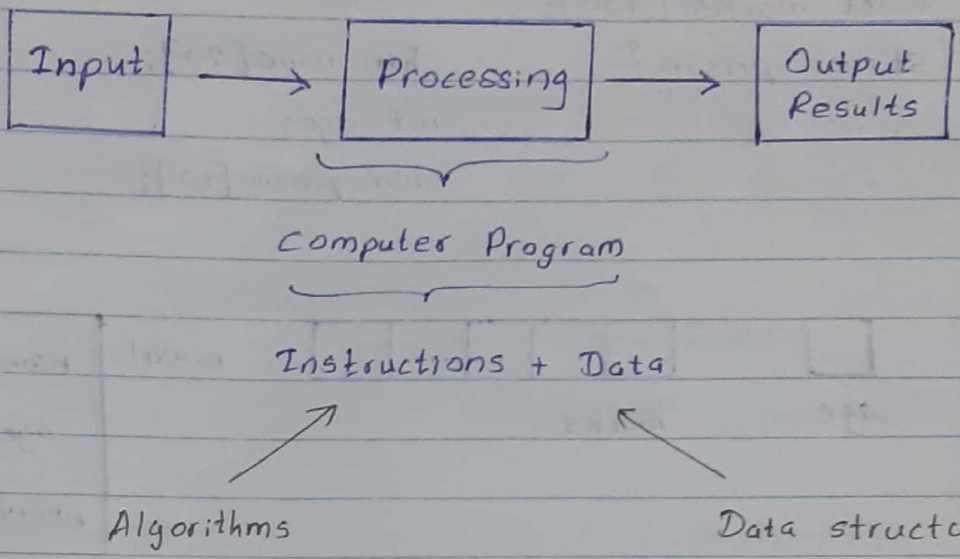
Algorithms and Data Structures.

Context of Computing



What are data structures?

A data structure is a specific format for storing, organizing, and processing data.

Programming languages support data storage via,

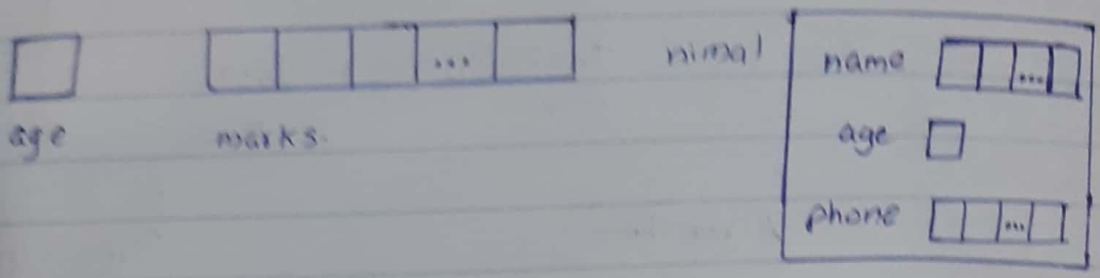
- Variables of basic data types
- Arrays
- Records / structures

More complex data structures can be implemented using the basic data types and other language elements available.

- Linked lists
- Trees

Examples

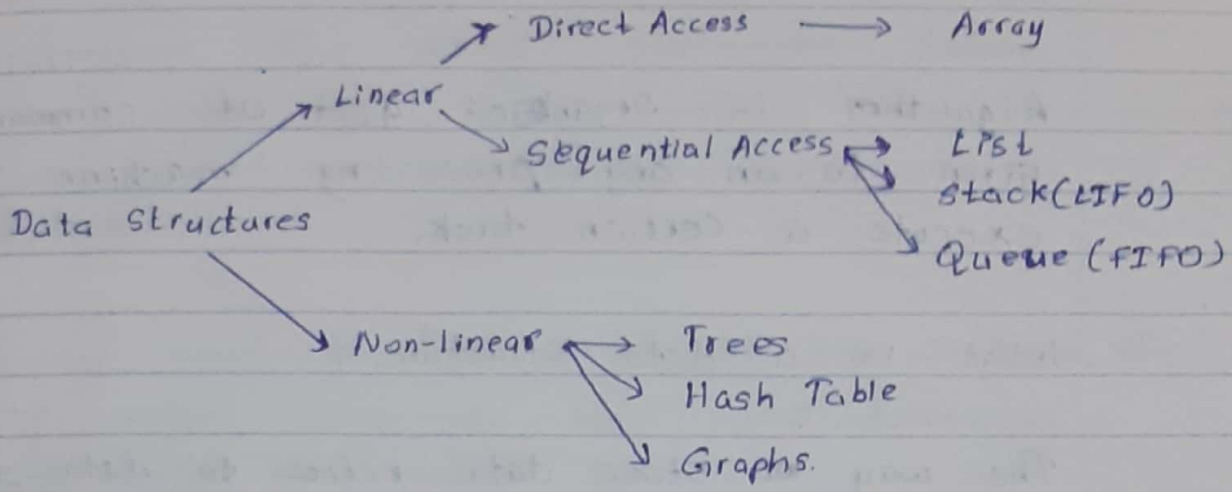
```
int age;
float marks[100];
struct person {
    char name[20];
    int age;
    char phone[10];
} nimal;
```



Data Structures Types

- There are several common data structures: arrays, linked lists, queues, stacks, binary trees, hash tables, etc.
- These data structures and be classified as either linear or nonlinear data structures, based on how the data is conceptually organized or aggregated.

Data Structures Classification



Linear Vs Non-Linear Data Structures

Linear Structures.	Non-Linear Structures.
Array, list, queue, and stack belong to this category	Trees and graphs are classical non-linear structures
Each of them is a collection that stores its entries in a linear sequence.	Data entries are not arranged, but with different rules.
They differ in the restrictions they place on how these entries may be added, removed, or accessed	
common restrictions include FIFO and LIFO	

Algorithm

What is a algorithm ?

Algorithm is organized set of commands given to an any processing machine to execute a certain task.

What is a data structure ?

The way we store data refers to data structures

Structures

names
Age
Address
Marks

```
int main()
{
    char nam1[50];
    int age1;
    char add1[50];
    float mark1;
```

What is a structure ?

Structure is a user defined data type available in C that allows to combine data items of different kinds.

Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows

Saman want to keep track of his cars in sale. He might want to tract the following attributes about each car.

- make
- model
- chasi number
- eng capacity

```
struct cars {
        char make [50];
        char model [50];
        char chasino [100];
        int engcpty;
};
```

Creating variables from structures

```
Struct cars {
        char make [50];
        char model [50];
        char chasino [100];
        int engcpty;
};
int main() {
struct cars c1 ; /* Declare c1 of type car */
Struct cars c2 ; /* Declare c2 of type cars */
```

```
# include <stdio.h>
# include <stdlib.h>
struct cars {
        char make [20];
        char model [20];
        Char chasi [20];
        float engncpy
};
int main ()
{
    Struct cars c1 ;

    strcpy ( c1.make, "Nissan");
    strcpy ( c1.model, "Leaf");
    strcpy ( c1.chasi, "NKL45632");
    c1. engncpy = 1500;

    printf ("%s\n", c1.make);
    printf ("%s\n", c1. model);
    printf ("%s\n", c1.chasi);
    printf ("%f\n", c1 .engncpy);
```

Stacks.

Storing data one over another in a column format refers to stack concept.

Stack data structure.

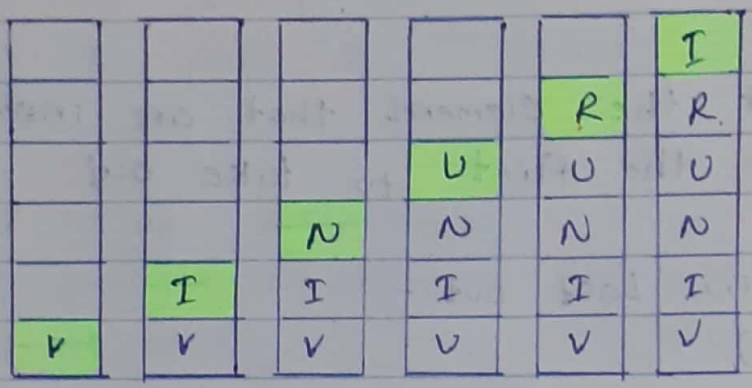
* Stack allows us to access only one data at a time.

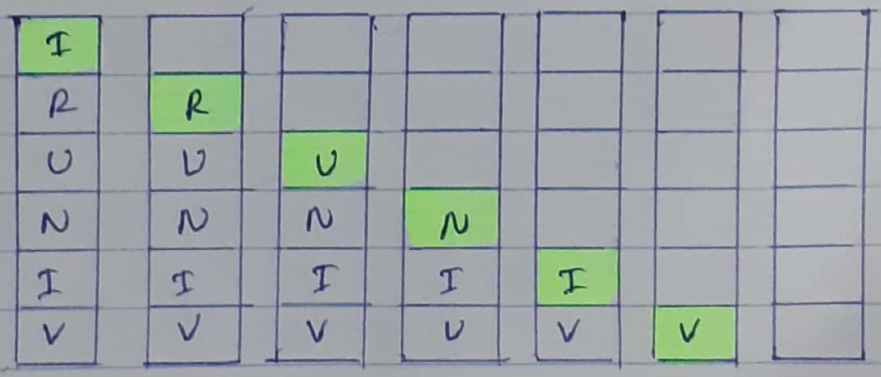
* Inside a stack the elements that are insert to last will be the first to take out.

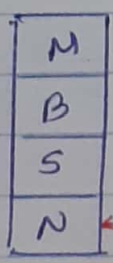
  - First in Last out -
    (FILO)

Graphical Representation

Inserting letter V
Inserting letter I
Inserting letter N
Inserting letter U
Inserting letter R
Inserting letter I







← Open edge (In / out)

← close edge.

- Since the stack having only one end it requires only one pointing variable.
- In the begining, when the stack is empty pointer variable pointing to index minus one (-1)
- When we are inserting values pointer will increment,

  pointer ++
- When we are taking out values pointer will be decrement

Implementing an integer stack size 10

pointer ++

```
int main()
{
    int stack[10];
    int top = -1;

    return 0;
}
```

Inserting values into the stack — PUSH()

* Check whether stack is full or not

step 1: Increase the top by 1

step 2: assign the value into the cell pointing by top

if it is not

step 1 :

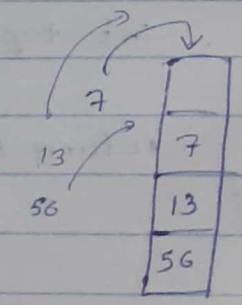
step 2 :

else

output can't add values stack is full

```
void push(int data)
{
    top = top + 1;
    stack[top] = data;
}
```

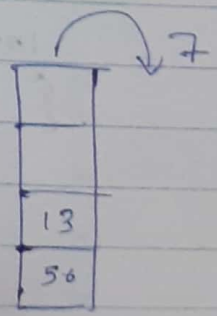
```
void push (int data)
{
    top = top + 1;
    stack[top] = data;
}
void printdata()
{
    int i;
    for (i = top ; i >= 0; i--)
    {
        printf ("%d\n", stack[i]);
    }
}
int main()
{
    push(56);
    push(13);
    push(7);
    printdata();
}
```



Removing values from the stack **POP()**

* check wether the stack is empty or not
  step 1 : remove the value point by top
  step 2 : decrement top by one.

```
int popO
{
    int temp;
    temp = stack [top];
    top = top-1

    return temp;
}
int main()
{
    push (56);
    push (13);
    push (7);
    printdata ();
    printf ("Removed value is % d \n", popO);
    printdata ();

    return 0;
}
```



Checking the stack is empty or not

ISEMPTY ()

If top == -1 ⟶ stack is empty.

```
int pop()
{
   if (isempty() ==0)
   {
      int temp;
      temp = stack[top];
      top = top-1;

      return temp;
   }
   else
   {
      printf ("can't extract values stack is empty \n");
   }
}
int isempty()
{
   if (top == -1)
   {
      return 1;   // stack is empty it will returns 1
   }
   else
   {
      return 0;   // stack is not empty it will returns 0
   }
}
```

Checking the stack is full or not **ISFULL()**

If top == size - 1 → stack is full

```
int isfull()
{
   int size;
   if (top == size-1)
   {
      return 1;      // stack is empty it will return 1
   }
   else
   {
      printf ("cannot extract values stack is empty \n")
   }
}
void printdata()
{
   int i;
   printf("Stack contents\n");
   for (i = top; i>=0; i--)
   {
      printf("%d\n", stack[i]);
   }
}
int main()
{
   push(56);
   push(12);
   push(9);
   push(128);
   push(12);
   printdata();
   return 0;
}
```

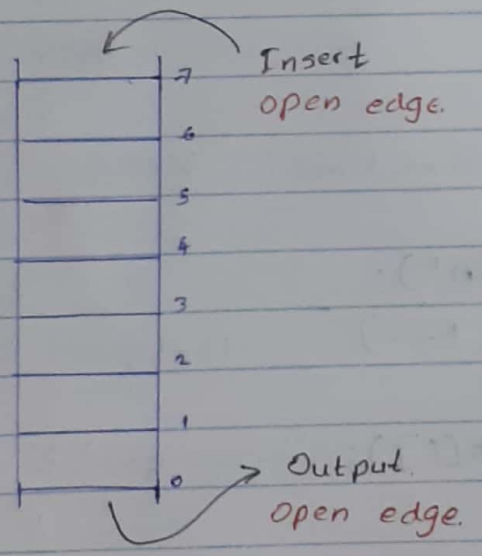
Queue

Queue consider as a linear data structure. which storing data in contegers cell according to first in out method.

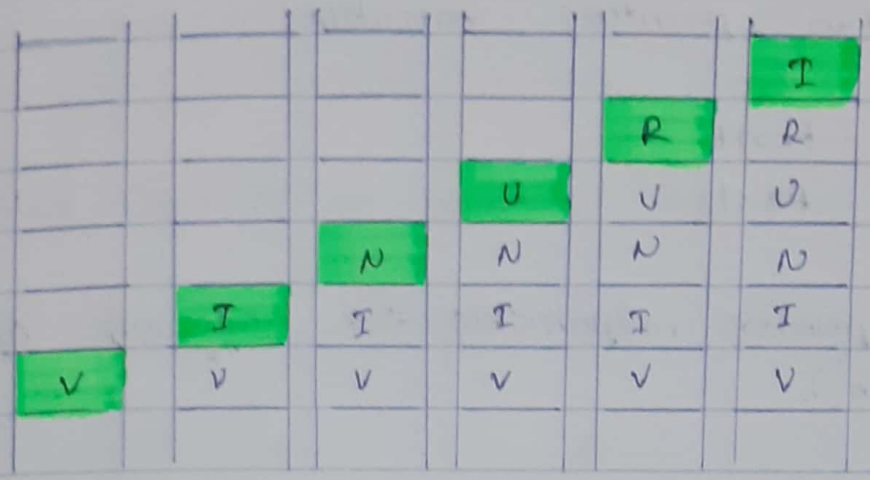
The first element to be remove from the queue is the first element we inserted to the queue.

Queue, data structure is similar to stack

Structure of a Queue



(a) Inserting my name into a queue graphical representation



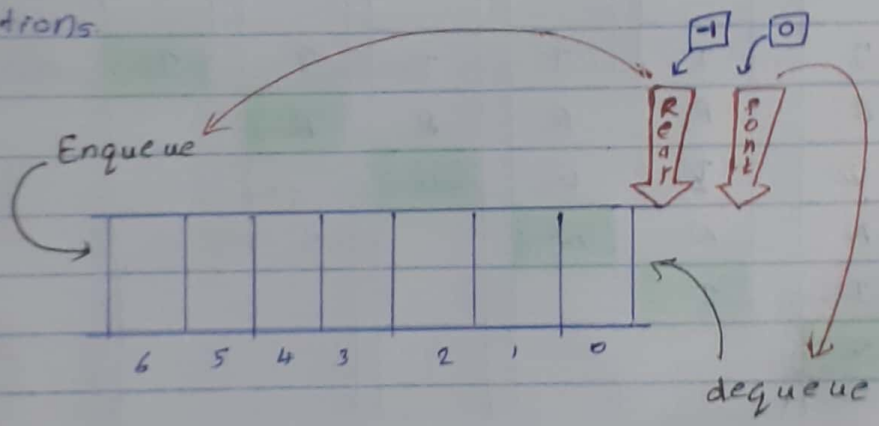
Implementation of Queue

- Queue has two open edges. Therefore we need two pointing variables

  (1) Front
  (2) Real

- Front pointer responsible for Enqueue operations (Insertin)

- Rear pointer is responsible for dequeue operations



- Real is starting from -1

- Front is starting from 0.

01. When we are enqueue a value, [Enqueue()]

Check if the queue is empty or not
- If its empty → can't add more values
- If not empty → lets add values

(i) Rear needs to be increment by one.

(ii) The data will occupied that index cell.

```
void enqueue (int data)
{
   if ( isfull () )
   rear = rear + 1 ;
   queue [ rear] = data ;
}

void print()
{
   int i;
   For( i=0 ; i<=rear ; i++ )
   {
     printf ("%d \n", queue[i]);
   }
}

int main ()
{
   enqueue (25);
   enqueue (75);
   print();
}
```

02. When we extracting values, [dequeue()]

Check if the queue is empty or not
- If its empty → Can't dequeue more values
- If not empty → lets dequeue values

(i). Remove the value point by front.

(ii). Front needs to be increment by one.

```
void dequeue()
{
   printf("Value extracted is %d \n", queue[front])
   front = front + 1;
}

void print()
{
   int i;
   printf("The Queue \n");
   for (i=0; i<=rear; i++)
   {
      printf ("%d \n", queue[i]);
   }
}
int main()
{
   enqueue(25);
   enqueue(79);
   print();
   dequeue();
   print();

   return 0;
}
```

09. ISFULL ()

If the pointer . rear == maximum size of the queue ----> full

Else,

queue ----> is not full

```
void isfull()
{ int max;
  if (rear = max-1)
  {
    printf("Queue is full !!\n");
  }
  else
  {
    printf("Queue is not full !!\n");
  }
}

int main()
{
    isfull();
    enqueue(25);
    enqueue(75);
    isfull();
    enqueue(127);
    enqueue(90);
    isfull();

    return 0;
}
```

```
//enqueue
void enqueue (int data)
{
   if (isfull() == 0)
   {
      rear = rear + 1;
      queue[rear] = data;
   }
   else
   {
      printf("cannot add more values
      queue is full");
   }
}
// isfull
void isfull()
{ if (rear = max-1)
  { return 1;
  }
  else
  { return 0;
  }
}
```

04. ISEMPTY ()

- for a queue to be empty there are two conditions that can happen.

(i). If rear == -1 queue is empty

(ii). If the front overrides rear again queue is empty

(If rear == -1 OR (front > rear) → empty)

```
// deque
void dequeue()
{
    if (isempty () == 0)
    {
      printf (" value extracted is %d \n", queue[front]);
      front = front + 1 ;
    }
    else
    {
      printf ("Cannot dequeue. Queue is empty !!.! \n");
    }
}

//isempty
int isempty ()
{
    if (rear == -1 || front > rear)
    {
      return 1;
    }
    else
    {
      return 0;
    }
}.
```

(01) Jhon's live in two stores house with two rooms, one bathroom, one kitchen and one living room. Jhon wants to go out by his car. Then he finds he can't remember the place that he kept the car key. Write a pseudo code for Jhone to find the car key.

```
Begin
    key (x) ;
    if x == Pocket then
        key is in Pocket
    elseif x == Room 1 then
        key is in Room 1
    elseif x == Room 2 then
        key is in Room 2
    elseif x == bathroom
        key is in bathroom
    elseif x == kitchen then
        key is in the kitchen
    else
        key is in a living room
End
```

What is an Algorithm

Series of steps break down from a specific task that carries out to overcome from a problem refers to can algorithm.

Searching

Find elements in large amounts of data.
Determine whether array contains value maching key value.

- Linear search.
- Binary search.

Linear Search

- Start at first element of array.
- Compare value to value (key) for which you are searching.
- Continue with next element of the array until you fing a match or reach the last element in the array.
- Note:- On the Average we have to compare the search key with half the elements in the array..

Characteristics of linear Search

- Data inside the array need to be stored.
- It easy to implement.
- Not suitable for large data set.
- We have 10 000 data sets.
- Best case :- we search one number it is in first, This is best case
- Wast case:- we search one number it is in End of data set. This is wast case.

Graphically represents how key 53 going to found in the following data set.

12	85	-6	0	36	74	62	53	11	9

Iteration 1	12	85	-6	0	36	74	62	53	11	9	
Iteration 2	12	85	-6	0	36	74	62	53	11	9	
Iteration 3	12	85	-6	0	36	74	62	53	11	9	
Iteration 4	12	85	-6	0	36	74	62	53	11	9	
Iteration 5	12	85	-6	0	36	74	62	53	11	9	
Iteration 6	12	85	-6	0	36	74	62	53	11	9	
Iteration 7	12	85	-6	0	36	74	62	53	11	9	
Iteration 8	12	85	-6	0	36	74	62	53	11	9	Found at position 7.

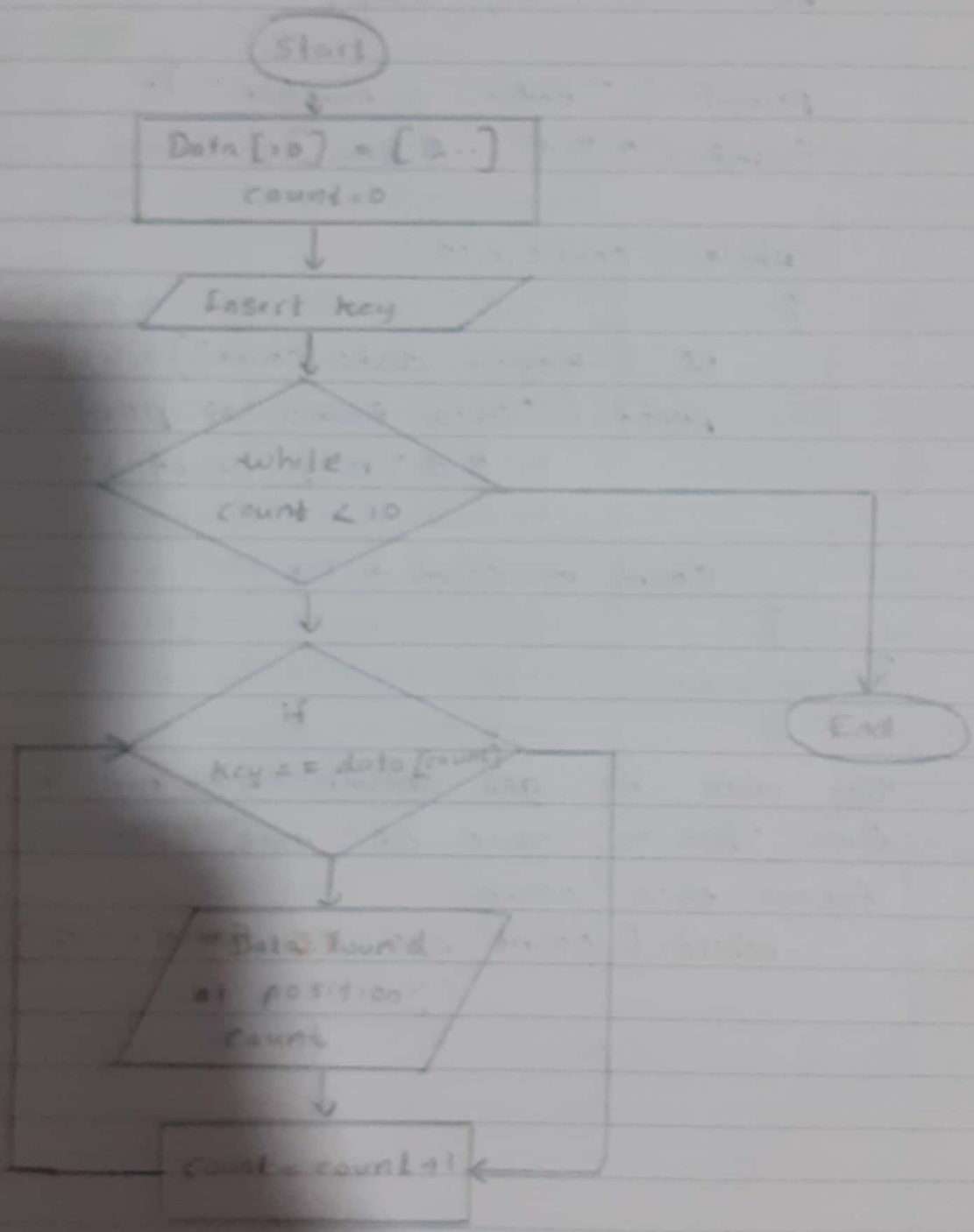
Draw a flow chart to find the position at an element of an integer array size 10 (Linear search flow chart).

Declare Dataset [10] = [. . . .]

Insert key

Set a loop for 10 times

- Check key == data [position] ?



```
#include <stdio.h>
int main()
{
    int data[10] = {45, 78, -98, 34, 87, 1, -8, 9, 14};
    int count = 0, key;

    printf("Enter a number");
    scanf("%d", &key);

    while (count < 10)
    {
        if (key == data[count])
        printf("Data found at position count
                %d \n", count);

        count = count + 1;
    }
}
```

This code, we get answer and run in unnessaary time. This is issue of this code we can fixed this issue.

```
while (count < 10 && Found == 0)
```

Completes the following trace table for Linear Search algorithm with following data

Data = { 45, 89, 63, -3, 34, 74, 11, 25, 100, 98 }

Key = 25

Count	while → Count < 10	→ if → Key == data[count]	→ Output →	Count = count + 1
0	0 < 10 true	25 == 45 false	–	0 + 1 = 1
1	1 < 10 true	25 == 89 false	–	1 + 1 = 2
2	2 < 10 true	25 == 63 false	–	2 + 1 = 3
3	3 < 10 true	25 == -3 false	–	3 + 1 = 4
4	4 < 10 true	25 == 34 false	–	4 + 1 = 5
5	5 < 10 true	25 == 74 false	–	5 + 1 = 6
6	6 < 10 true	25 == 11 false	–	6 + 1 = 7
7	7 < 10 true	25 == 25 true	Position 7	7 + 1 = 8
8	8 < 10 true	25 == 100 false	–	8 + 1 = 9
9	9 < 10 true	25 == 98 false	–	9 + 1 = 10
10	10 < 10 false	–	–	–

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

int main()
{
    int data[10] = { 49, 89, 63, -3, 34, 74, 11, 25, 100, 88 };
    int count = 0, key;
    int found = 0;

    printf("Enter number to search \n");
    scanf("%d", &key);

    while ( count < 10 && found == 0)
    {
        if ( key == data[count] )
        {
            printf("Data is found at position %d \n",
                                        count);
            found = 1;
        }
        count = count + 1;
    }
    printf("count value %d \n", count);

    If (found == 0)
    printf("data is not in the list");
    return 0;
}
```

20.2 Paper.

write down a source code to implement the linear search function (can use pseudo code or C language)

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

int main()
{
    int arr [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int count = 0,  key;
    int found = 0;

    printf ( "Enter number to search \n");
    scanf (" %d", &key );

    while ( count < 10  && found == 0 )
    {
      if ( key == arr [count] )
      {
          printf ("Data is found at position %d\n",
                                        count );
          found = 1;
      }
      Count = Count + 1
    }
}
```

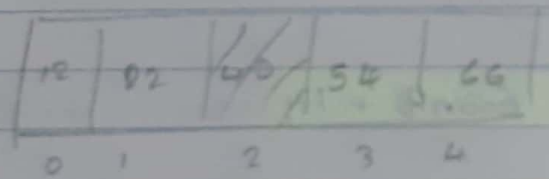
Binary Search Algorithm

- May only be used on a sorted array.
  Eliminates one half of the elements after each comparison.

- Binary search only works for sorted data sets

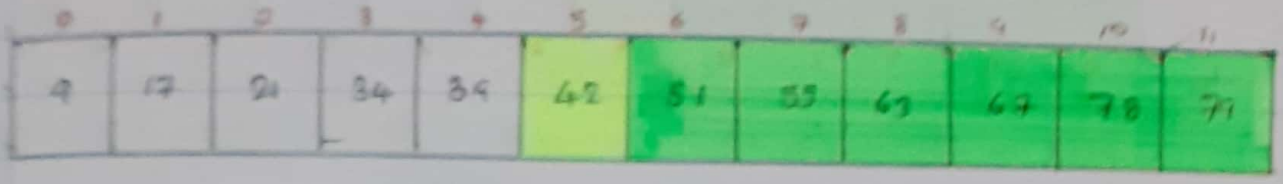
- It start searching numbers always from the middle of the array.

- If the value feature in the middle of the array position is find.
  Otherwise its needs to decide which half of the array needs to consider and did not he other half



Graphical Representation of Binary Search.

Find 78

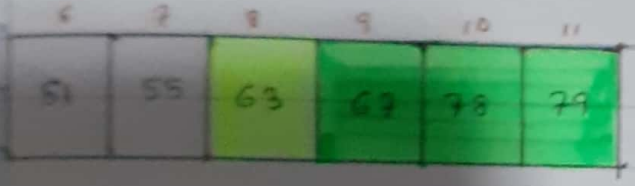


Mid = 0 + 11/2 = 5

Is found 78 == 42 → False

Is it < 42 → False

Is it > 42 (current mid) → true

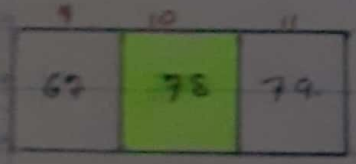


Mid = (6 + 11)/2 = 8

Is found 78 == 63 → false

Is it < 63 → false.

Is it > 63 (current mid) → true

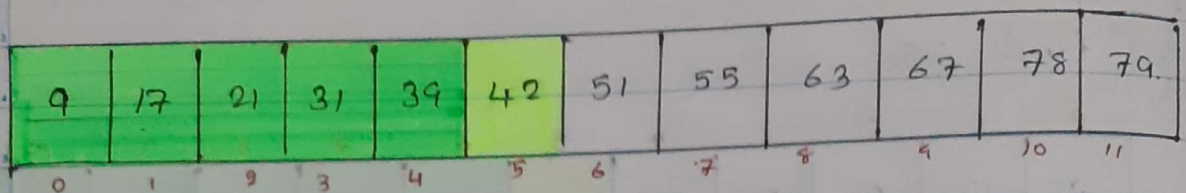


Mid = (9 + 11)/2 = 10

Is found 78 == 78 → true

We found it at index 10.

(02) Find 17.



Mid = (0+11)/2 == 5

Is found 17 == 42 ⟶ false.

Is it < 42 ⟶ True.

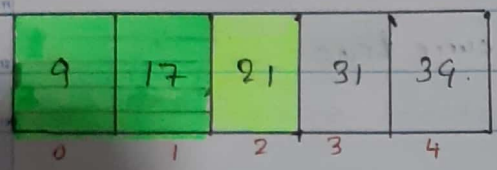
Is it > 42 ⟶ false.

Mid = (start + end)/2
= (0+11)/2 = 5

Are you 17? -- 42 == 17?
false

Are you smaller to 17?
42 < 17 ⟶ false

Are you greater to 17? 42 > 17
true.

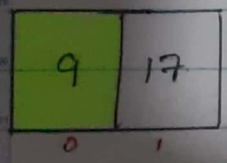


Mid = (0+4)/2 == 2

Is found 17 == 21 ⟶ False

Is it < 21 ⟶ True.

Is it > 21 ⟶ false.

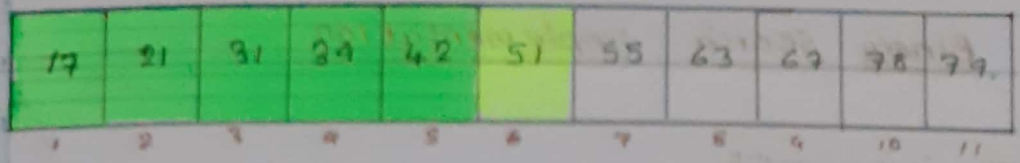


Mid = (0+1)/2 = 0

Is found 17 == 9 ⟶ false

Is it < 9 ⟶ false

Is it > 9 ⟶ True.

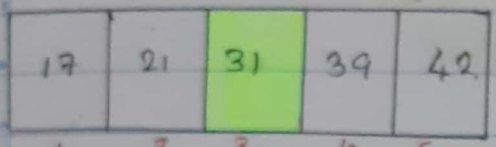


Mid = (1 + 11)/2 = 6

Is found 17 == 51 → false.

Is it < 51 → True.

Is it > 51 → false.

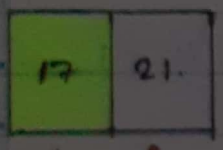


Mid = (1+5)/2 = 3

Is found 17 == 31 → false.

Is it < 31 → True.

Is it > 31 → false



Mid = (1+2)/2 == 1

Is found 17 == 17 → True

We found it at index 1 //

Binary Search implementation

- key components

  - Mid = (first index + last index) / 2.
  - If data [mid] == key ?

  - If key < data [mid]
    - Last = mid - 1

  - If key > data [mid]
    - Last = mid + 1

```
# include < stdio.h >
int main()
{
   int arr [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   int key, first = 0, last = 9, mid;

   printf (" Enter key that you want to search: ");
   scanf (" %d ", & key);

   while (first <= last)
   {
      mid = ( first + last ) / 2;
      if ( key == arr [mid] )
       {
         printf ("%d is in index %d", key, arr [mid] );
       }
      if ( key < arr [mid] )
       {
         last = mid - 1;
       }
      if ( key > arr [mid] )
       {
         first = mid + 1;
       }
   }
```

Section 03

Sorting Algorithms – Bubble Sort

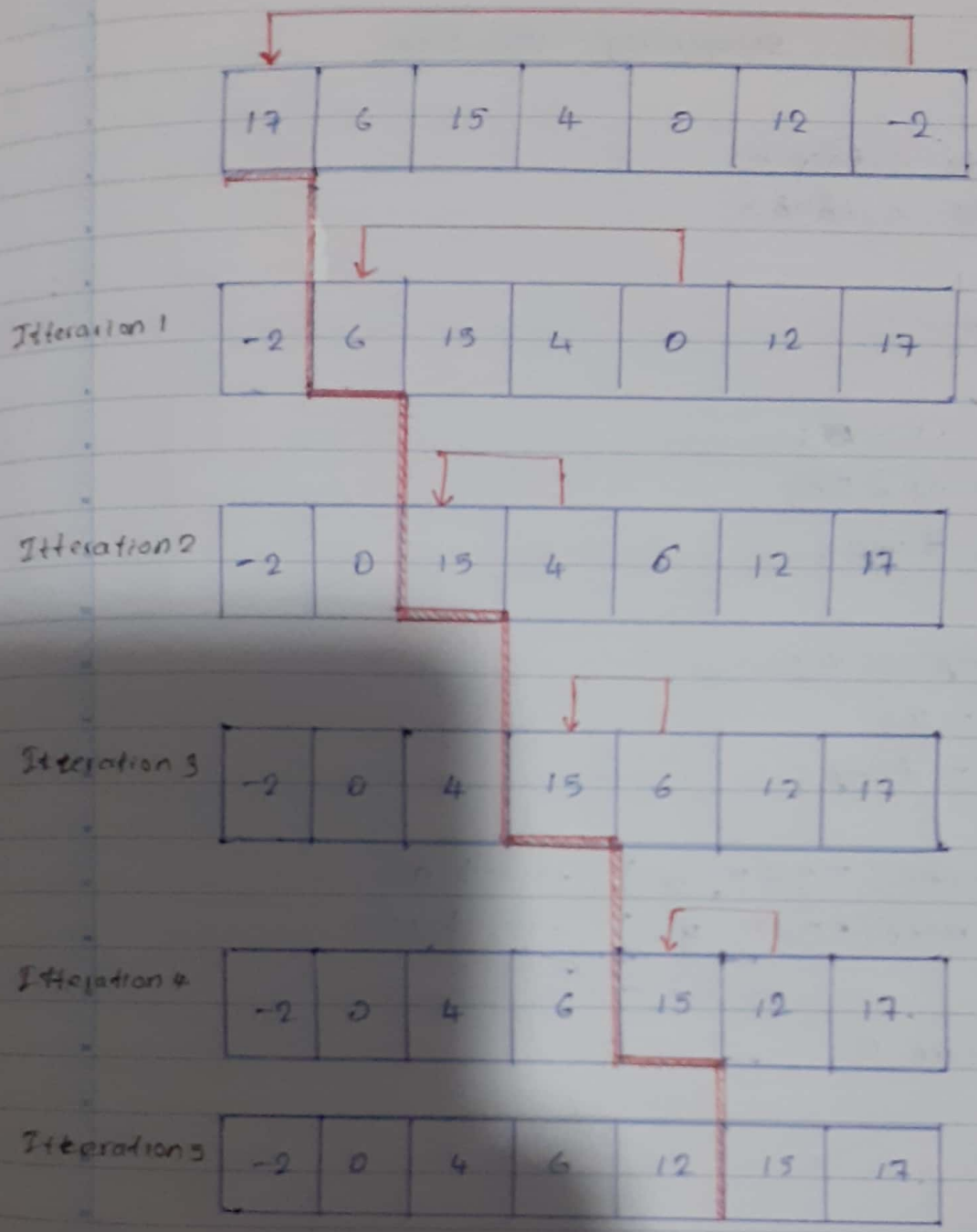
- Selection sort is one of the most simple sorting algorithm.

Introduction to Sorting Algorithms

- Sort: arrange values into an order:
  - Alphabetical
  - Ascending numeric
  - Descending numeric.
- Two basic algorithms considered here:
  - Bubble sort
  - Selection sort.

Bubble Sort

- Concept:
- Compare 1st two elements
  - If out of order, exchange them to put in order
- Move down one element, compare 2nd and 3rd elements, exchange if necessary. Continue until end of array
- Pass through array again, exchanging as necessary
- Repeat until pass made with no exchanges.



According to the above diagram,

(1) per each iteration its selecting current position to replace with the minimum value.

(2) Selecting the minimum value from the un sorted region

(3) Exchanging * with the current position

swapping Vanables

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

int main
{
    int x = 15;
    int y = 50;
    int temp;

    temp = x;
    x = y;
    y = temp;

    printf("%d", x) -----> 50
    printf("%d", y) ----> 15

    return 0;
}
```

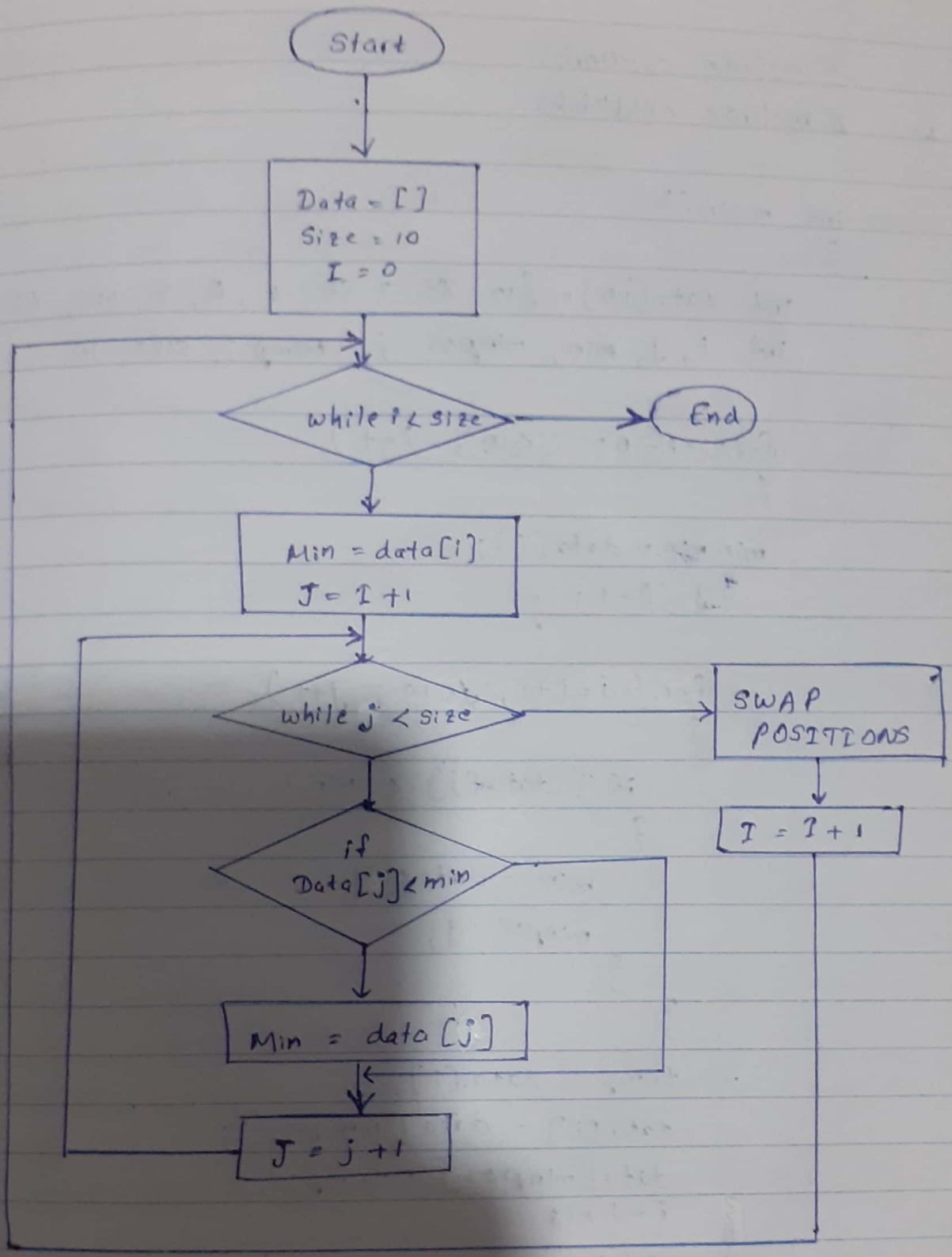
15	16	6	8	5

Selection Sort Implementation

Iteration 01	12	85	3	98	-4	0	7	100	10	31
Iteration 02	-4	85	3	98	12	0	7	100	10	31
Iteration 03	-4	0	3	98	12	85	7	100	10	31
Iteration 04	-4	0	3	98	12	85	7	100	10	31
Iteration 05	-4	0	3	7	12	85	98	100	10	31
Iteration 06	-4	0	3	7	10	85	98	100	12	31
Iteration 07	-4	0	3	7	10	12	98	100	85	31
Iteration 08	-4	0	3	7	10	12	31	100	85	98
Iteration 09	-4	0	3	7	10	12	31	85	100	98
Iteration 10	-4	0	3	7	10	12	31	85	98	100

For the selection sort we need two loops.

1. locate the next starting minimum
2. locate the actual minimum from the remaining date.

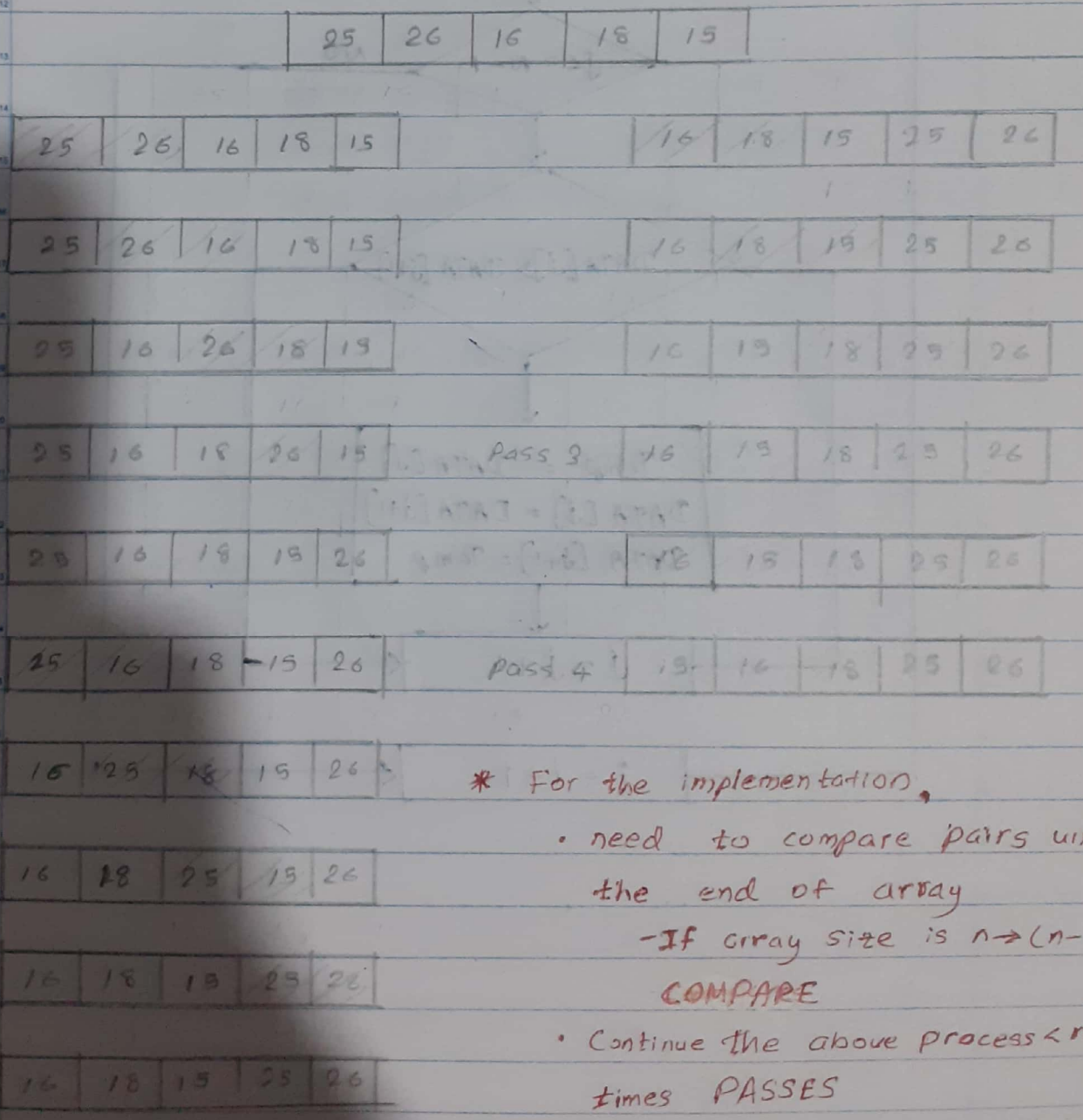


```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int data[10] = {12, 85, 3, 98, -4, 0, 7, 100, 10, 3};
    int i, j, min, minpos = j, temp, size = 10

    while (i < size)
    {
        min = data[i];
        j = i + 1;

        while (j < size)
        {
            if (data[j] < min)
            {
                min = data[j];
                minpos = j;
            }
        }
        temp = data[i];
        data[i] = data[minpos];
        data[minpos] = temp;
        i = i + 1;
    }
    printf("Sorted dataset\n");

    For (i=0; i<10; i++)
    {
        printf("%d\t", data[i]);
    }
    return 0;
}
```

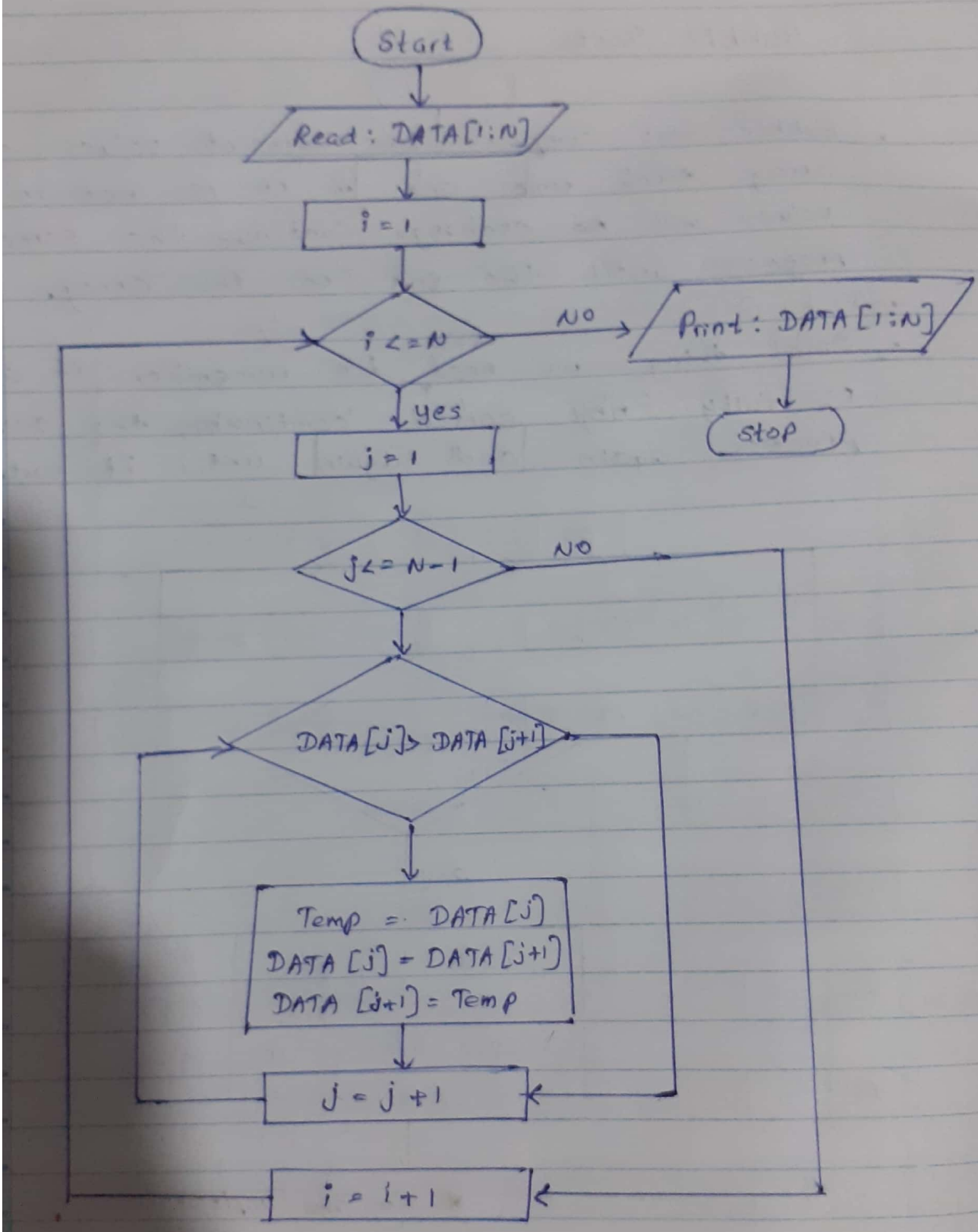
Bubble Sort

- Bubble Sort compairs two adjesent values of an array each time and if it is not in order values will be exchange. Continue the same progess until the end of the array.

- Final thing we need to remember if array is not fully sorted continuing the same process again and again until it sorted.



* For the implementation,
  - need to compare pairs until the end of array
    - If array size is $n \rightarrow (n-1)$ COMPARE
  - Continue the above process $< n$ times PASSES



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

int main()
{
    int data [5] = {9, 3, 0, 5, 2};
    int i, j, temp, size = 5;

    for ( i=0; i < size-1 ; j++ )
    {
        if (data [j] > data [j+1])
        {
            temp = data [j];
            data [j] = data [j+1];
            data [j+1] = temp;
        }
    }
    for ( i=0 ; i<size ; i++)
    {
        printf ("%d \t", data [i]);
    }
    return 0;
}
```

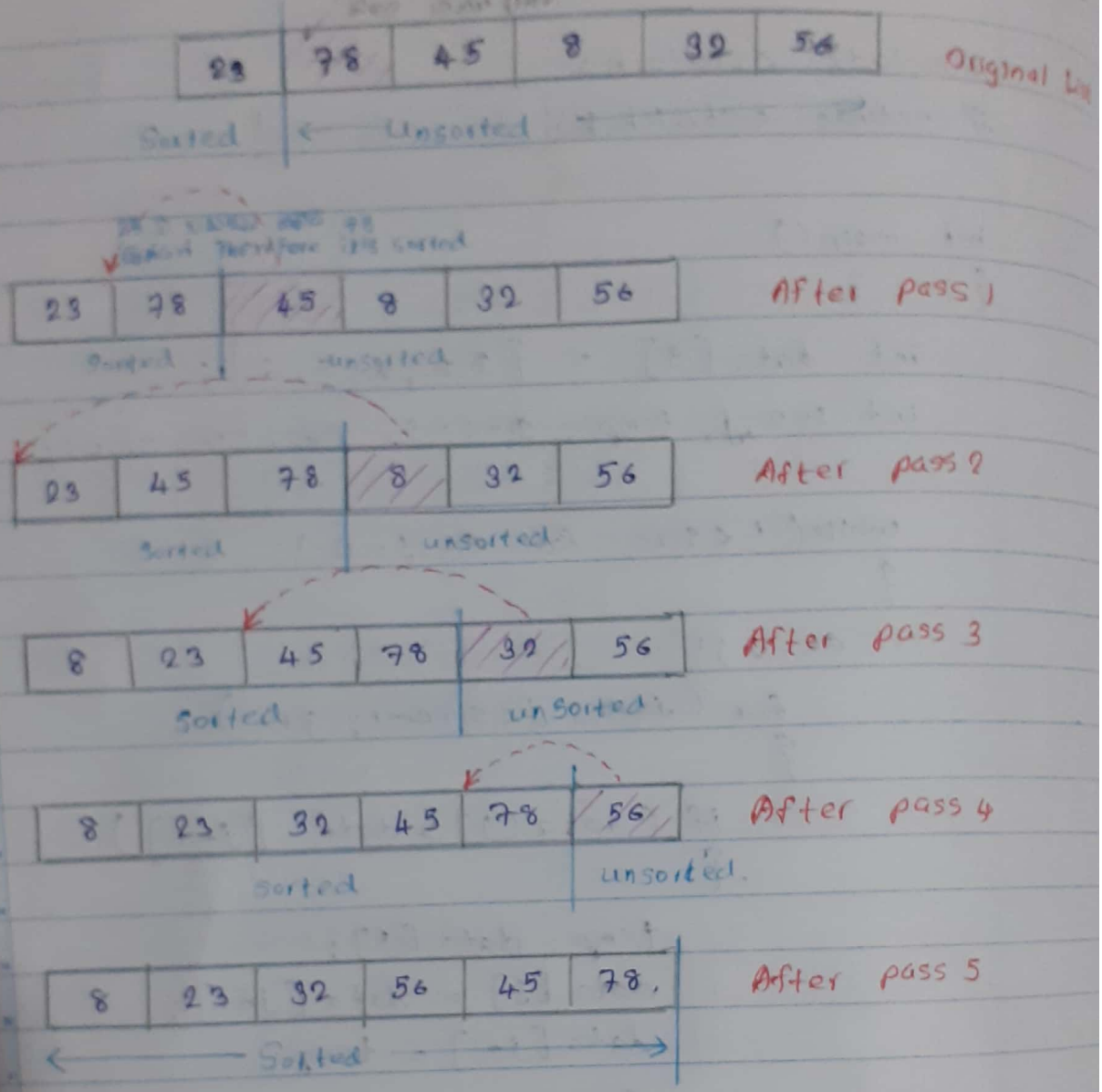
— Bubble Sort —

```
do {
    flag = false;
    // move through the array

    for (j=0; j<n-1; j++)
    {
        // if it's not in order
        if (arr[j] > arr[j+1])
        {
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
            flag = true;
        }
    }
} while (flag);
```

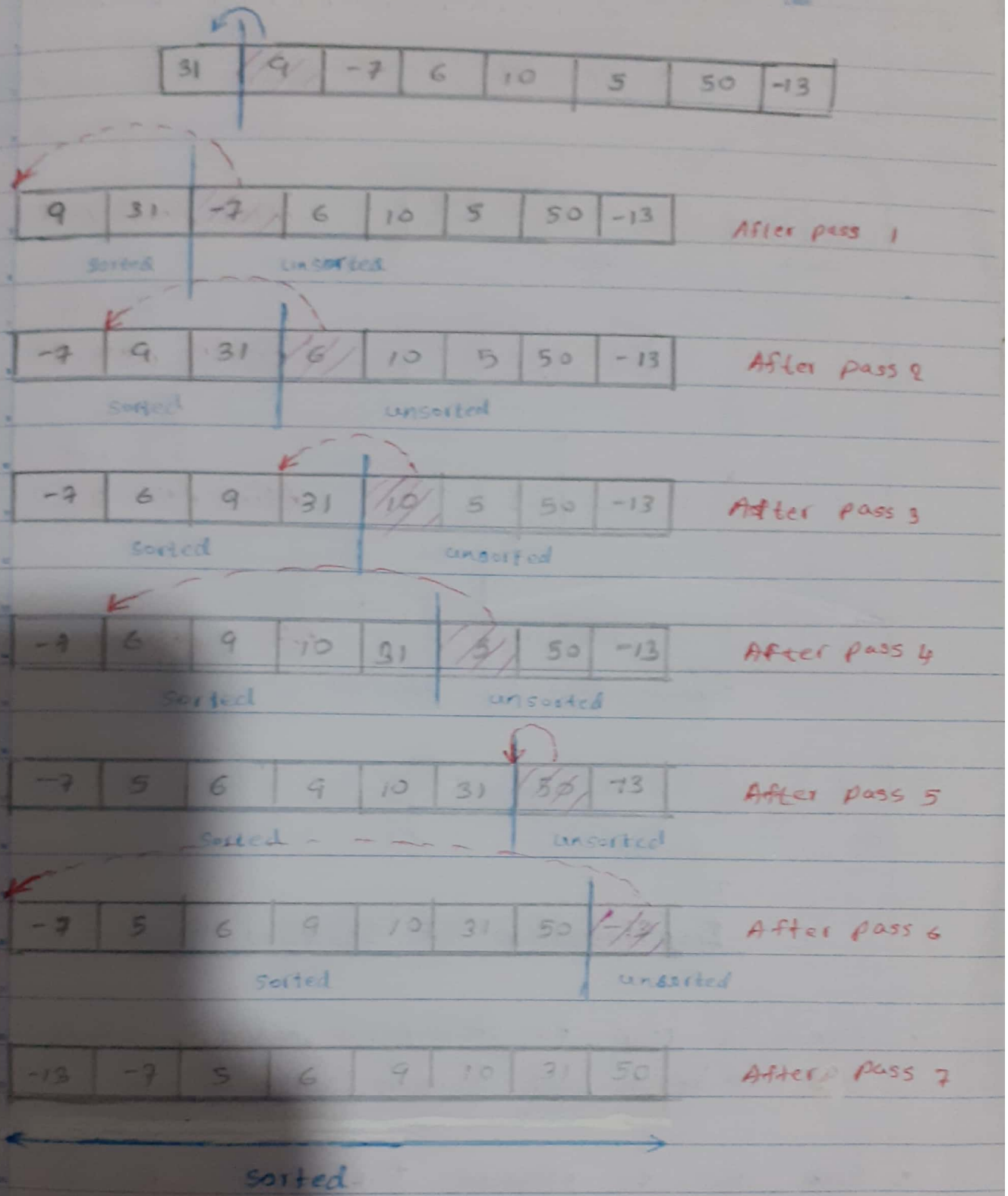
Optimizing bubble sort:

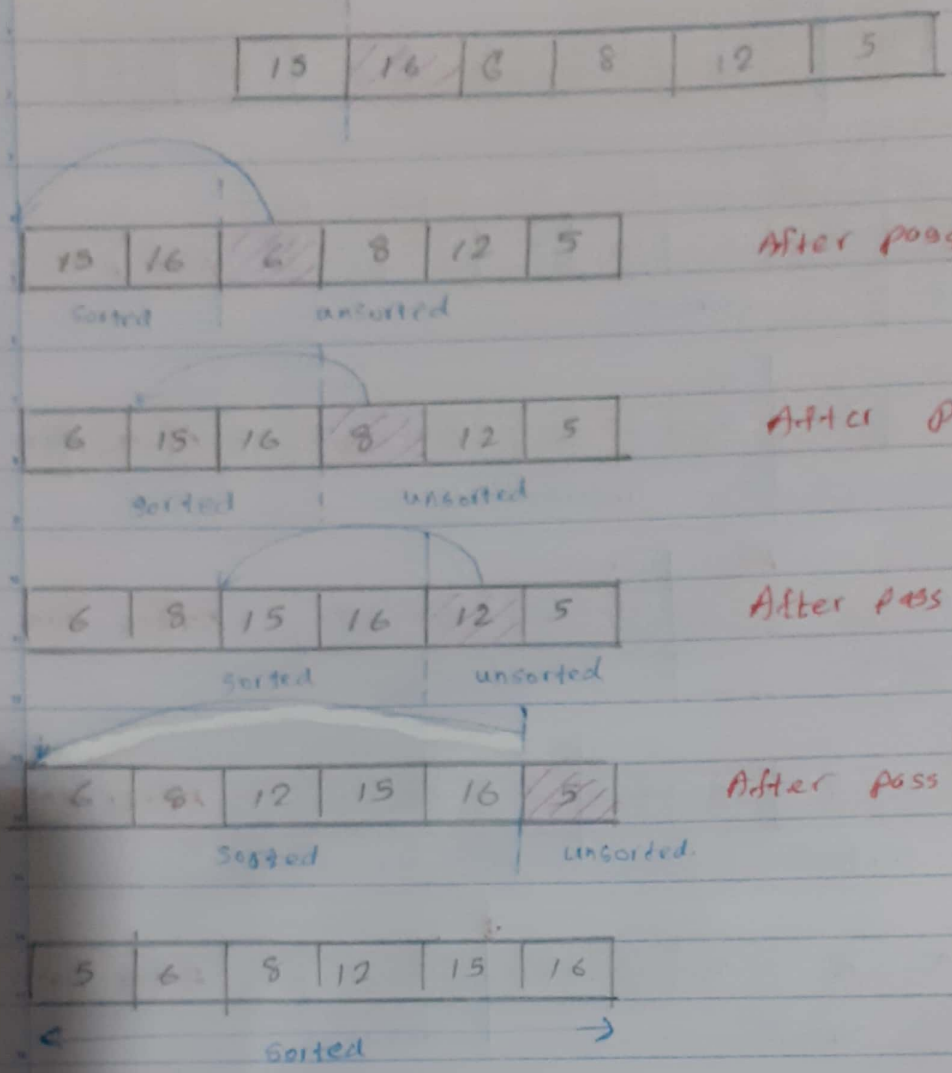
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int data[5] = {3,2,4,5,6};
    int i=0,j, temp, size=5, pass=0, ex=1;

    while( i<size && ex!=0)
    {
       ex=0;
         for (j=0; j<size-1; j++)
         {
            if (data[j] > data [j+1])
            {
               temp = data[j];
               data[j] = data[j+1];
               data[j+1] = temp;
               ex = ex+1;
            }
         }
       pass = pass+1;
       i = i+1;
    }
    for(i=0;i<size;i++)
    {
       printf("%d \t", data[i]);
    }
    printf("\n Number of passes %d \n", pass);
```

Insertion Sort



It devides the region into two divisions as sorted and unsorted. Then from element to element select elements from unsorted regions and inserting it to the exact correct position in the sorted region.





Implementation of Insertion Sort.

- Need a loop to start from second element and and to reach last element one by one.
- Now have to compare this vale with the elements in the sorted region.

If data < unsorted value → increment position by 1

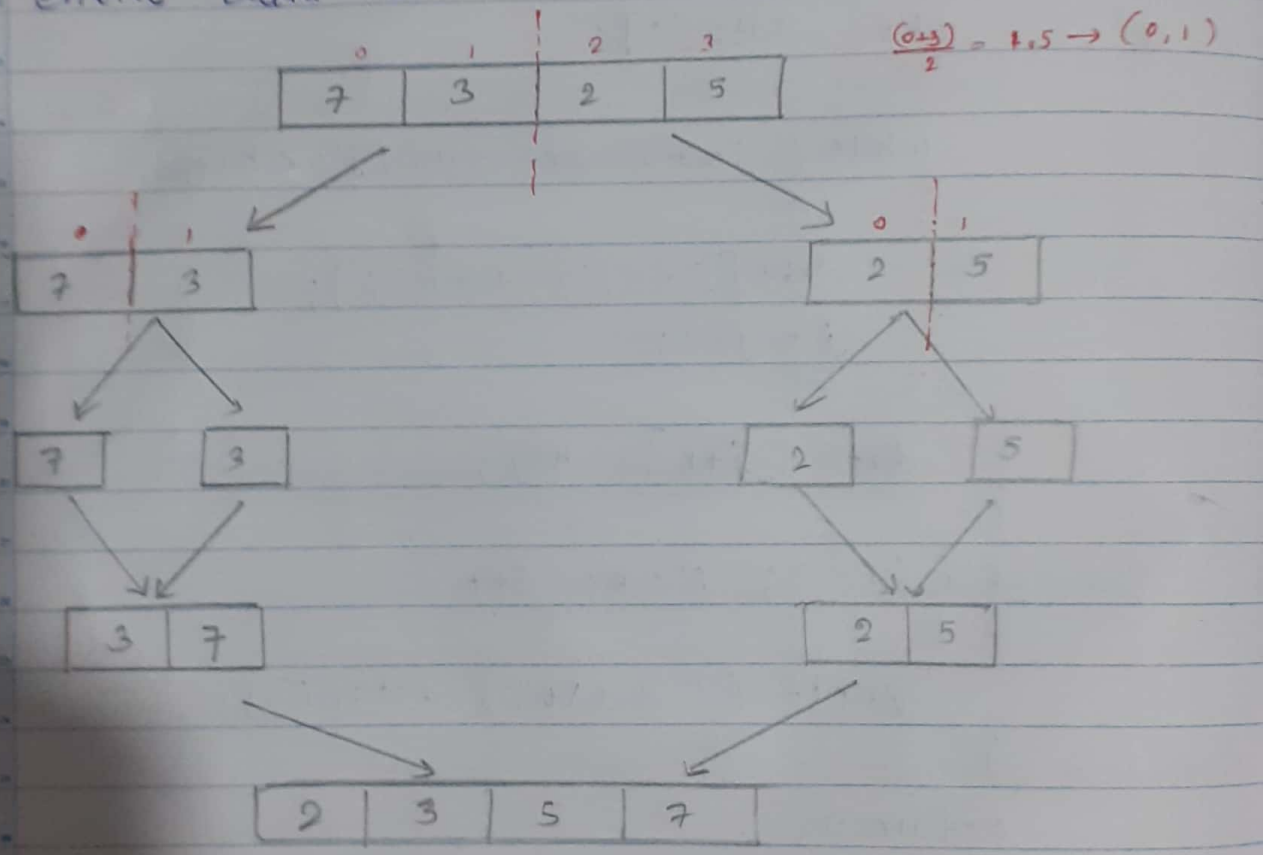
If not stop the comparison and place it where it's stoped.

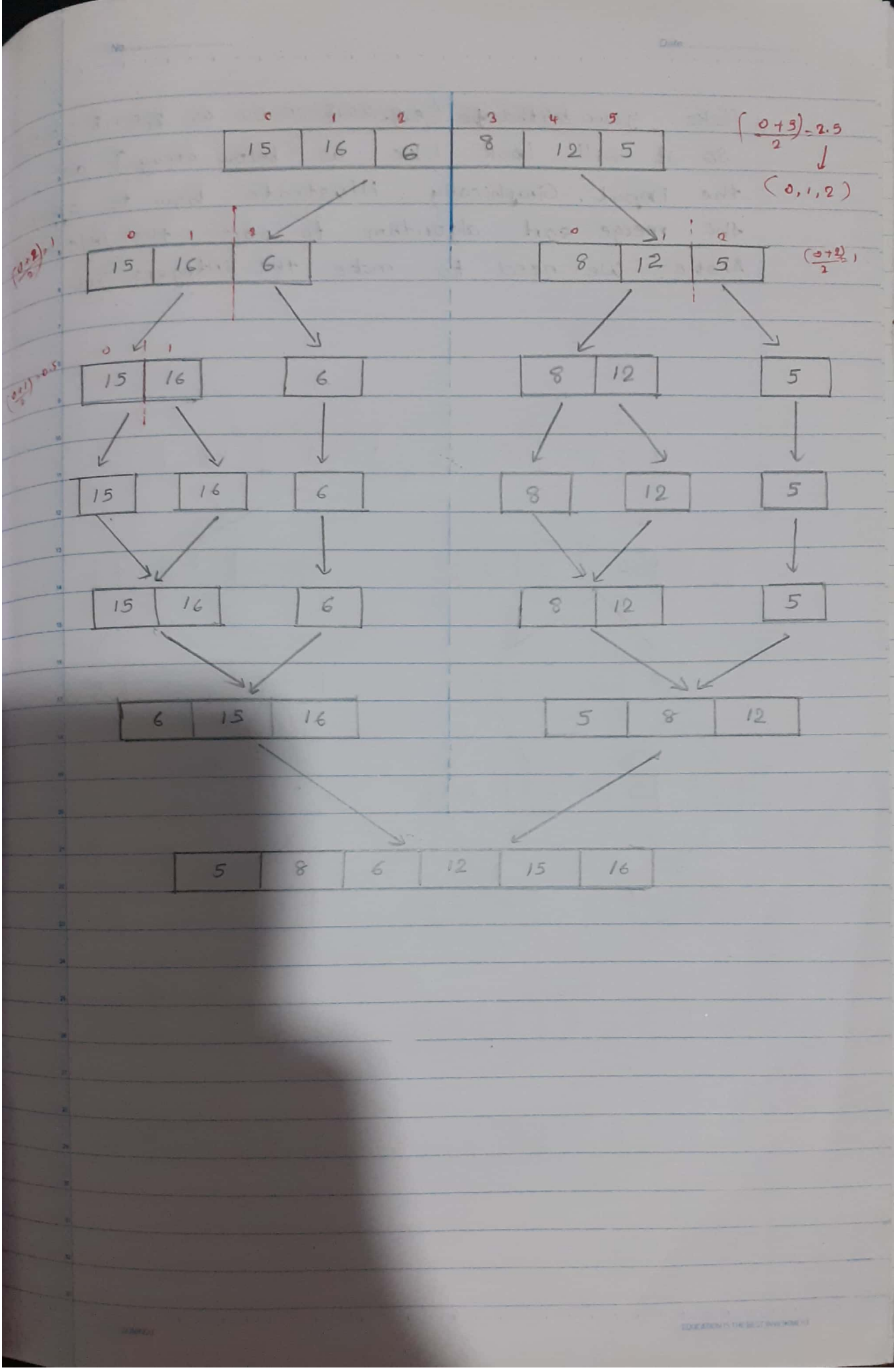
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int data[6] = {15, 16, 6, 8, 12, 5};
    int i, j, key;

    for (i=1; i<6; i++)
    {
        j = i-1;
        key = data[i];

        while (j>=0 && data[j] > key)
        {
            data[j+1] = data[j];
            j = j-1;
        }
        data[j+1] = key;
    }
    for (i=0; i<5; j++)
    {
        printf ("%d\t", data[i]);
    }
    return 0;
}
```

Merge Sort

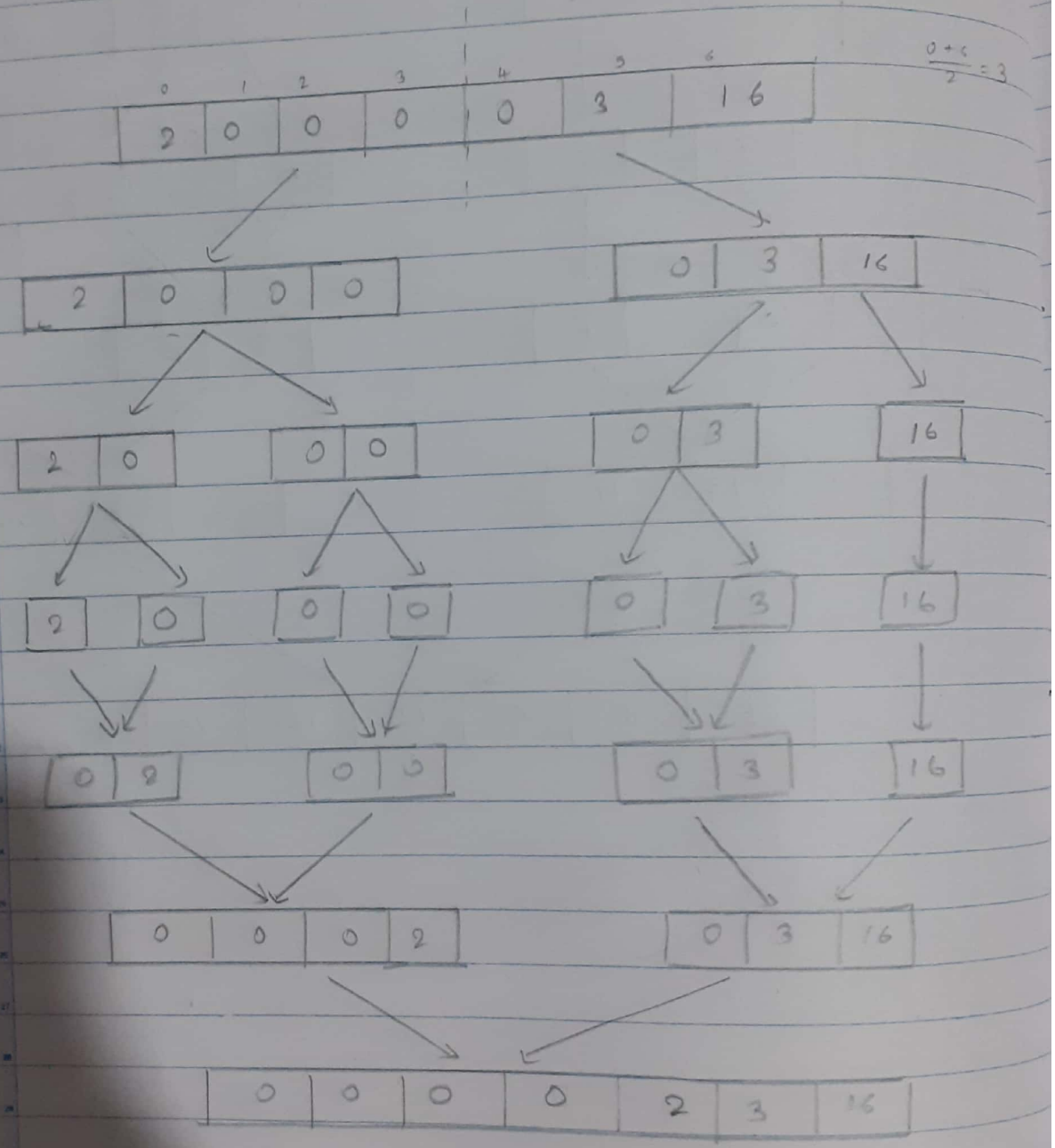
In merge sort the algorithm b aking down the given data sets into two subsets per each iteration until each element is an isolated subset.

now comparing elements by adjesing and sorting them in correct order and merge this subsets until it creats the entire data set





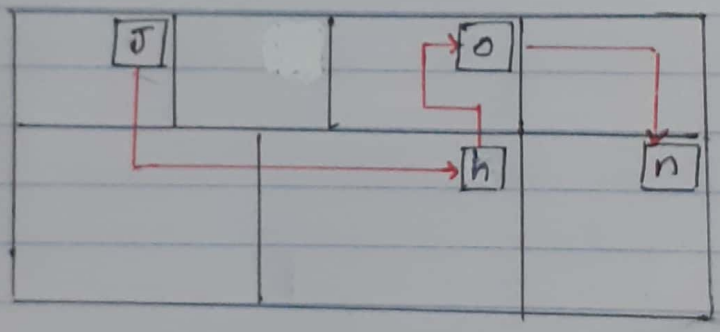
Take your birthdate (e.g. 2018-10-02 as 2,0,1,8,1,0,0,2 so it will look like as below array] as the input. Graphically illustrate how to apply the merge sort algorithm to sort the integer set. Note: We need to make the integers set.



Algorithmic Analysis

Linked Lists

- A linked list is a sequence of data structures, which are connected together via links.
- Linked List is a sequence of links which contains items.
- Each link contains a connection to another link.
- Linked list is the second most-used data structure after array.

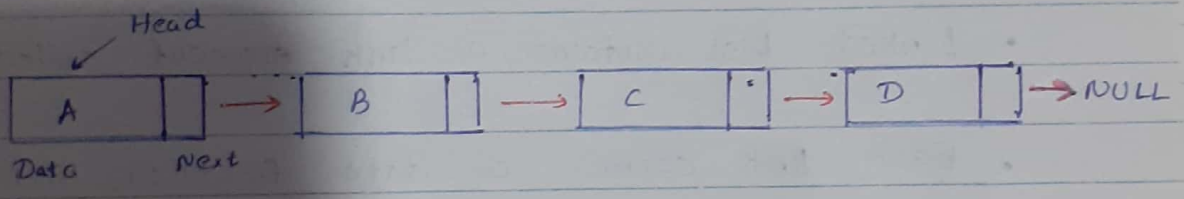


- In here we are storing data in different locations and not continuous places. (Storing data in non contigious storing areas)
- Then we are having a link to other position.
- Then from that link we can get the next value to till the end of the value we have stored.

Introduction

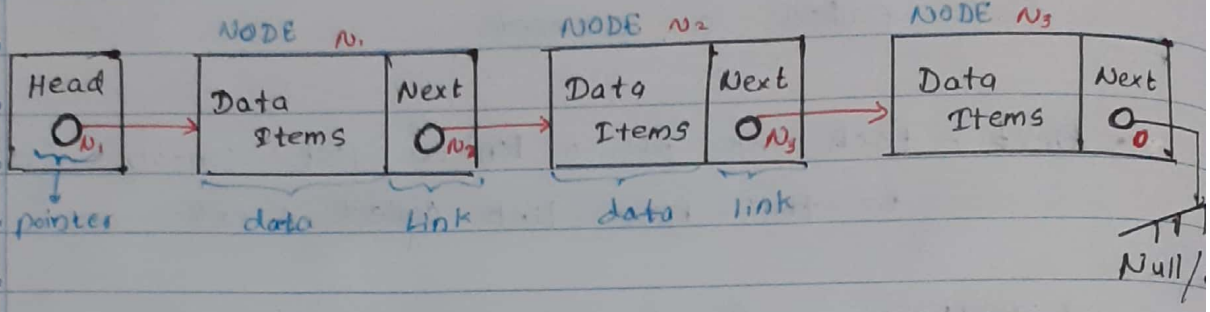
following are the important terms to understand and concept of Linked List.

- Link - Each link of a linked list can store a data called an element.
- Next - Each link of a linked list contains a link to the next link called Next
- LinkedList - A Linked list contains the connection link to the first link called First



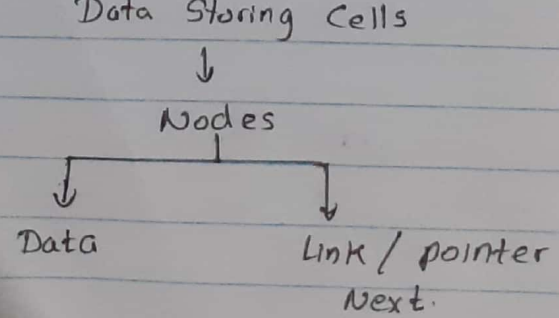
Graphical Representation

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 / head
- 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.

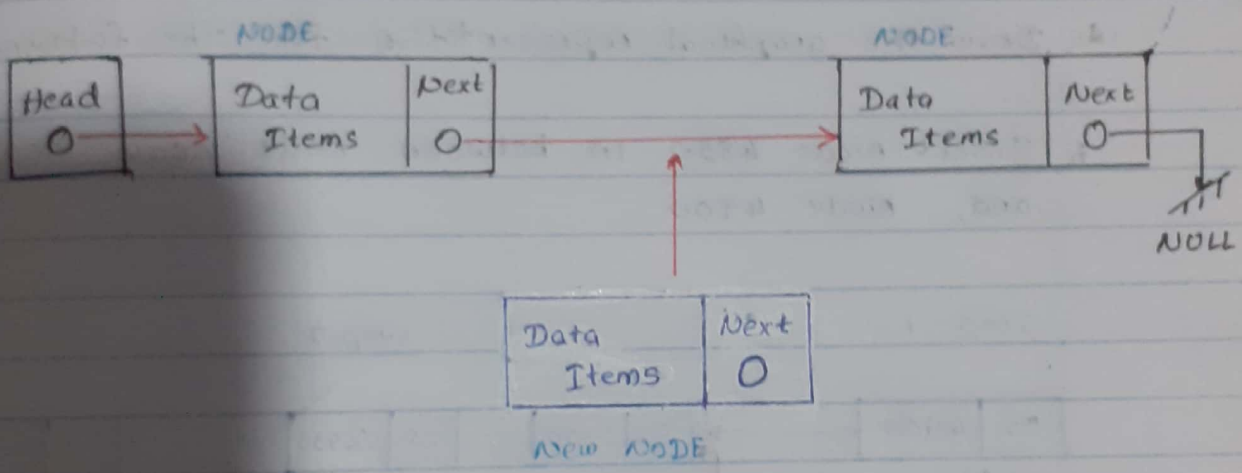


Insertion

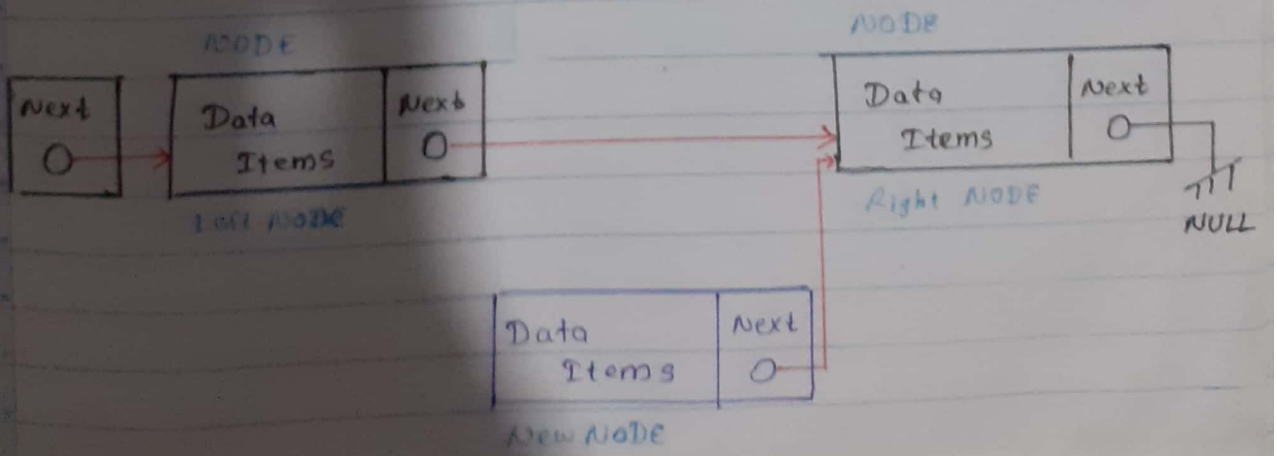
Insertion of a new Node in between two nodes

Adding a new node in linked list is a more than one step activity. Refer to the diagram,

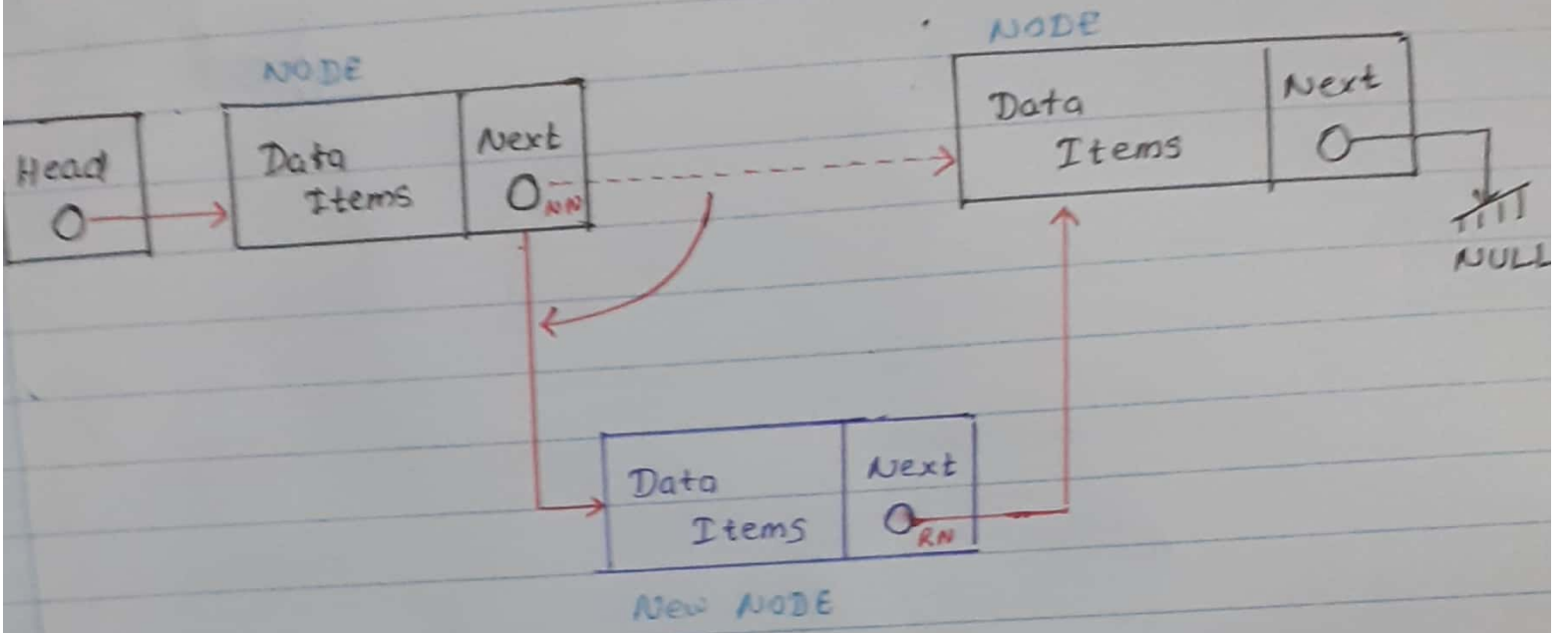
First, create a node using the same structure and find the location where it must be inserted.



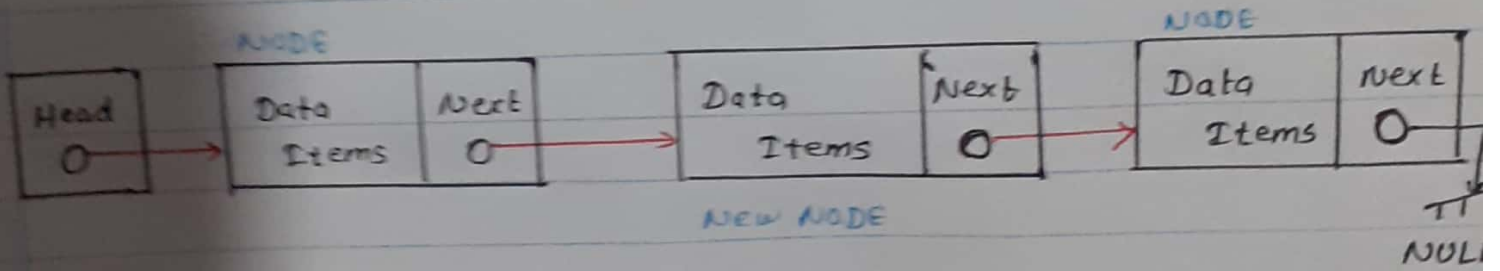
Imagine that we are inserting a node NewNode, between LeftNode and RightNode. Then point NewNode. next to rightnode.



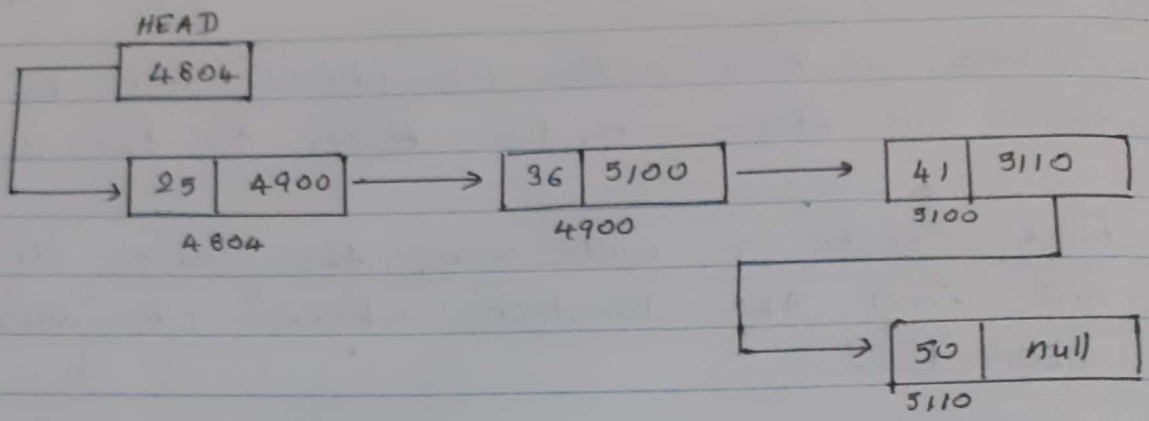
Now, the next node at the left should point to the new node.



This will put the new node in the middle of the two. The new list should look like this,



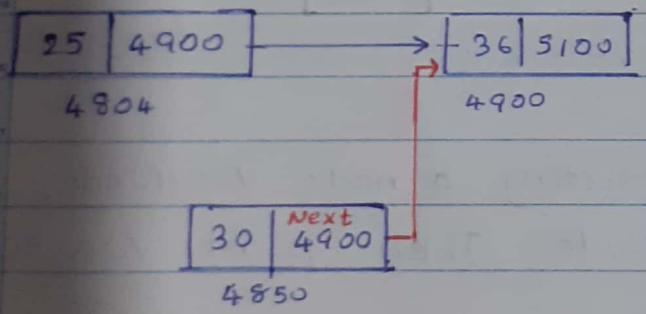
(Q1) Consider the following linked list and answer the following questions



(a) Draw the graphical representation for the following option

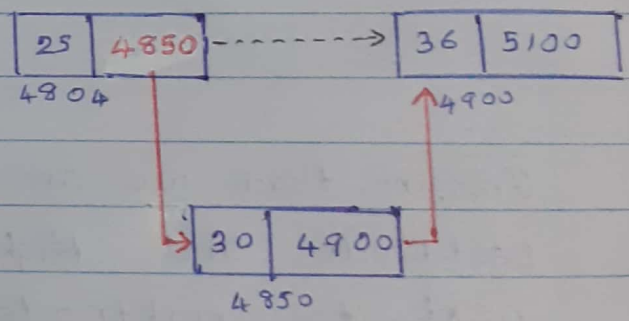
(i) Insert Node 4850 in between node 4804 and, Node 4900

Step 1



4850. Next → 4900

Step 2

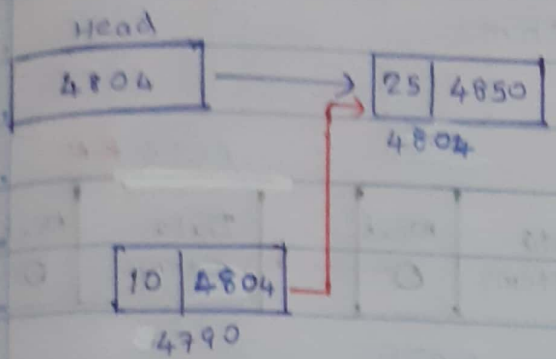


4804. Next → 4850

(ii) Insert a new Node 4790 after head befor

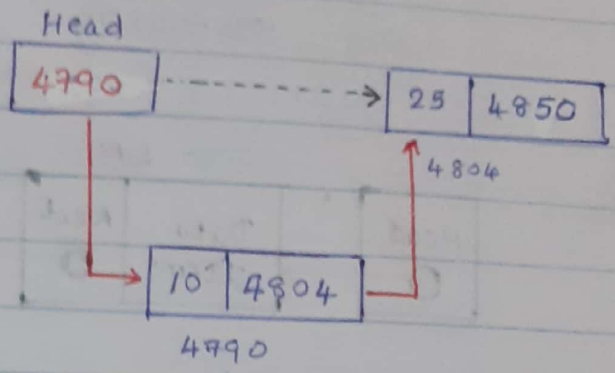
4804 [10 |]

Step 1



4790.Next → 4804

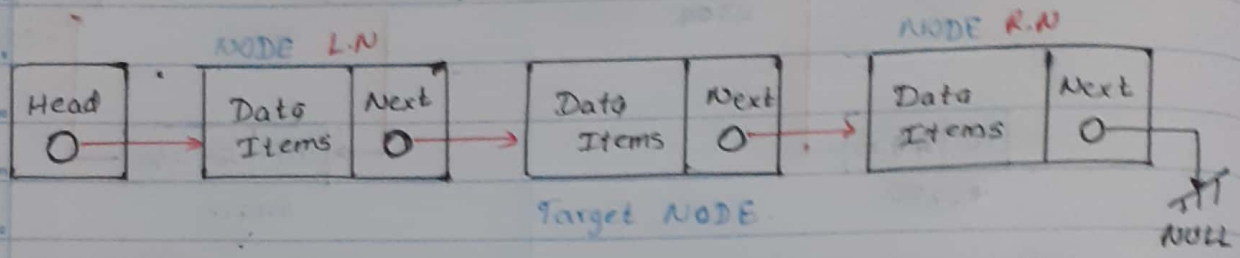
Step 2



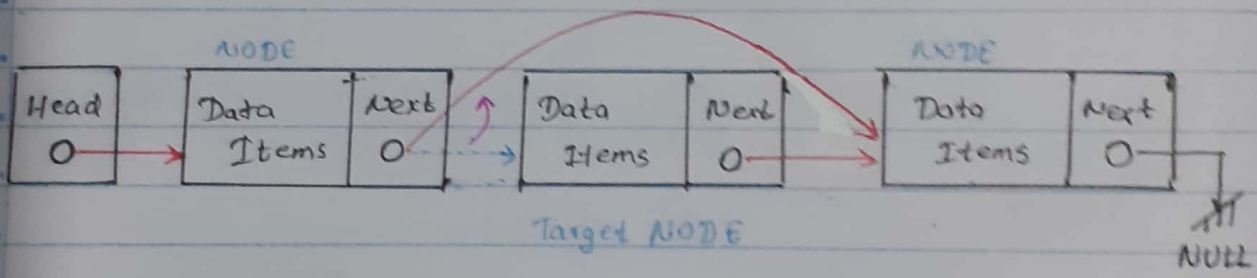
Head.Next → 4790

Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First locate the target node to be removed, by using searching algorithms.

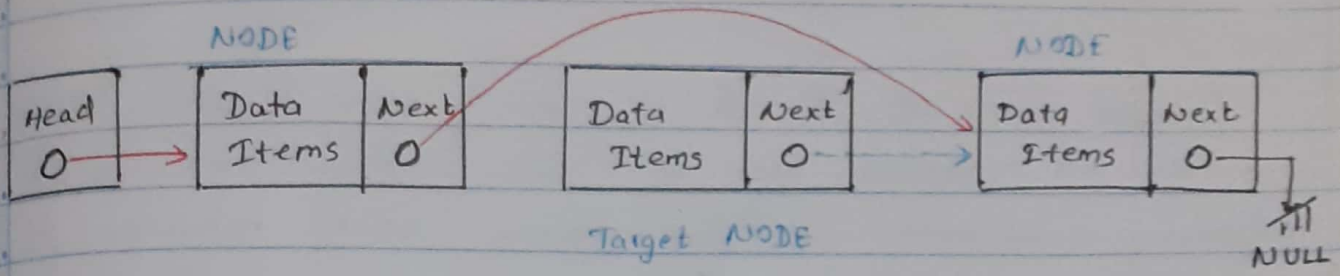


The left (previous) node of the target node now should point to the next node of the target node.

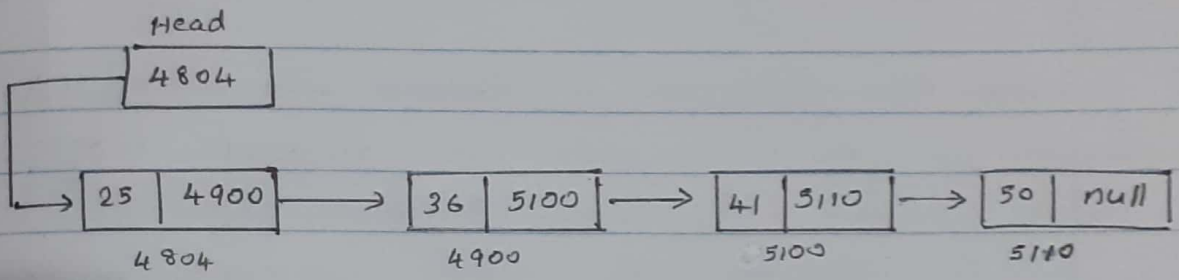


This will remove the link that was pointing to the target Node. Now, using the following code, we will remove what the target node is pointing at.

LeftNode.next ⟶ NULL



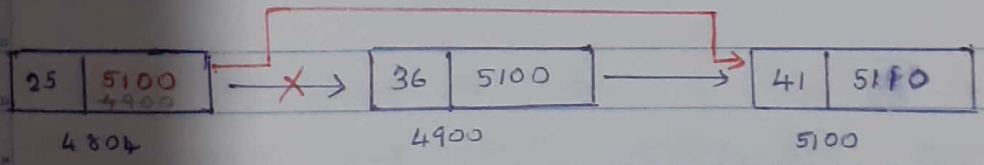
Question 01



(1) Delete Node 4900

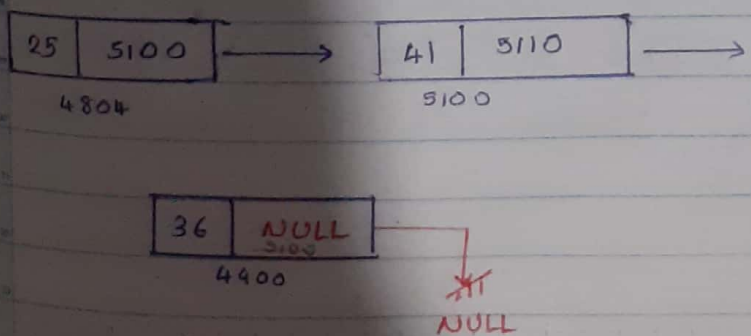
Step 1

4804.Next ⟶ 5100



Step 2

4900.Next ⟶ NULL



(2) Delete Node 5110

Why Linked List ?

Arrays can be used to store linear data of similar types, but arrays have a following limitations

- The size of the arrays is fixed: So, we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.

- Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

For example, in a system, if we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved,

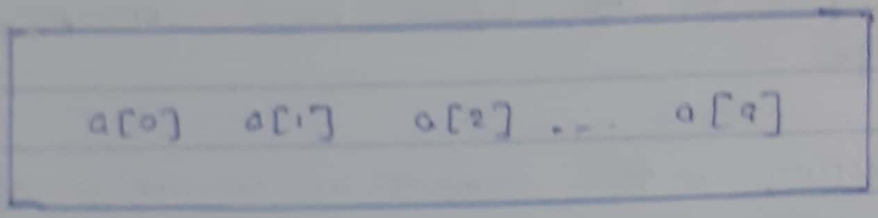
Array Vs Linked List

We've seen arrays :

```
int [] my_array = new int [10];
```

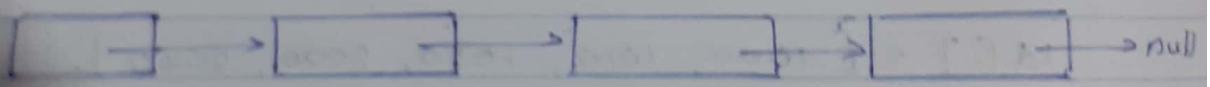
my_array is a chunk of memory of size 10 * size of (int)

my_array has a fixed size.



A linked list is fundamentally different way of storing collections

each element stores a reference to the element after it



Arrays

- have a pre-determined fixed size
- easy access to any element a[i] in constant time
- no space overhead

Linked Lists

No fixed size; grow one element at a time
space overhead.
Each element must store an additional reference
No easy access to i-th element should start with the head of the list.

need to hop through all previous elements

* One disadvantage of using arrays to store data is that arrays are static structures and therefore cannot be easily extended or reduced to fit the data set. (cannot change data)

* A linked list is a dynamic data structure (we can insert data at anypoint of the list at anytime)

* The number of nodes in a list is not fixed and can grow and shrink on demand

* Any application which has to deal with an unknown number of objects will need to use a linked list.

Introduction to Recursion and Recursive Algorithms

```
void main()
{
    int x = 10;
    printf ("%d", x);
}
```

```
void A (int a)
{
   printf ("%d", a);
}
```

```
Void main()
{
    int x = 10;
    x++;
    A(x);
}
```

```
void A(int a) {
        B(2 * a);
}
```

```
void B(int b) {
        printf ("%d", b);
}
```

```
Void main() {
        int x = 10;
        A(x+1);
}
```

Introduction to Recursion

A recursive function is a function that calls itself

Recursive functions can be useful in solving problems that can be broken down into smaller or simpler subproblems of the same type.

A base case should eventually be reached, at which time the breaking down (recursion) will stop.

Recursive Functions

All recursive functions there should be a starting value and there should be a condition that its stop where.

Consider a function for solving the count-down problem for some number `num` down to `0`:

* The base case is when `num` is already `0`: the problem is solved and we "blast off!"

* If `num` is greater than `0`, we count of `num` and then recursively count down from `num - 1`.

Recursive Functions : Base case / Recursive call

A recursive function for counting down to 0;

```
void countDown (int num)
{
    if (num == 0)
        print Done!;

    else
    {
        print n;
        countDown (num-1); // recursive call
    }
}
```

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

void countdown (int x)
{
    if (x==0)
    {
        printf ("\n\n Done Printing.\n");
    }
    else
    {
        printf ("%d\t", x);
        countdown (x-1);
    }
}
int main()
{
    countdown (10);
    return 0;
}
```

Output

10 9 8 7 6 5 4 3 2 1

Done Printing.

Write a program in C to find the factorial of a number using recursion
(5! = 5 * 4 * 3 * 2 * 1 = 120)

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

int factorial (int x)
{
    if (x==0)
    {
        return 1;
    }
    else
    {
        return x * factorial (x-1);
    }
}
int main()
{
    printf ("Factorial is %d \n", factorial(5));
    return 0;
}
```

Types of Recursion

* Direct recursion
  - A function calls itself

* Indirect recursion
  - Function A calls function B, and function B calls function A. or,
  - Function A calls Function B, which calls...., which calls function A.

(02) Write a program in C to print even or odd numbers in given range using recursion.

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

void even ( int first, int last )
{
   if (first > last)
   {
     printf ("\n\n Done. \n");
   }
   else
   {
        if ( first % 2 ==0 )
        {
          printf ("%d \t ", first );
          even(first +2 , last );
        }
        else
        {
          first = first +1;
          printf ("%d \t ", first );
          even (first +2 , last );
        }
   }
int main()
{
  even( 2, 12);
  return 0;
}
```