Darren Lee
CSE13S

Assignment 6: Huffman Coding
Design Document

Purpose

The purpose of this assignment is to create a Huffman coding module with an encoder and a decoder using node, priority queue, code, and stack abstract data types. Huffman coding creates static encodings of information and allows for data compression. The optimal static encoding assigns the least number of bits to the most common symbol and the greatest number of bits to the least common symbol according to the entropy equation

$$H(X) = \sum_{i=1}^{n} \Pr[x_i] I(x_i) = -\sum_{i=1}^{n} \Pr[x_i] \log_2 \Pr[x_i].$$

for the set X with I(x) as the measure of the amount of information in the set.

Layout/Structure

- Encode.c-contains the implementation for the Huffman encoder
- Decode.c-contains the implementation for the Huffman decoder
- Entropy.c-provided by the resources repository and contains the code for measuring the entropy of a file
- defines.h - contains macro definitions for the assignment
- Header.h - contains the struct definition for a file header
- Node.h-provided by resource repository, contains the node ADT interface
- Node.c-contains the implementation for the node ADT
- pq.h -provided by resources repository, contains the priority queue ADT interface
- pq.c -contains the implementation of the priority queue ADT, priority queue struct is defined in this file
- Code.h- contains the code ADT interface
- Code.c-this file will contain your implementation of the code ADT
- Io.h- provided by resources repository. This file contains the I/O module interface.
- Io.c - contains the implementation for the I/O module
- stack.h - contains the stack ADT interface
- Stack.c -contains the implementation of the stack ADT, the stack struct is defined in this file
- Huffman.h-provided by the resources repository, contains the Huffman coding module interface
- Huffman.c-contains the implementation of the Huffman coding module interface
- Makefile-allows user to compile the programs

Node ADT

- The Huffman trees are made up of nodes.

- Each node has a pointer to its left child, a pointer to its right child, a symbol and a frequency of that symbol. The node's frequency is used by the decoder.
- A symbol is a uint8_t and the input is read as raw bytes. The definition of a node is made transparent.

Priority Queue ADT

- High priority elements are dequeued before low priority elements.
- Enqueue adds elements to a position based on priority.
- The priority queue will be implemented using an insertion sort to enqueue nodes.
- When nodes have matching frequencies, the node with a smaller ASCII symbol will be dequeued first.
- A lower frequency node has a higher priority.

Code ADT

- Represents a stack of bits
- The struct definition of a code is transparent so we can pass a struct by value
- The MAX_CODE_SIZE macro is defined in defines.h

I/O

- Uses low level syscalls like read(), write(), open(), and close()

Stacks

- The stack will be used to store nodes.

Functionality

Encoder

- The Huffman encoder reads from an input file, gets the Huffman encoding of the contents, and compresses the file using the encoding. It supports the command line options [-h] [-i infile] [-o outfile] [-v].
    - -h option prints out a help message the explains the purpose of the program and the command line options the encoder accepts then exits the program
    - -i infile option specifies the input file to encode with Huffman coding. The default input is stdin
    - -o outfile option specifies the output file to write the compressed input to. The default output is stdout
    - -v option specifies to print the compression statistics to stderr. The statistics are the uncompressed file size, the compressed file size and the space saving (100 * (1-(compressed size/uncompressed size))
- The encoder function process::
    - Counts the number of occurrences of each unique symbol in the file and computers a histogram
    - The counts for symbols 0 and 255 are both incremented once. Their priorities are increased in the priority queue to account for the artificially increased count.
    - Constructs a Huffman tree using the histogram with a priority queue
    - Creates a code table. The indices in the table represent symbols and the value at each index represents the symbol's code. This step uses a stack of bits to perform a traversal of the Huffman tree.

    - Puts an encoding of the Huffman tree called a tree dump in the file using a post-order traversal of the Huffman tree
    - Go through each symbol in the input file and emit each symbol's code to the output file
- If the infile is stdin, the file is not seekable, so a temporary file is created to copy stdin to. Then that temporary file is used as the infile.

Decoder

- The Huffman decoder reads a compressed input file and decompresses it back to its original, uncompressed size. It supports the command line options [-h] [-i infile] [-o outfile] [-v].
    - -h option tells the program to print out a help message describing the purpose of the decoder and the available command-line options then exits the program
    - -i infile specifies the input file to decode using Huffman coding. The default input is stdin
    - -o outfile specifies the output file to write the decompressed input to. The default output is stdout
    - -v prints decompression statistics to stderr. The statistics are compressed file size, the decompressed file size, and space saving (100 * (1 - (compressed size/decompressed size))
- Process for decode/decompress:
    - Reads the tree dump from the input file, using a stack of nodes to reconstruct the Huffman tree
    - Reads in the rest of the file bit-by-bit, traversing down the Huffman tree. A read of 0 leads to walking down the left link while a read of 1 leads to walking down the right link. When a leaf node is reached, its symbol is emitted and the program traverses again from the root.
- If the infile is stdin, a temporary file is created to copy stdin to. Then that temporary file is used as the infile.

Error handling

- If an invalid command line option is entered into the encoder or decoder, the program will print a help and usage message and exit.
- If the infile fails to open for the encoder or decoder, the program will print an error message and exit.
- If the outfile fails to open for the encoder or decoder, the program will print an error message and exit.
- If the decoder is unable to read the header, the program prints an error message and exits.
- If the magic number for the header does not match MAGIC defined in defines.h, the decoder will print an error message and exit.
- If the tree size in the header is greater than the MAX_TREE_SIZE of 3*ALPHABET - 1 defined in defines.h, the decoder will print an error message and exit.

Pseudocode

node.c

```
Node *node_create(int symbol, int frequency):
        Node *n=(Node *)malloc(sizeof(Node))
        If (n):
                n->symbol=symbol
                n->frequency=frequency
                n->left=null
                n->right =null
                Return n
        Else:
                free(n)
                Return null

Void node_delete(Node **n):
        free(*n)
        *n=null

Node *node_join(Node *left, Node *right):
        Node *n=node_create('$',left->frequency+right->frequency)
        n->left=left
        n->right=right
        Return n

Void node_print(Node *n):
        printf(symbol:<n->symbol>)
        Print new line
        Printf(frequency:<n->frequency>)
        Print new line
```

pq.c

```
Struct PriorityQueue {
        Head
        Tail
        Slot
        Size
        Capacity
        Node **elements
}

U32 get_left(PriorityQueue *q, u32 i):
        Return (i-1+q->capacity)%q->capacity

U32 get_right(PriorityQueue *q,u32 i):
        Return (i+1)%q->capacity
```

```
PriorityQueue *pq_create(int capacity)
        PriorityQueue *q=(PriorityQueue *)malloc(sizeof(PriorityQueue))
        If (q):
                q->capacity=capacity
                q->head=0
                q->tail=0
                q->slot=0
                q->size=0
                q->elements=(Node **)calloc(capacity,sizeof(Node *))
                if(!q->elements):
                        free(q)
                        q=null
        Return q

Void pq_delete(PriorityQueue **q):
        free((*q)->elements)
        free(*q)
        *q=null

Bool pq_empty(PriorityQueue *q):
        Return !q->size

Bool pq_full(PriorityQueue *q):
        Return q->size==q->capacity

Int pq_size(PriorityQueue *q):
        Return q->size

Bool enqueue(PriorityQueue *q, Node *n):
        If pq_full(q):
                Return false
        If (n->symbol==0 or n->symbol==255):
                n->frequency--
        q->slot=q->tail
        while(q->slot!=q->head and the element to the left of slot has a greater frequency than
the element at slot or the frequencies are equal and the element to the left has a greater symbol
than the element at slot):
                q->elements[q->slot]=q->elements[get_left(q,q->slot))]
                q->slot=get_left(q,q->slot)
                ->elements[q->slot]=n
        q->elements[q->tail]=get_right(q,q->tail)
        q->size++
        Return true
```

```
Bool dequeue(PriorityQueue *q, Node **n):
        If pq_empty(q):
                Return false
        *n=q->elements[q->head]
        q->head=get_right(q,q->head)
        q->size--
        Return true

Void pq_print(PriorityQueue *q):
        If (!pq_full(q)):
                For i in range q->head to q->tail:
                        node_print(q->elements[i])
        Else:
                For i in range q->head to get_left(q,q->tail):
                        node_print(q->elements[i])
```

code.c

```
Define BYTE_SIZE 8

Code code_init(void):
        Code *c
        c->top=0
        For i in range MAX_CODE_SIZE:
                c->bits[i]=0
        Return c

Int code_size(Code *c):
        Return c->top

Bool code_empty(Code *c):
        Return !c->top

Bool code_full(Code *c):
        Return c->top==ALPHABET

Bool code_push_bit(Code *c, int bit):
        If code_full(c):
                Return false
        c->bits[c->top/BYTE_SIZE] |= (bit <<(c->top % BYTE_SIZE)
        c->top++
        Return true

Bool code_pop_bit(Code *c,int *bit):
```

```
        If code_empty(c):
                Return false
        c->top--
        *bit=(c->bits[c->top/BYTE_SIZE] & (1 << (c->top % BYTE_SIZE)))>>(c->top %
BYTE_SIZE)
        Return true

Void code_print(Code *c):
        For i in range c->top:
                Print <c->bits[i/BYTE_SIZE] & (1<<(i % BYTE_SIZE)))>>(i%BYTE_SIZE)]>
        Print new line
```

io.c

```
Define BYTE_SIZE 8

Static u8 buffer[BLOCK]= { 0 }
Static u32 bufindex

U8 code_get_bit(Code *c, u32 i ):
        Return (c->bits[i/BYTE_SIZE]&((u64) 1<<(i%BYTE_SIZE)))>>(i%BYTE_SIZE)

U8 buf_get_bit(u8 *buf, u32 i):
        Return (buf[i/BYTE_SIZE] & ((u64) 1<<(i%BYTE_SIZE)))>>(i%BYTE_SIZE)

Void buf_set_bit(u8 *buf, u32 i):
        buf[i/BYTE_SIZE] |= ((u64) 1<<(i%BYTE_SIZE)

Void buf_clr_bit(u8 *buf,u32 i):
        buf[i/BYTE_SIZE] |= ((u64) 1 << (i%BYTE_SIZE))

Int read_bytes(int infile, int *buf, int nbytes):
        Int bytes=0
        Int total=0
        While (total!=nbytes):
                bytes=read(infile,bu,nbytes-total)
                If (!bytes):
                        Break
                total+=bytes
        bytes_read+=total
        Return total

Int write_bytes(int outfile, int *buf, int nbytes):
        Int bytes=0
        Int total=0
```

```
        while(total!=nbytes):
                bytes=write(outfile,buf,nbytes-total)
                if(!bytes):
                        Break
                total+=bytes
        bytes_written+=total
        Return total

Influenced by Eugene's lab section on 5/11
Bool read_bit(int infile, int *bit):
        U32 last_bit=0
        U32 read=0
        If (!bufindex)::
                read=read_bytes(infile,buffer,BLOCK)
                if(read<BLOCK):
                        last_bit=read*BYTE_SIZE+1
        *bit=buf_get_bit(buffer,bufindex)
        bufindex++
        If (bufindex==BLOCK*BYTE_SIZE):
                bufindex=0
        If (bufindex==last_bit):
                Return false
        else:
                Return true

Inspired by Eugene's lab section on 5/11
Void write_code(int outfile, Code *c)
        For i in range code_size(c):
                If (code_get_bit(c,i)):
                        buf_set_bit(buffer, bufindex)
                Else:
                        buf_clr_bit((buffer, bufindex)
        Bufindex++
        If bufindex==BLOCK*BYTE_SIZE:
                write_bytes(outfile,buffer,BLOCK)
                bufindex=0

Void flush_codes(int outfile):
        If bufindex>0:
                write_bytes(outfile,buffer, bufindex/BYTE_SIZE+(bufindex%BYTE_SIZE then 1
else 0)
```

stack.c

Struct code provided by Professor Long in assignment pdf

```
Struct Stack {
        Int top
        Int capacity
        Node **items
}

Stack *stack_create(int capacity):
        Stack *s=(Stack *s) malloc(sizeof(Stack))
        If (s):
                s->top=0
                s->capacity=capacity
                s->items=(Node **)calloc(capacity,sizeof(Node *))
                If (!s->items):
                        free(s)
                        s=null
        Return s

Void stack_delete(Stack **s):
        if(*s and (*s)->items):
                free((*s)->items)
                free(*s)
                *s=null

Bool Stack_empty(Stack *s):
        Return !s->top

Bool stack_full(Stack *s):
        Return s->top==s->capacity

Int stack_size(Stack *s):
        Return s->top

Bool stack_push(Stack *s, Node *n):
        if(stack_full(s)):
                Return false
        s->items[s->top]=n
        s->top++
        Return true

stack_pop(Stack *s, Node **n):
        if(stack_empty(s)):
                Return false
        s->top--
        *n=s->items[s->top]
```

```
        Return true

stack_print(Stack *s):
        For i in range s->top:
                node_print(s->items[i])
```

huffman.c

```
Code inspired by Eugene's 5/11 lab section
postorder(Node *n, Code table[static ALPHABET], Code c):
        U8 bit
        If (n):
                If n is a leaf:
                        table[n->symbol]=c
                code_push_bit(&c,0)
                postorder(n->left)
                code_pop_bit(&c, &bit)
                code_push_bit(&c,1)
                postorder(n->right)
                code_pop_bit(&c,&bit)

Node *build_tree(int hist[static ALPHABET]):
        PriorityQueue *q=pq_create(ALPHABET)
        Node *left
        Node *right
        Node *root
        For i in range ALPHABET:
                if (hist[i] > 0):
                        enqueue(q, node_create(i,hist[i])
        While (pq_size>1):
                dequeue to get left child
                Dequeue to get right child
                Enqueue node_join(left child, right child)
        Dequeue last node into root
        Delete q
        Return root

Void build_codes(Node *root, Code table[static ALPHABET]):
        Code c=code_init()
        postorder(root,table,c)

Node *rebuild_tree(int bytes, int tree_dump[static nbytes]):
        Stack *s=stack_create(ALPHABET)
        Node *left
```

```
        Node *right
        Node *root
        For i in range nbytes:
                If (tree_dump[i]==’L’):
                        Push node_create(tree_dump[i+1],0) onto the stack
                        I++
                Else if (tree_dump[i]==’I’):
                        Pop to get the right child
                        Pop to get the left child
                        Push node_join(left,right) onto the stack
        Pop the last node of the stack into root
        Delete s
        Return root

Void delete_tree(Node **root):
        If (*root):
                delete_tree(root->left)
                delete_tree(root->right)
                node_delete(root)
```

encode.c

```
Define OPTIONS hi:o:v

Static u8 buffer[BLOCK]
U64 bytes_written, bytes_read

postorder_tree(Node *n, int outfile):
        If node is not null:
                Postorder_tree left child
                Postorder_tree right child
                If node is a leaf
                        Write L
                        Write n’s symbol
                Else
                        Write I

Int main():
        Struct stat statbuf
        Set infile to stdin
        Set outfile to stdout
        While getopt is not -1
                switch(opt):
                Case h:
                        Print help message
```

```
                Exit program
        Case i:
                Specify infile
                Break
        Case o:
                Specify outfile
                break
        Case v:
                V flag=1
                Break
If infile is -1
        Print error message
        Exit program
If outfile is -1
        Print error message
        Exit program
If infile is stdin
        Seek flag =0
        Create tempfile
        While read(infile) is not end of file
                Write to tempfile
                Decrease bytes_written by how many bytes written
        Set permissions for temp file
        infile=tempfile
        Seek beginning of infile
Create histogram
Get stats of infile
Set permissions of outfile
Increment histogram index 0
Increment histogram index 255
While read(infile) is not end of file
        Histogram of symbol ++
Create code table
Build tree
Build codes
Create header
Count unique symbols
Set header magic
Set header tree size
Set header file size
Write header to outfile
postorder_tree(root,outfile)
Seek beginning of infile
While read(infile) is not end of file
```

Write code of symbol to outfile
Flush codes
If (v flag is 1):
Print statistics
Delete tree
If file is not seekable:
Delete tempfile
Close infile
Close outfile
Return 0

decode.c
Define BYTE_SIZE 8
Define OPTIONS hi:o:v

Static u8 buffer[BLOCK]
U64 bytes_written, bytes_read

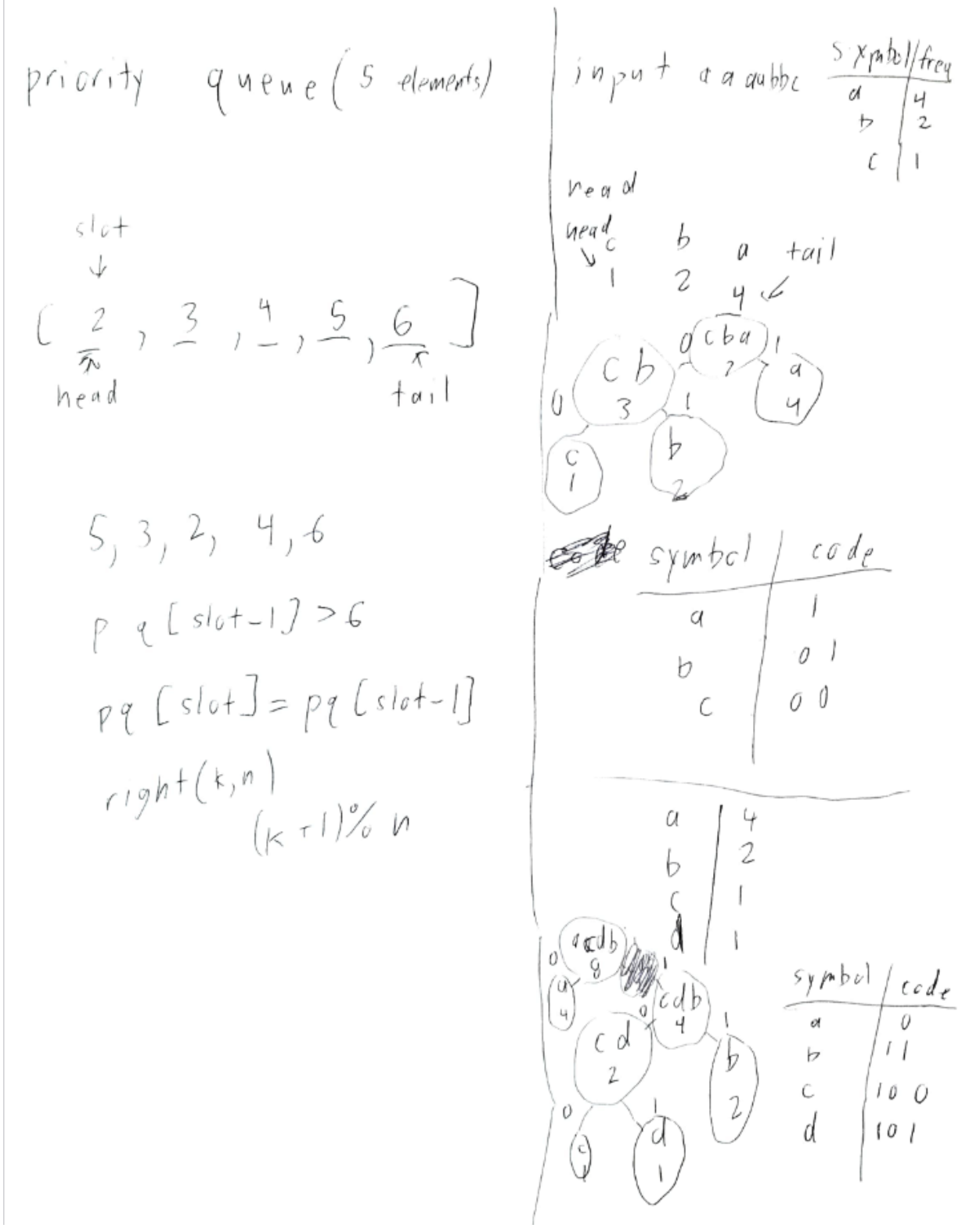
walk_tree(Node walk, node root, int outfile, int infile):
U8 bit
If (walk is not null):
Write walk's symbol to outfile
walk=root
Increment decoded
Read_bit from infile
If bit is 0:
walk=walk's left child
Else:
walk =walk's right child

Int main():
Struct stat statbuf
Set infile to stdin
Set outfile to stdout
While getopt is not -1
switch(opt):
Case h:
Print help message
Exit program
Case i:
Specify infile
Break
Case o:
Specify outfile

```
                break
        Case v:
                V flag=1
                Break
If infile is -1
        Print error message
        Exit program
If outfile is -1
        Print error message
        Exit program
If infile is stdin
        Temp flag =1
        Create tempfile
        While read(infile) is not end of file
                Write to tempfile
                Decrease bytes_written by how many bytes written
        Set permissions for temp file
        infile=tempfile
        Seek beginning of infile
Create header
Read infile's header
If program cannot read header because the size is wrong:
        Print error message
        Exit program
If magic number is invalid:
        Print error message
        Exit program
Set permissions for outfile
If tree_size>MAX_TREE_SIZE:
        Print error message
        Exit program
Create tree dump of size tree_size
Read tree_size bytes from infile
For i in range tree_size:
        Create the tree dump with values read
root=rebuild_tree
walk=root
While (decoded!=file_size):
        walk_tree(walk,root,outfile,infile)
If (v flag is 1):
        if (temp flag is 1):
                bytes_read/=2
        Print statistics
Delete tree
```

```
If (temp flag is 1):
        Delete tempfile
Close infile
Close outfile
Return 0
```

Draft Work


priority queue (5 elements)
slot
[2 , 3 , 4 , 5 , 6]
head
tail
5, 3, 2, 4, 6
pq[slot-1] > 6
pq[slot] = pq[slot-1]
right(k, n)
(k+1)% n
input a a a a b b c
Symbol | freq
a | 4
b | 2
c | 1
symbol | code
a | 1
b | 0 1
c | 0 0
a | 4
b | 2
c | 1
d | 1
symbol | code
a | 0
b | 1 1
c | 1 0 0
d | 1 0 1

input: a a a a b b c d

a a a a b b c d
0 0 0 0 11 11 100 101

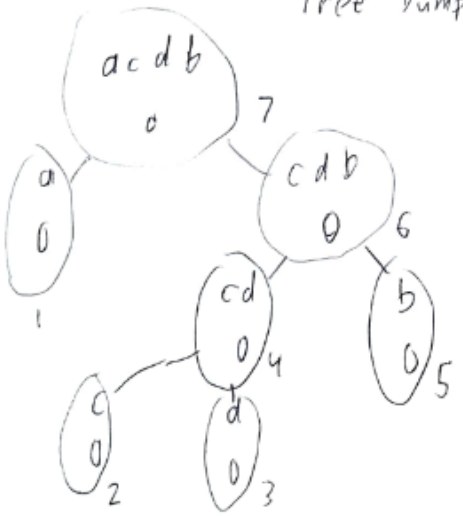
if both children are NULL, then it's a leaf

Postorder traversal

```
def postorder(Node *n):
    if n is not null:
        postorder(n->left)
        postorder(n->right)
        node_print(n)
}
```

9, 10, 5, 7, 8, 6, 3

Tree Dump



LaLcLd I Lb II