```
Node * getMin(H)
 {
   list<Node*>:: iterator it = H.begin();
    Node *tmp = *it;
   while (it != H.end())
    {
       if ((*it)->data < tmp->data)
               tmp = *it;
       it++;
    }
    return tmp;
 }


extractMin(H)
 {  list <Node *> new_H, lo;
    Node * tmp;
    tmp = getMin(H);
    list<Node *> :: iterator it;
    it = H.begin()
     while (it != H.end())
      {
         if (*it != tmp)
             new_H. push_back(*it);

         it++;
      }
     lo = removeMinFromTreeReturnBHeap(tmp);
    new_H = unionBinomialHap(new_H, lo);
   new_H = adjust(new_H);
   return new_H;
}
```

```
removeMinFromTreeReturnBHeap(tree)
{
    list<Node*> H;
    Node *tmp = tree->child;
    Node *lo;
    while(tmp)
    {
        lo = tmp;
        tmp = tmp->sibiling;
        lo->sibiling = NULL;
        H.push_front(lo);
    }
    return H;
}
```

```
insert (H, k):
 {
   Node *tmp = newNode (key);
   return insertATreeInHeap (H, tmp);
 }


insertATreeInHeap ( H, * t)
  {
    list < Node * > tmp;
    tmp.push_back (t);
    tmp = unionBinomialHeap (H, tmp);
    return adjust (tmp);
  }


adjust (H)
 {
  if (H.size() <= 1)
       return H;
  list <Node*> new_H;
  list <Node*>::iterator it1, it2, it3;
  it1 = it2 = it3 = H.begin();
   if (H.size() == 2)
   {
      it2 = it1;
      it2++;
      it3 = H.end()
   }
   else { it2++;
          it3 = it2;
          it3++; }
while (it1 != H.end())
 {
  if (it2 == H.end())
      it1++;
  else if ((*it1)->degree < (*it2)->degree)
   {
     it1++;
     it2++;
     if (it3 != H.end())
        it3++
   }
  else if (it3 != H.end() && (*it1)->degree == (*it2)->degree &&
                              (*it1)->degree = (*it3)->degree)
     {
      it1++; it2++; it3++; }
   else if ((*it1)->degree == (*it2)->degree)
     {
      Node *tmp;
       *it1 = mergeBinomialTrees (*it1, *it2);
        it2 = H.erase (it2);
         if (it3 != H.end())
              it3++;
           }  }
        return H;
   }


MageBinomialTrees (*b1, *b2)
 {   if (b1->data > b2->data)
     swap (b1, b2);
     b2->parent = b1;  b2->sibling = b1->child;  b1->child = b2;
     b1->degree++;
    return b1;
 }
```

```
unionBinomialHeap( h1, h2)
 {    H ← makeBinomialheap ()
  list <Node*> new;
  list <Node*>:: iterator it = h1. begin();
  list <Node*>:: iteratd ot = h2. begin();
  head[H] ← mergeBinomialTrees(h1,h2)
   if (head[H] = NIL)
       return H
   prev-x ← NIL
    x ← head[H]
   next -→x ← sibling(x)
   while next-x ≠ NIL
        do{ if (degree[x] ≠ degree[next-x]) || (sibiling[next-x] ≠ NIL
                            && degree (sibling[next-x]) = degree[x])
           {
               prev-x ← x
               x ← next-x
           }
          else if (key[x] ≤ key[next-x])
             {
               sibling[x] ← sibiling[next-x]
            binolink (next-x, x)
             }
           else if (prev-x = NIL)
             {  head[H] ← next-x
             }.
            else { sibiling[prev-x] ← next x }
              binolink (x, next-x)
            x ← next-x
          next-x ← sibiling[x]
       }
    return H
 }
```