CHAPTER ONE

1. ALGORITHM ANALYSIS CONCEPTS

1.1. Introduction to Data Structures and Algorithms Analysis

A program is written in order to solve a problem. A solution to a problem actually consists of two things:

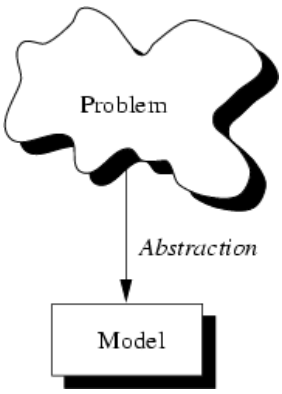
- A way to organize the data
- Sequence of steps to solve the problem

The way data are organized in a computer's memory is said to be *data structure* and the sequence of computational steps to solve a problem is said to be an *algorithm*. Therefore, a program is nothing but data structures plus algorithms.

A data structure is a systematic way of organizing and accessing data and an algorithm is a step-by-step procedure for performing some task in a finite amount of time.

1.1.1. Introduction to Data Structures

Given a problem, the first step to solve the problem is obtaining one's own abstract view, or *model*, of the problem. This process of modeling is called *abstraction*.



The model defines an abstract view to the problem. This implies that the model focuses only on problem related stuff and that a programmer tries to define the *properties* of the problem.

These properties include

- The *data* which are affected and
- The *operations* that are involved in the problem.

With abstraction you create a well-defined entity that can be properly handled. These entities define the *data structure* of the program.

An entity with the properties just described is called an *abstract data type* (ADT).

i. Abstract Data Types

An abstract data type (ADT) is a set of objects together with a set of operations. Abstract data types are mathematical abstractions; nowhere in an ADT's definition is there any mention of how the set of operations is implemented. Objects such as lists, sets, and graphs, along with their operations, can be viewed as ADTs, just as integers, real's, and booleans are data types. Integers, real's, and booleans have operations associated with them, and so do ADTs. For the set ADT, we might have such operations as add, remove, size, and contains.

An ADT is a mathematical model of a data structure that specifies the type of the data stored, the operations supported on them, and the types of the parameters of the operations. An ADT specifies what each operation does, but not how it does it.

The ADT specifies:

1. What can be stored in the Abstract Data Type?
2. What operations can be done on/by the Abstract Data Type?

For example, if we are going to model employees of an organization:

- This ADT stores employees with their relevant attributes and discarding irrelevant attributes.
- This ADT supports hiring, firing, retiring, operations.

A data structure is a language construct that the programmer has defined in order to implement an abstract data type.

ii. Abstraction

Abstraction is a process of classifying characteristics as relevant and irrelevant for the particular purpose at hand and ignoring the irrelevant ones.

The notion of abstraction is to distill a complicated system down to its most fundamental parts and describe these parts in a simple, precise language. Typically, describing the parts of a system involves naming them and explaining their functionality. Applying the abstraction paradigm to the design of data structures gives rise to abstract data types (ADTs).

Applying abstraction correctly is the essence of successful programming

How do data structures model the world or some part of the world?

- The value held by a data structure represents some specific characteristic of the world
- The characteristic being modeled restricts the possible values held by a data structure
- The characteristic being modeled restricts the possible operations to be performed on the data structure.

Note: Notice the relation between characteristic, value, and data structures

Where are algorithms, then?

1.1.2. Algorithms

An algorithm is a well-defined computational procedure that takes some value or a set of values as input and produces some value or a set of values as output. Data structures model the static part of the world. They are unchanging while the world is changing. In order to model the dynamic part of the world we need to work with algorithms. Algorithms are the dynamic part of a program's world model.

An algorithm transforms data structures from one state to another state in two ways:

- An algorithm may change the value held by a data structure
- An algorithm may change the data structure itself

The quality of a data structure is related to its ability to successfully model the characteristics of the world. Similarly, the quality of an algorithm is related to its ability to successfully simulate the changes in the world.

However, independent of any particular world model, the quality of data structure and algorithms is determined by their ability to work together well. Generally speaking, correct data structures lead to simple and efficient algorithms and correct algorithms lead to accurate and efficient data structures.

Properties of an algorithm

What makes an algorithm good?

- ***Finiteness***: Algorithm must complete after a finite number of steps.
- ***Definiteness***: Each step must be clearly defined, having one and only one interpretation. At each point in computation, one should be able to tell exactly what happens next.
- ***Sequence***: Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted.
- ***Feasibility***: It must be possible to perform each instruction.
- ***Correctness***: It must compute correct answer for all possible legal inputs.
- ***Language Independence***: It must not depend on any one programming language.
- ***Completeness***: It must solve the problem completely.
- ***Effectiveness***: It must be possible to perform each step exactly and in a finite amount of time.
- ***Efficiency***: It must solve with the least amount of computational resources such as time and space.
- ***Generality***: Algorithm should be valid on all possible inputs.
- ***Input/Output***: There must be a specified number of input values, and one or more result values.

1.2. Algorithm Analysis Concepts

Algorithm analysis refers to the process of determining the amount of computing time and storage space required by different algorithms. In other words, it's a process of predicting the resource requirement of algorithms in a given environment.

It is the study of efficiency of programs. Input size of the program, machine type used, implementation quality, running time of algorithm and others affect the efficiency of a program.

Once an algorithm is given for a problem and determined to be correct, the next step is to determine the amount of resources that the algorithm will require. This step is called algorithm analysis.

In order to solve a problem, there are many possible algorithms. One has to be able to choose the best algorithm for the problem at hand using some scientific method. To classify some data structures and algorithms as good, we need precise ways of analyzing them in terms of resource requirement. The main resources are:

- Running Time
- Memory Usage
- Communication Bandwidth

multi resource

Algorithm analysis tries to estimate these resources required to solve a problem at hand. Running time is usually treated as the most important since computational time is the most precious resource in most problem domains.

There are two approaches to measure the efficiency of algorithms:

- **Empirical (Experimental) method**: Programming competing algorithms and trying them on different instances. This method is used for absolute time measurement.
- **Analytical (Theoretical) Method**: Determining the quantity of resources required mathematically (like execution time, memory space, etc.) needed by each algorithm.

However, it is difficult to use actual clock-time as a consistent measure of an algorithm's efficiency, because clock-time can vary based on many things. For example,

- Specific processor speed
- Current processor load
- Specific data for a particular run of the program
  - Input Size
  - Input Properties
- Operating Environment

Accordingly, we can analyze an algorithm according to the number of operations required, rather than according to an absolute amount of time involved. This can show how an algorithm's efficiency changes according to the size of the input.

1.2.1. Complexity Analysis

Complexity Analysis is the systematic study of the cost of computation, measured either in time units or in operations performed, or in the amount of storage space required.
The goal is to have a meaningful measure that permits comparison of algorithms independent of operating platform.
There are two things to consider:

- **Time Complexity**: Determine the approximate number of operations required to solve a problem of size n. The running times of operations on the data structure should be as small as possible.
- **Space Complexity:** Determine the approximate memory required to solve a problem of size n. The data structure should use as little memory as possible.

The critical resource for a program is most often its running time. *Running time is the amount of time that any algorithm takes to run.* However, you cannot pay attention to running time alone. You must also be concerned with other factors such as the space required to run the program (both main memory and disk space). The primary analysis tool we use in this course involves characterizing the running times of algorithms and data structure operations. Running time is a natural measure of "goodness," since time is a precious resource - computer solutions should run as fast as possible.

Many factors affect the running time of a program. Some relate to the environment in which the program is compiled and run. Such factors include the speed of the computer's CPU, bus, and peripheral hardware. The programming language and the quality of code generated by a particular compiler can have a significant effect also. It is also affected by amount of input that it must process, the hardware environments (like the processor, memory, disk, etc.) and

software environments (such as operating system, programming language, compiler, interpreter, etc.) in which the algorithm is implemented, compiled, and executed.

Nevertheless, in spite of the possible variations that come from different environmental factors, we would like to focus on the relationship between the running time of an algorithm and the size of its input. The running time of an algorithm is thus a function of the input size.

Complexity analysis involves two distinct phases:

- **Algorithm Analysis**: Analysis of the algorithm or data structure to produce a function T (n) that describes the algorithm in terms of the operations performed in order to measure the complexity of the algorithm.
- **Order of Magnitude Analysis**: Analysis of the function T (n) to determine the general complexity category to which it belongs (constant time, linear time, logarithmic time, quadratic time, exponential time or other).

1.2.2. Complexity of Algorithms

There is no generally accepted set of rules for algorithm analysis. However, an exact count of operations is commonly used. Arbitrary time unit is assumed for analyzing the algorithm, and we have the following set of analysis rules to be considered in so doing the analysis.

Rule 1: Basic operations

The execution of the following operations takes one (1) time unit.

- Assignment Operation
- Single Input/ Output Operation
- Single Boolean Operations
- Single Arithmetic Operations
- Function Return

Rule 2: Selection statements

Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection. This can be an overestimate in some cases, but it is never an underestimate.

Rule 3: Loops

Running time for a loop is equal to the running time for the statements inside the loop body multiplied (*) by number of iterations of the loop.

Rule 4: Nested Loops

The total running time of a statement inside a group of nested loops is the running time of the statements multiplied by the product of the sizes of all the loops.

For nested loops, analyze inside out. Always assume that the loop executes the maximum number of iterations possible.

Rule 5: Function call

Running time of a function call is 1 for setup + the time for any parameter calculations + the time required for the execution of the function body.

Rule 6: Consecutive statements

For consecutive statements, the running time will be computed as the sum of the running time of the separate blocks of code.

The following examples show us how running time of the code fragments is computed.

Examples 1:

```
int count ( )
{
   int k=0;
   cout<< "Enter an integer";
   cin>>n;
   for (i=0; i<n; i++)
         k=k+1;
   return 0;
}
```

Time Units to Compute

- ✓ 1 for the assignment statement: int k=0
- ✓ 1 for the output statement.
- ✓ 1 for the input statement.
- ✓ For the loop (for):
  - 1 assignment (i = 0), *n+1* tests, and *n* increments.
  - *n* loops of 2 units for an assignment and addition.
- ✓ 1 for the return statement.

T (n)= $1+1+1+(1+n+1+n)+2n+1 = \mathbf{4n+6} = O(n)$

Example 2:

```
 int total(int n)
   {
   int sum=0;
   for (int i=1;i<=n;i++)
        sum=sum+1;
    return sum;
    }
```

Time Units to Compute

- ✓ 1 for the assignment statement: int sum=0
- ✓ In the for loop:
  - 1 assignment, *n+1* tests, and *n* increments.
  - *n* loops of 2 units for an assignment and addition.
- ✓ 1 for the return statement.

T (n)= $1+ (1+n+1+n)+2n+1 = \mathbf{4n+4} = O(n)$

Example 3:

```
void func( )
   {      int x=0, i=0, j=1;
          cout<< "Enter an Integer value";
          cin>>n;
          while (i<n){
             x++;
              i++;   }
   while (j<n)
         j++;
  }
```

Time Units to Compute

- 1 for the first assignment statement: x=0;
- 1 for the second assignment statement: i=0;
- 1 for the third assignment statement: j=1;
- 1 for the output statement.
- 1 for the input statement.
- In the first while loop:
  - *n+1* tests
  - *n* loops of 2 units for the two increment (addition) operations
- In the second while loop:
  - n tests
  - n-1 increments

T (n)= $1+1+1+1+1+(n+1+2n)+(n+n-1) = \mathbf{5n+5} = O(n)$

Example 4:

```
int sum (int n)
  {
   int partial_sum = 0;
   for (int i = 1; i <= n; i++)
         partial_sum = partial_sum +(i * i * i);
   return partial_sum;
  }
```

Time Units to Compute

- 1 for the assignment.
- 1 assignment, n+1 tests, and n increments for the loop expression (for).
- n loops of 4 units for an assignment, addition, and two multiplications.
- 1 for the return statement.

T (n) = $1+(1+n+1+n)+4n+1 = \mathbf{6n+4} = O(n)$

Example 5:

```
void func ( )
{
  int i =1, sum = 0;
  while (i < = n)
  {
   for (int j =0;j<n; j++)
     {
      sum = i + j;
     }
   i ++;
  }
}
```

Time Units to Compute

- 1 for the first assignment (i = 1).
- 1 for the second assignment (sum = 0).
- In the while loop
  - n+1 tests
  - n loops of the following
    - For the for loop

- 1 assignment, n+1 tests, and n increments.
- n loops of 2 units for an assignment and addition.

○ 1 for the increment operation.

T (n) = $1+1+(n+1)+n[(1+n+1+n+2n)+1] = \mathbf{4n^2+4n+3} = O(n^2)$

Example 6:

```
int k=0;
for (int i=1; i<n; i*=2)
  for(int j=1;j<=n;j++)
    k++;
```

Time Units to Compute

- 1 for the first assignment (k = 0).
- For the first loop (for)
  - 1 assignment, $1+\log_2 n$ tests, and $\log_2 n$ multiplication of i*=2.
  - $\log_2 n$ iterations of the following
    - For the second loop (for)
      - 1 assignment, n+1 tests, and n increments.
      - n loops of one unit (increment operation)

T (n) = $1+1+1+\log_2 n+\log_2 n+\log_2 n[(1+n+1+n)+n] = \mathbf{3}n\log_2 n+\mathbf{4}\log_2 n+\mathbf{3} = O(n\log_2 n)$

1.2.3. Formal Approach to Analysis

In the above examples we have seen that analysis is a bit complex. However, it can be simplified by using some formal approach in which case we can ignore initializations, loop control, and book keeping.

For Loops: Formally

- In general, a for loop translates to a summation. The index and bounds of the summation are the same as the index and bounds of the for loop.

```
for (int i = 1; i <= N; i++) {
    sum = sum+i;
}
```

$$\sum_{i=1}^{N} 1 = N$$

- ✓ Suppose we count the number of additions that are done. There is 1 addition per iteration of the loop, hence *N* additions in total.

Nested Loops: Formally

- Nested for loops translate into multiple summations, one for each for loop.

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= M; j++) {
        sum = sum+i+j;
    }
}
```

$$\sum_{i=1}^{N}\sum_{j=1}^{M} 2 = \sum_{i=1}^{N} 2M = 2MN$$

- ✓ Again, count the number of additions. The outer summation is for the outer for loop.

Consecutive Statements: Formally

❖ Add the running times of the separate blocks of your code

```
for (int i = 1; i <= N; i++) {
    sum = sum+i;
}
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        sum = sum+i+j;
    }
}
```

$$\left[\sum_{i=1}^{N} 1\right] + \left[\sum_{i=1}^{N}\sum_{j=1}^{N} 2\right] = N + 2N^2$$

Conditionals: Formally

❖ If (test) s1 else s2: Compute the maximum of the running time for s1 and s2.

```
if (test == 1) {
   for (int i = 1; i <= N; i++) {
       sum = sum+i;
}}
else for (int i = 1; i <= N; i++) {
          for (int j = 1; j <= N; j++) {
              sum = sum+i+j;
}}
```

$$\max\left(\sum_{i=1}^{N} 1, \sum_{i=1}^{N}\sum_{j=1}^{N} 2\right) = \max\left(N, 2N^2\right) = 2N^2$$

Example:
Suppose we have hardware capable of executing 10^6 instructions per second. How long would it take to execute an algorithm whose complexity function was T (n) = $2n^2$ on an input size of $n=10^8$?

Solution
The total number of operations to be performed would be T (10^8):

$T(10^8) = 2*(10^8)^2 = 2*10^{16}$

The number of seconds required would be given by

$T(10^8)/10^6$ so:

Running time $=2*10^{16}/10^6 = 2*10^{10}$

The number of seconds per day is 86,400 so this is about 231,480 days (634 years).

1.3. Measures of Times

In order to determine the running time of an algorithm it is possible to define three functions $T_{best}(n)$, $T_{avg}(n)$ and $T_{worst}(n)$ as the best, the average and the worst case running time of the algorithm respectively.

Average Case (T_{avg}): The amount of time the algorithm takes on an "average" set of inputs.
Worst Case (T_{worst}): The amount of time the algorithm takes on the worst possible set of inputs.
Best Case (T_{best}): The amount of time the algorithm takes on the smallest possible set of inputs.

We are interested in the worst-case time, since it provides a bound for all input – this is called the "Big-Oh" estimate.

1.4. Asymptotic Analysis

Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

The seven most important functions used in the analysis of algorithms are *constant*, *linear*, *logarithm*, *N log N*, *quadratic*, *cubic* and *exponential* functions. We use only these seven simple functions for almost all the analysis we do in this course.

The simplest function we can think of is the **constant function**. This is the function, f (n) = c, for some fixed constant c, such as c = 5, c = 27, or c = 2^{10}. That is, for any argument n, the constant function f (n) assigns the value c. In other words, it doesn't matter what the value of n is, f (n) is always be equal to the constant value c. Since we are most interested in integer functions, the most fundamental constant function is g(n) = 1, and this is the typical constant function we use in this course. Note that any other constant function, f (n) = c, can be written as a constant c times g(n). That is, f (n) = c g(n) in this case.

One of the interesting and sometimes even surprising aspects of the analysis of data structures and algorithms is the ubiquitous presence of the **logarithm function**, f(n) = $\log_b n$, for some constant b > 1. This function is defined as $x = log_b n$ if and only if $b^x = n$. By definition, $log_b 1 = 0$. The value b is known as the base of the logarithm.

Another simple yet important function is the **linear function**, $f(n) = n$. That is, given an input value n, the linear function f assigns the value n itself. This function arises in algorithm analysis any time we have to do a single basic operation for each of n elements. For example, comparing a number x to each element of an array of size n requires n comparisons. The linear function also represents the best running time we can hope to achieve for any algorithm that processes a collection of n objects that are not already in the computer's memory, since reading in the n objects itself requires n operations.

The next function is the **n-log-n function**, $f(n) = n \log n$. That is, the function that assigns to an input n the value of n times the logarithm base of n. This function grows a little faster than the linear function and a lot slower than the quadratic function. Thus, as we show on several occasions, if we can improve the running time of solving some problem from quadratic to n-log-n, we have an algorithm that runs much faster in general.

Another function that appears quite often in algorithm analysis is the **quadratic function**, $f(n) = n^2$. That is, given an input value n, the function f assigns the product of n with itself (in other words, "n squared"). The main reason why the quadratic function appears in the analysis of algorithms is that there are many algorithms that have nested loops, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times. Thus, in such cases, the algorithm performs $n \cdot n = n^2$ operations.

The other function is the **cubic function**, $f(n) = n^3$, which assigns to an input value n the product of n with itself three times. This function appears less frequently in the context of algorithm analysis than the constant, linear, and quadratic functions previously mentioned, but it does appear from time to time.

Another function used in the analysis of algorithms is the **exponential function**, $f(n) = b^n$, where b is a positive constant, called the base, and the argument n is the exponent. That is, function f (n) assigns to the input argument n the value obtained by multiplying the base b by itself n times. In algorithm analysis, the most common base for the exponential function is $b = 2$. For instance, if we have a loop that starts by performing one operation and then doubles the number of operations performed with each iteration, then the number of operations performed in the nth iteration is 2^n.

There are five notations used to describe a running time function. These are:

- Big-Oh Notation (O)
- Big-Omega Notation (Ω)
- Theta Notation (Θ)
- Little-o Notation (o)
- Little-Omega Notation (ω)

1.4.1. The Big-Oh Notation

Big-Oh notation is a way of comparing algorithms and is used for computing the complexity of algorithms; i.e., the amount of time that it takes for computer program to run. It's only concerned with what happens for very a large value of n. Therefore only the largest term in the expression (function) is needed. For example, if the number of operations in an algorithm is $n^2 - n$, n is insignificant compared to n^2 for large values of n. Hence the n term is ignored. Of course, for small values of n, it may be important. However, Big-Oh is mainly concerned with large values of n.

Formal Definition: f (n)= O (g (n)) if there exist c, k ∈ $\mathscr{R}^+$ such that for all n≥ k, f (n) ≤ c.g (n).

Examples: The following points are facts that you can use for Big-Oh problems:

- 1<=n for all n>=1
- $n<=n^2$ for all n>=1
- 2^n <=n! for all n>=4
- $\log_2 n<=n$ for all n>=2
- $n<=n\log_2 n$ for all n>=2

O(asi kessati funtion bareefamu sun gudaadha jechuudha)

1) f(n)=10n+5 and g(n)=n. Show that f(n) is O(g(n)).

To show that f(n) is O(g(n)) we must show that constants c and k such that
f(n) <=c.g(n) for all n>=k Or 10n+5<=c.n for all n>=k
Try c=15. Then we need to show that 10n+5<=15n
Solving for n we get: 5<5n or 1<=n.
So f(n) =10n+5 <=15.g(n) for all n>=1. (c=15,k=1).

constant ignore ta'an sana walitti ida'uun
coefficient isa gudaa sanaaf godhuudha

2) f(n) = $3n^2$ +4n+1. Show that f(n)=O(n^2).

4n <=$4n^2$ for all n>=1 and 1<=n^2 for all n>=1
$3n^2$ +4n+1<=$3n^2$+$4n^2$+n^2 for all n>=1
<=$8n^2$ for all n>=1
So we have shown that f(n)<=$8n^2$ for all n>=1
Therefore, f (n) is O(n^2) (c=8,k=1)

c=ida'ama conficient yoo ta'u
k=baka n irraa jalqabuudha

<u>Typical Orders</u>

Here is a table of some typical cases. This uses logarithms to base 2, but these are simply proportional to logarithms in other base.

N	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$
1	1	0	1	0	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4,096
1024	1	10	1,024	10,240	1,048,576	1,073,741,824

Demonstrating that a function f(n) is big-O of a function g(n) requires that we find specific constants c and k for which the inequality holds (and show that the inequality does in fact hold).

Big-O expresses an *upper bound* on the growth rate of a function, for sufficiently large values of n.

An *upper bound* is the best algorithmic solution that has been found for a problem. " What is the best that we know we can do?"

Exercise: if $f(n) = (3/2)n^2+(5/2)n-3$, then show that $f(n)= O(n^2)$

In simple words, f (n) =O(g(n)) means that the growth rate of f(n) is less than or equal to g(n).

Big-O Theorems

For all the following theorems, assume that f(n) is a function of n and that k is an arbitrary constant.

Theorem 1: k is O(1) big oh of constanit is 1

Theorem 2: A polynomial is O(the term containing the highest power of n).
Polynomial's growth rate is determined by the leading term.
If *f(n)* is a polynomial of degree *d*, then *f(n)* is $O(n^d)$
In general, f(n) is big-O of the dominant term of f(n).

Theorem 3: k*f(n) is O(f(n)). Constant factors may be ignored.
E.g. $f(n) =7n^4+3n^2+5n+1000$ is $O(n^4)$

Theorem 4(Transitivity): If f(n) is O(g(n))and g(n) is O(h(n)), then f(n) is O(h(n)).

Theorem 5: For any base b, $\log_b(n)$ is O(logn). All logarithms grow at the same rate.
$\log_b n$ is $O(\log_d n)$ $\forall b, d > 1$

Theorem 6: Each of the following functions is big-O of its successors:

k
$\log_b n$ k=O(logn)=O(nlogn)=O(n2) jechuudha
n
$n\log_b n$
n^2

n to higher powers

2^n

3^n

larger constants to the nth power

n!

n^n

$f(n)= 3n\log_b n + 4 \log_b n+2$ is $O(n\log_b n)$ and $O(n^2)$ and $O(2^n)$

More Properties of the O Notation

1) Higher powers grow faster
   - n^r is $O(n^s)$ if $0 <= r <= s$
2) Fastest growing term dominates a sum
   - If f(n) is O(g(n)), then f(n) + g(n) is O(g)

   E.g $5n^4 + 6n^3$ is $O(n^4)$
3) Exponential functions grow faster than powers, i.e. n^k is $O(b^n)$ $\forall b > 1$ and $k >= 0$

   E.g. n^{20} is $O(1.05^n)$
4) Logarithms grow more slowly than powers
   - $\log_b n$ is $O(n^k)$ $\forall$ b > 1 and k >= 0

   E.g. $\log_2 n$ is $O(n^{0.5})$

big O=the growth rate of f(n) is less than or equal to g(n)
big omega= the growth rate of f(n)= greater than or equal to g(n)
theta notation=means that f(n) and g(n) have the same rate of growth

1.4.2. Big-Omega Notation

Just as O-notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Formal Definition: A function f(n) is Ω(g (n)) if there exist constants c and k ∈ $\mathcal{R}$+ such that f(n) >=c. g(n) for all n>=k.

f(n)= Ω(g (n)) means that f(n) is greater than or equal to some constant multiple of g(n) for all values of n greater than or equal to some k.

Example*:* If $f(n) =n^{2,}$ then f(n)= Ω(n)

In simple terms, f(n)= Ω(g (n)) means that the growth rate of f(n) is greater than or equal to g(n).

1.4.3. Theta Notation

find avergge case

A function f (n) belongs to the set of Θ (g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1.g(n) and c2.g(n), for sufficiently large values of n.

Formal Definition: A function f (n) is Θ (g(n)) if it is both *O(g(n))* and Ω *(g(n))*. In other words, there exist constants c1, c2, and k >0 such that c1.g (n)<=f(n)<=c2. g(n) for all n >= k

If f(n)= Θ (g(n)), then g(n) is an asymptotically tight bound for f(n).

tightness=mise qabu

In simple terms, f(n)= Θ (g(n)) means that f(n) and g(n) have the same rate of growth.

Example:

1. If f(n)=2n+1, then f(n) = Θ (n)
2. $f(n) =2n^2$ then

   $f(n)=O(n^4)$

   $f(n)=O(n^3)$

   $f(n)=O(n^2)$

You can think of big o as tightness and little o as < when comparing the growth rates of functions.

All these are technically correct, but the last expression is the best and tight one. Since $2n^2$ and n^2 have the same growth rate, it can be written as $f(n)= \Theta(n^2)$.

1.4.4. Little-o Notation

Big-Oh notation may or may not be asymptotically tight, for example:

$2n^2 = O(n^2)$

$= O(n^3)$

f(n)=o(g(n)) means for all c>0 there exists some k>0 such that f(n) < c.g(n) for all n>=k. Informally, f(n)=o(g(n)) means f(n) becomes insignificant relative to g(n) as n approaches infinity.

Example: f(n)=3n+4 is $o(n^2)$

In simple terms, f(n) has less growth rate compared to g(n).

g(n)= $2n^2$ g(n) =$o(n^3)$, $O(n^2)$, g(n) is not $o(n^2)$.

The difference between little o and big o is that little o is a strict upper bound while big o is an inclusive upper bound. This means that if f is in little o of g, then f grows much slower than g as the input size approaches infinity. However, if f is in big o of g, then f grows no faster than g, but it can also grow at the same rate as g.

For example, if f(n) = 2n^2 and g(n) = n^2, then f is in big o of g, because 2n^2 is always less than or equal to some constant times n^2 for large enough n. However, f is not in little o of g, because 2n^2 is not always less than any constant times n^2 for large enough n. In fact, f and g grow at the same rate asymptotically.

On the other hand, if h(n) = n and k(n) = n^2, then h is in both little o and big o of k, because n is always less than some constant times n^2 and also less than any constant times n^2 for large enough n. In this case, h grows much slower than k asymptotically. You can think of big o as <= and little o as < when comparing the growth rates of functions.

1.4.5. Little-Omega (ω notation)

Little-omega (ω) notation is to big-omega (Ω) notation as little-o notation is to Big-Oh notation. We use ω notation to denote a lower bound that is not asymptotically tight.

Formal Definition: f(n)= ω (g(n)) if there exists a constant no>0 such that 0<= c. g(n)<f(n) for all n>=k.

Example: $2n^2=\omega(n)$ but it's not $\omega(n^2)$. wal-qixa ta'uu hin qabu greater than littile-omega() ta'uu qaba

1.5. Relational Properties of the Asymptotic Notations

i. Transitivity

- if f(n)=Θ(g(n)) and g(n)= Θ(h(n)) then f(n)=Θ(h(n)),
- if f(n)=O(g(n)) and g(n)= O(h(n)) then f(n)=O(h(n)),
- if f(n)=Ω(g(n)) and g(n)= Ω(h(n)) then f(n)=Ω (h(n)),
- if f(n)=o(g(n)) and g(n)= o(h(n)) then f(n)=o(h(n)), and
- if f(n)=ω (g(n)) and g(n)= ω(h(n)) then f(n)=ω (h(n)).

ii. Symmetry

- f(n)=Θ(g(n)) if and only if g(n)=Θ(f(n)).

constant isaan bay'isudha garaagarumaa kan uumu male isaan walqixadha

iii. Transpose symmetry

- f(n)=O(g(n)) if and only if g(n)=Ω(f(n)),
- f(n)=o(g(n)) if and only if g(n)=ω(f(n)).

iv. Reflexivity

- f(n)=Θ(f(n)),
- f(n)=O(f(n)),
- f(n)=Ω(f(n)).

less than or equal to waan qabaniif ta'u danda'u mata isanif

CHAPTER TWO
2. TIME COMPLEXITY OF KNOWN ALGORITHMS

2.1. Searching algorithms

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.

Searching is a process of looking for a specific element in a list of items or determining that the item is not in the list. There are two simple searching algorithms:

- Sequential Search, and
- Binary Search

2.1.1. Linear Search (Sequential Search)

The sequential search algorithm begins at the first position in the array and looks at each value in turn until K is found. Once K is found, the algorithm stops. It steps through the data sequentially until a match is found.

It compares the key element with each element in the list A[n], and returns the index i if the key is found, or -1 if the key is not found.

The basic idea is:

- Loop through the array starting at the first element until the value of target matches one of the array elements.
- Return the index if match is found, and if a match is not found, return –1.

Implementation example:

```
int Linear_Search(int list[ ], int key, int n)
{   // n is size of the list
      int index=0;
      int found=0;
      do{
      if(key = = list[index])
                found=1;
      else
                index++;
      } while(found = = 0 && index < n);
      if(found= = 0)
                 index=-1;
      return index;
}
```

There is a wide range of possible running times for the sequential search algorithm. The first integer in the array could have value K, and so only one integer is examined. In this case the running time is short. This is the best case for this algorithm, because it is not possible for sequential search to look at less than one value. Alternatively, if the last position in the array contains K, then the running time is relatively long, because the algorithm must examine N values. This is the worst case for this algorithm, because sequential search never looks at more than N values. If we implement sequential search as a program and run it many times on many different arrays of size N, or search for many different values of K within the same array, we expect the algorithm on average to go halfway through the array before finding the value we seek. On average, the algorithm examines about N/2 values.

The complexity of the algorithm is analyzed in three ways. First, we provide the cost of an unsuccessful search. Then, we give the worst-case cost of a successful search. Finally, we find the average cost of a successful search. Analyzing successful and unsuccessful searches separately is typical. Unsuccessful searches usually are more time consuming than are successful searches.

An unsuccessful search requires the examination of every item in the array, so the time will be O(N). In the worst case, a successful search, too, requires the examination of every item in the array because we might not find a match until the last item. Thus the worst-case running time for a successful search is also linear. On average, however, we search only half an array. That is, for every successful search in position i, there is a corresponding successful search in position N-1- i (assuming we start numbering from 0). However, N/2 is still O(N).

2.1.2. Binary Search

This searching algorithms works only on an ordered list. Suppose you have a sorted array, and you wish to search it for a particular value, *x*. The key idea is that since the array is sorted, you can narrow your focus to a section of the array that must contain *x* if *x* is in the array at all. In particular, *x* must come before every element greater than *x* and after every element less than *x*. We call the section of the array that could contain *x* the *section of interest*. We can iteratively home in on *x* with a loop that shrinks the section of interest while maintaining the loop invariant that *x* is in that section, if it is in the array at all.

Binary search is performed from the middle of the array rather than the end. We keep track of *low* and *high*, which delimit the portion of the array in which an item, if present, must reside. Initially, the range is from *0* to *N - 1*. If *low* is larger than *high*, we know that the item is not present, so we return *NOT-FOUND*. Otherwise, we let *mid* be the halfway point of the range (rounding down if the range has an even number of elements) and compare the item we are searching for with the item in position *mid*. If we find a match, we are done and can return. If the item we are searching for is less than the item in position *mid*, then it must reside in the range *low* to *mid-1*. If it is greater, then it must reside in the range *mid+l* to *high*.

The basic idea is:

- Locate midpoint of array to search
- Determine if target is in lower half or upper half of an array.
    - If in lower half, make this half the array to search
    - If in the upper half, make this half the array to search
- Loop back to step 1 until the size of the array to search is one, and this element does not match, in which case return –1.

Implementation example:

```
int Binary_Search(int list[ ], int key, int n)
{ //n is size of the list
  int low = 0;
  int high = n-1;
  int found = 0;
  do{
     mid = (low + high)/2;
     if(key = = list[mid])
        found=1;
    else{
          if(key < list[mid])
                high = mid-1;
          else
                low = mid+1;
          }
     }while(found = = 0 && low <= high);
    if(found = = 0)
     index = -1;
    else
     index = mid;
  return index;
  }
```

For an unsuccessful search, the number of iterations in the loop is *logN + 1*. The reason is that we halve the range in each iteration (rounding down if the range has an odd number of elements); we add 1 because the final range encompasses zero elements. For a successful search, the worst case is log N iterations because in the worst case we get down to a range of only one element. The average case is only one iteration better because half the elements require the worst case for their search, a quarter of the elements save one iteration, and only one in 2^i elements save *i* iterations from the worst case. The mathematics involves computing the weighted average by calculating the sum of a finite series. The end result, however, is that the running time for each search is O(log N).

Initially, the number of candidate entries is n; after the first call to Binary_Search, it is at most $n/2$; after the second call, it is at most $n/4$; and so on. In general, after the ith call to Binary_Search, the number of candidate entries remaining is at most $n/2^i$. In the worst case (unsuccessful search), the recursive calls stop when there are no more candidate entries. Hence, the maximum number of recursive calls performed, is the smallest integer m such that $n/2^m < 1$. In other words (recalling that we omit a logarithm's base when it is 2), $m > \log n$. Thus, we have $m = \log n + 1$, which implies that binary search runs in $O(\log n)$ time.

As an example, if we have 128 entries, then the remaining candidate entries after each iteration are 64, 32, 16, 8, 4, 2, 1, 0. Thus, the running time is O(logN).

2.2. Sorting Algorithms

Sorting is one of the most important operations performed by computers. Sorting is a process of reordering a list of items in either increasing or decreasing order. The following are simple sorting algorithms used to sort small-sized lists.

- Insertion Sort
- Selection Sort
- Bubble Sort

2.2.1. Insertion Sort

An insertion sort starts by considering the two first elements of the array data, which are data[0] and data[1]. If they are out of order, an interchange takes place. Then, the third element, data[2], is considered. If data[2] is less than data[0] and data[1], these two elements are shifted by one position; data[0] is placed at position 1, data[1] at position 2, and data[2] at position 0. If data[2] is less than data[1] and not less than data[0], then only data[1] is moved to position 2 and its place is taken by data[2]. If, finally, data[2] is not less than both its predecessors, it stays in its current position. Each element data[i] is inserted into its proper location j such that $0 \leq j \leq i$, and all elements greater than data[i] are moved by one position.

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

It's the most instinctive type of sorting algorithm. The approach is the same approach that you use for sorting a set of cards in your hand. While playing cards, you pick up a card, start at the beginning of your hand and find the place to insert the new card, insert it and move all the others up one place.

The basic idea:

Find the location for an element and move all others up, and insert the element.

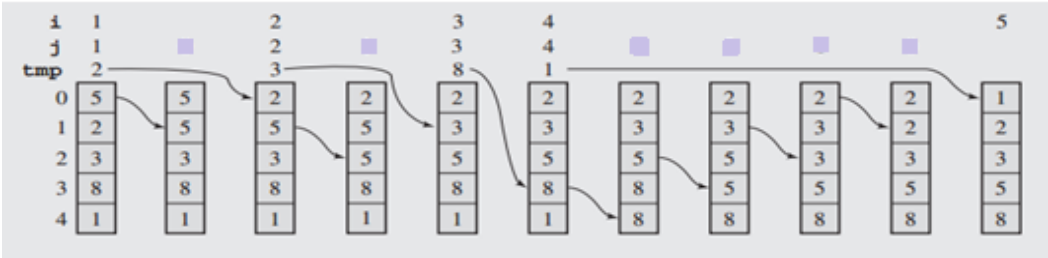
The process involved in insertion sort is as follows:

- The left most value can be said to be sorted relative to itself. Thus, we don't need to do anything.
- Check to see if the second value is smaller than the first one. If it is, swap these two values. The first two values are now relatively sorted.
- Next, we need to insert the third value in to the relatively sorted portion so that after insertion, the portion will still be relatively sorted.
- Remove the third value first. Slide the second value to make room for insertion. Insert the value in the appropriate position.
- Now the first three are relatively sorted.
- Do the same for the remaining items in the list.

Implementation example:

```
void Insertion_Sort(int list[ ], int n )
{
  int i, j, temp;
 for( i = 1; i < n; i++)
    {
      temp=list[i];
       for( j=i; j>0 && temp<list[j-1];j--)
            { // work backwards through the array finding where temp should go
                  list[j]=list[j-1];
             }//end of inner loop
         list[j]=temp;
       }//end of outer loop
}//end of Insertion_Sort
```

The following figure shows how the array [5 2 3 8 1] is sorted by insertion sort.



Because an array having only one element is already ordered, the algorithm starts sorting from the second position, position *1*. Then for each element *temp = data[i]*, all elements greater than *temp* are copied to the next position, and *temp* is put in its proper place.

To find the number of comparisons performed by insertion sort algorithm, observe first that the outer for loop always performs n-1 iterations. The best case is when the data are already in order. Only one comparison is made for each position i, so there are n - 1 comparisons, which is O(n), , and 2(n - 1) moves, all of them redundant.

The worst case is when the data are in reverse order. In this case, for each i, the item data[i] is less than every item data[0], . . . , data[i-1], and each of them is moved by one position. For each iteration i of the outer for loop, there are i comparisons, and the total number of comparisons for all iterations of this loop is

$$\sum_{i=1}^{n-1}(i) = 1 + 2 + 3 + \ldots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

The number of times the assignment in the inner for loop is executed can be computed using the same formula. The number of times temp is loaded and unloaded in the outer for loop is added to that, resulting in the total number of moves

$$\frac{n(n-1)}{2} + 2\,(n - 1) = \frac{n^2+3n-4}{2} = O(n^2)$$

2.2.2. Selection Sort

Selection sort is an attempt to localize the exchanges of array elements by finding a misplaced element first and putting it in its final place. The element with the lowest value is selected and exchanged with the element in the first position. Then, the smallest value among the remaining elements data[1], . . . , data[n-1] is found and put in the second position. This selection and placement by finding, in each pass i, the lowest value among the elements data[i], . . . , data[n-1] and swapping it with data[i] are continued until all elements are in their proper positions.

The following pseudo code reflects the simplicity of the algorithm:

```
selectionsort(data[ ] , n)
  for i = 0 to n-2
    select the smallest element among data[i], . . . , data[n-1];
    swap it with data[i];
```

It is rather obvious that n-2 should be the last value for i, because if all elements but the last have been already considered and placed in their proper positions, then the nth element (occupying position n-1) has to be the largest.

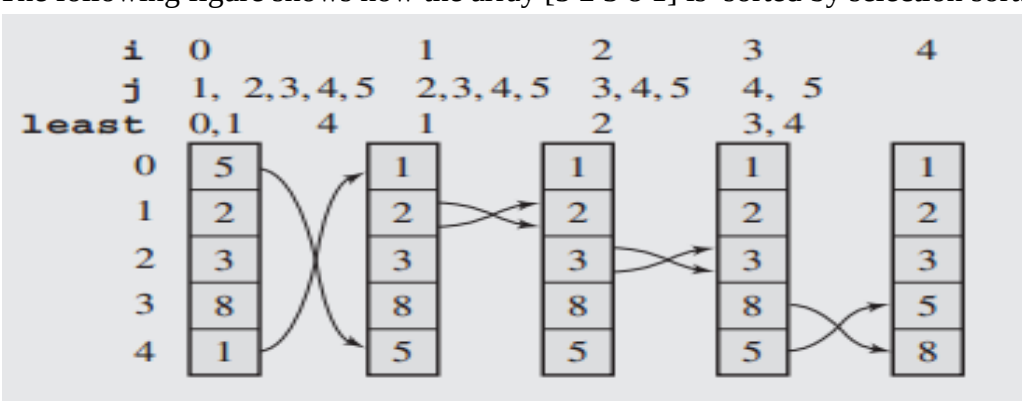
The basic idea:

- Loop through the array from i=0 to n-1.
- Select the smallest element in the array from i to n
- Swap this value with value at position i.

Implementation example:

```
void Selection_Sort(int data[ ], int n)
{
   int i,j, least,temp;
  for( i = 0; i<n-1; i++ )
   {
     least = i;
     for( j = i+1; j<n; j++ )
       {
        if(data [j] < data [least])
            least = j;
      }//end of inner loop
      temp = data [least];
      data [least] = data [i];
      data [i] = temp;
   } //end of outer loop
}//end of Selection_Sort
```

The following figure shows how the array [5 2 3 8 1] is sorted by selection sort.



Note: *least* in the implementation and figure above is not the smallest element but its position.

The analysis of the performance of the function Selection_Sort ( ) is simplified by the presence of two for loops with lower and upper bounds. The outer loop executes *n - 1* times, and for each *i* between *0* and *n - 2*, the inner loop iterates *j = (n -1) - i* times. Because comparisons of keys are done in the inner loop, there are

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1 = \frac{n(n-1)}{2} = O(n^2)$$

comparisons. This number stays the same for all cases. There can be some savings only in the number of swaps. Note that if the assignment in the if statement is executed, only the index j is moved, not the item located currently at position j. Array elements are swapped unconditionally in the outer loop as many times as this loop executes, which is **n-1**. Thus, in all cases, items are moved the same number of times, **3(n - 1)**, i.e **O(n)**.

2.2.3. Bubble Sort

A bubble sort can be best understood if the array to be sorted is envisaged as a vertical column whose smallest elements are at the top and whose largest elements are at the bottom. The array is scanned from the bottom up, and two adjacent elements are interchanged if they are found to be out of order with respect to each other. First, items data[n-1] and data[n-2] are compared and swapped if they are out of order. Next, data[n-2] and data[n-3] are compared, and their order is changed if necessary, and so on up to data[1] and data[0]. In this way, the smallest element is bubbled up to the top of the array.

However, this is only the first pass through the array. The array is scanned again comparing consecutive items and interchanging them when needed, but this time, the last comparison is done for data[2] and data[1] because the smallest element is already in its proper position, namely, position 0. The second pass bubbles the second smallest element of the array up to the second position, position 1. The procedure continues until the last pass when only one comparison, data[n-1] with data[n-2], and possibly one interchange are performed.

A pseudo code of this algorithm is as follows:

```
bubblesort(data[ ] , n)
 for i = 0 to n - 2
   for j = n-1down to i+1
    swap elements in positions j and j-1 if they are out of order;
```

Bubble sort is the simplest algorithm to implement and the slowest algorithm on very large inputs.

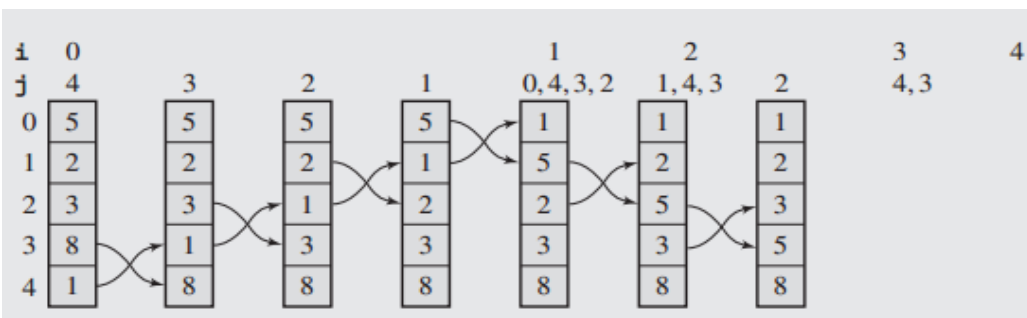
The basic idea of the algorithm is:

- Loop through array from i=0 to n and swap adjacent elements if they are out of order.

Implementation example:

```
void Bubble_Sort( int data[ ], int n)
{
   int i, j, temp;
   for( i = 0; i < n-1; i++)
     {
       for( j = n-1; j > i; j--)
         {
             if(data [ j ] < data [ j-1 ] )
                 {
                   temp = data [ j ];
                   data [ j ] = data [ j-1 ];
                   data [ j-1 ] = temp;
                 }//swap adjacent elements
           }//end of inner loop
     }//end of outer loop
}//end of Bubble_Sort
```

The following figure shows how the array [5 2 3 8 1] is sorted by bubble sort.



The number of comparisons is the same in each case (best, average, and worst) and equals the total number of iterations of the inner for loop

$$\sum_{i=2}^{n-2}(n-1-i) = \frac{n(n-1)}{2} = O(n^2)$$

comparisons. This formula also computes the number of swaps in the worst case when the array is in reverse order. In this case, $3\frac{n(n-1)}{2} = O(n^2)$ moves have to be made. The best case, when all elements are already ordered, requires no swaps.

1. Bubble Sort:
   - Worst Case: O(n^2)
   - Average Case: O(n^2)
   - Best Case: O(n)
2. Selection Sort:
   - Worst Case: O(n^2)
   - Average Case: O(n^2)
   - Best Case: O(n^2)
3. Insertion Sort:
   - Worst Case: O(n^2)
   - Average Case: O(n^2)
   - Best Case: O(n)
4. Merge Sort:
   - Worst Case: O(n log n)
   - Average Case: O(n log n)
   - Best Case: O(n log n)

Here, "n" denotes the number of elements
in the input array, and "k" represents the number of digits in the largest element.

5. Quick Sort:
   - Worst Case: O(n^2)
   - Average Case: O(n log n)
   - Best Case: O(n log n)
6. Heap Sort:
   - Worst Case: O(n log n)
   - Average Case: O(n log n)
   - Best Case: O(n log n)

7. Radix Sort:
   - Worst Case: O(nk)
   - Average Case: O(nk)
   - Best Case: O(nk)

There are various searching algorithms with different worst, average, and best-case complexities. Here are some common searching algorithms:

1. Linear Search:
   - Worst-case complexity: O(n)
   - Average-case complexity: O(n/2)
   - Best-case complexity: O(1)
2. Binary Search (applicable for sorted lists):
   - Worst-case complexity: O(log n)
   - Average-case complexity: O(log n)
   - Best-case complexity: O(1)
3. Hashing (using hash tables):
   - Worst-case complexity: O(n) (in case of collisions, leading to linear search within each bucket)
   - Average-case complexity: O(1) (assuming a good hash function and no or minimal collisions)
   - Best-case complexity: O(1)
4. Interpolation Search (applicable for uniformly distributed sorted lists):
   - Worst-case complexity: O(n) (in case of uniformly distributed elements)
   - Average-case complexity: O(log log n) (if the elements are uniformly distributed)
   - Best-case complexity: O(1) (when the desired element is found at the first comparison)
5. Ternary Search (applicable for sorted lists):
   - Worst-case complexity: O(log3 n)
   - Average-case complexity: O(log3 n)
   - Best-case complexity: O(1) (when the desired element is found at the first comparison)

CHAPTER THREE
LINKED LISTS

Unlike arrays, linked lists do not require contiguous memory allocation, allowing for efficient dynamic memory management.

3.1. Structures

In many occasions what we want to store are not mere sequences of elements of the same data type (as in arrays), but sets of different elements with different data types. Structures offer a way of joining these heterogeneous elements together to form complex structures.

A structure is aggregate data types built using elements of primitive data types. It is a group of data elements grouped together under one name. These data elements known as *members* (*also called fields*) can have different types, some can be *int*, some can be *float*, some can be *char* and so on, and different lengths.

The difference between array and structure is the element of an array has the same type while the elements of structure can be of different type. Another difference is that each element of an array is referred by its position while each element of structure has a unique name.

Declaration of structures

Structures are declared in C++ using the following syntax:

```
struct   structure_name
{
Member_type 1        member_name1;
Member_type 2        member_name2;
Member_type 3        member_name3;
.
.
} object_names;
```

Where *struct* is a keyword, *structure_name* is a valid identifier name for the structure type; object_name can be a set of valid identifiers for objects that have the type of this structure. Within braces { } there is a list with the data members, each one is specified with a data type and a valid identifier as its name.

For example:

```
struct student
 {
  char name [20];
  char Idd [10];
  char Department [25];
  float GPA;
 };
```

Creating structure variables

The structure variables are created in different ways.

1. We can create structure variables at the time of declaration

Example:

```
struct student
{
 char name [20];
 char Idd [10];
 char Department [25];
 int age;
} stud1, stud2, stud3;
```

2. We can create structure variables in such a way

```
struct student
{
 char name [20];
 char Idd [10];
 char Department [25];
 int age;
} ;
struct student stud1, stud2, stud3;
```

3. We can also create structure variables as follows

```
struct student
{
 char name [20];
 char Idd [10];
 char Department [25];
 int age;
} ;

student stud1, stud2, stud3;
```

Accessing Members of Structures

We use the *dot operator* (.) to access members of a structure. We use this format to *read (accept)*, to *write* (*display)* or *assign* data to members of a structure. We can assign value to the name *Stud1* as follows:

Stud1.name = "Abrham"; i.e.
Structure_variable_name . member_name;

We use also the *Arrow operator (->)* to access data members of pointer variables pointing to the structure. If we have, for example, a pointer variable for *student* structure created above like *struct student *studptr;* ; then we can set value for *name* member of the structure for this variable using *studptr -> name = "Abrham";*. Note that *studptr->name* is the same as (**studptr).name*.

Self-Referential Structures

Structures can hold pointers to instances of themselves, and such structures are called self-referential structures. Example of self-referential structure is the following.

```
struct student{
        char name[10];
        float GPA;
        student *next;
      };
```

For printing member of GPA of next variable is done in the following manner.
cout<<next->GPA; or cout<<(*next).GPA;

3.2. Singly linked lists

3.2.1. Introduction

Linked lists are the most basic self-referential structures. Linked lists allow you to have a chain of structs with related data. A linked list is made up of a series of objects, called the nodes of the list that are connected by links. Because a list node is a distinct object (as opposed to simply a cell in an array), it is good practice to make a separate list node class. An additional benefit to creating a list node class is that it can be reused by the linked implementations for the stack and queue data structures presented later in the future chapters. Objects in the link class contain an element field to store the element value, and a next field to store a pointer to the next node on the list. The list built from such nodes is called a singly linked list, or a one-way list, because each list node has a single pointer to the next node on the list, i.e. each node in a singly linked list has a link only to its successor in the sequence.

The most flexible way of implementing linked structures is pointer.

A linked list is made up of a chain of nodes. Each node contains:

- the data item, and
- a pointer to the next node

A linked list, in its simplest form, is a collection of nodes that together form a linear ordering. As in the children's game "Follow the Leader," each node stores a pointer, called next, to the next node of the list. In addition, each node stores its associated element. See the following figure.

Figure 1: Example of a singly linked list of airport codes. The next pointers are shown as arrows. The null pointer is denoted by ∅.

The next pointer inside a node is a link or pointer to the next node of the list. Moving from one node to another by following a next reference is known as link hopping or pointer hopping. The first and last nodes of a linked list are called the *head* and *tail* of the list, respectively. Thus, we can link-hop through the list, starting at the head and ending at the *tail*. We can identify the tail as the node having a null next reference.

Array vs Linked lists

Arrays are *simple* and *fast but we m*ust specify their size at construction time. This has its own drawbacks. If you construct an array with space for *n*, tomorrow you may need *n*+1. Here comes a need for a more flexible system.

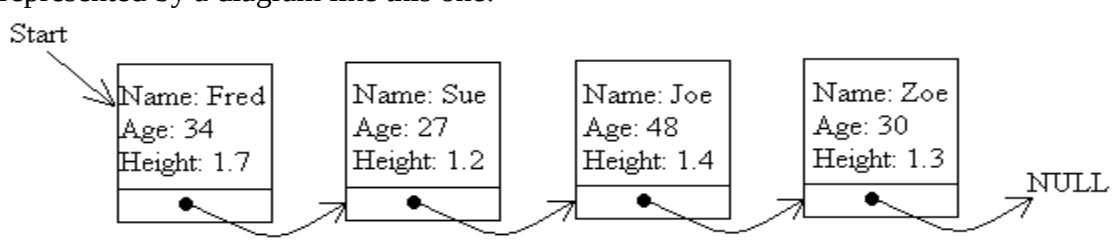
Arrays are nice and simple for storing things in a certain order. However, they are not very adaptable. For instance, we have to fix the size n of an array in advance, which makes resizing an array difficult. Insertions and deletions are difficult because elements need to be shifted around to make space for insertion or to fill empty positions after deletion.

Array has at least two limitations: (1) its size has to be known at compilation time and (2) the data in the array are separated in computer memory by the same distance, which means that inserting an item inside the array requires shifting other data in this array. This limitation can be overcome by using linked structures. Flexible space use by dynamically allocating space for each element as needed. This implies that one need not know the size of the list in advance. Memory is efficiently utilized.

Like an array, a singly linked list maintains its elements in a certain order, as determined by the chain of next links. Unlike an array, a singly linked list does not have a predetermined fixed size. It can be resized by adding or removing nodes.

3.2.2. Creating Linked Lists in C++

A linked list is a data structure that is built from structures and pointers. It forms a chain of "nodes" with pointers representing the links of the chain and holding the entire thing together. A linked list can be represented by a diagram like this one:



This linked list has four nodes in it, each with a link to the next node in the series. The last node has a link to the special value ***NULL***, which any pointer (whatever its type) can point to, to show that it is the last link in the chain. There is also another special pointer, called ***Start*** (also called ***Head***), which points to the first link in the chain so that we can keep track of it. It is important to note that ***head*** is not a node, rather the address of the first node of the list.

The key part of a linked list is a structure, which holds the data for each node (the name, address, age or whatever for the items in the list), and, most importantly, a pointer to the next node. Here we have given the structure of a typical node:

```
struct node
{
  char name[20];    // Name of up to 20 letters
  int age
  float height;     // In metres
  node *next;// Pointer to next node
 };
struct node *start_ptr = NULL;
```

The important part of the structure is the line before the closing curly brackets. This gives a pointer to the next node in the list. This is the only case in C++ where you are allowed to refer to a data type (in this case **node**) before you have even finished defining it!

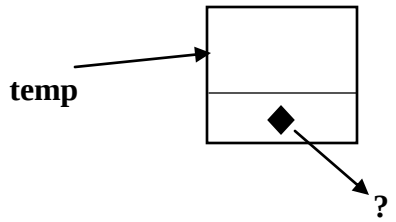
We have also declared a pointer called **start_ptr** that will permanently point to the start of the list. To start with, there are no nodes in the list, which is why **start_ptr** is set to *NULL*.

3.2.3. Adding a node to the list

The first problem that we face is how to add a node to the list. For simplicity's sake, we will assume that it has to be added to the end of the list, although it could be added anywhere in the list (a problem we will deal with later on).

Firstly, we declare the space for a pointer item and assign a temporary pointer to it. This is done using the **new** statement as follows:

```
temp = new node;
```



We can refer to the new node as ***temp**, i.e. **"the node that temp points to"**. When the fields of this structure are referred to, brackets can be put round the ***temp** part, as otherwise the compiler will think we are trying to refer to the fields of the pointer. Alternatively, we can use the arrow pointer notation. That's what we shall do here.

Having declared the node, we ask the user to fill in the details of the person, i.e. the name, age, address or whatever:

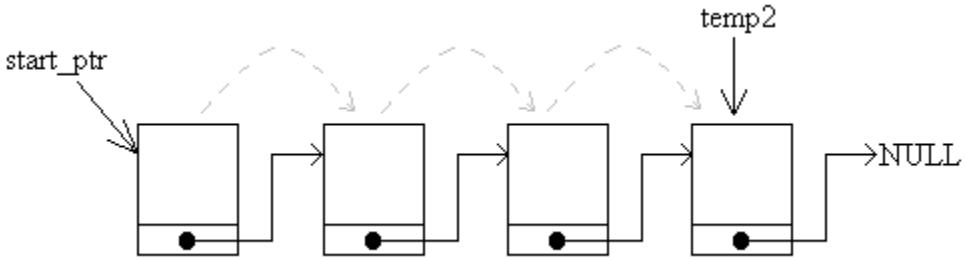
```
cout<< "Please enter the name of the person: ";
cin>> temp->name;
cout<< "Please enter the age of the person : ";
cin>> temp->age;
cout<< "Please enter the height of the person : ";
cin>> temp->height;

temp->next = NULL;
```

The last line sets the pointer from this node to the next to *NULL*, indicating that this node, when it is inserted in the list, will be the last node. Having set up the information, we have to decide what to do with the pointers. Of course, if the list is empty to start with, there's no problem - just set the Start pointer to point to this node (i.e. set it to the same value as temp):

```
if (start_ptr == NULL)
start_ptr = temp;
```

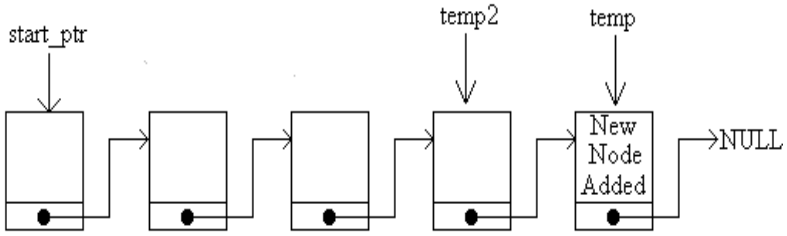
It is harder if there are already nodes in the list. In this case, the secret is to declare a second pointer, **temp2**, to step through the list until it finds the last node.




```
temp2 = start_ptr;
  // We know this is not NULL - list not empty!
while (temp2->next != NULL){
   temp2 = temp2->next;   // Move to next link in chain
  }
```

The loop will terminate when **temp2** points to the last node in the chain, and it knows when this happened because the **next** pointer in that node will point to *NULL*. When it has found it, it sets the pointer from that last node to point to the node we have just declared:

```
temp2->next = temp;
```



The link **temp2->next** in this diagram is the link joining the last two nodes. The full code for adding a node at the end of the list is shown below, in its own little function:

```
void add_node_at_end ( )
{
   node *temp, *temp2;   // Temporary pointers
    // Reserve space for new node and fill it with data
   temp = new node;
   cout<< "Please enter the name of the person: ";
   cin>> temp->name;
   cout<< "Please enter the age of the person : ";
   cin>> temp->age;
   cout<< "Please enter the height of the person : ";
   cin>> temp->height;
   temp->next = NULL;
    // Set up link to this node
     if (start_ptr = = NULL)
       start_ptr = temp;
     else
      {
         temp2 = start_ptr;
         // We know this is not NULL - list not empty!
        while (temp2->next != NULL)
          {
           temp2 = temp2->next; // Move to next link in chain
           }
         temp2->next = temp;
      }
 }
```

3.2.4. Displaying the list of nodes

Having added one or more nodes, we need to display the list of nodes on the screen. This is comparatively easy to do. Here is the method:

1. Set a temporary pointer to point to the same thing as the start pointer.
2. If the pointer points to *NULL*, display the message "End of list" and stop.
3. Otherwise, display the details of the node pointed to by the start pointer.
4. Make the temporary pointer point to the same thing as the **next** pointer of the node it is currently indicating.
5. Jump back to step 2.

The temporary pointer moves along the list, displaying the details of the nodes it comes across. At each stage, it can get hold of the next node in the list by using the **next** pointer of the node it is currently pointing to. Here is the C++ code that does the job:

```
temp = start_ptr;
 do
  {
    if (temp = = NULL)
      cout << "End of list" << endl;
    else
     {
       // Display details for what temp points to
        cout << "Name : " << temp->name << endl;
        cout << "Age : " << temp->age << endl;
        cout << "Height : " << temp->height << endl;
         // Move to next node (if present)
         temp = temp->next;
       }
   } while (temp != NULL);
```

Check through this code, matching it to the method listed above. It helps if you draw a diagram on paper of a linked list and work through the code using the diagram.

3.2.5. Navigating through the list

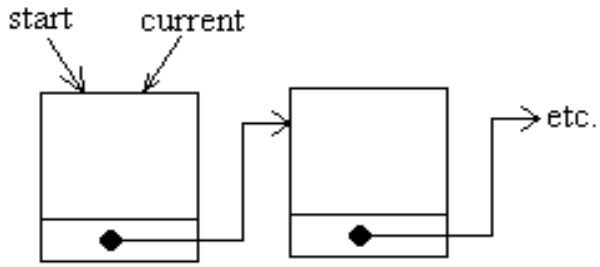
One thing you may need to do is to navigate through the list, with a pointer that moves backwards and forwards through the list, like an index pointer in an array. This is certainly necessary when you want to insert or delete a node from somewhere inside the list, as you will need to specify the position.

We will call the mobile pointer **current**. First of all, it is declared, and set to the same value as the **start_ptr** pointer:

```
node *current;

current = start_ptr;
```

Notice that you don't need to set *current* equal to the address of the *start* pointer, as they are both pointers. The statement above makes them both points to the same thing:



It's easy to get the current pointer to point to the next node in the list (i.e. move from left to right along

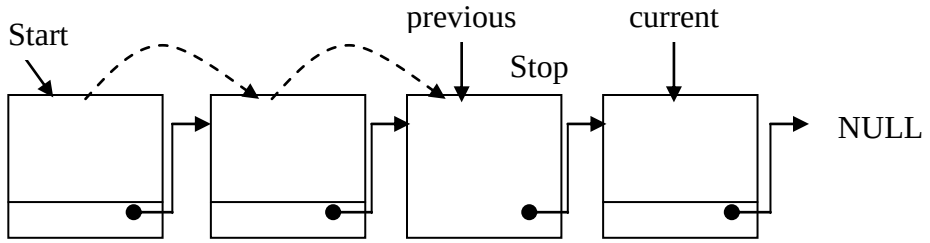
the list). If you want to move current along one node, use the next field of the node that it is pointing to at the moment:

```
current = current->next;
```

In fact, we had better check that it isn't pointing to the last item in the list. If it is, then there is no next node to move to:

```
if (current->next = = NULL)
    cout << "You are at the end of the list." << endl;
else
    current = current->next;
```

Moving the current pointer back one step is a little harder. This is because we have no way of moving back a step automatically from the current node. The only way to find the node before the current one is to start at the beginning, work our way through and stop when we find the node before the one we are considering at the moment. We can tell when this happens, as the next pointer from that node will point to exactly the same place in memory as the current pointer (i.e. the current node).



First of all, we had better check to see if the current node is also the first one. If it is, then there is no "previous" node to point to. If not, check through all the nodes in turn until we detect that we are just behind the current one.

```
if (current = = start_ptr)
    cout << "You are at the start of the list" << endl;
else
  {
   node *previous;     // Declare the pointer
    previous = start_ptr;

    while (previous->next != current)
       {
          previous = previous->next;
       }
    current = previous;
  }
```

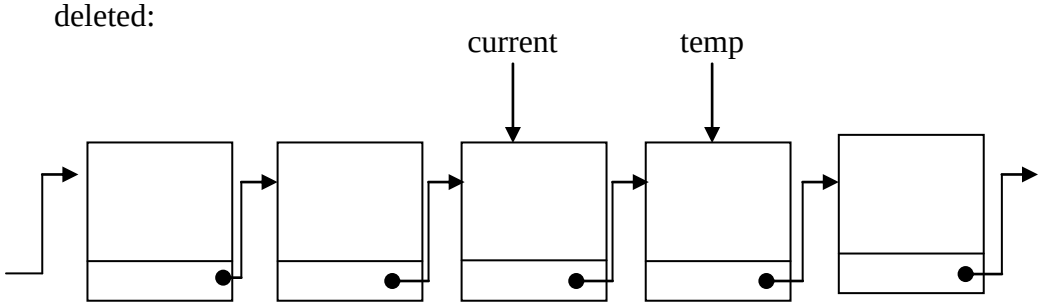
The else clause translates as follows: Declare a temporary pointer (for use in this else clause only). Set it equal to the start pointer. All the time that it is not pointing to the node before the current node, move it along the line. Once the previous node has been found, the current pointer is set to that node - i.e. it moves back along the list.

Now that you have the facility to move back and forth, you need to do something with it. Firstly, let's see if we can alter the details for that particular node in the list:

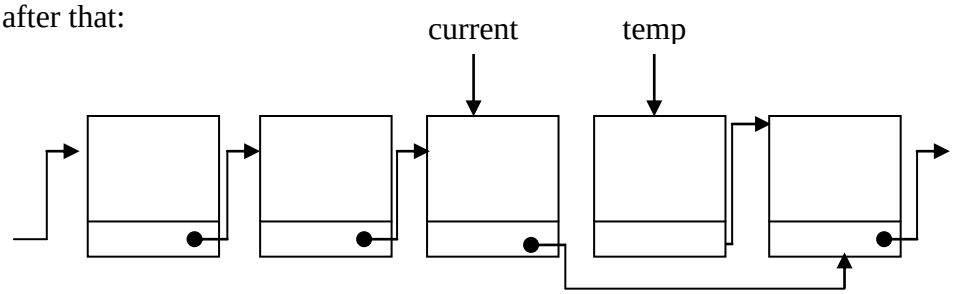
```
cout << "Please enter the new name of the person: ";
cin >> current->name;
cout << "Please enter the new age of the person : ";
cin >> current->age;
cout << "Please enter the new height of the person : ";
cin >> current->height;
```

The next easiest thing to do is to ***delete*** a node from the list directly after the current position. We have to use a temporary pointer to point to the node to be deleted. Once this node has been "anchored", the pointers to the remaining nodes can be readjusted before the node on death row is deleted. Here is the sequence of actions:

1. First, the temporary pointer is assigned to the node after the current one. This is the node to be deleted:



2. Now the pointer from the current node is made to leap-frog the next node and point to the one after that:



3. The last step is to delete the node pointed to by **temp**.

Here is the code for deleting the node. It includes a test at the start to test whether the current node is the last one in the list:

```
if (current->next = = NULL)
    cout << "There is no node after current" << endl;
else
  {
    node *temp;
    temp = current->next;
    current->next = temp->next;     // Could be NULL
    delete temp;
  }
```

Here is the code to ***add*** a node after the current one. This is done similarly, but we haven't illustrated it with diagrams:

```
if (current->next  = = NULL)
    add_node_at_end();
else
    {
      node *temp;
      new temp;
      get_details(temp);
      // Make the new node point to the same thing as the current node
      temp->next = current->next;
      // Make the current node point to the new link in the chain
      current->next = temp;
        }
```

We have assumed that the function **add_node_at_end()** is the routine for adding the node to the end of the list that we created near the top of this section. This routine is called if the current pointer is the last one in the list so the new one would be added on to the end.

Similarly, the routine **get_temp(temp)** is a routine that reads in the details for the new node similar to the one defined just above.

... and so ...

3.2.6. Deleting a node from the list

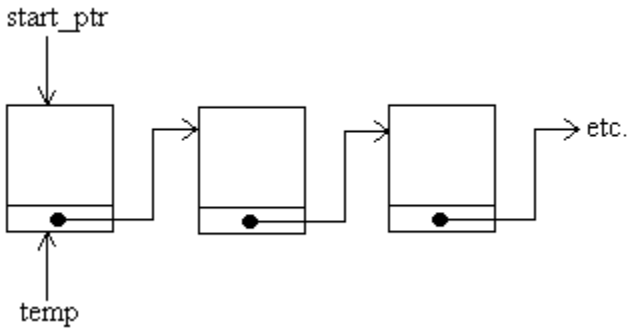
When it comes to deleting nodes, we have three choices: Delete a node from the start of the list, delete one from the end of the list, or delete one from somewhere in the middle. For simplicity, we shall just deal with deleting one from the start or from the end.

When a node is deleted, the space that it took up should be reclaimed. Otherwise the computer will eventually run out of memory space. This is done with the **delete** instruction:

```
delete temp;     // Release the memory pointed to by temp
```

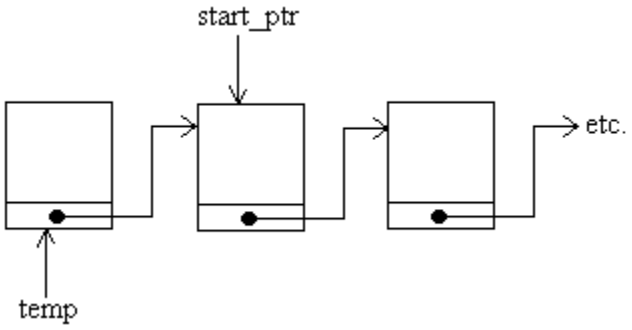
However, we can't just delete the nodes willy-nilly as it would break the chain. We need to reassign the pointers and then delete the node at the last moment. Here is how we go about deleting the first node in the linked list:

temp = start_ptr; // Make the temporary pointer identical to the start pointer

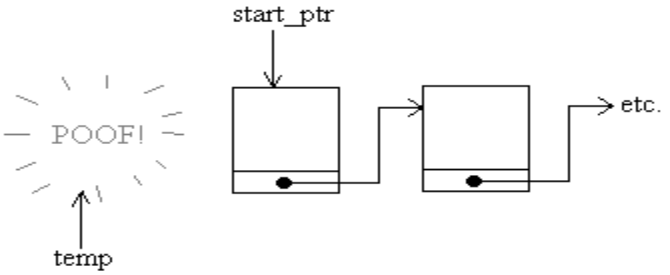


Now that the first node has been safely tagged (so that we can refer to it even when the start pointer has been reassigned), we can move the start pointer to the next node in the chain:

start_ptr = start_ptr->next; // Second node in chain.



delete temp; // Wipe out original start node



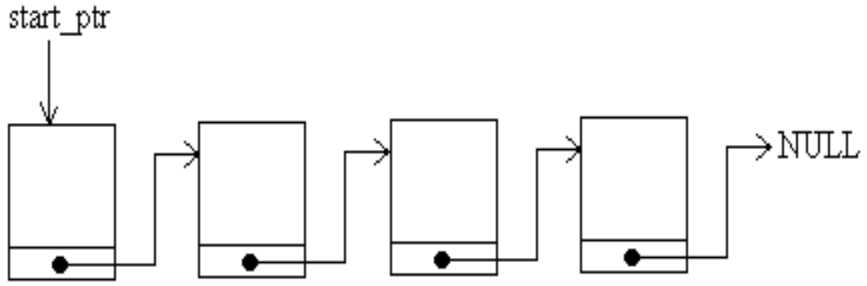
Here is the function that deletes a node from the start:

```
void delete_start_node()
  {
    node *temp;
    temp = start_ptr;
    start_ptr = start_ptr->next;
    delete temp;
  }
```

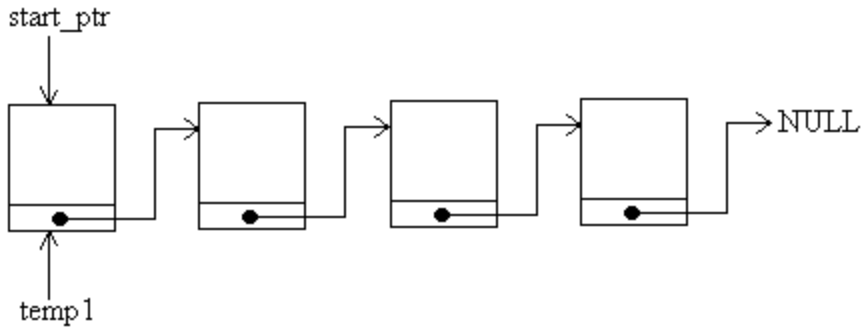
Deleting a node from the end of the list is harder, as the temporary pointer must find where the end of the list is by hopping along from the start. This is done using code that is almost identical to that used to insert a node at the end of the list. It is necessary to maintain two temporary pointers, **temp1** and **temp2**. The pointer **temp1** will point to the last node in the list and **temp2** will point to the previous node. We have to keep track of both as it is necessary to delete the last node and immediately afterwards, to set the **next** pointer of the previous node to **NULL** (it is now the new last node).

1. Look at the start pointer. If it is NULL, then the list is empty, so print out a "No nodes to delete" message.
2. Make **temp1** point to whatever the start pointer is pointing to.
3. If the **next** pointer of what **temp1** indicates is NULL, then we've found the last node of the list, so jump to step 7.
4. Make another pointer, **temp2**, point to the current node in the list.
5. Make **temp1** point to the next item in the list.
6. Go to step 3.
7. If you get this far, then the temporary pointer, **temp1**, should point to the last item in the list and the other temporary pointer, **temp2**, should point to the last-but-one item.
8. Delete the node pointed to by **temp1**.
9. Mark the **next** pointer of the node pointed to by **temp2** as NULL - it is the new last node.

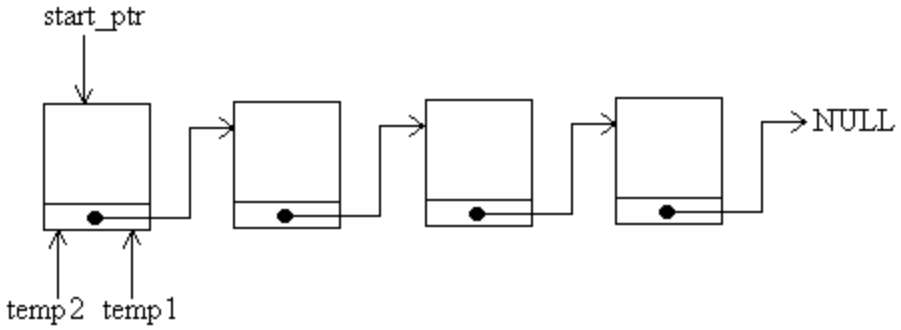
Let's try it with a rough drawing. This is always a good idea when you are trying to understand an abstract data type. Suppose we want to delete the last node from this list:



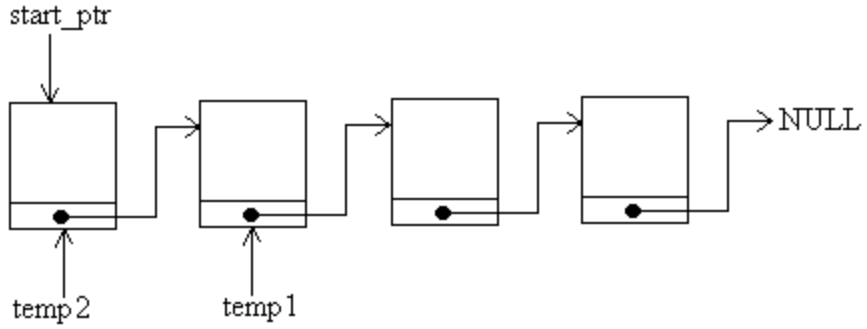
First, the start pointer doesn't point to NULL, so we don't have to display " No nodes to delete " message. Let's get straight on with step2 - set the pointer **temp1** to the same as the start pointer:



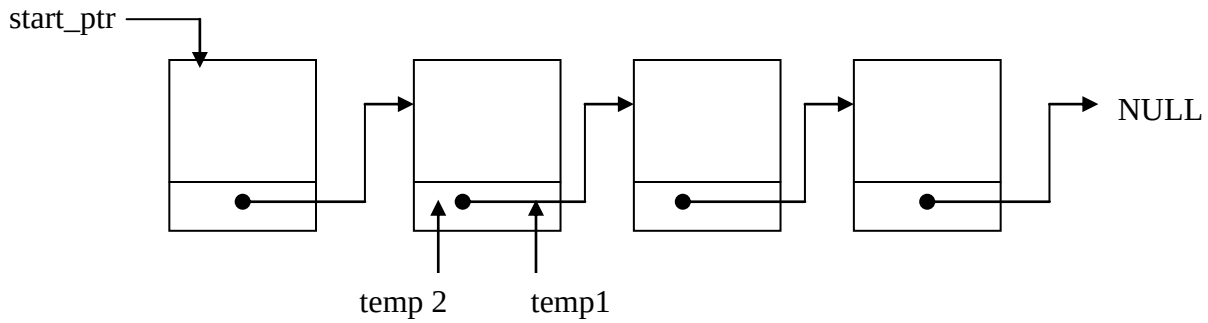
The **next** pointer from this node isn't NULL, so we haven't found the end node. Instead, we set the pointer **temp2** to the same node as **temp1** as shown below.



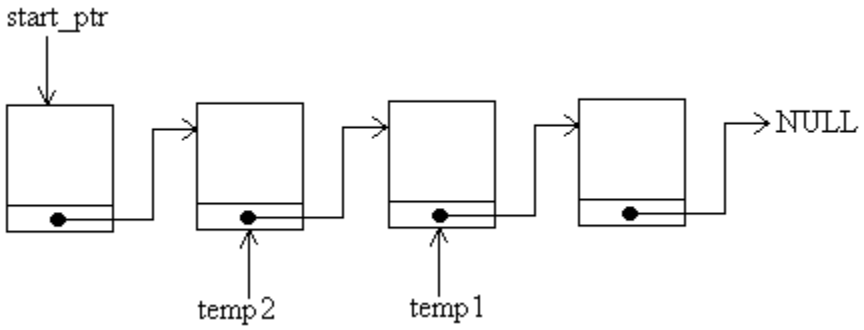
and then move temp1 to the next node in the list:



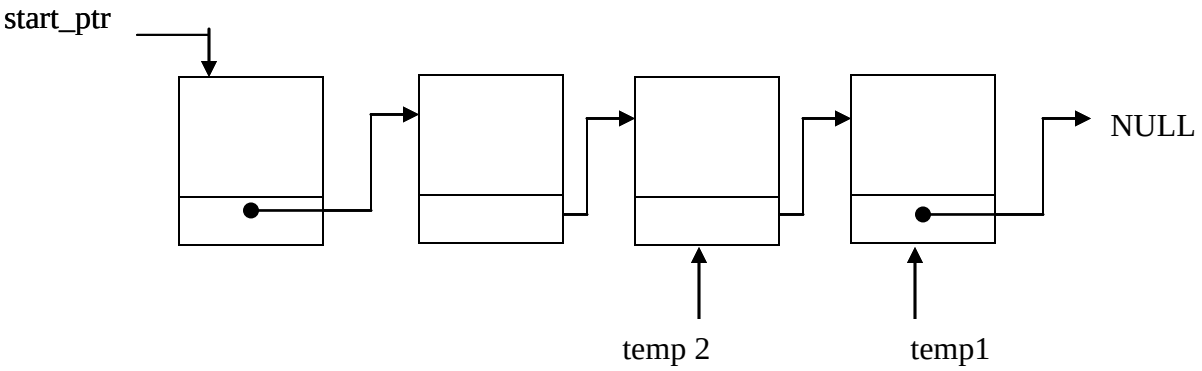
Going back to step 3, we see that **temp1** still doesn't point to the last node in the list, so we make **temp2** point to what **temp1** points to



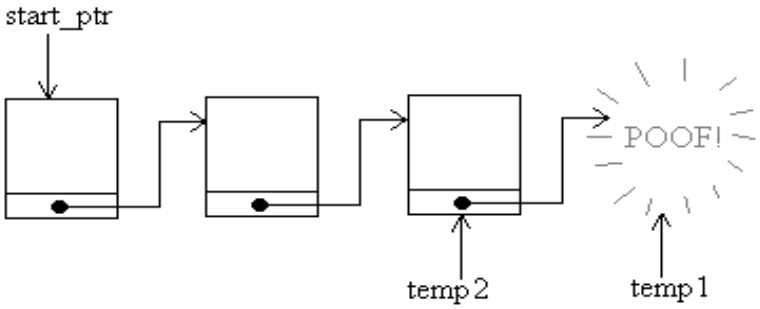
and temp1 is made to point to the next node alone:



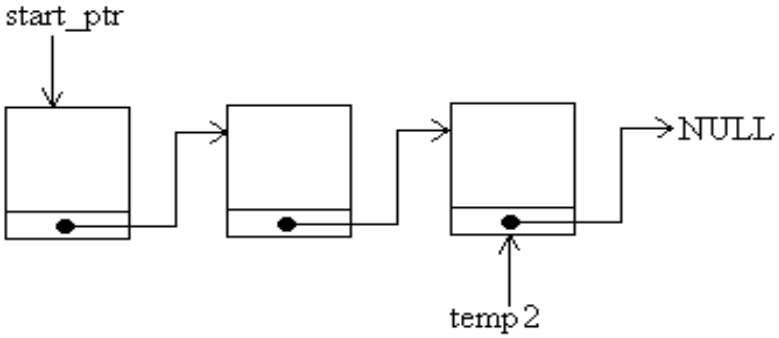
Eventually, this goes on until **temp1** really is pointing to the last node in the list, with **temp2** pointing to the second last node:



Now we have reached step 8. The next thing to do is to delete the node pointed to by **temp1**



and set the **next** pointer of what **temp2** indicates to NULL as indicated below:



We suppose you want some code for all that! All right then,

```
void delete_end_node ( )
  {
    node *temp1, *temp2;
    if (start_ptr = = NULL)
        cout << "The list is empty!" << endl;
    else
      {
        temp1 = start_ptr;
        while (temp1->next ! = NULL)
          {
            temp2 = temp1;
            temp1 = temp1->next;
          }
        delete temp1;
        temp2->next = NULL;
      }
  }
```

The code seems a lot shorter than the explanation!

Now, the sharp-witted amongst you will have spotted a problem. If the list only contains one node, the code above will malfunction. This is because the function goes as far as the **temp1 = start_ptr** statement, but never gets as far as setting up **temp2**. The code above has to be adapted so that if the first node is also the last (has a NULL **next** pointer), then it is deleted and the **start_ptr** pointer is assigned to NULL. In this case, there is no need for the pointer **temp2**:

```
void delete_end_node( )
  {
    node *temp1, *temp2;
    if (start_ptr = = NULL)
        cout << "The list is empty!" << endl;
    else
      { temp1 = start_ptr;
        if (temp1->next = = NULL)     // This part is new!
          { delete temp1;
            start_ptr = NULL;
          }
        else
      { while (temp1->next ! = NULL)
              { temp2 = temp1;
                temp1 = temp1->next;
              }
            delete temp1;
            temp2 -> next = NULL;
          }
      }
  }
```

3.3. Doubly Linked Lists

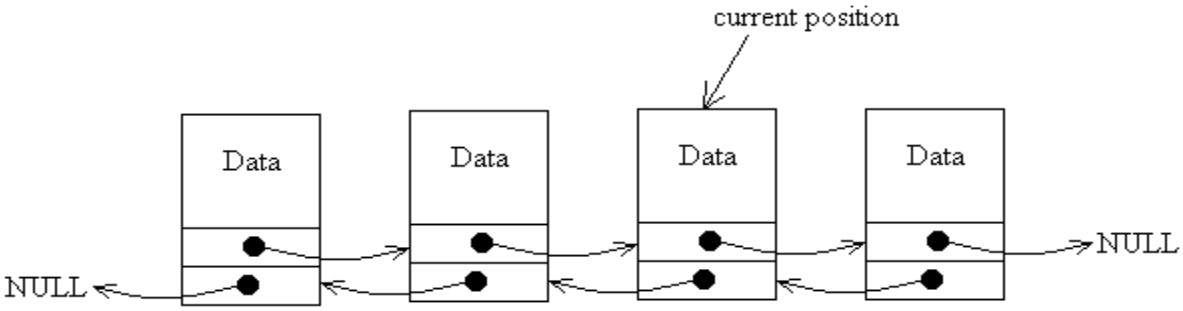
3.3.1. Introduction

As we saw in the previous section, removing an element at the tail of a singly linked list is not easy. Indeed, it is time consuming to remove any node other than the head in a singly linked list, since we do not have a quick way of accessing the node immediately preceding the one we want to remove. There are many applications where we do not have quick access to such a predecessor node. For such applications, it would be nice to have a way of going both directions in a linked list.

There is a type of linked list that allows us to go in both directions—forward and reverse—in a linked list. It is the doubly linked list. In addition to its element member, a node in a doubly linked list stores two references—a ***next*** link which points to the next node in the list, and a ***prev*** link, which points to the previous node in the list. Such lists allow for a great variety of quick update operations, including efficient insertion and removal at any given position.

If you've mastered how to do singly linked lists, then it shouldn't be much of a leap to doubly linked lists. A doubly linked list is one where there are links from each node in both directions. We can traverse in both directions in a doubly linked list as each and every node of a double linked list contains address of next node along with address of previous node also. Thus a node in a doubly linked list contains three fields.

i. Data field
ii. Address of next node
iii. Address of previous node



You will notice that each node in the list has two pointers; one to the next node and one to the previous one - again, the ends of the list are defined by NULL pointers. Also there is no pointer to the start of the list. Instead, there is simply a pointer to some position in the list that can be moved left or right, called **current** pointer.

The reason we needed a start pointer in the ordinary linked list is because, having moved on from one node to another, we can't easily move back, so without the start pointer, we would lose track of all the nodes in the list that we have already passed. With the doubly linked list, we can move the current pointer backwards and forwards at will.

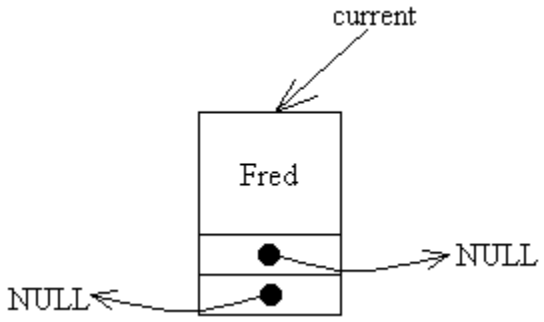
The most common reason to use a doubly linked list is because it is easier to implement than a singly linked list. While the code for the doubly linked implementation is a little longer than for the singly linked version, it tends to be a bit more "obvious" in its intention, and so easier to implement and debug. Member functions for processing doubly linked lists are slightly more complicated than their singly linked counterparts because there is one more pointer member to be maintained. Removing an element at the tail of a singly linked list is not easy. Indeed, it is time consuming to remove any node other than the head in a singly linked list, since we do not have a quick way of accessing the node in front of the one we want to remove.

3.3.2. Creating Doubly Linked Lists

The nodes for a doubly linked list would be defined as follows:

```
struct node{
        char name[20];
        node *next;    // Pointer to next node
        node *prev;    // Pointer to previous node
         };
   node *current;
current = new node;
current -> name = "Fred";
current -> next = NULL;
current -> prev = NULL;
```

We have also included some code to declare the first node and set its pointers to NULL. It gives the following situation:



We still need to consider the directions 'forward' and 'backward', so in this case, we will need to define functions to add a node to the start of the list (left-most position) and the end of the list (right-most position).

3.3.3. Inserting Node to a Doubly Linked List

Because of its double link structure, it is possible to insert a node at any position within a doubly linked list. To add a node to a list, the node has to be created, its data members properly initialized, and then the node needs to be incorporated into the list.

Move the current pointer to the *start* (*head*) node in the doubly linked list, and then inserting a node at the start of a doubly linked list involves the following steps.

1. Create a new node, and make the data field of the node with a value being inserted.
2. Make the *previous* address of the new node to *NULL*.
3. Make the *next* address of the node to the value of *head* so that this member points to the first node in the list. But now, the new node should become the first node; therefore,
4. Head is set to point to the new node. But the new node is not yet accessible from its successor; to rectify this,
5. The *previous* member of the successor of the new node is set to point to the new node

The algorithm for inserting a node to the start of a doubly linked list is the following.

```
void Add_Node_At_Start ( String new_name)
  {
    // Declare a temporary pointer and move it to the start
     node *temp = current;
     while (temp -> prev ! = NULL)
         temp = temp -> prev;
    // Declare a new node and link it in
      node *temp2;
      temp2 = new node;
      temp2 -> name = new_name;  // Store the new name in the node
      temp2 -> prev = NULL;       // This is the new start of the list
      temp2 -> next = temp;       // Links to current list
      temp -> prev = temp2;
  }
```

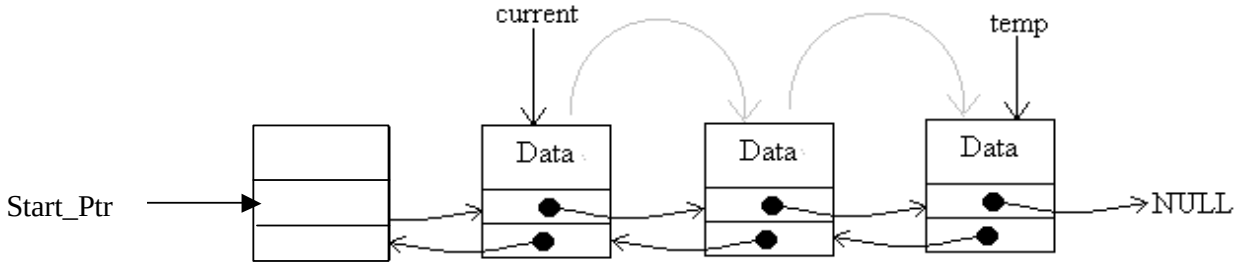
Move the current pointer of the list to the last node (*tail*), then inserting a node at the end of a doubly linked list consists of the following actions:

1. A new node is created, and then initialize its data filed with value being inserted,
2. Set the *next* member of the node to *NULL*,
3. Set the *previous* member to the value of *tail* so that this member points to the last node in the list. But now, the new node should become the last node; therefore,
4. Make *tail* to point to the new node. But the new node is not yet accessible from its predecessor; to rectify this,
5. Set the *next* member of the predecessor is set to point to the new node.

The algorithm for inserting a new node at the end of a doubly linked list is the following.

```
void Add_Node_At_End (String  new_name)
 {
   // Declare a temporary pointer and move it to the end
   node *temp = current;
   while (temp -> next ! = NULL)
      temp = temp -> next;
    // Declare a new node and link it in
     node *temp2;
     temp2 = new node;
     temp2 -> name = new_name;  // Store the new name in the node
     temp2 -> next = NULL;       // This is the new start of the list
     temp2 -> prev = temp;       // Links to current list
     temp -> next = temp2;
  }
```

Here, the new name is passed to the appropriate function as a parameter. We'll go through the function for adding a node to the right-most end of the list. The method is similar for adding a node at the other end. Firstly, a temporary pointer is set up and is made to march along the list until it points to last node in the list.



After that, a new node is declared, and the name is copied into it. The next pointer of this new node is set to NULL to indicate that this node will be the new end of the list. The previous pointer of the new node is linked into the last node of the existing list. The next pointer of the current end of the list is set to the new node.

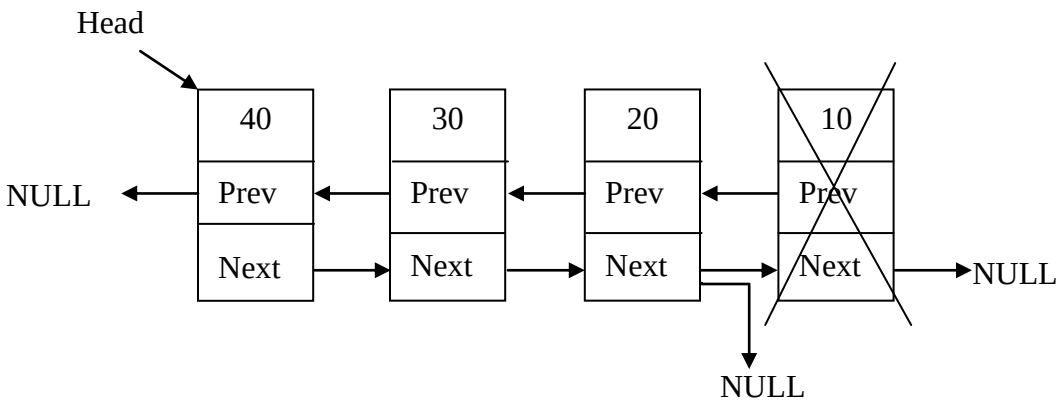
Inserting a node in the middle of a doubly linked list is also possible. Given a node *v* of a doubly linked list (which could possibly be the header, but not the trailer), let *z* be a new node that we wish to insert immediately after *v*. Let *w* be the node following *v*, that is, *w* is the node pointed to by *v*'s next link. To insert *z* after *v*, we link it into the current list, by performing the following operations:

- Make *z*' s *prev* link point to *v*
- Make *z*'s *next* link point to *w*
- Make *w*'s *prev* link point to *z*
- Make *v*'s *next* link point to *z*

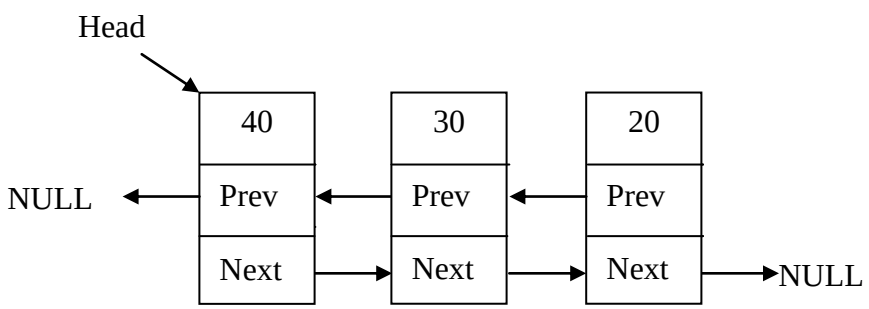
3.3.4. Deleting a Node from Doubly Linked List

When it comes to deleting nodes, we have three choices: Delete a node from the start (front end) of the list, delete one from the end (tail end) of the list, or delete one from somewhere in the middle.

Here let us see to delete node from the end of a double linked list (for others workout by your on). Deleting the last node from the doubly linked list is straight forward because there is direct access from the last node to its predecessor. When deleting the last node from the list, the temporary variable is set to the value in the node, then *tail* is set to its predecessor, and the last and now redundant node is deleted. In this way, the *next* to last node becomes the last node. The *next* member of the *tail* node is a dangling reference; therefore, *next* is set to *NULL*.



After deleting the last node, the doubly linked list looks the following.



The algorithm for deleting the last element from a doubly linked list is the following.

```
void Delete_Node_From_End ( )
 {
   // Declare a temporary pointer and move it to the end
   node *temp = current;
      while (temp -> next ! = NULL)
           temp = temp -> next;
  tail = temp -> prev;
  delete temp;
  tail -> next = NULL;
}
```

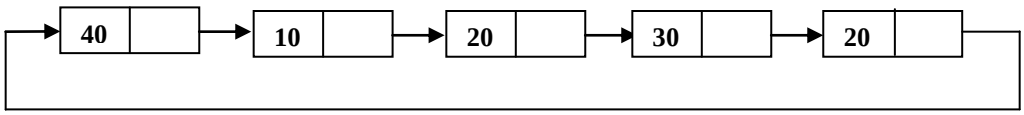
Likewise, it is easy to remove a node *v* in the middle of a doubly linked list. We access the nodes *u* and *w* on either side of *v* using *v*'s *Prev* and *Next* values (these nodes must exist). To remove node *v*, we simply have *u* and *w* point to each other instead of to *v*. We refer to this operation as the linking out of *v*. We also *NULL* out *v*'s *prev* and *next* pointers so as not to retain old references into the list.

- Make *u*'s *next* link point to *w* (i.e. u -> next = v -> next)
- Make *w*' s *prev* link point to *u* (i.e. w -> prev = v -> prev)
- Delete node *v*

3.4. Circular Linked Lists

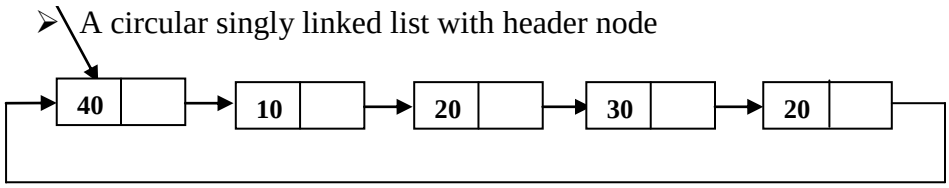
The next fields in the last node contain a pointer back to the first node rather than NULL pointer, make last node point to the first (instead of null). The pointer from the last element in the list points back to the first element; such lists are known as a circular list. A circular list is when nodes form a ring: the list is finite and each node has a successor. A circular list does not have a natural "first" or "last" node.

- A circular singly linked list without header node

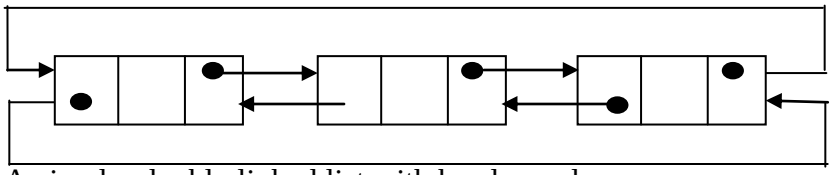


Head

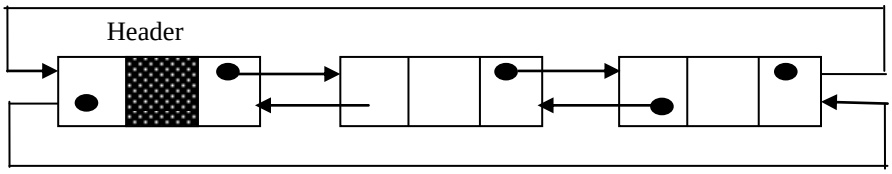
- A circular singly linked list with header node



- A circular doubly linked list without header node



- A circular doubly linked list with header node



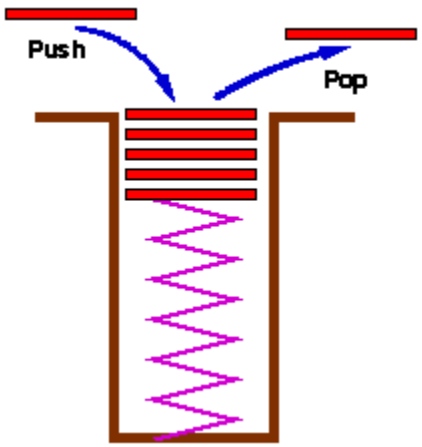
CHAPTER FOUR

4. DATA STRUCTURES AND APPLICATIONS : STACKS

4.1. Introduction

The stack is a list-like structure in which elements may be inserted or removed from only one end. While this restriction makes stacks less flexible than lists, it also makes stacks both efficient (for those operations they can do) and easy to implement. Many applications require only the limited form of insert and remove operations that stacks provide. In such cases, it is more efficient to use the simpler stack data structure rather than the generic list.

Stack is a simple data structure, in which insertion and deletion occur at the same end. It is a LIFO (Last In First Out) structure. The accessible element of the stack is called the **top** element. Elements are not said to be inserted, they are *pushed* onto the stack. When removed, an element is said to be *popped* from the stack.



A Stack is an ordered collection of data items, into which new items may be inserted and from which items may be deleted at only one end; and that end is called Top of the Stack.

A common model of a stack is a plate or coin stacker. Plates are "pushed" onto to the top and "popped" off the top.

4.2. Fundamental operations on stack

The fundamental operations on a stack are *push*, which is equivalent to an insert, and *pop*, which deletes the most recently inserted element. The most recently inserted element can be examined prior to performing a *pop* by use of the *top* routine. A *pop* or *top* on an empty stack is generally considered an error in the stack ADT. On the other hand, running out of space when performing a push is an implementation limit.

Formally, a stack is an abstract data type (ADT) that supports the following operations:

- **Push (x):** Insert the element *x* at the top of the stack.
- **Pop ( ):** Remove the top element from the stack; an error occurs if the stack is empty.

6 operation

- **Empty ( ):** Check to see if the stack is empty or not. Return true if the stack is empty and false otherwise.
- **Full ( ):** Check to see if the stack is full or not.
- **top ( )**: Return the top most element in the stack without removing it; an error occurs if the stack is empty.
- **Size ( )**: Return the number of elements in the stack.

A series of push and pop operations is shown in the following figure. After pushing number 10 onto an empty stack, the stack contains only this number. After pushing 5 on the stack, the number is placed on top of 10 so that, when the popping operation is executed, 5 is removed from the stack, because it arrived after 10, and 10 is left on the stack. After pushing 15 and then 7, the topmost element is 7, and this number is removed when executing the popping operation, after which the stack contains 10 at the bottom and 15 above it.

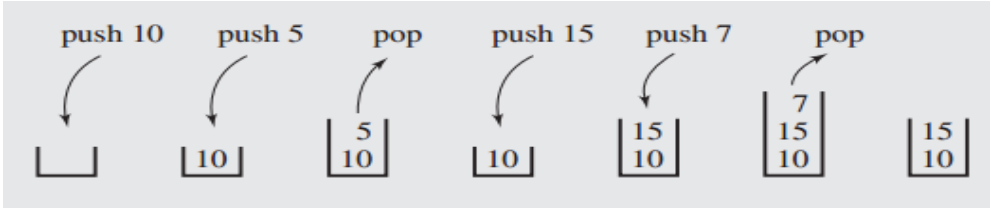


Figure 4.1: A series of operations executed on a stack.

The following table shows a series of stack operations and their effects on an initially empty stack of integers.

Operation	*Output*	*Stack Contents*
push(5)	–	(5)
push(3)	–	(5,3)
pop()	–	(5)
push(7)	–	(5,7)
pop()	–	(5)
top()	5	(5)
pop()	–	()
pop()	"error"	()
top()	"error"	()
empty()	true	()
push(9)	–	(9)
push(7)	–	(9,7)
push(3)	–	(9,7,3)
push(5)	–	(9,7,3,5)
size()	4	(9,7,3,5)
pop()	–	(9,7,3)
push(8)	–	(9,7,3,8)
pop()	–	(9,7,3)
top()	3	(9,7,3)

4.3. Implementation of Stacks

Stacks can be implemented both as an array (contiguous list) and as a linked list. We want a set of operations that will work with either type of implementation: i.e. the method of implementation is hidden and can be changed without affecting the programs that use them.

As with lists, there are many variations on stack implementation. The two approaches presented here are array-based and linked stacks, which are analogous to array-based and linked lists, respectively.

For the implementation of stack using array, *top* is defined to be the array index of the first free position in the stack. Thus, an empty stack has *top* set to 0, the first available free position in the array. (Alternatively, *top* could have been defined to be the index for the *top* element in the stack, rather than the first free position. If this had been done, the empty list would initialize *top* as -1). Methods *push* and *pop* simply place an element into, or remove an element from, the array position indicated by *top*. Because *top* is assumed to be at the first free position, *push* first inserts its value into the *top* position and then increments *top*, while *pop* first decrements *top* and then removes the *top* element.

4.3.1. Array Implementation of Stacks

Although an array cannot be a stack it can be the home of the stack i.e., an array can be declared with a range that is large enough for the maximum size of the stack. During the course of program execution the stack will grow and shrink within the space reserved for it. One end of the array will be fixed as bottom of the stack, while the top of the stack will constantly shift as items are popped and pushed. Thus another field is needed to keep track of the current position of the top of the stack.

i). <u>The Push operation</u>

To remind you, addition of an element is known as the push operation. So, if an array is given to you, which is supposed to act as a stack, you know that it has to be a static stack; meaning, data will overflow if you cross the upper limit of the array. So, keep this in mind.

The idea of push operation is the following

```
Push ( )
    {
     if there is room
       put an item on the top of the stack
     else
       give an error message
     }
```

Push operation is similar to inserting an element at the end of the list (ordinary not linked list).

Algorithm:

Step-1: Increment the Stack TOP by 1. Check whether it is always less than the Upper Limit of the stack. If it is less than the Upper Limit go to step-2 else report -"Stack Overflow"
Step-2: Put the new element at the position pointed by the TOP

Implementation:

```
static int stack[UPPERLIMIT];
int top= -1; /*stack is empty*/
void push(int item)
  {
    top = top + 1;
    if(top < UPPERLIMIT)
         stack[top] = item;
    else
         cout<<"Stack Overflow";
   }
```

Note: - In array implementation, we have taken TOP = -1 to signify the empty stack, as this simplifies the implementation. While pushing an element, top value will be incremented by '1' and one element will be added to stack.

ii). The Pop operation

POP is the synonym for delete when it comes to Stack. So, if you're taking an array as the stack, remember that you'll return an error message, "Stack underflow", if an attempt is made to Pop an item from an empty Stack.

The idea of pop operation is the following.

```
Pop ( )
      {
        if stack not empty
         {
          return the value of the top item
          remove the top item from the stack
         }
        else
            give an error message
      }
```

Pop operation is similar to deleting the end element of an ordinary list.

Algorithm

Step-1: If the Stack is empty then give the alert "Stack underflow" and quit; or else go to step-2
Step-2:

a) Hold the value for the element pointed by the TOP
b) Put a NULL value instead
c) Decrement the TOP by 1

Implementation:

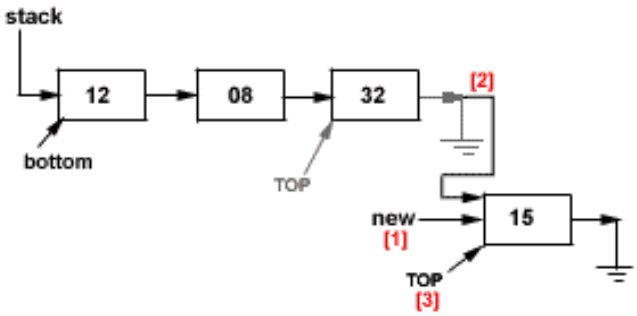
```
static int stack[UPPPERLIMIT];
int top = -1;
int pop( )
 {
   int del_val = 0;
   if(top = = -1)
   cout<<"Stack underflow"; /*step-1*/
    else
    {
   del_val = stack[top];  /*step-2*/
   stack[top] = NULL;
   top = top -1;
    }
   return(del_val);
 }
```

Note: - Step-2: (b) signifies that the respective element has been deleted. While popping an element, top value will be decremented by '1' and one element will be deleted from the stack.

4.3.2. Linked List Implementation of Stacks

i). The PUSH operation

It's very similar to the insertion operation in a dynamic singly linked list. The only difference is that here you'll add the new element only at the end of the list, which means addition can happen only from the TOP. Since a dynamic list is used for the stack, the Stack is also dynamic, means it has no prior upper limit set. So, we don't have to check for the Overflow condition at all!



In Step [1] we create the new element to be pushed to the Stack. In Step [2] the TOP most element is made to point to our newly created element. In Step [3] the TOP is moved and made to point to the last element in the stack, which is our newly added element.

Algorithm

Step-1: If the Stack is empty go to step-2 or else go to step-3

Step-2: Create the new element and make your "stack" and "top" pointers point to it and quit.

Step-3: Create the new element and make the last (top most) element of the stack to point to it

Step-4: Make that new element your TOP most element by making the "top" pointer point to it.

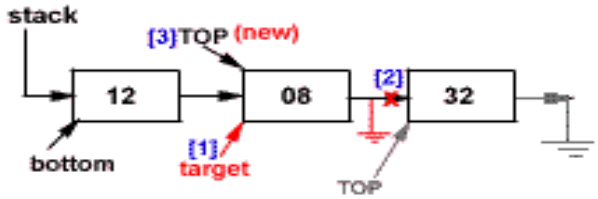
Implementation:

```
struct node
  {
   int item;
   struct node *next;
  };
struct node *stack = NULL; /*stack is initially empty*/
struct node *top = stack;

    void push(int value)
    {
        if(stack = = NULL)  /*step-1*/
          {
           temp = new node  /*step-2*/
           temp -> item = value;
           temp -> next = NULL;
           stack = temp;
           top = stack; /*step-4*/
          }
       else
         {
          temp = new node; /*step-3*/
          temp -> item = value;
          temp -> next = NULL;
           top -> next = temp;
           top = temp;   /*step-4*/
         }
    }
```

ii). <u>The POP operation</u>

This is again very similar to the deletion operation in any Linked List, but you can only delete from the end of the list and only one at a time; and that makes it a stack. Here, we'll have a list pointer, "target", which will be pointing to the last but one element in the List (stack). Every time we POP, the TOP most element will be deleted and "target" will be made as the TOP most element.



In step [1] we got the "target" pointing to the last but one node. In step [2] we freed the TOP most element. In step [3] we made the "target" node as our TOP most element.

Supposing you have only one element left in the Stack, then we won't make use of "target" rather we'll take help of our "bottom" pointer.

Algorithm

Step-1: If the Stack is empty then give an alert message "Stack Underflow" and quit; or else proceed
Step-2: If there is only one element left go to step-3 or else step-4
Step-3: Free that element and make the "stack", "top" and "bottom" pointers point to NULL and quit
Step-4: Make "target" point to just one element before the TOP; free the TOP most element; make "target" as your TOP most element

Implementation:

```
struct node
 {
   int item;
   struct node *next;
 };
struct node *stack = NULL; /*stack is initially empty*/
struct node *top = stack;

int pop( )
{   int pop_val = 0;
    struct node *target = stack;
        if(stack = = NULL)  /*step-1*/
           cout<<"Stack Underflow";
        else
          {   if(top = = bottom)  /*step-2*/
              {  pop_val = top -> item;   /*step-3*/
                 delete top;
                 stack = NULL;
                 top = bottom = stack;
              }
            else   /*step-4*/
             {
                while(target -> next ! = top)
                    target = target -> next;
                pop_val = top -> item;
                delete top;
                top = target;
                target -> next = NULL;
             }
          }
    return(pop_val);
}
```

4.4. Applications of Stacks

4.4.1. Evaluation of Algebraic Expressions

E.g. **4 + 5 * 5**

Simple calculator: 45
Scientific calculator: 29 (correct)

Question:

Can we develop a method of evaluating arithmetic expressions without having to 'look ahead' or 'look back'? i.e. consider the quadratic formula:

x = (-b+(b^2-4*a*c)^0.5)/(2*a)

where **^** is the power operator, or, as you may remember it :

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

In its current form we cannot solve the formula without considering the ordering of the parentheses. i.e. we solve the innermost parenthesis first and then work outwards also considering operator precedence. Although we do this naturally, consider developing an algorithm to do the same, possible but complex and inefficient. Instead re-express the expression.

Re-expressing the Expression

Computers solve arithmetic expressions by restructuring them so the order of each calculation is embedded in the expression. Once converted an expression can then be solved in one pass.

Types of Expression

The normal (or human) way of expressing mathematical expressions is called infix form, e.g. **4+5*5**. However, there are other ways of representing the same expression, either by writing all operators before their operands or after them,

e.g.: **4 5 5 * +**

+ 4 * 5 5

This method is called Polish Notation (because this method was discovered by the Polish mathematician Jan Lukasiewicz).

When the operators are written before their operands, it is called the **prefix** form

e.g. **+ 4 * 5 5**

When the operators come after their operands, it is called **postfix** form (**suffix** form or **reverse polish notation**)

e.g. **4 5 5 * +**

Advantages of RPN (Reverse Polish Notation or postfix)

- Parentheses are unnecessary
- Easy for a computer (compiler) to evaluate an arithmetic expression Postfix (Reverse Polish Notation)

Postfix notation arises from the concept of post-order traversal of an expression tree (this concept will be covered when we look at trees).

For now, consider postfix notation as a way of redistributing operators in an expression so that their operation is delayed until the correct time.

Consider again the quadratic formula:

x = (-b+(b^2-4*a*c)^0.5)/(2*a)

In postfix form the formula becomes:

x b @ b 2 ^ 4 a * c * - 0.5 ^ + 2 a * / =

where @ represents the unary - operator.

Notice the order of the operands remain the same but the operands are redistributed in a non-obvious way (an algorithm to convert infix to postfix can be derived).

The reason for using postfix notation is that a fairly simple algorithm exists to evaluate such expressions based on using a stack.

Postfix Evaluation

Consider the postfix expression: **6 5 2 3 + 8 * + 3 + ***

Algorithm:

```
initialize stack to empty;
while (not end of postfix expression)
{
  get next postfix item;
  if(item is value)
    push it onto the stack;
  else if(item is binary operator) {
    pop the stack to x;
    pop the stack to y;
    perform y operator x;
    push the results onto the stack;
  } else if (item is unary operator) {
    pop the stack to x;
    perform operator(x);
    push the results onto the stack
  }
}
```

The single value on the stack is the desired result.

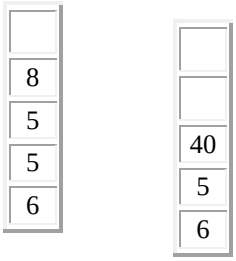
Binary operators: +, -, *, /, etc.,
Unary operators: **unary minus**, **square root**, **sin**, **cos**, **exp**, etc.,

So for **6 5 2 3 + 8 * + 3 + ***
the first item is a value (6) so it is pushed onto the stack
the next item is a value (5) so it is pushed onto the stack
the next item is a value (2) so it is pushed onto the stack
the next item is a value (3) so it is pushed onto the stack
and the stack becomes

the remaining items are now: + 8 * + 3 + *

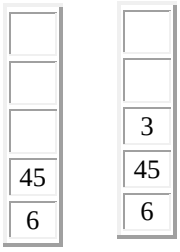
So next a '+' is read (a binary operator), so 3 and 2 are popped from the stack and their sum '5' is pushed onto the stack, and the stack becomes:

Next 8 is pushed and the next item is the operator *:



(8, 5 popped, 40 pushed)

Next the operator + followed by 3:



(40, 5 popped, 45 pushed, 3 pushed)

Next is operator +, so 3 and 45 are popped and 45 + 3 = 48 is pushed, the stack becomes:

Next is operator *, so 48 and 6 are popped, and 6 * 48 = 288 is pushed, the stack becomes:

Now there are no more items and there is a single value on the stack, representing the final answer 288.

Note the answer was found with a single traversal of the postfix expression, with the stack being used as a kind of memory storing values that are waiting for their operands.

4.4.2. Infix to Postfix (RPN) Conversion

Of course postfix notation is of little use unless there is an easy method to convert standard (infix) expressions to postfix. Again a simple algorithm exists that uses a stack:

Algorithm

```
initialize stack and postfix output to empty;
while(not end of infix expression)
{    get next infix item
     if(item is value) append item to postfix o/p
     else if(item = = '(') push item onto stack
     else if(item = = ')') {
        pop stack to x
      while(x ! = '(')
        app. x to pfix o/p & pop stack to x
   }
else {
while(precedence(stack top) > = precedence(item))
        pop stack to x & app. x to pfix o/p
      push item onto stack
   }
}
while(stack not empty)
   pop stack to x and append x to pfix o/p
```

Operator Precedence (for this algorithm):

4 : '**(**' - only popped if a matching '**)**' is found
3 : All unary operators
2 : / *****
1 : + -

The algorithm immediately passes values (operands) to the postfix expression, but remembers (saves) operators on the stack until their right-hand operands are fully translated.

Eg., consider the infix expression **a+b*c+(d*e+f)*g**

Stack	Output
+	**ab**
* +	**abc**
+	**abc*+**
* (+	**abc*+de**
+ (+	**abc*+de*f**
+	**abc*+de*f+**
* +	**abc*+de*f+g**
empty	**abc*+de*f+g*+**

Deque =double ended queue
but Dequeue is operation on queue
it is deleteing element from front

CHAPTER FIVE
5. QUEUES

In a queue, new elements are added at one end called the rear or tail, and existing elements are removed from the other end called the front or head. This ensures that the first element added to the queue is the first one to be removed.

5.1. Introduction

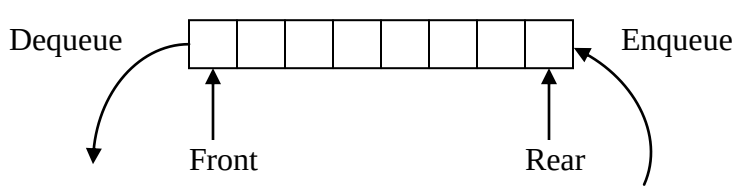
Like the stack, the queue is a list-like structure that provides restricted access to its elements. Queue elements may only be inserted at the back (called an enqueue operation) and removed from the front (called a dequeue operation). Queues operate like standing in line at a movie theater ticket counter. If nobody cheats, then newcomers go to the back of the line. The person at the front of the line is the next to be served. Thus, queues release their elements in order of arrival. A queue is a "FIFO" list, which stands for "First-In, First-Out."

Formally, the queue abstract data type defines a container that keeps elements in a sequence, where element access and deletion are restricted to the first element in the sequence, which is called the *front* of the queue, and element insertion is restricted to the end of the sequence, which is called the *rear* of the queue. This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in first-out (FIFO) principle.

A queue is an ordered collection of items where an item is inserted at one end and usually called REAR or TAIL, and an item is removed from the other end, usually called FRONT or HEAD of the queue. So, a queue is a dynamic list with limited across to elements. Elements can be inserted in a queue at any time, but only the element that has been in the queue the longest can be removed at any time.

A queue is simply a waiting line that grows by adding elements to its end and shrinks by taking elements from its front. Unlike a stack, a queue is a structure in which both ends are used: one for adding new elements and one for removing them. Therefore, the last element has to wait until all elements preceding it on the queue are removed.

In a multi-processing system, for example, when jobs are submitted to a printer, we expect the least recent or most senior job to be printed first. This order is not only fair, but it also is required to guarantee that the first job does not wait forever. Thus you can expect to find printer queues on all large systems.



5.2. Fundamental operations on queues

The queue abstract data type (ADT) supports the following two fundamental operations:

- ***Enqueue*** (el): Insert element el at the rear of the queue.
- ***Dequeue*** ( ): Remove and return from the queue the object at the front; an error occurs if the queue is empty.

Additionally, similar to the case with the Stack ADT, the queue ADT includes the following supporting methods:

- ***Size*** ( ): Return the number of objects in the queue.
- ***Empty*** ( ): Return true if the queue is empty and false otherwise.
- ***Front*** ( ): Return, but do not remove, the front object in the queue; an error occurs if the queue is empty.

A series of enqueue and dequeue operations is shown in the following figure. This time, unlike for stacks, the changes have to be monitored both at the beginning of the queue and at the end. The elements are enqueued on one end and dequeued from the other. For example, after enqueuing 10 and then 5, the dequeue operation removes 10 from the queue.

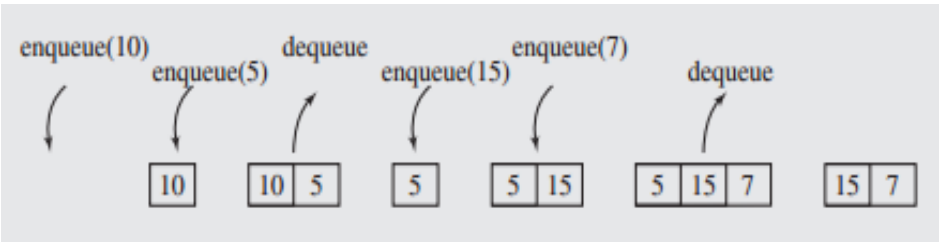


Figure 5.1: A series of operations executed on a queue.

The following table shows a series of queue operations and their effects on an initially empty queue, Q, of integers.

Operation	***Output***	*front* ← *Q* ← *rear*
enqueue(5)	–	(5)
enqueue(3)	–	(5,3)
front()	5	(5,3)
size()	2	(5,3)
dequeue()	–	(3)
enqueue(7)	–	(3,7)
dequeue()	–	(7)
front()	7	(7)
dequeue()	–	()
dequeue()	"error"	()
empty()	true	()

5.3. Implementation of queues

5.3.1. Array implementation

One possible queue implementation is an array, although this may not be the best choice. Elements are added to the end of the queue, but they may be removed from its beginning, thereby releasing array cells. These cells should not be wasted. Therefore, they are utilized to enqueue new elements, whereby the end of the queue may occur at the beginning of the array.

i. Simple array implementation of enqueue and dequeue

The analysis for the simple array based queue is stated here after.

Consider the following array: int Num[MAX_SIZE];

We need to have two integer variables that tell the index of the front element and the index of the rear (last) element. We also need an integer variable that tells the total number of data in the queue.

```
int FRONT = -1, REAR = -1;
int QUEUESIZE = 0;
```

- To enqueue data to the queue
  - ✓ check if there is space in the queue

```
REAR < MAX_SIZE - 1 ?
Yes: - Increment REAR
     - Store the data in Num[REAR]
     - Increment QUEUESIZE
       FRONT = = -1?
           Yes: - Increment FRONT
No:  - Queue Overflow
```

- To dequeue data from the queue
  - ✓ check if there is data in the queue

```
QUEUESIZE > 0 ?
Yes: - Copy the data in Num[FRONT]
     - Increment FRONT
     - Decrement QUEUESIZE
No:  - Queue Underflow
```

The implementation for these operations look like the following:

```
const int MAX_SIZE=100;
int FRONT =-1, REAR =-1;
int QUEUESIZE = 0;

void enqueue(int x)
 {
    if(REAR < MAX_SIZE - 1)
     {
         REAR++;
         Num[REAR] = x;
         QUEUESIZE ++;
         if(FRONT = = -1)
           FRONT++;
     }
     else
        cout<<"Queue Overflow";
     }
```

```
int dequeue ( )
{
  int x;
  if(QUEUESIZE > 0)
   {
    x = Num[FRONT];
    FRONT++;
    QUEUESIZE - -;
    return(x);
   }
  else
   cout<<"Queue Underflow";
  }
```

ii. <u>Circular array implementation of enqueue and dequeue</u>

A problem with simple arrays is we run out of space even if the queue never reaches the size of the array. Thus, simulated circular arrays (in which freed spaces are re-used to store data) can be used to solve this problem. Whenever front or rear gets to the end of the array, it is wrapped around to the beginning.

The queue is full if the first element immediately precedes the last element in the counterclockwise direction. However, because a circular array is implemented with a “normal” array, the queue is full if either the first element is in the first cell & the last element is in the last cell (Figure 5.2a) or if the first element is right after the last (Figure 5.2b). Similarly, enqueue ( ) and dequeue ( ) have to consider the possibility of wrapping around the array when adding or removing elements. For example, enqueue ( ) can be viewed as operating on a circular array (Figure 5.2c), but in reality, it is operating on a one-dimensional array. Therefore, if the last element is in the last cell and if any cells are available at the beginning of the array, a new element is placed there (Figure 5.2d). If the last element is in any other position, then the new element is put after the last, space permitting (Figure 5.2e). These two situations must be distinguished when implementing a queue viewed as a circular array (Figure 5.2f).

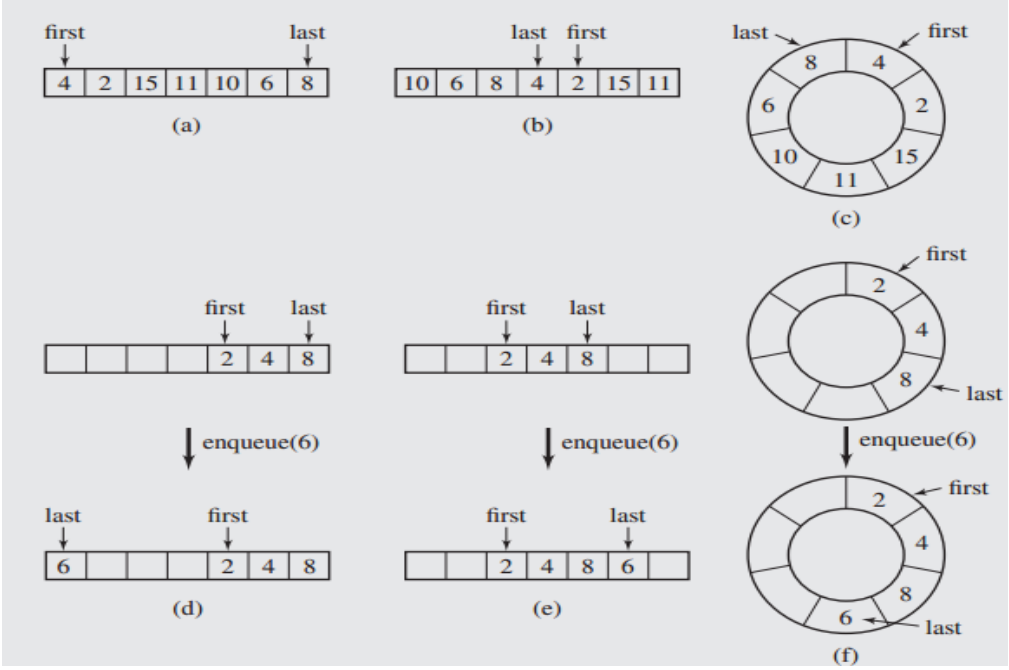


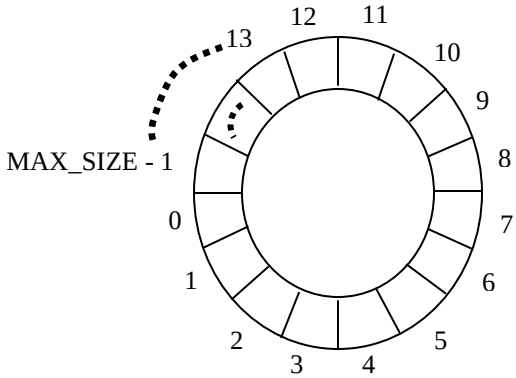
Figure 5.2: (a–b) Two possible configurations in an array implementation of a queue when the queue is full. (c) The same queue viewed as a circular array. (f) Enqueuing number 6 to a queue storing 2, 4, and 8. (d – e) The same queue seen as a one-dimensional array with the last element (d) at the end of the array and (e) in the middle.

Example: Consider a queue with MAX_SIZE = 4

The following table shows the same possible operations performed on both simple array based and circular array based queue.

Operation	Simple array							Circular array						
	Content of the array				Content of the Queue	QUEUE SIZE	Message	Content of the array				Content of the queue	QUEUESIZE	Message
Enqueue(B)	B				B	1		B				B	1	
Enqueue(C)	B	C			BC	2		B	C			BC	2	
Dequeue( )		C			C	1			C			C	1	
Enqueue(G)		C	G		CG	2			C	G		CG	2	
Enqueue (F)		C	G	F	CGF	3			C	G	F	CGF	3	
Dequeue( )			G	F	GF	2				G	F	GF	2	
Enqueue(A)			G	F	GF	2	Overflow	A		G	F	GFA	3	
Enqueue(D)			G	F	GF	2	Overflow	A	D	G	F	GFAD	4	
Enqueue(C)			G	F	GF	2	Overflow	A	D	G	F	GFAD	4	Overflow
Dequeue( )				F	F	1		A	D		F	FAD	3	
Enqueue(H)				F	F	1	Overflow	A	D	H	F	FADH	4	
Dequeue ( )					Empty	0		A	D	H		ADH	3	
Dequeue( )					Empty	0	Underflow		D	H		DH	2	
Dequeue( )					Empty	0	Underflow			H		H	1	
Dequeue( )					Empty	0	Underflow					Empty	0	
Dequeue( )					Empty	0	Underflow					Empty	0	Underflow

The circular array implementation of a queue with MAX_SIZE can be simulated as follows:



The analysis for the circular array based queue is stated in the following manner.

Consider the following array: int Num[MAX_SIZE];

We need to have two integer variables that tell the index of the front element and the index of the rear (last) element. We also need an integer variable that tells the total number of data in the queue.

```
int FRONT = -1, REAR = -1;
int QUEUESIZE = 0;
```

- To enqueue data to the queue
  - ✓ check if there is space in the queue

    QUEUESIZE < MAX_SIZE?

    Yes:
    - Increment REAR
      - REAR = = MAX_SIZE?
        - Yes: REAR = 0
    - Store the data in Num[REAR]
    - Increment QUEUESIZE
      - FRONT = = -1?
        - Yes: - Increment FRONT

    No: - Queue Overflow

- To dequeue data from the queue
  - ✓ check if there is data in the queue

    QUEUESIZE > 0 ?

    Yes:
    - Copy the data in Num[FRONT]
    - Increment FRONT
      - FRONT = = MAX_SIZE?
        - Yes: FRONT = 0
    - Decrement QUEUESIZE

    No: - Queue Underflow

The implementation for these operations look like the following:

```
const int MAX_SIZE=100;
int FRONT = -1, REAR = -1;
int QUEUESIZE = 0;
```

```
void enqueue(int x)
    {
      if(QUEUESIZE<MAX_SIZE)
        {
          REAR + +;
          if(REAR = = MAX_SIZE)
            REAR = 0;
          Num[REAR] = x;
          QUEUESIZE + +;
          if(FRONT = = -1)
            FRONT + +;
        }
      else
        cout<<"Queue Overflow";
    }
```

```
int dequeue()
    {
      int x;
      if(QUEUESIZE > 0)
       {
        x = Num[FRONT];
        FRONT + +;
        if(FRONT = = MAX_SIZE)
          FRONT = 0;
        QUEUESIZE - -;
        return (x);
        }
      else
       cout<<"Queue Underflow";
     }
```

5.3.2. Linked list implementation

We can efficiently implement the queue ADT using linked list. For efficiency reasons, we choose the front of the queue to be at the head of the list, and the rear of the queue to be at the tail of the list. In this way, we remove from the head and insert at the tail. Note that we need to maintain references to both the head and tail nodes of the list.

To implement the queue using singly linked list, assume we have two pointers head and tail, which point to the front and rear of the queue respectively. That is, we place the front of the queue at the head of the linked list and the rear of the queue at the tail.

Hence, enqueue is inserting a node at the end of a linked list and dequeue is deleting the first node in the list. Then, the algorithm for them looks like the following.

```
void enqueue (int elem)
{
 Node *node = new Node;
 node -> data = elem;
 node -> next = NULL; // node will be new tail node
 if(size = = 0)
   head = node; // special case of a previously empty queue
 else
   tail -> next = node; // add node at the tail of the list
 tail = node; // update the reference to the tail node
 size + +;
}
```

```
int dequeue ( )
{
  if (size = = 0)
    cout<<"Queue is empty";
  else
   {
     tmp = head -> data;
     head = head -> next;
     size - - ;
     if (size = = 0)
       tail = NULL; // the queue is now empty
     return tmp;
   }
}
```

Each of the methods of the singly linked list implementation of the queue ADT runs in O(1) time. We also avoid the need to specify a maximum size for the queue, as was done in the array-based queue implementation, but this benefit comes at the expense of increasing the amount of space used per element. Still, the methods in the singly linked list queue implementation are more complicated than we might like, for we must take extra care in how we deal with special cases where the queue is empty before an enqueue or where the queue becomes empty after a dequeue.

5.4. Deque (Double - ended Queue)

Consider now a queue-like data structure that supports insertion and deletion at both the front and the rear of the queue. Insertion and deletion can occur at either end. Such an extension of a queue is called a double-ended queue, or deque, which is usually pronounced "deck" to avoid confusion with the dequeue function of the regular queue ADT, which is pronounced like the abbreviation "D.Q." An easy way to remember the "deck" pronunciation is to observe that a deque is like a deck of cards in the hands of a crooked card dealer - it is possible to deal off both the top and the bottom.

The fundamental functions of the deque ADT are stated as follows.

- **EnqueueFront** (elem): Insert a new element *elem* at the beginning of the deque.
- **EnqueueRear** (elem): Insert a new element *elem* at the end of the deque.
- **DequeueFront** ( ): Remove the first element of the deque; an error occurs if the deque is empty.
- **DequeueRear** ( ): Remove the last element of the deque; an error occurs if the deque is empty.

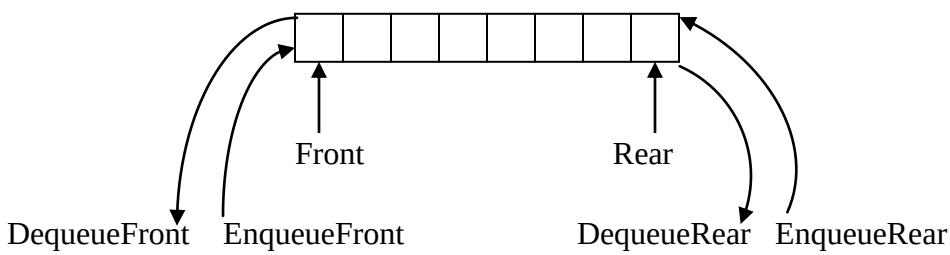
Additionally, the deque includes the following support functions:

- **Front** ( ): Return the first element of the deque; an error occurs if the deque is empty.
- **Rear** ( ): Return the last element of the deque; an error occurs if the deque is empty.
- **Size** ( ): Return the number of elements of the deque.
- **IsEmpty** ( ): Return true if the deque is empty and false otherwise.

The following table shows a series of operations and their effects on an initially empty deque of integers.

Operations	Output	Deque
EnqueueFront (3)	-	(3)
EnqueueFront (5)	-	(5, 3)
Front ( )	5	(5, 3)
DequeueFront ( )	-	(3)
EnqueueRear (7)	-	(3, 7)
Size ( )	2	(3, 7)
Rear ( )	7	(3, 7)
DequeueFront ( )	-	(7)
IsEmpty ( )	False	(7)
DequeueRear ( )	-	( )

A deque can be seen in the following figure.



As like that of queue, a deque can be implement both using array and linked list. Implementing using array may require shifting (for example inserting at front, when we don't have free spaces at front part), and this shifting requires cost. The best thing to implement a deque is using doubly linked list.

5.5. Priority Queue

In many situations, simple queues are inadequate, because first in/first out scheduling has to be overruled using some priority criteria. In a post office example, a handicapped person may have priority over others. Therefore, when a clerk is available, a handicapped person is served instead of someone from the front of the queue.

Similarly, in a multiuser environment, the operating system scheduler must decide which of several processes to run. Generally, a process is allowed to run only for a fixed period of time. One algorithm uses a queue. Jobs are initially placed at the end of the queue. The scheduler will repeatedly take the first job on the queue, run it until either it finishes or its time limit is up, and place it at the end of the queue if it does not finish. This strategy is generally not appropriate, because very short jobs will seem to take a long time because of the wait involved to run. Generally, it is important that short jobs finish as fast as possible, so these jobs should have precedence over jobs that have already been running. Furthermore, some jobs that are not short are still very important and should also have precedence.

This particular application seems to require a special kind of queue, known as a priority queue.

A priority queue is an abstract data type for storing a collection of prioritized elements that supports arbitrary element insertion but supports removal of elements in order of priority, that is, the element with first priority can be removed at any time. This ADT is fundamentally different from the position-based data structures such as stacks, queues, deques and lists. These other data structures store elements at specific positions, which are often positions in a linear arrangement of the elements determined by the insertion and deletion operations performed. The priority queue ADT stores elements according to their priorities, and has no external notion of "position."

In general, a situation in which items are queued not just by time of receipt, but on the basis of a value or priority is called a priority queue. In priority queues, elements are dequeued according to their priority and their current queue position.

Having described the priority queue abstract data type at an intuitive level, we now describe it in more detail. As an ADT, a priority queue *P* supports the following functions:

- **Size** ( ): Return the number of elements in *P*.
- **Isempty** ( ): Return true if *P* is empty and false otherwise.
- **Min** ( ): Return a reference to an element of P with the smallest associated key value (but do not remove it); an error condition occurs if the priority queue is empty.

- **Insert** (elem): Insert a new element *elem* into *P*.
- **RemoveMin** ( ): Remove from *P* the element referenced by *min* ( ); an error condition occurs if the priority queue is empty.

As mentioned above, the primary functions of the priority queue ADT are the *insert*, *min*, and *removeMin* operations. The other functions, *size* and *isempty*, are generic collection operations. Note that we allow a priority queue to have multiple entries with the same key.

The following table shows a series of operations and their effects on an initially empty priority queue P. Each element consists of an integer, which we assume to be sorted according to the natural ordering of the integers. Note that each call to min returns a reference to an entry in the queue, not the actual value. Although the "Priority Queue" column shows the items in sorted order, the priority queue need not store elements in this order.

Operation	Output	Priority Queue
Insert (5)	-	{ 5 }
Insert (9)	-	{ 5, 9 }
Insert (2)	-	{ 2, 5, 9 }
Insert (7)	-	{ 2, 5, 7, 9 }
Min ( )	[2]	{ 2, 5, 7, 9 }
RemoveMin ( )	-	{ 5, 7, 9 }
Size ( )	3	{ 5, 7, 9 }
Min ( )	[5]	{ 5, 7, 9 }
RemoveMin ( )	-	{ 7, 9 }
RemoveMin ( )	-	{ 9 }
RemoveMin ( )	-	{ }
IsEmpty ( )	True	{ }
RemoveMin ( )	" Error"	{ }

Example: Consider the following queue of persons where females have higher priority than males (gender is the key to give priority).

Abebe	Alemu	Aster	Belay	Kedir	Meron	Yonas
Male	Male	Female	Male	Male	Female	Male

Dequeue ( ) - deletes Aster

Abebe	Alemu	Belay	Kedir	Meron	Yonas
Male	Male	Male	Male	Female	Male

Dequeue ( ) - deletes Meron

Abebe	Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male	Male

Now the queue has data having equal priority and dequeue operation deletes the front element like in the case of ordinary queues.

Dequeue ( ) - deletes Abebe

Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male

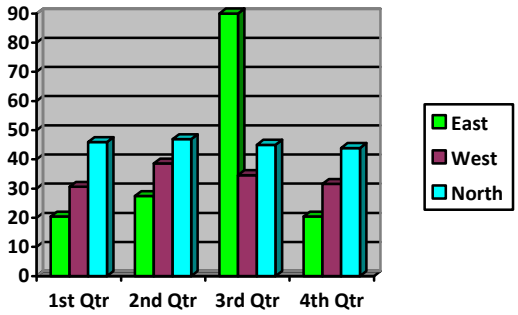
Dequeue ( ) - deletes Alemu

Belay	Kedir	Yonas
Male	Male	Male

5.6. Applications of Queue

There are many algorithms that use queues to give efficient running times. For now, we will give some simple examples of queue usage.

- When jobs are submitted to a printer, they are arranged in order of arrival. Thus, essentially, jobs sent to a printer are placed on a queue.
- Virtually every real-life line is (supposed to be) a queue. For instance, lines at ticket counters are queues, because service is first-come first-served.
- Another example concerns computer networks. There are many network setups of personal computers in which the disk is attached to one machine, known as the file server. Users on other machines are given access to files on a first-come first-served basis, so the data structure is a queue.
- In universities, where resources are limited, students must sign a waiting list if all computers are occupied. The student who has been at a computer the longest is forced off first, and the student who has been waiting the longest is the next user to be allowed on.
- Task scheduler in multiprocessing system - maintains priority queues of processes.



- Telephone calls in a busy en vironment - maintains a queue of telephone calls.

CHAPTER SIX
6. TREES

6.1. Introduction

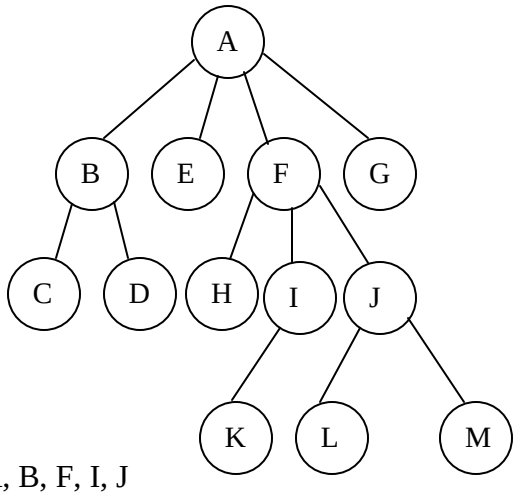
A tree is an abstract data type that stores elements hierarchically. With the exception of the top element, each element in a tree has a parent element and zero or more children elements. A tree is usually visualized by placing elements inside ovals, circles or rectangles, and by drawing the connections between parents and children with straight lines. We typically call the top element the root of the tree, but it is drawn as the highest element, with the other elements being connected below (just the opposite of a botanical tree).

Formally, we define tree T to be a set of nodes storing elements in a parent-child relationship with the following properties: (1) If T is non-empty, it has a special node, called the root of T, that has no parent. (2) Each node v of T different from the root has a unique parent node w; every node with parent w is a child of w.

Note that according to our definition, a tree can be empty, meaning that it doesn't have any nodes. This convention also allows us to define a tree recursively, such that a tree T is either empty or consists of a node r, called the root of T , and a (possibly empty) set of trees whose roots are the children of r. In a rooted tree, there is a unique path from the root to each node.

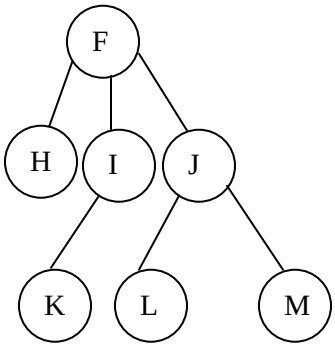
6.2. Tree Terminologies

Consider the following tree.

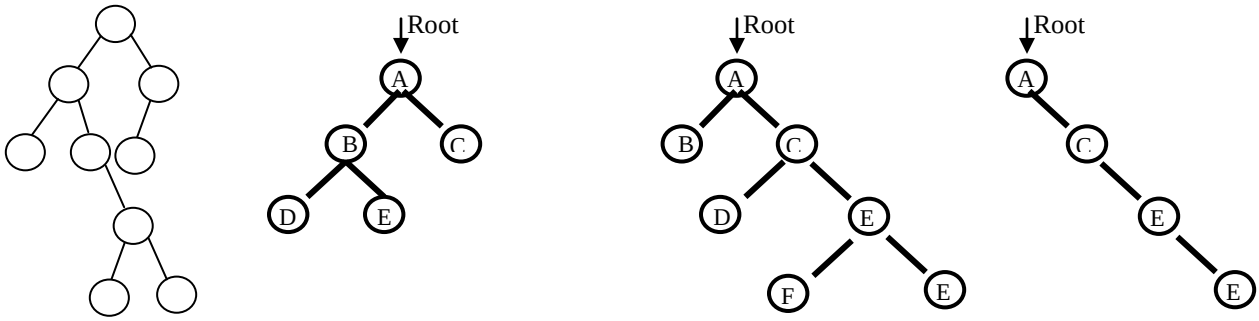


- **Root node**: a node without a parent. → A
- **Internal node**: a node with at least one child. →A, B, F, I, J
- **External (leaf) node**: a node without a child. → C, D, E, H, K, L, M, G
- **Ancestors of a node**: parent, grandparent, grand-grandparent, etc of a node.
  Ancestors of K → A, F, I
- **Descendants of a node**: children, grandchildren, grand-grandchildren etc of a node.
  Descendants of F → H, I, J, K, L, M
- **Depth of a node**: number of ancestors or length of the path from the root to the node.
  Depth of H → 2
- **Height of a tree**: depth of the deepest node. → 3
- **Siblings**: nodes with the same parent → e.g. H, I, J

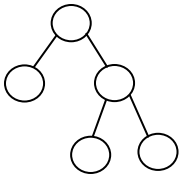
- ✓ **Subtree**: a tree consisting of a node and its descendants.



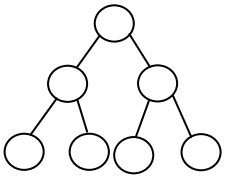
- ✓ **Binary tree**: a tree in which each node has at most two children called left child and right child.



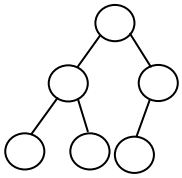
- ✓ **Full binary tree**: a binary tree where each node has either 0 or 2 children.



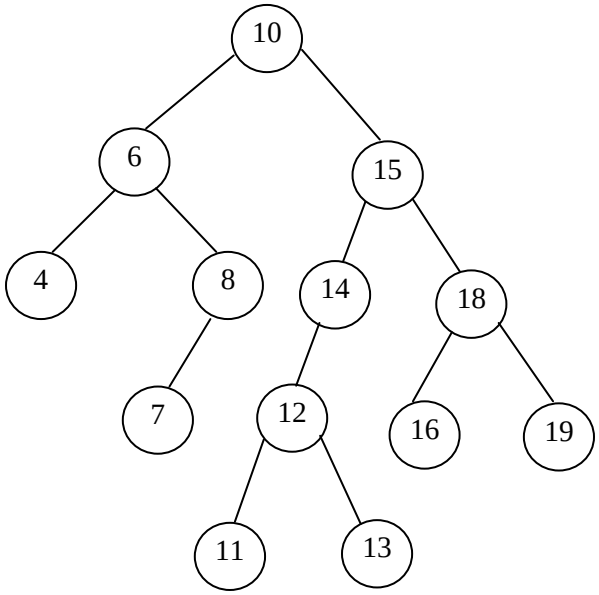
- ✓ **Balanced binary tree**: a binary tree where each node except the leaf nodes has left and right children and all the leaves are at the same level.



- ✓ **Complete binary tree**: a binary tree in which the length from the root to any leaf node is either h or h-1 where h is the height of the tree. The deepest level should also be filled from left to right.



- ✓ **Binary search tree (ordered binary tree)**: a binary tree that may be empty, but if it is not empty it satisfies the following.
  - Every node has a key and no two elements have the same key.
  - The keys in the right subtree are larger than the keys in the root.
  - The keys in the left subtree are smaller than the keys in the root.
  - The left and the right sub trees are also binary search trees.



6.3. Data Structure of a Binary Tree

```
struct DataModel
  {
   Declaration of data fields
   DataModel * Left, *Right;
  };
DataModel *RootDataModelPtr = NULL;
```

6.4. Operations on Binary Search Tree

Consider the following definition of binary search tree.

```
struct Node
 {
   int Num;
   Node * Left, *Right;
 };
Node *RootNodePtr = NULL;
```

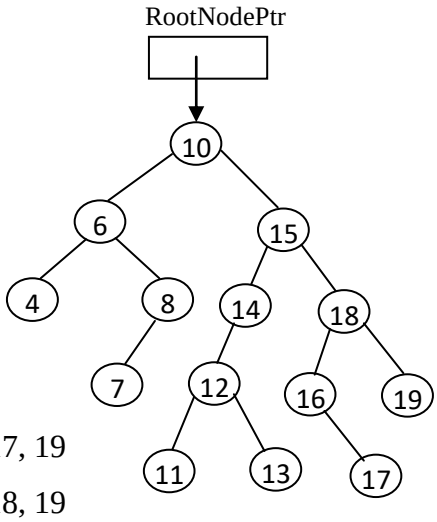
i. Traversing

Tree traversal is the process of visiting each node only one time. Traversal does not specify the order in which the nodes are visited. Hence, there are as many tree traversals as there are permutations of nodes; for a tree with n nodes, there are ***n!*** different traversals.

Binary search tree can be traversed in three ways.

a. Pre order traversal - traversing binary tree in the order of *parent*, *left* and *right*.
b. Inorder traversal - traversing binary tree in the order of *left*, *parent* and *right*.
c. Postorder traversal - traversing binary tree in the order of *left*, *right* and *parent*.

Example:



Preorder traversal - 10, 6, 4, 8, 7, 15, 14, 12, 11, 13, 18, 16, 17, 19

Inorder traversal - 4, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

==> Used to display nodes in ascending order.

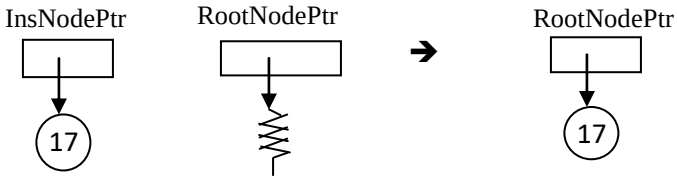
Postorder traversal - 4, 7, 8, 6, 11, 13, 12, 14, 17, 16, 19, 18, 15, 10

ii. Insertion

When a node is inserted, the definition of binary search tree should be preserved. Suppose there is a binary search tree whose root node is pointed by RootNodePtr and we want to insert a node (that stores 17) pointed by InsNodePtr.

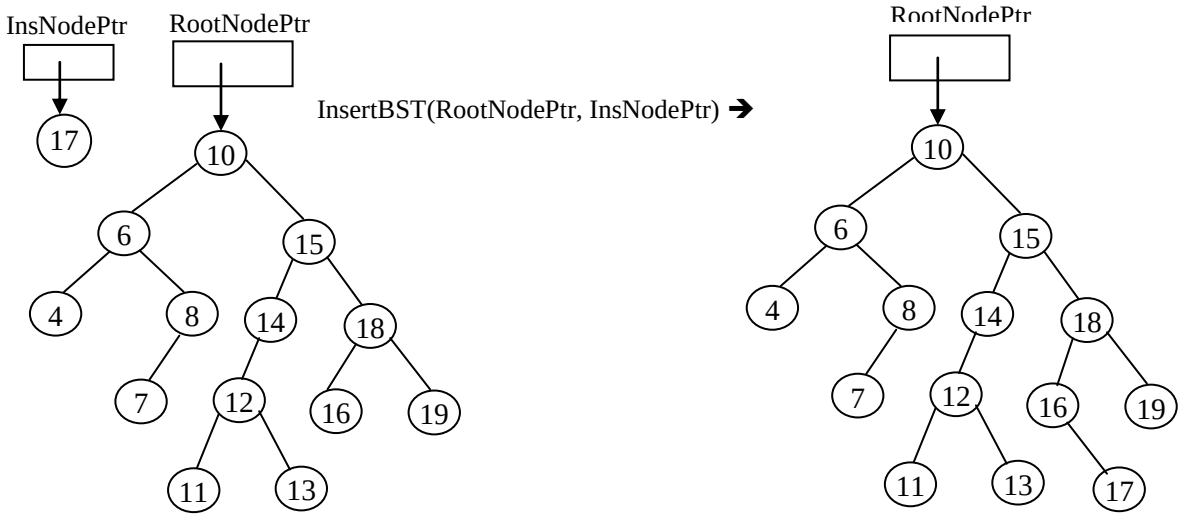
Case 1: - There is no data in the tree (i.e. RootNodePtr is NULL).

-The node pointed by InsNodePtr should be made the root node.



Case 2: There is data

- ✓ Search the appropriate position.
- ✓ Insert the node in that position.



- **Function call:**

```
if(RootNodePtr = = NULL)
        RootNodePtr=InsNodePtr;
else
        InsertBST(RootNodePtr, InsNodePtr);
```

- **Implementation**

```
void InsertBST(Node *RNP, Node *INP)
  {
     //RNP = RootNodePtr and INP = InsNodePtr
     int Inserted = 0;
     while(Inserted = =0)
       {
         if(RNP->Num > INP->Num)
             {
               if(RNP->Left = = NULL)
                 {   RNP->Left = INP;
                     Inserted = 1;
                 }
               else
                   RNP = RNP->Left;
             }
           else
             {
               if(RNP->Right = = NULL)
                 {   RNP->Right = INP;
                     Inserted = 1;
                 }
               else
                   RNP = RNP->Right;
             }
         }
  }
```

A recursive version of the function can also be given as follows.

```
void InsertBST(Node *RNP, Node *INP)
  {
    if(RNP->Num > INP->Num)
     {
      if(RNP->Left = = NULL)
         RNP->Left = INP;
      else
         InsertBST(RNP->Left, INP);
     }
    else
     {
      if(RNP->Right = = NULL)
         RNP->Right = INP;
      else
         InsertBST(RNP->Right, INP);
     }
  }
```

iii. Searching

Searching is locating an element in a tree. For every node, compare the element to be located with the value stored in the node currently pointed at. If the element is less than the value, go to the left subtree and try again. If it is greater than that value, try the right subtree. If it is the same, obviously the search can be discontinued. The search is also aborted if there is no way to go, indicating that the element is not in the tree.

To search a node (whose Num value is Number) in a binary search tree (whose root node is pointed by RootNodePtr), one of the three traversal methods can be used.

- **Function call**:

  ElementExists = SearchBST (RootNodePtr, Number);
  // ElementExists is a Boolean variable defined as: *bool ElementExists = false*;

- **Implementation**:

```
bool SearchBST (Node *RNP, int x)
  {
    if(RNP = = NULL)
        return(false);
    else if(RNP->Num = = x)
        return(true);
    else if(RNP->Num > x)
        return(SearchBST(RNP->Left, x));
    else
        return(SearchBST(RNP->Right, x));
  }
```

When we search an element in a binary search tree, sometimes it may be necessary for the SearchBST function to return a pointer that points to the node containing the element searched. Accordingly, the function has to be modified as follows.

- ✓ **Function call:**

```
SearchedNodePtr = SearchBST (RootNodePtr, Number);
// SearchedNodePtr is a pointer variable defined as: Node *SearchedNodePtr=NULL;
```

- ✓ **Implementation:**

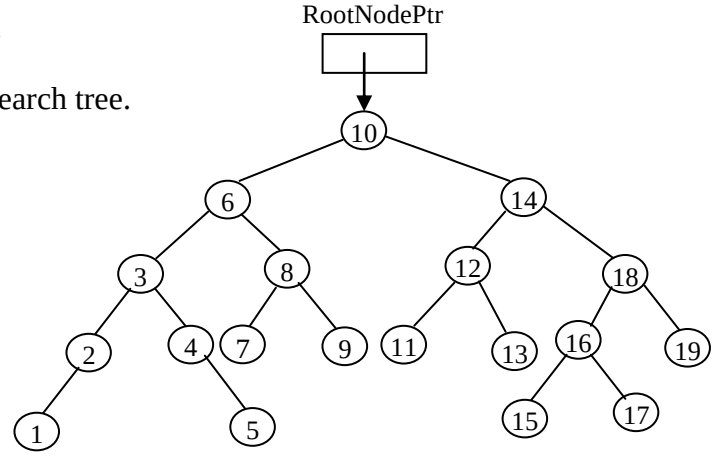
```
Node *SearchBST (Node *RNP, int x)
 {
   if((RNP = = NULL) || (RNP->Num = = x))
     return(RNP);
   else if(RNP->Num > x)
     return(SearchBST(RNP->Left, x));
   else
     return(SearchBST (RNP->Right, x));
 }
```

iv. Deletion

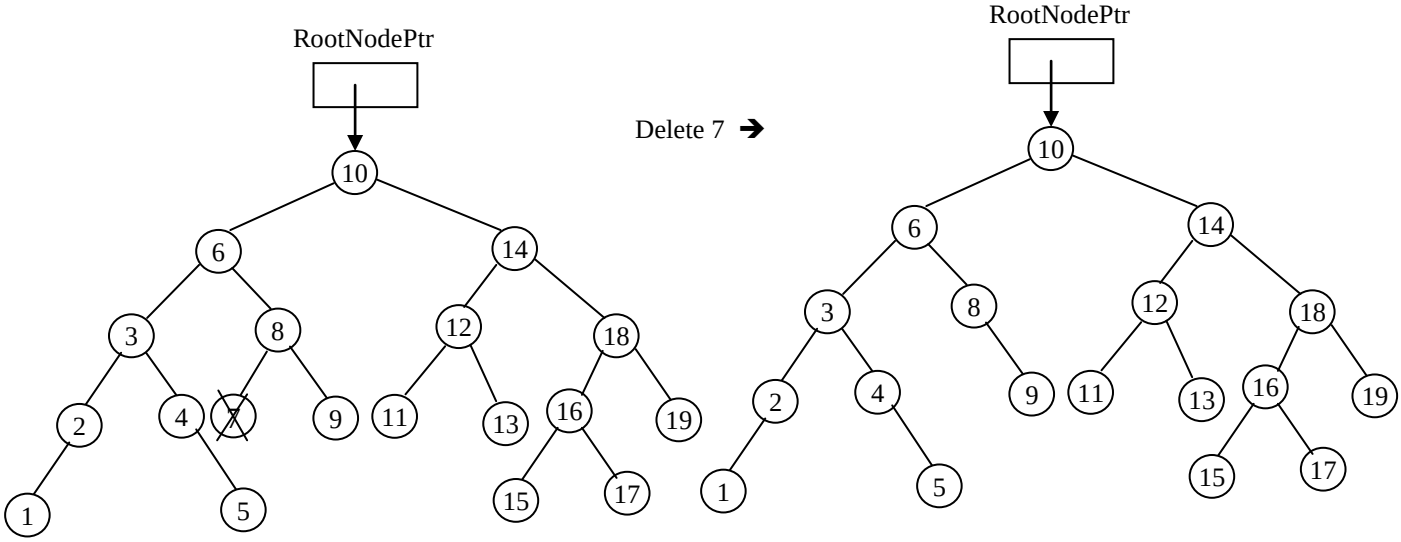
Deleting a node is another operation necessary to maintain a binary search tree. The level of complexity in performing the operation depends on the position of the node to be deleted in the tree. It is by far more difficult to delete a node having two subtrees than to delete a leaf; the complexity of the deletion algorithm is proportional to the number of children the node has.

To delete a node (whose Num value is N) from binary search tree (whose root node is pointed by RootNodePtr), four cases should be considered. When a node is deleted the definition of binary search tree should be preserved.

Consider the following binary search tree.



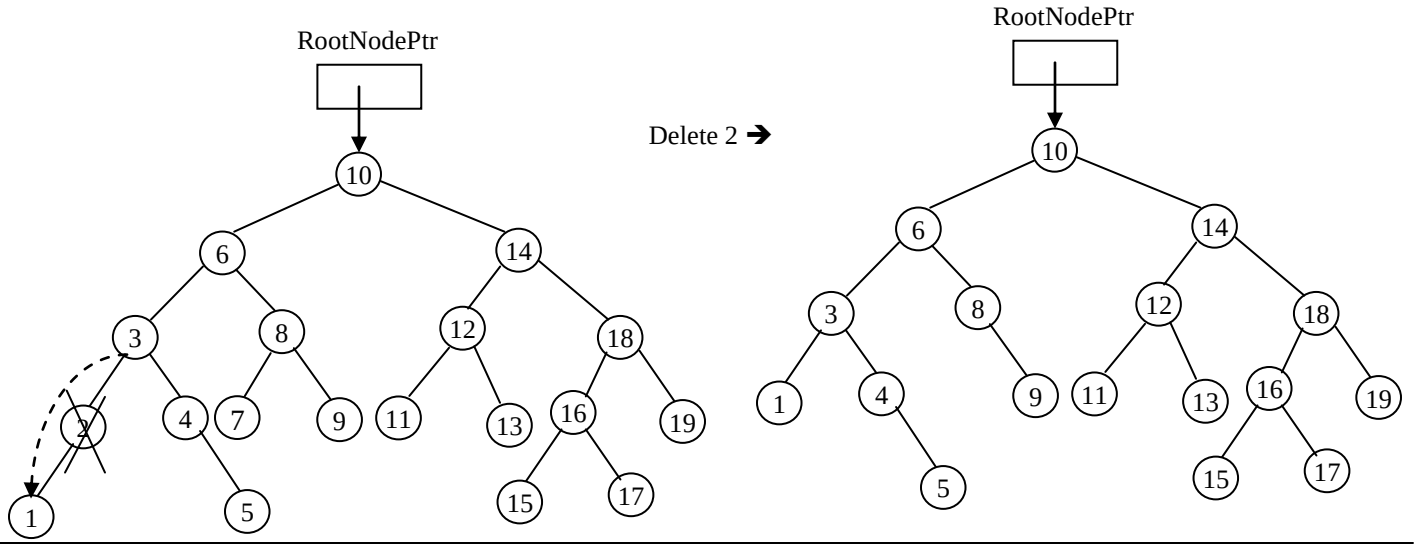
❖ **Case 1**: Deleting a leaf node (a node having no child), e.g. 7



❖ **Case 2**: Deleting a node having only one child, e.g. 2

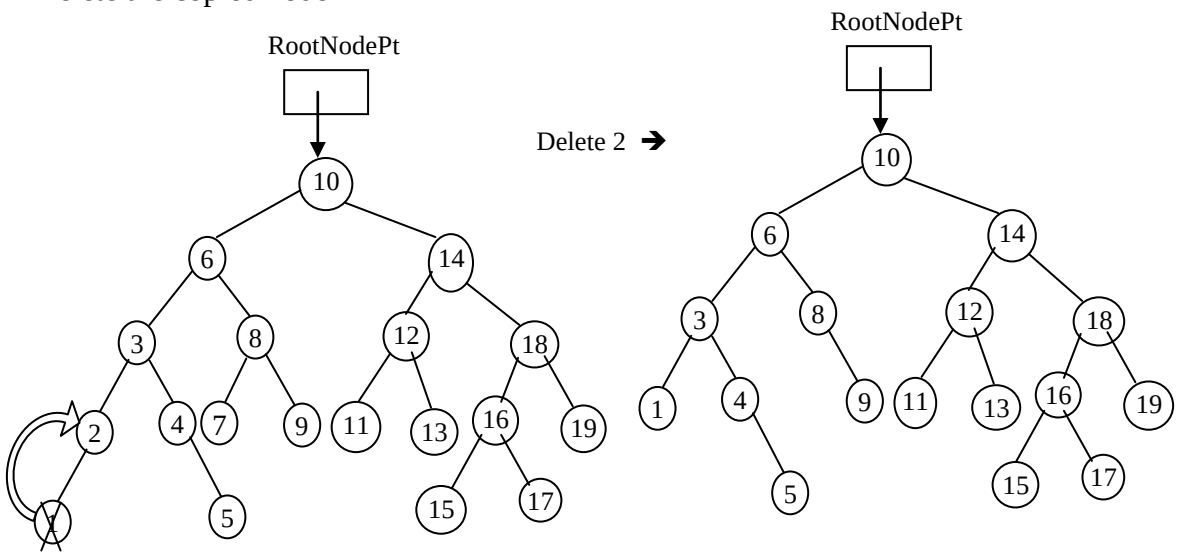
<u>Approach 1</u>: Deletion by merging – one of the following is done

- If the deleted node is the left child of its parent and the deleted node has only the left child, the left child of the deleted node is made the left child of the parent of the deleted node.
- If the deleted node is the left child of its parent and the deleted node has only the right child, the right child of the deleted node is made the left child of the parent of the deleted node.
- If the deleted node is the right child of its parent and the node to be deleted has only the left child, the left child of the deleted node is made the right child of the parent of the deleted node.
- If the deleted node is the right child of its parent and the deleted node has only the right child, the right child of the deleted node is made the right child of the parent of the deleted node.



Approach 2: Deletion by copying- the following is done

- Copy the node containing the largest element in the left (or the smallest element in the right) to the node containing the element to be deleted.
- Delete the copied node



❖ **Case 3**: Deleting a node having two children, e.g. 6

Approach 1: Deletion by merging – one of the following is done

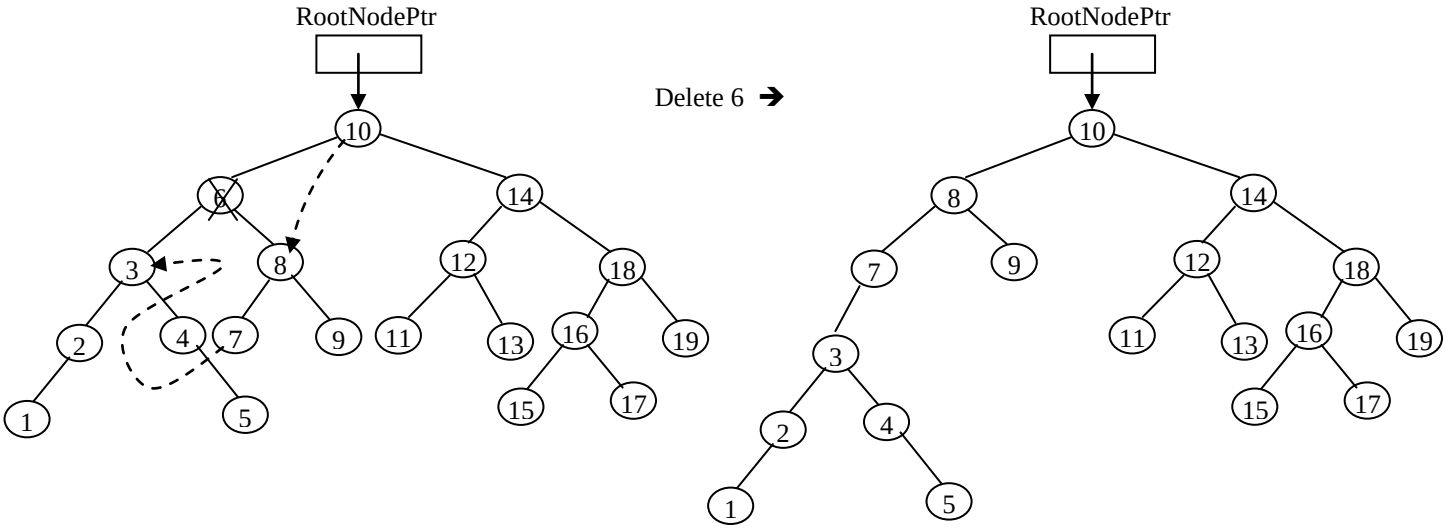
- If the deleted node is the left child of its parent, one of the following is done
  - The left child of the deleted node is made the left child of the parent of the deleted node, and
  - The right child of the deleted node is made the right child of the node containing largest element in the left of the deleted node

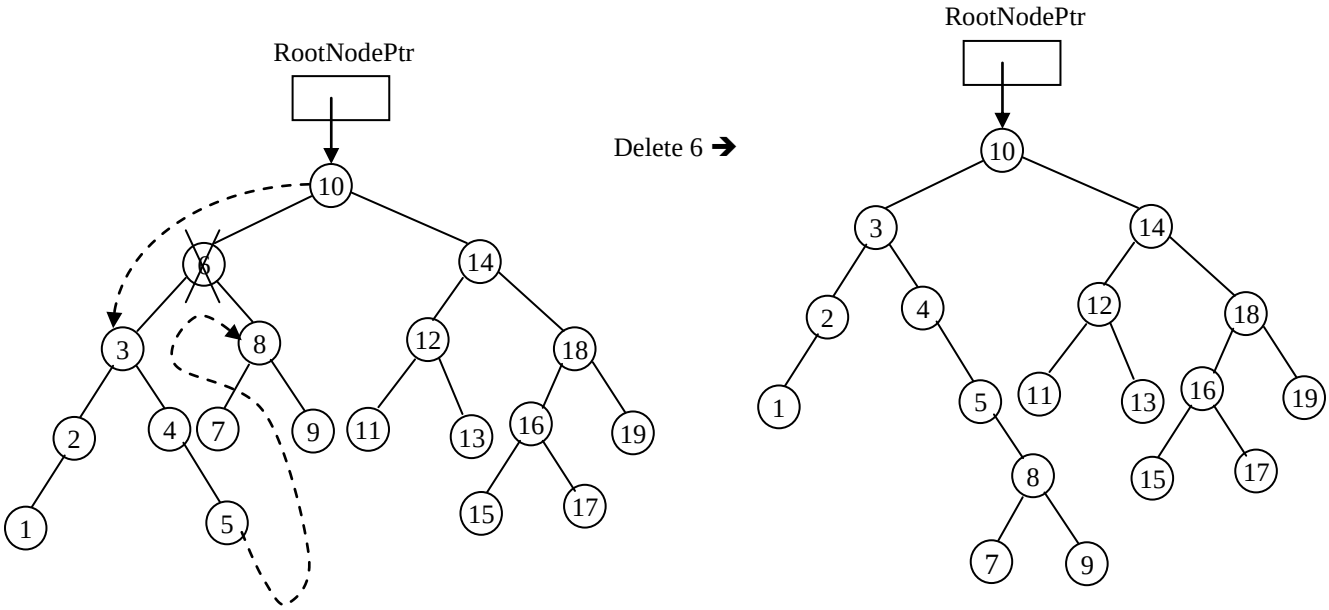
  OR

  - The right child of the deleted node is made the left child of the parent of the deleted node, and
  - The left child of the deleted node is made the left child of the node containing smallest element in the right of the deleted node
- If the deleted node is the right child of its parent, one of the following is done
  - The left child of the deleted node is made the right child of the parent of the deleted node, and
  - The right child of the deleted node is made the right child of the node containing largest element in the left of the deleted node

  OR

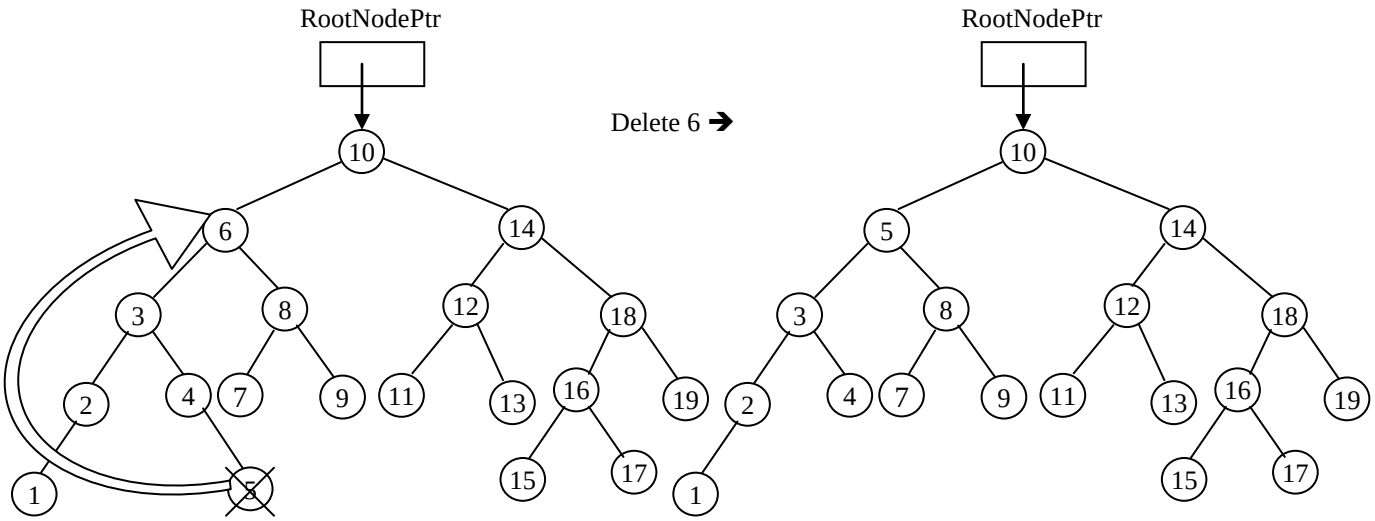
  - The right child of the deleted node is made the right child of the parent of the deleted node, and
  - The left child of the deleted node is made the left child of the node containing smallest element in the right of the deleted node.

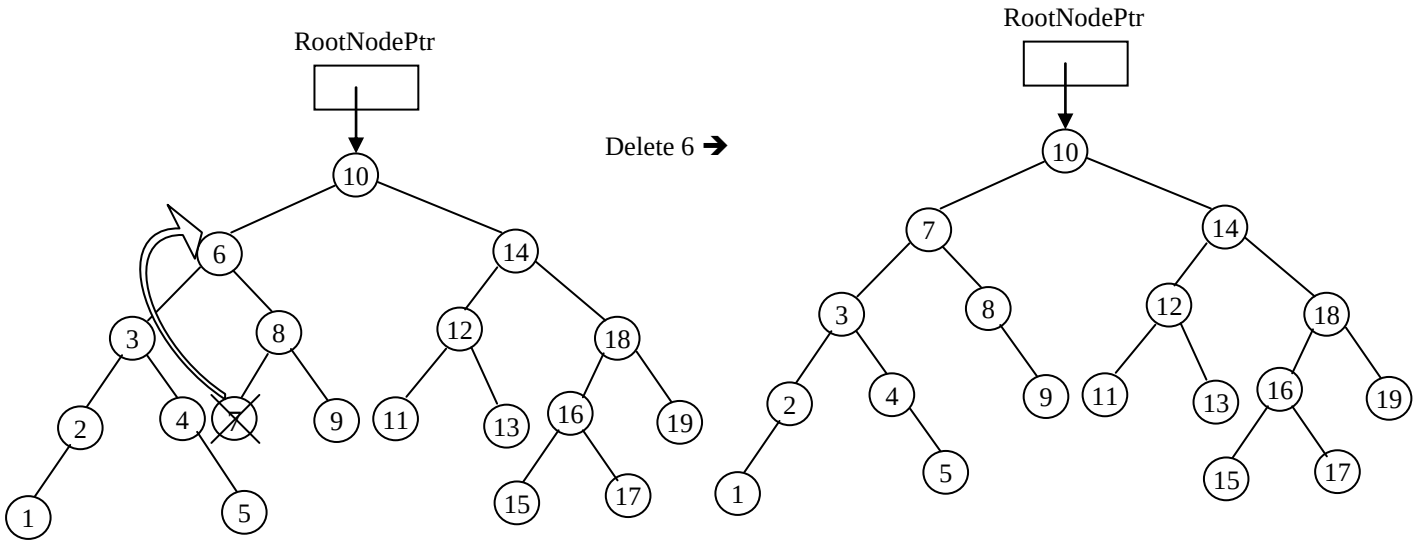




<u>Approach 2</u>: Deletion by copying- the following is done

- Copy the node containing the largest element in the left (or the smallest element in the right) to the node containing the element to be deleted
- Delete the copied node

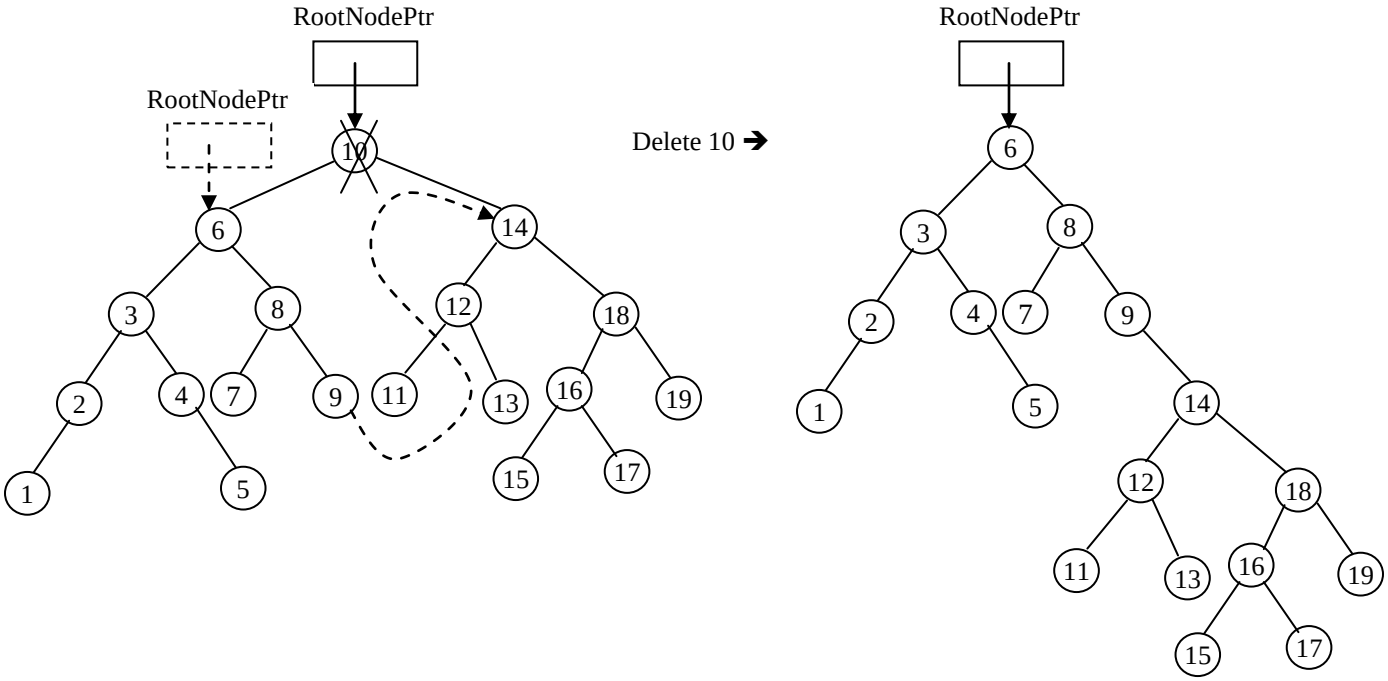


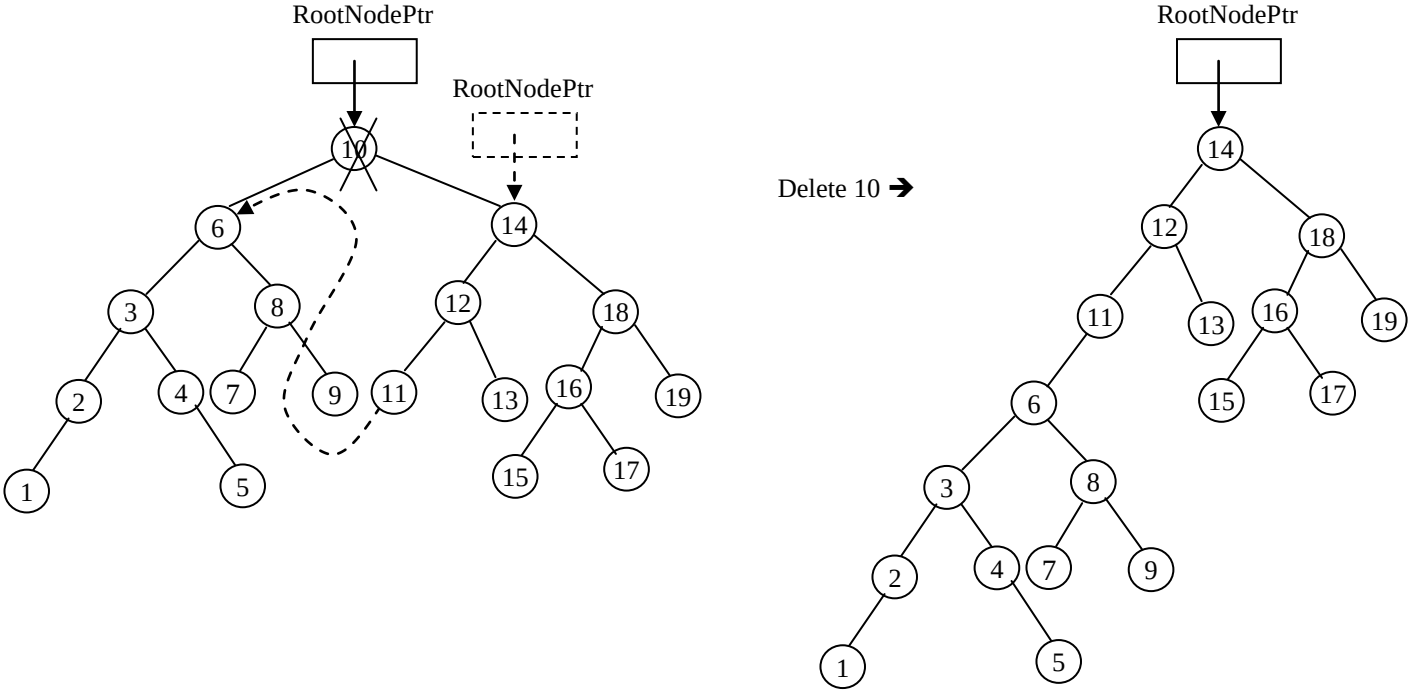


❖ **Case 4**: Deleting the root node, 10

<u>Approach 1</u>: Deletion by merging- one of the following is done

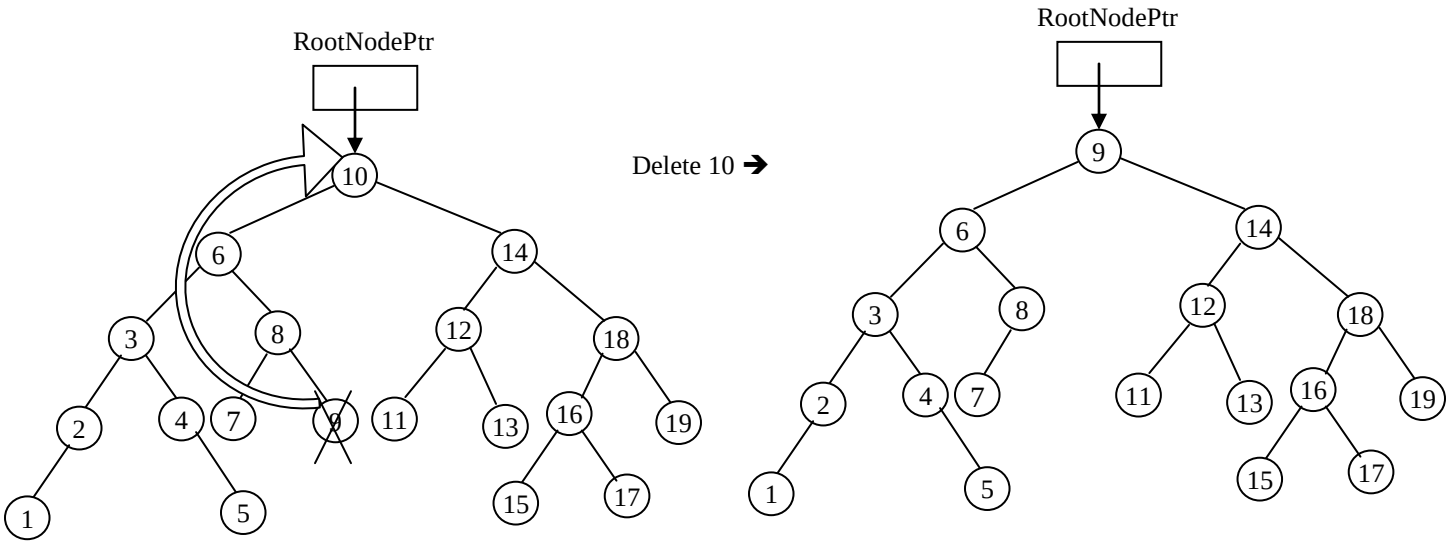
- If the tree has only one node the root node pointer is made to point to nothing (NULL)
- If the root node has left child
  - the root node pointer is made to point to the left child
  - the right child of the root node is made the right child of the node containing the largest element in the left of the root node
- If root node has right child
  - the root node pointer is made to point to the right child
  - the left child of the root node is made the left child of the node containing the smallest element in the right of the root node

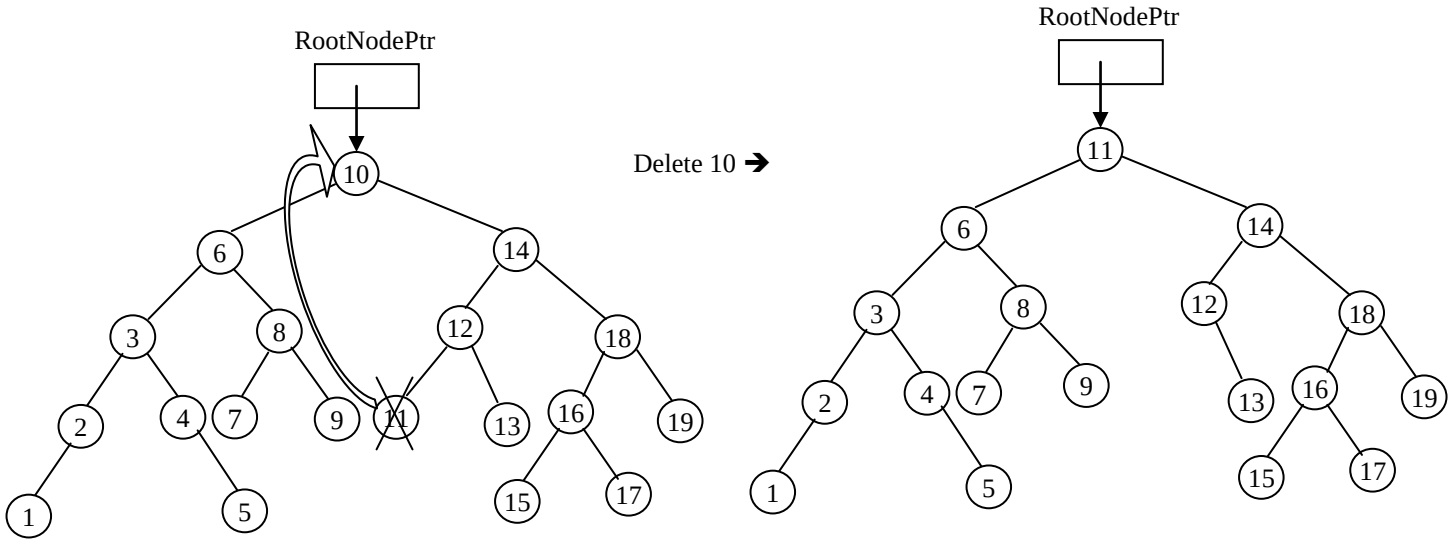




<u>Approach 2</u>: Deletion by copying- the following is done

- Copy the node containing the largest element in the left (or the smallest element in the right) to the node containing the element to be deleted
- Delete the copied node





- **Function call:**

```
if ((RootNodePtr->Left = = NULL)&&( RootNodePtr->Right = = NULL) && (RootNodePtr->Num = = N))
  {   // the node to be deleted is the root node having no child
      RootNodePtr = NULL;
      delete RootNodePtr;
  }
else
  DeleteBST(RootNodePtr, RootNodePtr, N);
```

- **Implementation:** (Deletion by copying)

```
void DeleteBST(Node *RNP, Node *PDNP, int x)
{
    Node *DNP; // a pointer that points to the currently deleted node
    // PDNP is a pointer that points to the parent node of currently deleted node
    if(RNP = = NULL)
        cout<<"Data not found\n";
    else if (RNP->Num > x)
        DeleteBST(RNP->Left, RNP, x);// delete the element in the left subtree
    else if(RNP->Num < x)
        DeleteBST(RNP->Right, RNP, x);// delete the element in the right subtree
    else
    {
      DNP = RNP;
      if((DNP->Left = = NULL) && (DNP->Right = = NULL))
       {
        if (PDNP->Left = = DNP)
          PDNP->Left = NULL;
        else
          PDNP->Right = NULL;
        delete DNP;
       }
      else
       {
         if(DNP->Left ! = NULL) //find the maximum in the left
          {
            PDNP = DNP;
            DNP = DNP->Left;
            while(DNP->Right ! = NULL)
             {
               PDNP = DNP;
               DNP = DNP->Right;
             }
            RNP->Num = DNP->Num;
            DeleteBST(DNP,PDNP,DNP->Num);
          }
         else //find the minimum in the right
           {
             PDNP = DNP;
             DNP = DNP->Right;
             while(DNP->Left ! = NULL)
              {
                PDNP = DNP;
                DNP = DNP->Left;
              }
             RNP->Num = DNP->Num;
             DeleteBST(DNP, PDNP, DNP->Num);
           }
       }
    }
}
```

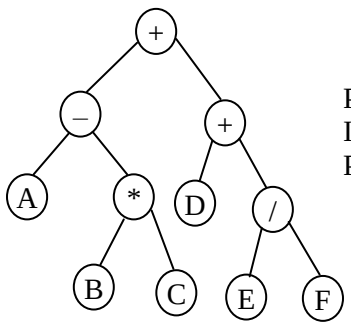
6.5. Application of binary trees

I) For tree traversal

- Store values on leaf nodes and operators on internal nodes

- Preorder traversal: used to generate mathematical expression in prefix notation.
- Inorder traversal: used to generate mathematical expression in infix notation.
- Postorder traversal: used to generate mathematical expression in postfix notation.

Example:



Preorder traversal - + – A * B C + D / E F → Prefix notation
Inorder traversal - A – B * C + D + E / F → Infix notation
Postorder traversal - A B C * – D E F / + + → Postfix notation

- **Function calls:**

```
Preorder(RootNodePtr);
Inorder(RootNodePtr);
Postorder(RootNodePtr);
```

- **Implementation**:

```
void Preorder (Node *CurrNodePtr)
  {
    if(CurrNodePtr ! = NULL)
     {
       cout<< CurrNodePtr->Num;              // or any operation on the node
       Preorder(CurrNodePtr->Left);
       Preorder(CurrNodePtr->Right);
     }
  }
```

```
void Inorder (Node *CurrNodePtr)
 {
   if(CurrNodePtr ! = NULL)
    {
      Inorder(CurrNodePtr->Left);
      cout<< CurrNodePtr->Num;            // or any operation on the node
      Inorder(CurrNodePtr->Right);
    }
 }

void Postorder (Node *CurrNodePtr)
 {
   if(CurrNodePtr ! = NULL)
    {
      Postorder(CurrNodePtr->Left);
      Postorder(CurrNodePtr->Right);
      cout<< CurrNodePtr->Num;            // or any operation on the node
    }
 }
```

II) For constructing an expression tree

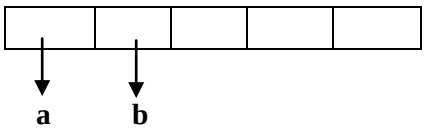
We can convert a postfix expression into an expression tree. Since we already have an algorithm to convert infix and prefix to postfix, we can generate expression trees from any types of input expression. This method strongly resembles the postfix evaluation algorithm.

Algorithm:

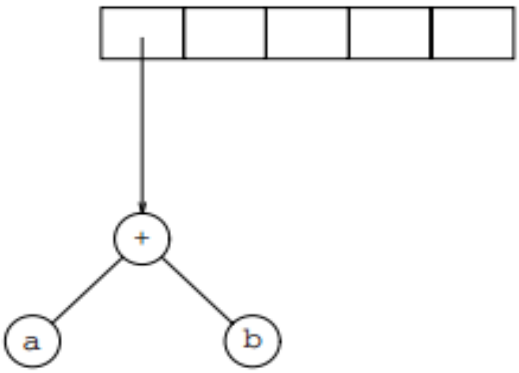
1. If the expression is not postfix, then convert the expression in to postfix.
2. Read the expression one symbol at a time.
3. If the symbol is an operand, we create a one node tree and push a pointer to it onto a stack.
4. If the symbol is an operator, we pop pointers to two trees T_1 and T_2 from the stack and form a new tree whose root is the operator and whose left and right children point to T1 and T2 respectively. A pointer to this new tree is then pushed onto the stack.

Ex. create an expression tree for a b + c d e + * *

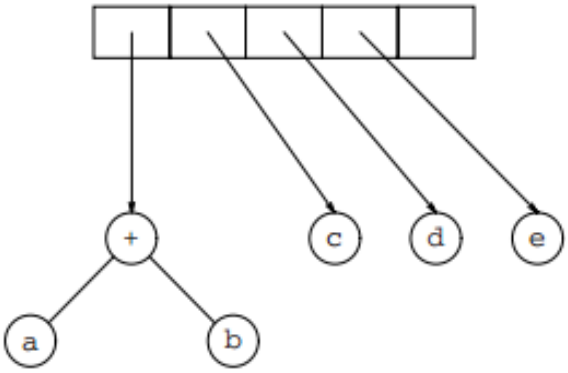
The first two symbols are operands, so we create one node trees and push pointers to them onto a stack.



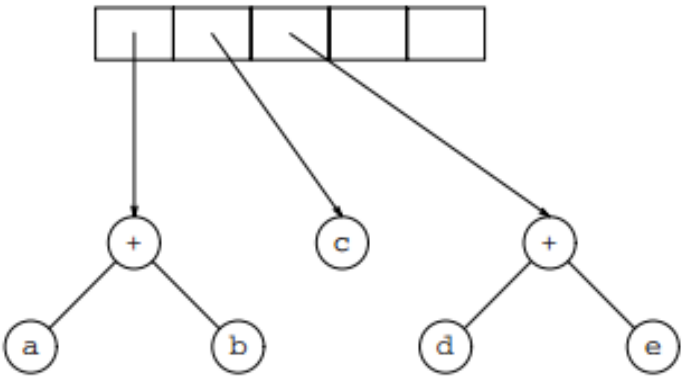
Next, a + is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



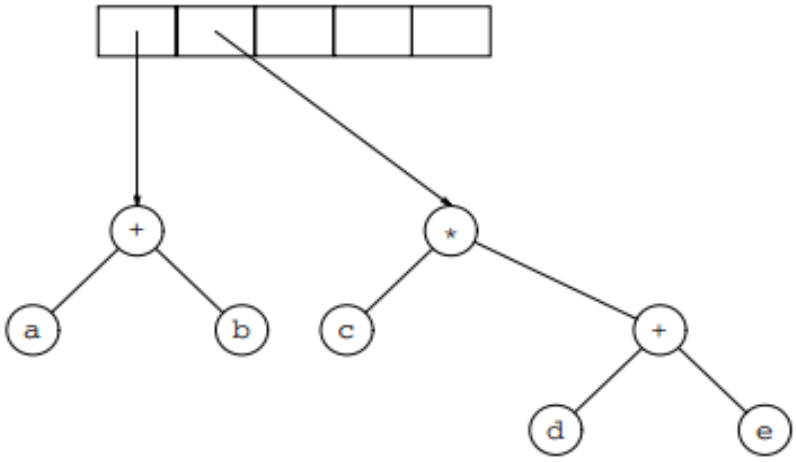
Next, **c**, **d**, and **e** are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



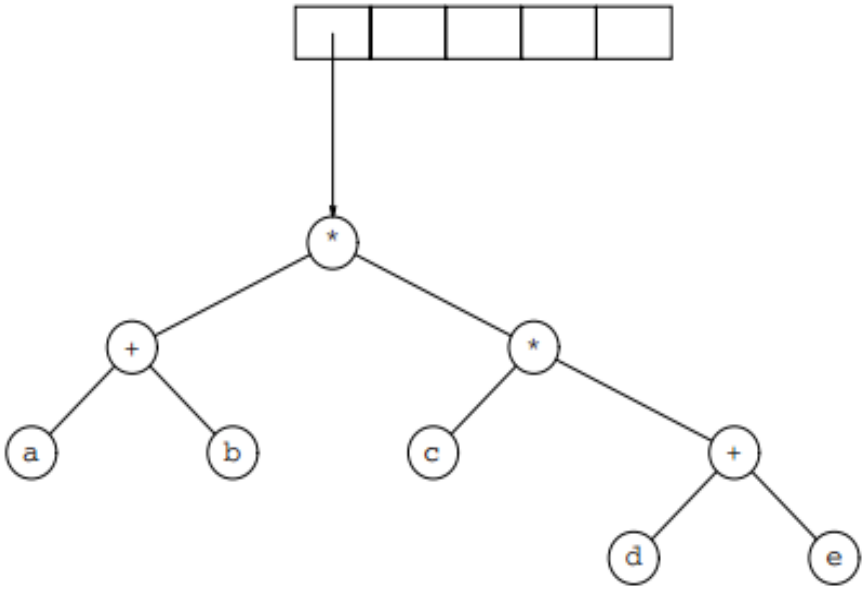
Now a + is read, so two trees are merged. Pop two pointers and create a new subtree by merging with + and push it on to the stack



Continuing, a ***** is read, so we pop two tree pointers and form a new tree with a ***** as root. Pop two pointers and create a new subtree by merging with * and push it on to the stack



Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack. Pop two pointers and create a new tree by merging with ***** and push it on to the stack. Thus we are having an expression tree.

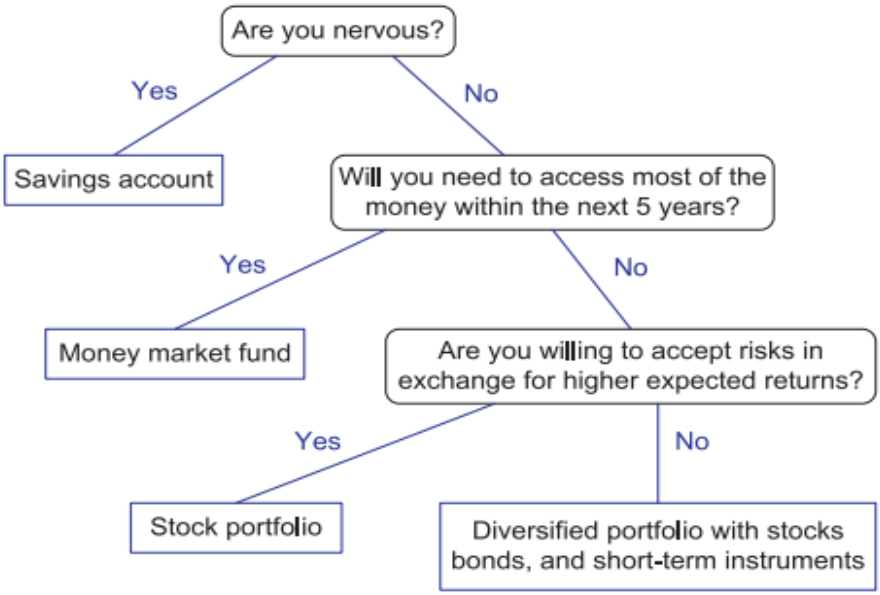


III) Some other applications of trees

An important class of binary trees arises in contexts where we wish to represent a number of different outcomes that can result from answering a series of yes-or-no questions. Each internal node is associated with a question. Starting at the root, we go to the left or right child of the current node, depending on whether the answer to the question is “Yes” or “No.” With each decision, we follow an edge from a parent to a child, eventually tracing a path in the tree from

the root to an external node. Such binary trees are known as decision trees, because each external node p in such a tree represents a decision of what to do if the questions associated with p's ancestors are answered in a way that leads to p.

The following figure illustrates a decision tree that provides recommendations to a prospective investor.

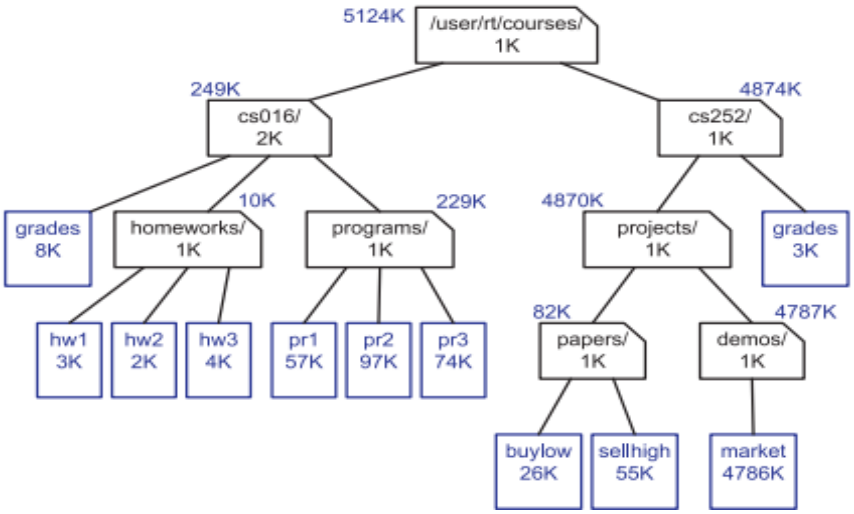


Another application of tree is for organizing file system hierarchically. In most operating systems, files are organized hierarchically into nested directories (also called folders), which are presented to the user in the form of a tree. More specifically, the internal nodes of the tree are associated with directories and the external nodes are associated with regular files. In the UNIX and Linux operating systems, the root of the tree is appropriately called the "root directory," and is represented by the symbol "/."

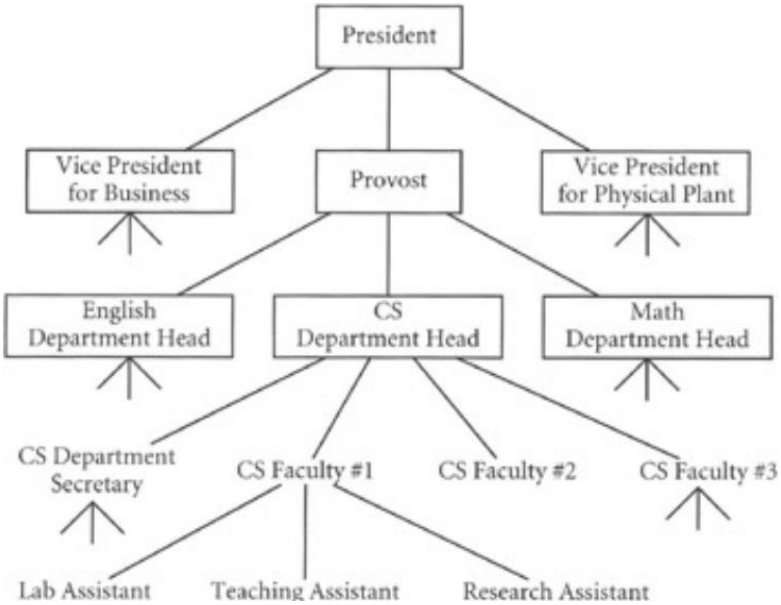
Consider a file-system tree *T*, where external nodes represent files and internal nodes represent directories. Suppose we want to compute the disk space used by a directory, which is recursively given by the sum of the following:

- ✓ The size of the directory itself
- ✓ The sizes of the files in the directory
- ✓ The space used by the children directories

The following tree represents a file system, showing the name and size of the associated file/directory inside each node, and the disk space used by the associated directory above each internal node.



Another application of tree is expressing organizational structures. Corporations or business entities often represent their internal organization in terms of a corporate hierarchy, similar to the following figure, with the president or CEO at the top and the rank-and-file employees at the bottom. Each connecting link represents a supervisory relationship. In general, each person has only one immediate boss, but each boss may have several underlings. Notice that the structure shows relative information about the individual employees: roughly, the closer a person is to the top of the tree, the higher the position in the chart—and on the corporate ladder. This particular mixed metaphor seems to make sense: a corporate hierarchy describes the corporation as a whole and is tree-structured, but "corporate ladder" views that same structure from the perspective of a single individual. In particular, it shows the linear path to the top. The chart even shows which persons are approximately equal in the organization; even though they have unrelated job titles and work in different divisions, they are placed at the same level of the tree.



ChatGPT

GPTGO.AI

The tree data structure has several applications in computer science and beyond. Here are some common applications of tree data structures:

1. File Systems: Tree structures are often used to represent hierarchical file systems, where each directory or folder is a node and the files within them are the child nodes.

2. HTML DOM: The Document Object Model (DOM) used in web development represents the HTML document structure as a tree, allowing manipulation and traversal of the content.

3. Database Systems: Many database systems use various tree structures, such as B-trees or AVL trees, to efficiently store and retrieve data, enabling fast searching, sorting, and indexing.

4. Network Routing: Tree structures, like the Spanning Tree Protocol (STP), play a crucial role in constructing and maintaining efficient routes within a network.

5. Compiler Design: Abstract Syntax Trees (ASTs) are used by compilers to represent the hierarchical structure of source code, allowing for efficient analysis and transformation.

6. AI and Decision Making: Decision trees are commonly used in machine learning algorithms for classification or regression tasks, aiding in decision-making processes.

7. Organizational Hierarchies: Tree structures can model hierarchical relationships within organizations, such as reporting structures or family trees.

8. Game Theory: Trees are used in game theory to model decision paths and outcomes, helping determine optimal strategies in various games.

These are just a few examples, and the applications of tree data structures extend beyond these areas. Trees are versatile data structures that provide efficient organization and storage of hierarchical relationships in various domains.

What are some common real-world examples where the tree data structure is used in applications?

How does the tree data structure make searching and storing data more efficient in computer science?

What are some advantages of using trees over other data structures such as linked lists or arrays?

How do tree data structures play a role in organizing hierarchical data, such as file systems or family trees?

What are some practical applications of binary search trees and balanced trees in algorithm design?

CHAPTER SEVEN
GRAPHS

7.1. Introduction

In spite of the flexibility of trees and the many different tree applications, trees, by their nature, have one limitation, namely, they can only represent relations of a hierarchical type, such as relations between parent and child. Other relations are only represented indirectly, such as the relation of being a sibling. A generalization of a tree, a graph, is a data structure in which this limitation is lifted. Intuitively, a graph is a collection of vertices (or nodes) and the connections between them. Generally, no restriction is imposed on the number of vertices in the graph or on the number of connections one vertex can have to other vertices. Graphs are versatile data structures that can represent a large number of different situations and events from diverse domains. They provide the ultimate in data structure flexibility. They can model both real-world systems and abstract problems, so they are used in hundreds of applications.

A graph is a way of representing relationships that exist between pairs of objects. That is, a graph is a set of objects, called vertices, together with a collection of pairwise connections between them.

7.2. Graph Terminologies

A graph G is simply a set V of *vertices* and a collection E of pairs of vertices from V, called *edges*. Thus, a graph is a way of representing connections or relationships between pairs of objects from some set V.

That is, G = (V, E), where V is the set of **vertices** and E is the set of **edges**. Each edge is a pair (v, w), where v, w ∈ V. Vertices are sometimes called ***nodes***, and edges are sometimes referred to as ***arcs***. Sometimes an edge has a third component, called the **edge cost** (or **weight**) that measures the cost of traversing the edge.

- Edges in a graph are either **directed** or **undirected**. An edge (u, v) is said to be directed from u to v if the pair (u, v) is ordered, with *u* preceding *v*. An edge (u, v) is said to be undirected if the pair (u, v) is not ordered.

- If all the edges in a graph are undirected, then we say the graph is an **undirected graph**. Likewise, a **directed graph**, is a graph whose edges are all directed. A graph that has both directed and undirected edges is often called a **mixed graph**. In a directed graph each edge is an ordered pairs of vertices, i.e. each edge is represented by a directed pair. If e = (v, w) then *v* is tail or initial vertex and *w* is head of final vertex. Subsequently (v, w) and (w, v) represent two different edges. In an undirected graph, pair of vertices representing any edge is unordered. Thus (v,w)

and (w, v) represent the same edge. Directed graph can be referred as digraph and undirected graph as graph.

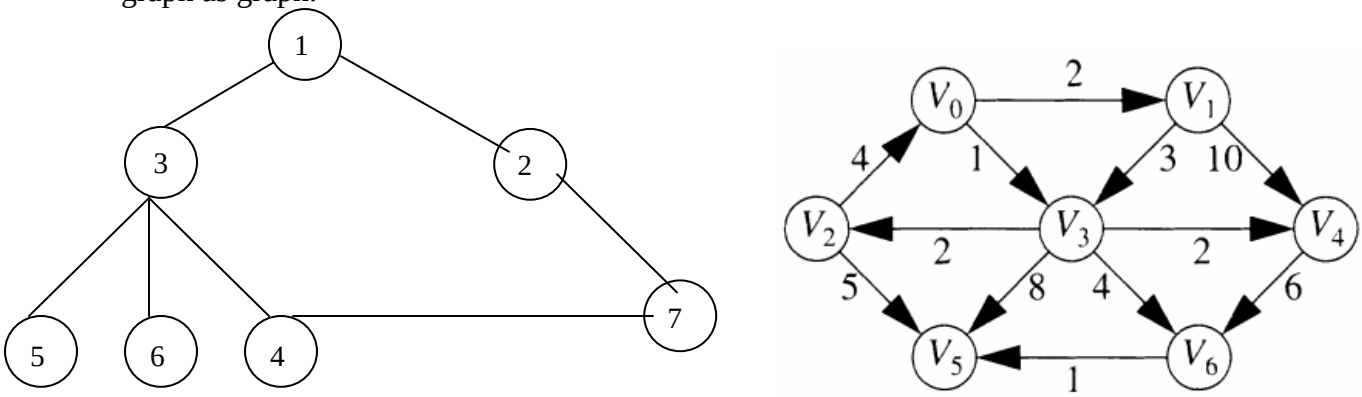


Figure 7.1: (a) Undirected and unweighted graph (b) directed and weighted graph

- ✓ For the above graph (a), the set of vertices and edges are

  V(G) = {1, 2, 3, 4, 5, 6, 7} and **E**(G) = { (1, 2), (1, 3), (2, 7), (3, 4), (5, 3), (3, 6), (7, 4) }

- ✓ For graph (b), the set of vertices and edges are

  V(G) = {V0, V1,V2, V3, V4, V5, V6} and

  E(G) = { (V0, V1, 2), (V0, V3, 1), (V1, V3, 3), (V1, V4, 10), (V2, V0, 4), (V2, V5, 5), (V3,V2, 2), (V3, V4, 2), (V3, V6, 4), (V3, V5, 8), (V4, V6, 6), (V6, V5, 1)}

- ➢ A **path** in a graph is a sequence of vertices connected by edges. In other words, it is the sequence of vertices w_1 , w_2, ..., w_N such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < N$. The **path length** is the number of edges on the path; namely N – 1; also called the **unweighted path length**. The **weighted path length** is the sum of the costs of the edges on the path. For example for figure 7.1 (b); V0, V3, V5 is a path from vertex V0 to V5. The path length is two edges, the shortest path between V0 and V5, and the weighted path length is 9. However, if cost is important, the weighted shortest path between these vertices has cost 6 and is V0, V3, V6, V5. A path $w_1$may exist from a vertex to itself. If this path contains no edges, the path length is 0, which is a convenient way to define an otherwise special case. A **simple path** is a path in which all vertices are distinct, except that the first and last vertices can be the same.

- ➢ Two vertices *u* and *v* said to be **adjacent** if they are joined by an edge, that is, if there is an edge whose end vertices are u and v. Such vertices are also called **neighbors**. Such an edge is said to be **incident** on vertices *u* and *v*.

For example for the above graph (a) at figure 7.1, adjacent vertices for vertex 3 are 1, 4, 5 and 6. For graph (b), V1, V3 and V6 are adjacent vertices for vertex V4.

- A **cycle** in a directed graph is a path that begins and ends at the same vertex and contains at least one edge. That is, it has a length of at least 1 such that $w_1 = w_N$; this cycle is simple if the path is simple. For undirected graphs, we require that the edges be distinct. The logic of these requirements is that the path u, v, u in an undirected graph should not be considered a cycle, because (u, v) and (v, u) are the same edge. In a directed graph, these are different edges, so it makes sense to call this a cycle. A **directed acyclic graph (DAG)** is a type of directed graph having no cycles.
- A graph with n vertices is called **complete** and is denoted K_n if for each pair of distinct vertices there is exactly one edge connecting them; that is, each vertex can be connected to any other vertex.
- An undirected graph is **connected** if there exists a path from every vertex to every other vertex. A directed graph with this property is called **strongly connected**. If a directed graph is not strongly connected, but the underlying graph (without direction to the arcs) is connected, then the graph is said to be **weakly connected**.

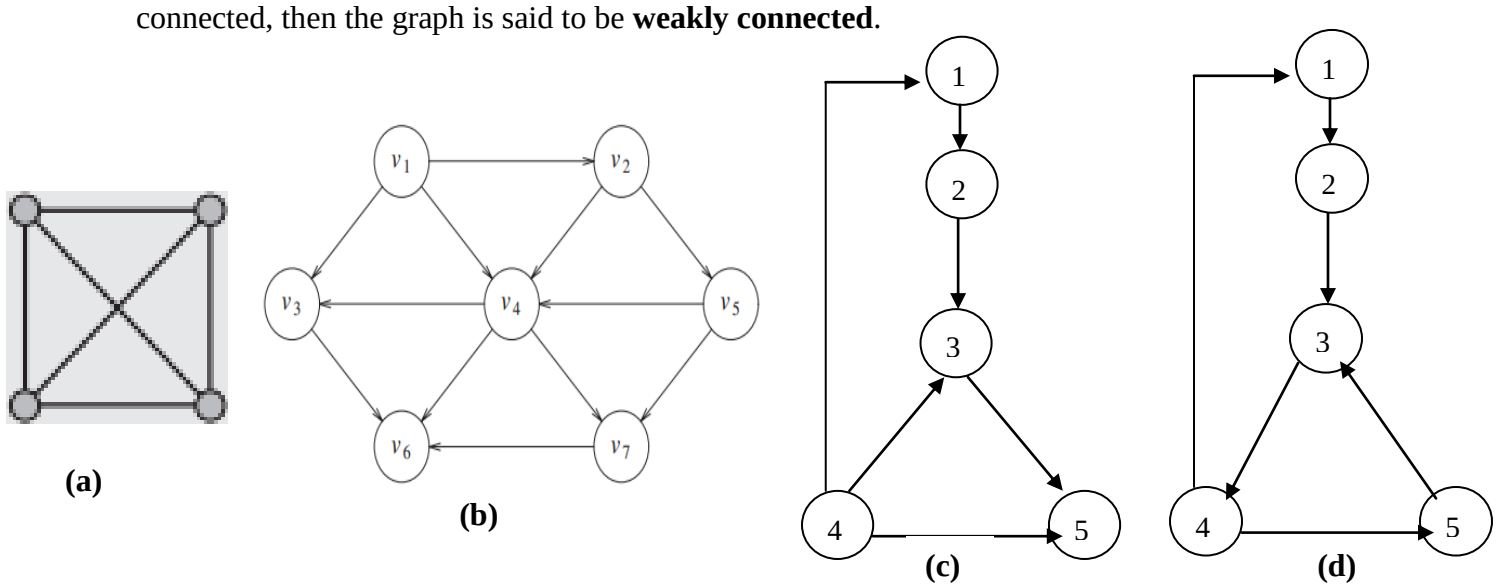


Figure 7.2: (a) complete graph, (b) acyclic directed graph, (c) weakly connected graph, (d) strongly connected graph

From this figure we observe that, in (a), there exists a path from each vertex to every other vertices, so it is called complete graph. In (b), there is no cycle formed in the graph, and hence it is directed acyclic graph. In (c) there does not exist a directed path from vertex 1 to vertex 4 and also from vertex 5 to other vertices and so on. Therefore it is a weakly connected graph. In (d) there is a directed path from any vertex to any other vertex, hence it is strongly connected graph.

- The two vertices joined by an edge are called the **end vertices** (or endpoints) of the edge. If an edge is directed, its first endpoint is its **origin** and the other is the **destination** of the edge. The **outgoing edges** of a vertex are the directed edges whose origin is that vertex. The **incoming edges** of a vertex are the directed edges whose destination is that vertex. The **degree** of a vertex v, denoted deg(v), is the number of incident edges of v. The **in-degree** and **out-degree** of a vertex v are the number of the incoming and outgoing edges of v, and are denoted indeg(v) and outdeg(v), respectively. Thus, the indegree of a vertex v is the number of edges ending at *v* and the outdegree of a vertex v is the number of edges starting from v.
- A vertex *v* is said to be **source** if outdeg(v) > 0 and indeg(v) = 0. A vertex *v* is called a **sink** if indeg(v) > 0 and outdeg(v) = 0.

  For example, from figure 7.1 (a), the degree of vertex 3 is 4. And from figure 7.2 (d) the indegree of vertex 4 is 1 and outdegree of vertex 4 is 2. From figure 7.2 (c), vertex 4 is a source vertex and vertex 5 is a sink vertex.

7.3. Graph Representations

The picture of graph with circles (vertices) and lines (edges) is *only a depiction*. Graph is a mathematical structure and finds its application in many areas of interest in which problems need to be solved using computers. Thus this mathematical structure must be represented as some kind of data structures.

There are two commonly used methods for representing graphs, ***adjacency matrix*** and ***adjacency list.*** The adjacency list structure only store the edges actually present in the graph, while the adjacency matrix stores a placeholder for every pair of vertices (whether there is an edge between them or not). The choice of representation depends on the application and function to be performed on the graph.

i) Adjacency Matrix: Representing a graph by a matrix

An adjacency matrix is a way of representing an n vertex graph G = (V; E) by an n X n matrix, whose entries are boolean values. In the adjacency matrix representation, we think of the vertices as being the integers in the set {0, 1, ... , n − 1} and the edges as being pairs of such integers. This allows us to store references to edges in the cells of a two-dimensional n × n array.

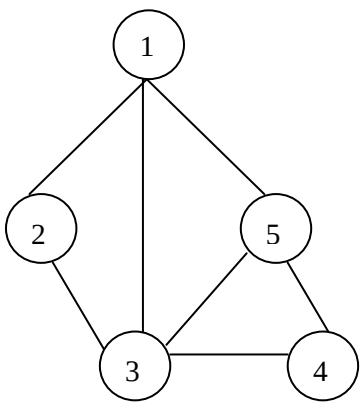
The adjacency matrix for a graph of total vertices |V| is a |V| x |V| array. Assume that |V| = n and that the vertices are labeled from v_0 through v_{n-1}. Row *i* of the adjacency matrix contains entries

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.

An adjacency matrix of graph G= (V, E) is a binary |V|× |V| matrix such that each entry of this matrix

$$A_{ij} = \begin{cases} 1 & \text{if there exists an edge } (v_i\ v_j) \\ 0 & \text{otherwise} \end{cases}$$

For example, the following undirected graph is represented using the adjacency matrix as follows.

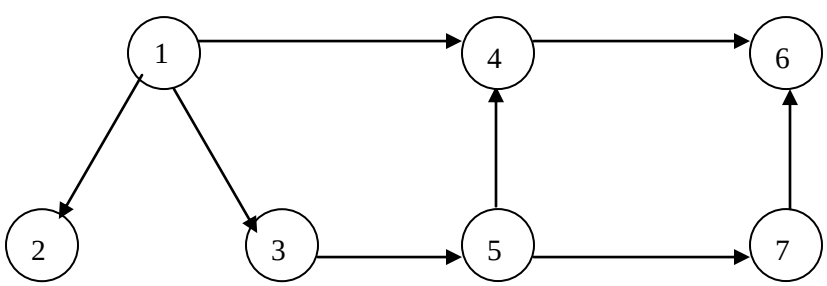


Vertices	1	2	3	4	5
1	0	1	1	0	1
2	1	0	1	0	0
3	1	1	0	1	1
4	0	0	1	0	1
5	1	0	1	1	0

Figure 7.3: Adjacency matrix of graph

You may observe that the adjacency matrix for an undirected graph is symmetric, as the lower and upper triangles are same. Also all the diagonal elements are zero.

Consider the following directed graph



Its adjacency matrix looks like the following.

Vertices	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	1	0	0
4	0	0	0	0	0	1	0
5	0	0	0	1	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

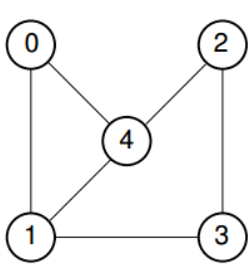
The total number of 1's account for the number of edges in the digraph. The number of 1's in each row tells the *out degree* of the corresponding vertex. The total number of 1's in each column tells the *in degree* of the corresponding vertex.

ii) Adjacency List: A graph as a collection of lists

Adjacency list representations of graphs take a more vertex-centric approach. In an adjacency list representation, the graph G = (V; E) is represented as an array, *adj*, of lists. The list *adj[i]* contains a list of all the vertices adjacent to vertex *i*. That is, it contains every index *j* such that (*i*,j) ∈ E. 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.

Adjacency list specifies all vertices adjacent to each vertex of the graph. This list can be implemented as a table, in which case it is called a *star* representation, which can be forward or reverse, as illustrated in Figure 7.4 a, or as a *linked list* (Figure 7.4b).

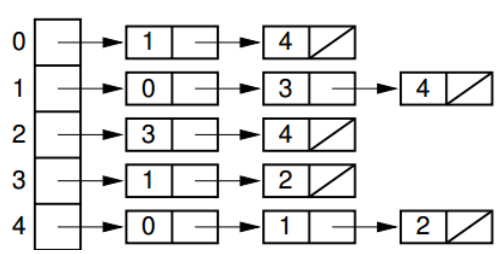
For example, for the following undirected graph, its adjacency list in star and linked list representation is given below.



(a)

0	1 4
1	0 3 4
2	3 4
3	1 2
4	0 1 2

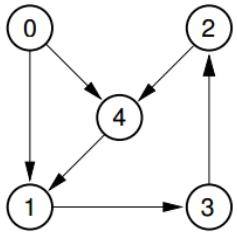
(b)



(c)

Figure 7.4: (a) Undirected graph. Adjacency list of graph (a) in (b) star representation (c) Linked list representation

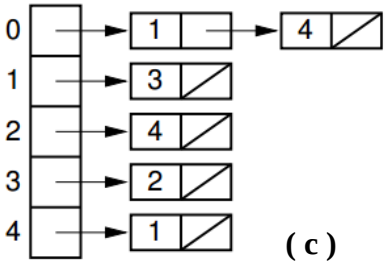
For the following directed graph, its adjacency list in star and linked list representation is given below.



(a)

0	1 4
1	3
2	4
3	2
4	1

(b)



(c)

Figure 7.5: (a) directed graph. Adjacency list of graph (a) in (b) star representation (c) Linked list representation

The entry for Vertex 0 in Figure 7.5(c) stores 1 and 4 because there are two edges in the graph leaving Vertex 0, with one going Vertex 1 and one going to Vertex 4. The list for Vertex 2 stores an entry for Vertex 4 because there is an edge from Vertex 2 to Vertex 4, but no entry for Vertex 3 because this edge comes into Vertex 2 rather than going out.

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. 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.

7.4. Graph Traversal

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, inorder, or postorder. Multiple tree traversals exist because various applications require the nodes to be visited in a particular order. For example, to print a binary search tree's nodes in ascending order requires an inorder 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.

Graph traversal algorithms can solve both of these problems by maintaining a mark bit for each vertex on the graph. At the beginning of the algorithm, the mark bit for all vertices is cleared. The mark bit for a vertex is set when the vertex is first visited during the traversal. If a marked vertex is encountered during traversal, it is not visited a second time. This keeps the program from going into an infinite loop when it encounters a cycle.

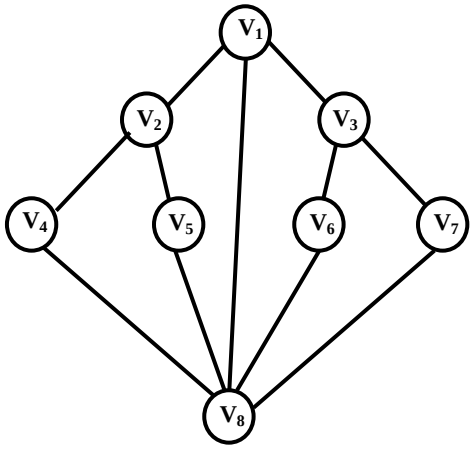
Once the traversal algorithm completes, we can check to see if all vertices have been processed by checking the mark bit array. If not all vertices are marked, we can continue the traversal from another unmarked vertex. Note that this process works regardless of whether the graph is directed or undirected.

In graphs, we do not have any start vertex or any special vertex signaled out to start traversal from. Therefore the traversal may start from any arbitrary vertex. Two graph traversal methods are commonly used. These are Depth First Search (DFS) and Breadth First Search (BFS).

i) Depth – First – Search (DFS) traversal

In this algorithm, each vertex *v* is visited and then each unvisited vertex adjacent to *v* is visited. If a vertex *v* has no adjacent vertices or all of its adjacent vertices have been visited, we backtrack to the predecessor of *v*. The traversal is finished if this visiting and backtracking process leads to the first vertex where the traversal started. If there are still some unvisited vertices in the graph, the traversal continues restarting for one of the unvisited vertices.

For example, consider the following undirected graph. Traversing its node using depth – first – search algorithm looks like as follows.



1) Let us start visiting from vertex V_1.
2) Adjacent vertices for vertex V_1 are V_2, V_8, and V_3.
   Let us take vertex V_2, and visit it.
3) Adjacent vertices for vertex V_2 are V_1, V_4, V_5.
   V_1 is already visited (at step 1). Let us take vertex V_4, and visit it.
4) Adjacent vertices for vertex V_4 are V_2, V_8.
   V_2 is already visited (at step 2). Let us take vertex V_8, and visit it.
5) Adjacent vertices for vertex V_8 are V_4, V_5, V_1, V_6, V_7.
   V_4 (at step 3) and V_1 are already visited. Let us take vertex V_5, and visit it.
6) Adjacent vertices for vertex V_5 are V_2, V_8.
   Both V_2 and V_8 (at step 4) are already visited. Therefore we back track.
7) We have V_6 and V_7 (at step 5) unvisited in the list of V_8. We may visit any. We visit V_6.
8) Adjacent vertices for vertex V_6 are V_8 and V_3.
   Obviously the choice is V_3.
9) Adjacent vertices for vertex V_3 are V_1, V_6 and V_7.
   We visit V_7.

All the adjacent vertices of V_7 are already visited, we backtrack and find that we have visited all the vertices. Therefore the sequence of traversal is

$$\mathbf{V_1, V_2, V_4, V_8, V_5, V_6, V_3, V_7.}$$

We may implement the depth first search method by using a stack, pushing all unvisited vertices adjacent to the one just visited and popping the stack to find the next vertex to visit.

The algorithm for depth – first – search traversal in iterative and recursive format is stated in the following manner.

Iterative version:

During the execution of these algorithms, each node will be in one of the following states, called the status of node n.

Status = 1 (Ready state):- Initial state of the node 'n'.

Status = 2 (Waiting state):- 'n' is on stack.

Status = 3(Processed state):- Node 'n' has been processed.

Step1: Initialize all nodes to the ready state i.e. Status =1.

Step 2: Push the starting node onto stack and change its status = 2.

Step 3: Repeat while (stack ! = empty)

a) Pop the top node 'n' of the stack. Process 'n' and change its status to the processed state (i.e, status = 3)

b) Push onto stack all the neighbors of 'n' that are still in the ready state and change their status to the waiting state (i.e. status = 2).

Recursive version:

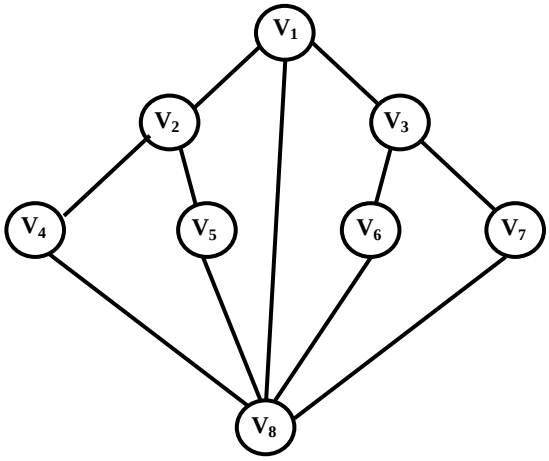
```
dfs (vertex V)
  {
     visited [V] = true;
     for each w adjacent to V
        if (!visited [w])
             dfs(w);
  }
```

ii) Breadth – First – Search (BFS) traversal

The 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.

In DFS we pick on one of the adjacent vertices; visit all of the adjacent vertices and back track to visit the unvisited adjacent vertices. In BFS we first visit all the adjacent vertices of the start vertex and then visit all the unvisited vertices adjacent to these and so on.

Consider the following graph, and let us see how BFS traversal algorithm looks like.



1) We start the traversal from vertex V_1. Its adjacent vertices are V_2, V_8, V_3. We visit all vertices one by one.
2) We take one of the adjacent vertex of V_1, let us take vertex V_2. The unvisited adjacent vertices to V_2 are V_4 and V_5. We visit both.
3) We go back to the remaining unvisited adjacent vertices of V_1 and pick on one of those say V_3. The unvisited adjacent vertices of V_3 are V_6 and V_7.
4) There are no more unvisited adjacent vertices of V_8, V_4, V_5, V_6 and V_7. Thus the sequence so generated is **V_1 V_2 V_8 V_3 V_4 V_5 V_6 V_7**

Here we need a queue instead of a stack to implement it. We add unvisited vertices adjacent to the one just visited, at the rear and read at front to find the next vertex to visit.

Abstract view of BFS algorithm:

Step1: Initialize all nodes to the ready state i.e. Status = 1.

Step 2: Put the starting node onto queue and change its status = 2.

Step 3: Repeat while (queue! = empty)

a) Remove the front node 'n' of the queue. Process 'n' and change its status to the processed state (i.e. Status = 3)
b) Add to the rear of the queue all the neighbors of 'n' that are still in the ready state and change their status to the waiting state (i.e. status = 2).

Detailed view of BFS algorithm:

```
bfs (vertex v)
 {  vertex w, queue q;
    visited [v] = true;
    initialise (q);
    addqueue (q, v)
    while (! Emptyqueue ( q ) )
    {  deletequeue (q, v);
       for all vertices w adjacent to v
          if ( ! visited [w] )
          {  addqueue (q, w);
             visited[w] = true;
          }
    }
 }
```

7.5. Application of graphs

An example of a real-life situation that can be modeled by a graph is the airport system. Each airport is a vertex, and two vertices are connected by an edge if there is a nonstop flight from the airports that are represented by the vertices. The edge could have a weight, representing the time, distance, or cost of the flight. It is reasonable to assume that such a graph is directed, since it might take longer or cost more (depending on local taxes, for example) to fly in different directions. We would probably like to make sure that the airport system is strongly connected, so that it is always possible to fly from any airport to any other airport. We might also like to quickly determine the best flight between any two airports. "Best" could mean the path with the fewest number of edges or could be taken with respect to one, or all, of the weight measures.

Traffic flow can be modeled by a graph. Each street intersection represents a vertex, and each street is an edge. The edge costs could represent, among other things, a speed limit or a capacity (number of lanes). We could then ask for the shortest route or use this information to find the most likely location for bottlenecks.

Here is a small sampling of the range of problems that graphs are routinely applied to.

- Modeling connectivity in computer and communications networks.
- Representing a map as a set of locations with distances between locations; used to compute shortest routes between locations.
- Modeling flow capacities in transportation networks.
- Finding a path from a starting condition to a goal condition; for example, in artificial intelligence problem solving.
- Modeling computer algorithms, showing transitions from one program state to another.
- Finding an acceptable order for finishing subtasks in a complex activity, such as constructing large buildings.
- Modeling relationships such as family trees, business or military organizations, and scientific taxonomies.

7.6. Shortest Path problems

Finding the shortest path is a classical problem in graph theory, and a large number of different solutions have been proposed. Edges are assigned certain weights representing, for example, distances between cities, times separating the execution of certain tasks, costs of transmitting information between locations, amounts of some substance transported from one place to

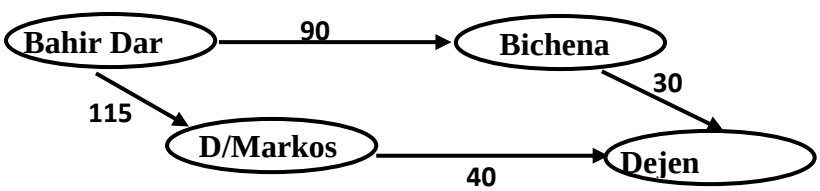
another, and so on. When determining the shortest path from vertex v to vertex u, information about distances between intermediate vertices w has to be recorded. This information can be recorded as a label associated with these vertices, where the label is only the distance from v to w or the distance along with the predecessor of w in this path. The methods of finding the shortest path rely on these labels.

Let G be a weighted graph. The length (or weight) of a path is the sum of the weights of the edges of P. That is, if P = ((v0, v1), (v1,v2), ..., (vk−1,vk)), then the length of P, denoted w(P), is defined as

$$w(P) = \sum_{i=0}^{k-1} w((v_i, v_{i+1})).$$

The distance from a vertex v to a vertex U in G, denoted d(v, U), is the length of a minimum length path (also called shortest path)from v to u, if such a path exists.

For example, in the graph we can go from Bahir Dar to Dejen through D/Markos at a cost of 155 or through Bichena at a cost of 120. (These numbers may reflect the bus ticket in birr.) In these and many other applications, we are often required to find a shortest path i.e. a path having minimum weight between two vertices.



Length of the path is the sum of weights of the edges on that path. The starting vertex of the path is called the source vertex and the last vertex of the path is called the destination vertex.

Single source shortest path problem

Given Vertex S in Graph G, find a shortest path from S to every other vertex in G. We might want only the shortest path between two vertices, S and T. How-ever in the worst case, while finding the shortest path from S to T, we might find the shortest paths from S to every other vertex as well. So there is no better algorithm (in the worst case) for finding the shortest path to a single vertex than to find shortest paths to all vertices.

<u>Reading assignment</u>

1. Minimum spanning tree (Kruskal's, Prim's and Sollin's algorithms)
2. Shortest path (Dijkstra algorithm)

Chapter eight

8. Advanced Sorting and Searching Algorithms

8.1. *Shell Sort*

Shell sort is an improvement of insertion sort. It is developed by Donald Shell in 1959. Insertion sort works best when the array elements are sorted in a reasonable order. Thus, shell sort first creates this reasonable order.

Algorithm:

1. Choose gap g_k between elements to be partly ordered.
2. Generate a sequence (called increment sequence) g_k, g_{k-1},…., g_2, g_1 where for each sequence g_i, A[j]<=A[j+g_i] for 0<=j<=n-1-g_i and k>=i>=1

It is advisable to choose g_k =n/2 and g_{i-1} = g_i/2 for k>=i>=1. After each sequence g_{k-1} is done and the list is said to be g_i-sorted. Shell sorting is done when the list is *1-sorted* (which is sorted using insertion sort) and A[j]<=A[j+1] for 0<=j<=n-2. Time complexity is O($n^{3/2}$).

Example: Sort the following list using shell sort algorithm.

5	8	2	4	1	3	9	7	6	0

Choose g_3 =5 (n/2 where n is the number of elements =10)

Sort (5, 3)	3	8	2	4	1	5	9	7	6	0
Sort (8, 9)	3	8	2	4	1	5	9	7	6	0
Sort (2, 7)	3	8	2	4	1	5	9	7	6	0
Sort (4, 6)	3	8	2	4	1	5	9	7	6	0
Sort (1, 0)	3	8	2	4	0	5	9	7	6	1
➔ 5- sorted list	3	8	2	4	0	5	9	7	6	1

Choose g_2 =2

Sort (3, 2, 0, 9, 6)	0	8	2	4	3	5	6	7	9	1
Sort (8,4,5,7,1)	0	1	2	4	3	5	6	7	9	8
➔ 2- sorted list										

Choose g_1 =1 (the same as insertion sort algorithm)

Sort (0, 1, 2, 4, 3, 5, 6, 7, 9, 8)	0	1	2	3	4	5	6	7	8	9
➔ 1- sorted (shell sorted) list	0	1	2	3	4	5	6	7	8	9

7.2. *Quick Sort*

Quicksort is a space-optimized version of the binary tree sort, and it can be done in-place, requiring small additional amounts of memory to perform the sorting

Quick sort is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).

Algorithm:

1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
   - the left part has items less than or equal to the pivot value
   - the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

The following algorithm can be used to position a pivot value and create partition.

```
Left=0;
Right=n-1; // n is the total number of elements in the list
PivotPos=Left;
while(Left<Right)
{
        if(PivotPos==Left)
        {
                if(Data[Left]>Data[Right])
                {
                        swap(data[Left], Data[Right]);
                        PivotPos=Right;
                        Left++;
                }
                else
                        Right--;
        }
        else
        {
                if(Data[Left]>Data[Right])
                {
                        swap(data[Left], Data[Right]);
                        PivotPos=Left;
                        Right--;
                }
                else
                        Left++;
        }
}
```

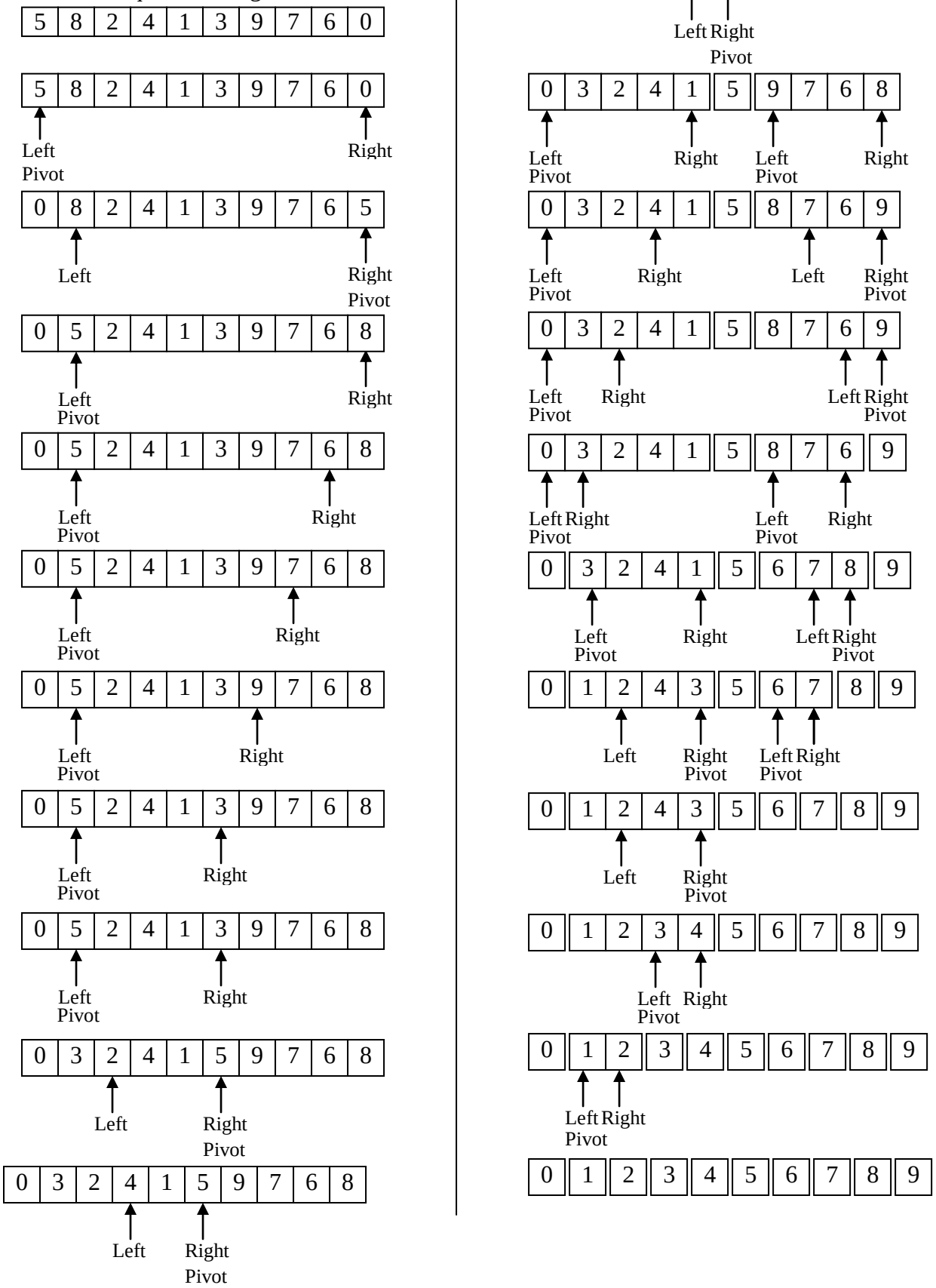
Quick Sort is an efficient algorithm with an average and best-case time complexity of O(n log n), and a worst-case time complexity of O(n^2) when the pivot selection is in the worst possible order.

Quick Sort is a popular sorting algorithm commonly used to arrange elements in a list or an array. It follows the divide-and-conquer approach, where it recursively divides the input into smaller sub-arrays and sorts them independently.

Here is a step-by-step explanation of the Quick Sort algorithm:

1. Choose a pivot element from the array. It can be any element, but commonly the first or last element is selected.
2. Partition the array by rearranging its elements in such a way that all elements less than the pivot are placed before the pivot, and all elements greater than the pivot are placed after it. The pivot element then reaches its correct sorted position.
3. Recursively apply the above two steps to the sub-arrays formed by dividing the array into two halves. The pivot element will be in its sorted position, so it doesn't need any further comparisons.
4. Repeat the process for the smaller sub-arrays until the entire array is sorted.
5. Finally, when the sorting process is complete, the array is sorted in ascending order.

Example: Sort the following list using quick sort algorithm.



8.3. *Heap Sort*

The time complexity of heap sort is O(nlogn) in both the best and worst cases . The space complexity of heap sort is O(1), which means that it is an in-place sorting algorithm.

Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).

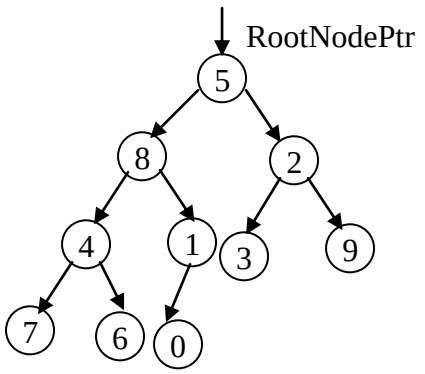
Algorithm:

1. Construct a binary tree
   - The root node corresponds to Data[0].
   - If we consider the index associated with a particular node to be *i*, then the left child of this node corresponds to the element with index 2**i*+1 and the right child corresponds to the element with index 2**i*+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.
2. Construct the heap tree from initial binary tree using "adjust" process.
3. Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

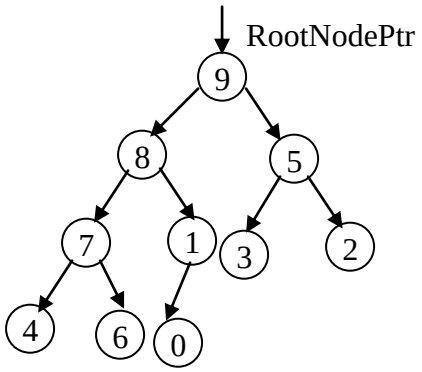
Example: Sort the following list using heap sort algorithm.

5	8	2	4	1	3	9	7	6	0

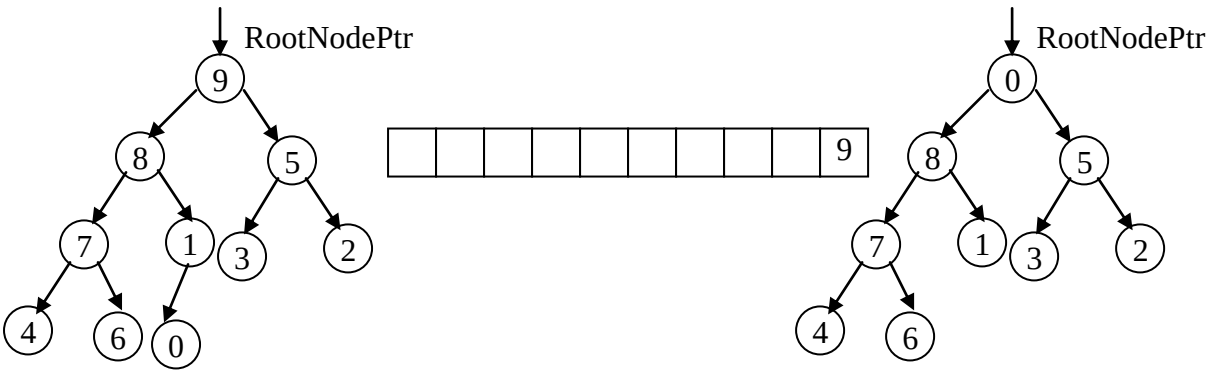
Construct the initial binary tree

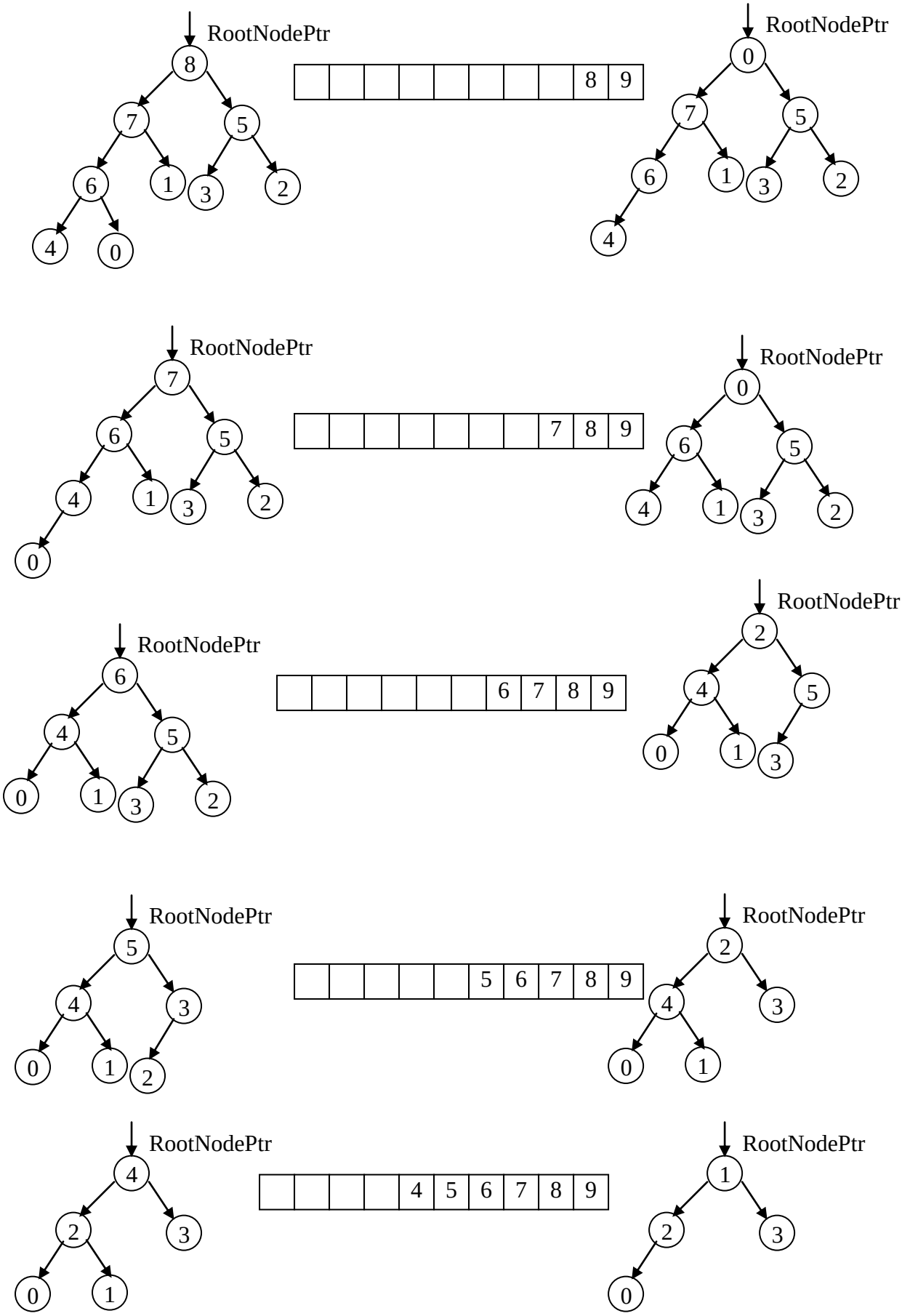


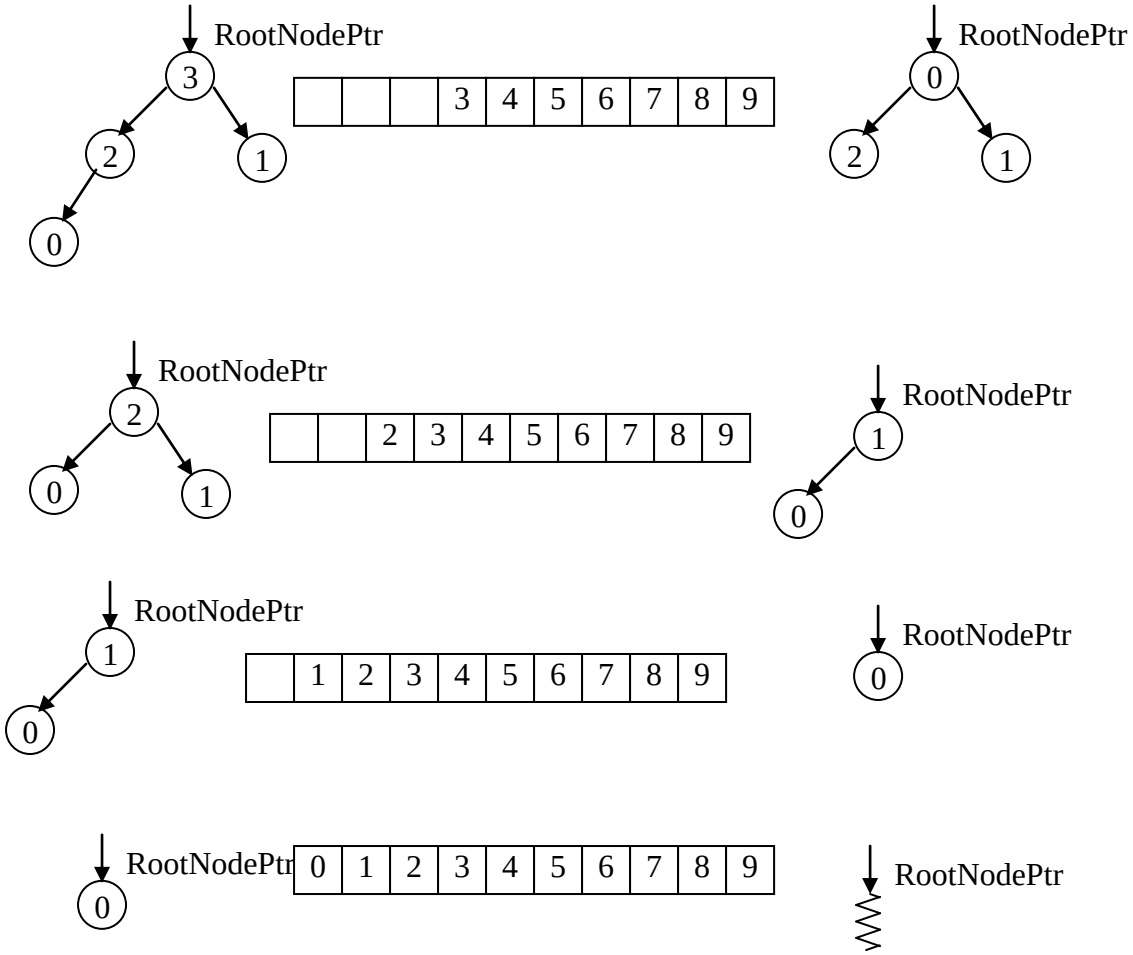
Construct the heap tree



Swap the root node with the lowest, right most node and delete the lowest, right most value; insert the deleted value in the array in its proper position; adjust the heap tree; and repeat this process until the tree is empty.




RootNodePtr
8 9
RootNodePtr
7 8 9
RootNodePtr
6 7 8 9
RootNodePtr
5 6 7 8 9
RootNodePtr
4 5 6 7 8 9



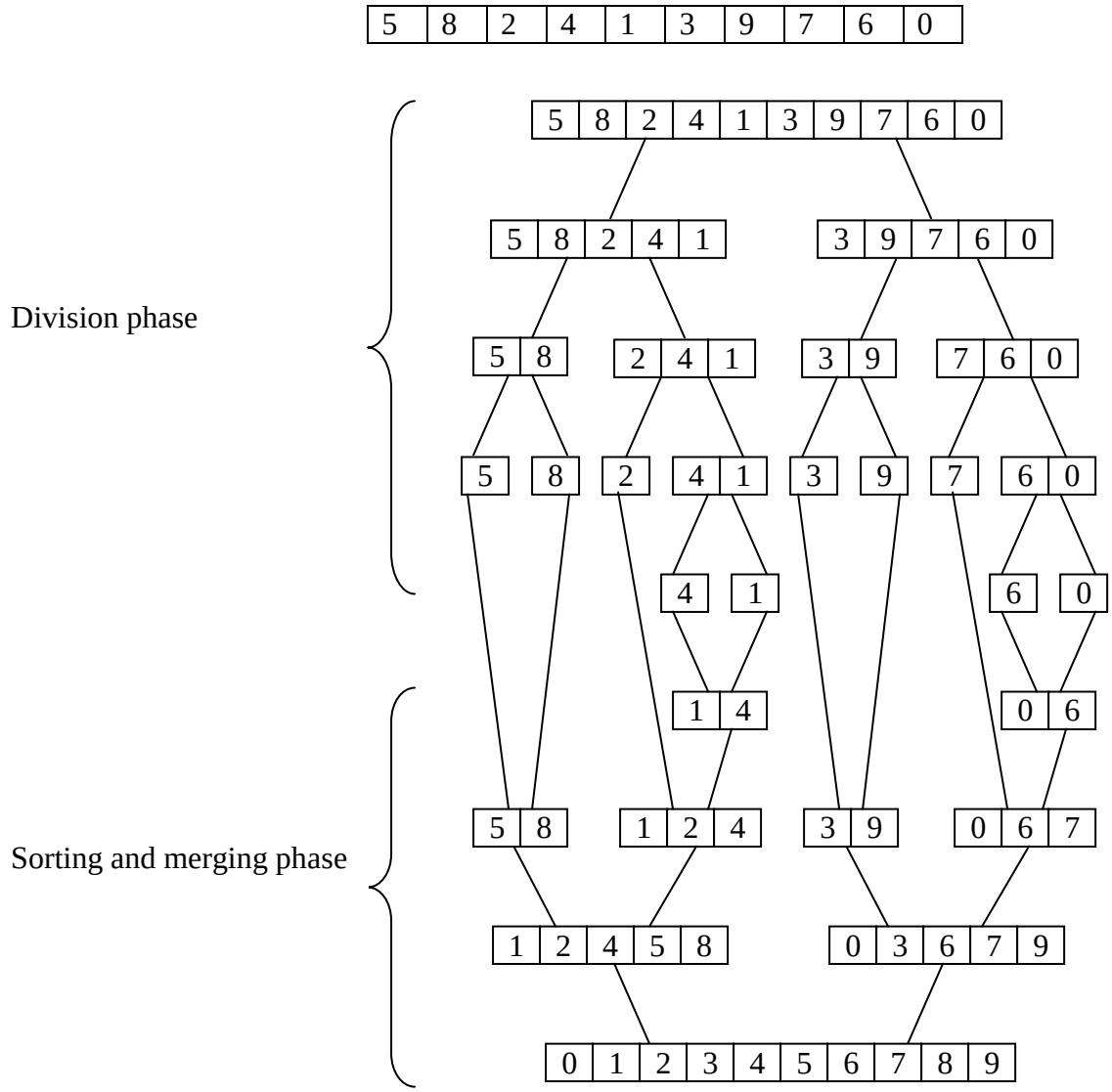
8.4. *Merge Sort*

Like quick sort, merge sort uses divide and conquer strategy and its time complexity is O(nlogn).

Algorithm:

1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Example: Sort the following list using merge sort algorithm.



Reading Assignment

1. Hashing searching technique

Quick Sort
Follows the divide and conquer method.
The array is parted into any ratio.
Worst case complexity is O(n^2).
More efficient and works faster than merge sort in case of smaller array size or datasets.
Internal sorting method where the data is sorted in main memory.
Mostly preferred for large unsorted arrays.
Merge Sort
Follows the divide and conquer method.
The array is parted into just 2 halves (i.e. n/2).
Worst case and average case has same complexities O(n log n).
More efficient and works faster than quick sort in case of larger array size or datasets.
External sorting method in which the data that is to be sorted cannot be accommodated in the memory and needed auxiliary memory for sorting.
Mostly applicable for the linked lists.

Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets, whereas Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets. Merge sort generally performs less comparisons than quick sort both worst case and on average. If performing a comparison is costly, merge sort will definitely have the upper hand in terms of speed. Merge sort requires more memory, whereas Quick sort is generally believed to faster in common real-life settings. This is mainly due to its lower memory consumption which usually affects time performance as well.

ChatGPT

GPTGO.AI

The time complexity of sorting algorithms can vary depending on the algorithm used. Here are some common sorting algorithms and their worst-case, average-case, and best-case time complexities:

1. Bubble Sort:
- Worst-case: O(n^2)
- Average-case: O(n^2)
- Best-case: O(n)

2. Selection Sort:
- Worst-case: O(n^2)
- Average-case: O(n^2)
- Best-case: O(n^2)

3. Insertion Sort:
- Worst-case: O(n^2)
- Average-case: O(n^2)
- Best-case: O(n)

4. Merge Sort:
- Worst-case: O(n log n)
- Average-case: O(n log n)
- Best-case: O(n log n)

5. Quick Sort:
- Worst-case: O(n^2)
- Average-case: O(n log n)
- Best-case: O(n log n)

6. Heap Sort:
- Worst-case: O(n log n)
- Average-case: O(n log n)
- Best-case: O(n log n)

It's important to note that these time complexities are based on comparisons between elements in the sorting process and assume the worst or average case scenarios. The best-case complexities are achieved when the input is already sorted or nearly sorted.

What is the difference between the worst case, average case, and best case time complexity of sorting algorithms?

Which sorting algorithm has the worst case time complexity of O(n^2)?

What is the average time complexity of quicksort compared to mergesort?

Which sorting algorithm has the best case time complexity of O(n)?

In which scenario would heap sort have the best time complexity - worst case, average case, or best case?

ChatGPT

GPTGO.AI

Sure! Here are the worst case, average case, and best case time complexities of some popular sorting algorithms, including Shell Sort:

1. Bubble Sort:
- Worst Case: O(n^2)
- Average Case: O(n^2)
- Best Case: O(n)

2. Selection Sort:
- Worst Case: O(n^2)
- Average Case: O(n^2)
- Best Case: O(n^2)

3. Insertion Sort:
- Worst Case: O(n^2)
- Average Case: O(n^2)
- Best Case: O(n)

4. Merge Sort:
- Worst Case: O(n log n)
- Average Case: O(n log n)
- Best Case: O(n log n)

5. Quick Sort:
- Worst Case: O(n^2)
- Average Case: O(n log n)
- Best Case: O(n log n)

6. Heap Sort:
- Worst Case: O(n log n)
- Average Case: O(n log n)
- Best Case: O(n log n)

7. Shell Sort:
- Worst Case: Depends on the gap sequence used (can vary between O(n log n) and O(n^2))
- Average Case: Unknown (depends on gap sequence)
- Best Case: O(n log n)

Please note that these time complexities represent the general scenario and may vary based on the actual input data. Additionally, the average case time complexities for some algorithms may be difficult to determine precisely.

What is the worst case time complexity of Shell Sort?

What is the average case time complexity of Insertion Sort?

What is the best case time complexity of Bubble Sort?

What is the worst case time complexity of Quick Sort?

What is the average case time complexity of Heap Sort?

ChatGPT

GPTGO.AI

Sure, I can provide you with the worst case, average case, and best case time complexity for some common search algorithms:

1. Linear Search:
- Worst Case: O(n)
- Average Case: O(n)
- Best Case: O(1)

2. Binary Search (on a sorted array):
- Worst Case: O(log n)
- Average Case: O(log n)
- Best Case: O(1)

3. Depth-First Search (DFS) on a graph:
- Worst Case: O(|V| + |E|) - where |V| is the number of vertices and |E| is the number of edges.
- Average Case: O(|V| + |E|)
- Best Case: O(|V| + |E|)

4. Breadth-First Search (BFS) on a graph:
- Worst Case: O(|V| + |E|)
- Average Case: O(|V| + |E|)
- Best Case: O(|V| + |E|)

5. Hash table lookup:
- Worst Case: O(n) in the case of many collisions
- Average Case: O(1) if the hash function is well-implemented and the data is evenly distributed
- Best Case: O(1)

Please note that these time complexities represent typical scenarios, and the actual running time can vary based on various factors. Also, there are several other search algorithms with different complexities depending on the specific problem they are used for.

What is the worst case time complexity of linear search algorithm?

What is the average case time complexity of binary search algorithm?

What is the best case time complexity of interpolation search algorithm?

What is the worst case time complexity of breadth-first search algorithm?

What is the best case time complexity of depth-first search algorithm?