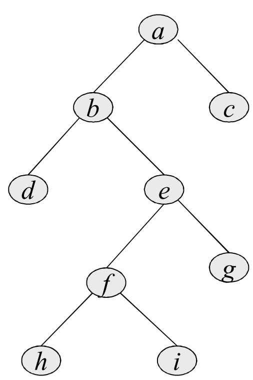
# Priority Queues

- Data type to keep track a dynamic set of elements with support for the following operations:
  - Insert($S$, $x$) : add element $x$ to $S$
  - Max($S$) : return the max of $S$
  - ExtractMax($S$) : remove max from $S$
- *Applications*: job scheduling, simulation, greedy heuristics, Huffman coding, other algorithms, etc.
- Data structure: *binary heap*
- Heap applications
  - Efficient implementation of priority queues
  - New sorting algorithm: *heapsort*
    - Runs in $O(n \log n)$ time, like mergesort
    - sorts in-place, like insertion sort

1

# Review of (Rooted) Trees

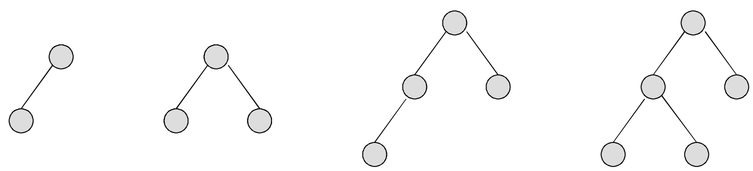



- Make sure you understand the *precise* meaning of:
  - root, leaf, internal node, sibling, parent, child, ancestor, descendant, degree, full tree, complete tree, height, depth

2

# Heaps

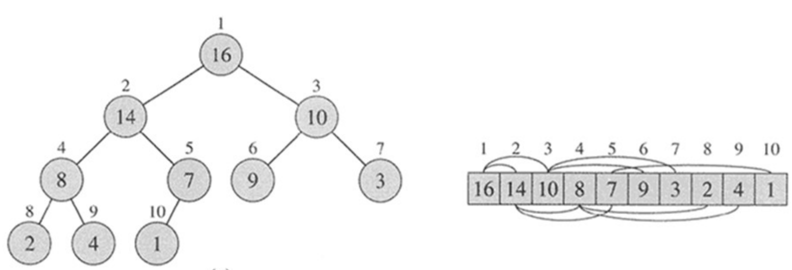
- A heap is a binary tree that satisfies two properties:
  1. *Structural property.*
     (Almost) complete binary tree
  2. *Order or heap property* (for max heaps)
     $A(\text{parent}(v)) \geq A(v)$, for all nodes $v$



3

# Array Representation of Heaps

- Tree is complete
  – All levels are full except possibly the last
  – Can represent compactly in an array




(a) (b)

- Where are the children of $A[i]$? The parent?

4

## Maintaining the heap property

- Basic operation: *Heapify*($A$, $i$)
- *Precondition*

  Subtrees rooted at *Left*($i$) and *Right*($i$) already satisfy the heap property
- *Postcondition*

  Subtree rooted at $i$ satisfies the heap property
- How do you accomplish this efficiently?

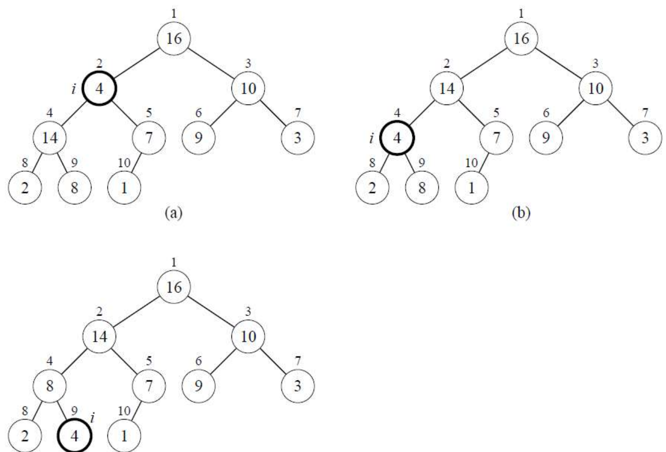


## Heapify

```
MAX-HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
   then largest ← l
   else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
   then largest ← r
if largest ≠ i
   then exchange A[i] ↔ A[largest]
        MAX-HEAPIFY(A, largest)
```

- What is the running time?

## Example: Heapify($A$, 2)



(a) (b) (c)

7

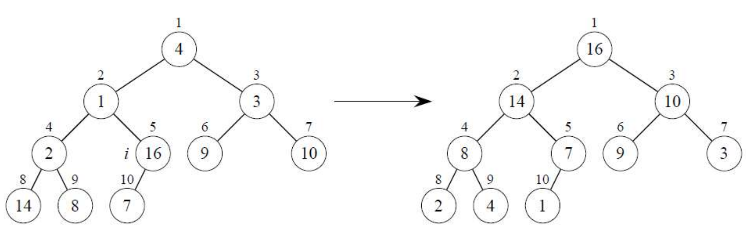
## Building A Heap

- Start with an unsorted array $A$

BUILD-MAX-HEAP($A$)

$heap\text{-}size[A] \leftarrow length[A]$
**for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
    **do** MAX-HEAPIFY($A$, $i$)

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |



8

## Correctness

- Loop invariant:

  *At the start of every iteration of the* ***for*** *loop, each node* $i + 1, i + 2, \ldots, n$ *is the root of a max-heap*
- Initialization
- Maintenance
- Termination

9

## Analysis

- A simple bound

  $O(n)$ calls to Heapify, each of which takes $O(\log n)$ time $\Rightarrow O(n \lg n)$
- A tighter bound
  - Number of nodes of height $h$ is $\leq \lceil n / 2^{h+1} \rceil$

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left( n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

  - Amortized analysis

10

## Selection Sort

- Repeatedly search through the array, finding and removing the remaining largest element

**for** $i = 1$ **to** $n$ **do**

a) Find the largest of the first $n - i + 1$ elements

b) Place it at location $n - i + 1$

- Takes $O(n(T_a + T_b))$
  - Arrays
  - Heaps

11

## Heap Sort

- Idea: after creating a max-heap, output the elements in descending order, one at a time
  1. Builds a max-heap from the array.
  2. Starting with the root, place the maximum element into the correct place in the array by swapping it with the element in the last position in the array
  3. "Discard" this last node (knowing that it is in its correct place) by decreasing the heap size
  4. Heapify on the new root
  5. Repeat "swap and discard" process until one node remains

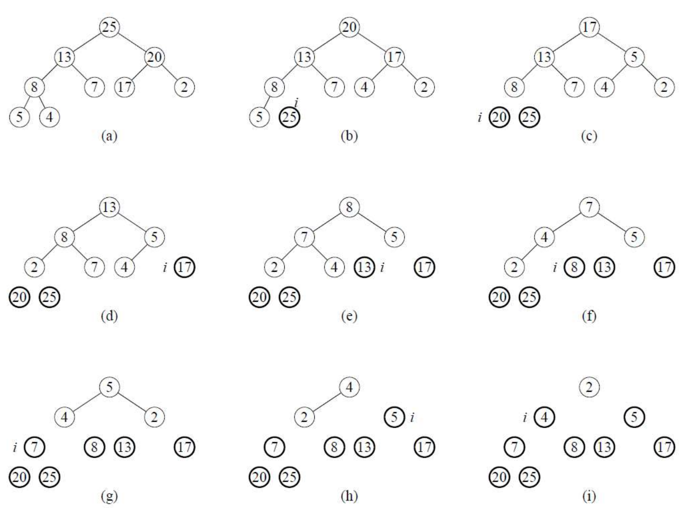
12

# Heapsort

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2
     do exchange A[1] ↔ A[i]
        heap-size[A] ← heap-size[A] − 1
        MAX-HEAPIFY(A, 1)
```

13

# Example: Heapsort



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $A$ | 2 | 4 | 5 | 7 | 8 | 13 | 17 | 20 | 25 |

14

## Analysis

- Cost

  Build-Heap: $O(n)$

  **for** loop: $n - 1$ times

  - exchange elements: $O(1)$
  - Heapify: $O(\lg n)$

  Total time: $O(n \lg n)$
- Though heapsort is a fast algorithm, a well-implemented quicksort usually runs faster

15

## Priority Queues

- Data structure to keep track a dynamic set $S$ of elements, each of which has a key (a priority)
- Operations
  - Insert($S$, $x$) : add element $x$ to $S$
  - Max($S$) : return the max element of $S$
  - ExtractMax($S$) : remove the max element from $S$
- Applications: job scheduling, simulation, sorting, Huffman encoding, other algorithms (shortest path, minimum spanning tree), etc.

16

# Max

```
HEAP-MAXIMUM(A)
    return A[1]
```

- Time: $O(1)$
- How about **Min**?

17

# Extract-Max

```
HEAP-EXTRACT-MAX(A)
    if heap-size[A] < 1
       then error "heap underflow"
    max ← A[1]
    A[1] ← A[heap-size[A]]
    heap-size[A] ← heap-size[A] − 1
    MAX-HEAPIFY(A, 1)
    return max
```

- Time?

18

# Increase-Key

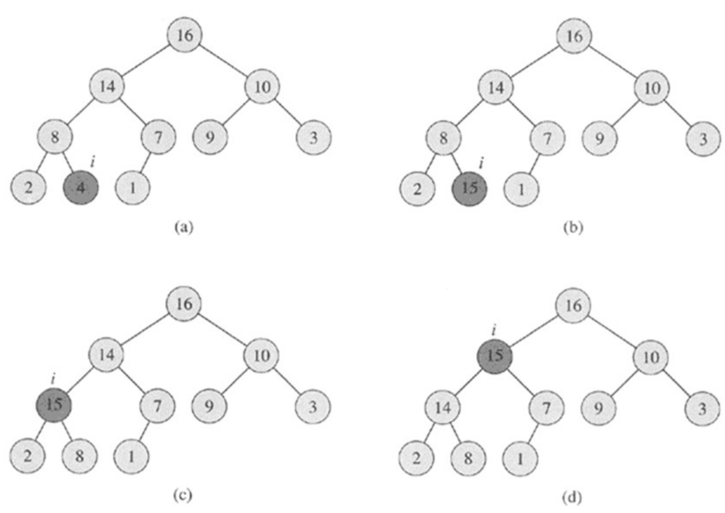
- Auxiliary operation
  - Assumes *key* is larger that *i*’s current priority

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
   then error "new key is smaller than current key"
A[i] ← key
while i > 1 and A[PARENT(i)] < A[i]
    do exchange A[i] ↔ A[PARENT(i)]
       i ← PARENT(i)
```

- Time?

19

# Example: Increase-Key




20

# Insert

MAX-HEAP-INSERT($A$, $key$)

$heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
$A[heap\text{-}size[A]] \leftarrow -\infty$
HEAP-INCREASE-KEY($A$, $heap\text{-}size[A]$, $key$)

- Time?



# Generalizations

- $k$-ary heaps
- Treaps
- Dual heaps
- Min-max heaps
- Bijective heaps
- Interval heaps

Both min-heap and max-heap at once

## Double-Ended Priority Queues

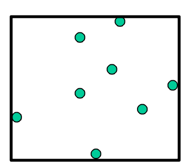
- Primary operations
  - Insert
  - Max
  - Min
  - Extract Max
  - Extract Min

Note: a single-ended priority queue supports just one of the extract operations.

23

## Applications

- Bounding box of a dynamic set of points in $R^d$



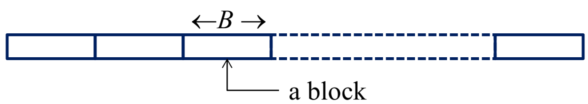
- External version of quicksort
  - Sorting very large data sets that do not fit in main memory

24

# Internal Quicksort

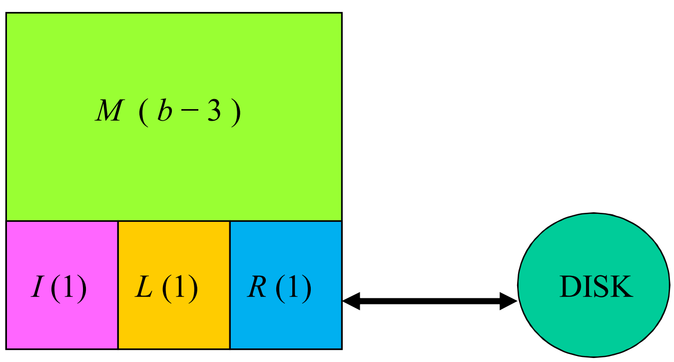
- Select a pivot from the *n* elements.
- Partition the *n* elements into 3 groups: *L*, *M*, *R*.
- The middle group *M* contains only the pivot
- All elements in *L* are ≤ pivot.
- All elements in the right group are > pivot.
- Sort *L* and *R* recursively.
- Answer consists of sorted *L*, followed by *M*, followed by sorted *R*.

- How do we adapt to external sorting?



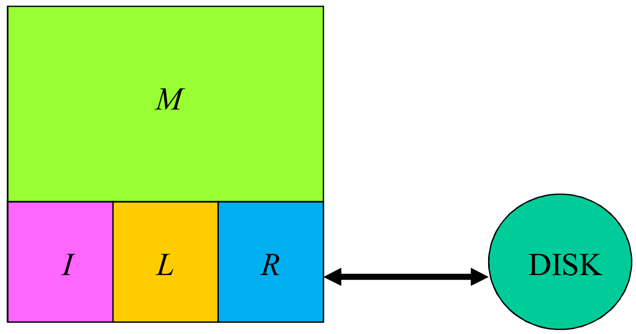

25

# External Quicksort



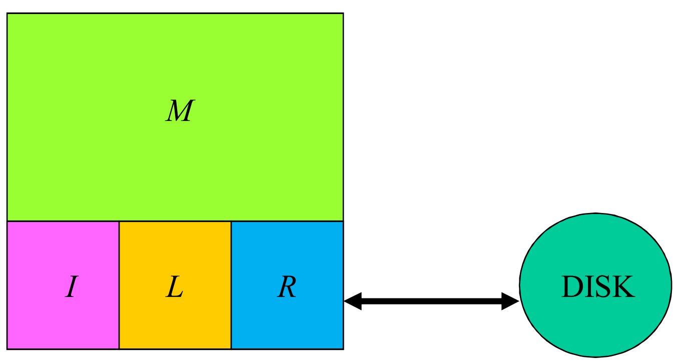

- Assume you have *b* blocks of total buffer space
- 3 I/O buffers: input (*I*), small (*L*), large (*R*)
- Rest is used for middle "pivot" group (*M*)

## External Quicksort…




- Fill middle group from disk into priority queue *M*
- if next record < min(*M*) append to *L*
- if next record > max(*M*) append to *R*
- else ExtractMin (resp. ExtractMax) from *M*, adding to *L* (resp. *R*), and insert new record into *M*

## External Quicksort…



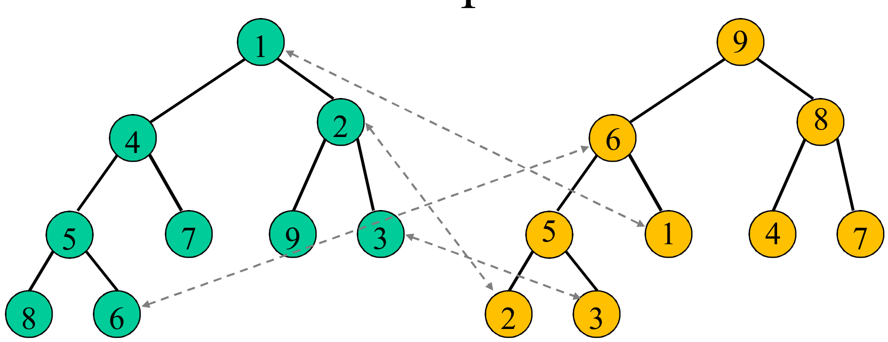

- Fill input buffer when it gets empty.
- Write output buffers (*L* and *R*) when full.
- Write middle group *M* in sorted order when done.

## Dual Heap

- Each element appears in both a min and a max single-ended priority queue.
- Each node in a queue has a pointer to the node in the other queue that stores the same element.
- Single-ended priority queue must support an arbitrary remove.

29

## Example




Min Heap

Max Heap

- Only 4 of 9 bidirectional pointers are shown.
- Operation time is ~ doubled relative to heap.
- Space is ~ doubled relative to heap

30

# Bijective Queue

- Use min and max single-ended priority queues, each with $\lfloor n/2 \rfloor$ of nodes, + 1 extra node buffer.
- When $n$ is odd, one element stored in the buffer.
- Remaining elements are in the single-ended priority queues.
- Establish a bijection $\boldsymbol{f}$ between the nodes of min queue and nodes of max queue.

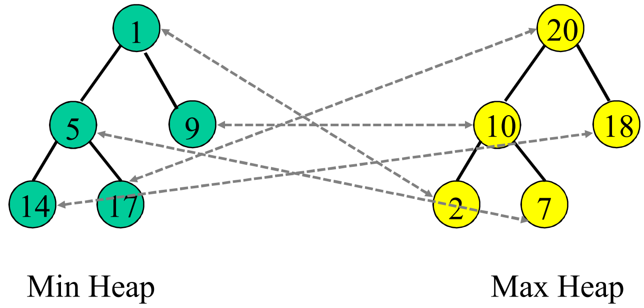
  $x$ and $f(x)$ are said to be "twins" **and** $x \le f(x)$
- Single-ended priority queues must support arbitrary remove method.

31

# Example

$S = \{1,2,5,7,9,10,12,14,17,18,20\}$

$= \{1,5,9,14,17\} \cup \{2,7,10,18,20\} \cup \{12\}$



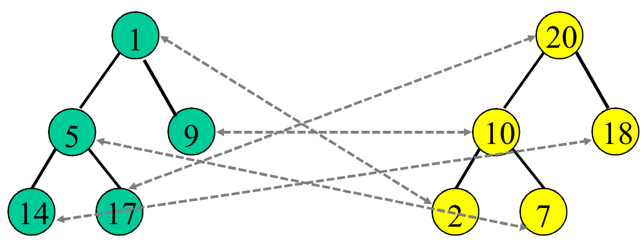

Buffer = 12

32

# Insert($x$)

- Buffer empty => place $x$ in buffer.
- Else, insert min{$x$, buffer} into min queue, max{$x$, buffer} into max queue, make them twins

Example: Insert 12, then 3 into the bijective heap below




Min Heap

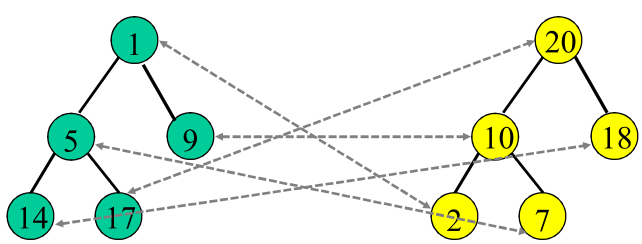
Max Heap

Buffer =

33

# Extract Min

- Buffer is min ⇒ empty buffer.
- Else, remove min from min queue and twin from max queue; reinsert twin




Min Heap

Max Heap

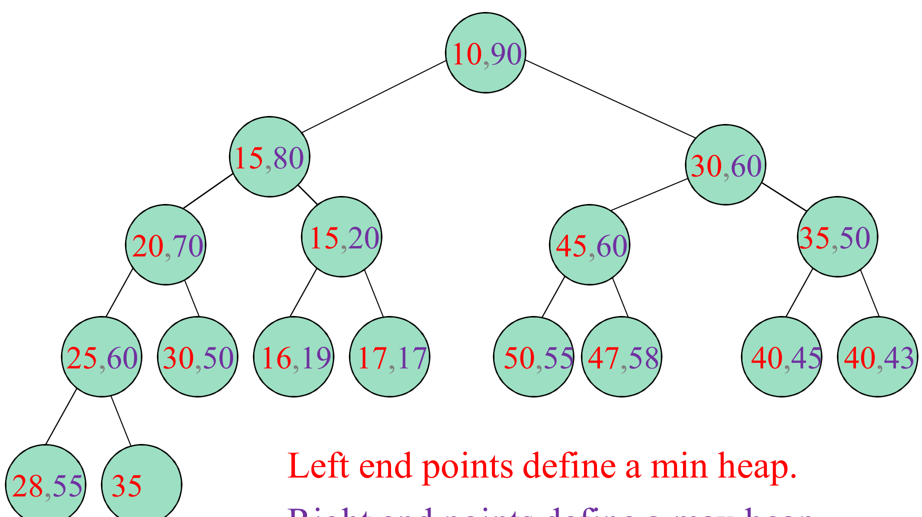
Buffer = 12

34

# Interval Heaps

- Complete binary tree
  - Stored in an array as an ordinary heap
  - Each array location has room for two elements
- Each non-last node has two elements
- Last node has one or two elements
- The interval of a node with values $a \le b$ is $[a,b]$
- The interval of a node with one value $a$ can be viewed as $[a, a]$
- Nesting property: if $q$ has interval $[c,d]$ and $q$'s parent has interval $[a,b]$. Then $a \le c \le d \le b$
- Left endpoints of intervals define a min-heap and right endpoints define a max-heap

# Example Interval Heap




Left end points define a min heap.

Right end points define a max heap.

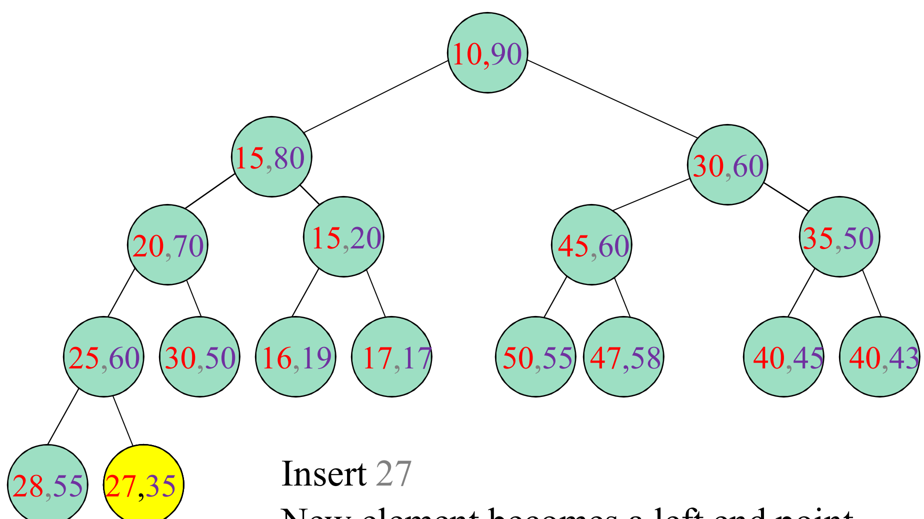
Min and max stored at the root

## Inserting an Element $x$

1. Create additional node if $n$ is even.
2. Let $q$ be the last node and $p$ its parent with interval $[a,b]$.
3. Three cases:
   1. $x \in [a,b] \Rightarrow$ insert $x$ in last node
   2. $x < a \Rightarrow$ insert $x$ into min-heap
   3. $x > b \Rightarrow$ insert $x$ into max-heap

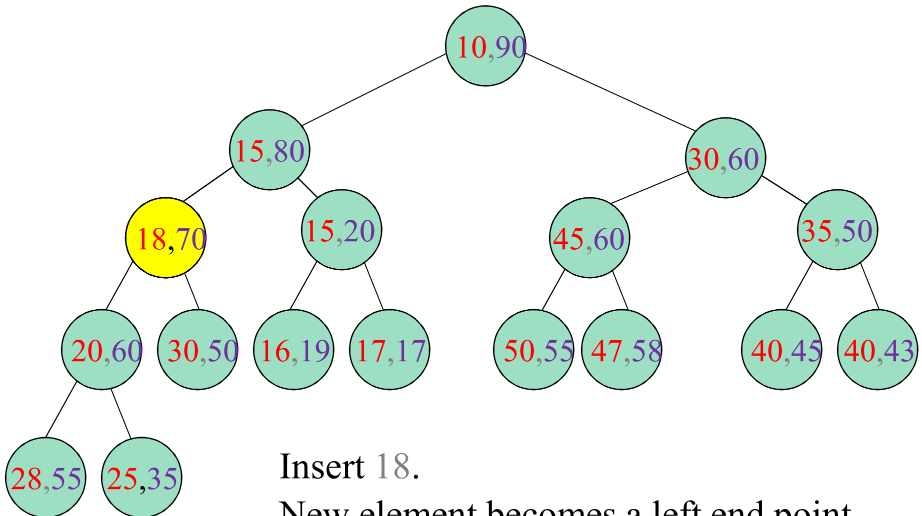
37

## Insert An Element



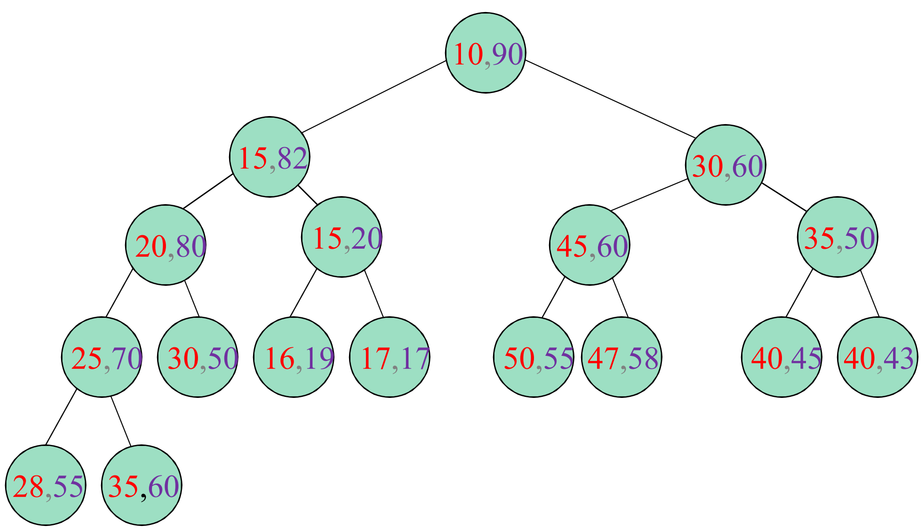

Insert 27
New element becomes a left end point

# Another Insert

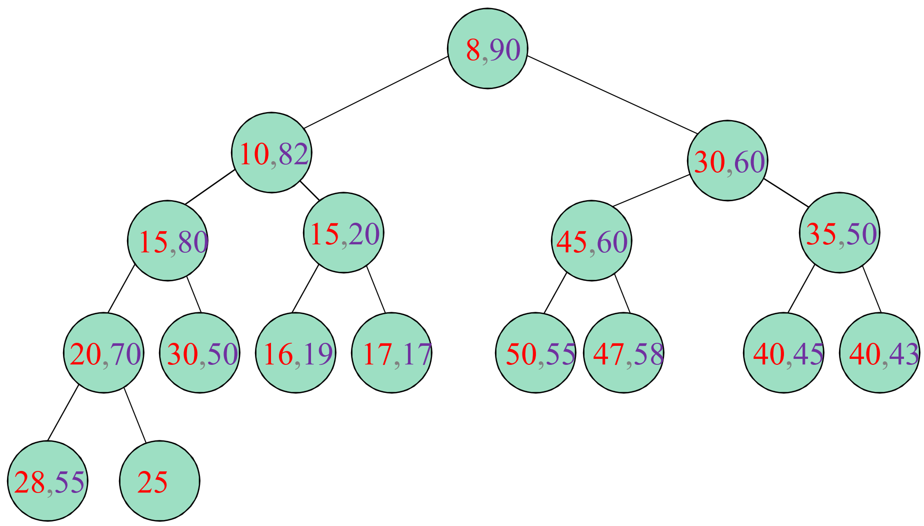



Insert 18.
New element becomes a left end point.
Insert new element into min heap.

# After 82 Inserted

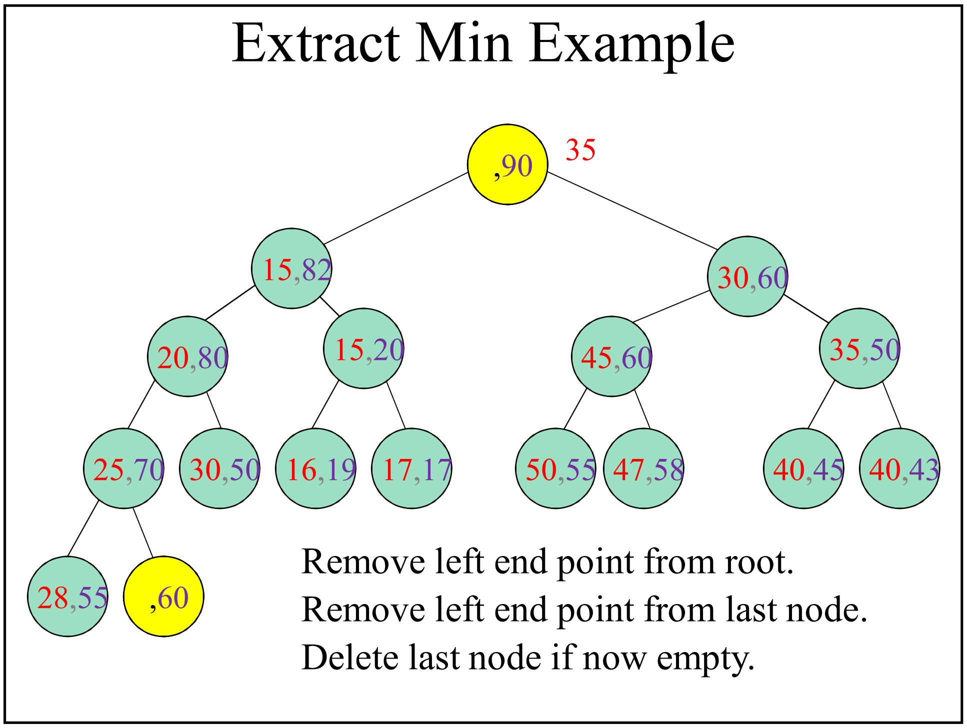


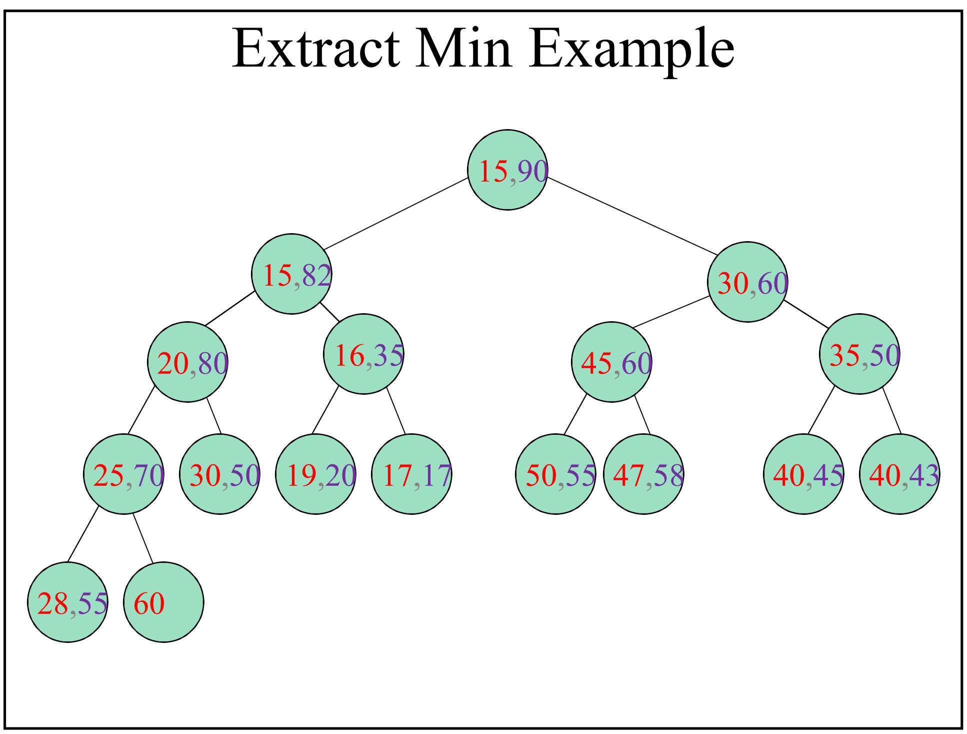
## After 8 Is Inserted



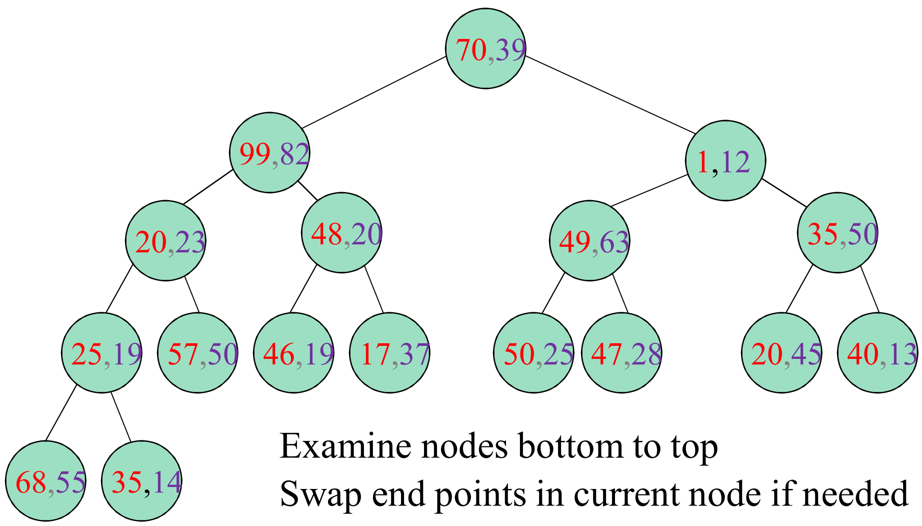

## Extract Min

- $n = 0 \Rightarrow$ fail.
- $n = 1 \Rightarrow$ heap becomes empty.
- $n = 2 \Rightarrow$ only one node, take out left end point.
- $n > 2 \Rightarrow$
  - Remove and return left endpoint of root
  - Remove the left endpoint $p$ from last node
  - If last node becomes empty remove it
  - Bubble $p$ down into min heap, beginning at root (it may become necessary to swap $p$ with right endpoint $r$ of node if $r < p$)


Extract Min Example
,90
35
15,82
30,60
20,80
15,20
45,60
35,50
25,70
30,50
16,19
17,17
50,55
47,58
40,45
40,43
28,55
,60
Remove left end point from root.
Remove left end point from last node.
Delete last node if now empty.

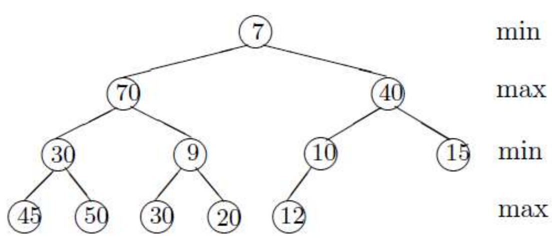

Extract Min Example
15,90
15,82
30,60
20,80
16,35
45,60
35,50
25,70
30,50
19,20
17,17
50,55
47,58
40,45
40,43
28,55
60

# Build-Interval-Heap




Examine nodes bottom to top
Swap end points in current node if needed
Bubble left end point down into min heap
Bubble right end point down into max heap

# Min-Max Heaps




- Alternates between min and max levels

  if depth($x$) is even then

  $x$ is the smallest of all elements in its subtree

  else

  $x$ is the largest of all elements in its subtree

53