EE 360C - ALGORITHMS

Lecture 7

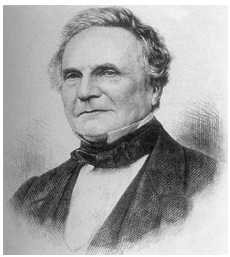
Priority Queues

Vallath Nandakumar
University of Texas at Austin

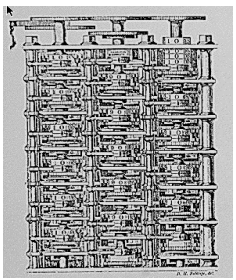
"There is a 15 minute wait for people we like, and a 45 minute wait for people like you."

Computational Tractability

As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time? - *Charles Babbage*



Charles Babbage (1864)



Analytic Engine (schematic)

PRIORITY QUEUE

- **Example - Process Scheduling Queue**
  - **Processes do not arrive in order of priority**
  - **Want to select active process with highest priority to execute**
  - **Need to**
    - **Add elements to set**
    - **Select element from set with highest priority**
    - **Delete element from set**

PRIORITY QUEUE

- Store set *S* of elements
  - Each element *v* has priority value key(*v*)
- Smaller key values denote higher priorities
- Operations supported
  - Find element with smallest key
  - Remove element with smallest key
  - Insert new element
  - Delete element
- Key update and element deletion require knowledge of position of element in priority queue

PRIORITY QUEUE

☞ **Priority queue can be used to sort *n* numbers**

```
1  for i = 1 to n
2     do insert unsorted[i] into priority queue
3  for i = 1 to n
4     do sorted[i] ← Extract smallest from priority queue
```

Priority Queue

- Recall priority queue
  - elements enqueued based on priority
  - dequeue removes the highest priority item
- Options?
  - List? Binary Search Tree?

	Linked List enqueue	BST enqueue
A.	O(N)	O(1)
B.	O(N)	O(logN)
C.	O(N)	O(N)
D.	O(logN)	O(logN)
E.	O(1)	O(logN)

Another Option

- A *heap*
  - not to be confused with the runtime heap, portion of memory for dynamically allocated variables
- A complete binary tree
  - all levels have maximum number of nodes except deepest where nodes are filled in from left to right
- Maintains the *heap order property*
  - in a min heap the value in the root of any subtree is less than or equal to all other values in the subtree

HEAP

- Heap
  - Combines benefits of lists and sorted arrays
  - Balanced binary tree
  - Maximum size N known in advance
  - Heap Order
    - Key of parent ≤ key of child
  - Store nodes of heap in array
    - Node at index i has children at indices $2i$ and $2i+1$
    - Node at index i has parent at index $\lfloor i/2 \rfloor$
    - Index 1 is root
    - Node at index i is leaf, if $2i > N$

Clicker Question 2

- In a max heap with no duplicates where is the largest value?

A. the root of the tree

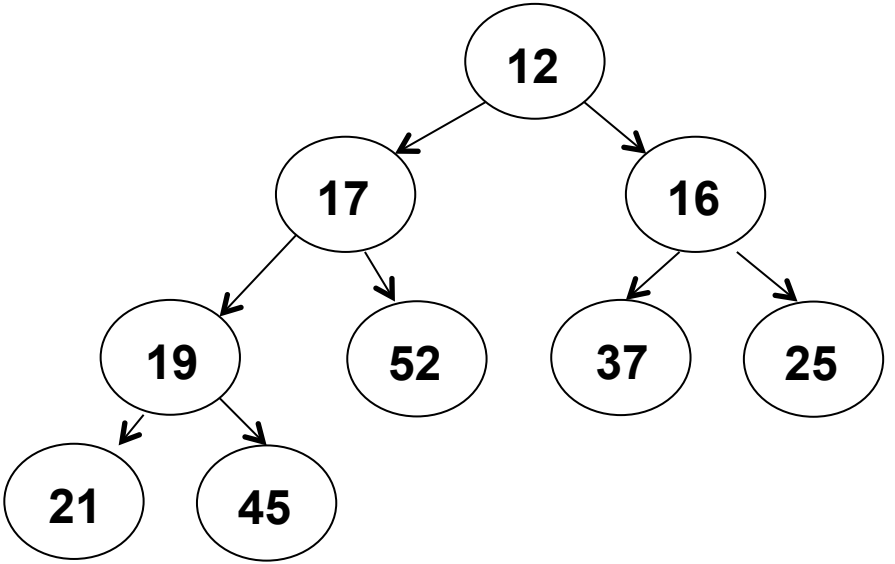
B. in the left-most node

C. in the right-most node

D. a node in the lowest level

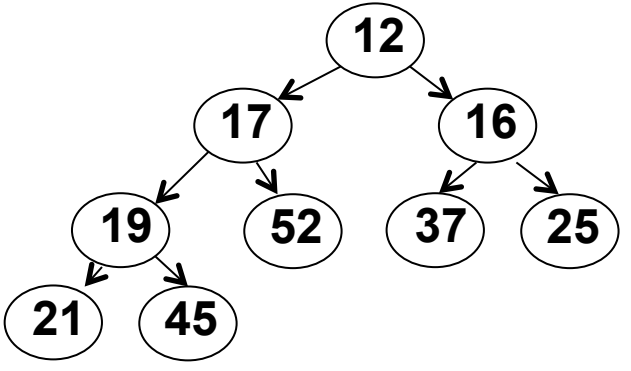
E. None of these

Example Min Heap



Internal Storage

- Interestingly heaps are usually implemented with an array instead of nodes



for element at position i:

parent index: *i / 2*

left child index: *i * 2*

right child index: *i * 2 + 1*

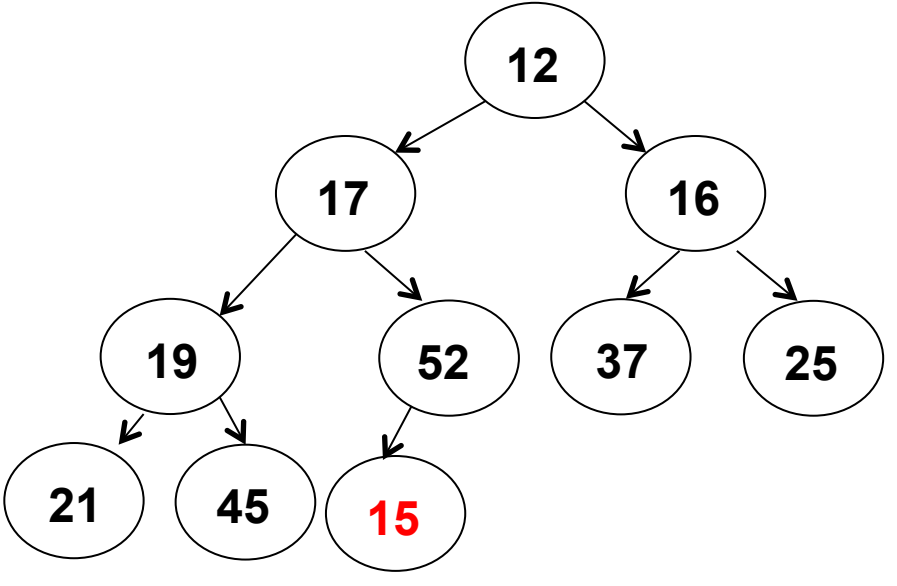
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	12	**17**	**16**	**19**	**52**	**37**	**25**	**21**	**45**						

Enqueue Operation

- Add new element to next open spot in array
- Swap with parent if new value is less than parent
- Continue back up the tree as long as the new value is less than new parent node

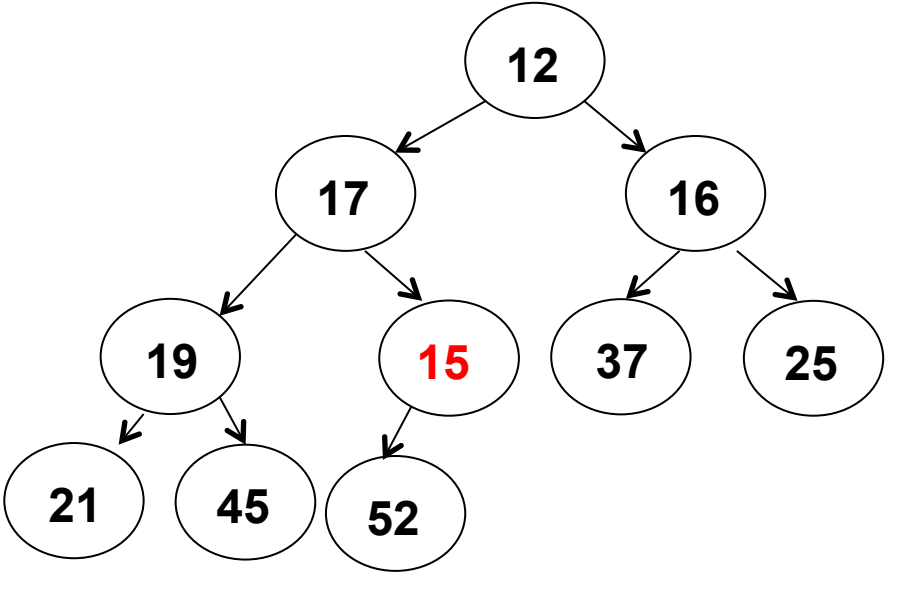
Enqueue Example

- Add 15 to heap (initially next left most node)



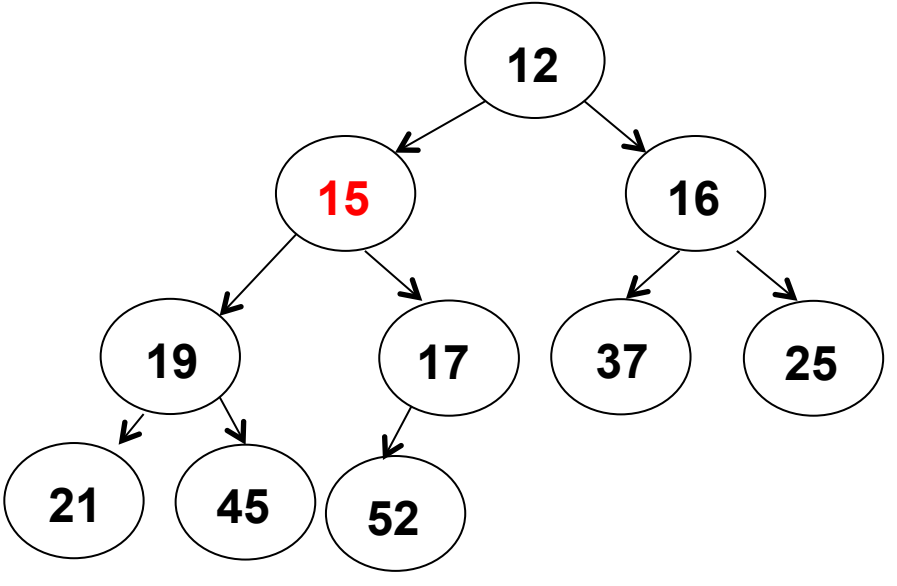
Enqueue Example

- Swap 15 and 52



Enqueue Example

- Swap 15 and 17, then stop

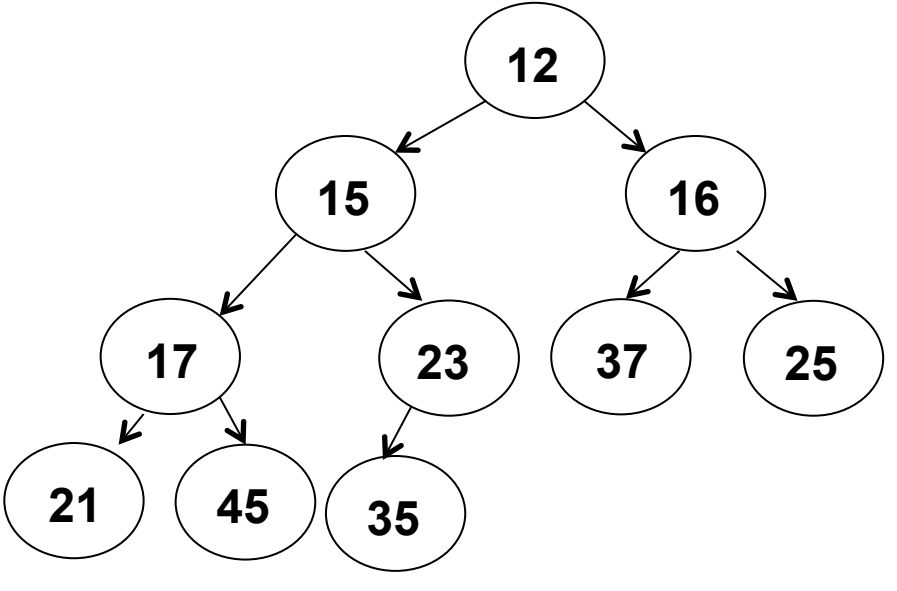


Dequeue

- min value / front of queue is in root of tree
- swap value from last node to root and move down swapping with smaller child unless values is smaller than both children

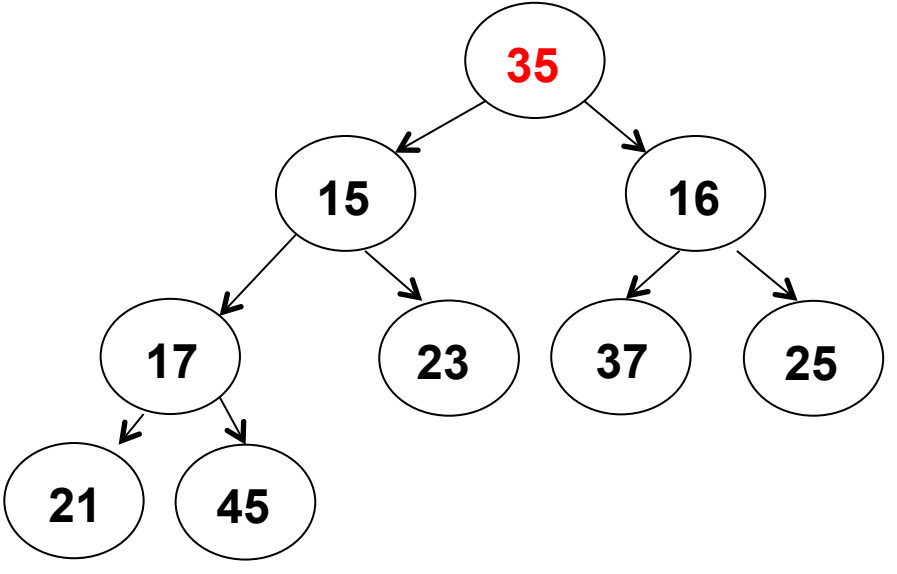
Dequeue Example

- Swap 35 into root (save 12 to return)



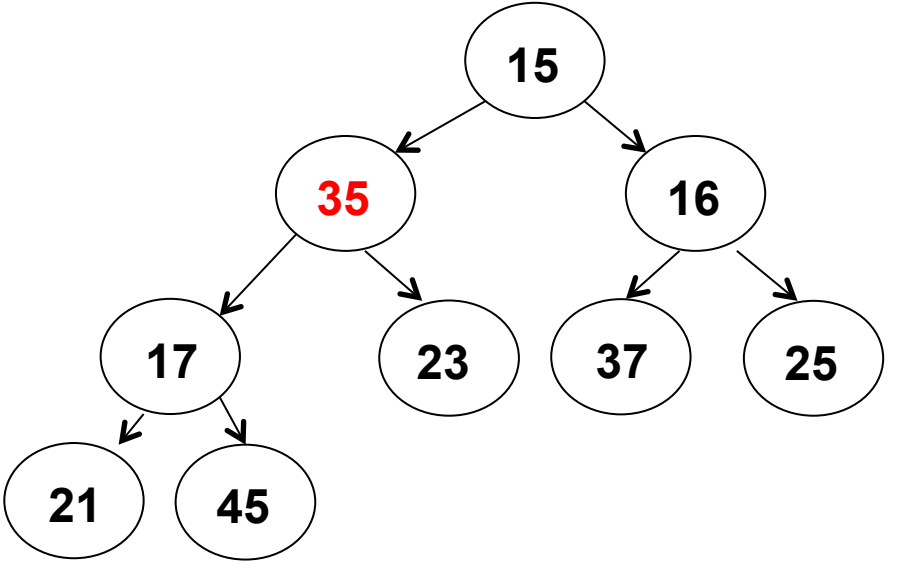
Dequeue Example

- Swap 35 into root (save 12 to return)



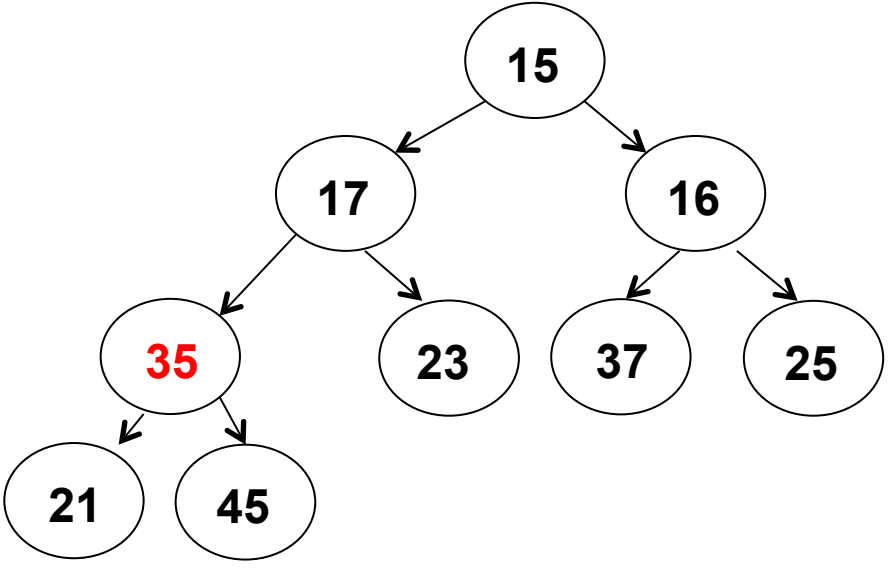
Dequeue Example

- Swap 35 with smaller child (15)



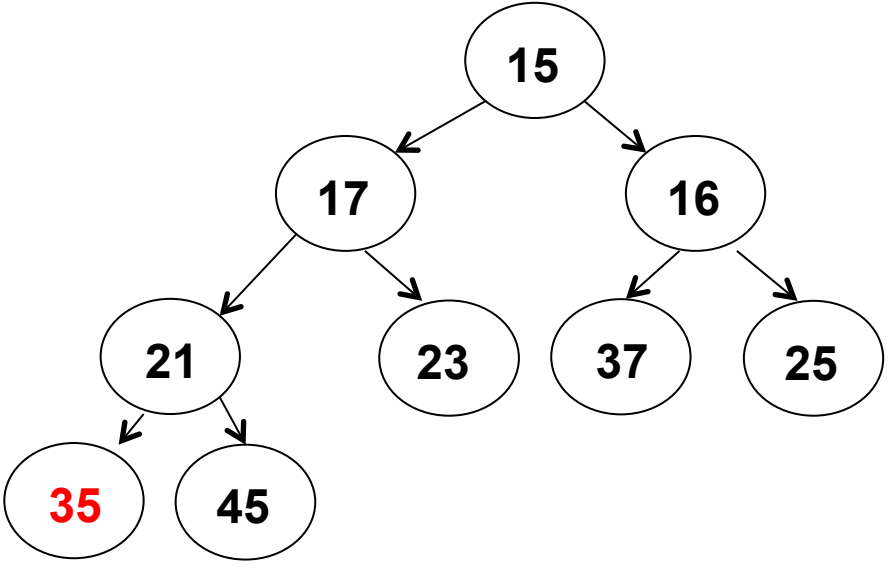
Dequeue Example

- Swap 35 with smaller child (17)



Dequeue Example

- Swap 35 with smaller child (21)



PriorityQueue Class

```
public class PriorityQueue<E extends Comparable<E>> {

    private int size;
    private E[] con;

    public PriorityQueue() {
        con = getArray(2);
    }

    private E[] getArray(int size) {
        return (E[]) (new Comparable[size]);
    }
```

PriorityQueue enqueue

```
public void enqueue(E val) {
    if ( size >= con.length - 1 )
        enlargeArray( con.length * 2 + 1 );

    size++;
    int indexToPlace = size;
    while ( indexToPlace > 1
            && val.compareTo( con[indexToPlace / 2] ) < 0 ) {

        con[indexToPlace] = con[indexToPlace / 2]; // swap
        indexToPlace /= 2; // change indexToPlace to parent
    }
    con[indexToPlace] = val;
}

private void enlargeArray(int newSize) {
    E[] temp = getArray(newSize);
    System.arraycopy(con, 1, temp, 1, size);
    con = temp;
}
```

Dequeue Code

```
public E dequeue( ) {
    E top = con[1];
    int hole = 1;
    boolean done = false;
    while ( hole * 2 < size && ! done ) {
        int child = hole * 2;
        // see which child is smaller
        if ( con[child].compareTo( con[child + 1] ) > 0 )
            child++;    // child now points to smaller

        // is replacement value bigger than child?
        if (con[size].compareTo( con[child] ) > 0 ) {
            con[hole] = con[child];
            hole = child;
        }
        else
            done = true;
    }
    con[hole] = con[size];
    size--;
    return top;
}
```

DELETING INTERNAL ELEMENT

- If H has n+1 elements
  - Delete $H[i]$ by moving element at $H[n+1]$ to $H[i]$
- If $key(H[i])$ too small
  - Fix heap order using Heapify-Up(H,i)
- If $key(H[i])$ too large
  - Fix heap order using Heapify-Down(H,i)

SORTING WITH PRIORITY QUEUE

- **Sort**
  - **Instance: Nonempty list $x_1, x_2, \ldots, x_n$ of integers**
  - **Solution: Permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \le y_{i+1}$ for all $1 \le i < n$**
- **Algorithm**
  - **Insert each number in priority queue *H***
  - **Repeatedly output smallest number in *H* and delete**
- **Each insertion and deletion take $O(\log n)$**
- **Need to do for each of the *n* elements**
  - **Total running time of $O(n \log n)$**