ECS 60 Programming Assignment #4, 50 points (2.5 hours coding) Fall 2017

Due: Wednesday, November 22nd at 11:59pm to p4 directory
Primary class file name: defragmenter.cpp Executable name: defrag.out Makefile name: Makefile
Minimal files to submit: authors.csv, defragmenter.cpp, defragmenter.h, Makefile.

For this program you are to write a disk defragmenter simulator that re-orgranizes the files on a simulated disk. Your program must rearrange the disk blocks so that: 1) the first file starts on disk block #2, 2) all of the blocks of each file are in sequence and contiguous, and 3) all unused blocks are located at the end of the disk.

The DiskDrive class provides simulated disk block access. A disk is simulated with three data structures: 1) a public array of bools, File Allocation Table (FAT), with each bool indicating whether the corresponding disk block is used (true) or unused (false); 2) a public array of DirectoryEntry's sorted by file names, that provide names, starting blockIDs, and sizes of the files; and 3) a private array of DiskBlock's, that contains the critical information for each block. Your class only has access to the nextBlockID of a DiskBlock. The filename and fileBlockNum are used to check to see if you preserved the ordering of the blocks in the file.

You are to handin all files upon which your program is dependent, except DefragRunner.cpp, DefragRunner.h, mynew.cpp, mynew.h, and CPUTimer.h. The grading script will copy those five files, and into your directory. You will find those five files, disk data files, a barebones defragmenter.cpp, a barebones degramenter.h, a barebones Makefile, my defrag.out, and testhw in ~ssdavis/60/p4.

Here are the specifications:

1. Dynamic Memory Allocation.
    1.1. You may not use malloc(), free(), maxRAM, or currentRAM anywhere in your source code, including any Weiss files. No entity larger the 80 bytes may be created without using new. If you use any outside files, other than template files, you must add #include "mynew.h" at the top of the files.
    1.2. Each call to readDiskBlock() dynamically allocates a DiskBlock (~512 bytes). writeDiskBlock() does NOT deallocate dynamic memory, so you will probably wish to explicitly call delete after calling writeDiskBlock().
    1.3. Your program cannot use more than eight million bytes of dynamic memory at any time.
2. Disk access
    2.1. Each time you call readDiskBlock(), or writeDiskBlock() counts as a disk access.
3. FAT table is a public array of bools that you may use as you see fit. Those disk blocks that are used are initially marked true in the FAT table.
4. CreateFile
    4.1. The name of a data file reflects parameters used to create it. For example, Disk20_3_80_50_892.dat has 20 disk blocks, 3 files, is ~80% full, ~50% of the files are fragmented, and was created using a seed of 892 for the random number generator.
    4.2. The first line of the file is the capacity and number of files.
    4.3. This first line is followed by a list of the directory for the disk, one file per line.
        4.3.1. Each line starts with the name of a file <int>, then first disk block used by the file <unsigned>, and then its size <int>.
    4.4. After the directory listing is a list of the contents of each of the disk blocks in the disk.
        4.4.1. The first two blocks, 0 and 1, are reserved, and should never be touched.
        4.4.2. The disk block information is arranged as follows: DiskBlockID (unsigned), File name (unsigned short), file block number (unsigned short), NextBlockID of file (unsigned). A next of 0 indicates an empty block, a next of 1 indicates the last block in a file.
5. Grading
    5.1. Performance will be tested with three disks, each with a capacity of 500,000 blocks filled to about 85% containing about 4000 files with about 30% fragmentation.
    5.2. (20 points) Correctly defragments the disk using no more than eight million bytes. If a program does not correctly defragment the disk or uses more than eight million bytes, then it will you will receive zero for the entire assignment.
    5.3. (15 points) CPU time: min (17, 15 * Sean's CPU Time / Your CPU Time);
        5.3.1. CPU time may not exceed 10.
        5.3.2. Programs must be compiled without any optimization options.
    5.4. (15 points) Disk accesses: min(18, 15 * Sean's Disk Accesses / Your Disk Accesses)
6. Suggestions

6.1. You should plan on moving DiskBlocks temporarily within the DiskDrive to keep below the eight million RAM limit. Since each DiskBlock takes ~512 bytes, you could have no more than ~16,000 DiskBlocks in RAM at any one time. After finishing your program, you can probably tune it to be as close to eight million bytes without exceeding it.

6.2. Remember to turn in dsexceptions.h if your program needs it!

```
[ssdavis@lect1 p4]$ cat DefragRunner.h
#ifndef DefragRunnerH
#define DefragRunnerH

#include <iostream>
#include <fstream>
using namespace std;

class DiskDrive;
class DiskBlock;

class DiskBlockInfo
{
  unsigned nextBlockID;
  unsigned short fileID;
  unsigned short fileBlockNum;
  friend class DiskDrive;
  friend class DiskBlock;
  void read(ifstream &inf){inf >> fileID >> fileBlockNum >> nextBlockID; }
public:
  DiskBlockInfo(): nextBlockID(0), fileID(0), fileBlockNum(0) {}
}; // class DiskBlockInfo


class DiskBlock
{
  friend class DiskDrive;
  DiskBlockInfo blockInfo;
  char stuff[500];
  DiskBlock(DiskBlockInfo &blockInf) {blockInfo = blockInf;}
public:
  DiskBlock(){}
  unsigned getNext() const  {return blockInfo.nextBlockID;}
  void setNext(unsigned nextBlockID) { blockInfo.nextBlockID = nextBlockID; }
  short getFileBlockNum() const {return blockInfo.fileBlockNum;}
}; // class DiskBlock

class DirectoryEntry
{

  unsigned short fileID;
  unsigned firstBlockID;
  unsigned size;
  friend class DiskDrive;
  void read(istream &inf) {inf >> fileID >> firstBlockID >> size;}
public:
  DirectoryEntry(unsigned short n = 0, unsigned f = 0, int s = 0) : fileID(n),
    firstBlockID(f), size(s){}
  bool operator< (const DirectoryEntry &d)const {return fileID < d.fileID;}
  bool operator== (const DirectoryEntry &d)const {return fileID == d.fileID;}
  DirectoryEntry& operator= (const DirectoryEntry *d){return *this;}
```

```
    // prevents cheating by altering original
  void print(ostream &outf) const
    {outf << fileID << ' ' << firstBlockID << ' ' << size << endl;}
  unsigned short getFileID() const {return fileID;}
  unsigned getFirstBlockID() const {return firstBlockID;}
  unsigned getSize() const {return size;}
  void setFirstBlockID(unsigned blockID) {firstBlockID = blockID;}
}; // DirectoryEntry class

class DiskDrive
{
  int numFiles;
  unsigned capacity;
  int diskAccesses;
  DiskBlockInfo *disk;
public:
  DirectoryEntry *directory;
  bool *FAT;
  DiskDrive():diskAccesses(0){}
  void readFile(const char filename[]);
  int getNumFiles()const {return numFiles;}
  int getCapacity()const {return capacity;}
  int getDiskAccesses() const {return diskAccesses; }
  DiskBlock* readDiskBlock(int diskBlockID)
    {diskAccesses++; return new DiskBlock(disk[diskBlockID]); }
  void writeDiskBlock(DiskBlock *diskBlock, int diskBlockID)
  {
    disk[diskBlockID] = diskBlock->blockInfo;
    diskAccesses++;
  }  // copies diskBlockInfo
  void check();
} ;  // DiskDrive class
#endif
[ssdavis@lect1 p4]$

int main(int argc, char *argv[])
{
  DiskDrive diskDrive;
  diskDrive.readFile(argv[1]);
  CPUTimer ct;
  currentRAM = maxRAM = 0;
  ct.reset();
  new Defragmenter(&diskDrive);
  cout << "CPU Time: " << ct.cur_CPUTime() << " Disk accesses: "
    << diskDrive.getDiskAccesses() << " RAM: " << maxRAM << endl;
  diskDrive.check();
  return 0;
}  // main

[ssdavis@lect1 p4]$ CreateDisk.out
Capacity with first two reserved (in disk blocks): 100
Number of files: 5
Percentage used by files: 80
Percentage fragmented: 33
Seed: 1
[ssdavis@lect1 p4]$
```

```
[ssdavis@lect1 p4]$ cat
Disk100_5_80_33_1.txt
100 5
7673 18 6
8183 92 38
17452 68 23
38754 2 5
63370 9 8
0 0 0 0
1 0 0 0
2 38754 1 94
3 0 0 0
4 8183 29 34
5 0 0 0
6 8183 3 67
7 0 0 0
8 38754 5 1
9 63370 1 10
10 63370 2 11
11 63370 3 12
12 63370 4 13
13 63370 5 14
14 63370 6 15
15 63370 7 16
16 63370 8 1
17 8183 18 39
18 7673 1 19
19 7673 2 20
20 7673 3 21
21 7673 4 22
22 7673 5 23
23 7673 6 1
24 8183 24 52
25 0 0 0
26 8183 5 35
27 0 0 0
28 8183 15 45
29 0 0 0
30 0 0 0
31 8183 38 1
32 8183 2 6
33 8183 26 43
34 8183 30 53
35 8183 6 38
36 8183 9 48
37 8183 28 4
38 8183 7 60
39 8183 19 56
40 38754 3 46
41 0 0 0
42 8183 13 50
43 8183 27 37
44 8183 12 42
45 8183 16 47
46 38754 4 8
47 8183 17 17
48 8183 10 64
49 0 0 0
50 8183 14 28
51 0 0 0
52 8183 25 33
53 8183 31 93
54 0 0 0
55 0 0 0
56 8183 20 58
57 8183 35 91
58 8183 21 98
59 0 0 0
60 8183 8 36
61 0 0 0
62 0 0 0
63 0 0 0
64 8183 11 44
65 0 0 0
66 8183 34 57
67 8183 4 26
68 17452 1 69
69 17452 2 70
70 17452 3 71
71 17452 4 72
72 17452 5 73
73 17452 6 74
74 17452 7 75
75 17452 8 76
76 17452 9 77
77 17452 10 78
78 17452 11 79
79 17452 12 80
80 17452 13 81
81 17452 14 82
82 17452 15 83
83 17452 16 84
84 17452 17 85
85 17452 18 86
86 17452 19 87
87 17452 20 88
88 17452 21 89
89 17452 22 90
90 17452 23 1
91 8183 36 99
92 8183 1 32
93 8183 32 96
94 38754 2 40
95 8183 23 24
96 8183 33 66
97 0 0 0
98 8183 22 95
99 8183 37 31
[ssdavis@lect1 p4]$
[ssdavis@lect1 p4]$ defrag.out Disk100_5_80_33_1.txt
CPU Time: 0 Disk accesses: 160 RAM: 966606
```