B+ 树的模拟

数据科学与计算机学院

18340206 张德龙

1356672774@qq.com

摘要

笔者和周思宇同学进行合作，共同完成了此次 B+ 树的仿真器程序的设计和测试。笔者负责的工作有：插入、查找、打印 B+ 树和设计模拟外部存取时间延迟的延时器的工作，因此本文对 B+ 树上述部分的设计过程进行了分析。

关键字： B+ 树　插入　查找　打印 B+ 树　模拟外部存取

目录

一、 引言

1.1 B+ 树的概念

结合教材中对 B 树的定义，阶为 M 的 B 树是一颗具有下列特性的 M 叉树：

1. 数据项存储在树叶上，不允许出现重复的数据。
2. 非叶结点存储直到 $M-1$ 个关键字以指示搜索的方向；关键字 i 代表子树 $i+1$ 中的最小的关键字。
3. 树的根不能是一片树叶，其儿子数在 2 和 M 之间。
4. 除根外，所有非叶节点的儿子数在 $\lceil M/2 \rceil$ 和 M 之间。
5. 所有的树叶都在相同的深度上，并且每片树叶拥有的数据项的个数在 $\lceil L/2 \rceil$ 和 M 之间。

1.2 B+ 树的逻辑结构

由 B+ 树的概念，我们使用完全 M 叉查找树作为 B+ 树的逻辑结构。

二、 解决方法

2.1 数据结构设计

根据 B+ 树的概念和逻辑结构，可以设计数据结构如下：

```
template<typename data_type, typename key_type, typename getKey, int order = 5, int L = 4>
class B_Tree
{
public:
   B_Tree();
private:
   struct Leaf
   {
      Leaf();
      int count;
      data_type data[L];
   };
   enum NodeTag:char { LEAF, NODE }; //大小为char 判断是否连接叶子结点
   struct Node
   {
      char count;
      NodeTag tag; //用于判断是否连接叶子结点
      key_type key[order-1]; //关键字
      union Branch
      {
```

```
        Node* node[order];
        Leaf* leaf[order];
      };
      Branch branch; //储存分支的指针
      Node();
   };
private:
   Node* root;
   mutable int accessTime; //记录访存次数
public:
   ......
```

设置阶数 $M=5$，$L=4$，所以在考虑字节对齐的情况下

$$sizeof(Node)=1+1+(2)+4*4+20=40$$

$$sizeof(Leaf)=4+8*4=36$$

其中 () 括起来的数字表示因字节对齐而填充的字节。由计算可得，在 $M=5$、$L=4$ 的情况下，设计的 B+ 树已最大限度地利用了题设所给的外部存储块的空间。

2.2 代码主模块命名清单

```
/*
前提：传入的getKey为一个仿函数，要求传入一个data_type能够返回它的key_type的键值
前提：认为B+树的根永远不会退化至叶
*/
template<typename data_type, typename key_type, typename getKey, int order = 5, int L = 4>
class B_Tree
{
public:
   B_Tree(); //默认初始化 固定的初始化值
   B_Tree(data_type data[(L/2)*2]); //采用传入的参数初始化B+树
   ~B_Tree();
public:
   /*
   功能：利用输入的(L/2)*2个数据初始化形成一颗最小的B+树
   前提：key_type重载了>或者是能够直接比较
   结果：形成由一个根和两片叶子组成的B+树
   */
   void init(data_type data [(L/2)*2]);
   private:
   //测试用初始化函数
   void init();
```

```
public:
    /*
    功能：查找B树中是否有含键值为x的数据
    结果：若包含则返回true 否则返回false
    */
    bool contains( const key_type& x ) const;
    /*
    功能：向B树中插入data x
    结果：若成功则返回true，重复返回false
    */
    bool insert( const data_type& x );
    /*
    功能：在B树中删除data x
    结果：若成功则返回true，若树中无x则返回false
    */
    bool erase( const data_type& x );
    /*
    功能：打印B树和总共对磁盘的访问次数
    */
    void display() const;
    /*
    功能：释放B+树
    */
    void clear();

private:
    enum State { success, overflow, duplicate }; //insert函数可能返回的状态
    /*
    功能：对内结点插入键值为key的数据data，若这个操作使得结点分裂
    产生的新键放入newKey中，产生的新分支放入newBranch中
    结果：若插入成功则返回true 若重复则返回false
    */
    State insert( Node*& n, const key_type& key, const data_type& data, key_type& newKey,
        Node*& newBranch);
    /*
    功能：对叶子插入键值为key的数据data，若这个操作使得叶子分裂
    产生的新键放入newKey中，产生的新分支放入newBranch中
    结果：若插入成功则返回true 若重复则返回false
    */
    State insert( Leaf*& l, const key_type& key, const data_type& data, key_type& newKey,
        Leaf*& newBranch);
    /*
    功能：向内结点n插入关键字key
    前提：n未满
    */
    void insert_key( Node* n, const key_type& newkey, void* newBranch, size_t pos );
    /*
```

```
    功能：向叶子l插入数据data
    前提：l未满
    */
    void insert_data( Leaf* l, const data_type& newdata, size_t pos );
    /*
    功能：在内结点n中的关键字中寻找关键字x应当出现的位置
    前提：x重载了<运算符
    */
    size_t findPos( Node* n, const key_type& x ) const;
    /*
    功能：在叶子l中的数据中寻找数据d应当出现的位置
    前提：x重载了<运算符
    */
    size_t findPos( Leaf* l, const data_type& d ) const;
    /*
    功能：通过插入cur_newKey和cur_newBranch使得结点n分裂 产生的新键置入newKey中
        产生的新分支置入newBranch中
    前提：n已满
    */
    void split( Node* n, const key_type& cur_newKey, Node* cur_newBranch, size_t pos,
        key_type& newKey, Node*& newBranch);
    /*
    功能：通过插入data使得结点n分裂
    产生的新键置入newKey中
    产生的新分支置入newBranch中
    前提：l已满
    */
    void split( Leaf* l, const data_type& data, size_t pos, key_type& newKey, Leaf*&
        newBranch );
    /*
    功能：查找n和其包含的分支中是否含有键为x的data
    前提：x重载了==运算符
    结果：若包含则返回true 否则返回false
    */
    bool contains( Node* n, const key_type& x ) const;
    /*
    功能：查找叶子l是否含有键为x的data
    前提：x重载了==运算符
    结果：若包含则返回true 否则返回false
    */
    bool contains( Leaf* l, const key_type& x ) const;
    /*
    功能：打印结点n及其所包含分支
    */
    void display( Node* n, int indent = 0 ) const;
    /*
    功能：打印叶子l中的数据
```

```
    */
    void display( Leaf* l, int indent = 0 ) const;

    /*
    功能：从树中删除叶子中的数据y
    结果：若删除成功则返回true,删除失败返回false
    */
    bool erase(Node*& n, const data_type& y);
    /*
    功能：给定叶节点指针current,删除该叶节点内的数据x
    结果：若删除成功则返回true,删除失败返回false
    */
    bool remove_inleaf(Leaf* current, const data_type& x);
    /*
    功能：删除单个数据后在叶节点和内部节点之间依据B+树规则对树进行重新调整
    前提：有叶子的数据数目小于L/2
    */
    void restore_inleaf(Node* current, const int& position);
    /*
    功能：删除单个数据在内部节点和内部节点之间依据B+树规则对树进行重新调整
    前提：有内部节点关键词数目小于(order-1)/2
    */
    void restore_innode(Node* current, const int& position);
    /*
    功能：父亲的一个关键字下拉，右兄弟一个关键字上提，实现左旋转
    前提：右兄弟关键词数据大于(order-1)/2
    */
    void movenode_left(Node* current, const int& position);
    /*
    功能：父亲的一个关键字下拉，左兄弟一个关键字上提，实现右旋转
    前提：左兄弟关键词数据大于(order-1)/2
    */
    void movenode_right(Node* current, const int& position);
    /*
    功能：合并父亲节点和有current和keyposition确定的两个子节点
    前提：左右兄弟关键词数目都小于(order-1)/2
    */
    void movenode_combine(Node* current, const int& keyposition);
    /*
    功能：由current和position确定的叶节点从其右兄弟叶子借得一个数据，并调整关键字
    前提：右兄弟的数据数目大于L/2
    */
    void moveleaf_left(Node* current, const int& position);
    /*
    功能：由current和position确定的叶节点从其左兄弟叶子借得一个数据，并调整关键字
    前提：左兄弟的数据数目大于L/2
    */
```

```
    void moveleaf_right(Node * current, const int& position);
    /*
    功能：删除一个关键字，合并其原来的两个叶子节点
    前提：左右叶子数据数目都小于L/2
    */
    void moveleaf_combine(Node* current, const int& keyposition);
    /*
    功能：采用后序遍历释放结点n及其分支
    */
    void clear( Node*& n );
    /*
    功能：采用后序遍历释放叶子l
    */
    inline void clear( Leaf*& l );
public:
    inline void displayAccessTime() const //打印访存次数;
```

2.3 查找算法

类比二叉查找树的思想，使用 $findpos$ 函数找到数据应当在 B+ 树中出现的位置并不断向下深入。寻找到叶子时，遍历叶子即可知道是否含有此数据。

```
template<typename data_type, typename key_type, typename getKey, int order, int L>
size_t B_Tree<data_type, key_type, getKey, order, L>::findPos( Node* n, const key_type& x )
    const
{
    for( int i = 0; i < n->count; ++i )
    {
        if( x < n->key[i] )
            return i;
    }
    return n->count;
}

template<typename data_type, typename key_type, typename getKey, int order, int L>
bool B_Tree<data_type, key_type, getKey, order, L>::contains( Node* n, const key_type& x )
    const
{
    increaseAccessTime(); //访问磁盘
    size_t pos = findPos( n, x );
    if( n->tag == NODE )
        return contains( n->branch.node[pos], x );
    else if( n->tag == LEAF )
        return contains( n->branch.leaf[pos], x );
}
```

```
template<typename data_type, typename key_type, typename getKey, int order, int L>
bool B_Tree<data_type, key_type, getKey, order, L>::contains( Leaf* l, const key_type& x )
    const
{
   increaseAccessTime(); //访问磁盘
   getKey get;
   for( int i = 0; i < l->count; ++i )
      if( get(l->data[i]) == x )
         return true;
   return false;
}
```

2.4 插入算法

2.4.1 插入主函数

因为插入的过程中会出现三种情况：溢出（插入的结点分裂）、数据重复和正常插入。因此，使用一个枚举类型来表示三种情况，也使用此枚举类型作为主函数的辅助函数的返回值。

主函数返回值为 *bool*，用来表示是否插入成功（若数据重复则表示插入不成功）。$newBranch$ 和 $newKey$ 用来接收溢出情况下生成的新键和新结点。若在根节点插入溢出，则表示应当生成一个新根，此时辅助函数 *insert* 已经将根节点分裂为 *root* 和 $newBranch$ 两个满足 B+ 树定义的分支，以新键 $newKey$ 生成新根，其中新根的左分支为 *root*，右分支为 $newBranch$。

```
template<typename data_type, typename key_type, typename getKey, int order, int L>
bool B_Tree<data_type, key_type, getKey, order, L>::insert( const data_type& data )
{
   getKey get;
   key_type key = get( data ); //获取data的关键字
   key_type newKey;
   Node* newBranch;
   State result = insert( root, key, data, newKey, newBranch );
   if( result == overflow ) //以newKey和newBranch创建新根
   {
      Node* newRoot = new Node; increaseAccessTime(); //访问磁盘
      newRoot->count = 1;
      newRoot->key[0] = newKey;
      newRoot->branch.node[0] = root;
      newRoot->branch.node[1] = newBranch;
      root = newRoot;
      result = success;
   }
   if( result == duplicate )
```

```
    return false;
  else
    return true;
}
```

2.4.2 递归插入

与查找算法的思想类似，使用 $findpos$ 函数找到数据应当出现的分支并插入，若插入的分支是树叶，则将数据插入。若数据已在叶子中，则返回 $duplicate$，若成功插入，则返回 $success$。若插入操作使得叶子分裂，则调用 $split$ 函数将叶子的数据按个数平分为两部分——原叶子和 $newBranch$，并将生成的新键 $newKey$ 和 $newBranch$ 一同传入上一函数中，返回状态 $overflow$。若调用此函数的上一层函数接收到 $overflow$ 信号，则将此函数生成的新关键字和分支插入上层函数对应的结点，若此操作造成此结点溢出，则继续调用 $split$ 函数进行分裂此结点并返回 $overflow$ 给上级的调用函数，重复此操作直到返回主函数为止。

```
union ptr
{
  Node* node;
  Leaf* leaf;
}; //一个可以是叶子又可以是内点的union

template<typename data_type, typename key_type, typename getKey, int order, int L>
typename B_Tree<data_type, key_type, getKey, order, L>::State
B_Tree<data_type, key_type, getKey, order, L>::insert( Node*& n, const key_type& key, const
    data_type& data, key_type& newKey, Node*& newBranch)
{
  increaseAccessTime(); //访问磁盘
  size_t pos = findPos( n, key );
  ptr next_node { n->branch.node[pos] };

  if( n->tag == LEAF )
  {
    for( int i = 0; i < next_node.leaf->count; ++i )
    if( next_node.leaf->data[i] == data )
    return duplicate;
  }

  key_type cur_newKey;
  ptr cur_newBranch;
  State result;
  if( n->tag == NODE )
    result = insert( next_node.node, key, data, cur_newKey, cur_newBranch.node );
  else if( n->tag == LEAF )
```

```
      result = insert( next_node.leaf, key, data, cur_newKey, cur_newBranch.leaf );

   if( result == overflow )
   {
      increaseAccessTime(); //访问磁盘 访问此节点
      if( n->count < order - 1 )
      {
         result = success;
         insert_key( n, cur_newKey, cur_newBranch.node, pos );
      }
      else
         split( n, cur_newKey, cur_newBranch.node, pos, newKey, newBranch );
      //通过下一级操作因overflow所产生的新键和新分支来更新newKey和newBranch给上一级使用
   }
   return result;
}

template<typename data_type, typename key_type, typename getKey, int order, int L>
typename B_Tree<data_type, key_type, getKey, order, L>::State
B_Tree<data_type, key_type, getKey, order, L>::insert( Leaf*& l, const key_type& key, const
    data_type& data, key_type& newKey, Leaf*& newBranch )
{
   increaseAccessTime(); //访问磁盘
   State result = success;
   size_t pos = findPos( l, data );
   if( l->count < L )
      insert_data( l, data, pos );
   else
   {
      split( l, data, pos, newKey, newBranch ); //通过向叶子中插入data 产生新的key和新的branch
      result = overflow;
   }
   return result;
}

template<typename data_type, typename key_type, typename getKey, int order, int L>
void B_Tree<data_type, key_type, getKey, order, L>::insert_data( Leaf* l, const data_type&
    newdata, size_t pos )
{
   increaseAccessTime(); //写入磁盘
   for( int i = l->count; i > pos; --i ) //包括pos在内都向前移位
   l->data[i] = l->data[i-1];
   l->data[pos] = newdata;
   l->count++;
}

template<typename data_type, typename key_type, typename getKey, int order, int L>
```

```
void B_Tree<data_type, key_type, getKey, order, L>::insert_key( Node* n, const key_type&
    newkey, void* newBranch, size_t pos )
{
   increaseAccessTime(); //写入磁盘
   for( int i = n->count; i > pos; --i ) //包括pos在内都向前移位
   {
   n->key[i] = n->key[i-1];
   n->branch.node[i+1] = n->branch.node[i];
   }
   n->key[pos] = newkey;
   n->branch.node[pos+1] = (Node*)newBranch;
   //branch的pos部分已经更新不需更改，只需插入新生成的分支即可
   n->count++;
}
```

2.4.3 split 函数

split 函数可以说是算法的核心，通过传入的分支和键 $curnewKey$ 和 $curnewBranch$ 将结点分裂，并产生新的键和分支 $newKey$ 和 $newBranch$。

首先，根据传入的参数 pos 来判断分支应处的位置，若 $pos >= mid$ 则让 mid 自增，以保证原结点剩余的关键字数大于等于分裂出去的结点，从而可以将分裂后的原节点的最后一个关键字作为新键。之后将包括 mid 在内的关键字和对应的右分支拷贝至新结点，将 $curnewKey$ 和 $curnewBranch$ 插入至应对应的结点（原结点或新结点）。以更新后的原节点的最后一个关键字更新 $newBranch$，将这个关键字对应的右分支置入新结点第一个元素的左分支，即完成了对结点的分裂，函数运行过程类似下图：

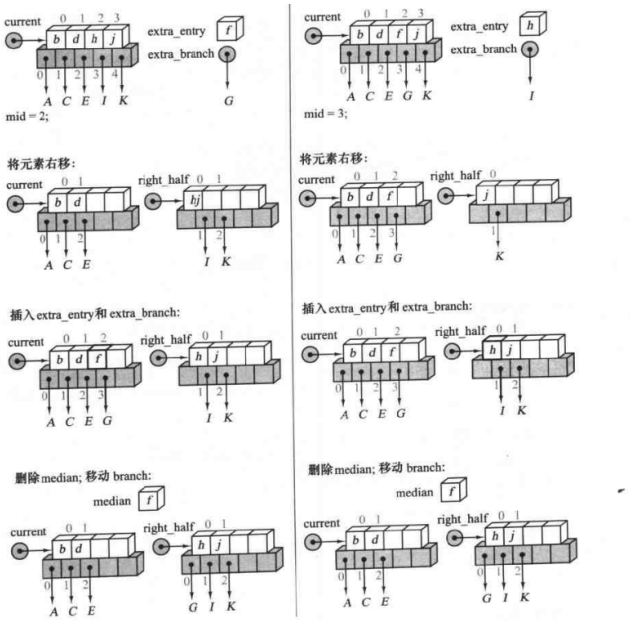

图 1　split 示意图

```
template<typename data_type, typename key_type, typename getKey, int order, int L>
void B_Tree<data_type, key_type, getKey, order, L>::split( Node* n, const key_type&
    cur_newKey, Node* cur_newBranch, size_t pos, key_type& newKey, Node*& newBranch)
{
   increaseAccessTime(); //访问磁盘
   newBranch = new Node( n->tag ); //分出的新结点的tag肯定和n相同
   size_t mid = order / 2;
   if( pos >= mid ) //使mid向后数的关键字个数总小于等于左半边的
   mid++;

   for( int i = mid; i < order-1; ++i ) //拷贝key和branch
   {
      //注意!!下标为order-2为第order-1个元素
      newBranch->key[i-mid] = n->key[i];
      newBranch->branch.node[i-mid+1] = n->branch.node[i+1];
   }
   n->count = mid;
   newBranch->count = order - 1 - mid;

   if( pos >= mid ) //与mid自不自增无关 位置都是pos-mid 因为拷贝都把mid拷贝过去了
      insert_key( newBranch, cur_newKey, cur_newBranch, pos - mid );
   else
      insert_key( n, cur_newKey, cur_newBranch, pos );

   //注意 插入完才能决定右半部分的branch【0】是什么
   newKey = n->key[n->count - 1];
   newBranch->branch.node[0] = n->branch.node[n->count];
   n->count--;
   //共计两次对磁盘写入
}

template<typename data_type, typename key_type, typename getKey, int order, int L>
void B_Tree<data_type, key_type, getKey, order, L>::split( Leaf* l, const data_type& data,
    size_t pos, key_type& newKey, Leaf*& newBranch )
{
   increaseAccessTime(); //访问磁盘
   newBranch = new Leaf;
   size_t mid = L / 2;
   if( pos >= mid ) //使mid向后数的关键字个数总小于等于左半边的
      mid++; //等号不可以去掉！！

   for( int i = mid; i < L; ++i ) //拷贝data和branch
   newBranch->data[i-mid] = l->data[i];

   l->count = mid;
   newBranch->count = L - mid;
```

```
    if( pos >= mid ) //与mid自不自增无关 位置都是pos-mid 因为拷贝都把mid拷贝过去了
        insert_data( newBranch, data, pos - mid );
    else
        insert_data( l, data, pos );

    getKey get;
    newKey = get(newBranch->data[0]); //新结点第一个data的关键字作为新键
}
```

至此，插入算法结束。

2.5 打印 B+ 树

采用类似中序遍历的思想，遍历 B+ 树。对于一个结点的关键字数组，从数组末尾元素开始向数组头元素遍历，对关键字进行遍历时，先遍历关键字所对应的右分支，再将关键字输出。最后再遍历分支数组的第一个元素。若遍历的分支是叶节点，则将叶节点所有的数据输出。在打印函数中传递一个 *indent* 参数记录深度，根据深度来打印缩进即可达到打印 B+ 树的效果。

```
void display() const
{
    display(root);
    displayAccessTime();
}

template<typename data_type, typename key_type, typename getKey, int order, int L>
void B_Tree<data_type, key_type, getKey, order, L>::display( Node* n, int indent ) const
{
    if( n->tag == NODE )
    {
        for( int i = n->count - 1; i >= 0; --i )
        {
            display( n->branch.node[i+1], indent + 1 );
            displayIndent(indent);
            cout << n->key[i] << endl;
        }
        display( n->branch.node[0], indent + 1 );
    }
    else if( n->tag == LEAF )
    {
        for( int i = n->count - 1; i >= 0; --i )
        {
            display( n->branch.leaf[i+1], indent + 1 );
            displayIndent(indent);
```

```
         cout << n->key[i] << endl;
      }
      display( n->branch.leaf[0], indent + 1 );
   }
}

template<typename data_type, typename key_type, typename getKey, int order, int L>
void B_Tree<data_type, key_type, getKey, order, L>::display( Leaf* l, int indent ) const
{
   displayIndent(indent);

   for( int i = 0; i < l->count; ++i )
      cout << l->data[i] << ' ';

   cout << endl;
}
```

打印效果图如下，将头部左旋 180° 效果更佳：

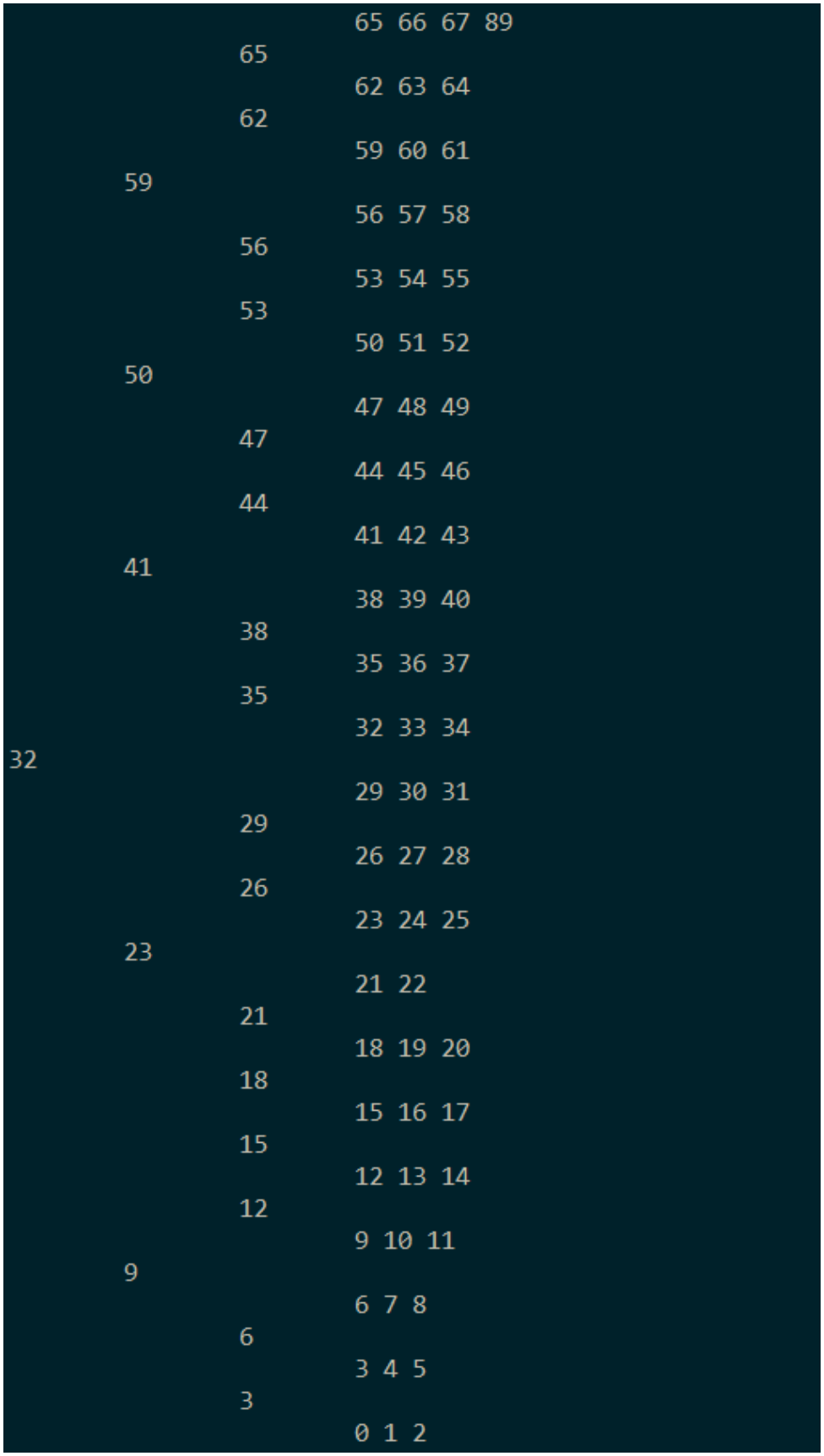

图 **2**　打印 **B+** 树

三、 模拟访存

3.1 延时器的设置

设置了如下结构来模拟访存，通过调用函数 $increaseAccessTime$ 来代表访问一次磁盘，$accessTime$ 变量设置为 $mutable$，是因为在诸如 $contains$ 等 $const$ 函数也需要对其值进行改变。

```
private:
  mutable int accessTime; //存储访存次数
  inline void increaseAccessTime() const
  {
    accessTime++;
    #ifdef DELAY
    Sleep(20);
    #endif
  }
```

3.2 何时进行访存

观察如下代码：

```
  Node* n;
  n->count++;
```

当我们需要对 n 进行解引用读取其值时，视为一次访存——对磁盘的读取。当控制未从此函数转移时，对 n 指向的数据进行多次读操作都仅视为一次访存——因为此时已经将外部存储块读入内存，不需再次访存。若需要对 n 指向的数据进行写入，则需要进行一次访存——磁盘写入与是否对磁盘进行读取无关；若控制权从其他函数再次返回此函数，再次对 n 指向的数据进行读或写也需要再次访存；认为 $split$ 函数不需要对传入的 $Node^*/Leaf^*$ 参数进行读取类型的访存，因为上一级函数已经将其读入内存，仅仅经过一级函数调用，认为内存可将访问的数据保留；在 $split$ 函数中总共对新分支和原分支进行了两次磁盘写入。

四、 程序使用和测试说明

程序使用说明可见附件 $README.md$。

编写了如下结构体进行测试，为了便于观察，将 $Record$ 结构体的键值和数据的值置为相同。$getKey$ 函数用于从数据获取关键字。

```
struct Record
{
   int data;
   int key;
   Record( int a, int b ) : data(a), key(b) {} //第一个参数为数据，第二个参数为键值
   Record( int a ) : data(a), key(a) {} //键值=数据
   Record() {}
   Record( const Record& o )
   {
      data = o.data;
      key = o.key;
   }
   bool operator==( const Record& o) const
   {
      return key == o.key;
   }
   bool operator< ( const Record& o) const
   {
      return key < o.key;
   }
};

ostream& operator <<( ostream& out, const Record& r )
{
   out << r.data;
   return out;
}

struct getKey
{
   int operator() (const Record& r) const
   {
      return r.key;
   }
};
```

思宇同学编写了初始化函数 *init*，使得能够根据数组中的存储的数据来对 B+ 树进行初始化。详情可参照他编写的报告或代码中 *init* 函数部分。

采用数据 12、23、50、89 对 B+ 树进行初始化[1]。

[1]虽然问题要求初始化用例至少包含 50 个数据，但由之前介绍的 B+ 树结构和方法可知，初始化 50 个数据和初始化 4 个数据并无区别——如果必须要认为初始化 50 个数据才可开始测试，可以认为测试在插入 46 个数据后测试才真正开始。

Test 1：

对初始化的树插入键值为 0 – 99 的数据，此项测试即包含了对 B+ 树各个部分进行插入操作的检测，部分测试截图如下：

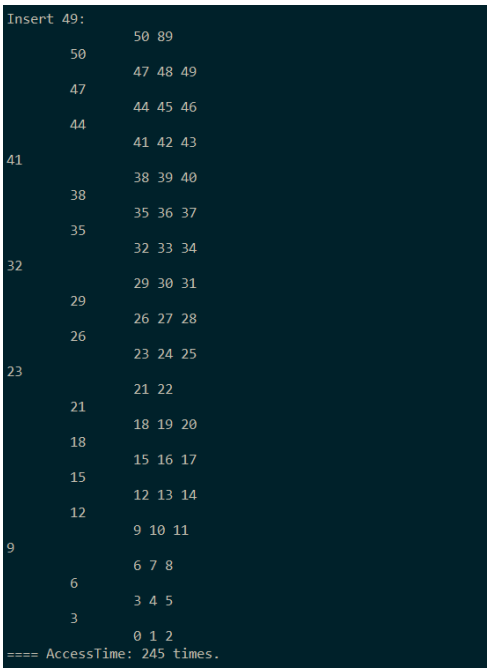

图 **3**　插入数据 **49**

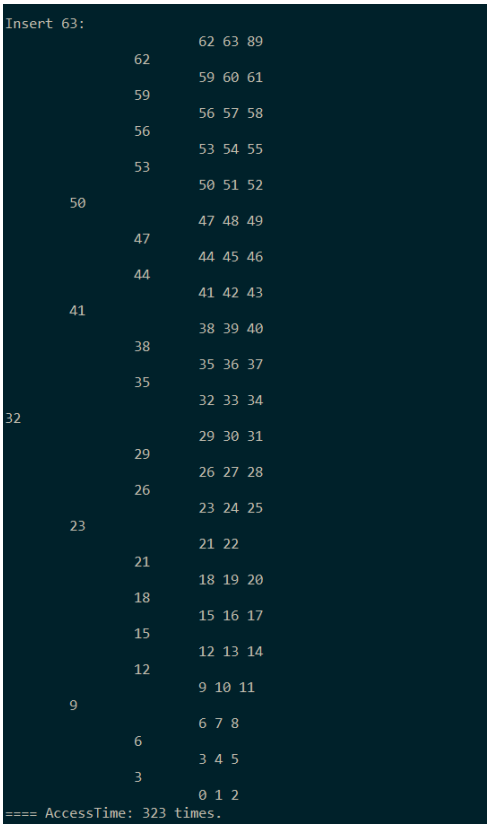

图 **4**　插入数据 **63**

之后再删除 10 – 59 的数据：

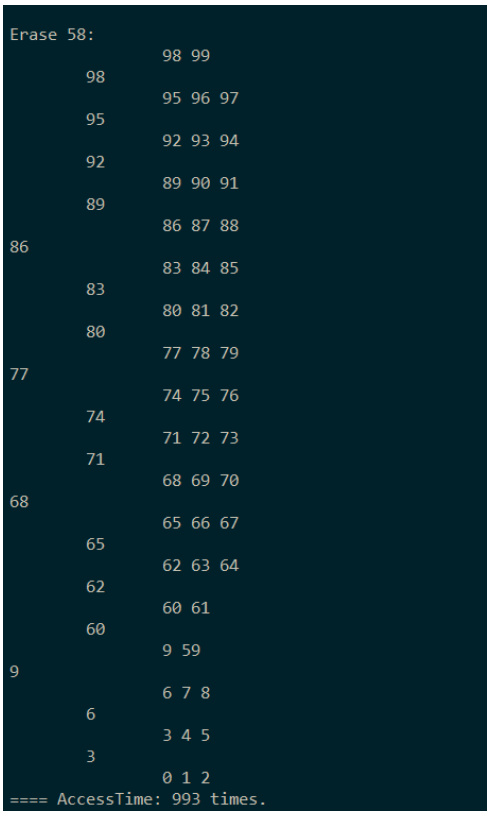

图 **5**　删除数据 **58**

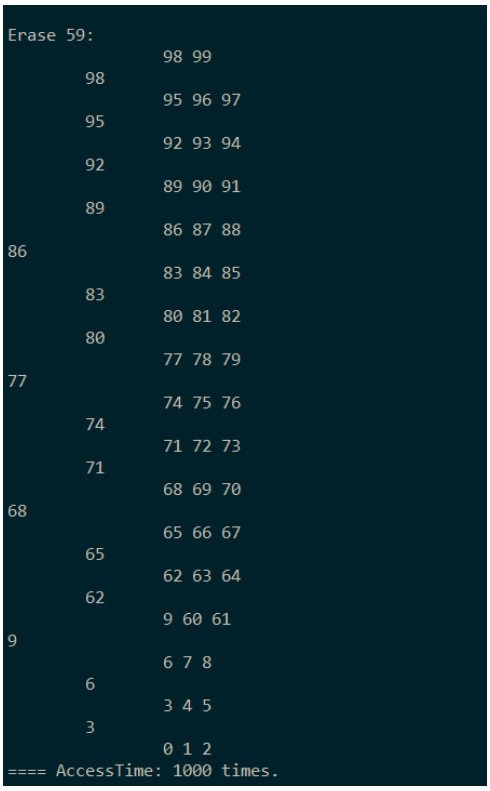

图 **6**　删除数据 **59**

逐个检查 B+ 树是否包含 0 – 150 的上数据，若出现不符合逻辑的情况输出 *Obs*。

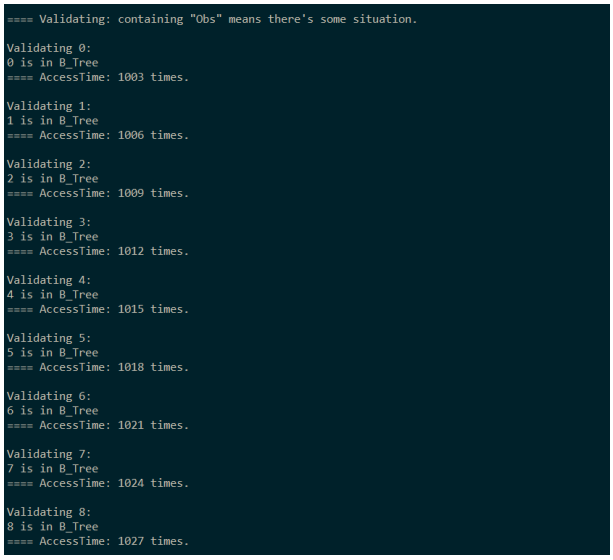

图 **7**　检查 **B+** 树是否含有某个数据

测试结果：未输出 *Obs*。借由延时来观察插入和删除的操作对 B+ 树形状的改变，并未观察到破坏 B+ 树结构的操作。

Test 2：
对插入操作的访存次数测试：

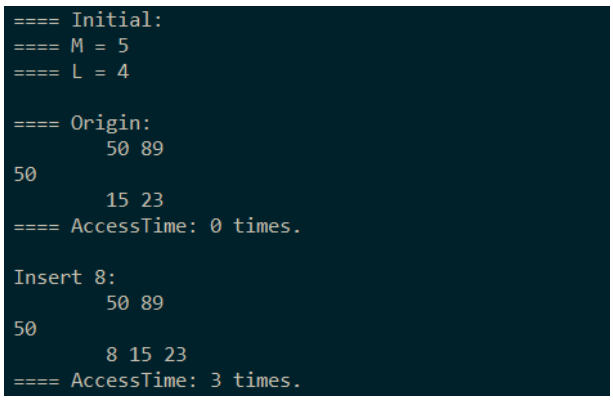

图 **8**　测试插入操作的访存次数

测试结果：两次读入，一次写入，和理论分析的访存次数一致。

Test 3：

对查找操作的访存次数测试：

```
==== Initial:
==== M = 5
==== L = 4

==== Origin:
        50 89
50
        15 23
==== AccessTime: 0 times.

Validating 23:
        50 89
50
        15 23
==== AccessTime: 2 times.

Validating 55:
        50 89
50
        15 23
==== AccessTime: 4 times.
```

图 **9**　测试查找操作的访存次数

测试结果：查找成功和查找失败都需要两次读入，和理论分析的访存次数一致。

Test 4：

在 *dev – cpp* 上使用 32 位编译器（此时指针的大小为 4 字节符合题设）使用笔者编写的 *displaySize* 函数使用 *sizeof* 查看 *Node* 和 *Leaf* 的大小：

```
Record ini[ 2 * 2 ];
ini[0] = Record(15);
ini[1] = Record(23);
ini[2] = Record(89);
ini[3] = Record(50);

B_Tree<Record,int,getKey> tree(ini);
cout << "==== Initial:\n==== M = 5\n==
<< endl << "==== Origin:\n";
tree.display();
tree.displaySize();
```

```
==== Origin:
        50 89
50
        15 23
==== AccessTime: 0 times.

==== Displaying Node and Leaf's size:
Node: 40 Bytes
Leaf: 36 Bytes
==== If using a 64-bits complier, sizeof a pointer is 8 Bytes,
==== then this function is just for fun.
==== If using a 32-bits complier (you can use dev-cpp to use it),
==== You will get the ideal result:
==== Node: 40 Bytes
==== Leaf: 36 Bytes
```

图 **10**　打印结点和叶子的大小

符合理论的分析。

五、 总结和讨论

由程序测试 1 中并未输出 Obs 可得，笔者和思宇同学编写的 B+ 树在能够合理实现插入删除查找操作。在对数据进行插入和删除过程中，并未出现破坏 B+ 树性质的操作。因此可以认为，编写的 B+ 树仿真器能够对 B+ 树的操作进行合理的模拟。

由测试 2 和 3 对访存次数的测试可得，编写的 B+ 树模拟器能够很好地模拟对磁盘的访问。由测试 4 可得，编写的 B+ 树结点符合外部存储块的大小设计。

通过本次对 B+ 树仿真器的设计，笔者对逻辑结构中树的认知更为地清楚深刻。在编写过程中笔者发现，B+ 树去除叶子就是一个 B-树，利用好这个特性便可较为轻松地对 B+ 树进行编写。在数据结构的设计上，笔者认为值得称道的是对 $union$ 联合体的妙用，使得 B+ 树能够最大限度地利用外部存储块的存储空间。在对 B+ 树的查找、插入算法的设计中，采用了 $bottom-up$ 方法编写。经过此次对 B+ 树仿真器的编写，笔者对于合作编写代码、对 $github$ 网站的运用更为熟练、对树形数据结构的理解也更为深刻，总而言之，收获颇丰。