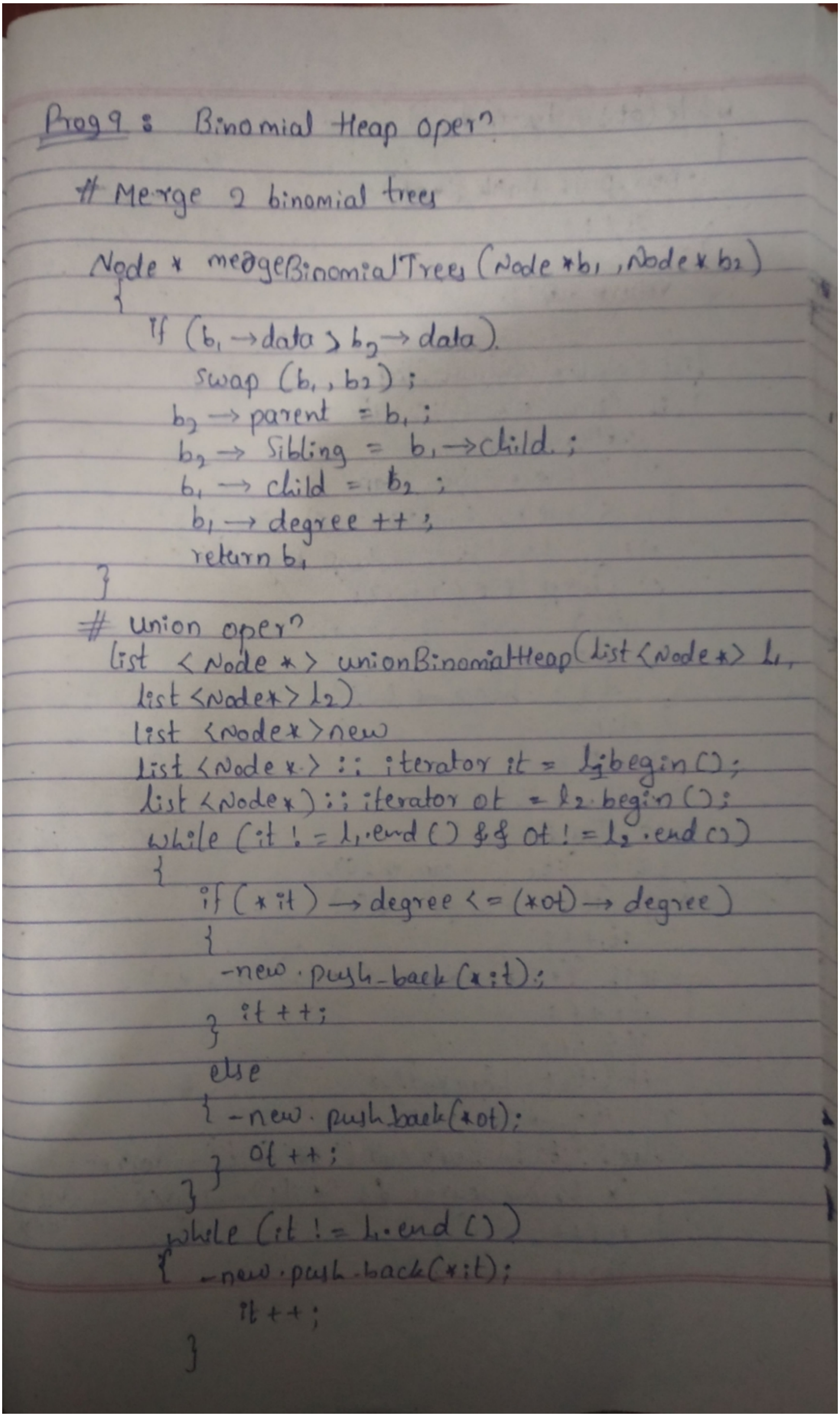


Prog 9 : Binomial Heap oper^n

Merge 2 binomial trees

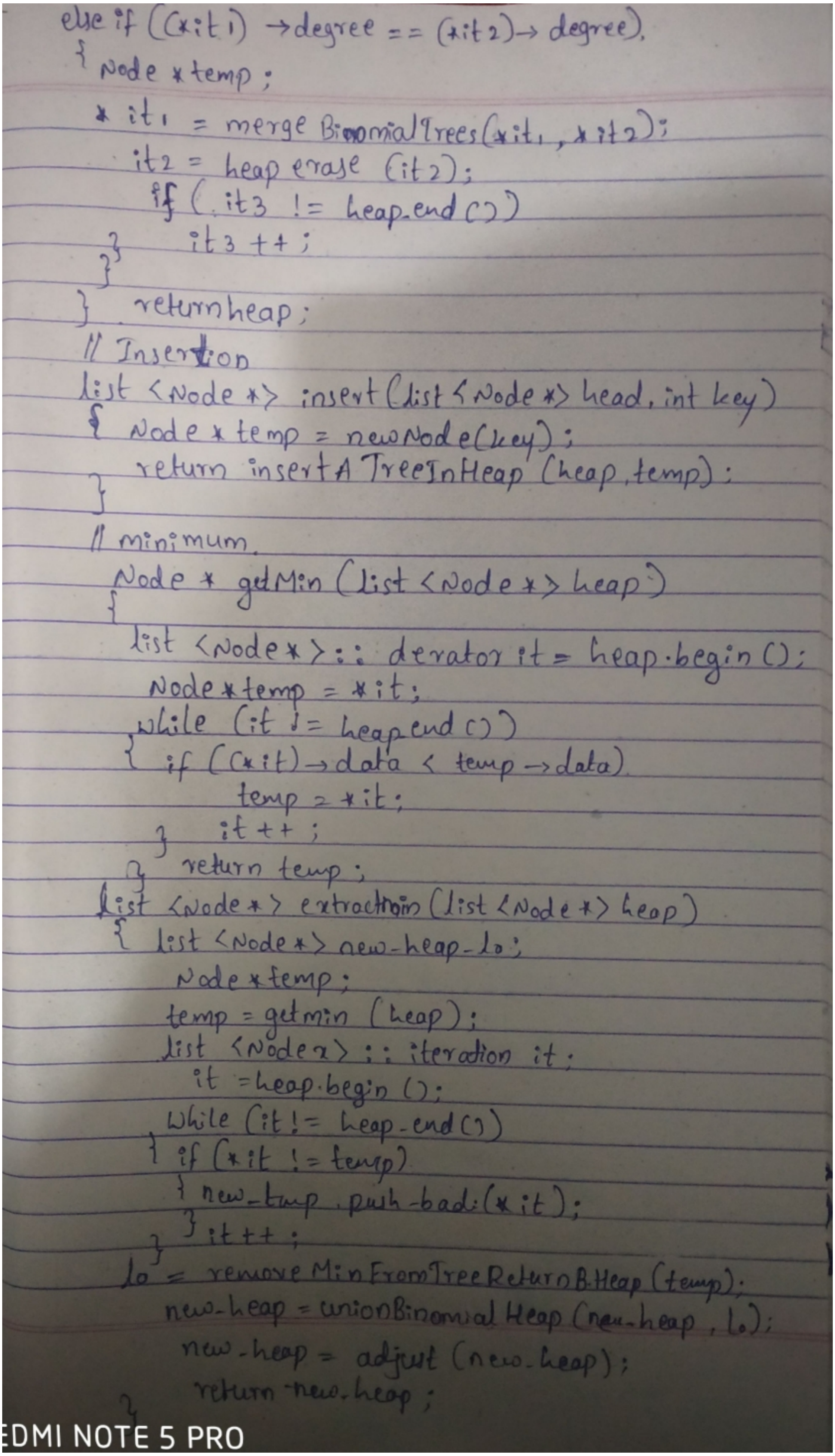
```
Node * meageBinomialTrees (Node *b1 ,Node* b2)
 {
    If (b1 -> data > b2 -> data).
       swap (b1 , b2) ;
     b2 -> parent = b1 ;
     b2 -> Sibling = b1 -> child. ;
     b1 -> child = b2 ;
     b1 -> degree ++ ;
     return b1
}
```

union oper^n

```
list <Node *> unionBinomialHeap(list <Node *> l1,
  list <Node*> l2)
  list <Node*>new
  list <Node *> :: iterator it = l1begin();
  list <Node*) :: iterator ot = l2.begin();
  while (it != l1.end() && ot != l2.end())
  {
      if (*it) -> degree <= (*ot) -> degree)
      {
        -new.push_back(*it);
         it ++;
      }
      else
      { -new.push_back(*ot);
         ot ++;
      }
   }
   while (it != l1.end())
  { -new.push_back(*it);
       it ++;
   }
```

```
while (ot ! = l2 · end())
{
  -new.push-back (*ot);
     ot++;
}
   return new;
}

 list <Node*> adjust (list <Node*> heap)
 { if (heap size () < = 1)
      return heap;
   list <Node *> new heap;
   list <Node *> : : iterator it1, it2, it3;
    it1 = it2 = it3 = heap-begin();
    if (heap-size () == 2)
    { it 2 = it1 ;
       it2 ++;
       it3 = heapend ();
   } else
    { it2 ++;
      it3 = it2;
     } it3 ++;
    while (it1 != heap-end())
   { if (it2 = = heap-end())
        it1++;
   else if ((*it1)->degree < (*it2)->degree)
   { it1 ++;
     it2 ++;
     if (it3 ! =heap end())
    }   it3 ++;
    else if (it3 != heap end () && (*it1)->degree ==
       (*it2)->degree && (*it1)->degree == (*it 3)->
                                                degree)
    { it1++;
      it2++;
      it3++;
    }
```



```
else if ((*it1) ->degree == (*it2)-> degree).
 { Node *temp ;
   * it1 = merge BinomialTrees(*it1 , * it2);
     it2 = heap erase (it2);
       if ( it3 != heap.end ())
   }      it3 ++;
  }
 } returnheap;
 // Insertion
 list <Node *> insert(list <Node *> head, int key)
 { Node * temp = newNode(key);
   return insertATreeInHeap (heap, temp);
 }
 // minimum.
  Node * getMin (list <Node *> heap)
  {
   list <Node *>:: derator it = heap.begin ();
    Node *temp = *it;
    while (it != heap.end ())
   { if ((*it)->data < temp->data)
          temp = *it;
     }   it ++;
   }  return temp;
 list <Node *> extractmin (list <Node *> heap)
  { list <Node *> new-heap-lo;
     Node *temp;
     temp = getmin (heap);
     list <Nodex> :: iteration it;
       it =heap.begin ();
     While (it != heap-end ())
    { if (*it != temp)
      { new_tmp .push-bad:(*it);
       } it ++;
    }
  lo = removeMinFromTreeReturnB.Heap (temp);
     new-heap = unionBinomial Heap (new-heap , lo);
      new-heap = adjust (new-heap);
       return new-heap ;
  }
```