CS 2124: DATA STRUCTURES
Spring 2024

8^{th} Lecture

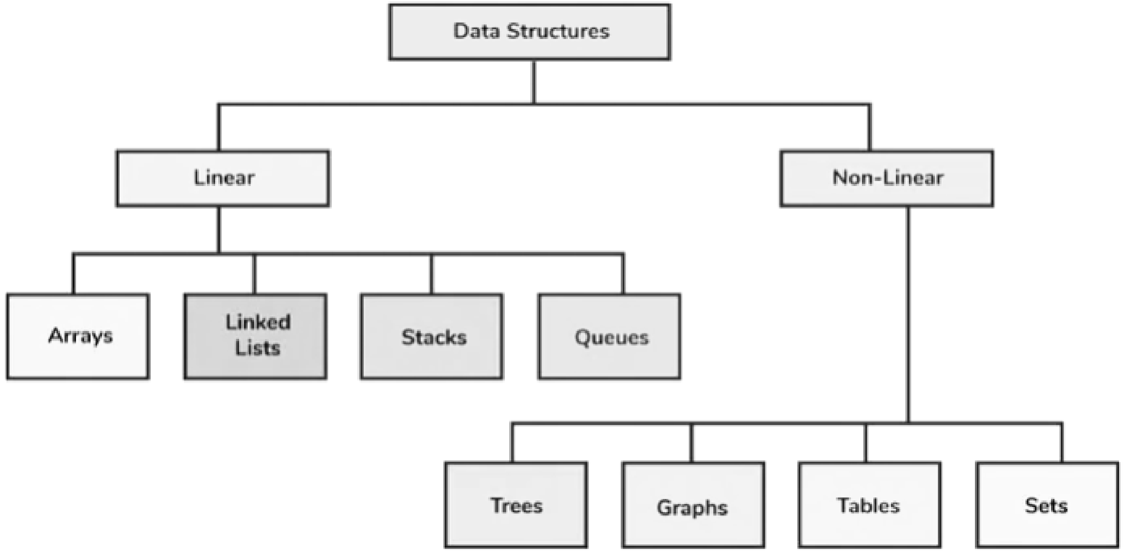
Topics: **Heaps**

Topics

- How to identify which Data Structure to use
- Heaps
- Adding a Node to a Heap
- Removing the Top of a Heap
- Implementing a Heap (Array)
  - Important Points About The Implementation
- From Array to Heap
- Heap (Applications)
- Heap (Advantages and Disadvantages)
- Huffman using heap
  - Applications of Huffman Coding

How to identify which Data Structure to use

- Understanding the data you will deal with before selecting a data structure is vital:
  1. When you need to access elements randomly from your data, **arrays** might be the best choice.
  2. In case, you constantly need to add or delete elements from a list, and the list size also might change, then **linked lists** can be particularly useful.
  3. When you need to effectively store multiple levels of data, such as record structures, and carry out operations like searching and sorting, then **trees** are useful.
  4. When you need to describe interactions between entities, such as those in social networks, and perform operations such as shortest path and connectivity, then **Graphs** are preferred.



5. Hash tables are helpful for speedy key lookups

Image Source: Interview Bit & is basically a classification of Data Structures

How to identify which Data Structure to use

- While choosing a data structure, you must also consider the operations to be performed on the data.
- Different data structures optimize numerous actions, such as sorting, searching, insertion, and deletion.
  - Linked lists are better for actions like insertion and deletion.
  - Binary trees are best for searching and sorting.
  - A hash table can be the best choice if your application requires simultaneous insertion and searching.

- **Evaluate the Environment:**
  - When considering a data structure, you must evaluate the environment in which the application will run.
  - The environment affects how well and how promptly accessible data structures are.
    - (i.e. Processing resources, Concurrency/Parallel processes, network latency, etc.)

How to identify which Data Structure to use

- Before picking a data structure, consider your application's data, obligations, and environment.
- While going with your choice, think about the following elements:
  - Time complexity
  - Space Complexity
  - Read vs. Write Operations
  - Type of Data
  - Hardware
  - Network
  - Data Synchronization

Data Structure	Illustration	Use Cases
List		Twitter feeds
Array	0 1 2 3	Math operations Large data sets
Stack		Undo/Redo of word editor
Queue		Printer jobs User actions in game
Heap		Task scheduling
Tree		HTML document AI decision
Suffix Tree		Search string in document
Graph		Friendship tracking Path finding
R-tree		Nearest neighbour
Hash Table		Caching systems

Image Source: ByteByteGo

Heaps

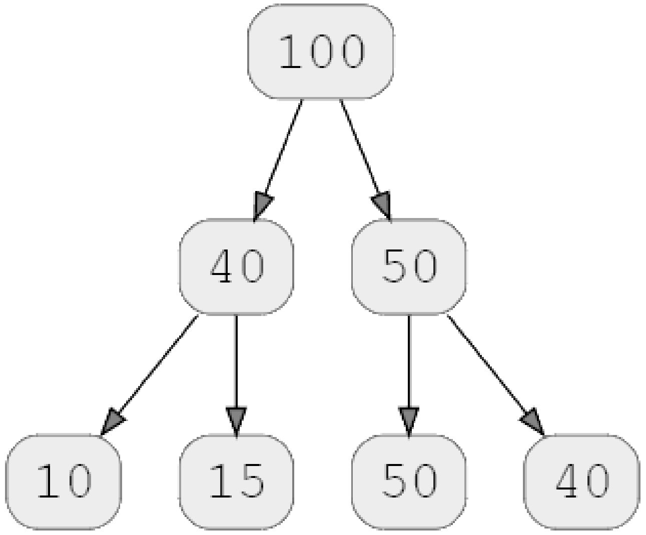
A heap is an advanced tree-based data structure used primarily for sorting and implementing priority queues.

- They are complete binary trees that have the following features:
  - Every level is filled except the leaf nodes (nodes without children are called leaves).
  - Every node has a maximum of 2 children.
  - All the nodes are as far left as possible, this means that every child is to the left of his parent.

- Heaps use complete binary trees to **avoid holes in the array (optimizing operations)**.

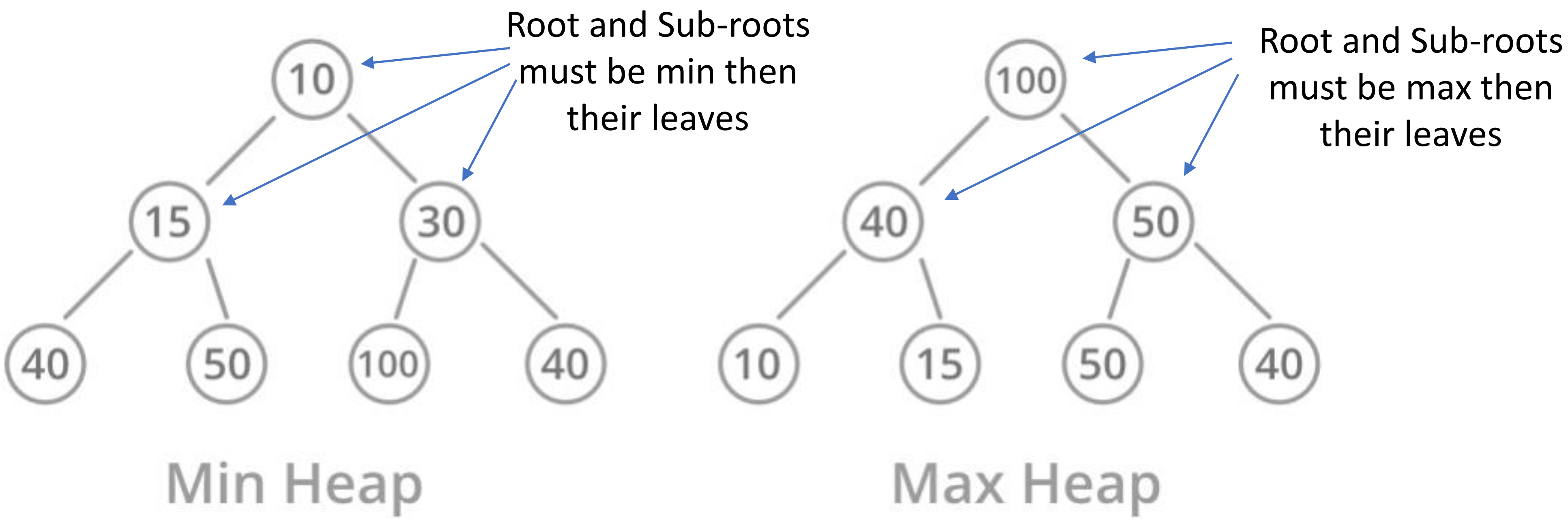
- A complete binary tree is a tree where each node has at most two children and the nodes at all levels are full, except for the leaf nodes which can be empty.

- Heaps are built based on the heap property, which compares the parent node key with its child node keys.



Heaps (Operations and Types)

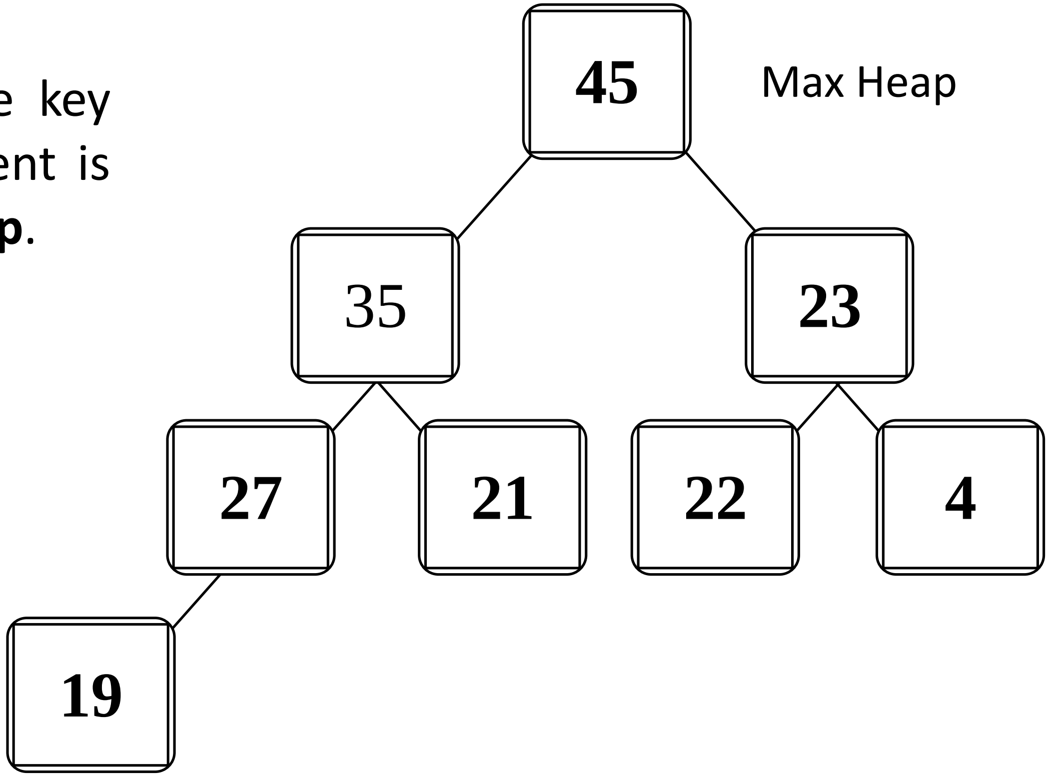
- **Heapify:** A process of creating a heap from an array.
- **Insertion:** Process to insert an element in existing heap time complexity *O(log N).*
- **Deletion**: Deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity *O(log N).*
- **Peek:** To check or find the most prior element in the heap, (max or min element for max and min heap).
- **Extract:** Returns the value of an item and then deletes it from the heap.
- **isEmpty:** Boolean, returns true if Boolean is empty and false if it has a node.



Heaps

It is important to note that heaps are **not always sorted**, the key condition that they follow is that the largest or smallest element is placed on the root node (top) depending if it is a **Max** or **Min Heap**.

The Heap data structure is not the same as heap memory.



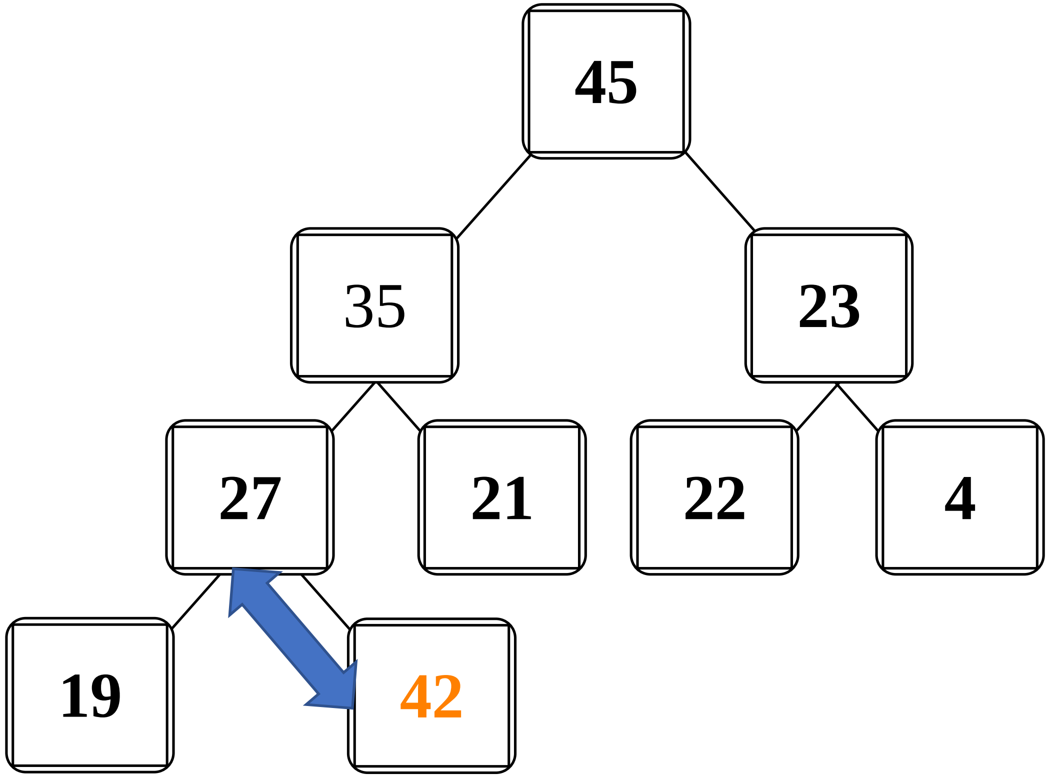
The "heap property" requires that each node's key is >=, <= the keys of its children

Each node in a heap contains a key that can be compared to other nodes' keys.

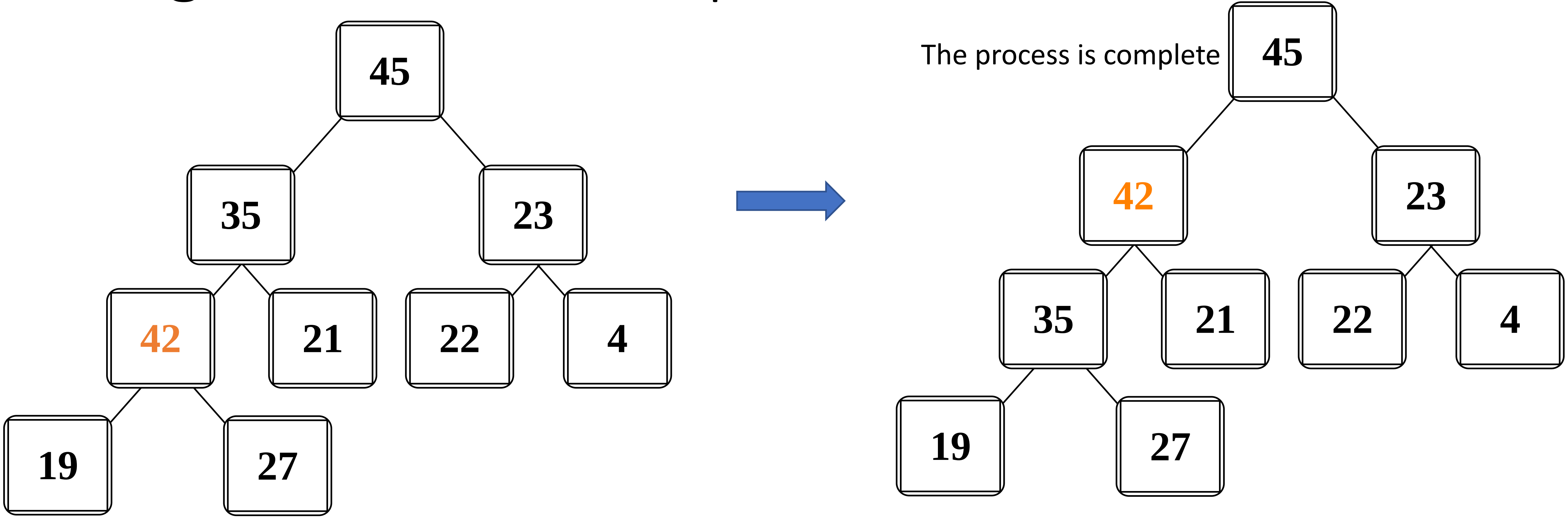
- Max Heap Parent >= to Child node key
- Min Heap Parent <= to Child node key

Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



Adding a Node to a Heap

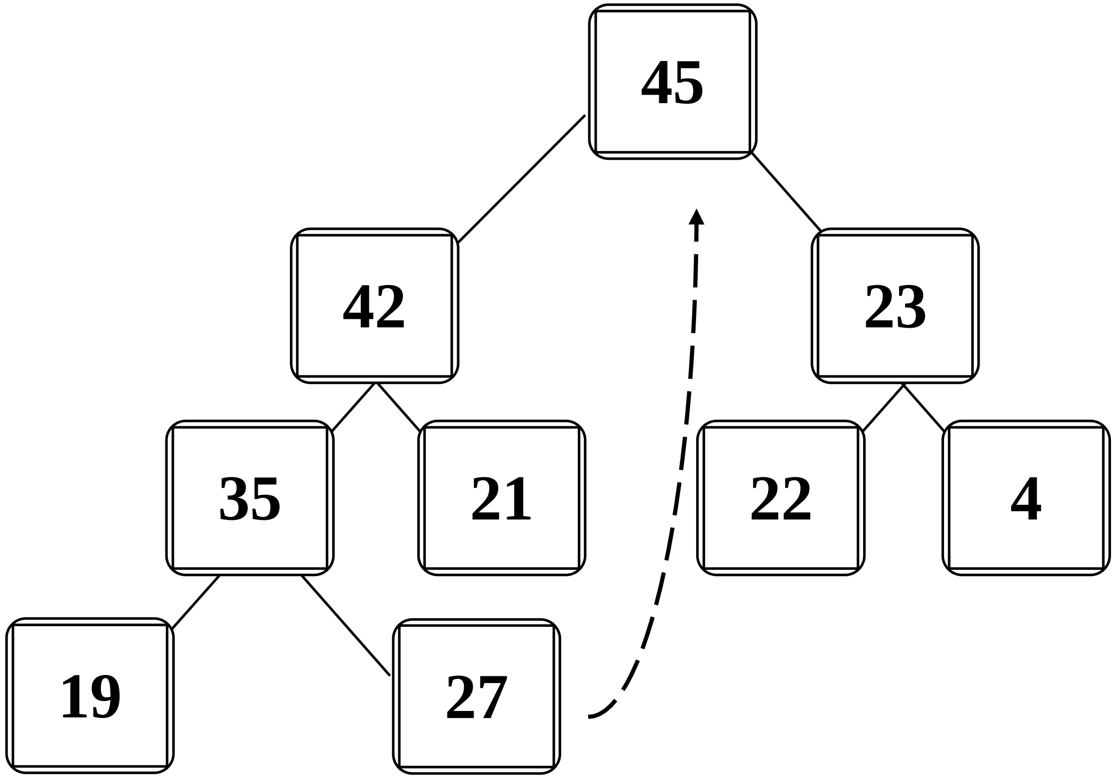


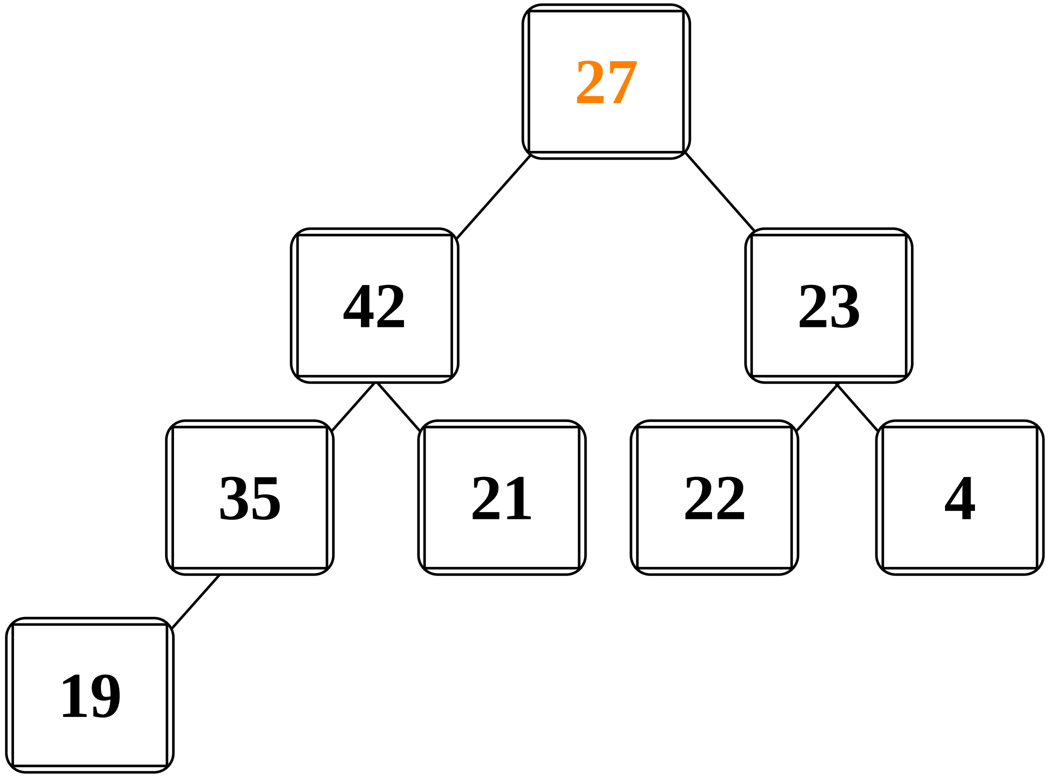
- The parent has a key that is >= new node,

or

- The node reaches the root.
- The process of pushing the new node upward is called **reheapification upward.**

Removing the Top of a Heap

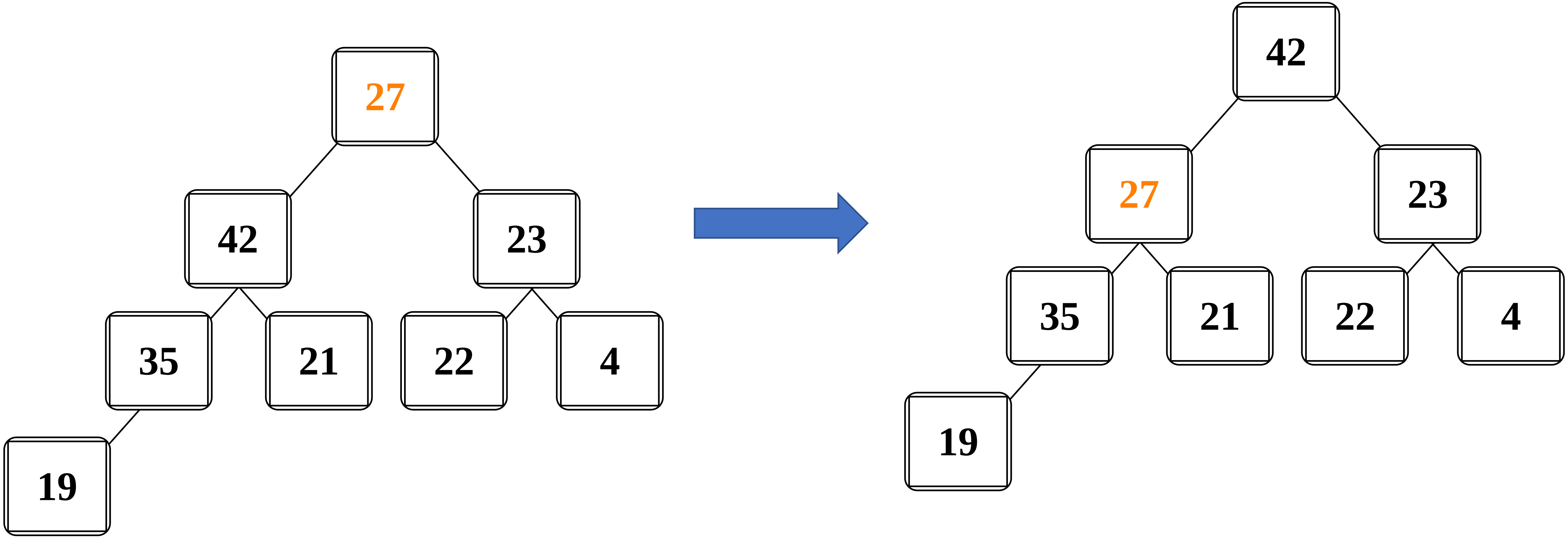




- Move the last node onto the root.

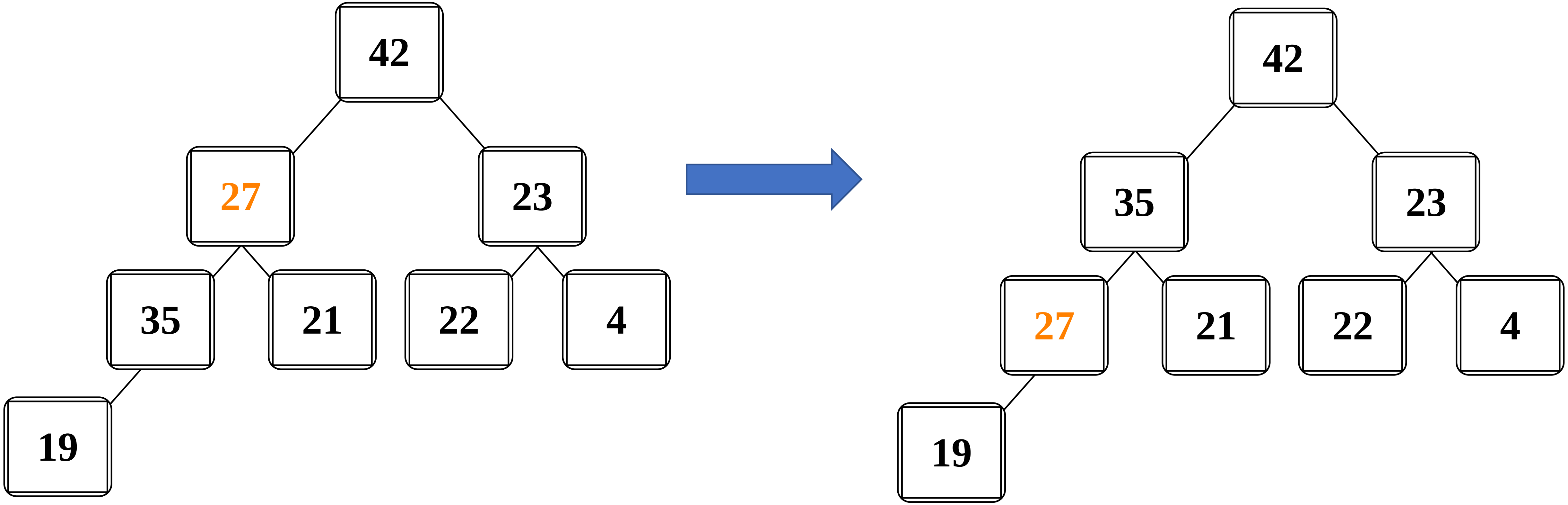
- Move the last node onto the root.

Removing the Top of a Heap



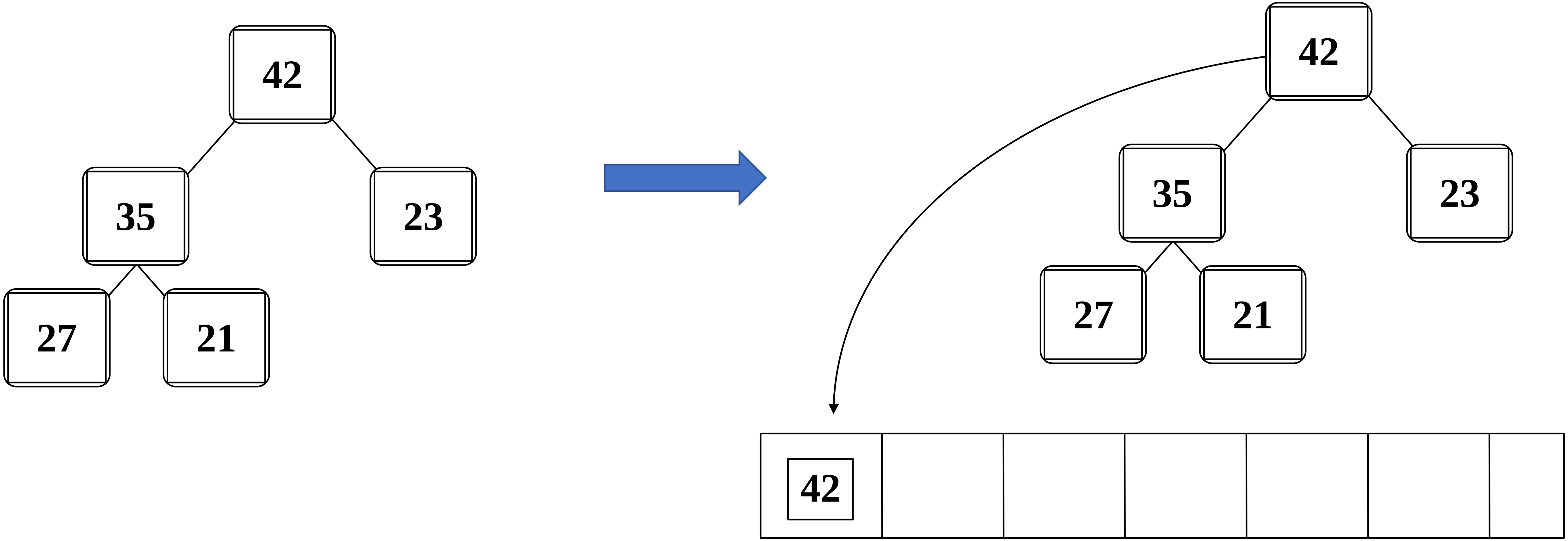
- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.

Removing the Top of a Heap



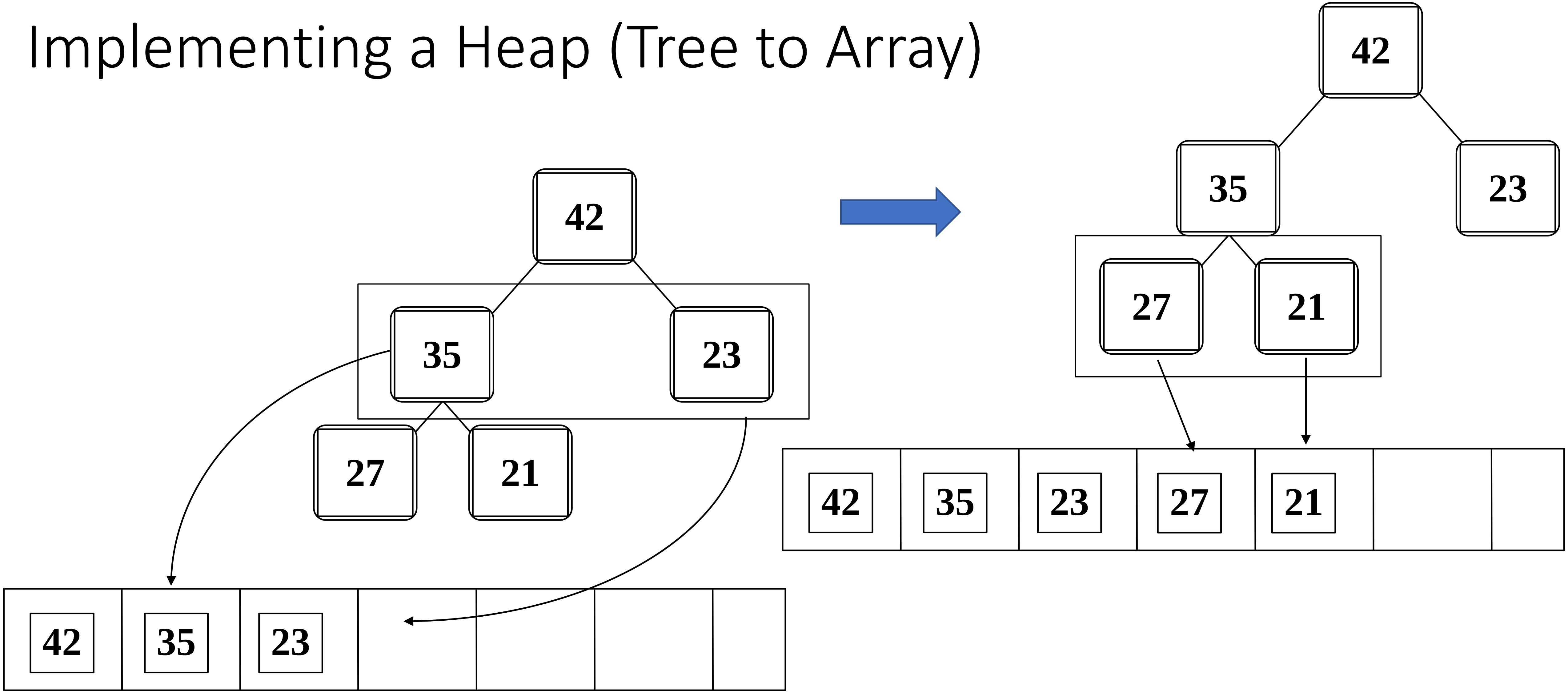
- The children all have keys <= the out-of-place node, or
- The node reaches the leaf.
- The process of pushing the new node downward is called **reheapification downward**.

Implementing a Heap (Tree to Array)



- We will store the data from the nodes in a partially-filled array.
- Data from the root goes in the first location of the array.

Implementing a Heap (Tree to Array)

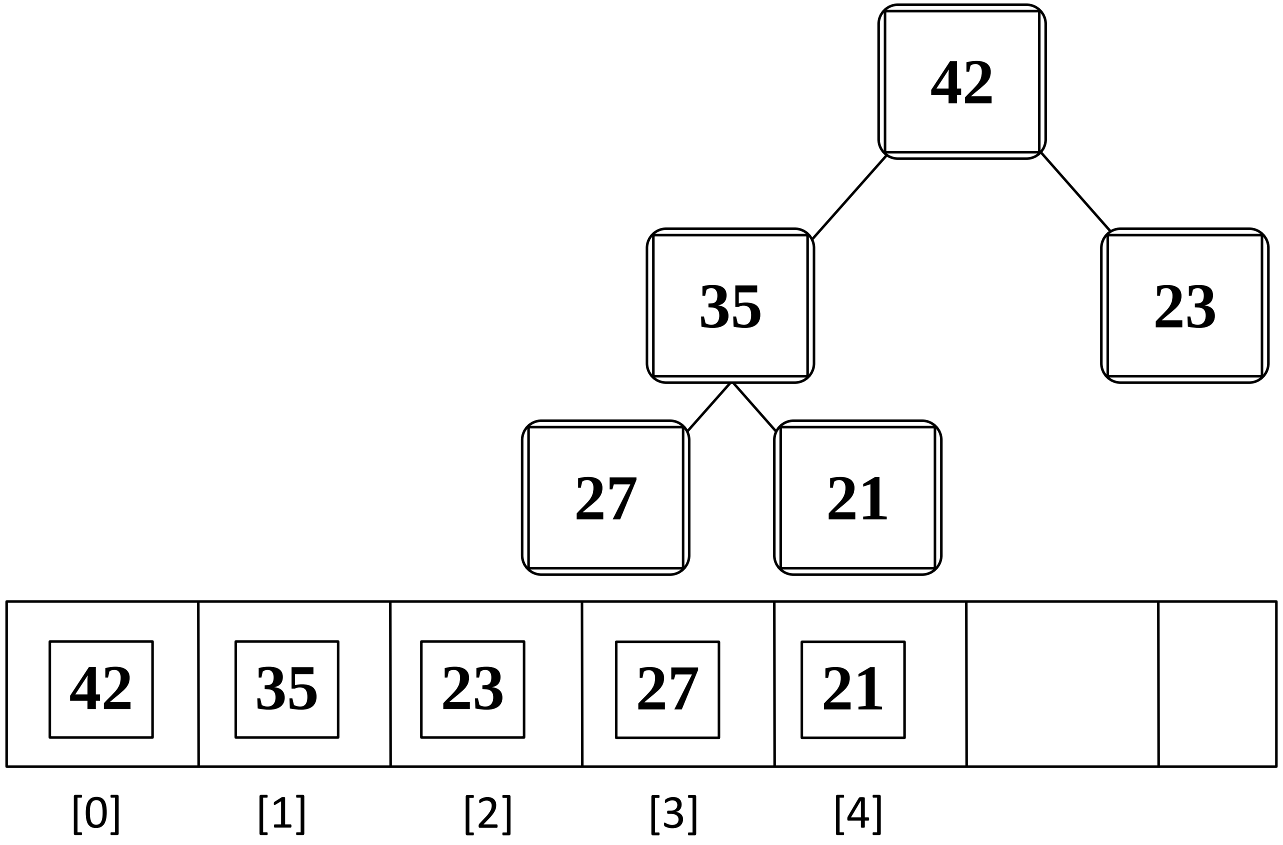


- Data from the next row goes in the next two array locations.

Important Points

- The links between the tree's nodes are not actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.

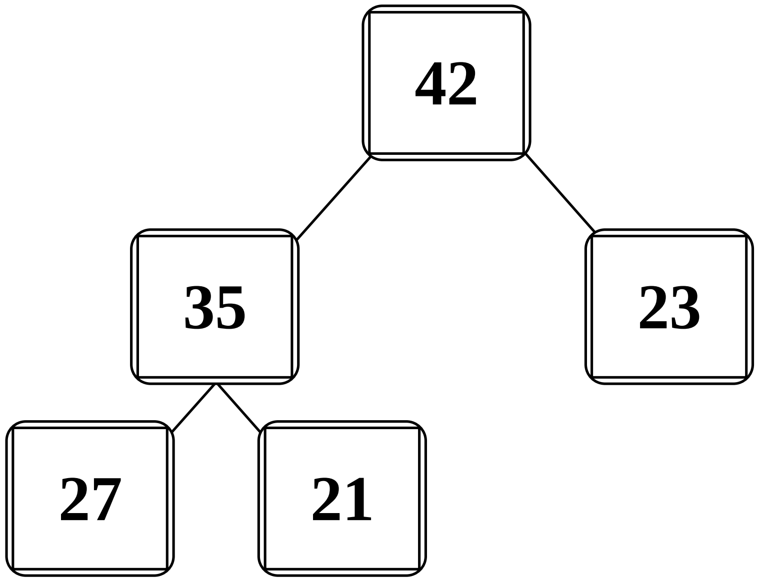
```
struct node* generateTree(){
    // Root Node
    struct node* root =  getNewNode(0);
    // Level 2 nodes
    root->left = getNewNode(1);
    root->right = getNewNode(2);
    // Level 3 nodes
    root->left->left = getNewNode(3);
    root->left->right = getNewNode(4);
  return root;
}
```

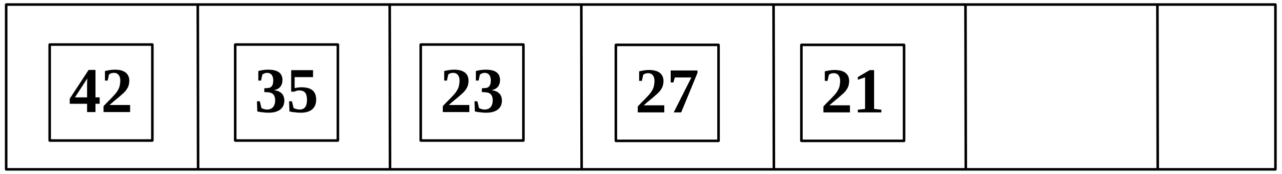


Implementing a Heap (Array to Tree conversion using mathematics)

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children.

1. The parent child of i will be at index $\left\lfloor \frac{i-1}{2} \right\rfloor$ (If array [0])
2. The parent child of i will be at index $\left\lfloor \frac{i}{2} \right\rfloor$ (If array [1])
3. The left child of i will be at index 2i + 1 (If array [0])
4. The left child of i will be at index 2i (If array [1])
5. The right child of i will be at index 2i + 2 (If array [0])
6. The right child of i will be at index 2i + 1 (If array [1])





[1] [2]

[0] [1]

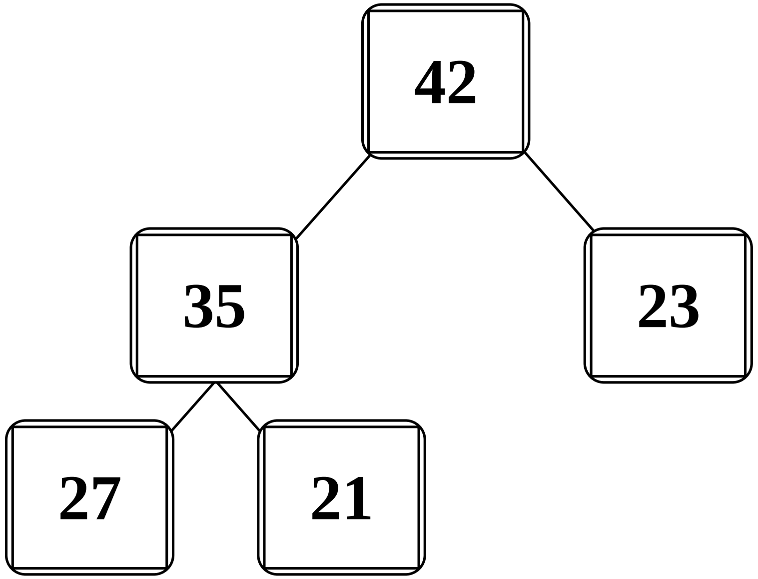
Which one is better?

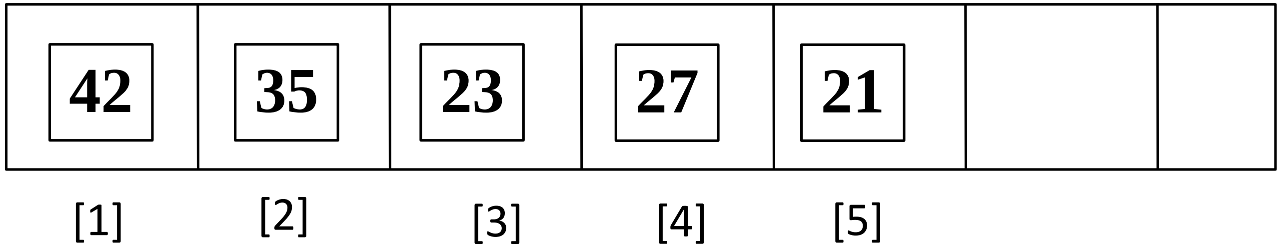
- Index = i

Implementing a Heap (Array to Tree conversion using mathematics)

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children.

1. The parent child of i will be at index $\left\lfloor \frac{i}{2} \right\rfloor$ (If array [1])
2. The left child of i will be at index 2i (If array [1])
3. The right child of i will be at index 2i + 1 (If array [1])

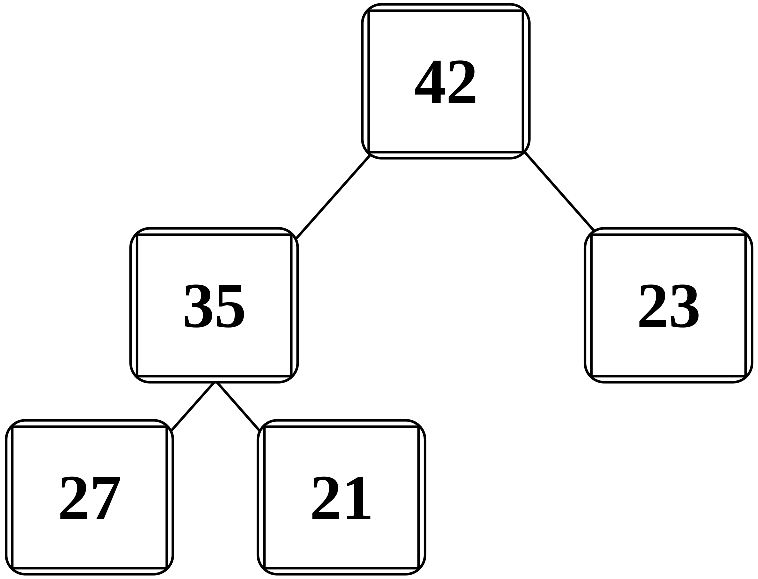




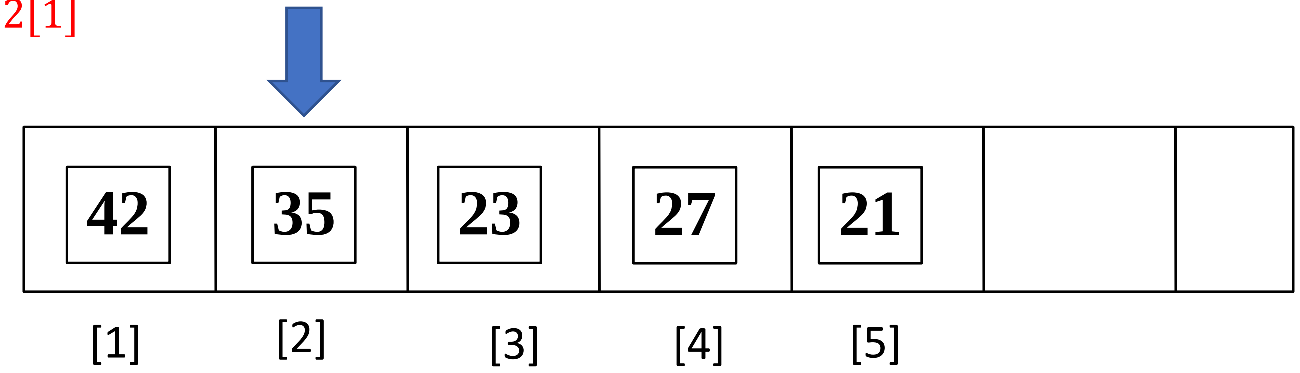
Implementing a Heap (Array to Tree conversion using mathematics)

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children.

1. The parent child of i will be at index $\left\lfloor \frac{i}{2} \right\rfloor$ (If array [1])
2. The left child of i will be at index 2i (If array [1])
3. The right child of i will be at index 2i + 1 (If array [1])



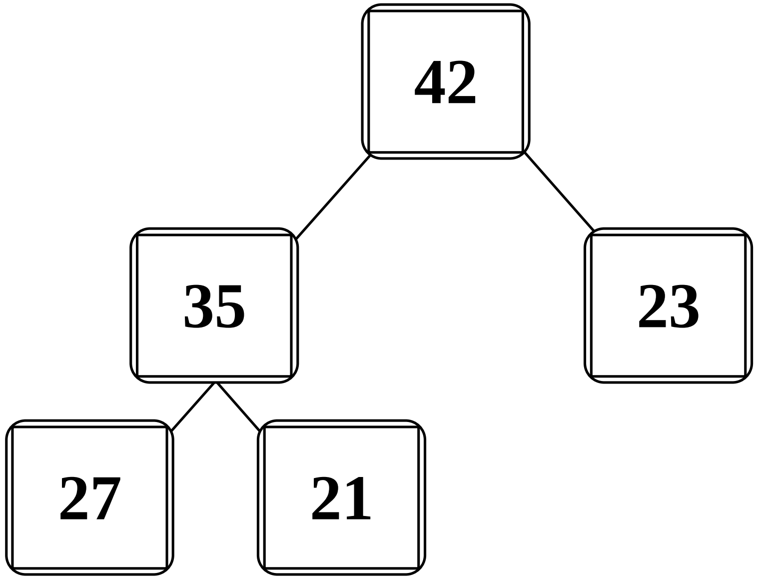
1. Find parent of 35 [2]
2. $\left\lfloor \frac{2}{2} \right\rfloor = 1 \Rightarrow Index\ value \Rightarrow 42[1]$



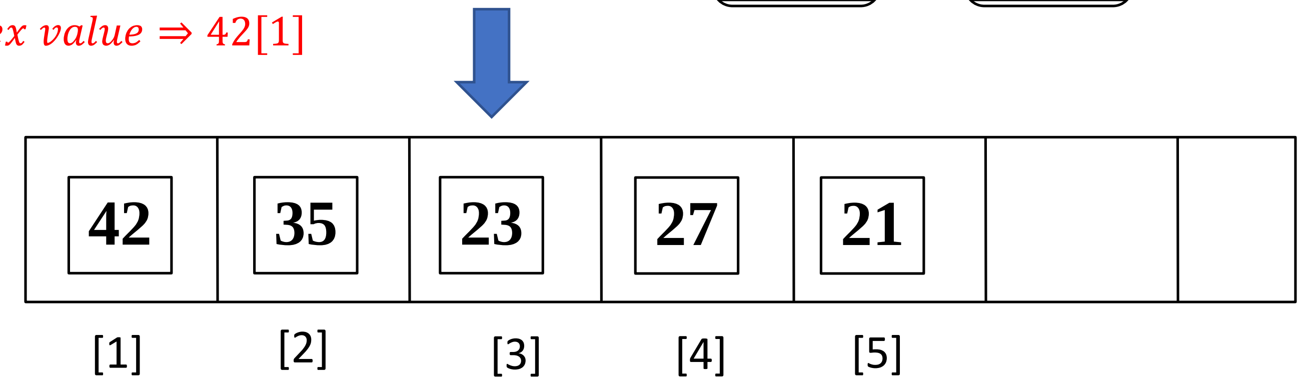
Implementing a Heap (Array to Tree conversion using mathematics)

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children.

1. The parent child of i will be at index $\left\lfloor \frac{i}{2} \right\rfloor$ (If array [1])
2. The left child of i will be at index 2i (If array [1])
3. The right child of i will be at index 2i + 1 (If array [1])



1. Find parent of 23 [3]
2. $\left\lfloor \frac{3}{2} \right\rfloor = 1.5 \Rightarrow \mathit{Floor\ Function\ Index\ value} \Rightarrow 42[1]$

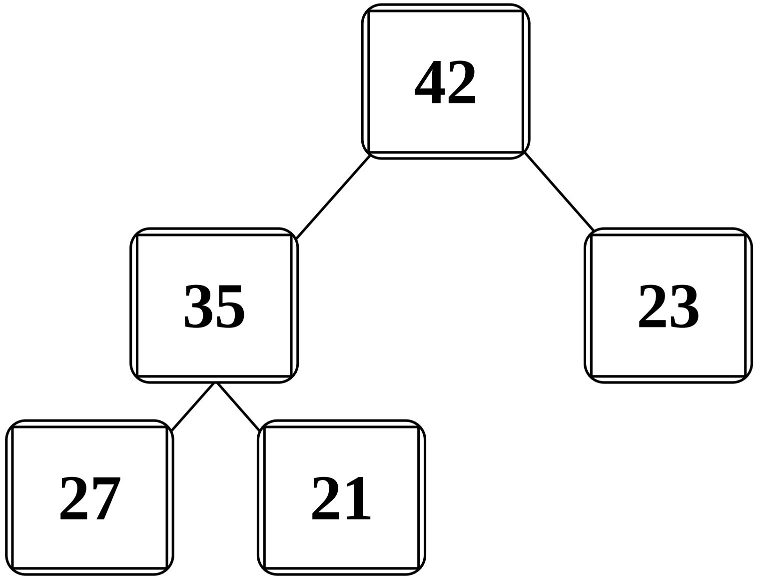


- Floor function => $\lfloor 2.3 \rfloor = 2$
- Ceiling function => $\lceil 4.5 \rceil = 5$

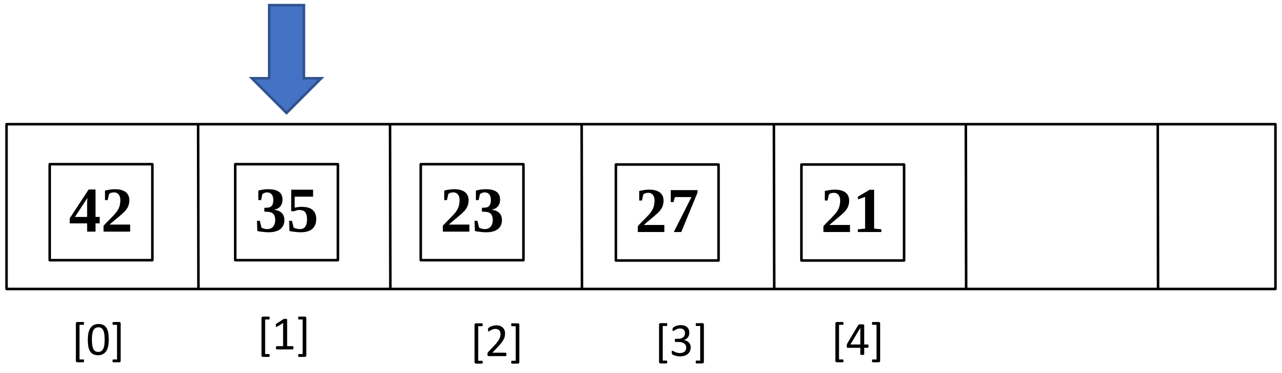
Implementing a Heap (Array to Tree conversion using mathematics)

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children.

1. The parent child of i will be at index $\left\lfloor \frac{i-1}{2} \right\rfloor$ (If array [0])
2. The left child of i will be at index 2i + 1 (If array [0])
3. The right child of i will be at index 2i + 2 (If array [0])



1. Find parent of 35 [1]
2. $\left\lfloor \frac{1-1}{2} \right\rfloor = 0 \Rightarrow Index\ value \Rightarrow 42[0]$

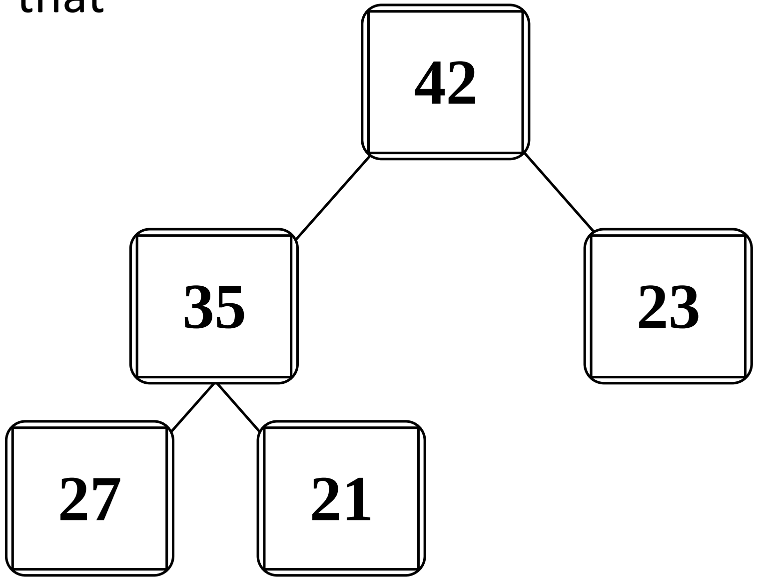


- Floor function => ⌊ 0.5 ⌋ = 0
- Ceiling function => ⌈ 0.5 ⌉ = 1

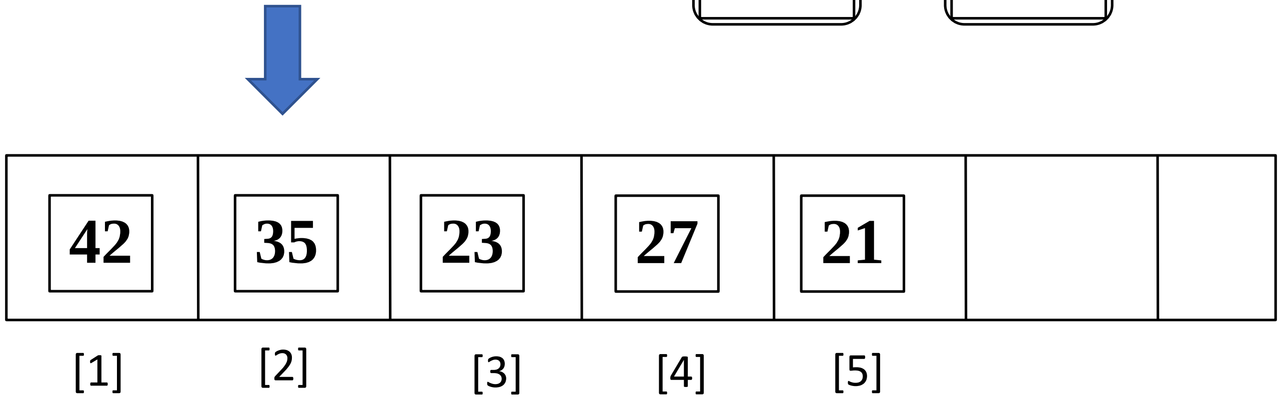
Implementing a Heap (Array to Tree conversion using mathematics)

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children.

1. The parent child of i will be at index $\left\lfloor \frac{i}{2} \right\rfloor$ (If array [1])
    1. Find parent of 35 [2]
    2. $\left\lfloor \frac{2}{2} \right\rfloor = 1 \Rightarrow Index\ value \Rightarrow 42[1]$
2. The left child of i will be at index 2i (If array [1])
3. The right child of i will be at index 2i + 1 (If array [1])



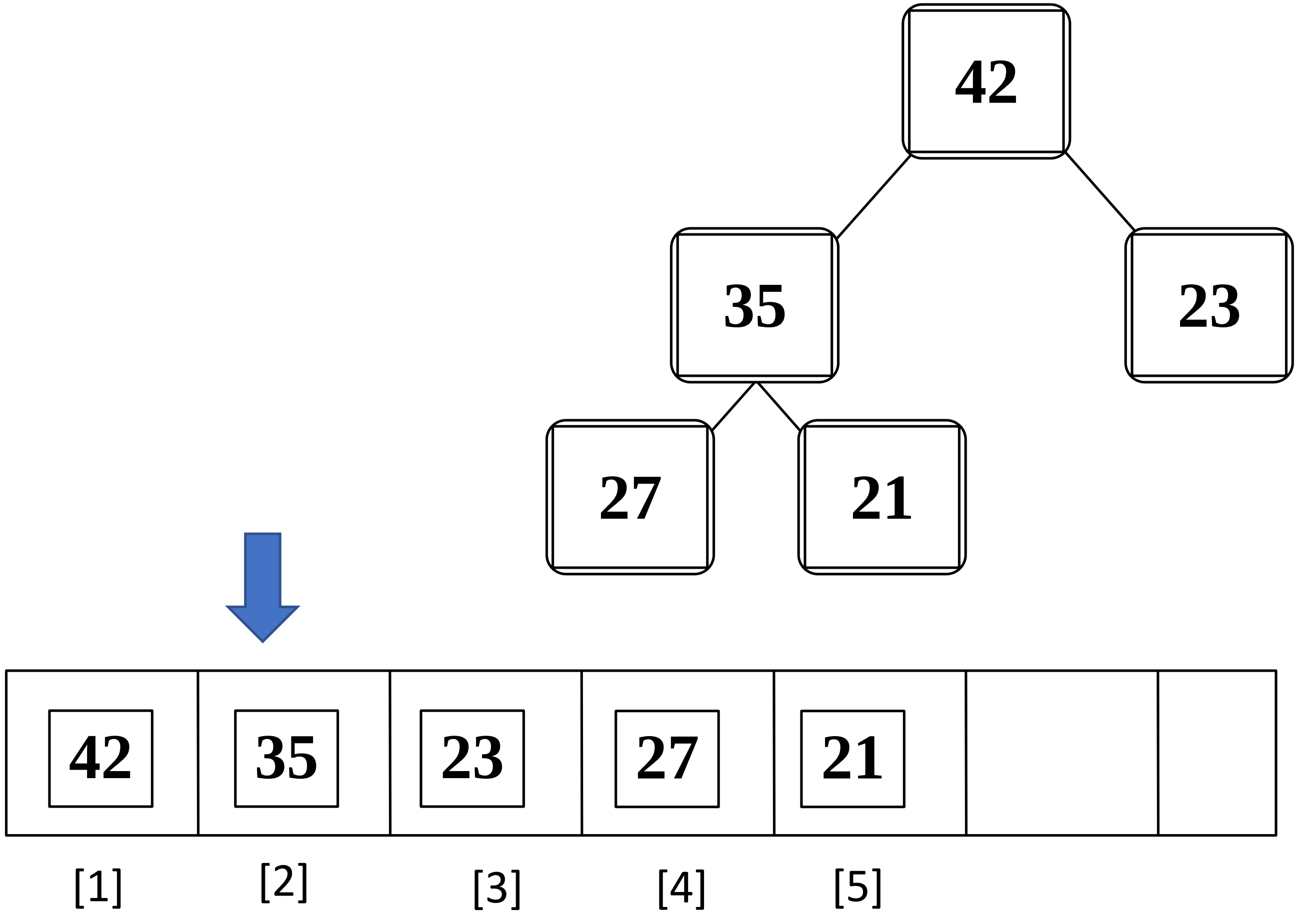
1. Find left-child of 35 [2]
2. $2i(i = 2) = 4 \Rightarrow Index\ value \Rightarrow 27[4]$



Implementing a Heap (Array to Tree conversion using mathematics)

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children.

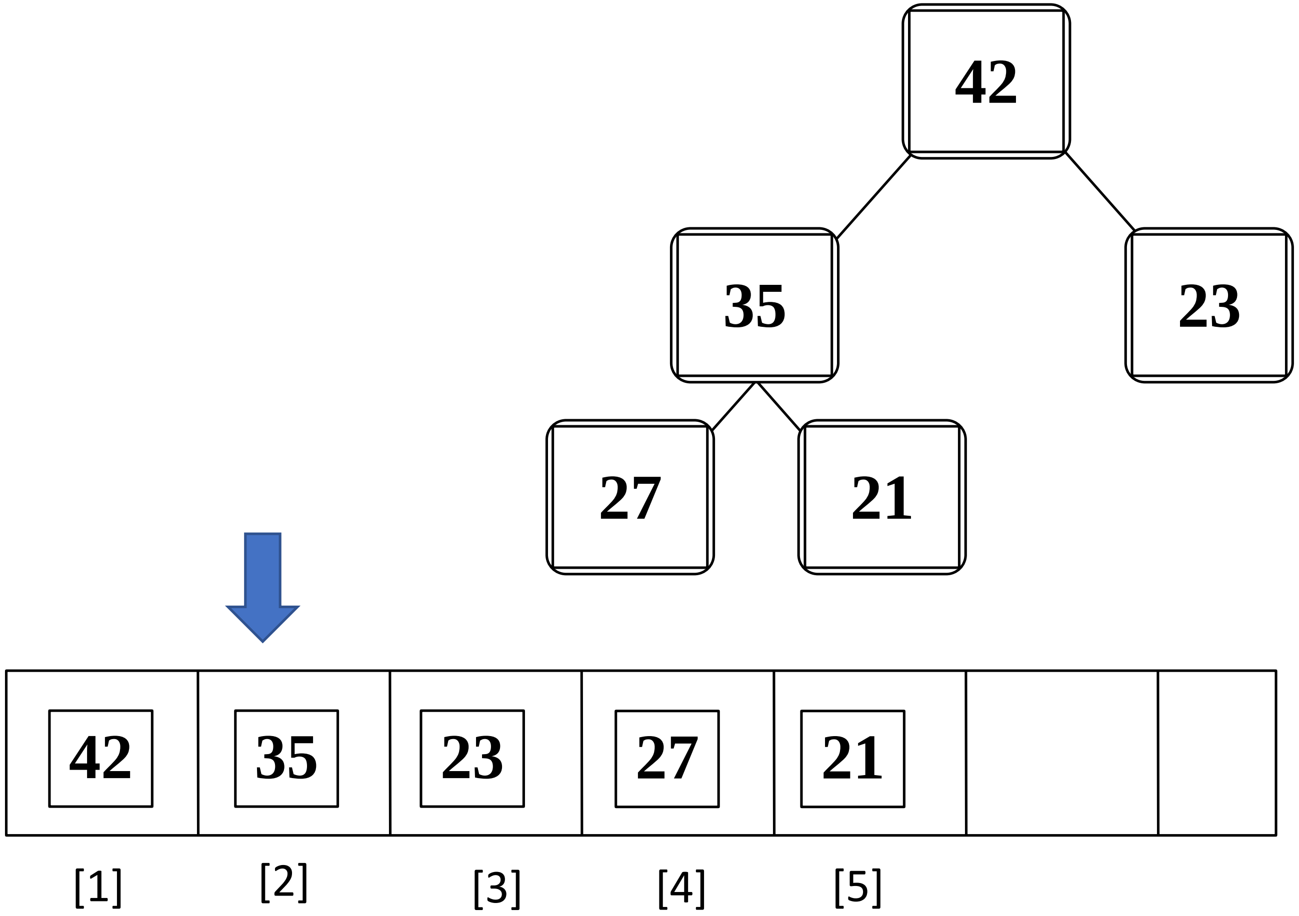
1. The parent child of i will be at index $\left\lfloor \frac{i}{2} \right\rfloor$ (If array [1])
    1. Find parent of 35 [2]
    2. $\left\lfloor \frac{2}{2} \right\rfloor = 1 \Rightarrow Index\ value \Rightarrow 42[1]$
2. The left child of i will be at index 2i (If array [1])
    1. Find left-child of 35 [2]
    2. $2i(i = 2) = 4 \Rightarrow Index\ value \Rightarrow 27[4]$
3. The right child of i will be at index 2i + 1 (If array [1])
    1. $2i + 1(i = 2) = 5 \Rightarrow Index\ value \Rightarrow 21[5]$



Implementing a Heap (Array to Tree conversion using mathematics)

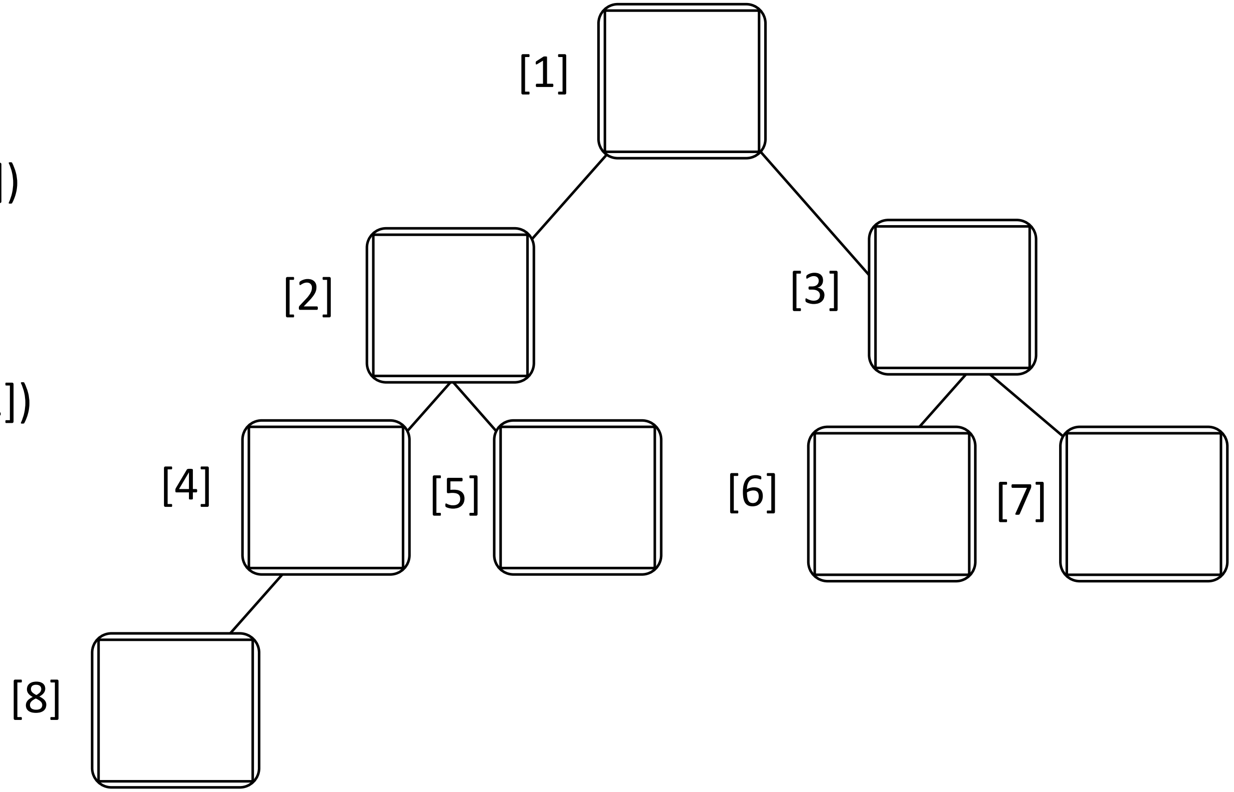
- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children.

1. The parent child of i will be at index $\left\lfloor\frac{i}{2}\right\rfloor$ (If array [1])
    1. Find parent of 35 [2]
    2. $\left\lfloor\frac{2}{2}\right\rfloor = 1 \Rightarrow Index\ value \Rightarrow 42[1]$
2. The left child of i will be at index 2i (If array [1])
    1. Find left-child of 35 [2]
    2. $2i(i = 2) = 4 \Rightarrow Index\ value \Rightarrow 27[4]$
3. The right child of i will be at index 2i + 1 (If array [1])
    1. $2i + 1(i = 2) = 5 \Rightarrow Index\ value \Rightarrow 21[5]$



From Array to Max Heap Tree (Using Formulas)

1. The parent child of i will be at index $\left\lfloor \frac{i}{2} \right\rfloor$ (If array [1])
2. The left child of i will be at index 2i (If array [1])
3. The right child of i will be at index 2i + 1 (If array [1])

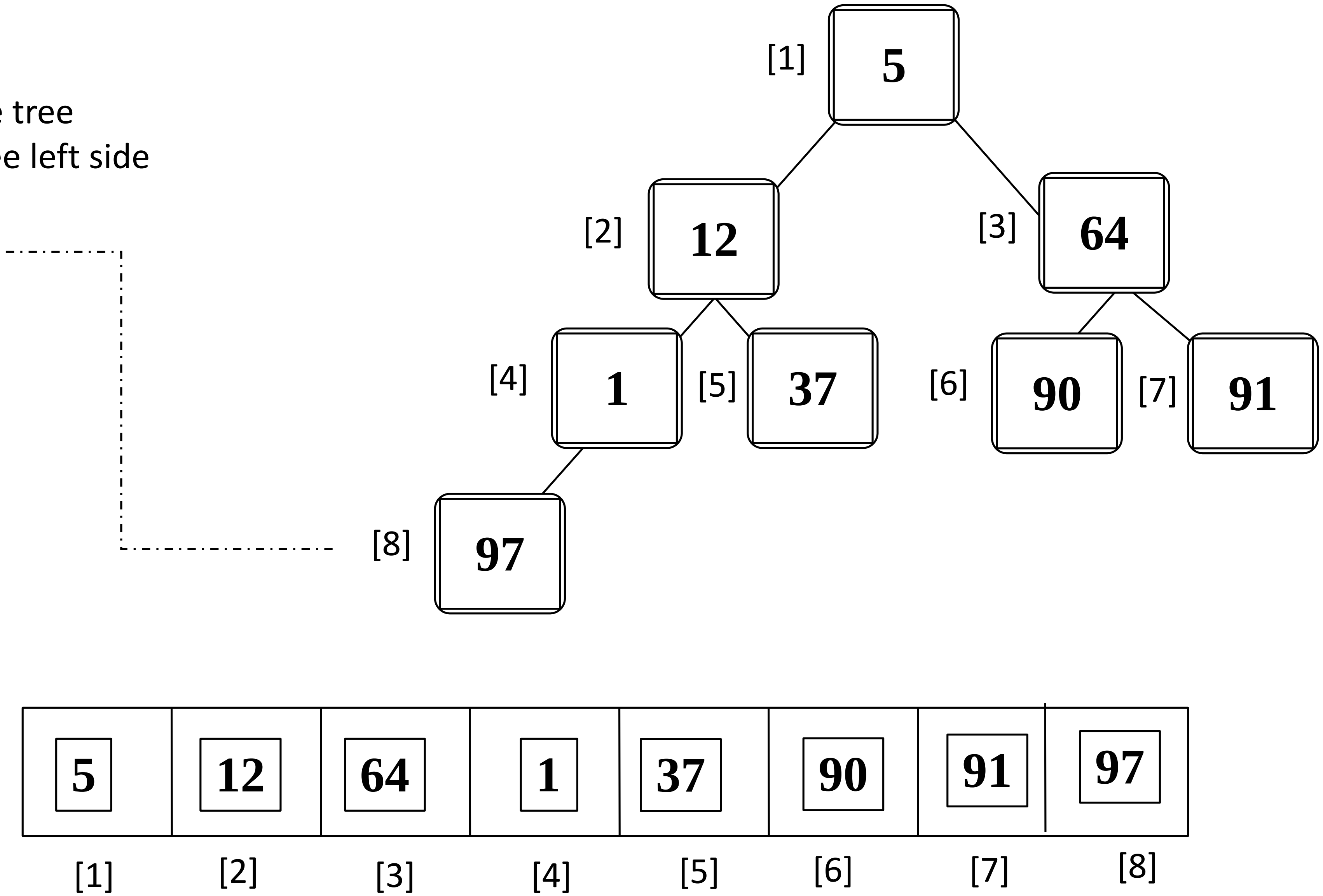


5	12	64	1	37	90	91	97
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

From Array to Max Heap Tree (Using Formulas)

1. First map the array into the tree
2. Began from the last sab-tree left side

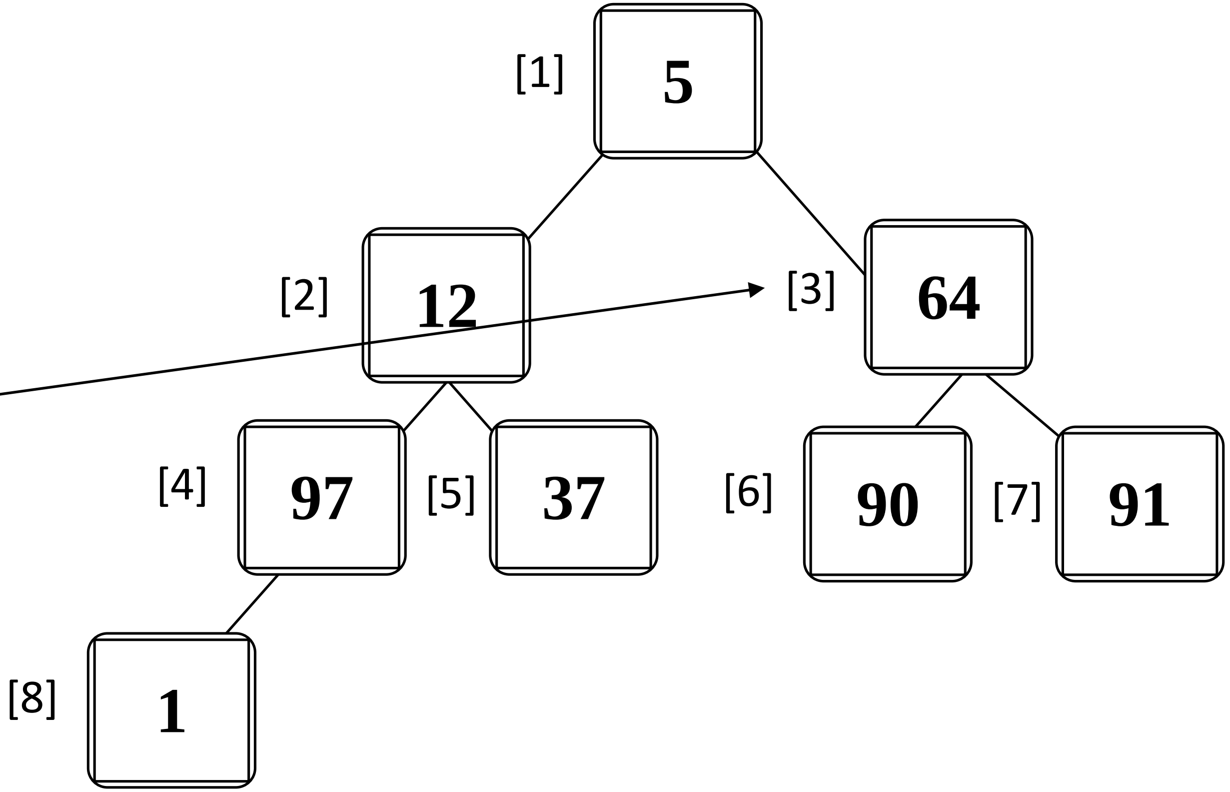
- index ⌊i/2⌋ (If array [1])
- i = 8 => ⌊8/2⌋ = 4



From Array to Max Heap Tree (Using Formulas)

Began from the last sab-tree

- index $\lfloor i/2 \rfloor$ (If array [1])
- $i = 8 \quad \Rightarrow \lfloor 8/2 \rfloor = 4$
- Node is moved to new position, next
- i-1
- Now value of i was 4 so:
- i-1 = 4-1 = 3
  - Now Check if i=3 follow Max Heap



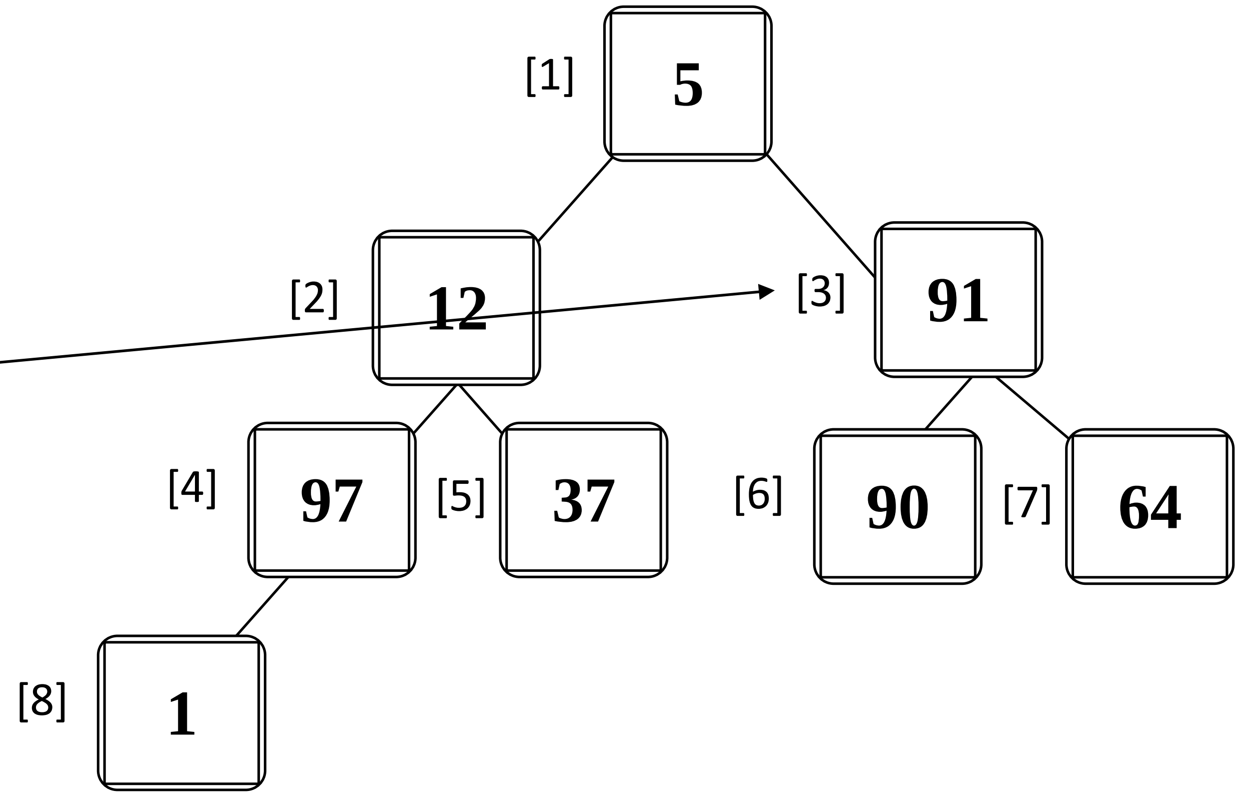
After moving 97 we check if it follow Max Heap

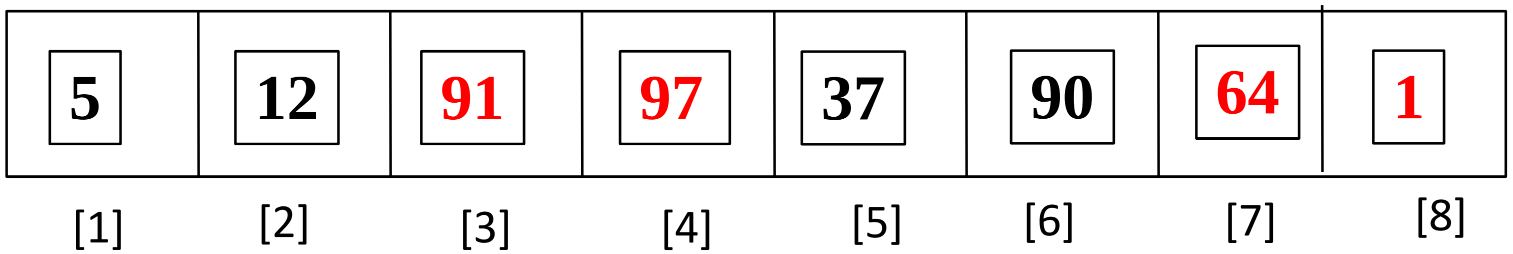
5	12	64	97	37	90	91	1
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

From Array to Max Heap Tree (Using Formulas)

Began from the last sab-tree

- index $\lfloor i/2 \rfloor$ (If array [1])
- i = 8 => $\lfloor 8/2 \rfloor$ = 4
- i-1
- Now value of i was 4 so:
- 4-1 = 3
- Child note with larger value becomes root of the sub-tree

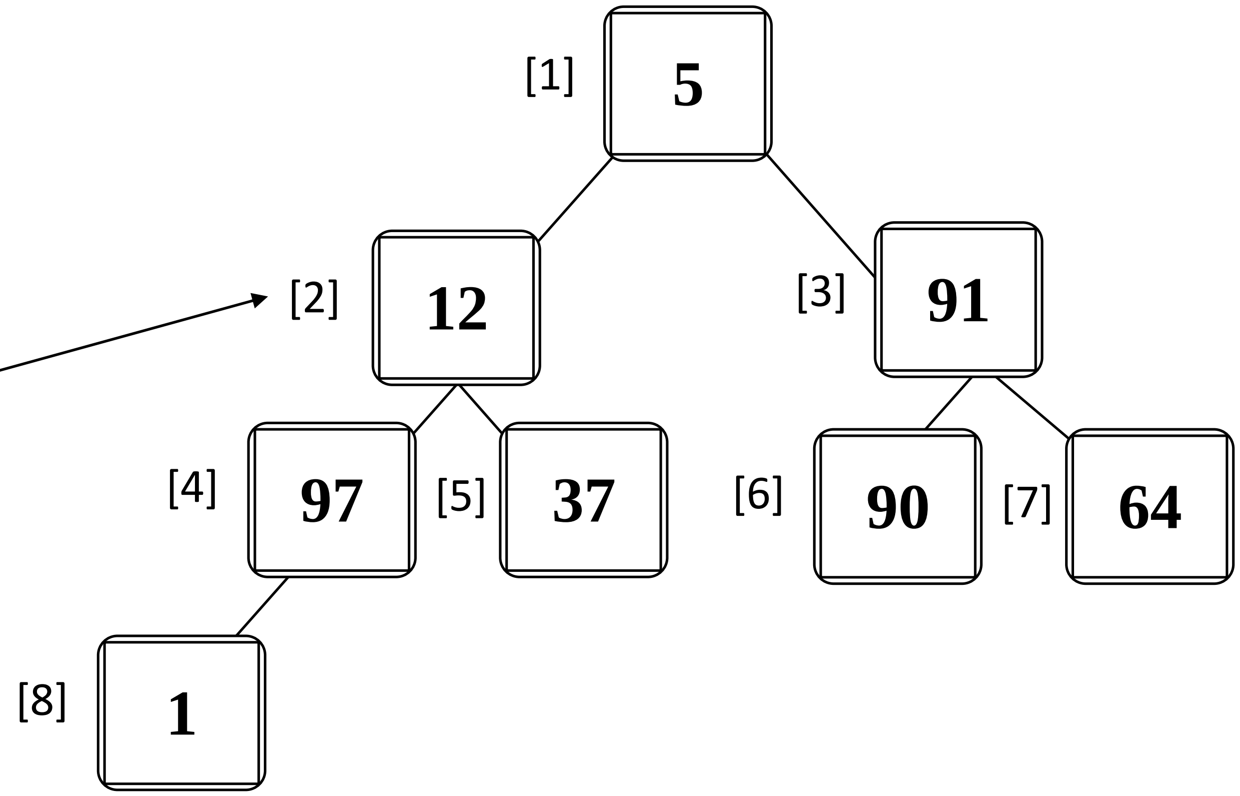




From Array to Max Heap Tree (Using Formulas)

Began from the last sab-tree

- index $\lfloor i/2 \rfloor$ (If array [1])
- i = 8 => $\lfloor 8/2 \rfloor$ = 4
- i-1
- Now value of i was 4 so:
- 4-1 = 3
- Now value of i was 3 so:
- 3-1 = 2
- Child note with larger value becomes root of the sub-tree

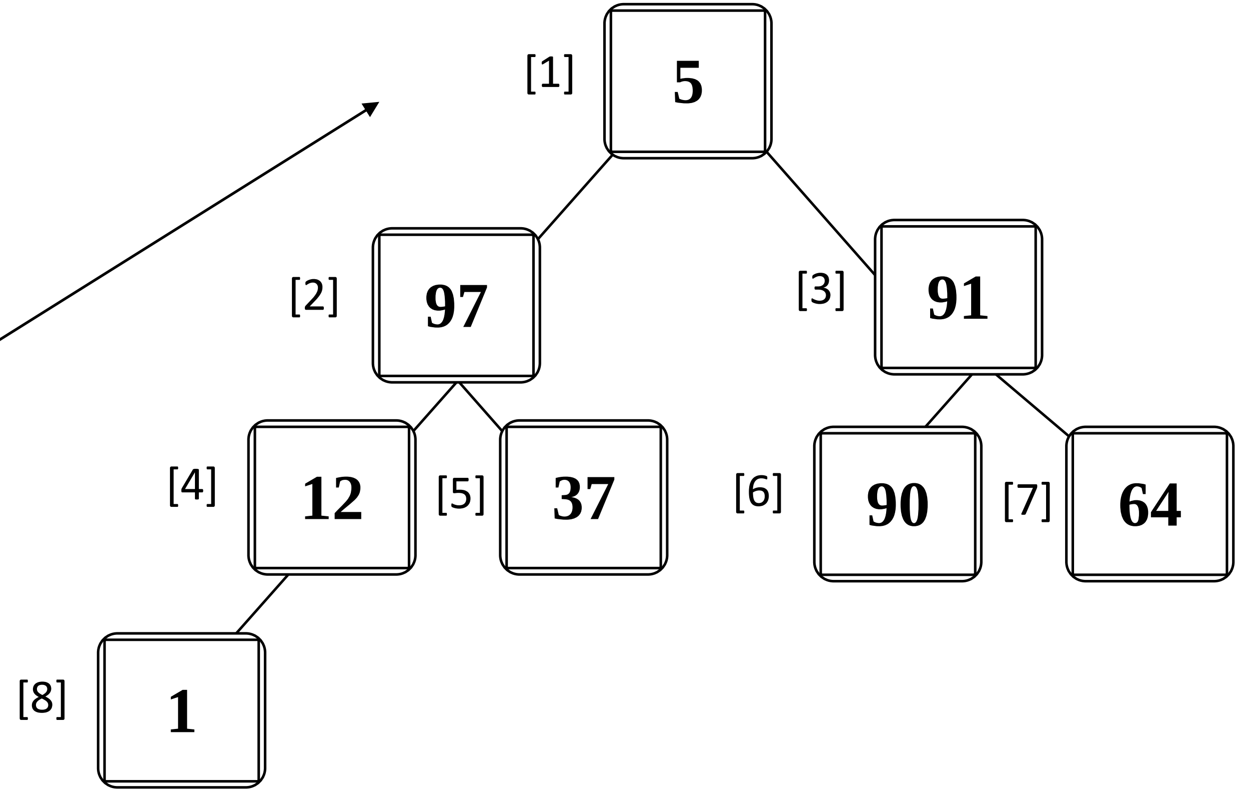


5	12	91	97	37	90	64	1
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

From Array to Max Heap Tree (Using Formulas)

Began from the last sab-tree

- index $\lfloor i/2 \rfloor$ (If array [1])
- i = 8   => $\lfloor 8/2 \rfloor$ = 4
- i-1
- Now value of i was 4 so:
- 4-1 = 3
- Now value of i was 3 so:
- 3-1 = 2
- Now value of i was 2 so:
- 2-1 = 1
- Child note with larger value becomes root of the sub-tree

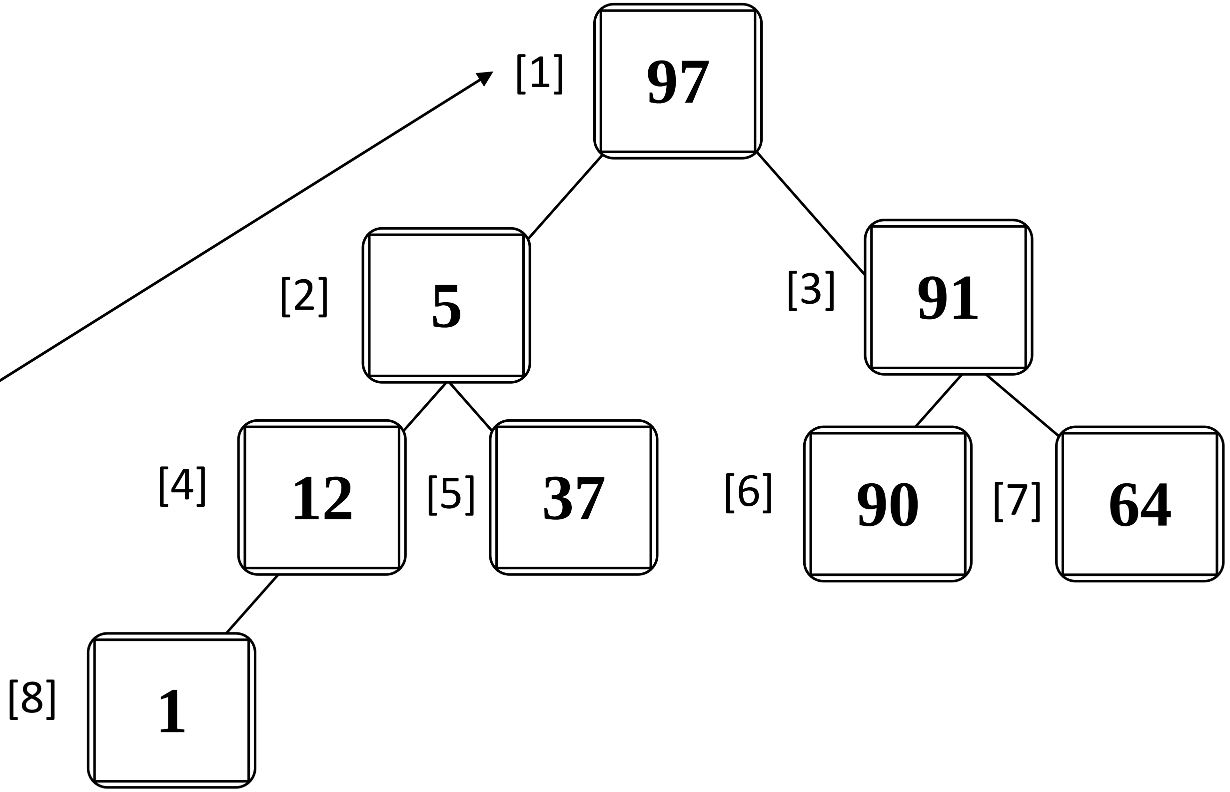


5	97	91	12	37	90	64	1
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

From Array to Max Heap Tree (Using Formulas)

Began from the last sab-tree
Began from the last sab-tree

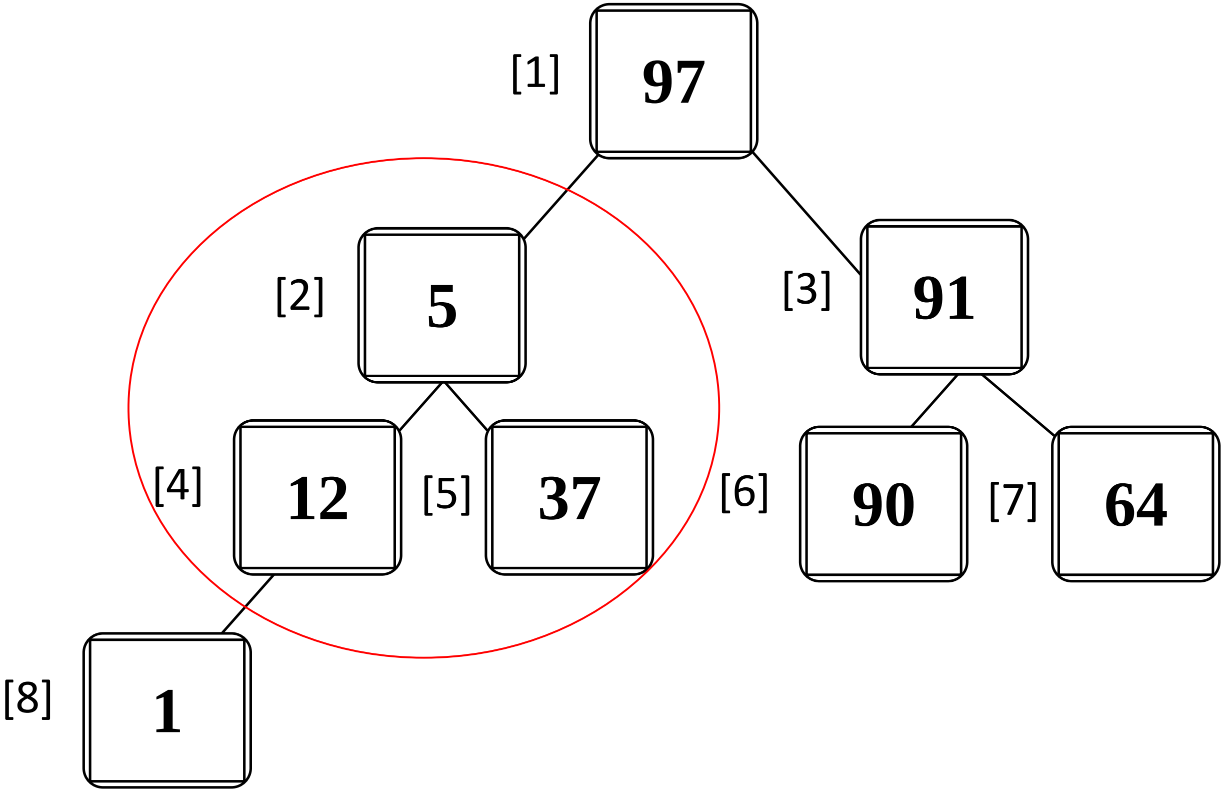
- index $\lfloor i/2 \rfloor$ (If array [1])
- i = 8 => $\lfloor 8/2 \rfloor$ = 4
- i-1
- Now value of i was 4 so:
- 4-1 = 3
- Now value of i was 3 so:
- 3-1 = 2
- Now value of i was 2 so:
- 2-1 = 1
- Child note with larger value becomes root of the sub-tree



97	5	91	12	37	90	64	1
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

From Array to Max Heap Tree (Using Formulas)

Is it complete ?

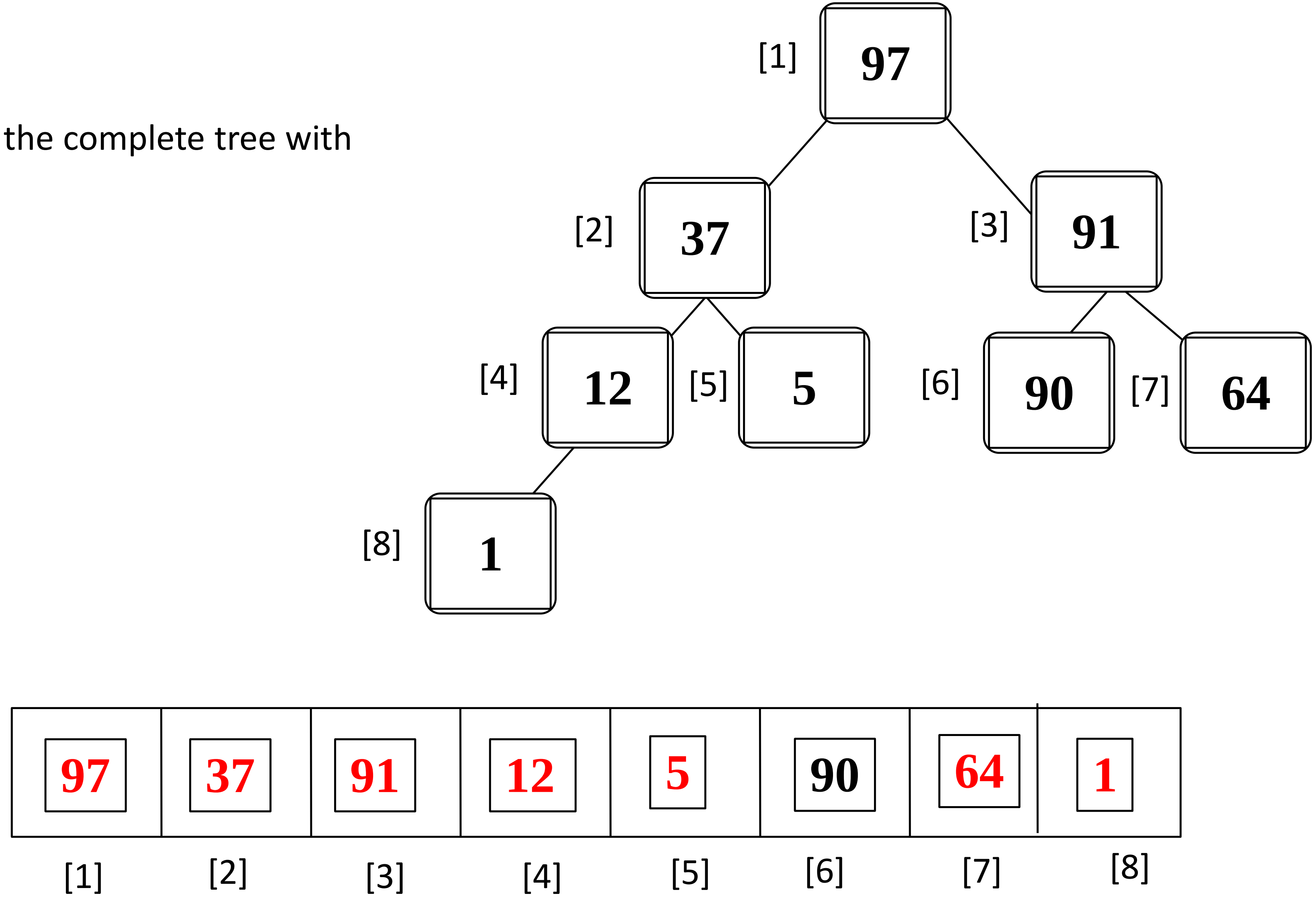


We run the same round on this tree again to achieve the final tree

97	5	91	12	37	90	64	1
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

From Array to Max Heap Tree (Using Formulas)

- After the 2nd round we have the complete tree with array representation.



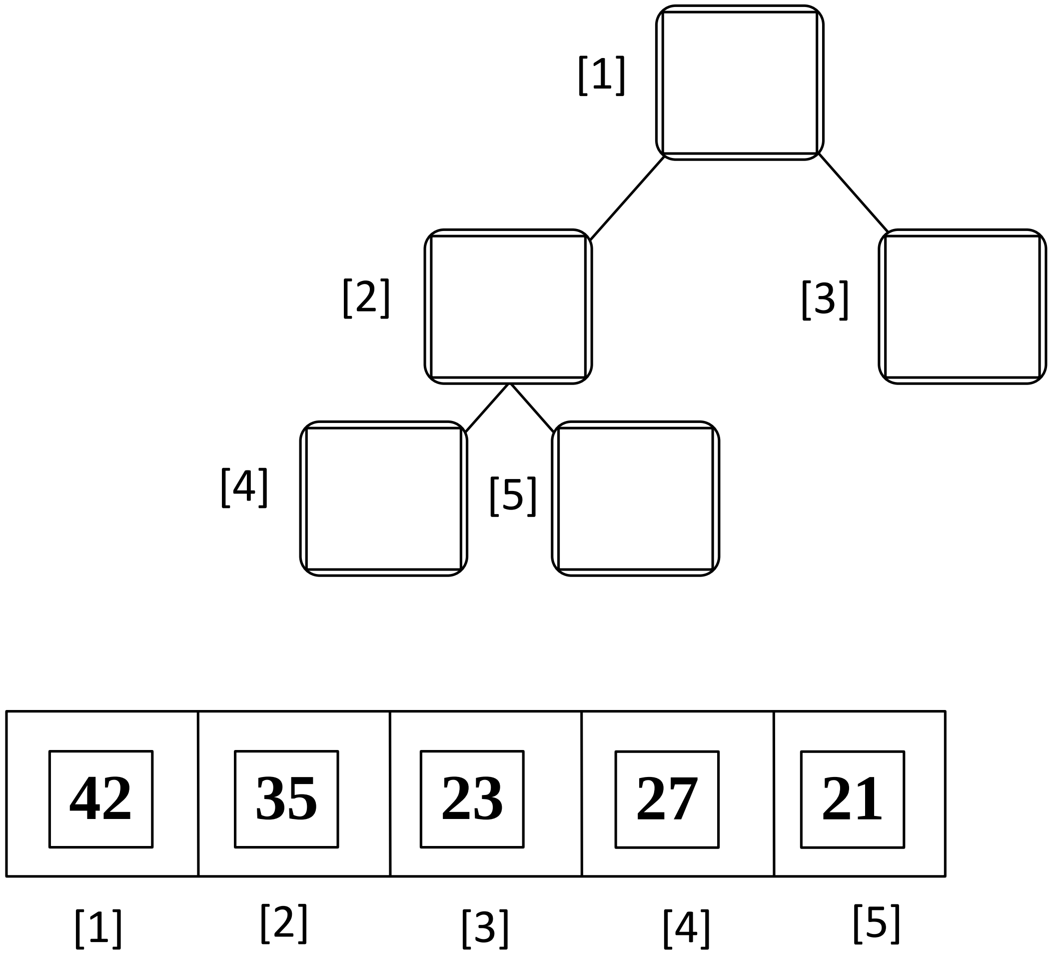
From Array to Min Heap Tree (Using Formulas)

Try this by your self using the formulas in previous slides

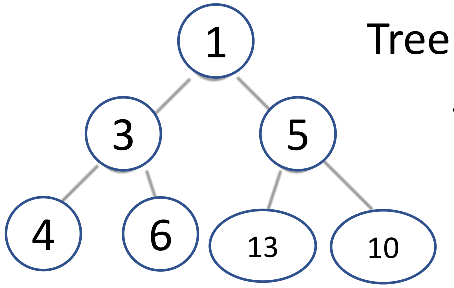
Condition:

Parent/Root node key <= Child node Key

1. The parent child of i will be at index $\left\lfloor \frac{i-1}{2} \right\rfloor$ (If array [0])
2. The parent child of i will be at index $\left\lfloor \frac{i}{2} \right\rfloor$ (If array [1])
3. The left child of i will be at index 2i + 1 (If array [0])
4. The left child of i will be at index 2i (If array [1])
5. The right child of i will be at index 2i + 2 (If array [0])
6. The right child of i will be at index 2i + 1 (If array [1])



```c
1  #include <stdio.h>
2  // To heapify a subtree rooted with node i which is
3  // an index in arr[]. N is size of heap
4  void swap(int *a, int *b)
5  {
6      int tmp = *a;
7      *a = *b;
8      *b = tmp;
9  }
10 void heapify(int arr[], int N, int i)
11 {
12     int largest = i; // Initialize largest as root
13     int l = 2 * i + 1; // left = 2*i + 1
14     int r = 2 * i + 2; // right = 2*i + 2
15     // If left child is larger than root
16     if (l < N && arr[l] > arr[largest])
17         largest = l;
18     // If right child is larger than largest so far
19     if (r < N && arr[r] > arr[largest])
20         largest = r;
21     // If largest is not root
22     if (largest != i) {
23         swap(&arr[i], &arr[largest]);
24     // Recursively heapify the affected sub-tree
25         heapify(arr, N, largest);
26     }
27 }
```



Tree representation of Array (line 50)

```c
28 // Function to build a Max-Heap from the given array
29 void buildHeap(int arr[], int N)
30 {
31     // Index of last non-leaf node
32     int startIdx = (N / 2) - 1;
33     // Perform reverse level order traversal
34     // from last non-leaf node and heapify
35     // each node
36     for (int i = startIdx; i >= 0; i--) {
37         heapify(arr, N, i);
38     }
39 }
40 // Function to print the heap array
41 void printHeap(int arr[], int N)
42 {
43     printf("Array representation of Heap is:\n");
44     for (int i = 0; i < N; ++i)
45         printf("%d ",arr[i]);
46     printf("\n");
47 }
48 int main()
49 {
50     int arr[] = {1, 3, 5, 4, 6, 13, 10};
51     int N = sizeof(arr) / sizeof(arr[0]); //N=7
52     buildHeap(arr, N);
53     printHeap(arr, N);
54     return 0;
55 }
```

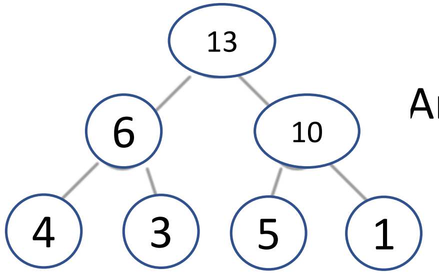
Source Video

```c
int main()
{
    int arr[] = {1, 3, 5, 4, 6, 13, 10};
    int N = sizeof(arr) / sizeof(arr[0]);
    printf("58.Int N =%d \n", N);
    buildHeap(arr, N);
    printHeap(arr, N);
    return 0;
}
```

```c
void heapify(int arr[], int N, int i)
{
    int largest = i; // Initialize largest as root
    printf("13.arr[%d] = %d \n", largest, arr[largest]);
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2
    // If left child is larger than root
    if (l < N && arr[l] > arr[largest])
        largest = l;
        printf("19.arr[%d] =%d \n",l, arr[l]);
    // If right child is larger than largest so far
    if (r < N && arr[r] > arr[largest])
        largest = r;
        printf("23.arr[%d] =%d \n", r, arr[r]);
    // If largest is not root
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        printf("27.arr[%d] =%d, arr[%d] =%d\n", i, arr[i], largest, arr[largest]);
        printf("-----------------\n");
    // Recursively heapify the affected sub-tree
        heapify(arr, N, largest);
    }
}
```

Index	Value
0	1
1	3
2	5
3	4
4	6
5	13
6	10

```c
1  #include <stdio.h>
2  // To heapify a subtree rooted with node i which is
3  // an index in arr[]. N is size of heap
4  void swap(int *a, int *b)
5  {
6      int tmp = *a;
7      *a = *b;
8      *b = tmp;
9  }
10 void heapify(int arr[], int N, int i)
11 {
12     int smallest = i; // Initialize smallest as root
13     int l = 2 * i + 1; // left = 2*i + 1
14     int r = 2 * i + 2; // right = 2*i + 2
15     // If left child is larger than root
16     if (l < N && arr[l] < arr[smallest])
17         smallest = l;
18     // If right child is larger than smallest so far
19     if (r < N && arr[r] < arr[smallest])
20         smallest = r;
21     // If smallest is not root
22     if (smallest != i) {
23         swap(&arr[i], &arr[smallest]);
24     // Recursively heapify the affected sub-tree
25         heapify(arr, N, smallest);
26     }
27 }
```



Tree representation of Array (line 50 - Max Heap)

```c
28 // Function to build a Min-Heap from the given array
29 void buildHeap(int arr[], int N)
30 {
31     // Index of last non-leaf node
32     int startIdx = (N / 2) - 1;
33     // Perform reverse level order traversal
34     // from last non-leaf node and heapify
35     // each node
36     for (int i = startIdx; i >= 0; i--) {
37         heapify(arr, N, i);
38     }
39 }
40 // Function to print the heap array
41 void printHeap(int arr[], int N)
42 {
43     printf("Array representation of Heap is:\n");
44     for (int i = 0; i < N; ++i)
45         printf("%d ",arr[i]);
46     printf("\n");
47 }
48 int main()
49 {
50     int arr[] = {13, 6, 10, 4, 3, 5, 1};
51     int N = sizeof(arr) / sizeof(arr[0]);
52     buildHeap(arr, N);
53     printHeap(arr, N);
54     return 0;
55 }
```