cse13s asgn6 DESIGN.pdf

Lucas Lee; CruzID: luclee

2/18/2022

1 Program Details

Program implements Huffman Encoder and Decoder, as well as structures for stacks and queues.

2 Huffman Encoder

encode():

1. -h prints out a help message
2. -i (infile) sets the file that is to be encoded. Default is stdin
3. -o (outfile) sets the file to be encoded to. Default is stdout
4. -v prints uncompressed file size, compressed file size, and space saving to stderr

To Huffman encode the file:

1. count the number of each unique symbol in the file - use readbytes() to read the infile to create a histogram
2. use priority queue to make a Huffman tree in buildtree()
3. use stack of bits to traverse Huffman tree. Make a code table to index each symbol.
4. use Code table to use buildcodes() function to create the codes for each symbol
5. create header struct values by using fstat() and write the header to outfile

6. encode the Huffman tree to the file using post-order traversal. Use dumptree() to write the huffman tree to outfile
7. for each symbol in the input file, write the corresponding code to output. Use writecodes() to write the correct corresponding code to outfile

decode():

1. -h prints help message
2. -i (infile) specifies the file to decode. Default is stdin
3. -o (outfile) specifies the file to decode to. Default is stdout
4. -v prints compressed file size, decompressed file size, space saving.

Algorithm steps for decoding:

1. read in the header from the input file and see if the magic number is correct
2. read Huffman tree from the input file. Using readbytes() read the dumped huffman tree into an array and use it to rebuild the tree
3. use a stack of nodes to reconstruct Huffman tree. This should be done in the rebuildtree() function
4. read each bit of the input file, reading a 0 when going down a left link (lower value) and reading a 1 when going down the right link (higher value)
5. when reaching an end node, emit the symbol and start traversing the tree again from the beginning and not going the same path
6. loop through this until you have written the same amount of symbols as the original file size

3 Nodes

We will make an ADT for nodes which will act as linked lists needed for Huffman trees.

1. make Node struct that has pointers to left and right Nodes, as well as variables to record the symbol and frequency of the variable

We will use the following functions in creating the Nodes:

1. NodeCreate(uint8 symbol, uint64 freq)
   (a) constructs node. Allocates space and initializes the variables to the values in the parameters
   (b) set left and right nodes to null
2. NodeDelete(Node N)
   (a) deletes Node struct N. Free memory allocated when creating node.
   (b) free node before setting the node pointer to NULL
3. NodeJoin(Node left, Node right)
   (a) creates a new node that returns a Node with the left and right value combined.
   (b) frequency will be the sum of left and right frequency
   (c) symbol will be $
   (d) set the left and right nodes used to create the parent to the parent's left and right node
4. NodePrint(Node n)
   (a) print node n for debugging purposes.

4 Priority Queue

We will also need to implement a priority queue struct for the enqueue.

1. pqCreate(uint32 capacity)
   (a) constructs a priority queue making the max capacity specified in the parameter
   (b) malloc using the size of the queue and set a variable size to keep track of the changing size of the queue
   (c) make an array of node pointers using malloc with the capacity given
2. pqDelete(PriorityQueue q)

(a) destroys Priority Queue q.
(b) set pointer to NULL after freeing the memory from q and the node array in q

3. bool pqEmpty(PriorityQueue q)
    (a) return true for empty q, false otherwise

4. bool pqFull(PriorityQueue q)
    (a) return true if full q, false otherwise

5. uint32 pqSize(PriorityQueue q)
    (a) variable to keep track of size of q
    (b) for each index in q, index size variable
    (c) return the size

6. bool enqueue(PriorityQueue q, Node n)
    (a) put node into priority queue, return false if priority queue is full beforehand, and true if successfully enqueued
    (b) increment the size of the queue and insert node n
    (c) sort the queue by frequency to correctly place node n with its correct priority

7. bool dequeue(PriorityQueue q, Node n)
    (a) dequeue node n from q, pass it back with a double pointer.
    (b) return false if priority queue is empty and true if successfully dequeued
    (c) remove the first element of the sorted array and pass it into n
    (d) shift every index in the array over by 1 to accommodate for the missing node at index 0

8. pqPrint(PriorityQueue q)
    (a) for each element in q, print it out for testing purposes
    (b) this function acts as a debug function

5 Codes

We will create a Code struct:

1. Code codeInit()
   (a) initialize code by setting variable top to 0 and zeroing out bits
   (b) NOTE: notice that this is not a struct with a pointer
2. uint32 codeSize(Code c)
   (a) return size of code. This is the number of bits pushed to Code c
3. bool codeEmpty(Code c)
   (a) return true if Code c is empty, false otherwise
4. bool codeFull(Code c)
   (a) return true if Code c is full, false otherwise
5. bool codeSetBit(Code c, uint32 i)
   (a) set Code c at index i to 1. If i is out of range return false. Return true if successful
   (b) use temporary values and bit logic to change a bit to a 1
   (c) NOTE: keep in mind that using mod 8 can get a certain bit location
6. bool codeClrBit(Code c, uint32 i)
   (a) clear Code c at index i by setting it to 0. If i is out of range return false. Return true if successful otherwise
   (b) NOTE: very similar to set bit, but it changes the bit to 0 instead of 1, think about bit logic
7. bool codeGetBit(Code c, uint32 i)
   (a) get bit in Code c at index i. If i is out of range or index i is equal to 0 return false. Return true only if index i is equal to 1
   (b) NOTE: possibly shift bits to the right in order to see what value is in the rightmost position (rightmost value = 1)
8. bool codePushBit(Code c, uint8 bit)

(a) push bit to Code c
(b) return false if Code is full before pushing bit
(c) return true if bit is successfully pushed
(d) if bit is 0 then you don't need to change values, just increment the top of the code stack by 1 to indicate that there is a 0 there
(e) otherwise set the desired bit to the top of the code and increment the top

9. bool codePopBit(Code c, uint8 bit)

   (a) pop bit from Code c by passing the value back into pointer bit.
   (b) return false if Code c is empty, and true if bit is successfully popped
   (c) clearing the top bit does not do anything, as anything above the top cannot be accessed anyways
   (d) make sure to decrement top to account for the removal of a bit

10. codePrint(Code c)

    (a) debug function that prints out elements in c
    (b) used to determine if push and pop are successful or not

6 I/O

We will need to make an IO file for reading and writing bytes

1. int readBytes(int infile, uint8 buf, int nbytes)

   (a) infile parameter is the number of bytes read from the file descriptor
   (b) loops calls to read() until nbytes are read from the file descriptor or there are no more bytes in the file to read (eof)
   (c) keep track of how many bytes you have read using read()
   (d) while you haven't read nbytes from infile, loop calls to read()
   (e) NOTE: when reading into the buffer, keep incrementing the buffer as calls to read() are looped, so buffer is not overwritten
   (f) NOTE: you do not want to read the same amount of bytes every call to read(), you will need to decrement it based on how many bytes you've read

(g) add the total amount of bytes read to the extern variable for bytes read

2. int writeBytes(int outfile, uint8 buf, int nbytes)

    (a) loop calls to write()
    (b) loop until nbytes are written, all bytes of buf are written, or no bytes are written
    (c) similar to read bytes, except with writing instead of reading
    (d) NOTE: add additional statement in the loop to break the loop when you have written the desired amount of bytes

3. bool readBit(int infile, uint8 bit)

    (a) read bytes into a buffer and read each bit from it one at a time
    (b) create a buffer of bytes to track which bit to return to the bit pointer.
    (c) buffer stores BLOCK number of bytes and returns false if there are no bits that can be read. Returns true if there are still bits to read.
    (d) the variables in this function should be initialized as static, because we will loop calls to this function and want to keep the same variable values
    (e) if we have read everything in the buffer, or we haven't started reading, then read a block of bytes to the buffer
    (f) we want to read a bit at a time from the buffer, so we can read the current bit and shift the bit that we have already read off the value in the buffer
    (g) keep track of where you currently are in the buffer and how many bits to shift off

4. writeCode(int outfile, Code c)

    (a) use bit buffer similar to readBit.
    (b) write each bit in Code c to buffer starting from 0th bit
    (c) when buffer has BLOCK bytes, write buffer contents to outfile
    (d) initialize static variables to manage the buffer outside of the function, it will need to be called for flush codes

(e) while we haven't written the amount of bits in the code, use bit logic to copy the Code c into a buffer

(f) when we have written out a block of bytes to the buffer, use writebytes() to write the buffer's contents to outfile

(g) NOTE: make sure to reset the buffer by clearing it and resetting the static index of the buffer

5. flushCodes(int outfile)

   (a) this function makes sure that there are no bits leftover after write-Code is called, so this writes out any left out bits in the buffer

   (b) this should be a relatively short function

   (c) if the buffer is not empty then write out however many bytes are in the buffer

7 Stacks

We need to make a stack of nodes to reconstruct the Huffman tree

1. stackCreate(uint32 capacity)

   (a) constructor for stack that sets max nodes to capacity

   (b) creates an array of node pointers similar to pq

2. stackDelete(Stack s)

   (a) destroys stack, sets pointer to NULL after freeing allocated memory from stack

3. bool stackEmpty(Stack s)

   (a) return true if s is empty, false if not

4. bool stackFull(Stack s)

   (a) return true if stack is full, false if not

5. uint32 stackSize(Stack s)

   (a) return number of nodes in s

6. bool stackPush(Stack s, Node n)

(a) push n onto s. Return false if stack is full beforehand and true if push is successful
(b) set the top of the stack to the given Node n

7. bool stackPop(Stack s, Node n)
   (a) pop n from s, and pass popped n back into double pointer n. Return false if stack was empty before pop and true if pop was successful
   (b) NOTE: top of the stack refers to 1 above the actual top of the stack
8. stackPrint(Stack s)
   (a) prints out the contents of s

8 Huffman code module

1. Node buildTree(uint64 hist(static ALPHABET))
   (a) builds node tree based on histogram given ALPHABET indices corresponding to different symbols
   (b) returns root node for the tree
   (c) to build the tree, we need to create a priority queue and create nodes based on our histogram
   (d) if an index in histogram has a frequency above 0, then we must create a node of it and enqueue it to our priority queue
   (e) after we have checked for every index of histogram, we have to dequeue 2 nodes at a time (left and right children) and join them together to make the parent node, and then push the parent node to the stack
   (f) we do this until there is only one parent node left in the queue, and we return this node as the root node of the tree
2. buildCodes(Node root, Code table(static ALPHABET))
   (a) makes a code table for each symbol in the Huffman tree, copies the codes to table, which has ALPHABET indices for each symbol
   (b) as long as the current root is not NULL, we can recursively check through the tree (root not null is the base case here)

(c) if the current node is a leaf node(no children) then set the table at that symbol equal to the current code

(d) otherwise, push a 0, recursively run function on the left child, then pop a bit

(e) afterwards, push a 1, recursively run function on the right child, then pop another bit

(f) NOTE: you cannot run codeinit() here because it does not work on compile time, use something else to initialize the code c outside the function.

3. dumpTree(int outfile, Node root)

    (a) writes to outfile L followed by byte of symbol for a leaf node and an I for interior nodes

    (b) do not write the symbol for interior nodes

    (c) this function will also be called recursively but at the beginning instead

    (d) if the root node is not null, recursively call this function on the left child, then the right child

    (e) if the root node is a leaf node, write an L followed by the symbol of the root

    (f) if the root node is an interior node, write an I without the symbol following it

4. Node rebuildTree(uint16 nbytes, uint8 treeDump(static nbytes))

    (a) reconstructs Huffman tree using treeDump

    (b) length of treeDump is nbytes. Returns root node of reconstructed tree

    (c) create a stack with the size of the treeDump

    (d) if an L is read, create a node for the symbol following it, then push that node to the stack

    (e) if an I is read, pop the stack twice, once for the right child and once for the left, and create a parent node of the two and push the parent node to the stack

    (f) once you have done this nbytes times, the last remaining node in the stack should be the root node, pop and return this node

5. deleteTree(Node root)
    (a) deconstructs Huffman tree
    (b) post-order traversal to free each node in the tree
    (c) set pointer root to NULL after freeing each node
    (d) this function can be called recursively to delete each child in the tree

9 File Encoder

Steps to implement the Huffman encoder

1. read infile to make a histogram using struct
2. increment index 0 and 255 of histogram array to make sure that there are at least 2 elements
3. buildTree() to make Huffman tree using histogram
    (a) make a priority queue and add each symbol that has a non 0 frequency to the queue and make a node of it
    (b) while there are two or more nodes in priority queue, dequeue two nodes, one left and one right
    (c) join these two dequeued nodes and queue the now created parent node
    (d) frequency of parent node = sum of children frequencies
    (e) when there is one node left in the queue, that is the root of the tree
4. make a code table by stepping through the Huffman tree created
    (a) create Code c using codeInit() starting at root of the tree
    (b) if current node is a leaf, c is the path to a node and has the node's symbol. Save the code to the table
    (c) if it is not a leaf, push 0 to Code c and go down the left link
    (d) once returning from the left, pop bits from c, push 1 to c and go down the right links
    (e) when returning from right, pop bit from c.

5. make a header supplied in header.h
6. write header to outfile
7. write Huffman tree to outfile using dumpTree
8. write corresponding symbol code to outfile by reading infile
9. flush remaining buffered bits using flushCodes
10. close both files

10 Decoder specifics

1. read header from infile and its magic number
2. if magic number does not match MAGIC in defines.h, then quit program
3. set outfile permissions using fchmod() with given permissions in the header
4. read dumped tree from infile and reconstruct Huffman tree (rebuildTree())
    (a) treeDump = array containing dumped tree
    (b) treeDump length = nbytes
    (c) loop over contents from 0 to nbytes in treeDump
    (d) if element of array is L (leaf), then the next element of the array is its symbol. Create a node using that symbol and push it to the stack
    (e) if element of array is I (interior node), then pop the stack to get the right child of that node, and pop again to get the left child of the node.
    (f) join the two popped nodes and join them to create a parent node and push it to the stack
    (g) once there is one node left in treeDump, that is the root of the Huffman tree
5. read infile for each bit using readBit()
    (a) begin at root of the tree, if a 0 is read then go left of current node. If a 1 is read then go right of the current node

(b) If a leaf node is encountered (next node is NULL) then write that symbol to outfile and reset the current node back to the root of the tree

(c) repeat for each read bit and until the decoded symbols equal those of the original file size from the header

6. close infile and outfile