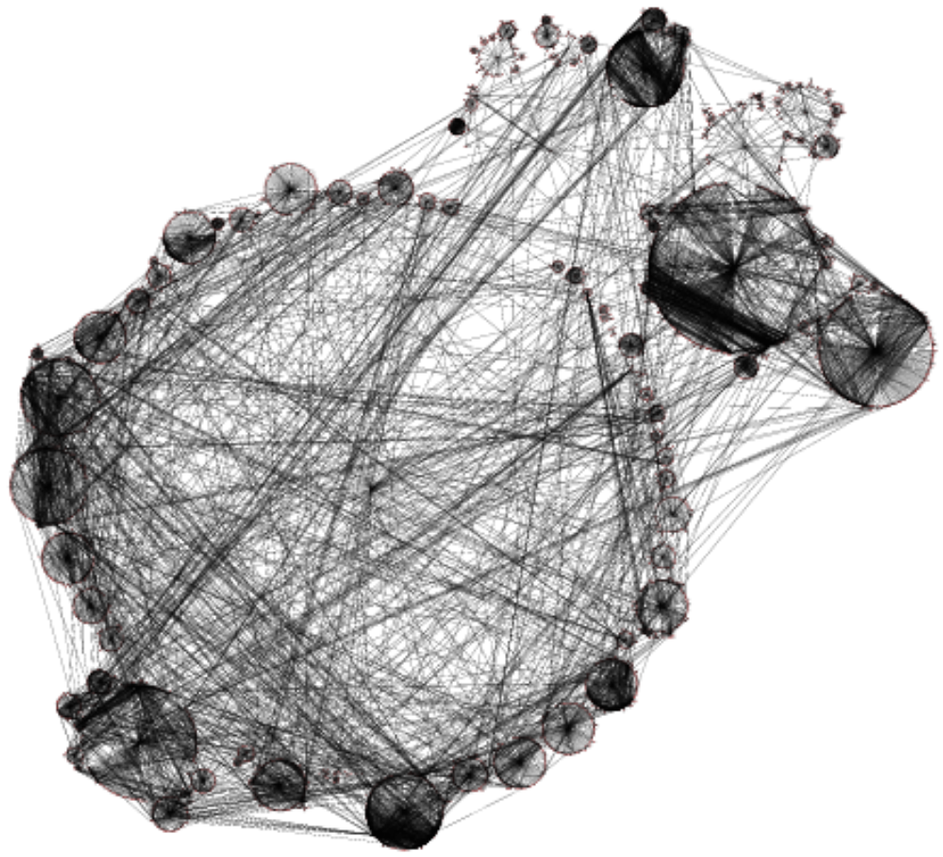


제8장

Case Study: 파일 압축

Huffman Coding

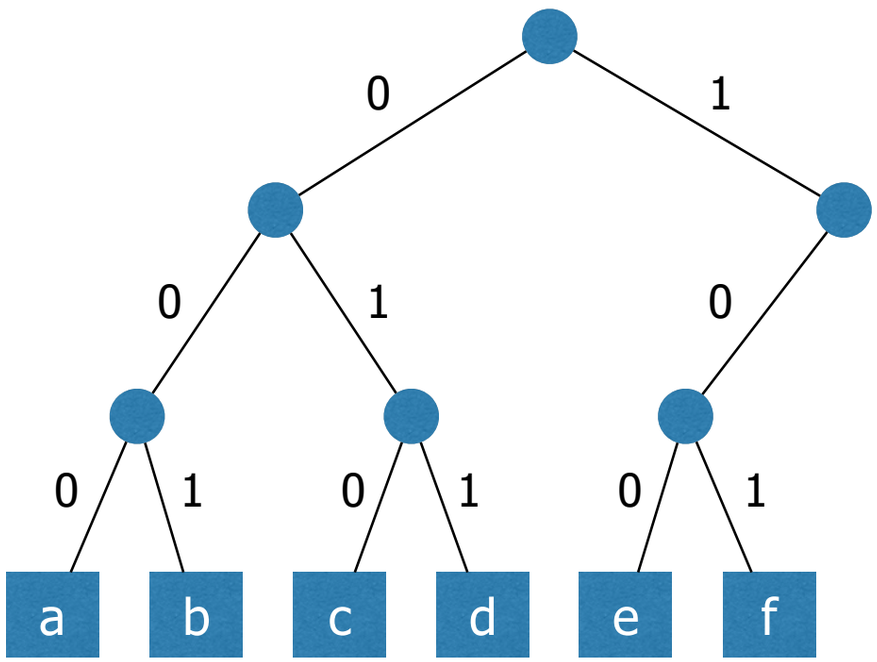
- 가령 6개의 문자 a, b, c, d, e, f로 이루어진 파일이 있다고 하자. 문자의 총 개수는 100,000개이고 각 문자의 등장 횟수는 다음과 같다.

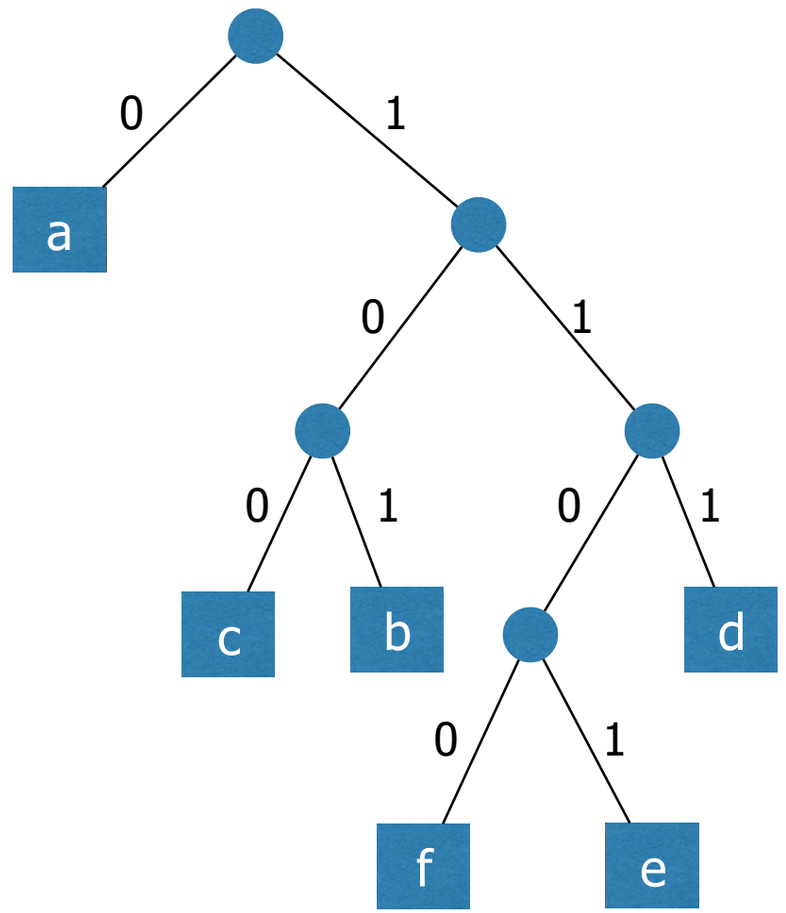
	a	**b**	**c**	**d**	**e**	**f**
Frequency	45	13	12	16	9	5
Fixed-length code	000	110	010	011	100	101
Variable-length	0	101	100	111	1101	1100

- 고정길이 코드를 사용하면 각각의 문자를 표현하기 위해서 3비트가 필요하며, 따라서 파일의 길이는 300,000비트가 된다.
- 위 테이블의 가변길이 코드를 사용하면 224,000비트가 된다.

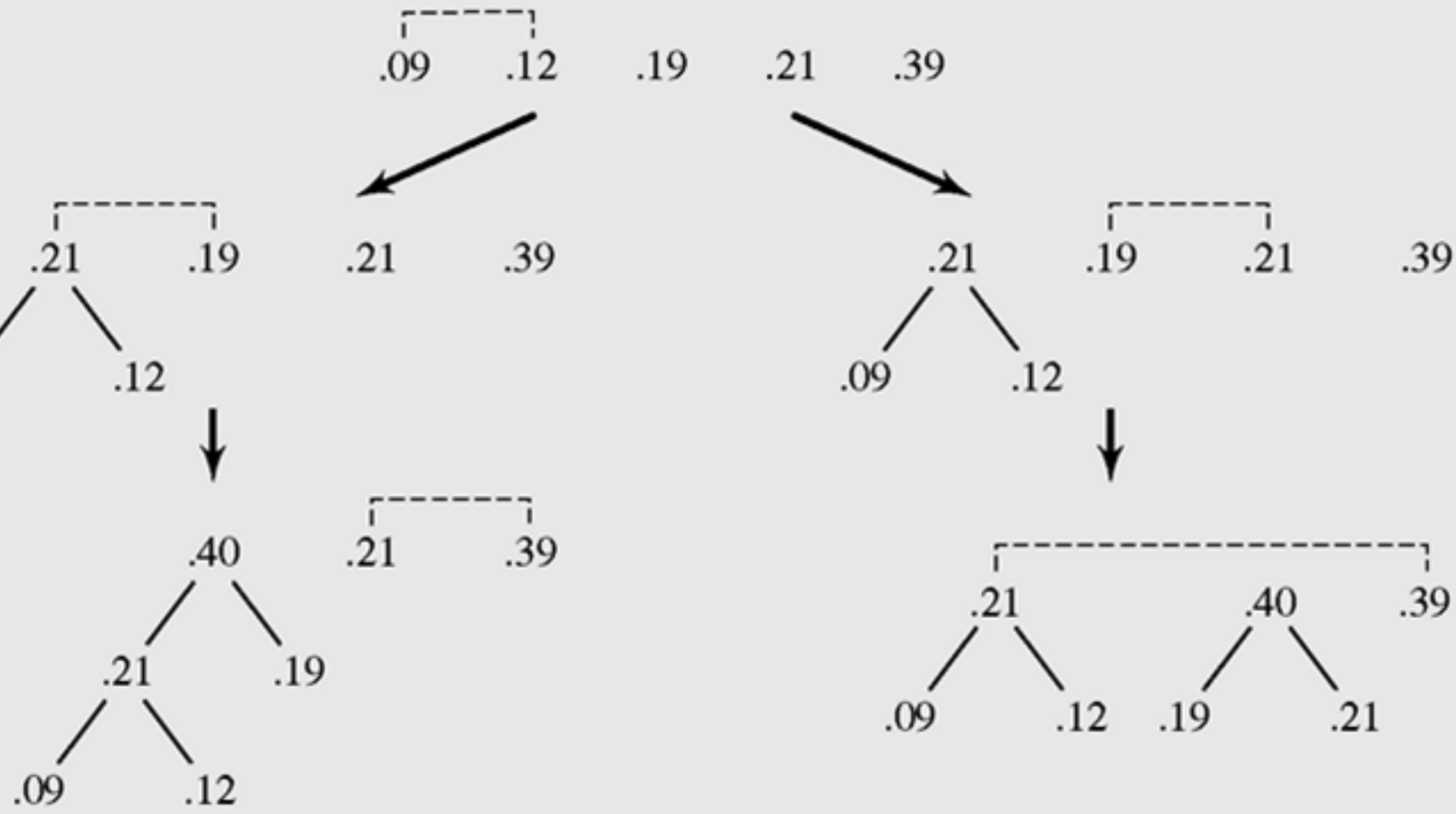
Prefix Code

- 어떤 codeword도 다른 codeword의 prefix가 되지 않는 코드 (여기서 codeword란 하나의 문자에 부여된 이진코드를 말함)
- 모호함이 없이 decode가 가능함
- prefix code는 하나의 이진트리로 표현 가능함

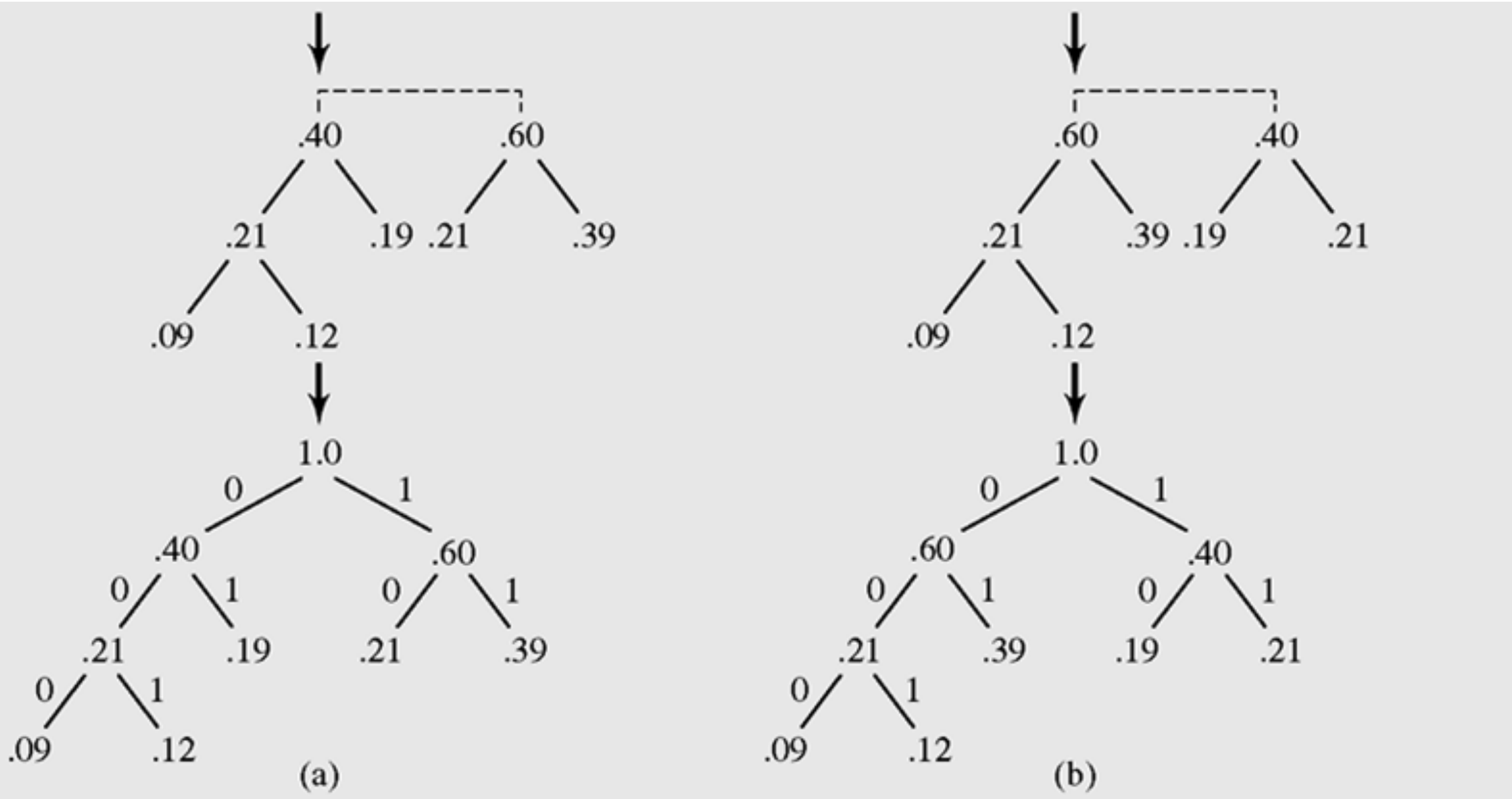




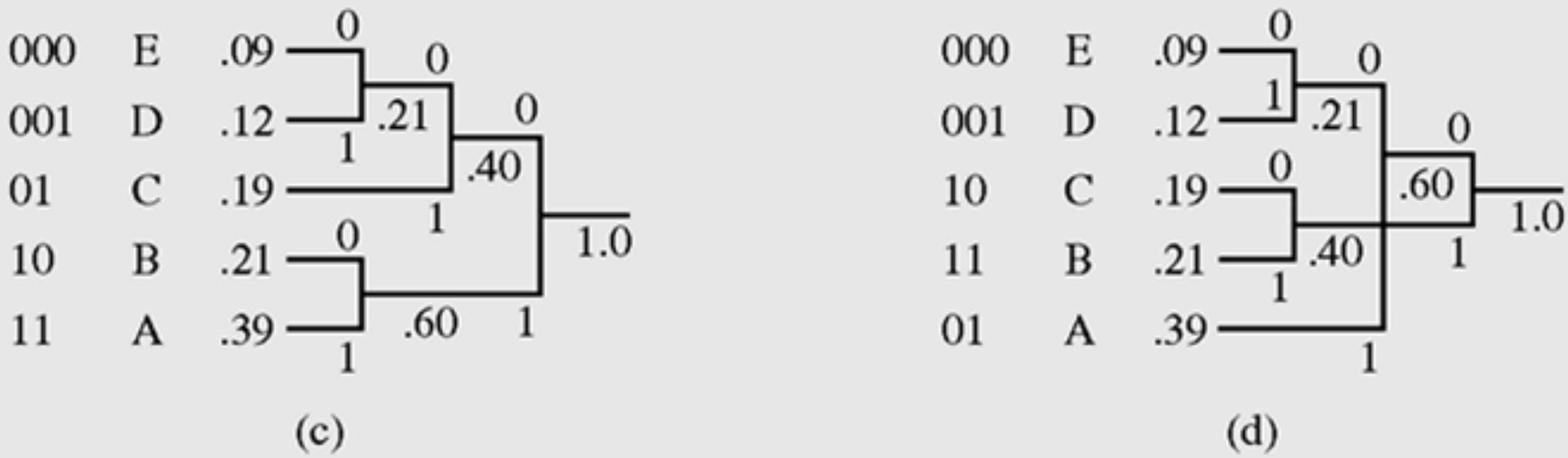
Huffman Coding



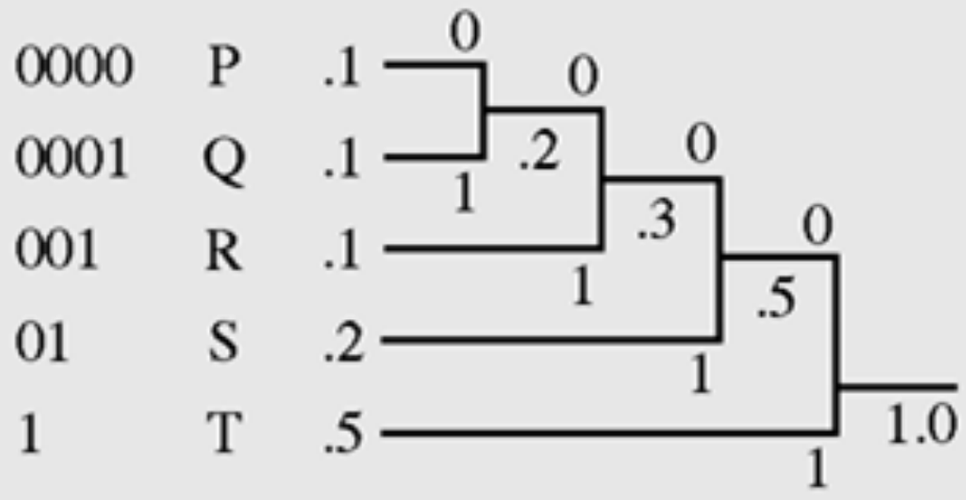
Two Huffman trees created for five letters A, B, C, D, and E with probabilities .39, .21, .19, .12, and .09

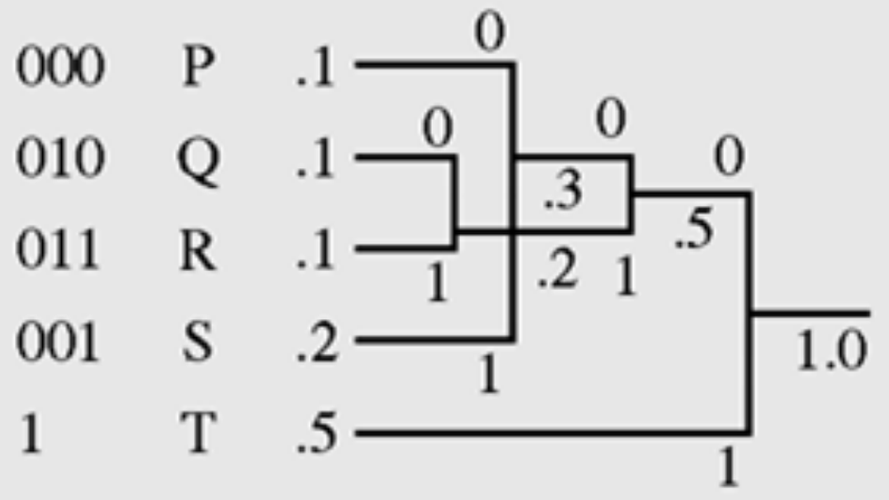


Two Huffman trees created for five letters A, B, C, D, and E with probabilities .39, .21, .19, .12, and .09 (continued)



Two Huffman trees created for five letters A, B, C, D, and E with probabilities .39, .21, .19, .12, and .09 (continued)





Two Huffman trees generated for letters P, Q, R, S, and T
with probabilities .1, .1, .1, .2, and .5

Run-Length Encoding

Run-Length Encoding

- 런(run)은 동일한 문자가 하나 혹은 그 이상 연속해서 나오는 것을 의미한다. 예를 들어 스트링 s = “aaabba”는 다음과 같은 3개의 run으로 구성된다: “aaa”, “bb”, “a”.
- run-length encoding에서는 각각의 run을 그 “run을 구성하는 문자”와 “run의 길이”의 순서쌍 (n, ch)로 encoding한다. 여기서 ch가 문자이고 n은 길이이다. 가령 위의 문자열 s는 다음과 같이 코딩된다: 3a2b1a.
- Run-length encoding은 길이가 긴 run들이 많은 경우에 효과적이다.

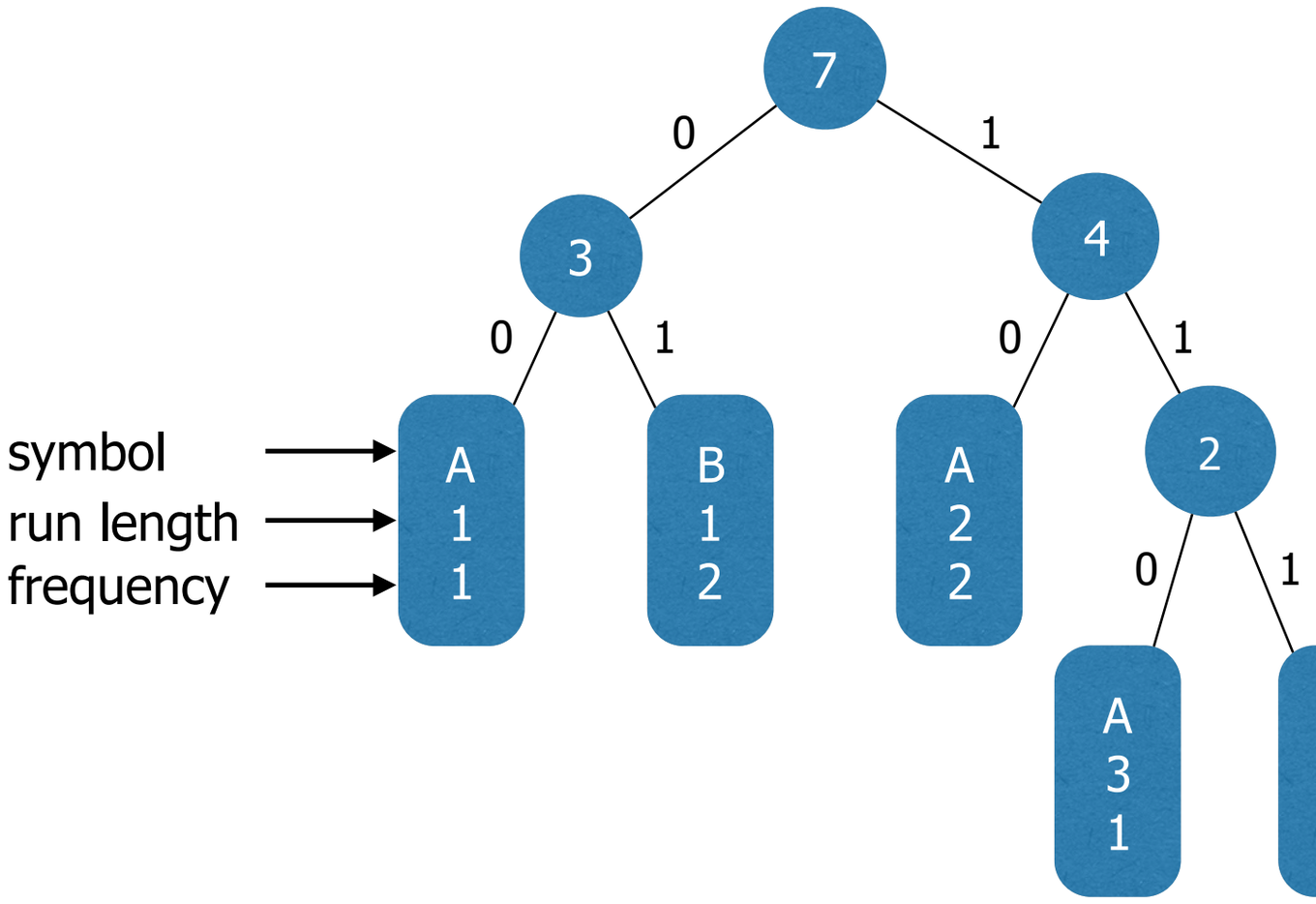
Huffman Method with Run-Length Encoding

Huffman Method with Run-Length Encoding

- 파일을 구성하는 각각의 run들을 하나의 super-symbol로 본다. 이 super-symbol들에 대해서 Huffman coding을 적용한다.
- 예를 들어 문자열 AAABAACCAABA은 5개의 super-symbol들 AAA, B, AA, CC, 그리고 A로 구성되며, 각 super-symbol의 등장횟수는 다음과 같다.

symbol	A	C	A	B	A
run length	3	2	1	1	2
frequency	1	1	1	2	2

Huffman Method with Run-Length Encoding



AAA	110
AA	10
A	00
B	01
CC	111

AAABAACCAABA은 110011011111001000으로 encoding됨

제1단계

Run과 frequency 찾기

Run과 frequency 찾기

- 압축할 파일을 처음부터 끝까지 읽어서 파일을 구성하는 run들과 각 run들의 등장횟수를 구한다.
- 먼저 각 run들을 표현할 하나의 클래스 class Run을 정의한다. 클래스 run은 적어도 세 개의 데이터 멤버 symbol, runLen, 그리고 freq를 가져야 한다. 여기서 symbol은 byte타입이고, 나머지는 정수들이다.
- 인식한 run들은 하나의 ArrayList에 저장한다.
- 적절한 생성자와 equals 메서드를 구현한다.

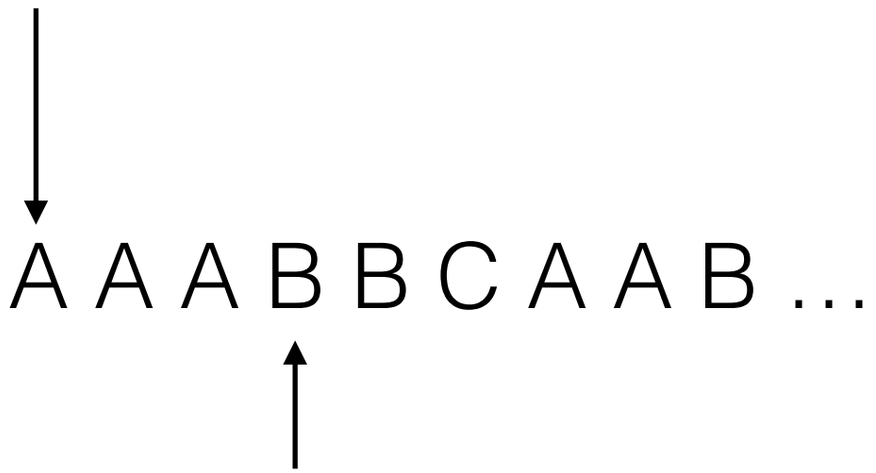
Run과 frequency 찾기

- 데이터 파일을 적어도 두 번 읽어야 한다. 한 번은 run들을 찾기 위해서, 그리고 다음은 실제로 압축을 수행하기 위해서.
- 여기서는 RandomAccessFile을 이용하여 데이터 파일을 읽어본다.

```
/* 읽을 데이터 파일을 연다 */
RandomAccessFile fIn = new RandomAccessFile(fileName,”r”);

/* 한 byte를 읽어 온다. 읽어온 byte는 0~255사이의 정수로 반환된다. */
/* 파일의 끝에 도달하면 -1을 반환한다.                              */
int ch = fIn.read();

/* byte로 casting해서 저장한다 */
byte symbol = (byte)ch;
```

Run 인식하기

1. 파일의 첫 byte를 읽고 이것을 start_symbol이라고 한다.



2. 파일의 끝에 도달하거나 혹은 start_symbol와 다른 byte가 나올 때까지
연속해서 읽는다. 현재까지 읽은 byte수를 count라고 하자.
이 예에서는 지금 byte=4이다.

3. (start_symbol, count-1)인 run이 하나 인식되었다. 이 run을 저장하고
가장 마지막에 읽은 byte를 start_symbol로, count=1로 reset하고 다시 반복한다.

Run과 frequency 찾기

```
class Run {
    public byte symbol;
    public int runLen;
    public int freq;

    /* 적절한 생성자와 equals 메서드를 완성하라. */

}
```

Run과 frequency 찾기

```
public class HuffmanCoding {
    /* 인식한 run들을 저장할 ArrayList를 만든다 */
    private ArrayList<Run> runs = new ArrayList<Run>();

    private void collectRuns(RandomAccessFile fIn) throws IOException {

      /* 데이터 파일 fIn에 등장하는 모든 run들과 각각의 등장횟수를 count하여 */
      /* ArrayList runs에 저장한다. */


    }

    static public void main (String args[]) {
        HuffmanCoding app = new HuffmanCoding();
        RandomAccessFile fIn;
        try {
            fIn = new RandomAccessFile(“sample.txt”,”r");
            app.collectRuns(fIn);
            fIn.close();
        } catch (IOException io) {
            System.err.println("Cannot open " + fileName);
        }
    }
}
```

Implement in C

```
typedef struct run Run;
typedef struct run {
   unsigned char symbol;
   int run_length;
   int freq;
};

Run *runs[MAX];
int number_runs = 0;
```

```
void collectRuns(FILE *fIn) {

  /* 데이터 파일 fIn에 등장하는 모든 run들과 각각의 등장횟수를 count하여 */
  /* 배열 runs에 저장한다. */
  /* Run객체들을 동적메모리할당으로 생성하여 배열 runs에서 객체의 주소를 저장하고 */
  /* 배열의 크기가 부족할 경우 array doubling을 하도록 구현하라. */
  /* byte단위로 파일을 읽기 위해서 fopen에서 “rb” 모드로 open하고 */
  /* fread 함수로 파일을 읽는다. */

}
```

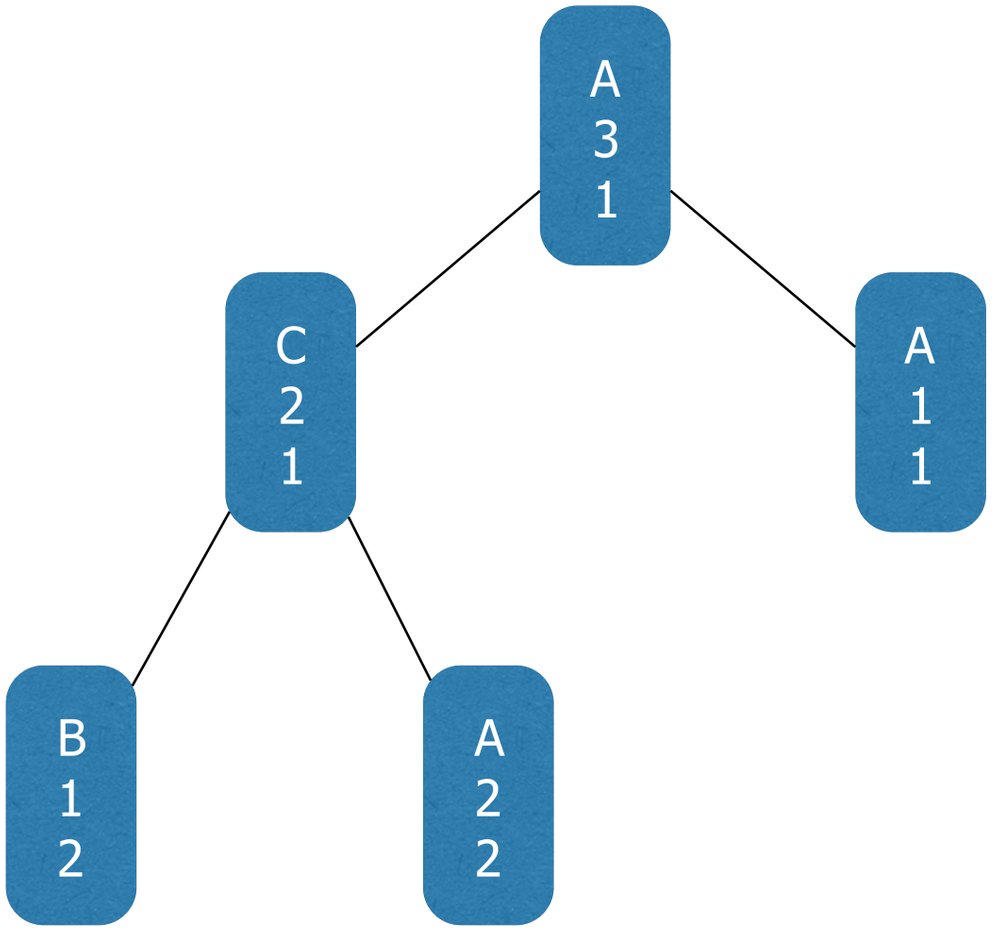
제2단계

Huffman Tree

Huffman Coding

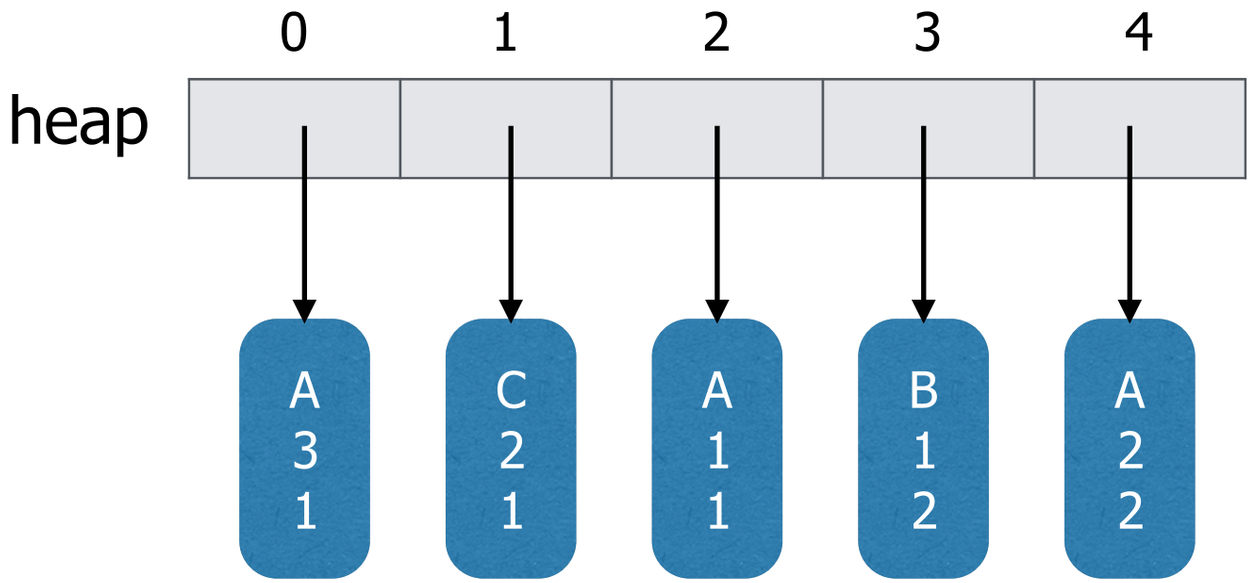
- Huffman coding 알고리즘은 트리들의 집합을 유지하면서
- 매 단계에서 가장 frequency가 작은 두 트리를 찾아서
- 두 트리를 하나로 합친다.
- 이런 연산에 가장 적합한 자료구조는 최소 힙(minimum heap)이다.
- 즉 힙에 저장된 각각의 원소들은 하나의 트리이다 (노드가 아니라).

최소 힙

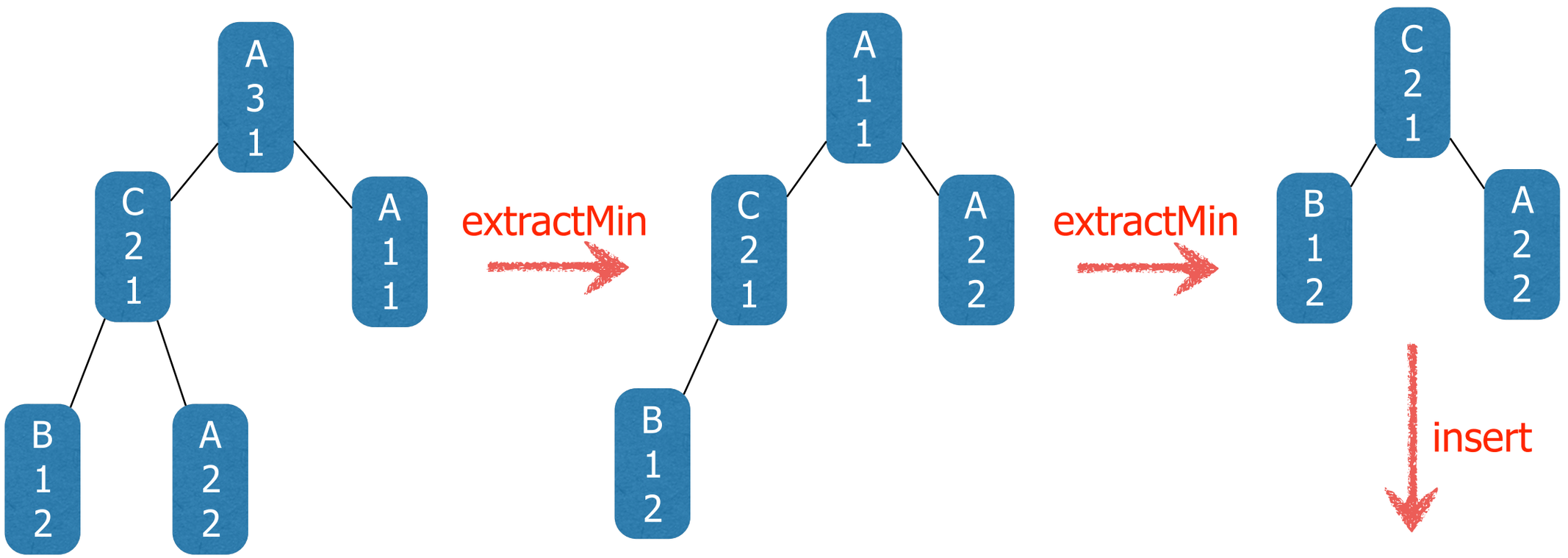


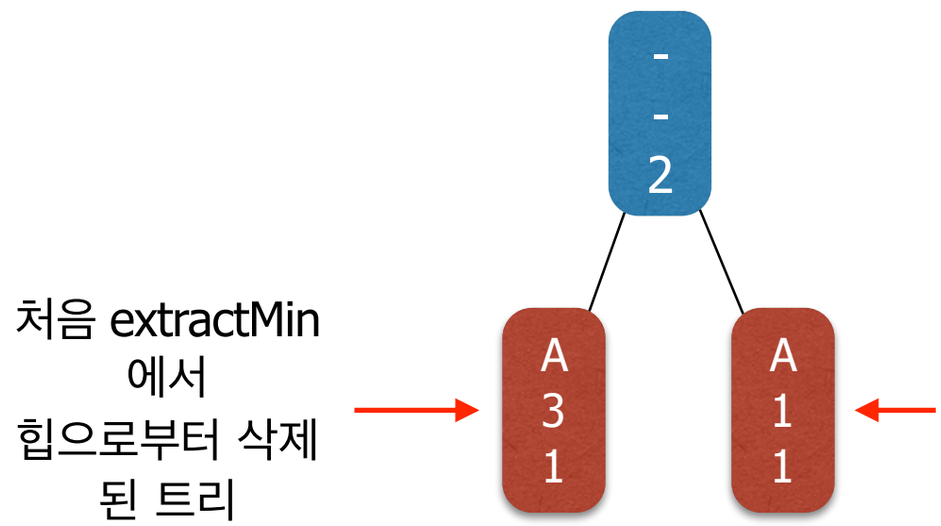
minimum heap

크기가 5인 힙. 5개의 트리가 저장되어 있다.
각 트리는 오직 하나의 노드로 구성



최소 힙

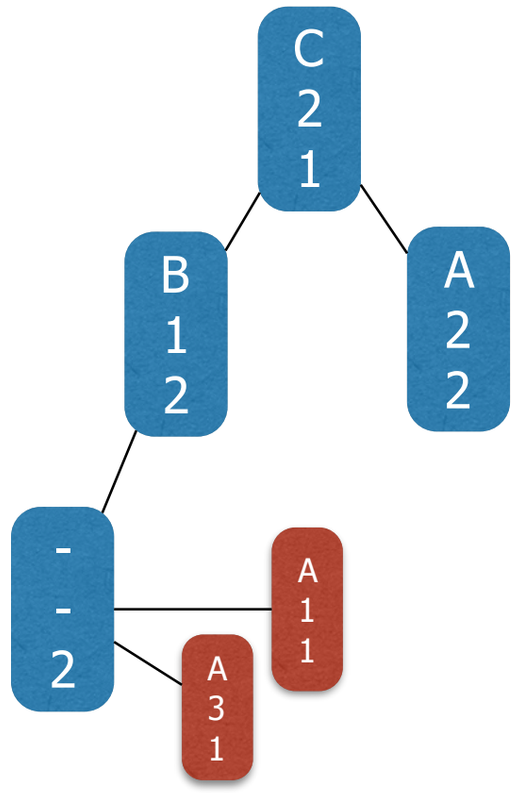




처음 extractMin 에서 힙으로부터 삭제 된 트리

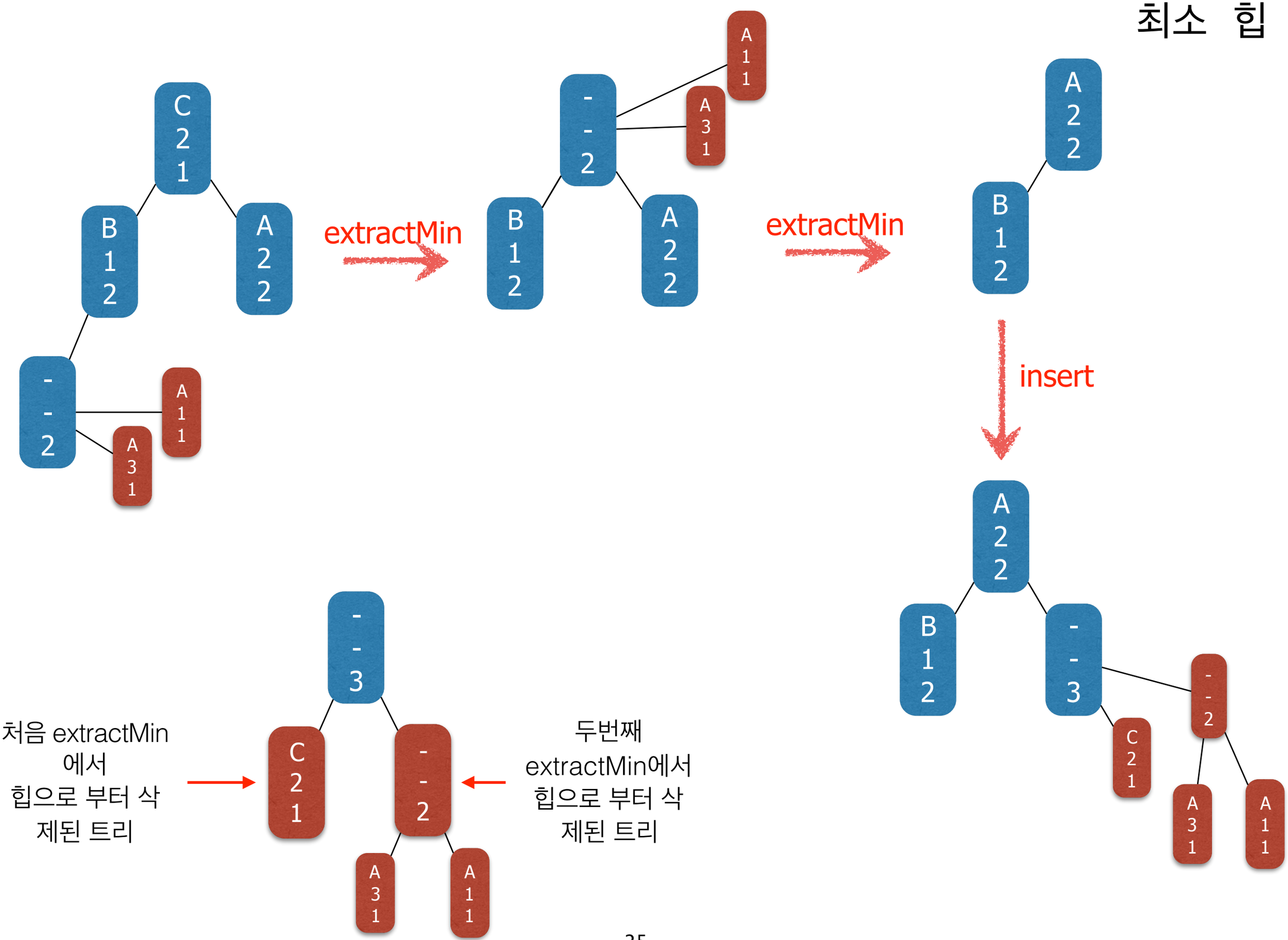
두 트리를 하나로 합친 트리

두번째 extractMin에서 힙으로부터 삭제 된 트리

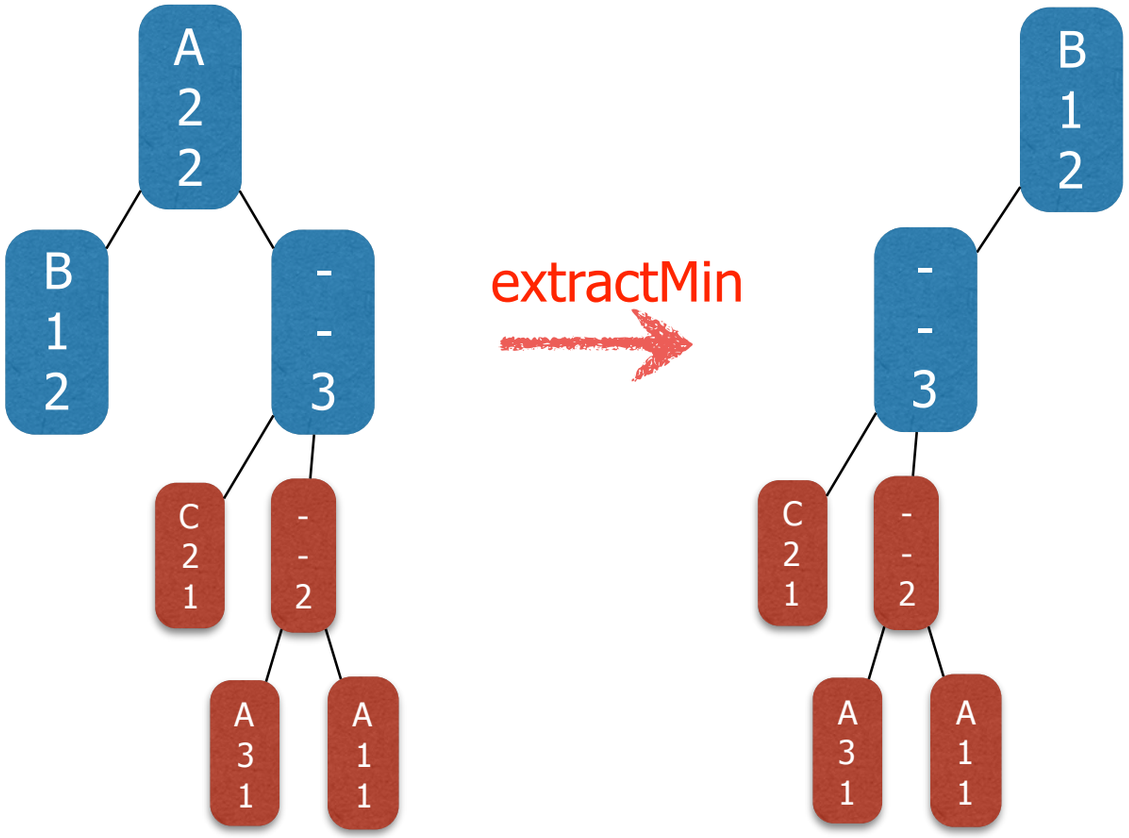


힙에서는 하나의 노드

최소 힙

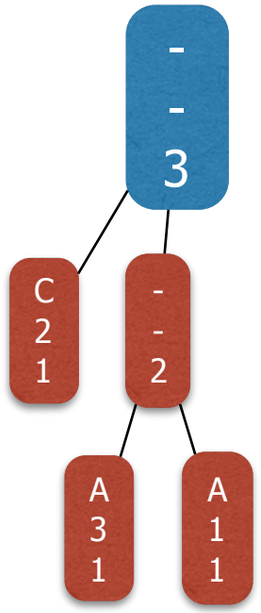

C 2 1
B 1 2
A 2 2
- - 2
A 1 1
A 3 1
extractMin
- - 2
A 3 1
A 1 1
B 1 2
A 2 2
extractMin
A 2 2
B 1 2
insert
A 2 2
B 1 2
- - 3
C 2 1
- - 2
A 3 1
A 1 1
처음 extractMin 에서 힙으로 부터 삭제된 트리
- - 3
C 2 1
- - 2
A 3 1
A 1 1
두번째 extractMin에서 힙으로 부터 삭제된 트리

최소 힙

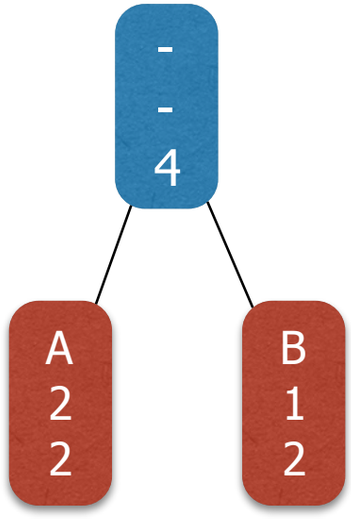


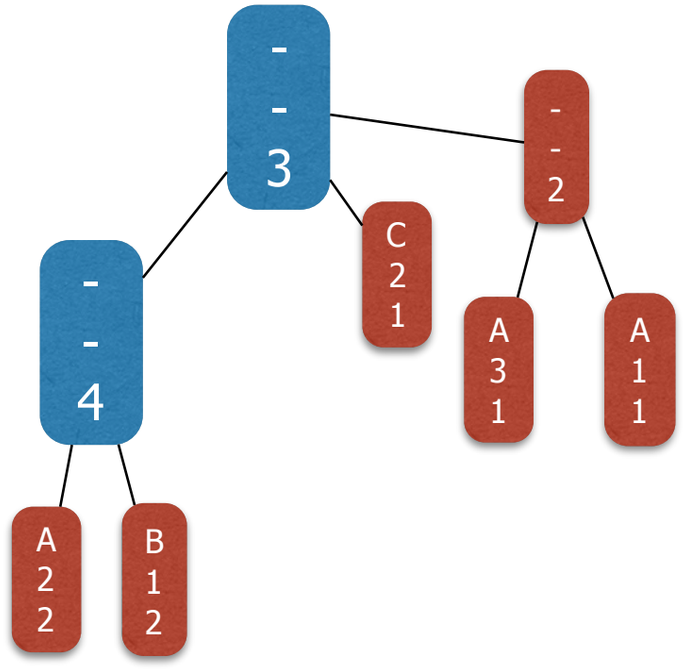
extractMin

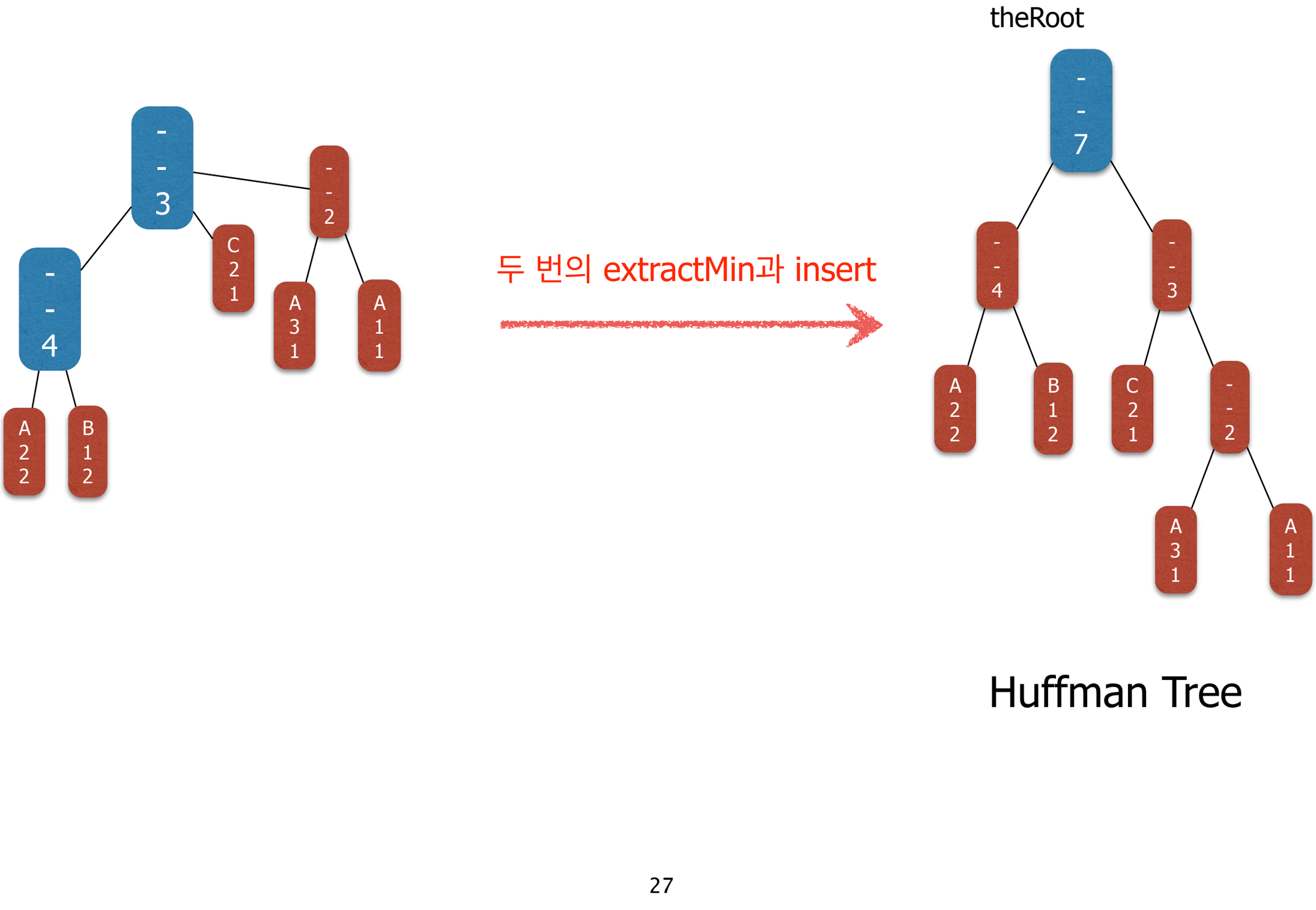
extractMin



insert







Huffman Tree

class Run 수정하기

```
class Run implements Comparable<Run> {
    public byte symbol;
    public int runLen;
    public int freq;

    /* 트리의 노드로 사용하기 위해서 왼쪽 자식과 오른쪽 자식 노드 필드를 추가한다.  */

    /* 두 run의 크기관계를 비교하는 compareTo 메서드를 overriding하라. */
    /* 비교의 기준은 freq이다.  */

}
```

class Heap<Run>

- 3장에서 작성했던 Heap클래스를 가져와서 사용한다. Generics로 수정하고, heapify, insert, extractMin등의 함수들을 min heap에 맞게 수정한다.
- C로 구현하는 사람들은 별개의 모듈로 min heap을 구현하라.

Huffman Tree 만들기

```
public class HuffmanCoding {
    private ArrayList<Run> runs = new ArrayList<Run>();

    private Heap<Run> heap;          /* minimum heap이다. */
    private Run theRoot = null;      /* root of the Huffman tree */

    private void createHuffmanTree() {

        heap = new Heap<Run>();

        /* 1. store all runs into the heap.                    */
        /* 2. while the heap size > 1 do                       */
        /*     (1) perform extractMin two times                */
        /*     (2) make a combined tree                        */
        /*     (2) insert the combined tree into the heap.     */
        /* 3. Let theRoot be the root of the tree.             */

    }
}
```

Huffman Tree 출력해보기

class Run에 적절한 toString 메서드를 추가하여 트리를 출력해보자.

```
private void printHuffmanTree() {
    preOrderTraverse(theRoot, 0);
}

private void preOrderTraverse(Run node, int depth) {
    for (int i=0; i<depth; i++)
        System.out.print("   ");
    if (node == null) {
        System.out.println("null");
    } else {
        System.out.println(node.toString());
        preOrderTraverse(node.left, depth + 1);
        preOrderTraverse(node.right, depth + 1);
    }
}
```

Implementation in C

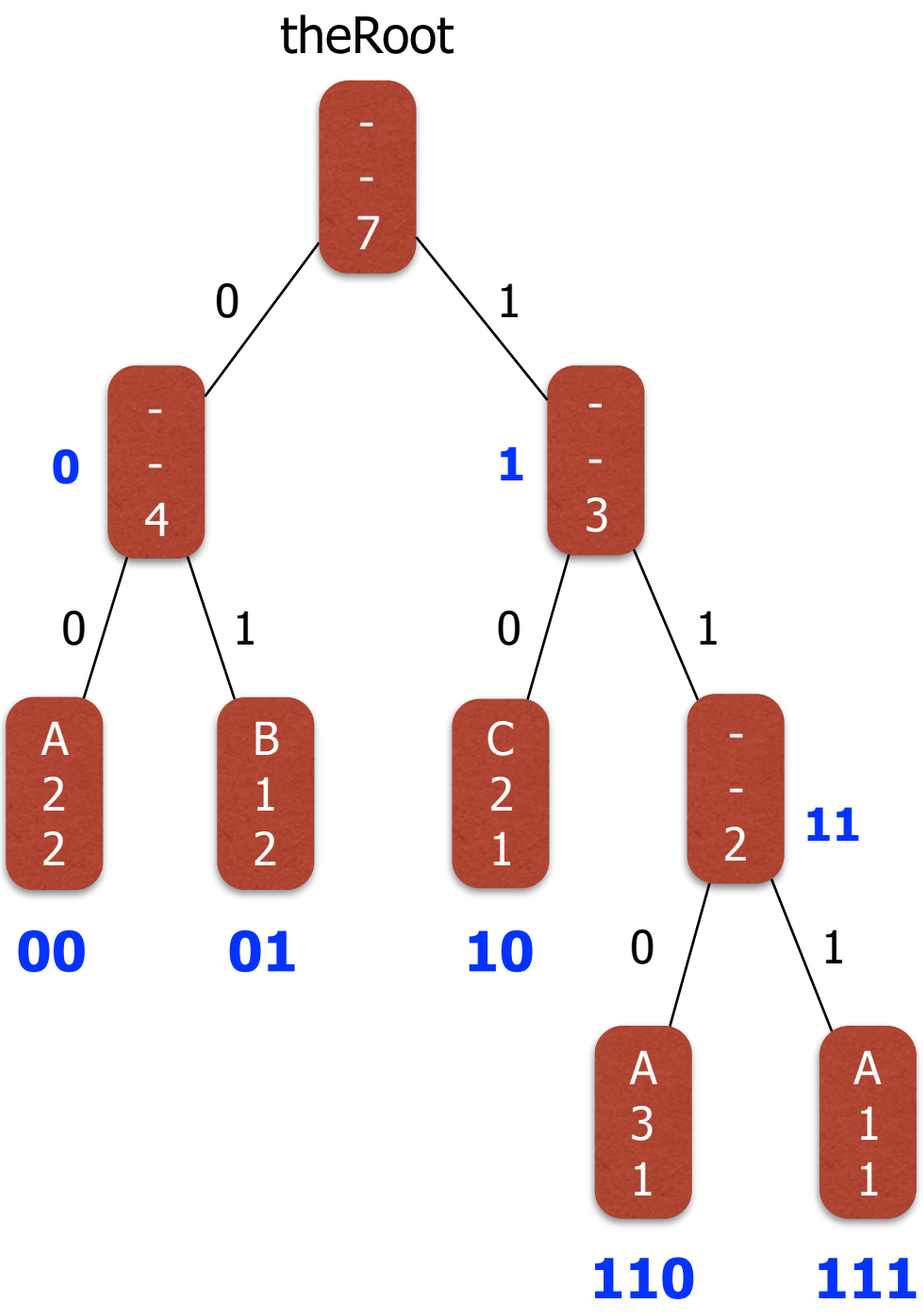
```
Heap heap;                  /* minimum heap이다. */
Run theRoot = null;         /* root of the Huffman tree */
void createHuffmanTree() {

    heap = create_heap();
    /* 1. store all runs into the heap.                    */
    /* 2. while the heap size > 1 do                       */
    /*    (1) perform extractMin two times                 */
    /*    (2) make a combined tree                         */
    /*    (2) insert the combined tree into the heap.      */
    /* 3. Let theRoot be the root of the tree.             */

}
```

제3단계

Codeword 부여하기

Codeword 부여하기

Huffman 트리의 리프 노드에 위치한 run들에게 이진 codeword를 부여할 차례이다.




```
assignCodeword(prefix, node)
if node is a leaf
    assign prefix to the node;
else
    assignCodeword(prefix+'0', node.left);
    assignCodeword(prefix+'1', node.right);
```

Codeword 부여하기

- 여기서 prefix를 하나의 32비트 정수로 표현한다. 하지만 32비트 중에서 하위 몇 비트만이 실제 부여된 codeword이다. 따라서 codeword의 길이를 따로 유지해야 한다.

class Run 수정하기

```
class Run implements Comparable<Run> {
    public byte symbol;
    public int runLen;
    public int freq;

    /* 트리의 노드로 사용하기 위해서 왼쪽 자식과 오른쪽 자식 노드 필드를 추가한다.  */

    /* 노드에 부여된 codeword를 저장하기 위한 필드들을 다음과 같이 추가한다.  */

    public int codeword;        /* 부여된 codeword를 32비트 정수로 저장 */
    public int codewordLen;     /* 부여된 codeword의 길이. 즉 codeword의 */
                                /* 하위 codewordLen비트가 실제 codeword  */
}
```

Java에서 비트(bit) 연산

```
public class Test {
    public static void main(String args[]) {
        int a = 60;         /* 60 = 0011 1100 */
        int b = 13;         /* 13 = 0000 1101 */
        int c = 0;

        c = a & b;              /* 12 = 0000 1100 */
        System.out.println("a & b = " + c );

        c = a | b;              /* 61 = 0011 1101 */
        System.out.println("a | b = " + c );

        c = a ^ b;              /* 49 = 0011 0001 */
        System.out.println("a ^ b = " + c );

        c = a << 1;         /* 120 = 0111 1000 */
        System.out.println("a << 1 = " + c );

        c = (a << 1) + 1;   /* 121 = 0111 1001 */
        System.out.println("(a << 1) + 1 = " + c );

        c = a << 2;         /* 240 = 1111 0000 */
        System.out.println("a << 2  = " + c );
    }
}
```

codeword 부여하기

assignCodewords(theRoot, 0, 0)
으로 호출한다.

노드 p에 부여된 codeword

노드 p에 부여된
codeword의 길이

```
private void assignCodewords(Run p, int codeword, int length) {
    if (p.left == null && p.right == null) {
        p.codeword    = codeword;
        p.codewordLen = length;
    }
    else {
        assignCodewords(                                    );
        assignCodewords(                                    );
    }
}
```

왼쪽 자식노드에게는 codeword의 뒤에 0을 추가하고,
오른쪽 자식에게는 1을 추가한다. 길이는 1 증가한다.

main과 compressFile 메서드

```
public class HuffmanCoding {

    …

    public void compressFile(RandomAccessFile fIn) {
        collectRuns(fIn);
        createHuffmanTree();
        assignCodewords(theRoot, 0, 0);
    }

    static public void main (String args[]) {
        HuffmanCoding app = new HuffmanCoding();
        RandomAccessFile fIn;
        try {
            fIn = new RandomAccessFile(“sample.txt”,”r");
            app.compressFile(fIn);
            fIn.close();
        } catch (IOException io) {
            System.err.println("Cannot open " + fileName);
        }
    }
}
```

제4단계

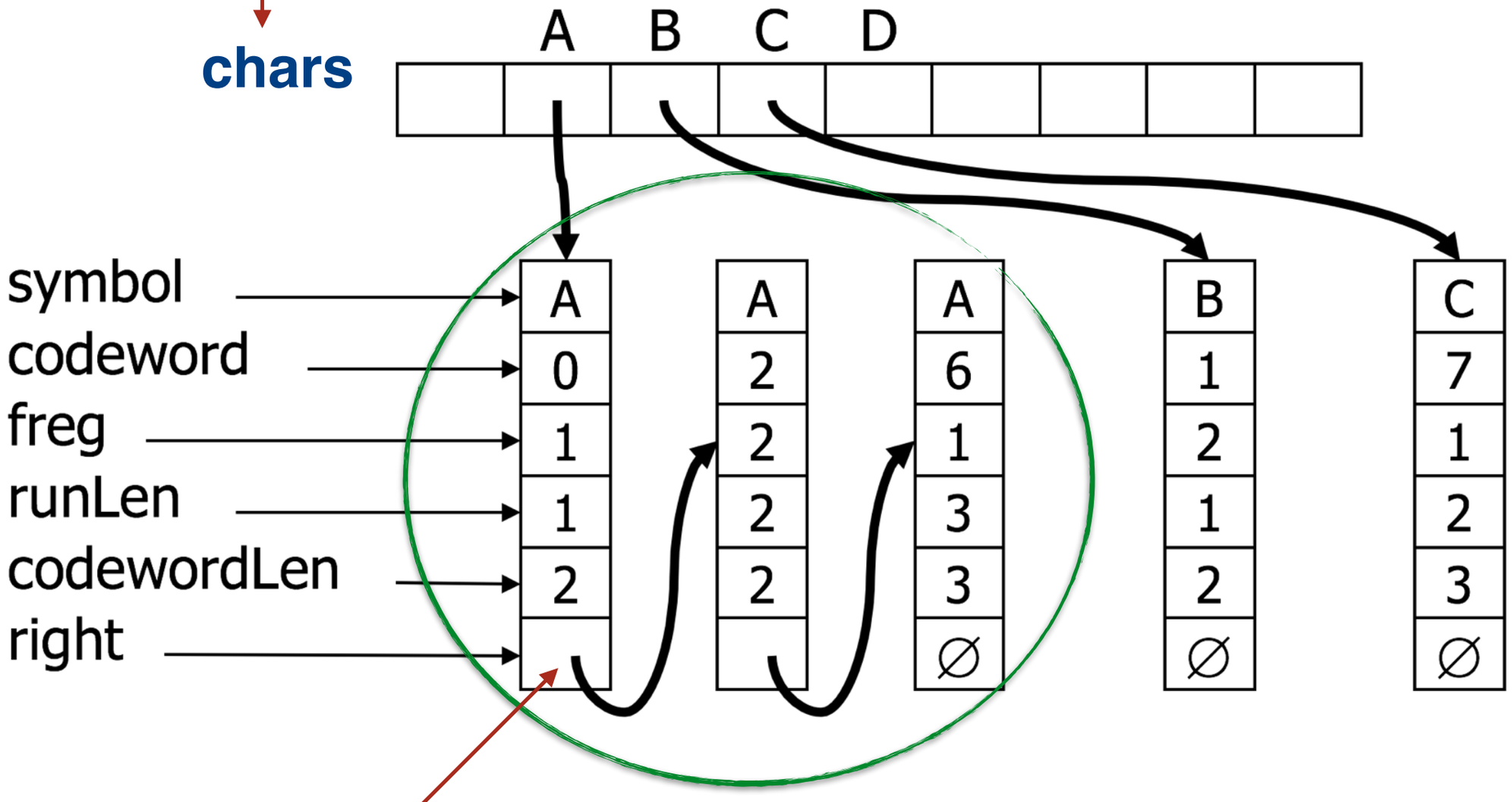
Codeword 검색하기

인코딩

- 데이터 파일을 압축하기 위해서는 데이터 파일을 다시 시작부터 읽으면서 run을 하나씩 인식한 후 해당 run에 부여된 codeword를 검색한다.
- Huffman트리에는 모든 run들이 리프노드에 위치하므로 검색하기 불편하다.
- 검색하기 편리한 구조를 만들어야 한다.

Array of Linked Lists

크기가 256인 배열



symbol이 동일한 run들을 하나의 연결리스트로 저장한다.

각 run의 right 필드를 다음 노드를 가리키는 링크필드로 사용한다.

storeRunsIntoArray

```
private Run [] chars = new Run [256];

/* Huffman 트리의 모든 리프노드들을 chars에 recursion으로 저장한다 */
private void storeRunsIntoArray(Run p) {
    if (p.left == null && p.right == null) {
        insertToArray(p);
    }
    else {
        storeRunsIntoArray(p.left);
        storeRunsIntoArray(p.right);
    }
}

public void compressFile(RandomAccessFile fIn) {
    collectRuns(fIn);
    createHuffmanTree();
    assignCodewords(theRoot, 0, 0);
    storeRunsIntoArray(theRoot);
}
```

배열 chars[(unsigned int)p.symbol]가 가리키는
연결리스트의 맨 앞에 p를 삽입한다.

Run 검색하기

- symbol과 runLength가 주어질 때 배열 chars를 검색하여 해당하는 run을 찾아 반환하는 메서드를 작성한다.

```
public Run findRun(byte symbol, int length) {

/* 배열 chars에서 (symbol, length)에 해당하는 run을 찾아 반환한다. */

}
```

제5단계

인코딩하기

인코딩

- 압축파일의 맨 앞부분(header)에 파일을 구성하는 run들에 대한 정보를 기록한다.
- 이때 원본 파일의 길이도 함께 기록한다 (왜 필요할까?)

outputFrequencies

fIn은 압출할 파일, fOut은 압축된 파일이다.

```
private void outputFrequencies(RandomAccessFile fIn,
                               RandomAccessFile fOut) throws IOException {

    fOut.writeInt(runs.size());
```

먼저 run의 개수를 하나의 정수로 출력한다.

```
    fOut.writeLong(fIn.getFilePointer());
```

원본 파일의 크기(byte단위)를 출력한다.

```
    for (int j = 0; j < runs.size(); j++) {
        Run r = runs.get(j);
        fOut.write(r.symbol); // write a byte
        fOut.writeInt(r.runLen);
        fOut.writeInt(r.freq);
    }
}
```

각각의 run들을 출력한다.

compressFile

fIN은 압축할 파일, inFileName은 그 파일의 이름이다. 파일의 이름을 추가로 받는 이유는 압축된 파일의 이름을 정하기 위해서이다.

```
public void compressFile(String inFileName, RandomAccessFile fIn)
                        throws IOException {

    String outFileName = new String(inFileName+".z");
```

압축파일의 이름은 압축할 파일의 이름에 확장자를 .z를 붙인 것이다.

```
    RandomAccessFile fOut = new RandomAccessFile(outFileName,"rw");
```

압축파일을 여기서 생성하여 outputFrequencies와 encode메서드에게 제공한다.

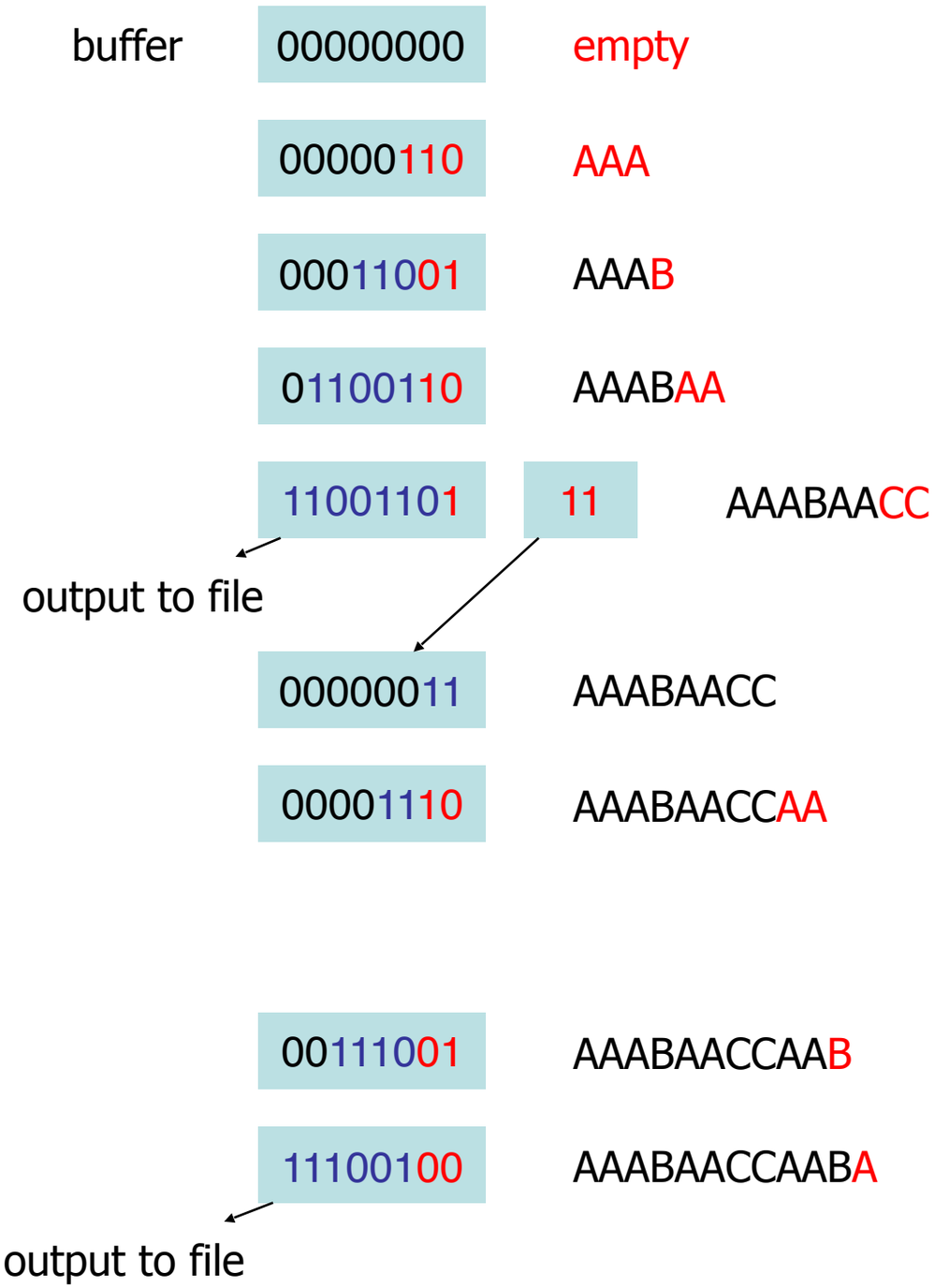
```
    collectRuns(fIn);
    outputFrequencies(fIn,fOut);
    createHuffmanTree();
    assignCodewords(theRoot, 0, 0);
    storeRunsIntoArray(theRoot);
    fIn.seek(0);
    encode(fIn, fOut);
}
```

main

```
public class HuffmanCoding {

    …

    public void compressFile(String inFileName, RandomAccessFile fIn)
                        throws IOException {
       …
    }

    static public void main (String args[]) {
        HuffmanCoding app = new HuffmanCoding();
        RandomAccessFile fIn;
        try {
            fIn = new RandomAccessFile(“sample.txt”,”r");
            app.compressFile(“sample.txt”, fIn);
            fIn.close();
        } catch (IOException io) {
            System.err.println("Cannot open " + fileName);
        }
    }
}
```

encode()

- encode을 위하여 하나의 buffer를 사용한다.



AAA	110
AA	10
A	00
B	01
CC	111

이 그림에서는 buffer를 1byte로 묘사하였으나 실제로는 32비트 정수를 사용하여 구현한다.

encode

```
private void encode(RandomAccessFile fIn, RandomAccessFile fOut)  {
    while there remains bytes to read in the file {
        recognise a run;
        find the codeword for the run;
        pack the codeword into the buffer;
        if the buffer becomes full
            write the buffer into the compressed file;
    }
    if buffer is not empty {
        append 0s into the buffer;
        write the buffer into the compressed file;
    }
}
```

class HuffmanEncoder

```
public class HuffmanEncoder {
    static public void main (String args[]) {
        String fileName = "";
        HuffmanCoding app = new HuffmanCoding();
        RandomAccessFile fIn;
        Scanner kb = new Scanner(System.in);
        try {
            System.out.print("Enter a file name: ");
            fileName = kb.next();
            fIn = new RandomAccessFile(fileName,"r");
            app.compressFile(fileName,fIn);
            fIn.close();
        } catch (IOException io) {
            System.err.println("Cannot open " + fileName);
        }
    }
}
```

제6단계
디코딩하기

class HuffmanDecoder

```
public class HuffmanDecoder {
    static public void main (String args[]) {
        String fileName = "";
        HuffmanCoding app = new HuffmanCoding();
        RandomAccessFile fIn;
        Scanner kb = new Scanner(System.in);
        try {
            System.out.print("Enter a file name: ");
            fileName = kb.next();
            fIn = new RandomAccessFile(fileName,"r");
            app.decompressFile(fileName,fIn);
            fIn.close();
        } catch (IOException io) {
            System.err.println("Cannot open " + fileName);
        }
    }
}
```

decompressFile

```
public void decompressFile(String inFileName, RandomAccessFile fIn)
                        throws IOException {
    String outFileName = new String(inFileName+".dec");
    RandomAccessFile fOut = new
                        RandomAccessFile(outFileName,"rw");
    inputFrequencies(fIn);
    createHuffmanTree();
    assignCodewords(theRoot,0,0);
    decode(fIn,fOut);
}
```

inputFrequencies

```
private void inputFrequencies(RandomAccessFile fIn)
                          throws IOException {
    int dataIndex = fIn.readInt();
    sizeOriginalFile = fIn.readLong();
    runs.ensureCapacity(dataIndex);
    for (int j = 0; j < dataIndex; j++) {
        Run r = new Run();
        r.symbol = (byte) fIn.read();
        r.runLen = fIn.readInt();
        r.freq = fIn.readInt();
        runs.add(r);
    }
}
```

이 메서드가 속한 class HuffmanCoding에 long타입의 변수 sizeOriginalFile을 멤버로 추가한다. 이것은 원본 파일의 길이다. 이 값은 decode메서드에서 사용된다.

decode

```
private void decode(RandomAccessFile fIn, RandomAccessFile fOut)
                                    throws IOException {
    int nbrBytesRead=0, j, ch, bitCnt = 1, mask = 1, bits = 8;
    mask <<= bits - 1;  // change 00000001 to 100000000
    for (ch=fIn.read(); ch!=-1 && nbrBytesRead<sizeOriginalFile;) {
        Run p = theRoot;
        while(true) {
            if (p.left == null && p.right == null) {
                for (j = 0; j < p.runLen; j++)
                    fOut.write(p.symbol);
                nbrBytesRead += p.runLen;
                break;
                }
                else if ((ch & mask) == 0)  /* if the most significant bit is 0 */
                    p = p.left;
                else                        /* if the most significant bit is 1 */
                    p = p.right;
                if (bitCnt++ == bits) {     /* if done with the current byte */
                    ch = fIn.read();
                    bitCnt = 1;
                }
                else
                    ch <<= 1;               /* left-shift the current byte */
        }
    }
}
```