Embedded Software Design Techniques

C programming 4: programming complex applications

Tsuyoshi Isshiki
Dept. Communications and Integrated Systems
Tokyo Institute of Technology
TAIST ICTES

Conduct by Asst. Prof. Dr. Kanjanapan Sukvichai

Lecture Outline

- Embedded software overview
  - What are “embedded systems” and “embedded software”?
- C programming 1: C language overview
  - Function, declaration, statement, expression
  - Data types, data structure, pointers and pointer dereferences
- C programming 2: algorithm complexity, program execution model
  - Bubble sort vs quick sort
  - Stack memory and program execution
- C programming 3: programming techniques in image processing
  - Dynamic memory allocation, image array implementation
  - Greyscaling, filtering, binarization, color quantization, dithering
- C programming 4: programming complex applications
  - Program development steps (ex. Huffman coding)
  - Binary tree construction, tree traversal
  - Bitstream handling
- Real time operating systems and application development
  - RTOS services, kernels
  - Context switching, task scheduling
  - Multi-task programming model

How to Write Complex Programs?

- Example: File compression using Huffman coding
- Step 1: Understand WHAT the problem is
  - How does Huffman coding compress information?
- Step 2: Determine (roughly) what COMPONENTS are required
- Step 3: Determine (roughly) what DATA STRUCTURES are required implement
- Step 4: Start writing codes
  - Don't write large codes at once. Code writing should always be accompanied with frequent debugging.
  - Don't try to implement the full functionality at first. Start with a simple case. But also specify in the code what kind of "simple cases" are assumed in the current code.
  - In this phase, you will find lots of things in Step 2 and Step 3 that are missing
  - In the intial code phases, ALWAYS assume that there will be bugs in the codes

In computer science and information theory, a Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding or using such a code proceeds by means of Huffman coding, an algorithm developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes" The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (weight) for each possible value of the source symbol.

Character	ASCII Value	ASCII Binary	Huffman Binary
`' '`	32	00100000	10
`'a'`	97	01100001	0001
`'b'`	98	01100010	0111010
`'c'`	99	01100011	001100
`'e'`	101	01100101	1100
`'z'`	122	01111010	00100011010

The steps you'll take to do perform a Huffman encoding of a given text source file into a destination compressed file are:

1. **count frequencies:** Examine a source file's contents and count the number of occurrences of each character, and store them in a map.
2. **build encoding tree:** Build a binary tree with a particular structure, where each node represents a character and its count of occurrences in the file. A priority queue (like the one you implemented in HW5!) is used to help build the tree along the way.
3. **build encoding map:** Traverse the binary tree to discover the binary encodings of each character.
4. **encode data:** Re-examine the source file's contents, and for each character, output the encoded binary version of that character to the destination file.

Normal Coding with fixed length binary

Character	Code	Frequency	Total Bits
A	000 Length = 3	10	30 Frequency x Bit Length
E	001	15	45
I	010	12	36
S	011	3	12
T	100	4	12
P	101	13	39
Newline	110	1	3

Total Bits Used: 174

Try to reduce number of total bits used (compress) by assigning variable-length code

more frequency → less number of code bits used

less frequency → more number of code bits used

Character	Code	Frequency
A	000 Length = 3	10
E	001	15
I	010	12
S	011	3
T	100	4
P	101	13
Newline	110	1

174 bits used in total

Huffman Coding

Char	Code	Freq	Total Bits
A	110	10	30
E	10	15	30
I	00	12	24
S	11111	3	15
T	1110	4	16
P	01	13	26
\n	11110	1	5

146 bits used in total

Size reduced by 16.1%

Lossless so don't worry !!!!

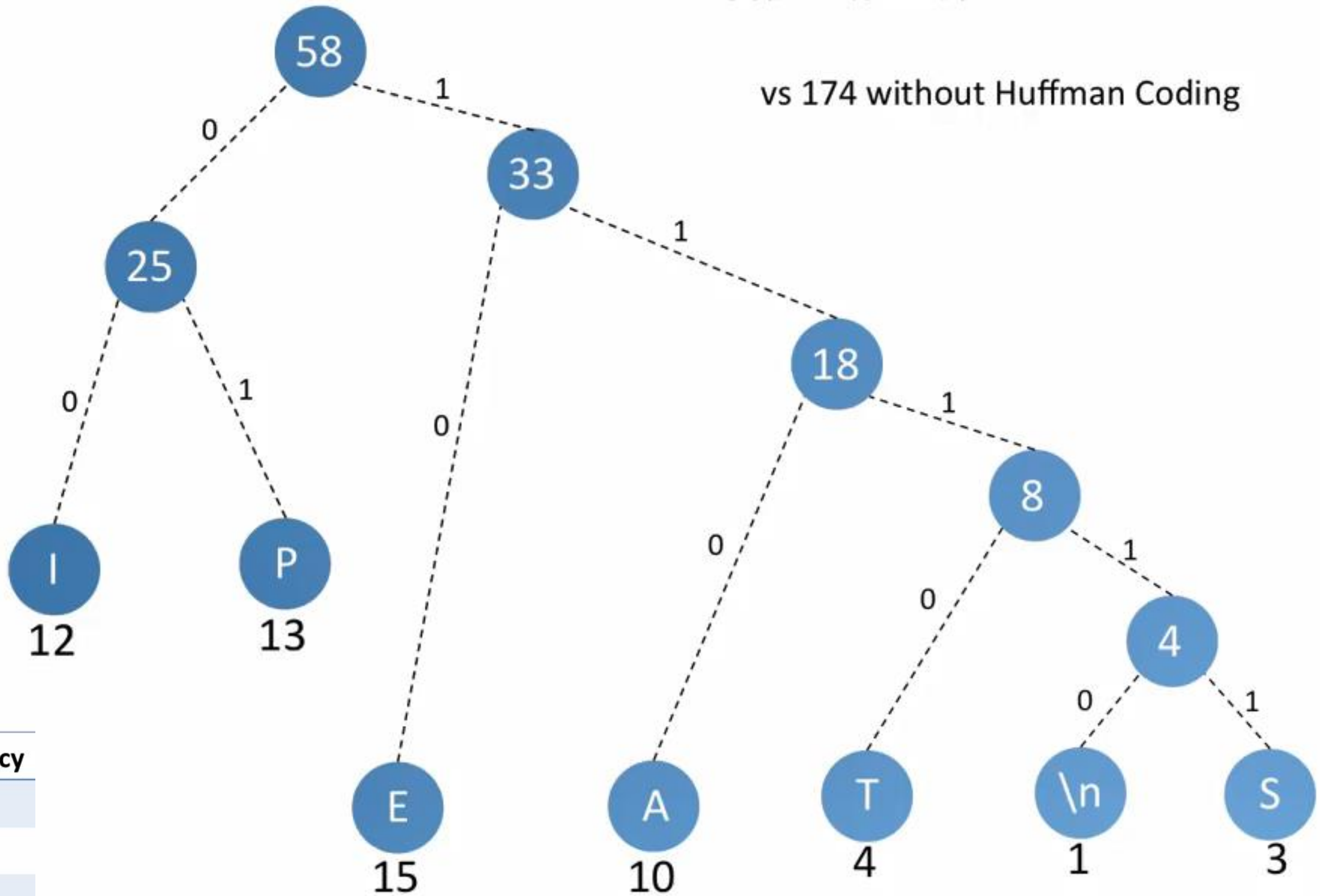
Create Code by Huffman Tree (Binary Tree)

Huffman Coding

Char	Code	Freq	Total Bits
A	110	10	30
E	10	15	30
I	00	12	24
S	11111	3	15
T	1110	4	16
P	01	13	26
\n	11110	1	5

Total Bits: 146

vs 174 without Huffman Coding



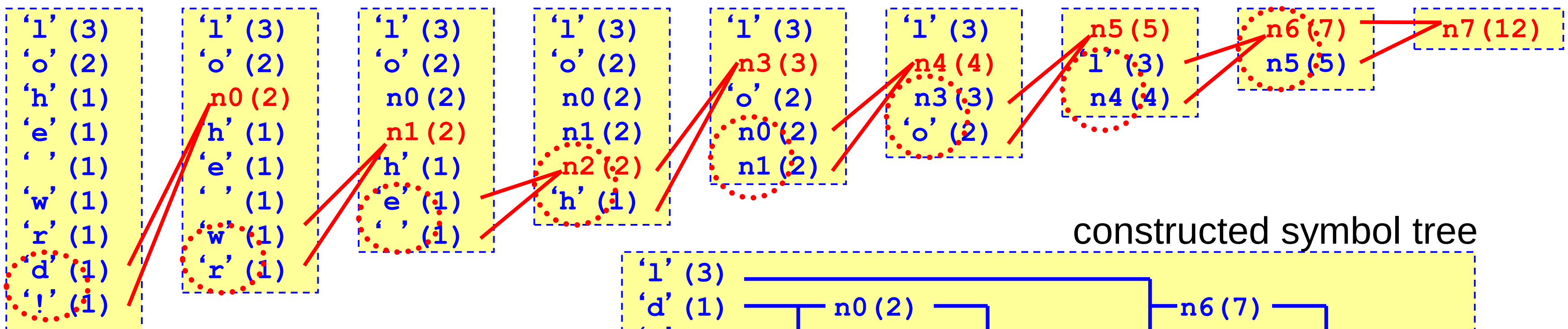
Character	Code	Frequency
A	000 Length = 3	10
E	001	15
I	010	12
S	011	3
T	100	4
P	101	13
Newline	110	1

How to Write Complex Programs?

- Example: File compression using Huffman coding
- Step 1: Understand WHAT the problem is
  - How does Huffman coding compress information?
  - → It assigns fewer bits to symbols that appear frequently in the data
  - → Need to count the occurrence of each symbol in the data (let's assume 1-byte character as a "symbol")
- Step 2: Determine (roughly) what COMPONENTS are required
  - count_symbols() : Reading a file, counting the occurrence of each symbol in the file
  - construct_huffman_tree() : Construct the binary tree where the leaves represent symbols and the tree structure represents how each symbol is encoded
  - write_compressed_file() : Write the compressed data to file
  - read_compressed_file() : Read the compressed file and restore the original information
- Step 3: Determine (roughly) what DATA STRUCTURES are required implement
  - struct symbol_info : symbol data, count, binary tree data (parent node, left child node, right child node)
  - struct file_info : various parameters and statistics of input file and symbols
- Step 4: Start writing codes
  - Don't write large codes at once. Code writing should always be accompanied with frequent debugging.
  - Don't try to implement the full functionality at first. Start with a simple case. But also specify in the code what kind of "simple cases" are assumed in the current code.
  - In this phase, you will find lots of things in Step 2 and Step 3 that are missing
  - In the intial code phases, ALWAYS assume that there will be bugs in the codes

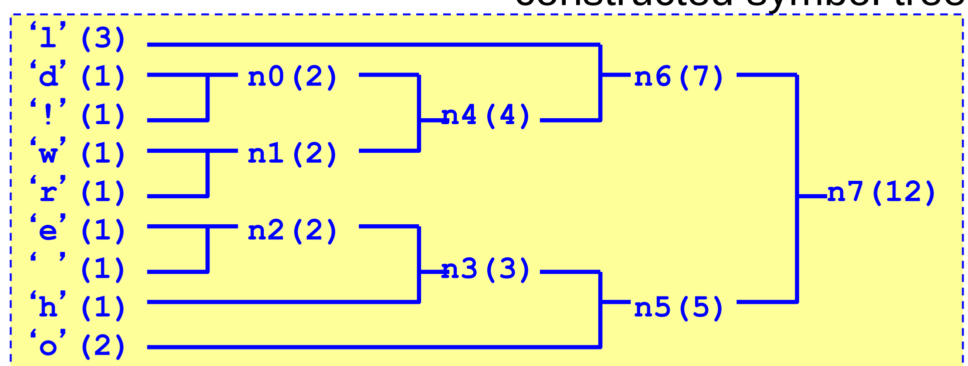
Step 1: Understand What the Problem is

- Basic concept of Huffman coding (sample text : “hello world!”)
  - Symbol statistics: count the occurrence of each symbol
    - ‘h’ = 1, ‘e’ = 1, ‘l’ = 3, ‘o’ = 2, <space> = 1, ‘w’ = 1, ‘r’ = 1, ‘d’ = 1, ‘!’ = 1
  - Huffman tree construction: build the symbol tree in the following way
    1. Initially, each symbol is a node without a parent or children
    2. For all nodes without parents, sort them in the order of symbol counts
    3. Choose the last two nodes, and create a parent node for these two nodes
    4. Set the parent’s symbol count as the sum of that of its two children
    5. Go to 2 until there are only one node without a parent (which is the root)



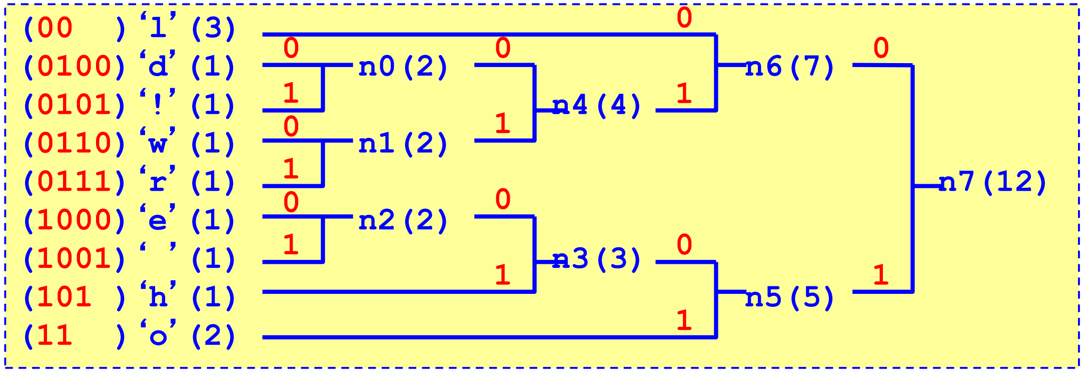
Since there are 9 different symbols in “hello world!”, we can code this text by assigning 4 bits per symbol which will result in a total of 48 bits = 6 bytes

constructed symbol tree



Step 1: Understand What the Problem is

- Symbol code assigment
  - At each node in the tree, assign ‘0’ to the left branch and ‘1’ to the right branch
  - Code for each symbol is the bit sequence generated by traversing the tree from the root to the symbol leaf



- Compression (encoding) : simply replace each symbol with its “code”
  - ‘h’ ‘e’ ‘l’ ‘l’ ‘o’ ‘ ’ ‘w’ ‘o’ ‘r’ ‘l’ ‘d’ ‘!’
  - → 101 1000 00 00 11 1001 0110 11 0111 00 0100 0101 **(= 37 bits total!!)**
- Decompression (decoding) : simply traverse the tree
  - 1011000000011100101101101110001000101

1. Scan the bit-stream 1 bit at a time (left to right)
2. Start from the root of the tree
3. If the current bit is ‘0’, then go to the left child, otherwise go to the right child
4. If the leaf is reached, output that symbol, and go to 2.

Step 2: Determine What Components are Required

- count_symbols() : count the occurrence of each symbol
  - ‘h’ = 1, ‘e’ = 1, ‘l’ = 3, ‘o’ = 2, <space> = 1, ‘w’ = 1, ‘r’ = 1, ‘d’ = 1, ‘!’ = 1
- create_huffman_tree() : build the symbol tree in the following way
  1. Initially, each symbol is a node without a parent or children
  2. For all nodes without parents, sort them in the order of symbol counts
     → sort_nodes() : maybe we can use quick_sort() for this ...
  3. Choose the last two nodes, and create a parent node for these two nodes
  4. Set the parent’s symbol count as the sum of that of its two children
  5. Go to 2 until there are only one node without a parent (which is the root)

Step 2: Determine What Components are Required

- write_compressed_file() : simply replace each symbol with its “code”
  - → assign_code_to_symbol() : symbol code assigment
    - At each node in the tree, assign ‘0’ to the left branch and ‘1’ to the right branch
    - Code for each symbol is the bit sequence generated by traversing the tree from the root to the symbol leaf
  - → *We need the symbol codes but don’t need the tree after assigning the codes*
- read_compressed_file() : simply traverse the tree
  - → *We need Huffman tree information to decode! (but don’t need the symbol code itself)*
  - → *Need to be able to write and read Huffman tree information to the compressed file*
  - → write_huffman_tree(), read_huffman_tree()

Step 3: Determine What Data Structures are Required

- What kind of data structure is needed for describing the Huffman tree?
  - Describing the tree structure:
    - left (child), right (child), parent : what data type? → a pointer to a “tree node”
  - Symbol information
    - Symbol value (char value: 1-byte)
    - Count (occurrence in the file) : 4 bytes
    - Code information : how many bits should be reserved? → Let’s assume that 32 bits are enough → Let’s worry later if this is not the case ... (but let’s remember that we assumed this)
    - Code length (since the code is “variable length”) : if we assume maximum of 32 bits per code, then the code length can be expressed with 5 bits only → 1 byte is enough

Step 3: Determine What Data Structures are Required

- Let's also have a data structure which maintains the Huffman tree and any other information
  - How many symbols? → 256 (this number can be fixed since we are assuming 8-bit symbols)
  - How many internal nodes → 255 (maximum) since this is a binary tree
  - → These fixed number of symbols and internal nodes can be simply stored in arrays
  - What other information is required?
    - Number of total symbols
    - Number of symbol types in the file

Step 4: Start Writing Codes

- But where do we start???
  - Start with main() function because:
    - We will be doing a lot of intermediate tests while writing the codes, and we need the main function to do this
    - Let's also consider what kind of parameters will be passed to the main function
  - Implement one or two simple functions that will be called right away
    - count_symbol() would be a good candidate
    - But maybe even before this, we may want to make sure we can open and read the file correctly → let's start from this part...

      → Even if you are an experienced coder (have written lots of programs with file IOs), it is always good to start the coding process this way → debugging your codes at the earliest possible time (because you can always make easy mistakes that can be very hard to detect)

Step 4.1: “read_file”

```
void read_file(FILE * fp)
{
        int ccount = 0;
        while(1){
            int c = fgetc(fp);
            if(c == EOF) break;
            printf(“%c”, c);
            ccount ++;
        }
        printf("ccount = %d\n", ccount);
}
int main(int argc, char * argv[])
{
        FILE * fp;
        if(argc != 2){
            printf("invalid command!\n");
            printf("usage: %s <filename>\n", argv[0]);
            return 0;
        }
        fp = fopen(argv[1], "rb");
        if(fp == 0){
            printf("cannot open file <%s>!!\n", argv[1]);
            return 0;
        }
        read_file(fp);
        fclose(fp)
        return 1;
}
```

`fgets(fp)` : read one character from file

→ Returns “int” value data between 0 and 255 for normal cases, but returns EOF (– 1) when the End-Of-File is reached

`printf(“%c”, c);` : simply outputs the character data read from the file back to the console → to check that we are correctly reading in the file

`argv[1]` : name of the file to compress

`fp = fopen(argv[1], "rb");` :

open the file with mode “rb”

- “r”: read mode
- “w”: write mode
- “rb”: read-binary mode
- “wb”: write-binary mode

Step 4.1: Start Writing Codes

- What we have accomplished so far:
  - Parsed the command argument argv[1] as file name
  - Opened the file to (eventually) compress
  - Read each character from the file and displayed on console → looking at the outputs, we can confirm that the program is correctly working (this part is quite important in any step)
- Next step:
  - count_symbol(): modify "read_file()" function
  - → But we may want to start considering the data structure at this point…
  - Data structures
    - typedef struct sym_info SINFO; (symbols, tree nodes)
    - typedef struct huffman_info HINFO; (global information)

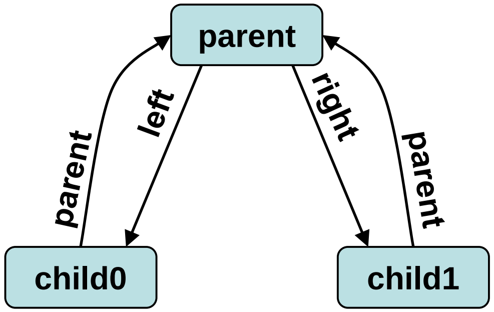
Step 4.2: Start Writing Data Structures

```
typedef struct sym_info
{
     int count;
     struct sym_info * left, * right, * parent;
     unsigned char index, code_length;
     unsigned int code;
} SINFO;
```

data structure for symbols and internal tree nodes

pointer to tree nodes

symbol code assumes 32 bits




```
typedef struct huffman_info
{
     SINFO sinfo[256], inode[256];
     int text_count, symbol_count,
               max_code_length;
     FILE * fp_txt;
} HINFO;
```

array of 256 symbols
array of 256 internal nodes
(we only need 255 internal nodes, but this will make things easier)

Step 4.2: Start Writing Data Structures

```
typedef struct sym_info
{
        int count;
        struct sym_info * left, * right, * parent;
        unsigned char index, code_length;
        unsigned int code;
} SINFO;
typedef struct huffman_info
{
        SINFO sinfo[256], inode[256];
        int text_count, symbol_count, max_code_length;
        FILE * fp_txt;
} HINFO;
void SINFO_initialize(SINFO * sinfo, int i)
{
        sinfo->index = i;
        sinfo->count = 0; sinfo->code = 0; sinfo->code_length = 0;
        sinfo->left = 0; sinfo->right = 0; sinfo->parent = 0;
}
void HINFO_initialize(HINFO * hinfo)
{
        int i;
        for(i = 0; i < 256; i ++){
            SINFO_initialize(&hinfo->sinfo[i], i);
            SINFO_initialize(&hinfo->inode[i], i);
        }
}
```

"index" holds the char value

Initialize index value for each SINFO data
(for symbol leaf nodes and internal nodes)

Step 4.3: “count_symbols”

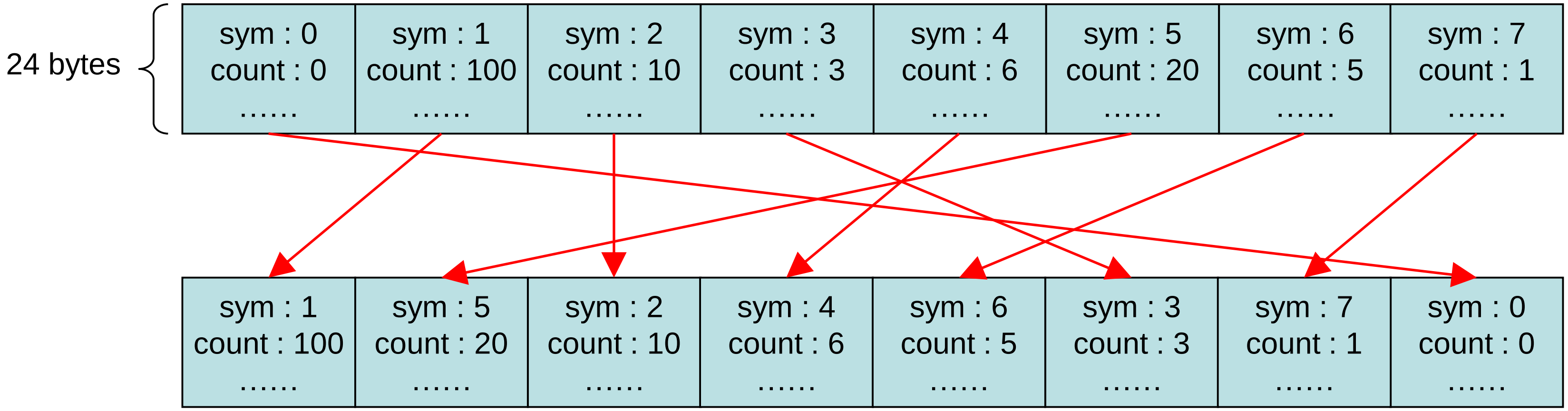
```
void HINFO_count_symbols(HINFO * hinfo)
{
        hinfo->text_count = 0;
        while(1){
            int c = fgetc(hinfo->fp_txt);
            if(c == EOF) break;
            hinfo->sinfo[c].count ++;
            hinfo->text_count ++;
        }
        printf(“text_count = %d\n", hinfo->text_count);
}
int main(int argc, char * argv[])
{
        HINFO hinfo;
        HINFO_initialize(&hinfo);
        if(argc != 2){
            printf("invalid command!\n");
            printf("usage: %s <filename>\n", argv[0]);
            return 0;
        }
        hinfo.fp_txt = fopen(argv[1], "rb");
        if(hinfo.fp_txt == 0){
            printf("cannot open file <%s>!!\n", argv[1]);
            return 0;
        }
        HINFO_count_symbols(&hinfo);
        fclose(hinfo.fp_txt)
        return 1;
}
```

Modified from **read_file()**

Counting the symbol occurrence

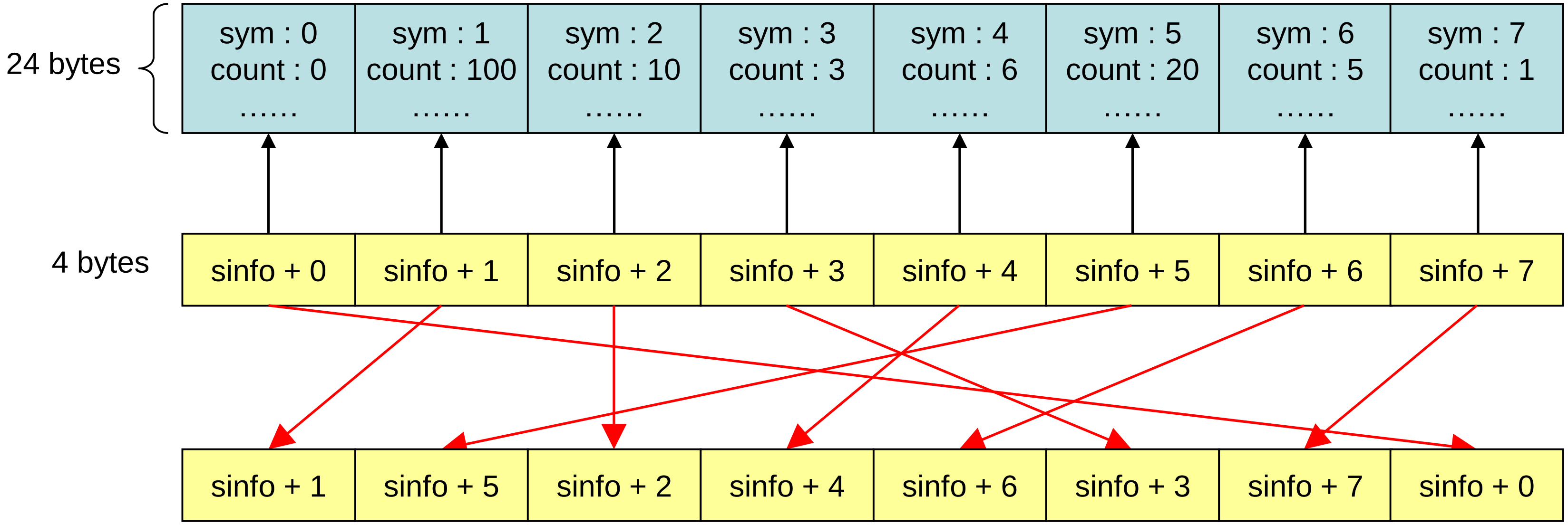
Step 4.4: “sort_nodes”

- Next big task is to create the Huffman tree → but we need to do a lot here, so let’s break up the task into little ones
  - Let’s first implement sort_nodes()
- Important design issue here is how to maintain the sort order information
  - Reorder the array sinfo[256] according to “count” → this can be very expensive since there are a lot of data in SINFO (22 bytes total, which will be aligned to 24 bytes) to move from one location to another (swapping)



Step 4.4: “sort_nodes”

- Important design issue here is how to maintain the sort order information
  - Better way is to prepare an array of *pointers* to SINFO and just swap the pointer values so that the order of the array of pointers to SINFO reflect the sorting order on “count”
  - If we are going to write a sorting function on an array of pointers, let’s make it so that it can sort on different data keys and even other data types → then we can reuse this code for other purposes



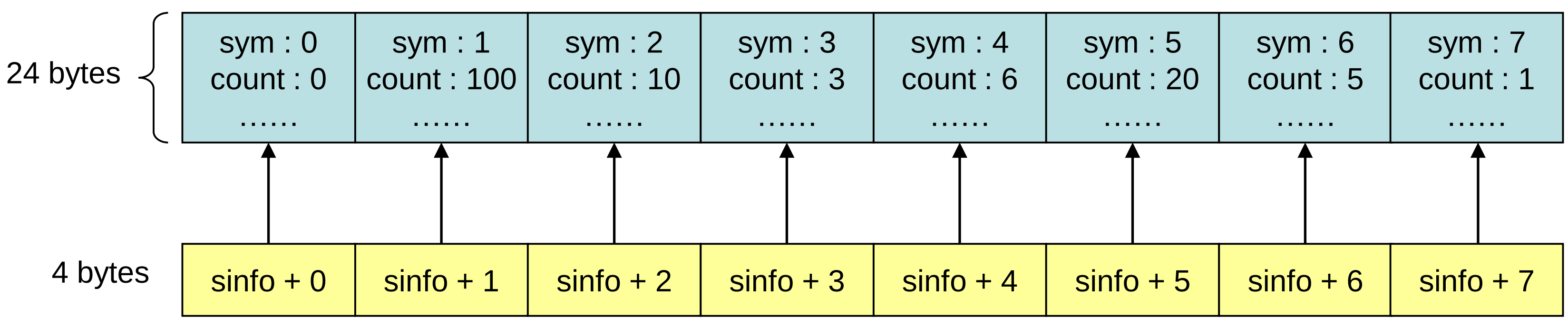
Array of Pointers to SINFO

```
typedef struct huffman_info
{
        SINFO sinfo[256], inode[256];
        SINFO * p_sinfo[256], * root_node;
        int text_count, symbol_count, max_code_length;
        FILE * fp_txt;
} HINFO;

void HINFO_initialize_psinfo(HINFO * hinfo)
{
        int i;
        for(i = 0; i < 256; i ++)
            hinfo->p_sinfo[i] = &hinfo->sinfo[i];
}
```

`SINFO * p_info[256]` : array of 256 pointers to **`SINFO`** elements
`SINFO * root_node` : pointer to root node

`p_info[256]` : initially in the same order as **`sinfo[256]`** (which is ordered by symbol char values)



Recall “quick_sort”

```
void swap(int * a0, int * a1)
{
        int t = *a0;
        *a0 = *a1;
        *a1 = t;
}
int partition(int a[], int left, int right)
{
        int pivot = a[right];
        int i = left, k = right;
        while(i < k){
            while(i < k && a[i] < pivot) i ++;
            while(i < k && a[k] >= pivot) k --;
            if(i < k) swap(&a[i], &a[k]);
        }
        if(right > k) swap(&a[right], &a[k]);
        return k;
}
void quick_sort(int a[], int left, int right)
{
        if(left < right){
            int pivot_pos = partition(a, left, right);
            quick_sort(a, left, pivot_pos - 1);
            quick_sort(a, pivot_pos + 1, right);
        }
}
```

Swapping “int” data pairs

Previous “quick_sort” worked on an array of “int” data
→We want to change this to an array of pointers

We want to describe this comparison operation in a way that allows us to use this function on a variety of data types → we can do this with “*function pointers*”

"quick_sort" on Array of Pointers

```
void swap_p(void ** a0, void ** a1)
{
        void * t = *a0;
        *a0 = *a1;
        *a1 = t;
}
int partition_p(void * a[], int left, int right, int (* is_ordered)(void *, void *))
{
        void * p_pivot = a[right];
        int i = left, k = right;
        while(i < k){
            while(i < k && is_ordered(a[i], p_pivot)) i ++;
            while(i < k && !is_ordered(a[k], p_pivot)) k --;
            if(i < k) swap_p(&a[i], &a[k]);
        }
        if(right > k) swap_p(&a[right], &a[k]);
        return k;
}
void quick_sort_p(void * a[], int left, int right, int (* is_ordered)(void *, void *))
{
        if(left < right){
            int pivot_pos = partition_p(a, left, right, is_ordered);
            quick_sort_p(a, left, pivot_pos - 1, is_ordered);
            quick_sort_p(a, pivot_pos + 1, right, is_ordered);
        }
}
```

Swapping "void *" data pairs (pointer to an unknown type)

is_ordered : pointer to a function that checks the order of two elements (defined elsewhere in the code)

void * p_pivot : pointer to the pivot element

Checking the order between each array element and the pivot element using the function **is_ordered**

Sorting the Array of Pointers to SINFO

```
int SINFO_is_ordered(void * a, void * b)
{
        return ((SINFO *) a)->count > ((SINFO *) b)->count;
}

void HINFO_sort_symbols(HINFO * hinfo)
{
        int i;
        HINFO_initialize_psinfo(hinfo);
        quick_sort_p((void **) hinfo->psinfo, 0, 255, SINFO_is_ordered);
        for(i = 255; i >= 0; i --){
            if(hinfo->p_sinfo[i]->count)
                        break;
        }
        hinfo->symbol_count = i + 1;
}
```

`SINFO_is_ordered()` : actual function called in **`quick_sort_p()`**

`quick_sort_p` called here (see how we pass the function pointer **`SINFO_is_ordered`** in the call argument)

`hinfo->symbol_count` is set here which is the number of symbols appearing in the target file (symbols that have **`count`** > 0)

Step 4.4: “sort_nodes”

```
void HINFO_print_sorted_symbols(HINFO * hinfo)
{
      int i;
      HINFO_sort_symbols(hinfo);
      for(i = 0; i < hinfo->symbol_count; i ++){
         SINFO * sinfo = hinfo->p_sinfo[i];
         printf("%3d<%c> : %5d\n", sinfo->index, sinfo->index, sinfo->count);
      }
}
```

This symbol (value = 32, <space>) appears most frequently in this file

```
int main(int argc, char * argv[])
{
      HINFO hinfo;
      ......... /* initialize hinfo, open file */
      HINFO_count_symbols(&hinfo);
      HINFO_print_sorted_symbols(&hinfo);
      fclose(hinfo.fp_txt)
      return 1;
}
```

This symbol (value = 10) is **‘\n’** (new line), so printing this directly on console will add new line

```
> a.exe cppinternals.info
text_count = 50235
 32< > :  8754
101<e> :  5035
116<t> :  3419
110<n> :  3139
105<i> :  2859
 97<a> :  2788
111<o> :  2670
115<s> :  2394
114<r> :  2218
104<h> :  1537
108<l> :  1443
 99<c> :  1425
100<d> :  1277
112<p> :  1089
 10<
> :  1035
102<f> :   850
109<m> :   792
......
```

Displaying Escape Characters

```
#define SET_STRING(c0, c1) __str[0] = c0; __str[1] = c1;
char * convert_char(int c)
{
       static char __str[3] = {0, 0, 0};
       switch(c){
          case '\a': SET_STRING('\\', 'a') break;
          case '\b': SET_STRING('\\', 'b') break;
          case '\f': SET_STRING('\\', 'f') break;
          case '\n': SET_STRING('\\', 'n') break;
          case '\r': SET_STRING('\\', 'r') break;
          case '\t': SET_STRING('\\', 't') break;
          case '\v': SET_STRING('\\', 'v') break;
          default: SET_STRING(c, 0); break;
       }
       return __str;
}
void HINFO_print_sorted_symbols(HINFO * hinfo)
{
       int i;
       HINFO_sort_symbols(hinfo);
       for(i = 0; i < hinfo->symbol_count; i ++){
          SINFO * sinfo = hinfo->p_sinfo[i];
          printf("%3d<%2s> : %5d\n",
                   sinfo->index, convert_char(sinfo->index), sinfo->count);
       }
}
```

`SET_STRING('\\', 'a')`

macro expansion

`__str[0] = '\\'; __str[1] = 'a';`

`__str is printed as \a`

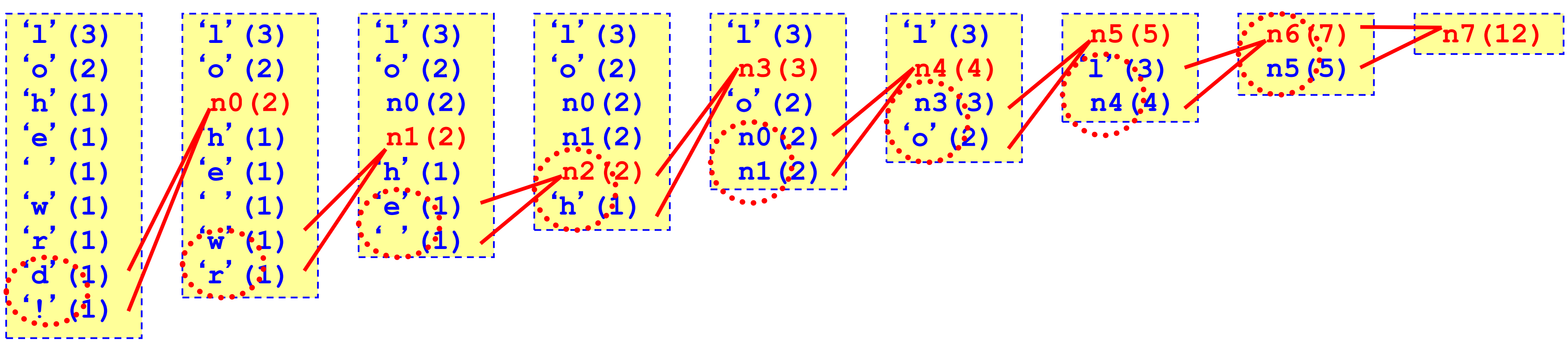
Displaying Escape Characters

```
#define SET_STRING(c0, c1) __str[0] = c0; __str[1] = c1;
char * convert_char(int c)
{
        static char __str[3] = {0, 0, 0};
        switch(c){
            case '\a': SET_STRING('\\', 'a') break;
            case '\b': SET_STRING('\\', 'b') break;
            case '\f': SET_STRING('\\', 'f') break;
            case '\n': SET_STRING('\\', 'n') break;
            case '\r': SET_STRING('\\', 'r') break;
            case '\t': SET_STRING('\\', 't') break;
            case '\v': SET_STRING('\\', 'v') break;
            default: SET_STRING(c, 0); break;
        }
        return __str;
}
void HINFO_print_sorted_symbols(HINFO * hinfo)
{
        int i;
        HINFO_sort_symbols(hinfo);
        for(i = 0; i < hinfo->symbol_count; i ++){
            SINFO * sinfo = hinfo->p_sinfo[i];
            printf("%3d<%2s> : %5d\n",
                        sinfo->index, convert_char(sinfo->index), sinfo->count);
        }
}
```

```
> a.exe cppinternals.info
text_count = 50235
 32<  > :  8754
101< e> :  5035
116< t> :  3419
110< n> :  3139
105< i> :  2859
 97< a> :  2788
111< o> :  2670
115< s> :  2394
114< r> :  2218
104< h> :  1537
108< l> :  1443
 99< c> :  1425
100< d> :  1277
112< p> :  1089
 10<\n> :  1035
102< f> :   850
109< m> :   792
......
```

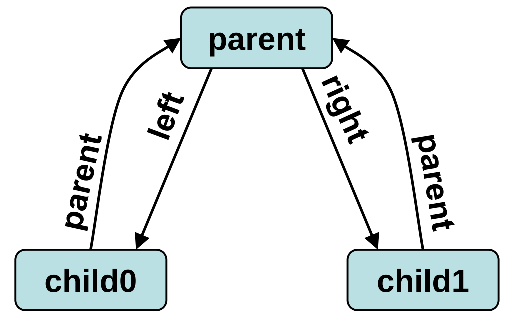
Step 4.5: "create_huffman_tree"

- Now we are ready to implement "create_huffman_tree"
  - Huffman tree construction: build the symbol tree in the following way
    1. Initially, each symbol is a node without a parent or children
    2. For all nodes without parents, sort them in the order of symbol counts
    3. Choose the last two nodes, and create a parent node for these two nodes
    4. Set the parent's symbol count as the sum of that of its two children
    5. Go to 2 until there are only one node without a parent (which is the root)



"create_huffman_tree"

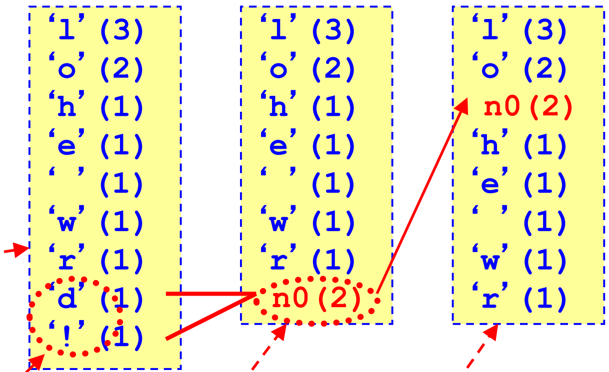
```
void SINFO_create_sinfo_node(SINFO * child0, SINFO * child1, SINFO * parent)
{
     child0->parent = parent;
     child1->parent = parent;
     parent->left = child0;
     parent->right = child1;
     parent->count = child0->count + child1->count;
}
void HINFO_create_huffman_tree(HINFO * hinfo)
{
     int i, tail, cindex = 0;
     HINFO_sort_symbols(hinfo);
     tail = hinfo->symbol_count - 1;
     while(tail > 0){
        SINFO * child0, * child1, * parent;
        child1 = hinfo->p_sinfo[tail];
        child0 = hinfo->p_sinfo[tail - 1];
        parent = &hinfo->inode[cindex ++];
        SINFO_create_sinfo_node(child0, child1, parent);
        hinfo->p_sinfo[tail - 1] = parent;
        quick_sort((void **) hinfo->p_sinfo, 0, tail - 1, SINFO_is_ordered);
        tail --;
     }
     hinfo->root_node = hinfo->p_sinfo[0];
     SINFO_assign_code(hinfo->root_node);
}
```

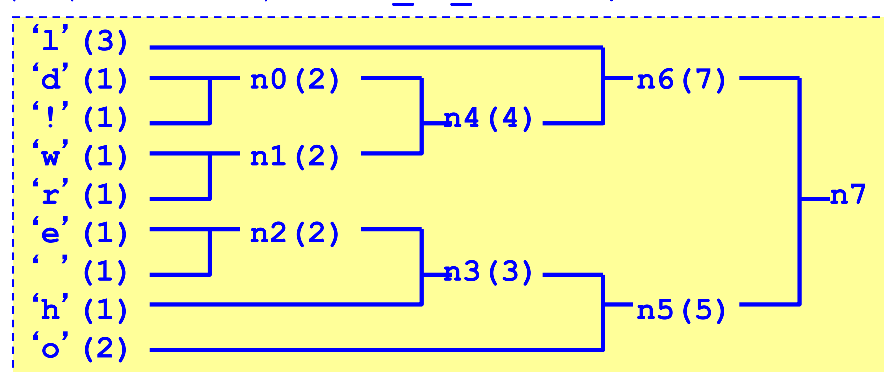


`hinfo->inode[256]` array is used for each `parent` node

“create_huffman_tree”

```
void HINFO_create_huffman_tree(HINFO * hinfo)
{
        int i, tail, cindex = 0;
        HINFO_sort_symbols(hinfo);
        tail = hinfo->symbol_count - 1;
        while(tail > 0){
            SINFO * child0, * child1, * parent;
            child1 = hinfo->p_sinfo[tail];
            child0 = hinfo->p_sinfo[tail - 1];
            parent = &hinfo->inode[cindex ++];
            SINFO_create_sinfo_node(child0, child1, parent);
            hinfo->p_sinfo[tail - 1] = parent;
            quick_sort((void **) hinfo->p_sinfo, 0, tail - 1, SINFO_is_ordered);
            tail --;
        }
        hinfo->root_node = hinfo->p_sinfo[0];
        SINFO_assign_code(hinfo->root_node);
}
```





	n0	n1	n2	n3	n4	n5	n6	n7
parent	n4	n4	n3	n5	n6	n6	n7	***
hinfo->inode[256]	n0	n1	n2	n3	n4	n5	n6	n7
left, right	‘d’ ‘!’	‘w’ ‘r’	‘e’ ‘ ’	n2 ‘h’	n0 n1	n3 ‘o’	‘l’ n4	n6 n7

“create_huffman_tree”

```
void HINFO_create_huffman_tree(HINFO * hinfo)
{
        int i, tail, cindex = 0;
        HINFO_sort_symbols(hinfo);
        tail = hinfo->symbol_count - 1;
        while(tail > 0){
            SINFO * child0, * child1, * parent;
            child1 = hinfo->p_sinfo[tail];
            child0 = hinfo->p_sinfo[tail - 1];
            parent = &hinfo->inode[cindex ++];
            SINFO_create_sinfo_node(child0, child1, parent);
            hinfo->p_sinfo[tail - 1] = parent;
            quick_sort((void **) hinfo->p_sinfo, 0, tail - 1, SINFO_is_ordered);
            tail --;
        }
        hinfo->root_node = hinfo->p_sinfo[0];
        SINFO_assign_code(hinfo->root_node);
}
```

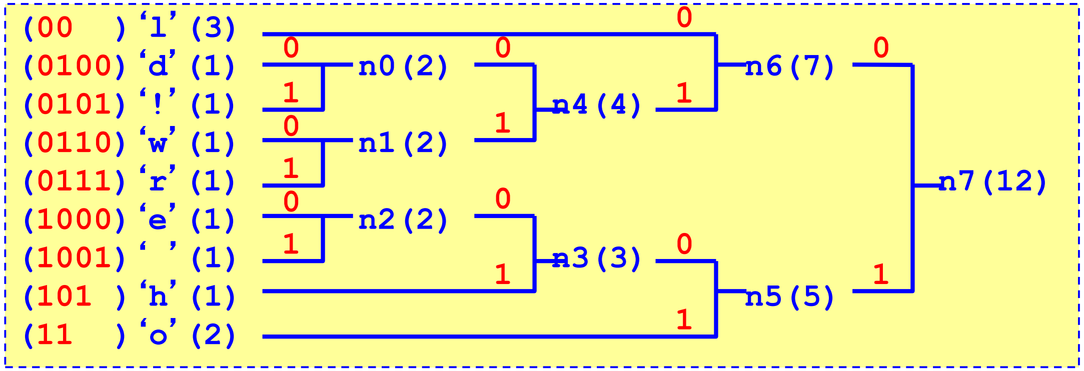
Finally, `hinfo->p_sinfo[0]` will be the root node of the tree

`SINFO_assign_code()` will be implemented next

Step 4.5: "create_huffman_tree"

- Next: symbol code assigment
  - At each node in the tree, assign '0' to the left branch and '1' to the right branch
  - Code for each symbol is the bit sequence generated by traversing the tree from the root to the symbol leaf
  - → It will also be helpful if we assign the codes to intermediate nodes (root node n7 does not have any code bits assigned)
    - n6: "0"
    - 'l' : "00" (left child of n6:"0")
    - n4: "01" (right child of n6:"0")
    - n1: "011" (right child of n4:"01")
    - n0: "010" (left child of n4:"01")
    - 'd' : "0100" (left child of n0:"010")
    - '!' : "0101" (right child of n0:"010")

Symbol code at each node can be derived RECURSIVELY by taking the parent's code and concatenating '0' or '1'



Step 4.5: "create_huffman_tree"

```
#define SIB_DIR(si, child) (si->parent && si->parent->child == si)
void SINFO_assign_code(SINFO * sinfo)
{
        if(sinfo->parent){
            sinfo->code_length = sinfo->parent->code_length + 1;
            sinfo->code = (SIB_DIR(sinfo, right) << sinfo->parent->code_length)
                          + (sinfo->parent->code);
        }
        if(sinfo->left) SINFO_assign_code(sinfo->left);
        if(sinfo->right) SINFO_assign_code(sinfo->right);
}
```

- `sinfo->code_length` can be calculated simply by adding 1 to `sinfo->parent->code_length`
- `SIB_DIR(sinfo, right)` is a macro call that will be replaced by :

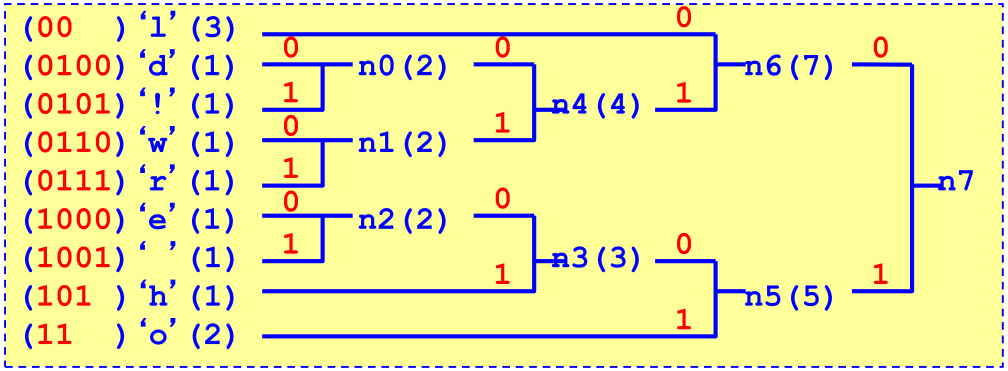
  `(sinfo->parent && sinfo->parent->right == sinfo)`

  → if `sinfo` is the right child of `sinfo->parent`, then it evaluates to `1` (otherwise `0`)
- `sinfo->code` is calculated by shifting `SIB_DIR(sinfo, right)` to left by `sinfo->parent->code_length` bits and adding it to `sinfo->parent->code`
  → this will cause the code to be stored in LSB-first order ("0101" will be stored as 10 (1010) in binary)
- `SINFO_assign_code` is recursively called for each child (if it exists)
  → It is initially called from the root node : `SINFO_assign_code(hinfo->root_node);`
  → Root node (as well as other nodes) is initialized in `SINFO_initialize()` as :
  `hinfo->root_node->code = 0;`
  `hinfo->root_node->code_length = 0;`

Step 4.5: “create_huffman_tree”

```
#define SIB_DIR(si, child) (si->parent && si->parent->child == si)
void SINFO_assign_code(SINFO * sinfo)
{
        if(sinfo->parent){
            sinfo->code_length = sinfo->parent->code_length + 1;
            sinfo->code = (SIB_DIR(sinfo, right) << sinfo->parent->code_length)
                          + (sinfo->parent->code);
        }
        if(sinfo->left) SINFO_assign_code(sinfo->left);
        if(sinfo->right) SINFO_assign_code(sinfo->right);
}
```

```
n7 (root node):
n7->code = 0
n7->code_length = 0
```




```
n6 = n7->left :
n6->code_length = 0 + 1 = 1
SIB_DIR(n6, right) = 0
n6->code = (0 << 0) + 0 = 0 (0)
```

```
n4 = n6->right :
n4->code_length = 1 + 1 = 2
SIB_DIR(n4, right) = 1
n4->code = (1 << 1) + 0 = 2 (10)
```

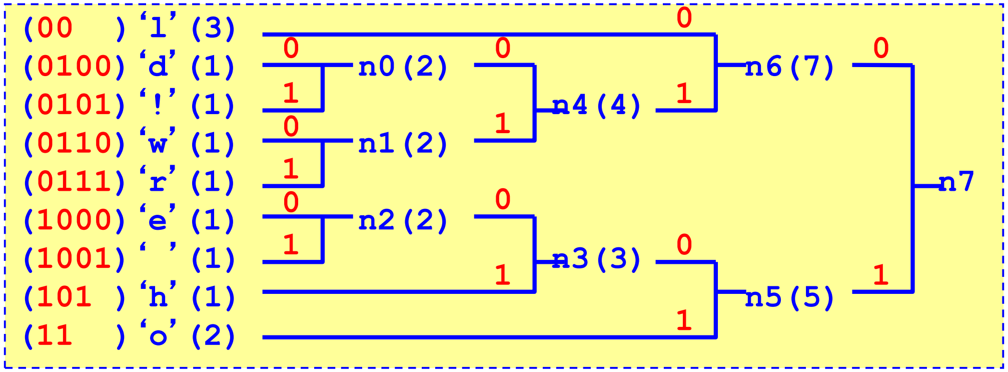
```
n0 = n4->left :
n0->code_length = 2 + 1 = 3
SIB_DIR(n0, right) = 0
n0->code = (0 << 2) + 2 = 2 (010)
```

```
n1 = n4->right :
n1->code_length = 2 + 1 = 3
SIB_DIR(n1, right) = 1
n1->code = (1 << 2) + 2 = 6 (110)
```

Step 4.5: “create_huffman_tree”

```
#define SIB_DIR(si, child) (si->parent && si->parent->child == si)
void SINFO_assign_code(SINFO * sinfo)
{
        if(sinfo->parent){
            sinfo->code_length = sinfo->parent->code_length + 1;
            sinfo->code = (SIB_DIR(sinfo, right) << sinfo->parent->code_length)
                          + (sinfo->parent->code);
        }
        if(sinfo->left) SINFO_assign_code(sinfo->left);
        if(sinfo->right) SINFO_assign_code(sinfo->right);
}
```

```
n7 (root node  ):code=(0000),code_length= 0
n6 (n7->left  :0):code=(0000),code_length= 1
'l'(n6->left  :0):code=(0000),code_length= 2
n4 (n6->right:1):code=(0010),code_length= 2
n0 (n4->left  :0):code=(0010),code_length= 3
'd'(n0->left  :0):code=(0010),code_length= 4
'!'(n0->right:1):code=(1010),code_length= 4
n1 (n4->right:1):code=(0110),code_legnth= 3
'w'(n1->left  :0):code=(0110),code_length= 4
'r'(n1->right:1):code=(1110),code_length= 4
n5 (n7->right:1):code=(0001),code_length= 1
n3 (n5->left  :0):code=(0001),code_length= 2
n2 (n3->left  :0):code=(0001),code_length= 3
'e'(n2->left  :0):code=(0001),code_length= 4
' '(n2->right:1):code=(1001),code_length= 4
'h'(n3->right:1):code=(0101),code_length= 3
'o'(n5->right:1):code=(0011),code_length= 2
```



Step 4.5: “create_huffman_tree”

```
void SINFO_print_huffman_code(SINFO * sinfo)
{
        int i, mask;
        printf("(%2d bits: ", sinfo->code_length);
        for(i = 0, mask = 1; i < sinfo->code_length; i ++, mask <<= 1)
           printf("%c", ((sinfo->code) & mask) ? '1' : '0');
        printf(" )");
}
```

```
'!'(n0->right):code = (1010), code_length = 4

mask = (0001): (1010) & (0001) == 0 → OUTPUT: 0
mask = (0010): (1010) & (0010) == 1 → OUTPUT: 01
mask = (0100): (1010) & (0100) == 0 → OUTPUT: 010
mask = (1000): (1010) & (1000) == 1 → OUTPUT: 0101
```

```
'r'(n1->right):code = (1110), code_length = 4

mask = (0001): (1110) & (0001) == 0 → OUTPUT: 0
mask = (0010): (1110) & (0010) == 1 → OUTPUT: 01
mask = (0100): (1110) & (0100) == 0 → OUTPUT: 010
mask = (1000): (1110) & (1000) == 1 → OUTPUT: 0101
```

Step 4.5: “create_huffman_tree”

```
void HINFO_print_sorted_symbols(HINFO * hinfo)
{
        int i, total_bits = 0, total_bytes;
        HINFO_sort_symbols(hinfo);
        for(i = 0; i < hinfo->symbol_count; i ++){
            SINFO * sinfo = hinfo->p_sinfo[i];
            printf("%3d<%2s> : %5d ",
                        sinfo->index, convert_char(sinfo->index), sinfo->count);
            SINFO_print_huffman_code(sinfo);
            printf("\n");
            total_bits += sinfo->code_length * sinfo->count;
        }
        total_bytes = (total_bits + 7) / 8;
        printf("total bits = %d\ntotal bytes = %d\n",
            total_bits, total_bytes);
        printf("compression rate = %f\n",
            (double) total_bytes / (double) hinfo->text_count);
}

int main(int argc, char * argv[])
{
        HINFO hinfo;
        ......... /* initialize hinfo, open file */
        HINFO_count_symbols(&hinfo);
        HINFO_create_huffman_tree(&hinfo);
        HINFO_print_sorted_symbols(&hinfo);
        fclose(hinfo.fp_txt)
        return 1;
}
```

We can precisely calculate the size of the compressed file since we know the code length and the occurrence of each symbol

```
> a.exe cppinternals.info
text_count = 50235
 32<  > :  8754 ( 3 bits: 000 )
101< e> :  5035 ( 3 bits: 110 )
116< t> :  3419 ( 4 bits: 0100 )
110< n> :  3139 ( 4 bits: 0101 )
105< i> :  2859 ( 4 bits: 1000 )
 97< a> :  2788 ( 4 bits: 1001 )
111< o> :  2670 ( 4 bits: 1011 )
115< s> :  2394 ( 4 bits: 1111 )
114< r> :  2218 ( 5 bits: 00100 )
104< h> :  1537 ( 5 bits: 01101 )
108< l> :  1443 ( 5 bits: 01111 )
 99< c> :  1425 ( 5 bits: 10100 )
100< d> :  1277 ( 5 bits: 11100 )
112< p> :  1089 ( 6 bits: 001010 )
 10<\n> :  1035 ( 6 bits: 001100 )
102< f> :   850 ( 6 bits: 001110 )
109< m> :   792 ( 6 bits: 011000 )
......
total bits = 229893
total bytes = 28737
compression rate = 0.572051
```

Step 4.6: Writing and Reading Bitstreams

- Final stage: writing and reading compressed file
  - In order to accomplish this, we have to:
    - Be able to write and read a variable number of bits to file (since symbol code lengths are different)
    - But the smallest data unit we can access the file is 1 byte
    - → Let's consider a data structure that helps us do this

```
typedef struct bit_buffer
{
	FILE * fp;
	unsigned int word;
	int bit_pos;
} BITBUF;
```

word : variable number of bits are temporarily stored here

bit_pos : this keeps track of how many bits are stored in **word**

fp : it is convenient to have the file pointer here

```
void BITBUF_initialize(BITBUF * bitbuf, FILE * fp)
{
	bitbuf->fp = fp;
	bitbuf->word = 0;
	bitbuf->bit_pos = 0;
}
```

Step 4.6.1: Writing Bitstreams

```
void BITBUF_write_bits(BITBUF * bitbuf, unsigned int val, int bits)
{

      bitbuf->word |= (val << bitbuf->bit_pos);
      bitbuf->bit_pos += bits;

      while(bitbuf->bit_pos >= 8){
         unsigned char c = bitbuf->word;
         fputc(c, bitbuf->fp);
         bitbuf->word >>= 8;
         bitbuf->bit_pos -= 8;
      }
}
```

This bitwise-OR operator writes the code data (**`val`**) in **`word`** at the next bit position (**`bit_pos`**)

`bit_pos` indicates how many bits is written on **`word`**

When more than 8 bits are written, take 1 byte from **`word`** and write it in the file → continue this until there are fewer than 8 bits stored in **`word`**

Step 4.6.1: Writing Bitstreams

```
void BITBUF_write_bits(BITBUF * bitbuf, unsigned int val, int bits)
{
        bitbuf->word |= (val << bitbuf->bit_pos);
        bitbuf->bit_pos += bits;
        while(bitbuf->bit_pos >= 8){
            unsigned char c = bitbuf->word;
            fputc(c, bitbuf->fp);
            bitbuf->word >>= 8;
            bitbuf->bit_pos -= 8;
        }
}
```

This bitwise-OR operator writes the code data (**val**) on **word** at the next bit position (**bit_pos**)

bit_pos indicates how many bits is written on **word**

```
 'h'  'e'  'l' 'l' 'o'  ' '   'w'  'o'  'r'  'l'  'd'   '!'
101 1000  00  00  11 1001 0110 11 0111 00 0100 0101
```

Remember that this code reads from left to right (which is the opposite of how we stored the symbol codes)

```
word = (0..000 00000000), bit_pos = 0  ← val = (101), bits = 3 ('h')
  word |= ((101) << 0) → (101), bit_pos += 3 → 0
word = (0..000 00000101), bit_pos = 3  ← val = (0001), bits = 4 ('e')
  word |= ((0001) << 3) → (0001101), bit_pos += 4 → 7
word = (0..000 00001101), bit_pos = 7  ← val = (00),   bits = 2 ('l')
  word |= ((00) << 7) → (000001101), bit_pos += 2 → 9
word = (0..000 00001101), bit_pos = 9  → fputc(00001101) (0x0d)
word = (0..000 00000000), bit_pos = 1
```

Step 4.6.1: Writing Bitstreams

```
void BITBUF_write_bits(BITBUF * bitbuf, unsigned int val, int bits)
{
        bitbuf->word |= (val << bitbuf->bit_pos);
        bitbuf->bit_pos += bits;
        while(bitbuf->bit_pos >= 8){
            unsigned char c = bitbuf->word;
            fputc(c, bitbuf->fp);
            bitbuf->word >>= 8;
            bitbuf->bit_pos -= 8;
        }
}
```

Remember that this code reads from left to right (which is the opposite of how we stored the symbol codes)

```
'h'  'e'  'l' 'l' 'o'  ' '   'w'  'o'  'r'  'l'  'd'   '!'
101 1000 00 00 11 1001 0110 11 0111 00 0100 0101
```

```
word = (0..000 00000000), bit_pos = 0  ← val = (101),  bits = 3 ('h')
word = (0..000 00000101), bit_pos = 3  ← val = (0001), bits = 4 ('e')
word = (0..000 00001101), bit_pos = 7  ← val = (00),   bits = 2 ('l')
word = (0..000 00001101), bit_pos = 9  → fputc(00001101) (0x0d)
word = (0..000 00000000), bit_pos = 1  ← val = (00),   bits = 2 ('l')
word = (0..000 00000000), bit_pos = 3  ← val = (11),   bits = 2 ('o')
word = (0..000 00011000), bit_pos = 5  ← val = (1001), bits = 4 (' ')
word = (0..001 00111000), bit_pos = 9  → fputc(00111000) (0x38)
word = (0..000 00000001), bit_pos = 1  ← val = (0110), bits = 4 ('w')
word = (0..000 00001101), bit_pos = 5  ← val = (11),   bits = 2 ('o')
word = (0..000 01101101), bit_pos = 7  ← val = (1110), bits = 4 ('r')
word = (0..111 01101101), bit_pos = 11 → fputc(01101101) (0x6d)
word = (0..000 00000111), bit_pos = 3  ← val = (00),   bits = 2 ('l')
word = (0..000 00000111), bit_pos = 5  ← val = (0010), bits = 4 ('d')
word = (0..000 01000111), bit_pos = 9  → fputc(01000111) (0x47)
word = (0..000 00000000), bit_pos = 1  ← val = (1010), bits = 4 ('!')
word = (0..000 00010100), bit_pos = 5  ← WE NEED TO DO SOMETHING ON THIS!!!
```

Things to Care About When Writing “Core” Functions

- “Core” functions has many uses in various scenarios
- You need to design these “core” functions so that they work correctly in all these scenarios
  - Not only do we need to make sure that these core functions are implemented correctly, but we also need to make sure that these core functions are USED correctly
- Let’s think about what can go wrong in this function
  - Places to check: function parameters, local variables
  - What kind of checks: out-of-range values, assumptions not satisfied, ... *many things* ...

```
void BITBUF_write_bits(BITBUF * bitbuf, unsigned int val, int bits)
{
      bitbuf->word |= (val << bitbuf->bit_pos);
      bitbuf->bit_pos += bits;
      while(bitbuf->bit_pos >= 8){
          unsigned char c = bitbuf->word;
          fputc(c, bitbuf->fp);
          bitbuf->word >>= 8;
          bitbuf->bit_pos -= 8;
      }
}
```

Things to Care About When Writing "Core" Functions

- Let's think about what can go wrong in this function

```
void BITBUF_write_bits(BITBUF * bitbuf, unsigned int val, int bits)
{
      bitbuf->word |= (val << bitbuf->bit_pos);
      bitbuf->bit_pos += bits;
      while(bitbuf->bit_pos >= 8){
          unsigned char c = bitbuf->word;
          fputc(c, bitbuf->fp);
          bitbuf->word >>= 8;
          bitbuf->bit_pos -= 8;
      }
}
```

1. Valid value of `bitbuf->word` : all bits above `bitbuf->bit_pos` must be 0 because we are storing `val` at this bit location
   Ex: `word = (0..000 00011000), bit_pos = 5` ← bit 5 and higher must all be 0s in `word`
2. Valid value of `val` :
   A) In order to satisfy condition 1, all bits above `bits` must be 0, otherwise condition 1 will be violated after executing these statements
   B) `(val << bitbuf->bit_pos)` must fit in 32 bits
   → depends on the value `bitbuf->bit_pos` at the beginning of this function
   → what kind of assumptions can we make on `bitbuf->bit_pos` at the beginning of this function?

Things to Care About When Writing "Core" Functions

- Let's think about what can go wrong in this function

```
void BITBUF_write_bits(BITBUF * bitbuf, unsigned int val, int bits)
{
      bitbuf->word |= (val << bitbuf->bit_pos);
      bitbuf->bit_pos += bits;
      while(bitbuf->bit_pos >= 8){
         unsigned char c = bitbuf->word;
         fputc(c, bitbuf->fp);
         bitbuf->word >>= 8;
         bitbuf->bit_pos -= 8;
      }
}
```

3. What is a reasonable assumption of `bitbuf->bit_pos` at the beginning of this function?
   - At the end of this function, `bitbuf->bit_pos` will be less than 8 → Then let's assume that this is true at the beginning as well
   - Then in order to guarantee that `(val << bitbuf->bit_pos)` fits in 32 bits, `val` needs to be 24 bits or less under the assumption of `(bitbuf->bit_pos < 8)`
4. Valid value of `bits` : between 0 and 24 (due to the condition on `val`)

Assertions

- Use “assertions” in the code to ensure “correctness”
  - Specify what you “assumed” in the code (than may not be applicable to all use-cases)
  - Specifiy what (you think) should be always satisfied

```
void abort_program(const char * msg)
{
	printf(“abort: (%s) is FALSE!\n", msg);
	exit(-1);
}
#define ENABLE_ASSERT
#ifdef ENABLE_ASSERT
#define ASSERT(n)  if(!(n))  abort_program(#n)
#else
#define ASSERT(n)  (n)
#endif
void BITBUF_write_bits(BITBUF * bitbuf, unsigned int val, int bits)
{
	ASSERT(bitbuf->bit_pos >= 0 && bitbuf->bit_pos < 8);
	ASSERT(bits >= 0 && bits <= 24);
	bitbuf->word |= (val << bitbuf->bit_pos);
	...
}
```

There can be many ways to implement assertion. One popular way is to just print some messages or store those message in a file.

Here is a most drastical way: simply abort the program if assertion fails

3. What is a reasonable assumption of `bitbuf->bit_pos` at the beginning of this function?
   - Let’s assume that `bitbuf->bit_pos` is less than 8
4. Valid value of `bits` : between 0 and 24 (due to the condition on `val`)

Assertions

```
#define ENABLE_ASSERT
#ifdef ENABLE_ASSERT
#define ASSERT(n)  if(!(n))  abort_program(#n)
#else
#define ASSERT(n)  (n)
#endif
void BITBUF_write_bits(BITBUF * bitbuf, unsigned int val, int bits)
{
      ASSERT(bitbuf->bit_pos >= 0 && bitbuf->bit_pos < 8);
      ASSERT(bits >= 0 && bits <= 24);
      bitbuf->word |= (val << bitbuf->bit_pos);
      ...
}
```

`#n` : replaces the *token* `n` into string `"n"`

`ASSERT(i<0)` → `if(!(i<0)) abort_program("i<0");`

- If `ENABLE_ASSERT` is defined by `#define`,

  ```
  ASSERT(bitbuf->bit_pos >= 0 && bitbuf->bit_pos < 8);
  ASSERT(bits >= 0 && bits <= 24);
  ```

  will be replaced by:

  ```
  if(!(bitbuf->bit_pos >= 0 && bitbuf->bit_pos < 8))
      abort_program("bitbuf->bit_pos >= 0 && bitbuf->bit_pos < 8");
  if(!(bits >= 0 && bits <= 24))
      abort_program("bits >= 0 && bits <= 24");
  ```

- If `ENABLE_ASSERT` is NOT defined,

  ```
  ASSERT(bitbuf->bit_pos >= 0 && bitbuf->bit_pos < 8);
  ASSERT(bits >= 0 && bits <= 24);
  ```

  will be replaced by:

  ```
  (bitbuf->bit_pos >= 0 && bitbuf->bit_pos < 8);
  (bits >= 0 && bits <= 24);
  ```

  which does not do anything here in this case

Assertions

```
#define ASSERT_BITS(word, bits)    ASSERT((word & ~((1 << bits) - 1)) == 0)

void BITBUF_write_bits(BITBUF * bitbuf, unsigned int val, int bits)
{
      ASSERT(bitbuf->bit_pos >= 0 && bitbuf->bit_pos < 8);
      ASSERT(bits >= 0 && bits <= 24);
      ASSERT_BITS(bitbuf->word, bitbuf->bit_pos);
      ASSERT_BITS(val, bits);
      bitbuf->word |= (val << bitbuf->bit_pos);
      ...
}
```

- If `ENABLE_ASSERT` is defined by `#define`,
  `ASSERT_BITS(val, bits);`
  will be replaced by:
  ```
  if(!((val & ~((1 << bits) - 1)) == 0))
      abort_program("val & ~((1 << bits) - 1)) == 0");
  ```

Ex1:

```
                  bit = 5, val = (0..000 00011000)
                   (1 << bits) = (0..000 00100000)
               (1 << bits) - 1 = (0..000 00011111)
            ~((1 << bits) - 1) = (1..111 11100000)
      val & ~((1 << bits) - 1) = (0..000 00000000)
```

ASSERT succeeds

Ex2:

```
                  bit = 5, val = (0..000 01011000)
                   (1 << bits) = (0..000 00100000)
               (1 << bits) - 1 = (0..000 00011111)
            ~((1 << bits) - 1) = (1..111 11100000)
      val & ~((1 << bits) - 1) = (0..000 01000000)
```

ASSERT fails

Step 4.6.1: Writing Bitstreams

```
void BITBUF_flush_bits(BITBUF * bitbuf)
{
        unsigned char c = bitbuf->word;
        ASSERT(bitbuf->bit_pos >= 0 && bitbuf->bit_pos < 8);
        ASSERT_BITS(bitbuf->word, bitbuf->bit_pos);
        if(bitbuf->bit_pos > 0){
            fputc(c, bitbuf->fp);
            bitbuf->word = 0;
            bitbuf->bit_pos = 0;
        }
}
```

After converting all symbols, we need to flush out the data still stored in **word** to the file

This indicates that there are some data left in **word**

```
word = (0..000 00000000), bit_pos = 0  ← val = (101),   bits = 3 ('h')
word = (0..000 00000101), bit_pos = 3  ← val = (0001),  bits = 4 ('e')
word = (0..000 00001101), bit_pos = 7  ← val = (00),    bits = 2 ('l')
word = (0..000 00001101), bit_pos = 9  → fputc(00001101) (0x0d)
word = (0..000 00000000), bit_pos = 1  ← val = (00),    bits = 2 ('l')
word = (0..000 00000000), bit_pos = 3  ← val = (11),    bits = 2 ('o')
word = (0..000 00011000), bit_pos = 5  ← val = (1001),  bits = 4 (' ')
word = (0..001 00111000), bit_pos = 9  → fputc(00111000) (0x38)
word = (0..000 00000001), bit_pos = 1  ← val = (0110),  bits = 4 ('w')
word = (0..000 00001101), bit_pos = 5  ← val = (11),    bits = 2 ('o')
word = (0..000 01101101), bit_pos = 7  ← val = (1110),  bits = 4 ('r')
word = (0..111 01101101), bit_pos = 11 → fputc(01101101) (0x6d)
word = (0..000 00000111), bit_pos = 3  ← val = (00),    bits = 2 ('l')
word = (0..000 00000111), bit_pos = 5  ← val = (0010),  bits = 4 ('d')
word = (0..000 01000111), bit_pos = 9  → fputc(01000111) (0x47)
word = (0..000 00000000), bit_pos = 1  ← val = (1010),  bits = 4 ('!')
word = (0..000 00010100), bit_pos = 5  ← WE NEED TO DO SOMETHING ON THIS!!!
```

Step 4.6.1: Writing Bitstreams

```
void BITBUF_flush_bits(BITBUF * bitbuf)
{
        unsigned char c = bitbuf->word;
        ASSERT(bitbuf->bit_pos >= 0 && bitbuf->bit_pos < 8);
        ASSERT_BITS(bitbuf->word, bitbuf->bit_pos);
        if(bitbuf->bit_pos > 0){
            fputc(c, bitbuf->fp);
            bitbuf->word = 0;
            bitbuf->bit_pos = 0;
        }
}
```

```
word = (0..000 00001101), bit_pos = 9  → fputc(00001101) (0x0d)
word = (0..001 00111000), bit_pos = 9  → fputc(00111000) (0x38)
word = (0..111 01101101), bit_pos = 11 → fputc(01101101) (0x6d)
word = (0..000 01000111), bit_pos = 9  → fputc(01000111) (0x47)
word = (0..000 00010100), bit_pos = 5  → fputc(00010100) (0x17) <flush_bits>
```

byte 4	byte 3	byte 2	byte 1	byte 0
00010100	01000111	01101101	00111000	00001101

Step 4.6.2: “write_compressed_file”

```
FILE * open_processed_file(const char * fname, const char * fmode, const char * fext)
{
        char comp_filename[MAX_FILENAME_LENGTH + 1 + 4];
        FILE * fp_comp;
        ASSERT(strlen(fname) <= MAX_FILENAME_LENGTH);
        sprintf(comp_filename, "%s%s", fname, fext);
        fp_comp = fopen(comp_filename, fmode);
        if(fp_comp == 0)
            printf("cannot open <%s> with mode(%s)\n", comp_filename, fmode);
        return fp_comp;
}
void HINFO_write_compressed_file(HINFO * hinfo, const char * fname)
{
        SINFO * sinfo;
        BITBUF bitbuf;
        FILE * fp_comp = open_processed_file(fname, "wb");
        if(fp_comp == 0) return;
        BITBUF_initialize(&bitbuf, fp_comp, ".hmc");
        rewind(hinfo->fp_txt);
        while(1){
            int c = fgetc(hinfo->fp_txt);
            if(c == EOF) break;
            sinfo = &hinfo->sinfo[c];
            BITBUF_write_bits(&bitbuf, sinfo->code, sinfo->code_length);
        }
        BITBUF_flush_bits(&bitbuf);
        fclose(fp_comp);
}
```

fname : “file.txt”

comp_filename : “file.txt.hmc”

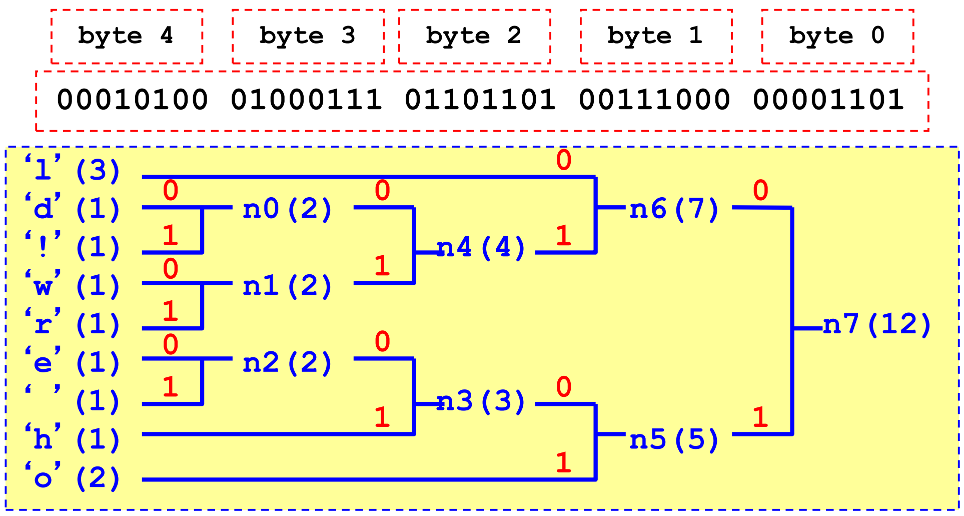
`rewind(hinfo->fp_txt);` bring the file pointer back to the beginning of file

Write symbol codes to the output file

Note that we have not yet stored the Huffman tree (we will do this later)

Step 4.6.3: Checking the Compressed File

- Let's check that the compressed file can be correctly decoded (remember that we also need to write the Huffman tree information in the file)
- Let's review the decoding process first
  1. Scan the bit-stream 1 bit at a time
  2. Start from the root of the tree
  3. If the current bit is '0', then go to the left child, otherwise go to the right child
  4. If the leaf is reached, output that symbol, and go to 2.

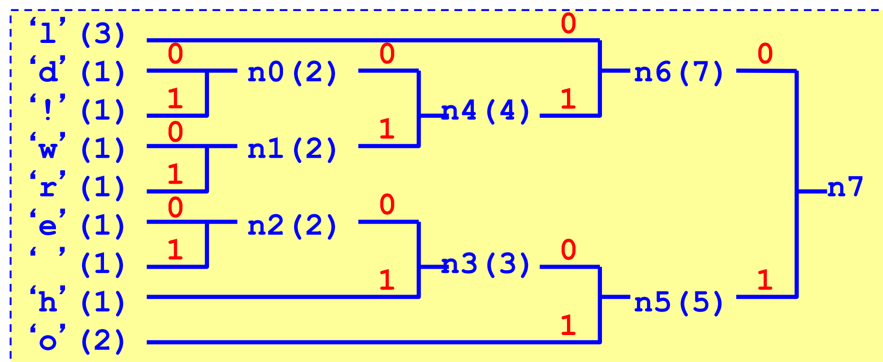


Decoding the Compressed File

1. Scan the bit-stream 1 bit at a time
2. Start from the root of the tree
3. If the current bit is ‘0’, then go to the left child, otherwise go to the right child
4. If the leaf is reached, output that symbol, and go to 2.

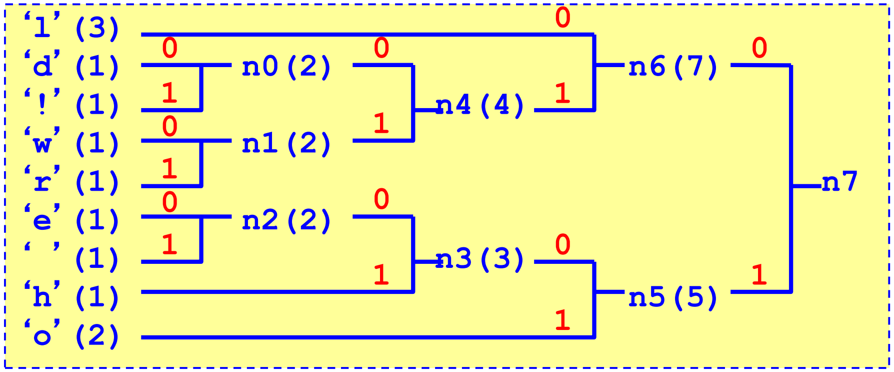
byte 4	byte 3	byte 2	byte 1	byte 0	
00010100	01000111	01101101	00111000	00001101	
76543210	76543210	76543210	76543210	76543210	← bit positions

```
byte(0),bit(0): n7(1)→ n5(0)→ n3(1)→ 'h'
byte(0),bit(3): n7(1)→ n5(0)→ n3(0)→ n2(0)→ 'e'
byte(0),bit(7): n7(0)→ n6(0)→ 'l'
byte(1),bit(1): n7(0)→ n6(0)→ 'l'
byte(1),bit(3): n7(1)→ n5(1)→ 'o'
byte(1),bit(5): n7(1)→ n5(0)→ n3(0)→ n2(1)→ ' '
byte(2),bit(1): n7(0)→ n6(1)→ n4(1)→ n1(0)→ 'w'
byte(2),bit(5): n7(1)→ n5(1)→ 'o'
byte(2),bit(7): n7(0)→ n6(1)→ n4(1)→ n1(1)→ 'r'
byte(3),bit(3): n7(0)→ n6(0)→ 'l'
byte(3),bit(5): n7(0)→ n6(1)→ n4(0)→ n0(0)→ 'd'
byte(4),bit(1): n7(0)→ n6(1)→ n4(0)→ n0(1)→ '!'
```



Step 4.6.3: Reading Bitstreams

```
unsigned int BITBUF_read_one_bit(BITBUF * bitbuf)
{
        unsigned int data;
        if(bitbuf->bit_pos == 0){
            int c = fgetc(bitbuf->fp);
            if(c == EOF) printf("End-Of-File!!\n);
            bitbuf->word = c;
            bitbuf->bit_pos = 8;
        }
        data = bitbuf->word & 1;
        bitbuf->bit_pos --;
        bitbuf->word >>= 1;
        return data;
}
```



(00001101)
n7(1)→ n5(0)→ n3(1)→ 'h'
n7(1)→ n5(0)→ n3(0)→ n2(0)→ 'e'
n7(0)→ n6 ......

```
int HINFO_decode_symbol(HINFO * hinfo, BITBUF * bitbuf)
{
        SINFO * sinfo = hinfo->root_node;
        while(1){
            unsigned int bit = BITBUF_read_one_bit(bitbuf);
            sinfo = (bit) ? sinfo->right : sinfo->left;
            if(sinfo->right == 0)
                    break;
        }
        return sinfo->index;
}
```

Traverse the Huffman tree from the root node

```
if(bit != 0)
  sinfo = sinfo->right;
else
  sinfo = sinfo->left;
```

`sinfo->index` is the symbol value itself

Step 4.6.3: Reading Bitstreams

```
unsigned int BITBUF_read_one_bit(BITBUF * bitbuf)
{
        unsigned int data;
        if(bitbuf->bit_pos == 0){
            int c = fgetc(bitbuf->fp);
            if(c == EOF) printf("End-Of-File!!\n");
            bitbuf->word = c;
            bitbuf->bit_pos = 8;
        }
        data = bitbuf->word & 1;
        bitbuf->bit_pos --;
        bitbuf->word >>= 1;
        return data;
}
int HINFO_decode_symbol(HINFO * hinfo, BITBUF * bitbuf)
{
        SINFO * sinfo = hinfo->root_node;
        while(1){
            unsigned int bit = BITBUF_read_one_bit(bitbuf);
            sinfo = (bit) ? sinfo->right : sinfo->left;
            if(sinfo->right == 0)
                        break;
        }
        return sinfo->index;
}
```

If there are no bits stored in `word`, read 1 byte from file

`data` is the LSB of `word`

Shift `word` 1 bit to right

```
word = (00000000), bit_pos = 0
fgetc()
word = (00001101), bit_pos = 8
data = 1
word = (00000110), bit_pos = 7
```

Step 4.6.3: Reading Bitstreams

```
unsigned int BITBUF_read_one_bit(BITBUF * bitbuf)
{
        unsigned int data;
        if(bitbuf->bit_pos == 0){
            int c = fgetc(bitbuf->fp);
            if(c == EOF) printf("End-Of-File!!\n);
            bitbuf->word = c;
            bitbuf->bit_pos = 8;
        }
        data = bitbuf->word & 1;
        bitbuf->bit_pos --;
        bitbuf->word >>= 1;
        return data;
}
int HINFO_decode_symbol(HINFO * hinfo, BITBUF * bitbuf)
{
        SINFO * sinfo = hinfo->root_node;
        while(1){
            unsigned int bit = BITBUF_read_one_bit(bitbuf);
            sinfo = (bit) ? sinfo->right : sinfo->left;
            if(sinfo->right == 0)
                  break;
        }
        return sinfo->index;
}
```

```
word = (00000000), bit_pos = 0  → fgetc()
word = (00001101), bit_pos = 8  → return 1
word = (00000110), bit_pos = 7  → return 0
word = (00000011), bit_pos = 6  → return 1
word = (00000001), bit_pos = 5  → return 1
word = (00000000), bit_pos = 4  → return 0
word = (00000000), bit_pos = 3  → return 0
word = (00000000), bit_pos = 2  → return 0
word = (00000000), bit_pos = 1  → return 0
word = (00000000), bit_pos = 0  → fgetc()
word = (00111000), bit_pos = 8  → return 0
word = (00011100), bit_pos = 7  → return 0
......
```

```
n7(1)→ n5(0)→ n3(1)→ 'h'
n7(1)→ n5(0)→ n3(0)→ n2(0)→ 'e'
n7(0)→ n6(0)→ 'l'
n7(0)→ n6(0)→ 'l'
n7(1)→ n5(1)→ 'o'
n7(1)→ n5(0)→ n3(0)→ n2(1)→ ' '
n7(0)→ n6(1)→ n4(1)→ n1(0)→ 'w'
n7(1)→ n5(1)→ 'o'
n7(0)→ n6(1)→ n4(1)→ n1(1)→ 'r'
n7(0)→ n6(0)→ 'l'
n7(0)→ n6(1)→ n4(0)→ n0(0)→ 'd'
n7(0)→ n6(1)→ n4(0)→ n0(1)→ '!'
```

Step 4.6.3: Reading Bitstreams

```
void HINFO_check_compressed_file(HINFO * hinfo, const char * fname)
{
      int ccount = 0;
      BITBUF bitbuf;
      FILE * fp_comp = open_processed_file(fname, "rb", ".hmc");
      if(fp_comp == 0)
         return;
      BITBUF_initialize(&bitbuf, fp_comp);
      while(ccount > 0){
         int c = HINFO_decode_symbol(hinfo, &bitbuf);
         printf("%c", c);
         ccount --;
      }
      fclose(fp_comp);
}
```

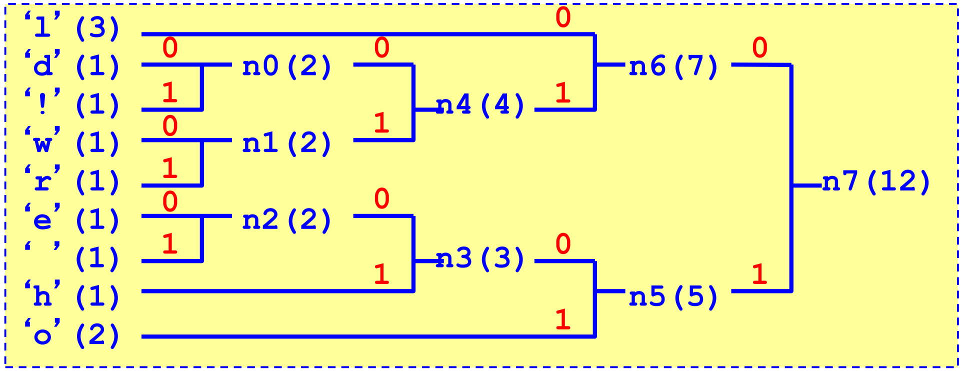
Step 4.6.3: Checking the Compressed File

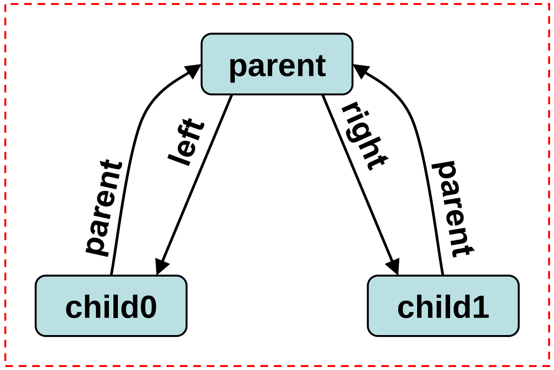
```
int main(int argc, char * argv[])
{
      HINFO hinfo;
      HINFO_initialize(&hinfo);
      if(argc != 2){
         printf("invalid command!\n");
         printf("usage: %s <filename>\n", argv[0]);
         return 0;
      }
      hinfo.fp_txt = fopen(argv[1], "rb");
      if(hinfo.fp_txt == 0){
         printf("cannot open file <%s>!!\n", argv[1]);
         return 0;
      }
      HINFO_count_symbols(&hinfo);
      HINFO_create_huffman_tree(&hinfo);
      HINFO_print_sorted_symbols(&hinfo);
      HINFO_write_compressed_file(&hinfo, argv[1]);
      HINFO_check_compressed_file(&hinfo, argv[1]);
      fclose(hinfo.fp_txt)
      return 1;
}
```

1. Initialize hinfo
2. Open input file
3. Count the occurrence of each symbol
4. Create Huffman tree
5. Write compressed file
6. Check compressed file

Step 4.6.4: Writing and Reading Huffman Tree

- We are FINALLY down to the last item: writing and reading the Huffman Tree
- How should we store the Huffman Tree → think about what kind of information is needed to reconstruct the Huffman Tree
  - – Information about left child, right child and parent
  - → But we don't need to store all these information because these are redundant
    `(sinfo->left != 0) && (sinfo->left->parent == sinfo)` always hold
  - → Let's store the child information at the internal nodes
  - → Do not store pointer values directly to the file (because memory allocation is different each time you run your program)
  - → Instead, use `index` value to specify the child nodes





`parent`	n4	n4	n3	n5	n6	n6	n7	***
`hinfo->inode[256]`	n0	n1	n2	n3	n4	n5	n6	n7
`left, right`	'd' '!'	'w' 'r'	'e' ' '	n2 'h'	n0 n1	n3 'o'	'l' n4	n6 n7

Step 4.6.4: “write_huffman_tree”

```
void SINFO_write_children(SINFO * sinfo, BITBUF * bitbuf)
{
        if(sinfo->left == 0 || sinfo->right == 0) return;
        BITBUF_write_bits(bitbuf, sinfo->left->left == 0, 1);
        BITBUF_write_bits(bitbuf, sinfo->left->index, 8);
        BITBUF_write_bits(bitbuf, sinfo->right->left == 0, 1);
        BITBUF_write_bits(bitbuf, sinfo->right->index, 8);
}
```

1 bit to indicate if the child is a leaf (1) or an internal node (0)

Store left child’s index (8 bits)

Store right child’s index (8 bits)

```
void HINFO_write_huffman_tree(HINFO * hinfo, BITBUF * bitbuf)
{
        int i;
        fwrite(&hinfo->text_count, 4, 1, bitbuf->fp);
        fputc(hinfo->symbol_count - 1, bitbuf->fp);
        for(i = 0; i < hinfo->symbol_count - 1; i ++){

            SINFO_write_children(&hinfo->inode[i], bitbuf);
        }
}
```

Store text_count (4 bytes)

Store symbol_count – 1 (1 byte)
(why symbol_count – 1???)

Write child information for all internal nodes

# of internal nodes = # of symbols – 1

parent	n4		n4		n3		n5		n6		n6		n7		***	
hinfo->inode[256]	n0		n1		n2		n3		n4		n5		n6		n7	
left, right	‘d’	‘!’	‘w’	‘r’	‘e’	‘ ’	n2	‘h’	n0	n1	n3	‘o’	‘l’	n4	n6	n7

Step 4.6.4: “read_huffman_tree”

This function is the generalized version of **`BITBUF_read_one_bit()`**
We need this function because we need to read variable bits from the file

```
unsigned int BITBUF_read_bits(BITBUF * bitbuf, int bits)
{
        unsigned int data;
        while(bitbuf->bit_pos < bits){
            int c = fgetc(bitbuf->fp);
            if(c == EOF) printf("End-Of-File!!\n);
            bitbuf->word |= (c << bitbuf->bit_pos);
            bitbuf->bit_pos += 8;
        }
        data = bitbuf->word & ((1 << bits) - 1);
        bitbuf->bit_pos -= bits;
        bitbuf->word >>= bits;
        return data;
}
```

If there are not enough bits stored in **`word`**, read 1 byte from file → continue this until you have enough data in **`word`**

`data` is the lower **`(bits)`** bits of **`word`**

Shift **`word (bits)`** bits to right

Step 4.6.4: "read_huffman_tree"

```
#define LINK_SINFO(ch_node, p_node, child)          \
      ASSERT(p_node->child == 0);                   \
      p_node->child = ch_node;                      \
      ch_node->parent = p_node
```

A macro that links parent and child nodes ("**child**" should be "**left**" or "**right**") '****' is needed at the end of line so that the macro definition can span multiple lines

```
void HINFO_read_huffman_tree(HINFO * hinfo, BITBUF * bitbuf)
{
      int i;
      SINFO * sinfo, * child;
      fread(&hinfo->text_count, 4, 1, bitbuf->fp);
      hinfo->symbol_count = fgetc(bitbuf->fp) + 1;
      printf("hinfo->text_count = %d, hinfo->scount = %d\n",
         hinfo->text_count, hinfo->symbol_count);

      for(i = 0; i < hinfo->symbol_count - 1; i ++){
         sinfo = &hinfo->inode[i];
         child = (BITBUF_read_bits(bitbuf, 1)) ? hinfo->sinfo : hinfo->inode;
         child += BITBUF_read_bits(bitbuf, 8);
         LINK_SINFO(child, sinfo, left);
         child = (BITBUF_read_bits(bitbuf, 1)) ? hinfo->sinfo : hinfo->inode;
         child += BITBUF_read_bits(bitbuf, 8);
         LINK_SINFO(child, sinfo, right);
      }
      hinfo->root_node = &hinfo->inode[hinfo->symbol_count - 2];
      ASSERT(hinfo->root_node->left && hinfo->root_node->right);
      SINFO_assign_code(hinfo->root_node);
}
```

Read text_count (4 bytes)

Read symbol_count – 1 (1 byte)

Read the child information for the internal nodes and link the nodes

Step 4.6.4: “read_huffman_tree”

```
#define LINK_SINFO(ch_node, p_node, child)        \
      ASSERT(p_node->child == 0);                 \
      p_node->child = ch_node;                    \
      ch_node->parent = p_node

      for(i = 0; i < hinfo->symbol_count - 1; i ++){
         sinfo = &hinfo->inode[i];

         child = (BITBUF_read_bits(bitbuf, 1)) ? hinfo->sinfo : hinfo->inode;
         child += BITBUF_read_bits(bitbuf, 8);
         LINK_SINFO(child, sinfo, left);

         child = (BITBUF_read_bits(bitbuf, 1)) ? hinfo->sinfo : hinfo->inode;
         child += BITBUF_read_bits(bitbuf, 8);
         LINK_SINFO(child, sinfo, right);
      }
```

Link left child

Link right child

`child = (BITBUF_read_bits(bitbuf, 1)) ? hinfo->sinfo : hinfo->inode;`

– `child` points to either `hinfo->sinfo` (symbol leaf node array) or `hinfo->inode` (internal node array)

`child += BITBUF_read_bits(bitbuf, 8);`

– `child` is incremented by the index value (from file) (`hinfo->sinfo[index]` or `hinfo->inode[index]`)

`LINK_SINFO(child, sinfo, left);` → macro call that will be replaced by:

```
ASSERT(sinfo->left == 0);
sinfo->left = child;
child->parent = sinfo;
```

Step 4.7: Putting it All Together

```
void HINFO_write_compressed_file(HINFO * hinfo, const char * fname)
{
      SINFO * sinfo;
      BITBUF bitbuf;
      FILE * fp_comp = open_compressed_file(fname, "wb");
      if(fp_comp == 0) return;
      BITBUF_initialize(&bitbuf, fp_comp);

      HINFO_write_huffman_tree(hinfo, &bitbuf);

      rewind(hinfo->fp_txt);
      while(1){
         int c = fgetc(hinfo->fp_txt);
         if(c == EOF) break;
         sinfo = &hinfo->sinfo[c];
         BITBUF_write_bits(&bitbuf, sinfo->code, sinfo->code_length);
      }
      BITBUF_flush_bits(&bitbuf);
      fclose(fp_comp);
}
```

Huffman tree information is stored at the beginning of the compressed file

Step 4.7: Putting it All Together

```
void HINFO_write_compressed_file(HINFO * hinfo, const char * fname)
{
      SINFO * sinfo;
      BITBUF bitbuf;
      FILE * fp_comp = open_compressed_file(fname, "wb");
      if(fp_comp == 0) return;
      BITBUF_initialize(&bitbuf, fp_comp);

      HINFO_write_huffman_tree(hinfo, &bitbuf);

      rewind(hinfo->fp_txt);
      while(1){
         int c = fgetc(hinfo->fp_txt);
         if(c == EOF) break;
         sinfo = &hinfo->sinfo[c];
         BITBUF_write_bits(&bitbuf, sinfo->code, sinfo->code_length);
      }
      BITBUF_flush_bits(&bitbuf);
      fclose(fp_comp);
}
```

Step 4.7: Putting it All Together

```
void HINFO_read_compressed_file(HINFO * hinfo, const char * fname)
{
        FILE * fp_comp, * fp_out;
        int ccount = hinfo->text_count;
        BITBUF bitbuf;
        HINFO_initialize(hinfo);
        fp_comp = open_processed_file(fname, "rb", ".hmc");
        if(fp_comp == 0) return;
        BITBUF_initialize(&bitbuf, fp_comp);

        HINFO_read_huffman_tree(hinfo, &bitbuf);

        ccount = hinfo->text_count;
        fp_out = open_processed_file(fname, "wb", ".txt");
        while((ccount --) > 0){
            int c = HINFO_decode_symbol(hinfo, &bitbuf);
            if(c == EOF)
                        break;
            fputc(c, fp_out);
        }
        fclose(fp_comp);
        fclose(fp_out);
        printf("text_count = %d\n", hinfo->text_count);
}
```

Step 4.7: Putting it All Together

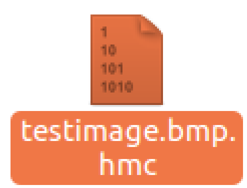
```
int main(int argc, char * argv[])
{
      HINFO hinfo, hinfo2;
      HINFO_initialize(&hinfo);
      if(argc != 2){
         printf("invalid command!\n");
         printf("usage: %s <filename>\n", argv[0]);
         return 0;
      }
      hinfo.fp_txt = fopen(argv[1], "rb");
      if(hinfo.fp_txt == 0){
         printf("cannot open file <%s>!!\n", argv[1]);
         return 0;
      }
      HINFO_count_symbols(&hinfo);
      HINFO_create_huffman_tree(&hinfo);
      HINFO_print_sorted_symbols(&hinfo);
      HINFO_write_compressed_file(&hinfo, argv[1]);
      HINFO_read_compressed_file(&hinfo2, argv[1]);
      fclose(hinfo.fp_txt)
      return 1;
}
```

If you want to use functions from the math library in C, it's not enough to put #include<math.h> at the top of your source code. In addition, you must add the -lm flag to the gcc compiler command in order to use math functions in your C code.

gcc huff_main.c -lm -o huff_main.o

`For Linux !!`

"testimage.bmp" selected (257.1 kB)



"result.bmp" selected (86.8 kB)

"testimage.bmp.hmc" selected (41.7 kB)

python

```
freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)
print(freq)
print("-------------------------------")
if DEBUG:
    print(" Char | Freq ")
    for key, c in freq:
        print (" %4r | %d" % (key, c))
    print ("--------------------")

nodes = freq

while len(nodes) > 1:
    key1, c1 = nodes[-1]
    key2, c2 = nodes[-2]
    nodes = nodes[:-2]

    node = NodeTree(key1, key2)
    #print(node)
    #print("----------------------")
    nodes.append((node, c1 + c2))
    # Re-sort the list
    nodes = sorted(nodes, key=lambda x: x[1], reverse=True)  # Sort by frequency which is x[1]

if DEBUG:
    print ("left: %s" % nodes[0][0].nodes()[0])
    print ("right: %s" % nodes[0][0].nodes()[1])


huffmanCode = huffmanCodeTree(nodes[0][0])

print (" Char | Freq  | Huffman code ")
print ("-----------------------------")
for char, frequency in freq:
    print (" %-4r | %5d | %12s" % (char, frequency, huffmanCode[char]))
```

huffman_main

Char	Freq	Huffman code
' '	381	110
'e'	216	000
'o'	164	1011
'n'	147	1000
'i'	138	0111
't'	133	0101
'a'	128	0100
's'	117	0010
'r'	104	11110
'd'	87	11100
'f'	81	10101
'h'	80	10100
'm'	77	10011
'c'	70	01101
'l'	60	00110
'u'	59	111111
'p'	48	111110
'g'	44	111010
'y'	35	100100
'b'	33	011000
'w'	21	1110110
'.'	19	1001010
','	18	0110011
'v'	17	0110010
'H'	16	0011110
'\n'	13	11101111
'"'	8	00111110
'x'	7	00111001
'T'	6	111011101
'I'	5	100101110
'M'	5	100101101
'-'	5	100101100
'A'	4	001111110
'q'	4	001110101
"'"	4	001110100
'D'	3	001110000
'1'	3	1001011111
'C'	3	1001011110
'('	3	1110111001
')'	3	1110111000
'9'	2	0011111110
'5'	2	0011100011
'R'	2	0011100010
'F'	2	0011101101
'k'	2	0011101100
'S'	2	0011101111
'P'	1	00111111110
'2'	1	001111111111
'/'	1	001111111110
'B'	1	00111011101
'j'	1	00111011100

```
left: e_s_l_D_R_5_x_'_q_k_F_j_B_S_H_"_A_9_P_/_2_a_t_b_v_,_c_i
right: n_y_._-_M_I_C_1_m_h_f_o_ _d_g_w_)_(_T_
_r_p_u
```

?

Exercise 4 (Compression)

1. Try compressing different kinds of files and observe the compression rate (see what happens when you compress an already compressed file)
2. Write a program that combines color quantization and Huffman coding
   - Input : 24-bit RGB image
   - Color quantization : 8-bit/pixel (colormap_size <= 256)
   - Huffman coding
     - Write compressed file (*.hmc)
     - Read compressed file and write the decompressed file as "result_huffman.bmp" → confirm that you can open this with the image viewer

   → Try different quantization levels, especially observe what happens to the compression rate when you decrease colormap_size