Programmieraufgabe 5

```
Robert Wettstädt 535161
Sona Pecenakova 540607
```

1. Implementieren Sie HeapSort. Vergleichen Sie die Laufzeit Ihrer Implementierung, angewendet auf zufa□llige Listen, mit der Laufzeit der Implementierung von Quicksort aus der Lehrveranstaltung (oder Ihrer eigenen Implementierung aus Hausaufgabe 4).

```
Code: sort.c
Methode: *hSort
```

Aufruf:

```
./sort
```

Test:

```
~~~~= Arraysize 10 =~~~~

  ~~= Random Array =~~
[ 342 832 899 523 633 558 574 637 80 693 ]

Heapsort:
[ 80 342 523 558 574 633 637 693 832 899 ]

  ~~= Presorted Array =~~
Heapsort:
[ 80 342 523 558 574 633 637 693 832 899 ]
```

Laufzeiten fu□r einen Lauf

```
~~~~= Arraysize 10 =~~~~

  ~~= Random Array =~~
Heapsort  done in 0.000001 seconds and 22 comparisons
Quicksort done in 0.000001 seconds and 8 comparisons

  ~~= Presorted Array =~~
Heapsort  done in 0.000000 seconds and 22 comparisons
Quicksort done in 0.000000 seconds and 9 comparisons


~~~~= Arraysize 100 =~~~~

  ~~= Random Array =~~
```

```
Heapsort  done in 0.000012 seconds and 675 comparisons
Quicksort done in 0.000008 seconds and 175 comparisons

  ~~= Presorted Array =~~
Heapsort  done in 0.000012 seconds and 675 comparisons
Quicksort done in 0.000013 seconds and 94 comparisons


~~~~= Arraysize 1000 =~~~~

  ~~= Random Array =~~
Heapsort  done in 0.000193 seconds and 11711 comparisons
Quicksort done in 0.000108 seconds and 2479 comparisons

  ~~= Presorted Array =~~
Heapsort  done in 0.000171 seconds and 11711 comparisons
Quicksort done in 0.000831 seconds and 803 comparisons


~~~~= Arraysize 10000 =~~~~

  ~~= Random Array =~~
Heapsort  done in 0.002251 seconds and 166410 comparisons
Quicksort done in 0.001045 seconds and 35728 comparisons

  ~~= Presorted Array =~~
Heapsort  done in 0.002145 seconds and 166410 comparisons
Quicksort done in 0.033853 seconds and 13983 comparisons


~~~~= Arraysize 100000 =~~~~

  ~~= Random Array =~~
Heapsort  done in 0.027329 seconds and 2160845 comparisons
Quicksort done in 0.011484 seconds and 505711 comparisons

  ~~= Presorted Array =~~
Heapsort  done in 0.027130 seconds and 2160845 comparisons
Quicksort done in 0.705482 seconds and 282310 comparisons
```

U□ber 5 Lau□fe gemittelte Laufzeiten und Vergleichen:

```
Average run times on random arrays:
Heapsort:  0.030341 seconds
Quicksort: 0.012891 seconds

Average run times on presorted arrays:
Heapsort:  0.030404 seconds
Quicksort: 0.745334 seconds

Average comparisons on random arrays:
Heapsort:  2338866 comparisons
Quicksort: 544976 comparisons

Average comparisons on presorted arrays:
Heapsort:  2338866 comparisons
```

```
Quicksort: 297290 comparisons
```

- Quicksort ist besser als Heapsort bei random Arrays, dabei aber wesentlich schlechter bei sortierten Arrays
- Bei Heapsort bleibt die Anzahl von Vergleichen gleich und die Zeit aehnlich bei den random und sortierten Arrays

2. Schreiben Sie ein Programm, mit dem Sie ein Textfile Huffman-codieren ko□nnen. Welche Reduktion der Filegro□ße erreichen Sie damit? (Sie sollten Ihr Programm testen, indem Sie einen Text codieren, wieder dekodieren und u□berpru□fen, ob Sie damit wieder den Auggangstext erhalten.)

```
Code: huffman.c
```

Reduktion von Filegröße

```
Original bits = 664
Compressed bits = 349
Saved 52.56% of memory

Original size = 83 bytes
Compressed size = 44 bytes
```

Test

Original File: text.txt

```
this is an exercise for the course algorithms that we are taking in our university
```

CodeTable:

```
u: 00000
c: 00001
h: 0001
r: 0010
s: 0011
e: 010
 : 011
n: 1000
b: 100100
x: 100101000
z: 100101001
j: 100101010
q: 1001010110
k: 1001010111
v: 1001011
p: 100110
```

```
y: 100111
i: 1010
o: 1011
a: 1100
l: 11010
d: 11011
g: 111000
f: 111001
w: 111010
m: 111011
t: 1111
```

CompressedFile: compressedFile.txt

```
0000000: 11110001 10100011 01110100 01101111 00100001 10101001  ..to!.
0000006: 01000010 00100000 11010001 10100111 11001101 10010011  B ....
000000c: 11110001 01001100 00110110 00000010 00110100 11110011  .L6.4.
0000012: 01011100 01011001 01010111 10001111 01100110 11111100  \YW.f.
0000018: 01110011 11011111 01001001 11100001 00100111 11111001  s.I.'.
000001e: 00101011 11010100 01110000 11101010 00011101 10000000  +.p...
0000024: 10011000 00100010 10100101 10100010 00111010 11111001  ."..:.
000002a: 11110110 11111000                                      ..
```

DecompressedFile: decompressedFile.txt

```
this is an exercise for the course algorithms that we are taking in our universityd
g
```

- Die Größe der Textdatei ist mit dem Huffman Coding ziemlich stark reduziert, mehr als 50%
- Die Dekompression funktioniert allerdings noch nicht perfekt, z.B. bei den letzten Bits wird der Originaltext nocht um ein paar Buchstaben erweitert
- Die Codetable wird anhand von den im Wikipedia gefundenen english letter frequencies bestimmt

Quellcode

heapsort.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

int heapSize = 0;

void swap(int A[], int i, int j) {
    const int t = A[i];
    A[i] = A[j];
    A[j] = t;
}
```

```
void maxHeapify(int A[], int i){

    int l = i*2+1;
    int r = i*2+2;
    int maximum = 0;

    if(l < heapSize && A[l] > A[i]){
        maximum = l;
    }
    else {
        maximum = i;
    }

    if(r < heapSize && A[r] > A[maximum]){
        maximum = r;
    }

    if(maximum != i){
        swap(A, i, maximum);
        maxHeapify(A, maximum);
    }
}

void buildMaxHeap(int A[], int size){
    heapSize = size;
    for(int i = (size-1)/2; i >= 0; i--){
        maxHeapify(A, i);
    }
}


int *hSort(int A[], int size){
    buildMaxHeap(A, size);
    for(int i = (size-1); i >= 1; i--){
        swap(A, 0, i);
        heapSize = heapSize - 1;
        maxHeapify(A, 0);
    }
    return A;
}


//HELPER FUNCTIONS FOR OUTPUT
void printArray (int array[], const int size) {
    printf("[ ");
    for (int i = 0; i < size; ++i) {
        printf("%d ", array[i]);
    }
    printf("]\n");
}

int *getRandomArray (const int size) {
    time_t t;
    srand((unsigned) time(&t));
    int *array = calloc(size, sizeof(int));

    for (int i = 0; i < size; ++i) {
```

```
        array[i] = rand() % 1000;
    }
    return array;
}

int *copyArray (int array[], const int size) {
    int *A = calloc(size, sizeof(int));
    memcpy(A, array, size * sizeof(int));
    return A;
}

int main(int argc, char *argv[])
{
    int arraysize;
    int *array;
    clock_t t;

    arraysize = 10;
    printf("\n\n~~~~= Arraysize %d =~~~~\n", arraysize);

    array = getRandomArray(arraysize);
    printArray(array, arraysize);

    int *A = copyArray(array, arraysize);
    t = clock();
    hSort(A, arraysize);

    t = clock() - t;

    double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
    printArray(A, arraysize);
}
```

huffman.c

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

// This constant can be avoided by explicitly calculating height of Huffman Tree
#define MAX_TREE_HT 100
#define len(x) ((int)log10(x)+1)

// A Huffman tree node
struct MinHeapNode
{
    char data;  // One of the input characters
    unsigned freq;  // Frequency of the character
    struct MinHeapNode *left, *right; // Left and right child of this node
};

// A Min Heap:  Collection of min heap (or Hufmman tree) nodes
struct MinHeap
{
    unsigned size;    // Current size of min heap
    unsigned capacity;   // capacity of min heap
```

```
    struct MinHeapNode **array;  // Attay of minheap node pointers
};

// A utility function allocate a new min heap node with given character
// and frequency of the character
struct MinHeapNode* newNode(char data, unsigned freq)
{
    struct MinHeapNode* temp =
    (struct MinHeapNode*) malloc(sizeof(struct MinHeapNode));
    temp->left = temp->right = NULL;
    temp->data = data;
    temp->freq = freq;
    return temp;
}

// A utility function to create a min heap of given capacity
struct MinHeap* createMinHeap(unsigned capacity)
{
    struct MinHeap* minHeap =
    (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->size = 0;  // current size is 0
    minHeap->capacity = capacity;
    minHeap->array =
    (struct MinHeapNode**)malloc(minHeap->capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}

// A utility function to swap two min heap nodes
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// The standard minHeapify function.
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->array[left]->freq < minHeap->array[smallest]->freq)
        smallest = left;

    if (right < minHeap->size &&
        minHeap->array[right]->freq < minHeap->array[smallest]->freq)
        smallest = right;

    if (smallest != idx)
    {
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}
```

```
// A utility function to check if size of heap is 1 or not
int isSizeOne(struct MinHeap* minHeap)
{
    return (minHeap->size == 1);
}

// A standard function to extract minimum value node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    struct MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}

// A utility function to insert a new node to Min Heap
void insertMinHeap(struct MinHeap* minHeap, struct MinHeapNode* minHeapNode)
{
    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && minHeapNode->freq < minHeap->array[(i - 1)/2]->freq)
    {
        minHeap->array[i] = minHeap->array[(i - 1)/2];
        i = (i - 1)/2;
    }
    minHeap->array[i] = minHeapNode;
}

// A standard funvtion to build min heap
void buildMinHeap(struct MinHeap* minHeap)
{
    int n = minHeap->size - 1;
    int i;
    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

// A utility function to print an array of size n
void printArr(int arr[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        printf("%d", arr[i]);
    printf("\n");
}

// Utility function to check if this node is leaf
int isLeaf(struct MinHeapNode* root)
{
    return !(root->left) && !(root->right) ;
}

// Creates a min heap of capacity equal to size and inserts all character of
// data[] in min heap. Initially size of min heap is equal to capacity
struct MinHeap* createAndBuildMinHeap(char data[], int freq[], int size)
{
```

```
    struct MinHeap* minHeap = createMinHeap(size);
    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);
    minHeap->size = size;
    buildMinHeap(minHeap);
    return minHeap;
}

// The main function that builds Huffman tree
struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size)
{
    struct MinHeapNode *left, *right, *top;

    // Step 1: Create a min heap of capacity equal to size.  Initially, there are
    // modes equal to size.
    struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

    // Iterate while size of heap doesn't become 1
    while (!isSizeOne(minHeap))
    {
        // Step 2: Extract the two minimum freq items from min heap
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        // Step 3:  Create a new internal node with frequency equal to the
        // sum of the two nodes frequencies. Make the two extracted node as
        // left and right children of this new node. Add this node to the min heap
        // '$' is a special value for internal nodes, not used
        top = newNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        insertMinHeap(minHeap, top);
    }

    // Step 4: The remaining node is the root node and the tree is complete.
    return extractMin(minHeap);
}

// Prints huffman codes from the root of Huffman Tree.  It uses arr[] to
// store codes

long long int codeTable[27]; long long int codeTable2[27];
void getCodes(struct MinHeapNode* root, int arr[], int top)
{
    // Assign 0 to left edge and recur
    if (root->left)
    {
        arr[top] = 0;
        getCodes(root->left, arr, top + 1);
    }

    // Assign 1 to right edge and recur
    if (root->right)
    {
        arr[top] = 1;
        getCodes(root->right, arr, top + 1);
    }
```

```
    // If this is a leaf node, then it contains one of the input
    // characters, print the character and its code from arr[]
    if (isLeaf(root))
    {
        char letter = root->data;
        printf("%c: ", letter);
        printArr(arr, top);

        //codes are saved as 1 for binary 0 and 2 for binary 1 for arithmetic
purposes of saving the code
        long long int code = 0;
        for(int i = 0; i < top; i++){
            if(arr[i] == 0){
                code = code*10+1;
            }
            else if(arr[i] == 1){
                code = code*10+2;
            }
        }

        //32 is ASCII for space, 97 is ASCII for 'a'
        if(letter == 32){
            codeTable[26] = code;
            //printf("codeTable[%d]: %d\n", 26, code);
        }
        else{
            codeTable[letter-97] = code;
            //printf("codeTable[%d]: %d\n", letter-97, code);
        }

    }


}

// The main function that builds a Huffman Tree and print codes by traversing
// the built Huffman Tree
void HuffmanCodes(char data[], int freq[], int size)
{
    //  Construct Huffman Tree
    struct MinHeapNode* root = buildHuffmanTree(data, freq, size);

    // Print Huffman codes using the Huffman tree built above
    int arr[MAX_TREE_HT], top = 0;
    getCodes(root, arr, top);
}


/*function to compress the input*/
void compressFile(FILE *input, FILE *output, long long int codeTable[]){
    /*function to compress the input*/
    char bit, c, x = 0;
    long long int n;
    int length,bitsLeft = 8;
    int originalBits = 0, compressedBits = 0;

    //32 is ASCII space, 97 is ASCII a
```

```
    while ((c=fgetc(input))!=EOF){
        originalBits++;
        if (c==32){
            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);
    }

    /*print details of compression on the screen*/
    fprintf(stderr,"Original bits = %d\n",originalBits*8);
    fprintf(stderr,"Compressed bits = %d\n",compressedBits);
    fprintf(stderr,"Saved %.2f%% of
memory\n",((float)compressedBits/(originalBits*8))*100);

    return;

}

void decompressFile(FILE *input, FILE *output, struct MinHeapNode *root){
    struct MinHeapNode *current = root;
    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 (isLeaf(current)){
                    if (current->data==32){
```

```
                        fputc(32, output);
                    }
                    else{
                        fputc(current->data,output);
                    }
                    current = root;
                }
            }

            else{
                current = current->right;
                if (isLeaf(current)){
                    if (current->data==32)
                        fputc(32, output);
                    else
                        fputc(current->data,output);
                    current = root;
                }
            }
        }
    }

    return;

}

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


// Driver program to test above functions
int main()
{

    /* 81 = 8.1%, 128 = 12.8% and so on. The 27th frequency is the space. Source is
Wikipedia */
    int freq [27] = {81, 15, 28, 43, 128, 23, 20, 61, 71, 2, 1, 40, 24, 69, 76, 20,
1, 61, 64, 91, 28, 10, 24, 1, 20, 1, 130};
    char data[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', ' '};
```

```
    //char letters[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    //int freq[] = {5, 9, 12, 13, 16, 45};

    int size = sizeof(data)/sizeof(data[0]);
    //HuffmanCodes(letters, freq, size);

    //  Construct Huffman Tree
    struct MinHeapNode* tree = buildHuffmanTree(data, freq, size);
    // Print Huffman codes using the Huffman tree built above
    int arr[MAX_TREE_HT], top = 0;
    getCodes(tree, arr, top);

    invertCodes(codeTable,codeTable2);


    int compress;
    char filename[20];
    FILE *fileToProcess, *compressedFile, *decompressedFile;

    /*get input details from user*/
    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);


    fileToProcess = fopen(filename, "r");

    if (compress==1){
        compressedFile = fopen("compressedFile.txt","w");
        compressFile(fileToProcess,compressedFile, codeTable2);
    }
    else{
        decompressedFile = fopen("decompressedFile.txt","w");
        decompressFile(fileToProcess,decompressedFile, tree);
    }


    return 0;
}
```