

1) **Network design (***telephone, electrical, hydraulic, TV cable, computer, road)*

The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

2) **Approximation algorithms for NP-hard problems (** *traveling salesperson problem*, *Steiner tree)*

A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once.
Note that if you have a path visiting all points exactly once, it's a special kind of tree. If you have a path visiting some vertices more than once, you can always drop some edges to get a tree. So in general the MST weight is less than the TSP weight, because it's a minimization over a strictly larger set.

On the other hand, if you draw a path tracing around the minimum spanning tree, you trace each edge twice and visit all points, so the TSP weight is less than twice the MST weight. Therefore this tour is within a factor of two of optimal.

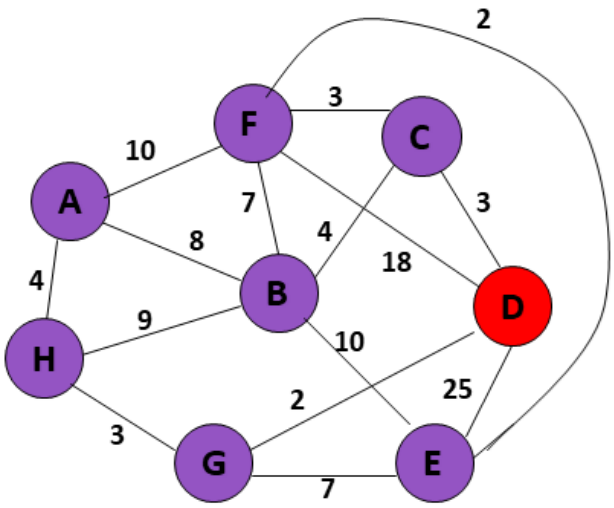
3) **Indirect applications.**
   – max bottleneck paths
   – LDPC codes for error correction
   – image registration with Renyi entropy
   – learning salient features for real-time face verification
   – reducing data storage in sequencing amino acids in a protein
   – model locality of particle interactions in turbulent fluid flows
   – auto configuration protocol for Ethernet bridging to avoid cycles in a network
4) **Cluster analysis**
   k clustering problem can be viewed as finding an MST and deleting the k-1 most expensive edges.

9.2.2 Prim's Algorithm

The first of our two algorithms for finding MSTs is commonly referred to as Prim's algorithm. Prim's algorithm is very simple. Start with any Vertex N in the graph, setting the MST to be N initially. Pick the least-cost edge connected to N. This edge connects N to another vertex; call this M. Add Vertex M and Edge (N, M) to the MST. Next, pick the least-cost edge coming from either N or M to any other vertex in the graph. Add this edge and the new vertex it reaches to the MST. This process continues, at each step expanding the MST by selecting the least-cost edge from a vertex currently in the MST to a vertex not currently in the MST.

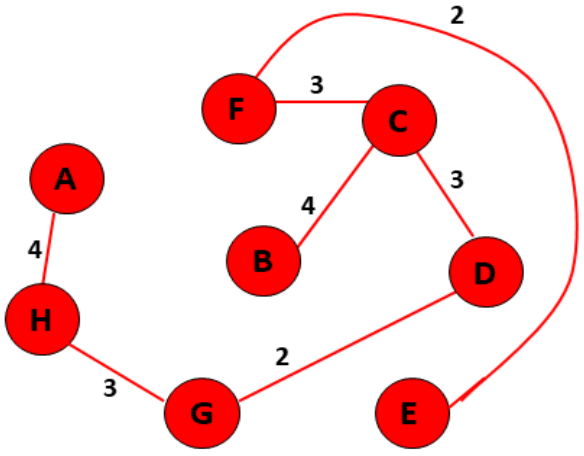
Pseudocode

```
T = a spanning tree containing a single node s;
E = set of edges adjacent to s;
while T does not contain all the nodes {
      remove an edge (v, w) of lowest cost from E
      if w is already in T then discard edge (v, w)
      else {
            add edge (v, w) and node w to T
            add to E the edges adjacent to w
      }
```



	K	d_v	p_v
A			
B			
C			
D	T	0	–
E			
F			
G			
H			

Figure 9.13: An example of undirected weighted graph and a table where K represents visit information, d_v represents distance and p_v represents parent for each vertex v.



	K	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Done

Figure 9.14: Final MST with minimum cost 21.

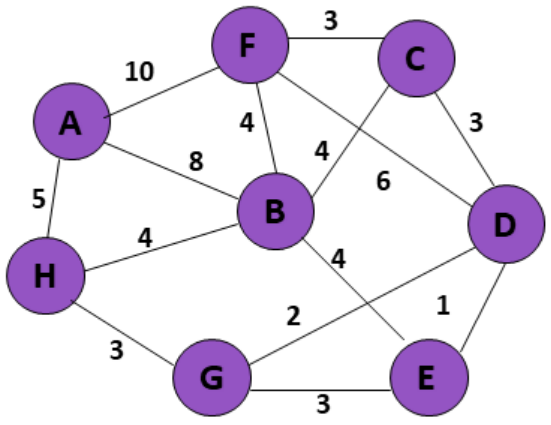
9.2.3 Kruskal's Algorithm

Our next MST algorithm is commonly referred to as Kruskal's algorithm. Kruskal's algorithm is also a simple, greedy algorithm. First partition the set of vertices into |V| equivalence classes, each consisting of one vertex. Then process the edges in order of weight. An edge is added to the MST, and two equivalence classes combined, if the edge connects two vertices in different equivalence classes. This process is repeated until only one equivalence class remains.

Pseudocode :

```
T = empty spanning tree;
E = set of edges;
N = number of nodes in graph;

while T has fewer than N - 1 edges {
      remove an edge (v, w) of lowest cost from E
      if adding (v, w) to T would create a cycle
           then discard (v, w)
           else add (v, w) to T
}
```

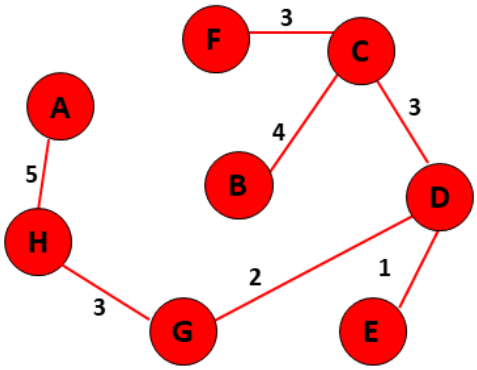
We can find an edge of lowest cost just by sorting the edges. Efficient testing for a cycle requires a fairly complex algorithm (UNION-FIND) which we don't cover in this course. Consider an undirected weighted graph in figure 9.15. The figure 9.16 shows the final MST after applying kruskal's algorithm.



edge	d_v	
(D,E)	1	
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Figure 9.15: An example of undirected weighted graph and a table for sorted list of edges.



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

Done

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

} not considered

Figure 9.16: Final MST with minimum cost 21.

9.3 Graph Traversal Algorithms

Often it is useful to visit the vertices of a graph in some specific order based on the graph's topology. This is known as a graph traversal and is similar in concept to a tree traversal. Recall that tree traversals visit every node exactly once, in some specified order such as preorder, in order, or post order. Multiple tree traversals exist because various applications require the nodes to be visited in a particular order. For example, to print a BST's nodes in ascending order requires an in order traversal as opposed to some other traversal. Standard graph traversal orders also exist. Each is appropriate for solving certain problems. For example, many problems in artificial intelligence programming are modeled using graphs. The problem domain may consist of a large collection of states, with connections between various pairs of states. Solving the problem may require getting from a specified start state to a specified goal state by moving between states only through the connections. Typically, the start and goal states are not directly connected. To solve this problem, the vertices of the graph must be searched in some organized manner.

Graph traversal algorithms typically begin with a start vertex and attempt to visit the remaining vertices from there. Graph traversals must deal with a number of troublesome cases. First, it may not be possible to reach all vertices from the start vertex. This occurs when the graph is not connected. Second, the graph may contain cycles, and we must make sure that cycles do not cause the algorithm to go into an infinite loop.

9.3.1 DFS

The first method of organized graph traversal is called depth-first search (DFS). The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the

nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows: Pick a starting node and push all its adjacent nodes into a stack. Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack. Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop. The figure 9.17 shows an illustration of how DFS works.

```
depthFirstSearch(v)
{  Label vertex v as reached.
   for (each unreached vertex u adjacenct from v)
       depthFirstSearch(u);
}
```

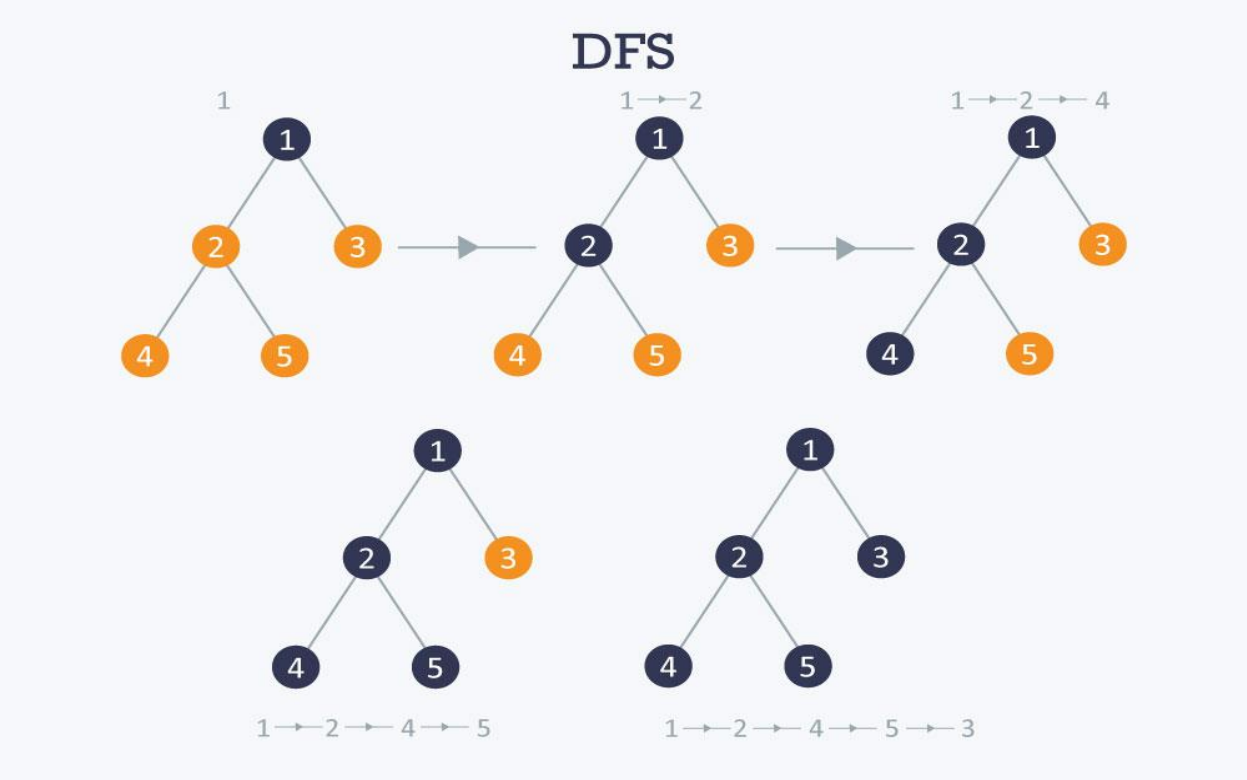


Figure 9.17: An illustration of DFS Algorithm

9.3.2 DFS: Classification of Edges

The traversal algorithm DFS can be used to classify edges of G. Following are the list of edges which can be found by using DFS algorithm.

- **Tree edges**: Edges in the depth-first forest.
- **Back edges**: Edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree (where v is not the parent of u). It also applies for self-loops.
- **Forward edges**: Non-tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
- **Cross edges**: All other edges.

DFS yields valuable information about the structure of a graph. In DFS of an undirected graph we get only tree and back edges; no forward or back-edges. The graphs in figure 9.18 show the classification of edges.

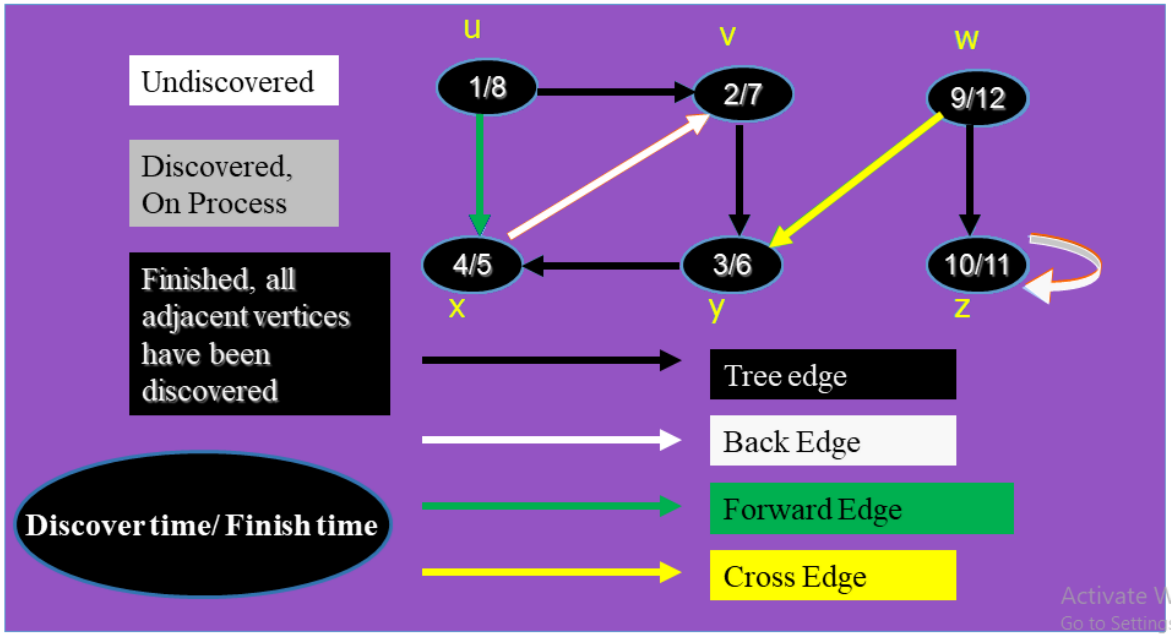


Figure 9.18: An illustration of classifying edge using DFS Algorithm

9.3.3 Applications of DFS

Depth-first search (DFS) is an algorithm (or technique) for traversing a graph. Following are the problems that use DFS as a building block.

1) For a weighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

2) **Detecting cycle in a graph**
A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See this for details)

3) **Path Finding**
We can specialize the DFS algorithm to find a path between two given vertices u and z.

i) Call DFS(G, u) with u as the start vertex.
ii) Use a stack S to keep track of the path between the start vertex and the current vertex.
iii) As soon as destination vertex z is encountered, return the path as the contents of the stack

4) **Topological Sorting**

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when re computing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in make files, data serialization, and resolving symbol dependencies in linkers.

5) **To test if a graph is bipartite**
We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black! See this for details.

6) **Finding Strongly Connected Components of a graph**

A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See this for DFS based algorithm for finding Strongly Connected Components)

7) **Solving puzzles with only one solution**, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

9.3.4 BFS

Our second graph traversal algorithm is known as a breadth-first search (BFS). BFS examines all vertices connected to the start vertex before visiting vertices further away. BFS is implemented similarly to DFS, except that a queue replaces the recursion stack. Note that if the graph is a tree and the start vertex is at the root, BFS is equivalent to visiting vertices level by level from top to bottom. Figure 9.19 shows a graph and the corresponding breadth-first search tree. Figure 9.20 illustrates the BFS process for the graph of Figure 9.19(a).

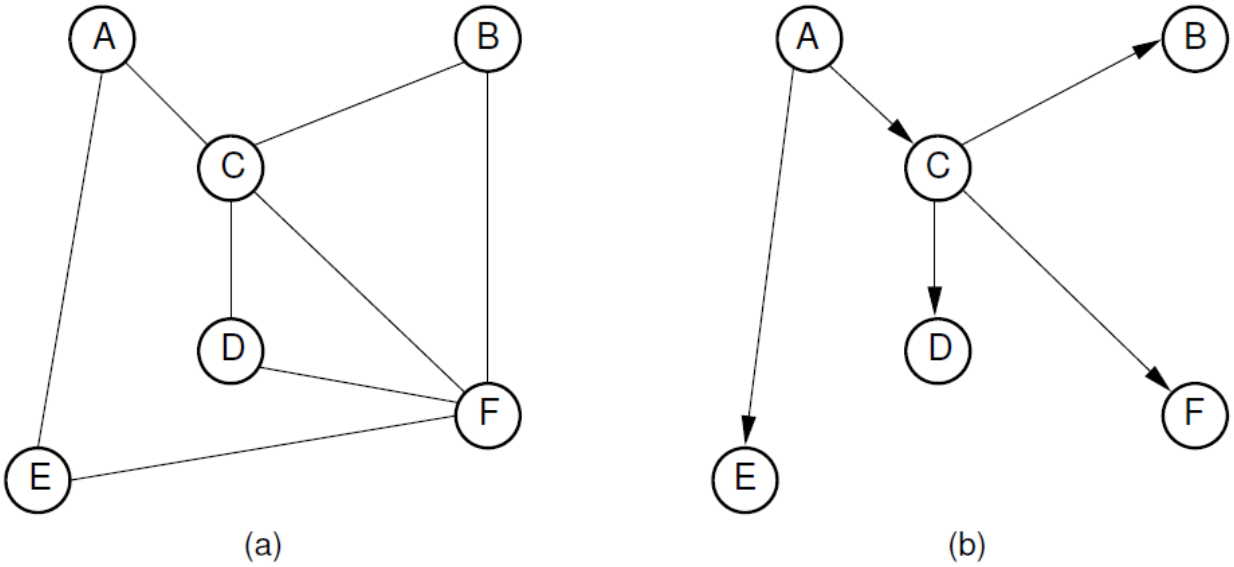


Figure 9.19 (a) A graph. (b) The breadth-first search tree for the graph when starting at Vertex *A*.

A	

Initial call to BFS on A.
Mark A and put on the queue.

C	E	

Dequeue A.
Process (A, C).
Mark and enqueue C. Print (A, C)
Process (A, E).
Mark and enqueue E. Print(A, E).

E	B	D	F	

Dequeue C.
Process (C, A). Ignore.
Process (C, B).
Mark and enqueue B. Print (C, B).
Process (C, D).
Mark and enqueue D. Print (C, D).
Process (C, F).
Mark and enqueue F. Print (C, F).

B	D	F	

Dequeue E.
Process (E, A). Ignore.
Process (E, F). Ignore.

D	F	

Dequeue B.
Process (B, C). Ignore.
Process (B, F). Ignore.

F	

Dequeue D.
Process (D, C). Ignore.
Process (D, F). Ignore.

Dequeue F.
Process (F, B). Ignore.
Process (F, C). Ignore.
Process (F, D). Ignore.
BFS is complete.

Figure 9.20 A detailed illustration of the BFS process for the graph of Figure 11.11(a) starting at Vertex *A*. The steps leading to each change in the queue are described.

9.3.5 Applications of BFS

We have earlier discussed Breadth First Traversal Algorithm for Graphs. We have also discussed applications of Depth First Traversal. Now we will discuss the applications of Breadth First Search.

1) **Shortest Path and Minimum Spanning Tree for unweighted graph**
   In an unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using the minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2) **Peer to Peer Networks**
   In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

3) **Crawlers in Search Engines**
   Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of the built tree can be limited.

4) Social Networking Websites
   In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.

5) **GPS Navigation systems**
   Breadth First Search is used to find all neighboring locations.

6) **Broadcasting in Network**
   In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

7) In Garbage Collection
   Breadth First Search is used in copying garbage collection using Cheney's algorithm. Refer this and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:

8) Cycle detection in undirected graph
   In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. We can use BFS to detect cycle in a directed graph also,

9) **Ford–Fulkerson algorithm**
   In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to $O(VE^2)$.

10) **To test if a graph is Bipartite**
We can either use Breadth First or Depth First Traversal.

11) **Path Finding**
We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

12) Finding all nodes within one connected component
We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

9.4 Exercises

1) Draw the adjacency matrix representation for the graph of Figure 9.21.
2) Draw the adjacency list representation for the same graph.
3) Show the DFS tree for the graph of Figure 9.21, starting at Vertex 1.
4) Show the BFS tree for the graph of Figure 9.21, starting at Vertex 1.
5) Does either Prim's or Kruskal's algorithm work if there are negative edge weights?
6) List the order in which the edges of the graph in Figure 9.21 are visited when running Prim's MST algorithm starting at Vertex 3. Show the final MST.
7) List the order in which the edges of the graph in Figure 9.21 are visited when running Kruskal's MST algorithm. Each time an edge is added to the MST, show the result on the equivalence array. Show the final MST.

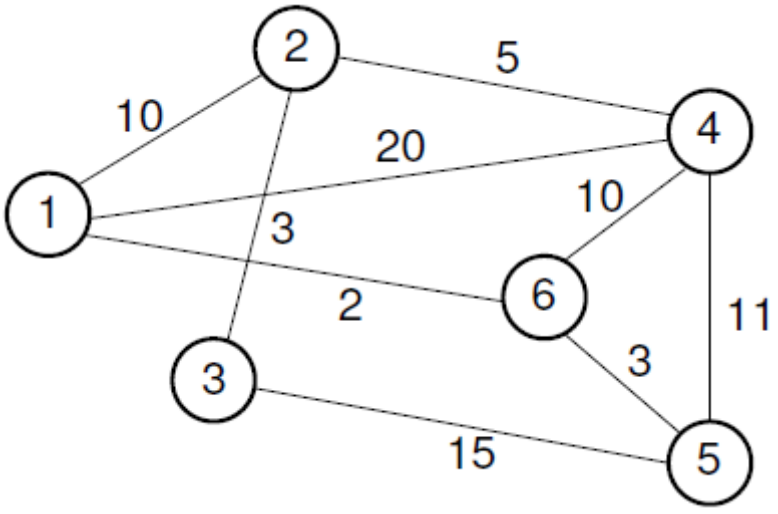


Figure 9.21: Example graph for Chapter exercises.

9.5 References

[1] https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

[2] https://www.geeksforgeeks.org/applications-of-depth-first-search/

[3] https://en.wikipedia.org/wiki/Data_structure

[4] https://visualgo.net/en/dfsbfs?slide=1

[5] "Schaum's Outline of Data Structures with C++". By John R. Hubbard (Can be found in university Library)

[6] "Data Structures and Program Design", Robert L. Kruse, 3rd Edition, 1996.

[7] "Data structures, algorithms and performance", D. Wood, Addison-Wesley, 1993

[8] "Advanced Data Structures", Peter Brass, Cambridge University Press, 2008

[9] "Data Structures and Algorithm Analysis", Edition 3.2 (C++ Version), Clifford A. Shaffer, Virginia Tech, Blacksburg, VA 24061 January 2, 2012

[10] "C++ Data Structures", Nell Dale and David Teague, Jones and Bartlett Publishers, 2001.

[11] "Data Structures and Algorithms with Object-Oriented Design Patterns in C++", Bruno R. Preiss

[12] https://www.geeksforgeeks.org/applications-of-breadth-first-traversal/