Project1

1.implement RedBlackTree and B-Tree

具体代码见RB_Tree.c，B_Tree.c，Tree.h ，其中相关结构体定义及函数声明在Tree.h。

下面是红黑树和 B 树的基本操作的思路，包括插入、删除、前序遍历和搜索。

insert

1.红黑树 (Red-Black Tree)
过程：按照二叉搜索树的插入规则插入节点，初始时将新节点的颜色设为红色。检查父节点颜色，如果是红色，则进行调整。如果父节点也是红色，通过叔叔节点的颜色判断进行不同的调整（调整颜色、旋转）。

2.B 树 (B-Tree)

过程：首先找到叶子节点的位置。将新键添加到该叶子节点，并保持按排序顺序。如果该节点已满，则需要分裂该节点，将中间键上升到父节点。继续递归处理，直到根节点。如果根节点分裂，树高增加。

delete

1.红黑树 (Red-Black Tree)

过程：对于红黑树的删除，首先需要找到要删除的节点，然后删除它并替换为其后继节点（如果是内部节点）。删除后如果删除的节点是黑色，会破坏红黑树的性质，需要进行修复。修复可以通过借用兄弟节点，合并或进行旋转来实现。

2.B 树 (B-Tree)

过程：查找要删除的键。如果是内部节点，用后继或前驱替代删除的键。如果是叶子节点, 直接删除。处理可能因删除导致的节点个数不足的情况，可以通过借用兄弟节点的键，或者合并两个节点。删除可能导致根节点下移，保持树的平衡。

preorderPrint

1.红黑树 (Red-Black Tree)

过程：递归地访问节点，首先访问根节点，接着依次访问左子树和右子树，输出节点的键和值到文件。

2.B 树 (B-Tree)

过程：从根节点开始，顺序访问每个节点，先访问节点的键，然后向下遍历其子节点，将节点的键和值输出到文件。

search and search_in_range

1.红黑树 (Red-Black Tree)

过程：从根节点开始，比较要查找的关键字与当前节点的关键字，根据比较结果决定向左子树或右子树递归查找，直到找到目标节点或者到达空节点。

2.B 树 (B-Tree)

过程：从根节点开始，比较查找的关键字与当前节点的键，利用有序特性，决定从哪个子节点继续查找，直到找到该键或达到叶子节点。

2.Compare RedBlackTree and B tree

具体代码见project1。（编译运行请参考README.md）

main.c相关执行代码截取如下：

```
    // Step 1: Insert into trees from 1_initial.txt
    printf("Step 1: Insert initial data\n");
    process_file("1_initial.txt", 0, rb_tree, b_tree); // B-tree
    process_file("1_initial.txt", 1, rb_tree, b_tree); // Red-Black tree

    // Step 2: Delete from trees based on 2_delete.txt
    printf("Step 2: Delete data\n");
    process_file("2_delete.txt", 0, rb_tree, b_tree); // B-tree
    process_file("2_delete.txt", 1, rb_tree, b_tree); // Red-Black tree

    // Step 3: Add data from 3_insert.txt
    printf("Step 3: Add data\n");
    process_file("3_insert.txt", 0, rb_tree, b_tree); // B-tree
    process_file("3_insert.txt", 1, rb_tree, b_tree); // Red-Black tree

    // Step 4: Query a word
    char *single_query[] = {"android"}; // Replace with an actual word to query
    printf("Step 4: Query a word\n");
    query_words(rb_tree, b_tree, 0, single_query, 1); // B-tree
    query_words(rb_tree, b_tree, 1, single_query, 1); // Red-Black tree

    // Step 5: Query some words
    char *multi_query[] = {"and", "android"}; // search words between word[0]
and word[1]
    printf("Step 5: Query some words\n");
    query_words(rb_tree, b_tree, 0, multi_query, 2); // B-tree
    query_words(rb_tree, b_tree, 1, multi_query, 2); // Red-Black tree
```

main.c执行结果如下：

```
Step 1: Insert initial data
B Tree
Processed 100 entries, time used: 121.400000 us
B Tree
Processed 100 entries, time used: 96.400000 us
B Tree
Processed 100 entries, time used: 58.900000 us
B Tree
Processed 100 entries, time used: 79.400000 us
B Tree
Processed 100 entries, time used: 56.300000 us
B Tree
Processed 100 entries, time used: 61.600000 us
B Tree
Processed 100 entries, time used: 57.500000 us
B Tree
Processed 100 entries, time used: 82.100000 us
```

```
B Tree
Processed 100 entries, time used: 53.500000 us
B Tree
Processed 100 entries, time used: 61.800000 us
B Tree
Processed 100 entries, time used: 49.600000 us
B Tree
Processed 100 entries, time used: 74.100000 us
B Tree
Processed 100 entries, time used: 53.300000 us
B Tree
Processed 100 entries, time used: 58.500000 us
B Tree
Processed 100 entries, time used: 45.600000 us
B Tree
Processed 100 entries, time used: 69.300000 us
B Tree
Processed 100 entries, time used: 50.600000 us
B Tree
Processed 100 entries, time used: 68.800000 us
B Tree
Processed 100 entries, time used: 46.000000 us
B Tree
Processed 100 entries, time used: 49.900000 us
B Tree
Processed 100 entries, time used: 78.600000 us
B Tree
Processed 100 entries, time used: 50.400000 us
B Tree
Processed 100 entries, time used: 38.100000 us
B Tree
Processed 100 entries, time used: 69.700000 us
B Tree
Processed 100 entries, time used: 55.600000 us
B Tree
Processed 100 entries, time used: 60.500000 us
B Tree
Processed 100 entries, time used: 50.300000 us
B Tree
Processed 100 entries, time used: 58.300000 us
B Tree
Processed 100 entries, time used: 50.000000 us
B Tree
Processed 100 entries, time used: 83.700000 us
B Tree
Processed 100 entries, time used: 37.300000 us
B Tree
Processed 100 entries, time used: 53.000000 us
B Tree
Processed 100 entries, time used: 45.100000 us
Total time: 2025.200000 us
It has been printed to bt.txt
RedBlack Tree
Processed 100 entries, time used: 57.900000 us
RedBlack Tree
Processed 100 entries, time used: 55.900000 us
RedBlack Tree
Processed 100 entries, time used: 49.000000 us
```

```
RedBlack Tree
Processed 100 entries, time used: 69.800000 us
RedBlack Tree
Processed 100 entries, time used: 47.000000 us
RedBlack Tree
Processed 100 entries, time used: 60.300000 us
RedBlack Tree
Processed 100 entries, time used: 41.600000 us
RedBlack Tree
Processed 100 entries, time used: 57.300000 us
RedBlack Tree
Processed 100 entries, time used: 88.500000 us
RedBlack Tree
Processed 100 entries, time used: 58.000000 us
RedBlack Tree
Processed 100 entries, time used: 54.300000 us
RedBlack Tree
Processed 100 entries, time used: 49.800000 us
RedBlack Tree
Processed 100 entries, time used: 50.800000 us
RedBlack Tree
Processed 100 entries, time used: 45.500000 us
RedBlack Tree
Processed 100 entries, time used: 49.100000 us
RedBlack Tree
Processed 100 entries, time used: 50.400000 us
RedBlack Tree
Processed 100 entries, time used: 45.500000 us
RedBlack Tree
Processed 100 entries, time used: 60.200000 us
RedBlack Tree
Processed 100 entries, time used: 52.900000 us
RedBlack Tree
Processed 100 entries, time used: 58.500000 us
RedBlack Tree
Processed 100 entries, time used: 39.900000 us
RedBlack Tree
Processed 100 entries, time used: 48.400000 us
RedBlack Tree
Processed 100 entries, time used: 77.600000 us
RedBlack Tree
Processed 100 entries, time used: 47.000000 us
RedBlack Tree
Processed 100 entries, time used: 51.300000 us
RedBlack Tree
Processed 100 entries, time used: 71.000000 us
RedBlack Tree
Processed 100 entries, time used: 133.500000 us
RedBlack Tree
Processed 100 entries, time used: 174.600000 us
RedBlack Tree
Processed 100 entries, time used: 134.900000 us
RedBlack Tree
Processed 100 entries, time used: 91.900000 us
RedBlack Tree
Processed 100 entries, time used: 40.700000 us
RedBlack Tree
Processed 100 entries, time used: 62.100000 us
```

```
RedBlack Tree
Processed 100 entries, time used: 47.500000 us
Total time: 2122.700000 us
It has been printed to rbt.txt
Step 2: Delete data
B Tree
Processed 100 entries, time used: 95.000000 us
Total time: 95.000000 us
It has been printed to bt.txt
RedBlack Tree
Processed 100 entries, time used: 170.000000 us
Total time: 170.000000 us
It has been printed to rbt.txt
Step 3: Add data
B Tree
Processed 100 entries, time used: 97.600000 us
Total time: 97.600000 us
It has been printed to bt.txt
RedBlack Tree
Processed 100 entries, time used: 97.300000 us
Total time: 97.300000 us
It has been printed to rbt.txt
Step 4: Query a word
B Tree
Meaning of 'android' : 机器人
RedBlackTree
Meaning of 'android' : 机器人
Step 5: Query some words
B Tree
Meaning of 'andirons': 铁制柴架
Meaning of 'andaman': 安达曼人
Meaning of 'andersen': 安徒生
Meaning of 'andesite': 安山石
Meaning of 'androecium': 雄蕊
Meaning of 'android': 机器人
6 words in range [and, android] found.
RedBlackTree
Meaning of 'andaman': 安达曼人
Meaning of 'andersen': 安徒生
Meaning of 'andesite': 安山石
Meaning of 'andirons': 铁制柴架
Meaning of 'androecium': 雄蕊
Meaning of 'android': 机器人
6 words in range [and, android] found.
```

1. 插入数据（Insert data and add data)

B 树：共进行了多次插入 100 个条目的操作，时间数据有：121.400000 us、96.400000 us等。这些时间数据波动较大，总时间为 2025.200000 us。从数据来看，处理速度有快有慢，但大部分时间在几十微秒的范围内。最快的一次插入 100 个条目仅用了 37.300000 us，最慢的一次用了 121.400000 us。分析：插入操作相对复杂一些。当插入一个新元素时，可能需要对节点进行分裂操作。不过，由于 B 树的多路特性，其树的高度增长相对较慢。在数据量较大时，平均插入性能较好，时间复杂度也为O(log n) 。例如，对于大量数据的存储和插入，B 树可以利用其节点存储多个元素的特点，减少磁盘 I/O 次数。

红黑树：同样是插入 100 个条目的操作，时间数据如 57.900000 us、55.900000 uss 等。总时间为 2122.700000 us。其时间数据也有波动，不过在插入操作中，大部分时间也在几十微秒，有几次时间较长，如 133.500000 us、174.600000 us、134.900000 us 等。最快的一次插入 100 个条目仅用了 39.900000 us，最慢的一次达到了 174.600000 us。

分析：插入新节点时，需要通过变色和旋转操作来维持红黑树的性质。在最坏情况下，插入操作可能需要进行多次旋转和变色，时间复杂度为O(log n)，但在频繁插入的情况下，可能由于调整操作较多而导致一定的性能开销。例如，当插入的节点序列恰好是有序的，红黑树可能需要较多的调整来保持平衡。

2. 删除数据（Delete data）

B 树：处理 100 个条目删除操作的时间为 95.000000 us。相对插入操作中的某些时间值，这个删除时间处于中等水平。删除操作可能涉及节点的合并或借元素等操作来维持 B 树的性质。一般情况下，B 树的删除操作也具有较好的性能，时间复杂度为O(log n)。但如果删除操作导致节点频繁合并或调整，可能会有一定的性能损耗，不过相比红黑树，B 树在处理大量数据删除时可能更具优势，因为它的高度较低，调整操作对整体性能的影响相对较小。

红黑树：处理 100 个条目删除操作的时间为 170.000000 us，比 B 树的删除时间长，可能意味着在这种情况下红黑树的删除操作相对更耗时。删除节点时，同样需要通过变色和旋转来维护红黑树的性质，以保证树的平衡。在最坏情况下，时间复杂度为O(log n)。例如，当删除的节点导致树的平衡结构受到较大影响时，可能需要较多的调整操作。

3. 查询操作（Query）

在查询单个单词（如 “android”）和查询多个单词（如 “andirons”、“andaman” 等）的操作中，B 树和红黑树都能正确查询到结果。

B 树：同样时间复杂度为O(log n)，但由于 B 树的多路特性，每个节点存储多个元素，可以在一次磁盘 I/O（如果数据存储在磁盘上）中获取更多的信息，减少查询过程中的 I/O 次数，在数据量较大且存储在磁盘等外部存储设备时，查询性能可能更优。

红黑树：由于是二叉树结构，查询操作的时间复杂度为 O(log n)。每次查询时，需要从根节点开始，沿着二叉树的路径向下比较。

总结

红黑树的优缺点

- 优点：**平衡性好**：通过红黑规则保证了树的高度相对平衡，从根节点到叶子节点的最长路径不会超过最短路径的两倍。这使得查找、插入和删除等操作的时间复杂度稳定在 ，在动态数据集合中能保持较好的性能。例如，在频繁插入和删除数据的情况下，红黑树能够自动调整平衡，保证操作效率。

  适合内存操作：在内存中，由于其二叉树的特性，遍历和操作的逻辑相对清晰，对于小规模数据或者对内存数据结构要求较高的场景（如一些编程语言的标准库中的数据结构实现）表现良好。

- 缺点：**存储效率相对B树较低**：每个节点只存储一个数据元素（加上颜色等额外信息），与 B 树相比，在存储大量数据时，可能会占用更多的节点，导致树的高度相对较高，在需要频繁磁盘 I/O 操作的场景下（如数据库索引）性能可能会受到影响。

  频繁调整影响性能：在插入或删除操作时，可能需要多次的旋转和变色操作来维持平衡。当连续插入或删除的数据具有一定顺序性时，调整操作可能会比较频繁，一定程度上影响操作的效率。

B 树的优缺点

- 优点：**多路存储提高效率**：作为多路平衡查找树，B 树的每个节点可以存储多个数据元素和多个子节点指针。这使得 B 树的高度相对较低，能够减少磁盘 I/O 操作次数（在数据存储在磁盘的场景下）。例如，在数据库索引中，B 树可以在一个磁盘块中存储多个索引项，提高查询效率。

数据批量处理优势：对于批量插入和删除操作，B 树可以利用其节点存储多个元素的特点，更好地处理数据的分布和调整，减少树结构的频繁变动。

空间利用率较高：一个节点能存储多个数据元素，相比二叉树，能更有效地利用磁盘空间和内存空间，尤其是在处理大规模数据时。

- 缺点：**不适合小规模数据或内存操作**：由于其节点结构相对复杂，在内存中处理小规模数据时，可能会因为额外的开销（如节点分裂、合并的判断和操作）而导致性能不如红黑树。而且对于内存中的数据操作，二叉树的简单性在某些情况下更具优势。

3.Dictionary操作指南

具体代码见文件夹Dictionary，其中main函数位于dictionary.c，文件夹中1.txt,2.txt,3.txt仅为示例文件，（编译运行请参考README.md）。

```
//操作指南
void print_help()
{
    printf("Chinese-English Dictionary\n");
    printf("------------------------------------------------------------------------
\n");
    printf("i <document>        initialize with document\n"
           "d <document>        delete words with document\n"
           "n <document>        insert new words with document\n"
           "R                   use Red-Black Tree\n"
           "B                   use B Tree\n"
           "s <word1> <word2>   search words from word1 to word2\n"
           "s <word>            search a word and its meaning\n"
           "I <word> <meaning>  insert a single word\n"
           "D <word>            delete a single word\n"
           "pr                  print RedBlackTree to rbt.txt\n"
           "pb                  print B-Tree to bt.txt\n"
           "q                   quit\n"
           "h                   print help\n");
    printf("------------------------------------------------------------------------
\n");
}
```

上述是一个基于二叉搜索树的汉英词典操作说明，包括以下几部分：

1.自行挑选红黑树和B树（R，B），开始时请挑选一种树R or B，不挑选则默认RedBlackTree。

2.初始化（i），在对树用文件初始化后，之前该种树的数据清零，并添加文件中的数据。

示例：

```
i D:\dictionary_big.txt //请不要给文件路径加上双引号
```

注意：为衔接dictionary_big.txt内容，文件第一行请不要包含：INSERT.

3.插入（n，I）/删除（d，D）

- 支持从文件中批量操作单词，文件第一行表示操作类型（INSERT，DELETE），后续行均为数据，每次操作完成后调用前序打印方法将树打印到相应文件（rbt.txt或bt.txt）。
- 支持单个单词的插入和删除操作。

4.搜索（s）：

- 支持给定范围查询单词及其释义，边界值不要求为确切单词。
- 支持单个单词查询中文释义。

5.退出程序（q）：在释放内存后直接退出程序。

6.打印（pr，pb）:将RedBlackTree，B-Tree内容打印至rbt.txt或bt.txt文件中。

6.帮助（h）：打印帮助。

操作示例：

```
PS C:\Users\33513\Desktop\Dictionary> [Console]::OutputEncoding = [System.Text.UTF8Encoding]::new()
PS C:\Users\33513\Desktop\Dictionary> gcc -std=c99 -o main Dictionary.c B_Tree.c RB_Tree.c Tree.h -fexec-charset=utf-8
PS C:\Users\33513\Desktop\Dictionary> ./main
Chinese-English Dictionary
------------------------------------------------------------------
i <document>        initialize with document
d <document>        delete words with document
n <document>        insert new words with document
R                   use Red-Black Tree
B                   use B Tree
s <word1> <word2>   search words from word1 to word2
s <word>            search a word and its meaning
I <word> <meaning>  insert a single word
D <word>            delete a single word
pr                  print RedBlackTree to rbt.txt
pb                  print B-Tree to bt.txt
q                   quit
h                   print help
------------------------------------------------------------------
First you should choose which tree to implement: R or B
i 1.txt
Dictionary has been initialized.
```

```
Chinese-English Dictionary
------------------------------------------------------------------
i <document>        initialize with document
d <document>        delete words with document
n <document>        insert new words with document
R                   use Red-Black Tree
B                   use B Tree
s <word1> <word2>   search words from word1 to word2
s <word>            search a word and its meaning
I <word> <meaning>  insert a single word
D <word>            delete a single word
q                   quit
h                   print help
------------------------------------------------------------------
s asan
Meaning of 'asan': n.(Asan)人名；(俄、塞、罗)阿桑.
s asan asana
Meaning of 'asan': n.(Asan)人名；(俄、塞、罗)阿桑
Meaning of 'asana': n.瑜珈的任何一种姿势
2 words in range [asan, asana] found.
```

4.Bonus

具体代码已写在bonus.c中。

具体思路：修改普通读取文件代码，改为mmap方法读取。

main函数读取：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "B_Tree.h"
#include <windows.h>

#define BUFFER_SIZE 8192 // 增大缓冲区大小
```

```c
// 读取超大文件b树使用磁盘存取大数据集
int main()
{
    // B - tree
    BTree *b_tree = malloc(sizeof(BTree));
    if (b_tree == NULL)
    {
        perror("Memory allocation failed for BTree");
        return 1;
    }
    b_tree->root = NULL;
    // 读取文件，并插入B树
    const char *path = "D:\\dictionary_big.txt"; // 路径可根据存储路径修改，注意路径为
\\

    // Create file mapping
    HANDLE hFile = CreateFile(path, GENERIC_READ, 0, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE)
    {
        printf("Could not open file %s\n", path);
        free(b_tree);
        return 0;
    }

    HANDLE hMapping = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
    if (hMapping == NULL)
    {
        printf("Could not create file mapping.\n");
        CloseHandle(hFile);
        free(b_tree);
        return 0;
    }

    LPVOID pMap = MapViewOfFile(hMapping, FILE_MAP_READ, 0, 0, 0);
    if (pMap == NULL)
    {
        printf("Could not map view of file.\n");
        CloseHandle(hMapping);
        CloseHandle(hFile);
        free(b_tree);
        return 0;
    }

    // 逐行处理映射的内存
    char *ptr = (char *)pMap;
    char *line_end;
    char word[100], meaning[200];

    while ((line_end = strchr(ptr, '\n')) != NULL)
    {
        *line_end = '\0'; // 替换换行符为字符串结束符
        char *separator = strchr(ptr, ' ');
        if (separator != NULL)
        {
            *separator = '\0'; // 分别处理键和值
            strcpy(word, ptr);
            strcpy(meaning, separator + 1);
```

```
            B_insert(b_tree, word, meaning);
        }
        ptr = line_end + 1; // 移动到下一行
    }

    // 释放资源
    UnmapViewOfFile(pMap);
    CloseHandle(hMapping);
    CloseHandle(hFile);

    B_free_tree(b_tree);
    return 0;
}
```

为在 Windows 系统下读取超大文件dictionary_big.txt，做出修改：

内存映射文件（Memory - Mapped Files）：创建内存映射文件：使用Windows API中的 CreateFile 函数打开文件。使用 CreateFileMapping创建文件映射。使用 MapViewOfFile 将映射视图映射到进程的地址空间。逐行处理内存映射的文件：使用指针遍历映射的内存区域，查找换行符来分割行。对每一行进行处理，分割出单词和词义，并将其插入到B树中。综上，使用mmap方法提高文件读取性能，从而实现对超大文件的读取，并插入B树中。