```
struct SingletonDict{K,V} <: AbstractDict{K,V}
    key::K
    value::V
end
SingletonDict(p::Pair{K,V}) where {K,V} = SingletonDict{K,V}(p.first, p.second)

Base.length(::SingletonDict) = 1
Base.keys(d::SingletonDict) = (d.key,)
Base.values(d::SingletonDict) = (d.value,)
Base.haskey(d::SingletonDict, key) = isequal(d.key, key)
Base.getindex(d::SingletonDict, key) = isequal(d.key, key) ? d.value :
throw(KeyError(key))
Base.get(d::SingletonDict, key, default) = isequal(d.key, key) ? d.value : default

Base.iterate(d::SingletonDict, s = true) = s ? ((d.key => d.value), false) :
nothing

struct VectorDict{K,V} <: AbstractDict{K,V}
    keys::Vector{K}
    values::Vector{V}
end
VectorDict{K,V}() where {K,V} = VectorDict{K,V}(Vector{K}(), Vector{V}())
function VectorDict{K,V}(kv) where {K,V}
    keys = Vector{K}()
    values = Vector{V}()
    if Base.IteratorSize(kv) !== SizeUnknown()
        sizehint!(keys, length(kv))
        sizehint!(values, length(kv))
    end
    for (k,v) in kv
        push!(keys, k)
        push!(values, v)
    end
    return VectorDict{K,V}(keys, values)
end
VectorDict(kv::Pair{K,V}...) where {K,V} = VectorDict{K,V}(kv)
VectorDict(g::Base.Generator) = VectorDict(g...)

Base.length(d::VectorDict) = length(d.keys)
Base.sizehint!(d::VectorDict, newsz) = (sizehint!(d.keys, newsz);
sizehint!(d.values, newsz); return d)

@propagate_inbounds getpair(d::VectorDict, i::Integer) = d.keys[i] => d.values[i]

Base.copy(d::VectorDict) = VectorDict(copy(d.keys), copy(d.values))
Base.empty(::VectorDict, ::Type{K}, ::Type{V}) where {K, V} = VectorDict{K, V}()
Base.empty!(d::VectorDict) = (empty!(d.keys); empty!(d.values); return d)

function Base.delete!(d::VectorDict, key)
    i = findfirst(isequal(key), d.keys)
    if !(i == nothing || i == 0)
        deleteat!(d.keys  , i)
        deleteat!(d.values, i)
    end
```

```
    return d
end

Base.keys(d::VectorDict) = d.keys
Base.values(d::VectorDict) = d.values
Base.haskey(d::VectorDict, key) = key in d.keys
function Base.getindex(d::VectorDict, key)
    i = findfirst(isequal(key), d.keys)
    @inbounds begin
        return i !== nothing ? d.values[i] : throw(KeyError(key))
    end
end
function Base.setindex!(d::VectorDict, v, key)
    i = findfirst(isequal(key), d.keys)
    if i === nothing
        push!(d.keys, key)
        push!(d.values, v)
    else
        d.values[i] = v
    end
    return d
end

function Base.get(d::VectorDict, key, default)
    i = findfirst(isequal(key), d.keys)
    @inbounds begin
        return i !== nothing ? d.values[i] : default
    end
end

function Base.iterate(d::VectorDict, s = 1)
    @inbounds if s > length(d)
        return nothing
    else
        return (d.keys[s] => d.values[s]), s+1
    end
end

struct SortedVectorDict{K,V} <: AbstractDict{K,V}
    keys::Vector{K}
    values::Vector{V}
    function SortedVectorDict{K,V}(pairs::Vector{Pair{K,V}}) where {K,V}
        if !issorted(pairs, by=first)
            pairs = sort(pairs, by=first)
        end
        return new{K,V}(map(first, pairs), map(last, pairs))
    end
    SortedVectorDict{K,V}(keys::Vector{K}, values::Vector{V}) where {K,V} =
        new{K,V}(keys, values)
    SortedVectorDict{K,V}() where {K,V} = new{K,V}(Vector{K}(undef, 0),
Vector{V}(undef, 0))
end
SortedVectorDict{K,V}(kv::Pair{K,V}...) where {K,V} = SortedVectorDict{K,V}(kv)
function SortedVectorDict{K,V}(kv) where {K,V}
```

```
    d = SortedVectorDict{K,V}()
    if Base.IteratorSize(kv) !== SizeUnknown()
        sizehint!(d, length(kv))
    end
    for (k,v) in kv
        push!(d, k=>v)
    end
    return d
end
SortedVectorDict(pairs::Vector{Pair{K,V}}) where {K,V} =
SortedVectorDict{K,V}(pairs)
SortedVectorDict(kv::Pair{K,V}...) where {K,V} = SortedVectorDict{K,V}(kv)
SortedVectorDict(g::Base.Generator) = SortedVectorDict(g...)

Base.length(d::SortedVectorDict) = length(d.keys)
Base.sizehint!(d::SortedVectorDict, newsz) =
    (sizehint!(d.keys, newsz); sizehint!(d.values, newsz); return d)

Base.copy(d::SortedVectorDict{K,V}) where {K,V} =
    SortedVectorDict{K,V}(copy(d.keys), copy(d.values))
Base.empty(::SortedVectorDict, ::Type{K}, ::Type{V}) where {K, V} =
SortedVectorDict{K, V}()
Base.empty!(d::SortedVectorDict) = (empty!(d.keys); empty!(d.values); return d)

# _searchsortedfirst(v::Vector, k) = searchsortedfirst(v, k)
function _searchsortedfirst(v::Vector, k)
    i = 1
    @inbounds while i <= length(v) && isless(v[i], k)
        i += 1
    end
    return i
end

function Base.delete!(d::SortedVectorDict{K}, k) where {K}
    key = convert(K, k)
    if !isequal(k, key)
        return d
    end
    i = _searchsortedfirst(d.keys, key)
    if i <= length(d) && isequal(d.keys[i], key)
        deleteat!(d.keys, i)
        deleteat!(d.values, i)
    end
    return d
end

Base.keys(d::SortedVectorDict) = d.keys
Base.values(d::SortedVectorDict) = d.values
function Base.haskey(d::SortedVectorDict{K}, k) where {K}
    key = convert(K, k)
    if !isequal(k, key)
        return false
    end
    i = _searchsortedfirst(d.keys, key)
```

```
    return (i <= length(d) && isequal(d.keys[i], key))
end
function Base.getindex(d::SortedVectorDict{K}, k) where {K}
    key = convert(K, k)
    if !isequal(k, key)
        throw(KeyError(k))
    end
    i = _searchsortedfirst(d.keys, key)
    @inbounds if (i <= length(d) && isequal(d.keys[i], key))
        return d.values[i]
    else
        throw(KeyError(key))
    end
end
function Base.setindex!(d::SortedVectorDict{K}, v, k) where {K}
    key = convert(K, k)
    if !isequal(k, key)
        throw(ArgumentError("$k is not a valid key for type $K"))
    end
    i = _searchsortedfirst(d.keys, key)
    if i <= length(d) && isequal(d.keys[i], key)
        d.values[i] = v
    else
        insert!(d.keys, i, key)
        insert!(d.values, i, v)
    end
    return d
end

function Base.get(d::SortedVectorDict{K}, k, default) where {K}
    key = convert(K, k)
    if !isequal(k, key)
        return default
    end
    i = _searchsortedfirst(d.keys, key)
    @inbounds begin
        return (i <= length(d) && isequal(d.keys[i], key)) ? d.values[i] : default
    end
end
function Base.get(f::Union{Function,Type}, d::SortedVectorDict{K}, k) where {K}
    key = convert(K, k)
    if !isequal(k, key)
        return f()
    end
    i = _searchsortedfirst(d.keys, key)
    @inbounds begin
        return (i <= length(d) && isequal(d.keys[i], key)) ? d.values[i] : f()
    end
end
function Base.iterate(d::SortedVectorDict, i = 1)
    @inbounds if i > length(d)
        return nothing
    else
        return (d.keys[i] => d.values[i]), i+1
```

```
        end
    end
```