(1) $O(1) < O(\log^* n) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

$\log^* n = \min\{ i \geq 0 : log^{(i)} n \leq 1 \}$

(2) complete: except the last, every level should be full. Insert from the left

(3) Heap: ① root at index 0
① left child: left[i] = 2i+1 for $i < \frac{h}{2}$
② right — : right[i] = 2i+2 —
③ parent[i] = $\lfloor \frac{i-1}{2} \rfloor$ for i > 0

(4) Ex. Insert 19 to

(5) ex: Extract max for

(6)

	Time Complexity
BuildMaxHeap	$\Theta(n)$
Maximum	$\Theta(1)$
Extract Max	$\Theta(\log n)$
Delete	$\Theta(\log n)$
IncreaseKey	$\Theta(\log n)$
Heap Sort	$\Theta(n \log n)$

(7) Insertion sort

```
i ← 1
while i < length(A)
    x ← A[i]
    j ← i - 1
    while j >= 0 and A[j] > x
        A[j+1] ← A[j]
        j ← j - 1
    end while
    A[j+1] ← x
    i ← i + 1
end while
```

① Best case: $O(n)$
② Worst case: $O(n^2)$

(8) ex: 9 7 6 3 → 7 9 6 3 → 7 6 9 3 → 6 7 9 3 → 6 7 3 9 → ... → 3 6 7 9

Compare j and j+1

(9)
```
Successor(x):
  if x.right ≠ NIL:
      return TreeMinimum(x.right)
  y ← x.p
  while y ≠ NIL and x = y.right   #
      x = y
      y = y.p                     # keep go
  return y
```

(10)

	Unsorted list	Sorted Array	BST	Balanced BST
Search	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
Insert	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
Delete	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$

(11) ex: Insert 3, 2, 1, 4, 5, 6, 7, 16, 15, 14

(12) ex: delete 15 from

(13) (2) delete 2 from

(14) ① Las Vegas algorithm:
Deterministic answer, random time always correct, other f
② Monte Carlo algorithm:
— nondeter, — answer — fast, —

Prime # theorem:
In range [1, m], there are roughly $m/\ln(m)$ prime #s

(15) (i). For Max-in-Left(t),

(i). For AVL-Insert(root, x)
The general idea is that Update-Left(t, x) updates Max-in-Left(t) by choosing the bigger one from (the original Max-in-Left.key, x) if root.key < x. Notice that it takes $\Theta(1)$.

Thus, recursively, when x < root.key, we would insert x to the left subtree, which would affect Max-in-Left(root).
If there is no rebalance need, we could just call Update-Left(root, x).
If there is a rebalance need, we should go to the AVL-REBALANCE-TO-THE-RIGHT(root).
Thus, we need look at AVL-ROTATE-TO-THE-LEFT and AVL-ROTATE-TO-THE-RIGHT.
We only need to update root.left in AVL-ROTATE-TO-THE-LEFT, and root in AVL-ROTATE-TO-THE-RIGHT, which still takes a constant time.
Thus, the running time of Insert is still $\Theta(log(n))$

(16) chaining way. ① load factor, α: $\frac{n}{m}$
② unsuccessful search $\Theta(1+\alpha)$; succ. search:

$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} X_{ij}\right)\right]$
$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{n-i}{m}\right) = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}$
Thus, $\Theta(1+\alpha)$

(17) ① division: $h(k) = k \bmod m$ (choose prime) (2^r bad)
② multiplication: ex: let $m = 2^3$, $k = 110101$
A: 0.101101
$k \times A$: 100110.010111
shift .010111 by 3 bits to the right
Take integer part: $h(110101) = 011$

(18) ① Linear: $h(k,i) = (h(k) + i) \bmod m$ (easy probe seq)
② quadratic: $h = (h(k) + c_1 i^2 + c_2 i) \bmod m$,
$c_1, c_2 \neq 0$ (not probe)

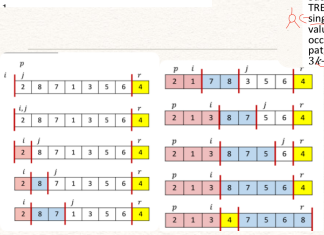
(19) ③ double: $h = (h_1(k) + i\,h_2(k)) \bmod m$ (more probe seq)

Assuming simple uniform hashing, the average-case number of probes in an unsuccessful search is $1/(1-\alpha)$.

For a successful search it is $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$

(20) ① Quick sort

```
Partition(A, p, r)
{
  x = A[r]          // x the last element is pivot
  i = p - 1
  for j = p to r - 1
  {
    do if A[j] ≤ x
       then {
            i = i + 1
            exchange A[i] ↔ A[j]
            }
  }
  exchange A[i + 1] ↔ A[r]
  return i + 1
}
```



```
Quicksort(A, p, r)
{
  if (p < r)
  {
    q = Partition(A, p, r)
    Quicksort(A, p, q − 1)
    Quicksort(A, q + 1, r)
  }
}
```

• Initial call is **Quicksort**$(A, 1, n)$, where n in the length of A

(21) An alternative method of performing an inorder tree walk of an n-node binary search tree finds the minimum element in the tree by calling TREE-MINIMUM and then making n-1 calls to TREE-SUCCESSOR. Prove that this algorithm runs in $\Theta(n)$ time.

To show this bound on the runtime, we will show that using this procedure we traverse each edge twice. This will suffice because the number of edges in a tree is one less than the number of vertices.
Consider a vertex x. Then we have the edge between $x.p$ and x gets used when successor is called on $x.p$ and gets used again when it is called on the largest element in the subtree rooted at x. Since these are the only two times that the edge can be used, apart from the initial finding of the tree min. We have that the runtime is $O(n)$. We trivially get the runtime is $\Omega(n)$ because that is the size of the input.

(22) Prove that no matter what node we start at in a height-h binary search tree, k successive calls to TREE-SUCCESSOR takes $O(k + h)$ time.

Let x be the node on which we have called TREE-SUCCESSOR and y be the kth successor of x. Let z be the lowest common ancestor of x and y. Successive calls will never traverse a single edge more than twice since TREE-SUCCESSOR acts like a tree traversal, so we will never examine a single vertex more than three times. Moreover, any vertex whose key value isn't between x and y will be examined at most once, and it will occur on a simple path from x to z or y to z. Since the lengths of these paths are bounded by h, the running time can be bounded by $3k+2h=O(k+h)$.

(23) ① TreeDelete(root, x)
② Delete node x from BST rooted at root while maintaining BST property, return the new root of the modified tree

(24) Pseudo code

```
TREE − DELETE(T, x)
if x.left == NIL
    TRANSPLANT(T, x, x.right)
elseif x.right == NIL
    TRANSPLANT(T, x, x.left)
else y = TREE − MINIMUM(x.right)
    if y.p ≠ x
        TRANSPLANT(T, y, y.right)
        y.right = x.right
        y.right.p = y
    TRANSPLANT(T, x, y)
    y.left = x.left
    y.left.p = y
```

Where TRANSPLANT is

```
TRANSPLANT(T, x, y)
if x.p == NIL
    T.Root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
if y ≠ NIL
    y.p = x.p
```