Presentation for use with the textbook Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Ch05 Priority Queues & Heapsort

1

1

Priority Queue ADT

- A priority queue stores a collection of elements which have a total order.
- Each **element** has a key value key(x).
- Main methods of the Priority Queue ADT
  - insert(x)
    inserts an entry with key k and value x
  - removeMin()
    removes and returns the element with smallest key.
- This is the min-queue. Replace “min” by “max” we obtain the max-queue.
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key
  - size()
  - isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

2

Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Every pair of such keys must be comparable according to a **total order**.
- Definition of **total order** relation $\leq$
  - **Comparability** property: either $x \leq y$ or $y \leq x$
  - **Reflexive** property: $x \leq x$
  - **Antisymmetric** property: $x \leq y$ and $y \leq x \Rightarrow x = y$
  - **Transitive** property: $x \leq y$ and $y \leq z \Rightarrow x \leq z$

3

3

Example

- A sequence of priority queue methods:

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

4

4

Priority Queue Sorting

- We can use a priority max-queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMax operations
- The running time of this sorting method depends on the priority queue implementation.

Algorithm *PQ-Sort*(*S*, *C*)
  Input sequence *S*, comparator *C* for the elements of *S*
  Output sequence *S* sorted in increasing order according to *C*
  $P \leftarrow$ priority queue with comparator *C*
  while ¬*S.isEmpty* ()
    $e \leftarrow$ *S.removeFirst* ()
    P.insert (*e*)
  while ¬*P.isEmpty*()
    $e \leftarrow$ *P.removeMax*()
    S.insertFirst(*e*)

5

Some Definitions

- Internal Sort
  - The data to be sorted is all stored in the computer's main memory.
- External Sort
  - Some of the data to be sorted might be stored in some external, slower, device.
- In Place Sort
  - The amount of extra space required to sort the data is o(n), where n is the input size.

6

Sequence-based Priority Queue

- Implementation with an unsorted list

  4 — 5 — 2 — 3 — 1

- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMax takes $O(n)$ time since we have to traverse the entire sequence to find the maximal key

- Implementation with a sorted list

  1 — 2 — 3 — 4 — 5

- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item
  - removeMax takes $O(1)$ time, since the smallest key is at the beginning

How does the Priority Queue Sorting behave?

7

Selection-Sort, Insertion-Sort

- Selection-sort is a variation of PQ-sort where the priority queue is implemented with an unsorted sequence.
  - If an array is used, it can be implemented as in-place selection sort.
- Insertion-sort is a variation of PQ-sort where the priority queue is implemented with a sorted sequence.
  - If an array is used, it can be implemented as in-place insertion sort.

8

Selection-Sort Example

	Priority Queue P	Sorted Sequence
Input:	(7,4,8,2,5,3,9)	()
removeMax():	(7,4,8,2,5,3)	(9)
removeMax():	(7,4,2,5,3)	(8,9)
removeMax():	(4,2,5,3)	(7,8,9)
removeMax():	(4,2,3)	(5,7,8,9)
removeMax():	(2,3)	(4,5,7,8,9)
removeMax():	(2)	(3,4,5,7,8,9)
removeMax():	()	(2,3,4,5,7,8,9)

9

Insertion-Sort Example

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
insert(7):	(4,8,2,5,3,9)	(7)
insert(4):	(8,2,5,3,9)	(4,7)
insert(8):	(2,5,3,9)	(4,7,8)
insert(2):	(5,3,9)	(2,4,7,8)
insert(5):	(3,9)	(2,4,5,7,8)
insert(3):	(9)	(2,3,4,5,7,8)
insert(9):	()	(2,3,4,5,7,8,9)

10

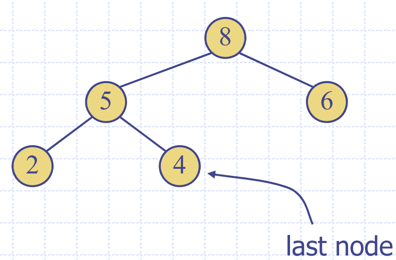
Balanced Search Tree Based Priority Queue

- Both insert and removeMax can be implemented using O(log n) time.
- Thus, PQ-sort can run in O(n log n).
- Can we have an in-place PQ-sort whose complexity is in O(n log n)?
  - Yes, use heaps for PQ.

11

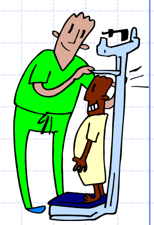
What is a heap?

- A (max) heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
  - (Max) Heap-Order: for every node v other than the root, $\boldsymbol{key}(\boldsymbol{v}) \leq \boldsymbol{key}(\boldsymbol{parent}(\boldsymbol{v}))$
  - Complete Binary Tree: let $\boldsymbol{h}$ be the height of the heap
    - for $\boldsymbol{i} = 0, \ldots, \boldsymbol{h} - 2$, there are $2^{\boldsymbol{i}}$ nodes of depth $\boldsymbol{i}$
    - at depth $\boldsymbol{h}$**-1**, the nodes are listed from left to right without gaps.
- The last node of a heap is the rightmost node of depth $\boldsymbol{h} - 1$



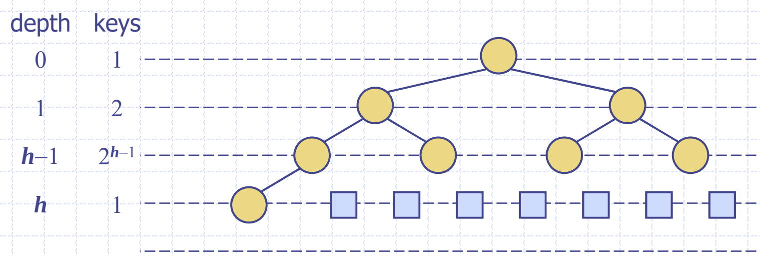
12

Height of a Heap



- Theorem: A heap storing n keys has height $O(\log n)$

  Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log n$.

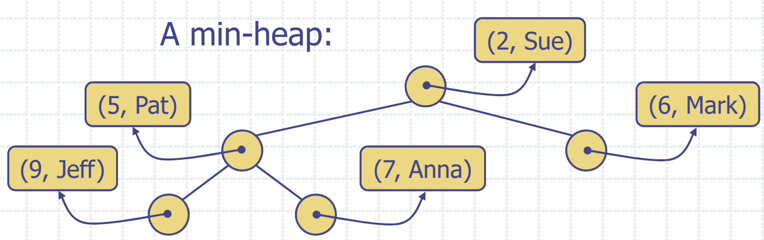


13

Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store an item (key, element) at each node
- We keep track of the position of the last node
- For simplicity, we will show only the keys in the pictures

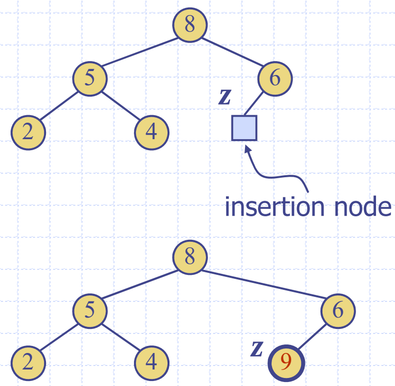
A min-heap:



14

Insert into a Heap

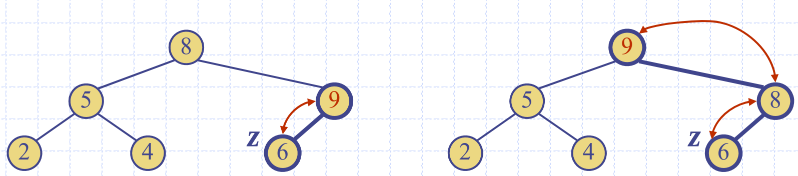
- Method insert of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the position for a new node and create a new node z
  - Store k at z
  - Restore the heap-order property by up-heap bubble (discussed next)



15

Up-Heap Bubbling

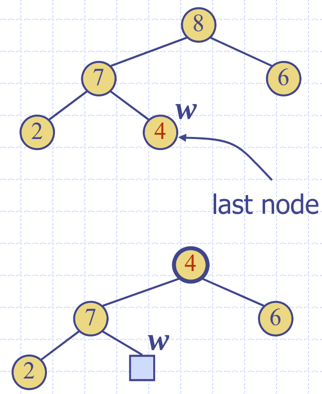
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm up-heap-bubble restores the heap-order property by swapping k along an upward path from the insertion node
- Up-heap-bubble terminates when the key k reaches the root or a node whose key is greater than or equal to k
- Since a heap has height $O(\log n)$, up-heap-bubble runs in $O(\log n)$ time



16

removeMax from a Heap

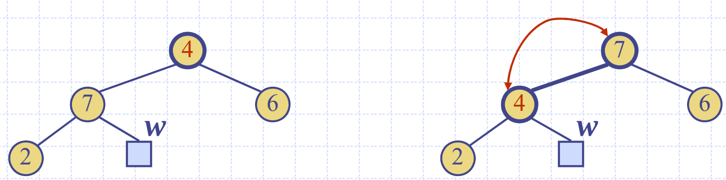
- Method removeMax of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Release node w
  - Restore the heap-order property by down-heap-bubble (discussed next)



17

Down-heap bubbling (Heapify)

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm down-heap-bubble (or heapify) restores the heap-order property by swapping key k along a downward path from the root
- Down-heap-bubble terminates when key k reaches a leaf or a node whose key is less than or equal to k
- Since a heap has height $O(\log n)$, down-heap-bubble runs in $O(\log n)$ time



18

Heap-Sort

- Consider a priority queue with n items implemented by means of a max-heap
  - The input and the heap can share the array, so the additional space used is $O(1)$
  - Methods insert and removeMax take $O(\log n)$ time.
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- It can be implemented in-place ($O(1)$ additional space).
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort, when n is very large.

19

Array-based Heap Implementation

- We can represent a heap with n keys by means of an array of length n.
- For the node at index i
  - the left child is at index $2i + 1$
  - the right child is at index $2i + 2$
- Links between nodes are not explicitly stored
- The (first portion of) input array A is used as heap.
- In-place (no additional array is needed) heap-sort:

```
For k = 1 to n-1
    A.insert(A[k]);
For k = n-1 downto 1
    A[k] = A.removeMax();
```

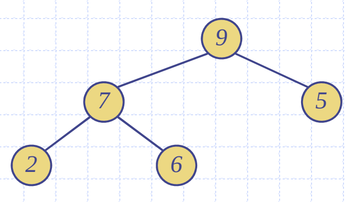
- Time Complexity: O(n log n)

Input:

0	1	2	3	4
2	6	7	9	5

Heap:

0	1	2	3	4
9	7	5	2	6



20

Possible Quiz Questions

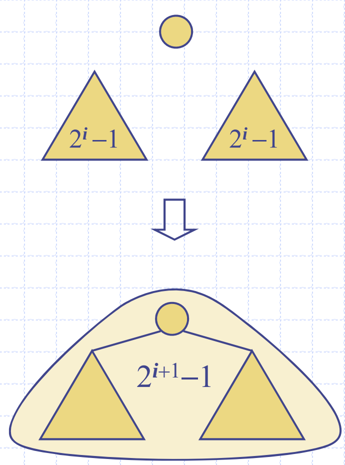
- Show the contents of the following arrays during the heap sort whenever there is a change.
  - A = [1, 2, 4, 3]
  - B = [3, 4, 2, 1]
  - C = [4, 1, 2, 3]

21

21

Bottom-up Heap Construction

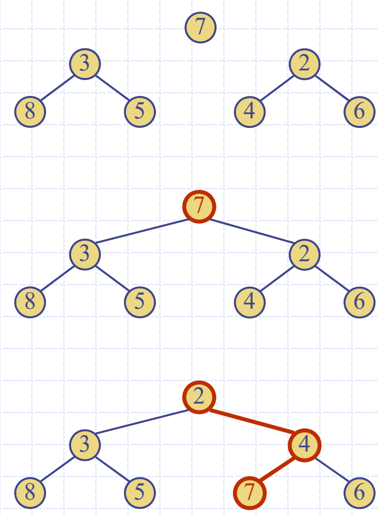
- We can construct a heap storing n given keys using a bottom-up construction with $\log n$ phases, so that the time of building a heap of n elements is O(n), instead of O(n log n).
- The process is divided into log n phases.
- In phase i, pairs of heaps with 2^i-1 keys plus one item are merged into heaps with $2^{i+1}-1$ keys



22

Merging Two (Min) Heaps

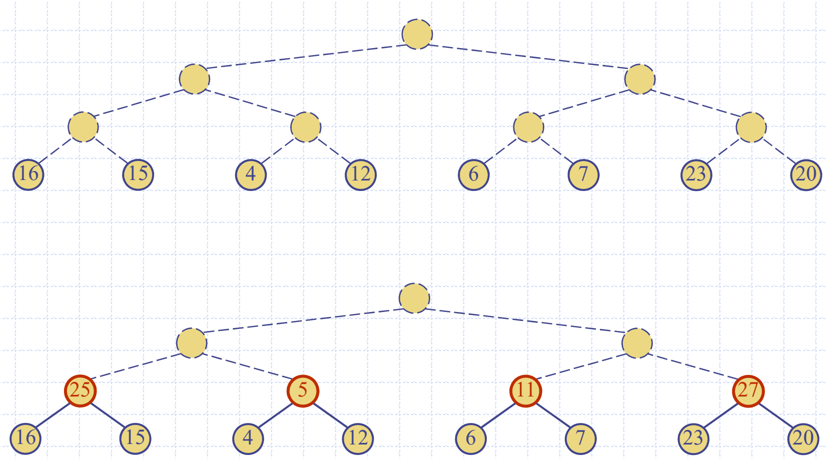
- We are given two heaps and a key ***k***
- We create a new heap with the root node storing ***k*** and with the two heaps as subtrees
- We perform down-heap-bubble to restore the heap-order property



23

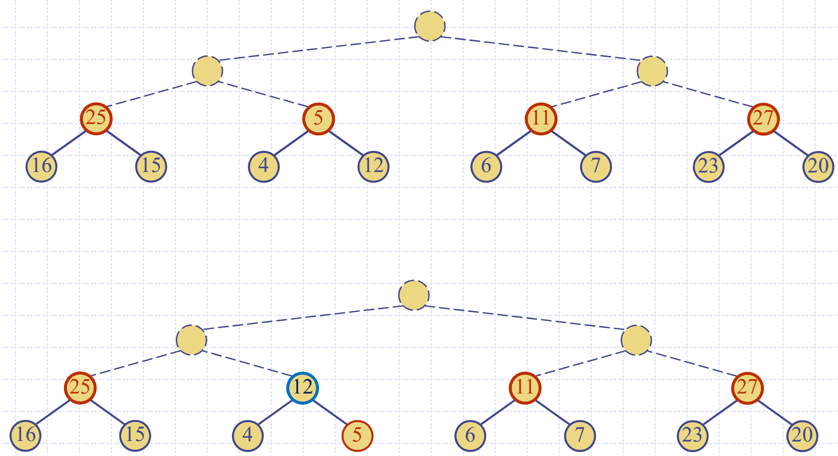
Example of Max Heap

A = [10, 7, 8, 25, 5, 11, 27, 16, 15, 4, 12, 6, 7, 23, 20]



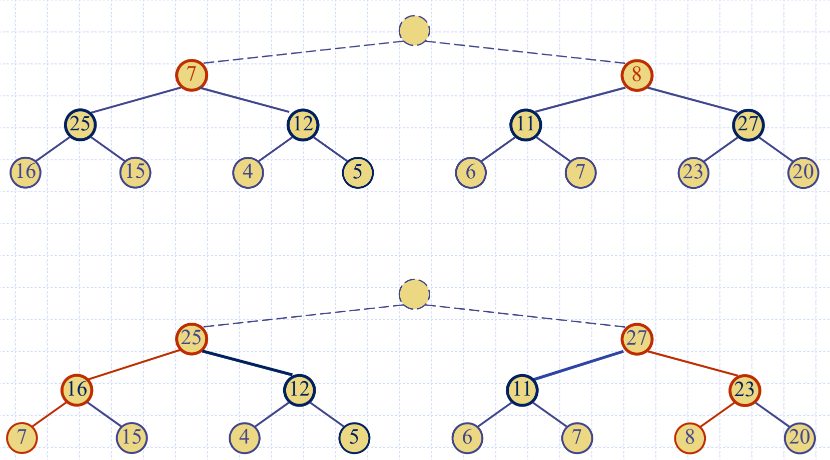
24

Example (contd.)



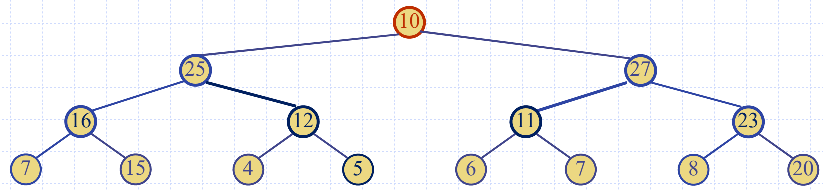
25

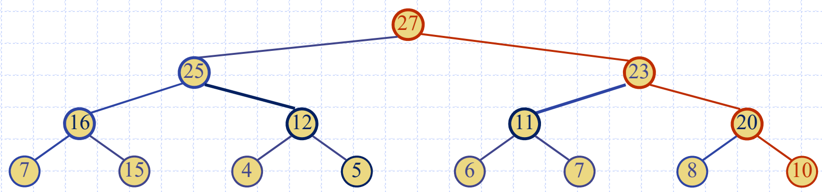
Example (contd.)



26

Example (end)





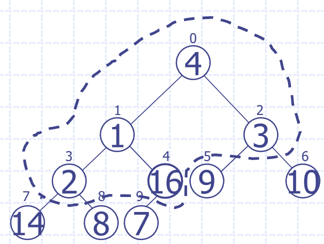
27

Building a Heap

- Convert an array $A[0 \dots n\text{-}1]$ into a max-heap (n = $length[A]$)
- The elements in the subarray $A[(\lfloor n/2 \rfloor .. n\text{-}1]$ are leaves
- Apply MaxHeapify on elements between 0 and $\lfloor n/2 \rfloor$-1

Alg: BuildMaxHep(*A*)

```
n = length[A]
 for i ← ⌊n/2⌋-1 downto 0
     do MaxHeapify(A, i, n)
```



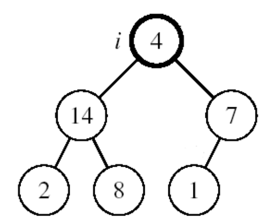
A:

4	1	3	2	16	9	10	14	8	7

28

Maintaining the Heap Property

- Assumptions:
  - Left and Right subtrees of i are max-heaps
  - $A[i]$ may be smaller than its children



Alg: MaxHeapify(*A*, i, n) {

1. l ← Left(i); // Left(i) = 2i+1
2. r ← Right(i); // Right(i) = 2i+2
3. max ← i ;
4. **if** (l < n && *A*[l] > *A*[max]) max ← l;
5. **if** (r < n && *A*[r] > *A*[max]) max ← r;
6. **if** (max ≠ i) {
7. exchange *A*[i] ↔ *A*[max];
8. MaxHeapify(*A*, max, n);
9. }}

29

Running Time of BUILD MAX HEAP

Alg: BuildMaxHeap(*A*)

1. n = length[A]
2. **for** i ← ⌊n/2⌋-1 **downto** 0
3. **do** MaxHeapify(*A*, i, n) $O(\lg n)$

$\Big\}\ O(n)$

⇒ Running time: $O(n \lg n)$

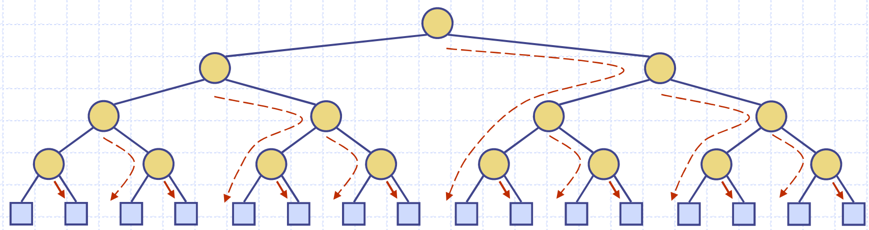
- This is not an asymptotically tight upper bound

30

30

Analysis

- We visualize the worst-case time of a heapify (or bubble-down) with a given path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual heapify path)
- Since each edge is traversed by at most once by these paths, the total length of these paths is $O(n)$.
- Thus, bottom-up heap construction runs in $O(n)$ time.
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort.



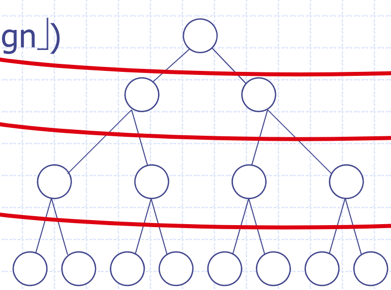
31

Running Time of BUILD MAX HEAP

- MaxHeapify takes $O(h)$ $\Rightarrow$ the cost of MaxHeapify on a node i is proportional to the height of the node i in the tree

$$\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i = \sum_{i=0}^{h} 2^i (h-i) = O(n)$$

Height	Level	No. of nodes
$h_0 = 3$ ($\lfloor \lg n \rfloor$)	i = 0	2^0
$h_1 = 2$	i = 1	2^1
$h_2 = 1$	i = 2	2^2
$h_3 = 0$	i = 3 ($\lfloor \lg n \rfloor$)	2^3



$h_i = h - i$ height of the heap rooted at level i

$n_i = 2^i$ number of nodes at level i

32

32

Running Time of BUILD MAX HEAP

$$T(n) = \sum_{i=0}^{h} n_i h_i$$ Cost of MaxHeapify at level i ∗ number of nodes at that level

$$= \sum_{i=0}^{h} 2^i (h-i)$$ Replace the values of n_i and h_i computed before

$$= \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$$ Multiply by 2^h both at the nominator and denominator and write 2^i as $\frac{1}{2^{-i}}$

$$= 2^h \sum_{k=0}^{h} \frac{k}{2^k}$$ Change variables: $k = h - i$

$$\le n \sum_{k=0}^{\infty} \frac{k}{2^k}$$ The sum above is smaller than the sum of all elements to ∞

$$= O(n)$$ The sum above is smaller than 2

Running time of BuildMaxHeap: $T(n) = O(n)$

33

33

HeapSort(A)

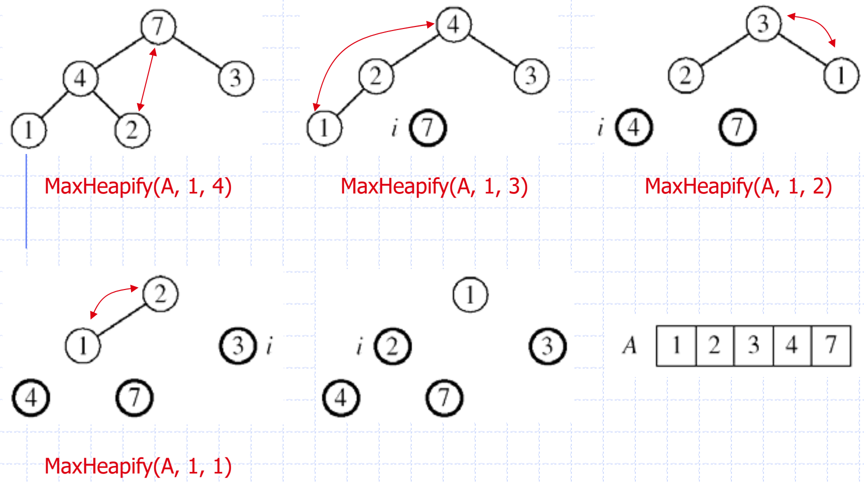
- Convert an array A[0 … n-1] into a max-heap
  - The elements in the subarray $A[\lfloor n/2 \rfloor .. n-1]$ are leaves.
  - Apply MaxHeapify on elements between 0 and $\lfloor n/2 \rfloor$-1
- Repeatedly swap the max heap element with the last unsorted element and call MaxHeapify to maintain the heap property.

Alg: HeapSort(A) {

```
    n = A.length;
    for i ← ⌊n/2⌋-1 downto 0
        MaxHeapify(A, i, n);
    for i ← n - 1 downto 1 {     // A[0..i] is a max heap
        exchange A[i] ↔ A[0];
        MaxHeapify(A, 0, i);     // A[i..n-1] is sorted with max (n - i)
}}                               // elements of the original array.
```

34

Example: A=[7, 4, 3, 1, 2]



35

35

Possible Quiz Questions

- Show the contents of the following arrays during the heap sort whenever there is a change, when the bottom-up heap construction is used.
  - A = [1, 2, 4, 3]
  - B = [3, 4, 2, 1]
  - C = [4, 1, 2, 3]

36

36

Stability

- A STABLE sort preserves relative order of records with equal keys

Sorted on first key:

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

Sort file on second key:

Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Furia	3	A	766-093-9873	22 Brown
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Gazsi	4	B	665-303-0266	113 Walker
Aaron	4	A	664-480-0023	097 Little

Records with key value 3 are not in order on first key!!

37

Summary

- A priority queue stores a collection of items
- Each item has a key value.
- Main methods of the Priority Queue ADT
  - insert(x)
    inserts an item x
  - removeMax() (or removeMin())
    removes and returns the item with max (or smallest) key
- Using an array-based priority queue, each insert and removeMax can be implemented in O(log n).
- For Heap Sort, we create an array-based max heap in O(n) and each removeMax takes O(log n), so the total time is O(n log n).
- Heap Sort is a non-stable, in-place, optimal sorting method.

38