DSA PROJECT PRESENTATION

MOUNDS : ARRAY-BASED CONCURRENT PRIORITY QUEUES

ARYAMAN CHAUHAN
ARNAV GUJARATHI
AVYAKT GARG
DEVANSH YADAV
TANUSHI GARG

MOUND STRUCTURE

Mound is a tree of sorted lists. Attached is a graphical visualization for the same in the next slide.

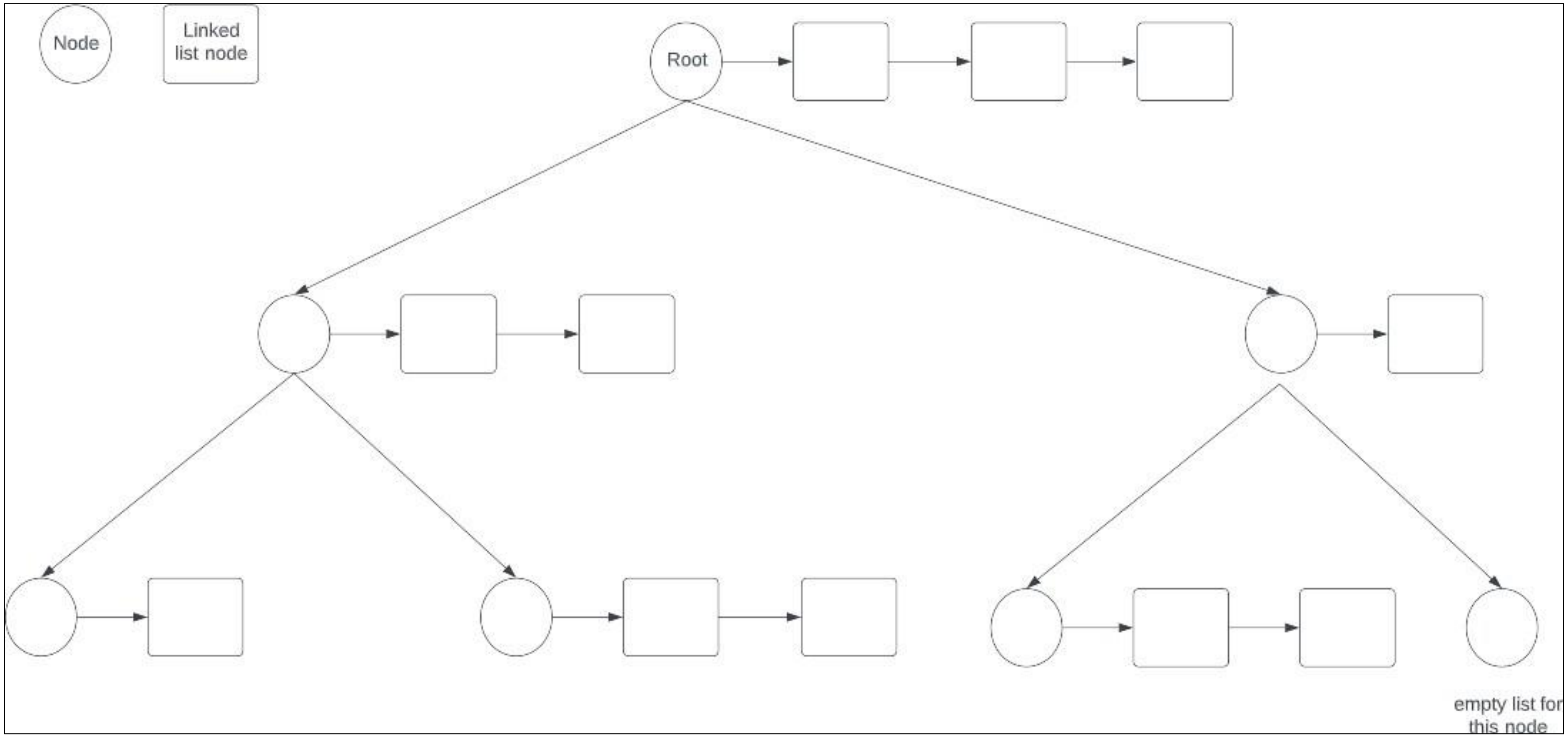
The nodes of this structure store the following parameters -

- A list of sorted values
- A bool variable (Dirty Bit) for checking if inherent property holds true
- A variable for size

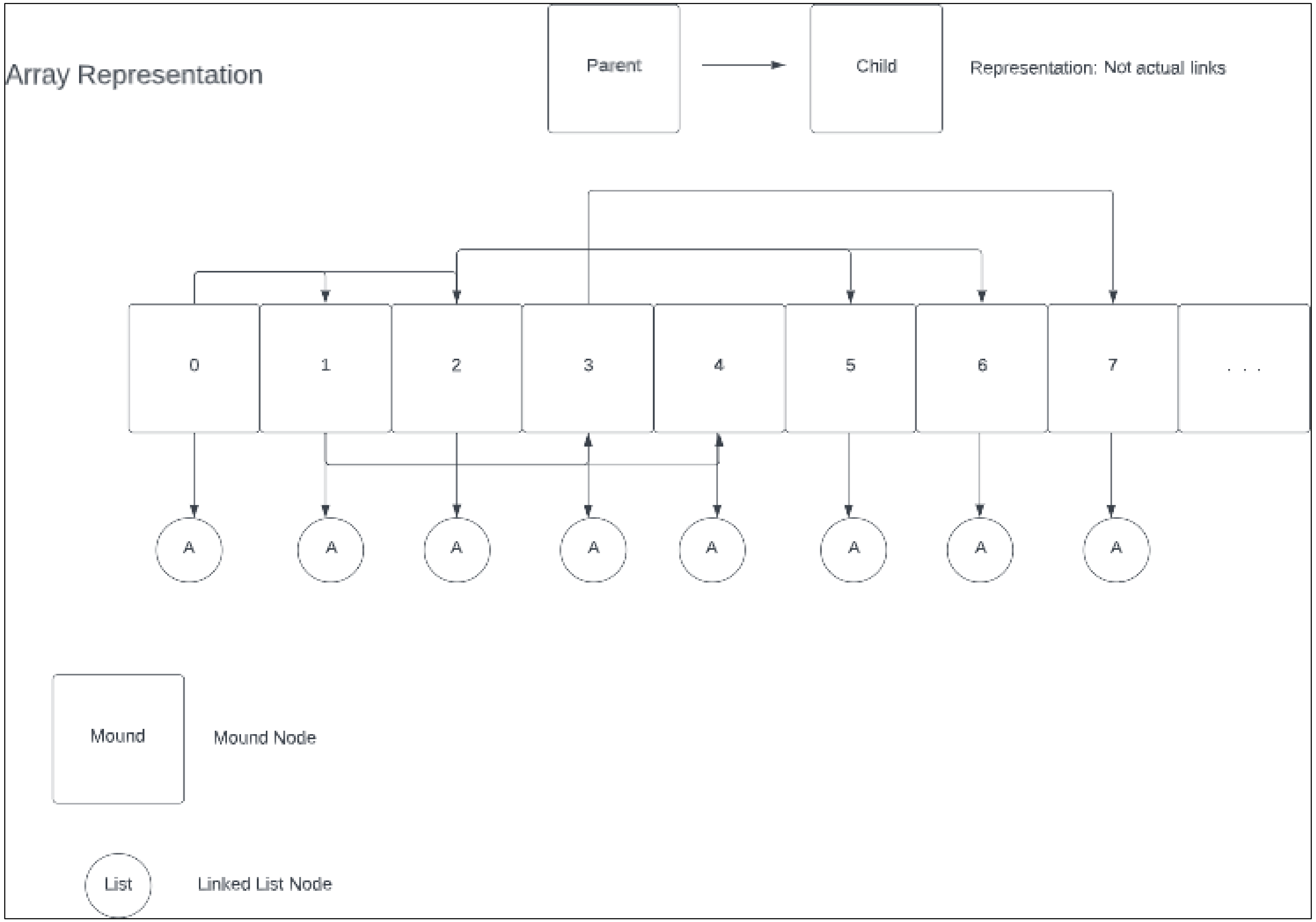
The inherent property of a min-heap is that a child's value must not be less than the value of its parent.

Similarly for mound, 'Dirty Bit' will be set if for a node, there exists a children with a value less than node's value. This also serves as an invariant property for the mound data structure.

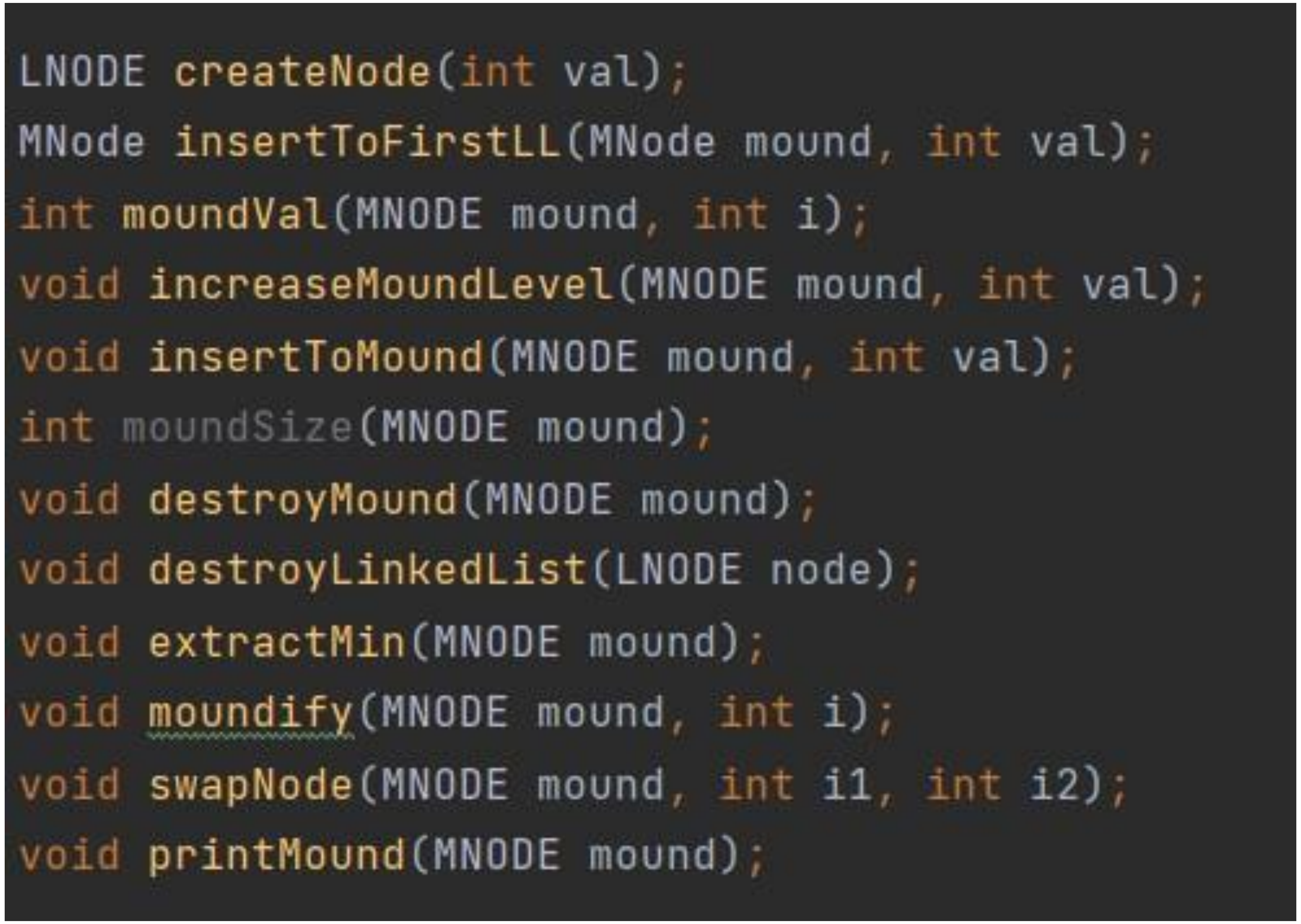
MOUND STRUCTURE



MOUND STRUCTURE



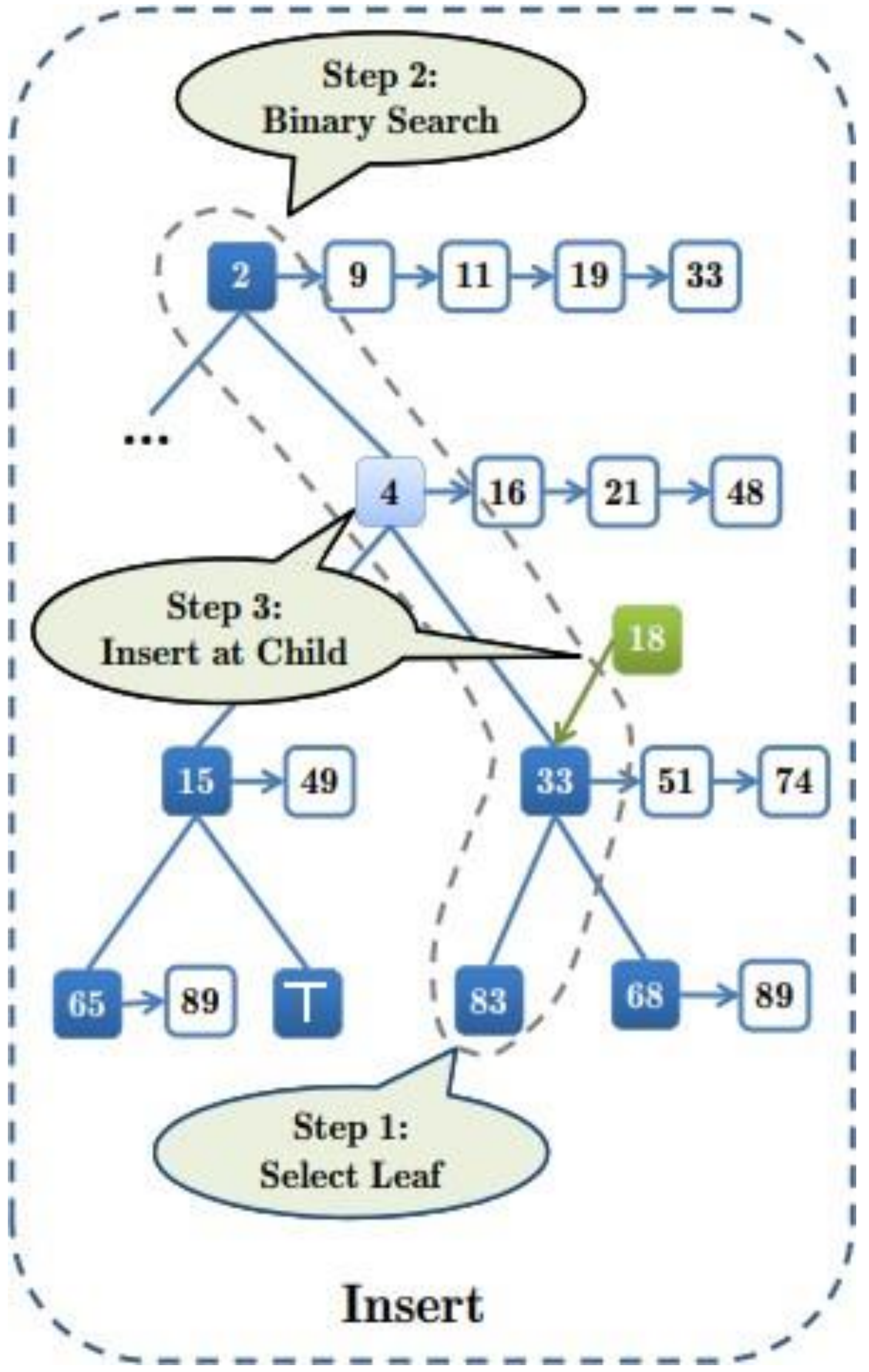
FUNCTION DECLARATIONS

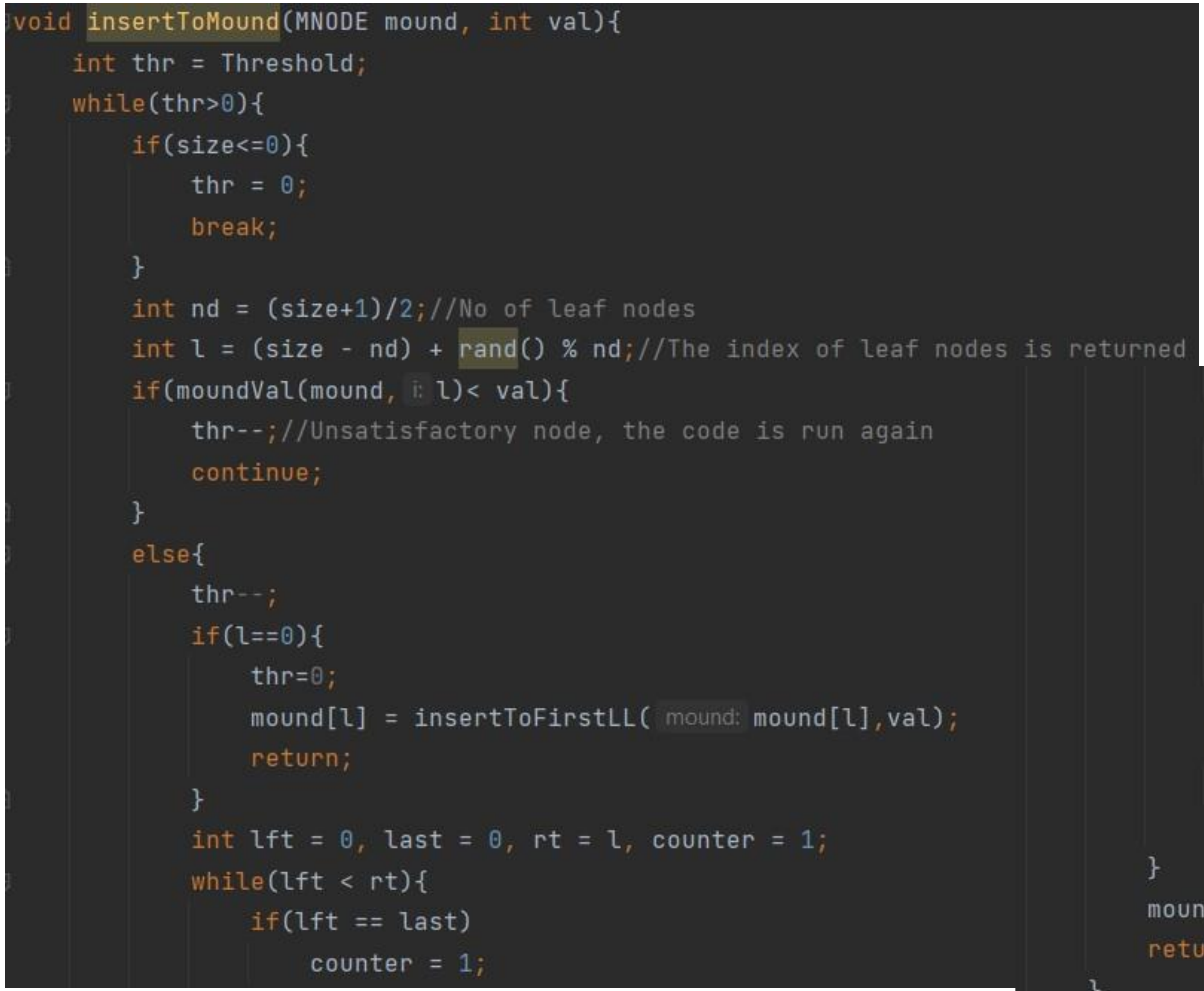


```
LNODE createNode(int val);
MNode insertToFirstLL(MNode mound, int val);
int moundVal(MNODE mound, int i);
void increaseMoundLevel(MNODE mound, int val);
void insertToMound(MNODE mound, int val);
int moundSize(MNODE mound);
void destroyMound(MNODE mound);
void destroyLinkedList(LNODE node);
void extractMin(MNODE mound);
void moundify(MNODE mound, int i);
void swapNode(MNODE mound, int i1, int i2);
void printMound(MNODE mound);
```

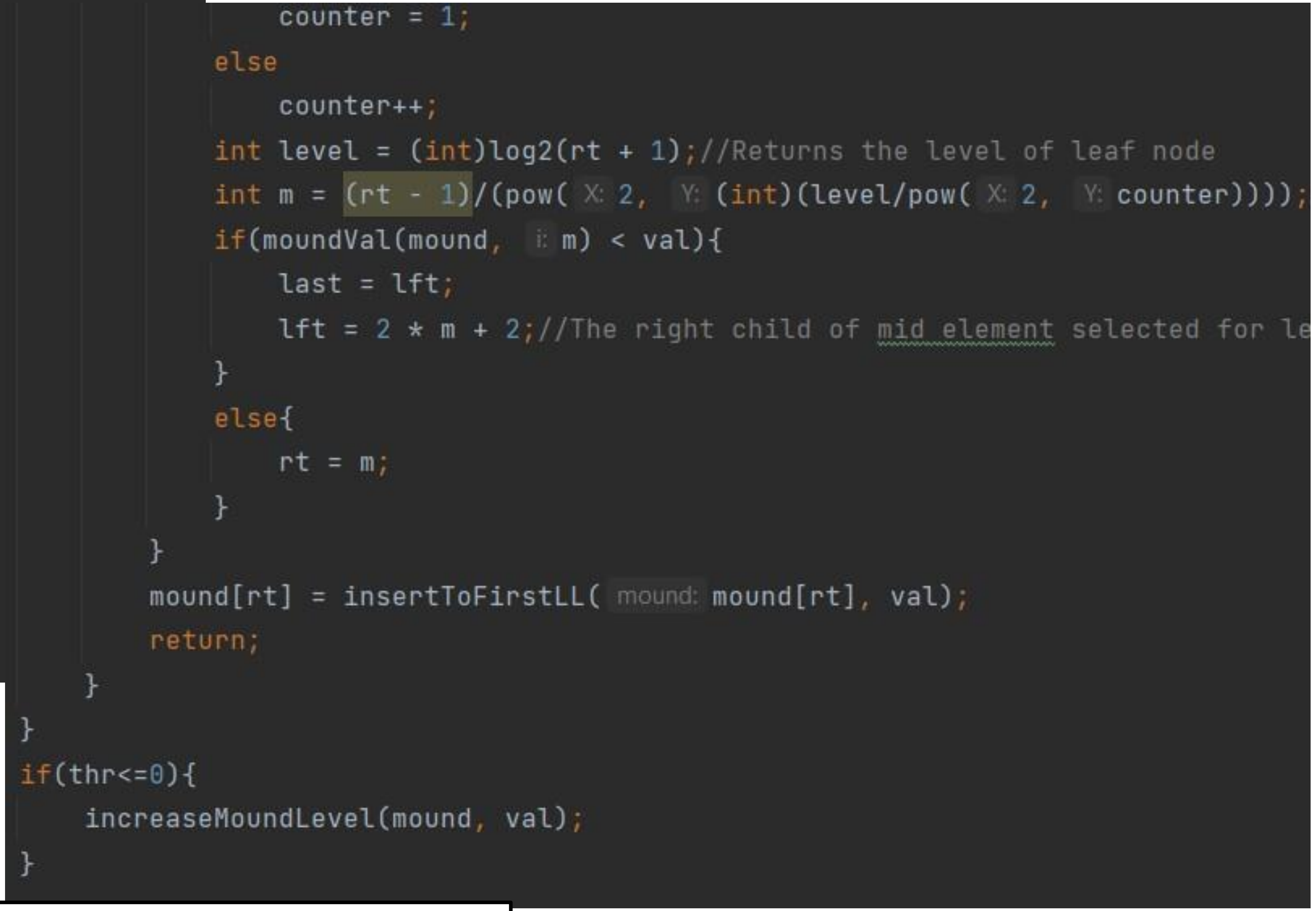
The insertToMound implementation

- We use a threshold variable for checking the while condition.
- Random selection of leaf nodes to search for a valid node for insertions is made until a Threshold is reached. If the Threshold is exceeded, the mound level is incremented by calling the increaseMoundLevel() function and again insertion is tried.
- If Threshold is not exceeded, we check for the size of the mound. If size is 0 then we break out of loop and directly insert at the root using a linked list function
- Else we use a random function to generate the index of one of the leaves of the mound and then check if the node is valid or not
- By validity of node we mean that the value of that node is greater than that of the insertion element and so it will be added in one of its ancestors
- Else we will ignore the selection, and try again taking into consideration the threshold
- To chose which ancestor we will use we take help of a binary search





```
void insertToMound(MNODE mound, int val){
    int thr = Threshold;
    while(thr>0){
        if(size<=0){
            thr = 0;
            break;
        }
        int nd = (size+1)/2;//No of leaf nodes
        int l = (size - nd) + rand() % nd;//The index of leaf nodes is returned
        if(moundVal(mound, i: l)< val){
            thr--;//Unsatisfactory node, the code is run again
            continue;
        }
        else{
            thr--;
            if(l==0){
                thr=0;
                mound[l] = insertToFirstLL( mound: mound[l],val);
                return;
            }
            int lft = 0, last = 0, rt = l, counter = 1;
            while(lft < rt){
                if(lft == last)
                    counter = 1;
```



```
                    counter = 1;
                else
                    counter++;
                int level = (int)log2(rt + 1);//Returns the level of leaf node
                int m = (rt - 1)/(pow( X: 2,  Y: (int)(level/pow( X: 2,  Y: counter))));
                if(moundVal(mound,  i: m) < val){
                    last = lft;
                    lft = 2 * m + 2;//The right child of mid element selected for le
                }
                else{
                    rt = m;
                }
            }
            mound[rt] = insertToFirstLL( mound: mound[rt], val);
            return;
        }
    }
    if(thr<=0){
        increaseMoundLevel(mound, val);
    }
```

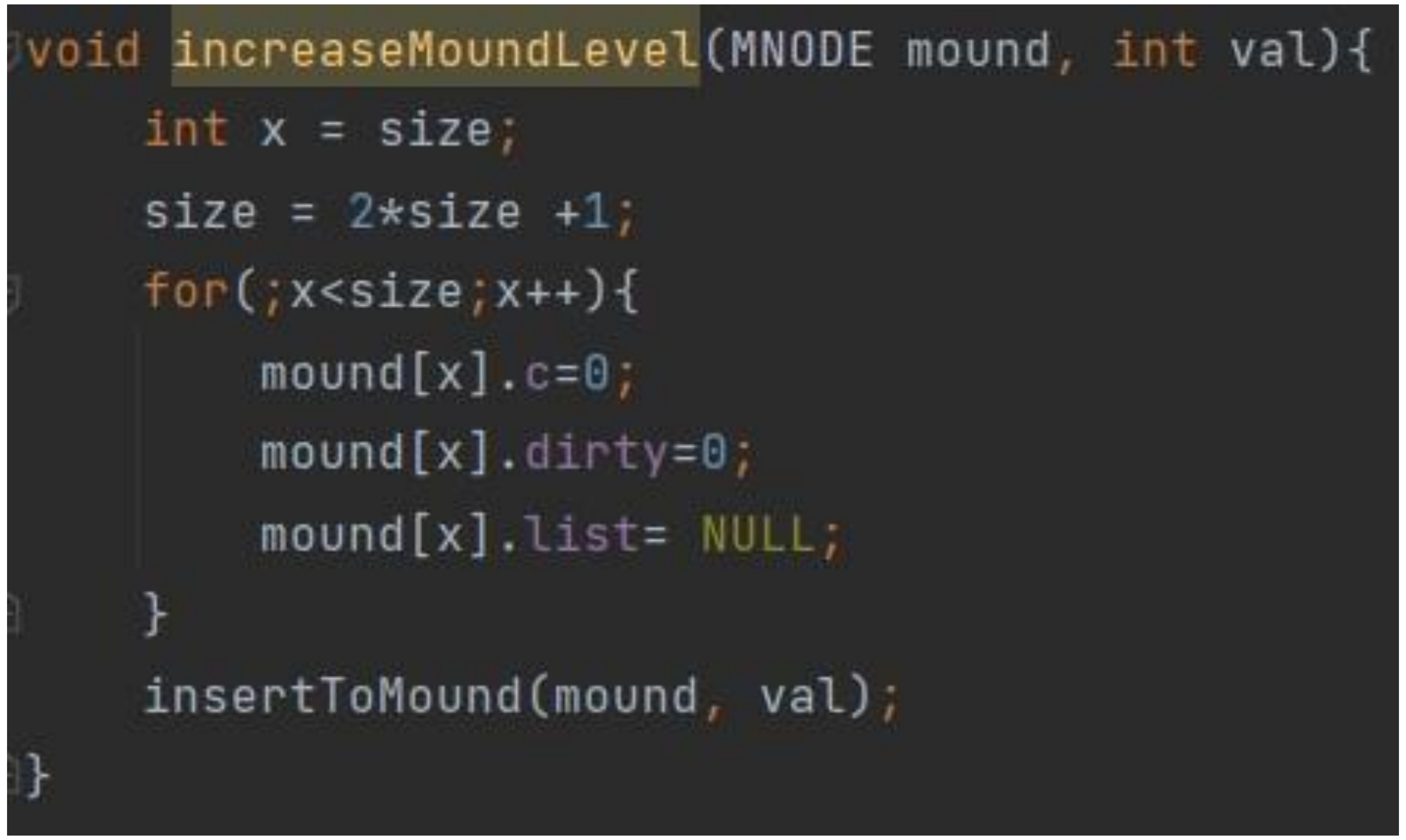
Code snippet for insertToMound()

Time complexity insertToMound

- We know binary search log(n) time
- But in a tree the height will be log(n) on which we are running the search Therefore the total time complexity of insertToMound is log(log(n)).

increaseMound

- This function is called when we could not enter value in mound This function is called in insertToMound() function.
- Once new mound level is added, the insertToMound() function is again called with newly added empty level.

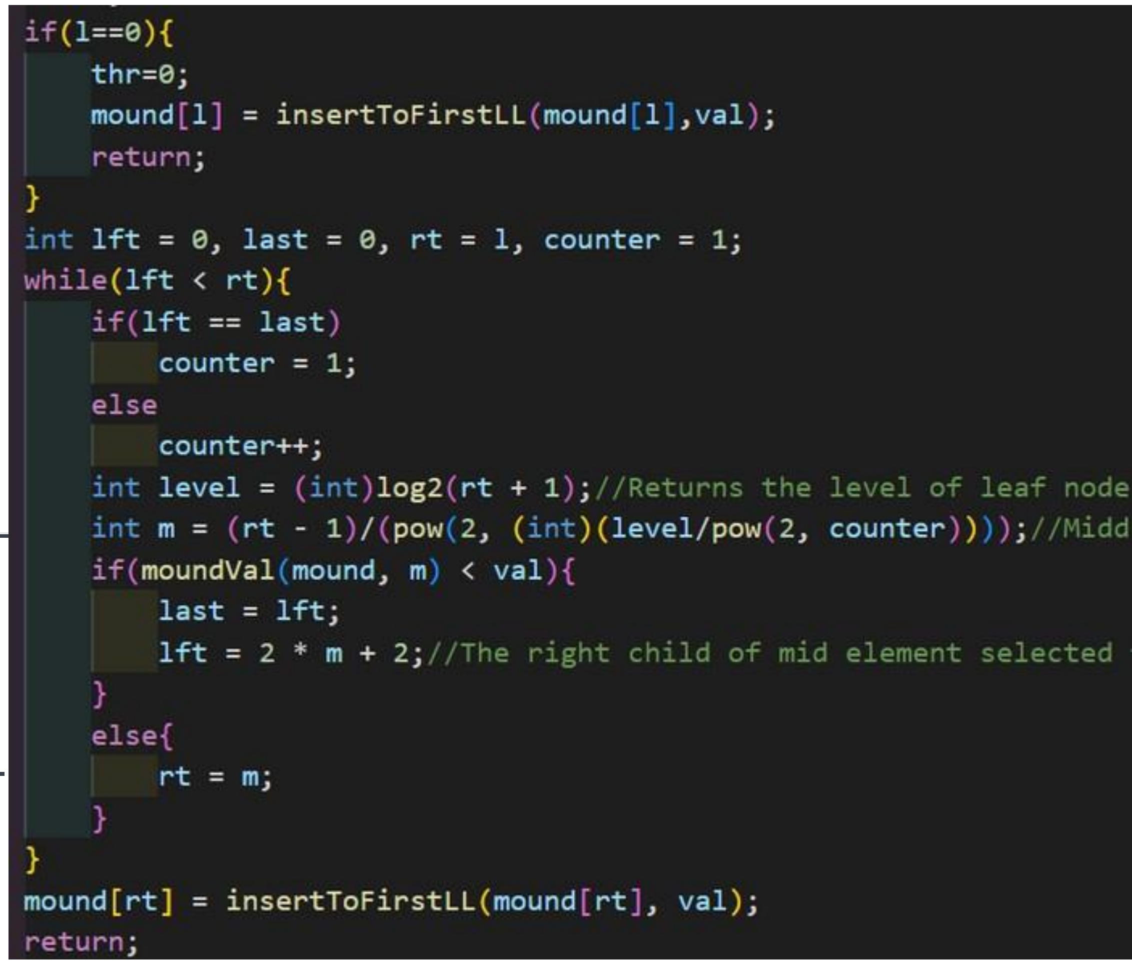


```
void increaseMoundLevel(MNODE mound, int val){
    int x = size;
    size = 2*size +1;
    for(;x<size;x++){
        mound[x].c=0;
        mound[x].dirty=0;
        mound[x].list= NULL;
    }
    insertToMound(mound, val);
}
```

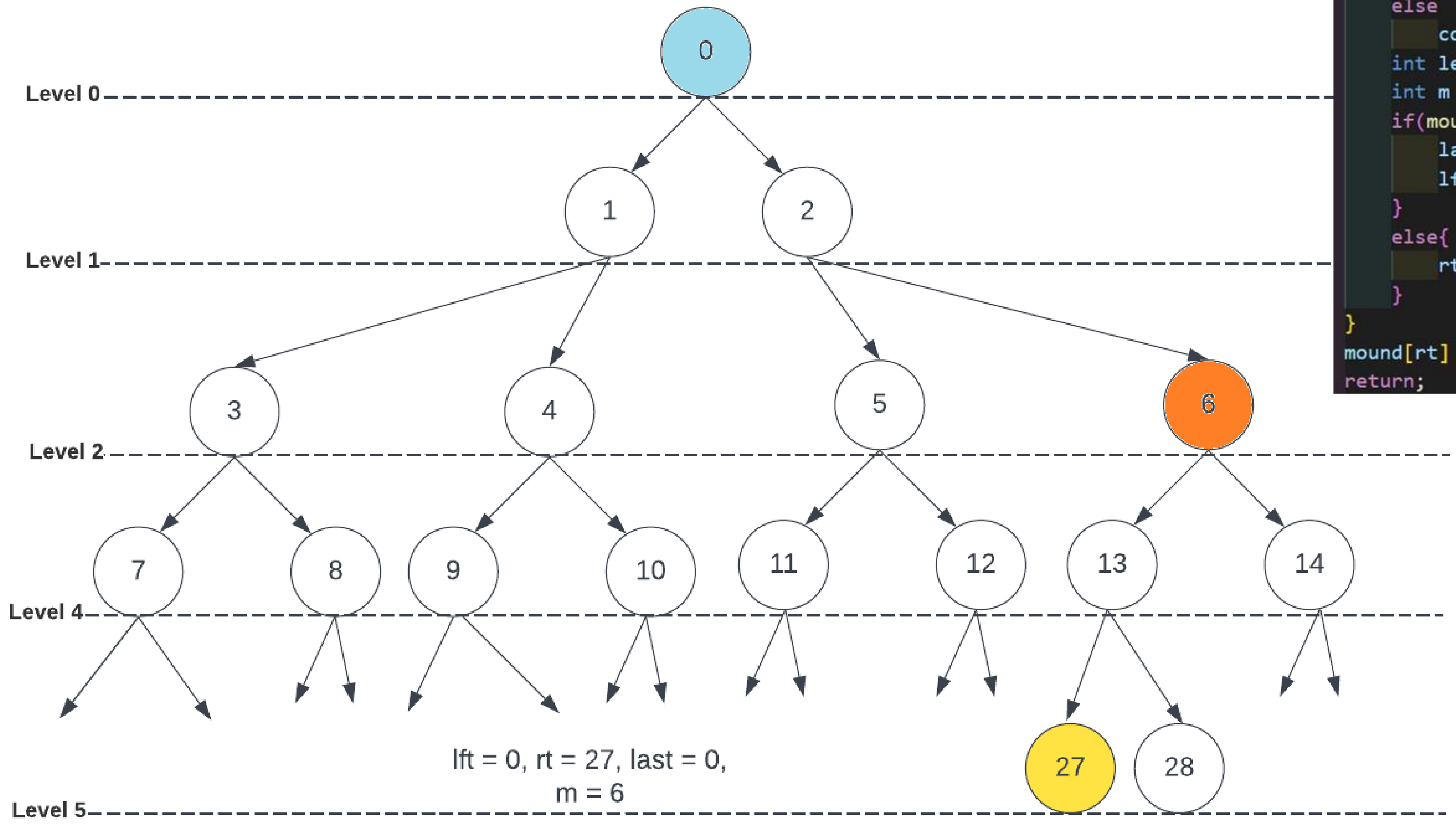
Binary Search of Insertion

We have used a few formulas to implement this:

- Level = log2(i + 1), where i is the index of current node
- m = (i-1)/(2^[Level/{2^counter}]), here, we get m, which is the central index of the branch ending at i
    - Proceed with the binary search operation
    - If we l = 0, i.e, root node, directly add the value to the root node
- We will help understand the exact procedure through an example



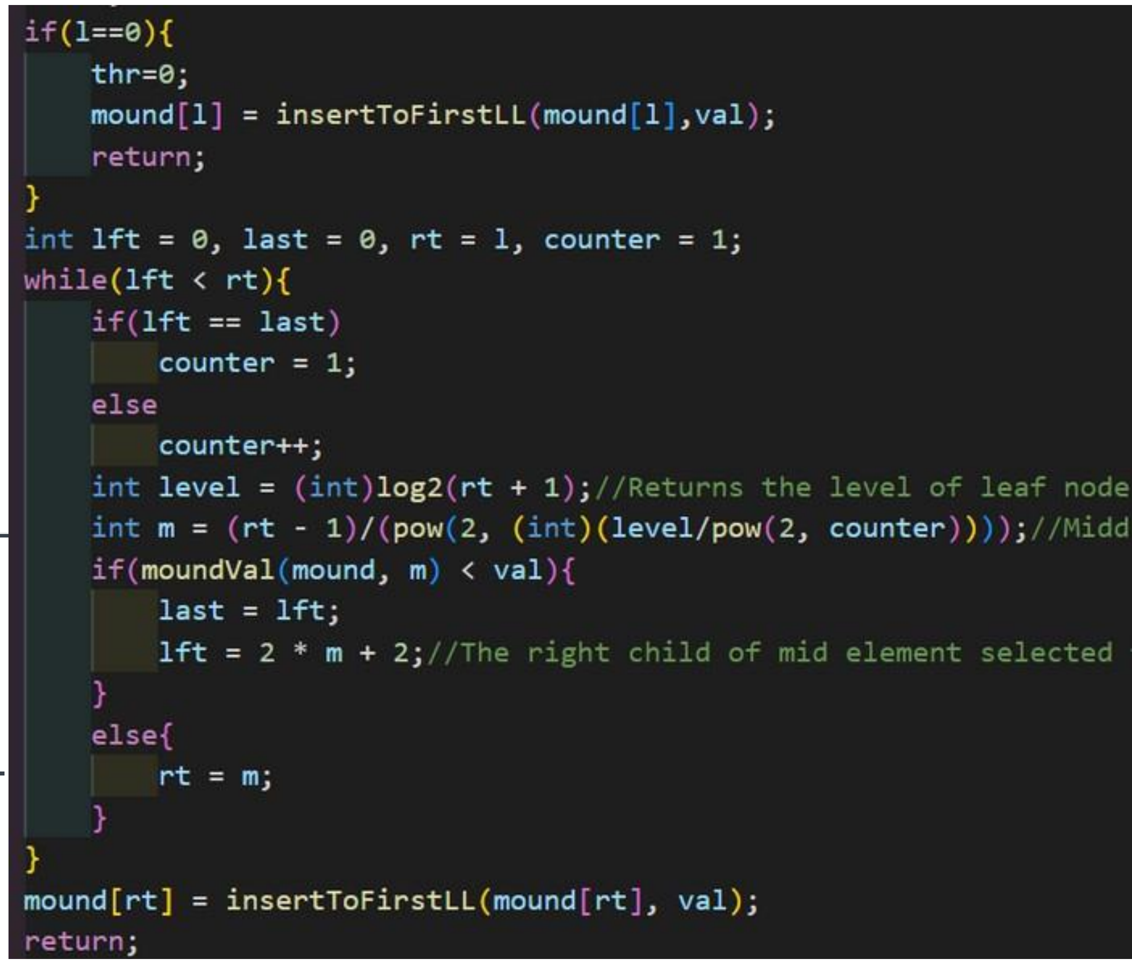
```
if(l==0){
    thr=0;
    mound[l] = insertToFirstLL(mound[l],val);
    return;
}
int lft = 0, last = 0, rt = 1, counter = 1;
while(lft < rt){
    if(lft == last)
        counter = 1;
    else
        counter++;
    int level = (int)log2(rt + 1);//Returns the level of leaf node
    int m = (rt - 1)/(pow(2, (int)(level/pow(2, counter))));//Midd
    if(moundVal(mound, m) < val){
        last = lft;
        lft = 2 * m + 2;//The right child of mid element selected
    }
    else{
        rt = m;
    }
}
mound[rt] = insertToFirstLL(mound[rt], val);
return;
```



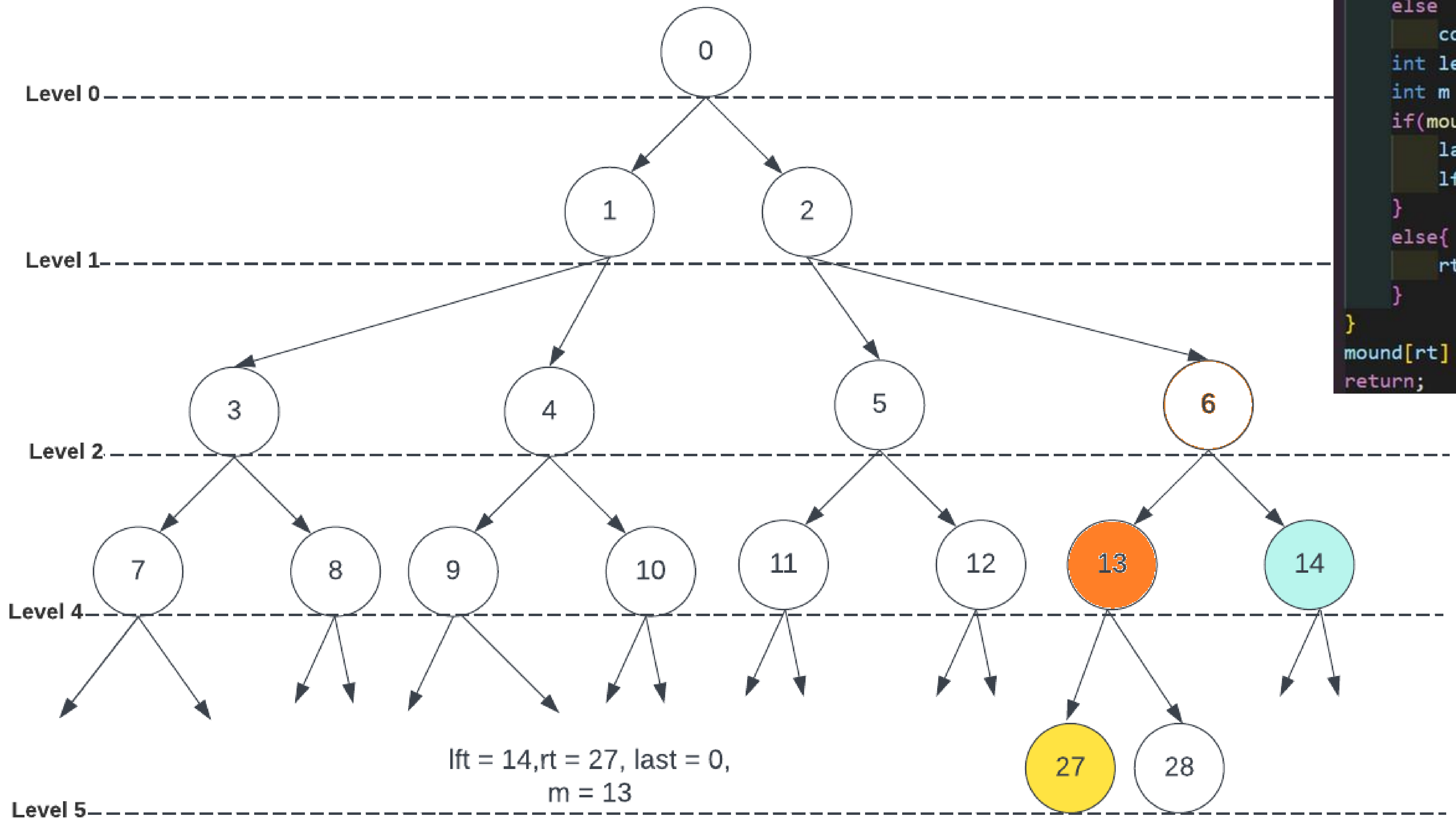
Binary Search of Insertion

We have used a few formulas to implement this:

- Level = log2(i + 1), where i is the index of current node
- m = (i-1)/(2^[Level/{2^counter}]), here, we get m, which is the central index of the branch ending at i
    - Proceed with the binary search operation
    - If we l = 0, i.e, root node, directly add the value to the root node
- We will help understand the exact procedure through an example



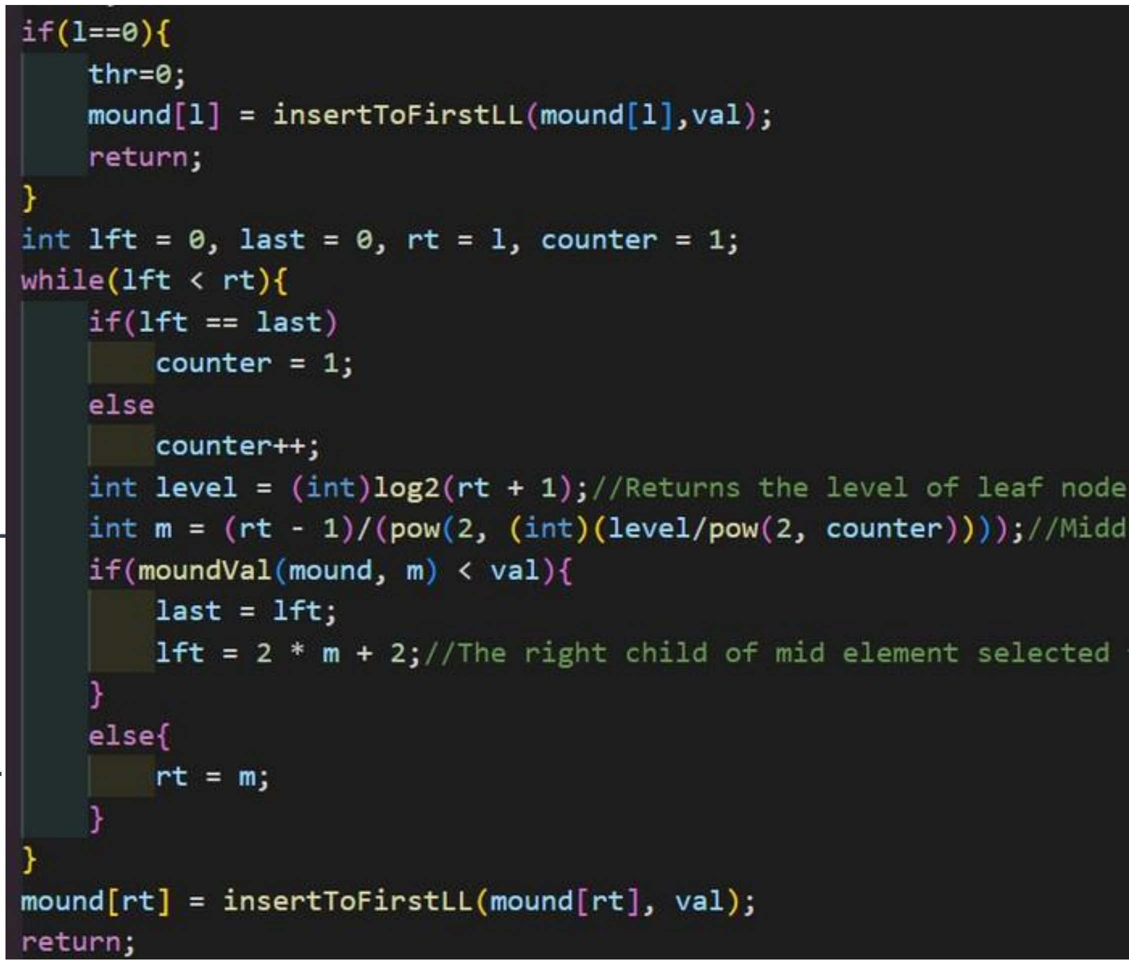
```
if(l==0){
    thr=0;
    mound[l] = insertToFirstLL(mound[l],val);
    return;
}
int lft = 0, last = 0, rt = 1, counter = 1;
while(lft < rt){
    if(lft == last)
        counter = 1;
    else
        counter++;
    int level = (int)log2(rt + 1);//Returns the level of leaf node
    int m = (rt - 1)/(pow(2, (int)(level/pow(2, counter))));//Midd
    if(moundVal(mound, m) < val){
        last = lft;
        lft = 2 * m + 2;//The right child of mid element selected
    }
    else{
        rt = m;
    }
}
mound[rt] = insertToFirstLL(mound[rt], val);
return;
```



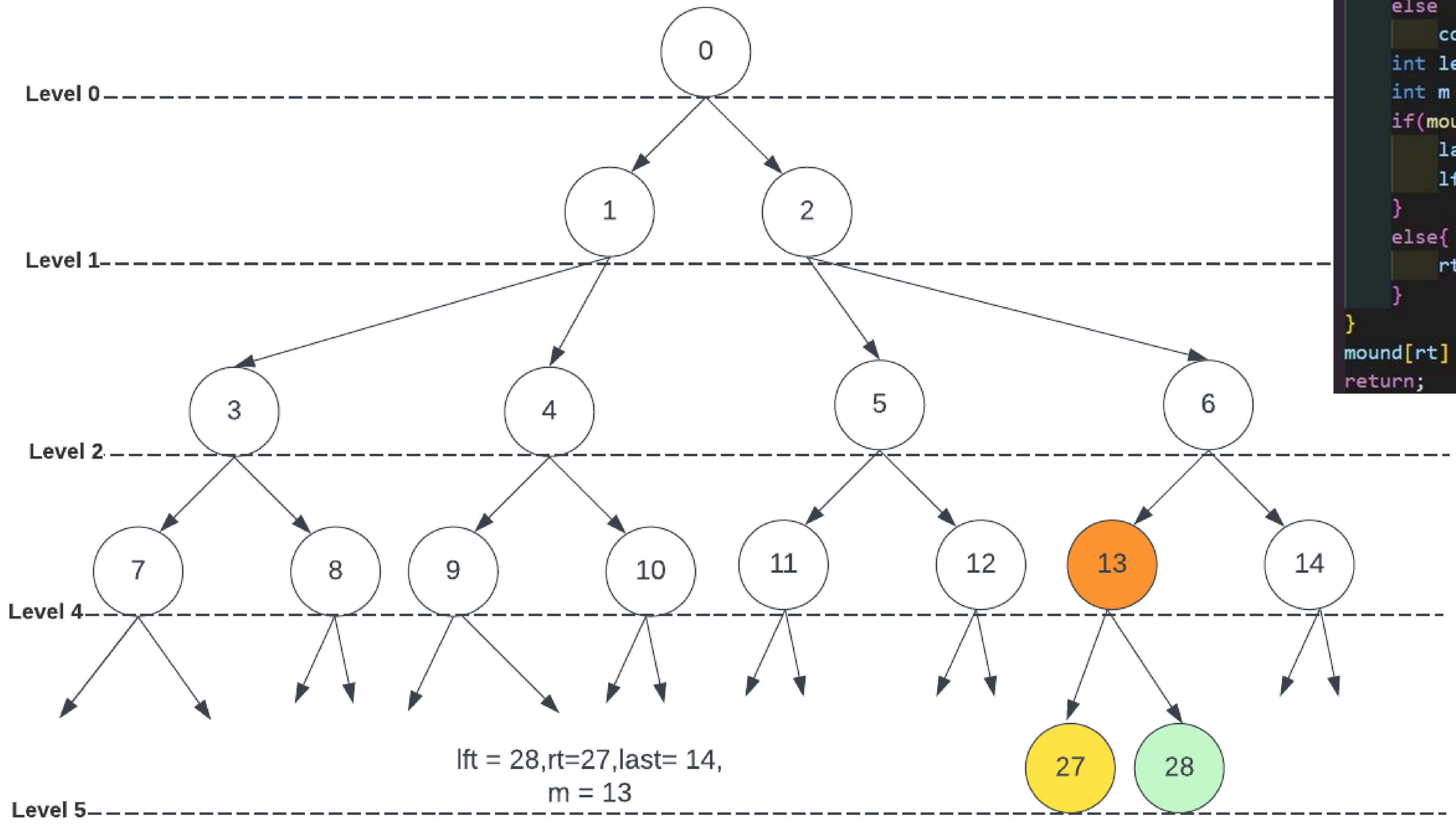
Binary Search of Insertion

We have used a few formulas to implement this:

- Level = log2(i + 1), where i is the index of current node
- m = (i-1)/(2^[Level/{2^counter}]), here, we get m, which is the central index of the branch ending at i
  - Proceed with the binary search operation
  - If we l = 0, i.e, root node, directly add the value to the root node
- We will help understand the exact procedure through an example



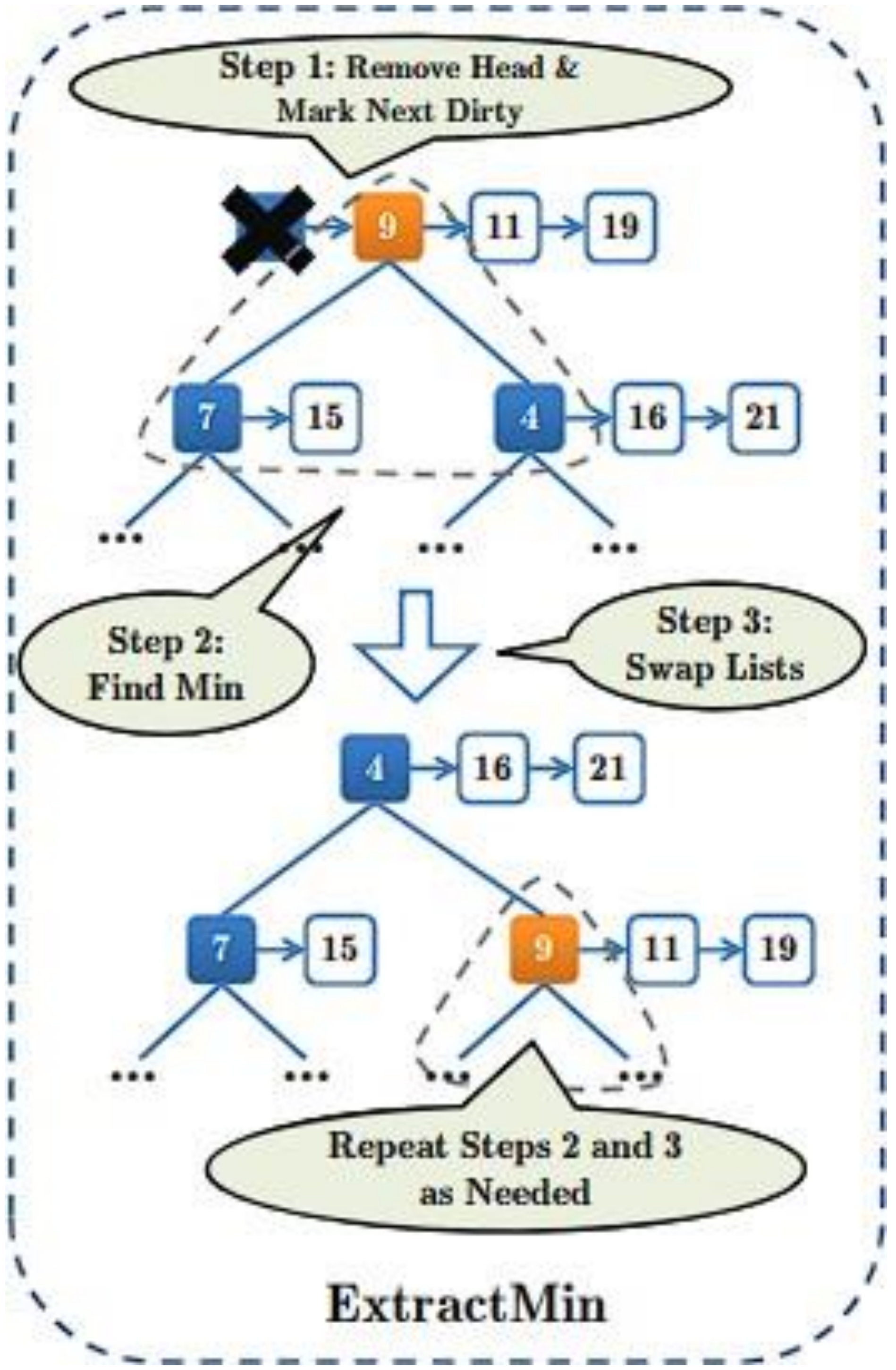
```
if(l==0){
    thr=0;
    mound[l] = insertToFirstLL(mound[l],val);
    return;
}
int lft = 0, last = 0, rt = 1, counter = 1;
while(lft < rt){
    if(lft == last)
        counter = 1;
    else
        counter++;
    int level = (int)log2(rt + 1);//Returns the level of leaf node
    int m = (rt - 1)/(pow(2, (int)(level/pow(2, counter))));//Midd
    if(moundVal(mound, m) < val){
        last = lft;
        lft = 2 * m + 2;//The right child of mid element selected
    }
    else{
        rt = m;
    }
}
mound[rt] = insertToFirstLL(mound[rt], val);
return;
```

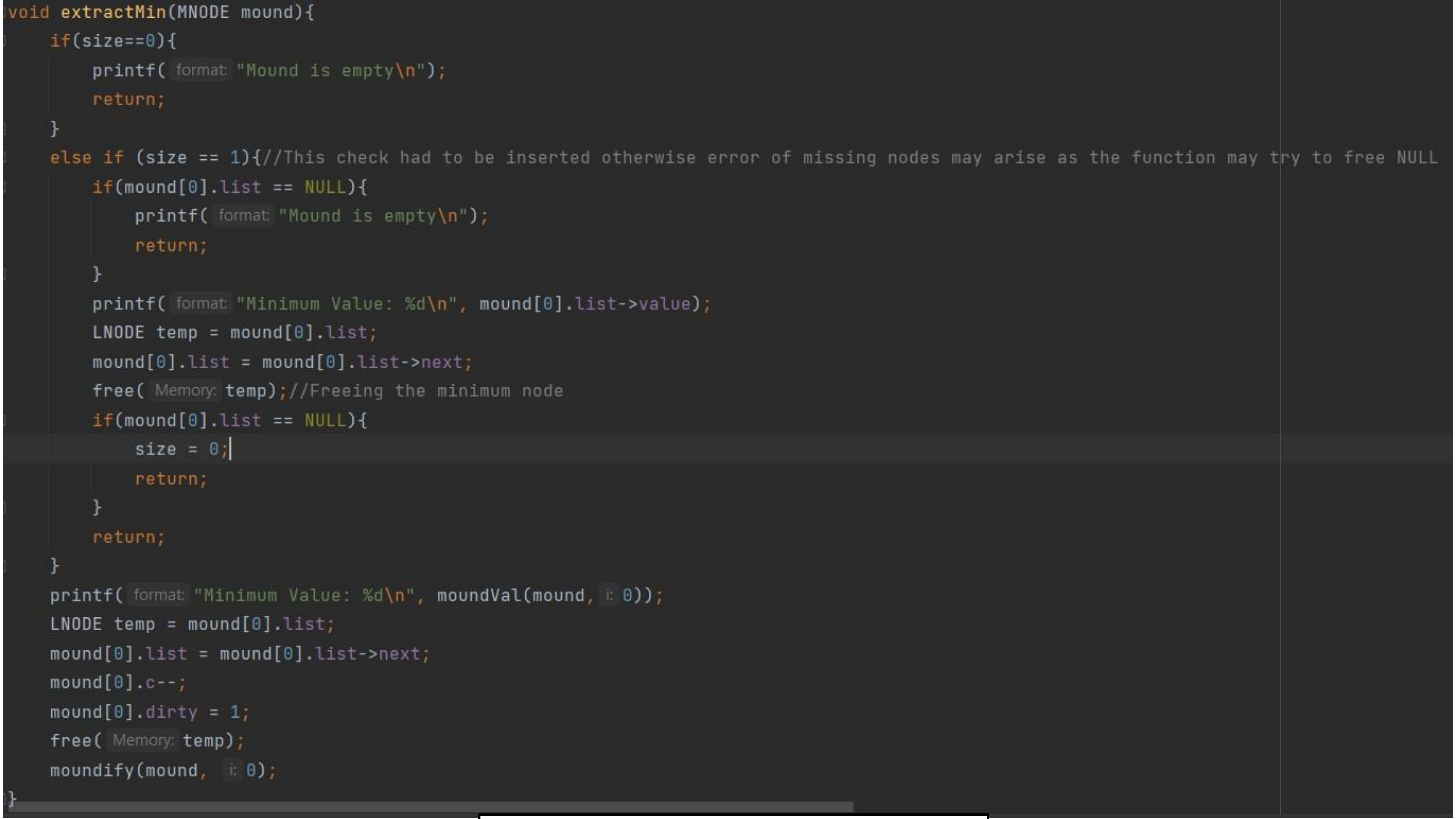


The extractMin operation implementation

- This operation is used to get minimum value from mound
- In extractMin operation, we have kept several checks

1. (size==0), this check is used to see whether the mound is empty or not. If this condition is satisfied then "Mound is empty" is returned.
2. (size==1), this condition is satisfied when there is only one list present and there are no child node. Another check is put for checking if this list is empty or not. If not empty then minimum value is printed, else "Mound is empty" is printed. The Min node is deleted and next node is pointed. If next node is empty then we make size = 0 for the mound(because this check is only for size = 1).
3. If none of the above checks are satisfied then the mound contains more than one list. Same step as previous point are repeated. We could have combined both of these steps but we were getting segmentation fault in certain cases, hence we decided to divide this step in 2 parts



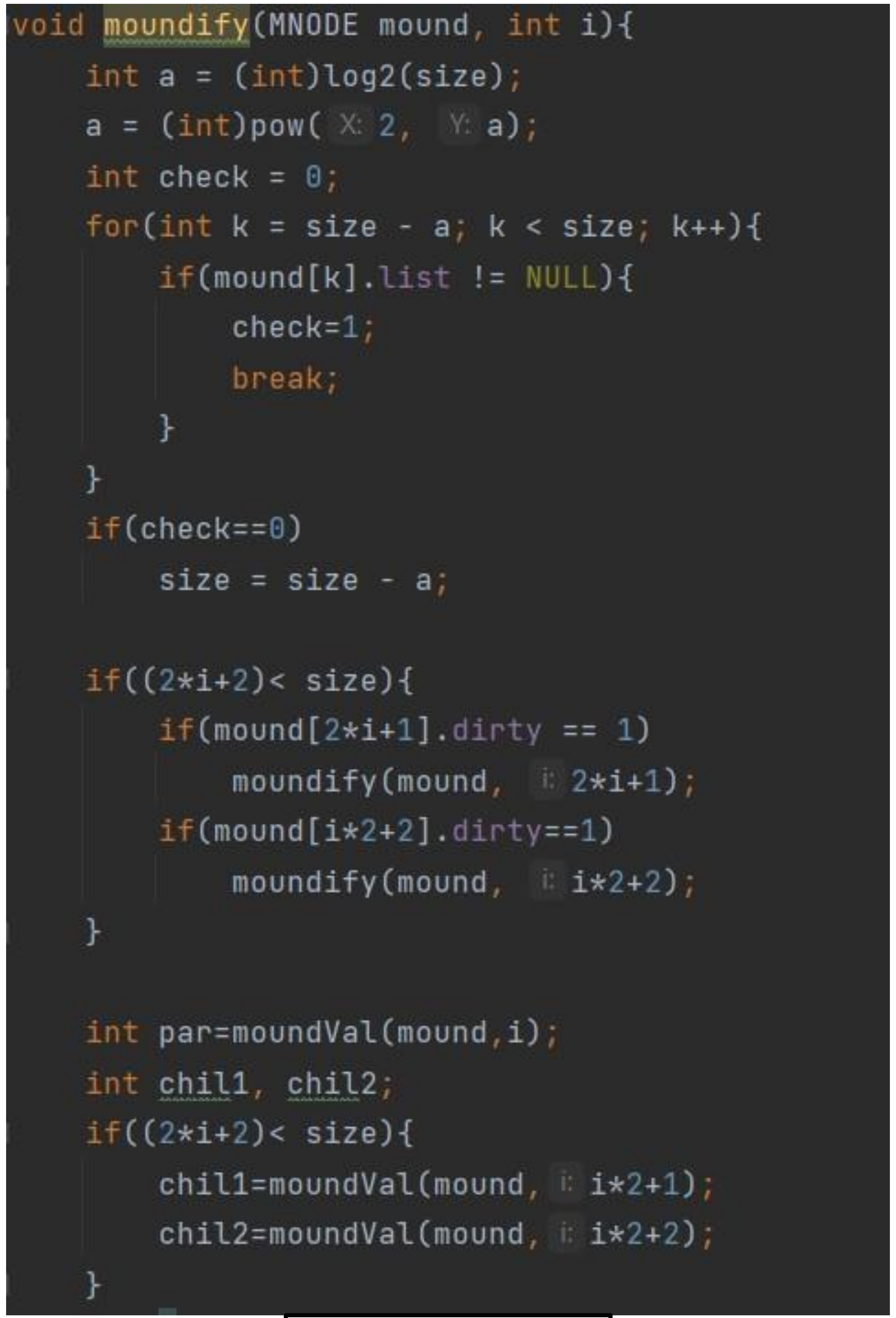


```c
void extractMin(MNODE mound){
    if(size==0){
        printf( format: "Mound is empty\n");
        return;
    }
    else if (size == 1){//This check had to be inserted otherwise error of missing nodes may arise as the function may try to free NULL
        if(mound[0].list == NULL){
            printf( format: "Mound is empty\n");
            return;
        }
        printf( format: "Minimum Value: %d\n", mound[0].list->value);
        LNODE temp = mound[0].list;
        mound[0].list = mound[0].list->next;
        free( Memory: temp);//Freeing the minimum node
        if(mound[0].list == NULL){
            size = 0;
            return;
        }
        return;
    }
    printf( format: "Minimum Value: %d\n", moundVal(mound, i: 0));
    LNODE temp = mound[0].list;
    mound[0].list = mound[0].list->next;
    mound[0].c--;
    mound[0].dirty = 1;
    free( Memory: temp);
    moundify(mound,  i: 0);
}
```

Code snippet for extractMin()

The moundify function

- moundify is a helper function which is used to restore the mound property
- It inspects the val() of tree and its children, and determines which is smallest.
- In moundify we send the node as parameter which is dirty and restore the mound property of that subtree
- The first code snippet is used to delete a null floor
- After this we check if the child node are dirty or not, if they are dirty then moundify function is called on them sending their indices as parameter
- If none of the above condition are satisfied then it means the child node are satisfying mound condition. We create chil1, and chil2 to store their values, if they are null then we give then INT_MAX value
- We then check if parent is smaller than child node, if this condition is satisfied then it means the node is not dirty.

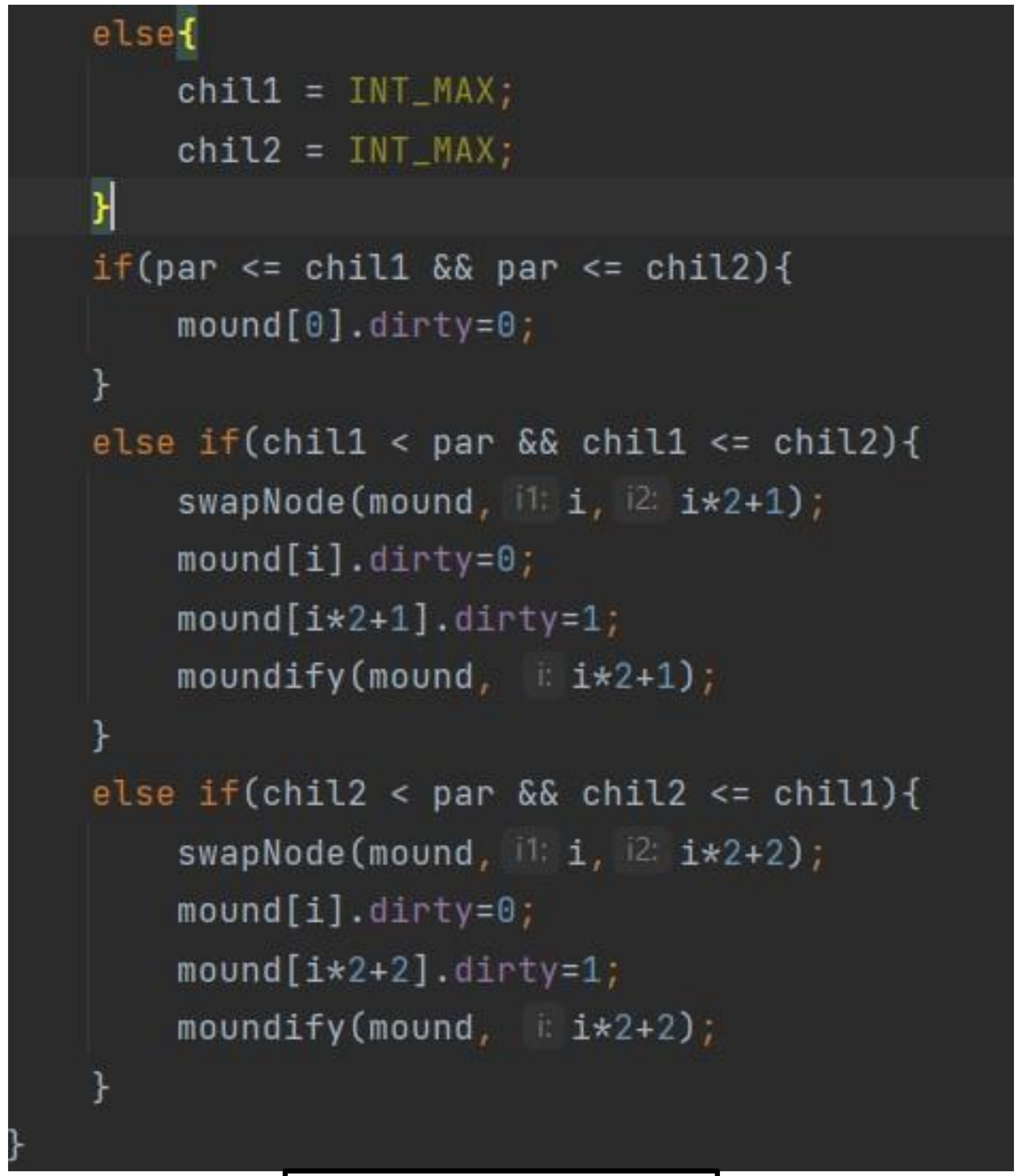


```
void moundify(MNODE mound, int i){
    int a = (int)log2(size);
    a = (int)pow( X: 2,  Y: a);
    int check = 0;
    for(int k = size - a; k < size; k++){
        if(mound[k].list != NULL){
            check=1;
            break;
        }
    }
    if(check==0)
        size = size - a;

    if((2*i+2)< size){
        if(mound[2*i+1].dirty == 1)
            moundify(mound,  i: 2*i+1);
        if(mound[i*2+2].dirty==1)
            moundify(mound,  i: i*2+2);
    }

    int par=moundVal(mound,i);
    int chil1, chil2;
    if((2*i+2)< size){
        chil1=moundVal(mound,  i: i*2+1);
        chil2=moundVal(mound,  i: i*2+2);
    }
```

S N I P P E T - 1

The moundify function

- If the above condition is not satisfied then we find the smallest among chil1 and chil2 and swap them with their parent node.
- But an issue may arise i.e. the swapped node may have become and so we set it as dirty and run moundify on it



```
    else{
        chil1 = INT_MAX;
        chil2 = INT_MAX;
    }
    if(par <= chil1 && par <= chil2){
        mound[0].dirty=0;
    }
    else if(chil1 < par && chil1 <= chil2){
        swapNode(mound, i1: i, i2: i*2+1);
        mound[i].dirty=0;
        mound[i*2+1].dirty=1;
        moundify(mound, i: i*2+1);
    }
    else if(chil2 < par && chil2 <= chil1){
        swapNode(mound, i1: i, i2: i*2+2);
        mound[i].dirty=0;
        mound[i*2+2].dirty=1;
        moundify(mound, i: i*2+2);
    }
}
```

SNIPPET-2

Time complexity of extractMin function

- The time complexity of extractMin is log(n), the mound is a complete binary tree whose height is log(n).
- Therefore the worst case time complexity is log(n) which is same as mound.

IMPORTANT LINKS

1. https://drive.google.com/drive/folders/1Ve-56-v7vWNOrTFPOLdhqLdMwfjHJa55

Contains Research Papers, Articles & Code Implementations

THANK YOU

ARYAMAN CHAUHAN	2020B5A7PS2006P
ARNAV GUJARATHI	2020A7PS0066P
AVYAKT GARG	2020B1A7PS1902P
DEVANSH YADAV	2020B5A7PS2001P
TANUSHI GARG	2020B1A7PS0648P