ESE 150 – Lab 10: File Systems (or Organizing Data)

LAB 10

Today's Lab has the following objectives:

1. Learn how to organize data for ease and optimize retrieval
2. Design an Application-Specific File Systems

Background:

FILE SYSTEMS

Data can be stored compactly and inexpensively on bulk storage devices like magnetic and solid-state "disk" drives. Between both the sheer capacity of data (Gigabytes to Terabytes) and the high ratio between capacity and bandwidth, it is necessary to carefully organize our data so that we can find data and access it quickly. This is the goal of a file system. The file system puts structure on top of the raw capacity so that we can find where data is physically stored. File system hierarchy and directories allow us to minimize the time required to navigate the data on the disk to find and retrieve our data.

ESE 150 – Lab 10: File Systems (or Organizing Data)

Prelab:

Assume:

- a 64GB solid-state drive with a read bandwidth of 100MB/s.
- the drive primarily holds MP3 files for songs; a typical song is 3 minutes long.
- the MP3 files are encoded at 128Kbits/s.

Questions:

1. How many songs can the drive hold?
2. How many hours of unique playtime?
3. How long does it take to read through the entire contents of the solid-state drive?
4. Would it be reasonable to perform this operation to list the contents of the drive?
5. About how many bytes would it take to store the title, artist, and album name for a song? [State necessary assumptions.] How does this compare to the length of the encoded song?
6. Assuming you stored this information (title, artist, song) along with an 8 Byte address pointer for every song in one large index file, how much data would this require? How long would it take to read this data from the disk?
7. Assuming the consumer keeps the device for 3 years and listens to music on average 2 hours per day every day of the year, how many times (on average) is each song played?
8. If you (the engineer designing the file system for this consumer component) have a choice of performing: (a) more work when the song is initially loaded onto the machine to accelerate consumer selection or (b) minimal work on load but more work when the consumer tries to find the song to play it, what does your answer to 7 suggest about making this choice?

Read through entire lab.

Lab Procedure:

Goal: Design and provide psuedocode implementation for a file system that allows a user to store MP3 songs and quickly select them by album or artist.

We would like to be able to browse and select music by:

1. browse artists, then browse albums by that artists, then by title within the album (for the selected artists)
2. browse all albums, then all titles on the selected album (for multi-artist albums, the title list for case one will be different from the list in case two)

You may assume:
• album names are unique.
• songs are assigned to a single artist.

Your solution should be economical in song storage space. In particular, it should store each song only once. It may store short data about songs (like title, artist, album, pointer to song) multiple times. Redundant data in directory nodes is acceptable.

1. Show how your solution would represent the sample set of songs that follows (This is a figure like the one on page 9. You may use a spreadsheet or a drawing program in any software.)
2. Show the trace of operations performed when you add a song with a new album and artists. E.g. show what happens when you add the song "Fight Song" by Rachel Platten on the album *Wildfire* to set of songs represented in the previous question.
3. Document your design.

Single Hierarchy Design

- As a starter and example, we detail an implementation that only handles the first case above. After understanding this example, you will need to modify it to handle both selection by artist and selection by album.

Overview

Our solution is simply to represent the necessary structure directly as a directory hierarchy. At the top level, we have a directory of artists. In each artist directory, we place a directory of albums by that artist. In each of these album directories, we finally have a directory of the actual songs that the specified artist has on the album. When a new song is added, we create directory nodes as needed for artist and album, then link the song from the album directory.

Implementation

We need two operations: one to allow the user to navigate to a song, and one to allow the user to add a new song.

- songBNode ← Selector() //Select a song by choosing artist then album and then title

```
//Choose artist
artistList ← GetKeyStringList(rootArtistBNode)
selectedArtist ← DisplayChooseString(artistList)
artistBNode ← GetValueForKey(rootArtistBNode, selectedArtist)
//Choose album for chosen artist
albumList ← GetKeyStringList(artistBNode)
selectedAlbum ← DisplayChooseString(albumList)
albumBNode ← GetValueForKey(artistBNode, selectedAlbum)
//Choose song for chosen album
titleList ← GetKeyStringList(albumBNode)
selectedTitle ← DisplayChooseString(titleList)
songBNode ← GetValueForKey(albumBNode, selectedTitle)
```

- AddSong(artist, album, title, songBytes[])

```
//Create BNodes for artist, album and title
//if necessary and store song in title BNode
artistBNode ← GetValueForKey(rootArtistBNode, artist)//get artist BNode
if (NotPresent(artistBNode))//check if artistBNode exists
{
        //artistBNode does not exist,
        //create it and add it to rootArtistBNode
        artistBNode ← NewDirectoryBnode()
        InsertInOrder(rootArtistBNode, artist, artistBNode)
```

```
}

albumBNode ← GetValueForKey(artistBNode, album) //get album BNode
if (NotPresent(albumBNode))//check if albumBNode exists
{
        //albumBNode does not exist,
        //create it and add it to artistBNode
        albumBNode ← NewDirectoryBnode()
        InsertInOrder(artistBNode, album, albumBNode)
}
titleBNode ← GetValueForKey(albumBNode, title)//get title BNode
if (NotPresent(titleBNode))//check if titleBNode exists
{
        //titleBNode does not exist, create it
        titleBNode ← NewDataBnode(songBytes.length, mp3)
        InsertInOrder(albumBNode, title, titleBNode)
}
StoreIntoBNode(titleBNode, songBytes)//store songBytes in titleBNode
```

Low Level Support Routines

We assume the following low-level support routines exist.

- stringList ← GetKeyStringList(directoryBNode) Given a directoryBNode, returns a list of items in the directory.
- string ← DisplayChooseString(stringList) Displays a stringList and returns the chosen item.
- bNode ← GetValueForKey(directoryBNode, string) Given a directoryBNode and a string corresponding to an item in the directory, returns the BNode for that item in that directory. If the string does not exist, returns a designated bNode token indicating the non-presence of the string.
- directoryBNode ← NewDirectoryBnode() Creates a new BNode of type directory and returns the created BNode.
- InsertInOrder(directoryBNode, string, bNode) Inserts into directoryBNode the item called string with corresponding bNode. Maintains alphabetical order within directoryBNode.
- bNode ← NewDataBnode(length, type) Creates a new data BNode of length bytes and of type type.
- StoreIntoBNode(bNode, bytes[]) Stores the array bytes bytes[] representing data into bNode.

Note:

1. You may assume that functions InsertInOrder and StoreIntoBNode internally handle creating or expanding trees of BNodes to accommodate the data.
2. You may also assume that rootAlbumBNode (like rootArtistBNode) exists.
3. Also, note that the left arrow '←' in the pseudocode means "assignment", and is equivalent to '=' in most programming languages you've used.
4. Each song should only be stored once.
5. It's okay to have duplicate directory nodes.

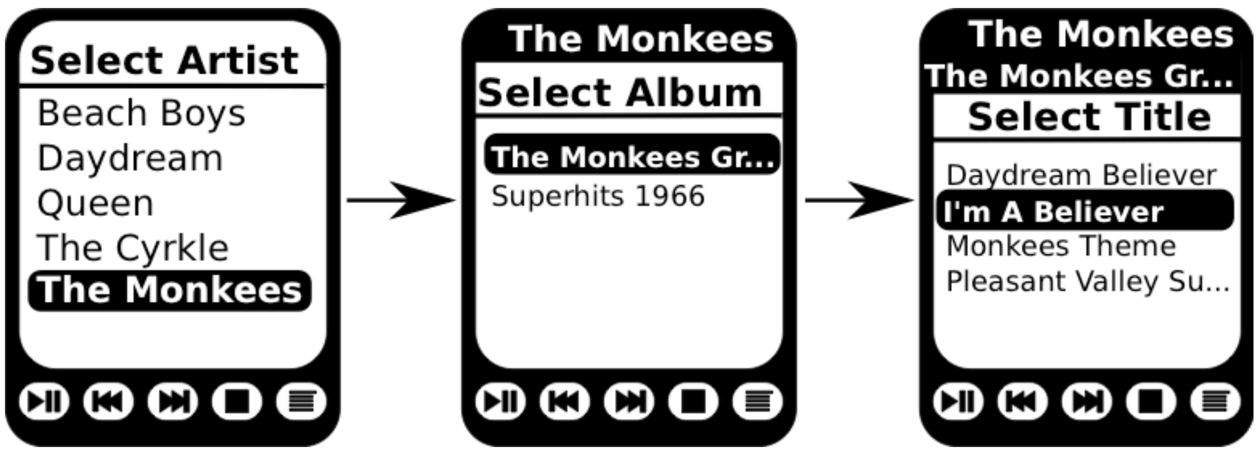
Now, modify and add to this example to write your own implementation. Talk with your TA if you find there is a function you need that is missing.

Sample Set of Songs

Artist	**Album**	**Title**
Queen	Queen's Greatest hits	Bohemian Rhapsody
Queen	Queen's Greatest hits	We Will Rock You
Queen	Queen's Greatest hits	We Are The Champions
Queen	Queen's Greatest hits	A Kind of Magic
The Cyrkle	Superhits 1966	Red Rubber Ball
The Monkees	Superhits 1966	Last Train to Clarksville
Beach Boys	Superhits 1966	Sloop John B
Daydream	Superhits 1966	Lovin' Spoonful
The Monkees	The Monkees Greatest hits	Monkees Theme
The Monkees	The Monkees Greatest hits	I'm A Believer
The Monkees	The Monkees Greatest hits	Daydream Believer
The Monkees	The Monkees Greatest hits	Pleasant Valley Sunday
Beach Boys	Pet Sounds	Wouldn't It Be Nice
Beach Boys	Pet Sounds	God Only Knows
Beach Boys	Pet Sounds	I Know There's An Answer
Beach Boys	Pet Sounds	Pet Sounds

Interactions

Sample Selection by Artist



Sample Selection by Album

Single Hierarchy Design Representation for Sample Set of Songs

Root	Type:Dir
Name	**Node**
Beach Boys	2
Daydream	3
Queen	4
The Cyrkle	5
The Monkees	6

2	Type:Dir
Name	**Node**
Pet Sounds	7
Superhits 1966	8

3	Type:Dir
Name	**Node**
Superhits 1966	9

4	Type:Dir
Name	**Node**
Queen's Greatest Hits	10

5	Type:Dir
Name	**Node**
Superhits 1966	11

6	Type:Dir
Name	**Node**
Superhits 1966	12
The Monkees Greatest Hits	13

7	Type:Dir
Name	**Node**
God Only Knows	14
I Know There's An Answer	15
Pet Sounds	16
Wouldn't It Be Nice	17

8	Type:Dir
Name	**Node**
Sloop John B	18

9	Type:Dir
Name	**Node**
Lovin' Spoonful	19

10	Type:Dir
Name	**Node**
A Kind of Magic	20
Bohemian Rhapsody	21
We Are The Champions	22
We Will Rock You	23

11	Type:Dir
Name	**Node**
Red Rubber Ball	24

12	Type:Dir
Name	**Node**
Last Train to Clarksville	25

13	Type:Dir
Name	**Node**
Daydream Believer	26
I'm A Believer	27
Monkees Theme	28
Pleasant Valley Sunday	29

14	Type:MP3		
Data Blocks			
30	31	...	37

15	Type:MP3		
Data Blocks			
38	39	...	45

16	Type:MP3		
Data Blocks			
46	47	...	56

17	Type:MP3		
Data Blocks			
57	58	...	69

18	Type:MP3		
Data Blocks			
70	71	...	77

19	Type:MP3		
Data Blocks			
78	79	...	82

20	Type:MP3		
Data Blocks			
83	84	...	86

21	Type:MP3		
Data Blocks			
87	88	...	92

22	Type:MP3		
Data Blocks			
93	94	...	99

23	Type:MP3		
Data Blocks			
100	101	...	106

24	Type:MP3		
Data Blocks			
107	108	...	112

25	Type:MP3		
Data Blocks			
113	114	...	117

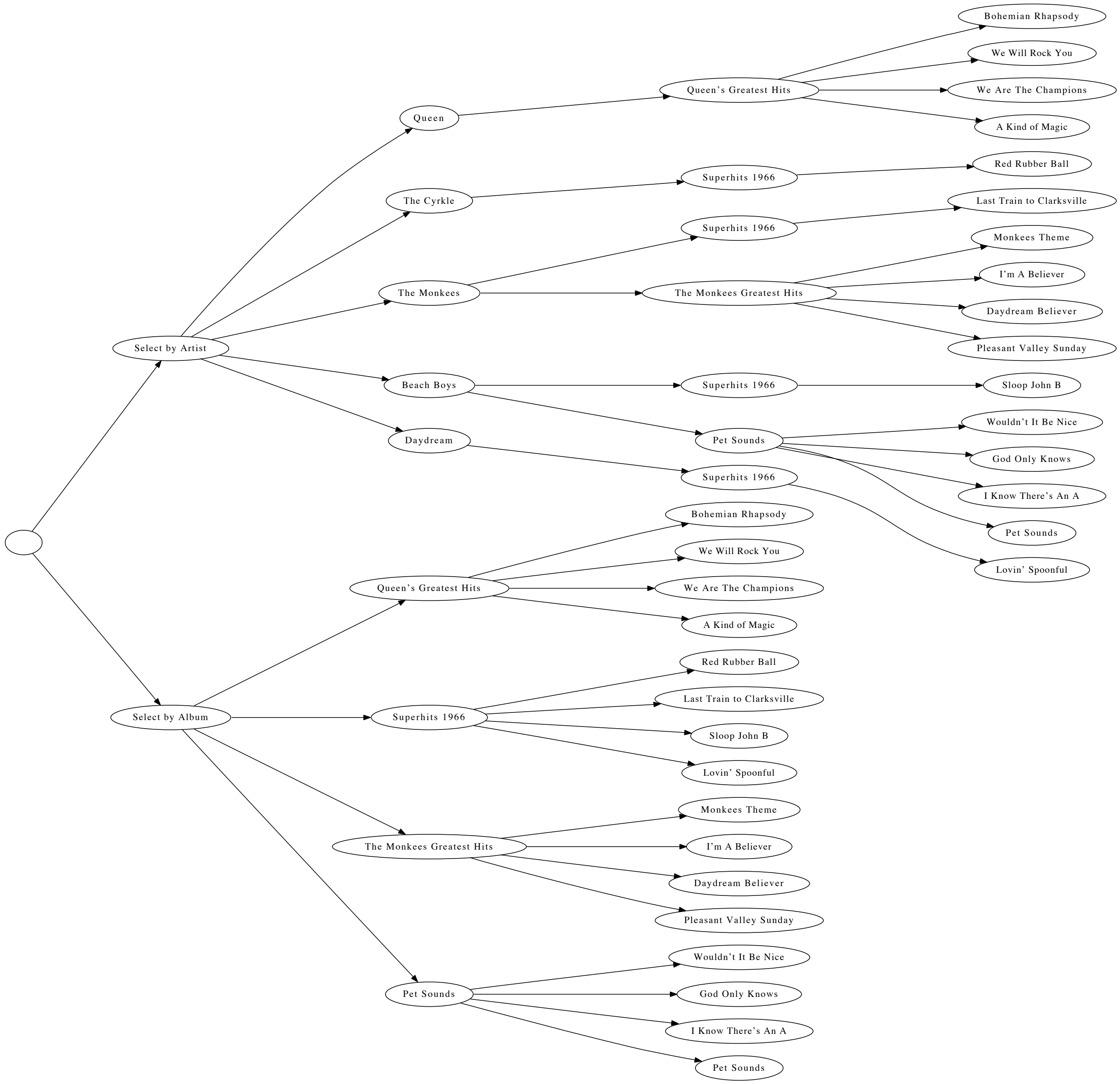
26	Type:MP3		
Data Blocks			
118	119	...	123

27	Type:MP3		
Data Blocks			
124	125	...	130

28	Type:MP3		
Data Blocks			
131	132	...	142

29	Type:MP3		
Data Blocks			
143	144	...	158

Full Interaction for Two View Case for Sample Set of Songs



Postlab

1. Assuming the minimum size of each BNode is 1KB, how much space does your solution add per song (For a song, make the same assumption as in prelab)?
   a. Absolute space in Bytes
   b. Relative space as a percentage to the space required for the song
2. Identify three other ways you might like to organize your music (beyond the by-artist and by-album that you designed for in this lab). For each:
   a. Describe the classification (one sentence)
   b. Describe how to extend your solution to handle this additional classification in one to two sentences.
3. The Internet is estimated to contain over a Zettabyte (10^{21} Bytes or 2^{70} Bytes) and we are adding 10^{18} Bytes per day (it's really an exponentially growing number, but even a constant rate is enough to illustrate the challenge). Extend the reasoning from the prelab to outline the importance of indexing and organization to make it possible to find information on the Internet. Assume your computer has a 100 MB/s connection to the Internet.

HOW TO TURN IN THE LAB

- Submit a PDF document to the designated canvas assignment containing:
  - Answers to prelab
  - Your design solution including
    - Overview description of the design
    - Implementation in psuedocode
    - Diagrams to show how your solution implements the sample set of songs
  - Answers to postlab