COMP 3500: Project Part 2 - Data Structures

Alex Lewin, Austin Newkirk, Charles Painter,
Chase Dumbacher, Daniel Corbett, Ethan Johnson, Payton Davis

3/3/20

---

1 Executive Summary

To properly implement all of the features in the project specifications, we decided to utilize a File Allocation Table (FAT) and a Free Space Table (FST). Both will be stored in the first block.

In designing our data structures, there are several trade-offs to consider. With our file system, we decided to optimize the performance of interacting with files, which sacrifices simplicity and some memory for files. Because of our design, nearly all operations (read, write, delete file, update FAT and FST) can be done in *constant time*. The only exception is creating a new file, which can be done in *linear time*.

---

2 File Allocation Table

2.1 Overview

The File Allocation Table (FAT), will be represented as an array of `Files`, which are defined below.

2.2 File Specifications

Each `file` will be represented by 128 Bytes in total. We need to know the file's name, size, and location. This will be represented by the following data.

- **fName** (up to 123 characters)
- **fSize** (Accommodates max file size of 1 full sector)
- **sectorNum** (Number 0-19, indicating which sector the file resides in)
- **offset** (Location relative to the start of the block, Accommodates any starting point within 1 sector)

- **pointer** (Location of pointer in open file, Set to all ones in closed file)

2.2.1 Detail Summary

Table 1: File fields

Field	Data Type	C++ Data Type	Size (in bytes)
fName	String	char[121]	121
fSize	integer	short	2
sector	integer	char	1
offset	integer	short	2
pointer	integer	short	2

2.2.2 Size Reasoning

For each field, we chose the smallest data size that would accommodate all the needed information. We also wanted the total space for one file to be 114 Bytes. This provides a maximum of 512 files to be stored in our FAT, still allowing to store given that our FAT occupies one complete sector.

- **fSize**: Minimum size to store max file size (65536 –> 2 Bytes)
- **sector**: Stores a number (0-19 –> 1 Byte)
- **offset**: Maximum value is equal to the size of a sector (65536 –> 2 Bytes)
- **pointer**: Maximum value is equal to the size of a sector (65536 –> 2 Bytes)
- **fName**: Left over memory so the total is equal to 128 Bytes ((128 - 7) –> 121 Bytes)

2.2.3 Name Reasoning

We chose the names because it clearly represents what those data entries are.

- **fName**: File Name

- **fSize**: File Size
- **sector**: Sector number
- **offset**: Number of bytes from start of block
- **pointer**: Location of pointer relative to beginning of the file, all ones indicates closed file

2.3 FAT Specifications

The File Allocation Table will be represented as an array of File structures. This will be represented by the following field.

- **Files**: Array of `File` structures

2.3.1 Detail Summary

Table 2: FAT fields

Field	Data Type	C++ Data Type	Size (in bytes)
Files	File Array	File[512]	65536

2.3.2 Size Reasoning

The array will contain 512 File structures, occupying a total of 58700 Bytes

- **Files**: 114 Bytes per File * 512 Files –> 58700 Bytes)

2.2.3 Name Reasoning

We chose the name Files because it clearly represents what those data entries are.

- **Files**: Array of Files

2.4 Code Representation

This code snippet represents all of the data types that are used in the FAT.

```
struct FAT {

  struct File {  // 114 Bytes
    fName char[107]; //107 Bytes
    fSize short;  //2 Bytes
    sector char;  //1 Bytes
    offset short;  //2 Bytes
    pointer short;  //2 Bytes
  }

  Files File[512]; //58700 Bytes
}
```

3 Free Space Table

3.1 Overview

To accomplish all file operations in constant time, with the exception of creating a new file, we decided to implement an FST.

The FST will track each **Free Section** (max 512 sections), which will allow a new file to be created in linear time with respect to the number free sections. A **Free Section** is a portion of a sector that is not being used to store a file. Because of how we will implement our functions in Part 3, there will never be two adjacent **Free Sections**.

The FST accomplishes a few things - it stores the location and size of a **Free Section**, and it maps files in the FAT to adjacent Free Sections. Storing the adjacent files will allow us to change the size of or delete a file in constant time.

3.2 Free Section Specifications

The FST will store a list of **Free Sections**. In order to accomplish defragmentation, we store the adjacent files to each **Free Section**. Each **Free Section** will contain the following.

- **sector**: Number 0-19, indicating which sector the file resides in
- **offset**: Location relative to the start of the block, Accommodates any starting point within 1 sector
- **size**: Size of the Section, max is equal to the size of a sector
- **fAbove**: index of file directly above
- **fBelow**: index of file directly below

3.2.1 Size Reasoning

For each field, we chose the smallest data size that would accommodate all the needed information. Each FreeSection object is represented by 9 Bytes.

- **sector**: Number 0-19 (1 Byte)
- **offset**: Max value is equal to the size of one sector (Max 65536 –> 2 Bytes)
- **size**: Max value is equal to the size of one sector (Max 65536 –> 2 Bytes)
- **fAbove**: Index of file directly above space (Max 512 –> 2 Bytes)
- **fBelow**: Index of file directly below space (Max 512 –> 2 Bytes)

3.2.2 Name Reasoning

We chose each name because it clearly represents what the data entry is.

- **sector**: Sector number
- **offset**: Number of bytes from start of block
- **size**: Size of space
- **fAbove**: Index of file directly above space
- **fBelow**: Index of file directly below space

3.2.3 Detail Summary

Table 3: FreeSection fields

Field	Data Type	C++ Data Type	Size (in bytes)
sector	integer	char	1
offset	integer	short	2
size	integer	short	2
fAbove	integer	short	2
fBelow	integer	short	2

3.3 FST Specifications

The FST will store an array of **Free Sections** and arrays that map files to adjacent Free Sections.

- **spaces**: Array of FreeSections
- **spaceAbove**: Maps index of files in FAT to the index of FreeSection directly above it
- **spaceBelow**: Maps index of files in FAT to the index of FreeSection directly below it

3.3.1 Size Reasoning

For each field, we chose the smallest data size that would accommodate all the needed information. Our FAT and FST will collectively occupy one entire sector. The entire FST will occupy 6836 bytes.

- **spaces**: 532 element * 9 Bytes per element (4788 Bytes)
- **spaceAbove**: 512 * 2 Bytes per element (1024)
- **spaceBelow**: 512 * 2 Bytes per element (1024)

3.2.2 Name Reasoning

We chose each name because it clearly represents what the data entries is.

- **spaces**: Array of FreeSections
- **spaceAbove**: Maps index of files in FAT to the index of FreeSection directly above it
- **spaceBelow**: Maps index of files in FAT to the index of FreeSection directly below it

3.3.3 Detail Summary

Table 4: FST Fields

Field	Data Type	C++ Data Type	Size (in bytes)
spaces	FreeSection array	FreeSection[532]	4788
spaceAbove	Integer array	short[512]	1024
spaceBelow	Integer array	short[512]	1024

3.4 Code Representation

This code snippet represents all of the data types that are used in the FST.

```
struct FST { // 6836 Bytes

  struct FreeSection{ // 9 Bytes
    sector char;  // 1 Byte
    offset short; // 2 Bytes
    size short; // 2 Bytes
    fAbove short; // 2 Bytes
    fBelow short; // 2 Bytes
  }

  spaces FreeSection[532]; //4788 Bytes


  // Maps File Index -> FreeSection Index
  // spaceAbove[ <File Index> ] = <FreeSection Index>
  // spaceBelow[ <File Index> ] = <FreeSection Index>

  spaceAbove short[512]; // 1024 Bytes
  spaceBelow short[512]; // 1024 Bytes

}
```