CHRIST
(DEEMED TO BE UNIVERSITY)
BANGALORE, INDIA

MCA 371
DATA STRUCTURES
Unit 1

What do analytics, machine learning, data science, and big data systems have in common?

What is the major common component for astronomy, biology, neuroscience, and all other data driven and computational sciences?

What do analytics, machine learning, data science, and big data systems have in common?

What is the major common component for astronomy, biology, neuroscience, and all other data driven and computational sciences?

Data Structures

There Is No Perfect Data Structure Design

Unit 1

ELEMENTARY DATA STRUCTURES

Introduction to Pseudo code - Overview of Time & Space Complexity – Recursion – Abstract Data Type - Array - Stack - Queue - Linked lists - Traversing - Searching - Insertion - Deletion – Circular Linked list - Two-way Lists (Doubly) - Linked List Implementation of Stack and Queue - Application of stacks and Queues.

Unit-2

SORTING AND SEARCHING

Bubble Sort – Insertion – Selection – Quick – Merge – Linear Search – Binary search – Hashing – Chaining – Collision Resolution – Open Addressing – String Matching Algorithms: Naive, KMP

Unit-3

GRAPHS& TREES

- Representation of Graphs - Operations on Graphs - Depth First and Breadth First Search - Topological Sort - Minimum Spanning Tree Algorithms - Binary Tree - Traversing Binary Trees - Binary Heap - Priority Queue - Heap sort.

Unit-4

SEARCH TREES

Binary Search Trees - Searching, Inserting and deleting in Binary Search Trees - AVL Trees – AVL -Balance Factor, Balancing Trees, AVL node structure, AVL Tree Rotate Algorithms

Unit-5

ADVANCED DATA STRUCTURES

B Trees – Operations on B Trees - B+ Trees - Red-Black Trees - Properties of Red-black Trees - Rotations - Insertion - Deletion operations

Program?

- A Set of Instructions
- Data Structures + Algorithms
- Data Structure = A Container stores Data
- Algorithm = Logic + Control

Analysis of Algorithm

Space Complexity

The space complexity of a program is the amount of memory it needs to run to completion

Time Complexity

The time complexity of a program is the amount of computer time that it needs to run to completion

Space Complexity

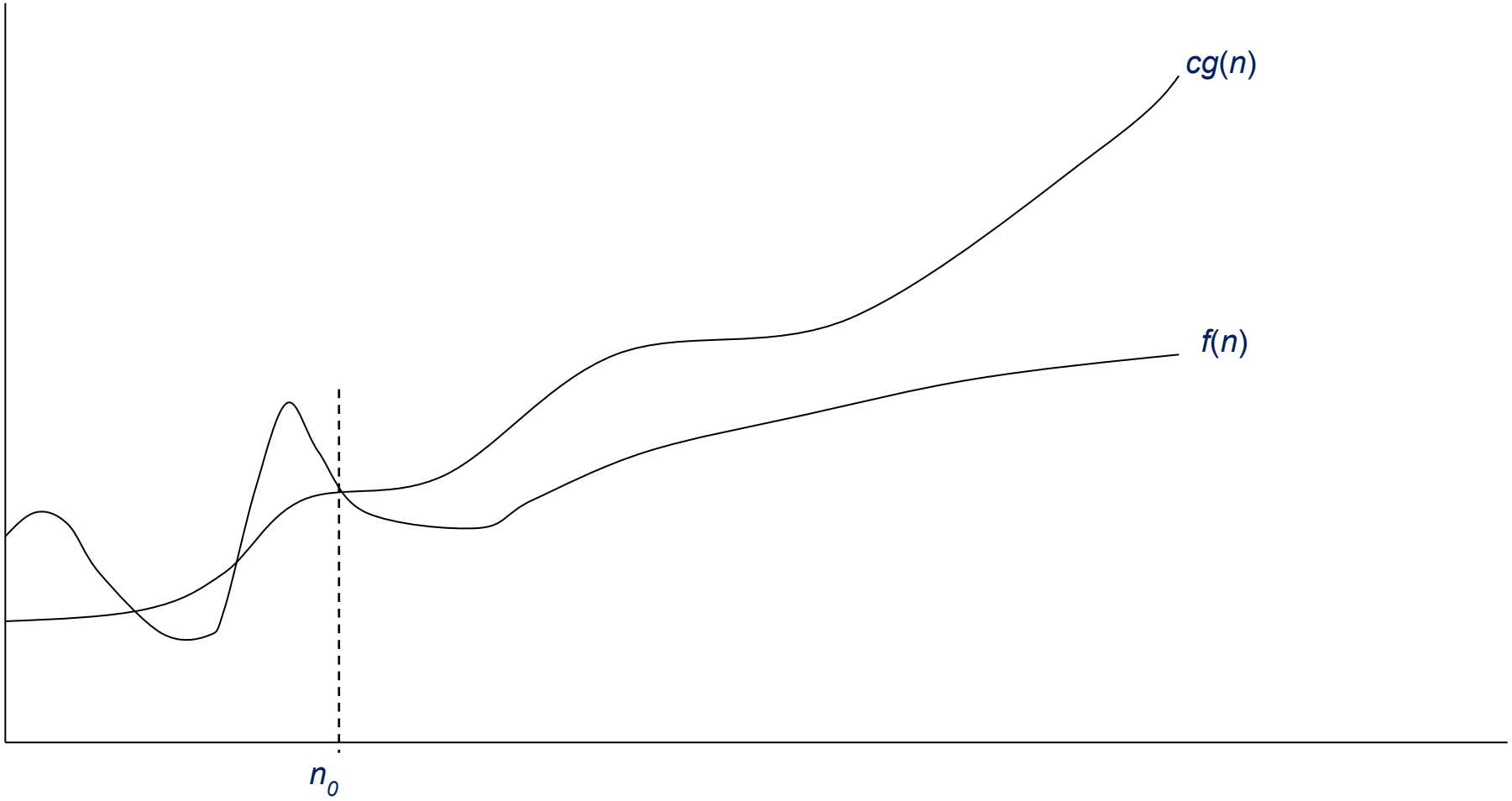
- Fixed space requirement
  - Instruction space
  - Simple variables
  - Fixed size structure variables
  - Constants
- Variable space requirement
  - Structured variables whose size depends on instance
  - Recursion
  - Number , size, values of the inputs and outputs

Total space requirement of any program $S(P)= c+Sp(l)$

Statement	s/e	Frequency	Total steps
float sum(float list[], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for (i = 0; i < n; i++)	1	$n+1$	$n+1$
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			$2n+3$

Statement	s/e	Frequency	Total steps
float rsum(float list[], int n)	0	0	0
{	0	0	0
if (n)	1	$n+1$	$n+1$
return rsum(list,n−1) + list[n−1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			$2n+2$

Visualization of $O(g(n))$



Big Oh Notation

The function $f(n)= O(g(n))$ iff there exist positive constants c and n_0 such that

$f(n) <= c * g(n)$ for all n, $n >= n_0$

Read as " f of n is big oh of g of n"

iff : if and only if

Examples

1)The function 2n+3 = O(n)

2n+3 < 2n + n for all n>=3

2n+ 3 < 3n for all n>=3

f(n)=O(g(n))

f(n) <= c * g(n)
f(n)<= 3 * n

O(n)

2) $100n+6=O(n)$

$100n+6 < 101\ n$ for all $n>=6$

$=O(n)$

3) $10\ n^2 + 4n + 2 = O(n^2)$

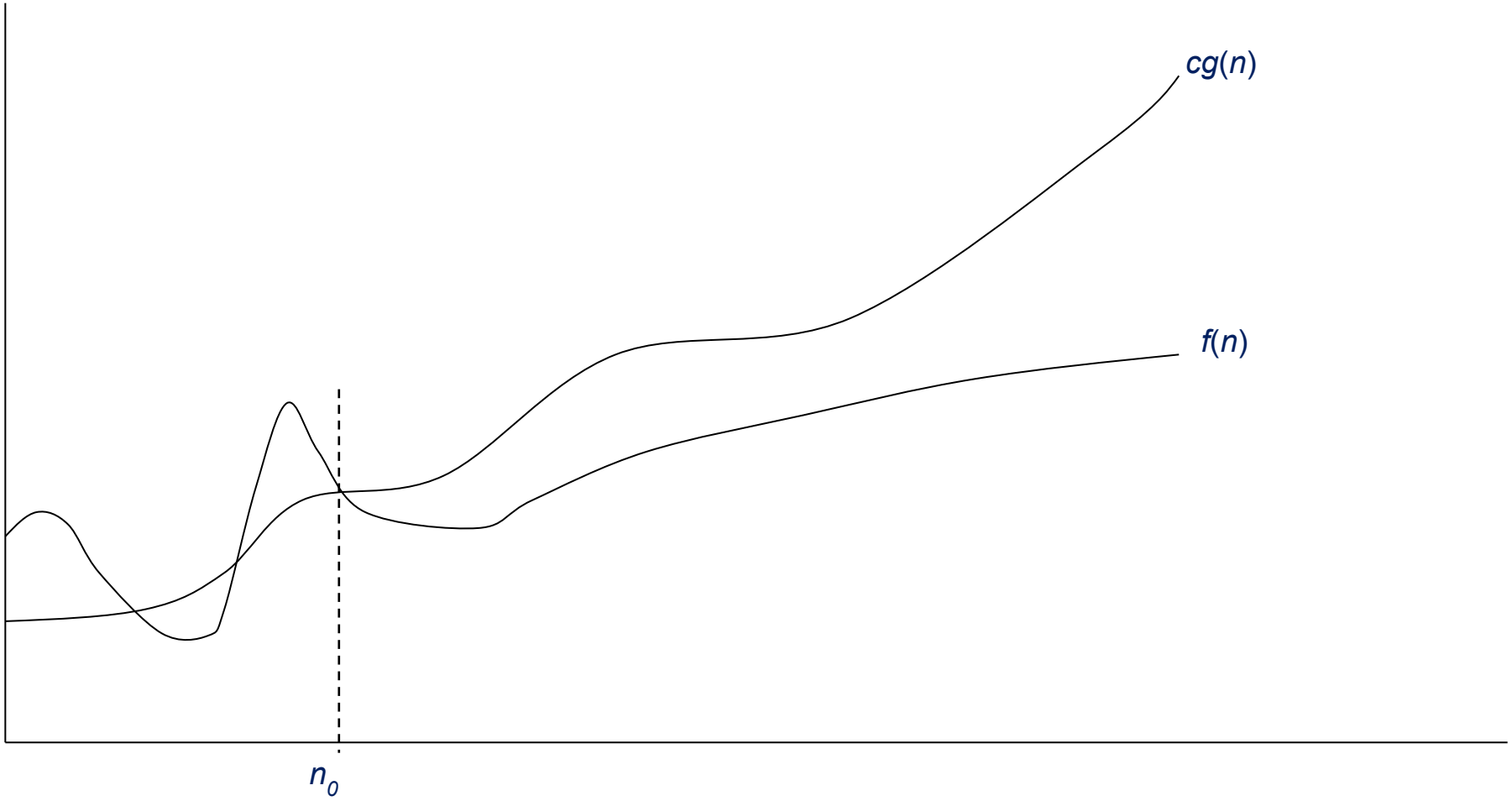
$10\ n^2 + 4n + 2 < 11\ n^2$ for all $n>= 5$

$=O(n^2)$

1	constant
log *n*	logarithmic
n	linear
n log *n*	*n*-log-*n*
n^2	quadratic
n^3	cubic
2^n	exponential
n!	factorial

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

Visualization of $O(g(n))$



Growth of Functions

n	1	lgn	n	nlgn	n^2	n^3	2^n
1	1	0.00	1	0	1	1	2
10	1	3.32	10	33	100	1,000	1024
100	1	6.64	100	664	10,000	1,000,000	1.2×10^{30}
1000	1	9.97	1000	9970	1,000,000	10^9	1.1×10^{301}

n	log n
16	4
64	6
256	8
1024	10
16384	14
262144	18
524288	19
1048576	20
1073741824	30

Time Complexity; O(1)

```
for (int i = 1; i<= c; i++)
{
      // some O(1) expressions
}
```

A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

Time Complexity: O(n)

```
for (int i = 1; i<= n; i += c)
{
      // some O(1) expressions
 }

for (int i = n; i> 0; i -= c)
{
      // some O(1) expressions
 }
```

Time Complexity of a loop is considered as **O(n)** if the loop variables is incremented / decremented by a constant amount

Time Complexity: $O(n^2)$

```
for (int i = 1; i<=n; i += c)
{
    for (int j = 1; j <=n; j += c)
    {
        // some O(1) expressions
     }
 }
```

Time complexity of nested loops is equal to the number of times the innermost statement is executed. **$O(n^2)$**

Time Complexity: O(Log n)

```
for (int i = 1; i<=n; i *= c)
{
      // some O(1) expressions
}
for (int i = n; i> 0; i /= c)
{
      // some O(1) expressions
}
```

Time Complexity of a loop is considered as **O(Log n)** if the loop variables is divided / multiplied by a constant amount

Time Complexity : O(Log Log n)

```
for (int i = 2; i<=n; i = pow(i, c))
{
      // some O(1) expressions
}

for (int i = n; i> 0; i = fun(i))
{
      // some O(1) expressions
}
```

Time Complexity of a loop is considered as **O(Log Log n)** if the loop variables is reduced / increased exponentially by a constant amount

Asymptotic Complexity

Asymptotic complexity is a way of expressing the main component of the cost of an algorithm, using idealized units of computational work

Asymptotic Notation

Asymptotic Notations are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases. This is also known as an algorithm's growth rate

Asymptotic Notation

Describes the behavior of the time or space complexity for large instance characteristic

- **Big Oh (O)** notation provides an **upper bound** for the function *f*
- **Omega (Ω)** notation provides a **lower-bound**
- **Theta (Q)** notation is used when an algorithm can be **bounded both from above and below** by the same function

Omega Notation

The function $f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that

$f(n) >= c * g(n)$ for all n, $n >= n_0$

Read as " *f* of n is Omega of *g* of n"
iff : if and only if

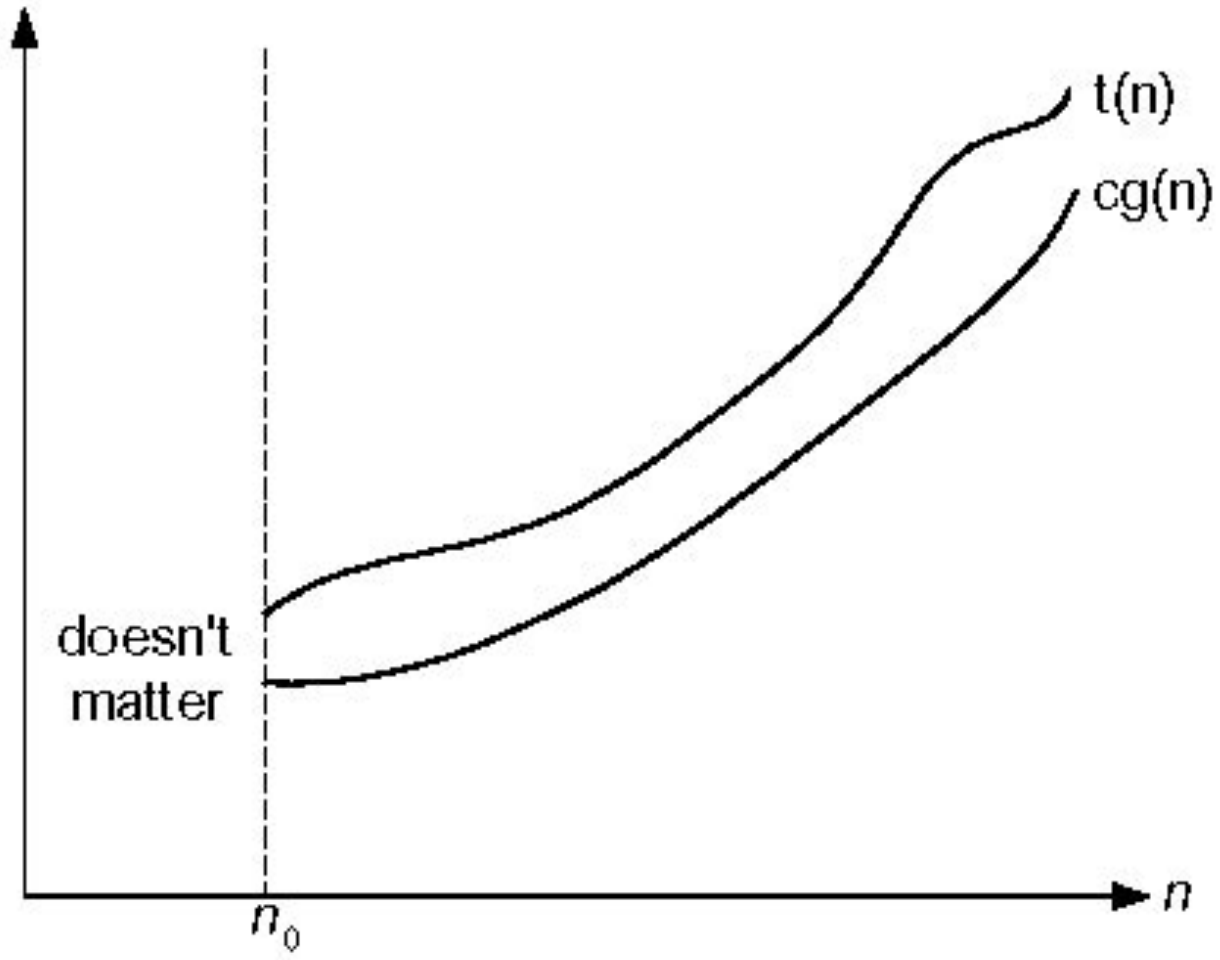


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

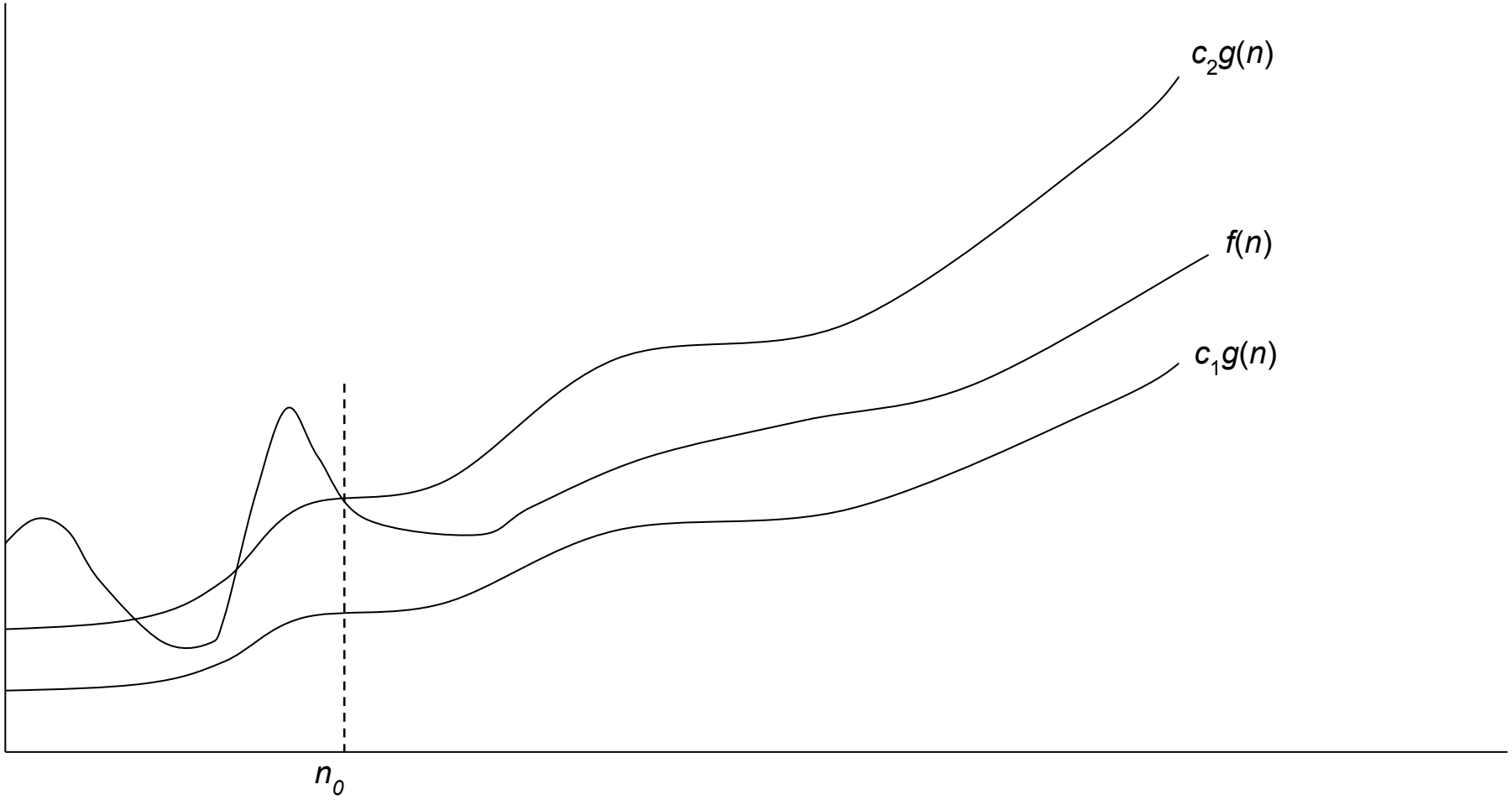
Theta Notation

The function $f(n)= \Theta(g(n))$ iff there exist positive constants c_1, c_2 and n_0 such that

$c_1\ g(n) <= f(n) <= c_2\ g(n)$ for all n, $n >= n_0$

Read as " *f* of n is Theta of *g* of n"
iff : if and only if

Visualization of $\Theta(g(n))$



Linear Search

```
for (i=1; i<=n; i++) n+1
{
        if (arr[i] == x)n
    {
        printf("%d%d", x, i);
    }
}
```

2n+1 comparisons
O(n)

Sentinel Linear Search

last=arr[n-1] 8,9,15,4,3 x=5 last=3 8 9 15 4 3

```
arr[n - 1] = x;
    int i = 0;

    while (arr[i] != x)
        i++;
     // Put the last element back
    arr[n - 1] = last;

    if ((i < n - 1) || (x == arr[n - 1]))
       printf(“%d%d”, x, i);
    else
        printf("Not found“);
```

N+2 comparisons

Compute the time complexity

```
float sum(float list[], int n)
{
    float tempsum=0;
    int i;
    for(int i=0;i<n;i++)
    {
        tempsum=tempsum+list[i];
    }
    return tempsum;
}
```

```
float sum(float list[], int n)
{
    float tempsum=0; count++
    int i;
    for(int i=0;i<n;i++)
    {
        count++;
        tempsum=tempsum+list[i];
        count++
    }
    count++;
    count++;
    return tempsum;
}
```

Compute the time complexity

```
for( int i=0;i<n;i++)
{
    for(int j=i;j<n;j++)
    {
        //statement
    }
}
```

$$1 + 2 + 3 + 4 + \ldots + n = \frac{n \times (n+1)}{2}.$$

```
for( int i=0;i<n;i++)
{
    for(int j=i;j<n;j++)
    {
        //statement
        count++;
    }
    count++;
}
count++;
```

Tutorial

Compute the complexity

Shift n items from one place to another . Distance is ‘x’

How many operations?

n pick-ups, n forward moves, n drops and n reverse moves □ 4 n operations

4n operations = c. n = O(c. n) = O(n)

Store manager gives gifts to first 10 customers

- There are n customers in the queue.
- Manager brings one gift at a time

Complexity?

Store manager gives gifts to first 10 customers

- There are n customers in the queue.
- Manager brings one gift at a time.
- Time complexity = O(c. 10) = O(1)
- Manager will take exactly same time irrespective of the line length.

Pseudocode

- Pseudocode is an English-like representation of the algorithm logic.
- It is part English, part structured code
- English part provides a relaxed syntax
- Code part consists of an extended version of the basic algorithmic constructs—sequence, selection, and iteration.

Pseudocode Example

```
Algorithm deviation
   Pre    nothing
   Post   average and numbers with their deviation printed
1  loop (not end of file)
   1   read number into array
   2   add number to total
   3   increment count
2  end loop
3  set average to total / count
4  print average
5  loop (not end of array)
   1   set devFromAve to array element - average
   2   print array element and devFromAve
6  end loop
end deviation
```

Header

```
Algorithm search  (list, argument, location)
Search array for specific item and return index location.
   Pre    list contains data array to be searched
          argument contains data to be located in list
   Post   location contains matching index
          -or- undetermined if not found
   Return true if found, false if not found
```

Algorithm

```
S:stack
while(more tokens)
  x<=next token
    if(x == operand)
      print x
    else
      while(precedence(x)<=precedence(top(s)))
        print(pop(s))
    push(s,x)
while(! empty (s))
  print(pop(s))
```

EXAMPLE:

A+(B*C-(D/E-F)*G)*H

Stack	Input	Output
Empty	A+(B*C-(D/E-F)*G)*H	-
Empty	+(B*C-(D/E-F)*G)*H	A
+	(B*C-(D/E-F)*G)*H	A
+(	B*C-(D/E-F)*G)*H	A
+(	*C-(D/E-F)*G)*H	AB
+(*	C-(D/E-F)*G)*H	AB

Data Structure

- To store data
- A data structure is a specialized format for organizing, processing, retrieving and storing data.
- A data structure is the way you define a certain object in a program language and is a layout for memory that represents some sort of data.
- In computer science, a **data structure** is a data organization, management, and storage format that enables efficient access and modification.
- A data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data

- Data structure is representation of the logical relationship existing between individual elements of data.

- In other words, a data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

Characteristics of Data Structures

- Linear or non-linear: This characteristic describes whether the data items are arranged in chronological sequence, such as with an array, or in an unordered sequence, such as with a graph.
- Homogeneous or non-homogeneous: This characteristic describes whether all data items in a given repository are of the same type or of various types.
- Static or dynamic: This characteristic describes how the data structures are compiled. Static data structures have fixed sizes, structures and memory locations at compile time. Dynamic data structures have sizes, structures and memory locations that can shrink or expand depending on the use.

Natural number Data Structure

structure NATNO

declare ZERO()->natno

ISZERO()→boolean

SUCC(natno)→ natno

ADD(natno,natno)→natno

EQ(natno,natno)→Boolean

For all x,y ∈ natno let

ISZERO(ZERO)::=TRUE

ISZERO(SUCC(X)::=FALSE

ADD(ZERO,Y)::=Y

ADD(SUCC(X),Y)::=SUCC(ADD(X,Y))

EQ(X,ZERO)::= IF ISZERO(X) THEN TRUE ELSE FALSE

EQ(ZERO,SUCC(Y))::=FALSE

EQ(SUCC(X),SUCC(Y)::= EQ(X,Y)

End

End NATNO

Data Structure

A data structure is a set of domains ‘D’ , a designated domain ‘d’ ∈ ‘D’, a set of functions ‘F’ and a set of axioms ‘A’. The triple (D,F,A) denotes the data structure ‘d’.

d= natno
D=(natno,Boolean)
F={ZERO,ISZERO,SUCC,ADD}
A={AXIOMS}

ARRAY

structure ARRAY(value, index)
declare CREATE()→ array
RETRIEVE(array,index)→value
STORE(array,index,value)→array
For all A ∈ array
i,j i ∈ ex
x ∈ lue
let
RETRIEVE(CREATE ,i)::= error
RETRIEVE(STORE(A,i,x),j)::= if EQUAL(i,j) then x else
RETRIEVE(A,J);
End
End ARRAY

```
structure ORDERED_LIST(atoms)
  declare MTLST( ) → list
          LEN(list) → integer
          RET(list,integer) → atom
          STO(list,integer,atom) → list
          INS(list,integer,atom) → list
          DEL(list,integer) → list;
  for all L ∈ list, i,j ∈ integer  a,b ∈ atom let
    LEN(MTLST) :: = 0; LEN(STO(L,i,a)) :: = 1 + LEN(L)
    RET(MTLST,j) :: = error
    RET(STO(L,i,a),j) :: =
      if i = j then a else RET(L,j)
    INS(MTLST,j,b) :: = STO(MTLST,j,b)
    INS(STO(L,i,a),j,b) :: =
      if i ≥ j then STO(INS(L,j,b), i + 1,a)
        else STO(INS(L,j,b),i,a)
    DEL(MTLST,j) :: = MTLST
    DEL(STO(L,i,a),j) :: =
      if i = j then DEL(L,j)
        else if i > j then STO(DEL(L,j),i − 1,a)
          else STO(DEL(L,j),i,a)
  end
end ORDERED_LIST
```

Abstract Data Type (ADT)

Mathematical description of an object and the set of operations on the object

Is a mathematical model for data types, where a data type is defined by its behavior from the point of view of a *user* of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects.

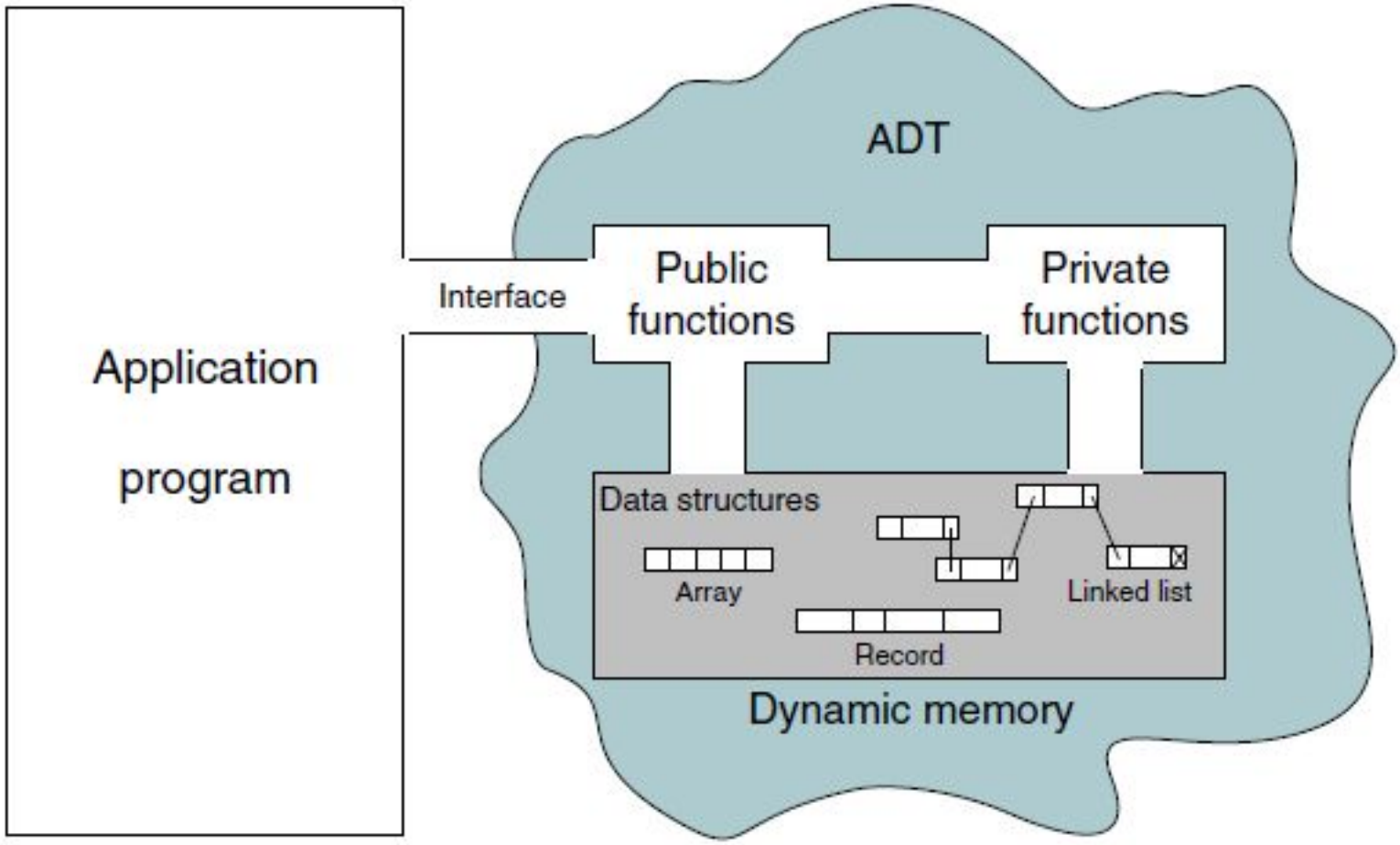
Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations.

Abstract Data Type(ADT)

An abstract data type (**ADT**) is basically a logical description or a specification of components of the data and the operations that are allowed, that is independent of the implementation.

Is a data type that is organized in such a way that the specification of the objects and specifications of the operations on the objects is separated from the representation of the objects and the implementation of the operations.

Model for an Abstract Data Type



Stack Data Structure

structure *STACK* (*item*)
1 **declare** *CREATE* ( ) *stack*
2 *ADD* (*item*, *stack*) *stack*
3 *DELETE* (*stack*) *stack*
4 *TOP* (*stack*) *item*
5 *ISEMTS* (*stack*) *boolean*;
6 **for all** *S stack*, *i item* **let**
7 *ISEMTS* (*CREATE*) :: = **true**
8 *ISEMTS* (*ADD* (*i*,*S*)) :: = **false**
9 *DELETE* (*CREATE*) :: = *error*
10 *DELETE* (*ADD* (*i*,*S*)) :: = *S*
11 *TOP*(*CREATE*) :: = *error*
12 *TOP*(*ADD*(*i*,*S*)) :: = *i*
13 **end**

Why data structure

A data structure helps you to understand the relationship of one data element with the other and organize it within the memory. Sometimes the organization might be simple and can be very clearly visioned.

Eg) List of names of months in a year –Linear Data Structure,
List of historical places in the world- Non-Linear Data Structure.

A data structure helps you to analyze the data, store it and organize it in a logical and mathematical manner

Structure

Array are collection of data of the same type

A structure is a collection of data items , where each item is identified as to its type and name.

A structure is a user-defined data type that allows to combine data items of different kinds

Syntax

```
struct [structure name]
   {
      member definition;
      member definition;
      ...
      member definition;
   };
```

```
struct address
{
char name[50];
char street[100];
char city[50];
char state[20];
int pin;
};
```

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   bookid;
} book;

struct Books Book1;
```

```
typedef struct {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   bookid;
} books

books Book1;
```

STACK

Is an ordered list in which insertion and deletion operations are performed through only one end.

A stack is a data structure that stores data in such a way that the last piece of data stored, is the first one retrieved

- also called last-in, first-out

Only access to the stack is the top element

LISTS

STACK

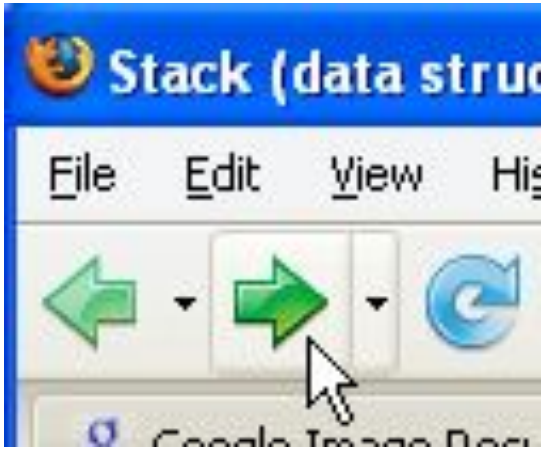
Stack

- Access is allowed only at one point of the structure, normally termed the top of the stack
  - access to the most recently added item only
- Operations are limited:
  - push (add item to stack)
  - pop (remove top item from stack)
  - top (get top item without removing it)
  - clear
  - isEmpty
- Described as a "Last In First Out" (LIFO) data structure

- Push
  - the operation to place a new item at the top of the stack
- Pop
  - the operation to remove the next item from the top of the stack

Use

- Consider entering keyboard text
  - Mistakes require use of backspace
    abcdd□ □efgg□



Stack ADT

objects: a finite ordered list with zero or more elements.

methods:

for all stack ∈ Stack, item ∈ element, max_stack_size ∈ positive integer

Stack createS(max_stack_size) ::=

create an empty stack whose maximum size is max_stack_size

Boolean isFull(stack, max_stack_size) ::=

if (number of elements in stack == max_stack_size)

return TRUE

else return FALSE

Stack push(stack, item) ::=

if (IsFull(stack)) stack_full

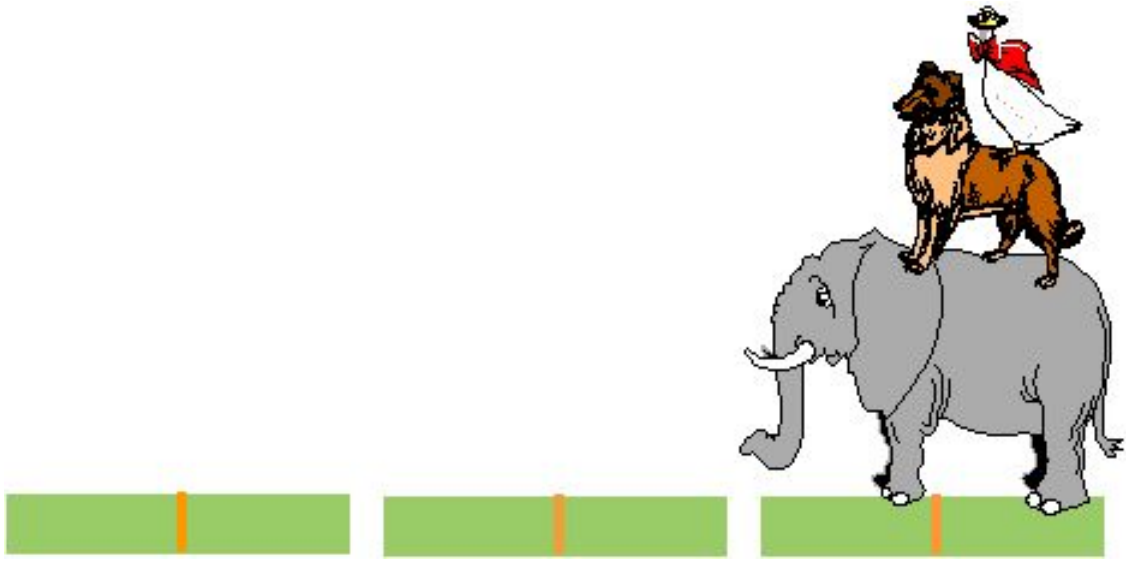
else insert item into top of stack and return

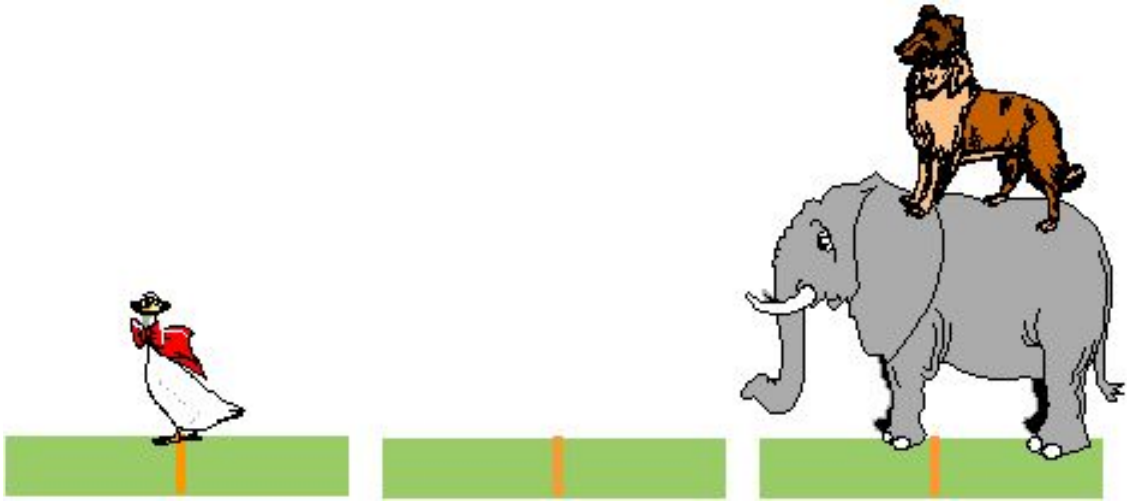
```
Boolean isEmpty(stack) ::=
                    if(stack == CreateS(max_stack_size))
                    return TRUE
                    else return FALSE
Element pop(stack) ::=
                    if(IsEmpty(stack)) return Stack_Empty
                    else remove and return the item on the
            top of the stack.
```

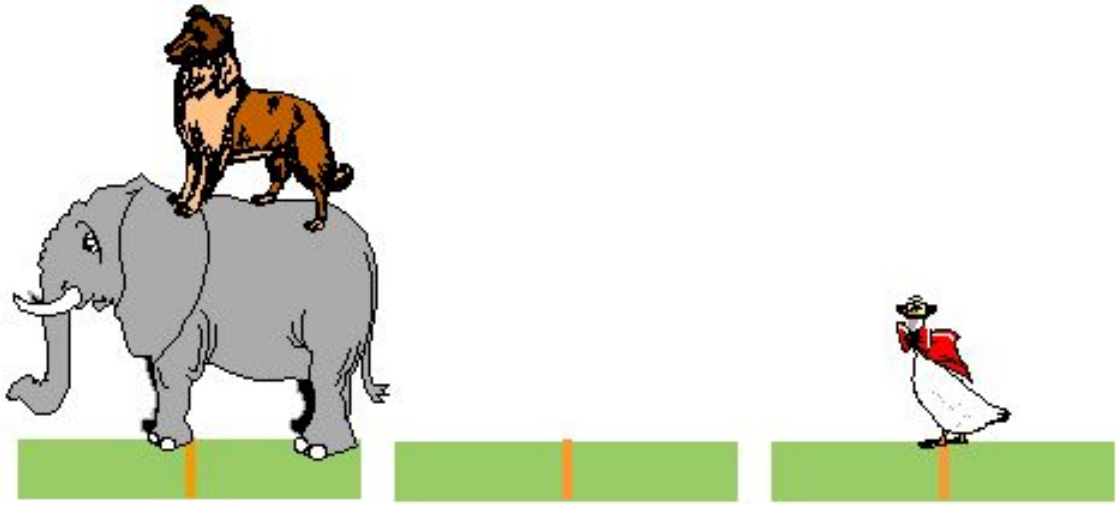
The Towers of Hanoi : A Stack-based Application

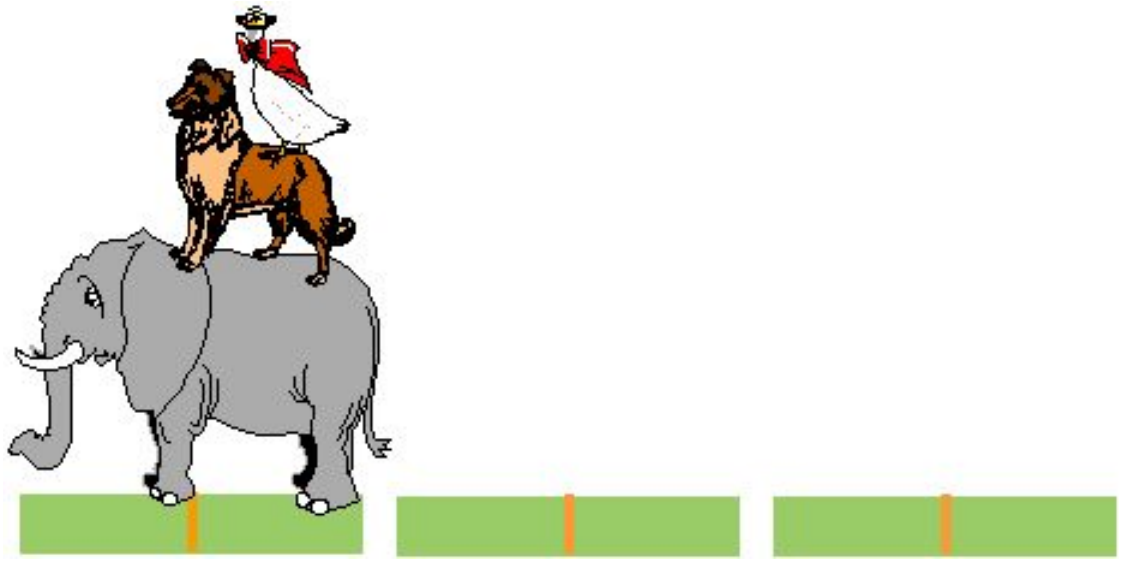
GIVEN: three poles

- a set of discs on the first pole, discs of different sizes, the smallest discs at the top
- GOAL: move all the discs from the left pole to the right one.
- CONDITIONS: only one disc may be moved at a time.
- A disc can be placed either on an empty pole or on top of a larger disc.









- Reverse data

Infix to Postfix

1) A + B * C
   a) (A + (B * C))
   b) (A + (B C *)) (A (B C *) +)
   c) A B C * +

2) (A + B) * C + D + E * F – G

   a) (((((A + B) * C) + D) + (E * F)) - G)

   b) (((((A B +) C *) D +) (E F *) +) G -)

   c) A B + C * D + E F * + G –

Infix to Postfix

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
…..**3.1** If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
…..**3.2** Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

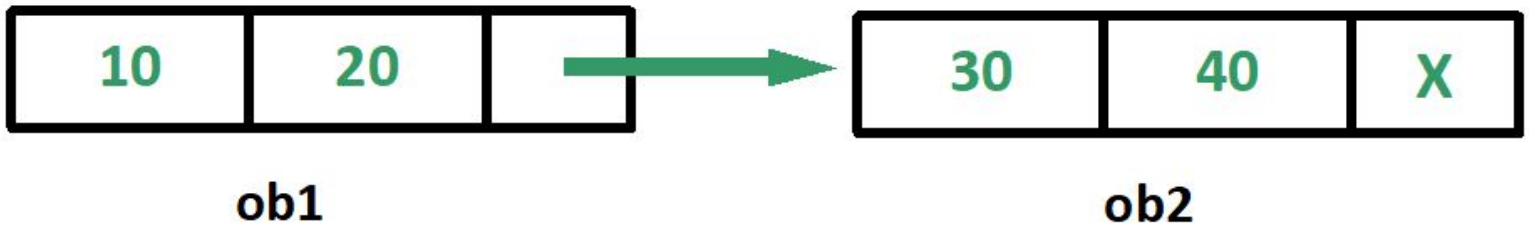
Symbol	In-Stack Priority	In-Coming Priority
)	-	-
**	3	4
*,/	2	2
binary +,-	1	1
(	0	4

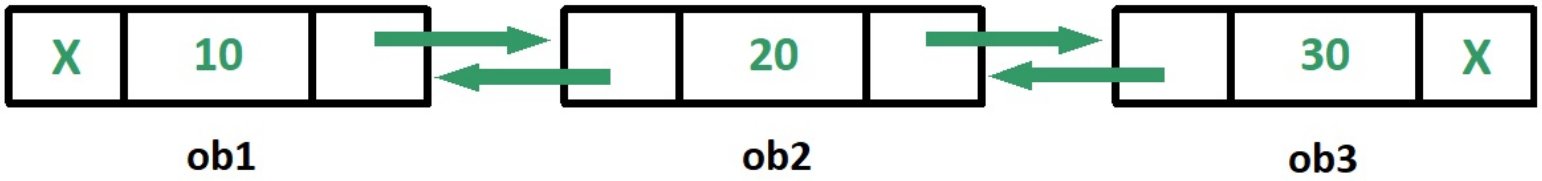
Self Referential Structures

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.

One in which one are more of its components is a pointer to itself.

```
struct node {
    int data1;
    char data2;
    struct node* link;
};
```


10
20
30
40
X
ob1
ob2


X
10
20
30
X
ob1
ob2
ob3

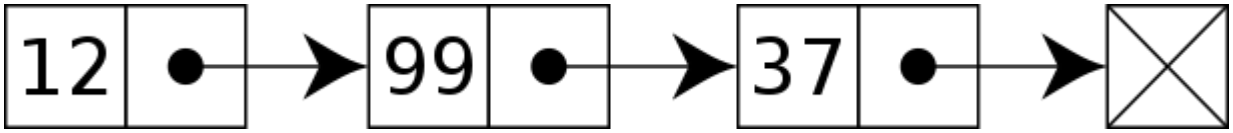
Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Linked List

- A linked list is a dynamic data structure.
- It is a data structure consisting of a collection of nodes which together represent a sequence.
- The number of nodes in a list is not fixed and can grow and shrink on demand
- In its most basic form, each node contains: data, and a reference (in other words, a *link*) to the next node in the sequence.
- This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration.



list elements can be easily inserted or removed without reallocation or reorganization of the entire structure

the data items need not be stored contiguously in memory

Linked lists do not allow random access to the data

An array is the data structure that contains a collection of similar type data elements whereas the Linked list is considered as non-primitive data structure contains a collection of unordered linked elements known as nodes.

Accessing an element in an array is fast, while Linked list takes linear time, so it is quite a bit slower

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

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
        int data;
        struct Node* next;
};
int main()
{
        struct Node* head = NULL;
        struct Node* second = NULL;
        struct Node* third = NULL;
        head = (struct Node*)malloc(sizeof(struct Node));
                second = (struct Node*)malloc(sizeof(struct Node));
        third = (struct Node*)malloc(sizeof(struct Node));
        head->data = 1;
        head->next = second;
        second->data = 2;
        second->next = third;
        third->data = 3;
        third->next = NULL;
        return 0;
}
```

Linked Stack

```
struct Node
{
    int data;
    struct Node* link;
};

Node* top;
```

Circular linked list

Circular linked list is a linked list where all nodes are connected to form a circle.

There is no NULL at the end.

A circular linked list can be a singly circular linked list or doubly circular linked list.

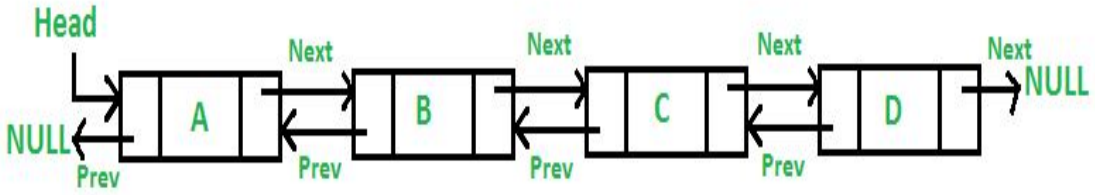
Advantages

- Any node can be a starting point.
- Useful for implementation of queue.
- Circular lists are useful in applications to repeatedly go around the list.
- Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

Doubly Linked List

A doubly linked list is one in which all nodes are linked together by multiple links which help in accessing both the successor and predecessor node for any arbitrary node within the list.

Every nodes in the doubly linked list has three fields: PrevPointer, NextPointer and DATA.



```
struct Node
{
    int data;
    struct Node* next;
    struct Node* prev;
};
```

Advantages

1) A DLL can be traversed in both forward and backward direction.

2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.

3)Can quickly insert a new node before a given node.

Disadvantages

1) Every node of DLL Require extra space for an previous pointer.

2) All operations require an extra pointer previous to be maintained.

Recursion

The process in which a function calls itself directly or indirectly is called recursion

Recursion means "defining a problem in terms of itself"

- Direct
- Indirect
- Tail End recursion

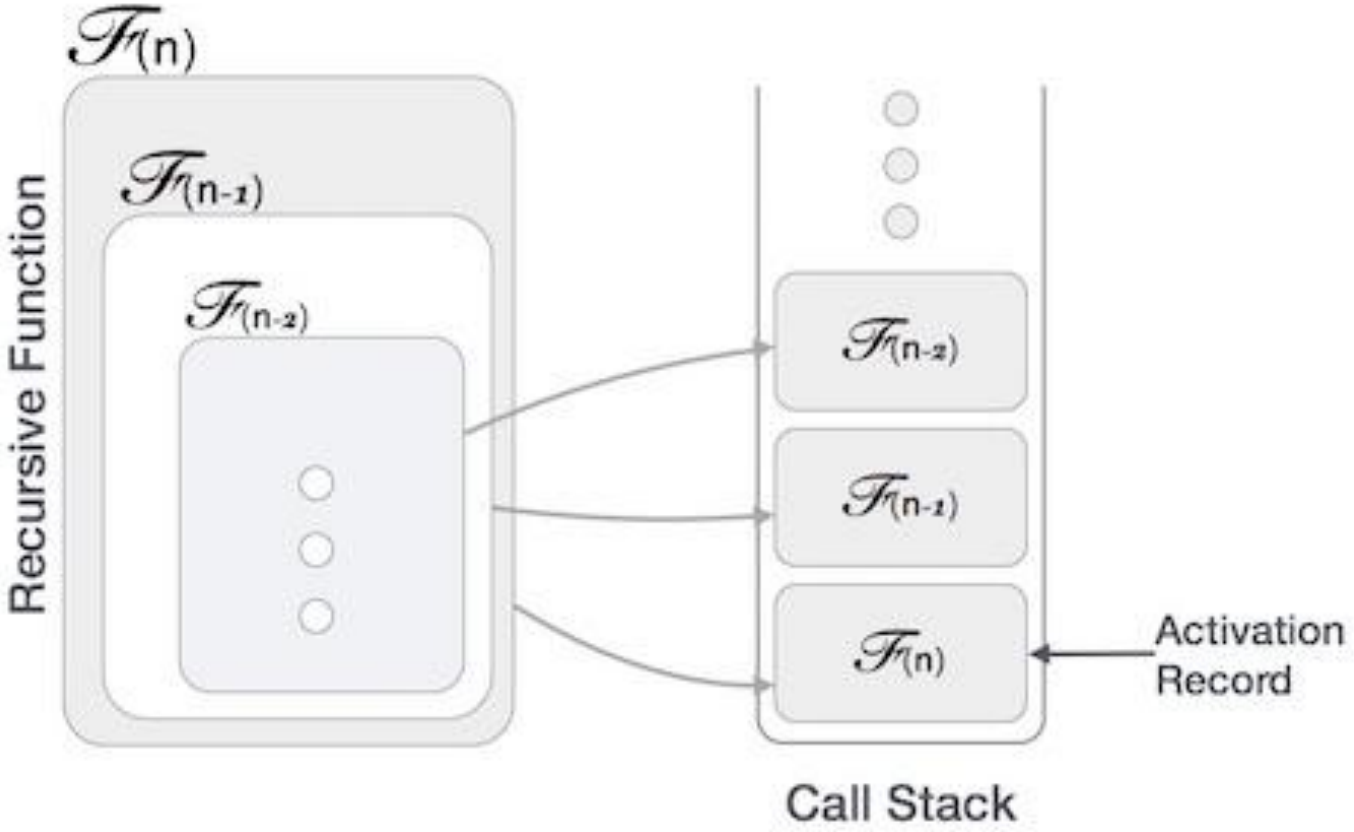
In recursion, a function **α** either calls itself directly or calls a function **β** that in turn calls the original function **α**. The function **α** is called recursive function.

Recursion is the process of defining a problem (or the solution to a problem) in terms of (a simpler version of) itself.

For example, we can define the operation "find your way home" as:

- ✔ If you are at home, stop moving.
- ✔ Take one step toward home.
- ✔ "find your way home".

Many programming languages implement recursion by means of stacks



Direct

```
int function(int value)
{

   if(value < 1)
      return;

   function(value - 1);

   printf("%d ",value);
}
function(3)
```

f(2) f(1) f(0)

Indirect

```
int function1(int value1)
 {

   if(value1 < 1)
      return;
   function2(value1 - 1);
   printf("%d ",value1);
}

int function2(int value2) {
   function1(value2);
}
```

wrong

```
int fact(int n)
{
    // wrong base case (it may cause stack overflow).
    if (n == 100)
        return 1;

    else
        return n*fact(n-1);
}

Fact(10)
```

```
#include<stdio.h>
void series(int n)
{
    if(n == 0)
     return;
     series(n-1);
     printf("%d ",n);
}
int main()
{
   series(10);
   return 0;
}
```

Output?

Properties

- Base Case (i.e., when to stop)
- Work toward Base Case
- Recursive Call (i.e., call itself)

- Base criteria – There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.

- Progressive approach – The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

Examples

Many mathematical functions can be defined recursively

Factorial
Fibonacci
Euclid's GCD (greatest common denominator)
Fourier Transform

- Many problems can be solved recursively
  eg games of all types from simple ones like the Towers of Hanoi problem to complex ones like chess

- Traversing a tree

Recursion or iteration?

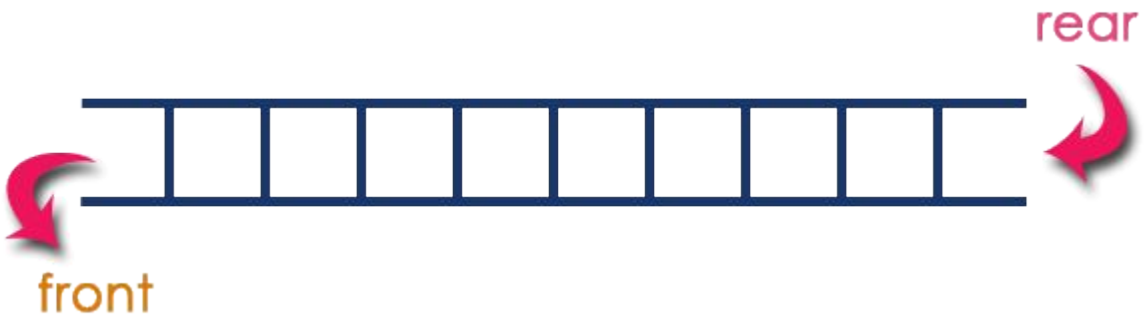
- Iterative algorithms have a bottom up approach
- Recursive algorithms are top-down.
- A recursion is always a suitable option when it comes to data abstraction.

- The implementation of recursion uses a lot of stack space, which can often result in redundancy.
- Every time we use recursion, we call a method that results in the creation of a new instance of that method.
- This new instance carries different parameters and variables, which are stored on the stack, and are taken on the return

Queue

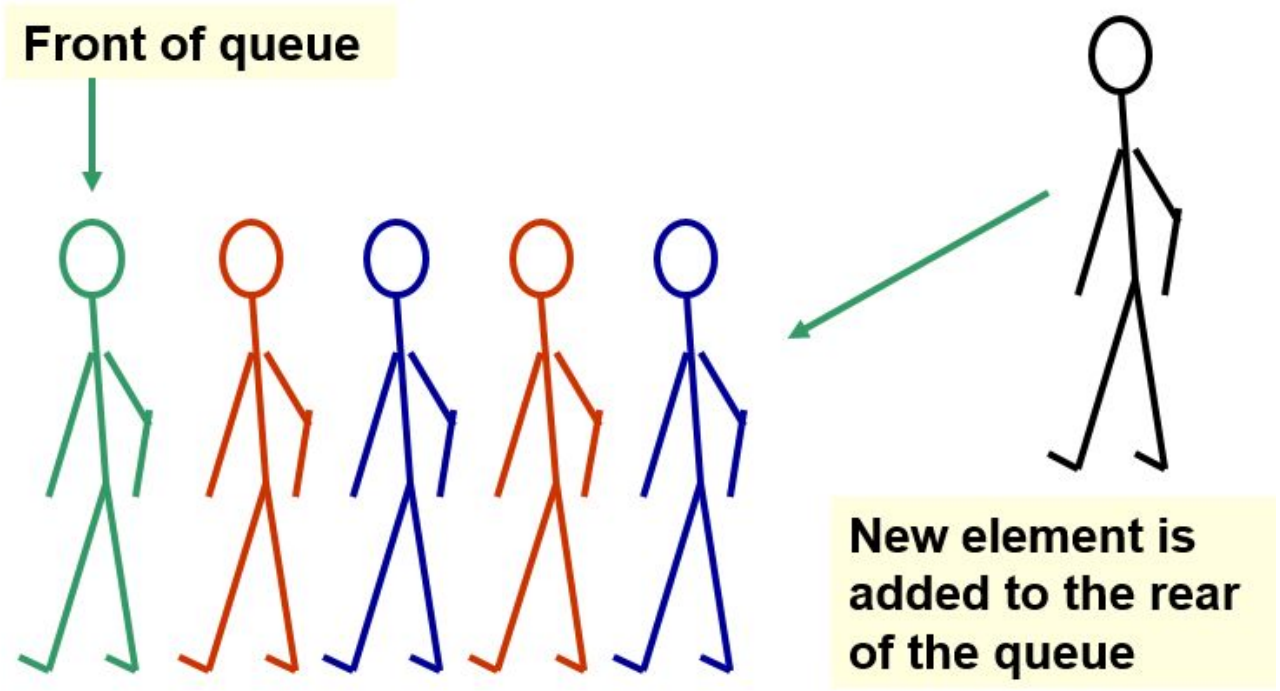
Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends.

In a queue data structure, the insertion operation is performed at a one end called '**rear**' and the deletion operation is performed at other end called '**front**'



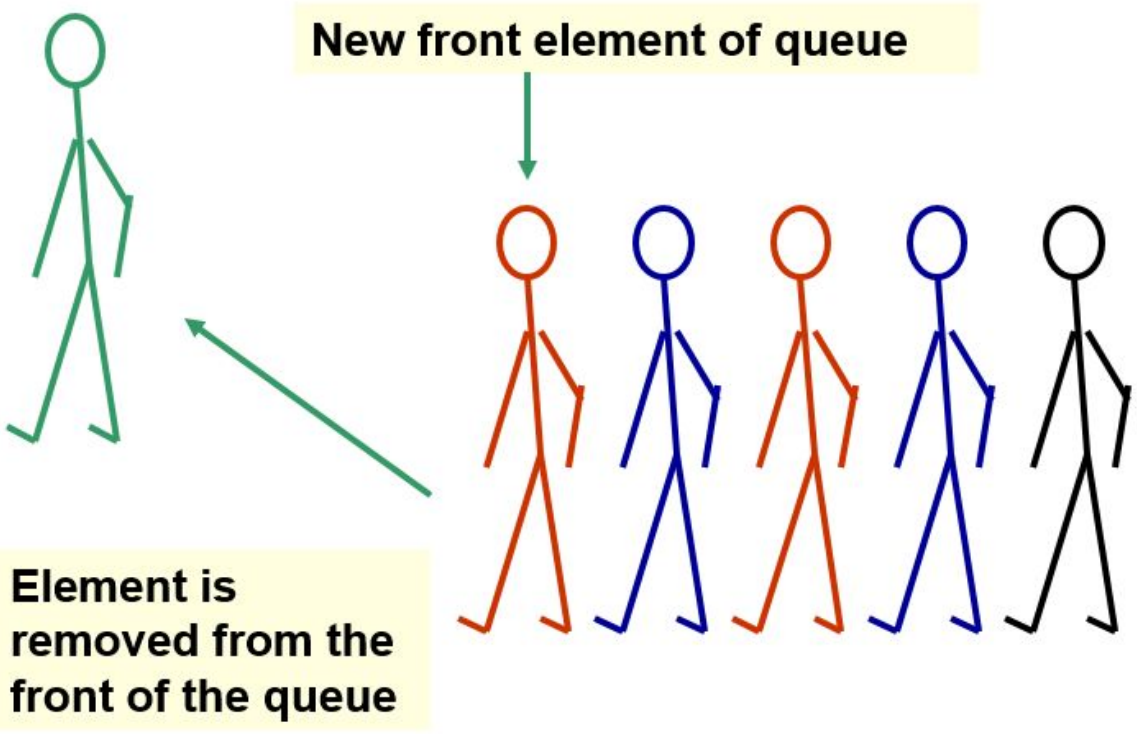
Conceptual View of a Queue

Adding an element



Conceptual View of a Queue

Removing an element



Operations on a Queue

Operation	Description
dequeue	Removes an element from the front of the queue
enqueue	Adds an element to the rear of the queue
first	Examines the element at the front of the queue
isEmpty	Determines whether the queue is empty
size	Determines the number of elements in the queue

- A queue is a FIFO (first in, first out) data structure