Implementing the Huffman Coding algorithm

Gagarine Yaikhom

In this note we implement the minimal prefix coding algorithm by David Huffman described in *A Method for the Construction of Minimum-Redundancy Codes* [Proceedings of the IRE 40 (9): 1098–1101 (1952)]. The algorithm itself is quite simple to understand; however, in this note we are particularly interested in the data structures.

In this implementation, we use an array-based data structure for representing binary trees, which takes advantage of the manner in which the coding tree is built from the bottom-up. I came across this representation while studying *Dynamic Huffman Coding* algorithm by Donald E. Knuth [Journal of Algorithms, 6, 163-180 (1985)], which we shall discuss in a subsequent note.

It is worth noting that the Huffman coding algorithm is very similar to agglomerative hierarchical clustering, discussed in one of my previous notes. The only difference is the manner in which the nodes to be combined are chosen at each stage of the tree building phase. In agglomerative hierarchical clustering, we choose nodes that are closest to one another; whereas, in Huffman coding, we choose the nodes with the lowest weights.

I. Usage

To test the implementation, compile the source code and run the program as follows:

```
$ make huffman
$ ./huffman encode message.txt encoded.dat;
$ ./huffman decode encoded.dat decoded.txt;
$ diff message.txt decoded.txt
```

A diagrammatic visualisation of this implementation as it is being executed is provided at the end of this note.

II. The source

We shall encode the input data stream one byte at a time. Hence, we shall use the frequencies for `num_alphabets` = 256 alphabets. Since all of the alphabets may not appear in the input data stream, we shall maintain the number of active alphabets `num_active` that do appear in the input data stream, and only transmit the frequencies for those to the receiver.

The frequencies of the alphabets are maintained in the `frequency` array. To know when to stop decoding, we shall also send `original_size`, the number of bytes in the original input data stream. This avoids assigning a special end-of-file meaning to one of the bytes. Note that `original_size` and size of each `frequency` element limits the maximum size of the input data stream that may be encoded.

⟨*Data structures*⟩≡

```
int num_alphabets = 256;
int num_active = 0;
int *frequency = NULL;
unsigned int original_size = 0;
```

The function `determine_frequency()` calculates the frequency for each of the alphabets by parsing the input data stream completely. It also counts the number of bytes in the stream. Note that this pre-processing step is required for classic Huffman coding. In a future note, we shall discuss Knuth's dynamic Huffman coding algorithm which does not require this pre-processing step.

⟨*Function definitions*⟩≡

```
void determine_frequency(FILE *f) {
    int c;
    while ((c = fgetc(f)) != EOF) {
        ++frequency[c];
        ++original_size;
    }
    for (c = 0; c < num_alphabets; ++c)
        if (frequency[c] > 0)
            ++num_active;
}
```

For each node in the binary coding tree, we shall maintain two values. An index to identify the node, and a weight assigned to that node. The weight of a leaf node equals the corresponding alphabet frequency; the weight of an internal node equals the sum of the weights of its children. To differentiate between leaf nodes and internal nodes, we use the sign of the index. Hence, a positive index means internal node; whereas, a negative index means a leaf (alphabet) node. Further to this, the index of a leaf node is $-(v+1)$, where v is the byte-value of the alphabet. We add one so that the index is not zero. On the other hand, for the internal nodes, the index gives the order in which the node was inserted to the coding tree. The indexes for internal nodes are consecutive and begins at 1.

⟨*Data structures*⟩+≡
```
typedef struct {
    int index;
    unsigned int weight;
} node_t;
```

The maximum number of nodes for the binary coding tree is constrained by the number of active alphabets. We maintain the nodes that are part of the tree in `nodes`, which grows as existing nodes are combined to form new nodes, as dictated by the Huffman algorithm. The number of nodes currently in the tree is maintained in `num_nodes`. For each alphabet, we also maintain a mapping from its byte value to a node index assigned to it. Furthermore, for each node in the coding tree, we also maintain its parent indices, which are described further in subsequent sections.

⟨*Data structures*⟩+≡
```
node_t *nodes = NULL;
int num_nodes = 0;
int *leaf_index = NULL;
int *parent_index = NULL;
```

We initialise the program by allocating space for maintaining the alphabet frequencies, and the leaf-to-node index look-up table, since they are already known in advance irrespective of the input data stream. We ensure that `leaf_index` is adjusted to match indexing that begins at 1. We defer allocation of space for the coding tree itself until the frequencies are know.

⟨*Function definitions*⟩+≡
```
void init() {
    frequency = (int *)
        calloc(2 * num_alphabets, sizeof(int));
    leaf_index = frequency + num_alphabets - 1;
}
```

Allocate space for the coding tree nodes and parent index look-up table. Due to the manner in which the coding tree is built, we can assert that every internal node has two children. This makes the coding tree a *full binary tree*. For a full binary tree with n leaves (i.e., number of active alphabets) there are $2n-1$ nodes; and therefore, $n-1$ internal nodes. Because indexing begins at 1, we allocate one extra space, discarding elements with index 0.

⟨*Function definitions*⟩+≡
```
void allocate_tree() {
    nodes = (node_t *)
        calloc(2 * num_active, sizeof(node_t));
    parent_index = (int *)
        calloc(num_active, sizeof(int));
}
```

Once we are done encoding or decoding, free the resources.

⟨*Function definitions*⟩+≡
```
void finalise() {
    free(parent_index);
    free(frequency);
    free(nodes);
}
```

To build the binary coding tree bottom-up, we combine existing nodes that have the lowest weights. Function `add_node()` adds a new node to the tree with the supplied node index and weight. This function is called by the tree builder. Note that all of the existing nodes are maintained in a non-decreasing order by weight. This allows us to pick the two nodes with the lowest weights by simply picking the two adjacent nodes with the lowest indices that have not yet been combined.

⟨*Function definitions*⟩+≡
```
int add_node(int index, int weight) {
    int i = num_nodes++;
    ⟨Move existing nodes with larger weights to the right⟩
    ⟨Add new node to its rightful place⟩
    return i;
}
```

We move the nodes one-by-one. This is so that, as the tree is being built, the indices of the nodes assigned when they were first added to the tree will have to be adjusted depending on the developing structure of the tree. We go through each of the nodes that have been moved and update their index in the corresponding look-up table with the new location. Note that adjusting leaf node indices requires index negation.

⟨*Move existing nodes with larger weights to the right*⟩≡
```
while (i > 0 && nodes[i].weight > weight) {
    memcpy(&nodes[i + 1], &nodes[i], sizeof(node_t));
    if (nodes[i].index < 0)
        ++leaf_index[-nodes[i].index];
    else
        ++parent_index[nodes[i].index];
    --i;
}
```

We now place the new node in its rightful place, preserving the non-decreasing weight order.

⟨*Add new node to its rightful place*⟩≡
```
++i;
nodes[i].index = index;
nodes[i].weight = weight;
if (index < 0)
    leaf_index[-index] = i;
else
    parent_index[index] = i;
```

Before we build the tree, we first insert all of the leaves for which we already know the weights. Since function `add_node()` maintains existing nodes in a non-decreasing order by weights, we do not need to explicitly sort the leaves. We only insert alphabets that appear in the input data stream as new leaves, and ignore inactive alphabets. Each of the leaf nodes are assigned an index value that corresponds to the alphabet's byte value, ensuring that they are non-zero and negative.

⟨*Function definitions*⟩+≡
```
void add_leaves() {
    int i, freq;
    for (i = 0; i < num_alphabets; ++i) {
        freq = frequency[i];
        if (freq > 0)
            add_node(-(i + 1), freq);
    }
}
```

We are now ready to build the tree by combining the leaf nodes and then combining the internal nodes, until only the root node is left. To track the next two nodes to combine, we maintain the next index to use in `free_index`. At the beginning, we start at the first node.

⟨*Data structures*⟩+≡
```
int free_index = 1;
```

When there are node pairs that could be combined, we create a new node with the pair as children, and add the new node to the tree. Notice here that because we always combined two consecutive free nodes, the parents for both these nodes can be maintain in the shared `parent_index` table using a single value. We store the same parent index in the node, which does not have a special meaning for internal nodes, so that node indices can be updated as the tree structure changes. At the end of this, we have the complete binary coding tree.

⟨*Function definitions*⟩+≡
```
void build_tree() {
    int a, b, index;
    while (free_index < num_nodes) {
        a = free_index++;
        b = free_index++;
        index = add_node(b/2,
            nodes[a].weight + nodes[b].weight);
        parent_index[b/2] = index;
    }
}
```

III. Encoding

We are now ready to encode the input data stream. We read the input bytes from `ifile` and write the encoded bit stream to `ofile`. We first determine the frequencies, write the header data with these frequencies so that the receiver can rebuild the tree for decoding. We then build the rest of the tree. Using this tree, we start encoding from the start of the input data stream.

⟨*Function definitions*⟩+≡
```
int encode(const char* ifile, const char *ofile) {
    FILE *fin, *fout;
    ⟨Open input and output files⟩
    determine_frequency(fin);
    ⟨Allocate space for coding tree and bit stack⟩
    add_leaves();
    write_header(fout);
    build_tree();
    fseek(fin, 0, SEEK_SET);
    int c;
    while ((c = fgetc(fin)) != EOF)
        encode_alphabet(fout, c);
    flush_buffer(fout);
    free(stack);
    ⟨Close files⟩
    return 0;
}
```

⟨*Close files*⟩≡
```
fclose(fin);
fclose(fout);
```

⟨*Open input and output files*⟩≡
```
if ((fin = fopen(ifile, "rb")) == NULL) {
    perror("Failed to open input file");
    return FILE_OPEN_FAIL;
}
if ((fout = fopen(ofile, "wb")) == NULL) {
    perror("Failed to open output file");
    fclose(fin);
    return FILE_OPEN_FAIL;
}
```

For each byte from the input data stream, we emit the bits encoding the alphabet. This bit is written to `fout`. Since byte decoding starts at the root and traverses left or right depending on the bit value, we must encode the bits to allow this. However, since we start encoding at the leaves moving towards the root, we cannot emit the bits directly. Instead, we use a stack to reverse the order of the bits.

⟨*Data structures*⟩+≡
```
int *stack;
int stack_top;
```

⟨*Allocate space for coding tree and bit stack*⟩≡
```
stack = (int *) calloc(num_active - 1, sizeof(int));
allocate_tree();
```

We first get the node index assigned to the alphabet. Then, we use the `parent_index` table to move up the tree, retrieving the new parent using the index stored in the internal nodes. The value of the bit to be emitted is simply encoded by whether the index assigned to the internal node is either odd or even. Remember that we always combine consecutive nodes, and hence, left children will always have odd indices and right children will always have even indices. Thus, if we are left child, we emit a `1`; or a `0`, otherwise.

⟨*Function definitions*⟩+≡
```
void encode_alphabet(FILE *fout, int character) {
    int node_index;
    stack_top = 0;
    node_index = leaf_index[character + 1];
    while (node_index < num_nodes) {
        stack[stack_top++] = node_index % 2;
        node_index = parent_index[(node_index + 1) / 2];
    }
    while (--stack_top > -1)
        write_bit(fout, stack[stack_top]);
}
```

IV. Decoding

To decode Huffman encoded bit stream, we first need to determine the alphabet frequency. This information is provided in the header of the input file. We therefore read the header file, and use the frequencies to build the binary coding tree. Once the tree is build, we process the remaining input bit stream until all of the bytes in the original byte stream has been decoded.

⟨*Function definitions*⟩+≡
```
int decode(const char* ifile, const char *ofile) {
    FILE *fin, *fout;
    ⟨Open input and output files⟩
    if (read_header(fin) == 0) {
        build_tree();
        decode_bit_stream(fin, fout);
    }
    ⟨Close files⟩
    return 0;
}
```

To decode the bit-stream, we retrieve one bit at a time and start at the top of the coding tree, the *root*. Depending on the bit value, we decide whether to visit the left subtree or the right subtree. Remember that the left subtree will always have an odd index, and the right tree an even index. Also remember that the `index` in each node stores the sequence when the node was inserted to the tree, which is used for node index look-up.

⟨*Function definitions*⟩+≡
```
void decode_bit_stream(FILE *fin, FILE *fout) {
    int i = 0, bit, node_index = nodes[num_nodes].index;
    while (1) {
        bit = read_bit(fin);
        if (bit == -1)
            break;
        node_index = nodes[node_index * 2 - bit].index;
        if (node_index < 0) {
            char c = -node_index - 1;
            fwrite(&c, 1, 1, fout);
            if (++i == original_size)
                break;
            node_index = nodes[num_nodes].index;
        }
    }
}
```

V. Reading and writing bits

To read and write bits to a file, we shall use a bit buffer that packs bits into bytes. The number of bytes in the buffer are given by `MAX_BUFFER_SIZE`. To mark failure to read or write bits from a file, we us the following constants.

⟨*Constant declarations*⟩≡
```
#define MAX_BUFFER_SIZE 256
#define INVALID_BIT_READ -1
#define INVALID_BIT_WRITE -1
```

We also maintain the number of bits in the buffer, and the current bit position to use when reading bits from the buffer.

⟨*Data structures*⟩+≡
```
unsigned char buffer[MAX_BUFFER_SIZE];
int bits_in_buffer = 0;
int current_bit = 0;
```

The function `write_bit` writes a bit value `bit` to file `f`. We first check if the buffer is full. If it is, we write the buffer to the file. Otherwise, the bit is packed to the existing buffer.

⟨*Function definitions*⟩+≡
```
int write_bit(FILE *f, int bit) {
    if (bits_in_buffer == MAX_BUFFER_SIZE << 3) {
        size_t bytes_written =
            fwrite(buffer, 1, MAX_BUFFER_SIZE, f);
        if (bytes_written < MAX_BUFFER_SIZE && ferror(f))
            return INVALID_BIT_WRITE;
        bits_in_buffer = 0;
        memset(buffer, 0, MAX_BUFFER_SIZE);
    }
    if (bit)
        buffer[bits_in_buffer >> 3] |=
            (0x1 << (7 - bits_in_buffer % 8));
    ++bits_in_buffer;
    return SUCCESS;
}
```

If the bits sent to the buffer stops before the buffer is fully packed, we must ensure that the bits in the buffer are written to the file. Function `flush_buffer()` does precisely that. It writes the bits in the buffer, packing them into the nearest bytes. If the bits are not fully packed, the bits are zero-padded. Since we maintain the number of bytes in the original input data stream, we need not worry about over-reading of padding bits.

⟨*Function definitions*⟩+≡
```
int flush_buffer(FILE *f) {
    if (bits_in_buffer) {
        size_t bytes_written =
            fwrite(buffer, 1,
                (bits_in_buffer + 7) >> 3, f);
        if (bytes_written < MAX_BUFFER_SIZE && ferror(f))
            return -1;
        bits_in_buffer = 0;
    }
    return 0;
}
```

To prevent bit reads from input files already at the end-of-file, we maintain the following Boolean state which is set to `1` when not more bits are available.

⟨*Data structures*⟩+≡
```
int eof_input = 0;
```

The function `read_bit()` reads a bit from the file `f`. We read a full buffer at a time, and return bits from this buffer. The buffer is replenished with more bits from the file when empty.

⟨*Function definitions*⟩+≡
```
int read_bit(FILE *f) {
    ⟨Fill buffer with bytes from file⟩
    if (bits_in_buffer == 0)
        return END_OF_FILE;
    int bit = (buffer[current_bit >> 3] >>
        (7 - current_bit % 8)) & 0x1;
    ++current_bit;
    return bit;
}
```

⟨*Fill buffer with bytes from file*⟩≡
```
if (current_bit == bits_in_buffer) {
    if (eof_input)
        return END_OF_FILE;
    else {
        size_t bytes_read =
            fread(buffer, 1, MAX_BUFFER_SIZE, f);
        if (bytes_read < MAX_BUFFER_SIZE) {
            if (feof(f))
                eof_input = 1;
        }
        bits_in_buffer = bytes_read << 3;
        current_bit = 0;
    }
}
```

VI. Communicating the alphabet frequencies

The classic Huffman coding requires transmitting the frequencies of the alphabets. This is written to the encoded file as a header. The header has the following structure:

Field	Size	Description
`original_size`	`sizeof(int)`	Number of bytes in the original data stream.
`num_active`	1 byte	Number of active alphabets.
Alphabet frequency	1 byte + `sizeof(int)`	First byte gives alphabet code; the rest gives the frequency.

We should use fixed length fields with proper byte-ordering; however, we have simplified it here for brevity. Furthermore, for real-life files, the frequency of alphabets can be much higher than stipulated here. Hence, in production implementations the header should take these limitations into account.

Since `write_header()` uses the `nodes` array, it is important to write this header after inserting all of the leaves, but before building the coding tree. This is necessary because once the tree starts to build, the leaves may move to the right to preserve non-decreasing weights, and therefore may not be consecutive.

⟨*Function definitions*⟩+≡

```
int write_header(FILE *f) {
    int i, j, byte = 0,
        size = sizeof(unsigned int) + 1 +
             num_active * (1 + sizeof(int));
    unsigned int weight;
    char *buffer = (char *) calloc(size, 1);
    if (buffer == NULL)
        return MEM_ALLOC_FAIL;

    j = sizeof(int);
    while (j--)
        buffer[byte++] =
            (original_size >> (j << 3)) & 0xff;
    buffer[byte++] = (char) num_active;
    for (i = 1; i <= num_active; ++i) {
        weight = nodes[i].weight;
        buffer[byte++] =
            (char) (-nodes[i].index - 1);
        j = sizeof(int);
        while (j--)
            buffer[byte++] =
                (weight >> (j << 3)) & 0xff;
    }
    fwrite(buffer, 1, size, f);
    free(buffer);
    return 0;
}
```

When decoding the encoded file, we simply read back the header and retrieve the active alphabets and corresponding frequencies. Once this is done, we can rebuild the coding tree.

⟨*Function definitions*⟩+≡

```
int read_header(FILE *f) {
    int i, j, byte = 0, size;
    size_t bytes_read;
    unsigned char buff[4];

    bytes_read = fread(&buff, 1, sizeof(int), f);
    if (bytes_read < 1)
        return END_OF_FILE;
    byte = 0;
    original_size = buff[byte++];
    while (byte < sizeof(int))
        original_size =
            (original_size << (1 << 3)) | buff[byte++];

    bytes_read = fread(&num_active, 1, 1, f);
    if (bytes_read < 1)
        return END_OF_FILE;

    allocate_tree();

    size = num_active * (1 + sizeof(int));
    unsigned int weight;
    char *buffer = (char *) calloc(size, 1);
    if (buffer == NULL)
        return MEM_ALLOC_FAIL;
    fread(buffer, 1, size, f);
    byte = 0;
    for (i = 1; i <= num_active; ++i) {
        nodes[i].index = -(buffer[byte++] + 1);
        j = 0;
        weight = (unsigned char) buffer[byte++];
        while (++j < sizeof(int)) {
            weight = (weight << (1 << 3)) |
                (unsigned char) buffer[byte++];
        }
        nodes[i].weight = weight;
    }
    num_nodes = (int) num_active;
    free(buffer);
    return 0;
}
```

⟨*Function definitions*⟩+≡

```
void print_help() {
     fprintf(stderr,
         "USAGE: ./huffman [encode | decode] "
         "<input out> <output file>\n");
}
```

⟨*The main program*⟩≡
```
int main(int argc, char **argv) {
    if (argc != 4) {
        print_help();
        return FAILURE;
    }
    init();
    if (strcmp(argv[1], "encode") == 0)
        encode(argv[2], argv[3]);
    else if (strcmp(argv[1], "decode") == 0)
        decode(argv[2], argv[3]);
    else
        print_help();
    finalise();
    return SUCCESS;
}
```

⟨*Constant declarations*⟩+≡
```
#define FAILURE 1
#define SUCCESS 0
#define FILE_OPEN_FAIL -1
#define END_OF_FILE -1
#define MEM_ALLOC_FAIL -1
```

⟨*Include libraries*⟩≡
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

⟨*Function declarations*⟩≡
```
int read_header(FILE *f);
int write_header(FILE *f);
int read_bit(FILE *f);
int write_bit(FILE *f, int bit);
int flush_buffer(FILE *f);
void decode_bit_stream(FILE *fin, FILE *fout);
int decode(const char* ifile, const char *ofile);
void encode_alphabet(FILE *fout, int character);
int encode(const char* ifile, const char *ofile);
void build_tree();
void add_leaves();
int add_node(int index, int weight);
void finalise();
void init();
```

⟨*huffman*⟩≡
⟨*Include libraries*⟩
⟨*Constant declarations*⟩
⟨*Data structures*⟩
⟨*Function declarations*⟩
⟨*Function definitions*⟩
⟨*The main program*⟩

VII. Visualising the algorithm

Let us take the example message `abracadabra` where ϕ is the end of message symbol. We get the following frequencies:

Byte	Symbol	Frequency
10	ϕ	1
97	a	5
98	b	2
99	c	1
100	d	1
114	r	2

From these frequencies, our aim is to build the binary coding tree as shown in Figure 1.

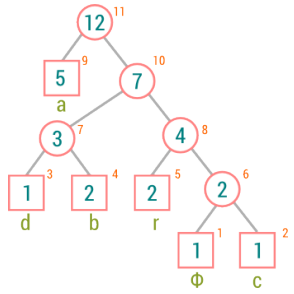

Fig. 1. Binary coding tree.

Each of the symbols form a leaf. We therefore insert these into the list of available nodes before we start building the tree. The leaves are inserted in the order of their byte value, adjusting the arrangement so that the weights are non-decreasing. Figure 2 shows the tree as we insert the first three leaves. Notice how the index of node `a` gets updated as it is moved to the right.

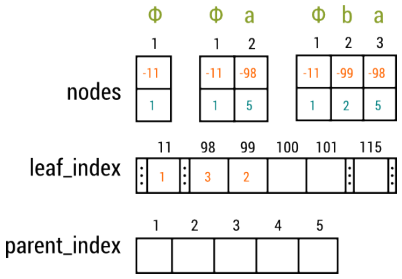

Fig. 2. State after inserting leaves ϕ, `a` and `b`.

Once all of the leaves have been inserted, we get the state of the data structures as shown in Figure 3. Notice how the nodes and indices are related. Also notice that the node index for the leaves are negative and one more than their byte value.

We can now build the coding tree by combining free nodes with the lowest weights, as shown in the following figures.

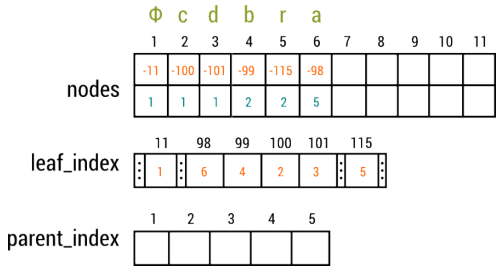

Fig. 3. State after leaves have been inserted.

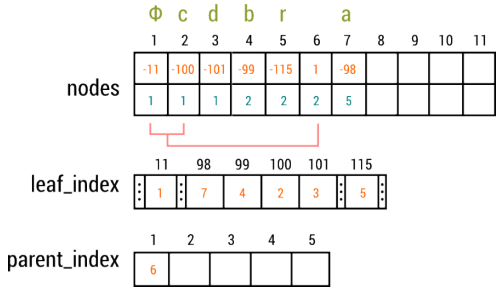

Fig. 4. State after combining ϕ and c.

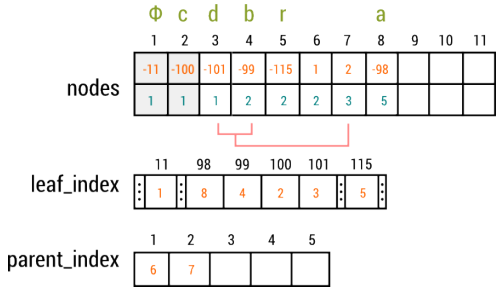

Fig. 5. State after combining d and b.

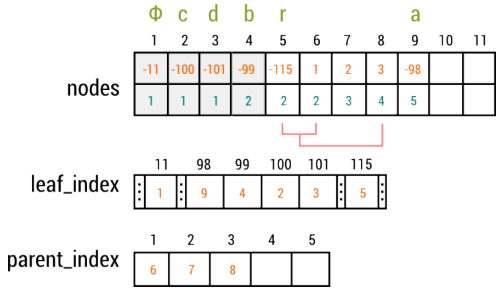

Fig. 6. State after combining r and 1.

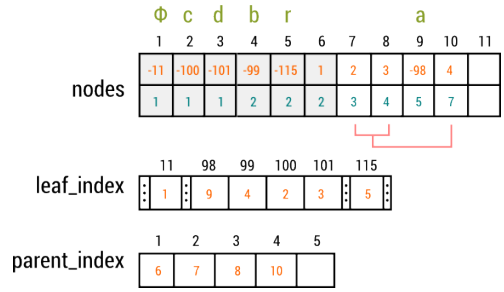

Fig. 7. State after combining 2 and 3.

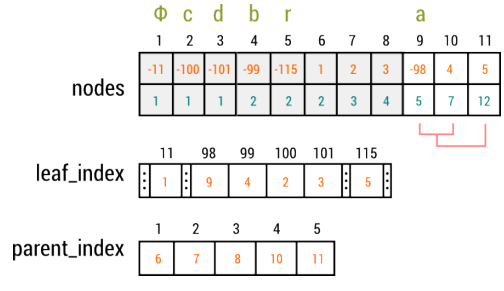

Fig. 8. State after combining a and 4.