Heapsort

Algorithm : Design & Analysis

[6]

In the last class…

- Mergesort
- Worst Case Analysis of Mergesort
- Lower Bounds for Sorting by Comparison of Keys
  - Worst Case
  - Average Behavior

Heapsort

- Heap Structure and Patial Order Tree Property
- The Strategy of Heapsort
- Keep the Partial Order Tree Property after the maximal element is removed
- Constructing the Heap
- Complexity of Heapsort
- Accelerated Heapsort

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(pq)=**true**
  - **boolean** isempty(PriorityQ pq)
    - precondition: none
  - **int** getMax(PriorityQ pq)
    - precondition: isEmpty(pq)=**false**
    - postconditions: **
  - **void** insert(PriorityQ pq, **int** id, **float** w)
    - precondition: none
    - postconditions: isEmpty(pq)=**false; ****
  - **void** delete(PriorityQ pq)
    - precondition: isEmpty(pq)=**false**
    - postconditions: value of isEmpty(pq) updated; **

**

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 Quere

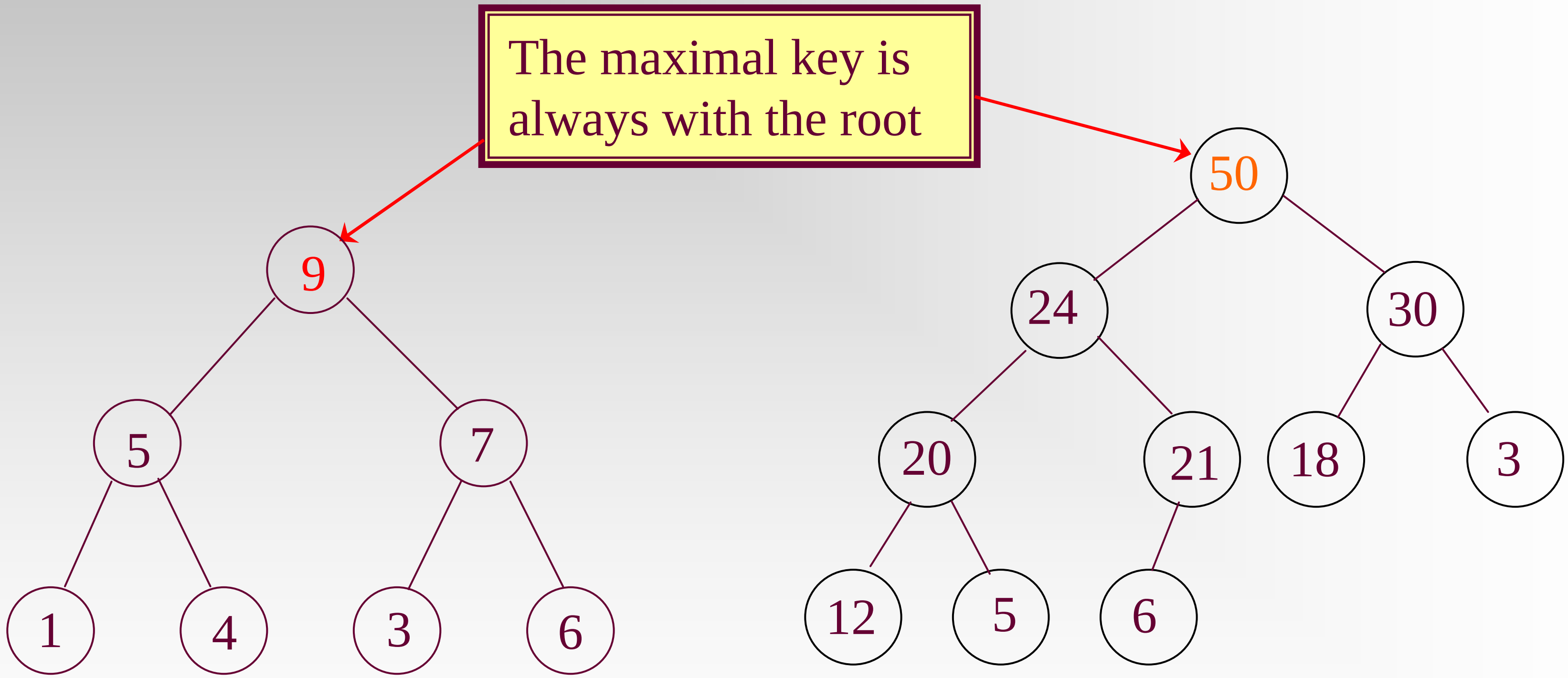
- A binary tree *T* is a **heap structure** if:
  - *T* is complete at least through depth *h*-1
  - All leaves
  - All path to ... all path to a leaf of dep
- Partial order t
  - A tree *T* is a (maximizing) **partial order tree** if and only if the key at any node is greater than or equal to the keys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

A heap is:

A heap structure, satisfying

Partial order tree property

Heap: Examples



Heapsort: the Strategy

heapSort(E,n)

Construct H from E, the set of n elements to be sorted;

for (i=n;i≥1;i--)

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: Keeping the Partial Order Tree Property

- 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:

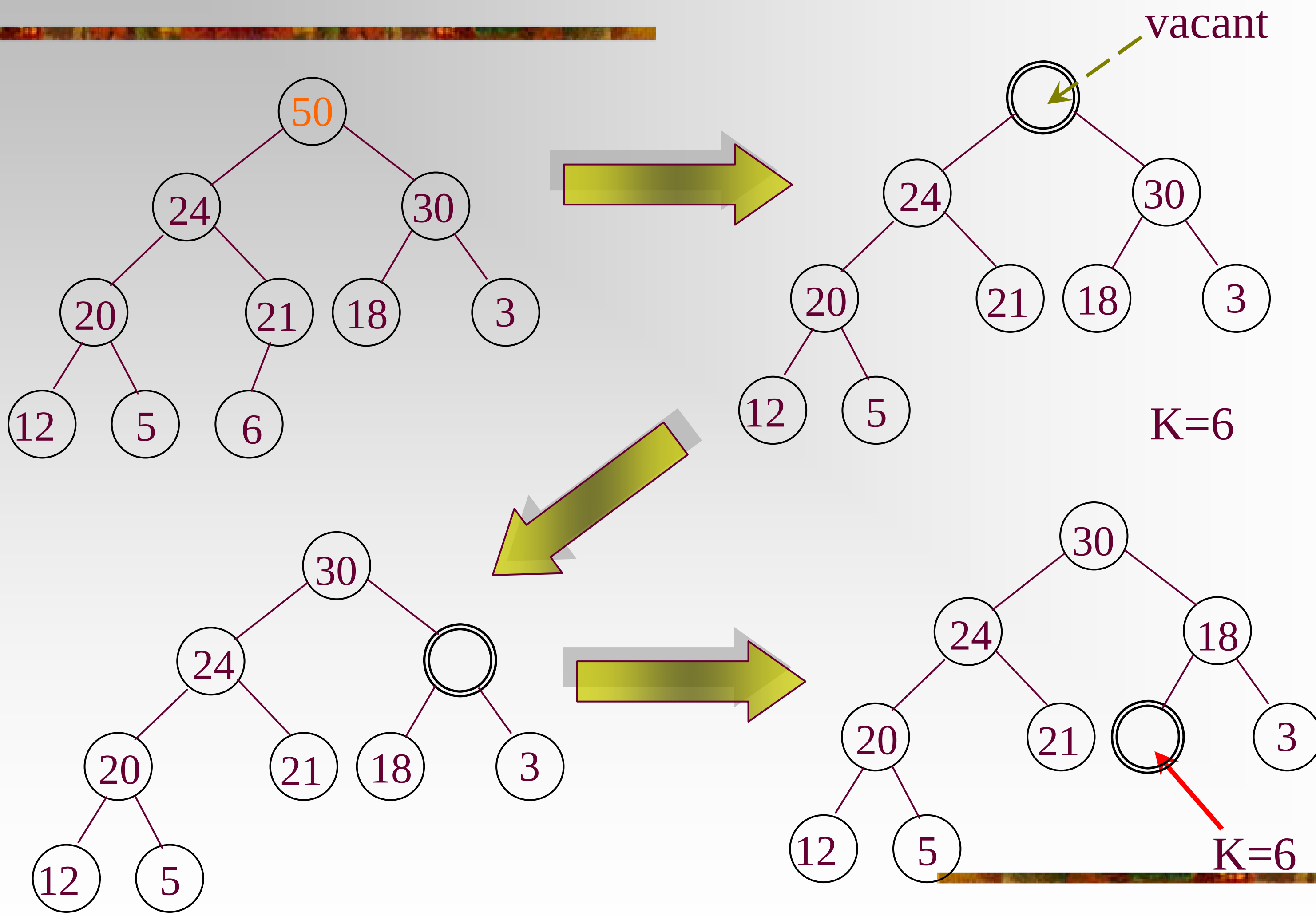
fixHeap(H,K)
    if (H is a leaf) insert K in root(H);
    else
        Set ***largerSubHeap***;
        if (K.key≥root(largerSubHeap).key) insert K in root(H)
        else
            insert root(largerSubHeap) in root(H);
            fixHeap(largerSubHeap, K);
    return

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

Recursion

“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

- Procesure:

  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

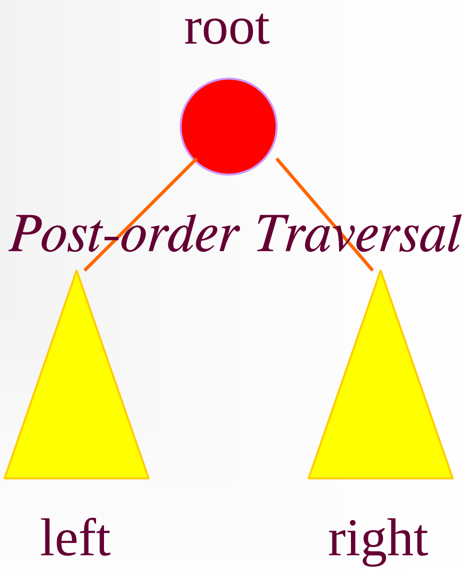
Recursion

“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?

The Heap Is Constructed in Linear Time!

- The recursion equation:

  W(n)=W(n-r-1)+W(r)+2lg(n)

  Number of nodes in right subheap

  Cost of fixHeap

- A special case: *H* is a complete binary tree:
  - The size $N=2^d-1$,

    (then, for arbitrary

  - Note: W(*N*)=
  - The **Master** T

    exponent E=1, *f*(*N*)=2lg(*N*)

  - Note: $$\lim_{N\to\infty}\frac{2\lg(N)}{N^{1-\varepsilon}}=\lim_{N\to\infty}\frac{2\ln N}{N^{1-\varepsilon}\ln 2}=\lim_{N\to\infty}\frac{2N^{\varepsilon}}{((1-\varepsilon)\ln 2)N}$$

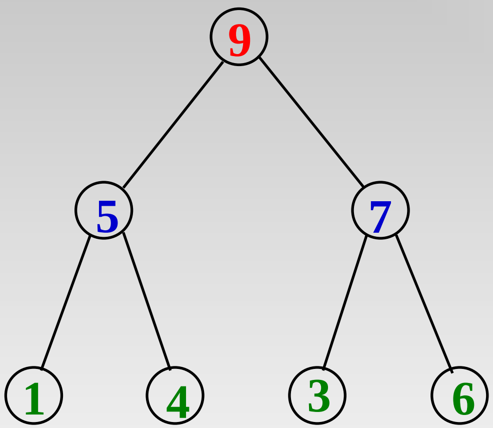
    L'Hôpital's Rule

  - When 0<ε<1, this limit is equal to zero
  - So, $2\lg(N)\in O(N^{E-\varepsilon})$, case 1 satisfied, we have $W(N)\in\Theta(N)$, so, $\boldsymbol{W(n)\in\Theta(n)}$

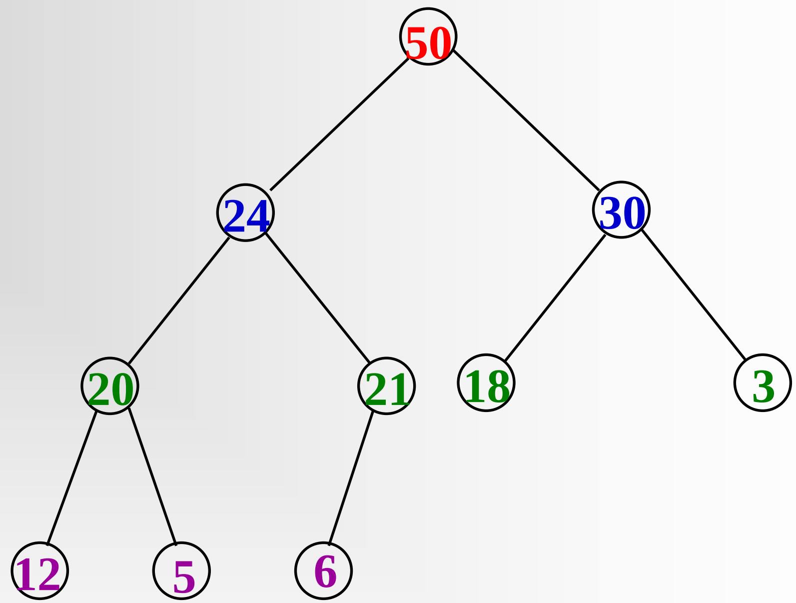
For your reference:

Master Theorem – case 1: If $f(n)\in O(n^{E-\varepsilon})$ for some positive ε, then $T(n)\in\Theta(n^E)$

Implementing Heap Using Array



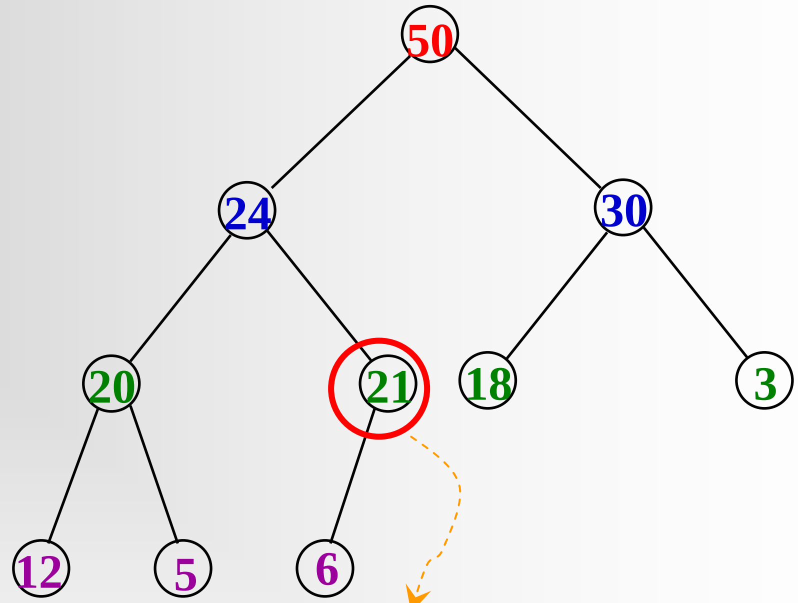
9	5	7	1	4	3	6



50	24	30	20	21	18	3	12	5	6

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 *i*-1 levels altogether.



50	24	30	20	21	18	3	12	5	6

<5> <10>

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

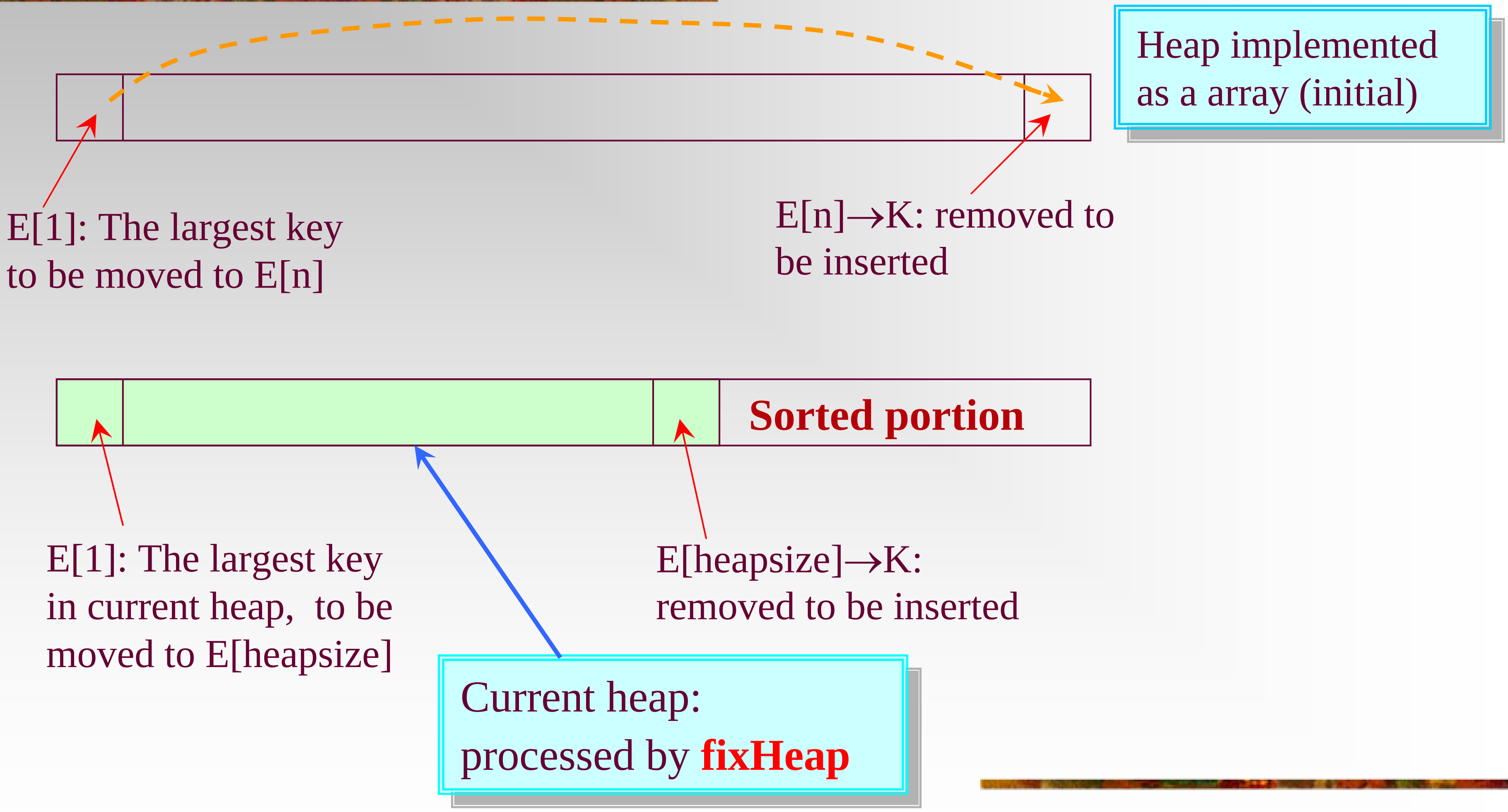
$$i+(2^{j-1}-k)+2(k-1)+1=2i$$

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]

For E[i]:
Left subheap: E[2i]
right subheap: E[2i+1]

In-space Implementation of Heapsort



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 nondecreasing order
- Procedure:

New Version

```
void heapSort(Element[] E, int n)
    int heapsize
    constructHeap(E,n,root)
    for (heapsize=n; heapsize≥2; heapsize--;)
        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 known that: $W_{cons}(n) \in \Theta(n)$ *and* $W_{fix}(k) \le 2\lfloor \lg k \rfloor$
- Recall that:

$$2\sum_{k=1}^{n-1}\lfloor \lg k \rfloor \le 2\int_1^n (\lg e)\ln x\,dx = 2(\lg e)(n\ln n - n) = 2(n\lg n - 1.443n)$$

- So, **W(n) ≤ 2*n*lg*n* + Θ(*n*)**, that is W(n)∈Θ(*n*log*n*)

Coefficient doubles that of mergeSort approximately

heapSort: the Right Choice

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

Number of Comparisons in fixHeap

Procesure:

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:

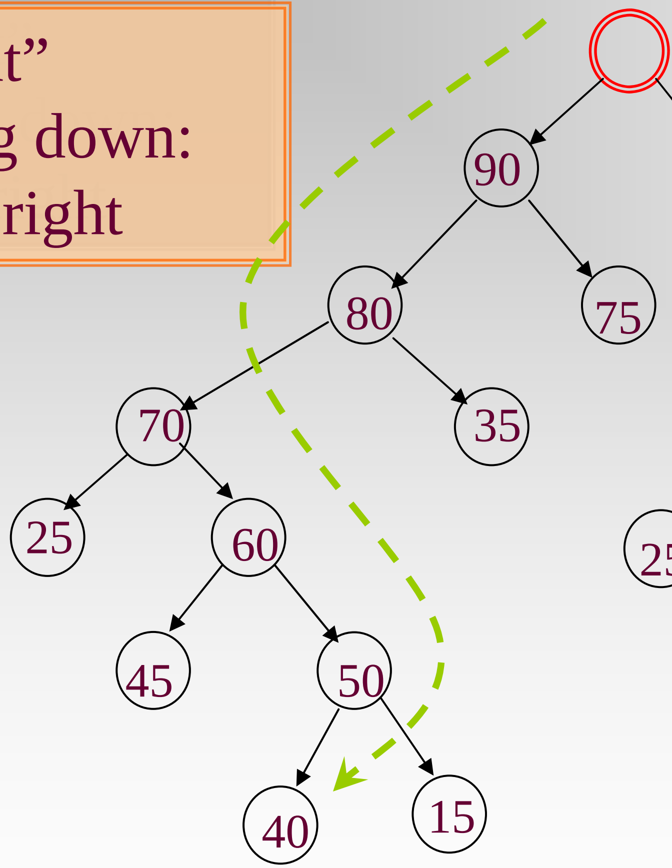
```
void bubbleUpHeap(Element []E, int root, Element K, int
    vacant)
  if (vacant==root) E[vacant]=K;
  else
    int parent=vacant/2;
    if (K.key≤E[parent].key)
      E[vacant]=K
    else
      E[vacant]=E[parent];
      bubbleUpHeap(E,root,K,parent);
```

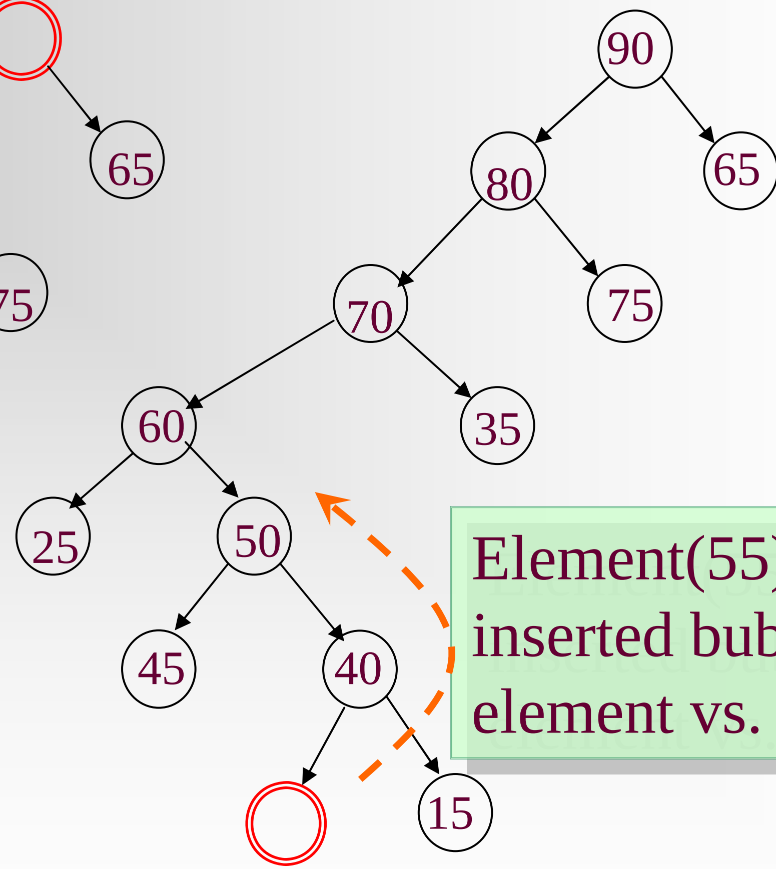
Bubbling up from vacant through to the root, recursively

Risky FixHeap

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

“Vacant” filtering down: left vs. right

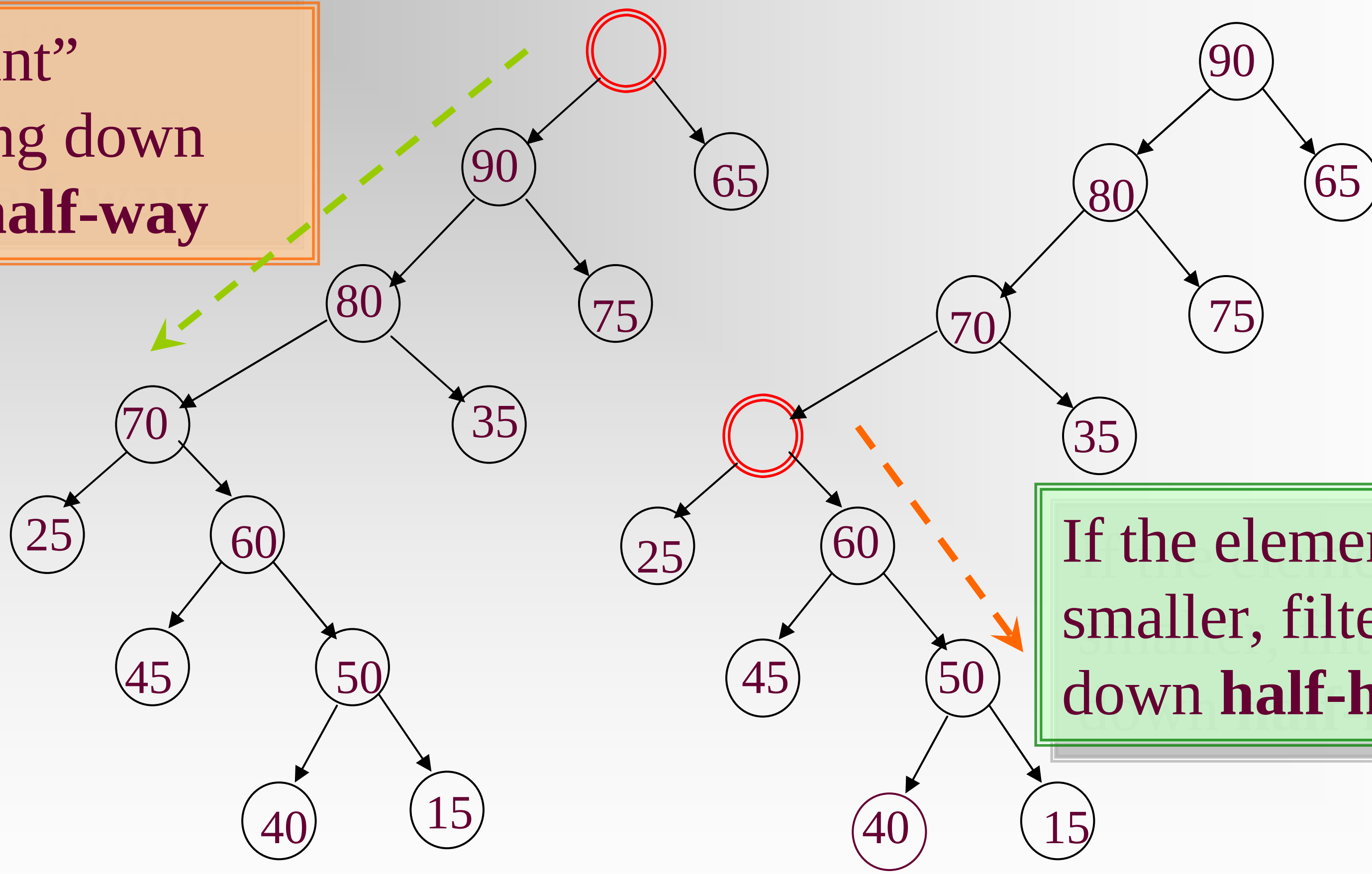




Element(55) to be inserted bubling up: element vs. parent

Improvement by Divide-and-Conquer

“Vacant” filtering down only **half-way**



If the element is smaller, filtering down **half-half-way**

The bubbling up will not beyond last vacStop

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*vacan+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=⌈lg(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 the Worst Case for 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:

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

- At most, lg(h) checks for reverse direction are executed. So, the number of comparisons in a cycle is at most h+lg(h)
- So, for accelerated heapSort: **W(*n*)=*n*lg*n*+Θ(*n*loglog*n*)**

Recursion Equation of Accelerated heapSort

The recurrence equation about *h*, which is about lg(*n*+1)

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

Assuming T(*h*)≥*h*, then:

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

Solving the Recurrence Equation by Recursive Tree

$T(n)$	$\lceil h/2 \rceil + 1$
$T(\lfloor n/2 \rfloor)$	$\lceil h/4 \rceil + 1$
$T(\lfloor n/4 \rfloor)$	$\lceil h/8 \rceil + 1$
$T(1)$	$1+1$

$\lceil \lg(h+1) \rceil$ levels

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

$$\boldsymbol{n.(h+\lceil \lg(h+1) \rceil)}$$

So, the total cost is:

$$\boldsymbol{h+\lceil \lg(h+1) \rceil}$$

Inductive Proof

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

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

$$T(\mathrm{h}) = \mathrm{h} + \lceil \lg(h+1) \rceil$$

  - 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:

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

(*It can be proved that for any positive integer*:

$\lceil \lg(\lfloor (\mathrm{h})/2 \rfloor + 1) \rceil + 1 = \lceil \lg(\mathrm{h}+1) \rceil$)

The sum is $h+1$

Home Assignment

- pp.213-
  - 4.37
  - 4.38
  - 4-39
  - 4-44