Contents

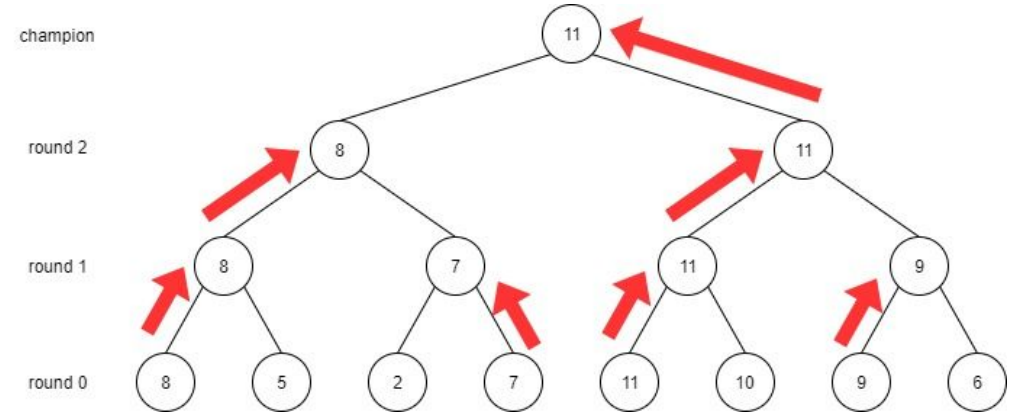
Writer: Mr. Chan Pruksapha

The contents is clickable :)

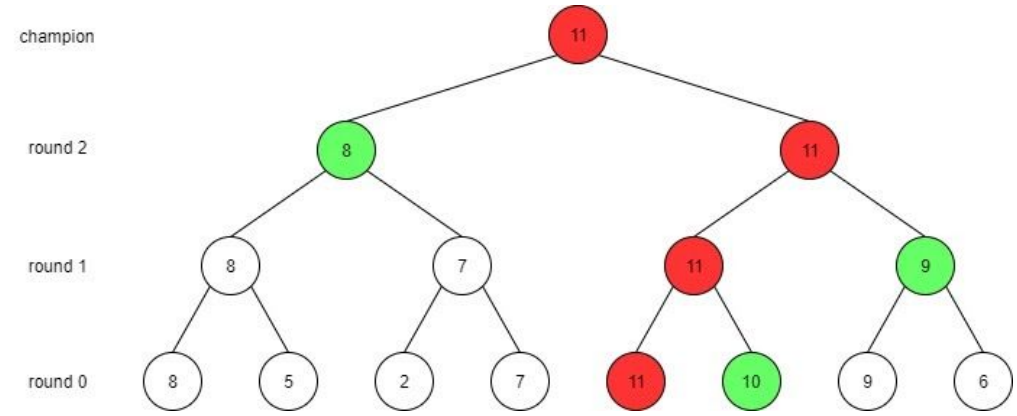
Link to all of the code being described in this documentation:
https://github.com/Chan-prksa/2ndMAX/tree/sliding_window.
Or, you can use links at page 3.

Overview

Phase 1



Phase 2



In this documentation, I will walk the reader through one variant of methods that can find the second maximum number in an array[1]. The method I use consists of two phases. In phase one, I recast the problem of finding the greatest (or maximum) number as finding a champion in single-elimination tournament, in which the winner of a match will climb up the tree.

The second phase is based on this observation: the second maximum will not lose to any other contestant except the champion and it will keep climbing the tree until it eventually has a match with the champion and lose. Because of this, all we have to do is to collect all contestants who lose to the champion at some point during the tournament. These collected

[1] According to the Python official documentation, “Python’s lists are variable-length arrays.” https://docs.python.org/3/faq/design.html#how-are-lists-implemented-in-cpython

contestants are candidates for being second maximum. Just find the maximum number among the candidates and we're done.

Restrict my Documentation to input of size 2^n

For the sake of simplicity of this documentation, I will restrict to the case when input size = 2^n only[2]. Under this condition, in every round there will always be an even number of contestants. This is important because if there is a round in which there are an odd number of contestants, then there will be one contestant who doesn't have an opponent because everyone else already have been paired with his/her own opponent.

In my implementation, however, I create a version that can deal with both an odd/even number of contestants. Curious readers are invited to check out this code: oddNumberOfNode.ipynb.

For the link to the code version that deal with an input of size 2^n only, the reader can check it out from this link powerOfTwo.ipynb. **From now on until the end of this documentation, pieces of code that are shown are all based on this version of code.**

The full links to the two versions of code above are:

- https://github.com/Chan-prksa/2ndMAX/blob/sliding_window/powerOfTwo.ipynb
- https://github.com/Chan-prksa/2ndMAX/blob/sliding_window/oddNumberOfNode.ipynb

[2] As a matter of fact, the requirement that input size is an even number isn't enough to prevent the situation that there are odd number of contestants in some further round. Consider an input with size of even number 2^n*k where k is an odd number. In the round zero a number of contestants will be 2^n*k but in round one the number will be reduced by half to 2^{n-1}*k. In next rounds number of contestants will be 2^{n-2}*k, 2^{n-3}*k, and so on. Eventually, in the round n^{th}, there will be k contestants.

Preparation

Coding Block 1

```
class Node:
    def __init__(self, val, id, left_child, right_child):
        self.val = val
        self.id = id
        self.left_child = left_child
        self.right_child = right_child
```

Coding Block 2

```
def buildTreeAndFindMax(leaf_nodes):
    ...
    return champion
```

Coding Block 3

```
input = [8,5,2,7,11,10,9,6]
champion = buildTreeAndFindMax(input)
```

Code Block 1 Node wrap a number and provide additional field 1) `self.val` represents the value of the number itself and will be used to compare in a match (which node with greater value win.) 2) `self.id` **is a unique identifier of each contestant. It will be used to confirm if two nodes represent the same contestant even though their positions in the tree are different. This variable will be useful when we reach phase 2.** `self.left_child` and `self.right_child` are children of Node.

Code Block 2 and 3 Function `buildTreeAdnAndFindMax` will return a node with maximum `self.val` (call it `champion`!), and along the way will also build the tree. Inside this function, `input` (a list of numbers) will be bound to a variable name `leaf_nodes`.

Phase 1: Building Tree and Find the Champion (Bottom Up Approach)

Round 0 & Initialization

Code Block 4

```
def buildTreeAndFindMax(leaf_nodes):
  round = 0
  round_list=[]
  # Transform a list of numerical values into a list of object from class 'Node'.
  # These newly created nodes will form leafs of our tree.
  round_list.append(
                  [ Node(val, id, None, None) for id,val in enumerate(leaf_nodes)]
                   )
```

In this context, round 0 means the competition haven't started yet. As shown in **Code Block 4**, a list of nodes, which represent initial group of contestants in round 0, is built from an argument `leaf_nodes` of the `buildTreeAndFindMax` function. Because these nodes will be at the bottom most part of the tree (they are leafs), and because these leaf nodes are a starting point for building nodes in higher positions in the tree, I decide to call it a ***bottom up approach***.

The reason why nodes in each round need be contained in a list is because these nodes will be accessed by using sliding window as shown in **Code Block 5.** Back to **Code Block 4,** after `round` *is initialised to be zero,* and after `round_list` append a list of leaf nodes. We have

```
round_list[round] == round_list[0]
== reference to [ Node(val, id, None, None) for id,val in enumerate(leaf_nodes)]
```

Build Round 1 from Round 0

When entering **Code Block 5** for the first time, `round_list[0]`, which contains leaf nodes, was assigned to a new variable `current_round_list`. So, `current_round_list` is responsible for nodes in round 0. Similarly, at this time `next_round_list` is in charge with taking care of nodes in round 1. This is because the third line `next_round_list = round_list[round+1]` in **Code Block 5**. However, the second line - `round_list.append([])` - means that, currently, `round_list[1]` (the current `round_list[round+1]`) is just an empty list awaiting to be filled.

Then, inside the loop `while(i < N):`, a sliding window of length two whose stride equals to two (because of `i += 2`) iterates over `current_round_list`. Every time the sliding window enclose a pair of nodes, `self.val` will be compared and the winner is the node that its `self.val` is greater than that of its opponent. Lastly, a newly created node `winner_node` is built based on winner from round 0 by copying `self.val, self.id`. and both nodes from round 0 enclosed by the sliding window are set to be children of this new winner. When everything is ready, the winner node is added to `next_round_list` and **this how round 1 goes from being**

empty to being full of nodes. Moreover, Because both `next_round_list` and `round_list[1]` contain a reference to the same list. **round_list[1] contains the same set of winner nodes.**

Code Block 5

```
current_round_list = round_list[round]
round_list.append([])
next_round_list = round_list[round+1] # queue for next level

N = len(current_round_list)
i = 0
while(i < N):
    #sliding window of length 2
    first_node = current_round_list[i]
    second_node = current_round_list[i+1]
    print( '\tA match between: {}-{}'.format(left_node.val, right_node.val))

    if first_node.val >= second_node.val:
        winner_node = Node(first_node.val, first_node.id, first_node, second_node)
    else:
        winner_node = Node(second_node.val, second_node.id, first_node, secode_node)

    # advance the winner to next round
    next_round_list.append(winner_node)
    i += 2 #stride equals to two
```

Figure 1

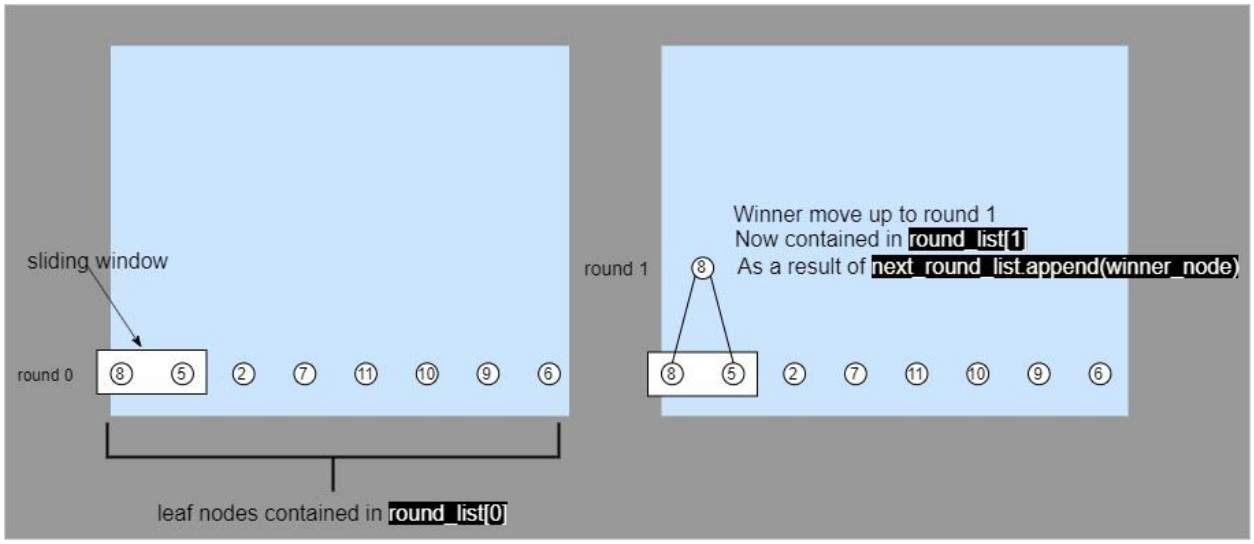


Figure 1 Illustrates what happens after initialization take place in **Code Block 4**, and also after the winner was append to round 1's list. Here, a sliding window encloses nodes with value 8 and 5 and Nodes with value 8 is a winner.

Figure 2

Remember our familiar "for loop"
`for (int i = 0; i < numbers.length; i++)`?
The condition for the loop to stop is when the counter `i` is no longer less than an array's length. In **Code Block 5** terminating condition is `while(i < N):`. This is much like the “for loop” because we compare counter i with list’s lenght `N = len(current_round_list)`.[3]

Figure 2 illustrates how all nodes in round_list[1] come from matches in round 0.

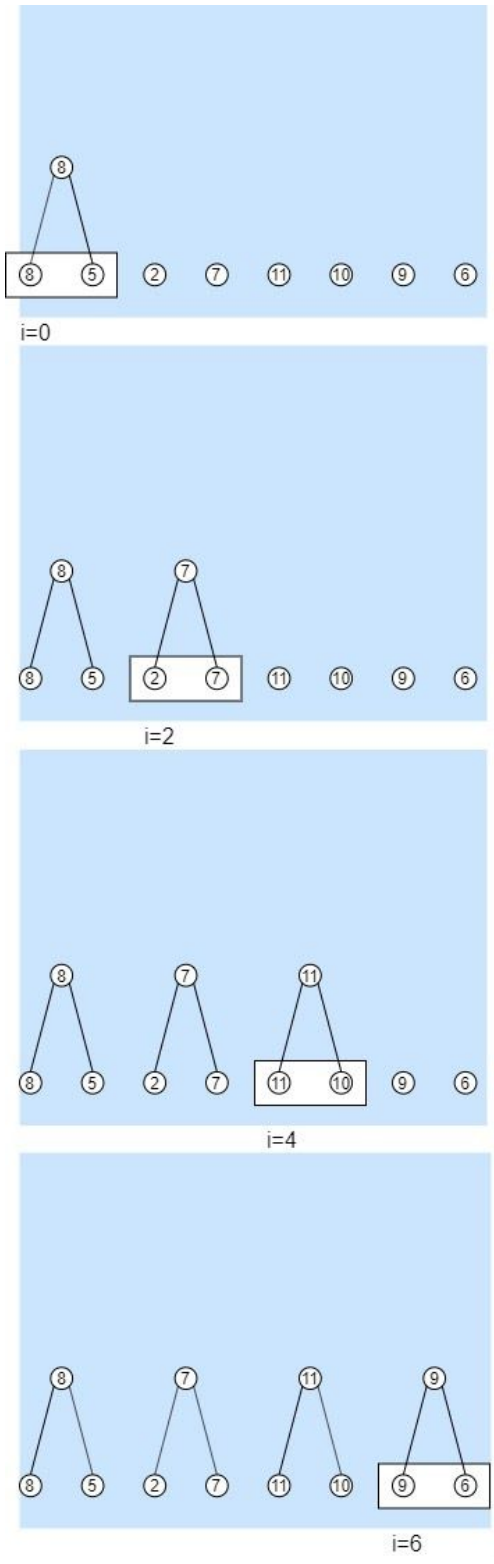
In a match between 8 and 5, 8 is a winner so it is added to round 1 with child nodes 8(left child) and 5(right child).

In a match between 2 and 7, 7 is a winner so it is added to round 1 with child nodes 2(left child) and 7(right child).

In a match between 11 and 10, 11 is a winner so it is added to round 1 with child nodes 11(left child) and 10(right child).

In a match between 9 and 6, 9 is a winner so it is added to round 9 with child nodes 9(left child) and 6(right child).

As shown at the bottom of figure, there is no node at i=8 in round 0.



[3] The fact that the length of `current_round_list` is divisible by the length of sliding window and that the length of sliding window equals stride is a sufficient condition.

Building Next Rounds

Code Block 6

```
while(True):
    current_round_list = round_list[round]
    round_list.append([])
    next_round_list = round_list[round+1] # queue for next level

    N = len(current_round_list)
    i = 0
    while(i < N):
        ...
        i += 2 #stride equals to two

    if len(next_round_list) == 1:
        # champion will act as a parent when finding 2nd max candidates
        champion = next_round_list[0]
        break
    else:
        round += 1
```

In the last section, **Code Block 5** demonstrate how to build round 1 by collecting winners from round 0. By analyzing **Code Block 6** in this section, I hope the reader will understand that the same will happen to all the next rounds, i.e., round 1 build round 2, round 2 build round 3, and so on.

Suppose that condition of `if` statement is met, then the program will break `while(True):`. So, the chunk below contains an important terminating condition worth talking about. By the way, **that termination will happen only after building round 3 from round 2 is finished**. I will come back to this terminating condition and the building of round 3 nodes together in the next section.

```
    if len(next_round_list) == 1:
        # champion will act as a parent when finding 2nd max candidates
        champion = next_round_list[0]
        break
```

First of all, when the program breaks out of the loop `while(i < N):`, it's already finished building round 1's nodes from round 0. Next, because terminating condition for `if` will not be met now, the program enter `else:` block (shown below) and `round` variable increment by one. Formerly, round was zero. Now it is one.

```
    else:
        round += 1
```

Next, the program enters a new iteration of the loop `while(True):`. Then, program run the below code chunk that I explained before (in section “Build Round 1 from Round 0”.)

```
    current_round_list = round_list[round]
    round_list.append([])
    next_round_list = round_list[round+1] # queue for next level
```

What has changed is that `current_round_list` now keep nodes in round 1 rather than round 0. This is because it was assigned `round_list[round]` and now `round==1`. It also is useful to remind the reader that nodes in round 1 kept by `current_round_list` used to be winner nodes when building round 1 from round 0. As for `next_round_list`, it responsible for round 2 but it is now just an empty list.

The below code shunk (**Code Block 7**) will do just like before: when program enter `while(i < N):` winner nodes from `current_round_list` (round 1) are selected and append to `next_round_list` (round 2)

Code Block 7

```
    N = len(current_round_list)
    i = 0
    while(i < N):
        ...
        i += 2 #stride equals to two
```

Figure 3

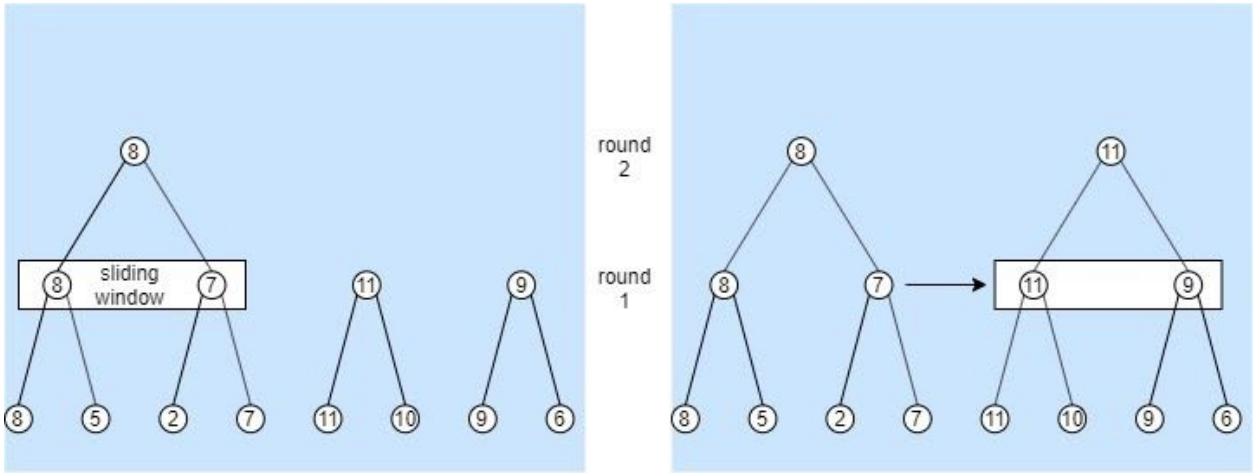


Figure 3 illustrates the result we’ve got after program finish building round 2 from round 1.

Terminating condition

Let's continue from where we left off in previous section. As explained in a above, program run **Code Block 7** to build round 2 nodes from round 1 nodes. Once program finish building round 2 nodes, it break while-loop `while(i < N):`, and reach this `if-else` block again.

Code Block 8

```
    if len(next_round_list) == 1:
        # champion will act as a parent when finding 2nd max candidates
        champion = next_round_list[0]
        break
    else:
        round += 1
```

For a reason that will be cleared soon in this section, the terminating condition is not met this time. So, this is what program will do next:

- enter `else` block
- variable `round` increment to 2
- the program enters a new iteration of the loop `while(True):` in **Code Block 6**

Just like the previous section, when program enter the loop `while(i < N):` winner nodes from `current_round_list` (round 2) are selected and append to `next_round_list` (round 3)[4]

Figure 4

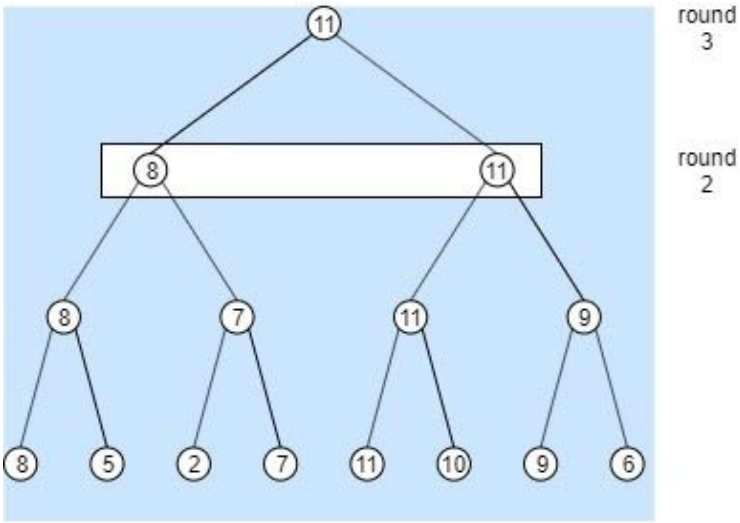


Figure 4 illustrates the result we've got after building round 3 is done and the program break the loop `while(i < N):`. It's important to notice that now the round 3 only has one node. So this node is a root node of the tree, another important fact is that its `self.val` is a maximum value in comparison to all other nodes in this tree. Next, the program will enter `if-else` block (**Code Block 8**) again. **It turns out that this time the terminate condition of `if` block is satisfied because the condition is `len(next_round_list) == 1`.**

[4] Building round 3 from round 2

Returning the Champion

Code Block 9

```
def buildTreeAndFindMax(leaf_nodes):
    ...
    champion = None # for keeping the node which has maximum value
    while(True):
            ...

        while(i < N):
            ...
            i += 2

        if len(next_round_list) == 1:
            # champion will act as parent when finding 2nd max candidates
            champion = next_round_list[0]
            break
        else:
            round += 1

    return champion
```

Code Block 9 give an outline of the whole function `buildTreeAndFindMax`. `If-else` block is in scope of the loop `while(true):`. However, after the terminating condition is met the root node of the tree, which possesses the largest `self.value`, is assigned to variable `champion` before the program break from while-loop. After that the program continues executing the code after the loop, that is `return champion`.

Phase 2: Collecting Candidates (Top Down Approach)

As explained in the overview section, The node whose `self.val` is a second largest number must have had a match against champion and lost. So, a logical way to find the second maximum nodes is to collect all the nodes which have had a match against champion before. I will simply call those collected nodes as candidates.

Basically, a winner in a match between two contestants is nothing but a contestant whose `self.val` is greater than his/her opponent. This fact is illustrated in a part of **Code Block 5** below. If *`first_node`* win[5] `second_node`, `winner_node` copy `first_node.val` and `first_node.id`. Similarly, if `second_node` win, then `winner_node` copy `second_node.val` and `second_node.id`. This is highlighted by using yellow character in the code block.
Also note that, when creating `winner_node`, two nodes in a match which compete to be a winner are also passed into constructor arguments: `left_child` (for `first_node`), `right_child` (for `second node`). This is highlighted in light blue characters.

Part of **Code Block 5**

```
    if first_node.val >= second_node.val:
        winner_node = Node(first_node.val, first_node.id, first_node,
second_node)
    else:
        winner_node = Node(second_node.val, second_node.id, first_node,
second_node)
```

In part of **Code Block 1**, signature of the class Node's constructor is provided below to remind the reader about the order and names of arguments.

Part of **Code Block 1**

```
class Node:
    def __init__(self, val, id, left_child, right_child):
```

Identify Losers & Track a Path of Winners

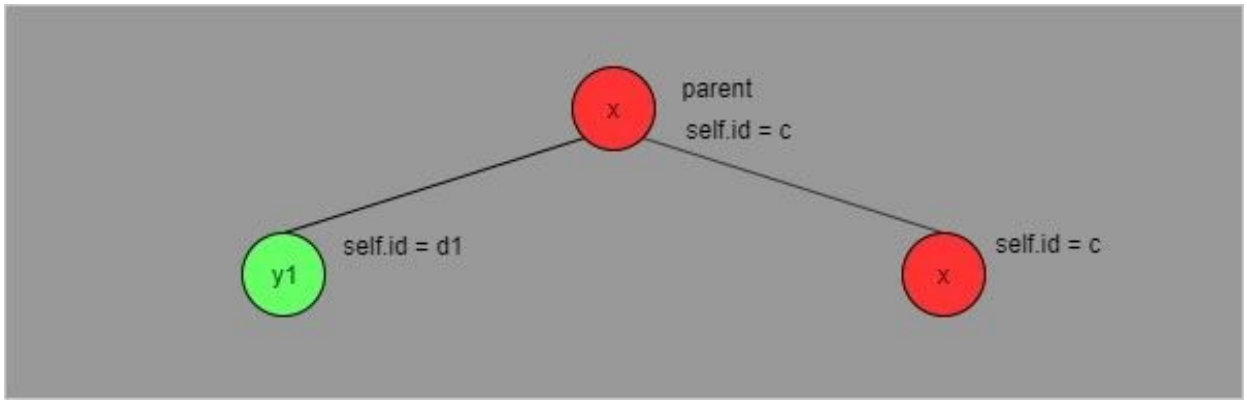
If a node is given to us and we know it is a `winner_node` obtained from some match between two contestants we know nothing about, this information can be recovered by accessing `winner_node.left_child` and `winner_node.right_child.`. Among these two nodes there must be one of them which has the same `self.id` as `winner_node`.**This child node which share its `self.id` with `winner_node` and `winner_node` itself actually both represent the same contestant.** This fact has two useful implications.

Firstly, it helps identify which child node is a loser. To be more specific, let's have a look at **Figure 5** on the next page. The figure illustrates a `winner_node` (node x at the topmost) as a parent node with two children. Because the right child node has the same `self.id` as the parent node. The right child must have won the left node (node y1) and move up into the

[5] The term "win" here means that a node have greater value of `self.val` than its competitor.

parent's position. In short, if right child (or left child) node has the same `self.id` as parent, then it is safe to say that left child (or right child) is a loser node.

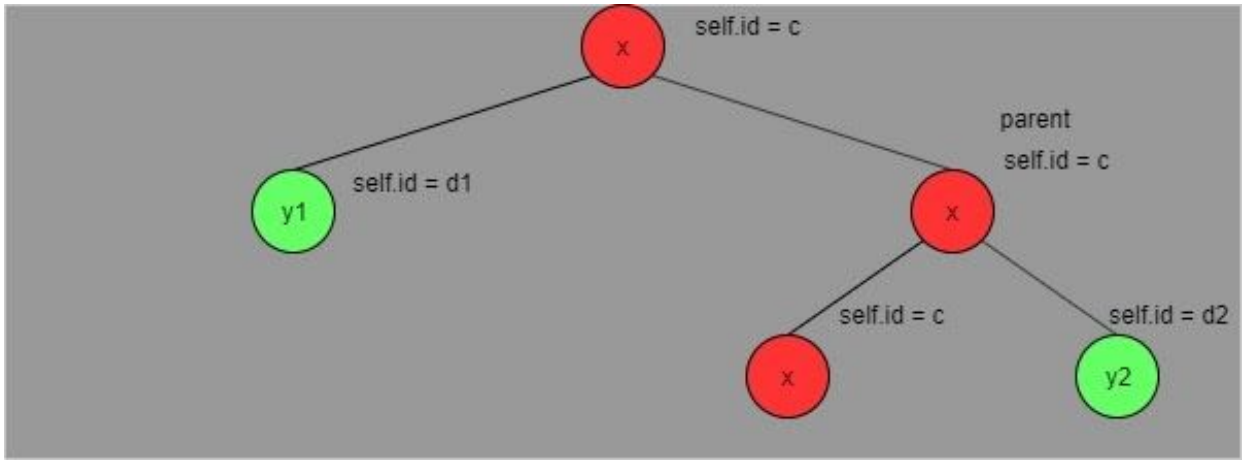
Figure 5



Secondly, `self.id` can help us track the positions of any given `winner_node` node. For example, let's have a look at **Figure 6** below. Previously in **Figure 5**, we've found that the right child node of the *topmost node x* (the one we label as *parent*) also had `self.id = c` and so *this second node x* also represents the same contestant as the topmost *node x*. Now suppose that the second node x isn't a leaf node. Then, the second *node x* must have two children. By labeling this node as parent and repeat the same process of accessing the node's children, we will get one child with `self.id = c` and one child with `self.id != c`. As illustrated by **Figure 6**, what we can achieve are:

1. Discover the third nodes x, which represent the same contestant as other two node x but competing in an earlier round. **If we repeat the process again and again we will get the full information that tell us where node x has been earlier before it move up to positions in Figure 6**.
2. Discover more nodes which has a match with the third node x. **So, we can collect more candidates(node y2)**. Moreover, at the same time as we discover more node x in earlier and earlier round, we can just collect those opponents of node x — in various positions — as candidates.

Figure 6



Code

Code Block 10

```
def secondMaxCandidates(parent):
    candidates = []

    while(parent.left_child or parent.right_child):
        left_child = parent.left_child
        right_child = parent.right_child
        if parent.id == left_child.id:
            candidates.append(right_child)
            parent = left_child
        elif parent.id == right_child.id:
            candidates.append(left_child)
            parent = right_child

    return parent, candidates
```

Code Block 11

```
leaf, candidates = secondMaxCandidates(champion)
```

In **Code Block 10** function `secondMaxCandidates` will does the task of collecting losers of any winner node that is passed to the function argument `parent`. In **Code Block 11**, I pass `champion` from Phase 1 to argument `parent` of `secondMaxCandidates`. The reason is obvious: I want to collect all losers of `champion` into `candidates` list.

Next, I want to match the concept explained in the immediately preceding section to relevant part **Code Block 10.**

When I talk about identifying loser in the prior section I said:

> **In short, if right child (or left child) node has the same self.id as parent, then it is safe to say that left child (or right child) is a loser node.**

This corresponds to the part of **Code Block 10** that is highlighted by using yellow character. Note that by appending a node to a list `candidates` is the same as saying that it is a loser.

When I talk about tracking a path of winner node, I mention how to continue going down from the current position in a tree (some current round in competition) to lower positions (earlier rounds in competition) in order to track the nodes whose `self.id` is the same as the topmost winner node.

> **Then, the second *node x* must have two children. By labeling this node as parent and repeat the same process of accessing the node's children…**

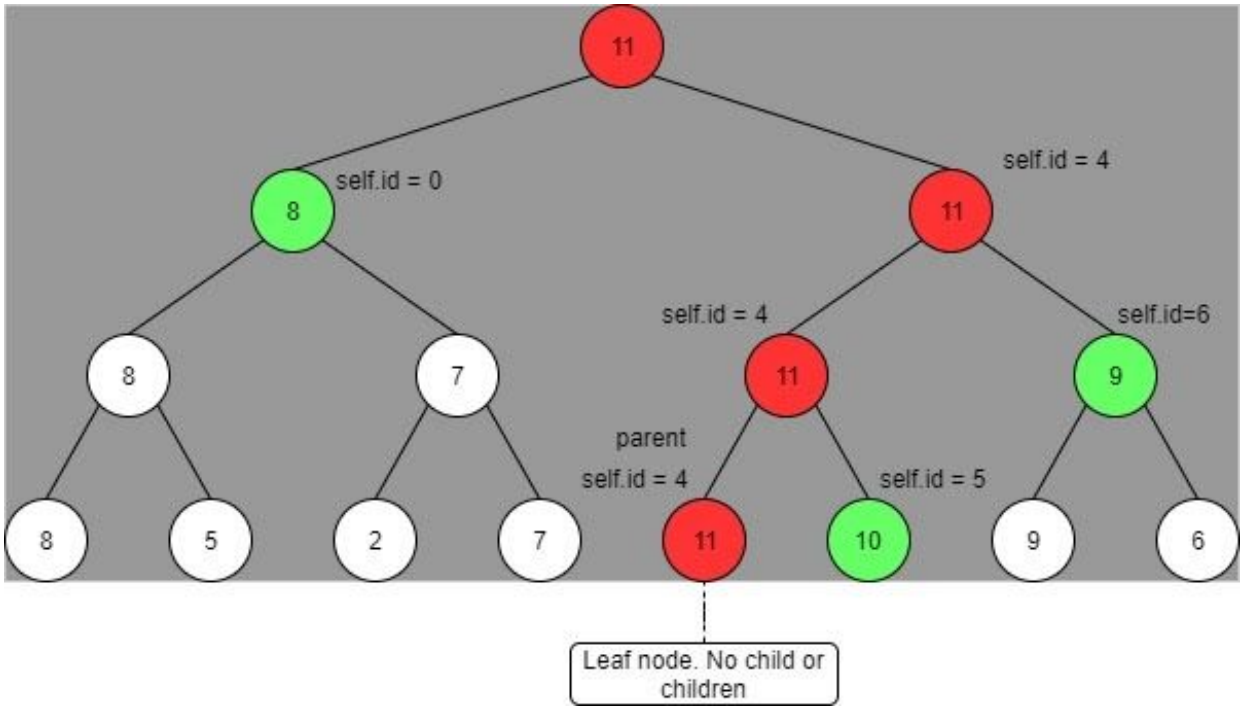
This corresponds to the part of **Code Block 10** that is highlighted by using red character.

```
while(parent.left_child or parent.right_child):
```

As for terminating condition for while-loop, it will be triggered after we keep labeling new nodes as parents on and on and on until this process reach a leaf node. Because leaf nodes have no children. The condition `parent.left_child or parent.right_child` will be false.

The reason we need to go deep down to the leaf node is illustrated in **Figure 7.** In this example, node 10(`self.id = 5`), which actually be the second maximum node, will not even be collected into candidates list, if we aren't aware that it has a match with node 11(champion, `self.id = 4`) while both are still leaf nodes.

Figure 7



Find Maximum Number among Candidates

To finish off the task of finding the second maximum number, we only need these two lines of code.

```
candidates_val = list(map(lambda nd: nd.val, candidates))
max(candidates_val)
```