# Имплементация Хаффмана

**Гэгэрин Якихом**

В этой заметке мы реализуем алгоритм минимального префиксного кодирования *A Method for the Construction of Minimum-Redundancy Codes* [Proceedings of the IRE 40 (9): 1098–1101 (1952)]. Сам алгоритм довольно просто понять; однако в этой заметке что нас особенно интересуют структуры данных. В этой реализации мы используем структуры данных на основе массива для представления двоичных деревьев, в котором используется способы построения дерева кодирования снизу вверх. Я столкнулся с этим представлением во время изучения *Dynamic Huffman Coding* Дональда Э. Кнута [Journal of Algorithms, 6, 163-180 (1985)], которые мы обсудим в последующее примечание. Стоит отметить, что алгоритм кодирования Хаффмана очень аналогично агломеративной иерархической кластеризации, обсуждаемой в одна из моих предыдущих заметок. Единственная разница - это способ каких узлов для объединения выбираются на каждом этапе фазы построения дерева. В агломеративной иерархической кластеризации выбираем наиболее близких к друг к другу узлы; тогда как в кодированием Хаффмана выбираем узлы с наименьшими весами.

# 1 Математические формулы

Текст имплементации, пуска комплектаций:

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

Диаграммная визуализация этой имплементаций.

# 2 Источник решении

В нашем случаи поток входных данных по одному байту за раз. С использованием постоянной частности для ‘‘num_alphabets’’ = 256 алфавитами. Поскольку все кодирующийся алфавиты могут не отображаться в потоке входных данных. Есть возможность активировать затронутые алфавиты ‘‘num_active’’ которые не выявляются во входном потоке данных, и передавать только частоты для приемника. Имеется в виду что потоки ‘‘frequency’’ алфавитов остаются минимальным размере ‘‘orginal_size’’ и размера каждого потока элиминируются и лимитируется максимальным размером входного потока данных, который может быть закодирован.

```
<Data structures>+≡
int num_alphabets = 256;
int num_active = 0;
int *frequency = NULL;
unsigned int original_size = 0;
```

Функция ‘‘determine_frequency()’’ измеряет каждый входной алфавитный поток и анализируя поток входных данных полностью. А также считает количество битов в потоке. Нужно иметь виду что это решения предварительной обработки требуется для классического Кодирование Хаффмана. В будущем мы обсудим алгоритм динамического кодирования Хаффмана, который не требует этот этап предварительной обработки.

```
1.  <Function definitions>+≡
2.  void determine_frequency(FILE *f) {
3.  int c;
4.  while ((c = fgetc(f)) != EOF) {
5.  ++frequency[c];
6.  ++original_size;
7.  }
8.  for (c = 0; c < num_alphabets; ++c)
9.  if (frequency[c] > 0)
10.  ++num_active;
11.  }
```

Для каждого узла бинарных веток кодирование, будет составляется из 2 значении. Индекс для расшифровки узлов, и подходящий размер к узлу. Размер веточного узла равен соответственному потоку алфавита; вес продлившего узла равен равный суммой сортировок. Чтобы разделить веточные узлы и продливший узлы, с использованием знаком индекса. То бишь, позитивный индекс обозначается как продливший узел; там где негативный индекс равен веточному (алфавитному) узлу. Дале индексного веточного узла имеет свойства $-(v + 1)$, где v является `byte-value` алфавита. Дополняем единицу для того чтобы индекс не бы равен нолю. С другой страны, для продливших узлов которые выделяет сам индекс для дерево кодирования, и последовательно начинаются с 1.

```
<Data structures>+≡
typedef struct {
int index;
unsigned int weight;
} node_t
```

Максимальное число узлов бинарного дерево кодов состоявшийся на число активных алфавитов ‘‘(num_alphabets)’’. Содержание узлов, которые является часть бинарных узлов, которые состоят из комбинации новых узлов, как сам алгоритм Хаффмана представляет. Число узлов в текущий момент держит состояние в ‘‘num_nodes’’. Для самой функций алфавитов, идет локализация с ее количество битов и парных совпадающих индексных узлов. Дале, для каждого узла в дереве кодирования, мы также парные сортировки.

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

Представляем программу с выделением пробела для постановки алфавитных решении, и таблица веточного решение узлов, по сколку они в продлившим скоплении уже в потоке данных. Подтверждено что этот ‘‘leaf_index’’ будет в положении для избирательных индексов которые начинаются с 1. Сравнительными выделениями пробелов для постановление кодированного дерева.

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

Выделение пробела для ПКД и парная таблица поиска. Является прикладной тактикой установки действии дерево кодирования, утверждено что каждый продливший узел имеет 2 траектории. Переводит дерево кодирования в полное бинарное дерево. Для бинарного дерево с n остатком (число активных алфавитов) есть 2n – 1 узлов; и потом, n – 1 продливших узлов. Потому что функция индекса начинается с 1, из того выделяем один дополнительный пробел с снятым индексным шифрованием 0.

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

Сразу когда кодирование или декодирование закончит процедуру, освобождает данные функции.

```
<Function definitions>+≡
free(parent_index);
free(frequency);
free(nodes);
```

Для установки кодированного бинарного дерево с низу верх, мы совмещаем существенные узлы с минимальным весом. Функция ‘‘add_node()’’ дополняет новый узел для дерево для снаряжённого индексного узла и веса. Эта функция называется выделенная установка деревьев.

Имея виду что все существующие узлы содержаться в не снижаемом распорядке веса.

Этот доставляет проверки 2 парных узлов с снижаемом коэффициент которые еще не были совмещены.

```
<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; }
```

Передвижения узла одним за другим. Причина того, чтобы дерево могло устанавливаться, коэффициенты узлов, которые задаются в первом случаи

для дерево выделяемое зависимости от структурного установление дерево. Пролистывая каждый перемещённый и обновлённый узел которые соответствует таблицы проверки в новом месте. Учитывая что выделяемые веточные узлы индексную навигацию.

```
<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;
}
```

Теперь мы кладем новый узел в свое нужное место, придерживая не пониженный распорядок веса.

```
<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;
```

Прежде чем построить дерево, веточная представление соответствует весу. Поскольку функция ‘‘`add_node()`’’ содержит существенные узлы в не снижаемому распорядка веса, нам не надо постоянно сортировать веточные потоки. Только с добавлением алфавитов которые появляется в потоке данных в новым формате и игнорирует не работающие алфавиты. Каждый веточные узлы выделяют индексное алфавитные равнение в битах, проверяет чтобы они были не 0 и негативные.

```
<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);
}}
```

Мы теперь можем установить дерево с помощью комбинации веточного узла и продлившего узла, до тех пор когда коренной узел будет находится с левой стронны. Чтобы найти следующих два узла добавление, нужно держать следующий индекс с использованием `free_index`. В начале с первым узлом.

```
<Data structures>+≡
Int free_index =1;
```

Когда есть пары узлов которые могут быть комбинированы, создание нового узла с командой ‘‘`pair as children`’’ и добавляем новый узел для дерево. С постоянным добавлением двух последовательных узлов, парные узлы для два добавление остаётся разделённой таблицей единого равнение ‘‘`parent_index`’’. Держим тот же самый парный узел индекса, для того которые не имеет обозначение для продлившего узла, сам коэффициент узла обновляется с изменением дерево. В конечном итоге, у нас есть полное бинарное дерево. `<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;
 }
 }
```

# 3 Кодирование

Теперь можно будет шифровать прибавление в потоке данных. Чтение вкладных битов с “ifile” предписание в поток “ofile”. С первым определением, заголовка должна предписать дополнительные принимающие данные для пере-установочного декодированного дерево. С использованием этого дерево мы начинаем с новой строки шифровать прибавление в поток данных.

```
<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;
}
```

Для каждого бита из потока данных, проверяем частицы кодированных алфавитов. Данная частица предписана для “fout”. Поскольку биты декодировали с корня и пересекая с лева на права от зависимости равнение бит. Изначальное кодирование веточного движение , не возможно переместить эти биты моментально. Из того чтобы предотвратить распорядок накопление битов.

```
<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();
```

Первый подходящий индексный узел к алфавиту. Потом с использованием таблицы ‘‘parent_index’’ для верхнего передвижение дерево, перебираем новые пары с использованием индексного хранение продливших узлов. Равнение бита проскакивает простую функцию кода с решением если индекс подходящий к продлившему узлу нечетный или четный. Добавлением этих последовательных узлов, и левое последствие нечетных индексов а так же правых последовательных узлов с четных индексов. С решением того что левая сторона запускает передвижение 1 и 0.

```
<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]);
```

```
}
```

# 4 Декодирование

Для декатировки битовый поток, мы получаем один бит одновременно и начинаем с высоты кодированного дерева, с ‘‘root’’. Зависимо от битого равнение, проверяем левые или правые ребра дерево. Запоминая что левая ребро будет иметь нечетный индекс, и правая ребро будет четный индекс. Сам ‘‘index’’ в каждом узле сохраняет фиксацию, когда узел совмещен с деревом, с использованием проверки индексного узла для самой проверки.

```
<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;
 }
 }
}
```

# 5 Чтение и писание бит.

Для чтения и писание битов к файлу, надо будет с копить определённую сумму битов, которые собраны в байтах. Число байтов в колонке выделено ‘‘MAX_BUFFER_SIZE’’. Отметить ошибки для чтения и писание бит от кого-либо файла, с проверкой определённого константа.

```
<Constant declarations>≡
#define MAX_BUFFER_SIZE 256
#define INVALID_BIT_READ -1
#define INVALID_BIT_WRITE -1
```

А также держим количество бит в колонке, и данную положение для чтения битов и колонки.

```
<Data structures>+≡
unsigned char buffer[MAX_BUFFER_SIZE];
int bits_in_buffer = 0;
int current_bit = 0;
```

Функция ‘‘write_bit’’ пишет битовое равнение ‘‘bit’’ для файла ‘‘f’’. Проверка если колонка заполнена. Если она заполнена, переписываем этот буфер к файлу. В любом случаи бит переводится в существенный буфер.

```
<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;
}
```

Если эти биты которые отправлены к буферу в полном скопление прерываются, есть повод проверки битов в буфере которые были закреплены в этот файл. Функция ‘‘flush_buffer()’’ проверяет и решает эти ближайшие действие. Если эти биты не полностью скопились, то они находится в нулевом потоке. Если держим число байтов в оригинальном потоке данных, не надо обращать внимание на нулевой поток.

```
<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;
}
```

Для того чтобы предотвратить чтение бит для поток файлов которые находится в конце файлов, настраиваем логический тип которые переводит 1 когда нет больше свободных бит.

```
<Data structures>+≡
int eof_input = 0;
```

Функция ‘‘read_bits()’’ читает биты из файла ‘‘f’’. С чтением полного буфера и возврата бит из буфера. В случаи полного наполнение бит когда файл остаются пустым.

```
<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;
}
}
```

# 6 Коммуникация алфавитных резонанс.

Современное кодирование Хаффмана требует передачу резонансов алфавита. Это указано для кодированного алфавита.

| Поле | Вес | Действие |
|---|---|---|
| `original_size` | `sizeof(int)` | Количество байтов в исходным потоке данных |
| `num_active` | 1byte | Количество активных алфавитов |
| Алфавитный поток | 1byte+`sizeof(int)` | Первый байт дает код алфавита;второй резонанс |

Обработка кодов с рассчитанными сложением подач байтов; но поскольку все в моментальном упрощённом виде. Эти файлы с алфавитами могут быть модулированы предусмотрены в верхней таблице. Сам процесс имплементации может достигнуть самый минимальный уровень.Поскольку код ‘‘`write_header()`’’ использует массив ‘‘`nodes`’’, очень важно выписать головку кода после того как вписаны веточные узлы, перед тем как установить дерево кодирования. Этот процесс очень нуждается в том что когда идет установка дерево веточные узлы могут продвигается в правую часть ветки чтобы не сниженный вес мог быть продвинутым в потоке данных, а не последовательным.

```
<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;
}
```

Если декодировать кодированный файл, проверяем заголовку и получаем активные алфавиты которые подходят к активному резонансу.

```
<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>");
}
<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>
```

# 7 Визуализация алгоритма

Возьмем слово ‘‘abracadbra’’ где $\phi$ символ конец сообщение. Получаем данные:

| Байт | Символ | Резонанс |
|---|---|---|
| 10 | $\phi$ | 1 |
| 97 | a | 5 |
| 98 | b | 2 |
| 99 | c | 1 |
| 100 | d | 1 |
| 114 | r | 2 |

В Fig.1 показывает установку бинарного кодированного дерева.

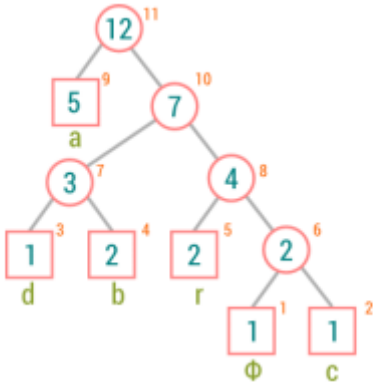


Fig.1

Обозначение каждого веточного символа. Можно будет вставить эти символы в веточных узлах для установки дерево. Веточные узлы могут скопится в случаи постановление количество байтов не совпадением их веса. В Fig.2 показывает само дерево с тремя входных веточных узлов. Проверяем как сам сниженный узел обновляется.

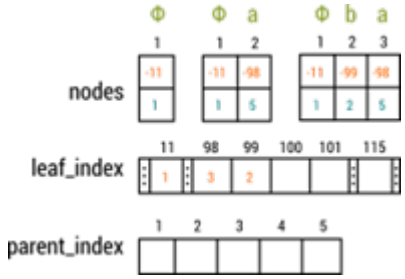


Fig.2 Проход символов $\phi$, a и b.

После того как все веточные узлы были сведены, получаем базу данных, которая представлена в Fig.3. Обращаем внимание что эти векторы узлов и индексов относительно схожи к друг другу с равнением достатка байт. Уставляем комбинации свободных узлов с минимальным весом.

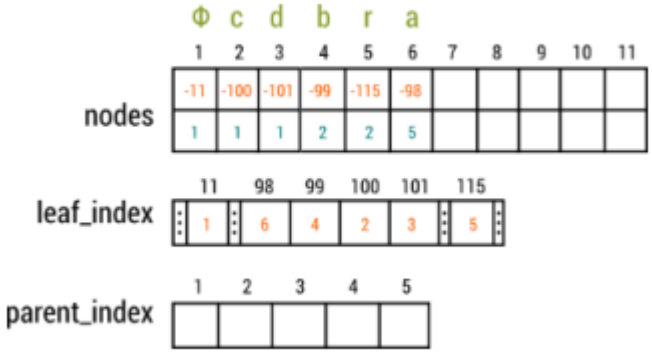


Fig.3 Полная установка узлов.

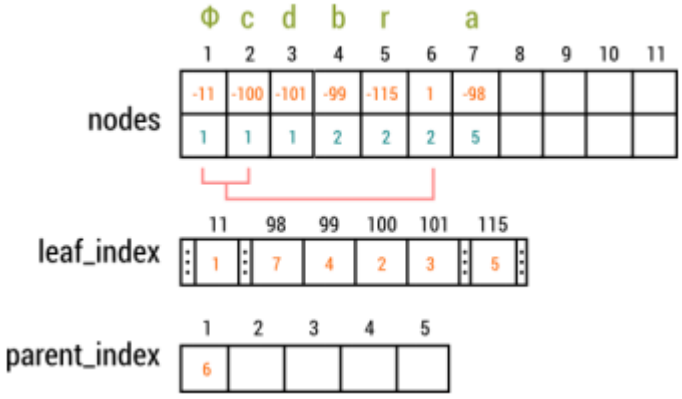

Fig.4 Комбинация $\phi$ и $c$.

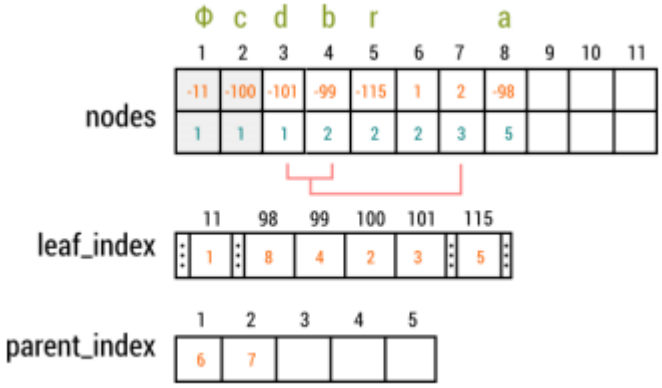


Fig.5 Комбинация d и b.

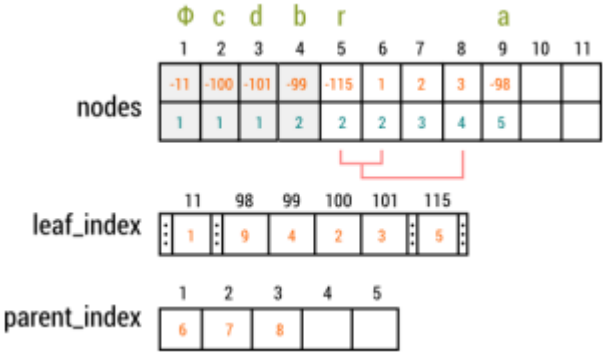


Fig.6 Комбинация r и 1.

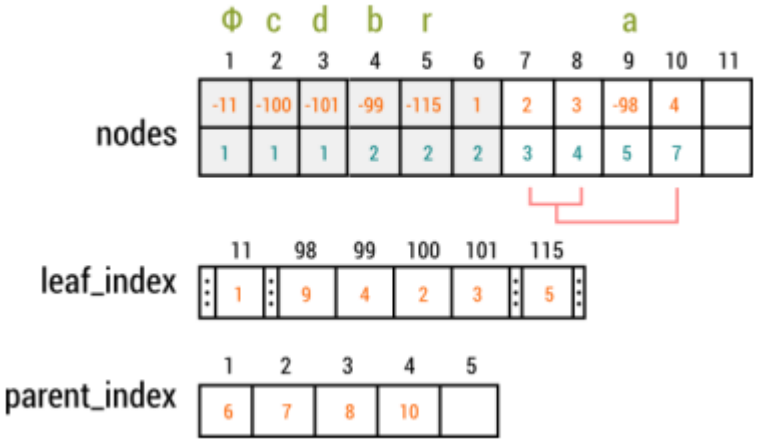


Fig.7 Комбинация 2 и 3.

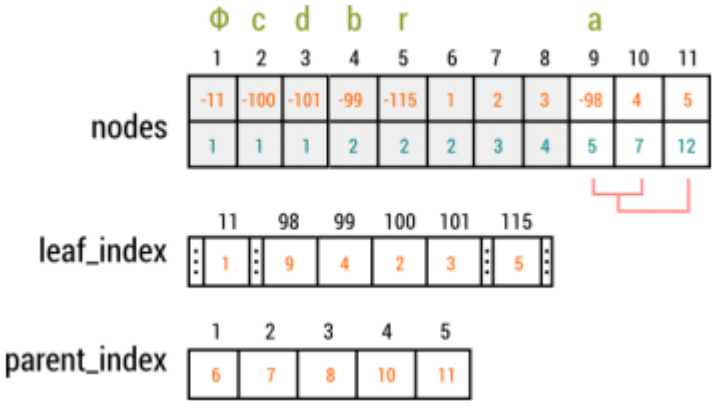


Fig.8 Комбинация a и 4.