Arba Minch University
Arba Minch Institute of Technology

Faculty of computing and software Engineering (FCSE)

Exit Exam Module for Data structure and Algorithm (SENG2082)

B.Sc. Software Engineering

Prepared by Yitayish Lema (MSC.)
Email: Yitulema@gmail.com
March 4, 2023 G.C.
Arba Minch, Ethiopia

Contents

The course aims:

- ✓ To introduce the most common data structures like stack, queue, linked list
- ✓ To give alternate methods of data organization and representation
- ✓ To enable students, use the concepts related to Data Structures and Algorithms to solve real world problems
- ✓ To practice Recursion, Sorting, and Searching on the different data structures
- ✓ To implement the data structures with a chosen programming language

Chapter one

1.1 Introduction

The **data structure** is a collection of data elements organized in a specified manner in computer's memory (or sometimes on a disk) and there exists an efficient method to store and retrieve individual data elements.

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.**

Algorithms manipulate the data in these structures in various ways, such as inserting a new data element, searching for a particular element, or sorting the elements. Generally, the data element is of specific data types. We may, therefore, say data are represented that:

Data Structure = Organized Data + Allowed Operations

NB. Designing and using data structures are an important programming skill.

- *Linear and Non-linear data structures*: In linear data structures the data elements are arranged in a linear sequence like in an array, data processed one by one sequentially. Linear data structures contain following types of data structures:
    - ✓ Arrays
    - ✓ Linked lists
    - ✓ Stacks
    - ✓ Queues

NB. Linked lists are considered both **linear** and **non-linear** data structures depending upon the application they are used for. When used for access strategies, it is considered as a linear data-structure. When used for data storage, it is considered a non-linear data structure.

- In **non-linear data structures**, the data elements are not in sequence that means insertion and deletion are not possible in a linear manner. Non-linear data structures contain following types of data structures:
    - ✓ Tree
    - ✓ Graph

- *Homogenous and Non-homogenous data structures*: In homogenous data structures the data elements are of same type like an array. In non-homogenous data structures, the data elements may not be of same type like structure in 'C'.
- *Static and Dynamic data structures*: **Static structures** are ones whose sizes and structures associated memory location are fixed at compile time, e.g., array. **Dynamic structures** are ones that expand or shrink as required during the program execution and their associated memory location change, e.g., linked list.

1.1.1 Data Structure Operations

Algorithms manipulate the data that is represented by different data structures using various operations. The following data structure operations play major role in the processing of data:

- ✓ *Creating*. This is the first operation to create a data structure. This is just declaration and initialization of the data structure and reserved memory locations for data elements.
- ✓ *Inserting*. Adding a new data element to the data structure.
- ✓ *Updating*. It changes data values of the data structure.
- ✓ *Deleting*. Remove a data element from the data structure.
- ✓ *Traversing*. The most frequently used operation associated with data structure is to access data elements within a data structure. This form of operation is called traversing the data structure or visiting these elements once.
- ✓ *Searching*. To find the location of the data element with a given value, find the locations of all elements which satisfy one or more conditions in the data structure.
- ✓ *Sorting*. Arranging the data elements in some logical order, e.g. in ascending or descending order of students' name.
- ✓ *Merging*. Combine the data elements in two different sorted sets into a single sorted set.
- ✓ *Destroying*. This must be the last operation of the data structure and apply this operation when no longer needs of the data structure.

1.1.2 some applications of Data Structures

- ✓ Numerical analysis
- ✓ operating system
- ✓ AI
- ✓ compiler design
- ✓ database management
- ✓ graphics
- ✓ statistical analysis and
- ✓ simulation.

1.2 Abstract Data Types (ADT)

A useful tool for specifying the logical properties of a data type is the **abstract data type**. Fundamentally, a data type is a collection of values and set of operations on those values. That collection and those operations form a mathematical construct that may be implemented using a particular hardware or software with data structure.

The term ADT refers to the basic **mathematical** concept that defines the **data types** by specifying the **mathematical** and **logical** properties of data type or structures.

The abstract datatype is special kind of datatype, whose behavior is defined by a set of values and set of operations. The keyword "Abstract" is used as we can use these datatypes, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive datatypes, but operation logics are hidden.

Some examples of ADT are Stack, Queue, List etc.

The ADT is a useful guideline to implementers and a useful tool to programmers who wish to use the data type commonly.

An ADT consists of two parts:

- ✓ Value definition
- ✓ An operator definition

The **value** definition defines the collection of values for the ADT and consists of two parts: a definition clause and a condition clause.

Immediately following the value definition comes the operator definition. Each operator is defined as an abstract function with three parts: a header, the optional preconditions and the postconditions. The postcondition specifies what the operation does.

1.2.1 Abstraction

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

Chapter two

2.1 Algorithm and Algorithm analysis

Definition

An algorithm is a finite set of instructions that takes some raw data as input and transforms it into refined data. An algorithm is a tool for solving a well-specified computational problem.

2.1.1 Properties of Algorithm

Every algorithm must satisfy the following criteria:

Input: In every algorithm, there must be zero or more data that are externally supplied.

Output: At least one data is produced.

Definiteness: Each instruction must be clear and unambiguous (i.e., clearly one meaning).

Finiteness: If we trace out the instructions of an algorithm, then for all cases, the algorithm will terminate after a finite number of steps.

Effectiveness: Every instruction must be feasible and provides some basis towards the solution.

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.

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.

Efficiency: It must solve with the least amount of computational resources such as time and space

2.2 Algorithm Specification

An algorithm can be described in many ways. A natural language such as **English** can be used to write an algorithm but generally algorithms are written in English-like pseudocode that resembles with high level.

Languages such as ‘C’ and pascal. An algorithm does not follow exactly any programming language, so that it does not require any **translator**. Therefore, we can’t run algorithm with the help of compiler/interpreter, just we can dry run to check the results. The following format conventions are used to write an algorithm:

1. **Name of algorithm:** Every algorithm is given an identifying name written in capital letters.
2. **Introductory comment:** The algorithm name is followed by a brief description of the tasks the algorithm performs and any assumptions that have been made. The description gives the name and types of the variables used in the algorithm. Comment begins with // and continues until end of line. Comments specify no action and are included only for clarity.
3. **Steps:** Each algorithm is made of a sequence of numbered steps and each step has an ordered sequence of statements, which describe the tasks to be performed. The statements in each step are executed in a left-to-right order.
4. **Data type:** Data types are assumed simple such as integer, real, char, boolean and other data structures such as array, pointer, structure are used. Array i^{th} element can be described as A[i] and (i, j) element can be described as A[i, j]. Structure data type can be formed as:

```
node =record
{
        data type-1     identifier 1;
        …
        data type-n     identifier
        n; node*link;
}
```

   link is a pointer to the record type node.

5. **Assignment statement:** The assignment statement is indicated by placing equal (=) between the variable (in left hand side) and expression or value(s) (in right hand side). For example, addition of a and b is assigned in variable a.

```
a = a + b
```

6. **Expression:** There are three expressions: arithmetic, relational and logical. The arithmetic expression used arithmetic operator such as /, *, +, –, relational expression used relational operator such as
   <, <=, >, >=. <>, = and logical expression used logical operator such as not, or, and. There are two boolean values, true or false.

7. **If statement:** If statement has one of the following two forms:

(a)
```
if  condition
then
statement(s)
```

(b)
```
if  condition
then
statement(s)
else
statement(s)
```

if the condition is true, statement(s) followed then are to be executed, otherwise if condition is false, statement(s) followed else are to be executed. Here we can use multiple statements, if required.

Case statement: In general case statement has the form: Select case (expression)

case value 1: statement (s)
case value 2: statement (s)

.
.
.

case value n: statement (s)
default:

the expression is evaluated then a branch is made to the appropriate case. If the value of the expression does not match that of any case, then a branch is made to the default case.

8. **Looping statements:** These statements are used when there is a need of some statements to be executed number of times. These statements also called iterative control statements.

(a)
```
while (condition) do
{
    statement(s)
}
```

As long the condition is true, the statement(s) within while loop are executed. When the condition tests to false then while loop is exited.

(b)
```
for variable = start value to final value step increment value do
{
    statement(s)
}
```

This for loop is executed from start value to final value with addition of increment value in start value. In the absence of step clause, assume increment value 1.

(c)
```
repeat
{
statement(s)
} until (condition)
```

This repeat loop is executed until the condition is false, and exited when condition becomes true.

9. **Input and output:** For input of data it used read(variable name) statement and for output of data it used write(variable name) statement. If more than one input or output data then we can use comma as separator among the variable names.

10. **Goto statement:** The goto statement causes unconditional transfer of control to the step referenced. Thus, statement goto step N will cause transfer of control to step N.
11. **End statement:** The end statement is used to terminate an algorithm. It is usually the last step and algorithm name is written after **the end**.
12. **Functions:** A function is used to return single value to the calling function. Transfer of control and returning of the value are accomplished by return(value) statement. A function begins as follows: function function_name (parameters list).

2.3 Analysis of Algorithm concepts

Why is it necessary to analyze an algorithm?

An algorithm analysis measures the efficiency of the algorithm. The efficiency of an algorithm can be checked by:

1. *The correctness of an algorithm.*
2. *The implementation of an algorithm.*
3. *The simplicity of an algorithm.*
4. *The execution time and memory space requirements of an algorithm.*
5. *The new ways of doing the same task even faster.*

The analysis emphasizes timing analysis and then, to a lesser extent, spaces analysis. It is very difficult to calculate how fast a problem can be solved. It requires an algorithm execution machine. An algorithm can be executed only of formal models of machines e.g. Turing Machine and Random Access Machine. An algorithm analysis does not depend on computers and programming languages but the program that is coded using the algorithm depends on both. Therefore, programs written in different programming languages using the same algorithm and same computer have different time and space requirements.

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.
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

Running time is usually treated as the most important since computational time is the most precious resource in most problem domains.

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.

2.4 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.
- **Space Complexity:** Determine the approximate memory required to solve a problem of size n.

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.

There is no generally accepted set of rules for algorithm analysis. However, an exact count of operations is commonly used.

2.5 Analysis Rules:

1. We assume an arbitrary time unit.
2. Execution of one of the following operations takes time 1:
    - Assignment Operation
    - Single Input Operation

- Single Output Operation
- Single Boolean Operations
- Single Arithmetic Operations
- Function Return

3. 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.
4. Loops: Running time for a loop is equal to the running time for the statements inside the loop * number of iterations.
   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.
5. 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.

Examples:

```
1. int count()
{
int k=0;
cout<< "Enter an integer";
cin>>n;
for (i=1;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.
In the for loop:
    1 assignment, *n+1* tests, and *n* increments.
    n loops of 2 units for an assignment, and an addition.
    1 for the return statement.

T (n) = *1+1+1+(1+n+1+n)+2n+1 = 4n+6 = O(n)*

```
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 an addition.

1 for the return statement.

T (n) = *1+ (1+n+1+n)+2n+1 = 4n+4 = O(n)*

```
3. void func()
{
int x=0;
int i=1;
int 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 = 5n+5 = O(n)*

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.
n loops of 4 units for an assignment, an addition, and two multiplications.
1 for the return statement.

T (n) = *1+(1+n+1+n)+4n+1 = 6n+4 = O(n)*

2.6 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;
}}
```

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

2.7 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.

2.8 Asymptotic Analysis

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

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)

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**: The function $f(n)$ is $O(g(n))$ if there exist positive numbers c and K such that $f(n) \leq c.g(n)$ for all $n \geq K$.

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

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).

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)

Typical Orders

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	1	1	1	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:

$f(n) = (3/2)n^2+(5/2)n-3$

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)

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)).

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 ∈ 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 that or equal to g(n).

Theta Notation

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).

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)=$O(n^2)$

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)= $\Theta(n^2)$.

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).

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: $2n^2$=ω(n) but it's not ω (n^2).

Chapter 3: Simple sorting and searching algorithms

3.1 Simple sorting Algorithms

A sort can be classified as being internal if the records (or elements) that it is sorting are in main memory, or external if some of the records that it is sorting are in auxiliary or secondary storage. Example of external sorting is a telephone directory. Here a record is a collection of fields or items and a key is usually a field of the record. A file is a collection of records. The file can be sorted on one or more keys. In the example of the telephone directory, the key upon which the file is sorted is the name field of the record. Each record also contains fields for an address and telephone number.

Internal sorting methods are to be used when a file to be sorted is small enough so that the entire sort can be carried out in main memory.

Internal sorting methods:

- ✓ *Selection sort*
- ✓ *Bubble sort*
- ✓ *Insertion sort*
- ✓ *Quick sort*
- ✓ *Merge sort*
- ✓ *Heap sort*
- ✓ *Radix sort*

The time required to read and write is **not** considered to be significant in evaluating the performance of internal sorting methods. The three important efficiency criteria are:

- ✓ *use of storage space*
- ✓ *use of computer time*
- ✓ *programming effort.*

External sorting methods are employed to sort records of file which are too large to fit in the main memory of the computer. These methods involve as much as external processing (e.g. Disk I/O) as compared to processing in the CPU.
The sorting requires the involvement of external devices such as magnetic tape, magnetic disk due to the following reasons:

1. The cost of accessing an element is much higher than any computational costs.
2. Depending upon the external device, the method of access has different restrictions.

External storage devices can be categorized into two types based on access method. These are **sequential access devices** (e.g. magnetic tapes) and **random access devices** (e.g. disks).

External sorting depends to a large extent on system considerations like the type of device and the number of such devices that can be used at a time.

The lists of algorithms are K-way merge, selection tree and polyphone merging. The general method of external sorting is the **merge sort**.

3.1.1 Complexity of Sorting Algorithms

The complexity of a sorting algorithm measures the execution time as a function of the number **n** of elements to be sorted. Usually, each algorithm made the following operations:

(i) Comparisons
(ii) Swaps
(iii) Assignments

Normally, the complexity function measures only the **number of comparisons**, since the number of other operations is at most a **constant factor** of the number of comparisons.

3.1.2 Selection Sort

A selection sort is one in which successive elements are selected in order and placed into their proper sorted positions.

The selection sort implements the descending priority queue as an unordered array.
The selection sort consists entirely of a selection phase in which the largest of the remaining elements is repeatedly placed in its proper position, i, at the end of the array. The large element is swap with the element x[i].

The first iterations make n–1 comparison and second iteration make n–2, and so on. Therefore, there is a total of n–1 + n–2 + n–3 + +1 = n* (n–1)/2 comparisons, which is $O(n^2)$, although it is faster than the bubble sort. The number of interchanges is always n–1.

3.1.2.1 Selection sort analysis

Iteration 1: Find smallest value in a list of n values: n-1 comparisons Exchange values and move marker.

Iteration 2: Find smallest value in a list of n-1 numbers: n-2 comparisons Exchange values and move marker …
Iteration n-2: Find smallest value in a list of 2 numbers: 1 comparison Exchange values and move marker Total: (n-1) + (n-2) + …. + 2 + 1 = n(n-1)/2.

- **Space efficiency**: No extra **space used** (except for a few variables)
- **Time efficiency**: The best-case and worst-case are same. All input sequences need same number of comparisons. the amount of work is the same: T(n) = n(n-1)/2

3.1.3 Bubble Sort

This is the most widely known sorting by comparison method. The basic idea underlying the bubble sort is to pass through the list of elements (e.g. file) sequentially several times. Each pass consists of comparing each element in the array (or file) with its successor (x[i] with x[i+1]) and interchanging the two elements if they are not in proper order.

What is the Efficiency of the Bubble Sort?

There are n–1 iterations and n–1 comparisons on each iteration. Thus, the total number of comparisons is (n–1) * (n–1) = n^2–2n+1, which is O(n^2). Of course, the number of interchanges depends on the original order of the file. However, the number of interchanges cannot be greater than the number of comparisons.

3.1.4 Insertion Sort

Insertion sort is one that sorts a set of elements by inserting elements into an existing sorted set. The simple method for insertion sort is given below:

1. x[0] by itself is sorted as only element
2. x[1] is inserted before or after x[0] so that x[0], x[1] is sorted
3. x[2] is inserted into its proper place with x[0], x[1], so that x[0], x[1], x[2] is sorted and so on up to x[n–1] is inserted into proper place so that x[0], x[1], x[2]…..x[n–1] is sorted.

If the initial file is sorted, only one comparison is made on each iteration, so that the sort is O(n).
If the set of elements is initially sorted to the reverse order, the sort is O(n^2), since the total number of comparison is n–1 + n–2 +3+2+1 = n*(n–1)/2 = O(n^2). However, the **insertion sort** is still usually better than the **bubble sort**.

The **space** requirements for the sort consists of only a temporary variable, **temp**.

The **speed** of the sort is improved using a **binary search** to find the proper position for x[i] in the sorted set. Another improvement can be made using **array link of pointers**. This reduces the **time** requirement for insertion but **not the time** required for searching for the proper position. The space or memory requirements are also increased because of extra link array. This sort is called **list insertion sort** and may be viewed as a general selection sort in which the priority queue is implemented by an ordered list.

Both **selection sort** and **insertion sort** are more efficient than bubble sort. Selection sort requires **fewer** assignments than insertion sort but more **comparisons**.

3.1.5 Space Complexity of Various Sorting Algorithms

The input space for all sorting algorithms is at least O(n), where n is the size of the array. It is also important to understand the auxiliary space being used by that algorithm.

1. **Bubble sort**: The sorting is done in place and there is generally one extra variable used to store the temporary location for swapping successive items. Hence, the auxiliary space used is O(1).
2. **Insertion sort**: The case is the same as bubble sort. The sorting is done in place with a constant number of extra variables, so the auxiliary space used in O(1).
3. **Merge sort**: In merge sort, we split the array into two and merge the two sub-arrays by using a temporary array. The temporary array is the same size as the input array, hence, the auxiliary space required is O(n).
4. **Heap sort**: Heap sort is also implemented in place and hence the auxiliary space required is O(1).
5. **Quick sort**: Depending on how you implement quick sort, generally you would need O(log(n)) additional space to sort an array

3.2 Simple searching Algorithms

Introduction to searching algorithms
Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in data structure is listed below:

1. Linear Search
2. Binary Search

3.2.1 Linear search

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found. It compares the element to be searched with all the elements present in the array and when the element is matched successfully, it returns **the index** of the element in the array, else it **returns -1**. Linear Search is applied on unsorted or unordered lists, when there are **fewer** elements in a list.

Features of Linear Search Algorithm

- ✓ It is used for unsorted and unordered small list of elements.
- ✓ It has a time complexity of O(n), which means the time is linearly dependent on the number of elements, which is not bad, but **not** that good too.
- ✓ It has a very simple implementation.

3.2.2 Binary search

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.

2. If we get a match, we return the index of the middle element.

3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.

4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element (as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.

5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

Binary Search is useful when there are large number of elements in an array and they are sorted.

So, a necessary condition for Binary search to work is that the list/array should be sorted. Features of Binary Search

1. It is great to search through large sorted arrays.

2. It has a time complexity of O(log n) which is a very good time complexity. It has a simple implementation. Binary search is **a fast search** algorithm with run-time complexity of O(log n). This search algorithm works on the principle of **divide and conquers**.

Chapter 4: Linked Lists

4.1 Review on pointer and introduction

Linked lists and arrays are similar since they both store collections of data. Array is the most common data structure used to store collections of elements. Arrays are convenient to declare and provide the easy syntax to access any element by its index number. Once the array is set up, access to any element is convenient and fast.
The disadvantages of arrays are: The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required. Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room. Deleting an element from an array is not possible.
Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. Generally, arrays allocate the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.
Here is a quick review of the terminology and rules of pointers.
The linked list code will depend on the following functions:
malloc () is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype of malloc () and other heap functions are in stdlib.h.
malloc () returns NULL if it cannot fulfill the request. It is defined by: void *malloc (number_of_bytes)

Since a void * is returned the C standard states that this pointer can be converted to any type. For example, char *cp; cp = (char *) malloc (100); Attempts to get 100 bytes and assigns the starting address to cp. We can also use the sizeof() function to specify the number of bytes.
For example, int *ip; ip = (int *) malloc (100*sizeof(int));
free () is the opposite of malloc (), which de-allocates memory. The argument to free () is a pointer to a block of memory in the heap a pointer which was obtained by a malloc () function. The syntax is: free (ptr); The advantage of free () is simply memory management when we no longer need a block.

4.1.1 Linked List Concepts:

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list. The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires an additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

Comparison between array and linked list:

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

Trade offs between linked lists and arrays:

FEATURE	ARRAYS	LINKED LISTS
Sequential access	efficient	efficient
Random access	efficient	inefficient
Resigning	inefficient	efficient
Element rearranging	inefficient	efficient
Overhead per elements	none	1 or 2 links

Types of Linked Lists:

Basically, we can put linked lists into the following four items:

1. Single Linked List.
2. Double Linked List.
3. Circular Linked List.
4. Circular Double Linked List.

A single linked list is one in which all nodes are linked together in some **sequential manner.** Hence, it is also called **as linear linked list.**

A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore, each node in a double linked list has **two link** fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has **no beginning and no end**. A single linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node. A circular double linked list is one, which has both the **successor pointer** and **predecessor pointer** in the **circular manner.**

4.2 Single Linked List.

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated in the heap using **malloc (),** so the node memory continues to exist until it is explicitly de-allocated using **free ().** The front of the list is a pointer to the "start" node.

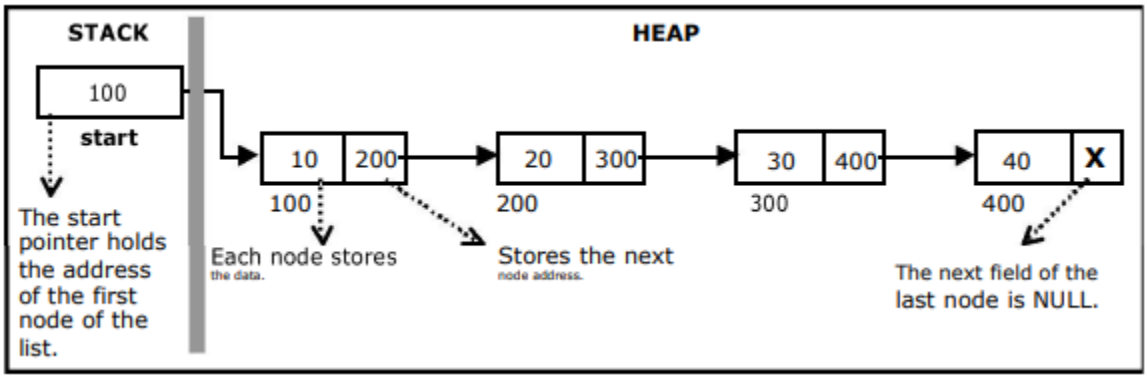


Figure: Singly linked list

The beginning of the linked list is stored in a "start" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to **mark the end** of the list. Code can access any node in the list by starting at the start and following the next pointers. The start pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.

4.2.2 Implementation of Single Linked List:

Before writing the code to build the above list, we need to create a start node, used to create and access other nodes in the linked list. The following structure definition will do (see figure 4): Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure. Initialize the start pointer to be NULL.

```
struct slinklist
{
        int data;
        struct slinklist* next;
};

typedef struct slinklist node;

node *start = NULL;
```

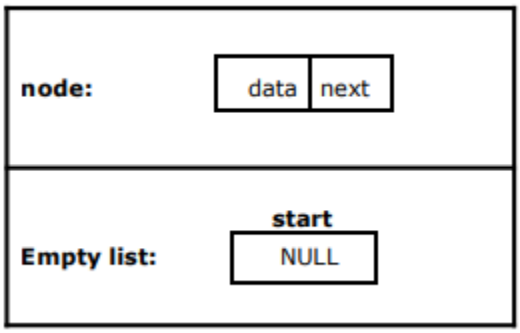


Figure: Structure definition, single link node and Empty list

The basic operations in a single linked list are:

- ✓ Creation.
- ✓ Insertion.
- ✓ Deletion.
- ✓ Traversing.

4.2.3 Creating a node for Single Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the **malloc ()** function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user, set next field to NULL and finally returns the address of the node. Figure 3.3 illustrates the creation of a node for single linked list.

```
node* getnode()
{
      node* newnode;
      newnode = (node *) malloc(sizeof(node));
      printf("\n Enter data: "); scanf("%d",
      &newnode -> data);
      newnode -> next = NULL;
      return newnode;
}
```

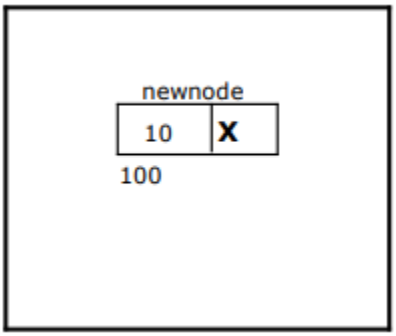


Figure: New node with the value of 10

4.2.4 Insertion of a Node:

One of the most primitive operations that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node (in a similar way that is done while creating a list) before reading the data. The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL.

The new node can then be inserted at three different places namely: Inserting a node at the beginning.

- ✓ Inserting a node at the end.
- ✓ Inserting a node at intermediate position.
- ✓ Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- ➢ Get the new node using getnode(). newnode = getnode();
- ➢ If the list is empty then start = newnode.
- ➢ If the list is not empty, follow the steps given below:

newnode -> next = start;
start = newnode;

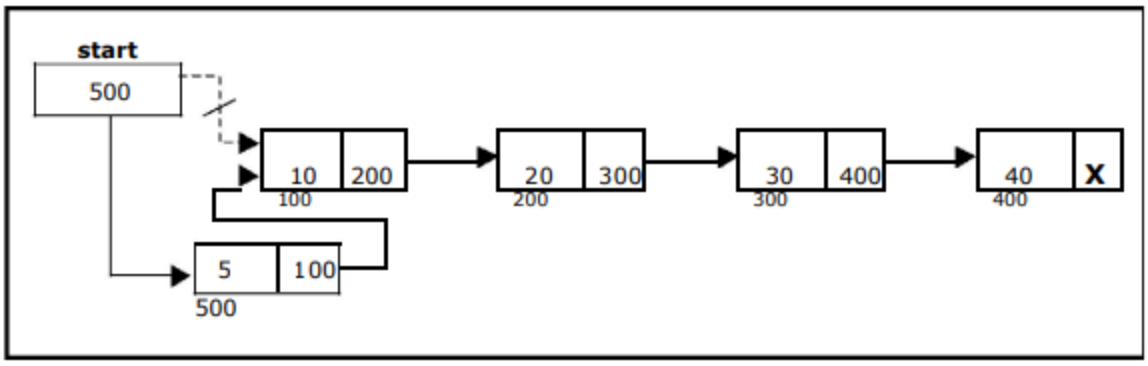


Figure: shows inserting a node into the single linked list at the beginning.

The function insert_at_beg(), is used for inserting a node at the beginning

```
void insert_at_beg()
{
node *newnode; newnode = getnode();
if(start == NULL)
{
start = newnode;
}
else {
newnode -> next = start;
start = newnode;
}
}
```

4.2.4.1 Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using getnode()

newnode = getnode();

- If the list is empty then start = newnode.
- If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> next != NULL)
temp = temp -> next;
temp -> next = newnode;
```

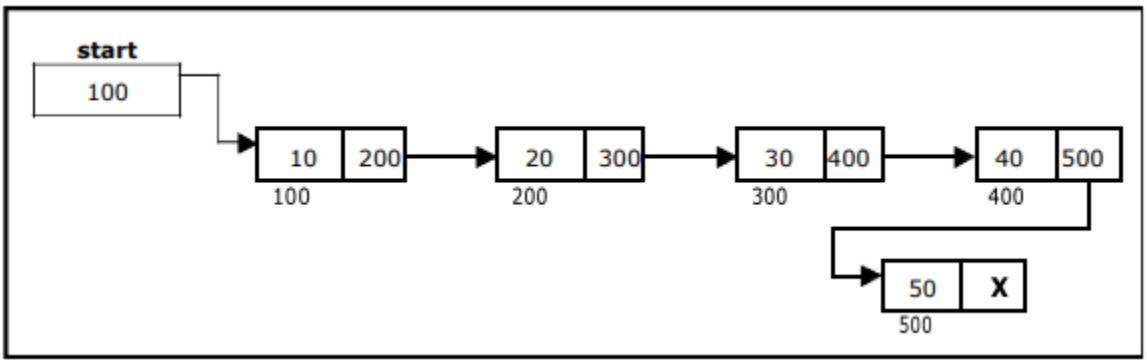


Figure: shows inserting a node into the single linked list at the beginning.

The function insert_at_end(), is used for inserting a node at the end.

```
void insert_at_end()
{
node *newnode, *temp;
newnode = getnode();
if(start == NULL)
{
start = newnode;
}
else
{
temp = start;
while(temp -> next != NULL)
temp = temp -> next;
temp -> next = newnode;
}
}
```

4.2.4.2 Inserting a node at intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using getnode().

Newnode = getnode();

- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by **countnode() function.**
- Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
- After reaching the specified position, follow the steps given below:

prev -> next = newnode;
newnode -> next = temp;

- ❖ Let the intermediate position be 3.

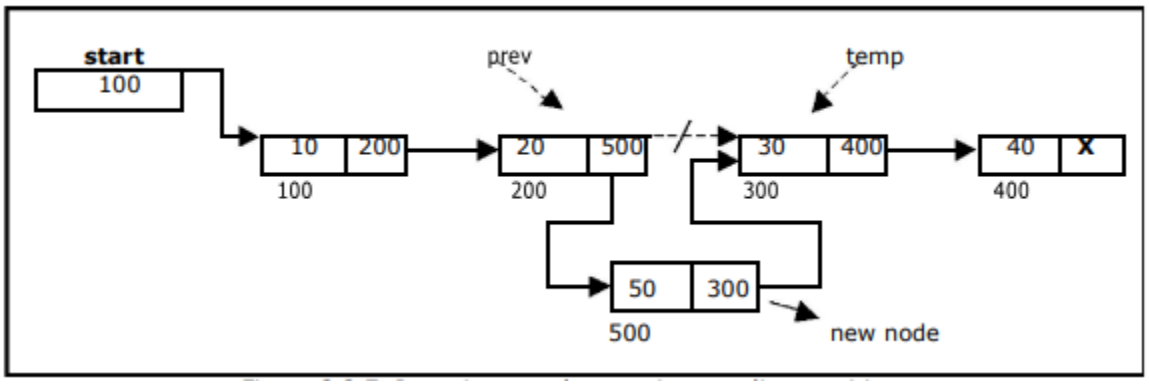


Figure: shows inserting a node into the single linked list at an intermediate position

4.2.5 Deletion of a node:

Another primitive **operation** that can be done in a singly linked list is the **deletion** of a node. Memory is to be released for the node to be deleted.

A node can be deleted from the list from three different places namely.

- ✓ Deleting a node at the beginning.
- ✓ Deleting a node at the end.
- ✓ Deleting a node at intermediate position.

4.2.5.1 Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

If the list is not empty, follow the steps given below:

temp = start;

start = start -> next;

free(temp);

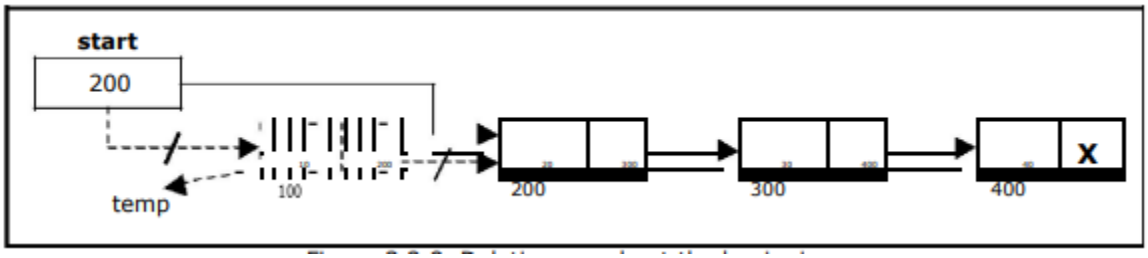


Figure: Deleting a node at the beginning

The function delete_at_beg(), is used for deleting the first node in the list.

```
void delete_at_beg()
{
node *temp;
if(start == NULL)
{
printf("\n No nodes are exist..");
return ;
}
else
{
temp = start;
start = temp -> next;
free(temp);
printf("\n Node deleted ");
}
}
```

4.2.5.1 Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

If the list is not empty, follow the steps given below:

```
temp = start;
prev = start;
while(temp -> next != start)
{
```

prev = temp;
temp = temp -> next;
}
prev -> next = start;
After deleting the node, if the list is empty then start = NULL. The function cll_delete_last(), is used for deleting the last node in the list.

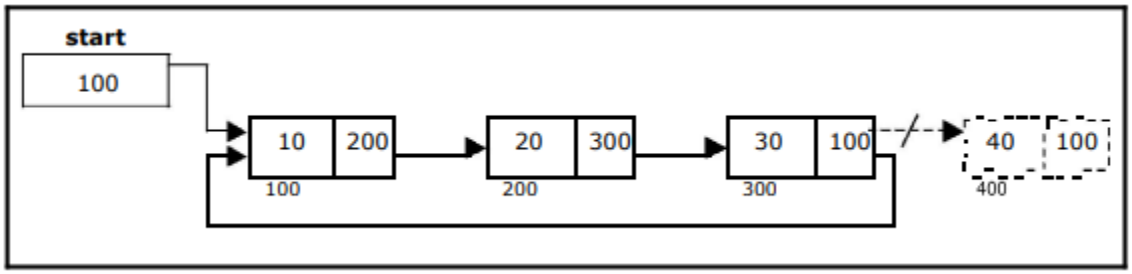


Figure: Deleting a node at the end

4.3 Double Linked List:

A double linked list is a two-way list in which all nodes will have **two links**. This helps in accessing both **successor node** and **predecessor node** from the given node position. It provides bi-directional traversing. Each node contains three fields: **Left link**. **Data. Right link**. The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data. Many applications require searching forward and backward thru nodes of a list. For example, searching for a name in a telephone directory would need forward and backward scanning thru a region of the whole list.
The basic operations in a double linked list are:

- ✓ Creation
- ✓ Insertion
- ✓ Deletion
- ✓ Traversing

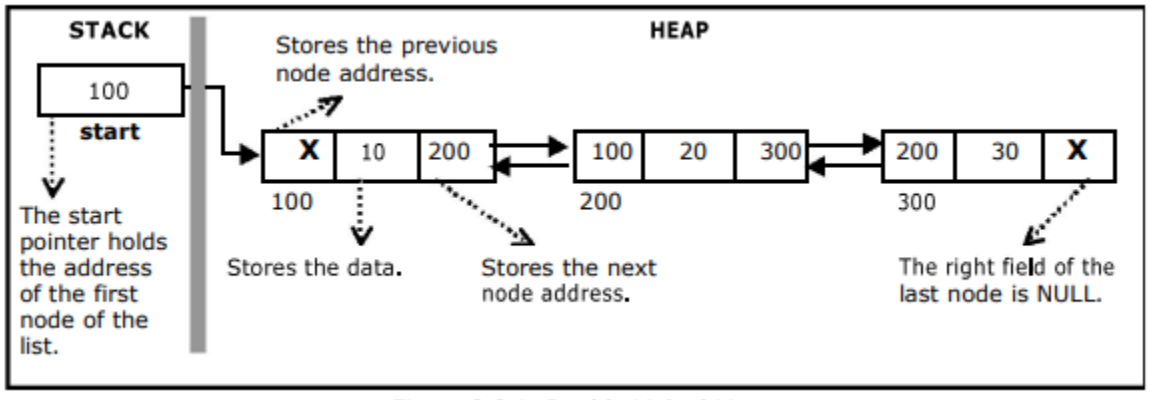


Figure: Double linked list

The beginning of the double linked list is stored in a "start" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.
The following code gives the structure definition:

```
struct dlinklist
{
        struct dlinklist *left;
        int data;
        struct dlinklist *right;

};
typedef struct dlinklist node;
node *start = NULL;
```

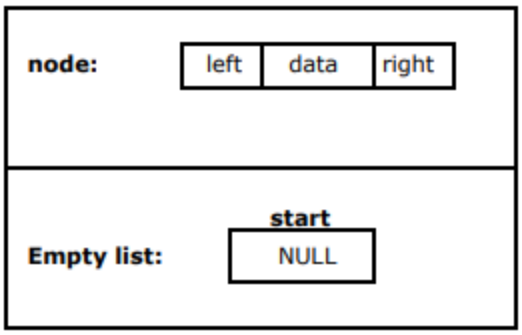


Figure: Double linked list

4.3.1 Creating a node for Double Linked List:

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc () function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user and set left field to NULL and right field also set to NULL.

```
node* getnode()
{
      node* newnode;
      newnode = (node *) malloc(sizeof(node));
      printf("\n Enter data: ");
      scanf("%d", &newnode -> data);
      newnode -> left = NULL;
      newnode -> right = NULL;
      return newnode;
}
```

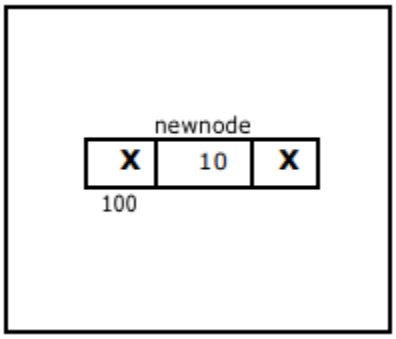


Figure: Creating new node with the value of 10.

4.3.2 Creating a doubly linked list with "n" number of nodes

The following steps are followed to create "n" number of nodes.

- Get the new node using

getnode(). newnode =getnode();

- If the list is empty then start = newnode.
- If the list is not empty, follow the steps given below:
  - ✓ The left field of the new node is made to point the previous node.
  - ✓ The previous nodes right field must be assigned with address of the new node.
- Repeat the above steps n times

The function createlist(), is used to create 'n' number of nodes:

```
void createlist(int n)
{
        int i;
        node * new node;
        node *tem p;
        for(i = 0; i < n; i+ +)
        {
                new node = getnode();
                if(start = = NULL)
                {
                        start = new node;
                }
                else
                {
                        temp = start;
                        while(temp - > right)
                                temp = temp - > right;
                        tem p - > right = new no de;
                        new node - > left = temp;
                }
        }
}
```

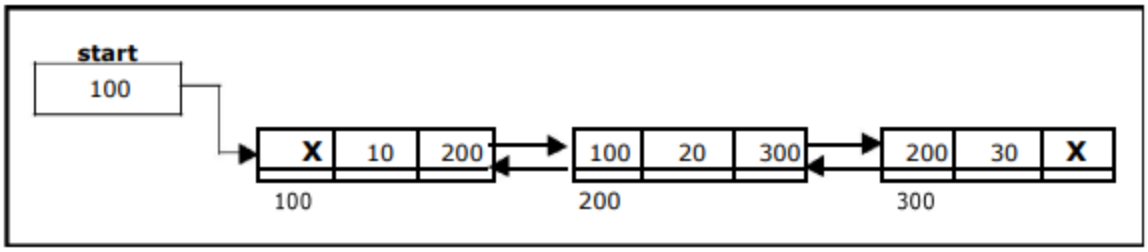


Figure: Double linked list with 3 nodes

4.3.3 Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using getnode().

newnode=getnode();

- If the list is empty then start = newnode.
- If the list is not empty, follow the steps given below:

newnode -> right = start;
start -> left = newnode;
start = newnode;
The function dbl_insert_beg(), is used for inserting a node at the beginning.

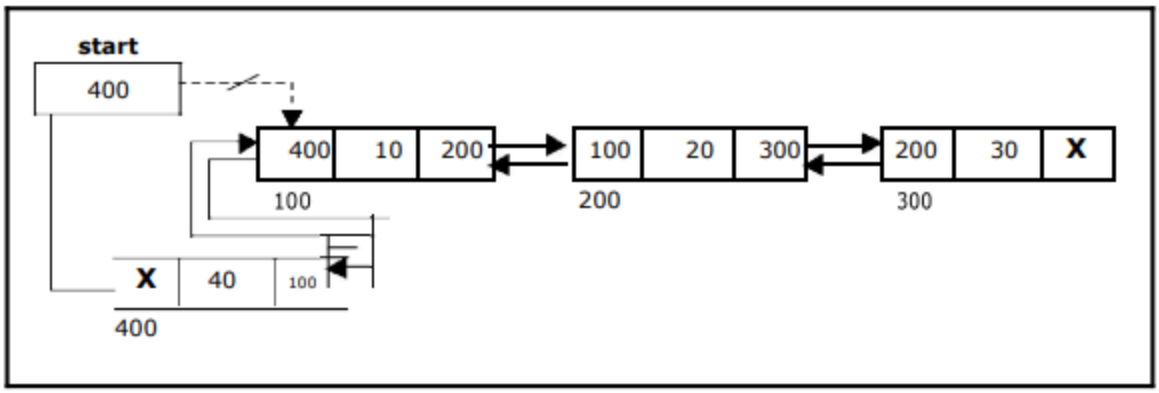


Figure: Inserting a node at the beginning

4.3.4 Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using *getnode()*

newnode=getnode();

- If the list is empty then *start = newnode.*
- If the list is not empty follow the steps given below:

temp = start;
while(temp -> right != NULL)
temp = temp -> right;
temp -> right = newnode;
newnode -> left = temp;

- The function dbl_insert_end(), is used for inserting a node at the end.

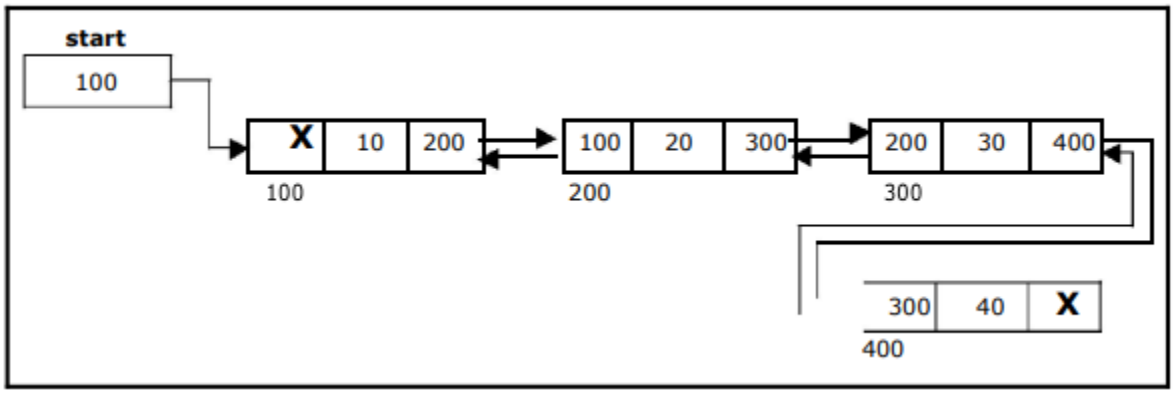


Figure: Inserting a node at the end

4.3.5 Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using *getnode().*

newnode=getnode();

- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.
- Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
- After reaching the specified position, follow the steps given below:

newnode -> left = temp;
newnode -> right = temp -> right;

temp -> right -> left = newnode;
temp -> right = newnode;

- The function dbl_insert_mid(), is used for inserting a node in the intermediate position.

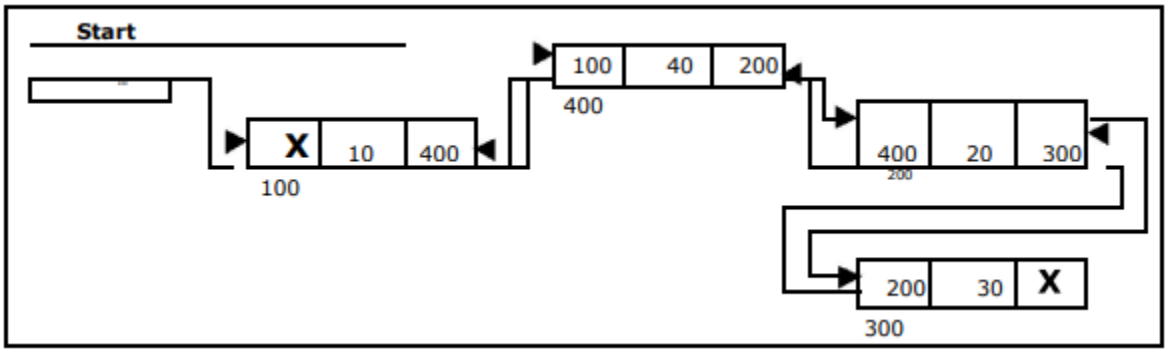


Figure: Inserting a node at an intermediate position

4.3.6 Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If the list is empty display "Empty List" message
- If the list is not empty, follow the steps given below:

temp = start;
start = start -> right;
start -> left = NULL;
free(temp);

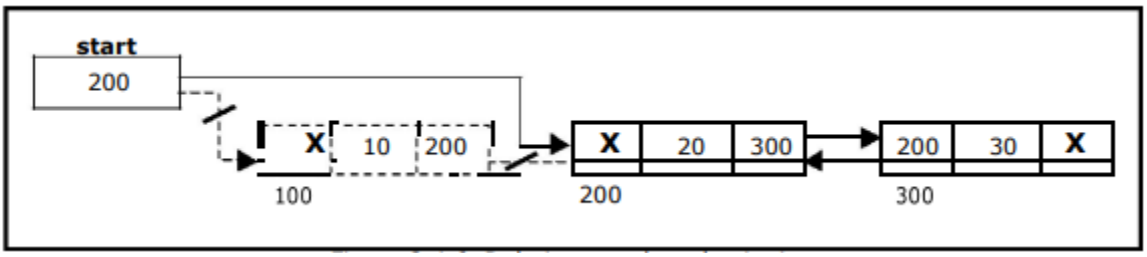


Figure: Deleting a node at the beginning

4.3.7 Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If the list is empty display "Empty List" message
- If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
{
temp = temp -> right;
}
temp -> left -> right = NULL;
free(temp);
```

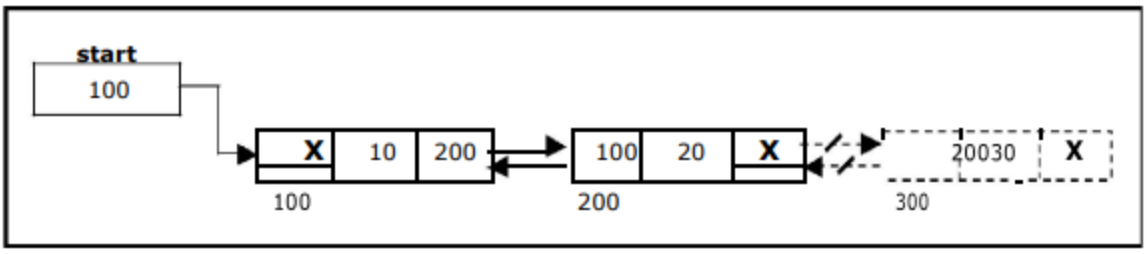


Figure: Deleting a node at the end

4.3.8 Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two nodes).

- If the list is empty display "Empty List" message
- If the list is not empty, follow the steps given below:
  - ✓ Get the position of the node to delete.
  - ✓ Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
  - ✓ Then perform the following steps:

```
if(pos > 1 && pos < nodectr)
{
temp = start;
i = 1;
while(i < pos)
{
temp = temp -> right;
i++;
}
temp -> right -> left = temp -> left;
temp -> left -> right = temp -> right;
free(temp);
printf("\n node deleted..");
}
```

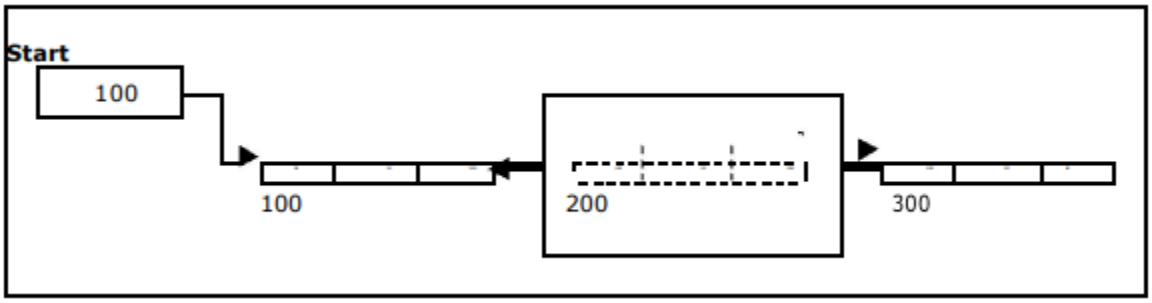


Figure: Deleting a node at an intermediate position

4.4 Circular Single Linked List*:*

It is just a single linked list in which the link field of the last node points back to the address of the **first node**. A circular linked list has **no beginning and no end**. It is necessary to establish a special pointer called start pointer always pointing to the first node of the list. Circular linked lists are **frequently used instead** of ordinary linked list because many operations are much

easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

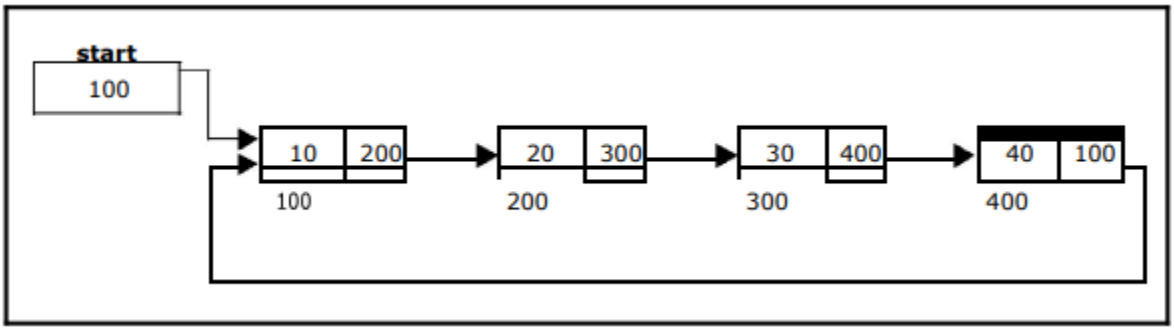


Figure: Circular single linked list

The basic operations in a circular single linked list are:

- ✓ Creation.
- ✓ Insertion.
- ✓ Deletion.
- ✓ Traversing.

4.5 Circular Double Linked List:

A circular double linked list has both **successor pointer** and **predecessor pointer** in circular manner. The objective behind considering circular double linked list is to simplify the insertion and deletion operations performed on double linked list. In circular double linked list, the right link of the right most node points back to the start node and left link of the first node points to the last node.

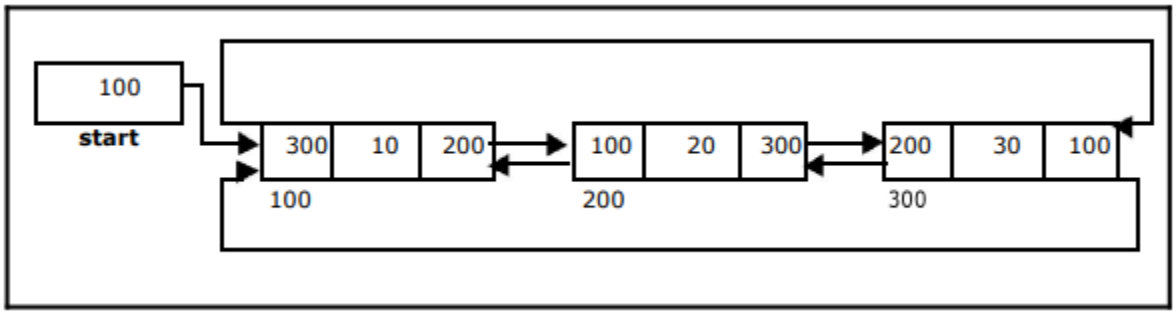


Figure: Circular double linked list

The basic operations in a circular double linked list are:

- ✓ Creation.
- ✓ Insertion.
- ✓ Deletion.
- ✓ Traversing.

Comparison of Linked List Variations:

Summary

The major disadvantage of doubly linked lists (over singly linked lists) is that they require more space (every node has two pointer fields instead of one). Also, the code to manipulate doubly linked lists needs to maintain the prev fields as well as the next fields; the more fields that have to be maintained, the more chance there is for errors. The major advantage of doubly linked lists is that they make some operations (like the removal of a given node, or a right-to-left traversal of

the list) more efficient. The major advantage of circular lists (over non-circular lists) is that they eliminate some extra-case code for some operations (like deleting last node). Also, some applications lead naturally to circular list representations. For example, a computer network might best be modeled using a circular list.

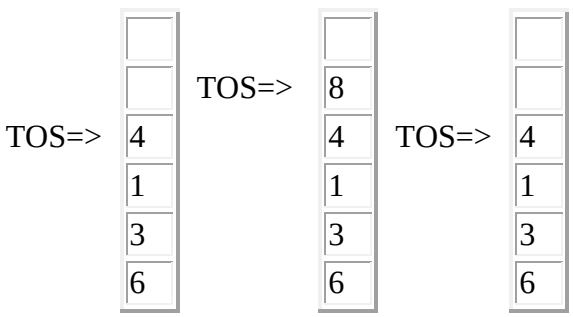
Chapter 5: Stacks and Queues

5.1 STACK :

"A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top".* stacks are sometimes referred to as Last In First Out (LIFO) lists

Stacks have some useful terminology associated with them:

- **Push** To add an element to the stack
- **Pop** To remove an element from the stock
- **LIFO** Refers to the last in, first out behavior of the stack
- **FILO** Refers to the first in, last out Equivalent to LIFO



Implementation:
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.

5.1.1. The Basic Operations:

Push()

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

Pop()

```
{
       if stack not empty {
             return the value of the top item
```

```
            remove the top item from the stack
                        }
        else {
            give an error message
            }
}
```

5.1.2 Array Implementation of Stacks:

5.1.2.1 The PUSH operation

Here, as you might have noticed, 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.

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:

```
int                                                 stack[UPPERLIMIT];
int          top=          -1;          /*stack          is          empty*/
..
..
main()
{
..
..
push(item);
..
..
}

push(int                                                            item)
{
                top          =          top          +          1;
    if(top < UPPERLIMIT)
        stack[top]          =          item;          /*step-1          &          2*/
                                                                    else
        cout<<"Stack                                          Overflow";
}
```

Note: - In array implementation, we have taken TOP = -1 to signify the empty stack, as this simplifies the implementation.

5.1.2.2 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.

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:

```
int stack[UPPPERLIMIT];
int top=-1;
..
..
main()
{
..
..
poped_val = pop();
..
..
}

int pop()
{
int del_val ;
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.

5.2 Applications of Stacks

5.2.1 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'? ie 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-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.

5.3 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 * +**

The valuable aspect 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 (see - 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).

Purpose

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

Postfix Evaluation

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

```
initialise 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
  }
}
```

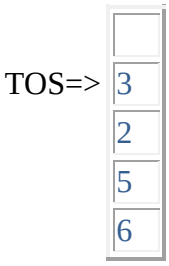
The single value on the stack is the desired result.

Binary operators: **+**, **-**, ***, /**, etc.,

Unary operators: **unary minus**, **square root**, **sin**, **cos**, **exp**, etc.,

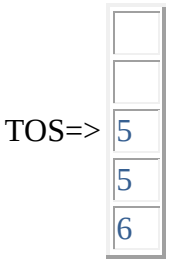
So for **6 5 2 3 + 8 * + 3 + ***

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

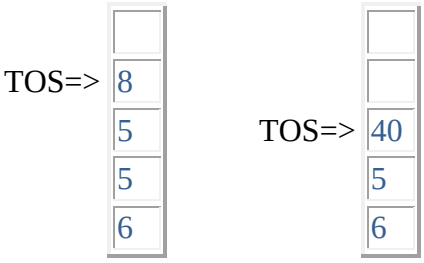


the remaining items are now: + 8 * + 3 + *

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:

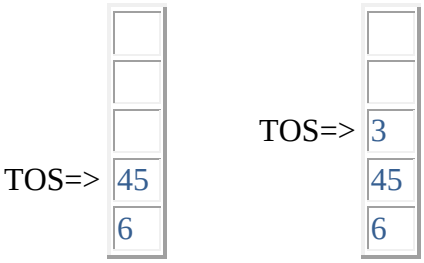


Next 8 is pushed and the next item is the operator *:



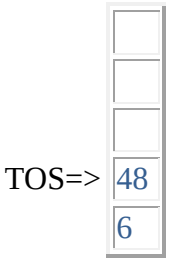
(8, 5 popped, 40 pushed)

Next the operator + followed by 3:

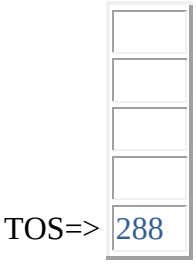


(40, 5 popped, 45 pushed, 3 pushed)

Next is operator +, so 3 and 45 are popped and 45+3=48 is pushed



Next is operator *, so 48 and 6 are popped, and 6*48=288 is pushed



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.

5.3.1 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

```
initialise stack and postfix output to empty;
while(not end of infix expression) {
  get next infix item
  if(item is value) append item to pfix 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

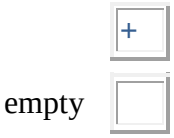
2 : / *****

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.

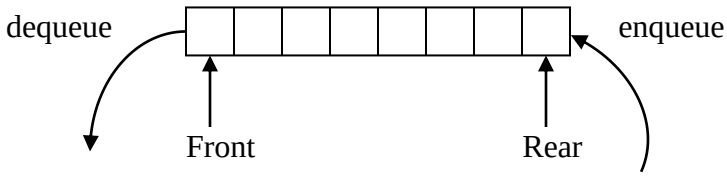
eg., consider the infix expression **a+b*c+(d*e+f)*g**

	Stack	**Output**
TOS=>	+	**ab**
TOS=>	* +	**abc**
TOS=>	+	**abc*+**
TOS=>	* (+	**abc*+de**
TOS=>	+ (+	**abc*+de*f**
TOS=>	+	**abc*+de*f+**
TOS=>	*	**abc*+de*f+g**



5.4 Queue

- A data structure that has access to its data at the front and rear.
- Operates on FIFO (Fast In First Out) basis.
- **LILO** Refers to the Last in, last out Equivalent to FIFO
- Uses two pointers/indices to keep tack of information/data.
- has two basic operations:
    - enqueue - inserting data at the rear of the queue
    - dequeue – removing data at the front of the queue



Example:

Operation	Content of queue
Enqueue(B)	B
Enqueue(C)	B, C
Dequeue()	C
Enqueue(G)	C, G
Enqueue (F)	C, G, F
Dequeue()	G, F
Enqueue(A)	G, F, A
Dequeue()	F, A

5.4.1 Simple array implementation of enqueue and dequeue operations

Analysis:

Consider the following structure: int Num[MAX_SIZE];
We need to have two integer variables that tell:
- the index of the front element
- the index of the rear 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

Implementation:

```
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--;

    }
    else
        cout<<"Queue Underflow";
    return(x);
```

}

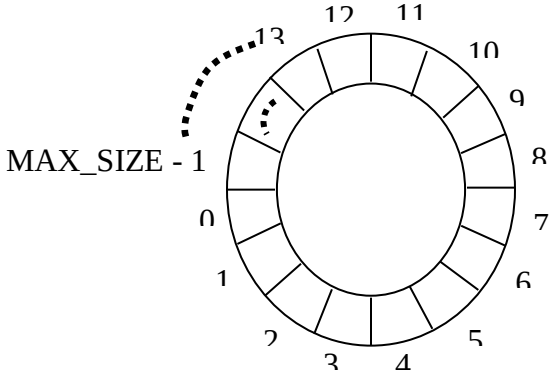
5.4.2 Circular array implementation of enqueue and dequeue operations

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.

Example: Consider a queue with MAX_SIZE = 4

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:



Analysis:

Consider the following structure: int Num[MAX_SIZE];

We need to have two integer variables that tell:

- the index of the front element

- the index of the rear 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

Implementation:

```
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--;

    }
    else
        cout<<"Queue Underflow";
    return(x);
}
```

5.4.3. Priority Queue

- is a queue where each data has an associated key that is provided at the time of insertion.
- Dequeue operation deletes data having highest priority in the list
- One of the previously used dequeue or enqueue operations has to be modified

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

Thus, in the above example the implementation of the dequeue operation need to be modified.

5.4.3.1 Demerging Queues

- is the process of creating two or more queues from a single queue.
- used to give priority for some groups of data

Example: The following two queues can be created from the above priority queue.

Aster	Meron
Female	Female

Abebe	Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male	Male

Algorithm:

```
create empty females and males queue
while (PriorityQueue is not empty)
{
  Data=DequeuePriorityQueue(); // delete data at the front
  if(gender of Data is Female)
  EnqueueFemale(Data);
else
  EnqueueMale(Data);
}
```

5.4.3.2 Merging Queues

- is the process of creating a priority queue from two or more queues.
- the ordinary dequeue implementation can be used to delete data in the newly created priority queue.

Example: The following two queues (females queue has higher priority than the males queue) can be merged to create a priority queue.

Aster	Meron
Female	Female

Abebe	Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male	Male

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

5.5 Application of Queues

i. Print server- maintains a queue of print jobs

ii. Disk Driver- maintains a queue of disk input/output requests

iii. Task scheduler in multiprocessing system- maintains priority queues of processes

iv. Telephone calls in a busy environment –maintains a queue of telephone calls

v. Simulation of waiting line- maintains a queue of persons

Chapter 6: Tree structures

6.1 Binary tree and binary search trees

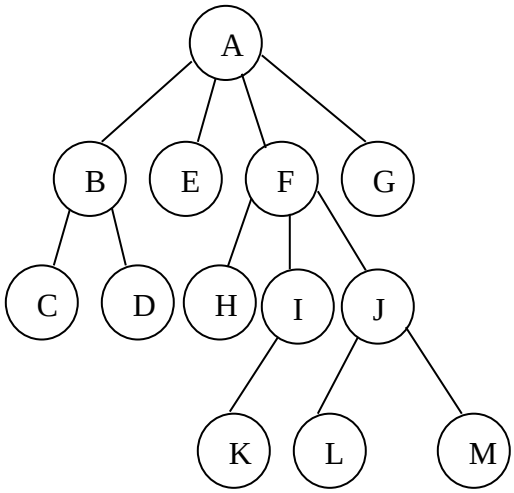
6.1.1 Trees

A tree is a set of nodes and edges that connect pairs of nodes. It is an abstract model of a hierarchical structure. Rooted tree has the following structure:

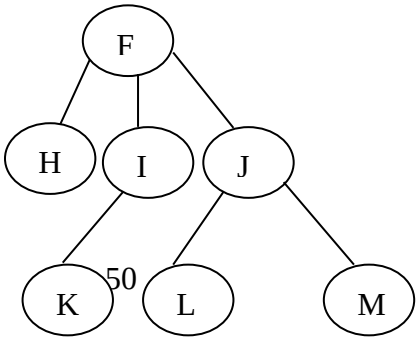
- One node distinguished as root.
- Every node C except the root is connected from exactly other node P. P is C's parent, and C is one of C's children.
- There is a unique path from the root to the each node.
- The number of edges in a path is the length of the path.

6.1.1.1 Tree Terminologies

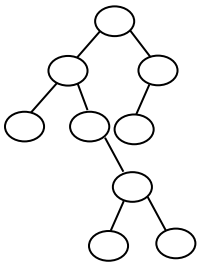
Consider the following tree.



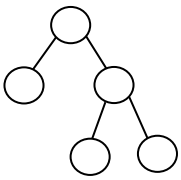
Root: a node without a parent. → A
Internal node: a node with at least one child. →A, B, F, I, J
External (leaf) node: a node without a child.→ C, D, E, H, K, L, M, G
Ancestors of a node: parent, grandparent, grand-grandparent, etc of a node.
Ancestors of K → A, F, I
Descendants of a node: children, 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
Subtree: a tree consisting of a node and its descendants.



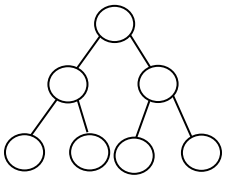
Binary tree: a tree in which each node has at most two children called left child and right child.



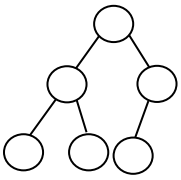
Full binary tree: a binary tree where each node has either 0 or 2 children.



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.



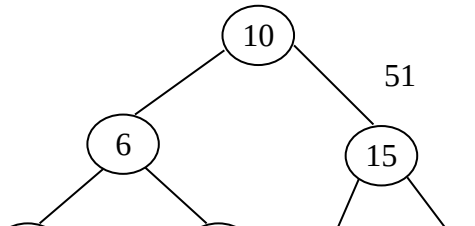
Complete binary tree: a binary tree in which the length from the root to any leaf node is either h or h-1 where h is the height of the tree. The deepest level should also be filled from left to right.



6.1.2 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 subtrees are also binary search trees.



6.1.3 Data Structure of a Binary Tree

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

6.2 Operations on Binary Search Tree

Consider the following definition of binary search tree.

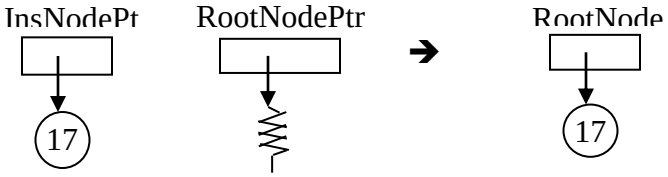
```
struct Node
{
        int Num;
        Node * Left, *Right;
};
Node *RootNodePtr=NULL;
```

6.2.1 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.

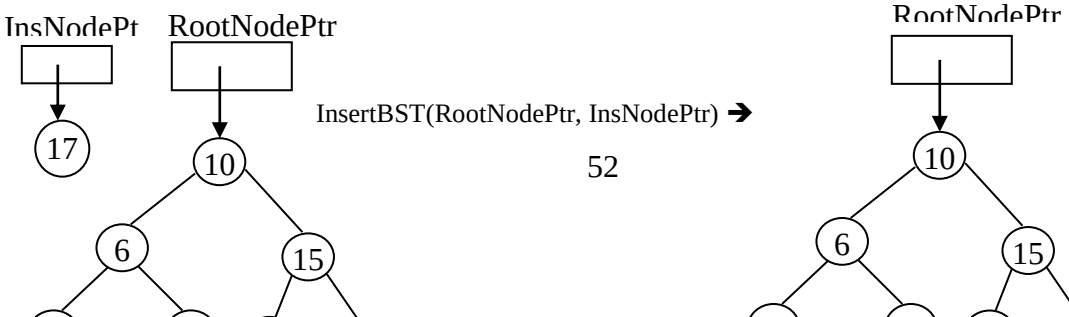
Case 1: There is no data in the tree (i.e. RootNodePtr is NULL)

- 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);
        }
}
```

6.2.3 Searching

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*;

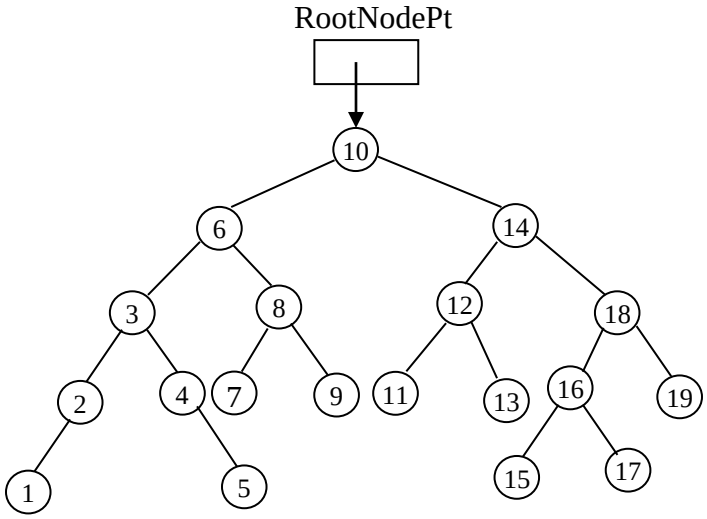
Implementation:

```
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));
}
```

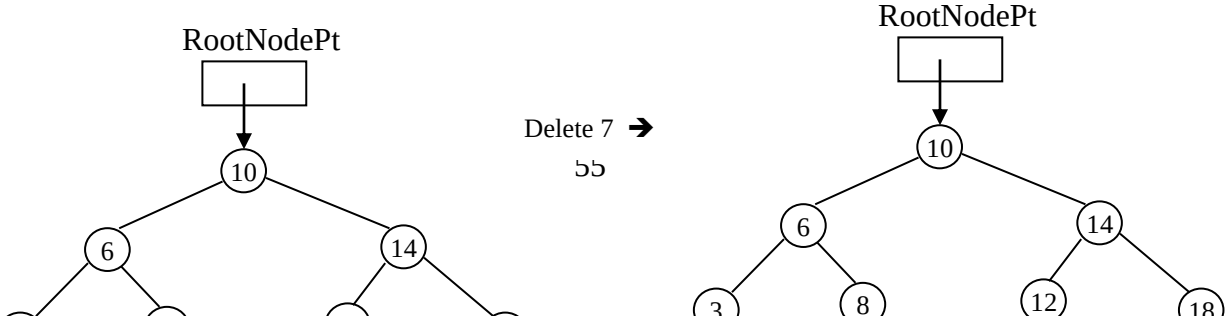
6.2.4 Deletion

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.



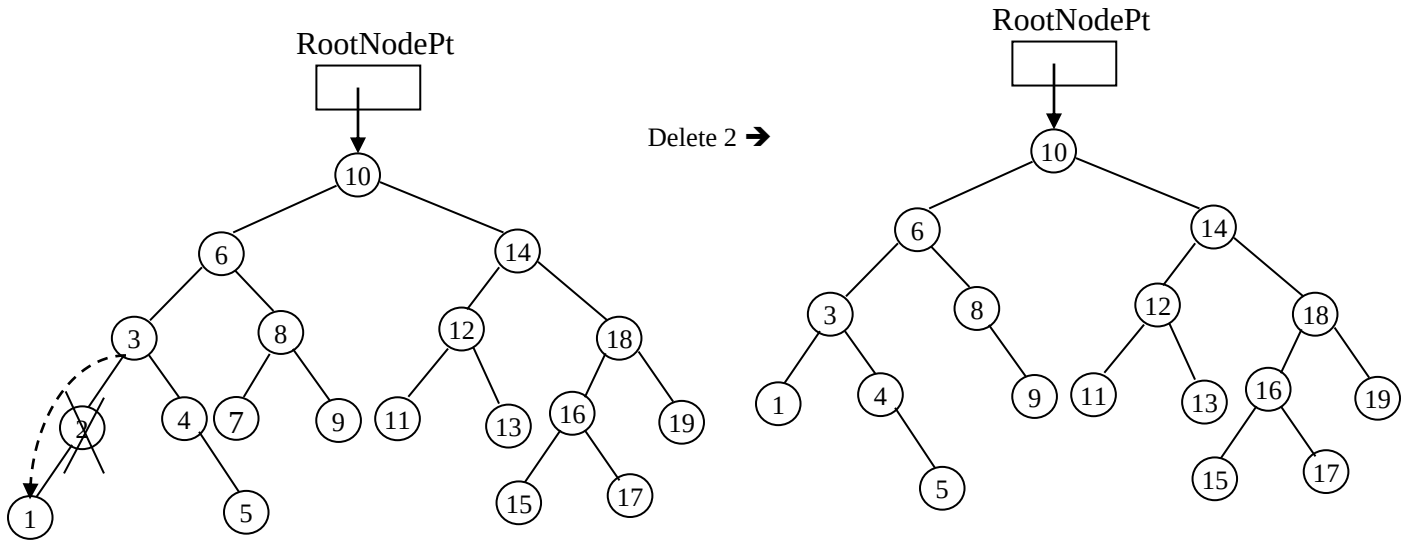
Case 1: Deleting a leaf node (a node having no child), e.g. 7



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

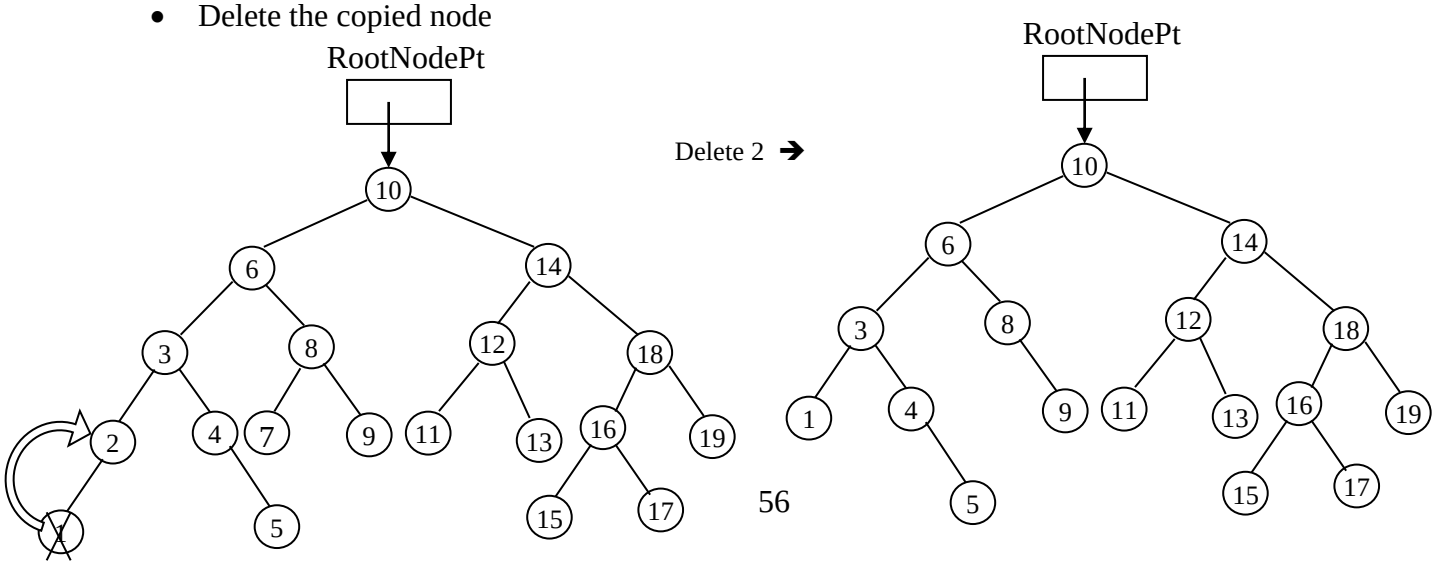
Approach 1: Deletion by merging – one of the following is done

- 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

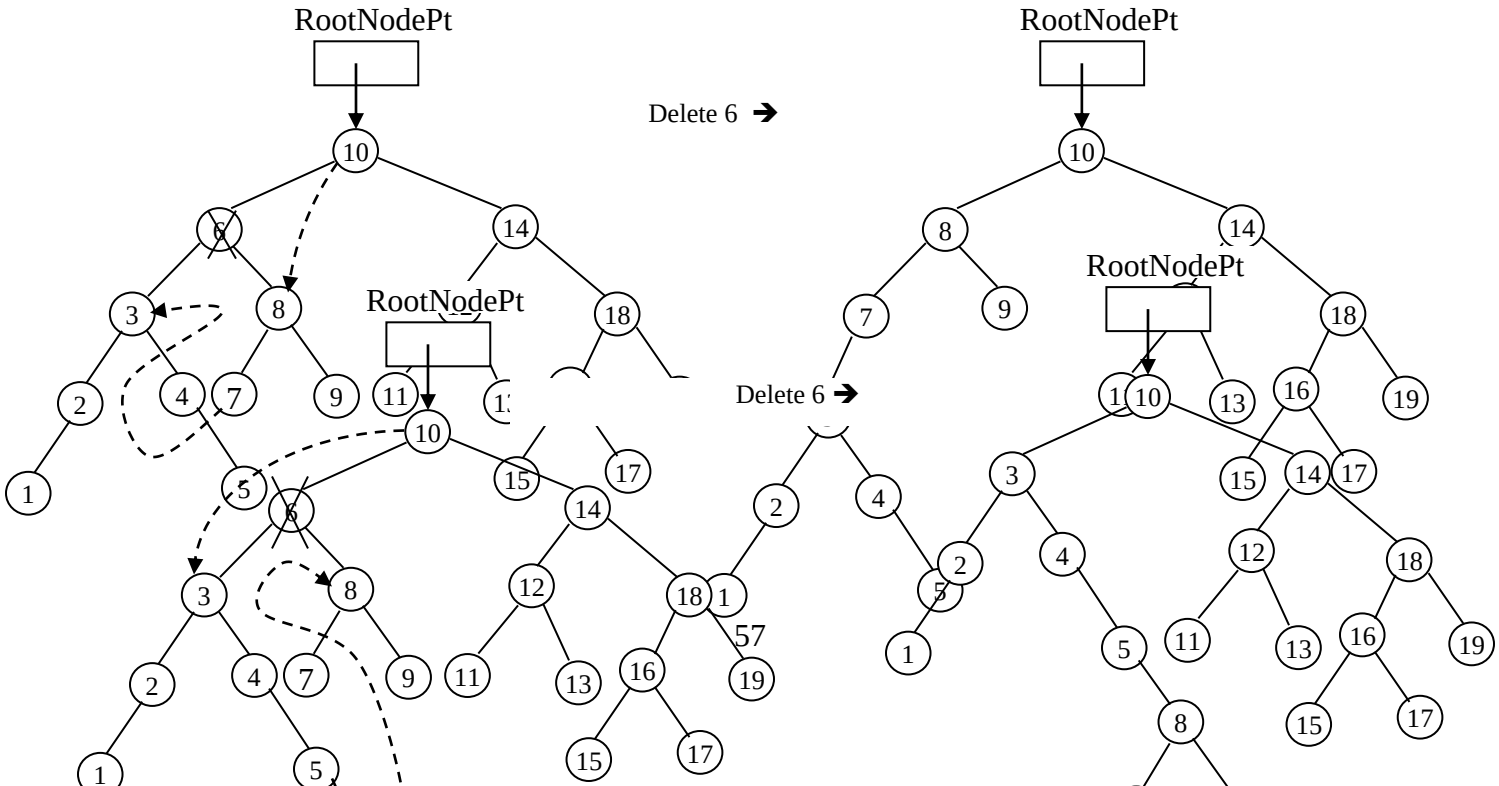
Approach 1: Deletion by merging – one of the following is done

- 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

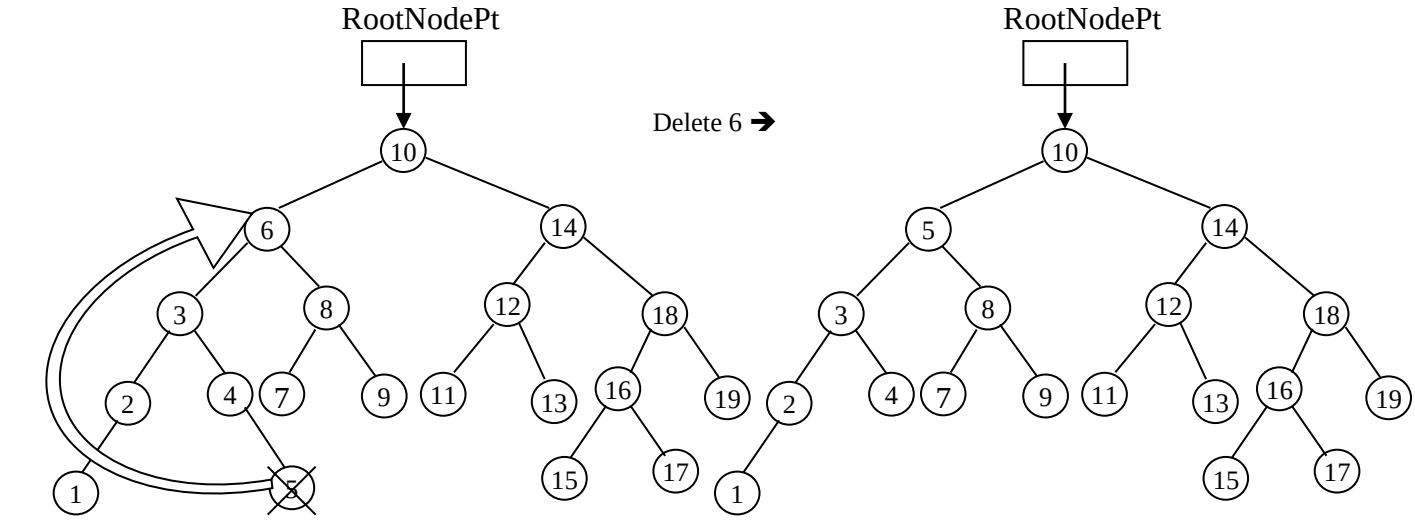
- If the deleted node is the right child of its parent, one of the following is done
  - The left child of the deleted node is made the right child of the parent of the deleted node, and
  - The right child of the deleted node is made the right child of the node containing largest element in the left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

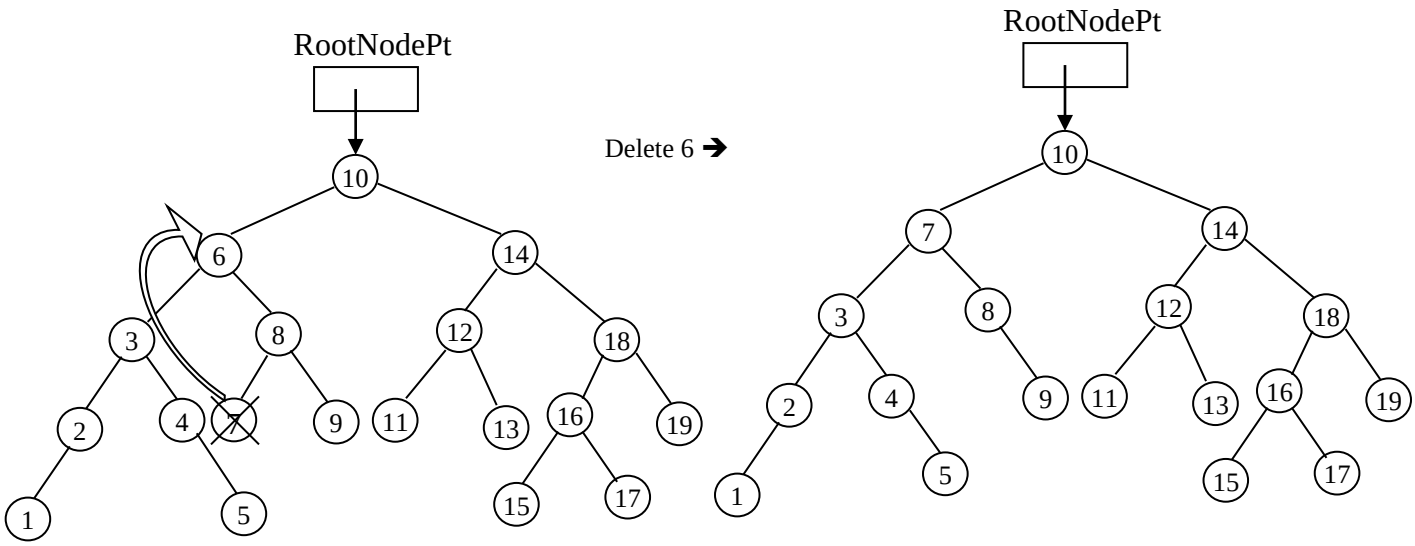
  OR
  - The right child of the deleted node is made the right child of the parent of the deleted node, and
  - The left child of the deleted node is made the left child of the node containing smallest element in the right of the deleted node



Approach 2: Deletion by copying- the following is done

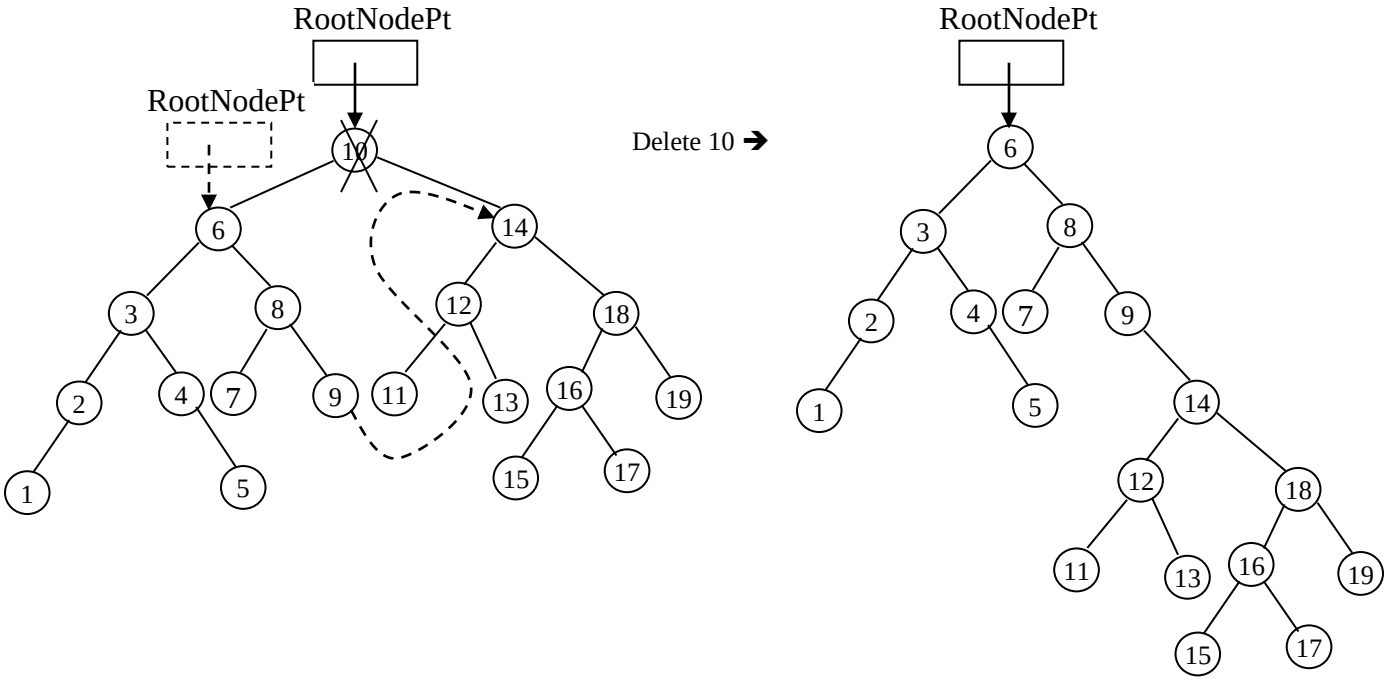
- Copy the node containing the largest element in the left (or the smallest element in the right) to the node containing the element to be deleted
- Delete the copied node

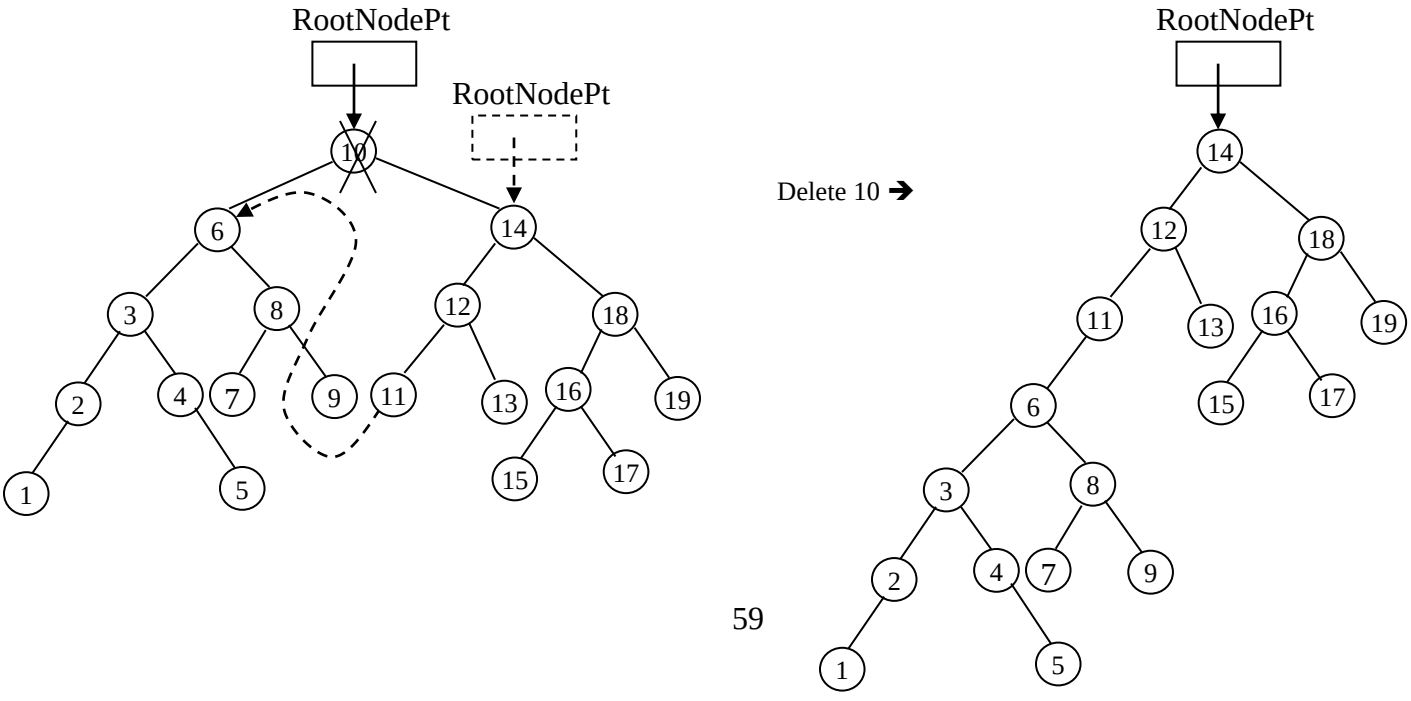




Case 4: Deleting the root node, 10
Approach 1: Deletion by merging- one of the following is done

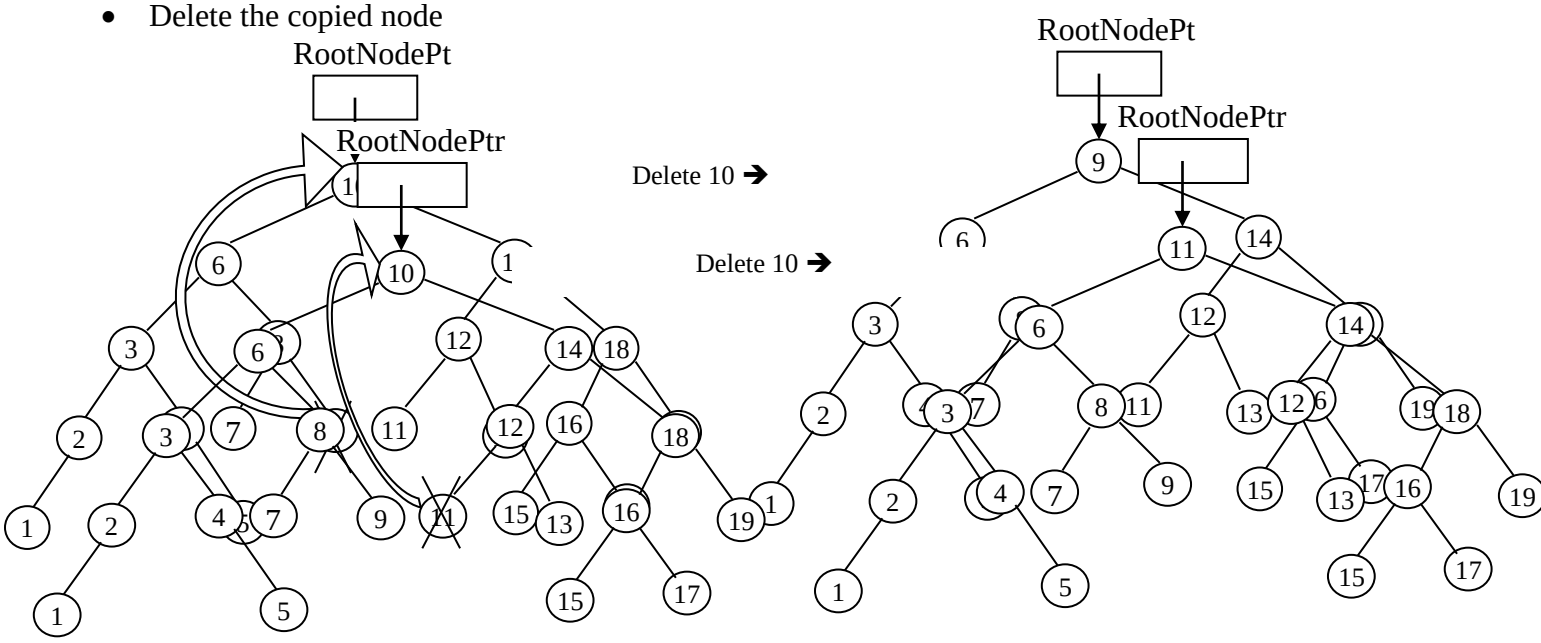
- If the tree has only one node the root node pointer is made to point to nothing (NULL)
- If the root node has left child
  - the root node pointer is made to point to the left child
  - the right child of the root node is made the right child of the node containing the largest element in the left of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





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



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))
            {
```

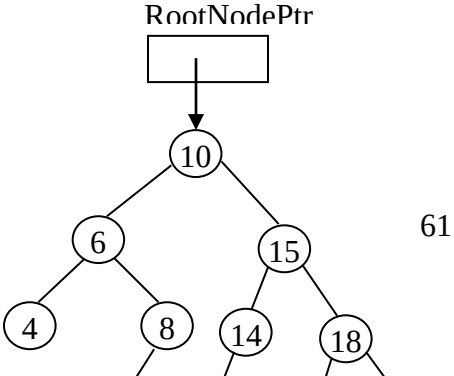
```
        if (PDNP->Left==DNP)
                PDNP->Left=NULL;
        else
                PDNP->Right=NULL;
        delete DNP;
}
else
{
        if(DNP->Left!=NULL) //find the maximum in the left
        {
                PDNP=DNP;
                DNP=DNP->Left;
                while(DNP->Right!=NULL)
                {
                        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);
        }}}}
```

6.3 Traversing

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. In-order traversal - traversing binary tree in the order of left, parent and right.
c. Post-order traversal - traversing binary tree in the order of *left*, *right* and ***parent***.

Example:



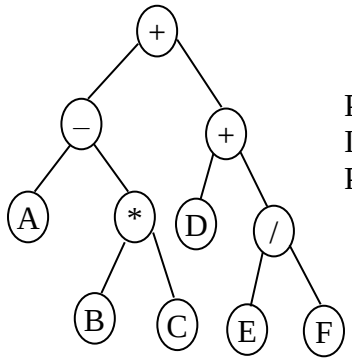
Preorder traversal - 10, 6, 4, 8, 7, 15, 14, 12, 11, 13, 18, 16, 17, 19
In-order traversal - 4, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
==> Used to display nodes in ascending order.
Post-order traversal- 4, 7, 8, 6, 11, 13, 12, 14, 17, 16, 19, 18, 15, 10

6.4 Application of binary tree traversal

- Store values on leaf nodes and operators on internal nodes

Preorder traversal - used to generate mathematical expression in prefix notation.
In-order traversal - used to generate mathematical expression in infix notation.
Post-order traversal - used to generate mathematical expression in postfix notation.

Example:



Preorder traversal - + – A * B C + D / E F → Prefix notation
In-order traversal - A – B * C + D + E / F → Infix notation
Post-order traversal - A B C * – D E F / + + → Postfix notation

Chapter 7: Non-Linear Data Structure: Graphs

7.1 Introduction

Graph is a **nonlinear** data structure which is used in many applications. Graph is a basically a collection of nodes or vertices that are connected by links called edges.

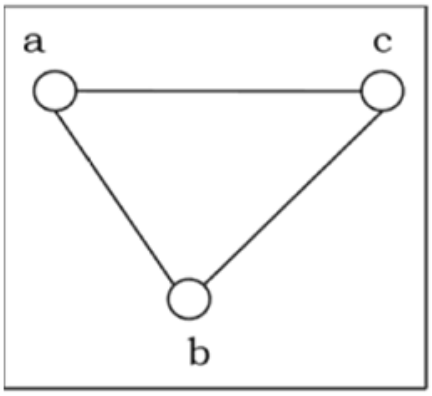
Graphs:

Definition of Graph:

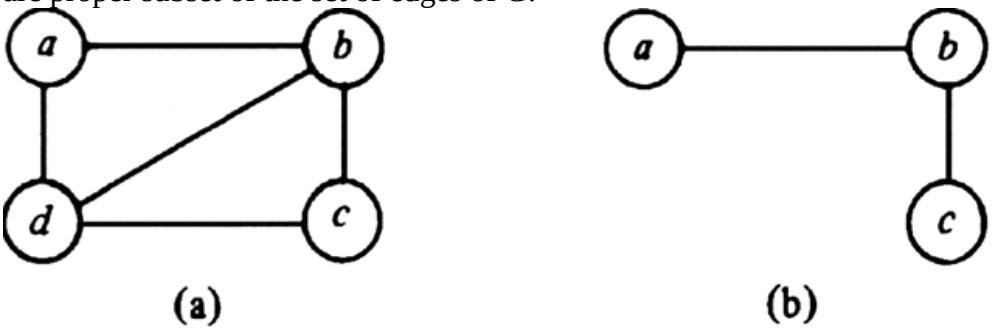
A graph is a collection of two sets V and E where Vis a finite non empty set of vertices and e is a finite non empty set of edges.
Vertices are nothing but the nodes in the graph and the two adjacent vertices are joined by edges. The graph is thus a set of two sets. Any Graph is denoted by G=(V,E)

7.2 Describing Graphs

Complete Graph: If an undirected graph of n vertices consists of n(n-1)/2 number of edges then it is called as complete graph.



Subgraph: A subgraph G' of graph G is a graph such that the set of vertices and set of edges G' are proper subset of the set of edges of G.



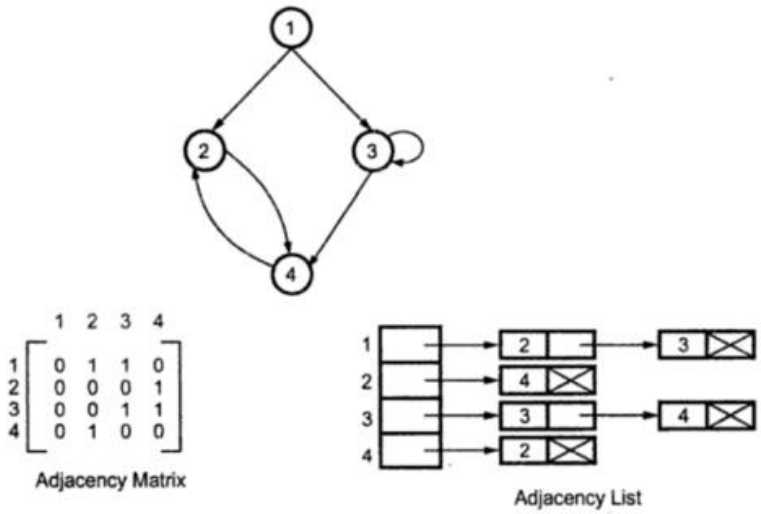
Connected Graph: An undirected graph is said to be connected if for every pair of distinct vertices Vl and Vj in V(G) there is a graph fromVi to Vj in G.

Representation of Graphs:

There are several representations of graphs, but we will discuss the two commonly used representation called Adjacency matrix and adjacency list

Adjacency matrix:

Consider a graph G of n vertices and the matrix M.if there is an edge present between vertices Vi and Vj then M[i][j]=1 else M[i][j]=0. Note that for an undierected graph if M[i][j]=1 then for M[i][j]is also 1.



Adjacency list:

Here the graph is created with linked list is called adjacency list.so all the advantage of linked list can be obtained in this type of graph. We need not have a prior knowledge of maximum number of nodes.

Application of Graph:

- Graphs are used in the design of communication and transportation networks, VLSI and other sorts of logic circuits
- Graphs are used for shape description in computer-aided design and Geographic information systems, precedence
- Graph is used in scheduling systems.

7.3 Graph Traversals:

Traversing a graph means searching the remaining nodes or vertices of graph from the given vertex. There are two commonly used techniques for traversing the graph.

1. Breadth first search
2. Depth first search

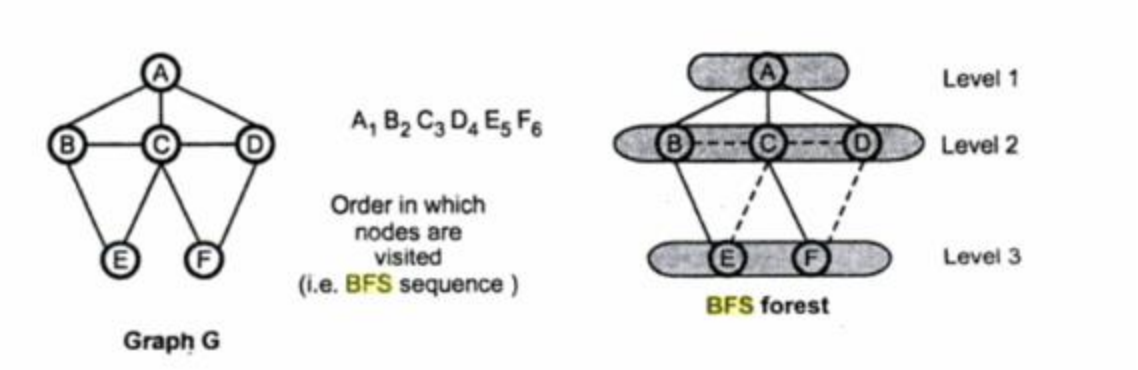
7.3.1 Breadth First search (BFS)

Some terminologies in BFS formally

Breadth first forest:

The breadth first forest is a collection of trees in which the traversal's starting vertex serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is visited then it is attached as a child to the vertex from which it is being reached.

Consider the following graph



- For finding the connected components in the graph.
- For checking if any cycle exists in the given graph
- To obtain shortest path between two vertices.

7.3.2 Depth First Search (DFS):

The depth first is a collection of trees in which traversals starting vertex serves as the root of the first tree in such a forest. whenever a new unvisited vertex is visited then it is attached as a child to the vertex from which it is being reached.

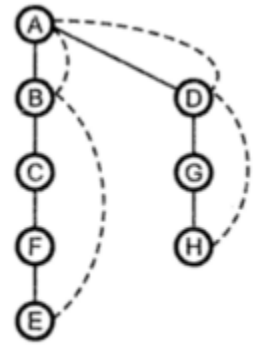
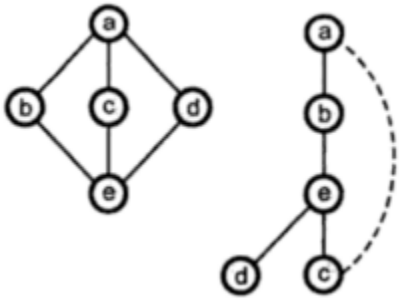


Fig. 5.9 DFS forest

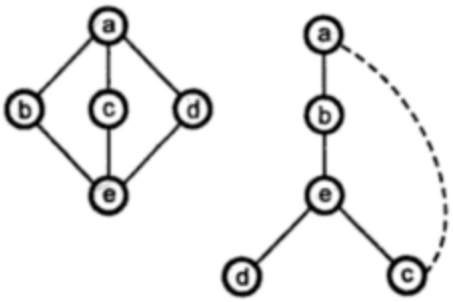
Application of Depth First search:

DFS is used for checking connectivity of a graph.

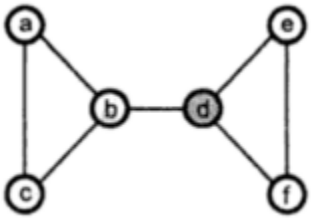
Start traversing the graph using depth first method and after an algorithm halt if all the vertices if graph are visited then the graph is said to be a connected graph



DFS is used for checking acyclicity of graph. If the DFS forest does not have back edge then the graph is said to be acyclic.



This graph contains cycle, because it contains back edge
DFS is used to find articulation point



.

A vertex of connected graph is said to be its articulation point If its removal with all its incident edges breaks the graph into disjoint pieces.
The vertex d is an articulation point

Comparison between BFS and DFS

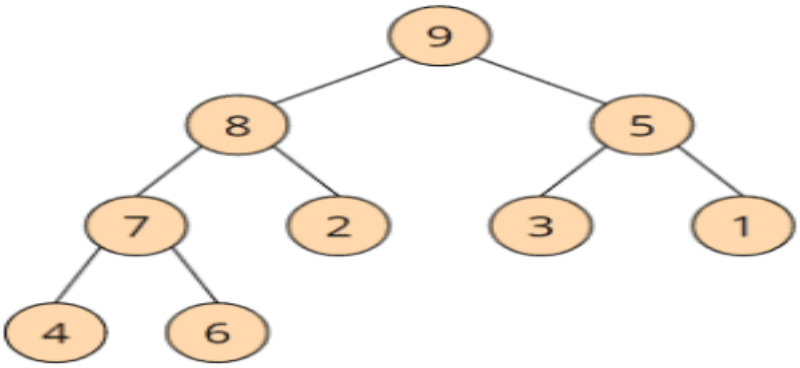
Depth First Traversal	Breadth First Traversal
This traversal is done with the help of stack data structure.	This traversal is done with the help of Queue data structure.
It works using two ordering. The first order is the order in which the vertices are reached for the first time (i.e the visited vertices are pushed onto the stacks) and the second order in which the vertices become dead end (the vertices are popped off the stack).	It works using one ordering. The order in which the vertices are reached in the same order they get removed from the queue.
The DFS sequence is composed of tree edges and back edges.	The BFS sequence is composed of tree edges and cross edges.
The efficiency of the adjacency matrix graph is $\Theta(V^2)$	The efficiency of the adjacency matrix graph is $\Theta(V^2)$
The efficiency of the adjacency list graph is $\Theta(\|V\| +\|E\|)$	The efficiency of the adjacency list graph is $\Theta(\|V\| +\|E\|)$
Application: To check connectivity, acyclicity of a graph and to find articulation point DFS is used	Application: To check connectivity, acyclicity of a graph and to find shortest path two vertices BFS is used

Chapter 6 summary questions

Chapter 8: Advanced sorting and searching Algorithms

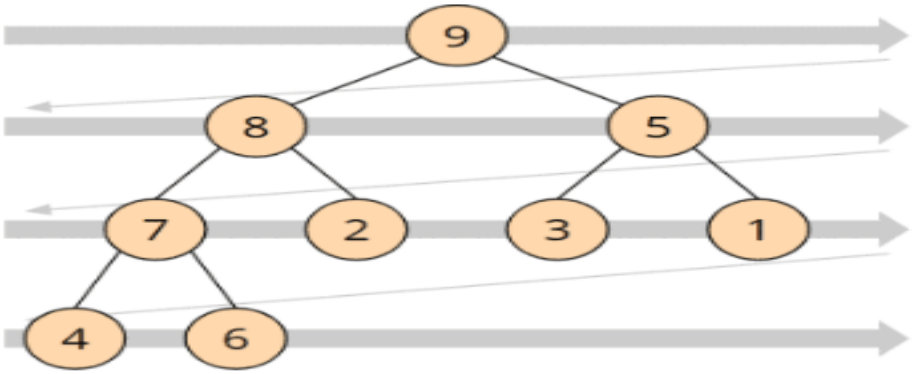
8.1 Advanced sorting algorithms

A "heap" is a **binary** tree in which each node is either greater than or equal to its children ("max heap") or less than or equal to its children ("min heap"). Here is a simple example of a "max heap":



- 9 is greater than the 8 and the 5; the 8 is greater than the 7 and the 2; etc.

A heap is projected onto an array by transferring its elements level by level from **top left to bottom right** into the array:



The heap presented above looks like this as an array:

9	8	5	7	2	3	1	4	6

In a "max heap", the largest element is always at the top – in the array form, it is, therefore, on the far left.

8.1.1 Heap Sort

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).

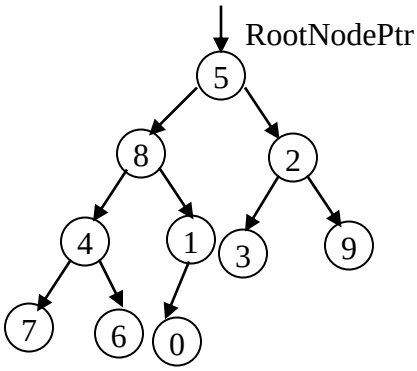
Algorithm:

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.

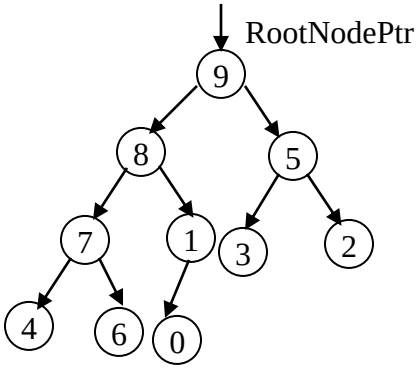
Example: Sort the following list using heap sort algorithm.

5	8	2	4	1	3	9	7	6	0

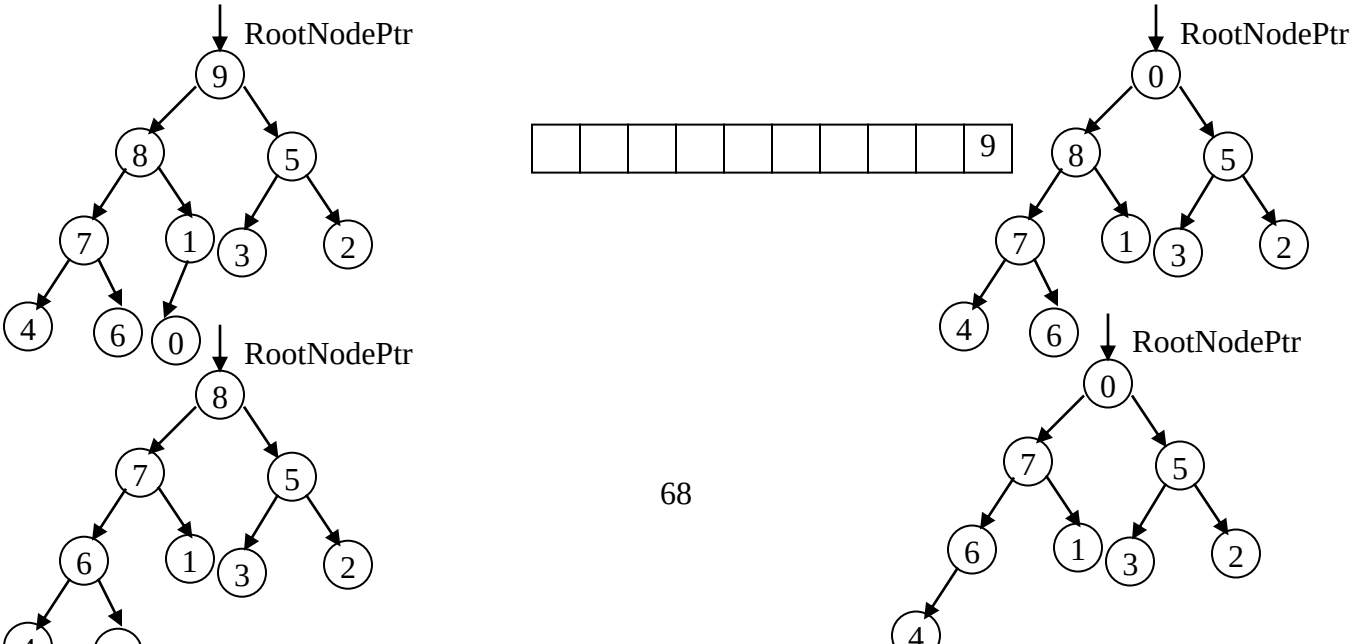
Construct the initial binary tree

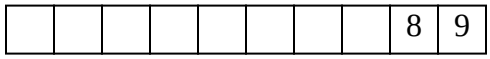


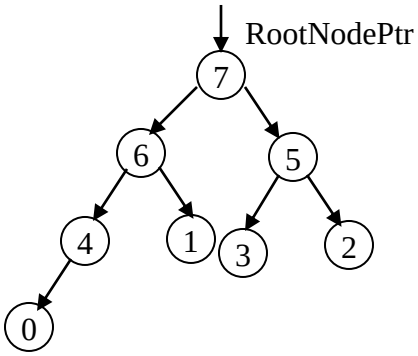
Construct the heap tree

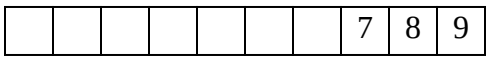


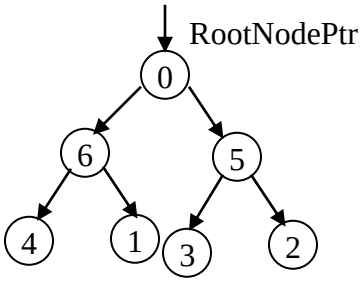
Swap the root node with the lowest, right most nodes and delete the lowest, right most value; insert the deleted value in the array in its proper position; adjust the heap tree; and repeat this process until the tree is empty.

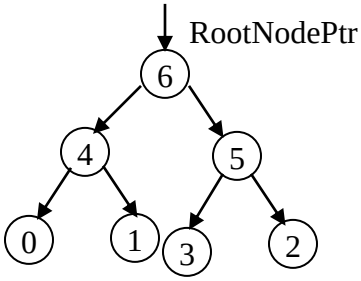


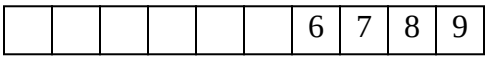

8
9

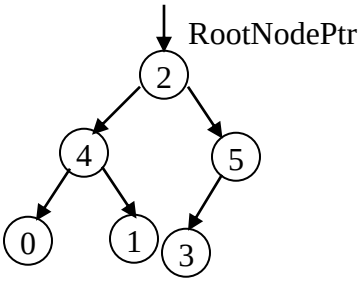

RootNodePtr
7
6
5
4
1
3
2
0

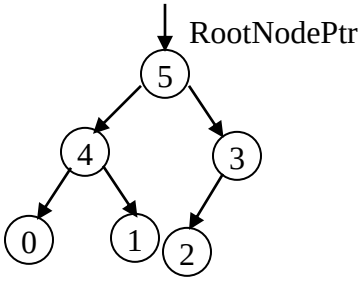

7
8
9

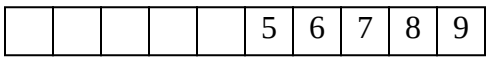

RootNodePtr
0
6
5
4
1
3
2

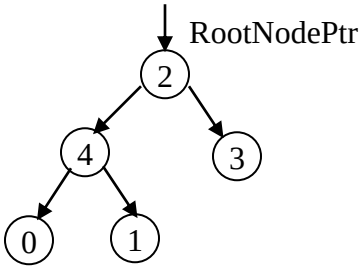

RootNodePtr
6
4
5
0
1
3
2

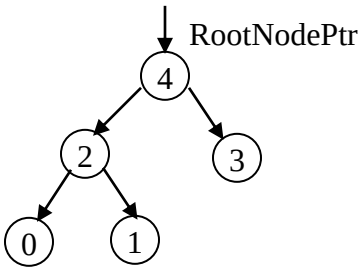

6
7
8
9

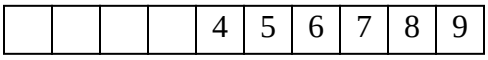

RootNodePtr
2
4
5
0
1
3

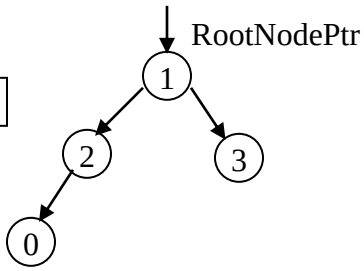

RootNodePtr
5
4
3
0
1
2

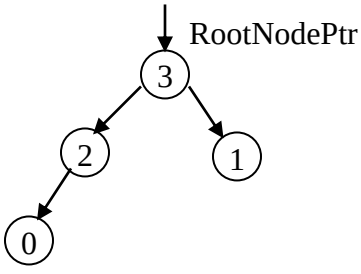

5
6
7
8
9

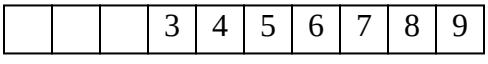

RootNodePtr
2
4
3
0
1

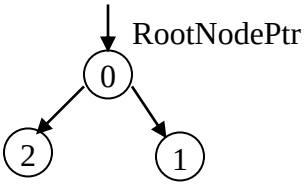

RootNodePtr
4
2
3
0
1

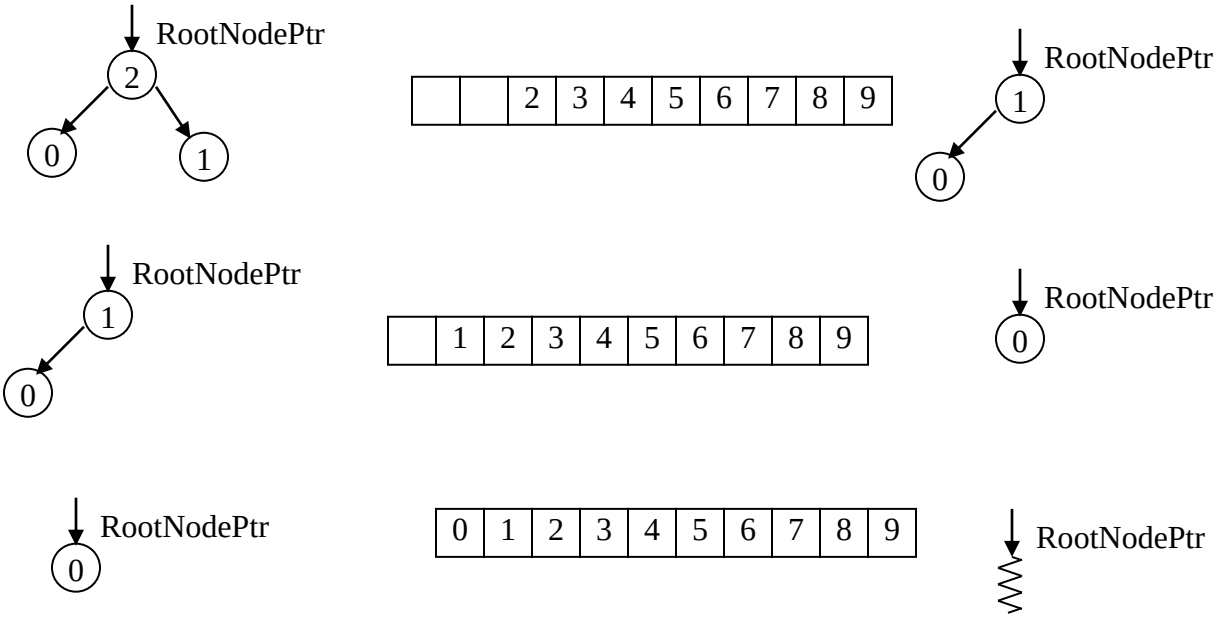

4
5
6
7
8
9


RootNodePtr
1
2
3
0


RootNodePtr
3
2
1
0


3
4
5
6
7
8
9


RootNodePtr
0
2
1



8.1.2 Quick Sort

Quick Sort

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:

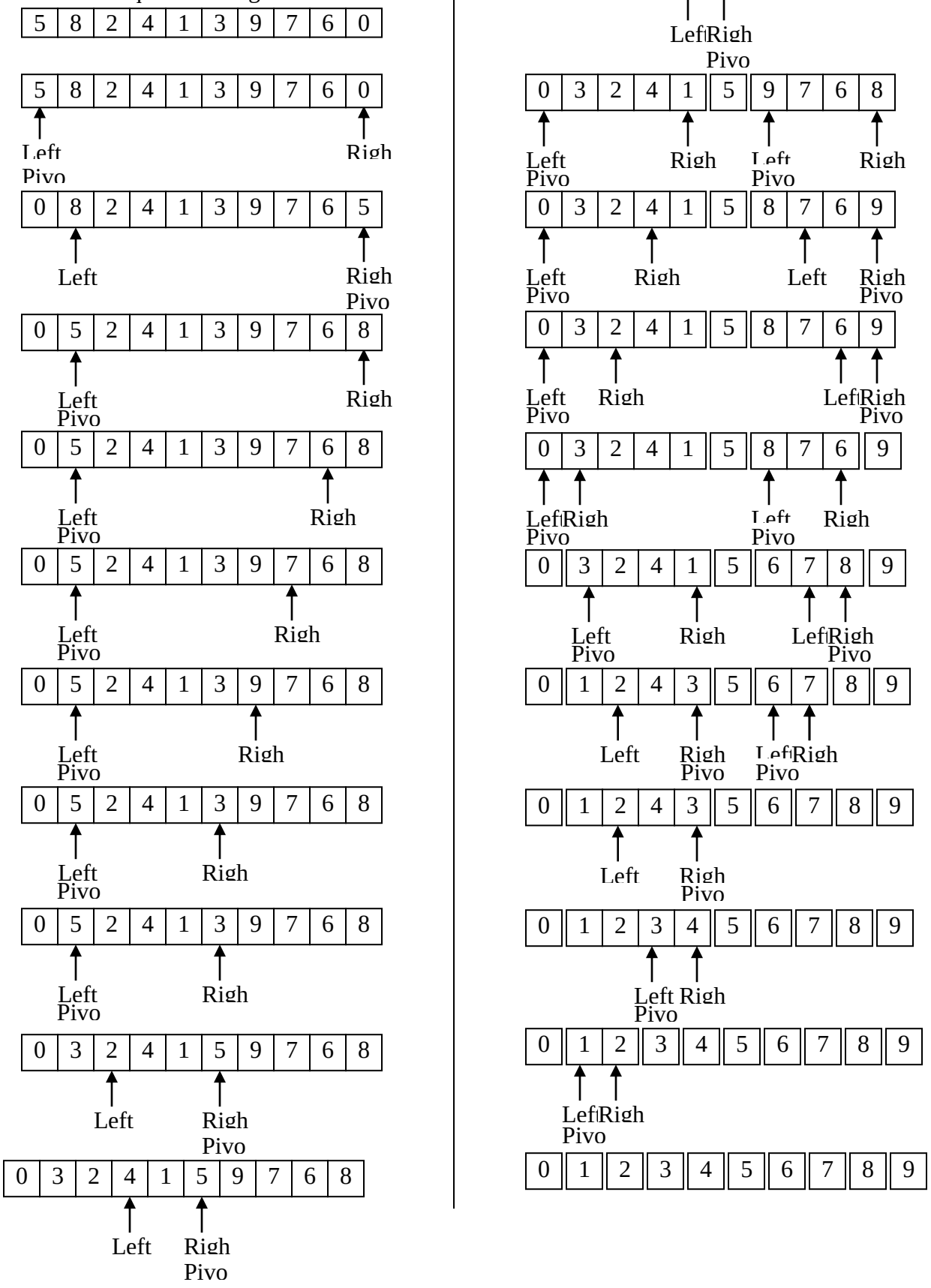
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
   - the left part has items less than or equal to the pivot value
   - the right part has items greater than or equal to the pivot value
3. Recursively sort the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

Example: Sort the following list using quick sort algorithm.



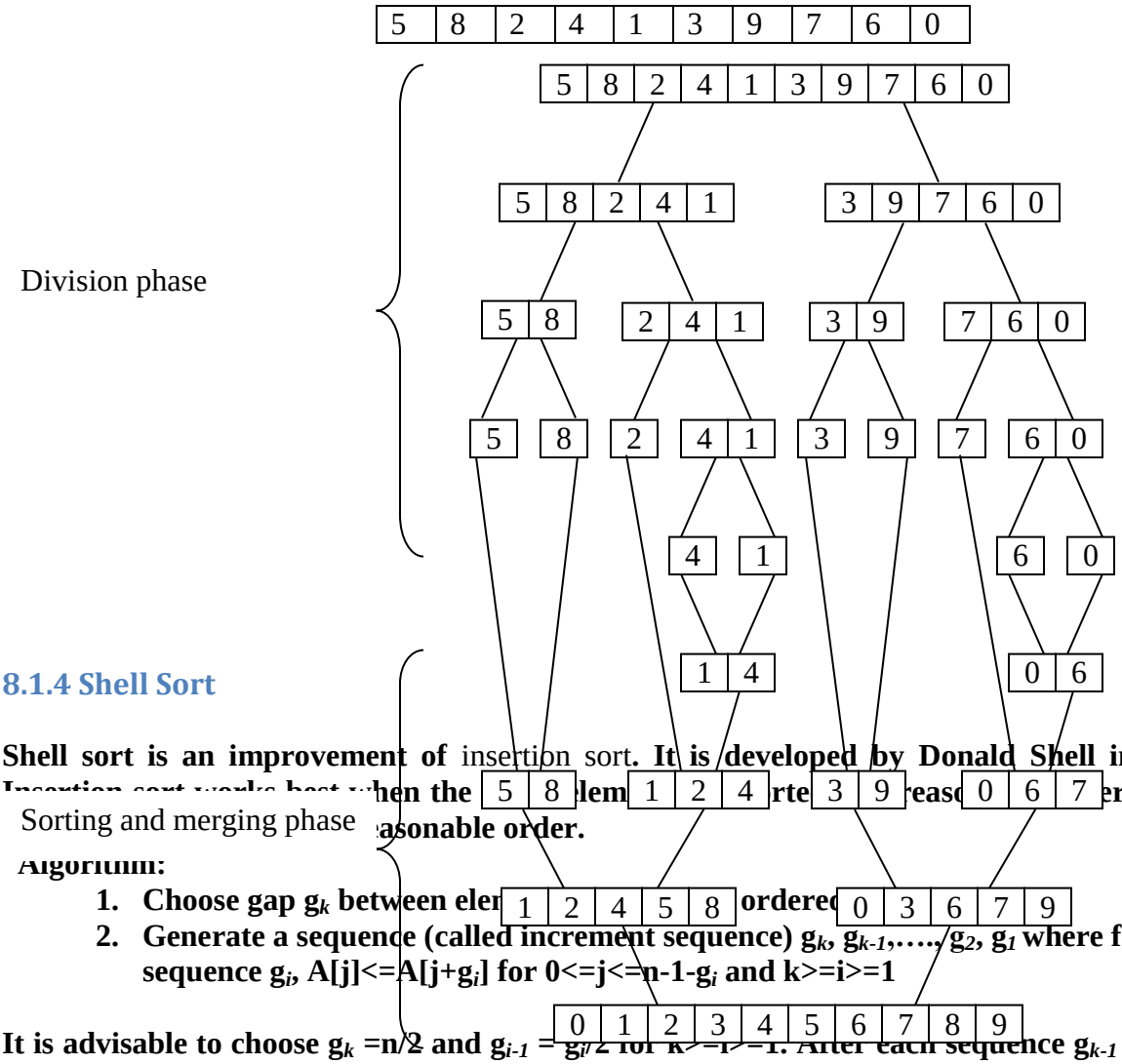
8.1.3 Merge sort

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.



8.1.4 Shell Sort

Shell sort is an improvement of insertion sort**. It is developed by Donald Shell in 1959. Insertion sort works best when the elements are partially sorted in reasonable order. Thus, Shell sort is also called diminishing increment sort.**

Algorithm:

1. **Choose gap g_k between elements of ordered sequence.**
2. **Generate a sequence (called increment sequence) $g_k, g_{k-1}, \ldots, g_2, g_1$ 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 =3

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

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

Sort (1, 0, 2, 3, 7, 5, 4, 8, 6, 9)	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

8.2 Advanced Searching Algorithms

8.2.1 Hashing:

- ✓ Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
- ✓ Hashing is also known as **Hashing Algorithm** or **Message Digest Function**.
- ✓ It is a technique to convert a range of key values into a range of indexes of an array.
- ✓ It is used to facilitate the next level searching method when compared with the linear or binary search.
- ✓ Hashing allows to update and retrieve any data entry in a constant time O(1).
- ✓ Constant time O(1) means the operation does not depend on the size of the data.
- ✓ Hashing is used with a database to enable items to be retrieved more quickly.
- ✓ It is used in the encryption and decryption of digital signatures.

What is Hash Function?

- ✓ A fixed process converts a key to a hash key is known as a **Hash Function.**
- ✓ This function takes a key and maps it to a value of a certain length which is called a **Hash value** or **Hash.**
- ✓ Hash value represents the original string of characters, but it is normally smaller than the original.
- ✓ It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.

- ✓ If the hash values are same, the message is transmitted without errors.

What is Hash Table?

- ✓ Hash table or hash map is a data structure used to store key-value pairs.
- ✓ It is a collection of items stored to make it easy to find them later.
- ✓ It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.
- ✓ It is an array of list where each list is known as bucket.
- ✓ It contains value based on the key.
- ✓ Hash table is used to implement the map interface and extends Dictionary class.
- ✓ Hash table is synchronized and contains only unique elements.

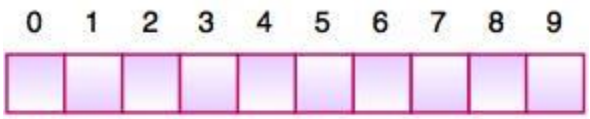


Fig. Hash Table

- ✓ The above figure shows the hash table with the size of n = 10. Each position of the hash table is called as **Slot**. In the above hash table, there are n slots in the table, names = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Slot 0, slot 1, slot 2 and so on. Hash table contains no items, so every slot is empty.
- ✓ As we know the mapping between an item and the slot where item belongs in the hash table is called the hash function. The hash function takes any item in the collection and returns an integer in the range of slot names between 0 to n-1.
- ✓ Suppose we have integer items {26, 70, 18, 31, 54, 93}. One common method of determining a hash key is the division method of hashing and the formula is :

Hash Key = Key Value % Number of Slots in the Table

- ✓ Division method or reminder method takes an item and divides it by the table size and returns the remainder as its hash value.

Data Item	Value % No. of Slots	Hash Value
26	26 % 10 = 6	6
70	70 % 10 = 0	0
18	18 % 10 = 8	8
31	31 % 10 = 1	1
54	54 % 10 = 4	4
93	93 % 10 = 3	3

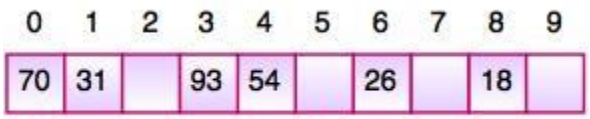


Fig. Hash Table

- ✓ After computing the hash values, we can insert each item into the hash table at the designated position as shown in the above figure. In the hash table, 6 of the 10 slots are occupied, it is referred to as the load factor and denoted by, λ = No. of items / table size. For example, λ = 6/10.
- ✓ It is easy to search for an item using hash function where it computes the slot name for the item and then checks the hash table to see if it is present.
- ✓ Constant amount of time O(1) is required to compute the hash value and index of the hash table at that location.

8.2.2 Linear Probing

- ✓ Take the above example, if we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem. This problem is called as **Collision** or **Clash**. Collision creates a problem for hashing technique.
- ✓ **Linear probing is used for resolving the collisions in hash table**, data structures for maintaining a collection of key-value pairs.
- ✓ Linear probing was invented by Gene Amdahl, Elaine M. McGraw and Arthur Samuel in 1954 and analyzed by Donald Knuth in 1963.
- ✓ It is a component of open addressing scheme for using a hash table to solve the dictionary problem.
- ✓ The simplest method is called Linear Probing. Formula to compute linear probing is:

P = (1 + P) % (MOD) Table_size

For example,

Fig. Hash Table

If we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem.

Linear probing solves this problem:

P = H(40)
44 % 10 = **0**

Position 0 is occupied by 70. so, we look elsewhere for a position to store 40.

Using Linear Probing:
P= (P + 1) % table-size
0 + 1 % 10 = **1**
But, position 1 is occupied by 31, so we look elsewhere for a position to store 40.

Using linear probing, we try next position: 1 + 1 % 10 = 2
Position 2 is empty, so 40 is inserted there.

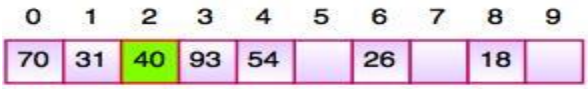


Fig. Hash Table

References

1. E. Horowitz, S.Sahni and Dinesh Mehta. Fundamentals of data structures in C++, W.H Freeman and Company
2. Sanjay Pahuja, A practical approach to data structures and algorithms, New age International publishers, 2008
3. Weiss, Mark. Data structures and algorithm analysis in C, Benjamin Cummings Publishing (2016)
4. The Design and Analysis of Computer Algorithms. Aho, Hopcroft, and Ullman

Chapter Review Questions

1. In order to use abstract data type (ADT) it is necessary to know how they are implemented

   A. True B. False

Answer B: Because the interface of ADT doesn't depend on the implementation.

2. _____ defines a set of instructions to be executed in a certain order to get the desired output.

   A. Program C. Algorithm

   B. Data structure D. None

Answer: C

3. Which one of the following data structures is non-linear?

   A. Graphs C. Arrays

   B. Linked lists D. Stacks

Answer: A

4. Which function is called in a pop() function?

   A. IsEmpty()

   B. IsFull()

5. Is a data structure in which elements can be inserted and removed at the same end is called_____?

   A. Linked list C. Stack

   B. Array D. Queue

Answer: C

6. The elements are removed from a stack in_____order?

   A. Reverse C. Hierarchical

   B. Sequential D. None

Answer: A

7. Which of the following is an abstract data structure?

   A. Queues C. Both A & B

   B. Stacks D. None

Answer: C

8. Push(A), Push(B), Pop(), Push(C),Pop(),Push(D),Push(E), after completion of those operations the total number of element present in stack is_____?

A. 1
C. 2
B. 3
D. 4

Answer B:

9. Which of the following data structure is used when you deal with an unknown number of objects or don't know how many items are in the list?

A. Linked list
C. Queues
B. Array
D. None

Answer A: Because linked list is a dynamic data structure, there is no need to give an initial size as it can grow and shrink at runtime by allocating and deallocating memory. However, the size is limited in an array as the number of elements is statically stored in the main memory.

10. If we want a random access of an element which of the following data structure is preferable?

A. Linked lists
C. Stacks
B. Arrays
D. Queues

Answer B: Some scenarios in which we use array over the linked list are:

- ✓ When we need to index or randomly access elements
- ✓ When we know the number of elements in the array beforehand, so we can allocate the correct amount of memory
- ✓ When we need speed when iterating through all the elements in the sequence
- ✓ When memory is a concern; filled arrays use less memory than linked lists, as each element in the array is the data but each linked list node requires the data as well as one or more pointers to the other elements in the linked list

11. Which of the following is an application of doubly linked list?

A. Music playlist with next and previous navigation button
B. A browser cache with forward visited page
C. Both A and B
D. None

Answer A: Doubly linked list allows traversal across the data elements in both directions.

Some applications are:

- ✓ A music playlist with next and previous navigation buttons
- ✓ The browser cache with BACK-FORWARD visited pages
- ✓ The undo and redo functionality on a browser, where you can reverse the node to get to the previous page

12. Which of the following data structure is used for checking parenthesis, prefix, infix evaluation and string reversal operations?

A. Queues
C. Arrays
B. Stacks
D. Linked lists

Answer B: Stacks are used for:

- ✓ Expression, evaluation, or conversion of evaluating prefix, postfix, and infix expressions
- ✓ Syntax parsing
- ✓ String reversal
- ✓ Parenthesis checking
- ✓ Backtracking

13. Which one of the following is **not** the application of queue data structure?

A. For the asynchronous transfer of data
B. As buffers in applications like MP3 media players and CD players
C. As waiting lists for a single shared resource in a printer
D. For the synchronous transfer of data

Answer D: Application of queue data structure

- ✓ As waiting lists for a single shared resource in a printer, CPU, call center systems, or image uploads; where the first one entered is the first to be processed
- ✓ In the asynchronous transfer of data; or example pipes, file IO, and sockets
- ✓ As buffers in applications like MP3 media players and CD players

- ✓ To maintain the playlist in media players (to add or remove the songs)

14. Which of the following sorting type is used if the file to be sorted is small enough and carried out in main memory?

A. Internal sorting
B. External sorting
C. Advanced sorting
D. None

Answer A: Explanation: refer chapter 2

15. One of the following algorithms is not follows a divide and conquer strategy?

A. Quick sort
B. Merge sort
C. Shell sort
D. None

Answer C

16. Asymptotic notations represent an algorithm running time with a given input size

A. True
B. False

Answer A: Asymptotic analysis is an essential diagnostic tool for programmers to analyze an algorithm's efficiency rather than its correctness with a given size of input. The purpose is to identify the best case, worst case, and average-case times for completing a particular activity.

17. Which of the following is true about graphs?

A. A graph may contain no edges and many vertices
B. A graph may contain many edges and no vertices
C. A graph may contain no edges and no vertices
D. A graph may contain no vertices and many edges

Answer A: A graph must contain at least one vertex.

18. Using asymptotic analysis, we can conclude the _____ scenario of an algorithm.

A. Best case
B. Average case
C. Worst case
D. A&B
E. All

Answer E: using asymptotic analysis we can analysis the best, worst and average case complexity

19. _____ is used to express the upper bound of an algorithm's running time.

A. Omega Notation
B. Theta Notation
C. Big Oh Notation
D. All of the above

Answer C: Big oh notation provides an upper bound and worst case complexity

20. Which of the following is a linear asymptotic notation?

A. O(1)
B. O(log n)
C. O(n)
D. O(n log n)

Answer C:

21. On average, a sequential search algorithm would make N/2 number of comparisons for a list of size N.

A. True
B. False

Answer B: a linear or sequential search algorithm makes n comparison on average and worst case for a size of n elements

22. Which of the following searching algorithm is best when we have small sorted list?

A. Hashing
B. Linear search
C. Binary search
D. None

Answer C: Binary search efficiently works if the list is already sorted

23. What is the worst-case time for quicksort to sort an array of n elements?

A. O(log n)
B. O(n)
C. O(n log n)
D. $O(n^2)$

Answer D: Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.

24. The code below is used to_____ in a singly linked list

```
temp = start;
start = start -> right;
start -> left = NULL;
free(temp);
```

A. Insert a node at the beginning
B. Delete a node from the beginning
C. Delete a node at intermediate position
D. Inset a node at the end

Answer B:

25. _____ a tree in which each node has at most two children called left child and right child.

A. Balanced binary tree
B. Complete binary tree
C. Binary tree
D. Binary search tree

Answer C:

26. An algorithm that calls itself directly or indirectly is known as_____

A. Recursive algorithm
C. Traversal algorithm
B. Iterative algorithm
D. None

Answer A:

27. Which one of the following is true about the in-order traversal of a binary search tree?

A. It traverses in a Decreasing order
C. It traverses in a random fashion
B. It traverses in an increasing order
D. It traverses based on priority of the node

Answer B: As a binary search tree consists of elements lesser than the node to the left and the ones greater than the node to the right, an in-order traversal (left-parent-right) will give the elements in an increasing order.

28. If several elements are competing for the same bucket in the hash table, what is it called?

A. Diffusion
C. Collision
B. Replication
D. Duplication

Answer C: In a hash table, if several elements are computing for the same bucket, then there will be a clash among elements. This condition is called Collision. The Collision is reduced by adding elements to a linked list and head address of linked list is placed in hash table.

29. Which of the following is not a technique to avoid a collision?

A. Increasing hash table size
C. Use uniform hashing
B. Use the chaining method
D. Make the hash function appear random

Answer A: If we increasing hash table size, space complexity will also increase as we need to reallocate the memory size of hash table for every collision. It is **not** the best technique to avoid a collision. We can avoid collision by making hash function **random**, **chaining** method and **uniform** hashing.

30. In simple chaining, which of the following data structure is appropriate?

A. Singly linked list
C. Circular linked list
B. Doubly linked list
D. Binary trees

Answer B: Deletion becomes easier with doubly linked list; hence it is appropriate.