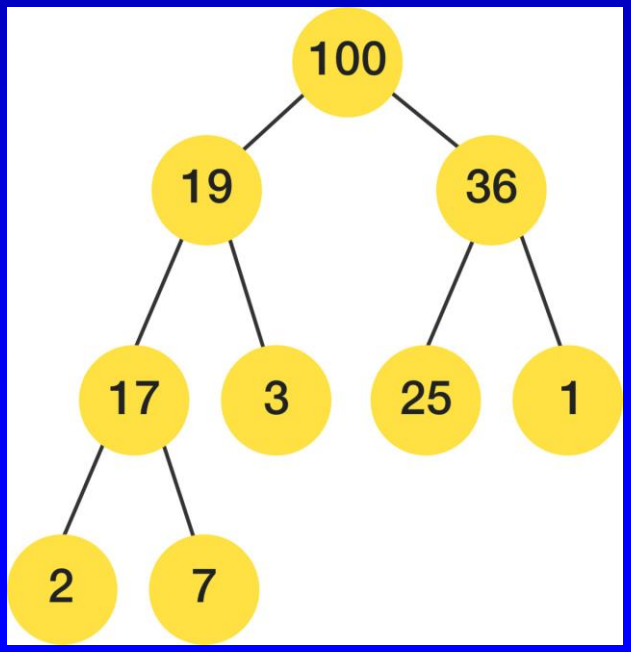
Heaps and Priority Queue

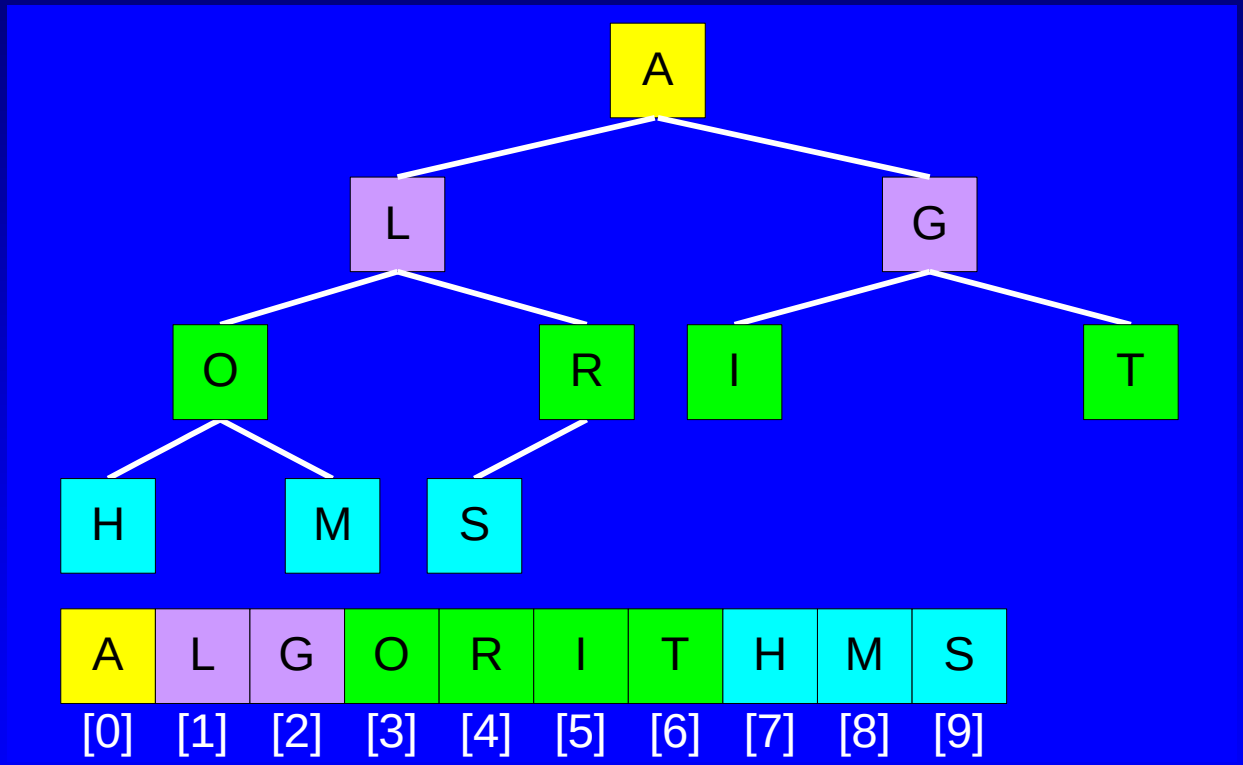
- **A complete binary tree** is a **binary tree** in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



Tree Representations

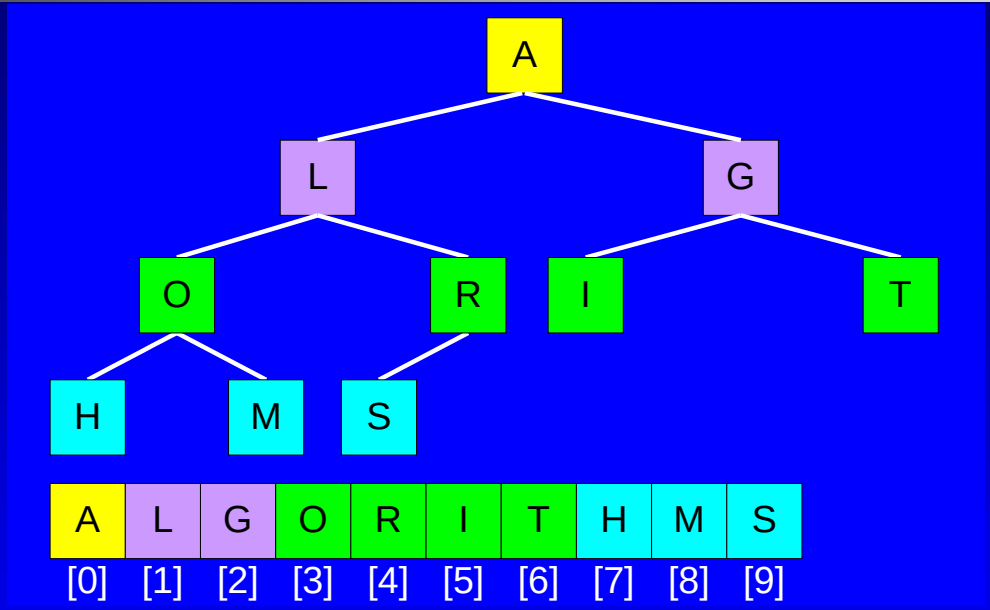
- Complete binary trees can be represented by
  - Array representation
  - Node representation

Array Representation



- Data from root is stored at [0]-th entry of array
- Each next level is stored at the next entries

Array Representation



- The node stored at [i]-th entry has children stored at entries
  - 2i + 1 (left child)
  - 2i +2 (right child)
- Non-root node stored at [i]-th entry has a parent at entry
  - $\lfloor (i - 1)/2 \rfloor$

Array Representation

- Class will have three data members
  - The array
  - Size of the array (how many entries of the array is used)
  - Capacity of the array

Heaps

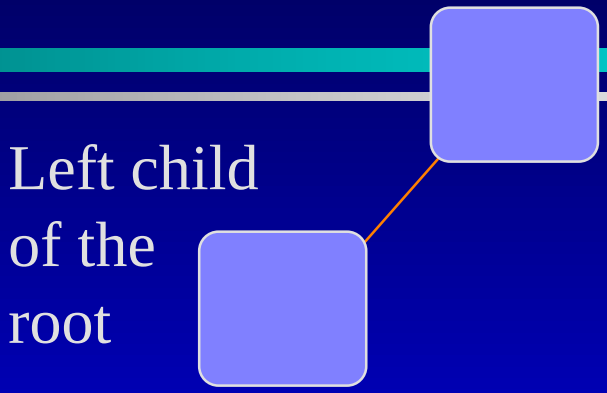
A **heap** is a certain kind of a complete binary tree

Heaps

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

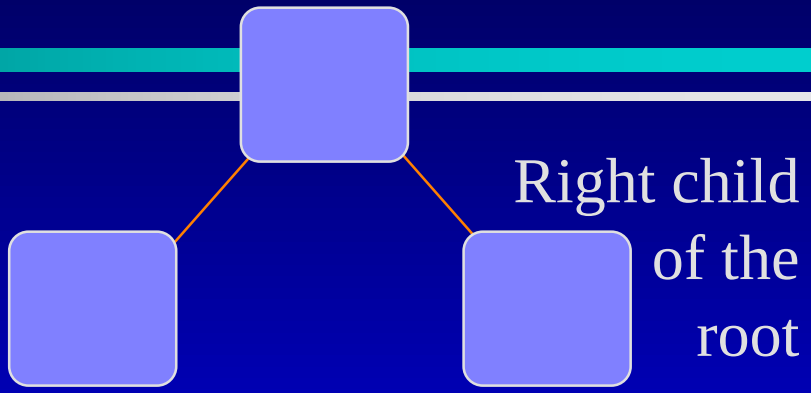
When a complete binary tree is built, its first node must be the root

Heaps



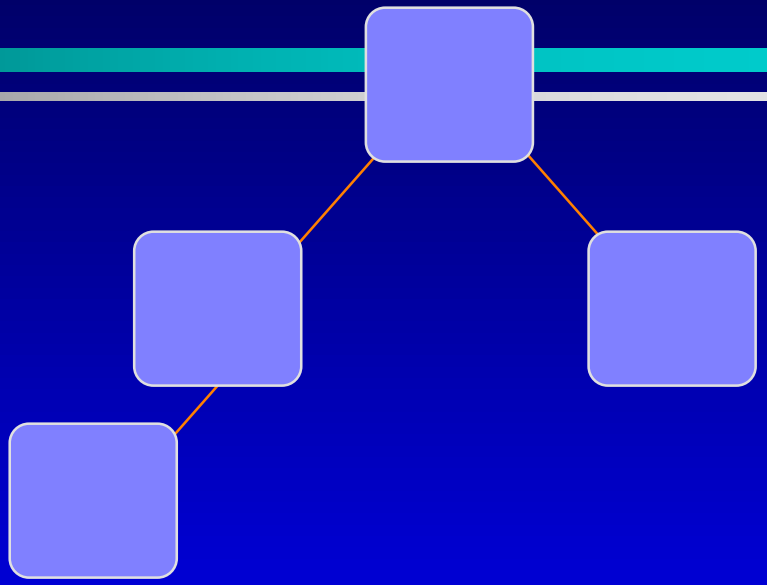
The second node is always the left child of the root

Heaps



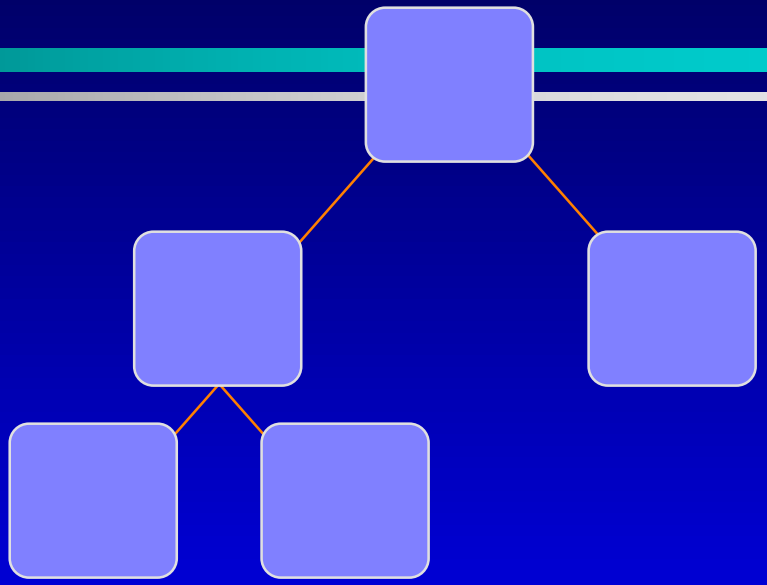
The third node is always the right child of the root

Heaps



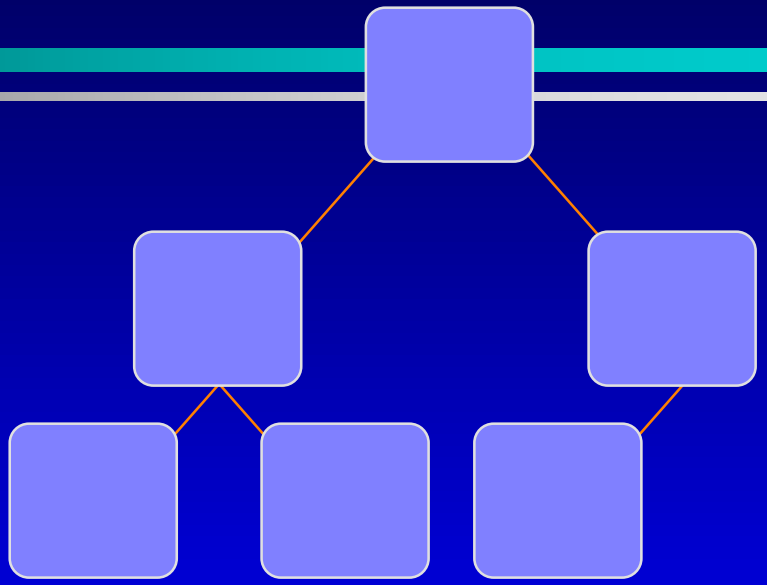
The next nodes always fill the next level from left-to-right

Heaps



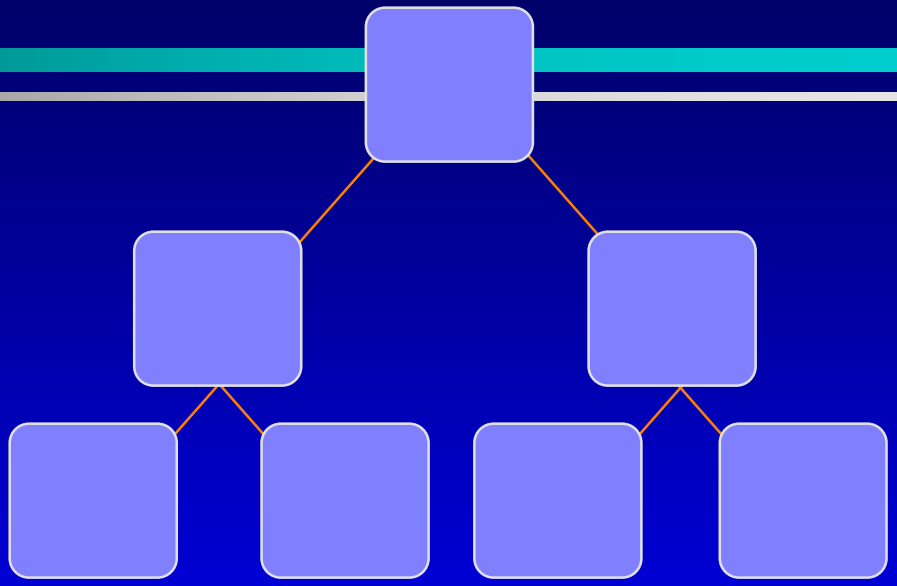
The next nodes always fill the next level from left-to-right

Heaps



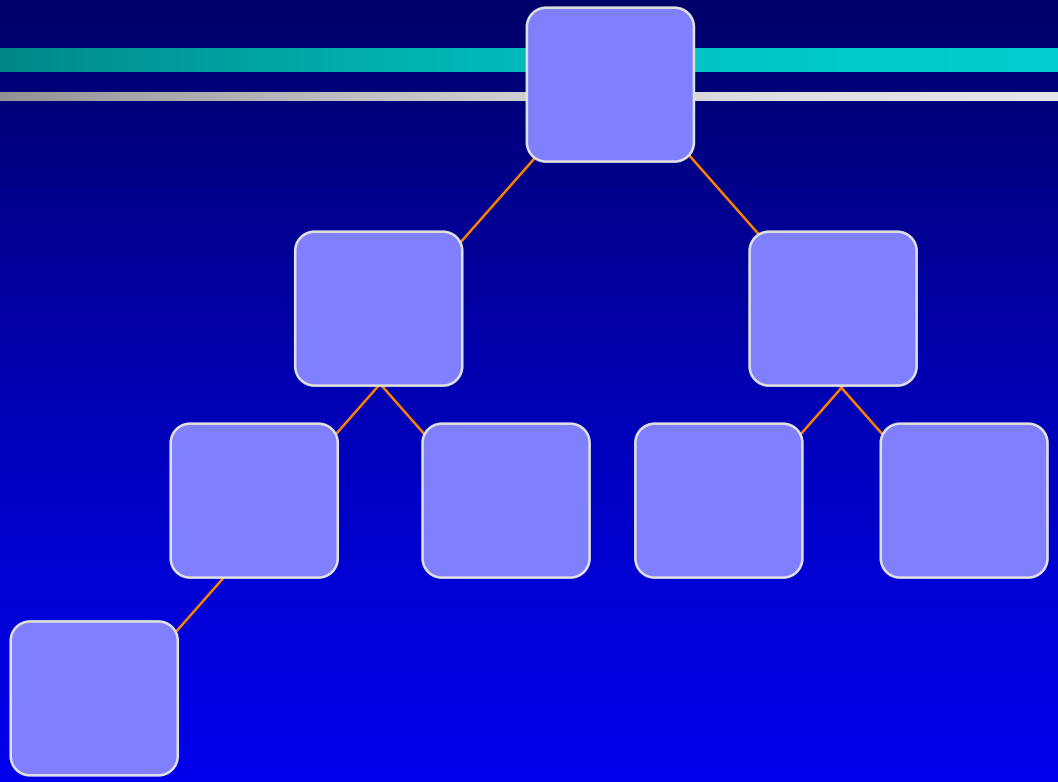
The next nodes always fill the next level from left-to-right.

Heaps

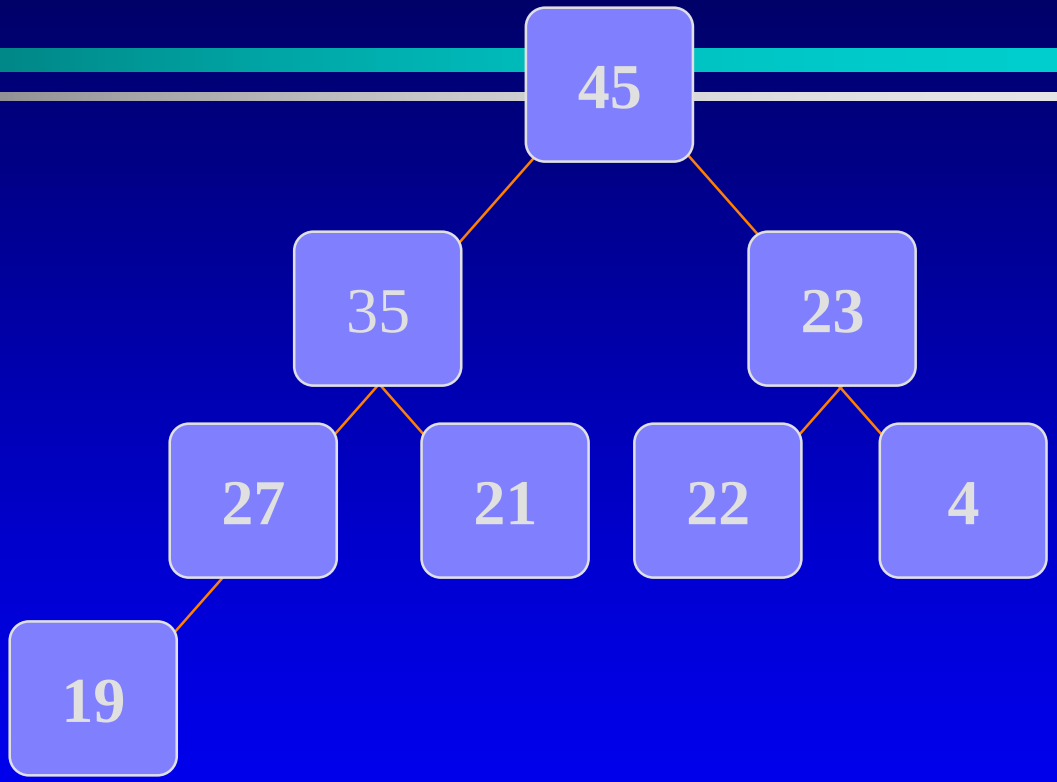


The next nodes always fill the next level from left-to-right

Heaps

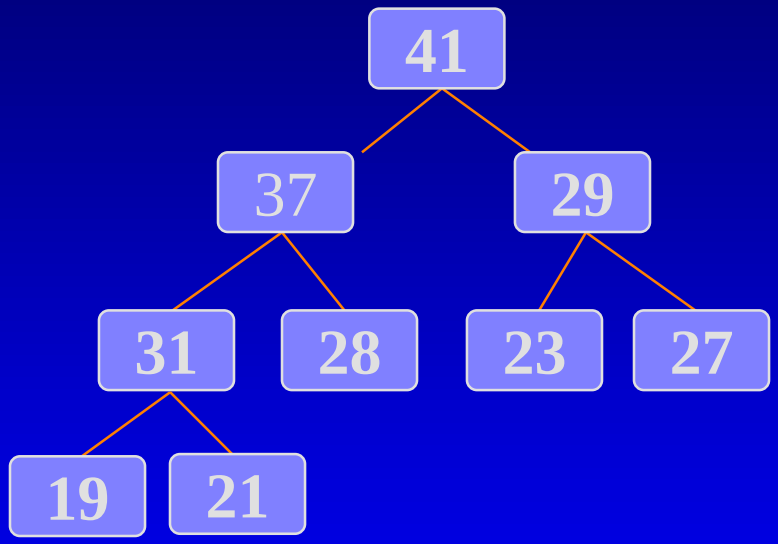


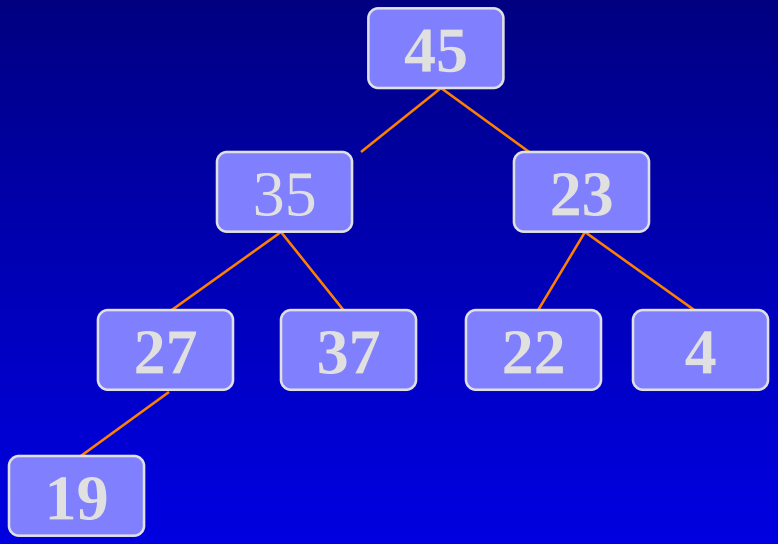
Heaps



A **Max-heap** is a complete binary tree, in which each node's key is greater than or equal to the keys of its children

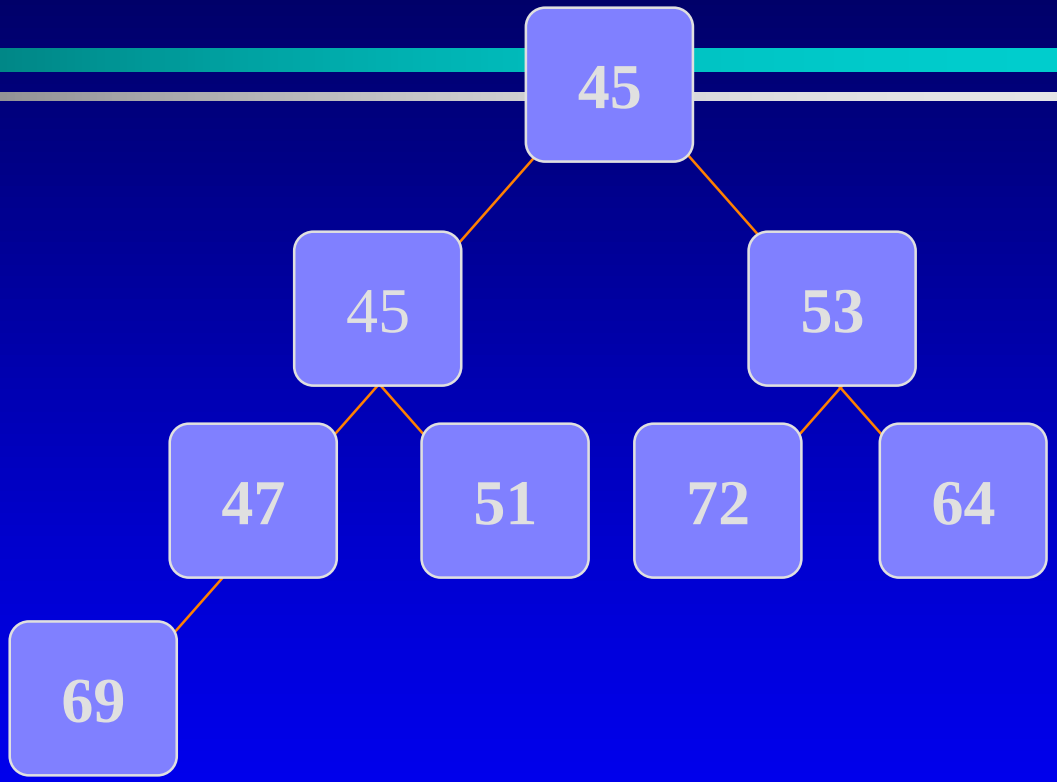
Heaps





Which of the trees is a MAX-heap?

Heaps

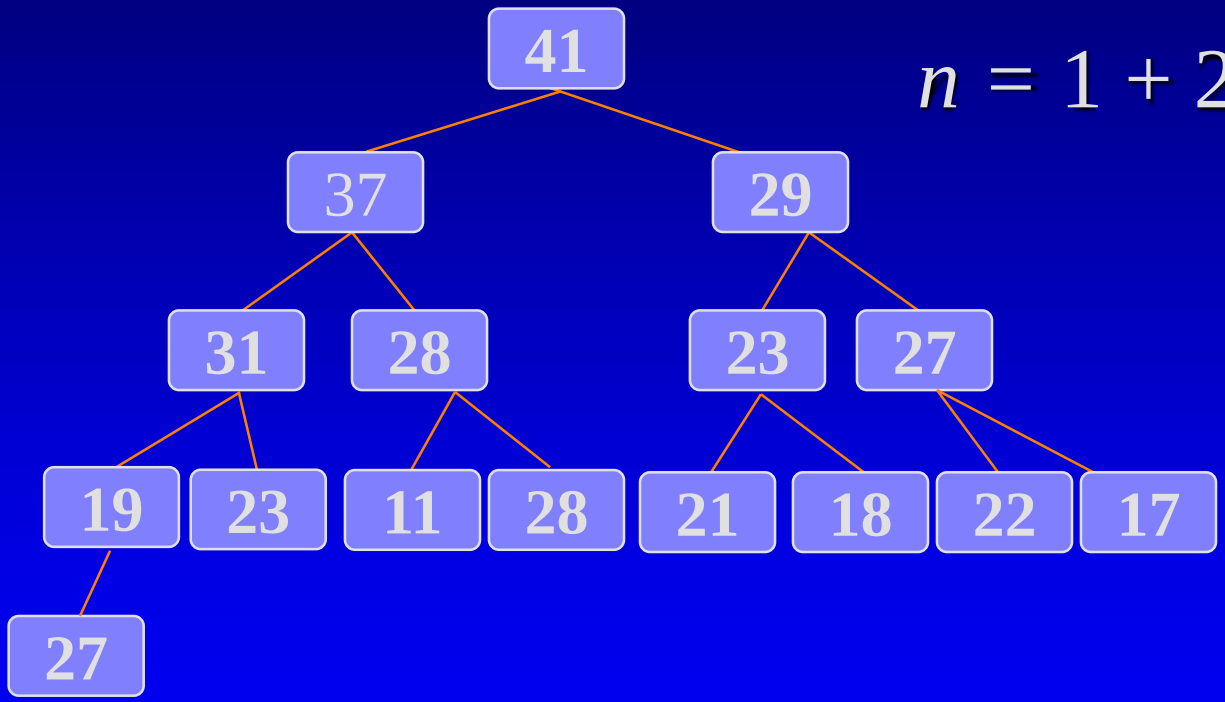


A **Min-heap** is a complete binary tree, in which each node's key is less than or equal to the keys of its children

Solve Problems

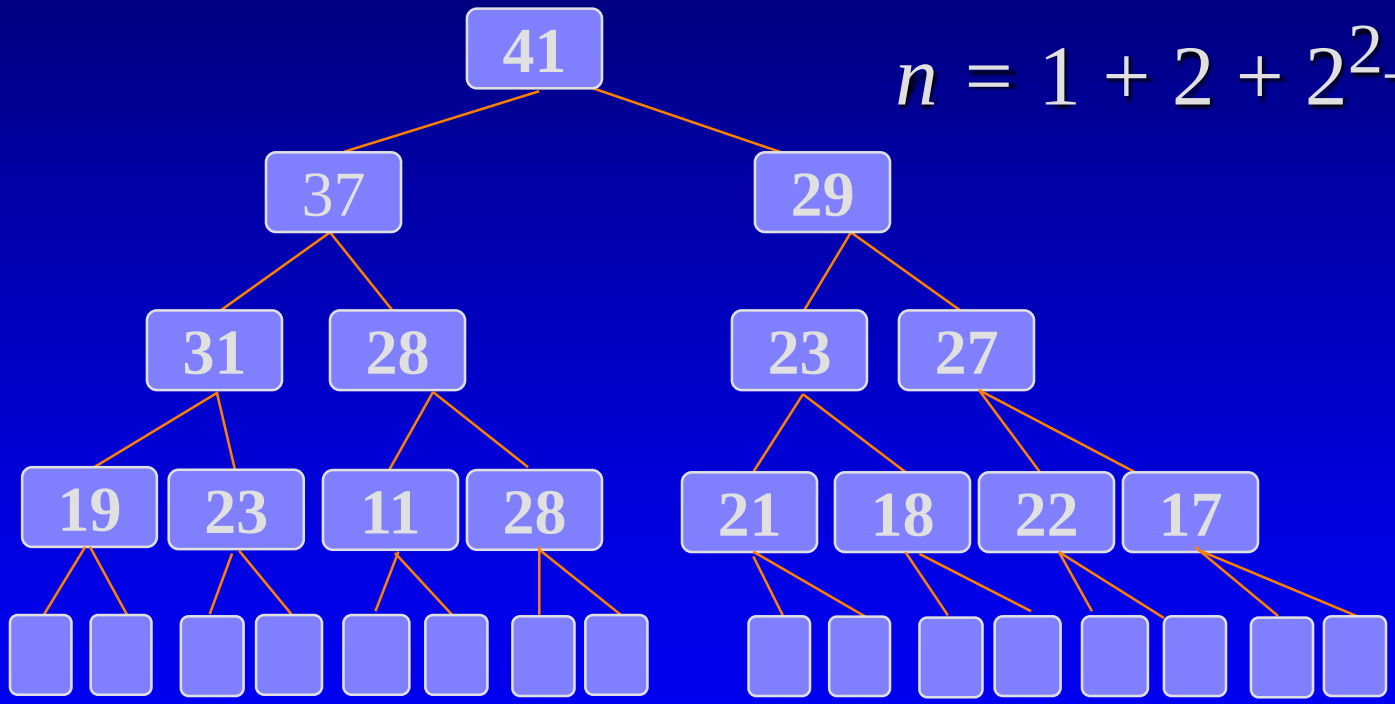
- What are the minimum and maximum numbers of elements in a heap of height h?
- Show that an n-element heap has height $\lfloor \lg(n) \rfloor$

What is the minimum number of nodes in a heap of height *h*?



$$n = 1 + 2 + 2^2 + \ldots + 2^{h-1} + 1$$

What is the maximum number of nodes in a heap of height *h*?



$$n = 1 + 2 + 2^2 + \ldots + 2^h$$

Show that n-element heap has the height $h = \lfloor \lg(n) \rfloor$

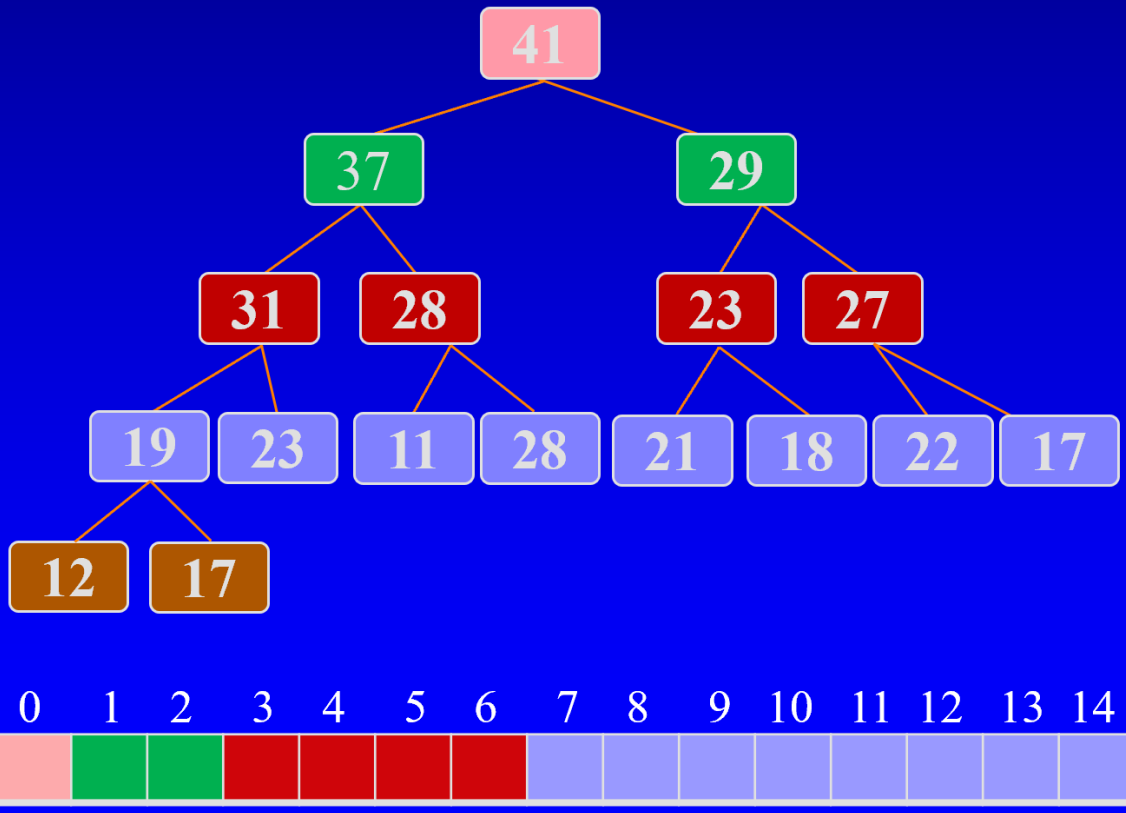
- $2^h \leq n \leq 2^{h+1} - 1$
- $h \leq \lg(n)$ (leftmost inequality)
- $\lg(n+1) \leq h + 1$ (rightmost inequality)
- $\lg n < \lg(n+1) \leq h + 1$
- $h \leq \lg(n) < h + 1$
- $h = \lfloor \lg(n) \rfloor$

Indices of leaves of an *n*-element heap

Show that the leaves are the nodes indexed by

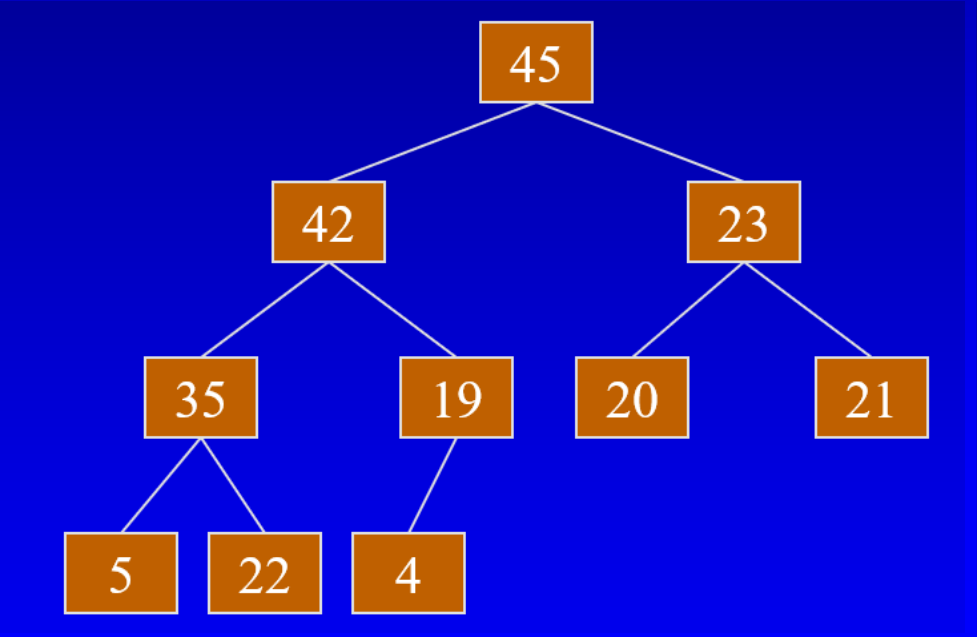
$m+1, m+2, \ldots,$

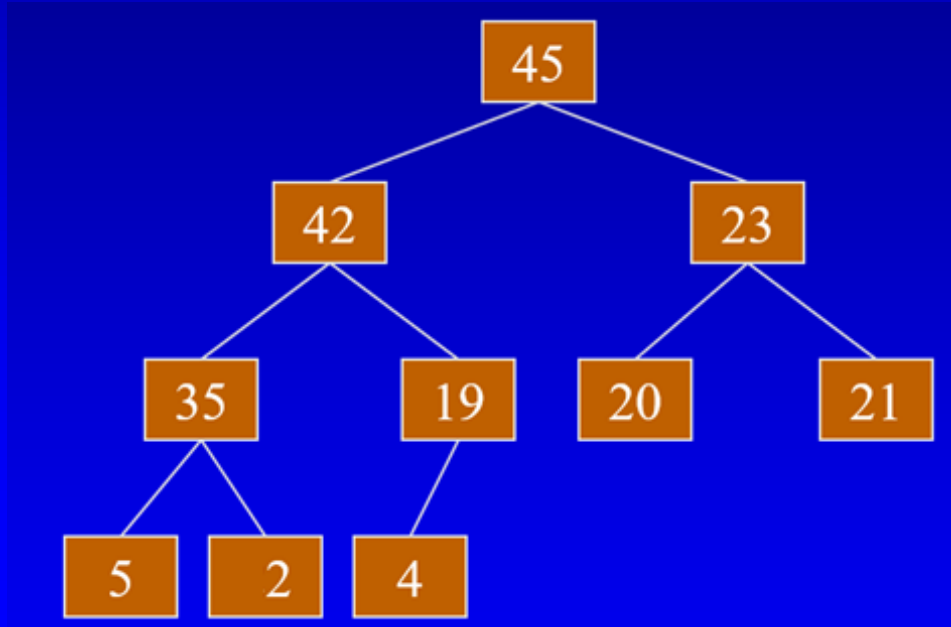
$m = \lfloor (n-2)/2 \rfloor$



Solve Problem

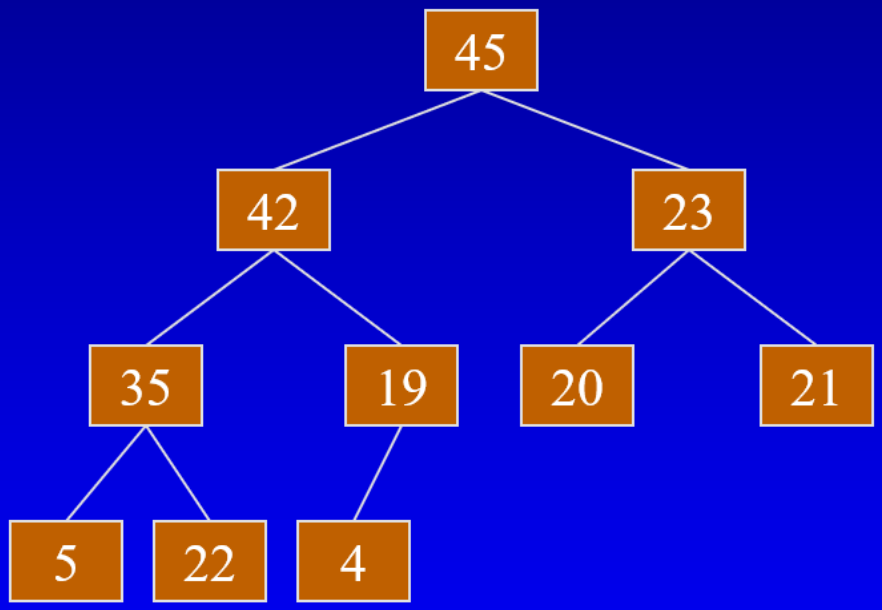
- Where in a max-heap might the smallest element reside, assuming that all elements are distinct

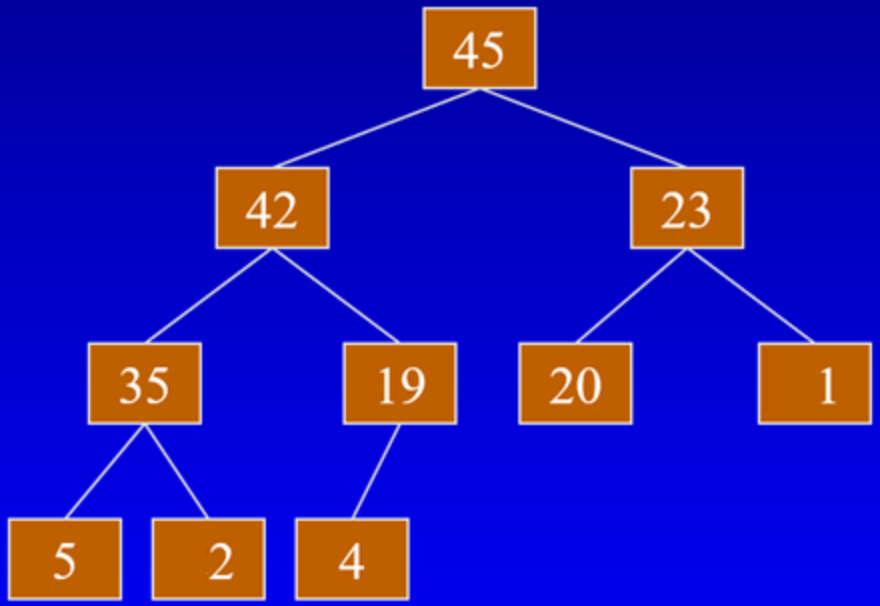




Solve Problem

- Where in a max-heap might the smallest element reside, assuming that all elements are distinct



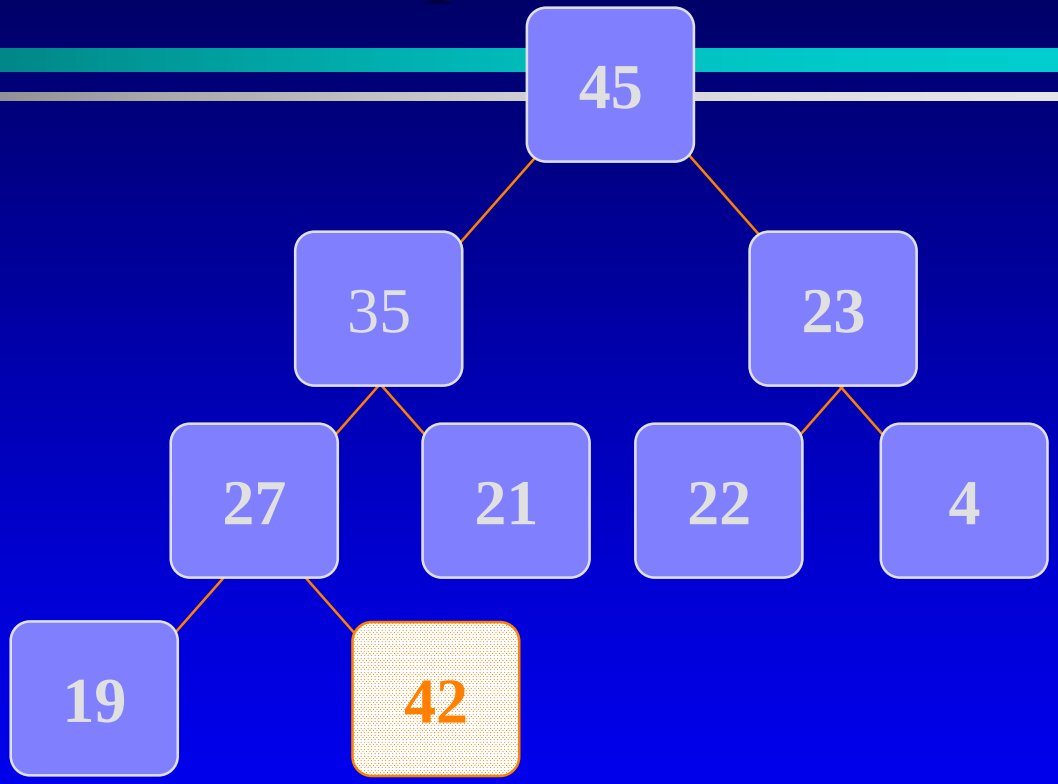


Operations on a Heap

1. Add node to a heap
2. Remove the root

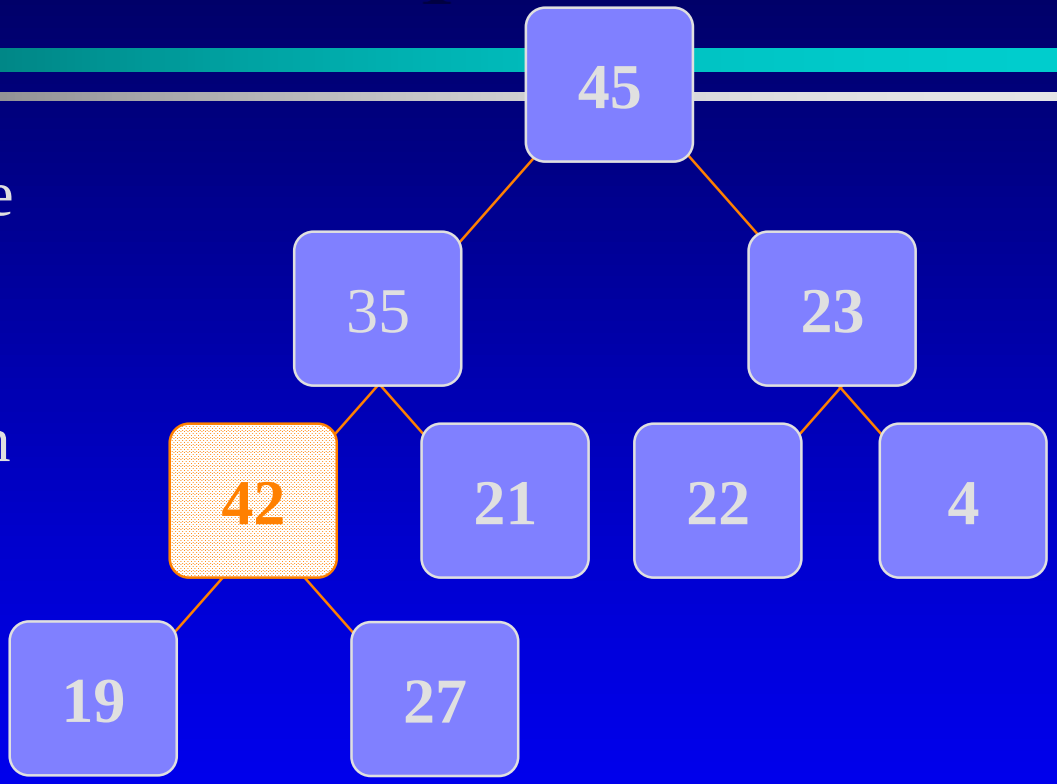
Adding a Node to a Heap

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



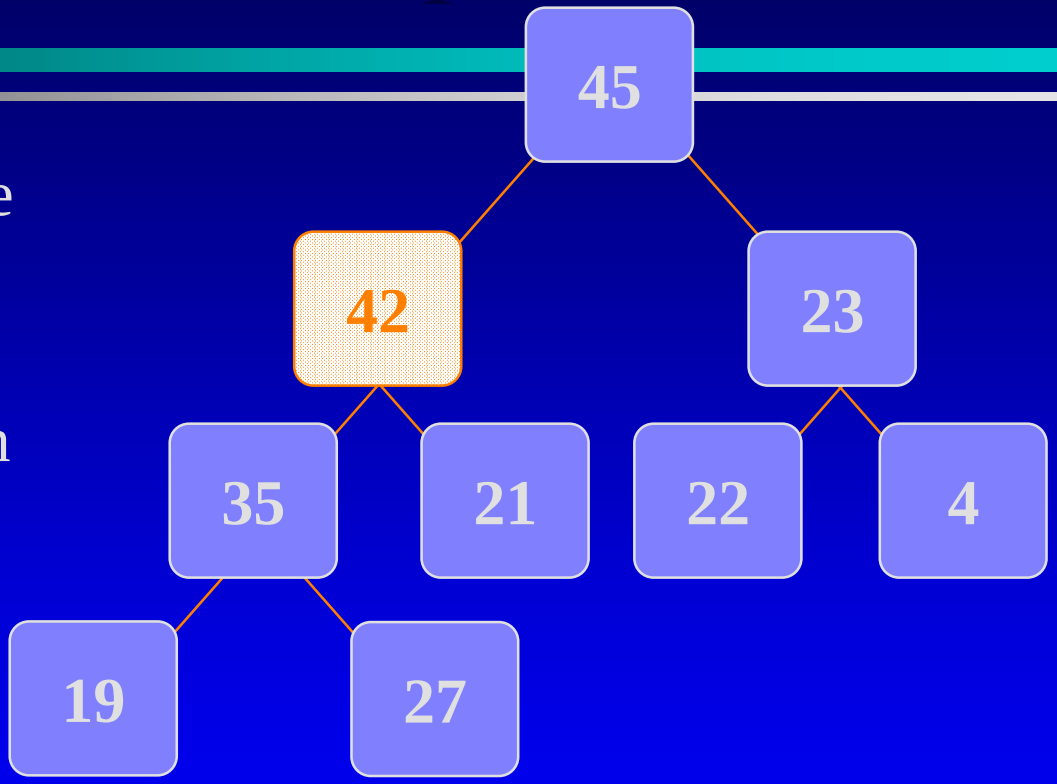
Adding a Node to a Heap

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



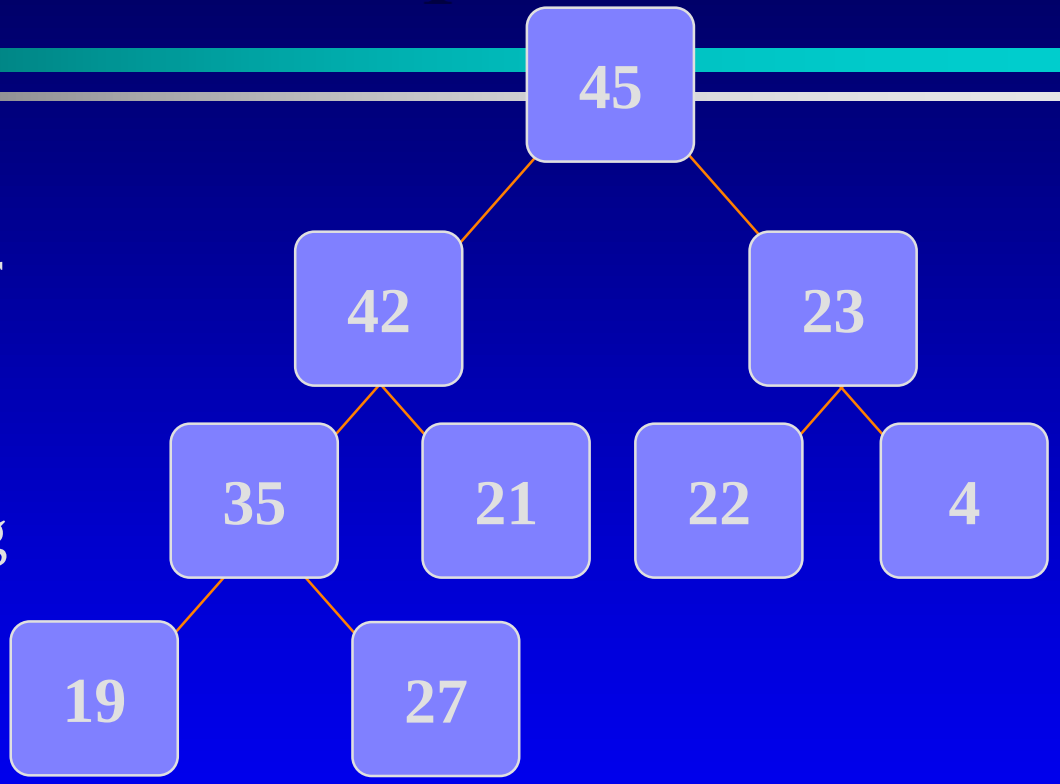
Adding a Node to a Heap

1. Put the new node in the next available spot.
2. 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**

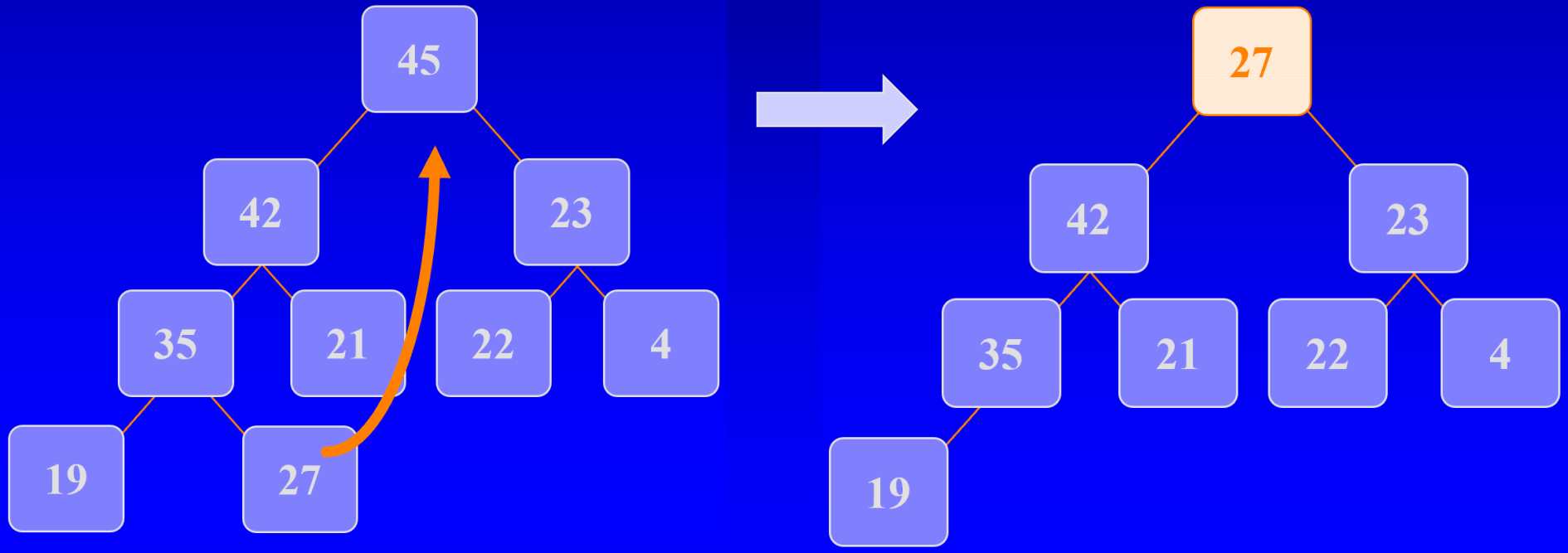


Adding a Node to a Heap

- Time to add a node to a heap is O(lg *n*)
- Justification:
  - Heap is a complete binary tree of *n* nodes
  - Height of a complete binary tree is lg *n*
  - To insert a new node to the first available spot is O(1) time
  - Reheapification upward takes constant number of operations (comparison, swap) taken at most height of the tree times, hence, O(lg *n*)

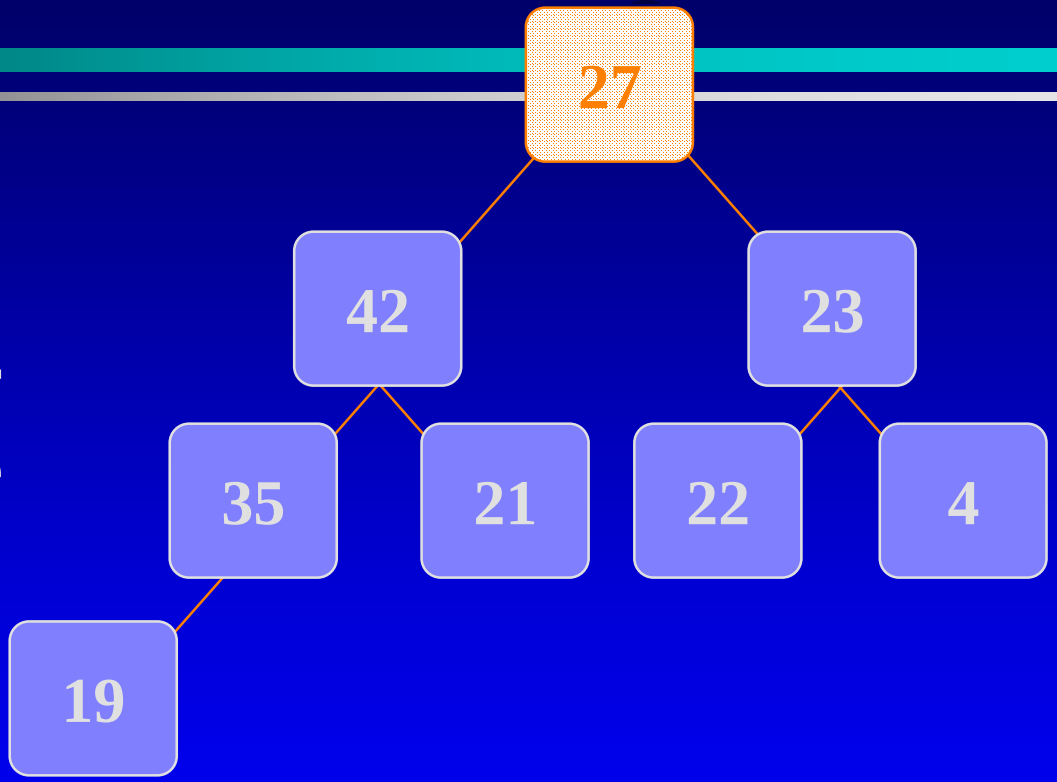
Removing the Root of a Heap

1. Replace the root node with the last node



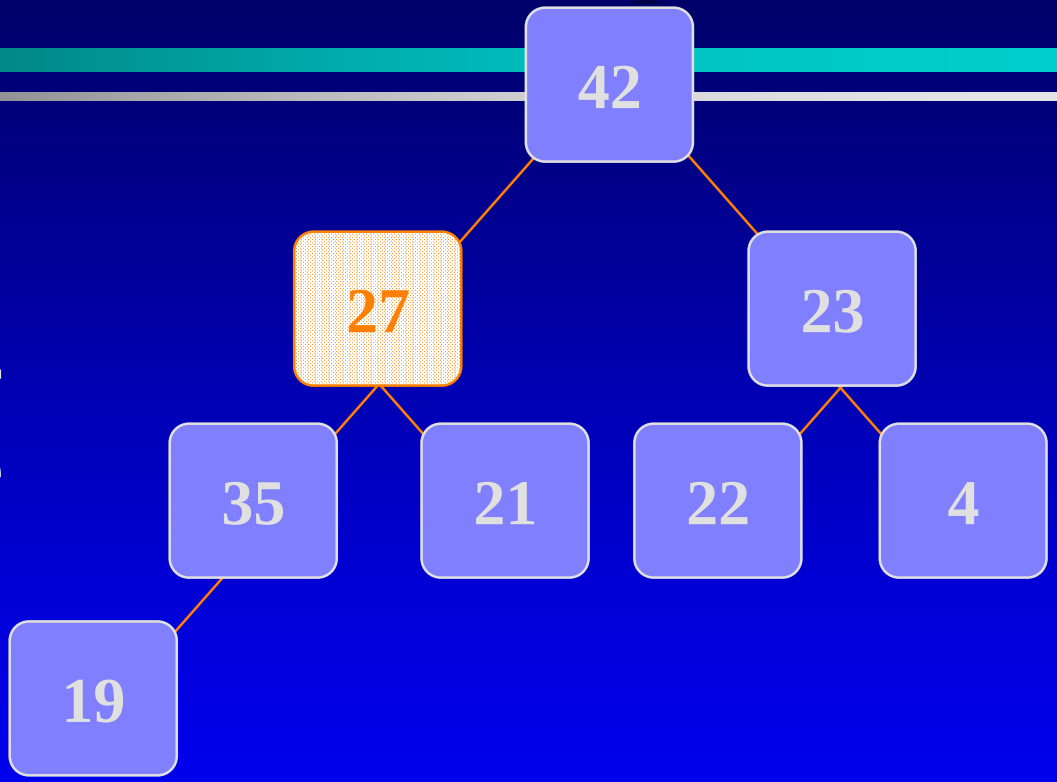
Removing the Root of a Heap

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



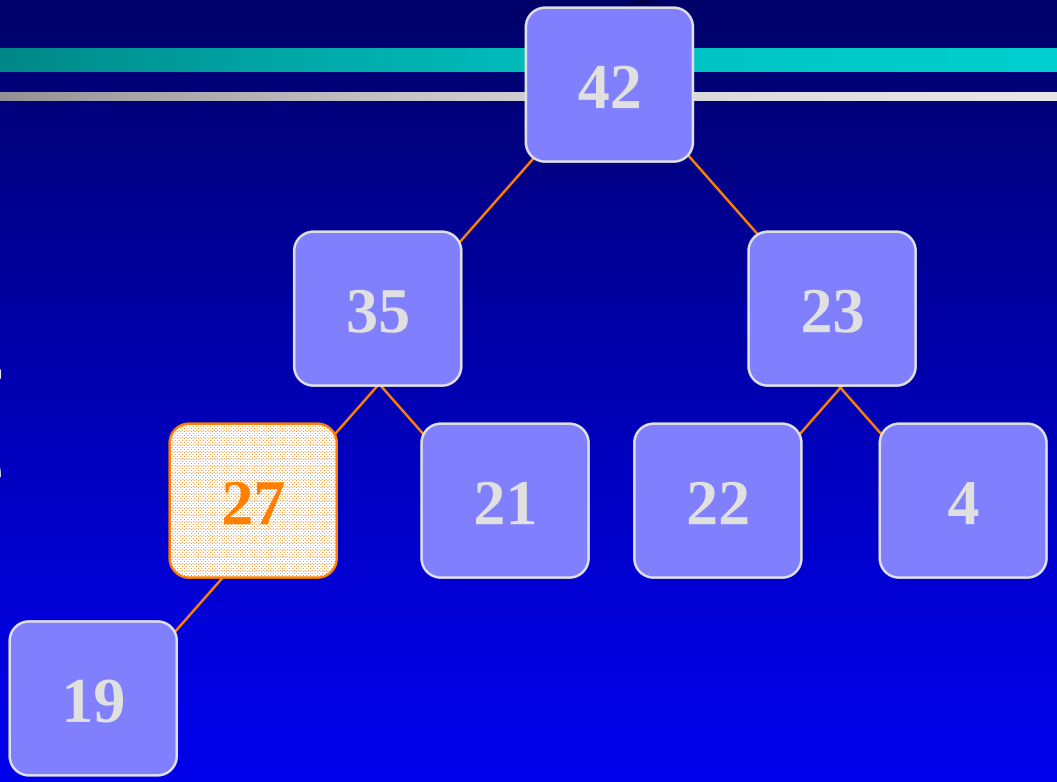
Removing the Root of a Heap

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



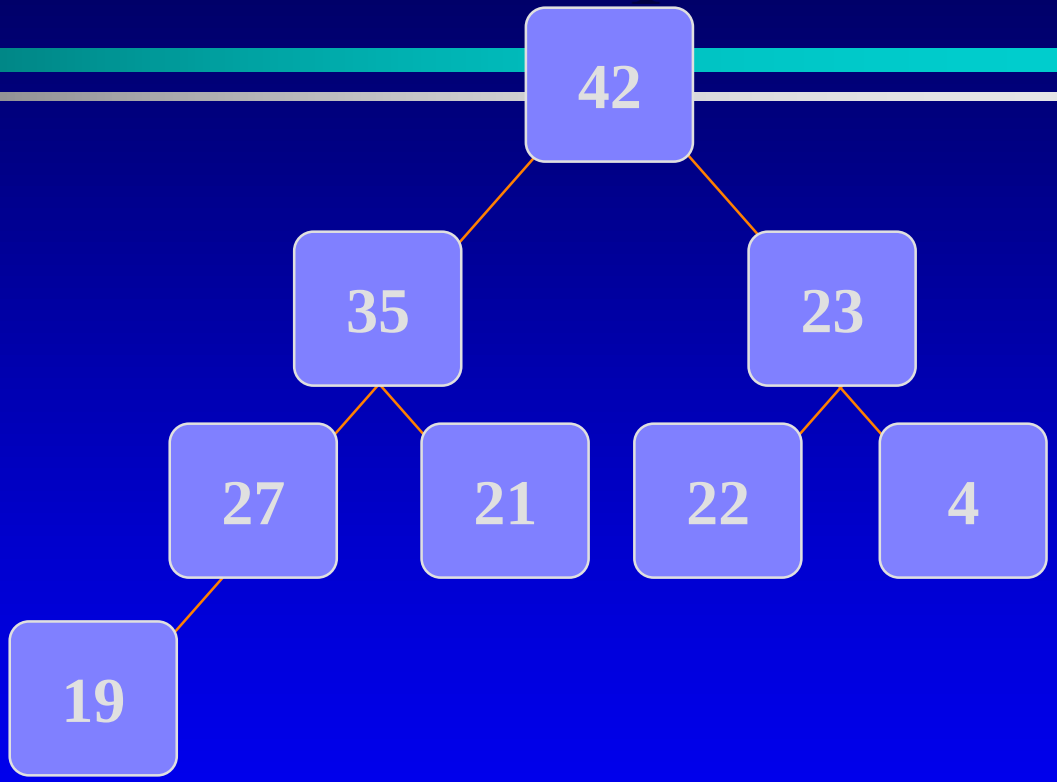
Removing the Root of a Heap

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



Removing the Root of a Heap

- ✓ Both children have keys <= key of the node or
- ✓ The node reaches the leaf
- ↘ The process of pushing the node downward is called **reheapification downward**

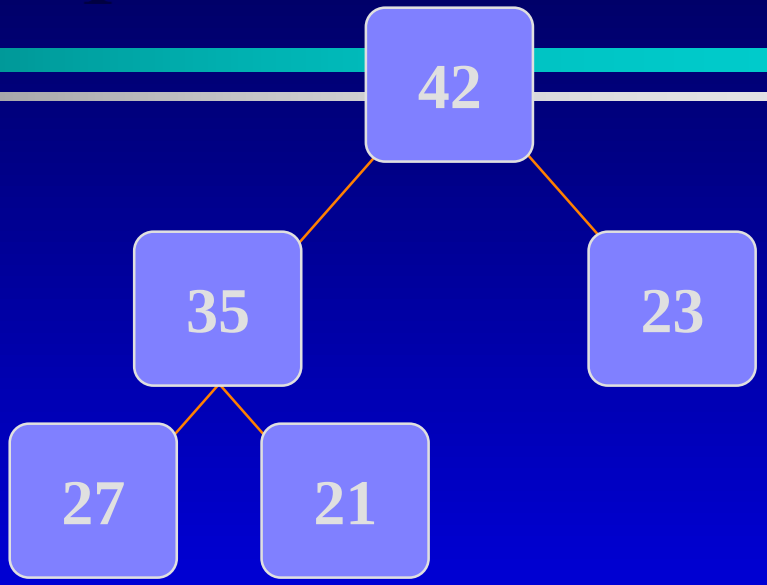


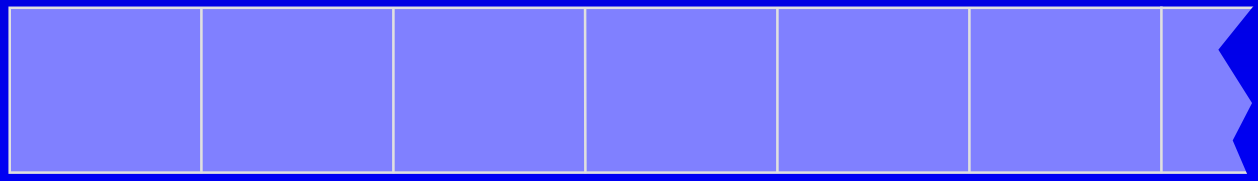
Removing the Root of a Heap

- Takes O(lg n) time
- Justification:
  - To replace the top with the last element takes O(1)
  - Reheapification downwards takes at most height times, each of which takes a constant number of operations
  - Height is lg n

Implementing a Heap

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

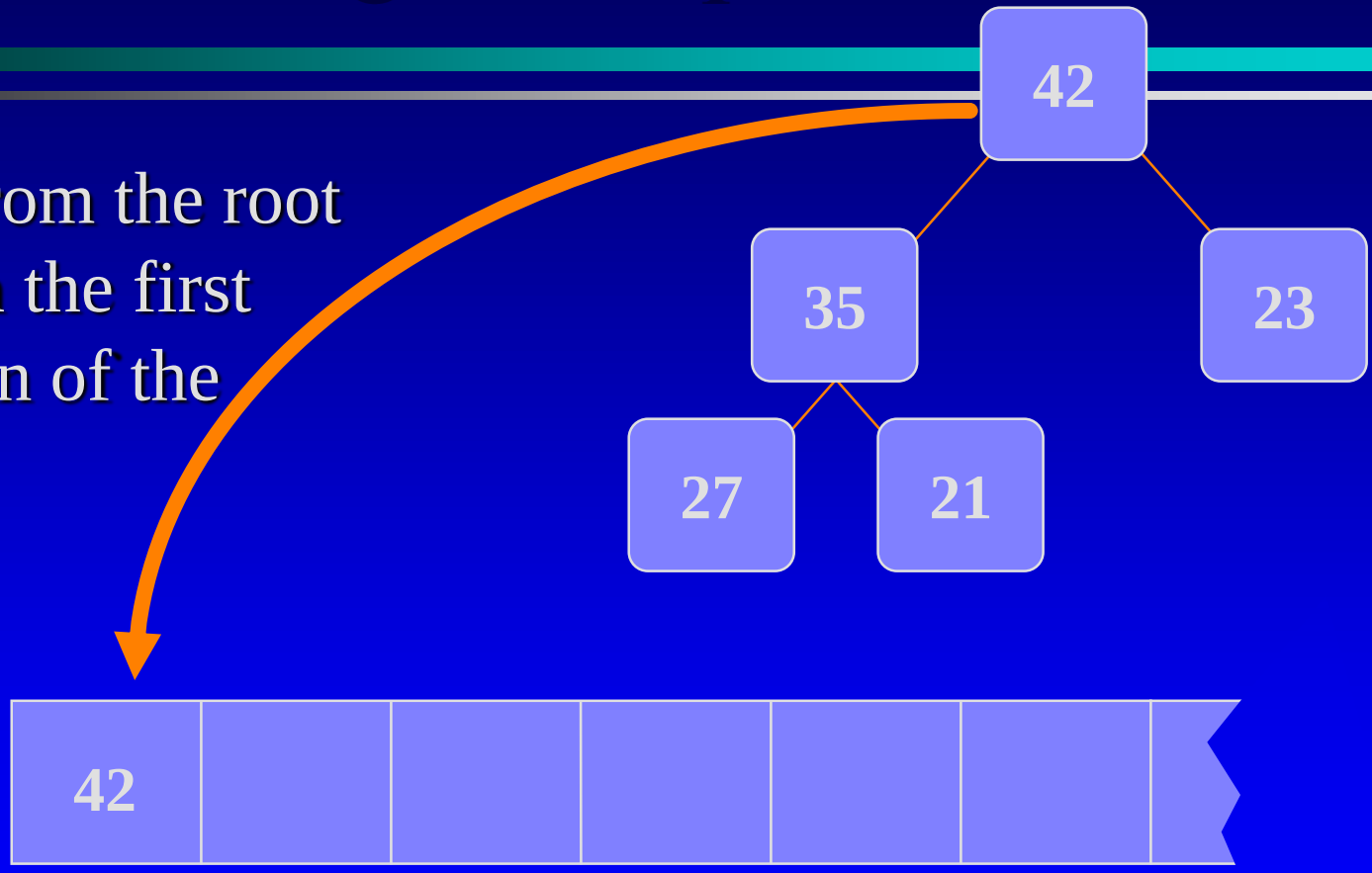




An array of data

Implementing a Heap

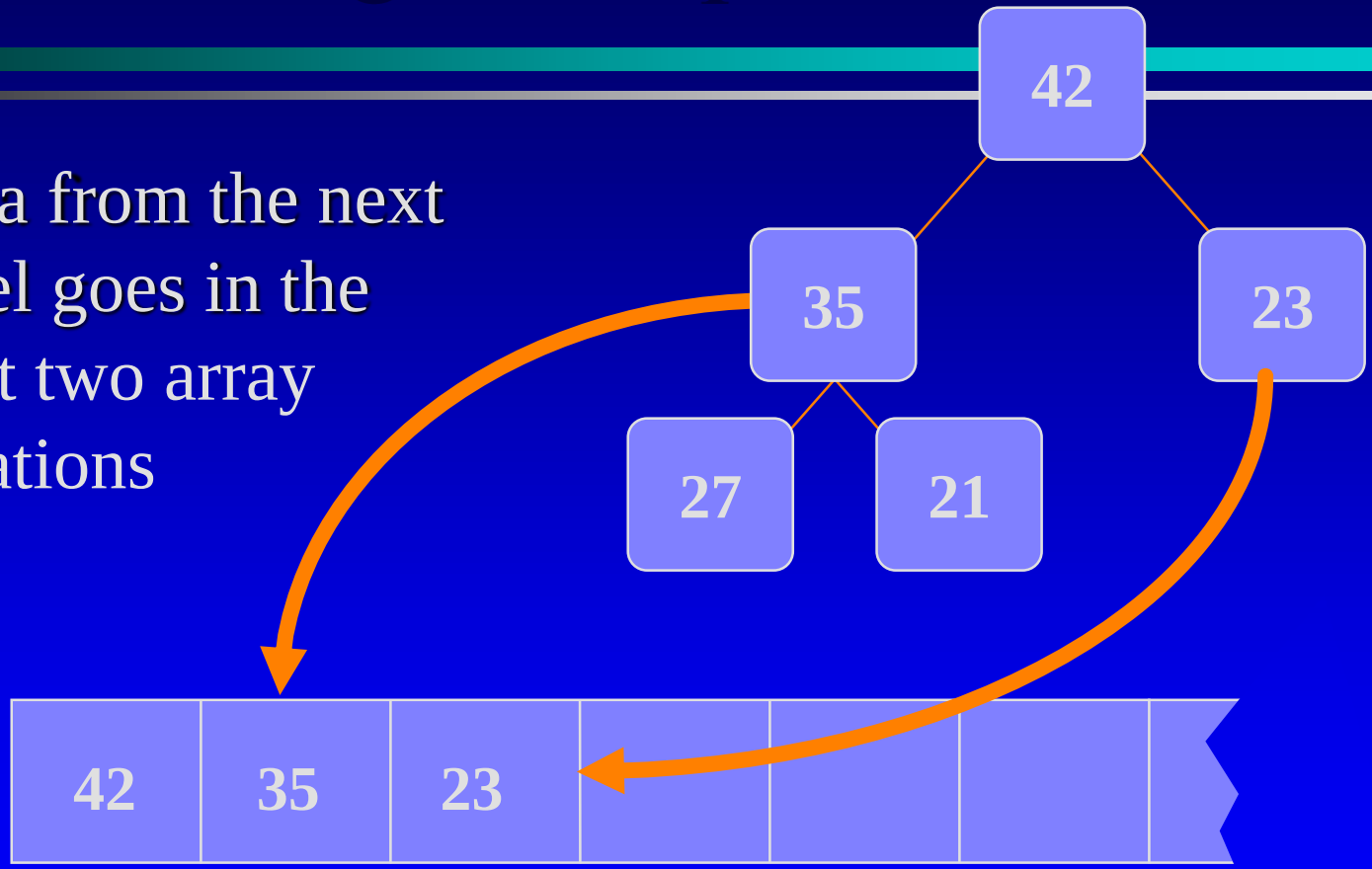
- Data from the root goes in the first location of the array



An array of data

Implementing a Heap

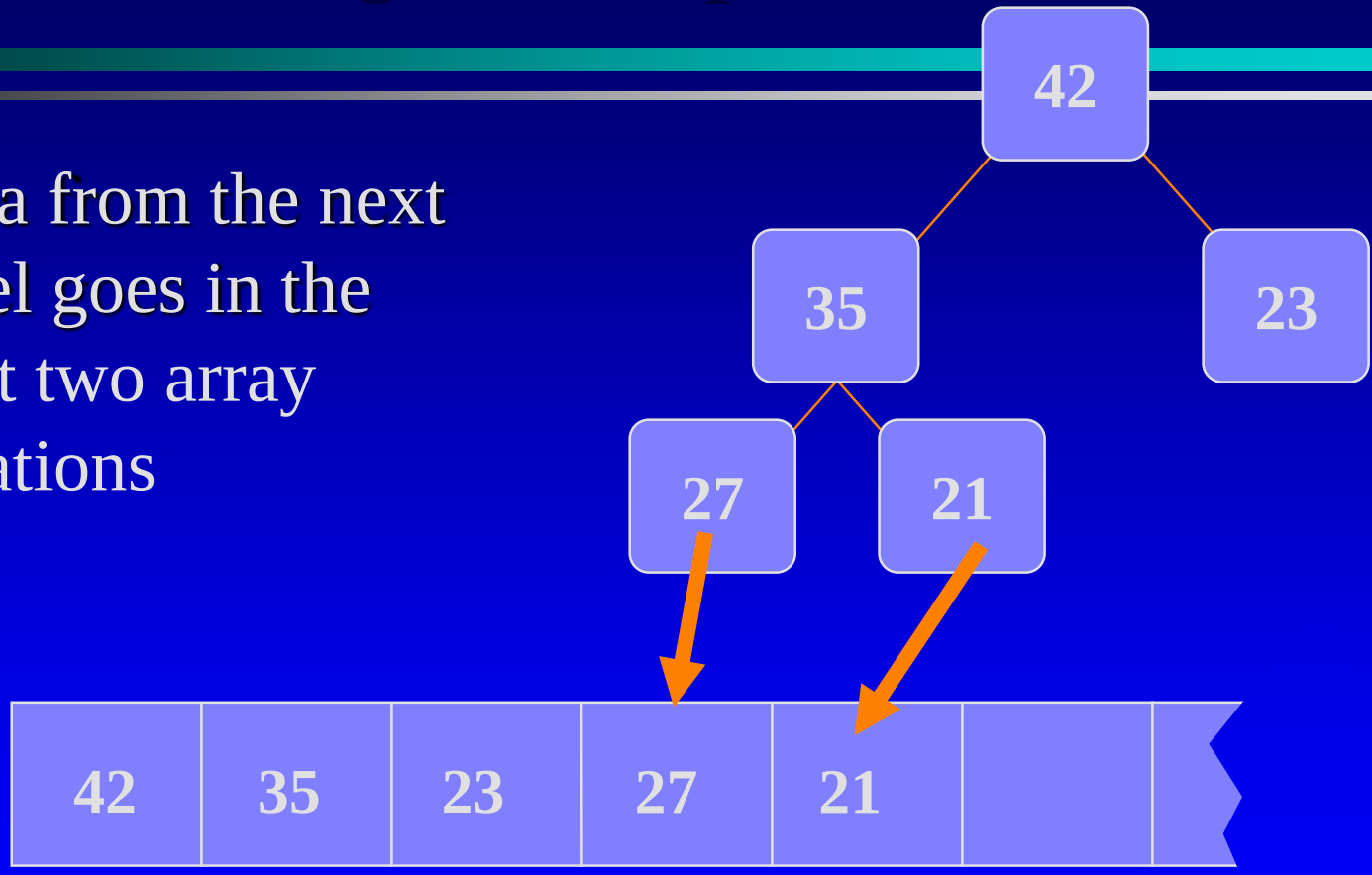
- Data from the next level goes in the next two array locations



An array of data

Implementing a Heap

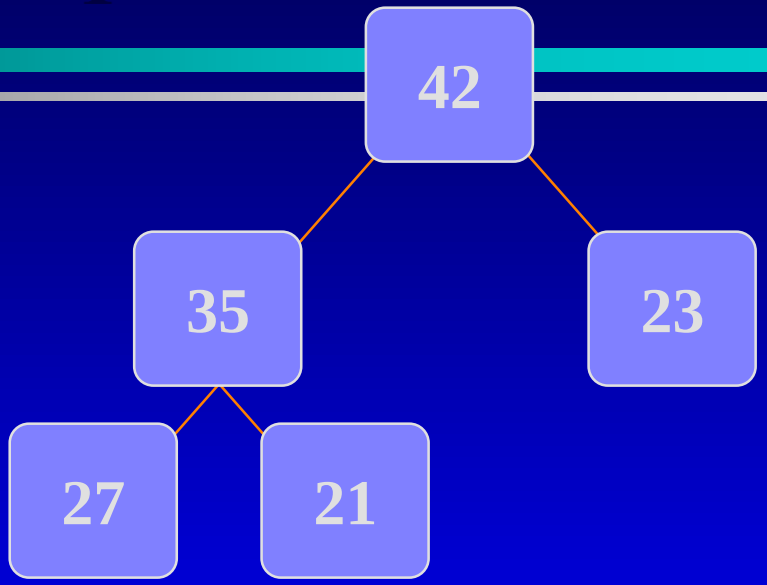
- Data from the next level goes in the next two array locations



An array of data

Implementing a Heap

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

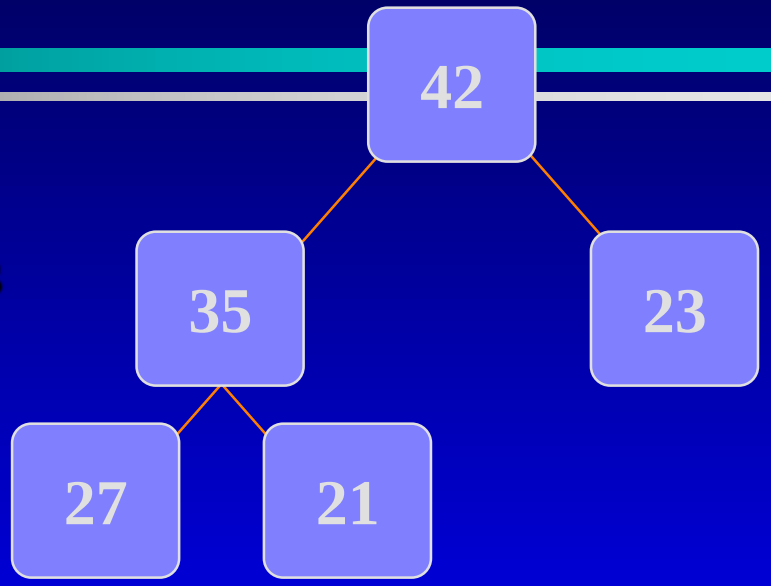


An array of data

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

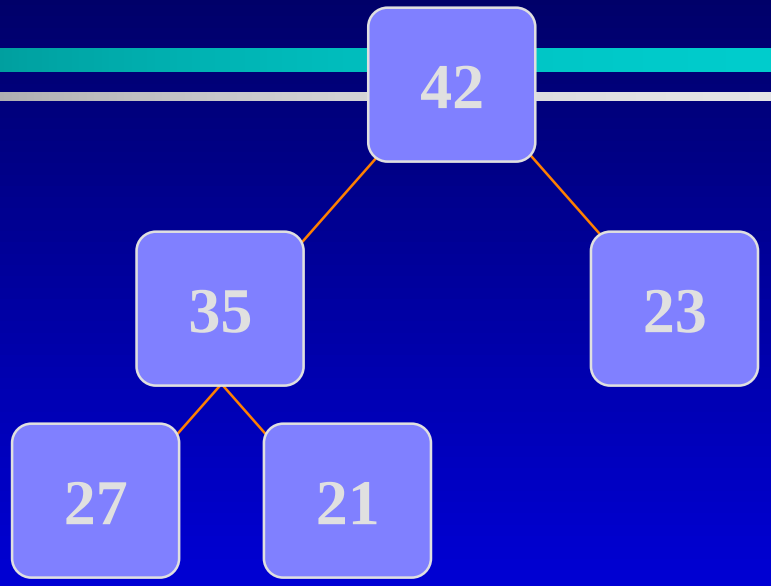
Important Points about the Implementation

- 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



An array of data

Important Points about the Implementation



- If you know the index of a node, then it is easy to figure out the indices of that node's parent and children
- If node is at [i]-th entry,

  its children are at 2i + 1, 2i + 2
- Parent of a child at [i]-th entry is at $\lfloor (i - 1)/2 \rfloor$

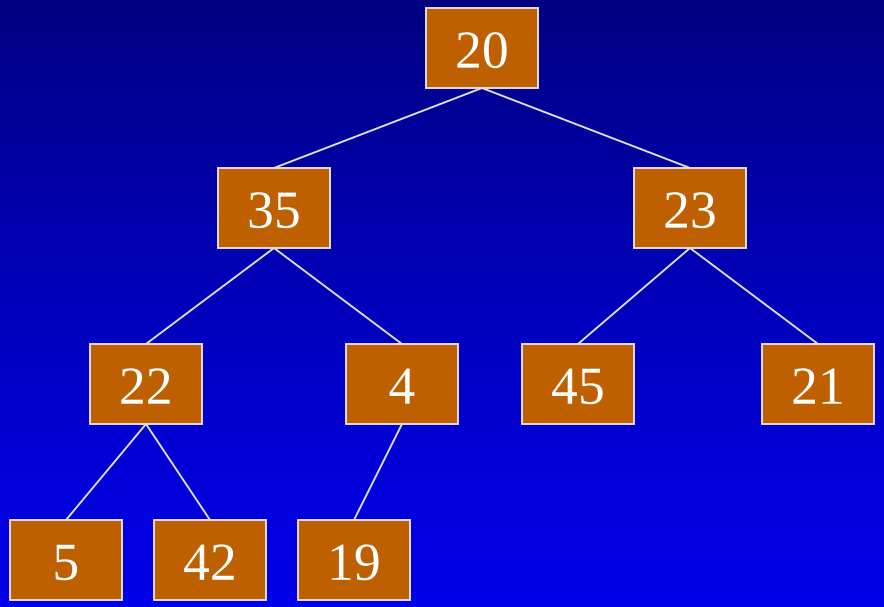
Building a Heap

- Two ways to build a heap with given values:
  - Insert one value at a time using an insertion
    - O($n \lg n$)
  - Build an arbitrary complete binary tree and then perform reheapification downward
    - O(n)
- The second way is more optimal

Building a Heap

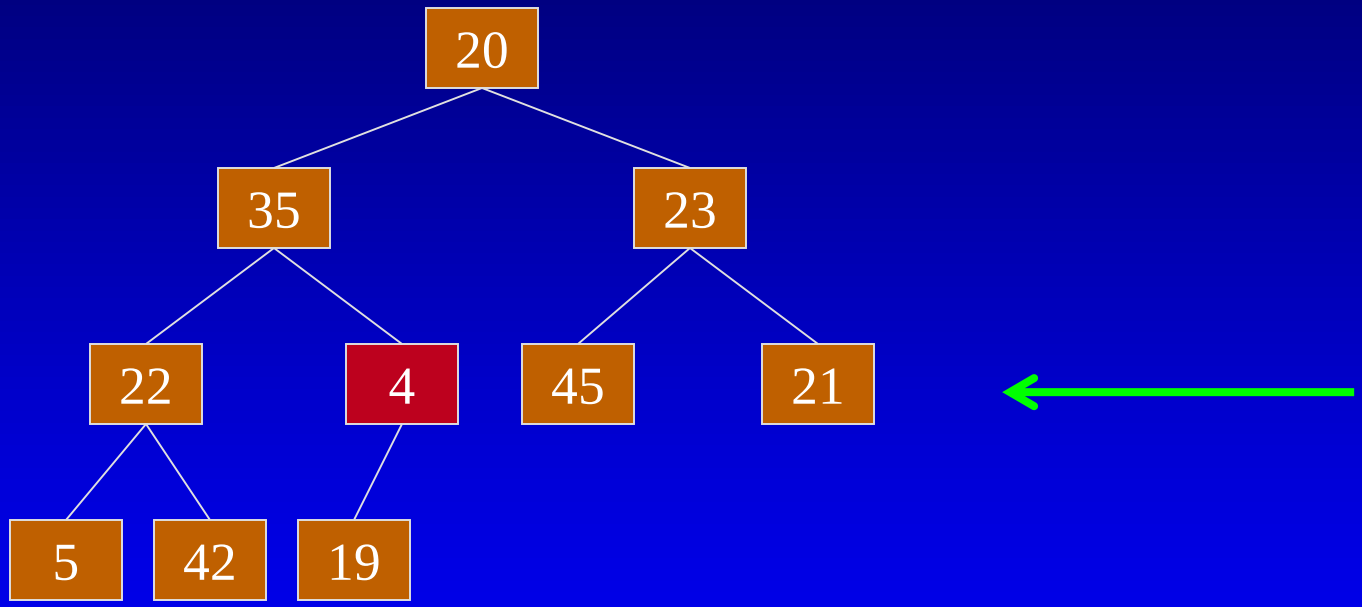
- Starting from the lowest level, move level by level up
- At each level, do reheapification downward of each node treating it as a root of a heap (the subtree rooted at that node)

Build a heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



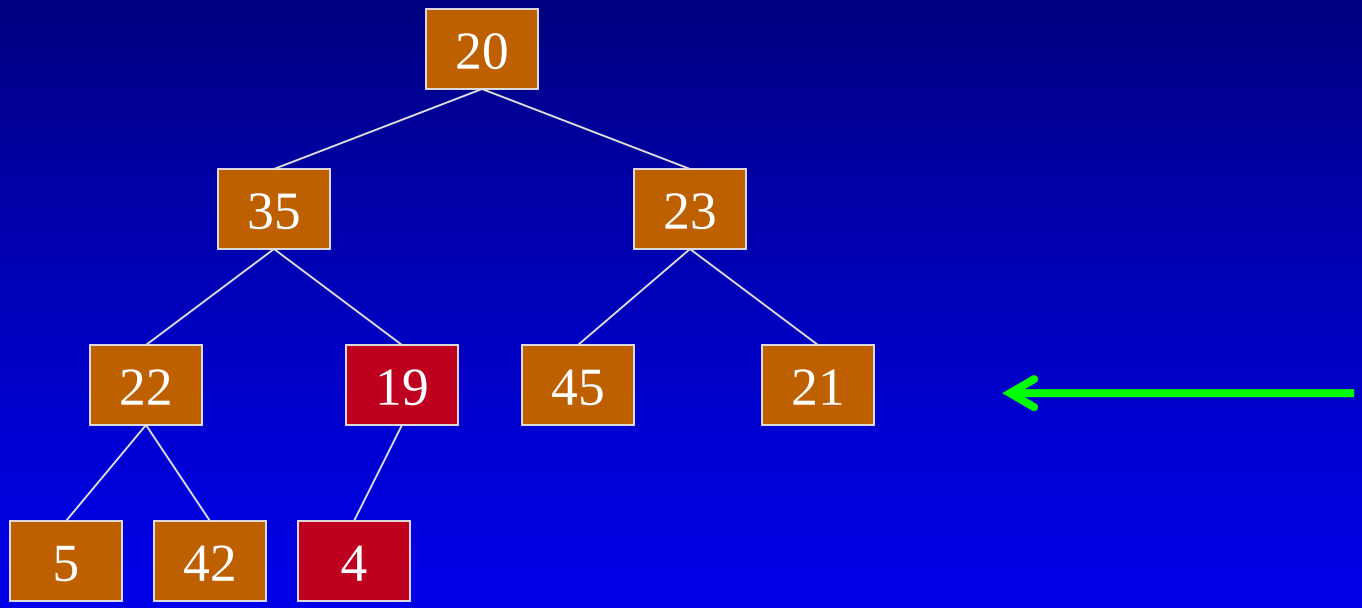
- First, we build an arbitrary complete binary tree.
- Then, starting from the bottom level
  - Reheapify downward each node
  - Swap with the larger child

Build a heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



4 < 19

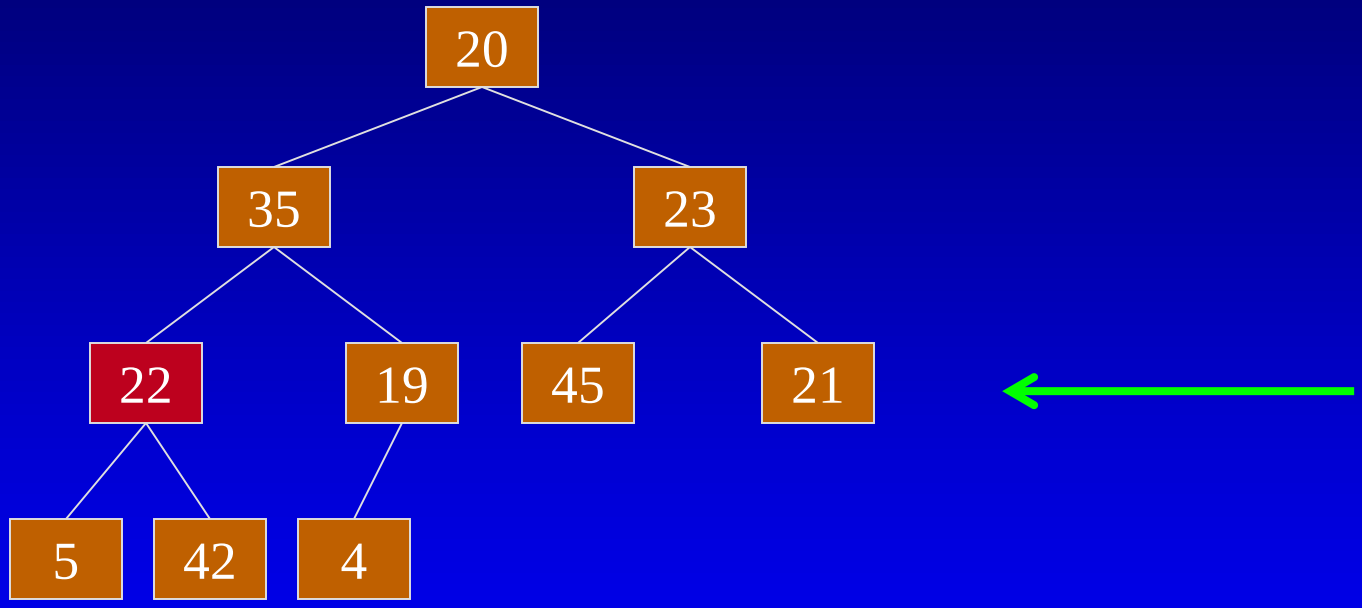
Build a heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



Swap 4 and 19

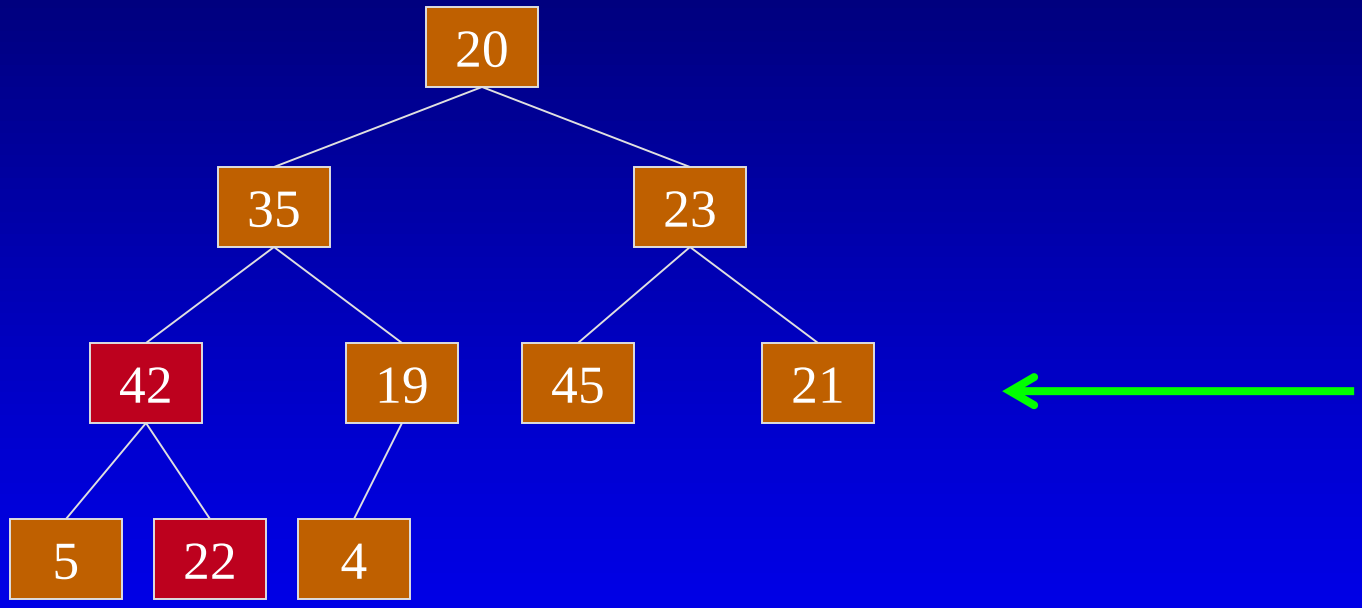
4 is a leaf node, so reheapification downward is complete

Build a heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



22 < 42

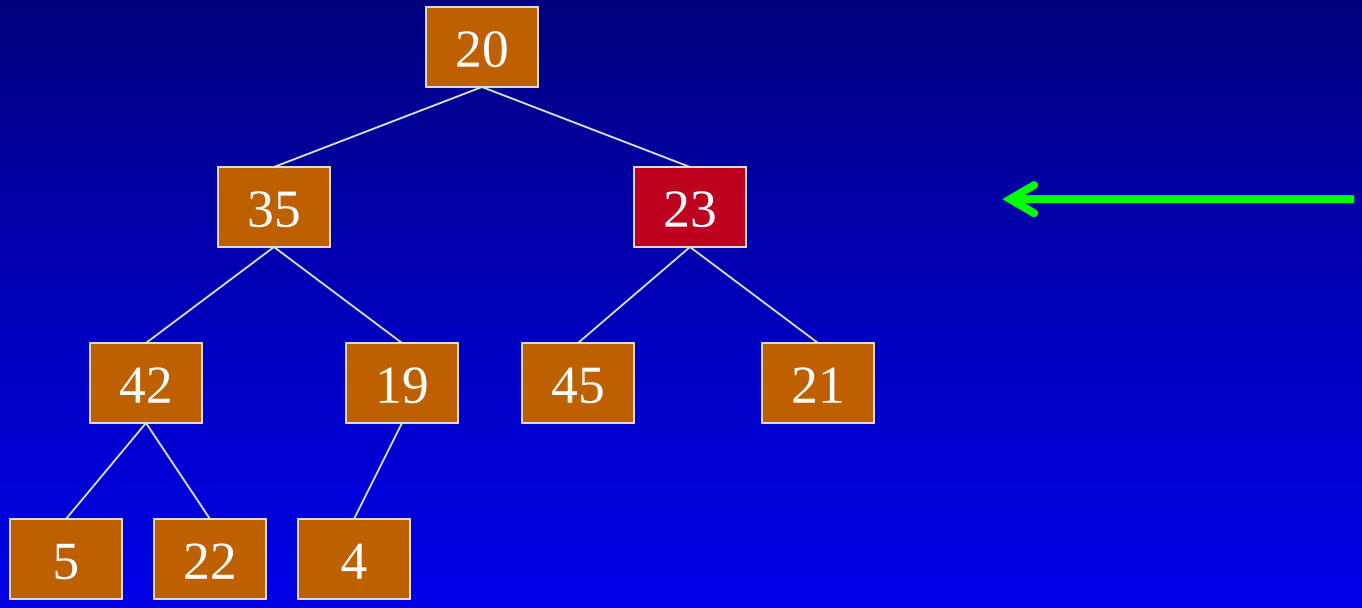
Build a heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



Swap 22 and 42

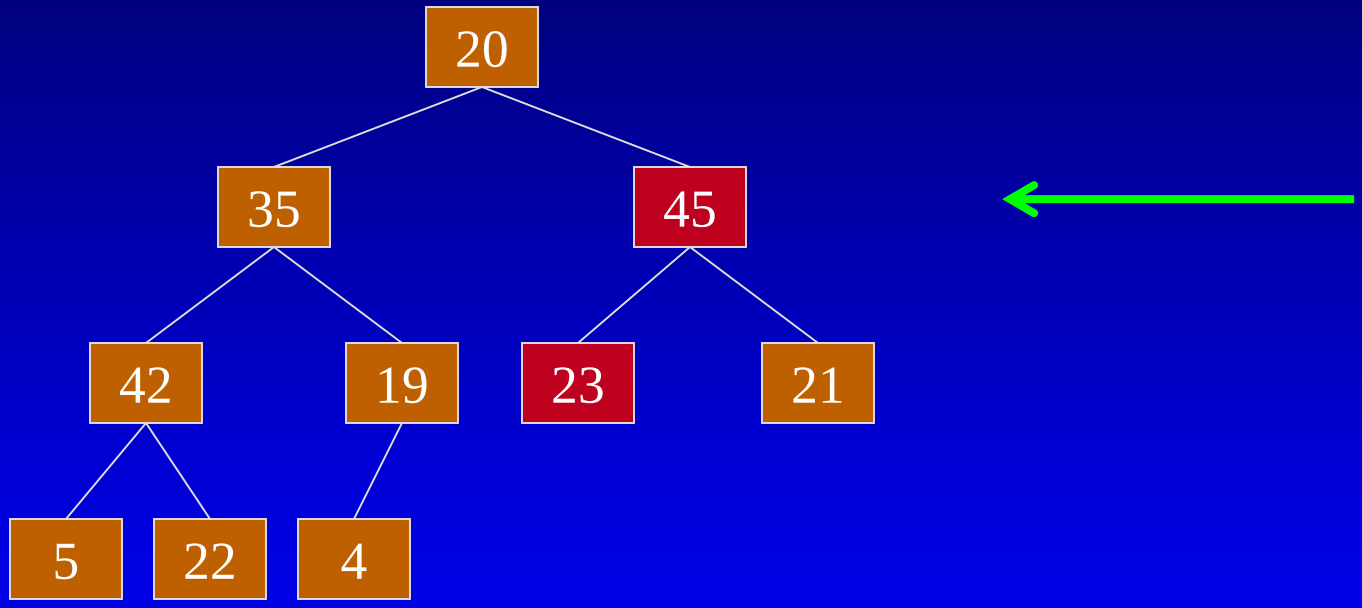
22 is a leaf node, so reheapification downward is complete

Build a heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



23 < 45

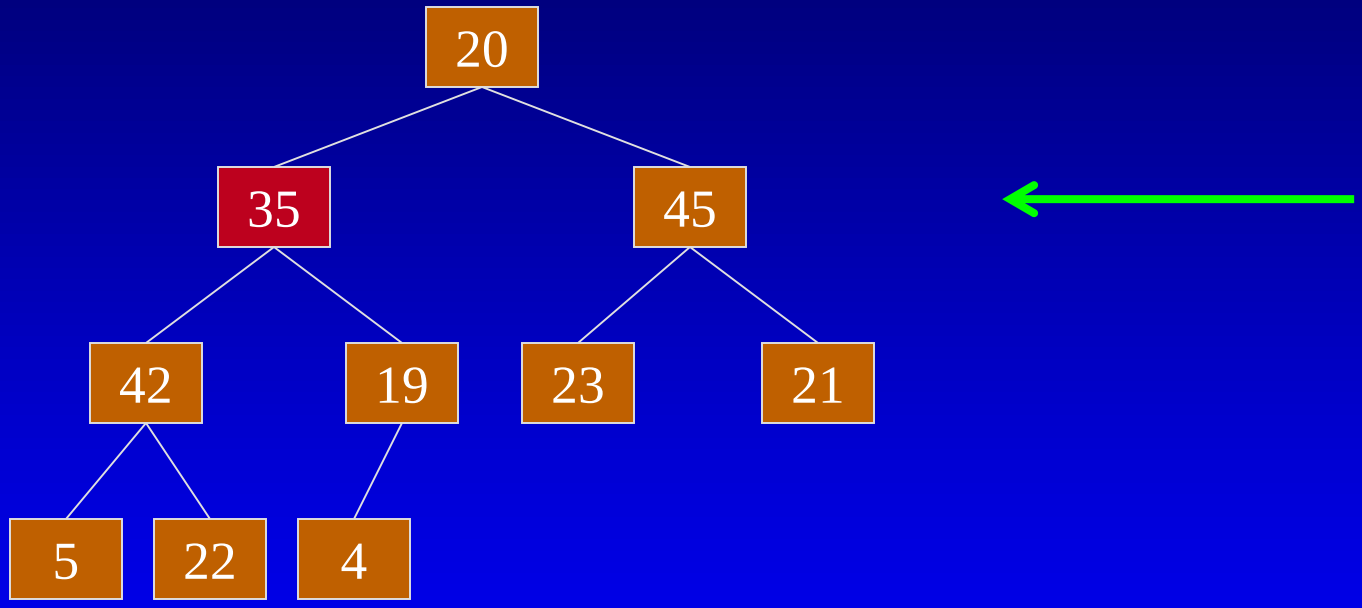
Build a heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



Swap 23 and 45

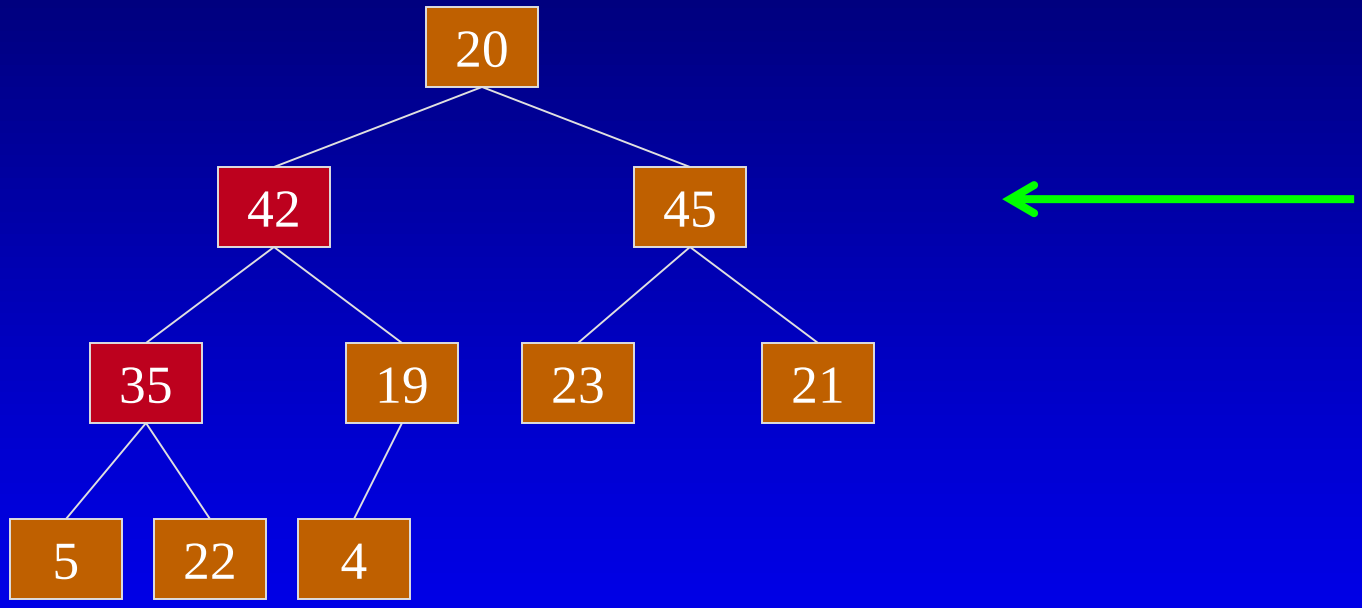
23 is a leaf node, so no reheapification downward is done

Build a heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



35 < 42

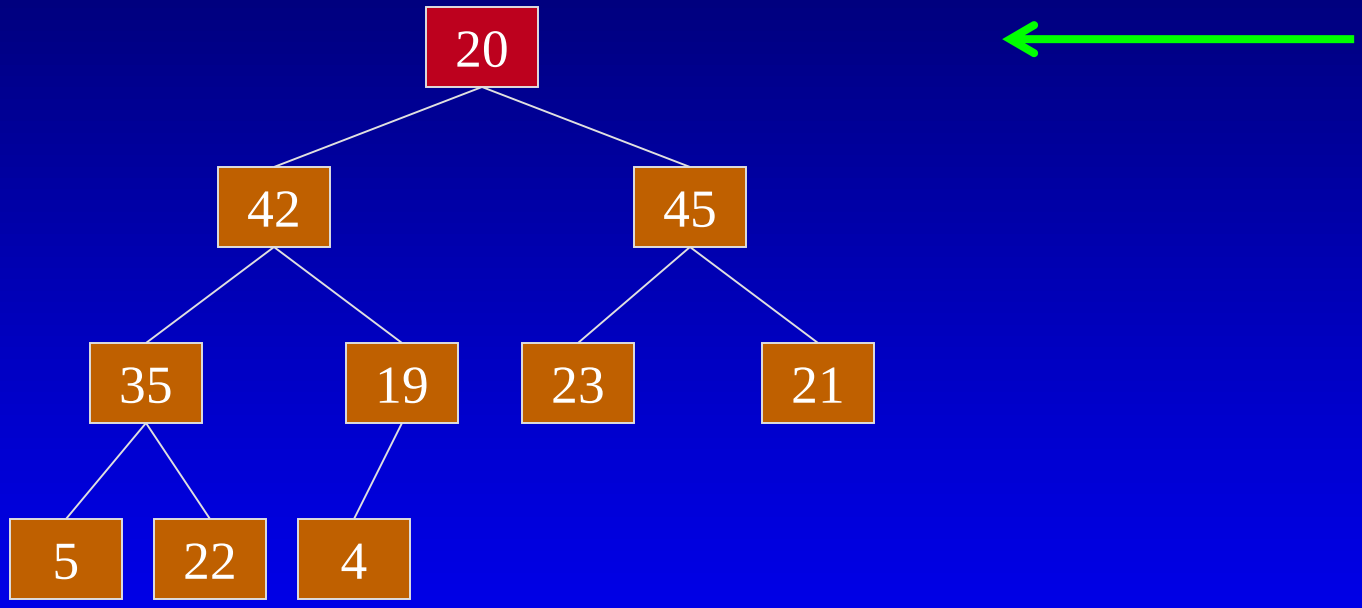
Build a heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



Swap 35 and 42

35 ≥ 5 and 35 ≥ 22, reheapification downward stops

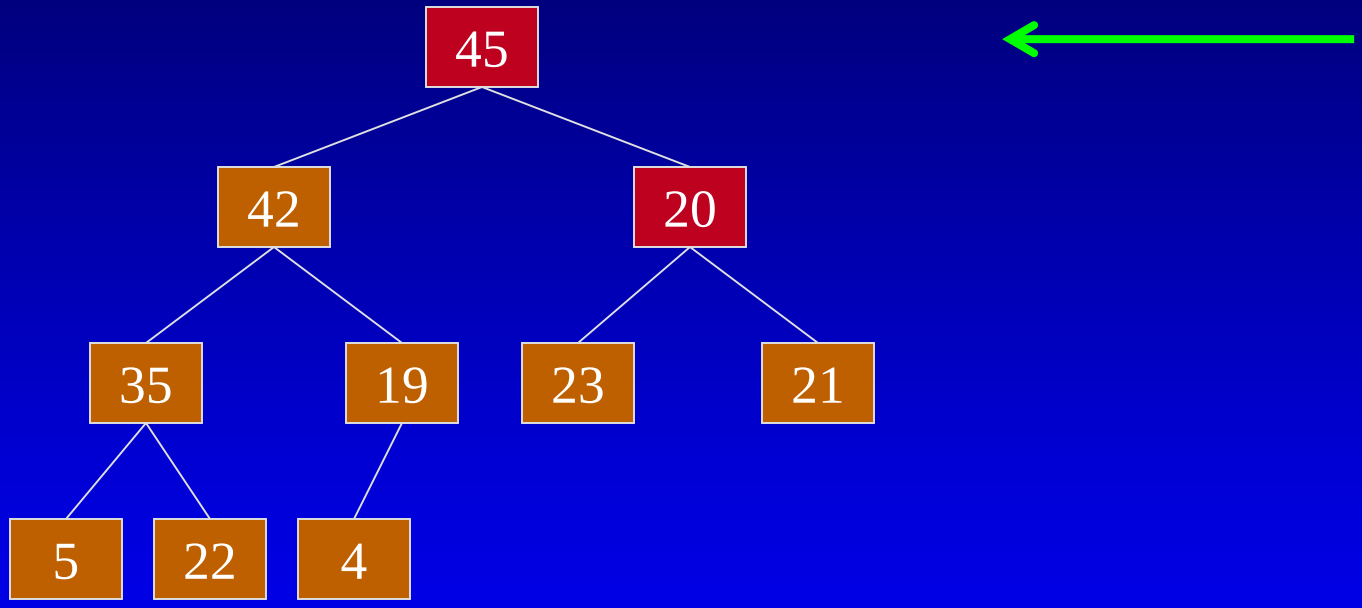
Build a heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



20 < 42 and 20 < 45

Choose the larger child for swapping

Build a heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19

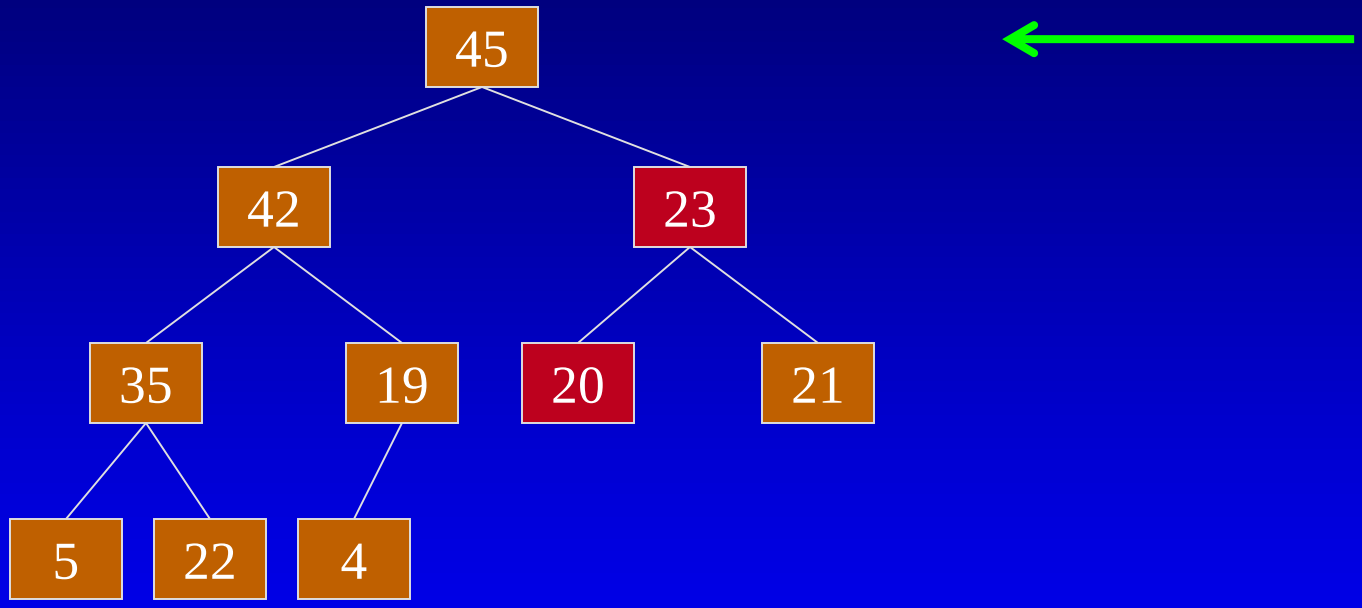


Swap 20 and 45

20 < 23 and 20 < 21, continue reheapificaiton downward

Choose the larger child for swapping

Build a heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



Swap 20 and 23

20 is a leaf node, stop reheapification downward

Build Heap

- Implementation details to think about:
  - How to move from lowest levels to upper?
  - How to ensure that we do not miss any node that has not been processed?

Building a Heap

- Correctness:
- Let *h* be the height of a complete binary tree
- All leaves of a complete binary tree are roots of heaps (each leaf is a root of a subtree of a single node)
- At each upper level *i*, nodes are reheapified downwards of correct heaps, resulting in heaps

Building a Heap

- Time analysis
  - We perform reheapification for each node at heights 0, 1, …, lg *n*
  - Each node is the root of a subtree of height *h*,

  $0 \le h \le \lg n$
  - At height h, there are total of nodes:

$$2^i = 2^{\log(n)-h} = \frac{2^{\log(n)}}{2^h} = \frac{n}{2^h}$$

Building a Heap

- Time analysis
  - There are total of h swaps for each node rooted at a subtree of height h
  - Total number of swaps is the product of the number of nodes at height h and the number of swaps

$$\sum_{h=0}^{h=\log(n)} \frac{n}{2^h} O(h) \leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(2n)$$

$$= O(n)$$

Priority Queue

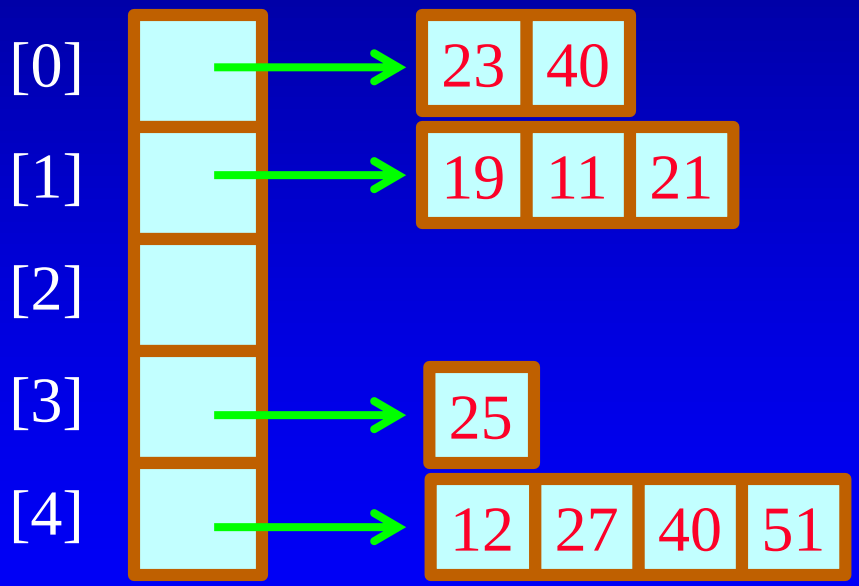
- A priority queue behaves much like an ordinary queue
  - Elements are placed in the queue and later taken out
  - Each element in addition to data has an associated number called **priority**
  - The highest priority element always leaves first

Priority Queue Implementation

- Using a heap
- Using an array of ordinary queues
  - An array index denotes a priority
  - At queues[i], the elements with i-th priority are stored
  - To insert an item of i-th priority, we insert it at the end of the queue at queues[i]
  - To remove an item, we check from highest to smallest priorities, removing an element at the front of a non-empty queue with highest priority

Priority Queue Implementation

- Using an array of ordinary queues



Disadvantage:

Priorities might vary with maximum priority not fixed

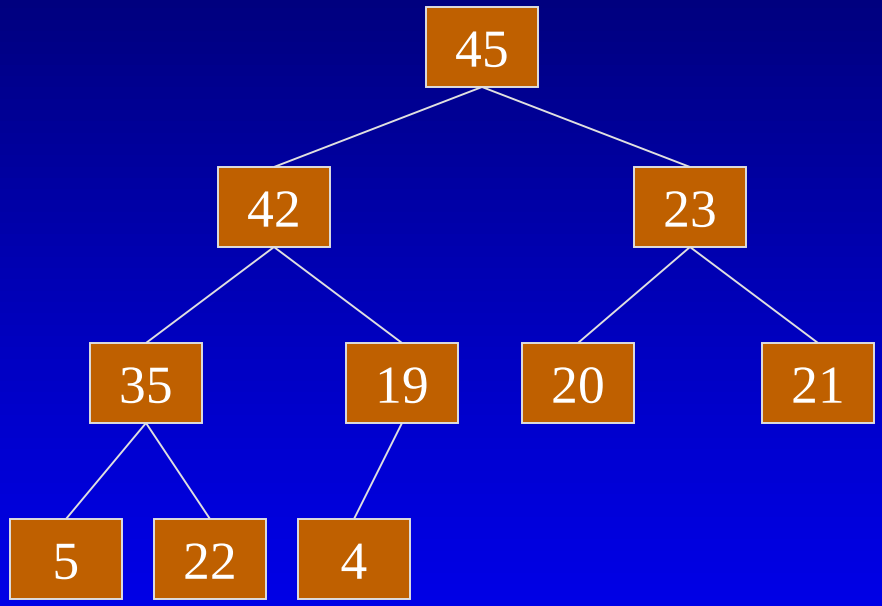
Not efficient in terms of space resources

Priority Queue Implementation

- Using a heap
- Heap is implemented as an array of elements
- Each element has a data item and priority value
- Heap rules are applied using the order according to priority values

Priority Queue using heap

Build a heap with 0:20, 1:35, 2:23, 3:22, 4:4, 5:45, 6:21, 7:5, 8:42 and 9:19 (key : priority)

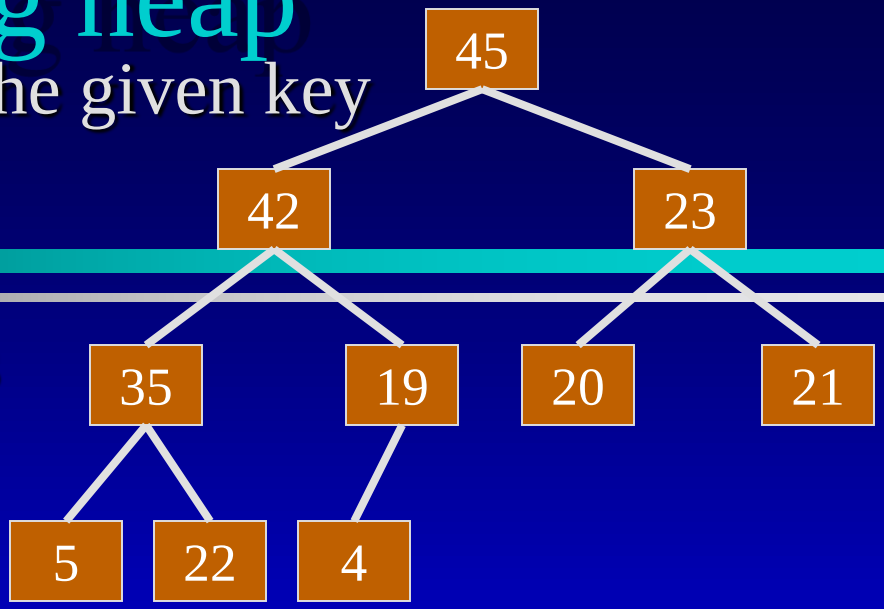


45	42	23	35	19	20	21	5	22	4	priorities
5	8	2	1	9	0	6	7	3	4	keys

Priority Queue using heap

Update priority of an item with the given key

update(key, priority)



Additional array indexed by keys

A[key] = index of key in Heap

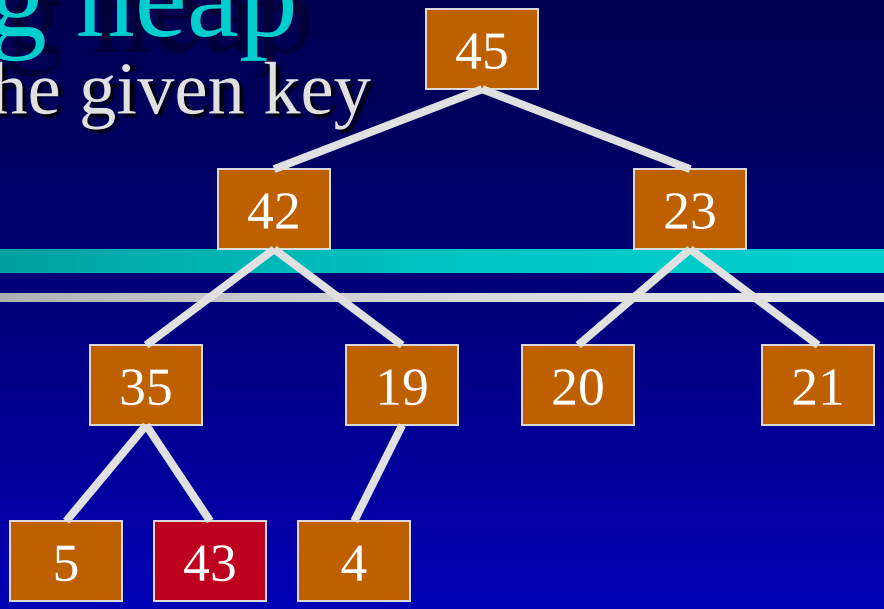
45	42	23	35	19	20	21	5	22	4	priorities
5	8	2	1	9	0	6	7	3	4	keys
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	

5	3	2	8	9	0	6	7	1	4	A
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	

Priority Queue using heap

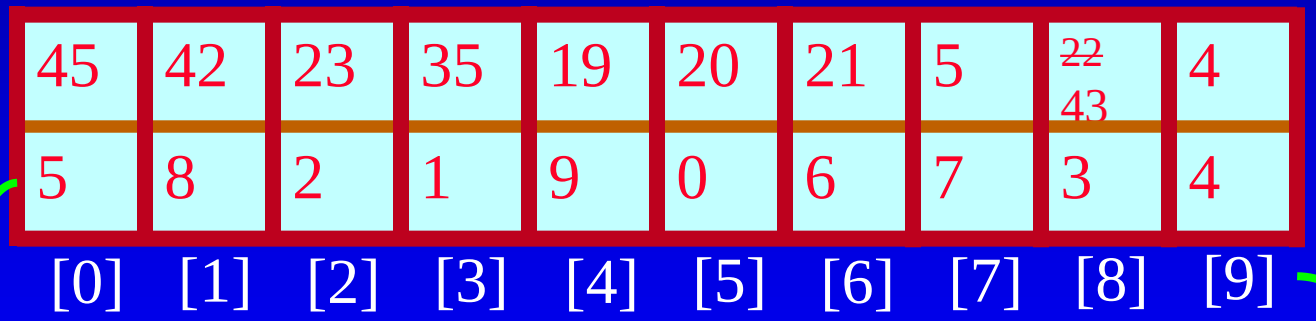
Update priority of an item with the given key

update(key, priority)



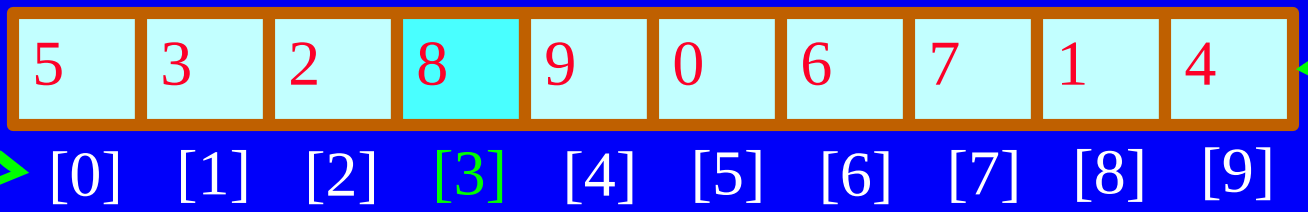
update(3, 43):

1. index ← A[3] //index = 8
2. Heap[index] ← 43
3. Reheapify upward (8, 43)



priorities

keys

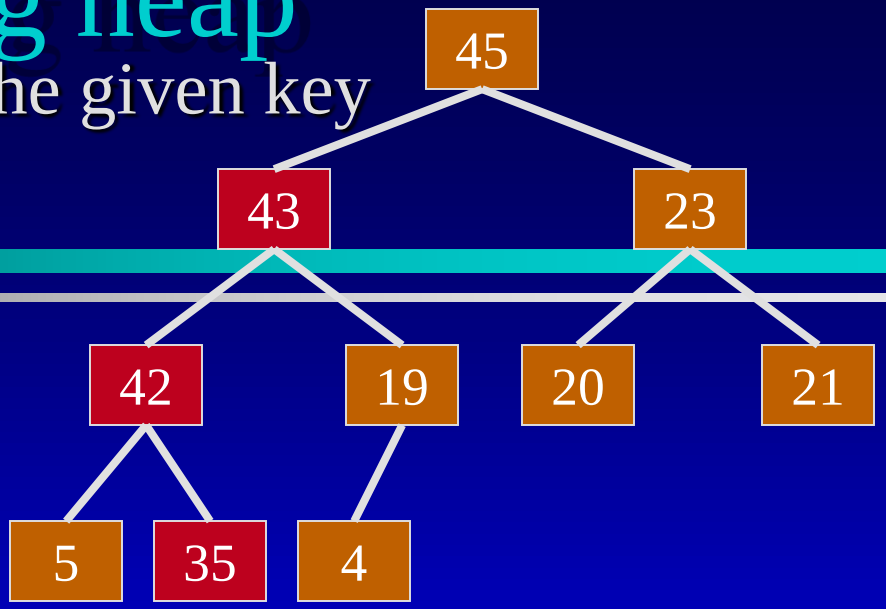


A

Priority Queue using heap

Update priority of an item with the given key

update(key, priority)



update(3, 43):

1. index ← A[3] //index = 8
2. Heap[index] ← 43
3. Reheapify upward (8, 43)

45	43	23	42	19	20	21	5	35	4
5	3	2	8	9	0	6	7	1	4
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

priorities

keys

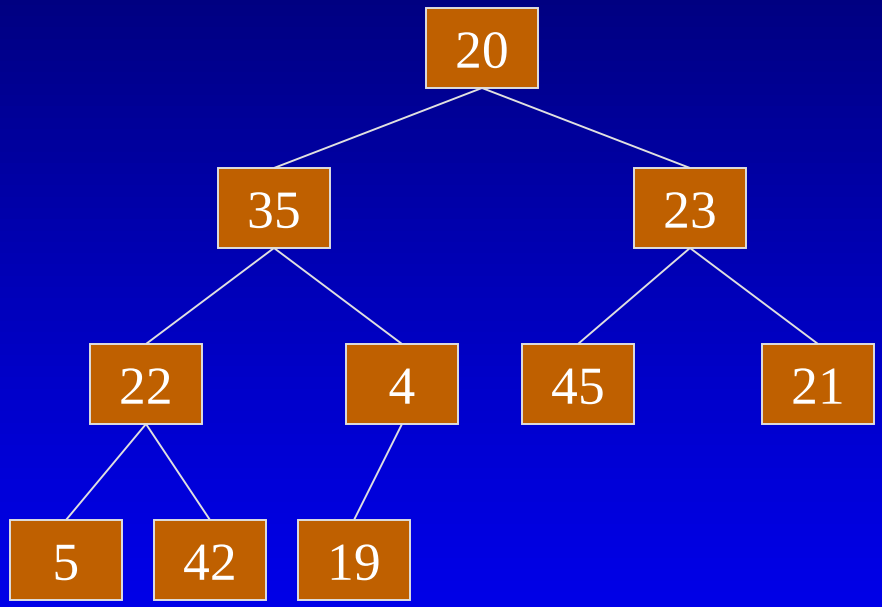
5	8	2	1	9	0	6	7	3	4
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

A

Priority Queue using heap

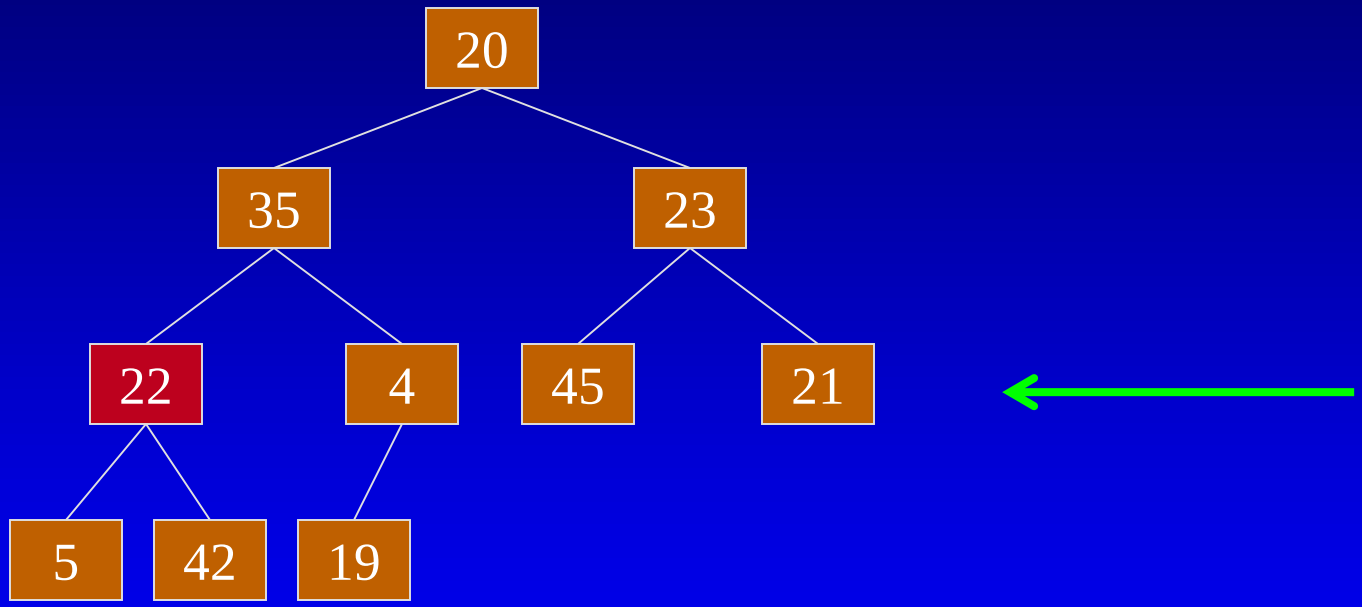
- Time complexity of update(key, priority)
- To find an element with given key takes O(1) at the cost of additional O(n) space
- Alternative is time to find an element is O(n) with no additional cost for space
- Reheapification takes O(log n) time

Build a MIN-heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



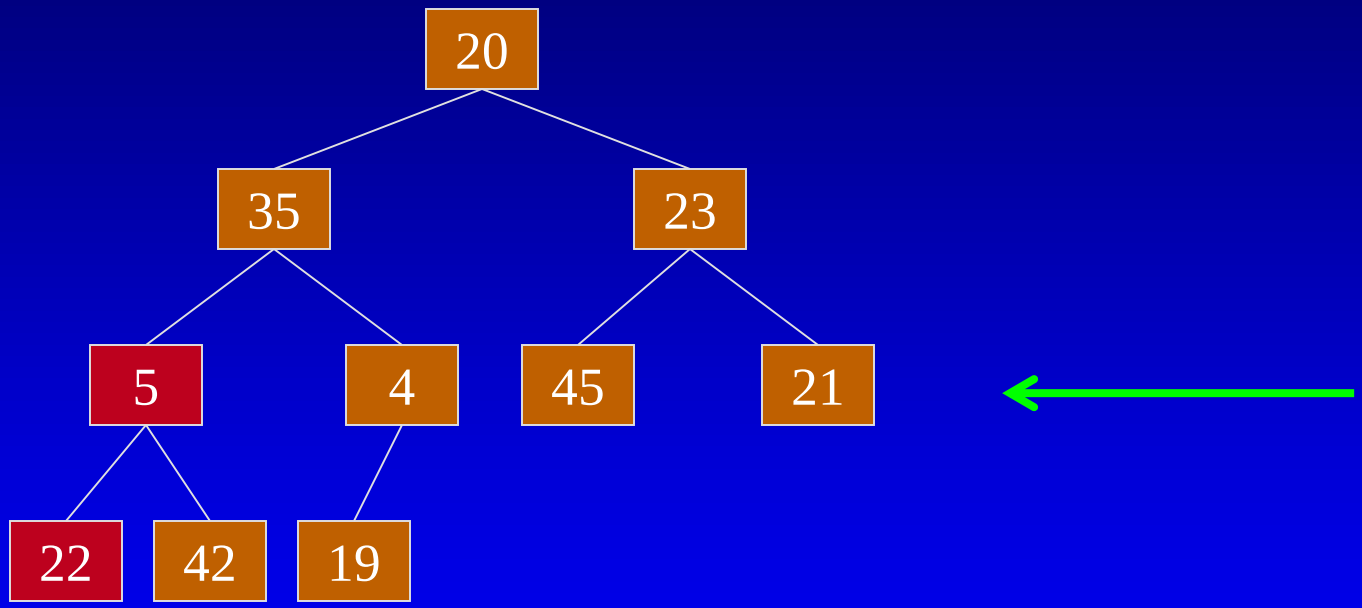
- First, we build an arbitrary complete binary tree.
- Then, starting from the bottom level
  - Reheapify downward each node
  - Swap with the **smaller** child.

Build a MIN-heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



22 > 5

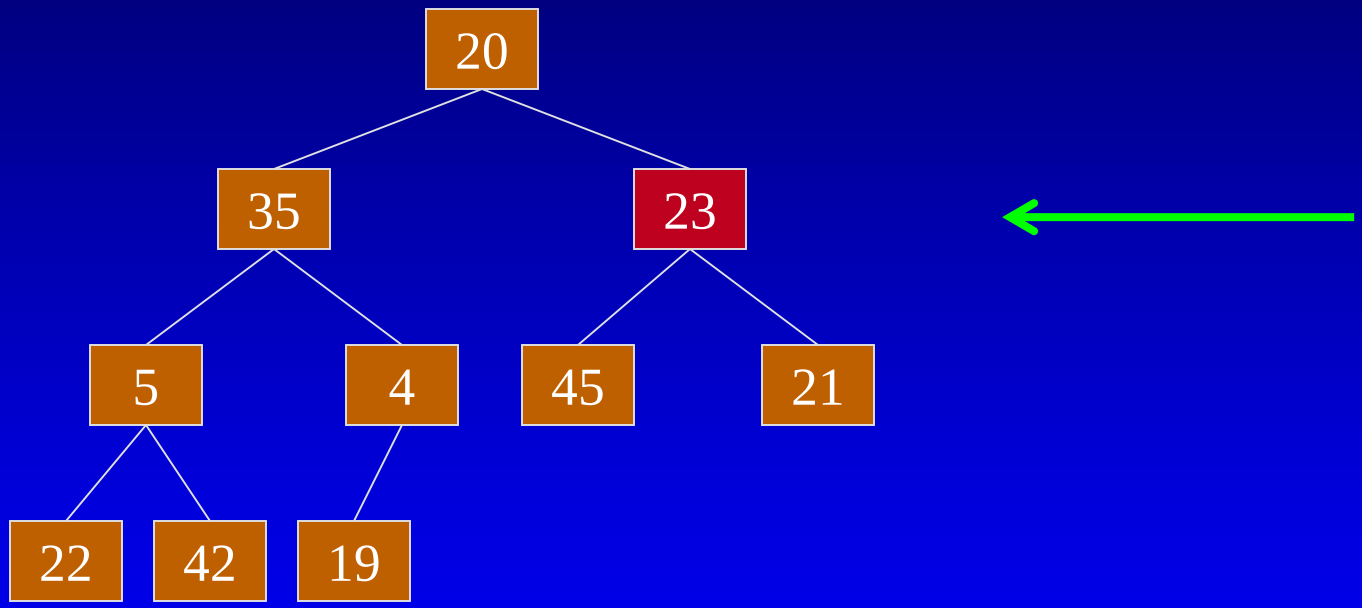
Build a MIN-heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



Swap 22 and 5

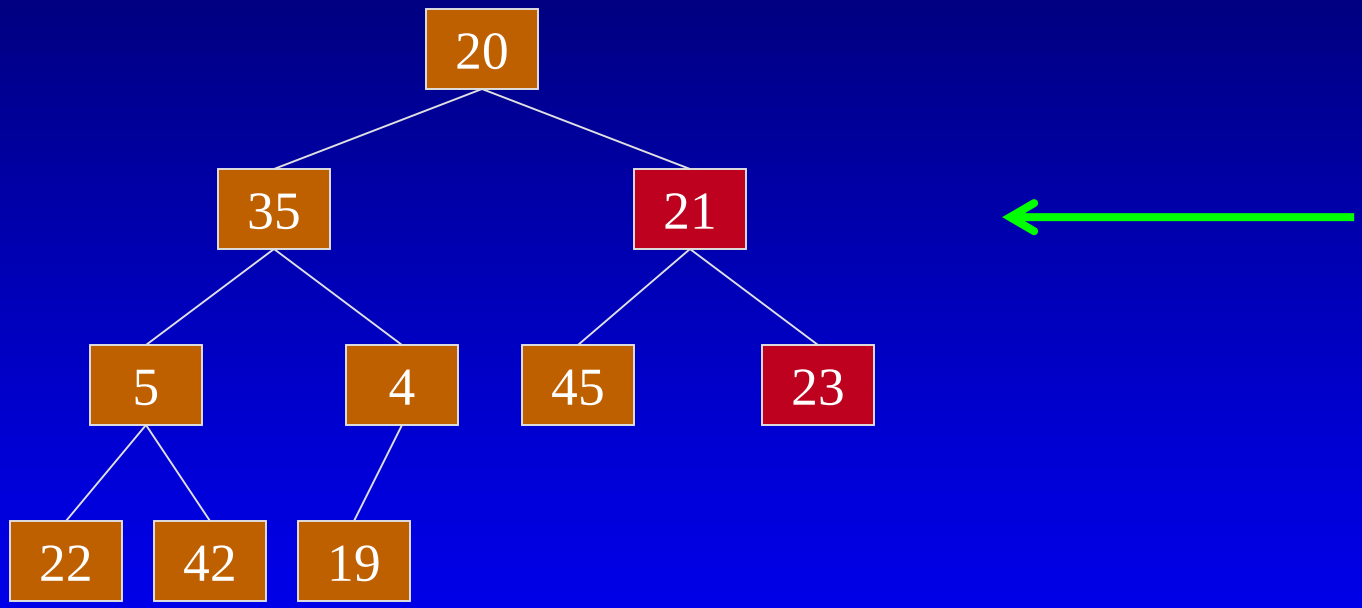
22 is a leaf, stop reheapification

Build a MIN-heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



23 > 21

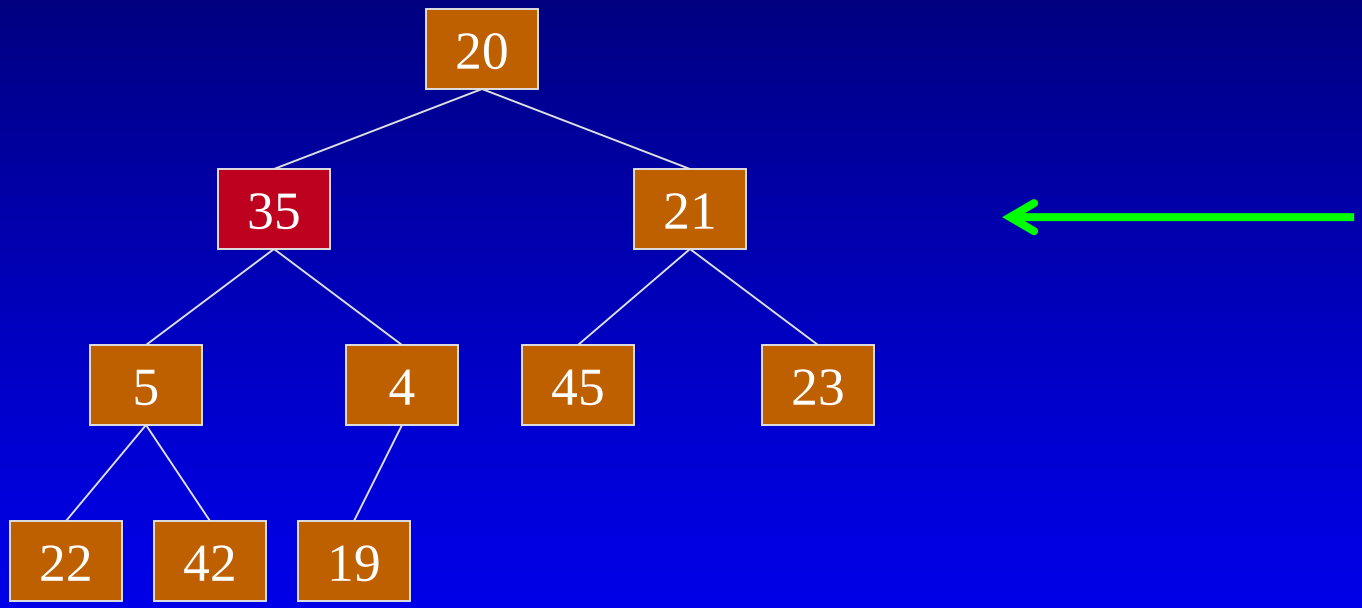
Build a MIN-heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



Swap 23 and 21

23 is a leaf node, stop reheapification

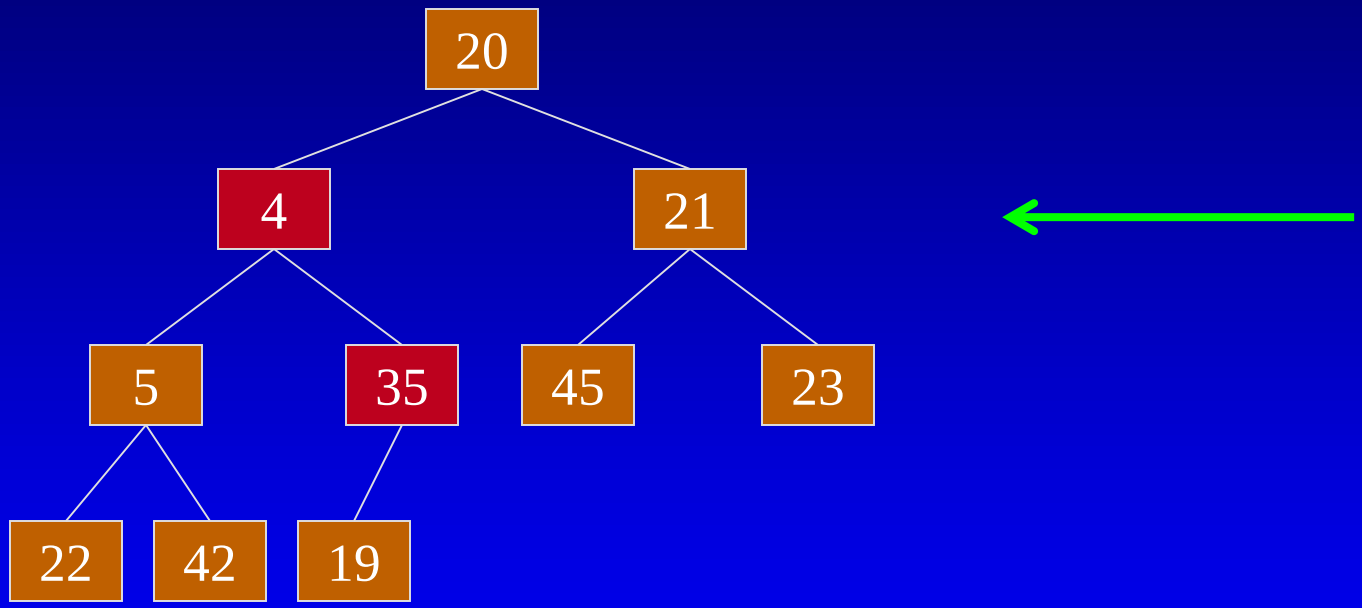
Build a MIN-heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



35 > 5 and 35 > 4

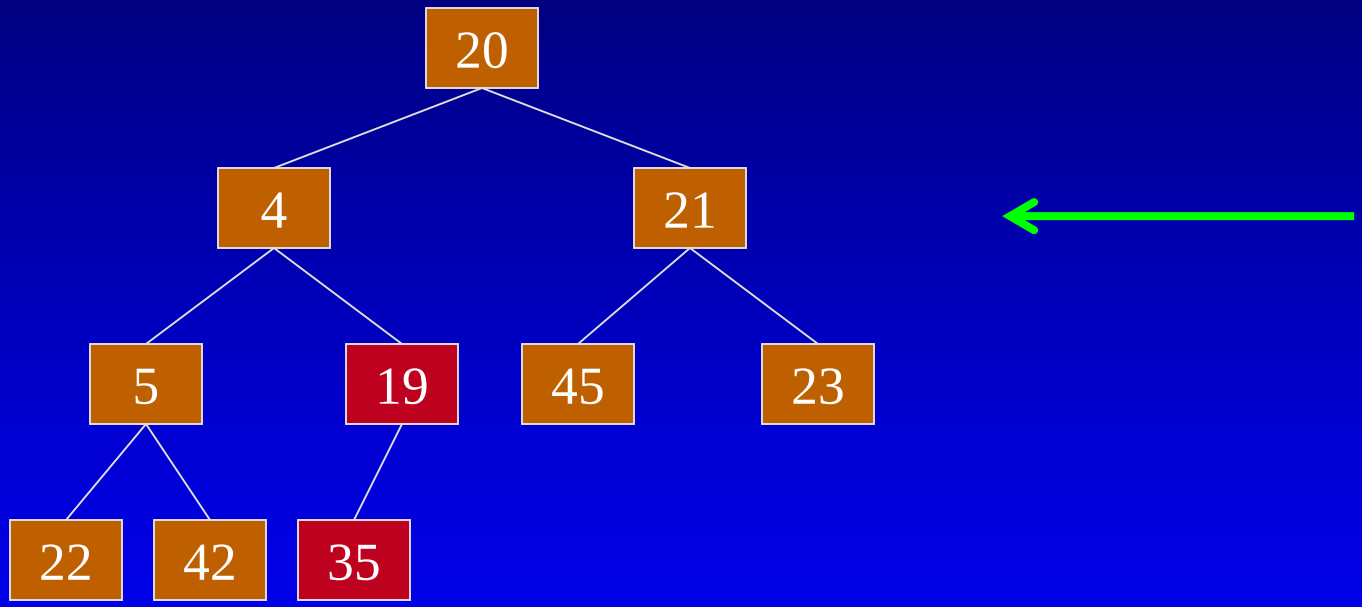
Choose the smaller child, 4

Build a MIN-heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



35 > 19, continue reheapification

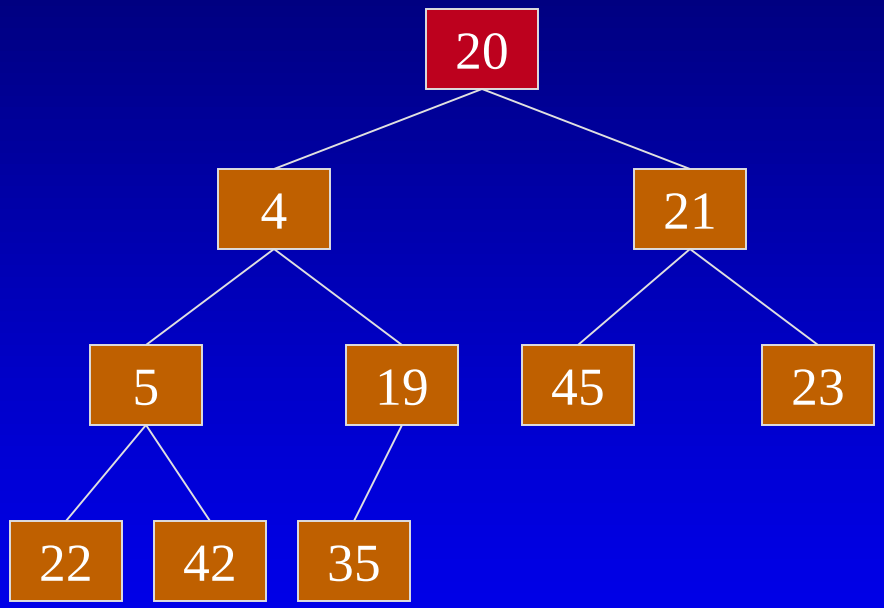
Build a MIN-heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



Swap 35 and 19

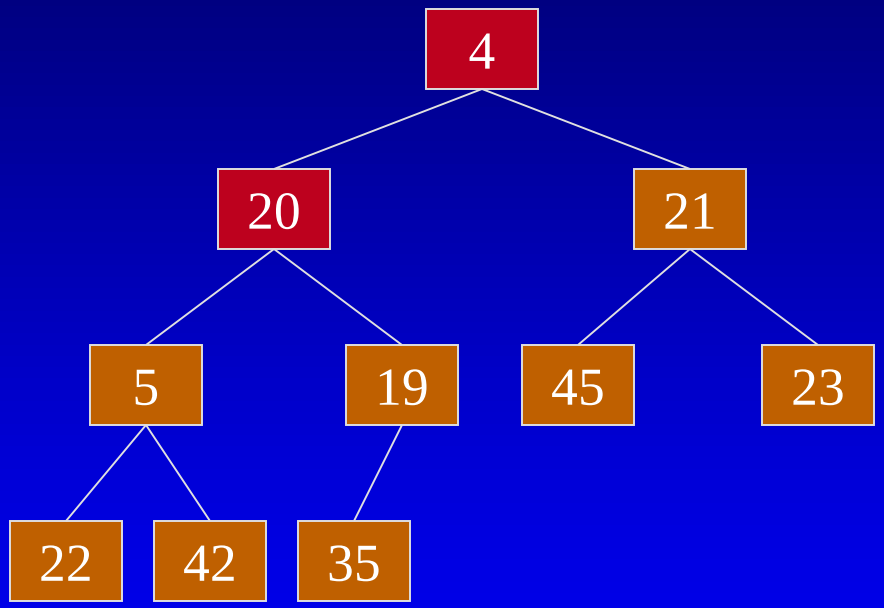
35 is a leaf node, stop reheapification

Build a MIN-heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



20 > 4

Build a MIN-heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19

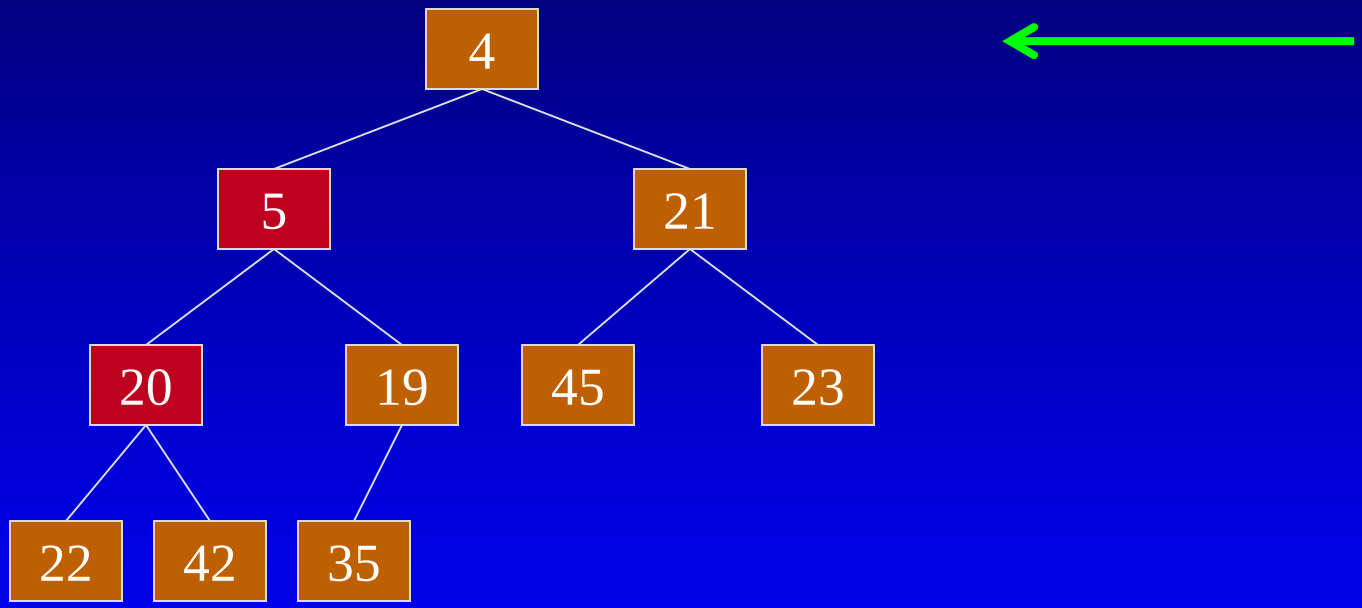


Swap 20 and 4

20 > 5 and 20 > 19

Choose the smaller child, 5

Build a MIN-heap with 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19



Swap 20 and 5

20 < 22 and 20 < 42, stop reheapification

Note, the resulting heap is a MIN-heap

Implementation details

```
class Element
{
    int key;
    int priority;
    Node(int akey, int apriority){
        key = akey;
        priority = apriority;
    }
    int getKey(){ return key; }
    int getPriority(){ return priority; }
    void setPriority( int anew ){ priority = anew; }
}
```

Implementation details

```
public class Heap{
    private Element aheap[];
    public int size;
    private int capacity;
     public Heap(){ size = 0; capacity = 10; }
    public void build();//using input from stdin
    public void upHeapify(int i);
     public void downHeapify(int i);
     public void add(Element a);
     public void remove();
     public bool update( int key, int priority );
}
```

Implementation details

```
public void build (Element array[]){
    size = array.length;
    capacity = size * 2;
    aheap = new Element[ capacity ];
    copy array into aheap
    for i = size/2 down to 0 do
        downHeapify( i )
}
```

Implementation details

```
public void downHeapify( int i ){
   int left = 2*i + 1; //index of the left child
   int right = 2*i + 2; //index of the right child
   int amin = i;
   if( left >= size && right >= size)
         return;// the node is a leaf
   if ( left < size && aheap[ left ].getPriority() < aheap[i].getPriority()){
         amin = left;
   }
   if(right < size &&
                  aheap[ right ].getPriority() < aheap[amin].getPriority() )
         amin = right; ……
}
```

Implementation details

```
public void downHeapify( int i ){
    ……
    if ( amin != i )
        swap aheap[amin] and aheap[i]
        downHeapify( amin );
}
```

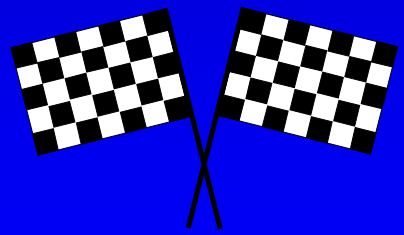
upHeapify is similar except that we swap with the parent of i:

int parent = (i – 1) / 2

You will need to modify upHeapify and downHeapify to accommodate update operation (need to change values in the additional array *indices* for finding the index of an element with the given key in O(1) time)

Summary

- A heap is a complete binary tree, where the entry at each node is greater 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.



THE END

Objectives

- List the rules for a heap and determine whether a tree satisfies these rules
- Insert a new element into a heap or remove the root from a heap
- Use the heap data structure to implement a priority queue

Prove

- Show that in any subtree of a max-heap, the root of the subtree contains a largest key occurring at any node in that subtree

Show that in any subtree of a max-heap, the root of the subtree contains a largest key occurring at any node in that subtree

- Proof: Assume that m is a key in the subtree rooted at k and m > k and assume that m is the closest such node in this subtree
- There exists a path from k to m in this subtree
- Since m is the first node such that m > k,
- Parent(m) on this path is less than or equal to k
- Parent(m) ≥ m (by the heap property)
- k ≥ Parent(m) ≥ m > k (contradiction)