TÉCNICO LISBOA

DEI
Departamento de Engenharia Informática

Advanced Algorithms

2023/2024

Project 1

Implement a Pairing Heap

This project considers the problem of constructing a pairing heap. The first delivery deadline for the project code is the 28th of February, at 12:00. The deadline for the recovery season is 5th July at 12:00, the system opens the 3rd of July at 12:00. The deadline for the special season is 20th July at 12:00, the system opens the 18th 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 Pairing Heaps

The challenge is to implement the pairing heap data structure. A description of this data structure was given by **?**. 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 v > 0.
            a value of v < 0 marks end of brother list. */
  node child; /* child pointer */
  node brother; /* brother or father if @ the end of list. */
};
```

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 1, until it is changed by a `Set` command. If this value is positive then it is the actual value and moreover this node is not at the end of its `brother` list. If this value is negative the actual value is its positive correspondent and moreover this node is the last one of the `brother` list. Note that this limits the values of v to always be positive numbers, hence this heap implementation does not support inserting non-positive numbers.

The `child` field points to the first child of the node. If the node does not have a children then this pointer should be set to `NULL`. Finally the `brother` field points to the brother of the current `node` when the `v` value is positive. If the `v` value is negative then the `brother` field actually points to the father of the node. The `brother` field can only be `NULL` when the current node is the root of the corresponding heap, in which case the value of `v` should be negative.

The `ExtracMin` operation should reset a `node` to its default initial state. Meaning that the field `v` should be set to -1 and the `child` and `brother` 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("child: %d ", ptr2loc(v->child, A));
    printf("brother: %d ", ptr2loc(v->brother, A));
    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 `showList` function that gives a description of the `brother` list at a node, i.e., it calls `showNode` for the current `node` and then recursively calls `showList` on the brother node if the current `v` value is not negative.

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 `child` and `brother` fields are all `NULL`. The function must receive a positive value `v`, but stores its negative correspondent.

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 `(root, 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. The `root` of the corresponding tree must be given as the first argument to this function.

M `(node)` The `Min` function returns the absolute value of `v` for the current node. When the argument `node` is a root the result is the heap minimum value.

E `(root)` The `ExtractMin` function removes the `root` node from the current heap. The function returns the, possibly new, 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 pointer fields are set to `NULL`. However the `v` fields must be manually set to `-1`.

In this initial configuration the `node` in number 0 is a tree and therefore also a pairing heap. In fact any index i represents a different 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 **??**. Each node contains its index in `A` and its `v` value. In this case the `v` values are mostly 1, except for `A[1]`, `A[13]` and `A[8]`. These heaps will occur in the first input given below. The arrows labeled `ch` represent the `child` pointers, the ones labeled `br` the `brother` pointers and the arrows labeled `f` the `brother` pointers when the `v` value is negative. Note that node 2 is the root of one tree and node 15 is the root of another tree. Now consider the command `U 2 15`, which will meld these two heaps.

Before executing this command we can execute the `P 15` command to obtain the description of node 15. We also execute the command `S 12` that shows a list that contains only node `12`.

Let us consider the execution of the command `U 2 15`. The output produced by this command is the following :

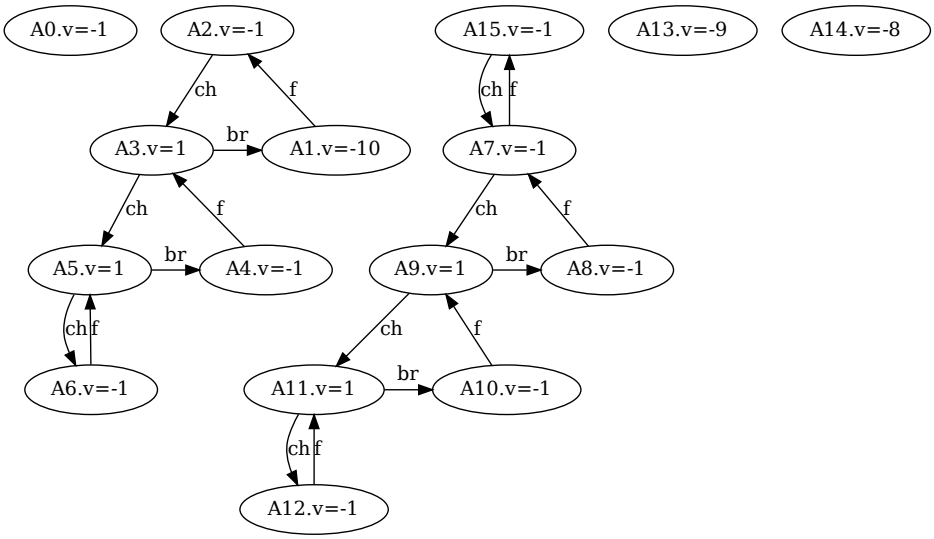

Figure 1: Representation of four meldable heap trees before `Meld` operation.

```
Meld A[2] A[15]
link A[15] as child of A[2]
2
```

In this particular call we have use 2 as the first argument to the `Meld` function and 15 as the second argument. Since both nodes are storing a represented value of 1 there is no point in swapping this other and thus node 2 becomes the father of node 15. The return value is node 2 as this is the root of the new tree. It very important to notice when both the represented values are equal swaps never occurs, i.e., in case of a tie there is no swap.

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 A[%d] and A[%d]\n", q1, q2);
```

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

```
static void
link(node f, node c)
```

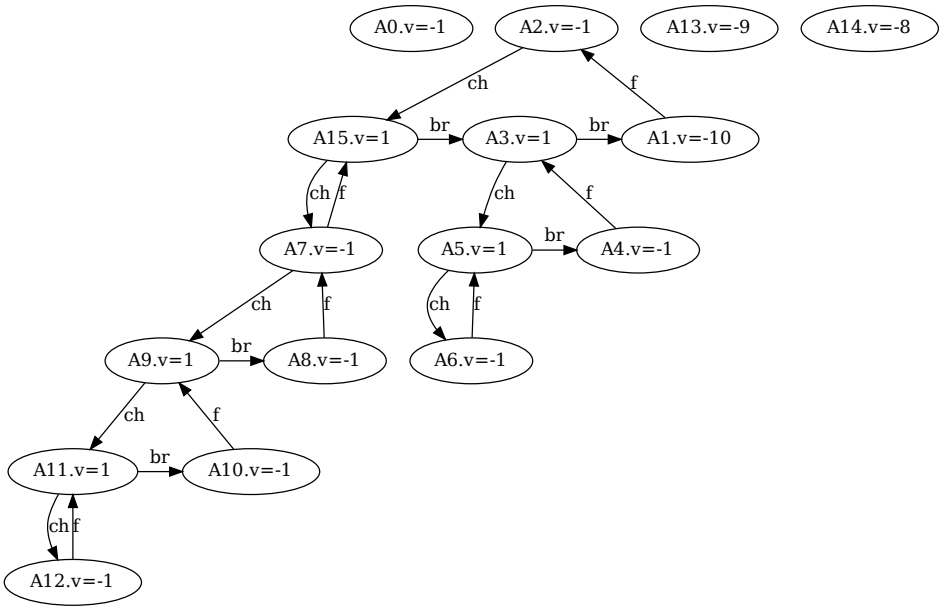

Figure 2: Representation after `Meld` operation.

```
{
  printf("link A[%d] as child of A[%d]\n",
         ptr2loc(c, A), ptr2loc(f, A));
```

The arguments of the function correspond to the father node `f`, the child node `c`.

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

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

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

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

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

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 **?**, however for completeness we review the necessary details here.

The `DecreaseKey` function assumes that the first argument is the root of the tree that contains the second argument. Moreover it also assumes that the argument value of `v` is indeed smaller that the represented value in the argument node. If both the node and root arguments are equal then we are decreasing the value at the root, which is a simple matter of changing the value. Otherwise the implementation relies on the `getHook` function which determines how the node is connected, i.e., which other node points to it and is it a `child` or `brother` pointer. The respective pointer is altered, so that the sub-tree rooted at the node becomes separate from the original tree. Then the `v` value is decreased and finally this new tree is melded back with the root of the original tree. In calling the `Meld` function the original root is used as first argument and the node (now also a root) as the second argument. In both cases the return value is the root of the resulting heap tree.

An important detail to consider is when the argument node is the only `child` of its parent node. In this case the respective `child` pointer should be set to `NULL`.

Consider for example the command `R 2 1 7` applied to the configuration given in Figure **??**. The result should be the configuration given in Figure **??** and this operation should give the following output:

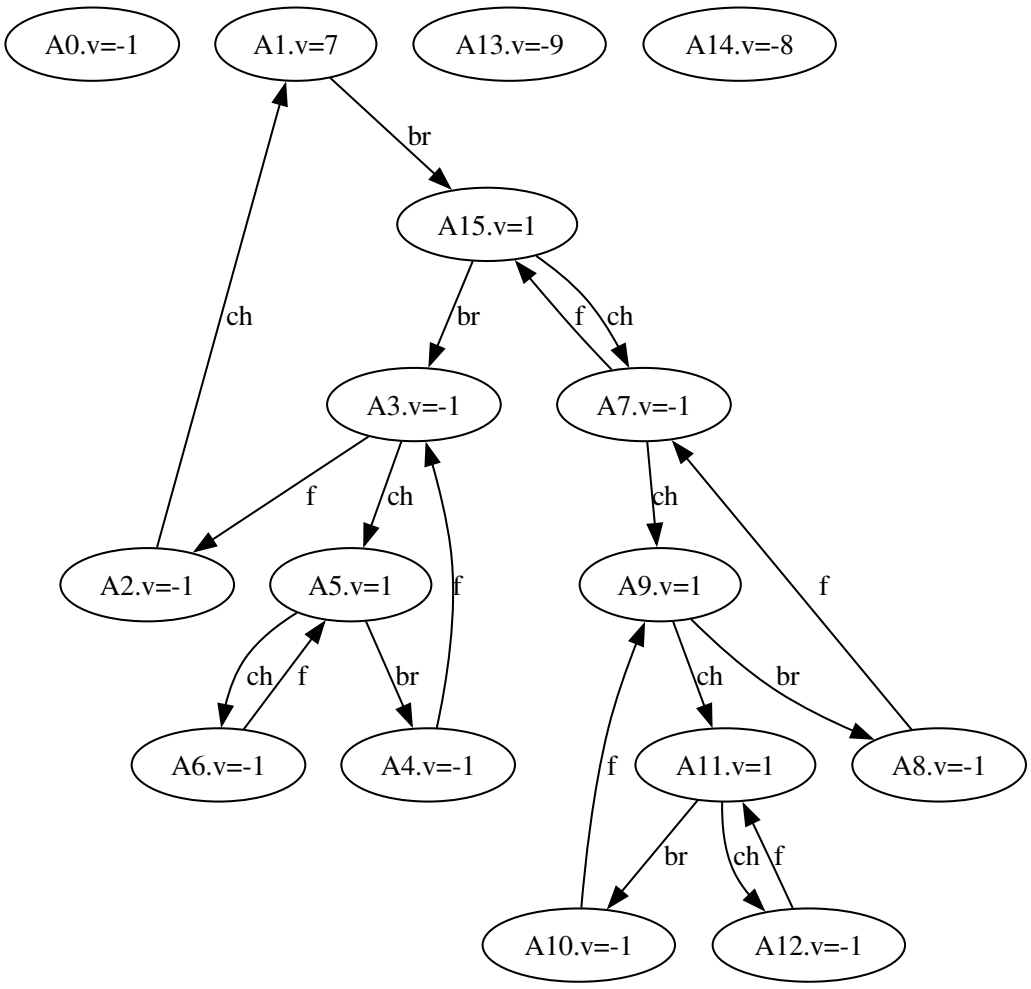

Figure 3: Representation after `DecreaseKey` operation.

```
decKey A[1] to 7
Meld A[2] A[1]
link A[1] as child of A[2]
2
```

The implementation of the `getHook` function consists in traversing the `brother` list all the way to the end then going to the parent node and then continuing the search until a field to the argument node is found.

The `Min` function simply calls the `abs` function on the `v` field of the corresponding argument node.

The `ExtractMin` function also receives a root node as argument. This argument root is detached from its children nodes and reset back to the initial configuration. The function works in a two pass process over the children nodes, also referred to as the `brother` list. Contrary to the description in the reference article **?** both passes are performed left to right on the `brother` list. In the first pass the elements on the list are melded in pairs. If the total number of children is odd the last element is left unpaired. The arguments to the meld function are in the same order as the elements appear in the list. In the second pass all members of the new `brother` list are melded into a single tree. As mentioned before this process proceeds in a left to right fashion. For examples of this process see examples 2 and 3. The function returns the root of resulting heap. This result is usually different from the argument root, except when the whole heap consists only off this node. In this corner case the result should be the argument root and moreover the node value should be kept instead of reset.

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.

The format of the input file is the following. The first line contains an 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
S 1
P 1
V 1 12
S 1
R 1 1 11
S 1
E 1
S 1
V 1 10
S 1
V 13 9
S 13
V 14 9
S 14
V 14 8
S 14
U 1 2
```

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

output 1

```
node: 1 v: -1 child: -1 brother: -1
node: 1 v: -1 child: -1 brother: -1
set A[1] to 12
node: 1 v: -12 child: -1 brother: -1
decKey A[1] to 11
1
node: 1 v: -11 child: -1 brother: -1
extractMin A[1]
1
node: 1 v: -11 child: -1 brother: -1
set A[1] to 10
node: 1 v: -10 child: -1 brother: -1
set A[13] to 9
```

```
node: 13 v: -9 child: -1 brother: -1
set A[14] to 9
node: 14 v: -9 child: -1 brother: -1
set A[14] to 8
node: 14 v: -8 child: -1 brother: -1
Meld A[1] A[2]
Swap A[1] and A[2]
link A[1] as child of A[2]
2
Meld A[3] A[4]
link A[4] as child of A[3]
3
Meld A[5] A[6]
link A[6] as child of A[5]
5
Meld A[3] A[5]
link A[5] as child of A[3]
3
Meld A[2] A[3]
link A[3] as child of A[2]
2
node: 0 v: -1 child: -1 brother: -1
Meld A[7] A[8]
link A[8] as child of A[7]
7
Meld A[9] A[10]
link A[10] as child of A[9]
9
Meld A[11] A[12]
link A[12] as child of A[11]
11
Meld A[9] A[11]
link A[11] as child of A[9]
9
Meld A[7] A[9]
link A[9] as child of A[7]
7
Meld A[15] A[7]
link A[7] as child of A[15]
15
node: 15 v: -1 child: 7 brother: -1
```

```
node: 12 v: -1 child: -1 brother: 11
Meld A[2] A[15]
link A[15] as child of A[2]
2
decKey A[1] to 7
Meld A[2] A[1]
link A[1] as child of A[2]
2
node: 1 v: 7 child: -1 brother: 15
node: 1 v: 7 child: -1 brother: 15
node: 15 v: 1 child: 7 brother: 3
node: 3 v: -1 child: 5 brother: 2
value A[2]
1
node: 7 v: -1 child: 9 brother: 15
node: 13 v: -9 child: -1 brother: -1
node: 14 v: -8 child: -1 brother: -1
decKey A[1] to 6
Meld A[2] A[1]
link A[1] as child of A[2]
2
Meld A[13] A[14]
Swap A[13] and A[14]
link A[13] as child of A[14]
14
decKey A[13] to 7
Meld A[14] A[13]
Swap A[14] and A[13]
link A[14] as child of A[13]
13
Final configuration:
node: 0 v: -1 child: -1 brother: -1
node: 1 v: 6 child: -1 brother: 15
node: 2 v: -1 child: 1 brother: -1
node: 3 v: -1 child: 5 brother: 2
node: 4 v: -1 child: -1 brother: 3
node: 5 v: 1 child: 6 brother: 4
node: 6 v: -1 child: -1 brother: 5
node: 7 v: -1 child: 9 brother: 15
node: 8 v: -1 child: -1 brother: 7
node: 9 v: 1 child: 11 brother: 8
```

```
node: 10 v: -1 child: -1 brother: 9
node: 11 v: 1 child: 12 brother: 10
node: 12 v: -1 child: -1 brother: 11
node: 13 v: -7 child: 14 brother: -1
node: 14 v: -8 child: -1 brother: 13
node: 15 v: 1 child: 7 brother: 3
```

input 2

```
16
V 1 2
V 2 3
V 3 4
V 4 5
V 5 6
V 6 7
V 7 8
V 8 9
V 9 10
V 10 11
V 11 12
V 12 13
V 13 14
V 14 15
V 15 16
U 0 1
U 2 0
U 0 3
U 4 0
U 0 5
U 6 0
U 0 7
U 8 0
U 0 9
U 0 10
U 0 11
U 0 12
U 0 13
U 0 14
U 0 15
P 0
```

```
P 1
E 0
E 0
X
```

output 2

```
set A[1] to 2
set A[2] to 3
set A[3] to 4
set A[4] to 5
set A[5] to 6
set A[6] to 7
set A[7] to 8
set A[8] to 9
set A[9] to 10
set A[10] to 11
set A[11] to 12
set A[12] to 13
set A[13] to 14
set A[14] to 15
set A[15] to 16
Meld A[0] A[1]
link A[1] as child of A[0]
0
Meld A[2] A[0]
Swap A[2] and A[0]
link A[2] as child of A[0]
0
Meld A[0] A[3]
link A[3] as child of A[0]
0
Meld A[4] A[0]
Swap A[4] and A[0]
link A[4] as child of A[0]
0
Meld A[0] A[5]
link A[5] as child of A[0]
0
Meld A[6] A[0]
Swap A[6] and A[0]
```

```
link A[6] as child of A[0]
0
Meld A[0] A[7]
link A[7] as child of A[0]
0
Meld A[8] A[0]
Swap A[8] and A[0]
link A[8] as child of A[0]
0
Meld A[0] A[9]
link A[9] as child of A[0]
0
Meld A[0] A[10]
link A[10] as child of A[0]
0
Meld A[0] A[11]
link A[11] as child of A[0]
0
Meld A[0] A[12]
link A[12] as child of A[0]
0
Meld A[0] A[13]
link A[13] as child of A[0]
0
Meld A[0] A[14]
link A[14] as child of A[0]
0
Meld A[0] A[15]
link A[15] as child of A[0]
0
node: 0 v: -1 child: 15 brother: -1
node: 1 v: -2 child: -1 brother: 0
extractMin A[0]
Meld A[15] A[14]
Swap A[15] and A[14]
link A[15] as child of A[14]
Meld A[13] A[12]
Swap A[13] and A[12]
link A[13] as child of A[12]
Meld A[11] A[10]
Swap A[11] and A[10]
```

```
link A[11] as child of A[10]
Meld A[9] A[8]
Swap A[9] and A[8]
link A[9] as child of A[8]
Meld A[7] A[6]
Swap A[7] and A[6]
link A[7] as child of A[6]
Meld A[5] A[4]
Swap A[5] and A[4]
link A[5] as child of A[4]
Meld A[3] A[2]
Swap A[3] and A[2]
link A[3] as child of A[2]
Meld A[14] A[12]
Swap A[14] and A[12]
link A[14] as child of A[12]
Meld A[12] A[10]
Swap A[12] and A[10]
link A[12] as child of A[10]
Meld A[10] A[8]
Swap A[10] and A[8]
link A[10] as child of A[8]
Meld A[8] A[6]
Swap A[8] and A[6]
link A[8] as child of A[6]
Meld A[6] A[4]
Swap A[6] and A[4]
link A[6] as child of A[4]
Meld A[4] A[2]
Swap A[4] and A[2]
link A[4] as child of A[2]
Meld A[2] A[1]
Swap A[2] and A[1]
link A[2] as child of A[1]
1
extractMin A[0]
0
Final configuration:
node: 0 v: -1 child: -1 brother: -1
node: 1 v: -2 child: 2 brother: -1
node: 2 v: -3 child: 4 brother: 1
```

```
node: 3 v: -4 child: -1 brother: 2
node: 4 v: 5 child: 6 brother: 3
node: 5 v: -6 child: -1 brother: 4
node: 6 v: 7 child: 8 brother: 5
node: 7 v: -8 child: -1 brother: 6
node: 8 v: 9 child: 10 brother: 7
node: 9 v: -10 child: -1 brother: 8
node: 10 v: 11 child: 12 brother: 9
node: 11 v: -12 child: -1 brother: 10
node: 12 v: 13 child: 14 brother: 11
node: 13 v: -14 child: -1 brother: 12
node: 14 v: 15 child: 15 brother: 13
node: 15 v: -16 child: -1 brother: 14
```

input 3

```
16
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
U 0 15
P 0
P 1
E 0
E 0
X
```

output 3

```
Meld A[0] A[1]
link A[1] as child of A[0]
0
Meld A[0] A[2]
link A[2] as child of A[0]
0
Meld A[0] A[3]
link A[3] as child of A[0]
0
Meld A[0] A[4]
link A[4] as child of A[0]
0
Meld A[0] A[5]
link A[5] as child of A[0]
0
Meld A[0] A[6]
link A[6] as child of A[0]
0
Meld A[0] A[7]
link A[7] as child of A[0]
0
Meld A[0] A[8]
link A[8] as child of A[0]
0
Meld A[0] A[9]
link A[9] as child of A[0]
0
Meld A[0] A[10]
link A[10] as child of A[0]
0
Meld A[0] A[11]
link A[11] as child of A[0]
0
Meld A[0] A[12]
link A[12] as child of A[0]
0
Meld A[0] A[13]
link A[13] as child of A[0]
0
```

```
Meld A[0] A[14]
link A[14] as child of A[0]
0
Meld A[0] A[15]
link A[15] as child of A[0]
0
node: 0 v: -1 child: 15 brother: -1
node: 1 v: -1 child: -1 brother: 0
extractMin A[0]
Meld A[15] A[14]
link A[14] as child of A[15]
Meld A[13] A[12]
link A[12] as child of A[13]
Meld A[11] A[10]
link A[10] as child of A[11]
Meld A[9] A[8]
link A[8] as child of A[9]
Meld A[7] A[6]
link A[6] as child of A[7]
Meld A[5] A[4]
link A[4] as child of A[5]
Meld A[3] A[2]
link A[2] as child of A[3]
Meld A[15] A[13]
link A[13] as child of A[15]
Meld A[15] A[11]
link A[11] as child of A[15]
Meld A[15] A[9]
link A[9] as child of A[15]
Meld A[15] A[7]
link A[7] as child of A[15]
Meld A[15] A[5]
link A[5] as child of A[15]
Meld A[15] A[3]
link A[3] as child of A[15]
Meld A[15] A[1]
link A[1] as child of A[15]
15
extractMin A[0]
0
Final configuration:
```

```
node: 0 v: -1 child: -1 brother: -1
node: 1 v: 1 child: -1 brother: 3
node: 2 v: -1 child: -1 brother: 3
node: 3 v: 1 child: 2 brother: 5
node: 4 v: -1 child: -1 brother: 5
node: 5 v: 1 child: 4 brother: 7
node: 6 v: -1 child: -1 brother: 7
node: 7 v: 1 child: 6 brother: 9
node: 8 v: -1 child: -1 brother: 9
node: 9 v: 1 child: 8 brother: 11
node: 10 v: -1 child: -1 brother: 11
node: 11 v: 1 child: 10 brother: 13
node: 12 v: -1 child: -1 brother: 13
node: 13 v: 1 child: 12 brother: 14
node: 14 v: -1 child: -1 brother: 15
node: 15 v: -1 child: 1 brother: -1
```

input 4

```
16
V 1 2
V 2 3
V 3 4
V 4 5
V 5 6
V 6 7
V 7 8
V 8 9
V 9 10
V 10 11
V 11 12
V 12 13
V 13 14
V 14 15
V 15 16
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
U 0 15
P 0
P 1
E 0
E 1
E 2
E 3
E 4
E 5
E 6
E 7
E 8
E 9
E 10
E 11
E 12
E 13
E 14
E 15
X
```

output 4

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

```
set A[11] to 12
set A[12] to 13
set A[13] to 14
set A[14] to 15
set A[15] to 16
Meld A[0] A[1]
link A[1] as child of A[0]
0
Meld A[0] A[2]
link A[2] as child of A[0]
0
Meld A[0] A[3]
link A[3] as child of A[0]
0
Meld A[0] A[4]
link A[4] as child of A[0]
0
Meld A[0] A[5]
link A[5] as child of A[0]
0
Meld A[0] A[6]
link A[6] as child of A[0]
0
Meld A[0] A[7]
link A[7] as child of A[0]
0
Meld A[0] A[8]
link A[8] as child of A[0]
0
Meld A[0] A[9]
link A[9] as child of A[0]
0
Meld A[0] A[10]
link A[10] as child of A[0]
0
Meld A[0] A[11]
link A[11] as child of A[0]
0
Meld A[0] A[12]
link A[12] as child of A[0]
0
```

```
Meld A[0] A[13]
link A[13] as child of A[0]
0
Meld A[0] A[14]
link A[14] as child of A[0]
0
Meld A[0] A[15]
link A[15] as child of A[0]
0
node: 0 v: -1 child: 15 brother: -1
node: 1 v: -2 child: -1 brother: 0
extractMin A[0]
Meld A[15] A[14]
Swap A[15] and A[14]
link A[15] as child of A[14]
Meld A[13] A[12]
Swap A[13] and A[12]
link A[13] as child of A[12]
Meld A[11] A[10]
Swap A[11] and A[10]
link A[11] as child of A[10]
Meld A[9] A[8]
Swap A[9] and A[8]
link A[9] as child of A[8]
Meld A[7] A[6]
Swap A[7] and A[6]
link A[7] as child of A[6]
Meld A[5] A[4]
Swap A[5] and A[4]
link A[5] as child of A[4]
Meld A[3] A[2]
Swap A[3] and A[2]
link A[3] as child of A[2]
Meld A[14] A[12]
Swap A[14] and A[12]
link A[14] as child of A[12]
Meld A[12] A[10]
Swap A[12] and A[10]
link A[12] as child of A[10]
Meld A[10] A[8]
Swap A[10] and A[8]
```

```
link A[10] as child of A[8]
Meld A[8] A[6]
Swap A[8] and A[6]
link A[8] as child of A[6]
Meld A[6] A[4]
Swap A[6] and A[4]
link A[6] as child of A[4]
Meld A[4] A[2]
Swap A[4] and A[2]
link A[4] as child of A[2]
Meld A[2] A[1]
Swap A[2] and A[1]
link A[2] as child of A[1]
1
extractMin A[1]
2
extractMin A[2]
Meld A[4] A[3]
Swap A[4] and A[3]
link A[4] as child of A[3]
3
extractMin A[3]
4
extractMin A[4]
Meld A[6] A[5]
Swap A[6] and A[5]
link A[6] as child of A[5]
5
extractMin A[5]
6
extractMin A[6]
Meld A[8] A[7]
Swap A[8] and A[7]
link A[8] as child of A[7]
7
extractMin A[7]
8
extractMin A[8]
Meld A[10] A[9]
Swap A[10] and A[9]
link A[10] as child of A[9]
```

```
9
extractMin A[9]
10
extractMin A[10]
Meld A[12] A[11]
Swap A[12] and A[11]
link A[12] as child of A[11]
11
extractMin A[11]
12
extractMin A[12]
Meld A[14] A[13]
Swap A[14] and A[13]
link A[14] as child of A[13]
13
extractMin A[13]
14
extractMin A[14]
15
extractMin A[15]
15
Final configuration:
node: 0 v: -1 child: -1 brother: -1
node: 1 v: -1 child: -1 brother: -1
node: 2 v: -1 child: -1 brother: -1
node: 3 v: -1 child: -1 brother: -1
node: 4 v: -1 child: -1 brother: -1
node: 5 v: -1 child: -1 brother: -1
node: 6 v: -1 child: -1 brother: -1
node: 7 v: -1 child: -1 brother: -1
node: 8 v: -1 child: -1 brother: -1
node: 9 v: -1 child: -1 brother: -1
node: 10 v: -1 child: -1 brother: -1
node: 11 v: -1 child: -1 brother: -1
node: 12 v: -1 child: -1 brother: -1
node: 13 v: -1 child: -1 brother: -1
node: 14 v: -1 child: -1 brother: -1
node: 15 v: -16 child: -1 brother: -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` 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.

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

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

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].child)
      fprintf(f, "A%d -> A%d [label=\"ch\"]\n",
              i, ptr2loc(A[i].child, A));
    if(NULL != A[i].brother)
      fprintf(f, "A%d -> A%d [label=\"%s\"]\n",
              i, ptr2loc(A[i].brother, A),
              (0 < A[i].v) ? "br" : "f"
              );
  }
  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].child = NULL;
      if(-1 != j)
        A[i].child = &A[j];

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

    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

Michael L Fredman, Robert Sedgewick, Daniel D Sleator, and Robert E Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1-4):111–129, 1986.