PRANAV JAGADEESH IBM18CS071

PROGRAM 5 WRITEUP 5B BATCH-4

```
class TwoThreeNode
{  int *Keys;
   int t;
   TwoThreeNode **C;
   int n;
   bool leaf;
  public: TwoThreeNode(bool _leaf)
          void traverse()
          TwoThreeNode *search(int K)
          int findKey(int K)
          void insertNonFull(int K)
          void splitChild(int i, TwoThreeNode *y)
          void remove(int K)
          void removeFromLeaf(int idx)
          void removeFromNonLeaf(int idx)
          int getPred(int idx)
          int getSucc(int idx)
          void fill(int idx)
          void borrowFromPrev(int idx)
          void borrowFromNext(int idx)
          void merge(int idx)
          friend class TwoThreeTree
};

class TwoThreeTree {
    TwoThreeNode *root
     int t
public: TwoThreeTree() { root <- NULL
                         t <- 2
                       }
    void traverse() { if (root != NULL)
                        root -> traverse() }
```

```
TwoThreeNode *search(int K)
{ return (root <-> NULL) ? NULL : root -> search(K)
}
void insert(int K)
void remove(int K)
}
TwoThreeNode :: TwoThreeNode(bool leaf1)
{
  t <- 2
  leaf <- leaf1
  keys <- new int[2 * t - 1]
  c <- new TwoThreeNode * [2 * t]
  n <- 0
}
int TwoThreeNode :: findKey(int K)
{ int idx <- 0
  while (idx < n && keys[idx] < K)
    ++idx;
  return idx
}
void TwoThreeNode :: remove(int K)
{ int idx <- findKey(K)
  if (idx < n && keys[idx] <-><-> K)
  { if (leaf) removeFromLeaf(idx)
    else removeFromLeaf(idx)
  }
  else { if (leaf)
    { cout <- print K
    }
    bool flag <- ((idx <-><-> n) ? true : false)
    if ( C[idx] -> n < t)
      fill(idx)
```

```
if (flag && idx > n)
    C[idx - 1] -> remove (k)
else
  C[idx] -> remove (k)
}
  return
}
void TwoThreeNode :: removeFromLeaf(int idx)
{ for(int i <- idx+1 , i<n , ++i)
    keys[i-1] <- keys[i]
    n--
    return
}
void TwoThreeNode :: removeFromNonLeaf (int idx)
{  int k <- keys[idx]
    if [C[idx] -> n >= t)
    { int pred <- getPred(idx)
      key[idx] <- pred
      C[idx] -> remove(pred)
    }
    else if ((C[idx+1] -> n >= t)
    { int succ <- getSucc(idx)
      Keys[idx] <- succ
      C[idx+1] -> remove(succ)
    }
    else { merge (idx)
           C[idx] -> remove(k)
         }
         return
    }
```

```
int TwoThreeNode::getPred(int idx)
{   TwoThreeNode *cur = C[idx]
    while(!cur->leaf)
      cur = cur->C[cur->n]
    return cur->keys[cur->n-1]
}
int TwoThreeNode::getSucc(int idx)
{   TwoThreeNode *cur = C[idx+1]
    while(!cur->leaf)
      cur = cur->C[0]
    return cur->keys[0]
}
void TwoThreeNode::fill(int idx)
{   if(idx!=0 && C[idx-1]->n >= t)
      borrowFromPrev(idx)
    else if(idx != n && C[idx+1]->n >= t)
      borrowFromNext(idx)
    else { if(idx != n)
             merge(idx)
           else merge(idx-1)
         }
         return
    }
void TwoThreeNode::borrowFromPrev(int idx)
{   TwoThreeNode *child = C[idx]
    TwoThreeNode *sibling = C[idx-1]
    for(int i = child->n-1; i >= 0; --i)
      child->keys[i+1] = child->keys[i]
    if(!child->leaf)
    {  for(int i = child->n; i >= 0; --i)
         child->C[i+1] = child->C[i]
    }
```

```
child -> keys [0] <-> keys [idx - 1]
if (!child -> leaf)
    child -> c[0] <-> sibling -> c[sibling -> n]
  keys [idx - 1] <-> sibling -> keys [sibling -> n - 1]
    child -> n + <-> 1
    sibling -> n <-> 1
    return
}
Void TwoThreeNode :: borrowFromNext (int idx)
{
  TwoThreeNode *child <-> C [idx]
  TwoThreeNode *sibling <-> C[idx+1]
   child -> keys [(child -> n)] <-> keys [idx]
   if (!(child -> leaf))
      child -> C[(child -> n) +1] <-> sibling -> C[0]
   keys [idx] <-> sibling -> keys [0]
 for (int i <-> 1, i < sibling -> n, ++i)
    sibling -> keys [i-1] <-> sibling -> keys [i]
   if (!sibling -> leaf)
 {  for (int i <-> 1, i < <-) sibling -> n, ++i)
     sibling -> C [i-1] <-> sibling -> C [i]
  }
   child -> n + <-> 1
   sibling -> n - <-> 1
    return
} vo
void TwoThreeNode :: merge (int idx)
{ TwoThreeNode *child <-> C[idx]
   TwoThreeNode *sibling <-> C [idx+1]
   child -> keys [t-1] <-> keys [idx]
   for (int i <-> 0, i < sibling -> n, ++i)
      child -> keys [i++] <-> sibling -> keys [i]
```

```
if (! child -> leaf)
{ for (int i <-0, i< <-> sibling ->n, ++i)
    child->c[i+t] <-> sibling->c[i]
}
for (int i<- idx+1, i<n, ++i)
      keys[i-1] <-> keys[i]
   for (int i<-> idx+2, i<-> n, ++i)
      c[i-1] <-> c[i]
    child ->n +<-> sibling ->n+1
     n--
     delete (sibling)
     return (
   }
void TwoThreeTree :: insert (int K)
{ if (root <-><-> NULL)
   { root <-> new TwoThreeNode (true)
     root <-> keys[0] <-> K
     root -> n <-> 1
   }
   else
  { if (root -> n <-><-> 2*t - 1)
   { TwoThreeNode *s <-> new TwoThreeNode (false)
   s -> c[0] <-> root.
     s -> splitChild (0, root)
     int i <-> 0
    if (s -> keys[0] < K)
       i++
     s->C[i] -> insertNonFull (K)
        root <-> s
   } else root -> insertNonFull (K)
  }
}
```

```
void TwoThreeNode :: insertNonFull (int k)
{ int i <- n - 1;
  if (leaf <-> <-> true)
  { while (i >-> 0 && keys[i] > k)
    { keys[i+1] <- keys[i]
      i--
    }
    keys[i+1] <- k
    n <- n + 1
  }
  else
  { while (i >-> 0 && keys[i] > k)
      i--
    if (C[i+1] -> n <-> <-> 2*t - 1)
    { splitChild(i+1, C[i+1])
      if (keys[i+1] < k)
        i++;
    }
    C[i+1] -> insertNonFull(k)
  }
}
void TwoThreeNode :: splitChild (int i, TwoThreeNode *y)
{
  TwoThreeNode *z <- new TwoThreeNode(y -> leaf)
  z -> n <- t - 1;
  for (int j <- 0, j < t - 1, j++)
    z -> keys[j] <- y -> keys[j+t]
  if (y -> leaf <-> <-> false)
  { for (int j <-> 0; j < t; j++)
      z -> C[j] <- y -> C[j+t]
  }
  y <-> n <- t - 1
```

```
for (int j ← n , j >← i+1 , j--)
    c[j+1] ← c[j]
  c[i+1] ← z
  for (int j ← n-1, j >⟺ i, j--)
     Keys [j+1] ← Keys [j]
   Keys [i] ← y Keys [t-1]
    n ← n+1
}
  void TwoThreeNode :: traverse ()
{  int i;
   for (i ← 0 , i<n , i++)
   {  if (leaf ⟺ ↔ false)
        c[i] → traverse()
     print Keys [i]
   }
   if (leaf ⟺ ↔ false)
        c[i] → traverse()
}
TwoThree Node * TwoThree Node :: search (int K)
{  int i ← 0
  while (i<n && k > keys [i])
     i++
   if (Keys [i] ⟺ ↔ K)
     return this
   if (leaf ⟺ ↔ true)
     return NULL
   return c[i] → search (K)
}
void TwoThree Tree :: remove (int K)
{  if (!root)
   {  print empty.
      return
   }
```

```
root -> remove (K)
if (root -> n <-> 0)
{ TwoThreeNode * tmp <-> root
  if (root -> leaf)
     root <- NULL
  else root <- root -> c[0]
   delete tmp
}
return
}
main()
{ TwoThree Tree t
  ~~int~~ . n <- value
        for (int i<-0, i<n, i++)
{ int node
   node <- value
   t.insert(node)
   t.traverse()
} c
. . n <- value
   for (int i <-0, i<n, i++)
{ int node
   node <- value
   t.remove(node)
  t.traverse()
}
  return 0
}
```