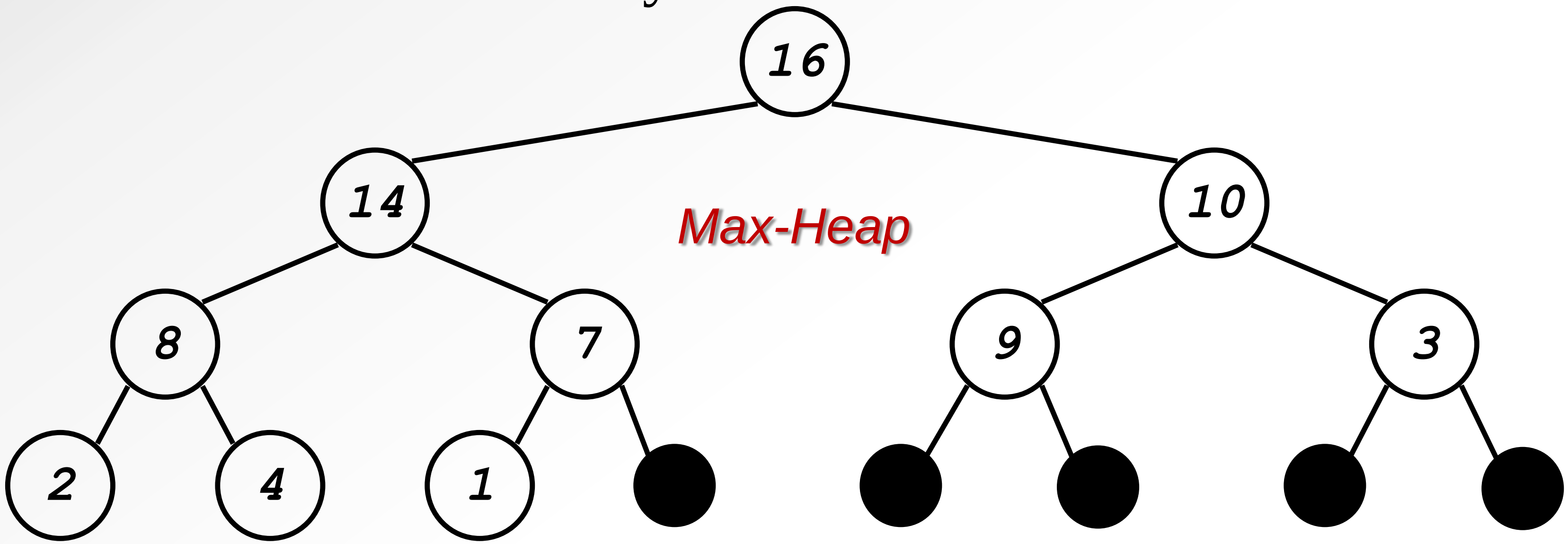
Topics

Heaps

Dr. S.S.Shehaby

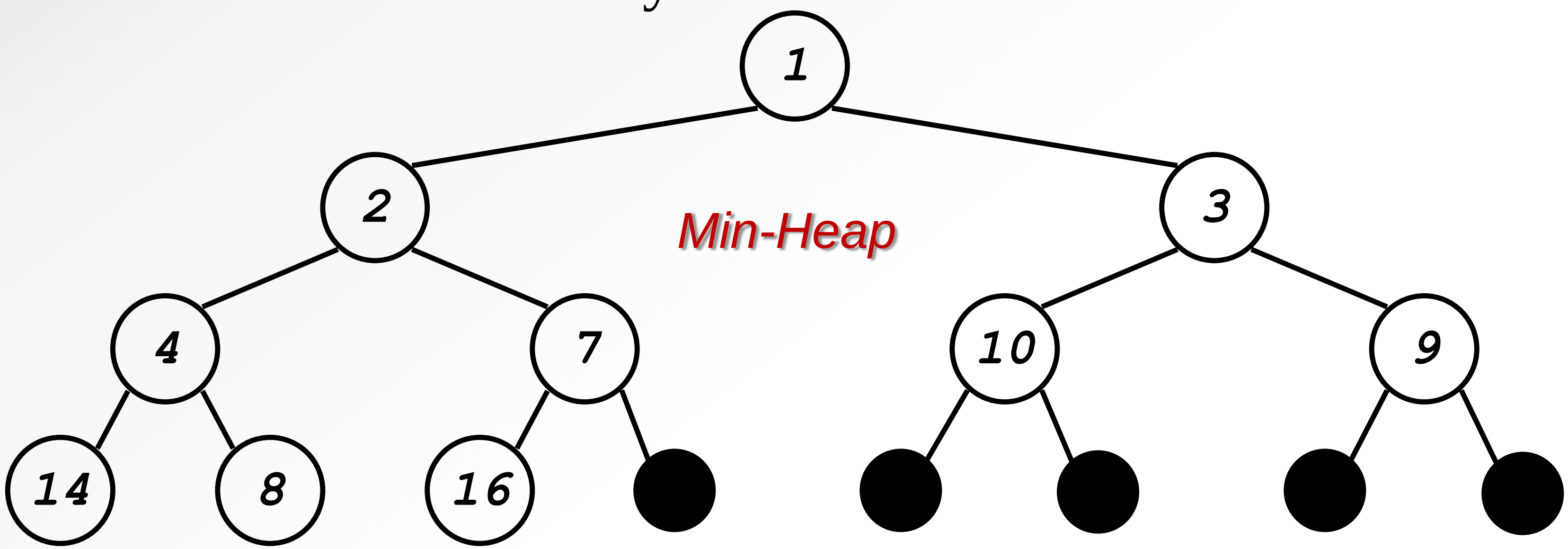
Heaps

- A **Heap** is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:
  - Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
  - Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.



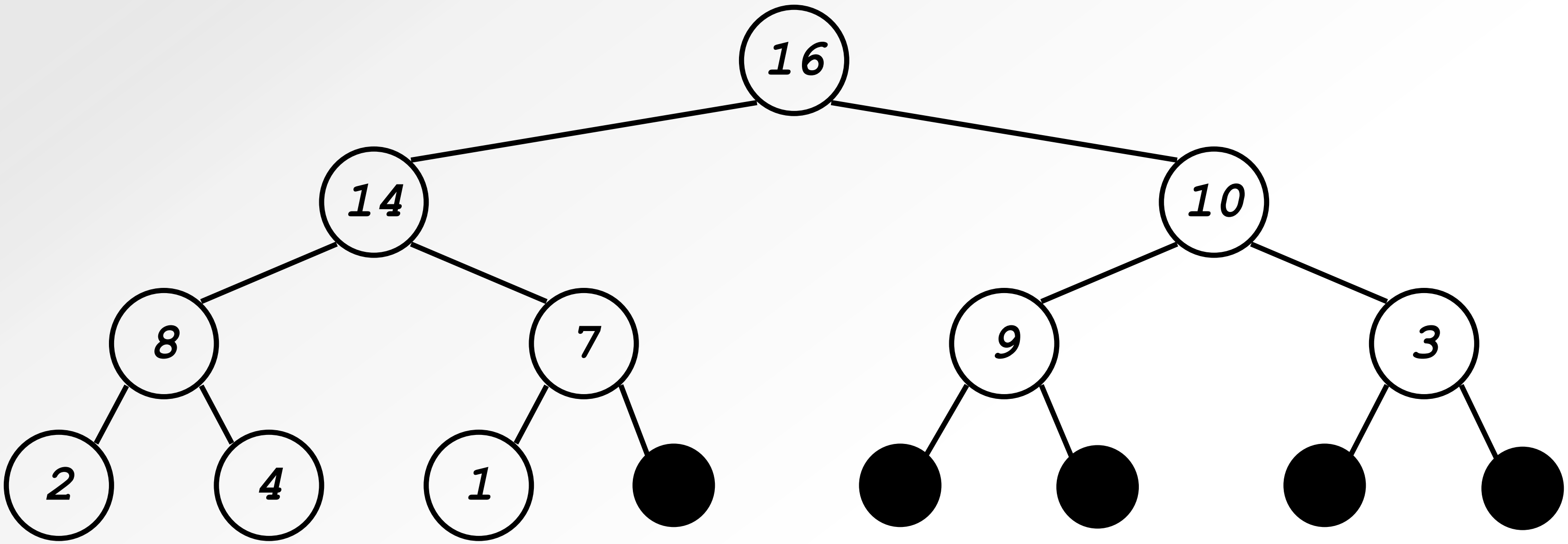
Heaps

- A **Heap** is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:
  - Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
  - Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.



Heaps

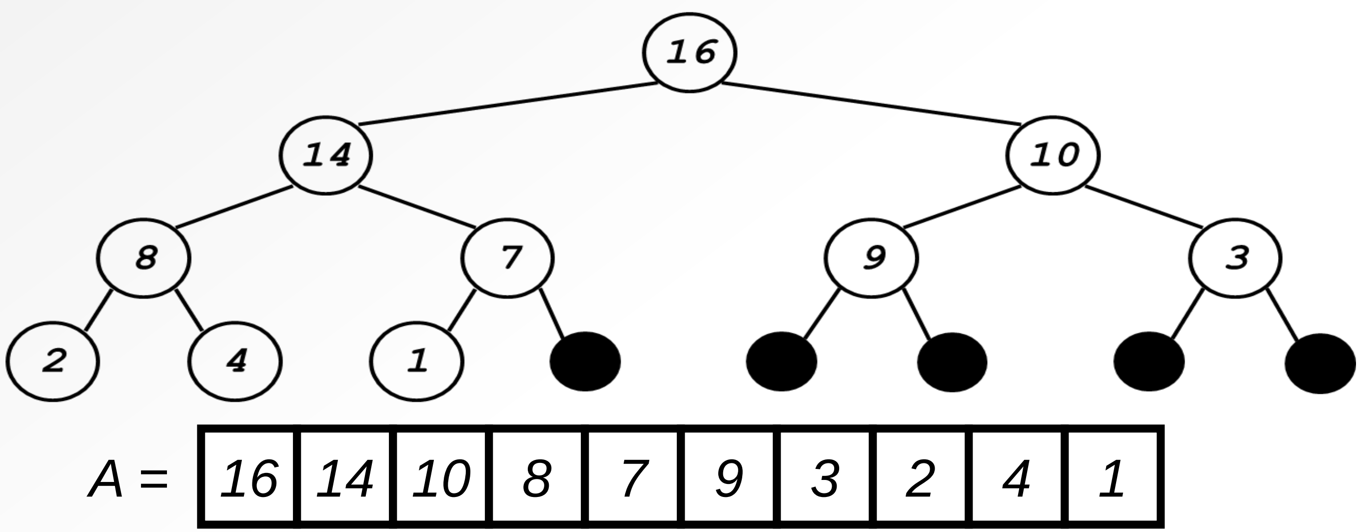
- In practice, heaps are usually implemented as arrays *(since complete)*



A =	16	14	10	8	7	9	3	2	4	1

Heaps

- To represent a complete binary tree as an array:
  - The root node is A[0]
  - Node *i* is A[*i*]
  - The parent of node *i* is A[(*i*-*1*)/2] (note: integer divide)
  - The left child of node *i* is A[2*i*+*1*]
  - The right child of node *i* is A[2*i* + 2]



Referencing Heap Elements

- So…

  Parent(i) { return $\lfloor(i-1)/2\rfloor$; }

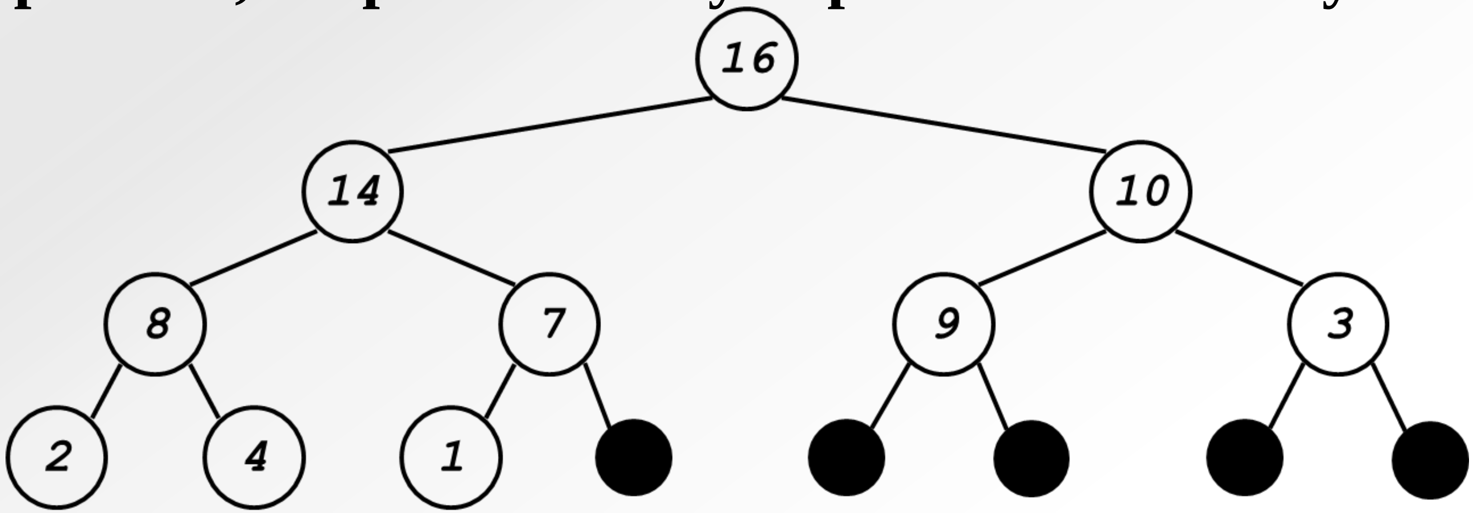
  Left(i) { return 2*i+1; }

  right(i) { return 2*i + 2;//Left(i)+1}

- An aside: How would you implement this most efficiently?

Heaps-sentinel

- **In practice, heaps are usually implemented as arrays:**



A =	11	16	14	10	8	7	9	3	2	4	1	·	·

A[0] = number of elements, sentinel

Parent(i) { return i/2;}
Left(i) { return 2*i; }
right(i) { return 2*i + 1;// //Left(i)+1}

The Heap Property

- Max-Heaps also satisfy the *heap property*:

  A[*Parent*(i)] $\geq$ A[i]   for all nodes $i > 0$

  - In other words, the value of a node is larger than its antecedents.
  - *In MIN-Heap* A[*Parent*(i)] $\leq$ A[i]
- Definitions:
  - The *height* of a node in the tree = the number of edges on the longest downward path to a leaf
  - The height of a tree = the height of its root=$\lceil \log_2(N) \rceil$ where N=number of nodes.

Some C++

File heap.hpp	File heap.cpp

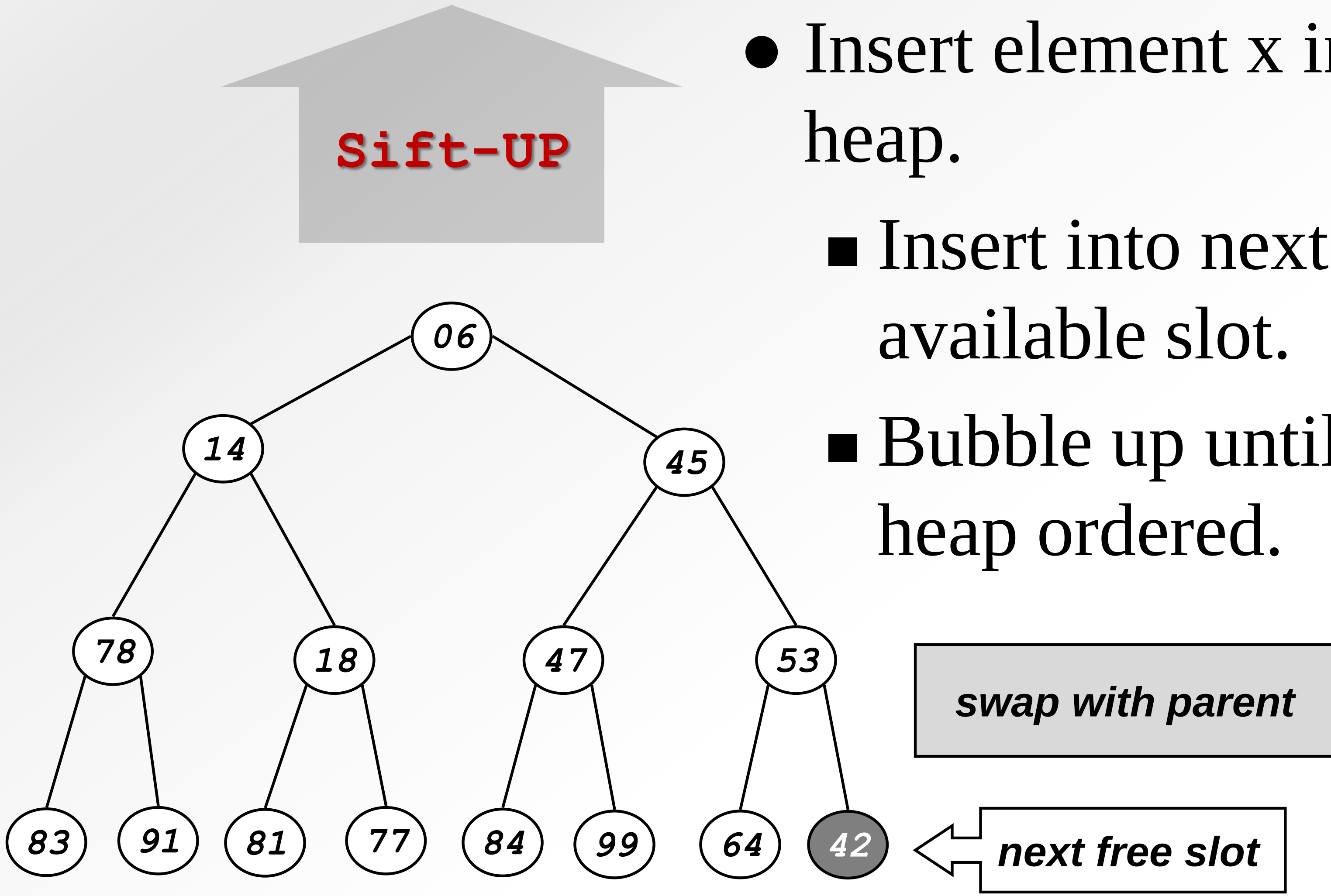
File heap.hpp

```
class Heap{
private:
    int n, capacity;
    int * arr; // 1 based
public:
    Heap();    //constructors
    Heap(int);
//methods
    void push(int value);
    int peek();
    int pop();
    int isEmpty()
                {return n==0;}
    int size()
            {return arr[0];}
    int isFull();
};
```

File heap.cpp

```
#include "heap.h"
//... Other includes
Heap::Heap(int c) {
// keeps blocks powers of 2
 capacity=pow(2,ceil(log2
           (c+1))) -1;
 arr=(int*)malloc((capacity+1)
                *sizeof(int));
 n=0;
}
Heap::Heap(){
    Heap(127);
}
```

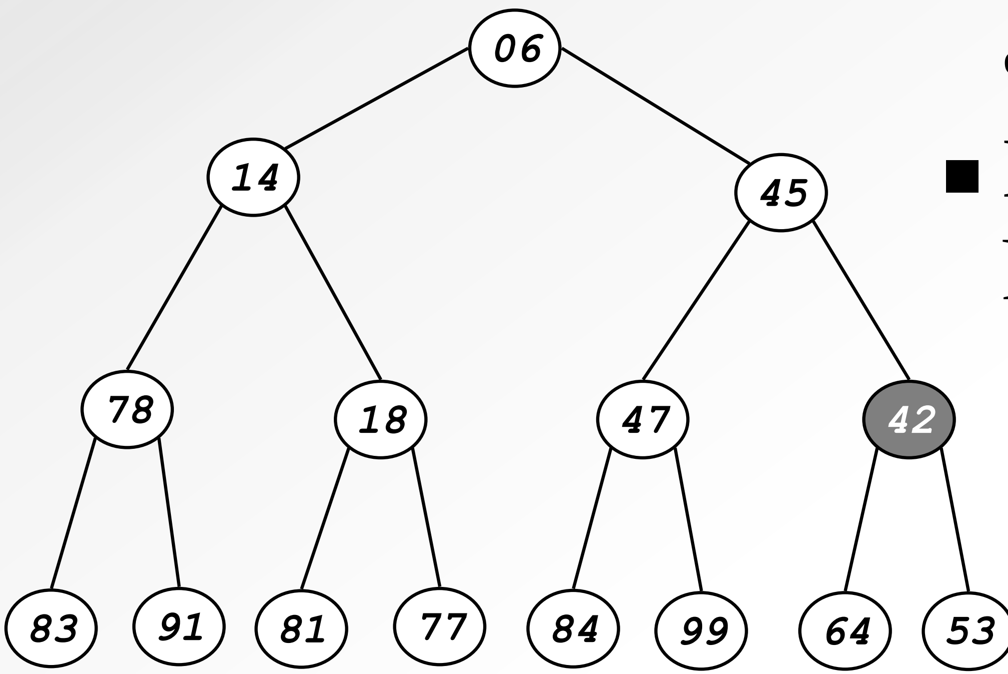
Binary Heap: Insertion



- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.

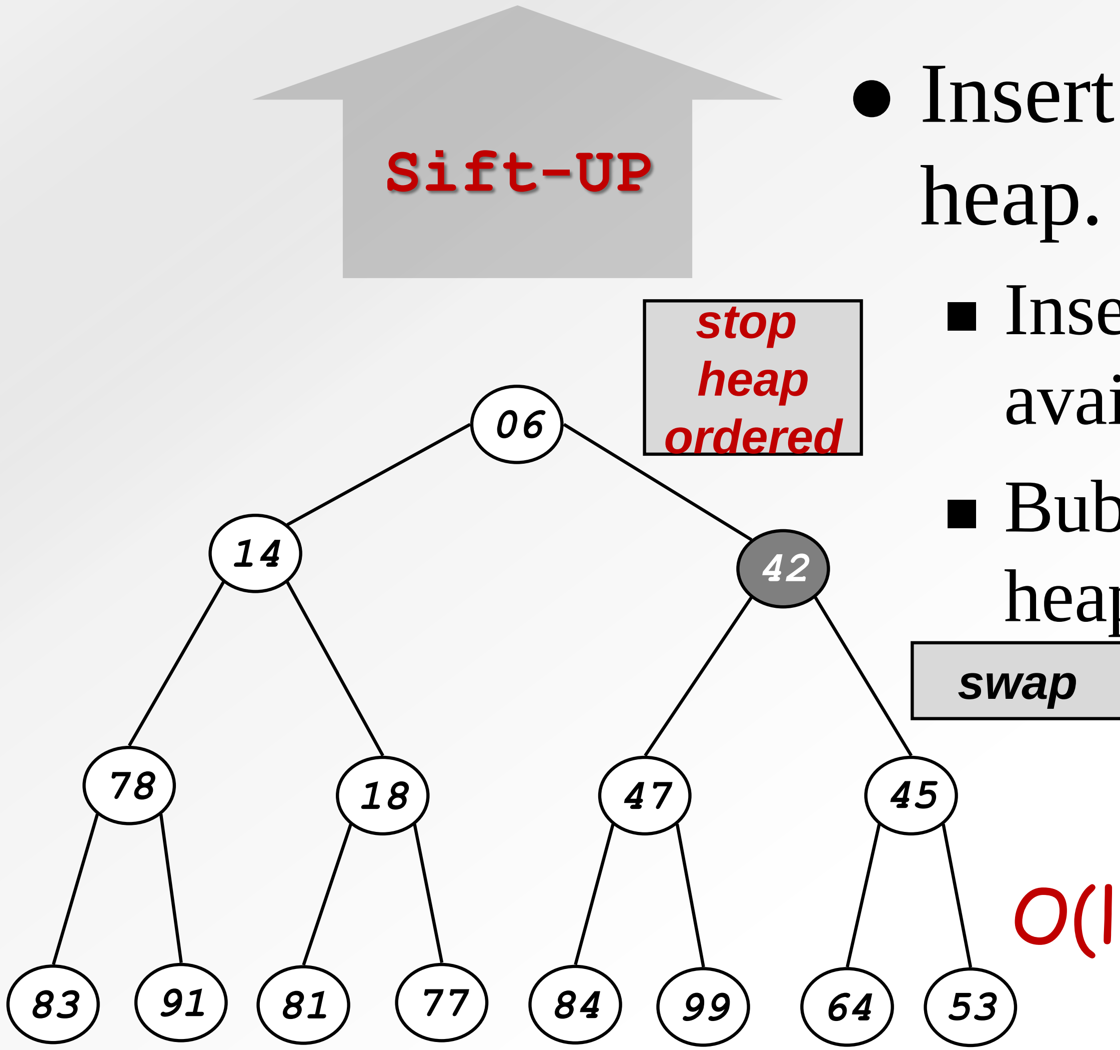
Binary Heap: Insertion

- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.



swap with parent

Binary Heap: Insertion



- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.

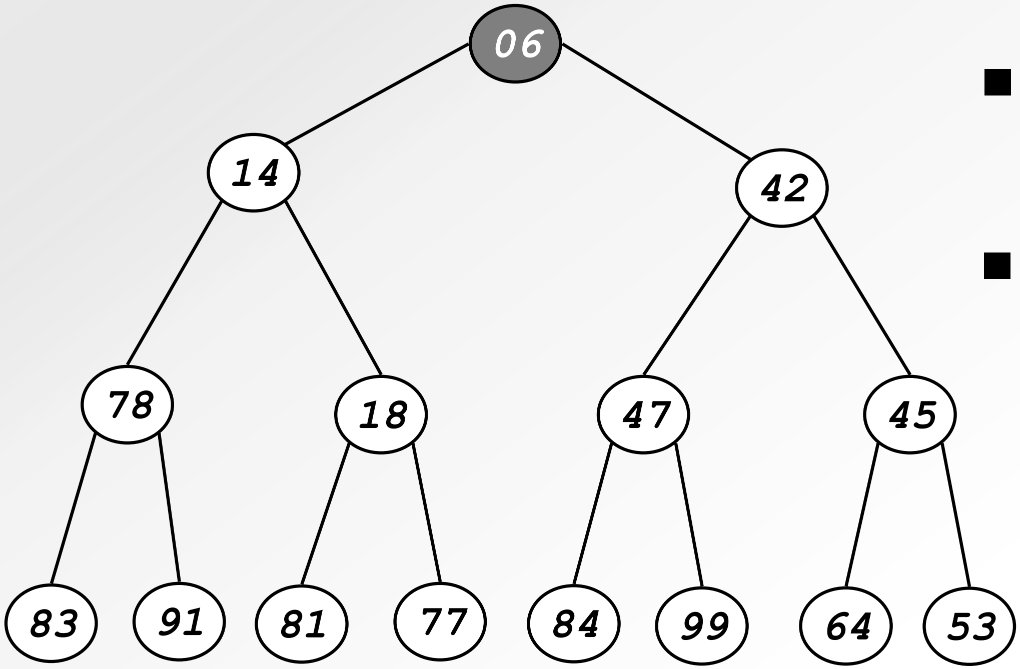
O(log N) operations.

Some C++

File heap.hpp

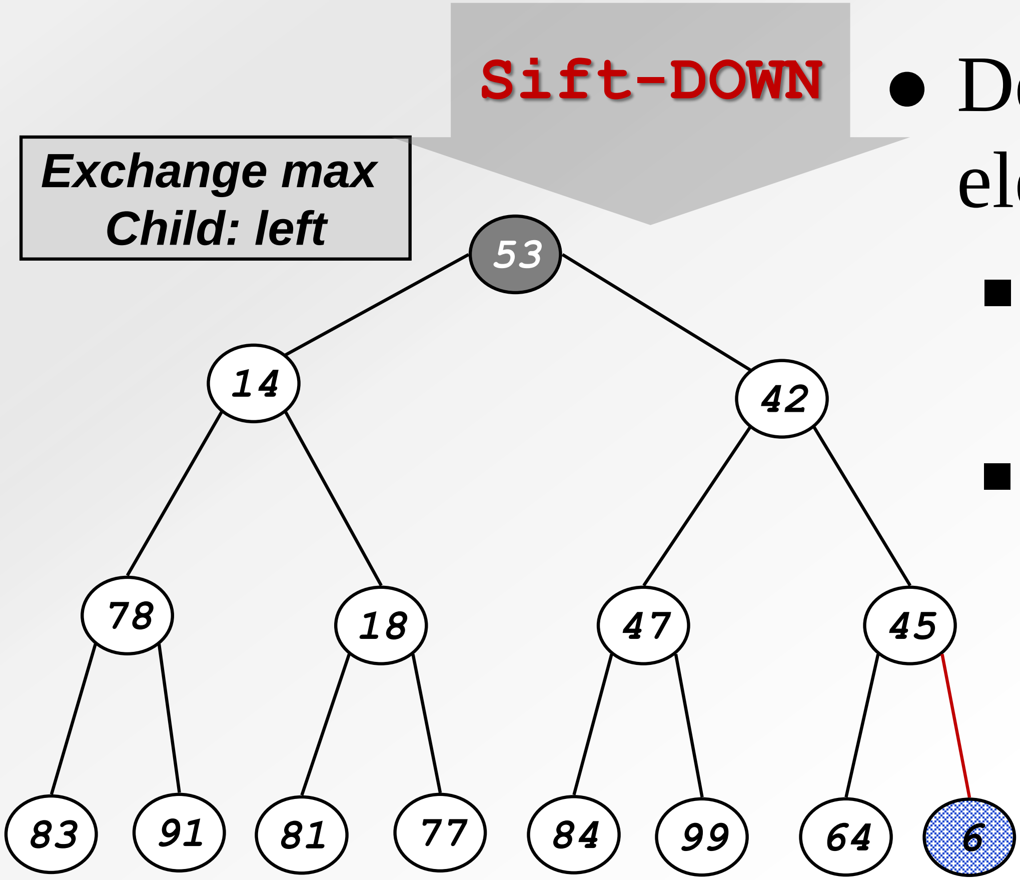
```
void Heap::push(int value){ //sift_up
   int i;n++;
   for(i=n; arr[i/2] > value && i>1; i/=2 )
                                   arr[ i ] = arr[ i / 2 ];
   arr[ i ] = value;
   }
```

Binary Heap: Delete Min



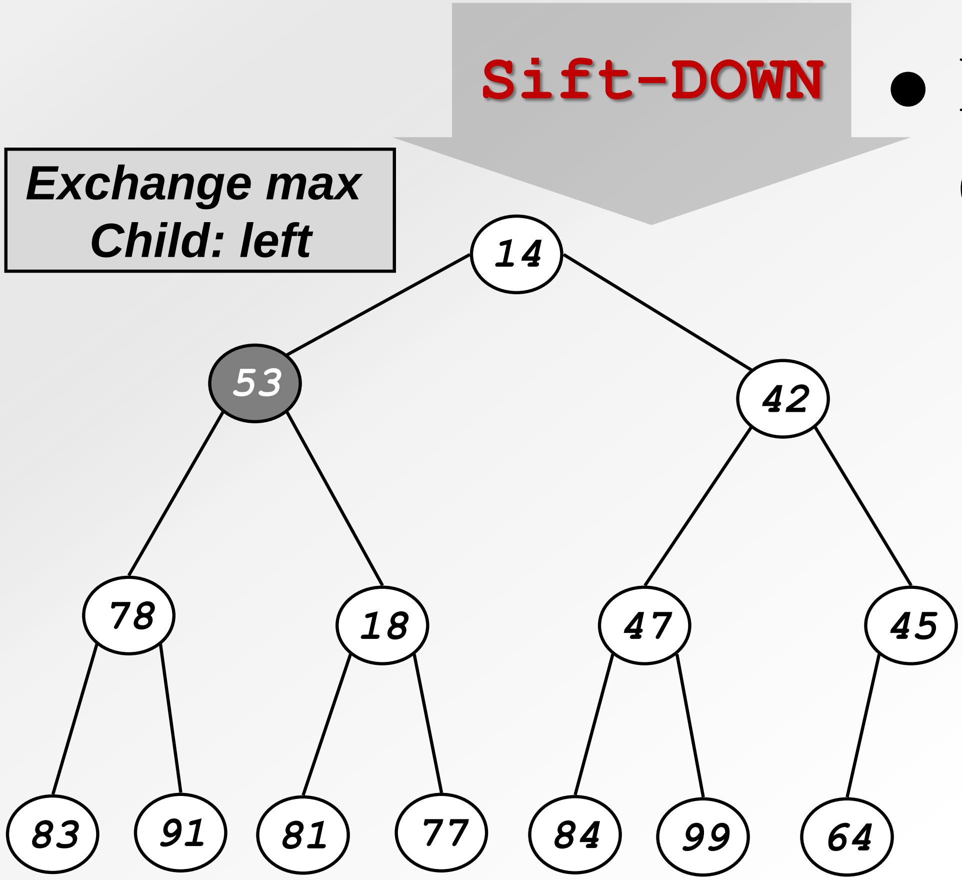
- Delete minimum element from heap.
  - Exchange root with max leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted (max in min-heap, min in max-heap)

Binary Heap: Delete Min



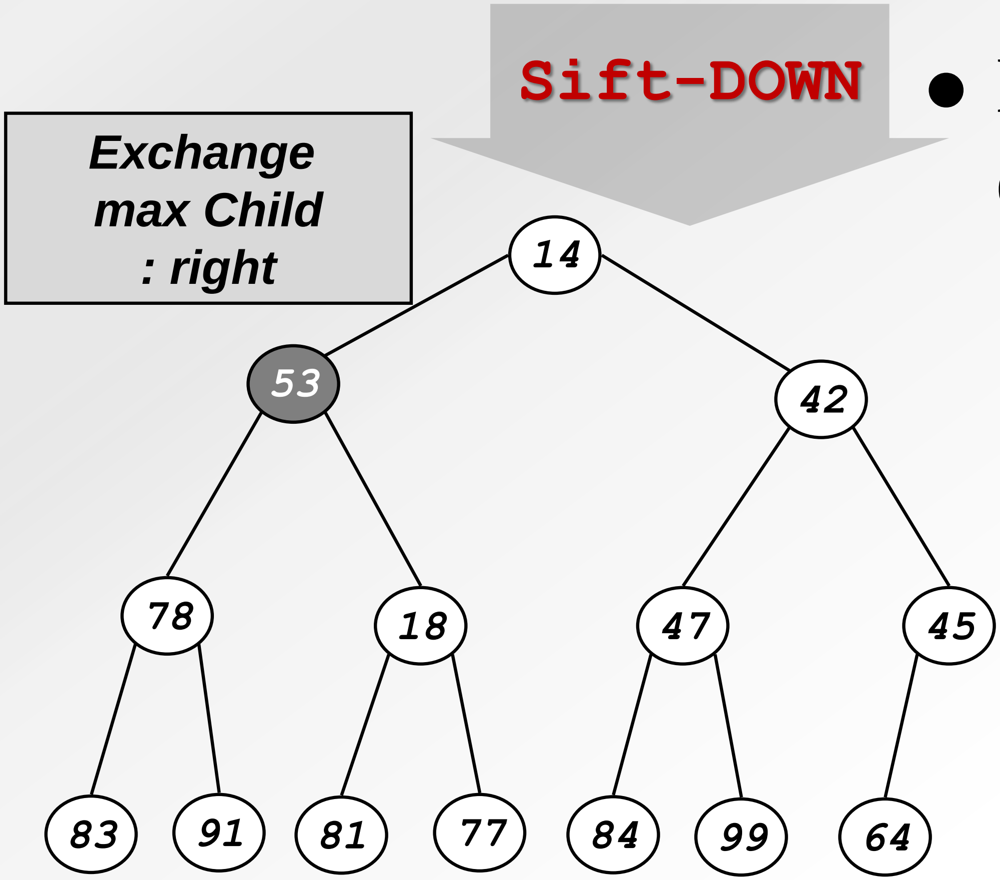
- Delete minimum element from heap.
  - Exchange root with max leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted (max in min-heap, min in max-heap)

Binary Heap: Delete Min



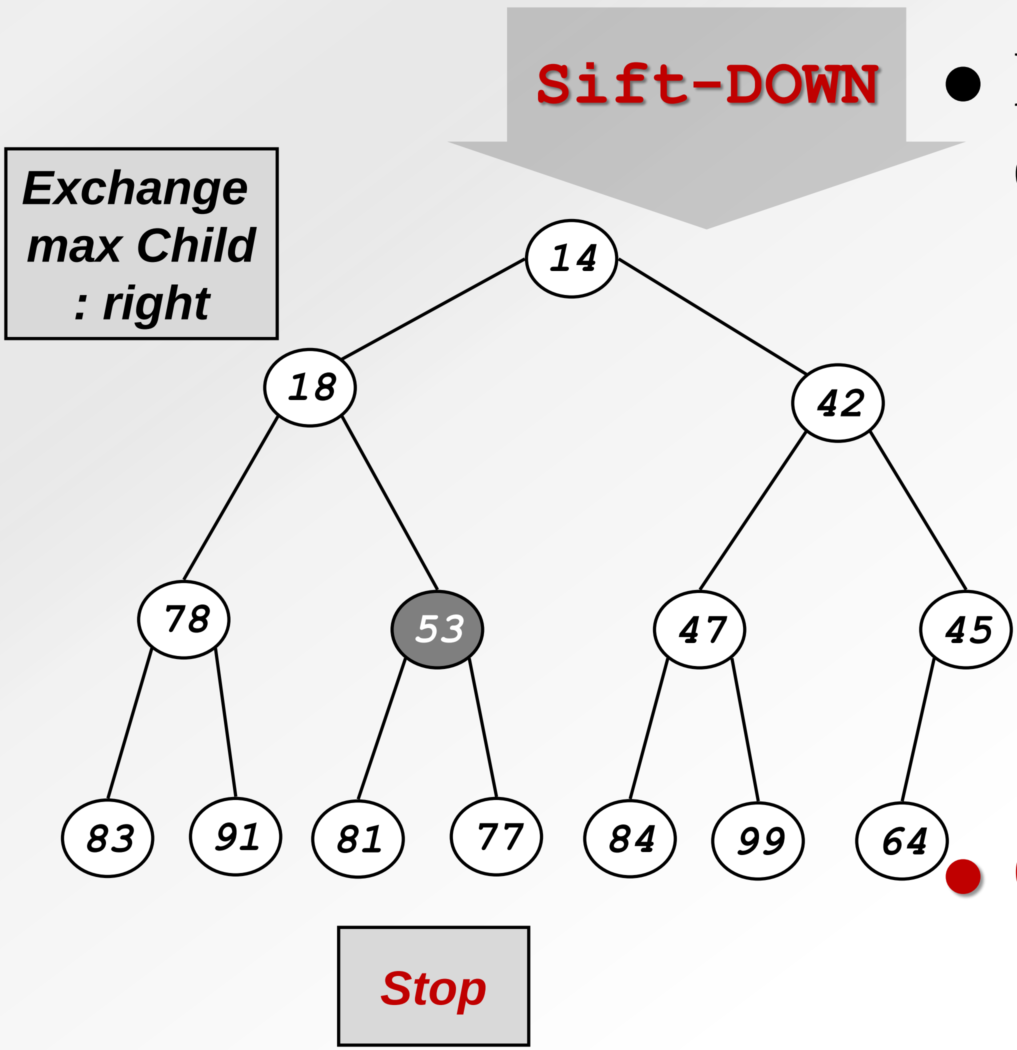
- Delete minimum element from heap.
  - Exchange root with max leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted (max in min-heap, min in max-heap)

Binary Heap: Delete Min



- Delete minimum element from heap.
  - Exchange root with max leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted (max in min-heap, min in max-heap)

Binary Heap: Delete Min



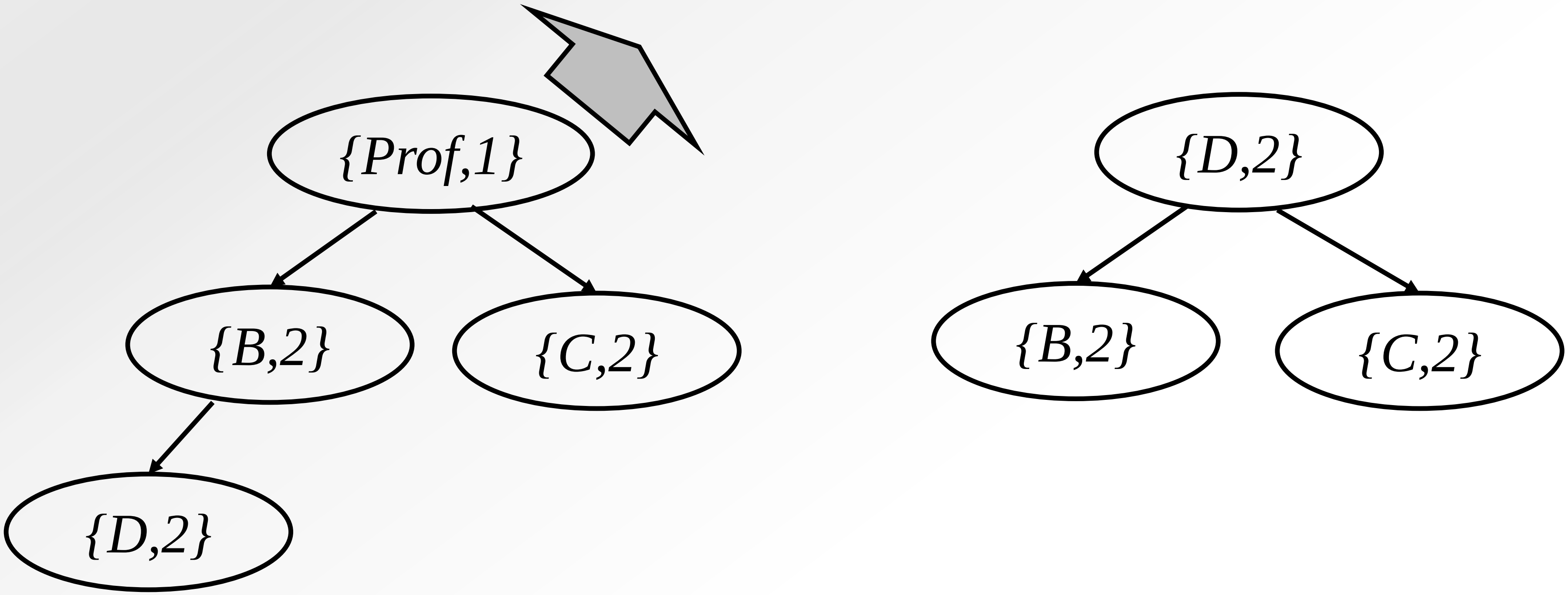
- Delete minimum element from heap.
  - Exchange root with max leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted (max in min-heap, min in max-heap)
- O(Log (N))

Some C++

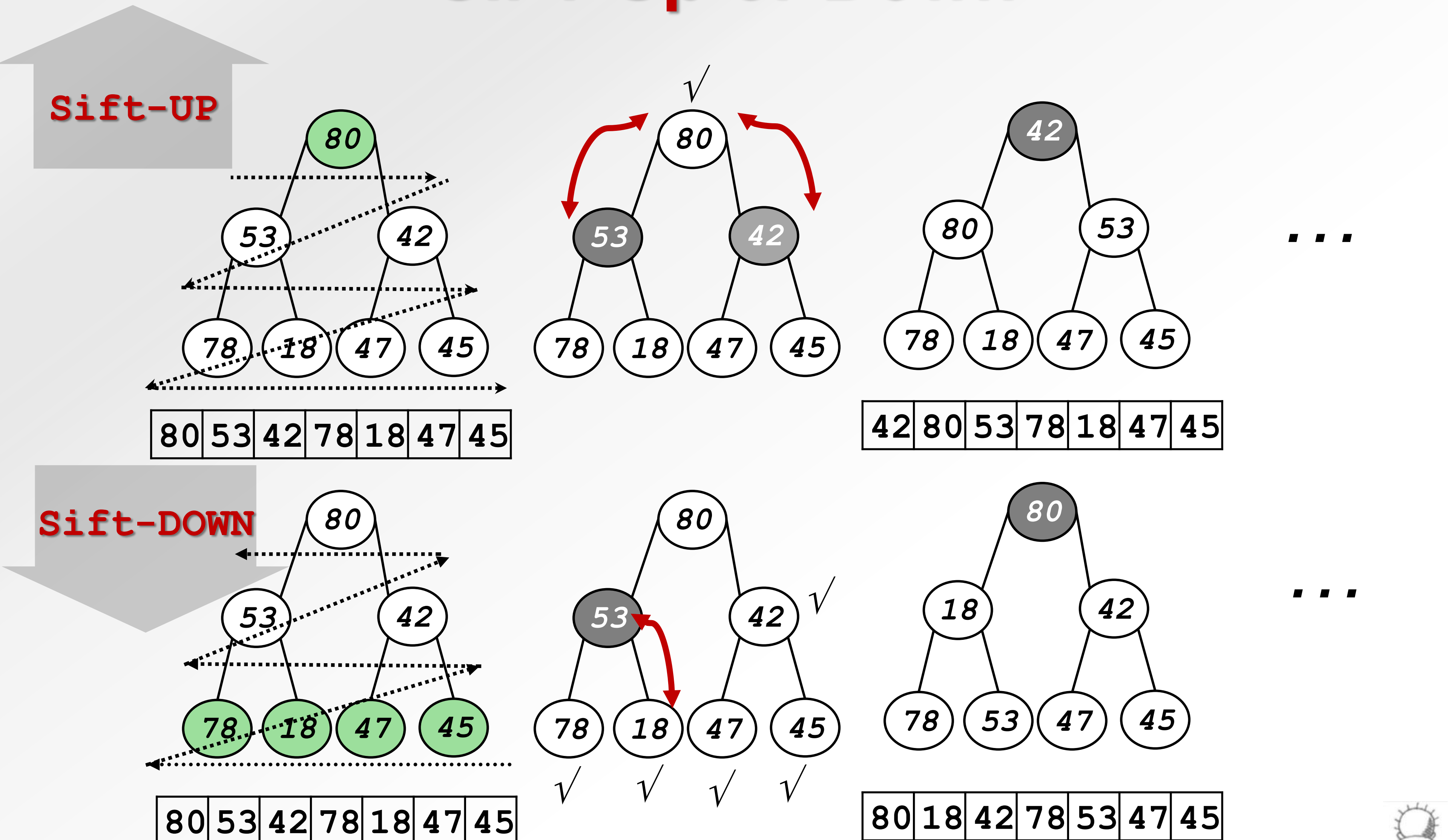
File heap.hpp

```
int Heap::pop() { //sift_down
  int last=arr[n--],i, child;
  int retValue=arr[1];
  for( i = 1; i * 2 <= n; i = child ) {
     child = i * 2; //child=left
     if(child != n && arr[child+1]
         <arr[child]) child++; //child=right
        /* Percolate one level */
     if( last>arr[child]) arr[i]=arr[child];
     else  break;
     }
    arr[ i ] = last;
  return retValue;
}
```

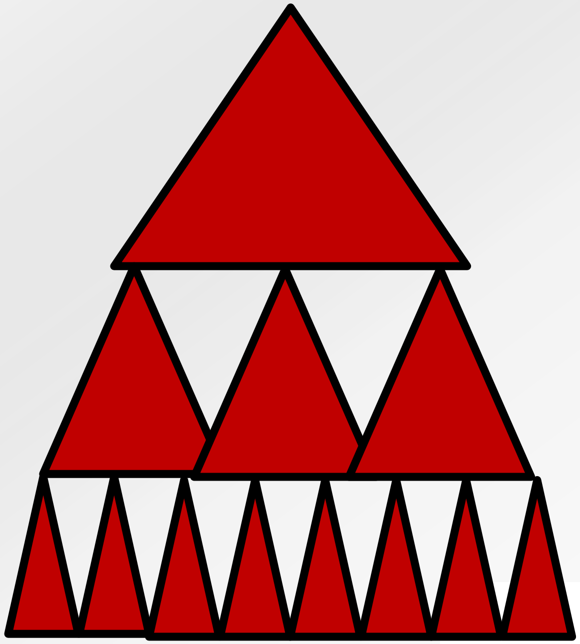
However, Unstable

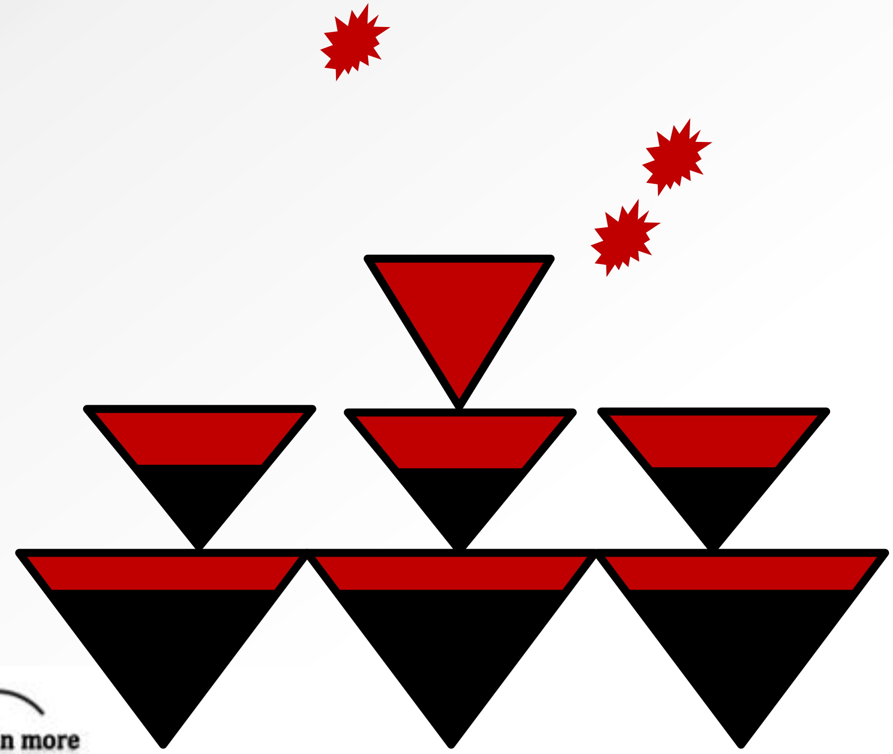


Heapify (in-place)
SIFT Up or Down



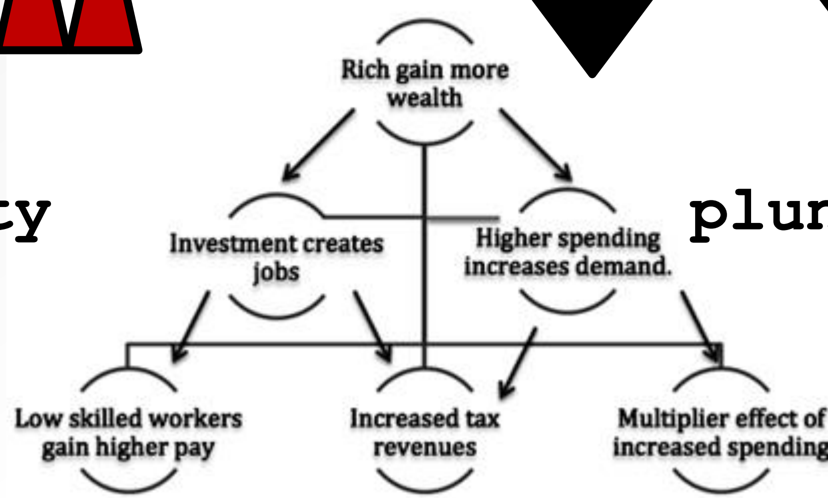
Trickle-down vs exploitation-up



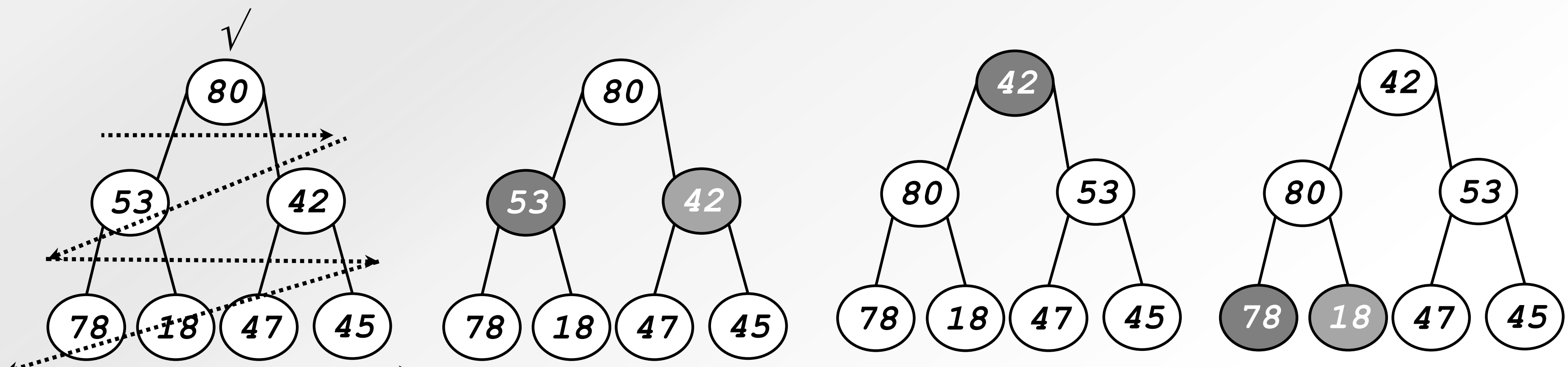


Fortune
prosperity

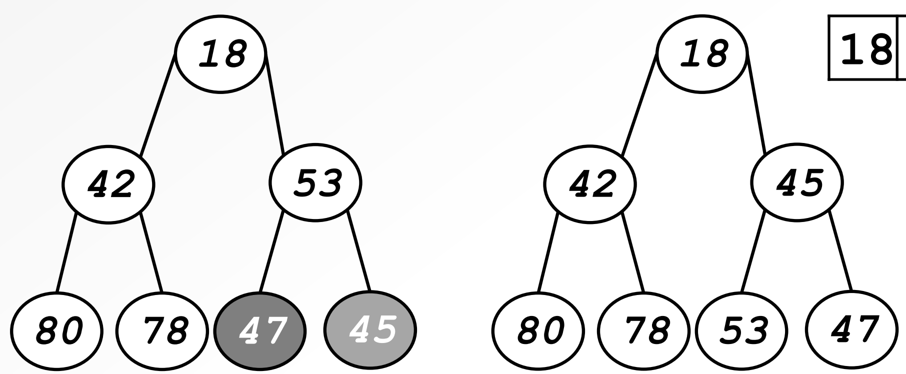
Labor
plunder, theft



Heapify (sift-up)



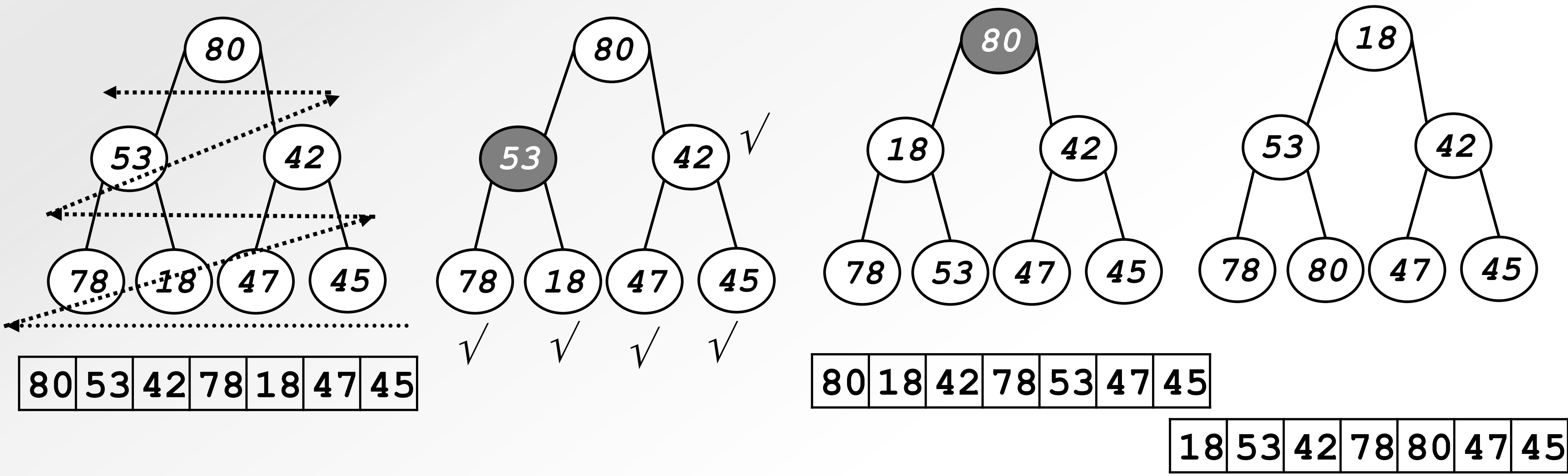
80	53	42	78	18	47	45



18	42	45	80	78	53	47

$$(h * n/2) + ((h-1) * n/4) + ((h-2)*n/8) + \ldots + (0 * 1).$$
where $h=\log(n)$

Heapify (sift-down)



*(0 * n/2) + (1 * n/4) + (2 * n/8) +(3*n/16)+ ... + (h * 1).*
where h=log(n)

Trickle down
vs
Sift-up- bottom-up-Excess value

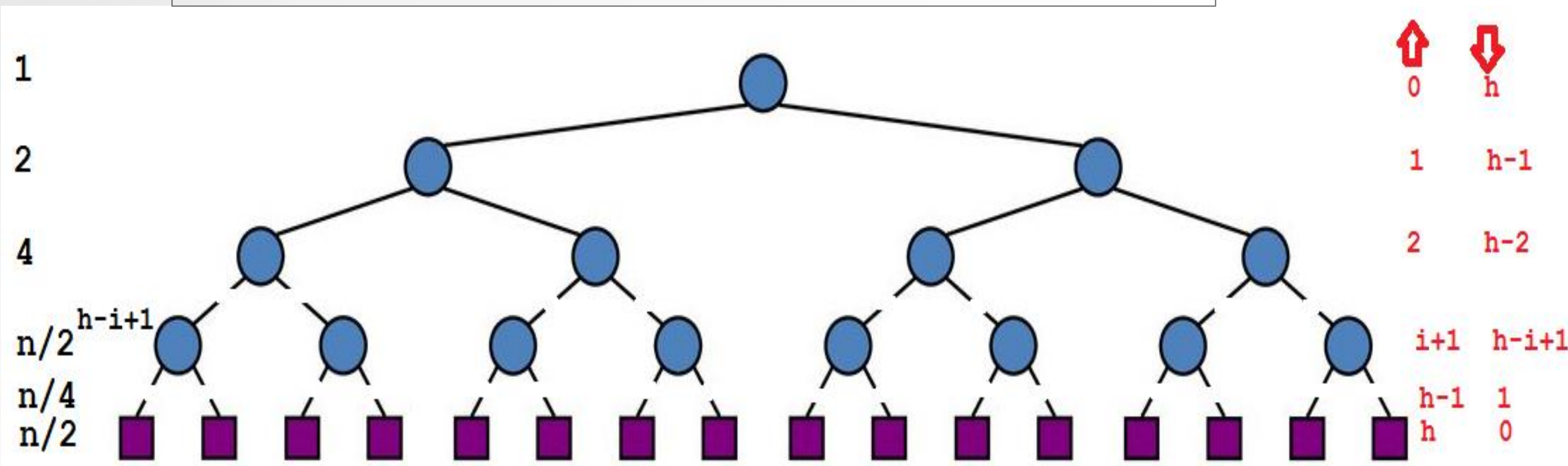
Assuming: let h=⌈log(n)⌉

Sift-up worst case:

*(h * n/2) + ((h-1) * n/4) + ((h-2)*n/8) + ... + (0 * 1).*

Sift-down worst case:

*(0 * n/2) + (1 * n/4) + (2 * n/8) +(3*n/16)+ ... + (h * 1)*



Which is better ?

- <u>Sift-down worst case:</u>

$X=(0*n/2)+(1*n/4)+(2*n/8)+(3*n/16)+ \ldots + (h*1)$

$= n/2[\ 1/2 + 2/4 + 3/8 + \log(n)\ ]$

$< n/2[\ 1/2 + 2/4 + 3/8 + \ldots\ ]$

$<n/2$ * <u>Sum</u>

$(1-x)^{-1}= 1+x+x^2+x^3+\ldots$

<u>Differentiating :</u>

$(1-x)^{-2}= 1+x+2x^2+3x^3+\ldots$, <u>Substituting :x=1/2</u>

$1/4 = 1 +$ <u>Sum</u>

<u>Hence:</u> $X<n$ *constant

$X=O(n)$

<u>Sift-up worst case:</u> $X=(h * n/2) + \ldots \geq n \log(n)$

HeapSort

```
def heapify(arr, n, i): #sift-down O(n)
    largest = i # Initialize largest
    l = 2 * i + 1      # left  = 2*i + 1
    if l>=n: return
    r = 2 * i + 2      # right = 2*i + 2
    if arr[i] < arr[l]: largest = l
    if r < n and arr[largest] < arr[r]:largest = r
    # Change root, if needed
    if largest !=i:
        arr[i],arr[largest]=arr[largest],arr[i]#swap
        heapify(arr, n, largest)
arr = [ 12, 11, 13, 5, 6, 7,14,20,23] ;n=len(arr)
for i in reversed(range(n)): heapify(arr, n, i)
print arr #[23, 20, 14, 12, 6, 7, 13, 11, 5]
```

HeapSort

```
def heapSort(arr): # O(n log(n))
    n = len(arr)
    # Build a maxheap.
    for i in reversed(range(n)):
        heapify(arr, n, i)
    # >> arr=[23, 20, 14, 12, 6, 7, 13, 11, 5]
    # One by one extract elements
    for i in reversed(range(1,n))
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)
arr = [ 12, 11, 13, 5, 6, 7,14,20,23] ;n=len(arr)
heapSort(arr)
print arr # [5, 6, 7, 11, 12, 13, 14, 20, 23]
```

HeapSort

- HeapSort with heapify() creating Max_heap produces Ascending sort; ex. 1,2,3.
- HeapSort with heapify() creating Min_heap produces Descending order sort; ex. 3,2,1.
- Time Complexity n log (n).
- No extra space required.

Sorting Revisited

Algorithm	Worst case	Average case	extra memory
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$
Insert in heap then pop	$O(n*\log_2 n)$	$O(n*\log_2 n)$	$O(n)$
Heap sort	$O(n*\log n)$	$O(n*\log n)$	$O(1)$
Radix sort (only integers, fixed)	$O(n*\log_R max)$	$O(n*\log_R max)$	$O(n)$
Quick sort	$O(n^2)$	$O(n*\log n)$	$O(1)$
Merge Sort	$O(n*\log n)$	$O(n*\log n)$	$O(n)$

max = maximum integer in list

$\log_R$ **log** *base R, if R not specified default R=2*

Priority Queues

- Heapsort is a nice algorithm, but in practice Quicksort usually wins
- But the heap data structure is incredibly useful for implementing *priority queues*
  - A data structure for maintaining a set *S* of elements, each with an associated value or *key*
  - Supports the operations **Insert()**, **Maximum()**, and **ExtractMax()**

Priority Queue Operations

- **Insert(S, x),push(S,x)** inserts the element x into set S [O(log(n)]
- **Maximum(S), peek(S)** returns the element of S with the maximum key
- **ExtractMax(S),pop(S)** removes and returns the element of S with the maximum key [O(log(n)]

Heap Sort: unstable

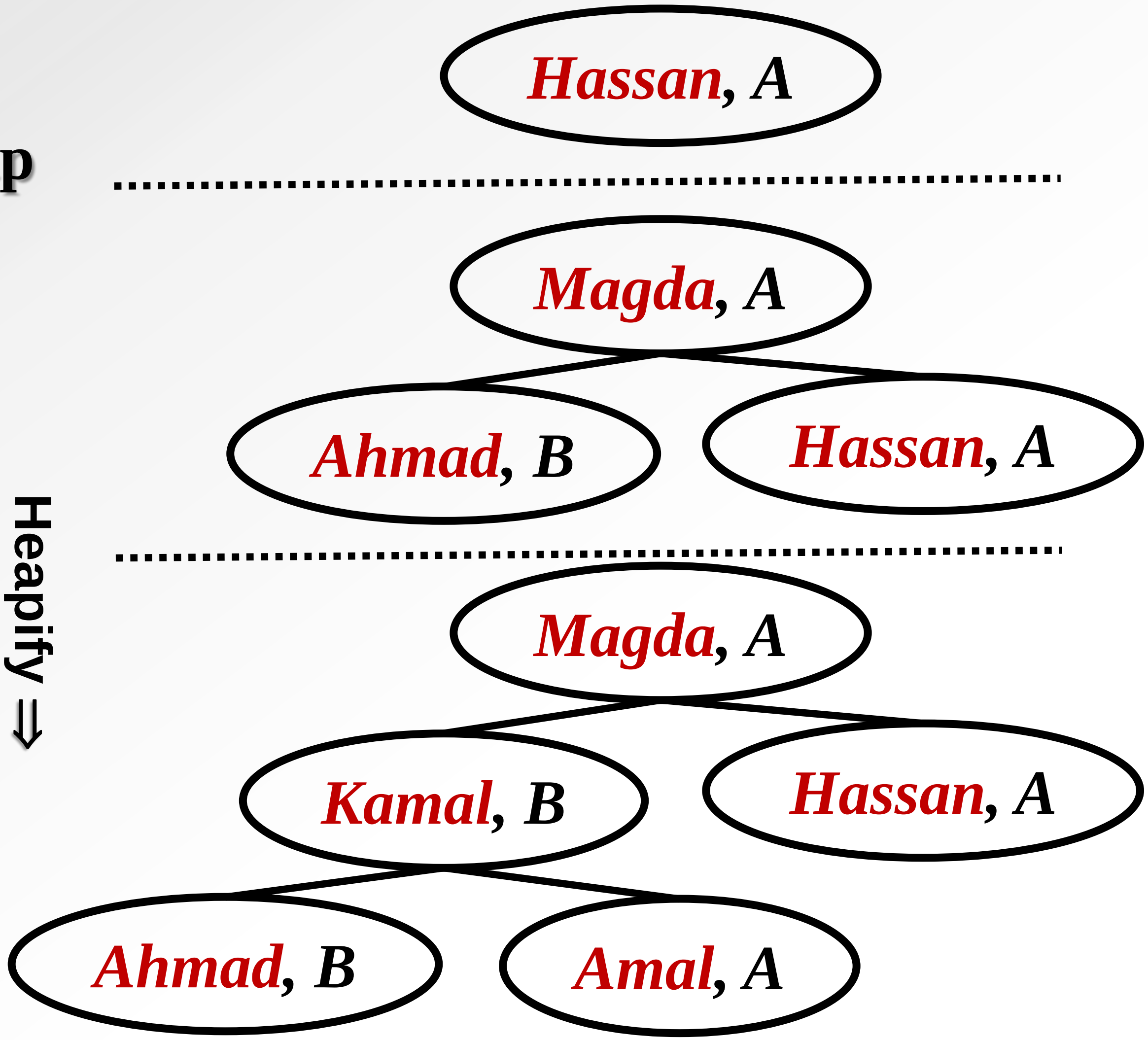
Key: Name: MaxHeap

{Hassan, A}
{Ahmad, B}
{Magda, A}
{Kamal, B}
{Amal, A}

$\Rightarrow$

{Ahmad, B}
{Amal, A}
{Hassan, A}
{Kamal, B}
{Magda, A}

Heapify $\Rightarrow$



HeapSort $\Rightarrow$ [['Ahmad', 'B'], ['Amal', 'A'], ['Hassan', 'A'], ['Kamal', 'B'], ['Magda', 'A']]

Heap Sort: unstable

[['Ahmad', 'B'],
['Amal', 'A'],
['Hassan', 'A'],
['Kamal', 'B'],
['Magda', 'A']]

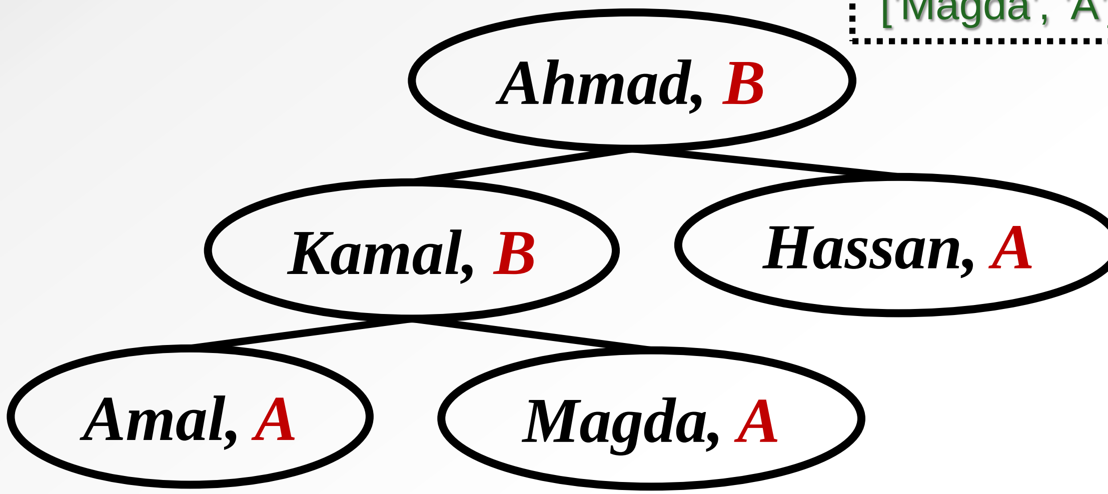
Key: Grade:
MaxHeap

{Ahmad, B}
{Amal, A}
{Hassan, A}
{Kamal, B}
{Magda, A}

⇒

{Magda, A}
{Hassan, A}
{Amal, A}
{Kamal, B}
{Ahmad, B}

Heapify ⇒



HeapSort ⇒

[['Magda', 'A'],
['Hassan', 'A'],
['Amal', 'A'],
['Kamal', 'B'],
['Ahmad', 'B']]

Stability of Sorting

- A sorting algorithm is stable if and only if it ensures that:
  - if $i < j$ and element A[j] comes before A[i] if and only if A[j] < A[i], (NOT EQUAL)here i, j are indices and.
  - Since $i<j$, the relative order is preserved if

    A[i]≡ A[j] i.e. A[i] comes before A[j].
- Bubble Sort, Insertion Sort, Merge Sort, Count Sort, Radix sort are stable while Quick Sort and Heap Sort are not, but forced to be stable by tajing position into consideration !

Thanks

- Any Q ?