# HEAP SORT

CS136

## RETURN TO BINARY TREES

Monday: Heap sort

Wednesday: BST implementation

Friday: BST balance

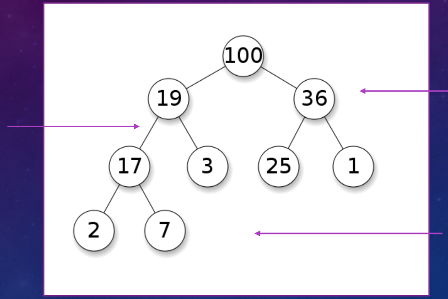
Monday: BST remove

## HEAP SORT

- Deterministic, but not stable
- O(n lg n) run time
- Only O(1) additional cost!
  - Works **in place**
  - Stackless
- Ideal for fixed-memory environments, like GPU kernel programming and embedded processors
  - Faster than insertion sort, and merge and quicksort are impossible in this environment
- Elegant implementation

## HEAP REVIEW

# HEAP = COMPLETE BINARY TREE WITH HEAP PROPERTY

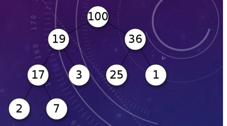


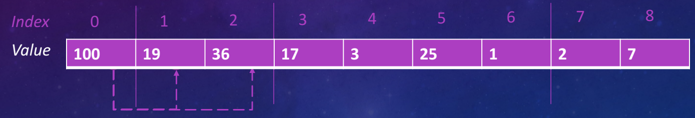
At most two children

Parents have greater values than children

All levels full except the last &
Last level filled from the left

# EFFICIENT IMPLICIT HEAP REPRESENTATION





| *Index* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| *Value* | 100 | 19 | 36 | 17 | 3 | 25 | 1 | 2 | 7 |

# HEAP SORT IMPLEMENTATION

# HEAP SORT IDEA

- Build a **max heap** using the **implicit complete** binary tree notation
  - Children of node at i are at 2i+1 and 2i+2
  - Parent is greater than its children (and has index floor((i-1)/2))
  - Repeated “sift down” operations
- Repeatedly extract the **max**
  - On step j, swap element 0 with element N – j – 1
  - Consider the end fixed and **sift down** the new root

## BE CAREFUL

- The tree is a *concept*
  - No explicit tree
  - No pointers
- The heap structure is *not* itself sorted
- We build a *max* heap to sort from least to greatest because we’re going to read the values out in backwards order

```
public class HeapSort {

/* Helper function: sift element[parent] down the tree */
void siftDown(Element[] element, int parent, final int end) {

    final Element value = element[parent];

    int maxChild = parent * 2 + 1;
    while (maxChild <= end) {

        // See if the other child is larger
        if (maxChild < end) {
            final int otherChild = maxChild + 1;
            maxChild = (element[otherChild] > element[maxChild]) ?
                          otherChild : maxChild;
        }

        // Stop when the parent is larger than the max child
        if (value >= element[maxChild]) break;

        element[parent] = element[maxChild];

        parent = maxChild;
        maxChild = parent * 2 + 1;
    }

    element[parent] = value;
}
```

```
public void heapSort(Element[] element) {
    // Form a max heap
    final int N = element.length;

    for (int i = N / 2; i >= 0; --i)
        siftDown(element, i, N - 1);

    // Read out the values
    for (int i = N - 1; i >= 1; --i) {
        // Swap out of the heap region
        final Element temp = element[0];
        element[0] = element[i];
        element[i] = temp;

        // Restore the heap property
        siftDown(element, 0, i - 1);
    }
}

}
```