# CS 590: Algorithms

## Sorting and Order Statistics I:

## Heapsort / Quicksort

# Outline

# 4.1. Heapsort

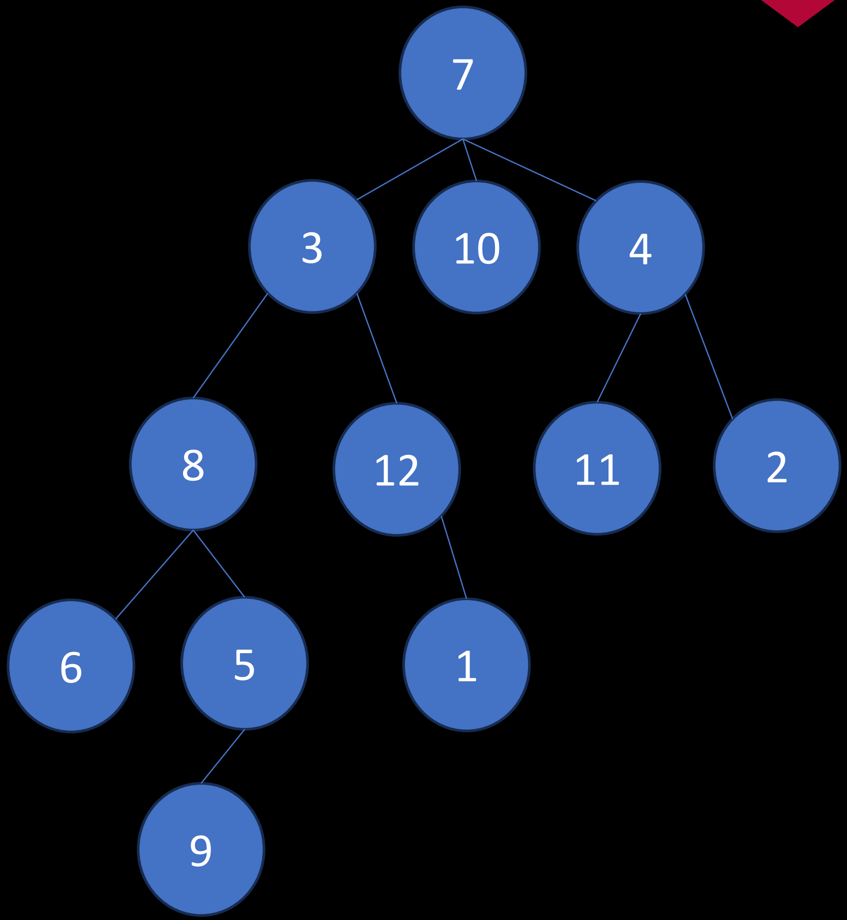
# 4.1.1. Tree and Binary Tree

**Tree:**

- The tree is a graphical way of representing a data structure.

**Rooted tree**:

- A rooted tree has one of the vertices distinguished from the others.
- The distinguished vertex is called the **root** of the tree and is located at the top of the tree.
- A vertex of a rooted tree is also called a **node,** represented by a circle.
- A pair of nodes are connected by an edge, represented by a segment.

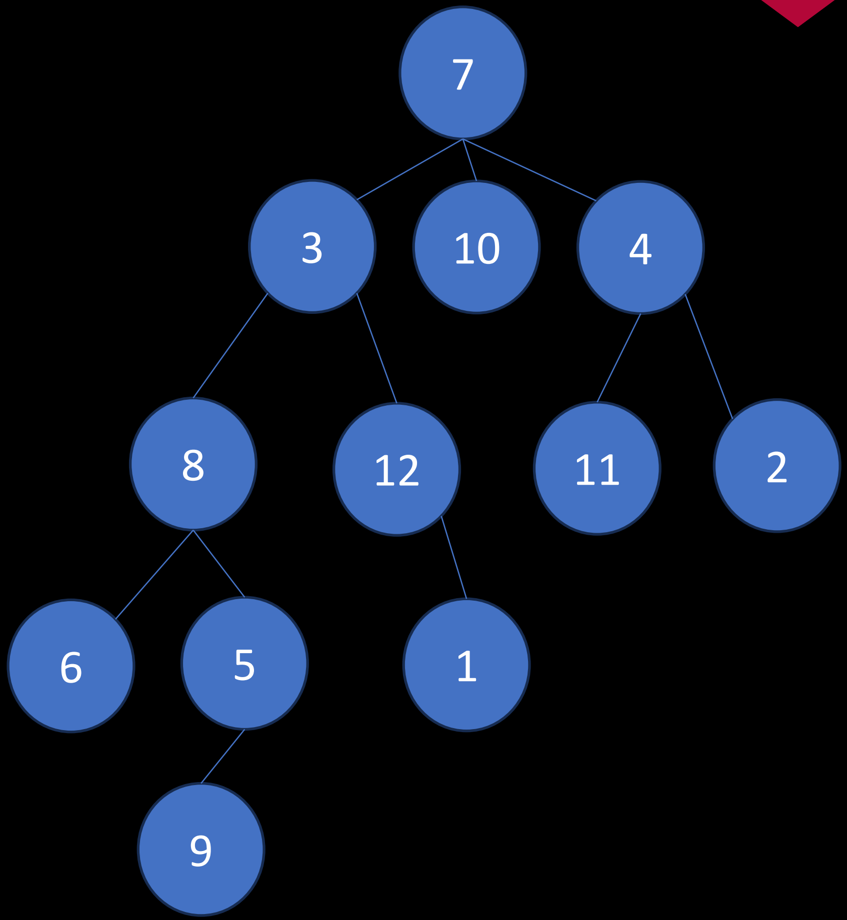


# 4.1.1. Tree and Binary Tree

**Properties:**

Consider a node $x$ in a rooted tree $T$ with root $r$.

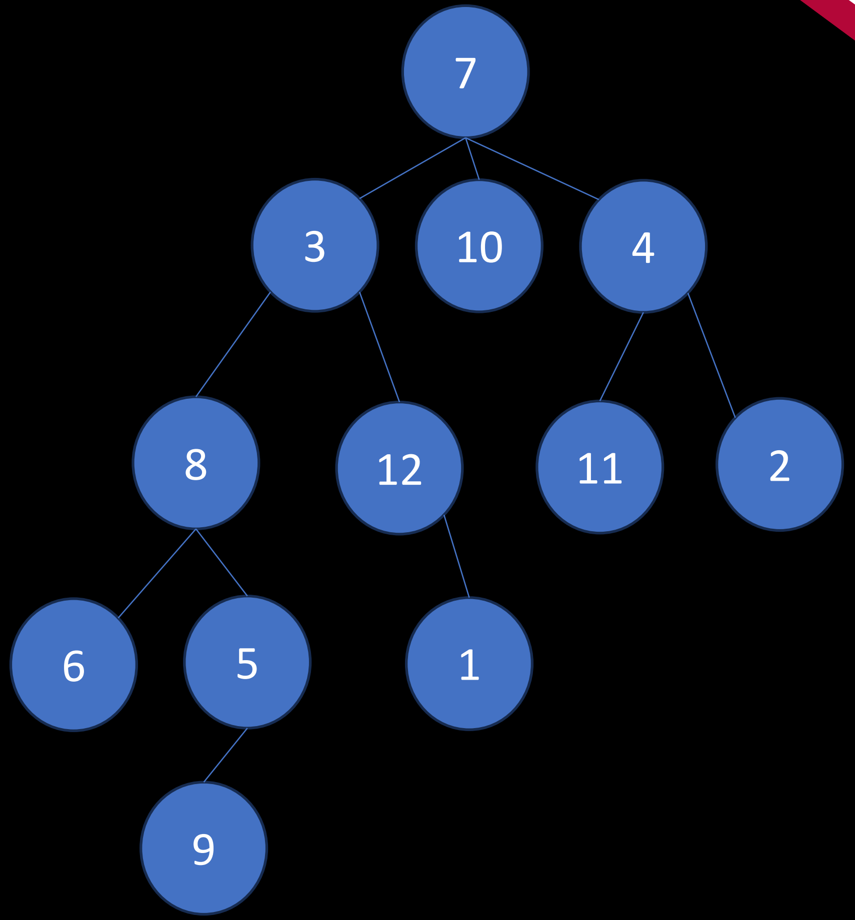
- **Parent and child:** A node $y$ at one lower depth connected to $x$. A node $z$ at one depth higher and connected to $x$ is a **child** of $x$.
- **Ancestor**: Any node $y$ on the unique simple path from $r$ to $x$, $y$ is an **ancestor** of $x$.
- **Descendant**: If $y$ is an ancestor of $x$, then $x$ is a descendant of $y$.
- The **subtree** rooted at x: A tree induced by a descendant of x.
- If two nodes have the same parent, they are **siblings**.



# 4.1.1. Tree and Binary Tree

<u>**Characteristics:**</u>

- **Degree** of x: the number of children of a node x.
- **Depth** of x: the length of the simple path from the root.
- **Level**: consists of all nodes at the same depth.
- **Height**: the number of edges on the **longest simple downward path** from a leaf (a node without a child) to the root. It is also equal to the largest depth of any node in the tree.



# 4.1.1. Tree and Binary Tree

A **binary tree** is a structure defined on a finite set of nodes that either

- contains no nodes or
- composed of three disjoint sets of nodes – a root node, a left subtree, and a right subtree.

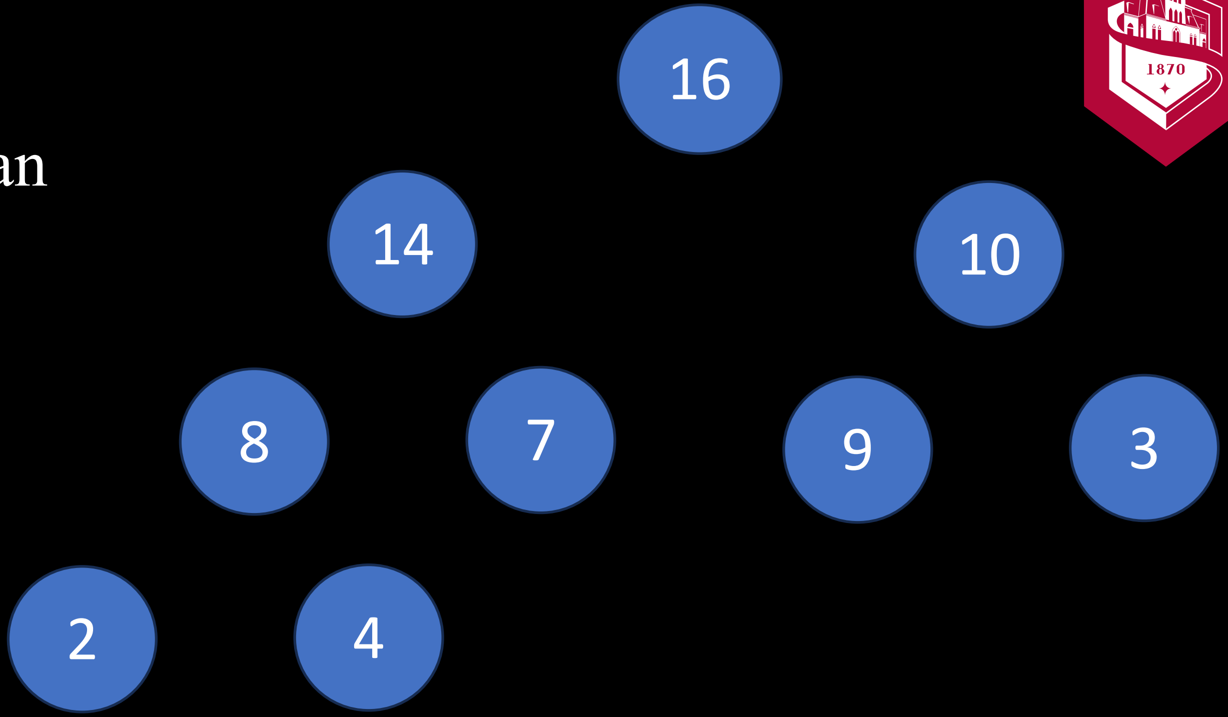
# 4.1.1. Tree and Binary Tree

- A **binary tree** is not simply an ordered tree in which each node has a degree at most of 2.
- **Full binary tree**: each node is either a leaf or has a degree of exactly 2. (there are no degree-1 nodes).
- **Perfect binary tree**: a full binary tree of height $h$ with exactly $2^h - 1$ nodes.
- **Complete binary tree**: a perfect binary tree through $h - 1$ with some extra leaf nodes at depth $h$, *all toward the left*.

# 4.1.2. Heap

A **heap** (or binary heap) data structure is an array object $A$ that can be considered as a complete binary tree with the following properties.

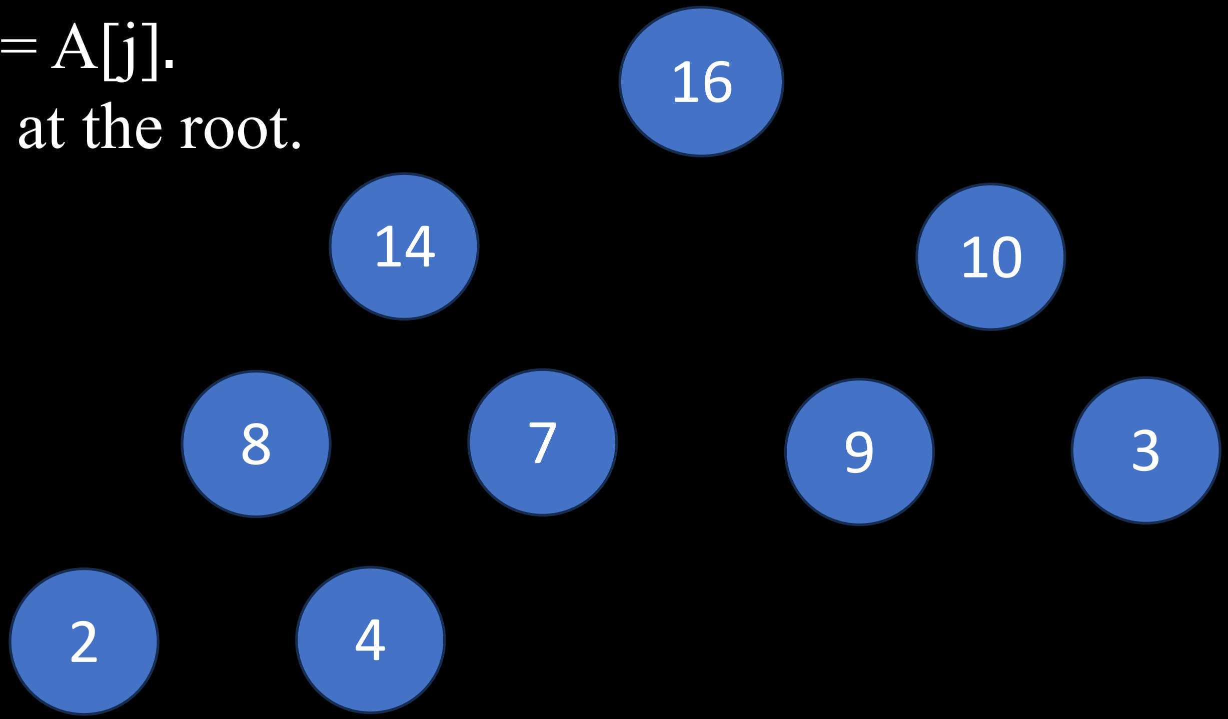
- The root of the tree is $A[1]$.
- Every subtree is a heap
  - $0 \le A.heapsize \le A.length$
- The indices of parent, left child, and right child for A[j] are $A\left[\left\lfloor \frac{j}{2} \right\rfloor\right]$, $A[2j]$, and $A[2j+1]$, respectively.



# 4.1.2. Heap

**Two kinds of binary heaps:**

- **Max-heap**:
  - The value of a node is the most the value of its parents, A[Parent(j)] >= A[j].
  - The largest element in a max-heap is stored at the root.
- **Min-heap**:
  - Opposite of Max-heap: A[Parent(j)] <= A[j].
  - The smallest element in a min-heap is at the root.



# 4.1.2. Heap

The height of the heap is the height of its root.

- A heap with n elements based on a complete binary tree runs $\Theta(\lg n)$ time.

**Proof:**

Given an n-element heap of height k and is an almost-complete binary tree.

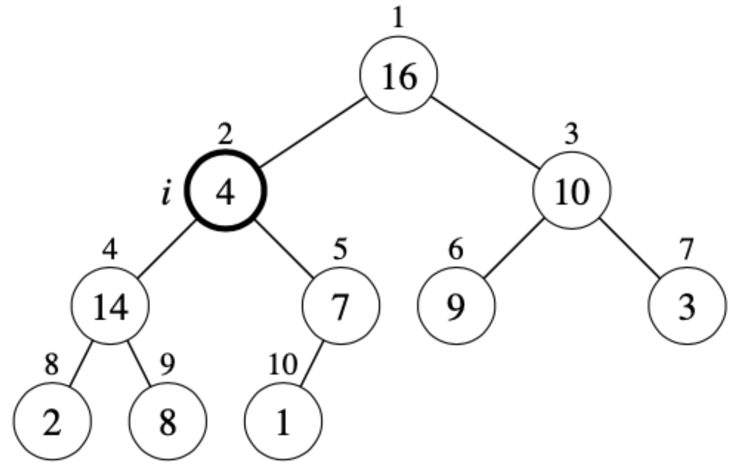
- it has at most $2^{k+1} - 1$ elements when all levels are complete.
- it has at least $2^k$ elements when the most-left node at the depth of k has a child.

# 4.1.2. Heap

**General Algorithmic Procedure of Heapsort**:

1. The MAX-HEAPIFY procedure is the key to maintaining the max-heap property.
2. The BUILD-MAX-HEAP procedure produces a max-heap from an unordered input array.
3. The HEAPSORT sorts an array in place.
4. The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, and HEAP-MAXIMUM allow the heap data structure to implement a priority queue.

# 4.1.2. Heap



(a)

Demonstration on board.

# 4.1.2. Heap

The MAX-HEAPIFY( ) assumption:

- Consider a subtree with a root of j at any depth.
- Suppose the subtrees rooted at $LEFT(j)$ and $RIGHT(j)$ are max-heaps.
- If $A[j]$ is smaller than its children, violates the max-heap property.

The MAX-HEAPIFY( ) operations as follow:

- It lets have value at $A[j]$ "float down" in the max-heap – so the subtree rooted at index $j$ obeys the max-heap property.
- Then, A[j] needs to be placed at its highest children node (either LEFT(j) or RIGHT(j)).
- The children node with the highest value replaces A[j].
- This comparison and swap between A[j] and children until the array is heapified.

# 4.1.2. Heap

```
MAX-HEAPIFY(A,j,n)
l = LEFT(j), r = RIGHT(j)
if (l <= n and A[l]>A[j]) then
    largest = l
else
    largest = j
if (r <= n and A[r] > A[largest])
then
    largest = r
if (largest != j) then
    swap A[j] and A[largest]
    MAX-HEAPIFY(A, largest, n)
```

# 4.1.2. Heap

- Heap is an almost complete binary tree.
- Compares 3 items and swaps 2 at the most.
  - Fixing up the relationships among the elements (A[j], A[LEFT(j), and A[RIGHT(j)]) at a given node j takes $\Theta(1)$ time.

Worst case:

- when the node A[j] becomes a leaf of the tree.
- The children's subtrees each have a size at most $2n/3$. (only half full case)
- The running time of the recursion equation becomes $T(n) \le T\left(\frac{2n}{3}\right) + \Theta(1)$.
- The solution of MAX-HEAPIFY( ) running time is $T(n) = O(\lg n)$.

# 4.1.2. Heap

## Building a heap

- Use MAX-HEAPIFY in a *bottom-up* manner to convert an array $A[1 \dots n]$ into a max-heap.
- $A\left[\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) \dots n\right]$ are all leaves of the tree so each is a 1-element heap to begin with.

```
BUILD-MAX-HEAP(A, n)
for (floor(n/2) >= j >= 1)
      MAX-HEAPIFY(A, j, n)
```

# 4.1.2. Heap

**Loop Invariant Correctness:**

- Initialization – since j $= \left\lfloor \frac{n}{2} \right\rfloor$ before the first iteration of the for loop, the invariant is initially true.
  - Why?
- Maintenance – Decrementing j reestablishes the loop invariant at each iteration.
  - Is the subsolution always a max-heap?
- Termination – When $i = 0$, the loop terminates.
  - What happens to the solution?

# 4.1.2. Heap

## BUILD-MAX-HEAP( ) Analysis:

1. **Simple Bound:** the running time is $O(n \lg n)$.
2. **tighter analysis:**
   - The running time for MAX-HEAPIFY( ) becomes linear in the node j's height for its run. Most nodes have small heights.
   - We have $\leq \left\lceil \frac{n}{2^{k+1}} \right\rceil$ nodes of height $k$ and height of heap is $\lfloor \lg n \rfloor$.

# 4.1.2. Heap

## BUILD-MAX-HEAP( ) Analysis:

**How?**

- Start with finding the number of leaves in tree T.
- The tree leaves (nodes at height 0) are at depths $K$ and $K - 1$.
- They consist of all notes at depth $K$, and the nodes at depth $K - 1$ that are not parents of depth-$K$ nodes.

# 4.1.2. Heap

## BUILD-MAX-HEAP( ) Analysis:

- When n is odd, the even number of leaves $x$ in the tree T at the depth $K$.
  - If n is even, then x is odd.
- If x is even,
  - x/2 nodes of nodes at $K-1$ are parents of $x$.
  - $2^{K-1} - x/2$ nodes at $K-1$ depth are leaves since there are a total of $2^{K-1}$ nodes at K-1 depth.
- The total number of leaves in T is $\left\lceil \frac{n}{2} \right\rceil$.
- If x is odd, the total number of leaves is also $\left\lceil \frac{n}{2} \right\rceil$.
  - Similar arguments can be made.

# 4.1.2. Heap

## BUILD-MAX-HEAP( ) Analysis:

- Let $n_h$ be the number of nodes at height $h$ in the $n$-node tree $T$.
- Form a new tree $T'$ formed after removing the leaves of tree $T$.
- Then it has $n' = n - n_h$ nodes.
- Since we know that $n_h = \left\lceil \frac{n}{2} \right\rceil$, $n' = \left\lfloor \frac{n}{2} \right\rfloor$.

# 4.1.2. Heap

## BUILD-MAX-HEAP( ) Analysis:

- Let $n'_{h-1}$ be the number of nodes at height $k-1$ in $T'$.
- Then $n_h = n'_{h-1}$.
- We can bound that $n_h = n'_{h-1} \leq \left\lceil \frac{n'}{2^h} \right\rceil = \left\lceil \frac{\lfloor n/2 \rfloor}{2^k} \right\rceil \leq \left\lceil \frac{n/2}{2^k} \right\rceil = \left\lceil \frac{n}{2^{k+1}} \right\rceil$.
- The time required by MAX-HEAPIFY when called on a node of height $h$ is $O(h)$, so the total cost of BUILD-MAX-HEAP is

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

- Thus, the running time of BUILD-MAX-HEAP is $O(n)$.
- The same argument works for min-heap and MIN-HEAPIFY.

# 4.1.3. Heapsort

- Finding the maximum value in an array A[1,…, n] is easy:
- A[…] becomes a max-heap using **BUILD-MAX-HEAP( )** function.
- Then, A[1] is always the maximum element.
- However, finding the smallest element A[j] is not easy.
- How can A[…] be sorted?
- Consider a max-heap array A=[7, 4, 3, 1, 2].

# 4.1.3. Heapsort

For a given input array, the Heapsort works as follows:

1. We build a max-heap from the array.
2. We start with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position.
3. We discard this last node by decreasing the heap size and calling MAX-HEAPIFY on the new and possibly incorrectly placed root.
4. We repeat this process until only one node (the smallest element) remains in the correct place.

# 4.1.3. Heapsort

- Demonstrate a heapsort with A=[7, 4, 3, 1, 2].
- On board

# 4.1.3. Heapsort

```
HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for (n >= j >= 2)
    swap A[1] and A[j]
    MAX-HEAPIFY(A, j, n)
```

- The running time is $O(n \lg n)$.
- Compare to Insertion Sort $\Theta(n^2)$ and Merge-Sort $\Theta(n \lg n)$
  - The heapsort is faster than the insertion sort and slower than the merge sort.
  - But it does not require massive recursion (less split) and multiple arrays (one subproblem).

# 4.1.4. Priority Queues

- Priority queue:
  - Maintains a dynamic set S of elements.
  - Each set element has a key – an associated value.
  - Common dynamic operations:
    - INSERT(S, x): inserts elements x into set S.
    - MAXIMUM(S): returns an element of S with the largest key.
    - EXTRACT-MAX(S): removes and returns element of S with the largest key.
    - INCREASE-KEY(S, x, k): increases value of element x’s key to k assuming k >= x.key.
    - Similarly, we can operate for the minimum key value.

# 4.1.4. Priority Queues

Discussion on board.

# Outline

4.1. Heapsort
- 4.1.1. Tree & Binary Tree
- 4.1.2. Heap
- 4.1.3. Heapsort
- 4.1.4. Priority Queues

4.2. Quick Sort
- 4.2.1. Description of Quicksort
- 4.2.2. Performance of Quicksort
- 4.2.3. Randomized Quicksort
- 4.2.4. Analysis of Quicksort

# 4.2.1. Description of Quicksort

Quicksort has a similar approach as the merge sort.

- It is another sorting algorithm based on the divide-and-conquer process.
- Divide the partition $A[p \ldots r]$ into two subarrays $A[p \ldots q-1]$ and $A[q+1 \ldots r]$.
- Note: Elements will be arranged before the split!
  - each element in $A[p \ldots q-1] \leq A[q]$ and
  - $A[q] \leq A[q+1 \ldots r]$.
- Conquer: We sort the two subarrays by recursive calls to QUICKSORT.
- Combine: No need to combine the subarrays, because they are sorted in place.

```
QUICKSORT(A, p, r) //initial call (A,1,n)
if (p < r) then
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q-1)
    QUICKSORT(A, q+1, r)
```

# 4.2.1. Description of Quicksort

```
PARTITION(A, p, r)
x = A[r] //the last element (call it a pivot)
i = p - 1
for (p <= j <= r-1)
    if (A[j]<= x):
       i = i + 1
       swap A[i] and A[j]
swap A[i+1] and A[r]
return i+1 //returns q the index to split
```

- PARTITION( ) rearranges the subarray in place before the split.
- PARTITION( ) always selects the last element $A[r]$ in the subarray $A[p \dots r]$ as the pivot (the element around which to partition).
- As the procedure executes, the array is partitioned into four regions which may be empty.

# 4.2.1. Description of Quicksort

A = [8, 1, 6, 4, 0, 3, 9, 5] becomes [1, 4, 0, 3, 5, 8, 9, 6] when PARTITION( A, 1, n) is called.

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for (p <= j <= r-1)
    if (A[j]<= x):
        i = i + 1
        swap A[i] and A[j]
swap A[i+1] and A[r]
return i+1
```

# 4.2.2. Performance of Quicksort

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for (p <= j <= r-1)
    if (A[j]<= x):
        i = i + 1
        swap A[i] and A[j]
swap A[i+1] and A[r]
return i+1
```

**Loop invariant:**

1. Array elements are arranged before a call.
   1. All entries in A[p…i] are < x (A[r]).
   2. All entries in A[i+1,…r-1] are > x.
2. Elements move to the positions to maintain the arrangement rules. (not sorted)
   - The pivot stays at the end of the array.
3. All elements are positioned and arrangement rules are satisfied.

Note: The additional region $A[j \ldots r-1]$ consists of elements that have not yet been processed. We do not yet know how they compare to the pivot element.

# 4.2.2. Performance of Quicksort

The running time of **QUICKSORT** depends on the partitioning of the subarrays.

- **QUICKSORT** is as fast as **MERGE-SORT** if the partitioned subarrays are balanced (even-sized).
- **QUICKSORT** is as slow as **INSERTIONSORT** if the partitioned subarrays are unbalanced (uneven-sized).

**Worst case**: Subarrays completely unbalanced.

- Have 0 elements in one subarray and $n - 1$ elements in the other subarray.
- The recurrence running time:

$$T(n) = T(n-1) + T(0) + \Theta(n) = O(n^2)$$

- See 4.2.4. Analysis of Quicksort
- The running time is like INSERTION-SORT.

# 4.2.2. Performance of Quicksort

**Best case**: Subarrays are always completely balanced.

- Each subarray has $\leq n/2$ elements.
- The recurrence running time:

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + \Theta(n) \\
&= \Theta(n \lg n)
\end{aligned}
$$

# 4.2.2. Performance of Quicksort

**Balanced partitioning:**

- Let's assume that PATITION always produces a 9 to 1 split.
- Then the recurrence is

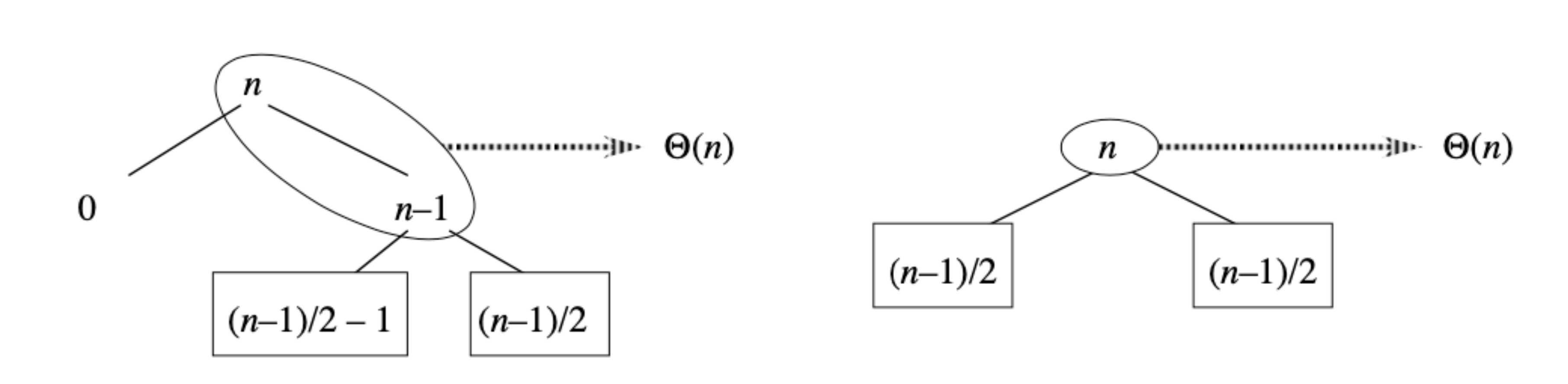
$$T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n) = O(n \lg n)$$

- As long as it's a constant, the base of the log does not matter in asymptotic notation.
- Any split of constant proportionality will yield a recursion tree of depth $\Theta(\lg n)$.

# 4.2.2. Performance of Quicksort

**Average case:**

- Splits in the recursion tree will not always be constant.
- There will usually be a mix of "good" and "bad" splits throughout the recursion tree.
- It does not affect the asymptotic running time – assume that levels alternate between best- and worst-case splits.
- The bad split only adds to the constant hidden in $\Theta$ notation.
- The same number of subarrays to sort but twice as much work is needed.
- Both splits result in $\Theta(n \lg n)$ time, though the constant on the bad split is higher.



# 4.2.3 Randomized Quicksort

- We assumed so far that all input permutations are equally likely which is not always the case.
- We introduce randomization to improve the quicksort algorithm.
- One option would be to use a random permutation of the input array.
- We use random sampling instead which is picking one element at random.
- Instead of using $A[r]$ as the pivot element we randomly pick an element from the subarray.

# 4.2.3 Randomized Quicksort

```
RANDOMIZED-PARTITION(A, p, r)
i = random(p, r)
exchange A[r] and A[i]
return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r:
   q = RANDOMMIZED-PARTITION(A, p, r)
   RANDOMIZED-QUICKSORT(A, p, q-1)
   RANDOMIZED-QUICKSORT(A, q+1, r)
```

# 4.2.4 Analysis of Quicksort

Work on board