Modelling for Science and Engineering
Optimization
Course 2025–26

1 Binary heap

Introduction

Organizing the data of a sorted queue can improve the performance of our programs. One of the easiest implementation can be a sorted linked list as presented below. In this delivery you must implement a binary heap queue, which encodes also the elements keeping the order and in a way that reduces the number of operations when adding new elements to the queue.

Initial code

The following code loads the information of a list of vertices from a given file that must be introduced as an argument. After loading the vertices in a vector, it creates a linked list of vertices sorted by a given field looking for the position sequentially and prints the result. To check the result, there is a function which prints the linked list after reading the file. To check that removing the first element works, the printing function is also called after removing the first element.

```c
// Reads a file with two fields organized with two columns separated by comma.
// Organizes the data as a sorted linked list.
// Prints the linked list.
// Removes first element of the linked list and prints again the linked list.
#include <stdio.h>
#include <stdlib.h>
#include <values.h> // To have MAXFLOAT

// The data we will deal with are vertices with just:
// an id, which is up to 4 characters.
// a value (we will use it to order the list).
typedef struct{
    char id[5];
    double value;
}Vertex;

// Defines a node in the ordered queue (a linked list node) which contains:
// a pointer to a Vertex and
// a pointer to the next element.
typedef struct OQueueElement{
    Vertex *vertex;
    struct OQueueElement *next;
}OQueueElement;

// Defines the structure of an ordered queue:
// only keeps a pointer to the first element
typedef struct{
    OQueueElement *first;
}OQueue;

int Oenqueue(Vertex *, OQueue *);
Vertex Odequeue(OQueue *);
void printOqueue(OQueue);

```

```c
int main(int argc, char *argv[])
{
    FILE *datafile;
    char ll;
    Vertex *vertexlist;
    unsigned vertexnumber = 0;

    if(argc<2){
        printf("Error: need the name of the file.\n");
        return 1;
    }
    datafile = fopen(argv[1],"r");
    if(datafile == NULL){
        printf("Error: cannot open the file.\n");
        return 1;
    }
    while((ll=fgetc(datafile)) != EOF){
        if (ll=='\n'){vertexnumber++;}
    }
    rewind(datafile);
    // After rewinding the file, it allocates memory for the vertices and reads them into the 'vertexlist' array.
    if((vertexlist = (Vertex *) malloc(vertexnumber * sizeof(Vertex))) == NULL){
        printf ("Error: cannot allocate memory for vertexs.\n");
        return 1;
    }
    for(int i=0;i<vertexnumber;i++){
        fscanf(datafile, "%[^,],%lf\n", vertexlist[i].id, &vertexlist[i].value);
    }
    fclose(datafile);
    OQueue vertexoqueue;
    vertexoqueue.first = NULL;
    for(int i=0;i<vertexnumber;i++){
        if(Oenqueue(&vertexlist[i],&vertexoqueue)!=0){
            printf("Error: problem adding elements to the queue.\n");
            return 666;
        }
    }
    printOqueue(vertexoqueue);
    Vertex v = Odequeue(&vertexoqueue);
    printf("We have extracted vertex %s from the queue.\n",v.id);
    printOqueue(vertexoqueue);
    return 0;
}

// The function Oenqueue inserts vertices into an ordered queue, keeping the elements sorted by the value field of the Vertex.
int Oenqueue(Vertex *vert, OQueue *Q)
{
    OQueueElement *element_aux, *element_iter;
    element_aux = (OQueueElement *) malloc(sizeof(OQueueElement));
    if(element_aux == NULL){
        printf("Error: cannot allocate memory for queue element.");
        return 1;
    }
    element_aux -> vertex = vert;
    element_aux -> next = NULL;
    if(Q->first == NULL){
        Q->first = element_aux;
        return 0;
    }
    if(Q->first->vertex->value > element_aux->vertex->value){
```

```
95          element_aux -> next = Q->first;
96          Q->first = element_aux;
97          return 0;
98      }
99      element_iter = Q->first;
100     while(element_iter->next != NULL && element_iter->next->vertex->value < vert
    ->value) element_iter = element_iter->next;
101     element_aux->next = element_iter->next;
102     element_iter->next = element_aux;
103     return 0;
104 }
105
106 // The Odequeue function removes and returns the first vertex from the queue.
107 Vertex Odequeue(OQueue *Q)
108 {
109     Vertex v;
110     if(Q->first == NULL) { //there is no queue!
111         v.id[0] = 0;
112     }
113     OQueueElement *element_aux = Q->first;
114     v = *(element_aux->vertex);
115     Q->first = Q->first->next;
116     free(element_aux);
117     return v;
118 }
119
120 // The printOqueue function prints the elements of an ordered queue.
121 void printOqueue(OQueue Q)
122 {
123     OQueueElement *element_iter=Q.first;
124     while(element_iter != NULL){
125         printf("Vertex id = %s, value = %lf.\n",element_iter->vertex->id,
    element_iter->vertex->value);
126         element_iter = element_iter->next;
127     }
128 }
```

Check the program compiling and executing it. For example, if you execute it with the argument `VertexList.txt` the program should return:

```
Vertex id = a, value = 25.000000.
Vertex id = c, value = 28.000000.
Vertex id = f, value = 29.000000.
Vertex id = g, value = 30.000000.
Vertex id = b, value = 70.000000.
Vertex id = d, value = 75.000000.
Vertex id = h, value = 78.000000.
Vertex id = e, value = 93.000000.
Vertex id = i, value = 99.000000.
We have extracted vertex a from the queue.
Vertex id = c, value = 28.000000.
Vertex id = f, value = 29.000000.
Vertex id = g, value = 30.000000.
Vertex id = b, value = 70.000000.
Vertex id = d, value = 75.000000.
Vertex id = h, value = 78.000000.
Vertex id = e, value = 93.000000.
Vertex id = i, value = 99.000000.
```

Binary heap

A **binary heap** is an abstract data structure that takes the form of a rooted binary tree (each node has at most two children), where each position contains a value (usually a pointer to a data structure with the value as part of this data) used to implement priority queues based on that value.

The data structure has two additional constraints:

- Shape property: The binary heap is a complete binary tree; that is, all levels of the tree, except possibly the last (the deepest), are completely filled (with two children per node), and if the last level of the tree is not complete, the nodes at that level are filled from left to right.
- Heap property: The value stored at each node is less than or equal to the values of its children, according to some total order.

We define the depth recursively: the depth of the root is zero, and the depth of each child is one more than the parent.

We can add elements and remove the root in a binary heap following the following algorithms:

- To **add** a new element we must **heapify up**: create a new element in the queue and add it after the last position of the queue keeping the *shape property*. Now, iterate the following procedure till it is in the correct position: if the value of the parent is bigger that the value of the new element, swap the position of these two elements.
- To **remove** the root we must **heapify down**: after removing the root the result is a non connected tree, so we have to decide which elements is the new root. To do that, move the last element of the binary heap to the root position. Now, iterate the following procedure till it is in the correct position: if the value of the smaller child is smaller than the value of this element, swap the position of these two elements.
- If we **modify** a value in a given binary tree we heapify the corresponding node up or down depending if the value has decreased (heapify up) or decreased (heapify down).

Position of an element in a complete binary tree: assume we enumerate the elements in a complete binary tree starting by number 1 on the root and proceeding by levels and ordering from left to right (in Figure 1 the enumeration would be **a**$= 1$, **b**$= 2$, ..., **i**$= 9$), we can find the position of the n-th element considering n in base 2 (for example the position 9th would be 1001), then, starting from the root and the second digit of n in base 2, and following left or right child depending whether there is a 0 or 1 in the representation of n in base 2. For example, the position of the 9th element (**1001** in base 2) is left-left-right.
Remark: there are functions in C which allow to extract the bits of an integer (in general, including long integers and unsigned), which correspond to the representation of the integer in base 2.

Exercise 1: Written exercise (1,5 pts)

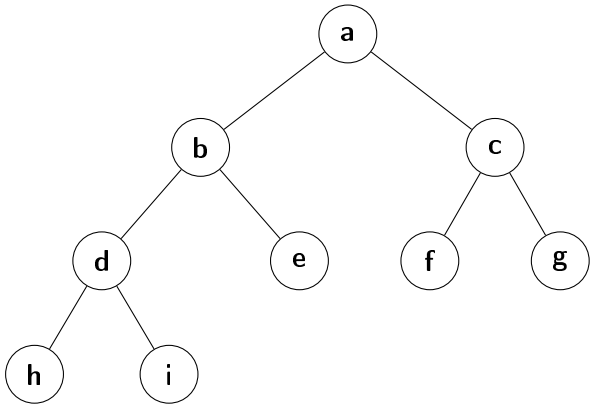
You must do this exercise by hand: upload a PDF file with the solution. Please, add your name and NIU to the file.

Consider the binary heap of Figure 1.

(a) Add a node **j** to **BH** with value the last to digits of your NIU and show the resulting "*binary heap*".

(b) Remove the root of **BH** and show the resulting "*binary heap*".

(c) Subtract the biggest digit of your NIU to the value of **f** in **BH** show the resulting "*binary heap*".



Level	Label	Value
0	**a**	25
1	**b**	70
1	**c**	28
2	**d**	75
2	**e**	93
2	**f**	29
2	**g**	30
3	**h**	78
3	i	99

Figure 1: "*Binary heap*" **BH**.

Exercise 2: Code in C (8,5 pts)

Modify the initial code and organize the data from a given file as in the **Initial code** in a queue as a binary heap (in particular, consider than the **id** of a Vertex is a string of at most 4 characters). Your code must include:

- A structure `BHQueueElement` to define each element of the binary heap. This structure must be:

```
typedef struct BHQueueElement{
  Vertex *vertex;
  struct BHQueueElement *parent;
  struct BHQueueElement *leftchild, *rightchild;
}BHQueueElement;
```

- A structure `BHQueue` to define the binary heap. It will also need more information for adding new elements: the position of the new element depends on the number of elements in the binary heap so, one option, is adding the `size` to the structure (see **Position of an element in a complete binary tree**).
- A function `BHenqueue` which adds a vertex to a given binary heap.
- A function `BHdequeue` which removes the root to a given binary heap.
- A function `BHrequeueelement` which modifies the position after changing the value of a given vertex of the binary heap.
- A function `printBH` which prints the binary heap in a human readable way. For example:

```
Binary heap with 9 elements and root node a.
Node a, value = 25.000000, left child: b, right child: c.
Node b, value = 70.000000, left child: d, right child: e.
Node c, value = 28.000000, left child: f, right child: g.
Node d, value = 75.000000, left child: h, right child: i.
Node e, value = 93.000000.
Node f, value = 29.000000.
Node g, value = 30.000000.
Node h, value = 78.000000.
Node i, value = 99.000000.
```

This function will traverse a graph, so you may need an ordinary queue to follow all the nodes.

With these functions you must deliver two `C` programs. Remember to add your name and NIU as a comment in the first line of code of each program.

- One program called `BH1.c` where the `main` function of the program must solve Exercise 1 using the defined functions, printing each result.
- A second program called `BH2.c` which just reads a given file (as an argument) and prints the data structured as a binary heap. This code will be tested with different files which may include a large number of vertices.

The evaluation of the code will consider:

- That the code compiles without errors and warnings. If your submission does not contain all the functions or does not work properly explain it as a comment at the beginning of each file.
- That the chosen structures and declared functions are defined according to these instructions.
- That the memory use is under control.
- That the code is human readable: we recommend to use indentation and comments when necessary.