Basic Terminology (CO1)

- **Data-** values or set of values
- Data item (Group items and Elementary items)
- **Entity** –something with **attributes** (properties)
- Entity set (Entities with similar attributes)
- **Information-** meaningful or processed data
- Field, records and files

Basic Terminology (contd..)

- The above organization may not be complex enough to maintain and efficiently process certain collections of data. For this reason, data are also organized into more complex types of structures. The study of such structures includes the following three steps:
  - Logical or mathematical description.
  - Implementation of the structure.
  - Quantitative analysis of the structure.
    - Memory and time required

Introduction (CO1)

Why Data Structure

Introduction (CO1)

Why Data Structure

- Human requirement with computer are going to complex day by day. To solve the complex requirements in efficient way we need this study.

- Provide fastest solution of human requirements.

- Provide efficient solution of complex problem.

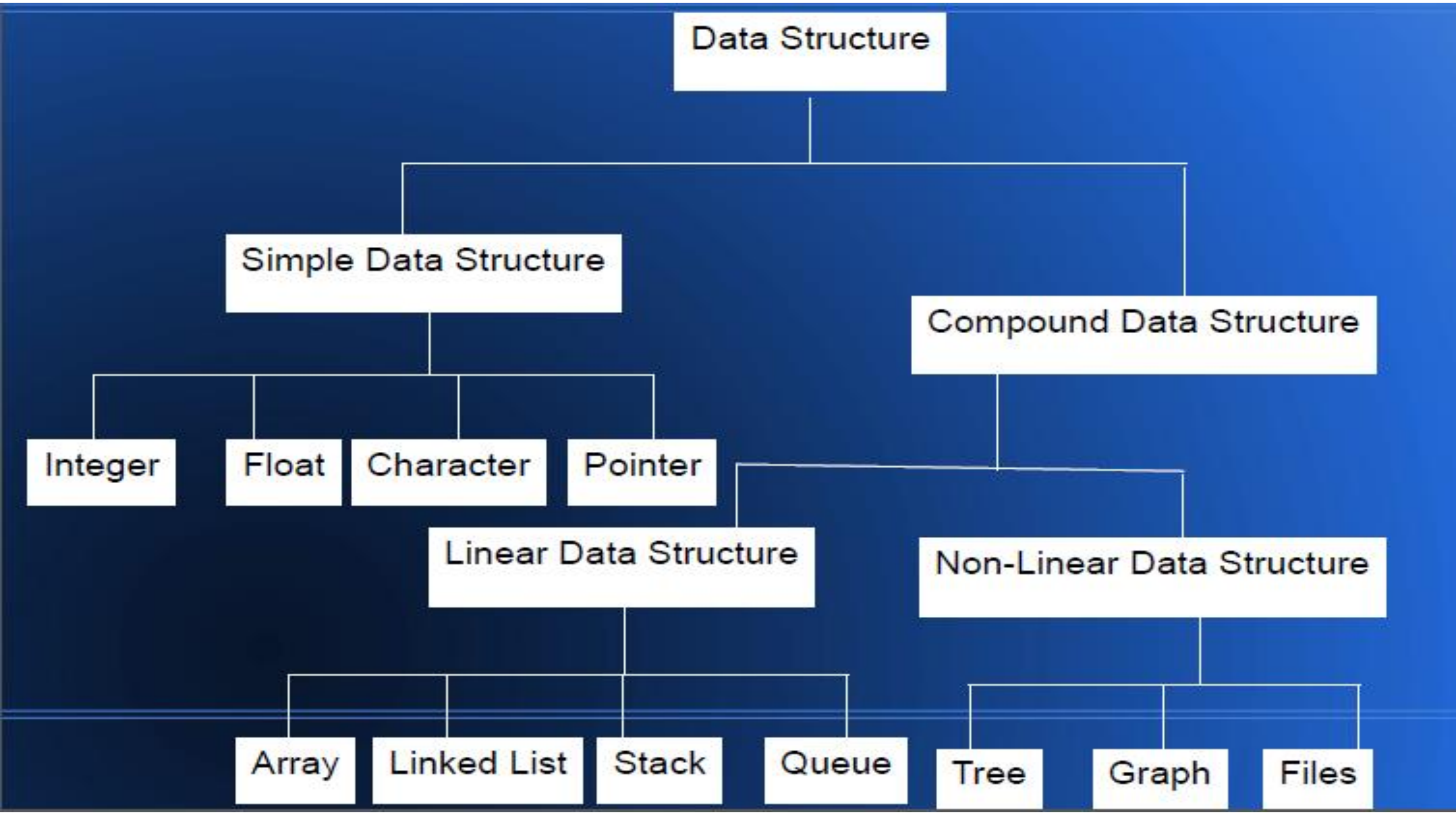
  >Space

  >Time

Data Structures(CO1)

- Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a data structure.
- The choice of a particular data model depends on two considerations.
  - It must be rich enough in structure to mirror the actual relationships of the data in the real world.
  - The structure should be simple enough that one can effectively process the data when necessary.

Classification of Data Structure



Introduction (CO1)

Classification of Data Structure ...

- Simple Data Structure /Primitive data structure: used to represent the standard data types of any one of the computer languages (integer, Character, float etc.).

- Compound Data Structure / Non Primitive Data Structure can be constructed with the help of any one of the primitive data structure and it is having a specific functionality. It can be designed by user. It can be classified as Linear and Non-Linear Data Structure.

Introduction (CO1)

Classification of Data Structure ...

- Linear Data Structures: A linear data structure traverses the data elements sequentially, in which only one data element can directly be reached. Ex: Arrays, Linked Lists.

- Non-Linear Data Structures: Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure. Ex: Trees, Graphs

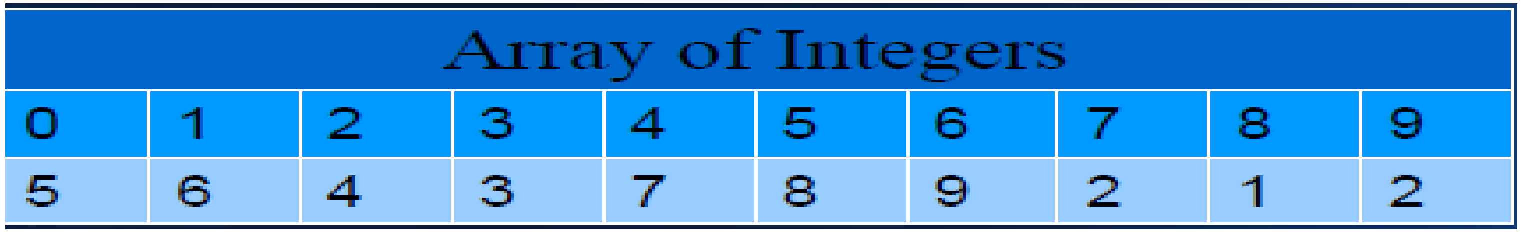
Operation on Linear/Non-Linear Data Structure

- Add an element
- Delete an element
- Traverse / Display
- Sort the list of elements
- Search for a data element

Introduction (CO1)

Types of Linear Data Structure

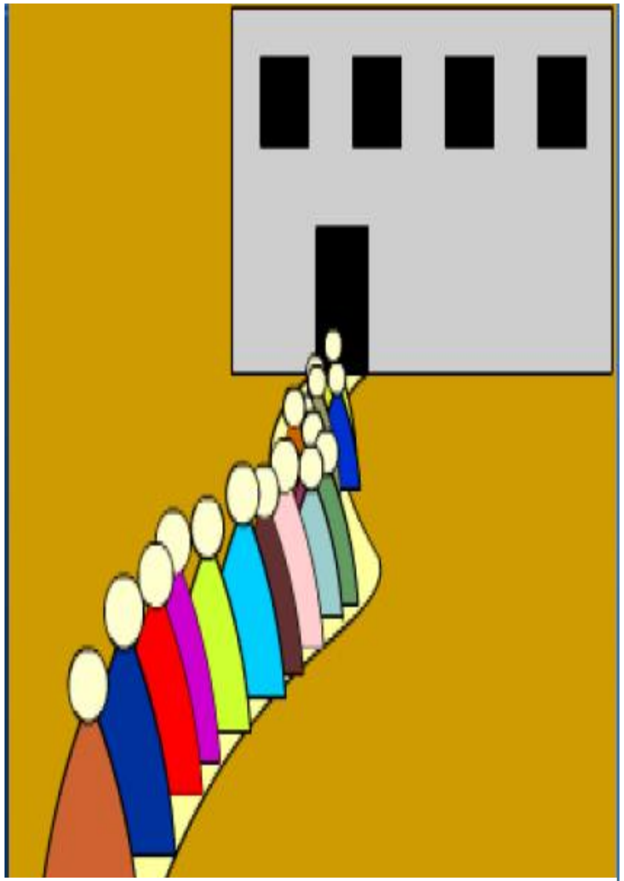
- Array : An array is the collection of the variables of the same data type that are referenced by the common name.
- int A[10], char B[10]



Array of Integers									
0	1	2	3	4	5	6	7	8	9
5	6	4	3	7	8	9	2	1	2

Types of Linear Data Structure…

- **Queue** is a linear data structure in which the insertion and deletion operations are performed at two different ends.
- The insertion is performed at one end and deletion is performed at another end.
- In a queue data structure, the insertion operation is performed at a position which is known as 'rear' and the deletion operation is performed at a position which is known as 'front'.
- In queue data structure, the insertion and deletion operations are performed based on FIFO (First In First Out) principle

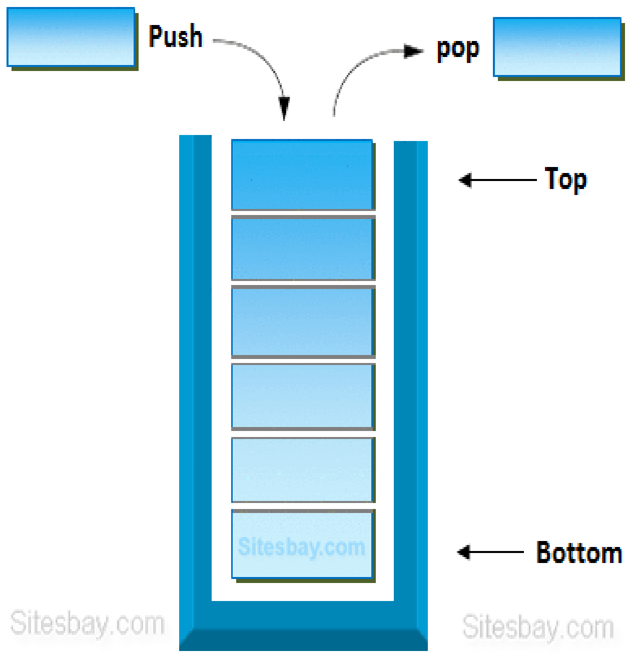


Introduction (CO1)

Types of Linear Data Structure….

- Stack

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at a single position which is known as "**top**". That means, a new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on **LIFO**(Last In First Out) principle.

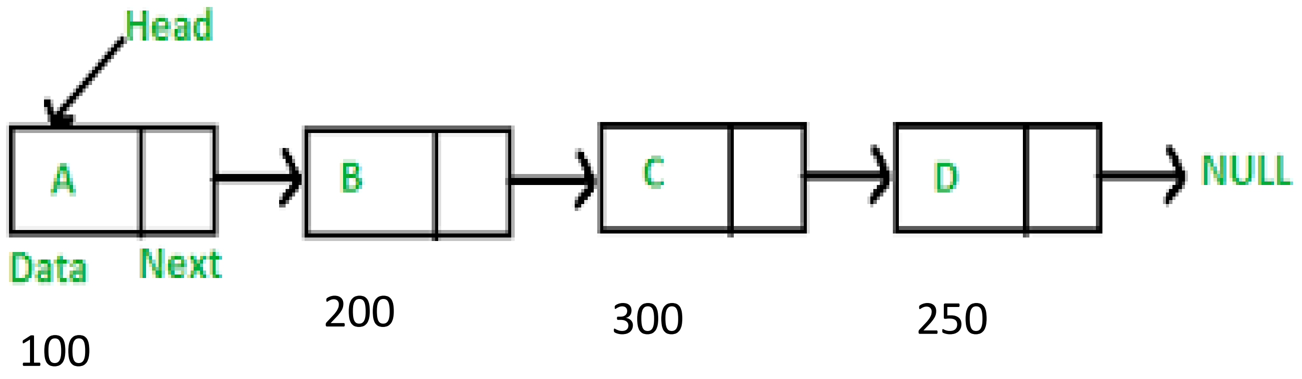


Types of Linear Data Structure…

- **LINKED LIST**

When we want to work with an unknown number of data values, we use a linked list data structure to organize that data.

The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "Node".

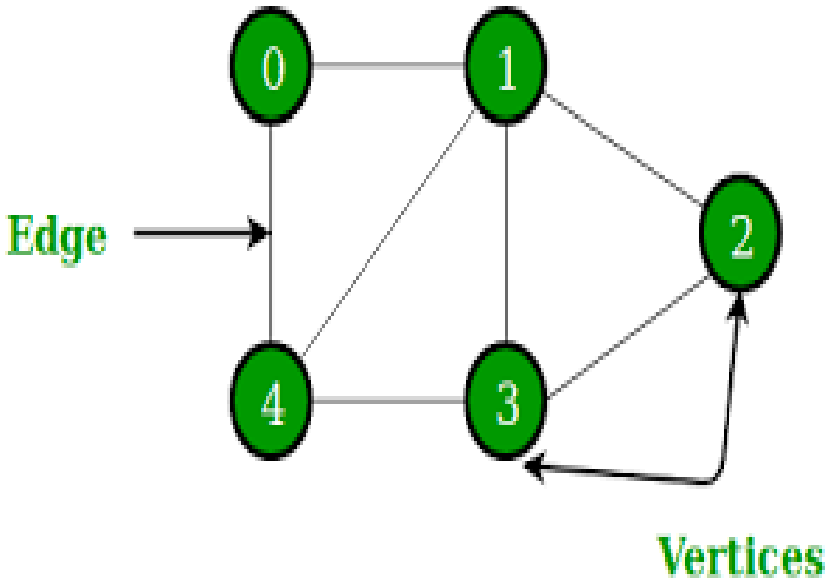


Introduction (CO1)

Types of NON Linear Data Structure..

- **Graph** is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices.

- Generally, a graph G is represented as G = (V , E), where V is set of vertices and E is set of edges



Types of NON Linear Data Structure

- **<u>Tree</u>** is a non-linear data structure which organizes data in hierarchical structure.

Data Structure Operations (CO1)

- **Traversing** : Accessing each record exactly once
- **Searching**: Finding the location of the record with a given key value
- **Inserting**: Adding a new record to the structure.
- **Deleting**: Removing a record from the structure.
- **Sorting**: Arranging the records in some logical order
- **Merging**: Combining the records in two different sorted files into a single sorted file

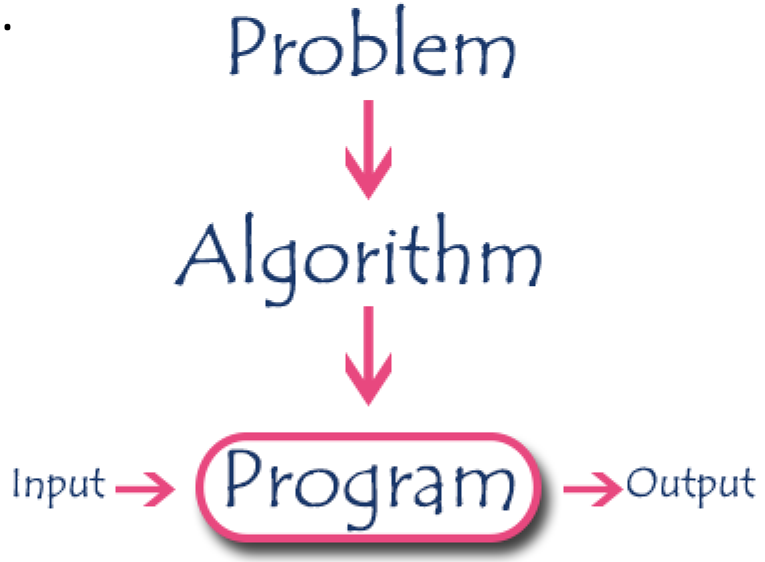
Algorithm(CO1)

- An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task.
- Every Algorithm must satisfy the following properties:
  - **Input**- There should be 0 or more inputs supplied externally to the algorithm.
  - **Output**- There should be at least 1 output obtained.
  - **Definiteness**- Every step of the algorithm should be clear and well defined.
  - **Finiteness**- The algorithm should have finite number of steps.
  - **Correctness**- Every step of the algorithm must generate a correct output.

Algorithms

- Algorithm is a **step-by-step** procedure, which defines a set of instructions to be executed in a **certain order** to get the **desired output.**

- Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.



Characteristics of an Algorithm

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code

Introduction to Algorithms(CO1)

Example:

An algorithm to sum N natural numbers.

```
Sum(A[],n)
{
        s=0;

        for(i=1 to n) do

                s=s+A[i]

        return s

}
```

An algorithm to add to matrix

```
Matrixadd(A[][],B[][],C[][],m,n)

{
        for(i=1 to n) do

                for(j=1 to n) do

                        c[i][j]=a[i][j]+b[i][j]

}
```

Example

Sum of n natural number

1. Take input an array from the user.
2. Initialize sum = 0
3. Repeat step 4 from 1 to size of the array
4. Add all the elements in the sum one by one
   sum = sum+a[i]
5. Print sum

Efficiency of Algorithm(CO1)

- An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space
- In order to compare algorithms, we must have some criteria to measure the efficiency of our algorithms
- The performance of an algorithm is measured on the basis of following properties :
  - Time Complexity
  - Space Complexity

Time and Space Complexity(CO1..)

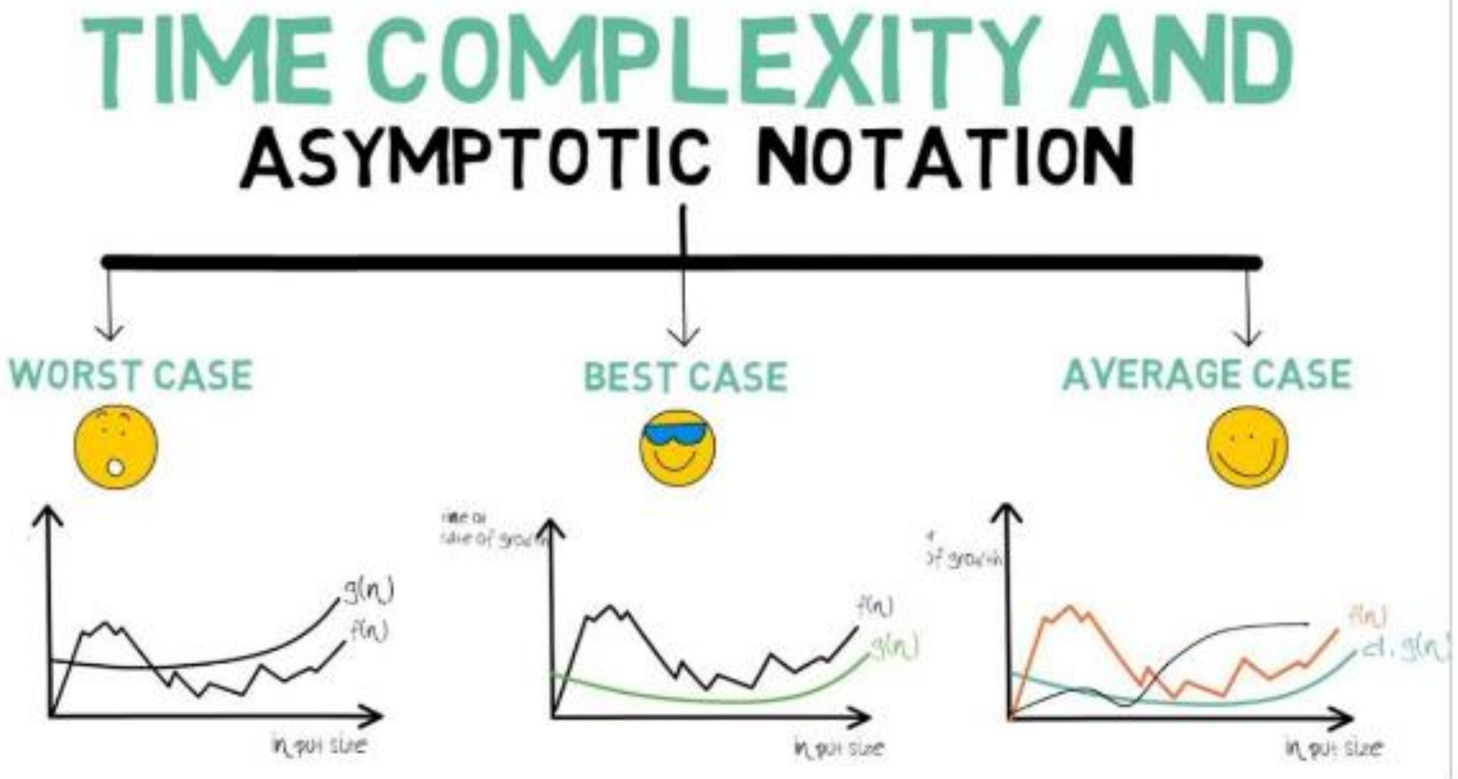
- **Time complexity**
  - Time Complexity is a way to represent the amount of time required by the program to run till its completion.
  - It's generally a good practice to try to keep the time required minimum, so that our algorithm completes its execution in the minimum time possible.
- **Space complexity**
  - It is the amount of memory space required by the algorithm, during the course of its execution.
  - An algorithm generally requires space for following components
    - **Instruction Space**
    - **Data Space**
    - **Environment Space**

Asymptotic Notations(CO1)

- Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance.
- Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.
- Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.
- Usually, the time required by an algorithm falls under three types −
  - **Best Case** − Minimum time required for program execution.
  - **Average Case** − Average time required for program execution.
  - **Worst Case** − Maximum time required for program execution.

Asymptotic Notations(CO1)

Asymptotic Notations



Introduction to Algorithms(CO1)

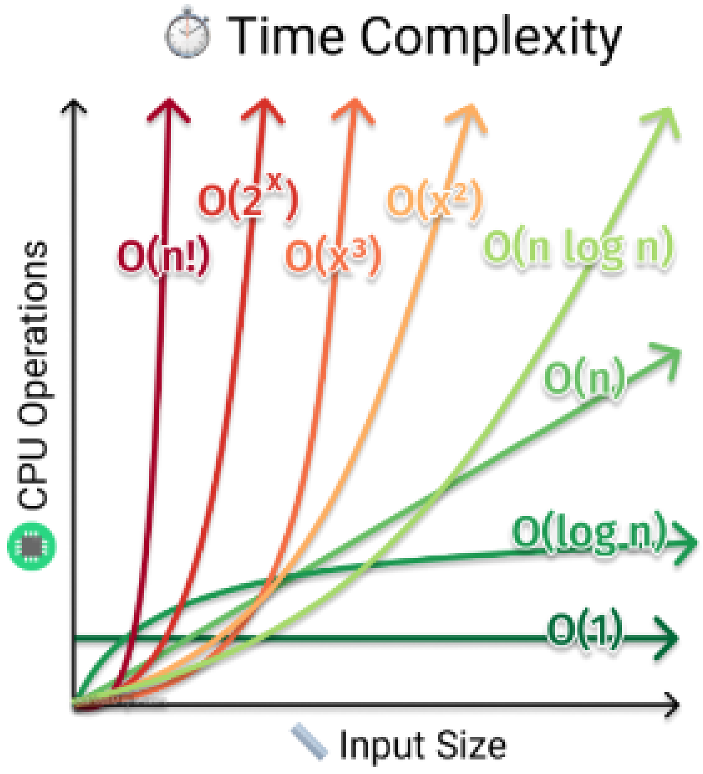
Purpose of Asymptotic Notations

1. To simplify the running time of the function
2. To describe the behavior of the function.
3. To provide asymptotic bound.
4. To describe how the running time of an algorithm increases with increase in input size

Growth of Function

Calculating Big O

- Break your algorithm/function into individual operations
- Calculate the Big O of each operation
- Add up the Big O of each operation together
- Remove the constants
- Find the highest order term — this will be what we consider the Big O of our algorithm/function

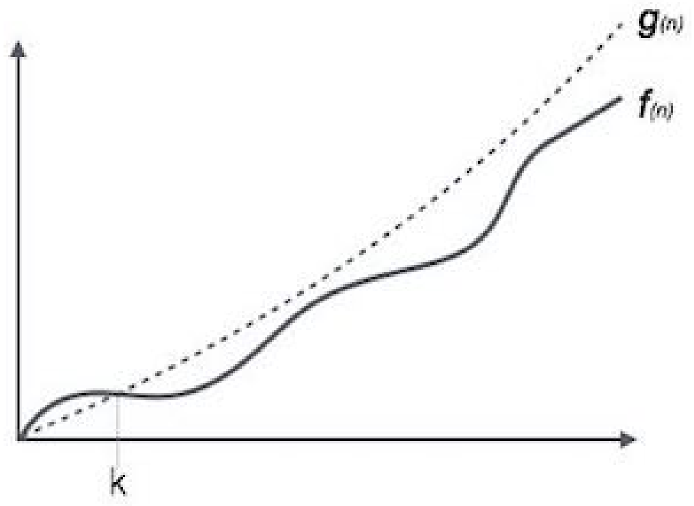


Asymptotic Notations(contd..)

- Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm:-
  - **O Notation (O)**
  - **Ω Notation (Ω)**
  - **θ Notation (Θ)**
  - **Little O (o) or Small O (o)**
  - **Little Omega ()**

Big Oh Notation, O

- The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

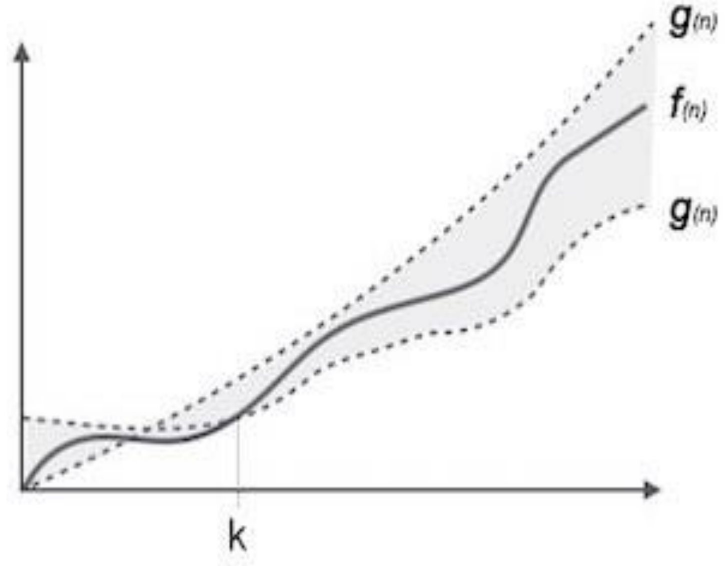


- k=n_0
- For example, for a function ***f*(n)**

  O(*f*(n)) = { *g*(n) : there exists c > 0 and n_0 such that *f*(n) ≤ c.*g*(n) for all n > n_0. }

Theta Notation, θ

- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.

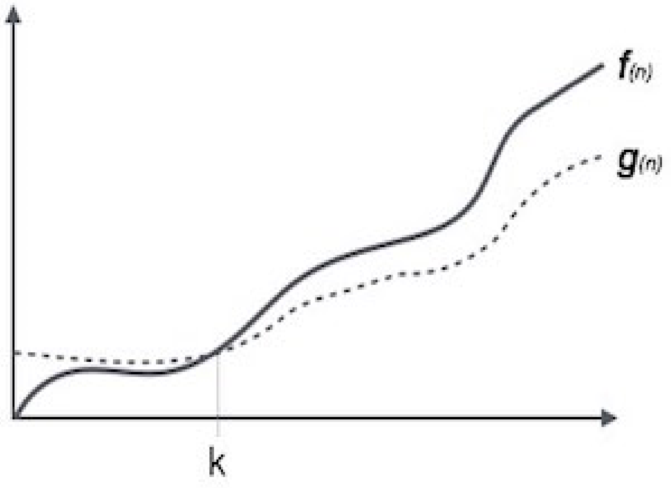


- k=n_0
- For example, for a function ***f*(n)**

  θ(*f*(n)) = { *g*(n) if and only if *g*(n) = O(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n_0. }

Omega Notation, Ω

- The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.



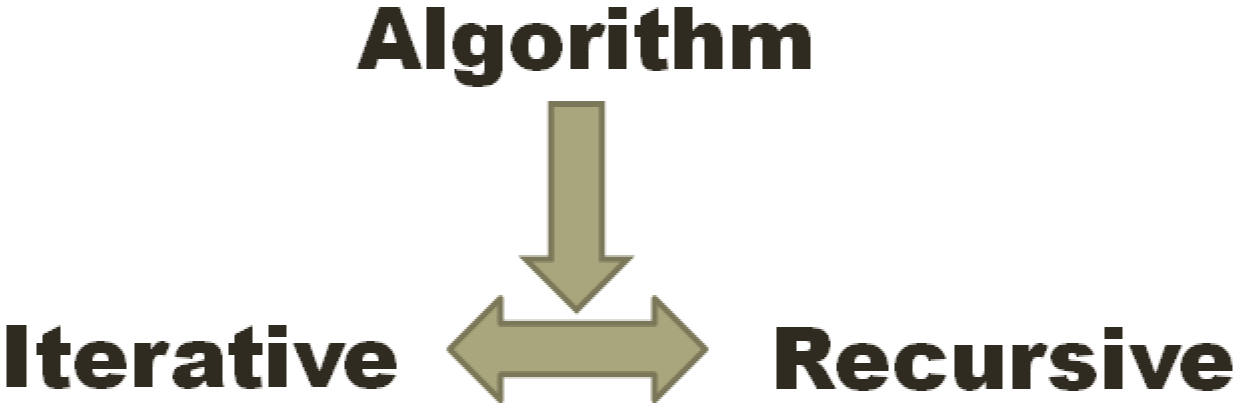
- $k=n_0$
- For example, for a function ***f*(n)**

  Ω(*f*(n)) ≥ { *g*(n) : there exists c > 0 and n_0 such that *g*(n) ≤ c.*f*(n) for all n > n_0. }

Common Asymptotic Notations

constant	$O(1)$
logarithmic	$O(\log n)$
linear	$O(n)$
n log n	$O(n \log n)$
quadratic	$O(n^2)$
cubic	$O(n^3)$
polynomial	$n^{O(1)}$
exponential	$2^{O(n)}$

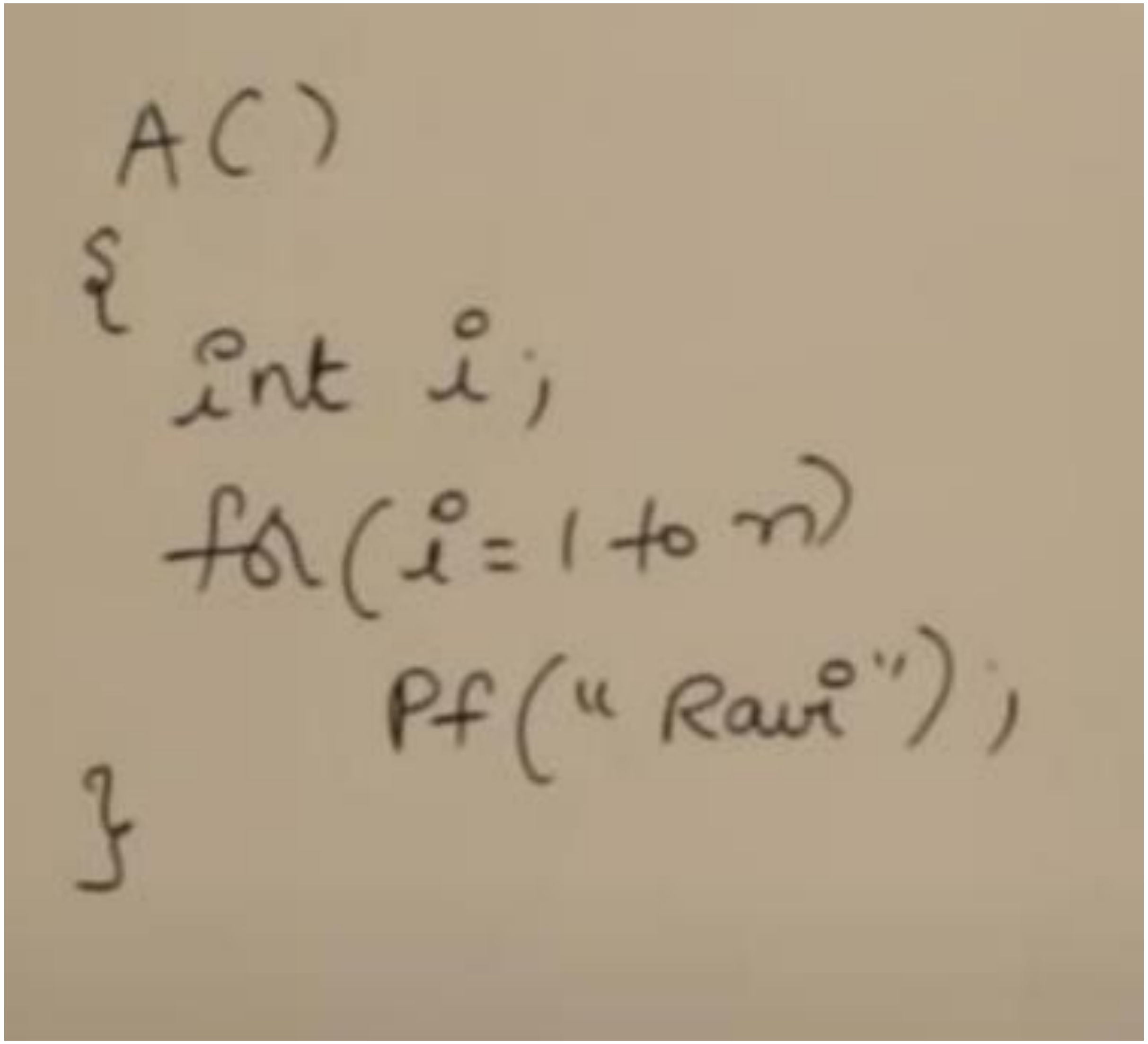
Algorithm



```
A()
{
for i=1 to n
max(a,b)
}
```

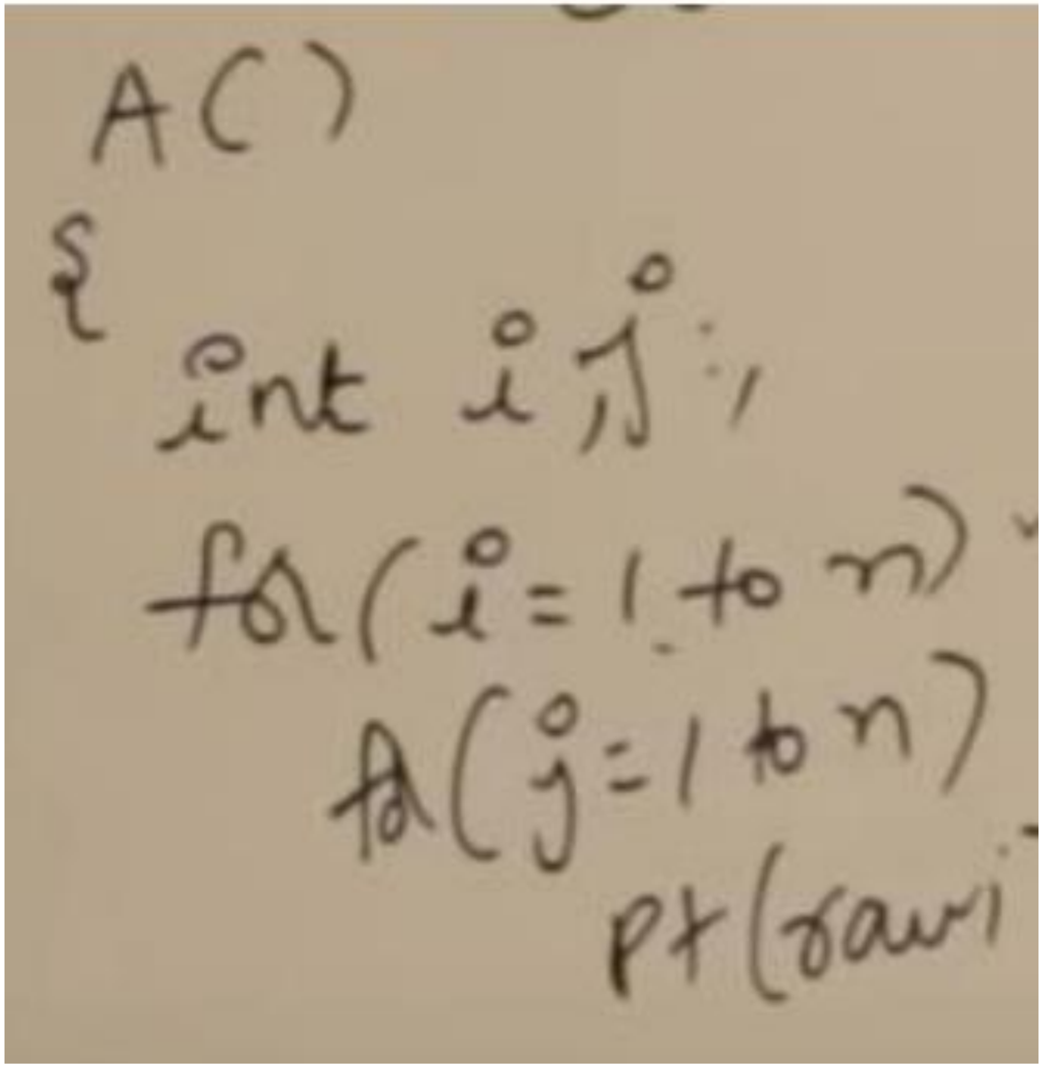
```
A(n)
{
If ()
A(n/2)
}
```

Complexity



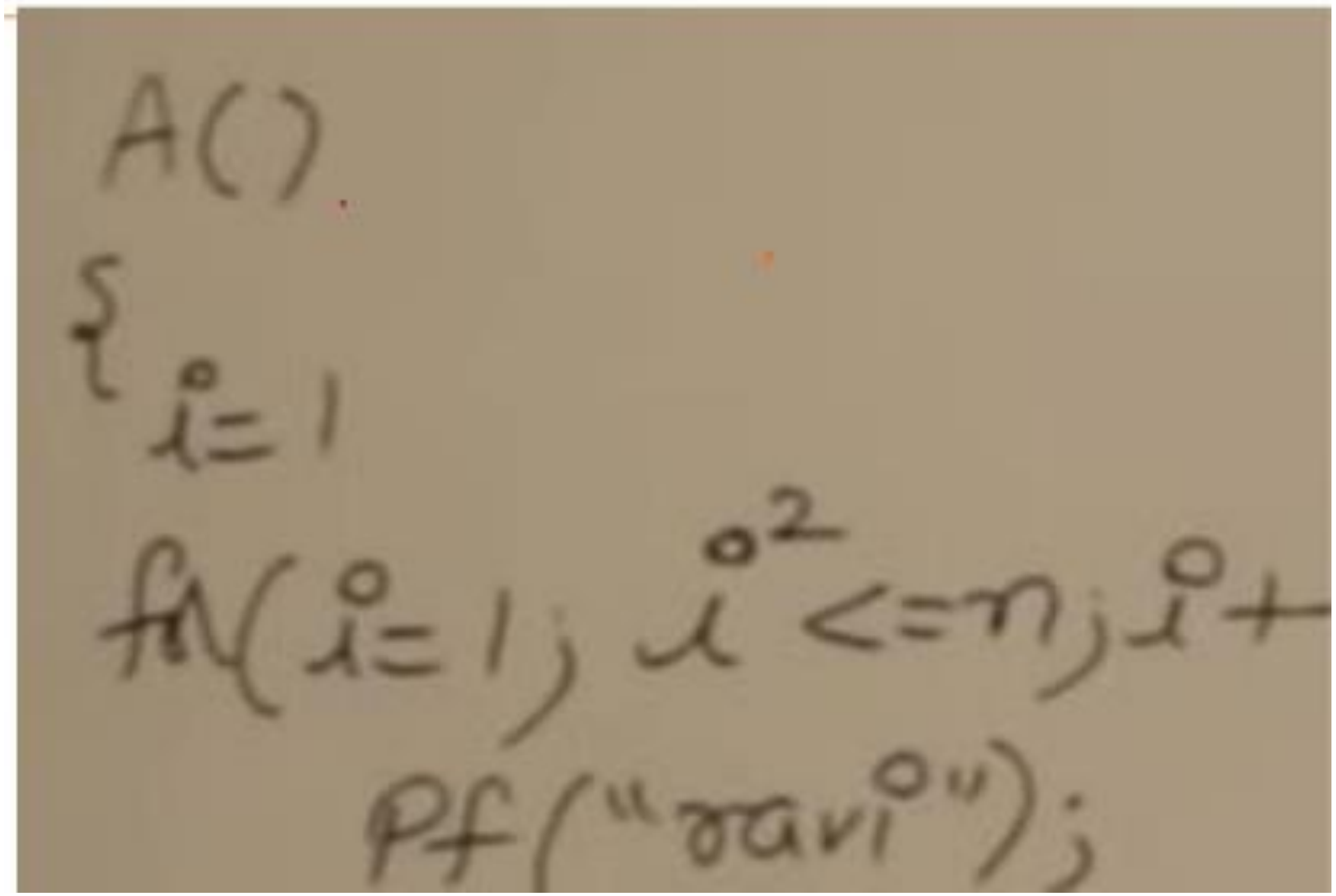
```
A()
{
  int i;
  for(i=1 to n)
       Pf("Ravi");
}
```

Complexity



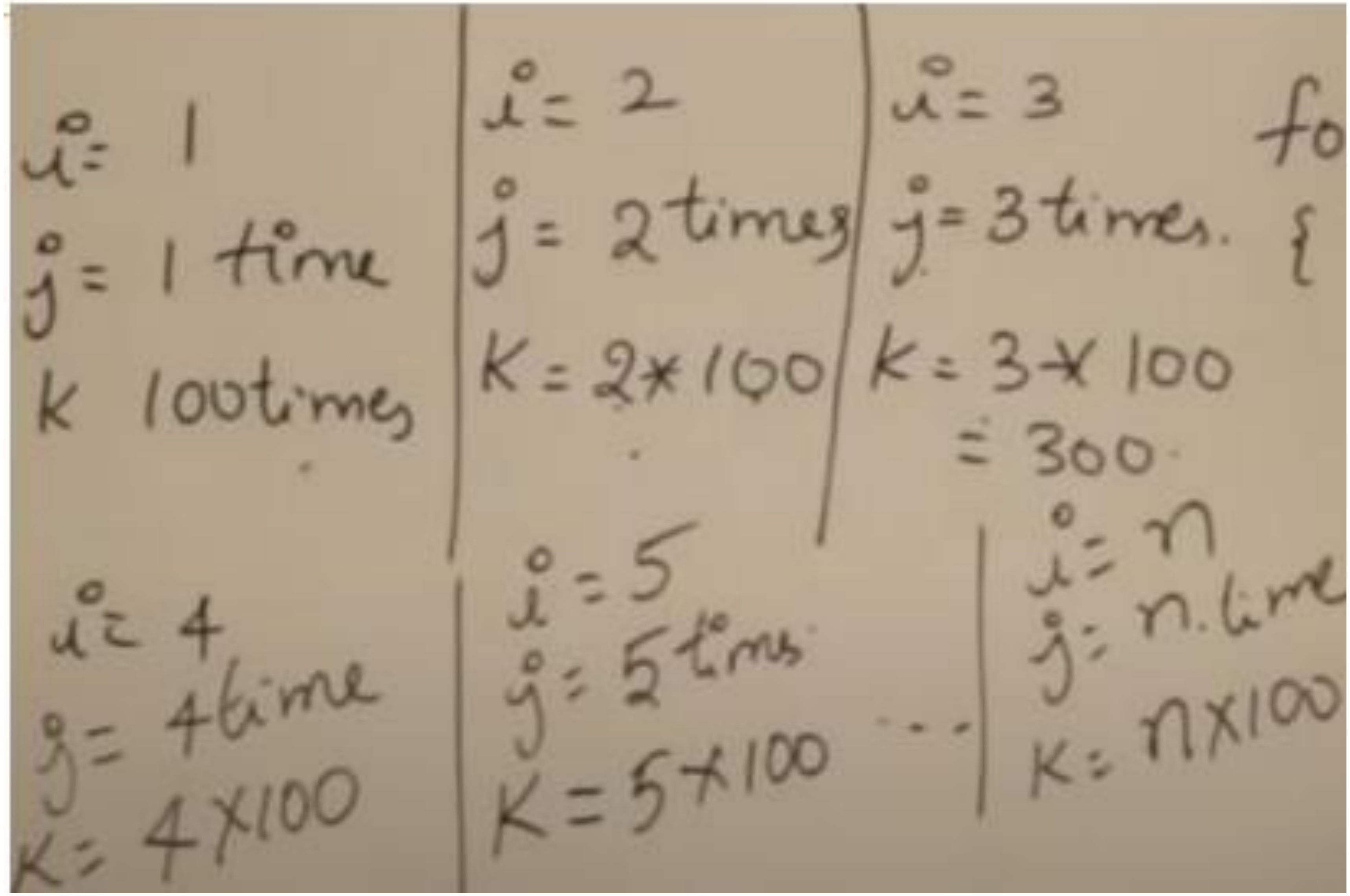
```
A()
{
    int i, j;
    for(i = 1 to n)
        for(j = 1 to n)
            pf(ravi
```

Complexity



```
A()
{
  i=1
  for(i=1; i^2<=n; i+
      Pf("ravi");
```

Complexity



$i = 1$
$j = 1$ time
k 100times

$i = 2$
$j = 2$ times
$K = 2 * 100$

$i = 3$
$j = 3$ times.
$k = 3 * 100$
$= 300$.

fo
{

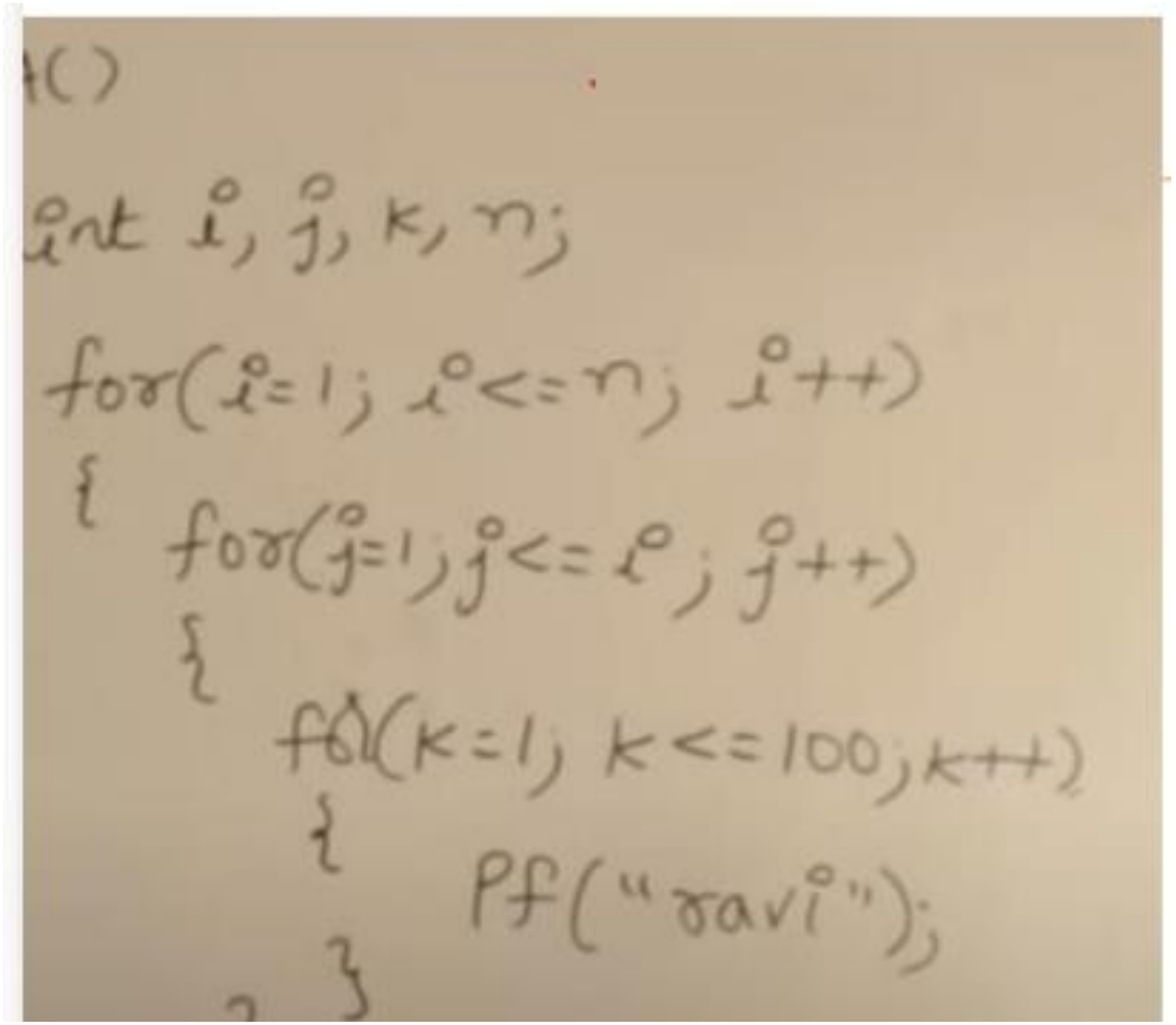
$i = 4$
$j = 4$time
$K = 4 \times 100$

$i = 5$
$j = 5$ tims
$K = 5 * 100$

. . .

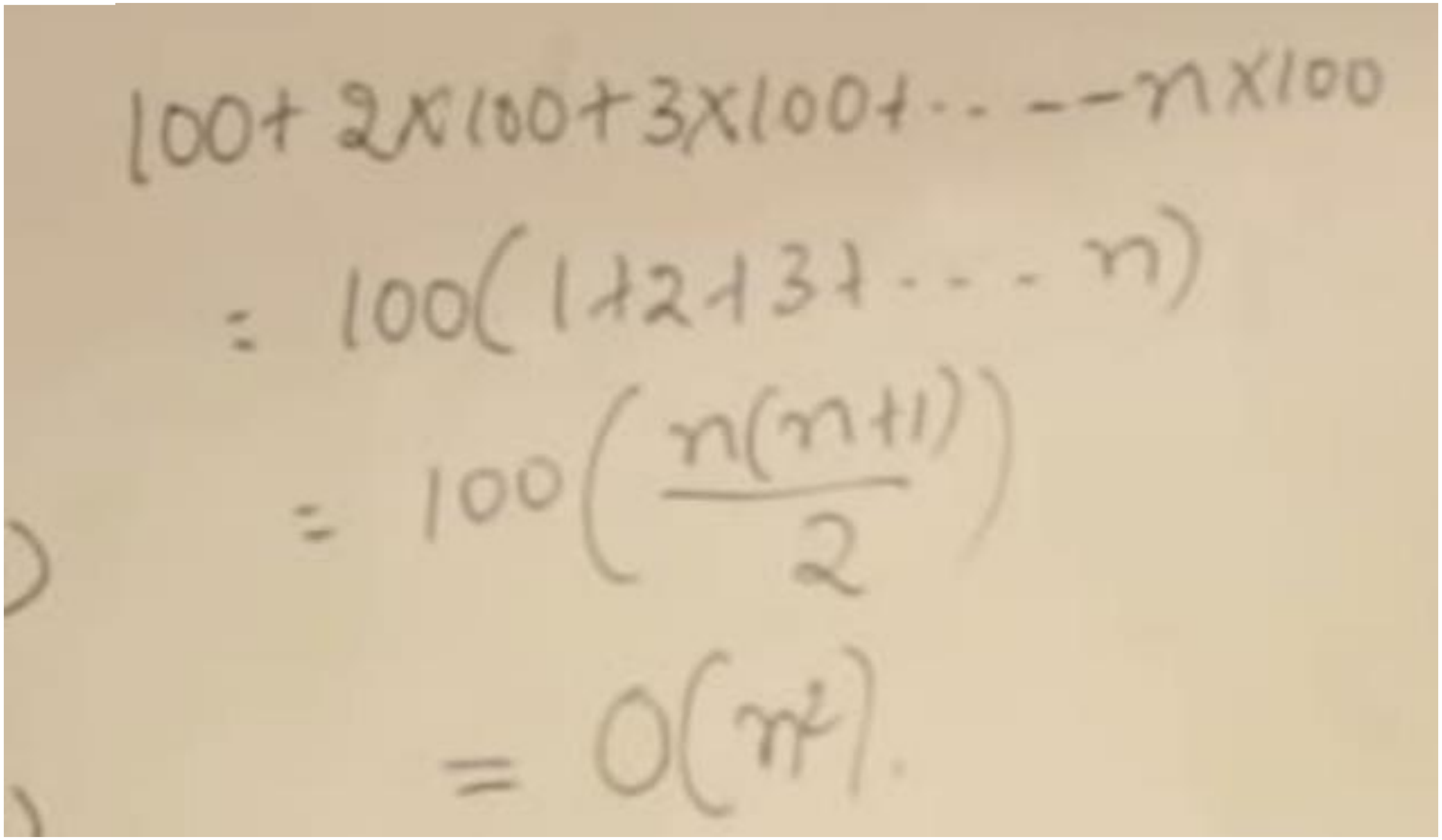
$i = n$
$j = n$.time
$K = n \times 100$

Complexity



```
A()
int i, j, k, n;
 for(i=1; i<=n; i++)
 {
     for(j=1; j<=i; j++)
     {
         for(k=1; k<=100; k++)
         {
              Pf("ravi");
         }
```

Complexity



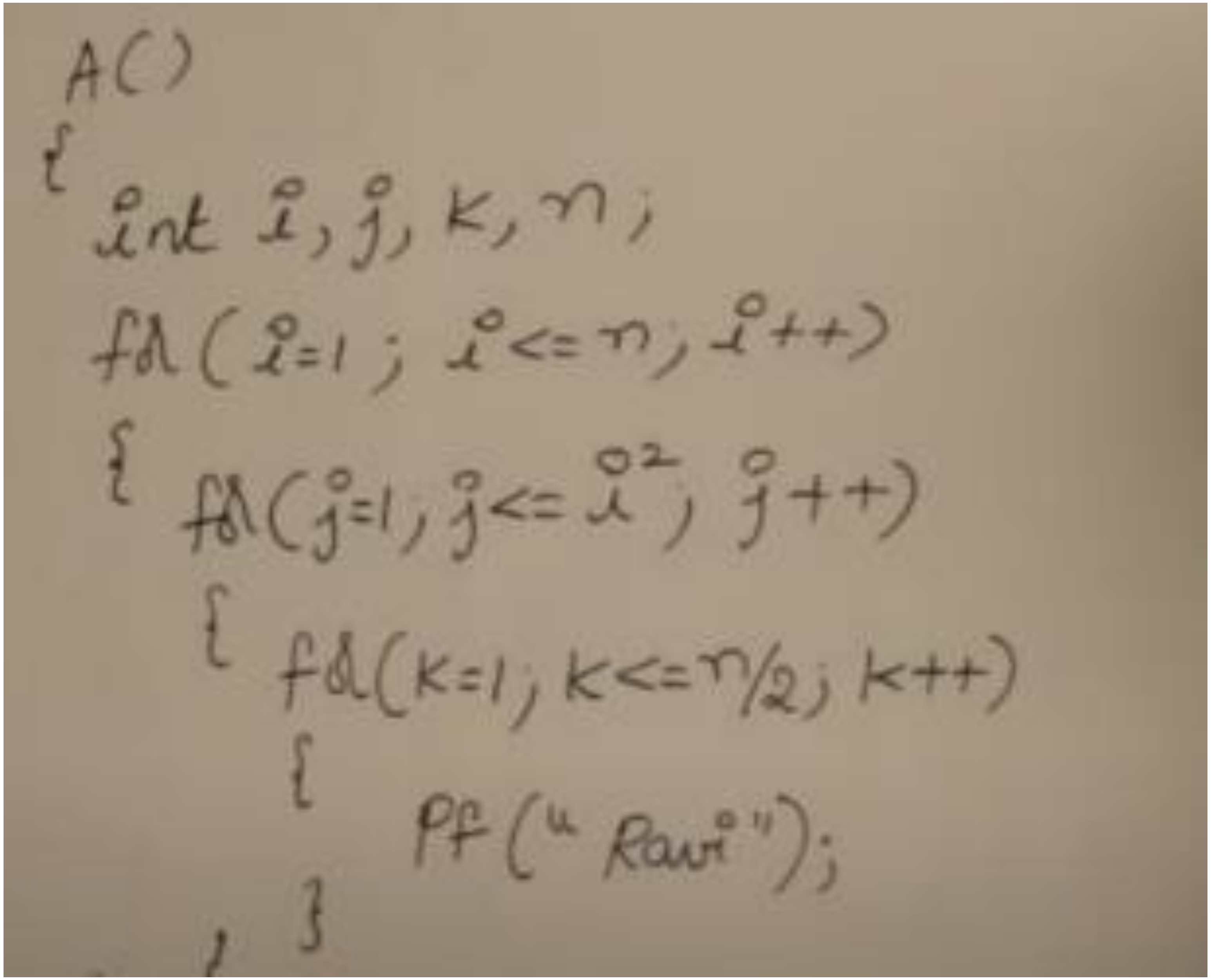
$$100 + 2\times100 + 3\times100 + \cdots - - n\times100$$

$$= 100(1+2+3+\cdots n)$$

$$= 100\left(\frac{n(n+1)}{2}\right)$$

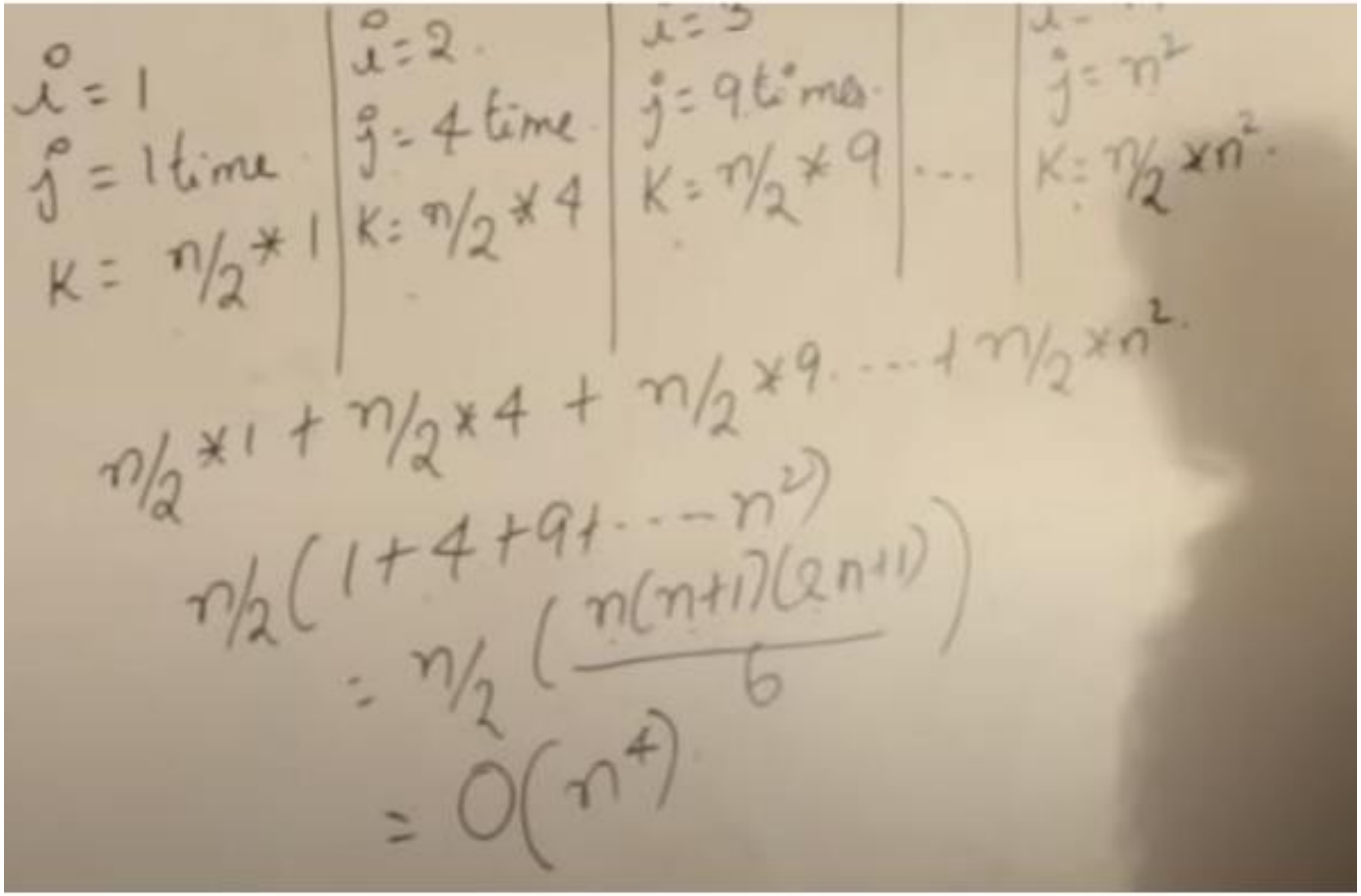
$$= O(n^2).$$

Complexity



```
A()
{
 int i, j, k, n;
 for(i=1; i<=n; i++)
 {
   for(j=1; j<=i^2; j++)
   {
     for(k=1; k<=n/2; k++)
     {
        Pf("Ravi");
     }
   }
```

Complexity



$i = 1$
$j = 1$ time
$K = n/2 * 1$

$i = 2$
$j = 4$ time
$K = n/2 * 4$

$i = 3$
$j = 9$ times
$K = n/2 * 9$

....

$i =$ [illegible]
$j = n^2$
$K = n/2 \times n^2$

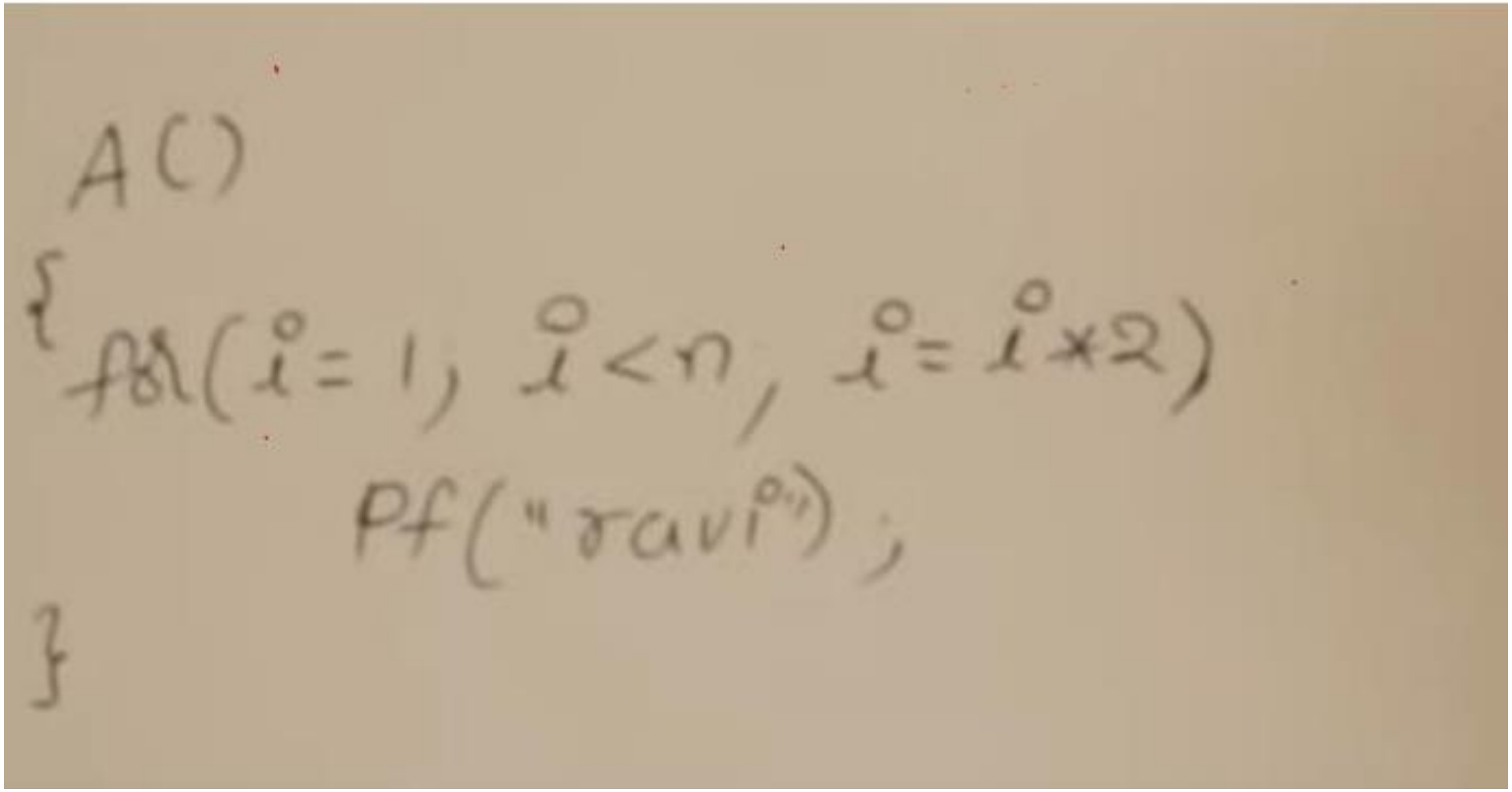
$$n/2 * 1 + n/2 * 4 + n/2 * 9 \cdots + n/2 \times n^2$$

$$n/2 (1 + 4 + 9 + \cdots n^2)$$

$$= n/2 \left(\frac{n(n+1)(2n+1)}{6} \right)$$

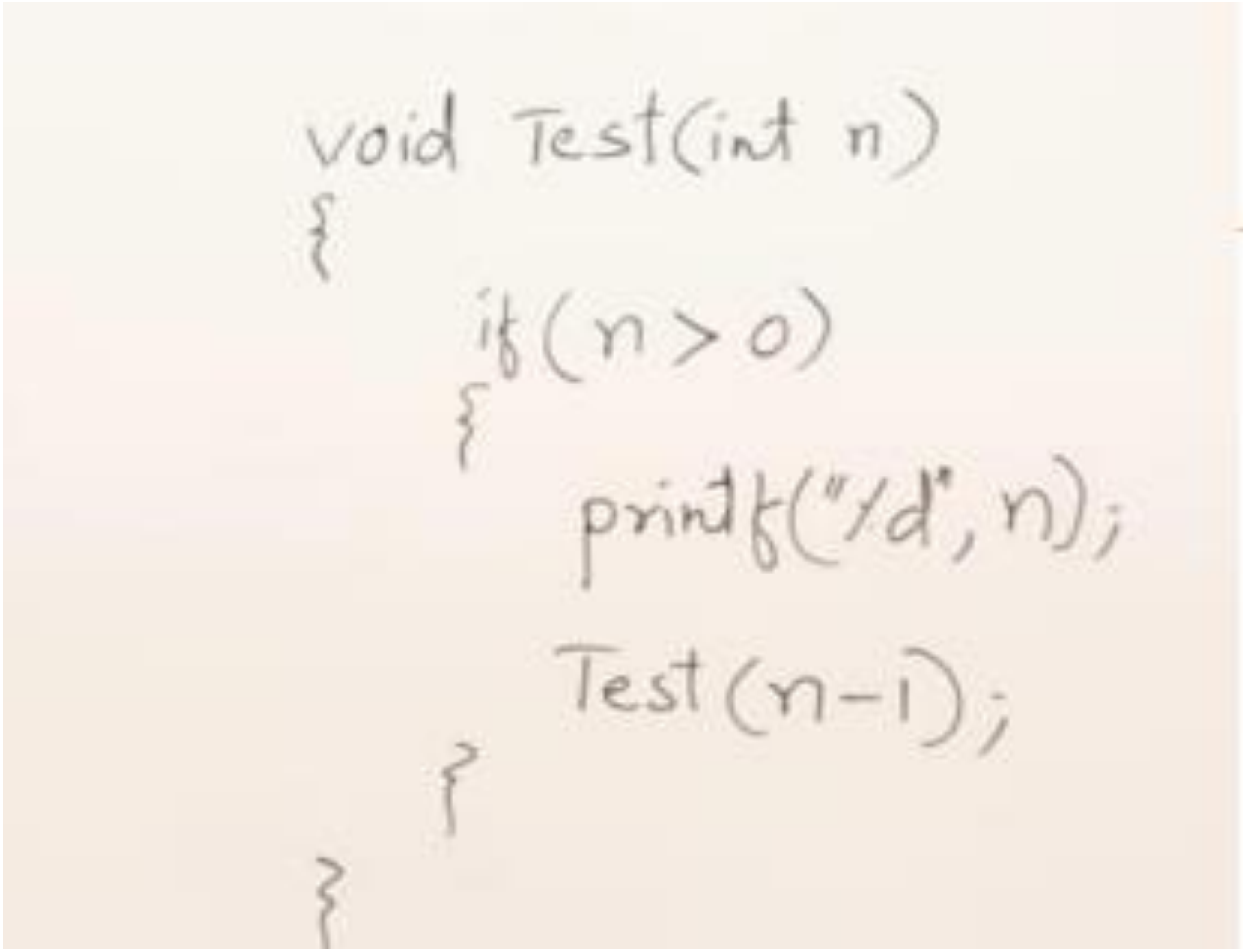
$$= O(n^4)$$

Complexity



```
A()
{
  for(i=1, i<n, i=i*2)
        Pf("ravi");
}
```

Complexity



```
void Test(int n)
{
    if(n>0)
    {
        printf("%d",n);
        Test(n-1);
    }
}
```

Time-space trade-off (CO1)

- Most computers have a large amount of space, but not infinite space.
- Also, most people are willing to wait a little while for a big calculation, but not forever.
- So if your problem is taking a long time but not much memory, a space-time trade-off would let you use more memory and solve the problem more quickly.
- Or, if it could be solved very quickly but requires more memory than you have, you can try to spend more time solving the problem in the limited memory.

Abstract Data Types (ADT) (CO1)

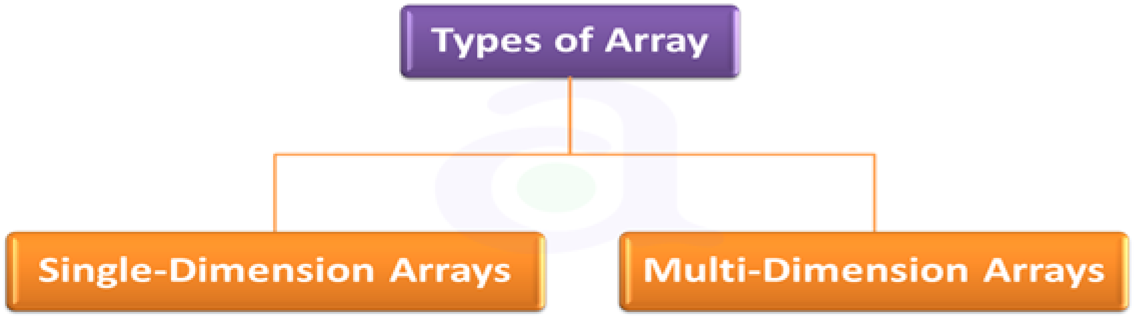
- The Data Type is basically a type of data that can be used in different computer program.
- It signifies the type like integer, float etc, the space like integer will take 4-bytes, character will take 1-byte of space etc.
- The abstract data type is special kind of data type, whose behavior is defined by a set of values and set of operations.
- The keyword "Abstract" is used as we can use these data types, we can perform different operations. But how those operations are working that is totally hidden from the user.
- The ADT is made of with primitive data types, but operation logics are hidden.

Abstract Data Types (contd..)

- Some examples of ADT are **Stack, Queue**, **List** etc.
- Let us see some operations of those mentioned ADT –
- Stack –
  - isFull(), This is used to check whether stack is full or not
  - isEmpty(), This is used to check whether stack is empty or not
  - push(x), This is used to push x into the stack
  - pop(), This is used to delete one element from top of the stack
- Queue –
  - isFull(), This is used to check whether queue is full or not
  - isEmpty(), This is used to check whether queue is empty or not
  - insert(x), This is used to add x into the queue at the rear end
  - delete(), This is used to delete one element from the front end of the queue

Arrays (CO1)

- An array is a collection of variables of the same type that are referred to through a common name.
- A specific element in an array is accessed by an index.
- In Python, all arrays consist of contiguous memory locations.



Topic Objective

- To learn about arrays and its types: single dimensional array and multi-dimensional array.
- Understand basic representation of array.
- To study applications of arrays.
- To learn sparse matrices and their representations.

Array Declaration

- Like other variables, arrays must be explicitly declared so that the compiler can allocate space for them in memory.
- The general form for declaring a single-dimension array is:
  - datatype var_name[size];
    - Here, **type** declares the base type of the array, which is the type of each element in the array
    - **size** defines how many elements the array will hold.

Array Declaration (Example)

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 27.

1	22	30	42	25	55	27	8	15	90

Array Creation in Python

Array is created in Python by importing array module to the python program. Then the array is declared as shown below.

```
from array import *
arrayName = array(typecode, [initializers])
```

Array Creation in Python (Cont.)

Typecode are the codes that are used to define the type of value the array will hold. Some common type codes used are:

TYPE CODE	VALUE
b	Represents signed integer of size 1 byte
B	Represents unsigned integer of size 1 byte
c	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
I	Represents unsigned integer of size 2 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

Array Creation in Python (Example)

The below code creates an array named array1.
LISTS as ARRAYS, however, to work with arrays in Python you will have to import a library, like the NumPy library.

```
array1 = array('i', [10,20,30,40,50])

for x in array1:
  print(x)
```

When we compile and execute the above program, it produces the following result −

Output

```
10
20
30
40
50
```

Accessing elements of an Array

We can access each element of an array using the index of the element. The below code shows how

```
from array import *

array1 = array('i', [10,20,30,40,50])

print (array1[0])

print (array1[2])
```

When we compile and execute the above program, it produces the following result − which shows the element is inserted at index position 1.

Output

```
10
30
```

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we add a data element at the middle of the array using the python in-built insert() method.

```
from array import *
array1 = array('i', [10,20,30,40,50])
array1.insert(1,60)
for x in array1:
  print(x)
```

Insertion Operation (Cont.)

When we compile and execute the above program, it produces the following result which shows the element is inserted at index position

Output

```
10
60
20
30
40
 50
```

Deletion Operation (Cont.)

When we compile and execute the above program, it produces the following result which shows the element is removed from the array.

Output

```
10
20
30
50
```

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Here, we remove a data element at the middle of the array using the python in-built remove() method.

```
from array import *
array1 = array('i', [10,20,30,40,50])
array1.remove(40)
for x in array1:
 print(x)
```

Search Operation (Cont.)

You can perform a search for an array element based on its value or its index. Here, we search a data element using the python in-built index() method.

```
from array import *

array1 = array('i', [10,20,30,40,50])

print (array1.index(40))
```

When we compile and execute the above program, it produces the following result which shows the index of the element. If the value is not present in the array then the program returns an error.

Output

```
  3
```

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Here, we simply reassign a new value to the desired index we want to update.

```
from array import *

array1 = array('i', [10,20,30,40,50])

array1[2] = 80

for x in array1:
 print(x)
```

Update Operation (Cont.)

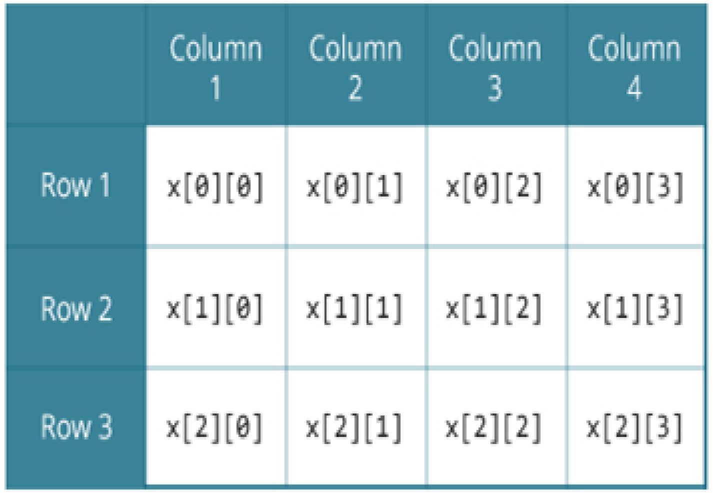
When we compile and execute the above program, it produces the following result which shows the new value at the index position 2.

Output

```
10
20
80
40
50
```

Multidimensional Array (CO1)

- In C programming, you can create an array of arrays.
- These arrays are known as multidimensional arrays.



	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

- For example,
  - float x[3][4];
    - Here, x is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array as a table with 3 rows and each row has 4 columns as shown in figure.

Multidimensional Array

- Similarly, you can declare a three-dimensional (3d) array.
- For example,
  - float y[2][4][3];
    - Here, the array y can hold 24 elements.

Row-Major Order

- Row Major Order is a method of representing multi dimension array in sequential memory.
- In this method elements of an array are arranged sequentially row by row.
- Thus elements of first row occupies first set of memory locations reserved for the array, elements of second row occupies the next set of memory and so on.
- Consider a Two Dimensional Array consist of N rows and M columns. It can be stored sequentially in memory row by row as shown below:

Row 0	**A[0,0]**	**A[0,1]**	**…………….**	**A[0,M-1]**
Row 1	A[1,0]	A[1,1]	…………….	A[1,M-1]
	…………………………………………………………			
Row N-1	A[N-1,0]	A[N-1,1]	…………….	A[N-1,M-1]

Representation of arrays (CO1)

- A 2D array’s elements are stored in continuous memory locations. It can be represented in memory using any of the following two ways:

  1. Row-Major Order
  2. Column-Major Order

Row-Major Order

- Example:
  Consider following example in which a two dimensional array consist of two rows and four columns is stored sequentially in row major order as:

2000	A[0][0]	Row 0
2002	A[0][1]	
2004	A[0][2]	
2006	A[0][3]	
2008	A[1][0]	Row 1
2010	A[1][1]	
2012	A[1][2]	
2014	A[1][3]	

Row-Major Order

- The Location of element A[i, j] can be obtained by evaluating expression:
  - **LOC (A [i, j]) = Base_Address + W [M (i-L1) + (j-L2)]**
    - **Here,**
      - **Base_Address** is the address of first element in the array.
      - **W** is the word size. It means number of bytes occupied by each element.
      - **N** is number of rows in array.
      - **M** is number of columns in array.
      - **L1** is lower bound of row.
      - **L2** is lower bound of column.

Row-Major Order

- Suppose we want to calculate the address of element A [1, 2].
  - It can be calculated as follow:
    - **Here,**

      Base_Address = 2000, W= 2, M=4, N=2, i=1, j=2

      LOC (A [i, j])=**Base_Address + W [M (i-L1) + (j-L2)]**

      LOC (A[1, 2])=2000 + 2 *[4*(1) + 2]

      =2000 + 2 * [4 + 2]

      =2000 + 2 * 6

      =2000 + 12

      =2012

Row-Major Order

1. Given an array, **arr[1………10][1………15]** with base value **100** and the size of each element is **1 Byte** in memory. Find the address of **arr[8][6]** with the help of row-major order.

Row-Major Order

Base address B = 100

Storage size of one element store in any array W = 1 Bytes

Row Subset of an element whose address to be found I = 8

Column Subset of an element whose address to be found J = 6

Lower Limit of row/start row index of matrix LR = 1

Lower Limit of column/start column index of matrix = 1

Number of column given in the matrix N = Upper Bound – Lower Bound + 1

= 15 – 1 + 1

= 15

Row-Major Order

- ***Formula:***
  *Address of A[I][J] = B + W * ((I – LR) * N + (J – LC))*
- ***Solution:***
  *Address of A[8][6] = 100 + 1 * ((8 – 1) * 15 + (6 – 1))*
  *= 100 + 1 * ((7) * 15 + (5))*
  *= 100 + 1 * (110)*
  Address of A[I][J] = 210

Row-Major Order

1. A 2-D Array defined as a[4….7] [-1…3] requires two bytes of storage space for each element. If the array is stored in the row major form then calculate the address of element. a[6][2] base address is 100.

Column Major Order

- Column Major Order is a method of representing multidimensional array in sequential memory. In this method elements of an array are arranged sequentially column by column. Thus elements of first column occupies first set of memory locations reserved for the array, elements of second column occupies the next set of memory and so on. Consider a Two Dimensional Array consist of N rows and M columns. It can be stored sequentially in memory column by column as shown below:

Column 0	**A[0,0]**	**A[1,0]**		**A[N-1,0]**
Column 1	A[0,1]	A[1,1]		A[N-1,1]
Column N-1	A[0,M-1]	A[1,M-1]		A[N-1,M-1]

Column Major Order

- Example:
  Consider following example in which a two dimensional array consist of two rows and four columns is stored sequentially in Column Major Order as:

2000	A[0][0]	Column 0
2002	A[1][0]	
2004	A[0][1]	Column 1
2006	A[1][1]	
2008	A[0][2]	Column 2
2010	A[1][2]	
2012	A[0][3]	Column 3
2014	A[1][3]	

Column Major Order

- The Location of element A[i, j] can be obtained by evaluating expression:
  - **LOC (A [i, j]) = Base_Address + W [N (j-L2) + (i-L1)]**
    - **Here,**
      - **Base_Address** is the address of first element in the array.
      - **W** is the word size. It means number of bytes occupied by each element.
      - **N** is number of rows in array.
      - **M** is number of columns in array.
      - **L1** is lower bound of row.
      - **L2** is lower bound of column.

Column Major Order

- Suppose we want to calculate the address of element A [1, 2].
  - It can be calculated as follow:
    - **Here,**

      Base_Address = 2000, W= 2, M=4, N=2, i=1, j=2

      LOC (A [i, j])=Base_Address + W [N (j-L2) + (i-L1)]

      LOC (A[1, 2])=2000 + 2 *[2*(2) + 1]

      =2000 + 2 * [4 + 1]

      =2000 + 2 * 5

      =2000 + 10

      =2010

Column Major Order

1. Given an array **arr[1………10][1………15]** with a base value of **100** and the size of each element is **1 Byte** in memory find the address of arr[8][6] with the help of column-major order.

Column Major Order

1. Each element of 2 D array a[-20:20, 10:25] require 1 byte of storage. The base address of 2 D array is 1000. Calculate the address of element for using column major implementation a[0,15]

Column Major Order

Base address B = 100
Storage size of one element store in any array W = 1 Bytes
Row Subset of an element whose address to be found I = 8
Column Subset of an element whose address to be found J = 6
Lower Limit of row/start row index of matrix LR = 1
Lower Limit of column/start column index of matrix = 1
Number of Rows given in the matrix M = Upper Bound – Lower Bound + 1
= 10 – 1 + 1
= 10

Column Major Order

Formula: used

*Address of A[I][J] = B + W * ((J – LC) * M + (I – LR))*

*Address of A[8][6] = 100 + 1 * ((6 – 1) * 10 + (8 – 1))*

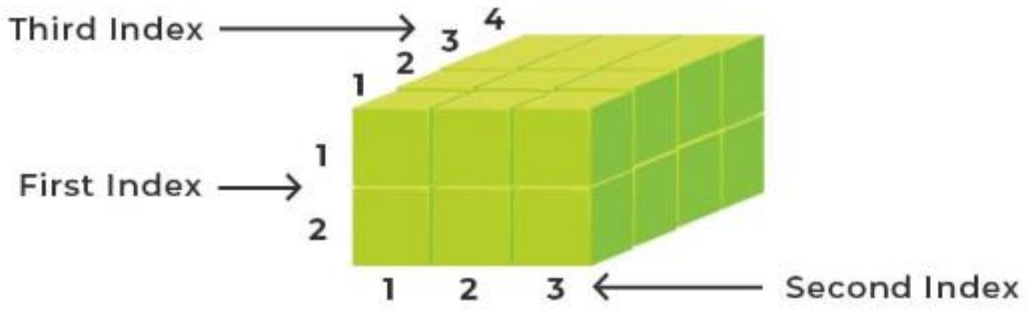
*= 100 + 1 * ((5) * 10 + (7))*

*= 100 + 1 * (57)*

Address of A[I][J] = 157

Address of any element in the 3-D Array:

- A **3-Dimensional** array is a collection of 2-Dimensional arrays. It is specified by using three subscripts:
  - Block size
  - Row size
  - Column size



Three-Dimensional Array
with 24 Elements

Row Major Order with 3 D Array:

1. Given an array, **arr[1:9, -4:1, 5:10]** with a base value of **400** and the size of each element is **2 Bytes** in memory find the address of element **arr[5][-1][8]** with the help of row-major order?

Row Major Order with 3 D Array:

Address of A[i][j][k] = B + W *(M * N * (i-x) + N *(j-y) + (k-z))
Here:
B = Base Address (start address)
W = Weight (storage size of one element stored in the array)
M = Row (total number of rows)
N = Column (total number of columns)
P = Width (total number of cells depth-wise)
x = Lower Bound of Row
y = Lower Bound of Column
z = Lower Bound of Width

Row Major Order with 3 D Array:

Solution:
Given:
Block Subset of an element whose address to be found I = 5
Row Subset of an element whose address to be found J = -1
Column Subset of an element whose address to be found K = 8
Base address B = 400
Storage size of one element store in any array(in Byte) W = 2
Lower Limit of blocks in matrix x = 1
Lower Limit of row/start row index of matrix y = -4
Lower Limit of column/start column index of matrix z = 5
M(row) = Upper Bound – Lower Bound + 1 = 1 – (-4) + 1 = 6
N(Column)= Upper Bound – Lower Bound + 1 = 10 – 5 + 1 = 6
Formula used:
Address of[I][J][K] =B + W (M * N(i-x) + N *(j-y) + (k-z))
Solution:
Address of arr[5][-1][8] = 400 + 2 * {[6 * 6 * (5 – 1)] + 6 * [(-1 + 4)]} + [8 – 5]
= 400 + 2 * (6*6*4)+(6*3)+3
= 400 + 2 * (165)

Column Major Order with 3 D Array:

Address of A[i][j][k]= B + W(M * N(i – x) + M *(k – z) + (j – y))

Here:

B = Base Address (start address)

W = Weight (storage size of one element stored in the array)

M = Row (total number of rows)

N = Column (total number of columns)

P = Width (total number of cells depth-wise)

x = Lower Bound of block (first subscipt)

y = Lower Bound of Row

z = Lower Bound of Column

Column Major Order with 3 D Array:

1. Given an array **arr[1:8, -5:5, -10:5]** with a base value of **400** and the size of each element is **4 Bytes** in memory find the address of element **arr[3][3][3]** with the help of column-major order?

Column Major Order with 3 D Array:

Solution:
Given:
Row Subset of an element whose address to be found I = 3
Column Subset of an element whose address to be found J = 3
Block Subset of an element whose address to be found K = 3
Base address B = 400
Storage size of one element store in any array(in Byte) W = 4
Lower Limit of blocks in matrix x = 1
Lower Limit of row/start row index of matrix y = -5
Lower Limit of column/start column index of matrix z = -10
M (row)= Upper Bound – Lower Bound + 1 = 5 +5 + 1 = 11
N (column)= Upper Bound – Lower Bound + 1 = 5 + 10 + 1 = 16
Formula used:
Address of[i][j][k] = B + W(M * N(i – x) + M * (j-y) + (k – z))
Solution:
Address of arr[3][3][3] = 400 + 4 * ((11*16*(3-1)+11*(3-(-5)+(3-(-10)))
= 400 + 4 * ((176*2 + 11*8 + 13)
= 400 + 4 * (453)

Application of arrays (CO1)

- Arrays are used to Store List of values
- Arrays are used to Perform Matrix Operations
- Arrays are used to implement Search Algorithms
  - Linear search
  - Binary search
- Arrays are used to implement Sorting Algorithms
  - Insertion sort
  - Selection sort
  - Quick sort
- Arrays are used to implement Data Structures
  - Stack using array
  - Queue using array
- Arrays are also used to implement CPU Scheduling Algorithms

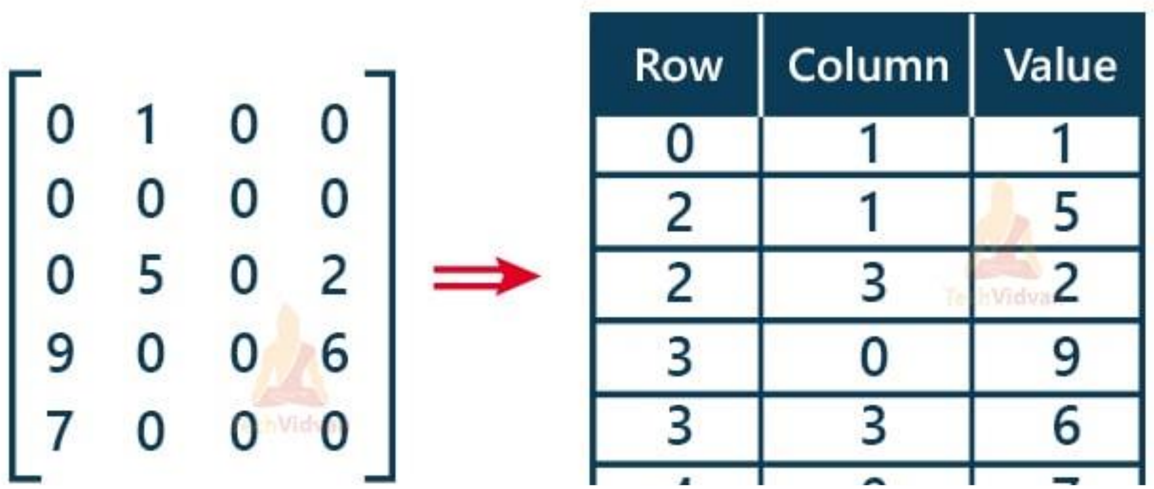
Sparse Matrix (CO1)

- A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have 0 value, then it is called a sparse matrix.
- **Why to use Sparse Matrix instead of simple matrix ?**
  - **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
  - **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..
  - Example

$$\begin{array}{ccccc} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{array}$$

Sparse Matrix Representation

- **Method 1: Using Arrays**
  - 2D array is used to represent a sparse matrix in which there are three rows named as
  - **Row:** Index of row, where non-zero element is located
  - **Column:** Index of column, where non-zero element is located
  - **Value:** Value of the non zero element located at index – (row, column)



$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 2 \\ 9 & 0 & 0 & 6 \\ 7 & 0 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	Column	Value
0	1	1
2	1	5
2	3	2
3	0	9
3	3	6

Sparse Matrix Representation

- Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

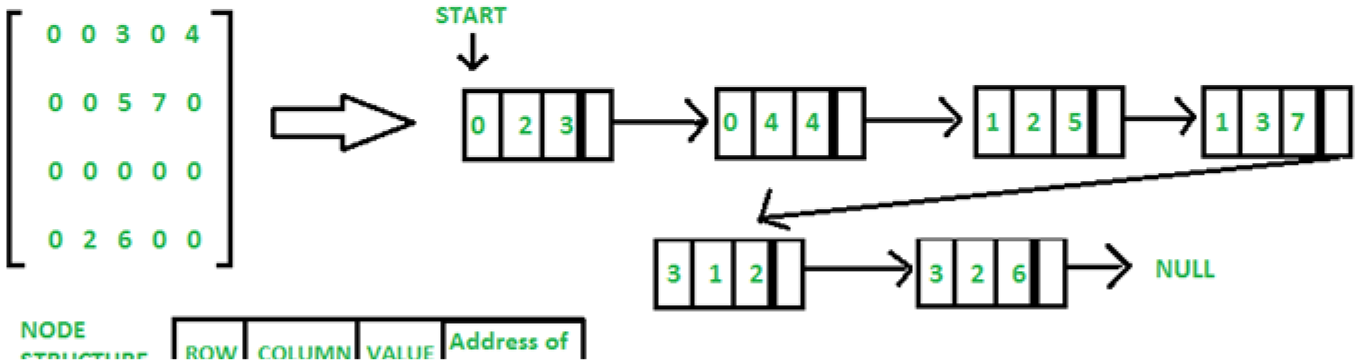
- Sparse Matrix Representations can be done in many ways following are two common representations:
  - Array representation
  - Linked list representation

Basic Terminology(CO1)

- **Linked List**
- **Doubly Linked List**
- **Circularly Linked List**
- **Circularly Doubly Linked List**

Sparse Matrix Representation

- **Method 2: Using Linked Lists**
  - In linked list, each node has four fields. These four fields are defined as:
  - **Row:** Index of row, where non-zero element is located
  - **Column:** Index of column, where non-zero element is located
  - **Value:** Value of the non zero element located at index – (row,column)
  - **Next node:** Address of the next node



Unit Content

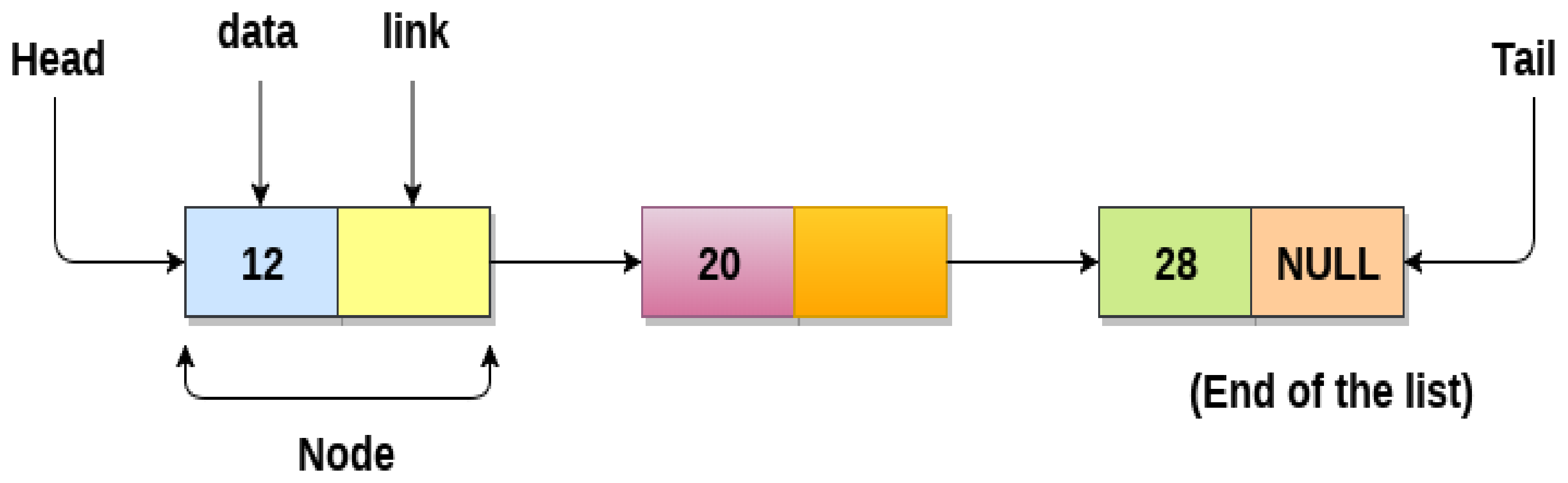
- Advantages of Linked List over Array
- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Operation on Linked List
  - Insertion
  - Deletion
  - Traversal
  - Reversal
  - Searching Polynomial Representation
  - Addition, Subtraction and Multiplication of Polynomials
- Implementation of Stack and Queue using Linked List

Unit -1 Syllabus

- Advantages of linked list over array,
- Self-referential structure,
- Singly Linked List, Doubly Linked List, Circular Linked List.
- **Operations on a Linked List:** Insertion, Deletion, Traversal, Reversal, Searching, Polynomial Representation and Addition of Polynomials.
- Implementation of Stack and Queue using Linked lists.

Linked List

- Linked List can be defined as collection of objects called nodes that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



Introduction to Linked List

Linked List

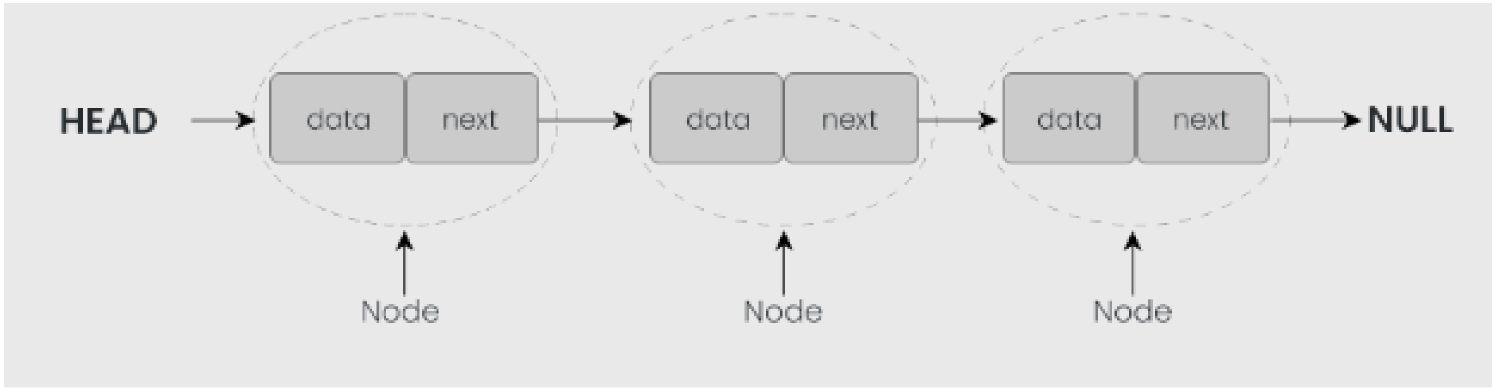
- The elements of a linked list are not stored in adjacent memory locations as in arrays.
- It is a linear collection of data elements, called nodes, where the linear order is implemented by means of pointers.

Continued….

Linked List

In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

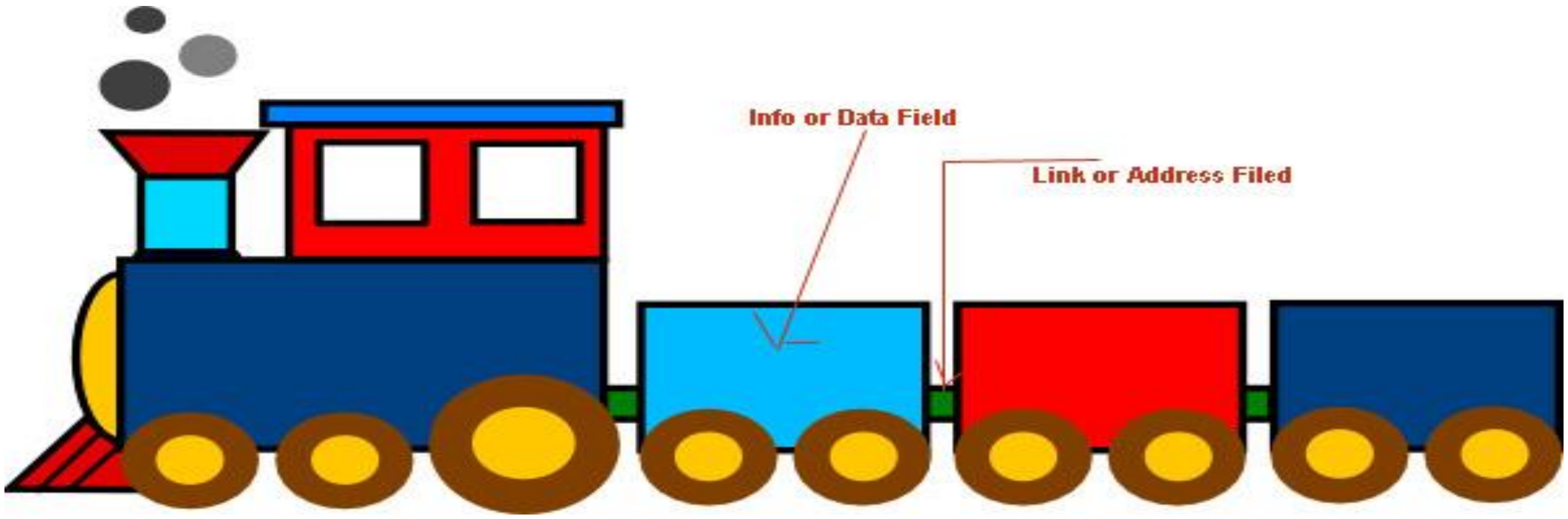
- *A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:*



- A node in this type of linked list contains two types of fields
  - data: which holds a list element
  - next: which stores a link (i.e. pointer) to the next node in the list.

Linked List

- A linked list is a linear data structure.
- Nodes make up linked lists.
- Nodes are structures made up of data and a pointer to another node.
- Usually the pointer is called next.



Continued….

Linked List

Node Structure: A node in a linked list typically consists of two components:
Data: It holds the actual value or data associated with the node.
Next Pointer: It stores the memory address (reference) of the next node in the sequence.
Head and Tail: The linked list is accessed through the head node, which points to the first node in the list. The last node in the list points to NULL or nullptr, indicating the end of the list. This node is known as the tail node.

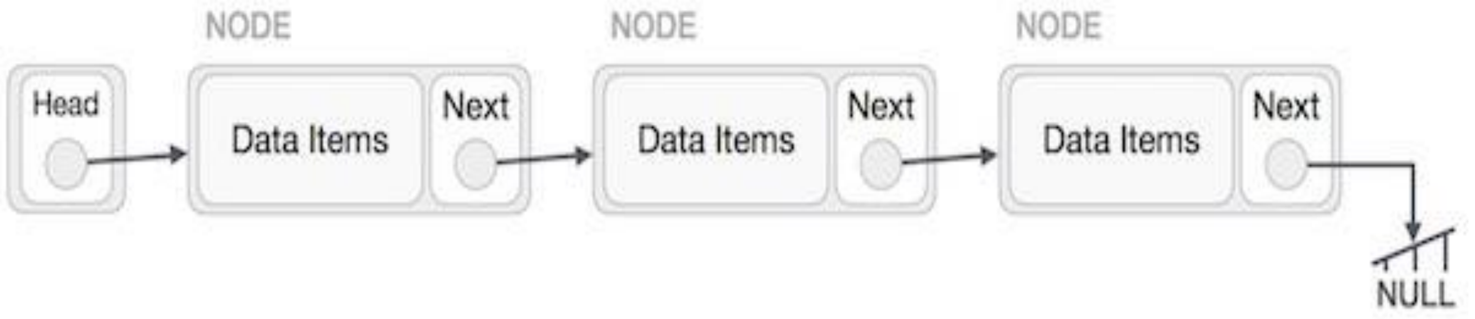
Continued….

Why linked list data structure needed?

Dynamic Data structure: The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.
Ease of Insertion/Deletion: The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
Efficient Memory Utilization: As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
Implementation: Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

Linked List Representation

- Linked list can be visualized as a chain of nodes, where every node points to the next node.



- As per the above illustration, following are the important points to be considered.
  - Linked List contains a link element called first.
  - Each link carries a data field(s) and a link field called next.
  - Each link is linked with its next link using its next link.
  - Last link carries a link as null to mark the end of the list.

Properties of linked list

- The nodes in a linked list are not stored contiguously in the memory
- You don’t have to shift any element in the list
- Memory for each node can be allocated dynamically whenever the need arises.
- The size of a linked list can grow or shrink dynamically

Basic Operations on Linked List

- Following are the basic operations supported by a list.
    - **Insertion** − Adds an element at the beginning of the list.
    - **Deletion** − Deletes an element at the beginning of the list.
    - **Display** − Displays the complete list.
    - **Search** − Searches an element using the given key.
    - **Delete** − Deletes an element using the given key.

Arrays & Linked list

Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

Types of Link List

- Following are the various types of linked list.
  - **Singly Linked List** − Item navigation is forward only.
  - **Doubly Linked List** − Items can be navigated forward and backward.
  - **Circular Linked List** − Last item contains link of the first element as next
  - **Circular Doubly Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous. Items can be navigated forward and backward.

Creating a node of linked list

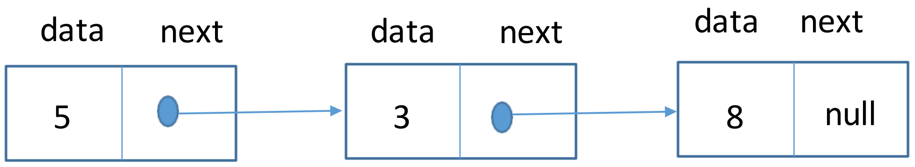
Self Referential Structure is structure which contains a pointer to a structure of the same type.

```
struct abc
{
   int data;
   char c;
   struct abc *next;
 };
```

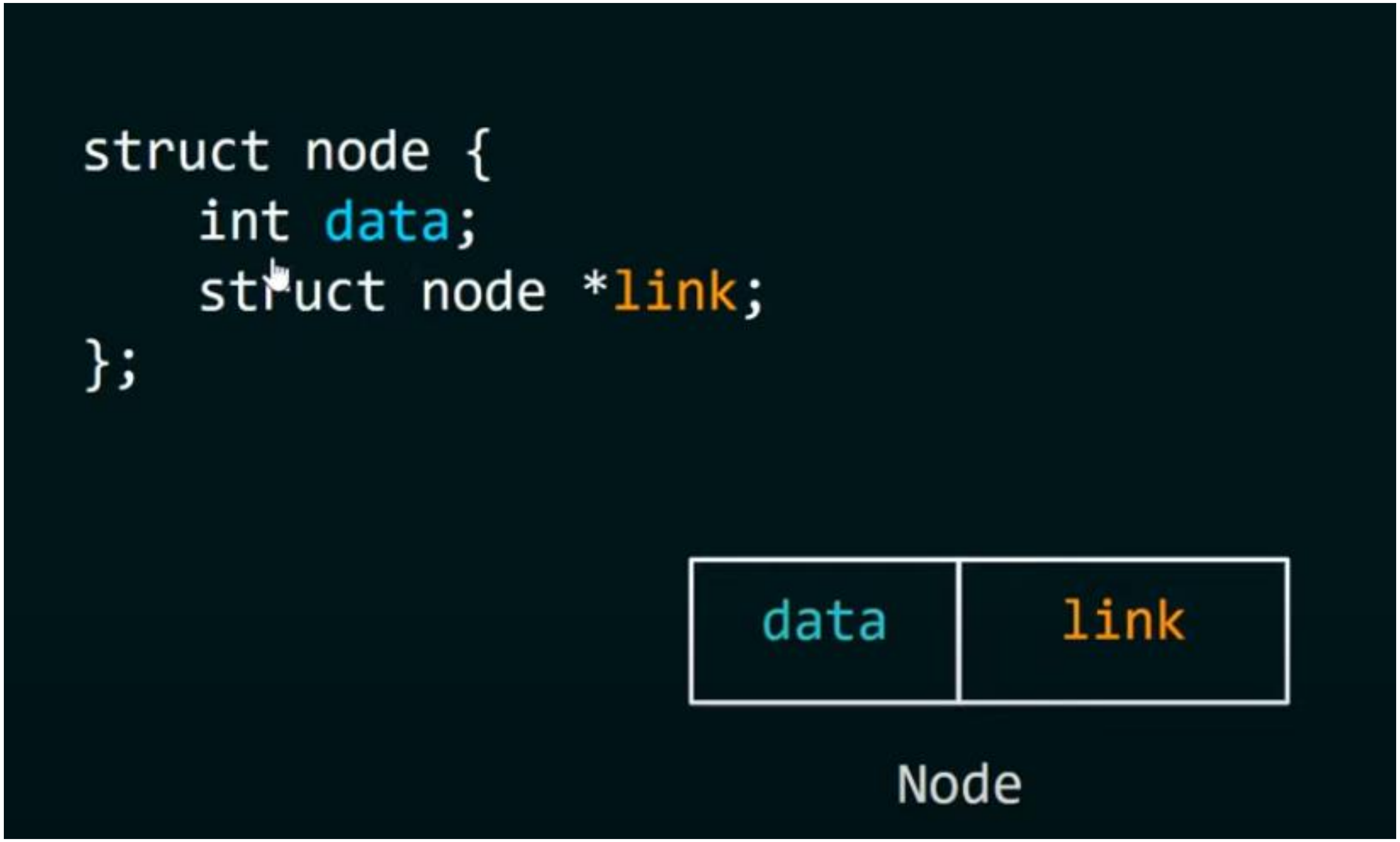
Singly Linked list

- A singly linked list is a dynamic data structure which may grow or shrink, and growing and shrinking depends on the operation made.

- In this type of linked list each node contains two fields one is data field which is used to store the data items and another is next field that is used to point the next node in the list.

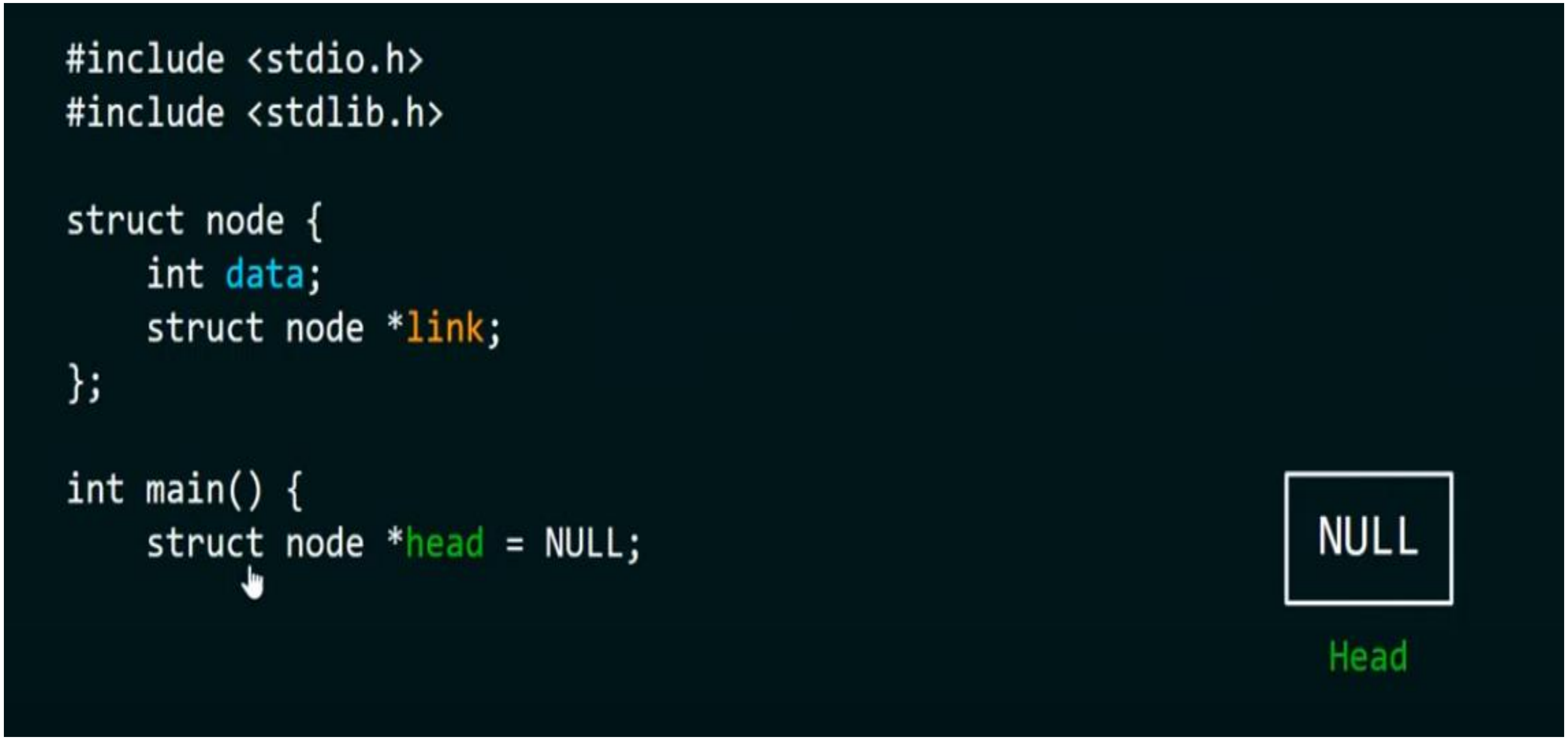


Creating a node of linked list



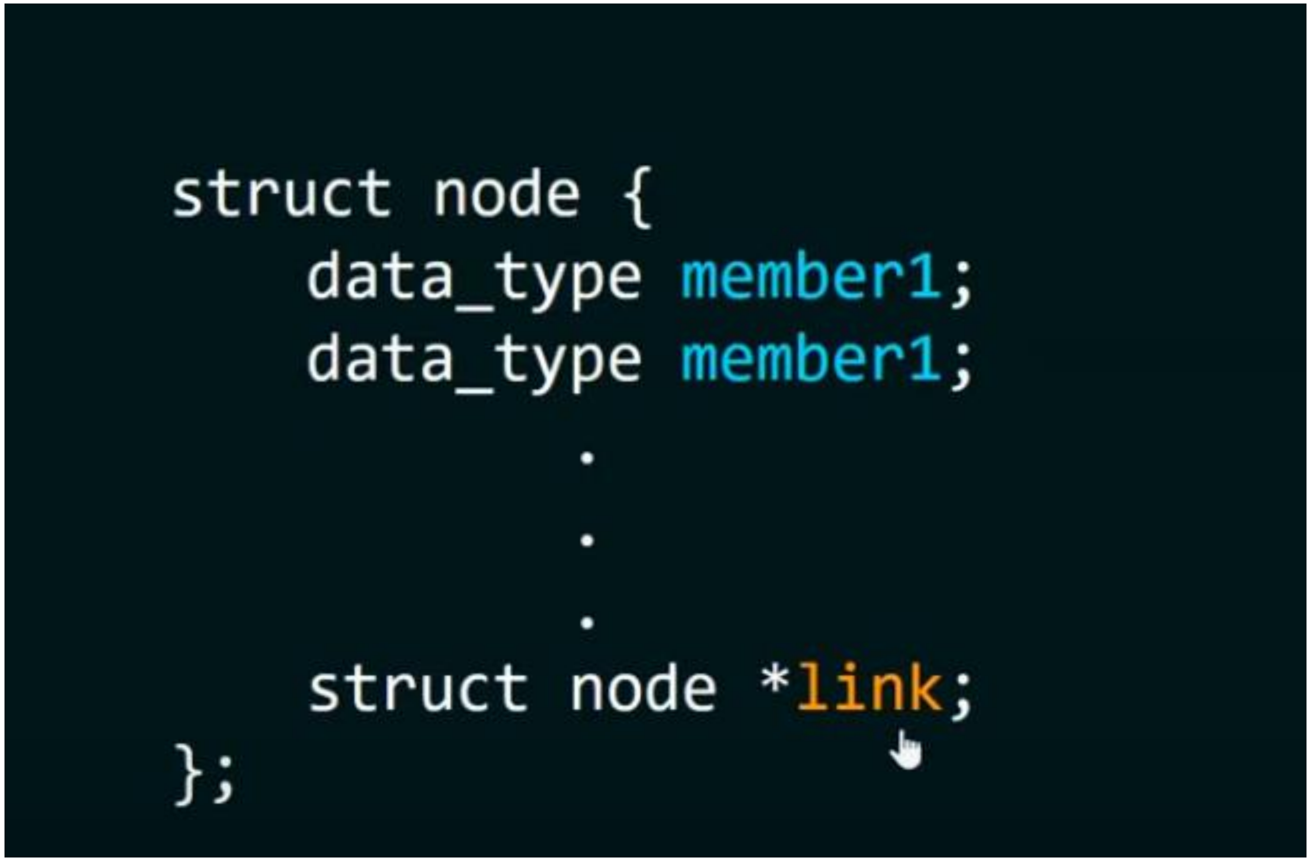
```
struct node {
    int data;
    struct node *link;
};
```

Creating a node of linked list



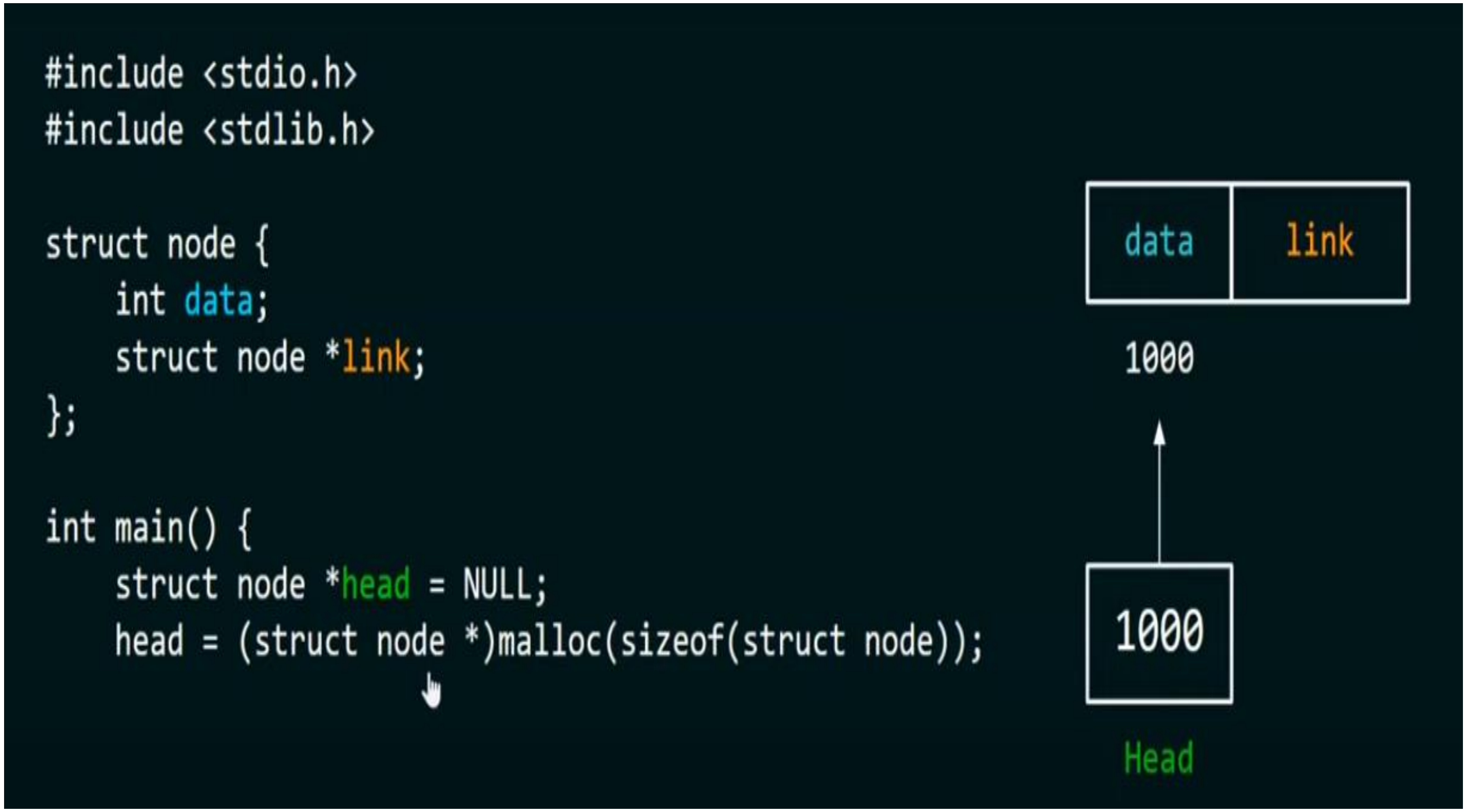
```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *link;
};

int main() {
    struct node *head = NULL;
```

Creating a node of linked list



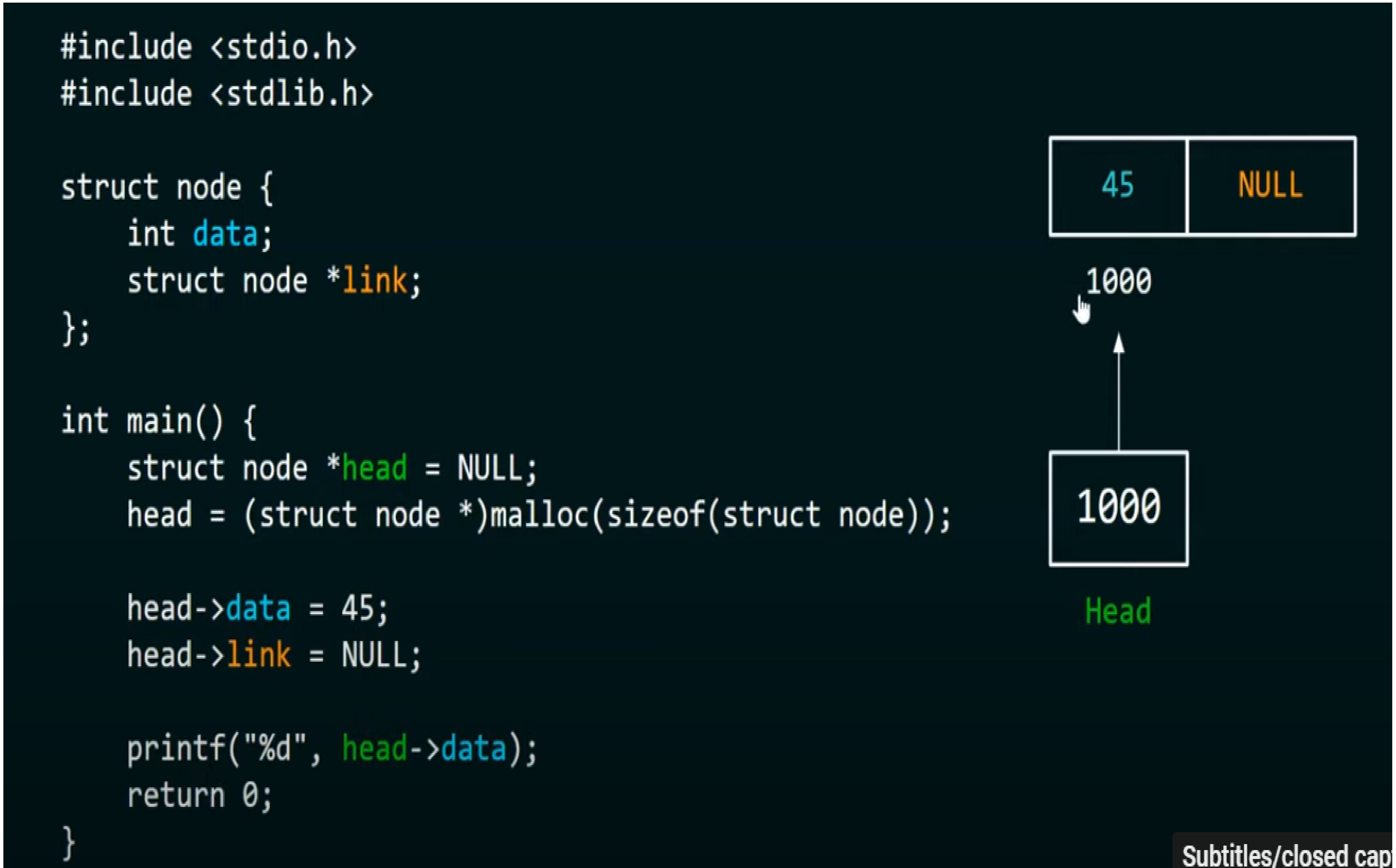
```
struct node {
    data_type member1;
    data_type member1;
              .
              .
              .
    struct node *link;
};
```

Creating a node of linked list



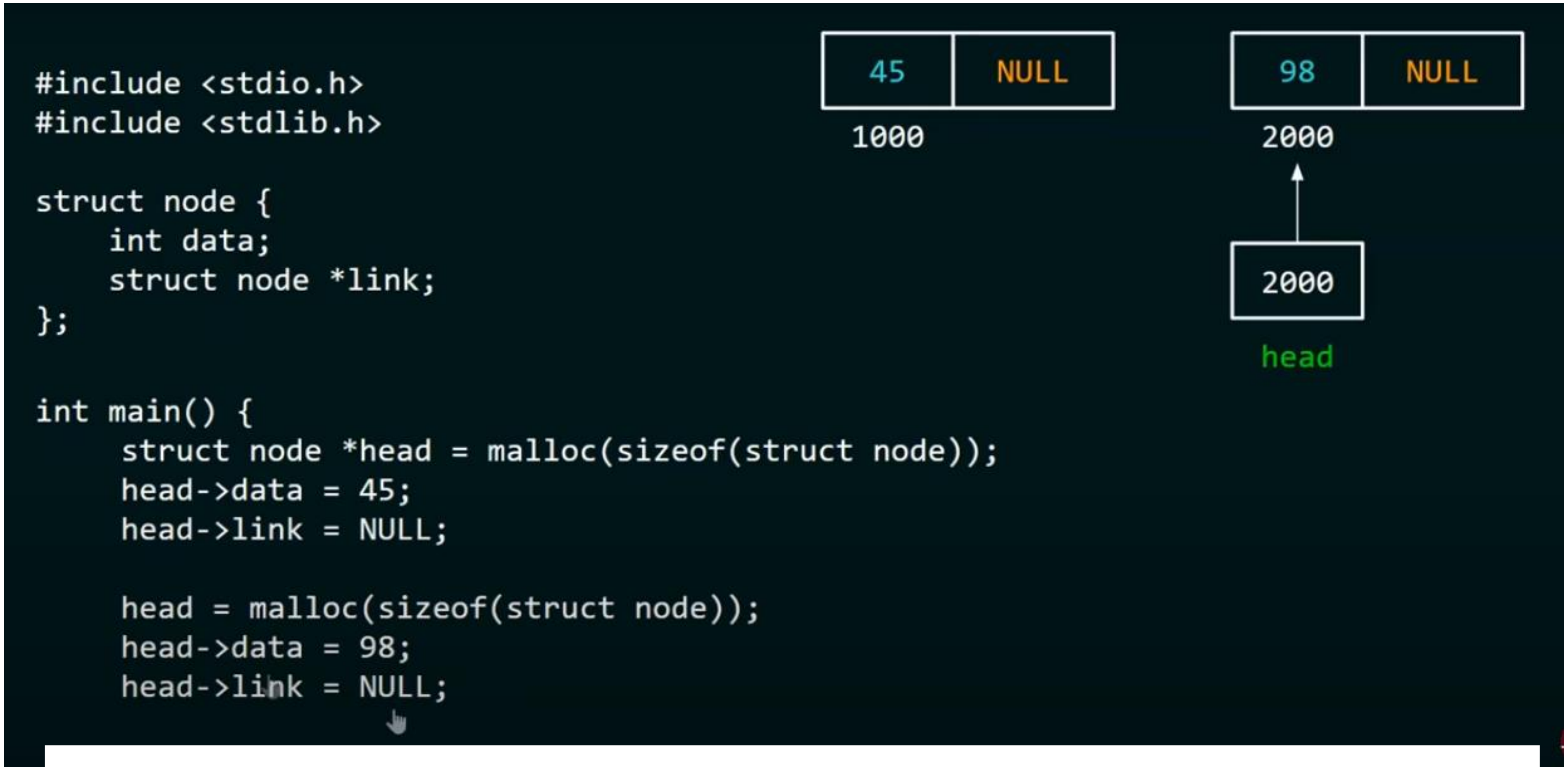
```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *link;
};

int main() {
    struct node *head = NULL;
    head = (struct node *)malloc(sizeof(struct node));
```

Creating a node of linked list



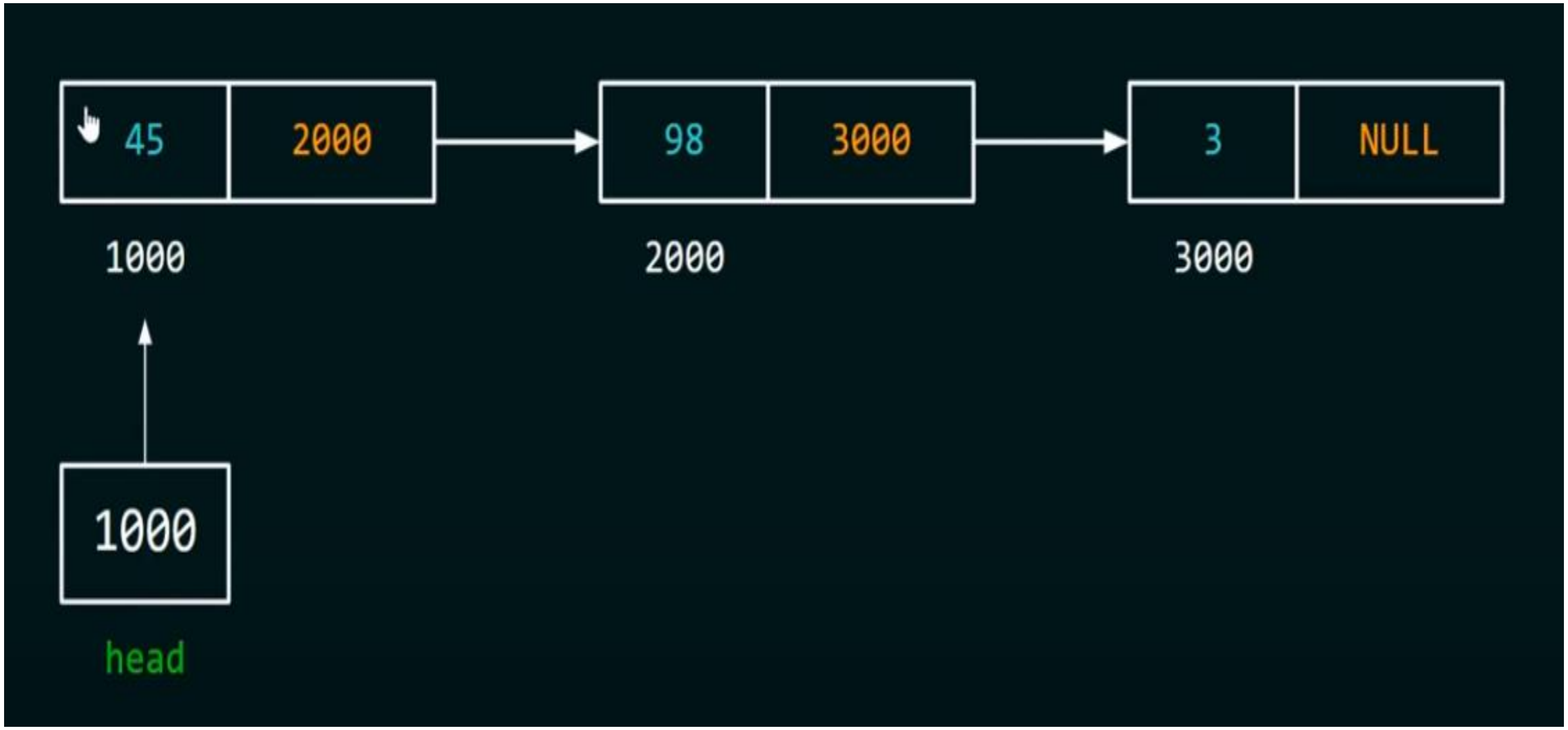
```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *link;
};

int main() {
    struct node *head = NULL;
    head = (struct node *)malloc(sizeof(struct node));

    head->data = 45;
    head->link = NULL;

    printf("%d", head->data);
    return 0;
}
```

Creating single linked list

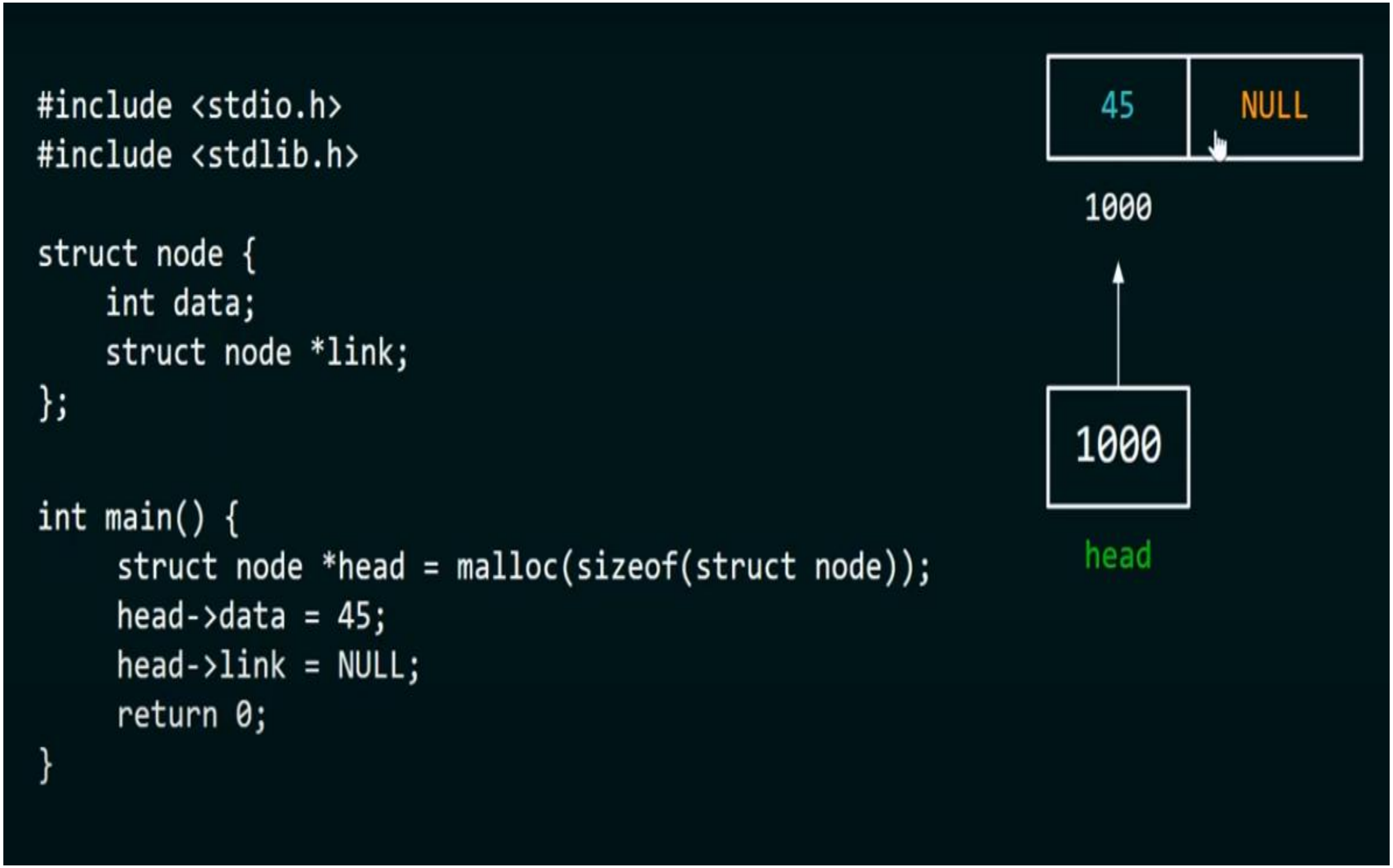


```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *link;
};

int main() {
     struct node *head = malloc(sizeof(struct node));
     head->data = 45;
     head->link = NULL;

     head = malloc(sizeof(struct node));
     head->data = 98;
     head->link = NULL;
```

Creating single linked list

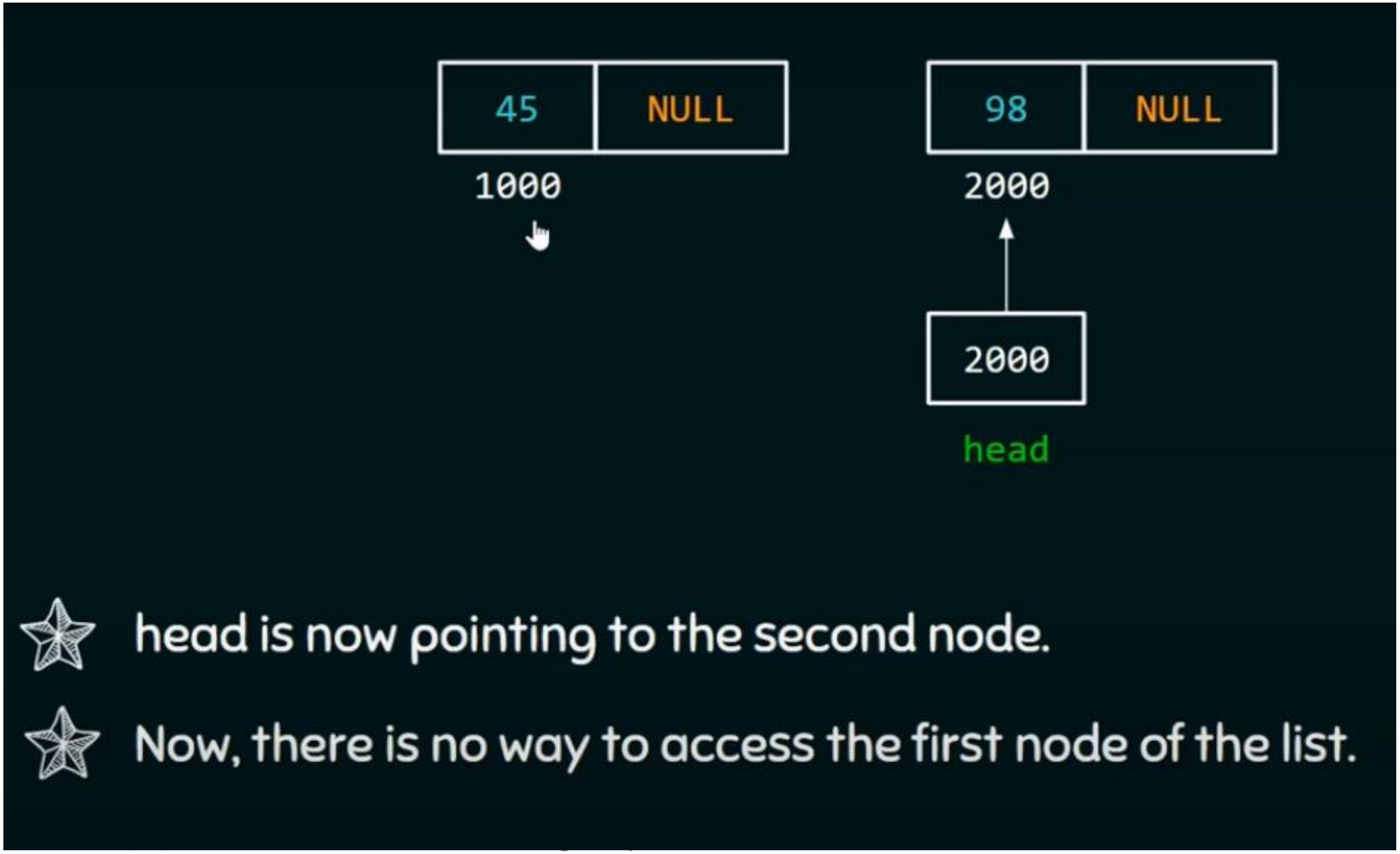


Creating single linked list



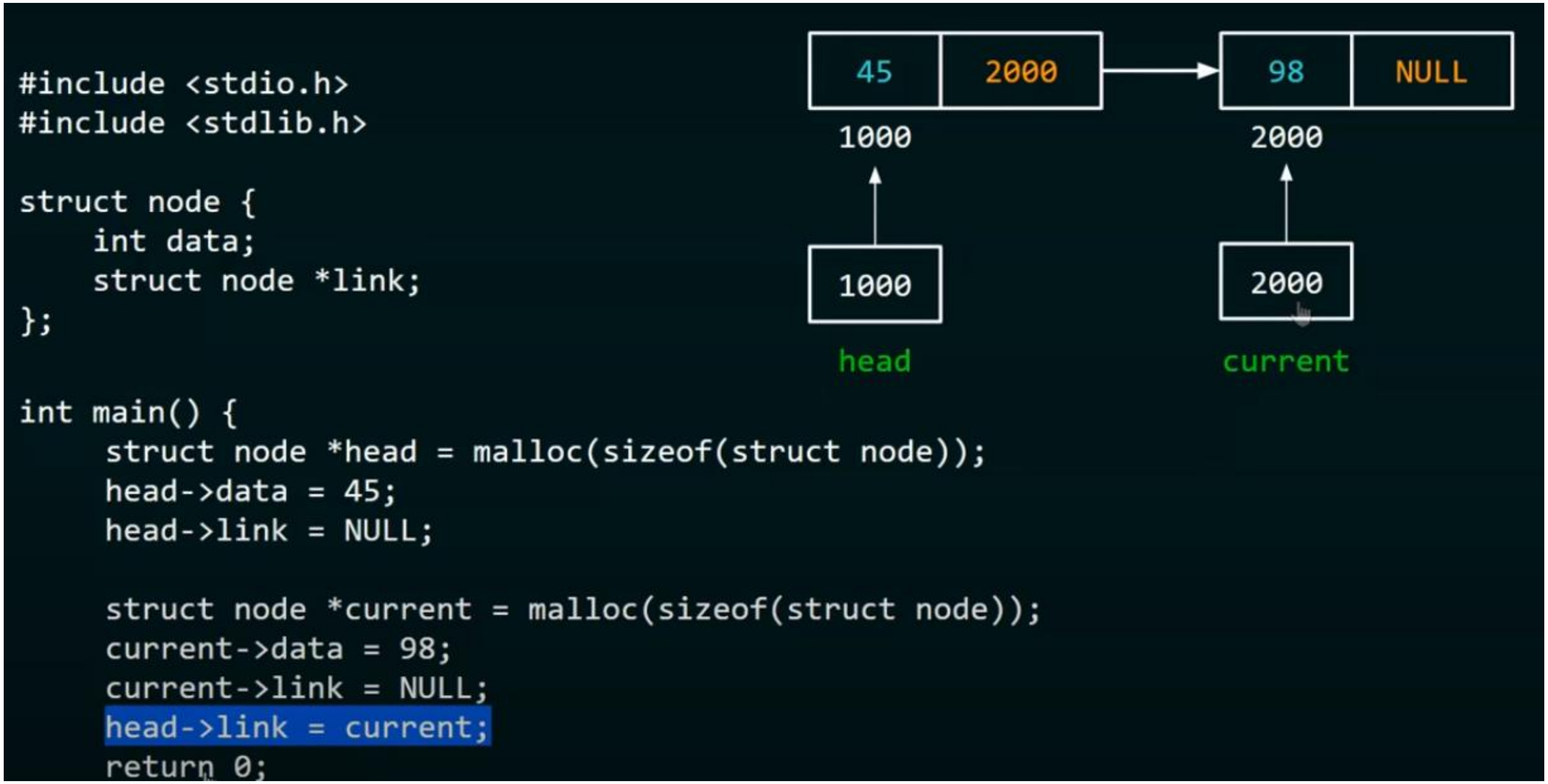
```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *link;
};

int main() {
     struct node *head = malloc(sizeof(struct node));
     head->data = 45;
     head->link = NULL;
     return 0;
}
```

Creating single linked list



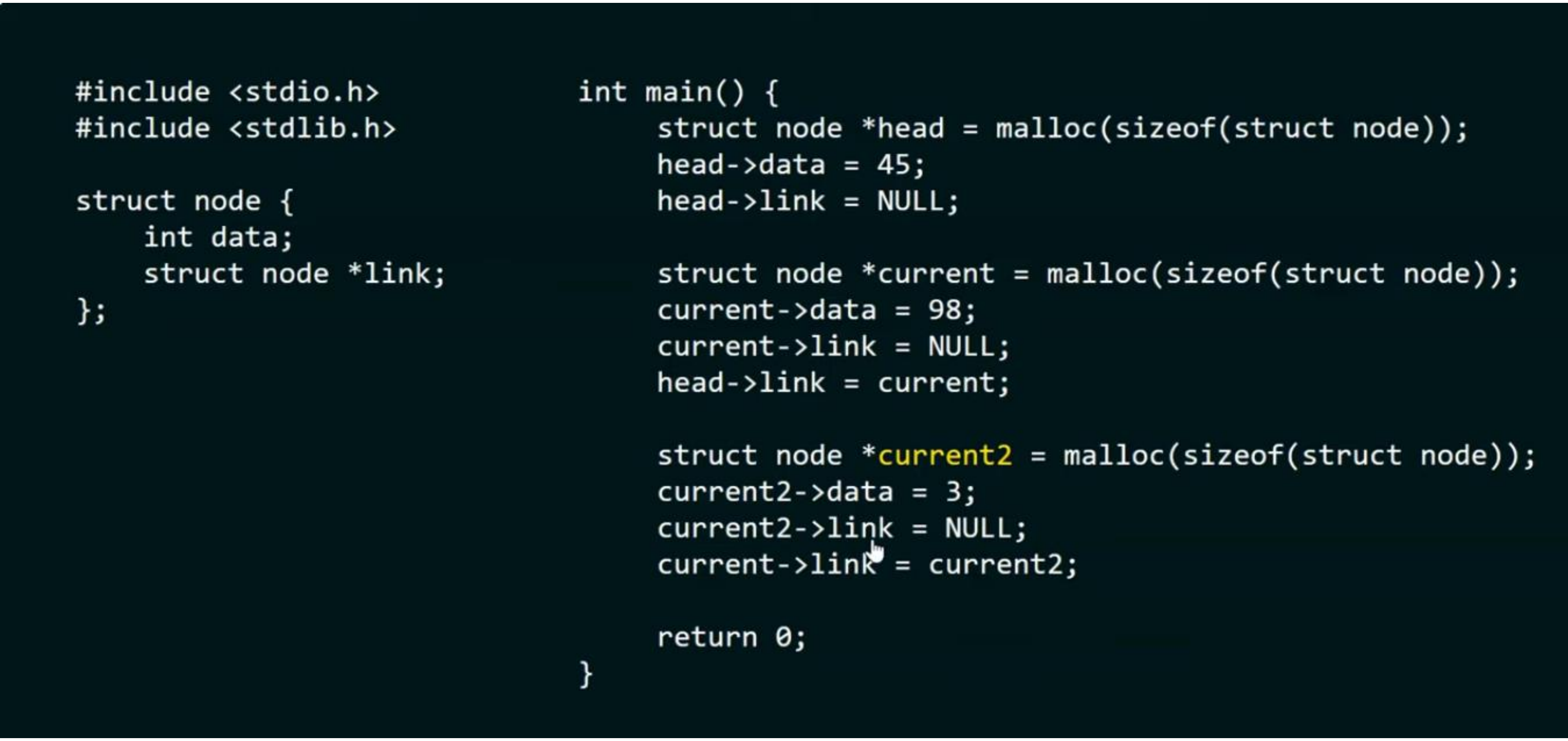
- head is now pointing to the second node.
- Now, there is no way to access the first node of the list.

Creating single linked list



```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *link;
};

int main() {
     struct node *head = malloc(sizeof(struct node));
     head->data = 45;
     head->link = NULL;

     struct node *current = malloc(sizeof(struct node));
     current->data = 98;
     current->link = NULL;
     head->link = current;
     return 0;
```

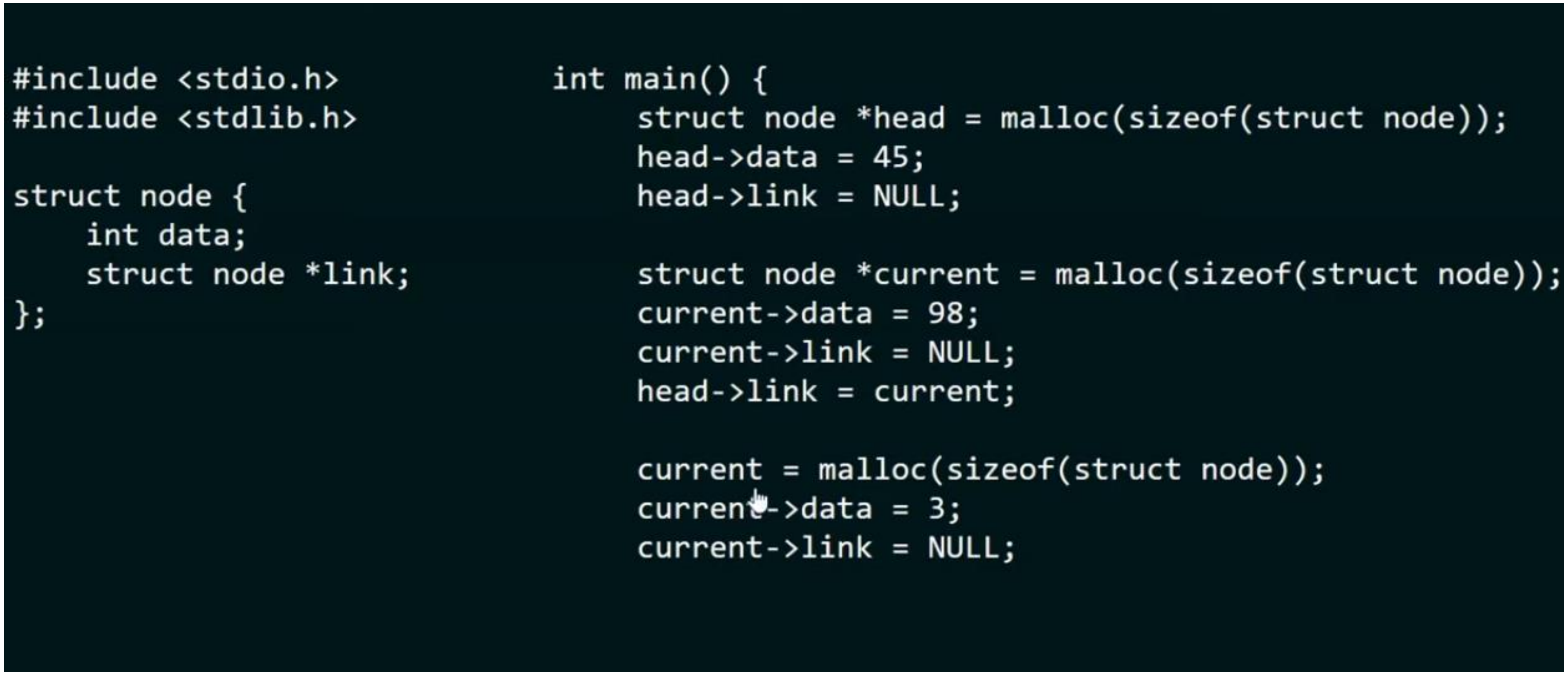
Method: 1 Creating 3rd node



```
#include <stdio.h>
#include <stdlib.h>

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

```
int main() {
    struct node *head = malloc(sizeof(struct node));
    head->data = 45;
    head->link = NULL;

    struct node *current = malloc(sizeof(struct node));
    current->data = 98;
    current->link = NULL;
    head->link = current;

    struct node *current2 = malloc(sizeof(struct node));
    current2->data = 3;
    current2->link = NULL;
    current->link = current2;

    return 0;
}
```

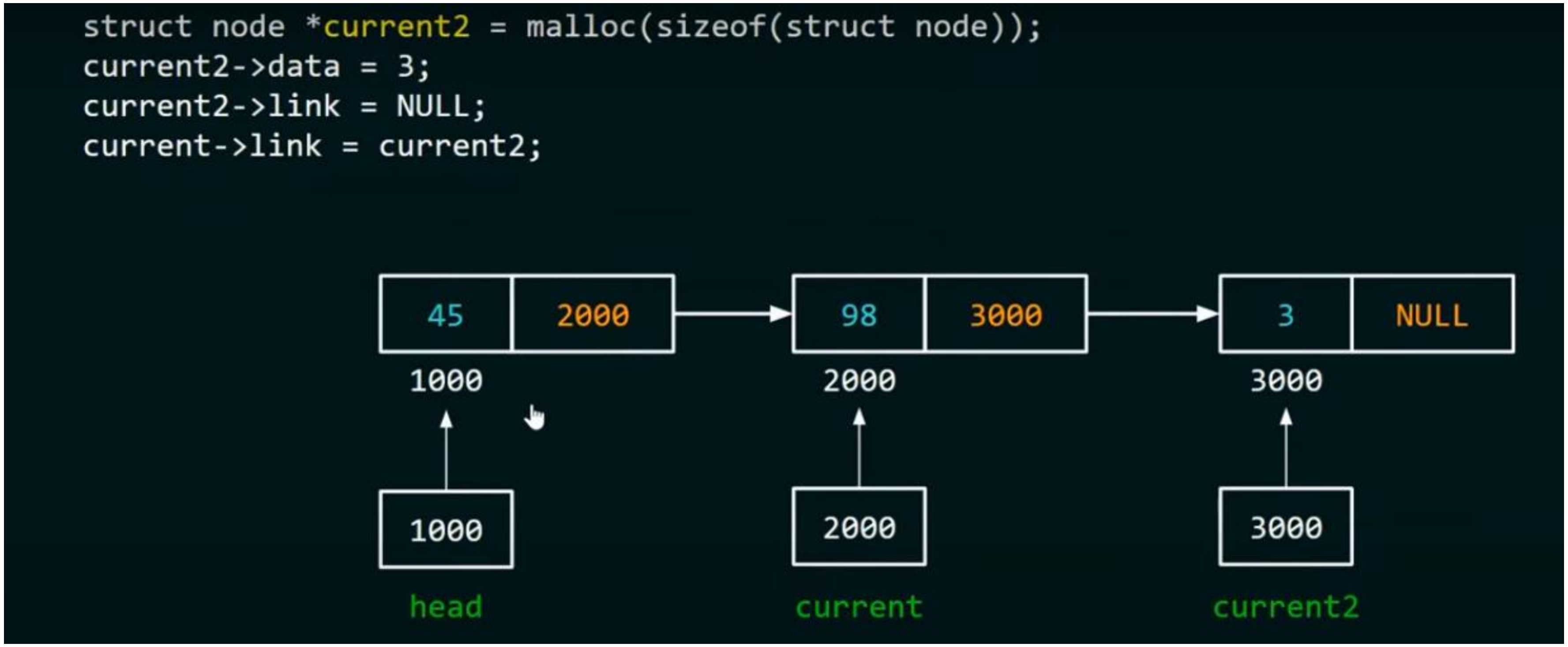
Method: 2 Creating 3rd node



```c
#include <stdio.h>
#include <stdlib.h>

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

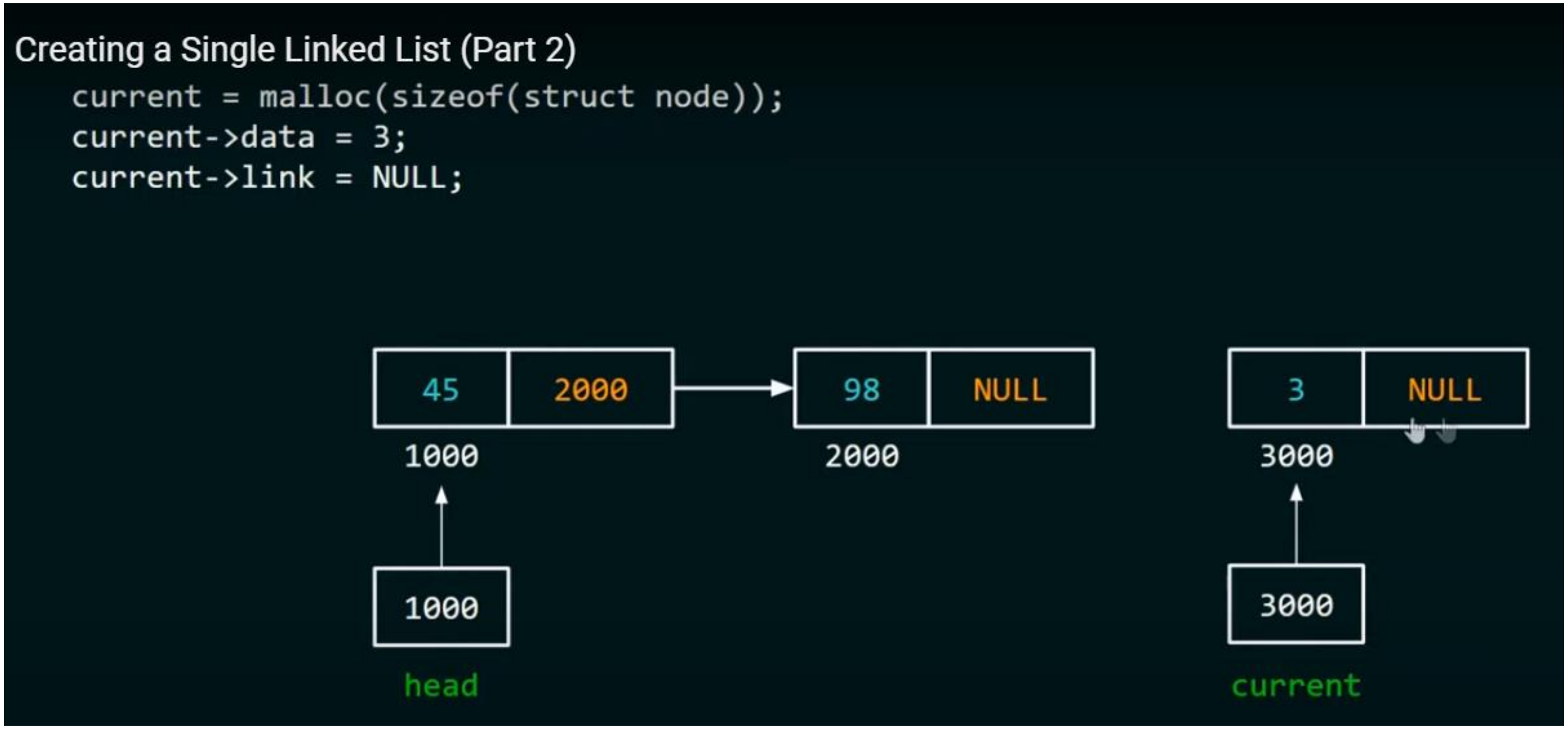
```c
int main() {
    struct node *head = malloc(sizeof(struct node));
    head->data = 45;
    head->link = NULL;

    struct node *current = malloc(sizeof(struct node));
    current->data = 98;
    current->link = NULL;
    head->link = current;

    current = malloc(sizeof(struct node));
    current->data = 3;
    current->link = NULL;
```

Method: 1 Creating 3rd node



```
struct node *current2 = malloc(sizeof(struct node));
current2->data = 3;
current2->link = NULL;
current->link = current2;
```

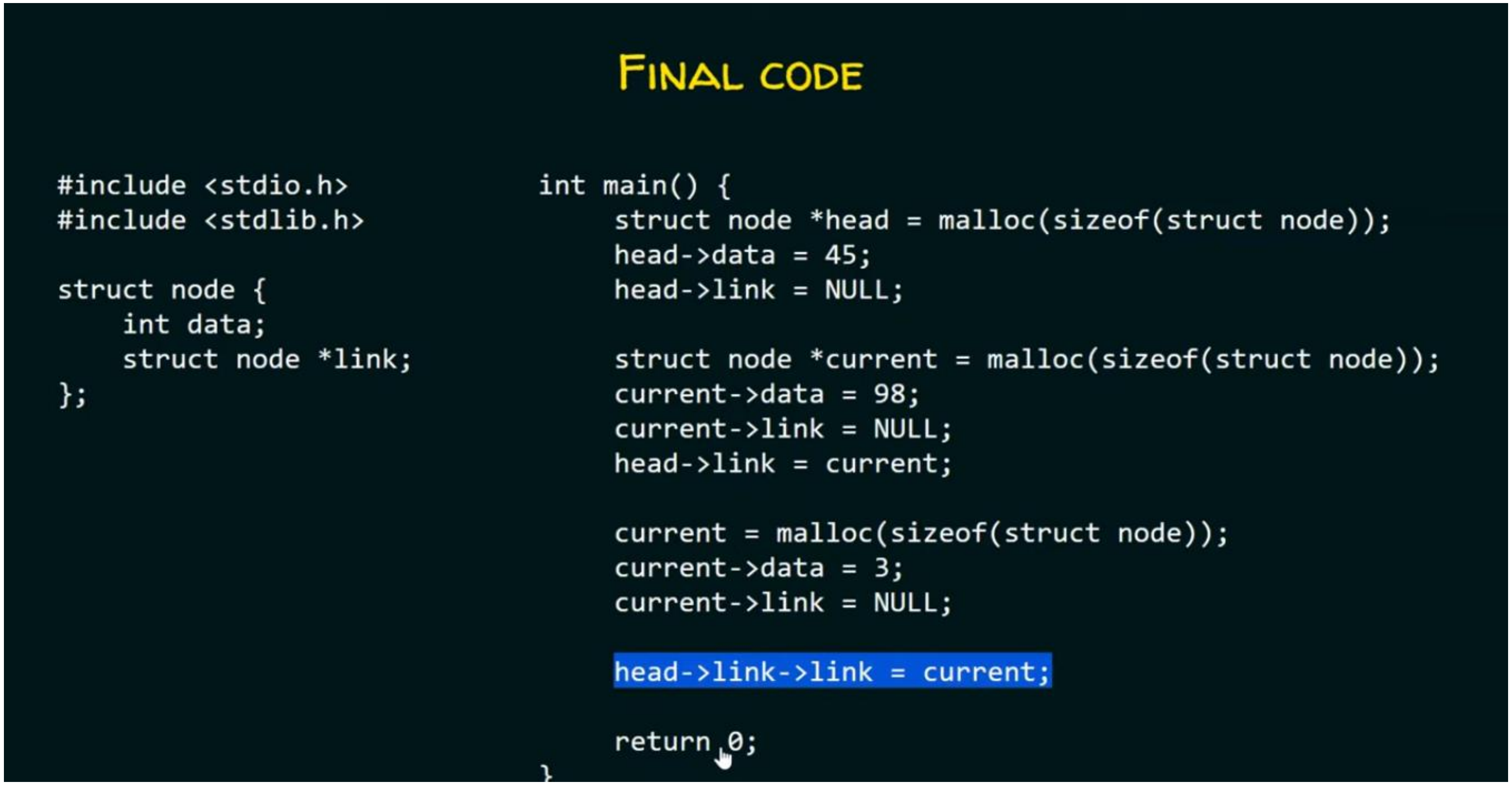
Method: 2 Creating 3rd node



Creating a Single Linked List (Part 2)

```
    current = malloc(sizeof(struct node));
    current->data = 3;
    current->link = NULL;
```

Method: 2 Creating 3rd node

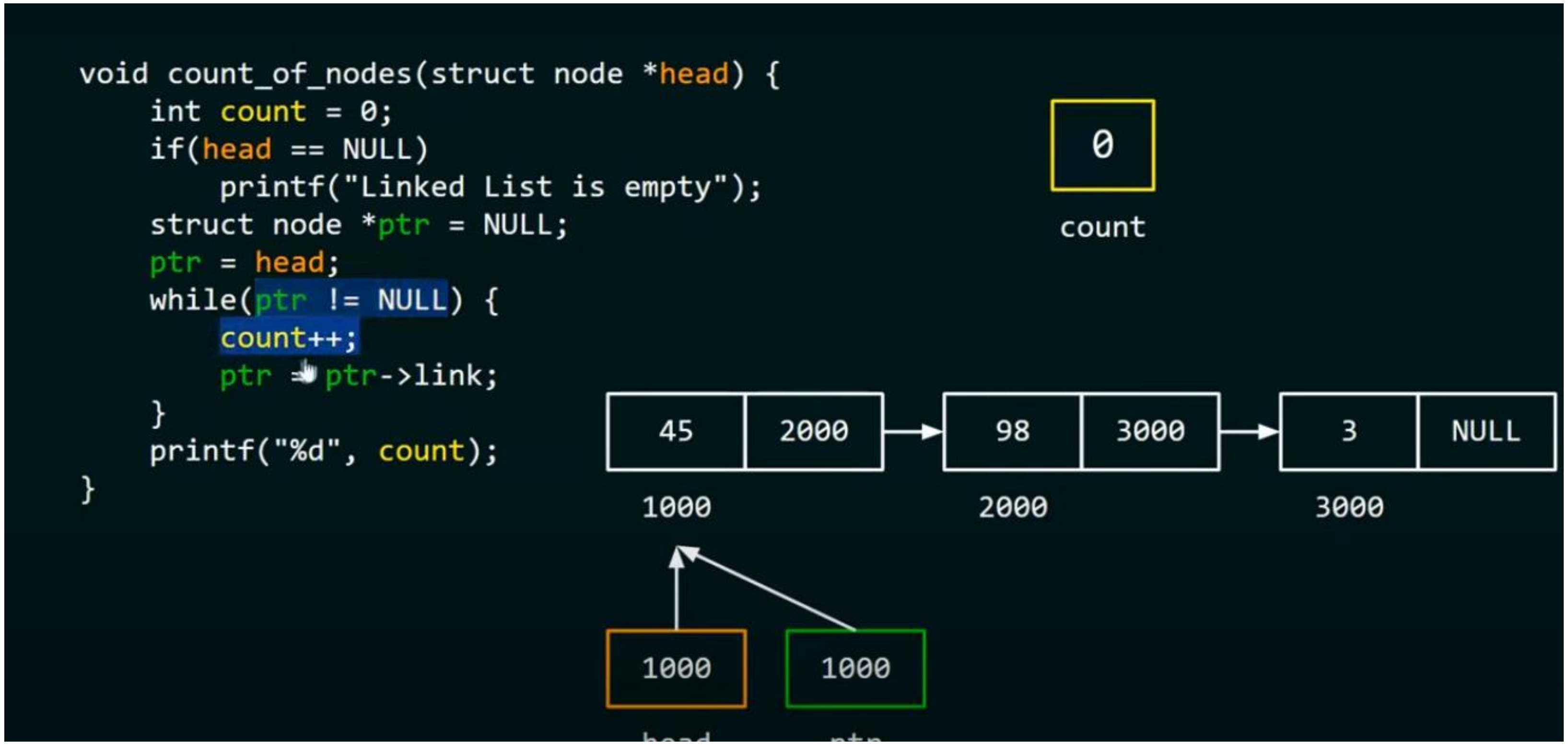


FINAL CODE

```
#include <stdio.h>
#include <stdlib.h>

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

```
int main() {
    struct node *head = malloc(sizeof(struct node));
    head->data = 45;
    head->link = NULL;

    struct node *current = malloc(sizeof(struct node));
    current->data = 98;
    current->link = NULL;
    head->link = current;

    current = malloc(sizeof(struct node));
    current->data = 3;
    current->link = NULL;

    head->link->link = current;

    return 0;
}
```

Linked list Traversal



```
void count_of_nodes(struct node *head) {
    int count = 0;
    if(head == NULL)
        printf("Linked List is empty");
    struct node *ptr = NULL;
    ptr = head;
    while(ptr != NULL) {
        count++;
        ptr = ptr->link;
    }
    printf("%d", count);
}
```

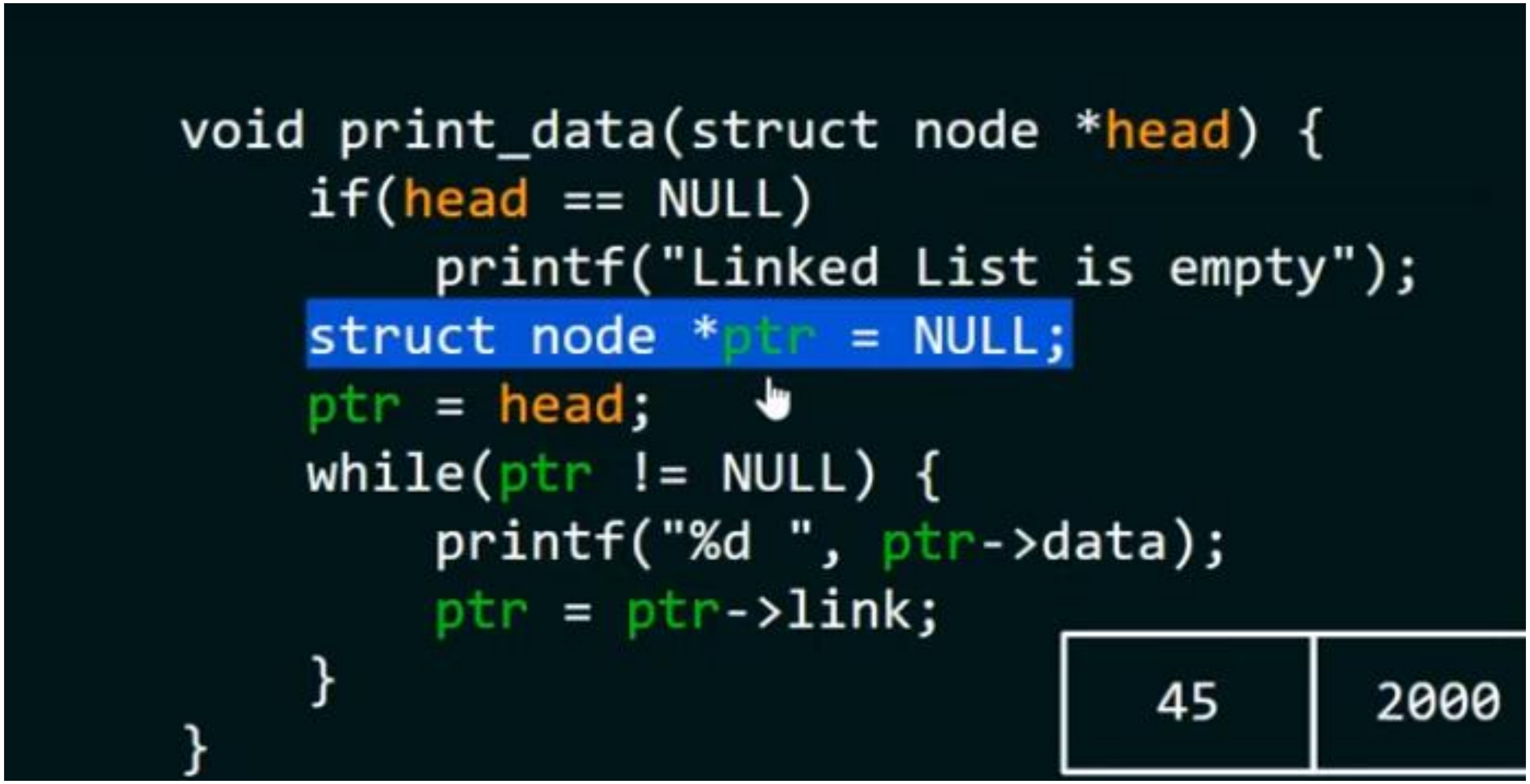
Linked list Traversal Algorithm

- STEP 1: SET PTR = HEAD.
- STEP 2: IF PTR = NULL.
- STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR != NULL.
- STEP 5: PRINT PTR→ DATA.
- STEP 6: PTR = PTR → NEXT.
- STEP 7: EXIT.

Linked list Traversal

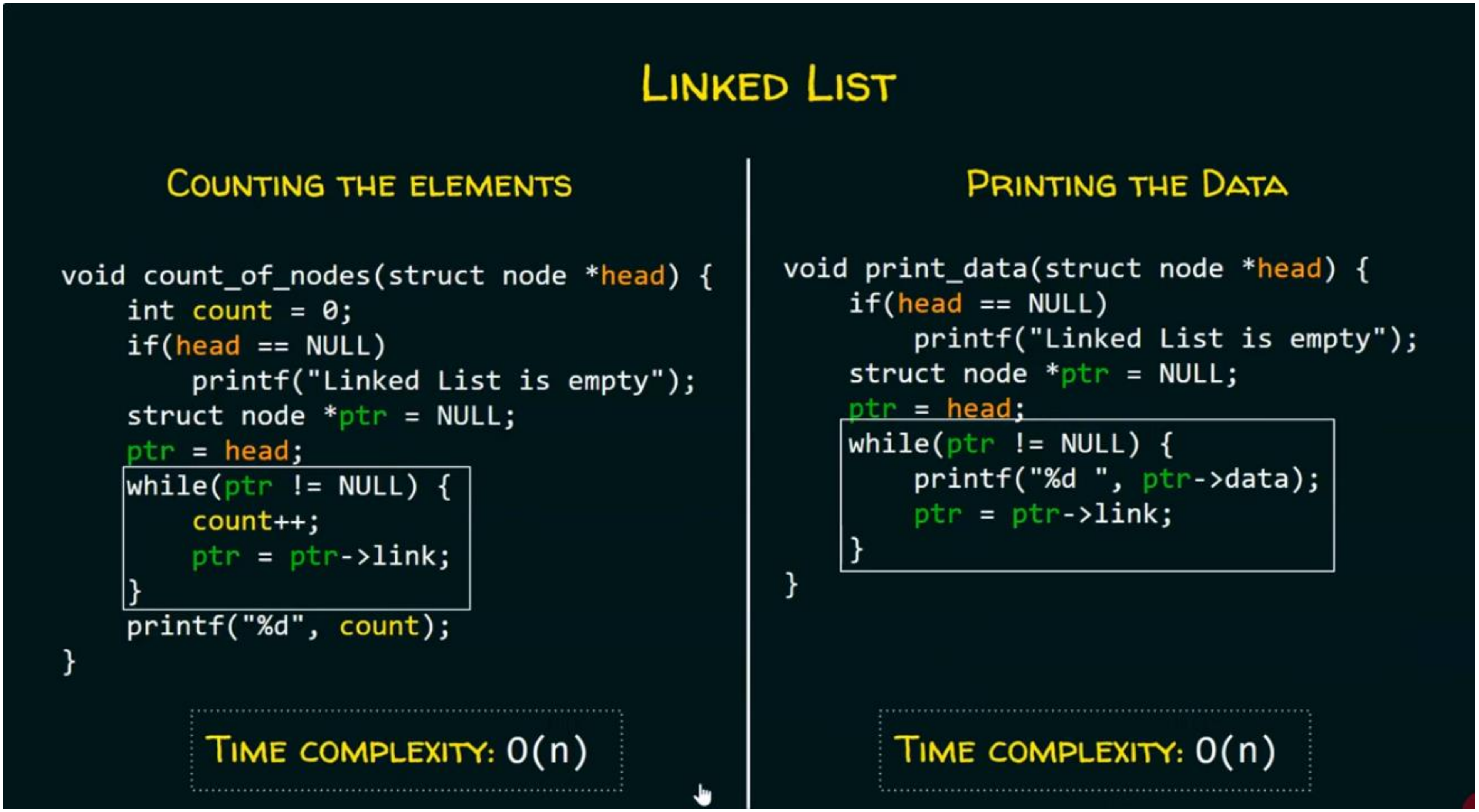
- Write a program to print data of linked list.

Linked list Traversal



```
void print_data(struct node *head) {
    if(head == NULL)
        printf("Linked List is empty");
    struct node *ptr = NULL;
    ptr = head;
    while(ptr != NULL) {
        printf("%d ", ptr->data);
        ptr = ptr->link;
    }
}
```

Time Complexity



LINKED LIST

COUNTING THE ELEMENTS

```
void count_of_nodes(struct node *head) {
    int count = 0;
    if(head == NULL)
        printf("Linked List is empty");
    struct node *ptr = NULL;
    ptr = head;
    while(ptr != NULL) {
        count++;
        ptr = ptr->link;
    }
    printf("%d", count);
}
```

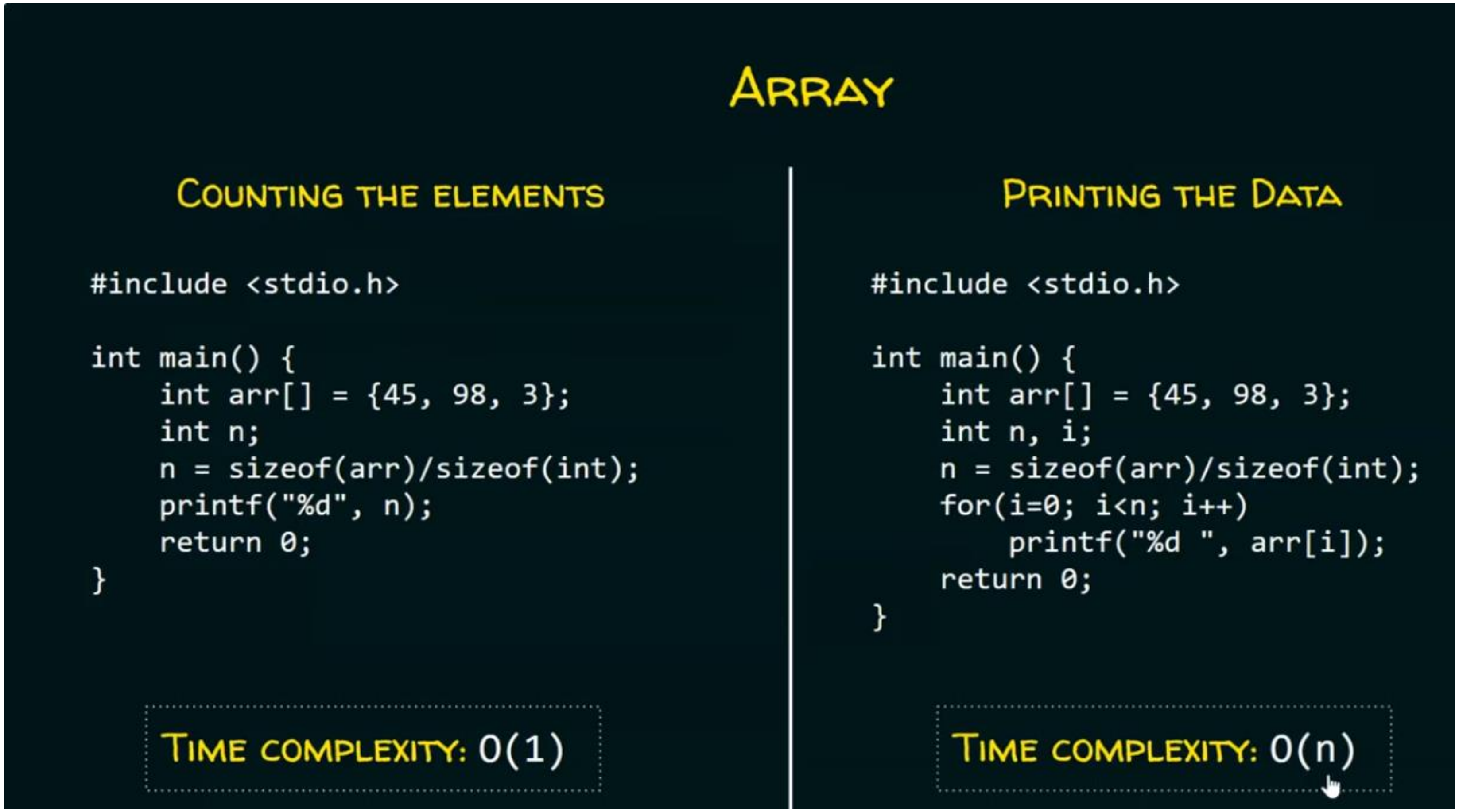
TIME COMPLEXITY: O(n)

PRINTING THE DATA

```
void print_data(struct node *head) {
    if(head == NULL)
        printf("Linked List is empty");
    struct node *ptr = NULL;
    ptr = head;
    while(ptr != NULL) {
        printf("%d ", ptr->data);
        ptr = ptr->link;
    }
}
```

TIME COMPLEXITY: O(n)

Time Complexity



ARRAY

COUNTING THE ELEMENTS

```
#include <stdio.h>

int main() {
    int arr[] = {45, 98, 3};
    int n;
    n = sizeof(arr)/sizeof(int);
    printf("%d", n);
    return 0;
}
```

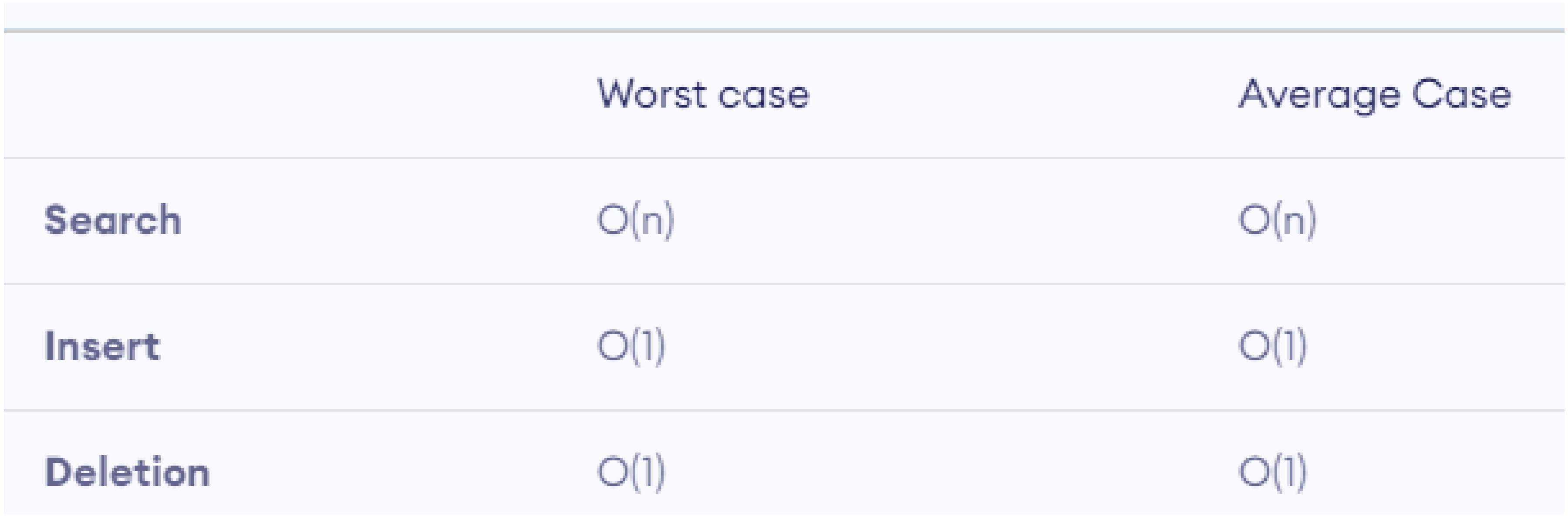
TIME COMPLEXITY: O(1)

PRINTING THE DATA

```
#include <stdio.h>

int main() {
    int arr[] = {45, 98, 3};
    int n, i;
    n = sizeof(arr)/sizeof(int);
    for(i=0; i<n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

TIME COMPLEXITY: O(n)

Linked List Complexity



	Worst case	Average Case
Search	O(n)	O(n)
Insert	O(1)	O(1)
Deletion	O(1)	O(1)

Traverse a Linked List

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

```
struct node *temp = head;
printf("\n\nList elements are - \n");
 while(temp != NULL)
{
printf("%d --->",temp->data);
 temp = temp->next;
}
```

```
List elements are –
 1 --->2 --->3 ---
```

Linked list implementation in C

```
#include <stdio.h>
#include <stdlib.h>
// Creating a node
struct node {
  int value;
  struct node *next;
};
// print the linked list
   value
void
   printLinkedlist(struct
   node *p)
 {
  while (p != NULL) {
   printf("%d ", p-
   >value);
   p = p->next;
  }
}
```

```
int main()
{
 // Initialize nodes
 struct node *head;
 struct node *one = NULL;
 struct node *two = NULL;
 struct node *three = NULL;
// Allocate memory
 one = malloc(sizeof(struct
  node));
 two = malloc(sizeof(struct
  node));
 three = malloc(sizeof(struct
  node));
 // Assign value values
 one->value = 1;
 two->value = 2;
 three->value = 3;
```

```
  // Connect nodes
  one->next = two;
  two->next = three;
  three->next = NULL;
// printing node-value
  head = one;
  printLinkedlist(head);
}
```

Traverse a Linked List

```
STEP 1: SET PTR = START.
STEP 2: PTR== NULL
WRITE 'EMPTY LIST' AND GO TO STEP 6
STEP 3: REPEAT STEP 4 AND 5 UNTIL
PTR != NULL.
STEP 4: PRINT PTR→ DATA.
STEP 5: PTR = PTR → NEXT.
STEP 6: EXIT.
```

Traverse a Linked List

```
STEP 1: SET PTR = START.
STEP 2: REPEAT STEP 3 AND 4 UNTIL PTR !=
NULL.
STEP 3: IF (PTR --> ITEM )
        PRINT SET LOC=PTR AND EXIT.
STEP 4: ELSE SET PTR = PTR → NEXT.
STEP 5: PRINT LOC=NULL
STEP 6: EXIT.
```

Insert node to a Linked List

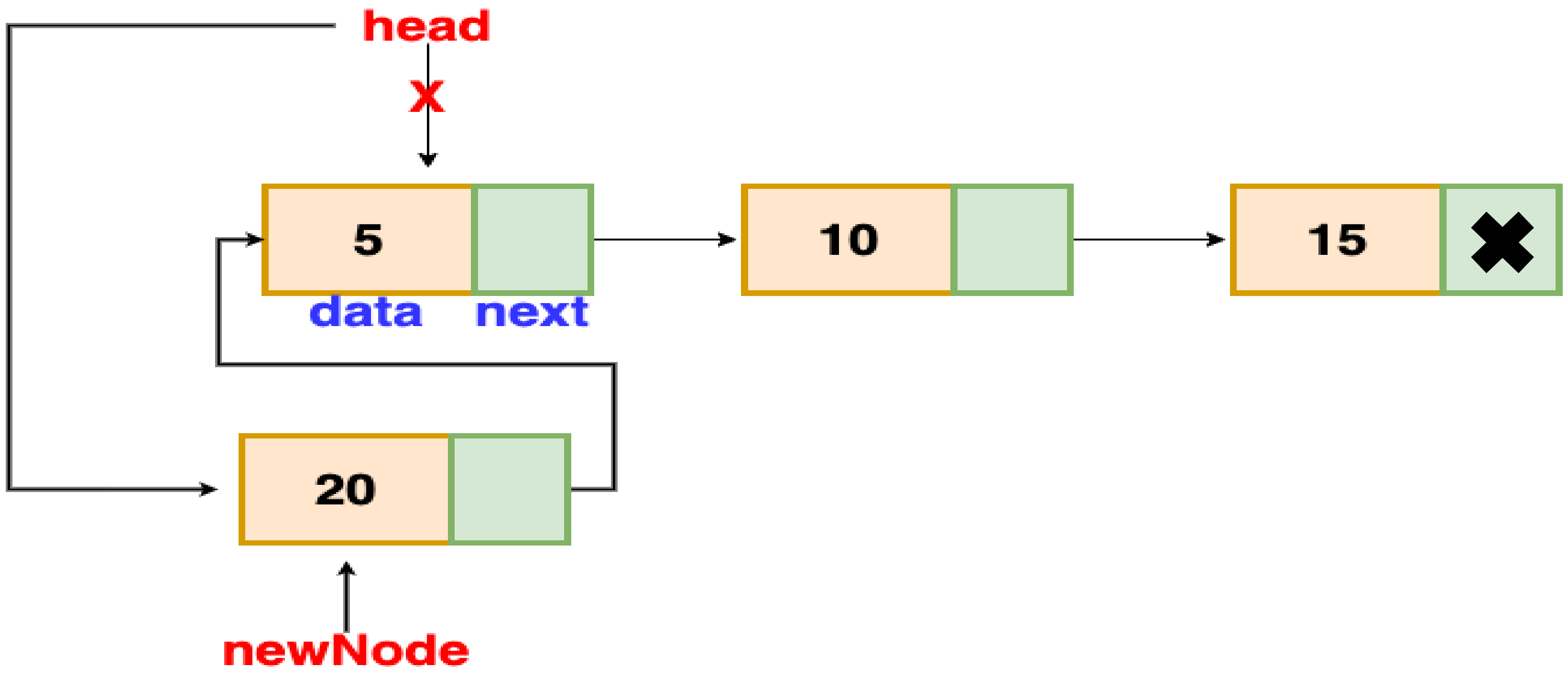
1. Insert at the beginning

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct
node));
newNode->data = 4;
 newNode->next = head;
head = newNode;
```

Insertion in a Single Linked List (at beginning)

- Insertion at beginning



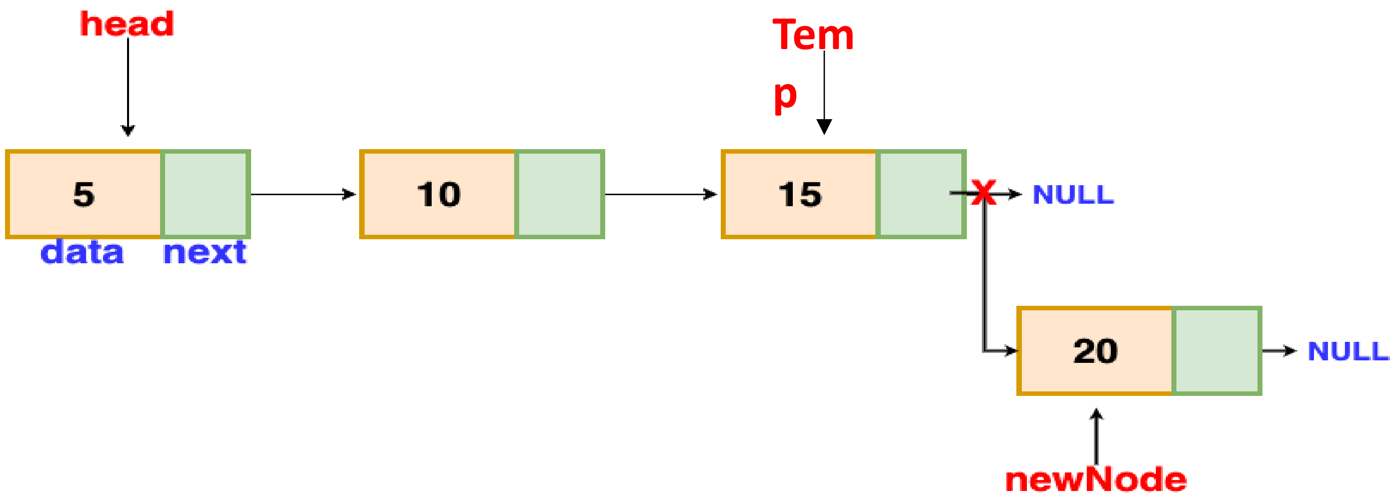
Insertion at the beginning

Algorithm Insert node at beg to a Linked List

```
Step 1: IF PTR == NULL. Print overflow & Exit
            ELSE PTR= MALLOC(SIZE OF(STRUCT LINK))
Step 2: SET PTR → INFO = ITEM.
Step 3: SET PTR→ NEXT = START.
Step 4: SET START = PTR.
Step 5: EXIT.
```

Insertion in a Single Linked List (at end)

- Insertion at end



Insertion at the end

Insert node to a Linked List

2. Insert at the End

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

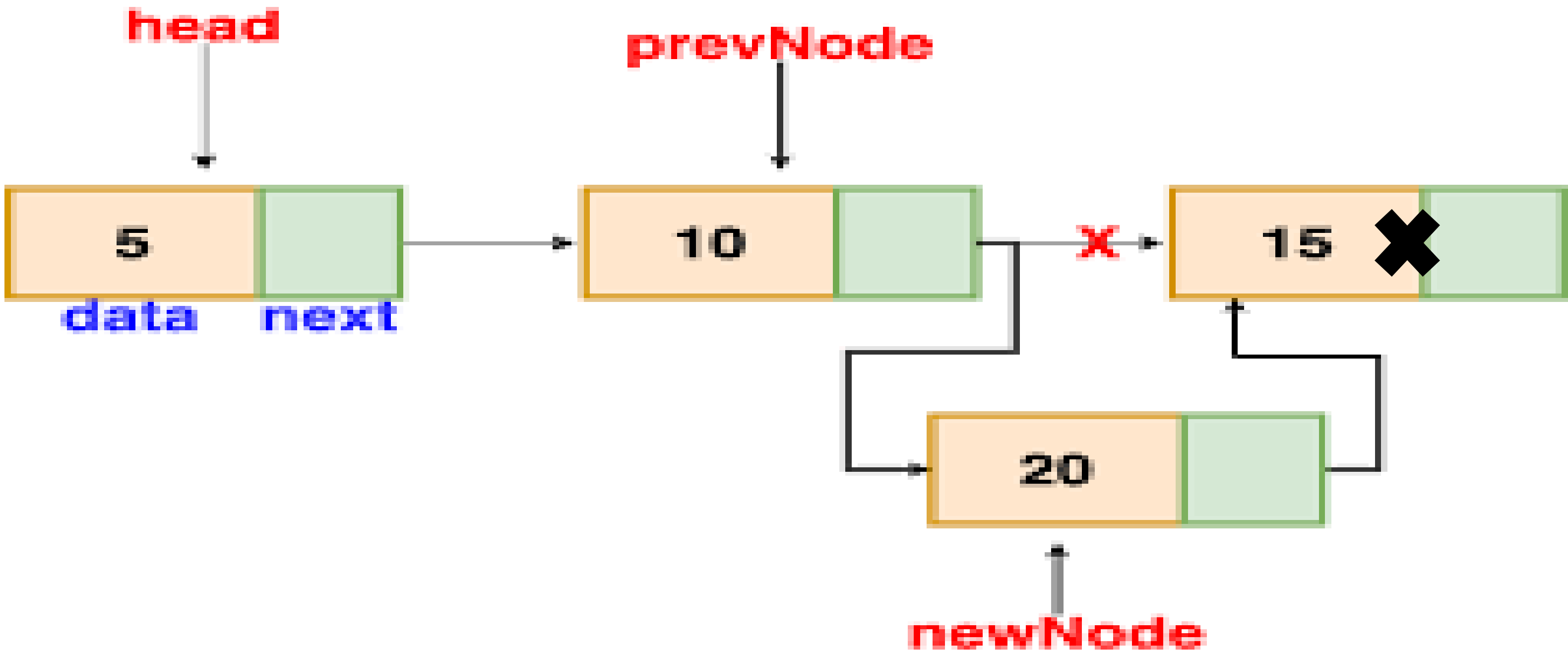
```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;
struct node *temp = head;
while(temp->next != NULL)
{
        temp = temp->next;
 }
 temp->next = newNode;
```

Insertion at the End of a Linked list:

```
Step 1: IF PTR == NULL. Print overflow & Exit
        ELSE PTR= MALLOC(SIZE OF(STRUCT LINK))
Step 2: SET PTR → INFO = ITEM.
Step 3: SET PTR→ NEXT = NULL.
Step 4: IF (START==NULL)
        SET START =PTR
Step 5: ELSE LOC=START
Step 6: REPEAT STEP 7 UNTIL LOC → NEXT !=NULL
Step 7: SET LOC= LOC → NEXT
Step 8: SET LOC → NEXT = PTR
```

Insertion in a Single Linked List (at given position)

- **Insertion at given position**



Insertion after a given node

Insert node to a Linked List

3. Insert at the Middle/ Specific location

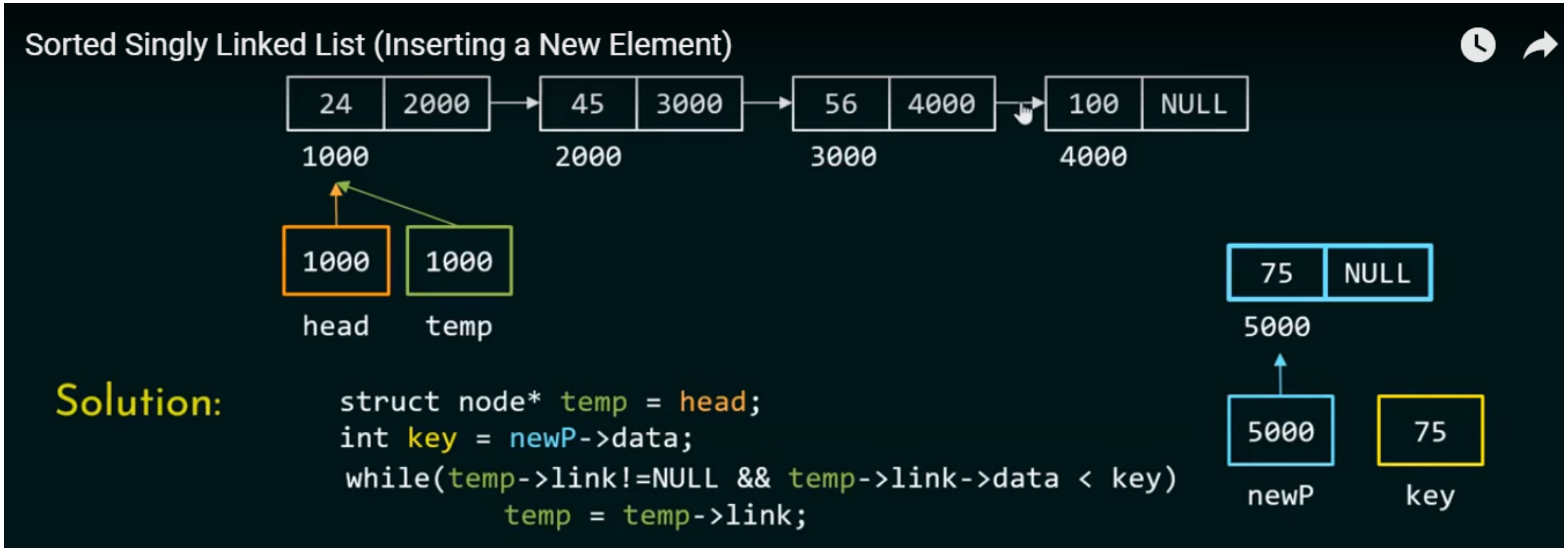
- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
struct node *temp = head;
for(int i=2; i < position; i++) {
  if(temp->next != NULL) {
    temp = temp->next;
  }
}
newNode->next = temp->next;
temp->next = newNode;
```

Insert node in sorted Linked List

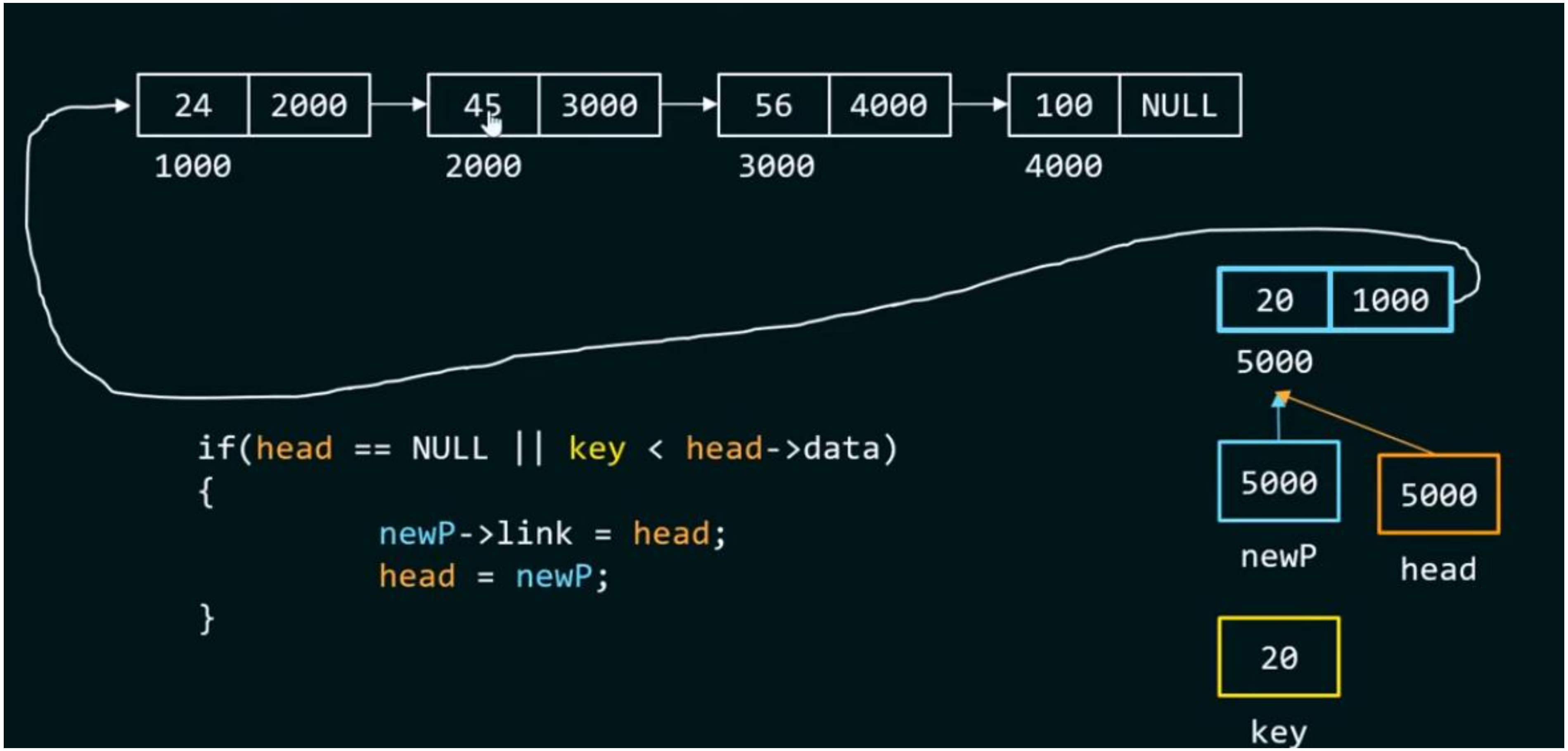
3. Insert at the Specific location by checking data also

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node and compare data of new node with all node.
- Change next pointers to include new node in between



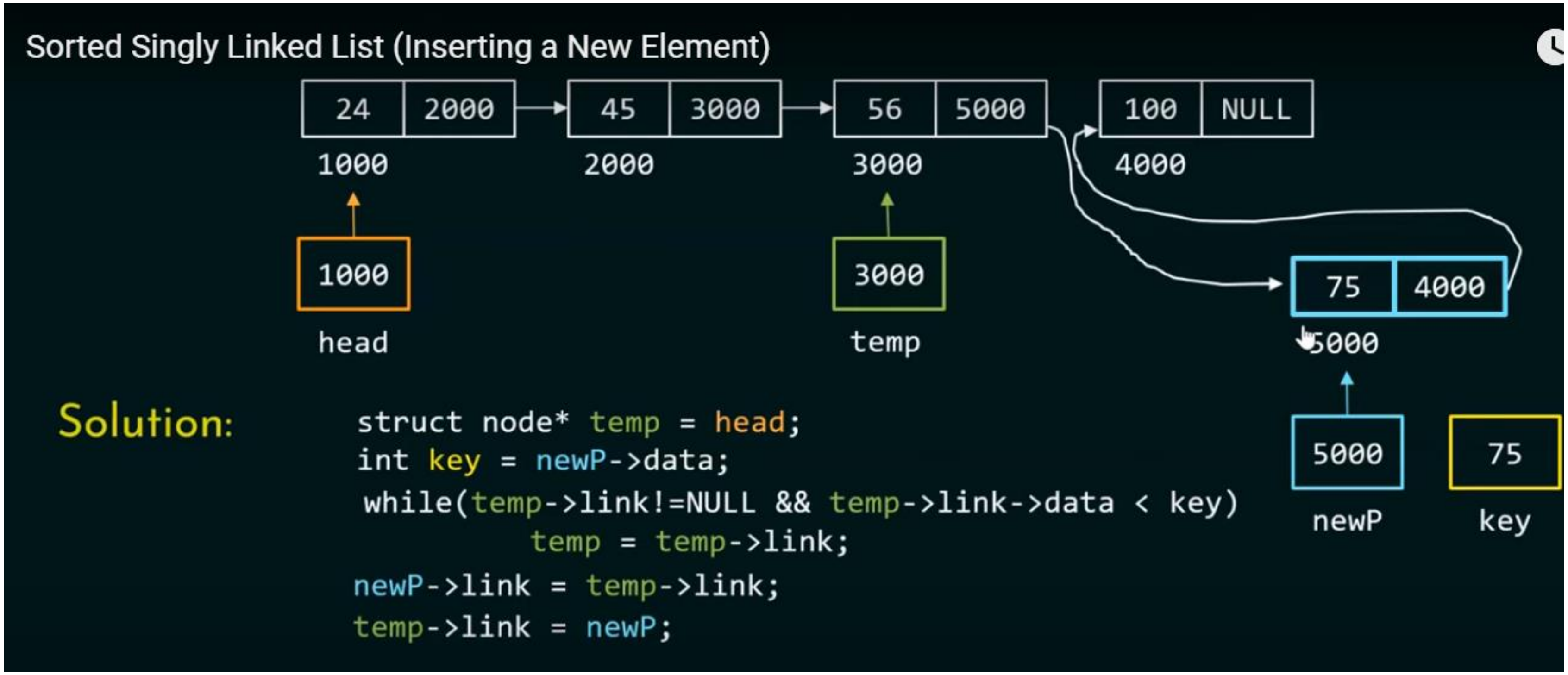
```
struct node* temp = head;
int key = newP->data;
while(temp->link!=NULL && temp->link->data < key)
        temp = temp->link;
```

Insert node in sorted Linked List



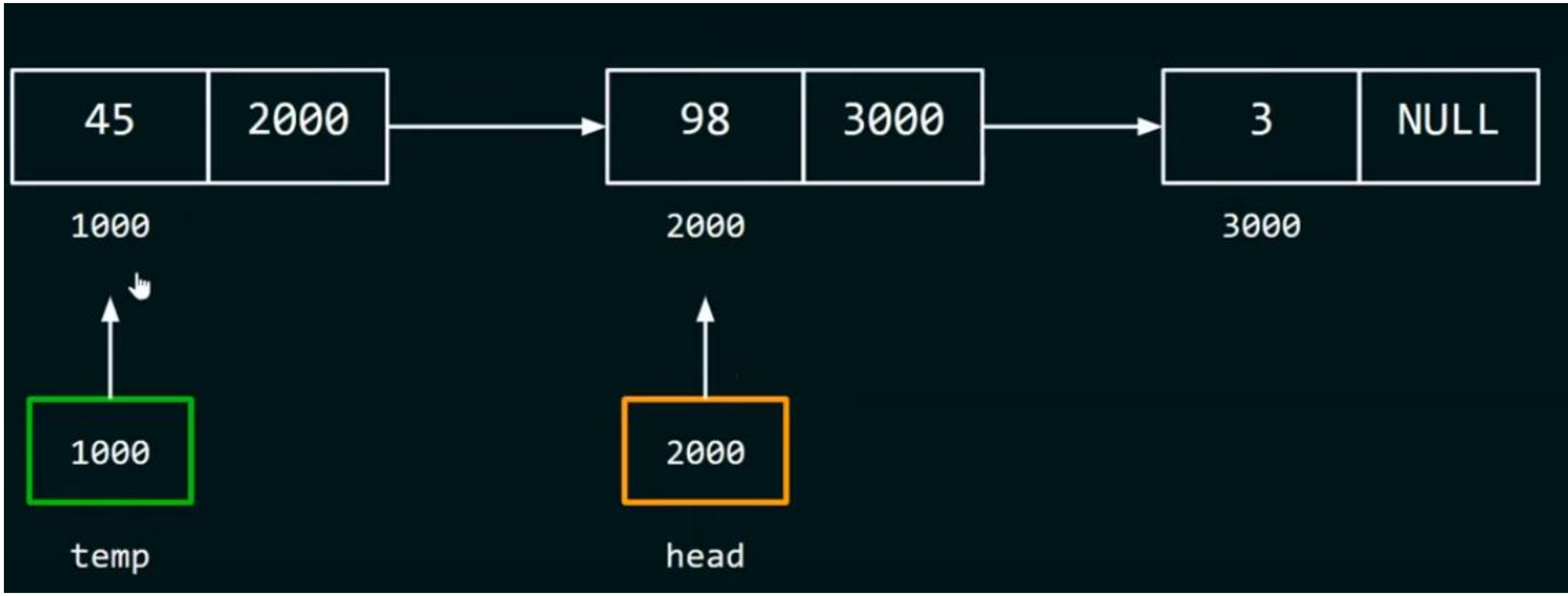
```
if(head == NULL || key < head->data)
{
            newP->link = head;
            head = newP;
}
```

Insert node in sorted Linked List



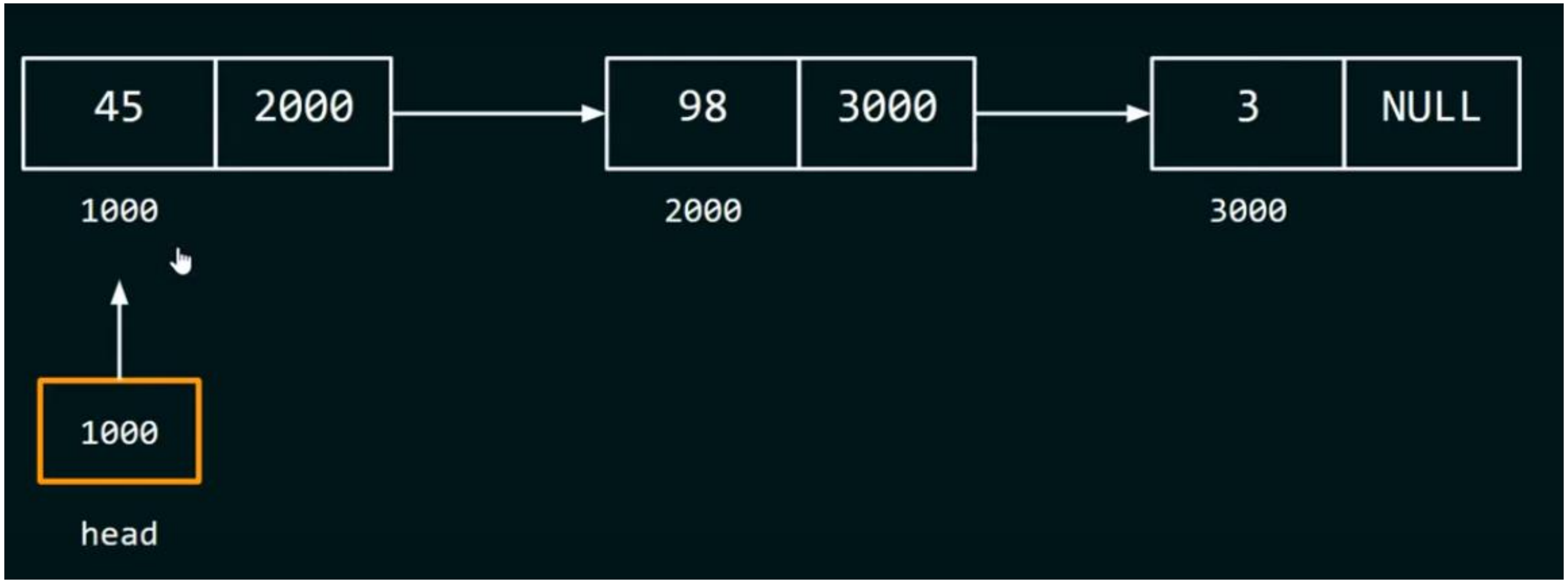
```
struct node* temp = head;
int key = newP->data;
 while(temp->link!=NULL && temp->link->data < key)
         temp = temp->link;
newP->link = temp->link;
temp->link = newP;
```

Deletion in Single Linked List (from beginning)

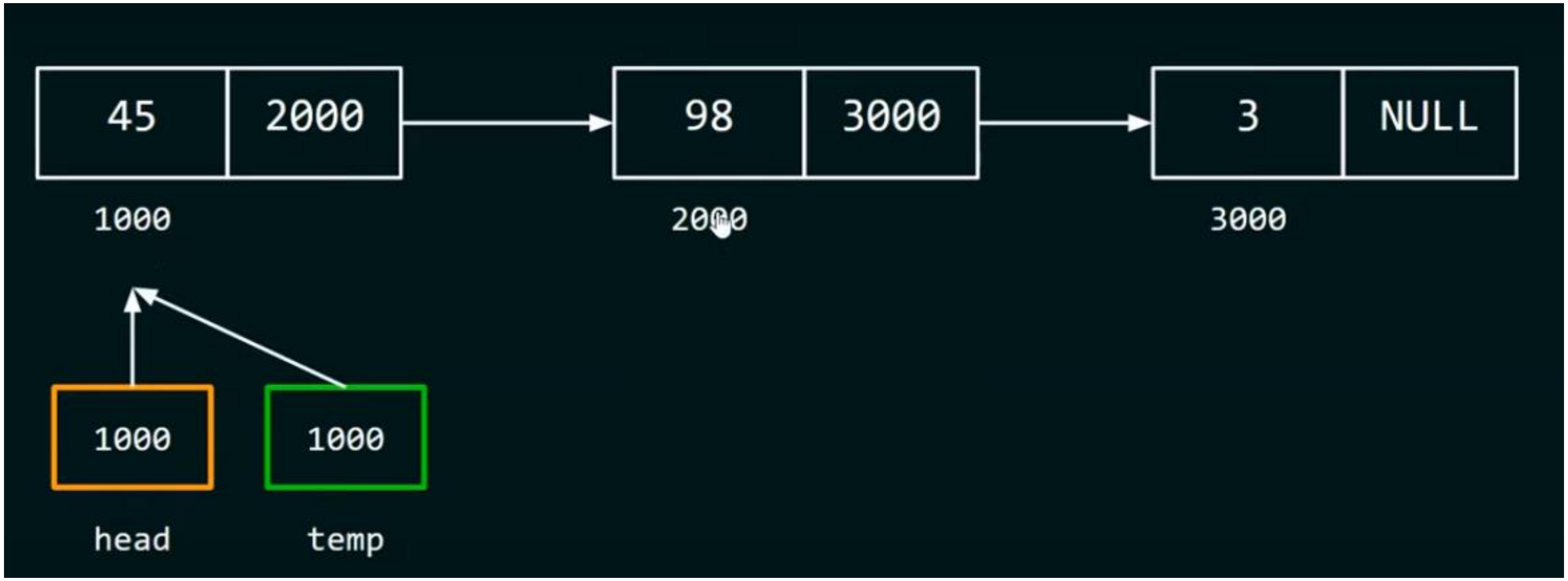


Deletion in Single Linked List (from beginning)

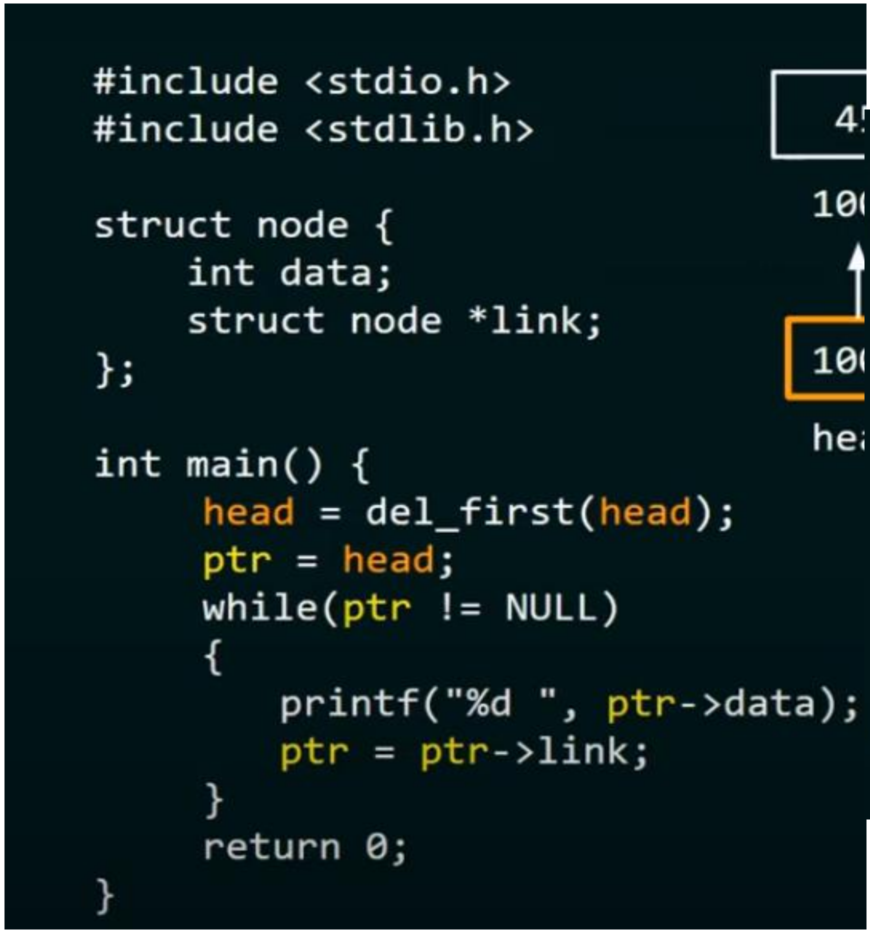
- **Deletion from beginning**



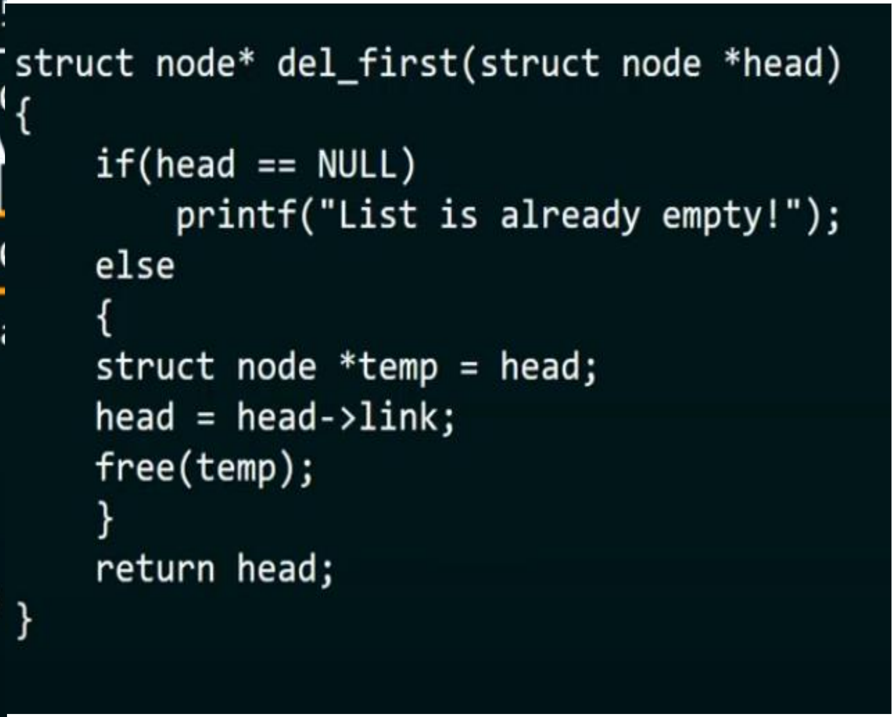
Deletion in Single Linked List (from beginning)



Deletion in Single Linked List (from beginning)

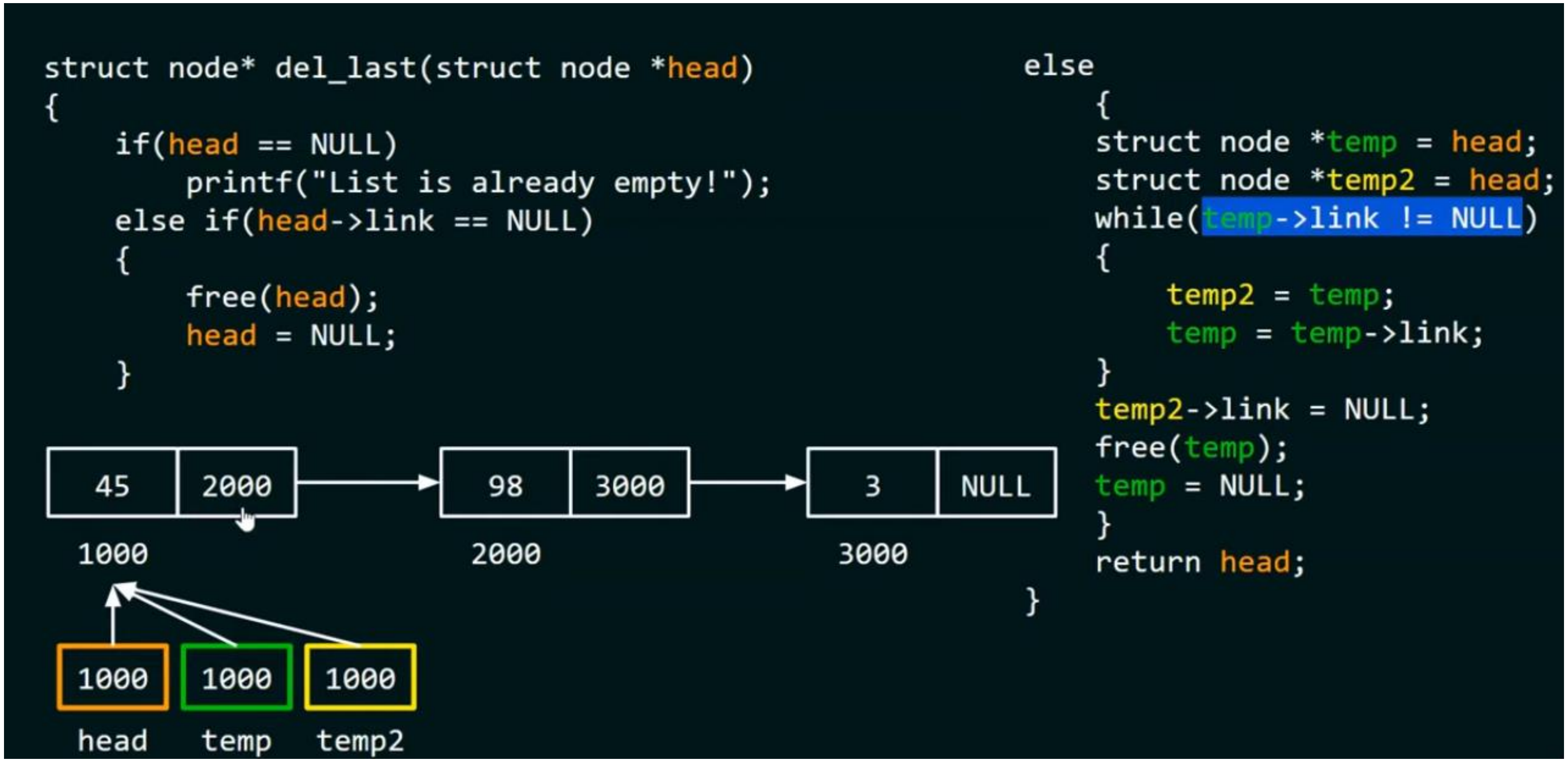


```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *link;
};

int main() {
     head = del_first(head);
     ptr = head;
     while(ptr != NULL)
     {
         printf("%d ", ptr->data);
         ptr = ptr->link;
     }
     return 0;
}
```



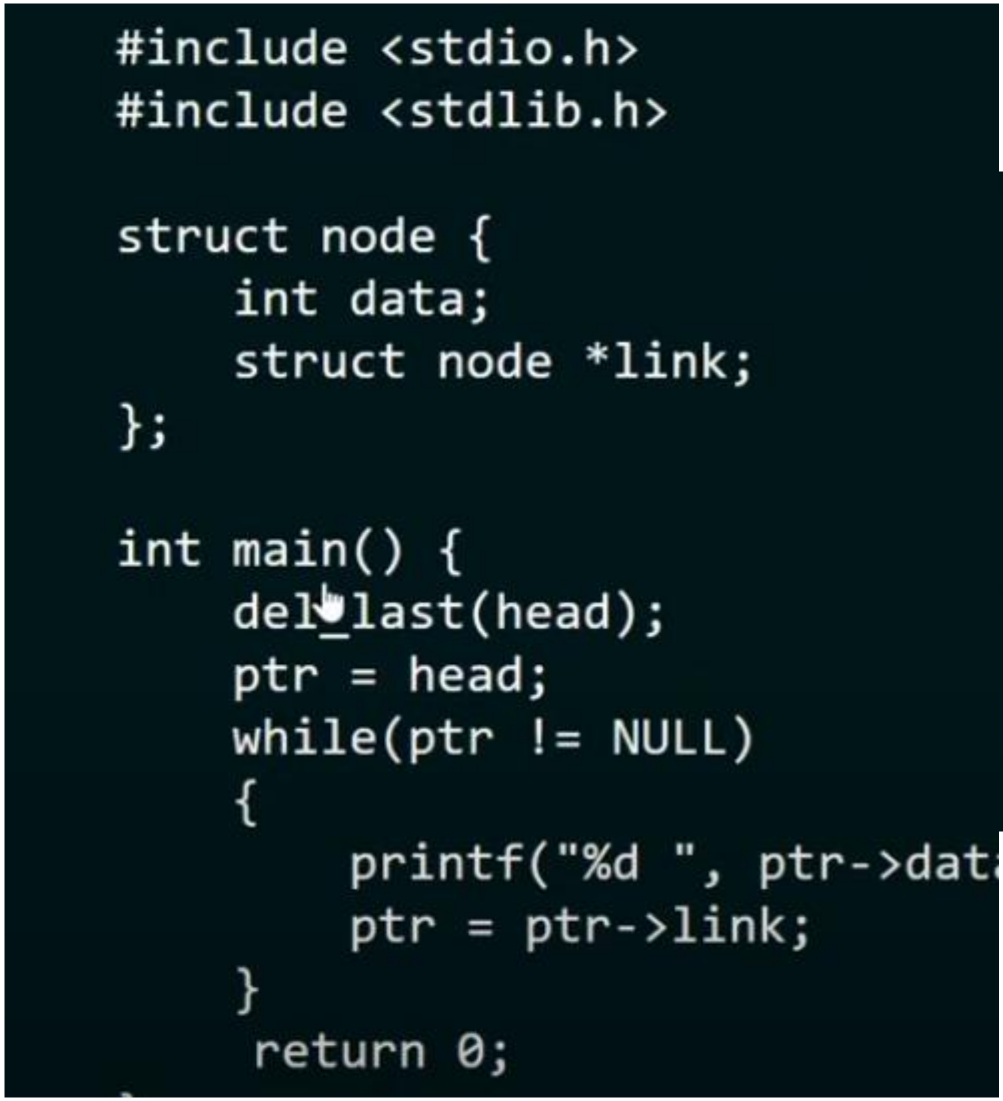
```c
struct node* del_first(struct node *head)
{
    if(head == NULL)
        printf("List is already empty!");
    else
    {
    struct node *temp = head;
    head = head->link;
    free(temp);
    }
    return head;
}
```

Deletion in Single Linked List (from end)

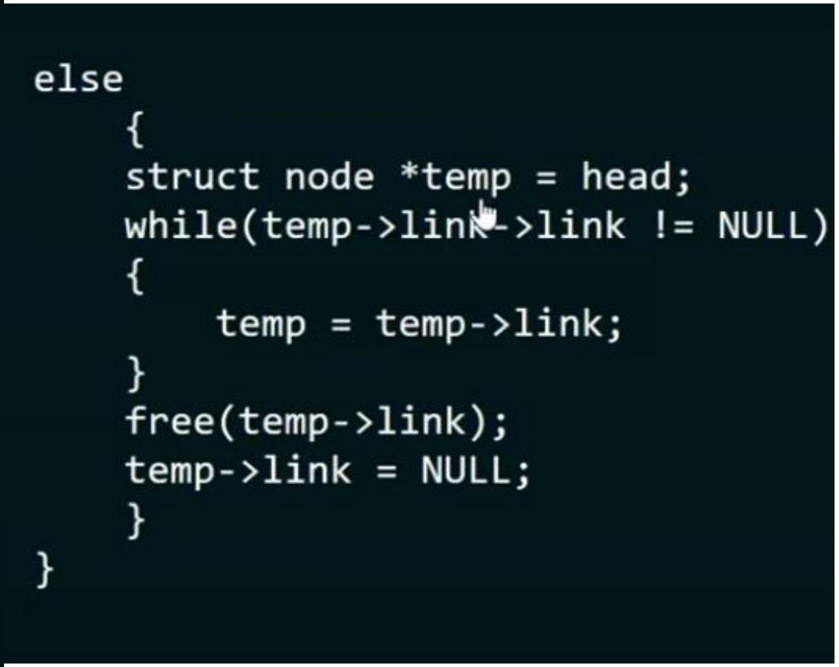


```
struct node* del_last(struct node *head)
{
    if(head == NULL)
        printf("List is already empty!");
    else if(head->link == NULL)
    {
        free(head);
        head = NULL;
    }
    else
    {
    struct node *temp = head;
    struct node *temp2 = head;
    while(temp->link != NULL)
    {
        temp2 = temp;
        temp = temp->link;
    }
    temp2->link = NULL;
    free(temp);
    temp = NULL;
    }
    return head;
}
```

Deletion in Single Linked List (from end)

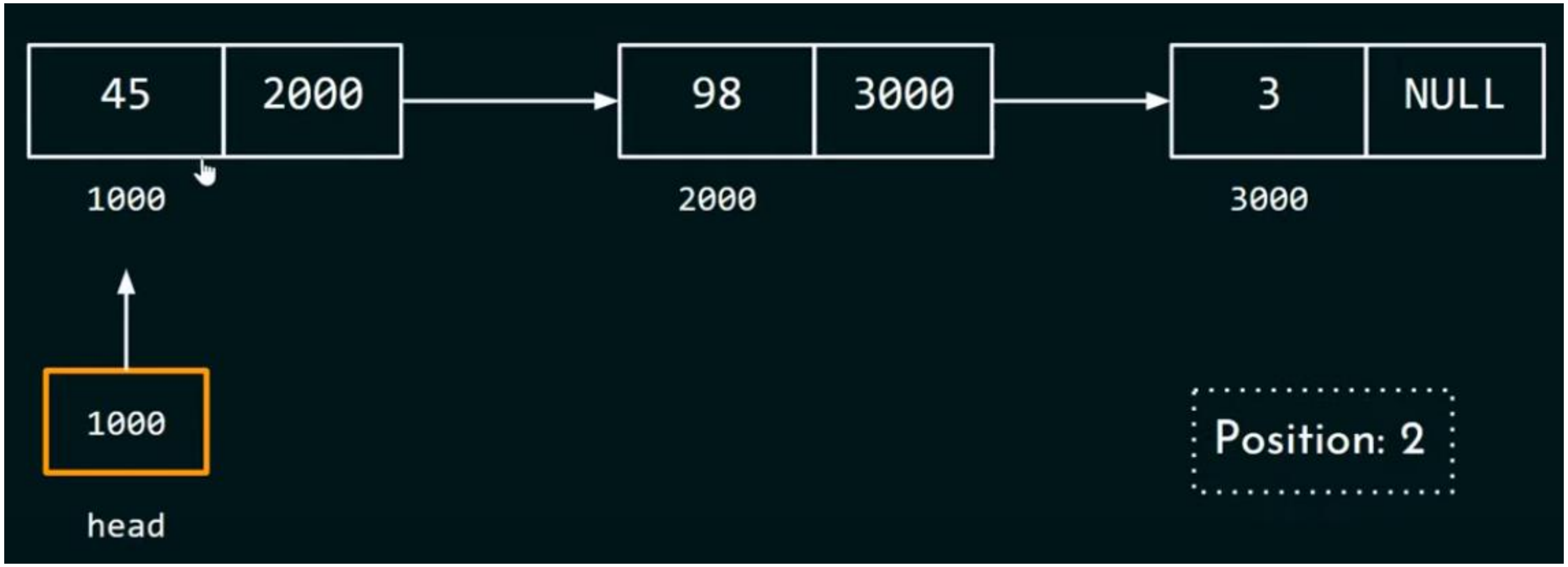


```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *link;
};

int main() {
    del_last(head);
    ptr = head;
    while(ptr != NULL)
    {
        printf("%d ", ptr->dat
        ptr = ptr->link;
    }
     return 0;
```

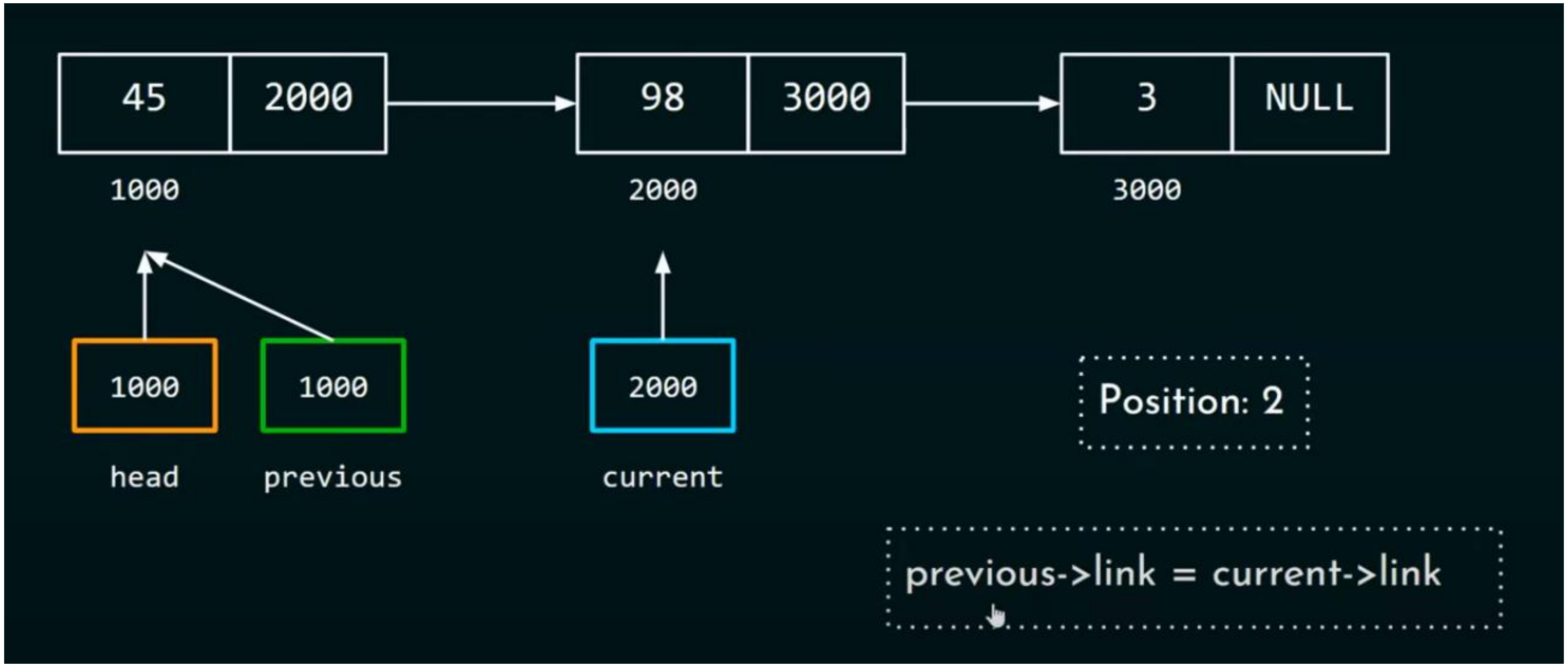


```
else
    {
    struct node *temp = head;
    while(temp->link->link != NULL)
    {
        temp = temp->link;
    }
    free(temp->link);
    temp->link = NULL;
    }
}
```

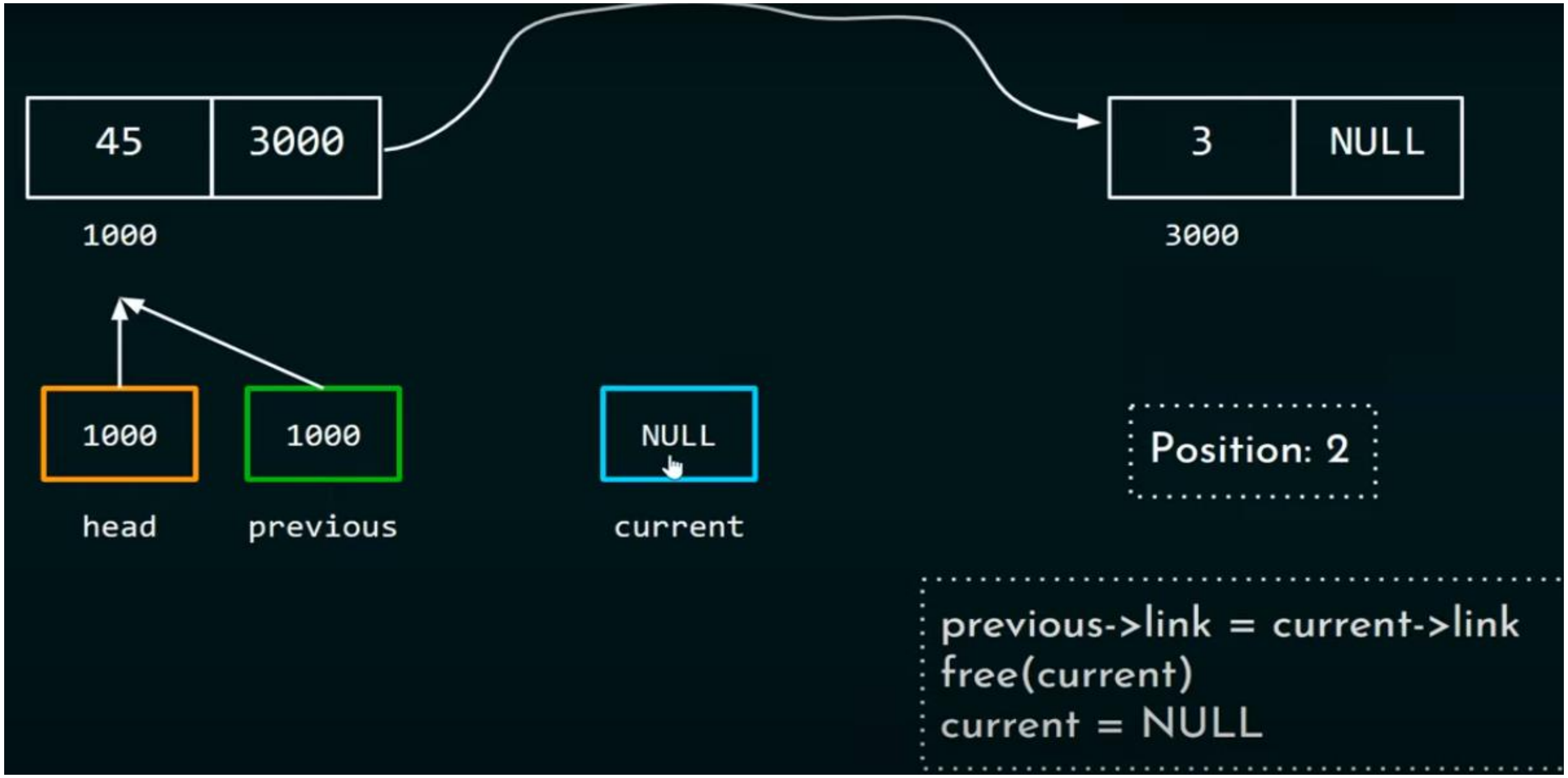
Deletion in Single Linked List (from position)



Deletion in Single Linked List (from position)



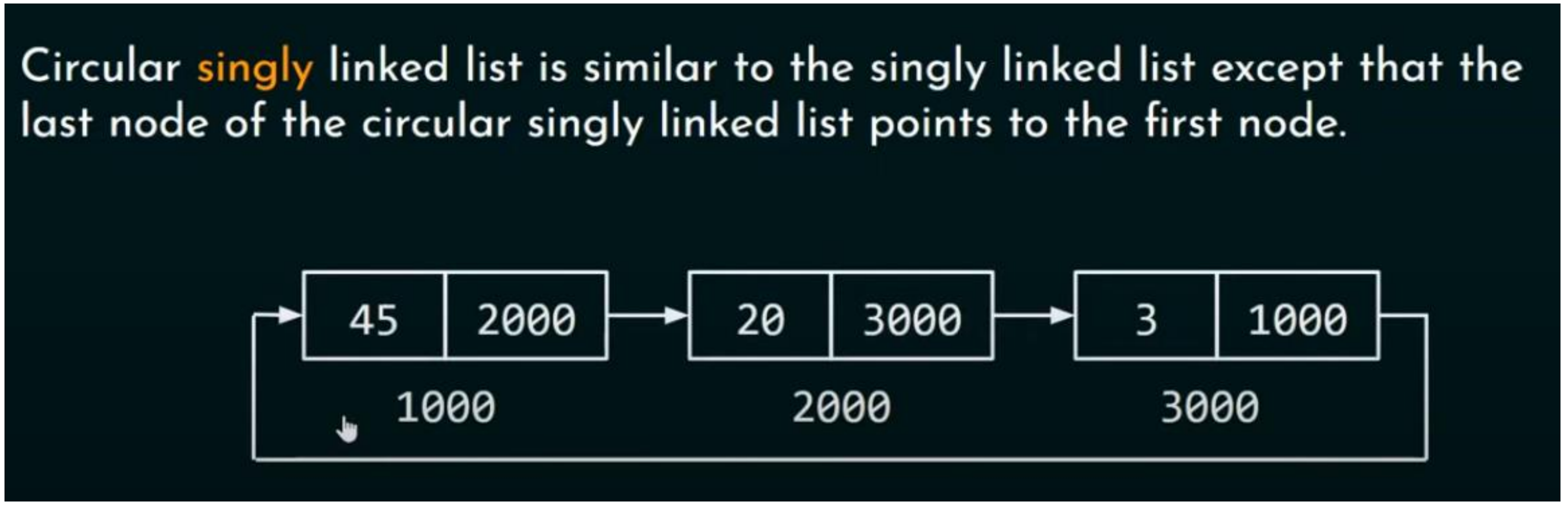
Deletion in Single Linked List (from position)



```
previous->link = current->link
free(current)
current = NULL
```

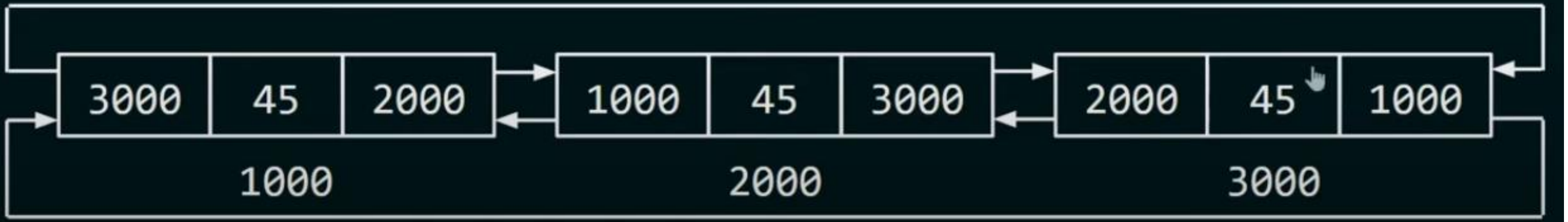
Singly Circular Linked list

- **The circular linked list** is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.

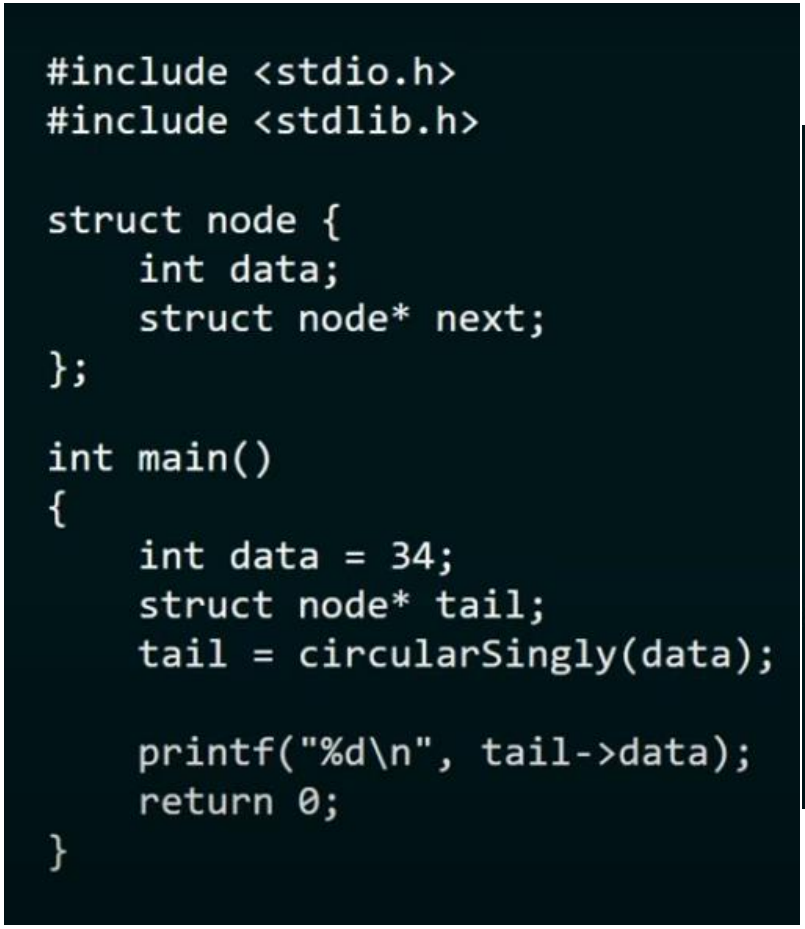

Circular singly linked list is similar to the singly linked list except that the last node of the circular singly linked list points to the first node.

Doubly Circular Linked list

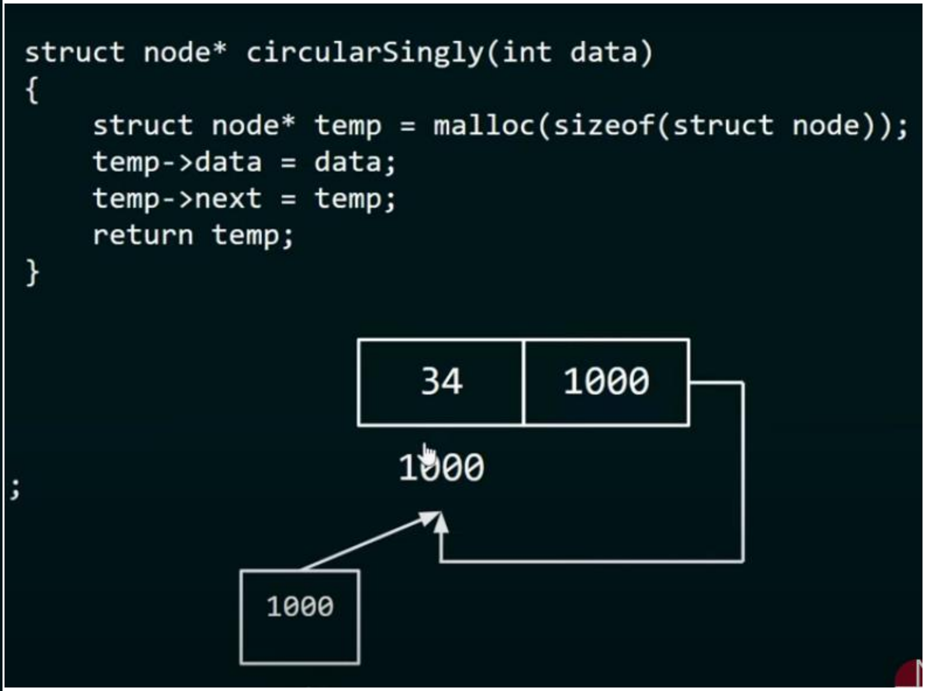
Circular doubly linked list is similar to the doubly linked list except that the last node of the circular doubly linked list points to the first node and the first node of the circular doubly linked list points to the last node.



Insertion in a Single Circular Linked List

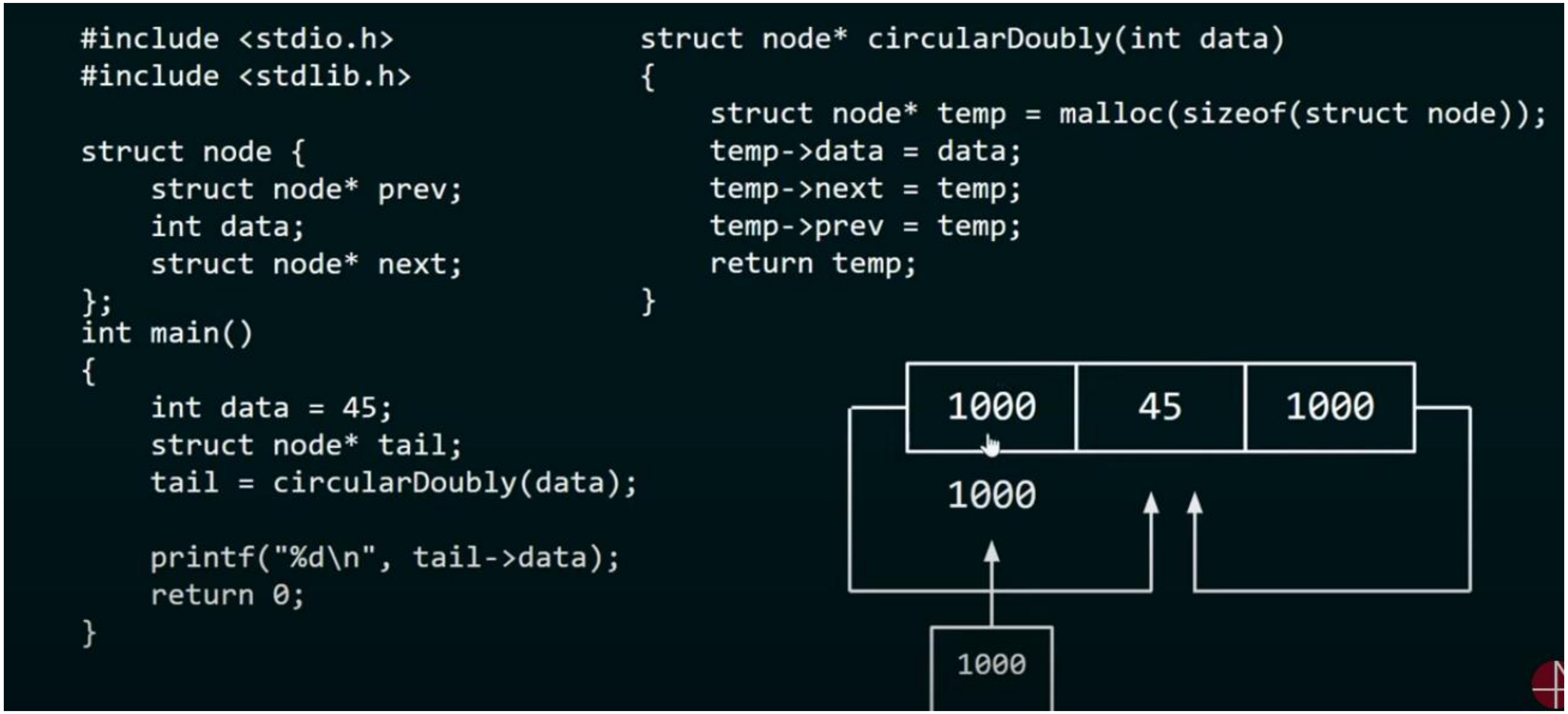


```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node* next;
};

int main()
{
    int data = 34;
    struct node* tail;
    tail = circularSingly(data);

    printf("%d\n", tail->data);
    return 0;
}
```



```c
struct node* circularSingly(int data)
{
    struct node* temp = malloc(sizeof(struct node));
    temp->data = data;
    temp->next = temp;
    return temp;
}
```

Insertion in a doubly Circular Linked List



```
#include <stdio.h>
#include <stdlib.h>

struct node {
    struct node* prev;
    int data;
    struct node* next;
};
int main()
{
    int data = 45;
    struct node* tail;
    tail = circularDoubly(data);

    printf("%d\n", tail->data);
    return 0;
}
```

```
struct node* circularDoubly(int data)
{
    struct node* temp = malloc(sizeof(struct node));
    temp->data = data;
    temp->next = temp;
    temp->prev = temp;
    return temp;
}
```

Deletion in a Single Circular Linked List

- There are three possible positions where we can enter a new node in a linked list –
  - **Deletion at beginning**
  - **Deletion at end**
  - **Deletion from given position**
  - **Deletion_by_value**

- Deleting new node in linked list is a more than one step activity.