DATA STRUCTURING CASE STUDIES

Case Study Two (II)

Finding the Maximum and Next Maximum of n integers

From the problem of obtaining the max and min in CASE study one; we can obtain the maximum of n numbers in n-1 mandatory comparisons; one approach to obtain the second max is to compare the remaining n-1 elements. This approach will obtain the max and second max in $n - 1 + n - 2$ comparisons i.e. $2n - 3 \approx 2n - 2$

Here is a function that takes an array of n integers and displays the maximum and second maximum:

```
void getMaxAndNextMax(int numbers[])
{
      int n=sizeof(numbers);
      int max, secondMax;

      if(numbers[0]>numbers[1])
      {
            max=numbers[0];
            secondMax=numbers[1];
      }
      else{
            max=numbers[1];
            secondMax=numbers[0];

      }

      for(int i=2; i<n; i++){
            if(numbers[i]>max)
                  max=numbers[i];
            else if(numbers[i]>secondMax)
                  secondMax=numbers[i];
      }

      cout<<"The maximum  is: "<<max;
      cout<<"The Next Max is: "<<secondMax;

}
```

NOTE

If you had an array `numbers={60,58,2,4,25,56,45}`
The number of comparisons will get to $2n - 2$ ……. Why?

This is the worst case scenario. This is the kind of value we struggled to reduce in our first case study; the question is, can we have a better value i.e. can we have an algorithm that never gets to this value?

A Deeper Look at the max – next_max problem

Question; is there a property of the next maximum (second maximum) that could help us obtain that number without having to do another n-1 comparisons. We must, of necessity, take n-1 comparisons to obtain the maximum but can we do something during the process of obtaining second maximum that will reduce these comparisons?

NOTE

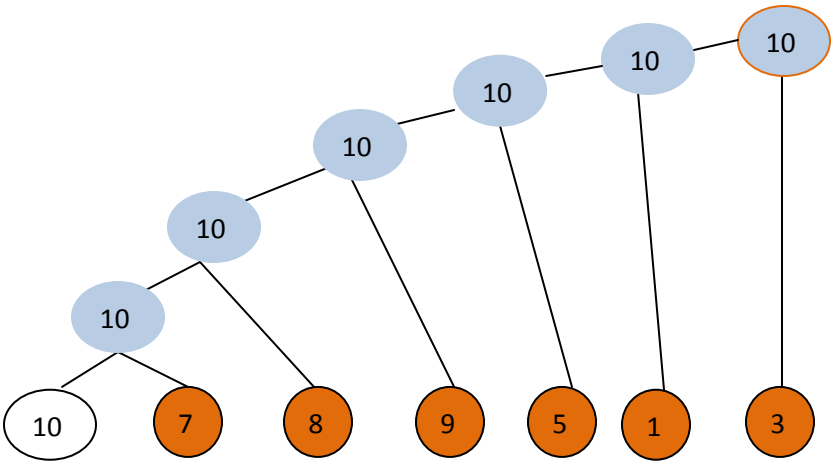
The one thing that is sure about the second maximum number is that it must have lost to the maximum number along the way. The second place in a tournament is taken by the player who loses to the winner.

i.e. we cannot declare a second place or second max unless we somehow compared it to the max or they competed with the ultimate winner and lost.

With this fact in mind then we know that to obtain the second max; we need to compare only those numbers that competed directly with the maximum all its comparisons. The second maximum must be among those numbers that ***directly*** competed with the winner and lost.

The task is to formulate this set and minimize it as much as possible.

Now let's look at the two structures we had previously and try to obtain this set from both.



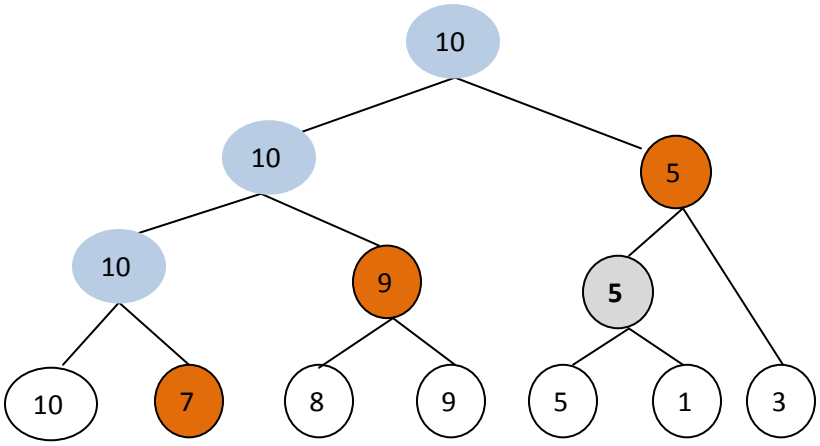
Approach A

Watch the winner and pick the players who competed directly with it; in this example: our winner is 10, giving us a set of:

SetOfSecondMax = **{7, 8, 9, 5, 1, 3}**

The set of those players who competed directly with the winner and lost is **n-1**

If we structured our tournament in this manner, then we have to carry out at most **2n-2** comparisons to obtain both the max and second max; this is the worst case.



Approach B: Pair wise comparison

A balanced tree….
Watch the winner and pick the players who competed directly with it; in this example: our winner is 10, giving us a set of:

SetOfSecondMax = **{7, 9, 5}**
Try this for 16 numbers. The number of elements in this set is obtained from the longest path in the tree.

e.g. for 8 numbers the set has 3 numbers for 16 numbers, the set would contains 4 numbers.

How do we generalize this for any n numbers?

8 is the 3rd power of two and 16 is the 4th power of 2; so we can deduce a relationship here:

$\mathbf{2^k = n}$; where k is the number of elements in the set for next maximum. If we solve for **k:**

$$\boldsymbol{k = log_2 n}$$

Therefore the total number of comparison required in this Approach is $\boldsymbol{n - 1}$ for obtaining max together with $\boldsymbol{log_2 n}$ for obtaining second max.

$$\boldsymbol{total\ time} = (\boldsymbol{n - 1}) + (\boldsymbol{log_2 n - 1})$$

$$= \boldsymbol{n + log_2 n - 2}$$

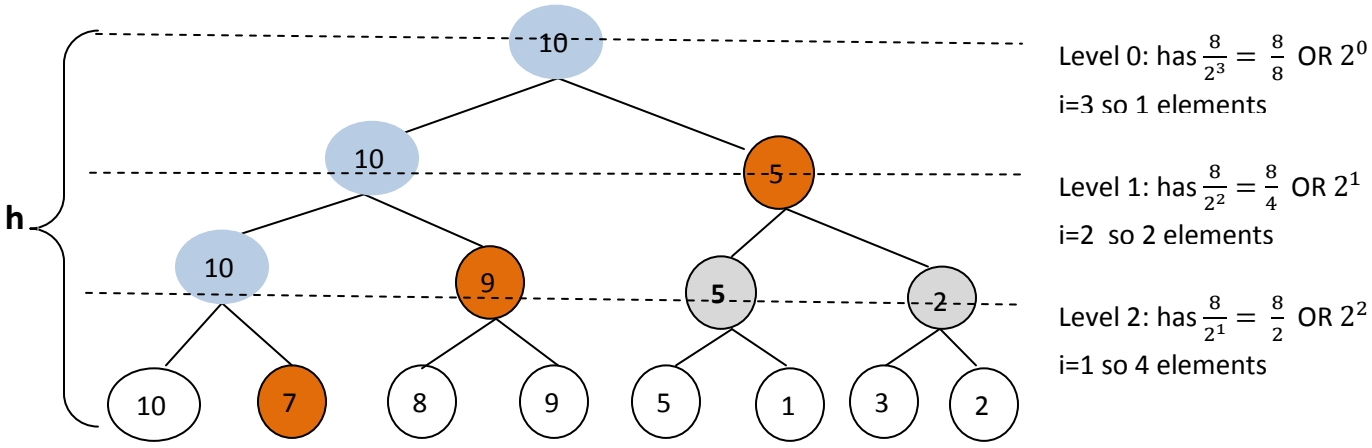
Which is much better than $\boldsymbol{2n - 2}$ obtained in approach A.

Observations

Depending on how we structure our tournament, we can reduce the number of comparisons hence increase program efficiency (running time and even space). We need to have a balanced tree to be able to minimize the set of these elements to be compared to obtain second max. If we structured our tournament as pair wise comparison then our tree is balanced and we have a way less number of comparisons.

How do we represent this Structure (Such that we can tell which numbers compared the max)?

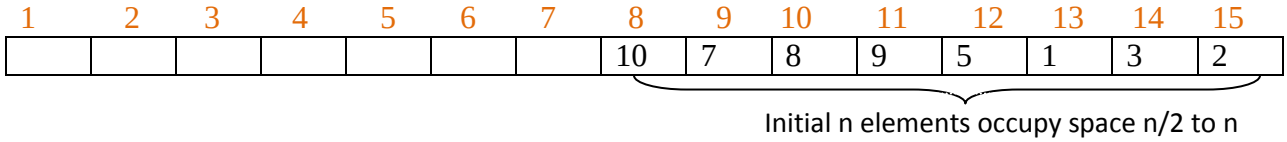
To build this tournament we first of all need space for the $\boldsymbol{n}$ numbers, so initially the array must have at least $\boldsymbol{n}$ spaces for just the numbers. W then need $\boldsymbol{n/2}$ spaces to hold the winners of the first comparisons, n/4 spaces for the winners of the next stage, n/8 for the subsequent stages n/16 and so on and so forth. We require $\frac{\boldsymbol{n}}{\boldsymbol{2^i}}$ spaces for every subsequent stage; where i is the inverse level of the tree.



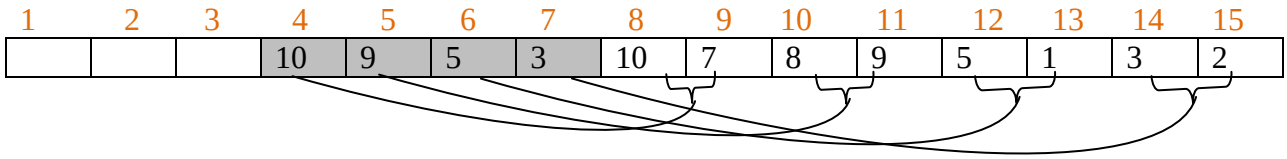
If we add all the elements at the derived levels (from level 0 to level h-1); we find that they are n-1 (in our case: $\boldsymbol{8-1=7}$); we therefore need $\boldsymbol{n-1}$ extra spaces in the array in addition to the initial $\boldsymbol{n}$ spaces for the elements. This gives $\boldsymbol{2n-1}$ spaces required.

The Array Representation

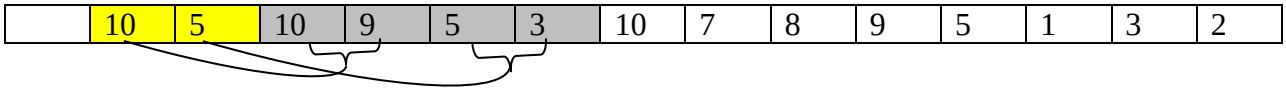
1. In or case when we have 8 elements we need 15 ($\boldsymbol{i.e.2*8-1}$). Then we represent the top of the tree as the first element in the array; this would mean that the initial $\boldsymbol{n}$ numbers are stored from index $\boldsymbol{n/2}$ to n.



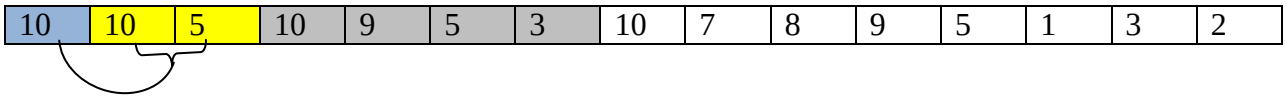
2. We now carry out the first pair wise comparison storing the elements in the right free spaces: comparison between element number 14 & 15 will be stored in index 7 (14/2); likewise comparison between element 12 & 13 is stored in index 6 (12/2) etc.



3. We take the second pair of comparisons as follows:



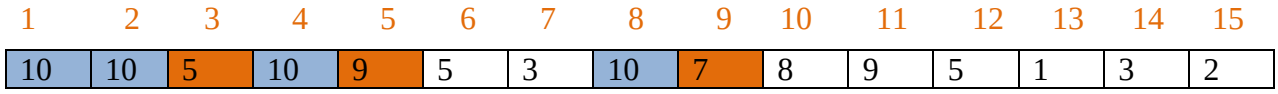
4. These pair wise comparisons continue in like manner until we get to the last comparison and we obtain the maximum.



At this point we have built the tournament and somehow we can just pick the max; i.e. the number at the top of the structure (the first index in the array). We return **10** as the maximum.

Obtaining the Second Max

Now that we know the maximum; the first step is to find a way of getting those numbers that competed directly with the maximum, the second maximum belongs to this set



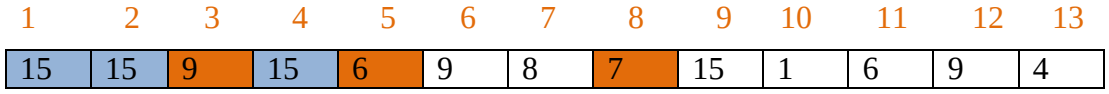
Is there an attribute that can help us obtain these numbers: {5, 9, and 7}?

Notice that if we double the index of our maximum, we get to the index where it next appears, then we can obtain the number it competed with.

Let $i = 1$ (the index of the maximum)
$i = i * 2$ gives us the next index that could contain the maximum i.e. the maximum is either in an index twice it's current index or an adjacent index.
From here we can tell that the larger number between the i^{th} and the $(i+1)^{th}$ element is the maximum while the other is a candidate for the second maximum. We pick the index of this maximum and place it in i such that if we double I once more then we get to another index where max appears.

We then pick the other element (the lesser one between the i^{th} and the $(i+1)^{th}$) compare it with the current second maximum if it is larger then it's value is stored in the second max and so on and so forth.

When the value of I exceeds the total size of the array then we stop. At this point the second max is already obtained.



LARGE=15

```
i= 2; NEXTLARGE = minimumOf(t[i],t[i+1])=9
i= i*2=4; CANDIDATE = minimumOf(t[i],t[i+1])
      NEXTLARGE = maximumOf(NEXTLARGE,CANDIDATE)
```

……

Pseudo code

```
i= 2; NEXTLARGE = minimumOf(t[i],t[i+1])=9

repeat
      i= i*2;
      IF(t[i]>t[i+1])
            CANDIDATE= t[i+1]
      ELSE
            CANDIDATE= t[i]
            i=i+1

      NEXTLARGE= maximumOf(NEXTLARGE,CANDIDATE)

Until i>n
```

```
void main(){
      int n;
      cout<<"Enter the number of elements [n]"<<endl;
      cin>>n;

      int tourn[n*2];

      //Enter the numbers and store them in space n to n*2
      for(int i=n; i<=2*n-1; i++){
            cin>>tourn[i];
      }

      //Call the function to build the tournament
      bouldTournament(tourn,n);

      cout<<"The Maximum is : "<<tourn[1]<<endl;
      cout<<"The Second Maximum is : "<<getNextMax(tourn,n); /* A call to the
function for next max*/
}

/*Function Definition for  buildTournament (The function that builds the tournament)*/
Void buildTournament(int tourn[], int n){
      int i;

      for(i=2*n-2; i>1; i-2){
            tourn[i/2]=maxi(tourn[i], tourn[i+1]);
      }
}

/*Function Definition for getNextMax (The function that obtains the  second maximum from the tournament)*/

int getNextMax(int tourn[], int n){
      int i=2, next;

      next = mini(tourn[2],tourn[3]);

      while(i <= 2*n-1){
            if(tourn[i]>tourn[i+1])
                  next=maxi(next, tourn[i+1]);
            else{
                  next=maxi(next, tourn[i]);
                  i++;
            }
      }
      return next;
}
```

Exercise

Write the definitions for the functions **`maxi`** and **`mini`** which are utilized by the **`buildTournament`** and the **`getNextMax`** functions respectively. Then try out this code by including the necessary libraries an debug any errors… Test it with some inputs.