Journal of The Korea Society of Computer and Information
Vol. 22 No. 8, pp. 73-83, August 2017
www.ksci.re.kr
https://doi.org/10.9708/jksci.2017.22.08.073

Ordinary B-tree vs NTFS B-tree: A Digital Forensics Perspectives

Gyu-Sang Cho*

Abstract

In this paper, we discuss the differences between an ordinary B-tree and B-tree implemented by NTFS. There are lots of distinctions between the two B-tree, if not understand the distinctions fully, it is difficult to utilize and analyze artifacts of NTFS. Not much, actually, is known about the implementation of NTFS, especially B-tree index for directory management. Several items of B-tree features are performed that includes a node size, minimum number of children, root node without children, type of key, key sorting, type of pointer to child node, expansion and reduction of node, return of node. Furthermore, it is emphasized the fact that NTFS use B-tree structure not B+ structure clearly.

▸Keyword: B-tree, B+tree, NTFS filesystem, Directory index, Digital forensics

I. Introduction

NTFS는 Windows NT3.1에서 1.0버전이 발표된 이래로 Windows의 기본 파일시스템으로 현재 3.1버전에 이르고 있다. 개발 초기부터 적용된 기술과 기본사양을 바탕으로 최신의 Windows 10에 이르기까지 오랜 기간 동안 기본 형태의 큰 변화 없이 안정된 파일시스템으로 잘 사용되고 있다[16].

모바일기기나 PC등에서 디지털 포렌식의 관점에서 디스크에 중요 정보가 저장된 디스크에 대한 증거 수집을 위한 파일시스템 분석, 즉 컴퓨터의 사용로그, 인터넷 접속과 검색에 대한 로그, 파일삭제/복구 등에 관한 분석이 빈번하게 수행되고 있고 이것에 관한 문헌[6, 17]과 인터넷 블로그[7, 8, 9]의 자료들이 많이 제공되고 있다.

Windows NTFS와 관련된 정보들은 개념적인 원리, 아키텍처, 파일과 디렉토리의 데이터 구조 등에 대해서 여러 문헌들[16]과 관련 사이트들에서 잘 기술되어 있다[14,15,16]. 그리고 NTFS에 관련된 디지털 포렌식 방법에 대해서도 잘 기술되어 있다[16].

그러나 디렉토리 인덱스에 관련된 내용들에 관해서는 데이터 구조를 잘 소개하고 있지만 디렉토리 목록을 관리하기 위해서 사용되고 있는 B-트리의 동작이나 구현 방법에 대해서는 내용을 찾아보기 힘들다.

최근에 Wicher Minnaard[17]의 NTFS의 타임스탬프 변조 문제를 다룬 연구에서 디렉토리 인덱스에 대한 분석을 수행한 것을 찾을 수 있다. 그리고 W. A. Bhat와 S. M. K. Quadri의 연구[18]에서도 디렉토리 인덱스에 대한 언급을 하고 있고 Petra Grd와 Miroslav Bača의 연구[19]에서는 디렉토리에서 파일을 삭제한 후에 인덱스 레코드에 남는 흔적에 대한 디지털 포렌식 소개하고 있으나 두 연구 모두 피상적인 수준에서 디렉토리 인덱스에 대한 설명에 그치고 있다.

적극적인 디렉토리 인덱스에 관한 연구는 Cho의 연구[9]에서 찾아볼 수 있다. 디렉토리 인덱스에 관한 디지털 포렌식 분석에서 디렉토리의 목록의 수가 늘어남에 따라 변화되는 B-트리 인덱스 구조에 대해서 기술하고 있다. 여기서는 NTFS에서의 B-트리의 동작 방식을 잘 설명하고 있다. 그의 다른 연구[10]에서는 NTFS에서 디렉토리 목록의 변화가 생길 때마다 인덱스 레코드의 슬랙 영역에 남는 파일명의 흔적들을 이용하여 데이터를 숨기는 방법을 연구하였다. 이 방법을 이용한 후속연구[11]에서 파일명에 사용할 수 없는 문제의 제약을 없애고 암호화 효과를 얻기 위한 유니코드 변환 방법을 제안하였다. 그 다음 연구[12]에서는 파일에 대한 복사나 삭제 명령 등을 수행하였을 때 타임스탬프의 변화가 디렉토리 인덱스의 관련 정보에의 변화가

• Corresponding Author: Gyu-Sang Cho, Corresponding Author: Gyu-Sang Cho
*Gyu-Sang Cho (cho@dyu.ac.kr), School of Public Technology Service, Dongyang University
• Received: 2017. 08. 03, Revised: 2017. 08. 08, Accepted: 2017. 08. 14.
• This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2016R1D1A1B03935646)

생기는 부분에 대한 분석을 수행하였다. 디렉토리 내에서 어떤 작업이 언제 실행되었는지 추정할 수 있는 방법이다.

디렉토리 인덱스에 대한 정보들은 연구문헌 보다는 컴퓨터 포렌식 관련 블로그들에서 많이 찾을 수 있다. W. Ballenthin[6]은 파이썬 프로그램으로 제작된 INDXParser.py를 블로그에서 공개하고 있다. 이 프로그램은 디렉토리 내의 파일이름, 파일크기, 시간정보 등으로 디렉토리의 목록에 대해서 분해하고 슬랙 영역에서 정보를 구하는 기능을 갖고 있다. C. Tilbury의 SANS DFIR 블로그[7]에서는 NTFS 파일시스템의 할당 인덱스 영역 안에 남은 인덱스의 엔트리들의 흔적을 통해서 어떤 파일들에 사용되었는지를 판단할 수 있는 분석방법을 소개하고 있다. W. Ballenthin 등[8]은 NTFS의 인덱스 구조에서 구성요소별로 데이터를 얻기 위한 도구들의 사용방법, 인덱스 구조에 대한 분석 등의 정보를 제공하였다.

이 연구에서는 Windows의 NTFS 파일시스템에서 사용되는 B-트리의 구현된 방법을 분석하여 원래의 B-트리와 어떤 차이점이 존재하는지 분석하는 작업을 수행한다. 2장에서 원래 B-트리의 기본요소와 정의에 대해서 언급하고 3장에서 NTFS에서 사용하는 디렉토리 인덱스에 필요한 데이터 구조를 소개한다. 4장에서는 원래 B-트리와 NTFS B-트리의 차이점을 항목별로 대조하고 5장에서는 해당 내용을 디렉토리 인덱스의 실제 사례를 통해 분석하기로 한다. 6장에서는 이 연구에서 얻어진 결과의 의미와 추후 연구에 대해서 논하며 결론을 마무리한다.

An Ordinary B-tree

1. Basic Idea of B-tree

NTFS의 B-트리구조는 인덱스를 빠르게 검색할 수 있다. 키 값의 갯수가 증가하여도 깊이 수준(depth level)많이 증가하지 않는 균형트리(balanced tree)이다[3]. 노드는 자식 노드의 포인터와 키 값을 갖고 있는데 키 값을 기준으로 작은 값은 왼쪽 아래, 큰 값은 오른쪽 아래에 자식 노드를 가진다. 그림 1은 차수 m=3인 B-tree의 예를 나타낸 것이다.

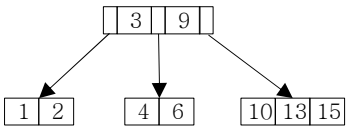

Fig. 1 Example of a B-tree

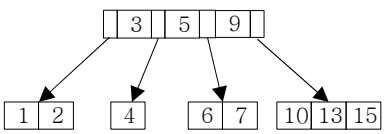

Fig. 2 Example of a B-tree Expansion

그림 1의 루트 노드는 3개의 자식 노드를 갖고 있다. 루트의 {3}키의 왼쪽 자식, {3}과 {9}키 사이의 가운데 자식, {9}키 오른쪽 자식을 갖는다. 즉, 3개의 서브트리를 갖는다. 가장 왼쪽 서브트리의 값들은 {3}키 값보다 작은 값들로 구성되고, {3}과 {9}키 사이의 서브트리는 3<Ksub<9 로 3보다 크고 9보다 작은 값을 갖는다. 가장 오른쪽의 서브트리의 킷값들은 {9}보다 큰 값으로 구성된다.

자식 노드들은 리프 노드이다. 리프의 특성상 자식노드에 대한 포인터를 갖지 않는다. 트리의 깊이가 2이기 때문에 중간노드는 갖고 있지 않은 상태이다. 각 리프노드에는 {1,2}, {4,6}, {10,13,15}의 값들을 갖는다.

그림 1에서 {5}, {7}을 삽입하였을 때 가운데 리프노드가 차수 m=3보다 많은 항목이 입력되어서 리프노드가 { 4, 5, 6, 7 }을 갖게 되면서 중간값인 {5}는 부모노드로 올라가고 좌측에는 {4}, 우측에는 {6,7}을 갖는다(그림 2).

2. Definition of B-tree

B-tree의 기본 요소들은 다음과 같다. 차수가 m인 B-tree는 다음의 특성을 만족해야 한다[2,13].

1) 모든 노드는 최대한 m개까지의 자식 노드를 갖는다.

2) 모든 리프가 아닌 노드는(루트 노드 제외)는 최소한 ⌈m/2⌉의 자식을 갖는다.

3) 루트는 리프노드가 아니라면 최소한 두 개의 자식 노드를 갖는다.

4) 리프가 아닌 노드가 k개의 자식을 가졌다면 키의 수는 k-1개이다.

5) 모든 리프들은 같은 깊이를 유지해야 한다.

다음은 내부노드, 루트노드, 그리고 리프노드에 대한 설명이다.

- 내부노드

루트나 리프가 아닌 노드를 내부 노드라고 한다. 모든 내부노드는 최소 L개, 최대 U개의 자식노드를 갖는다. 그러므로 내부노드의 포인터는 최소 L-1, 최대 U-1개를 갖게 된다. U는 항상 2L 또는 2L-1개 값을 갖는다. 그러므로 내부노드는 항상 반 이상 채워진 상태를 유지하게 된다. 꽉 채워진 노드에 한 개의 값이 추가될 때 노드가 두 개로 분리된다. 이 때 정렬 상태에서 중간의 키는 상위 노드로 보내고 나머지 키들은 중간 값보다 작은 값들과 중간값 보다 큰 값들로 두 개의 노드로 분리된다. 그러므로 분리된 중간 노드는 항상 반 이상 채워진 상태를 갖게 된다[2,13].

- 루트 노드

루트노드의 자식 노드는 내부노드와 마찬가지로 최대 U개를 갖는다. 그러나 최소수(하한)의 자식노드의 수에 대한 제약은 없다. 루트 노드의 경우는 자식 노드를 갖지 않을 수도 있다. 그러나 자식 노드를 가질 때는 최소한 두 개를 갖는다. 한 개만

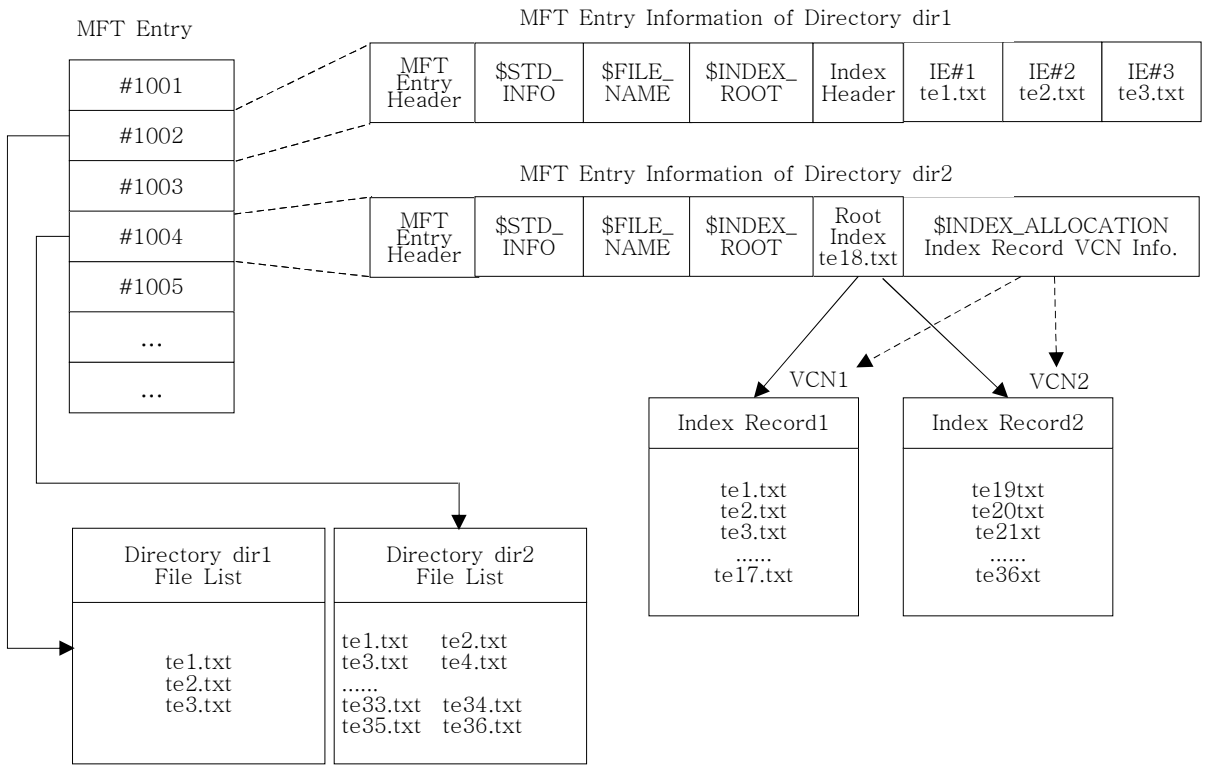

Fig. 3 Schematic Diagram of NTFS MFT entry and Index Record(Resident and Non-resident Cases)

가질 수는 없다[2,13].

-리프노드

리프노드는 다른 노드와 마찬가지로 최대 U개를 갖는다. 리프노드는 최하위 레벨에 존재하는 특성으로 인해서 더 이상의 자식노드를 갖지 않는다[2,13].

다른 균형트리들과 마찬가지로 B-트리는 항목의 수에 비하여 레벨의 높이가 높지 않은 편이다. 그러므로 삽입, 삭제, 검색에 대한 연산의 비용이 로그적으로 완만하게 증가하는 특징을 갖는다[2,13].

III. Directory Index of NTFS

1. Basic Idea of NTFS

NTFS에서는 디렉토리(폴더: Windows의 파일탐색기에서 사용하는 용어는 폴더이다. 즉, 폴더와 디렉토리는 동일한 의미로 사용된다)의 파일목록을 관리하기 위한 수단으로 B-트리 방식의 데이터 구조를 사용하고 있다. 디렉토리마다 서로 독립적인 B-트리 구조를 갖는다.

디렉토리의 목록을 관리하기 위하여 B-tree를 적용할 때 기본적인 동작방식은 전통적인 B-tree방식을 따르고 있지만 파일시스템으로 구현되면서 변형적으로 적용된 부분이 존재한다. 루트노드의 경우가 대표적이다. 디렉토리 인덱스의 루트 노드 역할은 MFT 엔트리에 들어있는 상주(Resident) 속성인 $INDEX_ROOT가 역할을 하고 있다. 비상주(Non-resident)속성으로 $INDEX_ALLOCATION에 런-리스트(Run-list 또는 Run-length) 정보에 의해 하위노드들이 유지된다[9,16,19].

이론적으로 B-tree는 루트노드 내에는 반드시 최소 한 개 키가 존재한다. 그리고 두 개의 하위 노드를 가져야 하지만 NTFS에서는 루트 노드인 $INDEX_ROOT속성만으로 노드 한 개가 사용된다. 이것은 MFT 엔트리의 내의 상주 속성이기 때문에 목록을 저장할 수 있는 공간이 한정적이다. 이 공간의 크기는 600여 바이트정도가 가능하다. 파일명 길이에 관련되지만 8.3형식을 사용하는 경우의 파일명의 경우는 5~6개 정도 이 공간에 기록될 수 있다[9,16,19]. (그림 3의 #1002 dir1 디렉토리의 경우에 해당)

디렉토리 내의 목록이 늘어나면 새로운 노드를 만들어 공간을 확장한다. 이 때 4KB 크기의 노드에는 여러 개의 디렉토리 목록들이 소팅된 상태로 저장되어 있어서 빠른 검색이 가능하다(그림 3의 #1004 dir2 디렉토리의 경우에 해당).

디렉토리의 목록들이 많아지면 여러 개의 노드들로 확장하게 된다. 지속적으로 확장하게 되면 2층, 3층으로 트리가 높아지게 된다. 중간 노드들에 파일명들이 키의 역할을 하며 하위노드를 가리키는 VCN 정보를 갖는다. 모든 노드들은 안에 들어있는 인덱스 엔트리의 숫자와 상관없이 모두 4KB의 고정 크기를 갖는다. 노드(인덱스 레코드)안에서 파일 목록들이 생성되고

삭제되면서 디지털 포렌식을 위한 유용한 정보가 남게 된다.

2. Structure of $INDEX_ROOT Attribute

모든 디렉토리는 파일들과 동일한 구조의 MFT엔트리를 갖는다. 차이점이 있다면 디렉토리의 경우는 이 안에 $INDEX_ROOT속성과 $INDEX_ALLOCATION속성이 들어있다는 점이다. MFT 엔트리에서 상주 속성으로 사용되는 $INDEX_ROOT속성(0x90)은 디렉토리 인덱스의 루트 노드 역할을 한다. B-트리 데이터 구조를 사용하여 파일들의 목록을 저장하고 관리하는 기능을 수행한다. MFT 엔트리의 저장공간이 1KB로 한정되어 있기 때문에 $STANDARD_INFORMATION과 $FILE_NAME속성 등의 필수 속성들이 저장되고 남은 공간에 $INDEX_ROOT속성 저장될 수 있다. 대략 600여 바이트 정도 사용될 수 있다. 이 공간에 저장 가능한 인덱스 엔트리의 수(파일 수)는 파일명의 길이가 가변이기 때문에 8.3포맷을 이용하는 경우를 가정한다면 5~6개 정도이다[13,16].

Table 1 Data Structure of $INDEX_ROOT Attribute[16]

Offset (byte)	Size (byte)	Descriptions
0-3	4	Attribute Type ID(0x90)
4-7	4	Length of Attribute
8-8	1	Type of Attribute
9-9	1	Length of Attribute Name
10-11	2	Offset to Attribute Name
12-13	2	Flag
14-15	2	Attribute ID
16-19	4	Size of Contents
20-23	4	Offset to Contents
24-31	8	Name of Attribute($I30)
32-35	4	Attribute Type in Index(0x30, $FILE_NAME Attribute)
36-39	4	Collation Sorting Rule (01=Filename)
40-43	4	Index Record Size(byte)
44-47	4	Index Record Size(cluster)
48-51	4	Offset to Index Entry from Start
52-55	4	Offset to End of Used List of Index Entry
56-59	4	Offset to End of Allocated Index Entry
60-63	4	Flag (0:Chile Exists,1:Last Entry)
64+	Variable	Index Entries

표 1은 $INDEX_ROOT속성의 데이터 구조를 나타낸 것이다. 이 속성은 세 부분으로 구성되어있는데 가장 앞쪽에 상주 속성헤더(0~31바이트)가 있고, 그 뒤에 $INDEX_ROOT 속성헤더(32~47바이트)가 들어 있다. 인덱스 노드 헤더(48~63바이트)가 그 뒤에 온다. 64바이트 위치부터는 인덱스 엔트리의 리스트(파일목록)들이 들어간다. 파일목록이 담긴 인덱스 엔트리는 이 모든 헤더구조가 구성된 뒤에 온다. 첫 인덱스 엔트리는 선두에서 64바이트 오프셋 위치에 저장된다.

3. Structure of $INDEX_ALLOCATION Attribute

MFT엔트리에 들어 있는 $INDEX_ROOT속성의 저장 공간이 부족해지면 인덱스 엔트리들을 비상주(non-resident)형식인 $INDEX_ALLOCATION 속성(0xA0)으로 저장하게 된다. $INDEX_ALLOCATION 속성의 기본정보는 MFT엔트리 내에 들어있지만 인덱스 엔트리가 저장된 인덱스 노드(또는 인덱스 레코드)는 MFT엔트리 밖에 저장하게 된다. 이 공간의 크기는 4KB이다. 표 2는 인덱스 레코드의 구조를 나타낸 것이다.

Table 2 Data Structure of $INDEX_ALLOCATION Attribute[16]

Offset (byte)	Size (byte)	Descriptions
0-3	4	Attribute TypeID(0xA0)
4-7	4	Length of Attribute
8-8	1	Type of Attribute
9-9	1	Length of Attribute Name
10-11	2	Offset to Attribute Name
12-13	2	Flag
14-15	2	Attribute ID
16-23	8	First VCN in Run-list
24-31	8	Last VCN in Run-list
32-33	2	Offset to Run-list
34-35	2	Compression Size
36-39	4	Unused
40-47	8	Allocated Size of Attribute Contents
48-55	8	Real Size of Attribute Contents
56-63	8	Initial Size of Attribute Contents
64-71	8	Attribute Name($I30)
72+	Variable	Run-list

$INDEX_ALLOCATION속성에는 여러 개의 인덱스 레코드에 대한 정보를 가질 수 있다. 디렉토리 안에 들어있는 파일/디렉토리의 목록의 수에 따라 이 숫자가 결정된다. B-트리의 루트 역할을 하는 $INDEX_ROOT속성에는 자식노드의 정보를 가진다. 한 개의 자식노드는 한 개의 4KB 크기의 인덱스 레코드가 된다. 자식에 대한 정보는 VCN(virtual cluster number) 값을 사용한다. 관련된 클러스터들의 일련번호이다. 해당하는 자식노드의 VCN번호를 키값 뒤에 저장하는 방식을 사용한다.

4. Structure of Index Record

MFT 엔트리에 들어 있는 $INDEX_ALLOCATION 속성에는 인덱스 레코드가 들어있는 런-리스트 정보가 들어 있다. 이것은 전체 B-트리를 유지하는데 사용하는 중요한 정보이다. 이 정보를 이용해서 인덱스 레코드가 저장되어있는 곳의 절대주소(LCN)을 알 수 있다.

인덱스 레코드는 B-트리의 노드역할을 한다. 이곳에는 인덱스 엔트리들이 저장된다. 즉, 디렉토리 안에 들어 있는 파일의 목록들이 저장된다. 인덱스 엔트리들은 언제나 정렬된 형태로 저장되어 목록을 검색하기 쉽다.

선두부터 인덱스 레코드 헤더(0~23바이트)와 인덱스 노드헤더로 구성되어 있다(24~47바이트). 이후에 인덱스 엔트리(64바이트)가 온다. 40바이트부터 63바이트까지는 픽스업어레이(fixup array)가 저장된다. 각 인덱스 레코드는 인덱스 노드헤더를 가진다. 인덱스 레코드의 크기는 4KB이다.

Table 3 Data Structure of Index Record[16]

Offset (byte)	Size (byte)	Descriptions
0-3	4	"INDX" Signature
4-5	2	Offset to Fixup Array
6-7	2	Entry Number of Fixup Array
8-15	8	$LogFile Sequence Number (LSN)
16-23	8	VCN of This Record
24-27	4	Offset to Start Position of Index Entry
28-31	4	Offset to End of Used List of Index Entry
32-35	4	Offset to End of Allocated Index Entry
36-39	4	Flags
40-55	24	Fixup Array
56~4095	Variable	Index Entries

5. Structure of Index Entry

인덱스 엔트리는 디렉토리에 들어있는 파일과 디렉토리의 목록을 구성하는데 사용하는 정보이다. 이 안에는 파일명, MFT 파일참조번호, 엔트리의 길이정보, $FILE_NAME의 속성 길이 정보, 플래그, $FILE_NAME 속성이 들어 있다(표 4). 여기서 $FILE_NAME 속성은 MFT 엔트리에 들어있는 $FILE_NAME 속성과 동일한 것이다(표 5).

표4는 인덱스 엔트리의 데이터 구조를 나타낸 것이다. 인덱스 엔트리의 헤더 정보는 0~15까지의 16바이트가 저장된다. MFT 파일참조번호에 8바이트, 엔트리 길이 정보 2바이트, $FILE_NAME 속성 길이 정보에 2바이트, 플래그에 4바이트가 할당된다. $FILE_NAME 속성은 그 후에 16바이트 떨어진 위치에 저장된다. 이 속성의 요소 중에서 파일명은 가변적이므로 모든 인덱스 엔트리마다 서로 다른 길이 값을 갖게 된다. 만일에 파일명이 8.3포맷인 경우라면 한 개의 인덱스 엔트리는 112바이트 크기를 갖게 된다. 인덱스 엔트리가 키 인덱스인 경우에는 $FILE_NAME 끝에 자식노드의 VCN 정보를 한 개 더 갖게 된다. 자식노드가 없는 경우라면 VCN 정보가 들어 있지 않게 된다.

Table 4 Data Structure of Index Entry[16]

Offset (byte)	Size (byte)	Descriptions
0-7	8	MFT Reference of This File Name
8-9	2	Length of Entry
10-11	2	Length of $FILE_NAME Attribute
12-15	4	Flags(1:Chile Exists, 2:Last Entry)
16+	Variable	$FILE_NAME Attribute
Last 8Bytes	8	(If Flags=1) VCN of Child Node

6. Structure of $FILE_NAME Attribute

$FILE_NAME 속성의 구조에서 8바이트 크기의 부모디렉토리의 파일참조번호가 제일 먼저 등장한다. 그 후에 4가지 타임스탬프가 저장된다. 생성시간, 수정시간, MFT수정시간, 접근시간 순으로 저장된다. 이 4가지 타임스탬프의 데이터 크기는 각기 8바이트이다. 파일에 할당크기정보는 40바이트 위치에 기록된다. 실제파일 크기 정보가 그 뒤에 오는데 크기는 8바이트이다. 4바이트 크기의 플래그 값과 4바이트 크기의 리파스(reparse)값이 저장된다. 60바이트 위치에 1바이트 크기의 파일명의 길이 값, 1바이트 크기의 파일명 스페이스(name space)값, 그 후에 유니코드 형식의 파일명이 저장된다.

Table 5 Data Structure of $FILE_NAME Attribute[16]

Offset (byte)	Size (byte)	Descriptions
0-7	8	File Reference of Parent Node
8-15	8	File Creation Time
16-23	8	File Modification Time
24-31	8	MFT Modification Time
32-39	8	File Access Time
40-47	8	File Allocation Size
48-55	8	Real File Size
56-59	4	Flag
60-63	4	Reparse
64-64	1	Length of File Name
65-65	1	Namespace
66 +	Variable	File Name

IV. Ordinary B-tree vs NTFS B-tree

1. Comparison of Ordinary B-tree and NTFS B-tree

1.1 Number of item in a node

일반 B-tree : m개(고정된 개수)

NTFS B-tree : 가변 개수-파일명의 길이가 255개까지 유니코드 문자가 사용되는 가변길이 방식이므로 한 개의 인덱스 엔트리의 크기는 파일명에 영향을 받는다. 그러므로 한 노드에 들어가는 항목의 수는 가변적이다. 최소 유니코드 4글자 파일명의 경우는 45개, 255자 파일명의 경우는 6개 저장할 수 있는 공간이다.

1.2 Node size

일반 B-tree : 응용하는 경우에 따라 결정할 수 있다.

NTFS B-tree : NTFS의 경우는 4KB 단위의 크기를 사용하고 있다. 이것을 인덱스 레코드(Index record)라고 칭한다. 루트 노드는 MFT엔트리의 상주 속성을 이용하고 있으므로 개략 600바이트 정도의 크기를 사용한다.

1.3 Root node with minimum node or leaf node

일반 B-tree : 최소 루트 노드와 2개의 자식 노드를 가진다.

NTFS B-tree : 디렉토리의 목록의 수가 한 개의 인덱스 레코드로 충분히 표현될 수 있는 경우라면 자식 노드 한 개만으로 구성된 형태의 B-트리도 가능하다.

1.4 Root node without child node or leaf node

일반 B-tree : 가능-자식노드를 갖지 않고 루트노드에만 항

목을 갖는 경우, 루트노드의 저장 공간이 허용하는 한도까지 키 값을 갖게 된다.

NTFS B-tree : 가능-상주 속성으로 MFT엔트리 내에 $ROOT_INDEX속성으로 저장된다. 디렉토리의 목록의 수가 $ROOT_INDEX속성에 저장 가능한 공간크기(약 600여 바이트 정도)까지 허용된다.

1.5 Key

일반 B-tree : 숫자, 문자 등의 데이터가 사용된다.

NTFS B-tree : 파일명이 키 값으로 사용됨. 파일명의 길이는 확장자를 포함하여 1문자(2바이트 유니코드)에서 255자까지 범위 내에서 가능함. 8.3포맷보다 긴 이름을 저장하는 경우는 8.3포맷으로 긴 이름을 축약한 이름과 더불어 저장이 된다. 한 파일에 긴 파일명과 짧은 파일명의 두 개의 키가 만들어지는 결과가 된다. 긴 파일명의 경우는 8.3 포맷의 짧은 파일명과 긴 파일명을 동시에 저장한다. 둘 중의 한 가지 옵션만 선택하는 것도 가능하다[20]. 8.3포맷의 짧은 파일명을 사용 중인지 체크하는 기능과 사용불능 설정에 해당하는 fsutil명령은 다음과 같다.

fsutil 8dot3name query c:
fsutil behavoir set disable8dot3 3

1.6 Key sorting

일반 B-tree : 키 값은 항상 오름차순으로 정렬된다.

NTFS B-tree : 키 값(파일/디렉토리명)은 항상 오름차순으로 정렬된다. 파일명 정렬의 기준은 특수문자, 숫자, 대소문자 구분 없이 영문자, 특수문자 순으로 정렬하게 된다. 즉, 대소문자를 구분하지 않고 아스키 코드순으로 정렬한다고 생각하면 간단히 이해된다.

1.7 Pointer to child node

일반 B-tree : 노드의 좌측, 우측을 가리키는 포인터 정보를 가지며 자식노드가 있는 곳의 포인터 값(위치값)을 저장한다.

NTFS B-tree : 내부 노드 역할을 하는 인덱스 레코드 안에 들어 있는 인덱스 엔트리들은 각 엔트리 정보의 끝 부분에 8바이트의 왼쪽 자식노드의 VCN값이 추가된다. 16바이트 크기의 “End of list”에 8바이트의 오른쪽 자식의 VCN 값이 붙는다. (예: 5장의 그림 10의 경우는 루트 키에 붙어있는 VCN값은 왼쪽 자식을 나타내고, “End of list”에 붙어 있는 값은 오른쪽 자식을 타나낸다)

1.8 Expansion of node

기본 B-tree : 노드에 들어 있는 항목이 m개이고 추가로 한 개 더 입력이 되면 두 개의 노드로 나뉜다. 노드는 { 1, 2, ... , ⌈m/2⌉-1, ⌈m/2⌉, ⌈m/2⌉+1, ... , m-1, m}중에서 중간값(m/2)가 상위노드의 키로 부모 노드로 이동하고 {1, 2, ... , ⌈m/2⌉}가 좌측의 노드로 구성되고 {⌈m/2⌉+1, ... , m-1, m}가 우측의 노드로 구성된다.

NTFS B-tree : 기본 B-트리의 경우와 같은 방식으로 동작한다. 차이점은 차수가 정해져있지 않기 때문에 인덱스 레코드(노드)안에 들어가는 항목들(인덱스 엔트리)의 수가 가변적이라는 점이다. 좌측과 우측노드로 분리되는 시점은 인덱스 레코드에 추가적으로 인덱스 엔트리를 저장할 수 없는 경우이다.

1.9 Reduction of node

기본 B-tree : 한 노드에 들어 있는 항목(또는 키)들이 삭제에 의하여 모두 제거되면 부모 노드의 좌우 관련 두 개의 노드와 부모노드 키의 왼쪽 키가 새로운 부모노드의 키가 된다.

NTFS B-tree : NTFS는 파일의 목록이 삭제되어서 어떤 인덱스 레코드가 빈 상태가 되어도 트리의 구조를 축소하는 방향으로 구조 변경을 하지 않는다. 모든 목록이 삭제되어 디렉토리가 비워지면 전체의 런-리스트를 삭제하면서 트리를 해체하게 된다.

1.10 Return of empty node

기본 B-tree : 항목이 삭제되어 노드가 지워지는 경우 할당된 메모리를 해제.

NTFS B-tree : 한 번 생성된 인덱스 레코드는 인덱스 레코드 내에 엔트리가 하나도 없더라도 전체 파일목록 중에서 1개라도 남아있으면 회수를 하지 않는다. 인덱스 레코드의 모든 파일목록이 삭제되면 할당된 빈 인덱스 레코드들이 런-리스트에서 삭제되면서 할당영역이 해제된다.

Table 6 Comparision List of Ordinary and NTFS B-tree

Items	Ordinary B-tree	NTFS B-tree
Variation of Node Size	Fixed	Variable
Node Size	Maintain initial size	$INDEX_ROOT:about 0.6KB $INDEX_ALLOCATION: 4KB
Min. No of Child	Left and Right Child of Root, (2 nodes)	1)One Node Available 2)Two Node Available
Root without Child	Available ROOT is used as leaf node	Available $INDEX_ROOT is used as leaf node
Key	Number, Character are available	File Name is used as a key
Key Sorting	Ascending	Ascending
Pointer to Child Node	Pointer/Location	8-Bytes added to last of index entry
Expansion of Node	Min item ⌈m/2⌉ move to parent node. Smaller items move to left child node, larger items to right child node.	Separate into two nodes if index recode is full acts as an ordinary B-tree
Reduction of Node	Empty node merge to sibling node	Maintain empty index record If at least one index entry exists.
Node Return	Node returned if entries are empty	If run-list exists, no empty index records are return to unallocated.

2. NTFS Uses B-tree Not B+tree

NTFS의 인덱스에서 사용하는 트리는 B+ 트리가 아닌 B트

리이다. NTFS에서 디렉토리 인덱스 유지하기 위한 방법으로 B+트리가 사용된다고 알려져 있다[1,3,4,5].

ReiserFS(Unix and Linux), XFS filesystem(IRIX, Linux), JFS2 filesystem (AIX, OS/2, Linux) NTFS filesystem (Microsoft Windows)등 여러 파일시스템에서 디렉토리 인덱스를 위해서 B+트리가 사용된다고 알려져 있다.

J. Huang과 S. Wu의 연구[1]에서 "NTFS adopts B+ tree to manage directory index. The same level directory adopts B+ tree structure and keeps sequence as file (or directory) name sorting. An index item is a node of B+ tree, the root node of B+ tree may reside in MFT record or index item."라 설명하고 있다. Wikipedia의 "B+tree" 설명에도 "NTFS uses B+ trees for directory and security-related metadata indexing.[3]"이라는 설명을 붙이고 있다. 또한, Dominic Giampaolo[4]의 저서에서도 "Directories in NTFS are stored in B+trees that keep their entries sorted in alphabetic order."라는 문장을 발견할 수 있다[4]. M. Russinovich의 비고[5]에서도 "The attribute indexing process sorts directory entries by name and stores the entries as a B+ tree (a form of a binary tree that stores multiple items at each node in the tree."라고 기술하고 있다.

그러나 NTFS에서 이런 설명들은 잘못된 것이다. NTFS에서는 변형된 B-트리를 사용하고 있다. B-트리를 사용하고 있다고 판단하는 가장 중요한 요소는 B+트리의 주요 특색으로 판단할 수 있다. B+트리는 인덱스셋(Index set)과 순차셋(Sequence set)으로 구성되어 있다. B+트리는 노드들 간의 검색을 위한 인덱스셋의 구조는 B-트리와 동일한 구조를 갖는다. 차이점은 리프노드에 있다. 순차셋 리프에 들어 있는 모든 항목들이 오름차순으로 순차적인 연결구조를 갖는다. 모든 항목들이 이 연결구조에 들어 있어야 한다. 그러므로 Index set에 들어있는 노드들의 키들은 리프노드의 값들이 중복적으로 존재하고 있다는 점이다. NTFS의 인덱스에는 이 방식을 사용하지 않는다. 또한 이전 절의 내용에서 언급한 NTFS B-트리의 특징(표 6)의 들은 B-tree 동작특성이 분명하므로 NTFS에서는 B+트리가 아닌 B-트리를 사용하고 있다고 확인할 수 있다.

V. NTFS B-tree Sample Cases

1. Experimental Environment

이 연구에서 수행하는 사례들은 다음과 같은 테스트 환경에서 작업이 이루어진다.

OS : Windows 10 Pro(Ver. 10.0.14393)
Disc Format : NTFS v3.1
Storage drive: Samsung SSD
Storage Space : 463GB
Disc Allocation Cluster Size : 4,096 bytes
Working Directory : d:₩testBtree
Test File : TestFile01.txt ~ TestFile36.txt
Disk analysis tool : X-Ways Forensics Ver. 18.7

2. Case 1: Leaf Noded Root

이 경우 4장의 1.4절에 해당하는 사례를 분석한 것으로 루트노드가 리프노드로 존재하는 경우이다. 일반 B-트리의 경우도 이런 경우가 사용될 수 있다.

이 사례에서는 파일명의 길이가 14자로 구성되어 있다. 이것의 영향으로 TestFile01.txt ~ TestFile5.txt까지의 5개의 파일을 생성했을 때 까지만 MFT엔트리 내부에 인덱스 엔트리가 저장될 공간이 허용된다. 이 경우는 인덱스 레코드가 할당되지 않은 상태이다. $INDEX_ROOT가 루트노드이며 그 안에 파일 목록(인덱스 엔트리)가 들어 있는 리프노드 형태이다.

그림 4는 $INDEX_ROOT속성 안에 5개의 엔트리의 목록이 들어 있는 상황을 나타낸 것이고 그림 5는 디스크 분석 툴(X-Ways Forensics)을 스크린 캡처한 것이다. MFT엔트리의 번호는 #218479이고 이것의 주소는 0x4980323C00이다. (인덱스 엔트리의 중간부분은 생략됨).

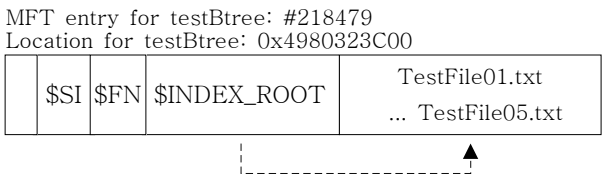

Fig. 4. Leaf Noded Root-Index Entry reside in MFT Entry(TestFile01.txt ~ TestFile05.txt)

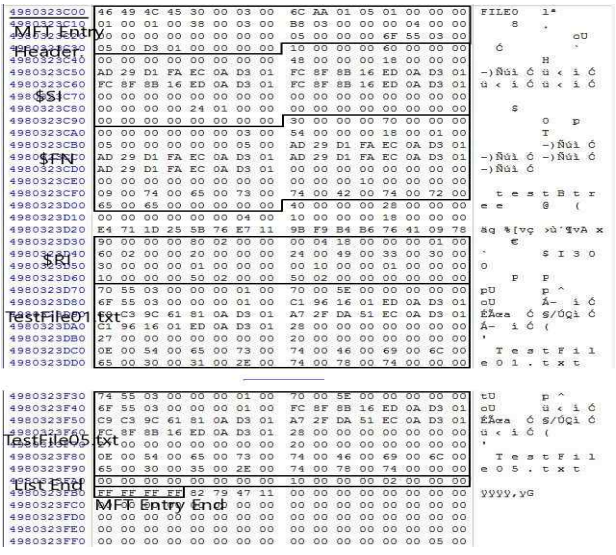

Fig. 5. Leaf Noded Root-Screen capture of Index Entry reside in MFT Entry(TestFile01.txt ~ TestFile05.txt)

3. Case 2: Single Child-One Index Record

이 경우는 4장의 1.3절에 해당하는 사례이다. 일반적인 B-트리

구조에서는 생길 수 없는 특별한 형태가 NTFS에서는 가능하다(2장 2절의 3항의 조건 참조)

TestFile01.txt ~ TestFile5.txt까지 파일이 생성되면 MFT엔트리 내의 $INDEX_ROOT속성에 인덱스 엔트리가 저장된다. TestFile6.txt 파일이 추가로 생성되면 상주공간이 부족해지기 때문에 인덱스 레코드 영역으로 확장된다. 이 때 $INDEX_ALLOCATION속성이 생성된다. 이 속성에는 인덱스 레코드들이 저장되어 있는 위치들을 알려주는 런-리스트 정보가 포함된다. 이 사례에서는 인덱스 레코드가 저장될 클러스터의 주소는 0x49E55800000가 할당되었다. $INDEX_ROOT속성에서도 변화가 생기는데 내부에 갖고 있던 인덱스 엔트리들을 모두 인덱스 레코드 #0로 보내고 $INDEX_ROOT속성에는 아무 인덱스 엔트리도 갖지 않는 상태로 변하게 된다. 루트노드에는 연결될 자식의 VCN번호가 0번이라는 정보만 갖고 실제 물리적 주소(LCN)를 나타내는 런-리스트 정보는 $INDEX_ALLOCATION속성이 갖는다(그림 6참조).

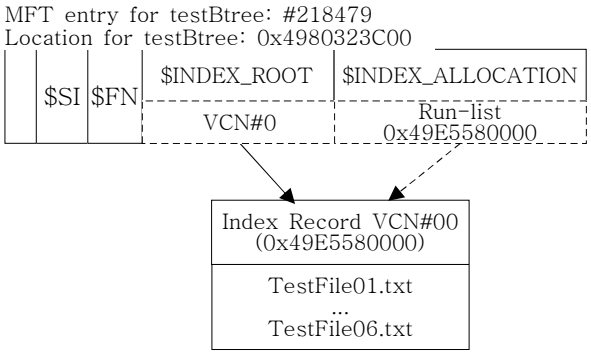

Fig. 6. Single Child-Diagram of MFT Entry with $IR and $IA

그림 7은 testBtree디렉토리의 MFT엔트리를 나타낸 것이다. $INDEX_ROOT속성의 끝 8바이트는 자식노드의 VCN번호 #0번을 나타낸다. $INDEX_ALLOCATION속성의 끝 부분에 런-리스트가 저장되어 있다. “41”에서 “4”의 의미는 리틀엔디언 적용된 LCN주소를 4자리로 표시한다는 의미이고, “1”은 연속된 클러스터의 개수가 저장된 곳의 바이트 크기 정보를 나타낸다. 해당 클러스터는 “01”개라고 표시되어 있다.

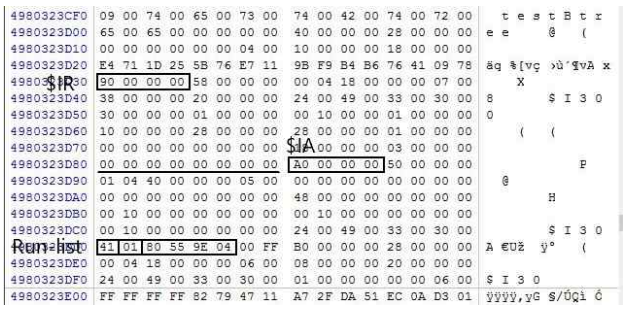

Fig. 7. Single Child-Screen Capture of MFT Entry

그림 8은 testBtree디렉토리의 디렉토리 목록들이 인덱스 엔트리로 저장된 모습이다. 인덱스 레코드 헤더 내의 2행의 8바이트 “00 00 00 00 00 00 00 00”는 해당 인덱스 레코드를 식별하기 위해 사용되는 VCN 값이다. 6개의 인덱스 레코드가 정렬된 상태로 저장된 모습이다. (일부 엔트리는 지면관계상 생략함)

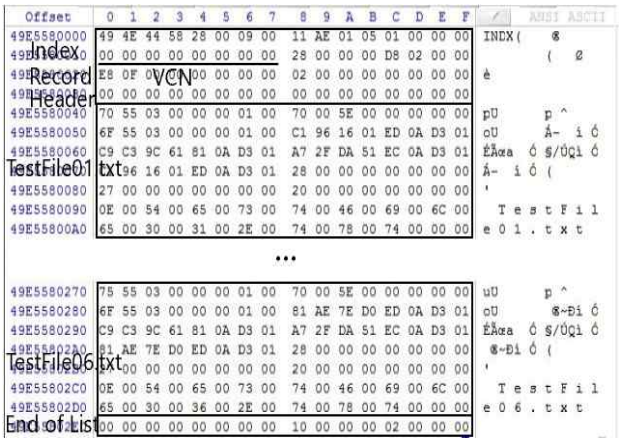

Fig. 8. Single Child-Screen Capture of MFT Entry

4. Case 3: Root Key in $INDEX_ROOT

이 사례는 $INDEX_ROOT속성 안에 루트 키가 저장되어 있고 자식노드로 두 개의 인덱스 레코드를 갖는 경우이다(4장 1.7절 참조). TestFile07.txt ~ TestFile36.txt까지 연속적으로 파일이 생성되면 현재 생성되어 있는 인덱스 레코드(VCN #0)의 공간이 부족해진다. 이 때 TestFile01.txt ~ TestFile36.txt의 파일들의 목록은 두 개의 인덱스 레코드로 나뉘게 된다. 중간 값에 해당하는 TestFile18.txt가 $INDEX_ROOT속성으로 올라가서 루트 키 역할을 하게 된다.

그림 9에서 TestFile18.txt 키의 왼쪽 아래에 VCN#0 인덱스 레코드가 자식노드가 되고 우측 아래에 VCN#1 인덱스 레코드가 자식노드로 구성된다. 이 두 VCN의 실제 주소는 $INDEX_ALLOCATION의 런-리스트에 저장되어 있다.

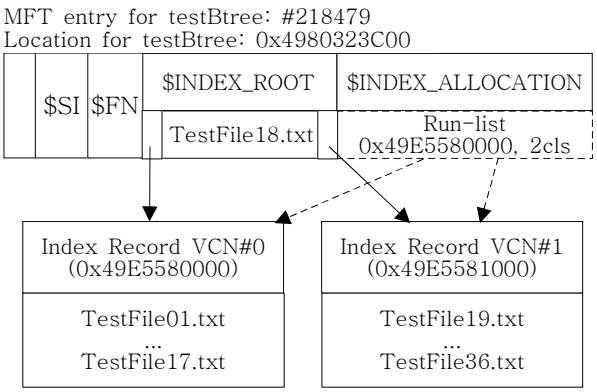

Fig. 9. Diagram of Root Key in $IR

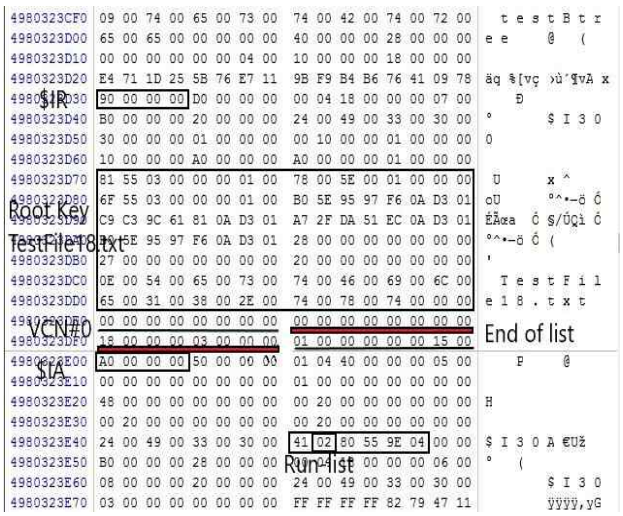

Fig. 10. Root Key in $IR -MFT Entry

그림 10에서 $INDEX_ROOT속성에 루트키 역할을 하는 TestFile18.txt의 인덱스 엔트리가 저장되어 있다. 이 엔트리의 끝에는 왼쪽 자식 노드인 VCN#0의 번호가 “00 00 00 00 00 00 00 00”처럼 8바이트로 저장되어 있다. 오른쪽 자식 노드의 키는 “End of list“가 갖고 있다. VCN#1의 실제 번호가 ”01 00 00 00 00 00 00 00“의 8바이트 값으로 저장되어 있다.

$INDEX_ALLOCATION의 런-리스트에 “41 02 80 55 0E 04”는 리틀엔디언으로 해석해야 하며 “02”는 연속된 클러스터 두 개 즉, VCN#(0x49E5580000), VCN#1(0x49E5581000)을 의미한다.

그림 11과 그림 12는 각각 인덱스 레코드 #0, #1을 나타내는데 인덱스 레코드의 2행의 8바이트 숫자가 인덱스 레코드의 VCN번호를 알려준다. 그림 11에는 TestFile01.txt ~ TestFile17.txt까지 들어 있고, 그림 12에는 TestFile19.txt ~ TestFile36.txt까지 들어 있다.

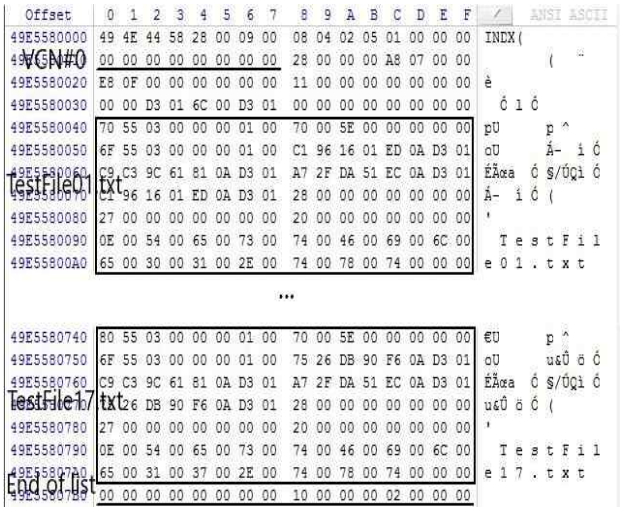

Fig. 11. Root Key in $IR -Index Record#0

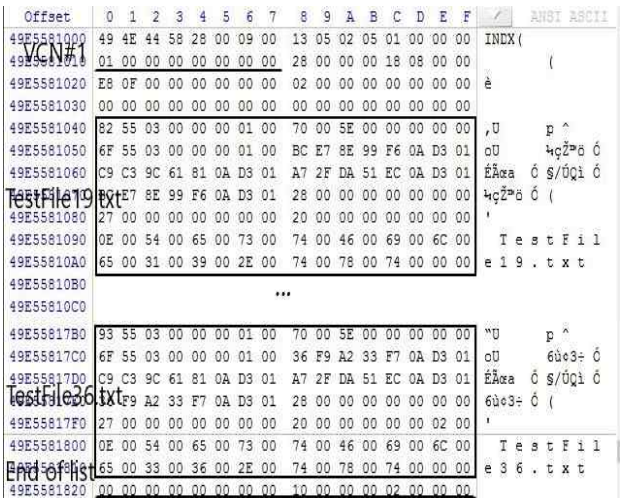

Fig. 12. Root Key in $IR -Index Record#1

5. Case 4: Deletion of entries

이것은 4장의 1.9절과 1.10절에 해당하는 내용을 설명하기 위한 사례이다. TestFile02.txt ~ TestFile36.txt까지의 파일을 모두 삭제하고 TestFile01.txt만 남겨둔다. 일반적인 B-트리에서라면 루트노드에 리프노드 형태로 한 개의 항목만 보유하고 있는 트리로 축소가 될 것이다. 그러나 NTFS B-트리에서는 그렇게 되지 않는다. 한번 할당한 인덱스 레코드들은 완전히 목록이 제거되기 전까지 전체 구조의 변화를 갖지 않는다. 다시 말하면 확장은 지속적으로 추구하지만 축소는 수행하지 않는다고 이해할 수 있다. 디렉토리의 모든 파일이 제거되면 전체의 런-리스트를 삭제하는 것으로 모든 할당된 인덱스 레코드를 해제한다.

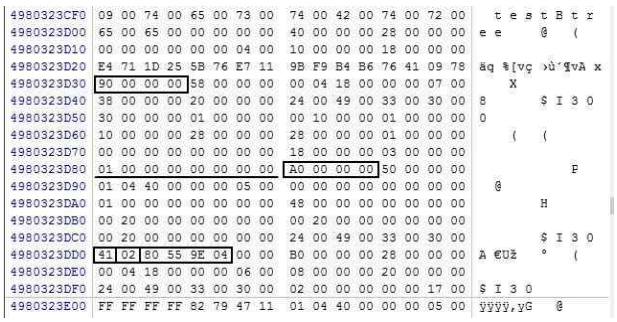

Fig. 13. Delete of Index Entry - MFT

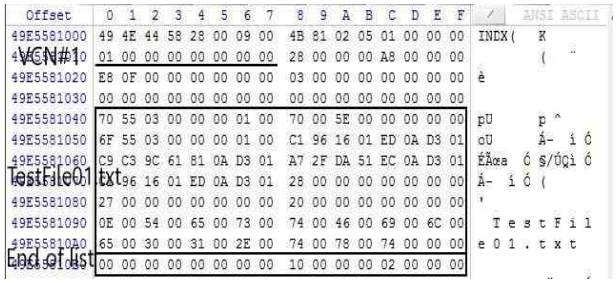

Fig. 14. Delete of Index Entry - Index Record #1

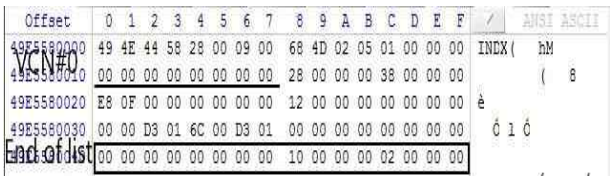

Fig. 15. Delete of Index Entry - Index Record #0

일반 B-트리라면 이 상황에서 그림 13의 $INDEX_ROOT속성에 TestFile01.txt 인덱스 엔트리를 저장해야 한다. 그러나 그렇게 되지 않는다. 이 속성의 맨 끝의 “01 00 00 00 00 00 00 00”의 값이 자식노드의 값이다. 자식노드의 번호가 VCN#0에서 VCN#1로 변경된 변화가 생겼지만 한 개의 인덱스 엔트리를 갖고 있음에도 불구하고 인덱스 레코드를 형성하고 있다는 사실에 주목해야 한다. $INDEX_ALLOCATION속성에서 그 사실을 확인 할 수 있다. 런-리스트의 값이 “41 02 80 55 9E 04”로 사례 3의 경우와 다르지 않음을 알 수 있다. 이 사실로 전체 트리가 유지되고 있다는 것을 확인할 수 있다.

그림 14는 인덱스 레코드 #1(VCN#1)에 TestFile01.txt 인덱스 엔트리가 저장되어 있는 모습을 나타낸 것이다. “End of list” 뒤에 많은 기록들이 남아 있지만 이전 작업에 의한 흔적들이다. 인덱스 레코드의 슬랙 영역에 지워진 기록들은 삭제되지 않고 그대로 남아서 디지털 포렌식의 증거 제공의 근거가 된다. 그림 15는 인덱스 레코드 #0(VCN#0)에 인덱스 엔트리가 하나도 없는 상태로 빈 인덱스 레코드임에도 그대로 유지되고 있음을 확인할 수 있다.

VI. Conclusions

이 연구에서는 전통적인 B-트리와 Windows NTFS 파일시스템에서 디렉토리의 관리를 위해 구현된 B-트리의 방식의 차이점을 대조해 보기 위한 분석을 수행하였다. 디렉토리의 인덱스의 정보가 디지털 포렌식 분석을 중요한 분석대상[1,6,7,8,9]으로 인식되고 있어서 이 연구에서 수행한 분석이 파일시스템의 연구와 포렌식 분석 연구를 더욱 명확하게 할 수 있는 정보를 제공한다는 의미에서 이 연구의 가치를 찾을 수 있다.

기존의 연구에서는 이 연구에서 제시한 분석자료 수준의 정보를 제공한 경우가 없었기에 이 연구의 가치를 상대적으로 평가해 볼 수 있다. 그리고 기존의 문헌에서 NTFS에서 디렉토리 인덱싱을 위해서 사용하는 데이터 구조가 B+ 트리라고 기술하고 있는 부분에 대해서 잘못된 정보라는 사실을 밝힘으로써 잘못된 기존의 문헌의 내용에 대한 수정을 할 수 있는 것에 이 연구의 의미가 있다.

본 연구의 결과를 바탕으로 향후의 연구과제로서 리눅스/안드로이드에서 사용하는 ext3/4와 MacOS에서 사용하는 HFS+ 등의 파일시스템에 대해서 B-트리의 구현된 방식의 차이점을 분석하는 연구를 수행하면 디지털 포렌식 분야에서 활용도가 높은 연구가 될 것으로 생각된다.

REFERENCES

[1] Jun Huang and Shunxiang Wu, “The Research of Fast File Destruction Based on NTFS”, ECICE 2012, AISC 146, pp. 613-619.

[2] Ellis Horowitz, Sartaj Sahni, Dinesh P. Mehta, Fundamentals of Data Structures in C++, 2nd Ed., Silicon Press, 2007.

[3] Wikipedia, “B+ tree”, https://en.wikipedia.org/wiki/B+tree.

[4] Dominic Giampaolo, Practical File System Design: The Be File System, pp. 40-44, Morgan Kaufmann Publishers, Inc., 1999

[5] M. Russinovich, "Inside Win2K NTFS, Part 1". MSDN. Microsoft. Retrieved 2008-04-18.

[6] William Ballenthin, “NTFS INDX Attribute Parsing,” http://www.williballenthin.com/forensics/indx/index.html.

[7] Chad Tilbury, “NTFS $I30 Index Attributes: Evidence of Deleted and Overwritten Files,” SANS Digital Forensics and Incident Response Blog, http://digital-forensics.sans.org.

[8] William Ballenthin and Jeff Hamm, “Incident Response with NTFS INDX Buffers - Parts 1, 2, 3 and 4,” https://www.mandiant.com/blog/author/william-ballenthin/

[9] Gyu-Sang Cho, “NTFS Directory Index Analysis for Computer Forensics”, Proceedings of IMIS 2015, , Brazil, pp. 441-446, July 2015,

[10] Gyu-Sang Cho, “A New NTFS Anti-Forensic Technique for NTFS Index Entry”, The Journal of Korea Institute of Information, Electronics, and Communication Technology (ISSN 2005-081X), vol. 8, no. 4, August 2015.

[11] Gyu-Sang Cho, “An Anti-Forensic Technique for Hiding Data in NTFS Index Record with a Unicode Transformation”, Journal of Korea Convergence Security Association, Vol. 16, No. 7, pp. 75-84, July 2015.

[12] Gyu-Sang Cho, “A Digital Forensic Analysis for Directory in Windows File System”, J. of Korea Society. of Digital Industry & Information Management, vol. 11, no. 2, pp. 73-90, June 2015.

[13] Wikipedia, “B-tree”, http://en.wikipedia.org/wiki/B-tree.

[14] Microsoft TechNet, “How NTFS Works”, https://technet.microsoft.com/en-us/library/cc781134(v=ws.10).aspx.
[15] Wikipedia, “NTFS”, http://en.wikipedia.org/wiki /NTFS.
[16] B. Carrier, File System Forensic Analysis, Addison-Wesley, 2005, pp. 273-396.
[17] Wicher Minnaard, "Timestomping NTFS," IMSc final research project report, University of Amsterdam, Faculty of Natural Sciences, Mathematics and Computer Science, 2014.
[18] Wasim Ahmad Bhat and S. M. K. Quadri, “A Quick Review of On-Disk Layout of Some Popular Disk File Systems”, Global Journal of Computer Science & Technology, Volume 11 Issue 1, April 2011.
[19] Petra Grd and Miroslav Bača, "Analysis of B-tree data structure and its usage in computer forensics", Proc. of the 21st Central European Conf. on Information and Intelligent Systems", pp. 423-428, Jan. 2010.
[20] Microsoft Technet, Fsutil behavior, https://technet.microsoft.com/en-us/library/cc785435(v=ws.11).aspx

Authors

Gyu-Sang Cho received the B.S., M.S. and Ph.D. degrees in Electronic Engineering from Hanyang University, in 1986, 1989 and 1997, respectively. Dr. Cho joined the faculty of the Department of Computer Inforamtion at Dongyang University, Yeongju, Korea, in 1996. He is currently a Professor School of Public Technology Service at Dongyang University, Dongducheon, since 2017. He is interested in digital forensics, system secirity, and IoT security.