15, 5, 10, 11, 9, 18, 28 (2-3 Tree)

delete(15), insert 21, 12, delete 5,

```
///2-3 tree implementation

#include<stdio.h>
#include<stdlib.h>
typedef struct Node
{
int key[3];
int NKeys;
struct Node *child[4];
struct Node *Parent;
}node;
node *root=NULL;
/*
node *create()
{
node *root;
root=NULL;
return(root);
}
*/

node *NodeCreate(int key,node *Left, node *Right, node *P)
{
        node *Node;
        Node=(node *)malloc(sizeof(node));
        Node->key[0]=key;
        Node->key[1]=32767;
        Node->key[2]=32767;
        Node->child[0]=Left;
        Node->child[1]=Right;
        Node->child[2]=NULL;
        Node->child[3]=NULL;
        Node->Parent=P;
        Node->NKeys=1;
        return(Node);
}

node *Nodesplit(node *Node)
{
        int i;
        node *New1,*New2;
        if(Node==root)
        {
        New1=NodeCreate(Node->key[2],Node->child[2],Node->child[3],NULL);
        New2=NodeCreate(Node->key[1],Node,New1,NULL);
        New1->Parent=New2;
        Node->Parent=New2;
        Node->NKeys=1;
```

```
Node->key[1]=32767;
Node->key[2]=32767;
Node->child[2]=NULL;
Node->child[3]=NULL;
root=New2;
return(New2);
}
if(Node==Node->Parent->child[0])
{
 New1=NodeCreate(Node->key[2],Node->child[2],Node->child[3],Node->Parent);
 Node->Parent->key[2]=Node->Parent->key[1];
 Node->Parent->key[1]=Node->Parent->key[0];
 Node->Parent->key[0]=Node->key[1];
 Node->Parent->child[3]=Node->Parent->child[2];
 Node->Parent->child[2]=Node->Parent->child[1];
 Node->Parent->child[1]=New1;
 Node->key[1]=32767;
 Node->key[2]=32767;
 Node->child[2]=NULL;
 Node->child[3]=NULL;
 Node->NKeys=1;
 Node->Parent->NKeys=Node->Parent->NKeys+1;
 return(Node->Parent);
}
else if(Node==Node->Parent->child[1])
{
 New1=NodeCreate(Node->key[2],Node->child[2],Node->child[3],Node->Parent);
 Node->Parent->key[2]=Node->Parent->key[1];
 Node->Parent->key[1]=Node->key[1];
 Node->Parent->child[3]=Node->Parent->child[2];
 Node->Parent->child[2]=New1;
 Node->key[1]=32767;
 Node->key[2]=32767;
 Node->child[2]=NULL;
 Node->child[3]=NULL;
 Node->NKeys=1;
 Node->Parent->NKeys=Node->Parent->NKeys+1;
 return(Node->Parent);
}
else
{
 New1=NodeCreate(Node->key[2],Node->child[2],Node->child[3],Node->Parent);
 Node->Parent->key[2]=Node->key[1];
 Node->Parent->child[3]=New1;
 Node->Parent->child[2]=Node;
 Node->key[1]=32767;
 Node->key[2]=32767;
 Node->child[2]=NULL;
```

```
        Node->child[3]=NULL;
        Node->NKeys=1;
        Node->Parent->NKeys=Node->Parent->NKeys+1;
        return(Node->Parent);
       }
}

node *Insert(node *Node, int key)
{
        int i;
        if(Node==NULL)
                {
                root=NodeCreate(key,NULL, NULL, NULL);
                return(root);
                }
         if(Node->child[0]!=NULL)
                {
                if(Node->key[0]>key)
                        Node=Insert(Node->child[0],key);
                else if(Node->key[0]<key && Node->key[1]>key)
                        Node=Insert(Node->child[1],key);
                else
                        Node=Insert(Node->child[2],key);
                }
          else
                {
                if(Node->key[0]>key)
                        {
                         for(i=Node->NKeys;i>0;i--)
                                Node->key[i]=Node->key[i-1];
                         Node->key[0]=key;
                        }
                 else if(Node->key[0]<key && Node->key[1]>key)
                        {
                         Node->key[2]=Node->key[1];
                         Node->key[1]=key;
                        }
                 else Node->key[2]=key;
                 Node->NKeys=Node->NKeys+1;
                 }
   return(Node);
   }

   void InsertElement(int key)
   {
           node *Node;
           Node=Insert(root,key);
           while(Node->NKeys > 2)
```

```
                {
                Node=Nodesplit(Node);
                }
}
node *Search(node *Node, int key)
{

if(Node==NULL)
        return(NULL);
else
        {
        if(Node->key[0]>=key)
                {
                if(Node->key[0]==key)
                        return(Node);
                else Node=Search(Node->child[0],key);
                }
        else if(Node->key[0]<key&&Node->key[1]>=key)
                {
                if(Node->key[1]==key)
                        return(Node);
                else Node=Search(Node->child[1],key);
                }
        else
                {
                Node=Search(Node->child[2],key);
                }
        return(Node);
        }
}

node *LeftMostNode(node *Node)
{
while(Node->child[0]!=NULL)
        Node=Node->child[0];
return(Node);
}


void MergeNode(node *Node)
{
node *Node1;
        if(Node==Node->Parent->child[0])
                {
                        Node1=Node->Parent->child[1];
                        Node->key[0]=Node->Parent->key[0];
                        Node->Parent->NKeys=Node->Parent->NKeys-1;
                        Node->key[1]=Node1->key[0];
```

```
                Node->key[2]=Node1->key[1];
                if(Node->child[0]!=NULL)
                {
                Node->child[1]=Node1->child[0];
                Node->child[2]=Node1->child[1];
                Node->child[3]=Node1->child[2];
                }
                if(Node->Parent->child[1]->NKeys==2)
                        Node->NKeys=3;
                else Node->NKeys=2;
                Node->Parent->child[1]=Node->Parent->child[2];
                Node->Parent->child[2]=Node->Parent->child[3];
                Node->Parent->key[0]=Node->Parent->key[1];
                Node->Parent->key[1]=Node->Parent->key[2];
        }
else if(Node==Node->Parent->child[1])
        {
                Node=Node->Parent->child[0];
                Node1=Node->Parent->child[1];
                Node->key[Node->NKeys]=Node->Parent->key[0];
                Node->NKeys=Node->NKeys+1;
                Node->Parent->key[0]=Node->Parent->key[1];
                Node->Parent->key[1]=Node->Parent->key[2];
                Node->Parent->NKeys=Node->Parent->NKeys-1;
                Node->Parent->child[1]=Node->Parent->child[2];
                Node->Parent->child[2]=Node->Parent->child[3];
                if(Node1->child[0]!=NULL)
                        Node->child[Node->NKeys+1]=Node1->child[0];
        }
 else if(Node==Node->Parent->child[2])
        {
                Node=Node->Parent->child[1];
                Node1=Node->Parent->child[2];
                Node->key[Node->NKeys]=Node->Parent->key[1];
                Node->NKeys=Node->NKeys+1;
                Node->Parent->key[1]=Node->Parent->key[2];
                Node->Parent->NKeys=Node->Parent->NKeys-1;
                Node->Parent->child[2]=Node->Parent->child[3];
                if(Node1->child[0]!=NULL)
                        Node->child[Node->NKeys+1]=Node1->child[0];
        }
 free(Node1);
 if(Node->NKeys>2)
        Nodesplit(Node);
 else if(Node->Parent->NKeys==0)
        {
                if(Node->Parent==root)
                        {
```

```
                                Node1==root;
                                root=Node;
                                Node->Parent==NULL;
                                }
                        else  MergeNode(Node->Parent);
                }
}


void Delete(int key)
{
node *Node, *LMNode;
Node=Search(root,key);
int flag=0;
if(Node->key[0]==key)
        {
        if(Node->child[1]!=NULL)
                {
                flag=1;
                LMNode=LeftMostNode(Node->child[1]);
                Node->key[0]=LMNode->key[0];
                }
        else
                {
                Node->key[0]=Node->key[1];
                Node->key[1]=Node->key[2];
                Node->NKeys=Node->NKeys-1;
                }
        }
else
        {
        if(Node->child[1]!=NULL)
                {
                flag=1;
                LMNode=LeftMostNode(Node->child[2]);
                Node->key[1]=LMNode->key[0];
                }
        else
                {
                Node->key[1]=Node->key[2];
                Node->NKeys=Node->NKeys-1;
                }
        }
if(flag==1)
        {
        if(LMNode->NKeys==2)
                {
                LMNode->key[0]=LMNode->key[1];
```

```
                LMNode->key[1]=LMNode->key[2];
                LMNode->NKeys=1;
                }
        else
                {
                LMNode->key[0]=LMNode->key[1];
                LMNode->NKeys=0;
                MergeNode(LMNode);
                }
        }
if(Node->NKeys==0)
        {
        MergeNode(Node);
        }
}

void traversal(node *myNode)
        {
    int i;
    if (myNode!=NULL)
                {
         for (i = 0; i < myNode->NKeys; i++)
                        {
                traversal(myNode->child[i]);
              printf("%d, ", myNode->key[i]);
                        }
                traversal(myNode->child[i]);
         }
         }

 void main()
 {
 InsertElement(15);
 InsertElement(5);
 InsertElement(10);
 InsertElement(11);
 InsertElement(9);
 traversal(root);
 printf("\n");
 InsertElement(18);
 InsertElement(28);
 Delete(15);
 traversal(root);
 printf("\n");
 InsertElement(21);
 InsertElement(12);
 Delete(5);
 traversal(root);
```

```
printf("\n");
}
```