TÉCNICO LISBOA

DEI
Departamento de Engenharia Informática

Advanced Algorithms

2022/2023

Project 1

Implement a Meldable Heap

This project considers the problem of constructing a meldable heap. The first delivery deadline for the project code is the 10th of March, at 12:00. The deadline for the recovery season is 14th July at 12:00, the system opens the 12th July at 12:00. The deadline for the special season is 26th July at 12:00, the system opens the 24th at 12:00.

Students should not use existing code for the algorithms described in the project, either from software libraries or other electronic sources. They should also not share their implementation with colleagues, specially not before the special season deadline.

It is important to read the full description of the project before starting to design and implementing solutions.

The delivery of the project is done in the mooshak system.

1 Meldable Heaps

The challenge is to implement the meldable heap data structure. A description of this data structure was given by Gambin and Malinowski [1998]. The implementation must strictly verify the specification given in this script, meaning that the resulting output must match exactly the one described in this document. Moreover the structure described in this script is designed to reduce the complexity of the implementation.

The input will consist of a sequence of heap commands, each one will correspond to a heap operation.

1.1 Description

Let us start by describing the basic building block of the binomial heap, the `struct node` that is used to store information about a node.

```
typedef struct node* node;

struct node
{
  int v; /* The value to store */
  node leftChild; /* Left child */
  node rightChild; /* Right child */
  node *hook; /* Pointer to field in the father node */
};
```

Most fields are explained by the comments. The field `v` is used to store the value of the node. Initially this value is set to 0, until it is changed by a `Set` command. The `leftChild` field points to the left child of the node. If the node does not have a left child then this pointer should be set to `NULL`. The `rightChild` field points to the right child of the node. If the node does not have a left child then this pointer should be set to `NULL`. Finally the `hook` field points to a child field in the father `node`, in case such a node, i.e., when the current node is not the root of the tree. When the current node is a root the `hook` pointer stores `NULL`. When the current node is the left child of its parent the `hook` pointer points to the `leftChild` field in its father struct. When the current node is the right child of its parent the `hook` pointer points to the `rightChild` field in its father struct.

The `ExtracMin` and `Delete` operations should reset a `node` to its default initial state. Meaning that the field `v` should be set to 0 and the `leftChild`, `rightChild` and `hook` fields should be set to `NULL`.

To print the information related to a `node` we will use the following code.

```
int ptr2loc(node v, node A)
{
  int r;
  r = -1;
  if(NULL != v)
    r = ((size_t) v - (size_t) A) / sizeof(struct node);

  return (int)r;
}
```

```
void showNode(node v)
{
  int f;

  if(NULL == v)
    printf("NULL\n");
  else {
    printf("node: %d ", ptr2loc(v, A));
    printf("v: %d ", v->v);
    printf("leftChild: %d ", ptr2loc(v->leftChild, A));
    printf("rightChild: %d ", ptr2loc(v->rightChild, A));

    f = ptr2loc((node)v->hook, A);

    printf("father: %c %d",
           &(A[f].leftChild) == v->hook ? 'l' : 'r',
           f);

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

First let us make a brief description of the operations the implementation must support.

S `(node)` The function `showNode` given above.

P `(node)` The `showHeaps` function that gives a pre-order description of the sub-tree rooted at a node, i.e., it calls `showNode` for the current `node` and then recursively call `showHeaps` on the leftChild and then recursively calls `showHeaps` on the rightChild.

V `(node, v)` The `Set` function that changes the `v` field of the current `node`. Note that this function can only be executed when the `node` is a heap by itself, i.e., its `leftChild`, `rightChild` and `hook` fields are all `NULL`.

U `(heap, heap)` The `Meld` function is used to join two heaps, i.e., merge the two roots into a single tree. This function returns the root of the resulting tree.

R `(node, v)` The `DecreaseKey` function is used to decrease the `v` field of the current `node`. This `node` may be part of a meldable heap tree and therefore this operation might need modify this tree.

M `(heap)` The `Min` function returns the minimum value that exists in the given meldable heap tree.

A `(heap)` The `ArgMin` function returns an identification of the node that contains the minimum value that exists in the given meldable heap tree.

E `(heap)` The `ExtractMin` function removes the `node` that contains the minimum value in the given tree and returns the, possibly new, identification of the resulting tree.

D `(node)` The `Delete` function removes a specific node from the its tree and returns the identification of the resulting tree.

We can now discuss these operations in more detail. First we consider the issue of node, tree and heap identifiers, i.e., how are we going to represent these structures in function arguments and outputs. We will use integers, which represent index positions. The first value that is given in the input is a value `n` that indicates how many `struct node` instances will be necessary for the commands that follow. We therefore first alloc an array `A` containing `n` of these structs as follows:

```
scanf("%d", &n);

A = (node)calloc(n, sizeof(struct node));
```

We can now refer to each `node` by its index in `A`, i.e., we will use i to represent the `node` in `A[i]`. Hence the number 0 represents the first node. Note that the `calloc` function guarantees that the allocated memory is zeroed, which guarantees that the `v` fields are set to 0 and also that the remaining fields are set to `NULL`.

In this initial configuration the `node` in number 0 is also a meldable heap tree and therefore also a meldable heap. In fact any index i represents a different meldable heap. To merge these heaps we will use the `Meld` operation that we will describe next.

To illustrate the `Meld` operation consider the two heaps in Figure 1. Each node contains its index in `A` and its `v` value. In this case the `v` values are mostly 0, except for `A[12]`, `A[13]` and `A[14]`. These heaps will occur in the first input given below. The arrows labeled `le` represent the `leftChild` pointers, the ones labeled `ri` the `rightChild` pointers and the arrows labeled `f` the `father` pointers. Note that the root of the first tree is the node 0, but the root of the second tree is the node 12. Now consider the command `U 0 12`, which will meld these two heaps.

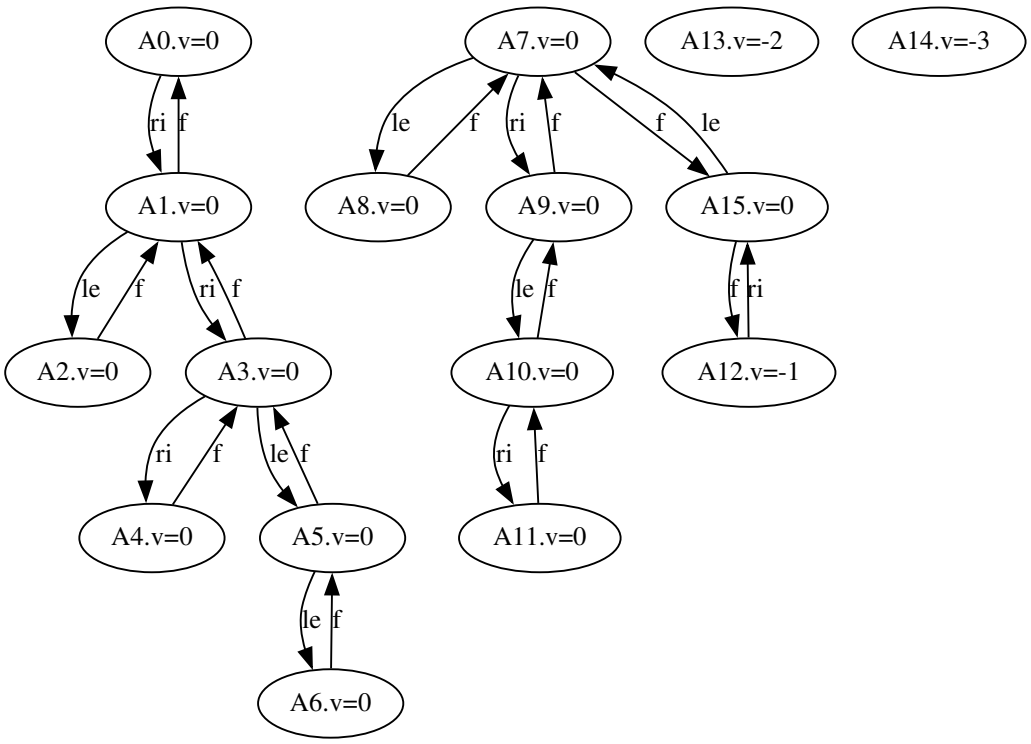

Figure 1: Representation of four meldable heap trees before `Meld` operation.

Before executing this command we can execute the `P 12` command to obtain the inorder representation of the second heap. The resulting output would then be the following:

```
node: 12 v: -1 leftChild: -1 rightChild: 15 father: r -1
node: 15 v: 0 leftChild: 7 rightChild: -1 father: r 12
node: 7 v: 0 leftChild: 8 rightChild: 9 father: l 15
node: 8 v: 0 leftChild: -1 rightChild: -1 father: l 7
node: 9 v: 0 leftChild: 10 rightChild: -1 father: r 7
node: 10 v: 0 leftChild: -1 rightChild: 11 father: l 9
node: 11 v: 0 leftChild: -1 rightChild: -1 father: r 10
```

Likewise to the `P 0` command would produce the following output:

```
node: 0 v: 0 leftChild: -1 rightChild: 1 father: r -1
node: 1 v: 0 leftChild: 2 rightChild: 3 father: r 0
node: 2 v: 0 leftChild: -1 rightChild: -1 father: l 1
node: 3 v: 0 leftChild: 5 rightChild: 4 father: r 1
node: 5 v: 0 leftChild: 6 rightChild: -1 father: l 3
node: 6 v: 0 leftChild: -1 rightChild: -1 father: l 5
node: 4 v: 0 leftChild: -1 rightChild: -1 father: r 3
```

Let us consider the execution of the command `U 0 12`. The output produced by this command is the following :

```
Meld A[0] A[12]
Swap to A[12] A[0]
Flipped a 1
Meld A[15] A[0]
Flipped a 1
Meld A[-1] A[0]
link A[0] as rightChild of A[15]
link A[15] as rightChild of A[12]
12
```

In this particular call we have that Q_1 is the tree rooted at 0 and Q_2 the tree rooted at 12. First we compare the trees rooted at 0 and at 12. Since the `v` value in $A[12].v = -1 < 0 = A[0].v$ the trees get swapped and therefore Q_1 becomes the tree rooted at 12 and Q_2 the tree rooted at 0. It very important to notice when both the `v` values are equal this kind of swap never occurs, i.e., in case of a tie there is no swap.

Now the algorithm flips a coin and obtains a value of 1. This means that the algorithm proceeds by execution a recursive call between the right child

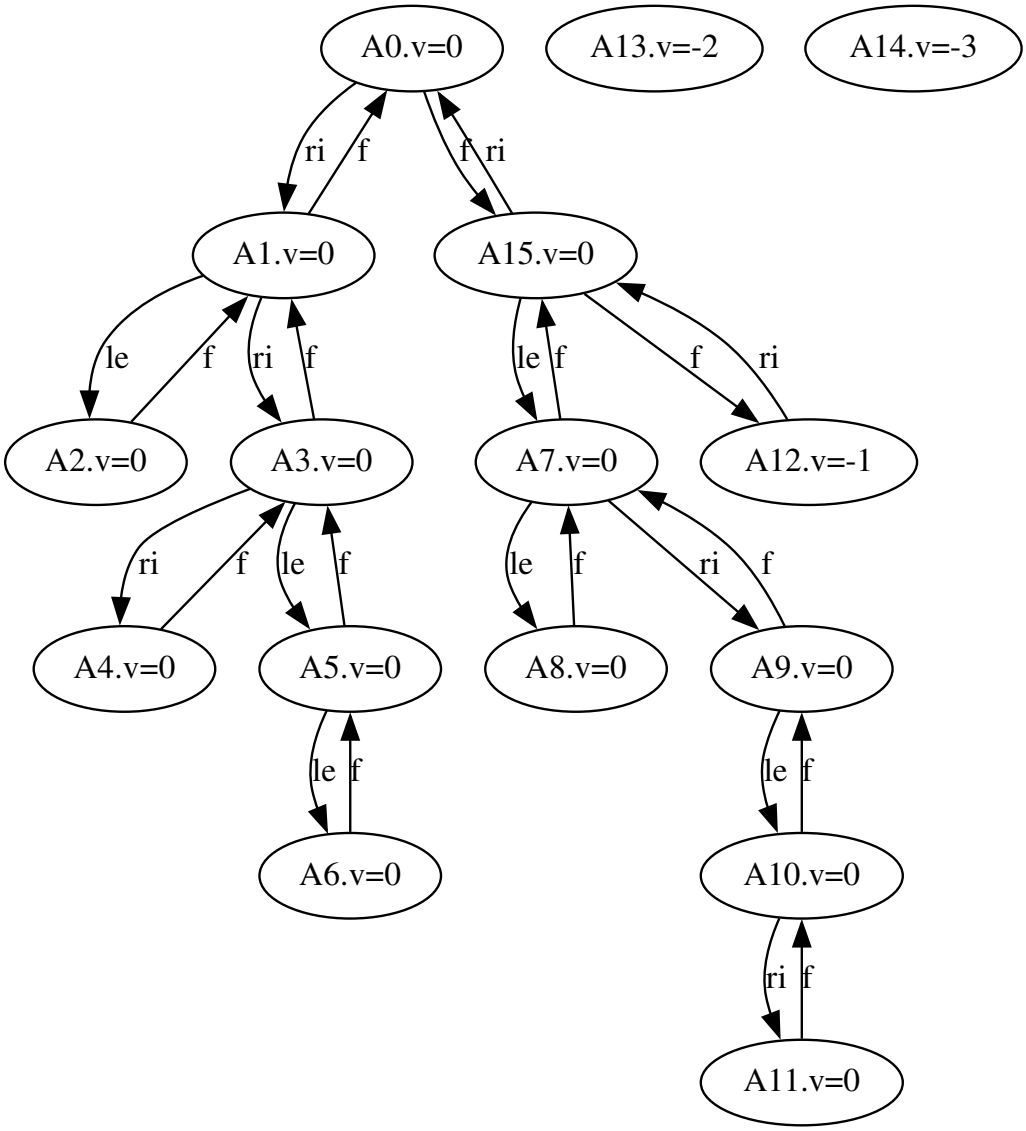

Figure 2: Representation after `Meld` operation.

of Q_1, which is $A[15]$, and Q_2, which is now $A[0]$. Again the algorithm flips a coin and obtains a value of 1. Hence it performs a recursive call between the right child of Q_1, which is an empty sub-tree, and Q_2, which is still $A[0]$. When the recursive call receives an empty sub-tree for Q_1 it simply returns Q_2, which becomes the `rightChild` of $A[15]$. Then the resulting tree, rooted at $A[15]$ becomes the `rightChild` of $A[12]$. Finally the procedure returns 12, as this is the root of the resulting tree. The resulting structure can be observed in Figure 2.

The format of the strings to produce the output for the `Meld` function is the following:

```
  printf("Meld A[%d] A[%d]\n", q1, q2);
    printf("Swap to A[%d] A[%d]\n", q2, q1);
```

An important sub-routine that is used in this process is the `link` function that changes the corresponding `child` and `hook` pointers. Here is a simple prototype for this function, with the corresponding message format.

```
static void
link(node f, node c, int d)
{
  printf("link A[%d] as %s of A[%d]\n",
         ptr2loc(c, A),
         d ? "rightChild" : "leftChild",
         ptr2loc(f, A));
```

The arguments of the function correspond to the father node `f`, the child node `c` and the flipped direction as `d`.

A very important issue is how the random values are obtained. You must use the following code for this process, so that your execution is reproducible and matches the correct output.

```
#define _DEFAULT_SOURCE
#include <stdlib.h>

static int
randBit(void)
{
  static unsigned int M = 0;
  static unsigned int P;
  int r;
```

```
  if(0 == M){
    M = ~0;
    P = random();
  }

  r = P & 1;

  M = M>>1;
  P = P>>1;

  printf("Flipped a %d\n", r);

  return r;
}
```

Every time the `Meld` operation requires a random bit this function can be called. You should not call this function anywhere else in your code. Note that this function relies on the `random` function, which generates pseudo-random numbers. To make sure the execution is reproducible a seed value is given in the input. You should feed this value to the generator by calling the `srandom` function. You should not call the `random` function anywhere else in your code.

Usually whenever a function is called a specific string must be printed to the output. The messages for `Meld`, `link` and `randBit` are given above. For the remaining operations the messages are the following:

```
  printf("set A[%d] to %d\n", p, x);

  printf("root A[%d] is ", p);
  printf("A[%d]\n", p);

  printf("decKey A[%d] to %d\n", p, v);

  printf("min A[%d]\n", p);

  printf("extractMin A[%d]\n", p);

  printf("delete A[%d]\n", p);
```

The corresponding functions are indicated in the message text. The `root` function is used to move upwards from a given node until the corresponding root is found. Note that this function actually requires two messages,

one in the beginning near the function call and one at the end when the corresponding root is known.

The `Set` function is straightforward to implement. It will never be invoked in an invalid way but you can still check the necessary conditions with the `assert` function. The remaining functions are implemented as described by cite, however for completeness we review the necessary details here.

The `DecreaseKey` function first checks if the given node is the root of the tree, by looking at the `hook` pointer, not by calling the `root` function. If this is the case we simple change `v` value of the node and return it. Otherwise we call the `root` function to determine the root of this tree, then change the `v` value and split the sub-tree rooted at the argument node from its parent, by using the `hook` pointer. Finally we call meld with the root and the original node. Note that in the `Meld` operation the argument order matters and the root should be the first argument, i.e., Q_1. Consider for example the command `R 7 -1` applied to the configuration given in Figure 3. The result should be the configuration given in Figure 4 and this operation should give the following output:

```
decKey A[7] to -1
root A[7] is A[14]
Meld A[14] A[7]
Flipped a 1
Meld A[13] A[7]
Flipped a 0
Meld A[-1] A[7]
link A[7] as leftChild of A[13]
link A[13] as rightChild of A[14]
14
```

The `Min` function simply calls the `root` function and returns the respective `v` value. Likewise the `ArgMin` function basically consists of calling the `root` function.

The `ExtractMin` function also starts by calling the `root` function. It first sets the `v` value of the root to 0. Then it separates the left and right sub-trees of the root, if they exist. If neither of these sub-trees exist then it simply returns the index of the argument node. Otherwise if only one of them exists then it returns the index of the root of that sub-tree. Otherwise if both sub-trees exist, it calls the `Meld` operation where the left sub-tree is the first argument (Q_1) and the right sub-tree the second argument (Q_2). In this case the return is the root that results from the `Meld` operation.

The `Delete` operation first checks if the argument node is a root, again by reading the `hook` pointer. If this is the case then this operation reduces

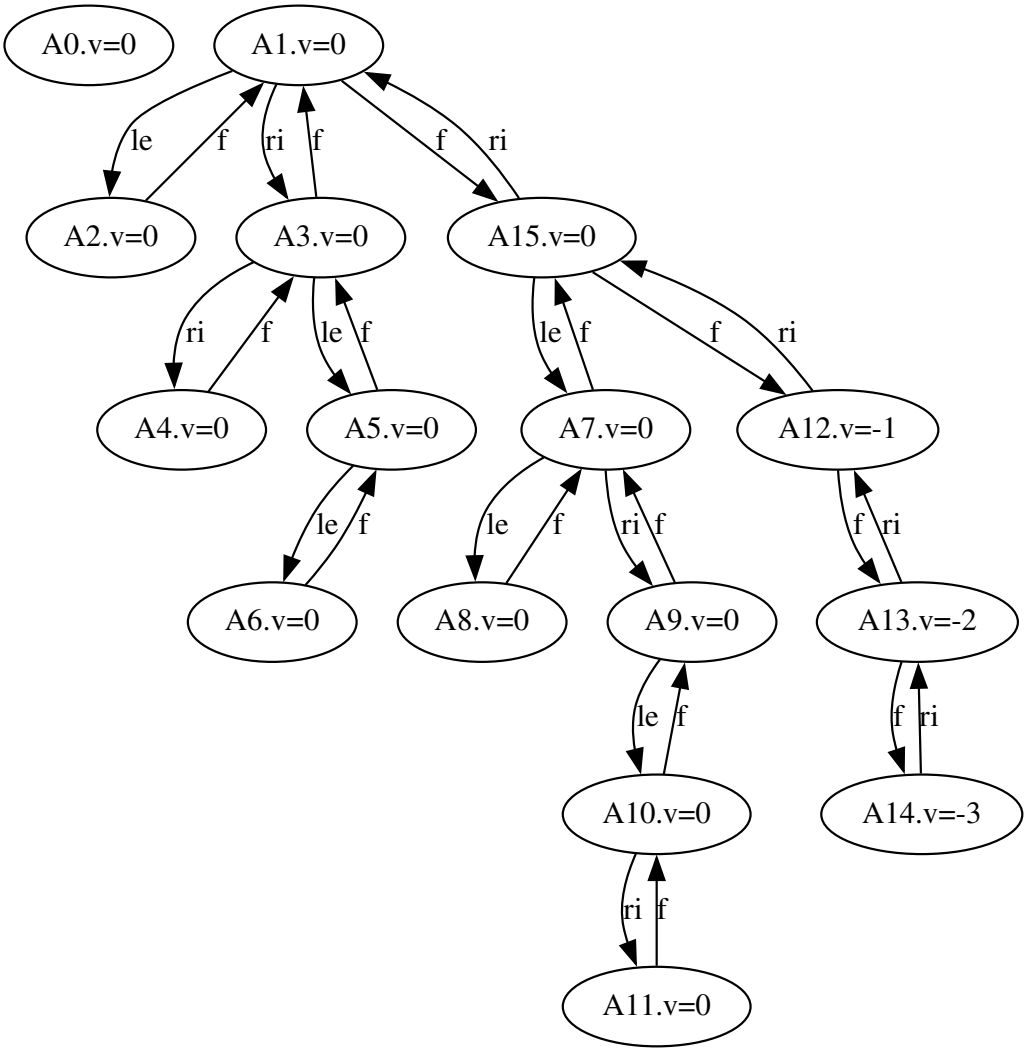

Figure 3: Representation before `DecreaseKey` operation.

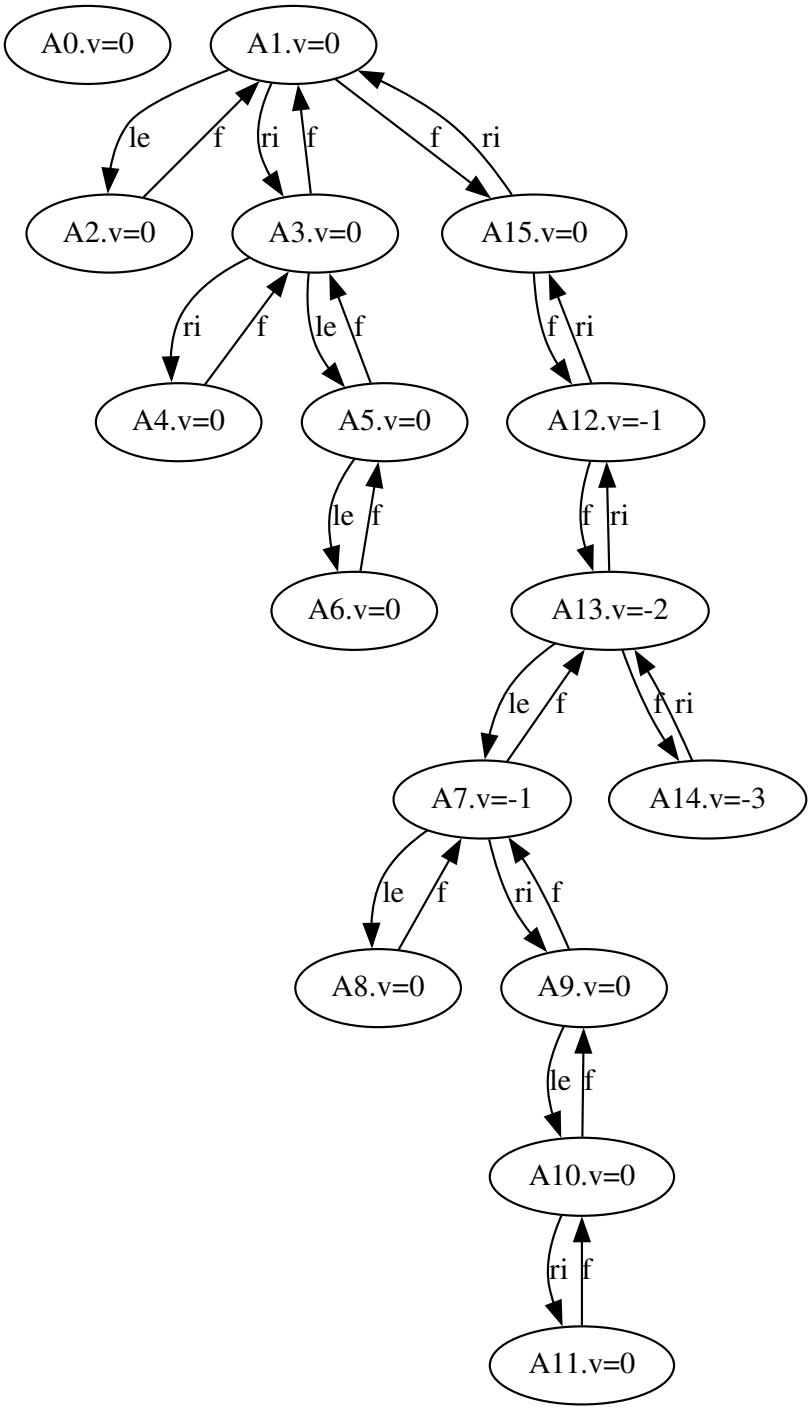

Figure 4: Representation after `DecreaseKey` operation.

to an `ExtractMin` operation on this root. Otherwise use the `root` function to identify the actual root node. Then cut the sub-tree rooted at the argument node from the larger tree. Next call `ExtractMin` for the small sub-tree rooted at the argument node. In case this sub-tree was a singular tree, i.e., contained only thee argument node then simply return the root identified before. Otherwise invoke the `Meld` operation where the root is used as the first argument and the sub-tree obtained from the `ExtractMin` operation is the second argument. In this case the return value is the result of the `Meld` operation.

Consider for example the command `D 1` applied to the configuration given in Figure 4. The result should be the configuration given in Figure 5 and this operation should give the following output:

```
delete A[1]
root A[1] is A[14]
extractMin A[1]
root A[1] is A[1]
Meld A[2] A[3]
Flipped a 0
Meld A[-1] A[3]
link A[3] as leftChild of A[2]
Meld A[14] A[2]
Flipped a 0
Meld A[-1] A[2]
link A[2] as leftChild of A[14]
14
```

1.2 Specification

To automatically validate the index we use the following conventions. The binary is executed with the following command:

`./project < in > out`

The file `in` contains the input commands that we will describe next. The output is stored in a file named `out`. The input and output must respect the specification below precisely. Note that your program should **not** open or close files, instead it should read information from `stdin` and write to `stdout`. The output file will be validated against an expected result, stored in a file named `check`, with the following command:

`diff out check`

This command should produce no output, thus indicating that both files are identical.

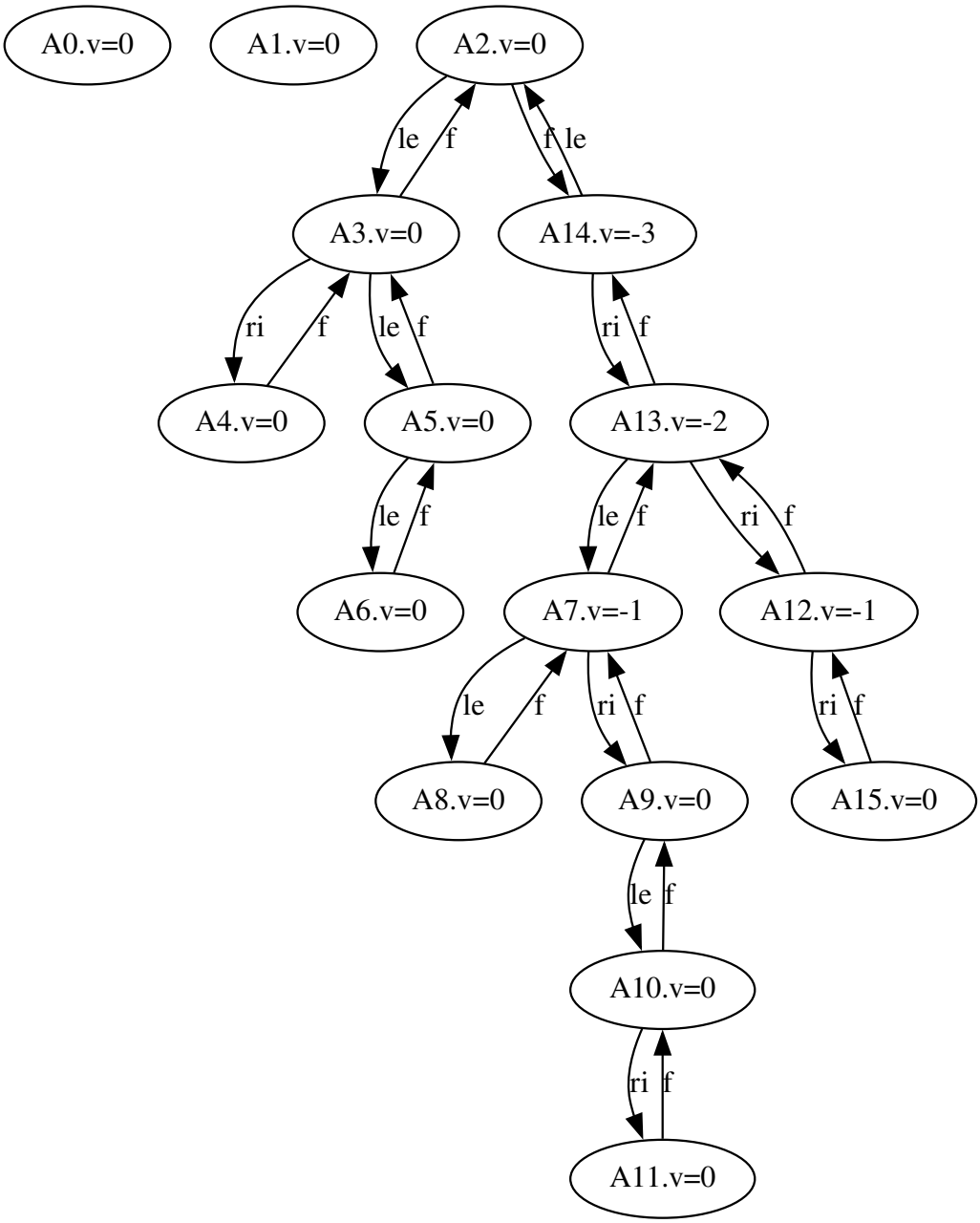

Figure 5: Representation after `Delete` operation.

The format of the input file is the following. The first line contains a single integer `n`, which is the number of nodes that should be allocated to the array `A`.

The rest of the input consists in a sequence of commands. Each command consists in a letter that indicates the command to execute followed by the respective arguments, separated by spaces. Except for the `P`, `S`, `V` and `U` commands the other commands should print their return value. The commands should print the information indicated above. The command `U` and the commands that depend on `Meld` also need to print information when linking nodes.

The `X` terminates the execution of the program. Hence no other commands are processed after this command is found. Before terminating the program should print the message ``` ``Final configuration:\n'' ``` followed by a list of lines that give the output of the `showNode` commands applied to all the indexes of `A` in increasing order. Recall to free the array `A`.

1.3 Sample Behaviour

The following examples show the expected `output` for the given `input`. These files are available on the course web page.

input 1

```
16 6166
S 1
P 1
V 1 12
S 1
R 1 11
S 1
E 1
S 1
V 1 10
S 1
D 1
S 1
V 13 -2
S 13
V 14 -2
S 14
V 14 -3
```

```
S 14
U 1 2
U 3 4
U 5 6
U 3 5
U 1 3
U 0 1
P 0
U 7 8
U 9 10
U 11 12
U 9 11
U 7 9
U 15 7
P 15
S 12
R 12 0
S 12
R 12 -1
P 12
U 0 12
P 12
M 0
A 0
U 13 12
P 13
M 13
A 13
U 14 13
P 14
M 14
A 14
D 0
P 14
R 7 -1
D 1
X
```

output 1

```
node: 1 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 1 v: 0 leftChild: -1 rightChild: -1 father: r -1
set A[1] to 12
node: 1 v: 12 leftChild: -1 rightChild: -1 father: r -1
decKey A[1] to 11
1
node: 1 v: 11 leftChild: -1 rightChild: -1 father: r -1
extractMin A[1]
root A[1] is A[1]
1
node: 1 v: 0 leftChild: -1 rightChild: -1 father: r -1
set A[1] to 10
node: 1 v: 10 leftChild: -1 rightChild: -1 father: r -1
delete A[1]
extractMin A[1]
root A[1] is A[1]
1
node: 1 v: 0 leftChild: -1 rightChild: -1 father: r -1
set A[13] to -2
node: 13 v: -2 leftChild: -1 rightChild: -1 father: r -1
set A[14] to -2
node: 14 v: -2 leftChild: -1 rightChild: -1 father: r -1
set A[14] to -3
node: 14 v: -3 leftChild: -1 rightChild: -1 father: r -1
Meld A[1] A[2]
Flipped a 0
Meld A[-1] A[2]
link A[2] as leftChild of A[1]
1
Meld A[3] A[4]
Flipped a 1
Meld A[-1] A[4]
link A[4] as rightChild of A[3]
3
Meld A[5] A[6]
Flipped a 0
Meld A[-1] A[6]
link A[6] as leftChild of A[5]
5
```

```
Meld A[3] A[5]
Flipped a 0
Meld A[-1] A[5]
link A[5] as leftChild of A[3]
3
Meld A[1] A[3]
Flipped a 1
Meld A[-1] A[3]
link A[3] as rightChild of A[1]
1
Meld A[0] A[1]
Flipped a 1
Meld A[-1] A[1]
link A[1] as rightChild of A[0]
0
node: 0 v: 0 leftChild: -1 rightChild: 1 father: r -1
node: 1 v: 0 leftChild: 2 rightChild: 3 father: r 0
node: 2 v: 0 leftChild: -1 rightChild: -1 father: l 1
node: 3 v: 0 leftChild: 5 rightChild: 4 father: r 1
node: 5 v: 0 leftChild: 6 rightChild: -1 father: l 3
node: 6 v: 0 leftChild: -1 rightChild: -1 father: l 5
node: 4 v: 0 leftChild: -1 rightChild: -1 father: r 3
Meld A[7] A[8]
Flipped a 0
Meld A[-1] A[8]
link A[8] as leftChild of A[7]
7
Meld A[9] A[10]
Flipped a 0
Meld A[-1] A[10]
link A[10] as leftChild of A[9]
9
Meld A[11] A[12]
Flipped a 0
Meld A[-1] A[12]
link A[12] as leftChild of A[11]
11
Meld A[9] A[11]
Flipped a 0
Meld A[10] A[11]
Flipped a 1
```

```
Meld A[-1] A[11]
link A[11] as rightChild of A[10]
link A[10] as leftChild of A[9]
9
Meld A[7] A[9]
Flipped a 1
Meld A[-1] A[9]
link A[9] as rightChild of A[7]
7
Meld A[15] A[7]
Flipped a 0
Meld A[-1] A[7]
link A[7] as leftChild of A[15]
15
node: 15 v: 0 leftChild: 7 rightChild: -1 father: r -1
node: 7 v: 0 leftChild: 8 rightChild: 9 father: l 15
node: 8 v: 0 leftChild: -1 rightChild: -1 father: l 7
node: 9 v: 0 leftChild: 10 rightChild: -1 father: r 7
node: 10 v: 0 leftChild: -1 rightChild: 11 father: l 9
node: 11 v: 0 leftChild: 12 rightChild: -1 father: r 10
node: 12 v: 0 leftChild: -1 rightChild: -1 father: l 11
node: 12 v: 0 leftChild: -1 rightChild: -1 father: l 11
decKey A[12] to 0
root A[12] is A[15]
Meld A[15] A[12]
Flipped a 1
Meld A[-1] A[12]
link A[12] as rightChild of A[15]
15
node: 12 v: 0 leftChild: -1 rightChild: -1 father: r 15
decKey A[12] to -1
root A[12] is A[15]
Meld A[15] A[12]
Swap to A[12] A[15]
Flipped a 1
Meld A[-1] A[15]
link A[15] as rightChild of A[12]
12
node: 12 v: -1 leftChild: -1 rightChild: 15 father: r -1
node: 15 v: 0 leftChild: 7 rightChild: -1 father: r 12
node: 7 v: 0 leftChild: 8 rightChild: 9 father: l 15
```

```
node: 8 v: 0 leftChild: -1 rightChild: -1 father: l 7
node: 9 v: 0 leftChild: 10 rightChild: -1 father: r 7
node: 10 v: 0 leftChild: -1 rightChild: 11 father: l 9
node: 11 v: 0 leftChild: -1 rightChild: -1 father: r 10
Meld A[0] A[12]
Swap to A[12] A[0]
Flipped a 1
Meld A[15] A[0]
Flipped a 1
Meld A[-1] A[0]
link A[0] as rightChild of A[15]
link A[15] as rightChild of A[12]
12
node: 12 v: -1 leftChild: -1 rightChild: 15 father: r -1
node: 15 v: 0 leftChild: 7 rightChild: 0 father: r 12
node: 7 v: 0 leftChild: 8 rightChild: 9 father: l 15
node: 8 v: 0 leftChild: -1 rightChild: -1 father: l 7
node: 9 v: 0 leftChild: 10 rightChild: -1 father: r 7
node: 10 v: 0 leftChild: -1 rightChild: 11 father: l 9
node: 11 v: 0 leftChild: -1 rightChild: -1 father: r 10
node: 0 v: 0 leftChild: -1 rightChild: 1 father: r 15
node: 1 v: 0 leftChild: 2 rightChild: 3 father: r 0
node: 2 v: 0 leftChild: -1 rightChild: -1 father: l 1
node: 3 v: 0 leftChild: 5 rightChild: 4 father: r 1
node: 5 v: 0 leftChild: 6 rightChild: -1 father: l 3
node: 6 v: 0 leftChild: -1 rightChild: -1 father: l 5
node: 4 v: 0 leftChild: -1 rightChild: -1 father: r 3
min A[0]
root A[0] is A[12]
-1
root A[0] is A[12]
12
Meld A[13] A[12]
Flipped a 1
Meld A[-1] A[12]
link A[12] as rightChild of A[13]
13
node: 13 v: -2 leftChild: -1 rightChild: 12 father: r -1
node: 12 v: -1 leftChild: -1 rightChild: 15 father: r 13
node: 15 v: 0 leftChild: 7 rightChild: 0 father: r 12
node: 7 v: 0 leftChild: 8 rightChild: 9 father: l 15
```

```
node: 8 v: 0 leftChild: -1 rightChild: -1 father: l 7
node: 9 v: 0 leftChild: 10 rightChild: -1 father: r 7
node: 10 v: 0 leftChild: -1 rightChild: 11 father: l 9
node: 11 v: 0 leftChild: -1 rightChild: -1 father: r 10
node: 0 v: 0 leftChild: -1 rightChild: 1 father: r 15
node: 1 v: 0 leftChild: 2 rightChild: 3 father: r 0
node: 2 v: 0 leftChild: -1 rightChild: -1 father: l 1
node: 3 v: 0 leftChild: 5 rightChild: 4 father: r 1
node: 5 v: 0 leftChild: 6 rightChild: -1 father: l 3
node: 6 v: 0 leftChild: -1 rightChild: -1 father: l 5
node: 4 v: 0 leftChild: -1 rightChild: -1 father: r 3
min A[13]
root A[13] is A[13]
-2
root A[13] is A[13]
13
Meld A[14] A[13]
Flipped a 1
Meld A[-1] A[13]
link A[13] as rightChild of A[14]
14
node: 14 v: -3 leftChild: -1 rightChild: 13 father: r -1
node: 13 v: -2 leftChild: -1 rightChild: 12 father: r 14
node: 12 v: -1 leftChild: -1 rightChild: 15 father: r 13
node: 15 v: 0 leftChild: 7 rightChild: 0 father: r 12
node: 7 v: 0 leftChild: 8 rightChild: 9 father: l 15
node: 8 v: 0 leftChild: -1 rightChild: -1 father: l 7
node: 9 v: 0 leftChild: 10 rightChild: -1 father: r 7
node: 10 v: 0 leftChild: -1 rightChild: 11 father: l 9
node: 11 v: 0 leftChild: -1 rightChild: -1 father: r 10
node: 0 v: 0 leftChild: -1 rightChild: 1 father: r 15
node: 1 v: 0 leftChild: 2 rightChild: 3 father: r 0
node: 2 v: 0 leftChild: -1 rightChild: -1 father: l 1
node: 3 v: 0 leftChild: 5 rightChild: 4 father: r 1
node: 5 v: 0 leftChild: 6 rightChild: -1 father: l 3
node: 6 v: 0 leftChild: -1 rightChild: -1 father: l 5
node: 4 v: 0 leftChild: -1 rightChild: -1 father: r 3
min A[14]
root A[14] is A[14]
-3
root A[14] is A[14]
```

```
14
delete A[0]
root A[0] is A[14]
extractMin A[0]
root A[0] is A[0]
Meld A[14] A[1]
Flipped a 1
Meld A[13] A[1]
Flipped a 1
Meld A[12] A[1]
Flipped a 1
Meld A[15] A[1]
Flipped a 1
Meld A[-1] A[1]
link A[1] as rightChild of A[15]
link A[15] as rightChild of A[12]
link A[12] as rightChild of A[13]
link A[13] as rightChild of A[14]
14
node: 14 v: -3 leftChild: -1 rightChild: 13 father: r -1
node: 13 v: -2 leftChild: -1 rightChild: 12 father: r 14
node: 12 v: -1 leftChild: -1 rightChild: 15 father: r 13
node: 15 v: 0 leftChild: 7 rightChild: 1 father: r 12
node: 7 v: 0 leftChild: 8 rightChild: 9 father: l 15
node: 8 v: 0 leftChild: -1 rightChild: -1 father: l 7
node: 9 v: 0 leftChild: 10 rightChild: -1 father: r 7
node: 10 v: 0 leftChild: -1 rightChild: 11 father: l 9
node: 11 v: 0 leftChild: -1 rightChild: -1 father: r 10
node: 1 v: 0 leftChild: 2 rightChild: 3 father: r 15
node: 2 v: 0 leftChild: -1 rightChild: -1 father: l 1
node: 3 v: 0 leftChild: 5 rightChild: 4 father: r 1
node: 5 v: 0 leftChild: 6 rightChild: -1 father: l 3
node: 6 v: 0 leftChild: -1 rightChild: -1 father: l 5
node: 4 v: 0 leftChild: -1 rightChild: -1 father: r 3
decKey A[7] to -1
root A[7] is A[14]
Meld A[14] A[7]
Flipped a 1
Meld A[13] A[7]
Flipped a 0
Meld A[-1] A[7]
```

```
link A[7] as leftChild of A[13]
link A[13] as rightChild of A[14]
14
delete A[1]
root A[1] is A[14]
extractMin A[1]
root A[1] is A[1]
Meld A[2] A[3]
Flipped a 0
Meld A[-1] A[3]
link A[3] as leftChild of A[2]
Meld A[14] A[2]
Flipped a 0
Meld A[-1] A[2]
link A[2] as leftChild of A[14]
14
Final configuration:
node: 0 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 1 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 2 v: 0 leftChild: 3 rightChild: -1 father: l 14
node: 3 v: 0 leftChild: 5 rightChild: 4 father: l 2
node: 4 v: 0 leftChild: -1 rightChild: -1 father: r 3
node: 5 v: 0 leftChild: 6 rightChild: -1 father: l 3
node: 6 v: 0 leftChild: -1 rightChild: -1 father: l 5
node: 7 v: -1 leftChild: 8 rightChild: 9 father: l 13
node: 8 v: 0 leftChild: -1 rightChild: -1 father: l 7
node: 9 v: 0 leftChild: 10 rightChild: -1 father: r 7
node: 10 v: 0 leftChild: -1 rightChild: 11 father: l 9
node: 11 v: 0 leftChild: -1 rightChild: -1 father: r 10
node: 12 v: -1 leftChild: -1 rightChild: 15 father: r 13
node: 13 v: -2 leftChild: 7 rightChild: 12 father: r 14
node: 14 v: -3 leftChild: 2 rightChild: 13 father: r -1
node: 15 v: 0 leftChild: -1 rightChild: -1 father: r 12
```

input 2

```
16 6166
V 1 1
V 2 2
V 3 3
V 4 4
```

```
V 5 5
V 6 6
V 7 7
V 8 8
V 9 9
V 10 10
V 11 11
V 12 12
V 13 13
V 14 14
V 15 15
U 0 1
U 0 2
U 0 3
U 0 4
U 0 5
U 0 6
U 0 7
U 0 8
U 0 9
U 0 10
U 0 11
U 0 12
U 0 13
U 0 14
P 0
X
```

output 2

```
set A[1] to 1
set A[2] to 2
set A[3] to 3
set A[4] to 4
set A[5] to 5
set A[6] to 6
set A[7] to 7
set A[8] to 8
set A[9] to 9
set A[10] to 10
set A[11] to 11
```

```
set A[12] to 12
set A[13] to 13
set A[14] to 14
set A[15] to 15
Meld A[0] A[1]
Flipped a 0
Meld A[-1] A[1]
link A[1] as leftChild of A[0]
0
Meld A[0] A[2]
Flipped a 1
Meld A[-1] A[2]
link A[2] as rightChild of A[0]
0
Meld A[0] A[3]
Flipped a 0
Meld A[1] A[3]
Flipped a 0
Meld A[-1] A[3]
link A[3] as leftChild of A[1]
link A[1] as leftChild of A[0]
0
Meld A[0] A[4]
Flipped a 1
Meld A[2] A[4]
Flipped a 1
Meld A[-1] A[4]
link A[4] as rightChild of A[2]
link A[2] as rightChild of A[0]
0
Meld A[0] A[5]
Flipped a 0
Meld A[1] A[5]
Flipped a 0
Meld A[3] A[5]
Flipped a 0
Meld A[-1] A[5]
link A[5] as leftChild of A[3]
link A[3] as leftChild of A[1]
link A[1] as leftChild of A[0]
0
```

```
Meld A[0] A[6]
Flipped a 0
Meld A[1] A[6]
Flipped a 1
Meld A[-1] A[6]
link A[6] as rightChild of A[1]
link A[1] as leftChild of A[0]
0
Meld A[0] A[7]
Flipped a 1
Meld A[2] A[7]
Flipped a 0
Meld A[-1] A[7]
link A[7] as leftChild of A[2]
link A[2] as rightChild of A[0]
0
Meld A[0] A[8]
Flipped a 1
Meld A[2] A[8]
Flipped a 1
Meld A[4] A[8]
Flipped a 1
Meld A[-1] A[8]
link A[8] as rightChild of A[4]
link A[4] as rightChild of A[2]
link A[2] as rightChild of A[0]
0
Meld A[0] A[9]
Flipped a 1
Meld A[2] A[9]
Flipped a 1
Meld A[4] A[9]
Flipped a 1
Meld A[8] A[9]
Flipped a 1
Meld A[-1] A[9]
link A[9] as rightChild of A[8]
link A[8] as rightChild of A[4]
link A[4] as rightChild of A[2]
link A[2] as rightChild of A[0]
0
```

```
Meld A[0] A[10]
Flipped a 1
Meld A[2] A[10]
Flipped a 1
Meld A[4] A[10]
Flipped a 1
Meld A[8] A[10]
Flipped a 1
Meld A[9] A[10]
Flipped a 0
Meld A[-1] A[10]
link A[10] as leftChild of A[9]
link A[9] as rightChild of A[8]
link A[8] as rightChild of A[4]
link A[4] as rightChild of A[2]
link A[2] as rightChild of A[0]
0
Meld A[0] A[11]
Flipped a 0
Meld A[1] A[11]
Flipped a 0
Meld A[3] A[11]
Flipped a 0
Meld A[5] A[11]
Flipped a 0
Meld A[-1] A[11]
link A[11] as leftChild of A[5]
link A[5] as leftChild of A[3]
link A[3] as leftChild of A[1]
link A[1] as leftChild of A[0]
0
Meld A[0] A[12]
Flipped a 1
Meld A[2] A[12]
Flipped a 1
Meld A[4] A[12]
Flipped a 0
Meld A[-1] A[12]
link A[12] as leftChild of A[4]
link A[4] as rightChild of A[2]
link A[2] as rightChild of A[0]
```

```
0
Meld A[0] A[13]
Flipped a 1
Meld A[2] A[13]
Flipped a 0
Meld A[7] A[13]
Flipped a 0
Meld A[-1] A[13]
link A[13] as leftChild of A[7]
link A[7] as leftChild of A[2]
link A[2] as rightChild of A[0]
0
Meld A[0] A[14]
Flipped a 1
Meld A[2] A[14]
Flipped a 0
Meld A[7] A[14]
Flipped a 0
Meld A[13] A[14]
Flipped a 0
Meld A[-1] A[14]
link A[14] as leftChild of A[13]
link A[13] as leftChild of A[7]
link A[7] as leftChild of A[2]
link A[2] as rightChild of A[0]
0
node: 0 v: 0 leftChild: 1 rightChild: 2 father: r -1
node: 1 v: 1 leftChild: 3 rightChild: 6 father: l 0
node: 3 v: 3 leftChild: 5 rightChild: -1 father: l 1
node: 5 v: 5 leftChild: 11 rightChild: -1 father: l 3
node: 11 v: 11 leftChild: -1 rightChild: -1 father: l 5
node: 6 v: 6 leftChild: -1 rightChild: -1 father: r 1
node: 2 v: 2 leftChild: 7 rightChild: 4 father: r 0
node: 7 v: 7 leftChild: 13 rightChild: -1 father: l 2
node: 13 v: 13 leftChild: 14 rightChild: -1 father: l 7
node: 14 v: 14 leftChild: -1 rightChild: -1 father: l 13
node: 4 v: 4 leftChild: 12 rightChild: 8 father: r 2
node: 12 v: 12 leftChild: -1 rightChild: -1 father: l 4
node: 8 v: 8 leftChild: -1 rightChild: 9 father: r 4
node: 9 v: 9 leftChild: 10 rightChild: -1 father: r 8
node: 10 v: 10 leftChild: -1 rightChild: -1 father: l 9
```

```
Final configuration:
node: 0 v: 0 leftChild: 1 rightChild: 2 father: r -1
node: 1 v: 1 leftChild: 3 rightChild: 6 father: l 0
node: 2 v: 2 leftChild: 7 rightChild: 4 father: r 0
node: 3 v: 3 leftChild: 5 rightChild: -1 father: l 1
node: 4 v: 4 leftChild: 12 rightChild: 8 father: r 2
node: 5 v: 5 leftChild: 11 rightChild: -1 father: l 3
node: 6 v: 6 leftChild: -1 rightChild: -1 father: r 1
node: 7 v: 7 leftChild: 13 rightChild: -1 father: l 2
node: 8 v: 8 leftChild: -1 rightChild: 9 father: r 4
node: 9 v: 9 leftChild: 10 rightChild: -1 father: r 8
node: 10 v: 10 leftChild: -1 rightChild: -1 father: l 9
node: 11 v: 11 leftChild: -1 rightChild: -1 father: l 5
node: 12 v: 12 leftChild: -1 rightChild: -1 father: l 4
node: 13 v: 13 leftChild: 14 rightChild: -1 father: l 7
node: 14 v: 14 leftChild: -1 rightChild: -1 father: l 13
node: 15 v: 15 leftChild: -1 rightChild: -1 father: r -1
```

input 3

```
16 6166
V 1 1
V 2 2
V 3 3
V 4 4
V 5 5
V 6 6
V 7 7
V 8 8
V 9 9
V 10 10
V 11 11
V 12 12
V 13 13
V 14 14
V 15 15
U 15 14
U 13 14
U 12 13
U 11 12
U 10 11
```

```
U 9 10
U 8 9
U 7 8
U 6 7
U 5 6
U 4 5
U 3 4
U 2 3
U 1 2
U 0 1
P 0
A 0
A 1
A 2
A 3
A 4
A 5
A 6
A 7
A 8
A 9
A 10
A 11
A 12
A 13
A 14
A 15
X
```

output 3

```
set A[1] to 1
set A[2] to 2
set A[3] to 3
set A[4] to 4
set A[5] to 5
set A[6] to 6
set A[7] to 7
set A[8] to 8
set A[9] to 9
set A[10] to 10
```

```
set A[11] to 11
set A[12] to 12
set A[13] to 13
set A[14] to 14
set A[15] to 15
Meld A[15] A[14]
Swap to A[14] A[15]
Flipped a 0
Meld A[-1] A[15]
link A[15] as leftChild of A[14]
14
Meld A[13] A[14]
Flipped a 1
Meld A[-1] A[14]
link A[14] as rightChild of A[13]
13
Meld A[12] A[13]
Flipped a 0
Meld A[-1] A[13]
link A[13] as leftChild of A[12]
12
Meld A[11] A[12]
Flipped a 0
Meld A[-1] A[12]
link A[12] as leftChild of A[11]
11
Meld A[10] A[11]
Flipped a 1
Meld A[-1] A[11]
link A[11] as rightChild of A[10]
10
Meld A[9] A[10]
Flipped a 1
Meld A[-1] A[10]
link A[10] as rightChild of A[9]
9
Meld A[8] A[9]
Flipped a 0
Meld A[-1] A[9]
link A[9] as leftChild of A[8]
8
```

```
Meld A[7] A[8]
Flipped a 0
Meld A[-1] A[8]
link A[8] as leftChild of A[7]
7
Meld A[6] A[7]
Flipped a 0
Meld A[-1] A[7]
link A[7] as leftChild of A[6]
6
Meld A[5] A[6]
Flipped a 0
Meld A[-1] A[6]
link A[6] as leftChild of A[5]
5
Meld A[4] A[5]
Flipped a 1
Meld A[-1] A[5]
link A[5] as rightChild of A[4]
4
Meld A[3] A[4]
Flipped a 1
Meld A[-1] A[4]
link A[4] as rightChild of A[3]
3
Meld A[2] A[3]
Flipped a 0
Meld A[-1] A[3]
link A[3] as leftChild of A[2]
2
Meld A[1] A[2]
Flipped a 1
Meld A[-1] A[2]
link A[2] as rightChild of A[1]
1
Meld A[0] A[1]
Flipped a 1
Meld A[-1] A[1]
link A[1] as rightChild of A[0]
0
node: 0 v: 0 leftChild: -1 rightChild: 1 father: r -1
```

```
node: 1 v: 1 leftChild: -1 rightChild: 2 father: r 0
node: 2 v: 2 leftChild: 3 rightChild: -1 father: r 1
node: 3 v: 3 leftChild: -1 rightChild: 4 father: l 2
node: 4 v: 4 leftChild: -1 rightChild: 5 father: r 3
node: 5 v: 5 leftChild: 6 rightChild: -1 father: r 4
node: 6 v: 6 leftChild: 7 rightChild: -1 father: l 5
node: 7 v: 7 leftChild: 8 rightChild: -1 father: l 6
node: 8 v: 8 leftChild: 9 rightChild: -1 father: l 7
node: 9 v: 9 leftChild: -1 rightChild: 10 father: l 8
node: 10 v: 10 leftChild: -1 rightChild: 11 father: r 9
node: 11 v: 11 leftChild: 12 rightChild: -1 father: r 10
node: 12 v: 12 leftChild: 13 rightChild: -1 father: l 11
node: 13 v: 13 leftChild: -1 rightChild: 14 father: l 12
node: 14 v: 14 leftChild: 15 rightChild: -1 father: r 13
node: 15 v: 15 leftChild: -1 rightChild: -1 father: l 14
root A[0] is A[0]
0
root A[1] is A[0]
0
root A[2] is A[0]
0
root A[3] is A[0]
0
root A[4] is A[0]
0
root A[5] is A[0]
0
root A[6] is A[0]
0
root A[7] is A[0]
0
root A[8] is A[0]
0
root A[9] is A[0]
0
root A[10] is A[0]
0
root A[11] is A[0]
0
root A[12] is A[0]
0
```

```
root A[13] is A[0]
0
root A[14] is A[0]
0
root A[15] is A[0]
0
Final configuration:
node: 0 v: 0 leftChild: -1 rightChild: 1 father: r -1
node: 1 v: 1 leftChild: -1 rightChild: 2 father: r 0
node: 2 v: 2 leftChild: 3 rightChild: -1 father: r 1
node: 3 v: 3 leftChild: -1 rightChild: 4 father: l 2
node: 4 v: 4 leftChild: -1 rightChild: 5 father: r 3
node: 5 v: 5 leftChild: 6 rightChild: -1 father: r 4
node: 6 v: 6 leftChild: 7 rightChild: -1 father: l 5
node: 7 v: 7 leftChild: 8 rightChild: -1 father: l 6
node: 8 v: 8 leftChild: 9 rightChild: -1 father: l 7
node: 9 v: 9 leftChild: -1 rightChild: 10 father: l 8
node: 10 v: 10 leftChild: -1 rightChild: 11 father: r 9
node: 11 v: 11 leftChild: 12 rightChild: -1 father: r 10
node: 12 v: 12 leftChild: 13 rightChild: -1 father: l 11
node: 13 v: 13 leftChild: -1 rightChild: 14 father: l 12
node: 14 v: 14 leftChild: 15 rightChild: -1 father: r 13
node: 15 v: 15 leftChild: -1 rightChild: -1 father: l 14
```

input 4

```
16 6166
V 1 1
V 2 2
V 3 3
V 4 4
V 5 5
V 6 6
V 7 7
V 8 8
V 9 9
V 10 10
V 11 11
V 12 12
V 13 13
V 14 14
```

```
V 15 15
U 15 14
U 13 14
U 12 13
U 11 12
U 10 11
U 9 10
U 8 9
U 7 8
U 6 7
U 5 6
U 4 5
U 3 4
U 2 3
U 1 2
U 0 1
P 0
A 15
E 15
A 15
E 15
A 15
E 15
A 15
E 15
A 15
E 15
A 15
E 15
A 15
E 15
A 15
E 15
A 15
E 15
A 15
E 15
A 15
E 15
A 15
E 15
```

```
A 15
E 15
A 15
E 15
A 15
E 15
A 15
E 15
X
```

output 4

```
set A[1] to 1
set A[2] to 2
set A[3] to 3
set A[4] to 4
set A[5] to 5
set A[6] to 6
set A[7] to 7
set A[8] to 8
set A[9] to 9
set A[10] to 10
set A[11] to 11
set A[12] to 12
set A[13] to 13
set A[14] to 14
set A[15] to 15
Meld A[15] A[14]
Swap to A[14] A[15]
Flipped a 0
Meld A[-1] A[15]
link A[15] as leftChild of A[14]
14
Meld A[13] A[14]
Flipped a 1
Meld A[-1] A[14]
link A[14] as rightChild of A[13]
13
Meld A[12] A[13]
Flipped a 0
Meld A[-1] A[13]
```

```
link A[13] as leftChild of A[12]
12
Meld A[11] A[12]
Flipped a 0
Meld A[-1] A[12]
link A[12] as leftChild of A[11]
11
Meld A[10] A[11]
Flipped a 1
Meld A[-1] A[11]
link A[11] as rightChild of A[10]
10
Meld A[9] A[10]
Flipped a 1
Meld A[-1] A[10]
link A[10] as rightChild of A[9]
9
Meld A[8] A[9]
Flipped a 0
Meld A[-1] A[9]
link A[9] as leftChild of A[8]
8
Meld A[7] A[8]
Flipped a 0
Meld A[-1] A[8]
link A[8] as leftChild of A[7]
7
Meld A[6] A[7]
Flipped a 0
Meld A[-1] A[7]
link A[7] as leftChild of A[6]
6
Meld A[5] A[6]
Flipped a 0
Meld A[-1] A[6]
link A[6] as leftChild of A[5]
5
Meld A[4] A[5]
Flipped a 1
Meld A[-1] A[5]
link A[5] as rightChild of A[4]
```

```
4
Meld A[3] A[4]
Flipped a 1
Meld A[-1] A[4]
link A[4] as rightChild of A[3]
3
Meld A[2] A[3]
Flipped a 0
Meld A[-1] A[3]
link A[3] as leftChild of A[2]
2
Meld A[1] A[2]
Flipped a 1
Meld A[-1] A[2]
link A[2] as rightChild of A[1]
1
Meld A[0] A[1]
Flipped a 1
Meld A[-1] A[1]
link A[1] as rightChild of A[0]
0
node: 0 v: 0 leftChild: -1 rightChild: 1 father: r -1
node: 1 v: 1 leftChild: -1 rightChild: 2 father: r 0
node: 2 v: 2 leftChild: 3 rightChild: -1 father: r 1
node: 3 v: 3 leftChild: -1 rightChild: 4 father: l 2
node: 4 v: 4 leftChild: -1 rightChild: 5 father: r 3
node: 5 v: 5 leftChild: 6 rightChild: -1 father: r 4
node: 6 v: 6 leftChild: 7 rightChild: -1 father: l 5
node: 7 v: 7 leftChild: 8 rightChild: -1 father: l 6
node: 8 v: 8 leftChild: 9 rightChild: -1 father: l 7
node: 9 v: 9 leftChild: -1 rightChild: 10 father: l 8
node: 10 v: 10 leftChild: -1 rightChild: 11 father: r 9
node: 11 v: 11 leftChild: 12 rightChild: -1 father: r 10
node: 12 v: 12 leftChild: 13 rightChild: -1 father: l 11
node: 13 v: 13 leftChild: -1 rightChild: 14 father: l 12
node: 14 v: 14 leftChild: 15 rightChild: -1 father: r 13
node: 15 v: 15 leftChild: -1 rightChild: -1 father: l 14
root A[15] is A[0]
0
extractMin A[15]
root A[15] is A[0]
```

```
1
root A[15] is A[1]
1
extractMin A[15]
root A[15] is A[1]
2
root A[15] is A[2]
2
extractMin A[15]
root A[15] is A[2]
3
root A[15] is A[3]
3
extractMin A[15]
root A[15] is A[3]
4
root A[15] is A[4]
4
extractMin A[15]
root A[15] is A[4]
5
root A[15] is A[5]
5
extractMin A[15]
root A[15] is A[5]
6
root A[15] is A[6]
6
extractMin A[15]
root A[15] is A[6]
7
root A[15] is A[7]
7
extractMin A[15]
root A[15] is A[7]
8
root A[15] is A[8]
8
extractMin A[15]
root A[15] is A[8]
9
```

```
root A[15] is A[9]
9
extractMin A[15]
root A[15] is A[9]
10
root A[15] is A[10]
10
extractMin A[15]
root A[15] is A[10]
11
root A[15] is A[11]
11
extractMin A[15]
root A[15] is A[11]
12
root A[15] is A[12]
12
extractMin A[15]
root A[15] is A[12]
13
root A[15] is A[13]
13
extractMin A[15]
root A[15] is A[13]
14
root A[15] is A[14]
14
extractMin A[15]
root A[15] is A[14]
15
root A[15] is A[15]
15
extractMin A[15]
root A[15] is A[15]
15
Final configuration:
node: 0 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 1 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 2 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 3 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 4 v: 0 leftChild: -1 rightChild: -1 father: r -1
```

```
node: 5 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 6 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 7 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 8 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 9 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 10 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 11 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 12 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 13 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 14 v: 0 leftChild: -1 rightChild: -1 father: r -1
node: 15 v: 0 leftChild: -1 rightChild: -1 father: r -1
```

2 Grading

The mooshak system is configured to a total 40 points. The project accounts for 4.0 values of the final grade. Hence to obtain the contribution of the project to the final grade divide the number of points by 10. To obtain a grading in an absolute scale to 20 divide the number of points by 2.

Each test has a specific set of points. The first four tests correspond to the input output examples given in this script. These tests are public and will be returned back by the system. The tests numbered from 5 to 12 correspond to increasingly harder test cases, brief descriptions are given by the system. Tests 13 and 14 are verified by the valgrind[1] tool. Test 13 checks for the condition `ERROR SUMMARY: 0 errors from 0 contexts` and test 14 for the condition `All heap blocks were freed -- no leaks are possible`. Test 15 to 17 are verified by the lizzard[2] tool, the test passes if the `No thresholds exceeded` message is given. Test 15 uses the arguments `-T cyclomatic_complexity=15`; test 16 the argument `-T length=150`; test 17 the argument `-T parameter_count=9 -T token_count=500`. To obtain the score of tests from 13 to 17 must it is necessary obtain the correct output, besides the conditions just described.

The mooshak system accepts the C programming language, click on `Help` button for the respective compiler. Projects that do not compile in the mooshak system will be graded 0. Only the code that compiles in the mooshak system will be considered, commented code, will not be considered for evaluation.

Submissions to the mooshak system should consist of a single file. The system identifies the language through the file extension, an extension `.c`

[1] `https://www.valgrind.org/`

[2] `https://github.com/terryyin/lizard`

means the C language. The compilation process should produce absolutely no errors or warnings, otherwise the file will not compile. The resulting binary should behave exactly as explained in the specification section. Be mindful that `diff` will produce output even if a single character is different, such as a space or a newline.

Notice that you can submit to mooshak several times, but there is a 10 minute waiting period before submissions. You are strongly advised to submit several times and as early as possible. Only the last version is considered for grading purposes, all other submissions are ignored. There will be **no** deadline extensions. Submissions by email will **not** be accepted.

3 Debugging Suggestions

There are several tools that can be used to help in debugging your project implementation. For very a simple verification a carefully placed `printf` command can prove most useful. Likewise it is also considered good practice to use the `assert` command to have your program automatically verify certain desirable properties. The flag `-D NDEBUG` was added to the gcc command of mooshak. This means that you may submit your code without needing to remove the `assert` commands, as they are removed by the pre-processor. Also if you wish to include code that gets automatically removed from the submission you can use `#ifndef NDEBUG`. Here is a simple example:

```
#ifndef NDEBUG
  /* structLoad(); */
#endif /* NDEBUG */
```

The following functions may also prove helpful.

```
void
vizShow(FILE *f, int n)
{
  int i;

  fprintf(f, "digraph {\n");
  for(i = 0; i < n; i++){
    fprintf(f, "A%d [label=\"A%d.v=%d\"]\n",
            i, i, A[i].v);
  }
  for( i = 0; i < n; i++){
    if(NULL != A[i].leftChild)
```

```
      fprintf(f, "A%d -> A%d [label=\"le\"]\n",
              i, ptr2loc(A[i].leftChild, A));
    if(NULL != A[i].rightChild)
      fprintf(f, "A%d -> A%d [label=\"ri\"]\n",
              i, ptr2loc(A[i].rightChild, A));
    if(NULL != A[i].hook)
      fprintf(f, "A%d -> A%d [label=\"f\"]\n",
              i, ptr2loc((node)(A[i].hook), A));
  }
  fprintf(f, "}\n");
}

void
structLoad(void)
{
  FILE *f;
  int i;
  int j;
  char c;

  f = fopen("load.txt", "r");

  if(NULL != f){
    size_t len = 1<<8;
    char *line = (char*)malloc(len*sizeof(char));

    while(-1 != getline(&line, &len, f)){
      char *tok;

      tok = strtok(line, " ");
      tok = strtok(NULL, " ");
      sscanf(tok, "%d", &i);

      tok = strtok(NULL, " ");
      tok = strtok(NULL, " ");
      sscanf(tok, "%d", &A[i].v);

      tok = strtok(NULL, " ");
      tok = strtok(NULL, " ");
      sscanf(tok, "%d", &j);
      A[i].leftChild = NULL;
```

```
      if(-1 != j)
        A[i].leftChild = &A[j];

      tok = strtok(NULL, " ");
      tok = strtok(NULL, " ");
      sscanf(tok, "%d", &j);
      A[i].rightChild = NULL;
      if(-1 != j)
        A[i].rightChild = &A[j];

      tok = strtok(NULL, " ");
      tok = strtok(NULL, " ");
      sscanf(tok, "%c", &c);
      tok = strtok(NULL, " ");
      sscanf(tok, "%d", &j);
      A[i].hook = NULL;
      if(-1 != j){
        if('l' == c)
          A[i].hook = &A[j].leftChild;
        else
          A[i].hook = &A[j].rightChild;
      }
    }

    free(line);
    fclose(f);
  }
}
```

The `vizShow` function produces a description of the current state of your data structure in the `dot` language, see `https://graphviz.org/`. This function can be invoked with the following snipped of code:

```
FILE *f = fopen("dotAF", "w");
vizShow(f, n);
fclose(f);
```

To produce a `pdf` file with the corresponding image you may use the command `dot -O -Tpdf dotAF`.

The `structLoad` function can be used to load a configuration directly into the array `A`, without having to specify a sequence of commands that

leads to that configuration. This way a configuration specification can be stored in the file `load.txt`.

For more complex debugging sessions it may be necessary to use a debugger, such as `gdb`, see `https://www.sourceware.org/gdb/`

The use of the `valgrind` tool, for memory verification is also highly recommended, see `https://valgrind.org/`

References

Anna Gambin and Adam Malinowski. Randomized meldable priority queues. In *25th Conference on Current Trends in Theory and Practice of Informatics, Jasná, Slovakia, November 21-27*, volume 1521 of *Lecture Notes in Computer Science*, pages 344–349. Springer, 1998. URL `https://doi.org/10.1007/3-540-49477-4\_26`.