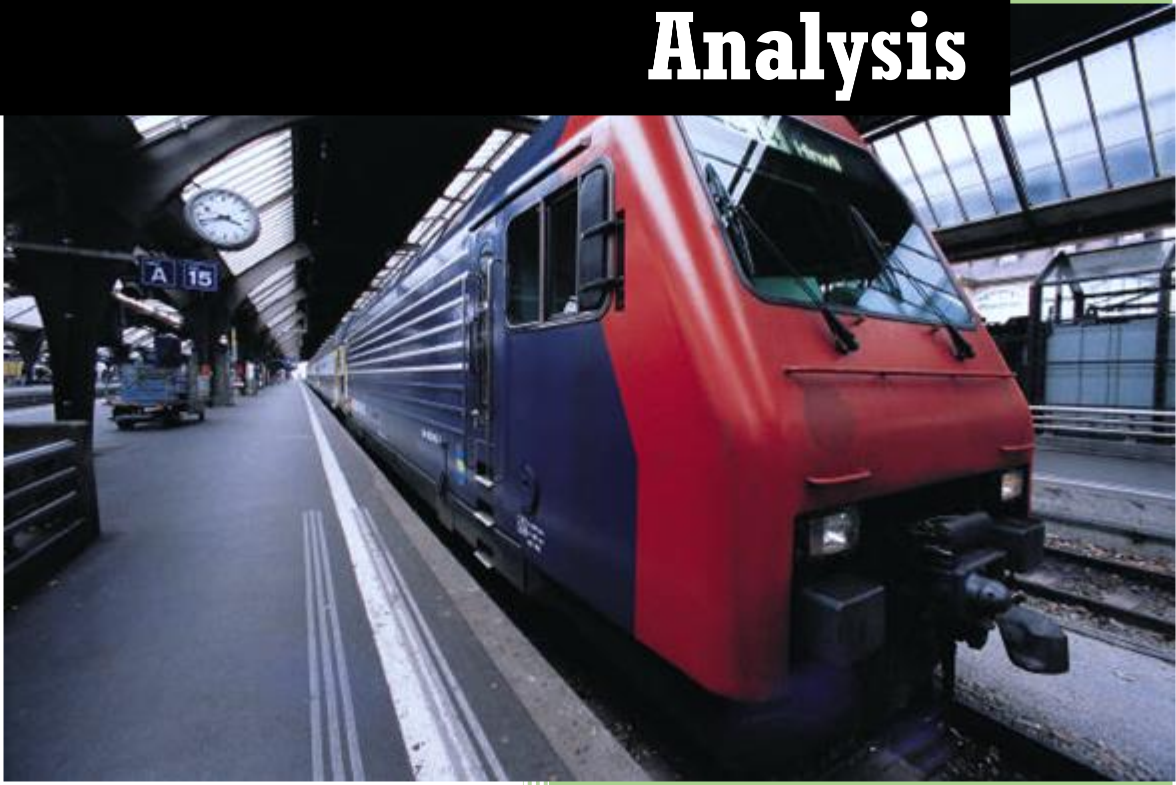
2017

Data Structure and Algorithm Analysis



Habtewold Desta (MTECH)

Mizan-Tepi University

5/23/2017

Table of Contents

Chapter One

1. Introduction to Data Structures and Algorithms Analysis

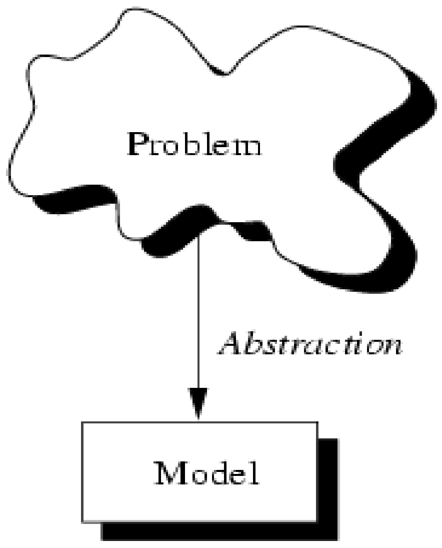
A program is written in order to solve a problem. A solution to a problem actually consists of two things:

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

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

1.1. Introduction to Data Structures

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



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

These properties include

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

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

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

1.1.1. Abstract Data Types

An ADT consists of an abstract data structure and operations. Put in other terms, an ADT is an abstraction of a data structure.

The ADT specifies:

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

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

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

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

There are lots of formalized and standard Abstract data types such as Stacks, Queues, Trees, etc.

Do all characteristics need to be modeled?
Not at all

- It depends on the scope of the model
- It depends on the reason for developing the model

1.1.2. Abstraction

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

Applying abstraction correctly is the essence of successful programming

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

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

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

Where are algorithms, then?

1.2. Algorithms

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

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

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

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

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

1.2.1. Properties of an algorithm

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

1.2.2. Algorithm Analysis Concepts

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

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.

There are two approaches to measure the efficiency of algorithms:

- Empirical: Programming competing algorithms and trying them on different instances.
- Theoretical: Determining the quantity of resources required mathematically (Execution time, memory space, etc.) needed by each algorithm.

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

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

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

1.2.3. 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.

1.2.3.1. Analysis Rules:

1. We assume an arbitrary time unit.
2. Execution of one of the following operations takes time 1:
   - Assignment Operation
   - Single Input/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=0;i<n;i++)
                   k=k+1;
   return 0;}
```

Time Units to Compute

----

1 for the assignment statement: int k=0
1 for the output statement.
1 for the input statement.
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=0;
    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)*

1.2.3.2. Formal Approach to Analysis

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

for Loops: Formally

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

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

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

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

Nested Loops: Formally

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

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

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

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

Consecutive Statements: Formally

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

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

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

Conditionals: Formally

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

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

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

Example:

Suppose we have hardware capable of executing 10^6 instructions per second. How long would it take to execute an algorithm whose complexity function was:

$T(n) = 2n^2$ on an input size of $n=10^8$?

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

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

The required number of seconds required would be given by

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

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

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

Exercises

Determine the run time equation and complexity of each of the following code segments.

```
1. for (i=0;i<n;i++)
       for (j=0;j<n; j++)
           sum=sum+i+j;

2. for(int i=1; i<=n; i++)
             for (int j=1; j<=i; j++)
            sum++;
```

What is the value of the sum if n=20?

```
3. int  k=0;
```

```
   for (int i=0; i<n; i++)
      for (int j=i; j<n; j++)
         k++;
```
What is the value of k when n is equal to 20?

```
4. int k=0;
   for (int i=1; i<n; i*=2)
       for(int j=1; j<n; j++)
             k++;
```
What is the value of k when n is equal to 20?

```
5.  int x=0;
    for(int i=1;i<n;i=i+5)
          x++;
```
What is the value of x when n=25?

```
6.  int x=0;
    for(int k=n;k>=n/3;k=k-5)
         x++;
```
What is the value of x when n=25?

```
7.  int x=0;
    for (int i=1; i<n;i=i+5)
        for (int k=n;k>=n/3;k=k-5)
            x++;
```
What is the value of x when n=25?

```
8.  int x=0;
    for(int i=1;i<n;i=i+5)
         for(int j=0;j<i;j++)
              for(int k=n;k>=n/2;k=k-3)
                     x++;
```

What is the correct big-Oh Notation for the above code segment?

1.3. Measures of Times

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

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

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

1.4. Asymptotic Analysis

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

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

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

1.4.1. The Big-Oh Notation

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

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

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

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

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).

1.4.1.1. 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)).

Theorem 5: For any base b, $\log_b(n)$ is O(logn).

All logarithms grow at the same rate

$\log_b n$ is $O(\log_d n)$ □$b, d > 1$

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

k
$\log_b n$
n
$n\log_b n$
n^2
n to higher powers
2^n
3^n
larger constants to the nth power
n!
n^n

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

1.4.1.2. Properties of the O Notation

Higher powers grow faster

•n^r □is □□O(n^s) if $0 <= r <= s$

Fastest growing term dominates a sum

• If f(n) is O(g(n)), then f(n) + g(n) is O(g)

E.g $5n^4 + 6n^3$ is O (n^4)

Exponential functions grow faster than powers, i.e. □is □□O(b^n) □∀b > 1 and k >= 0
E.g. n^{20} is O(1.05^n)

Logarithms grow more slowly than powers

•$\log_b n$ □is□□O(nk) ∀ □b > 1 and k >= 0

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

1.4.2. Big-Omega Notation

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

Formal Definition: A function f(n) is Ω(g (n)) if there exist constants c and k ∊ $\mathscr{R}$+ 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).

1.4.3. 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, $f(n)= \Theta (g(n))$ means that f(n) and g(n) have the same rate of growth.

Example:

1. If $f(n)=2n+1$, then $f(n) = \Theta (n)$

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

1.4.4. Little-o Notation

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

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

$=O(n^3)$

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

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

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

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

1.4.5. Little-Omega (ω notation)

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

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

Example: $2n^2=\omega(n)$ but it's not $\omega (n^2)$.

1.5. **Relational Properties of the Asymptotic Notations**

Transitivity

- if $f(n)=\Theta(g(n))$ and $g(n)= \Theta(h(n))$ then $f(n)=\Theta(h(n))$,
- if $f(n)=O(g(n))$ and $g(n)= O(h(n))$ then $f(n)=O(h(n))$,
- if $f(n)=\Omega(g(n))$ and $g(n)= \Omega(h(n))$ then $f(n)=\Omega (h(n))$,
- if $f(n)=o(g(n))$ and $g(n)= o(h(n))$ then $f(n)=o(h(n))$, and
- if $f(n)=\omega (g(n))$ and $g(n)= \omega(h(n))$ then $f(n)=\omega (h(n))$.

Symmetry

- $f(n)=\Theta(g(n))$ if and only if $g(n)=\Theta(f(n))$.

Transpose symmetry

- $f(n)=O(g(n))$ if and only if $g(n)=\Omega(f(n))$,
- $f(n)=o(g(n))$ if and only if $g(n)=\omega(f(n))$.

Reflexivity

- $f(n)=\Theta(f(n))$,
- $f(n)=O(f(n))$,
- $f(n)=\Omega(f(n))$.

Chapter Two

2. Simple Sorting and Searching Algorithms

2.1. Searching

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

- Sequential Search, and
- Binary Search

2.1.1. Linear Search (Sequential Search)

Pseudocode

Loop through the array starting at the first element until the value of target matches one of the array elements.

If a match is not found, return –1.

Time is proportional to the size of input *(n)* and we call this time complexity *O(n)*.

Example Implementation:

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

2.1.2. Binary Search

This searching algorithms works only on an ordered list.

The basic idea is:

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

The computational time for this algorithm is proportional to $\log_2 n$. Therefore the time complexity is O*(*log *n)*

Example Implementation:

```
int Binary_Search(int list[],int k)
{
int left=0;
int right=n-1;
int found=0;
do{
mid=(left+right)/2;
if(key==list[mid])
    found=1;
else{
     if(key<list[mid])
          right=mid-1;
     else
          left=mid+1;
      }
}while(found==0&&left<right);
if(found==0)
  index=-1;
else
  index=mid;
return index;
}
```

2.2. Sorting Algorithms

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

- Insertion Sort
- Selection Sort
- Bubble Sort

2.2.1. Insertion Sort

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

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

Basic Idea:

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

The process involved in insertion sort is as follows:

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

Implementation

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

Analysis

How many comparisons?

$1+2+3+\ldots+(n-1)= O(n^2)$

How many swaps?

$1+2+3+\ldots+(n-1)= O(n^2)$

How much space?

In-place algorithm

2.2.2. Selection Sort

Basic Idea:

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

Implementation:

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

Analysis

How many comparisons?

$(n-1)+(n-2)+\ldots+1= O(n^2)$

How many swaps?

$n=O(n)$

How much space?

In-place algorithm

2.2.3. Bubble Sort

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

Basic Idea:

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

Implementation:

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

Analysis of Bubble Sort

How many comparisons?

$(n-1)+(n-2)+\ldots+1= O(n^2)$

How many swaps?

$(n-1)+(n-2)+\ldots+1= O(n^2)$

Space?

In-place algorithm.

General Comments

Each of these algorithms requires *n*-1 passes: each pass places one item in its correct place. The *i*th pass makes either *i* or *n* - *i* comparisons and moves. So:

$$\begin{aligned} T(n) &= 1 + 2 + 3 + \ldots + (n-1) \\ &= \sum_{i=1}^{n-1} i \\ &= \tfrac{n}{2}(n-1) \end{aligned}$$

or O(n^2). Thus these algorithms are only suitable for small problems where their simple code makes them faster than the more complex code of the O(*n* log*n*) algorithm. As a rule of thumb, expect to find an O(*n* log*n*) algorithm faster for *n*>10 - *but the exact value depends very much on individual machines!*.

Empirically it’s known that Insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort. In short, there really isn't any reason to use the selection sort - use the insertion sort instead.

If you really want to use the selection sort for some reason, try to avoid sorting lists of more than a 1000 items with it or repetitively sorting lists of more than a couple hundred items.

Chapter Three

3. Data Structures

3.1. Structures

Structures are aggregate data types built using elements of primitive data types.

Structure are defined using the struct keyword:

```
E.g.  struct Time{
            int hour;
            int minute;
            int second;
        };
```

The struct keyword creates a new user defined data type that is used to declare variables of an aggregate data type.

Structure variables are declared like variables of other types.

Syntax: struct <structure tag> <variable name>;

```
E.g.     struct Time timeObject,

         struct Time *timeptr;
```

3.1.1. Accessing Members of Structure Variables

The Dot operator (.): to access data members of structure variables.

The Arrow operator (->): to access data members of pointer variables pointing to the structure.

E.g. Print member hour of timeObject and timeptr.

```
cout<< timeObject.hour; or

cout<<timeptr->hour;
```

TIP: timeptr->hour is the same as (*timeptr).hour.

The parentheses is required since (*) has lower precedence than (.).

3.1.2. Self-Referential Structures

Structures can hold pointers to instances of themselves.

```
struct list{
        char name[10];
        int count;
        struct list *next;
};
```

However, structures cannot contain instances of themselves.

3.2. Singly Linked Lists

Linked lists are the most basic self-referential structures. Linked lists allow you to have a chain of structs with related data.

Array vs. Linked lists

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

Advantages of Linked Lists

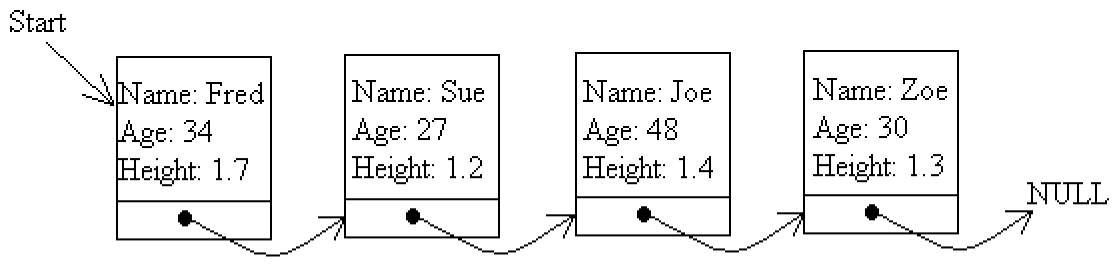
Flexible space use by dynamically allocating space for each element as needed. This implies that one need not know the size of the list in advance. Memory is efficiently utilized.

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

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

3.2.1. Creating Linked Lists in C++

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



This linked list has four nodes in it, each with a link to the next node in the series. The last node has a link to the special value NULL, which any pointer (whatever its type) can point to, to show that it is the last link in the chain. There is also another special pointer, called Start (also called head), which points to the first link in the chain so that we can keep track of it.

3.2.2. Defining the data structure for a linked list

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

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

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

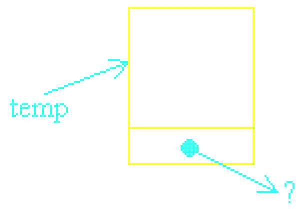
We have also declared a pointer called **start_ptr** that will permanently point to the start of the list. To start with, there are no nodes in the list, which is why **start_ptr** is set to NULL.

3.2.3. Adding a node to the list

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

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

```
temp = new node;
```



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

That's what we shall do here.

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

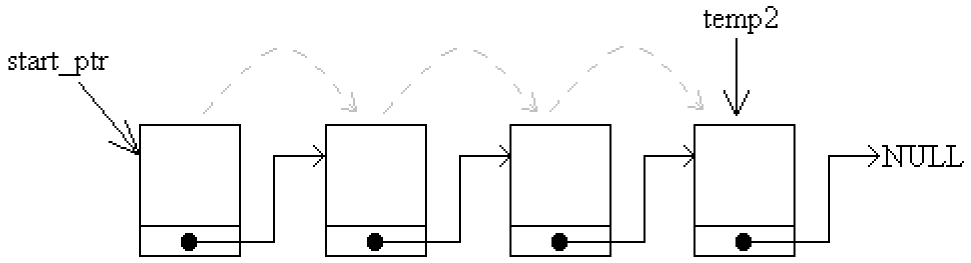
```
cout << "Please enter the name of the person: ";
cin >> temp->name;
```

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

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

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

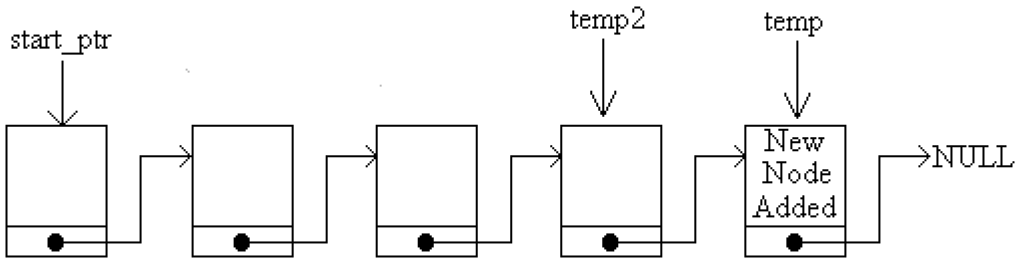
It is harder if there are already nodes in the list. In this case, the secret is to declare a second pointer, temp2, to step through the list until it finds the last node.



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

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

```
temp2->nxt = temp;
```



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

```
void add_node_at_end ()
  {  node *temp, *temp2;   // Temporary pointers

     // Reserve space for new node and fill it with data
     temp = new node;
     cout << "Please enter the name of the person: ";
     cin >> temp->name;
     cout << "Please enter the age of the person : ";
     cin >> temp->age;
     cout << "Please enter the height of the person : ";
     cin >> temp->height;
     temp->nxt = NULL;

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

3.2.4. Displaying the list of nodes

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

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

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

```
temp = start_ptr;
do
  {  if (temp == NULL)
       cout << "End of list" << endl;
     else
       {
```

```
                // Display details for what temp points to
                 cout << "Name : " << temp->name << endl;
                 cout << "Age : " << temp->age << endl;
                 cout << "Height : " << temp->height << endl;
                 cout << endl;      // Blank line

                 // Move to next node (if present)
                 temp = temp->nxt;
               }
            } while (temp != NULL);
```

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

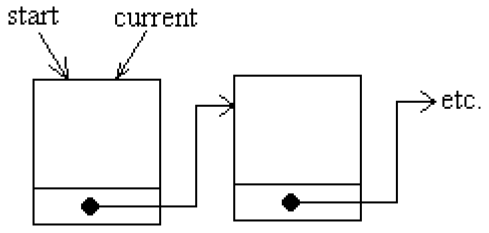
3.2.5. Navigating through the list

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

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

```
node *current;
current = start_ptr;
```

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



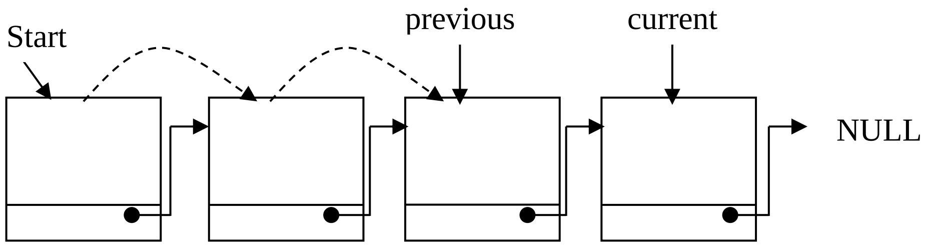
It's easy to get the current pointer to point to the next node in the list (i.e. move from left to right along the list). If you want to move current along one node, use the nxt field of the node that it is pointing to at the moment:

```
current = current->nxt;
```

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

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

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



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

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

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

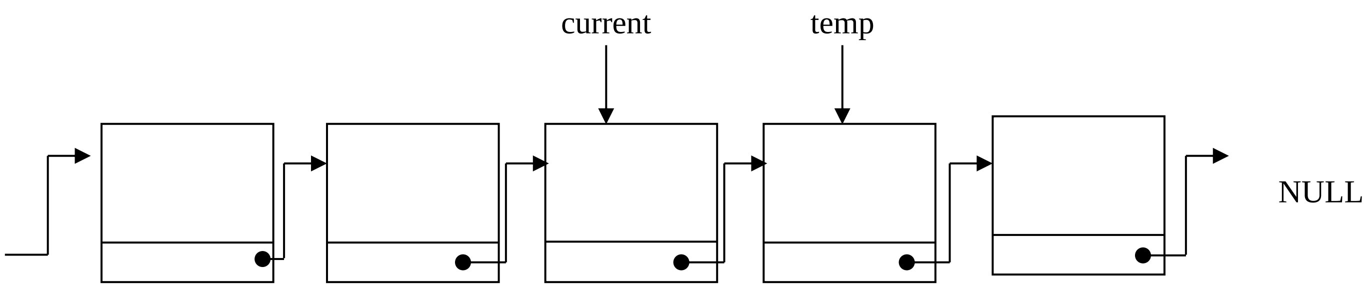
The else clause translates as follows: Declare a temporary pointer (for use in this else clause only). Set it equal to the start pointer. All the time that it is not pointing to the node before the current node, move it along the line. Once the previous node has been found, the current pointer is set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

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

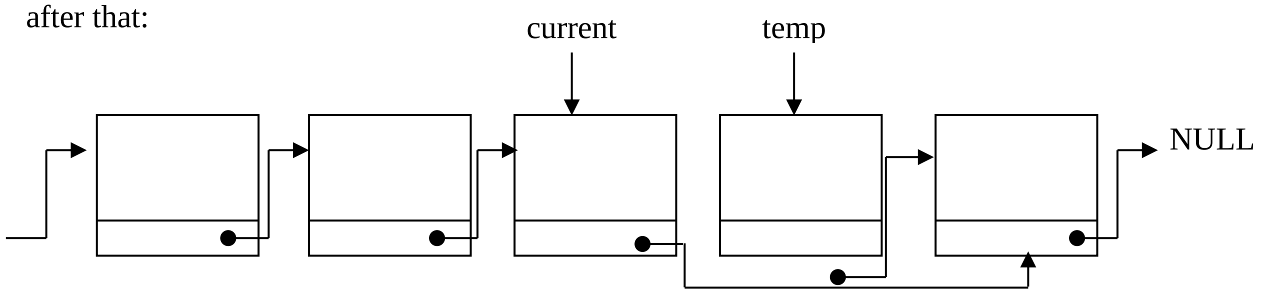
```
cout << "Please enter the new name of the person: ";
cin >> current->name;
cout << "Please enter the new age of the person : ";
cin >> current->age;
cout << "Please enter the new height of the person : ";
cin >> current->height;
```

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

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



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



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

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

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

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

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

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

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

... and so ...

3.2.6. Deleting a node from the list

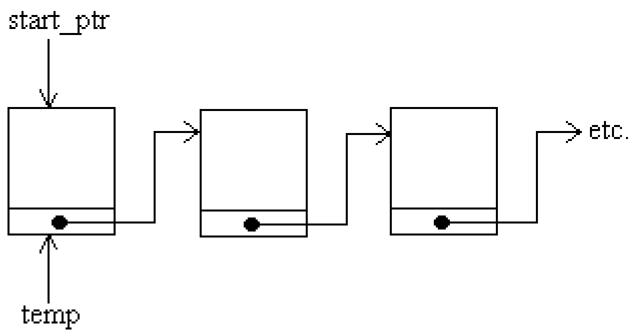
When it comes to deleting nodes, we have three choices: Delete a node from the start of the list, delete one from the end of the list, or delete one from somewhere in the middle. For simplicity, we shall just deal with deleting one from the start or from the end.

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

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

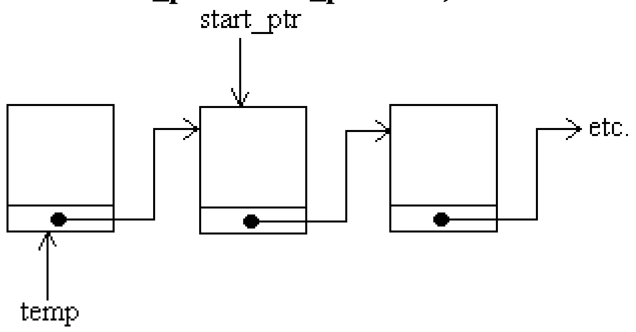
However, we can't just delete the nodes willy-nilly as it would break the chain. We need to reassign the pointers and then delete the node at the last moment. Here is how we go about deleting the first node in the linked list:

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

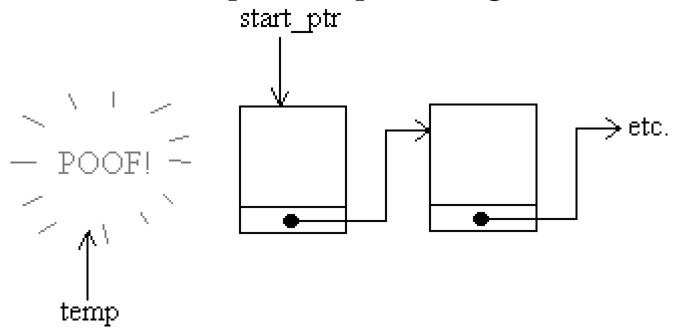


Now that the first node has been safely tagged (so that we can refer to it even when the start pointer has been reassigned), we can move the start pointer to the next node in the chain:

```
start_ptr = start_ptr->nxt;   // Second node in chain.
```



delete temp; // Wipe out original start node



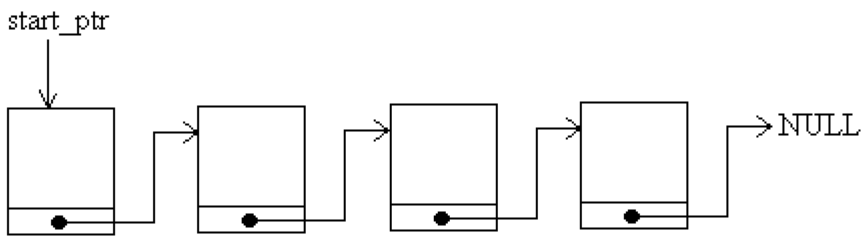
Here is the function that deletes a node from the start:

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

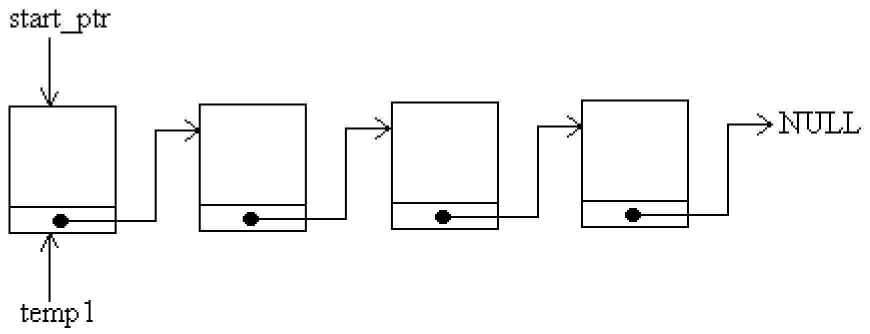
Deleting a node from the end of the list is harder, as the temporary pointer must find where the end of the list is by hopping along from the start. This is done using code that is almost identical to that used to insert a node at the end of the list. It is necessary to maintain two temporary pointers,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

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

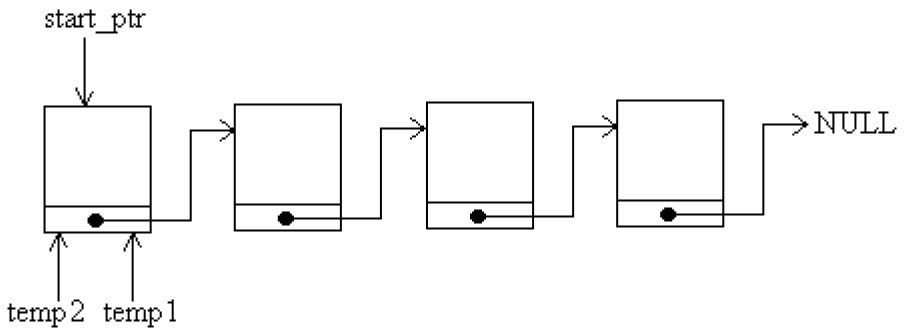
Let's try it with a rough drawing. This is always a good idea when you are trying to understand an abstract data type. Suppose we want to delete the last node from this list:



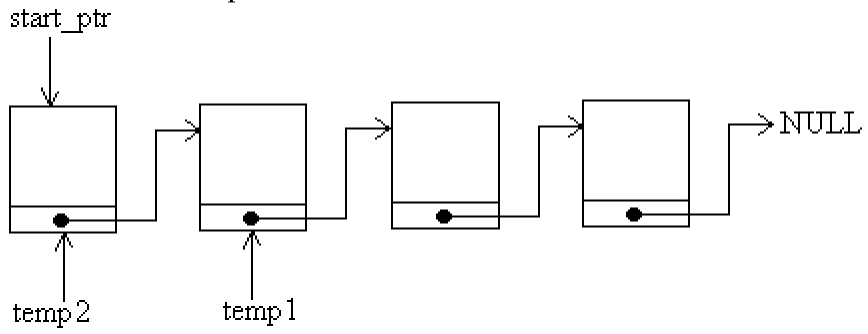
Firstly, the start pointer doesn't point to NULL, so we don't have to display a "Empty list, wise guy!" message. Let's get straight on with step2 - set the pointer **temp1** to the same as the start pointer:



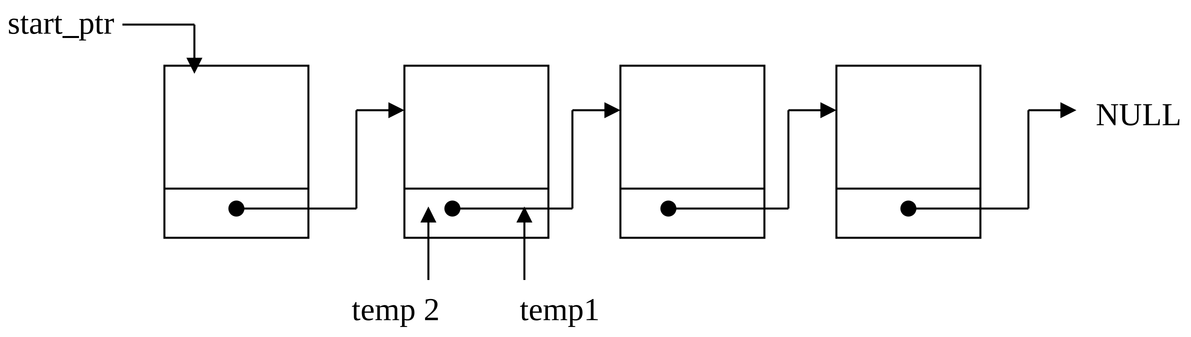
The **nxt** pointer from this node isn't NULL, so we haven't found the end node. Instead, we set the pointer **temp2** to the same node as **temp1**



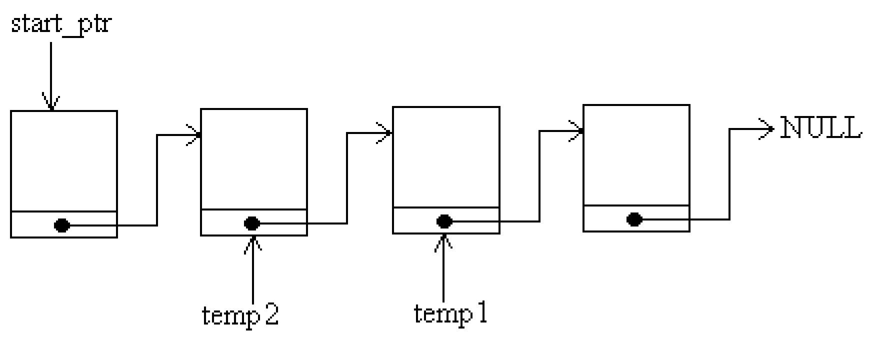
and then move temp1 to the next node in the list:



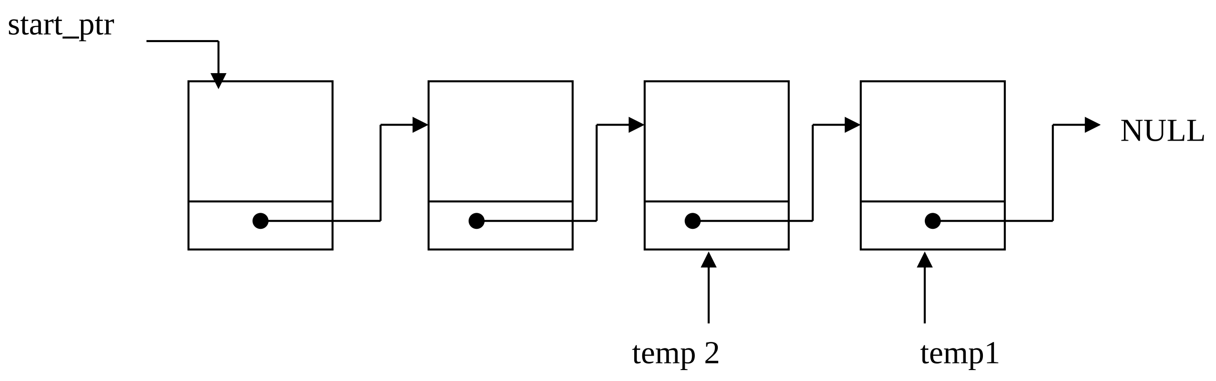
Going back to step 3, we see that temp1 still doesn't point to the last node in the list, so we make temp2 point to what temp1 points to



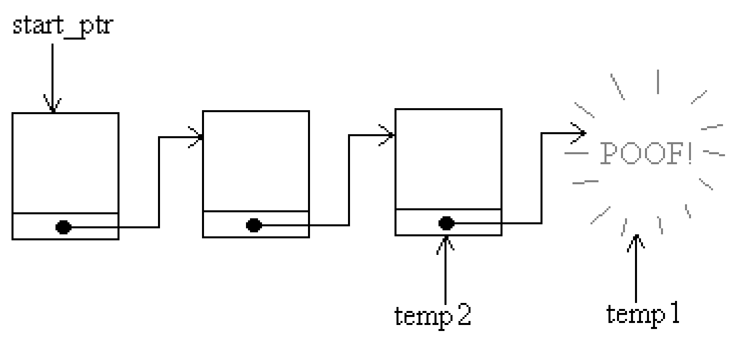
and **temp1** is made to point to the next node along:



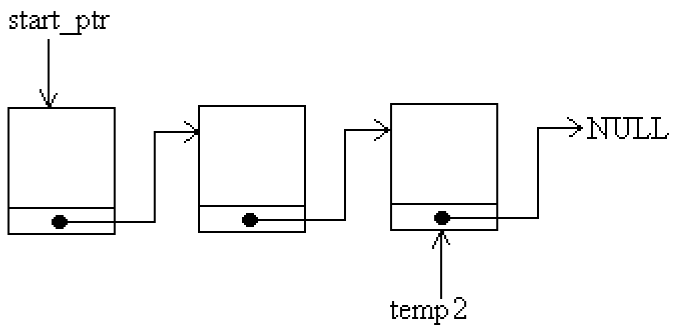
Eventually, this goes on until **temp1** really is pointing to the last node in the list, with **temp2** pointing to the penultimate node:



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



and set the **nxt** pointer of what **temp2** indicates to NULL:



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

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

The code seems a lot shorter than the explanation!

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

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

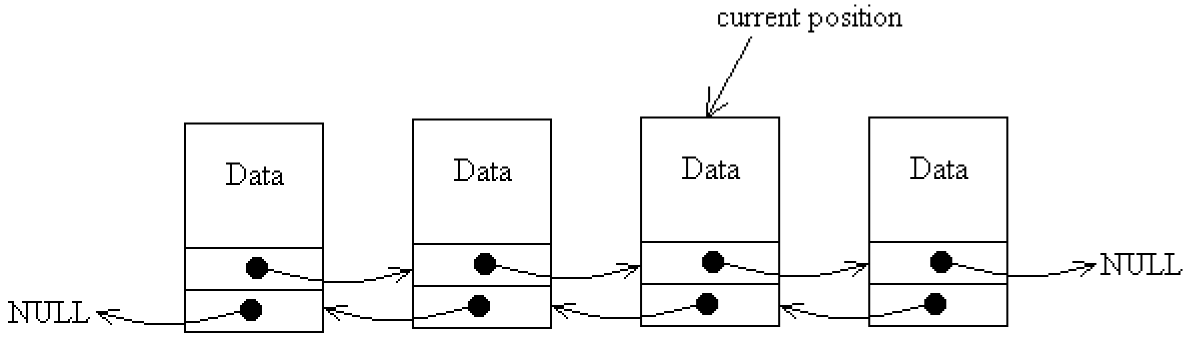
```
                    temp1 = temp1->nxt;
                  }
                delete temp1;
                temp2->nxt = NULL;
              }
          }

  }
```

3.3. Doubly Linked Lists

That sounds even harder than a linked list! Well, if you've mastered how to do singly linked lists, then it shouldn't be much of a leap to doubly linked lists

A doubly linked list is one where there are links from each node in both directions:



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

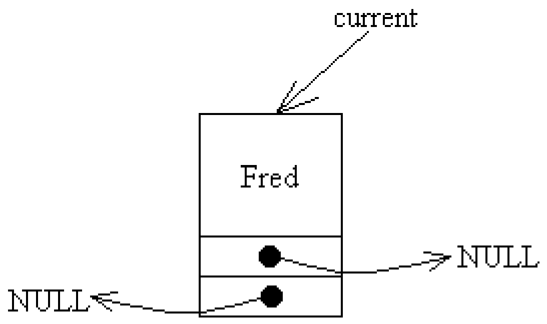
The reason we needed a start pointer in the ordinary linked list is because, having moved on from one node to another, we can't easily move back, so without the start pointer, we would lose track of all the nodes in the list that we have already passed. With the doubly linked list, we can move the current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

3.3.1. Creating Doubly Linked Lists

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

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

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



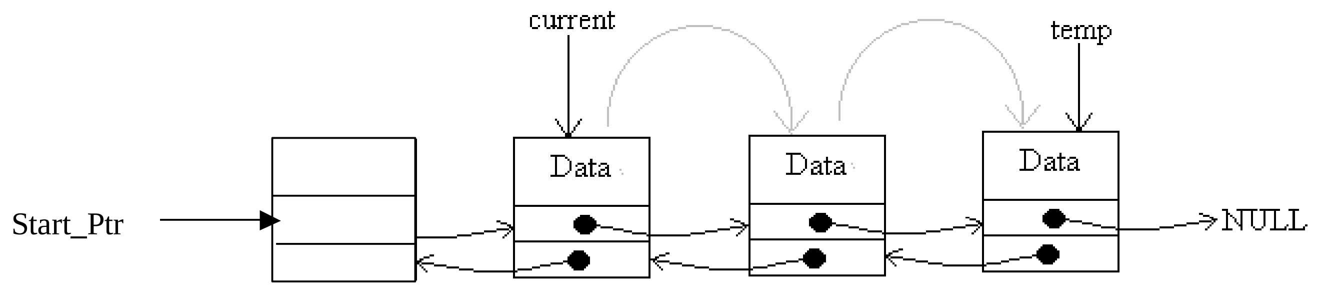
We still need to consider the directions 'forward' and 'backward', so in this case, we will need to define functions to add a node to the start of the list (left-most position) and the end of the list (right-most position).

3.3.2. Adding a Node to a Doubly Linked List

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

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

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



After that, a new node is declared, and the name is copied into it. The nxt pointer of this new node is set to NULL to indicate that this node will be the new end of the list.

The prv pointer of the new node is linked into the last node of the existing list.

The nxt pointer of the current end of the list is set to the new node.

Chapter Four

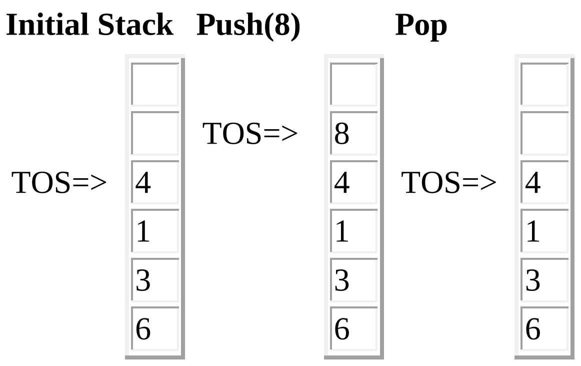
4. Stacks

A simple data structure, in which insertion and deletion occur at the same end, is termed (called) a stack. It is a LIFO (Last In First Out) structure.

The operations of insertion and deletion are called PUSH and POP

Push - push (put) item onto stack

Pop - pop (get) item from stack



Our Purpose:
To develop a stack implementation that does not tie us to a particular data type or to a particular implementation.

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.

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
        }
}
```

CreateStack()

```
{
remove existing items from the stack
initialise the stack to empty
}
```

3.4 Array

3.4.1. Array Implementation of Stacks: 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:

```
static 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.

3.4.2. Array Implementation of Stacks: 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. OK.

Algorithm

```
Step-1: If the Stack is empty then give the alert "Stack underflow" and quit; or else go to step-2
Step-2: a) Hold the value for the element pointed by the TOP
        b) Put a NULL value instead
        c) Decrement the TOP by 1
```

Implementation:

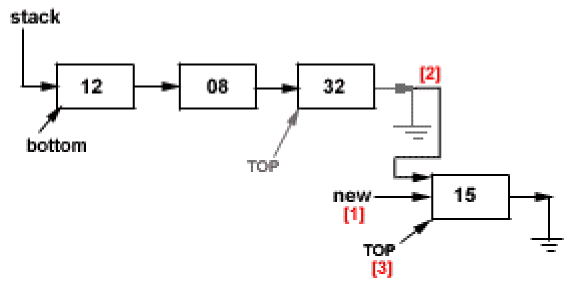
```
static int stack[UPPPERLIMIT];
int top=-1;
..
..
main()
{
..
..
poped_val = pop();
..
..
}

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

Note: - Step-2:(b) signifies that the respective element has been deleted.

3.4.3. Linked List Implementation of Stacks: the PUSH operation

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



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

Algorithm

Step-1: If the Stack is empty go to step-2 or else go to step-3
Step-2: Create the new element and make your "stack" and "top" pointers point to it and quit.
Step-3: Create the new element and make the last (top most) element of the stack to point to it
Step-4: Make that new element your TOP most element by making the "top" pointer point to it.

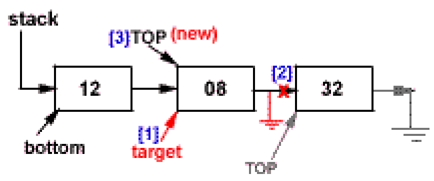
Implementation:

```
struct node{
    int item;
    struct node *next;
   }
struct node *stack = NULL; /*stack is initially empty*/
struct node *top = stack;
main()
{
..
..
push(item);
..
}
```

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

3.4.4. Linked List Implementation of Stacks: the POP Operation

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



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

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

<u>Algorithm:</u>

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

<u>Implementation:</u>-

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

main()
{
int newvalue, delval;
..
push(newvalue);
..
delval = pop();   /*POP returns the deleted value from the stack*/
}
```

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

3.4.5. Applications of Stacks

3.4.5.1. 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 it's 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.

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 Weiss p. 93 - 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 based 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.,

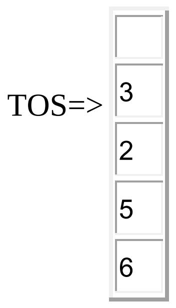
So for 6 5 2 3 + 8 * + 3 + *

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

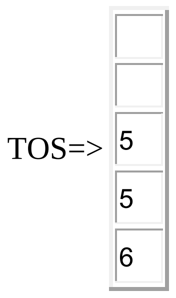
the next item is a value (2) so it is pushed onto the stack

the next item is a value (3) so it is pushed onto the stack and the stack becomes

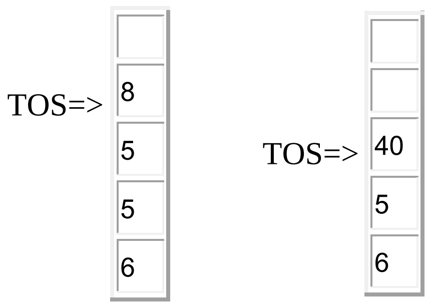


the remaining items are now: + 8 * + 3 + *

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:

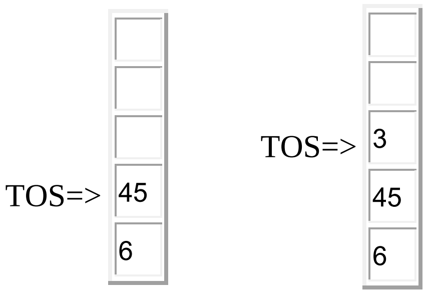


Next 8 is pushed and the next item is the operator *:



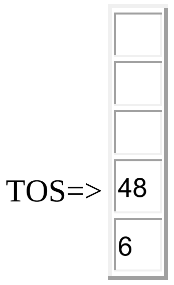
(8, 5 popped, 40 pushed)

Next the operator + followed by 3:

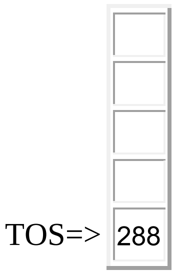


(40, 5 popped, 45 pushed, 3 pushed)

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.

3.4.5.2. Infix to Postfix (RPN) Conversion

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

Algorithm

```
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

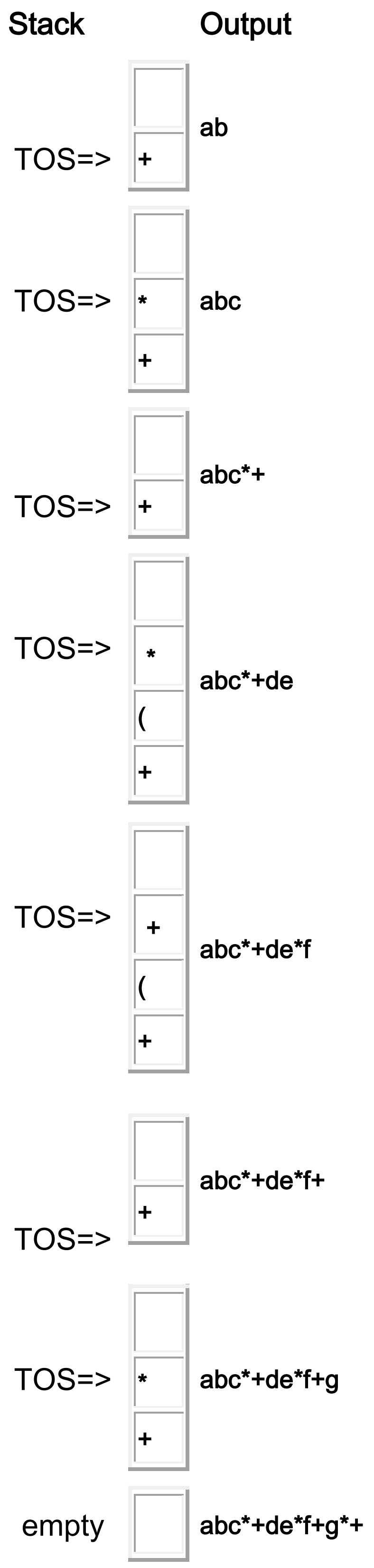
3 : All unary operators

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**



3.4.5.3. Function Calls

When a function is called, arguments (including the return address) have to be passed to the called function.
If these arguments are stored in a fixed memory area then the function cannot be called recursively since the 1st return address would be overwritten by the 2nd return address before the first was used:

```
call function abc();
continue;
   ...
 function abc;
 code;
 if (expression)
    call function abc();
 code
 return
```

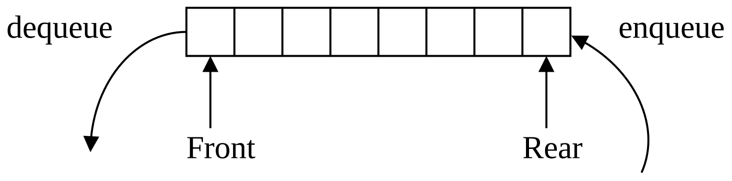
A stack allows a new instance of returns for each call to the function. Recursive calls on the function are limited only by the extent of the stack.

```
call function abc(); continue;
  ...
function abc;
code;
if (expression)
   call function abc();
code
return
```

3.5. Queue

Queue is a data structure that has access to its data at the front and rear. It operates on FIFO (Fast In First Out) basis. It uses two pointers/indices to keep stack 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

3.5.1. 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);
    }
```

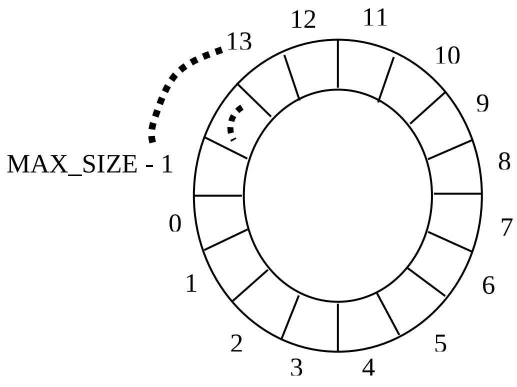
3.5.2. Circular array implementation of enqueue and dequeue operations

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	QUEUE SIZE	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:



<u>Analysis</u>:

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
```

<u>Implementation</u>:

```
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);
}
```

3.5.3. Linked list implementation of enqueue and dequeue operations

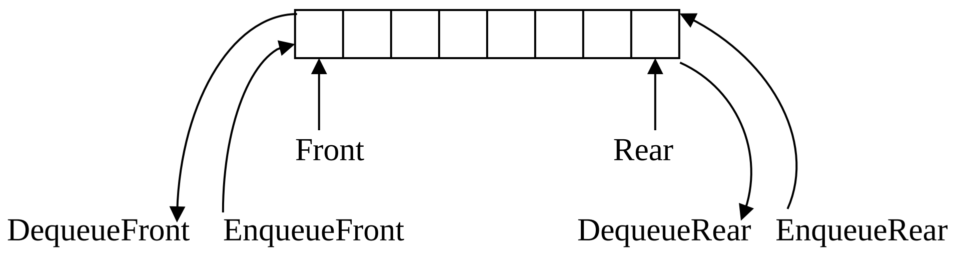
Enqueue- is inserting a node at the end of a linked list
Dequeue- is deleting the first node in the list

3.5.4. Deque (pronounced as Deck)

- is a Double Ended Queue
- insertion and deletion can occur at either end
- has the following basic operations

EnqueueFront – inserts data at the front of the list
DequeueFront – deletes data at the front of the list
EnqueueRear – inserts data at the end of the list
DequeueRear – deletes data at the end of the list

- implementation is similar to that of queue
- is best implemented using doubly linked list



3.5.5. *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.

3.5.5.1. 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);
}
```

3.5.5.2. 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

Algorithm:

```
create an empty priority queue
while(FemalesQueue is not empty)
              EnqueuePriorityQueue(DequeueFemalesQueue());
while(MalesQueue is not empty)
               EnqueuePriorityQueue(DequeueMalesQueue());
```

It is also possible to merge two or more priority queues.

Example: Consider the following priority queues and suppose large numbers represent high priorities.

ABC	CDE	DEF	FGH	HIJ
52	41	35	16	12

BCD	EFG	GHI	IJK	JKL
47	32	13	10	7

Thus, the two queues can be merged to give the following priority queue.

ABC	BCD	CDE	DEF	EFG	FGH	GHI	HIJ	IJK	JKL
52	47	41	35	32	16	13	12	10	7

3.5.6. Application of Queues

i. Print server- maintains a queue of print jobs

```
Print()
{
    EnqueuePrintQueue(Document)
}
EndOfPrint()
{
    DequeuePrintQueue()
}
```

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

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

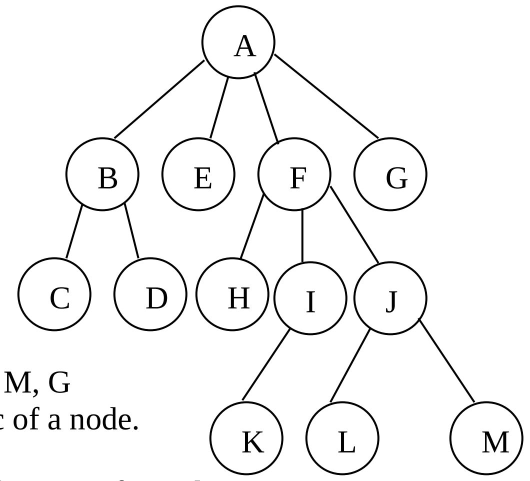
iv. Telephone calls in a busy environment –maintains a queue of telephone calls

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

3.6. Trees

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

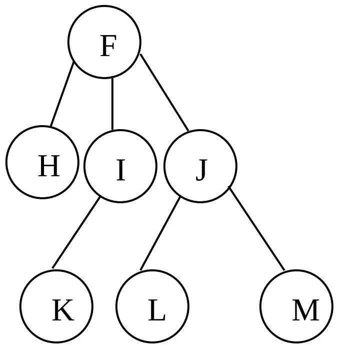
- One node distinguished as root.
- Every node C except the root is connected from exactly other node P. P is C's parent, and C is one of C's children.
- There is a unique path from the root to the each node.
- The number of edges in a path is the length of the path.



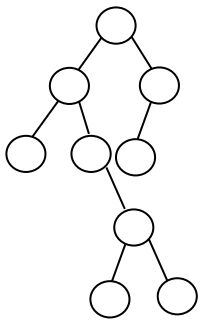
3.6.1. Tree Terminologies

Consider the following tree.

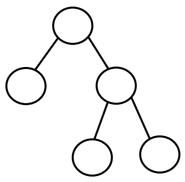
Root: a node without a parent. → A
Internal node: a node with at least one child. →A, B, F, I, J
External (leaf) node: a node without a child.→ C, D, E, H, K, L, M, G
Ancestors of a node: parent, grandparent, grand-grandparent, etc of a node.
Ancestors of K → A, F, I
Descendants of a node: children,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



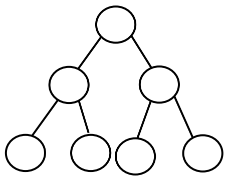
Binary tree: a tree in which each node has at most two children called left child and right child.



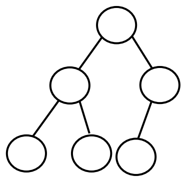
Full binary tree: a binary tree where each node has either 0 or 2 children.



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.

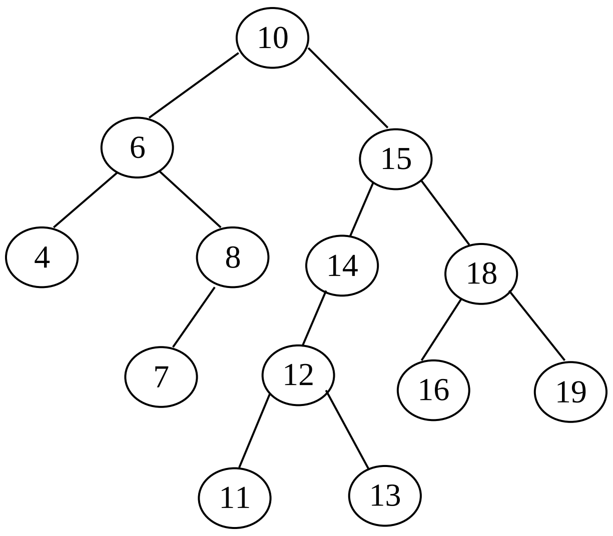


Complete binary tree: a binary tree in which the length from the root to any leaf node is either h or h-1 where h is the height of the tree. The deepest level should also be filled from left to right.



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.



3.6.2. Data Structure of a Binary Tree

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

3.6.3. Operations on Binary Search Tree

Consider the following definition of binary search tree.

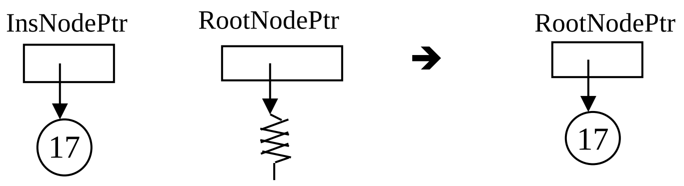
```
                struct Node
                {
                        int Num;
                        Node * Left, *Right;
                };
                Node *RootNodePtr=NULL;
```

3.6.3.1. 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.

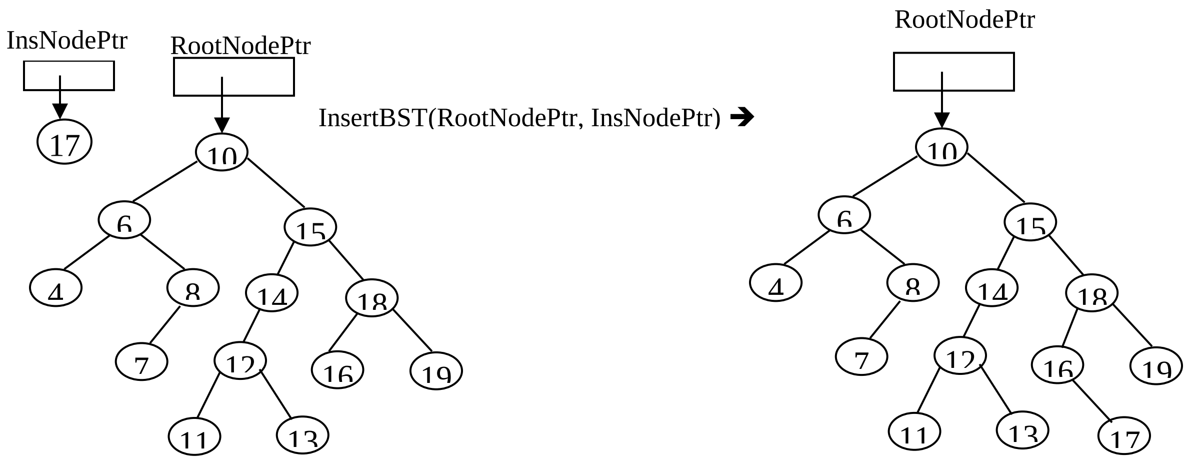
Case 1: There is no data in the tree (i.e. RootNodePtr is NULL)

- 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.

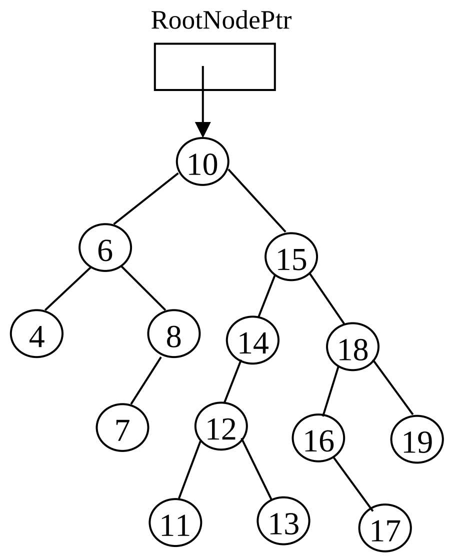
```
void InsertBST(Node *RNP, Node *INP)
{
    if(RNP->Num>INP->Num)
    {
            if(RNP->Left==NULL)
                    RNP->Left = INP;
            else
            InsertBST(RNP->Left, INP);
            }
            else
            {
            if(RNP->Right==NULL)
                    RNP->Right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

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

Example:



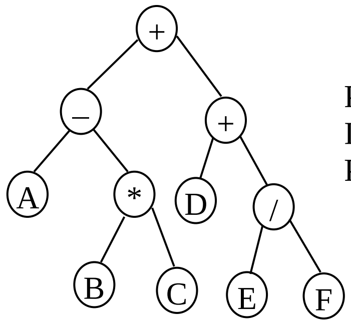
Preorder traversal -	10, 6, 4, 8, 7, 15, 14, 12, 11, 13, 18, 16, 17, 19
Inorder traversal -	4, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 ==> Used to display nodes in ascending order.
Postorder traversal-	4, 7, 8, 6, 11, 13, 12, 14, 17, 16, 19, 18, 15, 10

3.6.3.3. 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.
Inorder traversal -	used to generate mathematical expression in infix notation.
Postorder traversal -	used to generate mathematical expression in postfix notation.

Example:



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

Function calls:

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

Implementation:

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

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

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

3.6.3.4. 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;
```

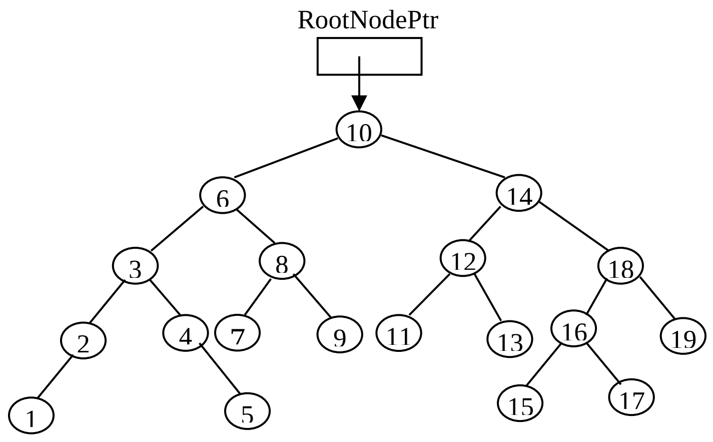
Implementation:

```
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));
}
```

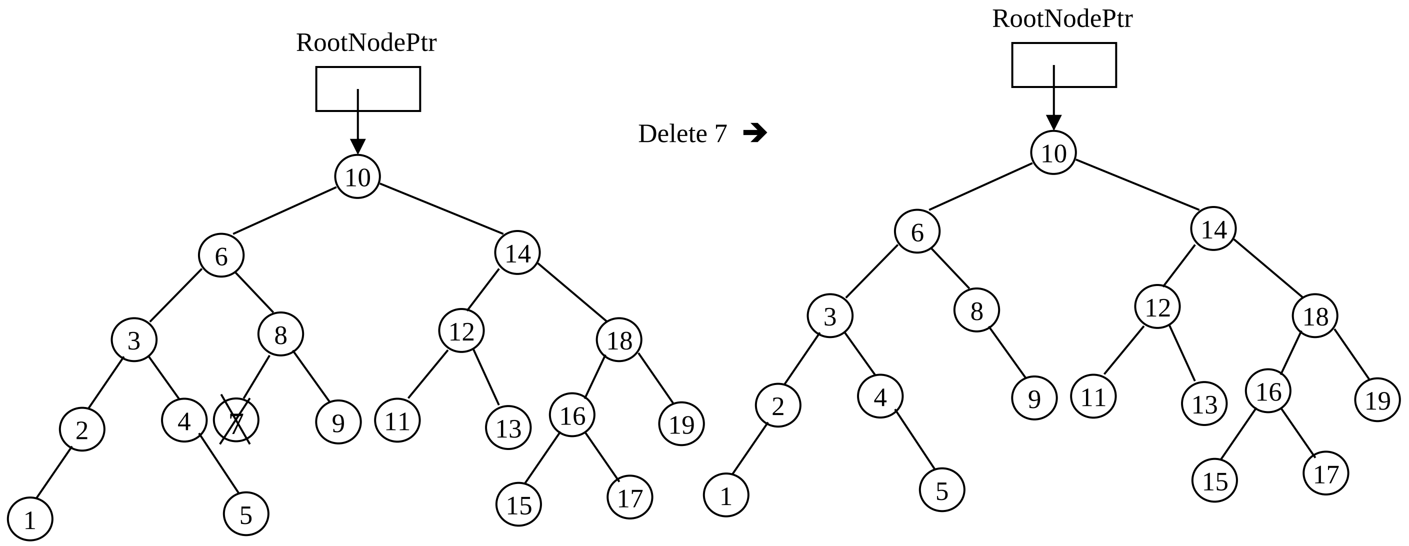
3.6.3.5. Deletion

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.



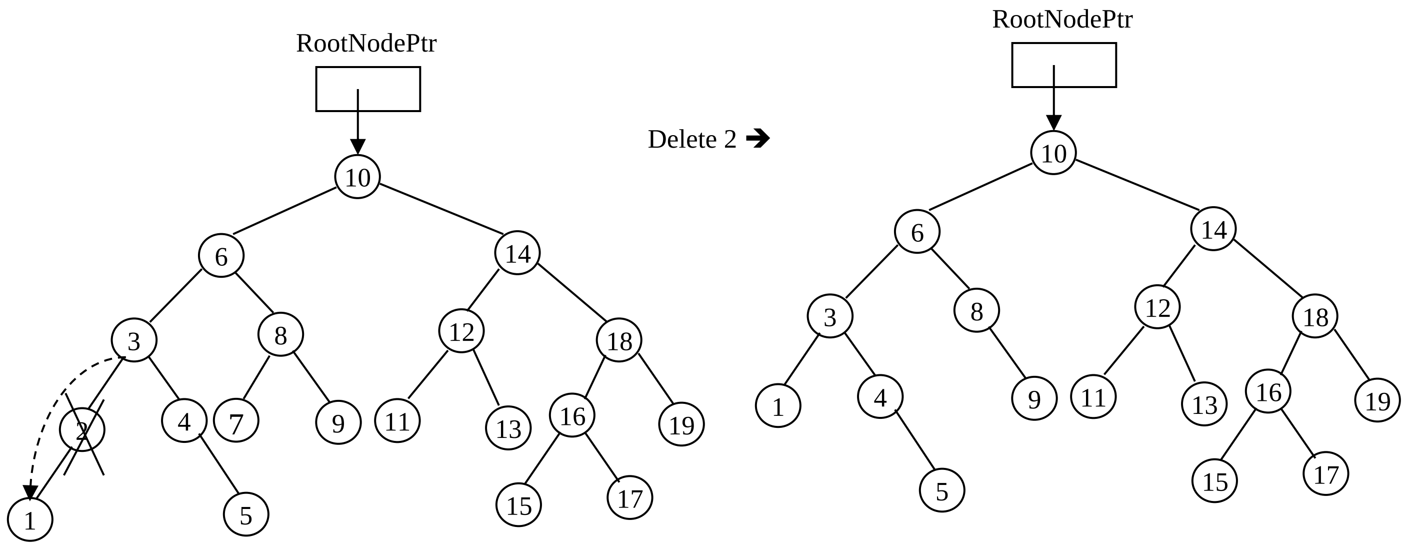
Case 1: Deleting a leaf node (a node having no child), e.g. 7



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

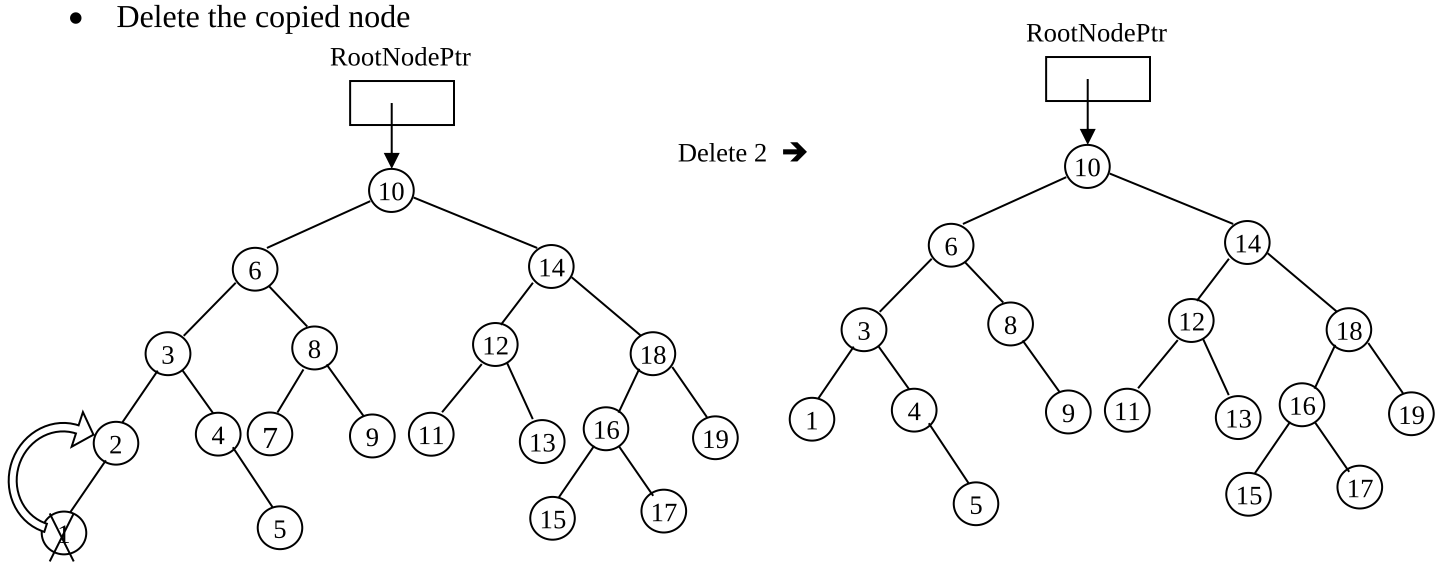
Approach 1: Deletion by merging – one of the following is done

- 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

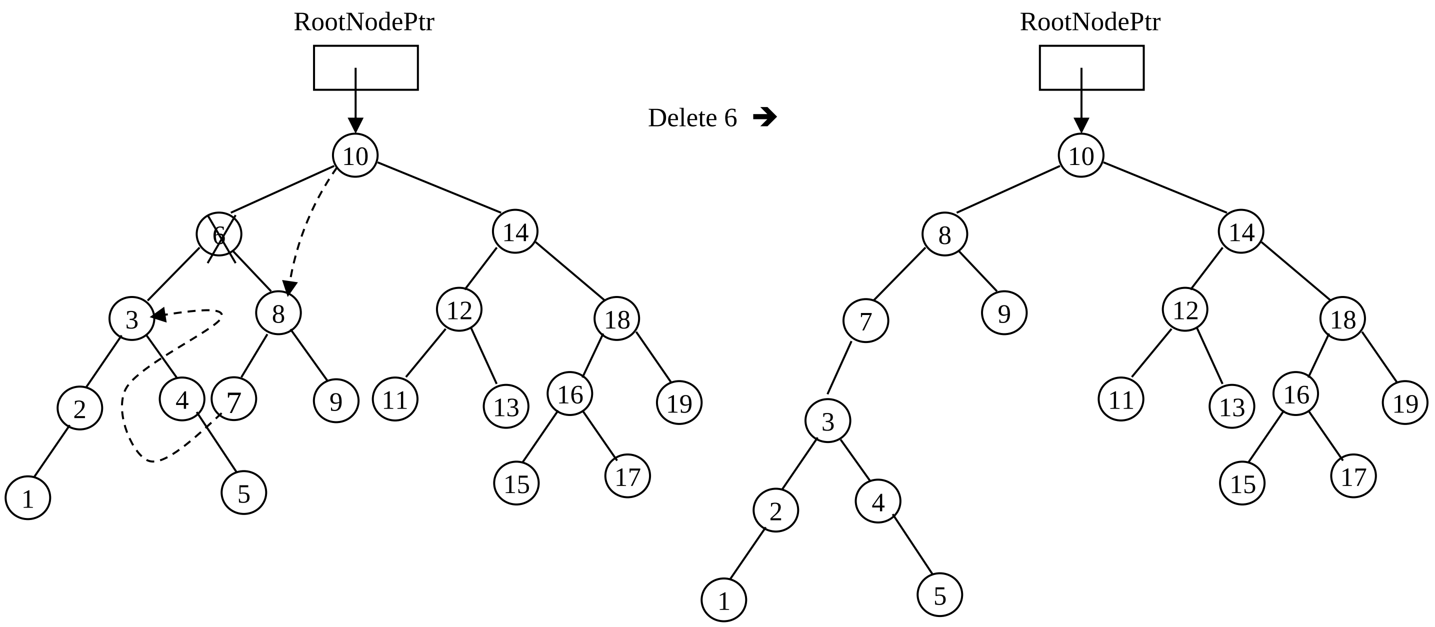
Approach 1: Deletion by merging – one of the following is done

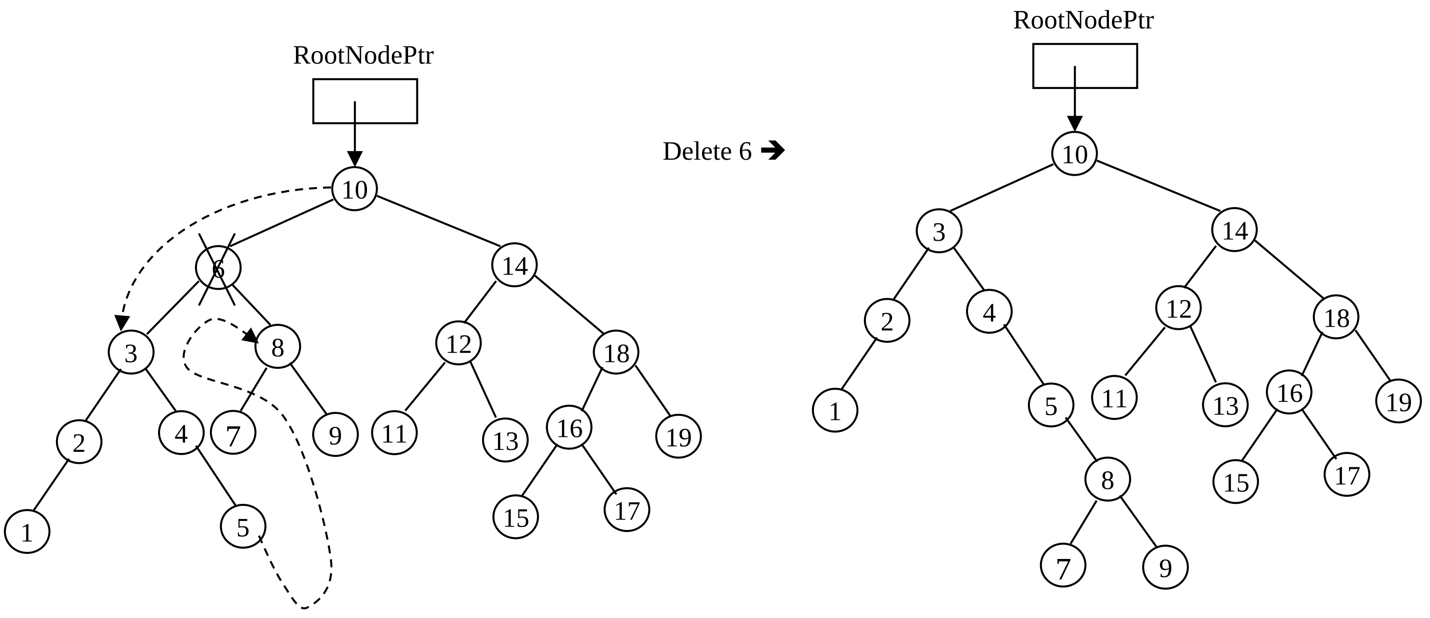
- If the deleted node is the left child of its parent, one of the following is done
  - The left child of the deleted node is made the left child of the parent of the deleted node, and
  - The right child of the deleted node is made the right child of the node containing largest element in the left of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

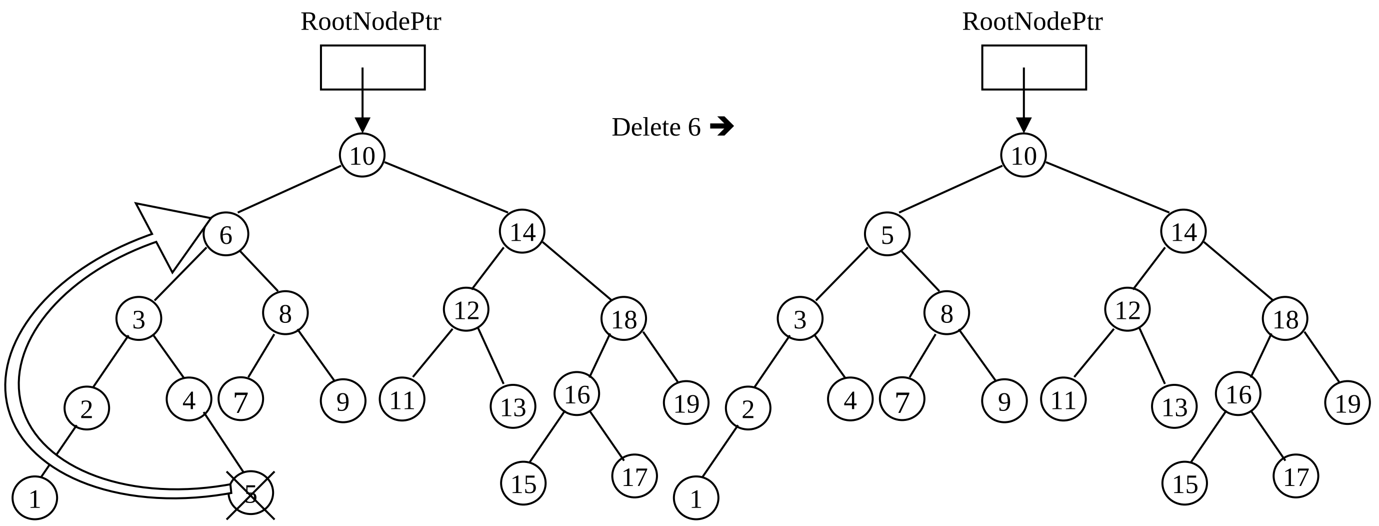
  OR
  - The right child of the deleted node is made the right child of the parent of the deleted node, and
  - The left child of the deleted node is made the left child of the node containing smallest element in the right of the deleted node

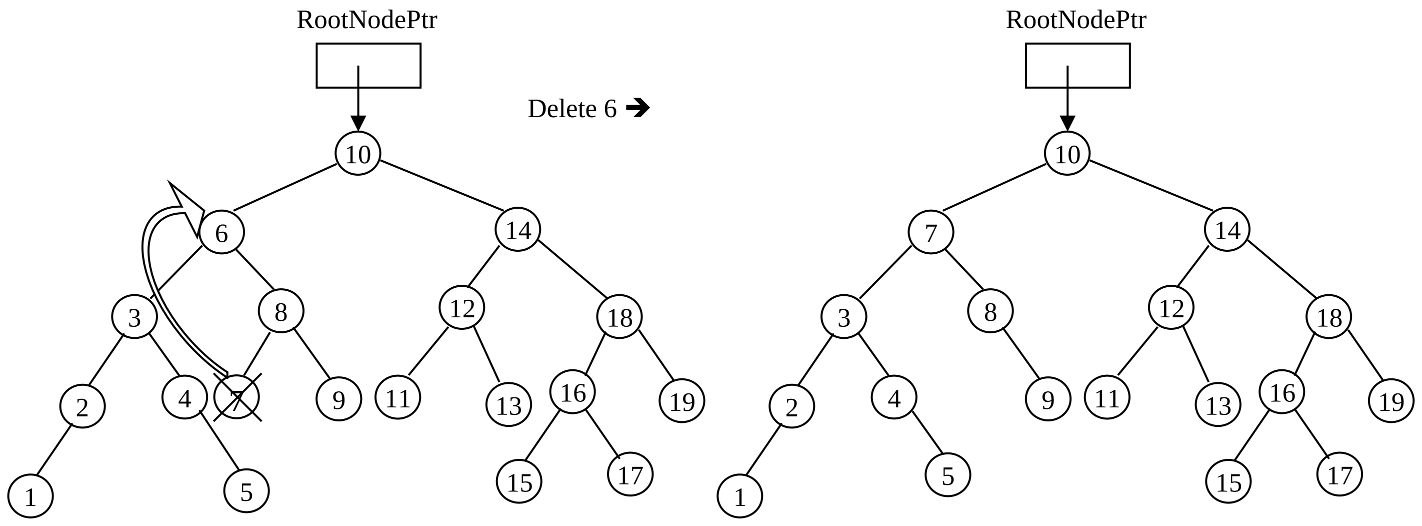




Approach 2: Deletion by copying- the following is done

- 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

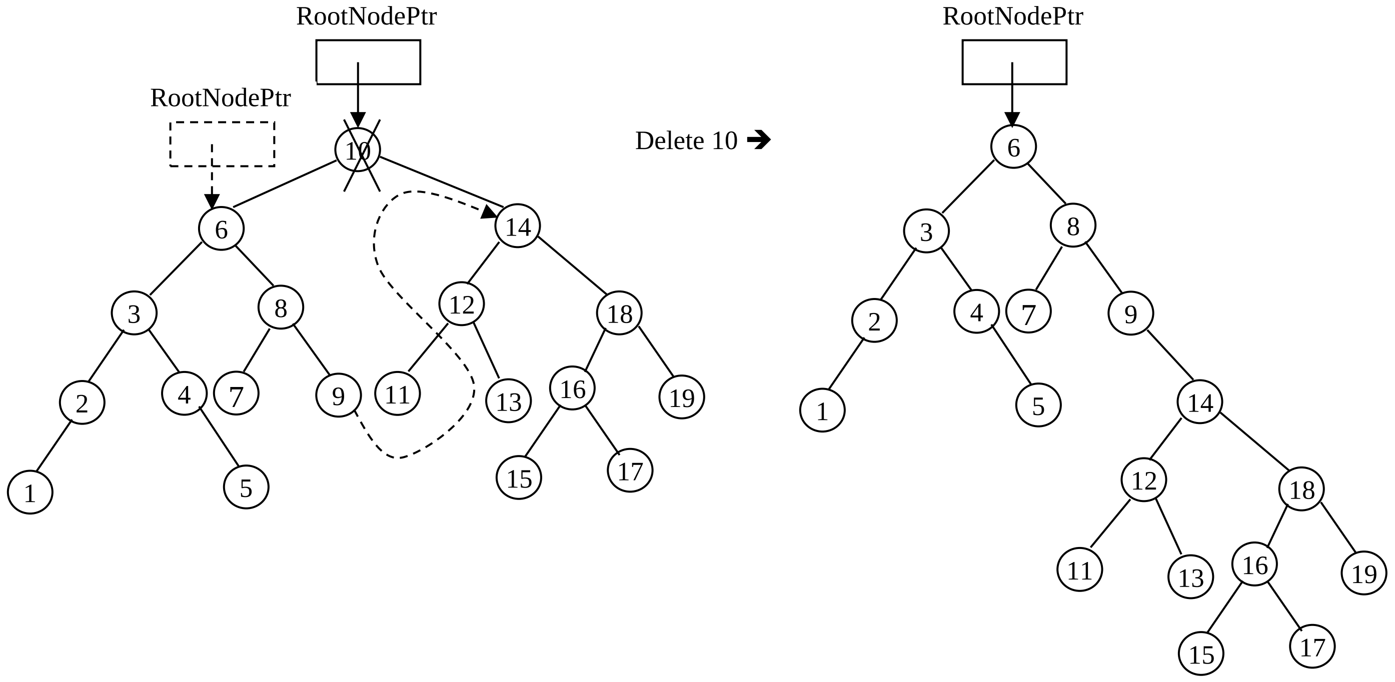

RootNodePtr
Delete 6 ➔
RootNodePtr

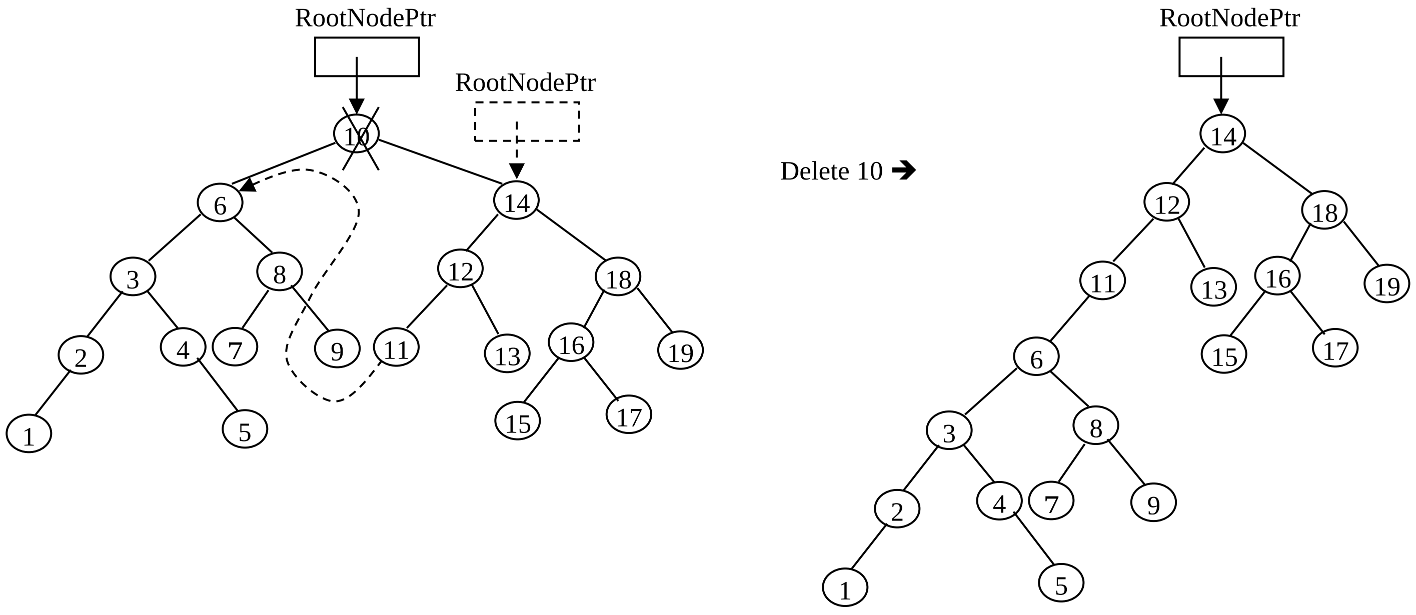

RootNodePtr
Delete 6 ➔
RootNodePtr

Case 4: Deleting the root node, 10

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

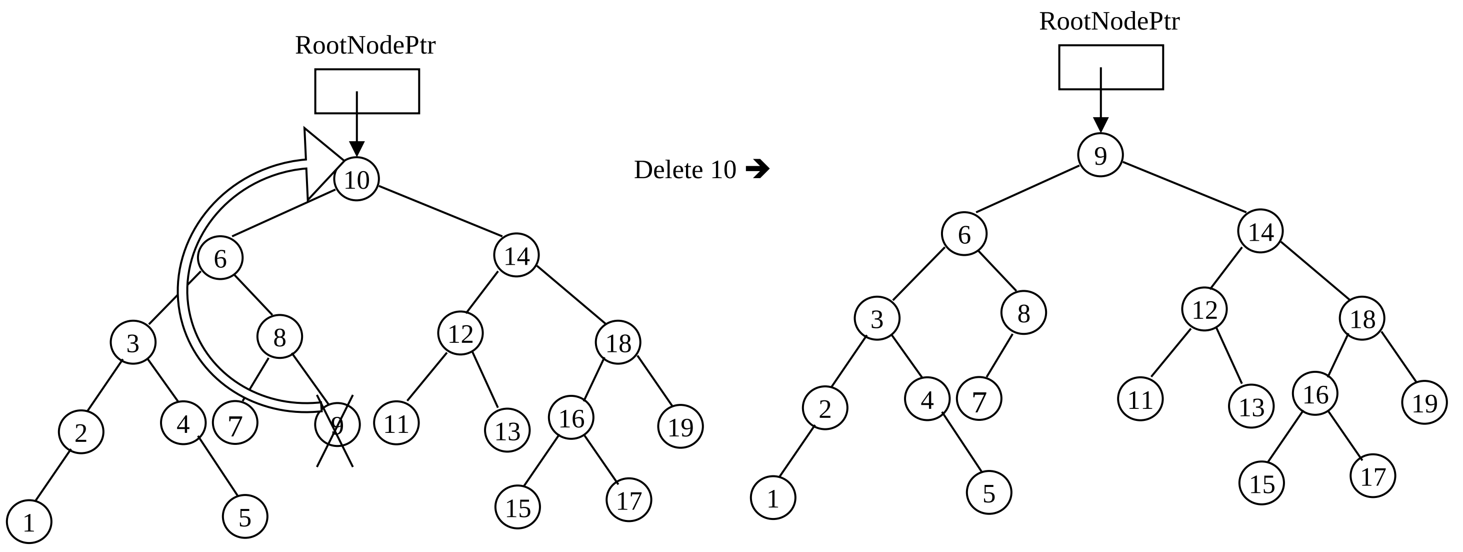
- If the tree has only one node the root node pointer is made to point to nothing (NULL)
- If the root node has left child
  - the root node pointer is made to point to the left child
  - the right child of the root node is made the right child of the node containing the largest element in the left of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

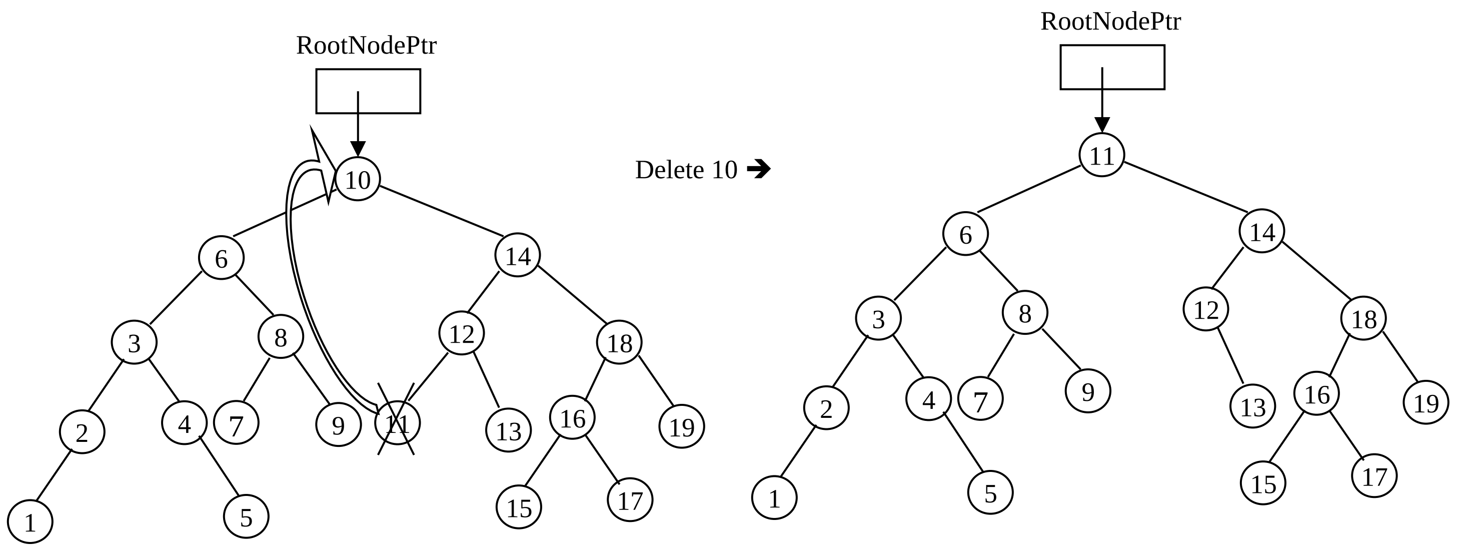




Approach 2: Deletion by copying- the following is done

- 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);
                        }
                }
        }
}
```

Chapter Four

4. Advanced Sorting and Searching Algorithms

4.1. Shell Sort

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

Algorithm:

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

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

Example: Sort the following list using shell sort algorithm.

5	8	2	4	1	3	9	7	6	0

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

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

Choose g_2 =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

4.2. 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).

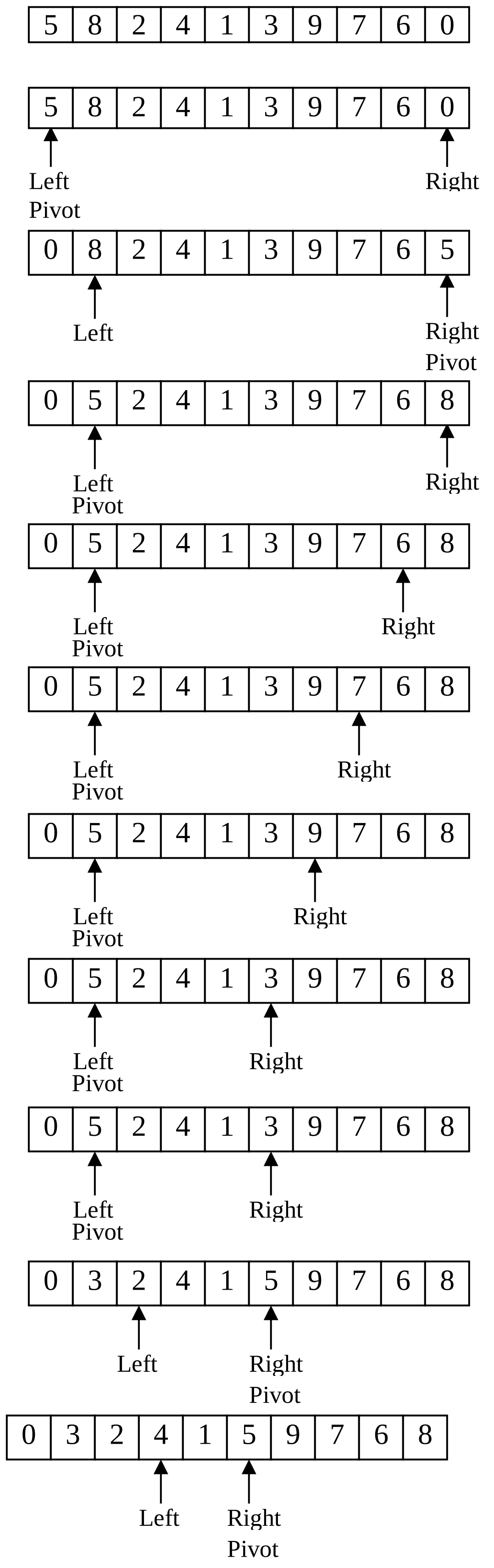
Algorithm:

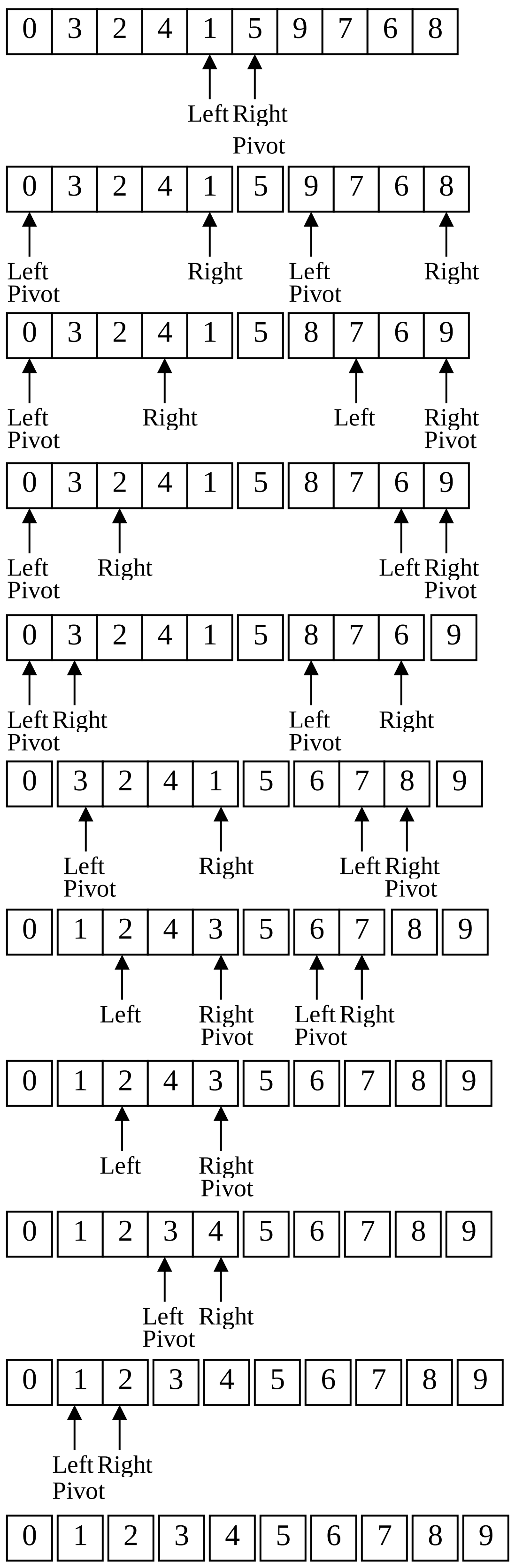
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
    - the left part has items less than or equal to the pivot value
    - the right part has items greater than or equal to the pivot value
3. Recursively sort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





4.3. *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).

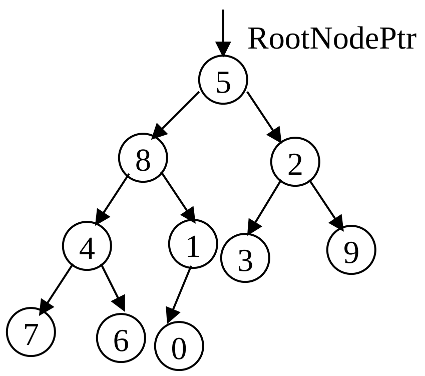
<u>Algorithm</u>:

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

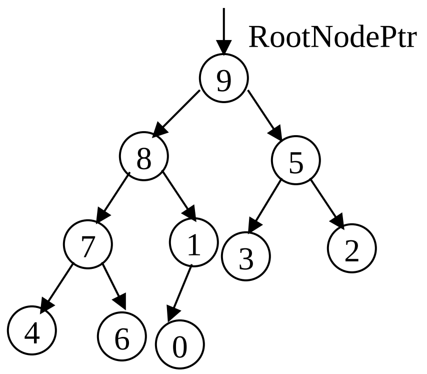
Example: Sort the following list using heap sort algorithm.

5	8	2	4	1	3	9	7	6	0

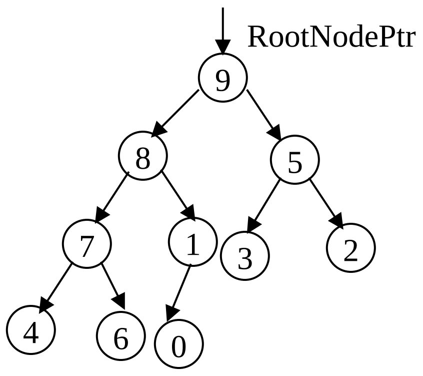
Construct the initial binary tree



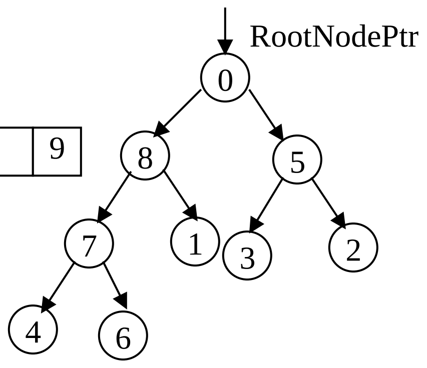
Construct the heap tree

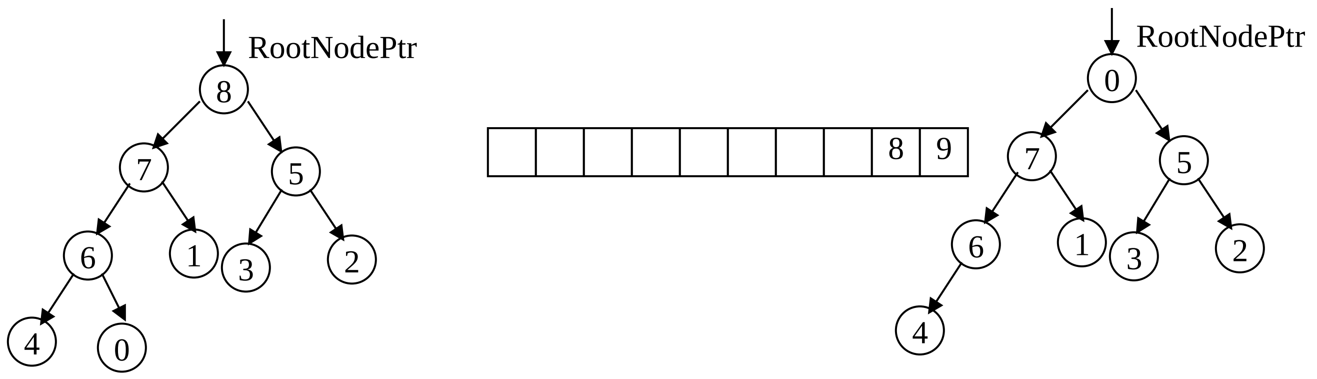


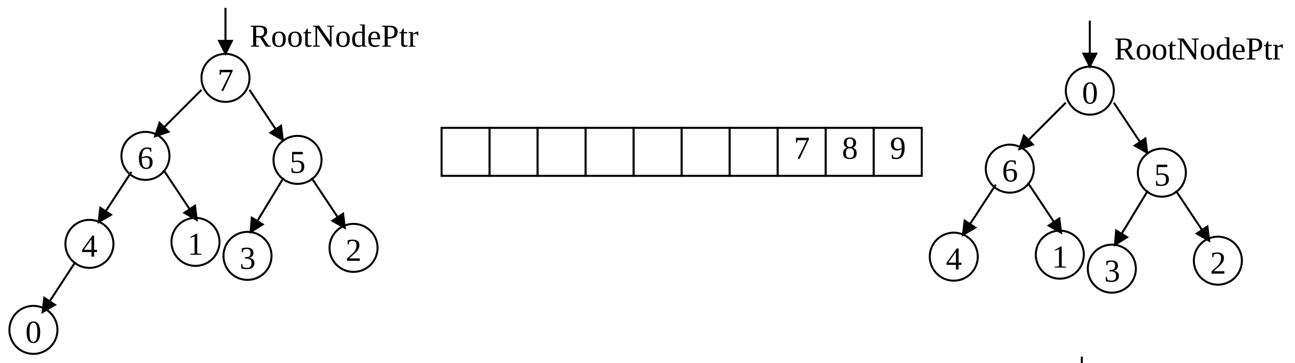
Swap the root node with the lowest, right most node and delete the lowest, right most value; insert the deleted value in the array in its proper position; adjust the heap tree; and repeat this process until the tree is empty.

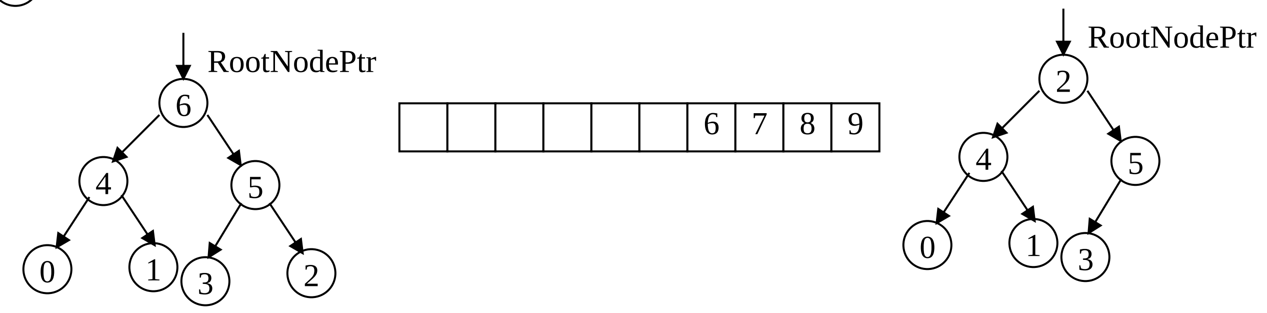


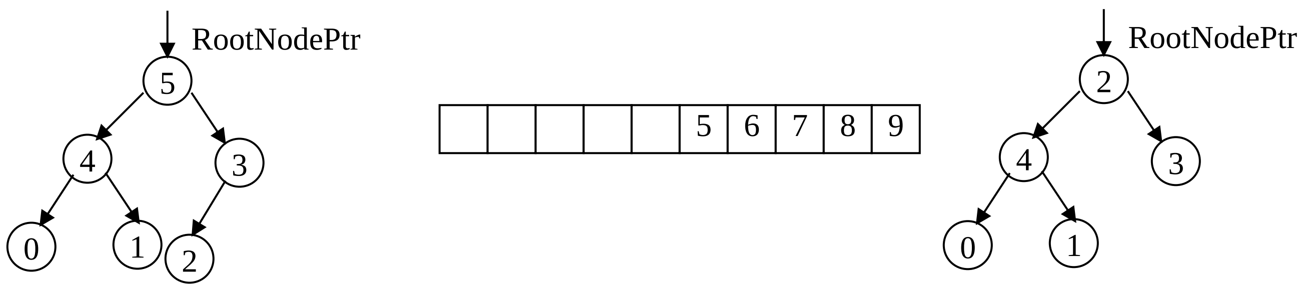
									9

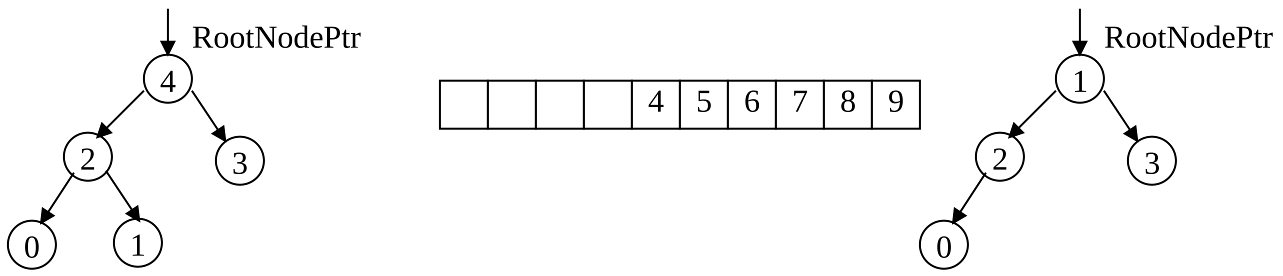


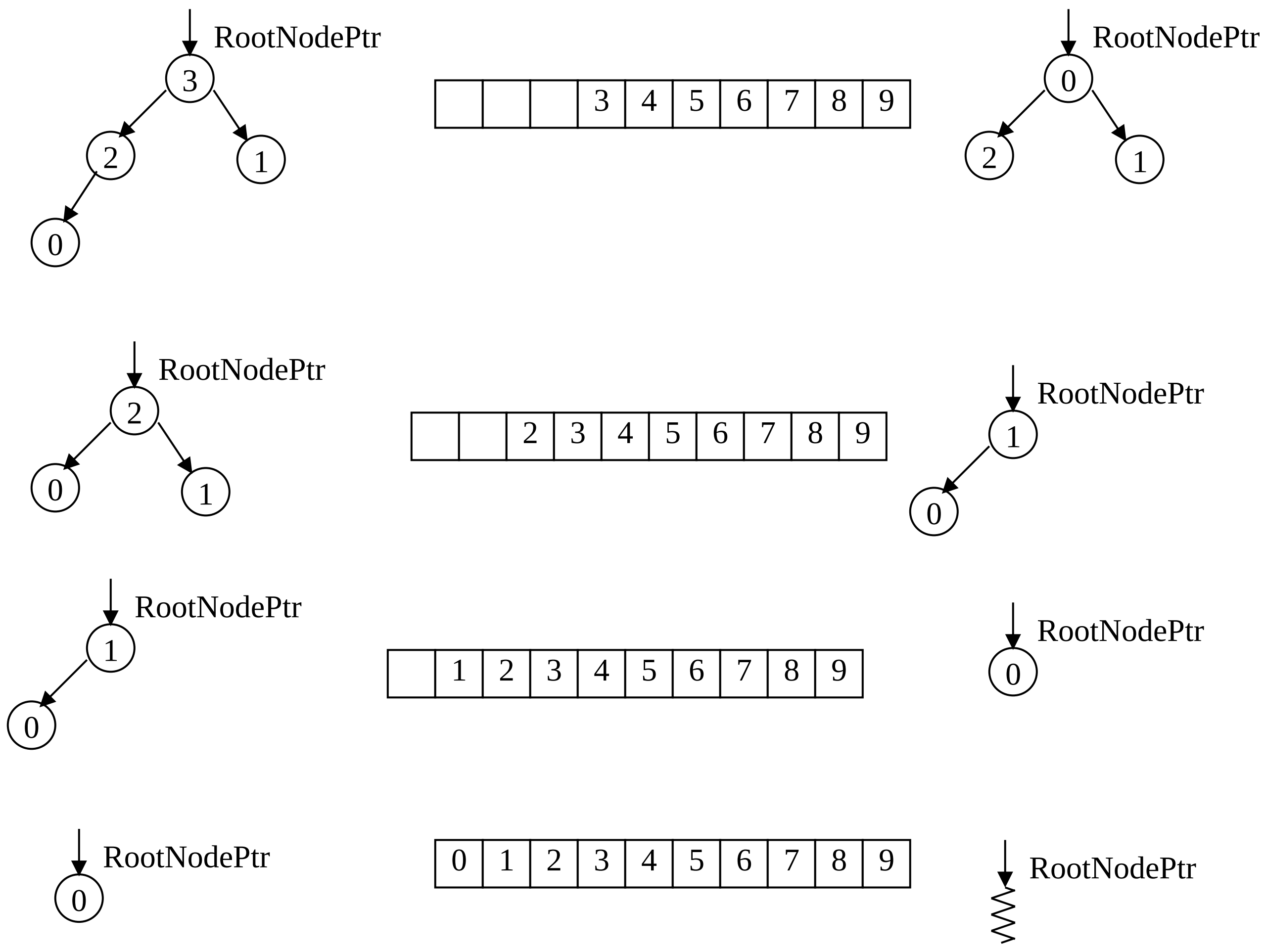

RootNodePtr
RootNodePtr
8 9


RootNodePtr
RootNodePtr
7 8 9


RootNodePtr
RootNodePtr
6 7 8 9


RootNodePtr
RootNodePtr
5 6 7 8 9


RootNodePtr
RootNodePtr
4 5 6 7 8 9


RootNodePtr
RootNodePtr
RootNodePtr
RootNodePtr
RootNodePtr
RootNodePtr
RootNodePtr
RootNodePtr

4.4. *Merge Sort*

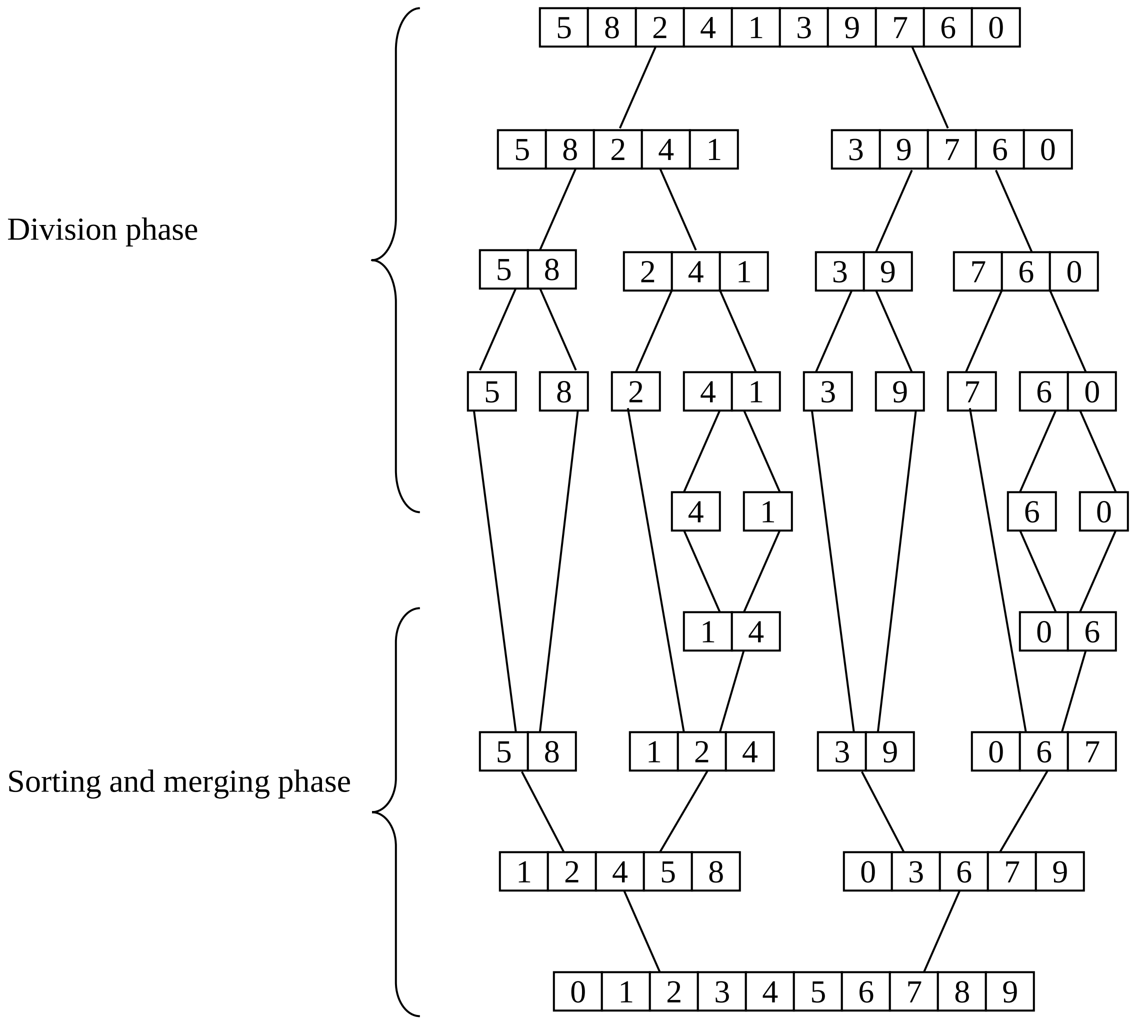
Like quick sort, merge sort uses divide and conquer strategy and its time complexity is O(nlogn).

Algorithm:

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

Example: Sort the following list using merge sort algorithm.

5	8	2	4	1	3	9	7	6	0



Chapter Five

5. Hashing Algorithm

All of the searching techniques we have seen so far operate by comparing the value being searched for with the values of a *key* value of each element. For example, when searching for an integer *val* in a binary search tree, we compare *val* with the integer (the *key*) stored at each node we visit. Such searching techniques vary in their complexity, but will always be more than *O(1)*.

Hashing is an alternative way of storing data that aims to greatly improve the efficiency of search operations. With hashing, when adding a new data element, the key itself is used to directly determine the location to store the element. Therefore, when searching for a data element, instead of searching through a sequence of key values to find the location of the data we want, the key value itself can be used to directly determine the location in which the data is stored. This means that the search time is reduced from *O(n)*, as in sequential search, or *O(log n)*, as in binary search, to *O(1)*, or constant complexity. Regardless of the number of elements stored, the search time is the same.

The question is, how can we determine the position to store a data element using only its key value? We need to find a function *h* that can transform a key value *K* (e.g. an integer, a string, etc.) into an index into a table used for storing data. The function *h* is called a *hash function*. If *h* transforms different keys into different indices it is called a *perfect hash function*. (A non-perfect hash function may transform two different key values into the same index.)

Consider the example of a compiler that needs to store the values of all program variables. The key in this case is the name of the variable, and the data to be stored is the variable's value. What hash function could we use? One possibility would be to add the ASCII codes of every letter in the variable name and use the resulting integer to index a table of values. But in this case the two variables abc and cba would have the same index. This problem is known as *collision* and will be discussed later in this handout. The worth of a hash function depends to a certain extent on how well it avoids collisions.

5.1. Hash Functions

Clearly there are a large number of potential hash functions. In fact, if we wish to assign positions for n items in a table of size m, the number of potential hash functions is m^n, and the number of perfect hash functions is $\frac{m!}{(m-n)!}$. Most of these potential functions are not of practical use, so this section discusses a number of popular types of hash function.

Division

A hash function must guarantee that the value of the index that it returns is a valid index into the table used to store the data. In other words, it must be less than the size of the table. Therefore an obvious way to accomplish this is to perform a modulo (remainder) operation. If the key *K* is a number, and the size of the table is *Tsize*, the hash function is defined as *h(K) = K mod TSize*. Division hash functions perform best if the value of *TSize* is a prime number.

5.1.1. Folding

Folding hash functions work by dividing the key into a number of parts. For example, the key value 123456789 might be divided into three parts: 123, 456 and 789. Next these parts are combined together to produce the target address. There are two ways in which this can be done: *shift folding* and *boundary folding*.

In shift folding, the different parts of the key are left as they are, placed underneath one another, and processed in some way. For example, the parts 123, 456 and 789 can be added to give the result 1368. To produce the target address, this result can be divided modulo *TSize*.

In boundary folding, alternate parts of the key are left intact and reverse. In the example given above, 123 is left intact, 456 is reversed to give 654, and 789 is left intact. So this time the numbers 123, 654 and 789 are summed to give the result 1566. This result can be converted to the target address by using the modulo operation.

5.1.2. Mid-Square Function

In the mid-square method, the key is squared and the middle part of the result is used as the address. For example, if the key is 2864, then the square of 2864 is 8202496, so we use 024 as the address, which is the middle part of 8202496. If the key is not a number, it can be pre-processed to convert it into one.

Extraction

In the extraction method, only a part of the key is used to generate the address. For the key 123456789, this method might use the first four digits (1234), or the last four (6789), or the first two and last two (1289). Extraction methods can be satisfactory so long as the omitted portion of the key is not significant in distinguishing the keys. For example, at Mizan-Tepi University many student ID numbers begin with the letters "SCIR", so the first three letters can be omitted and the following numbers used to generate the key using one of the other hash function techniques.

5.1.3. Radix Transformation

If *TSize* is 100, and a division technique is used to generate the target address, then the keys 147 and 247 will produce the same address. Therefore this would not be a perfect hash function. The radix transformation technique attempts to avoid such collisions by changing the number base of the key before generating the address. For example, if we convert the keys 147_{10} and 247_{10} into base 9, we get 173_9 and 304_9. Therefore, after a modulo operation the addresses used would be 47 and 04. Note, however, that radix transformation does not completely avoid collisions: the two keys 147_{10} and 66_{10} are converted to 173_9 and 73_9, so they would both hash to the same address, 73.

5.1.4. Collision Resolution

If the hash function being used is not a perfect hash function (which is usually the case), then the problem of collisions will arise. Collisions occur when two keys hash to the same address. The chance of collisions occurring can be reduced by choosing the right hash function, or by increasing the size of the table, but it can never be completely eliminated. For this reason, any hashing system should adopt a *collision resolution* strategy. This section examines some common strategies.

5.1.**4.1. Open Addressing**

In open addressing, if a collision occurs, an alternative address within the table is found for the new data. If this address is also occupied, another alternative is tried. The sequence of alternative addresses to try is known as the *probing sequence*. In general terms, if position *h(K)* is occupied, the probing sequence is

$$norm(h(K)+p(1)), norm(h(K)+p(2)), K, norm(h(K)+p(i)), K$$

Where function *p* is the probing function and *norm* is a normalization function that ensures the address generated is within an acceptable range, for example the modulo function.

The simplest method is *linear probing*. In this technique the probing sequence is simply a series of consecutive addresses; in other words the probing function *p(i) = i*. If one address is occupied, we try the next address in the table, then the next, and so on. If the last address is occupied, we start again at the beginning of the table. Linear probing has the advantage of simplicity, but it has the tendency to produce *clusters* of data within the table. For example, Figure 1 shows a sequence of insertions into a hash table using the following key/value pairs:

Key	Value
15	A
2	B
33	C
5	D
19	E
22	F
9	G
32	H

The first three insertions (A, B and C) do not result in collisions. However, when data D is inserted it hashes to the address 5, which is currently occupied by A, so it is placed in the next address. Similarly, when data F is inserted at address 2 it collides with B, so we try address 3 instead. Here it collides with C, so we have to place it at address 4. Data G also collides with E at address 9, so because 9 is the last address in the table we place it at address 1. Finally data H collides with 5 different elements before being successfully placed at address 7.

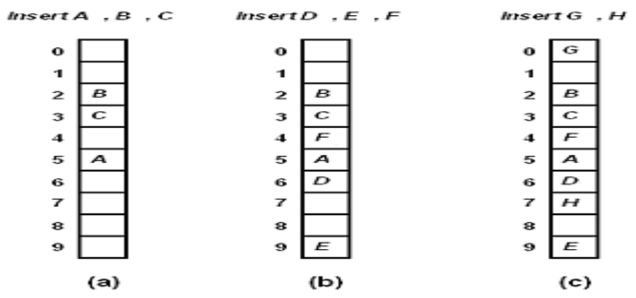


Figure 1 – Collision resolution using linear probing.

We can see in Figure 1 that there is a cluster of 6 elements (from addresses 2 to 7) stored next to each other. The problem with clusters is that the probability of a collision for a key is dependent on the address that it hashes to. Clustering can be avoided by using a more careful choice of probing function *p*. One possible choice is to use the sequence of addresses

$$h(K)+i^2, h(K)-i^2, \text{ for } i = 1, 2, \ldots, (Tsize - 1)/2.$$

Including the original attempt to hash *K*, this formula results in the sequence *h(K), h(K) + 1, h(K) – 1, h(K) + 4, h(K) – 4,* etc. All of these addresses should be divided modulo *Tsize*. For example, for the H_2 data in Figure 1, we first try address 2, then address 3 (2 + 1), and then address 1 (2 – 1), where the data is successfully placed. This technique is known as *quadratic probing*. Quadratic probing results in fewer clusters than linear probing, but because the same probing sequence is used for every key, sometimes clusters can build up away from the original address. These clusters are known as *secondary clusters*.

Another possibility, which avoids the problem of secondary clusters, is to use a different probing sequence for each key. This can be achieved by using a random number generator seeded by a value that is dependent on the key. Remember that random number generators always require a seed value, and if the same seed is used the same sequence of 'random' numbers will be generated. So if, for example, the value of the key (if it is an integer), were to be used, each different key would generate a different sequence of probes, thus avoiding secondary clusters.

Another way to avoid secondary clusters is to use *double hashing*. Double hashing uses two different hashing functions: one to find the primary position of a key, and another for resolving conflicts. The idea is that if the primary hashing function, *h(K)*, hashes two keys K_1 and K_2 to the same address, then the secondary hashing function, $h_p(K)$, will probably not. The probing sequence is therefore

$$h(K), h(K)+h_p(K), h(K)+2\cdot h_p(K), \ldots, h(K)+i\cdot h_p(K), \ldots$$

Experiments indicate that double hashing generally eliminates secondary clustering, but using a second hash function can be time-consuming.

5.1.4.2. Chaining

In *chaining*, each address in the table refers to a list, or *chain*, of data values. If a collision occurs the new data is simply added to end of the chain. Figure 2 shows an example of using chaining for collision resolution.

Provided that the lists do not become very long, chaining is an efficient technique. However, if there are many collisions the lists will become long and retrieval performance can be severely degraded. Performance can be improved by ordering the values in the list (so that an exhaustive search is not necessary for unsuccessful searches) or by using self-organising lists.

An alternative version of chaining is called *coalesced hashing*, or *coalesced chaining*. In this method, the link to the next value in the list actually points to another table address. If a collision occurs, then a technique such as linear probing is used to find an available address, and the data is placed there. In addition, a link is placed at the original address indicating where the next data element is stored. Figure 3 shows an example of this technique. When the keys D5 and F2 collide Figure 3b, linear probing is used to position the keys, but links from their original hashed addresses are maintained. Variations on coalesced hashing include always placing colliding keys at the end of the table, or storing colliding keys in a special reserved area known as the *cellar*. In both cases a link from the original hashed address will point to the new location. The advantage of coalesced hashing is that it avoids the need to make a sequential search through the table for the required data in the event of collisions.

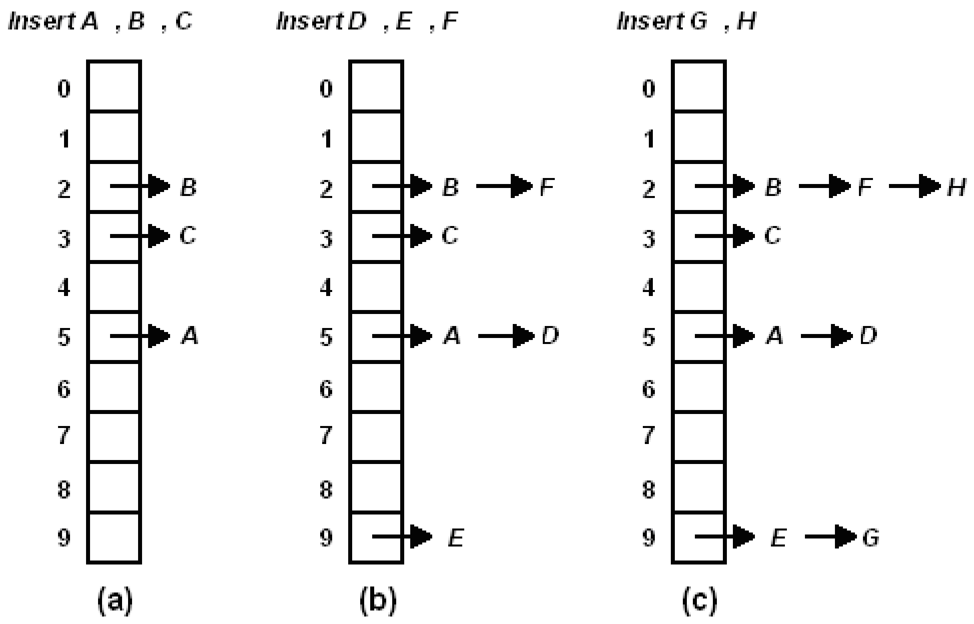


Figure 2 – Collision resolution using chaining.

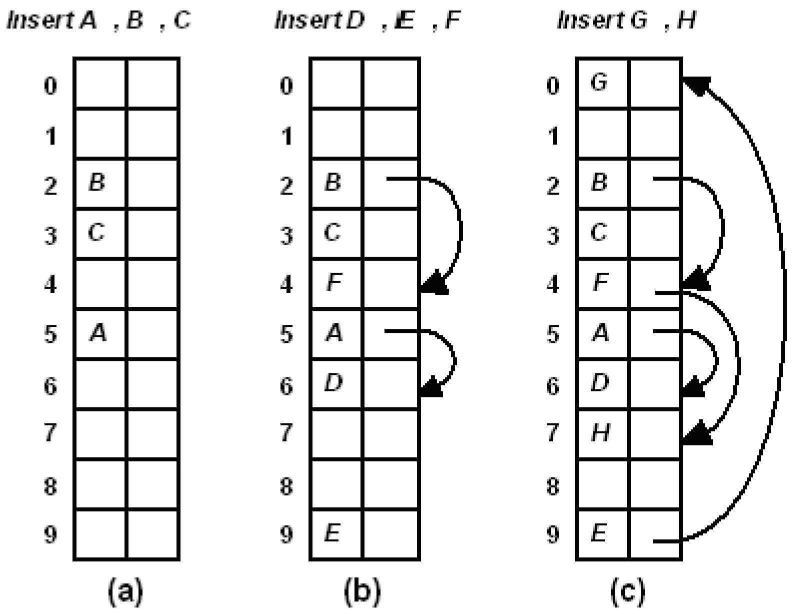


Figure 3 – Collision resolution using coalesced hashing.

5.1.4.3. Bucket Addressing

Bucket addressing is similar to chaining, except that the data are stored in a *bucket* at each table address. A bucket is a block of memory that can store a number of items, but not an unlimited number as in the case of chaining.

Bucketing reduces the chance of collisions, but does not totally avoid them. If the bucket becomes full, then an item hashed to it must be stored elsewhere. Therefore bucketing is commonly combined with an open addressing technique such as linear or quadratic probing. Figure 4 shows an example of bucketing that uses a bucket size of 3 elements at each address.

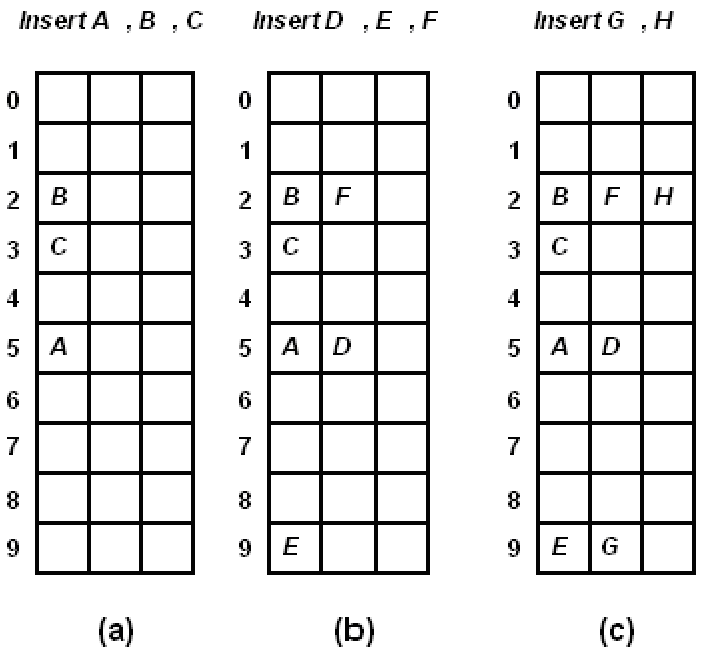


Figure 4 – Collision resolution using bucketing.