实 验 报 告

实验题目：B+树的模拟
日期：2020 年 11 月 14 日
院（系）：数据科学与计算机学院
专业（班级）： 计算机科学与技术（超算）

一. 实验目的

完成一个 B+树仿真器程序的系统设计、实现和报告。

二. 实验环境

本实验可基于Visual Studio Code等平台开发，参考主流的编码规范，如Google C++Style Guide（中文版）

2.1 编程语言和开发工具

编程语言： ANSI C/C++
开发工具： Visual Studio Code、编译器G++

2.2 编码规范

遵循良好的程序设计风格来设计和编写程序。基本编码规范：

1. 标识符的命名要到达顾名思义的程度；
2. 关键代码提供清晰、准确的注释；
3. 程序版面要求：
   a) 不同功能块用空行分隔；
   b) 一般一个语句一行；
   c) 语句缩进整齐、层次分明。

三、实验要求

完成一个 B+树仿真器程序的系统设计、实现和报告，具体包括：

0. 描述 B+树的概念和 B+树的逻辑结构
1. 用一个大小为 40Bytes 的内存单元模拟一个外部存储块，规定关键字大小为 4Bytes，地址大小为 4Bytes，记录信息数据大小为 8Bytes。确定上述 B+树的 M 值（用于内部 M-路搜索树）和 L 值（用于每个叶子块存储的记录数目）
2. 设计存储上述 B+树的数据结构设计 （程序设计语言描述）
3. 设计 B+树用于记录查找、插入和删除的算法（用伪代码描述），包含一个自行设计的 20ms 延时器模拟一次外部存取的时间延迟
4. 列出源代码各个主模块命名清单（不需要代码清单）
5. 测试用例设计（初始化至少包含 50 个记录数据）
6. 运行结果分析，包括外部存取延时统计

四、实验内容

1、描述 B+树的概念和 B+树的逻辑结构

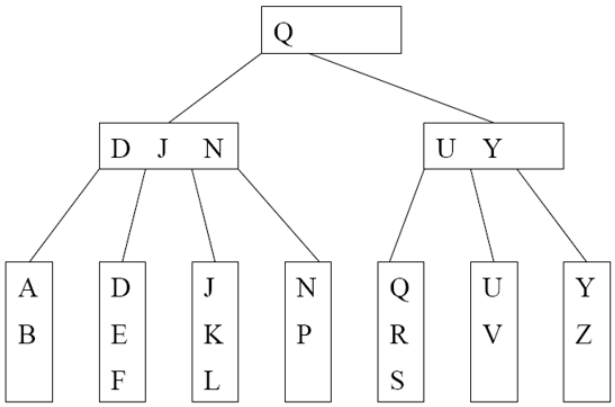
概念：

B+树是 B 树的一种变形形式，叶子结点以上各层作为索引使用，存储关键字以及相应记录的地址，叶子结点存储相应的记录。

一棵 M=m，L=l 的 B+树定义如下：

(1)每个结点至多有 m 个子女；

(2)除根结点外，每个结点至少有[m/2]个子女，根结点至少有两个子女，即除了根节点以外，每个结点至少有[m/2]-1 个关键字，根节点至少有一个关键字；
(3)有 k 个子女的结点必有 k-1 个关键字；
(4)所有叶子都在同一层；
(5)每个叶子结点至少存储[l/2]个记录，至多存储 l 个记录。
逻辑结构：
以一颗一颗 M=4，L=4 的 B+树为例



如图，以上为一颗 M=4，L=4 的 B+树，共有 3 个内部结点（包含一个根结点）和 7 个叶子结点。内部结点最多有 4 个孩子，最多存储 3 个关键字，因此包含 4 个指针（指向孩子结点）以及 3 块关键字大小的空间（存储关键字）；每个叶子结点是内存中的一个内存块，用于存储相应记录。

2、设计存储 B+树的数据结构设计
由要求 1 规定可知，（M-1）*4+M*4<=40、8*L+4<=40，得 M=5、L=5，即内部结点最多容纳 4 个键，叶子结点（外部存储块）最多存储 4 个记录。
为方便选取中间结点，记 B+树半满时的关键字和数据个数为 ORDER_V（即 ORDER_V=（M-1）/2），有如下关系：

```
#define ORDER_V 2    /* 为简单起见，把v固定为2，实际的B+树v值应该是可配的 */

#define MAXNUM_KEY (ORDER_V * 2)    /* 内部结点中最多键个数，为2v */
#define MAXNUM_POINTER (MAXNUM_KEY + 1)    /* 内部结点中最多指向子树的指针个数，为2v */
#define MAXNUM_DATA (ORDER_V * 2)    /* 叶子结点中最多数据个数，为2v */
```

2·1 结点设计
B+树的结点可分为叶子节点、根节点、内部结点，因此想到用继承和多态来实现他们之间的关系，设计 NODE_TYPE 类型用于表示结点类型。

结点类型

```
enum NODE_TYPE
{
    NODE_TYPE_ROOT     = 1,    // 根结点
```

```
    NODE_TYPE_INTERNAL = 2,    // 内部结点
    NODE_TYPE_LEAF     = 3,    // 叶子结点
};
```

根据内部节点和叶子结点的关系设想用派生类来实现，基类定义如下：
设置 m_Type 为结点类型，取值为 NODE_TYPE 类型；m_Count 为有效数据个数，对中间结点指键个数，对叶子结点指数据个数；m_pFather 作为指向父结点的指针，标准 B+树中并没有该指针，加上是为了更快地实现结点分裂和旋转等操作。

```
/* 结点数据结构，为内部结点和叶子结点的父类 */
class CNode{
public:
    CNode();
    virtual ~CNode();
    //获取和设置结点类型
    NODE_TYPE GetType() { return m_Type; }
    void SetType(NODE_TYPE type) {m_Type = type;}

    // 获取和设置有效数据个数
    int GetCount() { return m_Count;}
    void SetCount(int i) { m_Count = i; }

    // 获取和设置某个元素，对中间结点指键，对叶子结点指数据
    virtual KEY_TYPE GetElement(int i) {return 0;}
    virtual void SetElement(int i, KEY_TYPE value) { }

    // 获取和设置某个指针，对中间结点指指针，对叶子结点无意义
    virtual CNode* GetPointer(int i) {return NULL;}
    virtual void SetPointer(int i, CNode* pointer) { }

    // 获取和设置父结点
    CNode* GetFather() { return m_pFather;}
    void SetFather(CNode* father) { m_pFather = father; }

    // 获取一个最近的兄弟结点
    CNode* GetBrother(int& flag);

    // 删除结点
    void DeleteChildren();

protected:
    NODE_TYPE m_Type;    // 结点类型，取值为 NODE_TYPE 类型
    int m_Count;    // 有效数据个数，对中间结点指键个数，对叶子结点指数据个数
    CNode* m_pFather;     // 指向父结点的指针，标准 B+树中并没有该指针，加上是为了更快地实现结点分裂和旋转等操作
```

```
};
```

内部结点设计如下：

```
/* 内部结点数据结构 */
class CInternalNode : public CNode
{
public:
    CInternalNode();
    virtual ~CInternalNode();

    // 获取和设置键值，对用户来说，数字从 1 开始
    KEY_TYPE GetElement(int i);
    void SetElement(int i, KEY_TYPE key);
    // 获取和设置指针，对用户来说，数字从 1 开始
    CNode* GetPointer(int i);
    void SetPointer(int i, CNode* pointer);
    // 插入键
    bool Insert(KEY_TYPE value, CNode* pNode);
    // 删除键
    bool Delete(KEY_TYPE value);

    // 分裂结点
    KEY_TYPE Split(CInternalNode* pNode, KEY_TYPE key);
    // 结合结点
    bool Combine(CNode* pNode);
    // 从另一结点移一个元素到本结点
    bool MoveOneElement(CNode* pNode);

protected:
    KEY_TYPE m_Keys[MAXNUM_KEY];          // 键数组
    CNode* m_Pointers[MAXNUM_POINTER];     // 指针数组
};
```

叶子结点设计如下：

```
/* 叶子结点数据结构 */
class CLeafNode : public CNode
{
public:
    CLeafNode();
    virtual ~CLeafNode();

    // 获取和设置数据
    KEY_TYPE GetElement(int i);

    void SetElement(int i, KEY_TYPE data);
```

```
    // 获取和设置指针，对叶子结点无意义，只是实行父类的虚函数
    CNode* GetPointer(int i);

    // 插入数据
    bool Insert(KEY_TYPE value);
    // 删除数据
    bool Delete(KEY_TYPE value);

    // 分裂结点
    KEY_TYPE Split(CNode* pNode);
    // 结合结点
    bool Combine(CNode* pNode);
public:
    // 以下两个变量用于实现双向链表
    CLeafNode* m_pPrevNode;                // 前一个结点
    CLeafNode* m_pNextNode;                // 后一个结点
protected:
    KEY_TYPE m_Datas[MAXNUM_DATA];    // 数据数组
};
```

2·2 B+树结构设计

B+树应具有查找、插入、删除的基本功能，加入一个双向链表用于将外部存储块相连接，此外还添加一个 depth 用于记录 B+树的深度这一基本属性。

B+树类设计如下：

```
class BPlusTree{
public:

    BPlusTree();
    virtual ~BPlusTree();

    // 查找指定的数据
    bool Search(KEY_TYPE data, char* sPath);
    // 插入指定的数据
    bool Insert(KEY_TYPE data);
    // 删除指定的数据
    bool Delete(KEY_TYPE data);

    // 清除树
    void ClearTree();

    // 打印树
    void PrintTree();
```

```
    // 旋转树
    BPlusTree* RotateTree();

    // 检查树是否满足 B+树的定义
    bool CheckTree();
    void PrintNode(CNode* pNode);

    // 递归检查结点及其子树是否满足 B+树的定义
    bool CheckNode(CNode* pNode);

    // 获取和设置根结点
    CNode* GetRoot();
    void SetRoot(CNode* root);

    // 获取和设置深度
    int GetDepth();
    void SetDepth(int depth);

    // 深度加一
    void IncDepth();
    // 深度减一
    void DecDepth();

public:
    // 以下两个变量用于实现双向链表
    CLeafNode* m_pLeafHead;                // 头结点
    CLeafNode* m_pLeafTail;                  // 尾结点
protected:
    // 为插入而查找叶子结点
    CLeafNode* SearchLeafNode(KEY_TYPE data);
    //插入键到中间结点
    bool InsertInternalNode(CInternalNode* pNode, KEY_TYPE key, CNode* pRightSon);
    // 在中间结点中删除键
    bool DeleteInternalNode(CInternalNode* pNode, KEY_TYPE key);

    CNode* m_Root;    // 根结点
    int m_Depth;      // 树的深度
};
```

3、设计 B+树用于记录查找、插入和删除的算法，包含一个自行设计的 20ms 延时器

查找算法设计：

算法: B+树查找数据

输入: data，表示要查找的数据

输出: 一个 bool 值，表示该 data 是否在该 B+树中

```
CNode*pNode <-  the root of the current tree
//找要查找的叶子节点
  while pNode is not leafnode
      //找到第一个键值大于等于 key 的位置
      for pos<-1 to  number of effective node in pNode
      do
          if data > pNode->GetElement(pos)
              break
      end for
      pNode  <-  pNode->GetPointer(pos)
  end while

  if pNode=NULL then
      return false
  for pos<-1 to  pos<-1 to  number of effective node in pNode do
      if  pNode->GetElement(pos)=data then
          return true
  end for

return false
```

插入算法设计：

算法: B+树插入数据

输入: data，表示要插入的数据

输出: 一个 bool 值，表示该 data 是否成功插入该 B+树

```
    if search(data)=true then
        return false;//表示 data 已经在该 B+树中
// 查找理想的叶子结点
    pOldNode <- SearchLeafNode(data);
// 如果没有找到，说明整个树是空的，生成根结点
    if pOldNode=NULL then
        creat a new leafnode
        insert the data into the new leafnode and set it as root
    end if


    if the node is a leafnode then
        insert data into the node

        if the number of datas <=L then
            return true;
```

```
        end if
        else
            split the leafnode
            insert the (L/2+1)th data into fathernode
        end else
    end if

    if the node is an internalnode then
        insert data into the node
        if the number of datas <=M-1 then
            return true
        else
            split the node
            insert the (M/2)th data into fathernode
        end else
    end if
```

删除算法设计：

算法: B+树删除数据

输入: data，表示要删除的数据

输出: 一个 bool 值，表示该 data 是否成功删除

```
pOldNode <- the leafnode contains the data
    if pOldNode==NULL then //表示这个 data 不在该 B+树中
        return false;

    delete data in pOldNode

    if the number of values in pOldNode >=ceil(L/2) then
        modify the key in fathernode if necessary
        return true
    end if
    else
        if the number of values in the brothernode >ceil(L/2) then
            move the nearest data from brothernode to this node
            modify the keys in fathernode
        end if
        else
            combine this node with its brothernode
            combine fathernodes correspondingly
            recursion
        end else
    end else
```

```
return true
```

延时器的设计如下：

```
class Delay{
public:
    //静态数据成员 time，用于计算总时间
    static size_t time;
    //初始化总时间为 0
    static void initialize(){
        time = 0;
    }
    //延迟函数，每次调用延时 20ms，读取或写入时调用
    static void delay(){
        _sleep(20);
        time += 20;
    }
    //打印总花费时间
    static void print(){
        cout << "Time Delay: " << time << "ms" << endl;
    }
};
```

延时器调用的时机为每次在叶子结点进行读取和写入，如下所示：

插入时

```
bool CLeafNode::Insert(KEY_TYPE value){
    // 如果叶子结点已满，直接返回失败
    if (GetCount() >= MAXNUM_DATA)
        return false;
    // 找到要插入数据的位置
    int i;
    for (i = 0; (value > m_Datas[i]) && ( i < m_Count); i++){
        Delay::delay();
    }
    // 当前位置及其后面的数据依次后移，空出当前位置
    for (int j = m_Count; j > i; j--){
                        Delay::delay();//读取各一次延时
                        Delay::delay();
        m_Datas[j] = m_Datas[j - 1];
    }
    // 把数据存入当前位置
    m_Datas[i] = value;
    Delay::delay();//写入延时
    m_Count++;
    // 返回成功
    return true;
```

```
}
```

删除时：

```
bool CLeafNode::Delete(KEY_TYPE value)
{
    bool found = false;
           int i;
    for (i = 0; i < m_Count; i++){
    Delay::delay();//取出数据并比较
    if (value == m_Datas[i]){
         found = true;
         break;
        }
    }
    // 如果没有找到，返回失败
    if (false == found)
        return false;
    // 后面的数据依次前移
    int j;
    for (j = i; j < m_Count - 1; j++)
    {
        Delay::delay();//读取各一次延时
        Delay::delay();
        m_Datas[j] = m_Datas[j + 1];
    }
    m_Datas[j] = INVALID;
    m_Count--;
    // 返回成功
    return true;
}
```

在叶子结点对外接口中：

```
    // 获取和设置数据
    KEY_TYPE GetElement(int i)
    {
        if ((i > 0 ) && (i <= MAXNUM_DATA)){
            Delay::delay();
            return m_Datas[i - 1];
        }
        else
            return INVALID;
    }

    void SetElement(int i, KEY_TYPE data)
    {
        if ((i > 0 ) && (i <= MAXNUM_DATA)){
```

```
            Delay::delay();
            m_Datas[i - 1] = data;
        }
    }
}
```

从而实现延时效果，并且可打印累计时间。

4、列出源代码各个主模块命名清单（不需要代码清单）

main.cpp

main 函数包含了 7 个测试函数，可插入一定数目的随机数，能对 B+树进行插入、删除、查找以及清空树的功能，并且附加了打印树（按层打印）和检查树（检查 B+树是否符合定义）的功能。

BPlusTree.h

包含 5 个类的定义：Delay（延时器）、CNode（结点的基类）、CinternalNode（内部节点和根节点）、CleafNode（叶子结点）、BPlusTree（B+树）

BPlusTree.cpp

对 CNode（结点的基类）、CinternalNode（内部节点和根节点）、CleafNode（叶子结点）、BPlusTree（B+树）成员函数的实现。

5、测试用例设计（初始化至少包含 50 个记录数据）

为初始化向时 B+树插入一定数目的随机数，设计插入随机数函数：

Input: A pointer of B+ Tree pTree, a number symbolize the amount of random numbers need to be insert

```
void Test1(BPlusTree* pTree, int count)
{
    Delay::initialize();
    int i = 0;
    while (i < count)
    {
        //srand( (unsigned)time( NULL ) );//这是一个种子，如果不要随机功能，请把此句话注释掉
        int x = rand() % 999 + 1;
        if (pTree->Insert(x))
            ++i;
    }
    cout << "successed!" << endl;
    if(Delay::time)
        Delay::print();
}
```

通过 C 库函数 rand 和 srand 获得随机数 1~1000，向 B+树中插入，该方法获得的随机数可能存在重复的情况，因此只有在插入成功时，即 Insert 函数返回值为 true 的时候插入数目的统计值增加，否则继续获取随机数。当插入数目统计值和预定数目相等时结束插入。

运行该程序，并插入 50 个随机数：

```
    ********************************************************************
    * There is a B+ Tree which is empty, what do you want to do?       *
    *                1:Insert some random numbers                      *
    *                2:Search a number                                 *
    *                3:Insert a number                                 *
    *                4:Delete a number                                 *
    *                5:Print the tree                                  *
    *                6:Check the tree                                  *
    *                7:Clear the tree                                  *
    *                0:Exit                                            *
    ********************************************************************
Your choice:
1
How many random numbers do you want to insert?
50
successed!
Time Delay: 9480ms
```

显示插入成功后打印结果：

```
FLOOR 1: |  388 527 735 844 |

FLOOR 2: |  154 189 305 341 |  424 486   0   0 |  619 711   0   0 |  740 786   0   0 |  904 915 971   0 |

FLOOR 3: |   42  53   0   0 |  154 157 174   0 |  189 282 293   0 |  305 326   0   0 |  341 354 364   0 |
         |  388 395   0   0 |  424 439 442 453 |  486 489 490 492 |
         |  527 550   0   0 |  619 669 691 693 |  711 719 732   0 |
         |  735 738   0   0 |  740 748   0   0 |  786 832 835   0 |
         |  844 871 875   0 |  904 906   0   0 |  915 932 954   0 |  971 989 998   0 |
```

如图，FLOOR1 打印的是 B+树第一层结点的值，即根节点的值；FLOOR2 打印的是 B+树第二层结点的值，分别代表根节点的 5 个孩子；FLOOR3 打印的是 B+树第三层结点的值，每一行分别代表第二层结点的孩子，即第 i 行对应第二层第 i 个结点的孩子。
经分析可知，该树符合 B+树的定义。

6、运行结果分析，包括外部存取延时统计
查找功能：
i． 查找 998（998 存在 B+树中）

```
Please enter the number you want to search:
998
The serach path is: 388 --> 904 -->971 ,successed.
Time Delay: 80ms
```

查找路径从根节点出发，根节点首元素为 388，由于 998>844，因此沿着 844 的右孩子继续向下查找到首元素为 904 的结点，由于 998>971，因此验证 971 的右孩子继续向下找，直到在叶子结点[971 989 998 0]中找到 998，查找成功。

ii． 查找 104（不存在 B+树中）

```
Please enter the number you want to search:
104
The serach path is: 388 --> 154 -->42 ,failed.
Time Delay: 60ms
```

查找路径从根节点出发，根节点首元素为 388，由于 104<388，因此沿着 388 的左孩子继续向下查找到首元素为 154 的结点，由于 104<154，因此验证 154 的左孩子继续向下找，直到叶子结点[42 53 0 0]，由于 104 不存在[42 53 0 0]中，因此查找失败。

删除功能：

iii. 删除 711（711 索引在 B+树中的内部节点上）

删除后结果：

```
FLOOR 1: |  388 527 735 844 |

FLOOR 2: |  154 189 305 341 |  424 486   0   0 |  619 719   0   0 |  740 786   0   0 |  904 915 971   0 |

FLOOR 3: |   42  53   0   0 |  154 157 174   0 |  189 282 293   0 |  305 326   0   0 |  341 354 364   0 |
         |  388 395   0   0 |  424 439 442 453 |  486 489 490 492 |
         |  527 550   0   0 |  619 669 691 693 |  719 732   0   0 |
         |  735 738   0   0 |  740 748   0   0 |  786 832 835   0 |
         |  844 871 875   0 |  904 906   0   0 |  915 932 954   0 |  971 989 998   0 |
```

在叶子结点[711 719 732 0]中找到 711，将 711 在叶子节点删除，同时在内部节点中的索引 711 变为它的后继 719，删除成功。

iv. 删除 693（在叶子结点上）

删除后：

```
FLOOR 1: |  388 527 735 844 |

FLOOR 2: |  154 189 305 341 |  424 486   0   0 |  619 719   0   0 |  740 786   0   0 |  904 915 971   0 |

FLOOR 3: |   42  53   0   0 |  154 157 174   0 |  189 282 293   0 |  305 326   0   0 |  341 354 364   0 |
         |  388 395   0   0 |  424 439 442 453 |  486 489 490 492 |
         |  527 550   0   0 |  619 669 691   0 |  719 732   0   0 |
         |  735 738   0   0 |  740 748   0   0 |  786 832 835   0 |
         |  844 871 875   0 |  904 906   0   0 |  915 932 954   0 |  971 989 998   0 |
```

在叶子结点[619 669 691 693]中找到 693 并删除，删除成功。

插入功能

v. 插入 1000（在叶子结点上）

插入后：

```
FLOOR 1: |  388 527 735 844 |

FLOOR 2: |  154 189 305 341 |  424 486   0   0 |  619 719   0   0 |  740 786   0   0 |  904 915 971   0 |

FLOOR 3: |   42  53   0   0 |  154 157 174   0 |  189 282 293   0 |  305 326   0   0 |  341 354 364   0 |
         |  388 395   0   0 |  424 439 442 453 |  486 489 490 492 |
         |  527 550   0   0 |  619 669 691   0 |  719 732   0   0 |
         |  735 738   0   0 |  740 748   0   0 |  786 832 835   0 |
         |  844 871 875   0 |  904 906   0   0 |  915 932 954   0 |  971 989 9981000 |
```

在叶子结点[971 989 998 0]中找到空位插入 1000，删除成功。

插入 1000 时延统计：

```
Please enter the number you want to insert:
1000
successed!
Time Delay: 140ms
```

经上述分析，实验结果符合预期。

五、实验心得

通过这次实验我更加深入的了解和掌握了B+树的结构和插入、删除和查找的方法同时也实现了插入随机数、打印B+树和检查树的功能。通过这次实验，一方面我们对B+树的定义、实现等有了更加深入的认识，也更加熟悉了B+树的插入、查找、删除等算法，通过实现一个B+树的仿真我们从更底层出发理解了B+树的特点。

当然，在实验过程中也遇到一些困难，一开始我们难以确定内部节点和叶子节点在实现上的关系，最终通过查阅资料、相互讨论，确定了它们都由一个基类派生而来的方法。并且在网上查找资料如何实现时间延迟，最终根据延时函数设计延时类，设计了通

过利用静态成员和静态成员函数的方法实现全局的延时时间统计。

总而言之，此时实验使我们收益匪浅。

参考文献

《数据结构与算法实验实践教程》，乔海燕、蒋爱军、高集荣和刘晓铭编著，清华大学出版社出版，2012