DESIGN.pdf ASGN5

Nathan Ong

October 28, 2021

1 Description

This collection of files contains a Huffman encoder and decoder that encodes information using strings that represent symbols based on how often they appear in an input. The encoder and decoder operate based on specific algorithms that will be described in further detail in Section 3 (Pseudocode and Structure).

The encoder file used to encode the input has the following command line options:

- -h Prints out help message then exits the program.
- -i *infile* Specifies input file to encode. Default input file is *stdin*.
- -o *outfile* Specifies output file to send compressed input to. Default output file is *stdout*.
- -v Prints compression statistics to stderr. Statistics include uncompressed file size, compressed file size, and space saving. (Space saving = 100 * (1 - compressed file size/ uncompressed file size))

The decoder file used to decode a compressed input has the following command line options:

- -h Prints out help message then exits the program.
- -i *infile* Specifies input file to decode. Default input file is *stdin*.
- -o *outfile* Specifies output file to send decompressed input to. Deafult output file is *stdout*.
- -v Prints decompression statistics to stderr. Statistics include decompressed file size, compressed file size, and space saving. (Space saving = 100 * (1 - compressed file size/ decompressed file size))

2 Files Included in the Directory

1. encode.c
   This file contains the implementation of the Huffman encoder.
2. decode.c
   This file contains the implementation of the Huffman decoder.
3. defines.h
   This file contains the definitions of the macros used in the implementation of this assignment.
4. header.h
   This file contains the struct definition for a file header.
5. node.h
   This file contains the Node ADT interface.
6. node.c
   This file contains the implementation of the Node ADT.

7. pq.h
   This file contains the Priority Queue ADT interface.

8. pq.c
   This file contains the implementation of the Priority Queue ADT.

9. code.h
   This file contains the Code ADT interface.

10. code.c
    This file contains the implementation of the Code ADT.

11. io.h
    This file contains the I/O module interface.

12. io.c
    This file contains the implementation of the I/O module.

13. stack.h
    This file contains the Stack ADT interface.

14. stack.c
    This file contains the implementation of the Stack ADT.

15. huffman.h
    This file contains the Huffman coding module interface.

16. huffman.c
    This file contains the implementation of the Huffman coding module.

3 Pseudocode and Structure

3.1 encode.c

```
while opt isnt -1
    indice through arguments
    check for required arguments if necessary
read file and make histogram
create priority queue
dequeue two nodes
join dequeued nodes together
build huffman tree with priority queue
create code and traverse tree post-order (start at root)
if node is leaf, save current code into table
else push 0 to code and recurse down left
after return from left, pop bit from code
push 1 to code and recurse down right
pop bit after returning
construct header
write header to outfile
write tree with dump tree
write code for each symbol to outfile
flush buffered codes
close files
```

3.2 decode.c

```
while opt isnt -1
    indice through arguments
    check for required arguments if necessary
read header and verify magic number (0xBEEFD00D)
if not matching, it is an invalid file
    display error message
set permissions of outfile using fchmod
read tree from infile into array that is tree-size bytes long
rebuild tree
iterate over contents of tree array
if element if 'L'
    create node of next element
if element if 'I'
    pop stack to get right child
    pop again to get left child
    join left and right nodes together and push back into stack
read infile using read-bit()
while decoded symbols doesnt match file size
    if bit value of 0 is read, go to left child
    if bit value of 1 is read, go to right child
    if at leaf node, write symbol to outfile
        reset current node to root of tree
close files
```

3.3 node.c

```
node create:
allocate memory for Node
set symbol pointer to given symbol argument
set frequency pointer to given frequency argument
null left and right pointers
return node

node delete:
free node pointer
null node pointer

node join:
add frequency of left and right nodes
set new symbol
make left and right pointers given arguments
return new node

node print:
print node symbol
print left and right nodes
```

3.4 pq.c

```
pq create:
allocate memory for priority queue
set capacity pointer to given argument
set size pointer to 0
```

```
allocate memory for items array
return priroity queue
```

pq delete:

```
free items array pointer
free priority queue pointer
nullify both pointers
```

pq empty:

```
if size value is 0 return true
otherwise return false
```

pq full:

```
if size value is equal to capacity return true
otherwise return false
```

pq size:

```
return size value
```

enqueue:

```
return false if queue is full
put node into item array in queue
increment size value
fix heap
return true
```

dequeue:

```
return false if queue size is 0
remove node from item array
decrement size value
fix heap
return true
```

pq print:

```
for all values from 0 to size value
    print items array element
```

3.5 code.c

code init:

```
set top pointer to 0
zero out bits in bit array
return code
```

code size:

```
return top value
```

code empty:

```
return true if top value is 0
otherwise return false
```

code full:

```
return true if top value is the max code size
otherwise return false
```

code set bit:

```
if argument out of range return false
```

```
set bit at index in code array to 1
return true
```

code clr bit:

```
if argument out of range return false
sets bit at index in code array to 0
return true
```

code get bit:

```
if argument out of range return false
if bit at index in code array is 0 return false
if bit at index in code array is 1 return true
```

code push bit:

```
if top value is the max code size return false
set bit
return true
```

code pop bit:

```
if top value is 0 return false
set given pointer to bit from get bit
return true
```

code print:

```
for o to top value, print bit array elements
```

3.6 io.c

read bytes:

```
read infile
increment total number of bytes read
return total bytes read
```

write bytes:

```
write in outfile
increment total number of bytes written
return total bytes written
```

read bit:

write code:

flush codes:

3.7 stack.c

stack create:

```
allocate memory for Stack size
set top pointer to 0
set capacity pointer to capacity argument
allocate memory for items array
if items pointer is false, free and null stack pointer
return stack
```

stack delete:

```
free items pointer
free stack pointer
```

stack empty:

```
if top value is 0, return true
otherwise return false
```

stack full:

```
if top value is equal to capacity, return true
otherwise return false
```

stack size:

```
return top value
```

stack push:

```
if top is less than maximum capacity
    insert Node into stack
    increment top value
    return true
otherwise return false
```

stack pop:

```
if top isn't 0
    point given pointer to element on top of the stack
    decrement top value
    return true
otherwise return false
```

stack print:

```
for all values from 0 to top value
    print stack element
```

3.8 huffman.c

build tree:

```
create node pointers and priority queue
for 0 to alphabet limit
create node if symbol in histogram and enqueue it
while priority queue is more than 1
    dequeue right then left node
    join left and right nodes
    enqueue joined node
dequeue joined node
```

build codes:

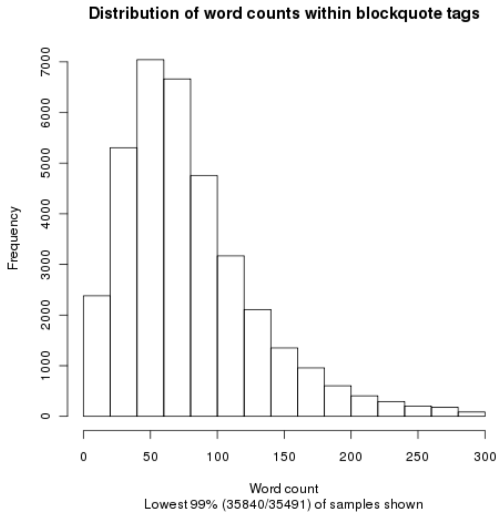
```
create code and traverse tree post-order (start at root)
if node is leaf, save current code into table
else push 0 to code and recurse down left
after return from left, pop bit from code
push 1 to code and recurse down right
pop bit after returning
```

dump tree:

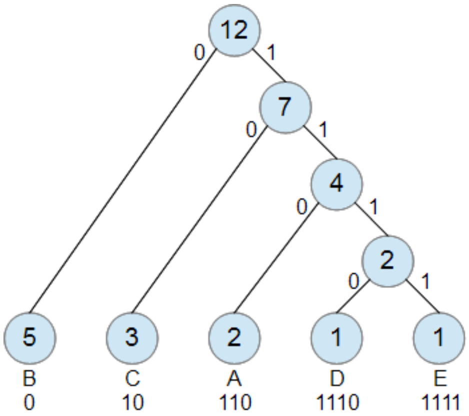
```
if there is still a root
    dump left and right nodes
    if left and right not roots
        write ’L’ (leaf)      else write ’I’ (interior node)
rebuild tree:
iterate over contents of tree array
if element if ’L’
    create node of next element
if element if ’I’
    pop stack to get right child
    pop again to get left child
    join left and right nodes together and push back into stack
return root node

delete tree:
if there is still a root
    delete left and right nodes
    delete root
```

4 Additional Diagrams

4.1 Histogram Example



4.2 Huffman Tree (String ”DAEBCBACBBBC”)



5 Additional Credits

1. The Makefile format was given by Sloan in his in-person section on October 5th.
2. Images use Creative Commons License CC BY-SA 3.0 or CC BY-SA 4.0.
   (a) CC BY-SA 3.0
   https://creativecommons.org/licenses/by-sa/3.0/deed.en
   (b) CC BY-SA 4.0
   https://creativecommons.org/licenses/by-sa/4.0/deed.en
3. Image Credits:
   - Histogram Example:
     Title: Preliminary Blockquote Word Count Histogram
     Author: Garamond Lethe
     Date Created: 2 Oct 2012
   - Huffman Tree Example:
     Title: Huffman Tree from 12 Letters
     Author: Cannot be listed
     Date Created: 2 July 2015