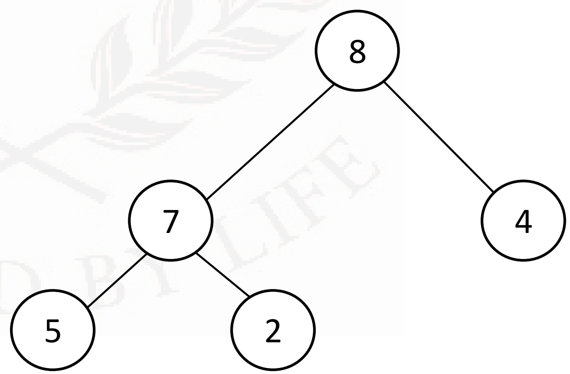
Heaps

Heaps

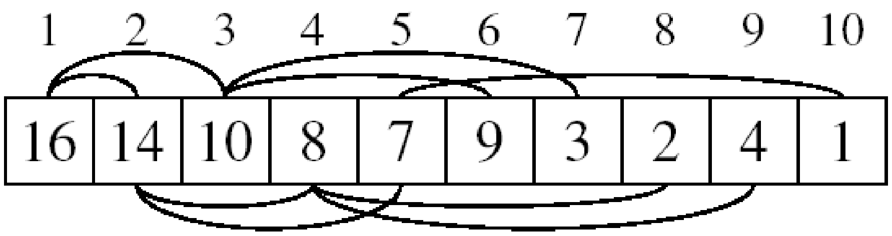
- *Def:* A **heap** is a complete binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node x
    Parent(x) ≥ x(max heap) , Parent(x)<=x(Min heap)

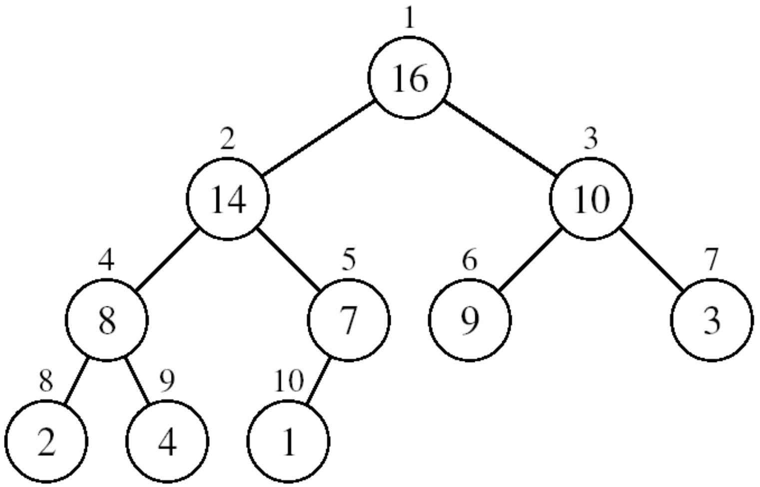
Heap



Array Representation of Heaps

- A heap can be stored as an array *A*.
  - Root of tree is $A[1]$
  - Left child of $A[i] = A[2i]$
  - Right child of $A[i] = A[2i + 1]$
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
  - Heapsize[A] ≤ length[A]
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves



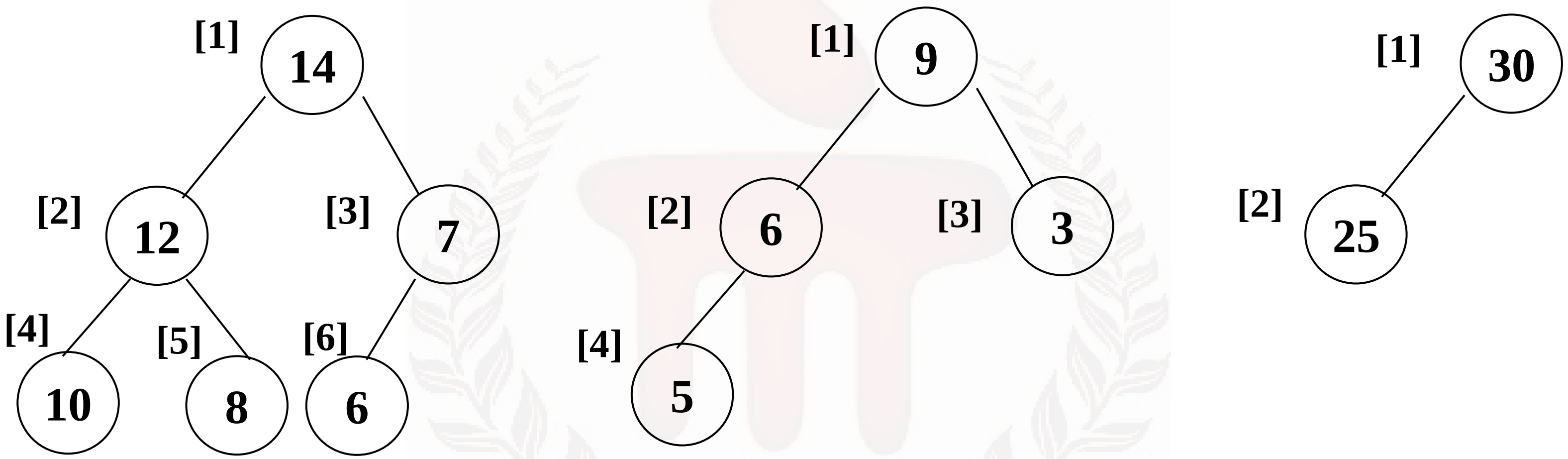


Heaps

- A *max tree* is a tree in which the key value in each node is greater than(equal to) the key values in its children. A *max heap* is a complete binary tree that is also a max tree.

- A *min tree* is a tree in which the key value in each node is smaller than(equal to) the key values in its children. A *min heap* is a complete binary tree that is also a min tree.

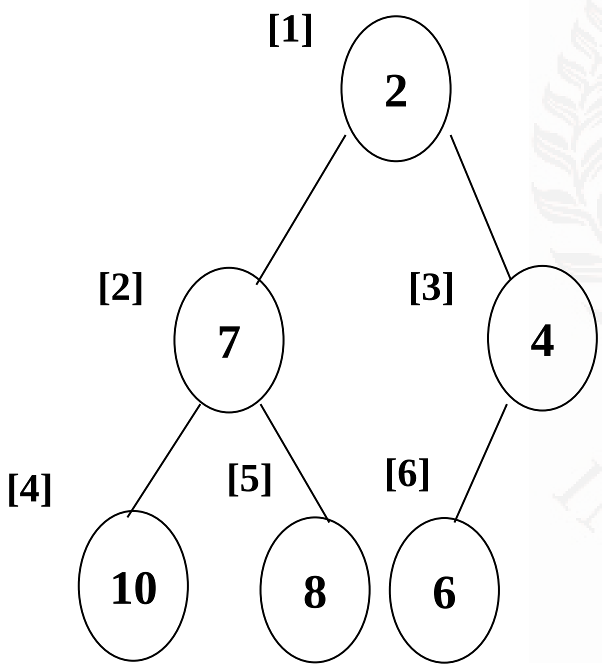
Max heap

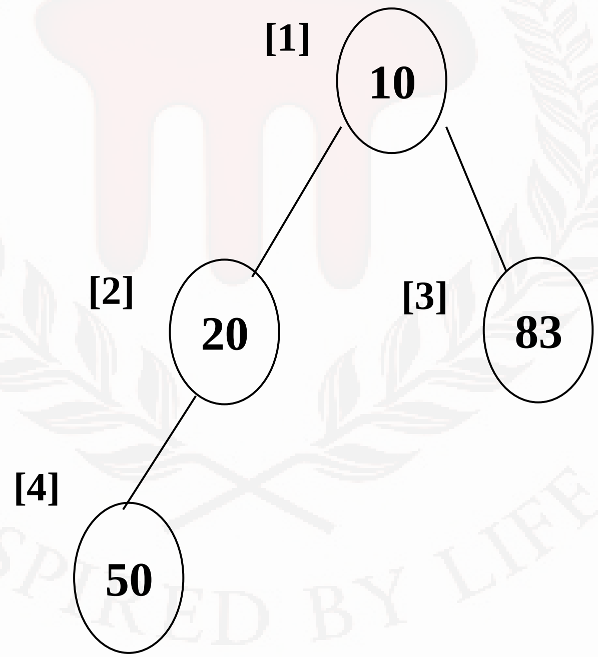


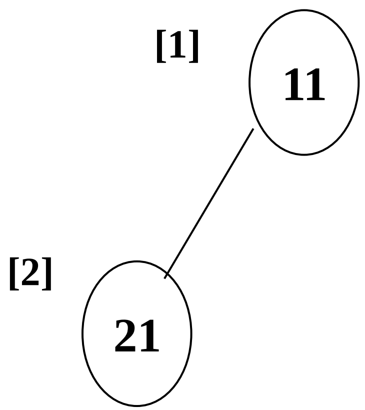
Property:

The root of max heap (min heap) contains the largest (smallest).

Min heap





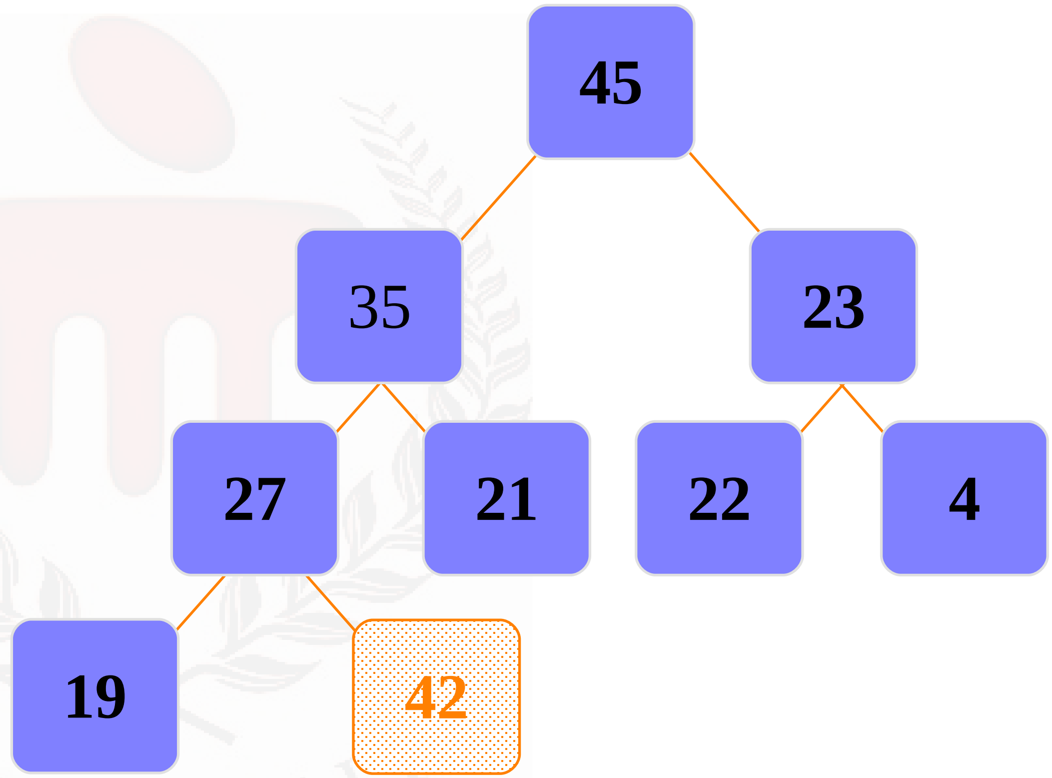


Steps to construct max heap

- Step 1 − Create a new node at the end of heap.
- Step 2 − Assign new value to the node.
- Step 3 − Compare the value of this child node with its parent.
- Step 4 − If value of parent is less than child, then swap them.
- Step 5 − Repeat step 3 & 4 until Heap property holds.

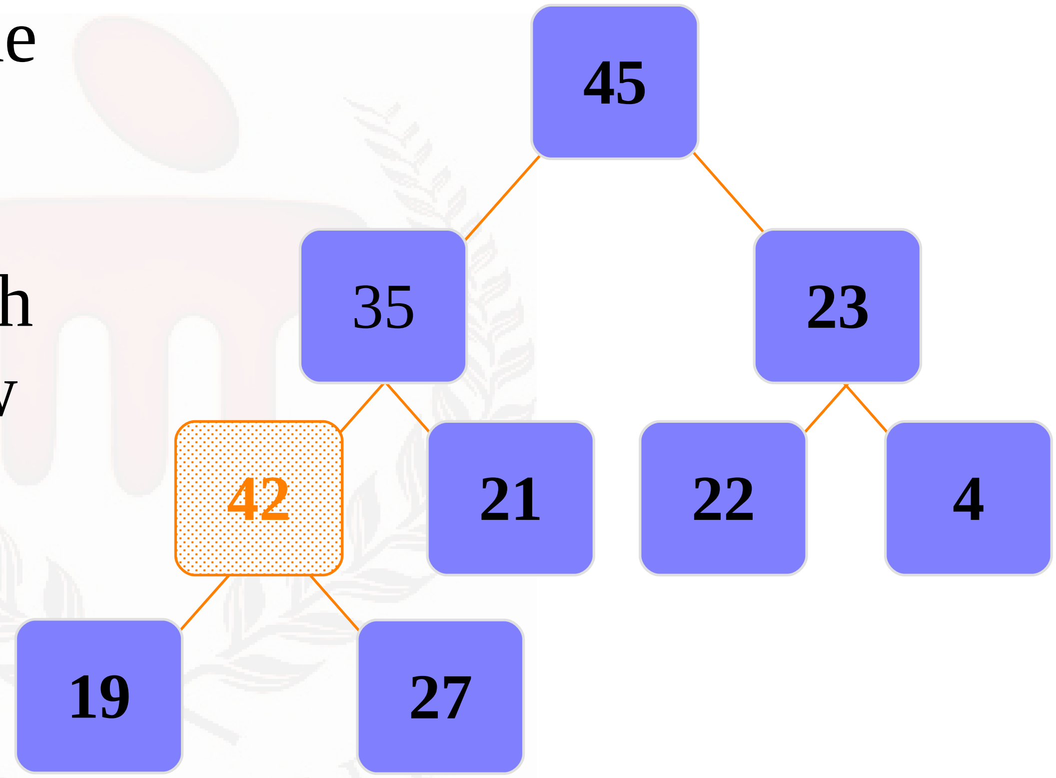
Adding a Node to a Heap

- 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.



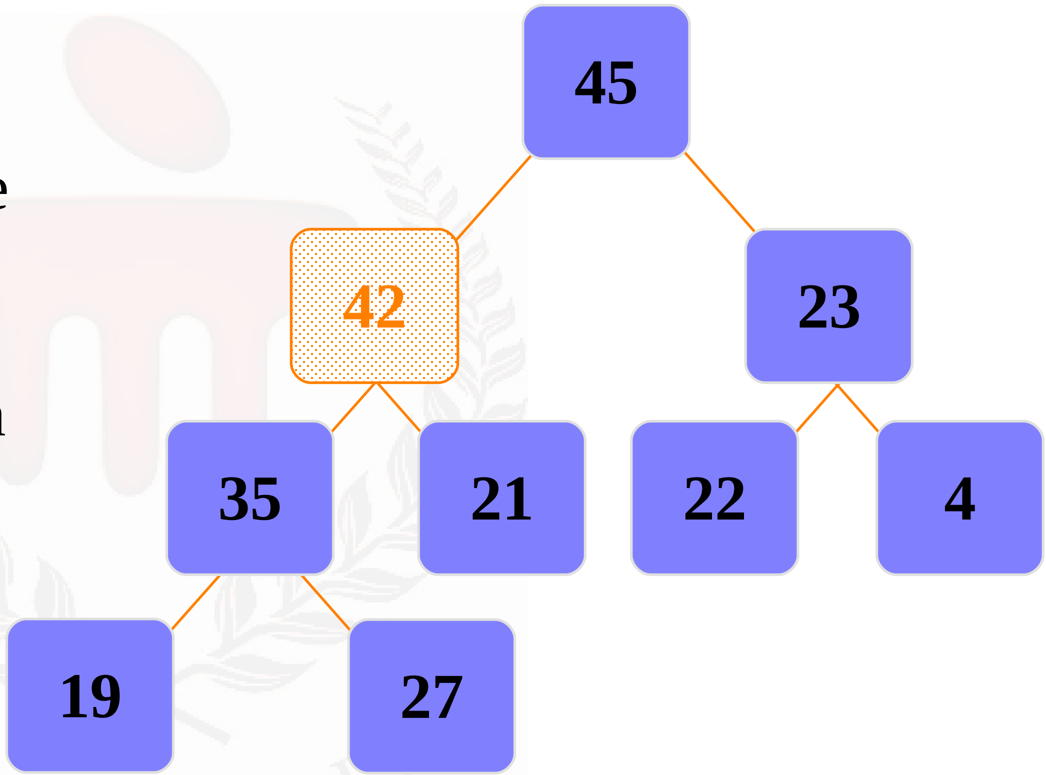
Adding a Node to a Heap

- 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.



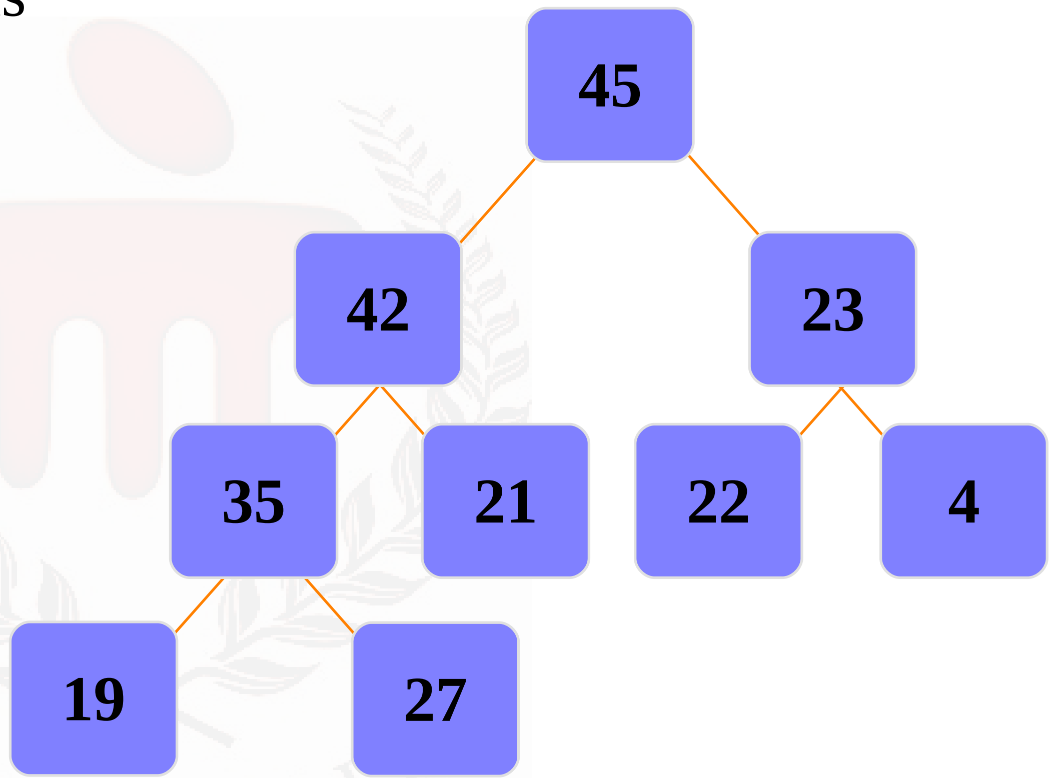
Adding a Node to a Heap

- 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**.

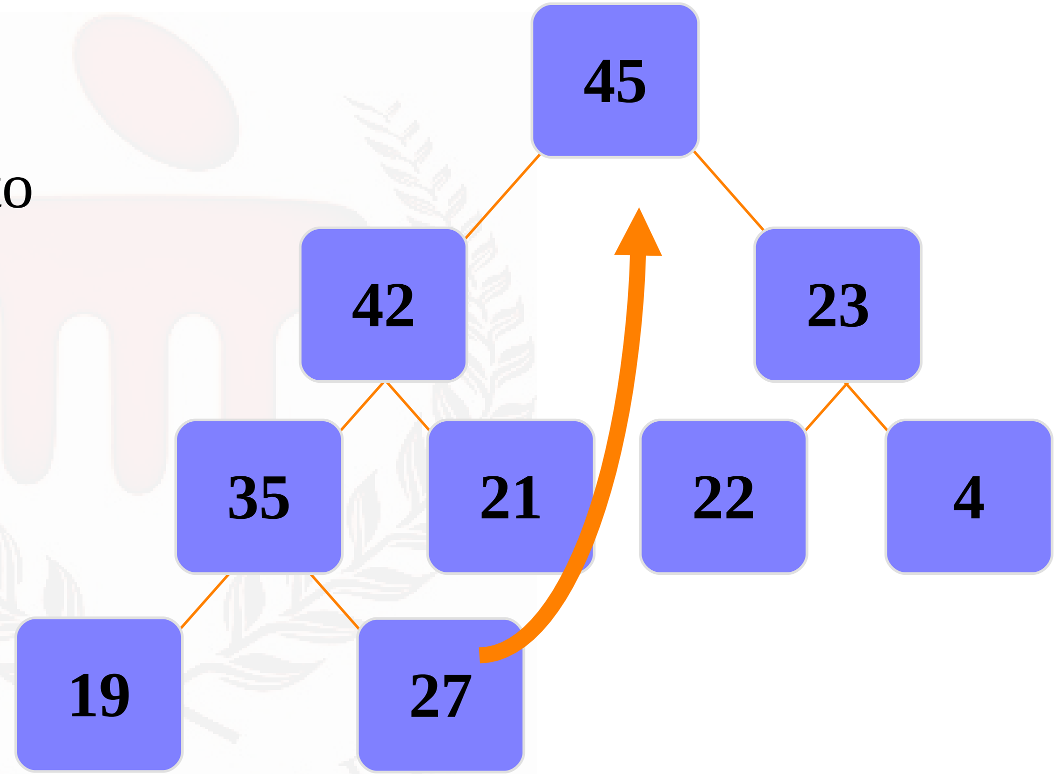


Deletion algorithm-Max - heap

- Step 1 – Remove root node.
- Step 2 – Move the last element of last level to root.
- Step 3 – Compare the value of this node with its child nodes.
- Step 4 – If value of the node is less than child nodes, then swap it with larger child.
- Step 5 – Repeat step 3 & 4 until Heap property holds.

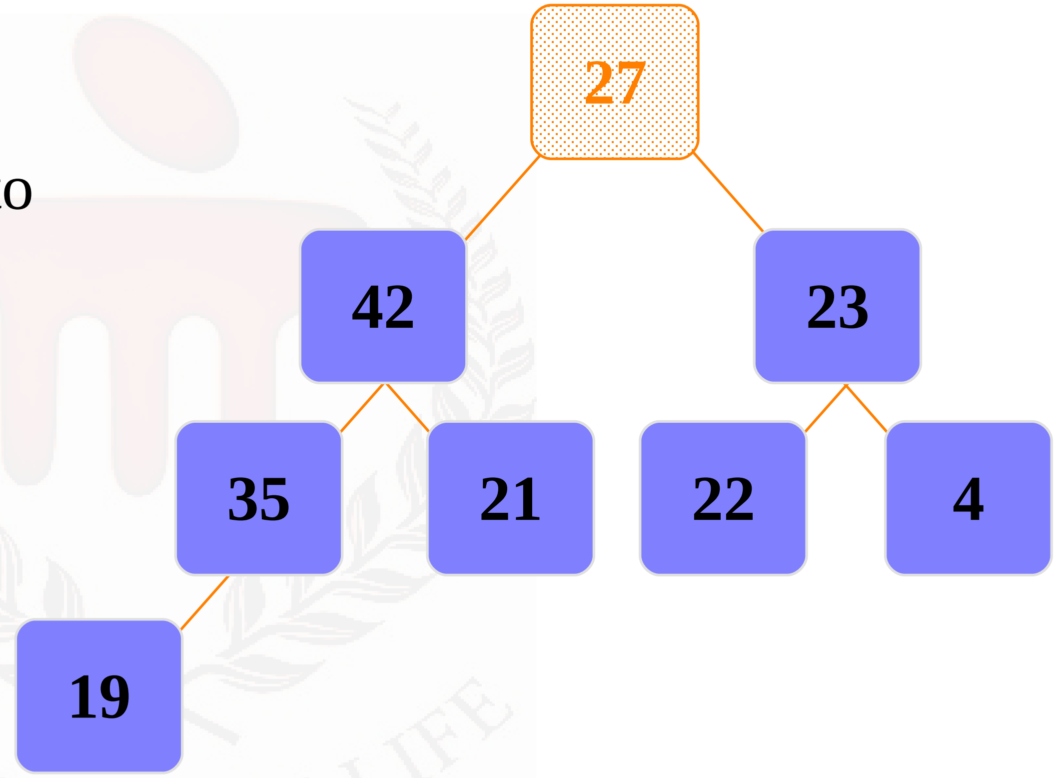
Removing the Top of a Heap

- Move the last node onto the root.



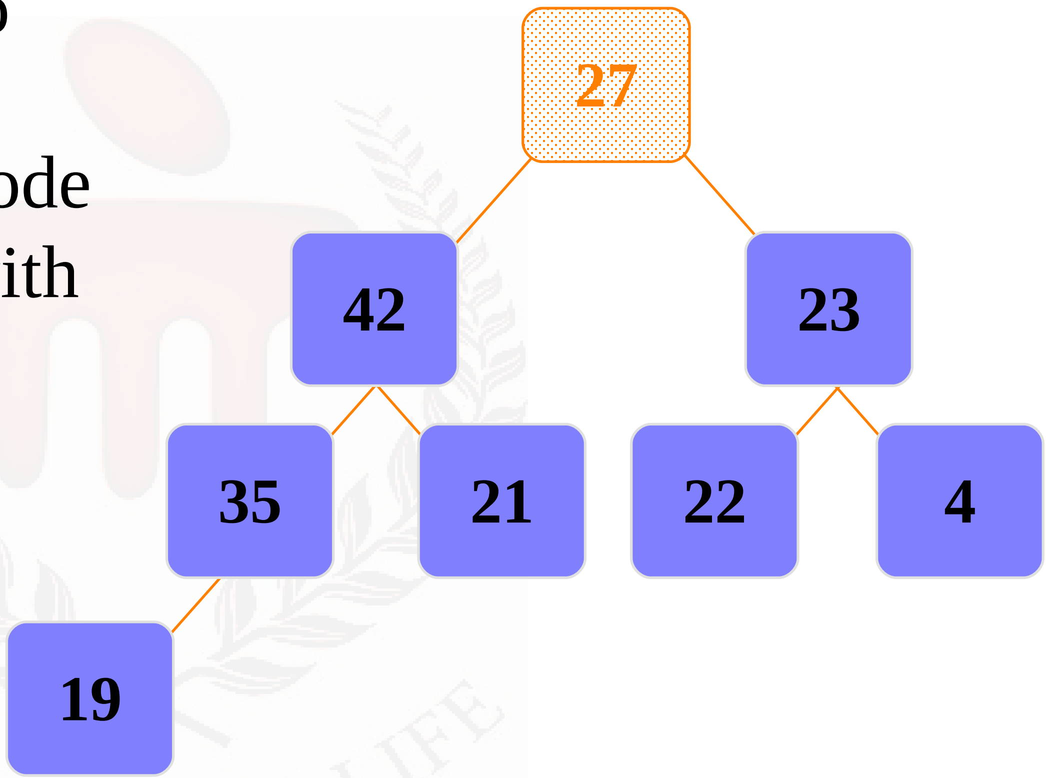
Removing the Top of a Heap

- Move the last node onto the root.



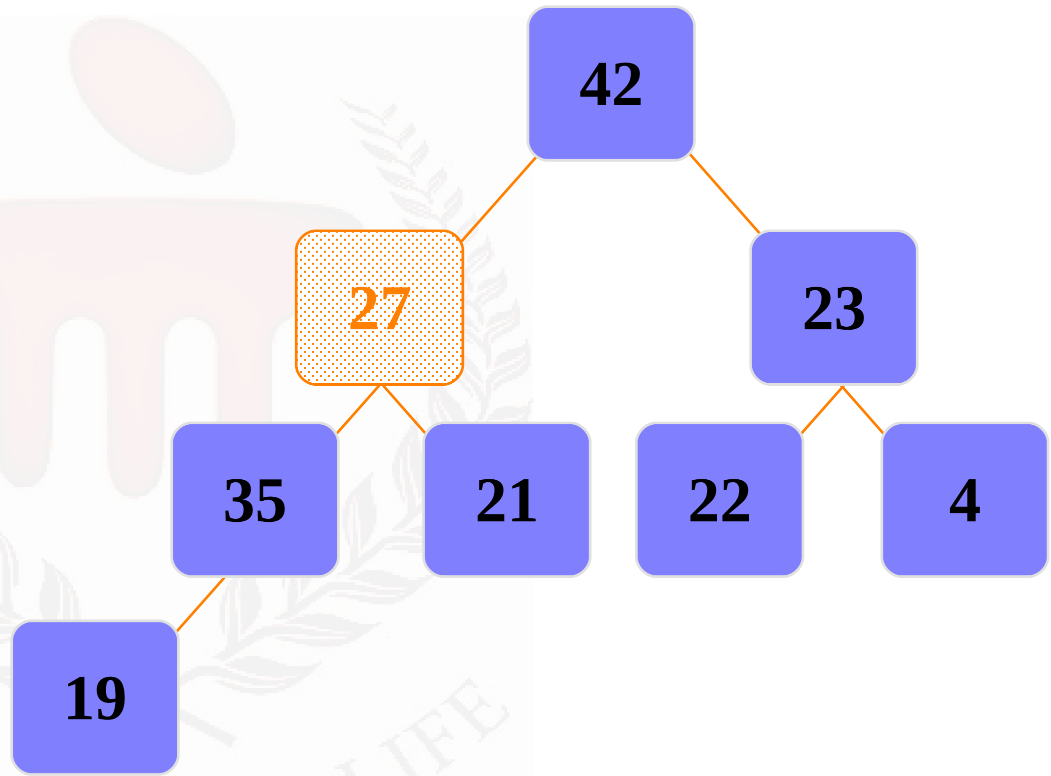
Removing the Top of a Heap

- 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.



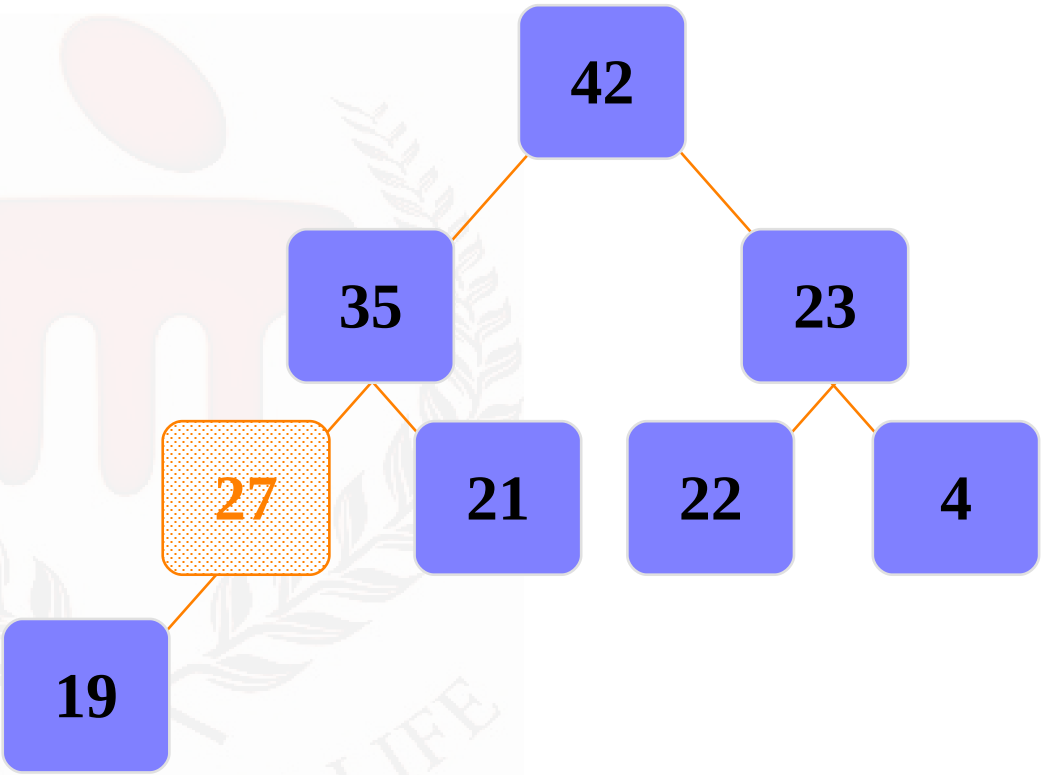
Removing the Top of a Heap

- 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.



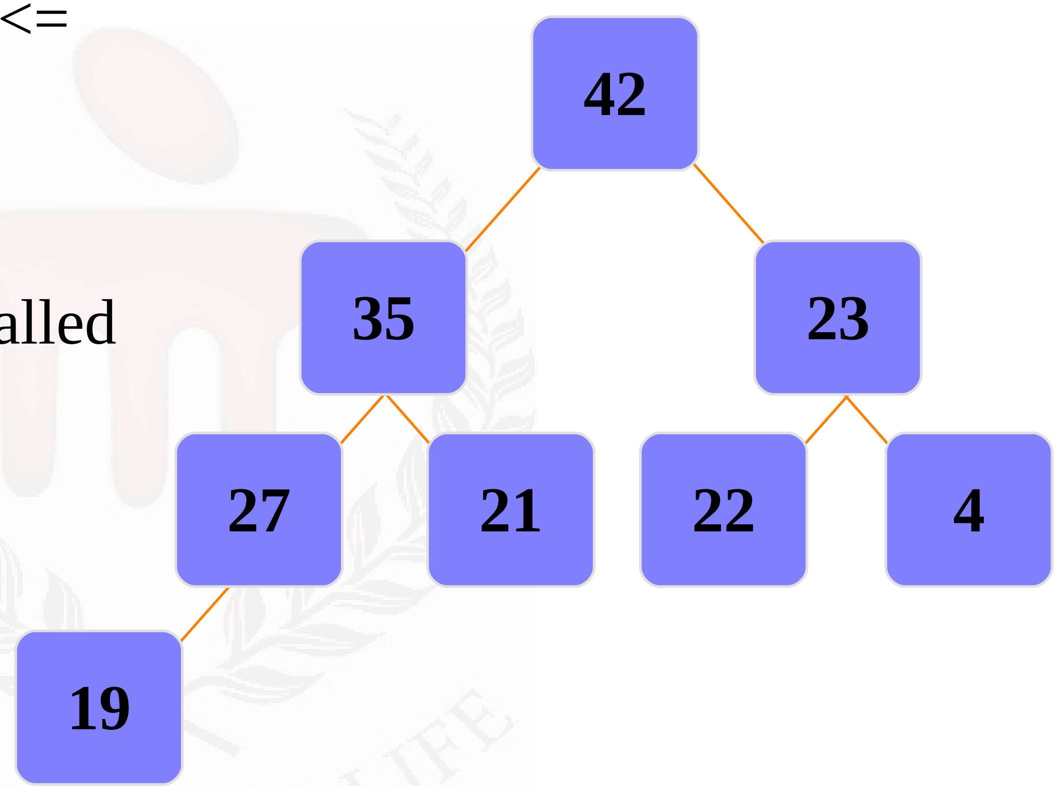
Removing the Top of a Heap

- 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.



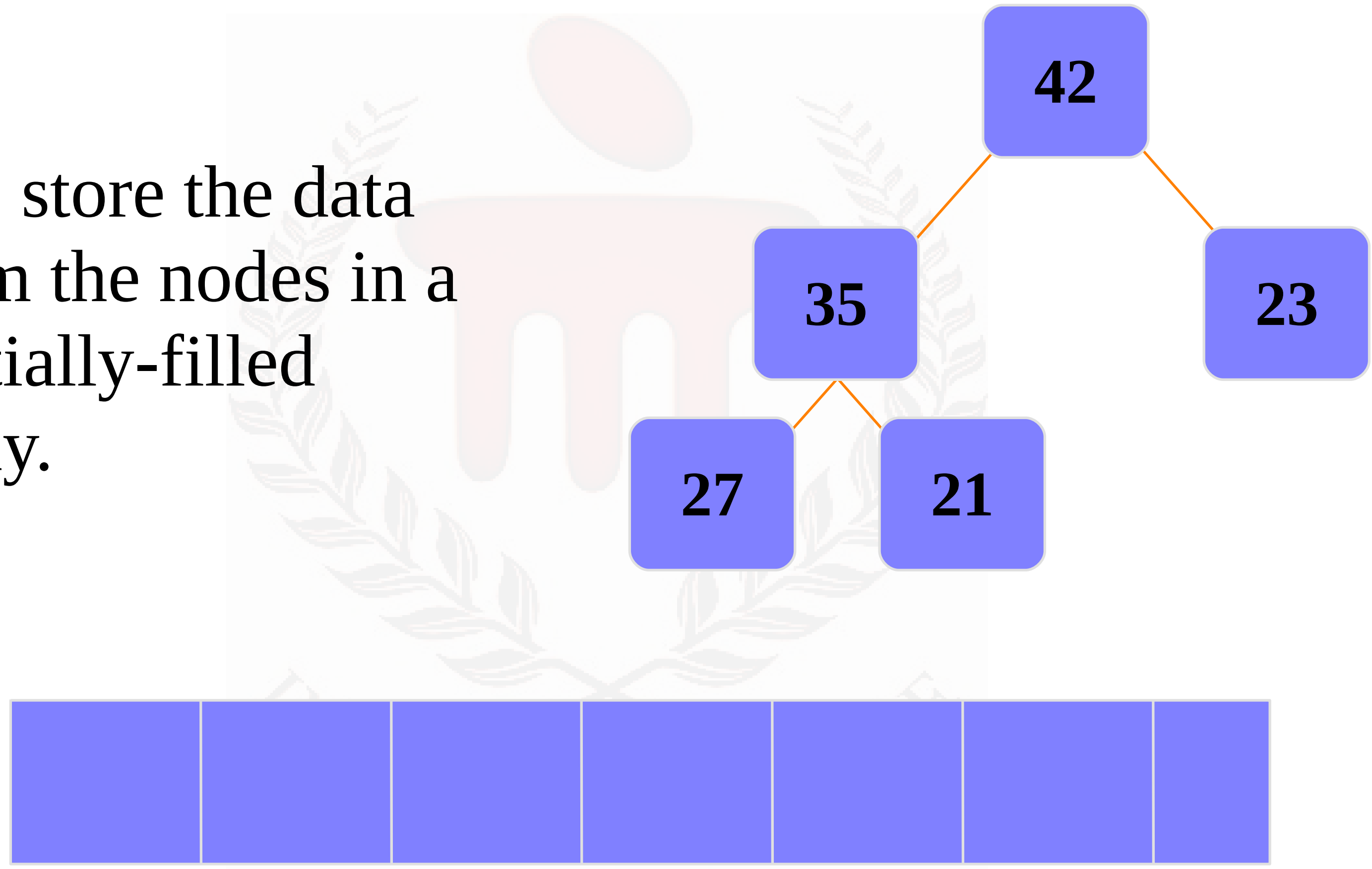
Removing the Top of a Heap

- 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**.



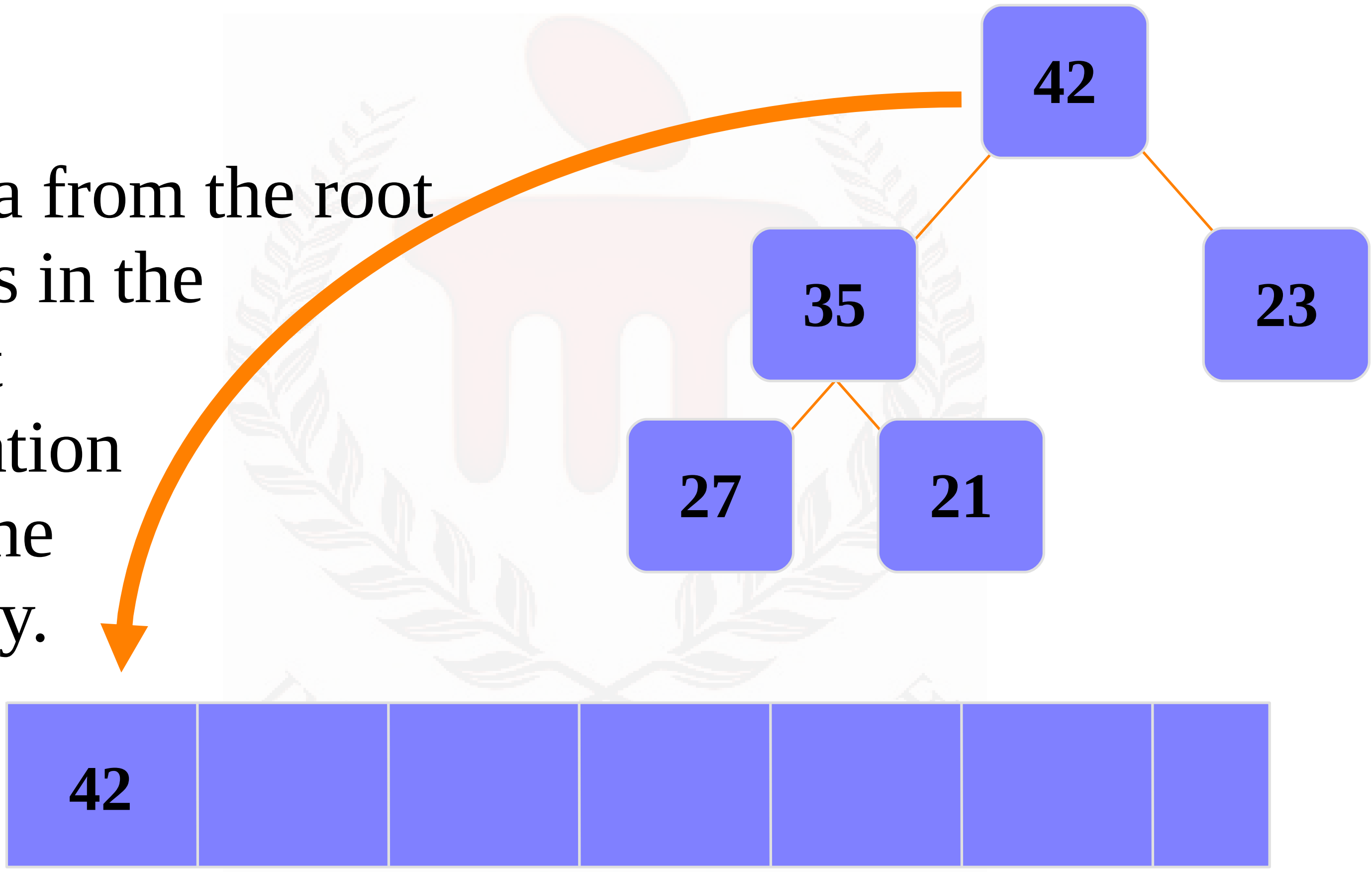
Implementing a Heap

- We store the data from the nodes in a partially-filled array.



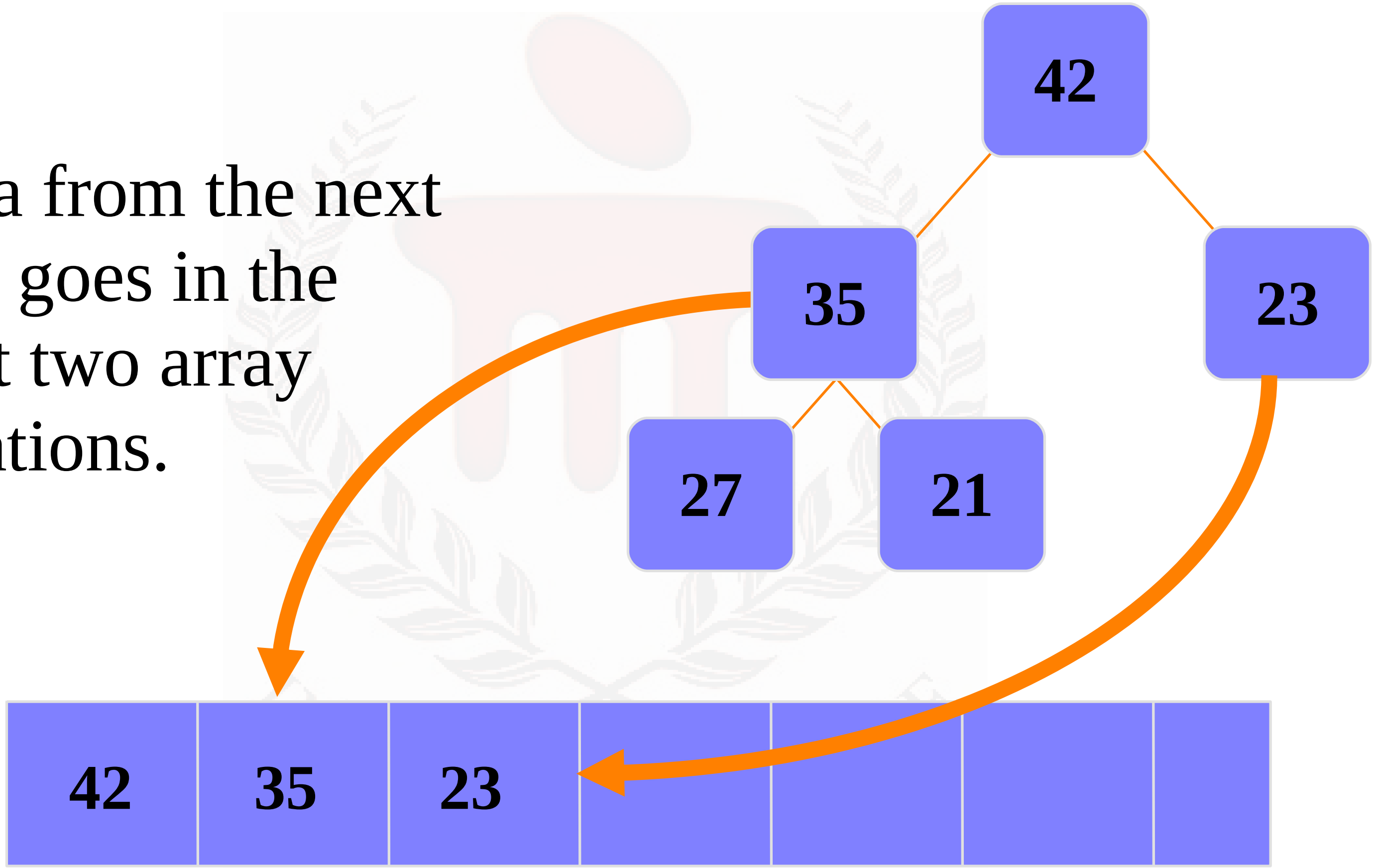
Implementing a Heap

- Data from the root goes in the first location of the array.



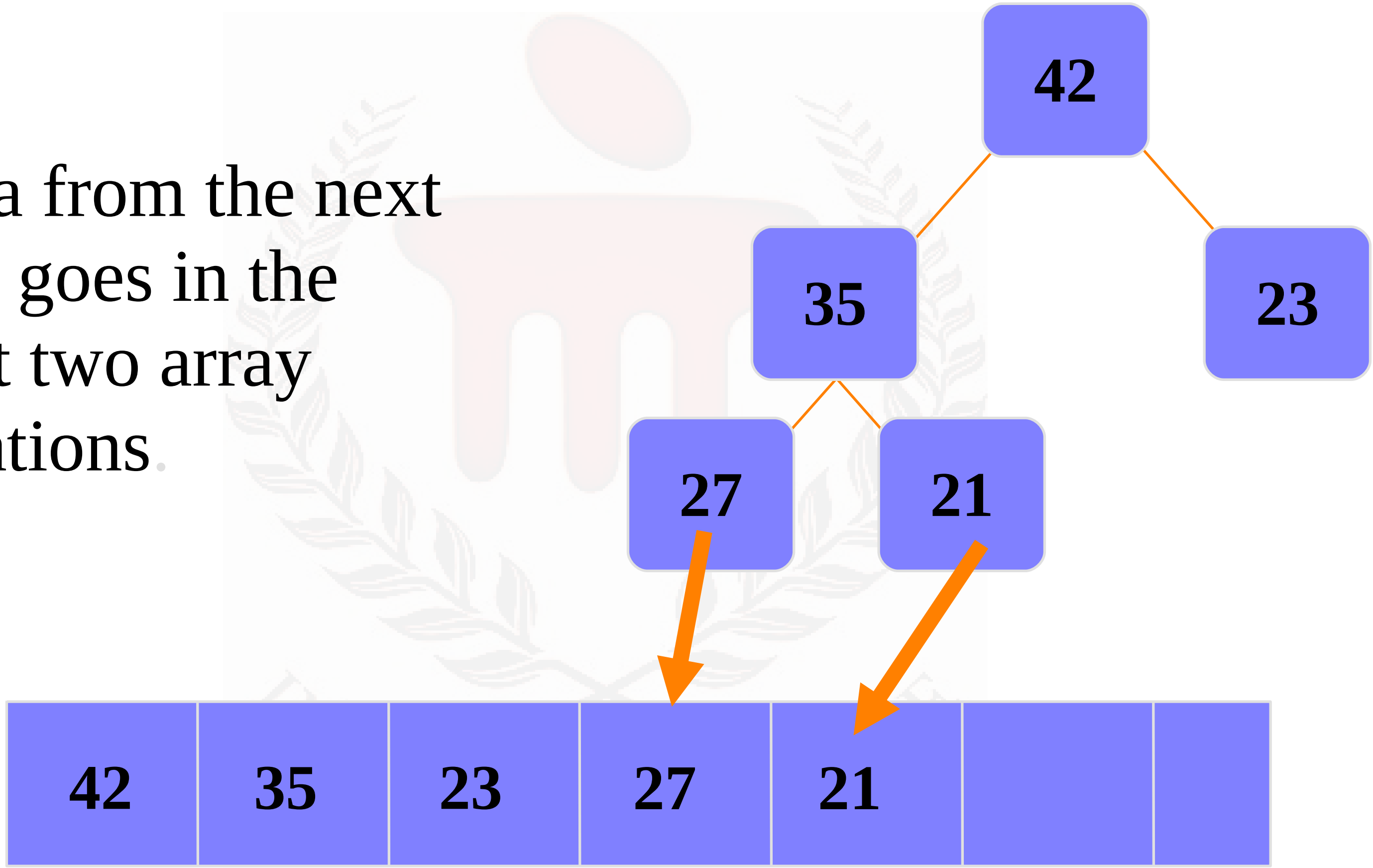
Implementing a Heap

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



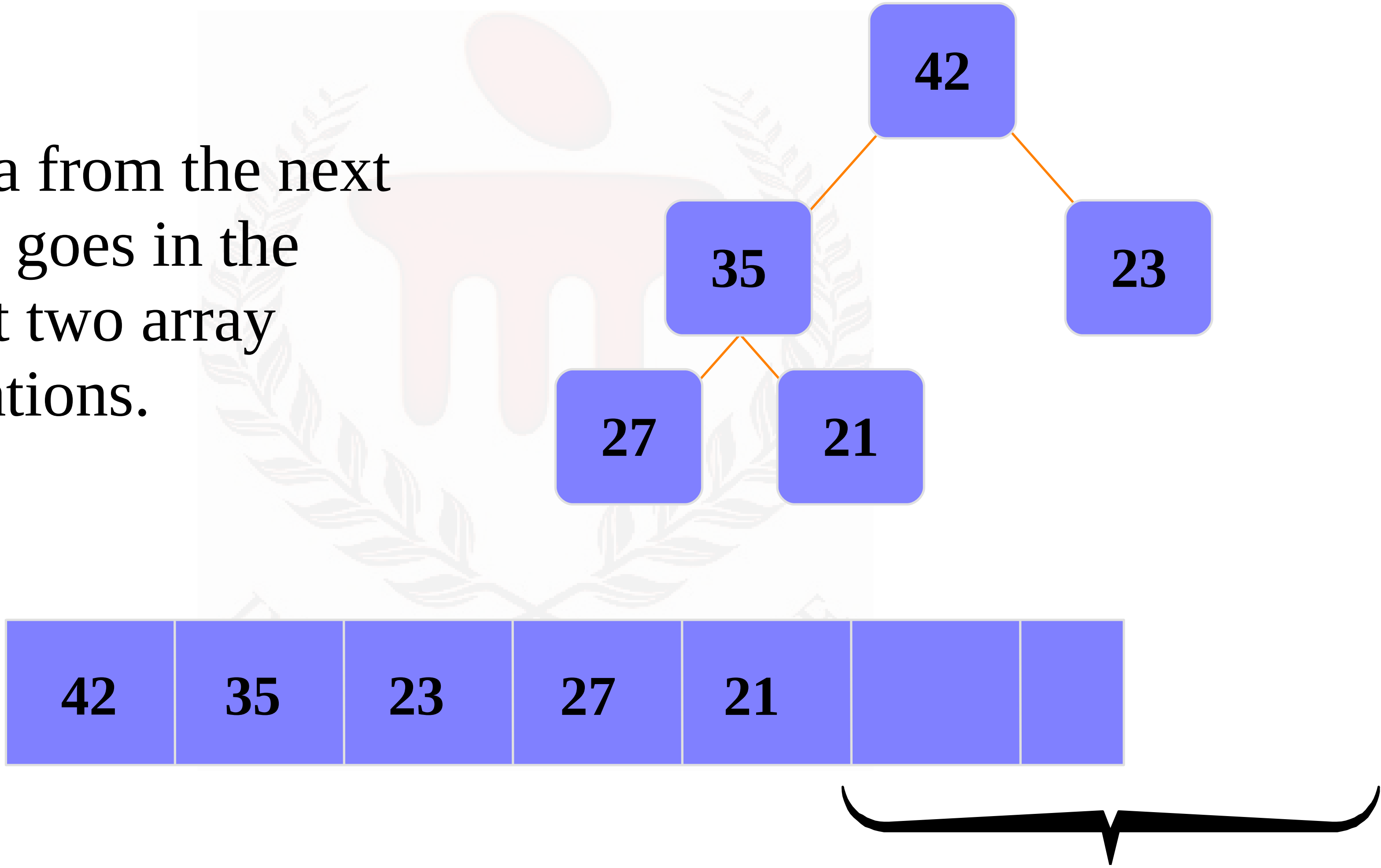
Implementing a Heap

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



Implementing a Heap

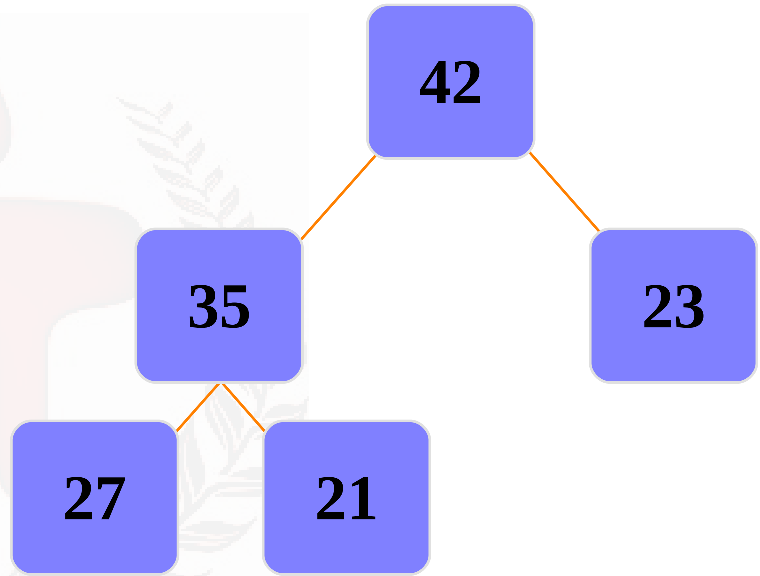
- Data from the next row goes in the next two array locations.



We don't care what's in this part of the array.

Important Points about the Implementation

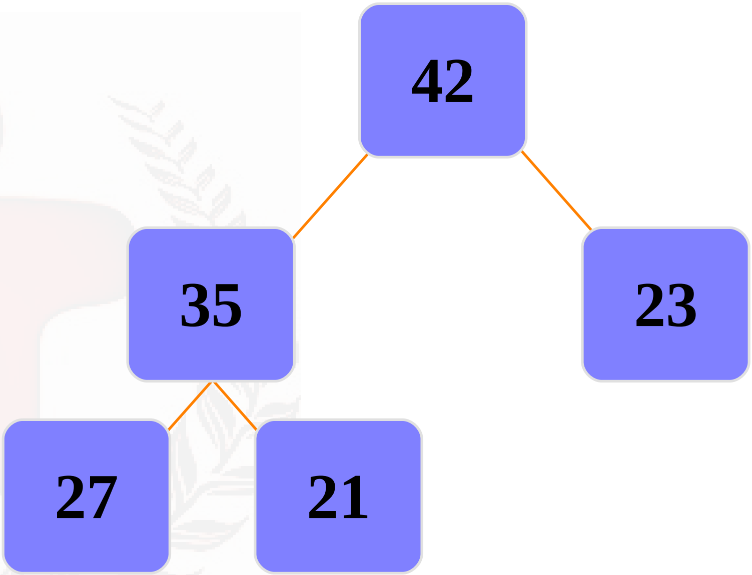
- The links between the tree's nodes are not actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.



42	35	23	27	21		

Important Points about the Implementation

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



42	35	23	27	21		

Heap code 1/3

```
#define MAX 25
int insertheap(int item, int a[], int n)
{   int c=n, p;
 if(c==MAX)
 {   printf("Heap is full\n“);   return n;   }
 c=n+1;  //c is child index
 p=c/2; // p is parent index
 while(c!=1 && item>a[p])//reheapification upward.
 {   a[c]=a[p];   c=p;  p=c/2;   }
 a[c]=item;
 return n+1;
}
```

Heap code 2/3

```
int delHeap(int a[], int n)
{  int c,p,temp;
 if(n==0)  {  printf("Heap is empty\n“);  return 0;  }
printf("Item deleted is: %d“,a[1]);
 temp=a[n--];   p=1;   c=2*p;
 while(c<=n) //reheapification downward
 {  if(a[c]<a[c+1])      c++;
  if(temp>=a[c])     break;
  a[p]=a[c];  p=c;  c=2*p;
 }
 a[p]=temp;
 return n;
}
```

Heap code 3/3

```
void print(int a[],int n)
{   for(int i=1;i<=n;i++) printf(" %d ",a[i]);  }
int main()
{  int a[20],ch=1,n=0,item;
 while(ch!=4)
 {  printf("\n1.Insert  2. Delete  3.Print  4. Exit \n");
    scanf("%d",&ch);
  switch(ch)
  {   case 1: printf("Enter the element of the heap\n");
             scanf("%d",&item);
               n=insertheap(item,a,n);          break;
   case 2: n=delHeap(a,n);                       break;
   case 3: print(a,n);                           break;
  }     }
 return 0;
}
```

Summary

- A heap is a complete binary tree, where the entry at each node is greater than/less than or equal to the entries in its children.
- To add an entry to a heap, place the new entry at the next available spot, and perform a reheapification upward.
- To remove the biggest entry, move the last node onto the root, and perform a reheapification downward.