1. a)

```
#include <stdio.h>
#include <stdlib.h>

/* Descriere logica:
 * Alogritmul parcurge vectorul in range-ul dat si pentru fiecare element
 * dina cel range, cauta elementul minim de la membrul respectiv la final.
 * Dupa terminarea parcurgerii, se interschimba primul element cu minimul gasit.
*/

// Functie comparator care compara a cu b
int cmp(int *a, int *b)  {
   return *a < *b;
}

// Functie auxiliara de interschimbare a elementelor
void swap(int *x, int *y) {
   // Iau un auxiliar
   int aux = *x;
   // AL doilea se duce in primul
   *x = *y;
   // Primul se duce in al doilea
   *y = aux;
}
```

```
void selectionSort(int nodes[], int first, int last, int (*cmp)(int *a, int *b))
{
   int i, j, min1;

   // Parucrg range-ul ce trebuie sortat de la first la last - 2
   for (i = first; i < last - 1; i++) {
      // Gasesc pozitia elemtuiului minim din [i, last - 1]
      min1 = i;
      // Parcurg de la i pana la finalul range-ului
      for (j = i + 1; j < last; j++) {
         // daca elementul e mai mic decat minimul gasit pana in acest moment
         if (cmp(&nodes[j], &nodes[min1])) {
            // actualizez noul minim
            min1 = j;
         }
      }

      // Interschimb minimul gasit cu primul element
      swap(&nodes[min1], &nodes[i]);
   }
}

int main() {
   // imi declar un vector de 10 elemente
   int n = 10;
```

```
    int nodes[10] = {10, 8, 1, 4, 2, 3, 7, 5, 6, 11};

    // il sortez
    selectionSort(nodes, 0, n, cmp);

    // il afisez
    for (int i = 0; i < n; i++) {
        printf("%d ", nodes[i]);
    }

    printf("\n");
}
```

Primul pas

10, 8, 1, 4, 2, 3, 7, 5, 6, 11

$i = 0$

$min = 10 = v[0]$ } la început

Caut minimul de la 0 ... 9

minimul este 1

=> interschimb 1 cu 10

avem : 1, 8, 10, 4, 2, 3, 7, 5, 6, 11

Pasul 2 :

$i = 1$

$min = 8 = v[1]$ } la început

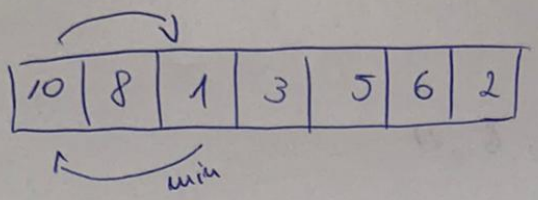
Caut minimul în 1, ..., 9 : $min = 2$

=> interschimb 2 cu 8

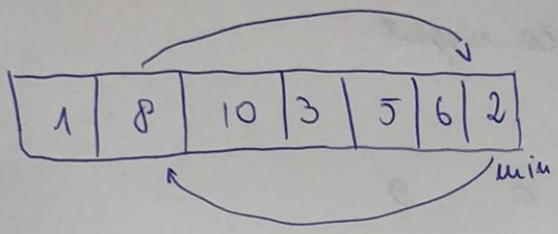
avem : 1, 2, 10, 4, 8, 3, 7, 5, 6, 11

La final vectorul a sortat crescător

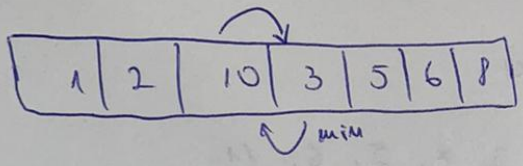
1)



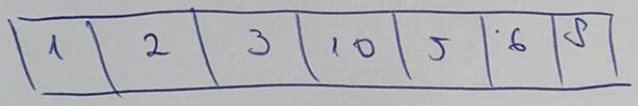
2)

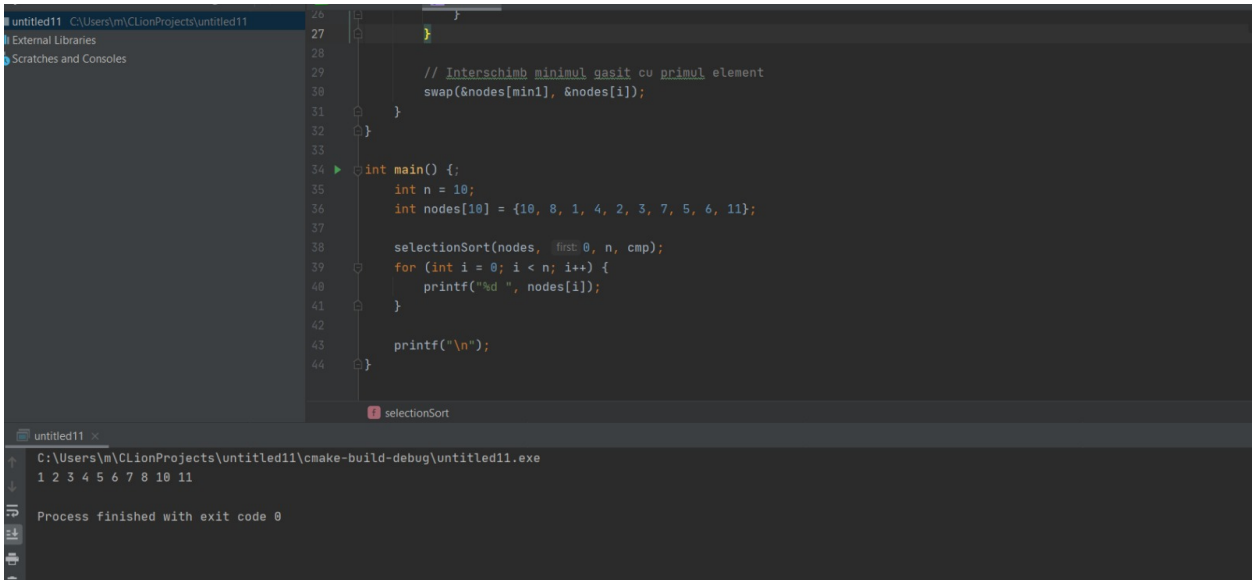


3)



4)





1.b)

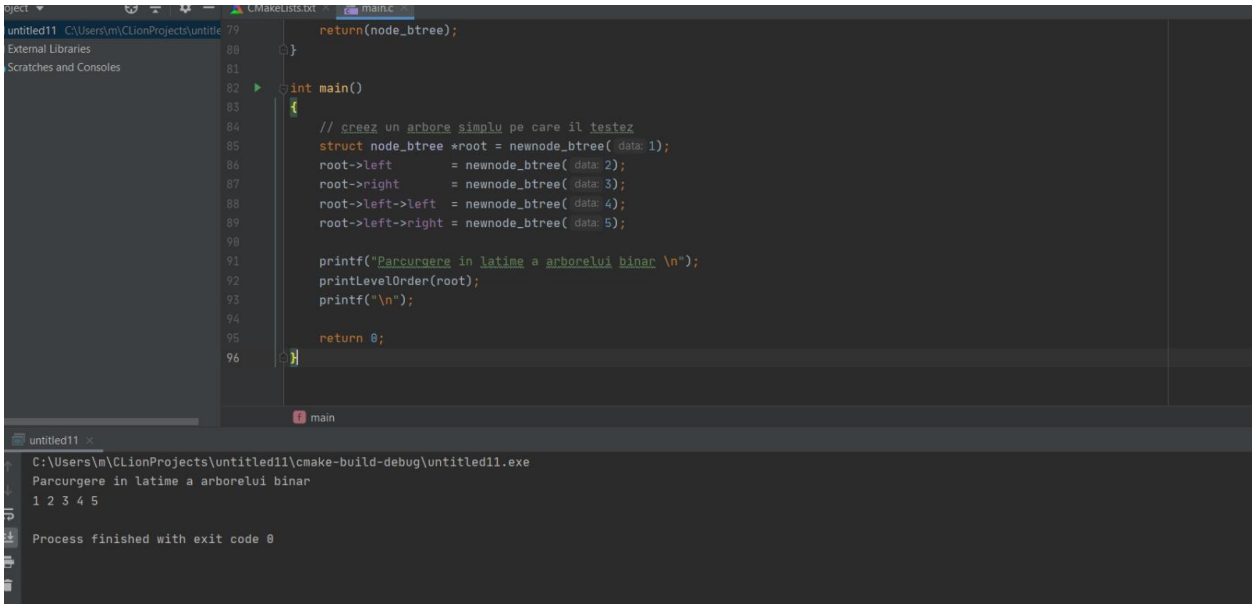
```
#include <stdio.h>

#include <stdlib.h>



// Structura node a arborelui binar

// Contine informatia si cate un pointer catre nodul stanga, respectiv nodul dreapta

struct node_btree

{

   int data;

   struct node_btree* left, *right;

};
```

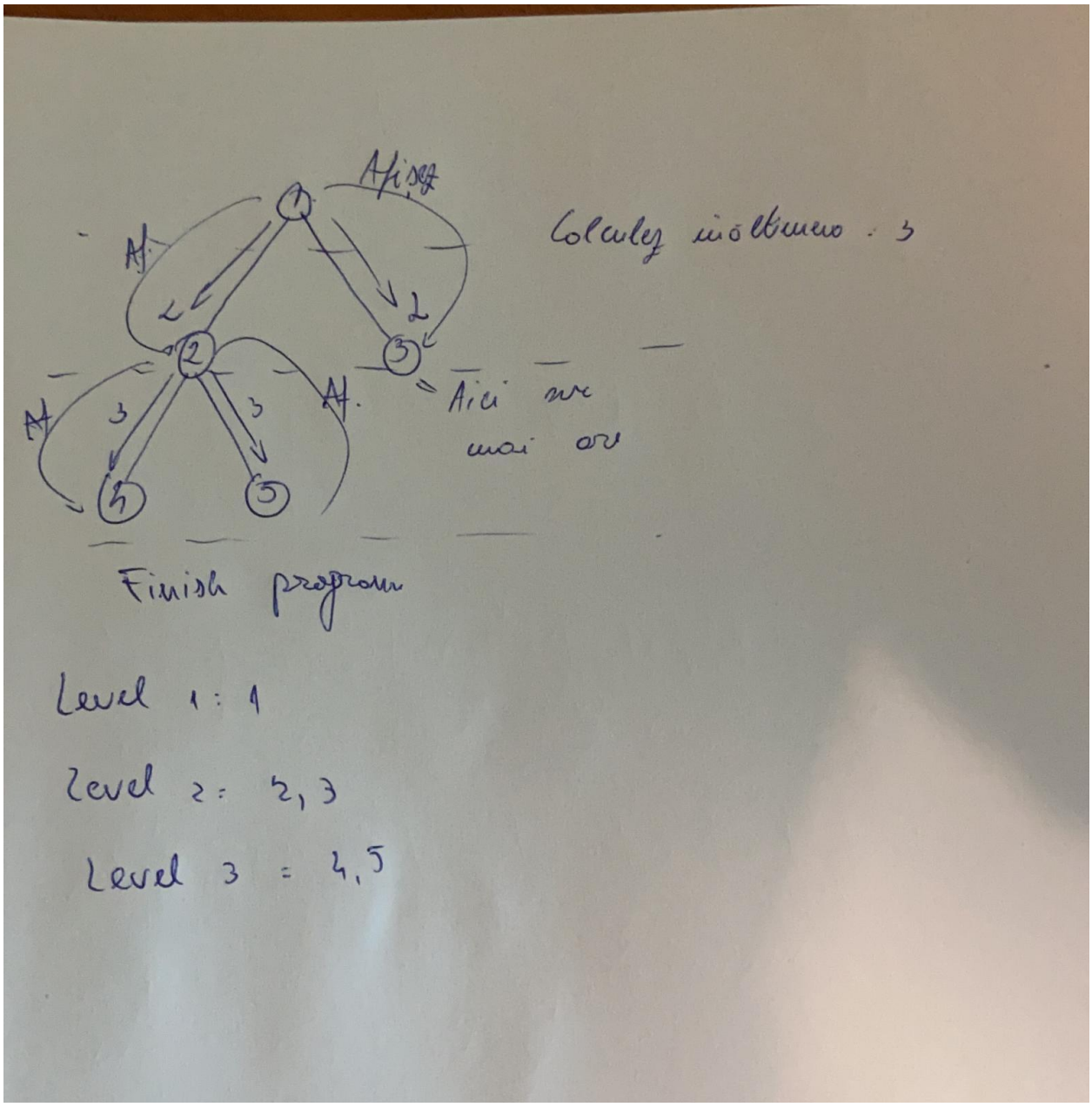
```
void printCurrentLevel(struct node_btree* root, int level);
int bheight(struct node_btree* node_btree);
struct node_btree* newnode_btree(int data);

// Functia ce printeaza fiecare nivel al arborelui, pornind de la radacina
void printLevelOrder(struct node_btree* root)
{
   // Calculez inaltimea arborelui
   int h = bheight(root);
   int i;
   // Afisez fiecare nivel, pe rand
   for (i = 1; i <= h; i++)
      printCurrentLevel(root, i);
}

// Printeaza nodurile de la nivelul curent al arborelui
void printCurrentLevel(struct node_btree* root, int level)
{
   // daca arborele binar e gol
   if (root == NULL)
      // algoritmul de termina
      return;

   // daca e primul nivel, se afiseaza doar radacina
   if (level == 1) {
```

```
        printf("%d ", root->data);
    } else if (level > 1) {
        // e nivel mai mare ca 1 si se afiseaza recursiv suboarborii stanga si dreapta
        printCurrentLevel(root->left, level - 1);
        printCurrentLevel(root->right, level - 1);
    }
}

// calculeaza inaltimea arbrorelui binar (cate nivele are)
// se numara cate noduri sunt de la radacina pana la cea mai indepartata frunza
int bheight(struct node_btree* node_btree) {
    // daca arborele este gol inaltimea e 0
    if (node_btree == NULL)
        return 0;
    else {
        // calculez recursiv inaltimea subarborului stang, respectiv drept
        int lheight = bheight(node_btree->left);
        int rheight = bheight(node_btree->right);

        // dintre cele doua inaltimi gasite, o aleg pe cea mai mare
        if (lheight > rheight)
            return(lheight + 1);
        else return(rheight + 1);
    }
}
```

```
// functie ce creaza un nou nod pentru arbore
struct node_btree* newnode_btree(int data) {
   // aloca dinamic un nod
   // aloca dinamic ca am nevoie de pointer, ca altfel informatia e alocata doar
   // in contextul local al functiei
   struct node_btree* node_btree = (struct node_btree*)
                  malloc(sizeof(struct node_btree));
   // populez cu informatii
   node_btree->data = data;
   // subarborele stang si cel drept sunt nuli
   node_btree->left = NULL;
   node_btree->right = NULL;

   return(node_btree);
}

int main()
{
   // creez un arbore simplu pe care il testez
   struct node_btree *root = newnode_btree(1);
   root->left       = newnode_btree(2);
   root->right      = newnode_btree(3);
   root->left->left  = newnode_btree(4);
   root->left->right = newnode_btree(5);
```

```
    printf("Parcurgere in latime a arborelui binar \n");
    printLevelOrder(root);
    printf("\n");

    return 0;
}
```





2.

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>
```

```
#define len(x) ((int)log10(x)+1)


/* Descriere logica:
 * Alogritmul parcurge vectorul in range-ul dat si pentru fiecare element
 * dina cel range, cauta elementul minim de la membrul respectiv la final.
 * Dupa terminarea parcurgerii, se interschimba primul element cu minimul gasit.
*/


// Functie auxiliara de interschimbare a elementelor
void swap(int *x, int *y) {
   // Iau un auxiliar
   int aux = *x;
   // AL doilea se duce in primul
   *x = *y;
   // Primul se duce in al doilea
   *y = aux;
}


void selectionSort(int nodes[], int first, int last, int (*cmp)(int *a, int *b))
{
   int i, j, min1;


   // Parucrg range-ul ce trebuie sortat de la first la last - 2
   for (i = first; i < last - 1; i++) {
      // Gasesc pozitia elemtuiului minim din [i, last - 1]
```

```
        min1 = i;
        // Parcurg de la i pana la finalul range-ului
        for (j = i + 1; j < last; j++) {
            // daca elementul e mai mic decat minimul gasit pana in acest moment
            if (cmp(&nodes[j], &nodes[min1])) {
                // actualizez noul minim
                min1 = j;
            }
        }

        // Interschimb minimul gasit cu primul element
        swap(&nodes[min1], &nodes[i]);
    }
}

// Structura de nod pentru arborelel Huffmann
struct node_btree{
    int value;
    char letter;
    struct node *left,*right;
};

typedef struct node Node;

// Functie care construieste arborele huffmann pe baza frecventelor fiecarui simbol
```

```
void buildHuffmanTree(Node **tree, char *string, struct pair *freq){
   node_btree *temp;
   node_btree *array[27];
   int i, subTrees = 0;
   struct pair smallOne, smallTwo;

   for (i = 0; i < 27; i++){
      array[i] = malloc(sizeof(node_btree));
      array[i]->value = string[i];
      array[i]->letter = i;
      array[i]->left = NULL;
      array[i]->right = NULL;
   }

   while (subTrees < 26) {
      smallOne = freq[subTrees];
      smallTwo = freq[subTrees + 1];
      temp = array[smallOne.freq];
      array[smallOne.freq] = malloc(sizeof(node_btree));
      array[smallOne.freq]->value = temp->value + array[smallTwo.freq]->value;
      array[smallOne.freq]->letter = 127;
      array[smallOne.freq]->left = array[smallTwo.freq];
      array[smallOne.freq]->right = temp;
      array[smallTwo.freq]->value = -1;
      subTrees--;
```

```
    }

    *tree = array[smallOne.freq];

return;
}

/* builds the table with the bits for each letter. 1 stands for binary 0 and 2 for binary 1 (used to facilitate arithmetic)*/
void fillTable(int codeTable[], node_btree *tree, int Code){
    if (tree->letter < 27)
        codeTable[(int)tree->letter] = Code;
    else {
        fillTable(codeTable, tree->left, Code * 10 + 1);
        fillTable(codeTable, tree->right, Code * 10 + 2);
    }

    return;
}

// Functie care face compresia la input
void compressFile(FILE *input, FILE *output, int codeTable[]){
    char bit, c, x = 0;
    int n,length,bitsLeft = 8;
    int originalBits = 0, compressedBits = 0;
```

```
// extrag valorile din fisier
while ((c=fgetc(input))!=10){
   originalBits++;
   if (c==32){
      // calculez lungimea codului din tabel
      length = len(codeTable[26]);
      n = codeTable[26];
   }
   else{
      length=len(codeTable[c-97]);
      n = codeTable[c-97];
   }

   while (length>0){
      compressedBits++;
      bit = n % 10 - 1;
      n /= 10;
      x = x | bit;
      bitsLeft--;
      length--;
      if (bitsLeft==0){
         fputc(x,output);
         x = 0;
         bitsLeft = 8;
```

```
            }
            x = x << 1;
        }
    }

    if (bitsLeft != 8){
        x = x << (bitsLeft-1);
        fputc(x,output);
    }

    fprintf(stdout, "Original bits = %dn",originalBits*8);
    fprintf(stdout, "Compressed bits = %dn",compressedBits);
    fprintf(stdout, "Saved %.2f%% of memoryn",((float)compressedBits/(originalBits*8))*100);

    return;
}

void decompressFile(FILE *input, FILE *output, node_btree *tree){
    node_btree *current = tree;
    char c,bit;
    char mask = 1 << 7;
    int i;

    while ((c=fgetc(input))!=EOF){
```

```
for (i=0;i<8;i++){
   bit = c & mask;
   c = c << 1;
   if (bit==0){
      current = current->left;
      if (current->letter!=127){
         if (current->letter==26)
            fputc(32, output);
         else
            fputc(current->letter+97,output);
         current = tree;
      }
   }

   else{
      current = current->right;
      if (current->letter!=127){
         if (current->letter==26)
            fputc(32, output);
         else
            fputc(current->letter+97,output);
         current = tree;
      }
   }
}
```

```
    }


    return;
}

/invert the codes in codeTable2 so they can be used with mod operator by compressFile function/
void invertCodes(int codeTable[],int codeTable2[]){
    int i, n, copy;


    for (i=0;i<27;i++){
        n = codeTable[i];
        copy = 0;
        while (n>0){
            copy = copy * 10 + n %10;
            n /= 10;
        }
        codeTable2[i]=copy;
    }


    return;
}


struct pair {
    char c;
    int freq;
```

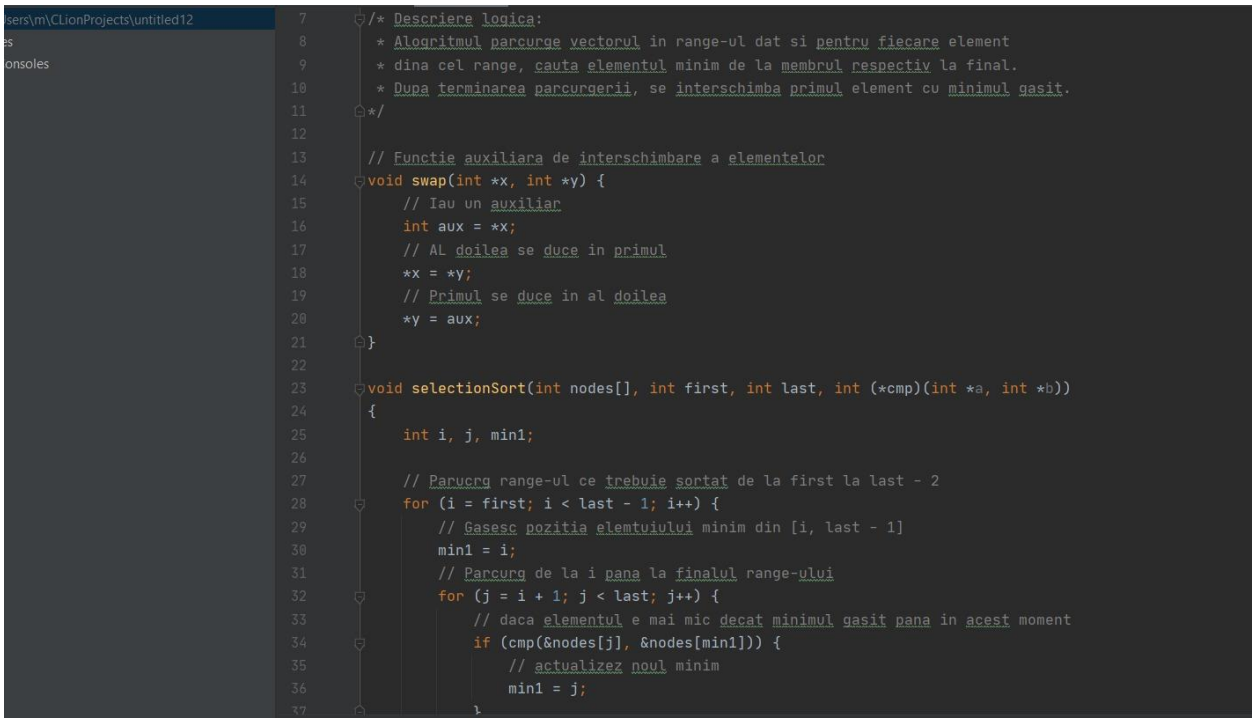
```
};

// Functie comparator care compara a cu b
int cmp(struct pair *a, struct pair *b)  {
   return a->freq < b->freq;
}



void printCurrentLevel(struct node_btree_btree* root, int level);
int bheight(struct node_btree* node_btree);
struct node_btree* newnode_btree(int data);

// Functia ce printeaza fiecare nivel al arborelui, pornind de la radacina
void printLevelOrder(struct node_btree* root)
{
   // Calculez inaltimea arborelui
   int h = bheight(root);
   int i;
   // Afisez fiecare nivel, pe rand
   for (i = 1; i <= h; i++)
      printCurrentLevel(root, i);
}

// Printeaza nodurile de la nivelul curent al arborelui
void printCurrentLevel(struct node_btree* root, int level)
```

```
{
    // daca arborele binar e gol
    if (root == NULL)
        // algoritmul de termina
        return;

    // daca e primul nivel, se afiseaza doar radacina
    if (level == 1) {
        printf("%d ", root->data);
    } else if (level > 1) {
        // e nivel mai mare ca 1 si se afiseaza recursiv suboarborii stanga si dreapta
        printCurrentLevel(root->left, level - 1);
        printCurrentLevel(root->right, level - 1);
    }
}

// calculeaza inaltimea arbrorelui binar (cate nivele are)
// se numara cate noduri sunt de la radacina pana la cea mai indepartata frunza
int bheight(struct node_btree* node_btree) {
    // daca arborele este gol inaltimea e 0
    if (node_btree == NULL)
        return 0;
    else {
        // calculez recursiv inaltimea subarborului stang, respectiv drept
        int lheight = bheight(node_btree->left);
```

```
        int rheight = bheight(node_btree->right);

        // dintre cele doua inaltimi gasite, o aleg pe cea mai mare
        if (lheight > rheight)
            return(lheight + 1);
        else return(rheight + 1);
    }
}

// functie ce creaza un nou nod pentru arbore
struct node_btree* newnode_btree(int data) {
    // aloca dinamic un nod
    // aloca dinamic ca am nevoie de pointer, ca altfel informatia e alocata doar
    // in contextul local al functiei
    struct node_btree* node_btree = (struct node_btree*)
                malloc(sizeof(struct node_btree));
    // populez cu informatii
    node_btree->data = data;
    // subarborele stang si cel drept sunt nuli
    node_btree->left = NULL;
    node_btree->right = NULL;

    return(node_btree);
}
```

```
int main(){
    Node *tree;
    int codeTable[27], codeTable2[27];
    int compress;
    char filename[20];
    FILE *input, *output;

    char string[] = {'a', 'b', 'b', 'c', 'd', 'd', 'd', 'e', 'f', 'z', 'x', 'x', 'x', 'a', 'a', 'g', 'g', 'g',
                'a', 'c', 'c', 'f', 'f', 'e', 'g', 'a'};

    struct pair freq[27];
    for (int i = 0; i < 27; i++) {
        freq[string[i] - 'a'].c = string[i];
        freq[string[i] - 'a'].freq += 1;
    }

    selectionSort(freq, 0, 27, cmp);

    buildHuffmanTree(&tree, string, freq);

    printLevelOrder(&tree);

    fillTable(codeTable, tree, 0);

    invertCodes(codeTable,codeTable2);
```

```
    printf("Type the name of the file to process:n");
    scanf("%s",filename);
    printf("Type 1 to compress and 2 to decompress:n");
    scanf("%d",&compress);

    input = fopen(filename, "r");
    output = fopen("output.txt","w");

    if (compress == 1)
        compressFile(input,output,codeTable2);
    else
        decompressFile(input,output, tree);

    return 0;
}
```




```c
/* Descriere logica:
 * Alogritmul parcurge vectorul in range-ul dat si pentru fiecare element
 * dina cel range, cauta elementul minim de la membrul respectiv la final.
 * Dupa terminarea parcurgerii, se interschimba primul element cu minimul gasit.
*/

// Functie auxiliara de interschimbare a elementelor
void swap(int *x, int *y) {
    // Iau un auxiliar
    int aux = *x;
    // AL doilea se duce in primul
    *x = *y;
    // Primul se duce in al doilea
    *y = aux;
}

void selectionSort(int nodes[], int first, int last, int (*cmp)(int *a, int *b))
{
    int i, j, min1;

    // Parucrg range-ul ce trebuie sortat de la first la last - 2
    for (i = first; i < last - 1; i++) {
        // Gasesc pozitia elemtuiului minim din [i, last - 1]
        min1 = i;
        // Parcurg de la i pana la finalul range-ului
        for (j = i + 1; j < last; j++) {
            // daca elementul e mai mic decat minimul gasit pana in acest moment
            if (cmp(&nodes[j], &nodes[min1])) {
                // actualizez noul minim
                min1 = j;
            }
```