TÉCNICO LISBOA

DEI
Departamento de Engenharia Informática

Advanced Algorithms

2021/2022

Project 1

Build a Binomial Heap

This project considers the problem of constructing a binomial heap. The first delivery deadline for the project code is the 18 March, at 13:30. The deadline for the recovery season is 22 June at 12:00. The deadline for the special season is 29 July 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 Binomial Heaps

The challenge is to implement the binomial heap data structure. An algorithm for this problem is explained in Chapter 19 of the second edition of the Introduction to Algorithms book [1]. 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 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 */
  int rank; /* Number of childen */
  node child; /* First child if it has any */
  node brother; /* In heap list points to larger rank.
                   Inside a tree points to smaller rank. */
  node father; /* Points up */
};
```

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 `rank` field is used to store the number of children of the current node, recall that a binomial heap consists in a list of binomial trees such that each `rank` value occurs at most once in the list. The `child` field points to the first child of the node. If the node contains no children this pointer should be set to `NULL`. If the node contains several children this pointer should indicate the one which has the largest `rank` value. The `brother` field is used to point to the next node that shares the same parent. If such a node exists then the `brother` node has a strictly smaller rank value. Otherwise if `node` has no parent, i.e., it is a tree in the root list, then the `brother` pointer is used to point to the next root, which has a strictly larger `rank` value. We will explain which operations alter this ordering shortly. Finally the `parent` field points to its father `node`, in case it exists. If the current `node` is the root of a binomial tree then the `parent` field stores the value `NULL`.

The `ExtracMin` and `Delete` operations should reset a `node` to its default initial state. Meaning that the fields `v` and `rank` should be set to 0 and the `child`, `brother` and `father` fields should be set to `NULL`. Note that in particular the `rank` field can be 0 if and only if the `child` field is `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)
{
  if(NULL == v)
    printf("NULL\n");
  else {
    printf("node: %d ", ptr2loc(v, A));
    printf("v: %d ", v->v);
    printf("rank: %d ", v->rank);
    printf("child: %d ", ptr2loc(v->child, A));
    printf("brother: %d ", ptr2loc(v->brother, A));
    printf("father: %d ", ptr2loc(v->father, 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 calls `showNode` for the current `node` and then follows the `brother` fields recursively until it reaches a `NULL` such field.

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`, `brother` and `father` fields are all `NULL`.

U `(heap, heap)` The `Unite` function is used to join two binomial heaps, i.e., merge two lists of roots into a single list. This function returns the identifier of the resulting heap.

R `(node, v)` The `DecreaseKey` function is used to decrease the `v` field of the current `node`. This `node` may be part of a larger binomial tree and therefore this operation might need to move the value upwards on the tree.

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

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

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

D `(heap, node)` The `Delete` function removes a specific node from the given heap and returns the identification of the resulting heap.

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` and `rank` 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 binomial tree and moreover also a binomial heap. In fact any index i represents a different binomial tree and binomial heap. To merge these heaps we will use the `Union` operation that we will describe next.

To illustrate the `Union` operation consider the two heaps in Figure 1. Each node contains two numbers. The first number is the identifier of the node, i.e., its position in `A`. The second number is the value stored in the field `v`. In this case the second values are all 0. These heaps will occur in the first input given below. Now consider the command `U 0 15`, which will unite these two heaps. Before executing this command we can execute the `P 0` command to obtain the list of roots in the binomial heap that starts at node 0. The resulting output would then be the following:

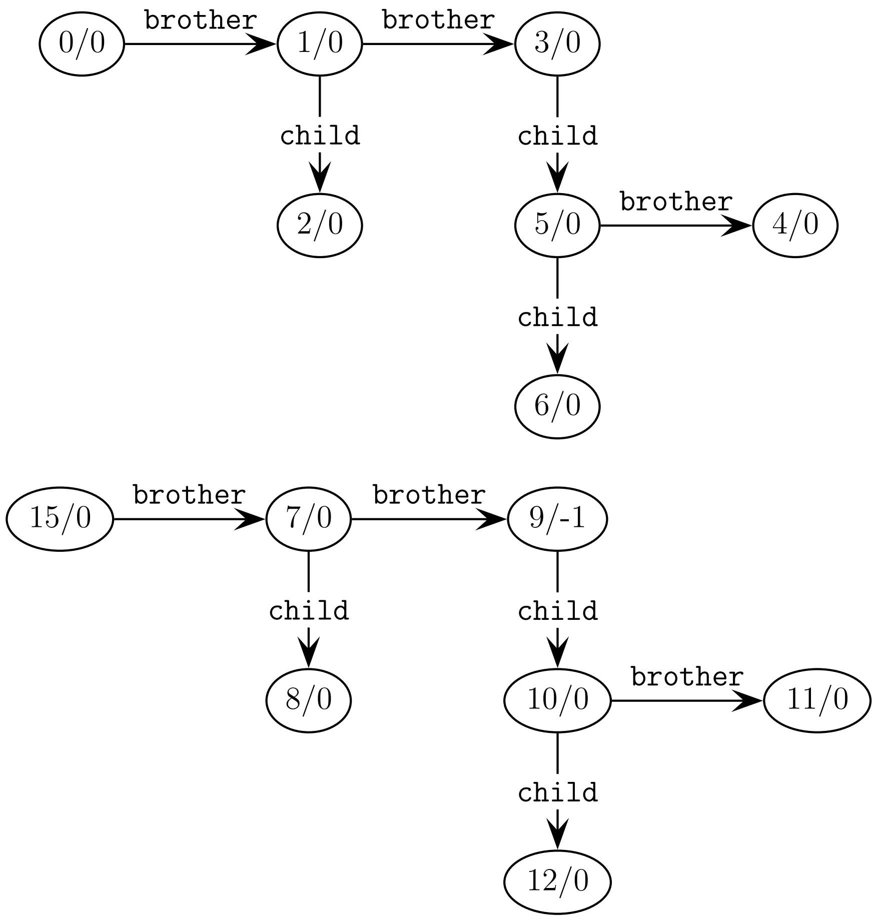

Figure 1: Representation of two binomial heaps before `unite` operation.

```
node: 0 v: 0 rank: 0 child: -1 brother: 1 father: -1
node: 1 v: 0 rank: 1 child: 2 brother: 3 father: -1
node: 3 v: 0 rank: 2 child: 5 brother: -1 father: -1
```

Likewise to the `P 15` command would produce the following output:

```
node: 15 v: 0 rank: 0 child: -1 brother: 7 father: -1
node: 7 v: 0 rank: 1 child: 8 brother: 9 father: -1
node: 9 v: -1 rank: 2 child: 11 brother: -1 father: -1
```

To execute the `U 0 15` command the following process occurs. First compare the trees rooted at 0 and at 15. If the `rank` field of these trees is different the heap that contains the tree with the smallest `rank` is processed. If the small `rank` tree is the only tree with such a `rank` in the heap then it passes to the final heap, otherwise if there are two tree with the same `rank` in the same heap (and the tree in the other heap has a different `rank`) then they get linked.

In the case we are considering the current trees of both heaps have the same `rank` value, therefore we decide to link them. To determine which tree remains the root we compare their `v` fields, the one with the smallest value remains. In this case both trees have the same value, 0. When there is a tie in the `v` fields we convention that the tree that belonged to the heap in the first argument of the `U` command remains the root. In this case the node with index 0 will remain the root of a new tree with `rank 1`. Every time there is a `Link` operation the program should print that information with this format `"link %d as child of %d\n"`.

In the case of the `U 0 15` command we should obtain this output:

```
link 15 as child of 0
link 7 as child of 1
link 3 as child of 9
```

After linking the node 15 as a child of the node 0 we need to select another node from the second heap. Note that to maintain consistency the binomial tree rooted at 15 needs to be removed from the second heap. Hence at this point we are considering the binomial trees rooted at 0 and at 7. A this point both the trees have `rank` 1, therefore we consider linking them. Using the same criteria as before, since their `v` value is also 0, we could link 7 as a child of 0. However this would constitute a problem for the first heap, because there would now be a binomial tree with `rank` 2 before a binomial tree with rank 1. Hence to avoid this situation we first look at the trees in the first heap. The first two trees are rooted at 0 and 1. After the initial link both these trees have `rank` 1.

We could link these two trees, as described before. However in this case the first tree of the second heap (rooted at 7) also has a `rank` value of 1. Therefore instead we immediately move the tree rooted at 0 to the final heap. This implies moving the index in the first heap and start considering the tree rooted at 1. Note that both approaches, linking 0 and 1 or moving 0 to the final heap, would avoid creating the out of order `rank` situation we described earlier. In general whenever the first two trees of a heap have the same `rank` value we link them, except if the first tree of the other heap also has the same rank, in which case the first tree is moved to the final heap and we move the index in the heap. This means that the tree with index 0 is moved to the final heap and instead we compare the trees 1 and 7. Like before these trees are tied in the `rank` and `v` fields and therefore the tree 1 remains the root because it is part of the heap in the first argument of the `U` command. Finally for the trees 3 and 9 the node with index 9 has a value of -1 in its `v` field which is smaller than the value of 0 in the `v` field of the node with index 0. Therefore in this case the node 3 becomes a child of the node 9.

A final important consideration is how to solve ties when linking two tree with the same rank, in the same heap. For example this would occur in the previous example if we had linked the trees rooted at 0 and 1, instead of moving the tree 0 to the final heap. In this case both the nodes have a value `v` of 0 which means that this value can not be used to decide which node ends up as the root. In this case we convention that the first tree in the heap ends up as the root therefore 0 would be the resulting root and we would obtain a message as follows:

```
link 1 as child of 0
```

Concerning the return value of the `Unite` function it should be the final merged heap. Therefore in the example we gave it should return the value 0 because the node with index 0 is the first element in the list of roots. If we now execute the `P 0` we would obtain the following result:

```
node: 0 v: 0 rank: 1 child: 15 brother: 1 father: -1
node: 1 v: 0 rank: 2 child: 7 brother: 9 father: -1
node: 9 v: -1 rank: 3 child: 3 brother: -1 father: -1
```

Next we consider the operations related to the minimum value. The simplest such operation is the `Min` operation. This operation works by traversing the list of trees. For each tree it considers only the root node, as these nodes contain the minimum `v` value that exists in the tree. The operation returns the overall minimum `v` value obtained in this search. The `ArgMin` uses a similar approach but instead returns the identifier of the root node that contains the minimum `v` value. In the case of ties, where there are several instances of this minimum value, the result should be the first such value that is obtained when traversing the list of roots. Since this list is sorted by increasing rank values the resulting node is the one that, among those that contain the minimum `v` value, has the smallest `rank`. For example the result of a `M 0` operation on the heap just created by the `Unite` function returns -1, whereas a `A 0` command returns 9.

The most complicated operation related to minimums is the `ExtractMin` operation. However it builds on top of the primitives described above. This operation works by first identifying the node with minimum `v` value, the same as the `ArgMin` operation. This node is removed from the list of roots. Recall that removing a node from heap implies that all its information should be zeroed. In a second step a `Unite` operation is executed. The first argument of this `Unite` operation is the list of roots that no longer contains the node identified by `ArgMin`. The second argument requires some processing, it consists of the nodes that are children of the node identified by `ArgMin`. However it is not enough to use the `child` field of this node, because this child is the head of a list that is ordered by decreasing `rank` value. The `Union` operation requires lists of roots that are ordered by increasing rank values. This implies that it is necessary to reverse the list pointed by the `child` field. Once reversed this list can thus be used as the second argument of the `Unite` operation. The `Unite` operation is used in the general case, however we must also discuss some special cases. If the argument of the `ExtractMin` consists of a list with a single root in its list, then this node will be the identified by the `ArgMin` operation. Hence the list of roots without this node will become empty, therefore there is no need to invoke the `Unite` operation. Its is enough to return the list of children, after it is reversed. Another special case occurs when the list of children of the node identified by `ArgMin` is empty, in this case it is also not necessary to invoke the `Unite` operation. It is enough to return the list of roots of the original argument, from which the node identified by `ArgMin` is removed. The very last special case is when both these conditions occur at the same time, i.e., the list of roots contains only one root and that root has no children. In this case the `ExtractMin` function should return this root.

Another function that the heap data structure must support is the `DecreaseKey` operation. This function receives as arguments a position in `A` and a new value for the field `v`. The objective is to reduce the value of the field `v` to the number given as argument. If the given argument is actually larger than the current value then no modification is executed. If the given argument is indeed smaller it is not immediate that this modification results in a valid heap, as the `v` field of the node should not be smaller than the `v` field of its parent. Hence it might be necessary to traverse upwards through the tree until the modification can be applied. Note that this process copies the `v` fields of the parents to the children, to make room for the new value. In case the new value is equal to a certain `v` value found in this process, i.e., there is a tie, the traversal should immediately stop. This means the traversal should be as small as possible. The function should return the position of the node that received the argument value.

We can combine the `Min`, `DecreaseKey` and `ExtractMin` to implement the `Delete` function. This function receives as arguments an index that represents a heap, i.e., a list of roots, and an index that represents a `node` inside that heap. To remove the node given in the second argument we start by identifying the overall minimum value inside the heap, with the `Min` operation. We then invoke the `DecreaseKey` operation using as arguments the desired `node` and the minimum value minus one. The procedure finishes by invoking the `ExtractMin` operation and uses its return value as the return of the `Delete` operation.

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 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 `P`, `S` should print the information indicated above. The command `U` and the commands that depend on `Unite` 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 11
S 1
R 1 12
S 1
E 1
S 1
V 1 10
S 1
D 1 1
S 1
V 13 -2
S 13
V 14 -2
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
S 12
S 9
P 15
U 0 15
P 0
M 0
A 0
U 13 0
P 13
M 13
A 13
U 14 13
P 14
M 14
A 14
P 3
E 14
P 13
X
```

output 1

```
node: 1 v: 0 rank: 0 child: -1 brother: -1 father: -1
node: 1 v: 0 rank: 0 child: -1 brother: -1 father: -1
node: 1 v: 11 rank: 0 child: -1 brother: -1 father: -1
1
node: 1 v: 11 rank: 0 child: -1 brother: -1 father: -1
1
node: 1 v: 0 rank: 0 child: -1 brother: -1 father: -1
node: 1 v: 10 rank: 0 child: -1 brother: -1 father: -1
1
node: 1 v: 0 rank: 0 child: -1 brother: -1 father: -1
node: 13 v: -2 rank: 0 child: -1 brother: -1 father: -1
```

```
node: 14 v: -3 rank: 0 child: -1 brother: -1 father: -1
link 2 as child of 1
1
link 4 as child of 3
3
link 6 as child of 5
5
link 5 as child of 3
3
1
0
node: 0 v: 0 rank: 0 child: -1 brother: 1 father: -1
node: 1 v: 0 rank: 1 child: 2 brother: 3 father: -1
node: 3 v: 0 rank: 2 child: 5 brother: -1 father: -1
link 8 as child of 7
7
link 10 as child of 9
9
link 12 as child of 11
11
link 11 as child of 9
9
7
15
node: 15 v: 0 rank: 0 child: -1 brother: 7 father: -1
node: 7 v: 0 rank: 1 child: 8 brother: 9 father: -1
node: 9 v: 0 rank: 2 child: 11 brother: -1 father: -1
node: 12 v: 0 rank: 0 child: -1 brother: -1 father: 11
12
node: 12 v: 0 rank: 0 child: -1 brother: -1 father: 11
9
node: 12 v: 0 rank: 0 child: -1 brother: -1 father: 11
node: 9 v: -1 rank: 2 child: 11 brother: -1 father: -1
node: 15 v: 0 rank: 0 child: -1 brother: 7 father: -1
node: 7 v: 0 rank: 1 child: 8 brother: 9 father: -1
node: 9 v: -1 rank: 2 child: 11 brother: -1 father: -1
link 15 as child of 0
link 7 as child of 1
link 3 as child of 9
0
node: 0 v: 0 rank: 1 child: 15 brother: 1 father: -1
```

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

```
node: 13 v: -2 rank: 0 child: -1 brother: 0 father: -1
node: 14 v: 0 rank: 0 child: -1 brother: -1 father: -1
node: 15 v: 0 rank: 0 child: -1 brother: -1 father: 0
```

input 2

```
32
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 2 3
U 0 4
U 4 5
U 4 6
U 6 7
U 0 8
U 8 9
U 8 10
U 10 11
U 8 12
U 12 13
U 12 14
U 14 15
U 17 16
U 18 17
U 19 18
U 20 19
```

```
U 21 20
U 22 21
U 23 22
U 24 23
U 25 24
U 26 25
U 27 26
U 28 27
U 29 28
U 30 29
U 31 30
X
```

output 2

```
link 1 as child of 0
0
2
link 3 as child of 2
link 2 as child of 0
0
4
link 5 as child of 4
4
6
link 7 as child of 6
link 6 as child of 4
link 4 as child of 0
0
8
link 9 as child of 8
8
10
link 11 as child of 10
link 10 as child of 8
8
12
link 13 as child of 12
12
14
link 15 as child of 14
link 14 as child of 12
link 12 as child of 8
link 8 as child of 0
0
link 16 as child of 17
17
18
link 18 as child of 19
link 17 as child of 19
19
20
link 20 as child of 21
21
22
link 22 as child of 23
link 21 as child of 23
link 19 as child of 23
23
24
link 24 as child of 25
25
26
link 26 as child of 27
link 25 as child of 27
27
28
link 28 as child of 29
29
30
link 30 as child of 31
link 29 as child of 31
link 27 as child of 31
link 23 as child of 31
31
Final configuration:
node: 0 v: 0 rank: 4 child: 8 brother: -1 father: -1
node: 1 v: 1 rank: 0 child: -1 brother: -1 father: 0
node: 2 v: 2 rank: 1 child: 3 brother: 1 father: 0
node: 3 v: 3 rank: 0 child: -1 brother: -1 father: 2
node: 4 v: 4 rank: 2 child: 6 brother: 2 father: 0
node: 5 v: 5 rank: 0 child: -1 brother: -1 father: 4
```

```
node: 6 v: 6 rank: 1 child: 7 brother: 5 father: 4
node: 7 v: 7 rank: 0 child: -1 brother: -1 father: 6
node: 8 v: 8 rank: 3 child: 12 brother: 4 father: 0
node: 9 v: 9 rank: 0 child: -1 brother: -1 father: 8
node: 10 v: 10 rank: 1 child: 11 brother: 9 father: 8
node: 11 v: 11 rank: 0 child: -1 brother: -1 father: 10
node: 12 v: 12 rank: 2 child: 14 brother: 10 father: 8
node: 13 v: 13 rank: 0 child: -1 brother: -1 father: 12
node: 14 v: 14 rank: 1 child: 15 brother: 13 father: 12
node: 15 v: 15 rank: 0 child: -1 brother: -1 father: 14
node: 16 v: 0 rank: 0 child: -1 brother: -1 father: 17
node: 17 v: 0 rank: 1 child: 16 brother: 18 father: 19
node: 18 v: 0 rank: 0 child: -1 brother: -1 father: 19
node: 19 v: 0 rank: 2 child: 17 brother: 21 father: 23
node: 20 v: 0 rank: 0 child: -1 brother: -1 father: 21
node: 21 v: 0 rank: 1 child: 20 brother: 22 father: 23
node: 22 v: 0 rank: 0 child: -1 brother: -1 father: 23
node: 23 v: 0 rank: 3 child: 19 brother: 27 father: 31
node: 24 v: 0 rank: 0 child: -1 brother: -1 father: 25
node: 25 v: 0 rank: 1 child: 24 brother: 26 father: 27
node: 26 v: 0 rank: 0 child: -1 brother: -1 father: 27
node: 27 v: 0 rank: 2 child: 25 brother: 29 father: 31
node: 28 v: 0 rank: 0 child: -1 brother: -1 father: 29
node: 29 v: 0 rank: 1 child: 28 brother: 30 father: 31
node: 30 v: 0 rank: 0 child: -1 brother: -1 father: 31
node: 31 v: 0 rank: 4 child: 23 brother: -1 father: -1
```

input 3

```
17
U 1 0
U 2 1
U 3 2
U 4 3
U 5 4
U 6 5
U 7 6
U 8 7
U 9 8
U 10 9
U 11 10
```

```
U 12 11
U 13 12
U 14 13
U 15 14
M 16
A 16
R 16 -20
M 16
A 16
M 15
A 15
R 0 -1
M 15
A 15
M 16
A 16
R 16 -21
M 16
A 16
R 0 -2
M 15
A 15
M 16
A 16
R 16 -22
M 16
A 16
R 0 -3
M 15
A 15
M 16
A 16
R 16 -23
M 16
A 16
R 0 -4
M 15
A 15
M 16
A 16
R 16 -24
```

```
M 16
A 16
R 0 -5
M 15
A 15
M 16
A 16
R 16 -25
M 16
A 16
R 2 -6
M 15
A 15
M 16
A 16
R 16 -26
M 16
A 16
R 4 -7
M 15
A 15
M 16
A 16
R 16 -27
M 16
A 16
R 4 -8
M 15
A 15
M 16
A 16
R 16 -28
M 16
A 16
R 6 -9
M 15
A 15
M 16
A 16
R 16 -29
M 16
```

```
A 16
R 8 -10
M 15
A 15
M 16
A 16
R 16 -30
M 16
A 16
R 8 -11
M 15
A 15
M 16
A 16
R 16 -31
M 16
A 16
R 8 -12
M 15
A 15
M 16
A 16
R 16 -32
M 16
A 16
R 10 -13
M 15
A 15
M 16
A 16
R 16 -33
M 16
A 16
R 12 -14
M 15
A 15
M 16
A 16
R 16 -34
M 16
A 16
```

```
R 12 -15
M 15
A 15
M 16
A 16
R 16 -35
M 16
A 16
R 14 -16
M 15
A 15
X
```

output 3

```
link 0 as child of 1
1
2
link 2 as child of 3
link 1 as child of 3
3
4
link 4 as child of 5
5
6
link 6 as child of 7
link 5 as child of 7
link 3 as child of 7
7
8
link 8 as child of 9
9
10
link 10 as child of 11
link 9 as child of 11
11
12
link 12 as child of 13
13
14
link 14 as child of 15
link 13 as child of 15
link 11 as child of 15
link 7 as child of 15
15
0
16
16
-20
16
0
15
15
-1
15
-20
16
16
-21
16
15
-2
15
-21
16
16
-22
16
15
-3
15
-22
16
16
-23
16
15
-4
15
-23
16
16
```

```
-24
16
15
-5
15
-24
16
16
-25
16
15
-6
15
-25
16
16
-26
16
15
-7
15
-26
16
16
-27
16
15
-8
15
-27
16
16
-28
16
15
-9
15
-28
16
16
-29
```

```
16
15
-10
15
-29
16
16
-30
16
15
-11
15
-30
16
16
-31
16
15
-12
15
-31
16
16
-32
16
15
-13
15
-32
16
16
-33
16
15
-14
15
-33
16
16
-34
16
```

```
15
-15
15
-34
16
16
-35
16
15
-16
15
Final configuration:
node: 0 v: -1 rank: 0 child: -1 brother: -1 father: 1
node: 1 v: -2 rank: 1 child: 0 brother: 2 father: 3
node: 2 v: -3 rank: 0 child: -1 brother: -1 father: 3
node: 3 v: -4 rank: 2 child: 1 brother: 5 father: 7
node: 4 v: -5 rank: 0 child: -1 brother: -1 father: 5
node: 5 v: -6 rank: 1 child: 4 brother: 6 father: 7
node: 6 v: -7 rank: 0 child: -1 brother: -1 father: 7
node: 7 v: -8 rank: 3 child: 3 brother: 11 father: 15
node: 8 v: -9 rank: 0 child: -1 brother: -1 father: 9
node: 9 v: -10 rank: 1 child: 8 brother: 10 father: 11
node: 10 v: -11 rank: 0 child: -1 brother: -1 father: 11
node: 11 v: -12 rank: 2 child: 9 brother: 13 father: 15
node: 12 v: -13 rank: 0 child: -1 brother: -1 father: 13
node: 13 v: -14 rank: 1 child: 12 brother: 14 father: 15
node: 14 v: -15 rank: 0 child: -1 brother: -1 father: 15
node: 15 v: -16 rank: 4 child: 7 brother: -1 father: -1
node: 16 v: -35 rank: 0 child: -1 brother: -1 father: -1
```

input 4

```
15
E 7
S 7
R 7 -1
P 7
D 7 7
P 7
U 1 0
U 2 1
```

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

output 4

```
7
node: 7 v: 0 rank: 0 child: -1 brother: -1 father: -1
7
```

```
node: 7 v: -1 rank: 0 child: -1 brother: -1 father: -1
7
node: 7 v: 0 rank: 0 child: -1 brother: -1 father: -1
link 0 as child of 1
1
2
link 2 as child of 3
link 1 as child of 3
3
4
link 4 as child of 5
5
6
link 6 as child of 7
link 5 as child of 7
link 3 as child of 7
7
8
link 8 as child of 9
9
10
link 10 as child of 11
link 9 as child of 11
11
12
link 12 as child of 13
13
14
7
11
node: 14 v: 0 rank: 0 child: -1 brother: 13 father: -1
node: 13 v: 0 rank: 1 child: 12 brother: 11 father: -1
node: 11 v: -2 rank: 2 child: 9 brother: 7 father: -1
node: 7 v: -1 rank: 3 child: 3 brother: -1 father: -1
node: 3 v: 0 rank: 2 child: 1 brother: 5 father: 7
node: 5 v: 0 rank: 1 child: 4 brother: 6 father: 7
node: 6 v: 0 rank: 0 child: -1 brother: -1 father: 7
link 6 as child of 14
link 5 as child of 13
link 3 as child of 11
14
```

```
node: 14 v: 0 rank: 1 child: 6 brother: 13 father: -1
node: 13 v: 0 rank: 2 child: 5 brother: 11 father: -1
node: 11 v: -2 rank: 3 child: 3 brother: -1 father: -1
node: 7 v: 0 rank: 0 child: -1 brother: -1 father: -1
7
node: 7 v: 0 rank: 0 child: -1 brother: 14 father: -1
node: 14 v: 0 rank: 1 child: 6 brother: 13 father: -1
node: 13 v: 0 rank: 2 child: 5 brother: 11 father: -1
node: 11 v: -2 rank: 3 child: 3 brother: -1 father: -1
11
node: 7 v: 0 rank: 0 child: -1 brother: 14 father: -1
node: 14 v: 0 rank: 1 child: 6 brother: 13 father: -1
node: 13 v: 0 rank: 2 child: 5 brother: 11 father: -1
node: 11 v: -2 rank: 3 child: 3 brother: -1 father: -1
node: 3 v: 0 rank: 2 child: 1 brother: 9 father: 11
node: 9 v: 0 rank: 1 child: 8 brother: 10 father: 11
node: 10 v: 0 rank: 0 child: -1 brother: -1 father: 11
link 10 as child of 7
link 9 as child of 14
link 3 as child of 13
7
node: 7 v: 0 rank: 1 child: 10 brother: 14 father: -1
node: 14 v: 0 rank: 2 child: 9 brother: 13 father: -1
node: 13 v: 0 rank: 3 child: 3 brother: -1 father: -1
node: 11 v: 0 rank: 0 child: -1 brother: -1 father: -1
11
node: 11 v: 0 rank: 0 child: -1 brother: 7 father: -1
node: 7 v: 0 rank: 1 child: 10 brother: 14 father: -1
node: 14 v: 0 rank: 2 child: 9 brother: 13 father: -1
node: 13 v: 0 rank: 3 child: 3 brother: -1 father: -1
0
11
7
node: 7 v: 0 rank: 1 child: 10 brother: 14 father: -1
node: 14 v: 0 rank: 2 child: 9 brother: 13 father: -1
node: 13 v: 0 rank: 3 child: 3 brother: -1 father: -1
node: 11 v: 0 rank: 0 child: -1 brother: -1 father: -1
Final configuration:
node: 0 v: 0 rank: 0 child: -1 brother: -1 father: 1
node: 1 v: 0 rank: 1 child: 0 brother: 2 father: 3
node: 2 v: 0 rank: 0 child: -1 brother: -1 father: 3
```

```
node: 3 v: 0 rank: 2 child: 1 brother: 5 father: 13
node: 4 v: 0 rank: 0 child: -1 brother: -1 father: 5
node: 5 v: 0 rank: 1 child: 4 brother: 12 father: 13
node: 6 v: 0 rank: 0 child: -1 brother: -1 father: 14
node: 7 v: 0 rank: 1 child: 10 brother: 14 father: -1
node: 8 v: 0 rank: 0 child: -1 brother: -1 father: 9
node: 9 v: 0 rank: 1 child: 8 brother: 6 father: 14
node: 10 v: 0 rank: 0 child: -1 brother: -1 father: 7
node: 11 v: 0 rank: 0 child: -1 brother: -1 father: -1
node: 12 v: 0 rank: 0 child: -1 brother: -1 father: 13
node: 13 v: 0 rank: 3 child: 3 brother: -1 father: -1
node: 14 v: 0 rank: 2 child: 9 brother: 13 father: -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, or including code in the report will not be considered for evaluation.

[1]`https://www.valgrind.org/`
[2]`https://github.com/terryyin/lizard`

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.

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

```
      fprintf(f, "A%d -> A%d [label=\"c\"]\n",
              i, ptr2loc(A[i].child, A));
    if(NULL != A[i].brother)
      fprintf(f, "A%d -> A%d [label=\"b\"]\n",
              i, ptr2loc(A[i].brother, A));
  }
  fprintf(f, "}\n");
}

void
structLoad(void)
{
  FILE *f;

  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;

      int i;
      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", &A[i].rank);

      int j;
      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];

      tok = strtok(NULL, " ");
      tok = strtok(NULL, " ");
      sscanf(tok, "%d", &j);
      A[i].father = NULL;
      if(-1 != j)
        A[i].father = &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 version shows only the `child` and `brother` pointers, but you may consider variations that show more information. 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

[1] Cormen, T. and Leiserson, C. and Rivest, R. and Stein, C. *Introduction to algorithms*. The Massachusetts Institute of Technology, 2nd Edition, 2001.