```
public class BinomialHeapImplementation {
public static void main(String[] args)
{
BinoHeap bh=new BinoHeap();
bh.insert(9);
bh.insert(34);
bh.insert(50);
bh.insert(11);
bh.insert(2);
bh.insert(8);
bh.insert(23);
bh.insert(1);
bh.insert(4);
bh.insert(12);
Search sh=new Search();
DecreaseKey(24,1,bh.head,sh);
PrintHeap(bh.head);
System.out.println();
System.out.println(GetMin(bh.head).key);
sh=new Search();
DecreaseKey(1,50,bh.head,sh);
PrintHeap(bh.head);
System.out.println();
System.out.println(bh.ExtractMin());
System.out.println();
```

```
PrintHeap(bh.head);
bh.Delete(9);
System.out.println();
PrintHeap(bh.head);
}

static Node GetMin(Node root)
{
Node res=root;
while(root!=null)
{
root=root.sibling;
if(root!=null && root.key  }
return res;
}

static void MinHeapify(Node root)
{
while(root!=null)
{
Node child=root;
Node res=child;

while(child!=null)
{
```

```
child=child.sibling;
if(child!=null && child.key {
res=child;
    }
   }
if(root.parent!=null && root.parent.key>res.key)
{
int a=res.parent.key;
res.parent.key=res.key;
res.key=a;
   }
root=res.child;
  }
}


static void Heapify(Node root)
{
while(root!=null && root.parent!=null)
{
if(root.key {
int a=root.parent.key;
root.parent.key=root.key;
root.key=a;
   }
root=root.parent;
```

```
}
}


static void DecreaseKey(int newKey,int
oldkey,Node root,Search sh)
{
if(root==null)
{
return;
 else if(root.key==oldkey)
{
sh.isComplete=true;
root.key=newKey;
MinHeapify(root.child);
Heapify(root);
return;
  }


if(!sh.isComplete)
{
DecreaseKey(newKey,oldkey,root.child,sh);
  }


if(!sh.isComplete)
```

```
{
DecreaseKey(newKey,oldkey,root.sibling,sh);
  }

}

static void PrintHeap(Node root)
{
if(root==null)
{
return;
  }

PrintHeap(root.child);
System.out.print(root.key + " ");
PrintHeap(root.sibling);
}

public static class BinoHeap
{
Node head;
int size;

void insert(int key)
{
```

```
if(head==null)
{
head=new Node(key);
size++;
 else
{

Node newNode=new Node(key);
Node last=head;
MergeAndUnite(last,newNode);
size++;
  }
  }

void Delete(int key)
{
Search sh=new Search();
DecreaseKey(Integer.MIN_VALUE,key,head,s
h);
ExtractMin();
  }

int ExtractMin()
{
Node min=GetMin(head);
```

```
int res=Integer.MIN_VALUE;
if(min!=null){res=min.key;}
Node last=min.child;
Node headSibling=min.sibling;
Node sibling=null;
boolean midMin=false;
if(min.sibling!=null)
{
midMin=true;
  }
if(head.key==min.key)
{
head=head.sibling;
midMin=false;
 else
{
Node task=head;
while(task.sibling!=min)
{
task=task.sibling;
  }
task.sibling=null;
  }

while(last!=null)
```

```
{
last.parent=null;
sibling=last.sibling;
last.sibling=null;
if(head==null)
{
head=last;
last=sibling;
sibling=last.sibling;
    }
MergeAndUnite(head,last);
last=sibling;
   }


if(headSibling!=null && midMin)
{
last=headSibling;
while(last!=null)
{
last.parent=null;
sibling=last.sibling;
last.sibling=null;
if(head==null)
{
head=last;
```

```
last=sibling;
sibling=last.sibling;
    }
MergeAndUnite(head,last);
last=sibling;
   }
   }


return res;
  }


void MergeAndUnite(Node last,Node newNode)
{
boolean headSet=false;

while(last!=null)
{
if(last.key {
newNode.sibling=last.child;
last.child=newNode;
newNode.parent=last;
head=last;
last.degree++;
 else if(last.degree==newNode.degree)
{
```

```
Node sibling=last.sibling;
last.sibling=newNode.child;
newNode.child=last;
last.parent=newNode;
newNode.sibling=sibling;
head=newNode;
last=newNode;
newNode.degree++;
 else if(newNode.degree {
newNode.sibling=last;
if(!headSet)
{
head=newNode;
headSet=true;
    }
 else if(newNode.degree>last.degree)
{
Node sibling=last.sibling;
last.sibling=newNode;
newNode.sibling=sibling;
if(!headSet)
{
head=last;
headSet=true;
    }
```

```
}
newNode=last;
last=last.sibling;
  }

  }
}

public static class search
{
boolean isComplete;
search()
{
this.isComplete=false;
  }
}

public static class Node
{
int key;
int degree;
Node sibling;
Node parent;
Node child;
```