Heaps

Heaps Definition

- A ***heap*** is a certain kind of ***complete*** binary tree.

Recall

Complete Binary Tree

•Every level but lowest has all possible nodes
•If lowest level is not full, all nodes as far left as possible

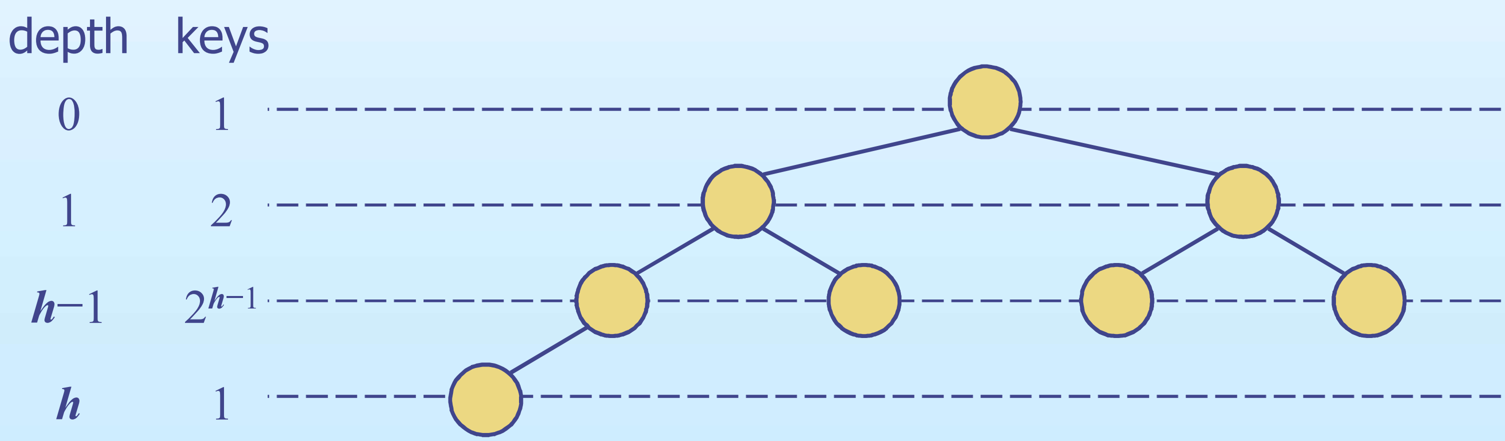
- It must satisfy the following property:
  - The entry contained by the node is ***NEVER*** less than the entries of the node's children

  // this is called the heap property

Height of a Heap

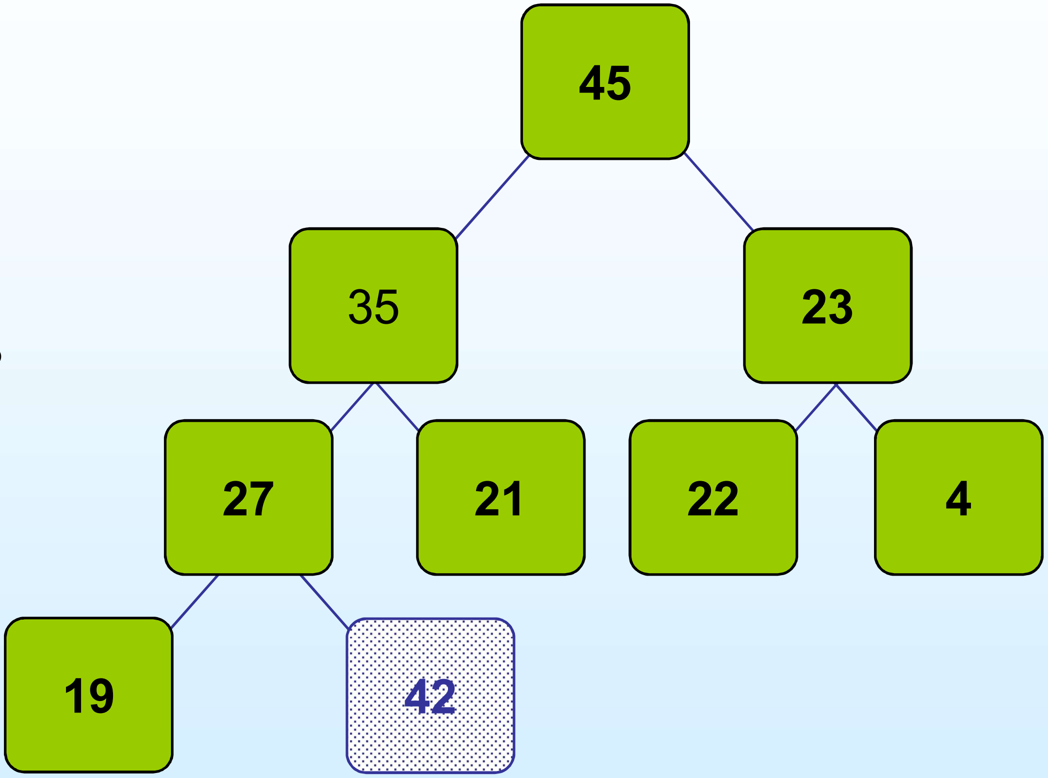
- Theorem: A heap storing n keys has height $h \leq \log n$

  Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h - 1$ and at least one key at depth h, we have $n <= 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
  - Thus, $n <= 2^h$, i.e., $h <= \log n$



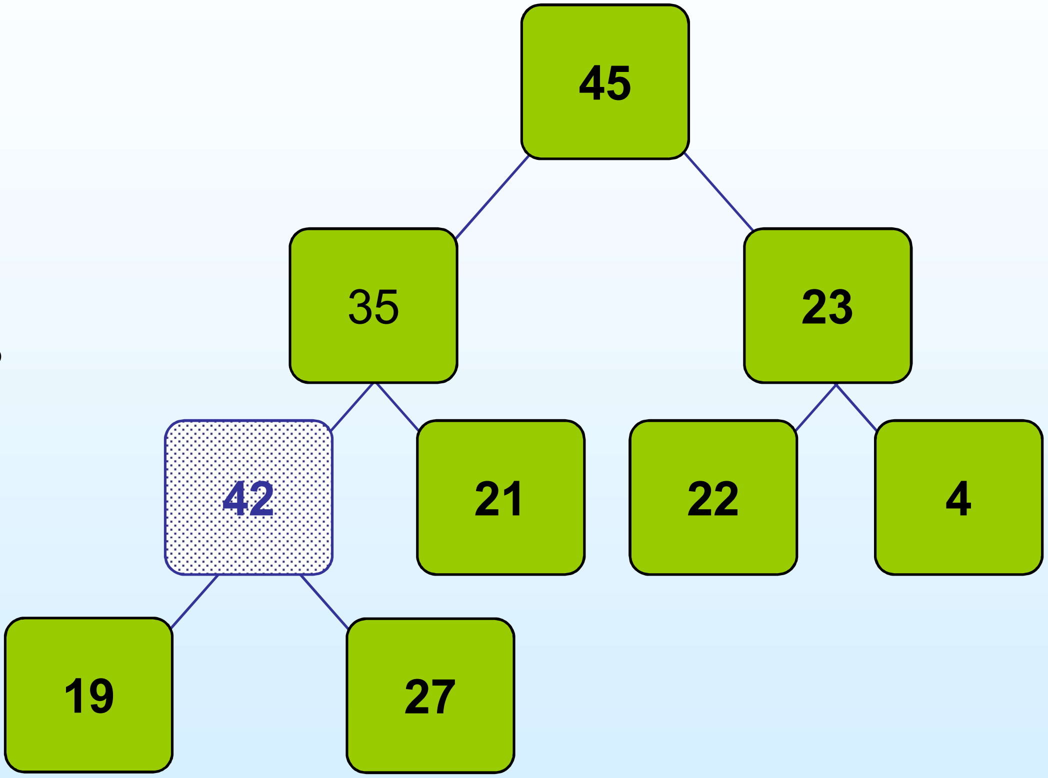
Adding a Node to a Heap

- Put the new node in the next available spot. (note: a heap is a complete tree)
- <u>Restore the heap property:</u> Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



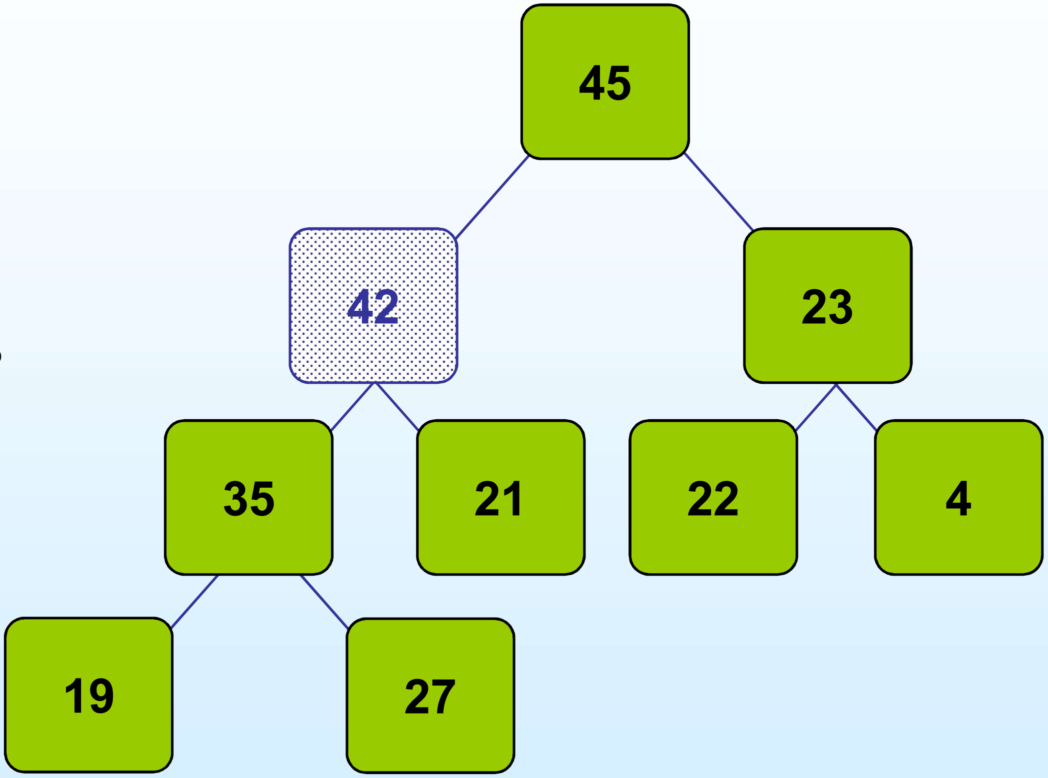
Adding a Node to a Heap

- Put the new node in the next available spot. (note: a heap is a complete tree)
- Restore the heap property: Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



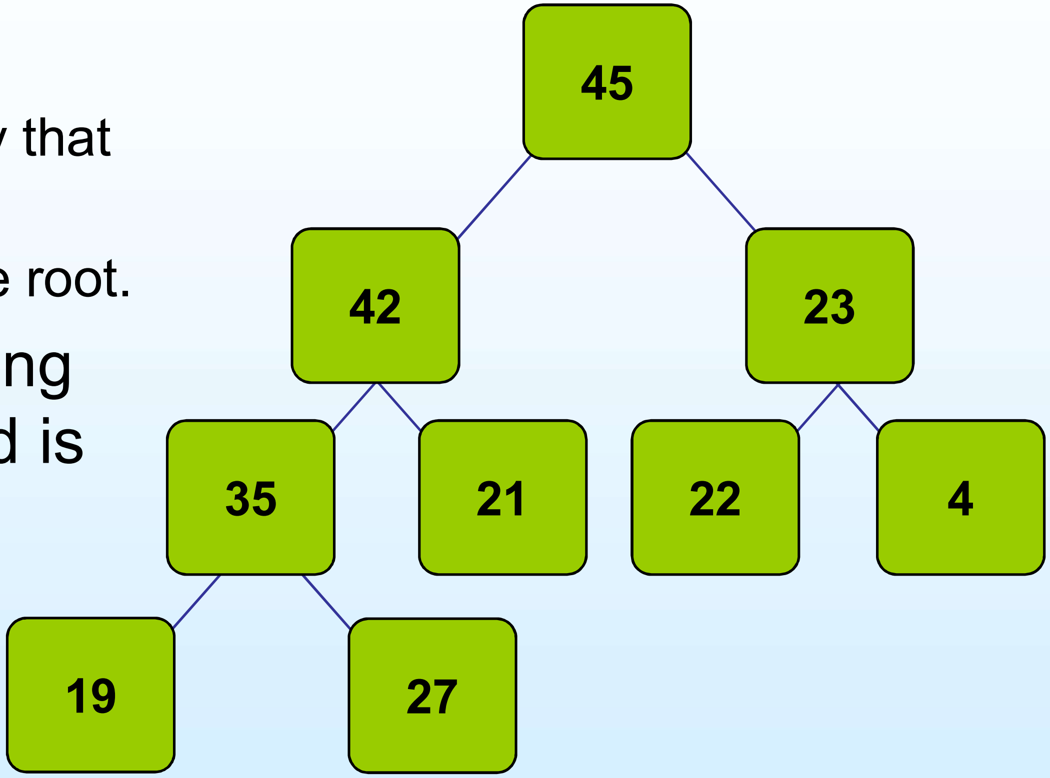
Adding a Node to a Heap

- Put the new node in the next available spot. (note: a heap is a complete tree)
- <u>Restore the heap property:</u> Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



Adding a Node to a Heap

- Stop when:
  - 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**.

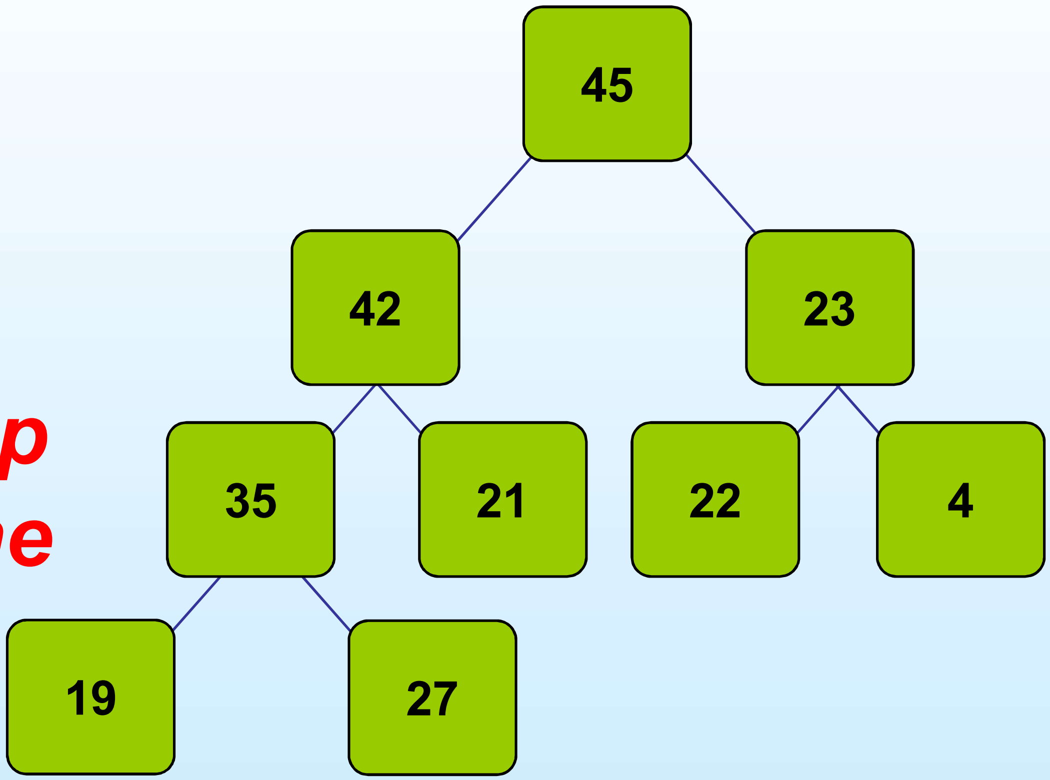


Note: we need to easily go from child to parent as well as parent to child.

Removing a Node from a Heap

- Which node to remove?

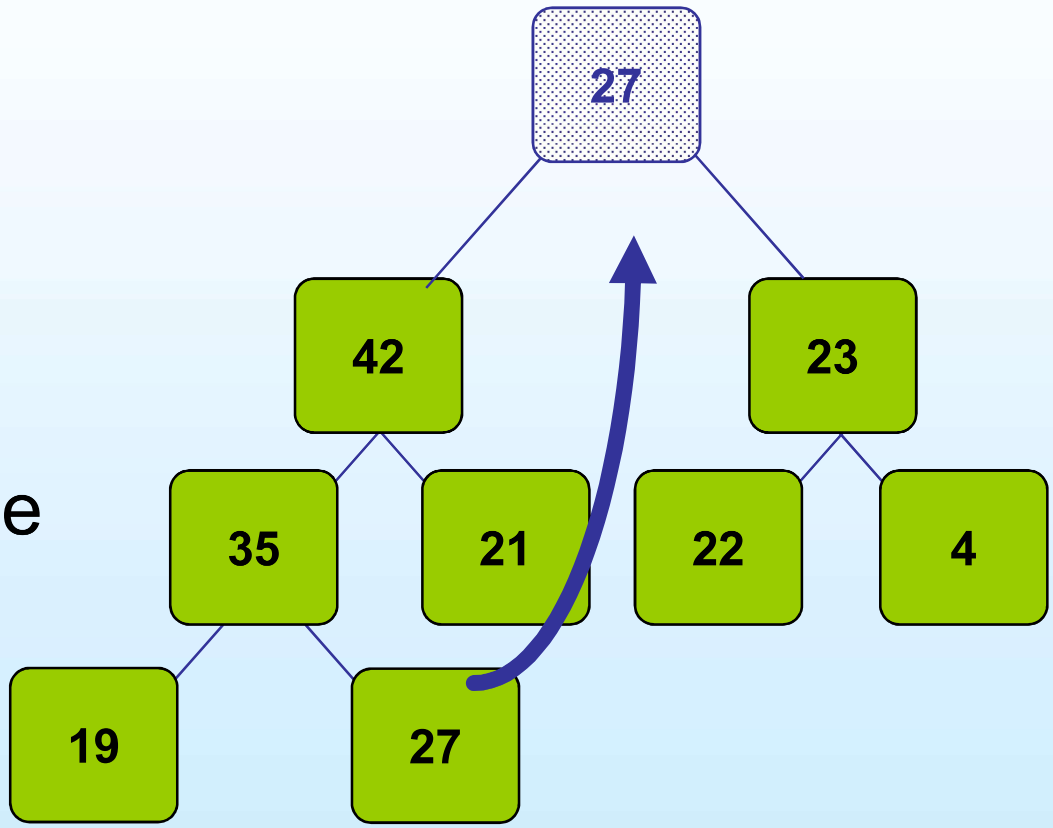
Remove the top of the Heap (the root node) !!!



Removing a Node from a Heap

- Reconstruct the *complete binary tree* property:

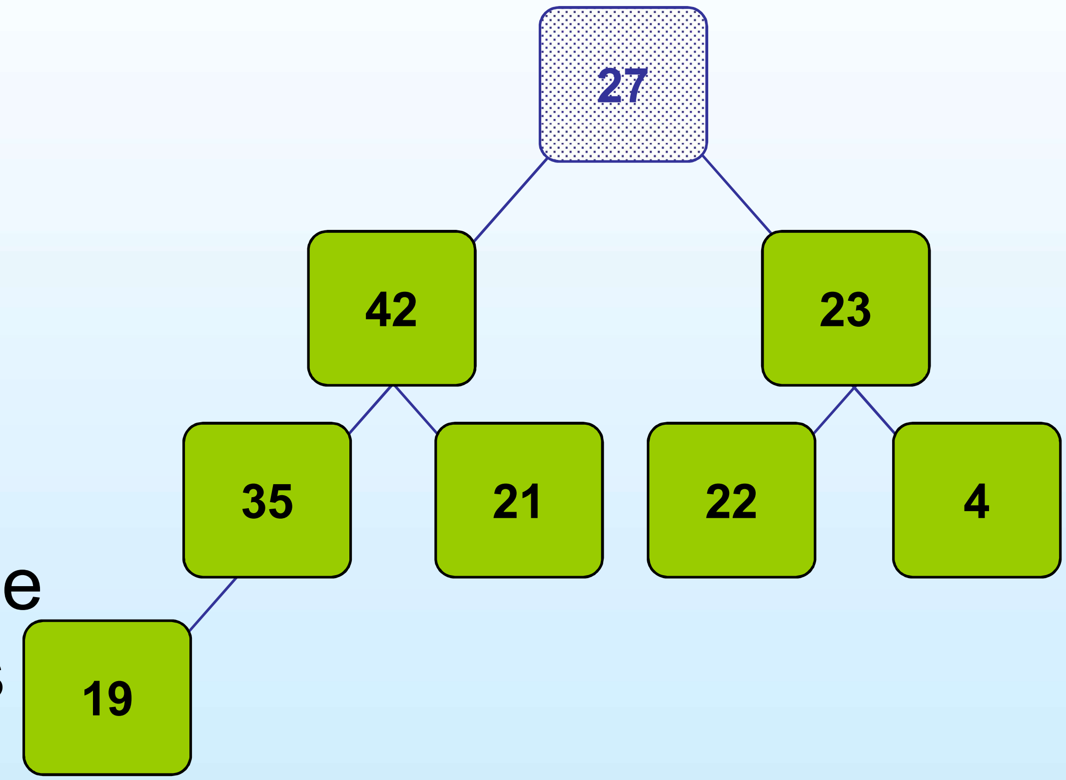
  Move the last node onto the root.



Removing a Node from a Heap

- Reconstruct the Heap property:

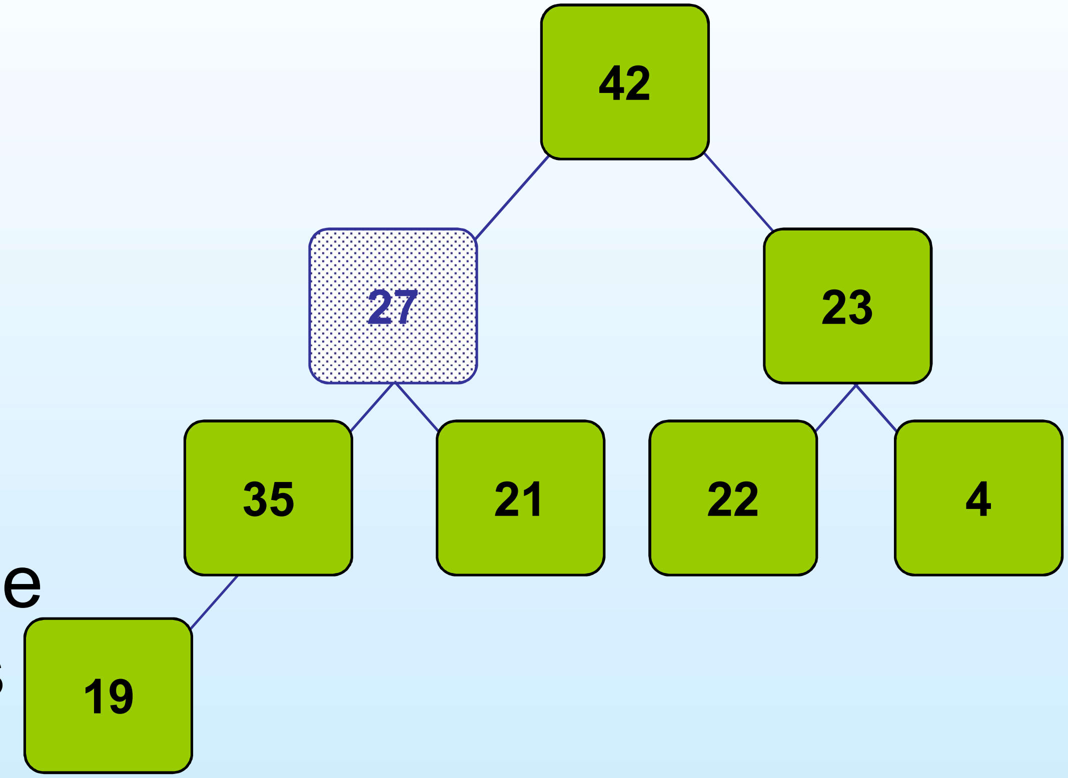
  Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



Removing a Node from a Heap

- Reconstruct the Heap property:

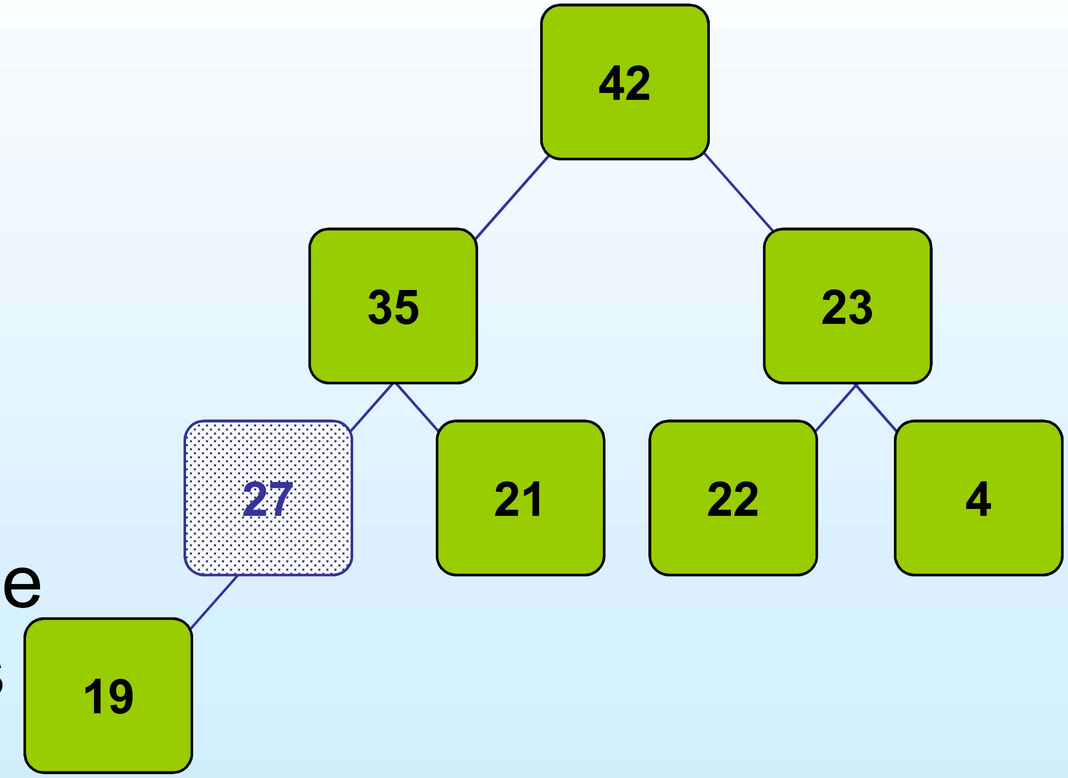
  Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



Removing a Node from a Heap

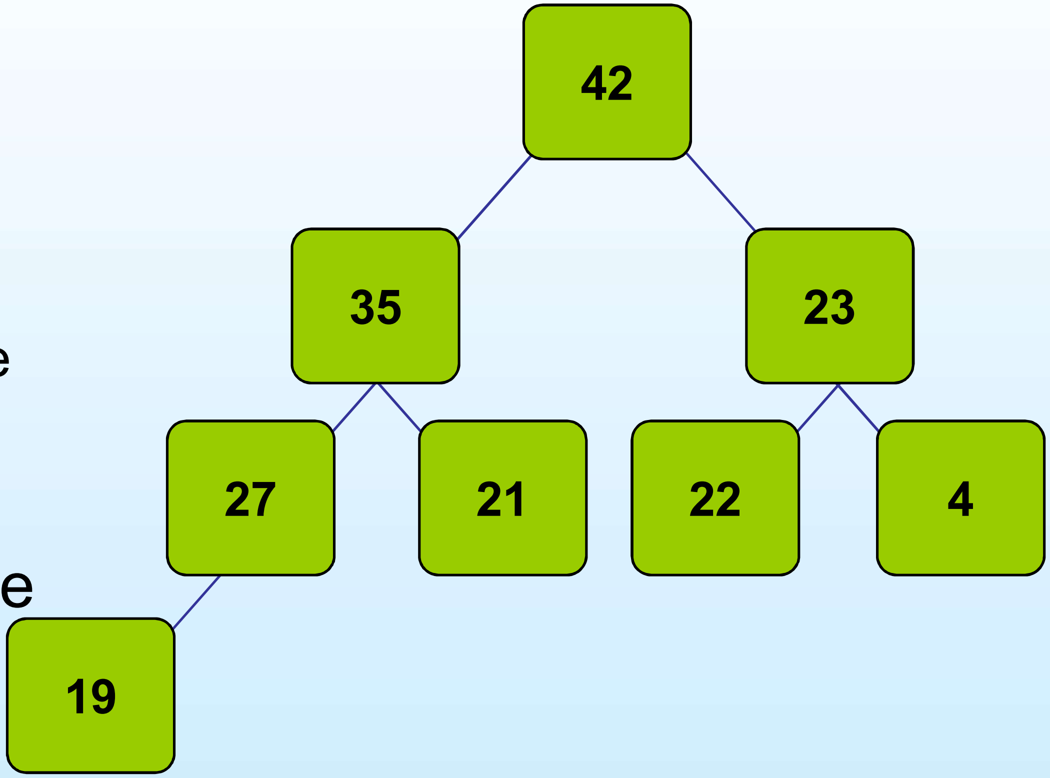
- Reconstruct the Heap property:

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



Removing a Node from a Heap

- Stop when:
  - All the children 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**.



Heap Implementation

- Use linked node
  - node implementation is for a general binary tree
  - but we may need to have doubly linked node
- Use arrays
  - A heap is a complete binary tree
  - which can be implemented more easily with an array than with linked nodes
  - and do two-way links

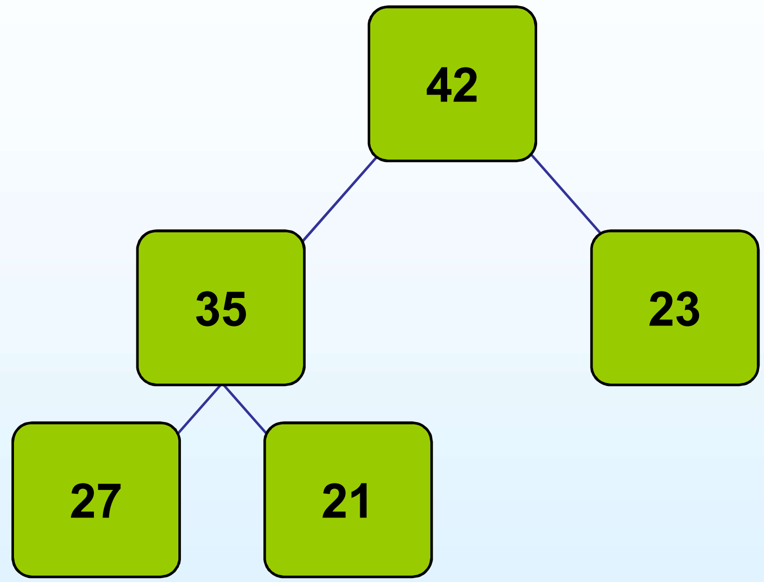
Array Representation of a Heap

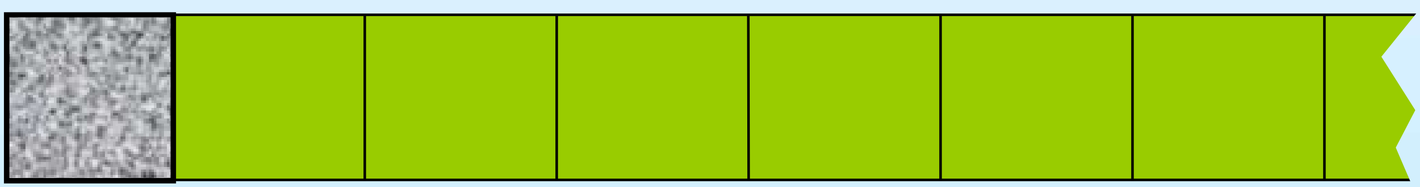
Recall

- For the node with index i
  - the left child is at index 2i
  - the right child is at index 2i + 1
- Links between nodes are not explicitly stored
- The cell at index 0 is not used

Implementing a Heap

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

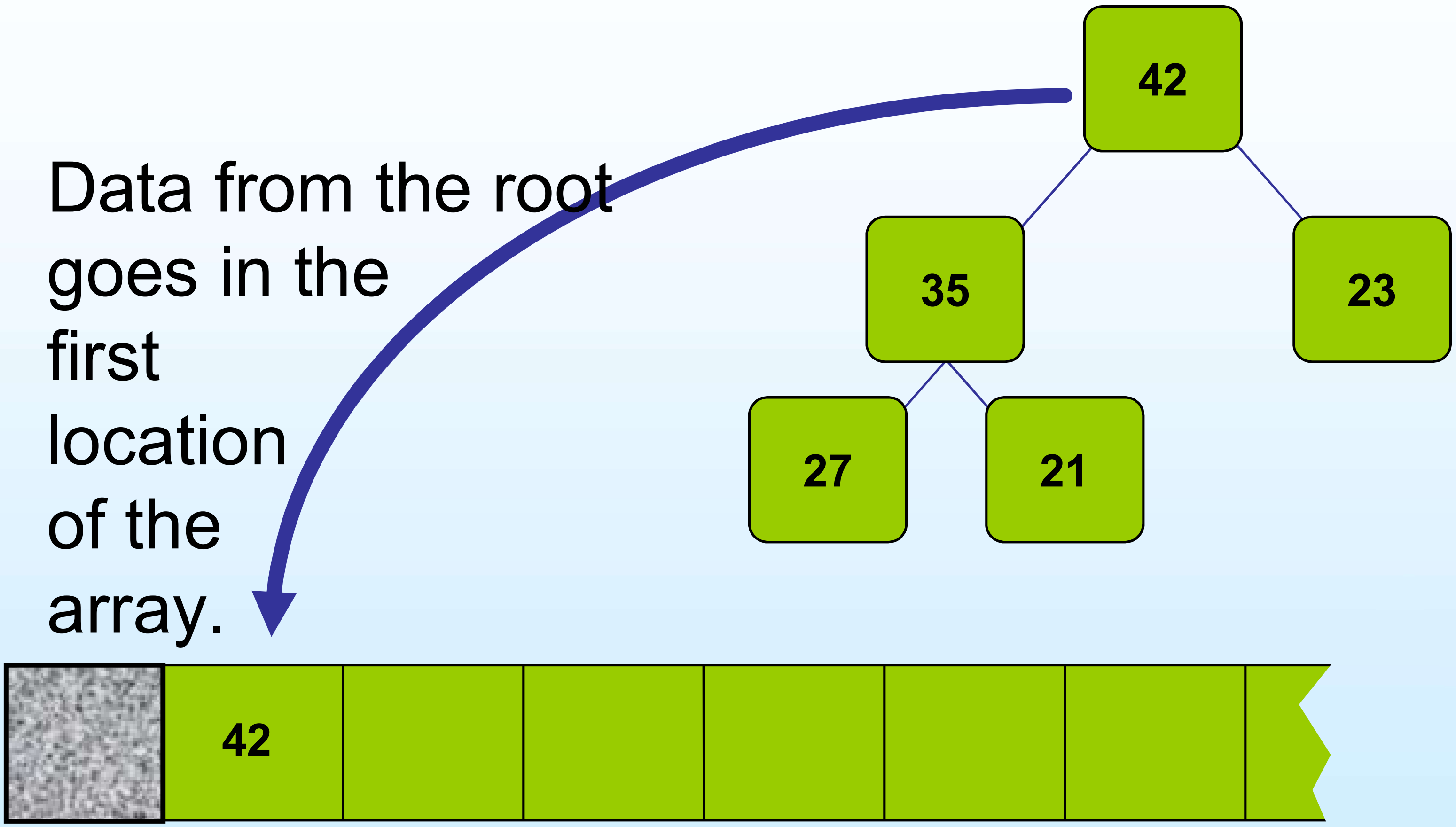




An array of data

Implementing a Heap

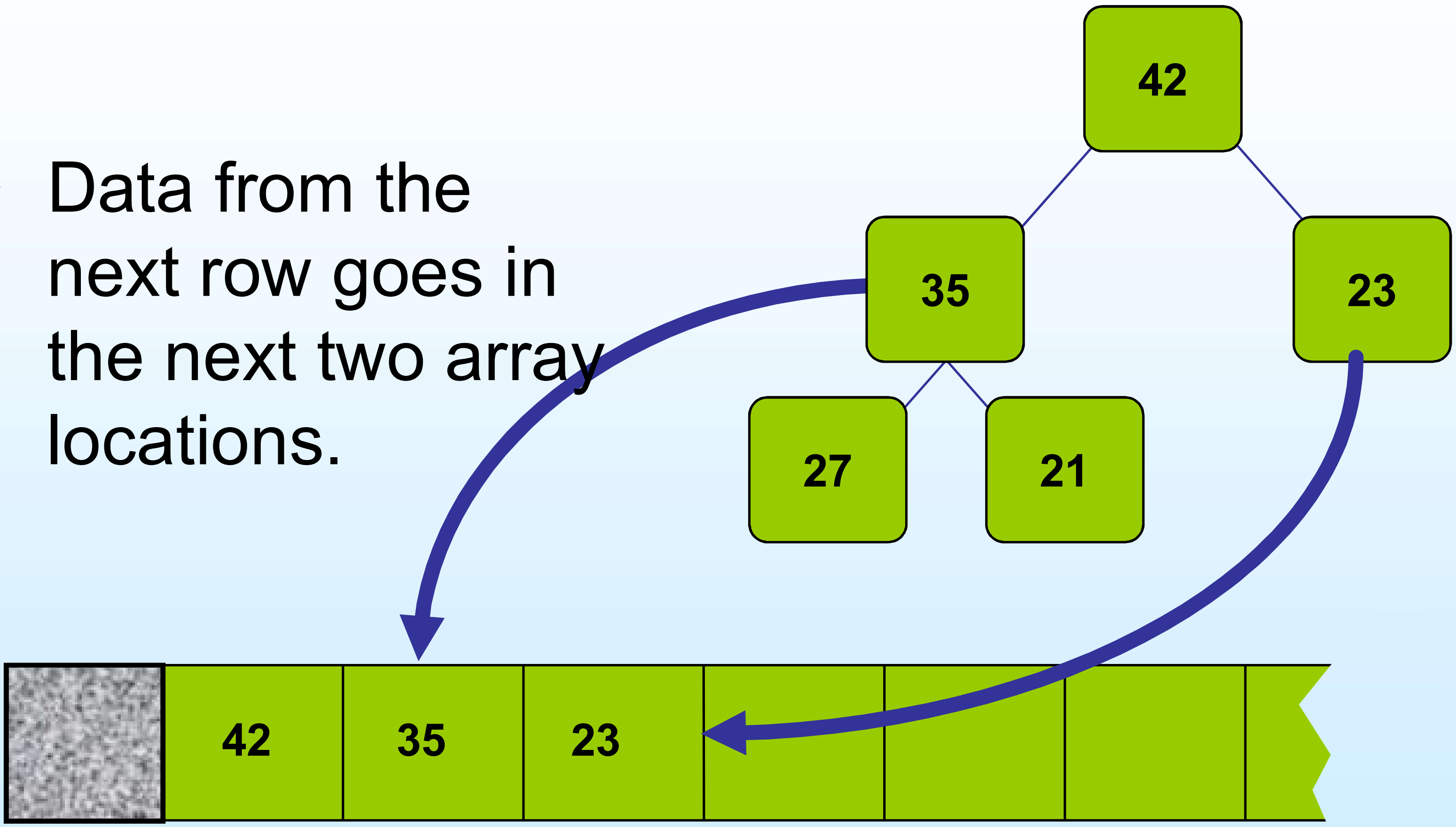
- Data from the root goes in the first location of the array.



An array of data

Implementing a Heap

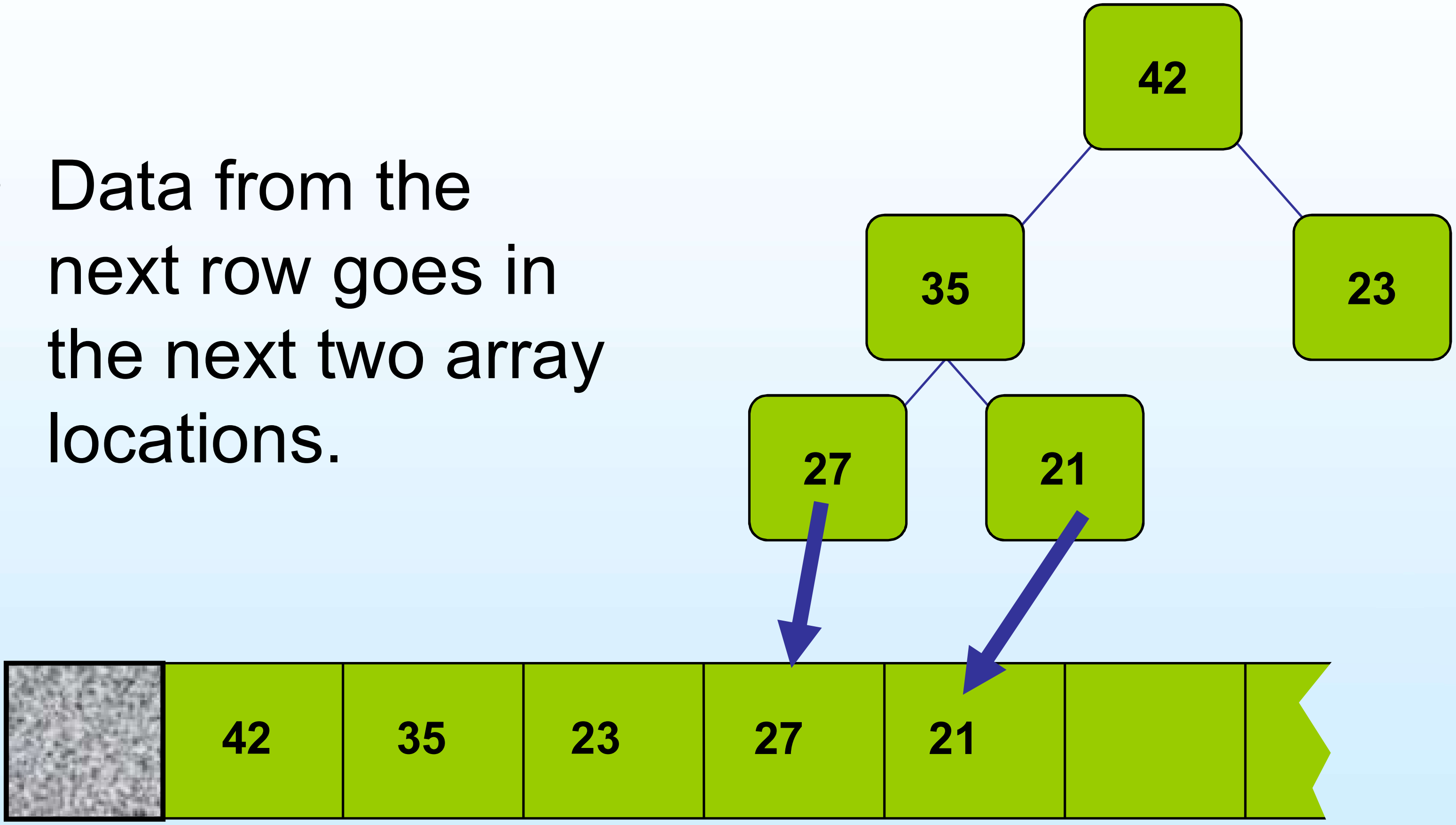
- Data from the next row goes in the next two array locations.



An array of data

Implementing a Heap

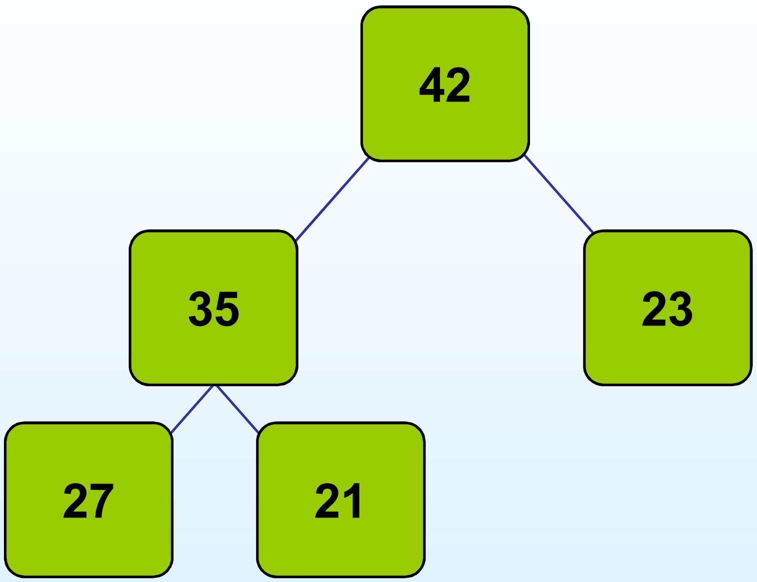
- Data from the next row goes in the next two array locations.

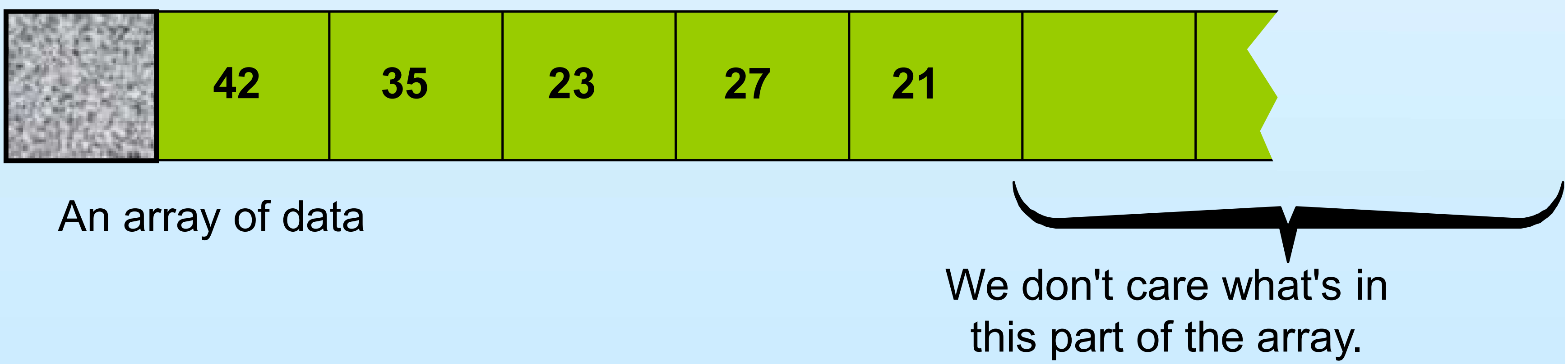


An array of data

Implementing a Heap

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





implementation in an ADT: array and size

```
Comparable[] a;
int size;

private boolean full()
{  // first element of array is empty
 return (size == a.length-1);
}
```

implementation in an ADT: insertion

```
public void add(Comparable data)
{
 if (full())  // expand array
    ensureCapacity(2*size);
 size++;
 a[size] = data;
 if (size > 1)
    heapifyUp();
}
```

implementation in an ADT: heapifyUp

```
private void heapifyUp()
{
 Comparable temp;
 int next = size;
 while (next != 1 &&
        a[next].compareTo(a[next/2]) > 0)
  {
   temp = a[next];
   a[next] = a[next/2];
   a[next/2] = temp;
   next = next/2;
  }
}
```

implementation in an ADT: deletion

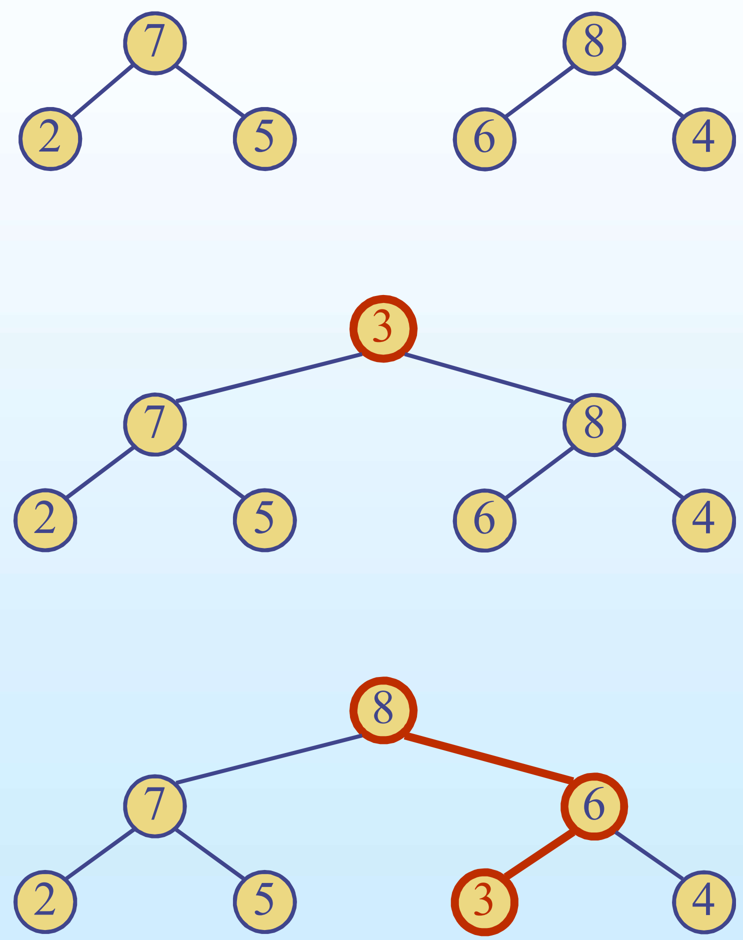
```
public Comparable removeMax()
{
 if (size == 0)
   throw new IllegalStateException("empty heap");
 Comparable max = a[1];
 a[1] = a[size];
 size--;
 if (size > 1)
    heapifyDown(1);
 return max;
}
```

implementation in an ADT: heapifyDown

```
private void heapifyDown(int root)
{
 Comparable temp;
 int next = root;
 while (next*2 <= size) // node has a child
 {
  int child = 2*next;  // left child
  if (child < size &&
      a[child].compareTo(a[child+1]) < 0)//left smaller than right
    child++;  // right child instead
  if (a[next].compareTo(a[child]) < 0)
  {
   temp = a[next];
   a[next] = a[child];
   a[child] = temp;
   next = child;
  }
  else;
   next = size;  // stop loop
 }//end while
}
```

Merging Two Heaps

- We are given two heaps and a key *k*
- We create a new heap with the root node storing *k* and with the two heaps as subtrees
- We perform heapifyDown to restore the heap-order property



Example

- **We have 15 element with keys:**
  24, 25, 36, 28, 34, 33, 17, 10, 15, 35, 29, 13, 33, 32, 30

- **First we construct (n+1)/2 element as**

➢ ***Add one more key for each pairs and do the merge process:***

Example

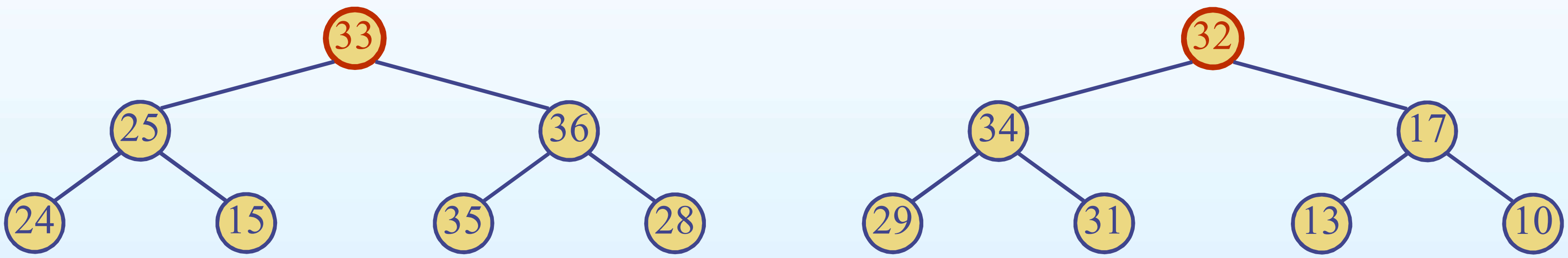
24, 25, 36, 28, 34, 33, 17, 10, 15, 35, 29, 13, 33, 32, 30

- **heapifyDown process:**

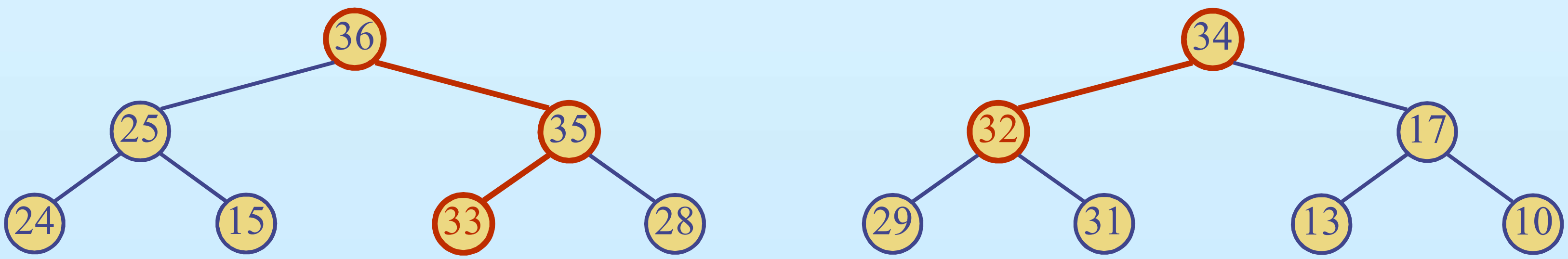
Example

24, 25, 36, 28, 34, 33, 17, 10, 15, 35, 29, 13, 33, 32, 30

- Add one more key for each two subtrees and do the merge process:



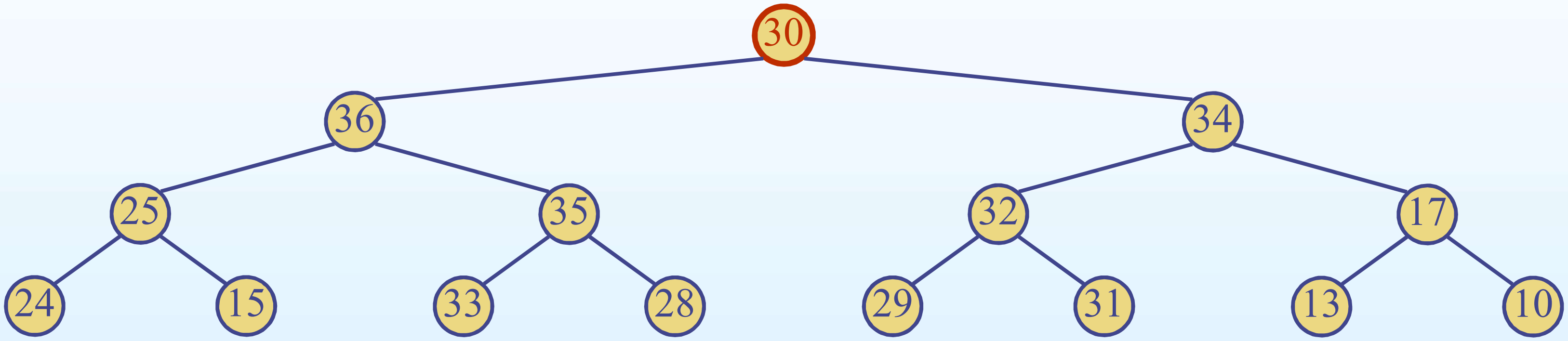
- heapifyDown process:



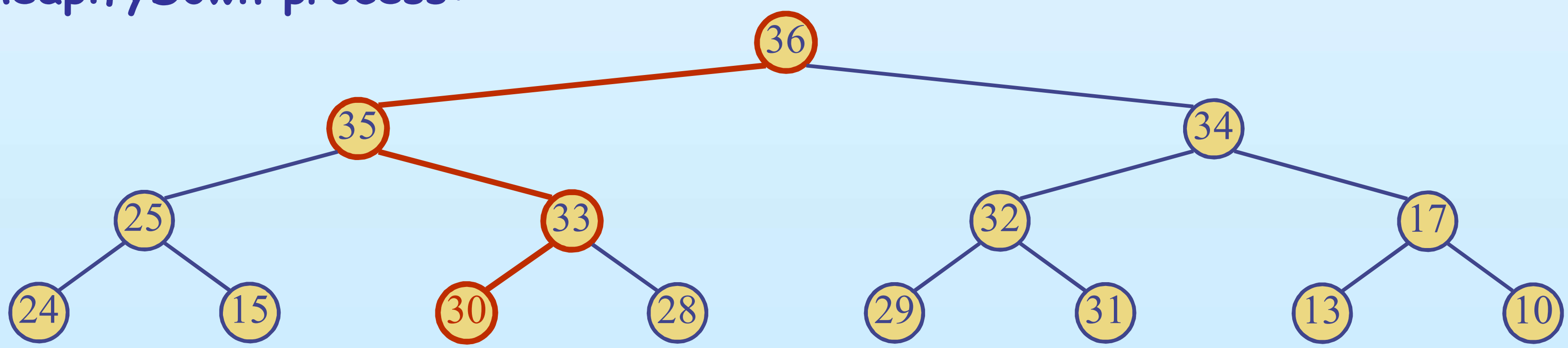
Example

24, 25, 36, 28, 34, 33, 17, 10, 15, 35, 29, 13, 33, 32, 30

- **Merge process:**



- **heapifyDown process:**



Bottom-up Heap Construction

- We can construct a heap storing n given keys in using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys