Design and Analysis of Algorithms

Chapter 1

Chapter description

This chapter focuses on elementary data structures.

Topics Include:

- Review of the basic data structures:
- Heap and Heap Sort, Hashing, sets representation UNION, FIND operation;
- Analysis of algorithm, Order notation

Objectives

After completing this chapter, you will be able to:

- Understand basic characteristics of algorithm
- Understand ways of algorithm analysis framework
- Having knowhow about elementary data structure for algorithm analysis

1.1. INTRODUCTION

- Although there is **no universally agreed**-on wording to describe this notation, there is general agreement about what the concept means:
- An **algorithm** is a set of **unambiguous** steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks.
- An algorithm is an efficient method that can be expressed within finite amount of **time and space**.
- An algorithm is the best way to represent the solution of a particular problem in a very simple and efficient way.
- If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that the algorithm is **independent** from any programming languages.

Here the following are another way of algorithm definitions.

- ✓ An algorithm is a sequence of computational steps that transform the input into the output.
- ✓ An algorithm is a sequence of operations performed on data that have to be organized in the data structures.
- ✓ A finite set of instruction that specify a sequence of operations to be carried out in order to
- ✓ solve a specific problem or class of problems is called an algorithm.
- ✓ An algorithm is an abstraction of a program to be executed on a physical machine (model computation).
- ✓ An algorithm is defined as set of instructions to perform a specific task within finite number of steps.
- ✓ Algorithm is defined as a step-by-step procedure to perform a specific task within finite number of steps.

✓ It can be defined as a sequence of definite and effective instructions, while terminates with the production of correct output from the given input.

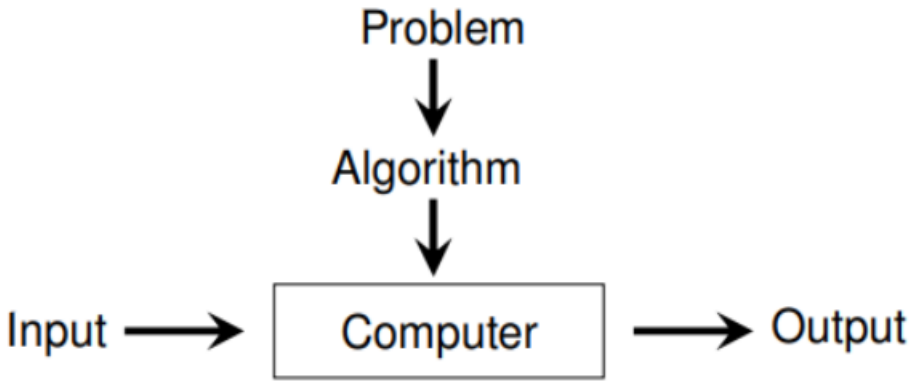


Fig1.1: the figure shows how the algorithm works.

In general, an Algorithm is a finite sequence of **unambiguous finite set of instructions** that, if followed, accomplishes to solve a particular problem.

In addition, **all algorithms should satisfy** the following criteria.

1) **input**: - Zero or more quantities are externally supplied.

2) **output: -** At least one quantity is produced.

3) **definiteness: -** Each instruction is clear and unambiguous.

4) **finiteness**: - If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

5) **Effectiveness**: Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

6) **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.

7) **feasibility**: It must be possible to perform each instruction.

8) **correctness**: It must compute correct answer for all possible legal inputs.

9) **language independence**: It must not depend on any one programming language.

10) **completeness**: It must solve the problem completely.

11) **efficiency**: It must solve with the least number of computational resources such as time and space.

12) **generality**: Algorithm should be valid on all possible inputs.

Algorithm Design

- The important aspects of algorithm design include **creating an efficient algorithm** to solve a problem in an efficient way using **minimum time and space**.
- To solve a problem, different approaches can be followed.
- Some of them can be efficient with respect to **time consumption**, whereas other approaches may be memory efficient.
- However, one has to keep in mind that both time consumption and memory usage **cannot** be optimized simultaneously.
- If we require an algorithm to **run** in **lesser time, we have to invest in more memory** and if we require an algorithm to **run** with **lesser memory**, we need to have **more time**.

Problem Development Steps

The following steps are involved in solving computational problems.

- Understand the problem
- Problem definition
- Development of a model
- Specification of an Algorithm
- Designing an Algorithm
- Checking the correctness of an Algorithm
- Analysis of an Algorithm
- Implementation of an Algorithm
- Program testing
- Documentation

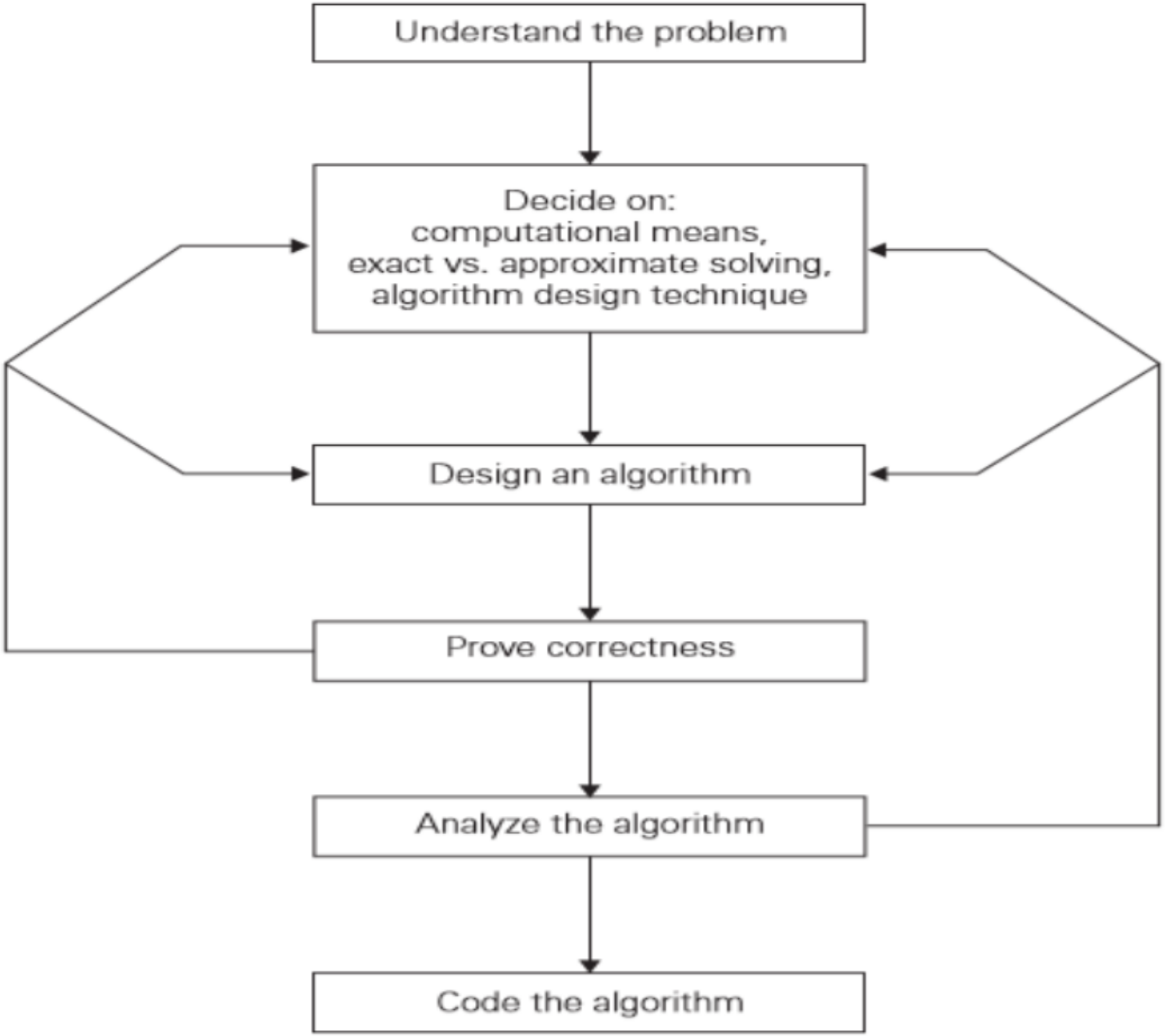


fig 1.2: Algorithm analysis and design process

⊕ Important problem types

In the limitless sea of problems, one encounters in computing, there are a few areas that have attracted particular attention from researchers. By and large, their interest has been driven either by the problem's practical importance or by some specific characteristics making the problem an interesting research subject; fortunately, these two motivating forces reinforce each other in most cases. In this section, we are going to introduce the most important problem types:

1. Sorting
2. Searching
3. String processing
4. Graph problems
5. Combinatorial problems
6. Geometric problems
7. Numerical problems

i. **Issues or study of Algorithm:** How to device or design an algorithm creating an algorithm.

- How to express algorithm definiteness.
- How to analysis an algorithm time and space complexity.
- How to validate algorithm fitness.
- Testing the algorithm checking for error

ii. Algorithm Specification:

An algorithm can be specified in

1. Graphical representation like flow chart
2. Simple English (pseudo code)
3. Programming language like c++/java
4. Combination of above methods.

1. **Flowchart**: - Graphic representation of an algorithm is called as flowchart and it's easy to understand the logic of complicated and lengthy problems

Example 1.1

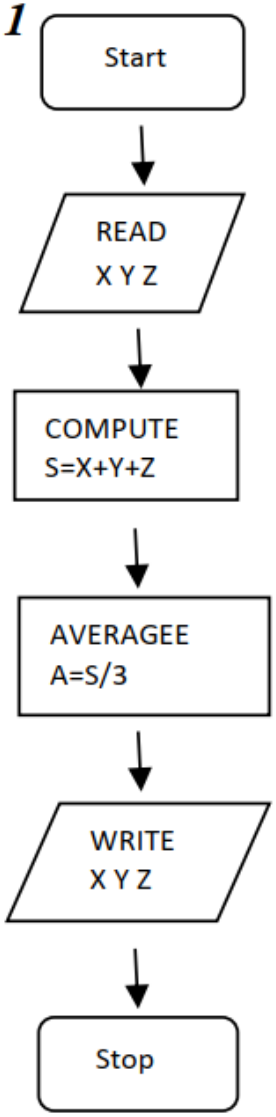


Fig-1.3: Algorithm for finding the average of three numbers

2. Pseudo-code Method: -

Pseudo code is a type of structured English that is used to specify an algorithm. This cannot be compiled and not executed. In this method, we should typically describe algorithms as program, which resembles language like Pascal & algol.

Pseudocode gives a high-level description of an algorithm without the ambiguity associated with plain text but also without the need to know the syntax of a particular programming language.

The running time can be estimated in a more general manner by using Pseudocode to represent the algorithm as a set of fundamental operations which can then be counted.

Example 1.2: pseudo-code to find the average of three numbers

```
START
READ values of X, Y, Z
COMPUTE S=X+Y+Z
COMPUTE A=S/3
WRITE the value of A
STOP
```

Difference between Algorithm and Pseudocode

- An **algorithm** is a formal definition with some specific characteristics that describes a process, which could be executed by a **Turing-complete** computer machine to perform a specific task.
- Generally, the word "algorithm" can be used to describe any high-level task in computer science.
- On the other hand, **pseudo code** is an informal and (often **rudimentary**) human readable description of an algorithm leaving many granular details of it.
- Writing a pseudo code has **no restriction** of styles and its only objective is to describe the high-level steps of algorithm in a much realistic manner in natural language.

For example, following is an algorithm for Insertion Sort.

Algorithm: **Insertion-Sort**

Input: A list L of integers of length n

Output: A sorted list L1 containing those integers present in L

Step 1: Keep a sorted list L1 which starts off empty

Step 2: Perform Step 3 for each element in the original list L

Step 3: Insert it into the correct position in the sorted list L1.

Step 4: Return the sorted list

Step 5: Stop

Here is a pseudocode which describes how the high-level abstract process mentioned above in the algorithm Insertion-Sort could be described in a more realistic way.

```
for i ← 1 to length(A)
   x ← A[i]
   j ← i
while j > 0 and A[j-1] > x
      A[j] ← A[j-1]
      j ← j - 1
   A[j] ← x
```

In general, Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

1.2. ANALYSIS OF ALGORITHM

- How to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.
- Algorithms are often quite different from one another, though the objective of these algorithms are the same.
- For example, we know that a set of numbers can be sorted using different algorithms.
- Number of comparisons performed by one algorithm may vary with others for the same input.
- At the same time, we need to calculate the memory space required by each algorithm.
- Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation).

1.2. ANALYSIS OF ALGORITHM ...cont

Generally, we perform the following types of analysis:

- **Worst-case:** The maximum number of steps taken on any instance of size a.
- **Best-case**: The minimum number of steps taken on any instance of size a.
- **Average case**: An average number of steps taken on any instance of size a.
- **Amortized**: A sequence of operations applied to the input of size a averaged over time.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as time complexity, or volume of memory, known as space complexity.

- To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa.
- In this context, if we compare bubble sort and merge sort.
- Bubble sort does not require additional memory, but merge sort requires additional space.
- Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment, where memory is very limited.

1.2.1. Space Complexity

The space complexity of an algorithm is the amount of memory it needs to run to compilation. The Space needed by each of these algorithms is seen to be the sum of the following component.

I. A fixed part that is independent of the characteristics (eg: number, size) of the inputs and outputs. The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

II. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that is depends on instance characteristics), and the recursion stack space.

Example 1.3

```
Algorithm sum (a, n)
{
  S: =0.0;
  for i :=1 to n do
        s: = s+a[i];
return s;
}
```

1.2.2. Time Complexity

- The time complexity of an algorithm is the amount of computer time it needs to run to compilation.
- The time T(p) taken by a program P is the sum of the compile time and the run time (execution time)
- The compile time does not depend on the instance characteristics. Also, we may assume that a compiled program will be run several times without recompilation.
- This rum time is denoted by T(p) (instance characteristics). The number of steps any problem statement is assigned depends on the kind of statement.
- For example, comments →0 steps.
- Assignment statements → 1 step.
- Interactive statement such as for, while & repeat-until→Control part of the statement.

Analysis Rules:

1. We assume an arbitrary time unit.
2. Execution of one of the following operations takes times 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, analyses 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; // *n* loops of 2 units
   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, & 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 // no assignment in two

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)*

Loop Incrementation Other than 1

```
for (int i = 0; i < n; i += c)
    statement(s);
```

- Adding to the loop counter means that the loop runtime grows linearly when compared to its maximum value ***n***.
- Loop executes its body exactly ***n / c*** times.

```
for (int i = 0; i < n; i *= c)
    statement(s);
```

- Multiplying the loop counter means that the maximum value ***n*** must grow exponentially to linearly increase the loop runtime; therefore, it is logarithmic.
- Loop executes its body exactly $\log^{c}_{n}$ times.

```
for (int i = 0; i < n * n; i += c)
    statement(s);
```

- The loop maximum is n^2, so the runtime is quadratic.
- Loop executes its body exactly (n^2/c) times.

Example:

1. int doubler (int n)

```
{
    int res = 0;
    for (int i = 1; i <= n; i+=2)
        res = res + i;
    return res;  }
```

Time Units to Compute

---

1 for the assignment.

In loop:

1 assignment, (*n/2)+1* tests,

n/2 increments.

n/2 loops of 2 units for an assignment and addition.

1 for the return statement.

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

Example 2.

```
int doubler (int n)
{    int res = 0;
     for (int i = 1; i <=n; i+=3)
          res = res + i;
     return res; }
```

Time Units to Compute

---

1 for the assignment.

In loop:

1 assignment, (*n/3)+1* tests,

n/3 increments.

n/3 loops of 2 units for an assignment and addition.

1 for the return statement.

T (n)= *1+(1+n/3+1+n/3)+2n/3+1 = 4n/3+4 = O(n)*

2.1.5. 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;
}}
```

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

$$\max\left(N, 2N^2\right) = 2N^2$$

In general, we can determine the number of steps needed by a program to solve a particular problem instance in **two ways**.

❖ **First**, we introduce a variable, count into the program statement to increment count with initial value 0. Statement to increment count by the appropriate amount are introduced into the program. This is done so that each time a statement in the original program is executed count is incremented by the step count of that statement.

Example 1.4

```
Algorithm sum(a,n)
{
s= 0.0;
count = count+1; // for assignment statement
for i=1 to n do
{
count =count+1; // For if
s=s+a[i];
count=count+1; // for assignment statement
}
count=count+1; // for last time of for
count=count+1; // for return
return s;
}
```

If the count is zero to start with, then it will be 2n+3 on termination. So each invocation of sum executes a total of 2n+3 steps.

❖ **The second** method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

→First determine the number of steps per execution **(s/e)** of the statement and the total number of times (i.e. frequency) each statement is executed.

→By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

Statement	*S/e*	*Frequency*	*Total*
1. Algorithm Sum(a,n)	0	-	0
2.{	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2n+3

Table 1.1: - algorithm analysis by step counting

✓ Best-Case Step count o Minimum number of steps executed by the algorithm for the given parameters.

✓ Worst-Case Step count o Maximum number of steps executed by the algorithm for the given parameters.

✓ Average-Case Step count o Average number of steps executed by an algorithm.

In exactness of step count

- Both the instructions x=y and x=y+z+(x/y) count as one step.
- Because of the inexactness of what a step count stands for, the exact step count is not very useful for comparison of algorithms.

Asymptotic efficiency :

- ✓ Asymptotic efficiency means study of algorithms efficiency for large inputs
- ✓ Asymptotic notation is a way to describe the performance of algorithms in terms of **how their runtimes grow** as the input size increases.
- ✓ This is helpful when we want to **compare different algorithms and determine which will be faster** for significant inputs.
- ✓ To compare two algorithms with running times *f(n)*and *g(n),*we need a rough measure that characterizes how fast each function grows as n grows.
  - ✓ *Hint:* **use *rate of growth***
- ✓ Compare functions **asymptotically!** (i.e., for large values of *n*)
- ✓ **1 < logn < n < nlogn < n2 < n3 < 2n < n!** (The Increasingly order of growth rate)
- ✓ **Why we need Asymptotic Notation?** Asymptotic notation is essential because it allows us to analyze and compare the efficiency of algorithms in a standardized and abstract way.

Rate of Growth

Example 1.5:- f(n)=n2+100n+log10n+1000

The low order terms and constants in a function are relatively insignificant for **large n** in n2+ 100n+ log10n+ 1000 **~ n2**

i.e., we say that **n2+ 100n+ log10n+ 1000** and **n2** have the same rate of growth

***Table 1.2*:-** function's growth rate description

Some more examples

1. $n^4 + 100n^2 + 10n + 50$ is $\sim n^4$
2. $10n^3 + 2n^2$ is $\sim n^3$
3. $n^3 - n^2$ is $\sim n^3$

Constants

1. 10 is ~ 1
2. 1273 is ~ 1

1.3. ASYMPTOTIC/ORDER NOTATIONS

- Asymptotic/order notation describes the behaviour of functions for the large input size.
- The asymptotic behaviour of a function $\boldsymbol{f(n)}$ refers to the growth of $\boldsymbol{f(n)}$ as n gets large.
- We typically ignore small values of n, since we are usually interested in estimating how slow the program will be on large inputs.
- In asymptotic notation, time function of an algorithm is represented by $\mathbf{T(n)}$, where n is the input size. A good rule of thumb is that the slower the asymptotic growth rate, the better the algorithm. Though it's not always true.
- **For example,** a linear algorithm $\boldsymbol{f(n) = d * n + k}$ is always asymptotically better than a quadratic one, $\boldsymbol{f(n) = c.n2}$ +q
- Different types of asymptotic notations are used to represent the complexity of an algorithm
- Following asymptotic notations are used to calculate the running time complexity of an algorithm.

1.3.1 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:

1. Big-Oh Notation (O)
2. Big-Omega Notation (Ω)
3. Theta Notation (Θ)
4. Little-o Notation (o)
5. Little-Omega Notation (ω)

Example: 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*.

1.3.1.2. The Big-Oh Notation

- We use O-notation to give an upper bound on a function, to within a constant factor. Since O-notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm, by implication we also bound the running time of the algorithm on arbitrary inputs as well.
  Formal Definition: f(n)= O(g(n)) if there exist constants c, k ∈ $\mathcal{R}$+ 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 <= n2 for all n>=1**
  - **2n <= n! for all n>=4**
  - **log2n <= n for all n>=2**
  - **n <= nlog2n 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).

1.3.1 Asymptotic Analysis

2. f(n) = 3n2 +4n+1. Show that f(n)=O(n2).

4n <=4n2 for all n>=1 and 1<=n2 for all n>=1

3n2 +4n+1<=3n2+4n2+n2 for all n>=1

3n2 +4n+1<=8n2 for all n>=1

So we have shown that f(n)<=8n2 for all n>=1

Therefore, f (n) is O(n2) (c=8,k=1)

- ✓ The big oh notation describes an upper bound on the asymptotic growth rate of the function f.

Definition: [Big "oh''']

- ✓ f(*n*) = O(g(*n*)) (read as "*f* of *n* is big oh of *g* of *n*") if and only if there exist positive constants c and n0 such that f(*n*) <cg(*n*) for all n, n >n_0.
- ✓ The definition states that the function f(n) is at most c times the function g(n) except when n is smaller than n0. In other words, f(n) grows **slower** than or same rate as g(n)

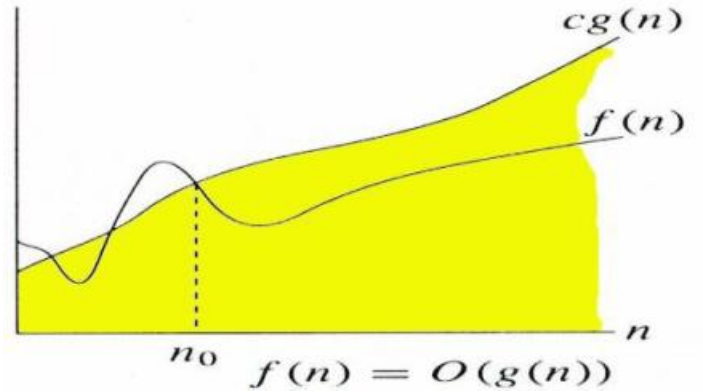


Fig 1.4 graphical description of Big oh notation

Examples 1.6

$f(n) = 3n+2$

$3n + 2 <= 4n$, for all $n >= 2$, $3n + 2 = \mathbf{O(n)}$

$f(n) = 10n^2+4n+2$

$10n^2+4n+2 <= 11n^2$, for all $n >= 5$, $10n2+4n+2 = \mathbf{O(n2)}$

b. Omega (Ω)notation:

The omega notation describes a lower bound on the asymptotic growth rate of the function f.

Definition: - [Omega]

$f(n) = \mathbf{\Omega}(g(n))$ (read as "*f* of *n* is omega of *g* of *n*") if and only if there exist positive constants *c* and n_0 such that $f(n)>cg(n)$ for all n, $n>n_0$.

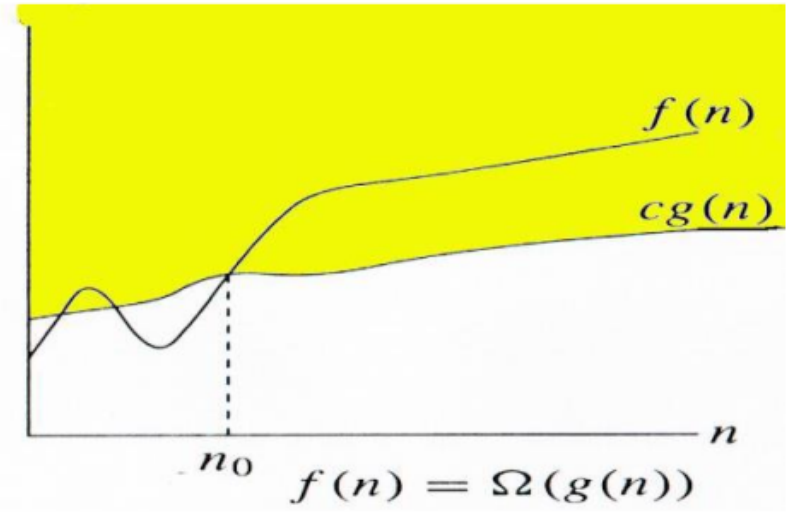


Fig 1.5 graphical description of Omega notation

The definition states that the function f(n) is at least c times the function g(n) except when n is smaller than n0. In other words, f(n)grows **faster** than or same rate as g(n).

Examples 1.7

f(n) = 3n+2

3n + 2 >= 3n, for all n >= 1, **3n + 2 = Ω (n)**

f(n) = **$10n^2+4n+2$**

$10n^2+4n+2 >= n^2$, for all n >= 1, **$10n^2+4n+2 = \Omega(n^2)$**

c. Theta (Θ) notation:

The Theta notation describes a tight bound on the asymptotic growth rate of the function f.

Definition: [Theta]

f(n) = Θ(g(n)) (read as "f of n is theta of g of n") if and only if there exist positive constants c1, c2, and n0 such that c1g(n) <f(n) <c2g(n)for all n, n>n_0.

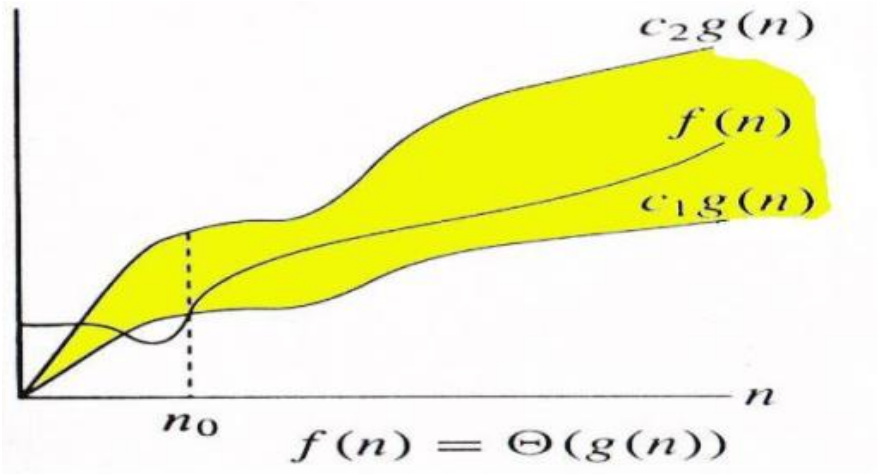


Fig 1.6 graphical description of Theta notation

The definition states that the function f(n) lies between c1 times the function g(n) and c2 times the function g(n) except when n is smaller than n0. In other words, f(n)grows same rate as g(n).

***Examples 1.8*: -**

i. Suppose f (n) = 3n+2, Prove that f(n) = **Θ(n)**

Solution

3n <= 3n + 2 <= 4n, for all n >= 2, 3n + 2 = **Θ(n)** Since c1 =3, c2=4 and n0=2

ii. Suppose f (n) = 10n2+4n+2, Prove that f(n)= **Θ(n)**

Solution

n2<= 10n2+4n+2 <= 11n2, for all n >= 5, 10n2+4n+2 = **Θ(n2)** Since c1 =1,c2=11 and n0=5

1 < logn < n < nlogn < n2 < n3 < 2n < n! (Increasingly order of growth rate)

To get a feel for how the various functions grow with n, you are advised to study the following table and figures:

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

Table 1.1: Basic asymptotic efficiency classes

Instance characteristic n							
Time	Name	1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
$\log n$	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
$n \log n$	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
$n!$	Factorial	1	2	24	40326	20922789888000	26313×10^{33}

Figure 1.7 Function values

$1 < \log^n < n < n\log^n < n^2 < n^3 < 2^n < n!$

(Increasingly order of growth rate)

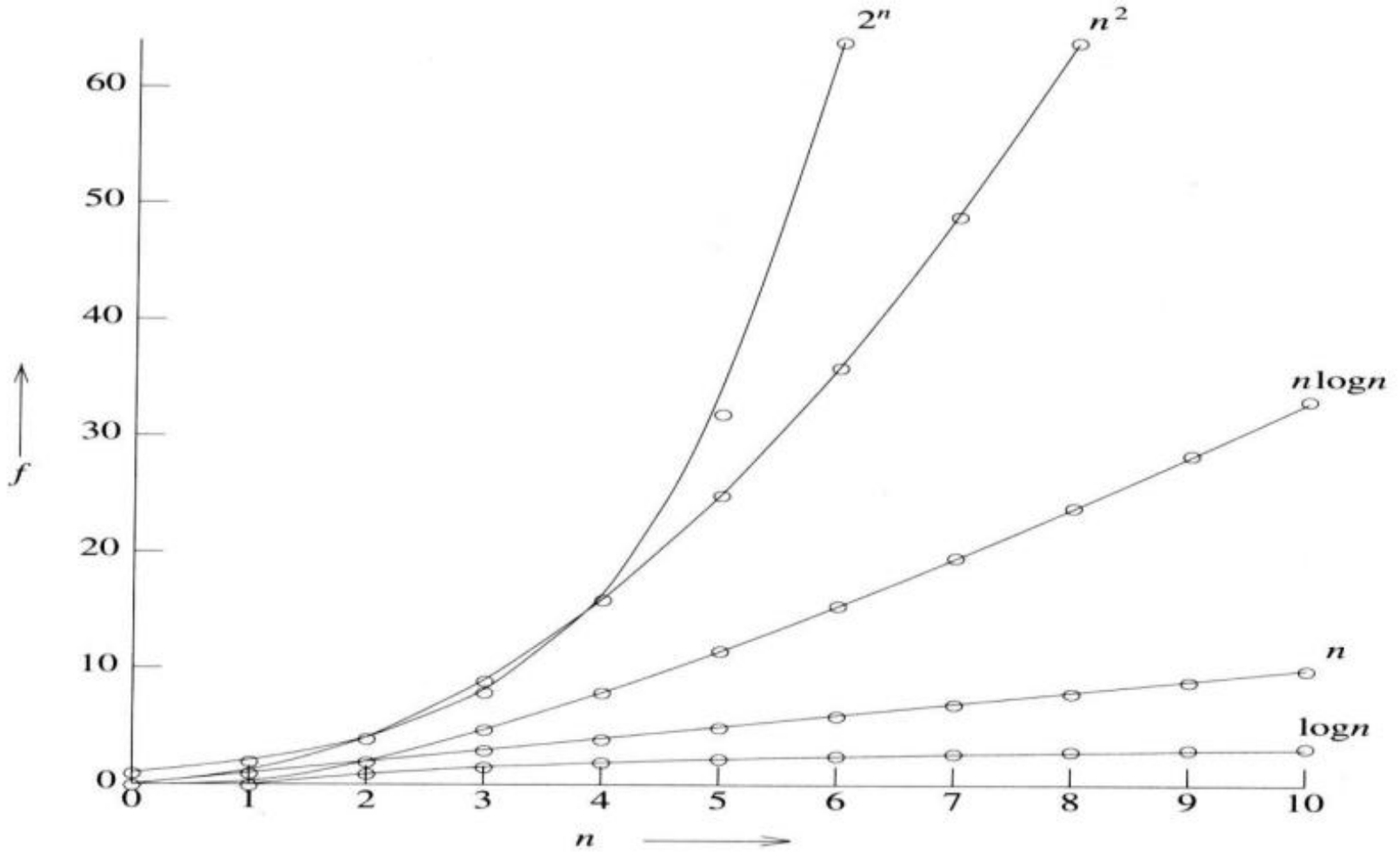


Figure 1.8 Plot of function values

The Big-Oh Notation (Order 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 ϵ $\mathcal{R}^+$ such that for all n$\geq$ k, f (n) $\leq$ c.g (n).

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

1.4. REVIEW OF ELEMENTARY DATA STRUCTURES

1.4.1. HEAP AND HEAP SORT

1.4.1.1. Priority Queue

- Many algorithms process items in a specific order. For example, suppose you must schedule jobs according to their importance relative to other jobs.
- Scheduling the jobs requires sorting them by importance, and then evaluating them in this sorted order.
- Priority queues are data structures that **provide more flexibility** than simple sorting, because they allow new elements to enter a system at arbitrary intervals.
- It is much **more cost-effective** to insert a new job into a priority queue than to re-sort everything on each such arrival.
- A data structure implementing a set S of elements, each associated with a key, The basic priority queue supporting the following four operations:
  - **Insert(S,x) :** insert element x into set S
  - **Max(S):** return element of S with largest key
  - **Extract_max(S**): return element of S with largest key and remove it from S
  - **Increase_key(S,x,k):** increase the value of element x' s key to new value k (assumed to be as large as current value)

$1 <= n$ 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 $n >= 1$.

$(c=15, k=1)$.

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).

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 $O(b^n)$ "$\forall b > 1$ and $k >= 0$

E.g. n^{20} is $O(1.05^n)$

Logarithms grow more slowly than powers

$\log_b n$ is $O(nk)$ $\forall$ "$b > 1$ and $k >= 0$

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

Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node

Doubly Linked List

- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes

Stacks

- The Stack ADT (Abstract Data Type) stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine or C++ runtime environment
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Queue

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - object front(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

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

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	Femal	Male	Male	Femal	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.

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

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

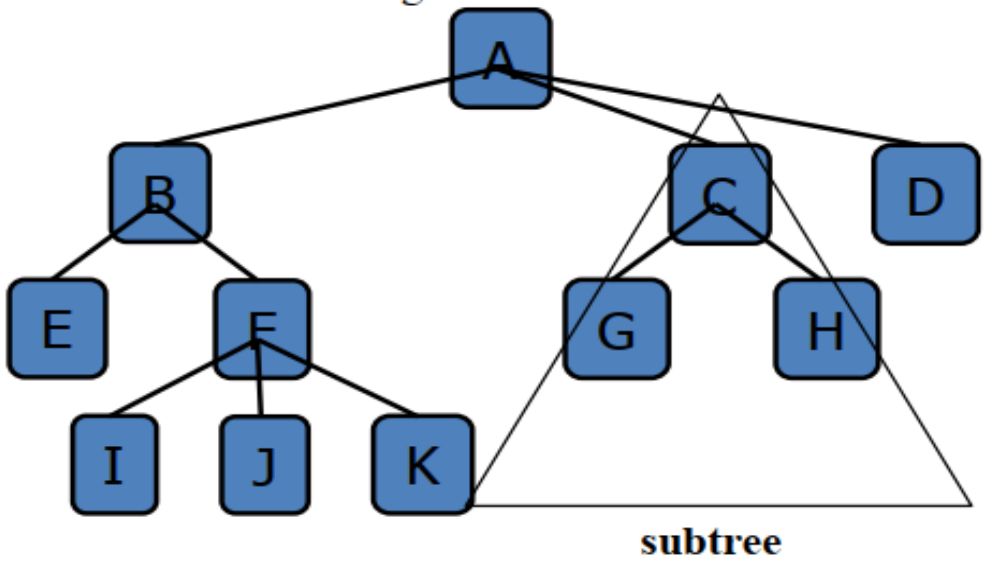
Aster	Meron	Abebe	Alemu	Belay	Kedir	Yonas
Femal	Femal	Male	Male	Male	Male	Male

Applications of Queues

- Direct applications
  - Waiting lines
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

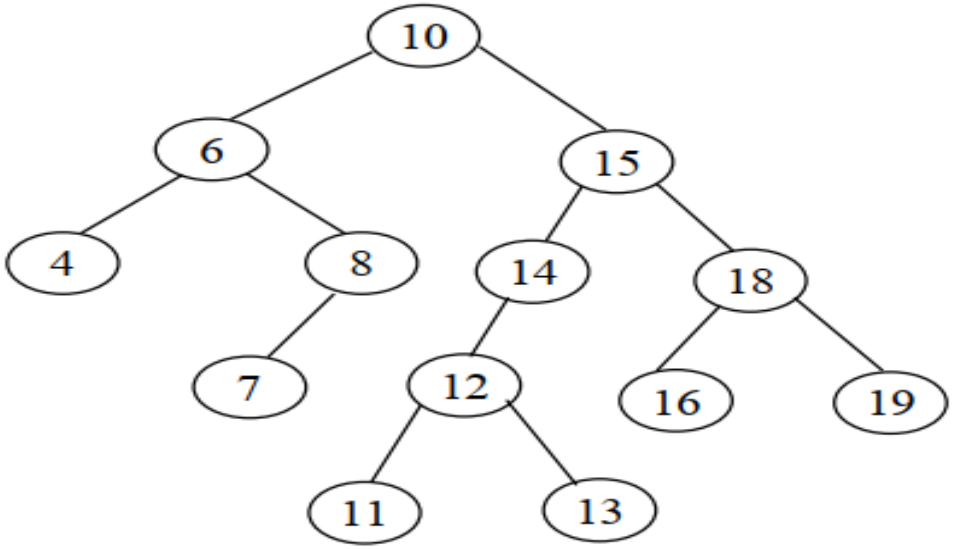
Trees

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments
- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.
- Subtree: tree consisting of a node and its descendants



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.



Traversing

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- In a postorder traversal, a node is visited after its descendants
- In an inorder traversal a node is visited after its left subtree and before its right subtree

1. Pre Order Traversal (root-left-right)
2. In order Traversal (Left-root-right)
3. Post Order Traversal (Left-right-root)

Pre Orders Traversal Recursively

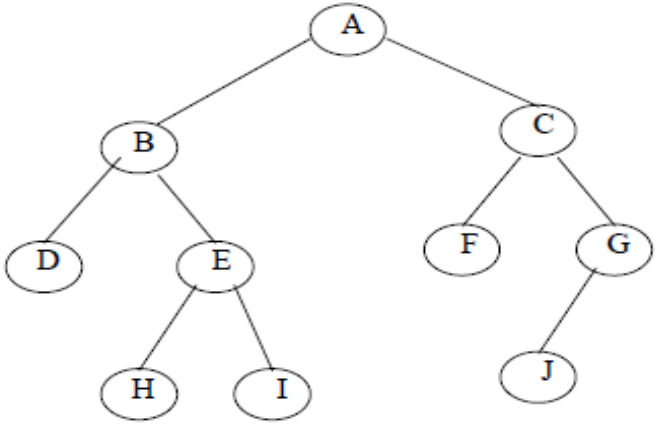
To traverse a **non-empty** binary tree in pre order following steps one to be processed

1. Visit the root node
2. Traverse the left sub tree in preorder
3. Traverse the right sub tree in preorder

That is, in preorder traversal, the root node is visited (or processed) first, before traveling through left and right sub trees recursively.

It can be implemented in C/C++ function as below:

```
void preorder (Node * Root)
{
If (Root != NULL)
{
cout <<Root → Info;
preorder(Root → L child);
preorder(Root → R child);
}
}
```



The preorder traversal of a binary tree in Figure is A, B, D, E, H, I, C, F, G, J.

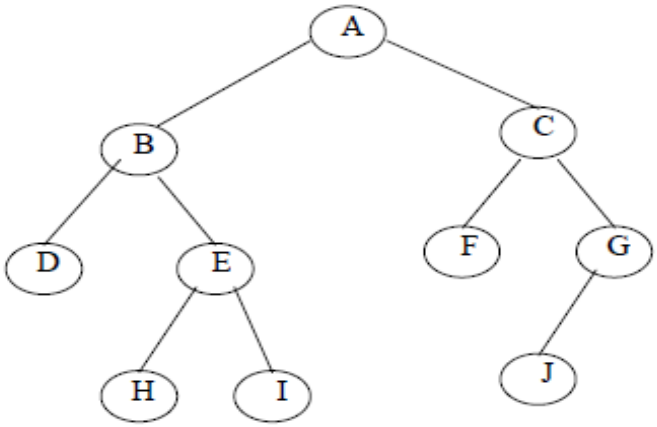
In Order Traversal Recursively

The in order traversal of a non-empty binary tree is defined as follows:

1. Traverse the left sub tree in order
2. Visit the root node
3. Traverse the right sub tree in order

In order traversal, the left sub tree is traversed recursively, before visiting the root. After visiting the root the right sub tree is traversed recursively, in order fashion. The procedure for an in order traversal is given below:

```
void inorder (NODE *Root)
{
If (Root != NULL)
{
inorder(Root → L child);
cout <<Root → info;
inorder(Root → R child);
}
}
```



The in order traversal of a binary tree is D, B, H, E, I, A, F, C, J, G.

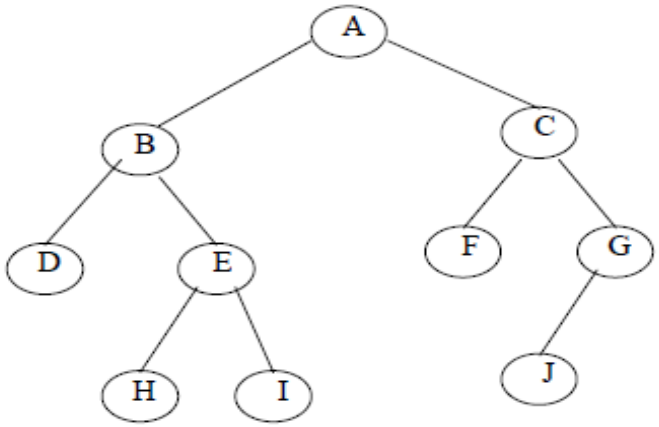
Post Order Traversal Recursively

The post order traversal of a non-empty binary tree can be defined as:

1. Traverse the left sub tree in post order
2. Traverse the right sub tree in post order
3. Visit the root node

In Post Order traversal, the left and right sub tree(s) are recursively processed before visiting the root.

```
void postorder (NODE *Root)
{
If (Root != NULL)
{
postorder(Root → Lchild);
postorder(Root → Rchild);
cout<<Root à info;
}
}
```



The post order traversal of a binary tree is D, H, I, E, B, F, J, G, C, A

Heap Sort

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

Algorithm:

1. Construct a binary tree 

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


3. Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

```
Heapsort (A as array)
    BuildMaxHeap(A)
    for i = n to 1
        swap (A[1], A[i])
        n = n - 1
        Heapify (A, 1)

BuildMaxHeap (A as array)
    n = elements_in(A)
    for i = floor (n/2) to 1
        Heapify (A,i)
```

```
Heapify (A as array, i as int)
    left = 2i
    right = 2i+1

    if (left <= n) and (A[left] > A[i])
        max = left
    else
        max = i

    if (right<=n) and (A[right] > A[max])
        max = right

    if (max != i)
        swap (A[i], A[max])
        Heapify (A, max)
```

O(nlogn)

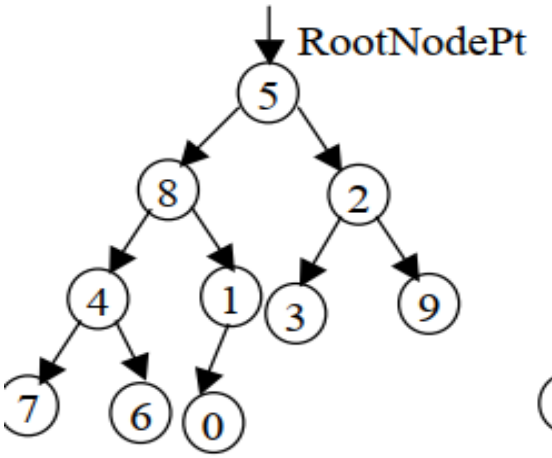
build-max-heap : O(n)

heapify : O(logn), called n–1 times

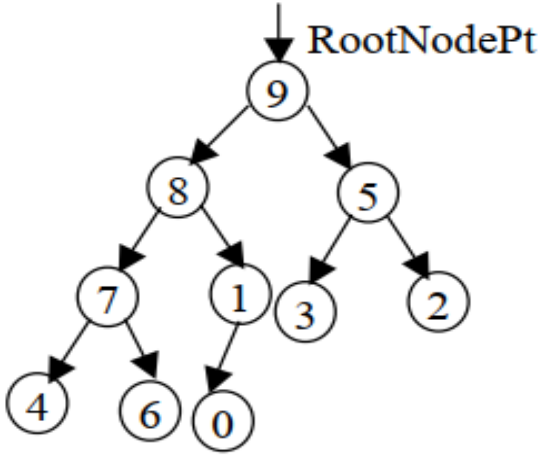
Example: Sort the following list using heap sort algorithm.

5	8	2	4	1	3	9	7	6	0

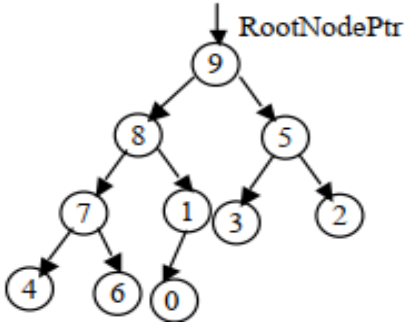
Construct the initial binary

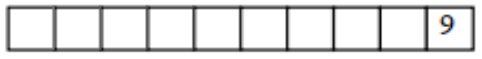


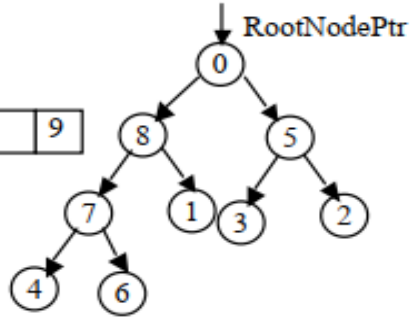
Construct the heap tree

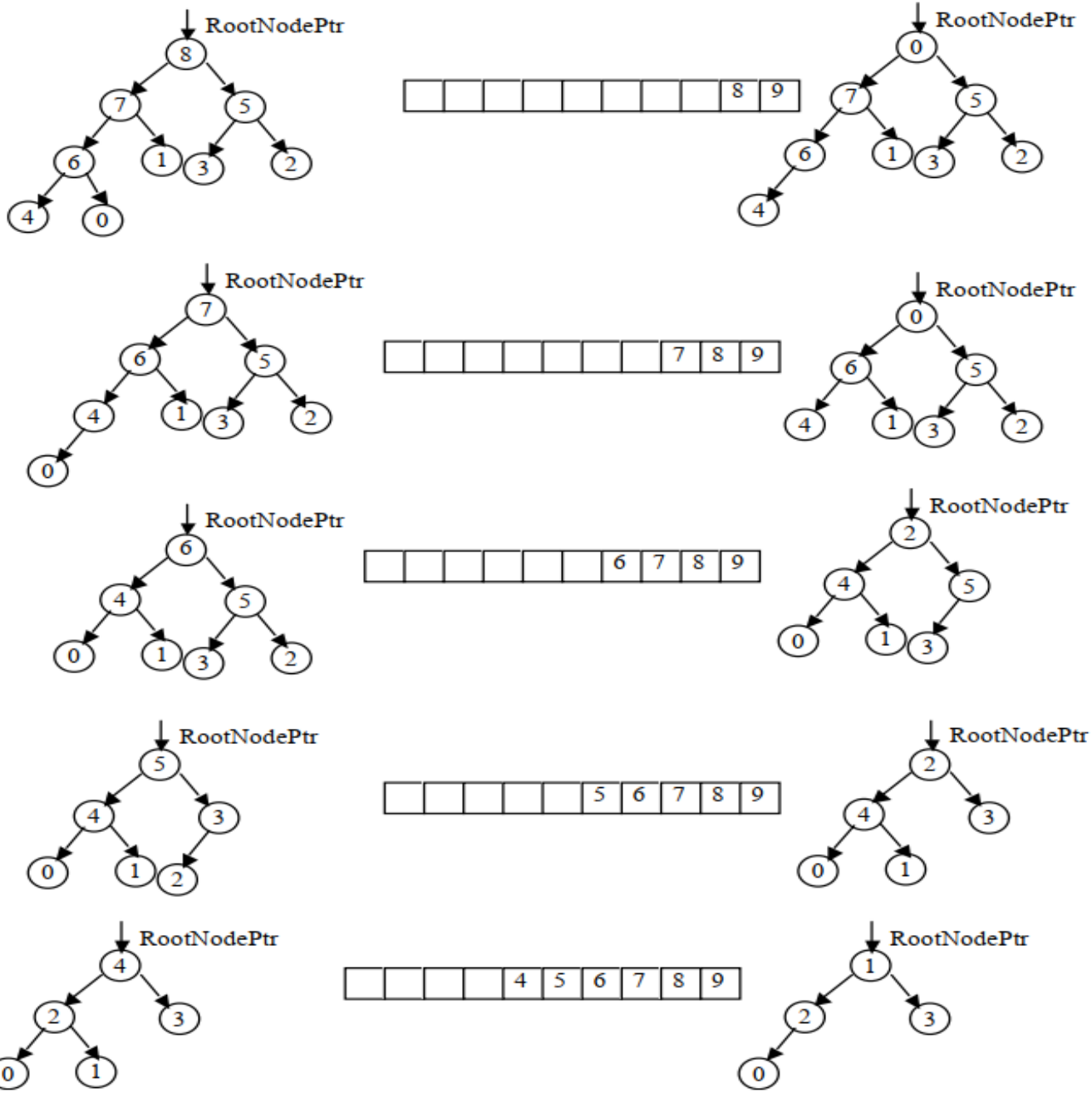


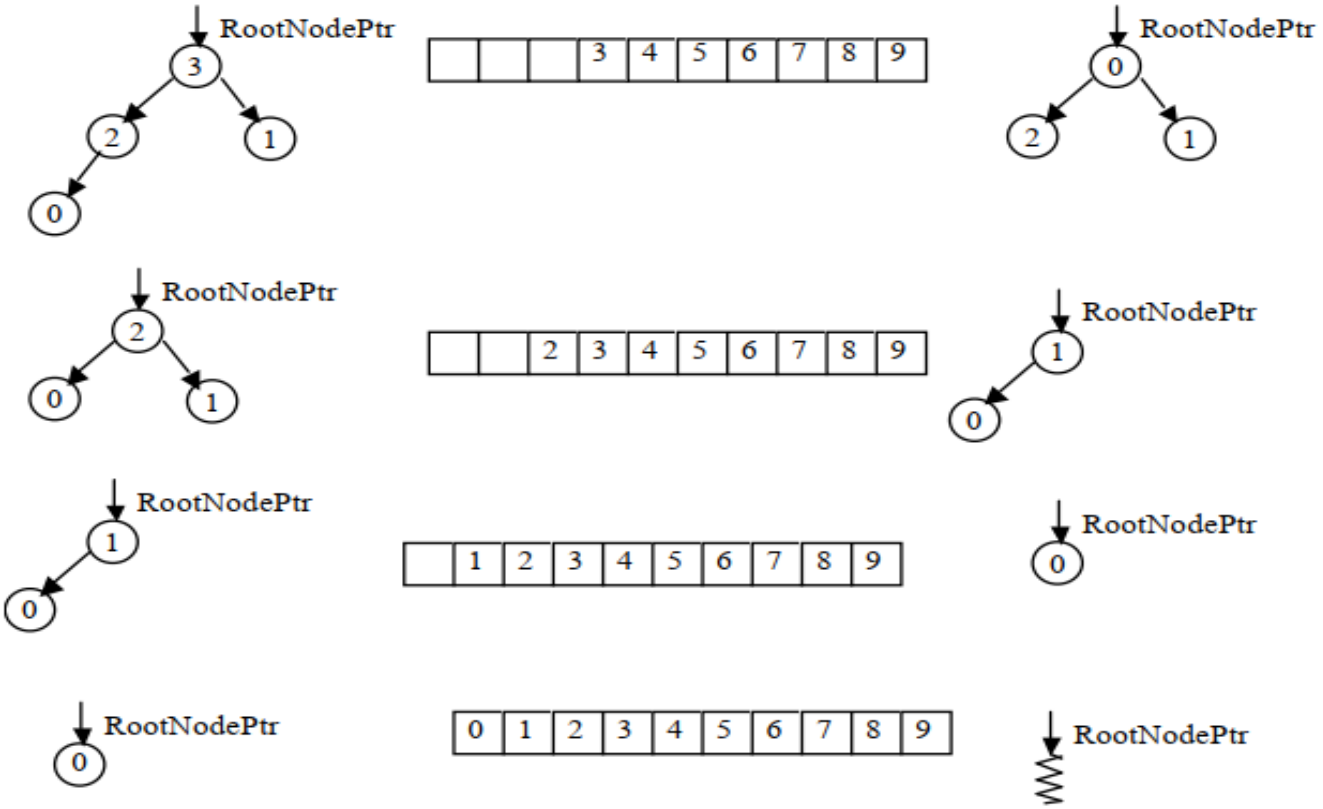
Swap the root node with the lowest, right most node and delete the lowest, right most value; insert the deleted value in the array in its proper position; adjust the heap tree; and repeat this process until the tree is empty.








RootNodePtr
8
7
5
6
1
3
2
4
0
8
9
RootNodePtr
0
7
5
6
1
3
2
4
RootNodePtr
7
6
5
4
1
3
2
0
7
8
9
RootNodePtr
0
6
5
4
1
3
2
RootNodePtr
6
4
5
0
1
3
2
6
7
8
9
RootNodePtr
2
4
5
0
1
3
RootNodePtr
5
4
3
0
1
2
5
6
7
8
9
RootNodePtr
2
4
3
0
1
RootNodePtr
4
2
3
0
1
4
5
6
7
8
9
RootNodePtr
1
2
3
0



Hashing

A hash table is a data structure that stores data and allows insertions, lookups, and deletions to be performed in constant, O(1), time. Hash tables provide a means for rapid searching for a record with a given key value and adapts well to the insertion and deletion of records. It is a technique used for performing insertions, deletions and finds in constant time. The aim of hashing is to map an extremely large key space onto a reasonable small range of integers such that it is unlikely that two keys are mapped onto the same integer.

- **A hash table**. This is a fixed size table that stores data of a given type.
- **A hash function**: This is a function that converts a piece of data into an integer. Sometimes we call this integer a **hash value**. The integer should be at least as big as the hash table. When we store a value in a hash table, we compute its hash value with the hash function, take that value modulo the hash table size, and that's where we store/retrieve the data. (Number of data elements in the hash table) / (Size of the hash table)
- **A collision resolution strategy**: There are times when two pieces of data have hash values that, when taken modulo the hash table size, yield the same value. That is called a *collision*. You need to handle collisions. We will detail four collision resolution strategies: Separate chaining, linear probing, quadratic probing, and double hashing.

Example: Suppose we have a set of strings {"abc", "def", "ghi"} that we'd like to store in a table. Our objective here is to find or update them quickly from a table, actually in O(1). We are not concerned about ordering them or maintaining any order at all. Let us think of a simple schema to do this. Suppose we assign "a" = 1, "b" =2, … etc to all alphabetical characters. We can then simply compute a number for each of the strings by using the sum of the characters as follows. "abc" = 1 + 2 + 3=6, "def" = 4 + 5 + 6=15, "ghi" = 7 + 8 + 9=24. If we assume that we have a table of size 5 to store these strings, we can compute the location of the string by taking the sum mod 5. So, we will then store "abc" in 6 mod 5 = 1, "def" in 15 mod 5 = 0, and "ghi" in 24 mod 5 = 4 in locations 1, 0 and 4 as follows.

0	1	2	3	4
def	abc			ghi