

Principled Programming

Introduction to Coding in Any Imperative Language

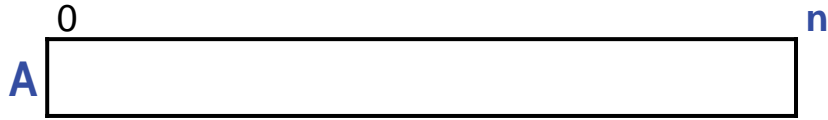
Tim Teitelbaum

Emeritus Professor

Department of Computer Science

Cornell University

Collections



Until now, we have searched in arrays, and have rearranged their values, but the set of values stored in the array has been static, i.e., fixed and unchanging during program execution.

A *collection* is a set of values that is dynamic, i.e., its size and members change during execution.

We consider three ways to represent a collection:

- Lists
- Histograms
- Hash Tables

The need to dynamically increase the size of an array leads naturally to the revelation that arrays are *objects*, and to a discussion of Two-Dimensional Arrays.



The following data structure, known as a *list*, represents a collection of integers.

```
# A[0..size-1] are the current items in A[0..n-1],  $0 \leq size \leq n$ .
A: list[int]    # A[] is the receptacle for int items of a list.
size: int      # size is the current number of elements in A[],  $0 \leq size \leq maxSize$ .
max_size: int  # n is the maximum number of elements storable in the list A[].
```

- The number of items in the list at any given moment is `size`.
- The number of items that can be stored in the list is limited by `n`, the length of the array.
- The items in the list are stored in `A[0..size-1]`.
- Array elements `A[size..n-1]` are unused, and are available for additional items.
- Because items may repeat, the collection is a *multiset*, i.e., a set with multiplicity.



Given a multiset \mathbf{M} and a value v , we want (at least) these operations:

- **Add** an instance of v to \mathbf{M} , i.e., increase its multiplicity.
- **Remove** an instance of v from \mathbf{M} if it is in \mathbf{M} , i.e., decrease its multiplicity.
- Test **membership** of v in \mathbf{M} , i.e., ask if its multiplicity is greater than zero.
- Obtain the **multiplicity** of v in \mathbf{M} .
- **Enumerate** the elements of \mathbf{M} in an arbitrary order, i.e., list them off, repeating m times an element with multiplicity m .



Add:

```
# Add v to A.
# =====
# Ensure that A has the capacity for another element.
if size == n: #.Make room for more values, or sound an alarm.

# Store v into A, which is known to have capacity.
A[size] = v
size += 1
```



Remove:

```
# Remove v from A.
k = index_of(v, A, size)
if k == size: #v is not in A.
else: A[k] = A[size-1]; size -= 1
```

where `index_of` is a method to find v in $A[0..size-1]$ using sequential-search:

```
def index_of(v: int, A: list[int], size: int) -> int:
    """Return k, a location of v in A, or return size if no v in A."""
    k = 0
    while (k < size) and (A[k] != v): k += 1
    return k
```



Membership:

```
# Set b to True if v is in A, and False otherwise.  
k = index_of(v, A, size)  
b: bool = (k < size)
```



Multiplicity:

```
# Set m to the multiplicity of v in A.  
m = 0  
for k in range(0, size):  
    if A[k] == v: m += 1
```




Enumeration:

```
# Enumerate elements of A.  
for k in range(0, size): #.Do whatever for A[k].
```



Performance:

Operation	Steps
add	constant
remove	worst case linear in size
membership	worst case linear in size
multiplicity	linear in size
enumeration	linear in size

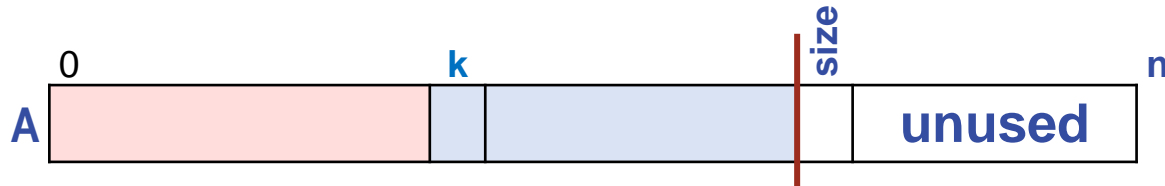


The items in the collection can be maintained as an *indexable* list, sometimes referred to as an *ordered* list.

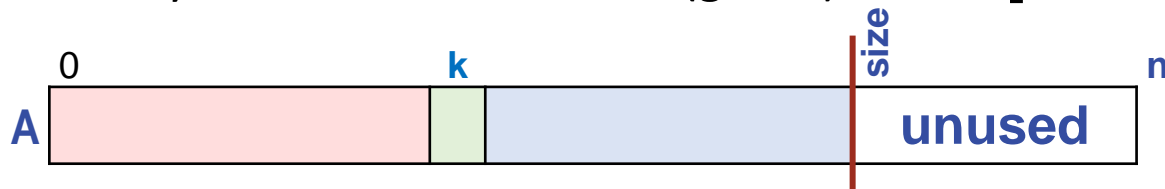
As a consequence, the enumeration of collection members will be in the given order.

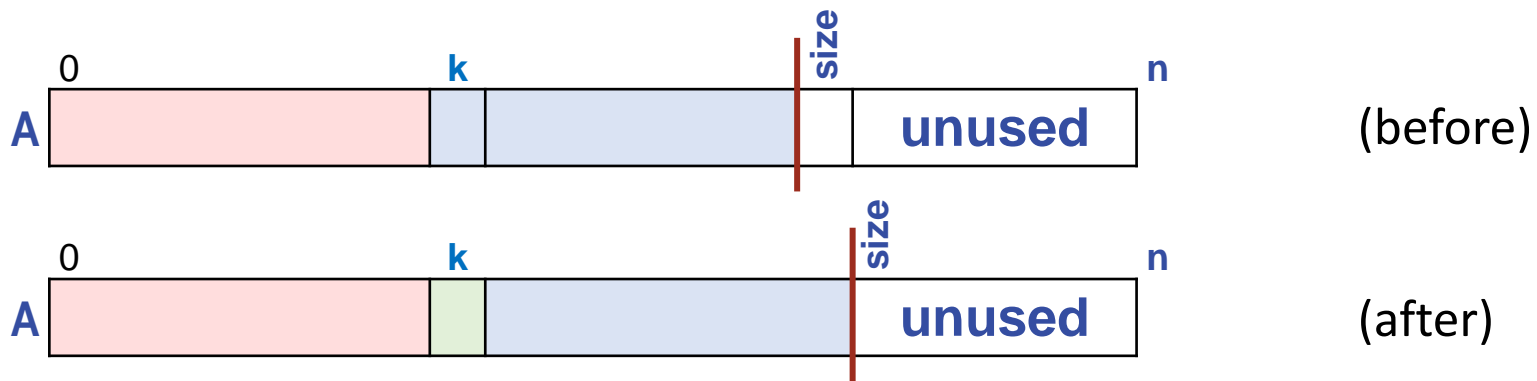
(Note that the ordering is determined by the client, and is not based on the normal arithmetic ordering of values.)

To insert an item (green) at the k^{th} index, first shift $A[k.. \text{size}-1]$ right one element:



Conversely, to delete the k^{th} item (green), shift $A[k+1.. \text{size}-1]$ left one element:





Add into ordered list:

```

;# Add v at position k of A.
# =====
# Check index.
if (k < 0) or (k > size): #.Alert: Bad index.

# Ensure that A has the capacity for another element.
if size == n: #.Make room for more values.

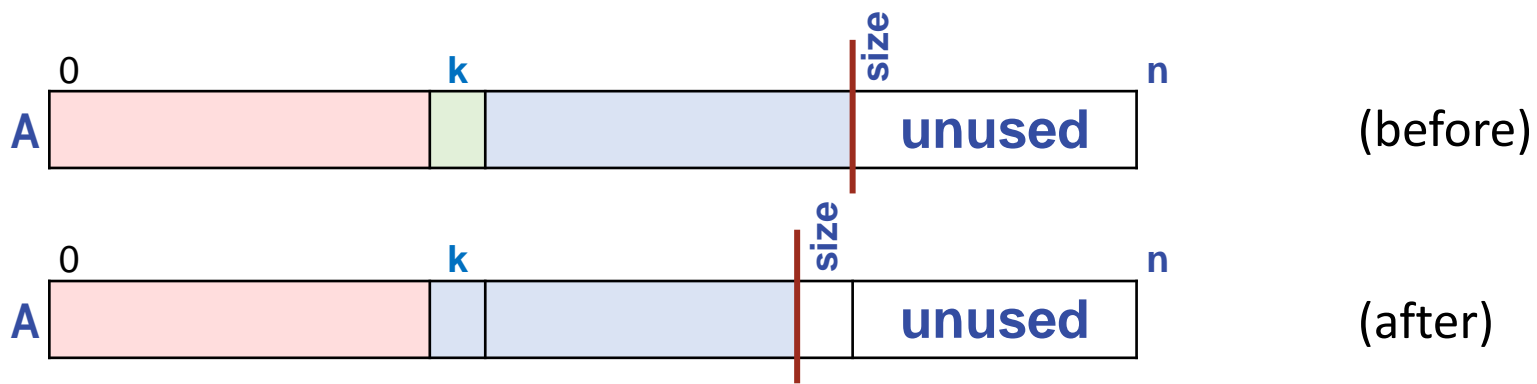
# Shift A[k..size-1] right one place to make room for v.
for j in range(size - 1, k - 1, -1): A[j + 1] = A[j]

# Store v into the hole at A[k] that is now available.
A[k] = v; size += 1

```

Performance:

To add an item at index k requires effort proportional to $size-k$.



Remove from ordered list:

```

# Remove v from position k of A.
# =====
# Check index.
if (k < 0) or (k >= size): #.Alert: Bad index.

# Reduce the number of elements.
size -= 1

# Shift A[k+1..size] left one place to squeeze out v.
for j in range(k, size): A[j] = A[j + 1]

```

Performance:

To remove an item at index k requires effort proportional to $\text{size}-k$.



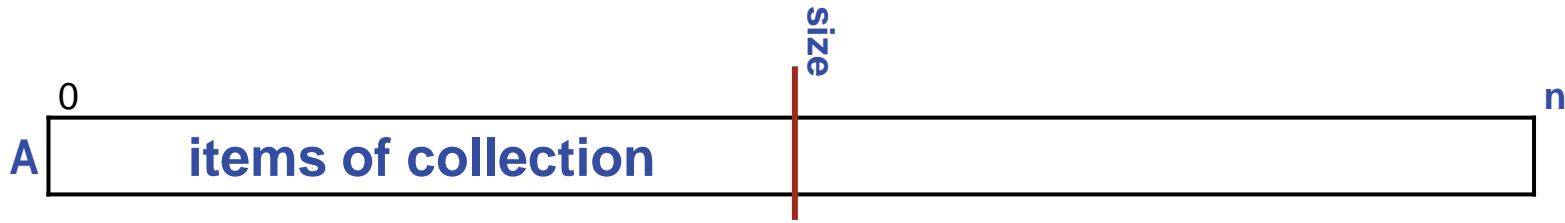
Add: When the array containing items of a collection is full, ...

```
# Add v to A.
# =====
# Ensure that A has the capacity for another element.
if (size == n) #.Make room for more values, or sound an alarm.

# Store v into the hole at A[k] that is known to be available.
A[size] = v;
size += 1
```

*C/C++

This section is not valid for C-style arrays in C/C++. Rather, it can be read as describing one of the alternatives to C-style arrays that are available in C++.



Add: When the array containing items of a collection is full, we wish to increase its capacity.

```
# Add v to A.  
# =====  
# Ensure that A has the capacity for another element.  
if (size == n): A = ensure_capacity(A); n = len(A)  
  
# Store v into the hole at A[size] that is known to be available.  
A[size] = v; size += 1
```

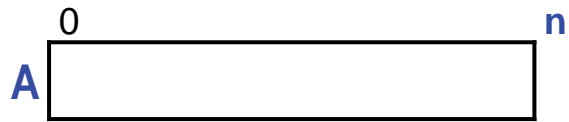
This statement doubles the length of A while retaining its values.

This statement updates n , where we have maintained the length of an array A in a separate variable. We have done so, but didn't need to because it was available as $\text{len}(A)$ all along.

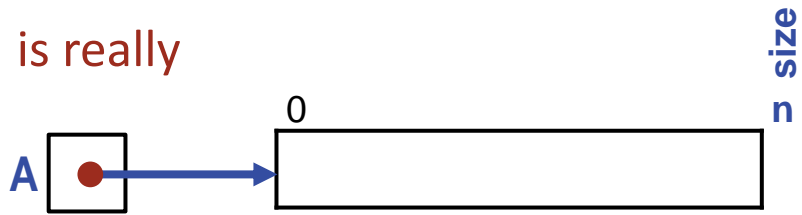
Unbounded Capacity

Add: To understand how this works, know that

What has been depicted as



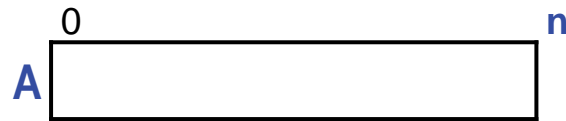
is really



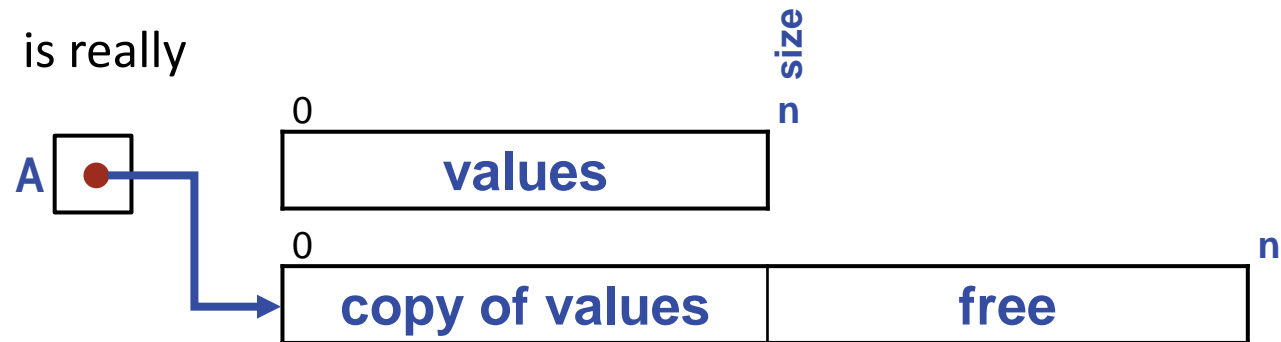
The value in A (the ●) is called a *reference* to an *object* (the **int** array).

Add: To understand how this works, know that

What has been depicted as



is really



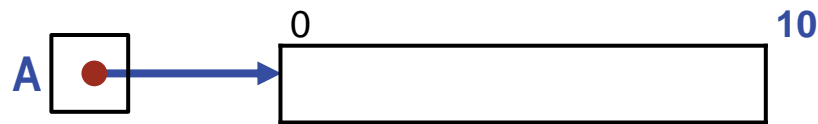
The value in A (the ●) is called a *reference* to an *object* (the **int** array).

The capacity of the array object referred to by A can be doubled by allocating a new **int** array object of twice the length, copying the values from the old object to the new object, and making A refer to the new object.

Aliases: To really understand how this works, consider this code:

```
A: list[int] = [0] * 10  
B: list[int] = A  
A[0] = 7  
B[0] = 8  
print(A[0]) # What does this line print?
```

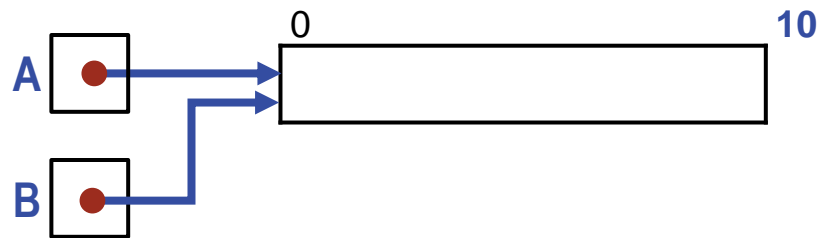
The first line declares `A`, allocates an array object of length 10, and assigns a reference to that object to `A`.



Aliases: To really understand how this works, consider this code.

```
A: list[int] = [0] * 10  
B: list[int] = A  
A[0] = 7  
B[0] = 8  
print(A[0]) # What does this line print?
```

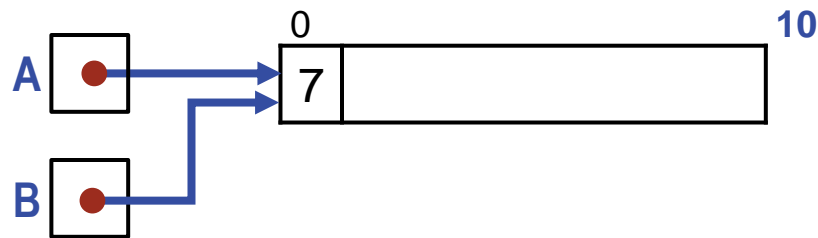
The second line declares B, and assigns the contents of A (the reference) to B.



Aliases: To really understand how this works, consider this code.

```
A: list[int] = [0] * 10  
B: list[int] = A  
A[0] = 7  
B[0] = 8  
print(A[0]) # What does this line print?
```

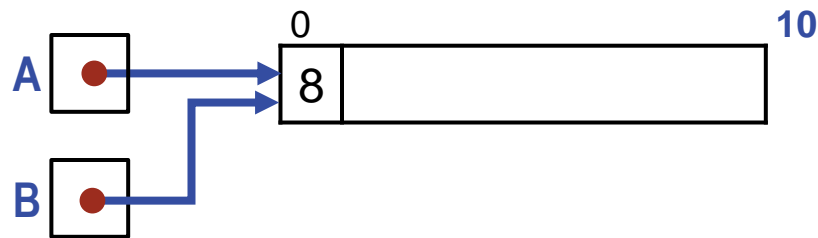
The third line assigns 7 to be the contents of `A[0]`, the 0th variable in the array object referred to by `A`.



Aliases: To really understand how this works, consider this code.

```
A: list[int] = [0] * 10  
B: list[int] = A  
A[0] = 7  
B[0] = 8  
print(A[0]) # What does this line print?
```

The fourth line assigns 8 to be the contents of `B[0]`, the 0th variable in the array object referred to by B.

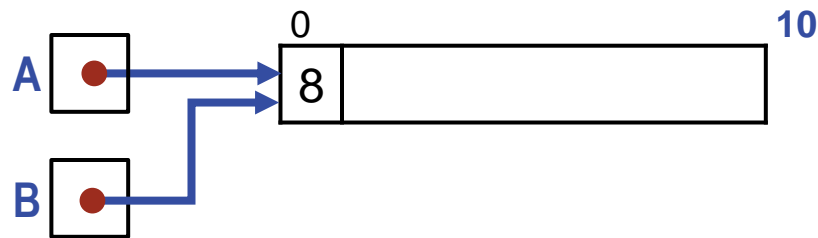


Aliases: To really understand how this works, consider this code.

```
A: list[int] = [0] * 10
B: list[int] = A
A[0] = 7
B[0] = 8
print(A[0]) # What does this line print?
```

The fifth line prints the contents of $A[0]$, the 0th variable in the array referred to by A.

It prints 8.



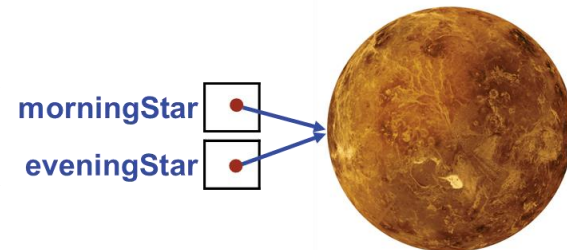
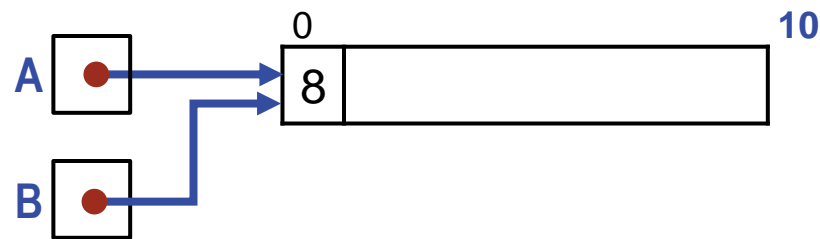
Aliases: To really understand how this works, consider this code.

```
A: list[int] = [0] * 10
B: list[int] = A
A[0] = 7
B[0] = 8
print(A[0]) # What does this line print?
```

The fifth line prints the contents of `A[0]`, the 0th variable in the array referred to by A.

It prints 8.

A and B are aliases that refer to the same object, just as `morningStar` and `eveningStar` are aliases that both refer to the same planet, Venus.

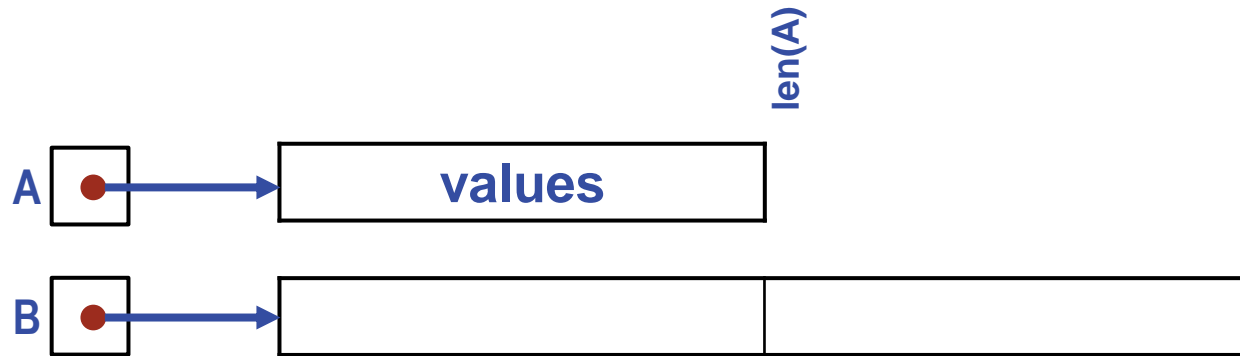


EnsureCapacity:



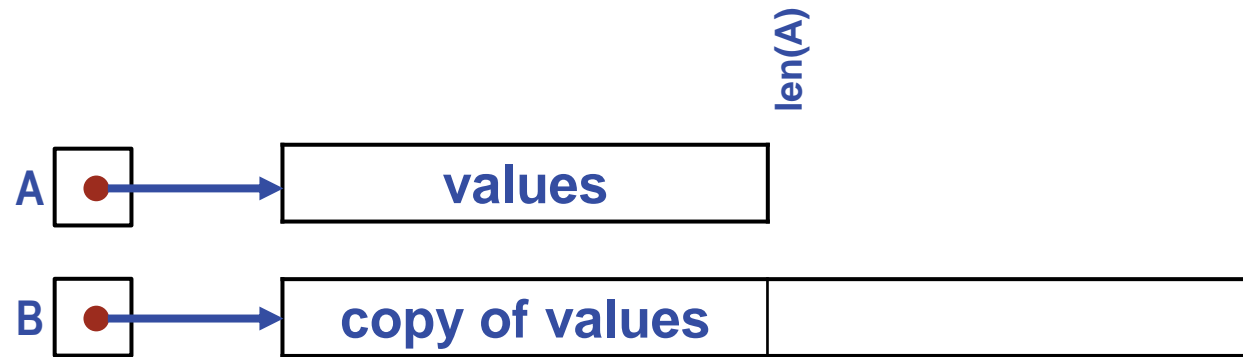
```
def ensure_capacity(A: list[int]) -> list[int]:  
    """Return a reference to a copy of A in an object that is twice as long."""  
    # Make B refer to an object that is twice as long as A.  
    B: list[int] = [0 for _ in range(2 * len(A))]  
  
    # Copy the values from A (the old object) to B (the new object).  
    for k in range(0, len(A)): B[k] = A[k]  
  
    # Return a reference to the copied array B.  
    return B
```


EnsureCapacity:



```
def ensure_capacity(A: list[int]) -> list[int]:  
    """Return a reference to a copy of A in an object that is twice as long."""  
    # Make B refer to an object that is twice as long as A.  
    B: list[int] = [0 for _ in range(2 * len(A))]  
  
    # Copy the values from A (the old object) to B (the new object).  
    for k in range(0, len(A)): B[k] = A[k]  
  
    # Return a reference to the copied array B.  
    return B
```

EnsureCapacity:



```
def ensure_capacity(A: list[int]) -> list[int]:  
    """Return a reference to a copy of A in an object that is twice as long."""  
    # Make B refer to an object that is twice as long as A.  
    B: list[int] = [0 for _ in range(2 * len(A))]  
  
    # Copy the values from A (the old object) to B (the new object).  
    for k in range(0, len(A)): B[k] = A[k]  
  
    # Return a reference to the copied array B.  
    return B
```

EnsureCapacity:

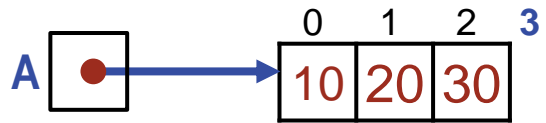
```
def ensure_capacity(A: list[int]) -> list[int]:  
    """Return a reference to a copy of A in an object that is twice as long."""  
    # Make B refer to an object that is twice as long as A.  
    B: list[int] = [0 for _ in range(2 * len(A))]  
  
    # Copy the values from A (the old object) to B (the new object).  
    for k in range(0, len(A)): B[k] = A[k]  
  
    # Return a reference to the copied array B.  
    return B
```

Array parameters: We can now finally understand how array parameters work.

```
A: list[int] = [10, 20, 30]
reverse(A, 0, 2)
```

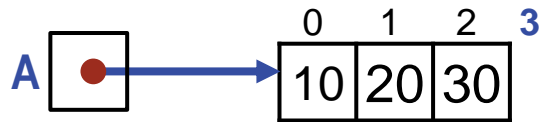
Array parameters: We can now finally understand how array parameters work.

```
A: list[int] = [10, 20, 30]  
reverse(A, 0, 2)
```



Array parameters: We can now finally understand how array parameters work.

```
A: list[int] = [10, 20, 30]
reverse(A, 0, 2)
```



```
reverse(A: list[int], L: int, R: int) -> None:
    while L < R:
        # Swap A[L] and A[R].
        temp:int = A[L]; A[L] = A[R]; A[R] = temp

        L += 1; R -= 1
```

A

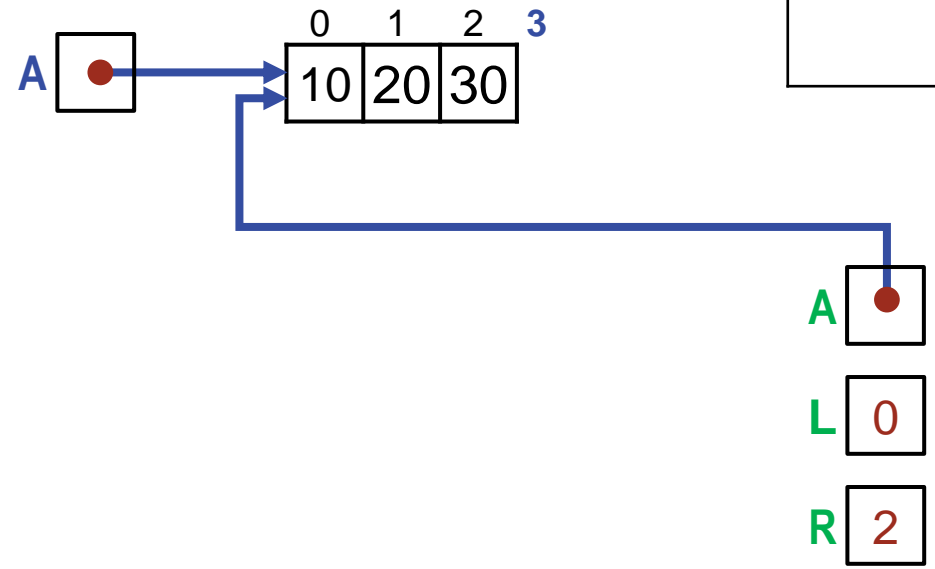
L

R

Array parameters: We can now finally understand how array parameters work.

```
A: list[int] = [10, 20, 30]  
reverse(A, 0, 2)
```

```
reverse(A: list[int], L: int, R: int) -> None:  
    while L < R:  
        # Swap A[L] and A[R].  
        temp:int = A[L]; A[L] = A[R]; A[R] = temp  
  
        L += 1; R -= 1
```

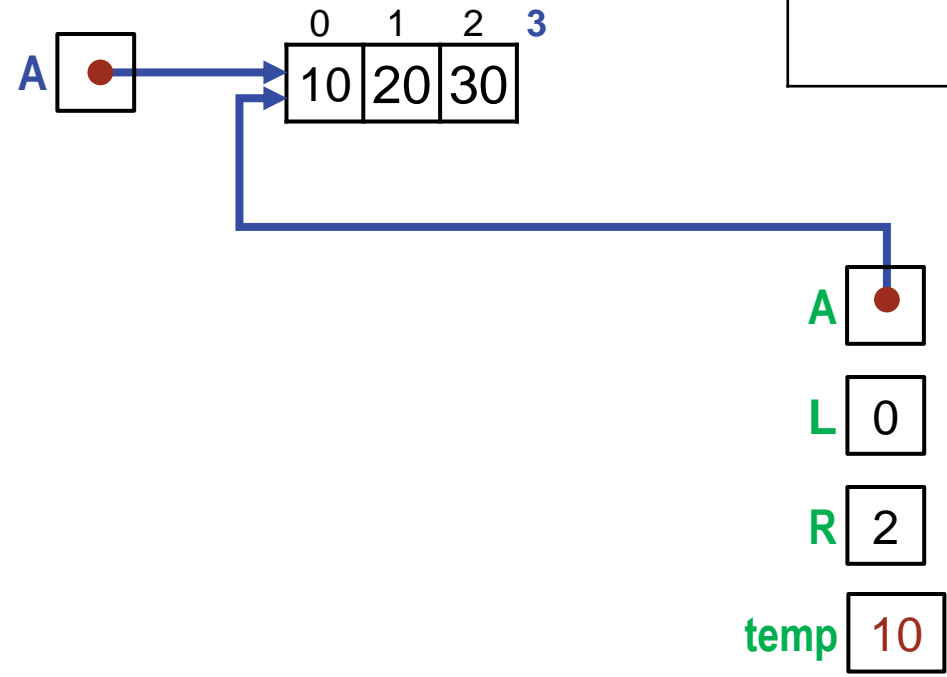


The blue A and the green A are aliases.

Array parameters: We can now finally understand how array parameters work.

```
A: list[int] = [10, 20, 30]  
reverse(A, 0, 2)
```

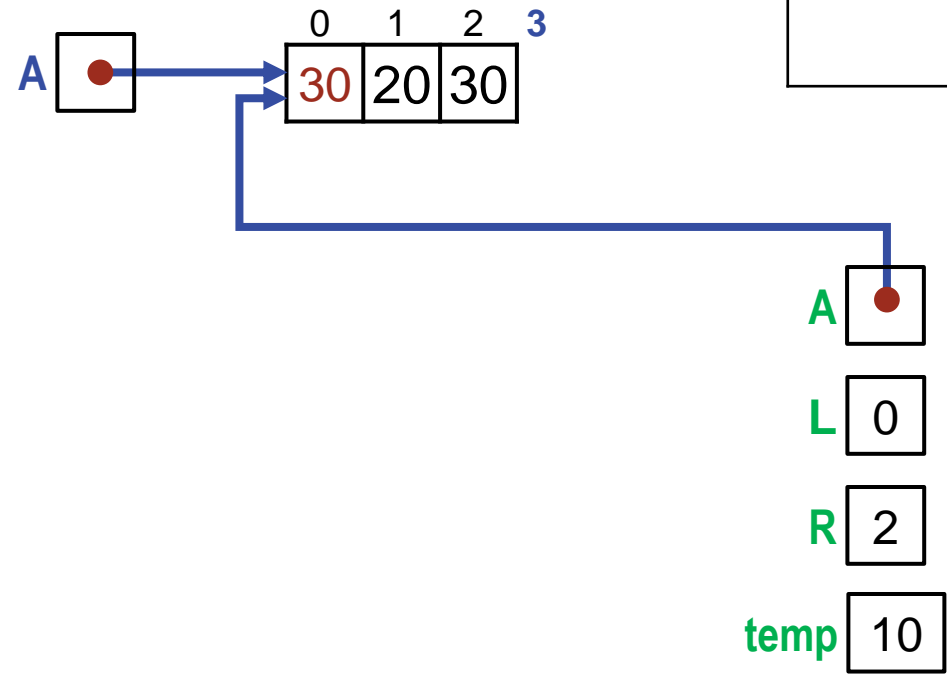
```
reverse(A: list[int], L: int, R: int) -> None:  
    while L < R:  
        # Swap A[L] and A[R].  
        temp:int = A[L]; A[L] = A[R]; A[R] = temp  
  
        L += 1; R -= 1
```



Array parameters: We can now finally understand how array parameters work.

```
A: list[int] = [10, 20, 30]  
reverse(A, 0, 2)
```

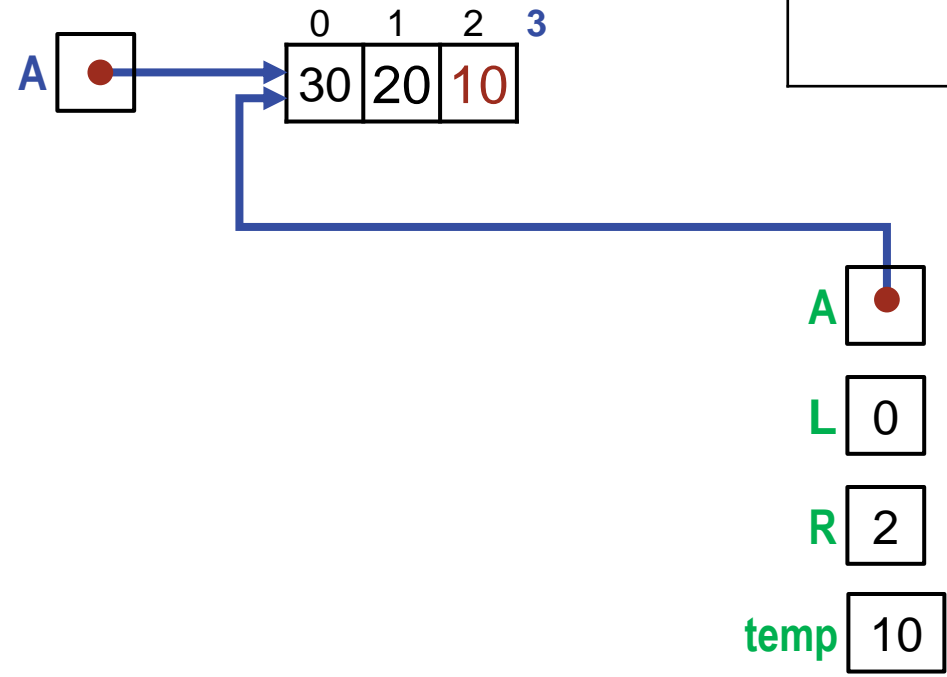
```
reverse(A: list[int], L: int, R: int) -> None:  
    while L < R:  
        # Swap A[L] and A[R].  
        temp:int = A[L]; A[L] = A[R]; A[R] = temp  
  
        L += 1; R -= 1
```



Array parameters: We can now finally understand how array parameters work.

```
A: list[int] = [10, 20, 30]  
reverse(A, 0, 2)
```

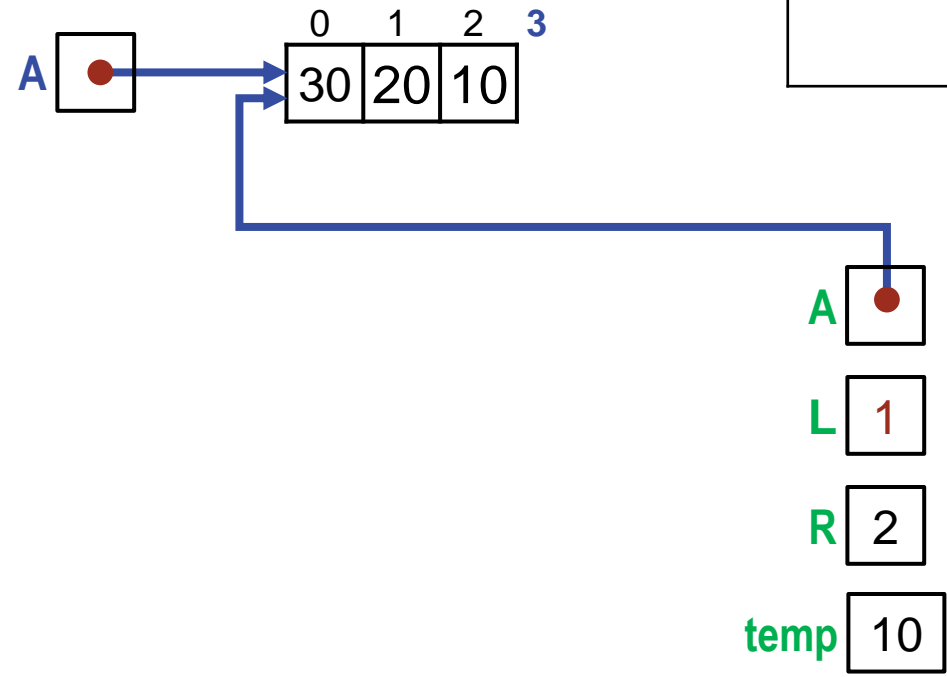
```
reverse(A: list[int], L: int, R: int) -> None:  
    while L < R:  
        # Swap A[L] and A[R].  
        temp:int = A[L]; A[L] = A[R]; A[R] = temp  
  
        L += 1; R -= 1
```



Array parameters: We can now finally understand how array parameters work.

```
A: list[int] = [10, 20, 30]  
reverse(A, 0, 2)
```

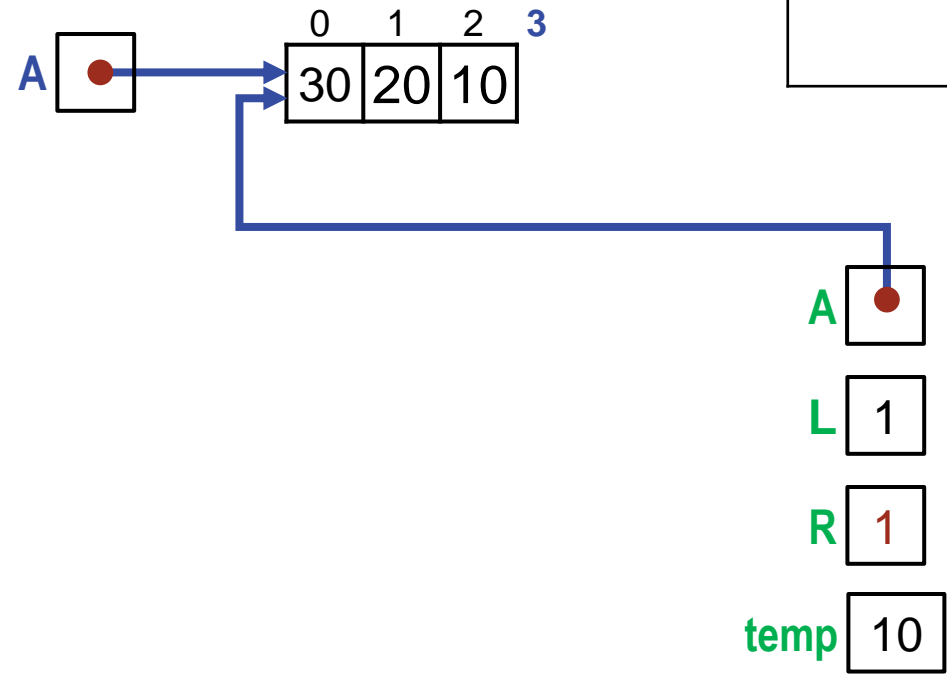
```
reverse(A: list[int], L: int, R: int) -> None:  
    while L < R:  
        # Swap A[L] and A[R].  
        temp:int = A[L]; A[L] = A[R]; A[R] = temp  
  
        L += 1; R -= 1
```



Array parameters: We can now finally understand how array parameters work.

```
A: list[int] = [10, 20, 30]  
reverse(A, 0, 2)
```

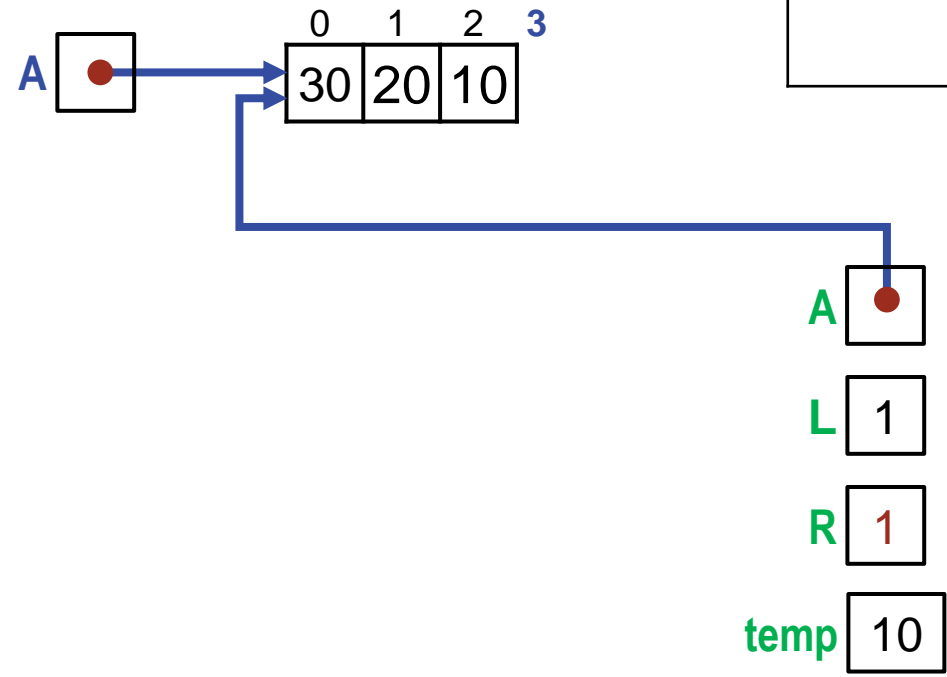
```
reverse(A: list[int], L: int, R: int) -> None:  
    while L < R:  
        # Swap A[L] and A[R].  
        temp:int = A[L]; A[L] = A[R]; A[R] = temp  
  
        L += 1; R -= 1
```



Array parameters: We can now finally understand how array parameters work.

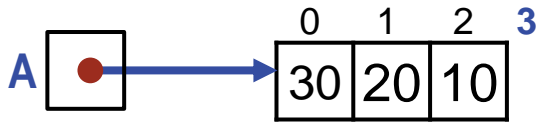
```
A: list[int] = [10, 20, 30]  
reverse(A, 0, 2)
```

```
reverse(A: list[int], L: int, R: int) -> None:  
    while L < R:  
        # Swap A[L] and A[R].  
        temp:int = A[L]; A[L] = A[R]; A[R] = temp  
  
        L += 1; R -= 1
```



Array parameters: We can now finally understand how array parameters work.

```
A: list[int] = [10, 20, 30]  
reverse(A, 0, 2)
```





Critique:

Representing a collection as a list of items in an array is fundamental, and the operations for doing so should be at your ready disposal. Thus, they are presented as patterns that you should master. However, writing such code directly in your program has several drawbacks:

- The collection has no single name, and thus it is not easily manipulated as one thing.
- The collection's implementation details are not hidden, and thus your program can both break the data structure's representation invariant and come to excessively depend on its details.

These limitations are addressed in Chapter 18 Classes and Objects, where the collection implementation is factored into a separate definition: `ArrayList`:

- References to instances of `ArrayList` can be manipulated as one thing.
- The details of an `ArrayList` are hidden using the class's visibility mechanism.

This allows easy replacement of one collection implementation with another.



Python-specific Aside:

Python's `list` type is implemented as the data structure we have described for collections, with array doubling built in, as needed. Thus, instead of writing:

```
# Add v to A.
# =====
# Ensure that A has the capacity for another element.
if size == n: A = ensure_capacity(A); n = len(A)

# Store v into the hole at A[k] that is known to be available.
A[size] = v
size += 1
```

you can just write:

```
# Add v to A.
# =====
# Append v onto A, increasing A's capacity, if needed.
A.append(v)
size += 1
```




Python-specific Aside: continued

For that matter, `size` is also built into Python's internal representation of lists, so instead of writing:

```
# Add v to A.
# =====
# Append v onto A, increasing A's capacity, if needed.
A.append(v)
size += 1
```

you can just write:

```
# Add v to A.
# =====
# Append v onto A, increasing A's capacity, if needed.
A.append(v)
```

and access size as `len(A)`.



Python-specific Aside: continued

Similarly, to remove an item from an (unordered) list, instead of writing:

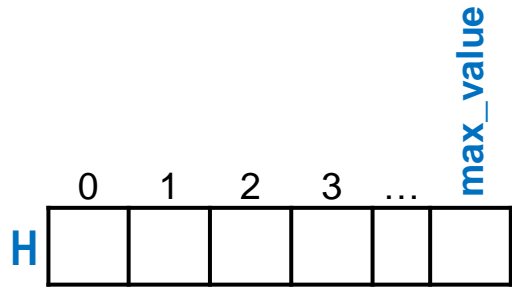
```
# Remove v from A.
k = index_of(v, A, size)
if k == size: # v is not in A.
else: A[k] = A[size - 1]; size -= 1;
```

you can just write:

```
# Remove v from A.
k = A.index(v) # abort execution if v not in A
A[k] = A[len(A) - 1]; del A[len(A)]
```

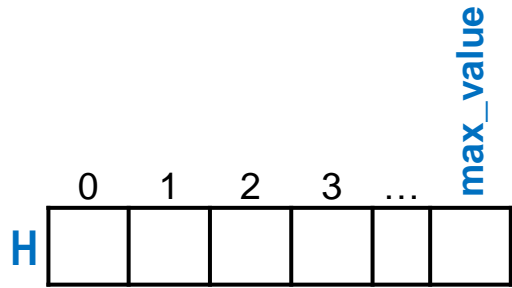
or (if you want to continue execution when an item is not found) write:

```
# Remove v from A.
try: k = A.index(20); A[k] = A[len(A) - 1]; del A[len(A)]
except ValueError: # v is not in A.
```



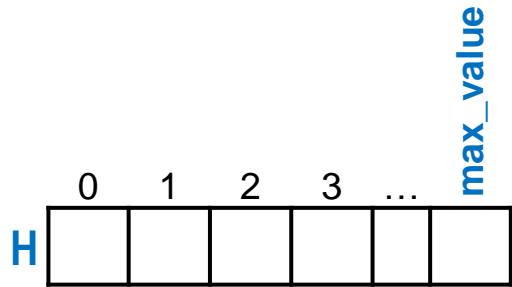
A multiset of integer values in the range 0 through max_value can be represented as a histogram.

```
# Collection of items in range 0..max_value, where multiplicity of v is H[v].  
H: list[int] = [0] * (max_value + 1)
```



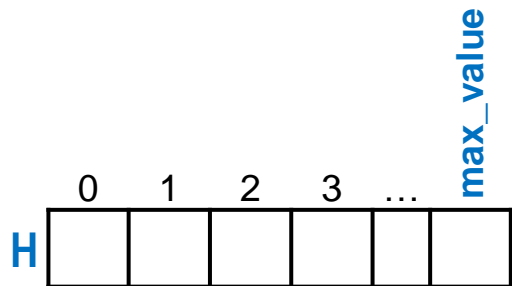
Add:

```
# Add v to H.  
H[v] += 1
```



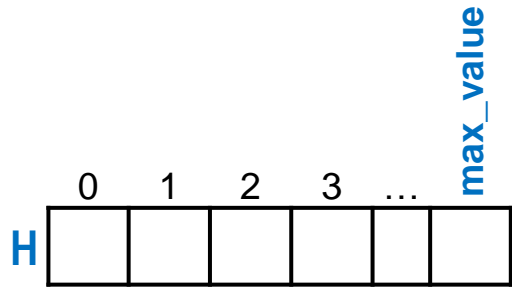
Remove:

```
# Remove v from H.  
if H[k] == 0: #.Alarm: attempt to remove a value not in H.  
else H[k] -= 1
```



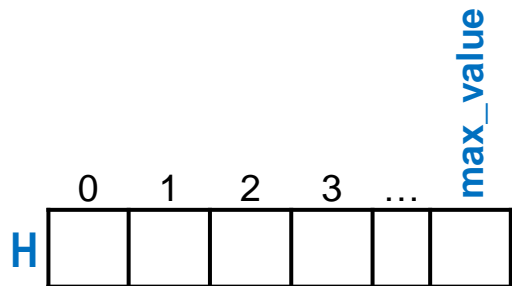
Membership:

```
# Set b to True iff v is in H.  
b: bool = (H[v] > 0)
```



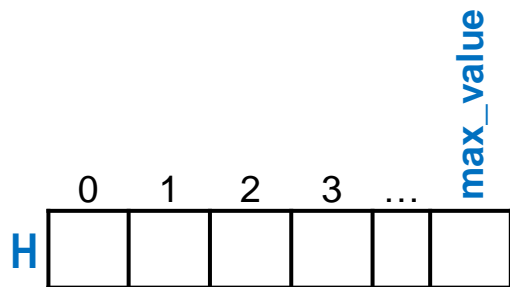
Multiplicity:

```
# Set m to Multiplicity of v in H.  
m = H[v]
```



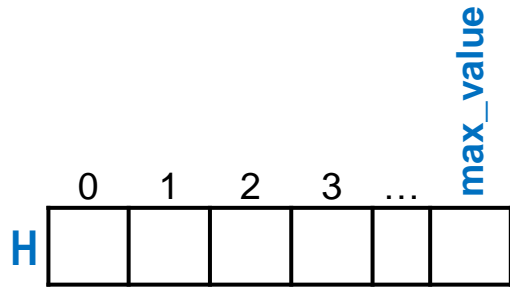
Enumeration:

```
# Enumerate elements of H.  
for k in range(0, max_value + 1):  
    for j in range(0, H[k]):  
        #.Enumerate k.
```


**Performance:**

Operation	Steps
add	constant
remove	constant
membership	constant
multiplicity	constant
enumeration	linear in <code>max_value</code> + number of elements in the multiset

Limitation. Enumeration of small multisets of values in a large range is not efficient.



Other Limitations:

- Integer items. Elements of the multiset must be integers. In contrast, lists can store any type of value, and Sequential Search can be used to find values of any type in a list, provided an equality operation is provided for that type.
- Limited range. The integer elements of the multiset must lie in a limited range for which there is enough memory for the histogram, $H[0..max_value]$.
- Associated values. The histogram representation does not provide an obvious way to represent the associated value components of $\langle key, value \rangle$ pairs. In contrast, to represent a multiset of $\langle key, value \rangle$ pairs in the list representation, not just integer keys, one can store the keys in one array, say, $A[0..n-1]$, and the values in a parallel array, say, $B[0..n-1]$. Alternatively, array A can contain references to $\langle key, value \rangle$ -pair objects, and the implementation of the multiset operations can be adapted to inspect the key fields of those objects.

Ramanujan Cubes, continued: An application of histograms

```
# Confirm Ramanujan's claim that 1729 is the smallest number that is the
# sum of two positive cubes in two different ways.
# =====
# Record the values of  $r^3+c^3$  that arise for all sets  $\{r,c\}$  of
# distinct nonnegative integers that are no larger than 12.
for r in range(1, 13):
    for c in range(0, r):
        #.Keep track of having seen  $r^3 + c^3$ .

#.Confirm that 1729 is the smallest integer that arose twice.
```

Ramanujan Cubes, continued: An application of histograms, manual review of output.

```
# Create histogram
N = 12**3 + 11**3 + 1      # (Max r)**3+(max c)**3+1, for r!=c in [0..12].
H: list[int] = [0] * N    # H[k] = # of {r,c}, r!=c, s.t. k=r**3+c**3.

# Confirm Ramanujan's claim that 1729 is the smallest number that is the
#   sum of two positive cubes in two different ways.
# =====
#.Let H be a histogram of  $r^3 + c^3$ , for each set  $\{r,c\}$  of distinct
#   nonnegative integers that are no larger than 12.
#.Output non-zero bins of histogram H.
```

Ramanujan Cubes, continued: An application of histograms, **manual review of output.**

```
# Create histogram
N = 12**3 + 11**3 + 1      # (Max r)**3+(max c)**3+1, for r!=c in [0..12].
H: list[int] = [0] * N    # H[k] = # of {r,c}, r!=c, s.t. k=r**3+c**3.

# Confirm Ramanujan's claim that 1729 is the smallest number that is the
#   sum of two positive cubes in two different ways.
# =====
# Let H be a histogram of r**3 + c**3, for each set {r,c} of distinct
#   nonnegative integers that are no larger than 12.
for r in range(1, 13):
    for c in range(0, r):
        H[r**3 + c**3] += 1

# Output non-zero bins of histogram H.
for k in range(0, N):
    if H[k] > 0: print(k, H[k])
```

A more automated solution

Ramanujan Cubes, continued: An application of histograms, **automated confirmation**.

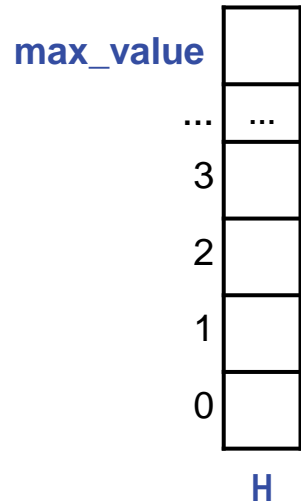
```
# Create histogram
N = 12**3 + 11**3 + 1      # (Max r)**3+(max c)**3+1, for r!=c in [0..12].
H: list[int] = [0] * N    # H[k] = # of {r,c}, r!=c, s.t. k=r**3+c**3.

# Confirm Ramanujan's claim that 1729 is the smallest number that is the
#   sum of two positive cubes in two different ways.
# =====
# Let H be a histogram of r**3 + c**3, for each set {r,c} of distinct
#   nonnegative integers that are no larger than 12.
for r in range(1, 13):
    for c in range(0, r):
        H[r**3 + c**3] += 1

# Let k be smallest index s.t. H[k] > 1.
k=0; while H[k] < 2: k += 1

if (H[k] == 2) and (k == 1729): print("confirmed")
else: print("not confirmed")
```

Hash Tables combine the good aspects of lists and histograms.

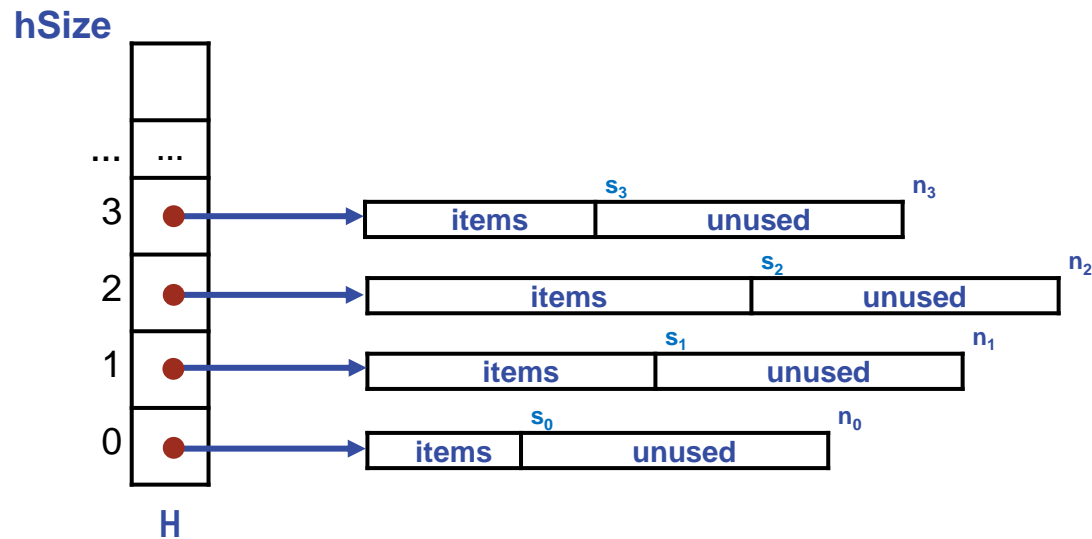


Start with a histogram $H[0..max_value]$.

*C/C++

This section is not valid for C-style arrays in C/C++. Rather, it can be read as describing one of the alternatives to C-style arrays that are available in C++.

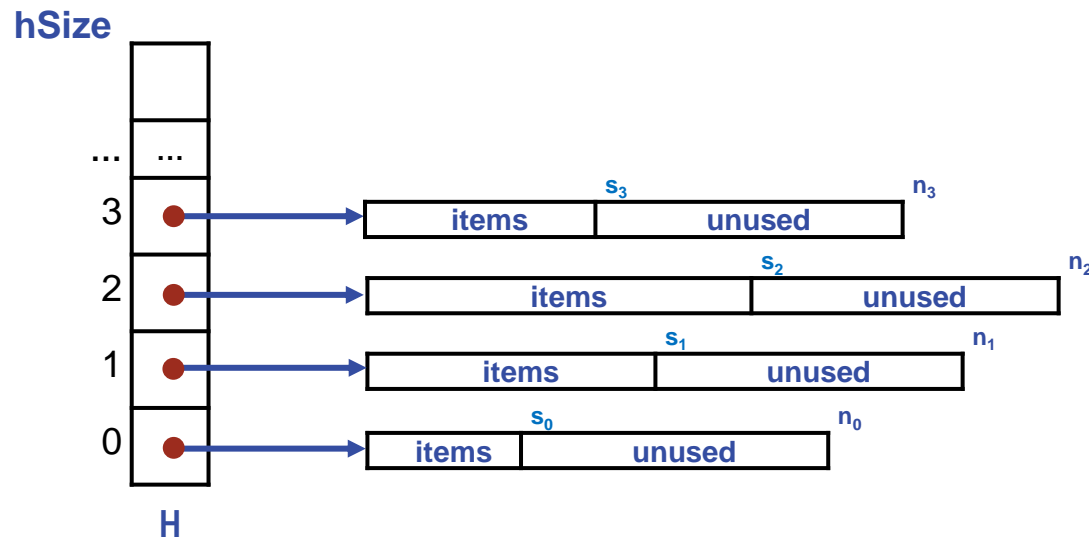
Hash Tables combine the good aspects of lists and histograms.



To implement a collection of items that have keys of an arbitrary type t :

- Introduce function $\text{hash}: t \rightarrow 0..2^{31}-1$ that maps type- t keys into uniformly-distributed integers in $0..2^{31}-1$.
- Replace the histogram multiplicities in $H[0..hSize]$ with references to sub-collections of items.
- All items that hash to k are stored in sub-collection $H[k \bmod hSize]$.
- Dynamically adjust $hSize$, as needed, to keep H and sub-collections not too big and not too small.

Hash Tables combine the good aspects of lists and histograms.



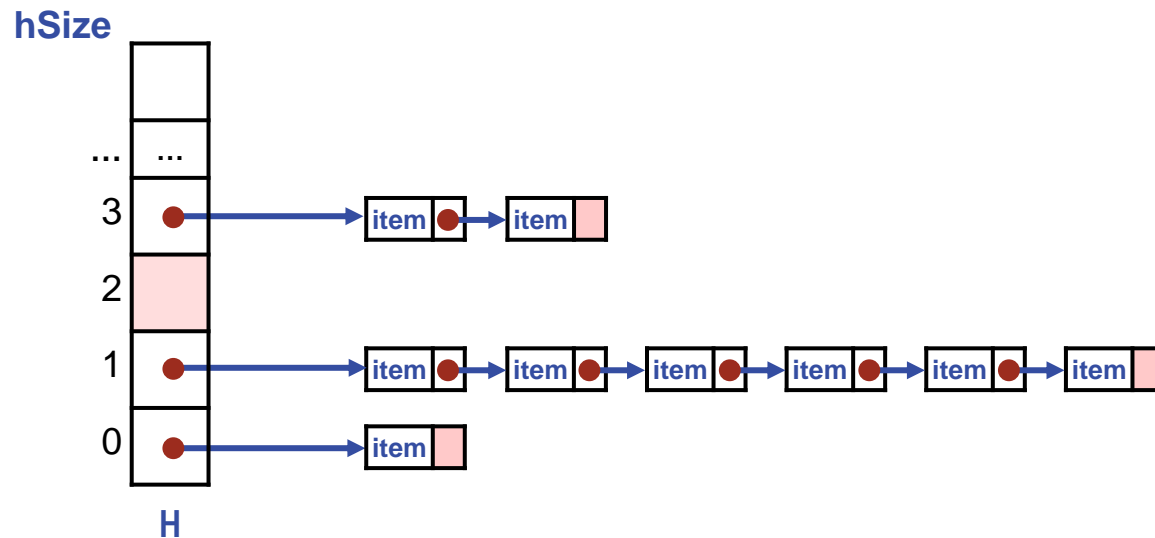
When the total number of items exceeds some given threshold, halve sub-collection Sequential Search times by:

- Doubling `hSize` and reallocating items to appropriate new half-length sub-collection.

To implement a collection of items that have keys of an arbitrary type t :

- Introduce function `hash: t → 0..231-1` that maps type- t keys into uniformly-distributed integers in `0..231-1`.
- Replace the histogram multiplicities in `H[0..hSize]` with references to sub-collections of items.
- All items that hash to k are stored in sub-collection `H[k mod hSize]`.
- Dynamically adjust `hSize`, as needed, to keep `H` and sub-collections not too big and not too small.

Hash Tables combine the good aspects of lists and histograms.



When the total number of items exceeds some given threshold, halve sub-collection Sequential Search times by:

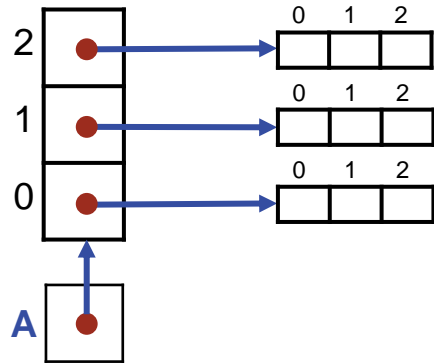
- Doubling $hSize$ and reallocating items to appropriate new half-length sub-collection.
- In practice, the sub-collections are often implemented as linked item holders, a data structure known as a *linked list*.

To implement a collection of items that have keys of an arbitrary type t :

- Introduce function $\text{hash}: t \rightarrow 0..2^{31}-1$ that maps type- t keys into uniformly-distributed integers ≥ 0 .
- Replace the histogram multiplicities in $H[0..hSize]$ with references to sub-collections of items.
- All items that hash to k are stored in sub-collection $H[k \bmod hSize]$.
- Dynamically adjust $hSize$, as needed, to keep H and sub-collections not too big and not too small.

Two-dimensional arrays are really one-dimensional arrays of one dimensional arrays.

```
A: list[list[int]] = [[0 for _ in range(3)] for _ in range(3)]
```



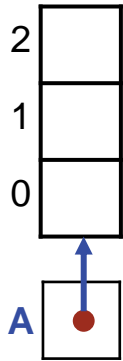
*C/C++

This section is not valid for C-style arrays in C/C++. Rather, it can be read as describing one of the alternatives to C-style arrays that are available in C++.

Create an array for the rows, but not the rows themselves.

Two-dimensional arrays are really one-dimensional arrays of one dimensional arrays.

```
# Create a 3-by-3 triangular array.  
A: list[list[int]] = [None] * 3
```

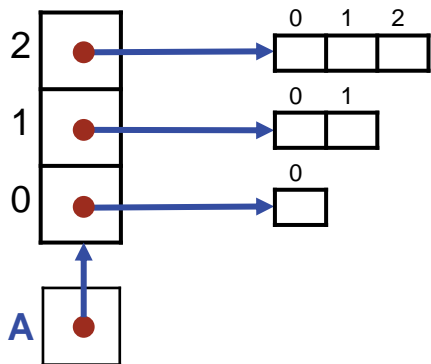


There is no requirement that “row” arrays have equal length.

Create an array for the rows, but not the rows themselves.

Two-dimensional arrays are really one-dimensional arrays of one dimensional arrays.

```
# Create a 3-by-3 triangular array.  
A: list[list[int]] = [None] * 3  
for r in range(0,3): A[r] = [0] * (r + 1)
```



Create the rows themselves, and refer to them in A.

There is no requirement that “row” arrays have equal length.

N.B. There are two ways to initialize a 1-D array:

```
A: list[int] = [0] * N
```

vs

```
A: list[int] = [0 for _ in range(N)]
```

What's the difference?

N.B. There are two ways to initialize a 1-D array:

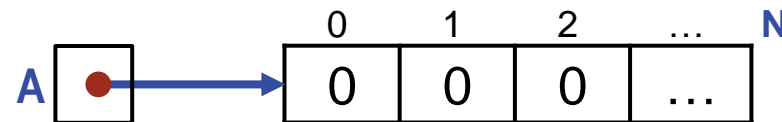
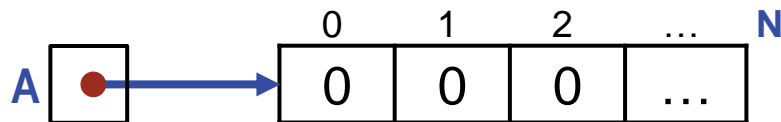
```
A: list[int] = [0] * N
```

vs

```
A: list[int] = [0 for _ in range(N)]
```

What's the difference?

For arrays whose elements are simple-type values like **int**, **float**, or **bool**, there is no difference.



N.B. There are two ways to initialize a 1-D array:

```
A: list[int] = [0] * N           vs           A: list[int] = [0 for _ in range(N)]
```

What's the difference?

For arrays whose elements are simple-type values like **int**, **float**, or **bool**, there is no difference.

But for arrays whose elements are themselves references to objects, there can be a big difference:

N.B. There are two ways to initialize a 1-D array:

A: `list[int] = [0] * N`

vs

A: `list[int] = [0 for _ in range(N)]`

What's the difference?

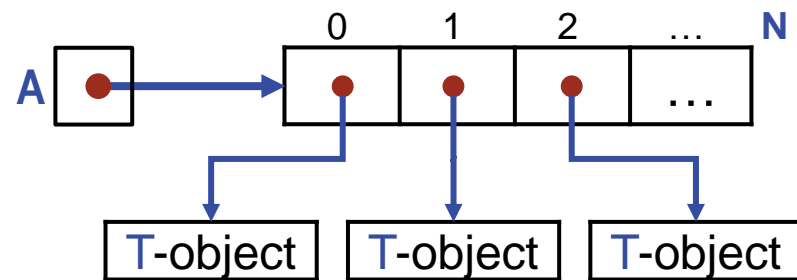
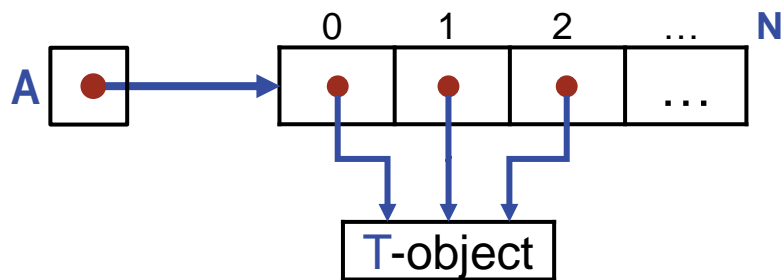
For arrays whose elements are simple-type values like **int**, **float**, or **bool**, there is no difference.

But for arrays whose elements are themselves references to objects, there can be a big difference:

A: `list[T] = [create-T-object] * N`

vs

A: `list[T] = [create-T-object for _ in range(N)]`



Use only the second way to initialize such an array unless you definitely want the sharing shown.