Chapter 1

Introduction to Data Structures and Algorithms

Data Types

- A data type is a collection of values and a set of operations on those values
- Most programming languages have data types such as integers, floating point numbers, characters etc.
- All these data types define set of values and operations on those values
- For example, the **int** data type in C program takes only integer values and we can apply addition, subtraction, multiplication, modulus etc. operations with them; The **float** data type in C takes floating point values and addition, subtraction, multiplication operations are allowed whereas modulus operation is not allowed

Abstract Data Type (ADT)

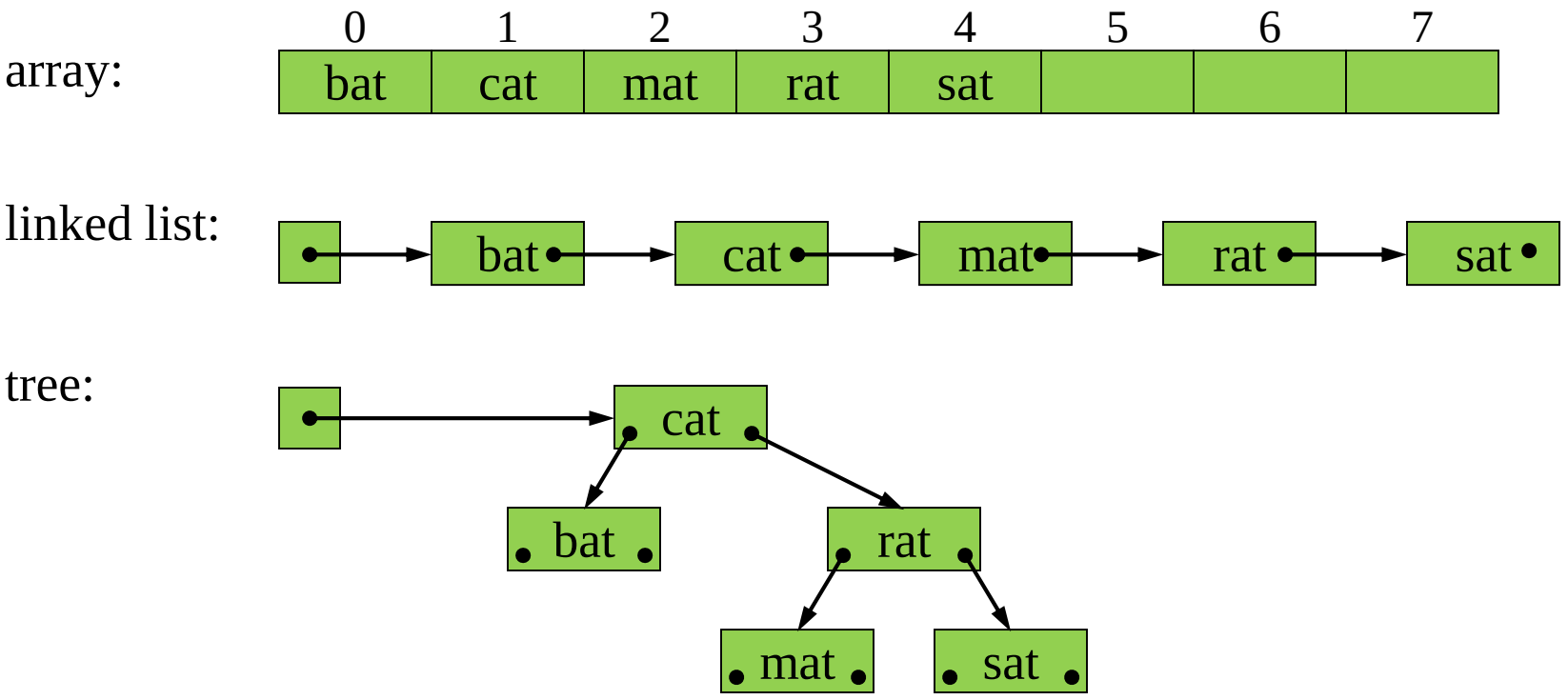
- Abstract Data Type (ADT) is special kind of datatype that hides inner structure and design of the data type from the user
- Most commonly used ADTs are **Stacks** and **Queues**
- There are multiple ways to implement same ADT; For example, Stack ADT can be implemented using arrays or linked lists
- If someone wants to use ADT in a program, he/she can simply use operations without knowing its implementation detail

Data Structures

- A data structure is a way to store and organize data in order to facilitate access and modifications; No single data structure works well for all purposes
- Data structures can be classified as **static** and **dynamic**; A **static data structure** is one whose capacity is fixed at creation (for example, array); A **dynamic data structure** is one whose capacity is variable, so it can expand or contract at any time (for example, linked list and tree)
- For each data structure we need algorithms for insertion, deletion, searching, sorting etc.

Data Structures

- Possible data structures to represent the set of words {bat, cat, mat, rat, sat}:



Dynamic Memory Allocation

- Dynamic memory allocation is the process of allocating memory in a program dynamically at run time.
- C programming provides two basic functions **calloc()** and **malloc()** for this purpose.
- The name calloc stands for “contiguous allocation” and the name malloc stands for “memory allocation”.
- The function calloc() takes two arguments both of unsigned integral type. When we call this function, it allocates contiguous space in memory. This function allocates multiple blocks of requested memory and the space is initialized with all bits set to zero.

Dynamic Memory Allocation

/* Program to read n numbers and display their sum and average using dynamic memory allocation*/

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
        float *a,sum=0,avg;
        int n,i;
        printf("How many numbers?");
        scanf("%d",&n);
        a=(float*)calloc(n,sizeof(float));
        printf("Enter numbers:\n");
        for(i=0;i<n;i++)
        {
                scanf("%f",(a+i));
                sum=sum+*(a+i);
        }
        avg=sum/n;
        printf("Sum=%f\n",sum);
        printf("Average=%f",avg);
        free(a);
}
```

Dynamic Memory Allocation

- The function malloc() allocates single block of requested memory and does not initialize the space in memory. In a large program, malloc() may take less time. We can use malloc as **a=(float*)malloc(n*sizeof(float));**
- Space that has been dynamically allocated with either calloc() or malloc() does not destroyed automatically. The programmer must use **free()** explicitely to destroy it. For example, free(a);
- **Note:** If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by **realloc()** function. In short, it changes the memory size. For example, **a = (float*)realloc(a, new-size);**

Introduction to Algorithms

- Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output
- An algorithm is thus a sequence of computational steps that transform the input into the output
- We can also view an algorithm as a tool for solving a well-specified computational problem
- Examples: sorting algorithms, searching algorithms etc.

Efficiency of Algorithms

- Algorithms devised to solve the same problem often differ dramatically in their efficiency
- Given several algorithms to solve the same problem, which algorithm is “best”?
- Given an algorithm, is it **feasible** to use it at all? In other words, is it efficient enough to be usable in practice?
- **Time efficiency:** How much **time** does the algorithm require? **[significant operations]**
- **Space efficiency:** How much **space** (memory) does the algorithm require? **[Extra space]**
- In general, both time and space requirements depend on the algorithm’s input (typically the “size” of the input)

Asymptotic Notations

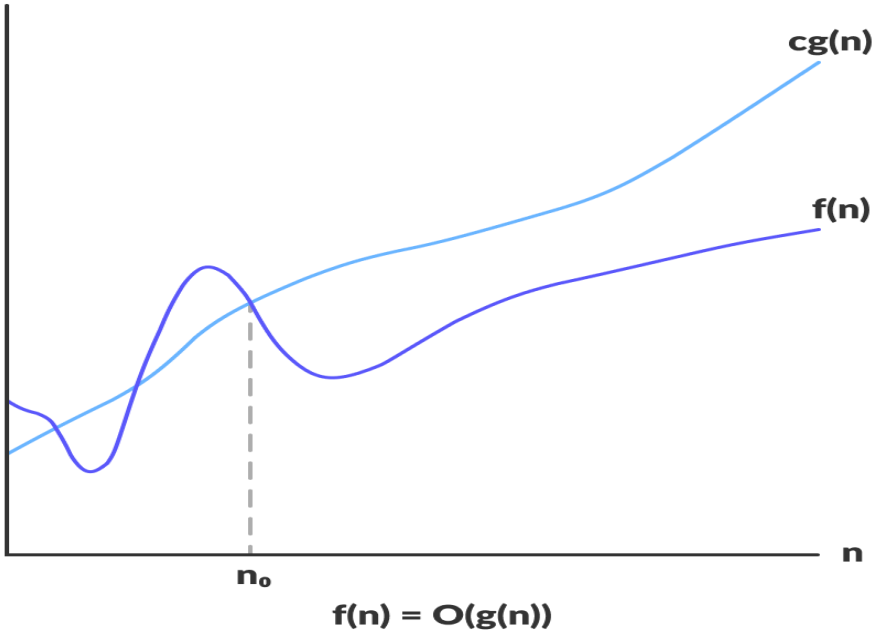
- The efficiency of algorithm is measured with the help of asymptotic notations.
- The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.
- Three most common asymptotic notations: Big-oh (O), omega (Ω), and theta (θ).
- **Big-oh (O):**
  - It is the formal method of expressing the upper bound of an algorithm's running time.
  - It is the measure of the longest amount of time.

Asymptotic Notations

- The function **f (n) = O (g (n))** [read as "f of n is big-oh of g of n"] if and only if there exist positive constant c and n_0 such that

$$0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0$$

- For example, 3n + 2 = O(n) as 3n+2 ≤ 4n **for** all n ≥ 2
- Hence, the complexity of **f(n)** can be represented as O (g (n))

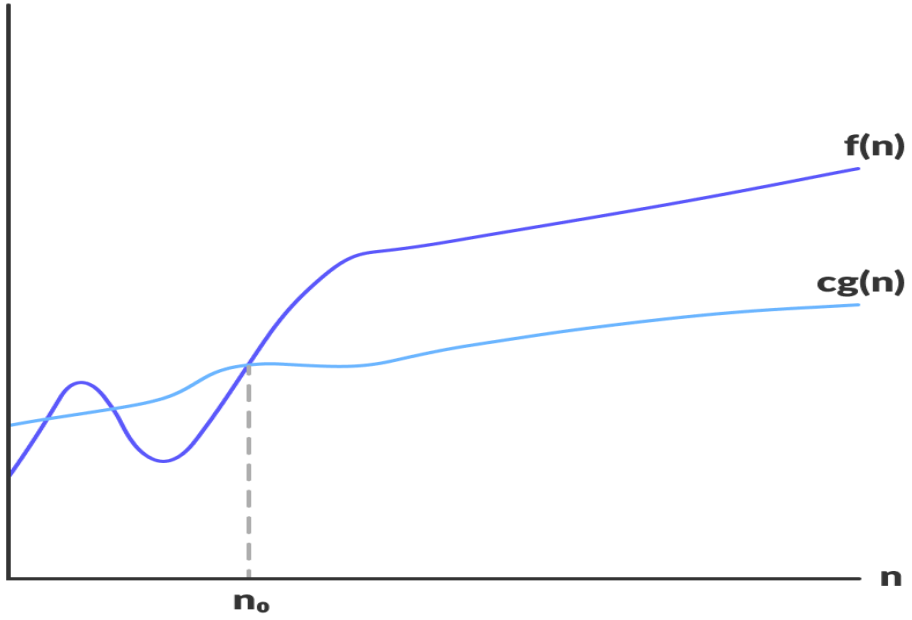


Asymptotic Notations

- **Omega (Ω):**
  - The function f (n) = Ω (g (n)) [read as "f of n is omega of g of n"] if and only if there exists positive constant c and n_0 such that

    $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$
  - It gives lower bound of an algorithms running time
  - For example, $8n^2 + 2n - 3 = \Omega(n^2)$ as $8n^2 + 2n - 3 \geq 7n^2$ for all $n \geq 1$
  - Hence, the complexity of **f(n)** can be represented as Ω (g (n))

Asymptotic Notations



$f(n) = \Omega(g(n))$

Asymptotic Notations

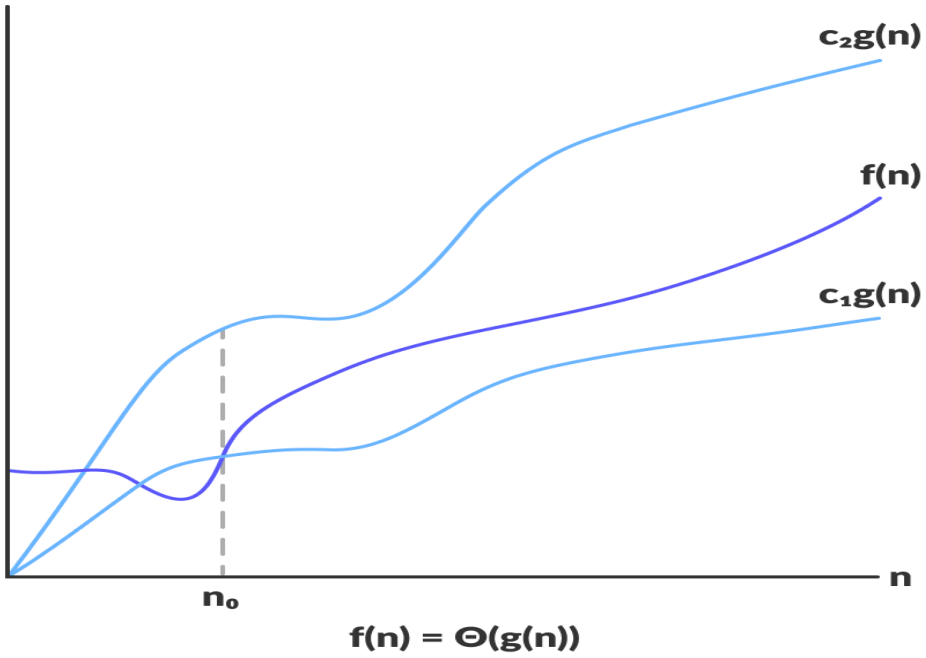
- **Theta (θ):**
  - The function f (n) = θ (g (n)) [read as "f of n is theta of g of n"] if and only if there exists positive constant c and n_0 such that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0$$

  - Gives both upper and lower bound of algorithms running time
  - The Theta Notation is more precise than both the big-oh and Omega notation. The function f (n) = θ (g (n)) if g(n) is both an upper and lower bound.
  - For example, 3n + 2 = θ(n) as 3n + 2 ≥ 3n and 3n+2 ≤ 4n, for c1 = 3, c2 = 4, and n_0 = 2

Asymptotic Notations

- The function f (n) = θ (g (n)) [read as "f of n is theta of g of n"] if and only if there exists positive constant c and n_0 such that



Finding Big oh (O) Notation

- For many interesting algorithms, the exact number of operations is too difficult to analyze mathematically.
- To simplify the analysis:
  - identify the fastest-growing term
  - neglect slower-growing terms
  - neglect the constant factor in the fastest-growing term.
- The resulting formula is the algorithm's **time complexity**. It focuses on the **growth rate** of the algorithm's time requirement.
- Similarly for **space complexity**.

Finding Big oh (O) Notation

- For example, suppose maximum number of operations of an algorithm is

  $f(n) = 5n^2 – 3n + 2$

  simplify to $5n^2$

  then to n^2

  Time complexity is **of order n^2**. This is written ***O*(n^2)**.

Finding Big oh (O) Notation

- Common time complexities:

$O(1)$	**constant** time	(feasible)
$O(\log n)$	**logarithmic** time	(feasible)
$O(n)$	**linear** time	(feasible)
$O(n \log n)$	**log linear** time	(feasible)
$O(n^2)$	**quadratic** time	(sometimes feasible)
$O(n^3)$	**cubic** time	(sometimes feasible)
$O(2^n)$	**exponential** time	(rarely feasible)

Finding Big oh (O) Notation

- Comparison of growth rate:

1	1	1	1	1
log n	3.3	4.3	4.9	5.3
n	10	20	30	40
n log n	33	86	147	213
n^2	100	400	900	1,600
n^3	1,000	8,000	27,000	64,000
2^n	1,024	1.0 million	1.1 billion	1.1 trillion

Finding Big oh (O) Notation

- Graphically: