09020328-王亮-数据结构实践报告2

External Sort: Quick Sort

1.问题描述

在对存储在磁盘中的大量数据进行行排序时，由于内存大小的限制，无法将整个文件完整地调入内存，故传统的快速排序算法无法满足外部排序的需求。本实验使用快速排序的思想实现外排。

2.算法思想

快排的排序依据在于每次能够定下来一个数字的绝对位置，即其左必更小，其右必更大，基于此条件，可递归实现对左右两侧的排序。外排下快排的不同在于每次排序不是定下来一个数字的位置，而是定下一个middle cache的位置。

3.功能模块设计

Cache类设计：

使用1格大小的input output作为输入 输出缓存

CACHE_CAPACITY 设置 middle缓存的大小

middle缓存用于排序，通过使用stl map实现最大最小堆（map红黑树实现，自动排序）

当从file读取到input时 readCount++ 从output写入file时writeCount++

```
class Cache {
  map<int, int> middle;
  int size = 0;

  int input = ILLEGAL;
  int output = ILLEGAL;

  int capacity = CACHE_CAPACITY;

public:
  int readCount = 0;
  int writeCount = 0;
```

file文件输入的模拟：

以数组方式模拟实现，索引1开始，填充随机数据

```
int file[FILE_SIZE + 2]{0};

void fillWithRandom() {
  srand(time(0));
  for(int i = 1; i <= FILE_SIZE; i++) {
    file[i] = rand();
  }
}
```

最大最小堆设计：

由于map内部自动排序，通过begin()迭代器获取最小，--end()获取最大，当元素对应value(即出现次数，若大于1则意味着重复)为0时清除此key值。

```
int getMin() {
    return (*(middle.begin())).first;
  }

  int getMax() {
    return (*(--middle.end())).first;
  }

  void popMin() {
    middle[getMin()]--;
    if (!middle[getMin()]) {
      middle.erase(middle.begin());
    }
  }

  void popMax() {
    middle[getMax()]--;
    if (!middle[getMax()]) {
      middle.erase(--middle.end());
    }
  }
```

4.测试结果与分析

（数据随机，可能存在一定波动）

如果文件大小小于middle(命名为CACHE_CAPACITY，实际上还有一个input和output)的大小，就是全部数据读一次，再多读一次终止，最后全部写入。

如果大于，middle大小不变时，读写次数随文件大小增加。

文件大小不变时，读写次数随middle大小减少。

```
constexpr int CACHE_CAPACITY = 50; // middle
constexpr int FILE_SIZE = 30;

Read: 31 Write: 30
```

```
constexpr int CACHE_CAPACITY = 50; // middle
constexpr int FILE_SIZE = 50;

Read: 51 Write: 50
```

```
constexpr int CACHE_CAPACITY = 50; // middle
constexpr int FILE_SIZE = 100;

Read: 156 Write: 153
```

```
constexpr int CACHE_CAPACITY = 50; // middle
constexpr int FILE_SIZE = 150;

Read: 308 Write: 303
```

```
constexpr int CACHE_CAPACITY = 70; // middle
constexpr int FILE_SIZE = 100;

Read: 133 Write: 131
```

```
constexpr int CACHE_CAPACITY = 90; // middle
constexpr int FILE_SIZE = 100;

Read: 113 Write: 111
```

5.实验总结

快排的思路迁移到外排并没有太过显著的思路改变，主要在于对于middle的一个整体性的理解。

6.源代码

```
#include <bits/stdc++.h>
using namespace std;

constexpr int ILLEGAL = INT_MAX;
constexpr int CACHE_CAPACITY = 90; // middle
constexpr int FILE_SIZE = 100;
int file[FILE_SIZE + 2]{0};

class Cache {
  map<int, int> middle;
  int size = 0;

  int input = ILLEGAL;
  int output = ILLEGAL;

  int capacity = CACHE_CAPACITY;

public:
  int readCount = 0;
  int writeCount = 0;

  void fileToInput(int *file, int offset) {
    input = *(file + offset);
    readCount++;
  }

  bool inputToMiddle() {
    if (size == capacity) return false;
    middle[input]++;
    size++;
    return true;
  }
```

```
void inputToOutput() {
  output = input;
}

int getInput() {
  return input;
}

int getOutput() {
  return output;
}

int getMin() {
  return (*(middle.begin())).first;
}

int getMax() {
  return (*(--middle.end())).first;
}

void popMin() {
  middle[getMin()]--;
  if (!middle[getMin()]) {
    middle.erase(middle.begin());
  }
}

void popMax() {
  middle[getMax()]--;
  if (!middle[getMax()]) {
    middle.erase(--middle.end());
  }
}

bool MinToOutput() {
  if (isEmpty()) {
    return false;
  }
  output = getMin();
  popMin();
  size--;
  return true;
}

bool MaxToOutput() {
  if (isEmpty()) {
    return false;
  }
  output = getMax();
  popMax();
  size--;
  return true;
}

void outputToFile(int *file, int offset) {
```

```
    *(file + offset) = output;
    writeCount++;
  }

  bool isEmpty() {
    return !size;
  }

} cache;

void fillWithRandom() {
  srand(time(0));
  for (int i = 1; i <= FILE_SIZE; i++) {
    file[i] = rand();
  }
}

void outerQuickSort(int l, int r) {

  if (l >= r) return;

  int leftFilePointer = l, rightFilePointer = r;
  int currentLeftPointer = l, currentRightPointer = r;

  cache.fileToInput(file, currentLeftPointer++);

  while (currentLeftPointer - 1 <= currentRightPointer) {
    if (!cache.inputToMiddle()) {
      if (cache.getInput() < cache.getMin()) {
        cache.inputToOutput();
        cache.fileToInput(file, currentLeftPointer++);
        cache.outputToFile(file, leftFilePointer++);
      } else if (cache.getInput() > cache.getMax()) {
        cache.inputToOutput();
        cache.fileToInput(file, rightFilePointer);
        cache.outputToFile(file, rightFilePointer--);
        currentRightPointer = rightFilePointer;
      } else {
        cache.MinToOutput();
        cache.outputToFile(file, leftFilePointer++);
        cache.inputToMiddle();
        cache.fileToInput(file, currentLeftPointer++);
      }
    } else {
      cache.fileToInput(file, currentLeftPointer++);
    }
  }

  int splitLeft = leftFilePointer, splitRight = rightFilePointer;

  while(!cache.isEmpty()) {
    cache.MinToOutput();
    cache.outputToFile(file, leftFilePointer++);
  }
```

```
  outerQuickSort(l, splitLeft);
  outerQuickSort(splitRight, r);
}

void printFile() {
  for (int i = 1; i <= FILE_SIZE; i++) {
    cout << file[i] << " ";
  }
  cout << endl;
}

bool check() {
  for (int i = 2; i <= FILE_SIZE; i++) {
    if (file[i - 1] > file[i]) {
      return false;
    }
  }
  return true;
}

int main() {

  fillWithRandom();

  cout << "------------Original File: -------------" << endl;
  printFile();
  cout << "------------End Original File -------------" << endl;

  outerQuickSort(1, FILE_SIZE);

  if (!check()) {
    cout << "false" << endl;
    return 0;
  }

  cout << "------------Modified File: -------------" << endl;
  printFile();
  cout << "------------End Modified File: -------------" << endl;

  cout << "------------Count Of File: -------------" << endl;

  cout << "Read: " << cache.readCount << " Write: " << cache.writeCount << endl;
}
```