Let $a \leq 1$ and $b > 1$ be constants. Also, let $T(n)$ be a function defined on the nonnegative integers by the following recurrence:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Then, $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O\left(n^{\log_b(a)-\varepsilon}\right)$ for some constant $\varepsilon > 0$, then $T(n) = \Theta\left(n^{\log_b(a)}\right)$.
2. If $f(n) = \Theta\left(n^{\log_b(a)}\right)$, then $T(n) = \Theta\left(n^{\log_b(a)} \log(n)\right)$.
3. If $f(n) = \Omega\left(n^{\log_b(a)+\varepsilon}\right)$ for some constant $\varepsilon > 0$, and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. Note that the second condition holds for most functions.

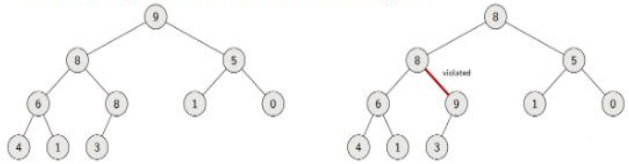
A priority queue maintains a dynamic set S of elements, where each element has a key (an associated value that regulates its importance). This is a more constraining datastructure than arrays, since we cannot access any element.
We want to have the following operations:

- `Insert(S, x)`: inserts the element x into S.
- `Maximum(S)`: Returns the element of S with the largest key.
- `Extract-Max(S)`: removes and returns the element of S with the largest key.
- `Increase-Key(S, x, k)`: increases the value of element x's key to k, assuming that k is greater than its current key value.

Usage Priority queue have many usage, the biggest one will be in Dijkstra's algorithm, which we will see later in this course.

A **heap** (or max-heap) is a nearly-complete binary tree such that, for every node i, the key (value stored at that node) of its children is less than or equal to its key.

Examples For instance, the nearly complete binary tree of depth 3 of the left is a max-heap, but not the one on the right:




```
procedure maxHeapify(A, i, n):
    l = left(i)  // 2i
    r = right(i)  // 2i + 1
    if l <= n and A[l] > A[i]  // don't want an overflow, so check l <= n
        largest = l
    else
        largest = i
    if r <= n and A[r] > A[largest]
        largest = r

    if largest != i
        swap(A, i, largest)  // swap A[i] and A[largest]
        maxHeapify(A, largest, n)
```

dans le cas d'une violation à i on rétablit en O(hauteur de i)

Building a heap To make a heap from an unordered array A of length n, we can use the following `buildMaxHeap` procedure:

```
procedure buildMaxHeap(A, n)
    for i = floor(n/2) downto 1
        maxHeapify(A, i, n)        O(n)
```

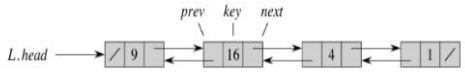
O(log n) car au plus h(i) c'est log(n)

Heapsort Now that we built our heap, we can use it to sort our array:

```
procedure heapsort(A, n)
    buildMaxHeap(A, n)
    for i = n downto 2
        swap(A, 1, i)  // swap A[1] and A[i]    O(n log n) + O(n log n)
        maxHeapify(A, 1, i-1)
```

INSERTIONSORT(A, n)

1: **for** $j = 2, \ldots, n$
2: $\quad k \leftarrow A[j]$
3: $\quad i \leftarrow j - 1$
4: $\quad$ **while** $i > 0$ and $A[i] > k$
5: $\quad\quad A[i+1] \leftarrow A[i]$
6: $\quad\quad i \leftarrow i - 1$
7: $\quad A[i+1] \leftarrow k$
8: $\quad$ **print**(A)



To implement `Increase-Key`, after having changed the key of our element, we can make it go up until its parent has a bigger key that it.

```
procedure HeapIncreaseKey(A, i, key)
    if key < A[i]:
        error "new key is smaller than current key"
    A[i] = key
    while i > 1 and A[Parent(i)] < A[i]
        exchange A[i] with A[Parent(i)]
        i = Parent(i)
```

This looks a lot like max-heapify, and it is thus $O(\log(n))$.
Note that if wanted to implement `Decrease-Key`, we could just run `Max-Heapify` on the element we modified.

To insert a new key into heap, we can increment the heap size, insert a new node in the last position in the heap with the key $-\infty$, and increase the $-\infty$ value to `key` using `Heap-Increase-Key`.

```
procedure HeapInsert(A, key, n)
    n = n + 1  // this is more complex in real life, but
               this is not important here
    A[n] = -infinity
    HeapIncreaseKey(A, n, key)
```

A stack is a data structure where we can insert (`Push(S, x)`) and delete elements (`Pop(S)`). This is known as a last-in, first-out (LIFO), meaning that the element we get by using the `Pop` procedure is the one that was inserted the most recently.

Let's consider the operations we can do with a linked list.

Search We can search in a linked list just as we search in an unsorted array:

```
procedure List-Search(L, k)
    x = L.head
    while x != nil and x.key != k
        x = x.next
    return x
```

We can note that, if k cannot be found, then this procedure returns `nil`. Also, clearly, this procedure is $O(n)$.

Insertion We can insert an element to the first position of a double-linked list by doing:

```
procedure List-Insert(L, x)
    x.next = L.head
    if L.head != nil
        L.head.prev = x
    L.head = x
    x.prev = nil
```

To push an element in our array, we can do:

```
procedure Push(S, x):
    S.top = S.top + 1
    S[S.top] = x
```

Note that, in reality, we would need to verify that we have the space to add one more element, not to get an `IndexOutOfBoundException`.
We can notice that this is executed in constant time.

Popping element is very similar to pushing:

```
procedure Pop(S, x):
    if StackEmpty(S)
        error "underflow"
    S.top = S.top - 1
    return S[S.top + 1]
```

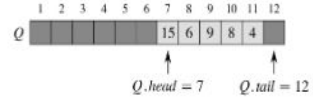
We can notice that this is also done in constant time.

A queue is a data structure where we can insert elements (`Enqueue(Q, x)`) and delete elements (`Dequeue(Q)`). This is known as a first-in, first-out (FIFO), meaning that the element we get by using the `Dequeue` procedure is the one that was inserted the least recently.

Intuition This is really like a queue in real life: people that get out of the queue are people who were there for the longest.

Usage Queues are also used a lot, for instance in packet switches in the internet.

We have an array Q, a pointer `Q.head` to the first element of the queue, and `Q.tail` to the place after the last element.



Enqueue To insert an element, we can simply use the `tail` pointer, making sure to have it wrap around the array if needed:

```
procedure Enqueue(Q, x)
    Q[Q.tail] = x
    if Q.tail == Q.length
        Q.tail = 1
    else
        Q.tail = Q.tail + 1
```

Note that, in real life, we must verify upon overflow. We can observe that this procedure is executed in constant time.

Dequeue To get an element out of our queue, we can use the `head` pointer:

```
procedure Dequeue(Q)
    x = Q[Q.head]
    if Q.head == Q.length
        Q.head = 1
    else
        Q.head = Q.head + 1
```

Given a pointer to an element x, we want to remove it from L:

```
procedure List-Delete(L, x)
    if x.prev != nil
        x.prev.next = x.next
    else
        L.head = x.next
    if x.next != nil
        x.next.prev = x.prev
```

We are basically just rewiring the pointers, by making sure we do not modify things that don't exist.
When we are working with a linked list with sentinel, this algorithm becomes very simple:

```
procedure Sentinel-List-Delete(L, x):
    x.prev.next = x.next
    x.next.prev = p.prev
```

Both those implementations run in $O(1)$.

```
MergeSort(A, p, r):
    // p is the beginning index and r is the end index; they represent the
        section of the array we try to sort.
    if p < r:  // base case
        q = floor((p + r)/2)  // divide
        MergeSort(A, p, q)  // conquer
        MergeSort(A, q+1, r)  // conquer
        Merge(A, p, q, r)  // combine
```

```
Merge(A, p, q, r):
    // p is the beginning index of the first subarray, q is the beginning
        index of the second subarray and r is the end of the second
        subarray
    n1 = q - p + 1 // number of elements in first subarray
    n2 = r - q  // number of elements in second subarray
    let L[1...(n1+1)] and R[1...(n2+1)] be new arrays
    for i = 1 to n1:
        L[i] = A[p + i - 1]
    for j = 1 to n2:
        R[j] = A[q + j]
    L[n1 + 1] = infinity
    L[n2 + 1] = infinity
    // Merge the two created subarrays.
   i = 1
   j = 1
   for k = p to r:
       // Since both subarrays are sorted, the next element is one of L[i]
            or R[j]
       if L[i] <= R[j]:
           A[k] = L[i]
           i = i + 1
       else:
           A[k] = R[j]
           j = j + 1
```

```
procedure Transplant(T, u, v):
    // u is root
    if u.p == Nil:
        T.root = v
    // u is the left child of its parent
    else if u == u.p.left:
        u.p.left = v
    // u is the right child of its parent
    else:
        u.p.right = v

    if v != Nil:
        v.p = u.p
```

We can then write our deletion procedure:

```
procedure TreeDelete(T, z):
    // z has no left child
    if z.left == Nil:
        Transplant(T, z, z.right)
    // z has just a left child
    else if z.right == Nil:
        Transplant(T, z, z.left)
    // z has two children
    else:
        y = TreeMinimum(z.right) // z's successor
        if y.p != z:
            // y is in z's subtree but is not its root
            Transplant(T, y, y.right)
            y.right = z.right
            y.right.p = y
        // Replace z by y
        Transplant(T, z, y)
        y.left = z.left
        y.left.p = y
```

The theorem above gives us the following recurrence, where $c[i, j]$ is the length of an LCS of X_i and Y_j:

$$c[i,j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1, & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i-1, j], c[i, j-1]), & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Conceptually, for $X = \langle B, A, B, D, BA \rangle$ and $Y = \langle D, A, C, B, C, B, A \rangle$, we are drawing the following table:

i \ j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←	1 ↖	1 ←
2	0	0 ↑	1 ↖	1 ←	1 ↑	1 ↑	1 ↑	2 ↖
3	0	0 ↑	1 ↑	1 ↑	2 ↖	2 ←	2 ↖	2 ↑
4	0	1 ↖	1 ↑	1 ↑	2 ↑	2 ↑	2 ↑	2 ↑
5	0	1 ↑	1 ↑	1 ↑	2 ↖	2 ↑	3 ↖	3 ←
6	0	1 ↑	2 ↖	2 ←	2 ↑	2 ↑	3 ↑	4 ↖

where the numbers are stored in the array `c` and the arrows in the array `b`. For this example, we would give us that the longest common subsequence is $Z = \langle A, B, B, A \rangle$.

Algorithm for Preorder Traversal

1. if the tree is empty
2. Return.
3. else
4. Visit the root.
5. Preorder traverse the left subtree.
6. Preorder traverse the right subtree.

Algorithm for Inorder Traversal

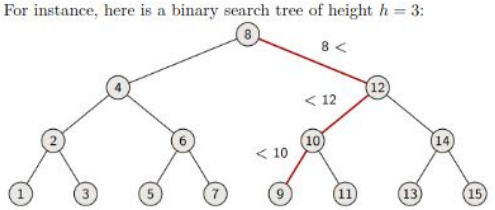
1. if the tree is empty
2. Return.
3. else
4. Inorder traverse the left subtree.
5. Visit the root.
6. Inorder traverse the right subtree.

Algorithm for Postorder Traversal

1. if the tree is empty
2. Return.
3. else
4. Postorder traverse the left subtree.
5. Postorder traverse the right subtree.
6. Visit the root.

A binary search tree is a binary tree (which is *not* necessarily nearly-complete), which follows the following core property: for any node x, if y is in its left subtree then $y.key < x.key$, and if y is in the right subtree of x, then $y.key \geq x.key$.

Example For instance, here is a binary search tree of height $h = 3$:



We could also have the following binary search tree of height $h = 14$:

```
procedure TreeMinimum(x)
    while x.left != Nil
        x = x.left
    return x
```

We can make a very similar procedure to find the maximum element, which also runs in complexity $O(h)$:

```
procedure TreeMaximum(x)
    while x.right != Nil
        x = x.right
    return x
```

We designed this data-structure for searching, so the implementation is not very complicated:

```
procedure TreeSearch(x, k)
    if x == Nil or k == key[x]
        return x
    else if k < x.key
        return TreeSearch(x.left, k)
    else
        return TreeSearch(x.right, k)
```

To insert an element, we can basically search for this element and, when finding its supposed position, we can basically insert it at that position.

```
procedure BinarySearchTreeInsert(T, z)
    y = Nil  // previous Node we looked at
    x = T.root  // current node to look at

    // Search
    while x != Nil
        y = x
        if z.key < x.key
            x = x.left
        else
            x = x.right

    // Insert
    if y == Nil
        T.root = z  // Tree was empty
    else if z.key < y.key
        y.left = z
    else
        y.right = z
```

```
procedure LCSLength(X, Y, m, n):
    // c is the length of an LCS
    // b is the path we take to get our LCS
    let b[1...m, 1...n] and c[0...m, 0...n] be new tables

    // base cases
    for i = 1 to m:
        c[i, 0] = 0
    for j = 0 to n:
        c[0, j] = 0

    // bottom up
    for i = 1 to m:
        for j = 1 to n:
            // Three cases given by recurrence relation
            if X[i] == Y[j]:
                c[i, j] = c[i-1, j-1] + 1
                b[i, j] = (-1, -1)
            // max of the two
            else if c[i-1, j] >= c[i, j-1]:
                c[i, j] = c[i-1, j]
                b[i, j] = (-1, 0)
            else:
                c[i, j] = c[i, j-1]
                b[i, j] = (0, -1)
    return c and b
```

$$S_n = \sum_{i=1}^{n} a_i r^{i-1} = a_1 \left(\frac{1 - r^n}{1 - r} \right)$$

$$\sum_{k=1}^{\infty} k r^k = \frac{r}{(1-r)^2} \quad \text{for } 0 < r < 1.$$