Heap Tree, Heap Sort, Priority Queue, Huffman Tree

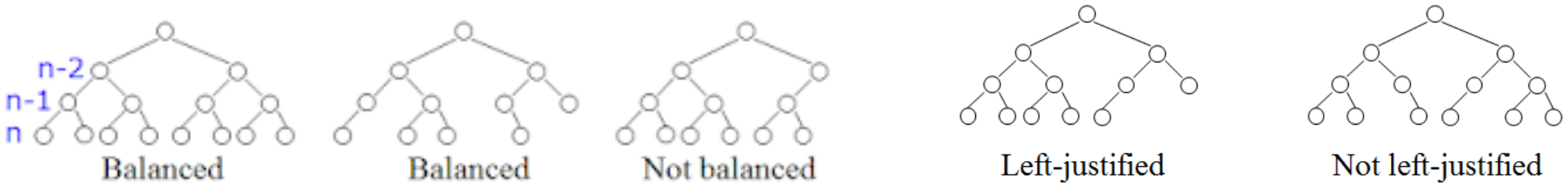
Heap Tree

- Heap tree is a complete binary tree : balanced, left justified
  - A binary tree of depth n is balanced if :

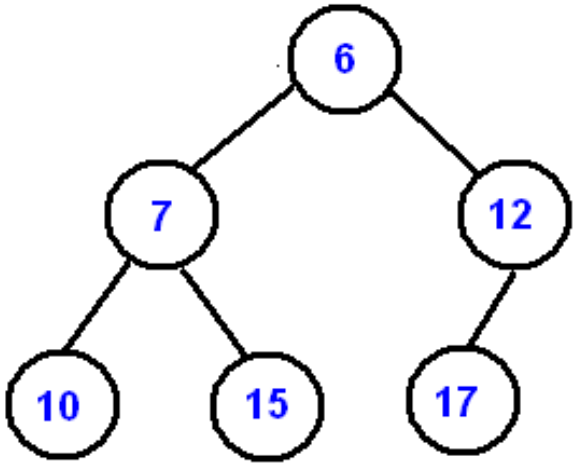
    all the nodes at depths 0 through n-2 have two children
  - A balanced binary tree of depth n is left-justified if:

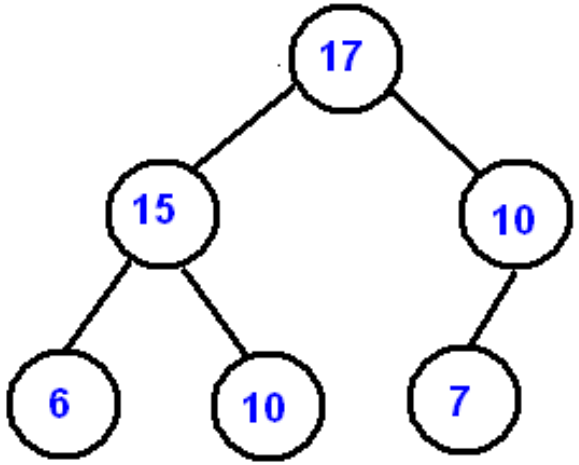
    All the leaves are the same depth, or

    All the leaves at depth **n** are to the left of all the nodes at depth **n-1**



Max-heap vs. Min-Heap





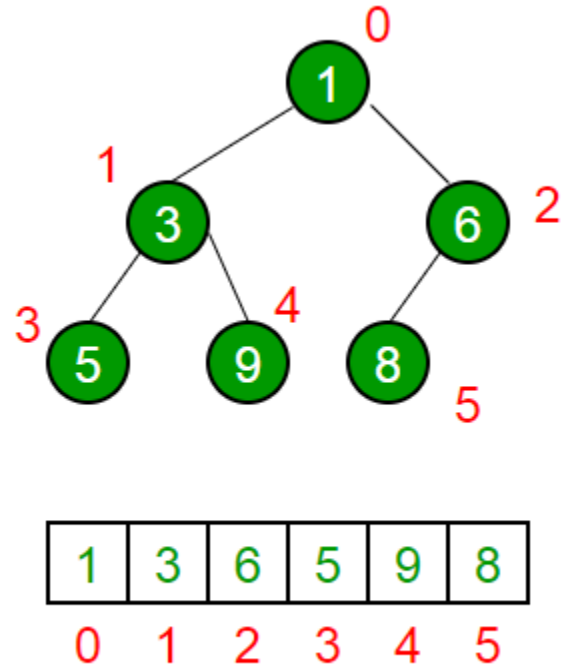
Min-heap : A[parent] <= A[child]

Max-heap : A[parent] >= A[child]

Array Representation of Heap Tree

- The root element will be at A[0].
- For the ith node A[i]:

A[(i-1)/2]	the parent node
A[2*i+1]	the left child node
A[2*i+2]	the right child node



Applications

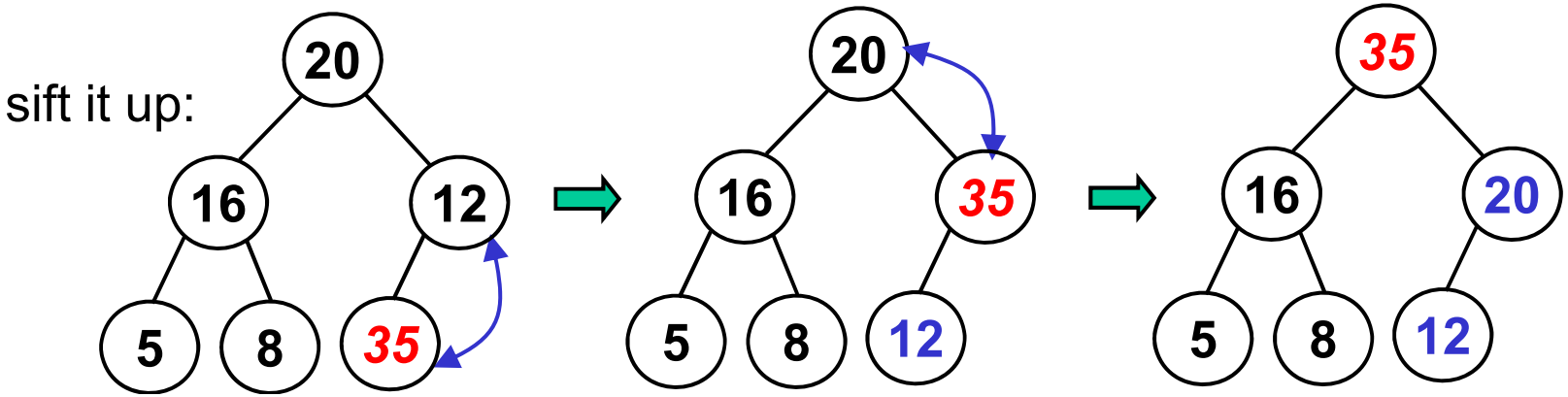
- Many problems can be efficiently solved using Heap Tree, such as:
  - Sorting an array with O(n*log n) complexity
  - Implementing priority queue
  - Finding kth largest in an array
  - Merging k sorted array

Inserting a new item in a heap tree :

1. put the item in the next available node
2. sift up the newItem until :

   newItem <= its parent (or it becomes the root item)

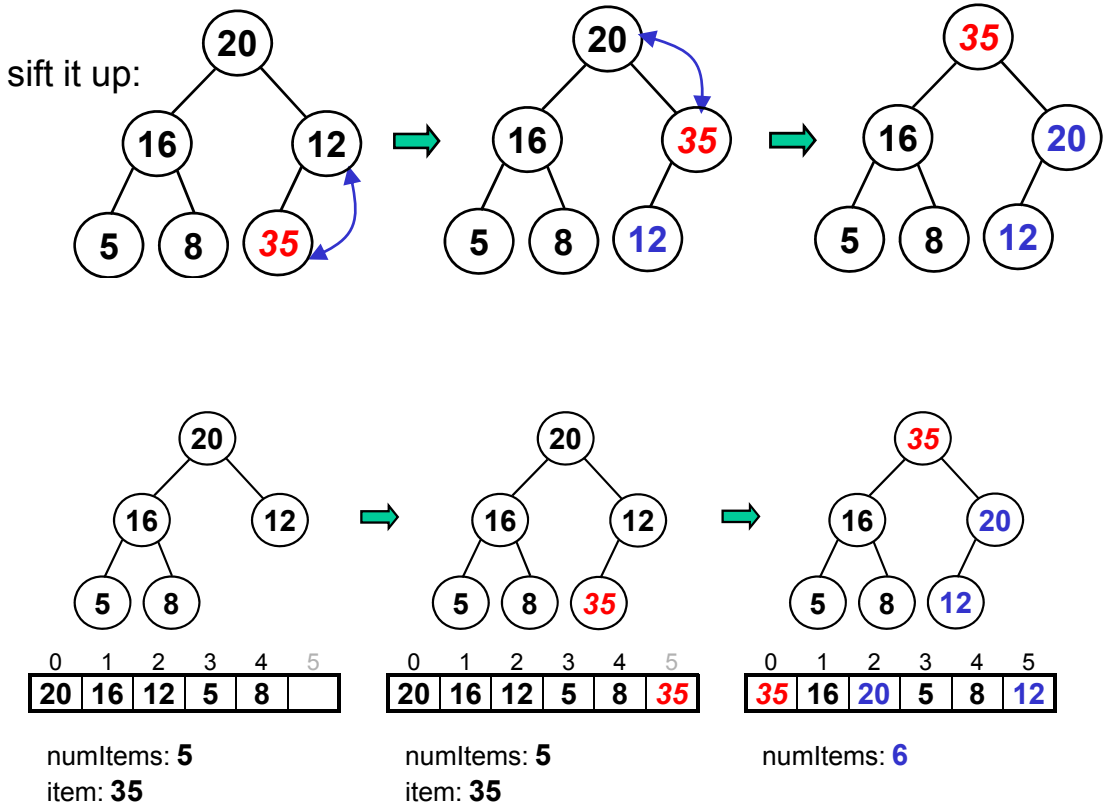
Example : insert 35



Inserting an item in a heap tree :

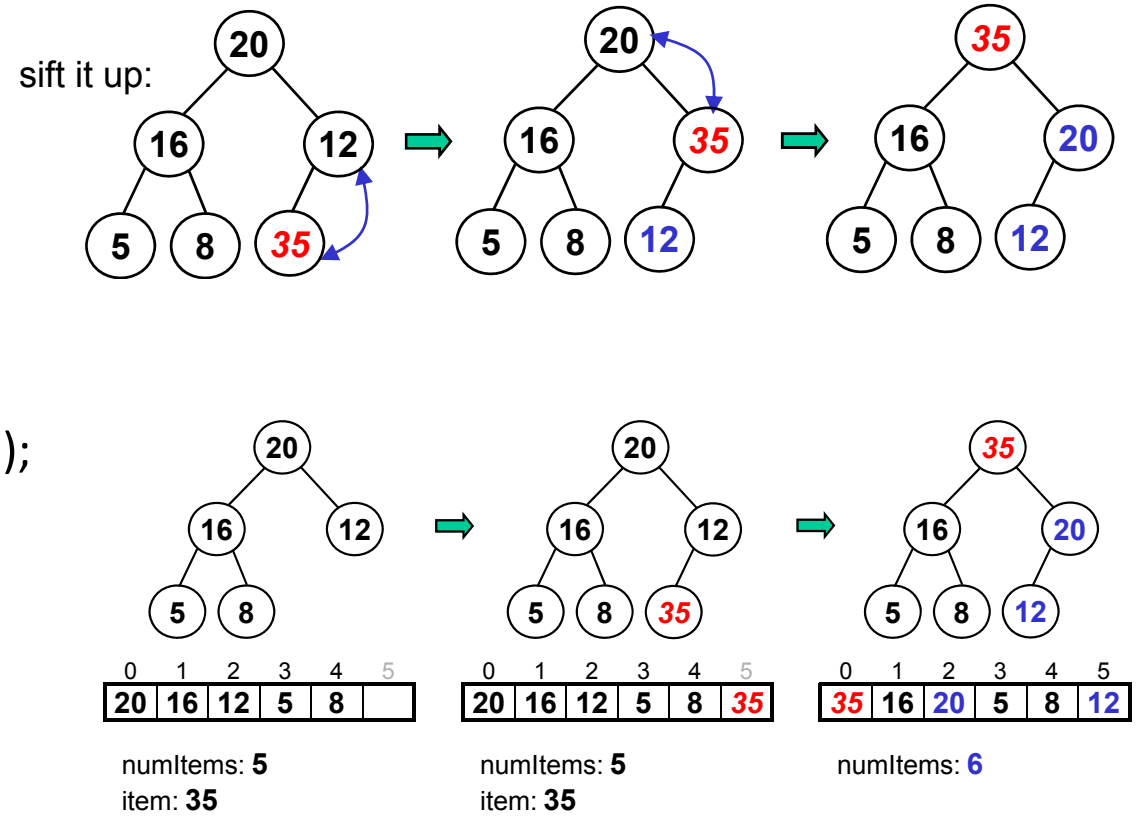
void insert(int item, int Heap[], int *n)

```
{
        Heap[*n] = item;
        siftUp(item,Heap,*n);
        (*n)++;
}
```



Inserting an item in a heap tree :

```
void siftUp(int item, int Heap[], int adr)
{
        int  stop = 0;
        while((adr>0) &&  (!stop))
        {
          parent = (adr-1) / 2;
          if(Heap[parent]  < item)
          {
                  swap(&Heap[adr],&Heap[parent]);
                   adr = parent ;
          }
          else
                  stop = 1;
           }
}
```



Removing the root node

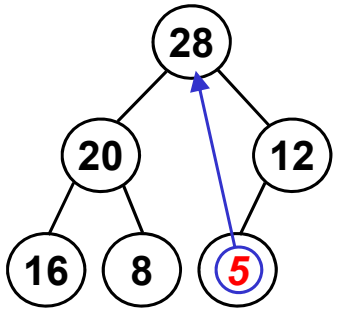
1.Make a copy of root node

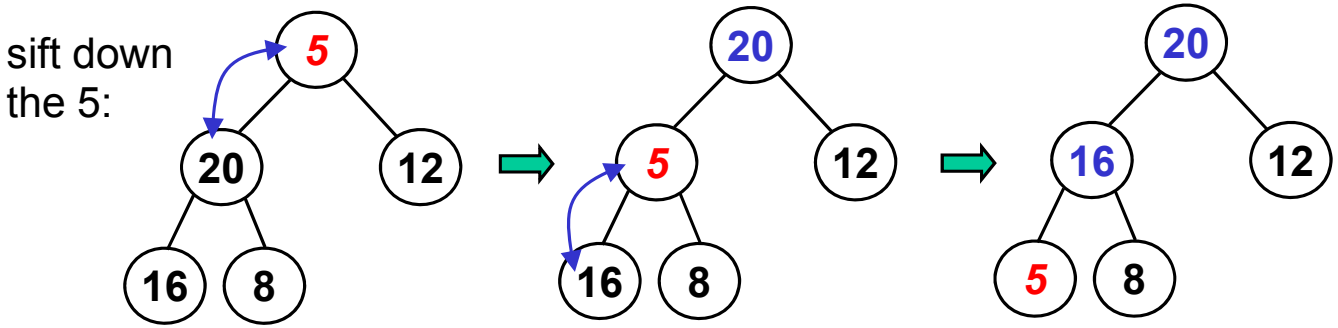
2. move the last item in the heap to the root

3. sift down the new root item until :

   it is >= its children (or it's a leaf)

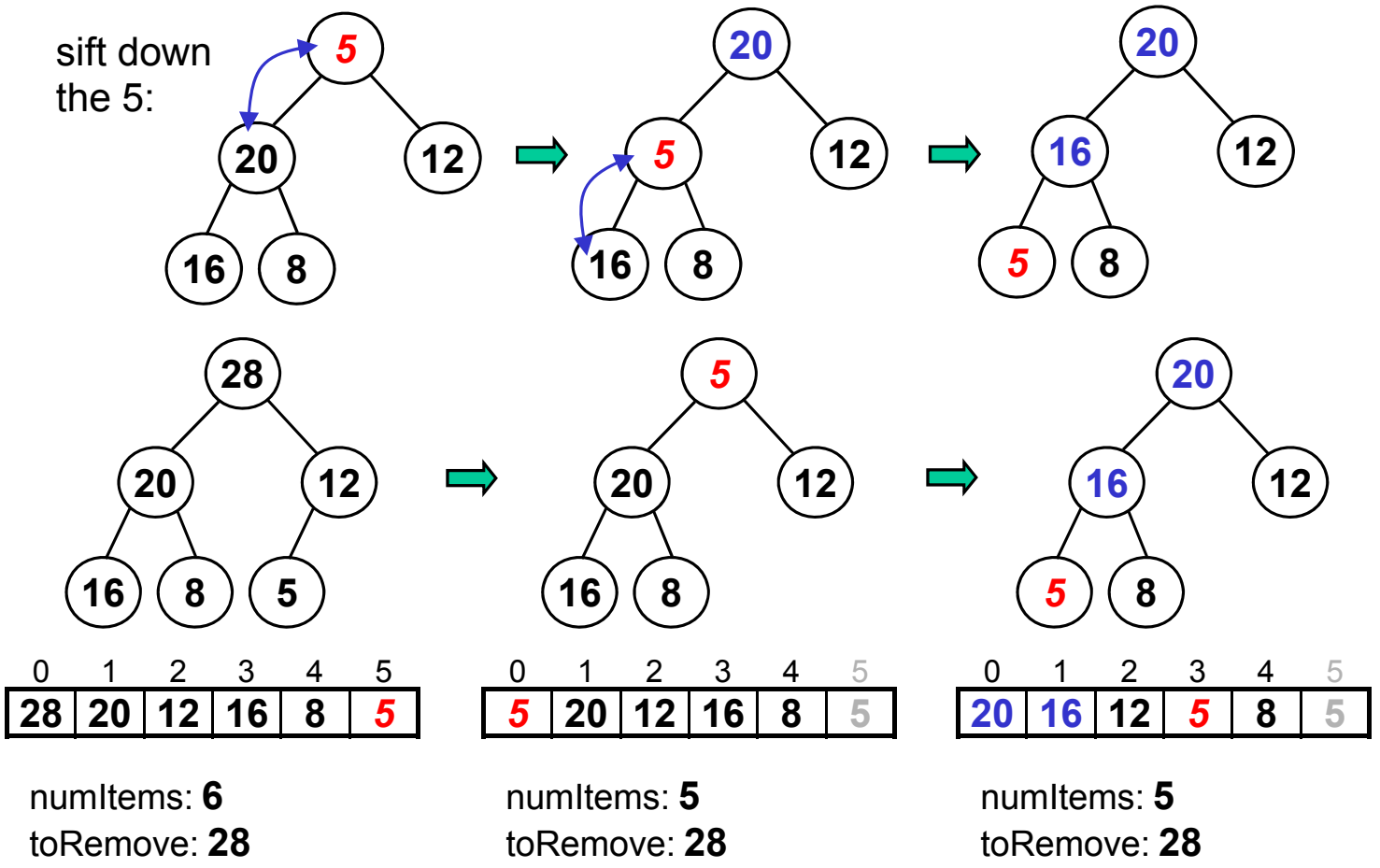
4. return the copy of old root node





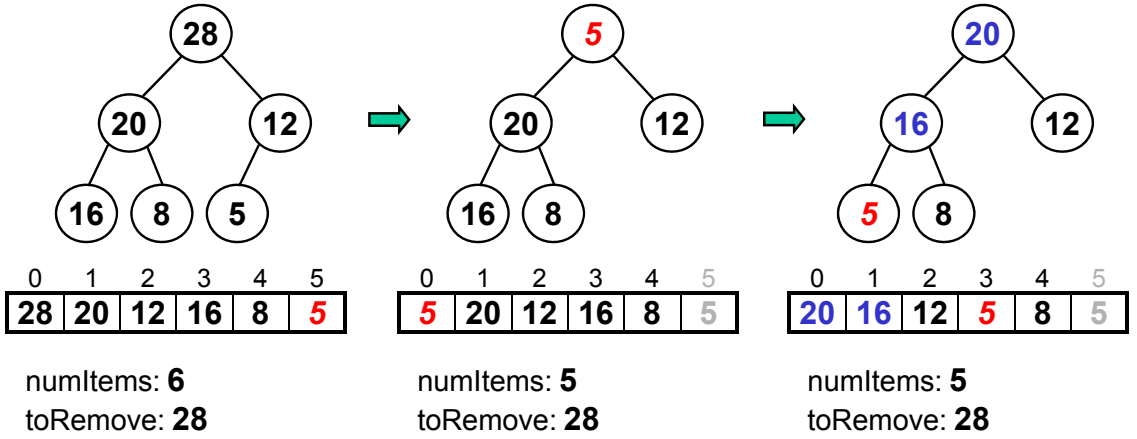
Removing the root node

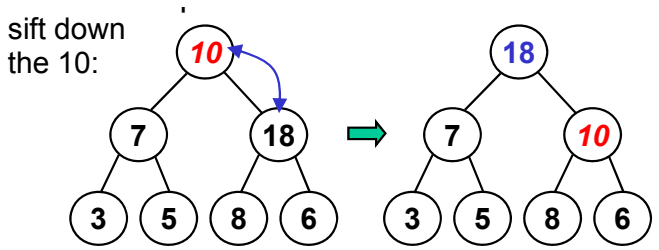
```
int removeMax(int Heap[], int *n)
{
    int retValue = Heap[0];
    Heap[0] = Heap[*(n-1)];
    (*n)--;
    siftDown(Heap, *n);
    return retValue;
}
```

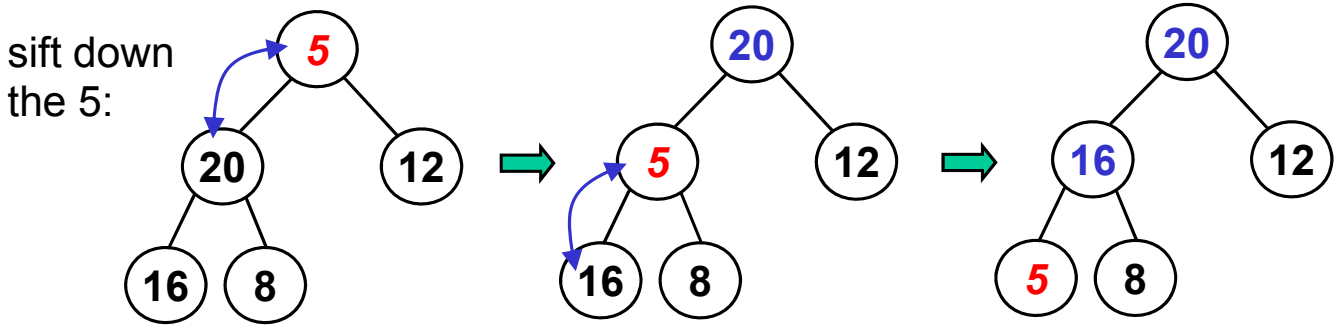


Removing the root node

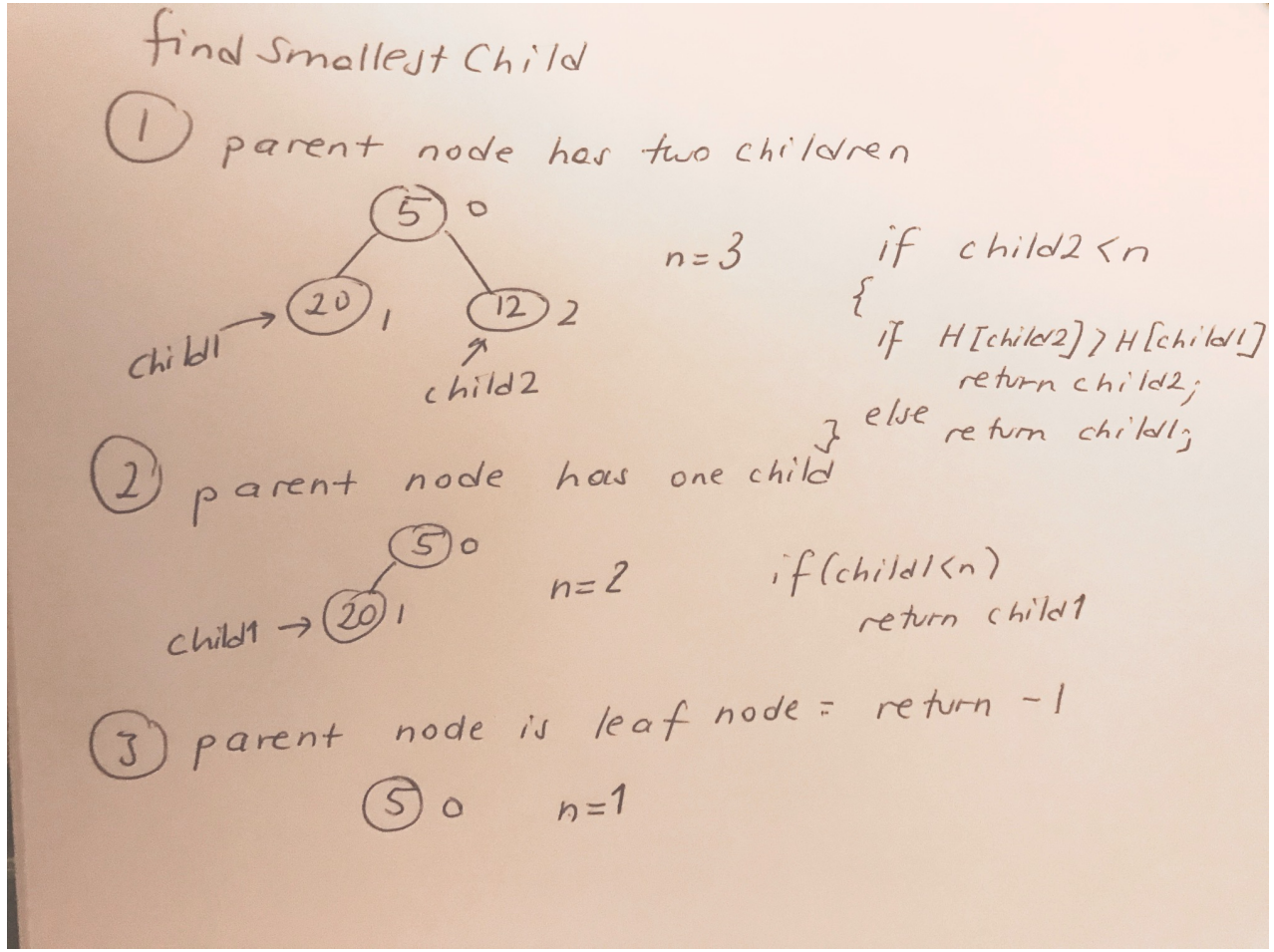
```
int siftDown(int Heap[], int n)
{
        int parent  = 0, child;
        child = findSmalestChild(Heap, parent,n));
        while((child !=-1) && (Heap[child] > Heap[parent]))
        {
                swap(&Heap[child],&Heap[parent]);
                 parent = child;
                 child = findSmallestChild(Heap, parent,n);
         }
 }
```





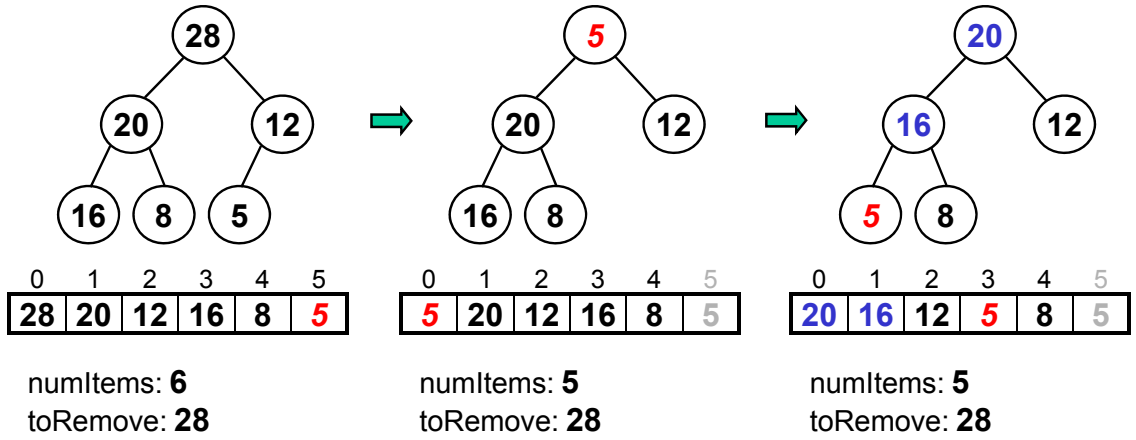


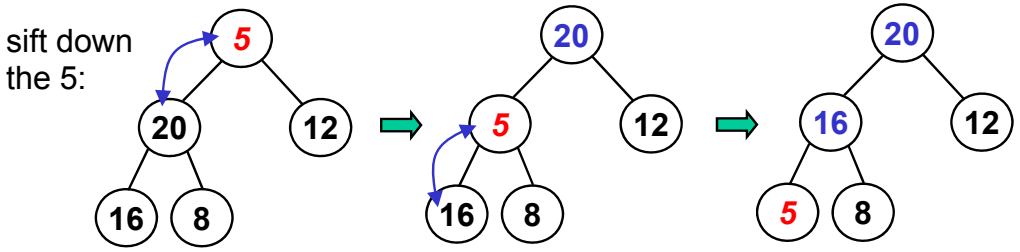
findSmallestChild function



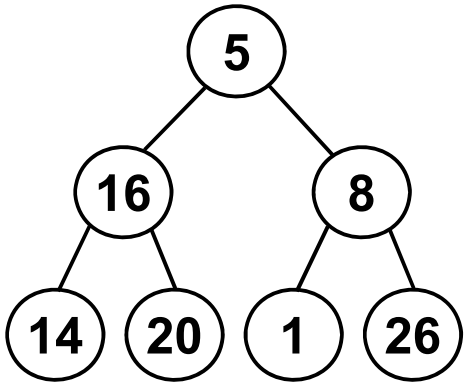
Removing the root node

```
int findSmallestChild(int Heap[], int parent, int n)
{
        int child1 = 2*parent +1 , child2 = 2*parent + 2;
        if(child2 < n) // parent node has two children
        {
            if(Heap[child2] > Heap[child1])
                return child2;
            else return child1;
        }
        else if(child1 < n)   // parent node has one child
             return child1;
            else return -1 ;    // parent node is a leaf
}
```





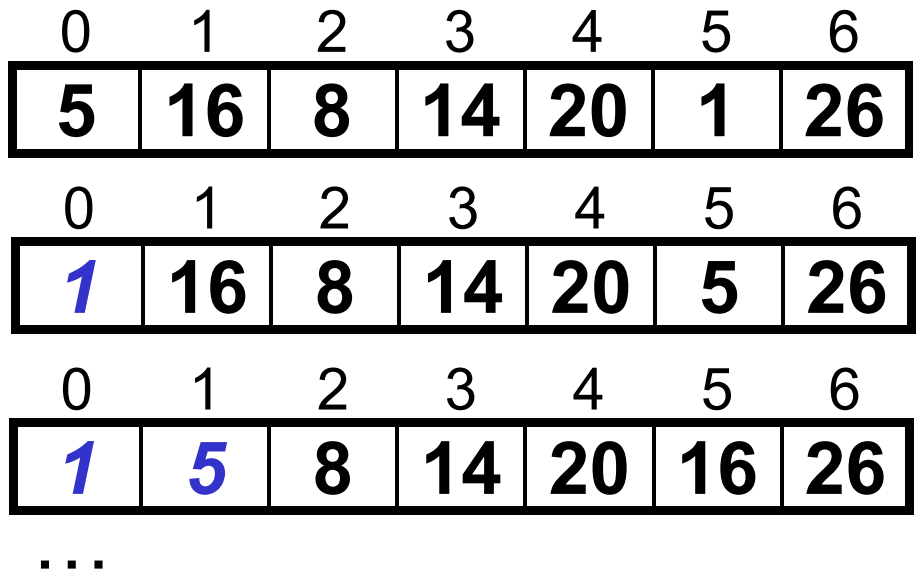
Time Complexity of a Heap



- A heap containing n items has a height <= $\log_2 n$.
- Thus, removal and insertion are both $O(\log n)$.
  - remove: go down at most $\log_2 n$ levels when sifting down from the root, and do a constant number of operations per level
  - insert: go up at most $\log_2 n$ levels when sifting up to the root, and do a constant number of operations per level

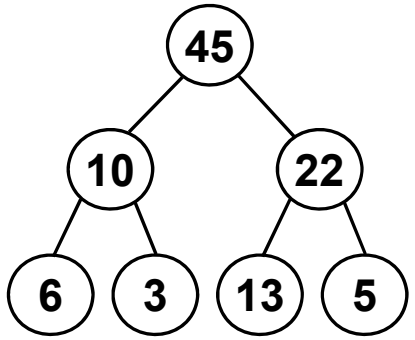
Heap Sort

- Recall selection sort: it repeatedly finds the smallest remaining element and swaps it into place
- Selection sort isn't efficient ($O(n^2)$), because it performs a linear scan to find the smallest remaining element ($O(n)$ steps per scan).



Heap Sort

- Heapsort is a sorting algorithm that repeatedly finds the *largest* remaining element and puts it in place.
- Heapsort *is* efficient (*O*(n * log n)), because it turns the array into a heap, which means that it can find and remove the largest remaining element in *O*(log n) steps.

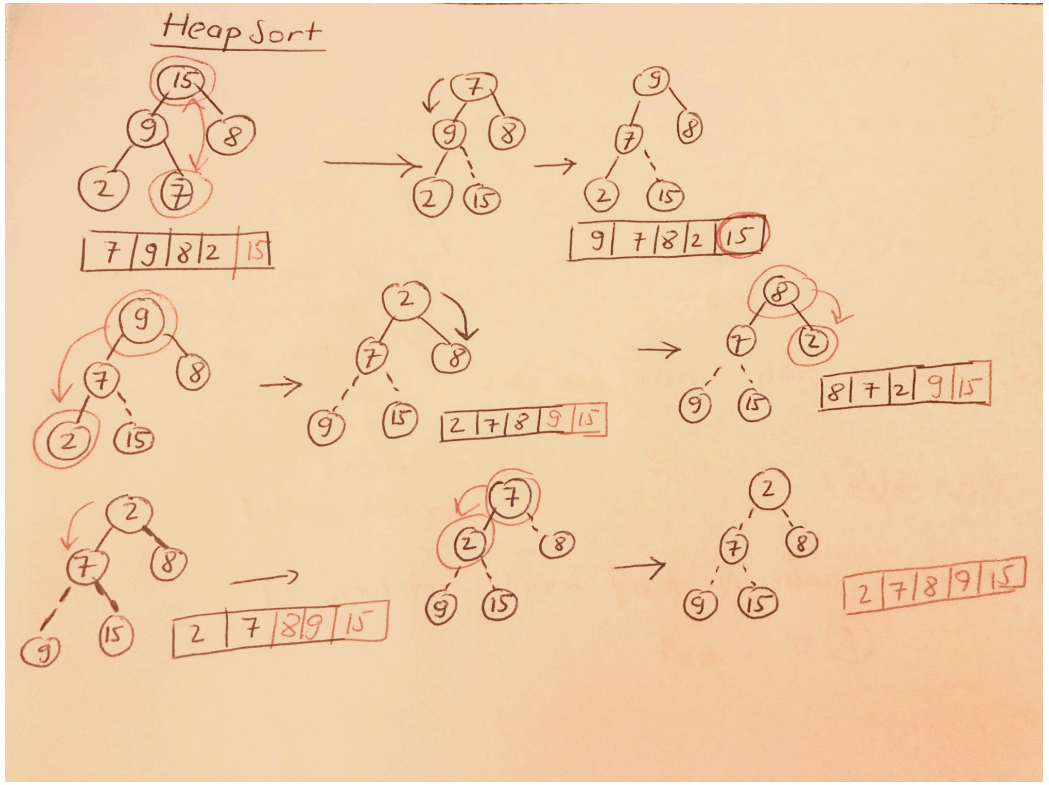


0	1	2	3	4	5	6
45	**10**	**22**	**6**	**3**	**13**	**5**

endUnsorted: **6**

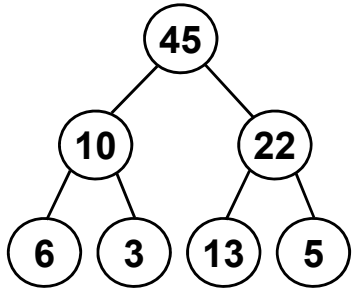
Heap Sort

```
void HeapSort(int Heap[], int n)
{
        for(i=n-1; i>0; i--)
        {
                swap(&Heap[0],&Heap[i]);
                siftDown(Heap, i);
        }
}
```



Heap Sort Example

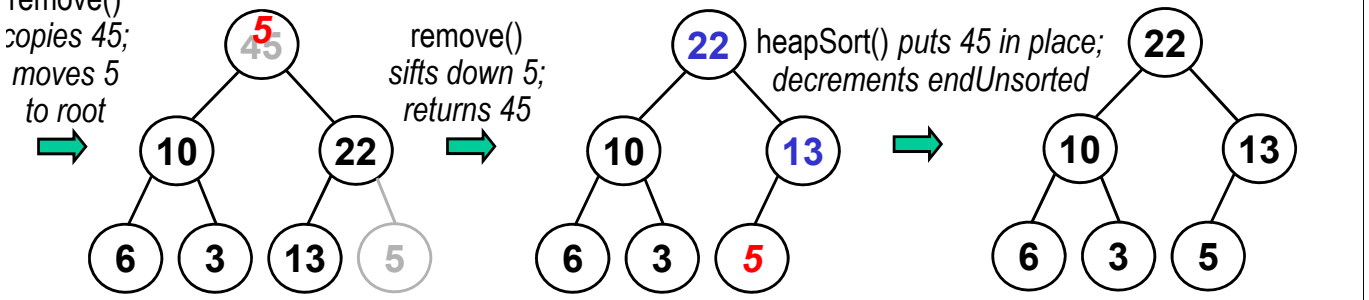
- Here's the heap in both tree and array forms:



0	1	2	3	4	5	6
45	10	22	6	3	13	5

endUnsorted: **6**

- Remove the largest item and put it in place:



toRemove: **45**

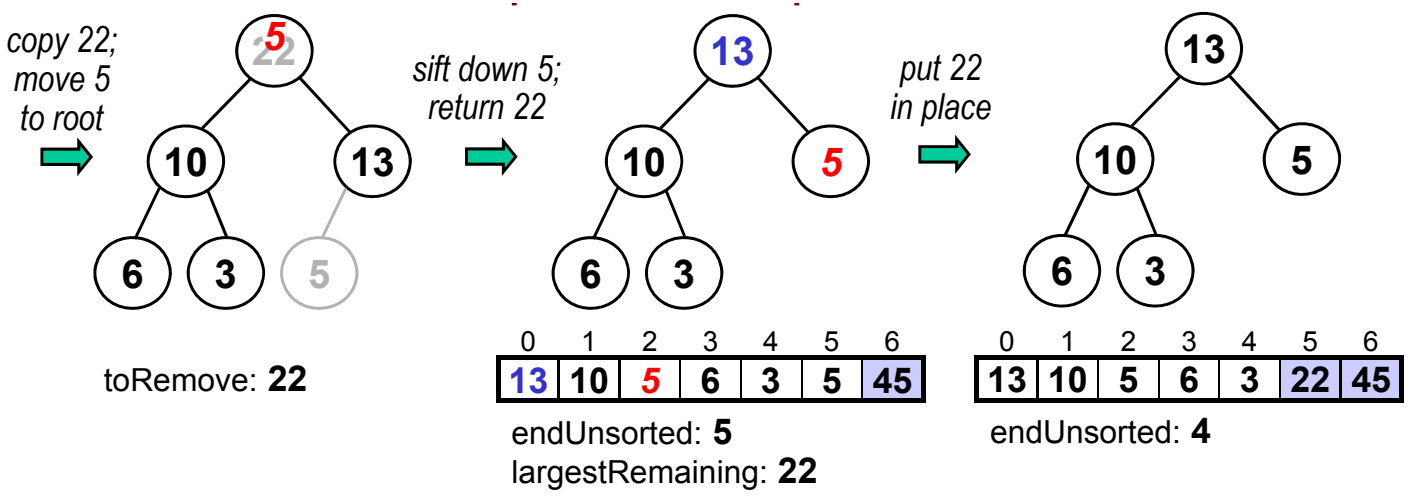
0	1	2	3	4	5	6
22	10	13	6	3	5	5

endUnsorted: **6**
largestRemaining: **45**

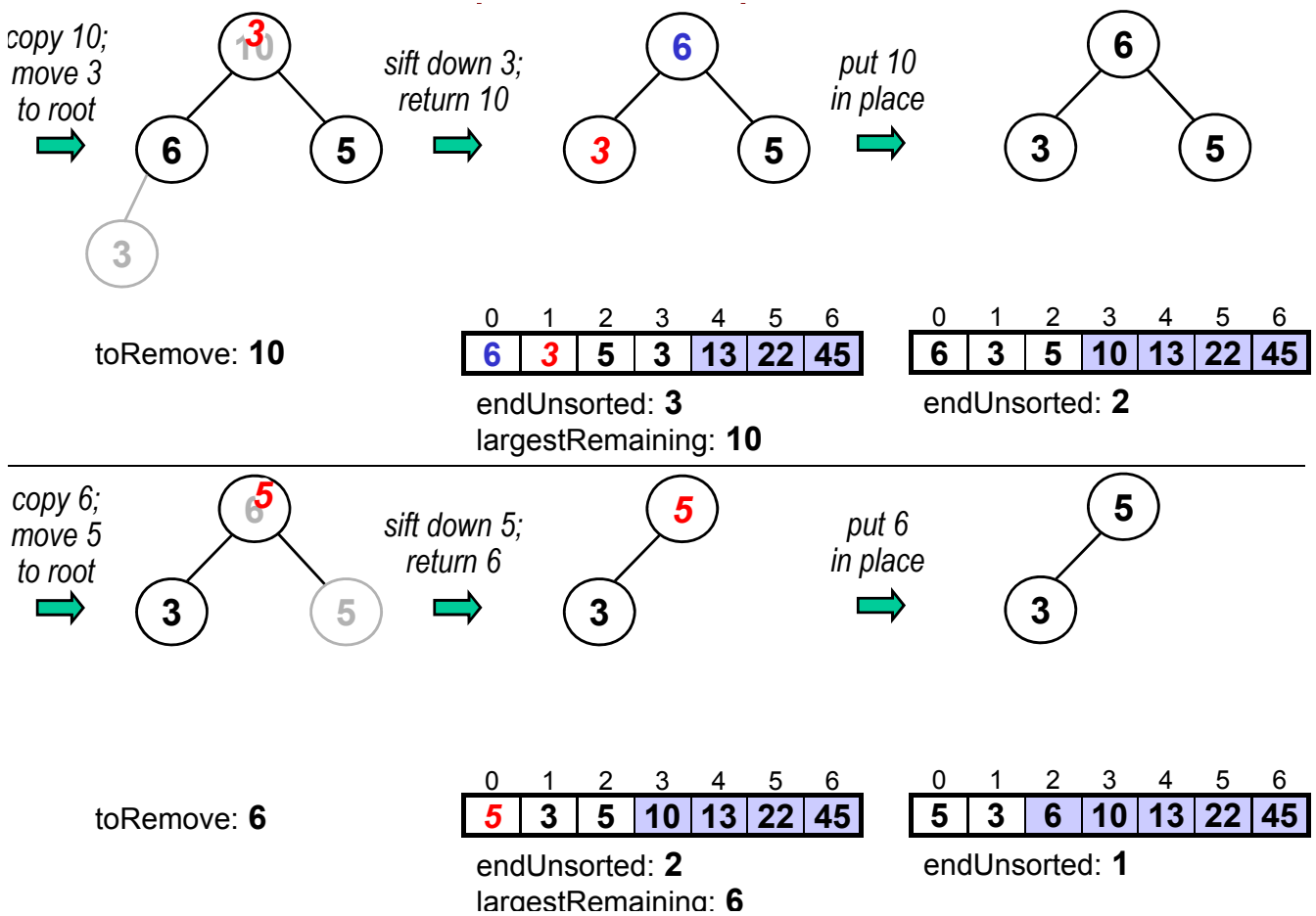
0	1	2	3	4	5	6
22	10	13	6	3	5	45

endUnsorted: **5**

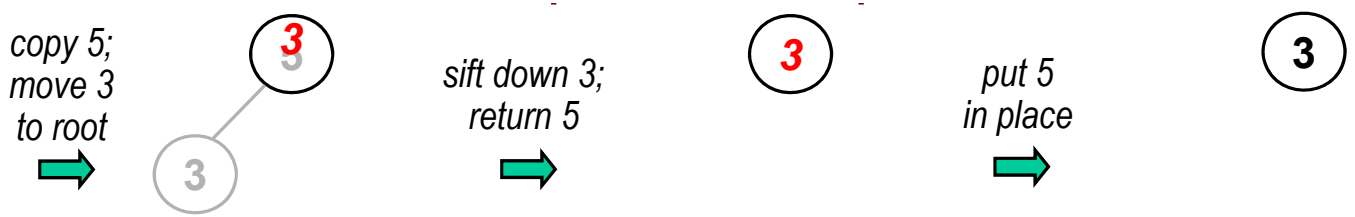
Heap Sort Example



Heap Sort Example



Heap Sort Example



toRemove: **5**

0	1	2	3	4	5	6
3	3	6	10	13	22	45

endUnsorted: **1**
largestRemaining: **5**

0	1	2	3	4	5	6
3	5	6	10	13	22	45

endUnsorted: **0**

How does Heap Sort performance ?

algorithm	best case	avg case	worst case	extra memory
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell sort	$O(n \log n)$	$O(n^{1.5})$	$O(n^{1.5})$	$O(1)$
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(1)$
mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
heapsort	$\boldsymbol{O(n \log n)}$	$\boldsymbol{O(n \log n)}$	$\boldsymbol{O(n \log n)}$	$\boldsymbol{O(1)}$

- Heapsort matches mergesort for the best worst-case time complexity, but it has better space complexity.
- Insertion sort is still best for arrays that are almost sorted.
  - heapsort will scramble an almost sorted array before sorting it
- Quicksort is still typically fastest in the average case.

Priority Queue

- A priority queue is similar to a regular queue, but each item in the queue has an additional piece of information: Priority
- A priority queue supports two basic operations: insert a new item, and remove the item with the highest priority.

- Applications of priority queues :
  - A hospital emergency room, for example, might prioritize patients waiting based on the severity of their need
  - Process manager of OS (which task should be executed next? One long task could prevent many short (but potentially important) ones from being accomplished.
  - Data Compression (Huffman Coding)

Operations on a Priority Queue

- *Construct* a priority queue from N given items.
- *Insert* a new item.
- *Remove the maximum* item.
- *Change the priority* of an arbitrary specified item.
- *Remove* an arbitrary specified item.
- *Join* two priority queues into one large one.

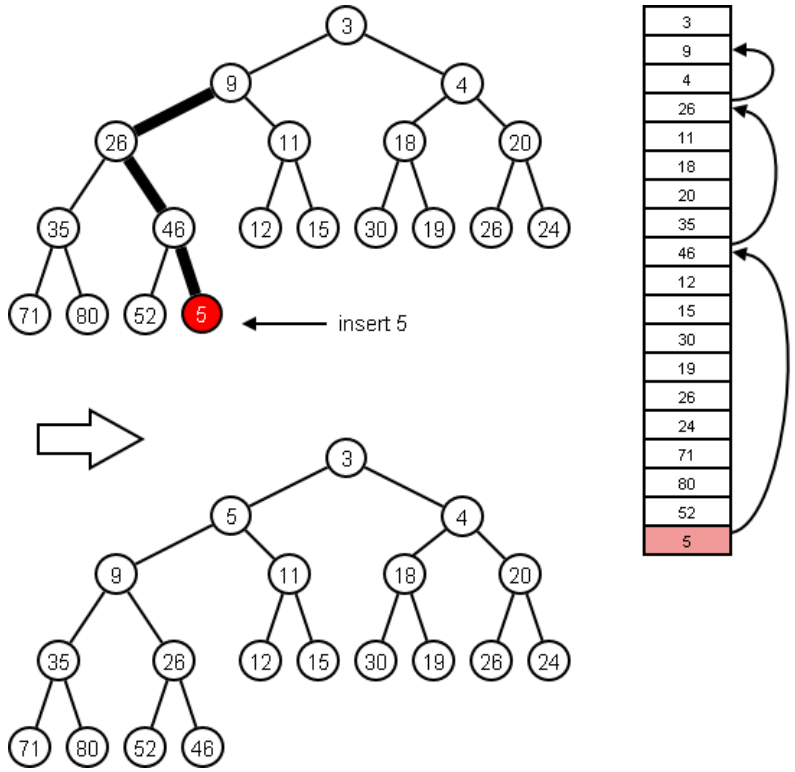
Worst-case costs of priority-queue operations

Operation	Insert	Remove Max	Find Max
ordered array	N	1	1
ordered list	N	1	1
unordered array	1	N	N
unordered list	1	N	N
binary heap	lg N	lg N	1

worst-case asymptotic costs for PQ with N items

Heap based Priority Queue - Insert

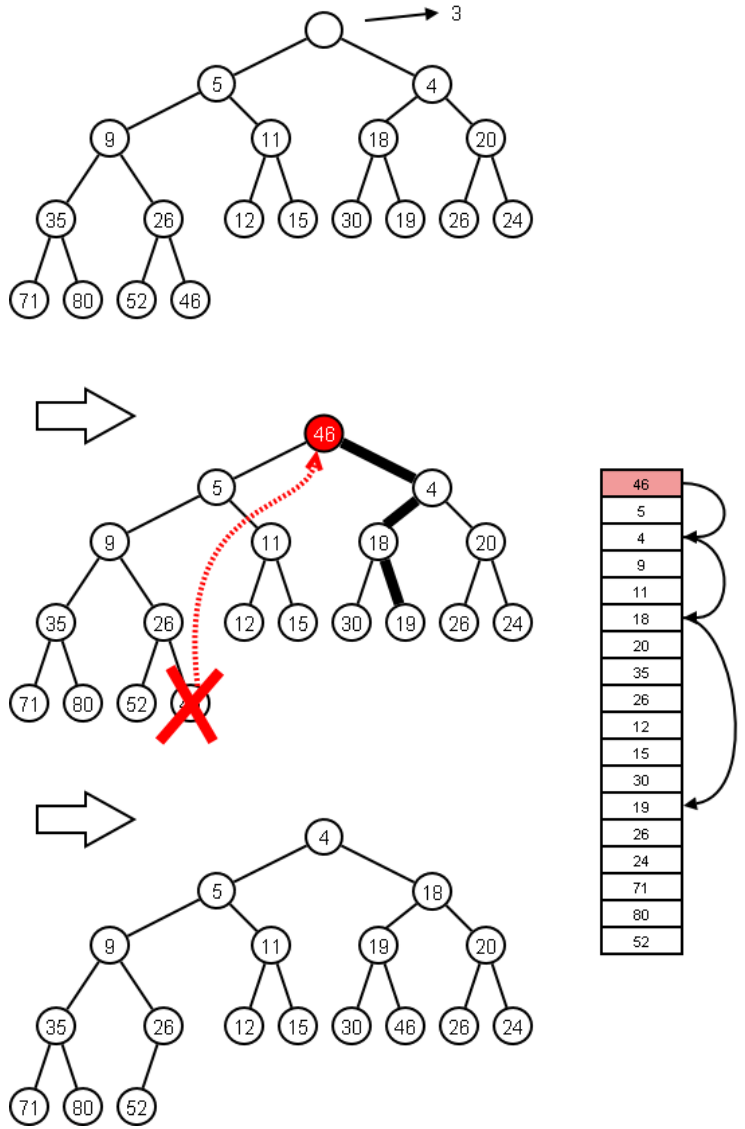
1. Insert the new element at the end of the tree
2. Heapify the tree



Heap based Priority Queue – Remove Min

1. return the value at the top,
2. move the last item in the heap to the root
3. sift down the new root item until :

   it is >= its children (or it's a leaf)



Huffman Coding

- Huffman coding is an algorithm devised by David A. Huffman of MIT in 1952 for *compressing* data.
- **What if we used fewer than 8 bits for characters that are more common?**
- A Huffman code is a type of optimal prefix code(no code is prefix of any other) used for lossless data compression.
- More common symbols are represented using fewer bits than less common symbols

Example

Consider the following (simple) example. Imagine we have the following data:

```
bbbbaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
cccaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
ddaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

How many bits to store with ASCII Coding?

First, we find the ASCII code for each letter:

a	$\rightarrow$	01100001,	b	$\rightarrow$	01100010,
c	$\rightarrow$	01100011,	d	$\rightarrow$	01100100

Since each line has 80 letters, and each letter code is 8 bits, the number of bits required is:

$$80 \cdot 8 \cdot 4 = 2560$$

How many bits to store with Huffman Coding?

Since a is most frequent, we use a short code for it, then we use the next longest code for b, etc:

a	$\rightarrow$	1,	b	$\rightarrow$	00,
c	$\rightarrow$	011,	d	$\rightarrow$	010

This data has 229 a's, 4 b's, 3 c's, and 2 d's. Since we need one bit for a, two for b, and three for c and d, the total count of bits is:

$$229 \cdot 1 + 4 \cdot 2 + 3 \cdot 3 + 2 \cdot 3 = 255$$

What is prefix code

- In prefix code no code is prefix of any other
- For example :

  0 , 11, 101 →prefix code

  0, 1, 11 → non prefix code

Constructing a Huffman Code

Example Text : aba ab cabbb

Encoding Steps:

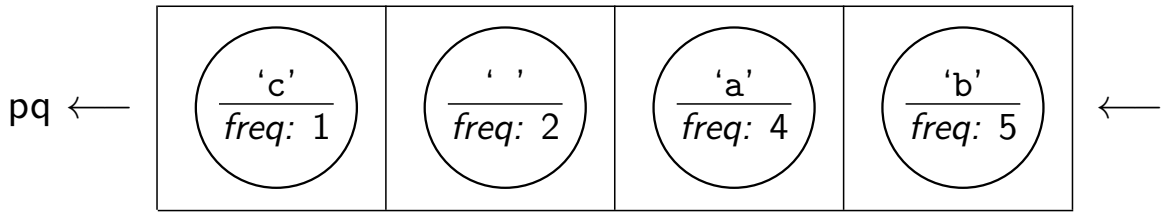
1. Count the Frequencies of the Characters

Character	Count
‘ ’	2
‘a’	4
‘b’	5
‘c’	1

Constructing a Huffman Code

2. Create a leaf node for each symbol and add it to the priority queue.

Character	Count
' '	2
'a'	4
'b'	5
'c'	1



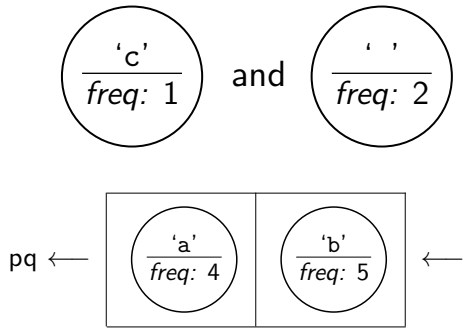
Constructing a Huffman Code

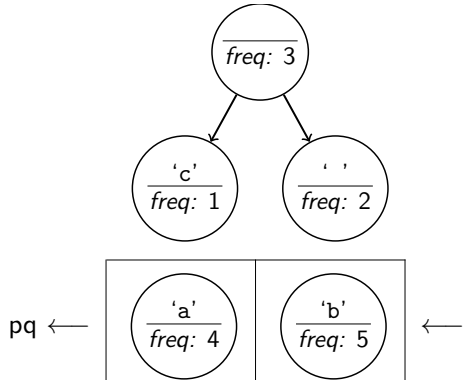
3. While there is more than one node in the queue:

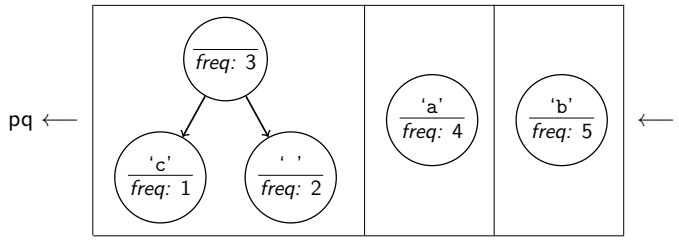
Remove the two nodes of highest priority (lowest probability) from the queue

Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.

Add the new node to the queue.

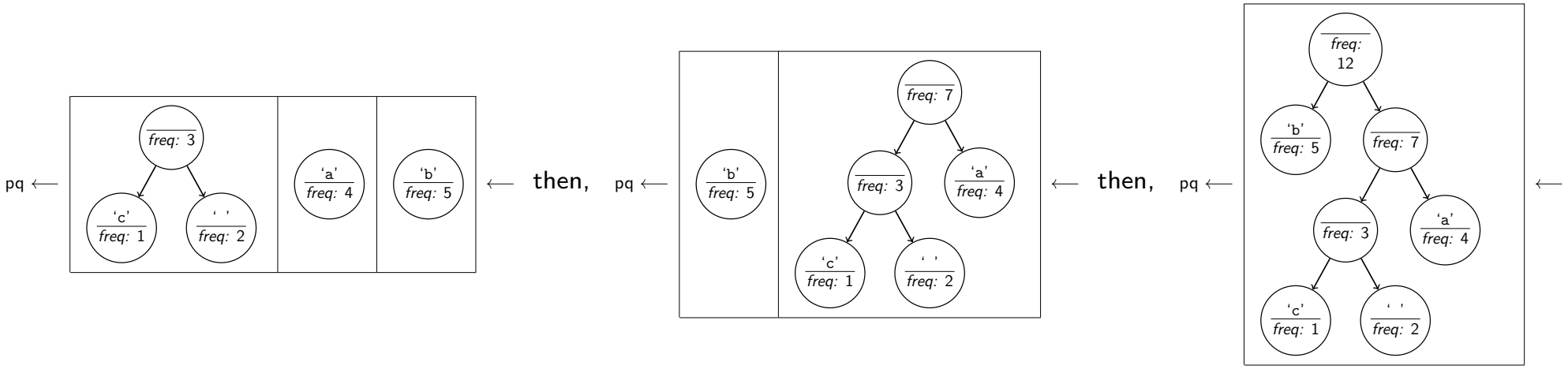






Constructing a Huffman Code

- Remaining Steps :

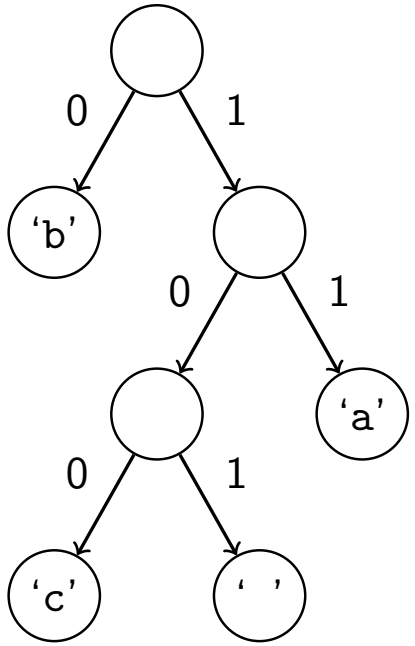


- this algorithm operates in O(*n* *log *n*) time, where *n* is the number of symbols.

Read off the Huffman Code

The tree we've constructed looks like the following:

character	count	symbol
b	5	0
a	4	11
' '	2	101
c	1	100



The string "ab ca" is coded as:

11010110011