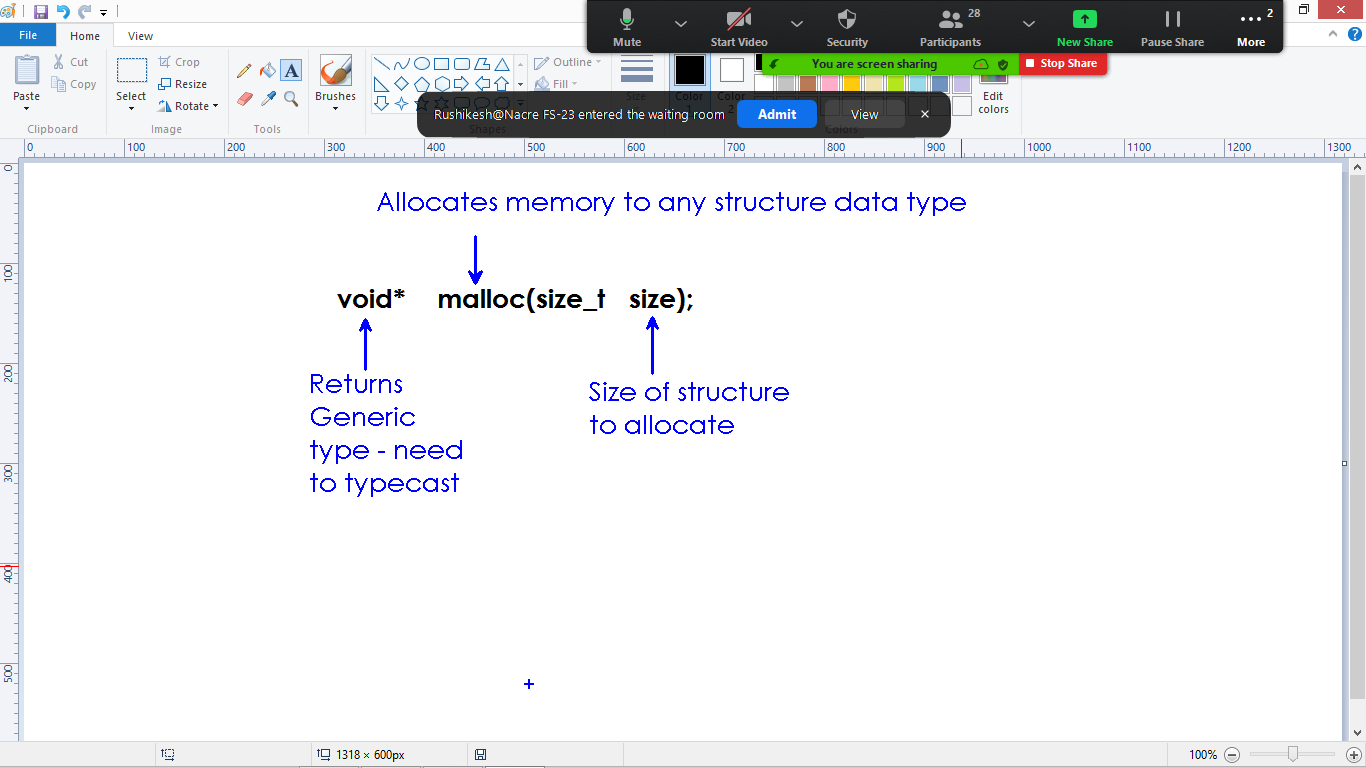
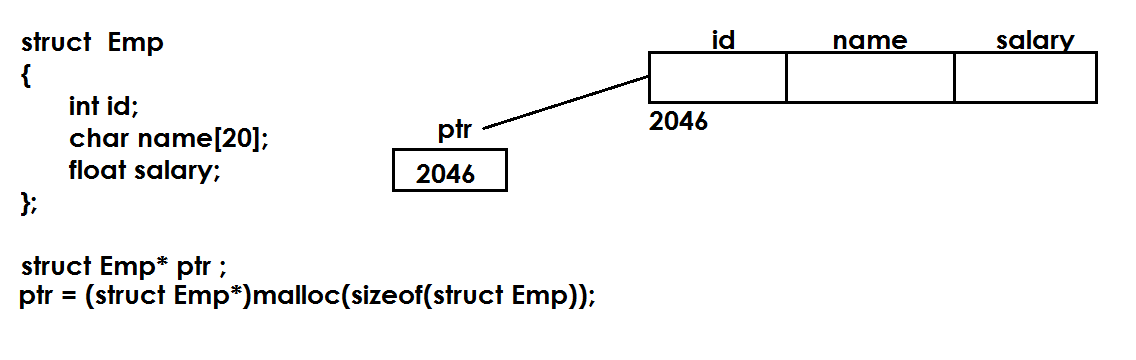
malloc( ):

* It is belongs to stdlib.h
* It is used to allocate memory dynamically to structures.
* Function prototype is
  + void\* malloc(size\_t size);
* It takes structure size as input to allocate memory
* On success, it allocates memory and returns base address of allocated block.
* On failure, it returns NULL pointer





**Accessing locations:**

* Once we allocate the memory to structure, we can store and process the information.
* Generally we access the structure information using dot(.) operator.
* When a pointer is pointing to structure, we use arrow( -> ) operator to store and process the data.

#include<stdio.h>

#include<stdlib.h>

struct Emp

{

int id;

char name[20];

float salary;

};

int main()

{

struct Emp\* ptr;

ptr = (struct Emp\*)malloc(sizeof(struct Emp));

if(ptr==NULL)

{

printf("Out of memory \n");

}

else

{

printf("Enter Emp details : \n");

scanf("%d%s%f", &ptr->id, ptr->name, &ptr->salary);

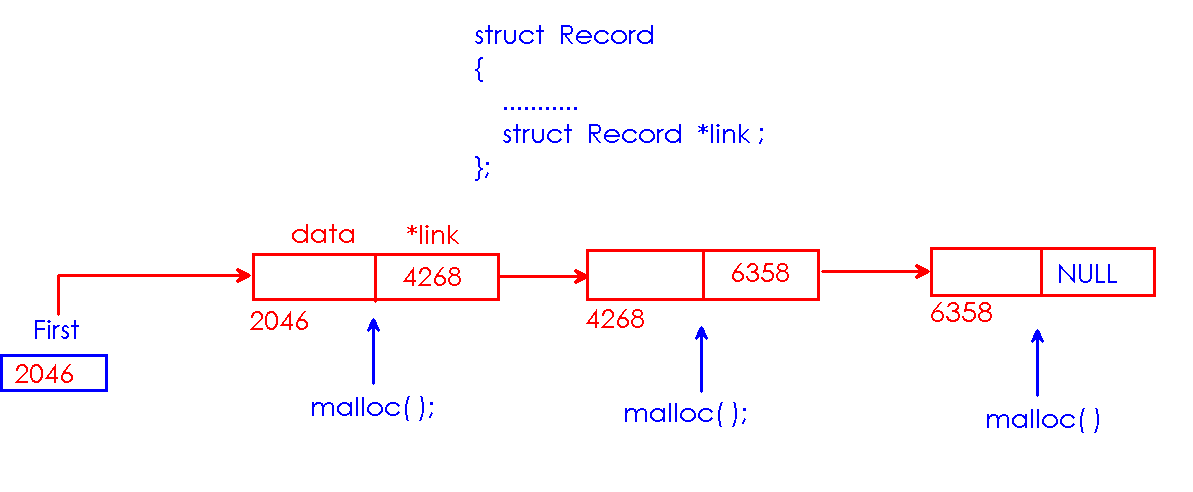
printf("Name : %s \n", ptr->name);

}

return 0;

}

**Allocate memory to linked list nodes:**

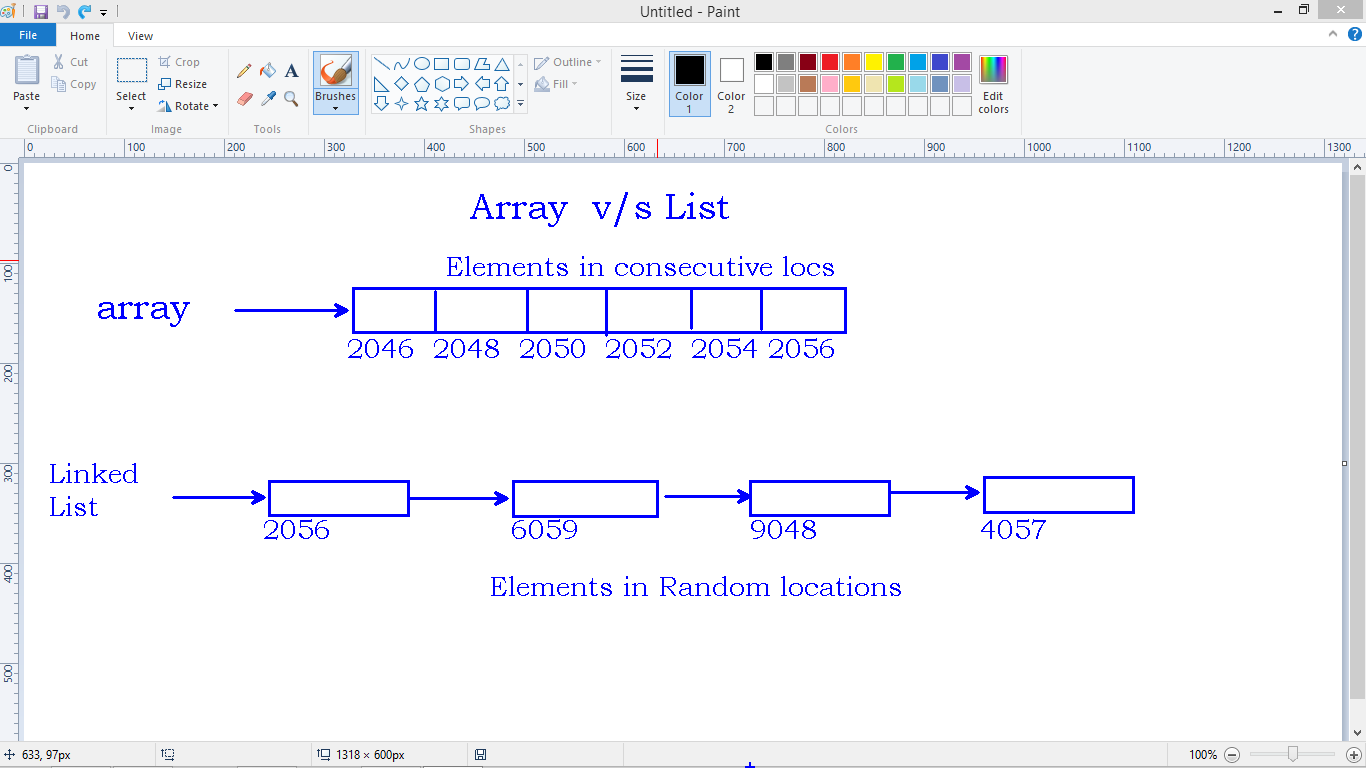


**Linked Lists**

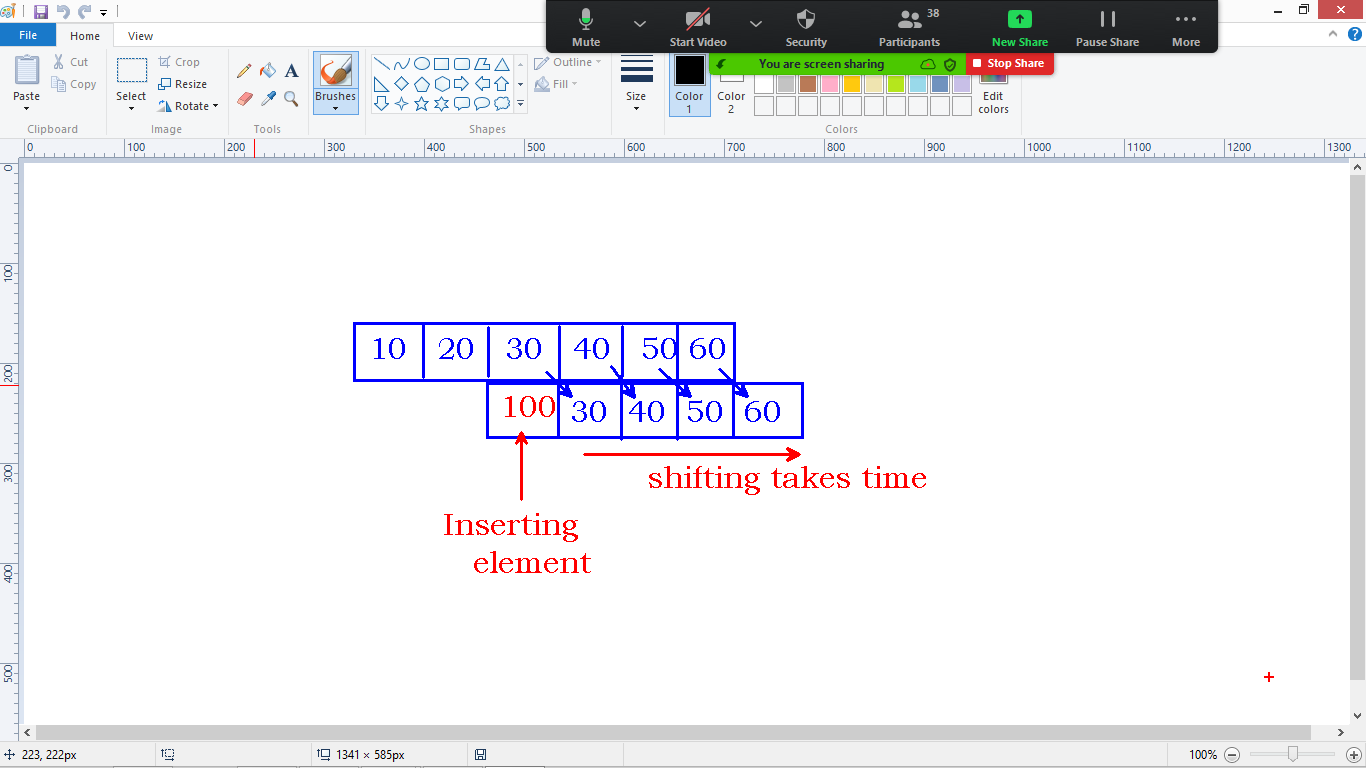
**Linked List:**

* It is a linear data structure.
* In linear data structure one element is connected to only one element.

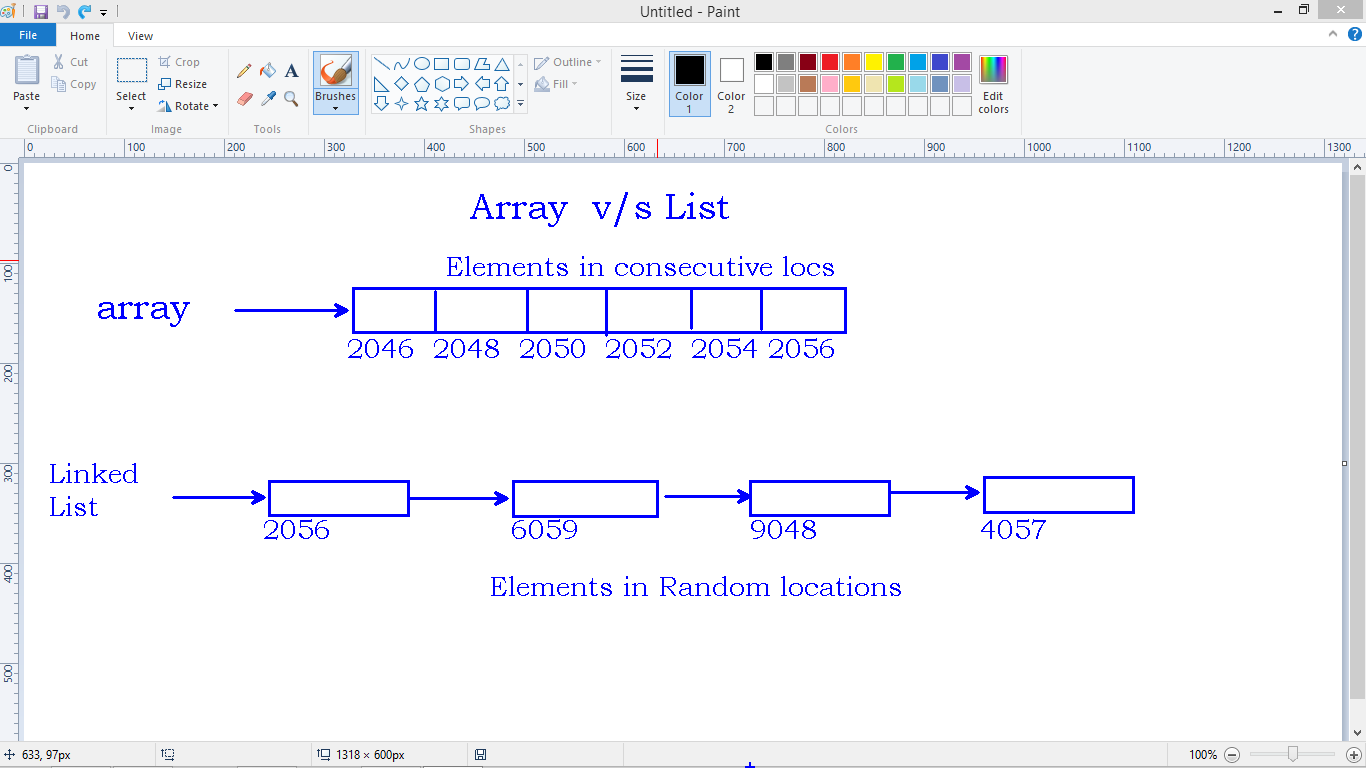
**Differentiate Arrays from Linked Lists?**

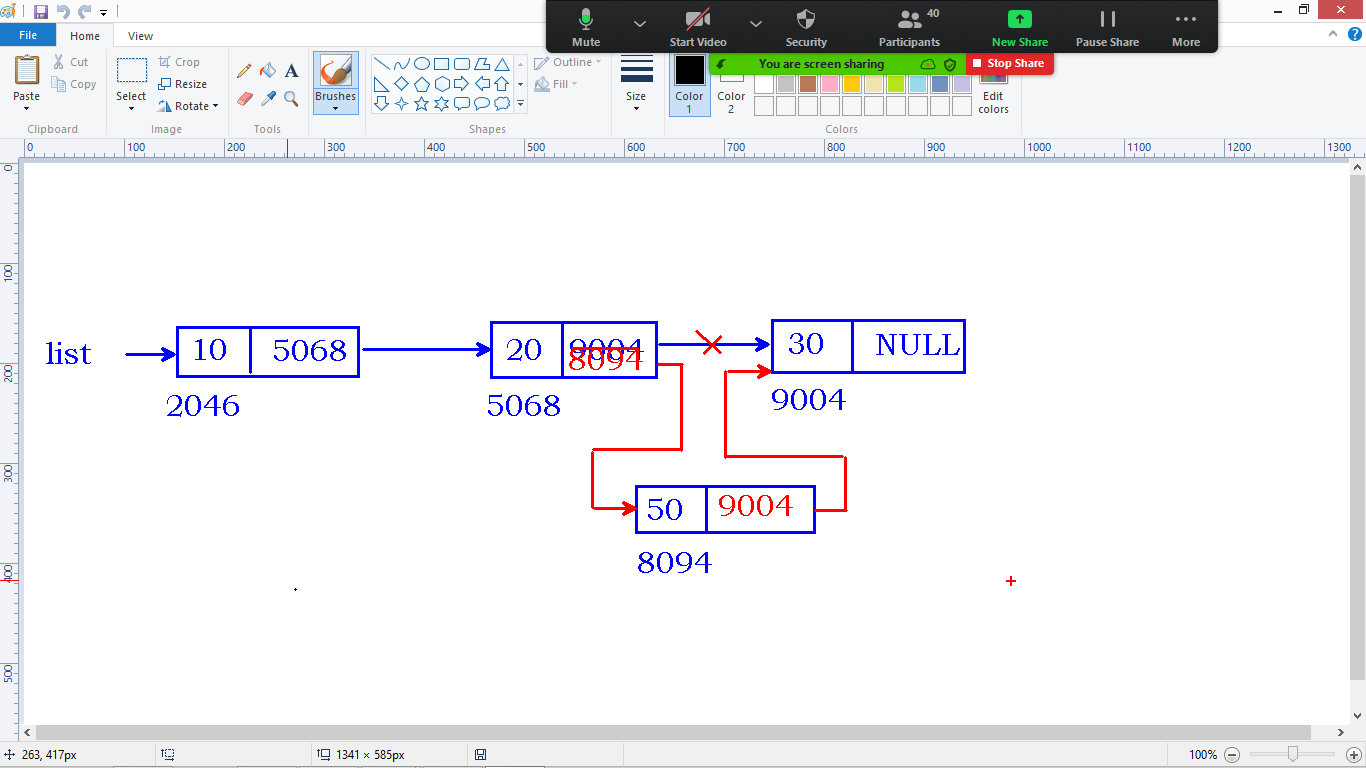
****

* Array is linear. Array elements get consecutive locations.
* We access array elements easily using index.
* **Insertions and Deletions taking time – Shifting of elements after insertion and Deletion takes much time.**



* Linked List stores information in the form of nodes.
* Nodes get memory in random locations and connected using links(pointers)
* Insertions and Deletions are faster compare to Array (Array List)
* Inserting and deleting nodes depends on connections.





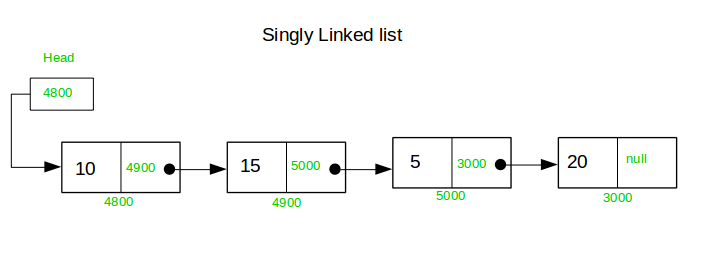
* In My application, we insert the records only once and process (access) every time. Array List is recommended.
* We use linked lists when our application requires continuous insertions and deletions of records.

**We have Different types of Linked Lists:**

1. Singly linked list
2. Doubly linked list
3. Circular linked list

**Singly Linked List:**

* Every Node has 2 Fields
  + Data field
  + Link field
* Link Filed is **pointer type**
* Data means int, float, structure…….
* Last node link is NULL – No other record to connect.



**How to create the node?**

* Node is a memory block with set of fields
* Node fields can be of any type.
* We use structure data type to define the Node.
* Structure can store different types of data elements.
* “link” field is pointer type.
* “link” is pointing to another node(structure type).

**Node representation:**

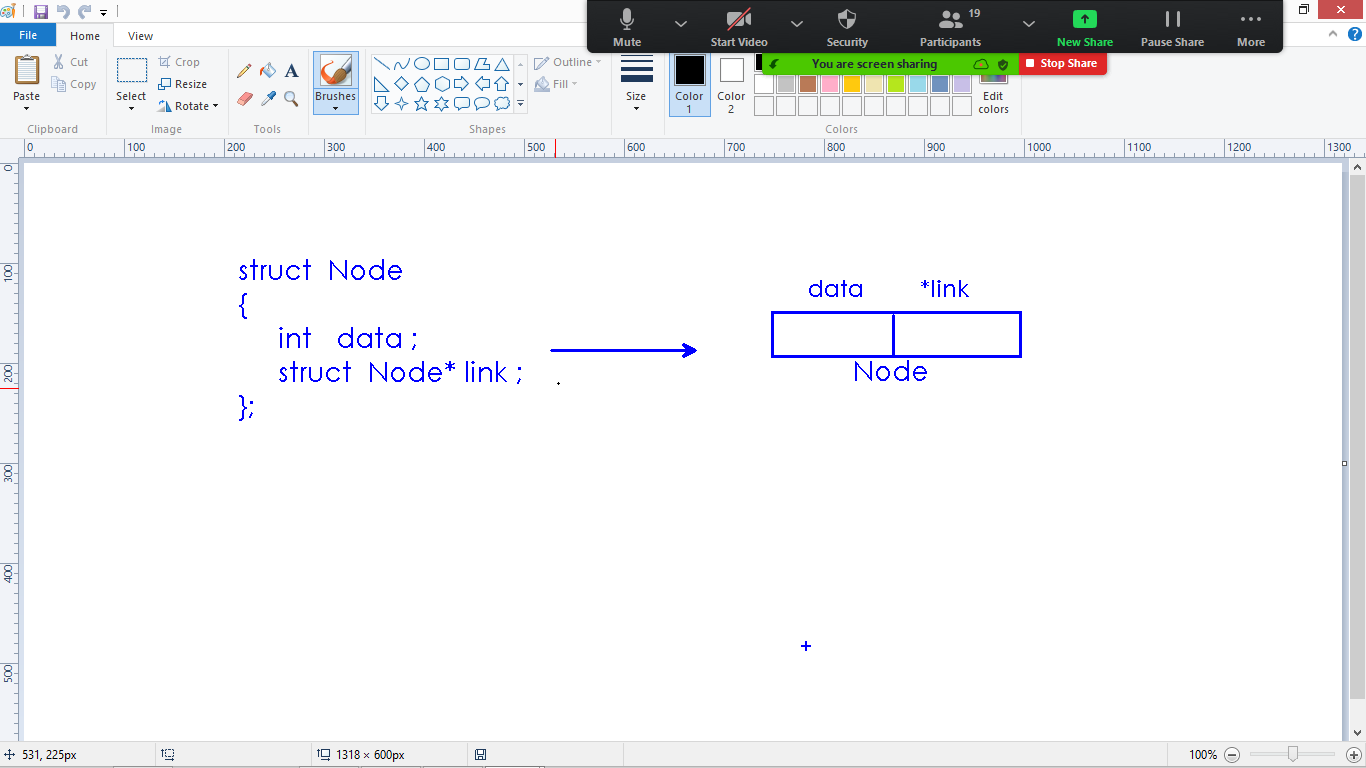
struct Node

{

int data;

struct Node\* link;

};

****

**To store Emp data into Linked List node, then Node as follows:**

struct Emp

{

int eno;

char ename[20];

float salary;

};

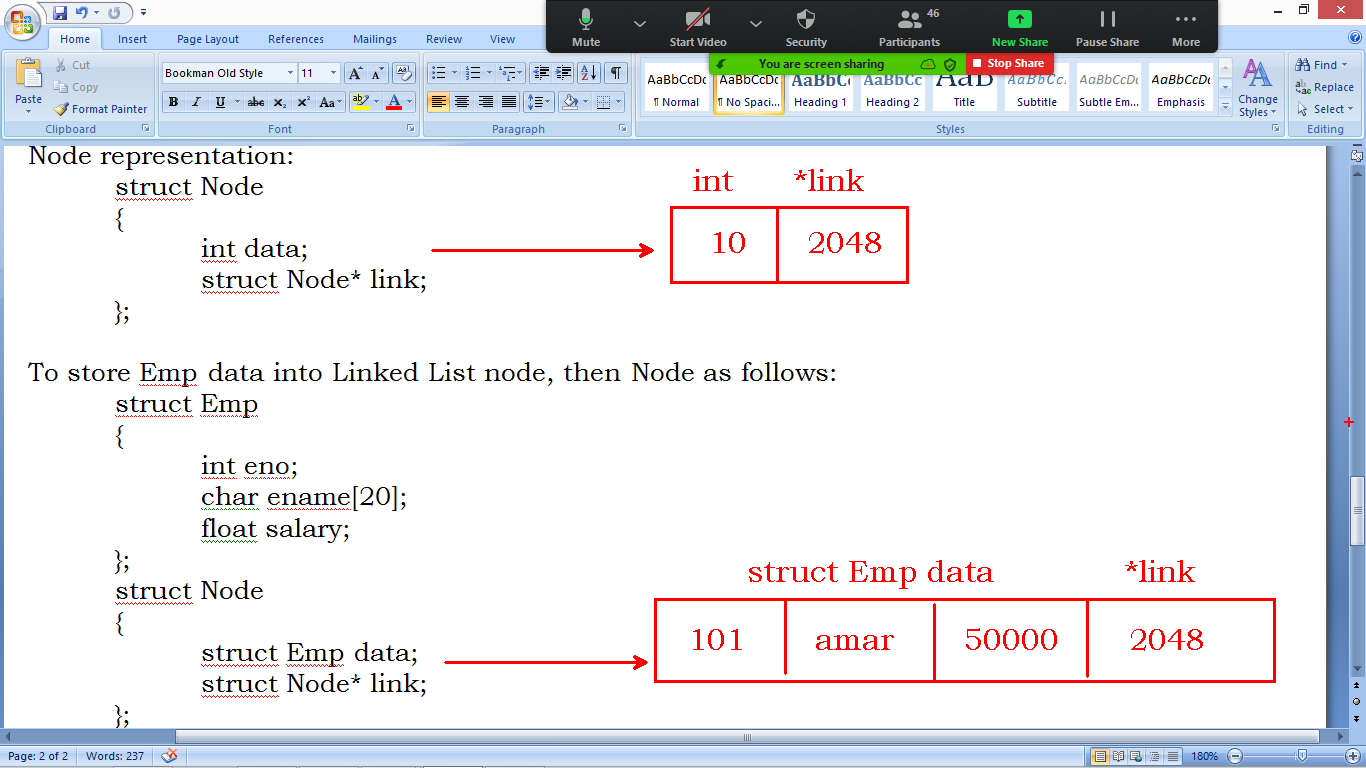
struct Node

{

struct Emp data;

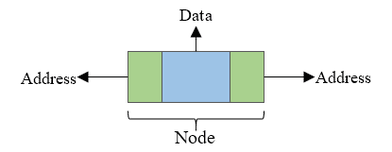
struct Node\* link;

};



**Doubly linked List:**

* Every node contains 3 fields.
  + Data filed
  + Link to previous node
  + Link to next node
* First Node left link is NULL
* Last Node right link is NULL



**List representation:**



**Node representation with structure:**

struct Node

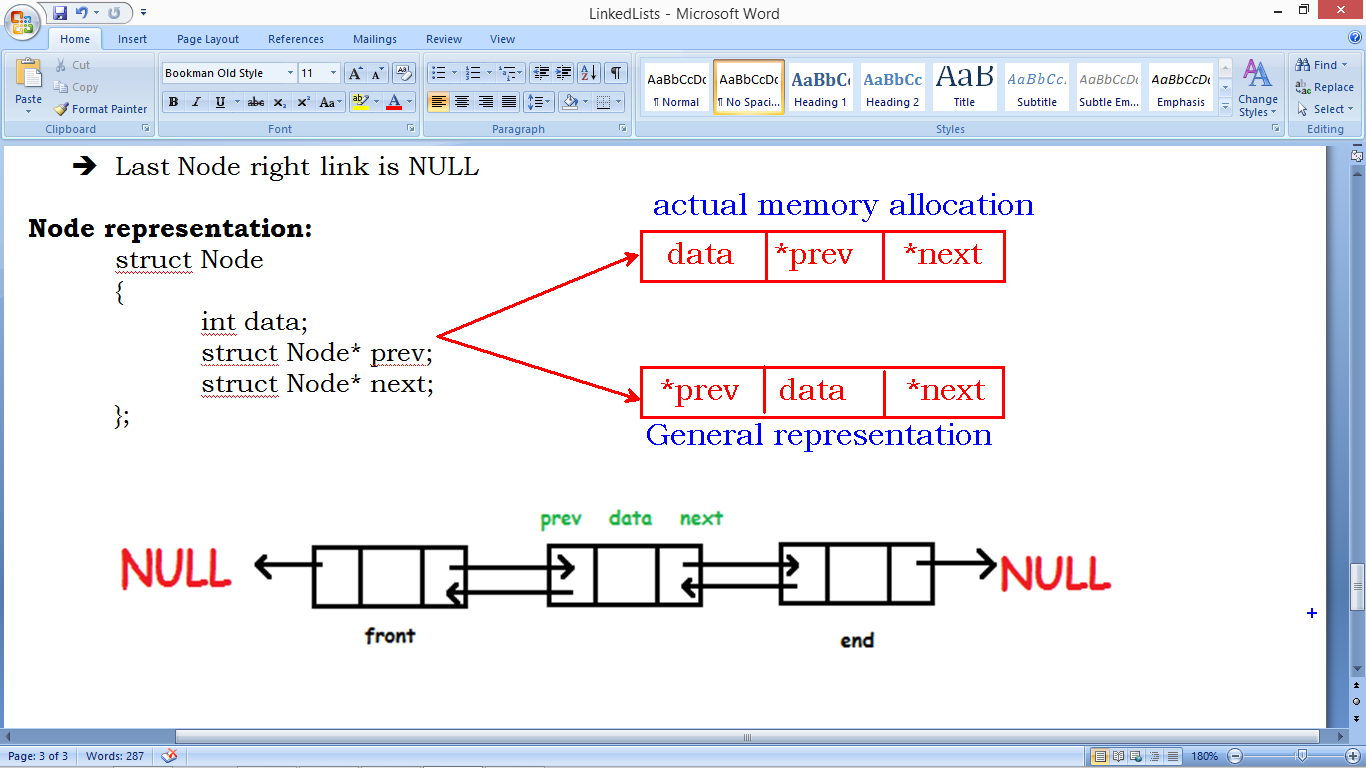
{

int data;

struct Node\* prev;

struct Node\* next;

};



**Circular Single Linked List:**

* Every node as data filed and link filed in Singly circular linked list
* Last node link is connected to first node.

**Node representation:**

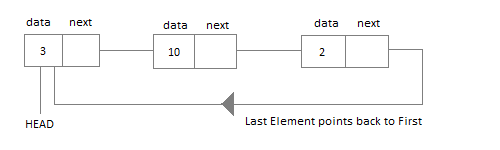
struct Node

{

int data;

struct Node\* link;

};



**Circular Double Linked List:**

* Every node as data filed and 2 link fields
* Last node right link is connected to first node.
* First node left link is connected to the last node.

**Node representation:**

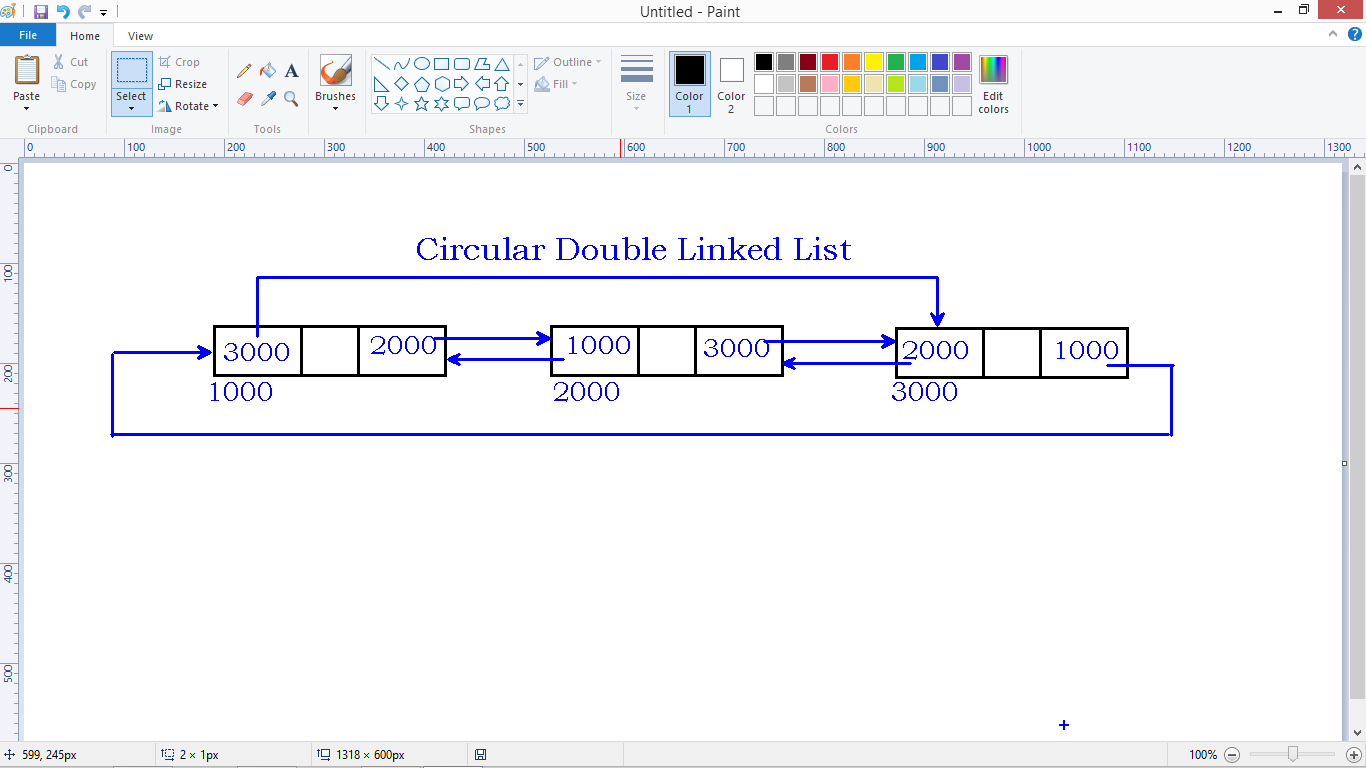
struct Node

{

int data;

struct Node \*left, \*right;

};



**Single Linked List operations**

**Linked List:**

* Linked List stores information in the form of nodes.
* Nodes get memory in random locations and connected using links(pointers)
* Insertions and Deletions are faster compare to Array (Array List)
* Inserting and deleting nodes depends on connections.

**Singly Linked List:**

* Every Node has 2 Fields
  + Data field
  + Link field
* Link Filed is pointer type
* Data means int, float, structure…….
* Last node link is NULL – No other record to connect.

**Node creation using structure:**

struct Node

{

int data;

struct Node\* link;

};

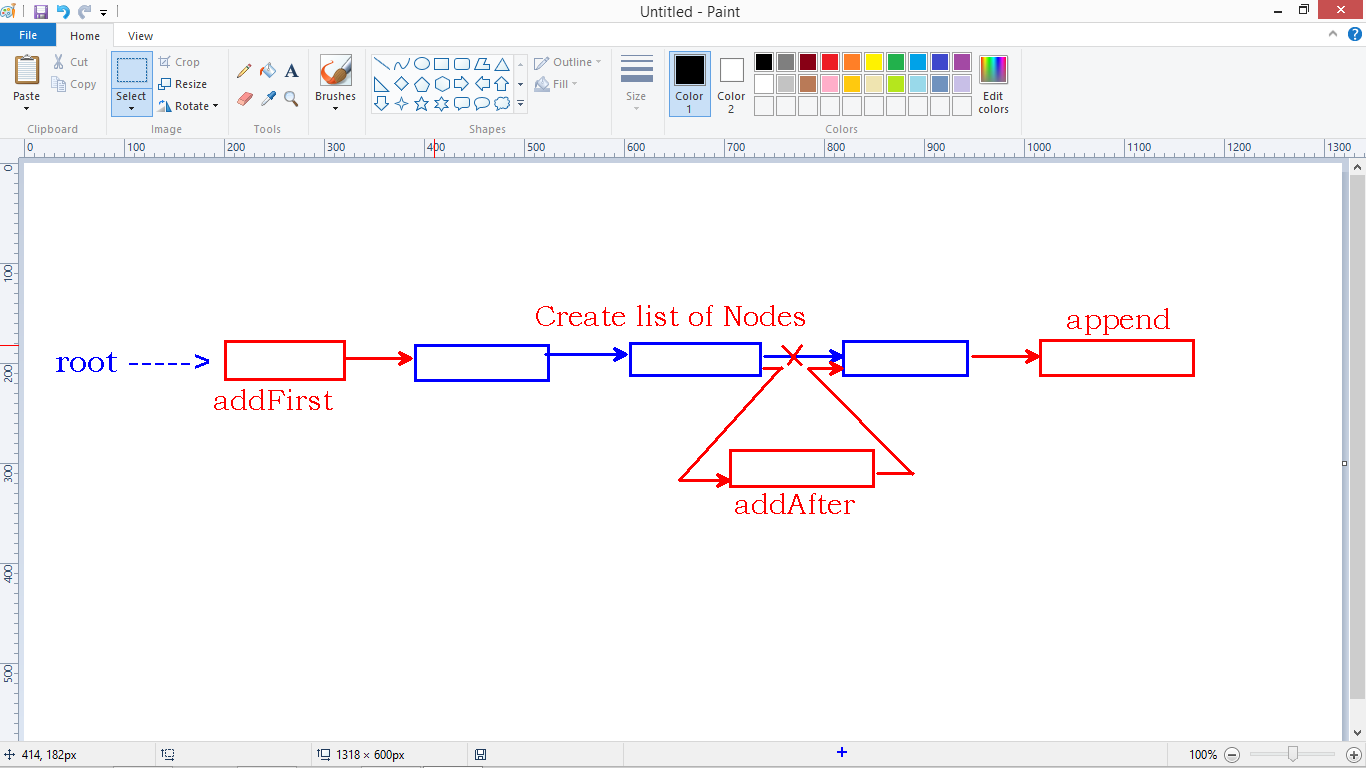
struct Node \*root = NULL;

**Root variable:**

* **“root”** is a variable need to create globally to structure.
  + **struct Node\* root = NULL**
* The linked list pointing by root node only
* This ‘root’ variable can be accessed though out the program from all functions, hence it is GLOBAL VARIABLE.
* “root” is always pointing to First Node in the List.
* Other nodes pointing by previous nodes links.

**Operations on Single Linked List:**

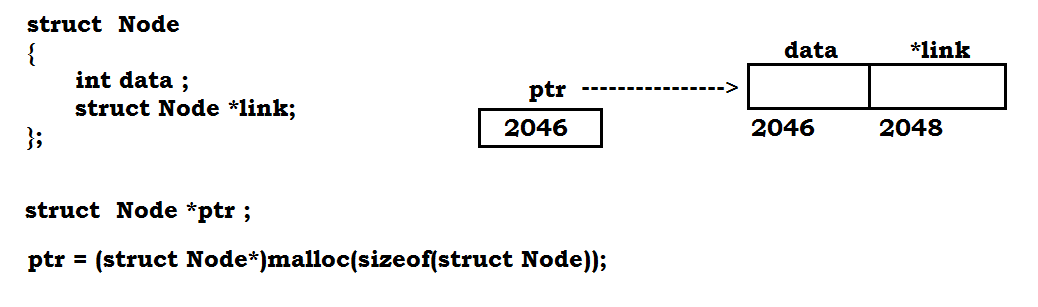
1. Create List (with set of nodes – continuous insertions)
2. Append (add node at end)
3. AddFirst
4. AddAfter



1. Delete
2. Length
3. Display
4. Swap node data
5. Swap nodes
6. Reverse the list
7. Sort the list

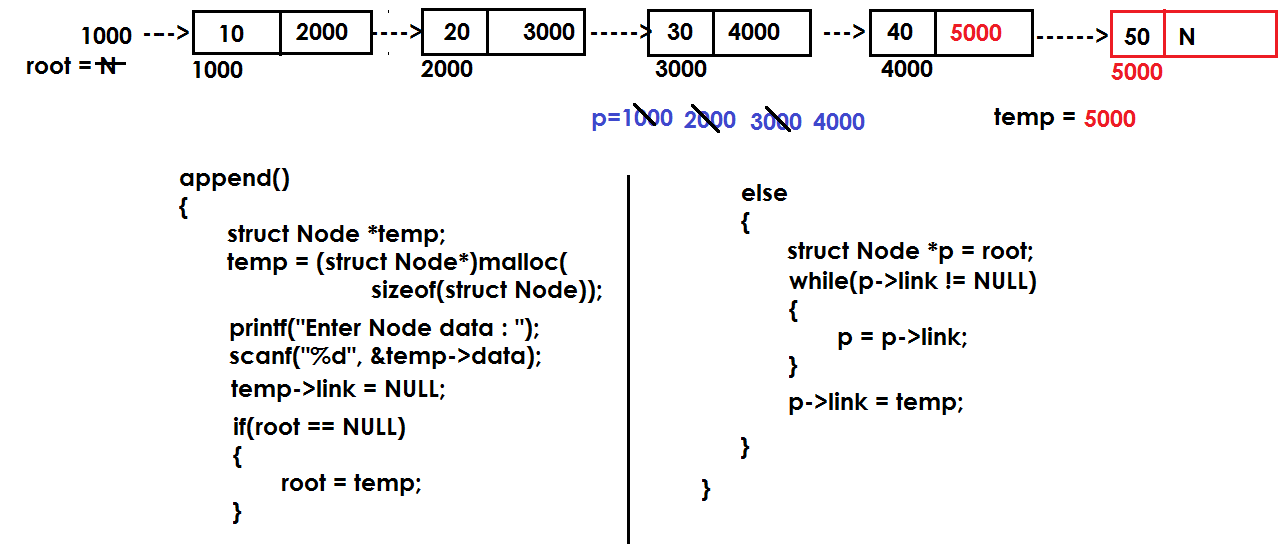
**Allocate memory to Node dynamically:**

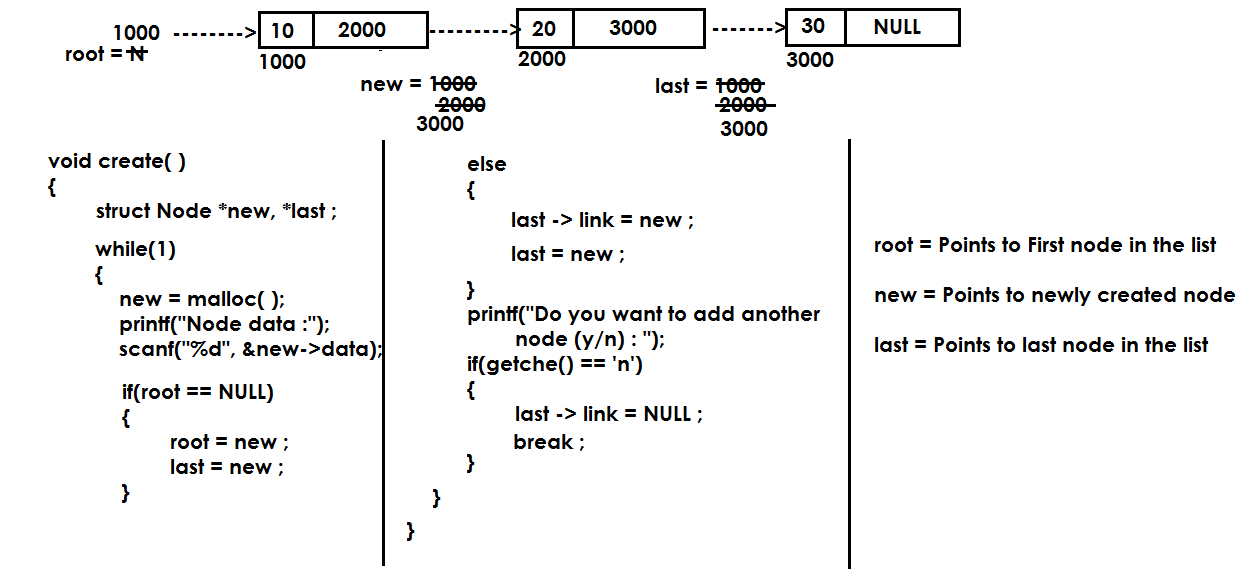
* Node is structure type.
* We use malloc() function to allocate memory to structures.
* Once memory allocated, malloc function returns base address of memory blck as void\*.
* We need to convert the void\* type into struct Node\* type.

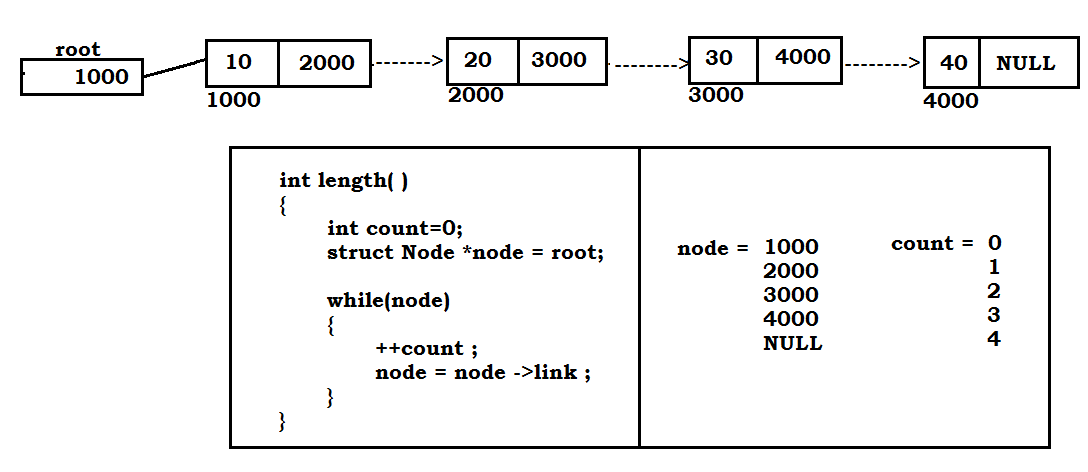


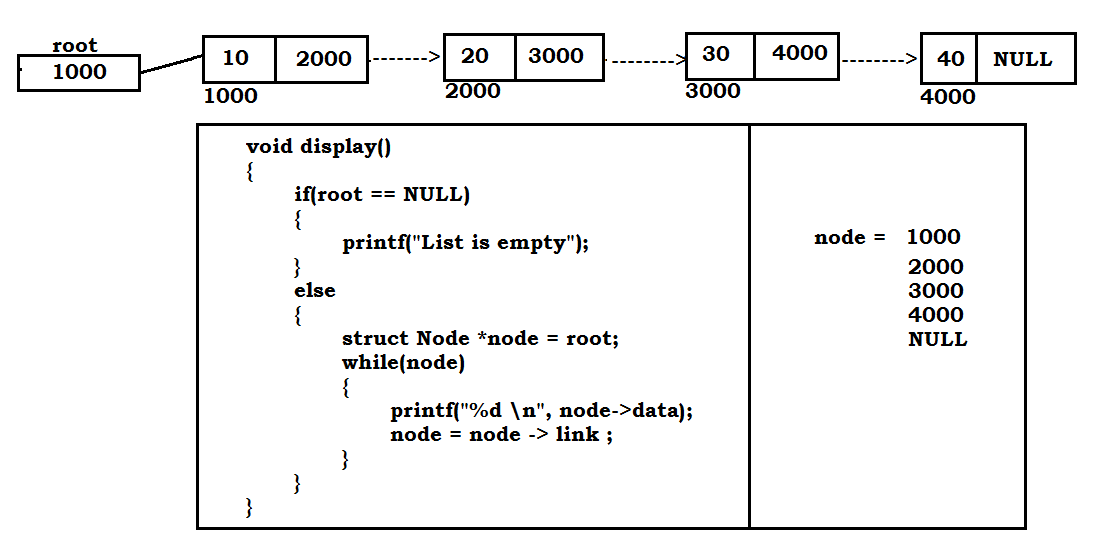
**Append():**

* Adding a node at the end of existing list is called “Append”.
* If no nodes in the list, newly created node become first node or root node.
* If list is not empty, we need to add the node at the end.









**Code: Append() , length(), display()**

#include<stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node\* link;

};

struct Node\* root=NULL;

void append();

int length();

void display();

int main()

{

int ch, len;

while(1)

{

printf("1.append \n");

printf("2.length \n");

printf("3.display \n");

printf("4.quit \n");

printf("Enter your choice : ");

scanf("%d", &ch);

switch(ch)

{

case 1 : append();

break;

case 2 : len = length();

printf("Length is : %d \n", len);

break;

case 3 : display();

break;

case 4 : exit(1);

default : printf("Invalid choice \n");

}

}

return 0;

}

void append()

{

struct Node \*new;

new = (struct Node\*)malloc(sizeof(struct Node));

printf("\nEnter node data : ");

scanf("%d", &new->data);

new->link = NULL;

if(root==NULL)

{

root = new;

}

else

{

struct Node \*last = root;

while(last->link != NULL)

{

last = last->link;

}

last->link = new;

}

}

int length()

{

int count=0;

struct Node \*node=root;

while(node)

{

++count;

node=node->link;

}

return count;

}

void display()

{

if(root==NULL)

{

printf("List is Empty\n");

}

else

{

struct Node \*node=root;

printf("List elements :\n");

while(node)

{

printf("%d \n", node->data);

node=node->link;

}

}

}

**Create(), length() and display():**

#include<stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node\* link;

};

struct Node\* root=NULL;

void createList();

int length();

void display();

int main()

{

int len;

createList();

len = length();

printf("\nlength is : %d \n", len);

display();

return 0;

}

void createList()

{

struct Node \*new, \*last;

while(1)

{

new = (struct Node\*)malloc(sizeof(struct Node));

printf("\nEnter node data : ");

scanf("%d", &new->data);

if(root==NULL)

{

root = new;

last = new;

}

else

{

last->link = new;

last = new;

}

printf("\nDo you want to add another node(y/n) : ");

if(getche()=='n')

{

last->link=NULL;

break;

}

}

}

int length()

{

int count=0;

struct Node \*node=root;

while(node)

{

++count;

node=node->link;

}

return count;

}

void display()

{

if(root==NULL)

{

printf("List is Empty\n");

}

else

{

struct Node \*node=root;

printf("List elements :\n");

while(node)

{

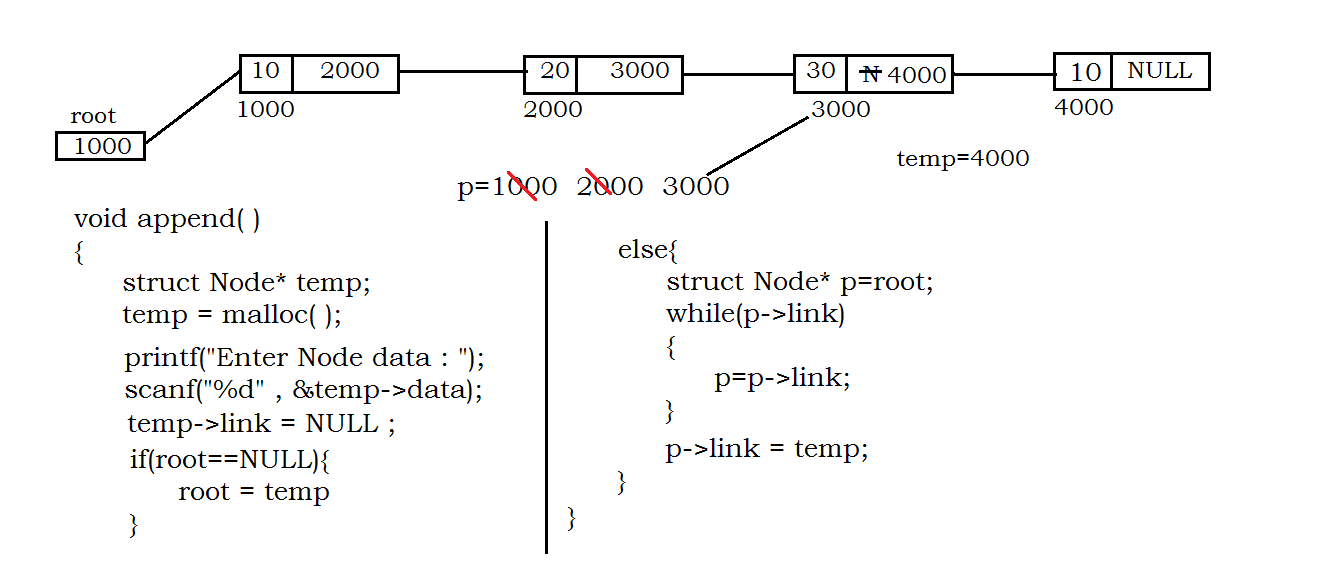
printf("%d \n", node->data);

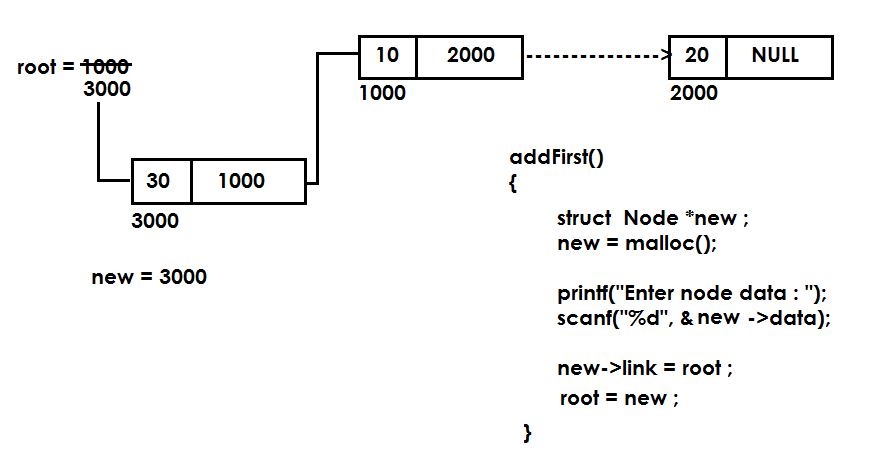
node=node->link;

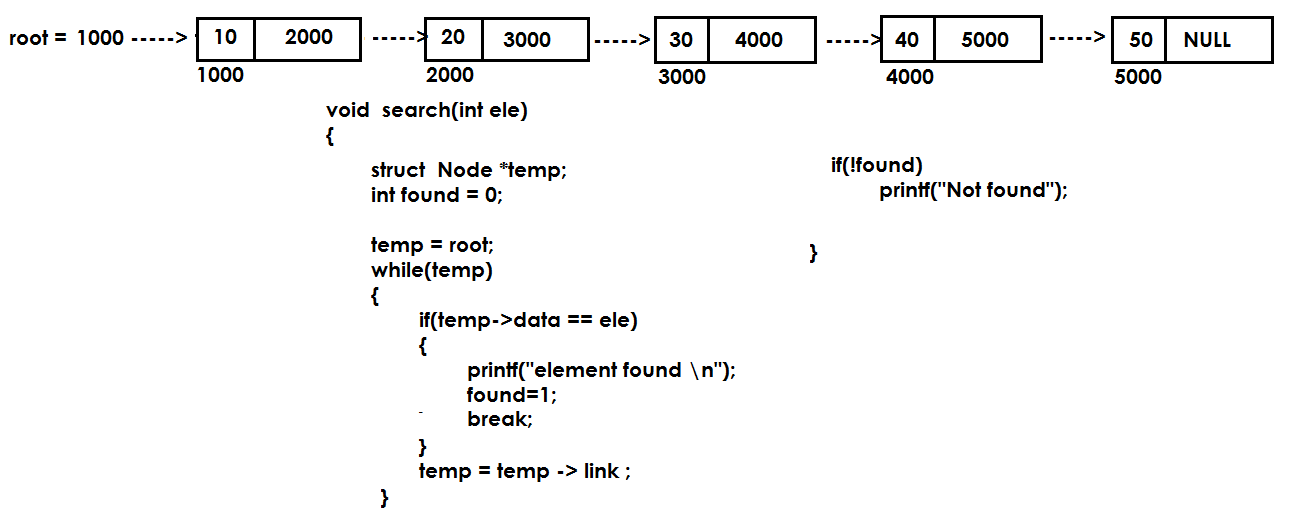
}

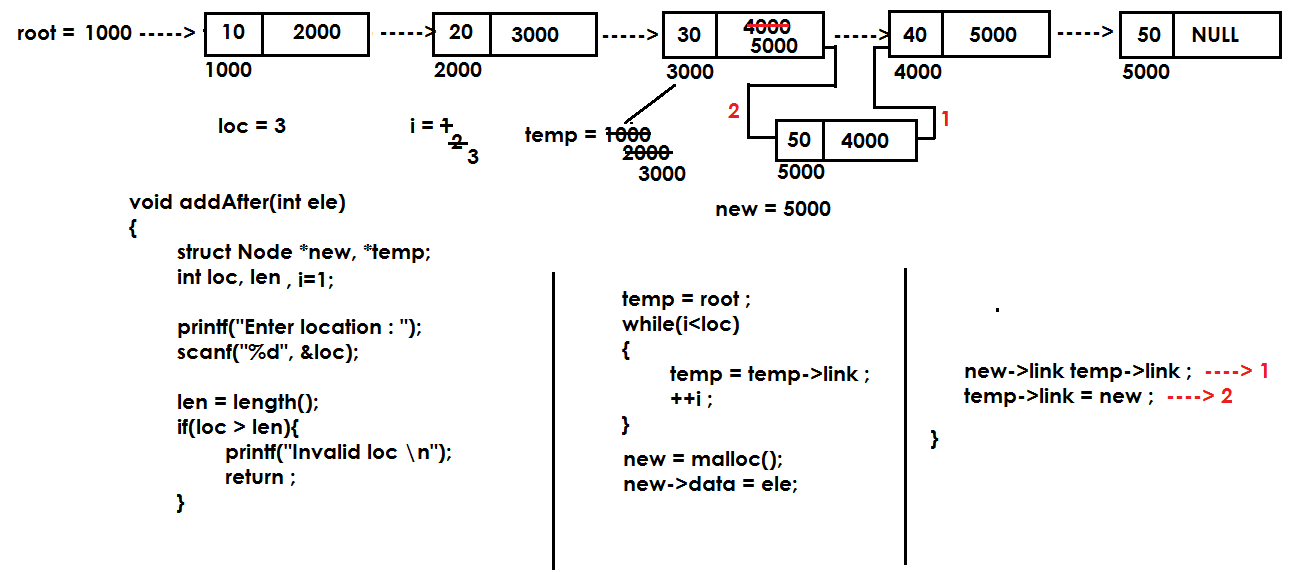
}

}









#include<stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node\* link;

};

struct Node\* root=NULL;

void append();

void addFirst();

int length();

void display();

int main()

{

int ch, len;

while(1)

{

printf("\n1.Display \n");

printf("2.Length \n");

printf("3.Append \n");

printf("4.Add First \n");

printf("5.Quit \n");

printf("Enter your choice : ");

scanf("%d", &ch);

switch(ch)

{

case 1 : display();

break;

case 2 : len = length();

printf("Node count is : %d\n", len);

break;

case 3 : append();

break;

case 4 : addFirst();

break;

case 5 : exit(1);

default : printf("Invalid choice...\n");

}

}

return 0;

};

void display()

{

struct Node\* node;

if(root == NULL)

{

printf("List is Empty \n");

}

else

{

node=root;

while(node)

{

printf("%d \n", node->data);

node=node->link;

}

}

}

int length()

{

struct Node\* node=root;

int count=0;

while(node)

{

++count ;

node = node->link ;

}

return count ;

}

void append()

{

struct Node \*node;

node = (struct Node\*)malloc(sizeof(struct Node));

printf("Enter node data : ");

scanf("%d" , &node->data);

node->link = NULL;

if(root==NULL)

{

root = node;

}

else

{

struct Node\* p=root;

while(p->link)

{

p=p->link;

}

p->link = node;

}

}

void addFirst()

{

struct Node \*node;

node = (struct Node\*)malloc(sizeof(struct Node));

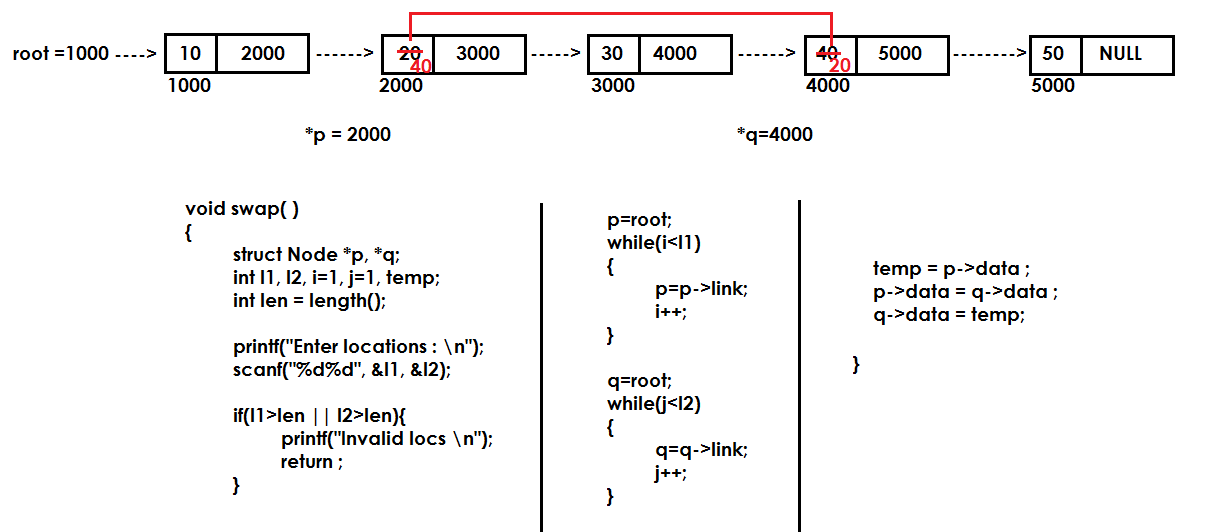
printf("Enter node data : ");

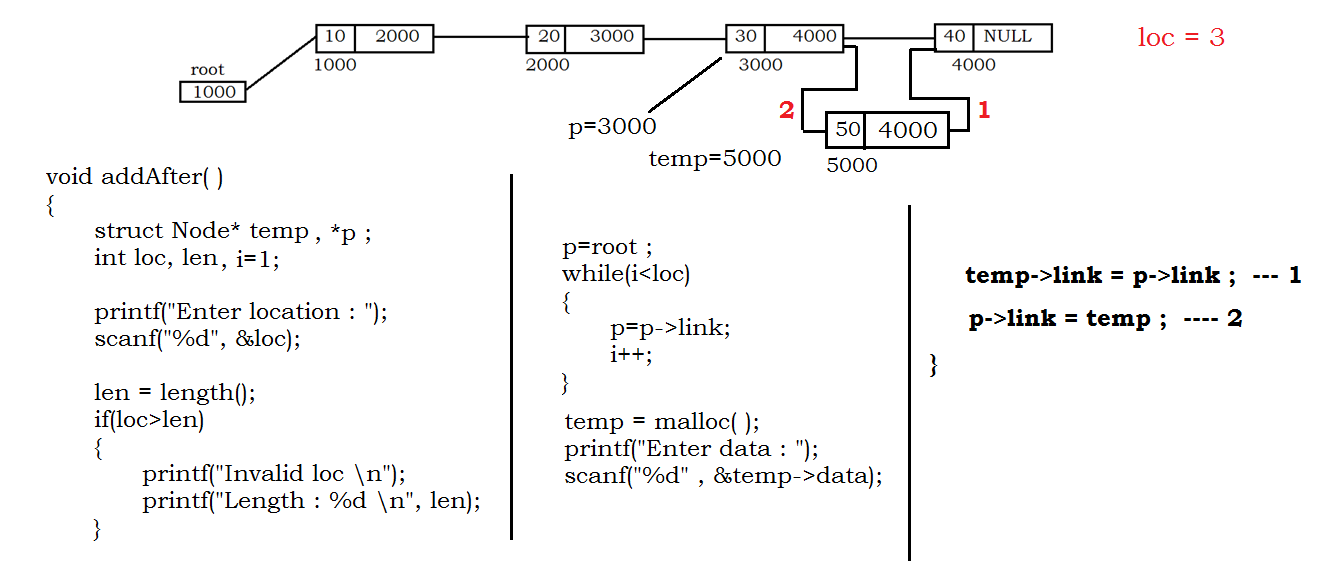
scanf("%d" , &node->data);

node->link = root;

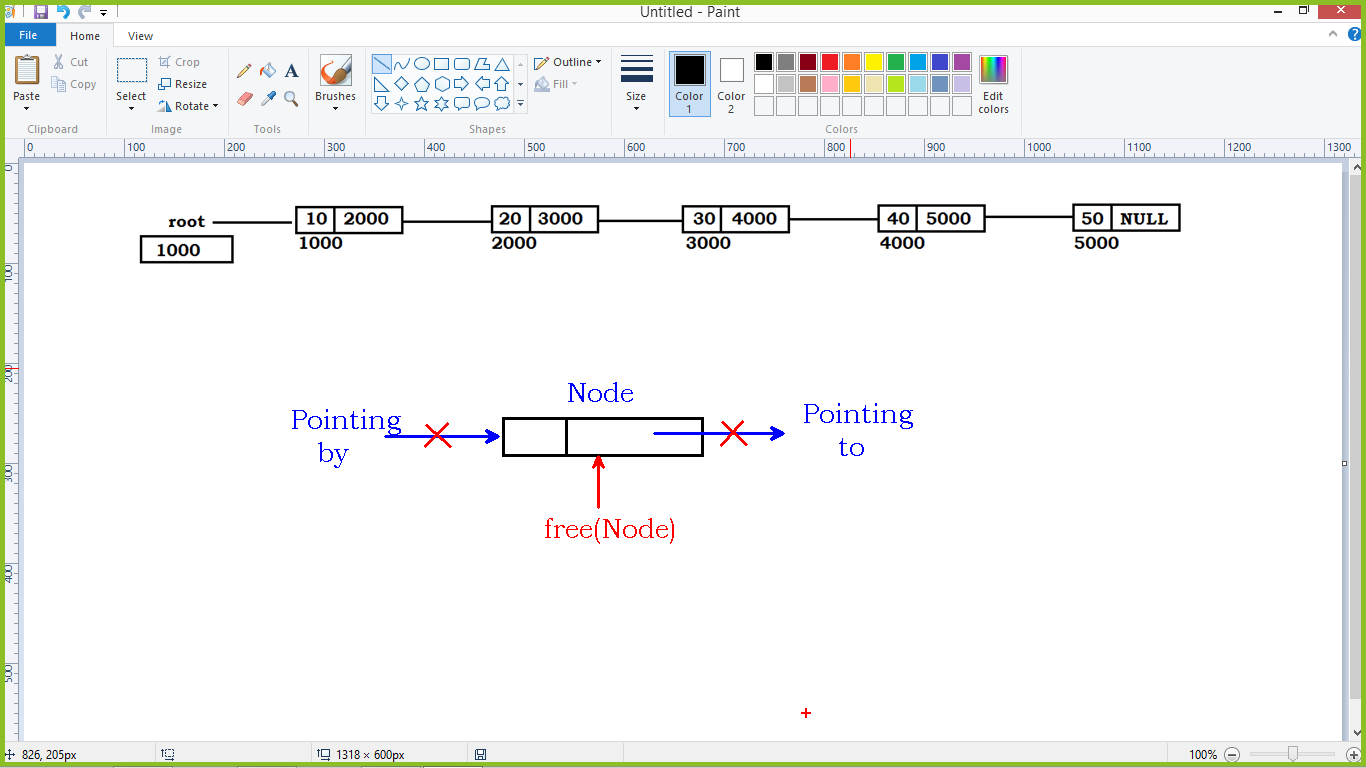
root = node;

}

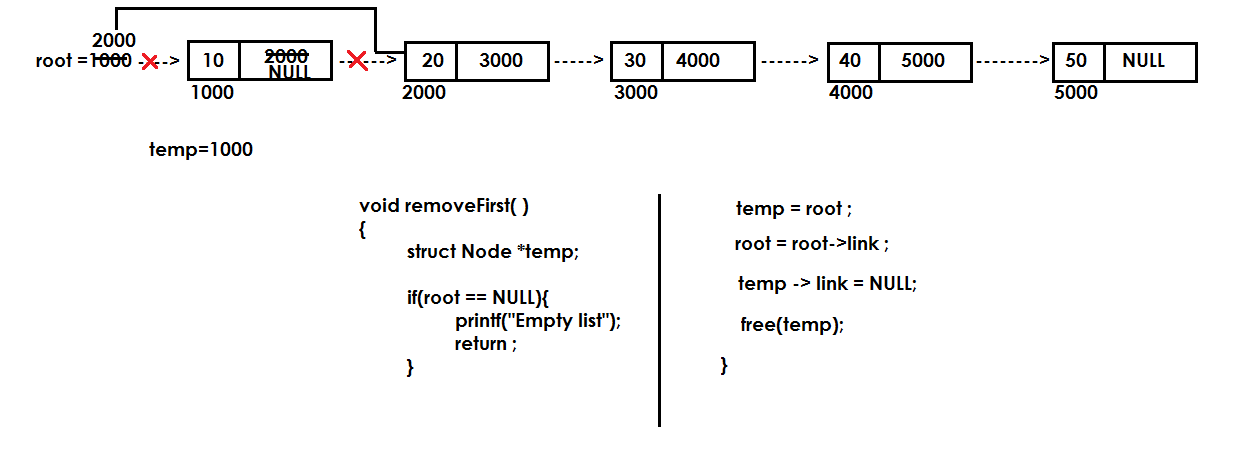




**Node removal:**

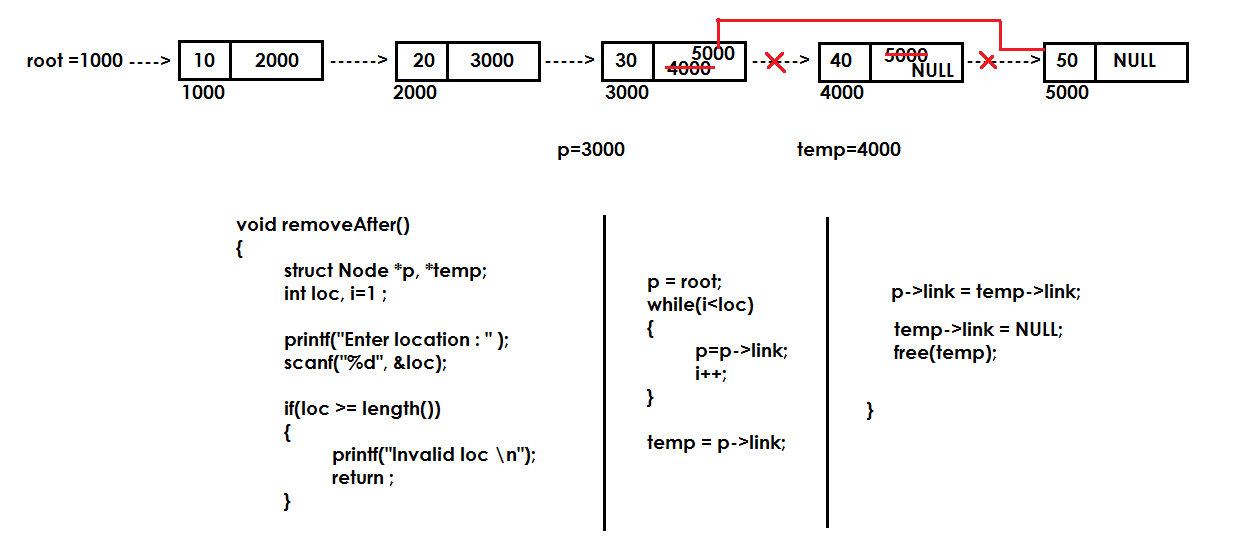
****

* We can call free() function to release the memory allocated to node.
* When we call free() function, the node should not connected to any other node(not pointing to and not pointing by).
* Calling free() function is just informing that the memory is not using in the application hence it can allocated to any other variable if required.



**removeAfter():**

* Read node location.
* If loc greater then length, return with error message
* If loc is present, make the pointer pointing to that location.
* Maintain the pointer to previous node of specified location to connect with next node of location node.



**Write linked list code to perform :**

1. Append()
2. addFirst()
3. addAfter()
4. removeFirst()
5. removeAfter()
6. length()
7. display()
8. swap()
9. search()

**Create Node with Emp details:**

We create structure variables in 2 ways.

1. Variable to structure:
   1. We access element using dot(.) operator
   2. Memory allocates to all variables in the structure.
2. Pointer to structure:
   1. We access elements using arrow(->) operator.
   2. Memory allocates only to pointer variable
   3. We need to allocate memory again using malloc() to that pointer variable.

**Variable to structure code:**

#include<stdio.h>

struct Emp

{

int eno;

char ename[20];

float esal;

};

struct Node

{

struct Emp e ;

struct Node \*link;

};

int main()

{

struct Node \*rec;

rec = (struct Node\*)malloc(sizeof(struct Node));

printf("Enter Emp details : \n");

scanf("%d%s%f", &rec->e.eno, rec->e.ename, &rec->e.esal);

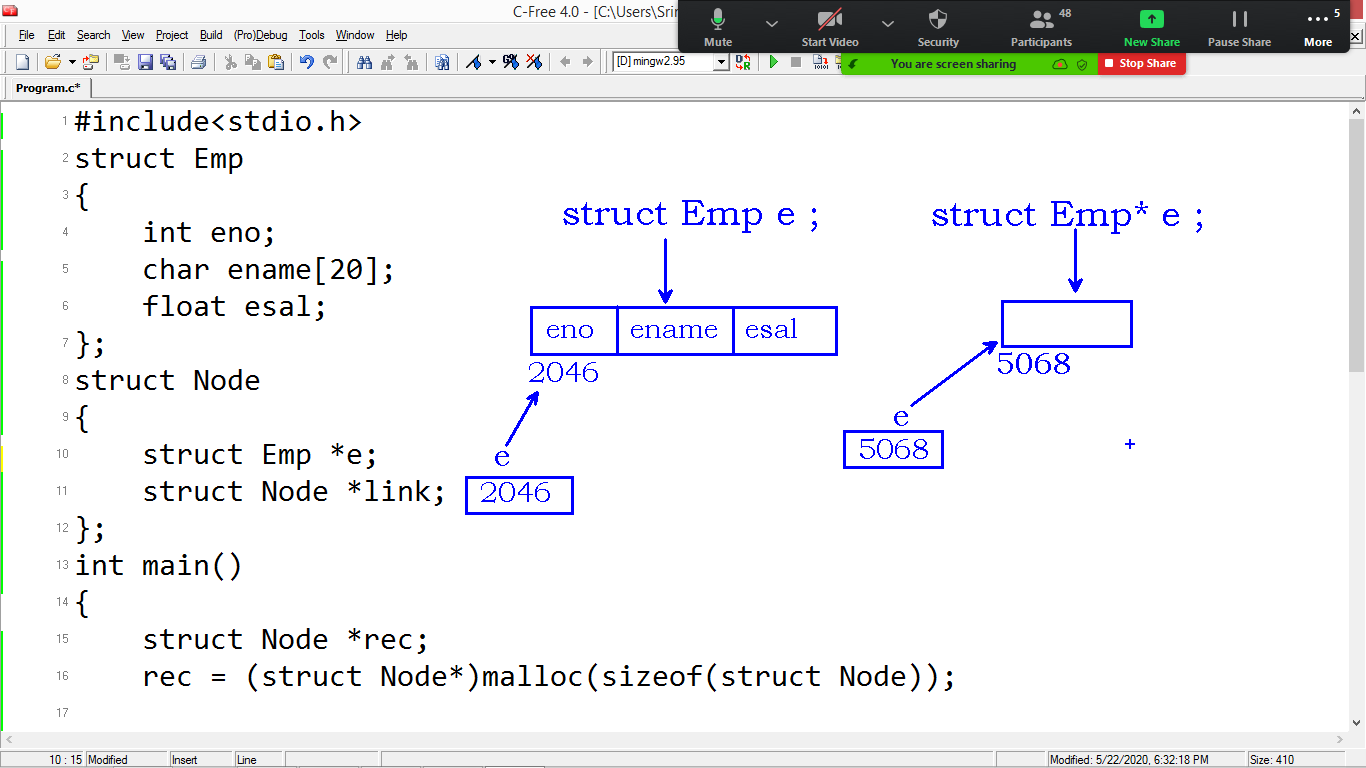
printf("Details are :\n");

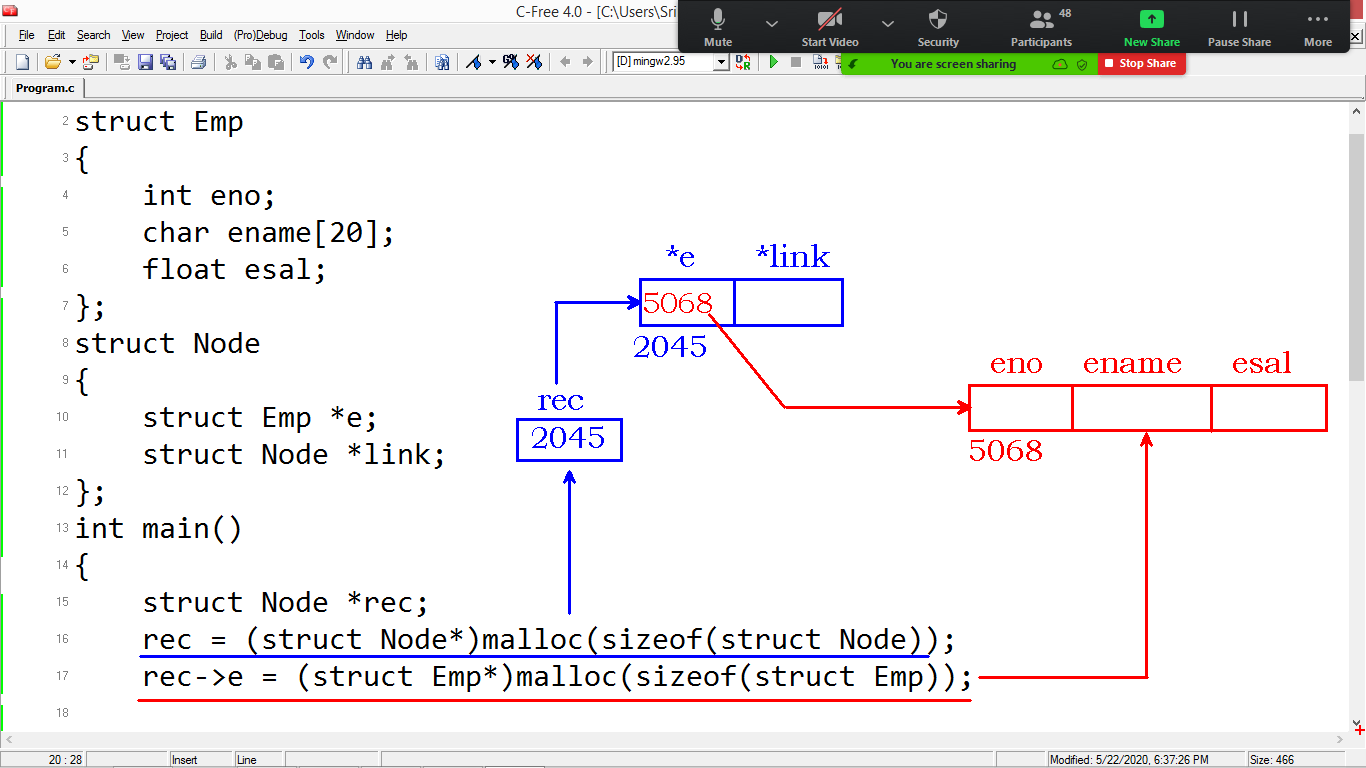
printf("Name : %s\n", rec->e.ename);

return 0;

}

**Pointer to structure code:**

****



**Code is:**

#include<stdio.h>

struct Emp

{

int eno;

char ename[20];

float esal;

};

struct Node

{

struct Emp \*e;

struct Node \*link;

};

int main()

{

struct Node \*rec;

rec = (struct Node\*)malloc(sizeof(struct Node));

rec->e = (struct Emp\*)malloc(sizeof(struct Emp));

printf("Enter Emp details : \n");

scanf("%d%s%f", &rec->e->eno, rec->e->ename, &rec->e->esal);

printf("Details are :\n");

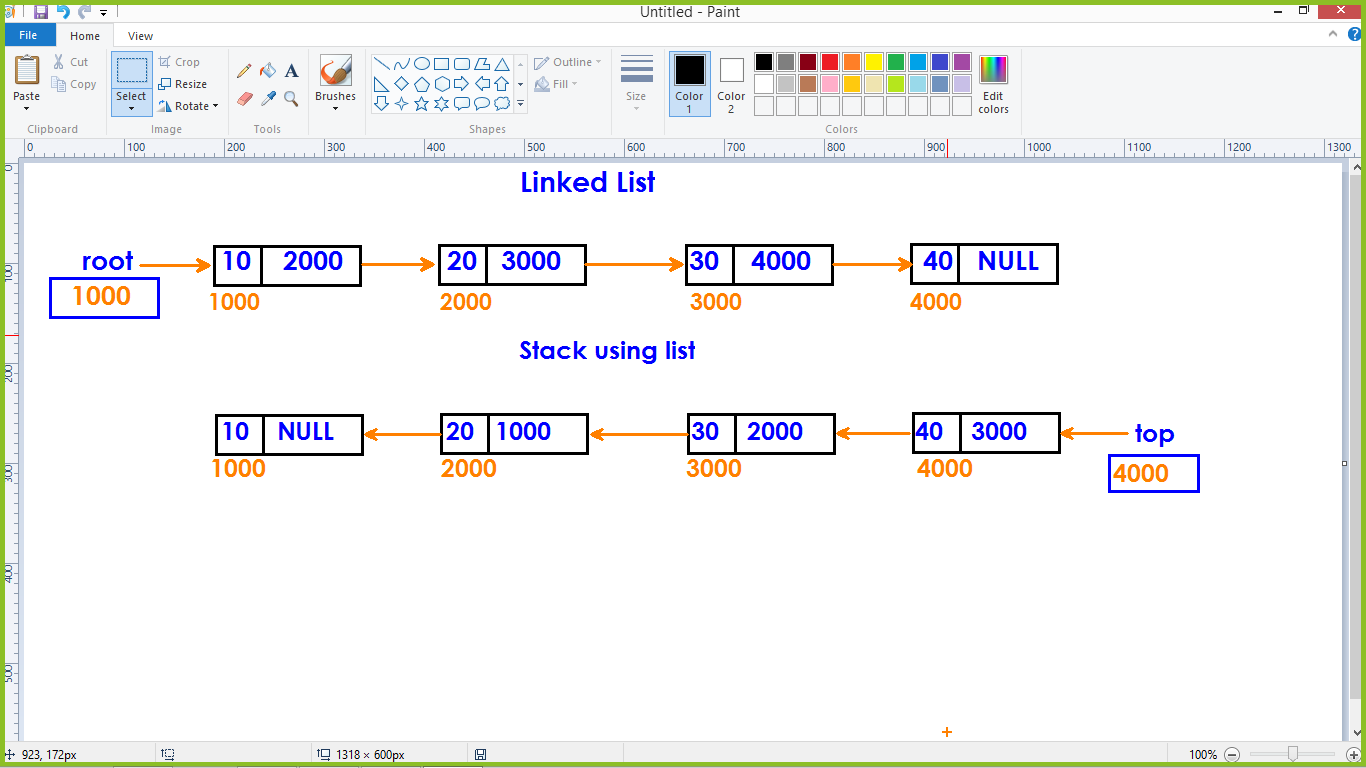
printf("Name : %s\n", rec->e->ename);

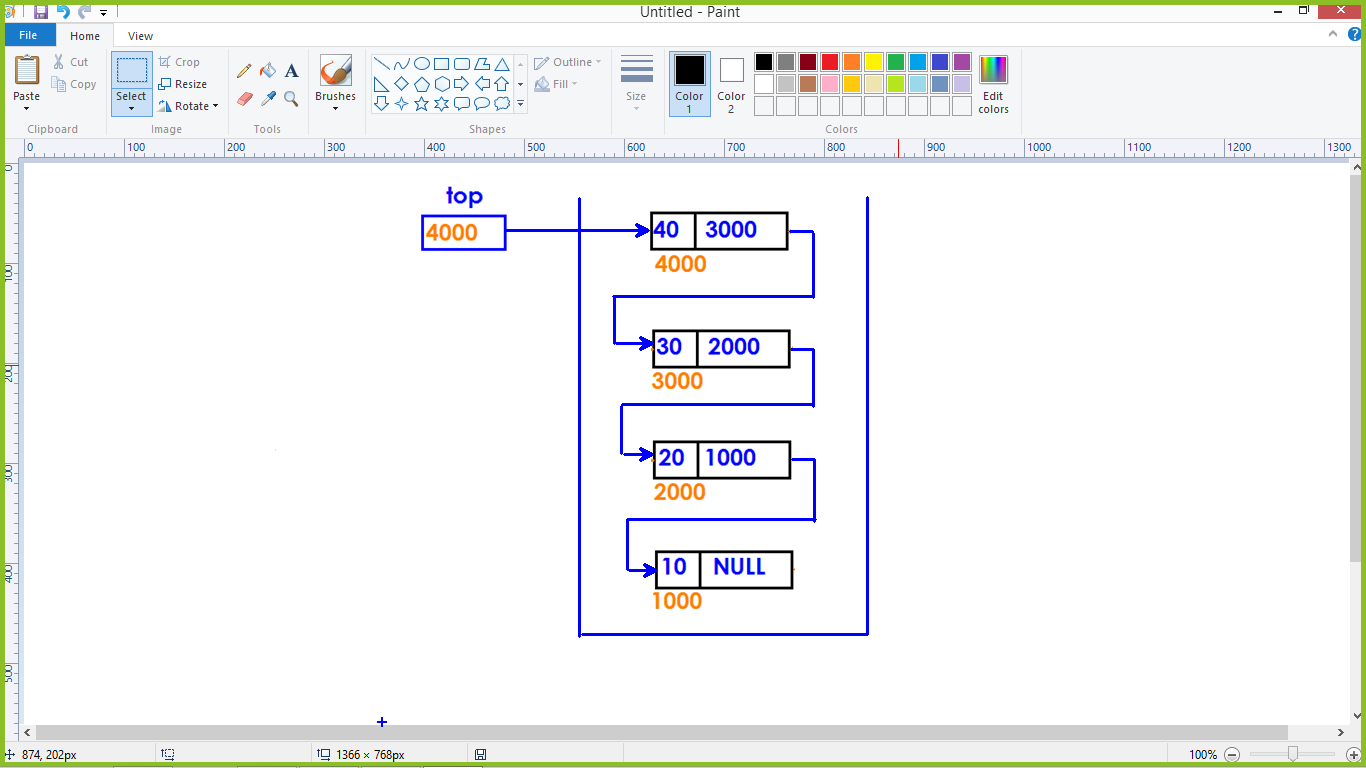
return 0;

}

**Stack Using Linked List:**

* We can create Stack using single linked list.
* Generally we use Stack using arrays.
* Stack using Linked list creation using Nodes.
* We implement the LIFO(Last In First Out) rule here also.
* “Top” should point to last node in the list.





**Node representation:**

struct Node

{

int data;

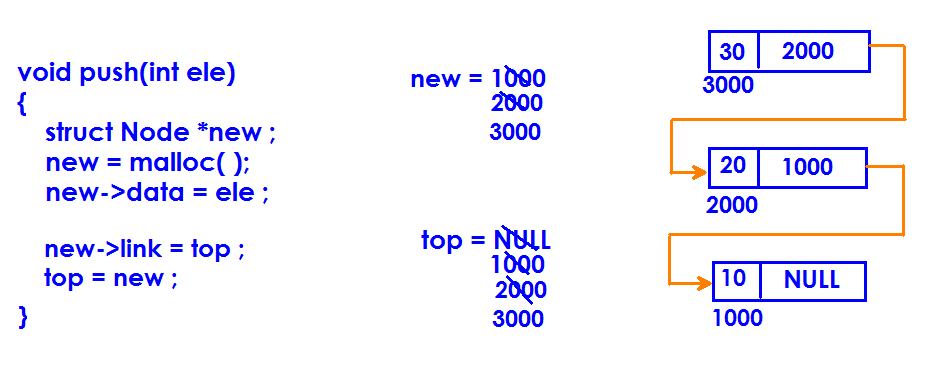
struct Node\* link;

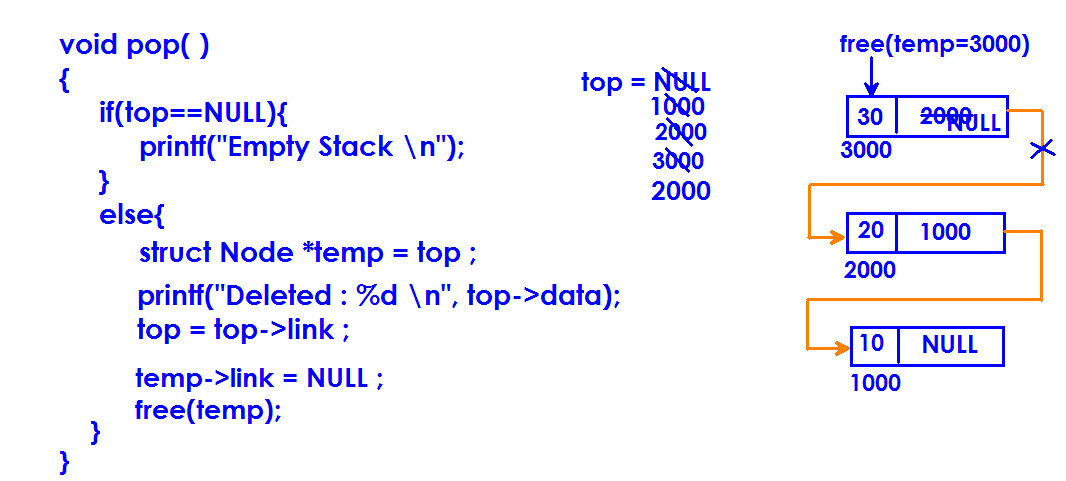
};

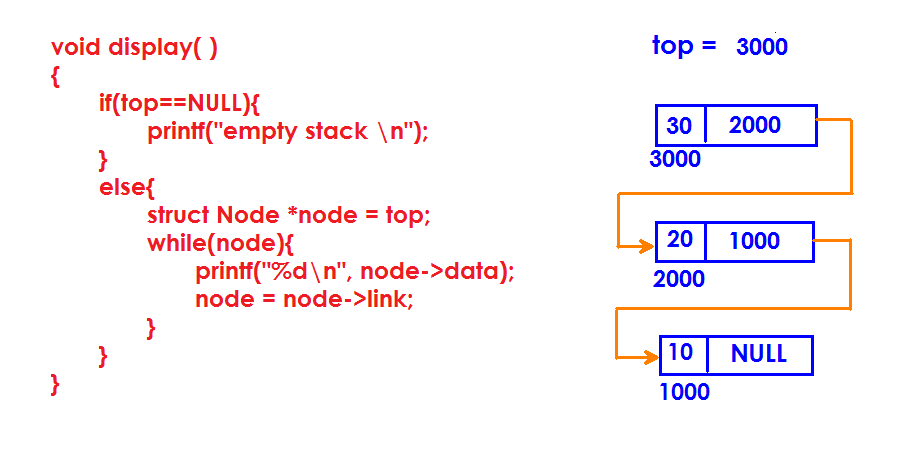
struct Node\* top=NULL;

**Operations:**

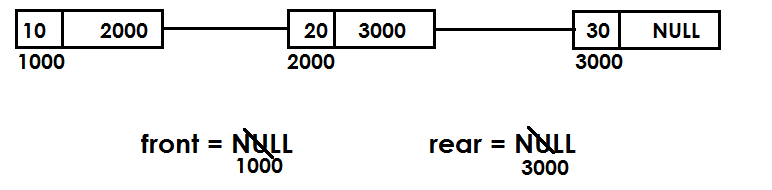
1. Push()
2. Pop()
3. Traverse()





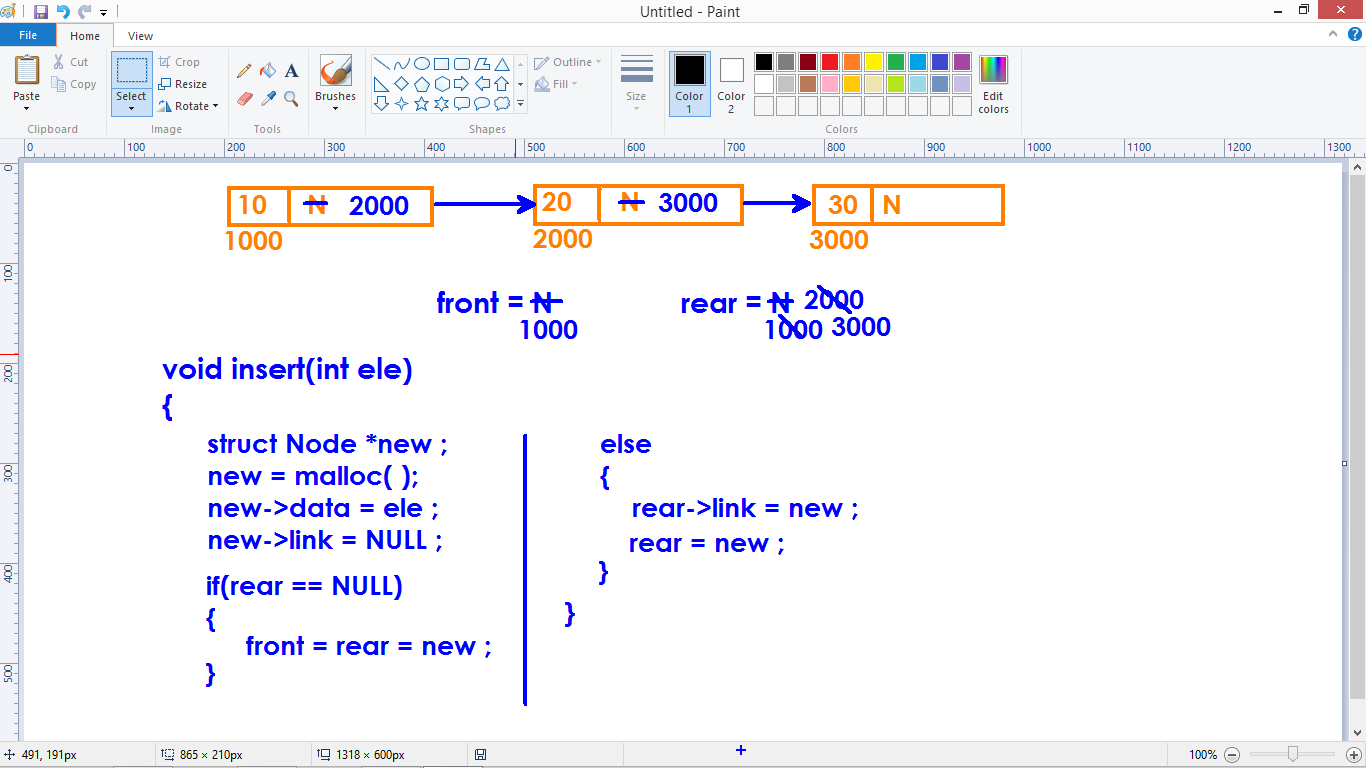


**Queue using Single Linked List**



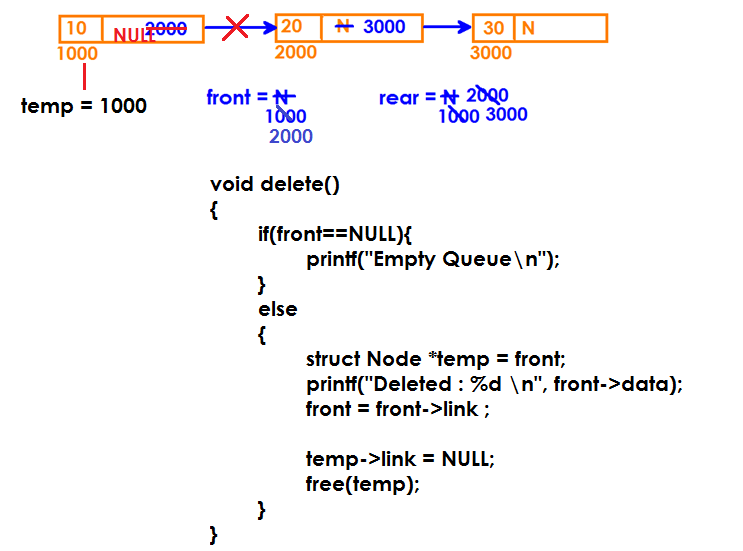
**Insert element into Queue:**

* **Initially front and rear values are NULLs**
* **Insertion must be done through rear variable.**
* **If no elements in the list, front = rear = new ;**
* **If list has elements, we need to add the node at the end (link to rear)**

****

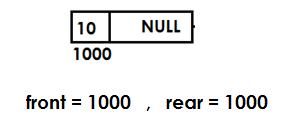
**Deleting element:**

* Deletion from front.
* We must use front pointer to remove the element.
* If front is null – return with message “List is empty”
* If elements are present, remove the first node and make the front pointing to second node.



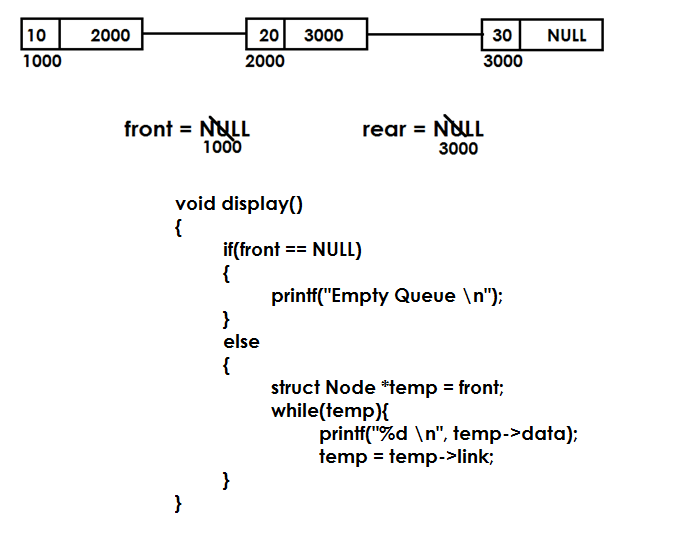
**Note:**

* If you are deleting the last node using front, we need to make rear pointer assign to NULL.
* We confirm last node by checking the condition **front == rear**



**Display Queue elements:**

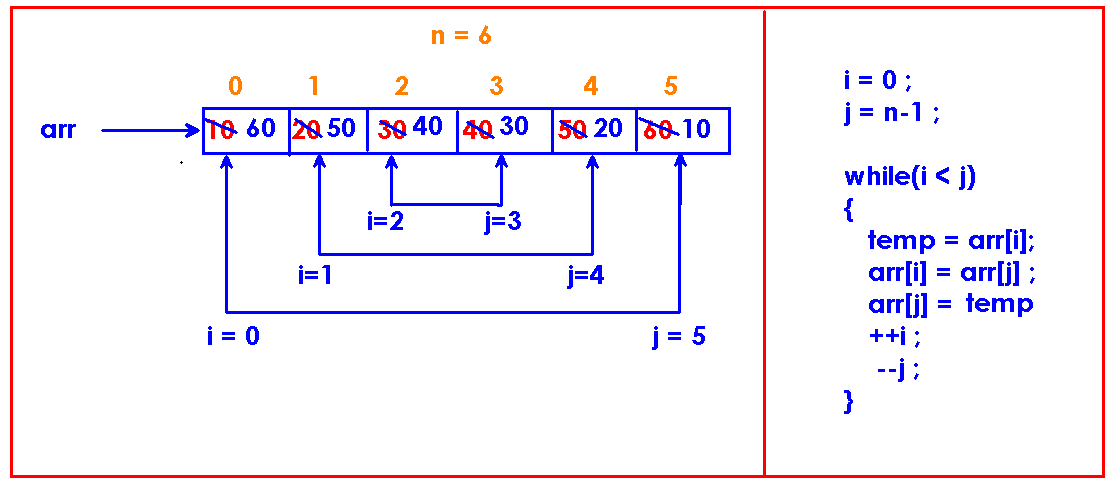
* Check the Queue is empty or not
* If empty , return with error message
* If not, display all elements in the Queue.



**Code program:**

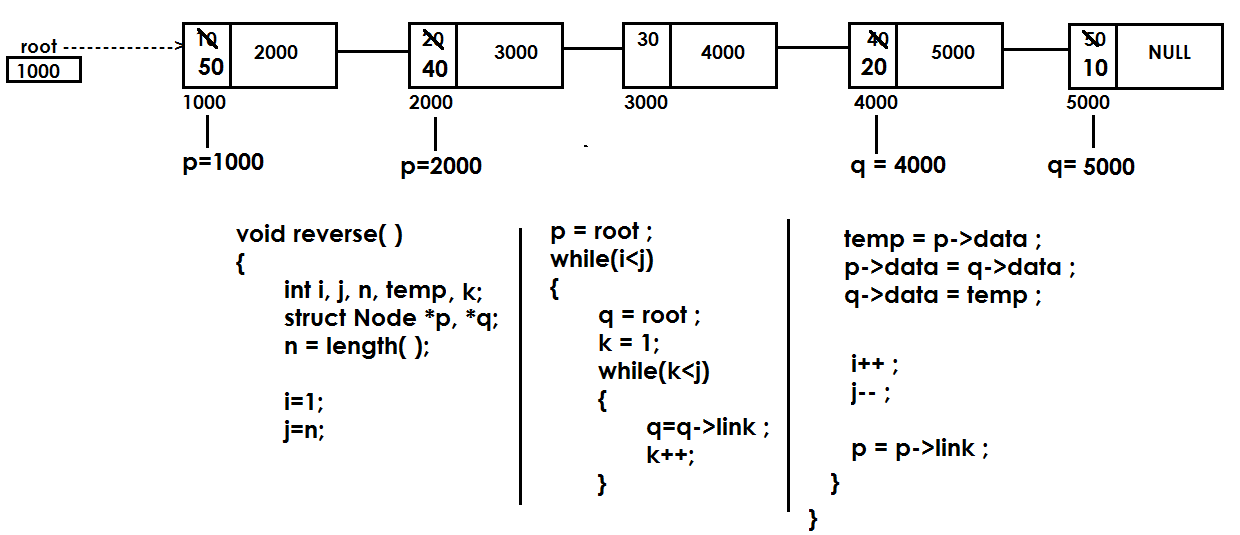
**How to reverse an array:**

* We access elements of array using index.
* We can reverse elements in array without taking any duplicate array.



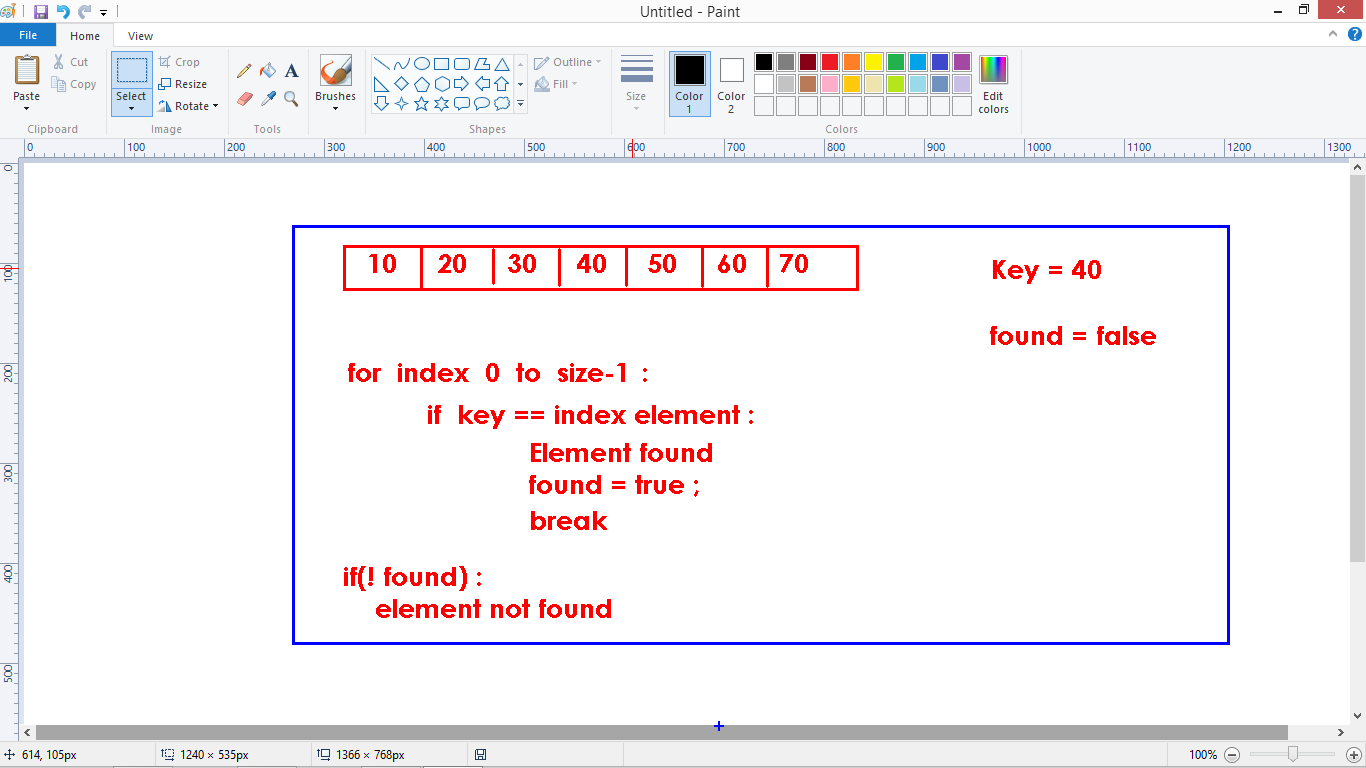
**How to reverse elements in single linked list:**

* It is complex to reverse single linked list.
* Single linked list is not bi-directional.
* We can move forward pointer easily.
* Coming in backward direction must be done through a loop every time.



**Search an element in the array:**

* Compare the key element with index element.
* If match – returns the index as element found
* If not found – display error message “Element not found”



**Code:**

#include<stdio.h>

int main()

{

int arr[6]={10,20,30,40,50,60}, key=30, found=0, i;

for(i=0 ; i<6 ; i++)

{

if(key == arr[i])

{

printf("found @ loc : %d \n", i);

found=1;

break;

}

}

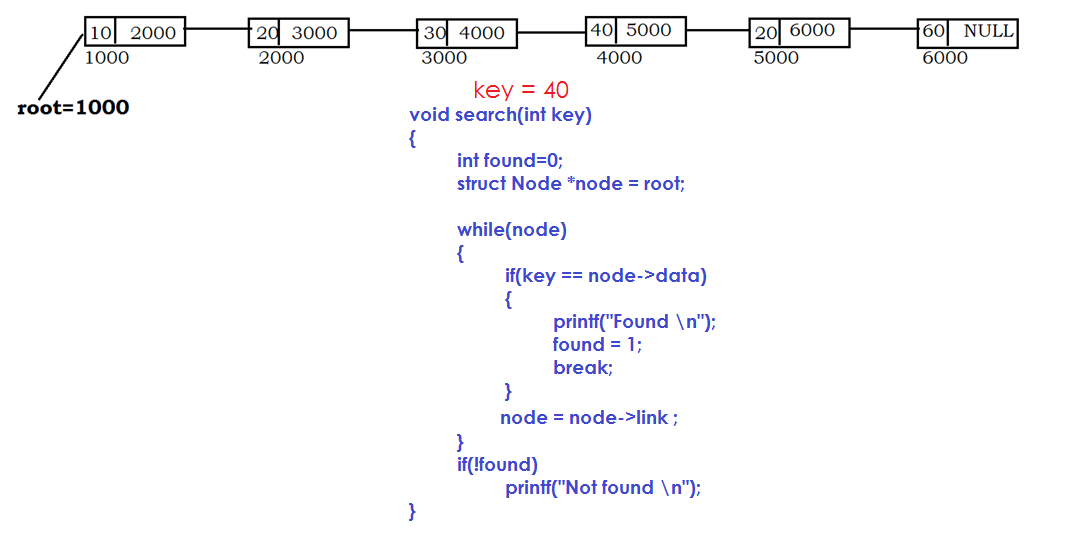
if(!found)

printf("element not found\n");

return 0;

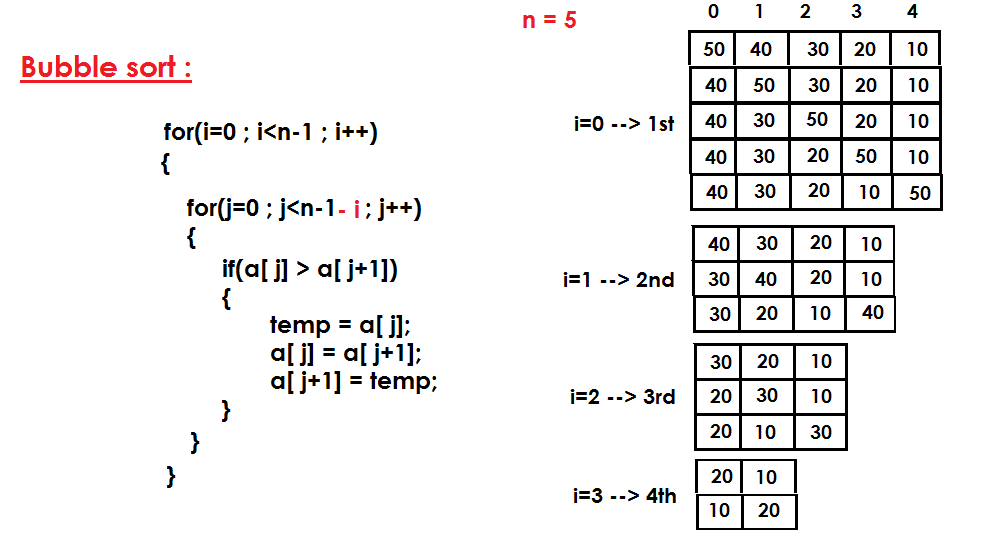
}

**Search an element in the list:**



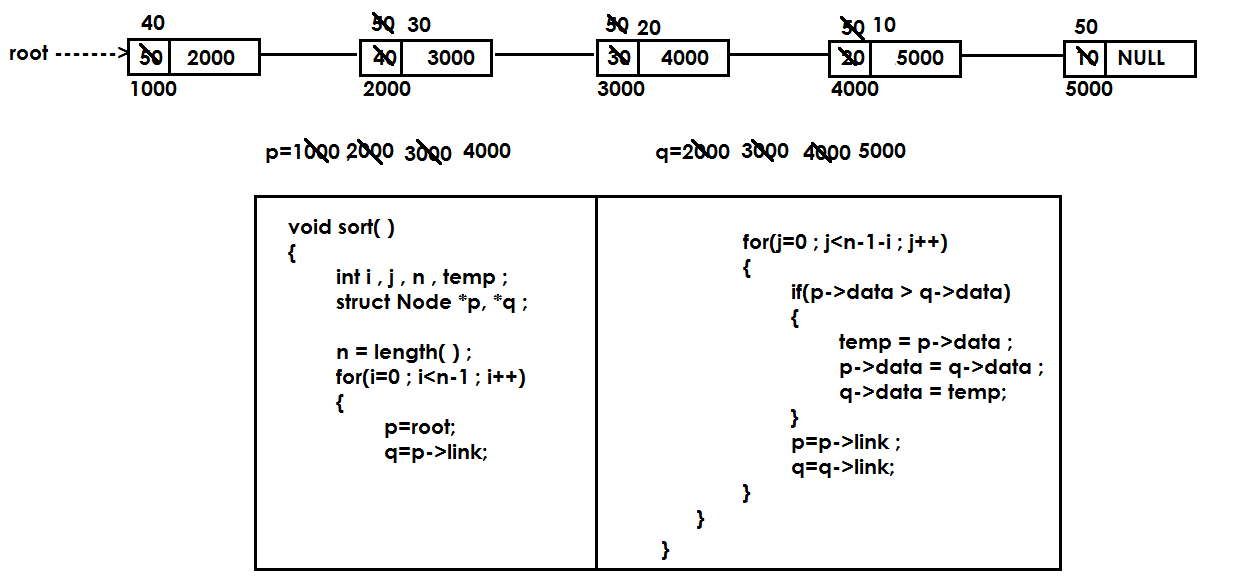
**Sorting single linked list:**

**Bubble sort:**

****

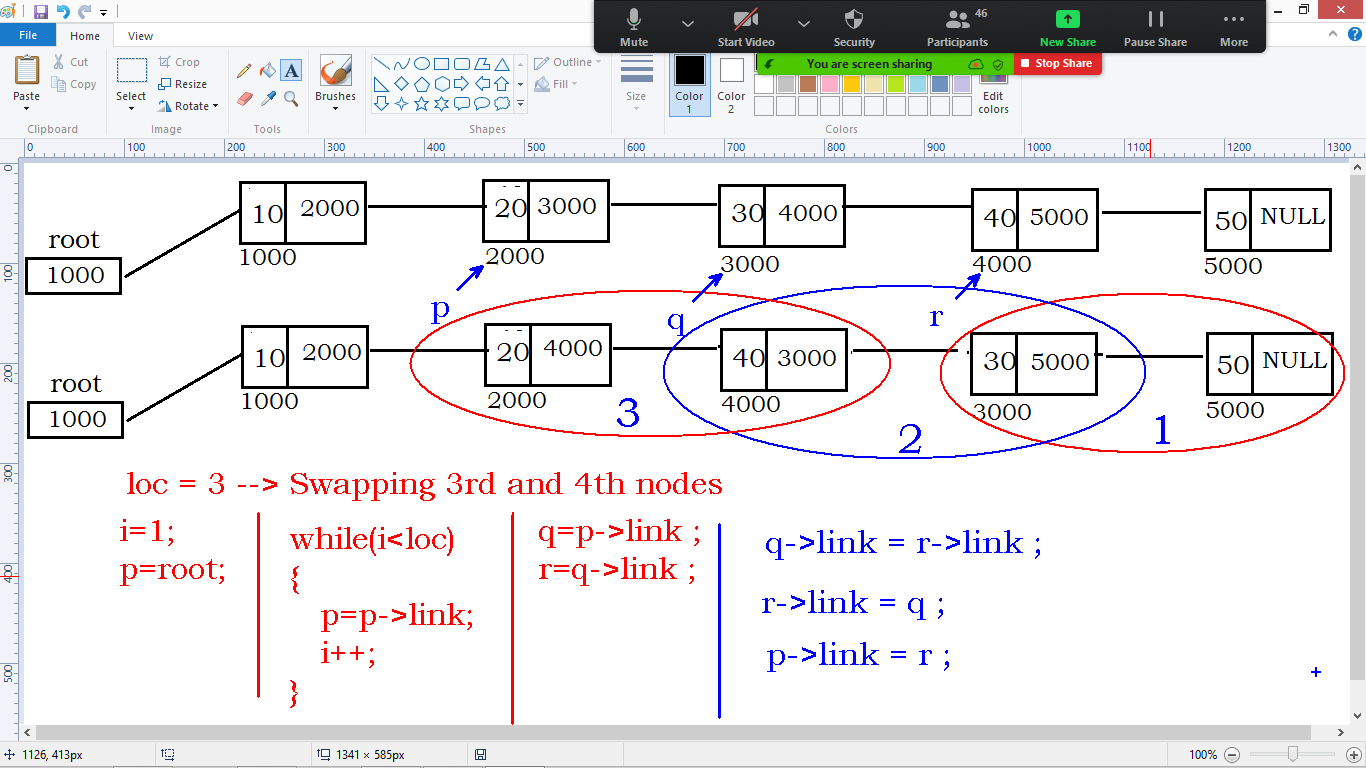
* Sorting is an arrangement of elements either in ascending order or in descending order.
* Bubble sort is a simple sorting technique to sort the elements.
* In every iteration; index element compare with the next element and swapping them if required.
* For each pass, highest element in the array bubbled to last location.
* This process continuous until all elements gets sorted.

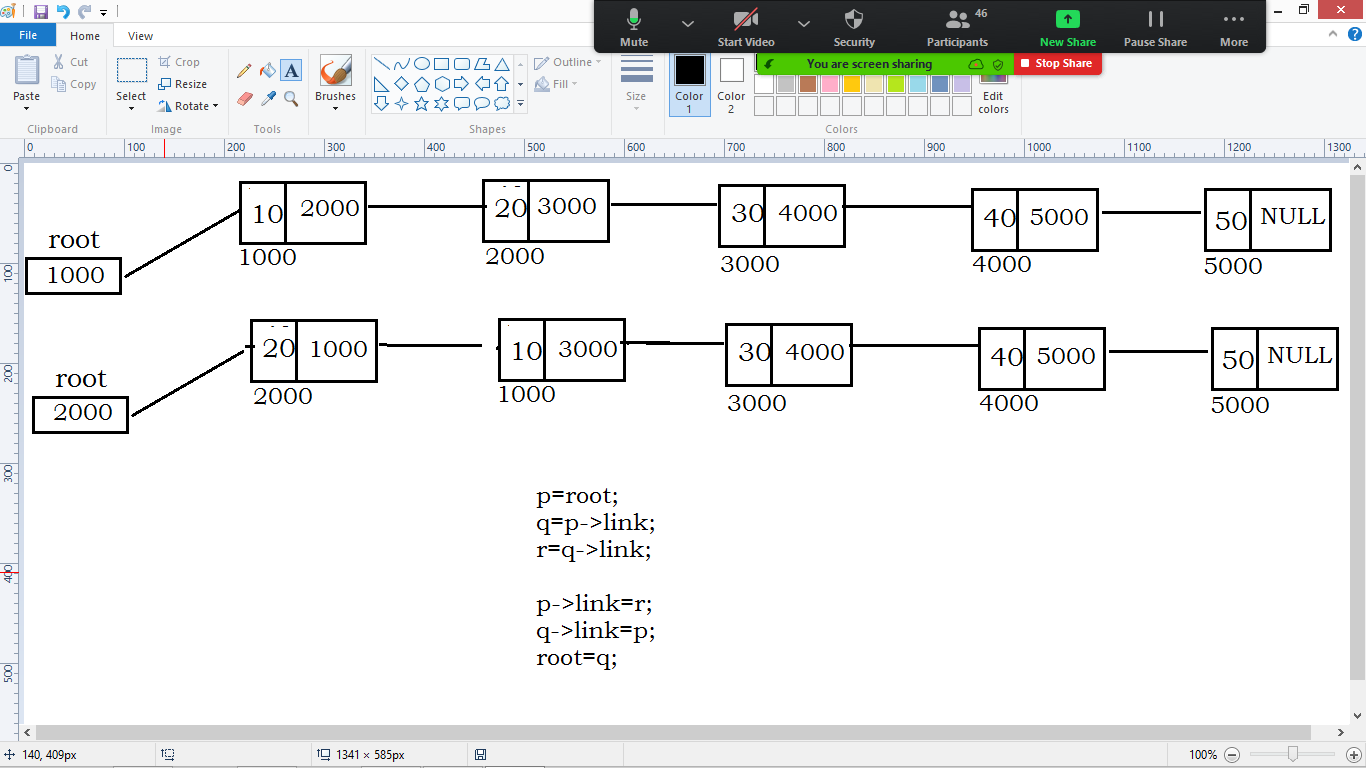
**Sorting list using Bubble sort:**



**Complete Node swapping:**

* We are swapping adjacent nodes.
* We take the location and swapping location node with next node





Sort Linked List:

Create()

Sort()

Delete specified element:

Create()

Delete()

Swap adjacent nodes:

Create()

Swap()