山东大学 计算机科学与技术 学院

数据结构与算法课程设计 课程实验报告

学号：202200130039	姓名：张向荣	班级：22.3
上机学时：4 小时	实验日期：2024.3.25-4.7	

课程设计题目：
跳表实现与分析

软件开发环境：
Dev-c++

报告内容：

1.需求描述

1.1 问题描述

实现并分析跳表结构。

1.2 基本要求

1. 构造并实现跳表 ADT，跳表 ADT 中应包括初始化、查找、插入、删除指定关键字的元素、删除关键字最小的元素、删除关键字最大的元素等基本操作。
2. 生成测试数据并验证你所实现的跳表结构的正确性。
3. 分析各基本操作的时间复杂性。
4. 对跳表维护动态数据集合的效率进行实验验证。

设计并生成一个操作序列，操作序列中包含插入、删除指定关键字的元素、删除关键字最小的元素、删除关键字最大的元素、查找操作，可设总共操作的次数是 M 次（操作数据随机生成）；
随机产生 N 个数据并将其初始化为严格跳表；
在以上跳表的基础上，依次执行 M 次操作，统计单个操作序列中各个操作执行所需的平均时间（以元素的比较次数衡量），获得随着 M 增加而导致操作时间的变化情况。分析产生这样变化的原因。当操作时间大到一定程度后应进行跳表的整理操作，设计相应的整理算法，并从数量上确定何时较为合适。观察在添加整理操作后执行时间的变化情况。

1.3 输入说明

输入文件

第一行两个数 M , N，分别表示操作个数和初始化跳表长度 第二行 N 个数为初始化跳表所用元素，保证没有重复数据。接下来 M-1 行分别为跳表各个操作。
具体操作为:

- 1 num，查找跳表中是否含有元素 num，含有则输出 YES，否则输入 NO
- 2 num，向跳表中插入元素 num，并输出跳表中所有元素的异或和
- 3 num，将跳表中的元素 num 删除，并输出跳表中所有元素的异或和

- 4，删除跳表中的最小元素，并将该元素输出
- 5，删除跳表中的最大元素，并将该元素输出输出文件

按要求输出，每个操作输出一行

输入界面设计

首先选择随机生成数据还是手动输入。

选择随机生成则由随机函数生成数据，选择手动输入则自己输入。

2.分析与设计

2.1 问题分析

题目让我们实现并分析跳表结构，所以我们要理解什么是跳表。

首先跳表是一种随机化的数据结构，它可以在有序链表基础上实现快速查询。跳表通过在原有有序链表上增加多级索引来实现快速查找，这些索引使得跳表可以进行类似二分查找的操作。调表的每一层都是其下一层子集的排序列表，且上一层的数据间隔比下一层大，这种结构使得在搜索可以从上层开始，一旦确定某个数据位于某层中的某个范围，就可以转移到下一层进行更精确的查找。

2.2 主程序设计

定义一个跳表节点结构体，然后定义一个跳表类，设计函数实现相关功能。

2.3 设计思路

实现跳表的相关功能，采用面向对象的思路。使用 rand()函数生成随机数据。定义一个变量计算每次操作的比较次数，最后求出平均比较次数，获得随着 m 增加而导致的时间变化情况。

2.4 数据及数据类型定义

定义一个节点结构体，使用模板类，元素为 T 类型。

定义跳表类，其中定义一个 float 类型的变量，用于确定级号，节点指针都是 skipNode<T>*类型。

2.5.算法设计及分析

首先定义 skiplist.h 文件。定义一个节点结构体，里面包含元素值，指针数组，构造函数。

然后定义一个跳表类，在其中定义包括插入、删除、查找、搜索等函数。

定义一个 skiplist.cpp 文件，里面实现函数的相关功能。

在构造函数中进行初始化，定义一个区间值，以及跳表所能创建的最大级数。并创建头节点，尾节点、数组 last，对每层进行初始化。

在查找函数中，如果传入的参数大于等于最大元素，直接返回。否则从最高级链表开始查找，直到 0 级链表。在每一级链表中，从左边尽可能逼近要查找的元素，在查找时当指针指向下一个节点值大于等于传入的值时，向下一级查找，直到到 0 级链表。当循环结束退出时，指针正好在要查找的元素的左边，与下一个元素比较，如果等于传入的值，则查找到，否则没有查找到。

在插入函数和删除函数实现之前，定义一个搜索函数。和查找操作相似，每到一个元素下一个节点的元素大于等于传入的参数时，用 last 数组记录该级最后查找的节点位置，然后向下一级查找，直到第 0 级链表，返回最后一个查找节点的下一个节点。

在插入函数实现之前，编写级的分配函数。通过 rand()函数产生随机数，如果落在区间

内，则增加一级，最后判断级数和最大级数，返回其中较小的值作为级数。

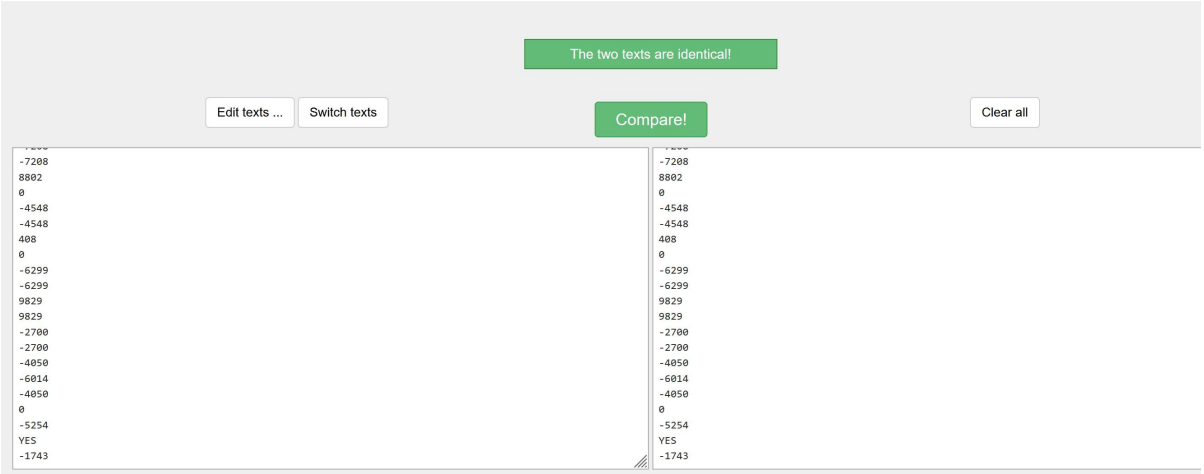
在插入操作中，如果传入的元素大于等于最大元素，直接返回。然后通过搜索函数找到传入元素对应位置的节点，以及用 last 数组记录该级最后查找的节点位置。如果节点元素的值等于传入的参数值，说明已经存在，直接返回。否则，通过级的分配函数，为新数指定它所属的级链表。先判断新节点的最高级和原最高级大小，如果原最高级小于新节点的最高级，更新最高级为新节点的最高级。然后从它分配的最高级开始在 last 数组记录的最后查找的节点后逐级插入节点。

在删除操作中，如果传入的元素大于等于最大元素，直接返回。然后通过搜索函数找到传入元素对应位置的节点，以及用 last 数组记录该级最后查找的节点位置。如果节点元素的值不等于传入的参数值，说明不存在，返回 false。否则，将每级 last 所指向的节点后移一个。将节点 delete。然后判断最高级的变化，如果最高级没有元素，将最高级删除。

在删除最小元素操作中，最小元素就是在第 0 级中 headerNode 所指向的元素。从最高级开始向下遍历，如果 headerNode 所指向的下一个元素是最小元素，则 headerNode 指向最小元素的下一个元素，将要删节点 delete。然后判断最高级的变化，如果最高级没有元素，将最高级删除。

在删除最大元素操作中，最大元素是第 0 级最后一个元素，通过遍历找到最大元素，并用 last 数组记录最后查看的节点，然后将要删除的元素传入 erase()函数中。

3. 测试



4. 分析与探讨

跳表采用随机技术决定元素应该在哪几层，其中插入、删除、查找操作的时间复杂度均为 O(logn)，然而最坏的情况下时间复杂度却变成 O(n)，相比之下在一个有序数组或链表中进行插入、删除操作的时间为 O(n)，最坏情况为 O(n)。因此当跳表的性能下降时，我们需要对跳表进行重构。

在删除时可能将最高级元素删除，这样我们无法像二分法一样很好的实现对元素的查找。因此需要进行重构。

5. 附录：实现源代码

```
#include<bits/stdc++.h>
```

```
#include"skiplist.h"
using namespace std;


template<class T>
skipList<T>::skipList(T largeElement,int maxs,float prob)
{//关键字小于 largeElement,元素个数最多为 maxs;0<prob<1
        cutOff=prob*RAND_MAX;//0x7ffff
        maxLevel=(int)ceil(logf((float)maxs));//返回大于等于自然对数的最小整数
        //最大级数

        levels=0;//对级号进行初始化
    dSize=0;
    tailElement=largeElement;

    //创建头节点， 尾节点以及数组 last
    headerNode =new skipNode<T>(tailElement,maxLevel+1);
    tailNode     =new skipNode<T>(tailElement,0);

    last= new skipNode<T> *[maxLevel+1];
    for(int i=0;i<=maxLevel;i++)
       headerNode->next[i]=tailNode; //对每层初始化

}
//小美女
template<class T>
T* skipList<T>::find(const T& theElement)
{
  //返回关键字 theElement 匹配的元素的指针
  //若不存在匹配的元素，返回 null

  if(theElement>=tailElement)//大于范围
   return NULL;

   //指针 beforeNode,指向可能与 theElement 匹配节点的前一节点
   com_ele=0;
   skipNode<T>* beforeNode=headerNode;
   for(int i=levels;i>=0;i--)
   { //在第 i 级链中搜索
        while(beforeNode->next[i]->element<theElement)
        {
            beforeNode=beforeNode->next[i];
             com_ele++;
        }
   if(beforeNode->next[i]==tailNode) com_ele--;//到达节点最后的位置
        com_ele++;
```

```
    }
    //遍历到了第 0 层
    //检查是否下一个节点拥有关键值 theElement
    if(beforeNode->next[0]->element==theElement)
    {
     return &beforeNode->next[0]->element;
    }

    return NULL; //不存在匹配的值
  }

template<class T>
skipNode<T> *skipList<T>::search(const T& theElement)
{
  //搜索 theElement ,并保存每一级链最后查看的节点位置在数组 last 中
  //返回可能含有 theElement 的节点位置

  //指针 beforeNode,指向可能与 theElement 匹配节点的前一个节点
   com_ele=0;
  skipNode<T>* beforeNode=headerNode;
  for(int i=levels;i>=0;i--)
  {
       while(beforeNode->next[i]->element<theElement)
       {
        beforeNode=beforeNode->next[i];
        com_ele++;
       }
       if(beforeNode->next[i]==tailNode) com_ele--;//到达节点最后的位置
       com_ele++;

       last[i]=beforeNode;//i 级链最后查看的节点位置,theElement(存在)前的节点
  }

  return beforeNode->next[0];
  //返回的位置是插入时 theElement 应该插入的位置，删除时 theElement 所在的位置
}

template<class T>
int skipList<T>::level()const
{
  //产生一个小于等于 maxlevel 的随机级号
  int lev=0;
  //产生随机数
  while(rand()<=cutOff) lev++;//prob 概率
  return (lev<=maxLevel)?lev:maxLevel;
}
```

```
template<class T>
void skipList<T>::insert(const T &theElement)
{
  //插入一个数字 theELement
  if(theElement>=tailElement)//关键值太大，不插入
  {
      return ;
  }
  //查看 theElement 是否已经存在
  skipNode<T> *theNode=search(theElement);//找到相应位置

  if(theNode->element==theElement)//已存在
  {
      return ;
  }

  //不存在，确定新节点的级
  int theLevel =level();
  if(theLevel>levels)
  {
      theLevel=++levels;
      last[theLevel]=headerNode;
  }

  //产生新节点，并插入
  skipNode<T>* newNode=new skipNode<T>(theElement,theLevel+1);
  for(int i=0;i<=theLevel;i++)
  {
      //插入到第 i 级链
      newNode->next[i]=last[i]->next[i];
      last[i]->next[i]=newNode;
  }
  dSize++;
  return ;
}

template<class T>
bool skipList<T>::erase(const T& theElement)
{
  //删除 theELement
  if(theElement>=tailElement) return false;
  skipNode<T>* theNode=search(theElement);
  if(theNode->element!= theElement) return false;//不存在
  //从跳表中删除节点
  //last 存储 theElement 前的元素
```

```
  for(int i=0;i<=levels&&last[i]->next[i]==theNode;i++)
   last[i]->next[i]=theNode->next[i];
   //更新 levels
   while(levels>0&&headerNode->next[levels]==tailNode)//此层就一个值
    levels--;

   delete theNode;
   dSize--;
   return true;
}

template<class T>
T skipList<T>::erasemin()
{
  skipNode<T>* theNode=headerNode->next[0];
  for(int i=0;i<=levels&&headerNode->next[i]==theNode;i++)
  {
   headerNode->next[i]=theNode->next[i];//将要删节点层的头节点
    //指向它下一个节点
    com_ele++;
  }


  //更新 levels
  while(levels>0&&headerNode->next[levels]==tailNode)
   levels--;//如果最高层只一个那就是原来最小的节点，将最高层整层删除
    //输出
     T data=theNode->element;
  delete theNode;
  dSize--;
  return data;
}

template<class T>
T skipList<T>::erasemax()
{
  skipNode<T>*beforeNode=headerNode;
  for(int i=levels;i>=0;i--)
  {
       while(beforeNode->next[i]->element<tailElement)
       beforeNode=beforeNode->next[i];
        last[i]=beforeNode;//i 级链最后查看的位置
  }
    T data=beforeNode->element;
  erase(beforeNode->element);
  return data;
```

```
}

template<class T>
long long skipList<T>::output()//输出元素异或和
{
  long long sum=0;
  for(skipNode<T>* currentNode=headerNode->next[0];currentNode!=tailNode;
  currentNode=currentNode->next[0])
  {
     sum^=currentNode->element;
  }
      return sum;
}



template<class T>
void skipList<T>::rebuild()
{
      vector<T> v;
  for(skipNode<T>* currentNode=headerNode->next[0];currentNode!=tailNode;
  currentNode=currentNode->next[0])
  {
       v.push_back(currentNode->element);
  }
  int len=dSize;
  dSize=0;
  levels=0;
      last= new skipNode<T> *[maxLevel+1];
  for(int i=0;i<=maxLevel;i++)
     headerNode->next[i]=tailNode; //对每层初始化
      for(int i=0;i<len;i++)
      {
        insert(v[i]);
  }
}
```