Introduction to

Algorithm Design and Analysis

[05] HeapSort

Jingwei Xu
http://cs.nju.edu.cn/ics/people/jingweixu
Institute of Computer Software
Nanjing University

In the last class …

- **The sorting problem**
  - Assumptions
- **InsertionSort**
  - Design
  - Analysis: inverse
- **QuickSort**
  - Design
  - Analysis

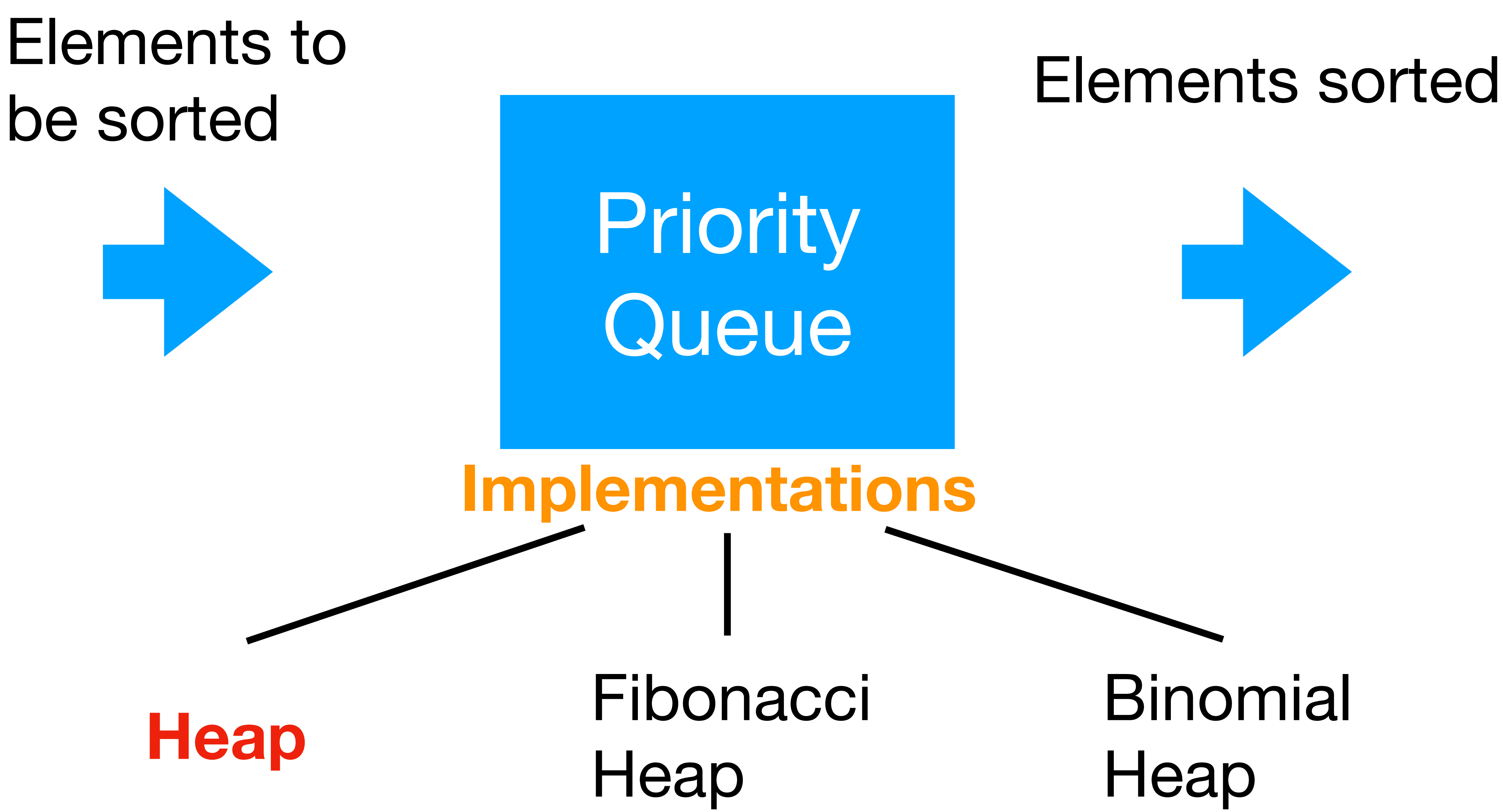
HeapSort

- Heap
- HeapSort

- FixHeap
- ConstructHeap

- Complexity of HeapSort
- Accelerated HeapSort

How HeapSort Works



Elementary Priority Queue ADT

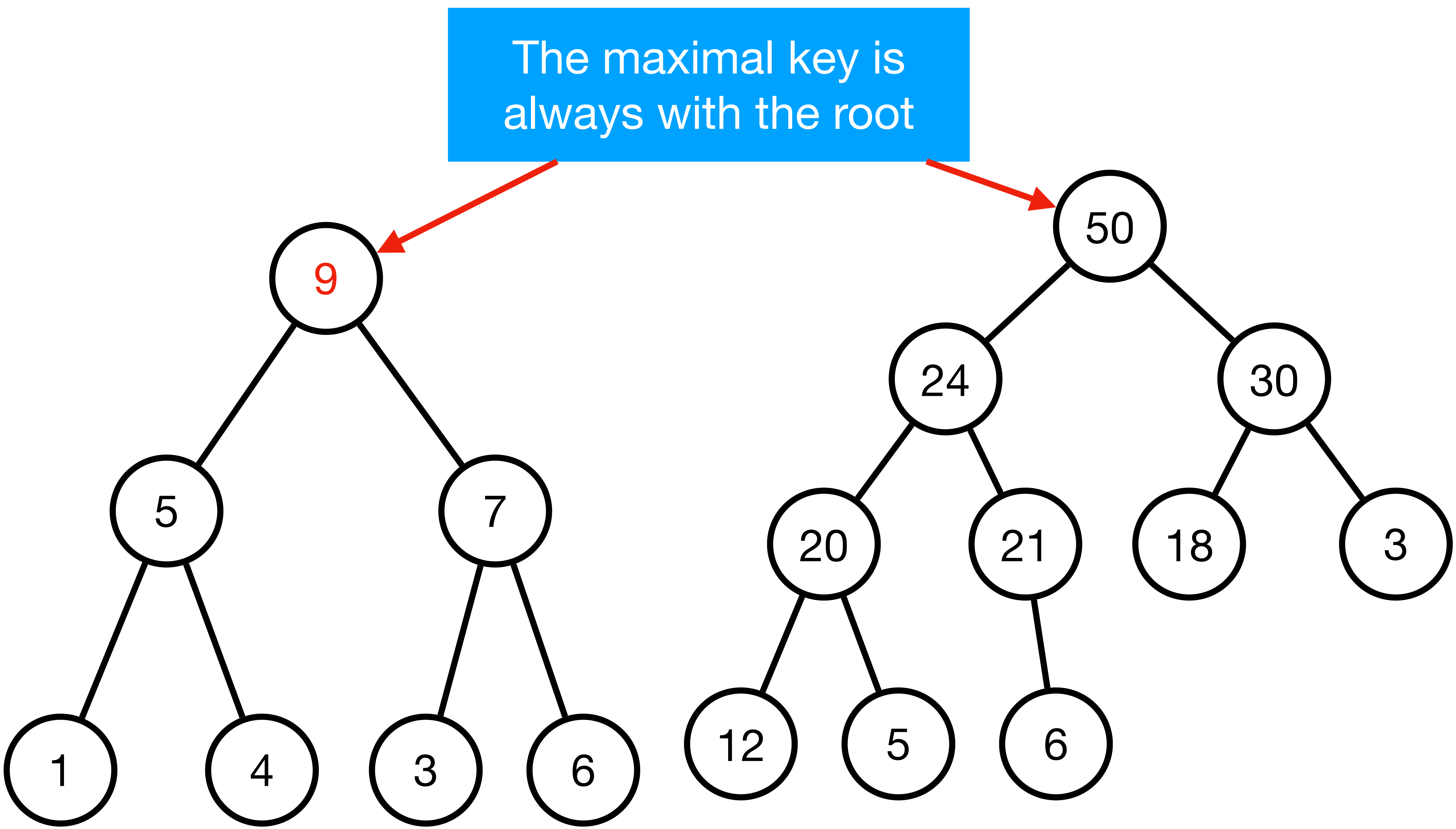
- "FIFO" in some special sense. The "first" means some kind of "priority", such as value (largest or smallest)
  - **PriorityQ** create()
    - Precondition: none
    - Postconditions: If pq=create(), then, pq refers to a newly created object and isEmpty(pt)=**true**
  - **boolean** isEmpty(PriorityQ pq)
    - precondition: none
  - **int** getMax(PriorityQ pq)
    - precondition: isEmpty(pq)=**false**
    - postconditions: **
  - **void** insert(PriorityQ pq, **int** id, **float** w)
    - preconditions: none
    - postconditions: isEmpty(pq)=**false**; **
  - **void** delete(PriorityQ pq)
    - precondition: isEmpty(pq)=**false**
    - postconditions: value of isEmpty(pq) updated; **
  - **void** increaseKey(PriorityQ pq, **int** id, **float** newKey)

**pq can always be thought as a sequence of pairs (id_i, w_i), in non-decreasing order of w_i

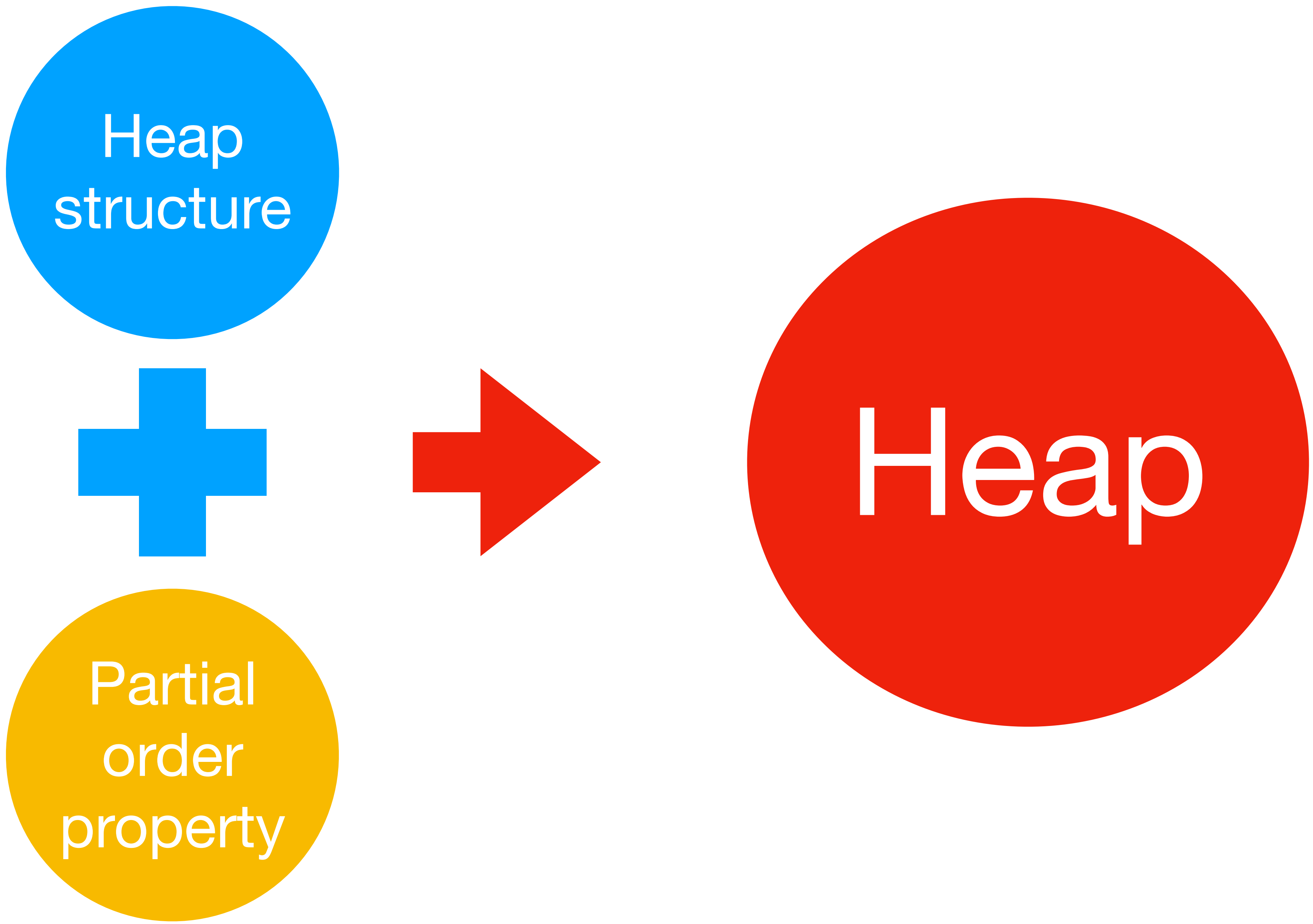
Heap: an Implementation of Priority Queue

- **A binary tree T is a heap structure if:**
  - T is complete at least though depth h-1
  - All leaves are at depth h or h-1
  - All paths to a leaf of depth h are to the left of all paths to a leaf of depth h-1
- **Partial order tree property**
  - A tree T is a (maximizing) partial order tree if and only if the key at any node is greater than or equal to the keys at each of its children (if it has any).

Heap: Examples



Heap: an Implementation of Priority Queue



HeapSort: the Strategy

```
heapSort(E, n)
    Construct H from E, the set of n elements to be sorted;
    for (i = n; i ≥ 1; i -=1){
        curMax = getMax(H);
        deleteMax(H);
        E[i] = curMax;
    }

deleteMax(H)
    copy the rightmost element on the lowest level of H into K;
    Delete the rightmost element on the lowest level of H;
    fixHeap(H, K);
```

FixHeap

- **Input: A nonempty binary tree H with a “vacant” root and its two subtrees in partial order. An element K to be inserted.**

- **Output: H with K inserted and satisfying the partial order tree property.**

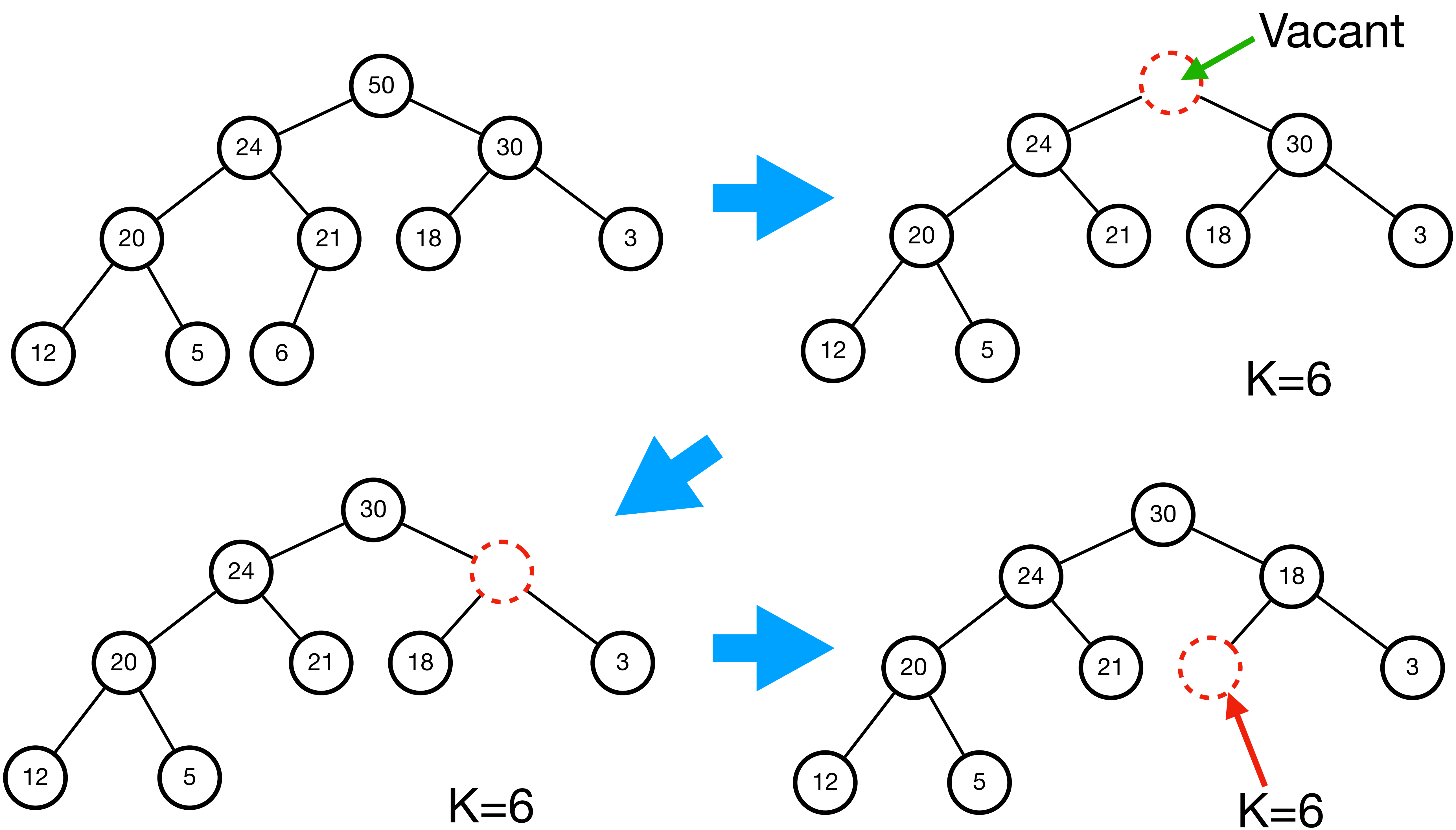
- **Procedure:**

```
fixheap(H, K)
    if(H is a leaf) insert K in root(H);
    else
        set largerSubHeap;
        if(K.key≥root(largerSubHeap).key)
            insert K in root(H);
        else
            insert root(largerSubHeap) in root(H);
            fixHeap(largerSubHeap, K);
    return;
```

One comparison:
largerSubHeap is left- or right-Subtree(H), the one with larger key at its root.
Special case: rightSubtree is empty.

“Vacant” moving down

FixHeap: an Example



Worst Case Analysis for fixHeap

- 2 comparisons at most in one activation of the procedure
- The tree height decreases by one in the recursive call
- So, 2h comparisons are needed in the worst case, where h is the height of the tree
- **Procedure:**

```
fixheap(H, K)
    if(H is a leaf) insert K in root(H);
    else
        set largerSubHeap;
        if(K.key≥root(largerSubHeap).key)
            insert K in root(H);
        else
            insert root(largerSubHeap) in root(H);
            fixHeap(largerSubHeap, K);
    return;
```

One comparison:
largerSubHeap is left- or right-Subtree(H), the one with larger key at its root.
Special case: rightSubtree is empty.

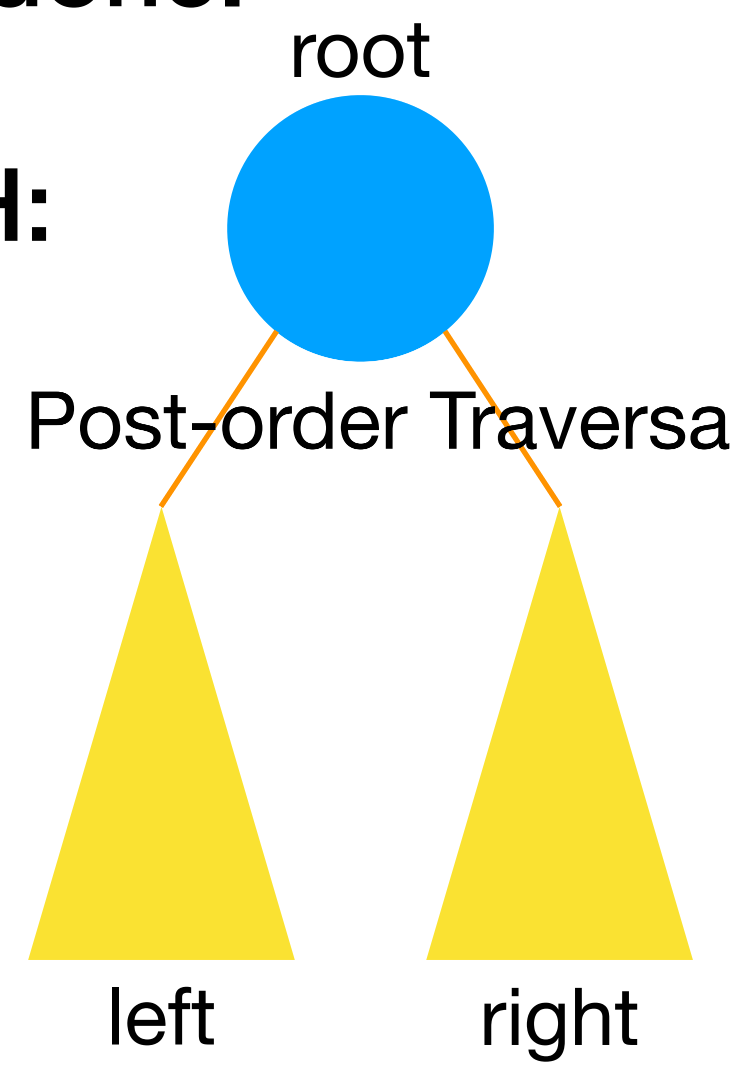
recursion

"Vacant" moving down

Heap Construction

- **Note: if left subtree and right subtree both satisfy the partial order tree property, then fixHeap(H, root(H)) gets the thing done.**

- **We begin from a Heap Structure H:**




```
void constructHeap(H)
    if(H is not a leaf)
        constructHeap(left subtree of H);
        constructHeap(right subtree of H);
        Element K = root(H);
        fixHeap(H, K);
    return;
```

Correctness of constructHeap

- **Specification**
  - Input: A heap structure H, not necessarily having the partial order tree property.
  - Output: H with the same nodes rearranged to satisfy the partial order tree property.

```
void constructHeap(H)
  if(H is not a leaf)
    constructHeap(left subtree of H);
    constructHeap(right subtree of H);
    Element K = root(H);
    fixHeap(H, K);
  return;
```

H is a leaf: base case, satisfied trivially.

Preconditions hold respectively?

Postcondition of constructHeap satisfied?

Linear Time Heap Construction!

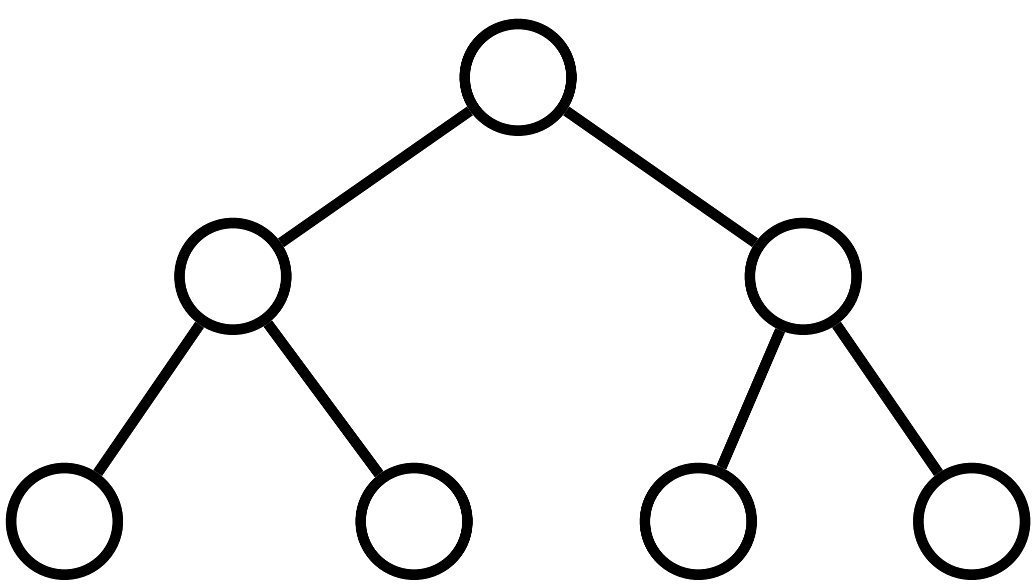
- **The recursion equation:**
  $W(n) = W(n-r-1) + W(r) + 2\log n$
- **A special case: H is a complete binary tree:**
  - The size N=2^d-1,
    (then, for arbitrary n, N/2<n<N≤2n, so W(n)≤W(N)≤W(2n))
  - Note: $W(N) = 2W((N-1)/2) + 2\log N$
  - The Master Theorem applies, with b=c=2, and the critical exponent E=1, $f(N) = 2\log N$
  - Note: $\lim_{N\to\infty} \frac{2\log N}{N^{1-\epsilon}} = \lim_{N\to\infty} \frac{2\log N}{N^{1-\epsilon}\log 2} = \lim_{N\to\infty} \frac{2N^{\epsilon}}{((1-\epsilon)\log 2)N}$
  - When 0<ε<1, this limit is equal to zero
  - So, $2\log N \in O(N^{E-\epsilon})$, case 1 satisfied, we have $W(n) \in \Theta(N)$

Direct Analysis of Heap construction

- **Heap construction**

$$\textbf{cost} = \sum_{h=0}^{\lfloor \log n \rfloor} n \frac{O(h)}{2^{h+1}} = O(n)$$

  - From recursion to iteration
  - Sum of row sums



c = logn fix; h = logn; # = 1

c = 2 fix; h = 2; # = n/8

c = 1 fix; h = 1; # = n/4

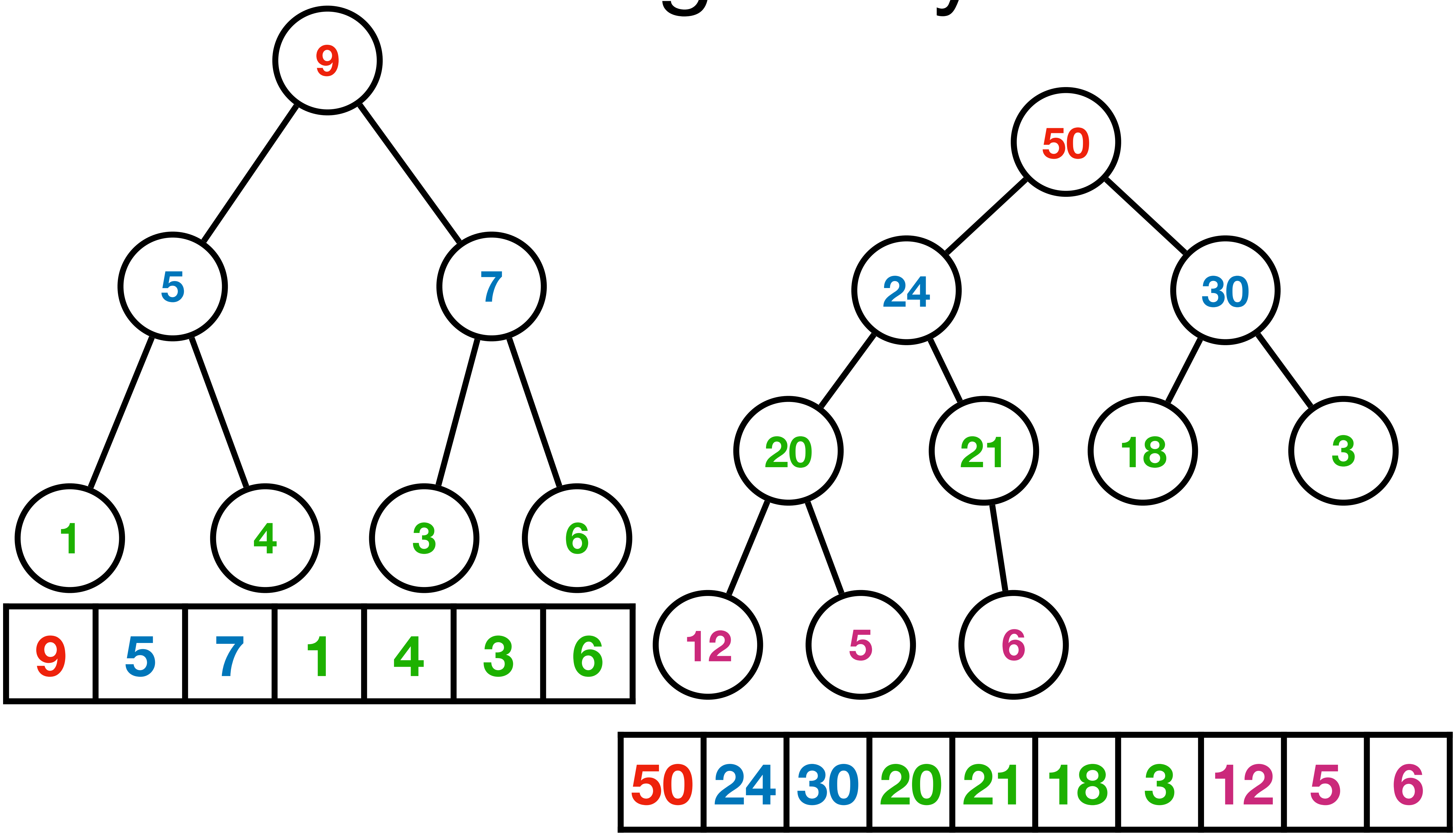
c = 0 fix; h = 0; # = n/2

1 fix = 2 comparisons

Understanding the Heap

- **Where is the k^{th} element in the heap?**
  - 1^{st}? 2^{nd}? 3^{rd}?
  - k^{th}? at what cost?
- **Sum of heights**
  - At most n-1
  - When the sum reaches n-1?

Implementing Heap Using Array



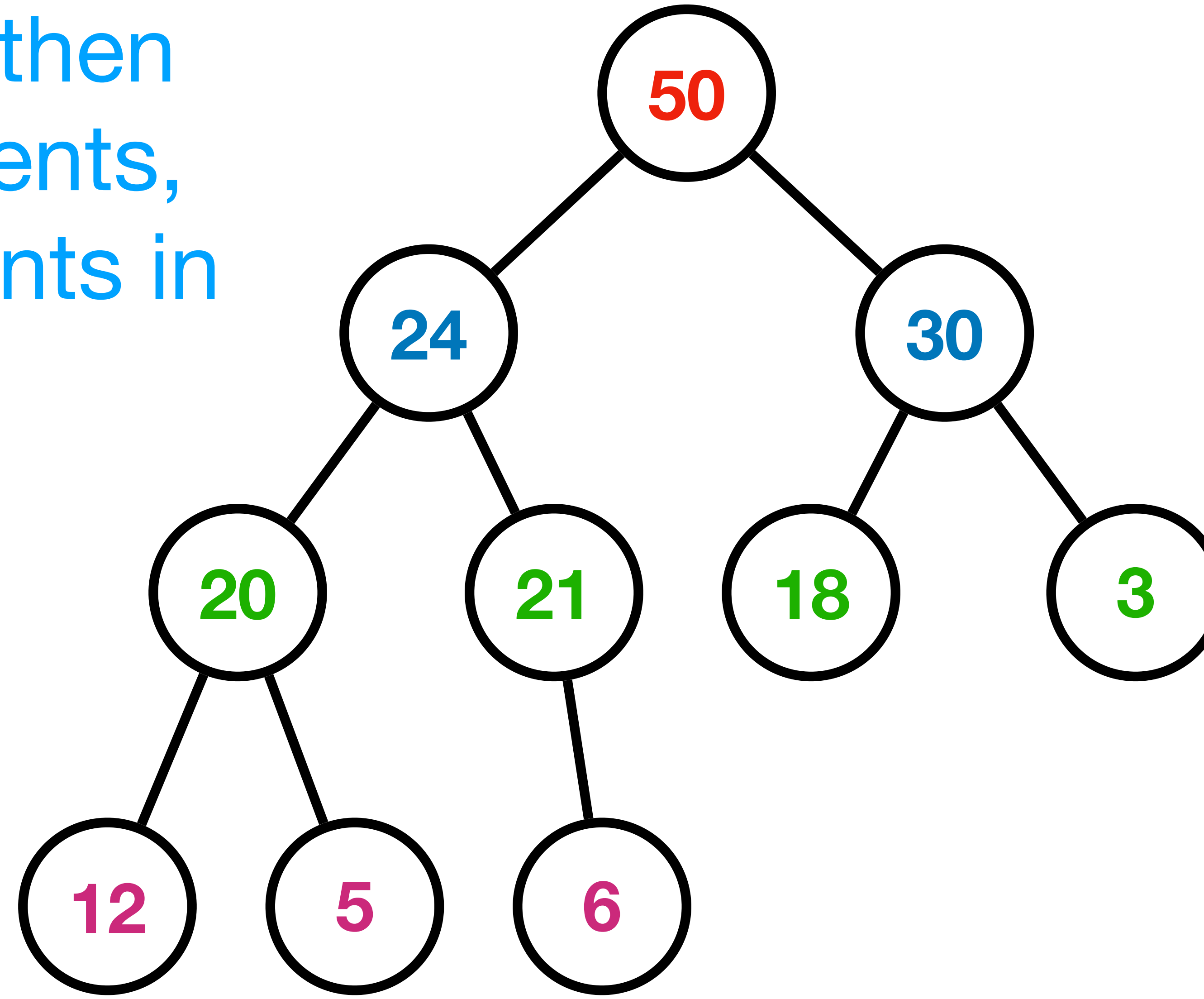
Looking for the Children Quickly

Starting from 1, not zero, then the j th level has 2^{j-1} elements, and there are $2^{j-1}-1$ elements in the proceeding j-1 levels altogether.

So, if E[i] is the kth element at level j, then $i=(2^{j-1}-1)+k$, and the index of its left child (if existing) is

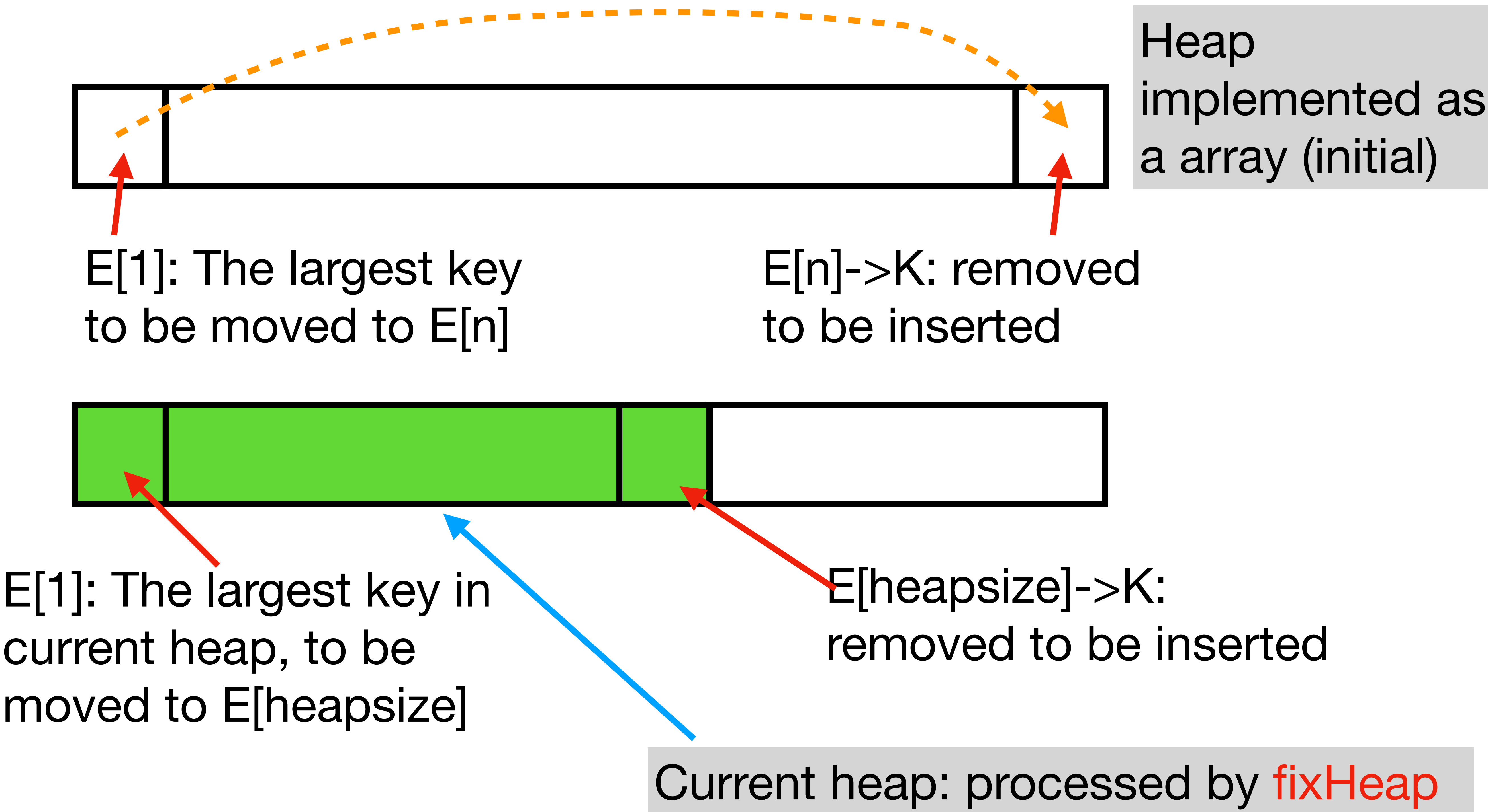
$i+(2^{j-1}-k)+2(k-1)+1=2i$

The number of node on the right of E[i] on level j **The number of children of the nodes on level j on the left of E[i]**



50	24	30	20	21	18	3	12	5	6

HeapSort: In-Space Implementation



FixHeap: Using Array

- void fixHeap(Element[ ] E, int heapSize, int root, Element K)
-     int left = 2 * root; right = 2 * root + 1;
-     if(left > heapSize) E[root] = K; // root is a leaf.
-     else
-         int largerSubHeap; // right or left to filter down.
-         if(left == heapSize) largerSubHeap = left; // no right SubHeap;
-         else if(E[left].key > E[right].key) largerSubHeap = left;
-         else largerSubHeap = right;
-         if(K.key ≥ E[largerSubHeap].key) E[root] = K;
-         else E[root] = E[largerSubHeap]; // vacant filtering down one level.
-         fixHeap(E, heapSize, largerSubHeap, K);
-     return;

HeapSort: the Algorithm

- **Input: E, an unsorted array with n (>0) elements, indexed from 1**
- **Sorted E, in non-decreasing order**
- **Procedure:**

“array version”

```
void heapSort(Element[ ] E, int n)
    int heapsize;
    constructHeap(E, n, root);
    for(heapsize = n; heapsize ≥ 2; heapsize -= 1)
        Element curMax = E[1];
        Element K = E[heapsize];
        fixHeap(E, heapsize - 1, 1, K);
        E[heapsize] = curMax;
    return;
```

Worst Case Analysis of HeapSort

- **We have:** $W(n) = W_{cons}(n) + \sum_{k=1}^{n-1} W_{fix}(k)$
- **It has been shown that:**

$W_{cons}(n) \in \Theta(n)$ **and** $W_{fix}(k) \leq 2\log k$

- **Recall that:**

$$2\sum_{k=1}^{n-1} \lceil \log k \rceil \leq 2\int_{1}^{n} \log e \ln x dx = 2\log e(n\ln n - n) = 2(n\log n - 1.443n)$$

- **So,** $W(n) \leq 2n\log n + \Theta(n)$**, that is** $W(n) \in \Theta(n\log n)$

Coefficient doubles that of mergeSort approximately

HeapSort: the Right Choice

- For heapSort, $W(n) \in \Theta(n \log n)$
- Of course, $A(n) \in \Theta(n \log n)$
- More good news: HeapSort is an in-space algorithm (using iteration instead of recursion)
- It will be more competitive if only the coefficient of the leading term can be decreased to 1

Number of Comparisons in fixHeap

Procedure:

fixHeap(H, K)

if(H is a leaf) insert K in root (H);

else

Set **largerSubHeap**;

if(K.key ≥ root(largerSubHeap).key) insert K in root(H)

else

insert root(largerSubHeap) in root(H);

fixHeap(largerSubHeap, K);

return

2 comparisons are done in filtering down for one level.

A One-Comparison-per-Level Fixing

Bubble-Up Heap Algorithm:

void bubbleUpHeap(Element [ ]E, **int** root, Element K, int **vacant**)

if(vacant == root) E[vacant] = K;

else

int parent = vacant/2;

if(K.key ≤ E[parent].key) E[vacant] = K;

else

E[vacant] = E[parent];

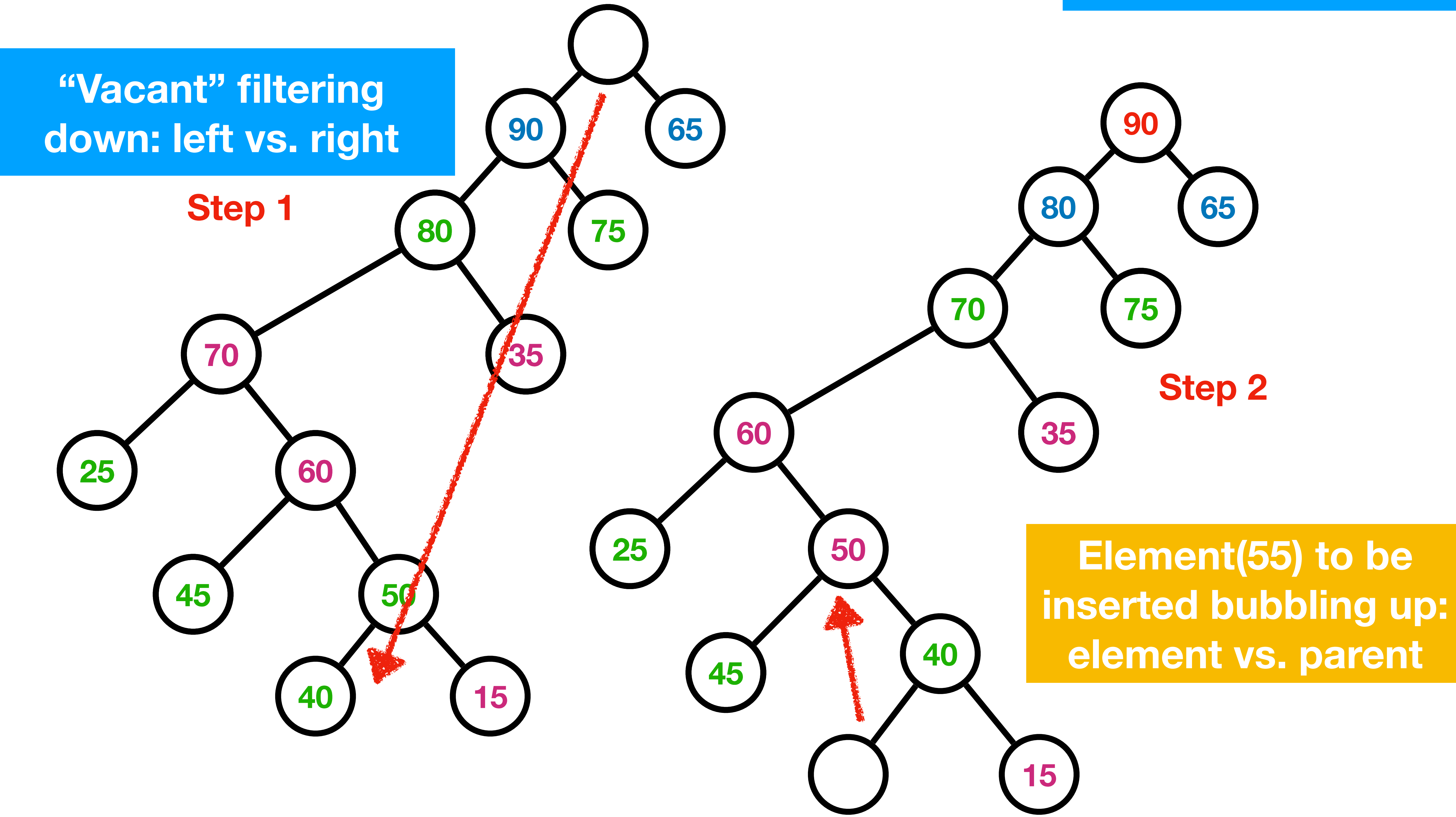
bubbleUpHeap(E, root, K, parent);

return

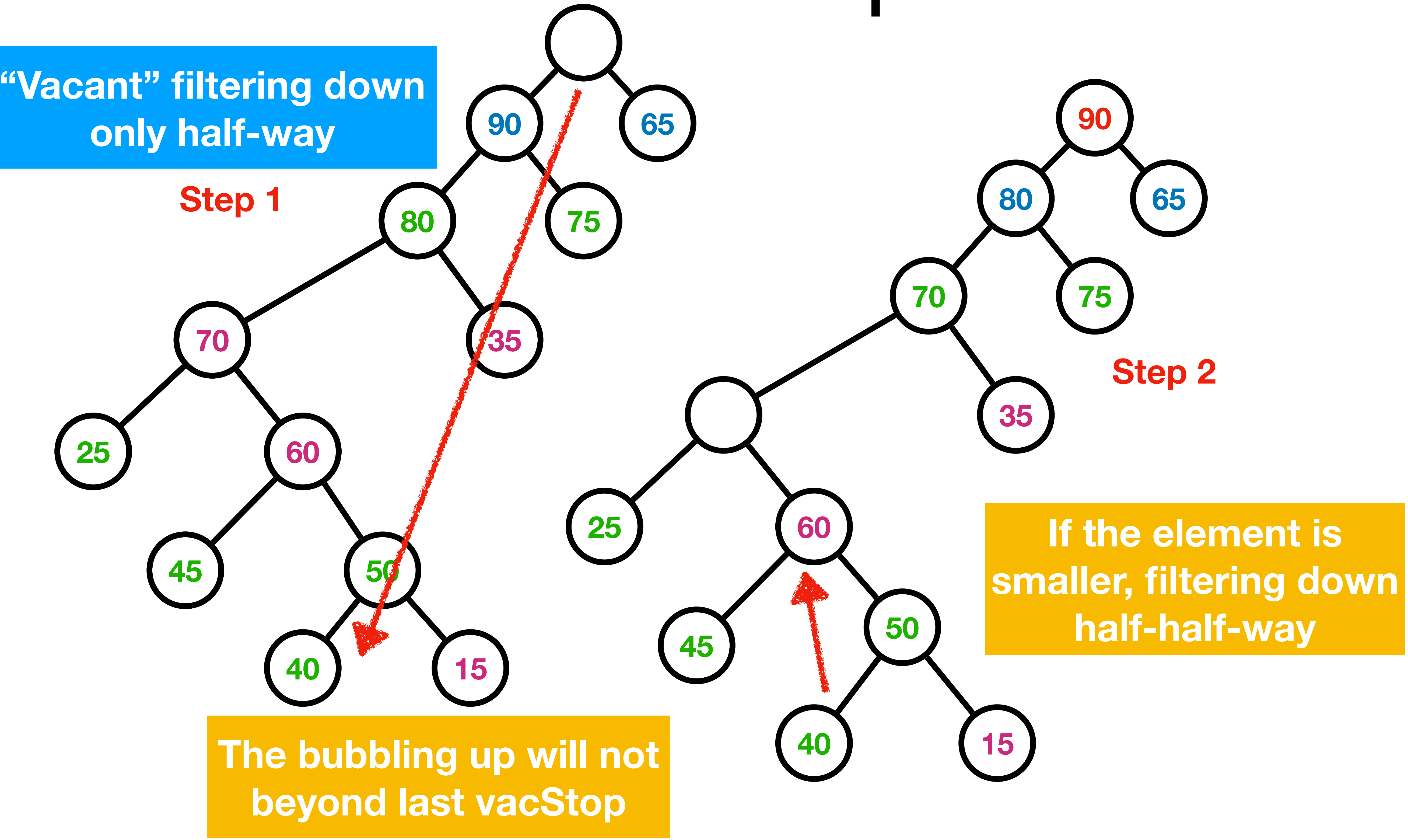
Bubbling up from vacant through to the root, recursively

Risky FixHeap

In fact, the “risk” is no more than “no improvement”



Improvement by Divide-and-Conquer



Depth Bounded Filtering Down

```
int promote(Element [ ]E, int hStop, int vacant, int h)
    int vacStop;
    if(h ≤ hStop) vacStop = vacant;
    else if(E[2*vacant].key ≤ E[2*vacant+1].key)
        E[vacant] = E[2*vacant + 1];
        vacStop = promote(E, hStop, 2*vacant + 1, h - 1);
    else
        E[vacant] = E[2*vacant];
        vacStop = promote(E, hStop, 2*vacant, h - 1);
    return vacStop;
```

Depth bound

FixHeap Using Divide-and-Conquer

```
void fixHeapFast(Element [ ]E, Element K, int vacant, int h)
    // h = ⌈log(n + 1)/2⌉ in uppermost call
    if(h ≤ 1) Process heap of height 0 or 1;
    else
        int hStop = h/2;
        Int vacStop = promote(E, hStop, vacant, h);
        int vacParent = vacStop/2;
        if(E[vacParent].key ≤ K.key)
            E[vacStop] = E[vacParent];
            bubbleUpHeap(E, vacant, K, vacParent);
        else
            fixHeapFast(E, K, vacStop, hStop);
```

Number of Comparisons in Accelerated FixHeap

- Moving the vacant one level up or down need one comparison exactly in promote or bubbleUpHeap.

- In a cycle, t calls of promote and 1 call of bubbleUpHeap are executed at most. So, the number of comparisons in promote and bubbleUpHeap calls are:

$$\sum_{k=1}^{t} \left\lceil \frac{h}{2^k} \right\rceil + \left\lceil \frac{h}{2^t} \right\rceil = h = \log(n+1)$$

- At most, lg(h) checks for reverse direction are executed. So, the number of comparisons in a cycle is at most h+log(h)

- So, for accelerated heapSort: $Wn(n) = n\log n + \Theta(n \log\log n)$

Recursion Equation of Accelerated heapSort

- The recurrence equation about h, which is about log(n+1)

$$T(1) = 2$$

$$T(h) = \left\lceil \frac{h}{2} \right\rceil + max\left(\left\lceil \frac{h}{2} \right\rceil, 1 + T\left(\left\lfloor \frac{h}{2} \right\rfloor \right) \right)$$

- Assuming T(h)≥h, then:

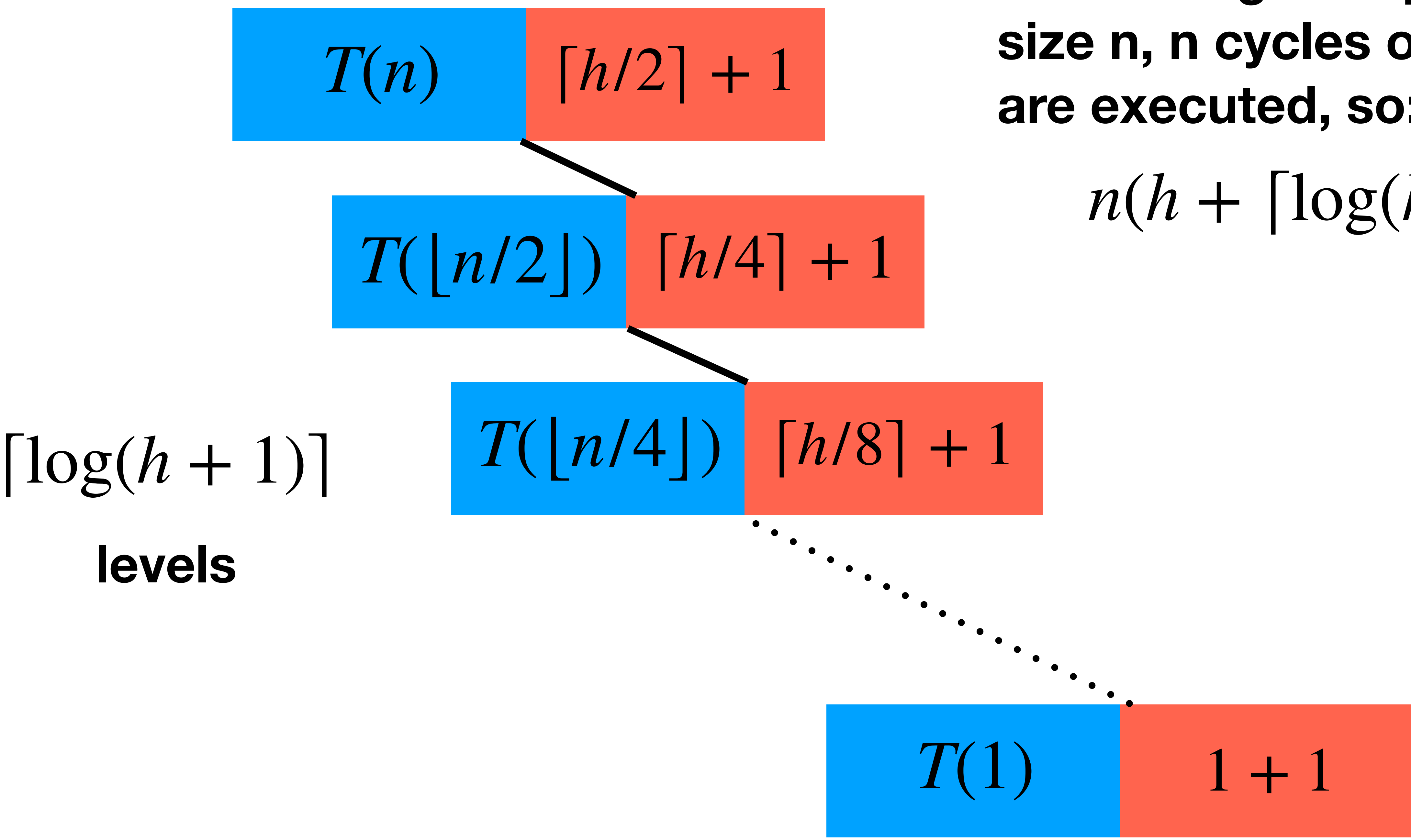
$$T(1) = 2$$

$$T(h) = \left\lceil \frac{h}{2} \right\rceil + 1 + T\left(\left\lfloor \frac{h}{2} \right\rfloor \right)$$

Solving the Recurrence Equation by Recursive Tree

For sorting a sequence of size n, n cycles of fixHeap are executed, so:

$$n(h + \lceil \log(h+1) \rceil)$$



Inductive Proof

$$T(1) = 2$$

$$T(h) = \left\lceil \frac{h}{2} \right\rceil + 1 + T\left(\left\lfloor \frac{h}{2} \right\rfloor\right)$$

- **The recurrence equation for fixHeapFast:**
- **Proving the following solution by induction:**

$$T(h) = h + \lceil \log(h+1) \rceil$$

  - According to the recurrence equation:

$$T(h+1) = \lceil (h+1)/2 \rceil + 1 + T(\lfloor (h+1)/2 \rfloor)$$

  - Applying the inductive assumption to the last term:

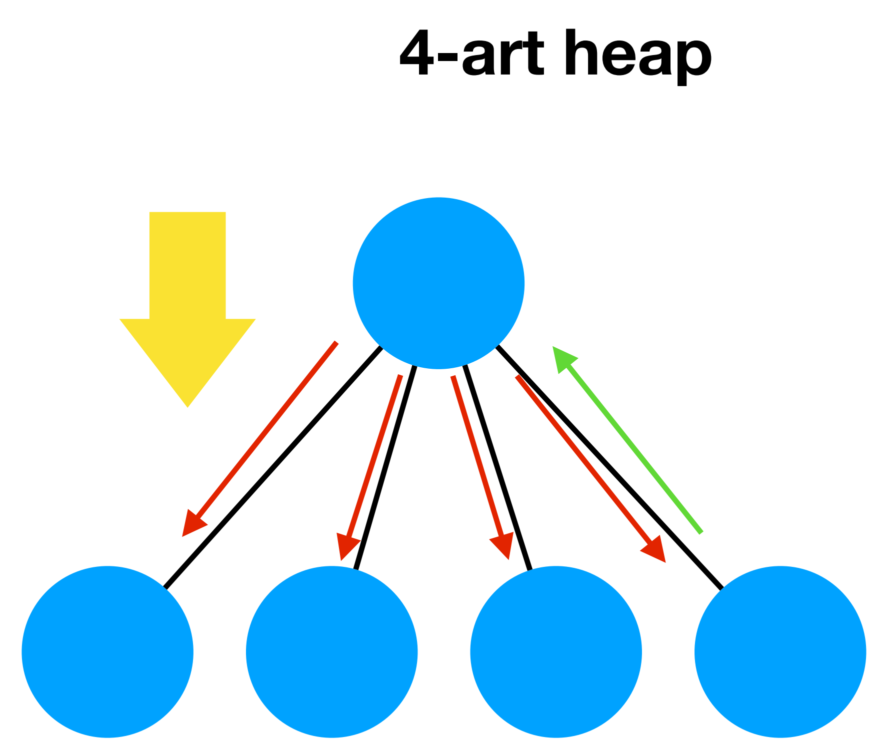
$$T(h+1) = \lceil (h+1)/2 \rceil + 1 + \lfloor (h+1)/2 \rfloor + \lceil \log(\lfloor (h+1)/2 \rfloor + 1) \rceil$$

(It can be proved that for any positive integer:
$\lceil \log(\lfloor (h)/2 \rfloor + 1) \rceil + 1 = \lceil \log(h+1) \rceil$ **)**

$Wn(n) = n \log n + \Theta(n \log \log n)$ **For Accelerated Heap**

Generalization of a Heap

- **d-ary heap**
  - Structure / partial order
- **How to choose "d"?**
  - Top-down: fix the parent node
    - Cost: d comparisons in the worst case
  - Bottom-up: fix the child node
    - Cos: always 1



Not only for Sorting

- **Eg1: how to find the k^{th} max element?**
  - The cost should be f(k)
- **Eg2: how to find the first k elements?**
  - In sorted order?
- **Eg3: how to merge k sorted lists?**
- **Eg4: how to find the median dynamically?**
- **...**

Thank you!
Q & A