|  |
| --- |
| // Opening input file in read-only mode  int fd1 = open(“sample.txt”, O\_RDONLY);  if (fd1 == -1) {      perror("Open Failed For Input File:\n");      exit(1);  }    // Creating output file in write mode  int fd2 = open(“sample - compressed.txt”,                 O\_WRONLY | O\_CREAT, S\_IRUSR | S\_IWUSR);  if (fd2 == -1) {      perror("Open Failed For Output File:\n");      exit(1);  } |

Function to Initialize and Create Min Heap:

|  |
| --- |
| // Structure for tree nodes  struct Node {      char character;      int freq;      struct Node \*l, \*r;  };    // Structure for min heap  struct Min\_Heap {      int size;      struct Node\*\* array;  };    // Function to create min heap  struct Min\_Heap\* createAndBuildMin\_Heap(char arr[],                                          int freq[],                                          int unique\_size)  {      int i;        // Initializing heap      struct Min\_Heap\* Min\_Heap          = (struct Min\_Heap\*)malloc(sizeof(struct Min\_Heap));      Min\_Heap->size = unique\_size;      Min\_Heap->array = (struct Node\*\*)malloc(          Min\_Heap->size \* sizeof(struct Node\*));        // Initializing the array of pointers in minheap.      // Pointers pointing to new nodes of character      // and their frequency      for (i = 0; i < unique\_size; ++i) {            // newNode is a function          // to initialize new node          Min\_Heap->array[i] = newNode(arr[i], freq[i]);      }        int n = Min\_Heap->size - 1;      for (i = (n - 1) / 2; i >= 0; --i) {            // Standard function for Heap creation          Heapify(Min\_Heap, i);      }        return Min\_Heap;  } |

Function to Build and Create a Huffman Tree:

* C

|  |
| --- |
| // Function to build Huffman Tree  struct Node\* buildHuffmanTree(char arr[], int freq[],                                int unique\_size)  {      struct Node \*l, \*r, \*top;      while (!isSizeOne(Min\_Heap)) {          l = extractMinFromMin\_Heap(Min\_Heap);          r = extractMinFromMin\_Heap(Min\_Heap);          top = newNode('$', l->freq + r->freq);          top->l = l;          top->r = r;          insertIntoMin\_Heap(Min\_Heap, top);      }      return extractMinFromMin\_Heap(Min\_Heap);  } |

Recursive Function to Print Binary Codes into Compressed File:

* C

|  |
| --- |
| // Structure to store codes in compressed file  typedef struct code {      char k;      int l;      int code\_arr[16];      struct code\* p;  } code;    // Function to print codes into file  void printCodesIntoFile(int fd2, struct Node\* root,                          int t[], int top = 0)  {      int i;      if (root->l) {          t[top] = 0;          printCodesIntoFile(fd2, root->l, t, top + 1);      }        if (root->r) {          t[top] = 1;          printCodesIntoFile(fd2, root->r, t, top + 1);      }        if (isLeaf(root)) {          data = (code\*)malloc(sizeof(code));          tree = (Tree\*)malloc(sizeof(Tree));          data->p = NULL;          data->k = root->character;          tree->g = root->character;          write(fd2, &tree->g, sizeof(char));          for (i = 0; i < top; i++) {              data->code\_arr[i] = t[i];          }          tree->len = top;          write(fd2, &tree->len, sizeof(int));          tree->dec              = convertBinaryToDecimal(data->code\_arr, top);          write(fd2, &tree->dec, sizeof(int));          data->l = top;          data->p = NULL;          if (k == 0) {              front = rear = data;              k++;          }          else {              rear->p = data;              rear = rear->p;          }      }  } |

Function to Compress the File by Substituting Characters with their Huffman Codes:

* C

|  |
| --- |
| // Function to compress file  void compressFile(int fd1, int fd2, unsigned char a)  {      char n;      int h = 0, i;        // Codes are written into file in bit by bit format      while (read(fd1, &n, sizeof(char)) != 0) {          rear = front;          while (rear->k != n && rear->p != NULL) {              rear = rear->p;          }          if (rear->k == n) {              for (i = 0; i < rear->l; i++) {                  if (h < 7) {                      if (rear->code\_arr[i] == 1) {                          a++;                          a = a << 1;                          h++;                      }                      else if (rear->code\_arr[i] == 0) {                          a = a << 1;                          h++;                      }                  }                  else if (h == 7) {                      if (rear->code\_arr[i] == 1) {                          a++;                          h = 0;                      }                      else {                          h = 0;                      }                      write(fd2, &a, sizeof(char));                      a = 0;                  }              }          }      }      for (i = 0; i < 7 - h; i++) {          a = a << 1;      }      write(fd2, &a, sizeof(char));  } |

Function to Build Huffman Tree from Data Extracted from Compressed File:

* C

|  |
| --- |
| typedef struct Tree {      char g;      int len;      int dec;      struct Tree\* f;      struct Tree\* r;  } Tree;    // Function to extract Huffman codes  // from a compressed file  void ExtractCodesFromFile(int fd1)  {      read(fd1, &t->g, sizeof(char));      read(fd1, &t->len, sizeof(int));      read(fd1, &t->dec, sizeof(int));  }    // Function to rebuild the Huffman tree  void ReBuildHuffmanTree(int fd1, int size)  {      int i = 0, j, k;      tree = (Tree\*)malloc(sizeof(Tree));      tree\_temp = tree;      tree->f = NULL;      tree->r = NULL;      t = (Tree\*)malloc(sizeof(Tree));      t->f = NULL;      t->r = NULL;      for (k = 0; k < size; k++) {          tree\_temp = tree;          ExtractCodesFromFile(fd1);          int bin[MAX], bin\_con[MAX];          for (i = 0; i < MAX; i++) {              bin[i] = bin\_con[i] = 0;          }          convertDecimalToBinary(bin, t->dec, t->len);          for (i = 0; i < t->len; i++) {              bin\_con[i] = bin[i];          }            for (j = 0; j < t->len; j++) {              if (bin\_con[j] == 0) {                  if (tree\_temp->f == NULL) {                      tree\_temp->f                          = (Tree\*)malloc(sizeof(Tree));                  }                  tree\_temp = tree\_temp->f;              }              else if (bin\_con[j] == 1) {                  if (tree\_temp->r == NULL) {                      tree\_temp->r                          = (Tree\*)malloc(sizeof(Tree));                  }                  tree\_temp = tree\_temp->r;              }          }          tree\_temp->g = t->g;          tree\_temp->len = t->len;          tree\_temp->dec = t->dec;          tree\_temp->f = NULL;          tree\_temp->r = NULL;          tree\_temp = tree;      }  } |

Function to Decompress the Compressed File:

* C

|  |
| --- |
| void decompressFile(int fd1, int fd2, int f)  {      int inp[8], i, k = 0;      unsigned char p;      read(fd1, &p, sizeof(char));      convertDecimalToBinary(inp, p, 8);      tree\_temp = tree;      for (i = 0; i < 8 && k < f; i++) {          if (!isroot(tree\_temp)) {              if (i != 7) {                  if (inp[i] == 0) {                      tree\_temp = tree\_temp->f;                  }                  if (inp[i] == 1) {                      tree\_temp = tree\_temp->r;                  }              }              else {                  if (inp[i] == 0) {                      tree\_temp = tree\_temp->f;                  }                  if (inp[i] == 1) {                      tree\_temp = tree\_temp->r;                  }                  if (read(fd1, &p, sizeof(char)) != 0) {                      convertDecimalToBinary(inp, p, 8);                      i = -1;                  }                  else {                      break;                  }              }          }          else {              k++;              write(fd2, &tree\_temp->g, sizeof(char));              tree\_temp = tree;              i--;          }      }  } |