Shweta Jones
CSE 13S - Winter 2021
Prof Darrell Long

# Assignment 6 Design - Huffman Coding

**Description:**

This programming assignment focuses on data compression using different ADTs including nodes, priority queues, code and stack ADTs. THe ecode.c function compresses the file and the decode.c file decompresses the file and should essentially work hand in hand, meaning the input of the encode should match the decode output. The program takes 3 command line options: h, i, o. The h options points the help information, i helps to specify the input file and o specifies the output file.

**Files:**

encode.c: Has huffman encoder
decode.c: Has huffman decoder
node.c: Has node ADT
pq.c: Has priority queue ADT
code.c: Has code ADT
stack.c: Has stack ADT
huffman.c: Has huffman coding module interface

**Initial Design:**

**encoder.c:**

-h : Prints out a help message
-i infile : Specify the input file path containing data
-o outfile : Specify the output file path to write the encoded data
-v: Prints the stats of the encodes result
While fgetc != EOF
    Reading the input file line by line
    Huffman encoding

**void helpInfo(void) - EXTRA FUNCTION**
    Prints help information

**decoder.c:**

-h : Prints out a help message
-i infile : Specify the input file path containing data
-o outfile : Specify the output file path to write the encoded data
-v: Prints the stats of the encodes result
While fgetc != EOF

Reading the input file line by line
Huffman decoding

**void helpInfo(void) - EXTRA FUNCTION**
Prints help information

**node.c**

```
struct Node {
   char *oldspeak
   char *newspeak
   Node *next
   Node *prev
}
```

Node Create

```
Node *node_create(char *oldspeak, char *newspeak)
   Node *n = (Node *) malloc(sizeof(Node));
   n->oldspeak = oldspeak ? Strdup(oldspeak) : NULL
   n->newspeak = newspeak ? Strdup(newspeak) : NULL
   n->next = NULL
   return n
```

Node Delete

```
void node_delete(Node **n)
   free((*n)->oldspeak)
   free((*n)->newspeak)
   free(*n)
   *n = NULL
```

Node Join

```
Node *node_join(Node *left, Node *right)
```

Node Print

```
void node_print(Node *n)
   if (n->oldspeak && n->newspeak)
      printf("%s -> %s\n", n->oldspeak, n->newspeak)
   else if (n->oldspeak)
      printf("%s\n", n->oldspeak)
```

**Pq.c:**

PriorityQueue *pq_create(uint32_t capacity)
  Creates empty priority queue given capacity
  Each value is 0

void pq_delete(PriorityQueue **q)
  Clears each value in the queue
  Removes the priority queue

bool pq_empty(PriorityQueue *q)
  if (pq->size == 0)
    return true
  else
    return false

bool pq_full(PriorityQueue *q)
  if (pq->size == pq->capacity)
    return true
  else
    return false

uint32_t pq_size(PriorityQueue *q)
  return q->size;

bool enqueue(PriorityQueue *q, Node *n)
  Allows for values to be added into the priority queue given node
  Returns true once done

bool dequeue(PriorityQueue *q, Node **n)
  Removes the node from the priority queue
  Returns true once this is done

void pq_print(PriorityQueue *q)
  Prints all the nodes in the priority queue

**Code.c:**

Code code_init(void) {
  Creates the code
  Nothing is returned
}

```
uint32_t code_size(Code *c)
   return c->top;

bool code_empty(Code *c)
   if (c->top == 0)
      return true
   else
      return false

bool code_full(Code *c)
   if (c->top > 0)
      return true
   else
      return false

bool code_push_bit(Code *c, uint8_t bit)
   c->top = bit
   return true

bool code_pop_bit(Code *c, uint8_t *bit)
   c->top = c->bit
   return true

void code_print(Code *c)
   Prints the entire code structure
```

**stack.c:**

```
struct Stack {
   uint32_t top
   uint32_t capacity
   int64_t *items
}
```

Stack Create

```
Stack *stack_create(uint32_t capacity)
   Stack *s = (Stack *) malloc (sizeof(Stack))
   if (s)
      s->top = 0
      s->capacity = capacity
      s->items = (int64_t *) calloc (capacity, sizeof(int64_t))
```

```
    if (!s->items)
        free(s)
        s = NULL
  return s
```

Stack Delete

```
void stack_delete(Stack **s)
  if (*s && (*s)->items)
    free((*s)->items)
      free(*s)
      *s = NULL
  return
```

Stack Empty

```
bool stack_empty(Stack *s)
  return s->top == 0
```

Stack Full

```
bool stack_full(Stack *s)
  return s->top == 1
```

Stack Size

```
uint32_t stack_size(Stack *s)
  return s->top
```

Stack Push

```
bool stack_push(Stack *s, uint32_t x)
  if (s->top == s->capacity)
    s->capacity = 2 * s-> capacity
      s->items = (int64_t *) realloc (s->items, s->capacity * sizeof(int64_t))
      if (s->items == NULL)
        return false
  s->items[s->top] = x
  s->top += 1
  return true
```

Stack Pop

```
bool stack_pop(Stack *s, uint32_t *x)
  if (s->top == 0)
    return false
```

```
    s->top -= 1
    *x = s->items[s->top]
    return true
```

Stack Print

```
void stack_print(Stack *s, FILE *outfile, char *cities[])
    for (uint32_t p = 0; p<s->top; p++)
        printf("%ld ", s->items[s->top])
    printf("\n")
```